

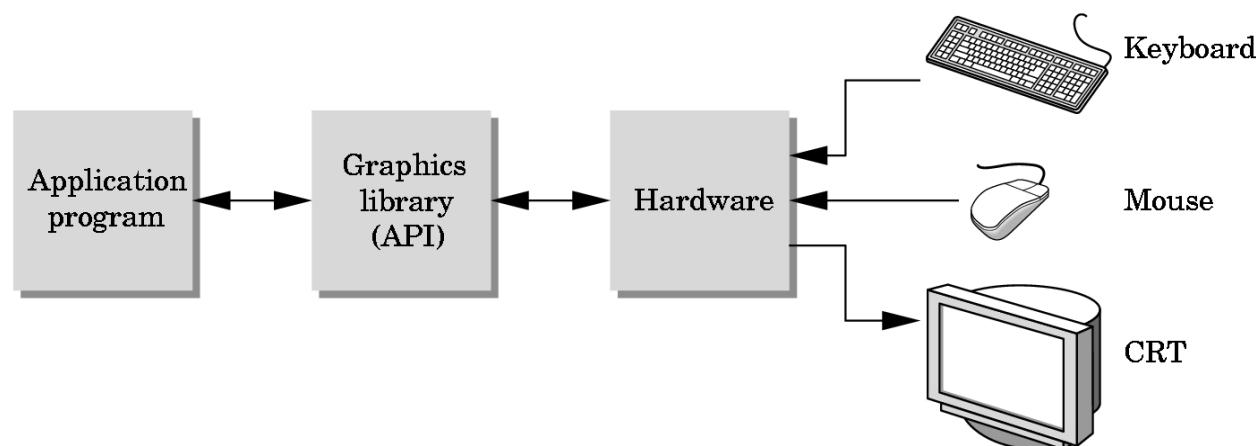
Descrição de Cenas

Descrição de Cenas

- Independentemente da técnica de síntese que se use para gerar a imagem, o conteúdo da cena necessita ser descrito pela aplicação usando a API gráfica
- Em última análise, e se ignorarmos o nível de abstração oferecido pela API usada, será necessário descrever:
 - Os objetos gráficos que compõem a cena (ou primitivas gráficas)
 - Geometria: posições 3D dos vértices
 - Topologia: como os vértices são usados para formar linhas ou polígonos
 - Atributos: informação necessária para o cálculo final da cor e brilho (usualmente sob a forma de materiais)
 - As fontes de luz
 - As características da câmara que servirá para “captar” a imagem

A API

O programador “vê” o sistema gráfico através da interface exposta ao nível do software (**Application Programmer’s Interface**)

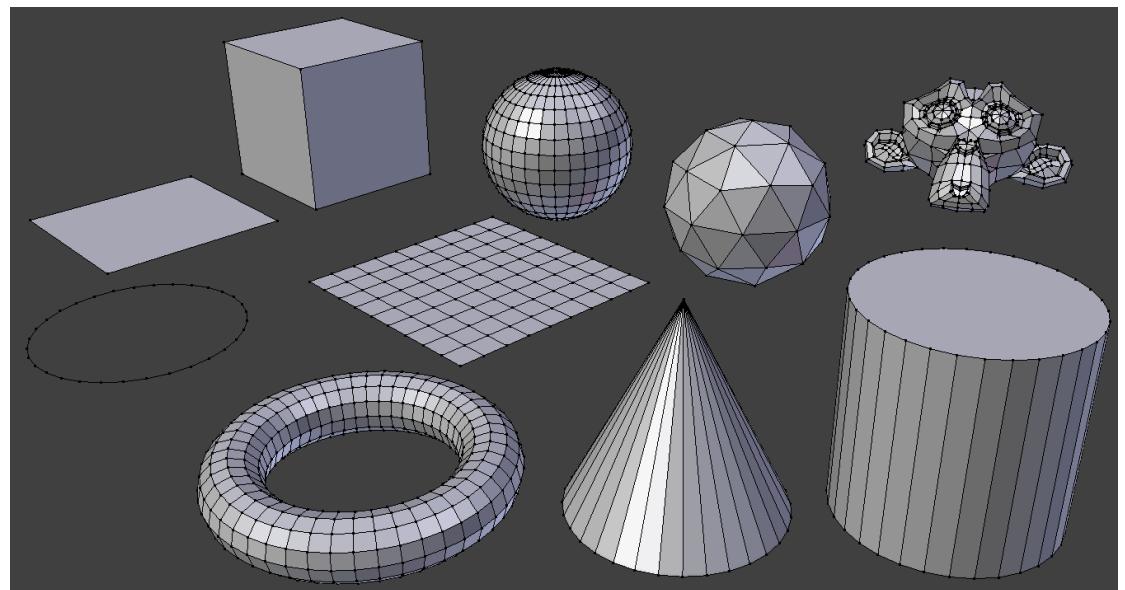


Conteúdo da API

- Funções para especificar os componentes que permitem sintetizar a imagem:
 - Objetos
 - Observador
 - Fontes de luz
 - Materiais
- Outros:
 - Input dos dispositivos (mouse, teclado)
 - Capacidade do sistema

Primitivas Gráficas

- A maior parte das APIs gráficas suportam um reduzido número de primitivas gráficas:
 - Pontos (Objetos 0D)
 - Segmentos de reta (Objetos 1D)
 - Polígonos (objetos 2D)
 - Linhas e superfícies curvas
- Todas as primitivas são definidas através de pontos no espaço - vértices.



