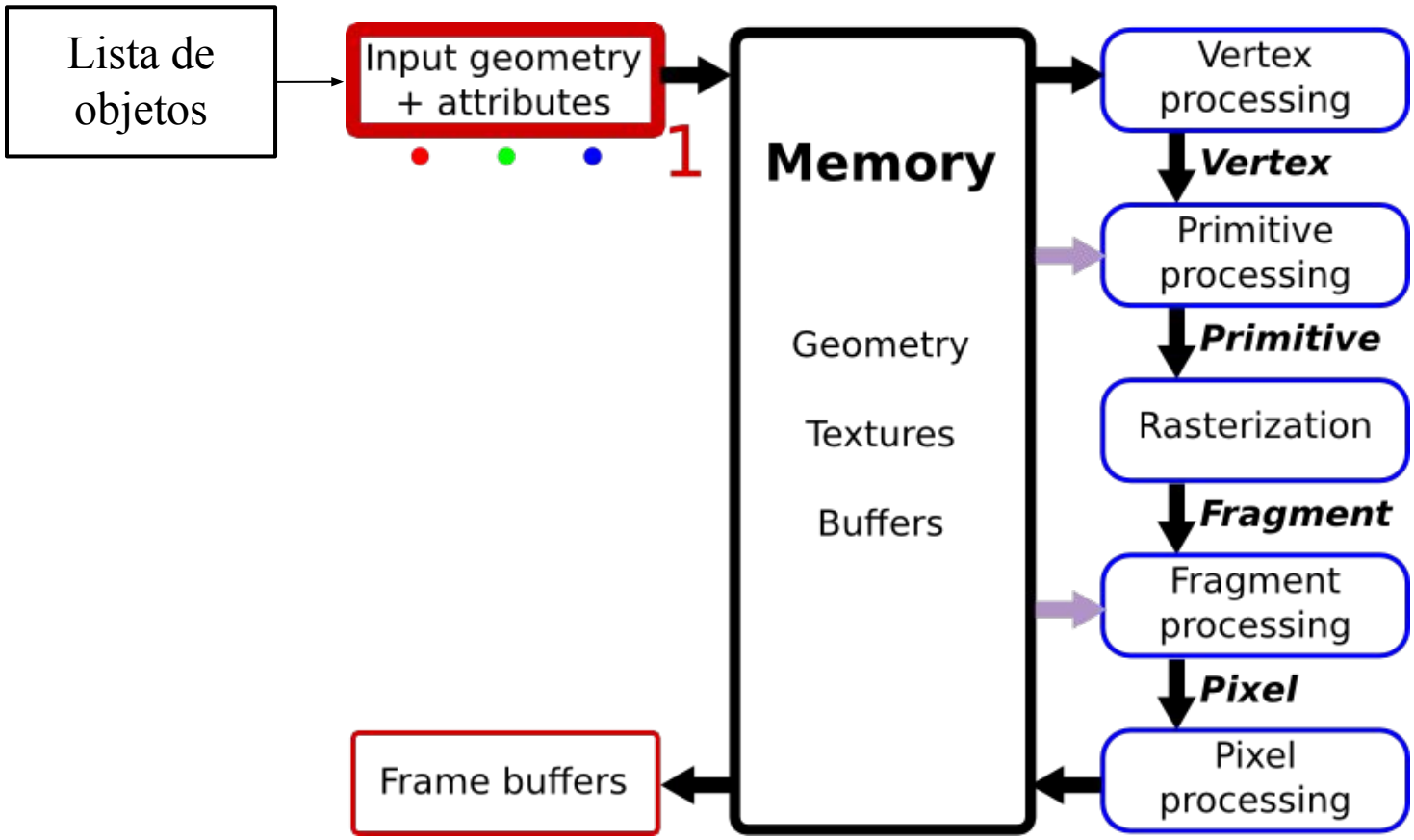
Algoritmos de Visibilidade

Computação Gráfica

Visibilidade

- *Algoritmos básicos de visibilidade*
  - *Algoritmo pintor*
  - *Z-buffer*
  - *Forward Ray-casting*
  - *Backward Ray-casting*

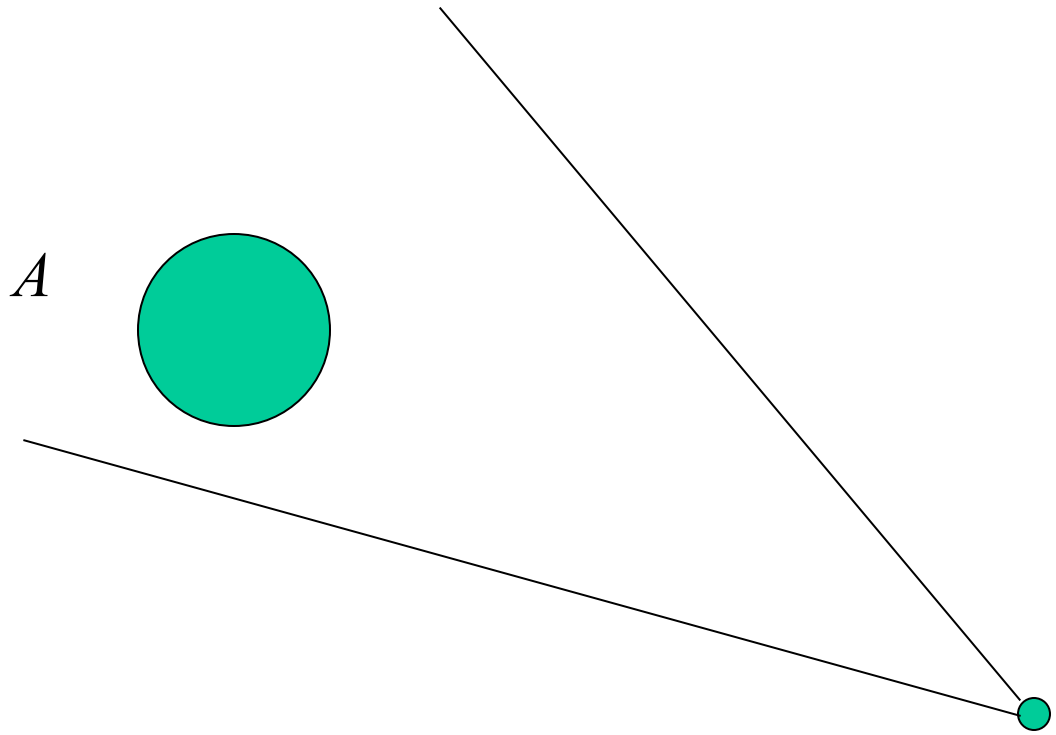
Pipeline de renderização



O Problema de Visibilidade

- *Dados:*
  - *Conjunto (ou lista) de objetos*
    - *Geometria*
    - *Fotometria*
  - *Imagem na tela (estrutura de dado na memória)*
    - *Matriz (de pixels) com origem no canto inferior esquerdo*
- *Problema:*
  - *Como pintar na imagem os objetos vistos?*

