Foundations of Raytracing

Motivation scene

- TODO: image rendered with ray-tracing based rendering engine

Raytracing-based rendering overview

Introduction

- Ray-tracing is considered one of the most elegant techniques in computer graphics. Many phenomena such as shadows, reflections and refracted light are intuitive and straightforward to implement.

Robert L. Cook Thomas Porter Loren Carpenter. "Distributed Ray-Tracing" (1984).

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.

Recap: three steps of ray-tracing

- Generate ray for each pixel of film plane → camera ray

- Ray-geometry intersection: testing intersection of generated ray and 3D scene objects to obtain what is visible from camera

- Shading: calculating the color and intensity of intersected points
  - Requires light-matter calculation
  - Requires light transport for gathering incoming light

Ray-tracing-based rendering: trace & shade

- Raytracing can be described with two functions:
  - **trace()**
  - **shade()**
- **trace()** is geometrical part of algorithm and it is responsible for finding closest intersection between ray and the primitives in 3D scene.
- Once intersection is found, **trace()** returns color of the ray by calling **shade()**

Trace() for camera rays

- As the goal of rendering is to find colors of pixel in the image, trace() function is used for rays generated by camera: eye or camera rays

- Now, the goal of trace() function is to find closest intersection of camera ray with 3D scene objects.

- Naive trace() function loops through all n objects in the scene
  - This is slow for complex scene → o(n) performance
  - Spatial acceleration structure (BVH or k-d tree) is used to achieve $O(log(n))$ performance

shade() for closest intersection

- Color in closest intersection is calculated using shade() it this is done using:
  - Material information of intersected point
  - Incoming light computed using trace() function
- Shade() can be arbitrarily complex:
  - It can just return the color of object
  - It can use material information with incoming light information
    - It can ask trace() for closest light sources
    - It can ask trace() to gather all incoming light
- Each shade() can call trace() and each trace() can call shade()
  - Ray depth is term with indicates number of rays that have been shot recursively along a ray path.

trace() for light transport

- Simplest case is that trace() calculate visibility between shaded point and light source

  – Shadow ray: is shaded point in the shadow or visible to light source?

- Trace() can also use normal at intersection and compute reflection or refraction ray

  – Perflectly sharp reflections and/or refractions with sharp shadows is called Whitted ray-tracing.

Digression: ray-casting

- Ray-casting is often used in trace() function for determining visibility between two points

- <image: raycasting>

- It is also very useful operation in different applications
  - Ambient occlusion
  - modeling

Practical foundations

- Overall structure of Whitted ray-tracer which is bases of many other rendering variatns (e.g., path-tracing) can be described with three functions.

- TODO: images - https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf

```
function RAYTRACEIMAGE
    for p do in pixels
        color of p = trace(camera ray through p);
    end for
end function

function TRACE(ray)
    pt = find closest intersection;
    return shade(pt);
end function

function SHADE(point)
    color = 0;
    for L do in light sources
        trace(shadow ray to L);
        color += evaluate material;
    end for
    color += trace(reflection ray);
    color += trace(refraction ray);
    return color;
end function
```

- Shade() function is implemented by user
  - This is were knowledge about materials: scattering functions and textures comes in

- Traversial and intersection testing takes place in trace() function often implemented on CPU but moving to GPU using compute shaders in Vulkan or DXR

Whitted ray-tracing and beyond

- Whitted ray-tracing is not full solution to global illumination
  - Light reflected from any direction than mirror reflection or refraction is ignored
  - Direct lights are only represented with point lights
- Fully evaluation of global illumination is is proposed by Kajiya as path-tracing method
  - Correct solution which generates global illumination
  - After camera ray intersection is found and during shading evaluation,may rays are generated in different directions
    - For diffuse surface, rays over hemisphere at intersection are shot
    - For glossy surface, rays in lobe at intersection are shot
    - Etc.
  - Problem with this approach is explosion of rays, thus Monte Carlo sampling methods are employed for generating paths through environment.
    - Several of such paths are averaged for each pixel
  - General problem with path-tracing is noise and amount of rays that have to be shot

Ray-tracing steps overview

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.
  - Ray is defined as point (origin of ray) and direction. <IMAGE: RAY>
- 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

Reading: Testing object intersections

- Most important task of raytracing is to compute visibility which is done using ray-object intersection tests. Mathematical solutions are needed to compute intersections of ray and objects in 3D scene

- 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 (triangulated, quad, etc.), curves, curved surfaces (NURBS), 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.

Reading: 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).

- Therefore, efficient ray-object intersections are needed!

