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

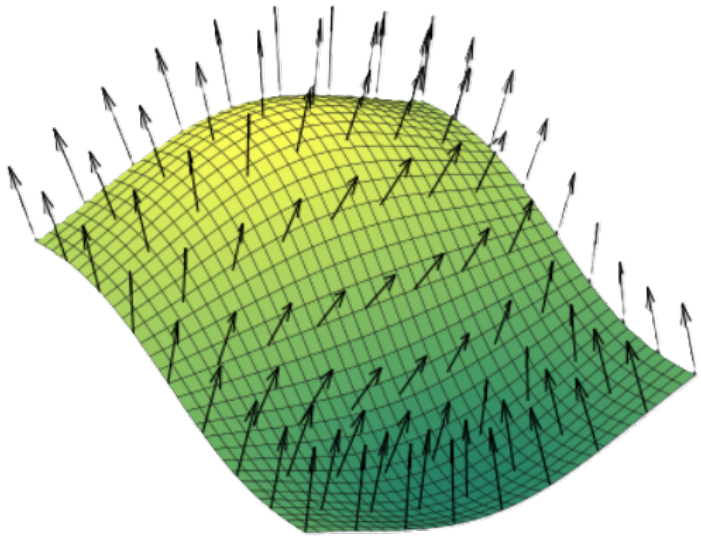
Contenido

Mallas

- 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).

Mallas y vectores normales

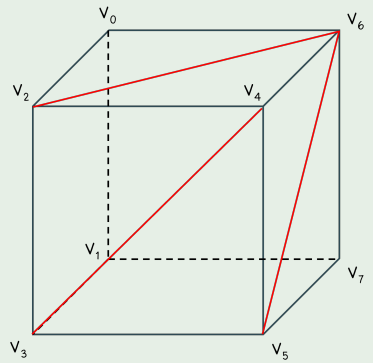
- Muchas caras de las mallas comparten vértices.
- Cada vértice y cada cara tiene un vector **normal**.
- La normal es importante porque:
  - descarta el renderizado de las caras posteriores
  - añade efectos de luz y sombra en función del ángulo por la normal y la dirección del foco



Repetición de información

Cubo de lado 2 y centrado en el origen

- El cubo tiene 8 vértices (numerados del 0 al 7).
- La primitiva que se usa es `GL_TRIANGLES` y, se neceitan 36 vértices (12 triángulos, 2 por cara).
- En el array de vértices, cada uno se repetiría 4 o 5 veces (pues 4 vértices forman parte de 4 triángulos y los otros 4, de 5 triángulos).



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.

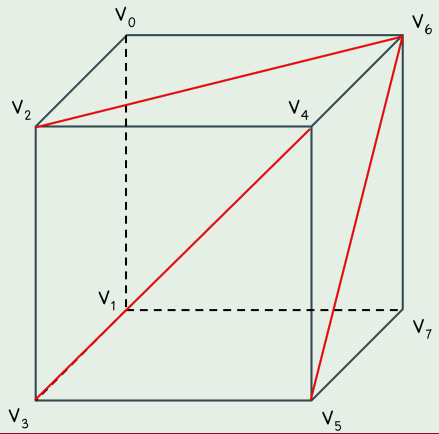
Vértices (8)

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**)



