Modelado de superficies en 3D

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

Formas de representación de superficies

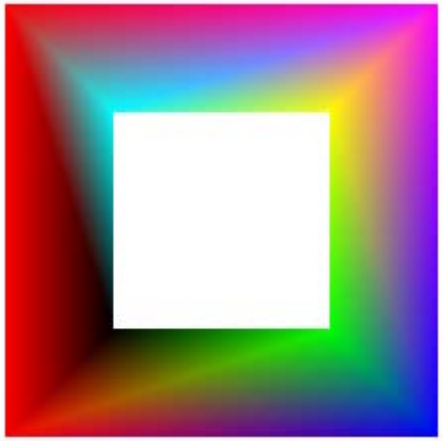
- Una malla (en inglés, *mesh*) es una colección de polígonos que se usa para representar la superficie de un objeto tridimensional
- Estándar de representación de objetos gráficos
  - Facilidad de implementación y transformación
  - Propiedades sencillas
- Representación exacta o aproximada de un objeto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

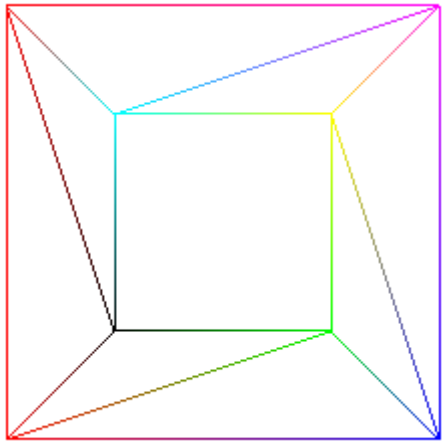
Renderización de primitivas

- 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 en OpenGL 3.0
  - Los vertex arrays son obligatorios en las versiones últimas (OpenGL 4.3)
  - 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

- 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();
```





Variante del modo inmediato

- 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();
```

- 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: los 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 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 **render()** 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();
```

- Los dos vertex arrays (**vertices** y **colors**) se activan/desactivan (con los comandos **glEnableClientState()/ glDisableClientState()**) antes/después de recuperar los datos con el código del ítem 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);
     ... // Aquí va el código del ítem anterior
// Desactivación de los vertex arrays
glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

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 los `(i % 8)`-ésimos vértice y color

- 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 con índices

- Se puede dibujar el cuadrado anular con un solo comando

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

en lugar de escribir

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

- Para hacerlo 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, sI);
```

  extrae los datos de 10 vertices en un solo comando

- Piénsese que es más eficiente una llamada a este comando que no 10 llamadas al comando

```
glArrayElement();
```

- La forma general del comando es

```
glDrawElements(primitive, 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 vértices y colores, 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.

Vertex arrays sin índices

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

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

  y se toman los elementos de los vertex arrays activos, tal como marque la primitiva y secuencialmente.

- La forma general de este comando es

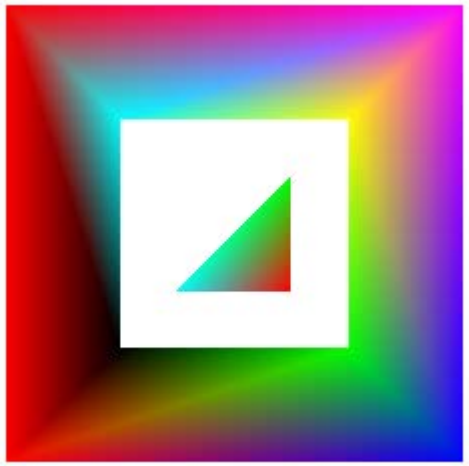
```
glDrawArrays(primitive, first, count);
```

  que dibuja con **`primitive`**, usando **`count`** elementos de los arrays de vértices y colores, empezando en la posición **`first`**.

Ejemplo con vertex arrays e índices

- Se quiere proporcionar la información del cuadrado anular mediante índices y la del triángulo interior mediante vertex arrays. Además, esta última se proporciona entremezclando información de coordenadas de vértice con su color
  - Información del triángulo interior

