Mallas (con vectores normales e) indexadas

A. Gavilanes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Recordatorio

- Los comandos que usamos para el anillo cuadrado:
  - **`glDrawArrays(GL_TRIANGLE_STRIP, 0, numvertices);`**
  - **`glDrawElements(GL_TRIANGLE_STRIP, numvertices, GL_UNSIGNED_INT, indices);`**

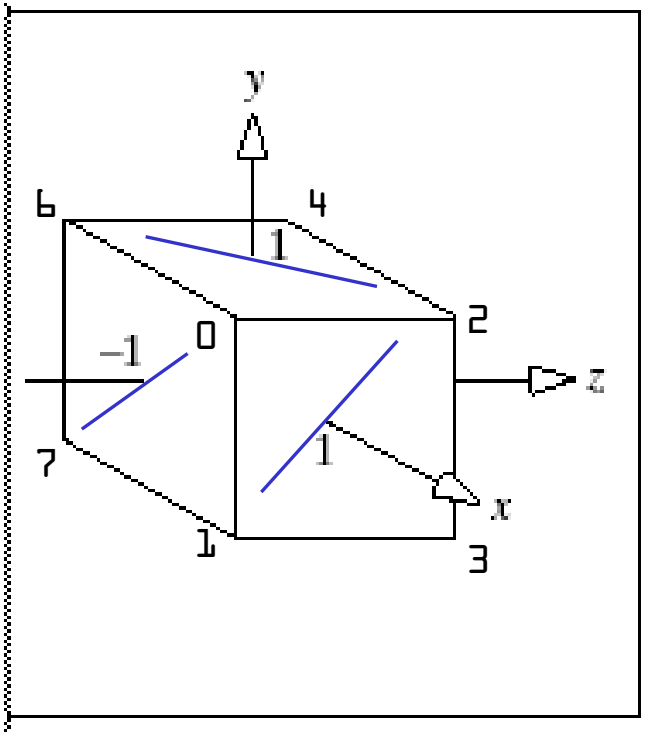
permiten mandar a GPU, un total de **`numvertices`** de aquellos (vertex) arrays activados en **`render()`** de **`Mesh`**, evitando así una sucesión de llamadas en modo inmediato del tipo **`glVertex()`**, **`glNormal()`**, etc.

- El primero de estos comandos no evita, si fuera necesario, repetir información en los (vertex) arrays. El segundo sí lo evita mediante el uso de índices

El problema de la repetición de la información

Cubo de lado 2, centrado en el origen

- El cubo tiene 8 vértices que abajo aparecen numerados del 0 al 7
- La primitiva que usamos para esta malla es **`GL_TRIANGLES`** y, para ella, necesitamos 36 vértices, pues hay 12 triángulos, 2 por cara
- En el vertex array, cada vértice se repetiría 4 o 5 veces (pues 4 vértices forman parte de 4 triángulos y los otros 4, de 5 triángulos)



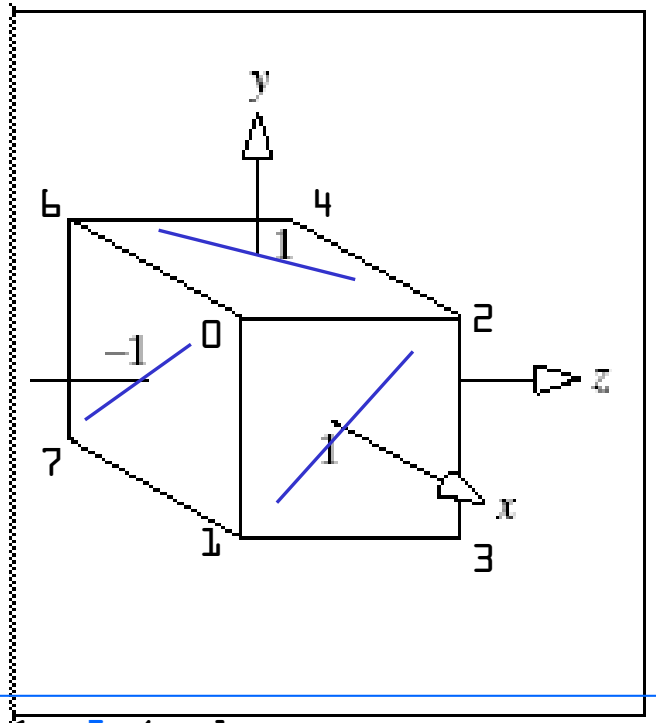
En lugar de repetir los vértices, para formar los triángulos se añade una tabla de índices:

vértices (8)

