# Fundamentals of Computer Graphics and Image Processing
# Raycasting (08)

doc. RNDr. Martin Madaras, PhD.
martin.madaras@fmph.uniba.sk

# Computer Graphics

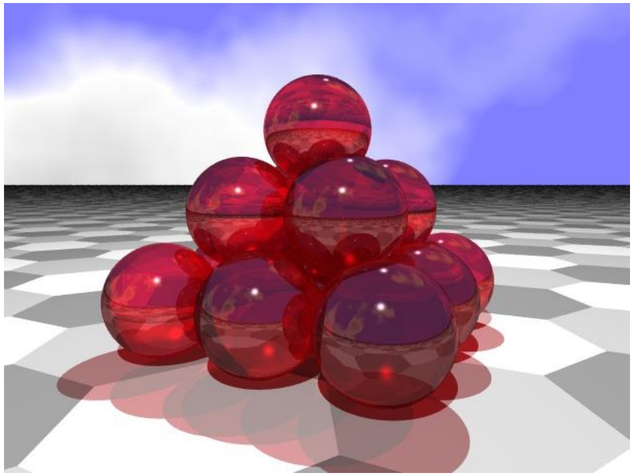
- Image processing
  - Representing and manipulation of 2D images
- Modeling
  - Representing and manipulation of 2D and 3D objects
- Animation
  - Simulating changes over time
- **Rendering**
  - Constructing images from virtual models

# How the lectures should look like #1

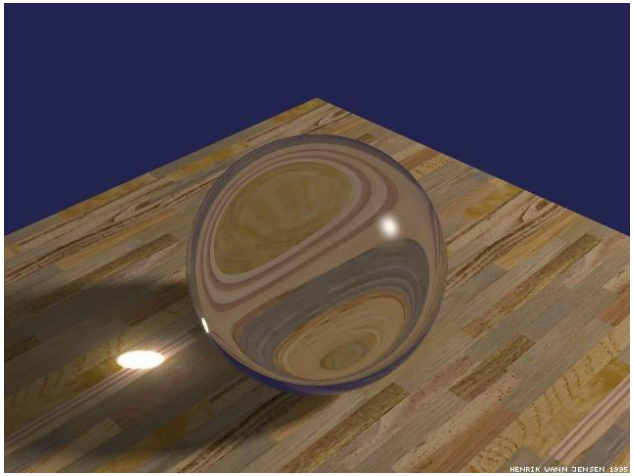
- Ask questions, please!!!
- Be communicative
- More active you are, the better for you!

# Towards Photorealism

- light refraction
- mutual object reflection
- caustics
- color bleeding
- (soft) shadows

