

# Representações de Objetos

# Objetivos

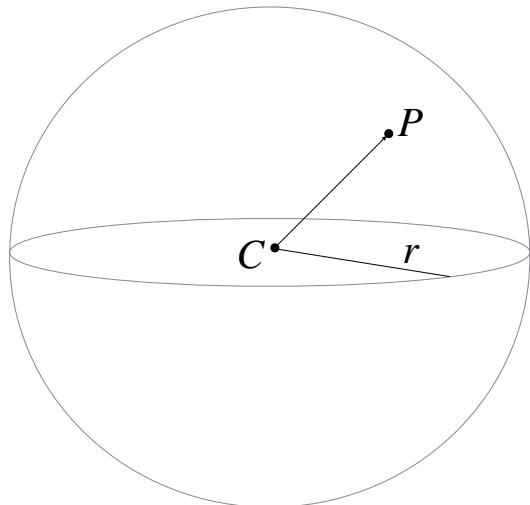
- Compreender as diversas formas para representar a geometria:
  - Para que objectos?
  - Para que operações?
- Perceber quais as vantagens e desvantagens de cada representação
- Estruturas de dados simples para representar malhas de polígonos

# Representações de objetos

- Que objetos?
  - Sólidos?  
(esfera, cubo, cone, torus, ...)
  - Superfícies planas?  
(plano, polígono, disco...)
  - Superfícies curvas ?  
(parabolóide, hiperbolóide, superfícies paramétricas bi-cúbicas, nurbs, ...)
  - Deformáveis ou maleáveis?  
(líquidos, fumo, cabelo, tecidos, ...)
- Que operações?
  - Visualizar usando um sistema baseado em “rasterização”?
  - Visualizar usando um ray-tracer?
  - Calcular características dos objetos tais como volumes, áreas, etc. ?
  - Efetuar operações de modelação usando Intersecção, diferença ou união de objetos ?
  - Distinguir entre o interior, o exterior e a superfície ?

# Representações algébricas

Exemplo para uma esfera



Considere uma esfera centrada em  $C = (c_x, c_y, c_z)$ , com raio  $r$ .

Um ponto  $P = (x, y, z)$ , poderá estar:

- No interior da esfera
- No exterior da esfera
- Na superfície da esfera

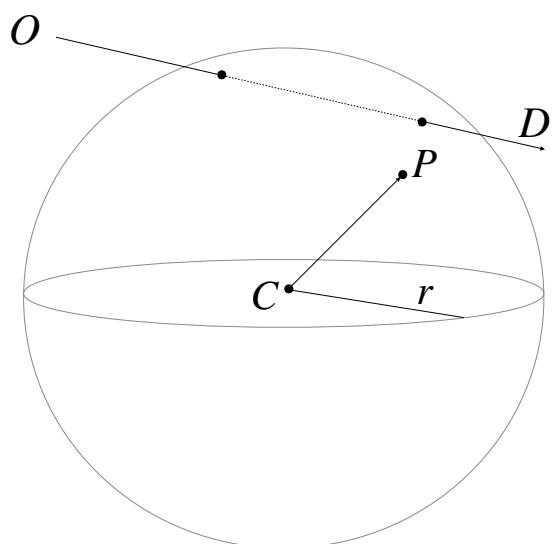
$$\sqrt{(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2} = r$$

<  
>

$$|P - C| = r \iff |P - C|^2 = r^2$$

# Representações algébricas

Exemplo para uma esfera



$$O = (o_x, o_y, o_z), C = (c_x, c_y, c_z), D = (d_x, d_y, d_z)$$

Apropriada para *ray tracing* para calcular a interseção de raios com a superfície do objeto, determinando assim os pontos de entrada e de saída.

$$R(t) = O + tD \quad (1)$$

$$|P - C|^2 = r^2 \quad (2)$$

Substituindo (1) em (2), no lugar de  $P$ :

$$(o_x + td_x - c_x)^2 + (o_y + td_y - c_y)^2 + (o_z + td_z - c_z)^2 - r^2 = 0$$

Desenvolvendo e agrupando, para ficar na forma  $At^2 + Bt + C = 0$ , obtém-se:

