Graphics & Visualization

Chapter 15

Ray Tracing
Introduction

- Direct-rendering algorithms:
  - Sorting is performed in image space (Z-buffer)
  - Object-to-screen space image synthesizers

- *Ray tracing* is a general algorithm that operates in the opposite manner:
  - It is a screen-to-object space image synthesizer

- In *ray tracing*, we follow rays along the line of sight passing through each pixel as they travel through the 3-D scene → registers what the observer sees along this direction

- As ray encounters geometric entities:
  - Specularly reflected, refracted, or attenuated (completely absorbed)
Introduction (2)

- Hidden surface elimination happens as part of this process:
  - Ray encounters surface interfaces closer to the viewer first while it travels through the 3-D world
- Simple direct rendering relies on local shading models:
  - Shadows & reflected/refracted light on surfaces need to be simulated separately & fused as color information in the local illumination model used during scan conversion
- Ray tracing, integrates all calculations that involve specular transmission of light in a single & elegant recursive algorithm, the *recursive ray tracing* algorithm
Principles of Ray Tracing

• Light transmission:
  - An infinite number of rays emanate from a light source, and a small number reach the eye after following complex paths within the scene.

• Light is diffusely scattered and specularly reflected or refracted.
• Part of the specularly & diffusely reflected light is directly received by the observer.
Principles of Ray Tracing (2)

- Light reaches the observer indirectly as well:
  - Following paths through transparent media or by being reflected off perfect mirrors

- But:
  - Given a very large but finite number of rays starting from the light sources, it is statistically unlikely that they will hit the image pixels and contribute to the final result
  - We must find a different sampling mechanism that ensures that the image pixels are crossed by the light paths and thus adequately sampled
Principles of Ray Tracing (3)

- In ray tracing: Light seen through a pixel of the rendered image is the cumulative contribution of the rays that directly or indirectly hit the surface point visible in this direction and that travel toward the viewpoint.

- Nearest point encountered by looking at the scene though pixel \((i, j)\) in general obstructs all other geometry behind it:
  - Point may or may not be directly illuminated by the light source(s), depending on whether other geometry prevents the light from reaching it.
  - Paths reaching intersection point from other directions traveling toward pixel \((i, j)\) can be followed to discover what light they contribute.

- This is possible due to the reciprocity of light propagation:
  - Light follows same path during refraction or perfect reflection on a material interface regardless of the direction of propagation.
Principles of Ray Tracing (4)

- If it is directly lit by the light source → local illumination model can be applied
- Cumulative illumination visible through a frame-buffer pixel \((i,j)\) due to the contribution of direct and indirect rays:
Since we are only interested in those rays that eventually reach the viewpoint through a viewport pixel → Trace back the light contributions by following the rays in the opposite direction of their propagation toward the source.

For each ray back to its source → evaluate the light propagated toward the viewer by applying a local illumination model & re-investigating for other secondary rays that reach that point:

- This is exactly the mechanism of ray tracing
- A computationally manageable problem
Principles of Ray Tracing (6)

- Compared to direct-rendering algorithms, ray tracing has 2 significant advantages:
  - Ray-geometry intersections can be directly performed using non-polygonal surfaces, such as geometric solids, implicit or parametric surfaces, and fractals, without requiring any conversion to polygons first → Mathematical surfaces that can be intersected by a ray can be rendered
  - Reflection & refraction phenomena can be accurately modeled
Ray Diversion

Reflection

Refraction
Reflection

- For an arbitrary ray of light from a direction $\hat{r}_i$ incident on a perfectly reflecting interface between 2 bodies, the reflected ray in the perfect mirror-reflection direction $\hat{r}_r$ is:

$$\hat{r}_r = \hat{r}_i - 2 \hat{n} (\hat{n} \cdot \hat{r}_i) \quad (15.1)$$

- Notice that here the incident direction is the opposite of the light direction vector $\hat{i}$ since we need to emphasize the direction of propagation for clarity.
Refraction

- **Simple index of refraction** (or **refractive index**) \( n \): Ratio between speed of light \( c \) in a vacuum & the phase velocity of light \( u \) in this medium:

\[
n = \frac{c}{u}
\]  
(15.2)

- \( n > 1 \): Transparent materials & \( n \approx 1 \): Air
- \( n \) depends on wavelength \( \lambda \) of the light \( \rightarrow n = n(\lambda) \)
  - For visible light: \( n \downarrow \) when wavelength \( \uparrow \)
- Phase velocity is responsible for the bending of the propagation direction as the light crosses the interface between them
- According to **Snell's law**: \( \frac{\sin \theta_t}{\sin \theta_i} = \frac{n_1}{n_2} \)  
(15.3)
  - Light entering a medium with larger index of refraction \( (n_2 > n_1) \) is bent toward the normal direction of the optically denser medium
When \( n_2 < n_1 \) total internal reflection may occur

