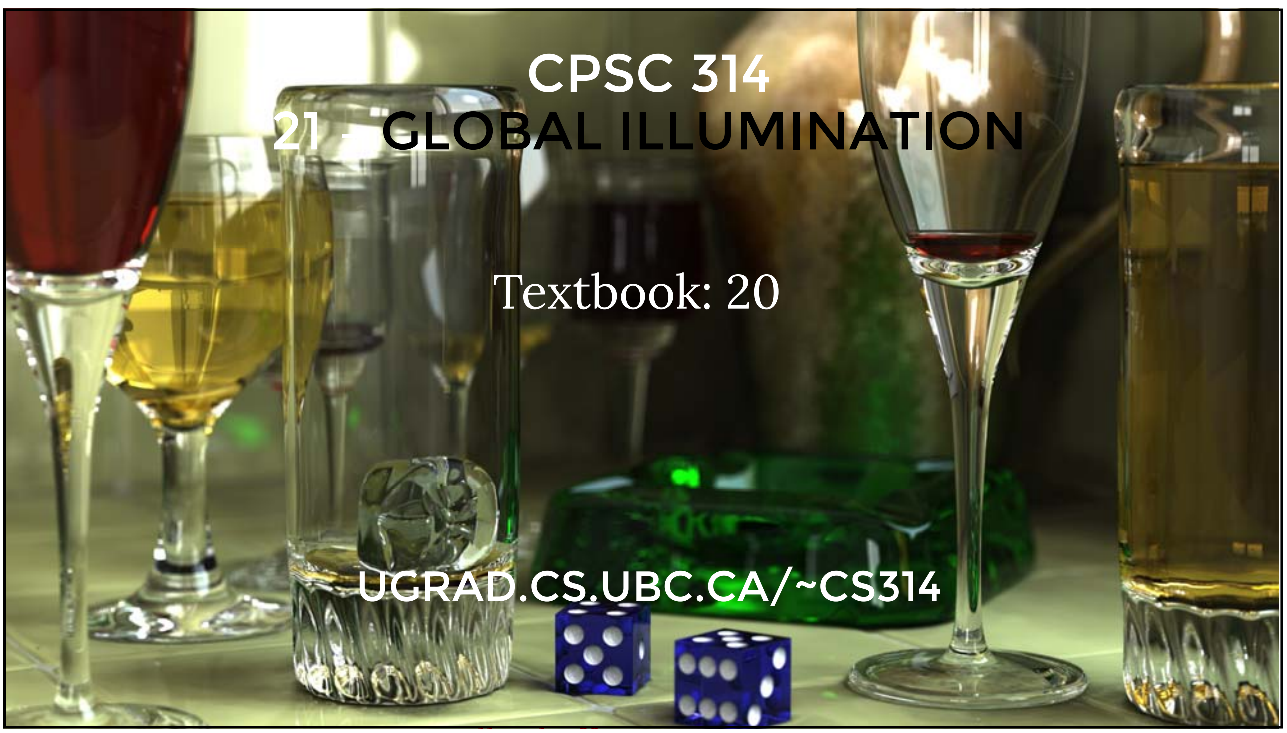
MIDTERM 2

- Viewing/Projections (orthographic, perspective)
- Clipping
- Rasterization
  - Scan conversion
  - Interpolation
- Lighting and shading
- Texturing. Bump/displacement/environment mapping.
- Shadow maps
- Depth test
- … and don't forget everything we learned before Midterm 1



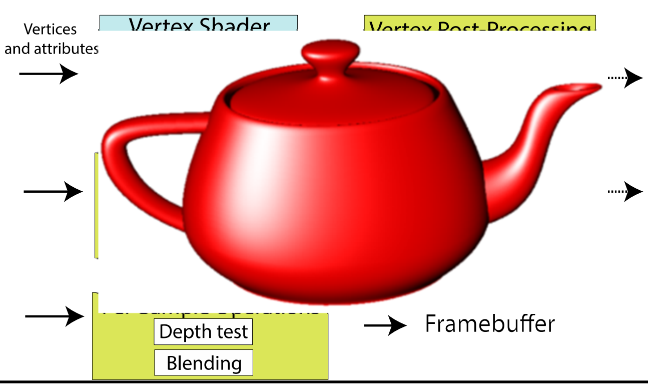
ILLUMINATION MODELS/ALGORITHMS

Local illumination - Fast
Ignore real physics, approximate the look
Interaction of each object with light
- Compute on surface (light to viewer)

Global illumination – Slow
Physically based
Interactions between objects

ILLUMINATION MODELS/ALGORITHMS

Local illumination - Fast
Ignore real physics, approximate the look
Interaction of each object with light
- Compute on surface (light to viewer)

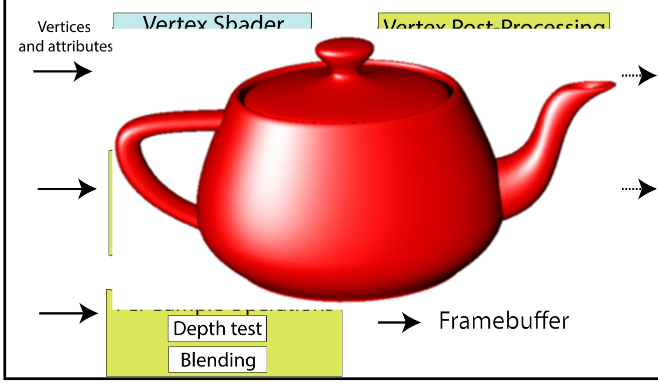


Global illumination – Slow
Physically based
Interactions between objects

ILLUMINATION MODELS/ALGORITHMS

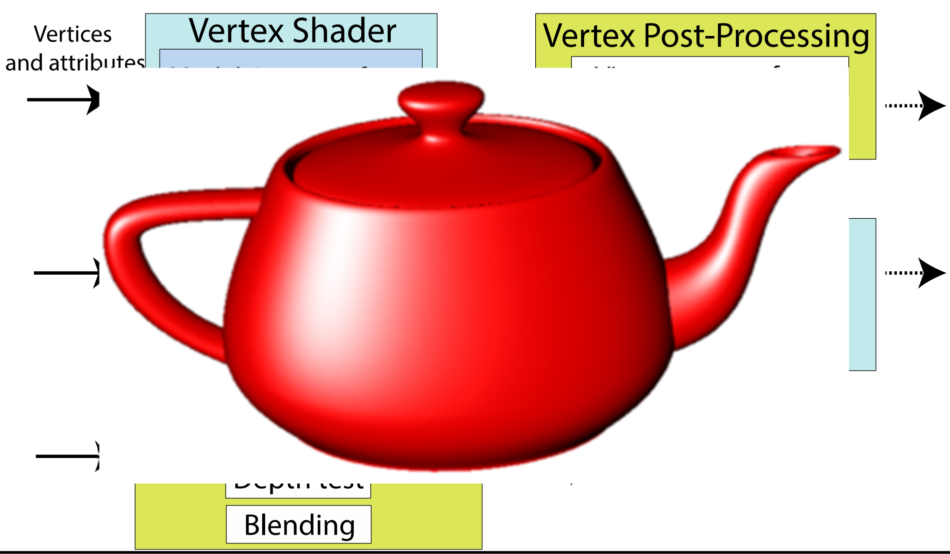
Local illumination - Fast
Ignore real physics, approximate the look
Interaction of each object with light

- Compute on surface (light to viewer)



Global illumination – Slow
Physically based
Interactions between objects

WHAT WAS NON-PHYSICAL IN LOCAL ILLUMINATION?



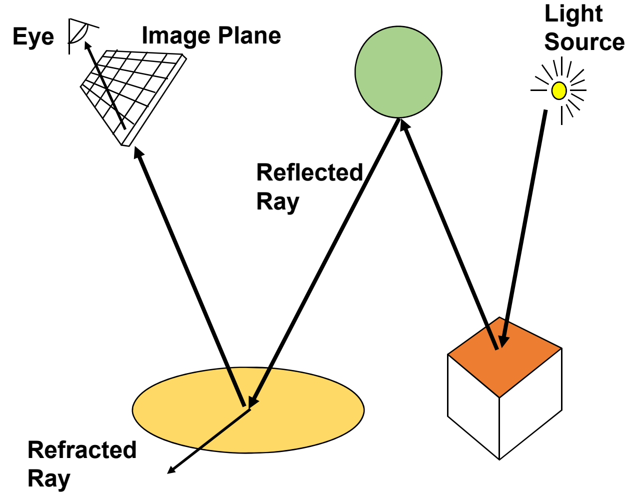
GLOBAL ILLUMINATION ALGORITHMS

- Ray Tracing
- Path Tracing
- Photon Mapping
- Radiosity
- Metropolis light transport
- …

HOW SHOULD GLOBAL ILLUMINATION WORK?

