Raytracing

SCC0250 - Computação Gráfica

Profa. Maria Cristina F. Oliveira

Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)

5 de dezembro de 2023

Sumário

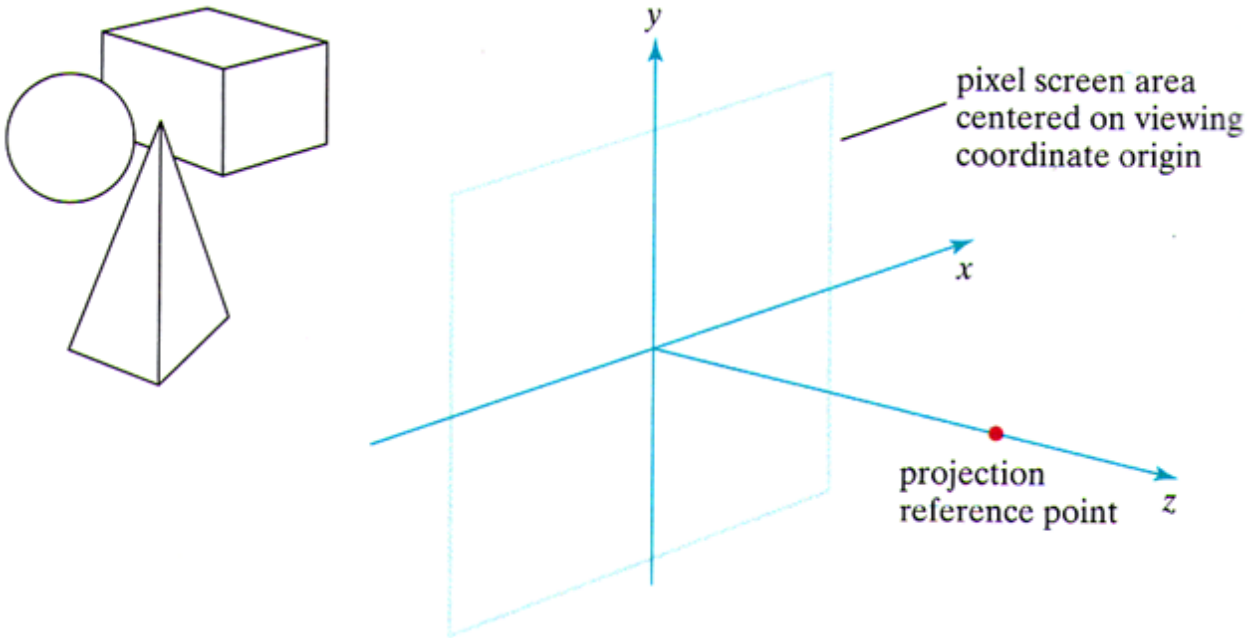
Sumário

Ray Casting e Ray Tracing

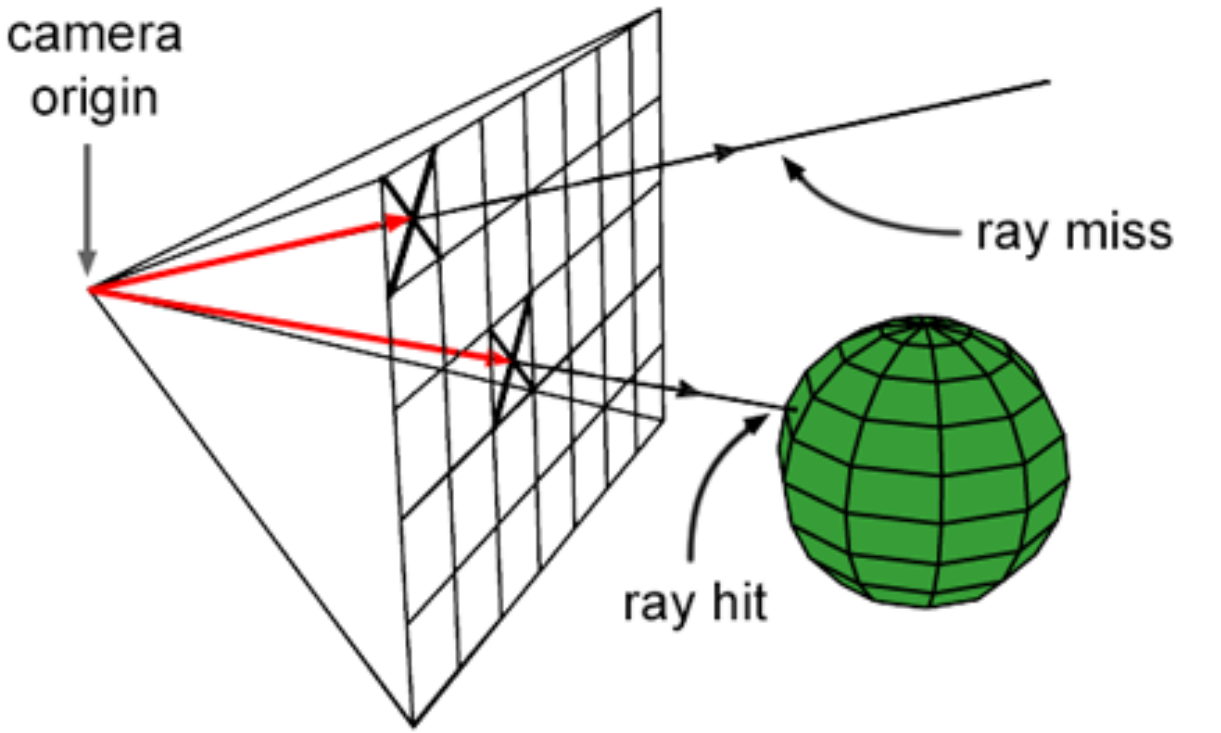
- O algoritmo de **Ray Tracing** é uma generalização do conceito mais simples de **Ray Casting**, ou 'disparo de raios'.
- Consideremos a cena já mapeada para o VCS (*Viewing Coordinate System*), com as fontes de luz e observador posicionados.

- **Ray Casting**: Lança raios a partir da posição do observador, passando por cada pixel da *viewport*. Os raios podem interceptar algum objeto da cena: o algoritmo calcula o modelo de iluminação no ponto de intersecção para determinar a cor do pixel.
- Algoritmo baseado em imagem (*image-based*)

Ray Casting - configuração



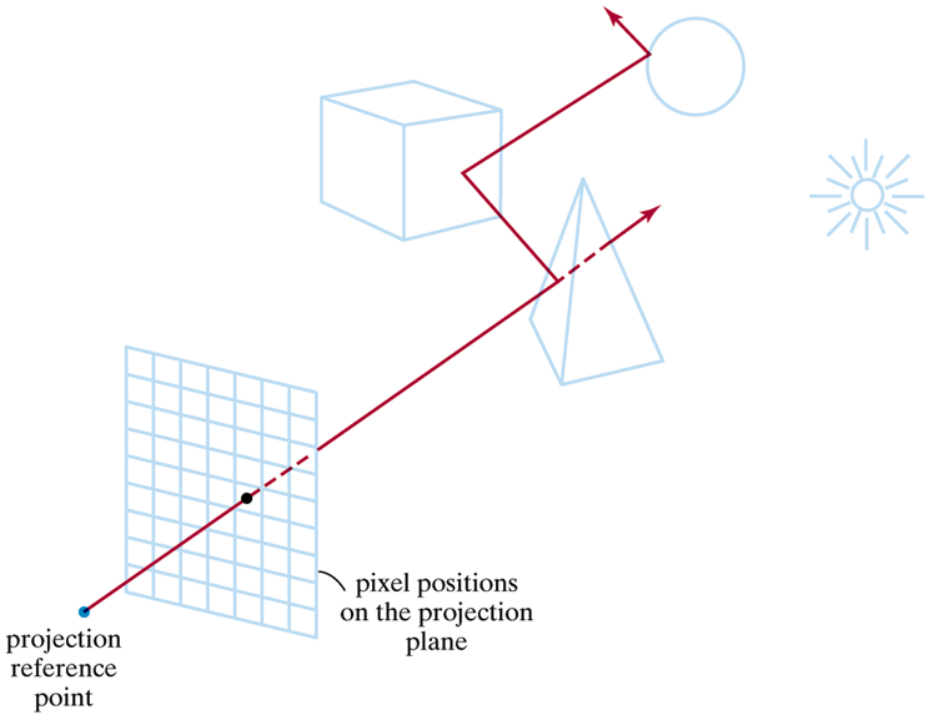
Ray Casting



Ray Tracing

- **Ray Tracing**: Lança raios a partir da posição do observador, passando por cada pixel: os raios podem interceptar algum objeto da cena, a partir desse ponto novos raios são disparados na cena
- Raio na direção de reflexão: considera a luz que incide no objeto e é refletida de volta para a cena
- Se o objeto é de material transparente, também gera um raio na direção de refração: considera que parte da luz incidente é transmitida de volta para a cena
- A cor no pixel será uma combinação dessas contribuições: luz incidente no ponto, luz refletida no ponto, luz refratada no ponto (se for o caso)

Ray Tracing


pixel positions
on the projection
plane
projection
reference
point

Ray Tracing

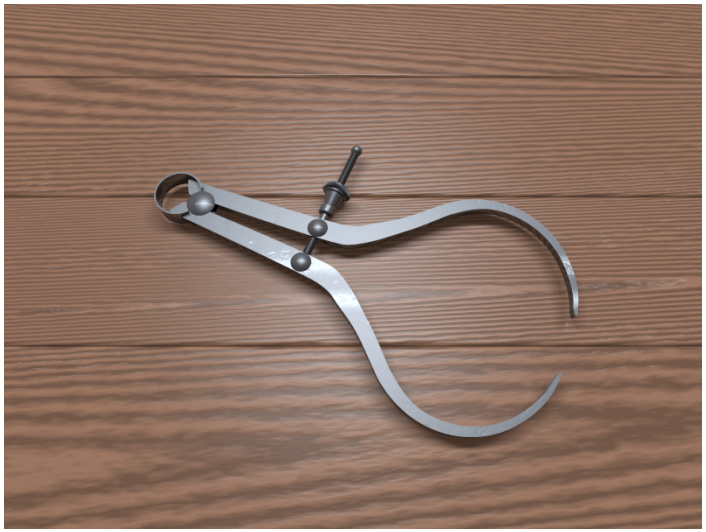
- Motivação: a iluminação em um ponto de uma superfície é resultado da contribuição da luz incidente vinda diretamente da fonte de luz (computada aplicando o modelo de iluminação), mais a contribuição da luz incidente refletida/transmitida por outros objetos da cena. O algoritmo incorpora:
  - Efeitos de **reflexão e transmissão** da luz pelos múltiplos objetos da cena (modela a distribuição da luz ambiente)
  - A identificação de **áreas de sombra**
  - Os efeitos de **transparência**
  - Os efeitos da **projeção perspectiva**
  - Os efeitos de iluminação devido à presença de **múltiplas fontes de luz**

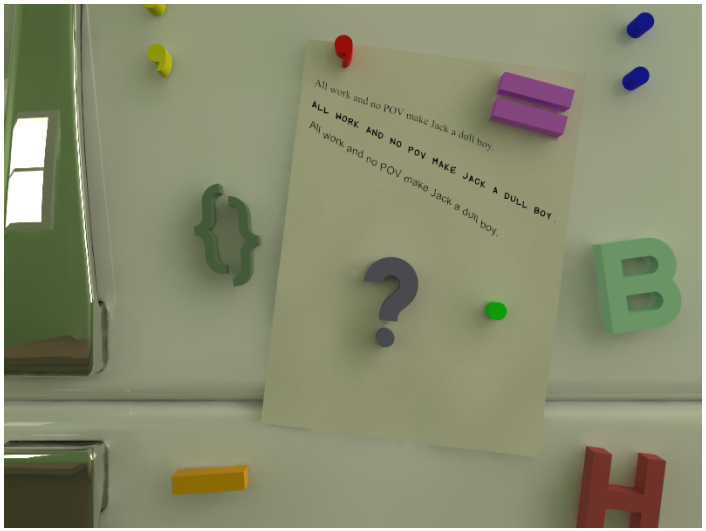
Ray Tracing

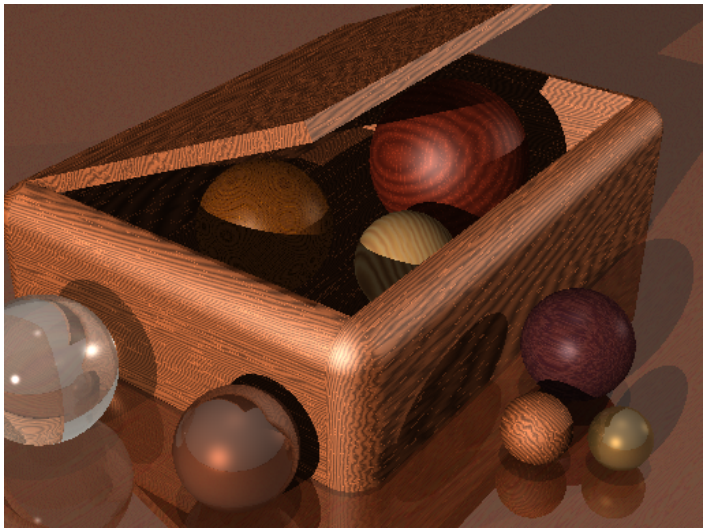
- Pode gerar **imagens realísticas**, particularmente em cenas com objetos de materiais especulares
- Porém, é um método **computacionalmente custoso**

Veja `https://hoxxep.github.io/webgl-ray-tracing-demo/`

Ray Tracing

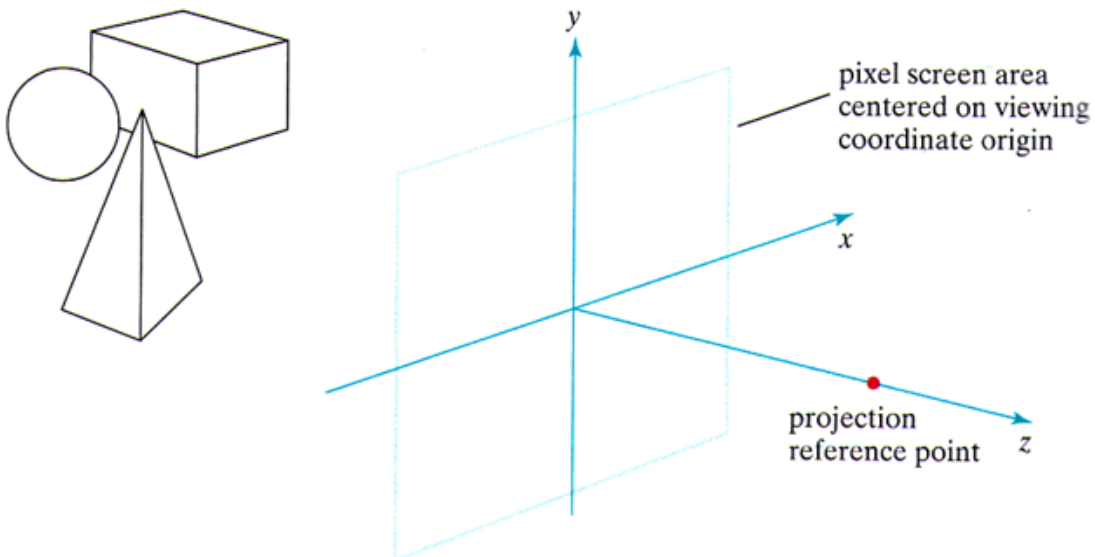



All work and no POV make Jack a dull boy.
ALL WORK AND NO POV MAKE JACK A DULL BOY
All work and no POV make Jack a dull boy.