0	(1, 1,-1)
1	(1,-1,-1)
2	(1, 1, 1)
3	(1,-1, 1)
4	(-1, 1, 1)
5	(-1,-1, 1)
6	(-1, 1,-1)
7	(-1,-1,-1)

índices (36) (= 3 vértices * 12 triángulos)

0,1,2, 2,1,3,
2,3,4, 4,3,5,
4,5,6, 6,5,7,
6,7,0, 0,7,1,
4,6,2, 2,6,0,
1,7,3, 3,7,5



Vértice de la posición 5 (¡el que no se ve en la figura!) de la tabla de vértices (se repite 4 veces)

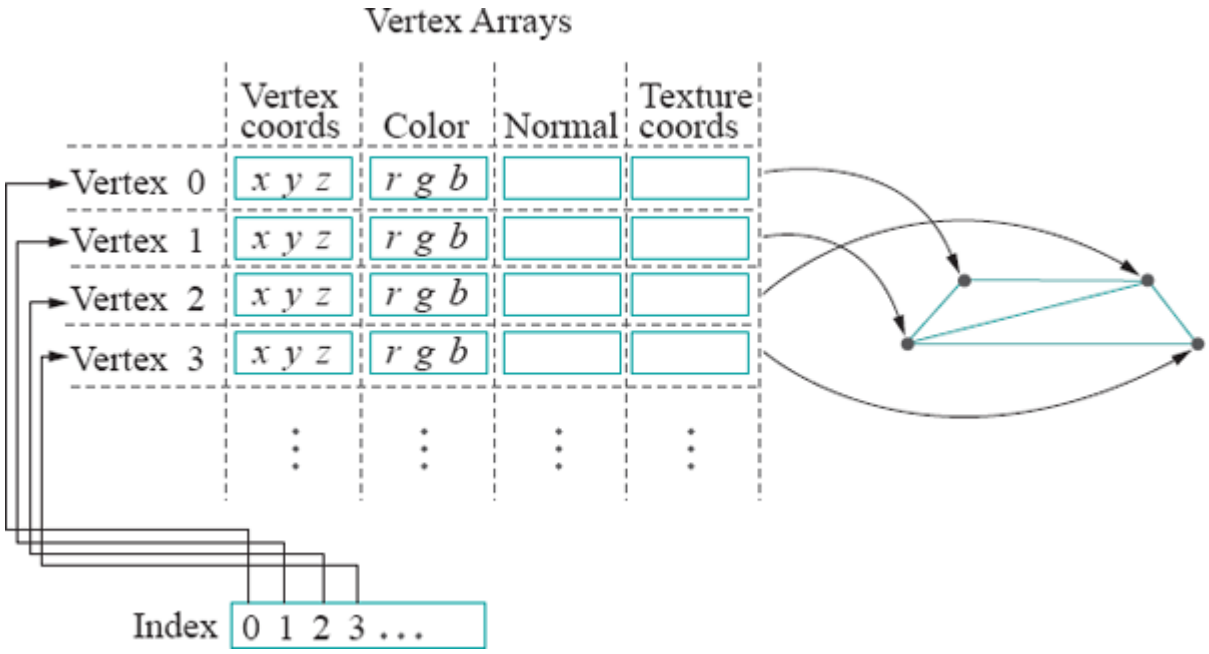
Mallas indexadas

Evitan la repetición de la información mediante el uso de índices. Están formadas por:

- *TABLA DE VÉRTICES:* contiene las coordenadas de los vértices de la malla (información de posición)
- *TABLA DE ÍNDICES:* contiene los índices (posiciones en la tabla de vértices) de los vértices de cada cara de la malla. Para la primitiva **`GL_TRIANGLES`**, esta tabla consta de numCaras * numVérticesCara elementos.
- *TABLA DE NORMALES:* contiene las coordenadas de los vectores normales a los vértices (información de orientación)
- *TABLA DE COLORES:* contiene información de los colores de los vértices
- *TABLA DE COORDENADAS DE TEXTURA:* contiene las coordenadas de textura de cada vértice

Estas últimas tablas tienen que ser del mismo tamaño que la tabla de vértices (numVertices).

Mallas con todos sus vertex arrays



Lighting equation in OpenGL

$$\text{vertex color} = \text{emission}_{\text{material}} +$$

$$\text{ambient}_{\text{light model}} * \text{ambient}_{\text{material}} +$$

$$\sum_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right)_i * (\text{spotlight effect})_i *$$

$$[\text{ambient}_{\text{light}} * \text{ambient}_{\text{material}} +$$

$$(\max \{ \mathbf{L} \cdot \mathbf{n} , 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}} +$$

$$(\max \{ \mathbf{s} \cdot \mathbf{n} , 0\})^{\text{shininess}} * \text{specular}_{\text{light}} * \text{specular}_{\text{material}}]_i$$

Vectores normales

- Cada componente de la tabla o array de normales es un vector normal de un vértice, es decir, es perpendicular a la tangente en ese vértice al objeto malleado
  - Hay tantos vectores normales como vértices
  - Cada vector normal:
    - constituye un atributo más del vértice
    - es perpendicular a la cara en ese vértice (es decir, el producto escalar con el vector tangente es igual a 0)
    - apunta hacia el exterior del objeto
    - debe estar normalizado (módilo 1)
      - Ojo con el comando **`glEnable(GL_NORMALIZE);`**
- Vector normal y sombreado del objeto (shading model).