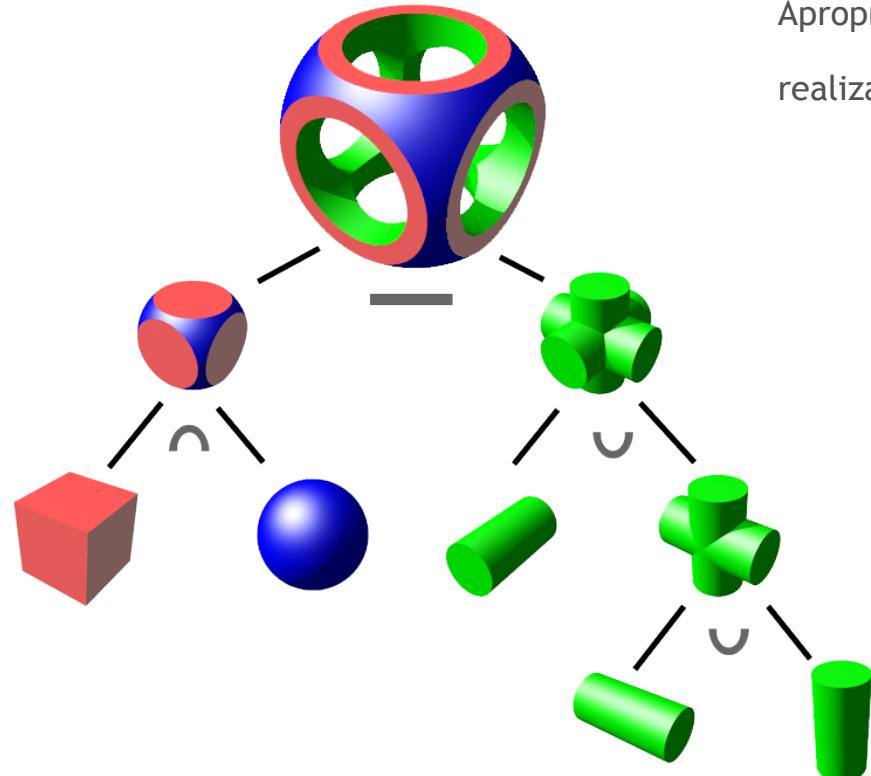
$$A = d_x^2 + d_y^2 + d_z^2$$

$$B = 2(o_x d_x + o_y d_y + o_z d_z - c_x d_x - c_y d_y - c_z d_z)$$

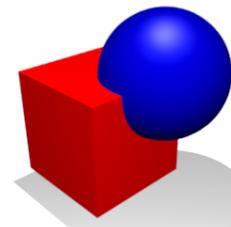
$$C = o_x^2 + o_y^2 + o_z^2 + c_x^2 + c_y^2 + c_z^2 - 2(o_x c_x + o_y c_y + o_z c_z) - r^2$$

Há 0, 1 ou 2 soluções. Para calcular  $R(t)$ , basta substituir as soluções.

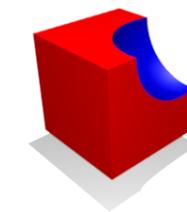
# Representações algébricas



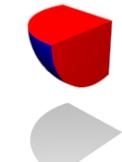
Apropriada para Geometria Sólida Construtiva (Constructive Solid Geometry), realizando operações entre conjuntos:



União



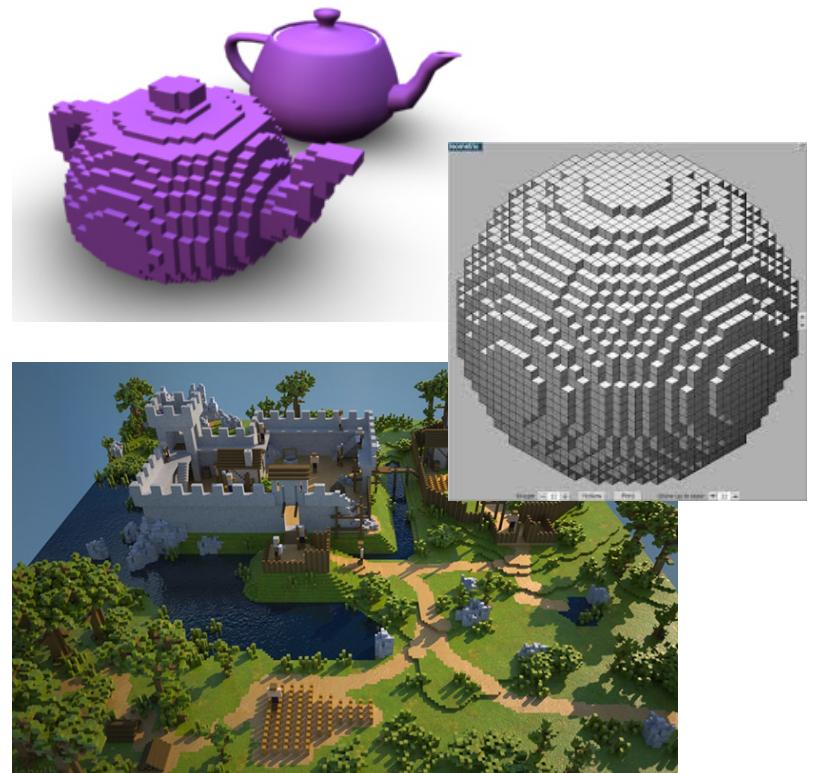
Diferença



Interseção

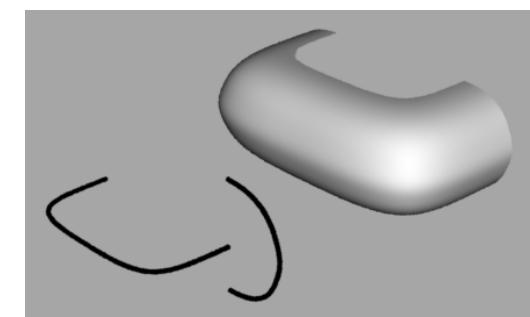
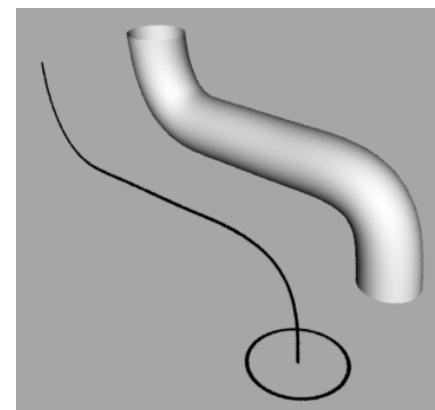
# Enumeração da ocupação espacial