```
float vertices2AndColors2Intertwined[] = {
  40.0, 40.0, 0.0,
  0.0, 1.0, 1.0,
  60.0, 40.0, 0.0,
  1.0, 0.0, 0.0,
  60.0, 60.0, 0.0,
  0.0, 1.0, 0.0
};
```



Ejemplo

- Se activan los vertex arrays

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
```

- Se especifica dónde están y cómo son los datos del cuadrado anular

```
glVertexPointer(3, GL_FLOAT, 0, vertices);
glColorPointer(3, GL_FLOAT, 0, colors);
```

- Se dibuja el cuadrado anular

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

Ejemplo

- Se especifica dónde están y cuál es el formato de los datos del triángulo interior. Observa que, como los datos de coordenadas y colores están mezclados, hay 6 floats de distancia entre coordenadas y entre colores

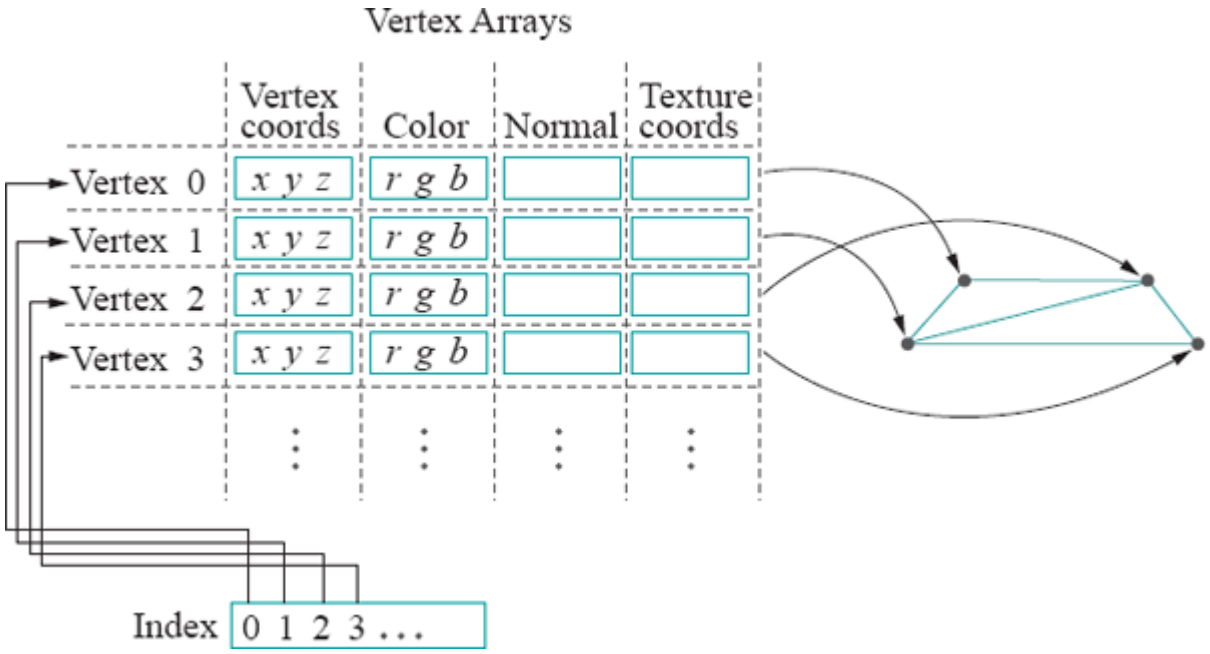
```
glVertexPointer(3, GL_FLOAT, 6 * sizeof(float),
        &vertices2AndColors2Intertwined[0]);

glColorPointer(3, GL_FLOAT, 6 * sizeof(float),
        &vertices2AndColors2Intertwined[3]);
```

