Ray Tracing
Constructing Rays

• For each pixel, make a ray and intersect it with objects in the scene
• The first intersection is used to determine a color for the pixel
• The ray is \( R(t) = A + (P - A)t \) where \( A \) is the aperture and \( P \) is the pixel location
• The ray is defined by \( t \in [0, \infty) \), although only \( t \in [1, t_{\text{far}}] \) will be inside the viewing frustum
• We only care about the intersection with the smallest \( t \) that is \( \geq 1 \)
Parallelization

- Ray tracing is a per pixel operation
- Ray tracing is inherently parallel, since the ray for each pixel is independent of the rays for other pixels

- Can utilize modern parallel CPU/GPU/Clusters to significantly accelerate a ray tracer
  - Threading (e.g., Pthread, OpenMP) distributes rays across cores
  - Message Passing Interface (MPI) distributes rays across processors on different machines
  - OptiX/CUDA distributes rays on the GPU

- Memory coherency helps when distributing rays to various threads/processors
  - Assign spatially neighboring rays (passing through neighboring pixels) to the same core/processor
  - These rays tend to intersect with the same objects in the scene, and thus access the same memory

- For the sake of comparison, scanline rendering is done per triangle, and is parallelized to handle one triangle at a time, usually on a GPU
Ray-Triangle Intersection

• Since so many triangles are typically present, many approaches have been implemented and tested in various software/hardware settings:

• Triangles are contained in planes, so it is often useful to look at Ray-Plane intersections first
• A Ray-Plane intersection yields a point, and subsequent testing can determine whether that intersection point is inside the triangle (or not)
  • Both the triangle and the intersection point can be projected into 2D, and the 2D triangle rasterization test (to the left of all 3 rays, discussed last week) can be used to determine “inside”
    • The projection can be done into the xy, xz, yz plane by merely dropping the z, y, x coordinate (respectively) from both the triangle vertices and the intersection point
    • The most robust coordinate to drop is the one with the largest component in the triangle’s normal (so that the projected triangle has maximal area)
  • Alternatively, there is a 3D version of the rasterization that works without projection to 2D

• One can skip the Ray-Plane intersection and consider the Ray-Triangle intersection directly
  • Thematically, this approach is similar to how ray tracing works for other non-triangle geometry (one advantage to the ray tracer is that it can consider non-triangle geometry!)
Ray-Plane Intersection

• Similar to the implicit equation for a line (discussed during rasterization), a plane is defined by a point on the plane $p_o$ and a normal direction $N$ (which need not be unit length)
• A point $p$ is on the plane if $(p - p_o) \cdot N = 0$
• A ray $R(t) = A + (P - A)t$ intersects the plane when $(R(t) - p_o) \cdot N = 0$ for some $t \geq 0$
• That is, $(A + (P - A)t - p_o) \cdot N = 0$ or $(A - p_o) \cdot N + (P - A) \cdot Nt = 0$
• So, $t = \frac{(p_o - A) \cdot N}{(P - A) \cdot N}$
  • Note: the length of $N$ cancels (so it need not be unit length)

• As always, if $t \notin [1, t_{far}]$ or there is an already computed intersection with a smaller $t$ value, then this intersection is ignored

• Note: the non-unit length triangle normal can be computed by taking the cross product of any two edges (as long as the triangle does not have zero area)
• Note: Any triangle vertex can be used as a point on the plane
3D Point Inside a 3D Triangle

- Given $t_{int} = \frac{(p_0 - A) \cdot N}{(P - A) \cdot N}$, evaluate $R(t_{int}) = R_o$ to find the intersection point.
- Then, given a directed edge of the triangle $e = p_1 - p_0$, compute a normal to that edge (in the plane of the triangle) via $n = (p_0 - p_2) - \left( (p_0 - p_2) \cdot \frac{e}{\|e\|} \right) \frac{e}{\|e\|}$.
- As usual, $R_o$ is interior to ray $e$ when $(R_o - p_0) \cdot n < 0$.
- As usual, if $R_o$ is interior to all three edges, it is interior to the triangle.
Recall: Triangle Basis Vectors