- Um objeto é uma lista de voxels (3D pixels)
- Cada voxel pode ser ocupado (pertencer a um determinado objeto) ou estar vazio
- Apropriada para aplicações de Tomografia Axial Computorizada (CAT) ou de Imagiologa por Ressonância Magnética (MRI)
- **Vantagens:**
  - Teste de adjacência
  - Teste de interior/exterior
  - Operações de Geometria Sólida Construtiva
- **Desvantagens:**
  - Não permite ocupação parcial
  - Precisão limitada pela resolução da grelha
  - Tamanho da estrutura de dados



# Representações por varrimento

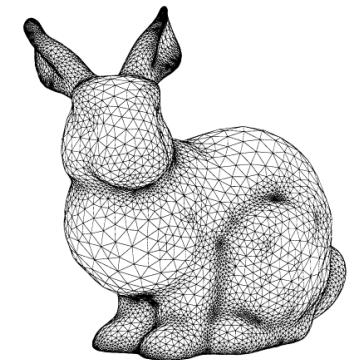
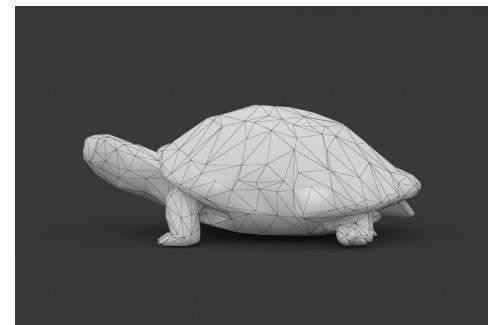
- Resultam do varrimento dum objeto ao longo duma trajetória
- Os varrimentos mais comuns são obtidos por rotação ou por uma sequência de translações.
- Durante o varrimento poderão aplicar-se mudanças de escala ao objeto.
- **Vantagens:**
  - Cálculo de Volumes ou de áreas
  - Apropriado para determinadas aplicações de manufatura assistida por computador (CAM)
- **Desvantagens:**
  - Não se adequa a operações de Geometria Sólida Construtiva