- Se dibuja el triángulo interior

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Vertex arrays



Lighting equation in OpenGL

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

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

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

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

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

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

Normales

- Cada componente del array de normales es un vector normal.
  - Tantos vectores normales como vértices.
  - Cada vector normal:
    - constituye un atributo más del vértice
    - es perpendicular a la superficie en ese vértice (i.e., producto escalar igual a 0 con el vector tangente)
    - apunta hacia el exterior del objeto
    - debe estar normalizado.
      - Ojo con el comando **`glEnable(GL_NORMALIZE);`**
  - Vector normal y sombreado del objeto (shading model).

Normales

- Se sigue el convenio de proporcionar los vértices de una cara en sentido anti-horario (CCW), cuando la cara se mira desde el exterior del objeto.
- El sentido de los vértices permite a OpenGL identificar las caras frontales (**GL_FRONT**) y las traseras (**GL_BACK**).
- El comando

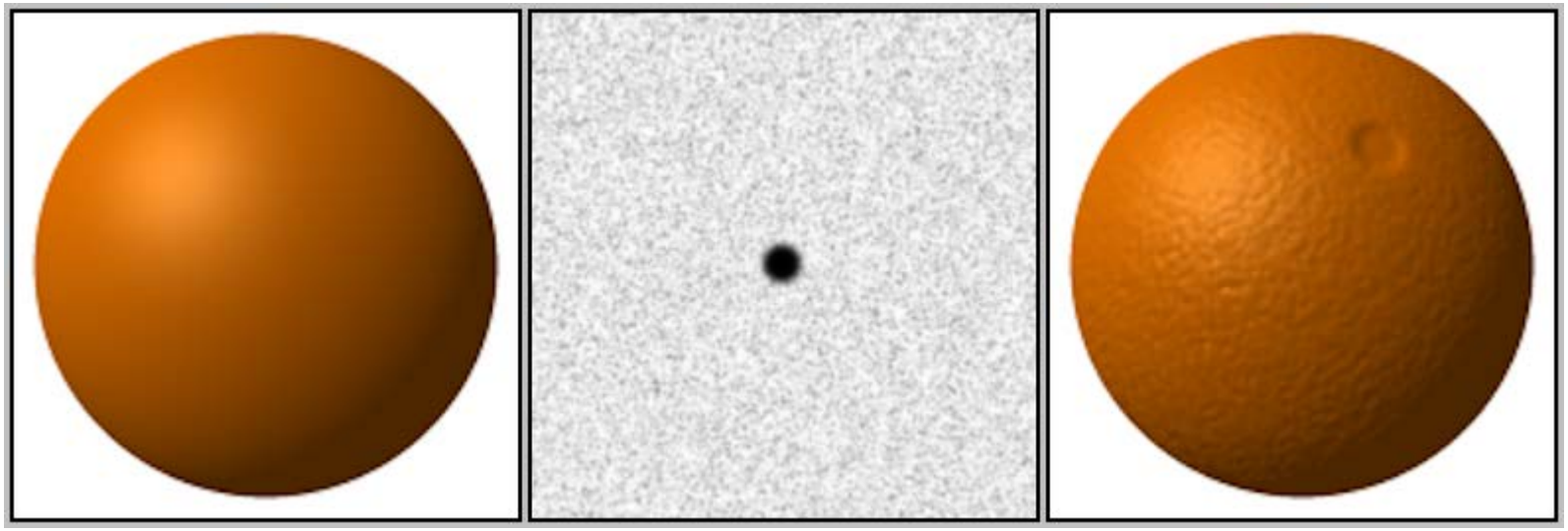
```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

  colorea caras traseras, invirtiendo el sentido de los vectores normales.

Cálculo de vectores normales

- Renderizado de objetos con caras poligonales bien definidas (por ejemplo, prismas): se calcula un vector normal por cara y se usa como vector normal para cada vértice de la misma
- **Como hay tantos vectores normales como vértices, el vector normal de un vértice se calcula 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
- Renderizado de objetos con superficies curvas (por ejemplo, esferas, cilindros, etc.): el vector normal de un vértice se calcula a partir de las ecuaciones (paramétricas o implícita) de la superficie
- Cuando se usan ciertas técnicas (*bump mapping*; aspecto de piel de naranja) se pueden especificar vectores normales por fragmento.

Bump mapping



- Permite dar cierto aspecto (por ejemplo, rugosidad) sin cambiar la geometría del objeto
- Las normales se realinean siguiendo un cierto patrón
- James Blinn

Cálculo del vector normal a una cara

- Producto vectorial
- Formas de calcular el vector normal a una cara formada por los vértices {v0, v1, …, vn}
  - Usando el producto vectorial:

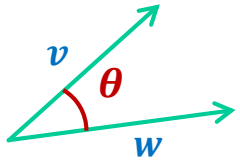
    n=(v2-v1)x(v0-v1) (ó (v1-v0)x(vn-v0))

    Inconvenientes
  - Usando el método de Newell

- v y w son dos vectores en 3D

$$v \times w = \begin{pmatrix} v_y w_z - v_z w_y \\ v_z w_x - v_x w_z \\ v_x w_y - v_y w_x \end{pmatrix}$$

- $v \times w$ es un vector perpendicular a v y w, el sentido lo da la regla de la mano derecha
- Propiedades algebraicas y geométricas
  - $v \times w = -(w \times v)$
  - $v \times (w + u) = v \times w + v \times u$
  - $k(v \times w) = (kv) \times w$
  - $\|v \times w\| = \|v\| \, \|w\| \, |sin\, \theta|$
  - $\|v \times w\| = 0 \iff v$ y w son paralelos



$0 \leq \theta \leq 180°$

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);
```

