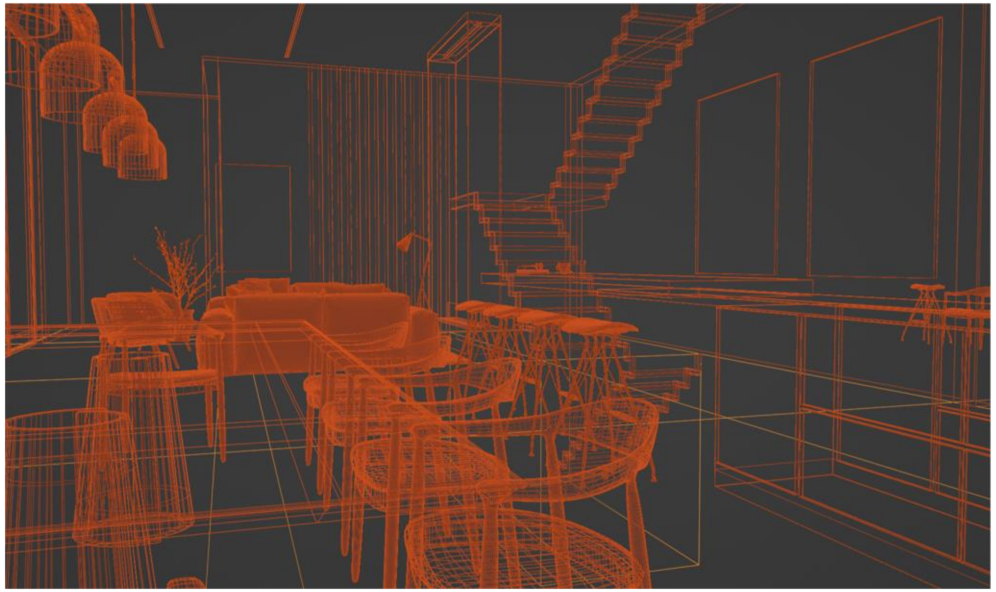


Scene by Lynxsdesign

Fundamentals of Ray Tracing

Computer Graphics 24/25 – Lecture 1

A few quick words on the format

- Lectures only every 2nd Monday (check webpage when in doubt)
- Lectures provide a condensed overview of the topics
  - All exam-relevant topics are covered
  - But details may be missing
  - Suggested reading materials in the slides supplement those
- Don't understand something?
  1. Check the reading materials
  2. Ask in the Q&A session (the Mondays where there is no lecture)

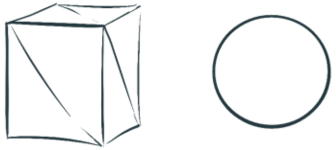
What do you need to know / understand?

- Guidance to answer that is offered via **Mini Tests** and **Assignments**
- The **Mini Tests**
  - Are **mandatory** but not graded
  - Take place before the Q&A session (every 2nd Monday)
  - Resemble the exam (if you don't get a question right, you should read up on that topic)
- The practical **assignments**
  - Are **mandatory** and **graded**
  - Released after each overview lecture
  - Implementation can require additional details; you should use the reading materials to study those
  - **Mandatory** presentation of your submission in the **tutorial** following the deadline

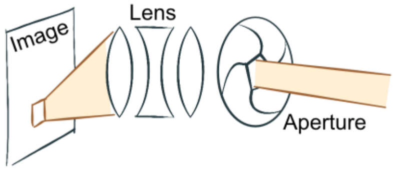
The Q&A session

- Voluntary attendance, but starts with a **mandatory** Mini Test
  - You can leave after the test if you are bored 🥱
- We'll discuss the test solutions immediately afterwards
- Then, the floor is open for **public questions**
  - Ask clarifications on stuff you think might interest your peers
  - E.g., questions about Mini Test, general understanding of topics, course formalities
- After, I'll be available for **individual questions**
  - Anything you think too specific or personal to concern everyone else
  - ... or that you are too shy to ask in front of everyone 😨
- Suggestion:
  - Use the Q&A session to work on reading materials and assignment
  - Ask questions as they arise

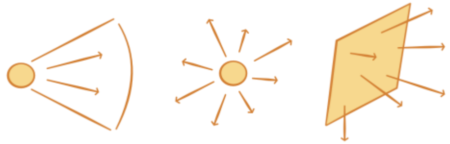
Rendering in a nutshell



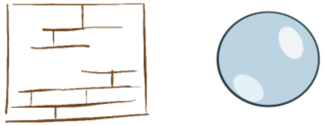
Geometries



Camera model



Light sources



Materials & Textures

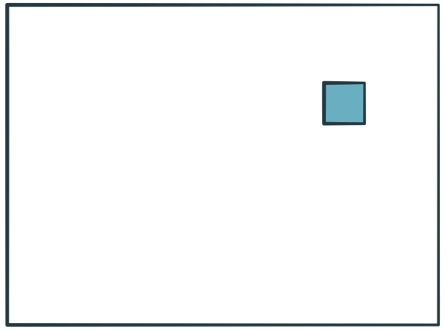
Volumes

Input data

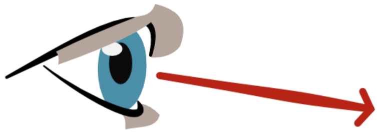
Renderer

Rendered image

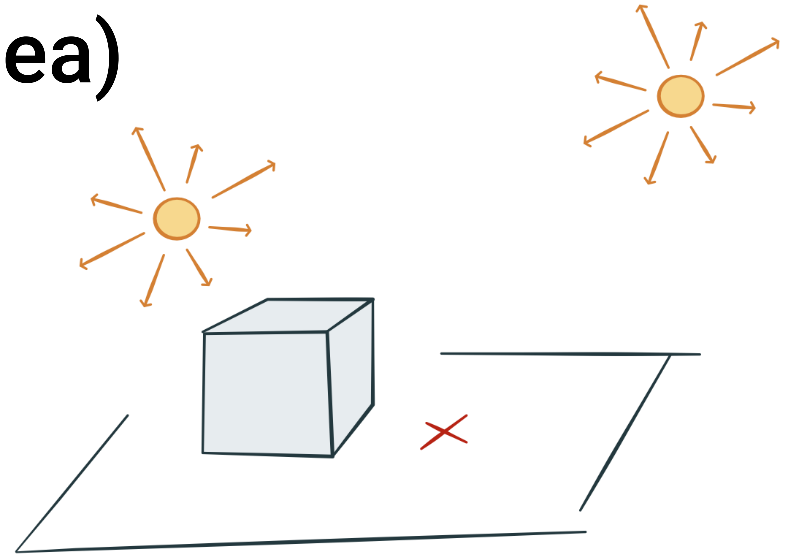
Rendering with ray tracing (the rough idea)



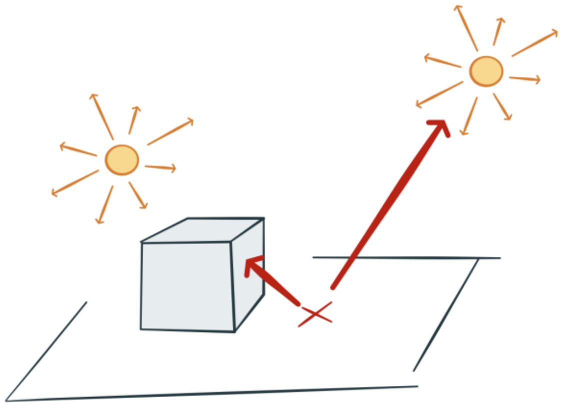
Select a pixel



Sample a ray from the camera

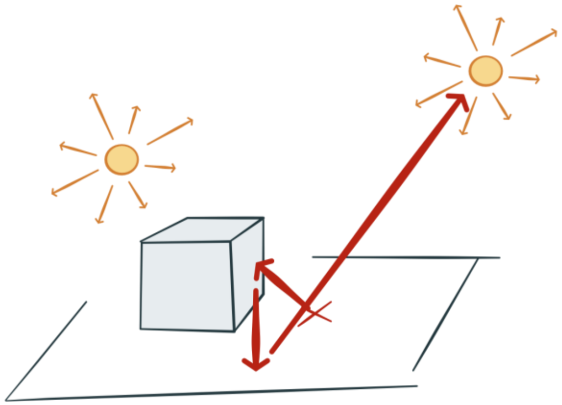


Find where it hits the scene



Direct illumination

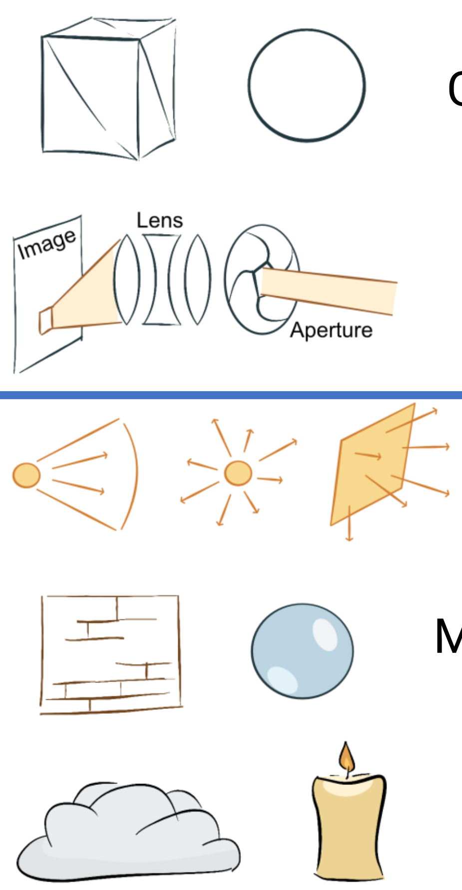
Trace rays to determine visibility of lights



Global illumination

Recursively continue paths

Today



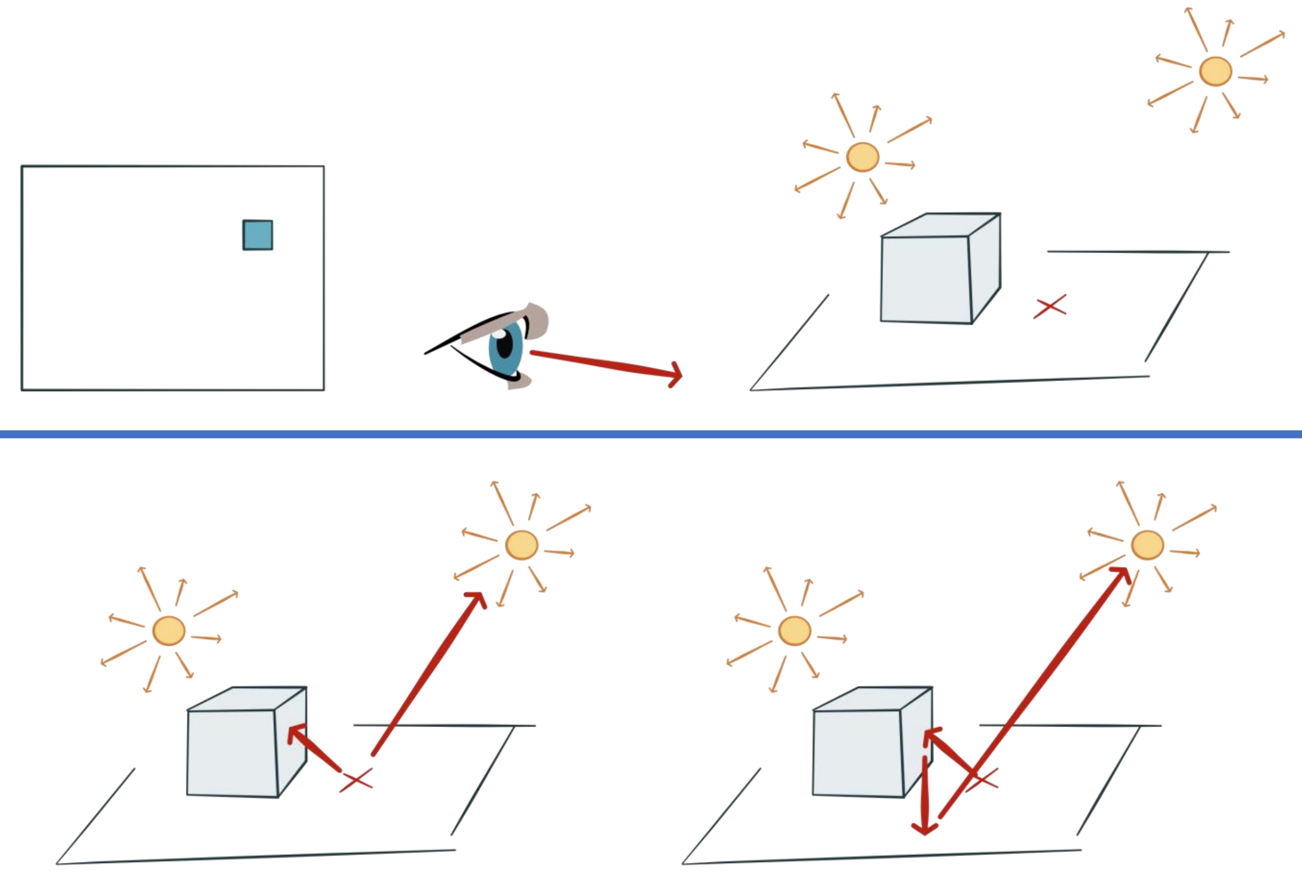
Geometries

Camera model

Light sources

Materials & Textures

Volumes

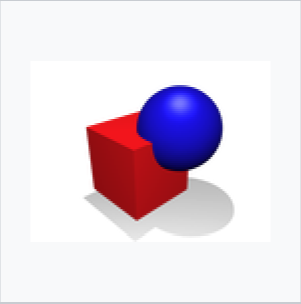


3D Scene Description

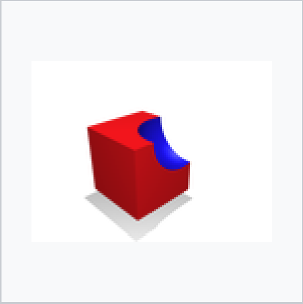
What data are we working with?