Exemplo (depreciado) em OpenGL

- A API OpenGL, nas versões 1.x e 2.x, permitia a utilização do denominado *direct mode*, entretanto depreciado.
- O código apresentado é código que é executado pelo CPU!
- Por cada polígono (um triângulo no exemplo), há uma série de chamadas da API...

```
glBegin(GL_POLYGON);
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(0.0, 1.0, 0.0);
    glVertex3f(0.0, 0.0, 1.0)
glEnd();
```

Início da construção duma primitiva

Tipo de primitiva

Localização dum vértice

Fim da especificação da primitiva

Exemplo WebGL (simplificado)

- Colocar os valores associados aos vértices (normalmente incluindo as coordenadas) em vetores

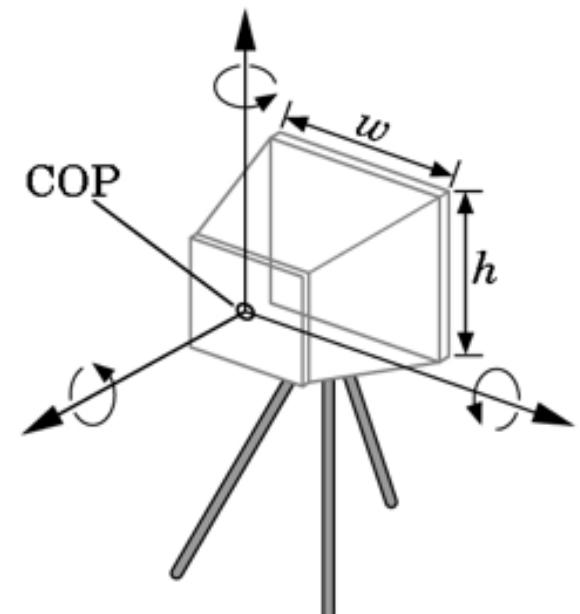
```
var points = [  
    vec3(0.0, 0.0, 0.0),  
    vec3(0.0, 1.0, 0.0),  
    vec3(0.0, 0.0, 1.0)  
];
```

Este é um exemplo redutor, com apenas um triângulo. Mas o vetor poderá conter $3N$ vértices para desenhar N triângulos efetuando uma única chamada.

- Enviar os dados para o GPU
- Pedir ao GPU para desenhar usando o tipo de primitiva pretendida (pontos, linhas, triângulos, etc.)

Especificação da câmara

- 6 DOF (graus de liberdade) - 3 para a posição e 3 para a orientação
 - Posição do centro da lente (center of projection)
 - Orientação (3 ângulos)
- Graus de liberdade adicionais
 - Lente (distância focal)
 - Tamanho do filme/sensor ($w \times h$)
 - Orientação do plano de projeção (filme/sensor) relativamente à direção em que a câmara está a apontar.

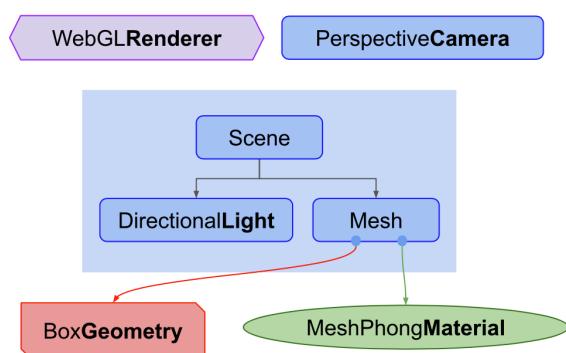


Especificação de Materiais e Luzes

- Características das luzes:
 - Fontes de luz pontual (point light source) vs. áreas luminosas (distributed light sources)
 - Próximas vs distantes
 - Cor
 - Como decai a luz à medida que nos afastamos da fonte numa determinada direção.
- Propriedades dos materiais
 - Absorção - cor do material
 - Tipo de reflexão (difusa vs especular)



Exemplo usando three.js



```
1 import * as THREE from 'https://threejsfundamentals.org/threejs/resources/threejs/r132/';
2
3 function main() {
4   const canvas = document.querySelector('#c');
5   const renderer = new THREE.WebGLRenderer({canvas});
6
7   const fov = 75;
8   const aspect = 2; // the canvas default
9   const near = 0.1;
10  const far = 5;
11  const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
12  camera.position.z = 2;
13
14  const scene = new THREE.Scene();
15
16  {
17    const color = 0xFFFFFF;
18    const intensity = 1;
19    const light = new THREE.DirectionalLight(color, intensity);
20    light.position.set(-1, 2, 4);
21    scene.add(light);
22  }
23
24  const boxWidth = 1;
25  const boxHeight = 1;
26  const boxDepth = 1;
27  const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);
28
29  const material = new THREE.MeshPhongMaterial({color: 0x44aa88}); // greenish blue
30
31  const cube = new THREE.Mesh(geometry, material);
32  scene.add(cube);
33
34  function render(time) {
35    time *= 0.001; // convert time to seconds
36
37    cube.rotation.x = time;
38    cube.rotation.y = time;
39
40    renderer.render(scene, camera);
41
42    requestAnimationFrame(render);
43  }
44  requestAnimationFrame(render);
45
46
47
48  main();
```