Superficies con ecuaciones conocidas

- Ecuación de un plano:
  - Forma paramétrica: $P(u, v) = C+u*\mathbf{a}+v*\mathbf{b}$

  donde C es un punto del plano, y **a**, **b** son vectores del plano linealmente independientes
  - Forma implícita: $F(x, y, z)$: $a*x + b*y + c*z + d= 0$

  El vector **n** = (a, b, c) es normal al plano.
- Cuando la superficie está dada en forma paramétrica, el producto vectorial de las derivadas parciales con respecto a los parámetros $(\partial P/\partial u)\times(\partial P/\partial v)$ es un vector normal a la superficie.
- Cuando la superficie está dada en forma implícita, el gradiente de la ecuación $\nabla F=(\partial F/\partial x, \partial F/\partial y, \partial F/\partial z)$ es un vector normal a la superficie.

- Esfera de radio R, centrada en el origen
  - Forma paramétrica:

    $P(\phi, \theta) = (R*sen(\phi)*cos(\theta), R*sen(\phi)*sen(\theta), R*cos(\phi))$

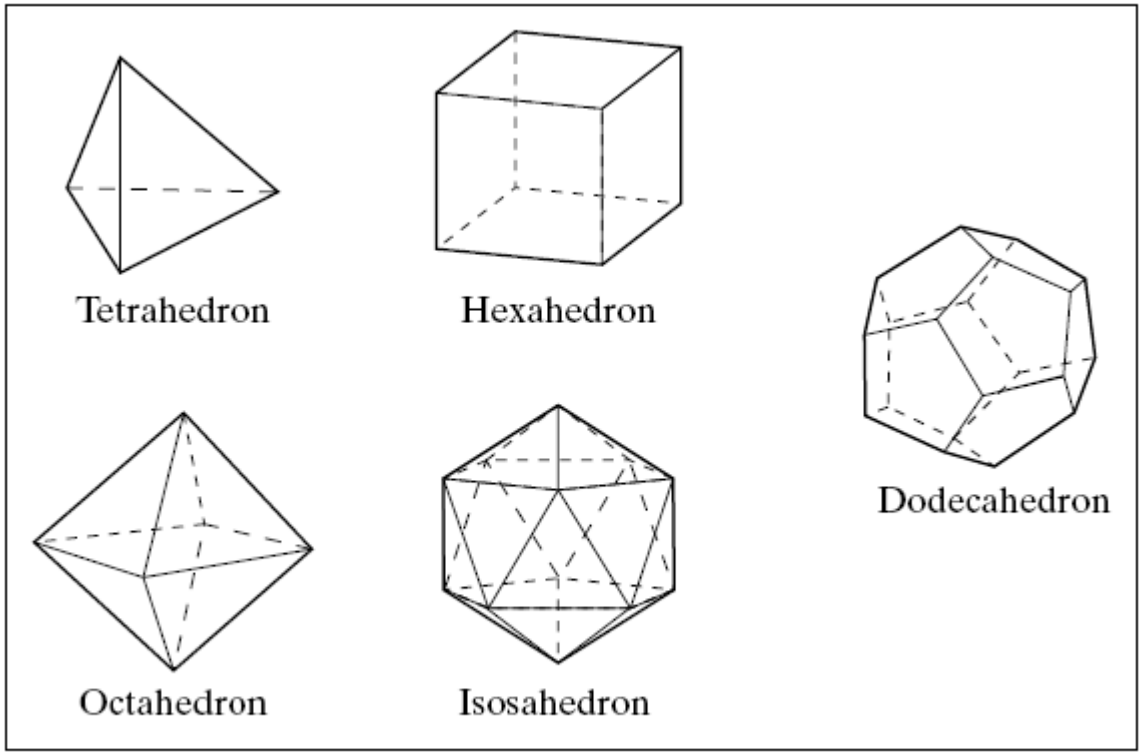
    donde $-\pi/2 \leq \phi < \pi/2$ (latitud), $0 \leq \theta < 2*\pi$ (longitud)
  - Forma implícita: $x*x + y*y + z*z = R*R$
  - Vector normal sin normalizar:

    $P(\phi, \theta) = (R*R*sen(\phi)*sen(\phi)*cos(\theta), R*R*sen(\phi)*sen(\phi)*sen(\theta), R*R*sen(\phi)*cos(\phi))$
- Cilindro de radio R y altura 2
  - Forma paramétrica:

    $P(\phi, \theta) = (R*cos(\phi), R*sen(\phi), \theta)$

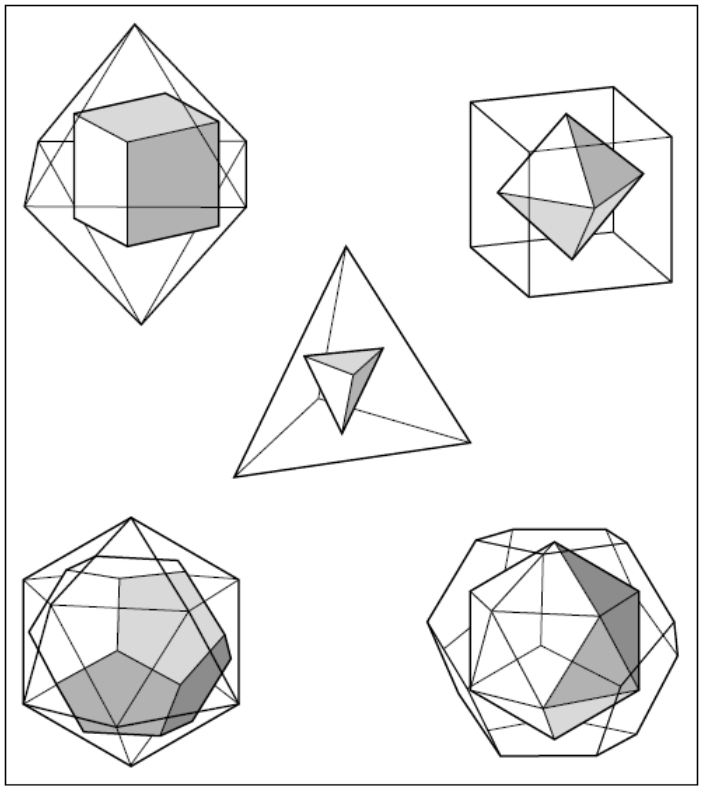
    donde $-\pi \leq \phi \leq \pi$, $-1 \leq \theta \leq 1$
  - Vector normal sin normalizar: $(R*cos(\phi), R*sen(\phi), 0)$