- Light not transmitted through the boundary but reflected back

- Min angle of incidence at which total internal reflection occurs is called a **critical angle** \( \theta_c \):

\[
\theta_c = \arcsin \left( \frac{n_2}{n_1} \right) \quad (15.4)
\]
Refraction (3)

• Calculation of the direction of the new, transmitted ray:

\[ \hat{r}_t = -\hat{n}\cos\theta_t - \hat{g}\sin\theta_t, \quad (15.5) \]

where \( \hat{g} \) : unit length vector \( \| \hat{r}_p \) \ and:

\[ \hat{r}_p = -\hat{r}_i - \hat{n}\cos\theta_i = -\hat{r}_i - \hat{n}( -\hat{r}_i \cdot \hat{n}) = -\hat{r}_i + \hat{n}(\hat{r}_i \cdot \hat{n}) \quad (15.6) \]

• \( \hat{r}_i \) is a unit vector \( \rightarrow \) length of \( \hat{r}_p = \sin\theta_i \)

• After normalizing it, we get:

\[ \hat{g} = \frac{\hat{r}_p}{\sin\theta_i} = -\hat{r}_i + \hat{n}(\hat{n} \cdot \hat{r}_i) \quad (15.7) \]

• Replacing \( \hat{g} \) from (15.7) into (15.5):

\[ \hat{r}_t = -\hat{n}\cos\theta_t - (\hat{n}(\hat{n} \cdot \hat{r}_i) - \hat{r}_i)\frac{\sin\theta_t}{\sin\theta_i} \quad (15.8) \]
Refraction (4)

- From (15.3) we can replace the sines in the above relation with the indices of refraction.
- Also, using Pythagorean trigonometric identity \( \cos \theta_i = \sqrt{1 - \sin^2 \theta_i} \) we get:
  \[
  \cos \theta_i = \sqrt{1 - \sin^2 \theta_i} = \sqrt{1 - \frac{n_1^2}{n_2^2} \sin^2 \theta_i} = \sqrt{1 - \frac{n_1^2}{n_2^2} \left(1 - \cos^2 \theta_i\right)}
  \]  
  \[
  \text{(15.9)}
  \]
- Introducing these relations in (15.8):
  \[
  \hat{r}_i = -\hat{n} \sqrt{1 - \frac{n_1^2}{n_2^2} \left(1 - \cos^2 \theta_i\right)} - \left(\hat{n} \cdot \hat{n} \hat{r}_i - \hat{r}_i\right) \frac{n_1}{n_2}
  \]
  \[
  \text{(15.10)}
  \]
- Final step \( \rightarrow \) replace cosine with corresponding inner product:
  \[
  \hat{r}_i = -\hat{n} \sqrt{1 - \frac{n_1^2}{n_2^2} \left(1 - (\hat{n} \cdot \hat{n} \hat{r}_i)^2\right)} - \left(\hat{n} (\hat{n} \hat{r}_i) - \hat{r}_i\right) \frac{n_1}{n_2} = \hat{r}_i \frac{n_1}{n_2} - \hat{n} \left((\hat{n} \cdot \hat{n} \hat{r}_i) \frac{n_1}{n_2} + \sqrt{1 - \frac{n_1^2}{n_2^2} \left(1 - (\hat{n} \cdot \hat{n} \hat{r}_i)^2\right)}\right)
  \]
  \[
  \text{(15.11)}
  \]
Note that quantity inside the radical of (15.11) is \( > 0 \):
- If \( < 0 \) \(\rightarrow\) total internal refraction & new ray is calculated from (15.1)
- Refracted ray calculation:
Reflectance & Transmittance

• Snell's law & the law of reflection do not provide an insight into the intensity distribution between reflected and refracted waves:
  ■ Amount of light that is reflected off an interface between materials with indices of refraction $n_1$ & $n_2$ is given by the Fresnel equations

• Fresnel equations provide reflection & refraction coefficients for light crossing the boundary between 2 dielectrics:
  ■ Correspond to the ratio between the amplitude of the reflected or transmitted electric field & the amplitude of the incident electric field
Reflectance & Transmittance

- Light is an electromagnetic field \( \rightarrow \) electric & magnetic fields are oscillating perpendicularly to the direction of propagation
- Electric (magnetic) field can be decomposed into 1 component parallel & 1 perpendicular to the plane of reflection
- Fresnel provided 2 equations for reflection coefficient \( r_p \) & \( r_s \) as well for the transmission coefficient \( t_p \) & \( t_s \) for the case of parallel and perpendicular polarization, respectively:

\[
\begin{align*}
  r_s &= \frac{n_1 \cos \theta_i - n_2 \cos \theta_t}{n_1 \cos \theta_i + n_2 \cos \theta_t}, \\
  r_p &= \frac{n_1 \cos \theta_t - n_2 \cos \theta_i}{n_1 \cos \theta_t + n_2 \cos \theta_i}, \\
  t_s &= \frac{2n_1 \cos \theta_i}{n_1 \cos \theta_i + n_2 \cos \theta_t}, \\
  t_p &= \frac{2n_1 \cos \theta_i}{n_1 \cos \theta_t + n_2 \cos \theta_i}.
\end{align*}
\]
Reflectance & Transmittance (2)