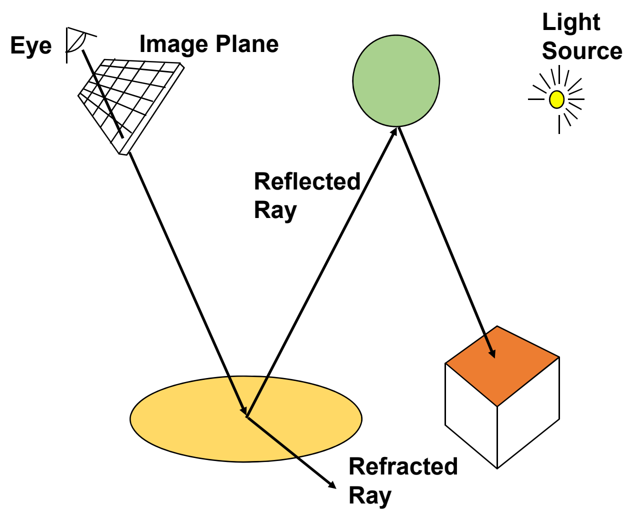
HOW SHOULD GLOBAL ILLUMINATION WORK?

Simulate light
- As it is emitted from light sources
- As it bounces off objects / get absorbed / refracted
- As some of the rays hit the camera/eye

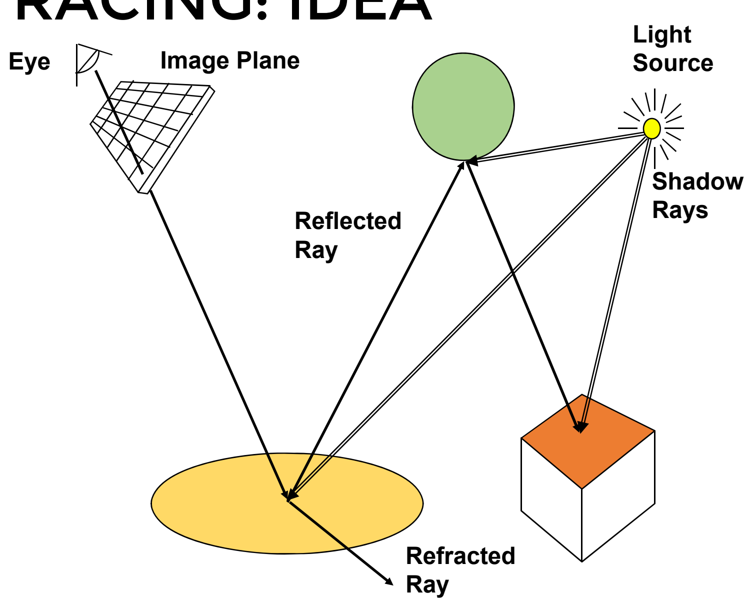


PROBLEM?

RAY TRACING: IDEA



RAY TRACING: IDEA

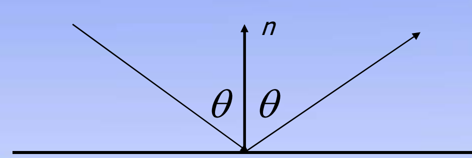


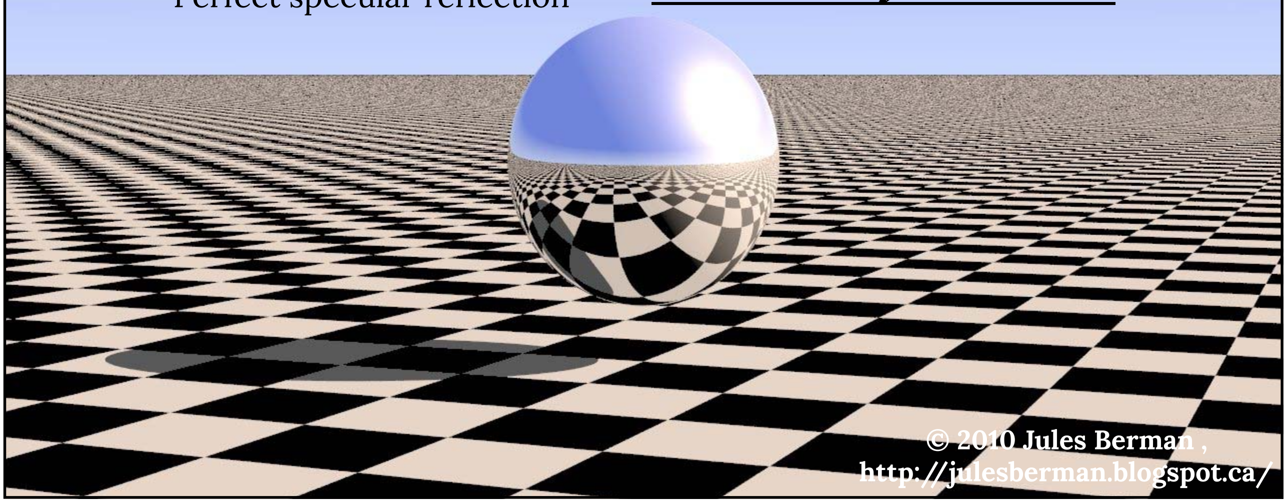
RAY TRACING

- Invert the direction of rays!
- Shoot rays from CAMERA through each pixel
  - "Trace the rays back"
- Simulate whatever the light rays do:
  - Reflection
  - Refraction
  - …
- Each interaction of the ray with an object adds to the final color
- Those rays are never gonna hit the light source, so
  - Shoot "shadow rays" to compute direct illumination

REFLECTION

- Mirror effects
  - Perfect specular reflection



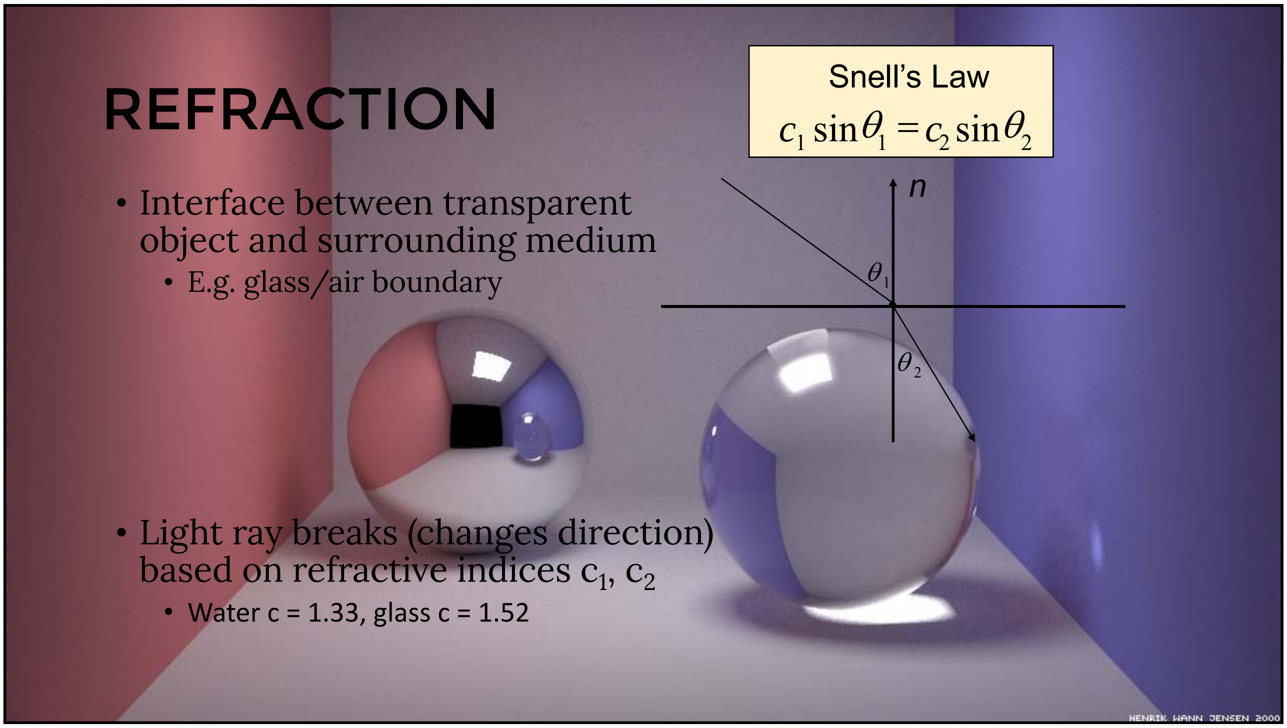


REFRACTION

Snell's Law

$$c_1 \sin\theta_1 = c_2 \sin\theta_2$$

- Interface between transparent object and surrounding medium
  - E.g. glass/air boundary