Many ways to describe geometry, e.g.,

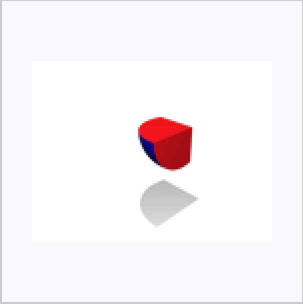
- Simple objects
  - Spheres, cylinders, boxes, ...
- Aggregation of simple objects
  - Boolean operations / constructive solid geometry (CSG)
- Curves
  - NURBS, hair
- **Polygon meshes**



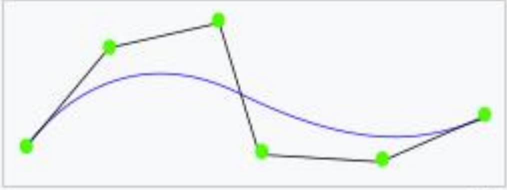
Union
Merger of two objects into one



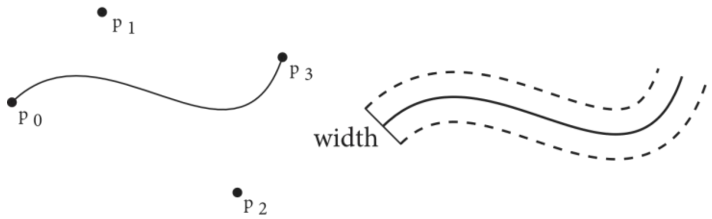
Difference
Subtraction of one object from another

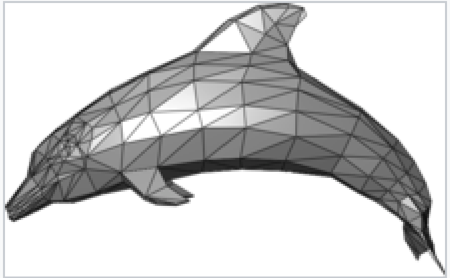


Intersection
Portion common to both objects

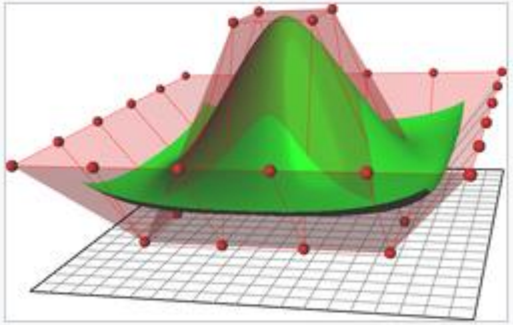


A NURBS curve. (See also: the animated creation of a NURBS spline.)





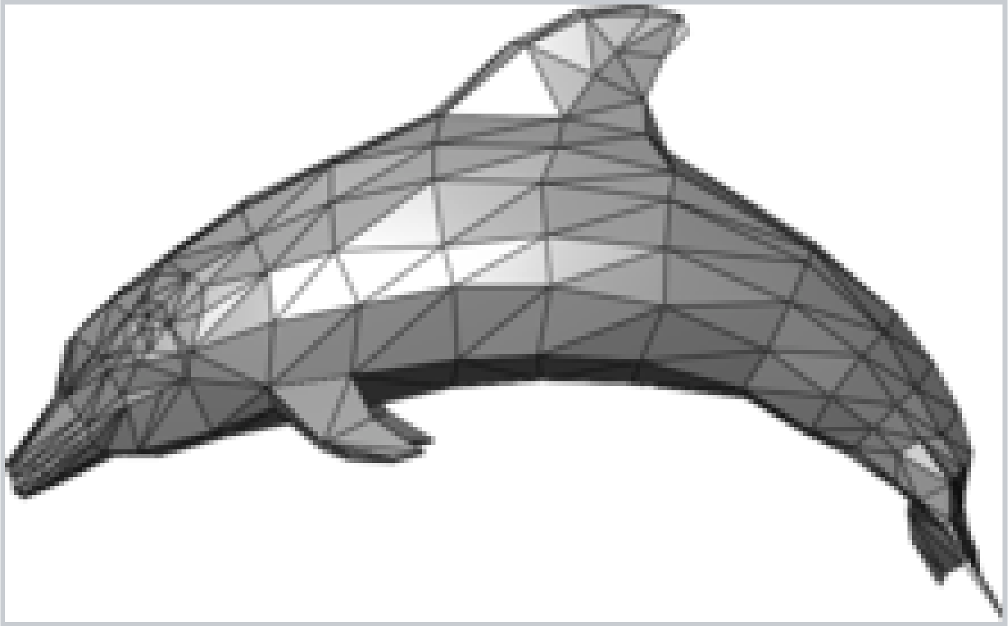
A polygon mesh of a dolphin



A NURBS surface

Why polygon meshes?

- Can (approximately) represent any shape
- Easy and fast to render and do other computations with



A polygon mesh of a dolphin

What polygons?

- Quad meshes are preferred for modeling and animation
  - Easier to manipulate
  - Artifact-free deformations
  - Artifact-free subdivision for smoothing

- Triangle meshes are popular for rendering
  - Least common denominator: Any polygon can be turned into triangles

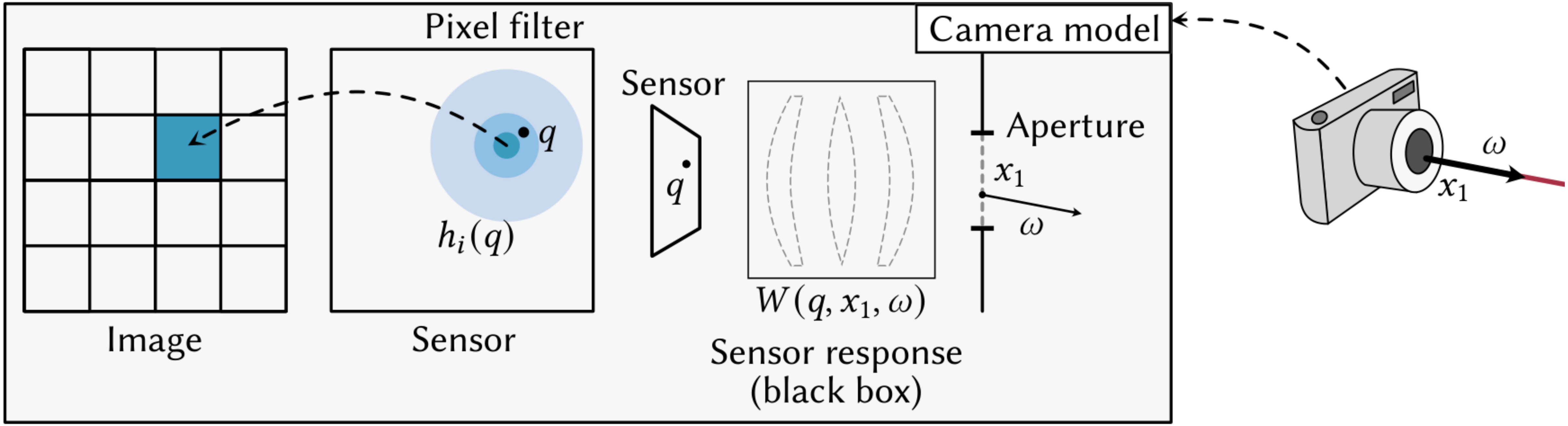
Further reading

- https://pbr-book.org/4ed/Shapes

- Try Blender to make your own meshes! https://www.blender.org/
  - Tutorial recommendations: https://www.blenderguru.com/ or https://cgcookie.com/

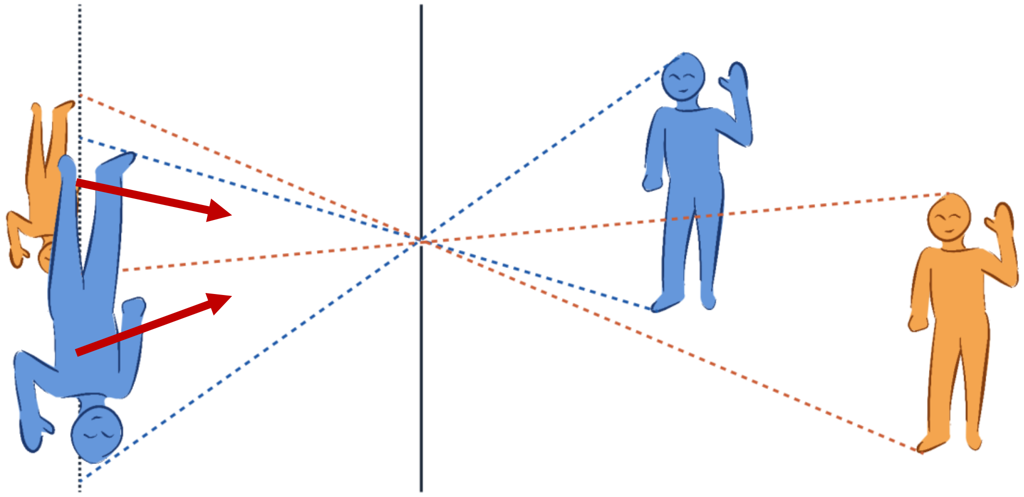
Camera models

Cameras describe how the 3D scene is projected onto the image



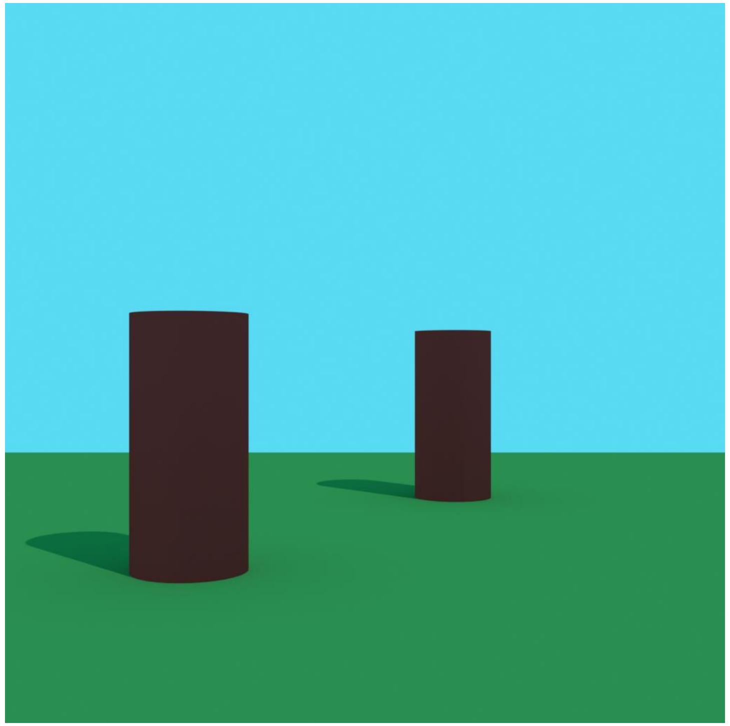
The perspective pinhole

- Camera obscura
- Crudely approximates human eye / typical camera



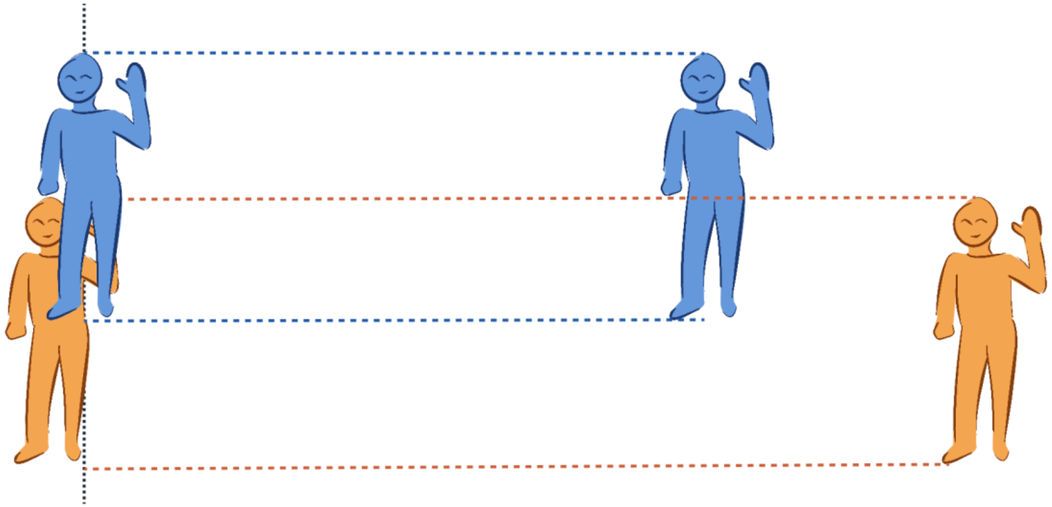
- Generating a ray from pixel x:
  - Ray origin is the camera position

→ Direction is the vector from the pixel to the pinhole

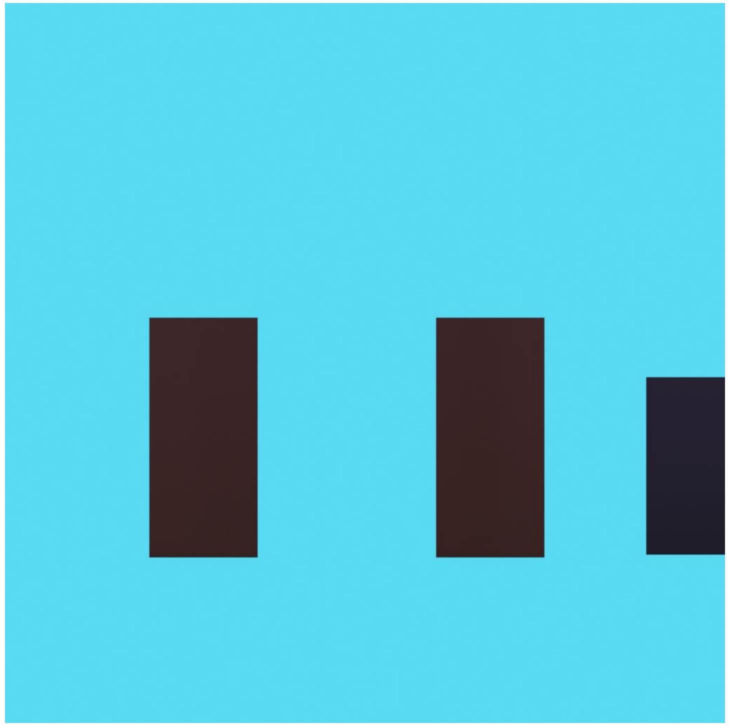


Orthographic camera

- Parallel projection of the scene onto the image plane
- Useful, e.g., during 3D modelling to judge sizes of objects