- Index of refraction depends on the wavelength of the light:
  - Reflection and refraction coefficients depend on incident angle and wavelength of the incoming ray
- Fresnel formulas for wave intensity can be derived by squaring (15.12):
  \[ R_s = r_s^2, \quad R_p = r_p^2, \quad T_s = t_s^2, \quad T_p = t_p^2 \]  
  (15.13)
- Intensity is flux per unit area & the incoming beam is spread or shrunk according to the relation between the refractive indices:
  - Should not expected that \( T_s = 1 - R_s \) or \( T_p = 1 - R_p \)
Reflectance & Transmittance (3)

- Exact oscillation direction & polarization of the incident wave is seldom considered in computer graphics → when the Fresnel reflection model is applied, the average reflection and refraction coefficients can be used:

\[
R = \frac{R_s + R_p}{2}, \quad T = \frac{T_s + T_p}{2}
\]  

(15.14)
Recursive Ray Tracing Algorithm

- Principle of algorithm:
  - For each pixel, a *primary ray* is created starting from viewpoint & passing through the center of the pixel
  - Ray is tested against the scene geometry → find the closest intersection with respect to the starting point
  - A successful hit is detected → local illumination model applied → determine the color of the point
  - Otherwise → color returned is the background color
  - If material of the surface hit is transparent → refracted ray is spawned
  - If surface is reflective → *secondary ray* also spawned toward the mirror-reflection direction
  - Both secondary rays (reflected, refracted) are treated the same way as the primary ray → cast & intersected with the scene
  - When & if they hit a surface → local illumination model is applied → new rays are potentially spawned e.t.c
Recursive Ray Tracing Algorithm (2)

- Recursive re-spawning of rays & their tracing through the scene
Recursive Ray Tracing Algorithm (3)

- Each time a ray hits a surface, a local color is estimated:
  - Color is the (+) of illumination from the local shading model as well as the contributions of refracted & reflected rays spawned at this point.
  - Each recursion returns cumulative color estimated from this level & below.
  - This color is added to the local color according to reflection & refraction coefficients & propagated to the higher recursion step.
  - Color returned after exiting all recursion steps is the final pixel color.

![Diagram showing recursive ray tracing algorithm](image)
Recursive Ray Tracing Algorithm (4)

- The depth of the recursion is controlled primarily by 3 factors:
  1. The ray hits a surface with no transparency or reflective quality $\Rightarrow$ no new rays generated
  2. A ray's contribution drops significantly $\Rightarrow$ no point in continuing to accumulate light on this particular path through the scene
  3. Prevent an uncontrollable spawning of rays in highly reflective or elaborate transparent environments $\Rightarrow$ max ray-tracing depth is usually defined
Recursive Ray Tracing Algorithm (4)

- Comparison between renderings with different ray-tracing depth:
  - Impact of maximum ray-tracing depth on the rendered image accuracy
Recursive Ray Tracing Algorithm (5)

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

    // Terminate the procedure if max recursion depth has been reached
    if (depth > MAXDEPTH) return backgroundColor;

    // Intersect ray with scene & 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 & 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); delete refl;
    }
    if (r->isect->surface->material->k_refr > 0) {
        tran = calculateRefraction(r);
        color_t = raytrace(tran, depth+1, world, lights); delete tran;
    }

