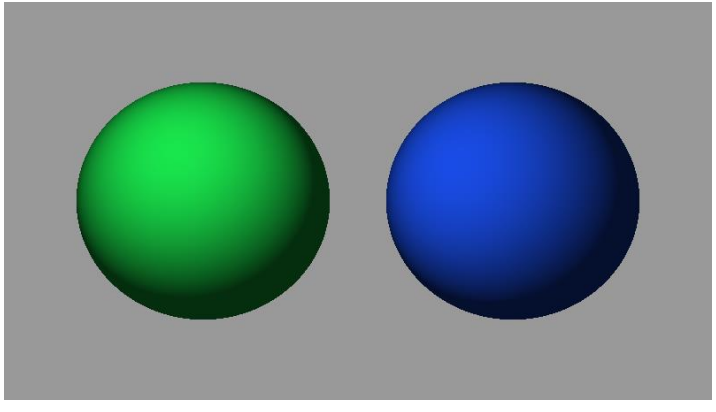
Insper

Computação Gráfica

Ray Marching 2

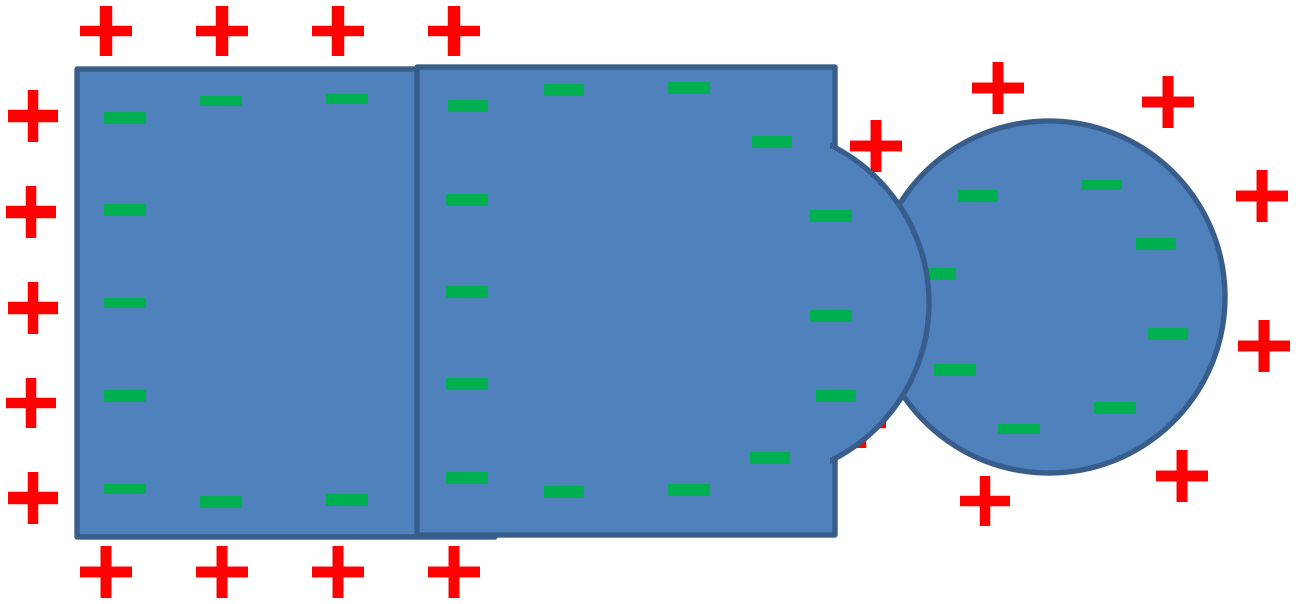
Cores (Truque)

```
float sdSphere(vec3 p, float r) {
  return length(p) - r;
}

vec4 minWithColor(vec4 obj1, vec4 obj2) {
  if (obj2.a < obj1.a) return obj2;
  return obj1;
}

vec4 sdScene(vec3 p) {
  vec4 sphereLeft = vec4(vec3(0.1, 0.9, 0.3), sdSphere(p - vec3(-2.5, 0, -2), 2.0));
  vec4 sphereRight = vec4(vec3(0.1, 0.3, 0.9), sdSphere(p - vec3(2.5, 0, -2), 2.0));
  vec4 co = minWithColor(sphereLeft, sphereRight);
  return co;
}
```



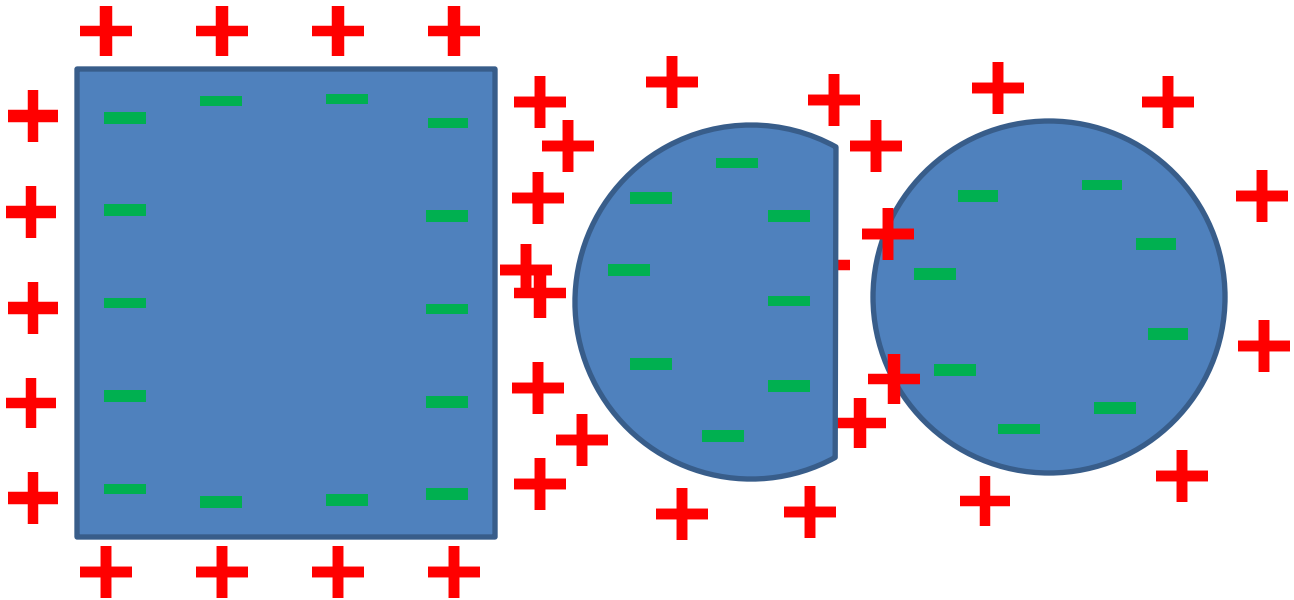
Operações de União, Intersecção e Diferença

A operação mais simples que temos é a **união**. Para isso imagine dois objetos sobrepostos. Se uma das funções de distância retornar um número negativo, vamos ficar com ele (e ignorar o positivo), assim uma das estratégias é usar a função **min()**.