• Compute edge vectors $u = p_1 - p_0$ and $v = p_2 - p_0$
• Then, any point $p$ interior to the triangle can be written as $p = p_0 + \beta_1 u + \beta_2 v$ where $\beta_1, \beta_2 \in [0,1]$ and $\beta_1 + \beta_2 \leq 1$
• Substitutions and collecting terms gives $p = (1 - \beta_1 - \beta_2)p_0 + \beta_1 p_1 + \beta_2 p_2$ implying the equivalence: $\alpha_0 = 1 - \beta_1 - \beta_2$, $\alpha_1 = \beta_1$, $\alpha_2 = \beta_2$
Direct Ray-Triangle Intersection

- Points on the triangle are given by $p = p_0 + \beta_1 u + \beta_2 v$ where $\beta_1, \beta_2 \in [0,1]$ and $\beta_1 + \beta_2 \leq 1$
- Points on the ray have $R(t) = A + (P - A)t$
- So an intersection point has $A + (P - A)t = p_0 + \beta_1 u + \beta_2 v$
- Or $\begin{pmatrix} u & v & A - P \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ t \end{pmatrix} = A - p_0$ where $(u \ v \ A - P)$ is a 3x3 matrix and $A - p_0$ is a 3x1 vector (3 equations with 3 unknowns)
- This 3x3 system is degenerate when the columns of the 3x3 matrix are not full rank
- This happens when the triangle has zero area or the ray direction, $P - A$, is perpendicular to the plane’s normal
- Otherwise, when there is a unique solution, the point $R(t_{int})$ is inside the triangle (not only on the plane containing it) when the unique solution has: $\beta_1, \beta_2 \in [0,1]$ and $\beta_1 + \beta_2 \leq 1$
- As always, if $t \notin [1, t_{far}]$ or there is an already computed intersection with a smaller $t$ value, then this intersection can be ignored
Solving with Cramer’s Rule

• Solving the 3x3 system with Cramer’s Rule allows for code optimization:

• First compute the determinant of the 3x3 coefficient matrix \( \Delta = |(u \quad v \quad A - P)| \), which is nonzero when a solution exists.

• Then compute \( t = \frac{\Delta_t}{\Delta} \) where the numerator is the determinant: \( \Delta_t = |(u \quad v \quad A - p_0)| \).

• When \( t \notin [1, t_{far}] \) or there is an earlier intersection, one can quit (and ignore this intersection altogether).

• Otherwise compute \( \beta_1 = \frac{\Delta_{\beta_1}}{\Delta} \) where \( \Delta_{\beta_1} = |(A - p_0 \quad v \quad A - P)| \).

• When \( \beta_1 \notin [0,1] \) one can quit early.

• Otherwise compute \( \beta_2 = \frac{\Delta_{\beta_2}}{\Delta} \) where \( \Delta_{\beta_2} = |(u \quad A - p_0 \quad A - P)| \).

• When \( \beta_2 \in [0,1] \) and \( \beta_1 + \beta_2 \leq 1 \) the intersection is marked as true.
Ray-Object Intersections

- Ray tracing can be applied to any representation of geometry, as long as a ray-geometry intersection routine can be written.
- This is in contrast to scanline rendering where objects need to be turned into triangles.
- Thus, in addition to triangle meshes, we may use analytic geometric descriptions, implicitly defined surfaces, parametric surfaces, etc.