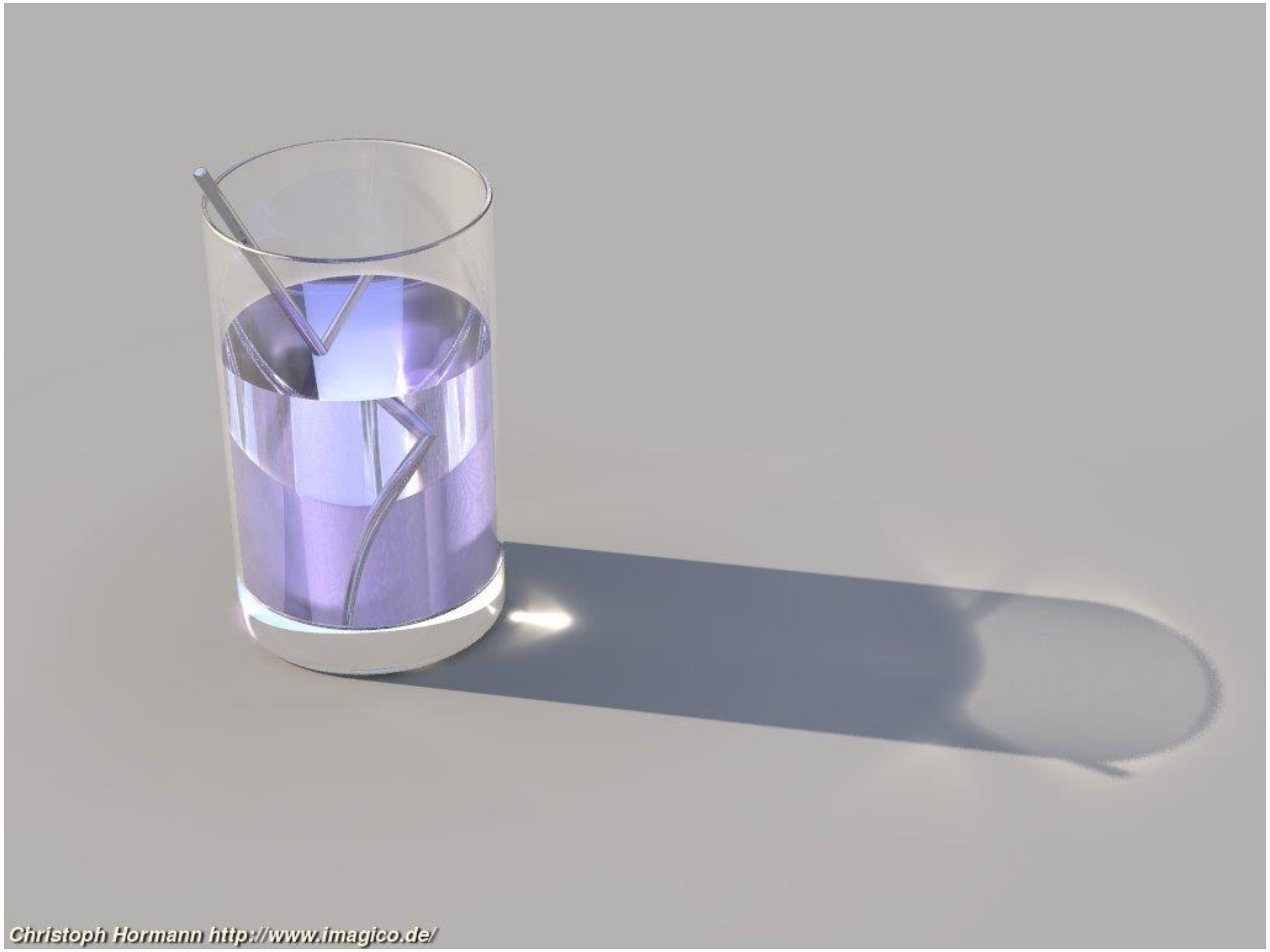


http://math.hws.edu/eck

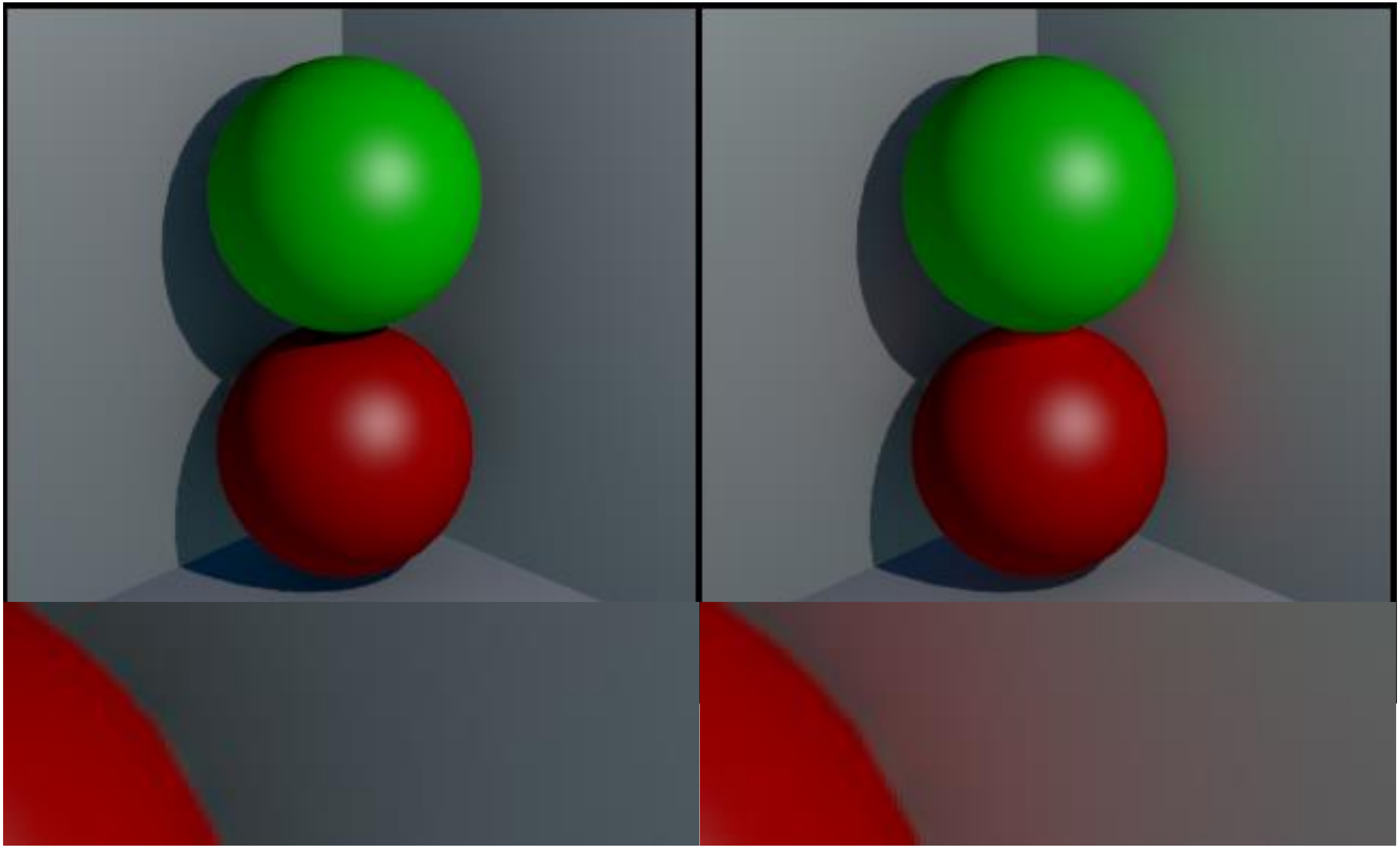



http://graphics.ucsd.edu/~henrik/

# Refraction & Caustics



# Global Illumination

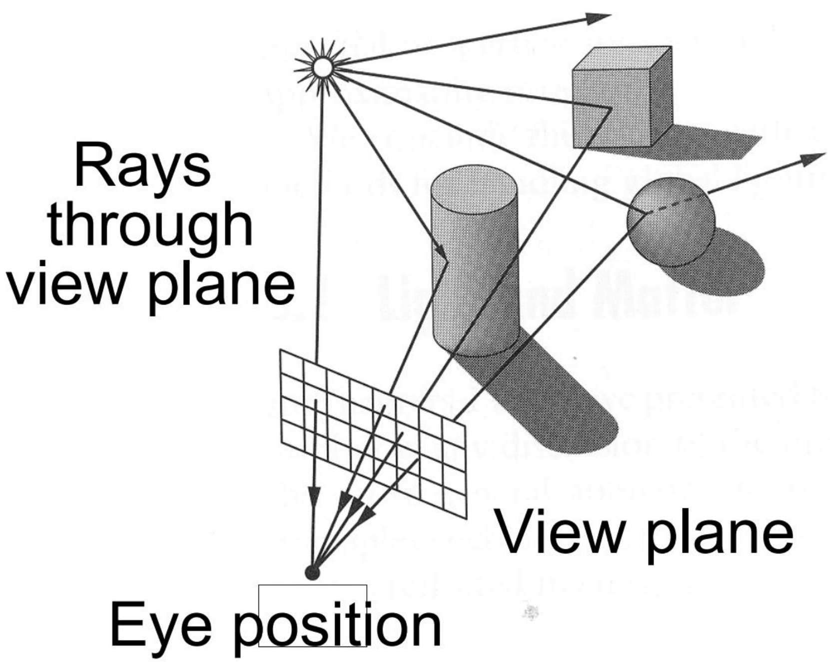


# Introduction

Raycasting

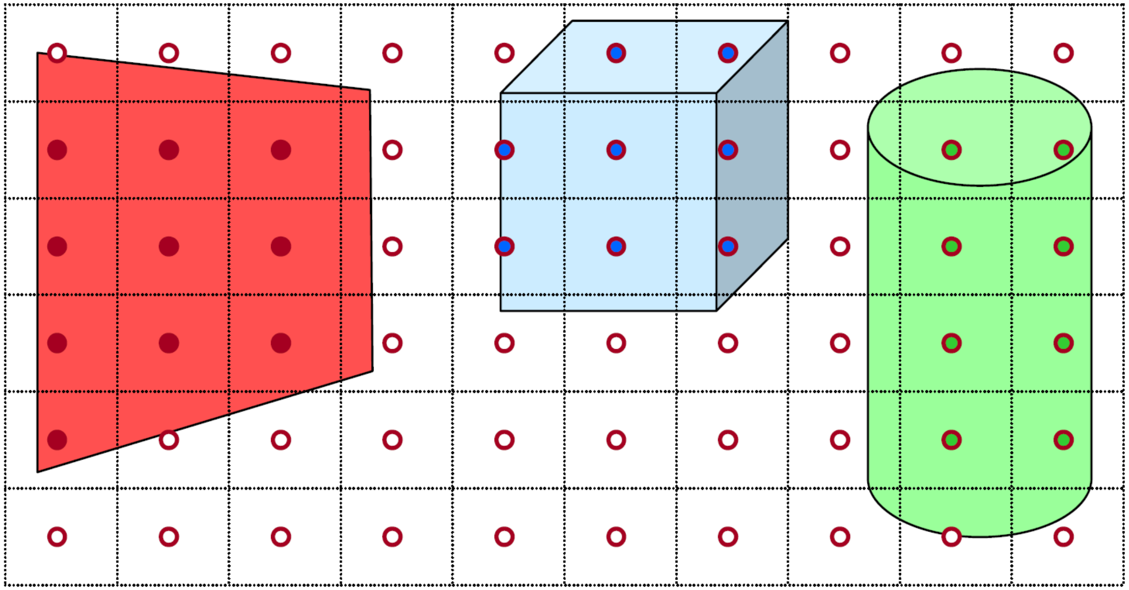
# 3D Rendering

- Color of each pixel on the view plane depends on the radiance emanating from visible surfaces



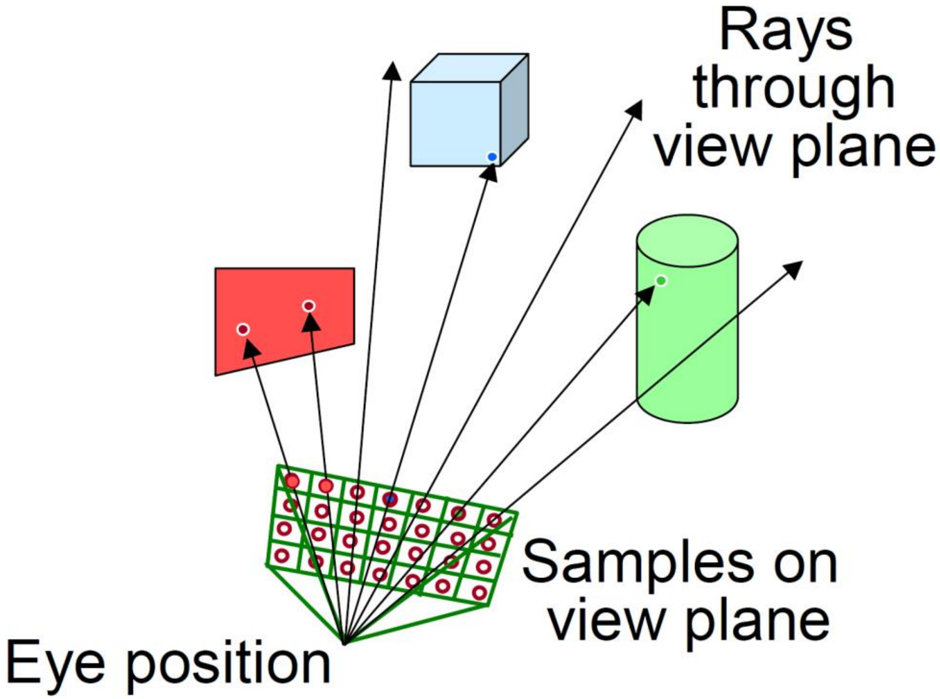
# Ray Casting

- For each sample ...
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color sample based on surface radiance



# Ray Casting

- For each sample ...
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color sample based on surface radiance



# Ray Casting

- Simple implementation

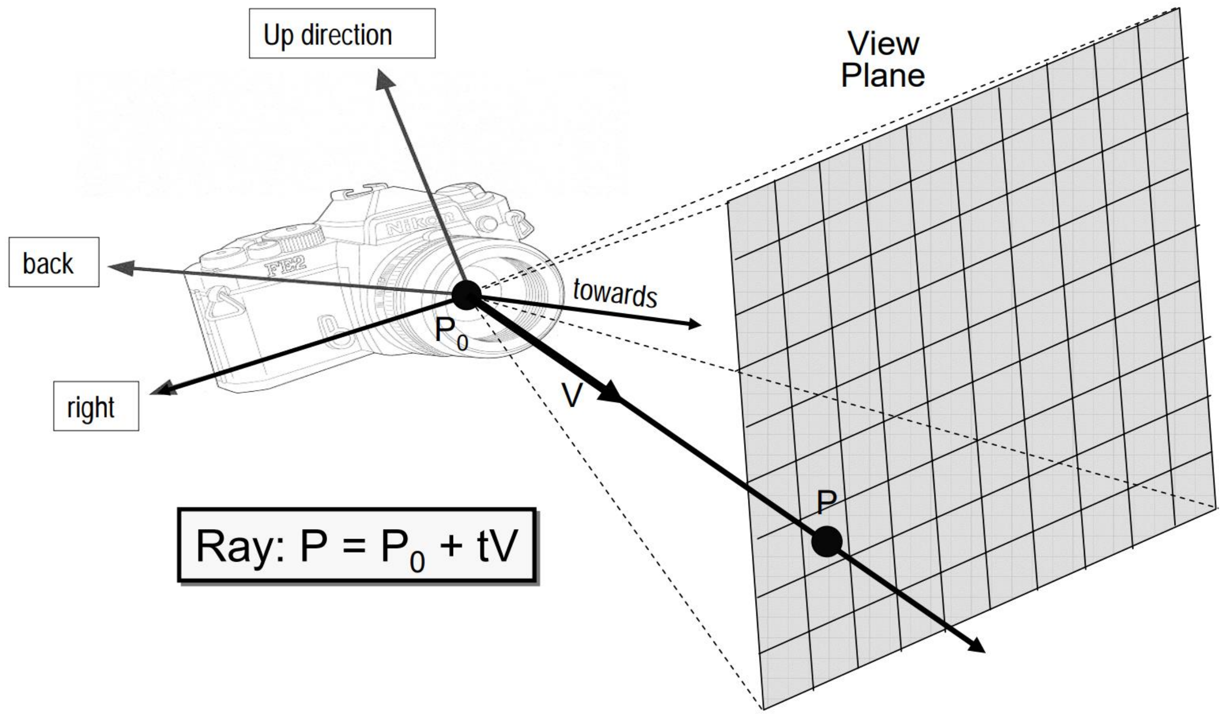
```
Image RayCast(Camera camera, Scene scene, int width, int height) {
  Image image = new Image(width, height);
  for(int i=0; i<width; i++) {
    for(int j=0; j<height; j++) {
      Ray ray = ConstructRayThroughPixel(camera, i, j);
      Intersection hit = FindIntersection(ray, scene);
      image[i][j]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

# Ray Casting

- Simple implementation

```
Image RayCast(Camera camera, 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(camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(scene, ray, hit);
        }
    }
    return image;
}
```

# Ray Construction

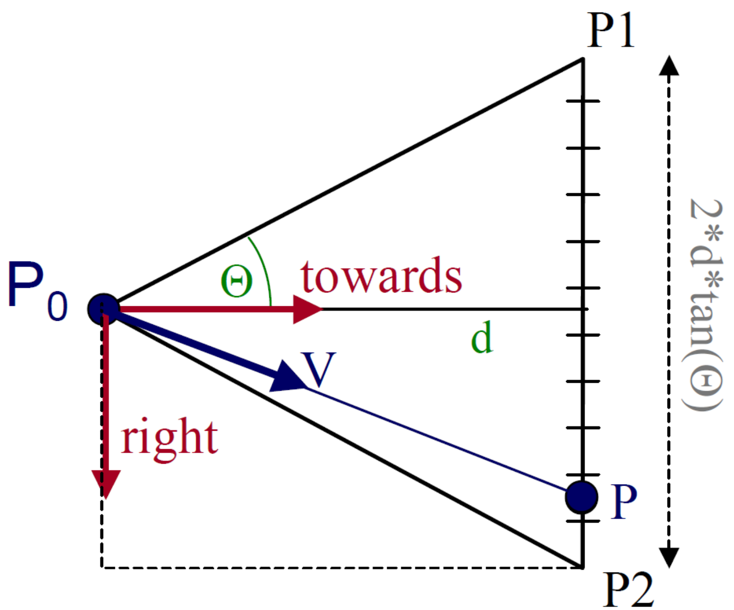


# Ray Construction

- 2D example

$\Theta$ = frustum half-angle
d = distance to view plane

right = towards x up




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

$P = P1 + (i/width + 0.5) * (P2 - P1)$
$\quad = P1 + (i/width + 0.5) * 2*d*\tan(\Theta)*right$
$V = (P - P_0) / ||P - P_0||$

$\boxed{\text{Ray: } P = P_0 + tV}$

# Ray Casting

- Simple implementation

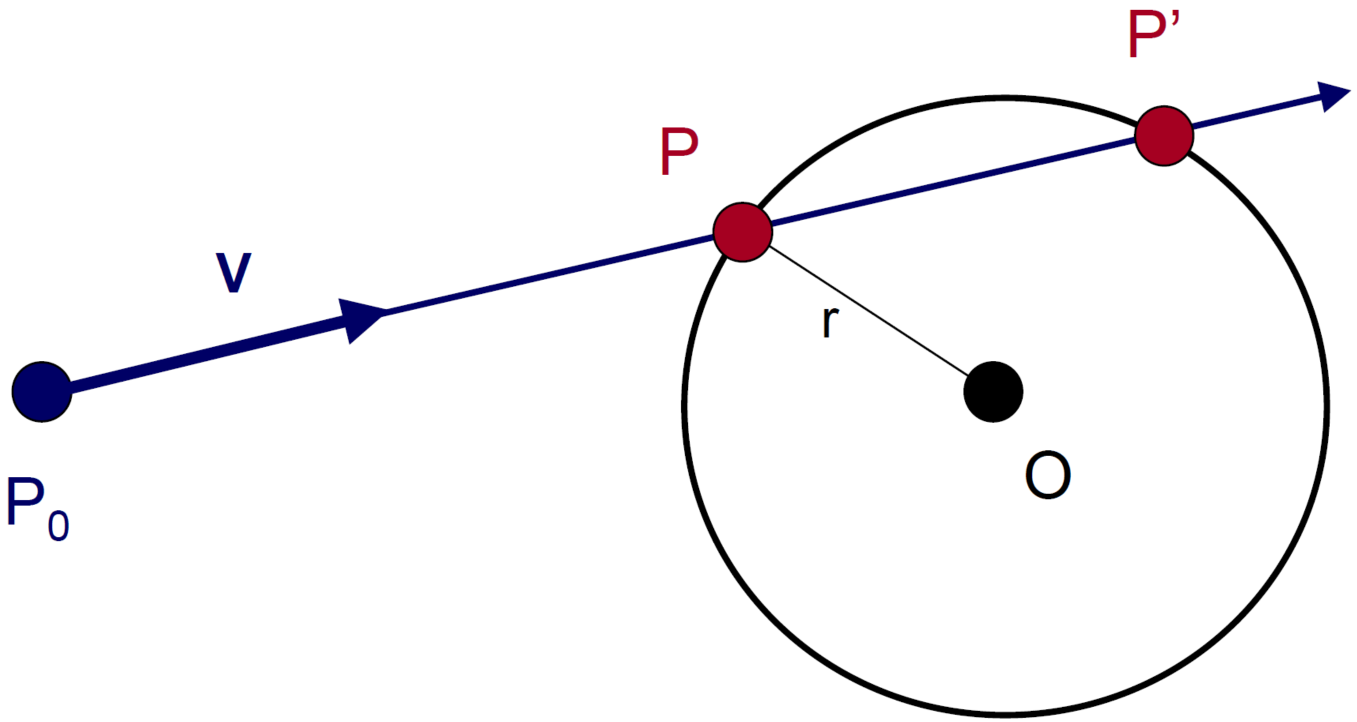
```
Image RayCast(Camera camera, Scene scene, int width, int height) {
    Image image = new Image(width, height);
    for(int i=0; i<width; i++) {
        for(int j=0; j<height; j++) {
            Ray ray = ConstructRayThroughPixel(camera, i, j);
            Intersection hit = FindIntersection(ray,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

