Preparacion-examen-2021.pdf

PruebaAlien

Informática Gráfica

3º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

¿Qué es la transformación de vista?

La transformación de vista es una conversión que permite cambiar de sistema de coordenadas, de sistemas de coordenadas de mundo a sistema de coordenadas vista. Esta conversión permite simular el posicionamiento de la cámara en cualquier orientación y posición. El sistema de coordenadas de vista se alinea con el sistema de coordenadas de mundo aplicando transformaciones geométricas (1 traslación y 3 rotaciones).

¿Qué parámetros de la cámara están implicados?

Los parámetros de la cámara que están implicados son 3:

- **VRP: La posición de la cámara, en el origen del sistema de coordenadas de vista. Es decir, es un punto en el sistema de coordenadas de mundo.**
- **VPN: Donde mira la cámara, es decir es el eje Z del sistema de coordenadas de vista, dicho de otro modo es el vector dado en el sistemas de coordenadas de mundo.**
- **VUP: La orientación de la cámara hacia arriba, es decir un vector dado en el sistema de coordenadas de mundo.**

¿Qué matriz de OpenGL almacena dicha transformación?

GL_MODELVIEW es la que guarda la matriz de transformaciones en OpenGL

Cree un ejemplo incluyendo las llamadas de OpenGL.

```
glMatrixMode(GL_MODELVIEW);

glPushMatrix();

glTranslatef(0.5,0.5,0);

glRotated(90,0,0,1);

glScalef(0.5,1,2);

objeto.draw(modo);

glPopMatrix();
```

Enumere y explique las propiedades de la transformación de perspectiva

1. **Acortamiento perspectivo: Los objetos que están más lejos producen una proyección de ellos más reducida de su tamaño.**
2. **Puntos de fuga: Cualquier conjunto par de líneas paralelas convergen en un mismo punto, llamado punto de fuga.**
3. **Inversión de vista: Los puntos que se encuentran detrás del centro de proyección se proyectan invertidos.**
4. **Distorsión topológica: cualquier objeto que se encuentre un extremo delante y otro detrás del centro de proyección produce 2 proyecciones, uno desde una posición positiva hasta el mas infinito y otra desde una posición negativa hasta el menos infinito, siendo un objeto irreal.**

Queremos realizar acercarnos a un objeto para ver sus detalles. Explicar cómo se podría hacer en una proyección de perspectiva, ventajas e inconvenientes.

Una posible solución sencilla es acercar el objeto, pero uno de los problemas es que si no se cambia el plano delantero, llegara a un punto en el que corta el objeto y es como si estuviéramos dentro del objeto. De tal forma que si colocásemos la cámara en una cierta posición donde los planos de corte no recorten el objeto, se podría hacer zoom, cambiando el tamaño de la ventana de proyección.

Dado un cubo definido por sus vértices, vector<_vertex3f> Vertices, y triángulos, vector<_vertex3ui> Triangles, indique lo siguiente si queremos mostrar el cubo texturado:

1) La estructura de datos para guardar las coordenadas de textura

Seria un vector flotante, que contiene 2 valores una la x y otra la y, y su tamaño sería el número de vértices que tenga el objeto.

```
Vector<_vertex2d> Vertices_textura;
```

2) La función que dibujaría el objeto texturado

```
glBegin(GL_TRIANGLES);

for(unsigned int i=0; i<Triangles.size(); ++i){

        glTexCoord2fv((GLfloat *) &Vertices_textura[Triangles[i]._0]);

        glVertex3fv((GLfloat *) & Vertices_textura[Triangles[i]._0]);

        glTexCoord2fv((GLfloat *) &Vertices_textura[Triangles[i]._1]);

        glVertex3fv((GLfloat *) & Vertices_textura[Triangles[i]._1]);

        glTexCoord2fv((GLfloat *) &Vertices_textura[Triangles[i]._2]);

        glVertex3fv((GLfloat *) & Vertices_textura[Triangles[i]._2]);

}

glEnd();
```

3) Un ejemplo de coordenadas de textura para cada vértice, si la textura se aplica a todas las caras sin repetirla (esto es, cada cuadrado NO muestra la textura completa)

Supongo que no tengo en cuenta el problema de los puntos repetidos. Entonces solo tendre que asignar las coordenadas de textura sobre el cubo. Por ejemplo:

```
Vértices_textura[0]=_vertex2f(0,0.5);

Vértices_textura[1]=_vertex2f(0.25,0.5);

Vértices_textura[2]=_vertex2f(0,0.75);

Vértices_textura[3]=_vertex2f(0.25,0.75);

Vértices_textura[4]=_vertex2f(0.75,0.5);

Vértices_textura[5]=_vertex2f(0.75,0.5);
```

KEEP CALM AND ESTUDIA UN POQUITO

WUOLAH + #QuédateEnCasa

OpenGL internamente no usa 3 sino 4 coordenadas, nombrándose la última como w. Esto se debe a que OpenGL realiza los cálculos en coordenadas homogéneas.

Los fragment shaders trabajan con los fragmentos.

Diferencia entre fragment (fragment shader) y un pixel (vertex):

Lo que hay que tener en cuenta es que cualquier escena donde se vaya a producir, se representa como una imagen, ya que cada pixel es un cuadrado que tiene solo las coordenadas (x, y) y un color ya sea con RGB o RGBA, siendo A (el canal alfa) que nos indica el valor de transparencia.

Por ejemplo un triangulo se transformara en un conjunto de fragmentos relacionadas uno a uno con un pixel.

Pero la diferencia semántica entre fragmento y pixel en OpenGL es que un pixel es un cuadrado que contiene un color y ocupa una posición (x, y) determinada, mientras que un fragmento es el mismo cuadrado, con un color, pero esta en la etapa de ser procesado.

Por ejemplo, puede que algunos fragmentos se eliminen, por lo que no se producirá el pixel de ese.

La función **initializeGL** se usa para inicializar OpenGL.

La función **resizeGL** se usa cuando la ventana cambia de tamaño y cuando carga la ventana, para conocer el tamaño inicial de la ventana.

La función **paintGL** se encarga de dibujar cada vez que lo necesite. Con lo cual es el que nos permite dibujar usando los shaders.

Sistemas de coordenadas:

- **Sistema de coordenadas del modelo:** Define el objeto en aquellas unidades que le sean más propias, siendo un sistema de coordenadas 3D que utiliza flotantes. Normalmente se encuentran los objetos en el centro de coordenadas.
- **Sistema de coordenadas del mundo:** Todos los objetos del escenario, también incluidas las fuentes de luz y la cámara, se definen con este sistema de coordenadas. Este define las unidades más apropiadas para la escena, siendo un sistema de coordenadas 3D que utiliza flotantes.
- **Sistema de coordenadas de la cámara:** Aquí ya cambia, ya que hay que transformar de 3 dimensiones a 2 dimensiones, para ello los objetos tienen que definirse con respecto a este sistema de coordenadas.
- **Sistema de coordenadas de dispositivo normalizado:** El volumen de visión se transforma a este sistema de coordenadas, que facilita ciertas operaciones.
- **Sistema de coordenadas de dispositivo:** Cuando se realiza la proyección, convierte esos objetos en pixeles, que se dibujan a través de este sistema de coordenadas, dependiendo de las dimensiones del **viewport**. Siendo un sistema de coordenadas 2D de enteros.

glBegin(GL_POINTS); indicamos el tipo de promitiva (punto, línea y relleno)

Función **update** se usa para indicar a Qt que cuando pueda (atiende a numerosos eventos) actualice el contenido de la imagen, es decir llamar a la función paintGL.

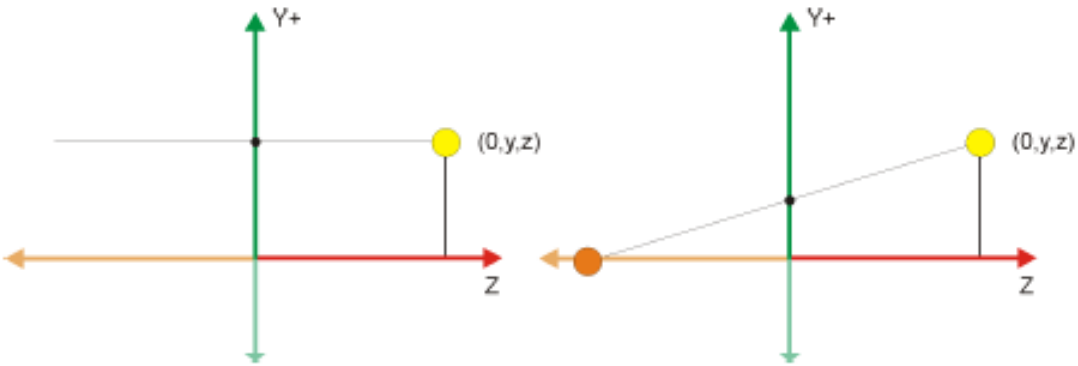
Proyecciónes

La idea es mostrar objetos que se definen en 3 dimensiones, en dispositivos que solo permite mostrarlo de forma bidimensional.