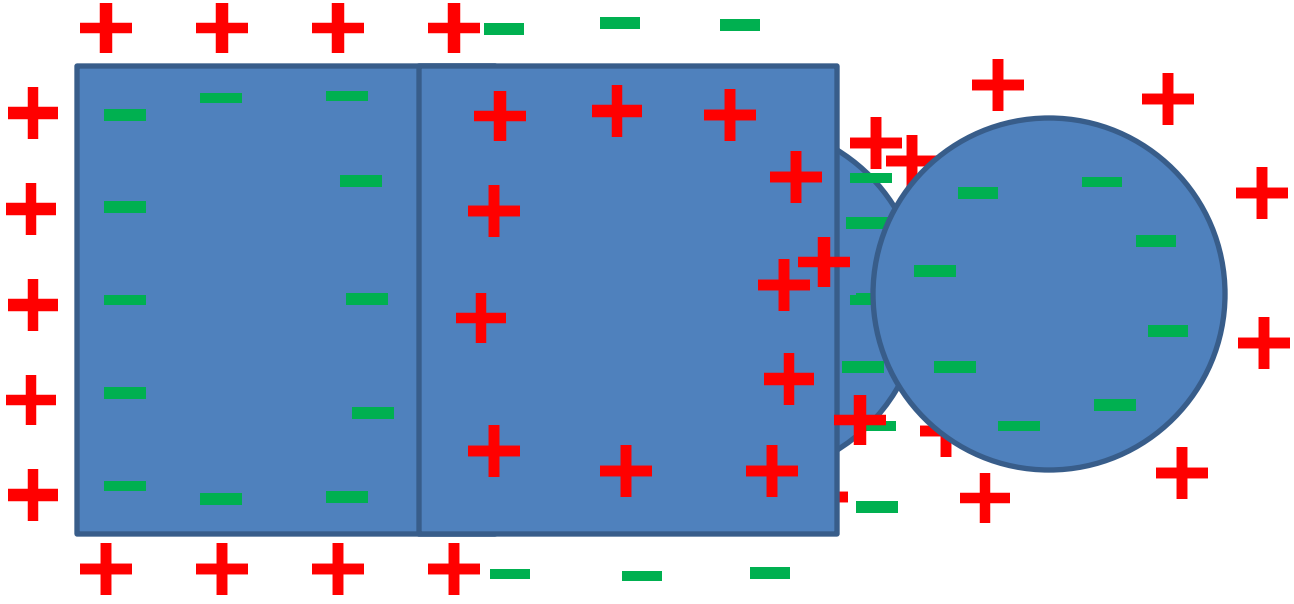
Operações de União, Intersecção e Diferença

Outra operação que temos é a intersecção. Para isso imagine que só queremos se ambas as funções retornarem valores negativos. Só nessa região estaremos dentro dos dois objetos. Para isso usamos uma função **max()**.



Operações de União, Intersecção e Diferença

Para a operação de intersecção, imagine que estamos invertendo uma geometria (o que está dentro fica fora e vice versa) e depois fazemos a intersecção. Para isso negamos o resultado de uma função de depois fazemos a interseção com o **max().**

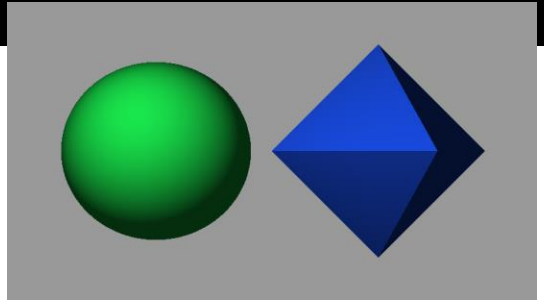


Operações de União, Intersecção e Diferença

Da mesma forma que fazíamos operações entre duas geometrias em 2D, podemos fazer em 3D.

Vamos criar um octaedro para fazer operações com a esfera.

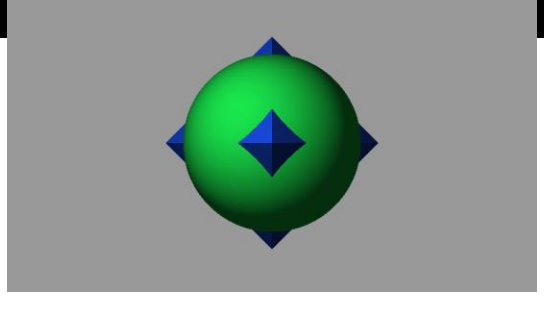
```
float sdOctahedron( vec3 p, float s){
  p = abs(p);
  return (p.x+p.y+p.z-s)*0.57735027;
}

vec4 sdScene(vec3 p) {
  vec4 sphere = vec4(vec3(0.1, 0.9, 0.3), sdSphere(p - vec3(-2.5, 0, -2), 2.0));
  vec4 octahedron = vec4(vec3(0.1, 0.3, 0.9), sdOctahedron(p - vec3(2.5, 0, -2), 2.5));
  vec4 co = minWithColor(sphere, octahedron);
  return co;
}
...
```



União

Já estávamos fazendo o mínimo para juntas as funções.
Assim, sem grandes novidades

```
float sdOctahedron( vec3 p, float s){
  p = abs(p);
  return (p.x+p.y+p.z-s)*0.57735027;
}

vec4 sdScene(vec3 p) {
  vec4 sphere = vec4(vec3(0.1, 0.9, 0.3), sdSphere(p - vec3(0, 0, -2), 2.0));
  vec4 octahedron = vec4(vec3(0.1, 0.3, 0.9), sdOctahedron(p - vec3(0, 0, -2), 2.5));
  vec4 co = minWithColor(sphere, octahedron);
  return co;
}
...
```



Intersecção

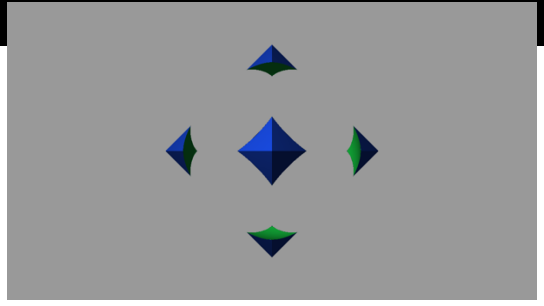
Vamos agora usar uma função max().

```
vec4 maxWithColor(vec4 obj1, vec4 obj2) {
  if (obj2.a > obj1.a) return obj2;
  return obj1;
}

vec4 sdScene(vec3 p) {
  vec4 sphere = vec4(vec3(0.1, 0.9, 0.3), sdSphere(p - vec3(0, 0, -2), 2.0));
  vec4 octahedron = vec4(vec3(0.1, 0.3, 0.9), sdOctahedron(p - vec3(0, 0, -2), 2.5));
  vec4 co = maxWithColor(sphere, octahedron);
  return co;
}
...
```

Diferença

Vamos agora inverter uma função e usar uma função max().

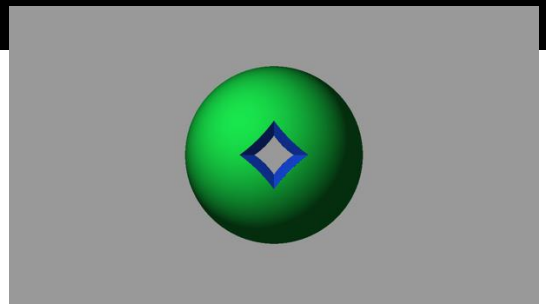
```
vec4 maxWithColor(vec4 obj1, vec4 obj2) {
  if (obj2.a > obj1.a) return obj2;
  return obj1;
}

vec4 sdScene(vec3 p) {
  vec4 sphere = vec4(vec3(0.1, 0.9, 0.3), sdSphere(p - vec3(0, 0, -2), 2.0));
  vec4 octahedron = vec4(vec3(0.1, 0.3, 0.9), sdOctahedron(p - vec3(0, 0, -2), 2.5));
  vec4 co = maxWithColor(sphere*vec4(1,1,1,-1), octahedron);
  return co;
}
...
```



