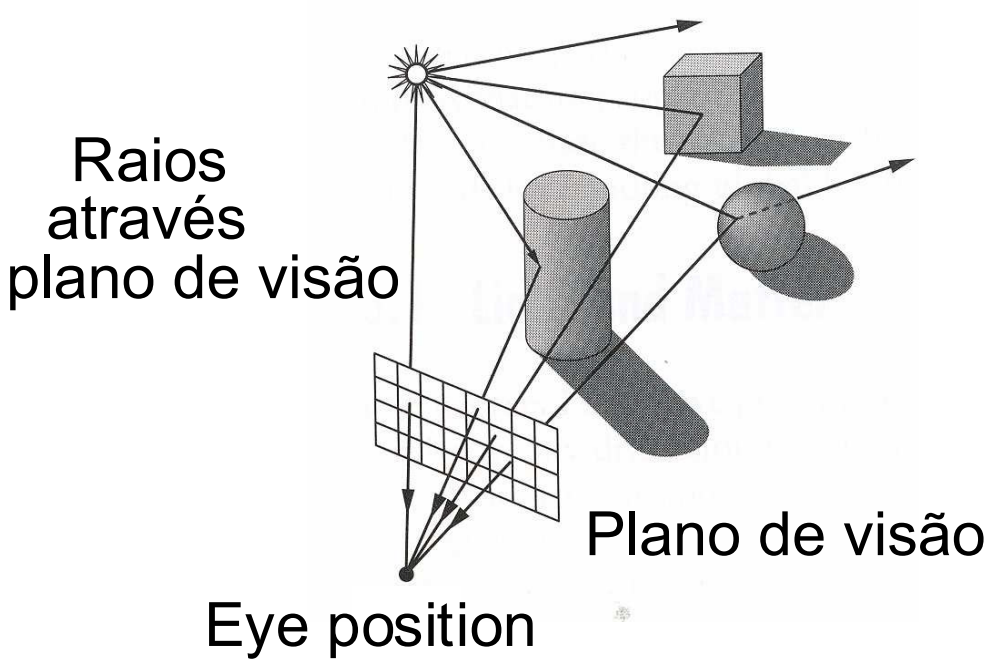
Rendering 3D

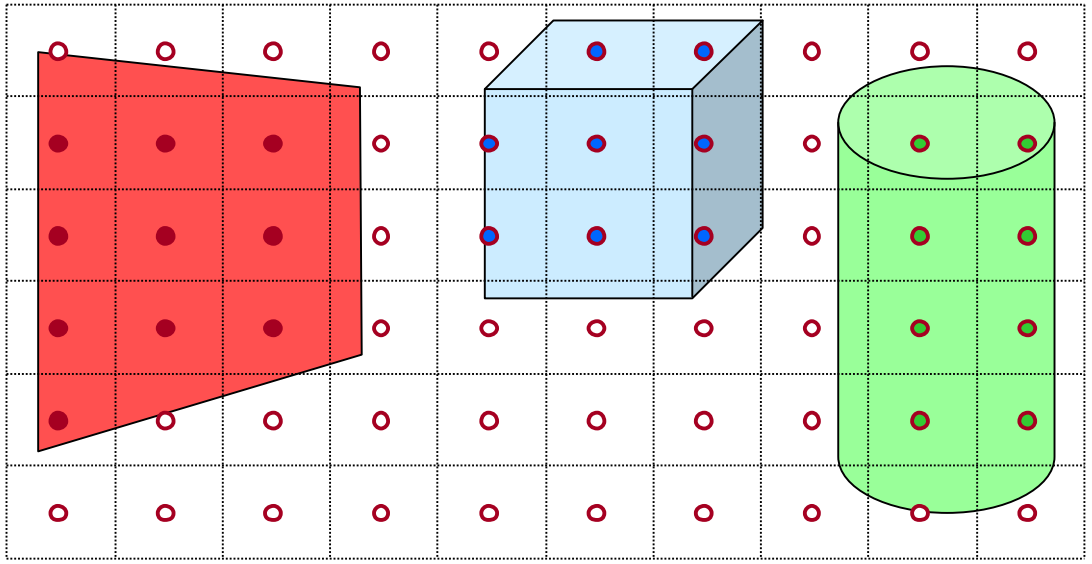
- A cor de cada pixel no plano de visão depende da radiância que emana das superfícies visíveis

Método simples é ray casting



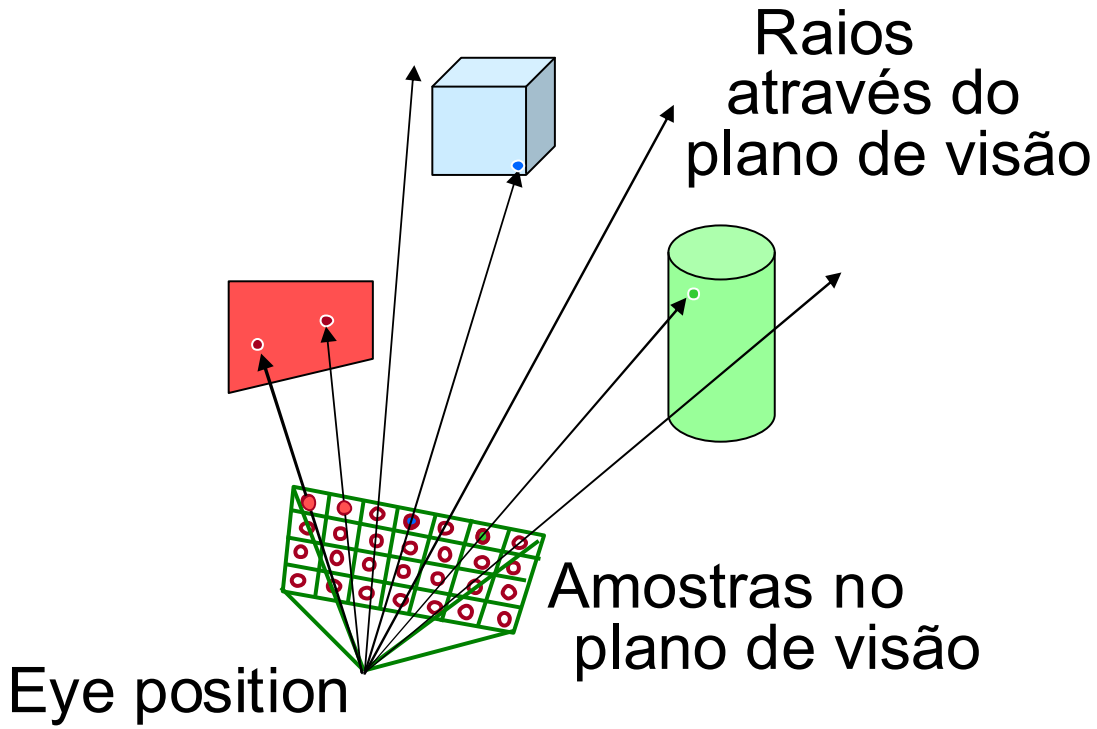
Ray Casting

- Para cada amostra …
  - Construir raio da câmera através do plano de visão
  - Encontrar a 1a. superfície que intersecta o raio
  - Computar a cor da amostra baseada na radiância



Ray Casting

- Para cada amostra …
  - Construir raio da câmera através do plano de visão
  - Encontrar a 1a. superfície que intersecta o raio
  - Computar a cor da amostra baseada na radiância