Sólidos platónicos

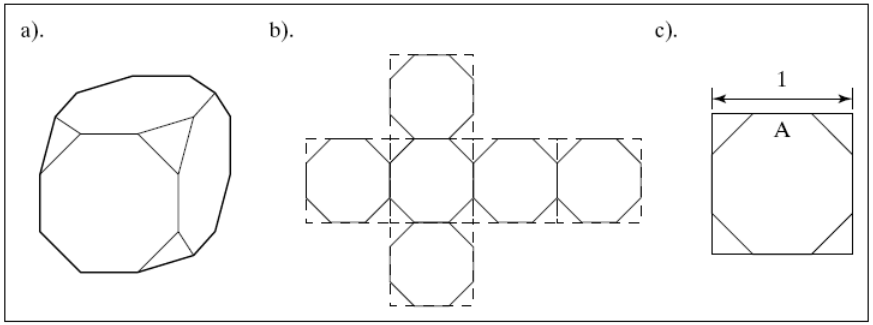


Sólidos platónicos y dualidad

Solid	V	F	E	Schläfli
Tetrahedron	4	4	6	(3, 3)
Hexahedron	8	6	12	(4, 3)
Octahedron	6	8	12	(3, 4)
Icosahedron	12	20	30	(3, 5)
Dodecahedron	20	12	30	(5, 3)