Diferença

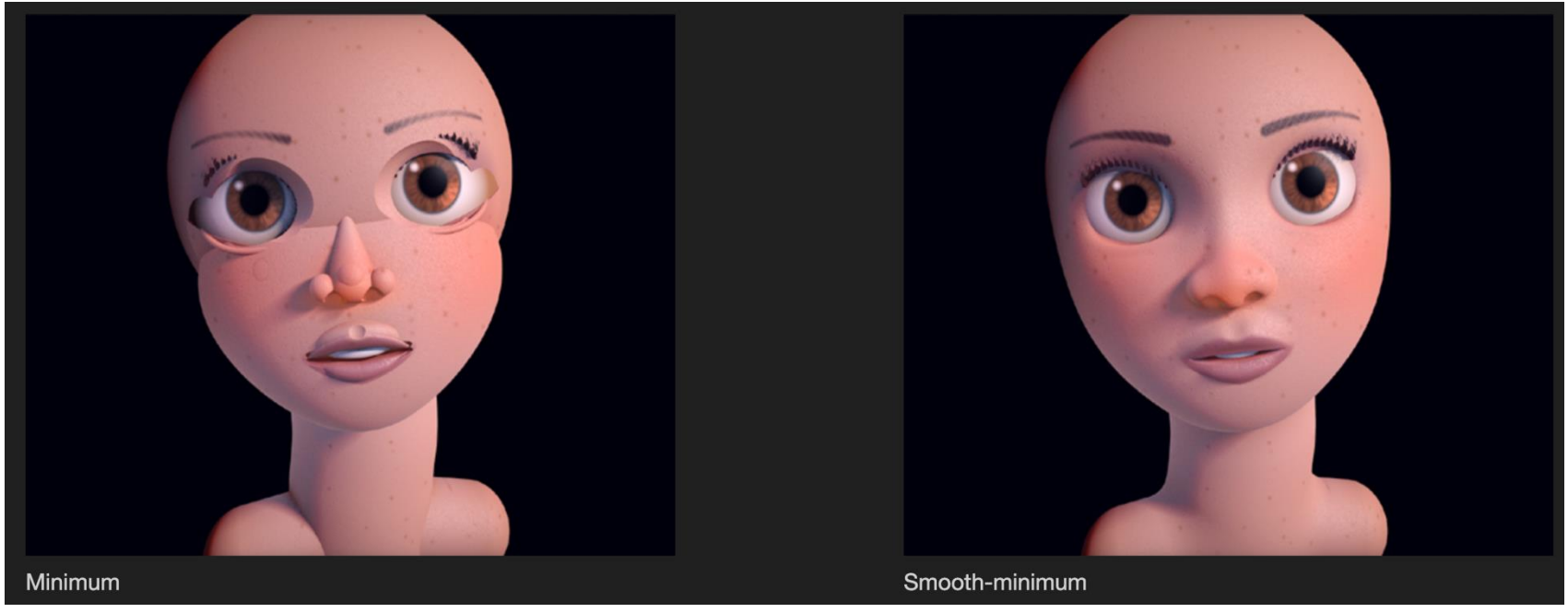
Ou ainda

```
vec4 maxWithColor(vec4 obj1, vec4 obj2) {
  if (obj2.a > obj1.a) return obj2;
  return obj1;
}

vec4 sdScene(vec3 p) {
  vec4 sphere = vec4(vec3(0.1, 0.9, 0.3), sdSphere(p - vec3(0, 0, -2), 2.0));
  vec4 octahedron = vec4(vec3(0.1, 0.3, 0.9), sdOctahedron(p - vec3(0, 0, -2), 2.5));
  vec4 co = maxWithColor(sphere, octahedron*vec4(1,1,1,-1));
  return co;
}
...
```



Smooth-Minimum em Ray Marching

O operador Smooth-Minimum ou Smooth-Union é uma versão suave da função min() que combina duas formas misturando-as e fundindo-as, se estiverem próximas o suficiente.

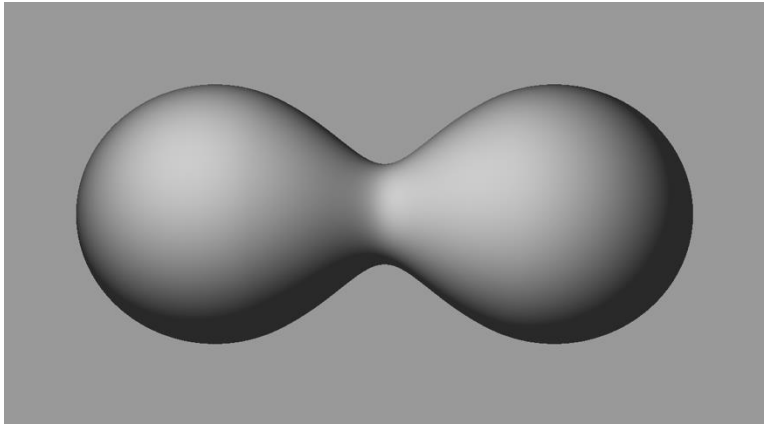


Fonte: https://iquilezles.org/articles/smin/

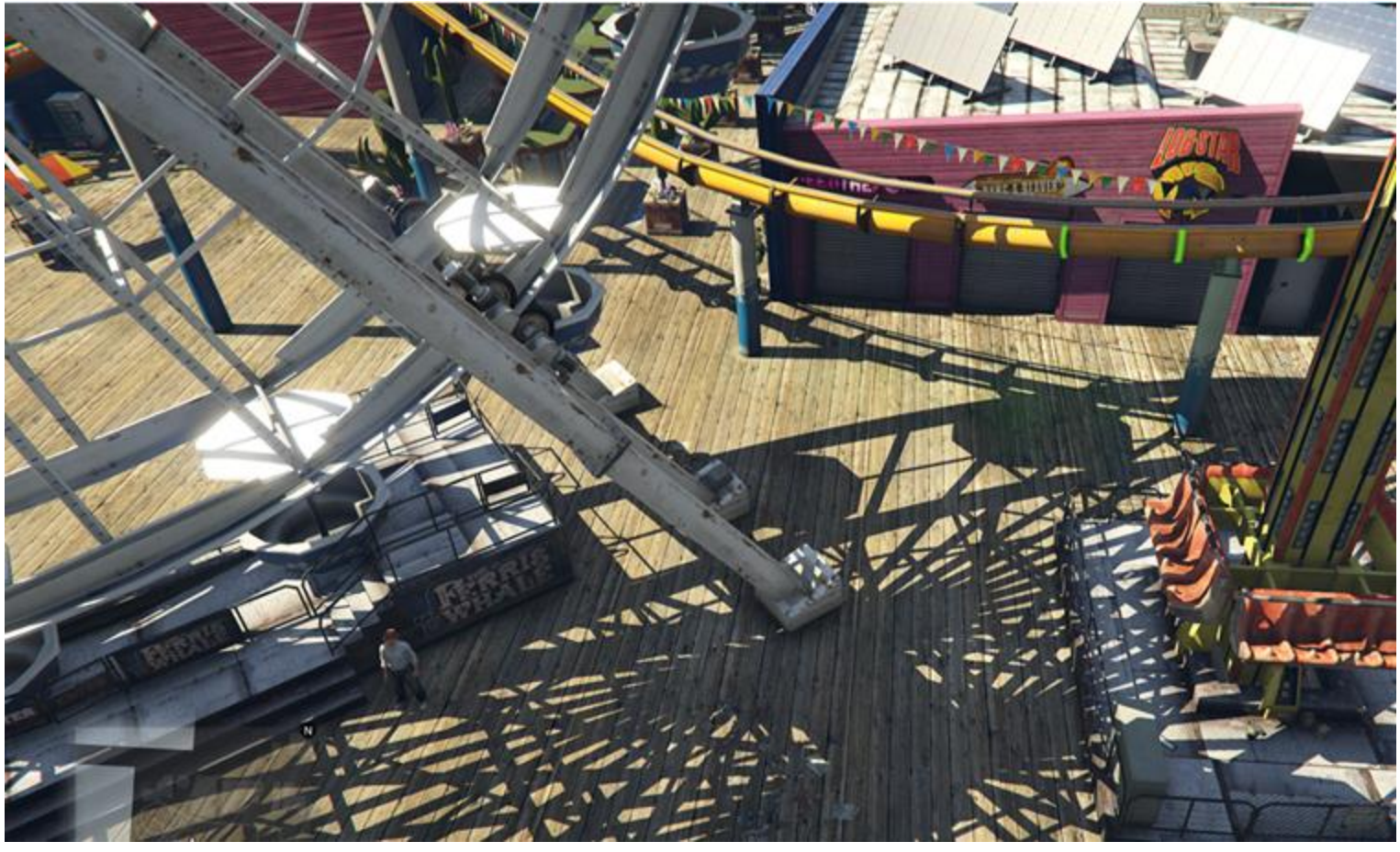
Aplicando a fórmula

Exemplo

```
float smin(float a, float b, float k) {
  float h = clamp(0.5 + 0.5 * (b - a) / k, 0.0, 1.0);
  return mix(b, a, h) - k * h * (1.0 - h);
}

vec4 minWithColor(vec4 obj1, vec4 obj2) {
  float t = smin(obj1.w, obj2.w, 2.5);
  return vec4(0.8, 0.8, 0.8, t);
}
```