- Light ray breaks (changes direction) based on refractive indices c_1, c_2
  - Water c = 1.33, glass c = 1.52

BASIC RAY-TRACING ALGORITHM

```
RayTrace(r,scene)
obj = FirstIntersection(r,scene)

if (no obj) return BackgroundColor;
else {
    if (Reflect(obj))
       reflect_color = RayTrace(ReflectRay(r,obj));
    else
       reflect_color = Black;

   if (Transparent(obj))
       refract_color = RayTrace(RefractRay(r,obj));
    else
       refract_color = Black;

   return Shade(reflect_color, refract_color, obj);
}
```

ONE BIG BUG....WHERE?

```
RayTrace(r,scene)
obj = FirstIntersection(r,scene)

if (no obj) return BackgroundColor;
else {
    if (Reflect(obj))
       reflect_color = RayTrace(ReflectRay(r,obj));
    else
       reflect_color = Black;

   if (Transparent(obj))
       refract_color = RayTrace(RefractRay(r,obj));
    else
       refract_color = Black;

   return Shade(reflect_color, refract_color, obj);
}
```

WHEN TO STOP?

- Algorithm above does not terminate…
- Termination Criteria
  - No intersection
  - Contribution of secondary ray attenuated below threshold – each reflection/refraction attenuates ray
  - Maximal depth is reached

SUB-ROUTINES

- ReflectRay(r,obj) – computes reflected ray (use obj normal at intersection)
- RefractRay(r,obj) - computes refracted ray
  - Note: ray is inside obj
- Shade(reflect_color,refract_color,obj) – compute illumination given three components

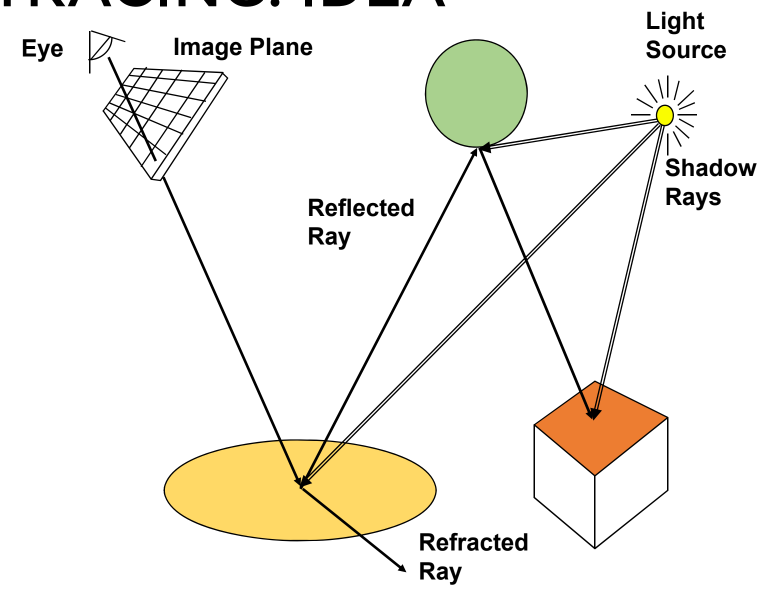
SIMULATING SHADOWS

- Trace ray from each ray-object intersection point to light sources
  - If the ray intersects an object in between ⇒ point is shadowed from the light source

```
shadow = RayTrace(LightRay(obj,r,light));

return Shade(shadow,reflect_color,refract_color,obj);
```

RAY TRACING: IDEA

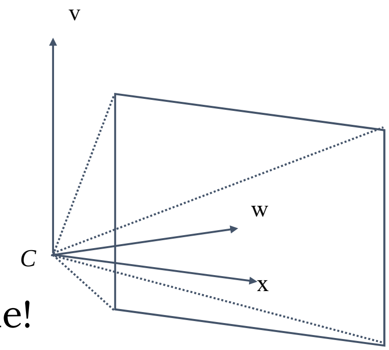


RAY-TRACING: PRACTICALITIES

- Generation of rays
- Intersection of rays with geometric primitives
- Geometric transformations
- Lighting and shading
- Speed: Reducing number of intersection tests
    - E.g. use BSP trees or other types of space partitioning

RAY-TRACING: GENERATION OF RAYS

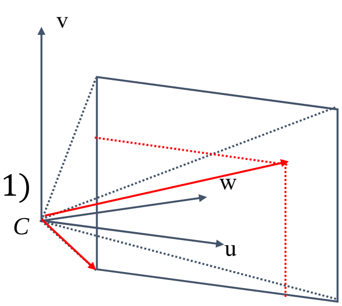
- Camera Coordinate System
  - Origin: C (camera position)
  - Viewing direction: **w**
  - Up vector: v
  - u direction: **u**= **w**×**v**



- Corresponds to viewing transformation in rendering pipeline!

RAY-TRACING: GENERATION OF RAYS

- Distance to image plane: d
- Image resolution (in pixels): N_x, N_y
- Image plane dimensions: l, r, t, b
- Pixel at position i, j $(i = 0, \dots, N_x - 1; j = 0, \dots, N_y - 1)$



$$\boldsymbol{O} = \boldsymbol{C} + d\vec{\boldsymbol{w}} + l\vec{\boldsymbol{u}} + t\vec{\boldsymbol{v}}$$

$$P_{i,j} = O + (i + 0.5) \cdot \frac{r - l}{N_x} \cdot \vec{u} - (j + 0.5) \cdot \frac{t - b}{N_y} \cdot \vec{v}$$

$$= O + (i + 0.5) \cdot \Delta u \cdot \vec{u} - (j + 0.5) \cdot \Delta v \cdot \vec{v}$$

RAY-TRACING: GENERATION OF RAYS

- Parametric equation of a ray:

$$R_{i,j}(t) = C + t \cdot (P_{i,j} - C) = C + t \cdot \mathbf{v}_{i,j}$$

where $t = 0 \ldots \infty$

RAY-TRACING: PRACTICALITIES

- Generation of rays
- **Intersection of rays with geometric primitives**
- Geometric transformations
- Lighting and shading
- Speed: Reducing number of intersection tests
  - E.g. use BSP trees or other types of space partitioning

RAY-OBJECT INTERSECTIONS

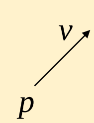
- In OpenGL pipeline, we were limited to discrete objects:
  - Triangle meshes
- In ray tracing, we can support analytic surfaces!
  - No problem with interpolating z and normals, # of triangles, etc.
    - Almost

RAY-OBJECT INTERSECTIONS

- Core of ray-tracing $\Rightarrow$ must be extremely efficient
- Usually involves solving a set of equations
  - Using implicit formulas for primitives

Example: Ray-Sphere intersection

ray: $x(t) = p_x + v_x t,\ \ y(t) = p_y + v_y t,\ \ z(t) = p_z + v_z t$



(unit) sphere: $x^2 + y^2 + z^2 = 1$

quadratic equation in *t* :

$$0 = (p_x + v_x t)^2 + (p_y + v_y t)^2 + (p_z + v_z t)^2 - 1$$
$$= t^2 (v_x^2 + v_y^2 + v_z^2) + 2t(p_x v_x + p_y v_y + p_z v_z)$$
$$+ (p_x^2 + p_y^2 + p_z^2) - 1$$