- Additional 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 –** object shape information is "sorted" in 3D scene and scene is spatially subdivided. This way, rays first test larger volumes of the scene and if that volume is not relevant, all objects inside do not have to be tested for intersections! Such acceleration structures require pre-computation overhead: if we are rendering static scenes, then it is computed only once before rendering. If we are rendering animated scenes, these datastructures must be pre-computed each time objects move (each frame) before rendering.

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 of camera ray 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 incoming 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.

Light transport

- Much of the shading process in shading point relies on simulating light bouncing off of the objects or light traveling through objects in 3D scene → **light transport**
  - Essentially, seeing an object means that light is reflected by this object in the direction of camera/eye.
  - Some (a lot actually!) objects in a 3D scene may reflect light in direction where camera is not placed, but due to indirect reflection, the light is reflected by another object in the scene*.
  - Opaque surfaces (e.g., shiny) cause light reflection
  - Transparent surfaces cause light refraction while entering, traveling and exiting such objects

- Light transport must simulate light scenarios in acceptable amount of time and yet providing best visual accuracy possible.
  - Light transport algorithm is strategy to solve light transport. Many strategies for light transport have been developed over time (and still are!)

- Simulation of light bouncing in the scene requires:
  - Models which determine how ray is reflected once it falls on object – for this purposes we use material and its component: BRDF/BSDF
  - Efficient intersection tests for any object in 3D scene computing visibility. Besides computing intersections with first visible objects (objects visible from camera), we also need to compute visibility for arbitrary points in the scene after**. Note that next to ray-tracing we might employ other methods for visibility calculation and thus light transport.

* Note here once more the importance of material information. We said that material information is used during shading process which is true. But material information is also used during light transport to determine how will light bounce. Note that specifically scattering function in material description will give this information. By now we have mostly looked at scattering function as a function that gives us information how much light from camera ray intersection (shading point) will reflect into eye from incoming light direction. But in light transport, the same function is used to determine where the light will scatter given the input direction – thus the name "bidirectional" in BRDF/BSDF.

** This is one of the largest differences between ray-tracing-based and rasterization-based rendering. Rasterization fast to calculate what is visible from camera, but is is not suitable at all to give information which surfaces are visible to surfaces visible from camera and not to mention arbitrary suface visibility which is important for quality light transport. There are some approaches in radiosity light transport utilizing hemisphere and hemicube but they are extremly slow. INVESTIGATE!

Note: light transport and visibility

- Important ingredient for photo-realistic images is high-quality light transport algorithm.

- Light transport, as discussed relies on visibility calculation.

- By now, we have introduced ray-tracing as one possible solution for visibility calculation. Another one is rasterization.

- When considering rendering algorithm/software, it is important to understand which kind of light transport it employs and which visibility calculation method it uses.
  - Visibility method dictates light transport and thus realism of rendered images.
  - Amount of global illumination effects that can be simulation depends on light transport algorithm.

- For example, light transport algorithms employing ray-tracing as visibility method are path-tracing and bidirectional path-tracing.

- Practical tip: In DCC Tools, users are often provided with faster and lower quality rendering methods and slower and higher quality rendering methods. First one is used for preview and modeling. Second one are used for final renders.

<IMAGE: Workbench vs EEVEE vs Cycles in Blender>

Light transport based on ray-tracing

- Reminder: difference between visibility method and light transport algorithm: light transport algorithm employs visibility for calculation.

- Classical example of light-transport algorithm utilizing ray-tracing for computing visibility is **Whitted-style-ray-tracing.** This method employs **backward tracing**.

- Whitted light transport algorithm simulates*:
  - Diffuse and specular (glossy?) surfaces
  - Reflections between objects. <Example: mirror-like and diffuse (diffuse one will be reflected in mirror-like surface) and two mirror-like objects (both objects will be reflected into each other)>
  - Refraction when light enters and exits transparent objects. <Example: transparent surface and bending rays due to refraction while light passing through them>

<IMAGE: reflections and refractions vs mirror like surfaces without reflections>

- When we discussed materials, we introduced concepts of reflection and refraction as well as their physical foundations. These are used in Whitted light transport for calculating light bouncing in 3D scene which contains mirror-like and transparent surfaces:
  - Reflection direction depends on surface orientation (normal - property of object shape) and incoming light direction.
  - Refraction direction depends on surface normal (property of object shape), incoming light direction and index of refraction (property of material).
    - Tip: Higher refractive index of transparent object → stronger specular reflections.
    - Tip: Higher angle between normal and angle of incidence → more light is reflected (Fresnel effect)

* Before Whitted introduced Whitted-ray-tracing light transport, there were already reflection models capable of simulating diffuse and specular surfaces – Phong scattering model. But note that specular surfaces (as they are mirror like) should reflect other objects in the scene – not only appear mirror-like. Also, nect to specular reflection we can also observe specular refraction that happens for example on glass. For exactly this effects, we need light-transport algorithm coupled with scattering model to produce them.