Síntese de Imagem

Percepção 3D

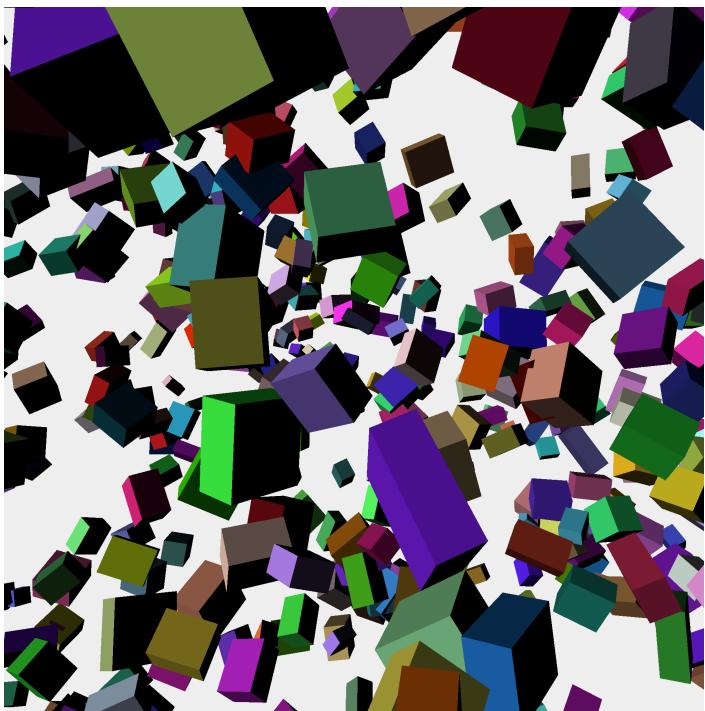
- Através de 2 imagens independentes, uma para cada olho