    return color_l + color_r + color_t;
}
```
Light is attenuated:
- Reflection and refraction coefficients
- Potential distance attenuation that we may applied to the rays
- Absorption
- Scattering as it travels through a dense body

Ray needs to keep track of its “strength” $\rightarrow$ properly modulate contributed local color at the intersection point $\rightarrow$

Facilitate the ray importance termination criterion for the recursion
RT Data Structures – Distance

• Ray tested for intersection with multiple surfaces → many intersection points are identified:
  ■ Ray structure must keep track of the closest point to the ray origin → able to compare it with the next intersection that may occur while an iterative ray-primitive intersection test is performed

• Keep distance between currently closest hit & ray origin:
  ■ Compare it with distance to the next intersection point
  ■ Useful in the case of distance or volumetric attenuation calculations
RT Data Structures – Intersected Point

• Intersection point is not a simple point in space in ray tracing:
  ■ Used in calculations involving normal vector at this location, reflection &
    refraction coefficients, other material properties
  ■ Intersection is identified \(\rightarrow\) number of parameters must be passed to ray

• For the intersection point we could keep:
  ■ Local normal,
  ■ Texture coordinates,
  ■ Reference to the material that is valid for this particular location
  ■ Reference to the primitive where the intersection point belongs \(\rightarrow\) able to
    retrieve additional information

• In ray tracing in polygonal scenes the ray could keep:
  ■ Intersection point ,
  ■ Distance,
  ■ Reference to the polygon, and
  ■ Set of barycentric coordinates \(\rightarrow\) derive attributes from vertex information
class Ray
{
public:
    IsectPoint *isect;
    int level;
    Vector4f origin;
    Vector3f dir;
    float strength; ... 
    //Methods
    transform (Matrix4X4 mat); ... 
}

class IsectPoint : Vector4f
{
public:
    Vector3f n; //Local normal
    Primitive *surface; //Intersected primitive
    double barycentric[3]; //for triangular meshes
    double t; //parametric distance between
               //origin and intersection point
}
Ray World Intersection

- **Problem**: Search for the closest intersection point is exhaustive:
  - Without intersection acceleration, rays are tested against the whole database of the scene at a primitive level
- Ray-primitive intersection tests are the most frequent operations in a ray tracer
- Ray tracing cost: exhaustive and repetitive nature of search for intersection points
- But: Ray tracing is also trivially parallel!
- Highly optimized intersection tests for different types of primitives have been proposed
Ray World Intersection (2)

- Usually, we assume a generic primitive class of type Primitive.
  - Provides a common intersection interface for all sub-classes of geometric primitives
- An unoptimized function for ray-world intersection point detection
  - Performs an exhaustive iteration of all objects
  - Exhaustively intersects the ray with each primitive in each object
  - Returns closest intersection and number of hits
int findClosestIntersection(Ray r, Scene world) {
    int hits=0; r.isect = new IsectPoint();
    r.isect->t = 10000000;    //A large intersection distance
    for ( j=0; j<world.numObjects(); j++ ) {
        for ( k=0; k<world.getObject(j)->numPrims(); k++ ) {
            Primitive *prim = world.getObject(j)->getPrim(k);
            IsectPoint *Q = prim->isect(r);
            if (Q==NULL)
                continue;
            hits++;
            //If found closer intersection, copy it in r
            if ( r.isect->t > Q->t ) r.isect->copy(Q);
        }
    }
    return hits;
}
Local Illumination Model & Shadows

• Evaluate a local illumination model at point of ray-surface intersection. For every light source in the scene:
  ■ Send a shadow ray to light source and determine visibility
  ■ If light can be either completely blocked or completely visible, the contribution of the particular light drops to 0 when an intersection is found
  ■ Objects are not always fully opaque \(\rightarrow\) color is filtered as shadow ray encounters the surfaces:
    ■ Contribution of the light is diminished at each intersection (ray strength)
    ■ If strength < threshold \(\rightarrow\) terminate the search for further obstacles in the shadow ray's path

• Shadow rays can be computed faster:
  ■ No sorting of intersected points
  ■ Interrupt intersection tests as soon as strength becomes too low.
Local Color Calculation Code