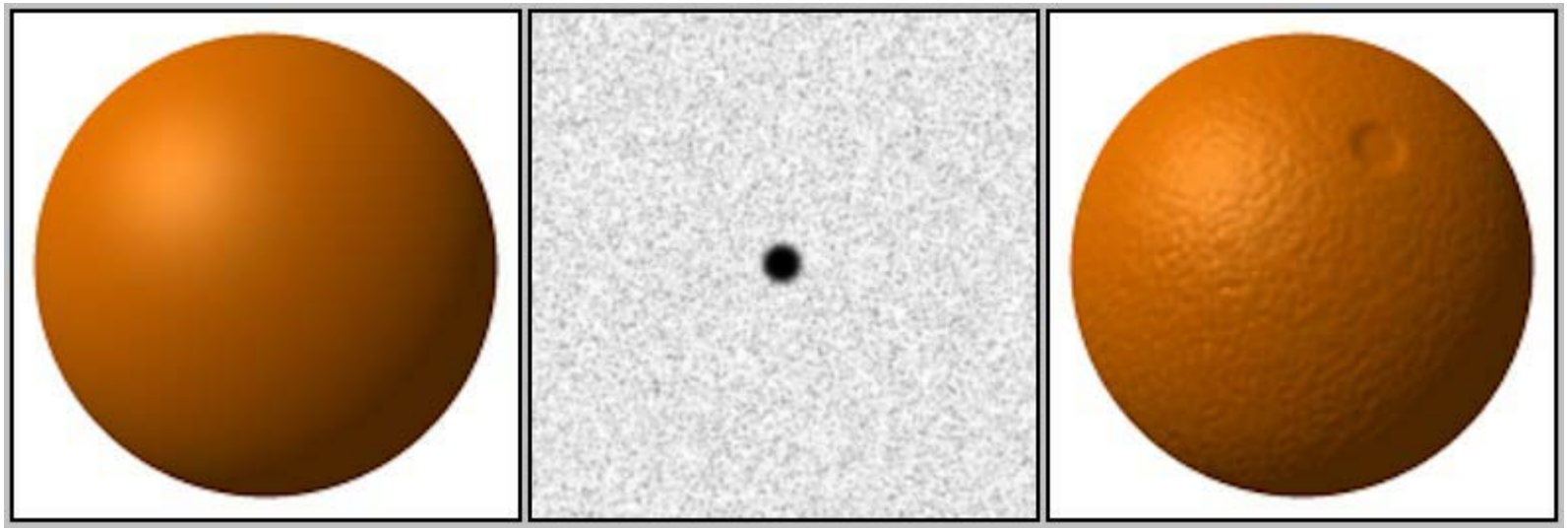
Vectores normales

- Los vectores normales de una cara se calculan a partir de los (índices de los) vértices que forman la cara
- Se sigue el convenio de proporcionar los índices de los vértices de cada cara en sentido antihorario (**`GL_CCW`**) según se mira la cara del objeto desde el exterior del mismo
- Este orden permite distinguir el interior y el exterior del objeto y a OpenGL le permite diferenciar entre caras frontales (**`GL_FRONT`**) y caras traseras (**`GL_BACK`**)
- El comando **`glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);`** colorea caras traseras, invirtiendo el sentido de los normales
- Los vectores normales se utilizan en el proceso de iluminación para determinar el color de los vértices

Cálculo de vectores normales

- En el renderizado de objetos malleados, los vectores normales son perpendiculares a caras, aunque hay tantos como vértices
- Lo habitual es calcular el vector normal a una cara y usarlo como parte del vector normal para los vértices de esa cara. ¿Cómo?
- Como hay tantos vectores normales como vértices, el vector normal de un vértice se puede calcular como la suma (normalizada) de los vectores normales de las caras en las que participa el vértice
- A veces se usan sumas ponderadas por el ángulo o por el área que forman las caras que concurren en un vértice
- En el renderizado de objetos con superficies curvas de ecuaciones conocidas (esferas, cilindros, etc.): el vector normal de un vértice se calcula a partir de las ecuaciones (paramétricas o implícita) de la superficie

Bump mapping



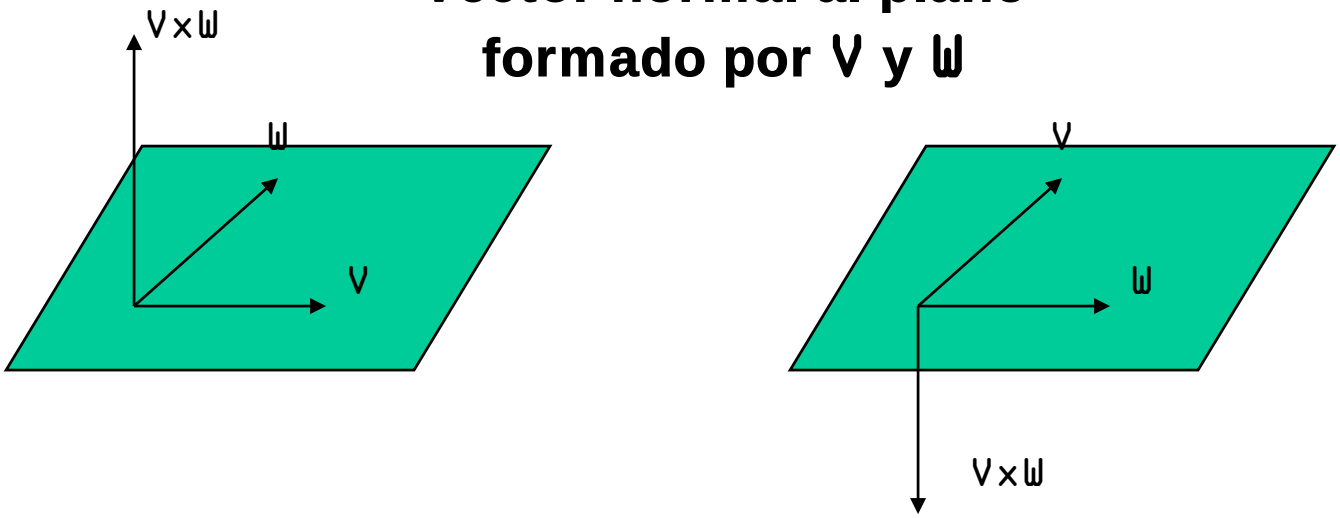
- Cuando se usan ciertas técnicas (bump mapping; aspecto de piel de naranja) se pueden especificar vectores normales por fragmento.
- Permite dar cierto aspecto (por ejemplo, rugosidad) sin cambiar la geometría del objeto
- Las normales se realinean siguiendo un cierto patrón
- James Blinn

