Mallas (con vectores normales e) indexadas

Informática Gráfica I

Material de: **Antonio Gavilanes**
Adaptado por: **Elena Gómez** y **Rubén Rubio**
{mariaelena.gomez,rubenrub}@ucm.es

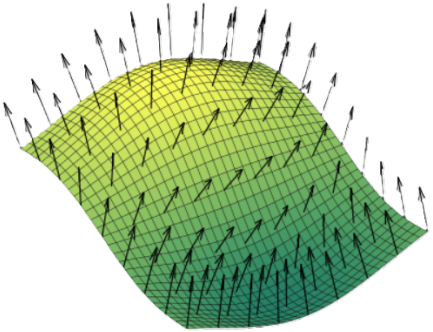
Contenido

Formas de representación de superficies

- Una **malla** (en inglés, *mesh*) es una colección de polígonos que se usa para representar la superficie de un objeto tridimensional.
- Estándar de representación de objetos gráficos:
  - Facilidad de implementación y transformación.
  - Propiedades sencillas.
- Representación exacta o aproximada de un objeto.
- Un elemento más en la representación de un objeto (color, material, textura).

Formas de representación de superficies

- Muchas caras de las mallas comparten vértices.
- Cada vértice y cada cara tiene un vector **normal**.
- La normal es importante porque:
  - descarta el renderizado de las caras posteriores
  - añade efectos de luz y sombra en función del ángulo por la normal y la dirección del foco



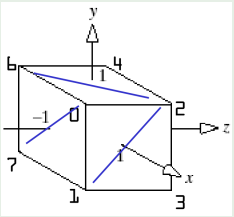
Repetición de información

Cubo de lado 2 y 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)

¡¡Ojo!!

El sistema de ejes cartesianos que se muestra no es el usual (se ve en el eje Z). Es *left-handed* frente al habitual que manejamos que es *right-handed*.



Repetición de información

Cubo de lado 2 y centrado en el origen

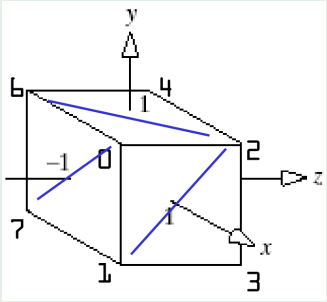
- En lugar de repetir los vértices para formar los triángulos se añade a una tabla de índices

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)

Vértices (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

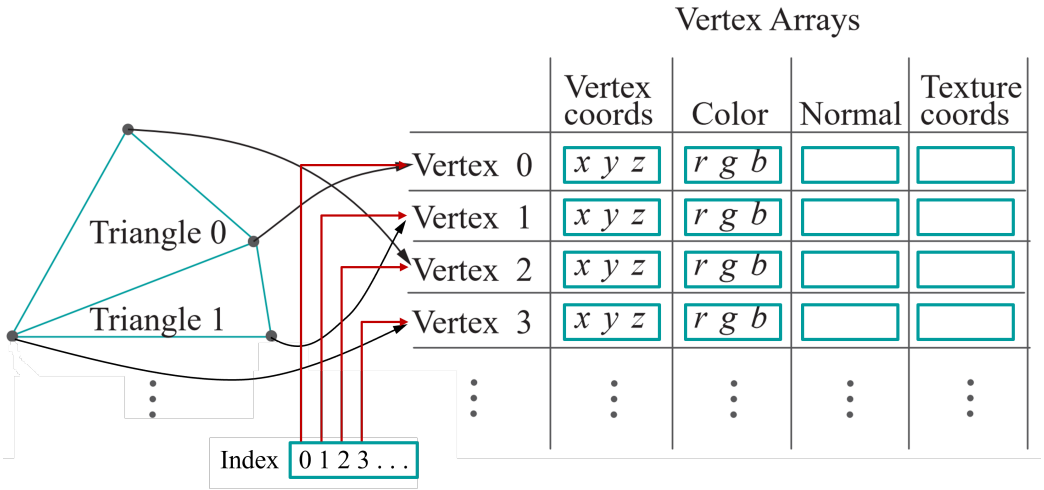


Vértice de la **posición 5** (¡el que no se ve en la figura!) de la tabla de vértices (se repite 4 veces)

