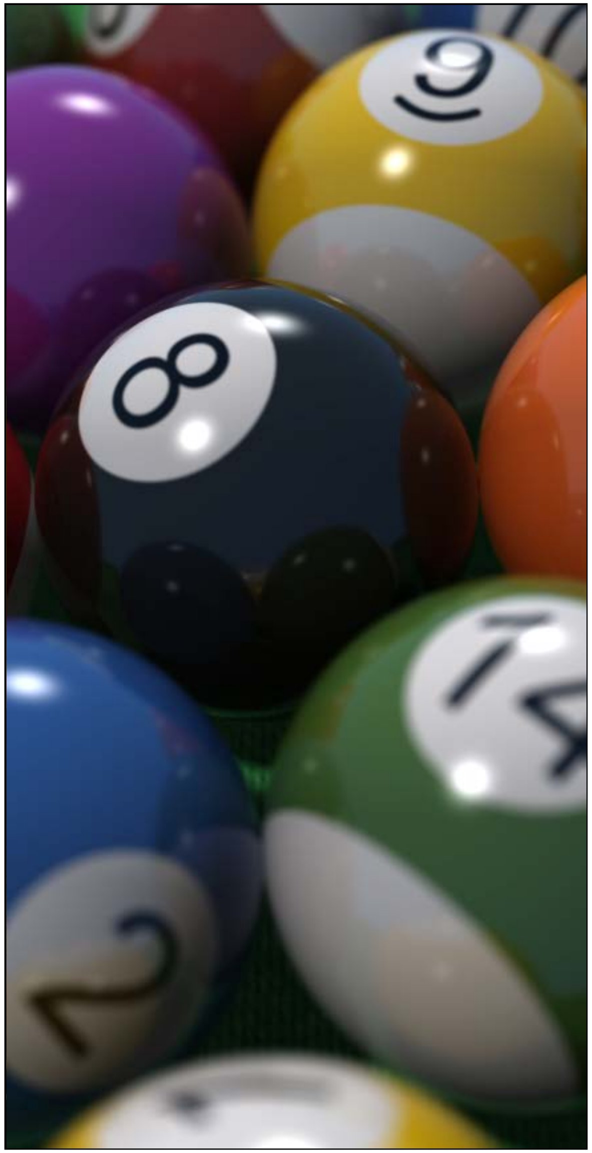
COMPUTER GRAPHICS COURSE

Ray Tracing



Georgios Papaioannou - 2015

RAY TRACING PRINCIPLES

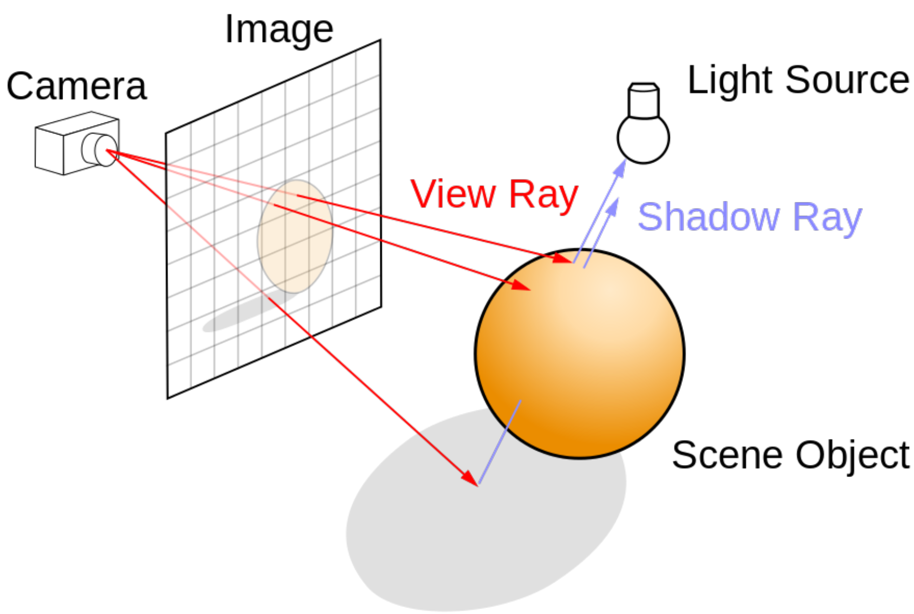
What is ray tracing?

- A general mechanism for sampling paths of light in a 3D scene
- We will use this mechanism in path tracing

- Ray Casting mechanism:
  - Rays are cast from a point in space towards a specified direction
  - Rays are intersected with geometry primitives
  - The closest intersection is regarded as a ray “hit”
  - Lighting or other attributes are evaluated at the hit location

Simple Ray Casting - Appel's Method

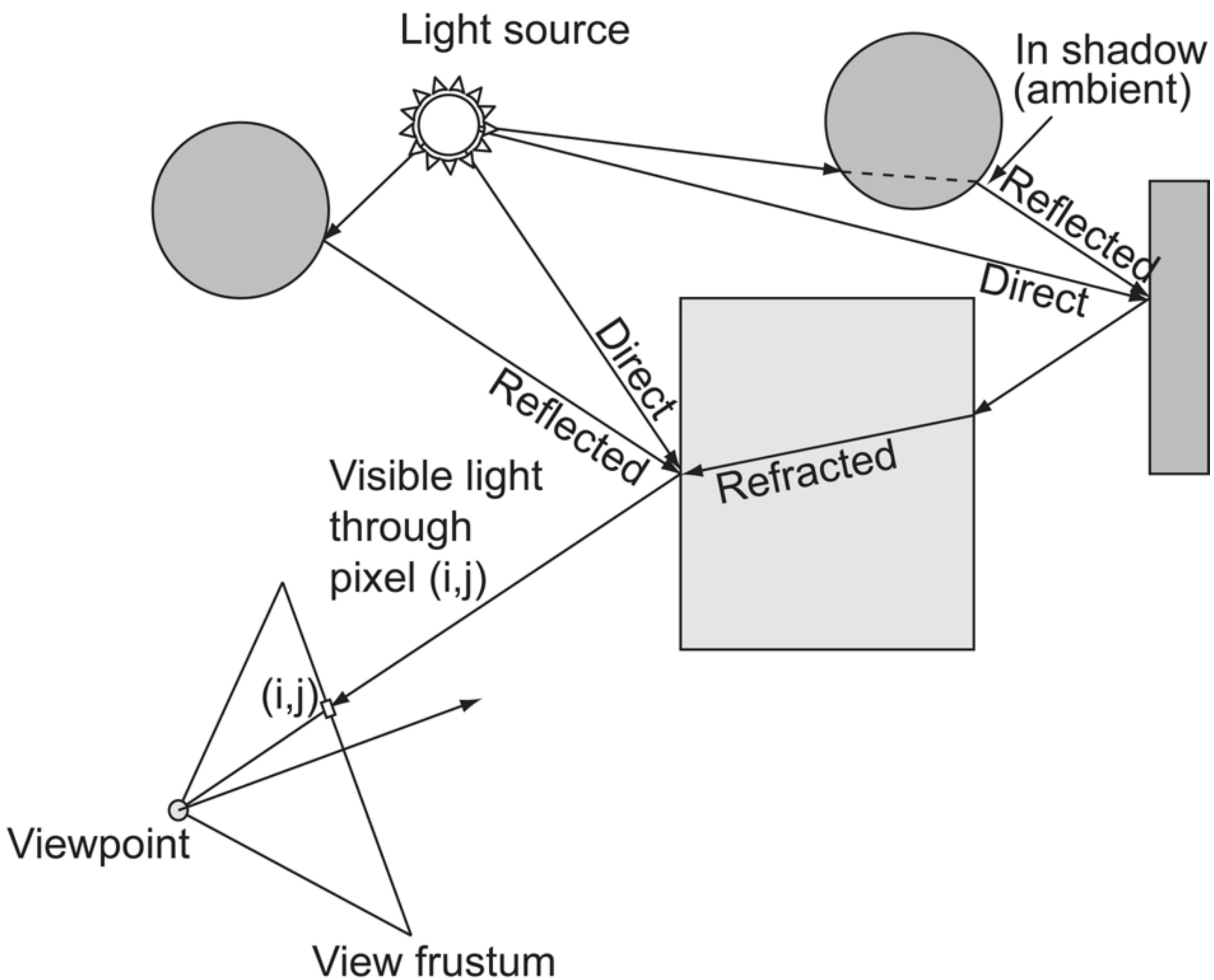
- For each image pixel, a ray (line segment) is cast from the viewpoint, crosses the pixel and is directed toward the scene
- For hit points:
  - Send (at least) one ray to each light source and check visibility (shadow)
  - Shade the point using a local illumination model



Whitted-style Ray Tracing (1)

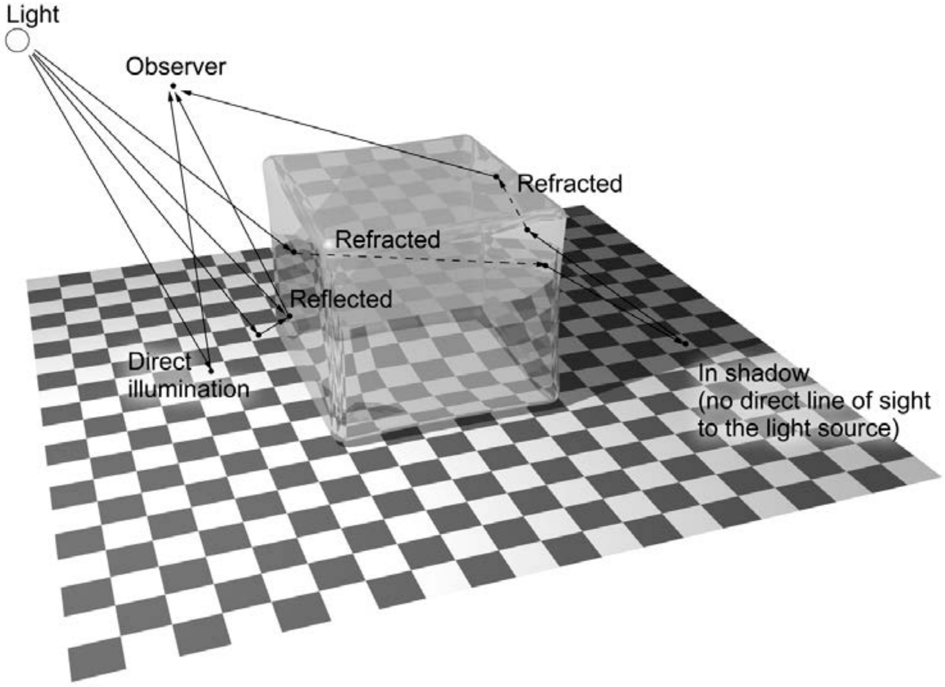
- For each image pixel, a ray (line segment) is cast from the viewpoint, crosses the pixel and is directed toward the scene
- The ray hits the objects, is absorbed or deflected in order to gather what the observer would “see” through each pixel
- This is a recursive algorithm that spawns new rays at each hit point

Whitted-style Ray Tracing (2)



Why not Trace Rays from the Lights? (1)

- Infinite rays leave a light source but only a small number lands on the viewport
  - Even fewer when a pinhole camera is considered
  - Extremely low probability to hit → Computationally intractable



Why not Trace Rays from the Lights? (2)

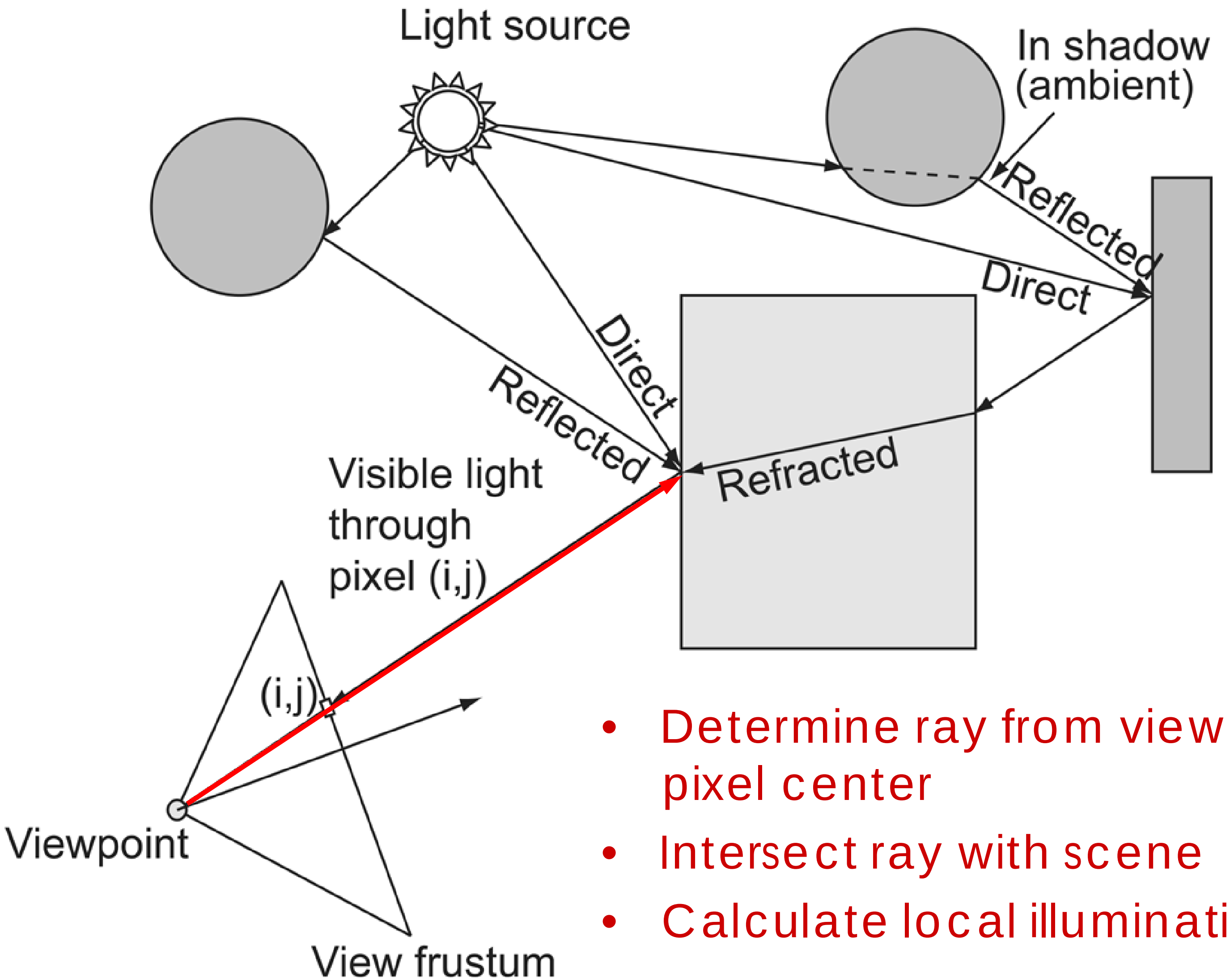
- In practice, many path tracing variants, which use the ray tracing mechanism to form the light paths, do trace rays from both the camera and the light source domain