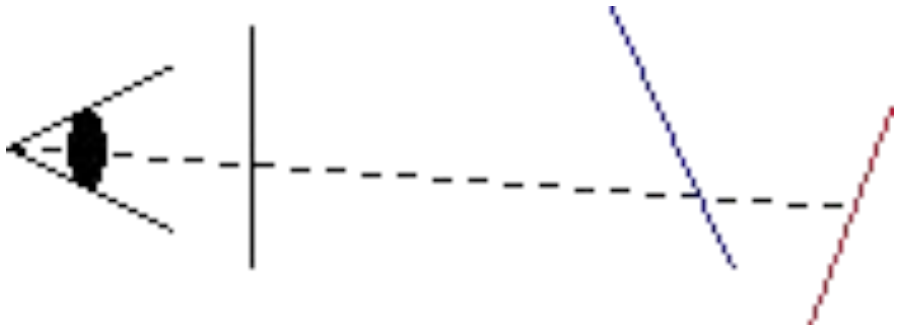
Desenhando apenas um objeto



Desenhando apenas um objeto

```
void draw_3d_object_A {
    Para cada pixel da imagem {
        verifica se pixel intersecta object_A;
        caso intersecte pinta pixel na cor da superfície;
        caso não intersecte pinta pixel da cor de fundo;
    }
}
```

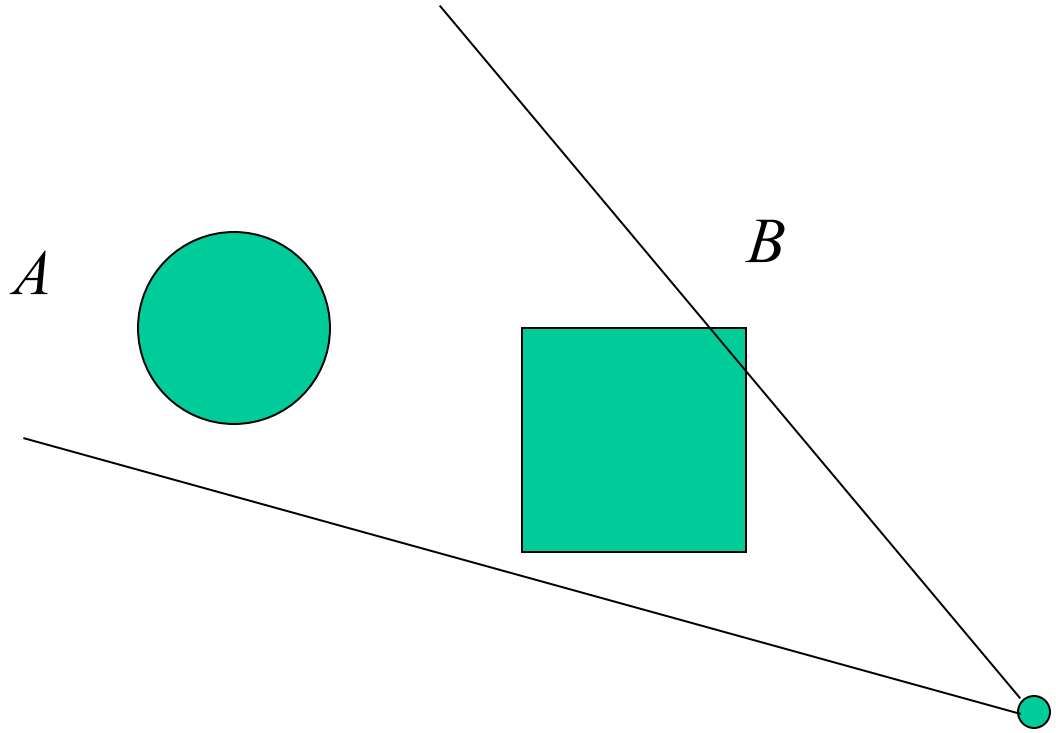
O Problema de Visibilidade



- *Qual é a superfície mais próxima, vista em um ponto na imagem?*

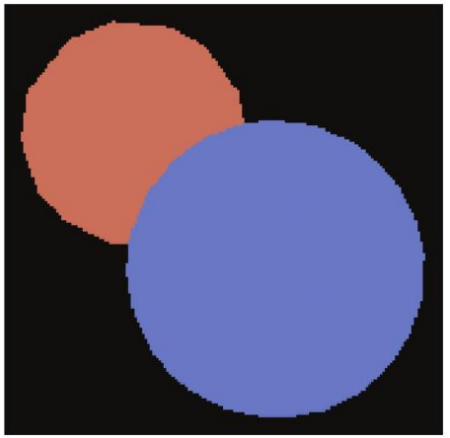
- *Como você resolveria este problema?*

Removendo superfícies ocultas

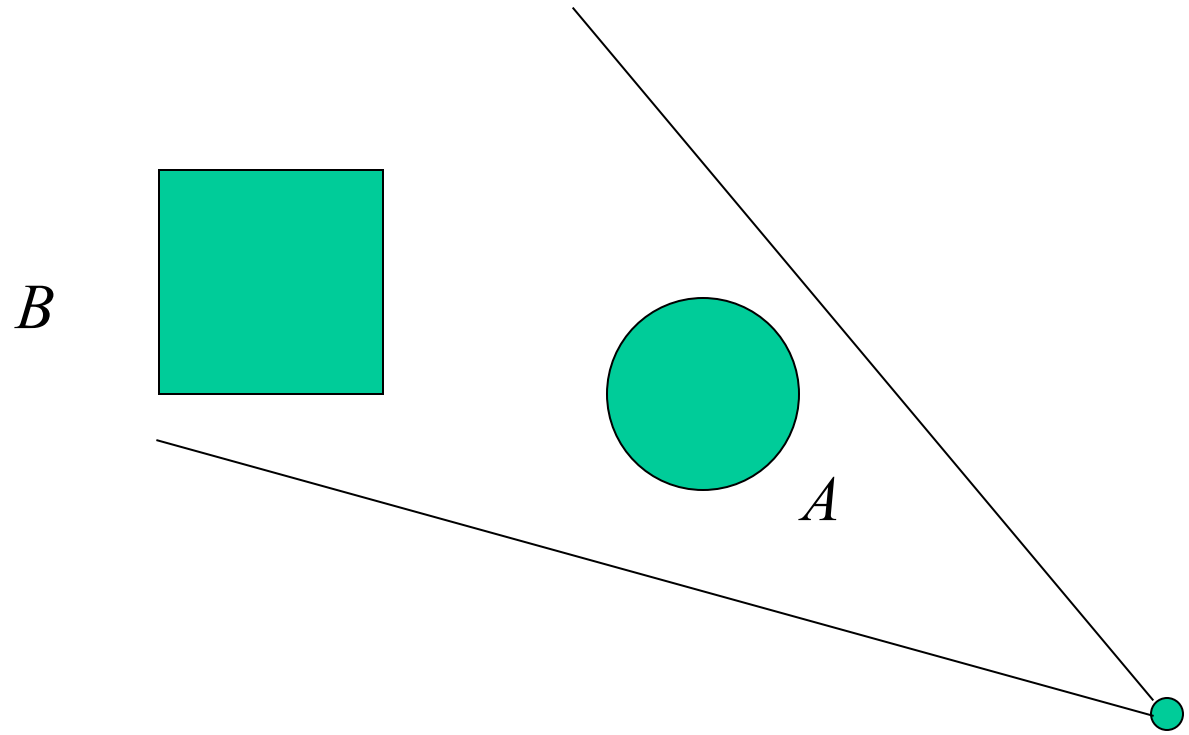


Removendo superfícies ocultas

- *while (1) {*
- *get_viewing_point_from_mouse_position();*
- *glClear(GL_COLOR_BUFFER_BIT);*
- *draw_3d_object_A();*
- *draw_3d_object_B();*
- *}*