Mallas indexadas

- Evitan la repetición de la información mediante el uso de índices. Están formadas por columnas de:
  - **Vértices**: contiene las coordenadas de los vértices de la malla (información de posición)
  - **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.
  - **Normales**: contiene las coordenadas de los vectores normales a los vértices (información de orientación)
  - **Colores**: contiene información de los colores de los vértices
  - **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



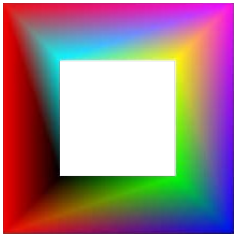
Renderización de primitivas

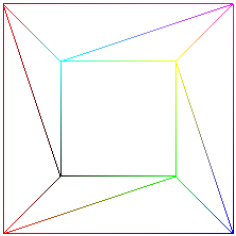
- Modo inmediato en OpenGL 1.0: `glVertex()`, `glNormal()`,...
  - Este modo y el siguiente son todavía ampliamente usados hoy.
- Vertex arrays en OpenGL 1.1
  - Aunque se decide deprecar este modo y el anterior, se pueden seguir usando todavía si la implementación de OpenGL soporta el perfil de compatibilidad.
  - Estos modos no almacenan datos en la GPU. Cada vez que se invocan, los datos implicados se pasan a la GPU.
- Vertex Buffer Objects (VBO) en OpenGL 1.5
  - Modo recomendado en la actualidad
  - A diferencia de los modos anteriores, los VBO's permiten almacenar los datos en la GPU
  - Son similares a los objetos de textura

Modo inmediato

- Cómo pasar vértices y colores en modo inmediato

```
glBegin(GL_TRIANGLE_STRIP);
  glColor3f(0.0, 0.0, 0.0); glVertex3f(30.0, 30.0, 0.0);
  glColor3f(1.0, 0.0, 0.0); glVertex3f(10.0, 10.0, 0.0);
  glColor3f(0.0, 1.0, 0.0); glVertex3f(70.0, 30.0, 0.0);
  glColor3f(0.0, 0.0, 1.0); glVertex3f(90.0, 10.0, 0.0);
  glColor3f(1.0, 1.0, 0.0); glVertex3f(70.0, 70.0, 0.0);
  glColor3f(1.0, 0.0, 1.0); glVertex3f(90.0, 90.0, 0.0);
  glColor3f(0.0, 1.0, 1.0); glVertex3f(30.0, 70.0, 0.0);
  glColor3f(1.0, 0.0, 0.0); glVertex3f(10.0, 90.0, 0.0);
  glColor3f(0.0, 0.0, 0.0); glVertex3f(30.0, 30.0, 0.0);
  glColor3f(1.0, 0.0, 0.0); glVertex3f(10.0, 10.0, 0.0);
glEnd();
```





Modo inmediato: Variante

- Cómo pasar vértices y colores mediante punteros

```
static float vertices[8][3] = {
  {30.0, 30.0, 0.0}, {10.0, 10.0, 0.0},
  {70.0, 30.0, 0.0}, {90.0, 10.0, 0.0},
  {70.0, 70.0, 0.0}, {90.0, 90.0, 0.0},
  {30.0, 70.0, 0.0}, {10.0, 90.0, 0.0}
};

static float colors[8][3] = {
  {0.0, 0.0, 0.0}, {1.0, 0.0, 0.0},
  {0.0, 1.0, 0.0}, {0.0, 0.0, 1.0},
  {1.0, 1.0, 0.0}, {1.0, 0.0, 1.0},
  {0.0, 1.0, 1.0}, {1.0, 0.0, 0.0}
};
```

```
// Dibujo del cuadrado anular
glBegin(GL_TRIANGLE_STRIP);
for (int i = 0; i < 10; ++i) {
  glColor3fv(colors[i % 8]);
  glVertex3fv(vertices[i % 8]);
}
glEnd();
```