```cpp
Color calculateLocalColor( Ray r, Vector<Light*> lights, Scene world ) {
  int i, j, k;
  Color col = ambientColor(); //Initialize color to min illumination
  for ( i=0; i<lights.size(); i++ ) {
    Ray *shadowRay = new Ray(r->isect, lights[i]->pos);
    //Measure how much light reaches the intersection
    float penetration=1.0f;
    for ( j=0; j<world.numObjects(); j++ )
      for ( k=0; k<world.getObject(j)->numPrims(); k++ ) {
        Primitive *prim = world.getObject(j)->getPrim(k);
        IsectPoint *Q = prim->isect(r);
        //Case 1: ray not blocked by prim: no attenuation
        if (Q==NULL) continue;
        //Case 2: light contribution is filtered
        penetration *= 1 - prim->material->alpha;
        if ( penetration < 0.02 ) { //Termination: light almost cut off
          penetration=0;
          break;
        }
      } // for all objects
    //Check if light[i] contributes to local illumination
    if (penetration==0) continue;
    col += localShadingModel( r, prim, lights[i]->pos, penetration );
  } //light[i]
  return col;
}
```
Shooting Rays: Primary Rays

- Many ways to determine primary rays shot toward each pixel
- Here: calculation for an arbitrary camera coord. system \((\hat{n}, \hat{u}, \hat{v})\) and a symmetrical view frustum centered at the optical axis

**Assume:**
- Pinhole camera model
- Square pixels
- Focal distance \(d\)
- Aspect ratio \(a = w/h\)
- \(w, h\) the width, height of the image (pixels)
- \(w_v, h_v\) the half-width/height of the view window at near clipping distance in world coordinates:

\[
w_v = d \tan \varphi, \quad h_v = \frac{w_v}{a} \quad (15.15)
\]
Shooting Rays: Primary Rays (2)

- Main loop in a ray tracer iterates through all image pixels \((i, j)\) and casts at least one ray from each one
  - Due to this iterative procedure \(\rightarrow\) convenient to formulate the calculation of the ray starting point \(p\) and direction \(\hat{r}\) in incremental manner
- Calculate point \(p_{UL}\) corresponding to the upper-left corner of the image
- Calculate incremental offsets \(\delta \hat{u}\) and \(\delta \hat{v}\), between successive pixels in world coordinates
- Point \(p_{UL}\) is calculated by adding an offset along view direction to the camera center \(c\) & moving across view window plane to the upper-left corner: \(p_{UL} = c + d \cdot \hat{n} - w_v \hat{u} + h_v \hat{v}\) \hspace{1cm} (15.16)
  or using (15.15):
  \[
  p_{UL} = c + d \left[ \hat{n} + \left( \frac{h}{w} \hat{v} - \hat{u} \right) \tan \phi \right]
  \] \hspace{1cm} (15.17)
Shooting Rays: Primary Rays (3)

- Incremental offsets $\delta \vec{u}$ & $\delta \vec{v}$ depend on the resolution of the image in each direction:
  $$\delta \vec{u} = \frac{2w_v}{w} \hat{u}, \quad \delta \vec{v} = -\frac{2h_v}{h} \hat{v} \quad (15.18)$$

- Square pixels assumed $\rightarrow$ image resolution only affects aspect ratio of horizontal vs vertical view aperture & pixel size, but not pixel shape:
  $$|\delta \vec{u}| = \frac{2w_v}{w} |\hat{u}| = \frac{2ah_v}{w} = \frac{2h_v}{h} = |\delta \vec{v}| \quad (15.19)$$

- Use center of the pixel as origin $\mathbf{p}$ of the ray, then for $i = 0..w-1$ & $j = 0..h-1$:
  $$\mathbf{p} = \mathbf{p}_{UL} + \left( i + \frac{1}{2} \right) \delta \vec{u} + \left( j + \frac{1}{2} \right) \delta \vec{v} \quad (15.20)$$

- Ray direction vector is the normalized difference between origin & camera focal point:
  $$\hat{\mathbf{r}} = \frac{\mathbf{p} - \mathbf{c}}{|\mathbf{p} - \mathbf{c}|} \quad (15.21)$$
Shooting Rays: Clipping

• In the direct rendering case (Z-buffer):
  ■ Ratio between near & far clipping distances has a significant impact on the accuracy of the depth sorting and a 0 near-distance is not allowed

• In ray tracing:
  ■ Near clipping plane can be set to the origin & far clipping plane to \( \infty \) without any side effect

• Distance sorting in ray-world intersection compares last and current distance of intersection point from the origin of the ray

• Given a parametric representation of a semi-infinite ray, a point along its path is defined as:

\[
q = q(t) = p_{\text{start}} + t \cdot \hat{r} \quad \text{(15.22)}
\]
Shooting Rays: Clipping (2)

• Ray vector is considered normalized $\rightarrow t$ is the signed distance along the ray from its starting point

• If $p_{\text{start}}$ lies on near clipping surface, intersections $q(t)$ with $t < 0$ are disregarded as invisible

• For planar clipping surface model, the focal length $d$: $d > 0$ defines near clipping distance & $p_{\text{start}} = p$

• For arbitrary clipping distance $n$ from the focal point, we get ray starting points on a spherical clipping surface:

$$p_{\text{start}} = c + n\hat{r}$$  \hspace{1cm} (15.23)
Shooting Rays: Secondary Rays

- Secondary rays are cast according to the direction of:
  - reflection,
  - refraction or
  - direct illumination

  depending on whether the ray path is followed due to reflection, transmission, or shadow test, respectively

- Starting point for those rays is always the intersection point of the previous recursion step
Shooting Rays: Secondary Rays (2)

• **Important observation:** Rays emanating from a surface point are prone to intersect with it again, unless we find a way to exclude this point from the procedure.

• **Easy test:**
  - Check if $t$ at the intersection is $> 0$
  - Or
    - Check if in the case of $t == 0$, the primitive is other than the primitive of ray origin.
class Ray {
    public: ...
        Primitive * startPrim;
    ...
}

int findClosestIntersection(Ray r, Scene world) {
    ...
    if (Q==NULL)
        continue;
    if (Q->t<0 || (nearZero(Q->t) && r.startPrim==prim))
        continue;
    hits++;
    // if found closer intersection, copy it in r
    if(r.isect->t>Q->t)        r.isect->copy(Q);