- Many object surfaces can be written in terms of functions.
- Typically, one states $f(p) = 0$ if and only if $p$ is on the surface (e.g., the equation for a plane).
- Sometimes there are additional constraints (such as on the barycentric weights for triangles).
- One broad/useful class of such objects are implicit surfaces (covered later in the class).
- The ray-object intersection routines for these types of surfaces all proceed down a similar path: substitute the ray equation in for the point, i.e., $f(R(t)) = 0$, and solve for $t$; then, check the solution against any other additional constraints.
Ray-Sphere Intersections

- A point $p$ in on a sphere with center $C$ and radius $r$ when $|p - C| = r$
- Or, after squaring both sides, when $(p - C) \cdot (p - C) = r^2$
- Substituting $R(t) = A + (P - A)t$ in for $p$ leads to a quadratic equation in $t$:
  \[(P - A) \cdot (P - A)t^2 + 2(P - A) \cdot (A - C)t + (A - C) \cdot (A - C) - r^2 = 0\]
- When the discriminant of this quadratic equation is negative, there are no solutions
- When the discriminant is identically zero, there is exactly one solution (where the ray tangentially grazes the sphere)
- When the discriminant is positive, there are two solutions (choose the one the ray hits first)
Transformed Objects

• As discussed previously, geometry is often stored/represented in a convenient object space
• The object space can make the geometry simpler to deal with
  • For example, spheres might have their center at the origin, objects are not sheared, coordinates may be non-dimensionalized for robustness, there may be auxiliary geometric acceleration structures, more convenient color and texture information, etc.

• Thus, we often prefer ray tracing in object space rather than world space
• This is accomplished by transforming the ray into object space, finding the ray-object intersection, and then transforming the relevant information back to world space

\[ M \cdot R_{\text{object}}(t_{\text{int}}) \]
Aside: Code Acceleration

- Ray-Object intersections can be expensive, so one often puts each complex object inside a simpler object, and first tests for intersections against the simpler object (potentially skipping the complex object)
- Simpler bounding volumes are typically spheres, axis-aligned bounding boxes (AABB), or oriented bounding boxes (OBB)
Aside: Code Acceleration

- For complex objects, one often builds a **hierarchical tree structure** in object space.
- The lower levels of the tree contain primitives used for intersections (and have simple geometry bounding them); these are combined hierarchically into a $\log n$ height tree.
- Starting at the top of a bounding volume hierarchy (BVH), one can prune out many nonessential (missed) ray-object collision checks.
Aside: Code Acceleration

- Instead of a bottom up bounding volume hierarchy approach, octrees and K-D trees take a top down approach to hierarchically partitioning objects (and space)
Normals

• Besides the point of intersection $R(t_{int})$, a ray tracer needs a local surface approximation
• This allows one to approximate how the surface interacts with light
  • thus, aiding in the determination of a final pixel color
• The local surface normal at the point $R(t_{int})$ is typically used to approximate a local tangent plane, which is subsequently used to determine the intensity of incoming photons

• Objects tilted towards the light are bombarded with more photons than those tilted away from the light
• Comparing the (unit) incoming light direction $\hat{L}$ with the local unit normal $\hat{N}$ leads to a tilting angle approximation of the from $-\hat{L} \cdot \hat{N} = \cos \theta$
• Incoming light with intensity $I$ is scaled down to $I \max(0, \cos \theta)$ based on tilting
  • the max with 0 accounts for triangles facing away from the light
• If $(k_R, k_G, k_B)$ is the RGB color of the triangle, where $k_R, k_G, k_B \in [0,1]$ are reflection coefficients, then the color applied to the pixel is $(k_R, k_G, k_B) \cdot I \max(0, \cos \theta)$
Ambient vs. Diffuse Shading

- Ambient shading colors a pixel when its ray intersects the object.
- Diffuse shading attenuates the object color based on tilting of the local unit normal away from the light source.
Computing Unit Normals

• The unit normal to a plane is used in the plane’s definition, and is thus readily accessible
  although it might need to be normalized to unit length

• The unit normal to a triangle can be computed by normalizing the cross product of two edges
  Be careful with the ordering in the cross product to make sure the normal points outwards from the object (as opposed to inwards)

• For more general objects, one needs to provide a function that returns an (outward) unit normal for the point of intersection

• For example, a sphere with intersection point $R(t_{int})$, has an (outward) unit normal of:

$$\hat{N} = \frac{R(t_{int})-C}{\|R(t_{int})-C\|_2}$$
Transformed Objects

- When ray tracing geometry in object space, the object space normal needs to be transformed back into world space along with the intersection point.