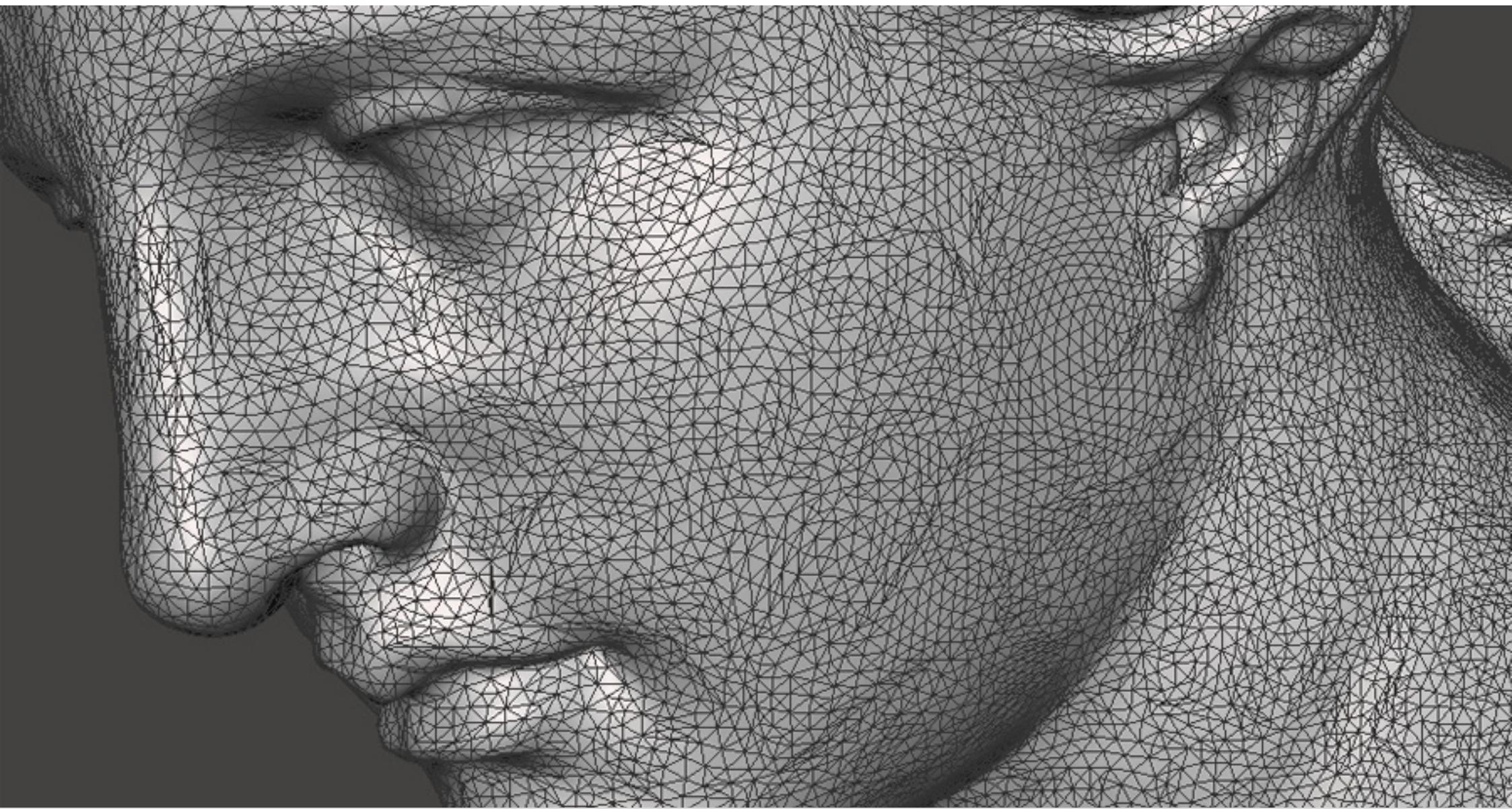


\* from <http://ayam.sourceforge.net>

# Representações pela fronteira (B-Reps)

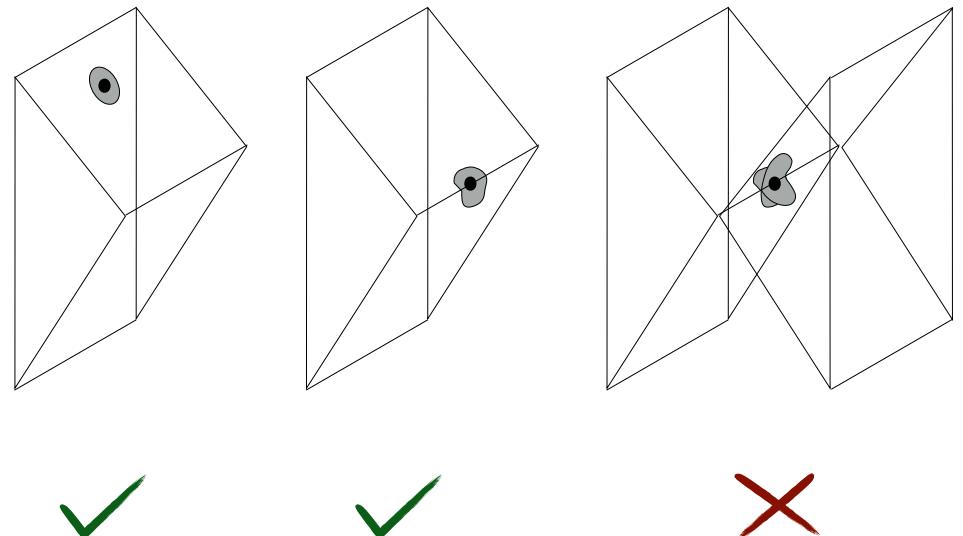
- Os objetos são descritos em termos das fronteiras da sua superfície: vértices, arestas e faces
- As superfícies curvas são possíveis, mas acabam sendo aproximadas por polígonos (resultantes de retalhos bi-cúbicos ou nurbs)
- Os polígonos convexos são o tipo mais comum de face encontrada em B-Reps
- Alguns sistemas apenas permitem o tipo mais simples de polígono (o triângulo) sendo necessário decompor polígonos mais complexos.
- Muitos sistemas que lidam com esta representação limitam o seu uso a objetos 2-manifold.





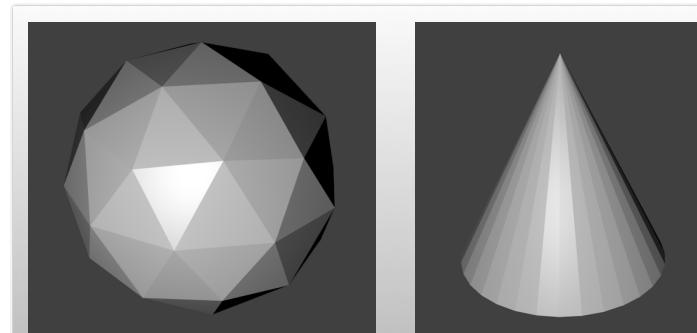
# Representações pela fronteira (B-Reps)

- Objetos **2-Manifold** são objetos em que todos os seus pontos contêm uma fronteira arbitrária que se assemelha, do ponto de vista da sua topologia, a um disco.
- Existe um mapeamento contínuo de 1:1 entre os pontos dessa fronteira e um disco.

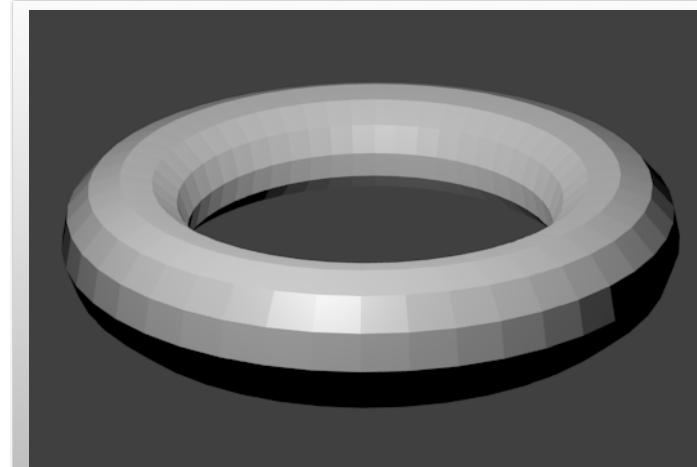


# Representações pela fronteira (B-Reps)

- **Poliedro:** sólido delimitado por um conjunto de polígonos cujas arestas pertencem a um número par de polígonos (2 para 2-manifolds)
- Os poliedros possuem faces planas, arestas rectilíneas e vértices “aguçados”.
- Um poliedro sem buracos pode ser deformado numa esfera.