# Ray-Scene Intersection

- Intersections with geometric primitives
  - **Sphere**
  - Triangle
  - Groups of primitives (scene)
  - Acceleration Techniques
    - Bounding volume hierarchies
    - Spatial partitions
      - Uniform grids
      - Octrees
      - BSP trees

# Ray-Sphere Intersection

- Ray: $P = P_0 + tV$
- Sphere: $|P - O|^2 - r^2 = 0$



# Ray-Sphere Intersection I

- Ray: $P = P_0 + tV$
- Sphere: $|P - O|^2 - r^2 = 0$
  - Algebraic method:

Substituting for P, we get:

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

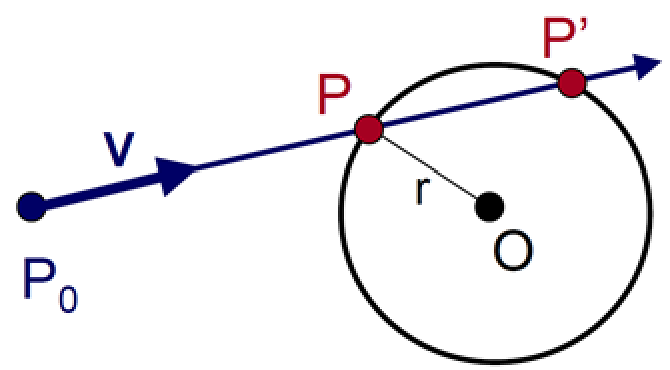
Solve quadratic equation:

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

where:

$$a = 1$$
$$b = 2\, V \bullet (P_0 - O)$$
$$c = |P_0 - O|^2 - r^2$$

$$P = P_0 + tV$$



# Ray-Sphere Intersection II

- Ray: $P = P_0 + tV$
- Sphere: $|P - O|^2 - r^2 = 0$
  - Geometric method:

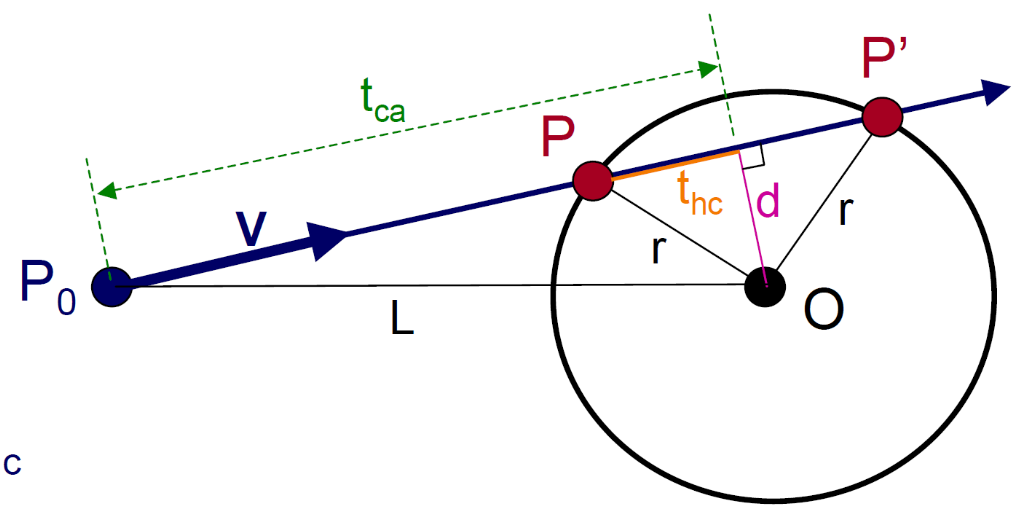
$L = O - P_0$

$t_{ca} = L \cdot V$
if ($t_{ca} < 0$) return 0

$d^2 = L \cdot L - t_{ca}^2$
if ($d^2 > r^2$) return 0

$t_{hc} = \mathrm{sqrt}(r^2 - d^2)$
$t = t_{ca} - t_{hc}$ and $t_{ca} + t_{hc}$

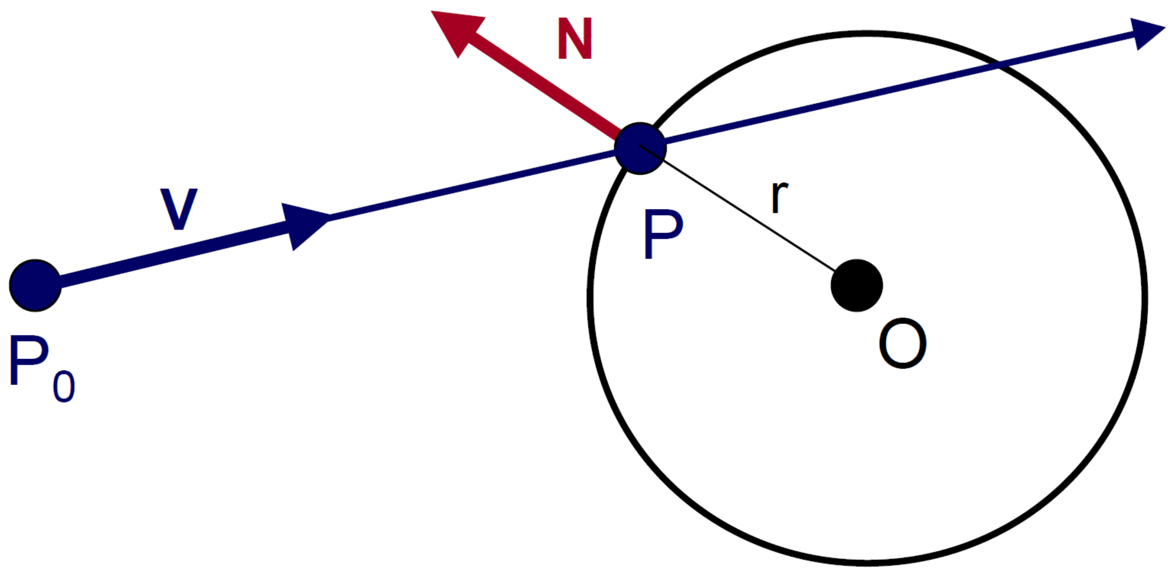
$P = P_0 + tV$



# Ray-Sphere Intersection

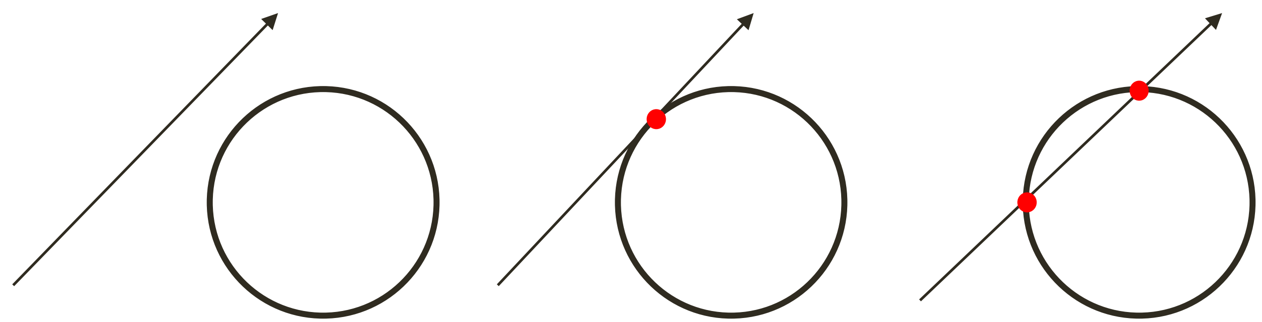
- We need normal vector at intersection for lighting calculations

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



# Ray-Sphere Intersection

- Multiple possible scenarios

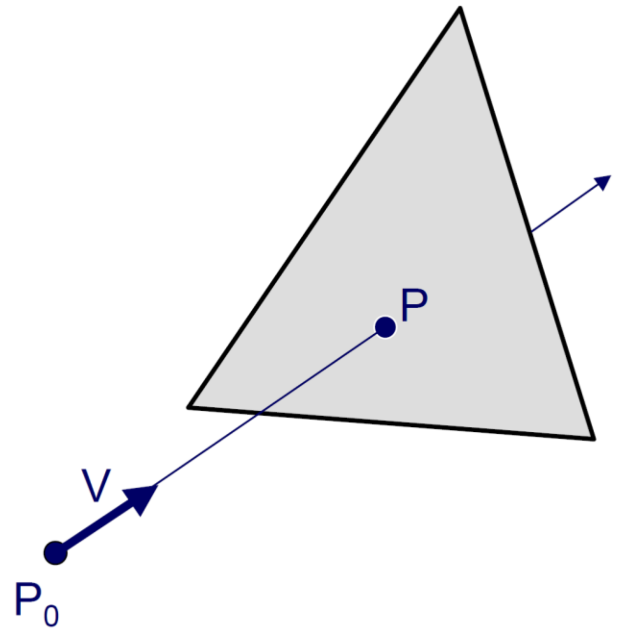


# Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - **Triangle**
  - Groups of primitives (scene)
  - Acceleration Techniques
    - Bounding volume hierarchies
    - Spatial partitions
      - Uniform grids
      - Octrees
      - BSP trees

# Ray-Triangle Intersection

- First, intersect ray with plane
- Then, check if the point is inside triangle



# Ray-Plane Intersection

- Ray: $P = P_0 + tV$
- Plane: $(P - L) \cdot N = 0$
  - Algebraic method:

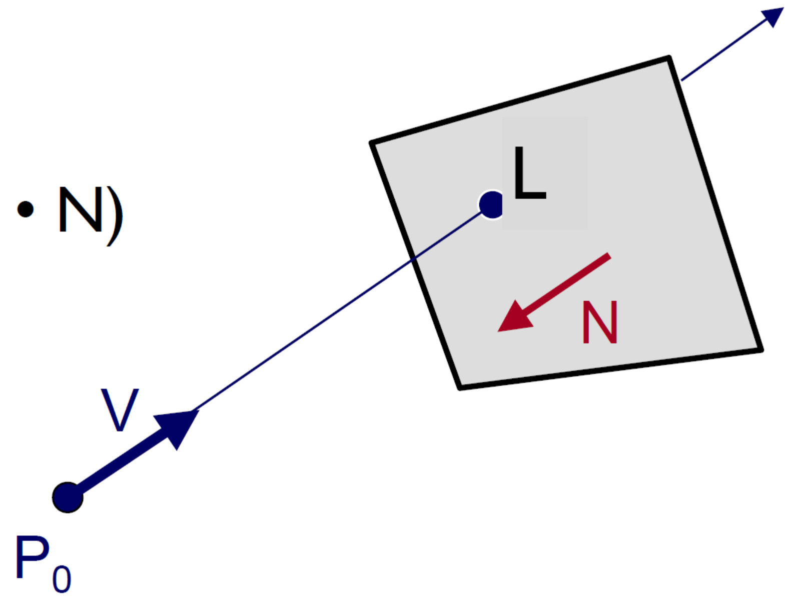
Substituting for P, we get:

$$(P_0 + tV - L) \cdot N = 0$$

Solution:

$$t = ((L - P_0) \cdot N) / (V \cdot N)$$

$$P = P_0 + tV$$



# Ray-Triangle Intersection

- Check if the point is inside triangle
  - Algebraic method:

For each side of triangle

$V_1 = T_1 - P_0$

$V_2 = T_2 - P_0$

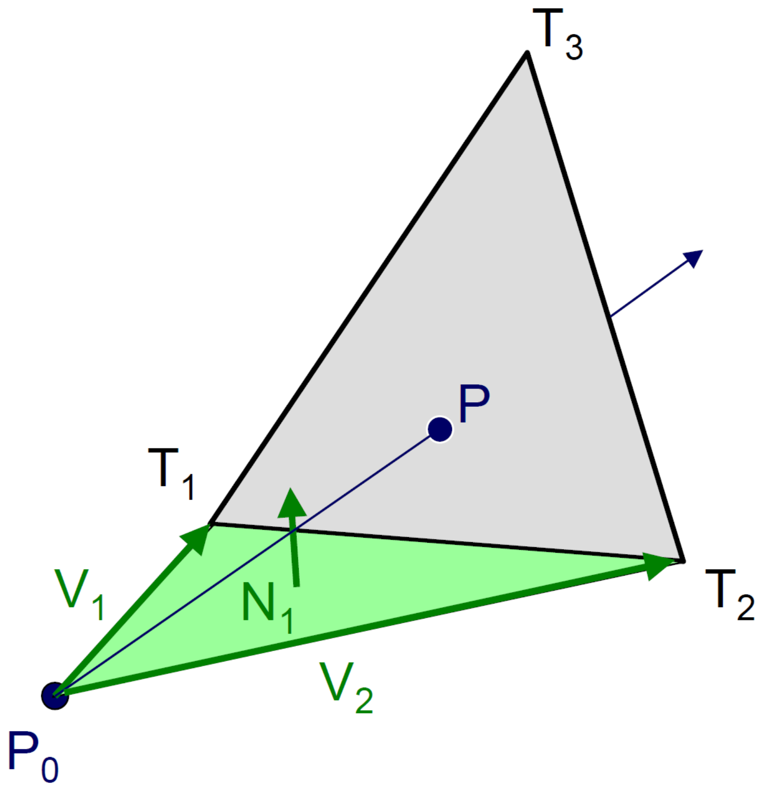
$N_1 = V_2 \times V_1$

Normalize $N_1$

if $((P - P_0) \cdot N_1 < 0)$

return FALSE;

end



# Ray-Triangle Intersection II

- Check if the point is inside parametrically

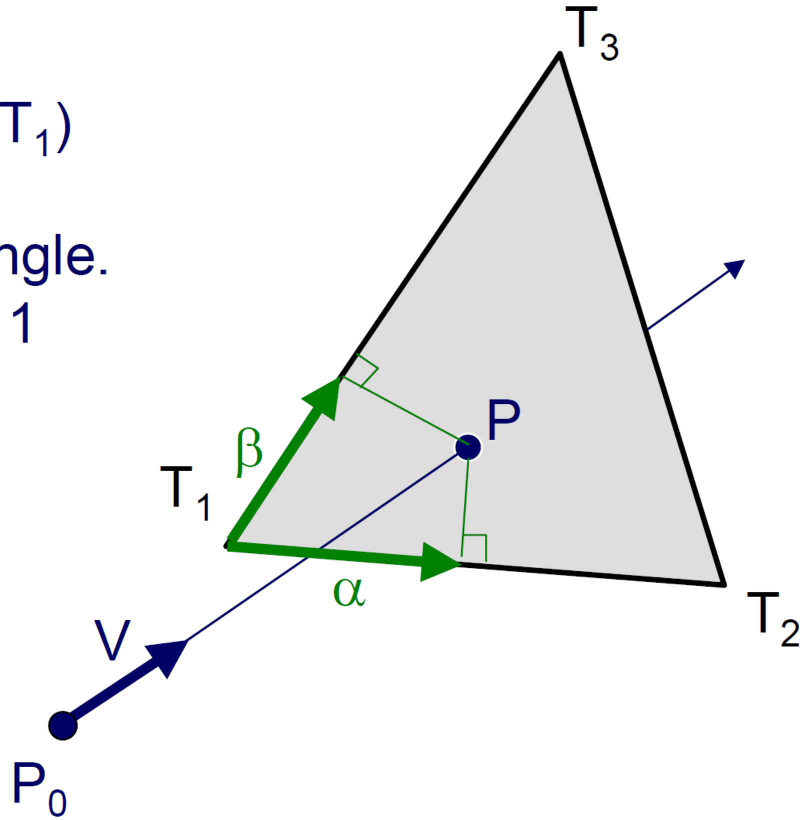
Compute $\alpha$, $\beta$:

$$P = \alpha\,(T_2\text{-}T_1) + \beta\,(T_3\text{-}T_1)$$

Check if point inside triangle.

$$0 \leq \alpha \leq 1 \text{ and } 0 \leq \beta \leq 1$$

$$\alpha + \beta \leq 1$$



# Other Ray-Primitive Intersection

- Cone, Cylinder, Ellipsoid
  - Similar to sphere
- Box
  - Intersect 3 front-facing planes, return closest
- Convex Polygon
  - Same as triangle
- Concave polygon
  - Same plane intersection
  - Complex point-in-polygon test

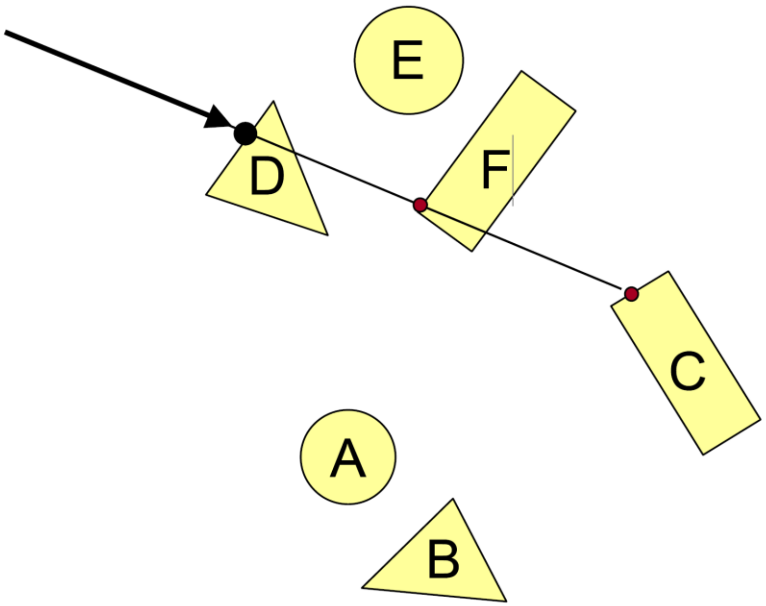
# Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle
  - **Groups of primitives (scene)**
  - Acceleration Techniques
    - Bounding volume hierarchies
    - Spatial partitions
      - Uniform grids
      - Octrees
      - BSP trees

# Ray-Scene Intersection

- Find intersection with closest primitive in group

```
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) {
         min_primitive = primitive
         min_t = t
      }
   }
   return Intersection(min_t, min_primitive)
}
```



# Ray-Scene Intersection

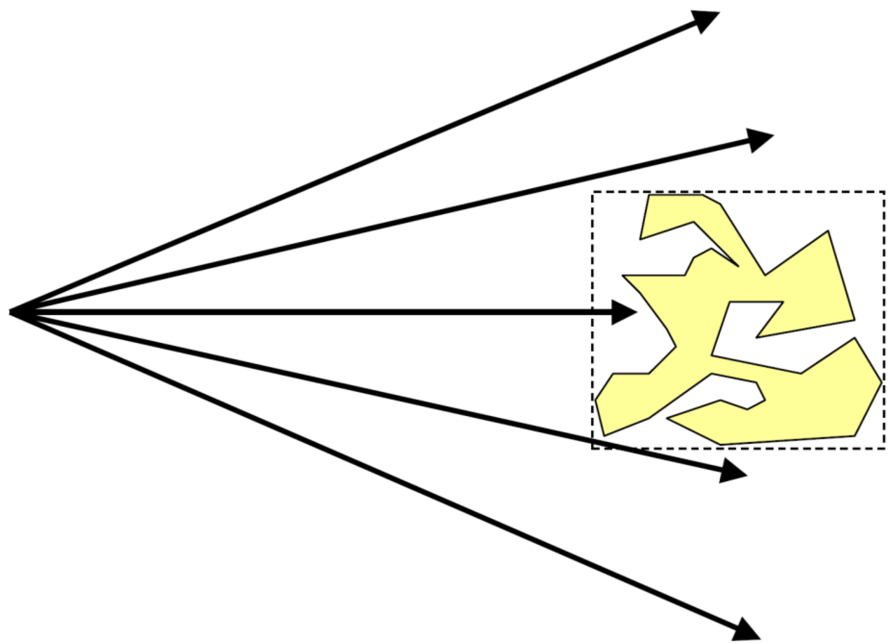
- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)
  - **Acceleration Techniques**
    - **Bounding volume hierarchies**
    - Spatial partitions
      - Uniform grids
      - Octrees
      - BSP trees

# Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)
  - **Acceleration Techniques**
    - **Bounding volume hierarchies**
    - Spatial partitions
      - Uniform grids
      - Octrees
      - BSP trees

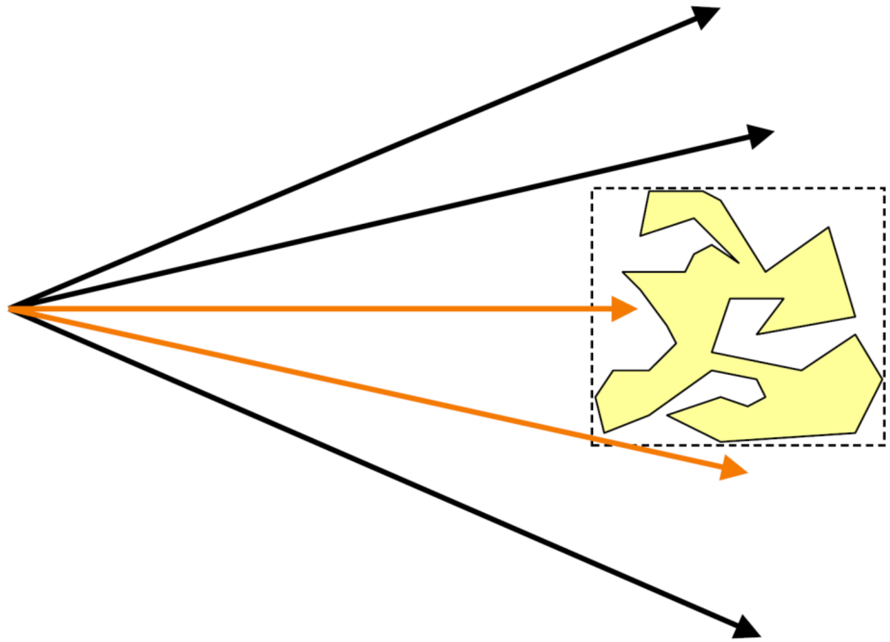
# Bounding Volumes

- Check intersection with simple shape first



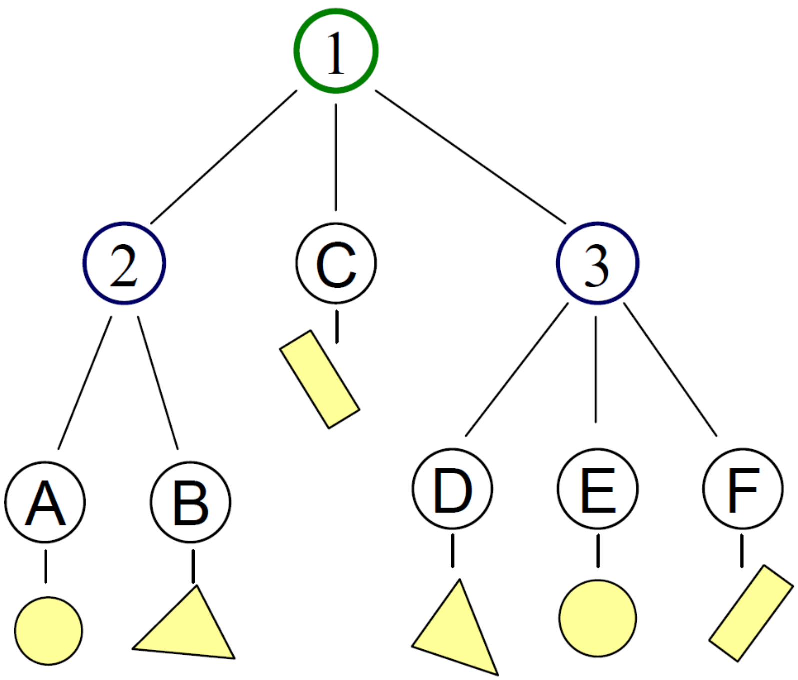
# Bounding Volumes

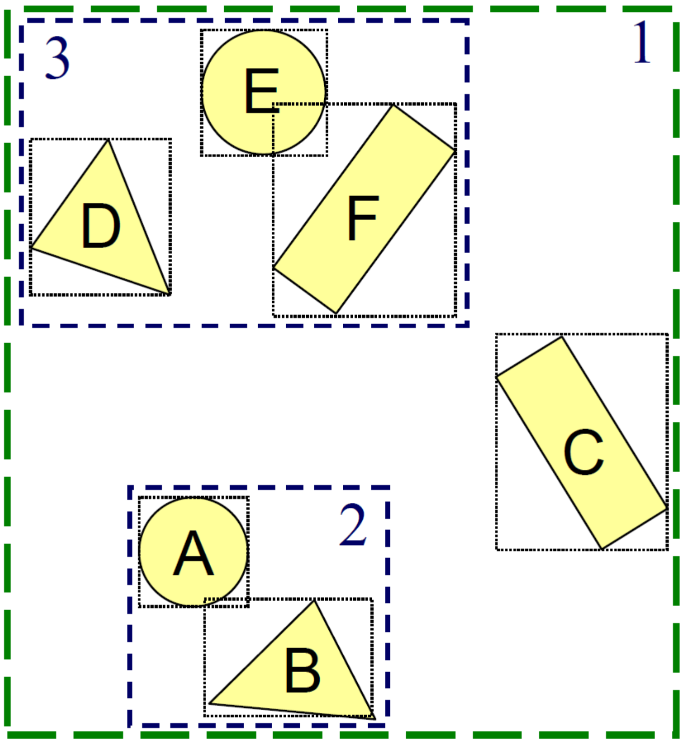
- Check intersection with simple shape first