Source: http://indigorenderer.com

More than Direct Illumination

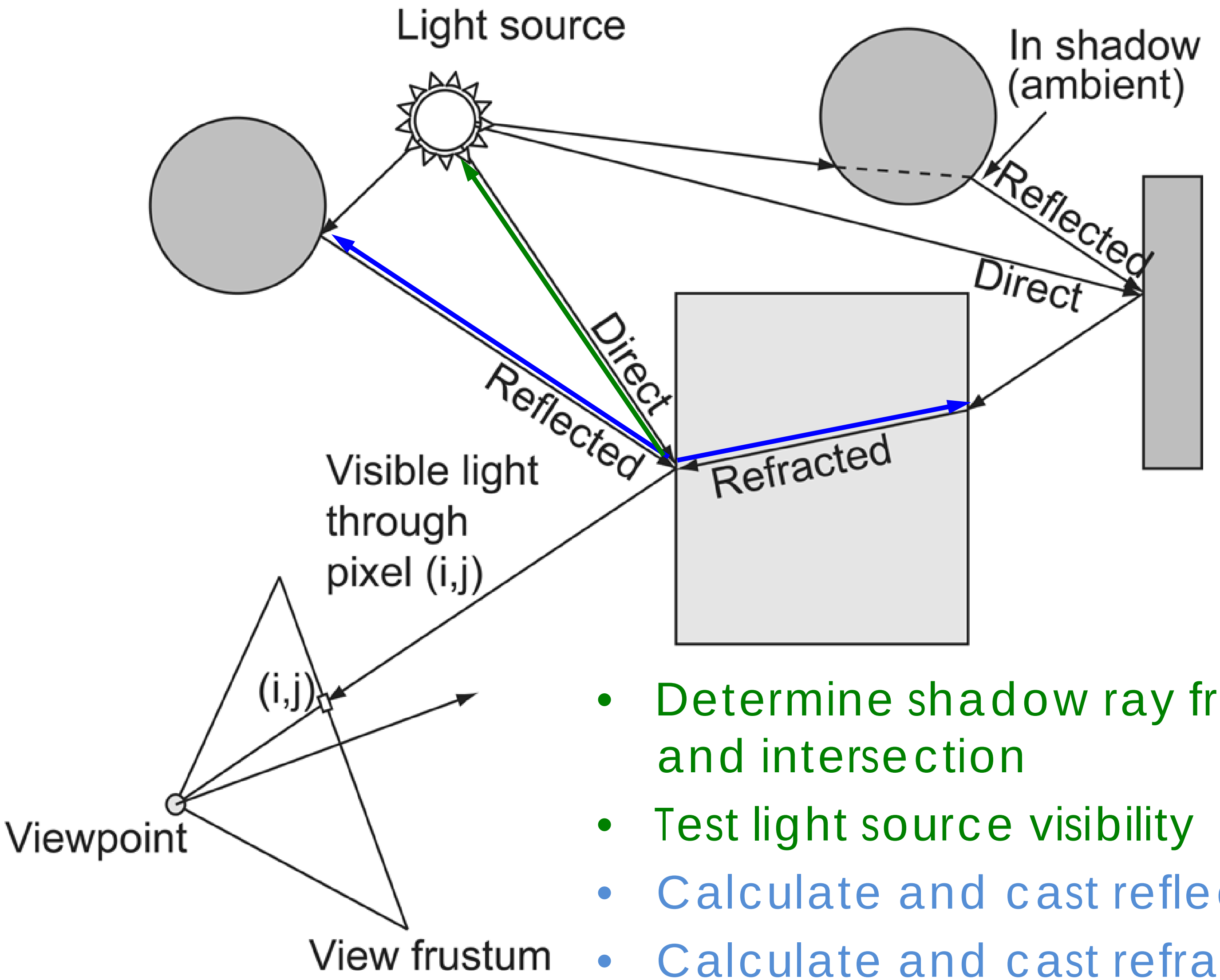
- Ray tracing is an elegant recursive algorithm
- The first (primary) rays spawned hit the nearest surface: same result as in direct rendering
- Secondary rays can be spawned from the intersection point to track indirect illumination
- Secondary rays capture:
  - Shadows (inherent part of ray-tracing, no special algorithm required)
  - Reflections (light bouncing off surfaces)
  - Refracted (transmitted) light through the objects

Tracing Rays: Level 0 (Primary Rays)



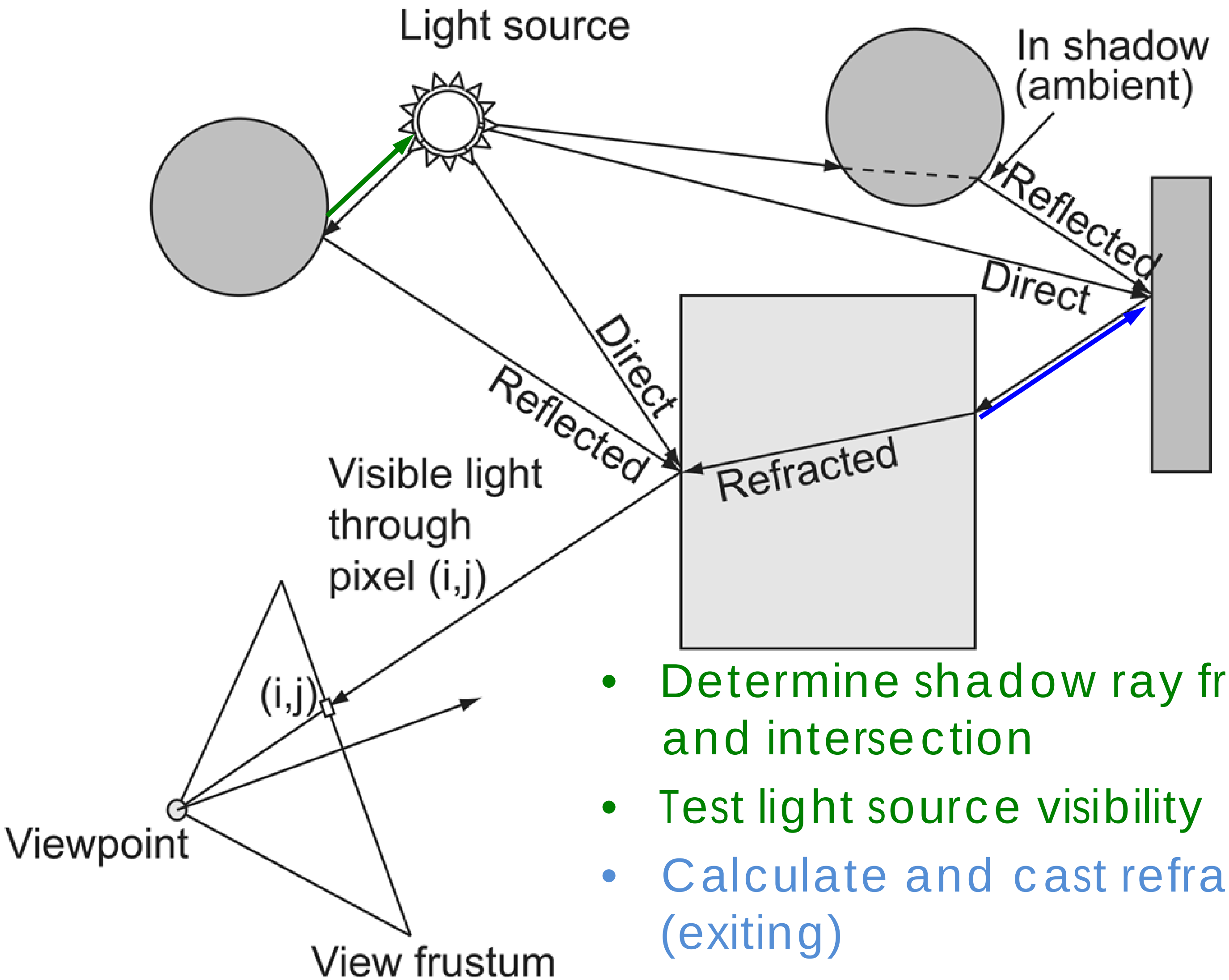
- Determine ray from viewpoint and pixel center
- Intersect ray with scene
- Calculate local illumination model

Tracing Rays: Level 1



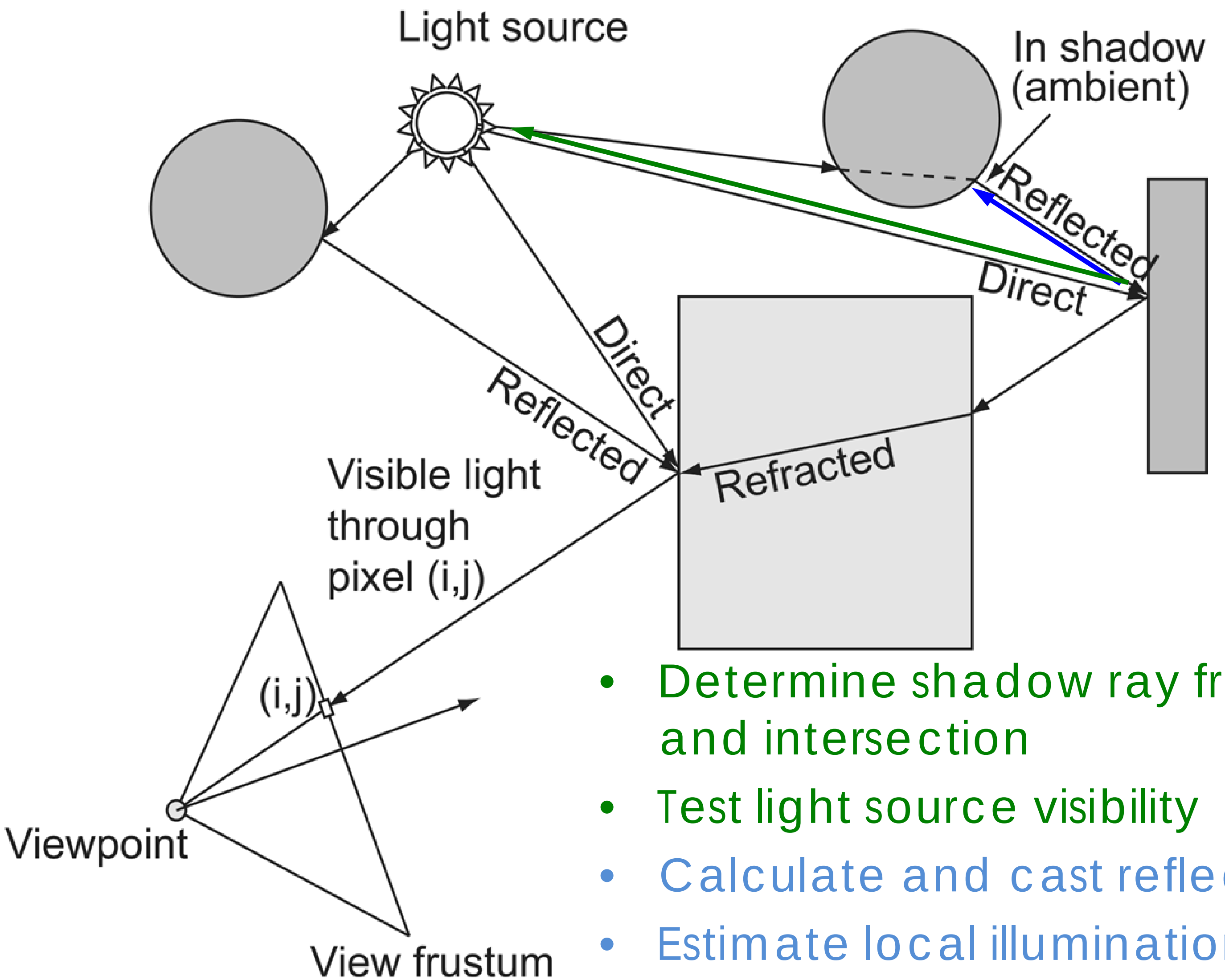
- Determine shadow ray from light and intersection
- Test light source visibility
- Calculate and cast reflection ray
- Calculate and cast refraction ray

Tracing Rays: Level 2



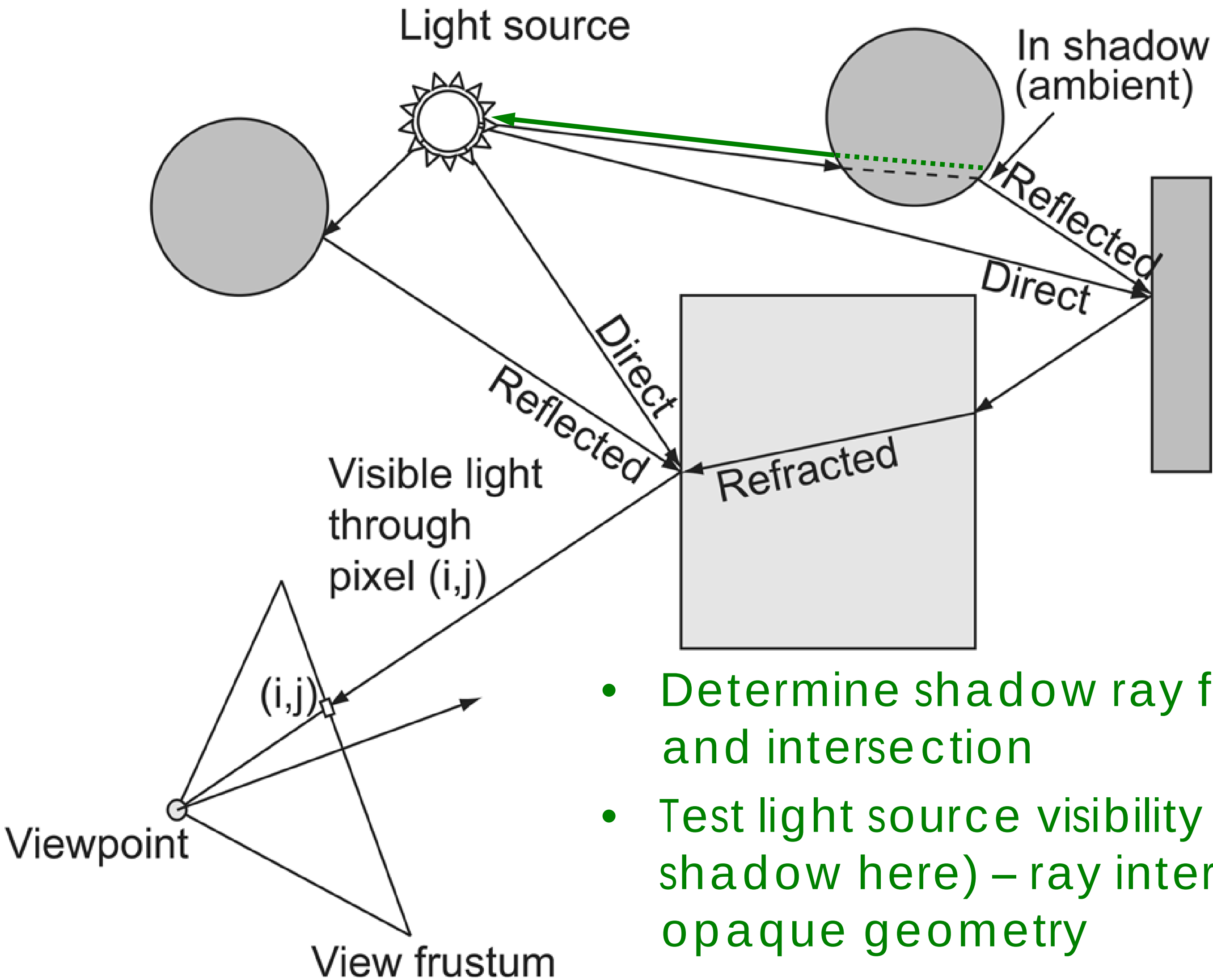
- Determine shadow ray from light and intersection
- Test light source visibility
- Calculate and cast refraction ray (exiting)
- Estimate local illumination

Tracing Rays: Level 3



- Determine shadow ray from light and intersection
- Test light source visibility
- Calculate and cast reflection ray
- Estimate local illumination

Tracing Rays: Level 4

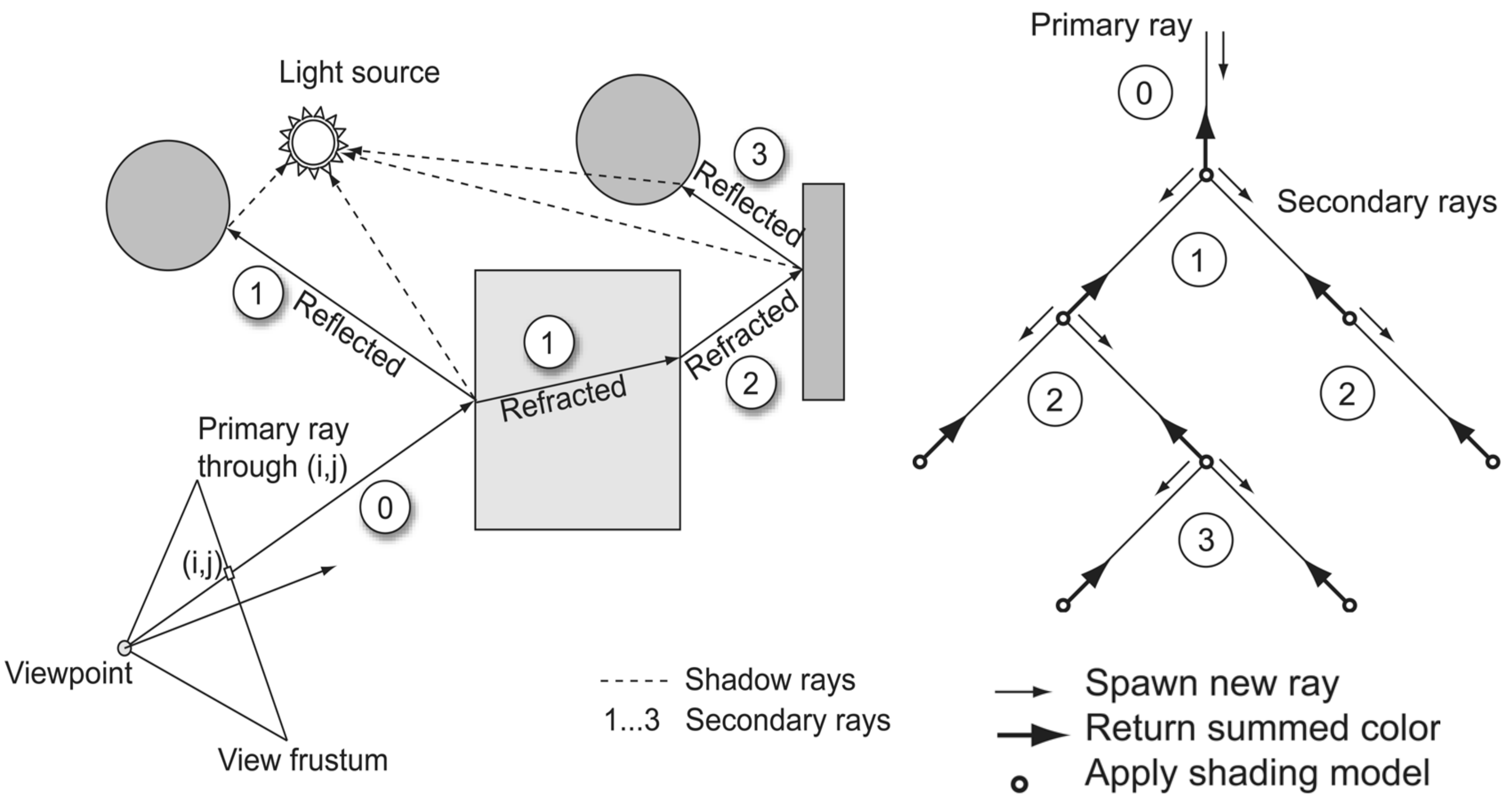


- Determine shadow ray from light and intersection
- Test light source visibility (in shadow here) – ray intersects opaque geometry

Who Determines What Rays to Spawn?