Poliedros  
sem  
buracos

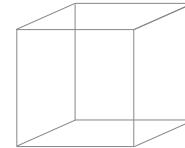


Poliedro  
com  
buraco

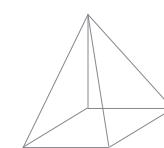
# Representações pela fronteira (B-Reps)

- A fórmula de Euler para poliedros é **necessária**, mas **não suficiente** para que um objeto possa ser um poliedro.
- Outros requisitos:
  - Cada aresta precisa estar ligada a dois vértices e ser partilhada exatamente por duas faces
  - Pelo menos 3 arestas deverão encontrar-se em cada vértice
  - As faces não se poderão atravessar umas às outras.

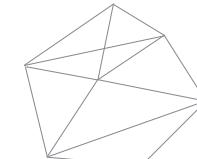
$$\text{Fórmula de Euler: } V - E + F = 2$$



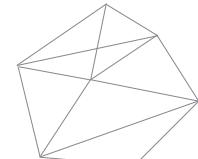
$$\begin{array}{l} V = 8 \\ E = 12 \\ F = 6 \end{array}$$



$$\begin{array}{l} V = 5 \\ E = 8 \\ F = 5 \end{array}$$



$$\begin{array}{l} V = 7 \\ E = 13 \\ F = 8 \end{array}$$

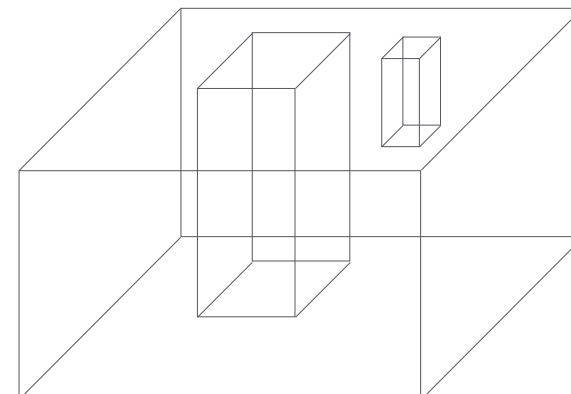


$$\begin{array}{l} V = 7 \\ E = 14 \\ F = 9 \end{array}$$

# Representações pela fronteira (B-Reps)

- A fórmula de Euler para poliedros é **necessária**, mas **não suficiente** para que um objeto possa ser um poliedro.
- Outros requisitos:
  - Cada aresta precisa estar ligada a dois vértices e ser partilhada exatamente por duas faces
  - Pelo menos 3 arestas deverão encontrar-se em cada vértice
  - As faces não se poderão atravessar umas às outras.

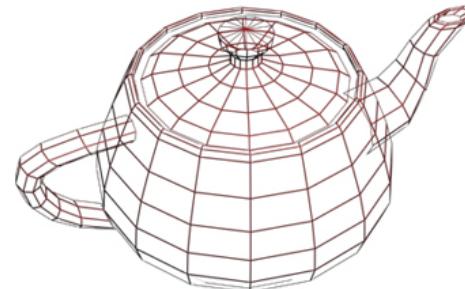
Fórmula de Euler para 2-manifolds com buracos:  $V - E + F - H = 2(C - G)$



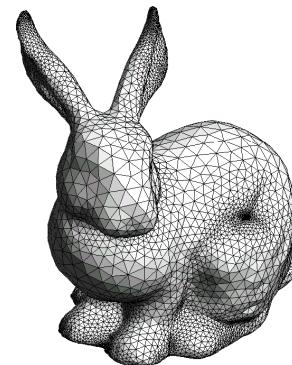
$$\begin{aligned}V &= 24 \\E &= 36 \\F &= 15 \\H &= 3 \\C &= 1 \\G &= 1\end{aligned}$$

# Estruturas de dados para malhas

- Para descrever uma malha é necessário:
  - Tabela de vértices - Localização de todos os vértices
  - Todas as arestas que ligam os vértices (definidas implicita ou explicitamente)
  - Todas as faces que constituem o modelo (definidas através das arestas ou dos vértices)



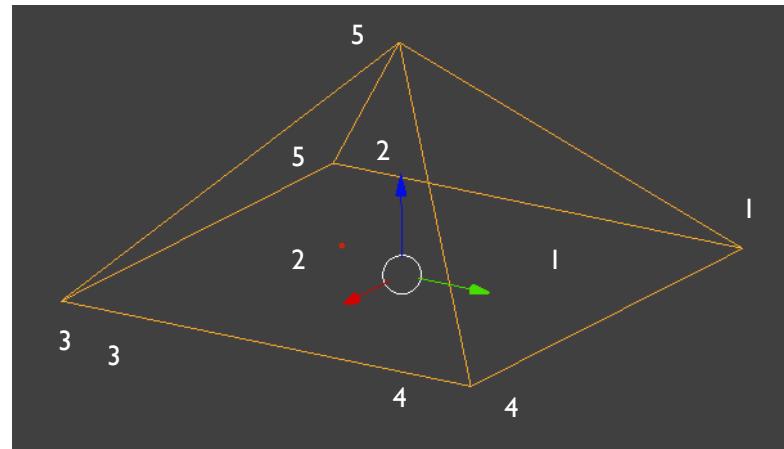
Original teapot by Martin Newell 1975



Original Stanford bunny by Greg Turk and Marc Levoy 1994

# Estruturas de dados para malhas

- Exemplo duma **pirâmide com arestas explícitas**
- As faces são definidas à custa das arestas e as arestas à custa dos vértices. Cada vértice tem duas formas distintas de ser alcançado, a partir duma face.
- Esta representação não impede ter arestas pendentes (arestas que não pertencem a qualquer face)
- De igual forma, vértices isolados são igualmente possíveis.