Algoritmo Básico de Ray Tracing

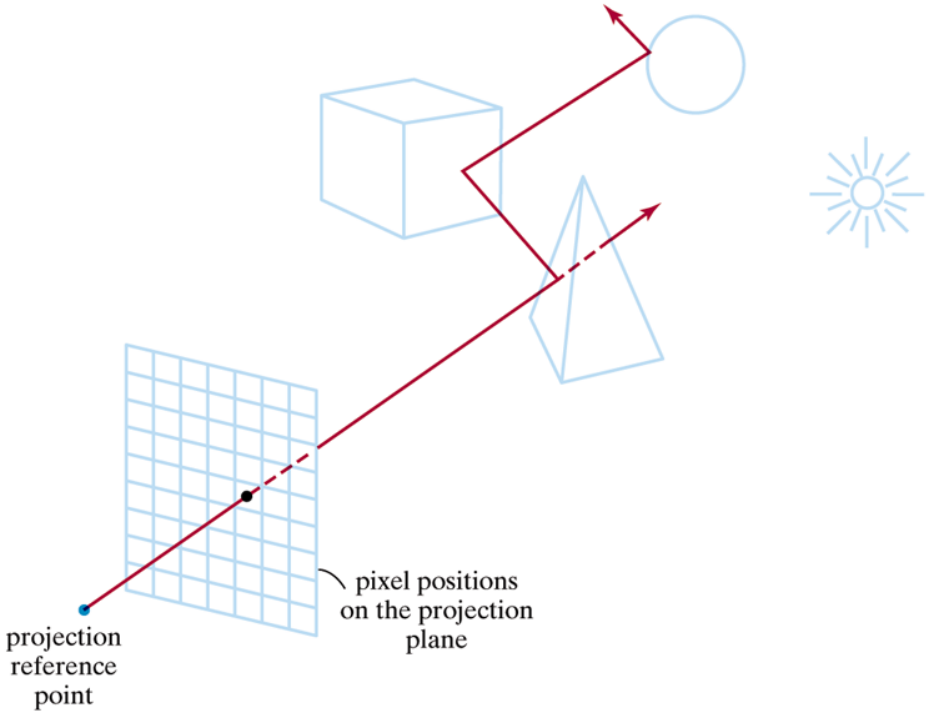
- O sistema de coordenadas para o algoritmo de *ray tracing* normalmente coloca o observador (PRP) ao longo do eixo z e a tela no plano xy
- Dada a cena descrita nesse sistema de coordenadas (VCS), os raios são gerados a partir do observador, passando pelos pixels da tela



Algoritmo Básico de Ray Tracing

- Cada raio tem origem no PRP, passa pelo centro de um pixel e segue seu caminho na cena, sendo refletido e transmitido a cada intersecção encontrada
- A cor (valores R,G,B) de um pixel é determinada pelas contribuições da iluminação acumuladas no primeiro ponto de interseção do raio com um objeto da cena
- O cálculo da cor deve incluir a luz da fonte que incide nesse ponto, mais a luz incidente no ponto devido à luz refletida e/ou transmitida por outros objetos da cena
- O traçado dos raios é recursivo: a cada intersecção, novos raios são gerados, que também precisam ser traçados

Algoritmo Básico de Ray Tracing



Algoritmo Básico de Ray Tracing

- Essa abordagem de *rendering* é baseada nos princípios da geometria ótica
- Infinitos 'raios´ de luz emanam da fonte e das superfícies em várias direções, e uma parcela deles vai atingir o plano de projeção (pixels), contribuindo para a cena que o observador vê
- Seria impossível rastrear o caminho de todos eles!

Método de Ray Tracing

- O algoritmo na verdade computa o processo reverso (*backward ray tracing*)
- Partindo dos pixels, quais raios de luz da cena chegam a esse pixel e contribuem para a sua cor?
- Feito esse cálculo para todos os pixels, tem-se a cena vista pelo observador!

Algoritmo Básico de Ray Tracing

Cálculo das intersecções

- Para cada raio gerado é preciso processar todas as superfícies da cena para encontrar os pontos de intersecção
- Um raio pode interceptar muitas superfícies!
- A cada ponto de intersecção encontrado, o algoritmo calcula a distância do ponto de intersecção ao pixel (para achar o mais próximo)

Reflete e Transmite o Raio

- O ponto de interseção mais próximo do observador (menor distância) identifica qual superfície é visível no pixel
  - O raio é então refletido (reflexão especular) a partir do ponto de intersecção
  - Para uma superfície de material transparente, o raio também é refratado (transmitido)

Algoritmo Básico de Ray Tracing

Raios Primários e Secundários

- O raio inicial é o **raio primário**, os raios refletidos e refratados gerados a partir de um ponto de intersecção são denominados **raios secundários**

Processa os Raios Secundários

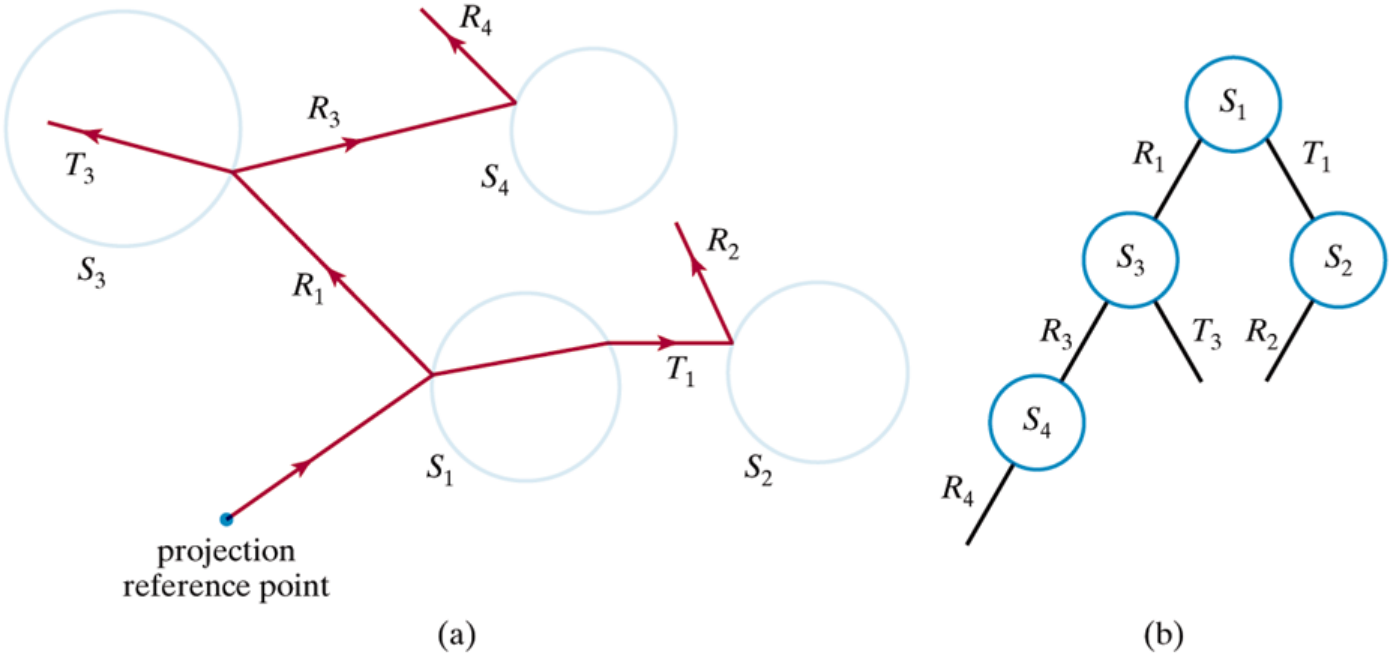
- Novamente, para cada raio busca-se pelas intersecções com as superfícies, considerando a intersecção mais próxima do observador para gerar recursivamente a próxima geração de raios de reflexão e refração

Algoritmo Básico de Ray Tracing

Estrutura de dados

- Conforme o raio ricocheteia na cena, as intersecções vão sendo adicionadas a uma **árvore binária de ray tracing**
  - Ramos a esquerda representam caminhos de reflexão e ramos a direita os caminhos de transmissão
- O usuário pode definir a profundidade da árvore, i.e., quantos níveis de raios secundários serão gerados

Algoritmo Básico de Ray Tracing



Algoritmo Básico de Ray Tracing

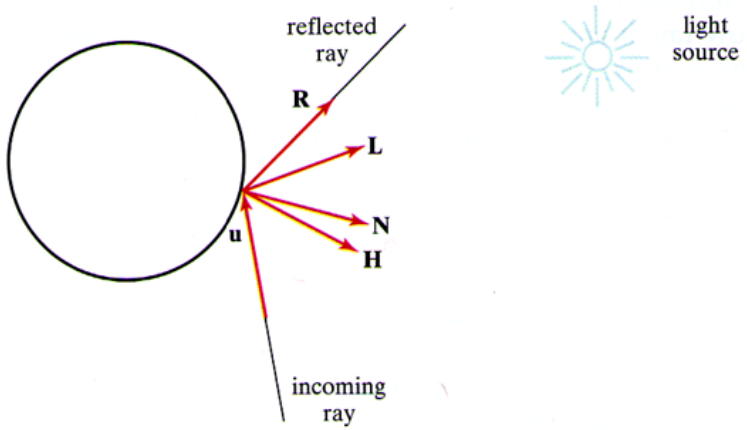
Critério de Parada

- Um **caminho na árvore** binária gerada para um pixel é **terminado** (i.e., novos raios param de ser gerados) se uma das seguintes condições for satisfeita
  - O raio **não intercepta** nenhuma superfície
  - O raio **intercepta uma fonte de luz** (que não é uma superfície refletora)
  - A **profundidade máxima** da árvore (especificada pelo usuário) foi alcançada
- A contribuição resultante de todos os ramos na árvore define a cor do pixel
- A contribuição dos raios vai sendo atenuada a cada nível: por isso é razoável especificar uma profundidade máxima

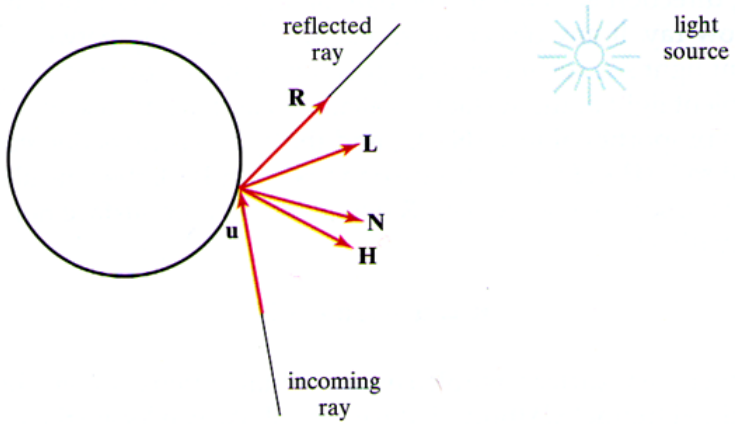
Algoritmo Básico de Ray Tracing

Tonalização

- A cada intersecção, o algoritmo aplica o modelo de iluminação para determinar a cor RGB no ponto, e o valor resultante é armazenado na árvore, na posição correspondente ao raio
  - Um raio que intercepta uma superfície não reflexiva pode não gerar novo raio refletido
  - Nesse caso, o pixel recebe apenas a cor computada pelo modelo de iluminação (contribuição da luz vinda diretamente da fonte)



Algoritmo Básico de Ray Tracing



- A figura mostra os vetores unitários no modelo de iluminação (Blinn-Phong)
- O vetor $\mathbf{u}$ dá a direção do raio incidente, $\mathbf{N}$ é o vetor normal à superfície no ponto, $\mathbf{R}$ dá a direção do raio de luz refletido, $\mathbf{L}$ dá a direção da luz incidente vinda da fonte, $\mathbf{V}$ dá a direção de observação, $\mathbf{H}$ é o vetor intermediário entre $\mathbf{L}$ e $\mathbf{V}$
- No RT o vetor direção de observação é o oposto do vetor que dá a direção do raio incidente, i.e., $\mathbf{V} = -\mathbf{u}$

Algoritmo Básico de Ray Tracing

Cálculo de Sombra

- A direção dada pelo vetor $\mathbf{L}$, que vai do ponto à fonte de luz, é conhecida como **raio de sombra** (*shadow ray*
- Um ponto da superfície está na sombra em relação a uma fonte de luz se o seu *shadow ray* interceptar algum objeto, i.e., existe um objeto entre a superfície e essa fonte
- Nesse caso, a fonte não contribui para a iluminação direta do ponto