* Exemplos retirados de <http://www.lindandavid.com/stereo.htm>

Percepção 3D

- Através da oclusão de objetos



https://threejs.org/examples/#webgl_interactive_cubes



https://threejs.org/examples/#webgl_loader_ldraw

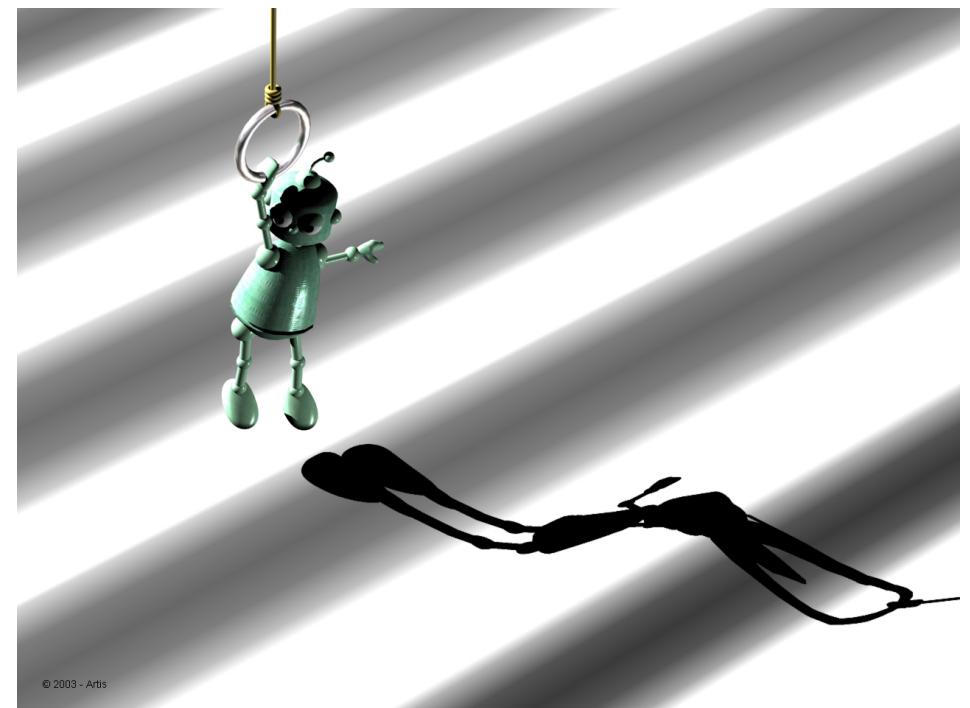
Percepção 3D

- Através de gradientes de cor

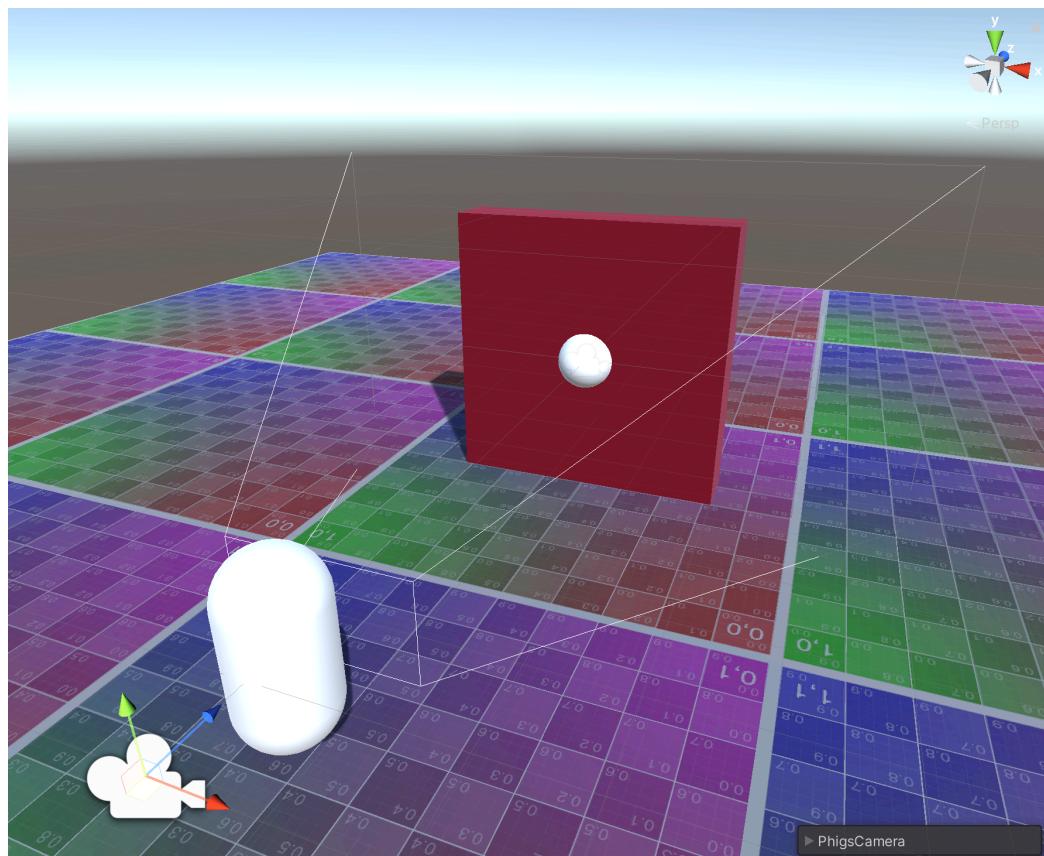


Percepção 3D

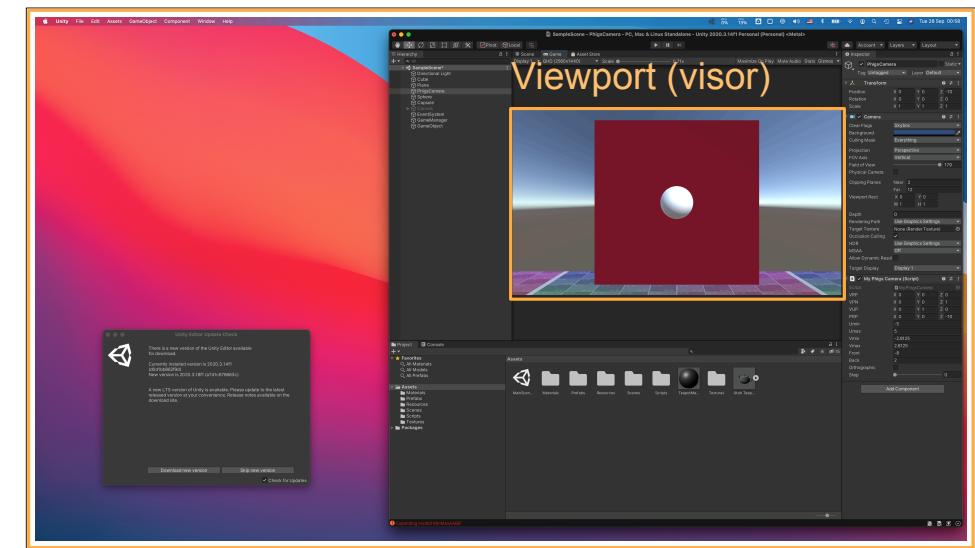
- Através das sombras



Volume de Visão* e Visor



Ecrã



*Em 2D, o Volume de visão é denominado por janela (de visualização)