    ...
}
Hierarchical Intersection Tests

- Geometry is organized in object hierarchies →
  - NOT exhaustively searching for intersections with the primitives BUT first test the ray with scene management hierarchy, a spatial subdivision hierarchy, or a combined scheme
- **Idea:** First perform a computationally efficient intersection test with a simple volume that bounds a cluster of primitives:
  - Simple solids (boxes & spheres) were utilized for this purpose
  - Most common types of bounding volumes used for ray-tracing acceleration are spheres, AABBs, OBBs & bounding slabs
- **OBBs** can fit significantly more tightly to the original object with a careful selection of the box orientation:
  - If 3 mutually perpendicular pairs of parallel planes of OBB are replaced by arbitrary number of parallel planes → object enclosed in a set of *bounding slabs* ensuring even less void space inside bounding volume
Hierarchical Intersection Tests (2)

- Common bounding volumes for ray tracing:
  (a) AABB       (b) OBB       (c) Bounding volume hierarchy
  (d) Bounding slabs
Hierarchical Intersection Tests (3)

- Scene organized as scene graph → bounding volume of each node provide a preliminary crude intersection rejection test for geometry contained
- On a positive bounding volume-ray hit → test recursively applied to children nodes
- At leaf level: Geometry primitives exhaustively tested for intersection or ray passed to a space subdivision structure for further early primitive rejection processing
- Intersection tests with AABBs and bounding spheres are inexpensive
- For OBBs → transform rays to bring them to local coordinate system of the bounding volume & perform the test as for AABB
Hierarchical Intersection Tests (4)

- Intersection of a ray & a bounding volume hierarchy:
  - Most intersection tests prevented by simple ray-bounding volume tests
Hierarchical Intersection Tests (5)

- **Factor that affects efficiency: Amount of void space:**
  - Scene organization with large bounding volumes at high levels \(\Rightarrow\) leaves lot of void space between actual geometry elements \(\Rightarrow\) many false hits
  - **Solution:** Tighter object-aligned bounding slab can be more efficient for volume testing than a large AABB

- Rays are hierarchically pruned most effectively if the bounding volume has as small a surface area as possible

- # of rays hitting a bounding volume is not the sole criterion for selection of particular type of container:
  - Computational complexity of intersecting the ray with the solid plays a significant during a typical rendering
Spatial Subdivision

• Different approach to speed up ray tracing

• Scene space is decimated into a large number of simple cells (often AABs) and each one references the primitives it intersects

• For each ray:
  ■ Find intersected cells
  ■ Test ray against contained primitives of above cells

• Cells can be hierarchically organized
  ■ Split cells into smaller ones until a subdivision criterion is met
  ■ Ray enters a cell → recursively tested against smaller cells

• Important benefit: if cells are visited in an ordered manner (direction of the ray) → sorting is performed at a container level

• Nearest intersection within a cell found → intersection traversal terminates!
• Extra preprocessing time is necessary in order to build & fill data structures representing the containers → taken into account when rendering frame sequences with many objects

• Dynamic scenes require the recalculation and update of the acceleration data structures of the space-partitioning scheme

• Simplest form of space partitioning for ray tracing is a regular subdivision of the space occupied by the primitives into voxels:
  - First all primitives pre-processed to determine which voxels they intersect
  - A reference to a particular primitive is created in all cells intersected by it
  - During ray casting, hit voxels are identified and their contents tested for intersections
  - Selection of voxels for each ray is done with an incremental algorithm similar to the 2D DDA, only for voxel space instead of image space
Spatial Subdivision (3)

- Subdivision for acceleration of ray-tracing intersection tests:

  (Left) A voxel space is generated around the scene

  (Middle) Primitives are indexed according to which voxels they intersect

  (Right) Ray is tested against primitives indexed by the voxels it passes through
Spatial Subdivision (4)

- **Mailboxing**: Acceleration technique which makes intersection tests for penetrating rays more efficient:
  - A unique ray identifier is stored in each intersected primitive
  - If a primitive spans more than one voxel → is intersected only once:
    - Ray identifier is compared to the one stored in the primitive before attempting to calculate the intersection
Resolution of the voxel space plays an important role for the performance of the uniform spatial subdivision method:

- Large cells lead to fewer intersected voxels $\rightarrow$ less redundant intersection tests
- Smaller voxels reduce the number of primitives indexed by each one of them $\rightarrow$ faster intra-voxel intersection searches

Voxels can be hierarchically refined:

- Attempt to create cells with a balanced number of referenced primitives

Most common hierarchical space-partitioning organization for ray tracing uses an octree
Octrees

- Space of a top-level cell is subdivided into 8 equally-sized voxels
- Voxels that contain no primitives are not subdivided further
- Voxels that contain primitives are split in the same manner
- Partitioning stops either when a max # of subdivisions is reached or when # of primitives a cell contains is too small
- Max # of subdivisions performed defines the depth of the tree