- Material properties:
  - Reflectance
  - Incandescence
  - Gloss
  - Permeability *a*
  - Index of refraction *n*
  - …
- Number, size and type of lights

Resulting Color



Spawn

Gather

The Basic Ray Tracing Algorithm

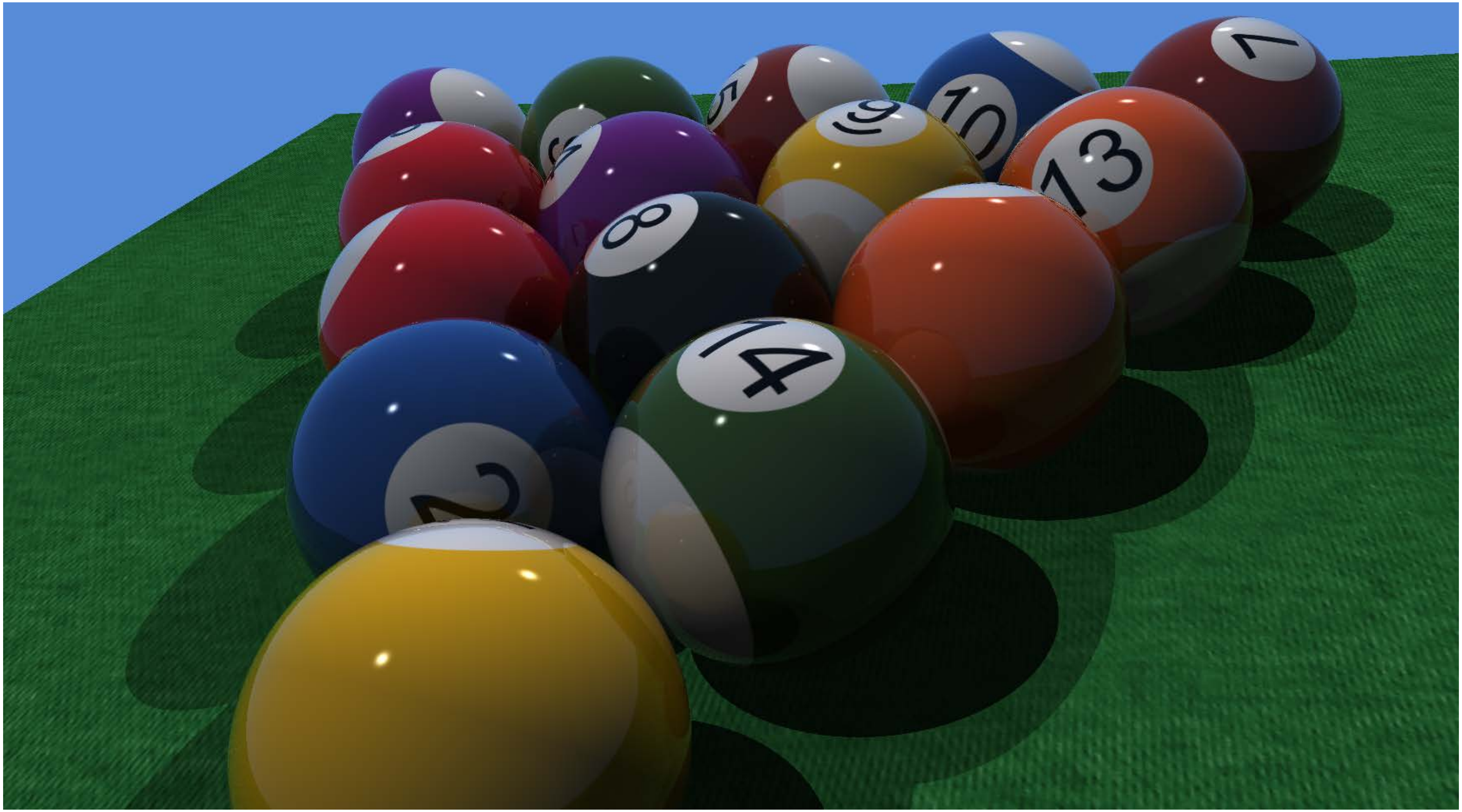
```
Color raytrace( Ray r, int depth, Scene world, vector <Light*> lights )
{ Ray   *refl, *tran;
  Color color_r, color_t, color_l;

  // Terminate if maximum recursion depth has been reached.
  if ( depth > MAX_DEPTH ) return backgroundColor;
  // Intersect ray with scene and keep nearest intersection point
  int hits = findClosestIntersection(r, world);
  if ( hits == 0 ) return backgroundColor;
  // Apply local illumination model, including shadows
  color_l = calculateLocalColor(r, lights, world);
  // Trace reflected and refracted rays according to material properties
  if (r->isect->surface->material->k_refl > 0)
  { refl = calculateReflection(r);
    color_r = raytrace(refl, depth+1, world, lights);
  }
  if (r->isect->surface->material->k_refr > 0)
  { tran = calculateRefraction(r);
    color_t = raytrace(tran, depth+1, world, lights);
  }
  return color_l + color_r + color_t;
}
```

Ray Tracing Results (1)

Source: http://hof.povray.org/images/ChristmasBaubles.jpg

Ray Tracing Results (2)



Simple ray-traced scene rendered at 60 frames per second on a modern GPU at 1080p

Source: NVIDIA OptiX SDK

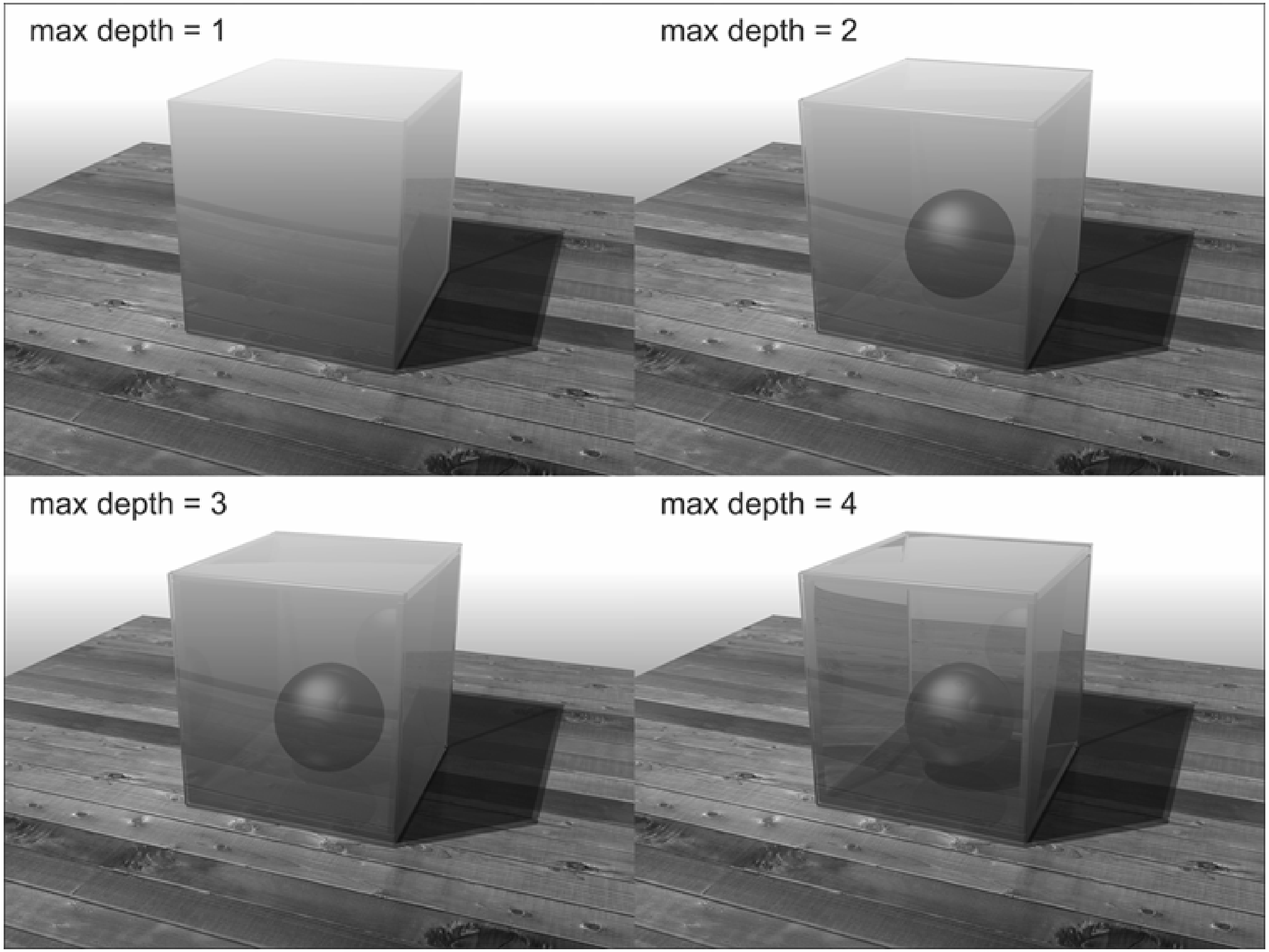
Comments

- Conventional primary and secondary ray – scene intersections must return the closest hit to the ray origin
- If reflection / refraction coefficients are (near) zero, no secondary rays are spawned
- If no surface is hit, the background color is returned
- Shadow determination is embedded in the local illumination calculation stage
- A maximum recursion depth is forced upon the algorithm; Necessary for complex scenes

Termination Criteria

- Maximum recursion depth is reached
- Zero reflectivity / transmission coefficients
- Ray contribution too small to be of significance: Attenuation due to participating medium density

Effect of Ray Tracing Depth on Images



Ray Data Structures – Requirements

- Degradation effects (absorption, scattering, splitting):
  - "strength" indicator (opposite of attenuation)
  - Optionally, recursion depth
- Distance sorting of hit points:
  - Avoid keeping all intersections and post-sort results
  - Keep nearest intersection point or
  - Cache distance to nearest hit point
- Local properties of hit point:
  - Need to keep track of hit material, primitive and local attributes (e.g. normal)

A Ray as a Data Structure - Minimum

```
class ray
{
public:
   ray(const vec3 & start, const vec3 & direction);
   void transform(const mat4 &m);

   vec3 origin;
   vec3 dir;
   vec3 n_isect;
   real t;              // real: defined as float or double
   void * payload;
};
```

Here, position is indirectly calculated from t

Pointer to an existing structure (e.g. a primitive) that holds the local attributes associated to the hit point

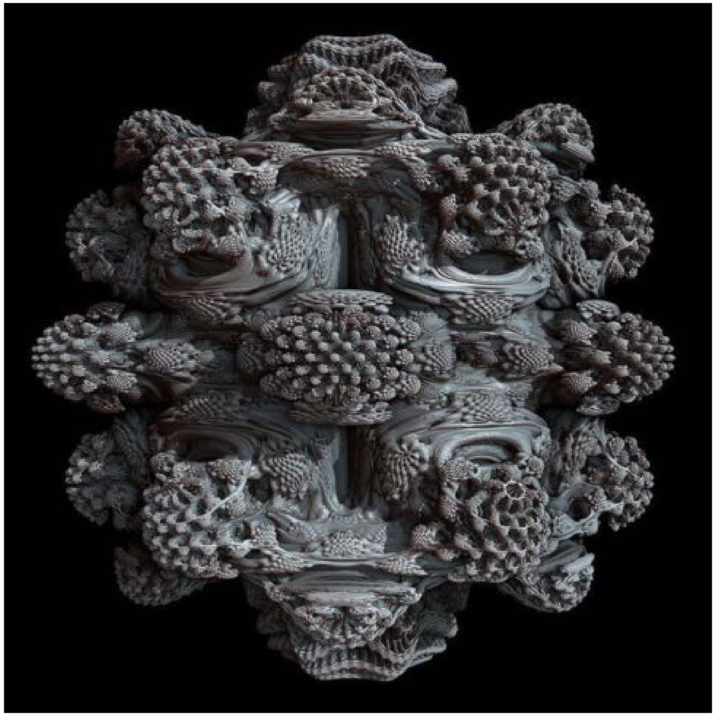
A Ray as a Data Structure - Extended

```
class ray
{
public:
   ray(void);
   ray(const vec3 & start, const vec3 & direction);
   void transform(const mat4 &m);

   vec3 origin;
   vec3 dir;
   int depth;
   vec3 p_isect;
   vec3 n_isect;
   vec3 barycentric;
   real t;
   real strength;
   bool hit;
   bool inside;
   class primitive *hit_primitive;
   void *payload;
};
```

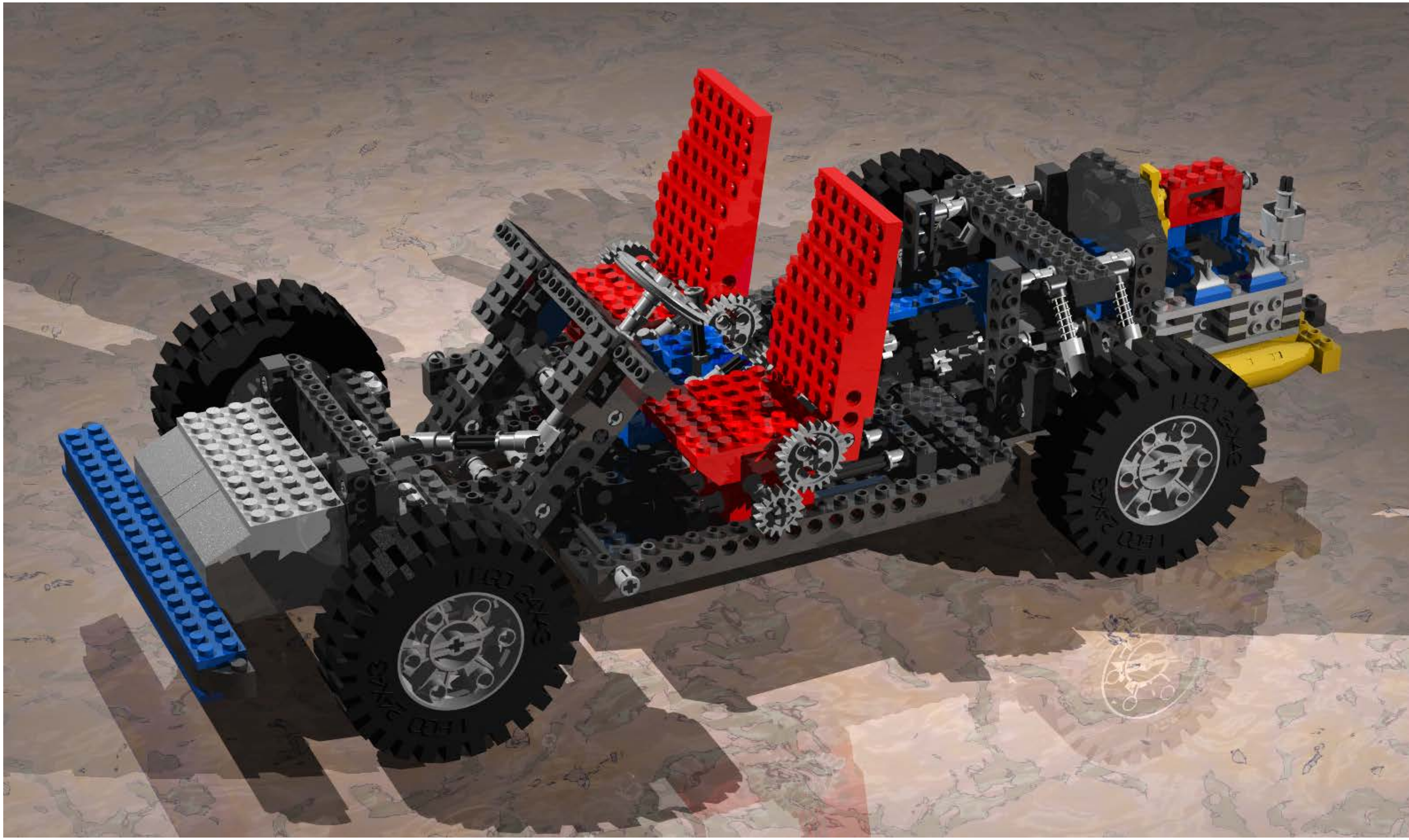
Ray – Scene Intersection: Primitives

- A primitive in ray tracing is any mathematical entity that can define a line-primitive equation (intersection points)
  - Polygons
  - Thresholded density (volume) data
  - Parametric surfaces
  - Analytical surfaces (e.g. spheres)
  - General equations (e.g. fractals)
  - Set operations (Constructive Solid Geometry)