- Let \( u \) and \( v \) be edge vectors of a triangle in object space, and \( M u \) and \( M v \) be their corresponding world space versions.

- Then, the object space normal \( \hat{N} \) needs to be transformed to world space via \( M^{-T} \hat{N} \) so that
  \[
  Mu \cdot M^{-T} \hat{N} = (Mu)^T M^{-T} \hat{N} = u^T M^T M^{-T} \hat{N} = u^T \hat{N} = u \cdot \hat{N} = 0 \quad \text{and} \quad Mv \cdot M^{-T} \hat{N} = 0.
  \]

- Note that \( M^{-T} \hat{N} \) needs to be normalized to make it unit length.

- Careful, do not use \( M\hat{N} \) as the world space normal, for example:

  ![Diagram](image)

  \( \hat{N}' \) is not the normal.
Shadows

• The incoming light intensity, $I$, might need to be reduced if photons are blocked by other objects or parts of the same object
• Ray tracers use shadow rays to determine if photons from a light source are able to directly hit a point under examination
• A shadow ray is cast from the intersection point $R(t_{int})$ in the direction of the light $-\hat{L}$:
  $$S(t) = R(t_{int}) - \hat{L}t \text{ where } t \in (0, t_{light})$$
• If no intersections are found with $t < t_{light}$, then the light source is unobscured
• Otherwise, the point is shadowed, and photons from the light source are not used to color the pixel
• Note: every light source in the scene is checked with a separate shadow ray
• Note: one often includes low intensity ambient shading for points completely shadowed, so that they are not completely black
Spurious Self-Occlusion

- Note: \( t = 0 \) is not included in \( t \in (0, t_{light}) \) in order to avoid incorrectly computing an intersection with the same object near \( R(t_{int}) \)
- This can happen because of issues with numerical precision
- Note: shadow rays cannot simply ignore the object in question (when aiming to avoid spurious self-intersection) because that prevents objects from correctly self-shadowing
Spurious Self-Occlusion

- A simple solution is to use $t \in (\epsilon, t_{light})$ for some $\epsilon > 0$ large enough to guarantee that the ray does not incorrectly re-intersect the same object
- This works well for many cases
- However, grazing (shadow) rays near the object’s silhouette may still incorrectly re-intersect the object
Spurious Self-Occlusion

- Another option is to perturb the starting point of the shadow ray to be slightly away from the object (typically in the normal direction), e.g. from \( R(t_{int}) \) to \( R(t_{int}) + \epsilon \hat{N} \)
- In addition, the light direction needs to be modified to go from the light to \( R(t_{int}) + \epsilon \hat{N} \)
- The new shadow ray is \( S(t) = (R(t_{int}) + \epsilon \hat{N}) - \hat{L}_{mod} \cdot t \) where \( t \in (0, t_{light}) \)
- This works well, but one needs to take care that the new starting point \( R(t_{int}) + \epsilon \hat{N} \) does not fall inside (or too close to) nearby geometry
Aside: Code Acceleration

• When there are many different objects in the scene, checking rays against their top level simple bounding volumes can be expensive
• Thus, world space bounding volume hierarchies, octrees, K-D trees are also used
• Additionally useful, but flat instead of hierarchical, are uniform spatial partitions (uniform grids) and viewing frustum partitions
Aside: Code Acceleration

- There are many variants such as rectilinear grids with movable lines, hierarchies of uniform grids, and a structure proposed by [Losasso et al. 2006] that allows octrees to be allocated inside the cells of a uniform spatial partition