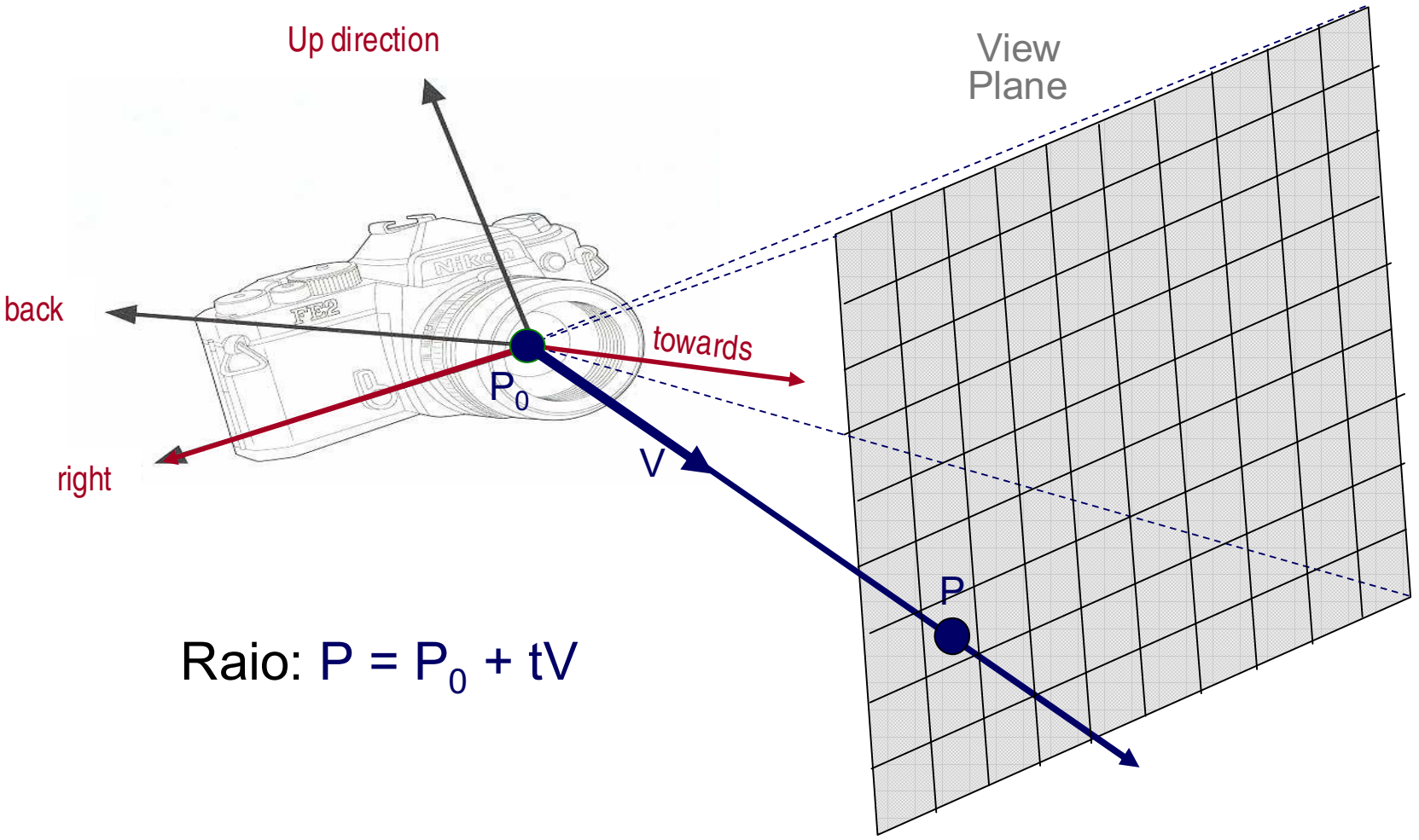


Ray Casting

- Implementação simples:

```
Image Image(Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(scene.camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(scene, ray, hit);
        }
    }
    return image;
}
```

Construindo Raio Através de um Pixel



Raio: $P = P_0 + tV$

Construindo Raio Através de um Pixel

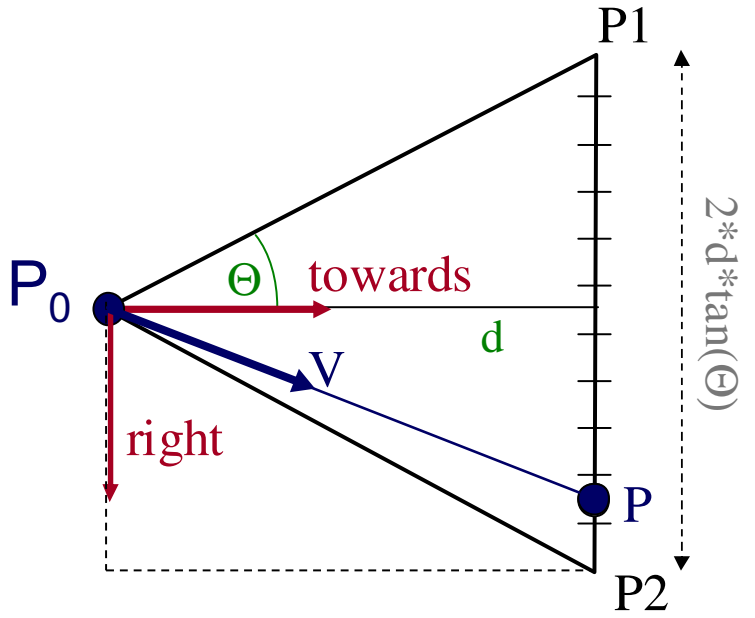
- Exemplo 2D

Θ = metade do ângulo (frustum)
d = distância até o plano de visão

right = towards x up

$P1 = P_0 + d*towards - d*tan(\Theta)*right$
$P2 = P_0 + d*towards + d*tan(\Theta)*right$

$P = P1 + ((i + 0.5) / width) * (P2 - P1)$
$V = (P - P_0) / \|P - P_0\|$



Raio: $P = P_0 + tV$

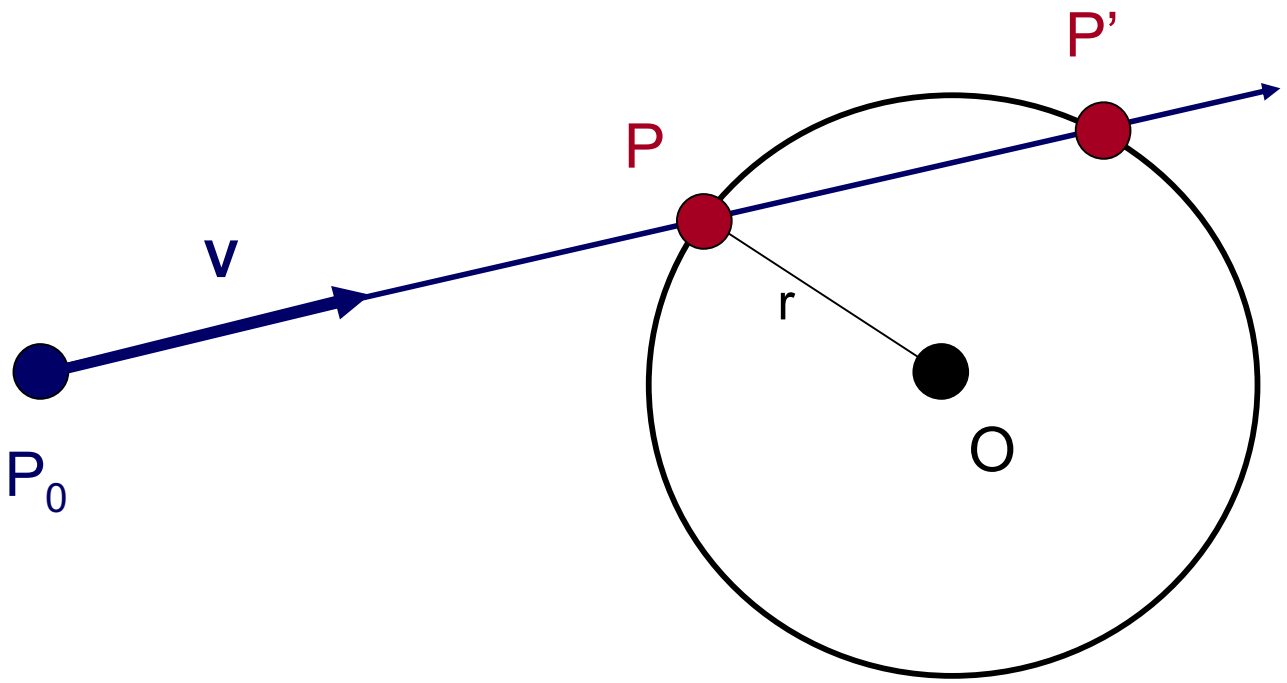
Intersecção Raio-Cena

- Intersecções com primitivas geométricas
  - Esfera
  - Triângulo
  - Grupos de primitivas (cena)
- Técnicas de aceleração
  - Hierarquias de volumes limítrofes
  - Partições espaciais
    - » Grids Uniformes
    - » Octrees
    - » BSP trees

Intersecção Raio-Esfera

Raio: $P = P_0 + tV$

Esfera: $|P - O|^2 - r^2 = 0$



Intersecção Raoi-Esfera I

Raio: $P = P_0 + tV$

Esfera: $|P - O|^2 - r^2 = 0$

Método algébrico

Substituindo para P, obtém-se:

$$|\mathbf{P_0 + tV} - O|^2 - r^2 = 0$$

Resolver a equação quadrática:

$$at^2 + bt + c = 0$$

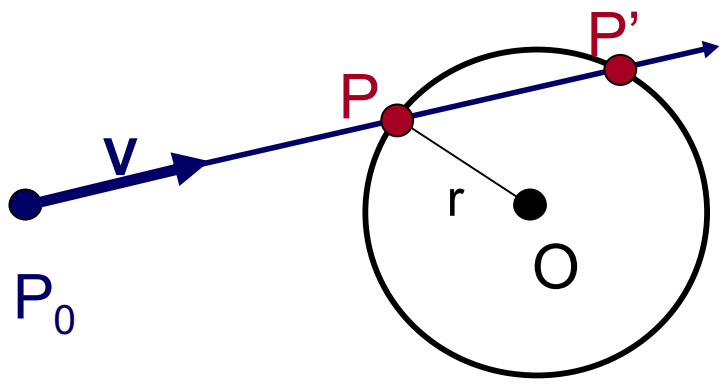
em que:

$a = 1$

$b = 2\,V \bullet (P_0 - O)$

$c = |P_0 - O|^2 - r^2 = 0$

$P = P_0 + tV$



Intersecção Raio-Esfera II

Raio: $P = P_0 + tV$

Esfera: $|P - O|^2 - r^2 = 0$

Método Geométrico

$L = O - P_0$

$t_{ca} = L \bullet V$

if ($t_{ca} < 0$) return 0

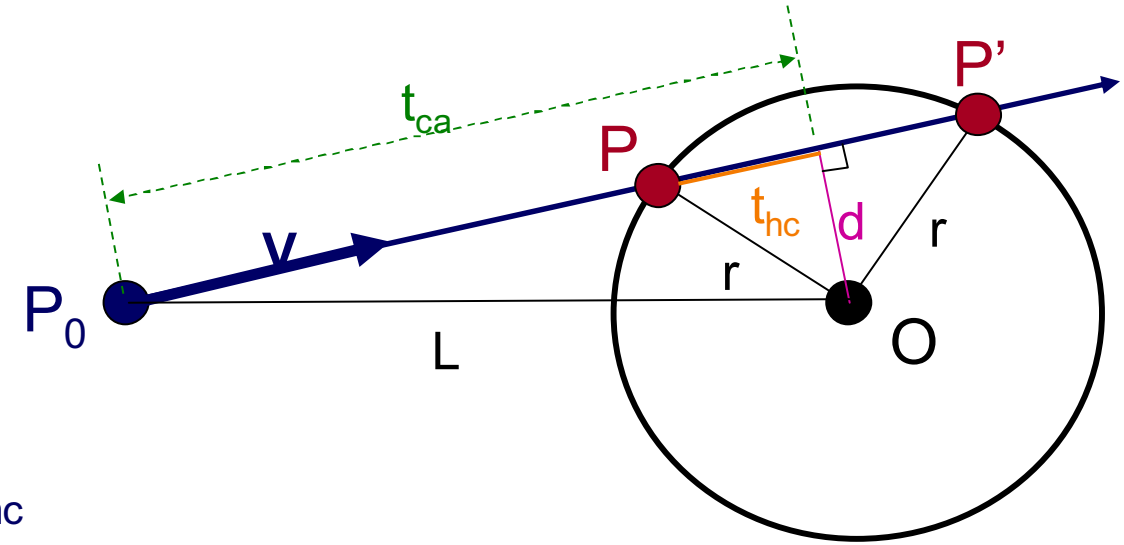
$d^2 = L \bullet L - t_{ca}^2$

if ($d^2 > r^2$) return 0

$t_{hc} = \text{sqrt}(r^2 - d^2)$

$t = t_{ca} - t_{hc}$ and $t_{ca} + t_{hc}$

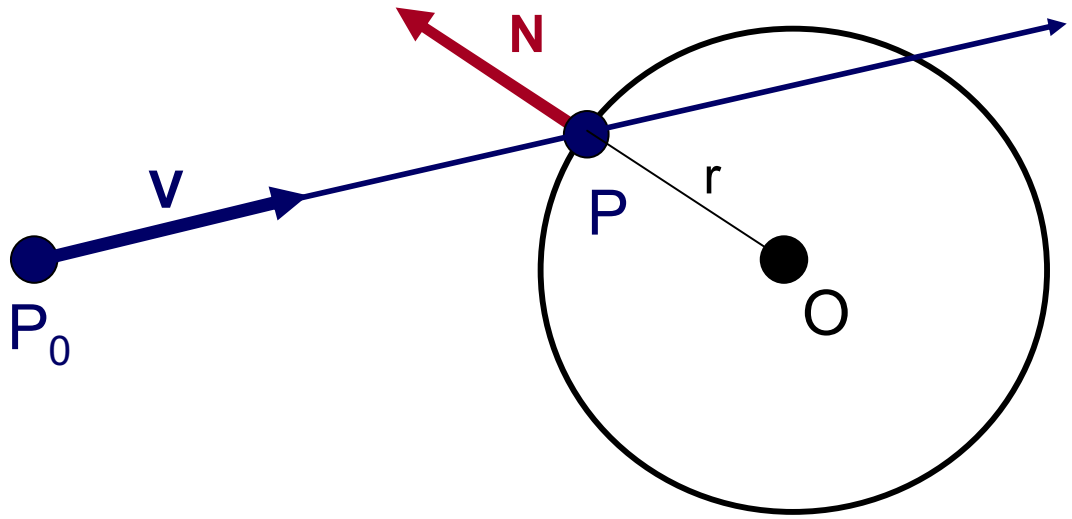
$P = P_0 + tV$



Intersecção Raio-Esfera

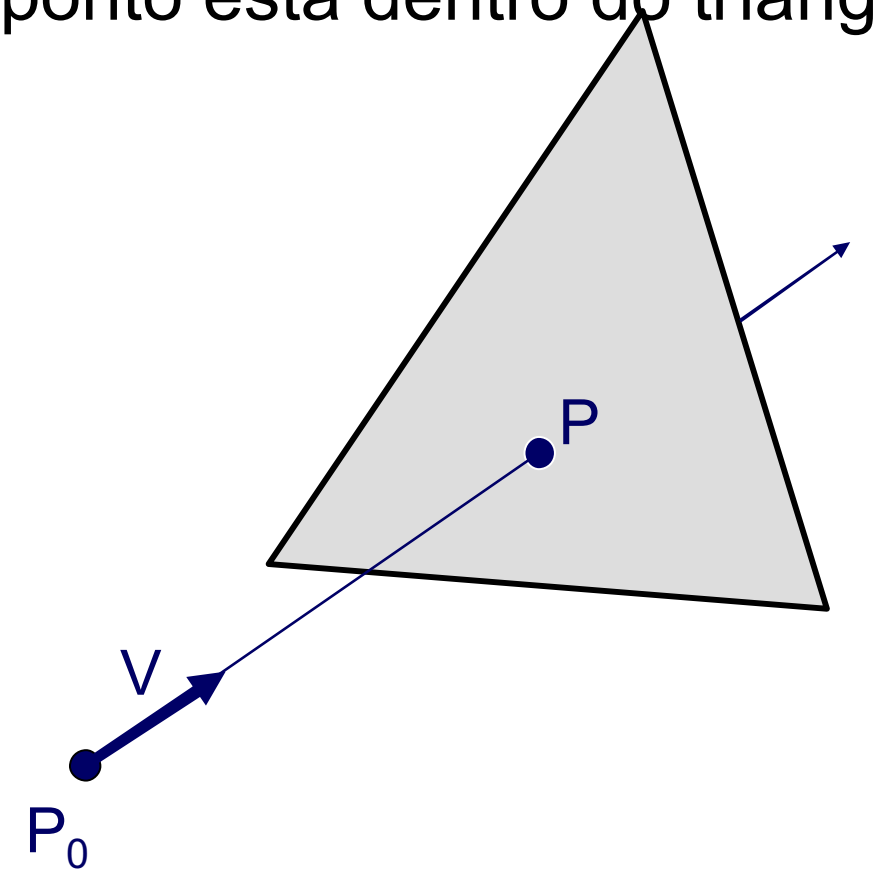
- Necessita-se da normal no ponto de intersecção para cálculos da iluminação

$$N = (P - O) / ||P - O||$$



Intersecção Raio-Triângulo

- Primeiro, calcular intersecção do raio com o plano
- Depois, verificar se o ponto está dentro do triângulo



Intersecção Raio-Plano

Raio: $P = P_0 + tV$
Plano: $P \bullet N + d = 0$

Método Algébrico

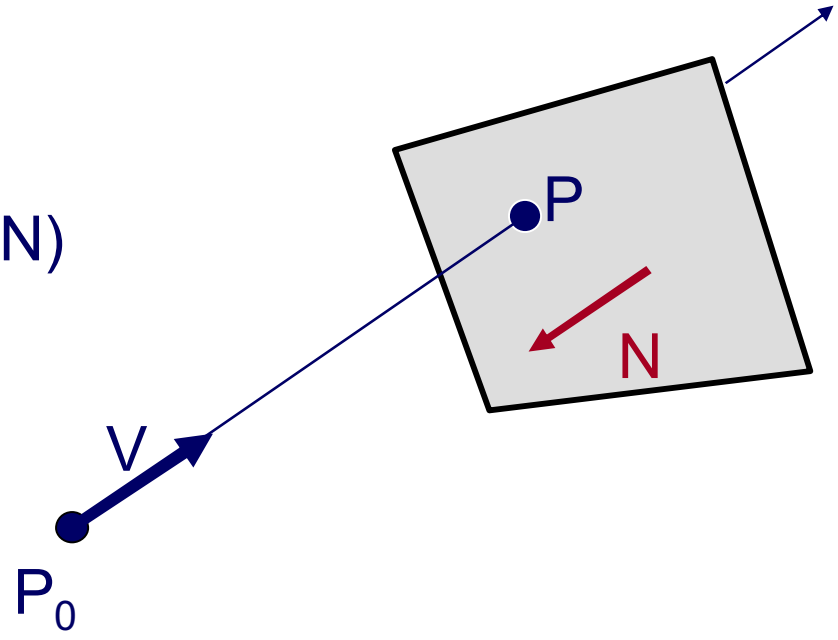
Substituindo para P, obtém-se:

$$(\mathbf{P_0 + tV}) \bullet N + d = 0$$

Solução:

$$t = -(P_0 \bullet N + d) / (V \bullet N)$$

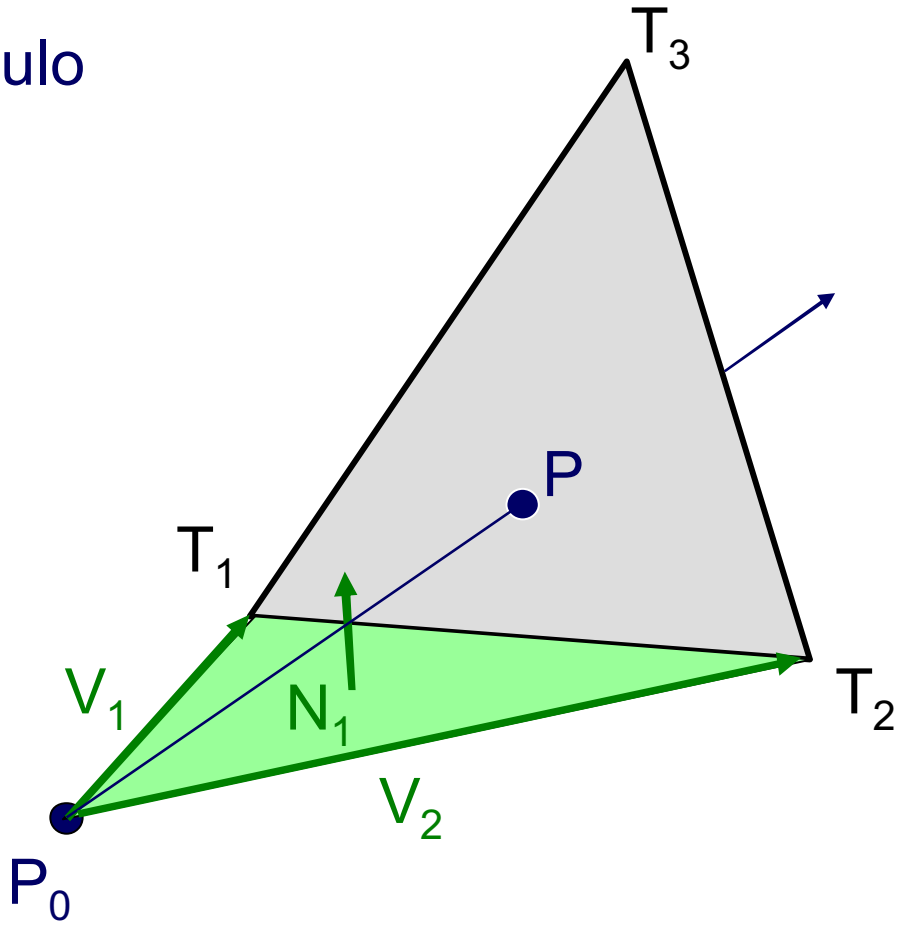
$$P = P_0 + tV$$



Intersecção Raio-Triângulo I

- Verificar se o ponto está dentro do triângulo

Para cada lado do triângulo
  $V_1 = T_1 - P$
  $V_2 = T_2 - P$
  $N_1 = V_2 \times V_1$
  Normalizar N_1
  se $((P - P_0) \bullet N_1 < 0)$
    retorna FALSO;
fim-para



Intersecção Raio-Triângulo II

- Verificar se o ponto está dentro do triângulo

Computar "coordenadas do baricentro" α, β:

α = Area(T_1T_2P) / Area($T_1T_2T_3$)

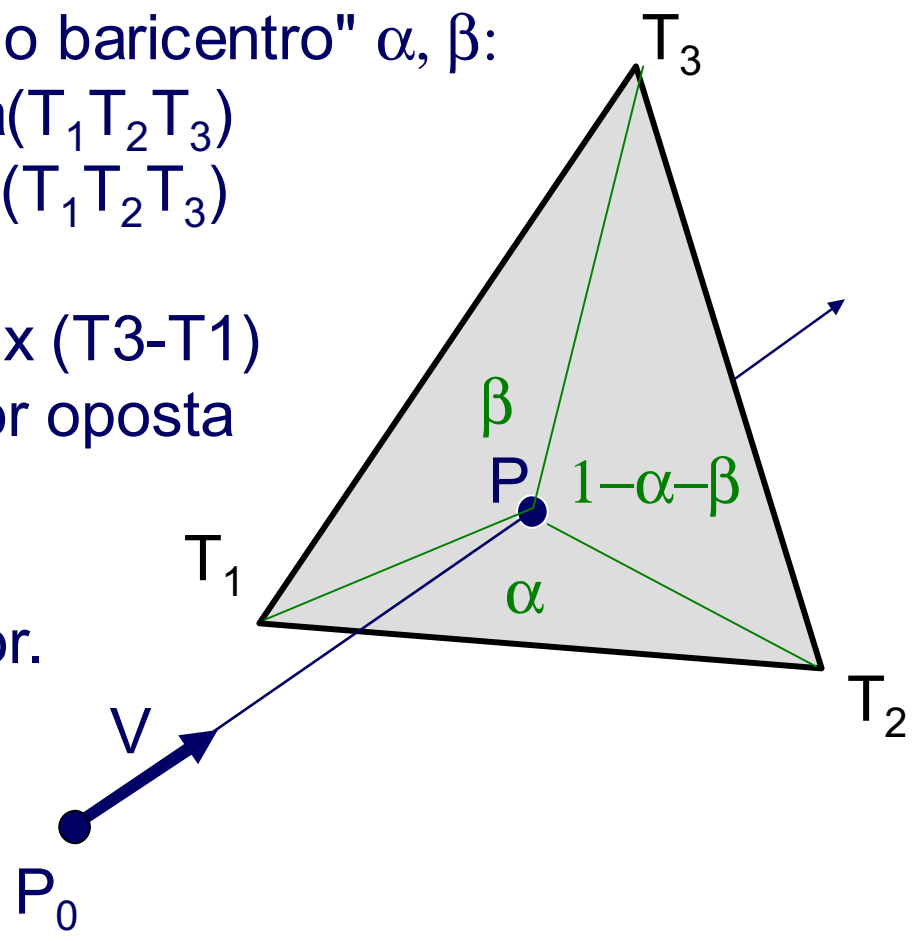
β = Area(T_1PT_3) / Area($T_1T_2T_3$)

Area($T_1T_2T_3$) = ½ (T2-T1) x (T3-T1)
(negativa se direção for oposta a da normal)

Verificar se ponto é interior.

$0 \leq \alpha \leq 1$ e $0 \leq \beta \leq 1$

$\alpha + \beta \leq 1$



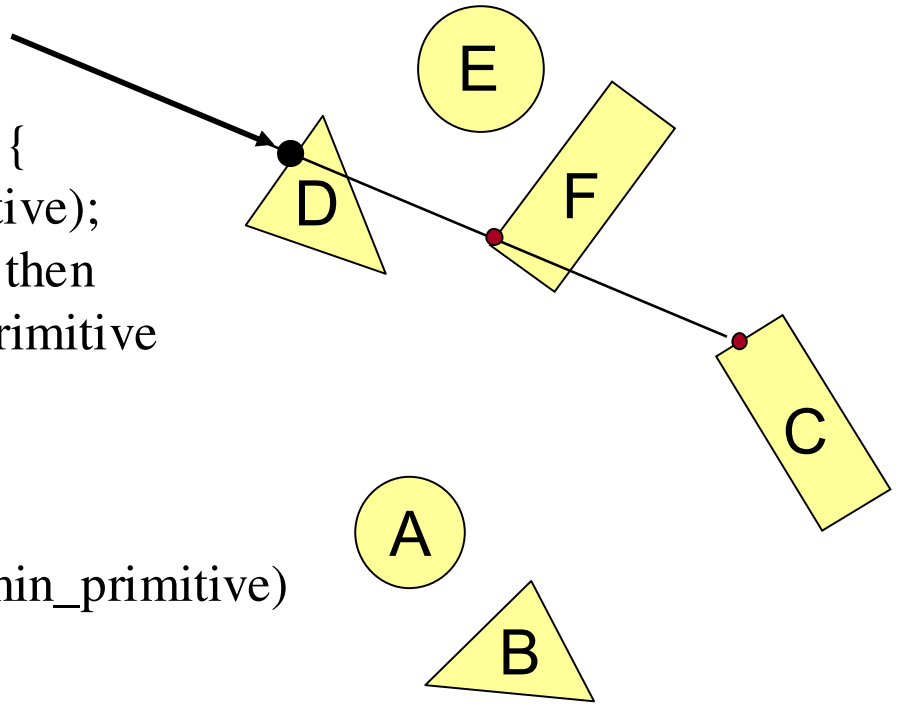
Intersecções Raio-Outras Primitivas

- Cone, cilindro, elipsóide:
  - Similar a esfera
- Box (cubo / paralelogramo)
  - Intersectar 3 faces frontais, retornar a mais próxima
- Polígono Convexo
  - Similar ao triângulo (verificar ponto interior algebricamente)
- Polígono Não Convexo
  - Similar ao polígono convexo
  - Teste mais complexo para verificar ponto interior

Intersecção Raio-Cena

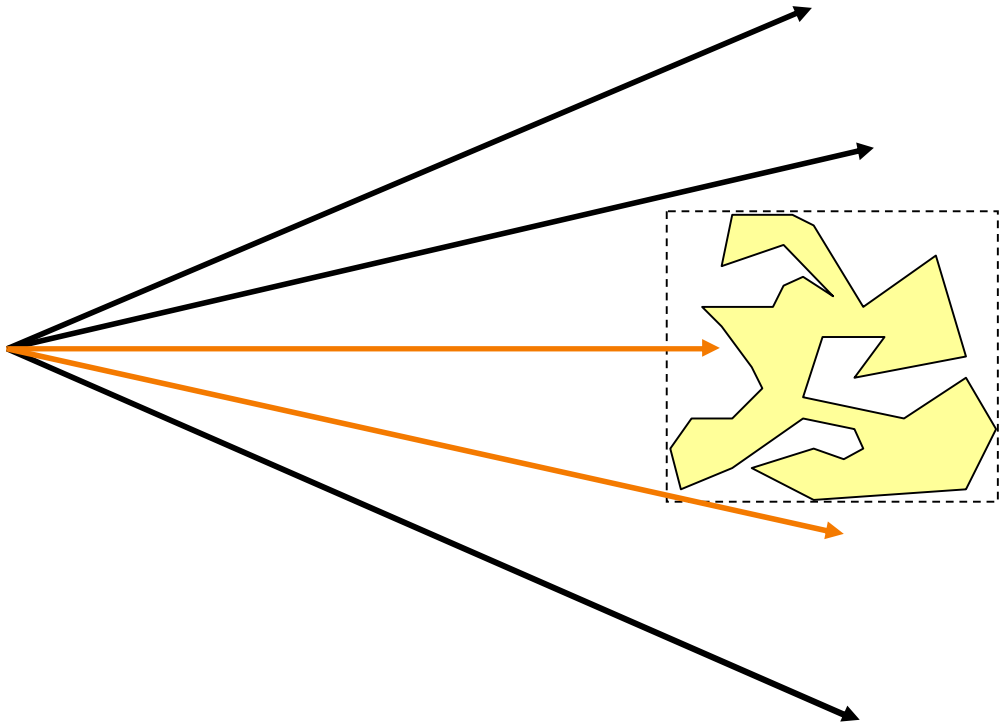
- Achar intersecção com a primitiva mais próxima

```
Intersection FindIntersection(Ray ray, Scene scene)
{
    min_t = infinity
    min_primitive = NULL
    For each primitive in scene {
        t = Intersect(ray, primitive);
        if (t > 0 && t < min_t) then
            min_primitive = primitive
            min_t = t
        }
    }
    return Intersection(min_t, min_primitive)
}
```