Modelo de Iluminação

- O modelo de iluminação é o modelo básico de Blinn-Phong
  - Luz ambiente na superfície, calculada como $k_a I_a$
  - Reflexão difusa, dada por $k_d(\mathbf{N} \cdot \mathbf{L})$
  - Reflexão especular, dada por $k_s(\mathbf{H} \cdot \mathbf{N})^{n_s}$

Algoritmo Básico de Ray Tracing

Reflexão

- A direção da reflexão especular $\mathbf{R}$ para os raios secundários depende da normal à superfície e da direção da luz indidente

$$\mathbf{R} = \mathbf{u} - (2\mathbf{u} \cdot \mathbf{N})\mathbf{N}$$

Refração

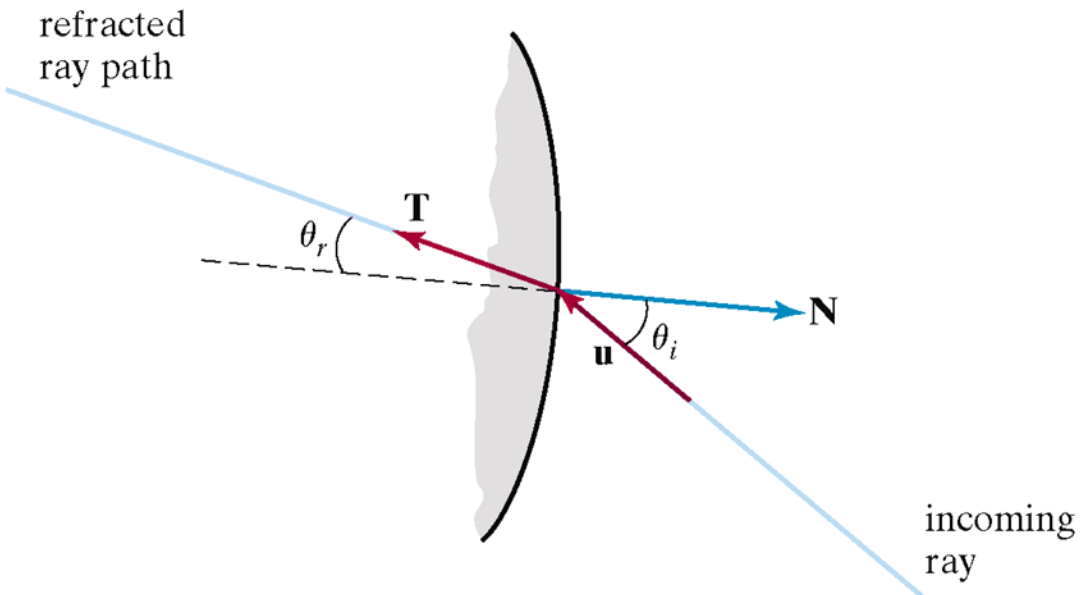
- Para superfícies transparentes, a direção da luz transmitida pode ser calculada traçando um raio secundário na direção de transmissão $\mathbf{T}$

$$\mathbf{T} = \frac{\eta_i}{\eta_r}\mathbf{u} - \left(\cos\theta_r - \frac{\eta_i}{\eta_r}\cos\theta_i\right)\mathbf{N}$$

Algoritmo Básico de Ray Tracing

- O ângulo de refração θ_r pode ser calculado a partir da lei de Snell

$$\cos\theta_r = \sqrt{1 - \left(\frac{\eta_i}{\eta_r}\right)^2 (1 - \cos^2\theta_i)}$$



Cálculos de Interseção Superfície-Raio

- Após completar a árvore binária para um pixel, a **intensidade do pixel é calculada** acumulando as contribuições dos raios a partir dos nós folha da árvore

- A **intensidade** da iluminação na superfície em cada nó da árvore pode ser **atenuada pela distância** à superfície pai (o nó no nível acima na árvore)

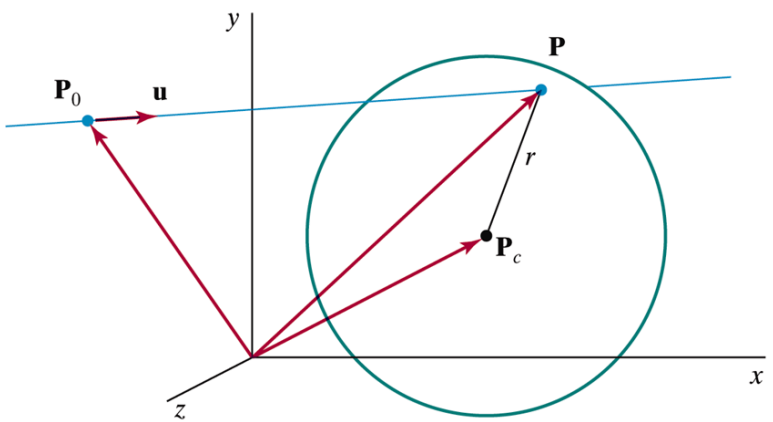
- A intensidade de um pixel é dada pela **soma das intensidades atenuadas** de todos os nós da árvore

- Se o raio primário de um pixel não interceptar nenhum objeto na cena a **árvore é vazia** e o pixel recebe a **cor de fundo**

Cálculos de Interseção Superfície-Raio

- Um raio é descrito por uma posição inicial $\mathbf{P}_0$ e um vetor unitário direcional $\mathbf{u}$
- A **equação paramétrica do raio** permite computador as coordenadas (x, y, z) de um ponto $\mathbf{P}$ ao longo do raio a uma distância s de $\mathbf{P}_0$

$$\mathbf{P} = \mathbf{P}_0 + s\mathbf{u}$$



Cálculos de Interseção Superfície-Raio

Raio Inicial

- A posição inicial $\mathbf{P}_0$ pode ser a posição $\mathbf{P}_{pix}$ do pixel no plano de projeção, ou pode ser a posição do observador $\mathbf{P}_{prp}$

- O vetor $\mathbf{u}$ pode ser obtido fazendo

$$\mathbf{u} = \frac{\mathbf{P}_{pix} - \mathbf{P}_{prp}}{|\mathbf{P}_{pix} - \mathbf{P}_{prp}|}$$

Cálculos de Interseção Superfície-Raio

- Para computar o ponto de intersecção entre a superfície e o raio basta substituir a posição $\mathbf{P}$ na equação da superfície e resolver para obter s
  - O valor de s resultante corresponde à distância, ao longo do raio, do ponto de intersecção ao ponto $\mathbf{P}_0$

- A cada intersecção, atualiza-se $\mathbf{P}_0$ e $\mathbf{u}$ para gerar os raios secundários
  - O novo ponto $\mathbf{P}_0$ é o ponto de intersecção com a superfície
  - O novo vetor $\mathbf{u}$ é a direção de reflexão especular $\mathbf{R}$ para o raio refletido, ou a direção da transmissão $\mathbf{T}$ para o raio transmitido

Intersecções Esfera-Raio

- Os **objetos mais simples** para computar o traçado de raios são as **esferas**

- Considere uma esfera de raio r e centro em $\mathbf{P}_c$ Qualquer ponto $\mathbf{P}$ na superfície da esfera satisfaz

$$|\mathbf{P} - \mathbf{P}_c|^2 - r^2 = 0$$

- Substituindo a equação do raio para $\mathbf{P}$ temos

$$|\mathbf{P}_0 + s\mathbf{u} - \mathbf{P}_c|^2 - r^2 = 0$$

Intersecções Esfera-Raio

- Se $\mathbf{P}_0 - \mathbf{P}_c$ for representado por $\mathbf{\Delta P}$ então

$$|\mathbf{\Delta P} + s\mathbf{u}|^2 - r^2 = 0$$
$$|\mathbf{\Delta P}|^2 + 2\mathbf{\Delta P}s\mathbf{u} + s^2\mathbf{u}^2 - r^2 = 0$$

- Como $||\mathbf{u}|| = 1$ então $\mathbf{u}^2 = 1$, assim

$$s^2 + (2\mathbf{\Delta P u})s + (|\mathbf{\Delta P}|^2 - r^2) = 0$$

- Essa é uma equação quadrática, que pode ser resolvida fazendo

$$s = \frac{-2\mathbf{\Delta P u} \pm \sqrt{4(\mathbf{\Delta P u})^2 - 4(|\mathbf{\Delta P}|^2 - r^2)}}{2}$$

- Portanto, o resultado é

$$s = -\mathbf{\Delta P u} \pm \sqrt{(\mathbf{\Delta P u})^2 - (|\mathbf{\Delta P}|^2 - r^2)}$$

Intersecções Esfera-Raio

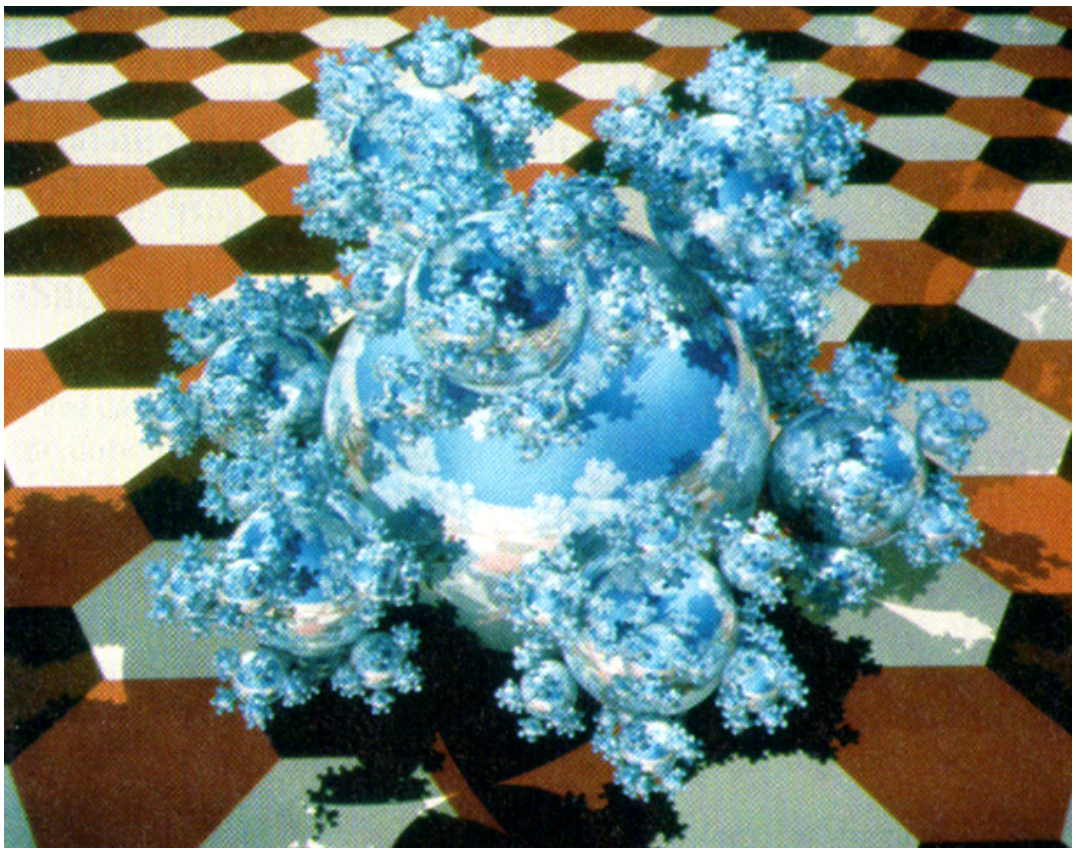
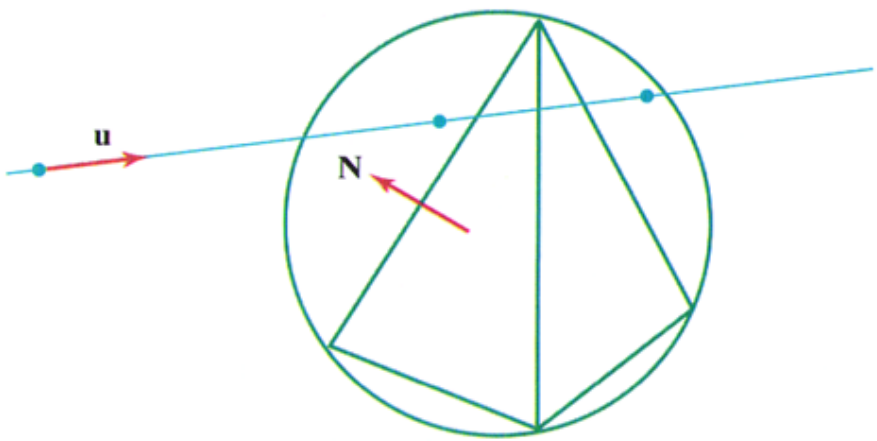


Figura: Um padrão de floco de neve composto apenas por esferas brilhantes, ilustrando as reflexões globais possíveis com o *ray tracing*.