Sombras

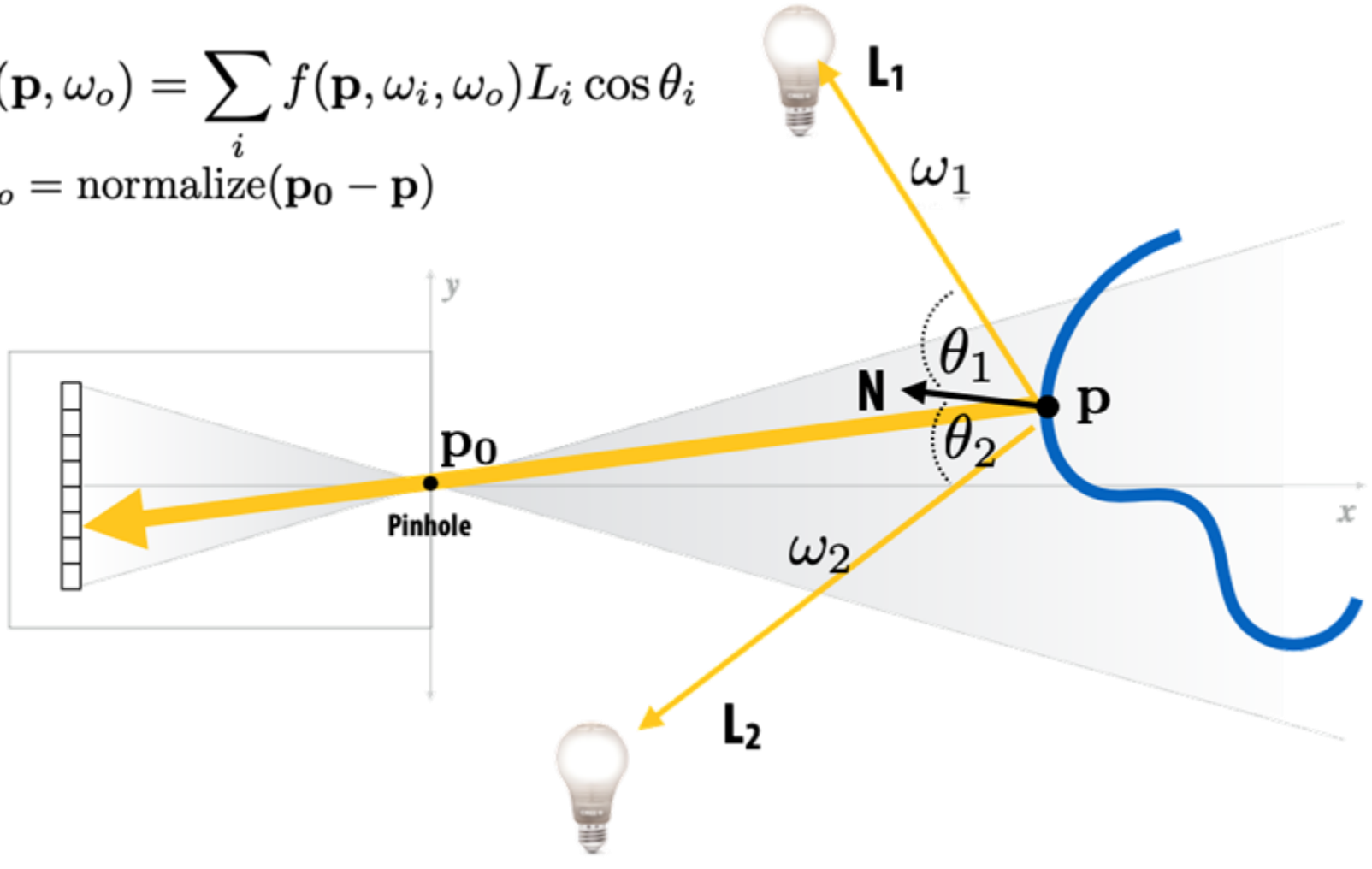


Créditos: Grand Theft Auto V

Revisão: quanta luz é refletida de P através do P_0

$$L(\mathbf{p}, \omega_o) = \sum_i f(\mathbf{p}, \omega_i, \omega_o) L_i \cos\theta_i$$

$$\omega_o = \text{normalize}(\mathbf{p_0} - \mathbf{p})$$



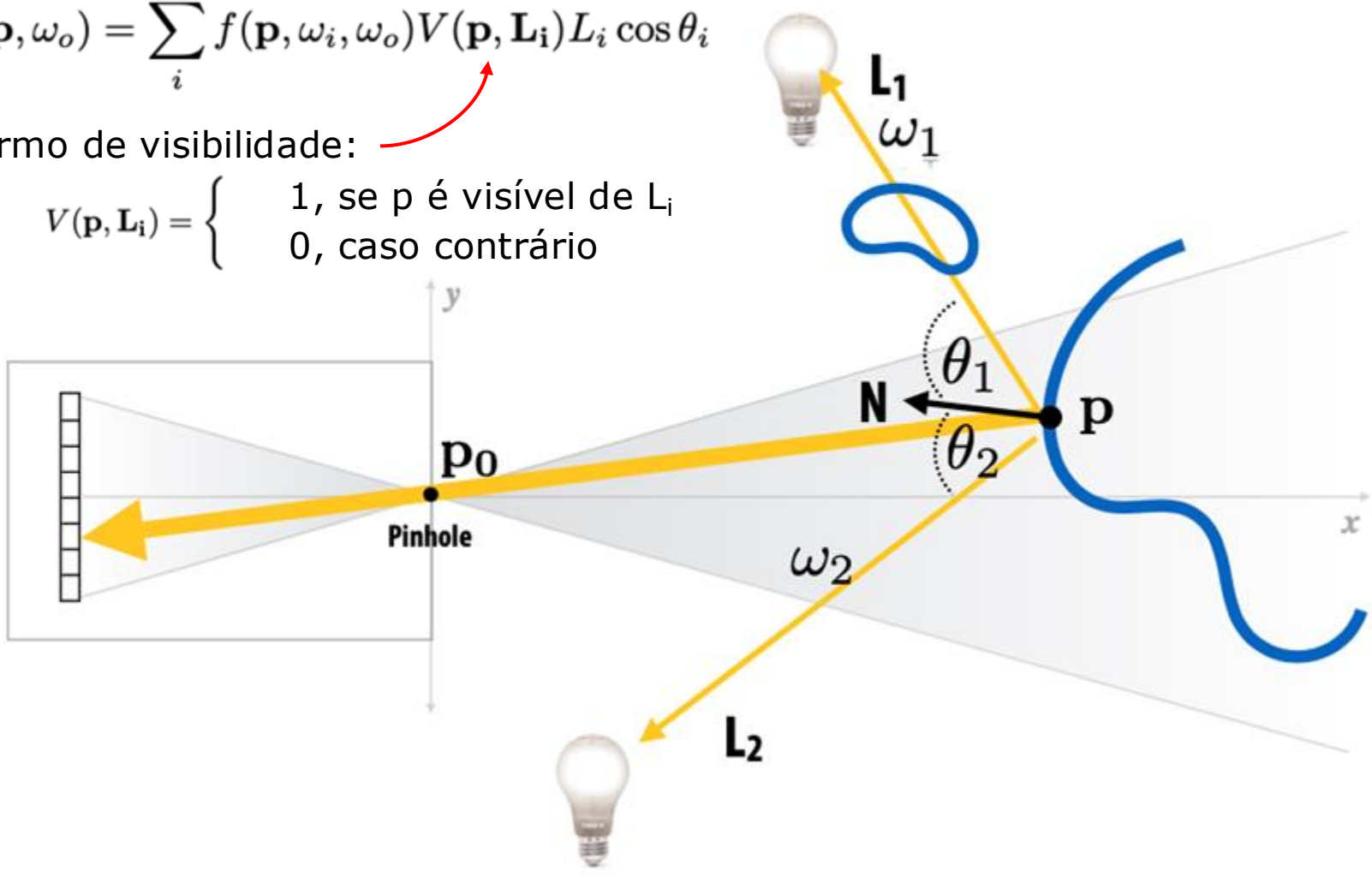
Insper

Visibilidade

$$L(\mathbf{p}, \omega_o) = \sum_i f(\mathbf{p}, \omega_i, \omega_o) V(\mathbf{p}, \mathbf{L_i}) L_i \cos\theta_i$$