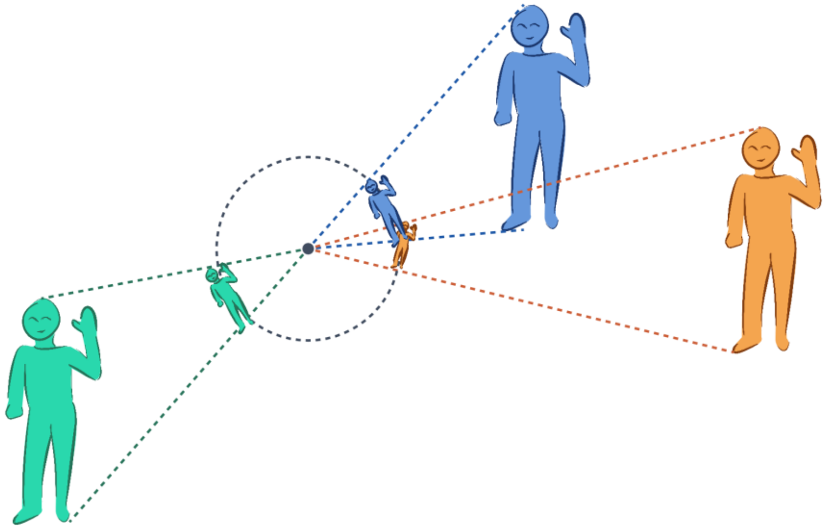


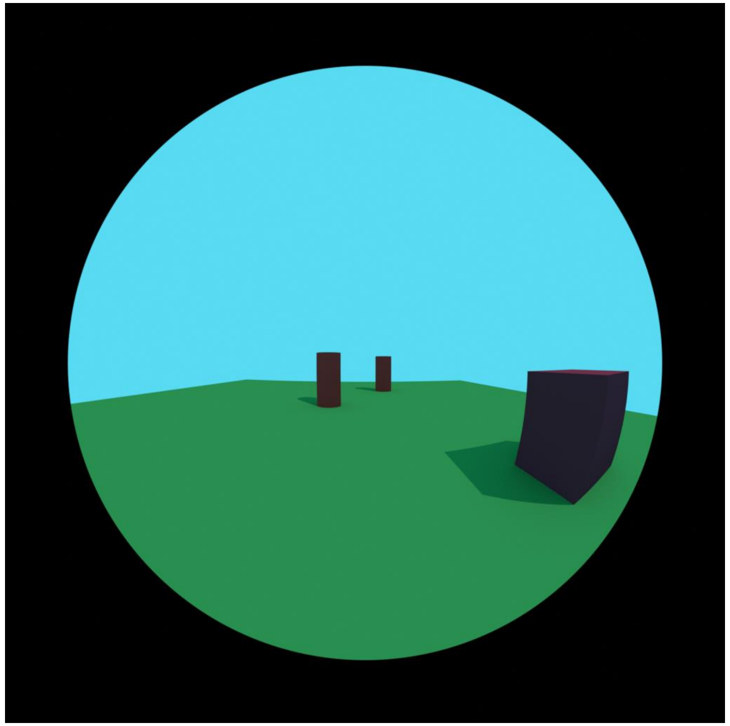
- Generating a ray from pixel x:
  - All rays have the same direction
  - Pixel position determines ray origin



Fisheye

- Projects a 180° or 360° view of the scene
- Useful for visualization, light probes, or scientific uses

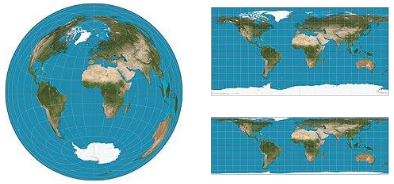




- Generating a ray from pixel x:
  - Origin is the camera position
  - Direction is computed from spherical coordinates, using a mapping

many options, just like a world map

https://en.wikipedia.org/wiki/List_of_map_projections

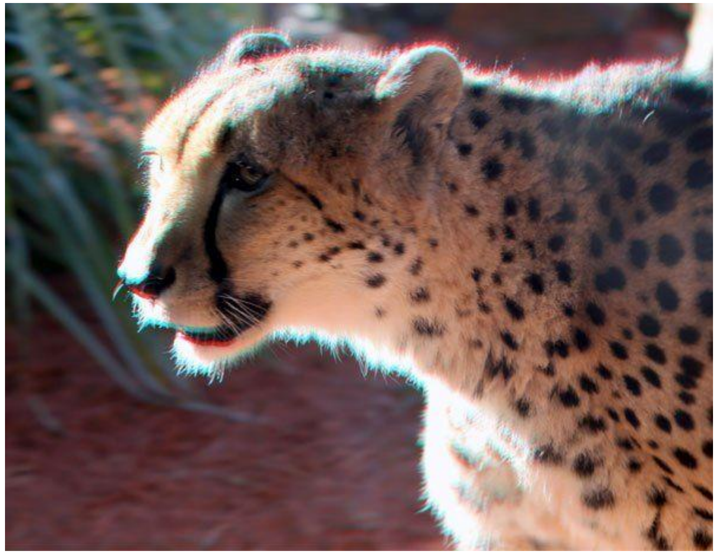


Advanced camera models simulate additional effects

↑ Depth of field and Bokeh ↓

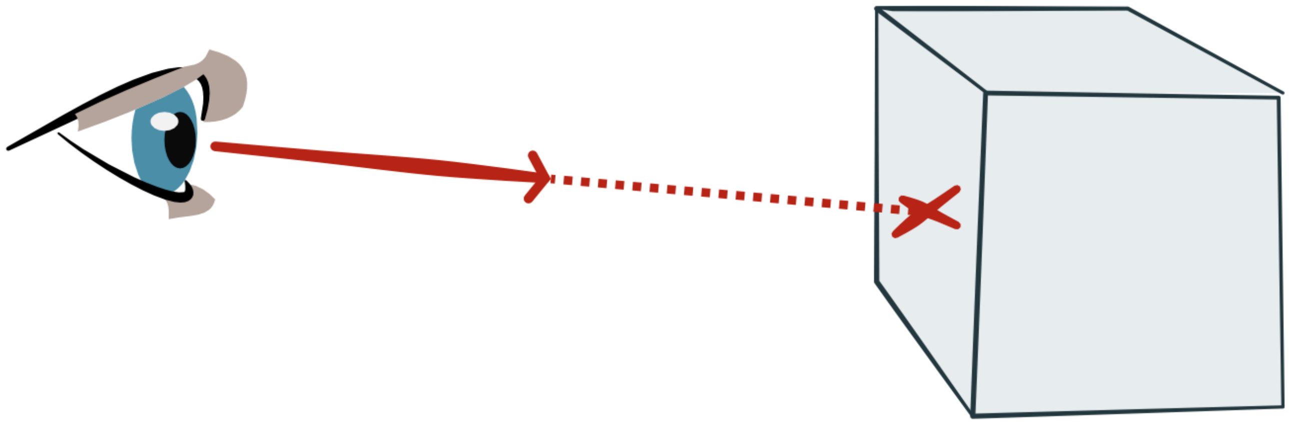
Lens flare

Chromatic aberration



Further reading

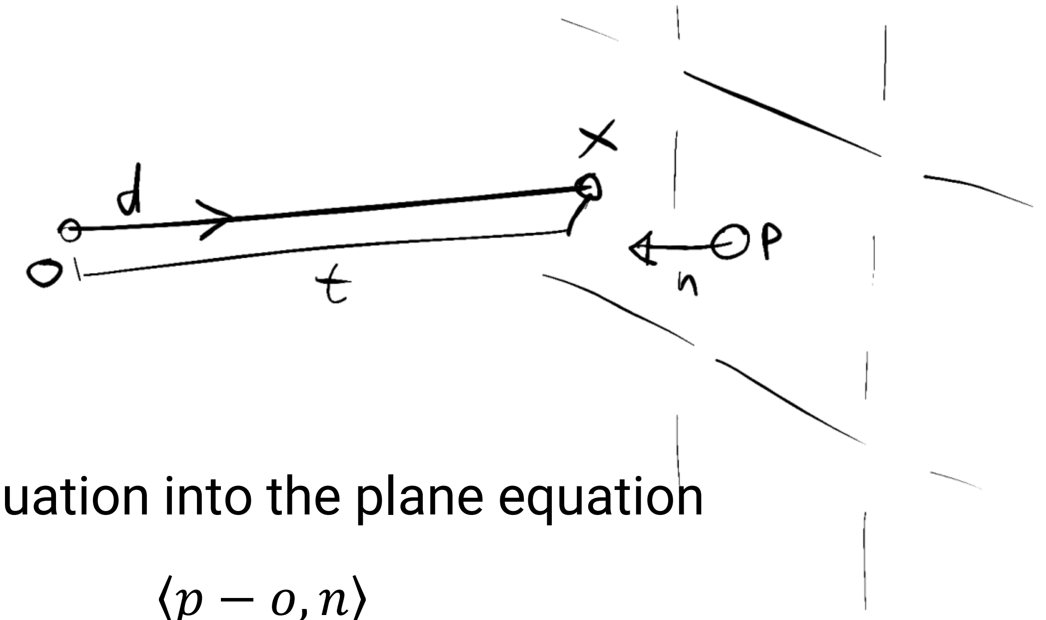
- https://www.pbr-book.org/4ed/Cameras_and_Film/Projective_Camera_Models
- https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays.html
- Hullin et al. 2012. Polynomial Optics: A Construction Kit for Efficient Ray-Tracing of Lens Systems. https://doi.org/10.1111/j.1467-8659.2012.03132.x



Ray tracing

Example: Ray-plane intersection

- A ray is defined by:
  - Origin o, direction d
  - x is on the ray if $x = o + td$
- A plane is defined by:
  - Point p, Normal n
  - x is on the plane if $\langle x - p, n \rangle = 0$



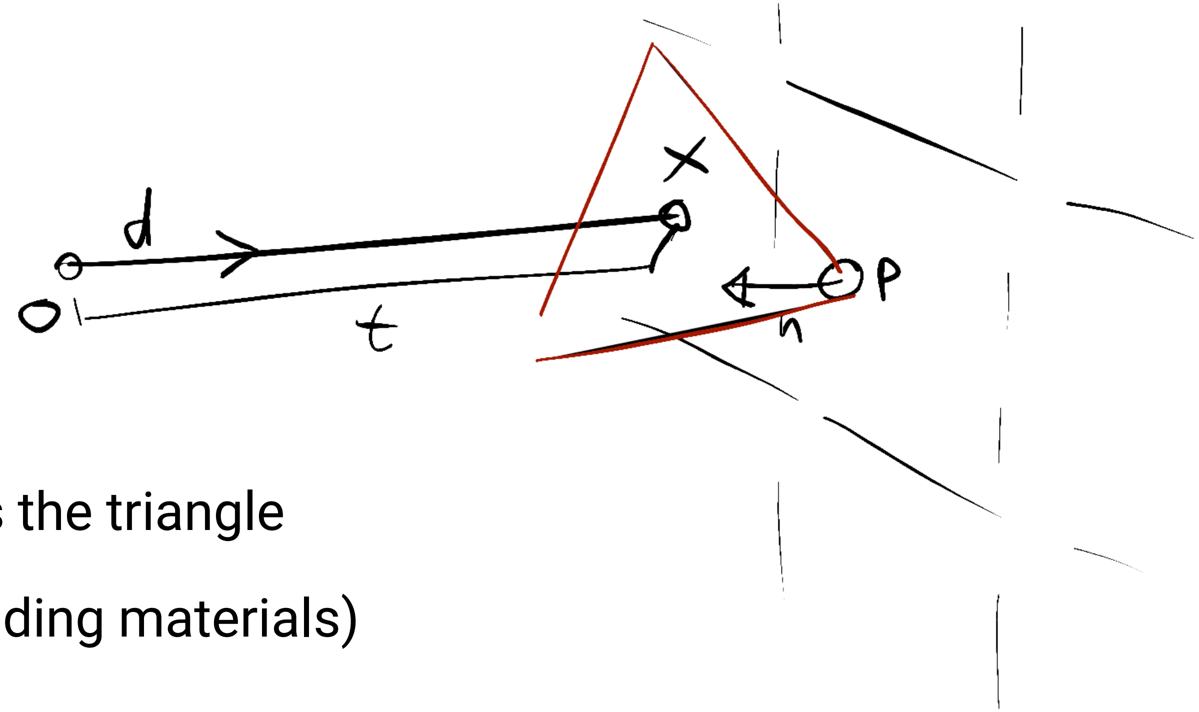
$\Rightarrow$ We find the intersection by substituting the ray equation into the plane equation

$$\langle o + td - p, n \rangle = 0 \Leftrightarrow t = \frac{\langle p - o, n \rangle}{\langle d, n \rangle}$$

- Same idea can be used for any other shape (sphere, cylinder, fractal, ...)