RAY INTERSECTIONS WITH OTHER PRIMITIVES

- Implicit functions:
  - Spheres at arbitrary positions
    - Same thing
  - Conic sections (hyperboloids, ellipsoids, paraboloids, cones, cylinders)
    - Same thing (all are quadratic functions!)
  - Higher order functions (e.g. tori and other quartic functions)
    - In principle the same
    - But root-finding difficult
    - Numerical methods

RAY INTERSECTIONS WITH OTHER PRIMITIVES

- Polygons:
  - First intersect ray with plane
    - linear implicit function
  - Then test whether point is inside or outside of polygon (2D test)
  - For convex polygons
    - Suffices to test whether point in on the right side of every boundary edge

RAY-TRACING: PRACTICALITIES

- Generation of rays
- Intersection of rays with geometric primitives
- **Geometric transformations**
- Lighting and shading
- Speed: Reducing number of intersection tests
  - E.g. use BSP trees or other types of space partitioning

RAY-TRACING: TRANSFORMATIONS

- Note: rays replace perspective transformation
- Geometric Transformations:
  - Similar goal as in rendering pipeline:
    - Modeling scenes convenient using different coordinate systems for individual objects
  - Problem:
    - Not all object representations are easy to transform
      - This problem is fixed in rendering pipeline by restriction to polygons (affine invariance!)

RAY-TRACING: TRANSFORMATIONS

- Ray Transformation:
  - For intersection test, it is only important that ray is in same coordinate system as object representation
  - Transform all rays into object coordinates
    - Transform camera point and ray direction by <u>inverse</u> of model/view matrix
  - Shading has to be done in world coordinates (where light sources are given)
    - Transform object space intersection point to world coordinates
    - Thus have to keep both world and object-space ray

RAY-TRACING: PRACTICALITIES

- Generation of rays
- Intersection of rays with geometric primitives
- Geometric transformations
- **Lighting and shading**
- Speed: Reducing number of intersection tests
  - E.g. use BSP trees or other types of space partitioning

RAY-TRACING: DIRECT ILLUMINATION

- Light sources:
  - For the moment: point and directional lights
  - More complex lights are possible
    - Area lights
    - Fluorescence

RAY-TRACING: DIRECT ILLUMINATION

- Local surface information (normal…)
  - For implicit surfaces $F(x,y,z)=0$:
    normal $\mathbf{n}(x,y,z)$ is gradient of F:

$$n(x,y,z) = \nabla F(x,y,z) = \begin{pmatrix} \partial F(x,y,z)/\partial x \\ \partial F(x,y,z)/\partial y \\ \partial F(x,y,z)/\partial z \end{pmatrix}$$

  - Example:

$$F(x,y,z) = x^2 + y^2 + z^2 - r^2$$

$$\mathbf{n}(x,y,z) = \begin{pmatrix} 2x \\ 2y \\ 2z \end{pmatrix}$$

Needs to be normalized!

RAY-TRACING: DIRECT ILLUMINATION

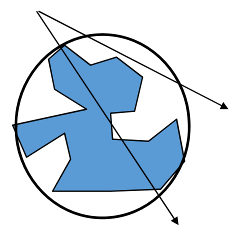
- For triangle meshes
  - Interpolate per-vertex information as in rendering pipeline
    - Phong shading!
    - Same as discussed for rendering pipeline
  - Difference to rendering pipeline:
    - Have to compute Barycentric coordinates for every intersection point (e.g plane equation for triangles)

RAY-TRACING: PRACTICALITIES

- Generation of rays
- Intersection of rays with geometric primitives
- Geometric transformations
- Lighting and shading
- **Speed:** Reducing number of intersection tests

OPTIMIZED RAY-TRACING

- Basic algorithm is simple but VERY expensive
- Optimize…
  - Reduce number of rays traced
  - Reduce number of ray-object intersection calculations
- Parallelize
  - Cluster
  - GPU
- Methods
  - Bounding Boxes
  - Spatial Subdivision
    - Visibility, Intersection/Collision
  - Tree Pruning

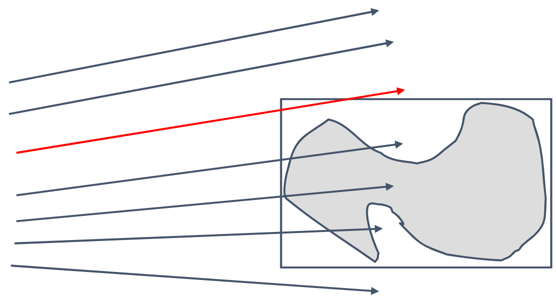


SPATIAL SUBDIVISION DATA STRUCTURES

- Goal: reduce number of intersection tests per ray
- Lots of different approaches:
  - (Hierarchical) bounding volumes
  - Hierarchical space subdivision
    - Octree, k-D tree, BSP tree

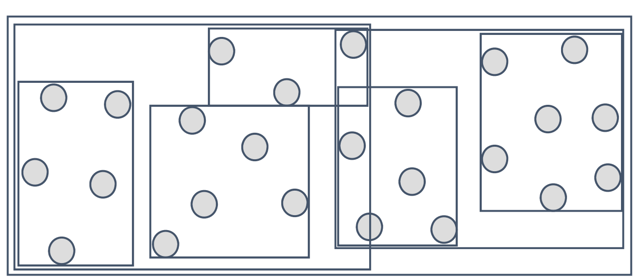
BOUNDING VOLUMES: IDEA

- Don't test each ray against complex objects (e.g. triangle mesh)
- Do a quick *conservative* test first which eliminates most rays
  - Surround complex object by simple, easy to test geometry (e.g. sphere or axis-aligned box)
    - Reduce false positives: make bounding volume as tight as possible!