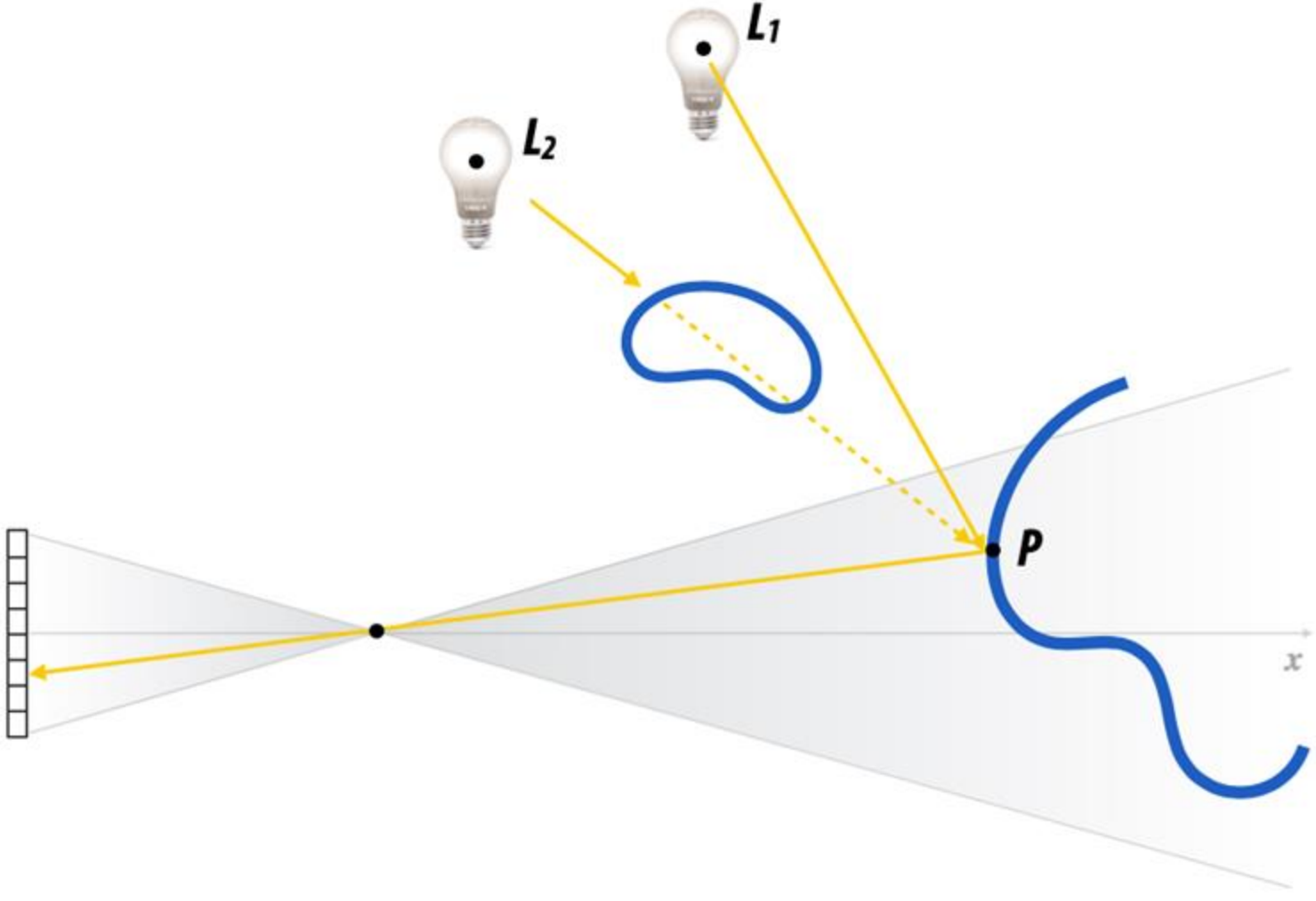
termo de visibilidade:

$$V(\mathbf{p}, \mathbf{L_i}) = \begin{cases} 1, \text{ se p é visível de } L_i \\ 0, \text{ caso contrário} \end{cases}$$



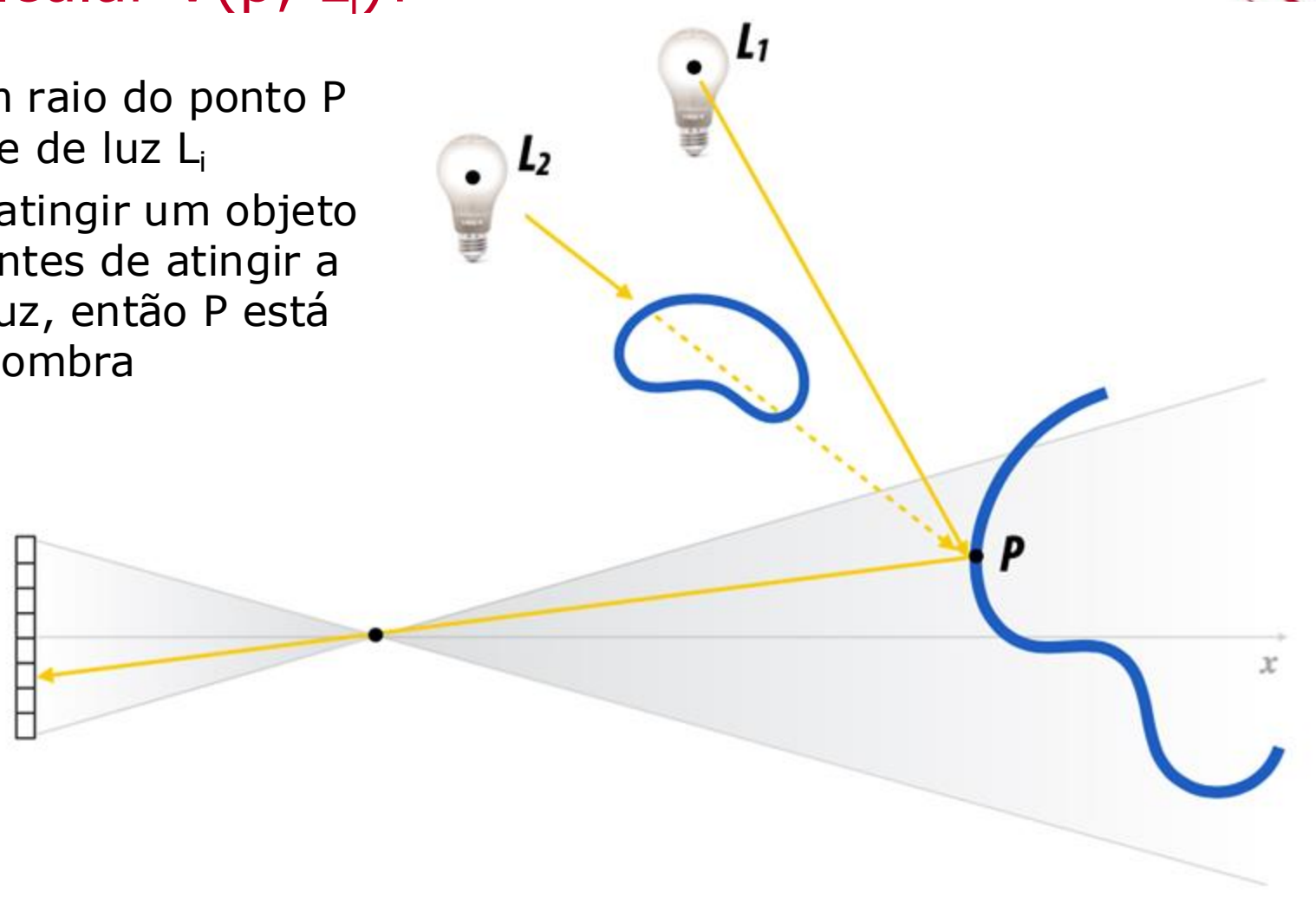
Insper

Como saber se superfície está em uma sombra?



Como calcular V(p, L_i)?