Example: Ray-triangle intersection (simplified)

- Triangle with corner points p_1, p_2, p_3
- Normal $n = (p_2 - p_1) \times (p_3 - p_1)$

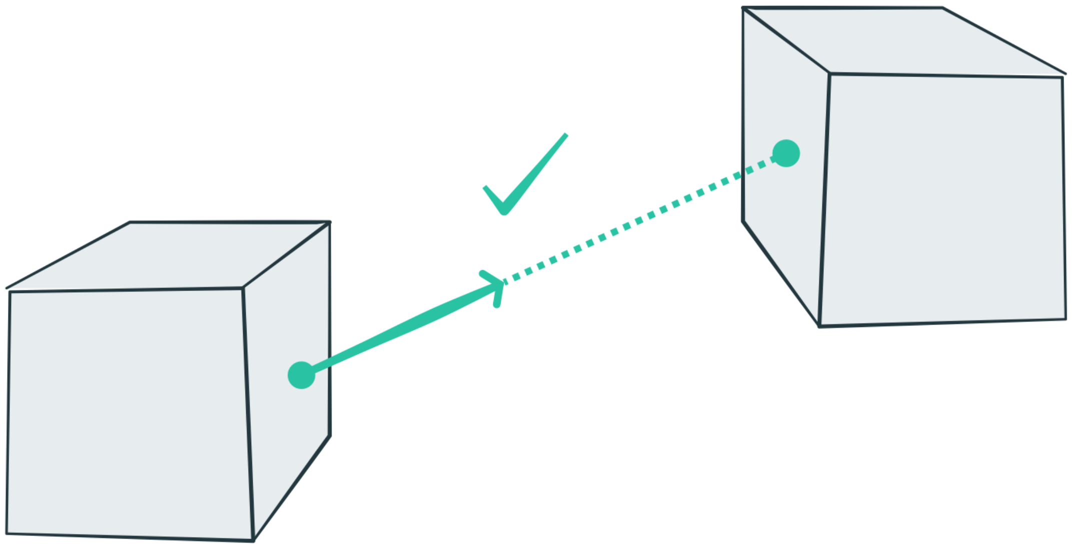


1. Intersect the ray with the plane that contains the triangle
2. Check if the point lies in the triangle (see reading materials)

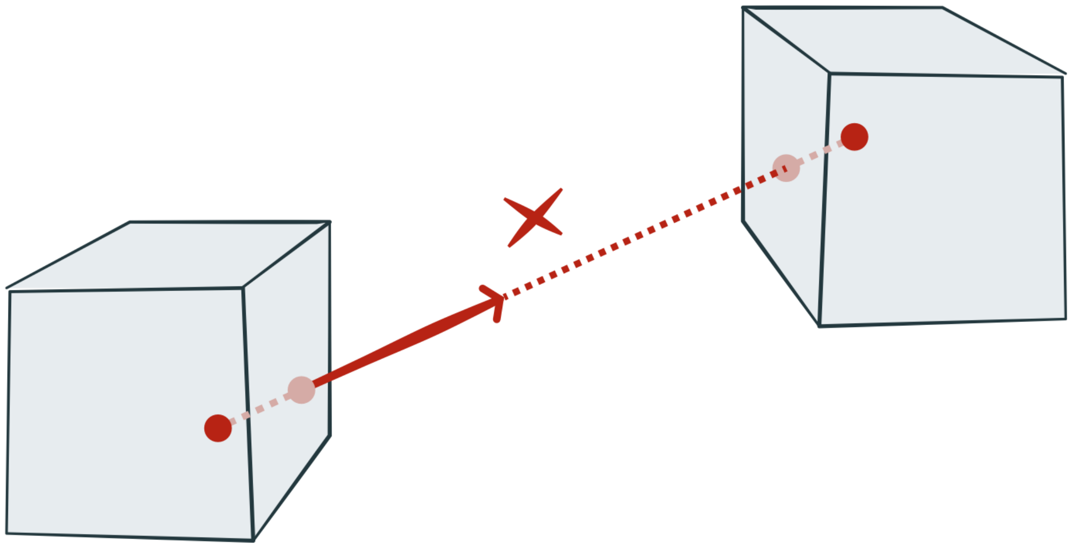
- Many algorithms exist to make this fast & precise (see https://www.realtimerendering.com/intersections.html for an overview)

Self-intersections and other numerical issues

Perfect world: no intersection in-between



Reality: floating point error



Solution:

- offset ray origin
- minimum and maximum distance for intersections

(We'll revisit this when talking about shadow rays and lighting computations)

Further reading

- Eric Lengyel. Mathematics for 3D Game Programming and Computer Graphics. 2011.
- https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection.html
- Sven Woop, Carsten Benthin, Ingo Wald. Watertight Ray/Triangle Intersection. JCGT. 2013.
- https://www.realtimerendering.com/intersections.html

Acceleration Structures

Make ray tracing scale to large geometries

- Intersecting meshes one triangle at a time is slow!
  - $O(n)$
- Acceleration structures build a tree (or similar) to prune non-visible
  - $O(\log n)$

This simple scene has

7.5 million triangles

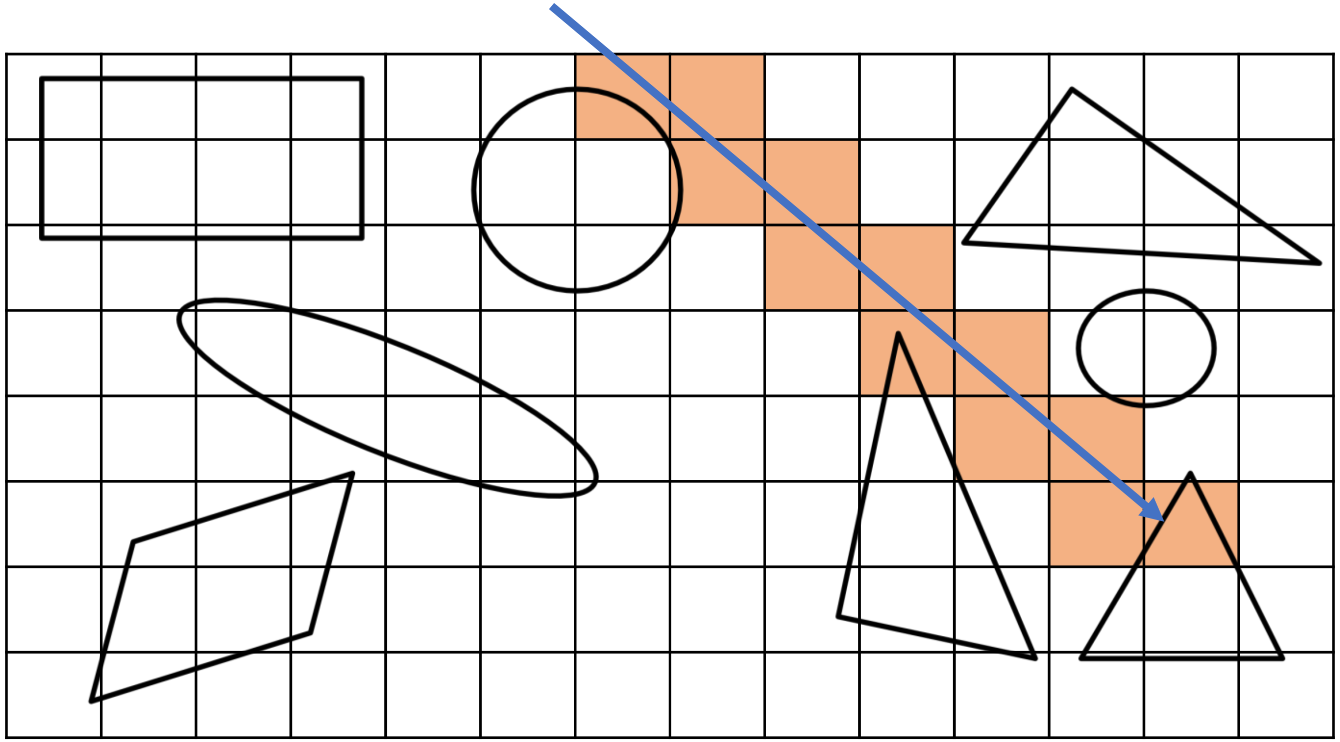
Two types

- Subdividing space
  - Grid
  - Octree
  - BSP / kd-Tree
- Subdividing objects
  - Bounding volume hierarchy (BVH)

- State-of-the-art:
  - BVH dominates
  - kd-Trees occasionally used

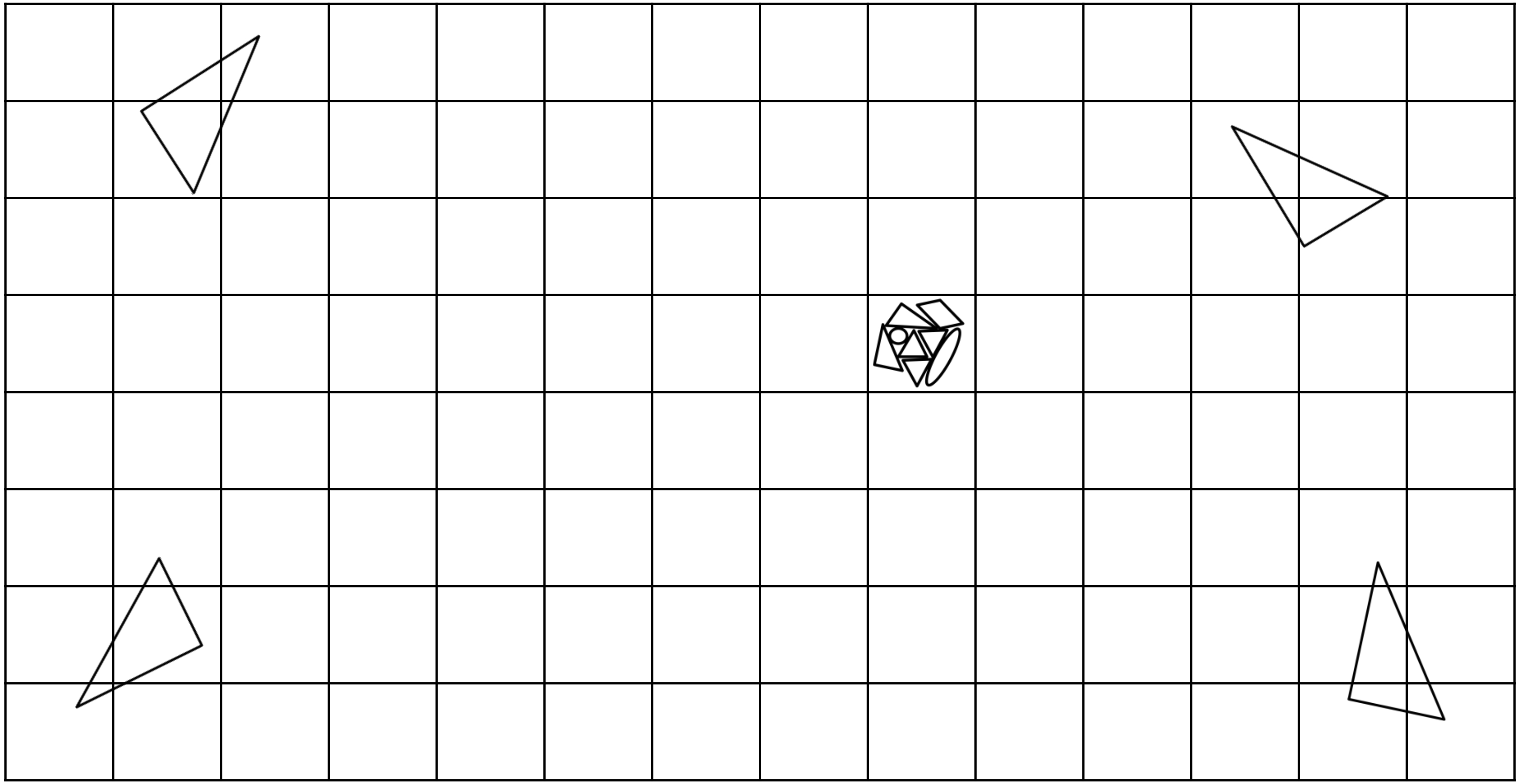
Using a grid

- March through the grid cells along the ray
- Intersect geometries within
  - Same object can be in multiple cells
  - Cache intersections per-ray