Whitted rendering method overview

Refraction or reflection rays falling on surface may have following cases*:

- If surface is opaque and diffuse, then use diffuse scattering model (e.g., Phong) and other material properties (e.g., color) to calculate reflected color and intensity. As discussed, we need to know information about incoming light (and if it obstructed from those - shadow) and thus simple raycasting from intersection point to light is performed.

- If surface is opaque and mirror-like, use specular reflection model to determine new ray from intersection

- If surface is transparent, use specular reflection and refraction model to determine reflected and refracted ray from intersection point.

<IMAGE: different reflection/refraction possibilities: mirror-mirror-diffuse and how second first mirror take the color of diffuse object.>

<IMAGE: different reflection/refraction possibilities: transparent-transparent-diffuse and how second first transparent take the color of diffuse object. Higher refractive index: stronger distortion (e.g., glass vs diamond)>

* All of those effects are relying on ray-object intersections and visibility calculation performed by ray-tracing!

Whitted light-transport properties: recursive nature

- Shading process, which starts when camera ray intersection with object in the 3D scene is found, requires gathering light at shading point using light transport. Direct illumination only is about computing visibility between intersection point and light source. Indirect (global) illumination is much more complex:

  - Each visible intersection of a ray with a surface produces more rays in reflection/refraction direction.

  - If scene contains lot of reflective (mirror-like) objects, then the shading point is reflective and another ray is generated and traced into scene. This ray can intersect another object, if this object is reflective, then another ray is generated and traced into scene → as we can see, this process can continue to infinity and it is of **recursive nature**.

  - More complex case is if 3D scene also contains transparent objects. In this case, each intersection causes two new rays that are traced in the scene. Note that in this case we have two paths that have to be processed.

  - To stop generation of new rays for path, it is required to set maximum number of bounces (recursions).

  - Based on the discussion above, we can conclude that render time increases exponentially with the level of depth

  <IMAGE: tracing rays and recursion>

Whitted light-transport properties: tree of rays

- All secondary rays (reflection or refraction) spawned by primary or other secondary rays can be represented tree-like structure

- Each intersection marks new depth/level of the tree and thus one level of recursion

<IMAGE: tree of rays>

Shading and light transport: algorithm

- TODO

Notes on ray-tracing based rendering

- Pros:
    - Ray-tracing based rendering is clear and straightforward method to implement*
    - It represent unified framework for computing what is visible from camera (primary ray intersections), with inherent perspective/orthographics projection and computing light transport – visibility between surfaces in 3D scene (secondary ray intersections) required for the shading process.
- Cons:
    - It requires efficient methods for testing ray-shape intersections: sometimes hard to develop (good understanding of mathematics, particularly geometry and linear algebra for geometrical or analytical solutions)
    - Ray-shape intersections test are core element during rendering and they are often very expensive
    - For advanced light transport needed in shading stage, ray-tracing-based renderer must store all scene objects must be readily available for intersection testing and material evaluation**. An example where is this needed is to compute indirect light or shadows casted by different objects.
    - Additional acceleration datastructures are often desired to accelerate ray-scene intersections. These datastructures require efficient development, cause additional precomputations (with tremendous acceleration times, though) and require a lot of memory – even more for animated scenes.

* Implementing production ray-tracing based rendering requires a lot of additional features than ones that we have discussed: support for render-time displacement mapping (shape vertices offsetting given height map), motion blur, programmable shading stage (a program which is evaluated for primary ray intersections).
** As we will discuss, this is not needed for rasterizer since only visible geometry is taken in account – which on the other hand disables advanced light transport (again, visibility calculation between objects in the scene) and thus shading.

Practical raytracing-based rendering

Code

Code

- TODO

- https://raytracing.github.io/books/RayTracingInOneWeekend.html#overview

- https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/light-transport-ray-tracing-whitted

-

Practical-raytracing based rendering

Production

Back to motivation scene

- Show cycles rendering in Blender

- TODO

Note: real-time ray-tracing

- Due to nature of ray-tracing-based visibility and shading calculations, they are mostly written for CPUs

- Porting ray-tracers to GPU was done in 90s

- Ray-tracing ideas are often used on GPU and rasterizer renderers, but not to a full extent we discussed
  - Hybrid approaches exists: rasterization might be used to determined objects/surfaces visible from camera. Raytracing may be employed for reflection and refraction calculations.

- Newer graphics hardware (currently led by Nvidia with RTX family of graphics cars) and new graphics APIs (DirectX 12, Vulkan) are supporting hardware-accelerated ray-tracing – a new hardware and then software element called raytracing kernel can be now used in combination with rasterizer-based renderer to compute complex scene effects such are reflections, soft shadows, indirect illumination, etc.