Abitur 2016
Julius Hinze

JAVA - RayTracing

Wie werden fotorealistische Bilder am Computer “synthetisiert” ?

Gutenbergschule
Wiesbaden

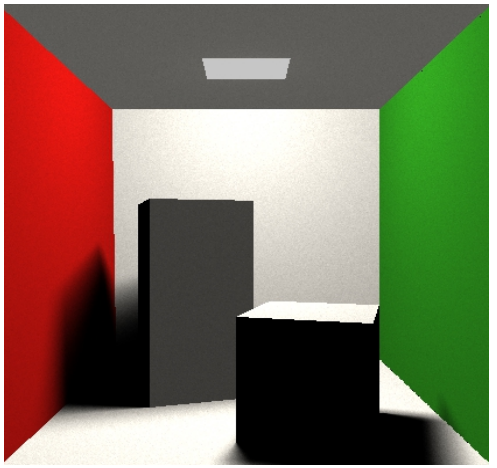
Übersicht

- Der RayTracing Algorithmus
- Rekursives RayTracing
- Beleuchtungsmodelle
- Monte Carlo Experiment : “PathTracing”

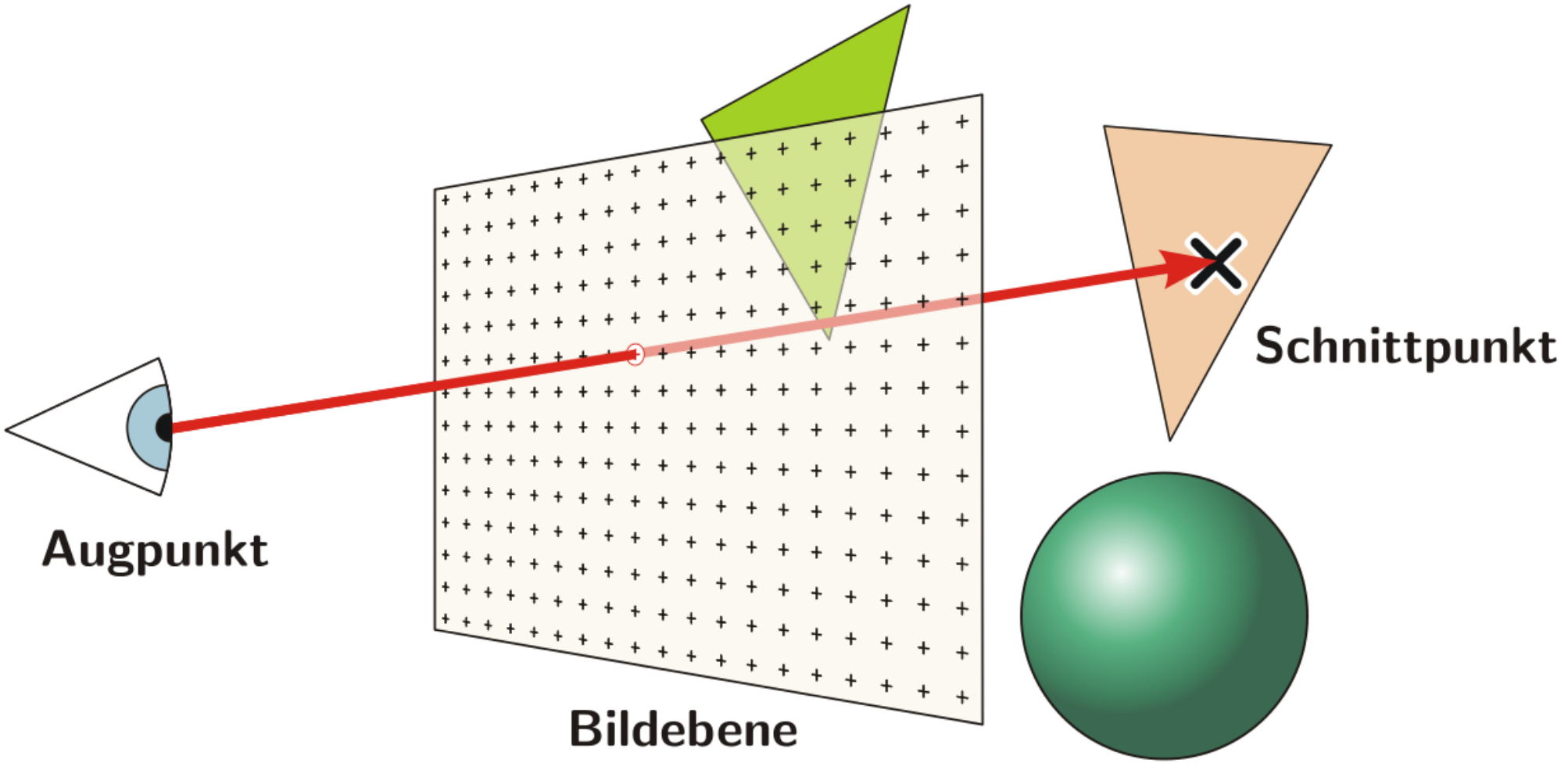
- Beschleunigungsverfahren
- Implementierung
- Quellen

Cornell Box

- Raum um physikalisch korrektes Rendering zu testen
- Vergleich: Realität & Simulation
- Wenige Dreiecke, einfacher Aufbau
- .obj Datei frei verfügbar



Was ist “RayTracing” ?



Schnittpunkt: Kugel

Kugel

- $| x-c |^2 = r^2$
- c → Mittelpunkt (“center”)
- r → Radius
- X → alle Punkte der Kugel

Strahl

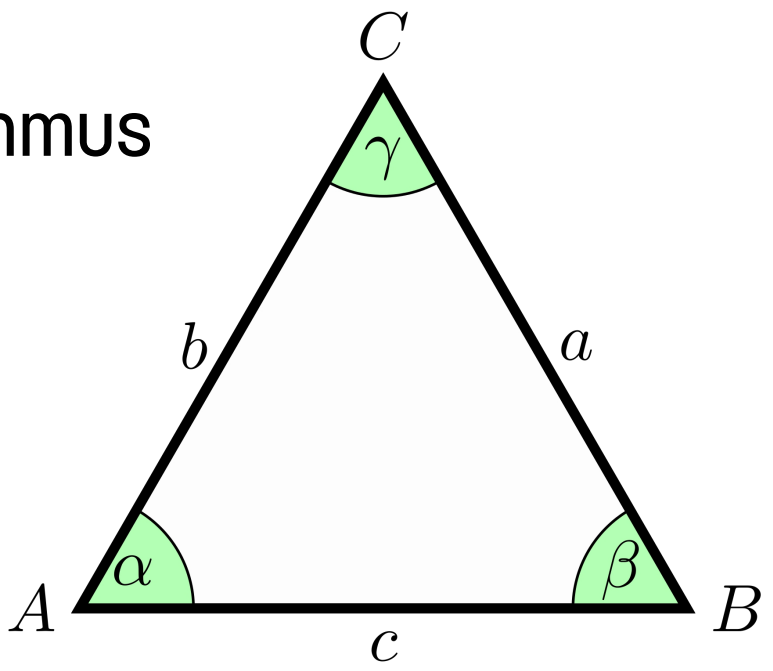
- X = O + d * l
- d → Distanz
- l → Richtung
- O → Ursprung
- X → Punkter der Linie

→ Einsetzen von Gerade in Kugelgleichung erzeugt bis zu zwei Schnittpunkte

Schnittpunkt: Dreieck

- Ebene **E** berechnen durch Punkte **V1**, **V2** und **V3**
- Schnittpunkt **S** zwischen Gerade und Ebene bestimmen
- Entscheiden ob **S** im Dreieck **V1 V2 V3** liegt