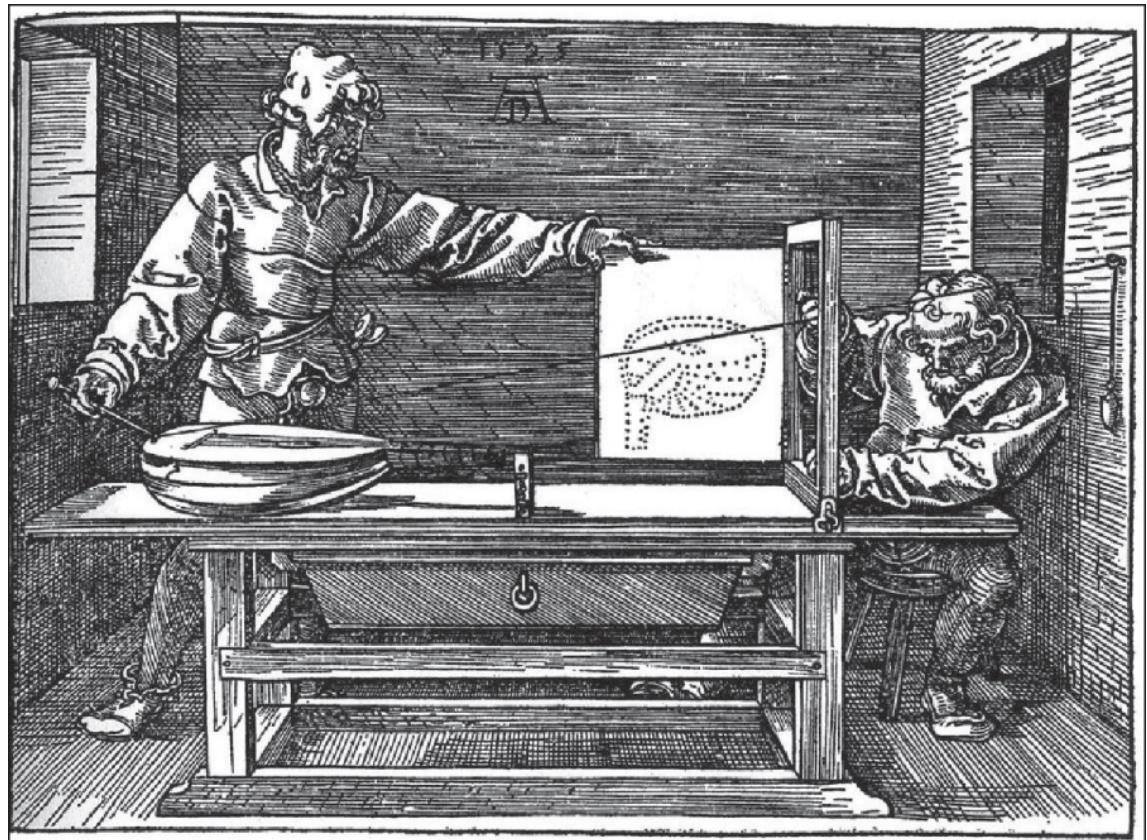
Métodos para Síntese de Imagens

Object driven

As técnicas de síntese que processam cada objeto separadamente constituem a família dos algoritmos baseados em *rasterização*.

Os pontos da figura podem ser os vértices dum modelo do objeto, unidos depois para formar pequenos polígonos. Cada polígono pode ser tratado de forma independente.

Como se pode testar se um ponto é visível ou não?



illustrations published in Albrecht Dürer's *The Painter's Manual*, 1525

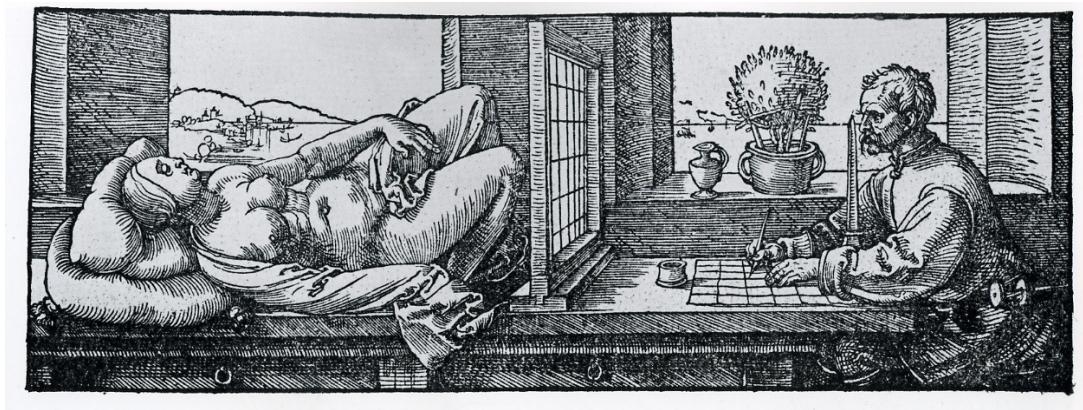
Image driven

As técnicas de síntese conduzidas pela imagem determinam quais os objectos visíveis em cada pixel.

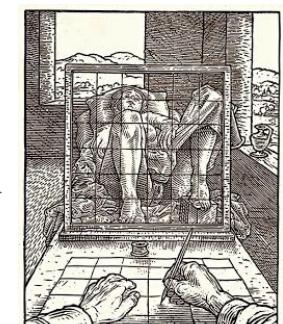
Se cada quadricula for suficientemente pequena, do tamanho de um pixel ou inferior, as duas grandes tarefas destes algoritmos são:

- Determinar qual o objecto visível nesse pixel
- Determinar a cor com que se deverá pintar esse pixel

Ray tracing é um exemplo desta abordagem.



Ponto de vista do artista - câmara



illustrations published in Albrecht Dürer's *The Painter's Manual*, 1525

Illuminação Global vs Local

Na realidade, não é possível determinar o aspecto de cada objecto de forma isolada ou independente dos restantes:

- Alguns objectos estão na sombra
- A luz pode refletir num objecto e ir em direcção a outro
- Alguns objectos podem deixar passar a luz (transparentes ou translúcidos)

Os algoritmos de **iluminação local** tratam cada objecto de forma independente dos outros, produzindo assim resultados aproximados

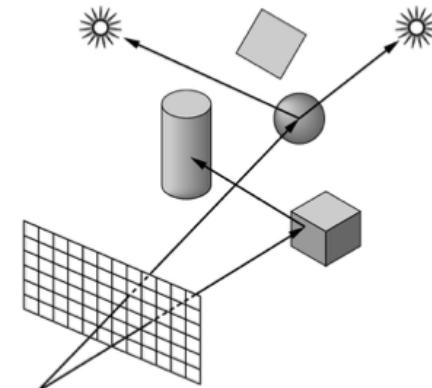
Os algoritmos de **iluminação global** tratam da cena como um todo, permitindo as interacções entre as superfícies de diferentes objectos.