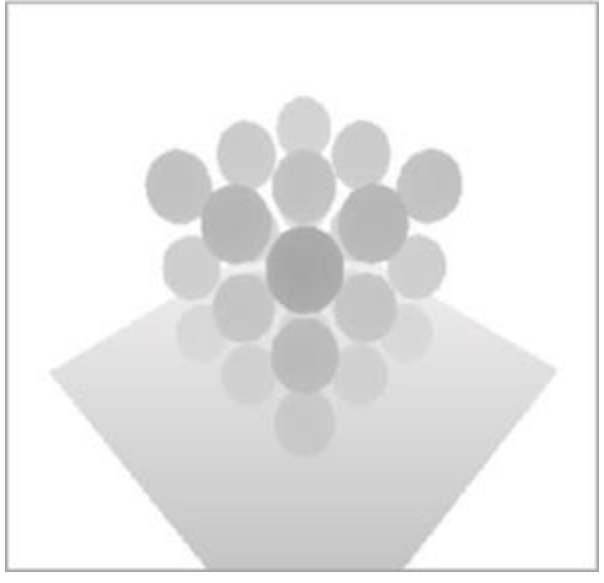
- Lançar um raio do ponto P até a fonte de luz L_i
- Se o raio atingir um objeto da cena antes de atingir a fonte de luz, então P está em uma sombra

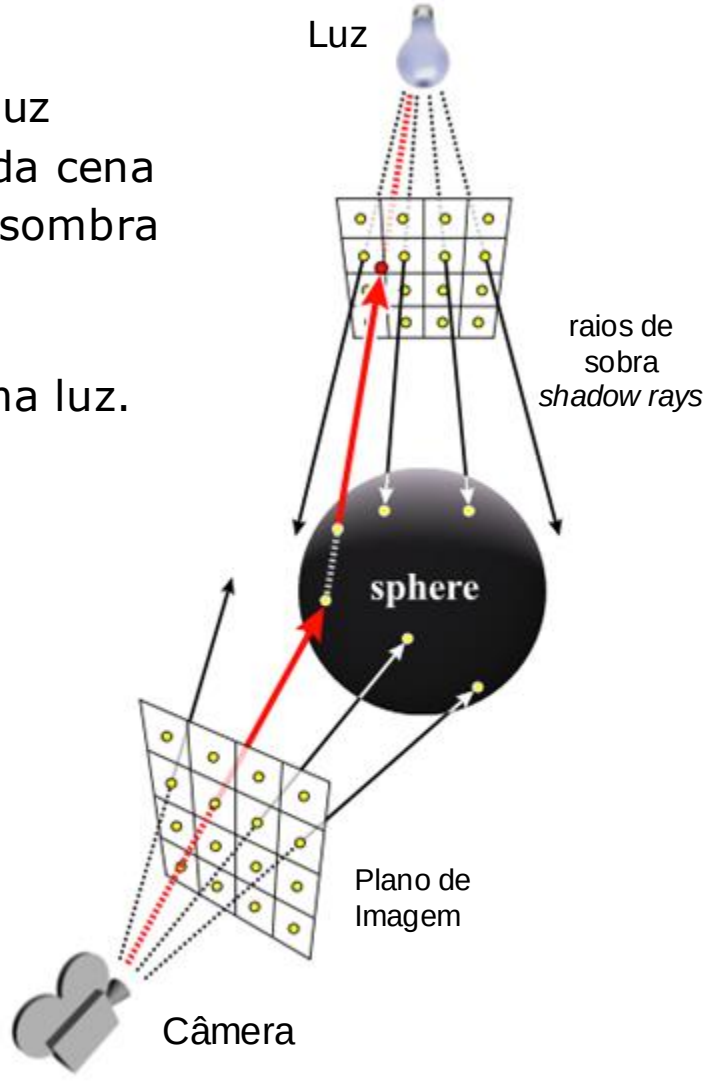


Shadow Mapping [Williams 78]

1. Coloque a câmera na posição de uma fonte de luz
2. Renderize a cena para calcular a profundidade da cena
3. Armazene resultados de interseção de raios de sombra pré-computados em uma textura

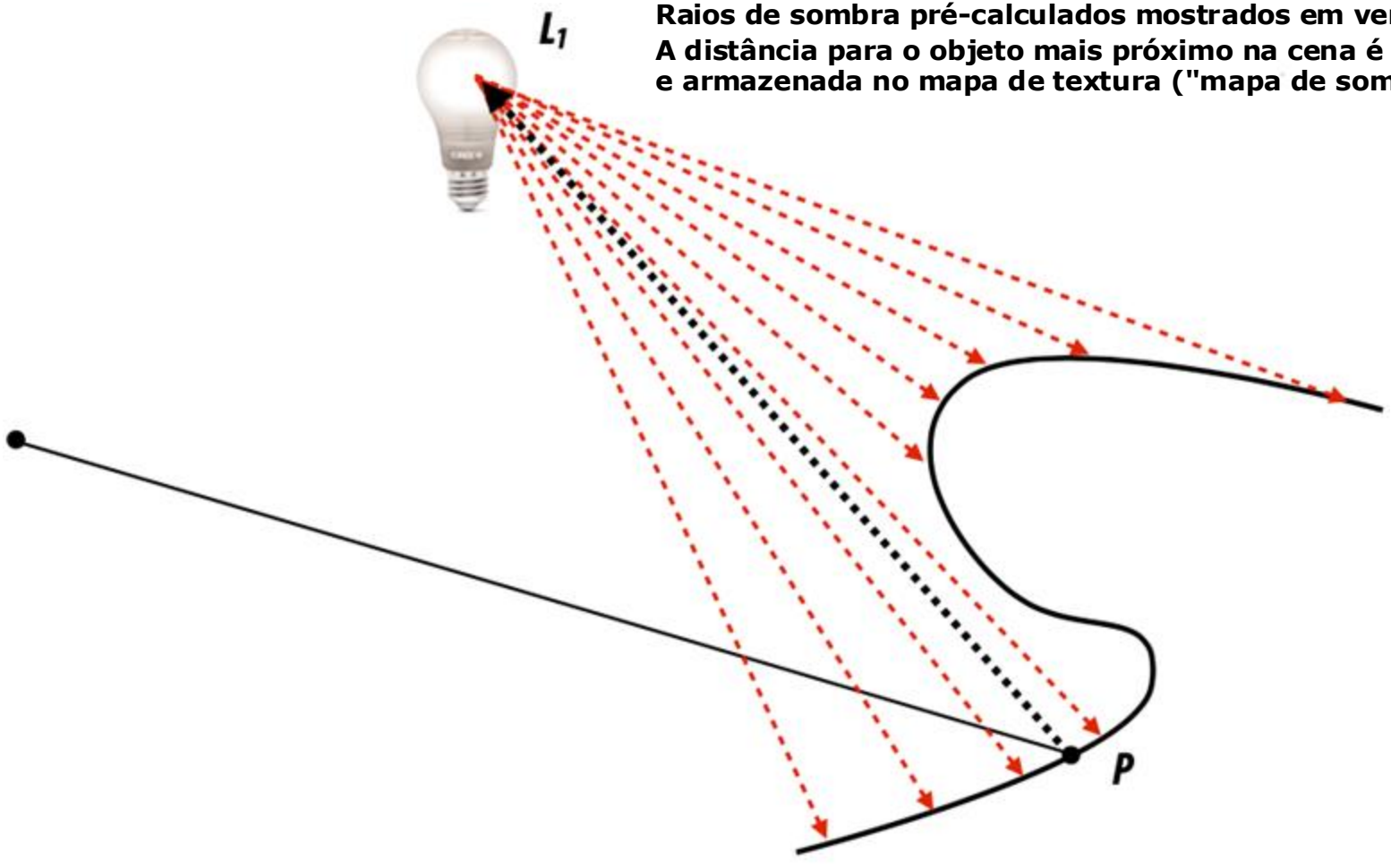
"Mapa de sombra" = mapa de profundidade de uma luz.