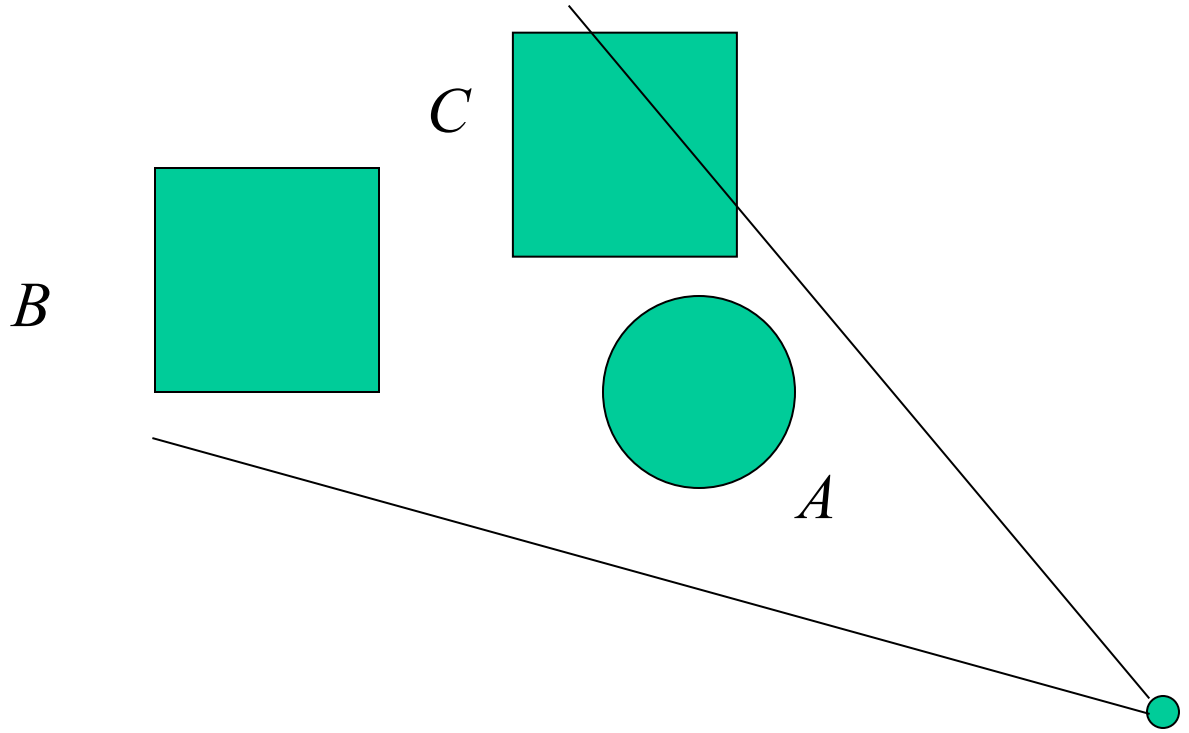
Removendo superfícies ocultas



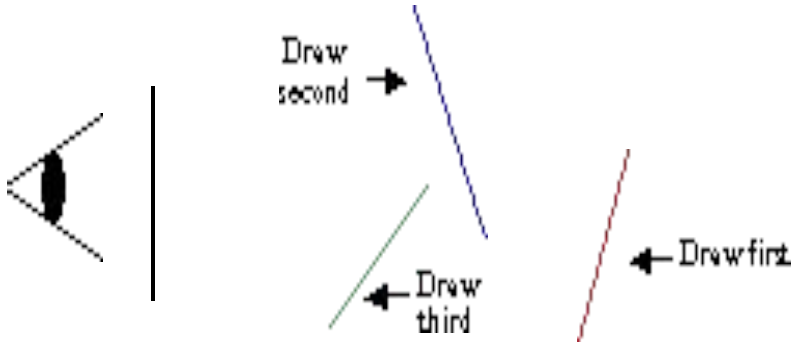
Removendo superfícies ocultas

- *while (1) {*
- *get_viewing_point_from_mouse_position();*
- *glClear(GL_COLOR_BUFFER_BIT);*
- *draw_3d_object_B();*
- *draw_3d_object_A();*
- *}*

Mas no caso geral?



Algoritmo Pintor (painter)



- *Sort objects by depth (Z)*
- *Loop over objects in back-to-front order*
- *Project to image*
- *scan convert: image[x,y] = shade(x,y)*

Sorting Objects by Depth

Depth ordering is a *partial ordering*. Outcomes are

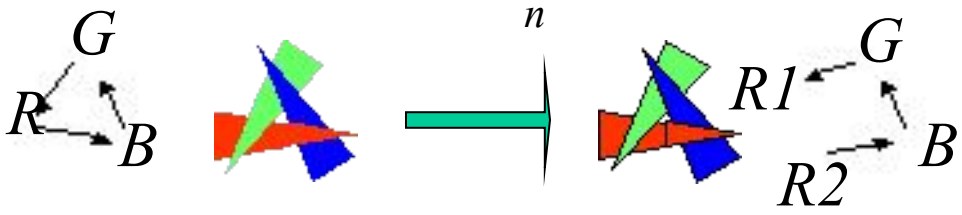
A occludes B
draw B before A

B occludes A
draw A before B

don't care
draw in either order

Sorting objects by their zmin doesn't always work! (same for zmax)

Sometimes ordering is cyclic! What to do? Split objects!