Abordagem física

Ray tracing: seguir o percurso dos raios de luz, partindo do centro de projeção até que sejam absorvidos ou se percam no espaço. Neste algoritmo é fácil produzir efeitos de iluminação global:

- Reflexões múltiplas
- Transparência
- Sombras

Dispendioso!

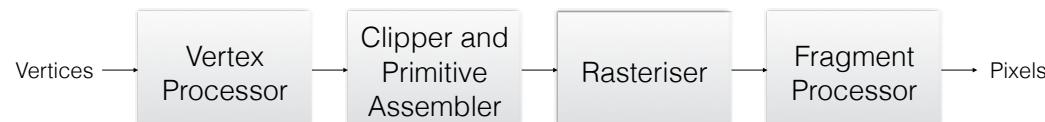


Radiosidade: método baseado nas transferências de energia entre os objectos presentes na cena

Ainda mais dispendioso!

Abordagem prática

- Processar os objetos isoladamente, na ordem pela qual são gerados pela aplicação
 - Apenas se podem considerar fenómenos locais de iluminação, sem ter em atenção os outros objetos.
- Arquitetura em pipeline

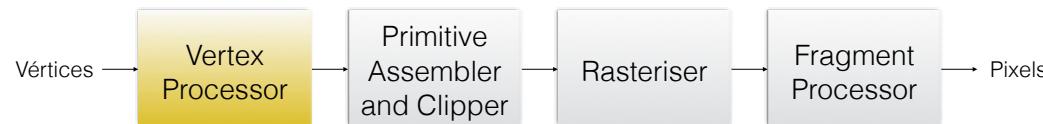


- Todas as fases podem ser implementadas no hardware gráfico (GPU)

Pipeline de Visualização 3D

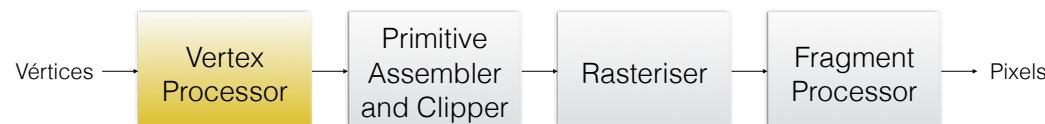
Processamento de vértices

- Uma grande parte do trabalho no pipeline gráfico consiste em efetuar mudanças de coordenadas dum referencial para outro
 - Coordenadas dos pontos em referenciais próprios do objeto
 - Coordenadas no referencial global da cena/mundo
 - Coordenadas no referencial da câmara
 - Coordenadas no referencial do ecrã
- Cada mudança de referencial equivale a uma operação matricial (mudança de base)
- O processamento de vértices também inclui a transformação de atributos (ex. cor), associados pela aplicação a cada vértice, em valores que irão variar durante o varrimento dos pixels gerados para as primitivas gráficas (triângulos na sua maioria)



Projeção

- A projeção permite combinar os objetos 3D e a forma como se observa a cena numa imagem 2D
 - Projeção perspetiva: as linhas projetantes encontram-se num ponto (centro de projeção)
 - Projeção paralela: as linhas projetantes são paralelas entre si (a mesma direção)
- A projeção é também realizada através duma operação matricial (multiplicação por uma matriz de projeção)



Vertex processor

```
#version 300 es

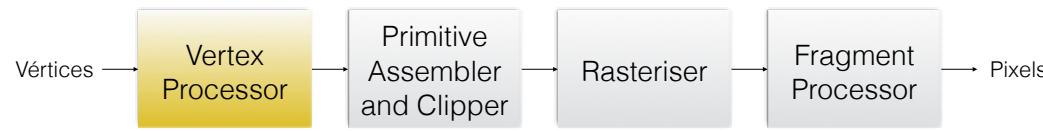
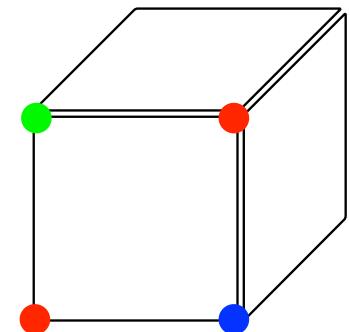
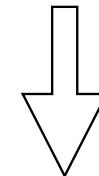
in vec4 a_position;
in vec4 a_color;

uniform mat4 u_projection;
uniform mat4 u_model_view;

out vec4 v_color;

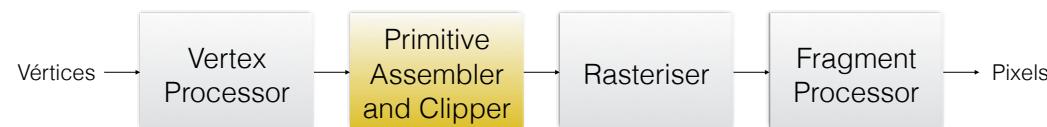
void main() {
    gl_Position = u_projection * u_model_view * a_position;
    v_color = a_color;
}
```

a_color	a_position
green	(-1,1,1)
red	(1,1,1)
red	(-1,-1,1)
blue	(1,-1,1)