- In contrast to the case of regular space partitioning, ray-octree data structure intersection tests are unbalanced, but intersection-test distribution at the leaf nodes is more even
Octrees (2)
Ray Transformations

- A ray may need to intersect an OBB volume aligned with an arbitrary set of axes
- Ray-object intersections are more efficient if geometry rays are expressed in the object’s reference frame
  - More efficient to compute a ray-AABB intersection instead of a ray-OBB
    \( \Rightarrow \) need to express ray in the local coordinate system of the OBB
- When transforming objects, it is more efficient to inversely transform the rays instead:
  - Recalculating coordinates of transformed primitives is far more expensive
  - Transforming a ray instead of the object also facilitates the use of spatial partitioning for complex models
    \( \Rightarrow \) rigid animation of the latter requires no recalculation of the acceleration structures
  - Analytically defined primitives are difficult to re-parameterize in order to build a transformed version of the object
Ray Transformations (2)

- If $M$ the composite transformation that has been applied to an object in a scene hierarchy $\rightarrow$ apply the inverse transformation to the ray and perform the intersection test in local space of object:

  \[
  q = M \cdot q' = M \cdot \text{Object.RayIntersection}(M^{-1} \cdot p, M^{-1} \cdot \hat{r})
  \]

  where
  
  - $q$: Resulting intersection point in the original reference frame of the ray
  - $q'$: Intersection point expressed in the local object coordinate system
  - $p$: Ray origin
  - $\hat{r}$: Direction vector of the ray

- For static parts of a scene (or when a ray is 1st tested against a dynamic object in an animation frame):
  - Inverse matrix is calculated & stored in the object to be reused as long as the current transformation of the geometry is valid
Ray Transformations (3)

- For oriented bounding boxes (or other solids):
  - Directly obtain the 3 local coordinate system axes \((\hat{a}_1, \hat{a}_2, \hat{a}_3)\) & corresponding dimensions of the container
  - Need to precompute & store in the oriented bounding volume the transformation that produces the resized & rigidly transformed solid from its normalized axis-aligned version (or its inverse):

\[
\begin{bmatrix}
    a_{1x} & a_{2x} & a_{3x} & 0 \\
    a_{1y} & a_{2y} & a_{3y} & 0 \\
    a_{1z} & a_{2z} & a_{3z} & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}^{-1}
\]

\[
M_{OBV} = T_{OBV} T_{OBV}^{-1} S_{OBV}
\]

where

- \(T_{OBV}\): Translation according to bounding volume origin offset

- \(S_{OBV}\): Scales the bounding volume to fit its new dimensions

- \((15.25)\)
Constructive Solid Geometry

• Ray tracing renders very quickly objects modeled as set operations on solids

• *Constructive solid geometry* (CSG) is a modeling method that uses Boolean operations on a binary hierarchy of simple solid primitives to generate new complex solids

• Bounding surface of a CSG-generated solid can be calculated:
  - Either during rendering or
  - After the operations have been performed in object space & the solids have been converted to a boundary representation

• In latter case → operations on the geometry of original surface models required (non-trivial & sensitive to numerical errors)

• In ray tracing: Union (A OR B), intersection (A AND B) & difference (A AND NOT B) are treated as classification tests of the ray-object intersection points
Constructive Solid Geometry (2)

- Combined result of Boolean operation between 2 solids is calculated at run time without modifying original solids
- Priority of operations can be changed or optimized without affecting the final model → not true in general
- Combined & primitive solids taking part in a CSG model form a binary tree → CSG tree:
  - In a CSG tree, Boolean operations are expressed as CSG nodes
  - Each CSG node combines 2 sub-trees into one solid model
  - Left- & right-CSG children sub-trees may contain transformations or any other modifiers before encountering a solid model or another CSG node
- From modeler's point of view, the CSG tree is constructed bottom up, by continuously combining intersected, subtracted, or merged aggregations of solids with new ones
Constructive Solid Geometry (3)

- Example CSG tree to create a solid model (top left) from a set of simple solid primitives (bottom right):
Constructive Solid Geometry (4)

• Solids are combined \(\rightarrow\) corresponding intersection points form segments that are inside or outside the resulting solid:
  - If ray segment is outside \(\rightarrow\) its endpoints have to be discarded
  - If intersection point lies inside resulting volume \(\rightarrow\) no consequence to the ray-tracing paradigm & must also be discarded

• Need to keep a set of boundary surface points from each Boolean operation

• A CSG combination step is essentially an intersection point classification step:
  - Find all intersection points between ray & left CSG node child
  - Find all intersection points between ray & right CSG node child
  - Merge all intersection points in one sorted list
Mark each point according to its containment in left & right CSG children as IN, OUT, or SURFACE

Classify each point as IN, OUT, or SURFACE for the combined solid according to a set of logical rules

Keep all SURFACE points as the resulting intersection points of the CSG node

• CSG tree is recursively traversed from root node to the leaves
  • If a node is operation node → intersection points from its 2 children requested & the ray is propagated down & transformed according to the geometric transformations encountered
    • Then, above steps are performed & a new set of intersection points is determined
  • If a node is solid primitive → is intersected with the transformed ray & all the resulting points are gathered and returned upwards