Intersecções Poliedro-Raio

- **Calcular a intersecção de raios com poliedros** é mais **complicado** do que com esferas
  - Normalmente, é mais eficiente calcular intersecções iniciais com um volume envoltório



- Se o raio não intercepta o volume envoltório, o poliedro (i.e., todas as suas faces) é eliminado de testes subsequentes

Intersecções Poliedro-Raio

- Caso o raio intercepte o volume envoltório, é preciso identificar as faces frontais do poliedro

- Para cada face que satisfaz a seguinte inequação (face frontal)

$$\mathbf{u} \cdot \mathbf{N} < 0$$

- Resolve-se a equação do plano

$$\mathbf{N} \cdot \mathbf{P} = -D$$

- Onde D é o quarto parâmetro da Equação do plano e $\mathbf{N} = (A, B, C)$

Intersecções Poliedro-Raio

- Uma posição $\mathbf{P}$ estará no plano e no raio se

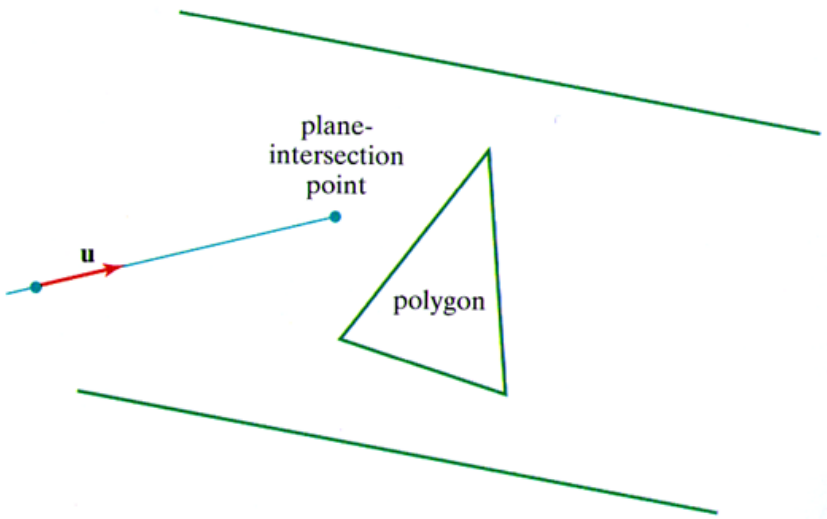
$$\mathbf{N} \cdot (\mathbf{P}_0 + s\mathbf{u}) = -D$$

- E a distância da posição inicial para a intersecção no plano é

$$s = -\frac{D + \mathbf{N} \cdot \mathbf{P}_0}{\mathbf{N} \cdot \mathbf{u}}$$

Intersecções Poliedro-Raio

- Esta equação dá uma posição no plano infinito que contém a face do polígono, mas o ponto de intersecção pode estar além dos limites do polígono



- Então, é preciso verificar se a intersecção está dentro ou fora da área do polígono
  - Similar ao teste executado pelo algoritmo *scanline*

Intersecções Poliedro-Raio

- Esse teste é executado para toda face que satisfaz a desigualdade $\mathbf{u} \cdot \mathbf{N} < 0$

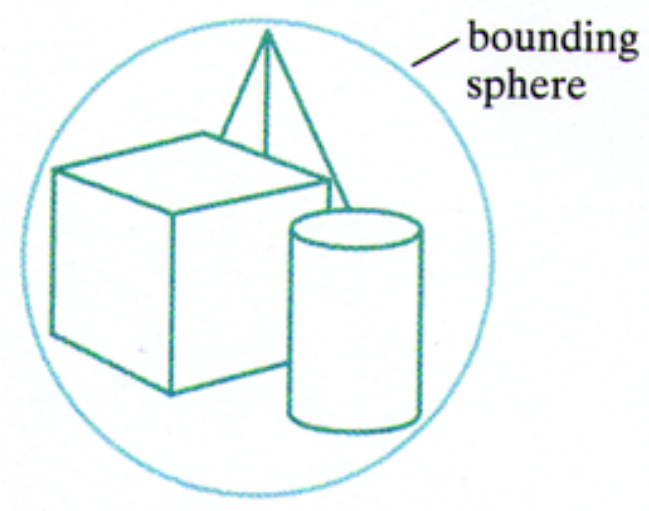
- A menor distância s a um polígono interceptado identifica a posição de intersecção com o poliedro mais próximo do observador
  - Se nenhuma intersecção estiver dentro dos polígonos, o raio não intercepta o poliedro

Reduzindo os Cálculos de Intersecção Raio-Objetos

- Cerca de 95% do processo de *raytracing* é consumido nos cálculos de intersecção entre os raios e as superfícies da cena
  - Muitos métodos foram introduzidos para reduzir o tempo gasto nesses cálculos

Reduzindo os Cálculos de Intersecção Raio-Objetos

- Um método usual consiste em é envolver um conjunto de objetos adjacentes em um volume envoltório, e verificar se o raio intercepta esse volume
  - Se um raio não intercepta o volume, todos os objetos contidos nele são eliminados dos testes de intersecção



- É possível introduzir uma hierarquia de volumes envoltórios para acelerar ainda mais o processo

Sumário

Sumário

Estruturas Básicas

```
1  public class Point3D {
2
3    public Point3D() {
4      this(0, 0, 0);
5    }
6
7    public Point3D(float x, float y, float z) {
8      this.x = x;
9      this.y = y;
10     this.z = z;
11   }
12
13   public float x;
14   public float y;
15   public float z;
16 }
```

Estruturas Básicas

```
1  public class Vector3D {
2
3    public Vector3D() {
4      this(0, 0, 0);
5    }
6
7    public Vector3D(float x, float y, float z) {
8      this.x = x; this.y = y; this.z = z;
9    }
10
11   public Vector3D(Point3D a, Point3D b) {
12     this(b.x - a.x, b.y - a.y, b.z - a.z);
13   }
14
15   public Vector3D normalize() {
16     float norm = norm();
17     x = x / norm; y = y / norm; z = z / norm;
18     return this;
19   }
20
21   public Vector3D multiply(float scalar) {
22     x = x * scalar; y = y * scalar; z = z * scalar;
23     return this;
24   }
25
26   public float norm() {
27     return (float) Math.sqrt(x * x + y * y + z * z);
28   }
29
30   public float x;
31   public float y;
32   public float z;
33 }
```

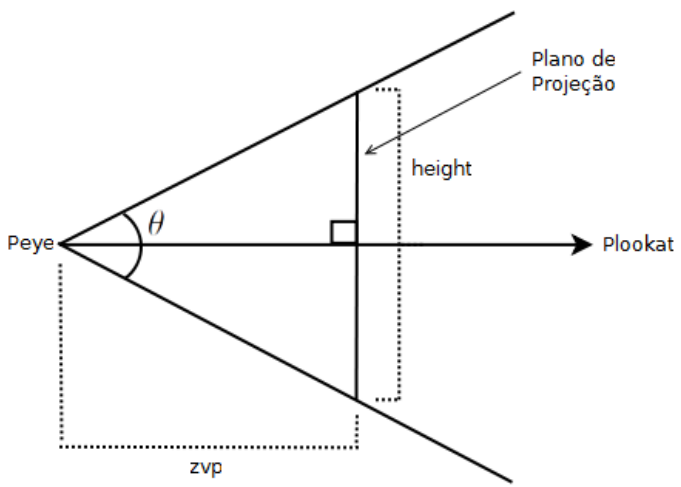
Estruturas Básicas

```
1  public class Util {
2
3    public static float scalar_product(Vector3D a, Vector3D b) {
4      return a.x * b.x + a.y * b.y + a.z * b.z;
5    }
6
7    public static Vector3D cross_product(Vector3D a, Vector3D b) {
8      return new Vector3D(a.y * b.z - a.z * b.y,
9              a.z * b.x - a.x * b.z,
10             a.x * b.y - b.x * a.y);
11   }
12 }
```

Sumário

Parâmetros da Câmera

- Considere os seguintes parâmetros da câmera
  - $\mathbf{P}_{eye}$ a posição da câmera
  - θ o ângulo do campo de visão da câmera
  - $\mathbf{P}_{lookat}$ a posição que a câmera aponta
  - $\mathbf{V}_{up}$ o vetor *view-up* da câmera
  - zvp distância da posição da câmera ao plano de projeção



Câmera Implementação

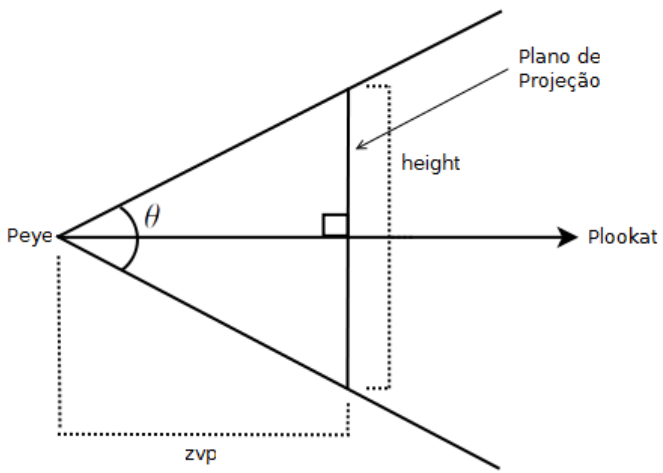
```
1  public class Camera {
2
3    public Camera() {
4      eye = new Point3D(0, 0, 0);
5      lookat = new Point3D(0, 0, -1);
6      viewup = new Vector3D(0, 1, 0);
7      fov = 45;
8      zvp = 1;
9    }
10
11   public float fov; //field of view angle (in degrees)
12   public float zvp; //plane distance from the camera position
13   public Point3D eye; //camera position
14   public Point3D lookat; //position that the camera is looking at
15   public Vector3D viewup; //the view-up vector
16 }
```

Calculando as Dimensões do Plano de Projeção

- Considerando um *frustum* simétrico de projeção perspectiva, a altura ($height$) do plano de projeção é calculada como

$$\tan\left(\frac{\theta}{2}\right) = \frac{heigth/2}{zvp}$$

$$height = 2 \cdot zvp \cdot \tan\left(\frac{\theta}{2}\right)$$



Calculando as Dimensões do Plano de Projeção

- Considere as dimensões da imagem a ser formada Im_{width} e Im_{height} em pixels

- A largura ($width$) do plano de projeção será

$$width = \frac{Im_{width}}{Im_{height}} \cdot heigth$$

Calculando Direções sobre o Plano de Projeção

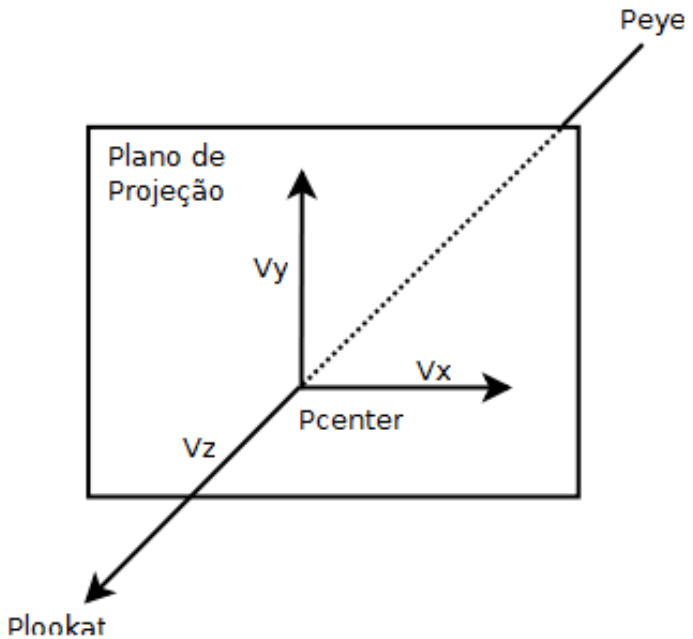
- Considerando a direção de projeção o vetor de $\mathbf{P}_{eye}$ para $\mathbf{P}_{lookat}$

$$\mathbf{V}_z = \frac{\mathbf{P}_{lookat} - \mathbf{P}_{eye}}{||\mathbf{P}_{lookat} - \mathbf{P}_{eye}||}$$

- As direções x e y do plano de projeção são encontradas fazendo

$$\mathbf{V}_x = \frac{\mathbf{V}_z \times \mathbf{V}_{up}}{||\mathbf{V}_z \times \mathbf{V}_{up}||}$$

$$\mathbf{V}_y = \frac{\mathbf{V}_x \times \mathbf{V}_z}{||\mathbf{V}_x \times \mathbf{V}_z||}$$



Encontrando o Primeiro Raio

- Para encontrar a direção do primeiro raio, fazemos as direções x e y terem magnitudes proporcionais ao tamanho de um pixel sobre o plano de projeção

$$\mathbf{V}_x = \frac{width}{Im_{width}} \cdot \mathbf{V}_x$$

$$\mathbf{V}_y = \frac{height}{Im_{height}} \cdot \mathbf{V}_y$$

- A direção do primeiro raio, no canto superior esquerdo do plano de projeção, será

$$\mathbf{V}_{first} = zvp \cdot \mathbf{V}_z + \frac{Im_{height}}{2} \cdot \mathbf{V}_y - \frac{Im_{width}}{2} \cdot \mathbf{V}_x$$

Percorrendo o Plano de Projeção

- Dado o pixel (x, y) na imagem, a direção de um raio qualquer pode ser calculada como

$$\mathbf{V}_{ray} = \mathbf{V}_{first} + x \cdot \mathbf{V}_x + y \cdot \mathbf{V}_y$$

- O pixel do canto superior esquerdo é $(0, 0)$ e do canto inferior direito é $(Im_{width} - 1, Im_{height} - 1)$

- $\mathbf{V}_{ray}$ deve ser normalizado

$$\mathbf{V}_{ray} = \frac{\mathbf{V}_{ray}}{||\mathbf{V}_{ray}||}$$

- A posição de origem dos raios é $\mathbf{P}_{eye}$

Percorrendo o Plano de Projeção

```
1  public class Ray {
2
3    public Ray(Point3D origin, Vector3D direction, float energy) {
4      this.origin = origin;
5      this.direction = direction;
6      this.energy = energy;
7    }
8
9    public Ray(Point3D origin, Vector3D direction) {
10     this(origin, direction, 1.0f);
11   }
12
13   public Point3D origin; //the ray origin
14   public Vector3D direction; //the ray direction
15   public float energy; //the ray energy
16 }
```