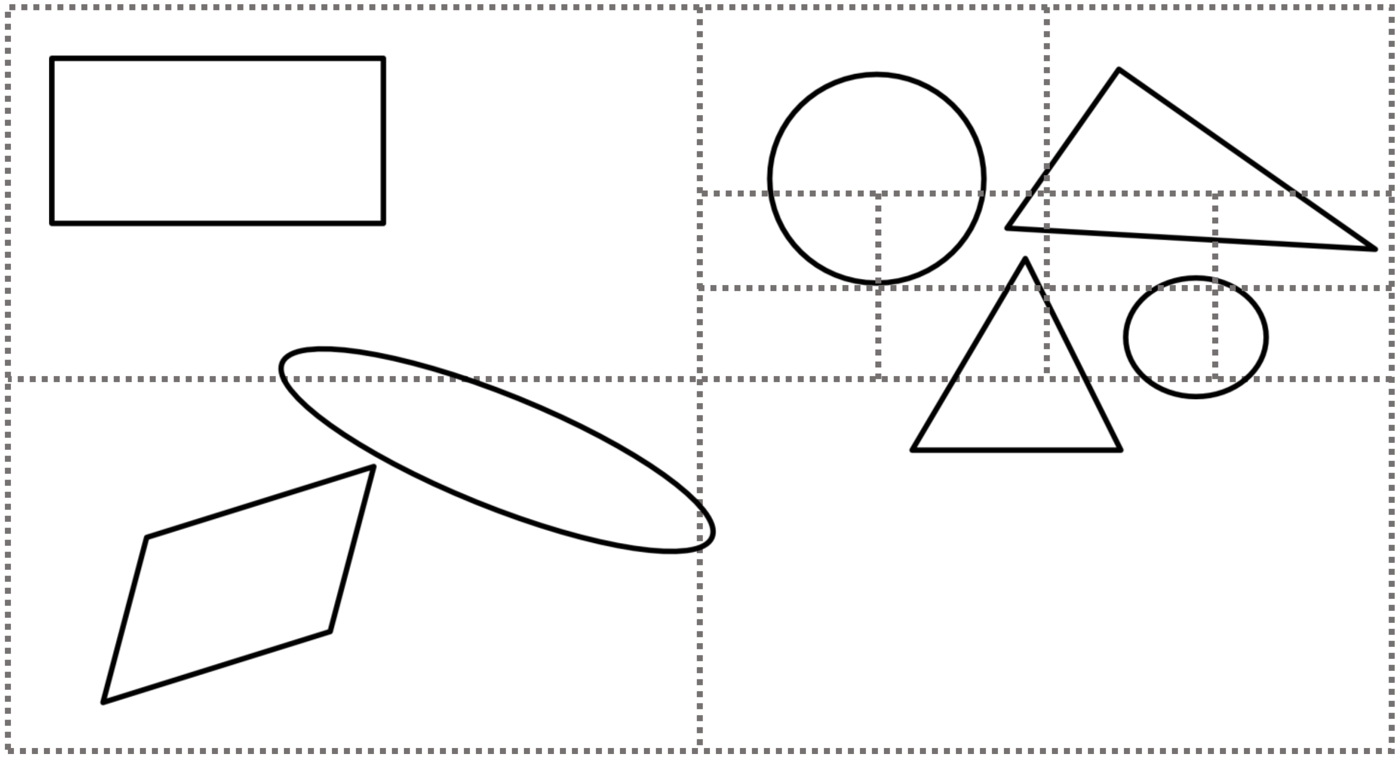


Grids are easy, but not very adaptive

- Often called the “teapot in a stadium” problem

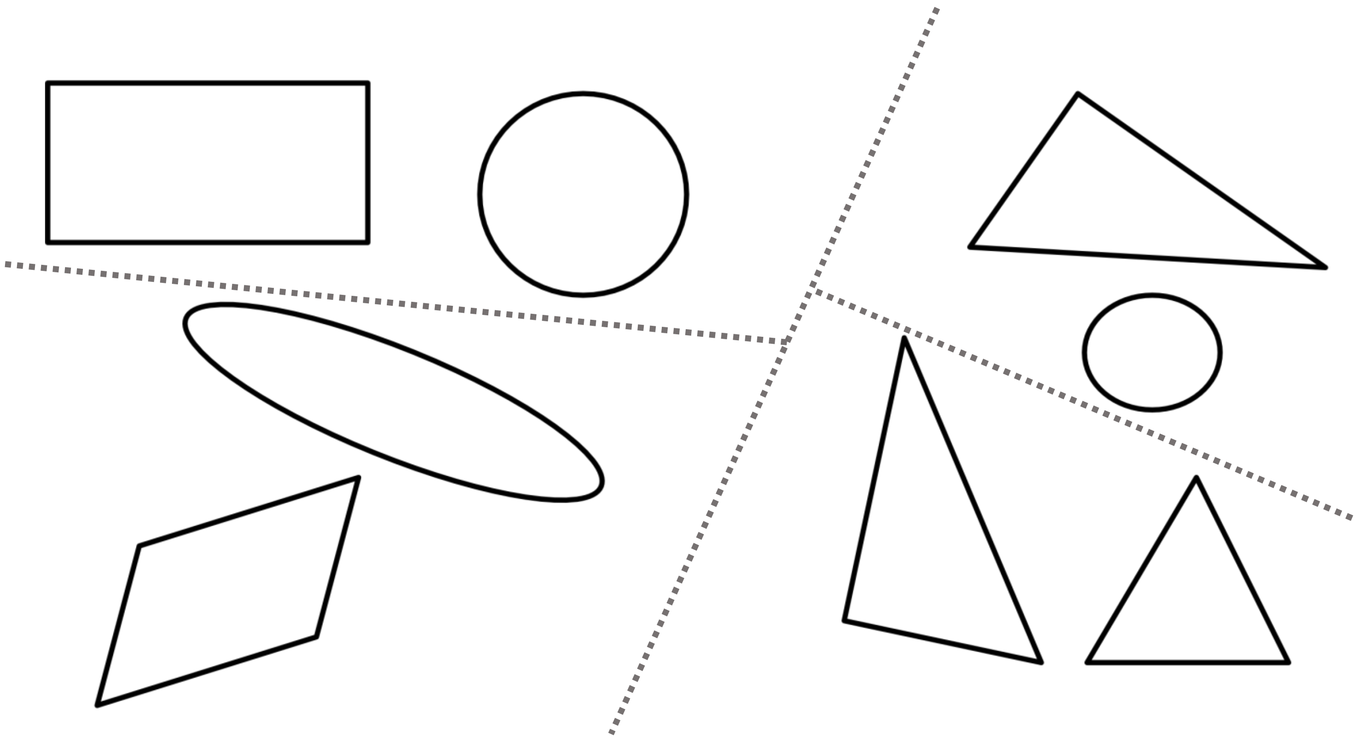


We can adapt the resolution locally by using, e.g., an octree

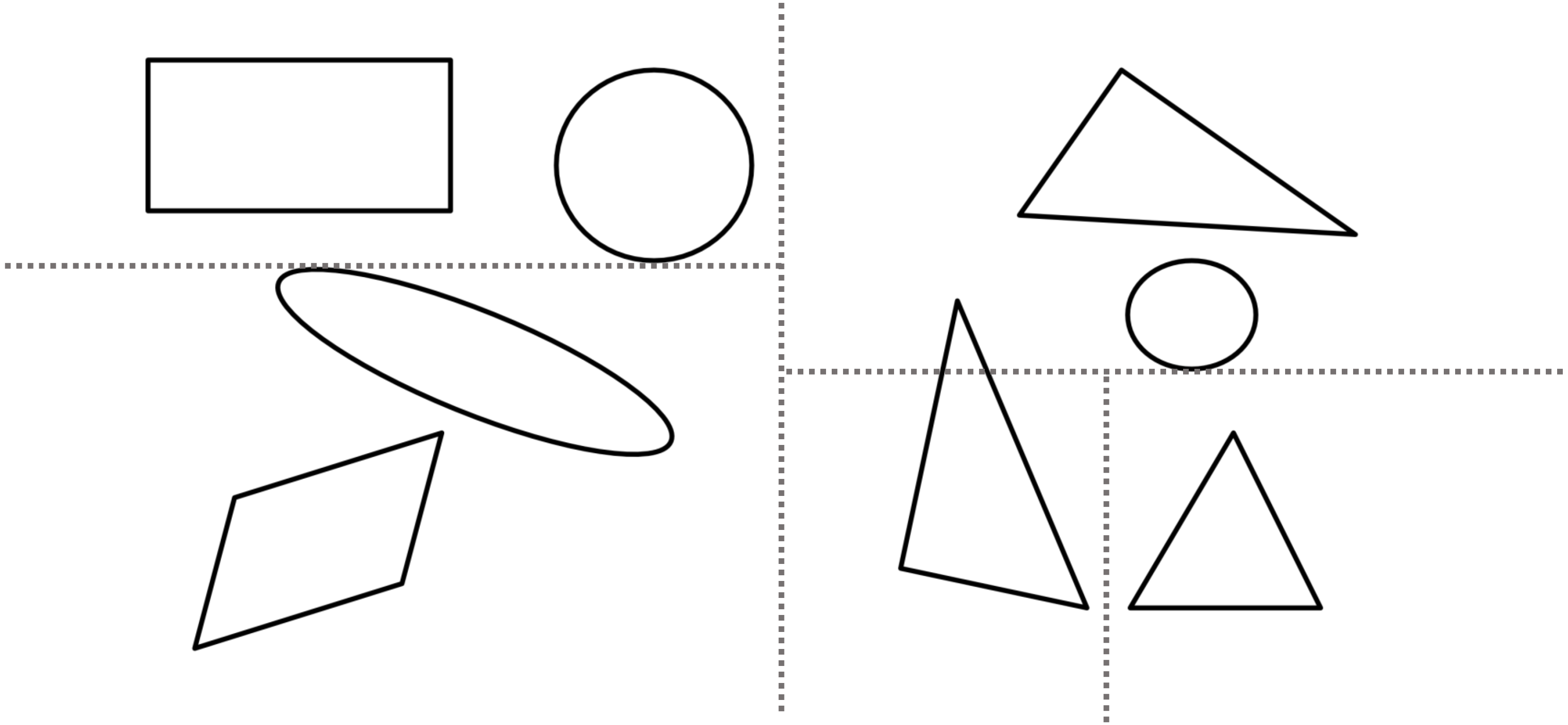


Binary space partitioning (BSP) – (e.g., used by Doom)

- Split recursively using planes
- Arbitrary split position
- Arbitrary split orientation

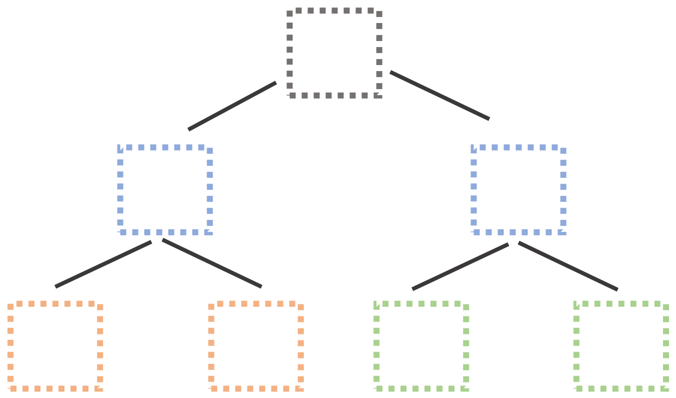


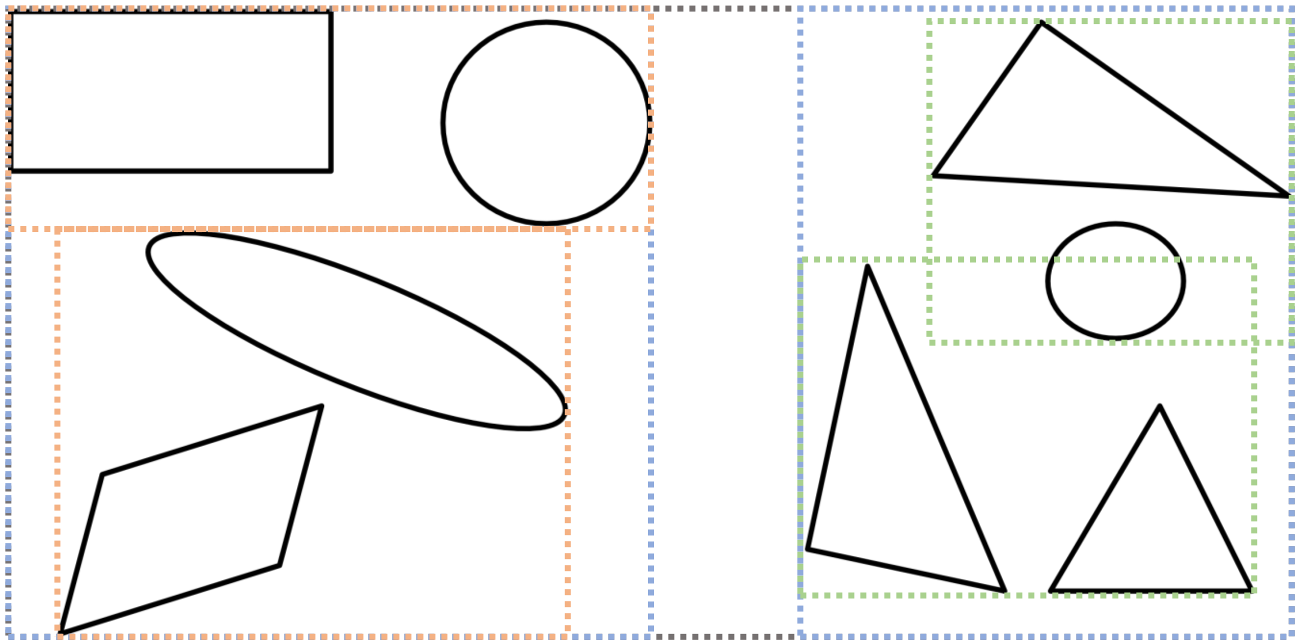
kd-Trees (axis-aligned BSP)



Grouping objects via a Bounding volume hierarchy (BVH)

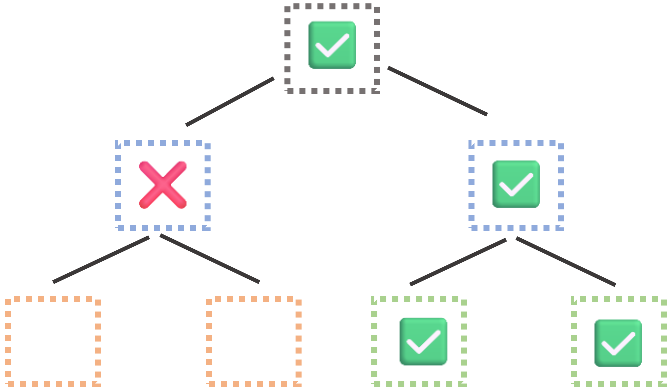
- Unlike spatial subdivision: bounding boxes can overlap

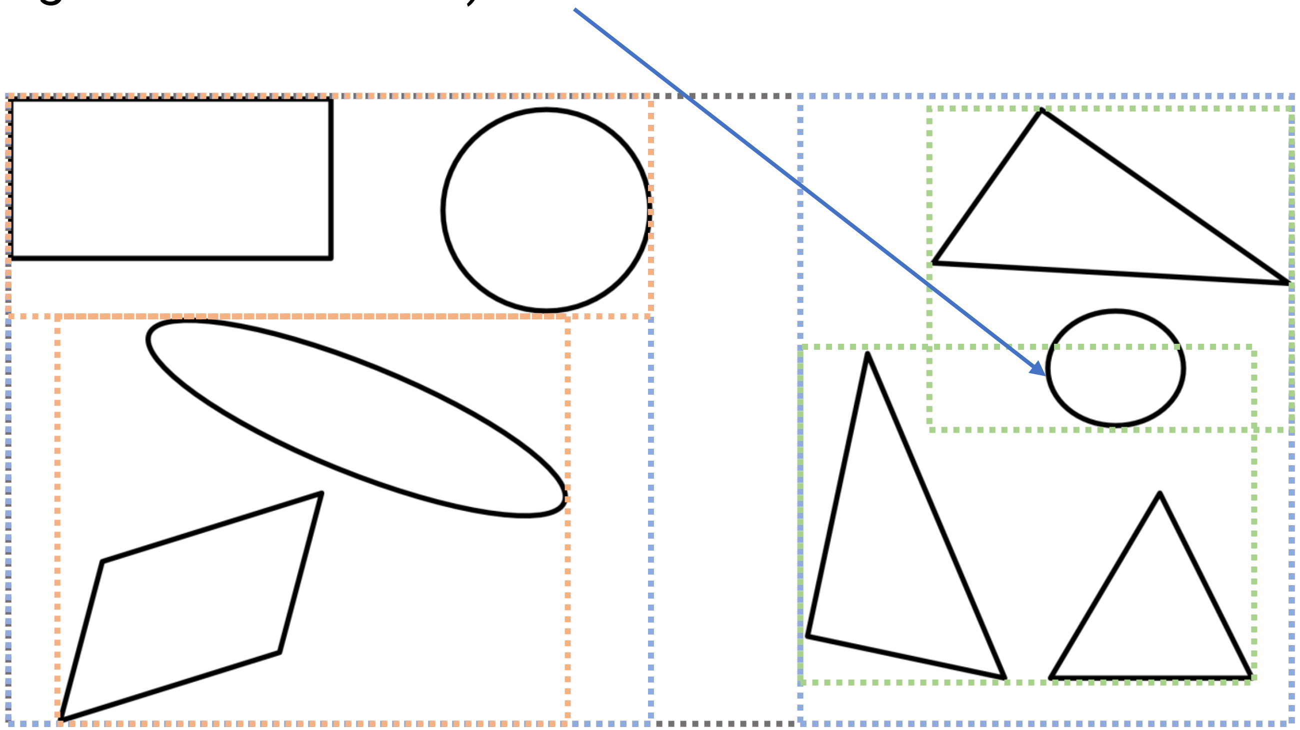




Traversing a BVH

- Intersect ray with bounding box
- If hit: Recursively visit children (best starting with the closest!)
- Cannot stop on first hit!