Sources: http://paulbourke.net/geometry/blob/ http://www.subblue.com/blog/2009/12/13/mandelbulb

Ray Tracing Results (3)



Ray tracing using only geometric solids (with CSG in POVRAY)

Ray – Scene Intersection

- A naïve test exhaustively tests a ray against every primitive in the scene
- Ray – primitive intersections are the most frequent operations in ray tracing
- We try to minimize their complexity and number:
  - Hierarchical data structure acceleration
  - Early visibility culling
  - Frequent tests are performed with low-complexity primitives → search refinement
  - Parallel execution: Ray tracing is inherently highly parallel at many levels

Nearest Hit Determination

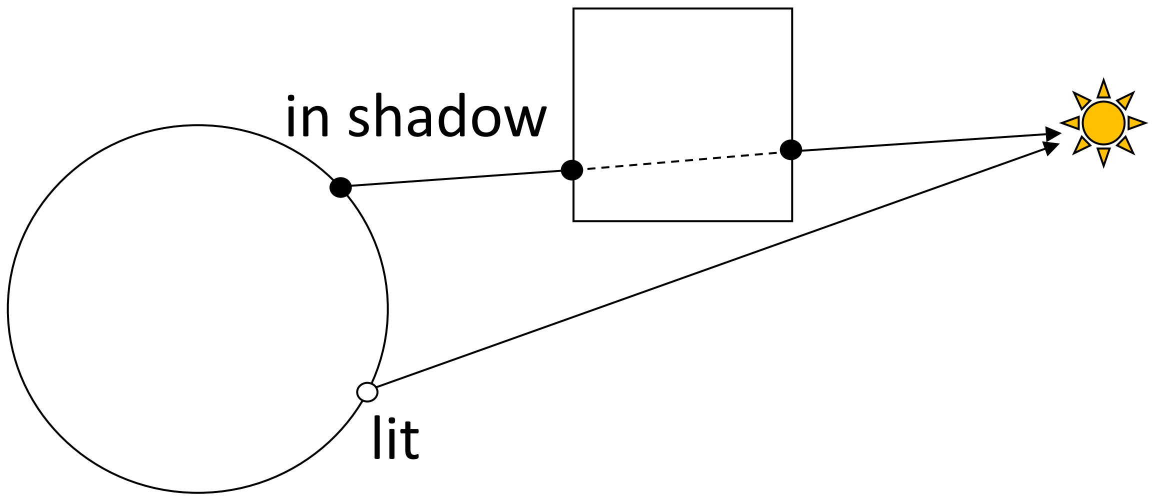
```
int findClosestIntersection(Ray r, Scene world)
{
    int hits=0;
    Ray r_temp = r;
    r.t = FLT_MAX;
    for ( j=0; j<world.numObjects(); j++ )
        for ( k=0; k<world.getObject(j)->numPrims(); k++ )
        {
            Primitive *prim = world.getObject(j)->getPrim(k);
            prim->intersect(r_temp);
             hits++;

            if ( r_temp.t < r.t )
                r = r_temp;
        }
    return hits;
}
```

All intersectable entities here are derived from the Primitive class and override the intersect() method

Shadows

- An intersection point is in shadow when the direct path to a light source is obstructed by a surface



Shadow Rays

- Cast “shadow rays” toward each light source during the local color estimation
- Shadow rays are cheaper:
  - Once light is blocked by a primitive, the search stops
  - No sorting is required
- Occluding geometry does not necessarily completely block or allow light through
  - Modulate the ray strength according to occluder transparency
  - Stop if strength becomes too low (in shadow)

Shadow Determination (1)

```
Color calculateLocalColor( Ray r, vector<Light*> lights,
      Scene world ) // point lights are assumed here
{
  int i, j, k;
  Color col = Color(0); // black

  // For all available lights, trace array towards them
  for ( i=0; i<lights.size(); i++ )
  {
    vec3 dir = normalize(lights[i]->pos-r.p_isect);
    Ray shadowRay = Ray(r.p_isect, dir);

    float strength = 1.0f;
    // Filter the light as it passes through the scene
    <SEE NEXT SLIDE>

    if (strength>0)
      col += strength * localShadingModel(r,prim,lights[i]->pos);
  }
  return col;
}
```

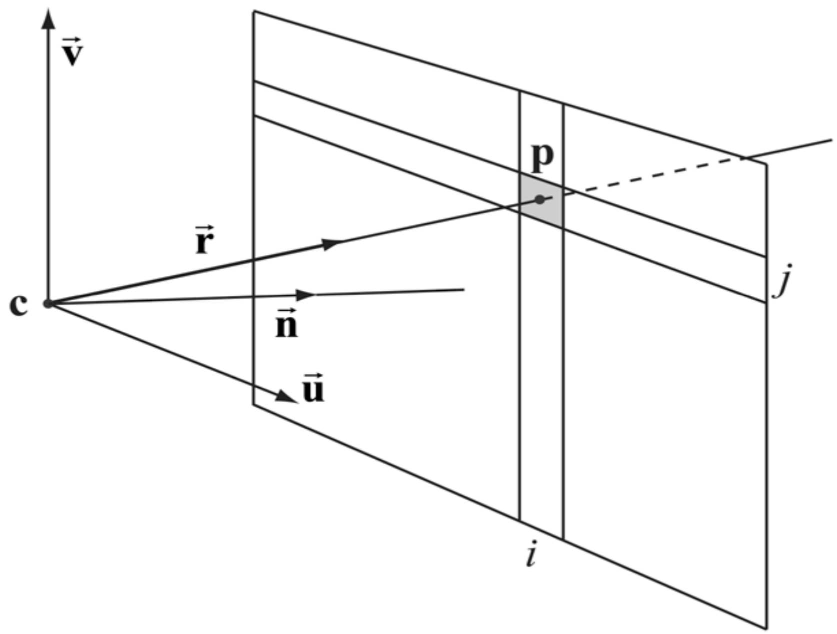
Shadow Determination (2)

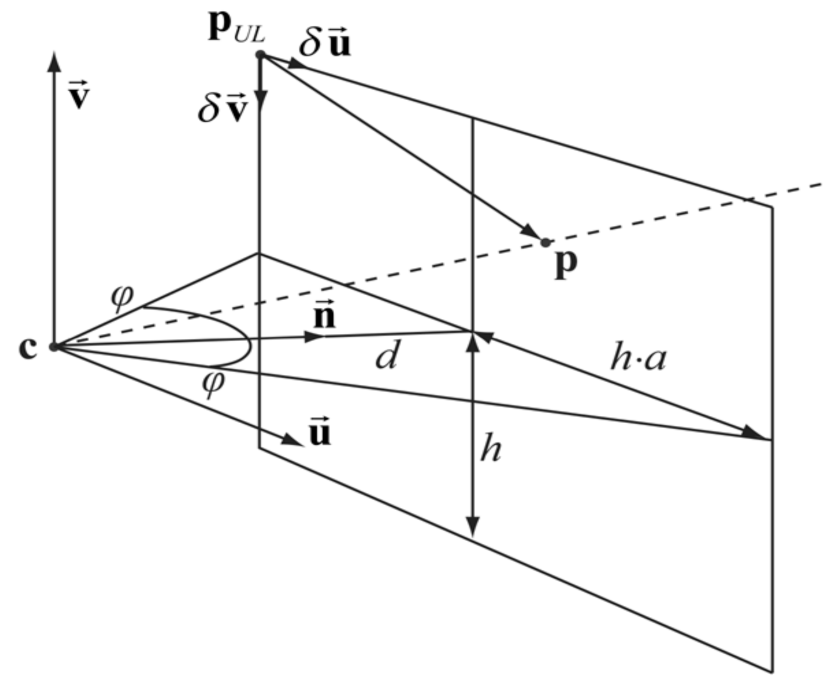
```
...
// Filter the light as it passes through the scene
for ( j=0; j<world.numObjects(); j++ )
   for ( k=0; k<world.getObject(j)->numPrims(); k++ )
   {
      Primitive *prim = world.getObject(j)->getPrim(k);
      if (prim->intersect(r));
         strength *= prim->material->getTransparency(r);

      // Termination criterion: light almost cut off
      if ( strength < 0.002 )
      {
         strength=0;
         break;
      }
   }
...
```

RAY GENERATION

Shooting Rays – Primary Rays (1)





$$w_v = d\tan\varphi \qquad h_v = w_v / a$$

$$\mathbf{p}_{UL} = \mathbf{c} + d\cdot\vec{\mathbf{n}} - w_v\vec{\mathbf{u}} + h_v\vec{\mathbf{v}} \Rightarrow \mathbf{p}_{UL} = \mathbf{c} + d\left[\vec{\mathbf{n}} + \left(\frac{h}{w}\cdot\vec{\mathbf{v}} - \vec{\mathbf{u}}\right)\tan\varphi\right]$$

$$\delta\vec{\mathbf{u}} = \frac{2w_v}{w}\vec{\mathbf{u}} \qquad \delta\vec{\mathbf{v}} = -\frac{2h_v}{h}\vec{\mathbf{v}}$$

Shooting Rays – Primary Rays (2)

- The center of each (i, j) pixel in WCS is:

$$\mathbf{p} = \mathbf{p}_{UL} + \left(i + \frac{1}{2} \right) \delta \vec{\mathbf{u}} + \left(j + \frac{1}{2} \right) \delta \vec{\mathbf{v}}$$

- And the corresponding ray (direction) that passes through it is given by:

$$\vec{\mathbf{r}} = \frac{\mathbf{p} - \mathbf{c}}{\left| \mathbf{p} - \mathbf{c} \right|}$$

Primary Rays - Ray Segment Definition

- Starting point:
  - Either $\mathbf{p} = \mathbf{p}_{UL} + \left(i + \frac{1}{2}\right)\delta\vec{\mathbf{u}} + \left(j + \frac{1}{2}\right)\delta\vec{\mathbf{v}}$ (planar near surface)
  - Or $\mathbf{p}_{start} = \mathbf{c} + n \cdot \vec{\mathbf{r}}$ (Spherical near surface – Can be zero!)
- Arbitrary ray point: $\mathbf{q} = \mathbf{q}(t) = \mathbf{p}_{start} + t \cdot \vec{\mathbf{r}}$
- t is the (signed) distance from the origin as ray vector is normalized

Primary Rays - Clipping Distances

- Depth buffer and perspective projection require a near and a far clipping distance (plane)
- In ray tracing, depth sorting is handled by ray hit sorting so no special ranges are required
- Distance from viewpoint can take arbitrary values (even negative – back of viewer)
- Far clipping distance determined by numerical limits
- Near clipping distance can be zero
- Depth resolution same as floating point precision

Shooting Rays - Secondary Rays

- Origin = Last intersection point
- Direction =
  - Reflected vector
  - Refracted vector
  - Vector to i-th light source: $\vec{\mathbf{r}} = \dfrac{\mathbf{l}_i - \mathbf{q}}{\left\| \mathbf{l}_i - \mathbf{q} \right\|}$

Secondary Rays – Coincidence Pitfall

- Sec. rays can and will intersect with originating surface point (self intersection)
- Fix:
  - Offset the origin along its direction before casting

Reflection Direction

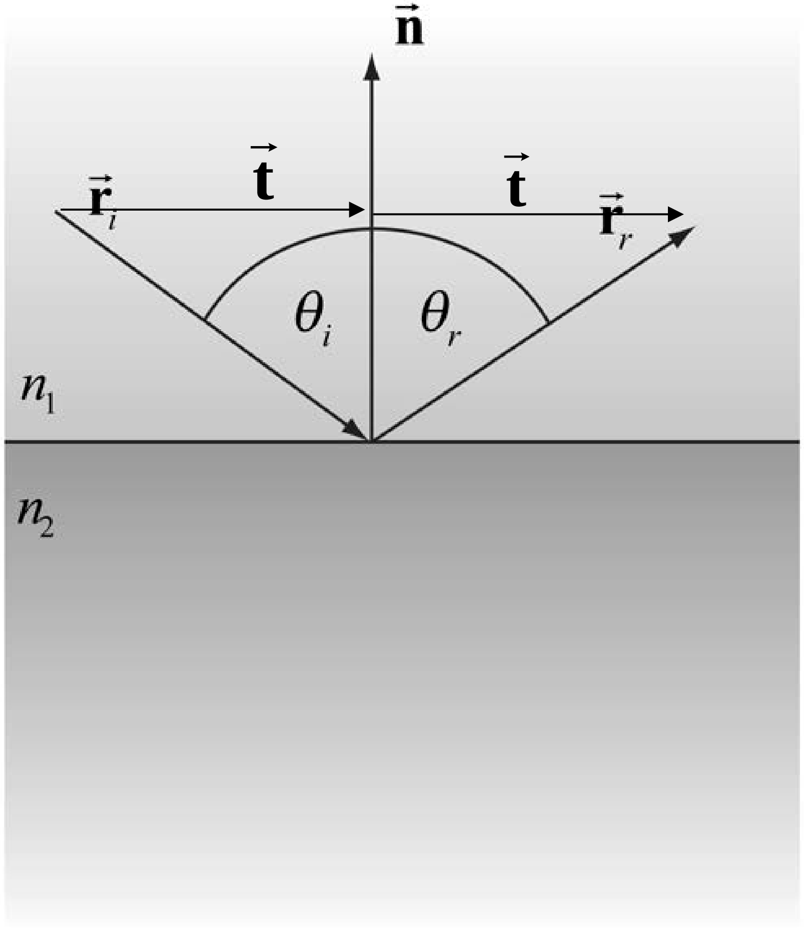
$$\vec{\mathbf{r}}_r = 2\vec{\mathbf{t}} - \vec{\mathbf{r}}_i$$

$$\vec{\mathbf{t}} = \vec{\mathbf{r}}_i + proj_{\vec{\mathbf{n}}}\vec{\mathbf{r}}_i =$$

$$\vec{\mathbf{r}}_i + \vec{\mathbf{n}}\cos\theta_i =$$

$$\vec{\mathbf{r}}_i - \vec{\mathbf{n}}(\vec{\mathbf{n}}\cdot\vec{\mathbf{r}}_i) \Rightarrow$$

$$\vec{\mathbf{r}}_r = \vec{\mathbf{r}}_i - 2\vec{\mathbf{n}}(\vec{\mathbf{n}}\cdot\vec{\mathbf{r}}_i)$$