Créditos: Segal et al. 92

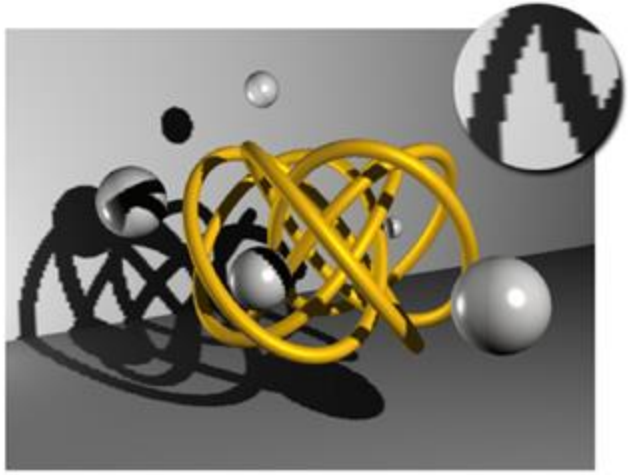
Pesquisa de textura de sombra



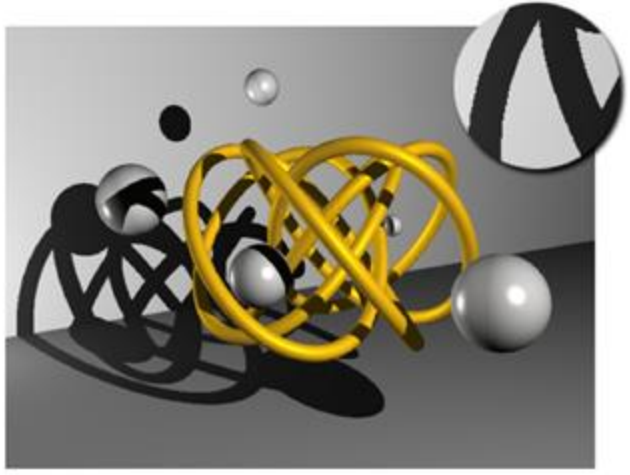
Raios de sombra pré-calculados mostrados em vermelho:
A distância para o objeto mais próximo na cena é pré-calculada e armazenada no mapa de textura ("mapa de sombras")

Insper

Artefatos na sombra devido a baixa amostragem



Sombras calculadas usando mapa de sombra

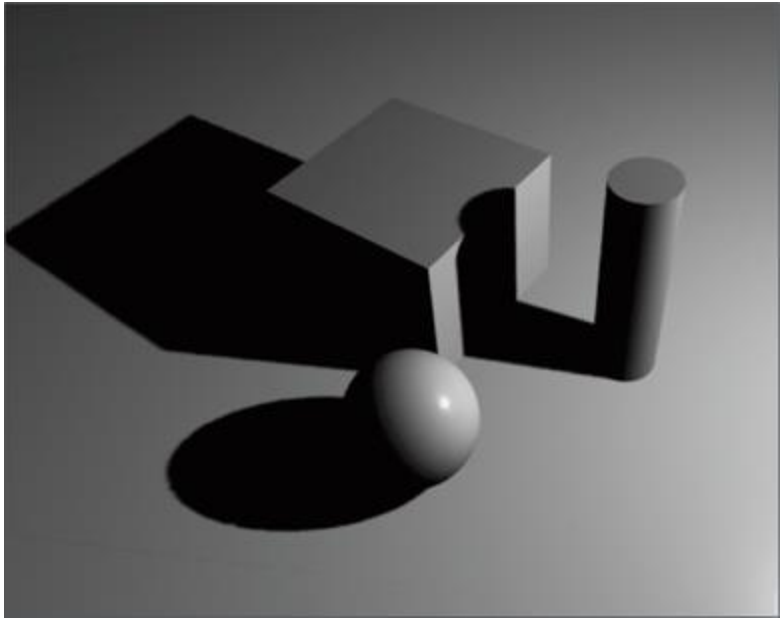


Sombras usando ray tracing

(resultado do cálculo da visibilidade ao longo do raio entre o ponto de superfície e a luz diretamente usando ray tracing)

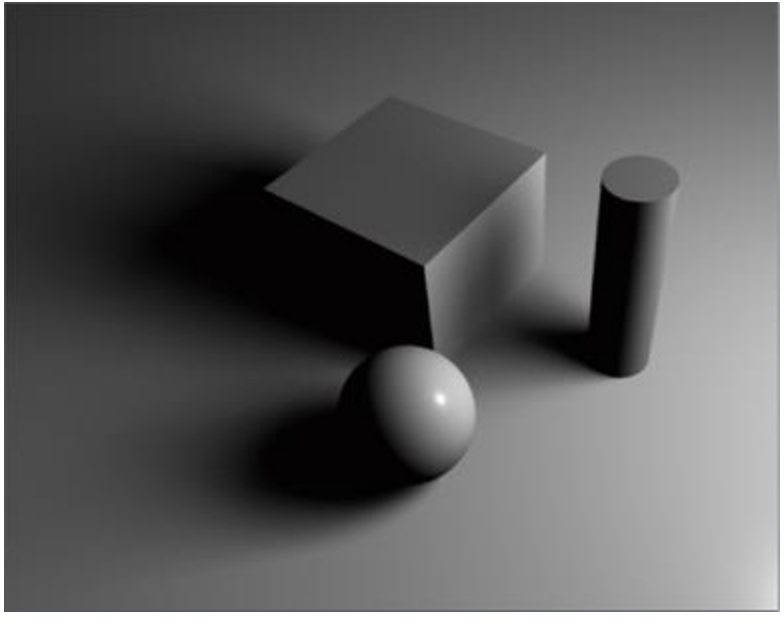
crédito: Johnson et al. TOG 2005

Tipos de sombras (Hard vs Soft Shadows)



Hard shadows

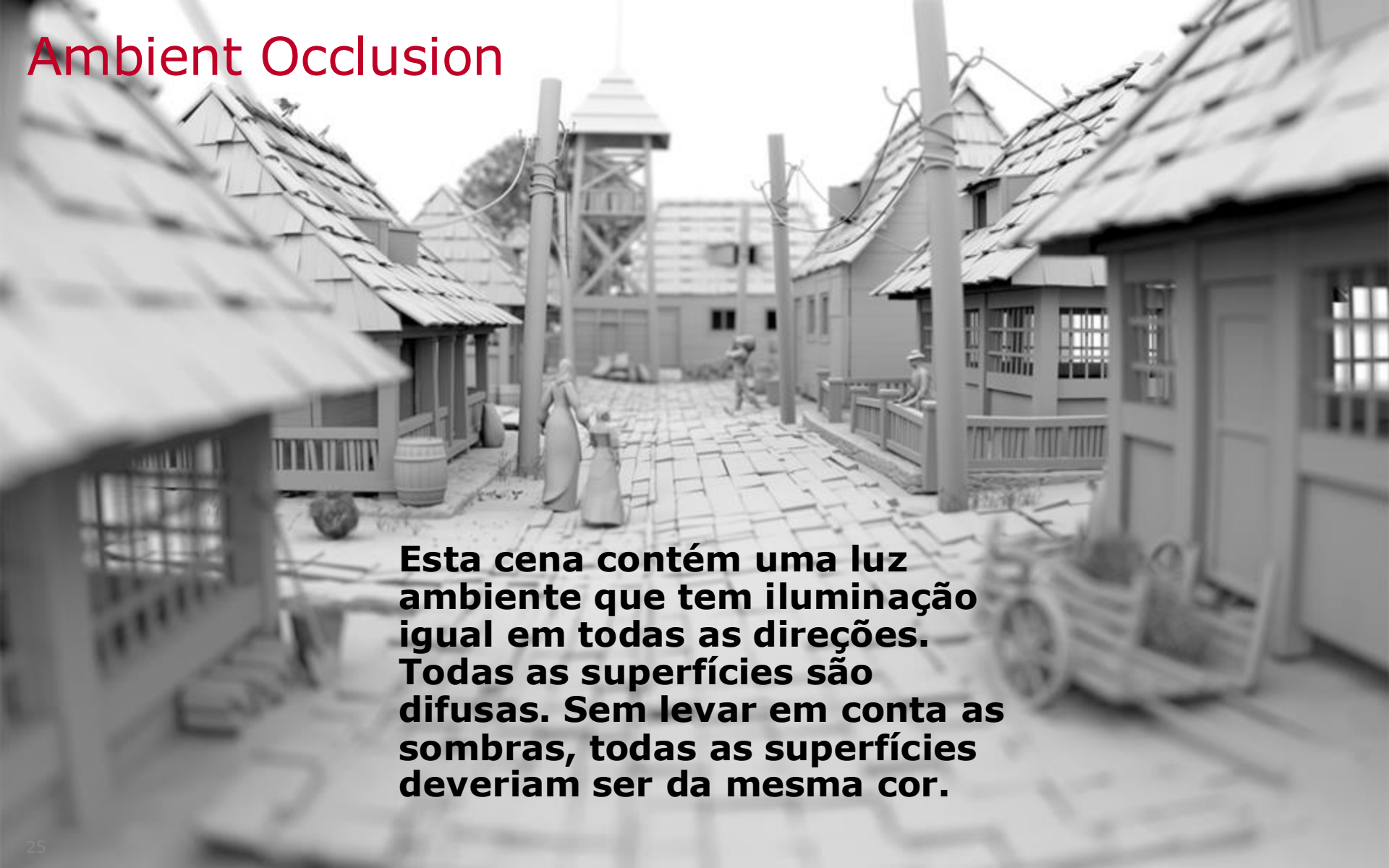
(criadas por uma luz pontual)