# Bounding Volume Hierarchies

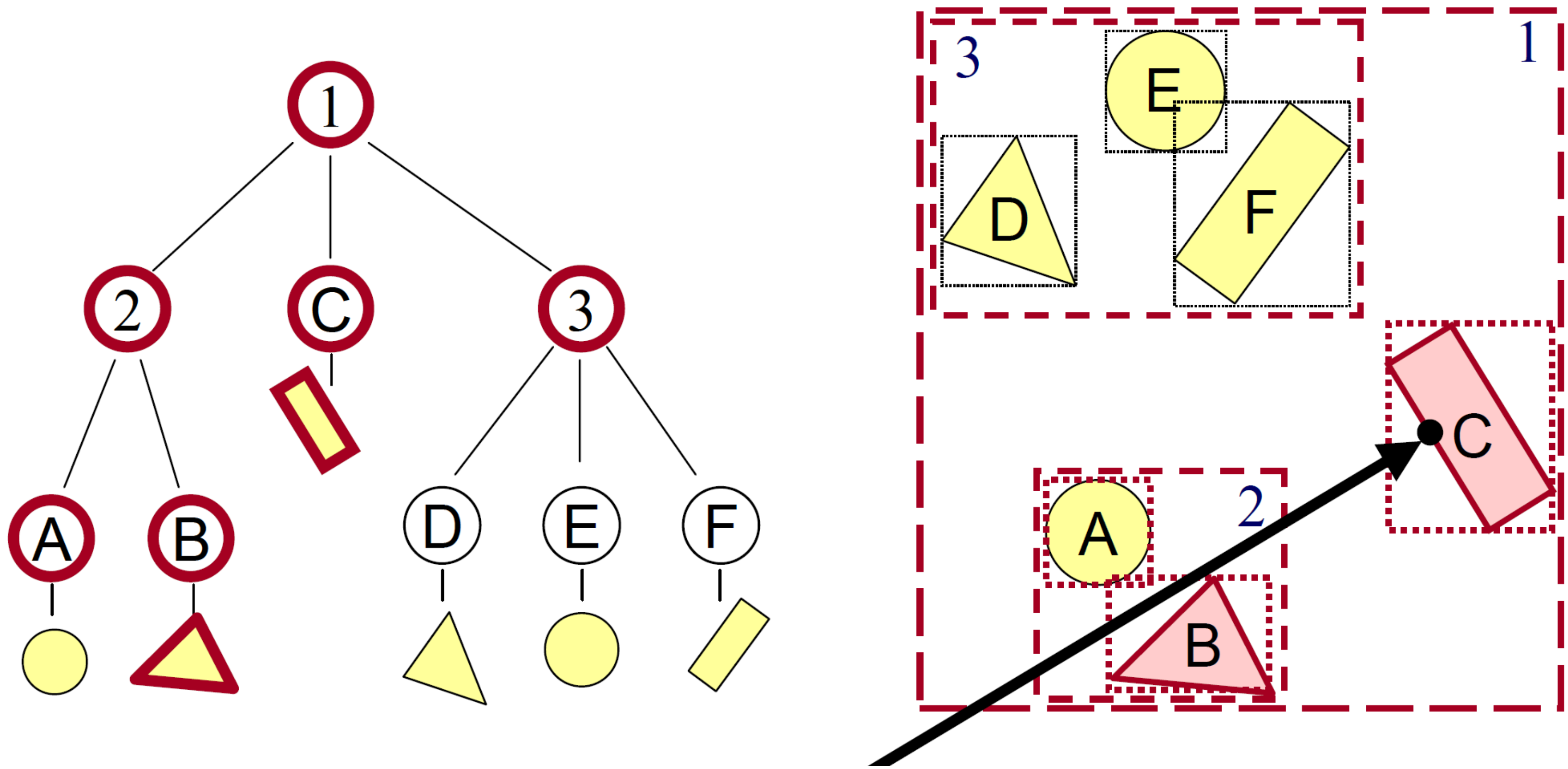
- Build hierarchy of bounding volumes
  - Bounding volume of interior node contains all children



# Bounding Volume Hierarchies

- Use hierarchy to accelerate ray intersections
  - Intersect node contents only if hit bounding volume

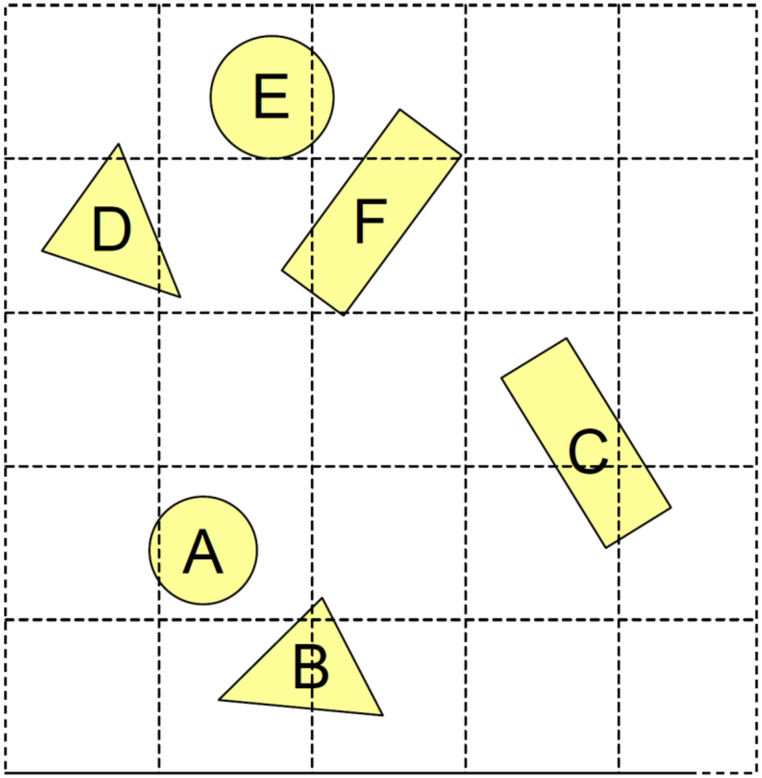


# Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)
  - **Acceleration Techniques**
    - Bounding volume hierarchies
    - **Spatial partitions**
      - Uniform grids
      - Octrees
      - BSP trees

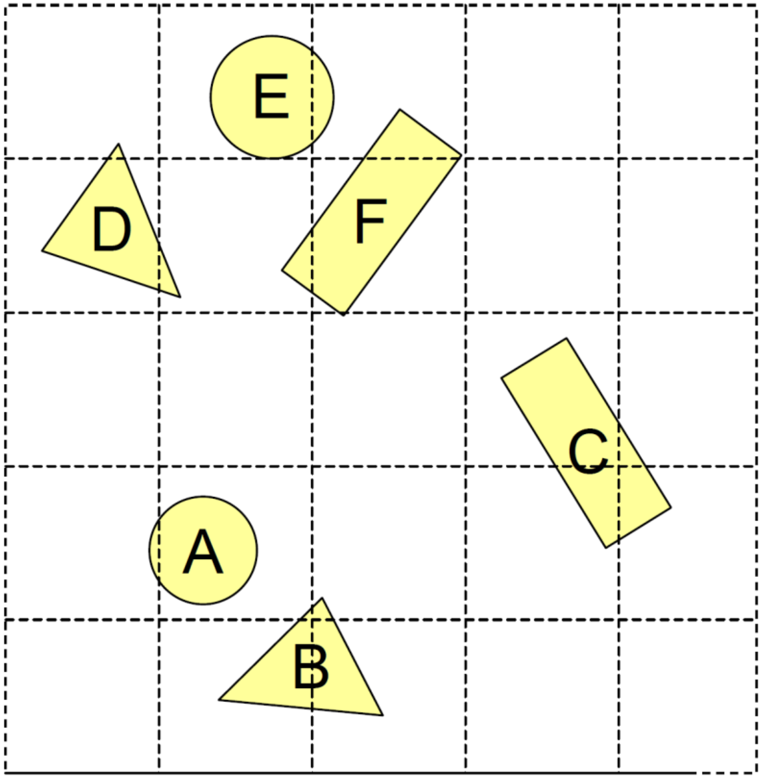
# Uniform Grid

- Construct uniform grid over scene
  - Index primitives according to overlaps with grid cells



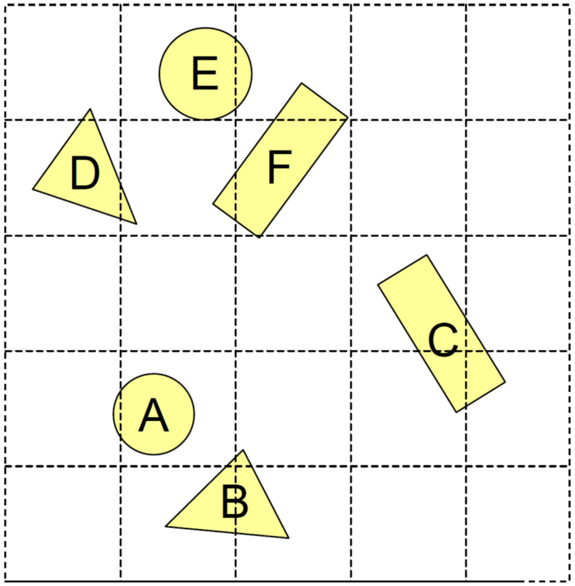
# Uniform Grid

- Trace rays through grid cells
  - Only intersect with primitives from traversed cells



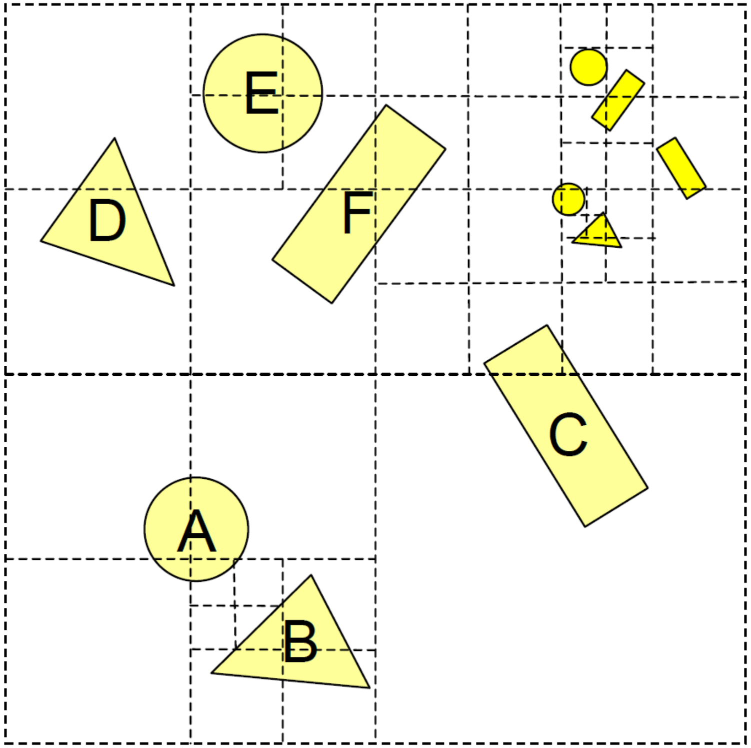
# Uniform Grid

- Potential problems:
  - How to choose grid size ?
    - Fine grid => Too computationally expensive
    - Coarse grid => Little benefit