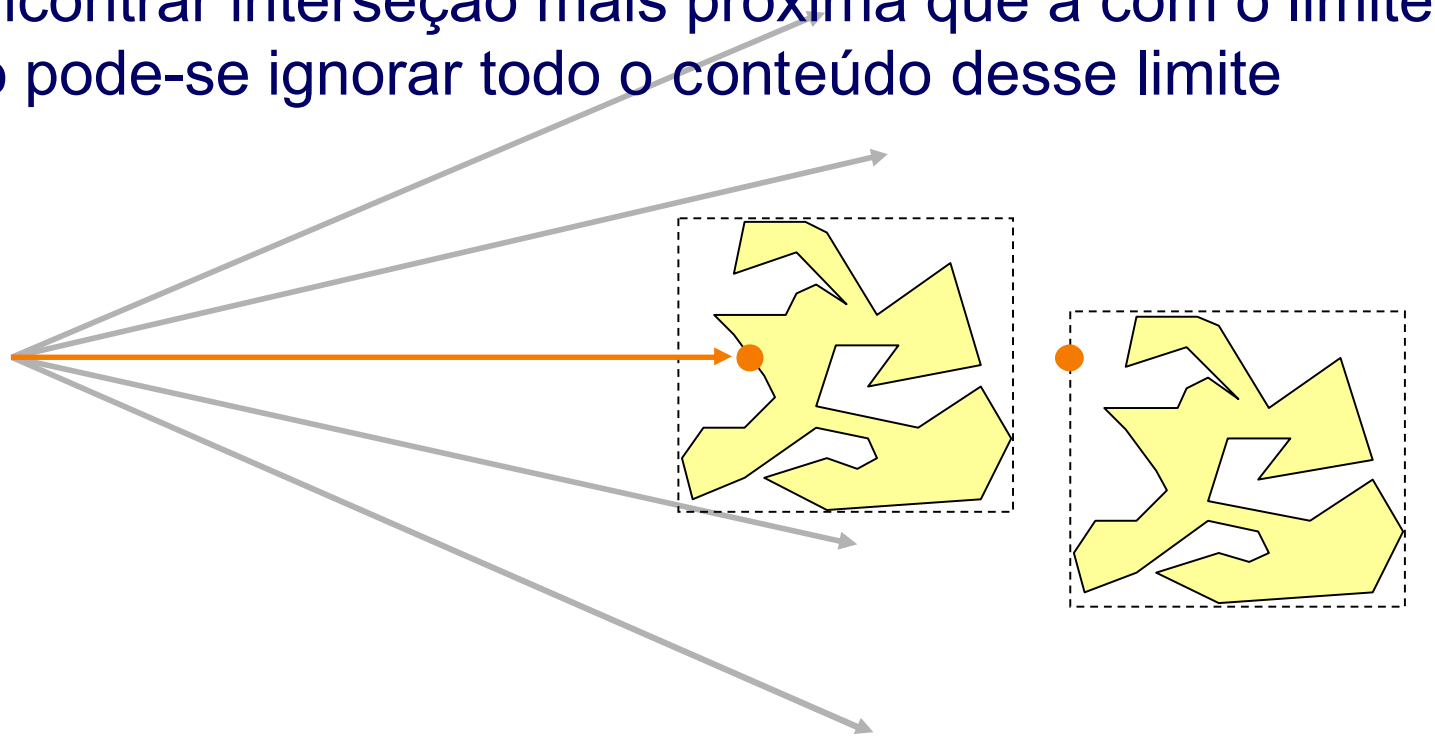
Volumes Limítrofes

- Verificar primeiro intersecção com forma mais simples
  - Se o raio não intersecta o volume limítrofe, então ele não intersecta o seu conteúdo



Volumes Limítrofes

- Verificar primeiro intersecção com forma mais simples
  - Se o raio não intersecta o volume limítrofe, então ele não intersecta o seu conteúdo
  - Se encontrar interseção mais próxima que a com o limite, então pode-se ignorar todo o conteúdo desse limite



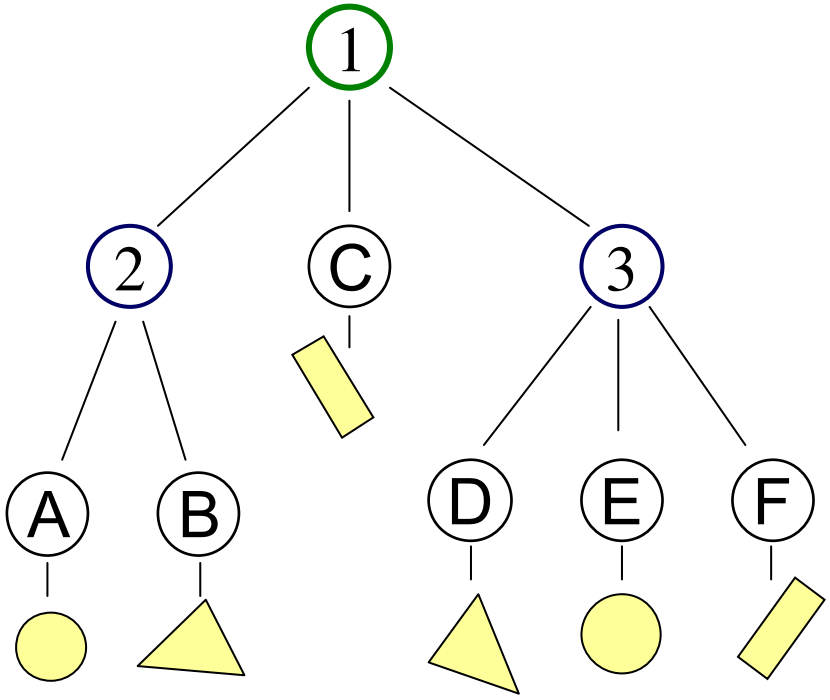
Volumes Limítrofes

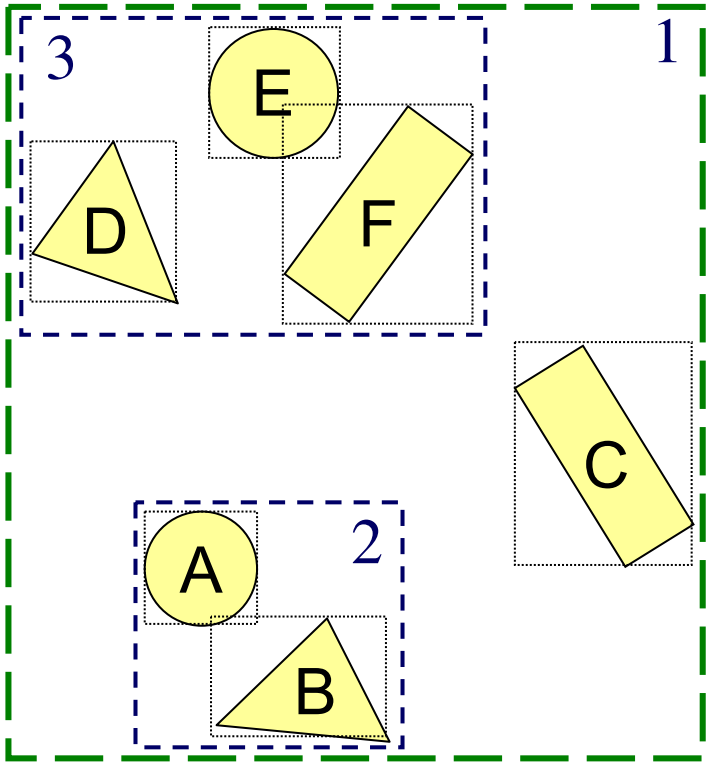
- Ordenar "hits" & detectar terminação prematura

```
FindIntersection(Ray ray, Scene scene)
{
    // Find intersections with bounding volumes
    ...
    // Sort intersections front to back
    ...
    // Process intersections (checking for early termination)
    min_t = infinity;
    for each intersected bounding volume i {
        if (min_t < bv_t[i]) break;
        shape_t = FindIntersection(ray, bounding volume contents);
        if (shape_t < min_t) { min_t = shape_t;}
    }
    return min_t;
}
```