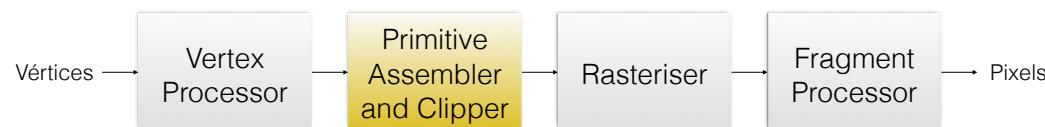
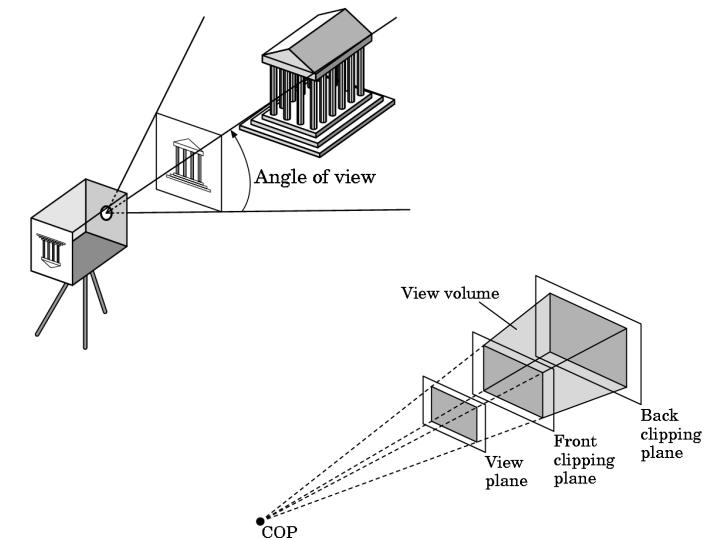
Primitive Assembly

- Os vértices submetidos ao pipeline são agrupados para a formação das primitivas, antes que se possa proceder ao seu varrimento (geração dos pixels que as constituem) e recorte (clipping)
 - Segmentos de reta
 - Polígonos (apenas triângulos no caso do WebGL)
 - Linhas curvas (discretizadas posteriormente em segmentos)
 - Superfícies curvas (discretizadas posteriormente em polígonos)



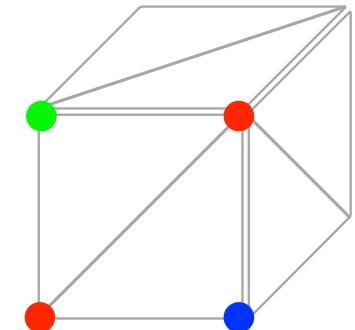
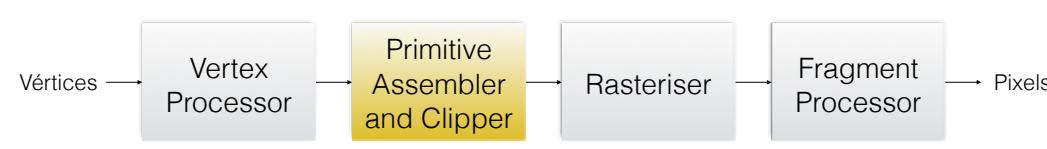
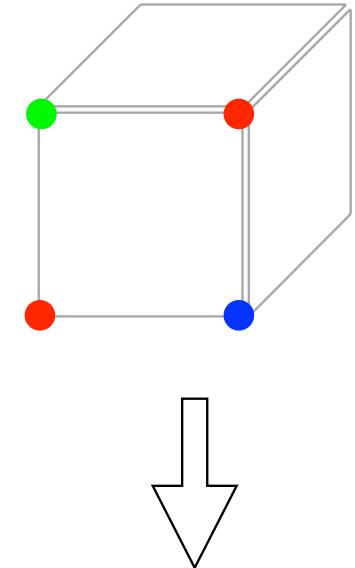
Clipper

- Uma câmara, quer seja real ou virtual, apenas consegue “capturar” parte do mundo
- Os objetos que não estão dentro dessa mesma parte visível (volume de visão) são recortados (total ou parcialmente)



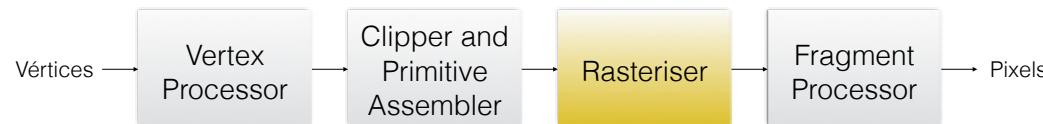
Primitive Assembly and Clipper

Neste exemplo vamos assumir que as primitivas a desenhar são triângulos e que nenhuma está fora do volume de visão, não havendo lugar a qualquer recorte.