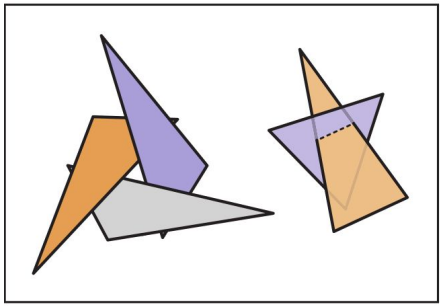
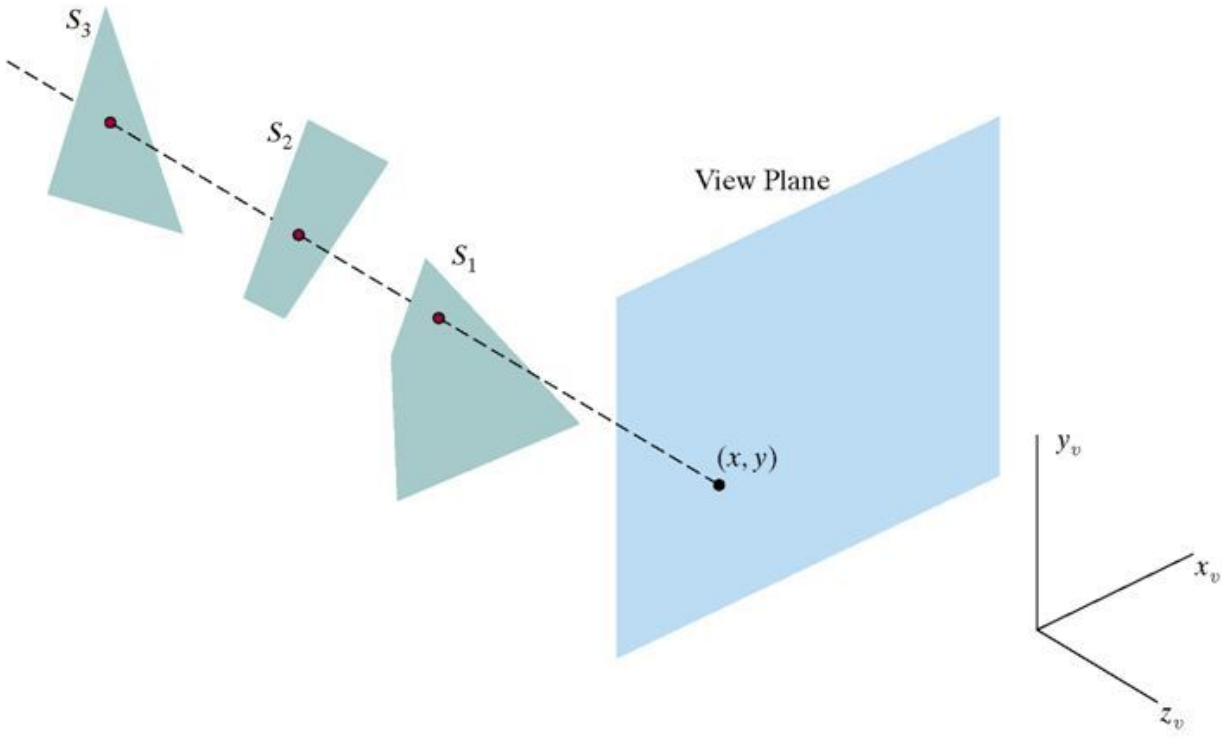


Figure 8.9. Two occlusion cycles, which cannot be drawn in back-to-front order.

Painter´s Algorithm

- ***Pontos fortes***
  - ***Simplicidade: desenha objetos um de cada vez, rasteriza cada um***
  - ***Trabalha bem com transparência***
- ***Problemas***
  - ***Ordenar pode ser caro (complexidade n log(n) do número de objetos)***
  - ***Ruim quando ordenar fica cíclico, necessidade de dividir***
  - ***Polígonos que se interpenetram, precisa dividir também***
  - ***Difícil de ordenar para objetos não poligonais (linhas, pontos)***
- ***Algumas vezes, não é necessário ordenar***
  - ***Se os objetos estão em uma grade (triângulos modelando terreno), num campo de alturas z(x,y),***
- ***Quem usa?***
  - ***Interpretadores PostScript***
  - ***OpenGL, se não habilitar Z-Buffer - glEnable(GL_DEPTH_TEST);***

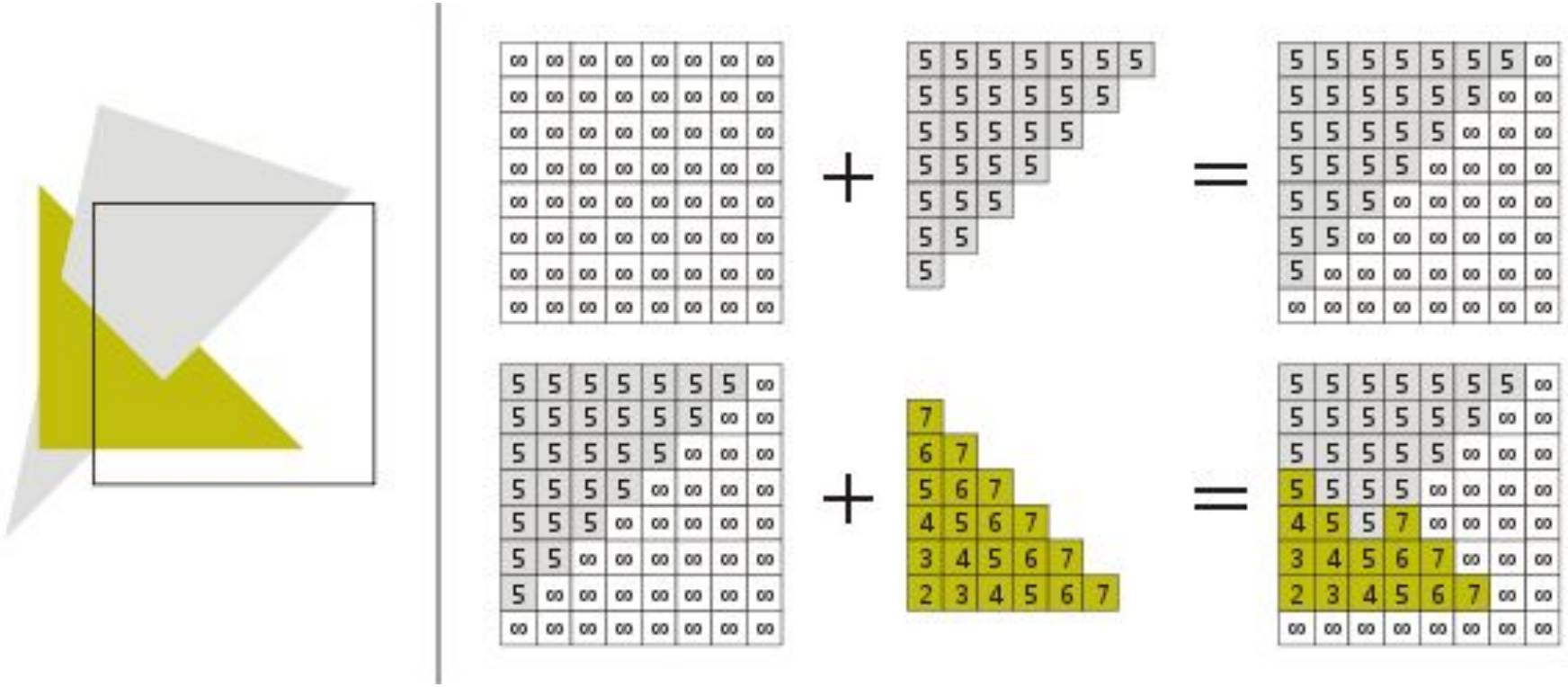
Algoritmo Z-Buffer



Algoritmo Z-Buffer

- ***Inicialização:***
- ***loop over all x,y***
- ***zbuf[x,y] = infinity***
- ***Rasterização:***
- ***loop over all objects***
- ***scan convert object (loop over x,y)***
- ***if z(x,y) < zbuf[x,y]*** ***compute z of this object***
- ***at this pixel & test***
- ***zbuf[x,y] = z(x,y)*** ***update z-buffer***
- ***image[x,y] = shade(x,y)*** ***update image (typically RGB)***

Algoritmo Z-Buffer



Algoritmo Z-Buffer

- ***Pontos fortes:***
  - ***Simples, não há necessidade de dividir***
  - ***Fácil de mixar com polígonos, esferas, e outras primitivas geométricas***
- ***Pontos fracos***
  - ***Não consegue lidar bem com transparência***
  - ***Precisa de uma boa resolução do Z-buffer ou gera artefatos (ordem dos objetos)***
    - ***No OpenGL, a resolução pode ser controlada pela escolha dos planos de corte (Near, Far) e o número de bits para profundidade)***
    - ***Escolha razão de profundidade entre os planos de corte (zfar/znear) para ser a menor possível (mais precisão no Z-buffer)***
- ***Quem usa?***
  - ***OpenGL, se habilitar Z-Buffer - glEnable(GL_DEPTH_TEST);***