- **Producto vectorial de dos vectores**:

$$\left.\begin{array}{l} V=(v_1,v_2,v_3) \\ W=(w_1,w_2,w_3) \end{array}\right\} V\times W=\begin{vmatrix} i & j & k \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{vmatrix}$$

$$|V\times W|=|V||W|\sin(\theta)$$

Vector normal al plano formado por V y W



Cálculo del vector normal a una cara

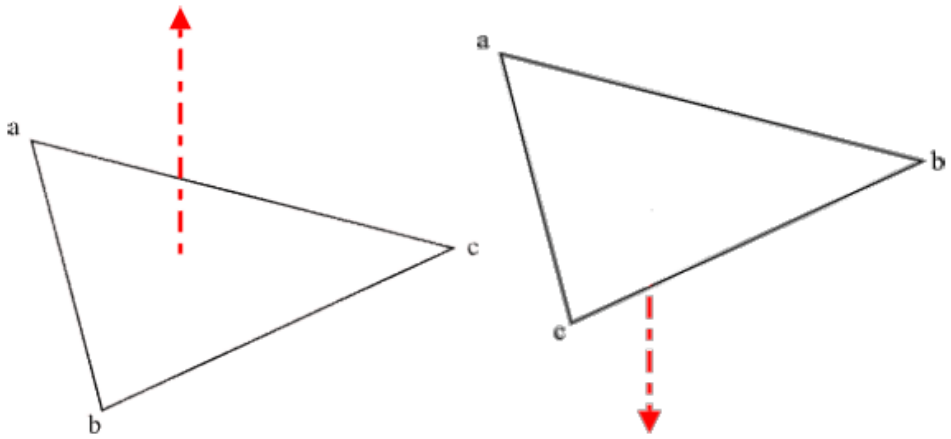
- Vector normal a un triángulo (a, b, c) dado en el orden CCW es un vector unitario (módulo 1) perpendicular al plano determinado por los vértices del triángulo y que apunta hacia fuera. Se puede obtener como el producto vectorial de dos vectores, normalizado

  Producto vectorial de dos vectores en CCW: $(\vec{ab} \times \vec{ac})$, $(\vec{ba} \times \vec{bc})$, ...

  normalize(cross(b - a, c – a))

- El orden de los vértices es importante:

  vector_normal(vértices en CW) = - vector_normal(vértices en CCW)



Cálculo del vector normal a una cara

- El vector **n** normal a una cara formada por los vértices de índices **{ind0, ind1, …, indn}** se puede calcular
  - Usando el producto vectorial. Sea **vi** el vértice de índice **indi**:

    n=normalize(cross((v2-v1),(v0-v1)))

    O también:

    n=normalize(cross((v1-v0),(vn-v0)))

    - Inconvenientes de este método
  - Usando el método de Newell

Cálculo del array de normales

- Construir el vector de normales del mismo tamaño que el de vértices

//m->indices

0	5	1	1	5	6	1	6	2	...

triáng 0 triáng 1 triáng 2

//m->vertices

(x, y, z)	(x, y, z)	(x, y, z)	(x, y, z)	(x, y, z)	...

m->normals = new ...

- Inicializar las componentes del vector de normales al vector 0

//m->normals

(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	...