Hierarquias de Volumes Limítrofes I

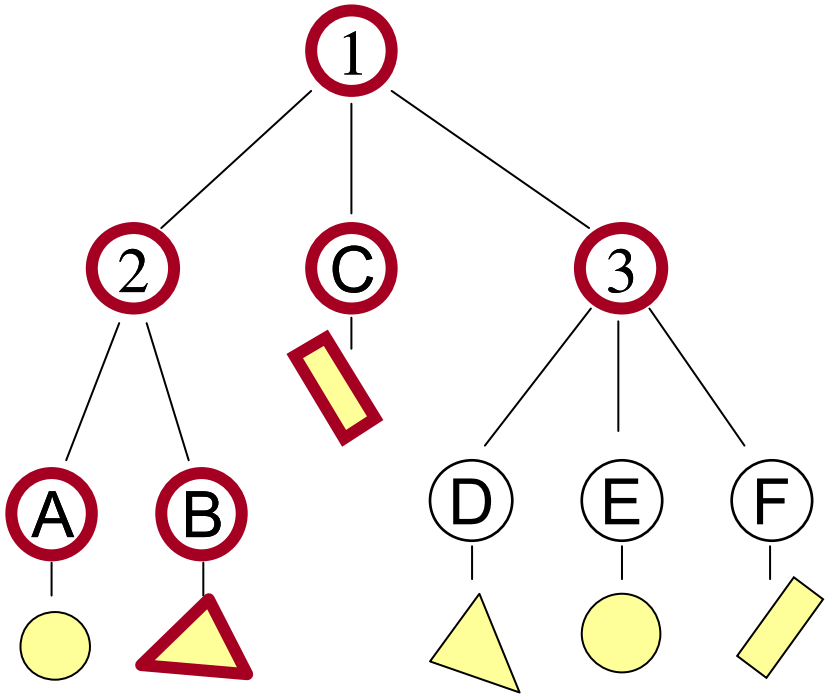
- Construir hierarquia de volumes limítrofes
  - Volume limítrofe de nó interior contém todos os fílhos

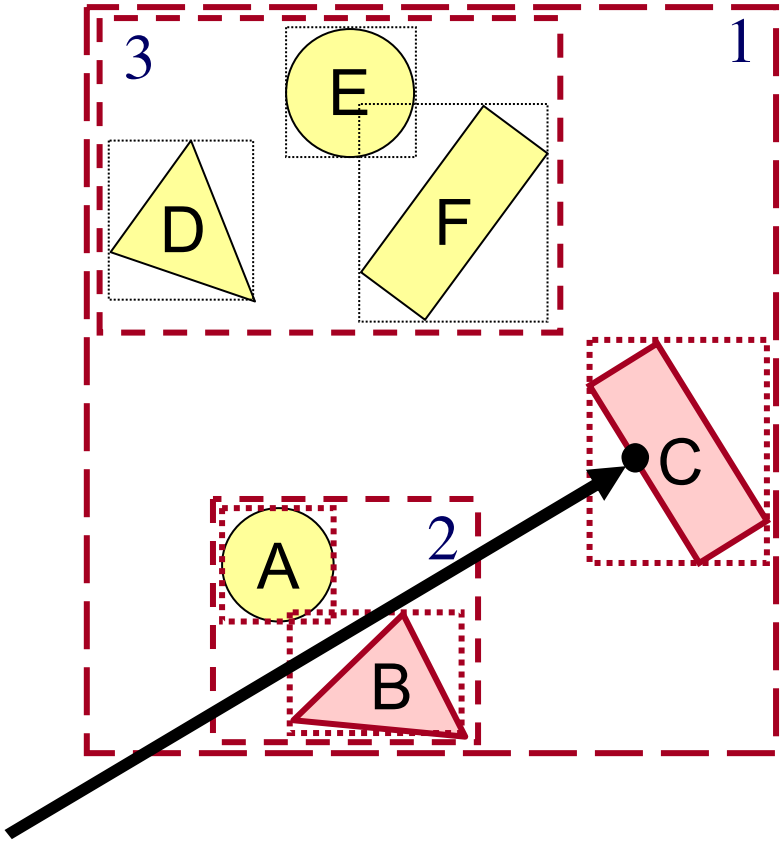




Hierarquias de Volumes Limítrofes II

- Usar hierarquia para acelerar intersecções
  - Testar conteúdo de nó apenas se intersectar volume limítrofe





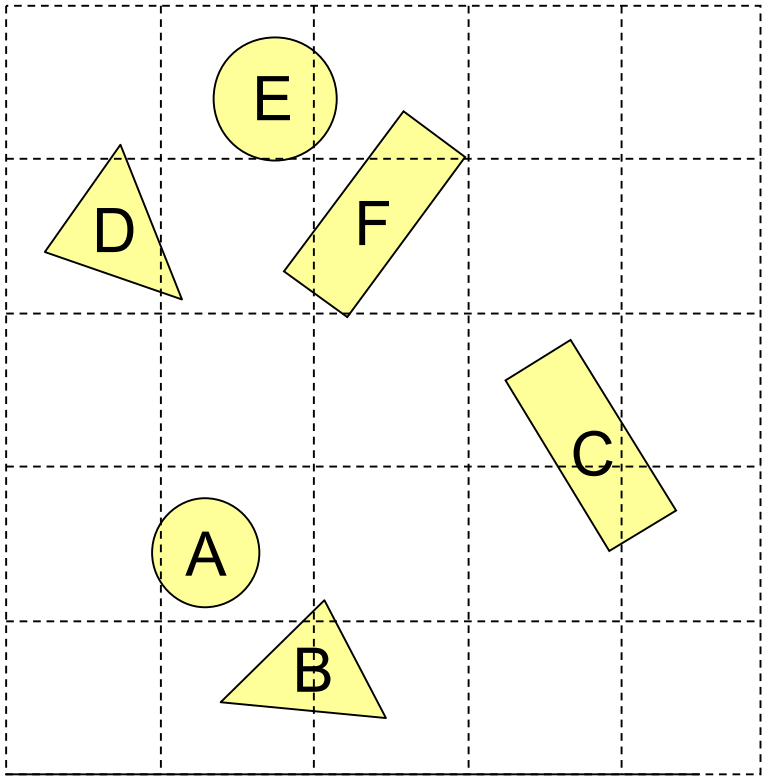
Hierarquias de Volumes Limítrofes III

- Percorrer os nós da cena recursivamente

```
FindIntersection(Ray ray, Node node)
{
      // Find intersections with child node bounding volumes
      ...
      // Sort intersections front to back
      ...
      // Process intersections (checking for early termination)
      min_t = infinity;
      for each intersected child i {
            if (min_t < bv_t[i]) break;
            shape_t = FindIntersection(ray, child);
            if (shape_t < min_t) { min_t = shape_t;}
      }
      return min_t;
}
```

Grid Uniforme

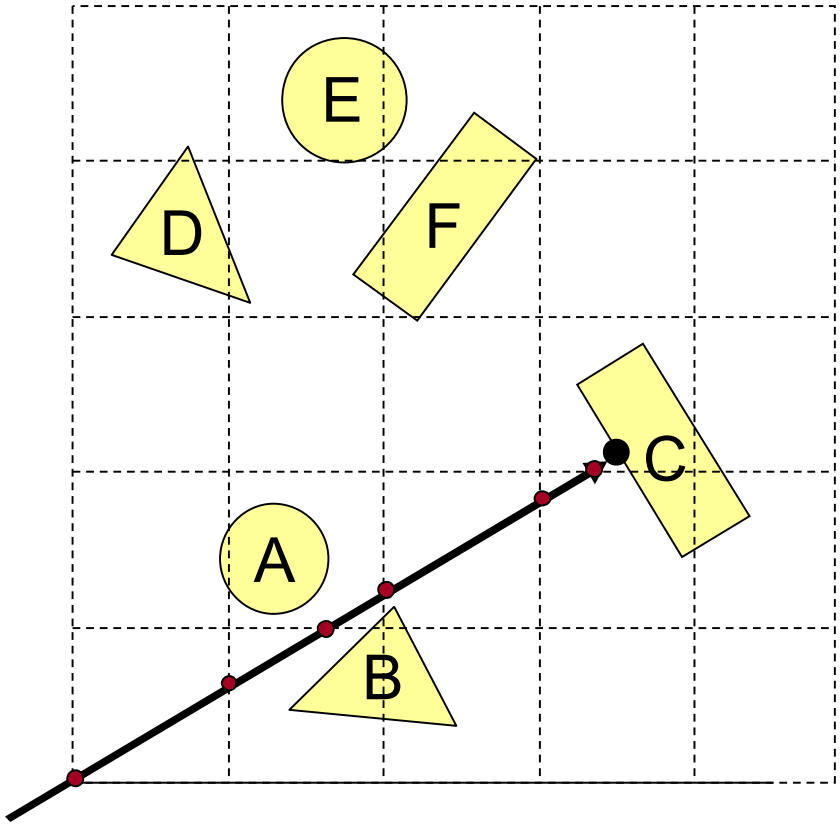
- Construir grid uniforme sobre a cena
  - Indexar primitivas de acordo com a sobreposição das células



