Foundations of Raytracing

Introduction

- Raytracing, foundamentally, **solves visibility problem**: two points are visible to each other if line segment that joins them does not intersect any obstacle.

    - First, we need to determine which objects are visible from camera – visibility problem! Based on defined camera, ray-tracing-based rendering is inherently providing **perspective or orthographic projection** in rendered image.

    - Secondly, we need to determine color and intensity of visible objects. This is solved using **shading** which relies on **light transport** (which relies on visibility problem) to obtain incoming light to surface.

Raytracing: generating rays

- Ray is fundamental element of raytracing. It is used for finding visible objects from camera as well as later in shading, that is, light transport.

- To generate ray, we use camera description in a 3D scene, e.g., pinhole camera.
  - Camera defines eye (aperture) position as well as film plane position, size and distance from aperture. Film plane is raster image: array of pixels

- Rays are generated by starting from eye and passing through the center of each pixel in the film plane.
  - It is possible, and in practice always desired, to generate multiple rays for each pixel. We will discuss this later, but it is important to note that large portion of the scene is actually represented by only one pixel. Since pixel can represent only one color, it is important to use multiple rays to obtain the color which is the most representative for that part of the scene covered by the pixel.

- Generated rays are called **camera/primary rays**. These rays will be used to compute the visible objects for current camera position. This method can be called **ray-casting**

<IMAGE: GENERATING RAYS>

Raytracing: generating rays algorithm

- TODO

Camera rays: testing for intersections

- Camera rays are **traced** into the 3D scene
  - They are tested for intersection with each object in 3D scene: we loop over all 3D objects and test them for intersection with camera ray.
  - Note that in this step, we are interested in a shape of 3D object. Later, once intersection is determined material will be used during the shading step. Therefore, this shows usefulness of decoupling material and shape of 3D object (if one decides on such rendering architecture decision).
- Two possibilities are:
  - Ray intersects an object
  - Ray doesn't intersects anything → intersect background

Testing object intersections

- In lesson on objects in 3D scene, we have seen that **various shape representations** exist for 3D objects.
  - Parametric representations (e.g., simple objects; spheres), mesh, curves, curved surfaces, subivision surfaces, voxels, SDFs, etc.
- Objects which shapes are represented parametrically can be tested analytically: spheres, planes, disks, etc.
  - Very often, you will see raytracing demo scenes containing (only) planes and spheres!
- For other objects, ray-shape intersection must be defined for each representation separately.
  - For example, different intersection test is performed for triangle meshes, quad meshes, curved surfaces, etc.
- Alternative solution (which is taken by almost all professional rendering software) is to convert each shape representation to same internal representation which is used for rendering. This internal representation is in almost all cases triangulated mesh and for this representation, the conversion method is called **tessellation**
  - As discussed in lectures on 3D scene modeling, it is worth highlighting again the two sides of the computer graphics/image generation: authoring of 3D scene and rendering it. Certain representations are much more easier and efficient to handle on authoring level and some are much more efficient to handle for rendering purposes. Therefore, efficient mapping of those representations quite important and researched.

Tessellation and triangulated mesh.

- Conversion of almost any type of surface to a triangulated mesh (tessellation) is well researched and feasible.

- Tessellation process can be done after modeling of shape is done and when exporting takes place or it can be done during rendering (render time).

- Why triangles?

  – Triangulated mesh is also basic rendering primitive for rasterization-based renderers. Furthermore, graphics hardware is adapted and optimized to efficiently process triangles.

  – Triangles are necessary co-planar which makes various computations, such as ray-triangle, much easier

  – Lot of research was devoted to efficient computation of ray-triangle

  – For triangles we can easily compute barycentric coordinates which are essential to shading.

Intersection tests

- For now we will discuss some popular ray-intersection tests:
  - Plane: TODO
  - Sphere: TODO
  - Triangle: TODO

- This serves as good basis for investigating further on ray-shape intersection which is highly researched and developed topic*

* https://www.realtimerendering.com/intersections.html

Testing intersections (triangulated meshes) algorithm

- TODO

-

Notes on testing intersections

- Time to render a scene is (directly) proportional to the number of triangles in the scene.

- Raytracing-based rendering, in shading step, has to test more triangles in the scene than compared to rasterizer where many of triangles can be discarded if not visible from camera (e.g., back faces of object – back face culling).

- Optimization: not all triangles have to be tested for intersections. If certain parts of the scene are not relevant for current ray, they can be skipped
  - **Spatial acceleration data structures**

Testing intersections: practical tip

- Same ray-shape intersection algorithm can be used for any kind of tracing ray into the scene since only information is current ray and objects in the scene. Therefore we can use it both for:

  - Camera rays used to determine what is visible from camera

  - Rays used in light transport for gathering light

- Therefore, this part of the code can be encapsulated

Testing intersections: multiple hits

- Imagine scene where we have several layers of the objects so that behind one object, there is another one. <IMAGE>

- Tracing ray in this case intersects multiple objects

- When determining what is visible from camera, we take the first intersection - one closest to the camera

- Note that based on material, objects behind will influence (transparent objects) or be simply discarded (opaque objects).

  - This computation is performed in shading step. This is important element to take in account - we need to trace additional rays for computing incoming light to intersected point not only from surfaces facing the intersected point – but also surfaces which are behind.

Testing intersections: multiple hits algorithm

- TODO

Object intersection: shading

- Once intersection with object is found, we need to calculate color and intensity (appearance) of that point. Method for calculating this is called **shading**.

- **Shading takes in account material and shape of the objects and couples it with light to compute color and intensity of intersected point.**

- Important shape information in intersected point used in shading are: position, normal, texture coordinates, surface derivatives.

Shading

- Color of an object at any point is the result of the way object returns or reflects light falling on that point to camera.

- Color and intensity of object varies across surface. In intersection point - **shading point****, color and intensity* depends on:
  - Amount of incoming light – how much surface is oriented to emitter (direct light) or other surface which reflects emitter or another surface (indirect light) and how much light actually falls onto the shading point.
  - Incoming light direction – from where is light falling on shading point
  - Surface material – as discussed in object modeling lesson, material can be decomposed in scattering (reflection) characteristic described by BSDF (BRDF) and variation of its properties (color, roughness, etc.) across surface – Texture.
  - Position of camera – as discussed in object modeling lesson, some materials (that is scattering functions) depend on viewer position and are not reflecting light equally in all directions.

* Several times it has been mentioned "color and intensity". In real world this is the same part of phenomena we see. In computer graphics this is decomposed for easier simulation and handling.
<TODO>
** At this stage, we can start call intersection point the shading point. This is popular and important term to keep in mind.

Shading and light transport

- As we have seen, shading depends on object characteristics (material and shape) but also very much on incoming light.

- The process of "gathering" incoming light on shading point is called light transport.

<IMAGE: gathering light in shading point>

- In raytracing, light is gathered by tracing rays from intersection point to the 3D scene.
  - Reminder: the ray generated by camera to find what is visible is called camera ray.
  - All rays that are casted after camera ray – from intersection of camera ray and object to the scene are called **secondary rays**
  - Secondary rays have specific names depending on their task:
    - Ray casted toward light is called **shadow ray**
    - Ray casted from mirror is called **reflection ray**

- **Secondary rays used in light transport are solving visibility problem!** For example secondary ray casted towards light connects two points: intersection point and light position. And it is important to evaluate if these two points are visible to each other – if there is another object on this segment.

Shading and light transport: algorithm

- TODO

Notes on ray-tracing based rendering

- Ray-tracing challenges: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/ray-tracing-rendering-technique-overview