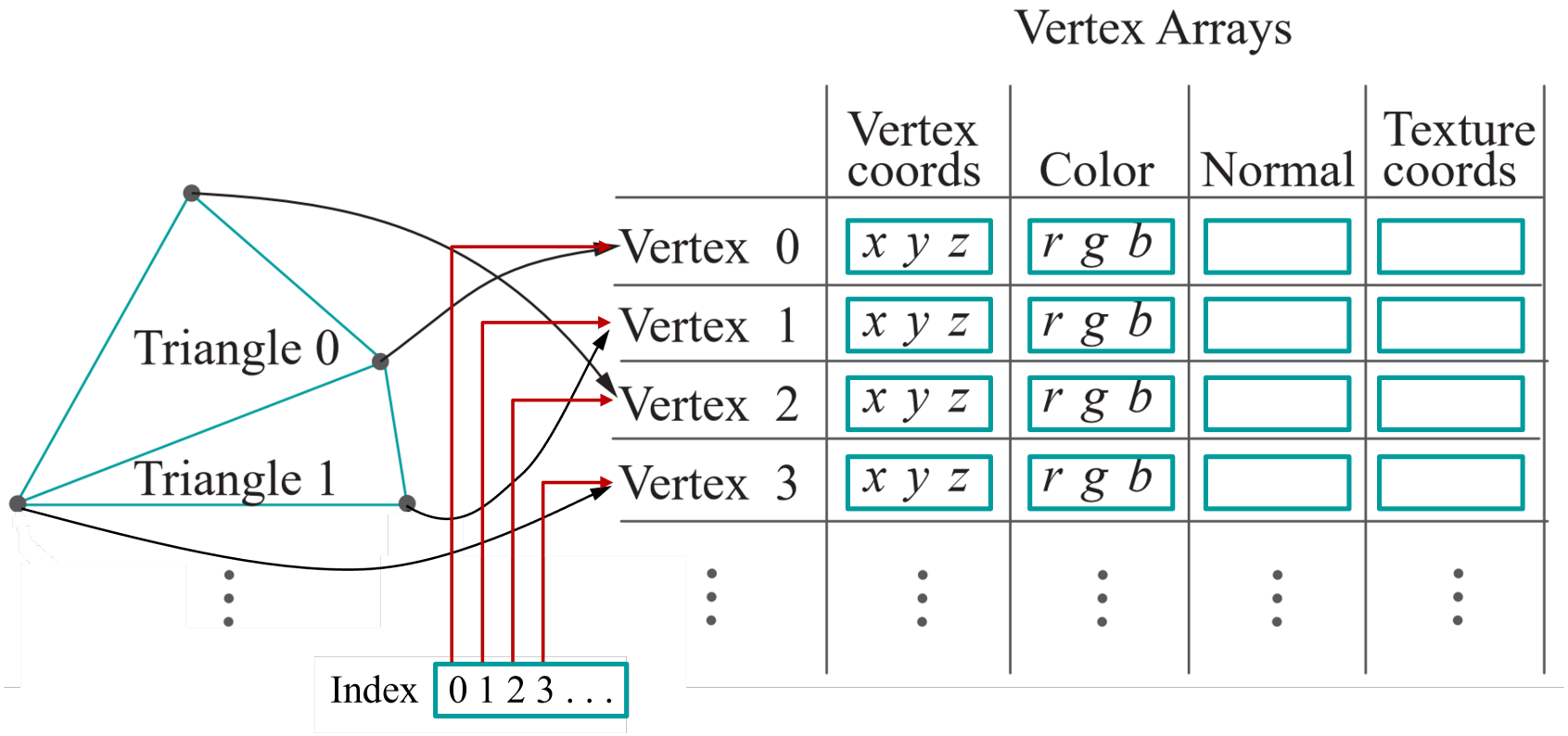
Mallas indexadas

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()`,...
  - Los vértices y sus atributos residen en el código
  - Se pasan uno a uno a la biblioteca gráfica con llamadas a función
  - Este modo y el siguiente son todavía ampliamente usados hoy.
- Vertex arrays en OpenGL 1.1
  - Los vértices y sus atributos reside en la memoria de la CPU
  - Se cargan vectores de vértices y atributos de la CPU a la GPU en cada renderizado
  - Aunque se decide deprecar este modo y el anterior, se pueden seguir usando todavía si la implementación de OpenGL soporta el perfil de compatibilidad.