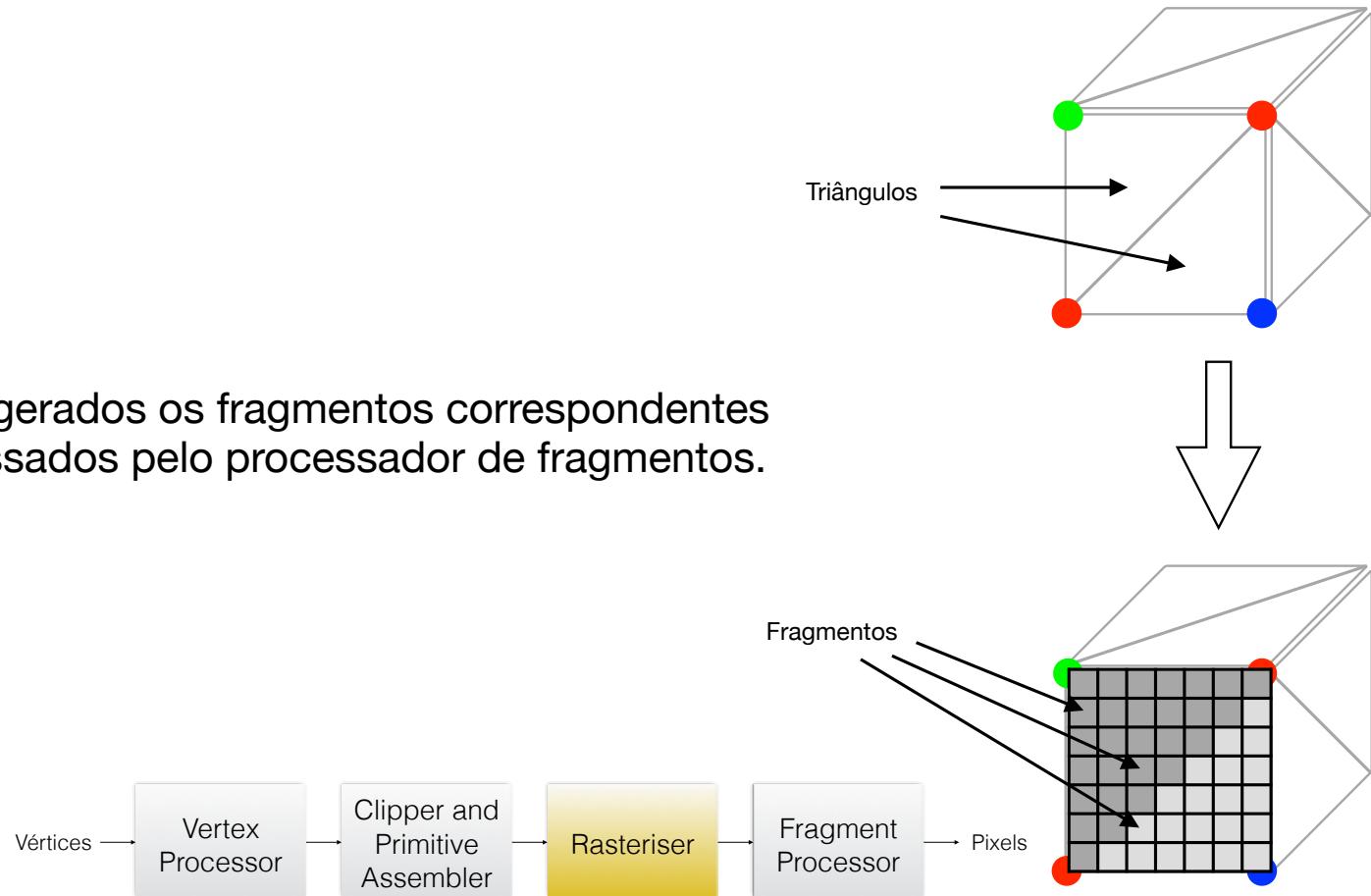
Rasterização - Varrimento

- Se um objeto não for recortado na sua totalidade (descartado), os pixels correspondentes no *frame buffer* terão que ser gerados (saber quais são)
- Um *rasteriser* produz um conjunto de fragmentos (potenciais pixels no final do pipeline) para cada objeto.
- Sendo potenciais pixels, os fragmentos possuem:
 - Localização no *frame buffer*
 - valores de cor e profundidade (coordenada z no referencial da câmara)
- Os atributos associados aos vértices (ou valores calculados a partir deles) poderão ser propagados para os fragmentos e interpolados sobre os objetos pelo módulo de varrimento.



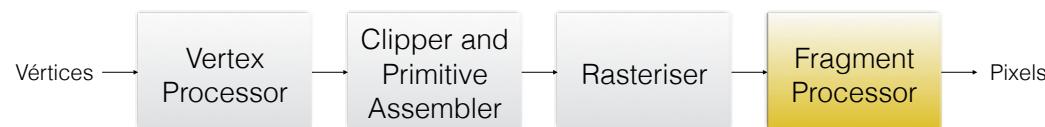
Rasteriser

Para cada primitiva são gerados os fragmentos correspondentes que serão depois processados pelo processador de fragmentos.



Processamento de fragmentos

- Os fragmentos são processados para determinar a cor final do pixel correspondente no *frame buffer*
- As cores podem ser determinadas por mapeamento de texturas e por interpolação de valores de atributos nos vértices
- Os fragmentos poderão ser “bloqueados” por outros mais perto da câmara
 - Remoção de superfícies ocultas (**Hidden Surface Removal**)



Fragment Processor

```
#version 300 es  
  
precision mediump float;  
  
in vec4 v_color;  
out vec4 color;  
  
void main() {  
    color = v_color;  
}
```