- Oder: Möller Trumbore Algorithmus

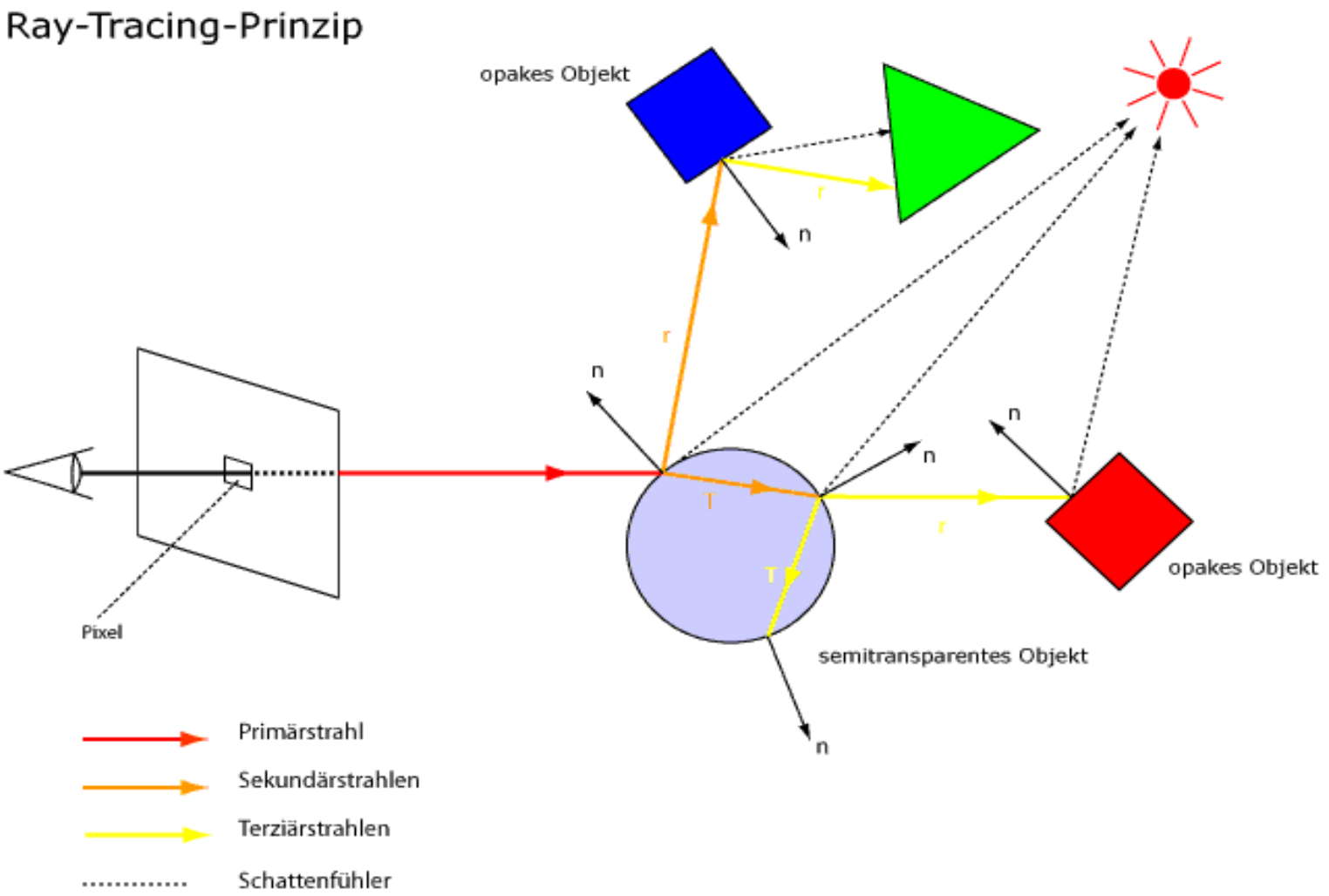


Schnittpunkt: Dreieck

```
#define EPSILON 0.000001

int triangle_intersection( const Vec3   V1,  // Triangle vertices
                           const Vec3   V2,
                           const Vec3   V3,
                           const Vec3    O,  //Ray origin
                           const Vec3    D,  //Ray direction
                                 float* out )
{
  Vec3 e1, e2;  //Edge1, Edge2
  Vec3 P, Q, T;
  float det, inv_det, u, v;
  float t;

  //Find vectors for two edges sharing V1
  SUB(e1, V2, V1);
  SUB(e2, V3, V1);
  //Begin calculating determinant - also used to calculate u paramet
  CROSS(P, D, e2);
  //if determinant is near zero, ray lies in plane of triangle
  det = DOT(e1, P);
  //NOT CULLING
  if(det > -EPSILON && det < EPSILON) return 0;
  inv_det = 1.f / det;

  //calculate distance from V1 to ray origin
  SUB(T, O, V1);

  //Calculate u parameter and test bound
  u = DOT(T, P) * inv_det;
  //The intersection lies outside of the triangle
  if(u < 0.f || u > 1.f) return 0;

  //Prepare to test v parameter
  CROSS(Q, T, e1);

  //Calculate V parameter and test bound
  v = DOT(D, Q) * inv_det;
  //The intersection lies outside of the triangle
  if(v < 0.f || u + v  > 1.f) return 0;

  t = DOT(e2, Q) * inv_det;

  if(t > EPSILON) { //ray intersection
    *out = t;
    return 1;
  }

  // No hit, no win
  return 0;
}
```

Pseudocode from: https://en.wikipedia.org/wiki/Möller-Trumbore_intersection_algorithm

Rekursives RayTracing



Rekursives RayTracing

- Schnittpunkt gefunden und Objekt **spiegelnd** bzw. **transparent**
- Physikalisch korrekter Lichttransport z.B.: **Reflexion** und **Brechung**

- Rekursion wird beendet wenn :