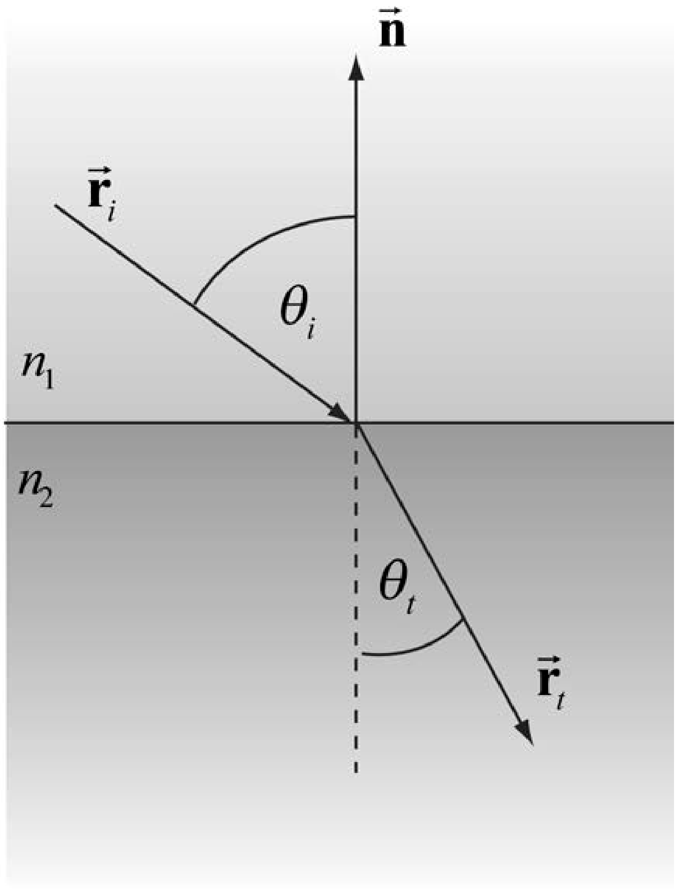


Refraction – Index of Refraction

- When light enters a dielectric medium, its phase velocity changes (const. frequency)
- The ratio of its phase velocity in the medium and c (vacuum) is the IOR n:

  $$n\upsilon = c$$

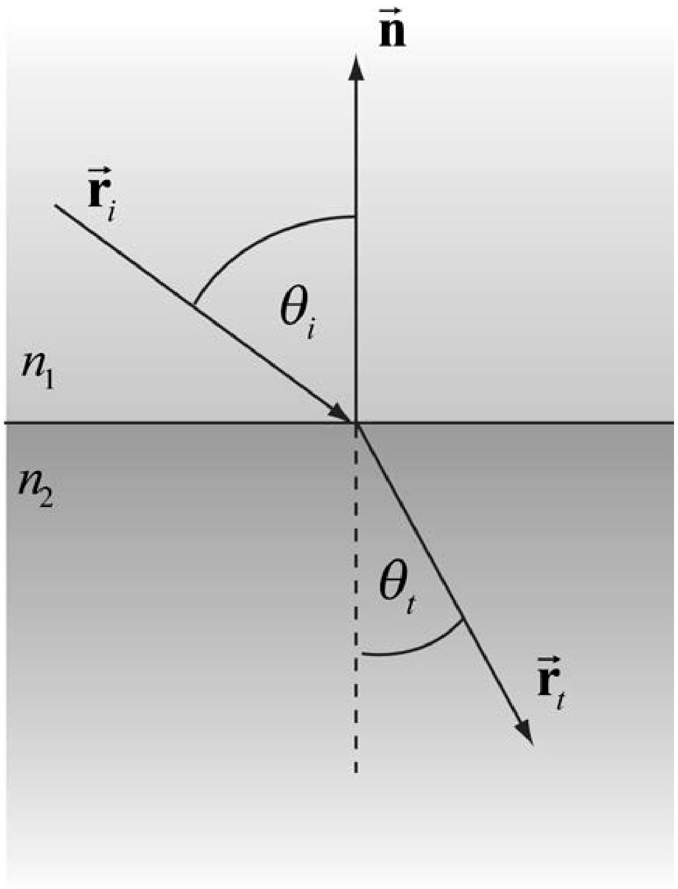
- $n \approx 1$ for thin air
- $n > 1$ for transparent materials



Refraction – Snell's Law

- At the interface between 2 media with IOR n_1 and n_2, the ray is bent according to the law:

$$\frac{\sin\theta_t}{\sin\theta_i} = \frac{n_1}{n_2}$$



Refraction Direction (1)

$$\vec{\mathbf{r}}_t = -\vec{\mathbf{n}}\cos\theta_t - \vec{\mathbf{g}}\sin\theta_t$$

$$\vec{\mathbf{g}} = \frac{\vec{\mathbf{r}}_p}{\sin\theta_i}$$

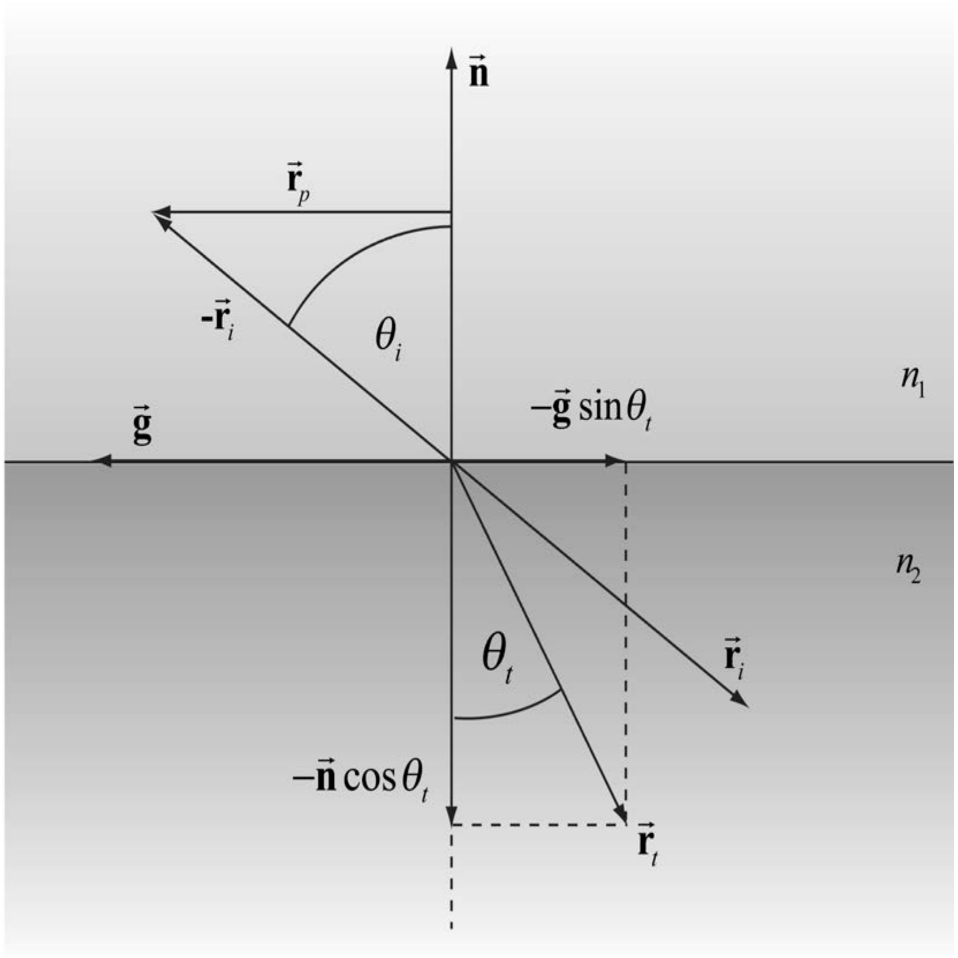
$$\vec{\mathbf{r}}_p = -\vec{\mathbf{r}}_i - \vec{\mathbf{n}}\cos\theta_i =$$

$$-\vec{\mathbf{r}}_i - \vec{\mathbf{n}}\cdot(-\vec{\mathbf{r}}_i\cdot\vec{\mathbf{n}}) =$$

$$-\vec{\mathbf{r}}_i + \vec{\mathbf{n}}(\vec{\mathbf{r}}_i\cdot\vec{\mathbf{n}})$$

$$\vec{\mathbf{g}} = \frac{\vec{\mathbf{r}}_p}{\sin\theta_i} = \frac{-\vec{\mathbf{r}}_i + \vec{\mathbf{n}}(\vec{\mathbf{n}}\cdot\vec{\mathbf{r}}_i)}{\sin\theta_i}$$

$$\vec{\mathbf{r}}_t = -\vec{\mathbf{n}}\cos\theta_t - \left(\vec{\mathbf{n}}(\vec{\mathbf{n}}\cdot\vec{\mathbf{r}}_i) - \vec{\mathbf{r}}_i\right)\frac{\sin\theta_t}{\sin\theta_i}$$



Refraction Direction (2)

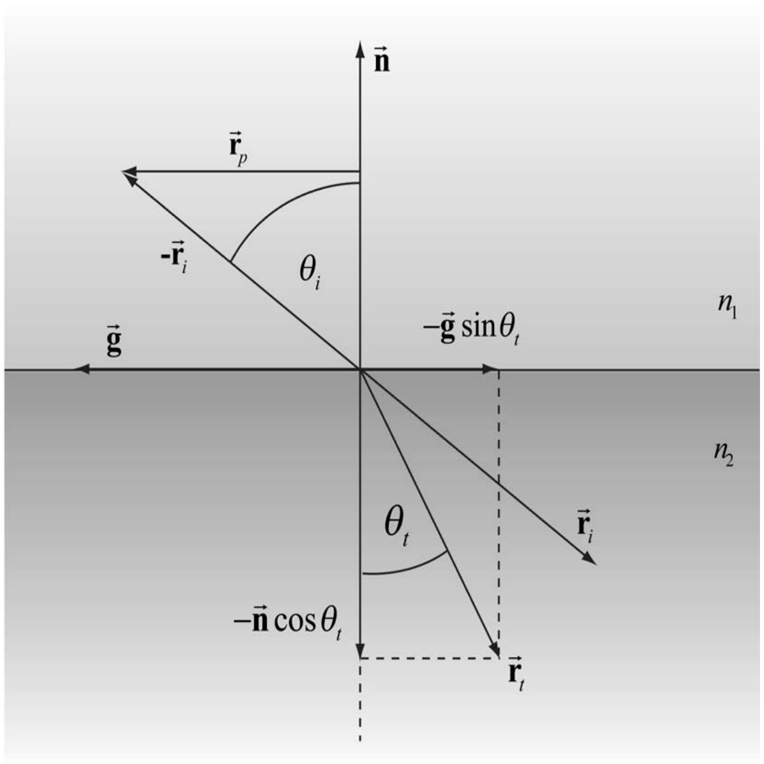
$$\vec{\mathbf{r}}_t = -\vec{\mathbf{n}}\cos\theta_t - \left(\vec{\mathbf{n}}(\vec{\mathbf{n}}\cdot\vec{\mathbf{r}}_i) - \vec{\mathbf{r}}_i\right)\frac{\sin\theta_t}{\sin\theta_i}$$

- From Pythagorean theorem:

$$\cos\theta_t = \sqrt{1-\sin^2\theta_t} =$$

$$\sqrt{1-\frac{n_1^2}{n_2^2}\sin^2\theta_i} =$$

$$\sqrt{1-\frac{n_1^2}{n_2^2}\left(1-\cos^2\theta_i\right)}$$



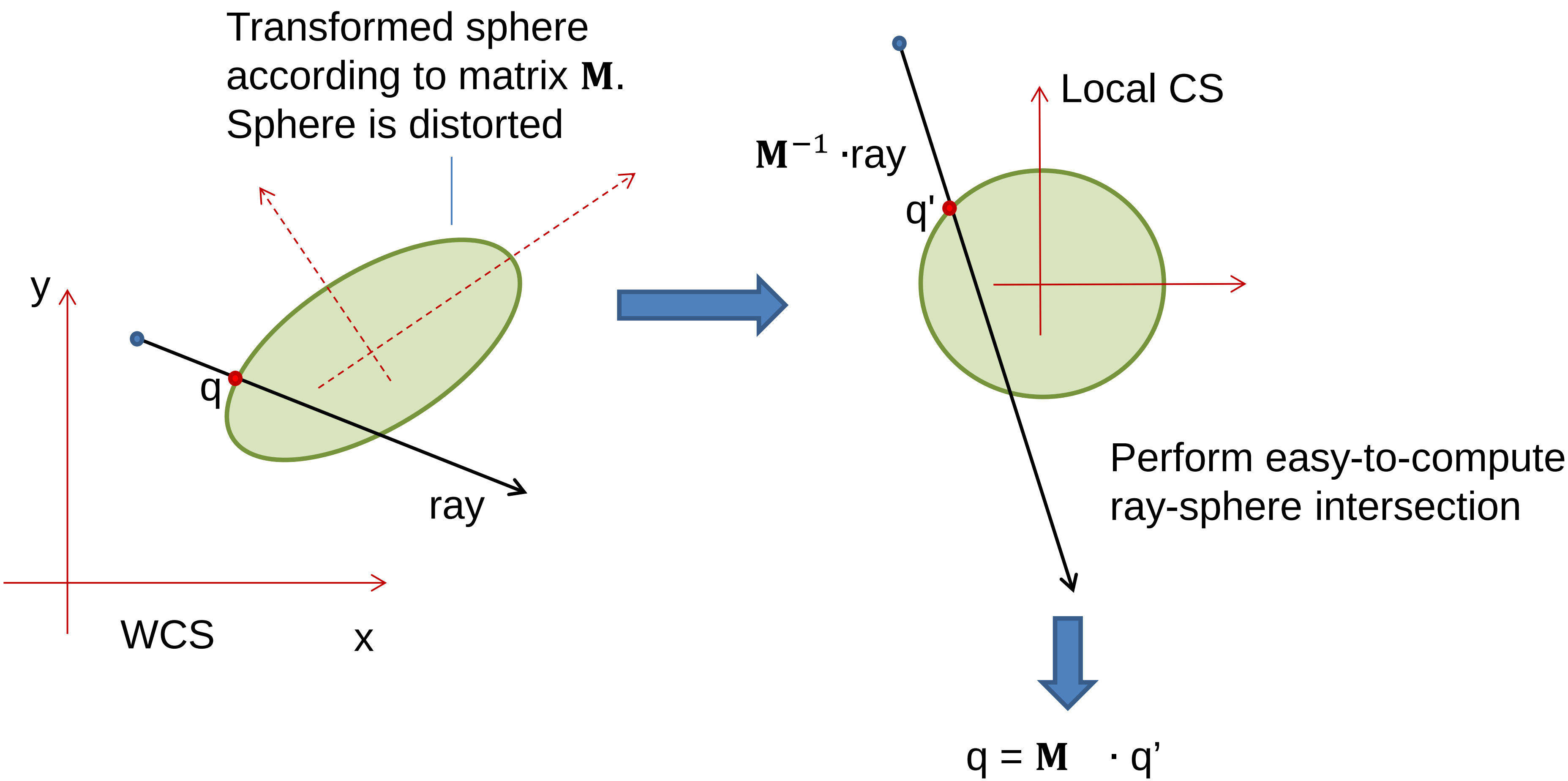
- Using dot product instead of cosine:

$$\vec{\mathbf{r}}_t = \vec{\mathbf{r}}_i\frac{n_1}{n_2} - \vec{\mathbf{n}}\left(\left(\vec{\mathbf{n}}\cdot\vec{\mathbf{r}}_i\right)\frac{n_1}{n_2} + \sqrt{1-\frac{n_1^2}{n_2^2}\left(1-\left(\vec{\mathbf{n}}\cdot\vec{\mathbf{r}}_i\right)^2\right)}\right)$$