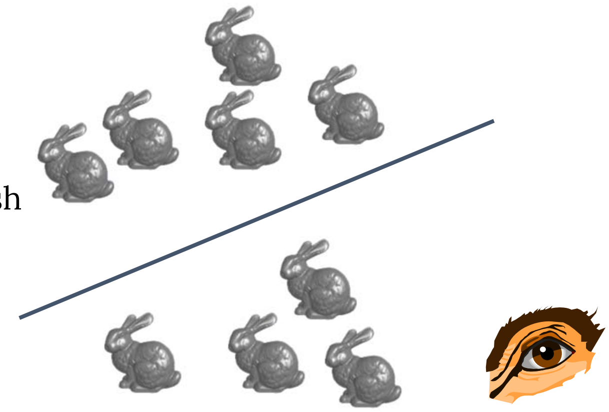
HIERARCHICAL BOUNDING VOLUMES

- Extension of previous idea:
  - Use bounding volumes for groups of objects

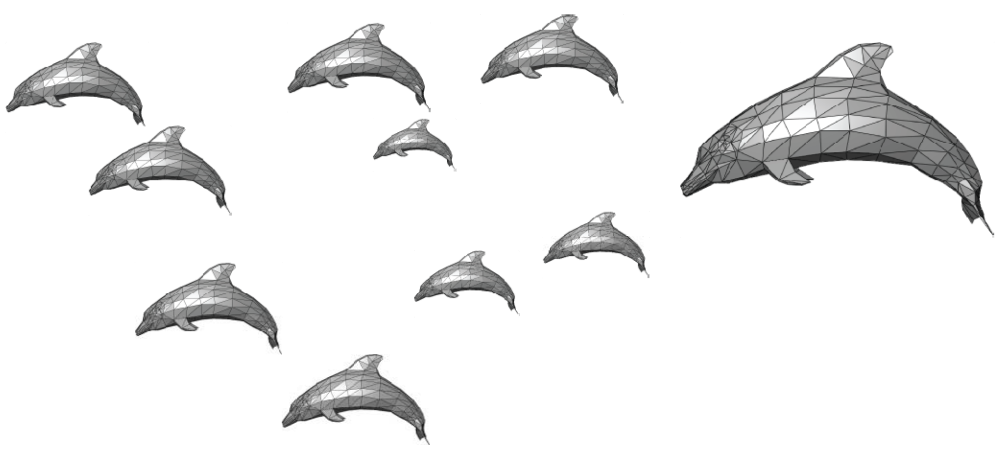


BSP TREES: IDEA

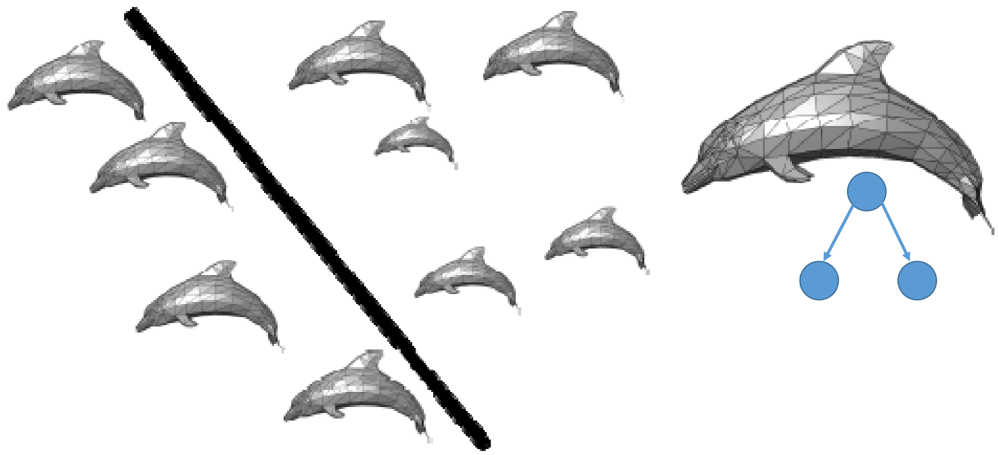
- For a plane, objects on the *same* side of plane as viewer CANNOT be occluded by objects on other side
- Intersect closer side first
- if ray doesn't intersect plane?
  - **can't intersect other side!**
- Idea:
  - Recursively split space by planes
  - Traverse resulting tree to establish rendering/intersection order
    - Test eye location w.r.t. each plane