  - Kein schnittpunkt gefunden wurde

  - die Maximale Rekursionstiefe erreicht wurde

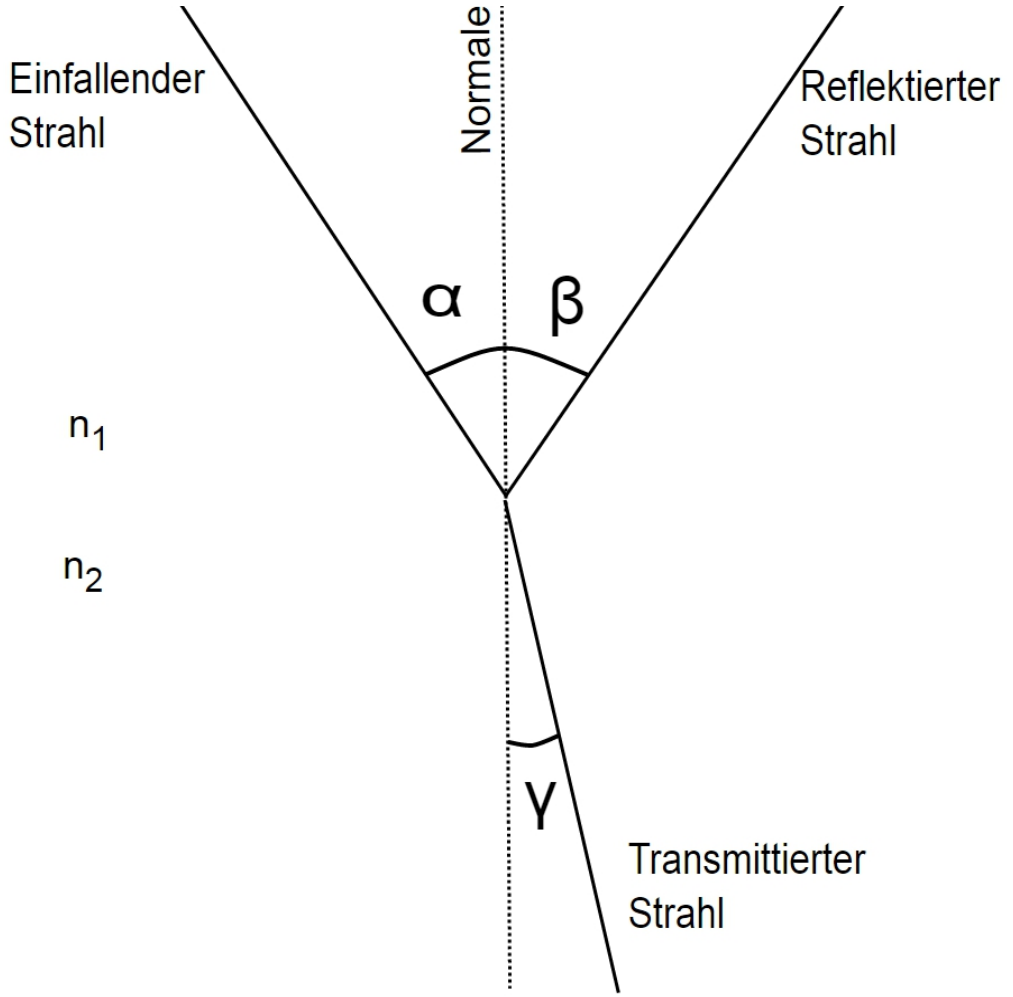
Rekursives RayTracing

```
Funktion FarbeFürStrahl(Strahl s) {
      Schnittpunkt sp = BerechneSchnittpunkt(s)
      Farbe farbe = BerechneFarbe()
      Wenn (Objekt spigelnd oder transparent) dann {
            NeuerStrahl strahlNeu = BerechneNeuenStrahl() //Reflektion
            Farbe farbe2 = FarbeFürStrahl(strahlNeu)
            farbe = MixFarben(farbe , farbe2) //Mischen der Farben
      }
      Return farbe
}
Für (jedes Pixel (sample) in der Bildebene) { //Für jedes Pixel
      Strahl  strahl = StrahlDurchPixel() //Konstruiere Strahl
      Farbe farbeFürPixel = FarbeFürStrahl(Strahl)
}
```

Reflektion

Reflektion

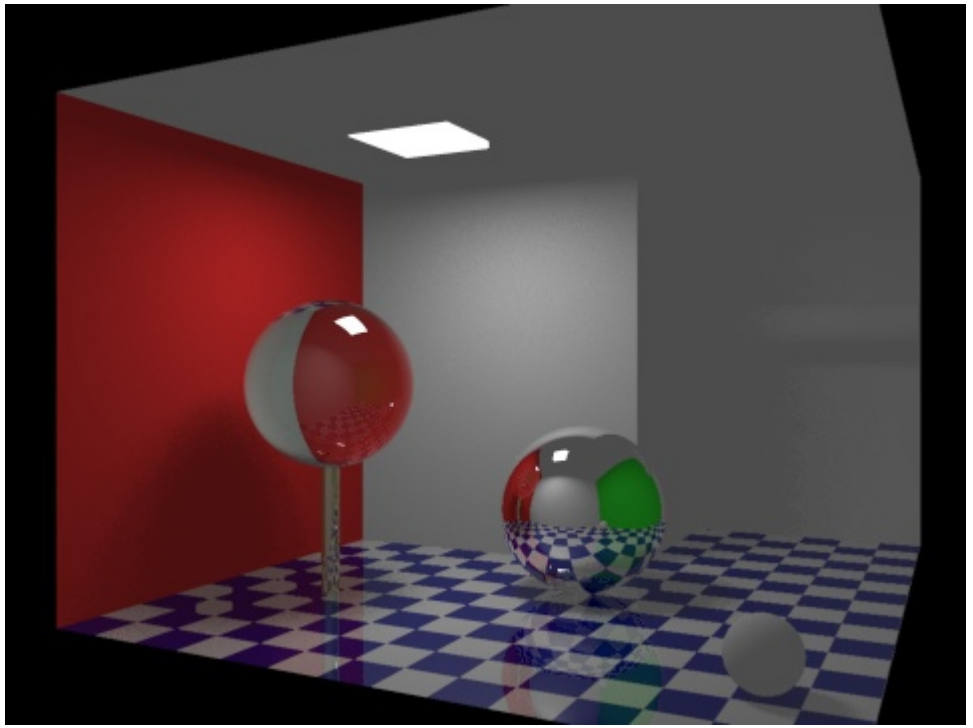
- R = V - 2 * (V · N) * N
- R = reflektierter Strahl
- N = Normale
- V = einfallender Strahl



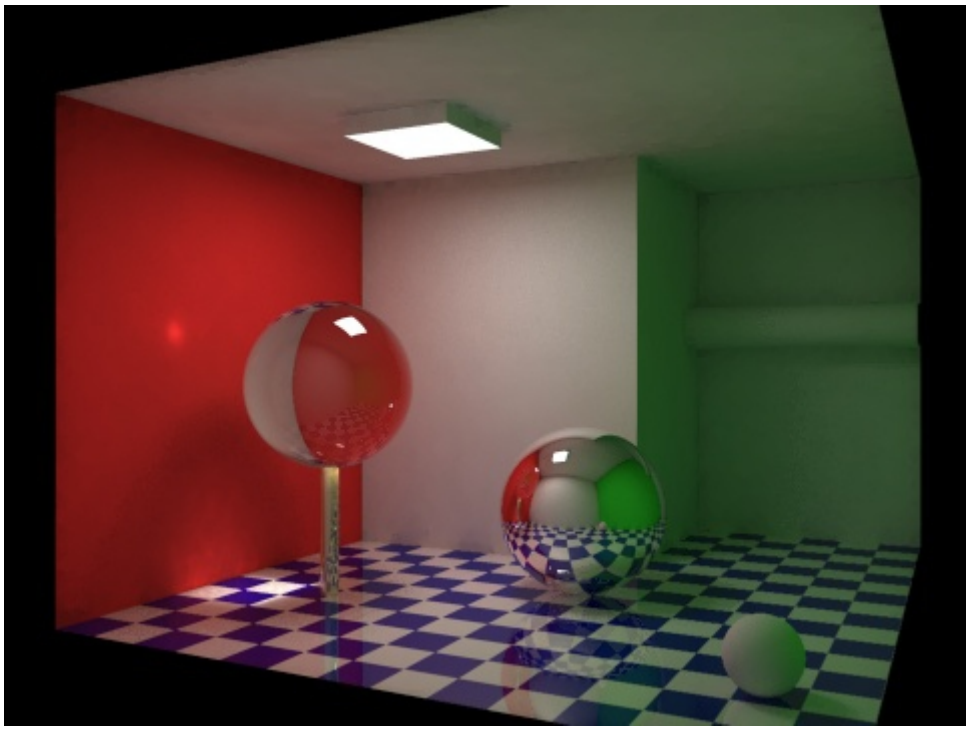
- n_1 = erstes Medium
- n_2 = zweites Medium

Beleuchtungsmodelle

- **Licht** und **Schatten** sind essentiell für ein physikalisch korrektes Bild
- Man unterscheidet **lokale** und **globale** Beleuchtung