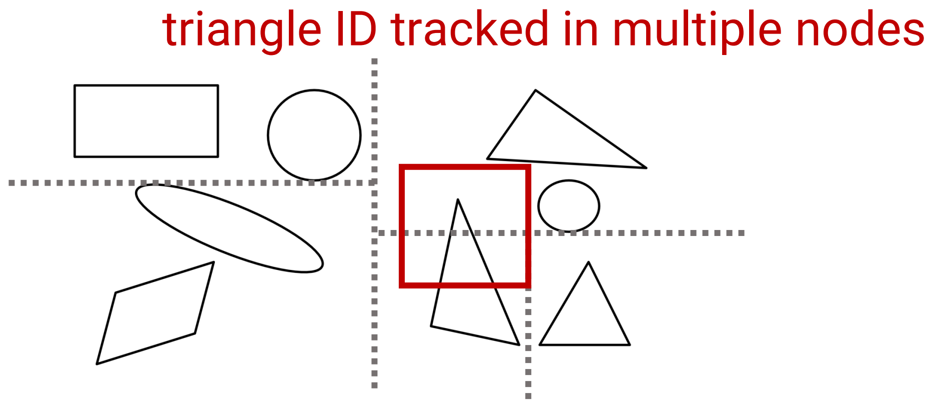




BVH vs kd-Tree

- KD trees require fewer intersection tests
  - kd: Terminate on first hit
  - BVH: First hit might be further away than others, due to spatial overlap

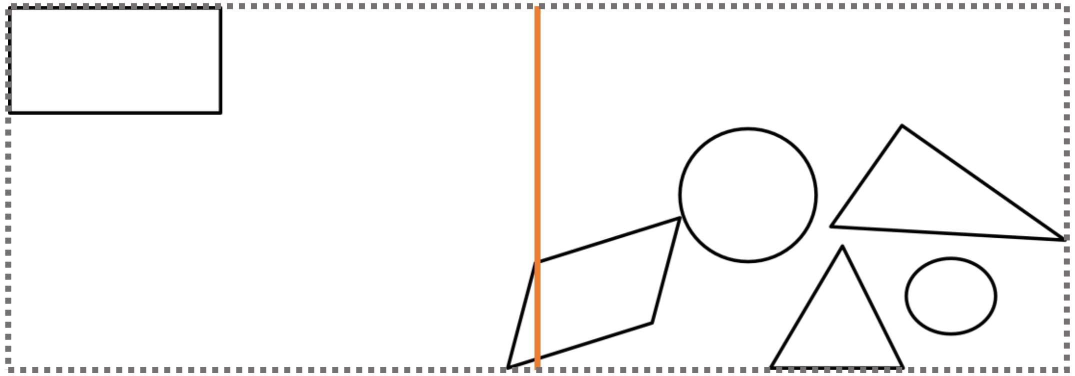
- BVH has faster build time and lower memory cost
  - Each triangle is in exactly one leaf node
    ➔ Fixed storage cost

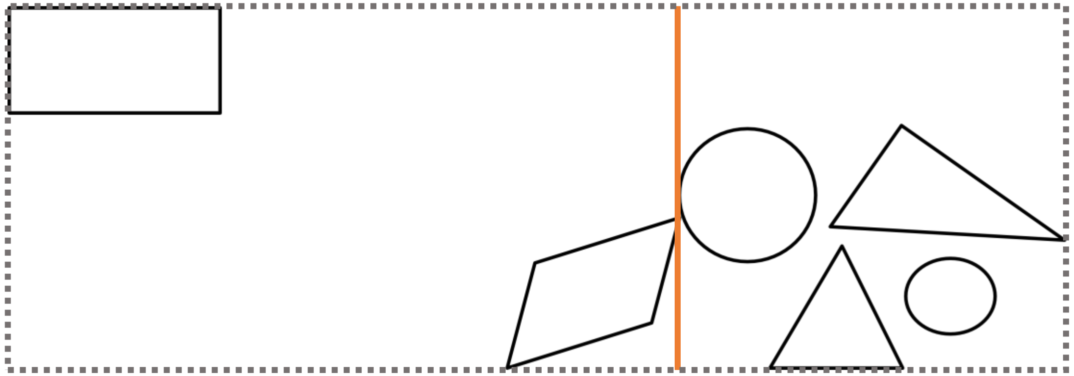


- BVHs do not have to be binary
  - E.g., 4ary BVH for SIMD on the CPU

Where to split?

- With BSP trees, kd-Trees, and BVHs alike, quality hinges on splitting locations

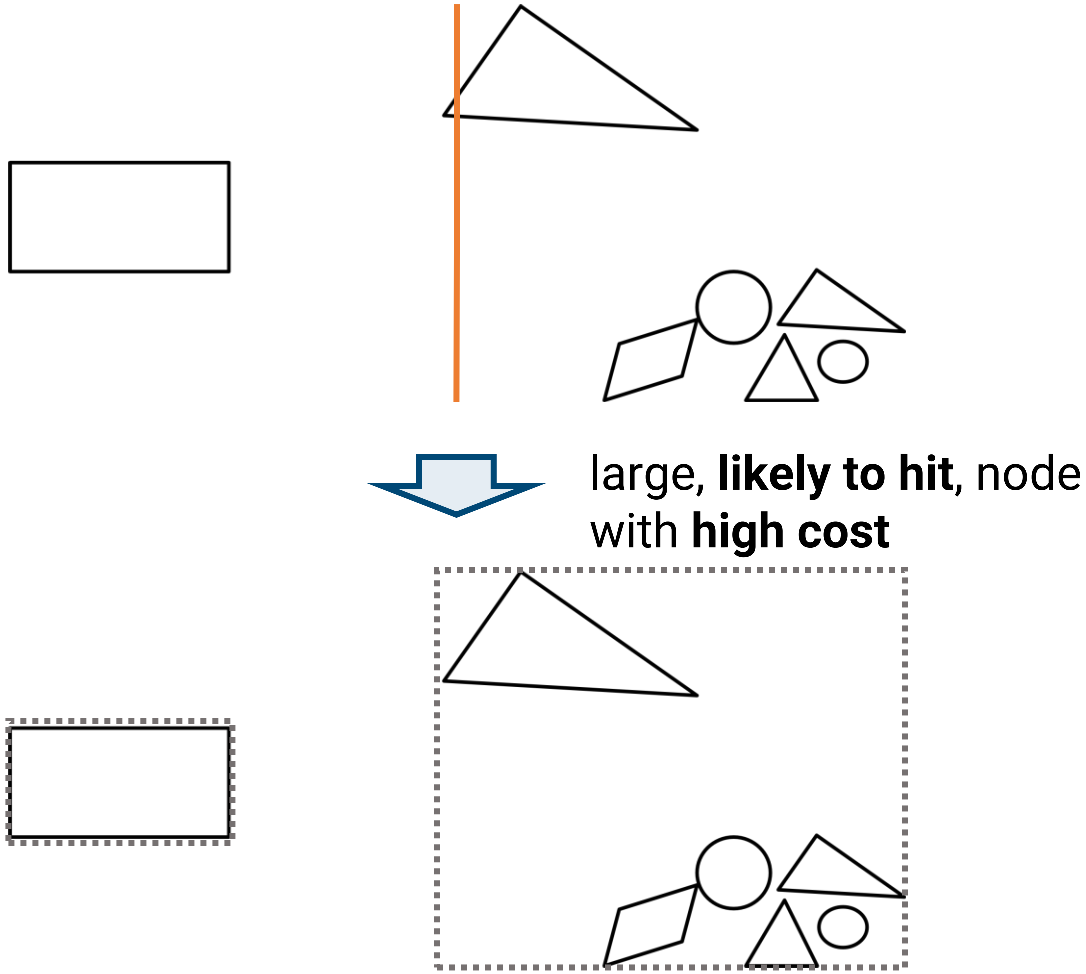




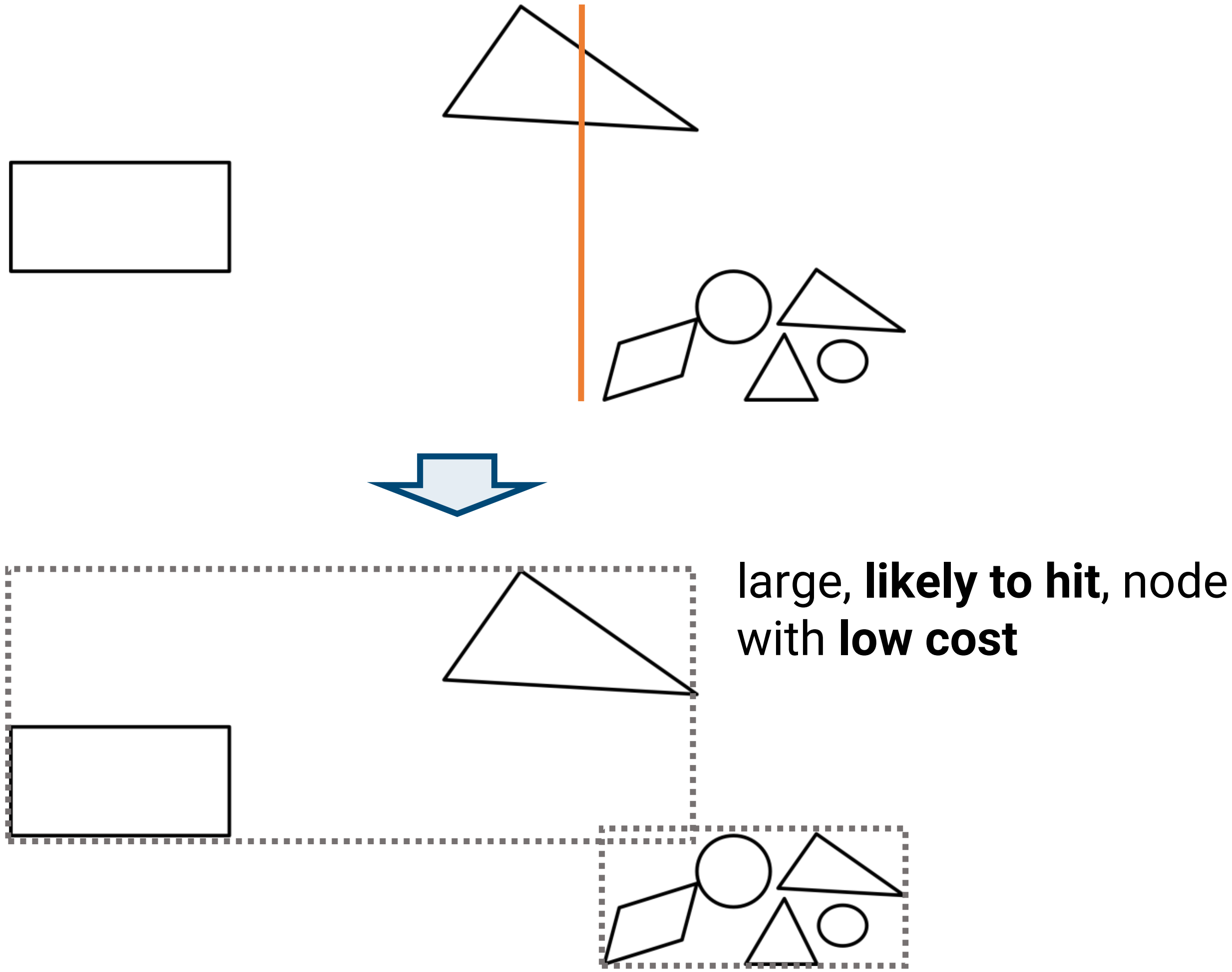
Split in the middle:
Produces large node with high intersection cost

Where to split?