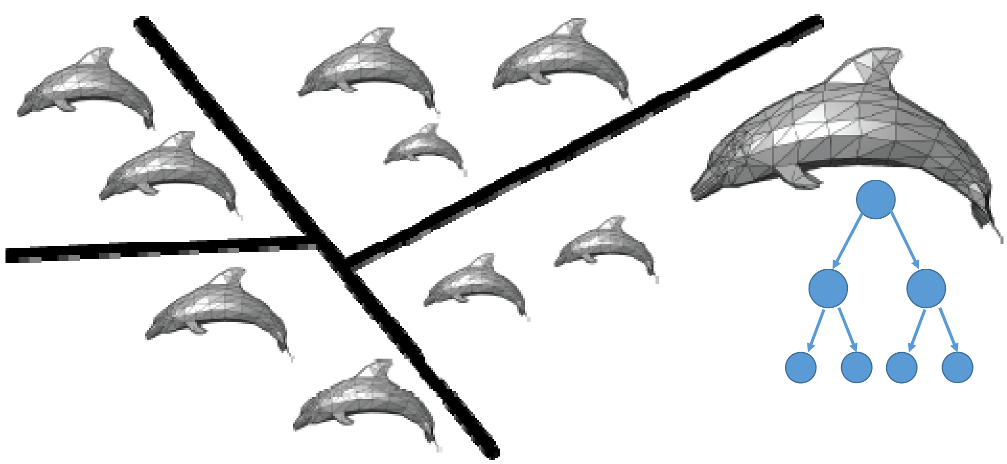
BSP TREES: CONSTRUCTION



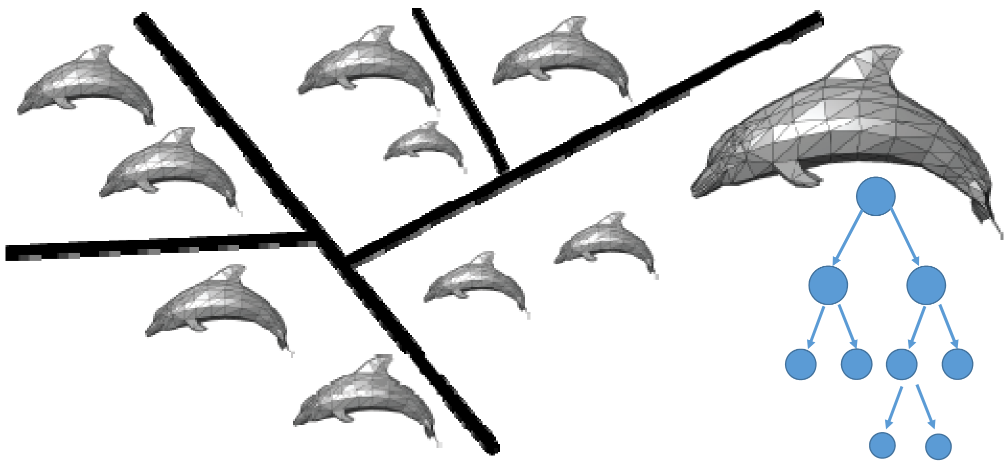
BSP TREES: CONSTRUCTION



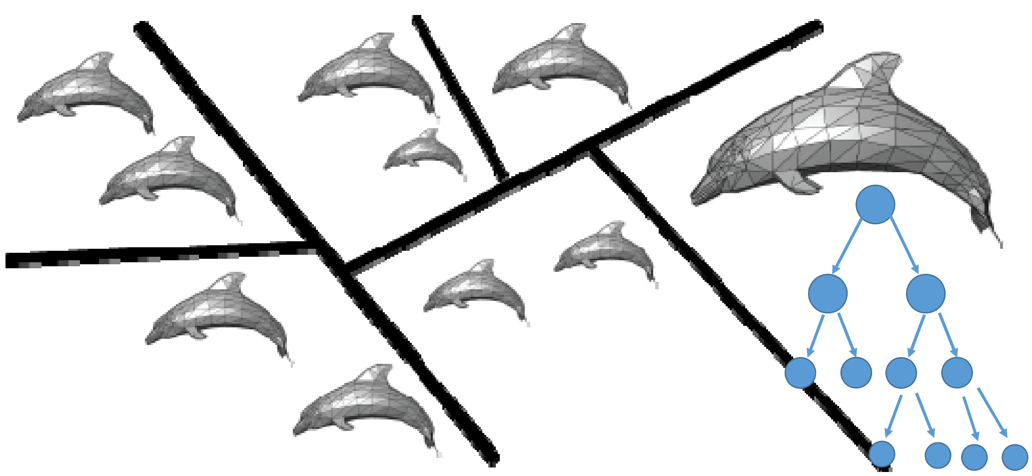
BSP TREES: CONSTRUCTION



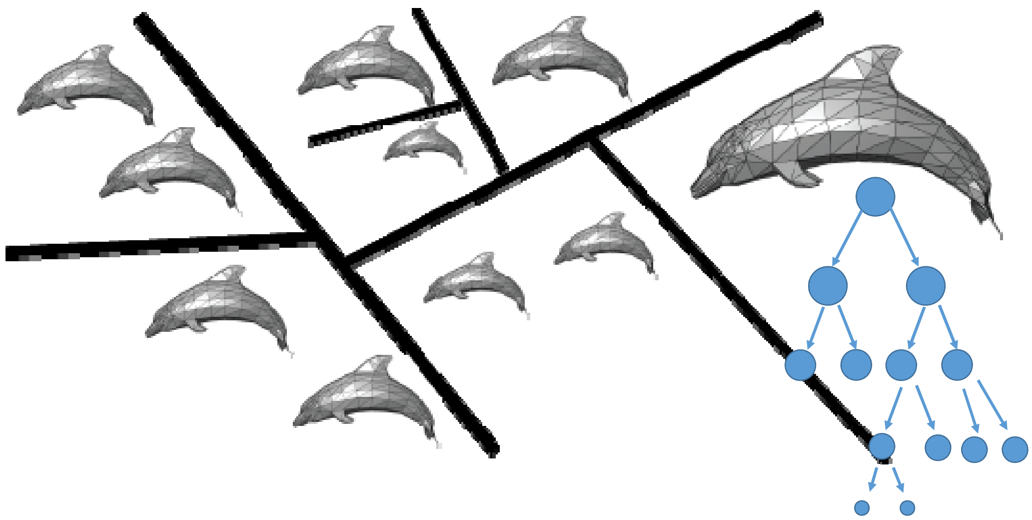
BSP TREES: CONSTRUCTION



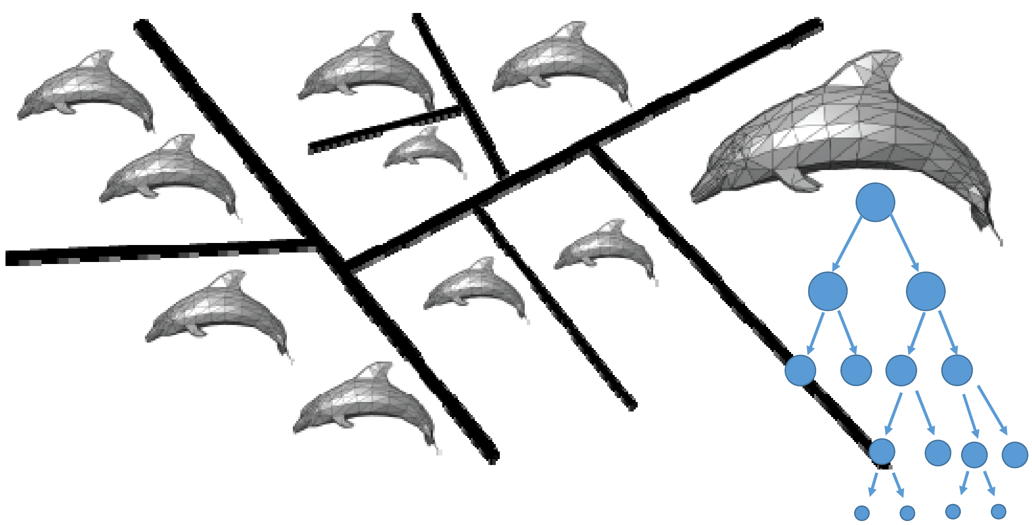
BSP TREES: CONSTRUCTION



BSP TREES: CONSTRUCTION

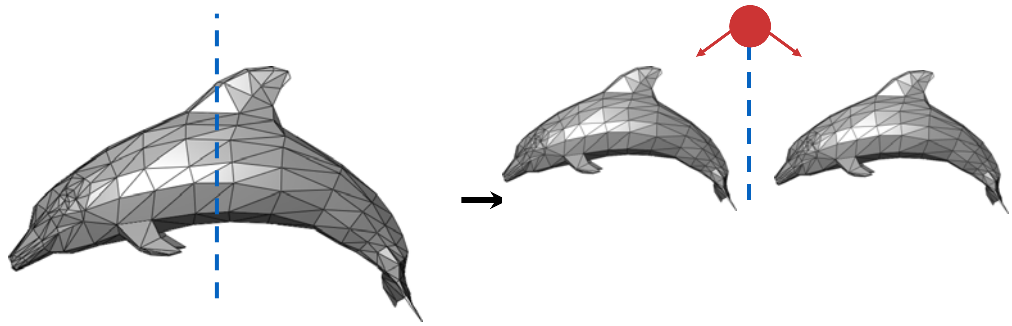


BSP TREES: CONSTRUCTION



SPLITTING OBJECTS

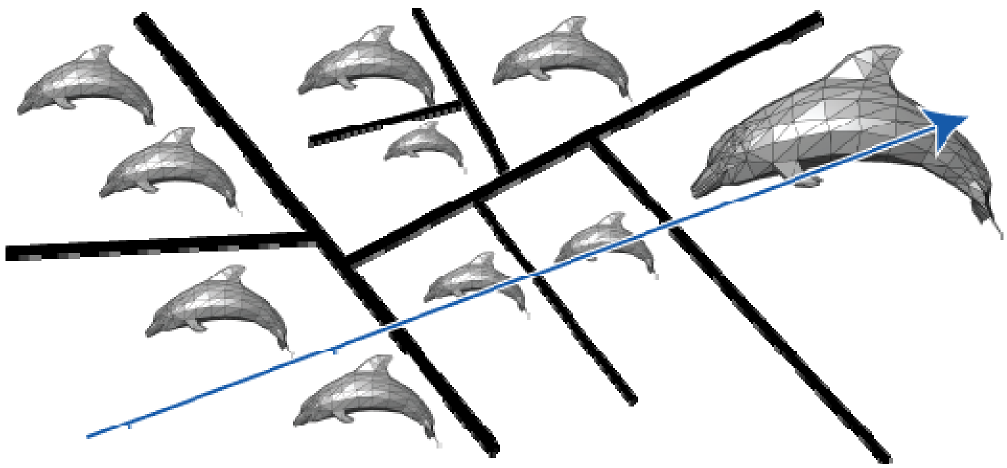
- But what if a splitting plane passes through an object?
  - Duplicate (Consider object in both half-spaces)



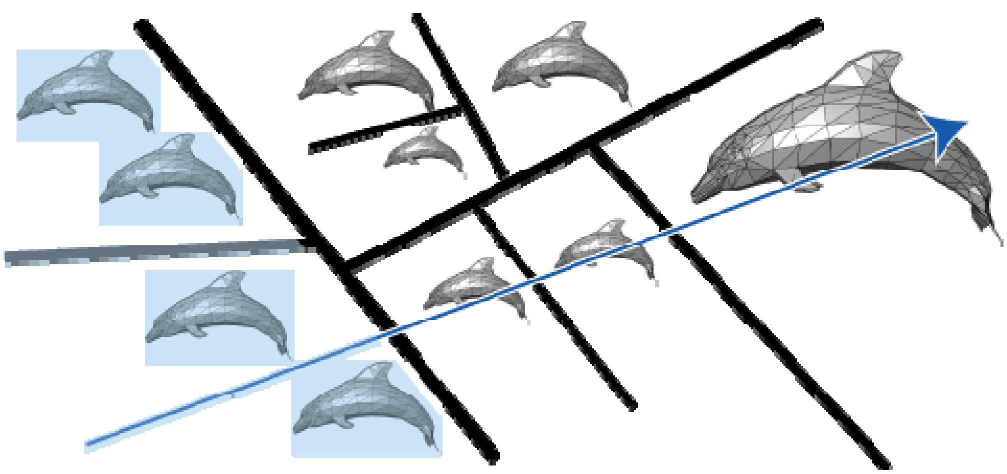
TRAVERSING BSP TREES

- Tree creation independent of viewpoint
  - Preprocessing step
- Tree traversal uses ray origin
  - Runtime, happens for many different rays (=different origins)