Constructive Solid Geometry (8)

- Point classification for Boolean CSG operations:

<table>
<thead>
<tr>
<th>Union</th>
<th>Branch</th>
<th>Right</th>
<th>IN</th>
<th>OUT</th>
<th>SURFACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>IN</td>
<td>IN</td>
<td>IN</td>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
<td>OUT</td>
<td>IN</td>
<td>SURFACE</td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td>IN</td>
<td>OUT</td>
<td>SURFACE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SURFACE</td>
<td>IN</td>
<td>SURFACE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Difference</th>
<th>Branch</th>
<th>Right</th>
<th>IN</th>
<th>OUT</th>
<th>SURFACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>IN</td>
<td>OUT</td>
<td>IN</td>
<td>SURFACE</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>OUT</td>
<td>OUT</td>
<td>OUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td>OUT</td>
<td>SURFACE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SURFACE</td>
<td>OUT</td>
<td>SURFACE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intersection</th>
<th>Branch</th>
<th>Right</th>
<th>IN</th>
<th>OUT</th>
<th>SURFACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>IN</td>
<td>IN</td>
<td>OUT</td>
<td>SURFACE</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
<td>OUT</td>
<td>SURFACE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td>OUT</td>
<td>OUT</td>
<td>OUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SURFACE</td>
<td>SURFACE</td>
<td>OUT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Constructive Solid Geometry Example

- Intersection point classification for ray-traced CSG model rendering:

  ![Diagram showing intersection point classification for CSG model rendering](image)
Constructive Solid Geometry Example (2)

- Ray is passed to the root of CSG tree & is propagated recursively to the leaves
- First CSG node that can be computed is the difference node:
  - Intersection points of the ray $a$, $b$ & $c$, $d$ with the sphere & the box, respectively, are calculated from left & right children of the difference node & returned to the CSG node for classification
  - In subtraction of the 2 solids, all surface points of the 1st operand that are not clipped by the 2nd operand's volume are maintained
  - Points of the 2nd operand that reside outside the volume of the 1st operand are discarded
  - Surface points of 2nd operand form the boundary surface of the clipped region → kept
  - Intersection points marked as SURFACE are then regarded as intersection points of the combined solids & propagated upward
Constructive Solid Geometry Example (3)

- At next level, CSG node is a union set operation:
  - Need to keep points defining the largest combined ray segments (a & f):
    - Keep only SURFACE points of one solid that are outside the volume of the other solid

- The last operation is an intersection:
  - Seek to keep intersection points that bound ray segments intersecting both solids simultaneously
  - Classify as SURFACE points the boundary points of the one solid that are inside the volume of the other & vice versa (g & f)
  - All other points inside both volumes are valid ones but do not contribute to the outlier of the combined solid
Deficiencies of Ray Tracing

• **Major drawback of ray tracing**: The rendering speed
  - Although ray tracing algorithm is accelerated by various techniques, it is slower than hardware-accelerated direct rendering
  - But ray tracing is inherently easy to implement in parallel at an image-space level or in a ray distribution/spatial manner

• **Other deficiencies of ray tracing**:
  - Quality of the generated images
  - Realism of the generated images
Deficiencies of Ray Tracing (2)

- With ray tracing:
  - Reflections, shadows, & refracted parts of 3-D world appeared shaded with a local illumination model
  - Images obtained a fresh, startlingly clear look that boosted the credibility of the displayed subject significantly:
  - Or is the synthetic image too provocatively clear to be credible?
    - Surface of solid objects possesses structural irregularities \rightarrow scatter incident light to various directions:
    - For computation of specular highlights this principle is respected
    - It should also apply to the reflected and refracted light during ray tracing
    - Images reflected on or transmitted through objects as calculated by a ray tracer appear extremely sharp \rightarrow a single ray is spawned for each intersection point encountered
Deficiencies of Ray Tracing (3)

- Material interfaces assumed smooth in the neighborhood of the intersection point:
  - Incoming light from slightly different direction than the perfect reflection or refraction direction that would normally reach our eyes from a non-ideal reflector or transparent object cannot appear in a ray-traced image.
- This super-realistic rendering of the reflected and refracted images is characteristic to ray tracing:
  - Gives the synthetic images a very “polished” look.
- A single shadow ray is shot from an intersection point → not possible to generate soft shadows (naturally produced by emitters of non-negligible size).
- Shadow rays may hit or completely miss an occluding surface when cast toward the light source → only sharp shadows are produced.
Deficiencies of Ray Tracing (4)

- In ray tracing, indirect illumination that reaches a small surface area via diffuse inter-reflection is considered constant:
  - Replaced by the ambient term

- More advanced models that better approximate the rendering equation also compute this term & simulate other phenomena:
  - Indirect diffuse illumination
  - Indirect specular effects (caustics)
  - Defocusing of refracted and reflected rays
  - Scattering
  - Soft shadows