Con lo cual necesitamos la **proyección**, que se encarga de transformar objetos en 3 dimensiones a 2 dimensiones.

Hay 2 tipos de proyección:

- **Y la proyección perspectiva:** Es la que muestra de forma más natural los objetos, como el funcionamiento del ojo humano. Una de sus principales características es el acortamiento perspectivo, es decir los objetos que se ven a lo lejos se ven mas pequeños que los objetos que están más cerca. En este caso los proyectores son líneas rectas que unen cada vertice con el centro de proyección (CP), dicho de otro modo todos los vértices convergen en el centro de proyección (CP) en un punto finito. Dado un plano de proyección, la proyección es la intersección de los proyectores con el plano de proyección. **(frustum: necesita los valores de la ventana, la distancia con respecto al origen del plano delantero y la distancias con respecto al origen del plano trasero**
  glFrustum(X_MIN,X_MAX,Y_MIN*relacion_asp,Y_MAX*relacion_asp,FRONT_PLANE_PERSPECTIVE,BACK_PLANE_PERSPECTIVE);)
- **La proyección paralela:** Mientras que en la paralela por muy lejos que este el objeto siempre se ven del mismo tamaño, ya que el centro de proyección convergen en el infinito. **(ortho:**
  glOrtho(X_MIN*mult,X_MAX*mult,Y_MIN*mult*relacion_asp,Y_MAX*mult*relacion_asp,FRONT_PLANE_PERSPECTIVE,BACK_PLANE_PERSPECTIVE);)



Proyección paralela **Proyección perspectiva**

En la realidad la **proyección paralela** no existe, pero se usa para el dibujo técnico y otro tipo de reproducciones en los que no queremos que se cambien las medidas con respecto a la distancia.

Los planos de corte, como su propio nombre indica, se encargan de recortar el objeto que está por delante del panel delantero y/o detrás del panel trasero, esto nos evita uno de los

principales problemas, la distorsión topológica y evitar cargar un objeto complejo en unos pocos pixeles cuando el objeto este demasiado lejos.

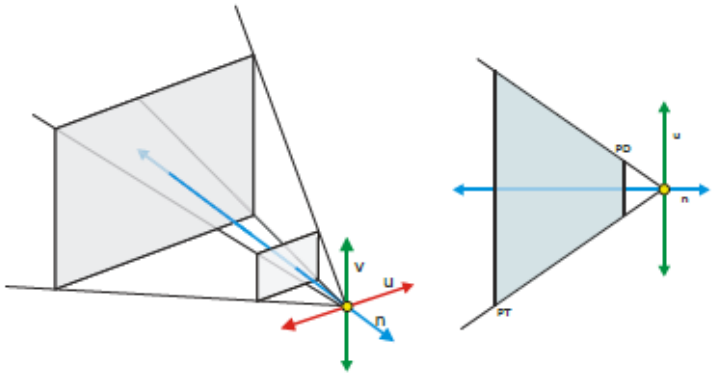


Figura 6.2: Frustum

En OpenGL hay que definir el **volumen de visión**, es decir todo el espacio tridimensional que podemos ver. Dicho de otro modo el volumen que podemos ver. En OpenGL se representa con una ventana rectangular, a parte hay que indicar el panel frontal y panel trasero. Para definir la ventana necesitamos 4 valores, es decir las coordenadas (x,y) de las esquinas inferior izquierdo y superior derecho del panel delantero. Y los paneles delantero y trasero son 2 valores más (Z), en total 6 valores para definir el volumen de visión, esto sirve tanto para la proyección perspectiva como la paralela.

Transformaciones

Las transformaciones de los objetos en OpenGL se representan mediante matrices de 4x4, representando las transformaciones a realizar. La transformación es la multiplicación de la matriz por el vector que representan la posición de cada vertice del objeto.

Para mover un objeto de una posición a otra se llama **traslación**. Siendo p0 el punto original y T el valor a trasladar, la posición donde se moverá el punto sería: $\boldsymbol{p0' = p0 + T}$**.** Con lo cual:

$$x' = x + T_x$$
$$y' = y + T_y$$
$$z' = z + T_z$$

Para cambiar el tamaño del objeto se le llama **escalado**. Para el escalado, siendo p0 el punto original y S el factor de escala, la nueva posición del punto es: $\boldsymbol{p0' = p0 * S}$**.** Con lo cual:

$$x' = x * S_x$$
$$y' = y * S_y$$
$$z' = z * S_z$$

Con lo cual para agrandar el objeto el factor de escala tiene que ser > 1.

Si queremos que encoja el objeto el factor de escala tiene que ser 0 > S < 1

Si el factor de escala es 0, el objeto se destruye poniendo todos los puntos del objeto en el (0,0,0)

Si los factores de escalas son negativos, el objeto se invertirá.

Y si los factores son iguales se forma un escalado homogeneo y si no son iguales forman un escalado heterogeneo.

Esto es asi, ya que el escalado se realiza con respecto al origen de coordenadas.

Para cambiar la orientación del objeto se le llama **rotación**. Como estamos en 3 dimensiones podemos rotar con respecto al eje **x**, **y** y **z**. De tal forma para rotar se aplica los siguientes cálculos:

Eje X

$$x' = x$$

$$y' = y * cos(\alpha) - z * sin(\alpha)$$

$$z' = y * sin(\alpha) + z * \cos(\alpha)$$

Eje Y

$$x' = x * cos(\alpha) + z * sin(\alpha)$$

$$y' = y$$

$$z' = -x * sin(\alpha) + z * \cos(\alpha)$$

Eje Z

$$x' = x * cos(\alpha) - y * \sin(\alpha)$$

$$y' = x * sin(\alpha) + y * \cos(\alpha)$$

$$z' = z$$

Con lo cual, hay que tener claro que sobre el eje al que queremos rotar no cambia. En caso de que queramos rotar con respecto otro pivote que no sean (x, y o z), tendrá que convertirlo en el origen.

Pero aplicar las transformaciones sueltas a un objeto es costoso, ya que tiene que realizar cada una de las transformaciones a cada punto. Con lo cual lo que se hace es una matriz 4x4, donde contendrá todas las transformaciónes y a la que se aplicara a cada punto. Esto nos solventa varios problemas, como reducción de operaciones de más, la matriz no depende de las coordenadas del punto y es mucho más eficiente. Antes de explicar la matriz 4x4 hay que tener en cuenta que para realizar los cambios se hace en un orden especifico con respecto al centro de coordenadas, ya que si lo hacemos en otro orden el resultado no es el mismo.

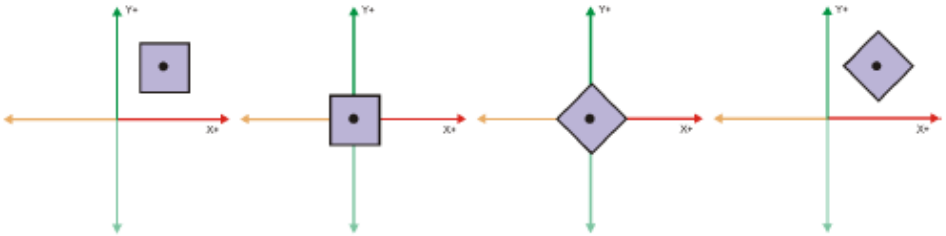
1. Escalado o rotación
2. La traslación

Al usar una matriz 4x4 estamos pasando de 3D a 4D.

¿Cómo pasamos un punto en coordenadas 3D a 4D?

Para ello usamos la siguiente formula: $x' = x * w,\ y' = y * w,\ z' = z * w$, siendo $w \neq 0$, con lo cual, las coordenadas son (x*w, y*w, z*w, w).

Y a la inversa (4D a 3D) →**(x/w, y/w, z/w, 1)**



distintas matrices de transformación para la traslación y el escalado:

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} E_x & 0 & 0 & 0 \\ 0 & E_y & 0 & 0 \\ 0 & 0 & E_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y las de las rotaciones:

X

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y

$$\begin{bmatrix} cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z

$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Con lo cual si queremos realizar multiples transformaciones es tan simple como multiplicar las matrices de cada transformación y la matriz resultante multiplicarla por cada vertice. Esto hace que las matrices 4x4 sea estándar en todas las bibliotecas graficas y GPUs, aunque haya que realizar más operaciones.

En vertex shader aparece 3 matrices que representan las transformaciones principales: **transformación de modelado, transformación de la cámara y transformación de proyección**. Esta separación es muy importante, ya que nos permite transformar un punto a cualquier sistema de coordenadas definidas por las transformaciones.

Con la **transformación de modelado** podemos pasar de coordenadas de modelado a coordenadas de mundo.