Modo inmediato

- **Ventajas** de la variante:
  - Los vértices y colores pueden ser reutilizados; de hecho, solo son necesarios 8 vértices (y 8 colores), y no 10, como exige la primitiva `GL_TRIANGLE_STRIP`.
  - La definición de vértices y colores tiene lugar en una parte específica del código.
  - Mejor uso de la memoria, menor redundancia.
  - En consecuencia, más facilidad para la depuración, más eficiencia.
- **Inconvenientes** (del modo inmediato):
  - Las mallas complejas se suelen leer de un archivo o se crean procedimentalmente y el proceso de mandar los datos a OpenGL supone un envío por vértice, desde el lugar donde se encuentren.
  - Sería mejor enviar directamente un array de datos que no ejecutar millones de llamadas a `glVertex()`, `glColor()`, ...

Vertex arrays

- Cómo pasar vértices y colores mediante **vertex arrays**:
  - Primero se definen los arrays (llamados *vertex arrays*) que se desean pasar.

```
static float vertices[] = {
  30.0, 30.0, 0.0,
  10.0, 10.0, 0.0,
  70.0, 30.0, 0.0,
  90.0, 10.0, 0.0,
  70.0, 70.0, 0.0,
  90.0, 90.0, 0.0,
  30.0, 70.0, 0.0,
  10.0, 90.0, 0.0 };
```

```
static float colors[] = {
  0.0, 0.0, 0.0,
  1.0, 0.0, 0.0,
  0.0, 1.0, 0.0,
  0.0, 0.0, 1.0,
  1.0, 1.0, 0.0,
  1.0, 0.0, 1.0,
  0.0, 1.0, 1.0,
  1.0, 0.0, 0.0 };
```

Vertex arrays

- Y en `draw()` de la clase `Mesh`, los vértices y colores se pueden recuperar con el comando `glArrayElement()`:

```
glBegin(GL_TRIANGLE_STRIP);
  // Cuando se usa glArrayElement(i);
  // vertices[i] y colors[i] se recuperan a la vez
  for (int i = 0; i < 10; ++i)
    glArrayElement(i % 8);
glEnd();
```

Vertex arrays

- Los dos *vertex arrays* (`vertices` y `colors` de la transparencia 15) se activan/desactivan (con los comandos `glEnableClientState()`/`glDisableClientState()`) antes/después de llamar a `draw()` de la transparencia anterior.
- Además, antes de recuperar los datos se especifica dónde están almacenados y en qué formato, con los comandos `glVertexPointer()`, `glColorPointer()`.

```
// Activación de los vertex arrays
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
  // Especificación del lugar y formato de los datos
  glVertexPointer(3, GL_FLOAT, 0, vertices);
  glColorPointer(4, GL_FLOAT, 0, colors);
  draw();
// Desactivación de los vertex arrays
glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

OJO!!!

Aparece `GL_FLOAT` porque los datos son `float`. Si fueran `double`, es necesario escribir `GL_DOUBLE`.

Vertex arrays

- Los vértices y colores aparecen en arrays unidimensionales, aunque podrían haberse introducido también mediante arrays bidimensionales
- La instrucción `glArrayElement(i % 8);` extrae simultáneamente el vértice y el color de lugar `(i % 8)`-ésimo.
  Por eso los arrays no repiten componentes y tienen tamaño 8.
- La especificación del lugar donde se encuentran los vertex arrays se hace con el comando

```
gl..Pointer(size, type, stride, *pointer);
```

  donde `size` es el número de datos (por vértice, por color, ...), `type` es el tipo de los datos, `stride` es el offset que se debe saltar antes de empezar a leer (0, si los datos aparecen consecutivos) y `*pointer` es la dirección del vertex array
- Ojo, para el vertex array de normales el comando solo tiene 3 parámetros