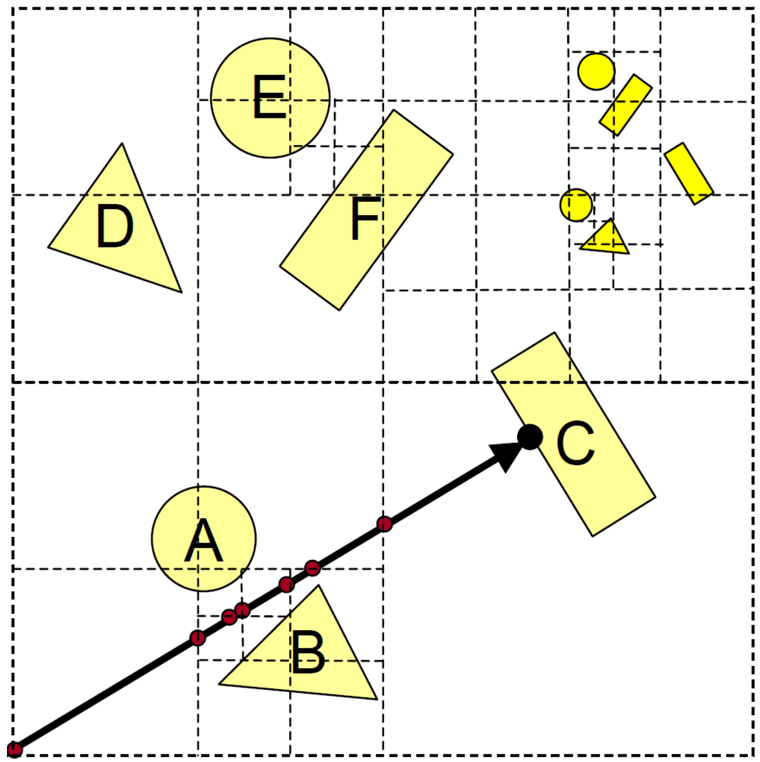
# Octree

- Construct adaptive grid over scene
  - Recursively subdivide box-shaped cells into 8 octants
    - 4 quadrants in 2D (Quadtree)
  - Index primitives by overlap with cells



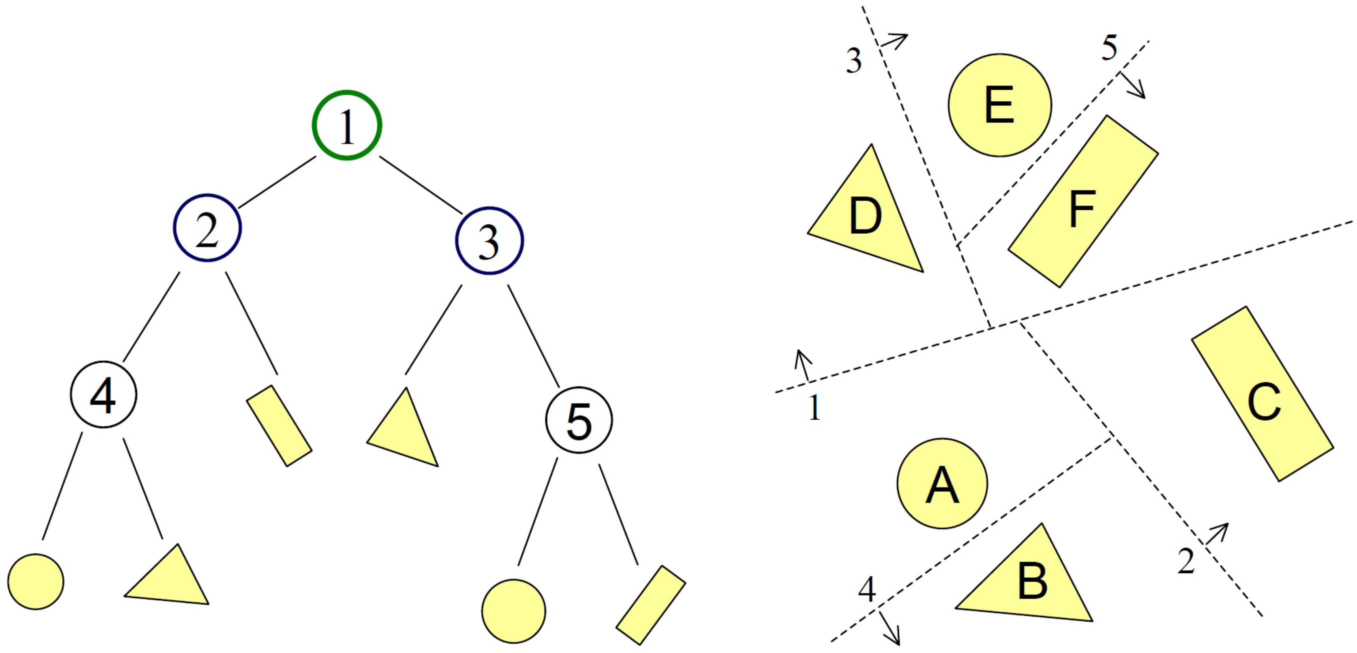
# Octree

- Trace rays through neighbour cells
  - Fewer cells
  - More complex neighbour finding



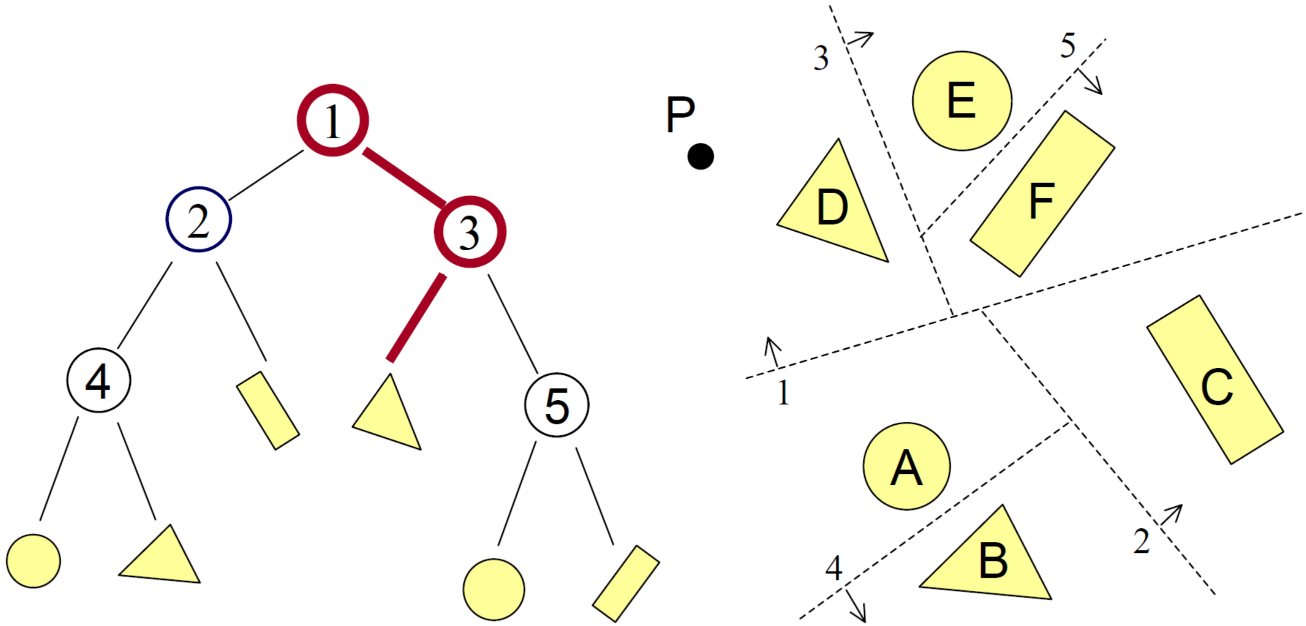
# Binary Space Partition (BSP) Tree

- Recursively partition space by planes
  - Every cell is a convex polyhedron



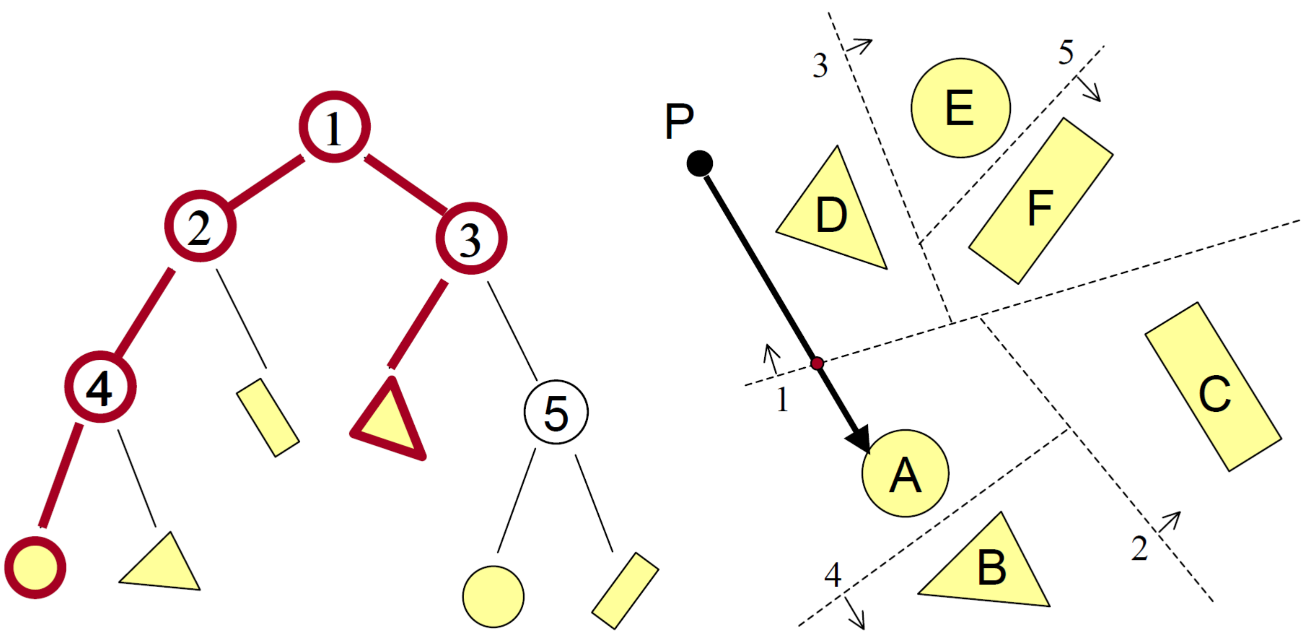
# Binary Space Partition (BSP) Tree

- Simple recursive algorithms
  - Example: Point finding



# Binary Space Partition (BSP) Tree

- Trace rays by recursion on trees
  - BSP construction enables simple front-to-back traversal



# Other Accelerations

- Screen space coherence
  - Check last hit first
  - Beam tracing
  - Pencil tracing
- Memory coherence
  - Large screens
- Parallelism
  - Ray tracing is “embarrassingly parallelizable”

# Ray Casting

- Simple implementation

```
Image RayCast(Camera camera, 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(camera, i, j);
         Intersection hit = FindIntersection(ray, scene);
         image[i][j] = GetColor(scene, ray, hit);
      }
   }
   return image;
}
```

# Shading

- Must derive computer models for ...
  - Emission at light sources
  - Scattering at surfaces
  - Reception at camera
- Desirable features ...
  - Concise
  - Efficient to compute
  - "Accurate"