Con la **transformación de la cámara** podemos pasar de coordenadas de mundo a coordenadas de cámara.

Y con la **transformación de proyección** podemos pasar de coordenadas de cámara a coordenadas de dispositivo normalizado.

Con lo cual podemos pasar de un sistema de coordenadas a otra gracias a las transformaciones o aplicando la transformación inversa.

Tres dimensiones

Hay que tener en cuenta que la cámara virtual realiza 2 tareas diferentes:

1. El proceso de proyección en la que se le pasa la información tridimensional a bidimensional.
2. Y la posición de los objetos donde se mostraran a través de la cámara.

Componentes de una cámara:

- **Posición de la cámara:** Para realizarlo hay que hacer un cambio de sistema de coordenadas de mundo a vista, es decir transforma el sistema de coordenadas de mundo de los vértices del objeto, por el sistema de coordenadas de vista. Para hacer ese cambio hay que realizar transformaciones geométricas.
- **Dirección de la cámara**
- **Orientación de la cámara (vertical, apaisada u otro angulo)**
- **Y el zoom (el cierre o apertura de la lente)**

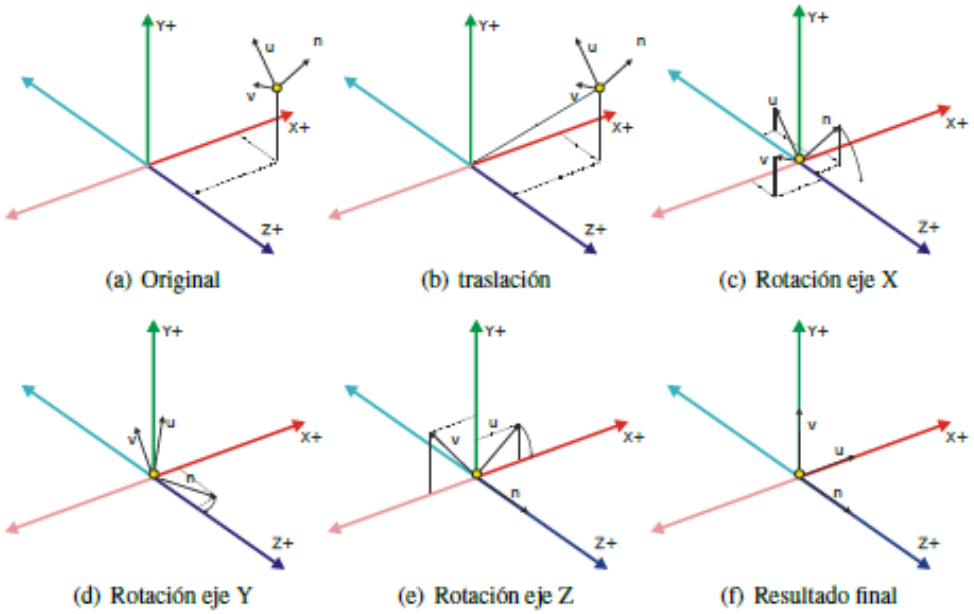


Figura 6.4: Proceso para obtener la transformación de vista.

La llamada transformación de vista TV en OpenGL es VIEW

En el mundo virtual tenemos que simular con todos los objetos representados como números y ecuaciones, para conseguir lo mismo o similar al mundo real.

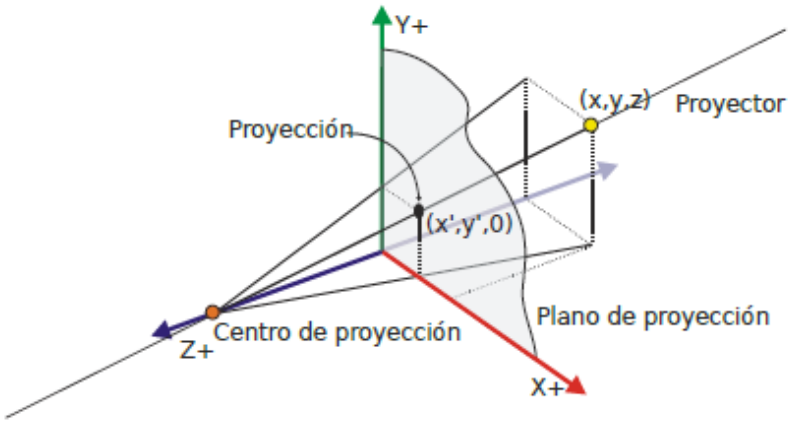


Figura 6.1: Proyección de perspectiva

Las transformaciones que representa la cámara es otra matriz 4x4. Para las transformaciones es muy importante el orden, por ejemplo si queremos aplicar una traslación y luego una rotación, entonces primero tiene que ir la rotación y luego la traslación, ya que se multiplica la matriz de transformaciones por cada vertice del objeto. Ya que en la vida real movemos la cámara, pero en el mundo virtual la cámara se queda fija y son los objetos los que se mueven con respecto a la camara, haciendo que parezca que es la cámara la que se mueve.

Pero esto hace que nos aparezca otro problema de la cámara.

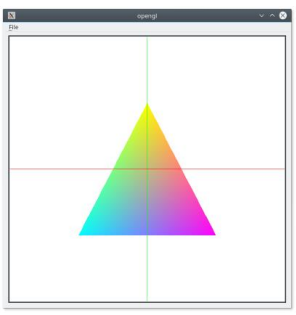
Triangulos

En OpenGL casi todo se construye con triángulos, esto se ha definido asi, puesto que es un polígono sencillo y por sus propiedades para su visualización. Entonces cualquier objeto que queramos representar y que tenga superficie, habrá que hacerlo a través de triángulos.

Para ello cada triangulo se crea mediante 3 vertices y habrá que indicarle en OpenGL que esos 3 vertices son un triangulo y no como vértices independientes o como una recta.

Interpolación de colores

Cada variable que pasamos del **vertex shader** al **fragment shader** se le aplica una interpolación.



La interpolación es necesaria, ya que se usa para la iluminación y para implementar otros procedimientos que necesiten valores intermedios, por los extremos.

En el dibujado de objetos no hay diferencia entre distintos objetos, ya que todos están compuestos por triángulos, salvo en el numero de triángulos.

Tenemos 3 funciones:

- **Model**: que se encarga de aplicar las transformaciones al modelo.
- **View**:
- **Projection**:

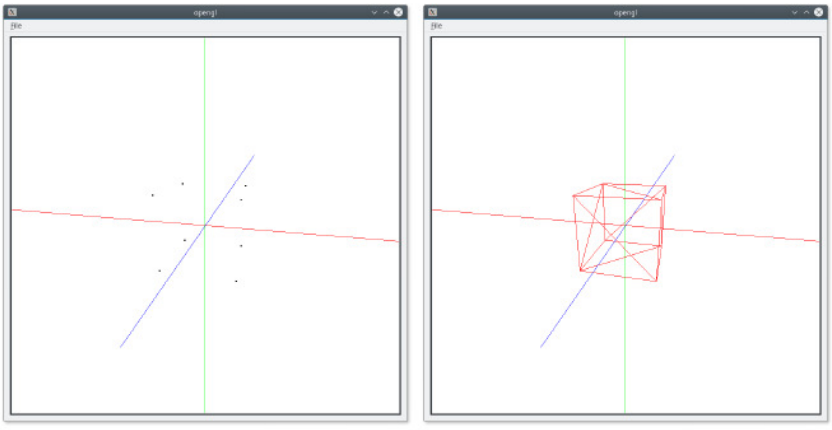
glPolygonMode(MODO_CARA,MODO_VISUALIZACIÓN);

MODO_CARA: indicamos que lados de la cara afecta, tenemos **GL_FRONT** dibuja la cara de alante**, GL_BACK** dibuja la cara de atras **y GL_FRONT_AND_BACK** dibuja ambas caras delantera y trasera.

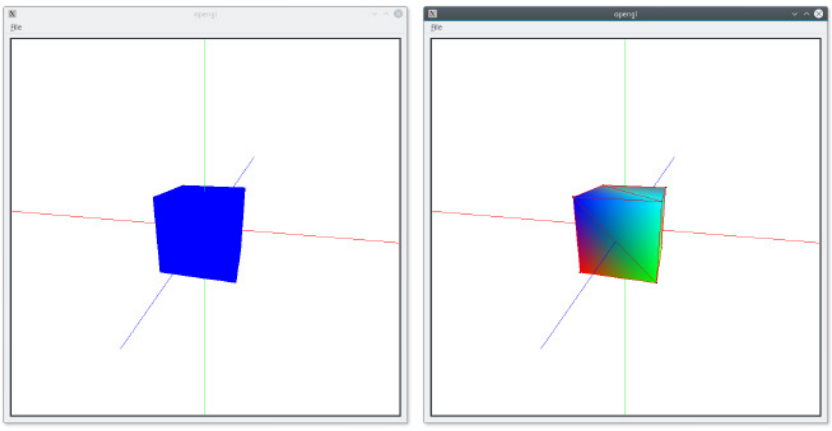
MODO_VISUALIZACIÓN: temenos **GL_POINT** que dibuja los vértices solo**, GL_LINE** dibuja las líneas de los triángulos que forman cada 3 vertices **y GL_FILL** rellena el hueco que une los 3 vertices que forman el triangulo**.**

Ejemplo: (rellenar los triangulos por las 2 caras)

glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);



(a) Puntos (b) Líneas



(c) Relleno (d) Todo

Revolución de objetos

Uno de los procedimientos es el **barrido por revolución**: dada una curva, podemos girar el objeto alrededor de un eje (x,y,z) un numero de veces y a partir de dichos perfiles obtener la superficie.