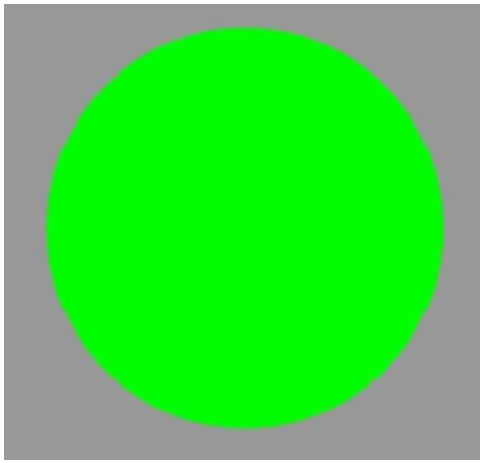
Lokale Beleuchtung

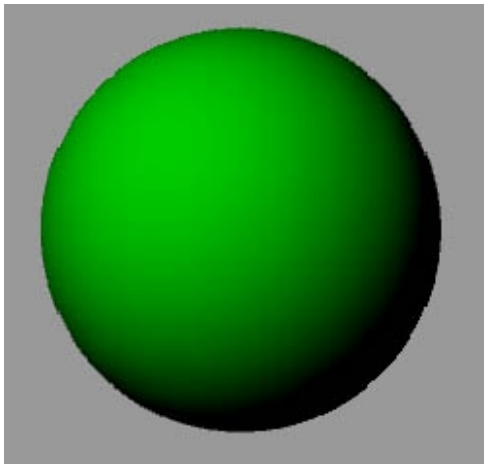


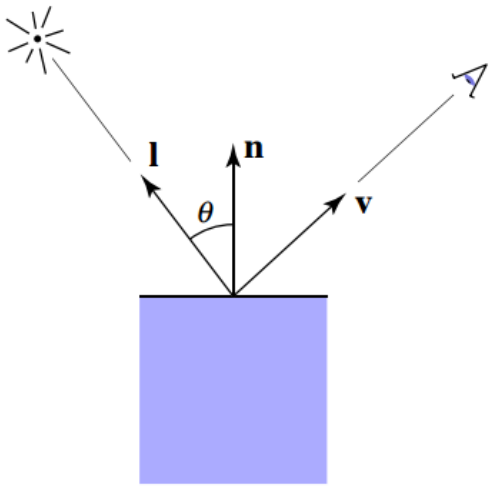
Globale Beleuchtung

Lokale Beleuchtung

- Lambertsches Gesetz
- V = Einfallender Strahl
- I = Lichtquelle
- L_d = Diffus reflektiertes Licht







illumination from source

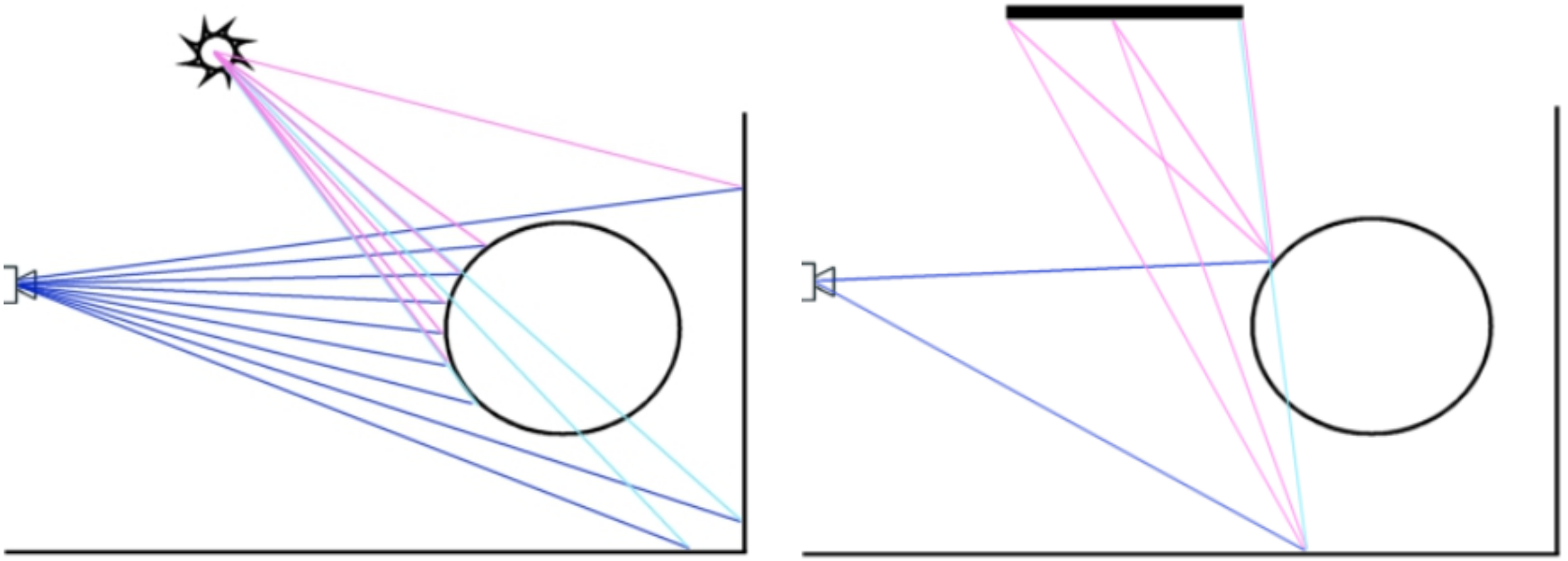
$$L_d = k_d \, I \max(0, \mathbf{n} \cdot \mathbf{l})$$

diffuse coefficient

diffusely reflected light

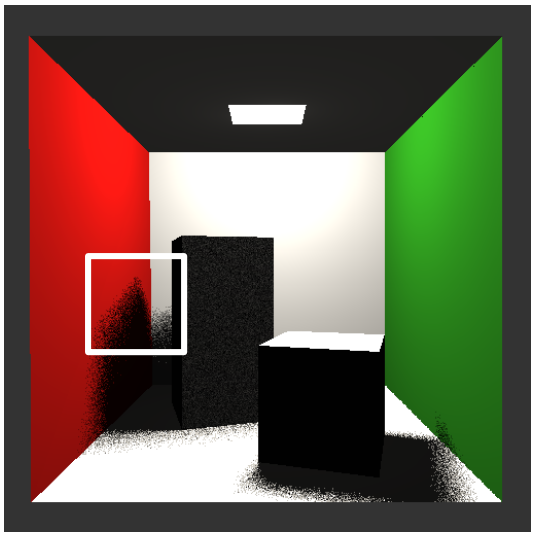
Lokale Beleuchtung

- Lichter können auch eine Fläche besitzen
- Sampling der Lichtquelle ergibt **weiche** Schatten
- Zu wenige samples führen zu rauschen