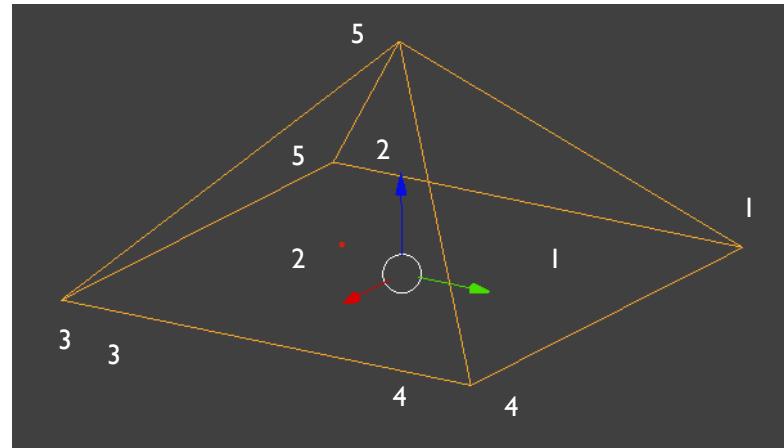


| Topologia |         | Arestas  | Geometria   |
|-----------|---------|----------|-------------|
| Faces     |         |          | Vértices    |
| 1         | 5, 8, 4 | 1 (1, 2) | 1 (1,0,-1)  |
| 2         | 6, 1, 5 | 2 (2,3)  | 2 (-1,0,-1) |
| 3         | 6, 2, 7 | 3 (3,4)  | 3 (-1,0,1)  |
| 4         | 7,3,8   | 4 (4,1)  | 4 (1,0,1)   |
| 5         | 4,3,2,1 | 5 (1,5)  | 5 (0,1,0)   |
| 6         |         | 6 (5,2)  | 6 (1,1,1)   |
| 7         |         | 7 (5,3)  |             |
| 8         |         | 8 (5,4)  |             |

# Estruturas de dados para malhas

- Exemplo duma **pirâmide com arestas implícitas**
- As faces são definidas à custa dos vértices.
- Não há representação explícita das arestas do modelo.
- Esta representação **impede** a existência de arestas pendentes (arestas que não pertencem a qualquer face)
- Contudo, vértices isolados são igualmente possíveis.

Arestas: (1,4); (4,3); (3,2); (2,1)



| Topologia |            |
|-----------|------------|
|           | Faces      |
| 1         | 1, 5, 4    |
| 2         | 5, 2, 1    |
| 3         | 5, 2, 3    |
| 4         | 5, 3, 4    |
| 5         | 1, 4, 3, 2 |

| Geometria |           |
|-----------|-----------|
|           | Vertices  |
| 1         | (1,0,-1)  |
| 2         | (-1,0,-1) |
| 3         | (-1,0,1)  |
| 4         | (1,0,1)   |
| 5         | (0,1,0)   |
| 6         | (1,1,1)   |

# Estruturas de dados para malhas

- O armazenamento de faces com um número arbitrário de vértices não é muito “computer friendly”
- As faces com mais dos que 3 vértices podem apresentar problemas de ambiguidade (não planares, não convexas)
- Cada face pode ser subdividida em triângulos (s o o sempre planos e convexos!)
- O hardware suporta diretamente o desenho de tri ngulos



| Faces |            |
|-------|------------|
| 1     | 1, 5, 4    |
| 2     | 5, 2, 1    |
| 3     | 5, 2, 3    |
| 4     | 5, 3, 4    |
| 5     | 1, 4, 3, 2 |

Malha de tri ngulos

| Faces |         |
|-------|---------|
| 1     | 1, 5, 4 |
| 2     | 5, 2, 1 |
| 3     | 5, 2, 3 |
| 4     | 5, 3, 4 |
| 5     | 1, 4, 3 |
| 6     | 1, 3, 2 |

# Malhas de triângulos em WebGL

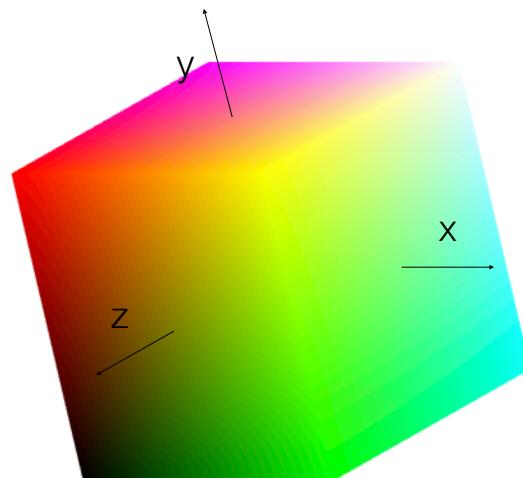
# Objetivos

- Ilustrar a construção dum objeto modelado como um B-rep, usando WebGL
- Analisar duas formas de indicação da geometria:
  - Usando arrays
  - Usando elementos (índices)