Los puntos de perfil generatriz giran con respecto a un eje y obteniendo una serie de perfiles a partir de los cuales obtener las caras. Para realizar el giro o las revoluciones se usan las funciones seno y coseno, ya que con eso podemos calcular las posiciones en una circunferencia

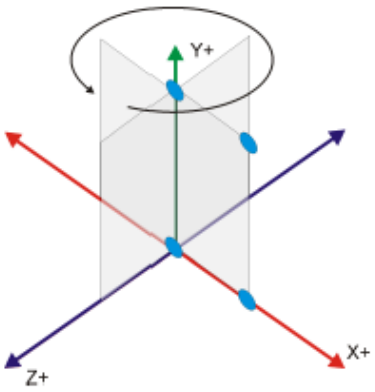


Figura 10.2: Perfil para obtener un cilindro

Figura 10.3: Despliegue de los puntos en un plano

$$x = R \cdot \cos(\alpha)$$
$$z = -R \cdot \sin(\alpha)$$
$$0 \leq \alpha \leq 2\pi$$

Una vez creado el perfil del objeto, se obtiene los triángulos que representa la superficie. Para hacerlo hay que olvidarse la forma del objeto y crear una forma más sencilla como si aplastásemos el objeto obteniendo un plano del objeto, manteniendo la posición relativa de los vertices. Para cerrar el objeto hay que repetir los puntos del principio al final, para unir los puntos y formar el objeto con los triángulos.

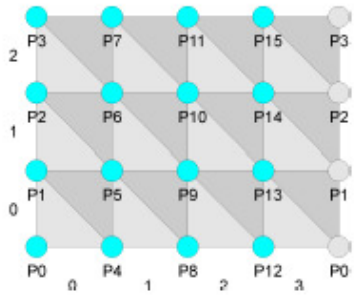


Figura 10.4: Posicionado lineal de los puntos

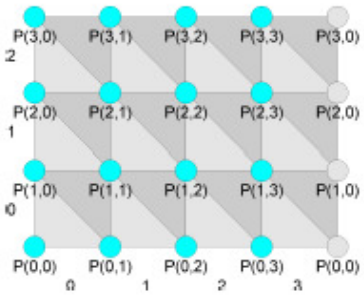


Figura 10.5: Posicionado 2D de los puntos

Para ver el patron de las caras dividimos las caras en 2 tipos de caras las pares y las impares, ejemplo la cara 2 par (fila 0, columna 1), esta compuesta por p5, p4 y p8 y la cara 2 impar (fila 0, columna 1), esta compuesta por p8, p9 y p5.

Si nos fijamos en las caras pares vemos que se forma el patron (punto1_anterior + 4, punto2_anterio +4 y punto3_anterior + 4). Ejemplo:

Triangulo 0: (p0, p4, p1)

Triangulo 2: (p4, p8, p5)

Triangulo 4: (p8, p12, p9)

El 4 viene del **numero de puntos que tiene el perfil**, en este caso como es un rectángulo tiene 4 puntos.

Con lo cual la formula para la generación de caras o triangulos pares e impares son:

- Cara par:
  $Cara_{(F \cdot ND+C) \cdot 2} = ((C \cdot NP+F)+1, (C \cdot NP+F), ((C+1) \cdot NP+F))$
- Cara impar:
  $Cara_{(F \cdot ND+C) \cdot 2+1} = (((C+1) \cdot NP+F), ((C+1) \cdot NP+(F+1)), (C \cdot NP+F)+1)$

Si lo queremos calcular cada cara como un vector de 2 componentes como este:

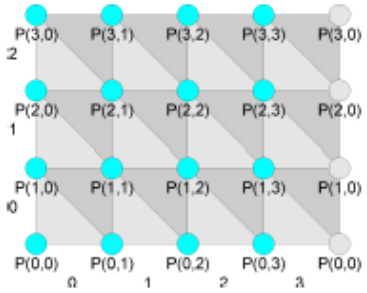


Figura 10.5: Posicionado 2D de los puntos , siendo p0 = p(0,0), p1 = p(0,1), …

Seria de la siguiente forma:

- Cara par:
  $Cara_{(F \cdot ND+C) \cdot 2} = (P(F+1,C), P(F,C), P(F,(C+1)\%ND))$
- Cara impar:
  $Cara_{(F \cdot ND+C) \cdot 2+1} = (P(F,(C+1)\%ND), P(F+1,(C+1)\%ND), P(F+1,C))$

Siendo F **Fila**, C **Columna**, NP **Numero de Puntos** y ND **Numero de Divisiones**.

Pero esto al ser sencillo, tiene sus inconvenientes, y es que genera puntos extremos duplicados y la creación de los triángulos que no tienen área, es decir triángulos degenerados.

Se pueden ver en este ejemplo en los triángulos rojos y los puntos grises:

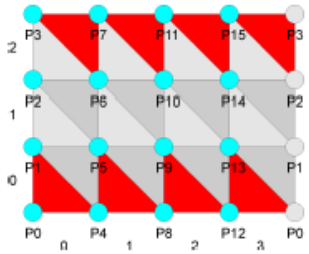


Figura 10.6: Triángulos degenerados

Aunque, podemos optimizarlo, eliminando esos puntos duplicados, para ello hay que distinguir 3 partes: tapa inferior, la zona central y la tapa superior. Con lo cual, tenemos que quitar los puntos de los extremos que se almacenan al final, a continuación generamos la parte central del objeto y por ultimo las tapas del objeto.

Modelado de objetos

Para el modelado de objetos necesitamos:

- Objetos
- Observador
- Y luces

En un **modelo** hay que tener una representación de los objetos tratados computacionalmente.

Un objeto real ocupa espacio, color, material y otros atributos.

Fisicamente formados por particulas que se agrupan en diferentes niveles, el nivel más básico son los quarks que componen los átomos.

Entonces, **¿Cuál es la representación tratable computacionalmente de un objeto?**

Al ser un computador, el punto se consideraría una partícula sin dimensión, que debe estar referenciado a un sistema de coordenadas.

Tipos de modelos:

- **Modelado de puntos:** Crea un objeto tridimensional con otra de 0 dimensiones. Su principal característica es que el punto se posiciona en un espacio tridimensional (x,y,z) en coordenadas cartesianas o polares. Ya que el punto se puede tratar computacionalmente con coordenadas. Dado un conjunto de vértices podemos obtener un objeto, pero esto nos lleva a un problema, y es como asociar los puntos a un objeto, la solución es utilizar los más representativos.
- **Modelado de alambres:** El modelado de puntos no nos permite ver con claridad que objeto es, excepto si hay una gran densidad de puntos, para ello la solución es mostrar la relaciones de los puntos, de tal forma que visualiza las aristas, mostrando la relación de los puntos.
- **Modelado de superficie:** Con el modelo de alambres se ve algo más claro el objeto, pero puede producir una solución ambigua y poco realista, entonces la solución es mostrar la superficie (modelo de fronteras).

Optimización de modelos

Los procedimientos para dibujar los puntos, aristas y caras no son muy eficientes, ya que generan una llamada por primitiva y no reutilizan la información. Esto se soluciona haciendo uso de primitivas especializadas:

- Tira de cuadriláteros.
- **Tira de triángulos**
- Abanico de triángulos

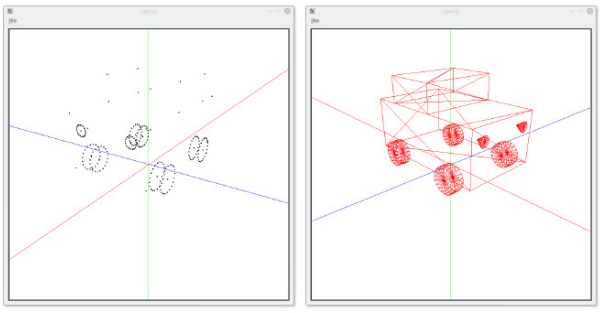
Modelado jerárquico sin movimiento

¿Qué ocurriría si se cambia la transformación de modelado y se vuelve a dibujar el cubo sin borrar la escena? **Que se obtendría un segundo cubo al compilar y ejecutar el programa, de tal forma que solo necesitamos crear un solo objeto cubo y podremos pintar tantas veces queramos ese objeto.**

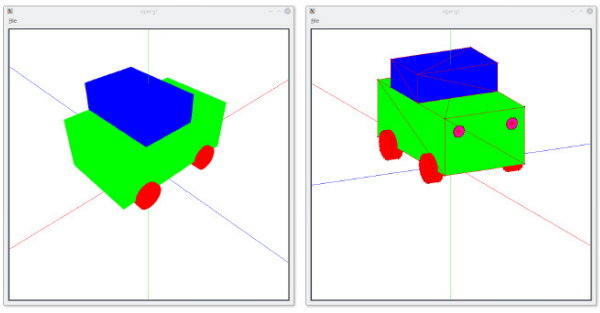
¿Y si añado una tercera transformación y se vuelve a dibujar el cubo? **Se añade un tercer cubo al ejecutar el programa.**

La jerarquización es básicamente establecer dependencias.