Renderización de primitivas

- Vertex Buffer Objects (VBO) en OpenGL 1.5
  - Los vértices y sus atributos reside en la GPU
  - Modo recomendado en la actualidad y el único compatible con el *core profile*
  - Similares a los objetos de textura
  - Es lo que hemos usado en las prácticas

Vertex arrays objects

- Los **Vertex Arrays Objects (VAO)** y **Vertex Buffer Object (VBO)** permiten almacenar en la memoria de la GPU una secuencia de vértices con sus características para enviarlos al cauce gráfico
- Un VBO es un búfer interno de la tarjeta gráfica que almacena arrays de vértices, colores, coordenadas de textura, etc.
- Un VAO es una estructura que agrupa en la GPU toda la información de vértices que requieren las primitivas gráficas
- El VAO no almacena directamente los datos de los vértices, sino referencias (**descriptores**) a los VBO que realmente contienen los datos

Vertex arrays objects: Creación

Generación	`glGenVertexArrays(1, &VaoId)`
Eliminación	`glDeleteVertexArrays(1, &VaoId)`
Activación	`glBindVertexArray(VaoId)`
Desactivación	`glBindVertexArray(0)`

- Generar un VAO reserva espacio para él en la memoria de la GPU, que se ha de liberar cuando ya no haga falta
- Solo puede haber un VAO activo en cada momento y se selecciona con `glBindVertexArray` (como con las texturas)
- Un VAO se necesita activar para renderizarlo y para modificarlo

Vertex buffer objects: Creación

Generación	`glGenBuffers(1, &VboId)`
Eliminación	`glDeleteBuffers(1, &VaoId)`
Activación	`glBindBuffer(tipo, VboId);`
Desactivación	`glBindBuffer(tipo, 0)`

- Se generan, eliminan, activan y desactivan de forma similar a los VAO (y a las texturas)
- Los tipos más habituales son `GL_ARRAY_BUFFER` (vértices, colores, coordenadas de textura, normales) y `GL_ELEMENT_ARRAY_BUFFER` (índices)
- Solo puede haber un VBO de cada tipo activo en cada momento y que activarlos para modificarlos

Vertex buffer objects: Carga de datos

Los datos se cargan con

```
glBufferData(tipo, tamaño, puntero a los datos, modo);
```

- El tamaño se indica en bytes y se pasa el puntero al inicio
- El tipo es el mismo que en `glBindBuffer`
- El modo es `GL_`*freq*`_`*nature* donde *freq* indica la frecuencia que se modificarán y usarán los datos (`STREAM`, `STATIC`, `DYNAMIC`) y *nature* el uso que se les dará (`DRAW`, `READ`, `COPY`).
- El modo habitual será `GL_STATIC_DRAW`: los datos se modifican una sola vez y se utilizan muchas en los comandos de renderizado de OpenGL

Vertex buffer objects: Carga de datos

Ejemplo de carga de datos

```
GLuint VboId;
std::vector<glm::vec3> vertices = /* ... */

glGenBuffers(1, &VboId);  // crea el VBO y guarda su ident.
glBindBuffer(GL_ARRAY_BUFFER, VboId);  // activa el VBO

glBufferData(GL_ARRAY_BUFFER,  // tipo de búfer
             vertices.size() * sizeof(glm::vec3),  // tamaño
             vertices.data(), GL_STATIC_DRAW);  // puntero, modo

glBindBuffer(GL_ARRAY_BUFFER, 0);  // desactiva el VBO
```

Atributos de un VAO

- Un VAO define varios *vertex attribute arrays* numerados para poder acceder a ellos desde el cauce gráfico y cuyos datos están en VBOs
- Vértices, colores, coordenadas de textura, etc. serán cada uno un array de atributos
- Suponiendo un VAO y VBO activos

```
glBindVertexArray(VaoId);
glBindBuffer(tipo, VboId);
```

el atributo k y sus propiedades se definen con

```
glVertexAttribPointer(k, tamaño, tipo, normalizar,
                      stride, offset);
glEnableVertexAttribArray(k);
```

Atributos de un VAO

```
glVertexAttribPointer(k, tamaño, tipo, normalizar,
                      stride, offset);
```

- El tamaño se refiere al número de componentes (1, 2, 3 o 4) de cada vector. Por ejemplo, en un color RGBA será 4 y unas coordenadas de textura 2.
- El tipo es el tipo de cada componente: `GL_FLOAT`, `GL_DOUBLE`, `GL_INT`, etc.
- `stride` y `offset` indican donde empieza el array en el VBO activo y la distancia entre elementos. Esto permite meter en un búfer toda la información como si cada vértice fuera un `struct` (nosotros no lo usamos así y pondremos a cero ambos parámetros).