Percorrendo o Plano de Projeção

```
1  public class RayShooter {
2
3    public RayShooter(Camera camera, Dimension image) {
4      this.camera = camera;
5
6      //projection direction
7      Vector3D zdir = new Vector3D(camera.eye, camera.lookat).normalize();
8
9      //projection plane x and y directions
10     xdir = Util.cross_product(zdir, camera.viewup).normalize();
11     ydir = Util.cross_product(xdir, zdir).normalize();
12
13     //calculating the width and height of the projection plane
14     float height = 2*camera.zvp * (float)Math.tan(Math.toRadians(camera.fov)/2);
15     float width = ((float) image.width / (float) image.height) * height;
16
17     //the x and y direction vectors will have the size of one pixel
18     xdir.multiply(width / image.width);
19     ydir.multiply(height / image.height);
20
21     //creating the first ray (top left of the projection plane)
22     fray = new Vector3D(camera.eye, camera.lookat).normalize().multiply(camera.↩
           zvp);
23     fray.x += image.height / 2 * ydir.x - image.width / 2 * xdir.x;
24     fray.y += image.height / 2 * ydir.y - image.width / 2 * xdir.y;
25     fray.z += image.height / 2 * ydir.z - image.width / 2 * xdir.z;
26   }
27
28   private Camera camera; //camera
29   private Vector3D xdir; //x-direction on the projection plane
30   private Vector3D ydir; //y-direction on the projection plane
31   private Vector3D fray; //the first ray
32 }
```

Percorrendo o Plano de Projeção

```
1  public class RayShooter {
2    ...
3
4    public Ray getRay(float x, float y) {
5      //calculating the ray direction based on the initial ray
6      Vector3D raydir = new Vector3D(fray.x, fray.y, fray.z);
7      raydir.x += x * xdir.x - y * ydir.x;
8      raydir.y += x * xdir.y - y * ydir.y;
9      raydir.z += x * xdir.z - y * ydir.z;
10
11     //the ray origin is the camera position
12     return new Ray(camera.eye, raydir.normalize());
13   }
14 }
```

Sumário

Cálculos de Intersecção

- Após o lançamento de um raio, é preciso calcular as possíveis intersecções com todos os objetos de uma cena

- Retornar o objeto cujo ponto de intersecção é o mais próximo da origem do raio

- Encontrado o ponto de intersecção, é preciso calcular a normal nesse ponto, necessária para o modelo de iluninação

Implementação

```
1  public abstract class AbstractObject {
2
3    public AbstractObject() {
4      material = new Material();
5    }
6
7    /**
8     * Return the distance of the intersection between the object and a ray.
9     * Return -1 if there is no intersection.
10    * @param ray The ray.
11    * @return The distance of the intersection point and the origin of the ray.
12    */
13   public abstract float intersect(Ray ray);
14
15   /**
16    * Return the normal vector given a intersection point with the object.
17    * @param intersection The intersection point.
18    * @return The normal vector.
19    */
20   public abstract Vector3D normal(Point3D intersection);
21
22   public Material material; //object material (shading)
23 }
```

Implementação

```
1  public class Scene {
2
3    public Scene() {
4      objects = new ArrayList<AbstractObject>();
5      lights = new ArrayList<Light>();
6      camera = new Camera();
7      background = new Color();
8    }
9
10   ...
11
12   public Color background; //background color of the scene
13   public Camera camera; //camera
14   public ArrayList<AbstractObject> objects; //objects on the scene
15   public ArrayList<Light> lights; //lights on the scene
16 }
```

Implementação

```
1  public class Scene {
2    ...
3
4    public Intersection intersect(Ray ray) {
5      float mindist = Float.POSITIVE_INFINITY;
6      Intersection pair = new Intersection();
7
8      //find the closest object given a ray origin
9      for (int i = 0; i < objects.size(); i++) {
10       float dist = objects.get(i).intersect(ray);
11
12       if (dist >= 0 && mindist > dist) {
13         mindist = dist;
14         pair.obj = objects.get(i);
15         pair.dist = dist;
16       }
17     }
18
19     return (pair.obj == null) ? null : pair;
20   }
21
22   public static class Intersection {
23
24     public AbstractObject obj;
25     public float dist;
26   }
27 }
```

Calculando Intersecção com Esferas

- Considere a equação de uma esfera centrada na origem

$$x^2 + y^2 + z^2 = r^2$$

- Considere a equação paramétrica de um raio (dado sua origem e direção)

$$x = x_{or} + s \cdot x_{dir}$$
$$y = y_{or} + s \cdot y_{dir}$$
$$z = z_{or} + s \cdot z_{dir}$$

- Substituindo na equação da esfera temos

$$s^2 \cdot x_{dir}^2 + 2 \cdot x_{or} \cdot x_{dir} + x_{or}^2 +$$
$$s^2 \cdot y_{dir}^2 + 2 \cdot y_{or} \cdot y_{dir} + y_{or}^2 +$$
$$s^2 \cdot z_{dir}^2 + 2 \cdot z_{or} \cdot z_{dir} + z_{or}^2 - r^2 = 0$$

Calculando Intersecção com Esferas

- Esta é uma quádrica da forma

$$A \cdot s^2 + B \cdot s + C = 0$$

- Com

$$A = x_{dir}^2 + y_{dir}^2 + z_{dir}^2$$
$$B = 2 \cdot (x_{or} \cdot x_{dir} + y_{or} \cdot y_{dir} + z_{or} \cdot z_{dir})$$
$$C = x_{or}^2 + y_{or}^2 + z_{or}^2 - r^2$$

- Então temos solução

$$s = \frac{-B \pm \sqrt{B^2 - 4 \cdot A \cdot C}}{2 \cdot A}$$

Calculando Intersecção com Esferas

- Como o raio é normalizado, então

$$A = x_{dir}^2 + y_{dir}^2 + z_{dir}^2 = 1$$

- Considerando que

$$B = 2 \cdot D$$

- Com

$$D = x_{or} \cdot x_{dir} + y_{or} \cdot y_{dir} + z_{or} \cdot z_{dir}$$

- Então, temos

$$s = \frac{-2 \cdot D \pm \sqrt{4 \cdot D^2 - 4 \cdot c}}{2} = -D \pm \sqrt{D^2 - C}$$

Calculando Intersecção com Esferas

- Se $D^2 - C < 0$ não existe solução, caso contrário uma ou duas soluções serão encontradas
  - Escolha a solução de menor valor (mais próxima à origem do raio)

Calculando a Normal

- Considerando que $\mathbf{P}_i$ é o ponto de intersecção sobre a esfera, $\mathbf{P}_c$ o centro da esfera, e r o raio, a normal no ponto de intersecção será

$$\mathbf{N} = (\mathbf{P}_i - \mathbf{P}_c)/r$$

Calculando Intersecção com Esferas

```
1  public class Sphere extends AbstractObject {
2
3    public Sphere() {
4      this(new Point3D(), 1);
5    }
6
7    public Sphere(Point3D center, float radius) {
8      this.center = center;
9      this.radius = radius;
10   }
11
12   ...
13
14   @Override
15   public Vector3D normal(Point3D intersection) {
16     Vector3D norm = new Vector3D();
17     norm.x = (intersection.x - center.x) / radius;
18     norm.y = (intersection.y - center.y) / radius;
19     norm.z = (intersection.z - center.z) / radius;
20     return norm;
21   }
22
23   public Point3D center;
24   public float radius;
25 }
```

Calculando Intersecção com Esferas

```
1  public class Sphere extends AbstractObject {
2    ...
3
4    @Override
5    public float intersect(Ray ray) {
6      //translate ray origin in order to put the center sphere on the system
7      //origin (0,0,0)
8      float x = ray.origin.x - center.x;
9      float y = ray.origin.y - center.y;
10     float z = ray.origin.z - center.z;
11
12     float b = x * ray.direction.x + y * ray.direction.y + z * ray.direction.z;
13     float t = b * b - x * x - y * y - z * z + radius * radius;
14
15     if (t < 0) { //no intersection
16       return -1;
17     }
18
19     float s = -b - (float) Math.sqrt(t);
20     if (s > 0) { //distance from the origin on the given direction
21       return s;
22     }
23
24     s = -b + (float) Math.sqrt(t);
25     if (s > 0) { //distance from the origin on the given direction
26       return s;
27     }
28
29     return -1; //no intersection
30   }
31 }
```

Sumário

Tonalização

- Tendo o ponto de intersecção e a normal $\mathbf{N}$ nesse ponto, pode-se calcular o modelo de iluminação

Modelo de Iluminação

- O modelo de iluminação é o modelo básico de Blinn-Phong

$$
\begin{aligned}
I &= I_{ambdiff} + \sum_{l=1}^{n} [I_{l,diff} + I_{l,spec}] \\
&= k_a \ I_a + \sum_{l=1}^{n} I_l [k_d \max(0.0, (\mathbf{N} \cdot \mathbf{L})) + k_s \max(0.0, (\mathbf{N} \cdot \mathbf{H})^{n_s})]
\end{aligned}
$$

Implementação

```
1  public class Material {
2
3    public Material() {
4      ambient = new Component(0.5f, 0.5f, 0.5f);
5      diffuse = new Component(0.5f, 0.5f, 0.5f);
6      specular = new Component(0.5f, 0.5f, 0.5f);
7      reflt = 1; transp = 0;
8    }
9    ...
10
11   public static class Component {
12     public Component() {
13       this(0, 0, 0);
14     }
15
16     public Component(float red, float green, float blue) {
17       this.red = red; this.green = green; this.blue = blue;
18     }
19
20     public float red;
21     public float green;
22     public float blue;
23   }
24
25   public Component ambient; //[0,1]
26   public Component diffuse; //[0,1]
27   public Component specular; //[0,1]
28   public float ns; //specular coefficient [0,inf]
29   public float reflt; //reflection [0,1]
30   public float transp; // transparency [0,1]
31 }
```

Implementação

```
1  public class Material {
2    ...
3
4    public void setAmbient(float red, float green, float blue) {
5      ambient.red = red; ambient.green = green; ambient.blue = blue;
6    }
7
8    public void setDifusse(float red, float green, float blue) {
9      diffuse.red = red; diffuse.green = green; diffuse.blue = blue;
10   }
11
12   public void setSpecular(float red, float green, float blue) {
13     specular.red = red; specular.green = green; specular.blue = blue;
14   }
15 }
```

Implementação

```
1  public class Light {
2
3    public Light() {
4      position = new Point3D(100, 100, 0);
5      ambient = new Color(125, 125, 125);
6      diffuse = new Color(255, 255, 255);
7      specular = new Color(255, 255, 255);
8    }
9
10   public Point3D position; //light position
11   public Color ambient; //[0,255]
12   public Color diffuse; //[0,255]
13   public Color specular; //[0,255]
14 }
```

Implementação

```
1  public class Color {
2
3    public Color() {
4      this(0, 0, 0);
5    }
6
7    public Color(float red, float green, float blue) {
8      this.red = red;
9      this.green = green;
10     this.blue = blue;
11   }
12
13   public Color multiply(float factor) {
14     return new Color(red * factor, green * factor, blue * factor);
15   }
16
17   public int toRGB() {
18     int r = Math.min(Math.max(Math.round(red), 0), 255);
19     int g = Math.min(Math.max(Math.round(green), 0), 255);
20     int b = Math.min(Math.max(Math.round(blue), 0), 255);
21     return new java.awt.Color(r, g, b).getRGB();
22   }
23
24   public float red;
25   public float green;
26   public float blue;
27 }
```

Tonalização

```
1  private Color phong(AbstractObject object, Scene scene, Vector3D normal,
2          Vector3D reflectionray, Point3D intersection) {
3     Color color = new Color(); //final color
4     Material material = object.material; //object's material
5
6     //for each light
7     for (int i = 0; i < scene.lights.size(); i++) {
8       //calculate the light direction
9       Light light = scene.lights.get(i);
10      Vector3D lightray = new Vector3D(intersection,light.position).normalize();
11
12      //adding the ambient contribution
13      color.red += material.ambient.red * light.ambient.red;
14      color.green += material.ambient.green * light.ambient.green;
15      color.blue += material.ambient.blue * light.ambient.blue;
16
17      float diff = Util.scalar_product(normal, lightray);
18      if (diff > 0) {
19        //calculating the difusse component
20        color.red += material.diffuse.red * light.diffuse.red * diff;
21        color.green += material.diffuse.green * light.diffuse.green * diff;
22        color.blue += material.diffuse.blue * light.diffuse.blue * diff;
23
24        //calculating the specular component
25        float spec = Util.scalar_product(reflectionray, lightray);
26        if (spec > 0) {
27          spec = Math.max(0, (float) Math.pow(spec, material.ns));
28          color.red += material.specular.red * light.specular.red * spec;
29          color.green += material.specular.green * light.specular.green * spec;
30          color.blue += material.specular.blue * light.specular.blue * spec;
31        }
32      }
33    }
34
35    return color;
36  }
```

Sumário

Ray Casting

- No *Ray Casting* são lançados os raios primários e somente a intersecção desses com os objetos são consideradas para criar a imagem final
  - Não há reflexões e refrações geradas pelo traçado de raios secundários

Ray Casting

```
1  public class RayCasting {
2
3    public BufferedImage execute(Scene scene, Dimension imagesize) {
4      //creating the final image
5      BufferedImage image = new BufferedImage(imagesize.width,
6             imagesize.height, BufferedImage.TYPE_INT_ARGB);
7
8      //processing the viewing parameters
9      RayShooter shooter = new RayShooter(scene.camera, imagesize);
10
11     for (int y = 0; y < imagesize.height; y++) {
12       for (int x = 0; x < imagesize.width; x++) {
13         Color color = scene.background; //final color is initially background color
14         Ray ray = shooter.getRay(x, y); //calculate the ray
15         Intersection intersect = scene.intersect(ray); //get the intersection
16
17         if (intersect != null) {
18           //calculate the intersection point
19           Point3D inters = new Point3D();
20           inters.x = ray.origin.x + intersect.dist * ray.direction.x;
21           inters.y = ray.origin.y + intersect.dist * ray.direction.y;
22           inters.z = ray.origin.z + intersect.dist * ray.direction.z;
23
24           Vector3D normal = intersect.obj.normal(inters); //calculate the normal
25
26           //calculate the reflected ray direction
27           float k = 2 * Util.scalar_product(ray.direction, normal);
28           Vector3D reflray = new Vector3D(ray.direction.x - k * normal.x,
29                  ray.direction.y - k * normal.y,
30                  ray.direction.z - k * normal.z).normalize();
31
32           color = phong(intersect.obj, scene, normal, reflray, inters);
33         }
34
35         //set the color
36         image.setRGB(x, y, color.toRGB());
37       }
38     }
39
40     return image;
41   }
42 }
```