Por ejemplo: a partir de un cilindro se puede crear el objeto rueda y a partir de ese objeto rueda podemos obtener las 4 ruedas del coche. Con el cono podemos crear el objeto faro y a partir del objeto faro obtener 2 faros. Con el cubo podemos crear el objeto chasis y al juntar todos los objetos formamos el objeto coche y a partir de el objeto coche poder dibujar diversos coches. Es decir, cuando tengamos creado los objetos rueda, faro y chasis, creamos un objeto coche donde se añadira los objetos anteriores mencionados con sus respectivas transformaciones.



(a) Puntos (b) Líneas



(c) Relleno (d) Todo y proyección paralela

Ejemplo de un objeto: para la creación de la rueda primero aplicamos un escalado (1,0.4,1) y a continuación una rotación de 90 grados con respecto al eje X. Para hacer que la rueda se apoye sobre el suelo tendremos que hacer una traslación de (0.5,0,0).

Es importante saber que las transformaciones que se realicen en un nivel superior también llegan a los niveles inferiores. Otra cosa importante es que desde una función de un nivel es llamada por otra de un nivel inferior, al enviar las transformaciones, al volver al nivel inferior se puede recuperar las transformaciones que tuviera.

Es decir, si tenemos un nivel X que tiene la transformación T, que viene del nivel Y y es llamada desde una función de nivel inferior Z pasandole T' transformaciones más las transformaciones T del nivel X, esto es pasando T'' = T * T', con lo cual cuando vuelva a ejecutar el nivel Z, se deberá recuperar la transformación T, para que puedan ser usadas en otras llamadas a funciones de nivel inferior.

El mecanismo que nos permite hacer este paso y la recuperación es la **pila**. Que es la forma de operar en la versión antigua de OpenGL. Un detalle a tener en cuenta es que al crear la pila se mete la transformación identidad por defecto.

Modelo jerárquico con movimiento

Ejemplo brazo mecanico:

El brazo mecanico consta de una pinza, la mano, brazo1, brazo2 y la base.

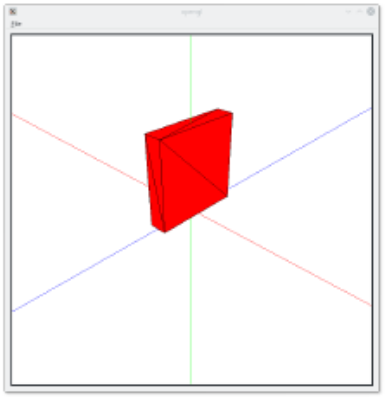


Figura 13.1: Pinza

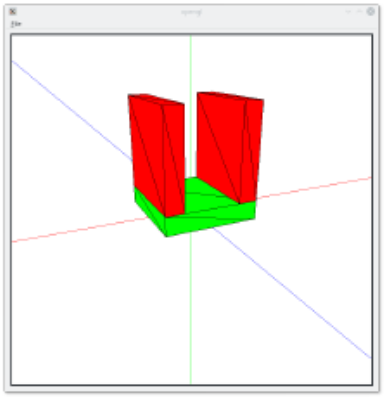


Figura 13.2: Mano

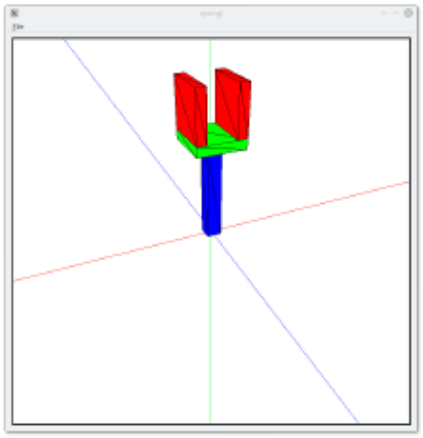


Figura 13.3: Brazo1

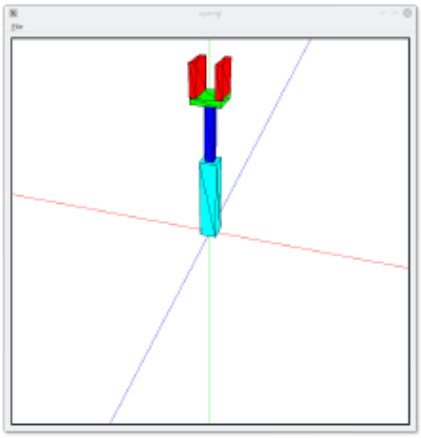


Figura 13.4: Brazo2

Entonces para formar el brazo mecánico la jerarquía de dependencia es la siguiente:

pinza ← mano ← brazo1 ← brazo2 ← base.

Para hacer los movimientos, en el caso de la mano, lo que vamos a hacer es que las pinzas se acerquen o se alejen aplicando una transformación sobre el eje X.

```
//pinza 1
glPushMatrix();
  glTranslatef(-traslacion,0.5,2.5);
  glTranslatef(0.4,0.2,0);
  pinza.draw();
glPopMatrix();
//pinza 2
glPushMatrix();
  glTranslatef(traslacion,0.5,2.5);
  glTranslatef(0.4,0.2,0);
  pinza.draw();
glPopMatrix();
```

Vamos a añadir otro movimiento al brazo1, haciendo que rote:

```
//rotación del brazo1
glPushMatrix();
  glTranslatef(0,2,0);
  glRotatef(angulo,0,1,0);
  brazo1.draw();
glPopMatrix();
```

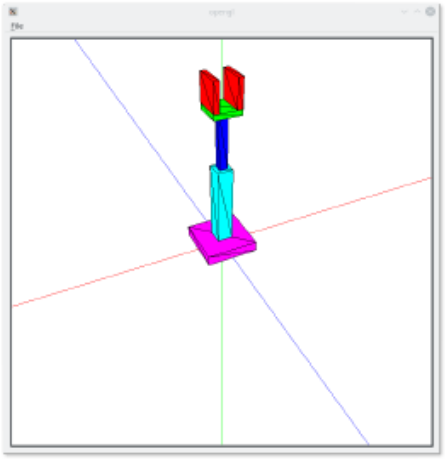


Figura 13.5: Base

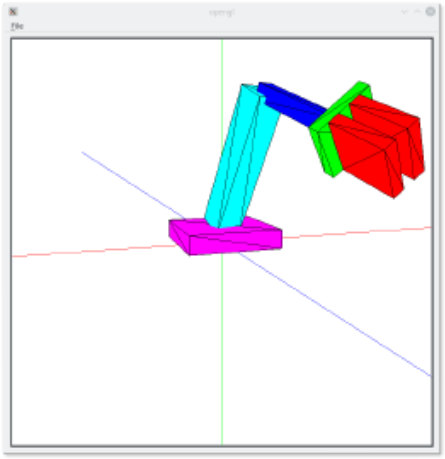


Figura 13.6: Ejemplo completo con movimiento

Iluminación

Para hacer que el objeto sea aun más real tenemos 3 aproximaciones al realismo que van de menos a mas:

- **Sombreado plano:** Se hace con los cálculos de sombreado en el vertex shader
- **Sombreado de Gouraud:** Se hace con los cálculos de sombreado en el vertex shader
- **Sombreado de Phong:** Se hacen con los cálculos de sombreado en el fragment shader, ya que no se puede hacer sobre un vector (vertex shader)

La iluminación necesita un modelo de reflexión, en OpenGL usamos 3 componentes sencillas: la ambiental, la difusa y la especular.

Difusa

La componente **difusa** consiste en entender que la orientación del objeto respecto a la fuente de luz modifica la cantidad de luz reflejada, es decir cuanto mas inclinado este la fuente de luz, los rayos de luz se reflejaran más.