- Sólidos por truncamiento. Cubo truncado

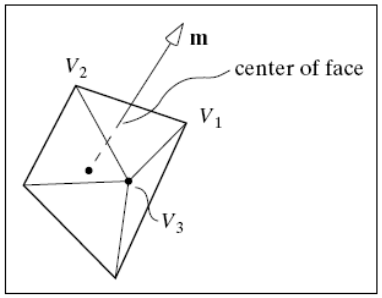


- Vértices obtenidos al truncar la arista CD

$$V=((1+A)/2)\ C + ((1-A)/2)\ D$$

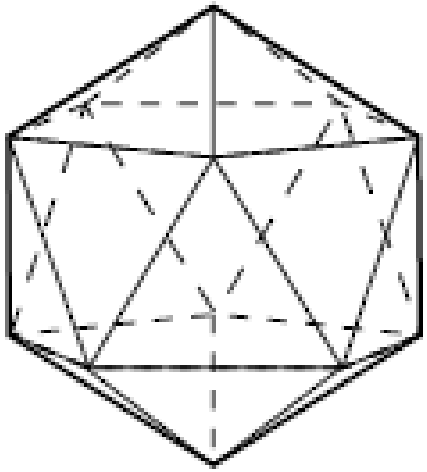
$$W=((1-A)/2)\ C + ((1+A)/2)\ D$$