Atributos de un VAO

Ejemplo de vinculación del array de vértices

```
GLuint vertexVBO, meshVAO;
glBindVertexArray(meshVAO);
glBindBuffer(GL_ARRAY_BUFFER, vertexVBO);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, 0);  // o vincular otro
glBindVertexArray(0);
```

. . . . . .

```
layout (location = 0) in vec3 aPos;  // así accedemos en GLSL
```

Dibujo

1. Se enlaza el VAO con `glBindVertexArray`.
2. Se dibuja con `glDrawArrays` o `glDrawElements`.
3. Se desenlaza el VAO

Ejemplo

```
glBindVertexArray(VaoId); // Enlaza el VAO
// Dibuja la geometría
glDrawArrays(GL_TRIANGLES, 0, numVertices);
glBindVertexArray(0); // Desactiva el VAO
```

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.

Vectores normales

- 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**).

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`).
- 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.

Cálculo de los vectores normales

- 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.

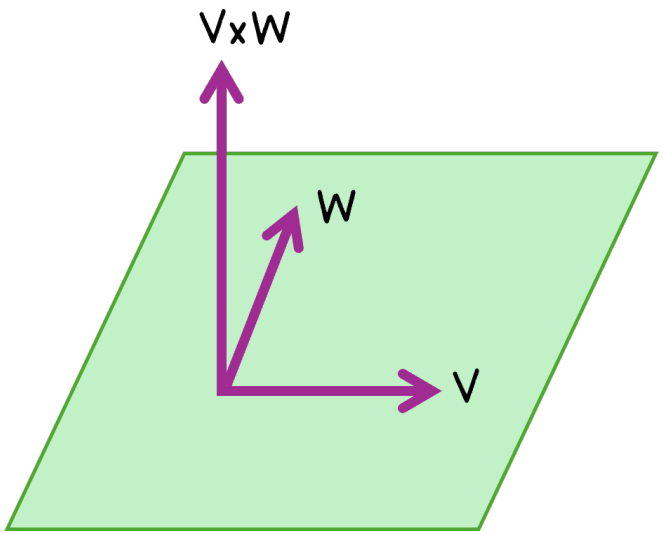
Producto vectorial de dos vectores

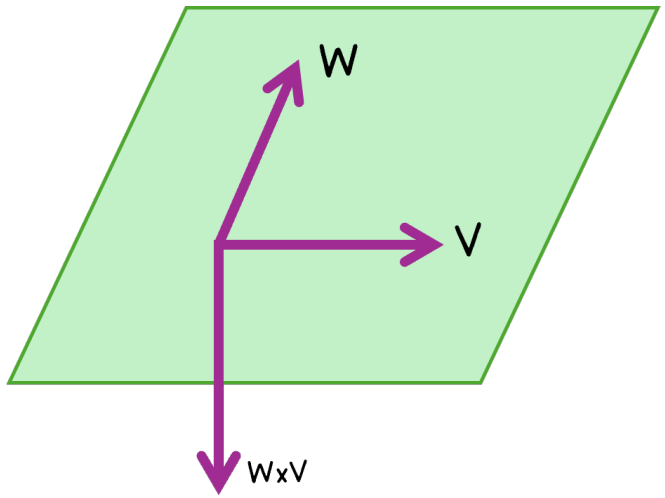
$$\left.\begin{aligned} V &= (v_1, v_2, v_3) \\ W &= (w_1, w_2, w_3) \end{aligned}\right\}$$

$$V \times W = \begin{vmatrix} i & j & k \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{vmatrix}$$