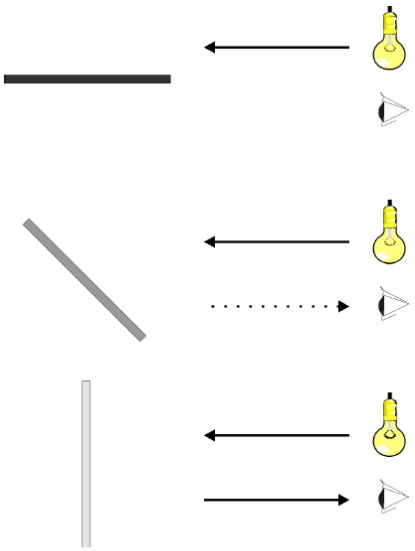


Figura 14.1: Reflexión difusa: cómo afecta la orientación

Puesto que estamos trabajando con superficies triangulares, esto implica que cuanto más perpendicular este el triangulo o triangulos, mayor será la cantidad de luz reflejada.

¿De que manera queda definido el plano? **Mediante la normal.**

Entonces para ello hay que calcular las normales del objeto, para trazar los rayos de luz. Pero existen 2 tipos de normales: **las normales de los triángulos** y **las normales de los vértices.**

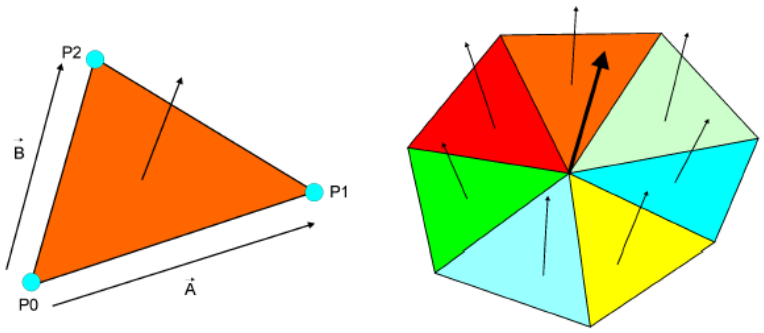
Para calcular las normales de los trinagulos, es sencillo, ya que, dado 3 puntos (P_0, P_1, P_2), podemos calcular los vectores $\vec{A} = P_1 - P_0$ y $\vec{B} = P_2 - P_0$, con lo cual si aplicamos el producto vectorial de los 2 vectores $\vec{N} = \vec{A} * \vec{B}$ obtenemos el vector normal del triangulo, cuyo sentido viene dado por la regla de la mano derecha.

Con esto, la normal de la cara no solo sirve para calcular la iluminación, sino también la orientación de la cara, ya que las normales de las caras apuntan al exterior del objeto.

Para calcular las normales de los vértices en un modelo de triángulos, se parte del calculo de las normales de los triángulos que confluyen en dicho vertice, es decir todos los triángulos que contengan ese vertice (triángulos vecinos) y se calcula el valor medio de dichas normales, dando la siguiente expresión:

$$\vec{N} = \frac{\sum_{i=1}^{n} \vec{N}_i}{n}$$

La normal que se obtiene es una aproximación. Otra cosa importante tanto en la normal de un triangulo como la de un vertice, es que deben estar normalizadas.



(a) Normal de un triángulo (b) Normal de un vértice

En el caso de la esfera, el calculo de las normales de los vértices son más sencillas, porque se obtienen de los propios vértices $\vec{N} = \vec{P} - \vec{C}$, siendo $\vec{C}$ el centro de la esfera.

Para calcular la **reflexión difusa**, dada la normal del triangulo, la reflexión tendera a cero, cuando sea perpendicular al vector de la dirección de la luz y será máximo cuando los 2 vectores sean paralelos. Dicho de otro modo si son perpendiculares los vectores (forman 90 grados), el valor de la **reflexión difusa tiende a 0**, y cuando son paralelos (forman 0 grados), el valor de la **reflexión difusa tiende al máximo**.

De tal forma que si le aplicamos: $\mathbf{cos(0) = 1}$ y $\mathbf{cos(90) = 0}$. Para calcular el coseno vamos a usar el producto escalar: $\vec{\boldsymbol{N}} \cdot \vec{\boldsymbol{L}} = |\vec{\boldsymbol{N}}| \cdot |\vec{\boldsymbol{L}}| \cdot \mathbf{cos}\ (\boldsymbol{\alpha})$. Como tenemos los vectores normalizados $\vec{\boldsymbol{N}} = \mathbf{1}$, $\vec{\boldsymbol{L}} = \mathbf{1}$ y por tanto la ecuación queda más simplificada: : $\vec{\boldsymbol{N}} \cdot \vec{\boldsymbol{L}} = \mathbf{cos}\ (\boldsymbol{\alpha})$.

Es importante saber que esta componente no se ve afectado frente a la posición del observador. Ya que cuando llega un rayo de luz el objeto refleja en todas partes.

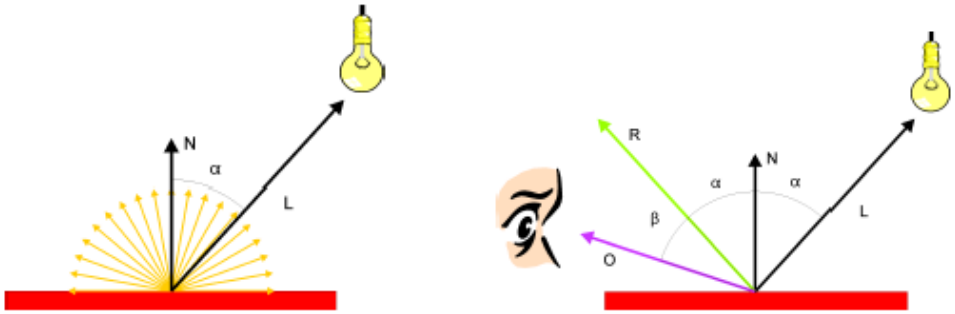
Especular

La **especular** actúa como un espejo, ya que cuando entra un rayo es reflejado, produciéndose un rayo de salida con el mismo angulo que el rayo de entrada. Con lo cual es importante la posición del observador, ya que dependiendo de donde se posicione podremos ver el rayo reflejado o no.

Normalmente, lo que ocurre es que los reflectores especulares no son ideales y se producen alguna dispersión alrededor del rayo reflejado.

Para modelar este comportamiento usamos el coseno, pero en este caso sobre el angulo beta que se forma entre $\vec{R}$ y $\vec{O}$, de tal forma que cuando el angulo es cero, obtiene la máxima reflexión y cuando el angulo es 90 grados, tiende a cero. Como podemos encontrar objetos mas o menos especulares, es necesario añadir un modificador para el coseno:

$\vec{\boldsymbol{R}} \cdot \vec{\boldsymbol{O}} = \mathbf{cos}^{\boldsymbol{n}}(\boldsymbol{\beta})$, siendo **n** un valor entre 0 e infinito.



(a) Reflexión difusa (b) Reflexión especular

Ambiental

La componente **ambiental** podemos modelar una especie de flujo de luz constante que viene de todas direcciones. Este flujo no es imaginario, ya que es un fundamento físico, es decir, es una componente que se forma a partir de varias reflexiones de los rayos de lux en los diferentes objetos. De tal forma que conforme los rayos se reflejan tomara la tonalidad del material del objeto donde se produce la reflexión.

En el modelo de iluminación global, todos los reflejos se tienen en cuenta.

En el modelo de iluminación local, esta componente se simplifica a través de un valor constante de baja intensidad. Este evita el efecto de que las caras que no reciban directamente la iluminación se vean de color negro.

Conclusión:

El modelo de reflexión de la luz: $\boldsymbol{I_r = ambiental + difusa + especular}$, la intensidad de la luz reflejada en el objeto es la suma de las 3 componentes.

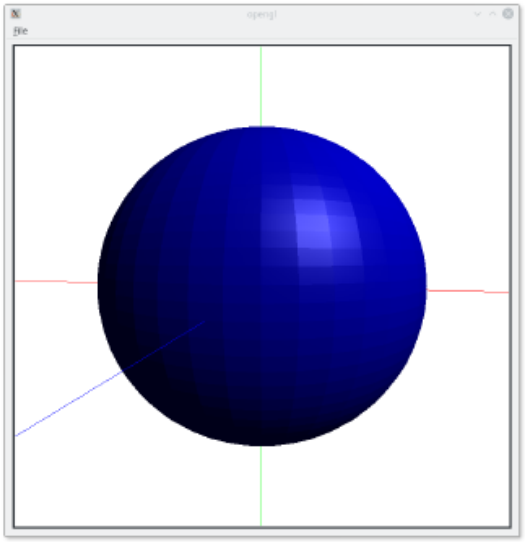
Entonces, si la intensidad de la fuente de luz es $\boldsymbol{I_l}$, la capacidad del material para reflejar la componente difusa es $\boldsymbol{K_d}$, la capacidad del material para reglejar la componente especilar es $\boldsymbol{K_e}$, y la reflexión ambiental se modela con $\boldsymbol{K_a}$, entonces el modelo de reglexion de la luz queda: $\boldsymbol{I_r = I_l \cdot K_a + I_l \cdot K_d \cdot cos(\alpha) + I_l \cdot K_e \cdot cos^n(\beta)}$

En OpenGL, la máxima intensidad de la luz es 1, es decir siendo $\boldsymbol{K_a = 0.25}$, $\boldsymbol{K_d = 0.8}$, $\boldsymbol{K_e = 0.5}$ el valor mínimo $\boldsymbol{I_r = 1 * 0.25}$, siendo los 2 angulos alfa y beta 90 grados, con lo cual el coseno de 90 grados es 0, y el valor máximo $\boldsymbol{I_r = 1 * 0.25 + 1 * 0.8 * 1 + 1 * 0.5 * 1 = 1.55}$, este valor se redondea a **1**, ya que en OpenGL solo admite hasta 1 como máximo, siendo los 2 angulos 0 grados, con lo cual el coseno de cero es 1.