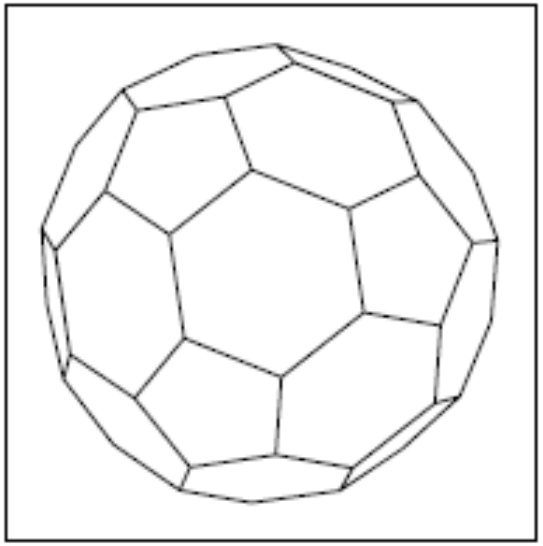
- Normales
  - **m**=(V1+V2+V3)/3

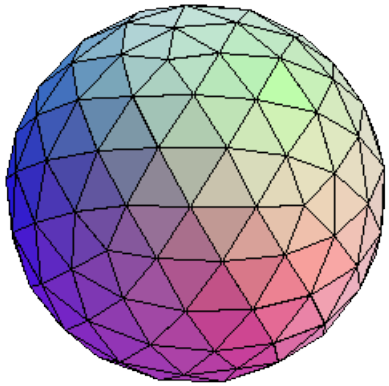


Icosaedro truncado

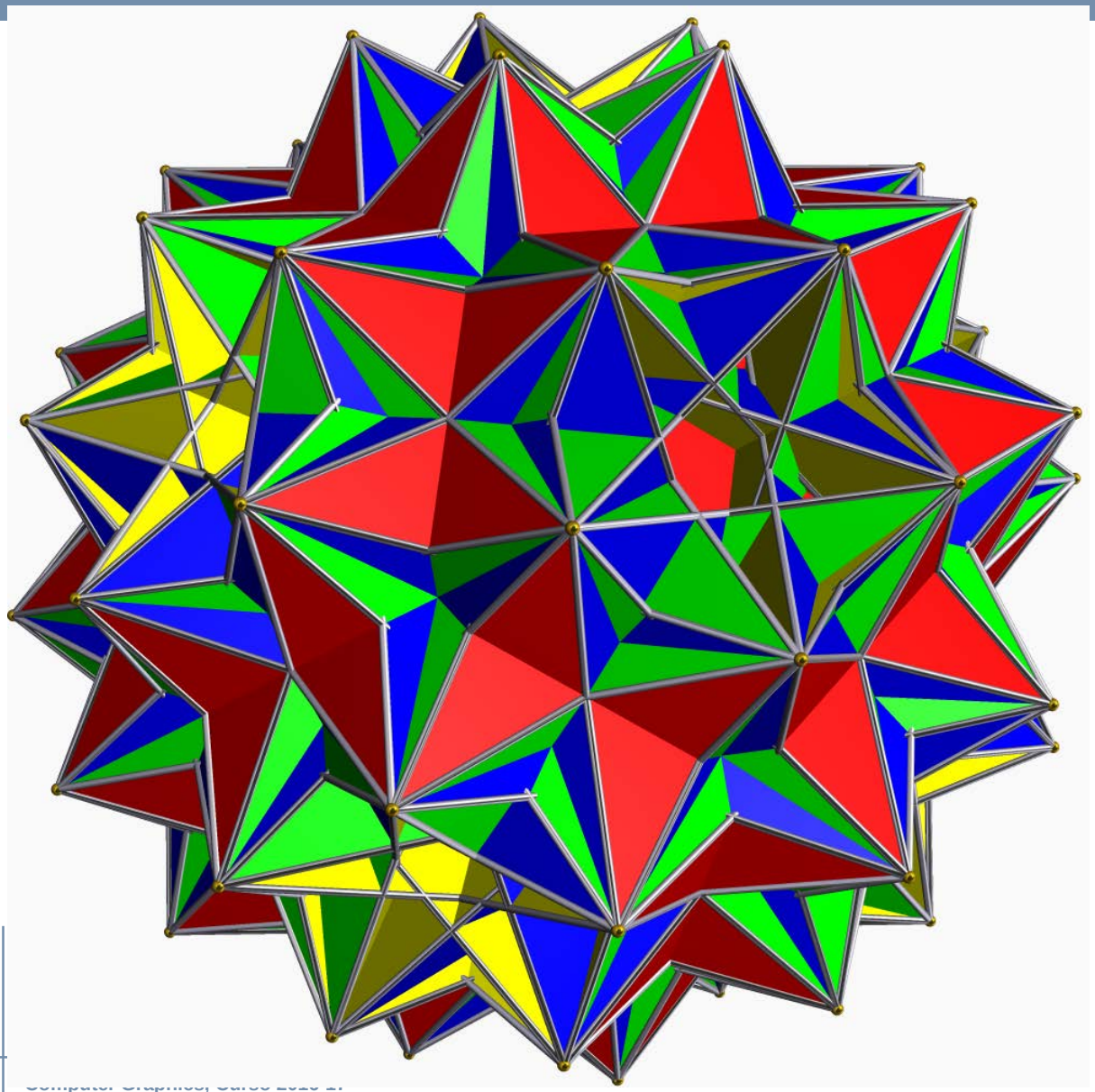
- Buckyball (fulereno), por Buckminster Fuller