Usando o buffer de profundidade

- ***glutInitDisplayMode (GLUT_DEPTH | ....);***
- ***glEnable(GL_DEPTH_TEST);***
- ***...***
- ***while (1) {***
- *** glClear(GL_COLOR_BUFFER_BIT |***
- *** GL_DEPTH_BUFFER_BIT);***
- *** get_viewing_point_from_mouse_position();***
- *** draw_3d_object_A();***
- *** draw_3d_object_B();***
- ***}***

Ray-Tracing

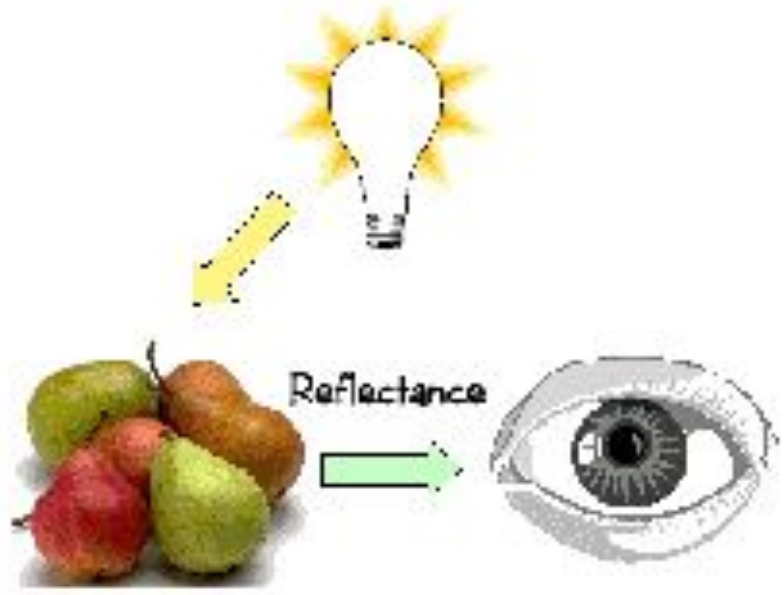
- *Luz é um feixe de fotons emitidos ou refletidos*
- *Fontes enviam fotons em todas as direções*
  - *Modelar como partículas que rebatem nos objetos da cena*
  - *Cada foton tem um comprimento de onda e energia (cor e intensidade)*
  - *Quando rebatem, parte da energia é absorvida, refletida e transmitida (refração).*

Princípio do Ray-Tracing

- *Seguir cada fóton (da fonte) até que:*
  - *Toda sua energia seja absorvida (depois de muitas reflexões)*
  - *Ele saia do universo conhecido (Frustum)*
  - *Ele bata no plano imagem e sua contribuição pode ser adicionada ao pixel respectivo*

Forward Ray tracing

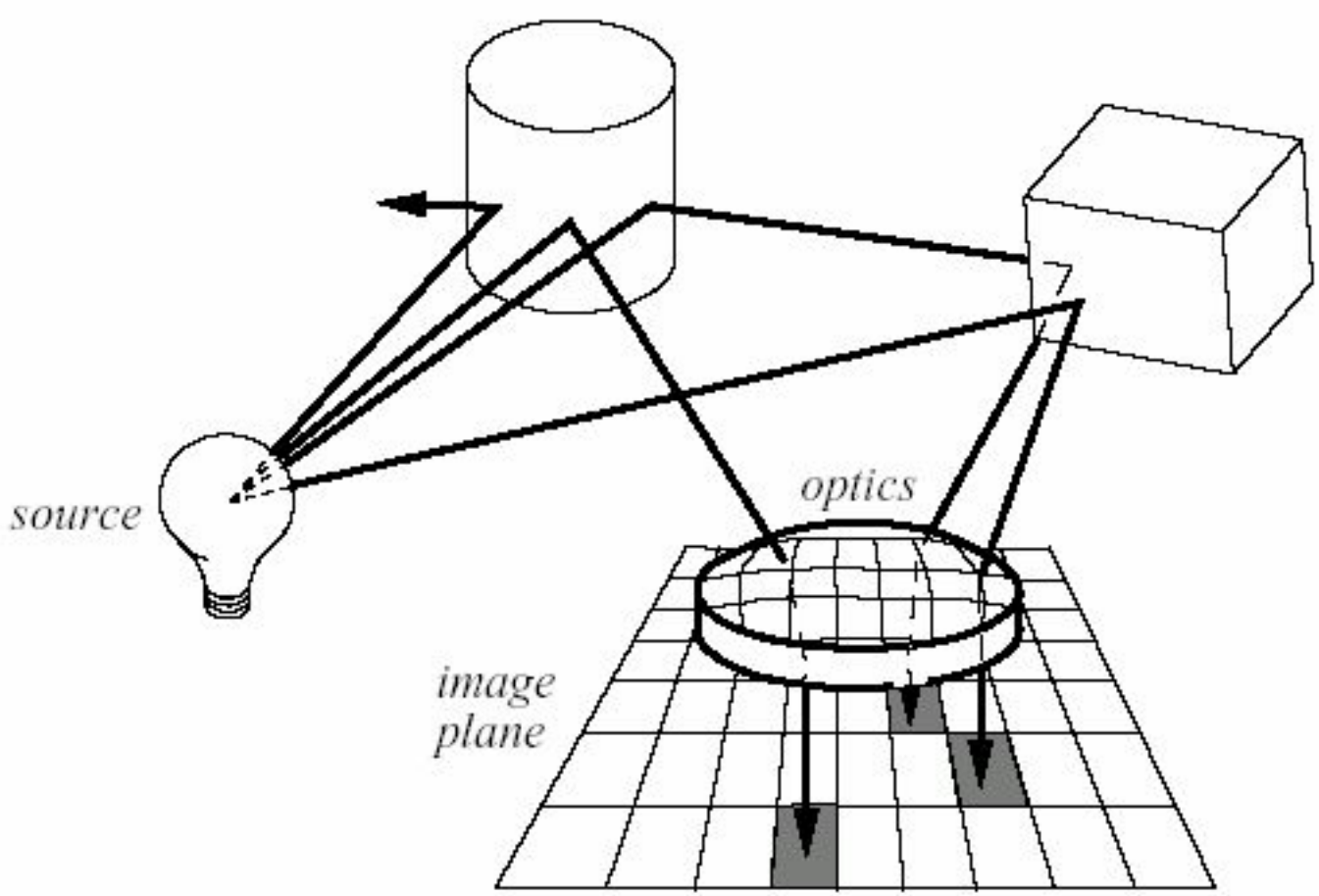
- *Raios são os caminhos desses fótons*
- *Este método de renderização seguindo os caminhos dos fótons é denominado de Ray-tracing*
- *Forward ray-tracing segue fótons na direção que a luz viaja*