Criando uma Cena

```
1  Scene scene = new Scene();
2
3  Light light1 = new Light();
4  light1.position = new Point3D(10, 10, 10);
5  scene.lights.add(light1);
6
7  Camera camera = new Camera();
8  camera.eye = new Point3D(4, 0, 4);
9  camera.lookat = new Point3D(0, 0, 0);
10 camera.viewup = new Vector3D(0, 1, 0);
11 camera.fov = 35;
12 scene.camera = camera;
```

Criando uma Cena

```
1  Material material1 = new Material();
2  material1.setAmbient(0.1f, 0.1f, 0.1f);
3  material1.setDifusse(0.8f, 0.08f, 0.8f);
4  material1.setSpecular(0.98f, 0.8f, 0.98f);
5  material1.ns = 300;
6  material1.reflt = 0.5f;
7
8  Sphere sphere1 = new Sphere();
9  sphere1.center = new Point3D(0, 0, 0);
10 sphere1.radius = 1;
11 sphere1.material = material1;
12 scene.objects.add(sphere1);
```

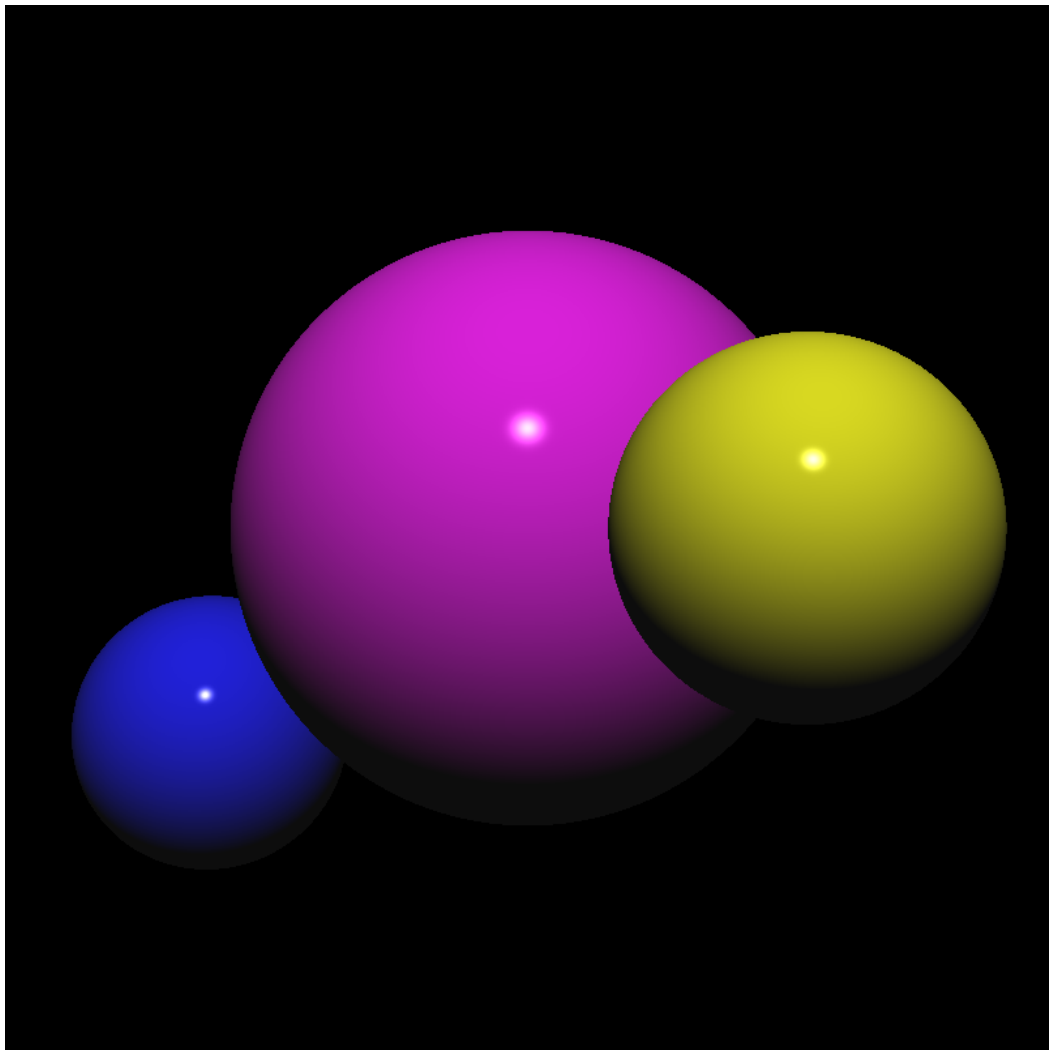
Criando uma Cena

```
1  Material material2 = new Material();
2  material2.setAmbient(0.1f, 0.1f, 0.1f);
3  material2.setDifusse(0.8f, 0.8f, 0.08f);
4  material2.setSpecular(0.98f, 0.98f, 0.8f);
5  material2.ns = 300;
6  material2.reflt = 0.15f;
7
8  Sphere sphere2 = new Sphere();
9  sphere2.center = new Point3D(1.5f, 0, 0.5f);
10 sphere2.radius = 0.5f;
11 sphere2.material = material2;
12 scene.objects.add(sphere2);
```

Criando uma Cena

```
1  Material material3 = new Material();
2  material3.setAmbient(0.1f, 0.1f, 0.1f);
3  material3.setDifusse(0.08f, 0.08f, 0.8f);
4  material3.setSpecular(0.98f, 0.98f, 0.98f);
5  material3.ns = 300;
6  material3.reflt = 0.25f;
7
8  Sphere sphere3 = new Sphere();
9  sphere3.center = new Point3D(-1.15f, -0.75f, 0.5f);
10 sphere3.radius = 0.5f;
11 sphere3.material = material3;
12 scene.objects.add(sphere3);
13
14 RayCasting rc = new RayCasting();
15 BufferedImage image = rc.execute(scene, new Dimension(800, 800));
16 ImageView.getInstance().display(image);
```

Ray Casting



Ray Casting

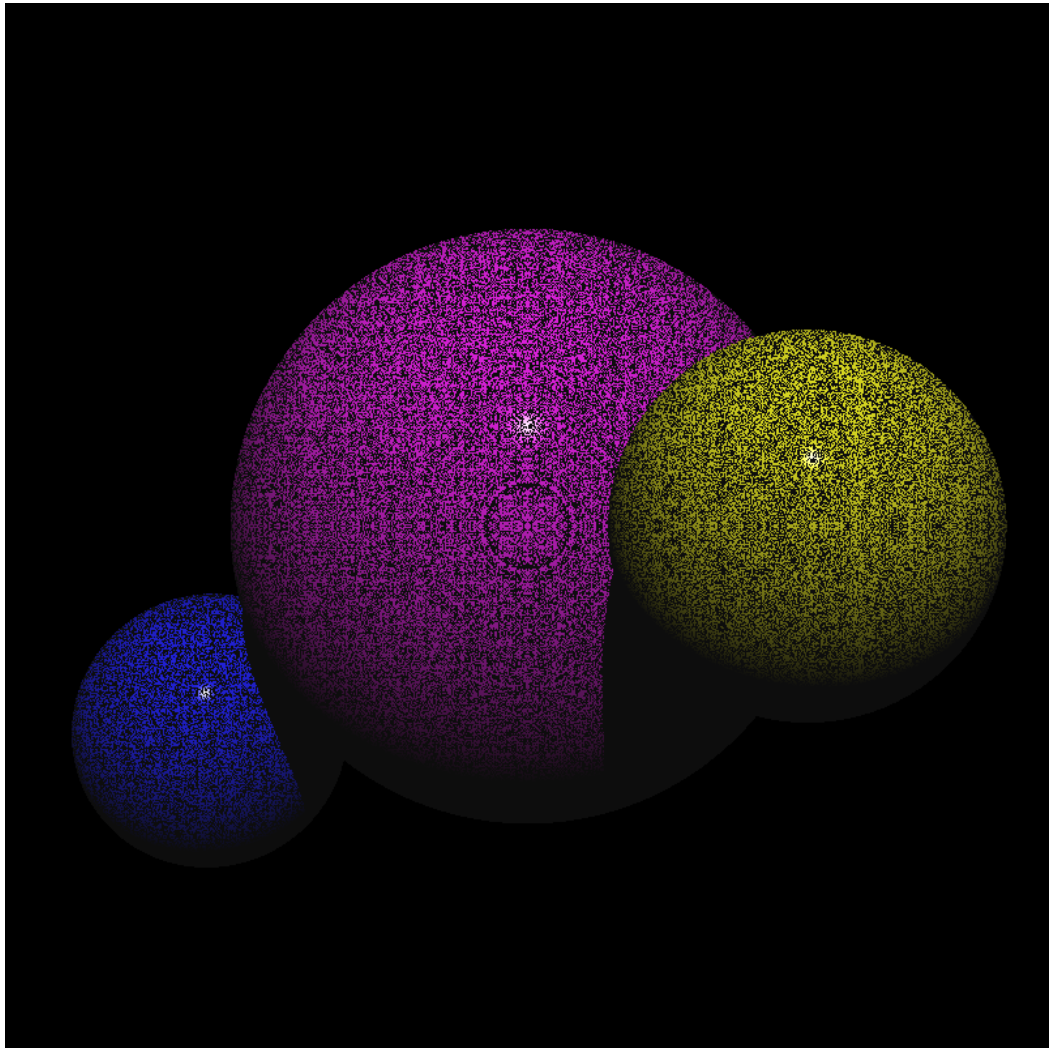
Sombras

- O efeito de sombras (*hard*) pode ser obtido modificando a rotina de Phong para considerar os **raios de sombra**
  - Gera raio entre o ponto de intersecção e a(s) fonte(s) de luz
  - Se esse raio intercepta algum objeto, o ponto de intersecção está na sombra em relação a essa fonte, e somente contribuição ambiente deve ser computada no modelo de iluminação
  - Se não houver intersecção, calcula o modelo de iluminação normalmente

Ray Casting

```
1  private Color phong(AbstractObject object, Scene scene, Vector3D normal,
2          Vector3D refldir, Point3D intersection) {
3    Color color = new Color(); //final color
4    Material material = object.material; //material of the object
5
6    //for each light
7    for (int i = 0; i < scene.lights.size(); i++) {
8      Light light = scene.lights.get(i);
9
10     //adding the ambient contribution
11     color.red += material.ambient.red * light.ambient.red;
12     color.green += material.ambient.green * light.ambient.green;
13     color.blue += material.ambient.blue * light.ambient.blue;
14
15     //shadow ray
16     Ray shadowray = new Ray(intersection, new Vector3D(intersection, light.↩
           position).normalize());
17     Intersection intersect = scene.intersect(shadowray);
18
19     //there is not an object between the light and the object
20     if (intersect == null) {
21       //calculate the light direction
22       Vector3D lightray = new Vector3D(intersection,light.position).normalize();
23
24       //calculate the phong
25       //...
26     }
27   }
28
29   return color;
30 }
```

Ray Casting : Erro Numérico



Ray Casting : Erro Numérico

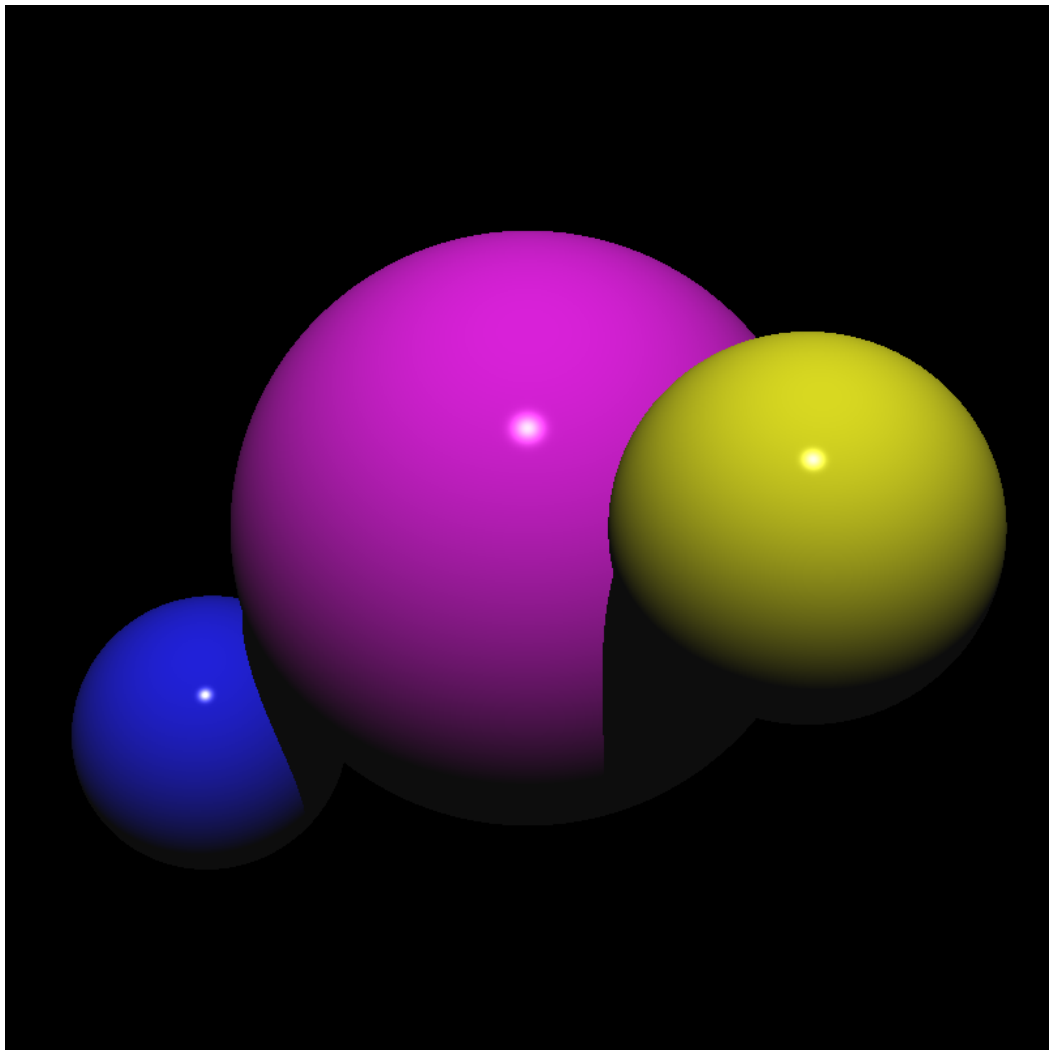
Problemas

- Por erros numéricos, o cálculo dos pontos de intersecção não é exato
- Os pontos podem pode cair dentro da esfera, e não exatamente na sua superfície
  - Solução: mover o ponto de intersecção um pouco na direção da normal para garantir que fique fora