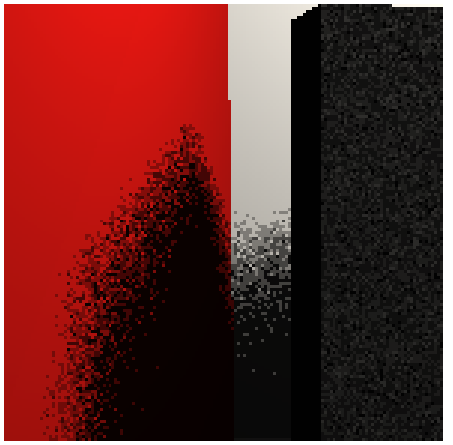


Lokale Beleuchtung

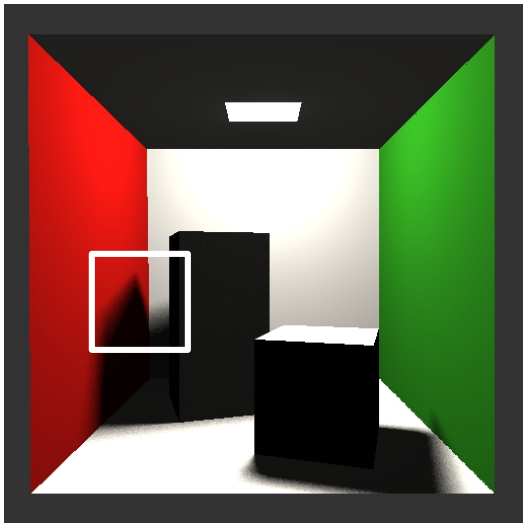
keine Schatten



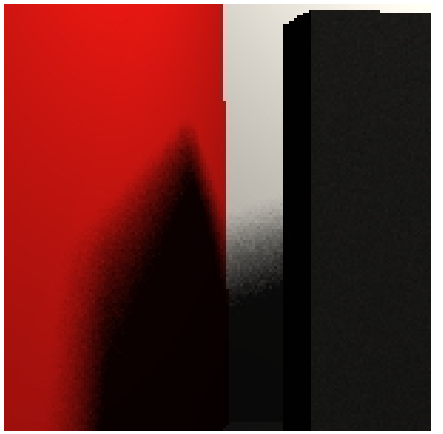
Weiche Schatten, 2 Samples



harte Schatten

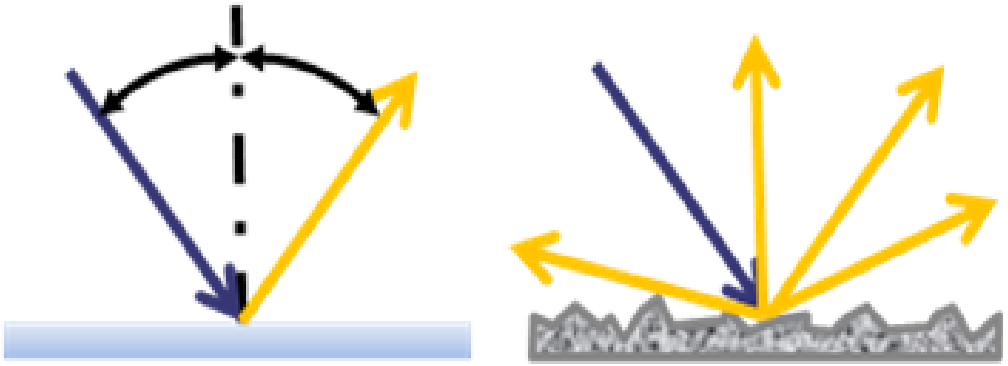


Weiche Schatten, 100 Samples



Globale Beleuchtung

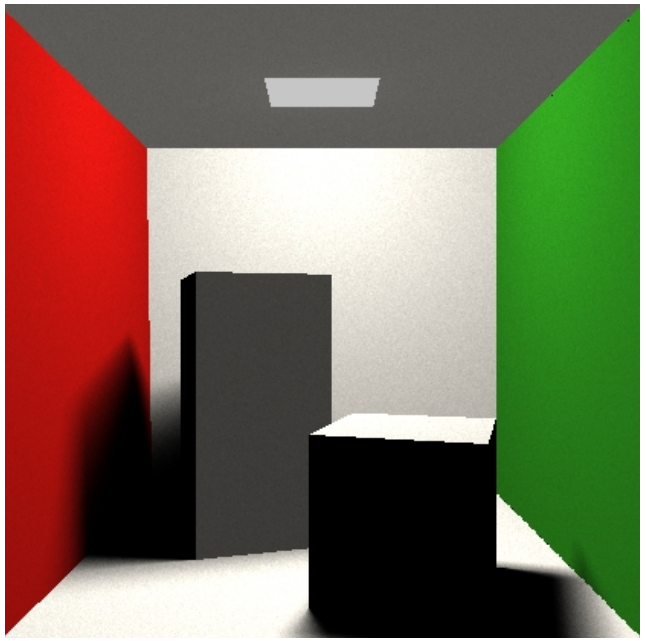
- Ermöglicht photorealistische **diffuse Reflektionen** und **Kaustiken**



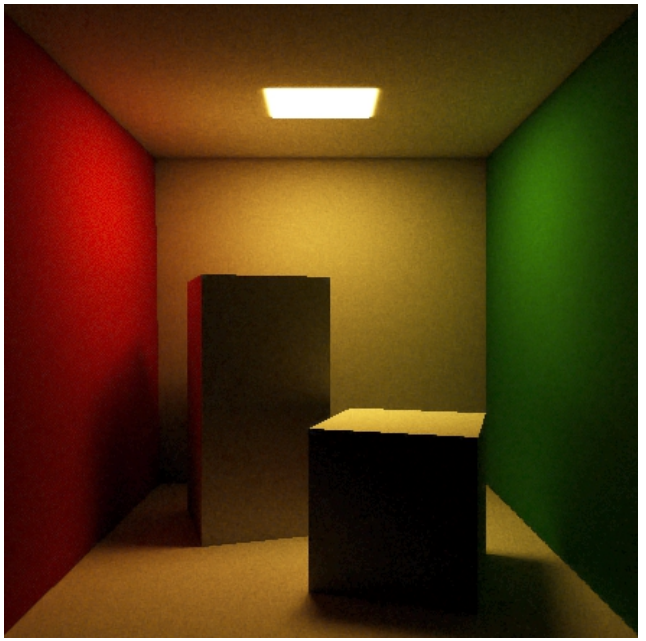
- **Photonen Mapping** (Strahlverfolgung von der Lichtquelle aus)
- **"PathTracing"** (Strahlverfolgung von der Kamera aus; hohe Rechenleistung notwendig da **MonteCarlo** Versuch)

PathTracing

- Beim Aufprallen an einer Oberfläche werden viele zufällige Strahlen ausgesendet (Samples) → Mittelwert
- Die Anzahl der "Abpraller" (Bounces) muss begrenzt werden → max. Rekursionstiefe



RayTracing (10s)



PathTracing (3000s)

$$L_o = \int \frac{c}{\pi} L_i \cos\theta, d\omega$$

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}$$

PathTracing

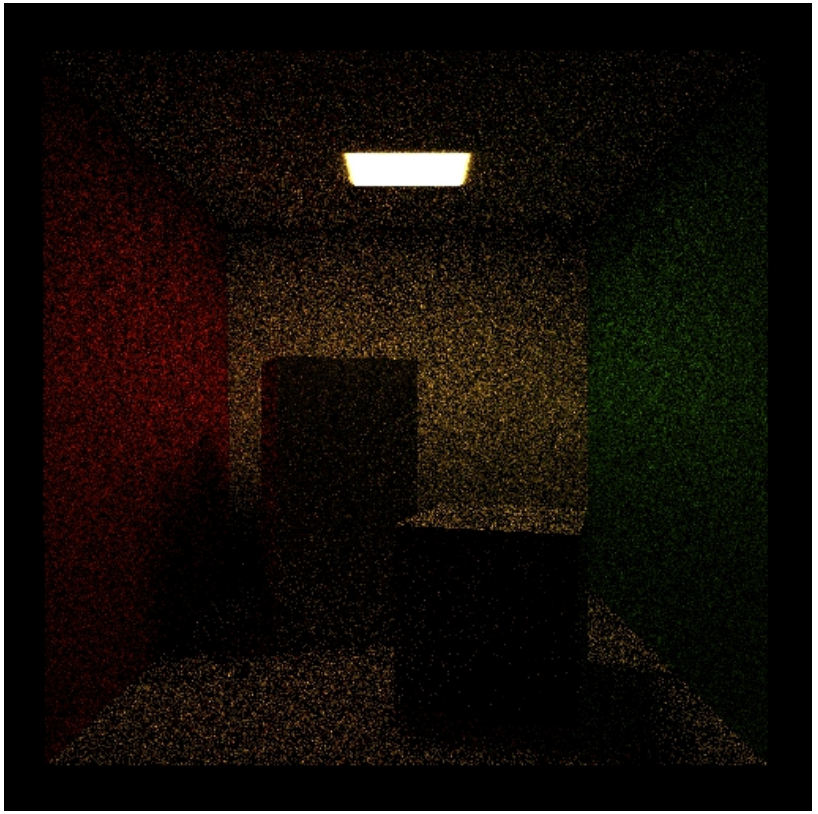
```
Funktion farbeFürPixel(Pixel pixel, int samples) {
      Farbe f = schwarz;
      Pixel pixel = ein Pixel in der Bildebene;

      // Mittelwert bilden
      Für (int I = 0; I < samples; i++){
            Strahl ray = Kamera.generiereStrahl(pixel)
             f = f + pathTrace(ray, 0)
      }

      Return f / samples
}
```