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

Vector normal al plano formado por V y W





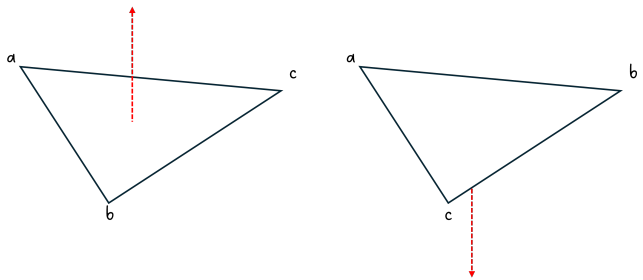
Cálculo del vector normal a una cara

- 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 en CCW, normalizando: $\overrightarrow{ab} \times \overrightarrow{ac}$, $\overrightarrow{ba} \times \overrightarrow{bc}$, ...

```
normalize(cross( b - a, c - a ))
```

- El orden de los vértices es importante:

```
vector_normal(vértices CW) = -vector_normal(vértices 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:

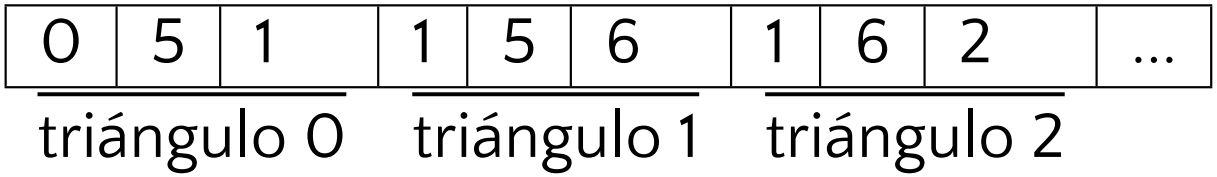
    ```
    n = normalize(cross((v1-v0),(vn-v0)))
    ```

    Inconvenientes de este método
  - Usando el método de Newell.

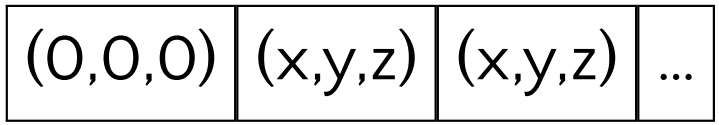
Cálculo array de normales

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

```
//m->indices
```



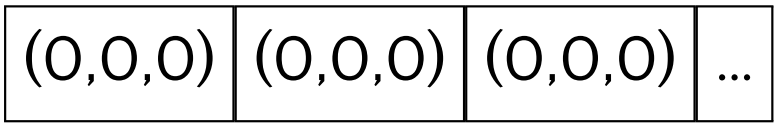

```
//m->vertices
```




```
m->normals = new ...
```

- Inicializar las componentes del vector de normales al vector 0

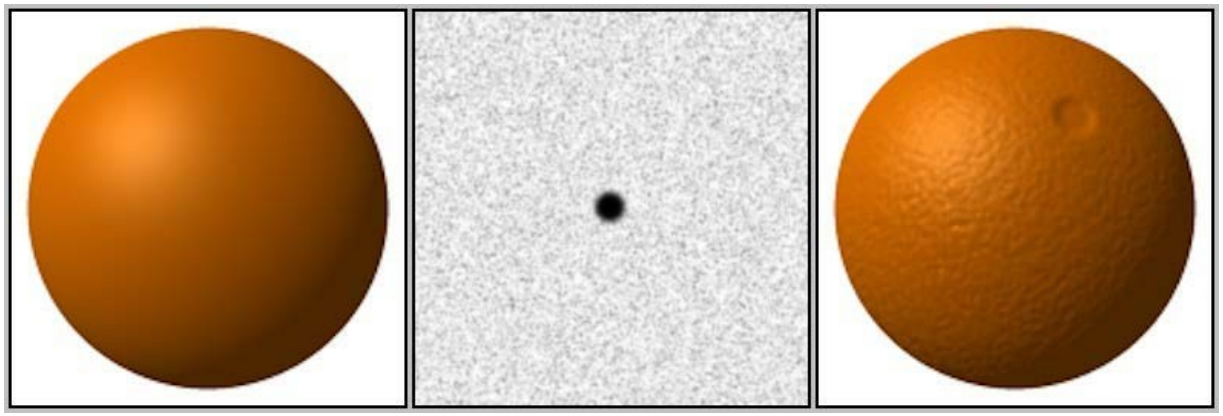
```
//m->normals
```



Cálculo array de normales

- 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`

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.