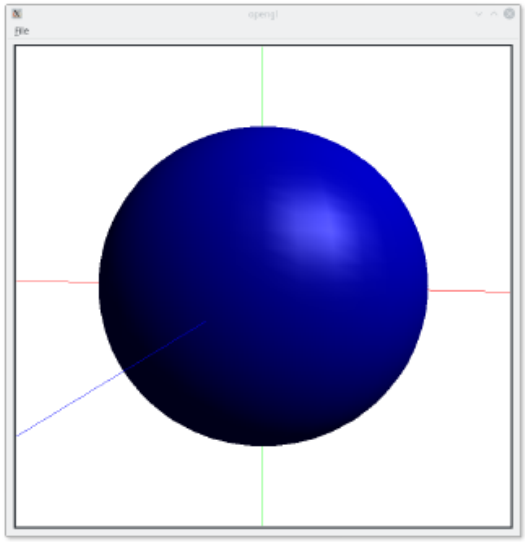
Resumen: las componentes **difusa** y **especular** dependen de la normal del objeto y la posición de la luz, y la componente **especular** depende de la posición del observador.

Los cálculos del modelo de iluminación se pueden hacer en cualquier sistema de coordenadas del modelo, de mundo o de cámara, aplicando en cada caso sus correspondientes ajustes.

Para obtener la posición de la cámara, usamos la inversa de la transformación de la cámara o transformación de vista.



(c) Color variable, iluminación y sombreado plano



(d) Color variable, iluminación y sombreado de Gouraud

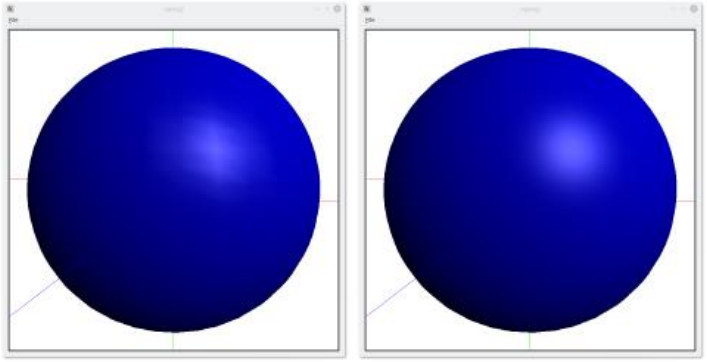
Iluminación: Phong

Diferencia entre **Phong** y **Gouraud**:

La diferencia es que **Gouraud** es una versión simplificada de **Phong**, en **Gouraud** se obtienen los valores de reflexión en los vértices, se calcula los colores correspondientes y los mismos se interpolan para obtener el resto de posiciones (fragmentos/pixeles), realizando la interpolación con el fragment shader. Pero con **Phong** se quiere obtener un resultado más exacto haciendo que se realicen los cálculos de reflexión en todos los puntos del objeto. El problema es que usando vertex shader conseguimos una definición continua de los objetos, líneas y triángulos, llegando a tener un numero infinito de posiciones. Pero los objetos continuos son discretizados en fragmentos (fragment shader) y siempre son finitos, con lo cual podemos calcular la reflexión para cada fragmento.

Entonces para poder usar los fragment shader debemos obtener la posición y la normal de cada fragmento, puesto que solo tenemos los vértices. Para obtenerlo usamos la interpolación, con lo cual debemos crear 2 variables de salida del vertex shader, que son la posición y la normal y ponerlas como variables de entrada en el fragment shader. Ya con eso OpenGL realiza la interpolación.

Hay que tener en cuenta que los cálculos de iluminación son en coordenadas de mundo, y otra cosa muy distinta es que los objetos tienen que convertirse en fragmentos, que finalmente se convierten en pixeles, en coordenadas de dispositivo.



(a) Sombreado Gouraud (b) Sombreado de Phong

Texturas

La textura, es como un envoltorio, es como si le pegásemos una foto en un objeto y se adaptase a la forma del objeto.

Pasos para aplicar la textura:

- Pasamos la imagen que esta en forma matricial, en un sistema de coordenadas normalizado, con las coordenadas **u** y **v**, siendo sus rangos entre 0 y 1 ambos incluidos ($0 \leq u \leq 1,\ \ 0 \leq v \leq 1$). Esto permite independizar el tamaño real de la imagen de su aplicación al modelo.
- Despues le asignamos a cada punto del modelo las coordenadas de textura correspondientes.

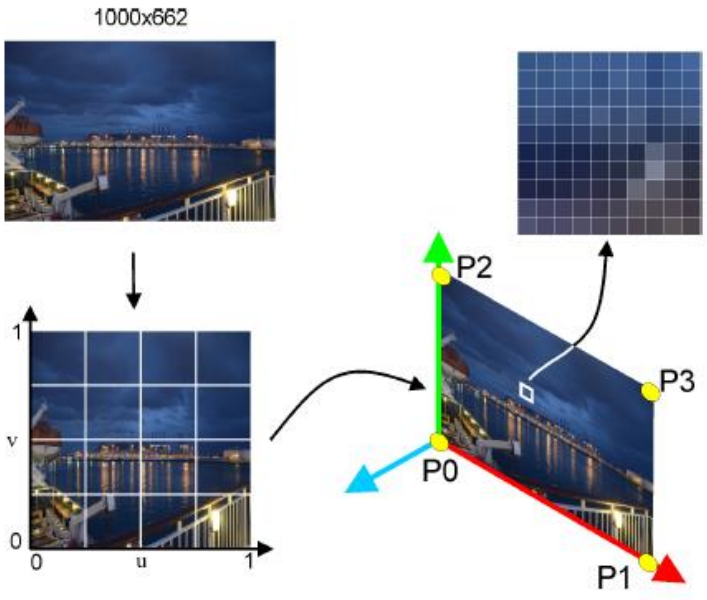
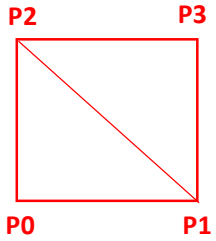


Figura 16.1: Pasos en la aplicación de una textura

Ejemplo: Supongamos que tenemos un cuadrado con 4 vertices y 2 triangulos, cuyos triángulos están formados por los puntos: (P2, P0, P1) y (P1, P3, P2), la asignación de coordenadas de textura sea la siguiente:

- (0, 0) → P0
- (1, 0) → P1
- (0, 1) → P2
- (1, 1) → P3



Si queremos mostrar la parte central de la imagen:

- (0.25, 0.25) → P0
- (0.75, 0.25) → P1
- (0.25, 0.75) → P2
- (0.75, 0.75) → P3

Con lo cual nos da a entender que solo se cambia las coordenadas de textura y no las coordenadas de los vértices.

El resto lo realiza OpenGL, para hacer el pegado de la textura sobre el objeto.

Uno de los factores importantes a tener en cuenta cuando se usa la textura, es su naturaleza matricial, frente a la naturaleza continua geométrica, ya que las imágenes tienen un tamaño finito.

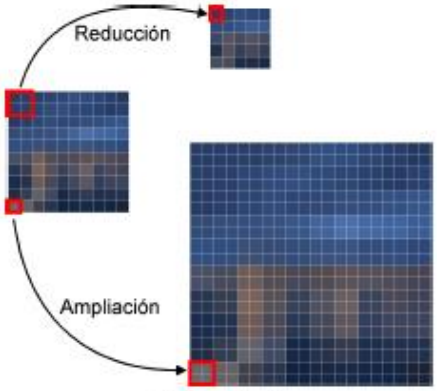
Ejemplo: Si tengo una textura aplicada a un objeto, dependiendo de la posición de la cámara y el tipo de proyección podemos encontrar 3 posibilidades:

- Una relación 1 a 1 entre los pixeles de la textura y los pixeles de salida, con lo cual no se hace nada.
- Hay más pixeles en la textura que pixeles de salida, en ese caso, el objeto esta lejos. OpenGL combina varios pixeles de la textura en 1 solo pixel de salida, esto hace que reduzca la cantidad de información.
- Hay menos pixeles en la textura que pixeles de salida, eso quiere decir que el objeto esta cerca. Con lo cual la imagen no tiene suficiente resolución, pero podemos mejorar un poco la salida, OpenGL intenta generar los pixeles que faltan mediante interpolación, pero esto es un problema, ya que genera nueva información. En algunos casos el resultado es exitoso en objetos con cambios suaves, pero en otros casos no.