PathTracing

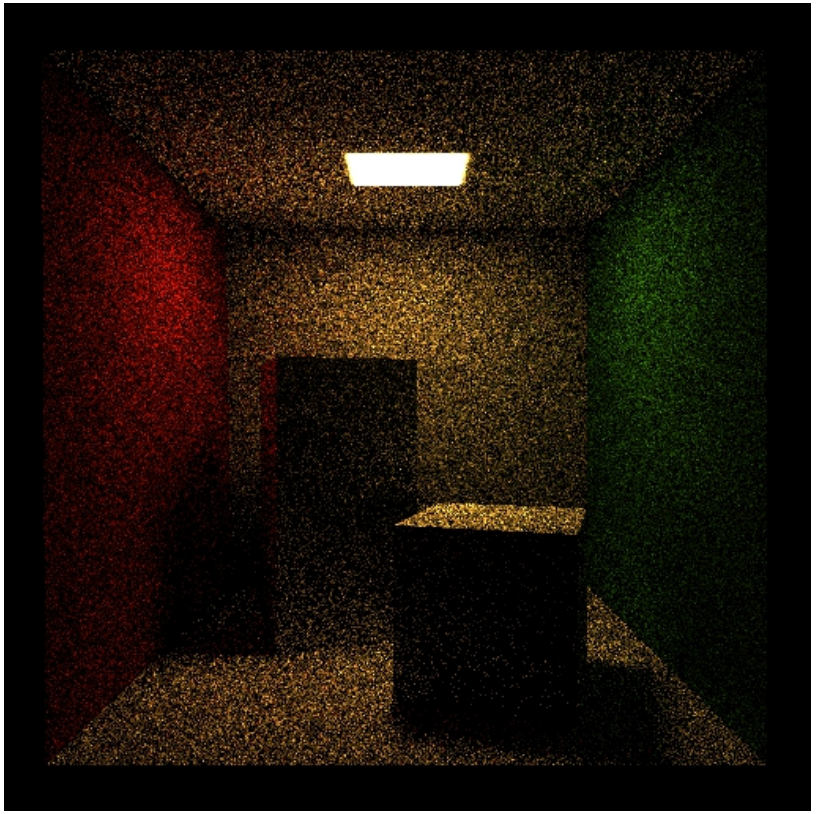
```
Funktion pathTrace(Strahl s, int tiefe) {
    Wenn (Abbruchkriterium erreicht) dann { return schwarz }
    Wenn (Objekt == Lichtquelle) dann { return Objekt.LichtEmission }

    Farbe alteFarbe = schwarz
    Strahl neuerStrahl = zufälligerStrahl()

    Double cos = DOT(neuerStrahl,  Objekt.normale) //Nach Lambert
    Vector BRDF = Objekt.diffuseFarbe * cos * 2

    Farbe neueFarbe = pathTrace(neuerStrahl, tiefe + 1) //Rekursion
    alteFarbe = alteFarbe + BRDF * neueFarbe
    Return alteFarbe;
}
```

Sampling – Cosinus Hemisphere

- Die Qualität der zufälligen Vektoren spielt eine wichtige Rolle



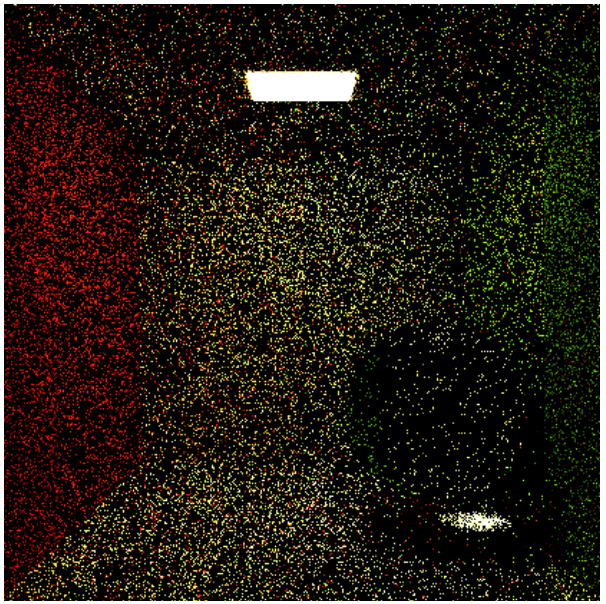
Uniform (100 Samples)



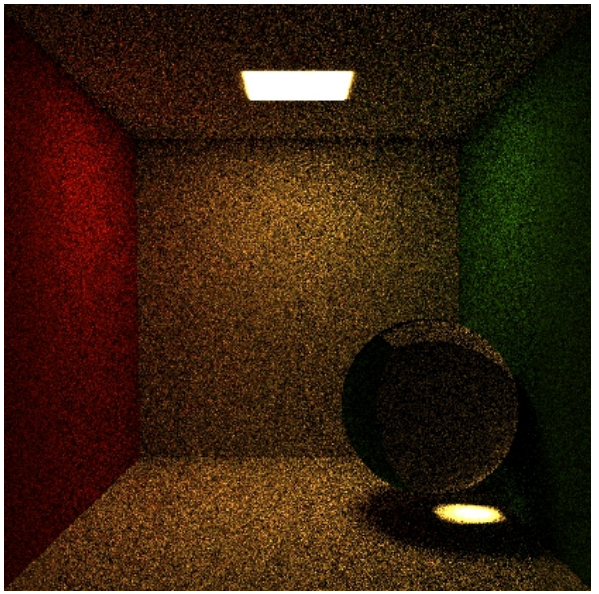
Cosinus (100 Samples)

PathTracing

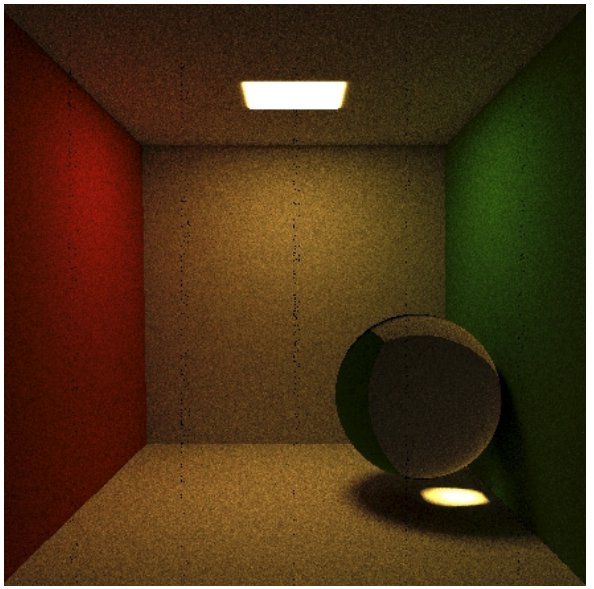
- Anzahl der Samples muss vervierfacht werden um die Bildqualität zu verdoppeln