Grid Uniforme

- Traçar raios através das células do grid
  - Rápido
  - Incremental

Apenas testa primitivas nas células intersectadas

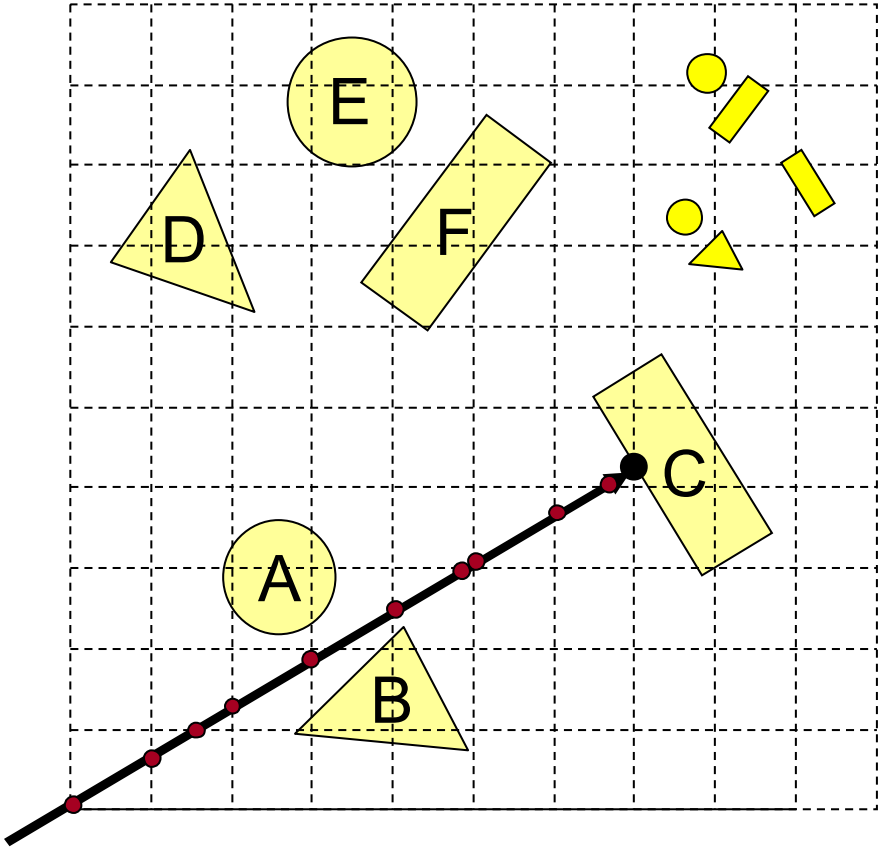


Grid Uniforme

- Problema "potencial":
  - Como escolher uma resolução adequada para o grid ?

Muito pequena
torna o grid esparso

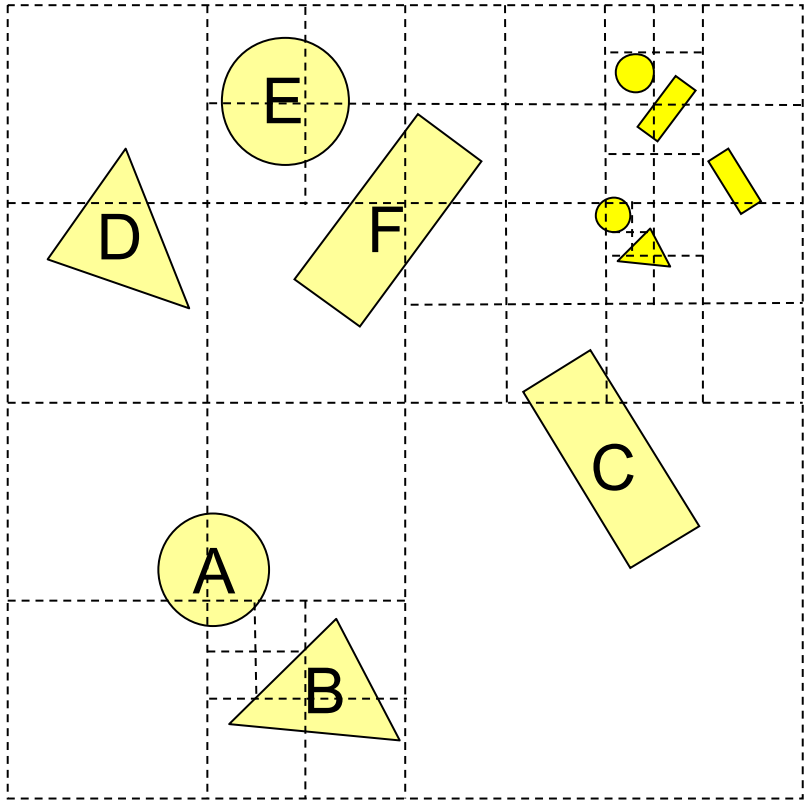
Muito cara se
o grid for denso



Octree

- Construir grid adaptivo sobre a cena
  - Subdividir recursivamente as células em 8 octantes
  - Indexar primitivas a partir da sobreposição com as células

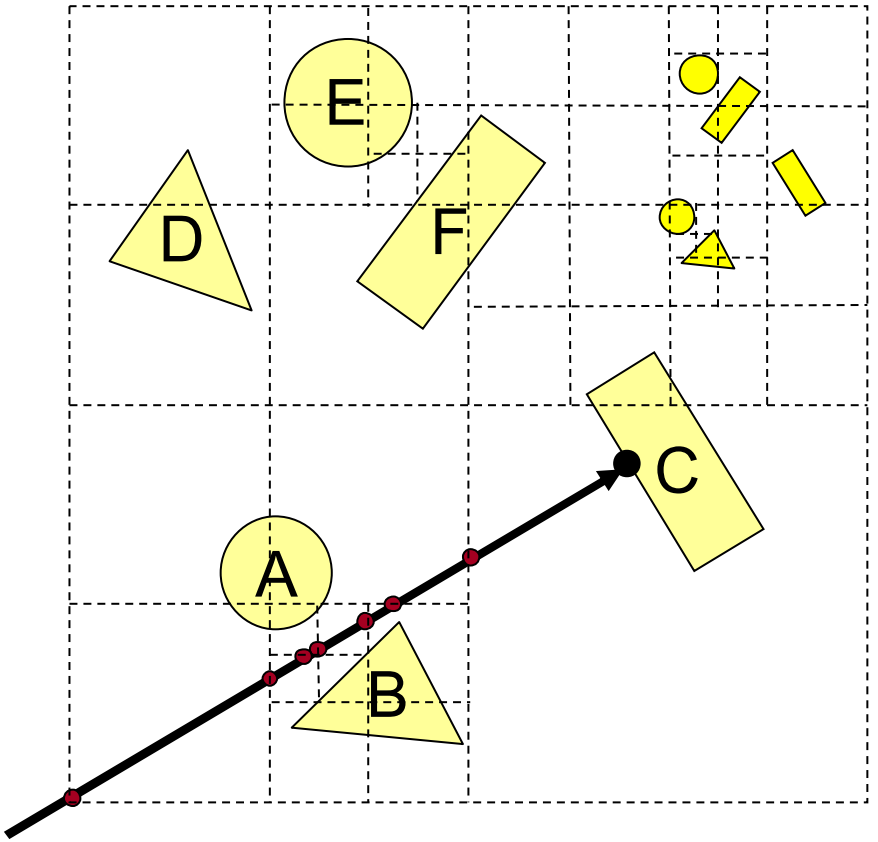
Gera menos células



Octree

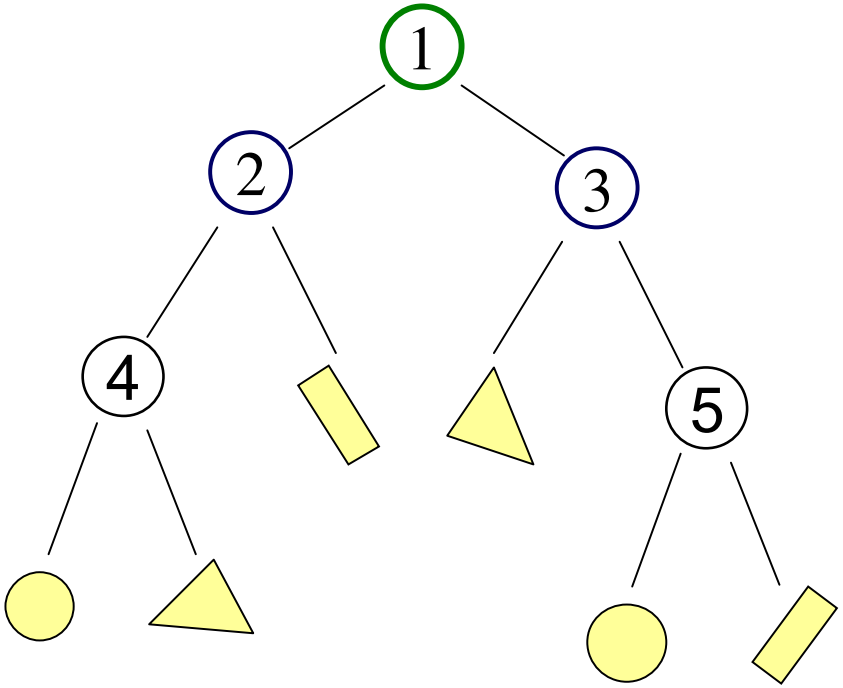
- Traçar raios através das células vizinhas
  - Poucas células
  - Maior complexidade na determinação de vizinhos

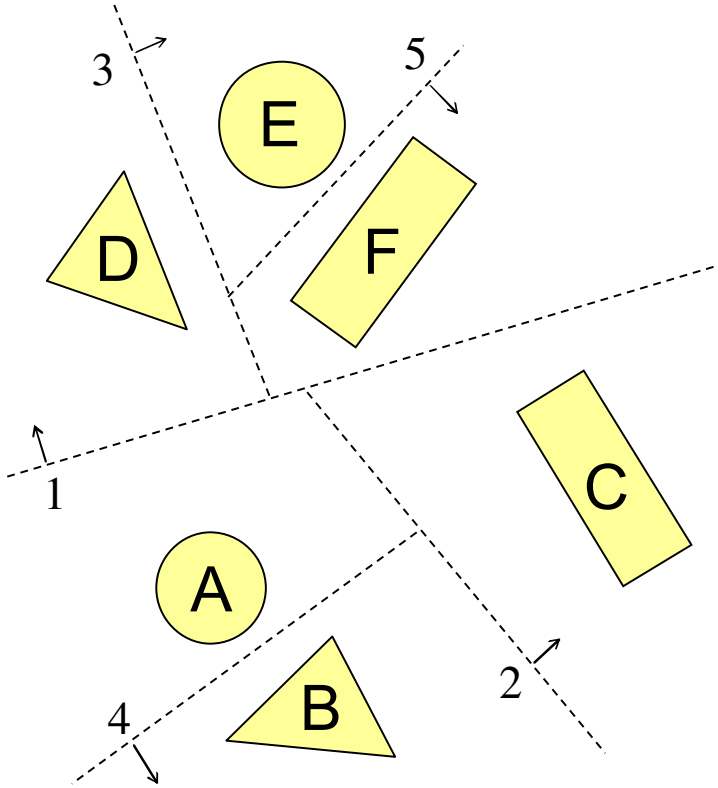
Troca menos células por mais caro caminhamento