- Recorrer los triángulos, es decir, recorrer m->indices haciendo:
  - Extraer los índices del triángulo a, b, c
  - Calcular el vector **n** normal al triángulo tal como se ha explicado
  - Sumar **n** al vector normal de cada vértice del triángulo

- Normalizar los vectores de m->normals

Mallas con vectores normales

- Para tener en cuenta los vectores normales añadimos a la clase **Mesh** un atributo para el array de vectores normales

```
class Mesh {
    protected:
        std::vector<glm::dvec3> vNormals;  // tabla de normales
        …
    public:
        virtual void render();
        …
};
```

- Es preciso también modificar **render()**

```
void Mesh::render() {
    … // se añaden comandos para la tabla de normales:
    if (vNormals.size() > 0) {
            glEnableClientState(GL_NORMAL_ARRAY);
            glNormalPointer(GL_DOUBLE, 0, vNormals.data());
    …
    glDisableClientState(GL_NORMAL_ARRAY);}
```

Mallas indexados

- Añadimos la clase **IndexMesh** como subclase de **Mesh** con un atributo más para el array de índices

```
class IndexMesh: public Mesh {
    protected:
        GLuint* vIndices = nullptr; // tabla de índices
        GLuint nNumIndices = 0;
        …
    public:
        IndexMesh() { mPrimitive = GL_TRIANGLES; }
        ~IndexMesh() { delete[] indices;}
        virtual void render() const;
        virtual void draw() const;
        …};
```

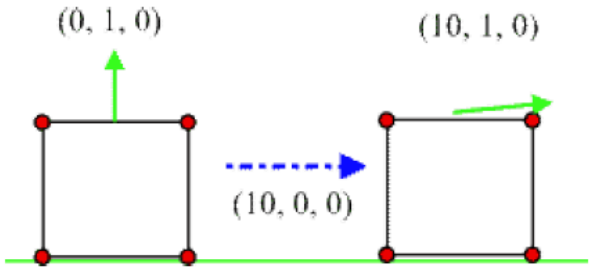
```
void IndexMesh::render() const {
     … // Comandos OpenGL para enviar datos de arrays a GPU
       // Nuevos comandos para la tabla de índices
       if (vIndices != nullptr) {
            glEnableClientState(GL_INDEX_ARRAY);
            glIndexPointer(GL_UNSIGNED_INT, 0, indices);
       }
      … // Comandos OpenGL para deshabilitar datos enviados
        // Nuevo comando para la tabla de índices:
       glDisableClientState(GL_INDEX_ARRAY);
}

// Comando para renderizar la malla indexada enviada
void IndexMesh::draw() const {
     glDrawElements(mPrimitive, nNumIndices,
       GL_UNSIGNED_INT, vIndices);
}
```

Método de Newell

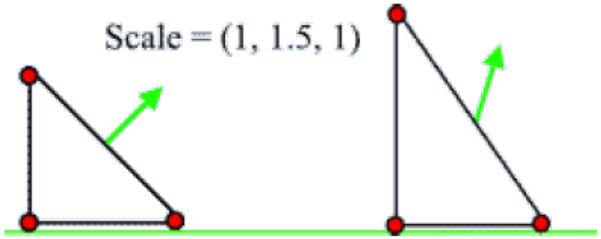
```
CalculoVectorNormalPorNewell(Cara C)
    n = (0, 0, 0);
    for i=0 to C.numeroVertices {
        vertActual=vertice[C->getVerticeIndice(i)];
         vertSiguiente=vertice[C->getVerticeIndice((i+1) % C.numeroVertices)];
        n.x+=(vertActual.y-vertSiguiente.y)*
             (vertActual.z+vertSiguiente.z);
        n.y+=(vertActual.z-vertSiguiente.z)*
             (vertActual.x+vertSiguiente.x);
        n.z+=(vertActual.x-vertSiguiente.x)*
             (vertActual.y+vertSiguiente.y);
    }
    return normaliza(n.x, n.y, n.z);
```

Transformación de vectores normales

- Vectores normales y matriz de modelado



Si la normal se transforma



Si la normal se transforma

- Matriz de transformación de vectores normales: ignora la translación e invierte la escala.