Si entendemos como funciona la interpolación, el **mip-mapping** es similar, dado que nos podemos alejar el objeto, el mip-mapping lo que hace es reducir la textura.

Cuando vayamos a crear una textura para un objeto, siempre hay que tener una textura por defecto. Hay 2 formas: crear una imagen o crearla.

Ojo, nuestra geometría representa a un cuadrado, si la imagen a texturizar no es cuadrada se producirá una deformación.



(a) Interpolación

(b) Mip-mapping

¿Se puede asignarle una posición que sirva como referencia y poder almacenar un valor con las variables que representan a las texturas? Si, pero se usa el modificador **location**, sino el modificador **binding**.

Hay que tener en cuenta que los formatos de imágenes suelen usar un sistema de coordenadas izquiedo, con el origen en la esquina superior izquierda, mientras que en OpenGL usa el sistema de coordenadas derecho con el origen en la esquina inferior izquierda. Para ello se aplica una operación de reflejo horizontal.

(c) Textura de una imagen

(d) Textura de una imagen con cambio de posición

Texturas e Iluminación

Para aplicar la textura a un objeto cerrado, por ejemplo a la esfera, se usa el barrido circular, en la que se repetían los vértices de los extremos, pero sin añadir los triángulos degenerados. Ya que la deformación que se produce con los triángulos de las tapas sea similar para cada uno de ellos.

Las diferencias que hay entre usar el **barrido circular con la repetición de los vértices de los extremos** a la optima es que, en la de los vértices repetidos en el caso de la esfera los vértices P3, P7, P11, P15, P3 son el mismo vertice pero con distintas posiciones, mientras que en el optimo solo tiene 1 vertice y tiene la misma posición.

Otra diferencia importante es que se hay un problema en la estructura con los vértices repetidos, y es que el ultimo perfil como es igual al primero y tienen posiciones de textura diferentes, por ejemplo el triangulo (P1, P4, P5) tiene las coordenadas de textura ((0,0.33), (0.25,0), (0.25,0.33)), ese triangulo se dibujara bien, pero el triangulo (P13, P0, P1) con las coordenadas de textura ((0.75,0.33), (0,0), (0, 0.33)), con lo cual no se dibuja bien, porque intentamos reutilizar los puntos P0, P1 y P2, para solucionarlo es tan simple como poner que son distintos puntos sus intuyéndolos por los puntos P16, P17 y P18, que ocupan las posiciones de los puntos P0, P1 y P2, teniendo las coordenadas de textura ((1,0), (1,0.33), (1, 0.66)), y de esta forma se dibuja de forma correcta los triángulos en la imagen.

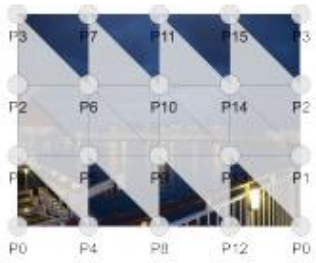


Figura 17.1: Localización de los puntos con respecto a las coordenadas de textura con distribución más regular pero errónea en el último perfil

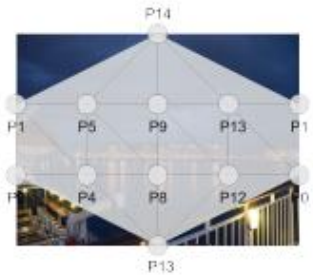
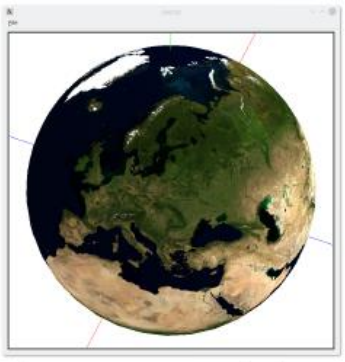


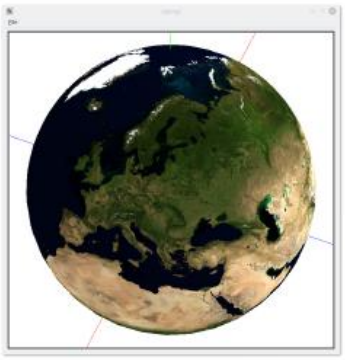
Figura 17.2: Localización de los puntos con respecto a las coordenadas de textura con distribución menos regular y errónea en el último perfil

Para añadir la iluminación a la textura, lo que hacemos es combinar la iluminación hecha con FLAT, GOURAUD o PHONG y añadírselas a la textura.

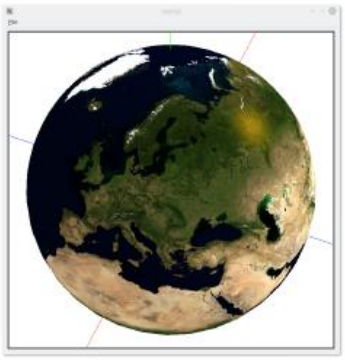
(a) Textura sin iluminación



(b) Textura con iluminación y sombreado plano



(c) Textura con iluminación y sombreado de Gouraud



(d) Textura con iluminación y sombreado de Phong

Selección

Una selección es dado el conjunto de elementos de la escena, queremos poder seleccionar individualmente cada uno de ellos, para aplicarle algún tipo de proceso (borrarlo, moverlo, etc).

Por ejemplo queremos seleccionar un triangulo y pintarlo de otro color, para ello tenemos que asociar un identificador a cada triangulo, para ello convertimos ese identificador en un color y también a la inversa de un color a un identificador. Con lo cual solo tenemos que pintar el objeto con el color correspondiente de cada identificador de cada triangulo.

Y ya solo nos faltaría cuando seleccione el ratón un triangulo obtenga el color de ese triangulo y lo convierta en un identificador para que podamos aplicar el cambio de color a ese triangulo.

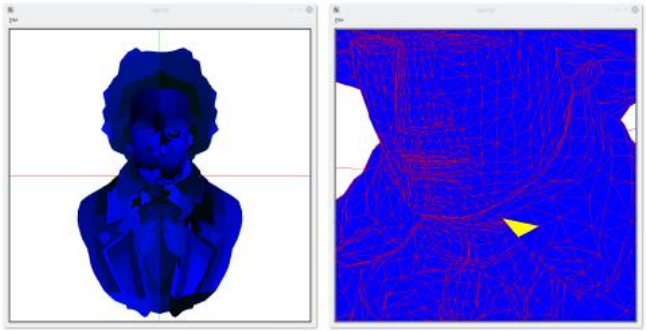
La conversión de identificador a color y viceversa, tenemos que tener claro que usamos el **modelo RGB**, de tal forma que cada componente tiene 1 byte, es decir **256 tonalidades**. De tal forma que para el **azul** van las posiciones **del 0 al 255**, para el **verde del 256 hasta 655355** y el **rojo desde 655356 hasta el 16777215**, con lo cual solo nos falta saber que valor corresponde a cada **componente RGB**, para ello **dividimos el numero entre 65536**, siendo el **cociente** de **la componente roja**, el **resto** de esa división se **divide entre 256**, siendo el **cociente** para **la componente verde** y el **resto** para **la componente azul**.

Solo nos falta saber que identificador tiene cada triangulo, es tan simple como obtener la posición de cada triangulo del vector.

Pero si a cada triangulo tiene un color diferente, la figura a mostrar se vera con multiples colores, para que la figura se vea de un solo color, pero pueda identificar cada triangulo con un color distinta usamos el **framebuffer**, es una zona de memoria donde se almacena la imagen final, no solo para visualizarla, sino también para realizar la selección, con esto no vamos a representar de forma visual el objeto, sino la codificación por posición (cada triangulo de un color). Al usar el **framebuffer** vamos a ver en la pantalla la visualización de la codificación solo un instante y después la visualización normal. La solución para evitar esto es utilizar otro **framebuffer**.

Con lo cual habrá que tener un **framebuffer** principal que será el que visualiza el objeto en pantalla y el resto de buffers que se puedan crear no se visualizaran por pantalla, a esos buffers se les llama **off-screen**.

Para ello tenemos que crear un **framebuffer** que tendrá asociado una zona de memoria para dibujar y otra zona de memoria para usar el **z-buffer**.



(a) Asignación de colores por identifcador (b) Visualización de la selección