Ray Casting

```
1  public class RayCasting {
2
3    public BufferedImage execute(Scene scene, Dimension imagesize) {
4      //creating the final image
5      BufferedImage image = new BufferedImage(imagesize.width,
6             imagesize.height, BufferedImage.TYPE_INT_ARGB);
7
8      //processing the viewing parameters
9      RayShooter shooter = new RayShooter(scene.camera, imagesize);
10
11     for (int y = 0; y < imagesize.height; y++) {
12       for (int x = 0; x < imagesize.width; x++) {
13         Color color = scene.background; //final color is background color
14         Ray ray = shooter.getRay(x, y); //calculate the ray
15         Intersection intersect = scene.intersect(ray); //get the intersection
16
17         if (intersect != null) {
18           //calculate the intersection point
19           Point3D inters = new Point3D();
20           inters.x = ray.origin.x + intersect.dist * ray.direction.x;
21           inters.y = ray.origin.y + intersect.dist * ray.direction.y;
22           inters.z = ray.origin.z + intersect.dist * ray.direction.z;
23
24           Vector3D normal = intersect.obj.normal(inters); //calculate the normal
25
26           //moving the intersection point on the direction of the normal
27           //a small fraction
28           inters.x = inters.x + EPSILON * normal.x;
29           inters.y = inters.y + EPSILON * normal.y;
30           inters.z = inters.z + EPSILON * normal.z;
31
32           ...
33         }
34
35         //set the color
36         image.setRGB(x, y, color.toRGB());
37       }
38     }
39
40     return image;
41   }
42
43   private static final float EPSILON = 0.001f;
44 }
```

Ray Casting



Sumário

Ray Tracing

- Raio primários, mais os raios secundários gerados nas intersecções
  - Raio refletido
  - Raio transmitido

- Computacionalmente mais caro

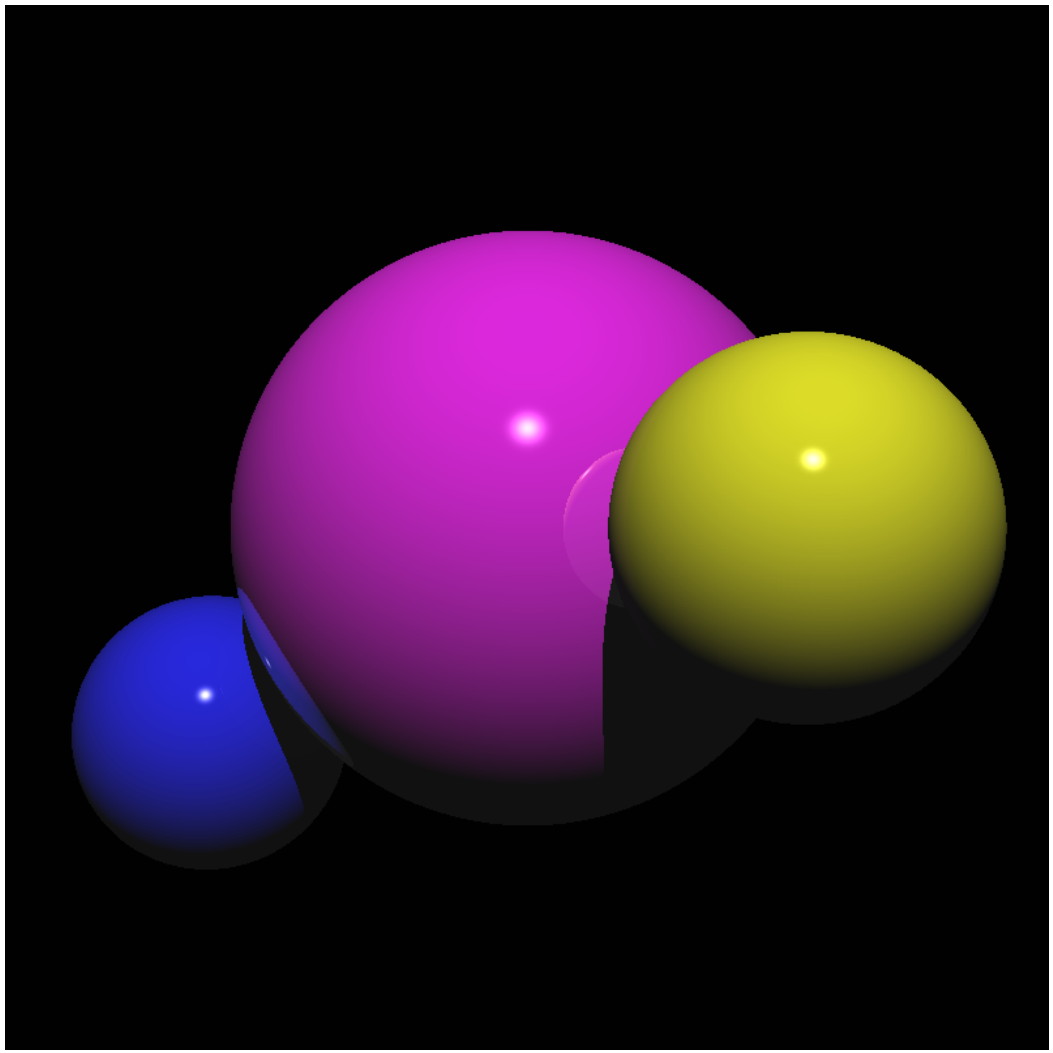
Ray Tracing

```
1  public class RayTracing {
2
3    public RayTracing() {
4      maxsteps = 5;
5    }
6
7    public BufferedImage execute(Scene scene, Dimension imagesize) {
8      //creating the final image
9      BufferedImage image = new BufferedImage(imagesize.width,
10            imagesize.height, BufferedImage.TYPE_INT_ARGB);
11
12     //processgin the viewing parameters
13     RayShooter viewing = new RayShooter(scene.camera, imagesize);
14
15     for (int y = 0; y < imagesize.height; y++) {
16       for (int x = 0; x < imagesize.width; x++) {
17
18         //calculating the ray
19         Ray ray = viewing.getRay(x, y);
20
21         //tracing the rays
22         Color color = trace(scene, ray, 0);
23
24         //set the color
25         image.setRGB(x, y, Util.gammacorrect(color).toRGB());
26       }
27     }
28
29     return image;
30   }
31
32   ...
33
34   private static final float MIN_ENERGY = 0.001f;
35   private static final float EPSILON = 0.001f;
36   private int maxsteps;
37 }
```

Ray Tracing

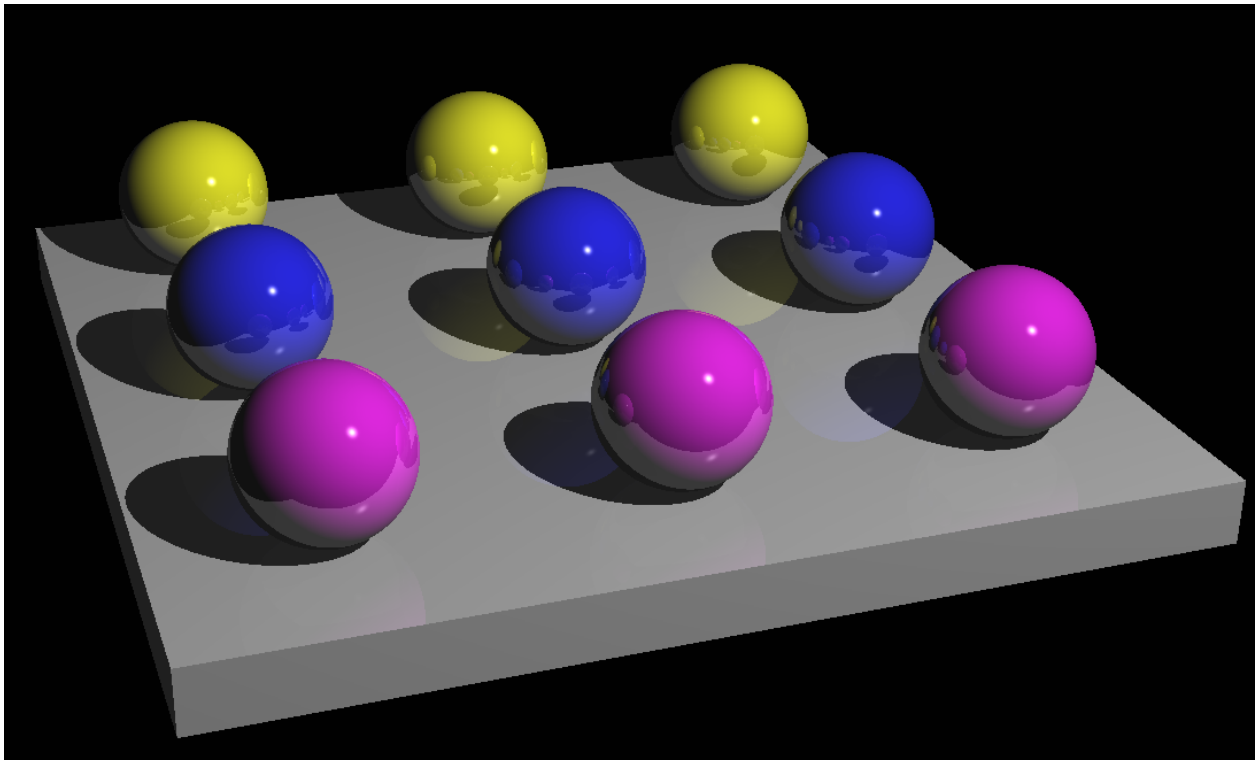
```
1  public class RayTracing {
2    private Color trace(Scene scene, Ray ray, int step) {
3      if (step < maxsteps && ray.energy > MIN_ENERGY) {
4        //getting the intersection
5        Intersection intersect = scene.intersect(ray);
6
7        if (intersect != null) {
8          //calculating the intersection point
9          Point3D intersection = new Point3D();
10         intersection.x = ray.origin.x + intersect.dist * ray.direction.x;
11         intersection.y = ray.origin.y + intersect.dist * ray.direction.y;
12         intersection.z = ray.origin.z + intersect.dist * ray.direction.z;
13
14         //calculating the normal
15         Vector3D normal = intersect.obj.normal(intersection);
16
17         //moving the intersection point on the direction of the normal
18         intersection.x = intersection.x + EPSILON * normal.x;
19         intersection.y = intersection.y + EPSILON * normal.y;
20         intersection.z = intersection.z + EPSILON * normal.z;
21
22         //calculate the reflected ray direction
23         float k = 2 * Util.scalar_product(ray.direction, normal);
24         Vector3D refldir = new Vector3D(ray.direction.x - k * normal.x,
25                ray.direction.y - k * normal.y,
26                ray.direction.z - k * normal.z).normalize();
27
28         //calculating the phong contribution
29         Color local = phong(intersect.obj, scene, normal, refldir, intersection);
30
31         //calculating the transmitted contribution (refraction)... TODO
32
33         //calculating the reflected contribution
34         Ray reflray = new Ray(intersection, refldir, ray.energy * intersect.obj.↩
               material.reflt);
35         Color reflected = trace(scene, reflray, step + 1);
36
37         return new Color((local.red*ray.energy)+(reflected.red*reflray.energy),
38                (local.green*ray.energy)+(reflected.green*reflray.energy),
39                (local.blue*ray.energy)+(reflected.blue*reflray.energy));
40       }
41     }
42
43     return scene.background.multiply(ray.energy);
44   }
45 }
```

Ray Tracing



Ray Tracing

- Para cenas mais ‘anguladas‘, os efeitos de *aliasing* podem ser mais perceptíveis