# Overview

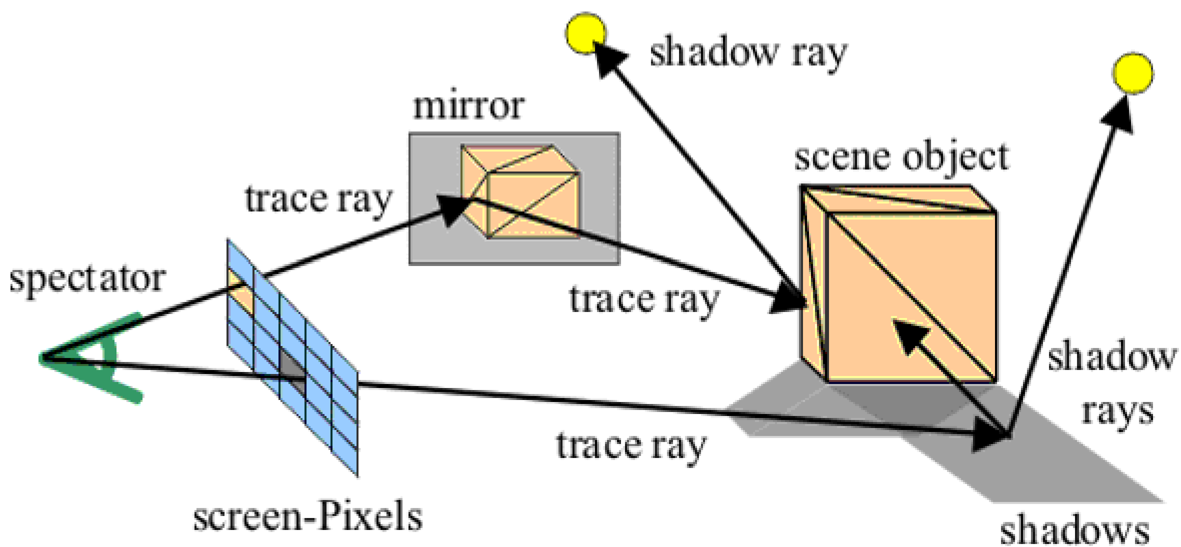
- Direct Illumination
  - Emission at light sources
  - Scattering at surfaces
  - Gouraud shading
- Global Illumination
  - Shadows
  - Refractions
  - Inter-object reflections

# Introduction

Raytracing

# Ray Tracing

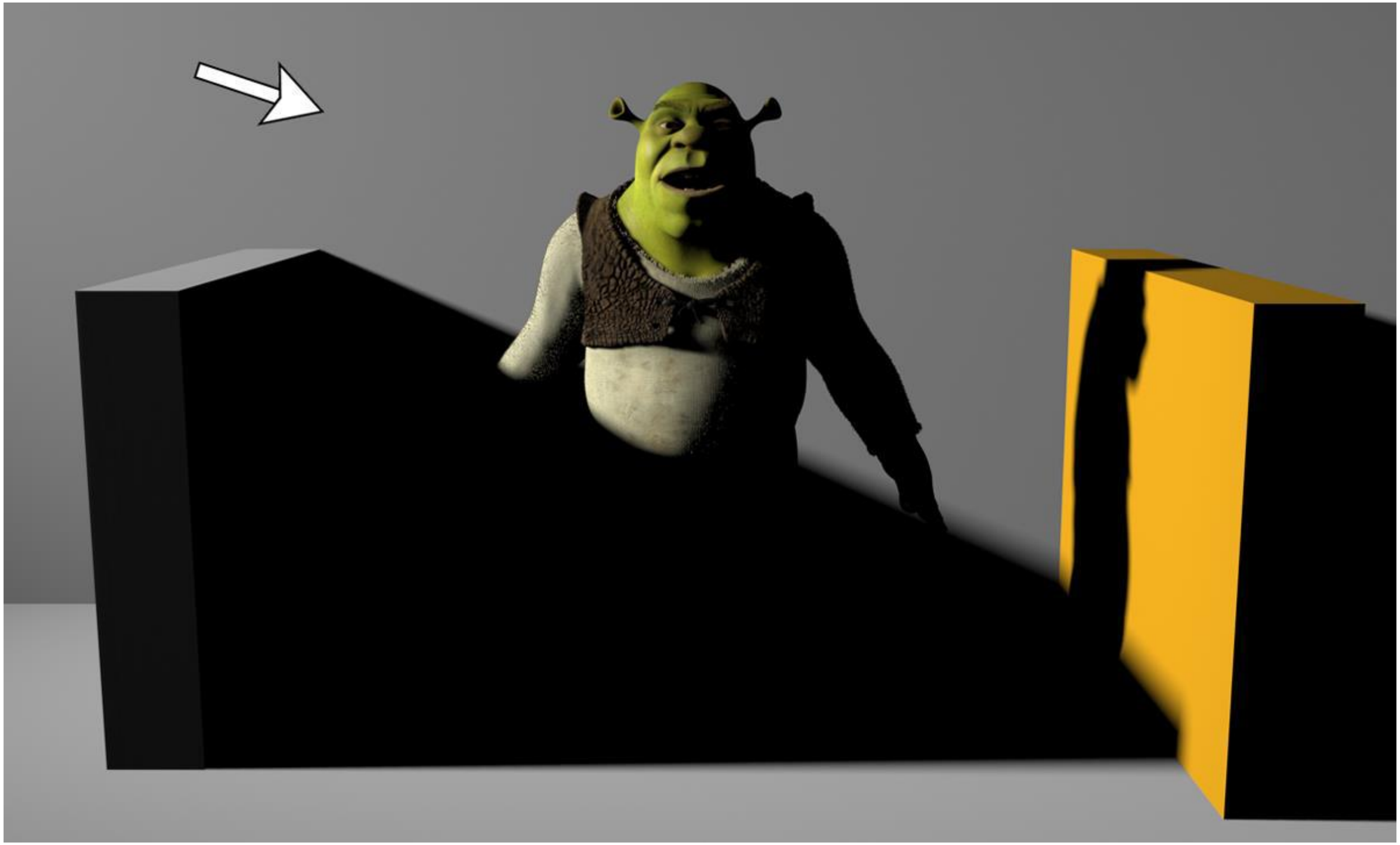
- Rays are casted and recursively traced
- Secondary reflected, refracted and shadow rays are casted