# Modelação dum cubo

- Um array com os vértices:

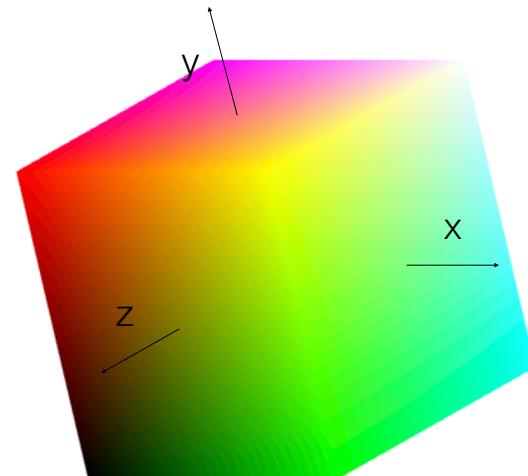
```
var vertices = [  
    vec4( -0.5, -0.5,  0.5, 1.0 ),  
    vec4( -0.5,  0.5,  0.5, 1.0 ),  
    vec4(  0.5,  0.5,  0.5, 1.0 ),  
    vec4(  0.5, -0.5,  0.5, 1.0 ),  
    vec4( -0.5, -0.5, -0.5, 1.0 ),  
    vec4( -0.5,  0.5, -0.5, 1.0 ),  
    vec4(  0.5,  0.5, -0.5, 1.0 ),  
    vec4(  0.5, -0.5, -0.5, 1.0 )  
];
```



# Modelação dum cubo

- Um array com as cores:

```
var vertexColors = [  
    [ 0.0, 0.0, 0.0, 1.0 ], // black  
    [ 1.0, 0.0, 0.0, 1.0 ], // red  
    [ 1.0, 1.0, 0.0, 1.0 ], // yellow  
    [ 0.0, 1.0, 0.0, 1.0 ], // green  
    [ 0.0, 0.0, 1.0, 1.0 ], // blue  
    [ 1.0, 0.0, 1.0, 1.0 ], // magenta  
    [ 1.0, 1.0, 1.0, 1.0 ], // white  
    [ 0.0, 1.0, 1.0, 1.0 ] // cyan  
];
```

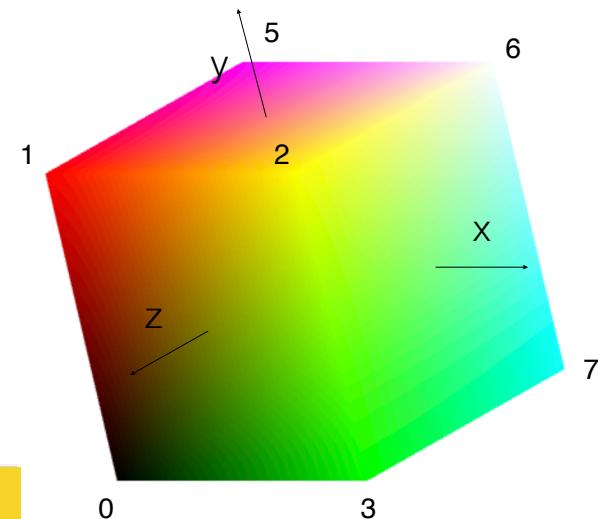


# Modelação dum cubo

- Construção das faces:

```
function colorCube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

Os vértices estão ordenados por forma a obter normais que apontam para fora do objeto. Cada chamada de **quad()** gera 2 triângulos.



# Modelação dum cubo

- Inicialização:

```
var canvas, gl;
var numVertices = 36;
var points = [];
var colors = [];

window.onload = function init(){
    canvas = document.getElementById( "gl-canvas" );
    gl = WebGLUtils.setupWebGL( canvas );

    colorCube();

    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 1.0, 1.0, 1.0 );
    gl.enable(gl.DEPTH_TEST);
```

Necessário para ativação do Z-Buffer  
(Hidden Surface Removal)

# Modelação dum cubo

- Preencher os arrays `points` e `colors` com a os dados em `vertices` e `vertexColors` para os 2 triângulos do quad:

```
function quad(a, b, c, d)
{
    var indices = [ a, b, c, c, d, a ];
    for ( var i = 0; i < indices.length; ++i ) {

        points.push( vertices[indices[i]] );
        colors.push( vertexColors[indices[i]] );

        // for solid colored faces use
        //colors.push(vertexColors[a]);
    }
}
```

Por cada chamada `quad()` é escrita a informação de 6 vértices nos respetivos buffers

# Modelação dum cubo

- Buffer com o atributo cor:

O array “inflacionado” com as cores produzidas por `quad()`

```
var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW );

var vColor = gl.getAttribLocation( program, "vColor" );
gl.vertexAttribPointer( vColor, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vColor );
```

# Modelação dum cubo

- Buffer com o atributo posição:

O array “inflacionado” com as posições produzidas por `quad()`