10 Samples



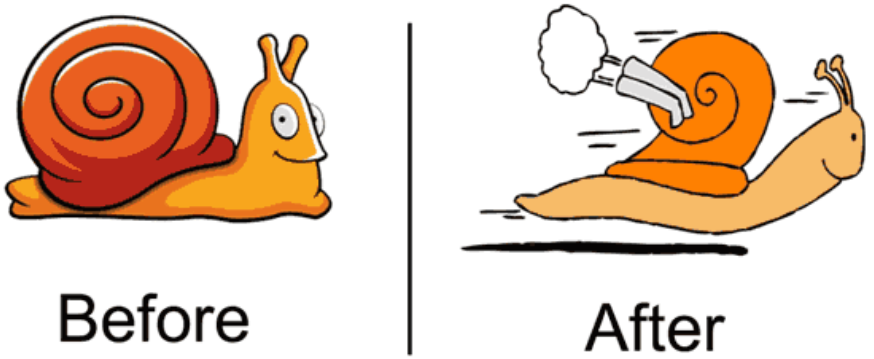
230 Samples



2000 Samples

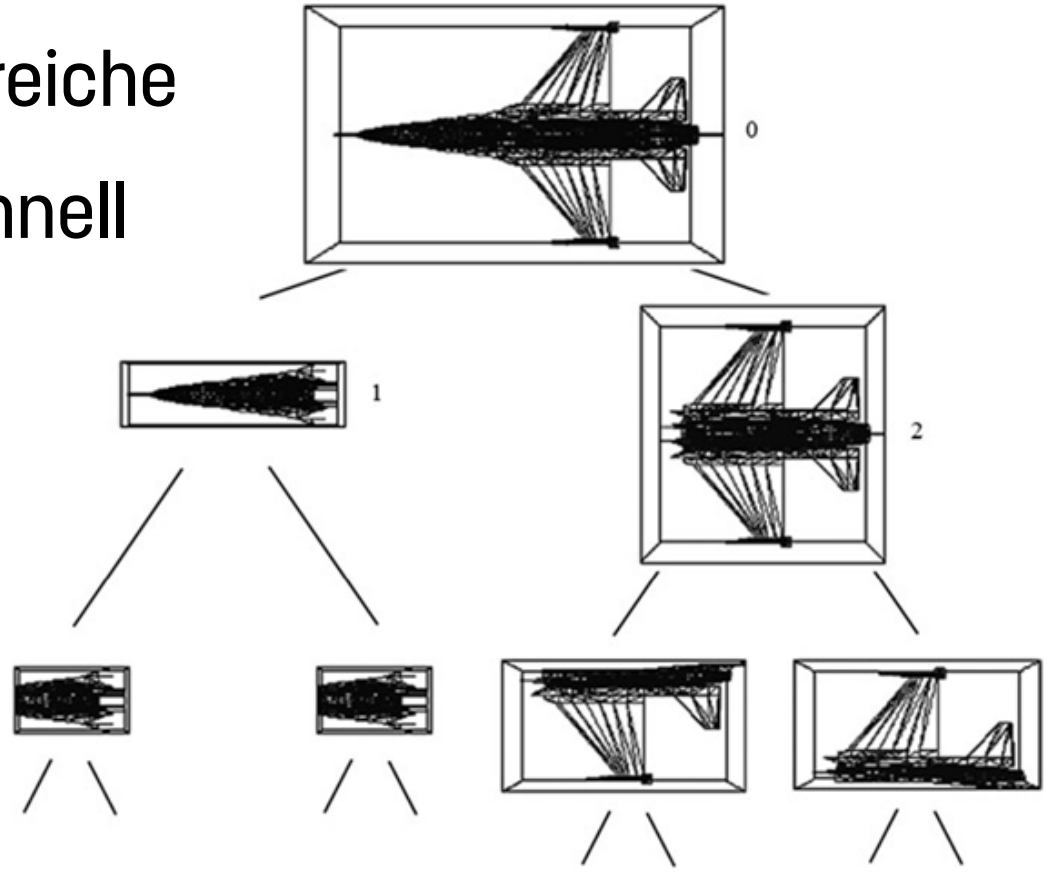
Beschleunigungsverfahren

- Berechnen der Schnittpunkte nimmt am meisten Zeit in Anspruch (ca. 95% der gesamten Rechenzeit) → Diesen Anteil reduzieren
- "Bounding Volumes" (Hüllkörper) für einzelne Objekte bzw. Abschnitte der Szene
- "Back Face Culling" für sämtliche Dreiecke von Konvexen Objekten



Bounding Volumes

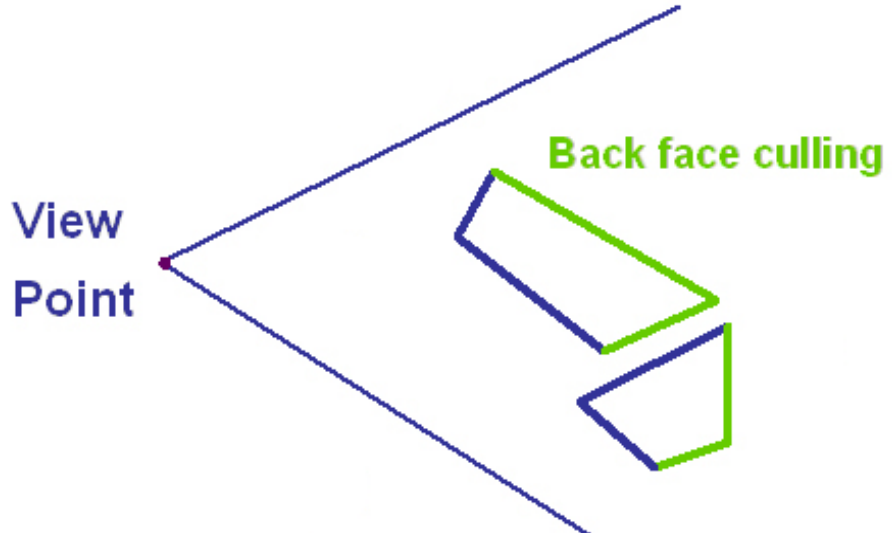
- Berechnen einer **“AABB”** um das eigentliche Objekt herum
- Einteilen in kleinere Bereiche
- **AABB** Schnittpunkt schnell



Back Face Culling

- Bei Konvexen Körpern sind die Normalen der Dreiecke vom Mittelpunkt weggerichtet

```
Wenn ( DOT(Strahl, Normal) <= 0) dann {
        berechneSchnittpunkt()
}
```