Forward ray-tracing

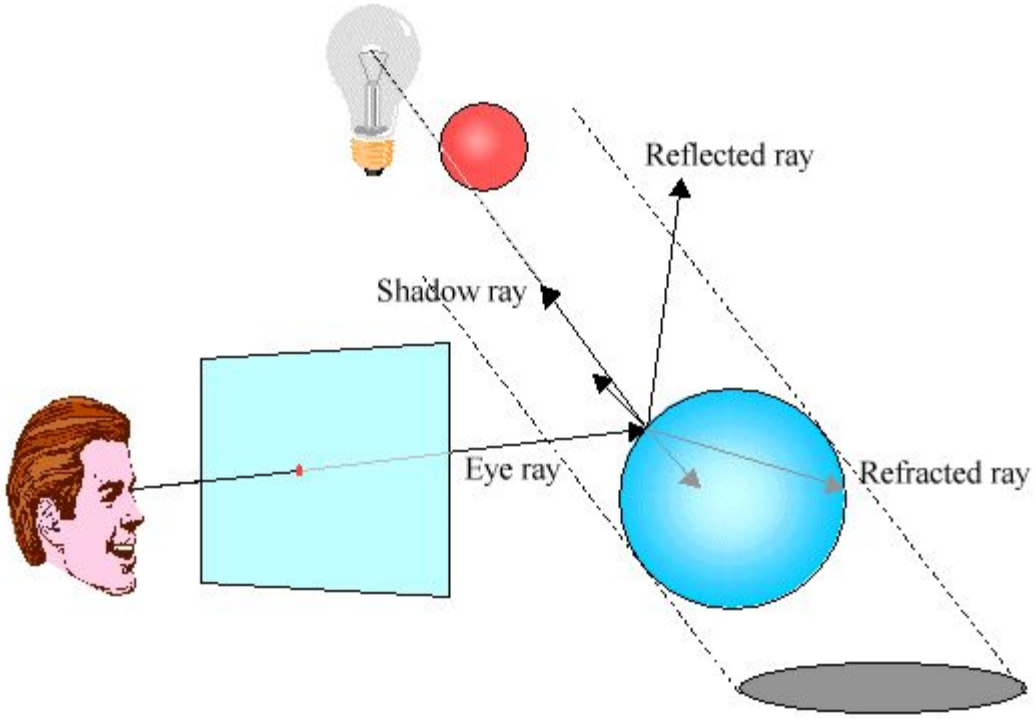
- *Grande problema:*
  - *Apenas uma pequena porção de raios alcançam a imagem*
  - *Extremamente lento calcular*
- *Cenário ideal:*
  - *Saber magicamente quais os raios que contribuem para a formação da imagem e traçar apenas esses*

Raios que chegam à imagem?



Backward ray tracing

Iniciar da imagem e fazer o caminho do raio traçado ao contrário



Backward ray tracing

- *Idéias básicas:*
  - *Cada pixel recebe luz de uma direção (raio definido pelo ponto imagem e ponto focal)*
  - *Qualquer fóton contribuindo para o pixel vem dessa direção*
  - *Então aponte nesta direção e encontre o que está enviando luz*
  - *Se encontrar a fonte de luz, parou (contribuição alta)*
  - *Se não encontrar nada, parou (contribuição zero)*
  - *Se bater numa superfície, veja de onde ela está iluminada*
- *Ao final, é forward raytracing, mas apenas para os raios que contribuem para a formação da imagem*
- *Mais conhecido como* ***Ray-Casting***

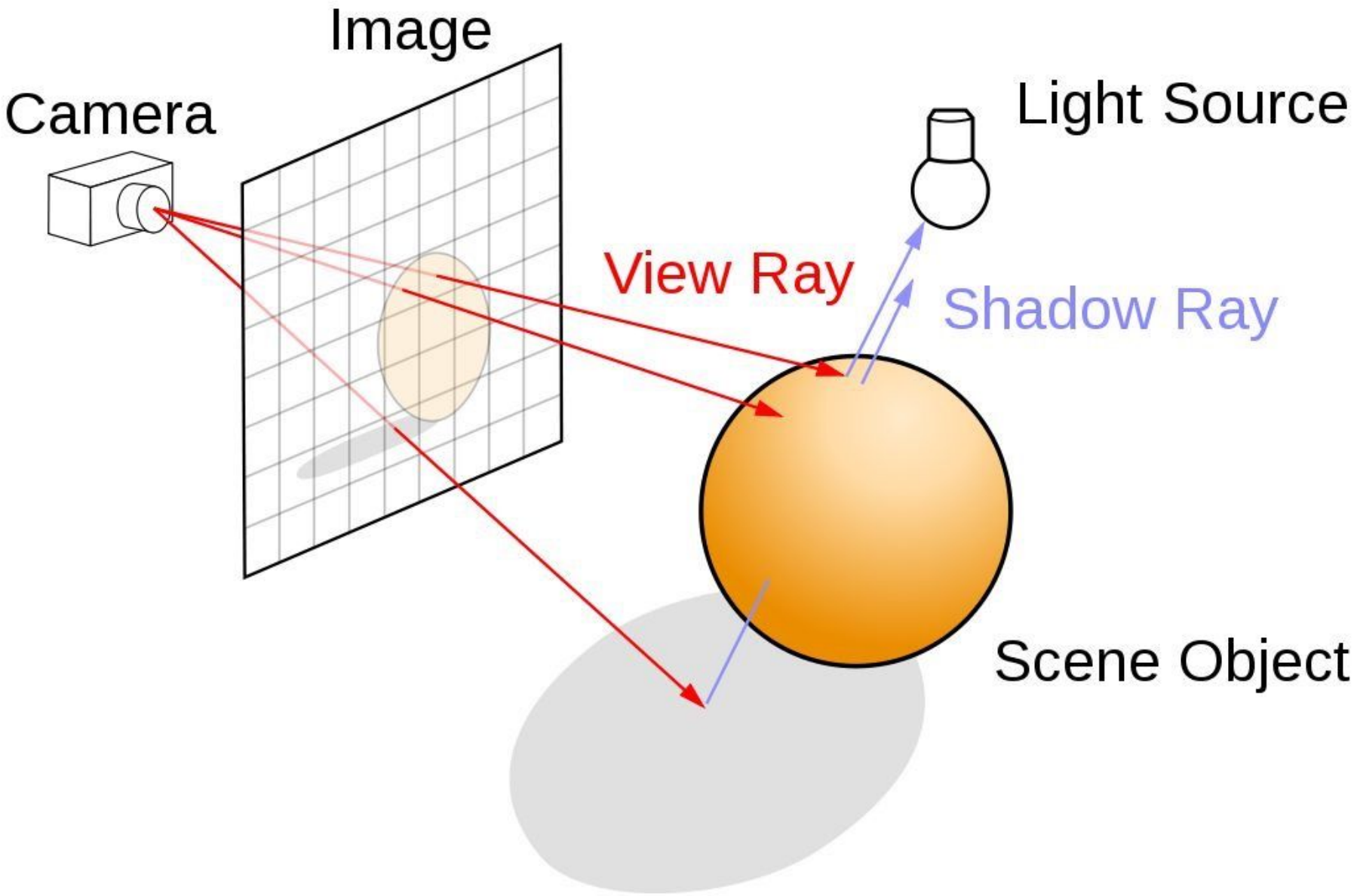


Image
Camera
Light Source
View Ray
Shadow Ray
Scene Object

Ray casting

- *Um algoritmo de visibilidade bem flexível*
- *loop y*
- *loop x*
- *shoot ray from eye through pixel (x,y) into scene*
- *intersect with all surfaces, find first one the ray hits*
- *shade surface point to compute pixel (x,y)'s color*
- *Dá para simular sombras (shadow buffer).*