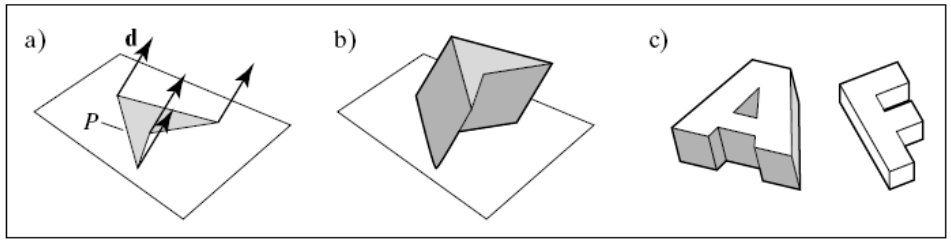


Poliedros de Skilling

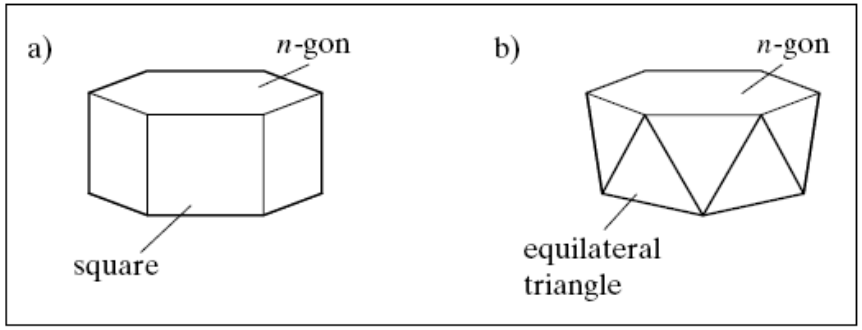


Prismas y antiprismas

- Extrudir (tr): Dar forma a una masa metálica, plástica, etc., haciéndola salir por una abertura especialmente dispuesta.
- Un prisma es un poliedro obtenido mediante extrusión de un polígono a lo largo de una línea recta.
  - Caras cuadrangulares



- Un antiprisma se obtiene a partir de un prisma, rotando la cara n-gonal superior, 180/n grados.
  - Caras triangulares
  - El octaedro es un antiprisma (Why?)



Malla para un prisma recto

- Vértices:

  (xi, yi, 0), (xi, yi, H), 0≤i≤N-1

  donde N es el número de lados del polígono que origina el prisma y H es su altura

- Caras:
  - Laterales

    (j, suc(j), suc(j)+N, j+N), 0≤j≤N-1
  - Base (0, 6, 5, 4, 3, 2, 1)
  - Tapa (7, 8, 9, 10, 11, 12,13)
- Ejemplo: Flecha de base heptagonal (N=7)