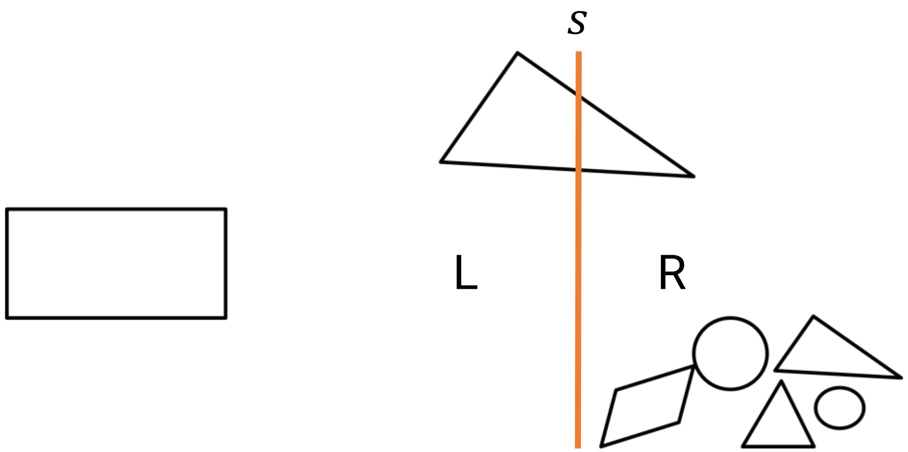
In the middle



Optimized



Surface area heuristic (SAH)



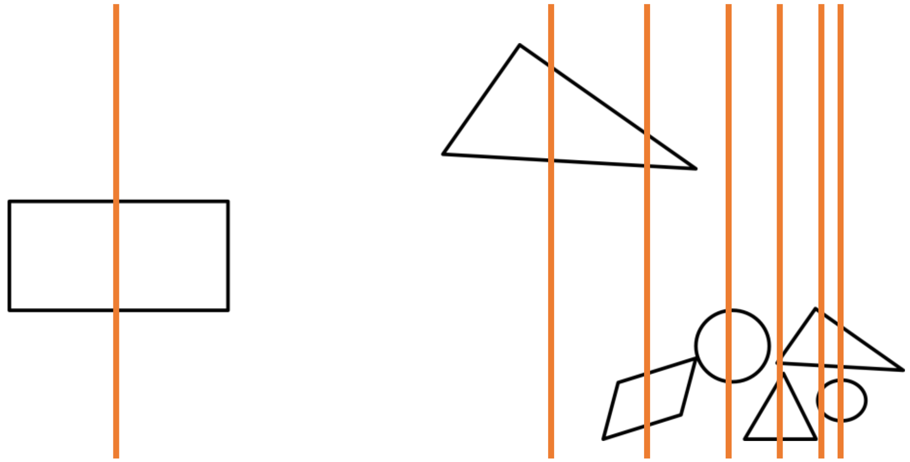
$$C(s) = C_t + P(L) \cdot C(L) + P(R) \cdot C(R)$$

C_t: Cost of traversal (AABB intersection etc.; implementation specific parameter)

$P(L)$: Probability to hit the left child
Given by ratio of *surface area* (for uniform rays) $P(L) = \frac{A(L)}{A(L+R)}$

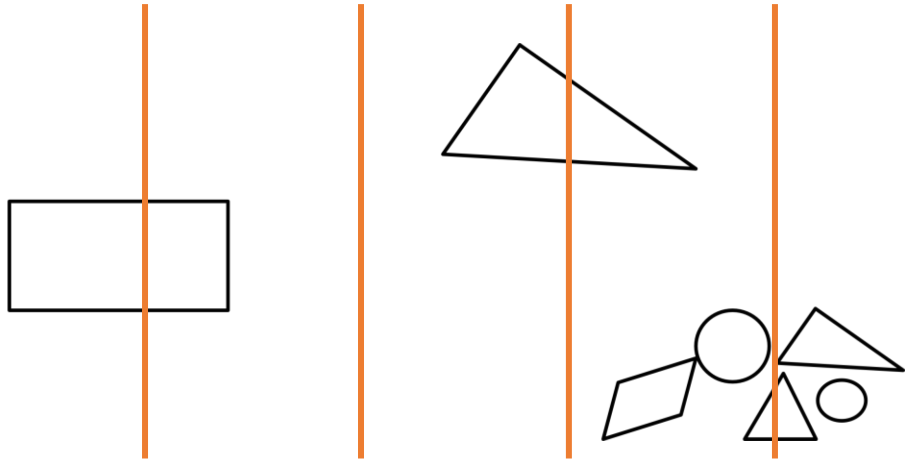
$C(L)$: Cost of intersecting the left child (usually set to the number of triangles)

The minima of the SAH cost occur where the primitive assignment changes



- SAH Computation is $O(n^2)$: n split candidates, each require n operations to compute cost

Binning: use fixed number of regularly spaced candidate positions



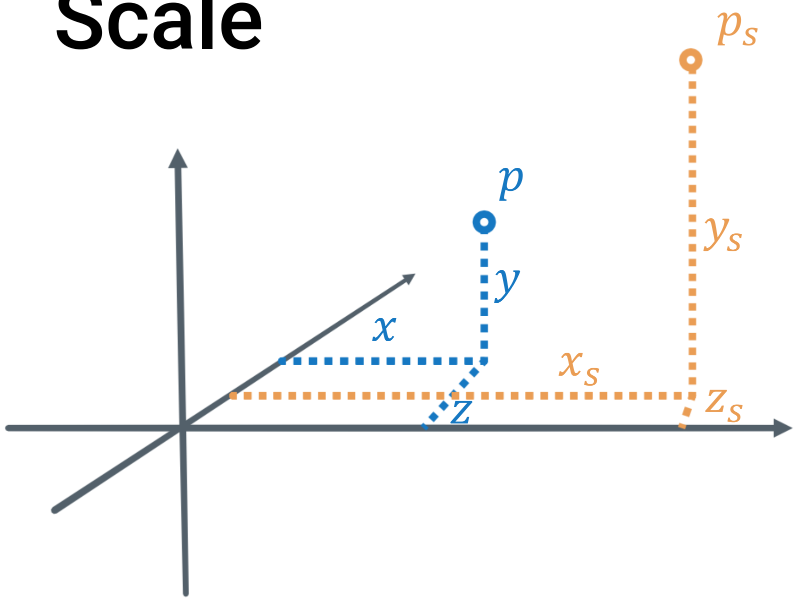
- SAH Computation is $O(kn)$
- Scales much better; often similar quality

Further reading

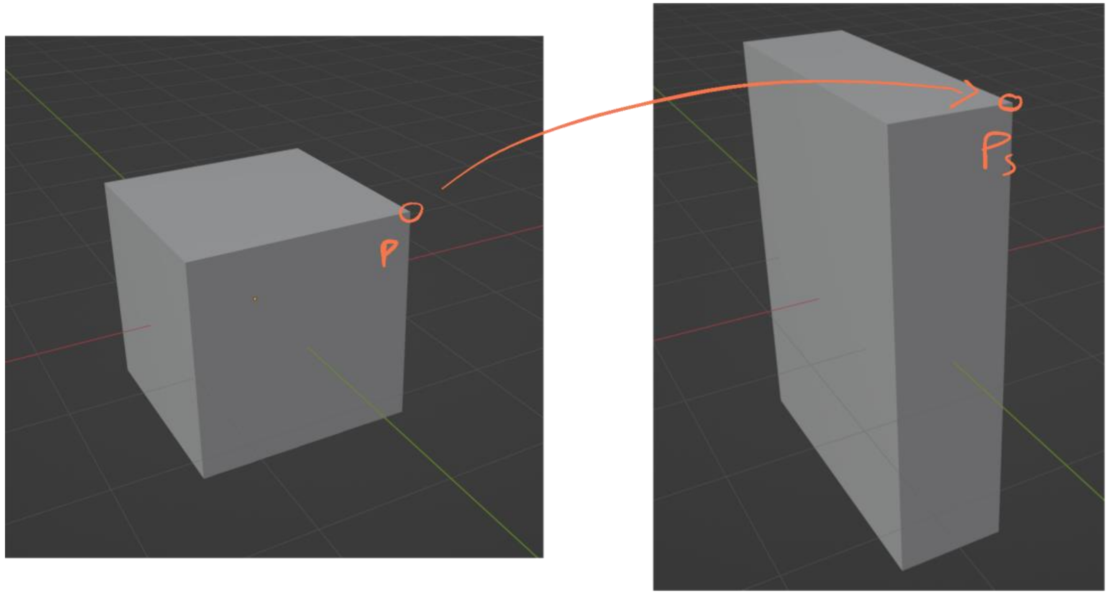
- https://www.pbr-book.org/4ed/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies
- https://jacco.ompf2.com/2022/04/13/how-to-build-a-bvh-part-1-basics/
- https://www.youtube.com/watch?v=C1H4zIiCOaI
- Vinkler et al. 2016. Performance Comparison of Bounding Volume Hierarchies and Kd-Trees for GPU Ray Tracing. Comput. Graph. Forum.
- Meister et al. 2021. A Survey on Bounding Volume Hierarchies for Ray Tracing. Comput. Graph. Forum.

Transformations

Scale



$$x_s = s_x x$$
$$y_s = s_y y$$
$$z_s = s_z z$$



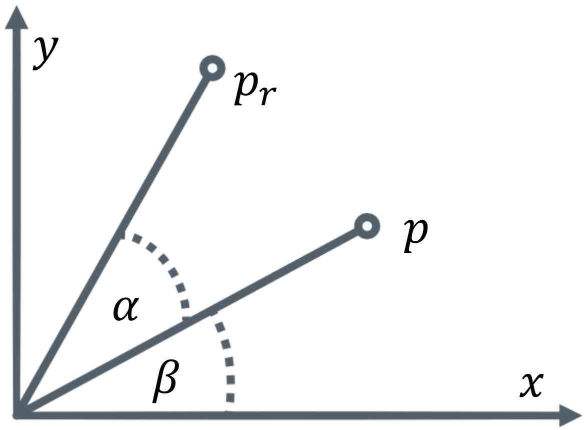
Can be written as a matrix – vector product:

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x_s \\ y_s \\ z_s \end{pmatrix}$$

Wait, but why matrices?

- Compact & clean
- Uniform treatment of all types of transformations
- Allows us to easily *combine* transformations
- Want to reverse a transformation? Use the inverse matrix!

Rotation around the z axis



Basic trigonometry:

$$x = \cos\beta \qquad x_r = \cos(\beta + \alpha)$$
$$y = \sin\beta \qquad y_r = \sin(\beta + \alpha)$$

More basic trigonometry (angle sum identity):

$$\boldsymbol{x_r} = \cos(\beta + \alpha) = \cos\beta\cos\alpha - \sin\beta\sin\alpha = \boldsymbol{x\cos\alpha - y\sin\alpha}$$
$$\boldsymbol{y_r} = \sin(\beta + \alpha) = \cos\beta\sin\alpha + \sin\beta\, cos\,\alpha = \boldsymbol{x\sin\alpha + y\cos\alpha}$$

In matrix form:

$$\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Rotation around x and y can be derived analogously

Translation

- Easy:
  - $x_t = x + a$
  - $y_t = y + b$
  - $z_t = z + c$
- How to write as a matrix? $\begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$
- Be mindful when transforming *directions*: $\begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}$

Transformations can be combined via matrix multiplication

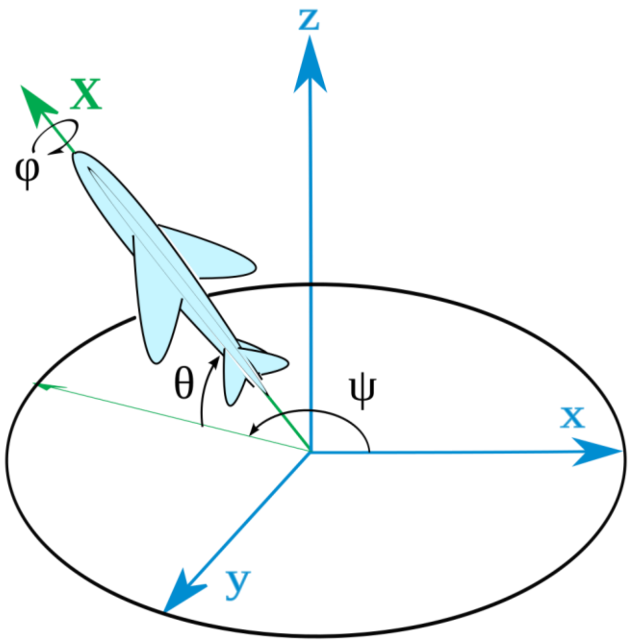
- Order matters! Generally, first scale (M_s), then rotate (M_r), then translate (M_t)

$$M = M_t M_r M_s$$

- Rotation is often expressed via Euler angles (yaw, pitch, roll)

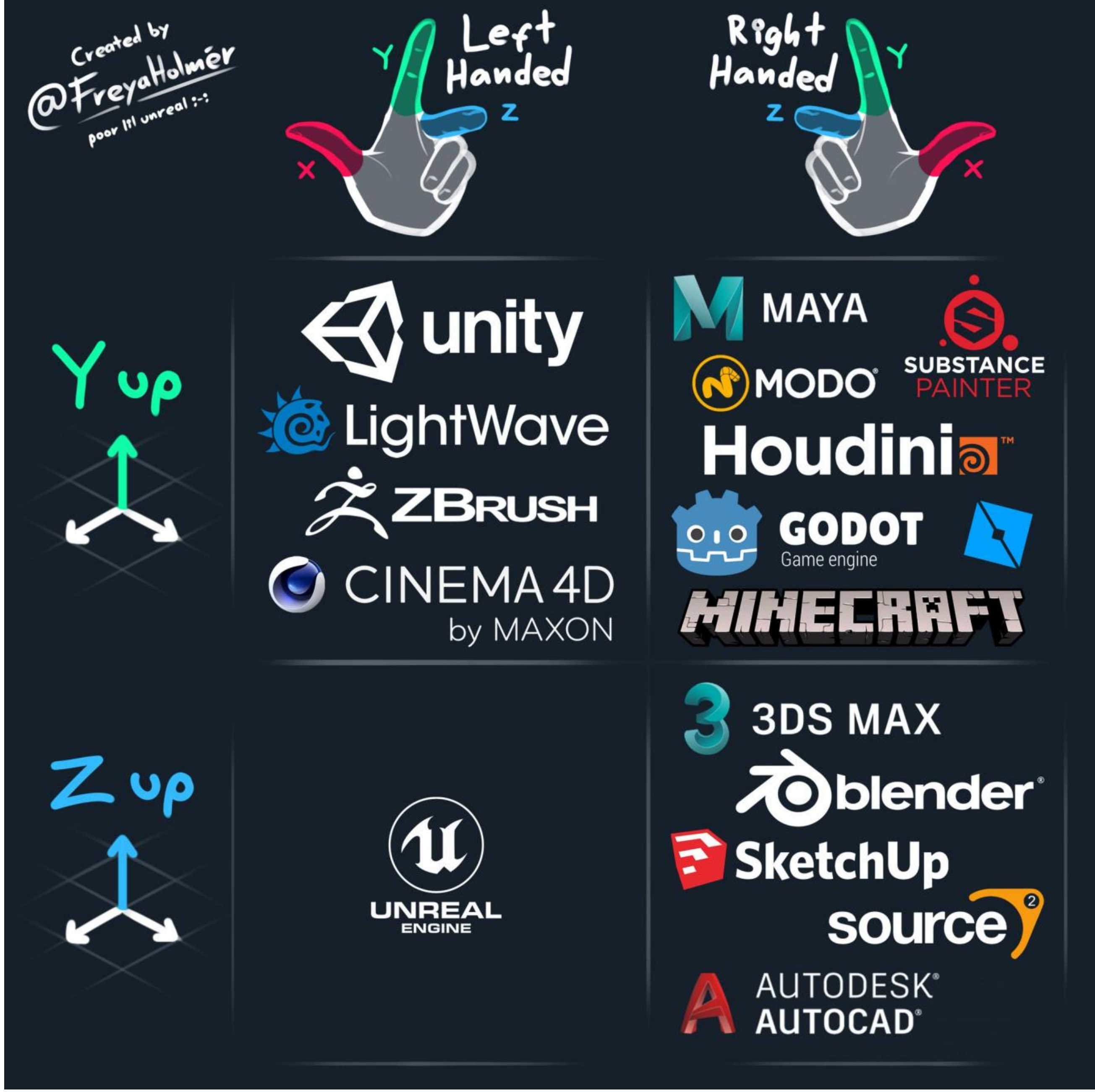
$$M_r = M_{yaw} M_{pitch} M_{roll}$$

  - Each corresponds to a rotation around one axis
  - Which axis? Depends on the coordinate system convention!
  - (The *order* of rotation is also up to convention...)