Ray Transformations

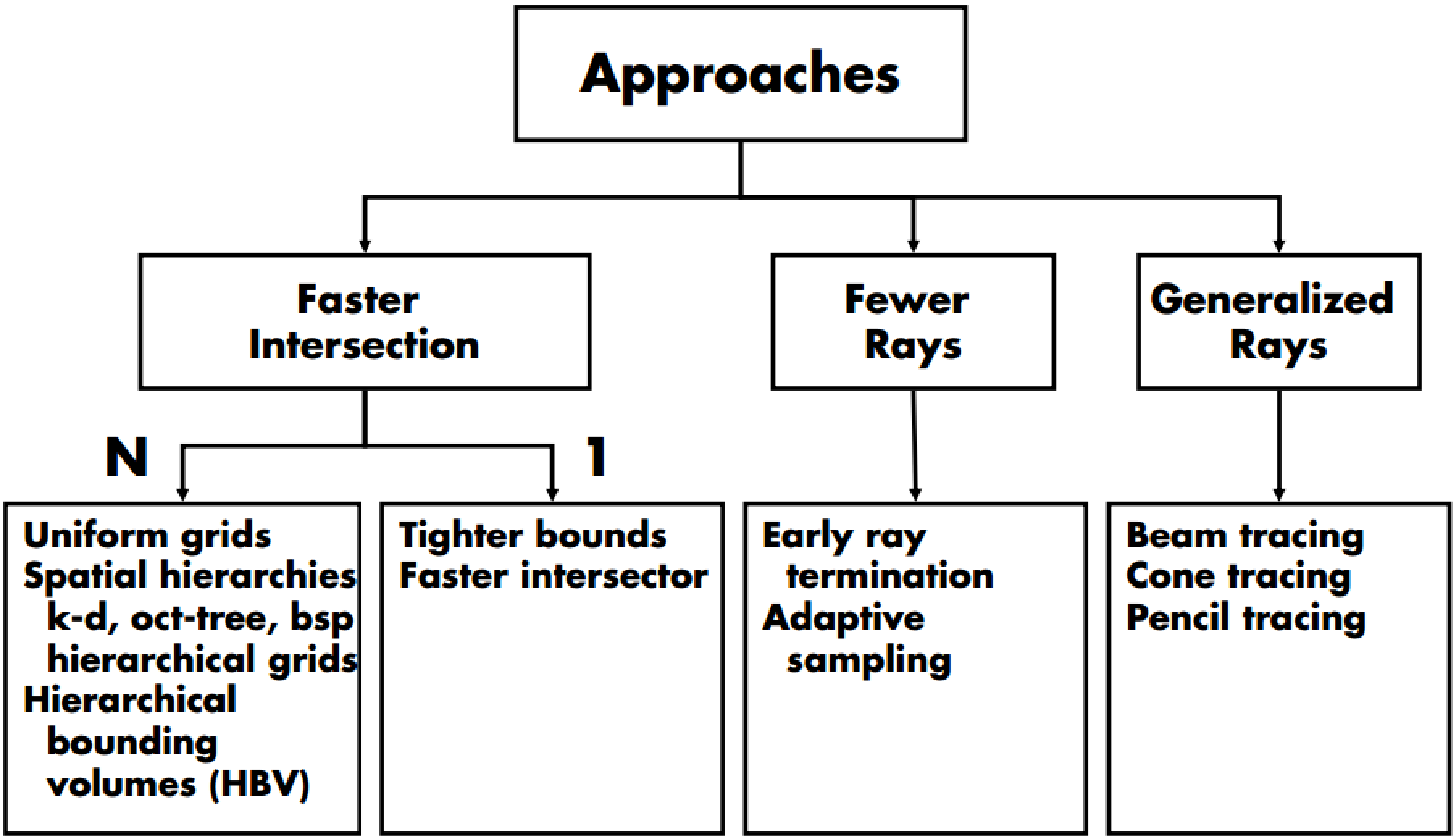
- When rays are intersected with moving geometry, BVH trees, or other elements with parameters defined in a local coordinate system:
  - It is more efficient to transform the ray instead of the object! (why?)
  - Example: OBB/BV hierarchies (common structure for scene graphs)
- $\mathbf{q} = \mathbf{M} \cdot \mathbf{q}' = \mathbf{M} \cdot \text{ Object.RayIntersection}(\mathbf{M}^{-1} \cdot \mathbf{p}, \mathbf{M}^{-1} \cdot \vec{\mathbf{r}})$
  - Ray expressed in the local reference frame
  - The result is expressed back in WCS

Ray Transformations - Example



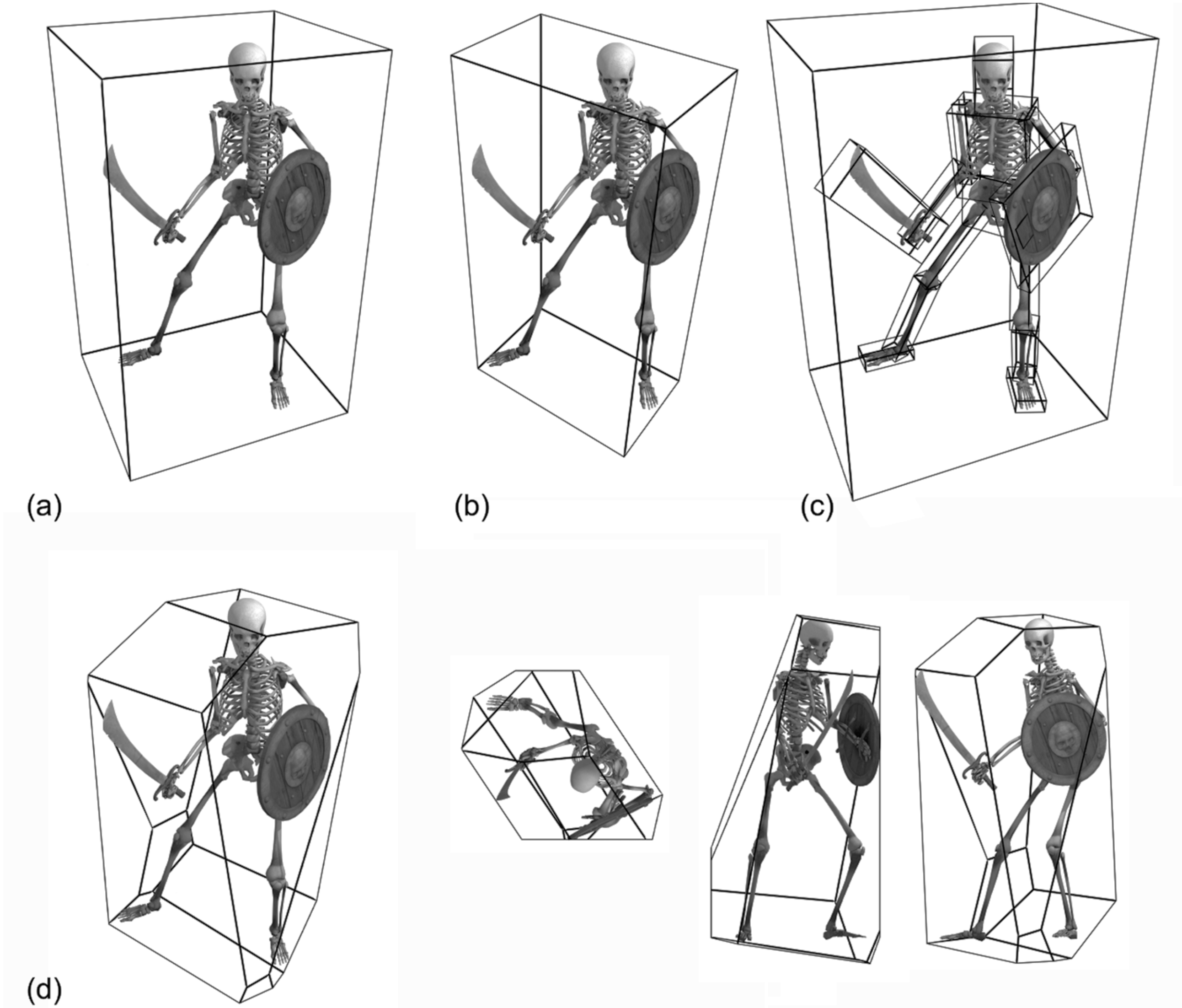
RAY TRACING ACCELERATION TECHNIQUES

Basic Acceleration Concepts



Source: https://graphics.stanford.edu/courses/cs348b-05/lectures/lecture3/raytrace_ii.pdf

Bounding Volumes

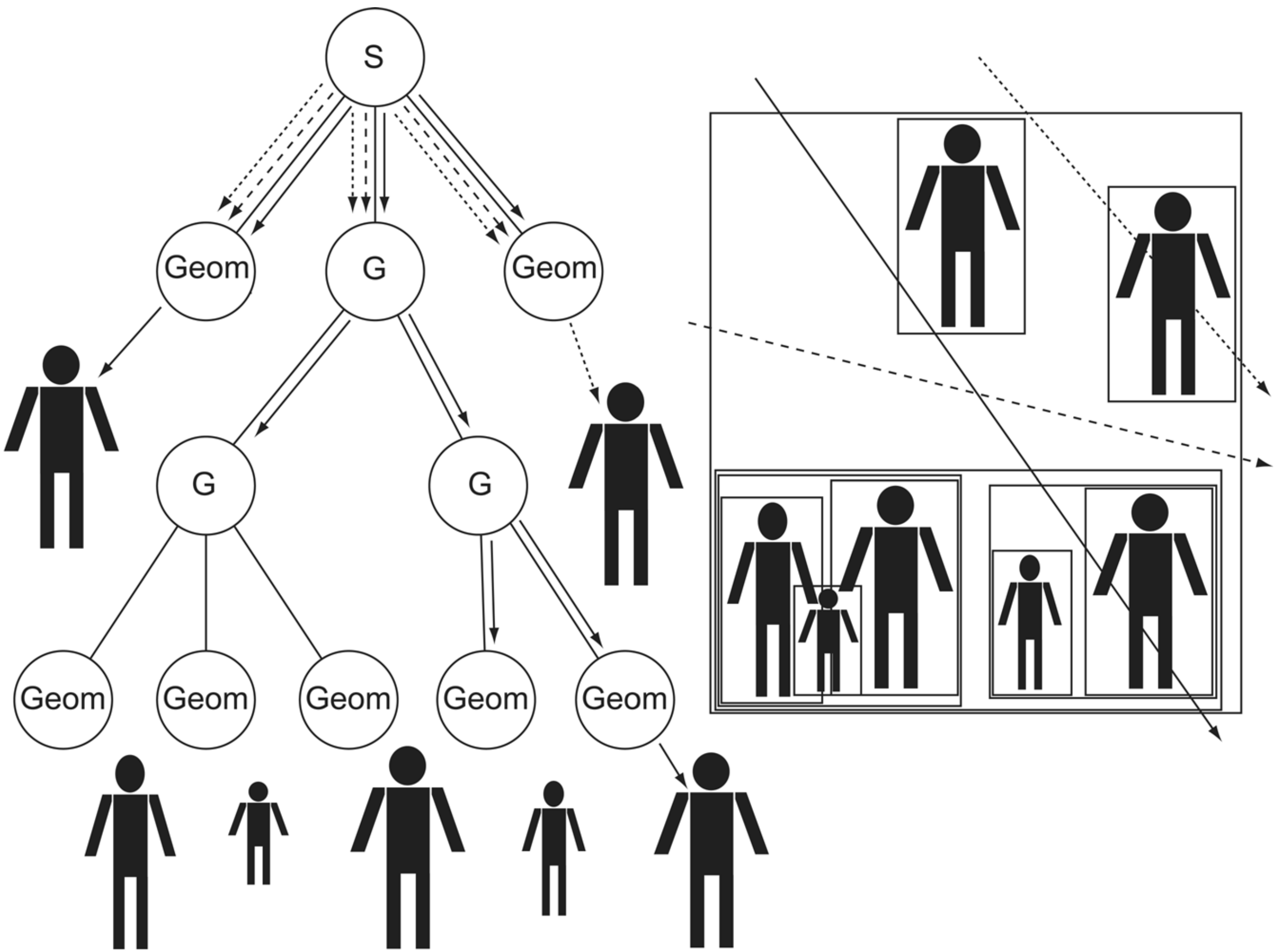


a. Axes-aligned bounding box (AABB)
b. Oriented bounding box (OBB)
c. BV hierarchy (BVH)
d. Bounding slabs

Bounding Volumes – Pros & Cons

- AABB:
  - Easy to implement and initialize
  - Fast test, no ray transformations required
  - Can leave too much void space → degraded pruning performance
- OBB:
  - Can be costly to initialize (e.g. PCA algorithm)
  - Fast test, ray transformation required
  - Ideal for animated hierarchies (no recalculation of extents required)
  - Tighter fitting than AABB
- Bounding Slabs:
  - Very efficient, even less void space
  - More computationally expensive than AABB/OBB

Ray - Scene Graph/BVH Intersection

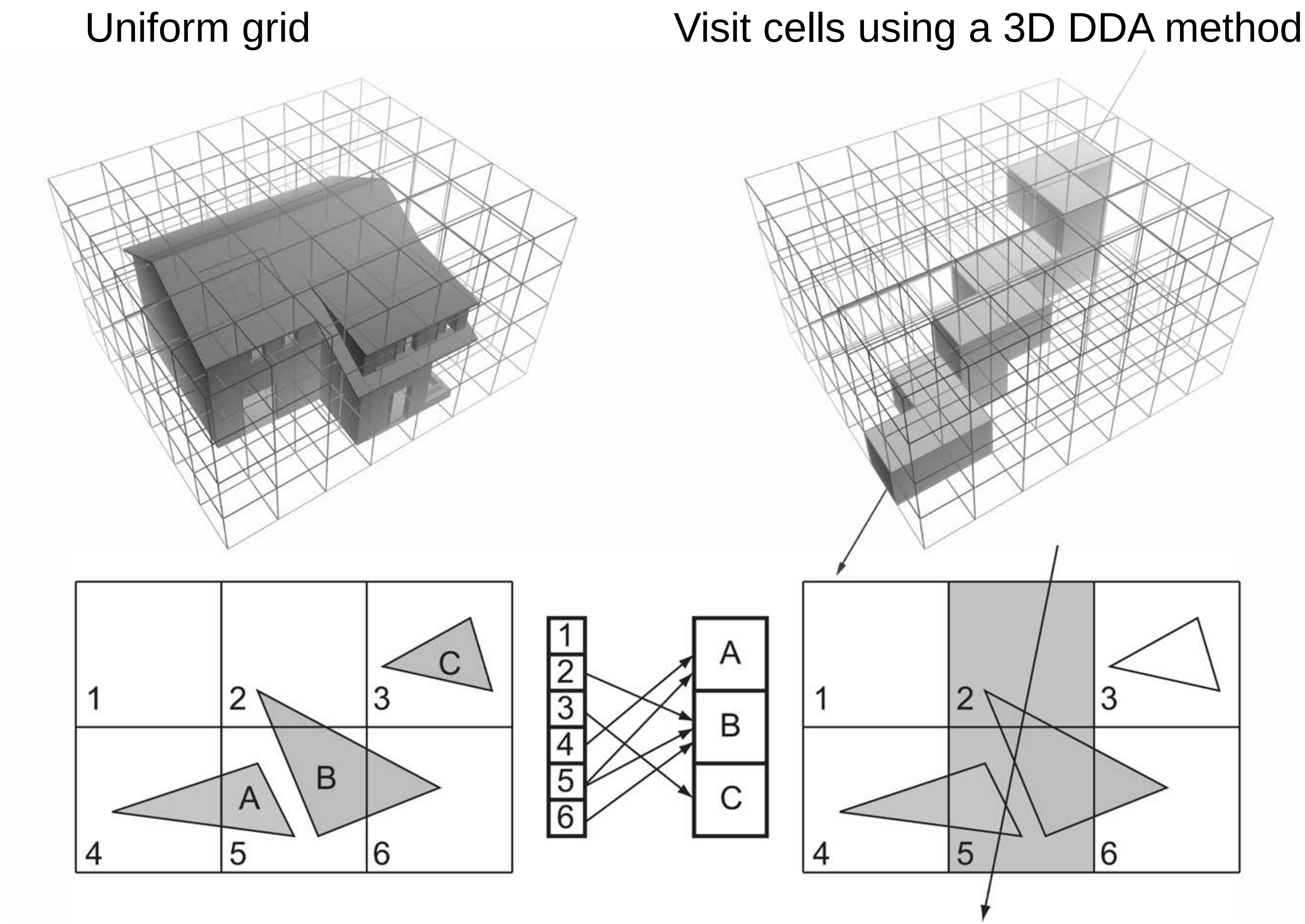


Spatial Subdivision Acceleration (1)

- Primitives can be organized into “bins”, according to rough position in space
- When a ray is cast, it registers the bins it passes through and only tests primitives inside those bins