Ray casting

- *Equação do raio é p+td: p é sua origem, e d sua direção*

  t=0 na origem do raio, t>0 na direção positiva

  – *Tipicamente, assume-se ||d||=1*

  – *p e d são tipicamente calculados no sistema de coordenadas de mundo*

- *Isto pode ser facilmente generalizado para implementação do Ray-Tracing recursivo.*

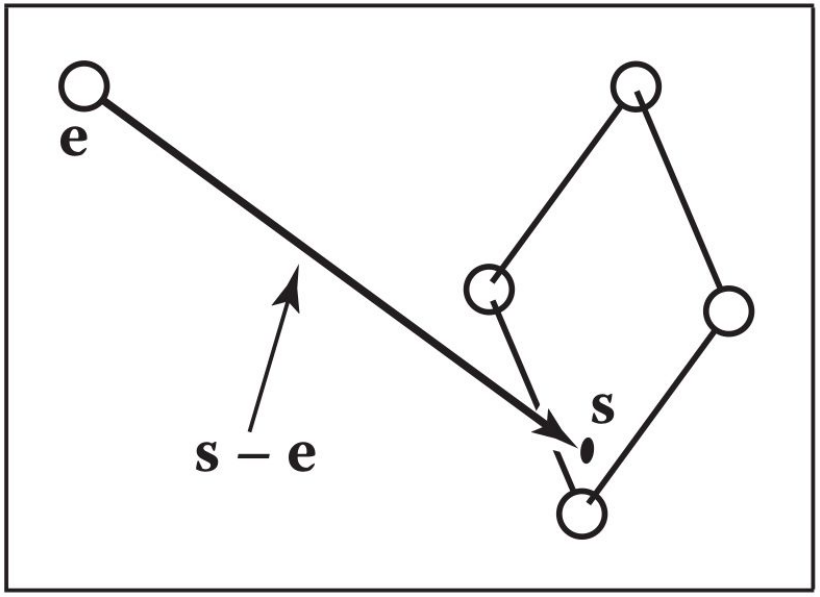
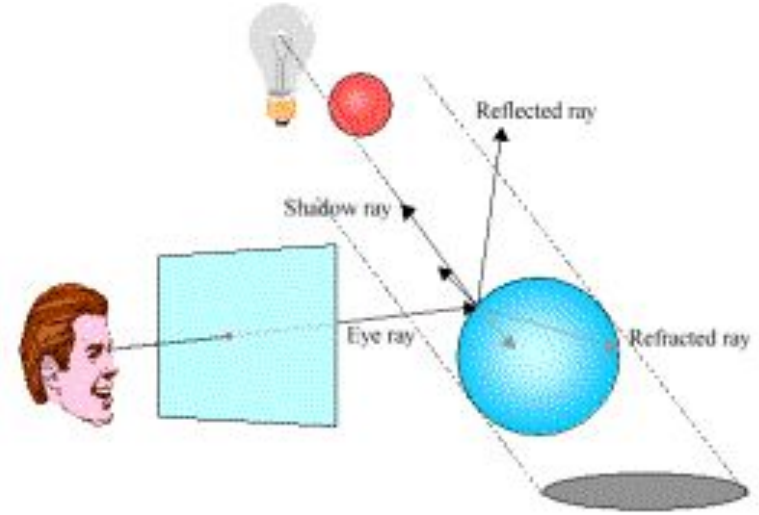


Figure 4.6. The ray from the eye to a point on the image plane.

Ray tracing recursivo

- *Distingue-se 4 tipos de raio:*
  - *Eye rays: orina-se no olho*
  - *Shadow rays: do ponto na superfície na direção da luz*
  - *Reflection rays: do ponto na superfície na direção refletida*
  - *Transmission rays: do ponto na superfície na direção refratada*
  - *Traçar todos estes recursivamente.*



Writing a simple ray caster

```
Raycast() { //Generate a picture
    for each pixel x,y {
        color(pixel) = Trace(ray_trhough_pixel(x,y));
    }
}
Trace(ray) { //fire a ray, return radiance (color/illumination formula)
    object_point = Closest_intersection(ray);
    if(object_point) return Shade(object_point, ray);
    else return Background_Color;
}
```

Writing a simple ray caster

```
Closest_intersection(ray, &associated_information) {
    closest_point_intersection = infinite;
    for each surface in scene {
        if (intersection(ray, surface)< closest_point_intersection)
            update closest_point_intersection;  //(also other information as normal,
                material properties, and so on, put in *info)
    }
    return closest_point_intersection; //(and associated_information)
}
Shade(point, ray) {//return radiance of light leaving point
    calculate surface normal (N) and other vectors (L,O,R);
    use Phong illumination (or similar) to calculate contributions to each light source;
    return contribution at the image corresponding pixel;
}
```

Algoritmos de Visibilidade

Painter's

```
sort objects by z (back-to-front)
loop objects
        loop y
                loop x
                        write pixel
```

Algoritmos de Visibilidade

Z-buffer

```
initialize z-buffer
loop objects
        loop y
                loop x
                        if z(x,y) < zbuf[x,y]
                                zbuf[x,y] = z(x,y)
                                write image pixel
```

Algoritmos de Visibilidade

Ray Casting

```
loop y
        loop x
                loop objects
                        find object with min z
                write pixel
```

Comparing visibility algorithms

- ***Painter's:***
  - ***Implementation:*** *moderate to hard if sorting & splitting needed*
  - ***Speed:*** *fast if objects are pre-sorted, otherwise slow*
  - ***Generality:*** *sorting & splitting make it ill-suited for general 3-D rendering*
- ***Z-buffer:***
  - ***Implementation:*** *moderate, it can be implemented in hardware*
  - ***Speed:*** *fast, unless depth complexity is high*
  - ***Generality:*** *good but won't do transparency*
- ***Ray Casting:***
  - ***Implementation:*** *easy, but hard to make it run fast*
  - ***Speed:*** *slow if many objects: cost is O((#pixels)´(#objects))*
  - ***Generality:*** *excellent, can even do CSG, transparency, shadows*

Ray surface intersections

- *Ray equation: (given origin p and direction d) $x(t) = p+td$*
  - **Surfaces can be represented by:**
    - **Implicit functions:** $f(\mathbf{x}) = 0$
    - **Parametric functions:** $\mathbf{x} = \mathbf{g}(u,v)$
- *Compute Intersections:*
  - *Substitute ray equation for x*
  - *Find roots*
  - *Implicit: $f(p_0 + td) = 0$*
  - *» one equation in one unknown – univariate root finding*
  - *Parametric: $p_0 + td - g(u,v) = 0$*
  - *»three equations in three unknowns (t,u,v) – multivariate root finding*
  - *For univariate polynomials, use closed form soln. otherwise use numerical root finder*