```
glNormalPointer(type, stride, *pointer);
```

Vertex arrays sin índices

- Para renderizar vértices y colores, sin referencia a normales ni a índices se puede usar el comando:

  ```
  glDrawArrays(GL_TRIANGLE_STRIP, 0, numvertices);
  ```

  y se toman los elementos de los vertex arrays activos, tal como marque la primitiva y secuencialmente. Pero como toma componentes, no lugares como `glArrayElement()`, los vertex arrays pueden repetir elementos.
- La forma general de este comando es:

  ```
  glDrawArrays(mPrimitive, first, size());
  ```

  que dibuja con `mPrimitive`, usando `size()` elementos de los arrays de vértices y colores, empezando en la posición `first`.
- Observa que esta instrucción es la que se usa en la definición de `draw()` de la clase `Mesh`.

Vertex arrays con índices

- Se puede dibujar el cuadrado anular con un solo comando y sin repetir vértices ni colores:

```
glDrawElements(GL_TRIANGLE_STRIP, 10, GL_UNSIGNED_INT, stripIndices);
```

  en lugar de escribir varios comandos como:

```
for (int i = 0; i < 10; ++i)
  glArrayElement(i % 8);
```

  o en lugar de repetir elementos como:

```
glDrawArrays(GL_TRIANGLE_STRIP, 0, numvertices);
```

- Para usar este comando es necesario proporcionar los índices de las componentes de los vertex arrays (`vertices` y `colors`) donde se encuentran los datos, en el orden en que los tomará la primitiva que se use. En nuestro caso, la primitiva es `GL_TRIANGLE_STRIP` con lo que los índices son:

```
unsigned int stripIndices[] = { 0, 1, 2, 3, 4, 5, 6, 7, 0, 1 };
```

Vertex arrays con índices

- En nuestro ejemplo, la instrucción:

```
glDrawElements(GL_TRIANGLE_STRIP, 10, GL_UNSIGNED_INT, stripIndices);
```

  extrae los datos de 10 vértices en un solo comando.
- Observad que es más eficiente una llamada a este comando que no 10 llamadas al comando:

```
glArrayElement();
```

- La forma general del comando es

```
glDrawElements(mPrimitive, count, type, indices);
```

  donde `count` es el número de índices y `type` es el tipo de los índices.
- Este comando extrae elementos de los arrays de `vertices` y `colors`, pero en el orden que indican los índices. Recuérdese que, en el código del ejemplo, los índices dictan que los vértices se extraigan en el mismo orden que se siguió en el `for`.

Vectores normales

- **Ecuación de la luz en OpenGL**

$$
\begin{aligned}
\text{vertex color} = {} & \text{emission}_{\text{material}} + \\
& \text{ambient}_{\text{light model}} \times \text{ambient}_{\text{material}} + \\
& \sum_{i=0}^{n-1} \frac{1}{k_c + k_l d + k_q d^2} \times (\text{spotlight effect})_i \times \\
& [\\
& \quad \text{ambient}_{\text{light}} \times \text{ambient}_{\text{material}} + \\
& \quad (max\{L \cdot n, 0\}) \times \text{diffuse}_{\text{light}} \times \text{diffuse}_{\text{material}} + \\
& \quad (max\{s \cdot n, 0\})^{\text{shininess}} \times \text{specular}_{\text{light}} \times \text{specular}_{\text{material}} \\
&]_i
\end{aligned}
$$

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 (vector de módulo 1)
- Vector normal y sombreado del objeto (**shading model**)

¡¡Ojo!!

Comando `glEnable(GL_NORMALIZE);`

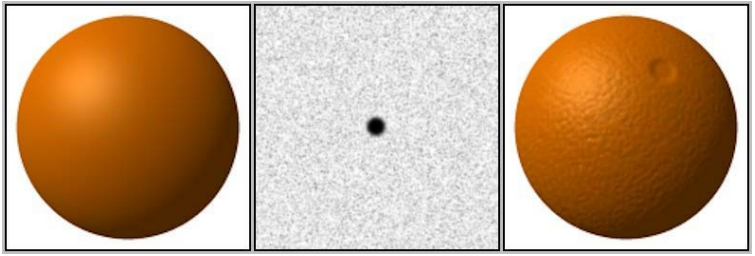
Vectores normales

- 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 los 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.
- 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

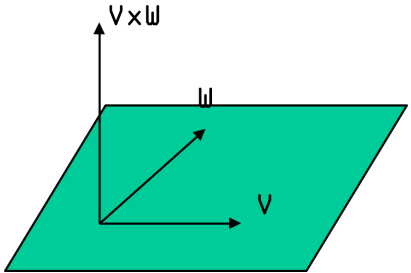


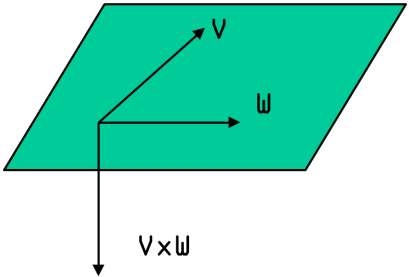
- Cuando se usan ciertas técnicas (*bump mapping* o aspecto de piel de naranja) se pueden especificar vectores normales por fragmento.
- Permite dar cierto aspecto (por ejemplo, rugosidad) sin cambiar la geometría del objeto
- Las normales se realinean siguiendo un cierto patrón
- Método desarrollado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

Producto vectorial de dos vectores

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

Vector normal al plano formado por V y W





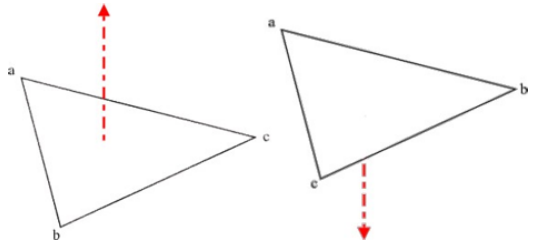
Cálculo del vector normal a una cara

- 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:

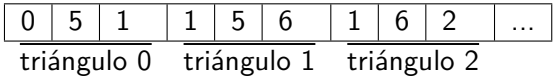
    ```
    n = normalize(cross((v1-v0),(vn-v0)))
    ```

    ☞ Inconvenientes de este método
  - Usando el método de Newell

Cálculo array de normales

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

  `//m->indices`



  `//m->vertices` (0,0,0) (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) ...

- 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;
  ...
public:
  virtual void render();
  ...
};
```

Mallas con vectores normales

- 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 indexadas

- 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[] vIndices;}
  virtual void render() const;
  virtual void draw() const;
  ...
};
```

Mallas indexadas

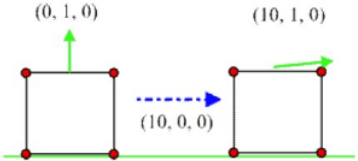
```
void IndexMesh::render() const {
  ... // Comandos       OpenGL para enviar datos de arrays a GPU
  // Nuevos comandos para la tabla de índices
  if (vIndices != nullptr) {
    glEnableClientState(GL_INDEX_ARRAY);
    glIndexPointer(GL_UNSIGNED_INT, 0, vIndices);
  }
  ... // Comandos       OpenGL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

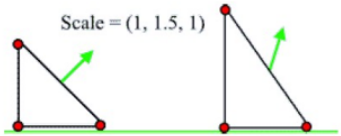
```
calculoVectorNormalPorNewell(Cara C) {
  n = (0, 0, 0);
  for i=0 to C.numeroVertices {
    vertActual=vertice[C->getVerticeIndice(i)];
    vertSiguiente=vertice[C->getVerticeIndice((i+1) % C.numeroVertices)];
    n.x +=(vertActual.y-vertSiguiente.y) * (vertActual.z+vertSiguiente.z);
    n.y +=(vertActual.z-vertSiguiente.z)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