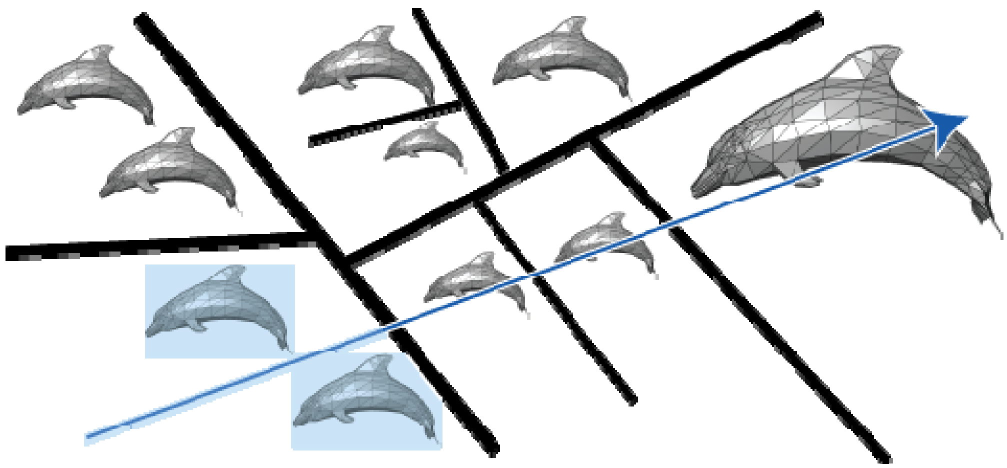
BSP TREES: TRAVERSAL



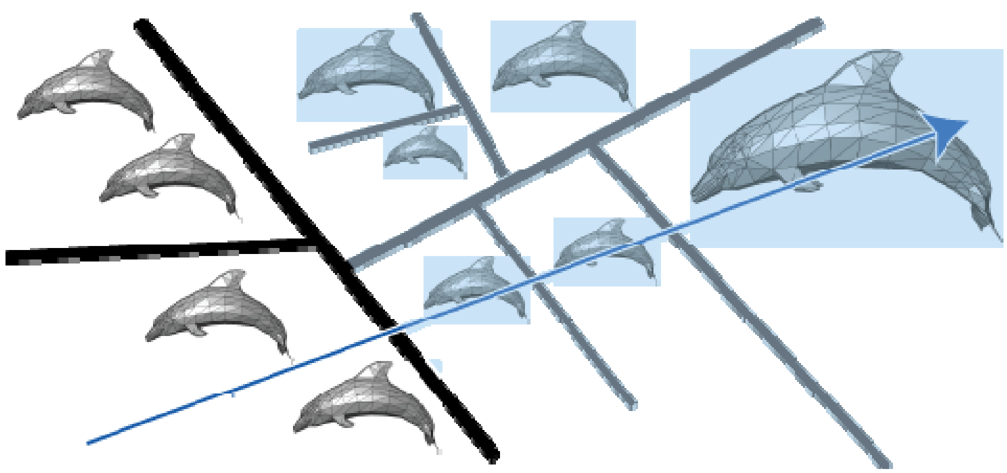
BSP TREES: TRAVERSAL



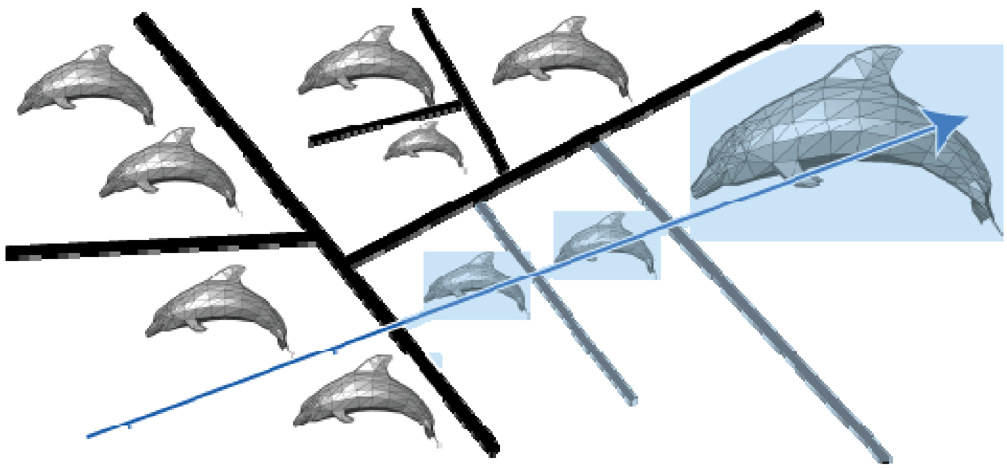
BSP TREES: TRAVERSAL



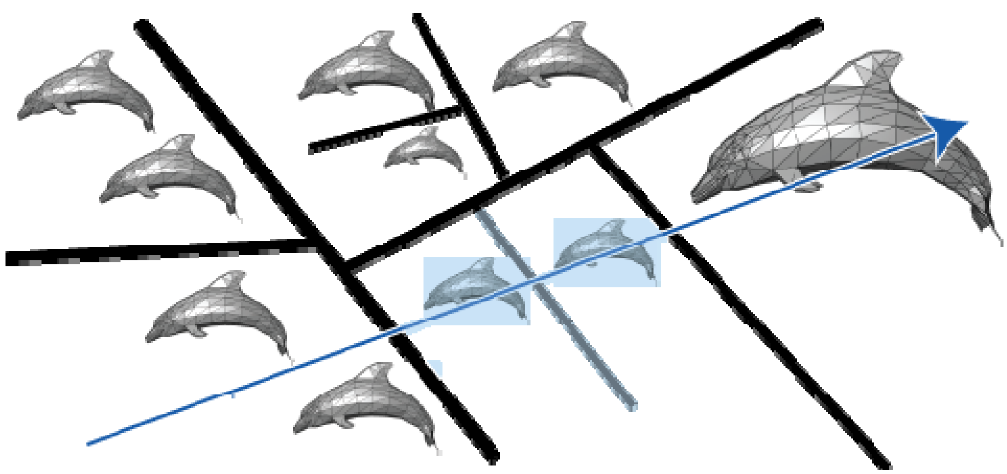
BSP TREES: TRAVERSAL



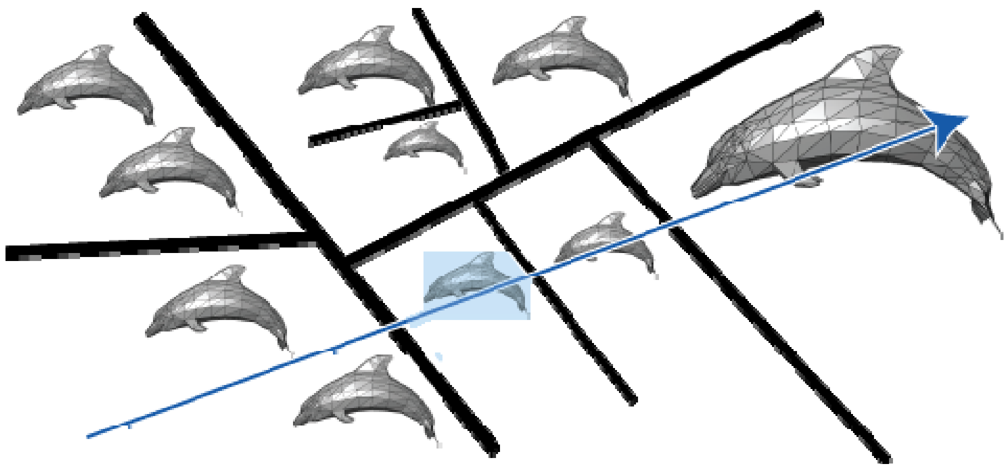
BSP TREES: TRAVERSAL



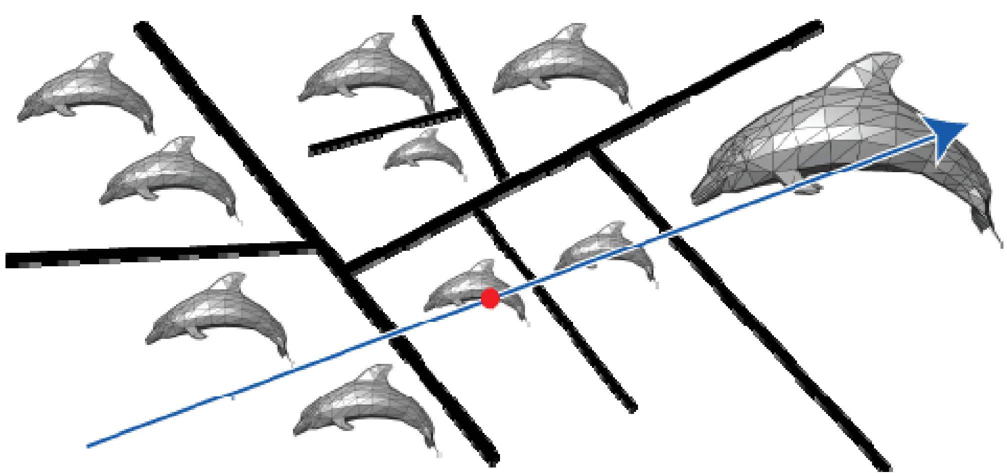
BSP TREES: TRAVERSAL



BSP TREES: TRAVERSAL



BSP TREES: TRAVERSAL



TRAVERSING BSP TREES

- Each plane divides world into near and far
  - For given ray, decide which side is near and which is far
    - Check which side of plane viewpoint is on independently for each tree vertex
    - Tree traversal differs depending on viewpoint!
  - Recursive algorithm
    - Intersect with near side
    - If no intersection, and ray intersects the plane,
      - Intersect with far side