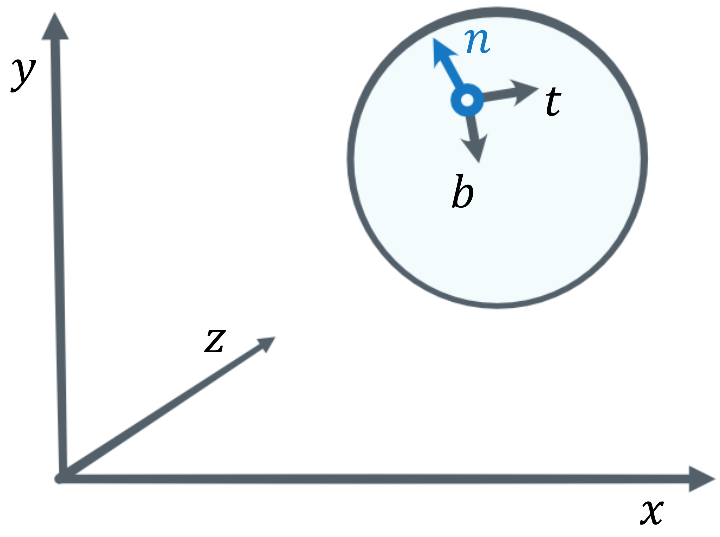


Coordinate systems

- Always important to define conventions...
- ... and keep them in mind



Moving to a new coordinate system



- Example: world space and shading space
- Why?
  - Shading is convenient if coordinate system aligned with normal
- How?
  - Construct orthonormal basis (n, t, b) from normal, tangent, and bitangent
  - If we set n to be the z axis and t and b to be x and y respectively:

$$p' = tx + by + nz$$

  - In matrix form:

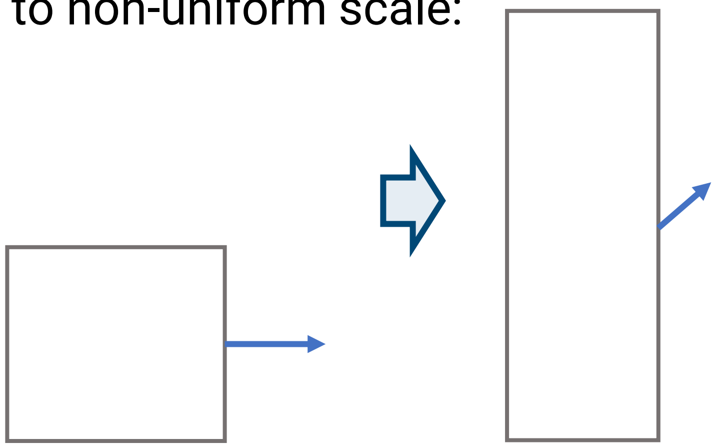
$$p' = \begin{pmatrix} t_x & b_x & n_x \\ t_y & b_y & n_y \\ t_z & b_z & n_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

To invert, we can use the fact that (n, t, b) are orthonormal:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{pmatrix} p'$$

Transforming normals

- Normals do not remain normalized nor perpendicular when transformed naively
  - e.g., due to non-uniform scale:



- Solutions:
  1. Recompute the normal based on the transformed tangent vectors
  2. Transform with the *inverse transpose*: $\mathrm{n}' = (M^{-1})^T n$
     - (see https://www.pbr-book.org/4ed/Geometry_and_Transformations/Applying_Transformations#Normals)

Further reading

- Eric Lengyel. Mathematics for 3D Game Programming and Computer Graphics. 2011.
- https://www.pbr-book.org/4ed/Geometry_and_Transformations/Transformations

- https://www.3blue1brown.com/lessons/linear-transformations
- https://www.3blue1brown.com/lessons/3d-transformations
- https://www.scratchapixel.com/lessons/3d-basic-rendering/transforming-objects-using-matrices/using-4x4-matrices-transform-objects-3D.html

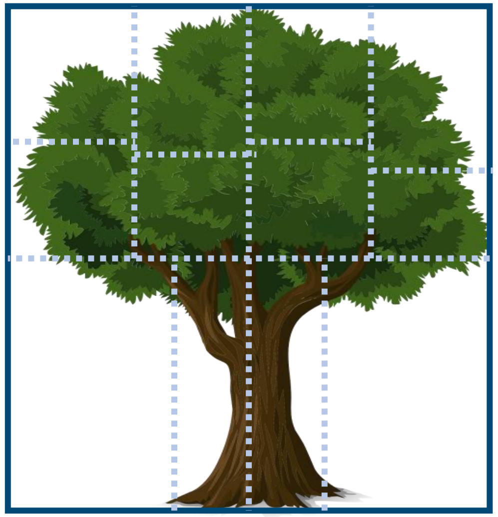
Instancing

We often have multiple copies (instances) of the same object

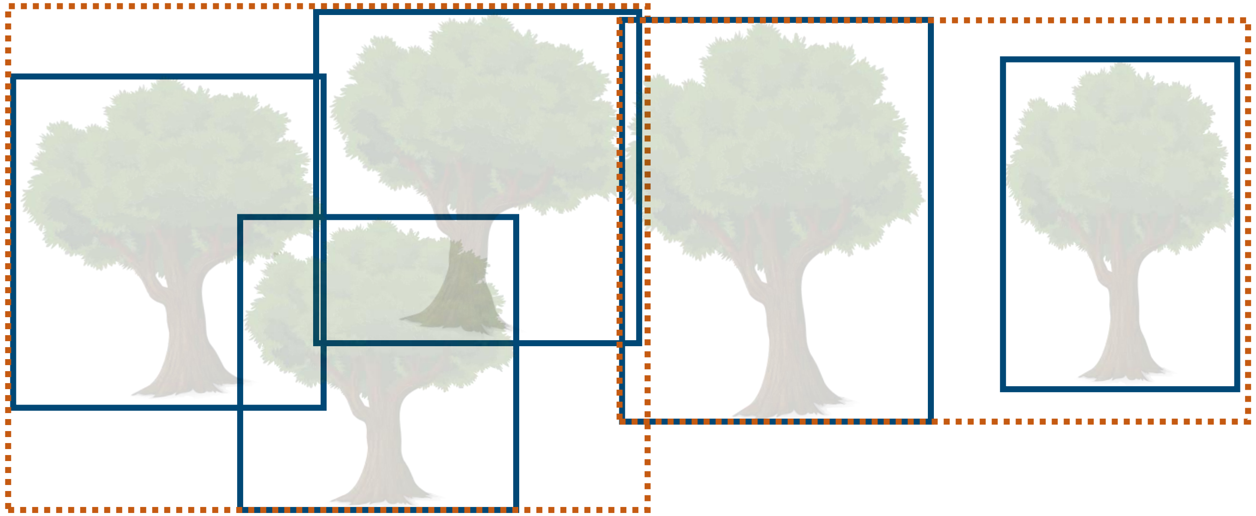
- No need to store all triangles of all these copies!
- Just track a list of *transformation matrices* per object

Bottom-level and top-level acceleration structures

- Bottom-level (often called “BLAS” by real-time folks) contains triangles of one mesh
- Top-level (often called “TLAS” by real-time folks) contains transformed AABBs of all instances



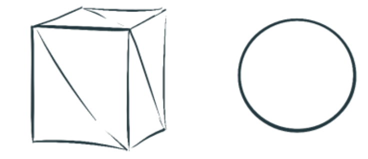
Bottom-level



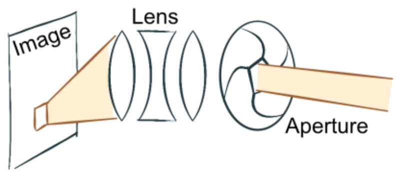
Top-level

Summary

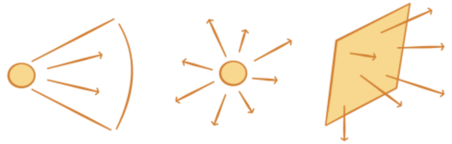
Now you can visualize geometries with a camera!



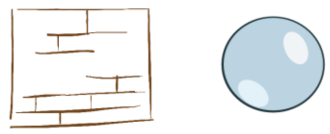
Geometries



Camera model

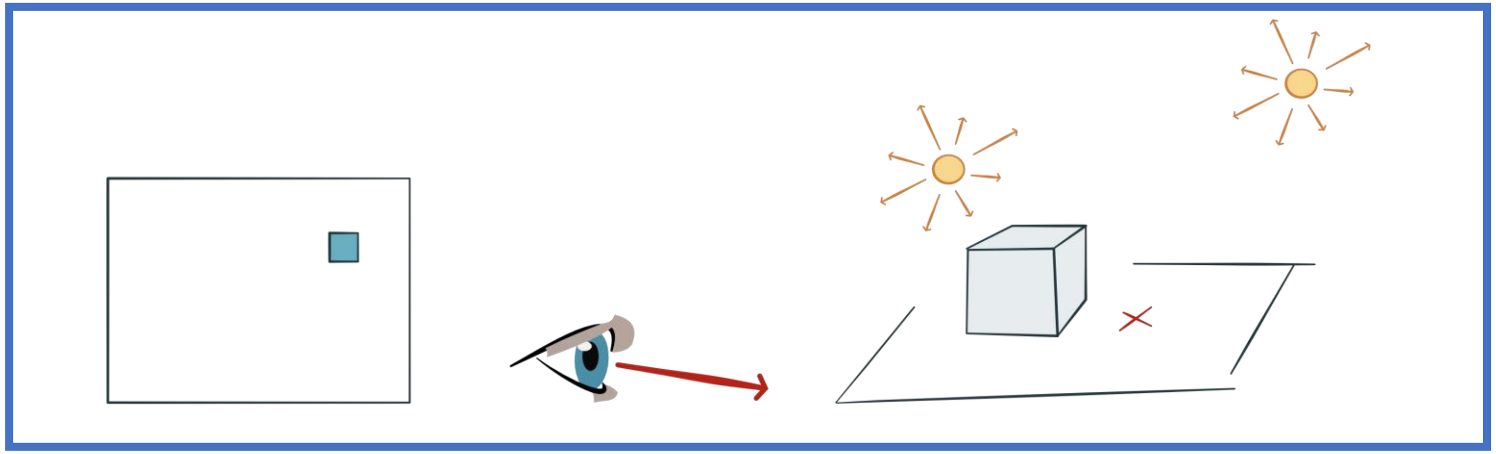


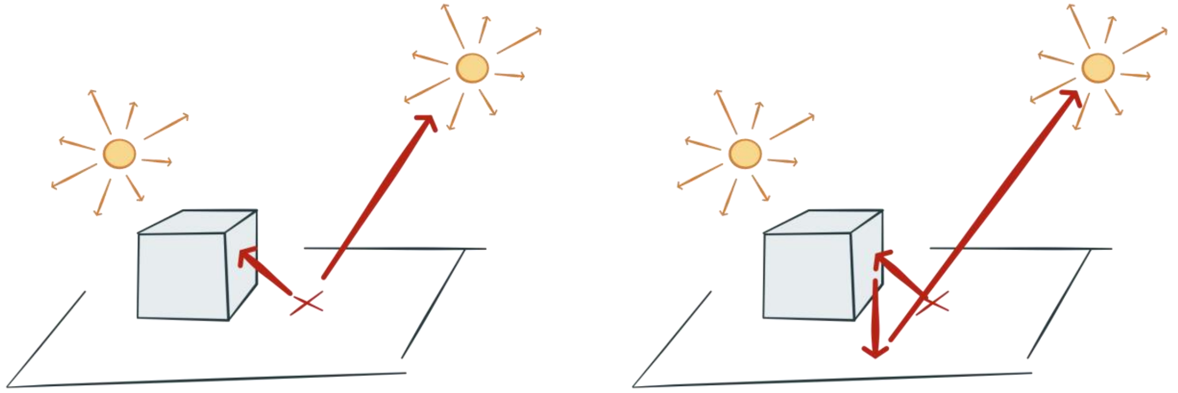
Light sources



Materials & Textures

Volumes





Don’t forget to register for a tutorial group on Teams **today**

Please make sure you have a University GitLab account **today**

Log in to: https://gitlab.cs.uni-saarland.de

Issues? https://sam.sic.saarland/

Looking for a teammate?

Suggestion: Meet up at the front now and see if you find one