Soft shadows

(criadas por uma luz não pontual)

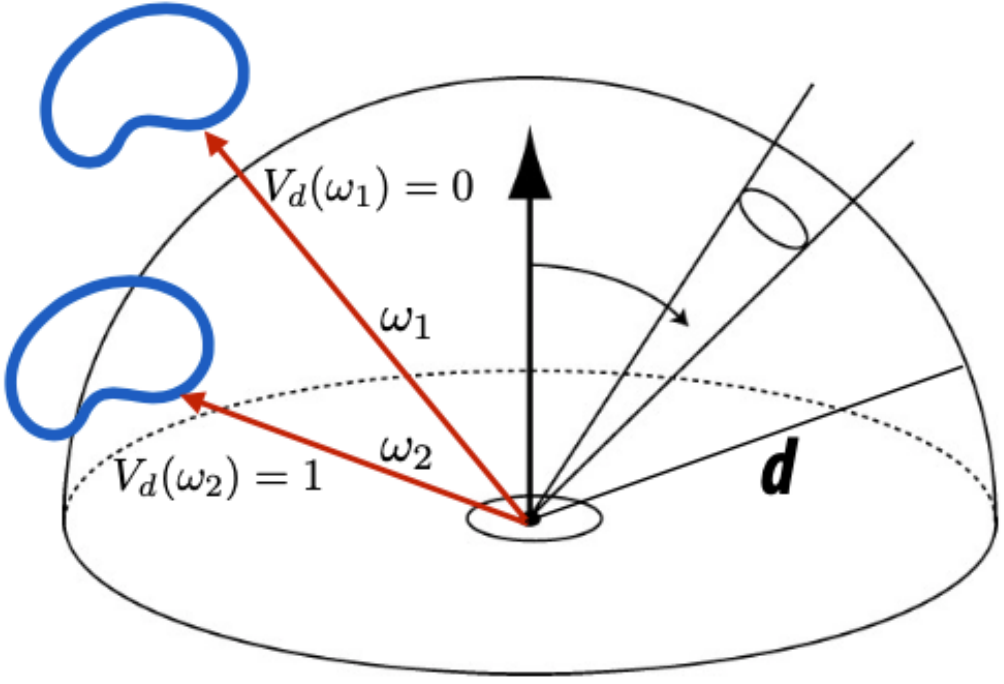
Ambient Occlusion



Esta cena contém uma luz ambiente que tem iluminação igual em todas as direções. Todas as superfícies são difusas. Sem levar em conta as sombras, todas as superfícies deveriam ser da mesma cor.

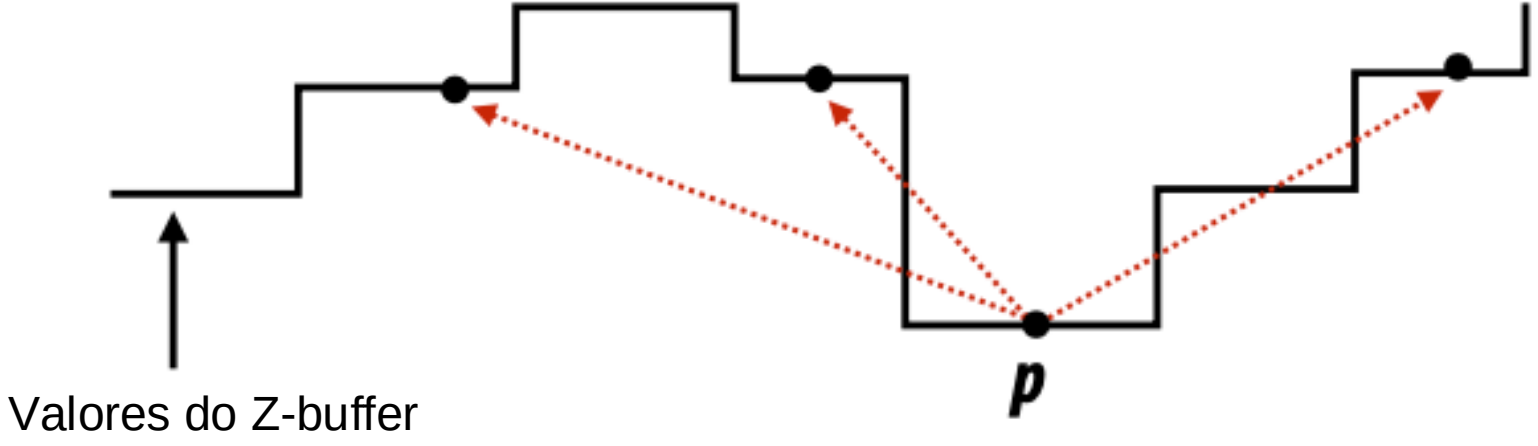
Ambient Occlusion

Idéia: Pré-calcule a "fração de hemisfério" que está ocluída (escondida) até uma distância d de um ponto. Ao sombrear, atenue a iluminação na quantidade proporcional.



Ambient Occlusion usando o "espaço da tela"

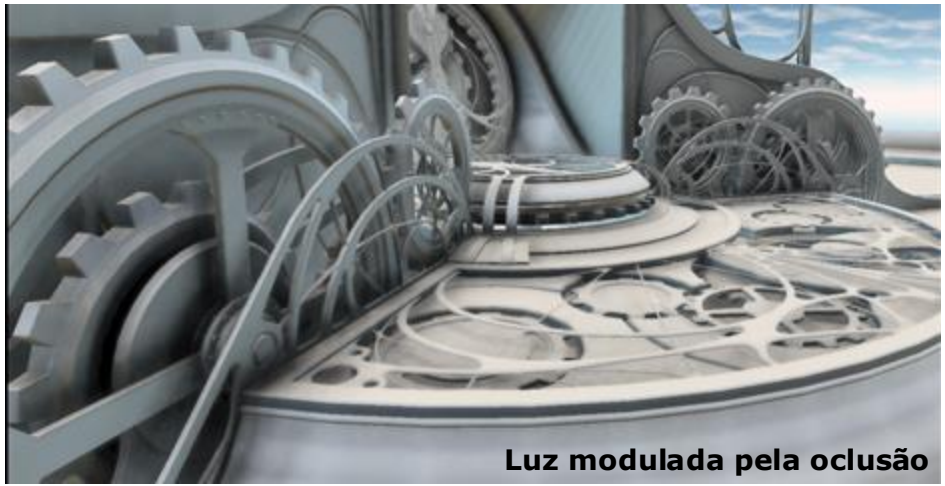
1. Renderizar a cena com o buffer de profundidade (z-buffer)
2. Para cada pixel p (estimar a oclusão do hemisfério traçando raios na vizinhança do buffer de profundidade)
3. Faça alguma média do mapa de oclusão para reduzir o ruído
4. Escureça a iluminação ambiente por quantidade de oclusão



Buda sem Ambiente Occlusion

Buda com Ambiente Occlusion

Ambient Occlusion



Insper

Shading Pré Computado

Shading Simples

Ambient Occlusion
texture map

com Ambient
Occlusion

Insper

Computação Gráfica

Luciano Soares
<lpsoares@insper.edu.br>

Fabio Orfali
<fabioo1@insper.edu.br>

Gustavo Braga
<gustavobb1@insper.edu.br>