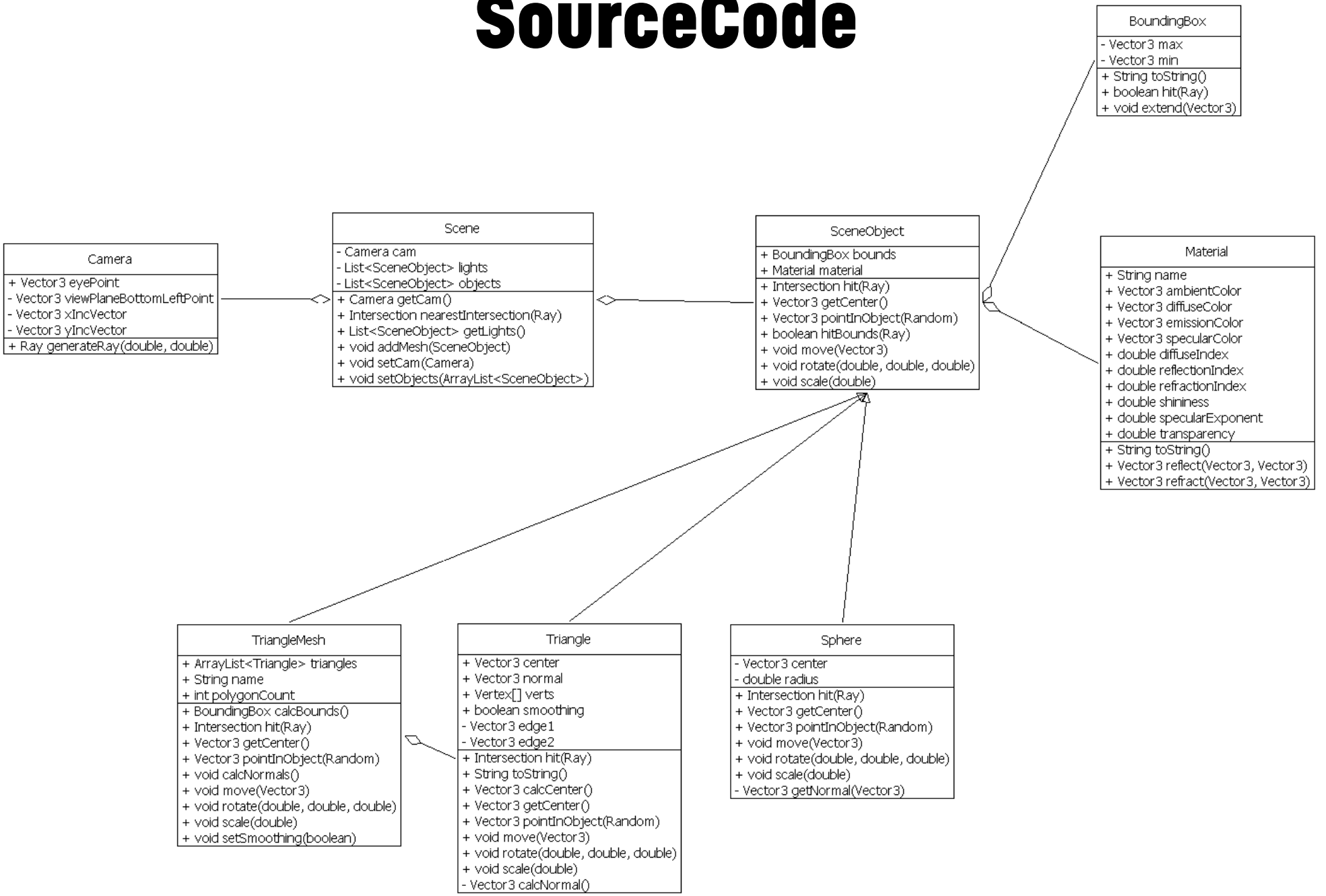


Implementierung

- **JAVA** nicht die schnellste Sprache da in VM, dafür einfach zu lesen, übersichtlich und objektorientiert
- **JAVA**, d.h. der RayTracer / PathTracer läuft auch auf dem Mobiltelefon (WORE – Write Once Run Everywhere)
- Multithreading support → Teilt das Bild in n Streifen auf (n = Anzahl der Rechenkerne)

- Objekt Orientiertes Design

SourceCode


BoundingBox
- Vector3 max
- Vector3 min
+ String toString()
+ boolean hit(Ray)
+ void extend(Vector3)
Camera
+ Vector3 eyePoint
- Vector3 viewPlaneBottomLeftPoint
- Vector3 xIncVector
- Vector3 yIncVector
+ Ray generateRay(double, double)
Scene
- Camera cam
- List<SceneObject> lights
- List<SceneObject> objects
+ Camera getCam()
+ Intersection nearestIntersection(Ray)
+ List<SceneObject> getLights()
+ void addMesh(SceneObject)
+ void setCam(Camera)
+ void setObjects(ArrayList<SceneObject>)
SceneObject
+ BoundingBox bounds
+ Material material
+ Intersection hit(Ray)
+ Vector3 getCenter()
+ Vector3 pointInObject(Random)
+ boolean hitBounds(Ray)
+ void move(Vector3)
+ void rotate(double, double, double)
+ void scale(double)
Material
+ String name
+ Vector3 ambientColor
+ Vector3 diffuseColor
+ Vector3 emissionColor
+ Vector3 specularColor
+ double diffuseIndex
+ double reflectionIndex
+ double refractionIndex
+ double shininess
+ double specularExponent
+ double transparency
+ String toString()
+ Vector3 reflect(Vector3, Vector3)
+ Vector3 refract(Vector3, Vector3)
TriangleMesh
+ ArrayList<Triangle> triangles
+ String name
+ int polygonCount
+ BoundingBox calcBounds()
+ Intersection hit(Ray)
+ Vector3 getCenter()
+ Vector3 pointInObject(Random)
+ void calcNormals()
+ void move(Vector3)
+ void rotate(double, double, double)
+ void scale(double)
+ void setSmoothing(boolean)
Triangle
+ Vector3 center
+ Vector3 normal
+ Vertex[] verts
+ boolean smoothing
- Vector3 edge1
- Vector3 edge2
+ Intersection hit(Ray)
+ String toString()
+ Vector3 calcCenter()
+ Vector3 getCenter()
+ Vector3 pointInObject(Random)
+ void move(Vector3)
+ void rotate(double, double, double)
+ void scale(double)
- Vector3 calcNormal()
Sphere
- Vector3 center
- double radius
+ Intersection hit(Ray)
+ Vector3 getCenter()
+ Vector3 pointInObject(Random)
+ void move(Vector3)
+ void rotate(double, double, double)
+ void scale(double)
- Vector3 getNormal(Vector3)

SourceCode

SourceCode verfügbar auf Github :

https://www.github.com/juliusmh

Quellen

- https://www.ice.rwth-aachen.de/fileadmin/user_upload/forschungsprojekte/tools/GRACE/raytracing.gif
- https://de.wikipedia.org/wiki/Raytracing#/media/File:Raytracing.svg
- http://www.cs.unc.edu/~bn/comp770/images/myScene.bmp
- https://upload.wikimedia.org/wikipedia/commons/e/ec/Glasses_800_edit.png
- https://cs.stanford.edu/people/eroberts/courses/soco/projects/1997-98/ray-tracing/implementation.html
- http://www.dma.ufg.ac.at/assets/8653/intern/raytracing.gif
- http://www.zbs-ilmenau.de/pdf/10V_3D_Rendering_2_Druck.pdf
- http://asawicki.info/news_1301_reflect_and_refract_functions.html
- https://de.wikipedia.org/wiki/Antialiasing_(Computergrafik)#/media/File:Checkerboard_Zone_Plate_Ordered4.png
- http://www.flipcode.com/archives/Raytracing_Topics_Techniques-Part_5_Soft_Shadows.shtml
- http://www.3dtutorialzone.com/making_materials/11.jpg
- http://www.puchner.org/Fotografie/technik/physik/reflexion_01.png
- https://root.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/GS/GS116AT3.DOC_AFrame_74.gif

Quellen

- http://web4.cs.ucl.ac.uk/teaching/4074/archive/2010/Slides/JK2010/03_path%20tracing.pdf
- http://what-when-how.com/wp-content/uploads/2012/07/tmpf9b3187_thumb2222.png
- https://root.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/GS/GS116AT3.DOC_AFrame_74.gif
- http://volkansahin.net/wp-content/uploads/2014/03/backface-culling.png
- http://www.flipcode.com/
- https://github.com/