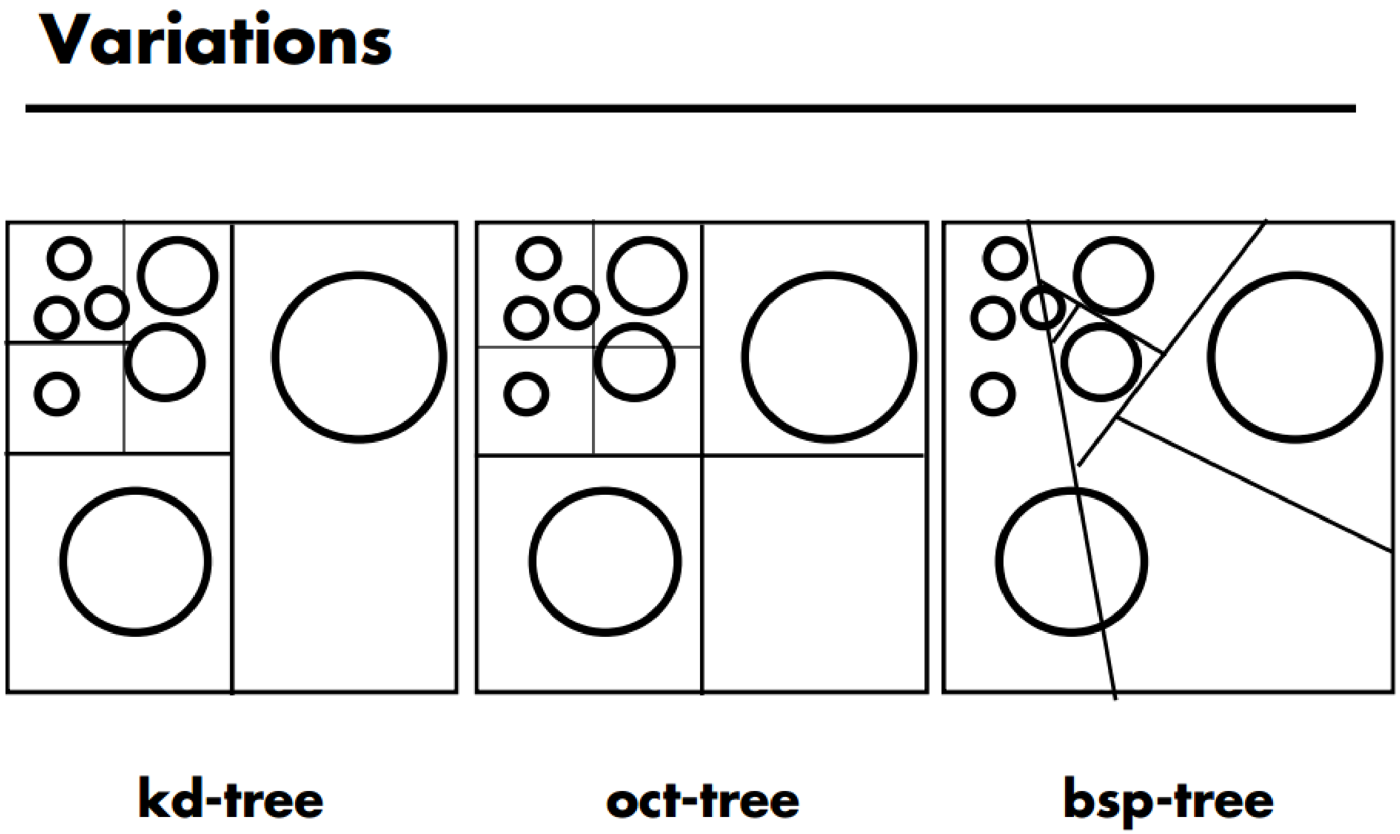
- Spatial subdivision structures can be local to aggregate scene nodes (groups)
- And nested
  - Use ray transformations to go from one local coordinate system to the next

Spatial Subdivision Acceleration (2)



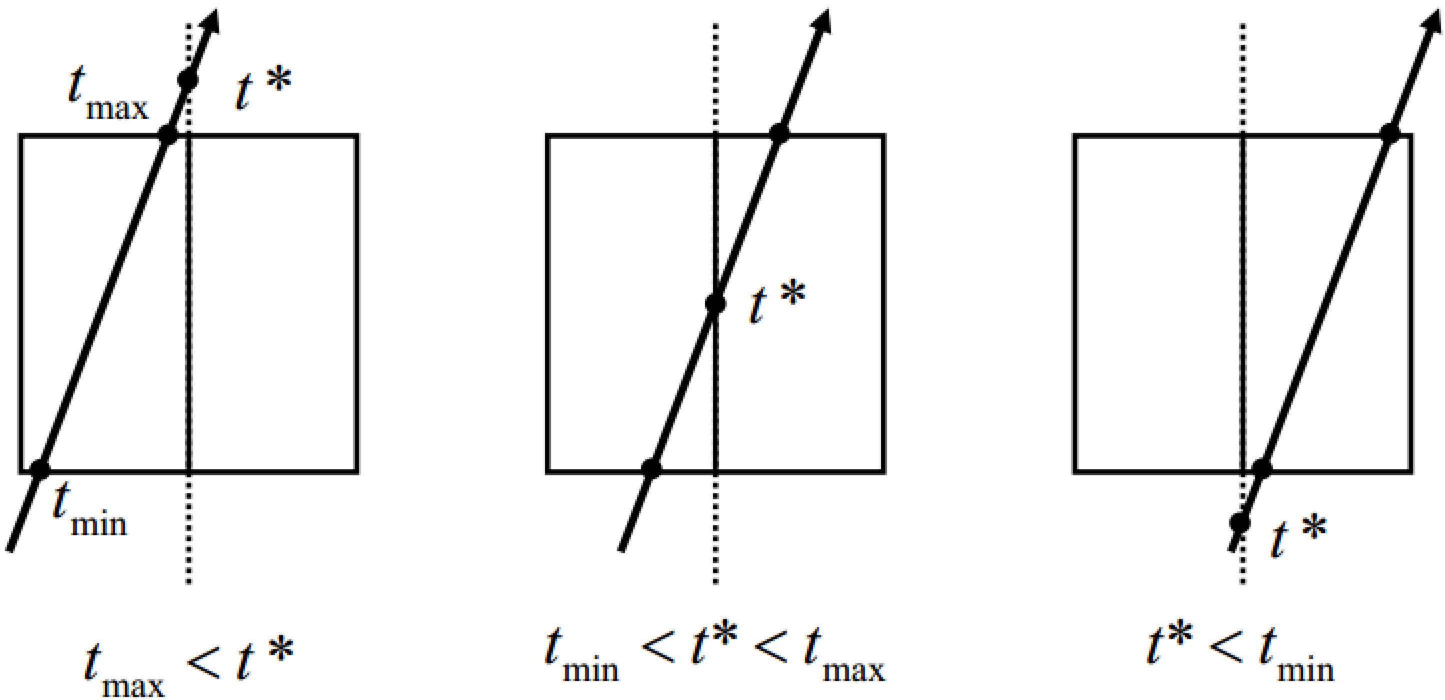
Hierarchical Spatial Subdivision

The spatial subdivision bins (cells) can be hierarchically organized too.



Hierarchical Spatial Subdivision

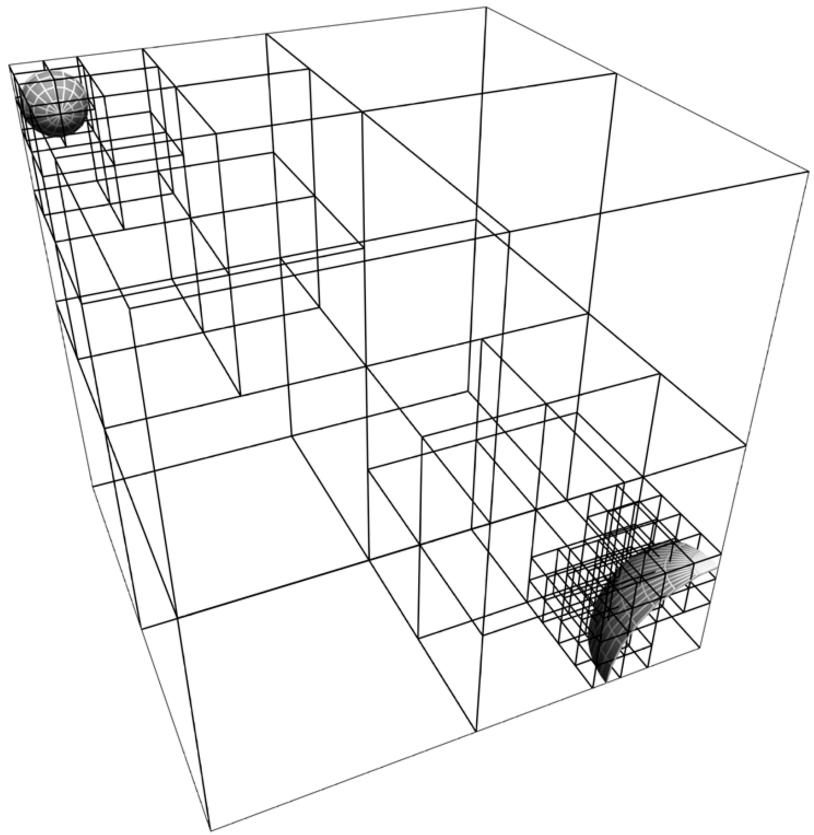
Recursive in-order traversal: rays are tested with subspaces of a splitting plane (binary subdivision)




```
Intersect(L,tmin,tmax)    Intersect(L,tmin,t*)    Intersect(R,tmin,tmax)
                          Intersect(R,t*,tmax)
```

Octree

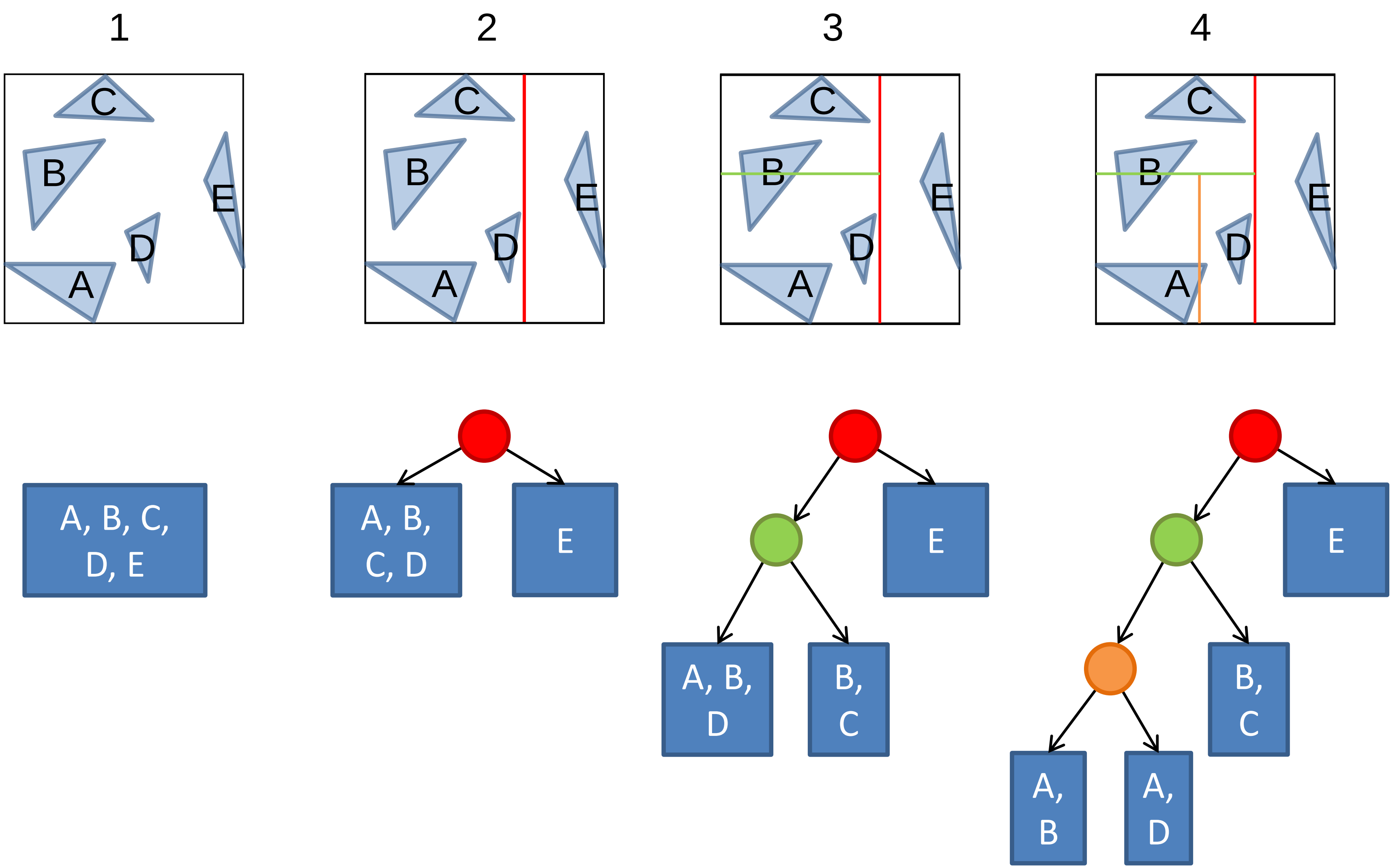
- Common structure is the octree:
- Subdivide space in 8 cells:
  - Up to max depth
  - Until cell contains no primitives



K-d Trees

- Typically K=3 in graphics (3D)
- With the K-d tree, 2 things must be determined at each level:
  - Which axis to split → usually the longest
  - Where to set the split
    - Median cut
    - Midpoint
    - SAH (surface area heuristic)

K-d Tree Construction Example



Complexity Analysis of a Split (1)

- To decide to split a cell, the cost of not splitting it C_{NS} should be greater than the cost of using a split C_S
- For N_O primitives in the cell, each with intersection cost C_O, the cost of using the cell undivided is: $C_{NS} = N_O \cdot C_O$

- The probability that a ray hits a convex shape A completely within another convex shape B is:

$P_A = \frac{SA(A)}{SA(B)}$, where $S(X)$ the surface area

Complexity Analysis of a Split (2)

- Consider only one splitting axis and a parameter $b \in [0,1]$, determining where the split occurs
  - For b=1/2: the spatial median, i.e. in the middle
- The maximum traversal cost (no intersections found, no early termination) of the split cell is the weighted sum of the cost for the two new cells:
- $C_S(b) = P_L(b)N_L(b)C_O + P_R(b)N_R(b)C_O =$

$$\frac{SA(L)}{SA(L \cup R)} N_L(b)C_O + \frac{SA(R)}{SA(L \cup R)} N_R(b)C_O$$

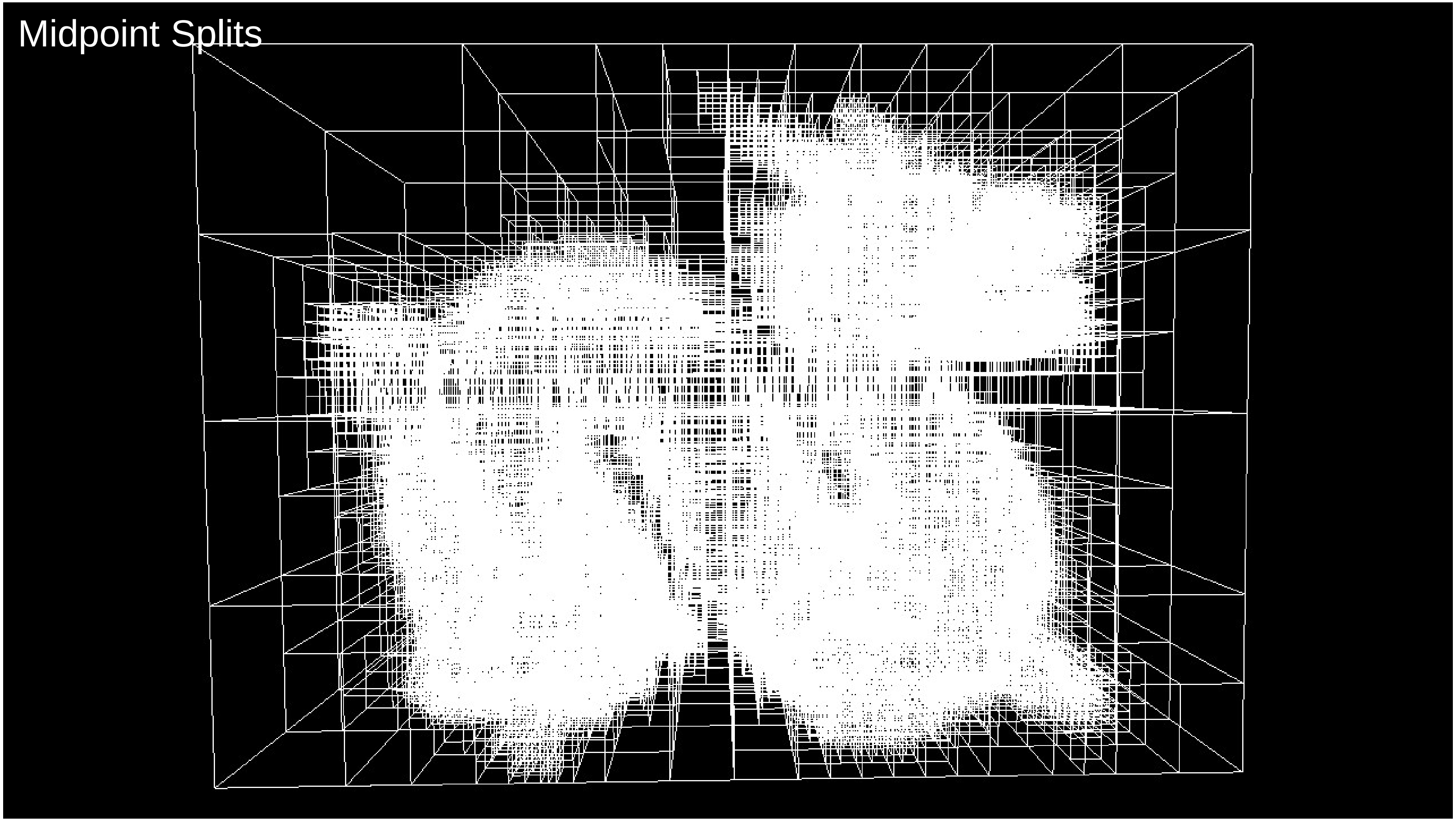
Complexity Analysis of a Split (3)