```
var vBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );

var vPosition = gl.getAttribLocation( program, "vPosition" );
gl.vertexAttribPointer( vPosition, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );
```

# Modelação dum cubo

- Função de desenho:

Limpa o conteúdo do Z-Buffer  
(Necessário para Hidden Surface Removal)

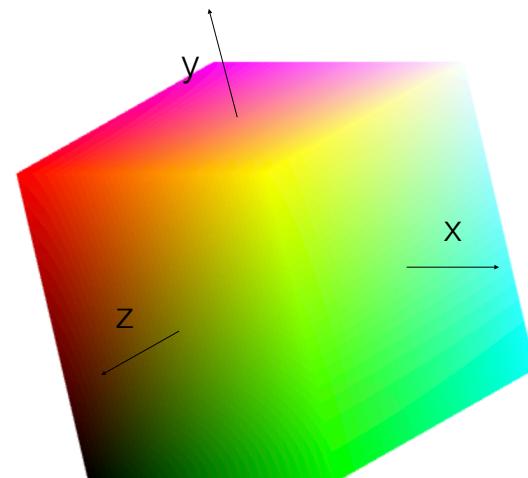
```
function render(){  
    gl.clear( gl.COLOR_BUFFER_BIT |gl.DEPTH_BUFFER_BIT );  
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );  
    requestAnimFrame( render );  
}
```

# Uma alternativa melhor...

## Modelação dum cubo (II)

- Um array com os vértices:

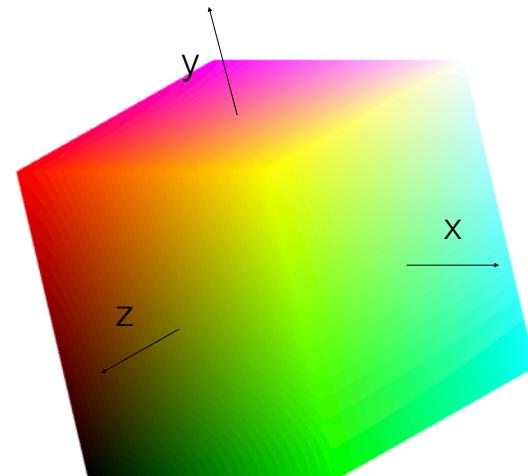
```
var vertices = [  
    vec4( -0.5, -0.5,  0.5, 1.0 ),  
    vec4( -0.5,  0.5,  0.5, 1.0 ),  
    vec4(  0.5,  0.5,  0.5, 1.0 ),  
    vec4(  0.5, -0.5,  0.5, 1.0 ),  
    vec4( -0.5, -0.5, -0.5, 1.0 ),  
    vec4( -0.5,  0.5, -0.5, 1.0 ),  
    vec4(  0.5,  0.5, -0.5, 1.0 ),  
    vec4(  0.5, -0.5, -0.5, 1.0 )  
];
```



## Modelação dum cubo (II)

- Um array com as cores:

```
var vertexColors = [  
    [ 0.0, 0.0, 0.0, 1.0 ], // black  
    [ 1.0, 0.0, 0.0, 1.0 ], // red  
    [ 1.0, 1.0, 0.0, 1.0 ], // yellow  
    [ 0.0, 1.0, 0.0, 1.0 ], // green  
    [ 0.0, 0.0, 1.0, 1.0 ], // blue  
    [ 1.0, 0.0, 1.0, 1.0 ], // magenta  
    [ 1.0, 1.0, 1.0, 1.0 ], // white  
    [ 0.0, 1.0, 1.0, 1.0 ] // cyan  
];
```



## Modelação dum cubo (II)

- Usar índices para os arrays `vertices`/`vertexColors` para descrever as faces:

```
var indices = [  
    1, 0, 3,  
    3, 2, 1,  
    2, 3, 7,  
    7, 6, 2,  
    3, 0, 4,  
    4, 7, 3,  
    6, 5, 1,  
    1, 2, 6,  
    4, 5, 6,  
    6, 7, 4,  
    5, 4, 0,  
    0, 1, 5  
];
```

As 12 faces triangulares que irão formar o cubo são guardadas como índices numa lista de vértices.

Os índices precisam ser enviados para o GPU e a chamada de desenho necessita ser ajustada para operar sobre índices em vez de arrays de vértices.

## Modelação dum cubo (II)

- Desenho com `drawElements()`:

Enviar os índices para o GPU

```
var iBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, iBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
              new Uint8Array(indices),
              gl.STATIC_DRAW);
```

- Ainda poderá ser mais eficiente se se usarem *triangle strips* ou *triangle fans*.

Desenhar com índices

```
gl.drawElements( gl.TRIANGLES, numVertices, gl.UNSIGNED_BYTE, 0 );
```

# Modelação dum cubo (II)

- Inicialização (cores):

O array original com as 8 cores

```
var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(vertexColors), gl.STATIC_DRAW );

var vColor = gl.getAttribLocation( program, "vColor" );
gl.vertexAttribPointer( vColor, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vColor );
```

# Modelação dum cubo (II)

- Inicialização (posição)

O array original com as 8 posições

```
var vBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW );

var vPosition = gl.getAttribLocation( program, "vPosition" );
gl.vertexAttribPointer( vPosition, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );
```

# Modelação dum cubo (II)

- Função de desenho

```
function render(){
    gl.clear( gl.COLOR_BUFFER_BIT |gl.DEPTH_BUFFER_BIT );
    theta[axis] += 2.0;
    gl.uniform3fv(thetaLoc, theta);
    gl.drawElements( gl.TRIANGLES, numVertices, gl.UNSIGNED_BYTE, 0 );
    requestAnimFrame( render );
}
```

O cubo apenas tinha 8 vértices, por isso um byte chega para os indexar (256 índices distintos). Para modelos maiores, usa-se gl.UNSIGNED\_SHORT