Binary Space Partition (BSP) Tree

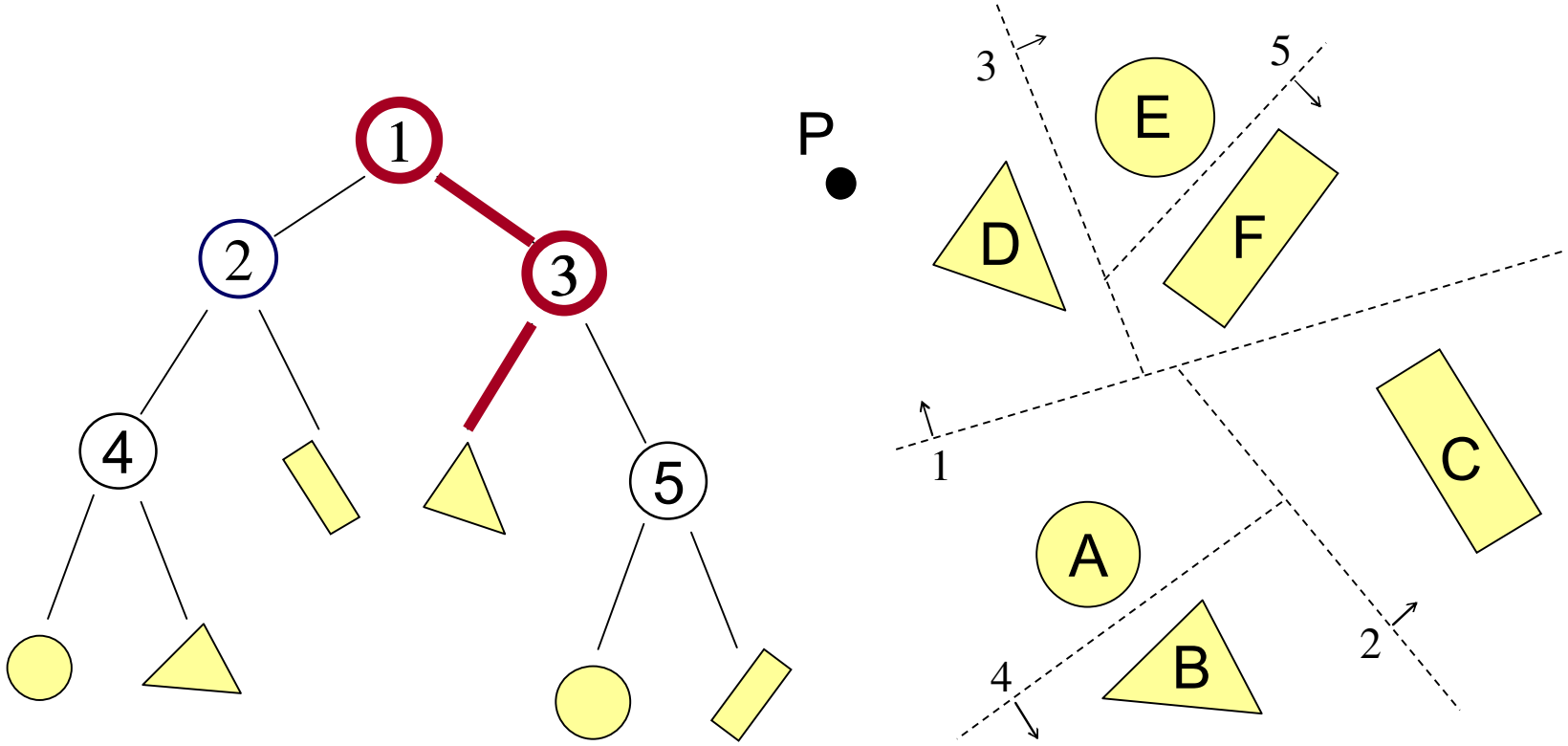
- Partição recursiva do espaço por planos
  - Cada célula é um poliedro convexo





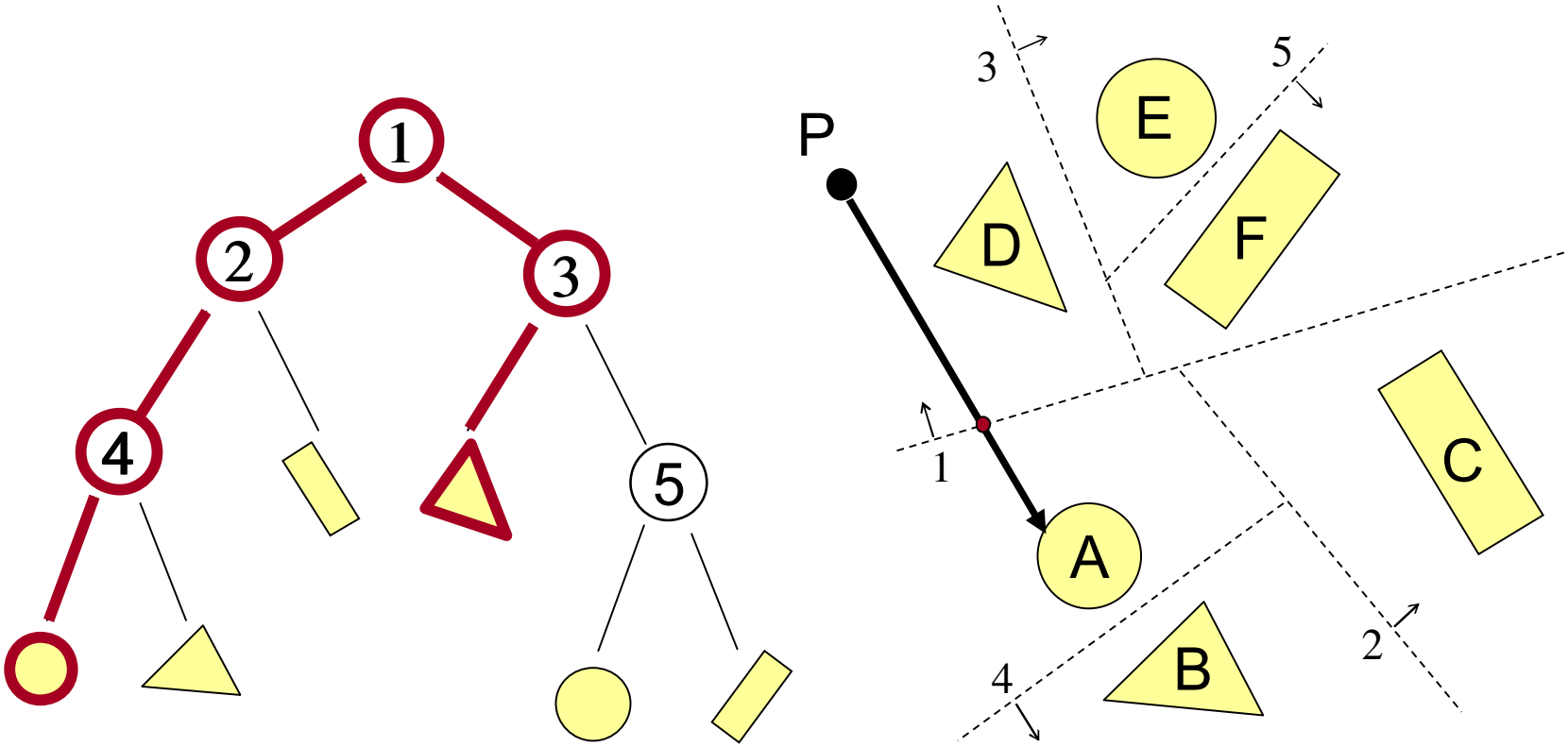
Binary Space Partition (BSP) Tree

- Algoritmos recursivos simples
  - Exemplo: localização de ponto



Binary Space Partition (BSP) Tree

- Traçar raios através da recursão sobre a árvore
  - BSP tree permite caminhamento simples (próximo-distante)



Binary Space Partition (BSP) Tree

```
RayTreeIntersect(Ray ray, Node node, double min, double max)
{
    if (Node is a leaf)
        return intersection of closest primitive in cell, or NULL if none
    else
        dist = distance of the ray point to split plane of node
        near_child = child of node that contains the origin of Ray
        far_child = other child of node
        if the interval to look is on near side
            return RayTreeIntersect(ray, near_child, min, max)
        else if the interval to look is on far side
            return RayTreeIntersect(ray, far_child, min, max)
        else if the interval to look is on both side
            if (RayTreeIntersect(ray, near_child, min, dist)) return …;
            else return RayTreeIntersect(ray, far_child, dist, max)
}
```

Aceleração

- Técnicas para acelerar interseção são importantes
  - Hierarquias de volumes limítrofes
  - Partições espaciais
- Conceitos gerais
  - Ordenar objetos espacialmente
  - Fazer rejeições triviais o mais rápido possível
  - Utilizar informações de coerência se possível

Tempo esperado é sublinear no núm. de primitivas