Ray Tracing

Antialiasing

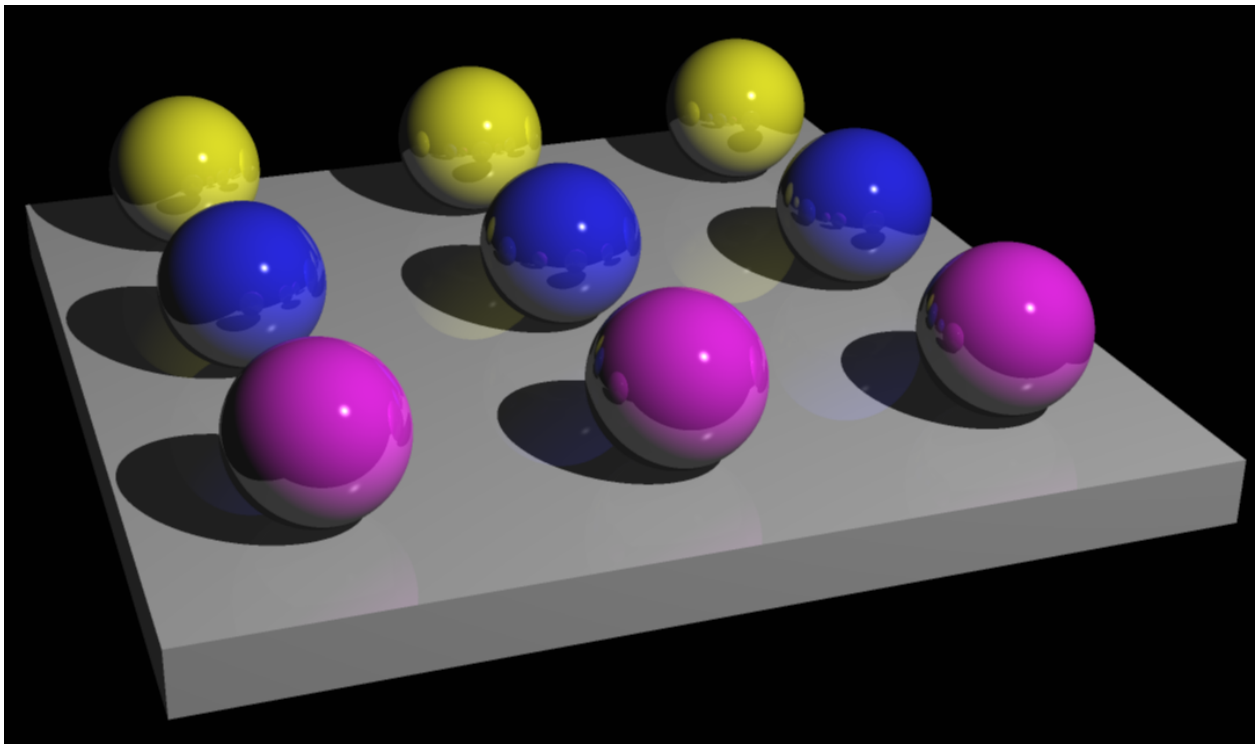
- Para contornar o problema do *aliasing*, pode-se subdividir um pixel em sub-pixels, e lançar raios primários para todos esses sub-pixels
  - A cor final do pixel será dada pela média da cor desses sub-pixels
  - O processo será x vezes mais lento se cada pixel for dividido em x sub-pixels

- O pixel pode ser dividido em sub-pixels de mesma tamanho, ou de tamanhos não uniformes

Ray Tracing

```
1  public class RayTracing {
2    public BufferedImage executeAntialiasing(Scene scene, Dimension imagesize) {
3      //creating the final image
4      BufferedImage image = new BufferedImage(imagesize.width,
5             imagesize.height, BufferedImage.TYPE_INT_ARGB);
6
7      //processgin the viewing parameters
8      RayShooter viewing = new RayShooter(scene.camera, imagesize);
9
10     for (int y = 0; y < imagesize.height; y++) {
11       for (int x = 0; x < imagesize.width; x++) {
12         Color color = new Color(); //the final color
13
14         for (float ypart = -0.75f; ypart < 1; ypart += 0.5f) {
15           for (float xpart = -0.75f; xpart < 1; xpart += 0.5f) {
16             //calculating the ray
17             Ray ray = viewing.getRay(x + xpart, y + ypart);
18
19             //tracing the rays
20             Color colorpart = trace(scene, ray, 0);
21
22             //summing-up to the final color
23             color.red += colorpart.red;
24             color.green += colorpart.green;
25             color.blue += colorpart.blue;
26           }
27         }
28
29         //compute the average of the anti-aliasing rays
30         color.red = color.red / 16.0f;
31         color.green = color.green / 16.0f;
32         color.blue = color.blue / 16.0f;
33
34         //set the color
35         image.setRGB(x, y, Util.gammacorrect(color).toRGB());
36       }
37     }
38
39     return image;
40   }
41 }
```

Ray Tracing



Ray Tracing x Ray Casting

- *Ray Tracing* é melhor do que *Ray Casting*?

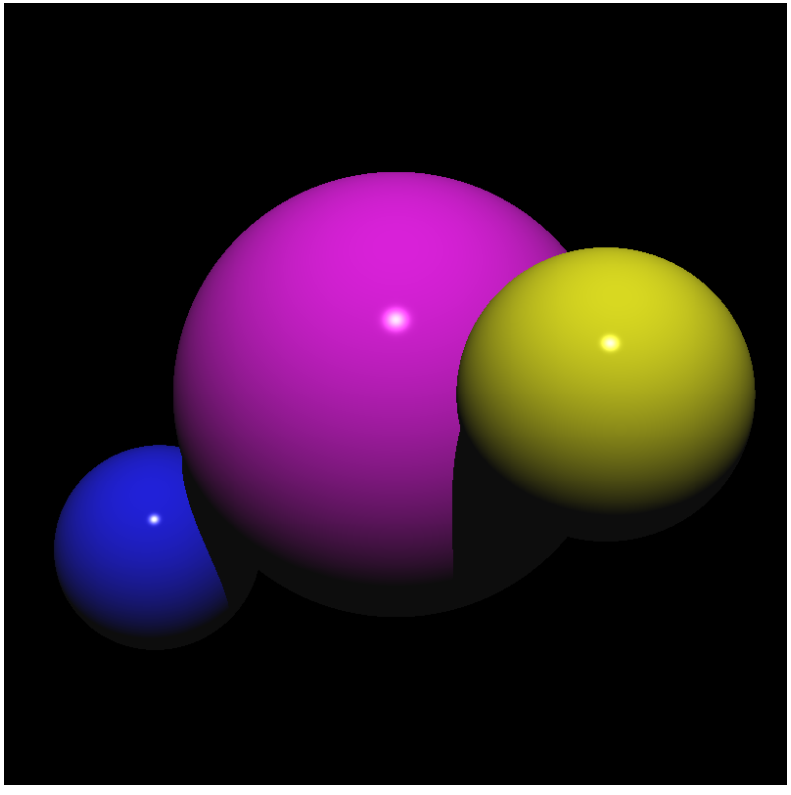


Figura: Ray Casting

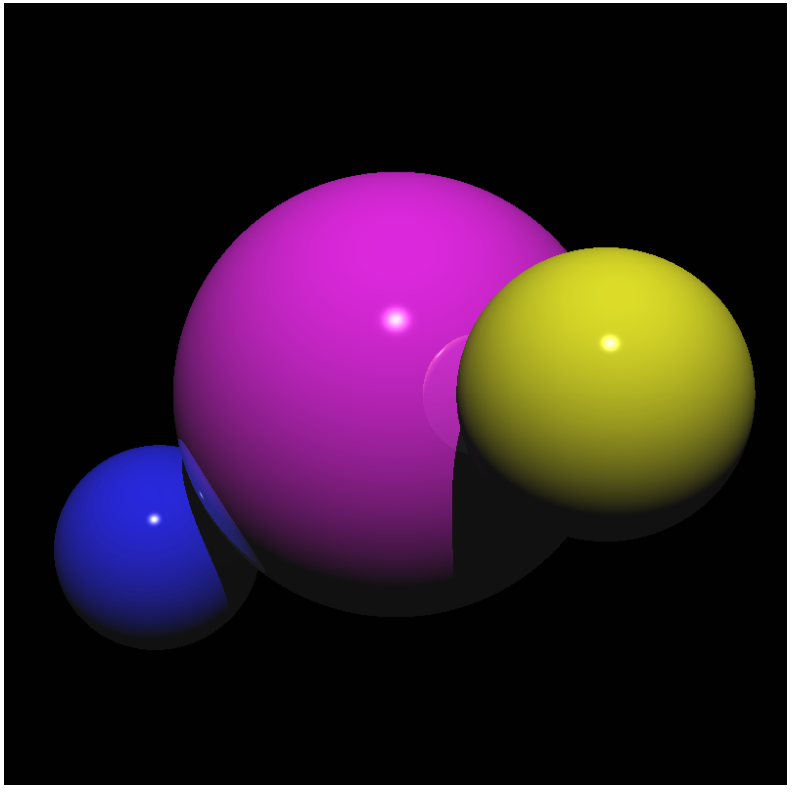


Figura: Ray Tracing

Ray Tracing

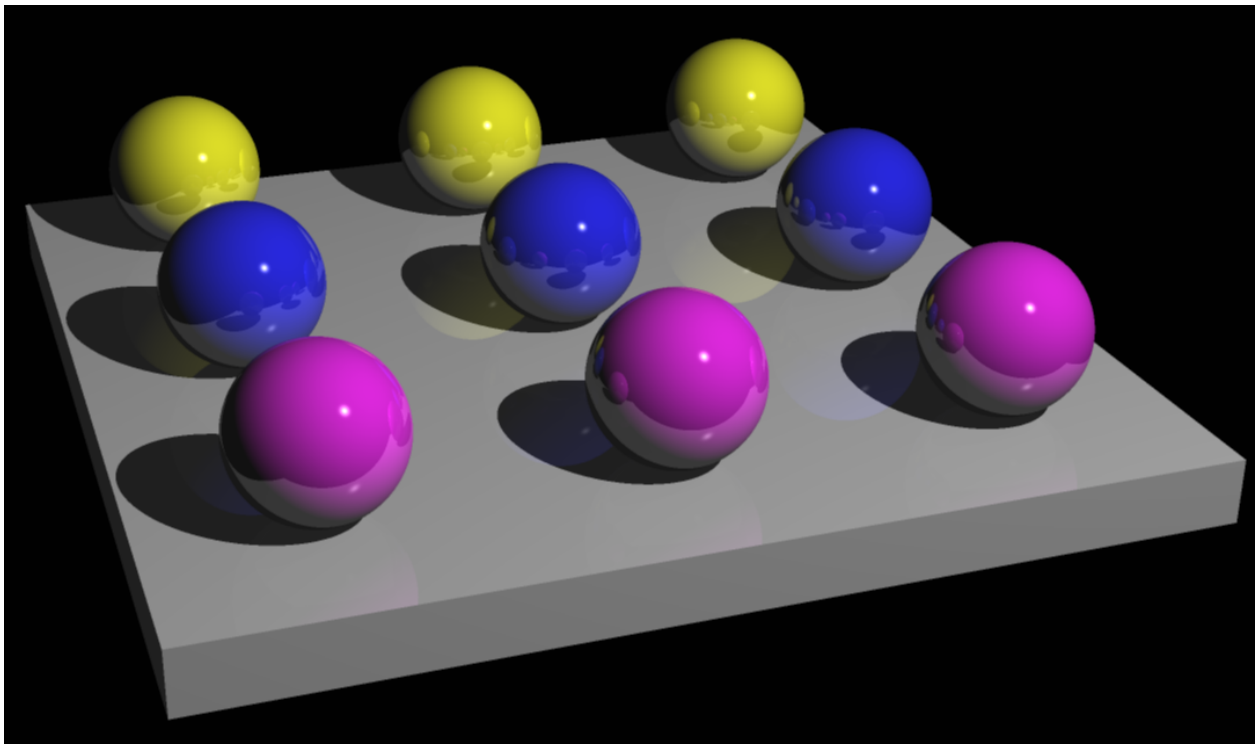


Figura: Ray Tracing

Ray Tracing

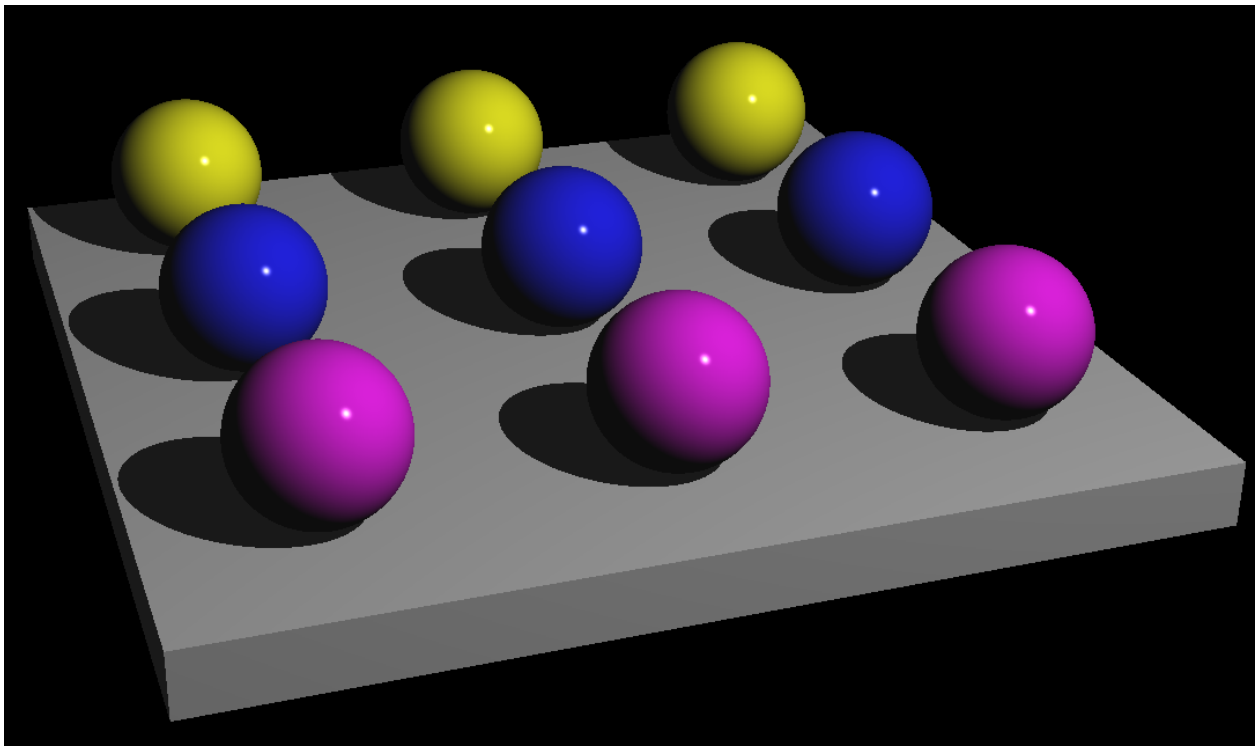


Figura: Ray Casting

Ray Casting

Melhorias possíveis

- Sombras mais realísticas: *soft shadows*
- Aceleração do processo usando subdivisão espacial (*octree*)
- Introdução de efeitos de transmissão (transparência)
- Adição de texturas
- Cálculo de intersecção com outros tipos de objetos (malhas poligonais)
- Efeitos de foco da lente da câmera