TRAVERSING BSP TREES

Let *v* be a node, *r* a ray
Intersect(*v*, *r*)

if *v* is leaf
then

intersect *r* with each object in *v* and return closest or nil if none found

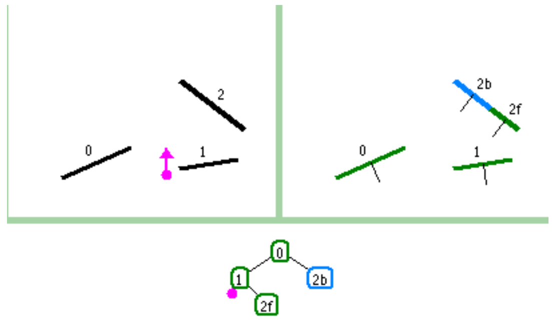
near = child node in half space containing the origin of *ray*
far = the other child
hit = Intersect(near, r)
if *hit* is nil and *ray* intersects plane defined by *v*
then

hit = Intersect(*far*, *r*)

return *hit*

BSP DEMO

- Useful demo:
  - http://symbolcraft.com/graphics/bsp



SUMMARY: BSP TREES

- Pros:
  - Simple, elegant scheme
  - Faster intersections
  - Correct version of painter's algorithm back-to-front rendering approach
  - Still very popular for video games
- Cons:
  - Slow(ish) to construct tree: O(n log n) to split, sort
  - Splitting increases polygon count: $O(n^2)$ worst-case
  - => Algorithm restricted to static scenes

SPATIAL SUBDIVISION DATA STRUCTURES

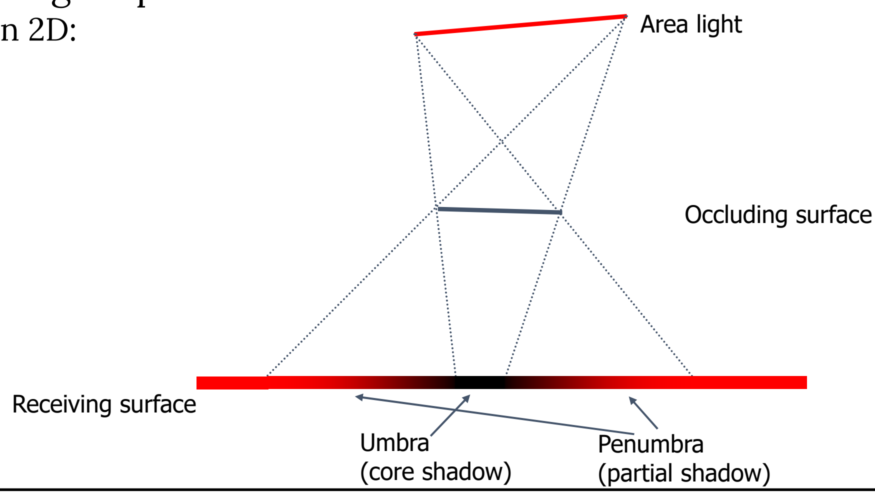
- Bounding Volumes:
  - Find simple object completely enclosing complicated objects
    - Boxes, spheres
  - Hierarchically combine into larger bounding volumes
- Spatial subdivision data structure:
  - Partition the whole space into cells
    - Grids, octrees, (BSP trees)
  - Simplifies and accelerates traversal
  - Performance less dependent on order in which objects are inserted

SOFT SHADOWS: AREA LIGHT SOURCES

- So far:
  - All lights were either point-shaped or directional
    - Both for ray-tracing and the rendering pipeline
  - Thus, at every point, we only need to compute lighting formula and shadowing for ONE direction per light
- In reality:
  - All lights have a finite area
  - Instead of just dealing with one direction, we now have to integrate over all directions that go to the light source

AREA LIGHT SOURCES

- Area lights produce soft shadows:
  - In 2D:

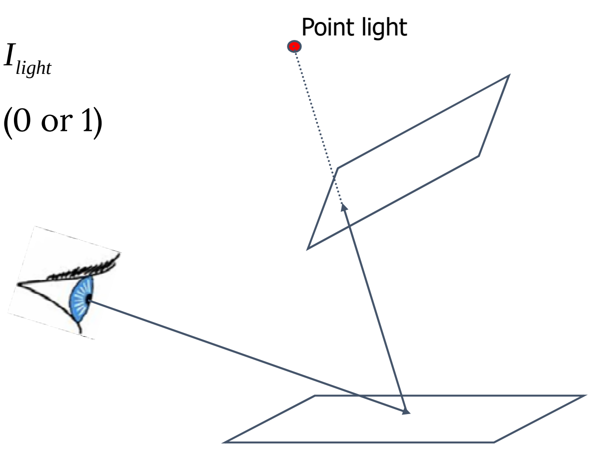


AREA LIGHT SOURCES

- Point lights:
  - Only one light direction:

$$I_{reflected} = \rho \cdot V \cdot I_{light}$$

  - V is visibility of light (0 or 1)
  - ρ is lighting model (e.g. diffuse or Phong)

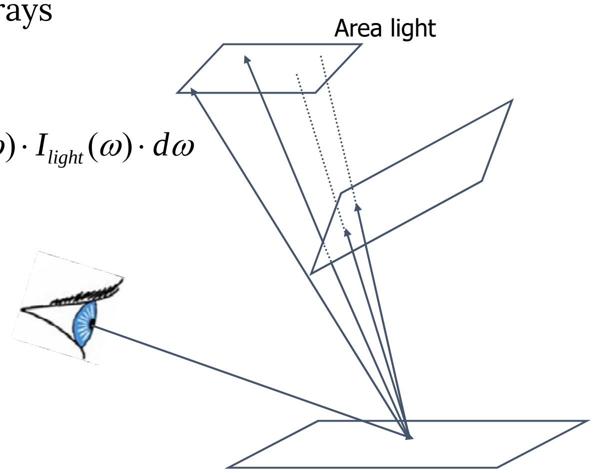


AREA LIGHT SOURCES

- Area Lights:
  - Infinitely many light rays
  - Need to integrate over all of them:

$$I_{reflected} = \int_{\substack{light \\ directions}} \rho(\omega) \cdot V(\omega) \cdot I_{light}(\omega) \cdot d\omega$$

  - Lighting model visibility and light intensity can now be different for every ray!



INTEGRATING OVER LIGHT SOURCE

- Rewrite the integration
  - Instead of integrating over directions

$$I_{reflected} = \int_{\substack{light \\ directions}} \rho(\omega) \cdot V(\omega) \cdot I_{light}(\omega) \cdot d\omega$$

  integrate over points on the light source

$$I_{reflected}(q) = \int_{s,t} \rho(p-q) \cdot V(p-q) I_{light}(p) \cdot ds \cdot dt$$

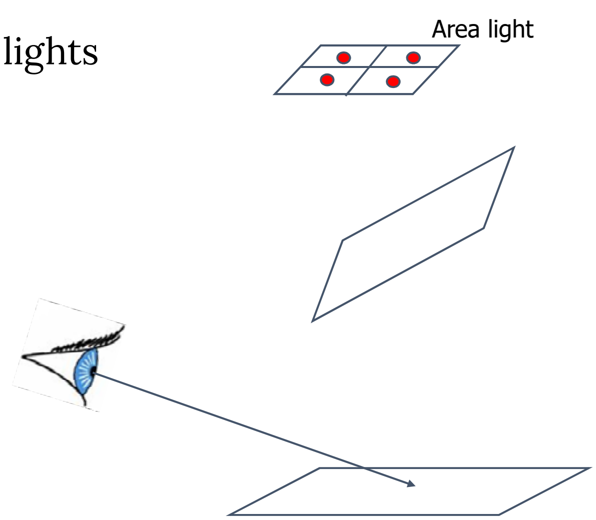
    - q point on reflecting surface
    - p= F(s,t) point on the area light
    - We are integrating over p

INTEGRATION

- Problem:
  - Except for basic case not solvable analytically!
    - Largely due to the visibility term
- So:
  - Use numerical integration = approximate light with lots of point lights

NUMERICAL INTEGRATION

- Regular grid of point lights
  - Problem: Too regular see 4 hard shadows
  - Need LOTS of points to avoid this problem



SOLUTION: MONTE-CARLO

- Next time!