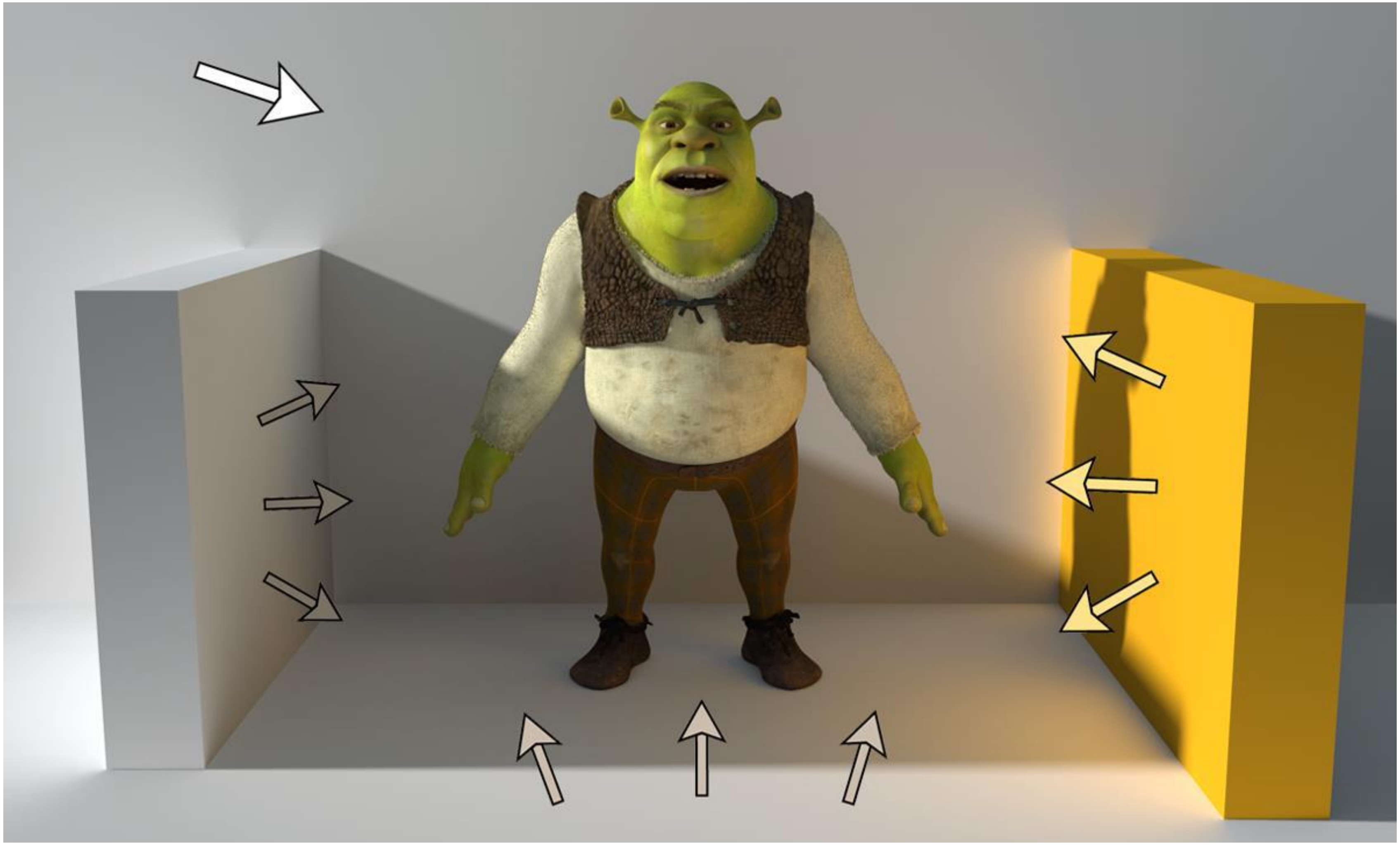
# Ray Tracing

- Photorealistic rendering
- Global illumination technique

# Local Illumination



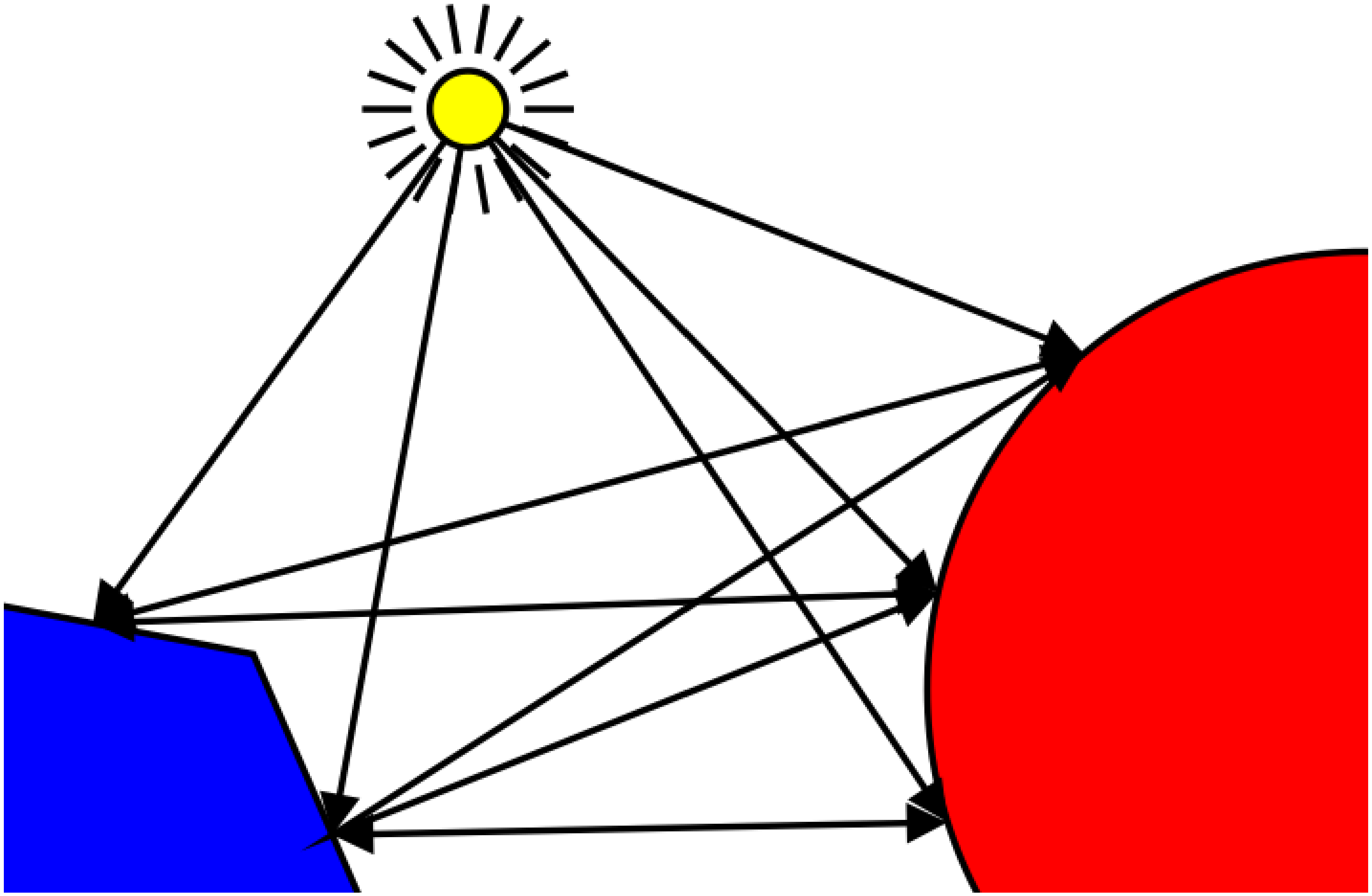
# Global Illumination



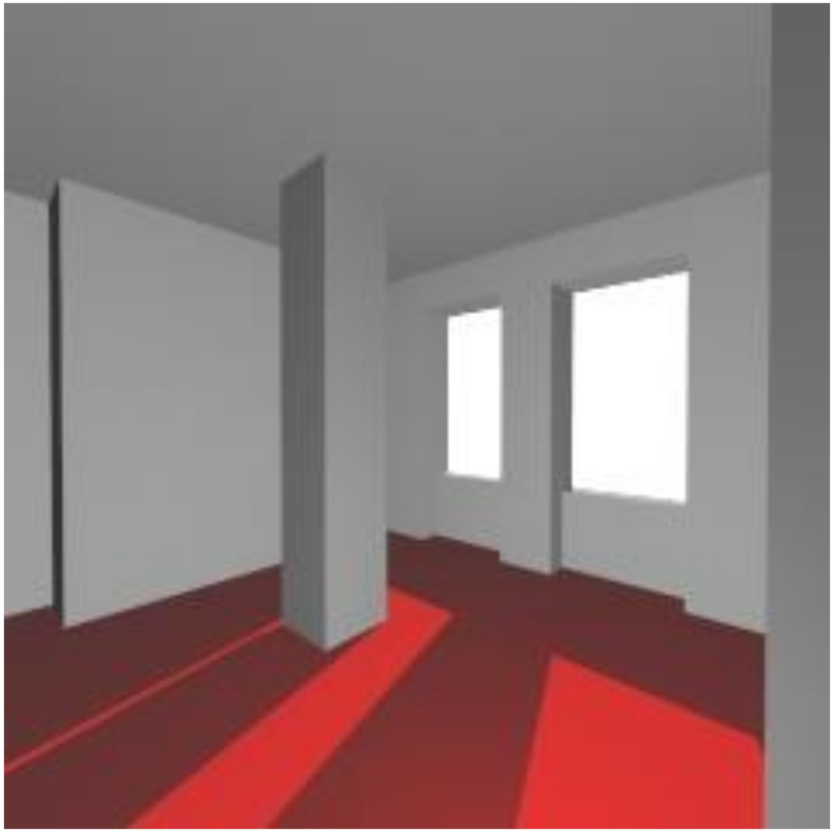
# Radiosity

- Physically based
- Object hit by light becomes a new light source
- Not only object-light interaction
- But also object-object light interaction
- Energy exchange between objects

# General situation



# Raytracing vs. radiosity



http://www.soe.ucsc.edu/classes/cmps161/Winter04/projects/aames/index.htm

# How the lectures should look like #2

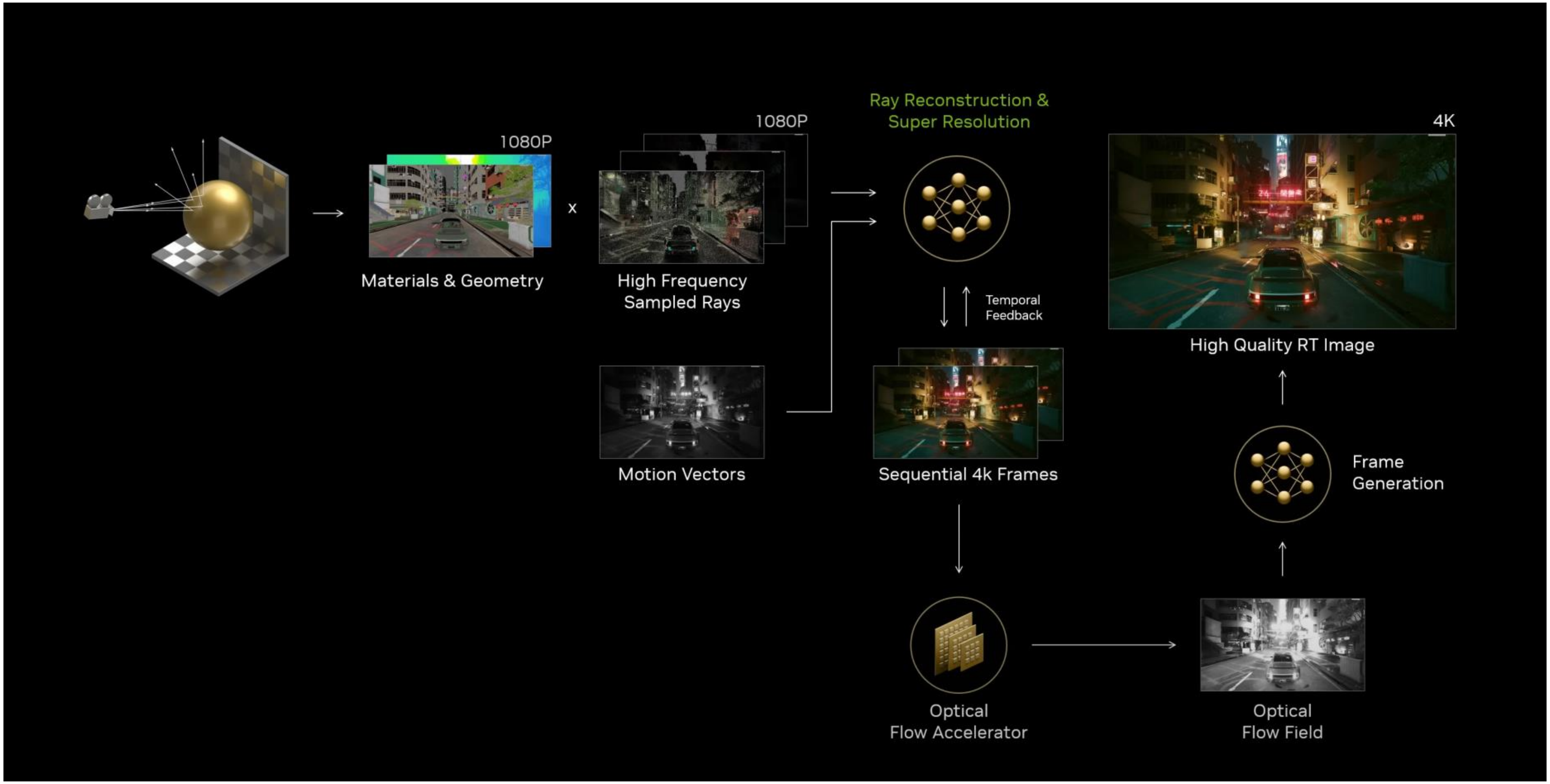
- Ask questions, please!!!
- Be communicative
- More active you are, the better for you!

# NVIDIA RTX

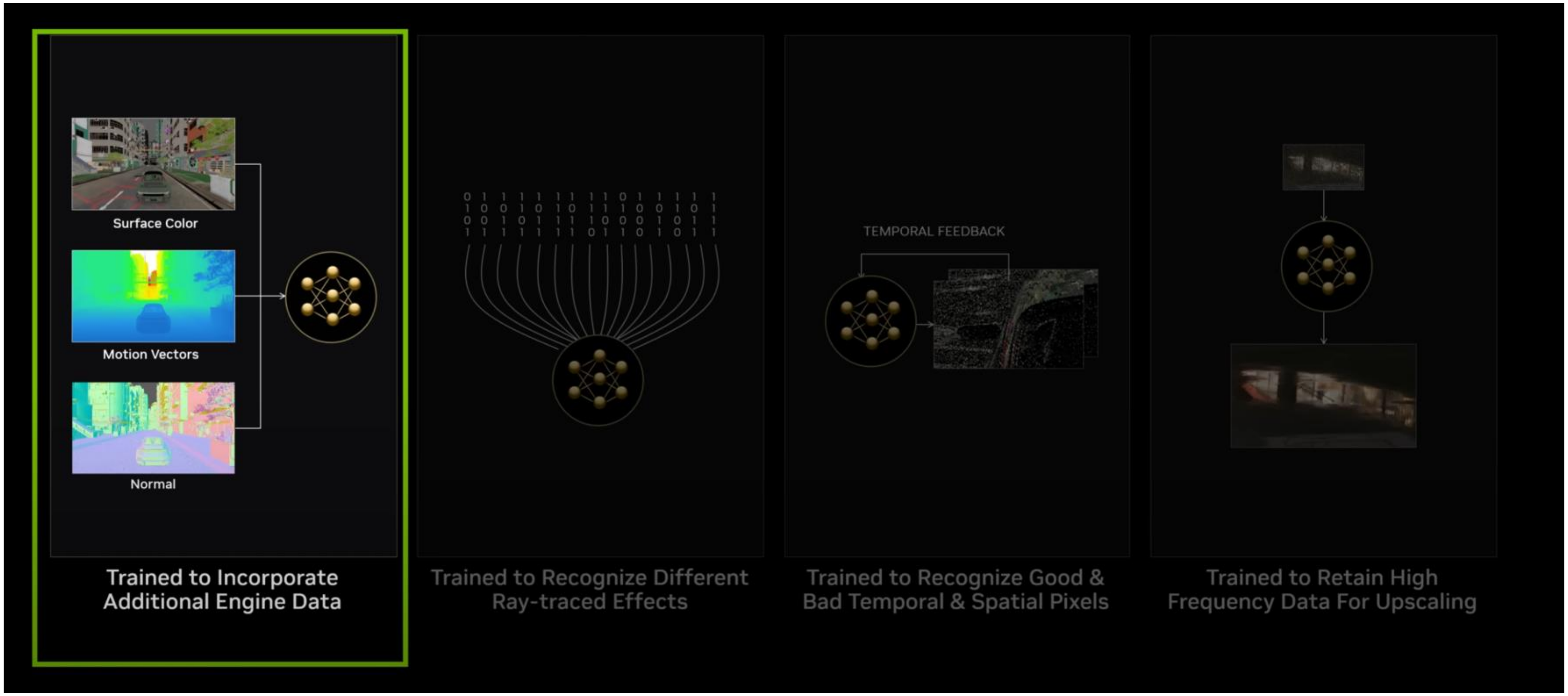
- NVIDIA DLSS
  - Deep Learning Super Sampling (DLSS)
  - Neural network + Tensor cores
    - convolutional auto-encoder neural network
  - Raytracing (trained to recognize RT effects)
    - edge enhancement
    - spatial anti-aliasing (supersampling to 64 samples per pixel)
  - Denoising (temporal feedback)
  - Upscaling (retaining high frequency data)

- DLSS 2.0
  - temporal anti-aliasing upsampling (TAAU)
- DLSS 3.0
  - motion interpolation + Optical Flow Accelerator (OFA)
- DLSS 3.5
  - multiple denoising algorithms replaced with a single AI model trained on 5x more data

# DLSS inference



# DLSS training



# NVIDIA DLSS

- DLSS (v 3.5)
- New Ray Reconstruction Enhances Ray Tracing with AI
  - https://youtu.be/sGKCrcNsVzo

# Acknowledgements

- Thanks to all the people, whose work is shown here and whose slides were used as a material for creation of these slides:

Matej Novotný, GSVM lectures at FMFI UK

Peter Drahoš, PPGSO lectures at FIIT STU

Output of all the publications and great team work

Very best data from 3D cameras

# Questions ?!

Skeletex
RESEARCH

www.skeletex.xyz

madaras@skeletex.xyz

martin.madaras@fmph.uniba.sk