Ray sphere intersection

- *Ray-sphere intersection is an easy case*
- *A sphere's implicit function is:* $x-x_c^2+y-y_c^2+z-z_c^2-r^2=0$ *if sphere at origin*
- *The ray equation is:* $x = x_0 + t\,x_d$
- $y = y_0 + t\,y_d$
- $z = z_0 + t\,z_d$
- *Substitution gives:* $(x_0+t\,x_d)^2 + (y_0+t\,y_d)^2 + (z_0+t\,z_d)^2 - r^2 = 0$
- *A quadratic equation in t.*
- *Solve the standard way:* $A = x_d^2+y_d^2+z_d^2 = 1$ *(unit vec.)*
- $B = 2(x_0x_d+y_0y_d+z_0z_d)$
- $C = x_0^2+y_0^2+z_0^2 - r^2$
- *Quadratic formula has two roots:* $t=(-B\pm\sqrt{B^2-4C})/2A$
  - *which correspond to the two intersection points*
  - *negative discriminant means ray misses sphere*

Equação da esfera

$$\|\mathbf{x} - \mathbf{c}\|^2 = r^2$$

- $\mathbf{x}$: points on the sphere
- $\mathbf{c}$: center point
- r : radius of the sphere

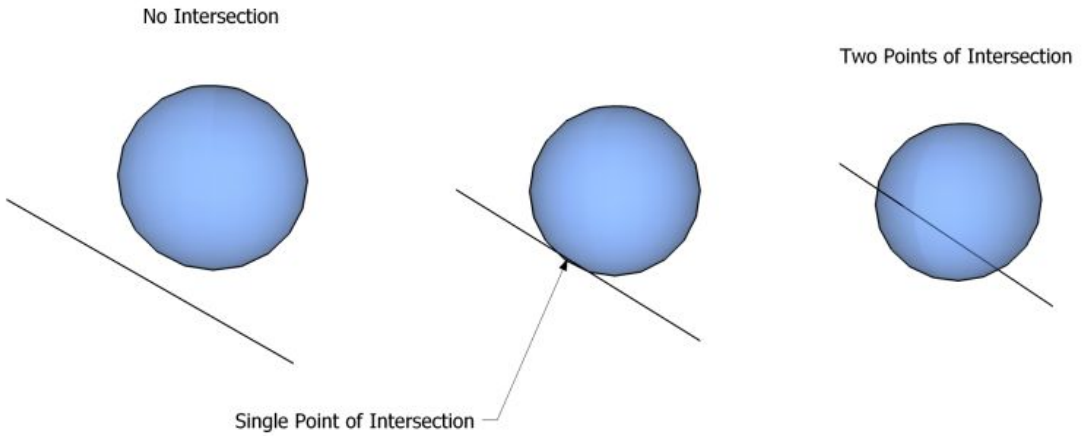
Equação da reta

$$\mathbf{x} = \mathbf{o} + d\hat{\mathbf{u}}$$

- $\mathbf{x}$: points on the line
- $\mathbf{o}$: origin of the line
- d : distance from the origin along the line
- $\hat{\mathbf{u}}$: direction of line (a unit vector)

$$d = -[\hat{\mathbf{u}} \cdot (\mathbf{o} - \mathbf{c})] \pm \sqrt{\nabla}$$
$$\nabla = [\hat{\mathbf{u}} \cdot (\mathbf{o} - \mathbf{c})]^2 - (\|\mathbf{o} - \mathbf{c}\|^2 - r^2)$$

- If $\nabla < 0$, then it is clear that no solutions exist, i.e. the line does not intersect the sphere (case 1).
- If $\nabla = 0$, then exactly one solution exists, i.e. the line just touches the sphere in one point (case 2).
- If $\nabla > 0$, two solutions exist, and thus the line touches the sphere in two points (case 3).



Ray plane intersection

- *Raio: $p = p_0 + td$*
  - *p_0 é a origem do raio, d é a direção (d_x, d_y, d_z)*
  - *$p = (x,y,z)$ é um ponto sobre o raio (para $t>0$)*
- *Plano: (p-q).n=0 (equação ponto-normal)*
  - *q é um ponto de referência no plano, n é a normal, p é um ponto no plano (x,y,z)*
    - *se tiver 3 pontos, ache dois vetores e determine n como o produto cruzado dos dois,*
  - *re-escreva como p.n+D=0 (equivalente a Ax+By+Cz+D=0, onde D = q.n)*

Raio –plano (interseção)

- *Substitua raio na equação do plano (uma equação, uma incógnita)*
- *Solução: $t=-(p.n+D)/d.n$*
  - *Se $d.n=0$ => raio é paralelo ao plano*
  - *Se $t<0$ => interseção está atrás da origem*
  - *Se $t>0$ => substitua ele na equação do raio para achar o ponto no plano*
- *Então basta testar ponto dentro de polígono*

Ray polygon intersection

- *Assuming we have a planar polygon*
  - *first, find intersection point of ray with plane*
  - *then check if that point is inside the polygon*
- *Latter step is a point-in-polygon test in 3-D:*
  - *inputs: a point x in 3-D and the vertices of a polygon in 3-D*
  - *output: INSIDE or OUTSIDE*
  - *problem can be reduced to point-in-polygon test in 2-D*
- *Point-in-polygon test in 2-D:*
  - *easiest for triangles*
  - *easy for convex n-gons*
  - *harder for concave polygons*
  - *most common approach: subdivide all polygons into triangles*
  - *for optimization tips, see article by Haines in the book Graphics Gems IV*

Programa exemplo (esfera)

- *//Esfera usando Z-Buffer*
- *#include <GL/gl.h>*
- *#include <GL/glu.h>*
- *#include <GL/glut.h>*

Programa exemplo (esfera)

```
void init(void)  {
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[] = { 50.0 };
   GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}
```

Programa exemplo (esfera)

```
void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glutSolidSphere (1.0, 20, 16);
   glFlush ();
}
```

Programa exemplo (esfera)

```
void reshape (int w, int h)
{
  glViewport (0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
    glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
      1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
  else
    glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
      1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
```

Programa exemplo (esfera)

```
int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize (500, 500);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);
  init ();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;
}
```

Resultado

Comentários

- *Definir vetores normais para todos vértices: glutSolidSphere() faz isso.*
- *Criar, posicionar e habilitar uma (ou mais) fontes de luz: glLightfv(), glEnable(), glLight*()*
- *Ex: GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };*
- *glLightfv(GL_LIGHT0, GL_POSITION, light_position);*
- *default: GL_POSITION é (0, 0, 1, 0), eixo-Z negativo*
- *Selecionar um modelo de iluminação: glLightModel*()*
- *Definir propriedades materiais para os objetos na cena*

Lembre-se

- *Voce pode usar valores defaults para alguns parâmetros, de iluminação; outros devem ser modificados*
- *Não se esqueça de habilitar todas as luzes que pretende usar e também habilitar o cálculo de iluminação*
- *Voce pode usar "display lists" para maximizar eficiência quando é necessário mudar as condições de iluminação*

Computing Z for Z-buffering

- ***How to compute Z at each point for z-buffering?***
- ***Can we interpolate Z?***
  - ***Linear interpolation along edge, along span***
  - ***Not quite. World space Z does not interpolate linearly in image space***
- ***But if we linearly interpolate image space Z, it works!***
  - ***Perspective transforms planes to planes***
  - ***Note that image space Z is a nonlinear function of world space Z***