- $SA(L \cup R)$ is the surface of the un-split cell
- Where $N_L(b), N_R(b)$ are the number of primitives in the left and right part of the subdivided cell
- Note that $N_L(b) + N_R(b) \neq N_O$ in general, as primitives may cross the split boundary

Surface Area Heuristic (1)

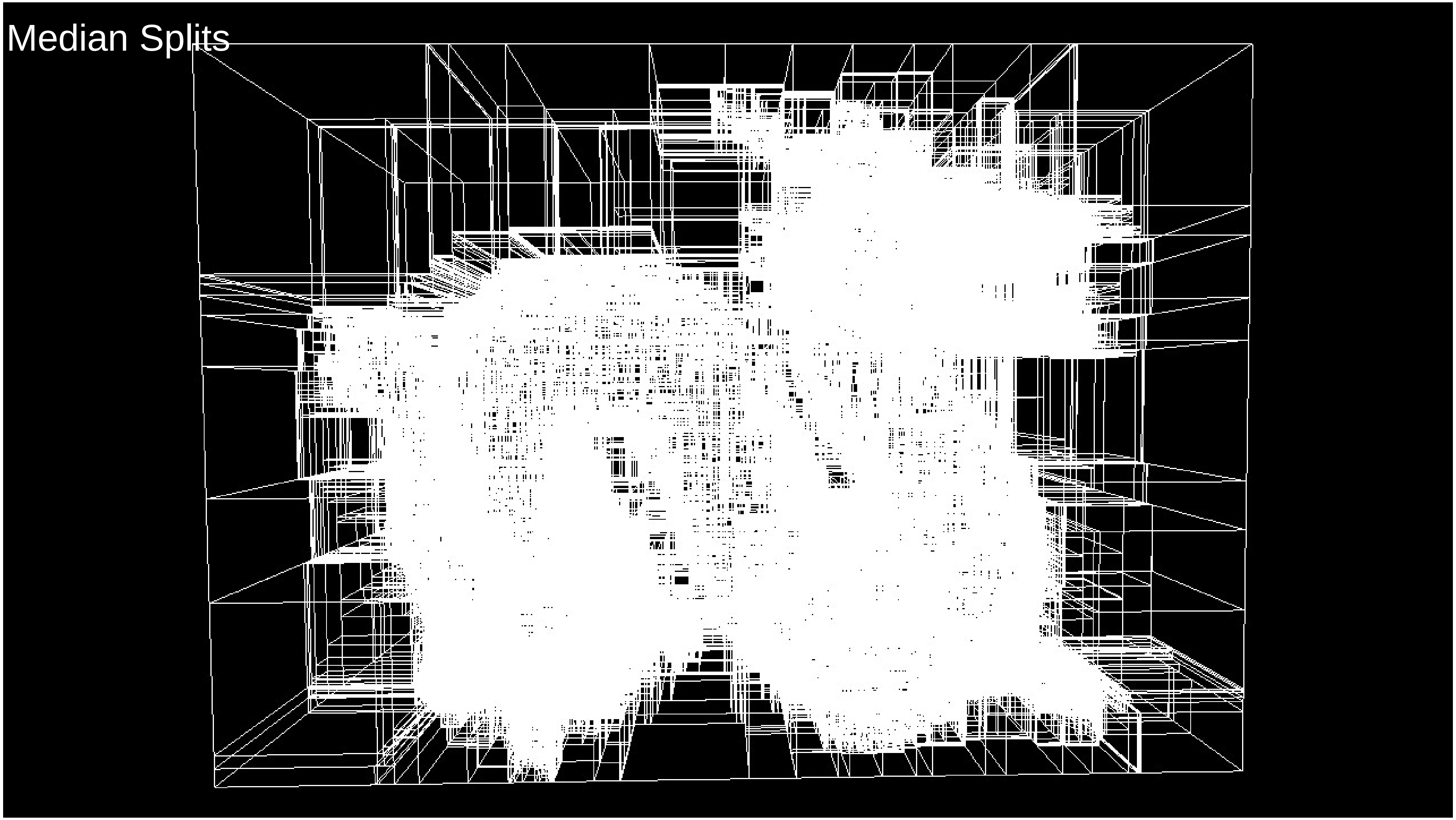
- Determines a splitting plane (and potentially axis, too), by minimizing the above cost function C_S
- Facts:
  - Discontinuous function
  - Optimal cut between spatial median and midpoint
- Two options:
  - Sort primitive bounds per axis, locate median and test bounds between median and midpoint
  - Greedily test all bounds
- Number of bounds: $2N_O$ or $6N_O$ for concurrent axis selection

Surface Area Heuristic (2)



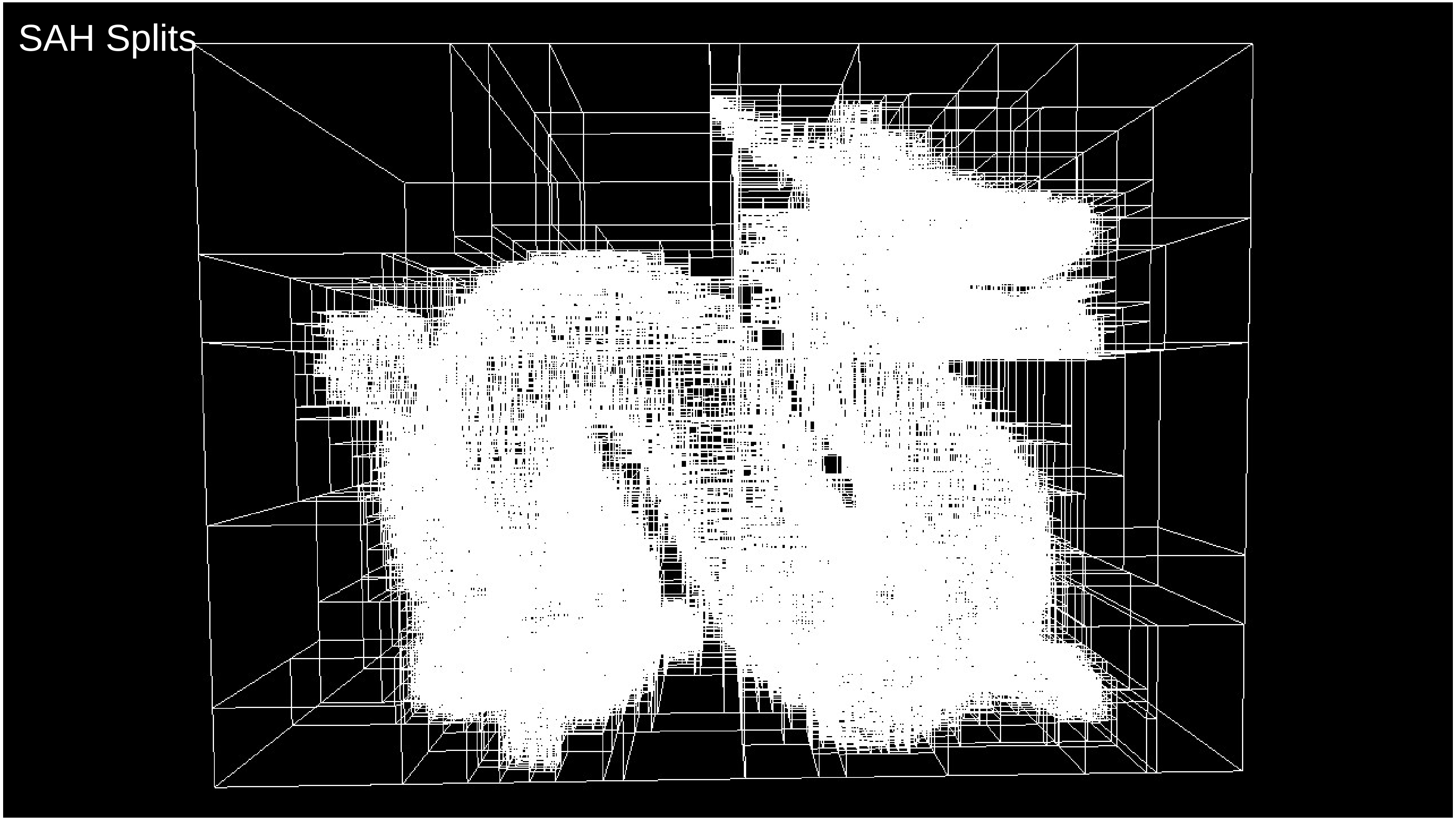
http://www.keithlantz.net/2013/04/kd-tree-construction-using-the-surface-area-heuristic-stack-based-traversal-and-the-hyperplane-separation-theorem/

Surface Area Heuristic (3)



http://www.keithlantz.net/2013/04/kd-tree-construction-using-the-surface-area-heuristic-stack-based-traversal-and-the-hyperplane-separation-theorem/

Surface Area Heuristic (4)



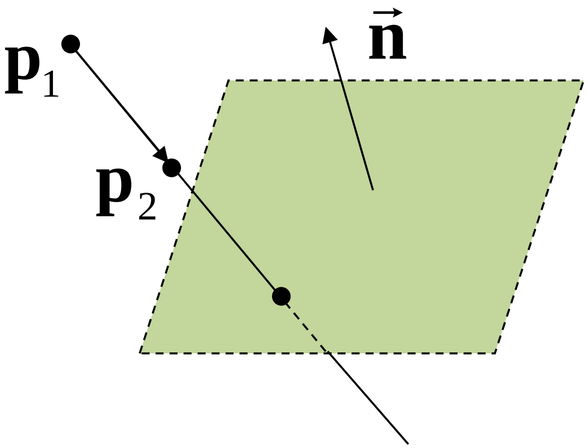
http://www.keithlantz.net/2013/04/kd-tree-construction-using-the-surface-area-heuristic-stack-based-traversal-and-the-hyperplane-separation-theorem/

INTERSECTION TESTS

Intersection Tests: Ray - Plane

- If the plane equation is: $\overrightarrow{\mathbf{n}} \cdot \mathbf{p} + d = 0$
- We substitute point $\mathbf{p}$ by the line definition:

$$\mathbf{p}(t) = \mathbf{p}_1 + t\left(\mathbf{p}_2 - \mathbf{p}_1\right)$$



- So: $\overrightarrow{\mathbf{n}} \cdot \left(\mathbf{p}_1 + t\left(\mathbf{p}_2 - \mathbf{p}_1\right)\right) + d = 0$

$$t = -\frac{\overrightarrow{\mathbf{n}} \cdot \mathbf{p}_1 + d}{\overrightarrow{\mathbf{n}} \cdot \left(\mathbf{p}_2 - \mathbf{p}_1\right)}$$

- If instead of $\left(\mathbf{p}_2 - \mathbf{p}_1\right)$ we use a normalized vector, t is the signed distance along the ray

Intersection Tests: Ray - Triangle (1)

- *Barycentric* triangle coordinates:
  - Any point in the triangle can be expressed as a weighted sum of the triangle vertices (affine combination):

$$\mathbf{q}(u,v,w) = w\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2,$$

$$u + v + w = 1$$

Intersection Tests: Ray - Triangle (2)

- Requiring intersection point in triangle:

$$\mathbf{p} + t\vec{\mathbf{d}} \boxed{=} (1-u-v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$$

- And in the form of a linear system (3 unknowns):

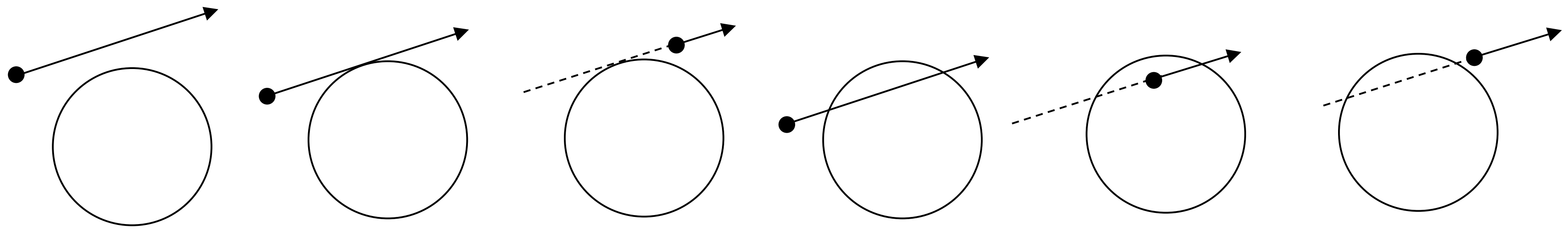
$$\begin{bmatrix} -\vec{\mathbf{d}} & \mathbf{v}_1 - \mathbf{v}_0 & \mathbf{v}_2 - \mathbf{v}_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \begin{bmatrix} \mathbf{p} - \mathbf{v}_0 \end{bmatrix}$$

- We solve it for t, u and v
- If u, v and 1- u - $v \leq 1$, then hit point inside triangle

- See [RTI] for an efficient implementation of the above

Intersection Tests: Ray – Sphere (1)

- A Ray intersects a sphere if:
  - Line – sphere equation has 1 root and $0 \leq t$ (otherwise the ray points away from the sphere)
  - Line – sphere equation has 2 roots:
    - 2 negative: ray points away (no intersection)
    - 1 positive, 1 negative: positive root defines the intersection point
    - 2 positive roots, smallest one corresponds to entry point



Intersection Tests: Ray – Sphere (2)

- Combining the sphere parametric equation $(\mathbf{p}-\mathbf{c})\cdot(\mathbf{p}-\mathbf{c})=r^2$ with the line parametric equation: $\mathbf{p}(t)=\mathbf{p}_1+t\,\overrightarrow{\mathbf{d}}$ we get:

$$(\mathbf{p}_1+t\,\overrightarrow{\mathbf{d}}-\mathbf{c})\cdot(\mathbf{p}_1+t\,\overrightarrow{\mathbf{d}}-\mathbf{c})=r^2$$

$$\Leftrightarrow(\overrightarrow{\mathbf{m}}+t\,\overrightarrow{\mathbf{d}})\cdot(\overrightarrow{\mathbf{m}}+t\,\overrightarrow{\mathbf{d}})=r^2$$

$$\Leftrightarrow(\overrightarrow{\mathbf{d}}\cdot\overrightarrow{\mathbf{d}})t^2+2(\overrightarrow{\mathbf{m}}\cdot\overrightarrow{\mathbf{d}})t+(\overrightarrow{\mathbf{m}}\cdot\overrightarrow{\mathbf{m}})-r^2=0$$

where $\overrightarrow{\mathbf{m}}=\mathbf{p}_1-\mathbf{c}$ is a vector from the center of the sphere to the ray origin

Intersection Tests: Ray – Sphere (3)

- This is a normal quadratic equation for t of the form:
  $at^2 + 2bt + c = 0$

where: $a = \vec{\mathbf{d}} \cdot \vec{\mathbf{d}}, \quad b = \vec{\mathbf{m}} \cdot \vec{\mathbf{d}}, \quad c = \vec{\mathbf{m}} \cdot \vec{\mathbf{m}} - r^2$

- The discriminant $b^2 - ac$ specifies the roots and corresponding intersection points:
  - D<0: No intersection
  - D=0: One intersection
  - D>0: 2 intersection points: $t = \dfrac{-b \pm \sqrt{D}}{a}$

Deficiencies of Simple Ray Tracing

- Marginally interactive method, even with optimizations only for simple scenes
- Extremely (and unnaturally) crisp and polished images
  - Ideal specular (mirror) reflection and transmission
  - Natural surfaces and media are not "ideal"
- No other light transport event is modelled

Contributors

- Georgios Papaioannou

References

[RTI]: Fast, Minimum Storage Ray/Triangle Intersection , Möller & Trumbore. Journal of Graphics Tools, 1997