Welcome!
Today's Agenda:

- Advanced Graphics
- Recap: Ray Tracing
- Lighthouse 2
- Assignment 1
INFOMAGR

Website

http://www.cs.uu.nl/docs/vakken/magr

- Site information overrules official schedule
- Downloads, news, slides, deadlines, links

Please check regularly.

Discord

- Please join!

Advanced Graphics

UNIVERSITEIT UTRECHT - INFORMATION AND COMPUTING SCIENCE

academic year 2019/20 - 2nd block

Navigation

News (Archive)
Schedule
Downloads
Format & Organization
Lecture Slides
Practicals
Abstract

In this course, we explore physically based rendering, with a focus on interactivity.

At the end of this course, you will have a solid theoretical understanding of efficient physically based light transport.

You will also have a good understanding of acceleration structures for fast ray/scene intersection for static and dynamic scenes.

You will have hands-on experience with algorithms for efficient realistic rendering of static and dynamic scenes using ray tracing on CPU and GPU.

Concrete / informal:

1. You’ll know how a photo-realistic image is produced
2. You know how to do this quickly / efficient
3. You have built such a renderer
4. You have built an interactive ray tracer
5. You know how to do this on the GPU
6. You got a great score
7. You had fun
We will cover the following topics:

- Ray tracing fundamentals;
- Whitted-style ray tracing;
- Acceleration structure construction;
- Acceleration structure traversal;
- Data structures and algorithms for animation;
- Stochastic approaches to AA, DOF, soft shadows, …;
- Path tracing;
- Variance reduction in path tracing algorithms;
- Filtering techniques;
- RTX, OptiX, Embree, RadeonRays;
- Various forms of parallelism in ray tracing.
Lectures

~17 lectures:
Tuesday 11:00 – 12:45, Thursday 13:15 – 15:00

~7 working colleges:
Thursday 15:15 – 17:00

Attendance is not mandatory, but of course highly recommended.

*We move fast; missing a key lecture may be a serious problem.*
Literature

Papers and online resources will be supplied during the course.

Slides will be made available after each lecture.

Recommended literature:


The 3rd edition is available for free: www.pbr-book.org
Dependencies

It is assumed that you have basic knowledge of rendering (INFOGR) and associated mathematics.

You also should be a decent programmer; this is explicitly not a purely theoretical course. You are expected to verify the theory and experience the good and the bad.

A brief introduction to GPGPU and SIMD will be provided for those that did not take INFOMOV (pdfs will be available from the website).
Resources

You will develop a ray tracing testbed for assignment 1. As a starting point, several ‘templates’ are available:

**tmpl_2019_v2.zip**
A basic C++ framework for graphics programming, which opens a window, provides a 32-bit RGB framebuffer, and some basic helper classes. A C# version is also available.

**tmpl_OCL_2019.zip**
CUDATemplate2019.zip
Tmpl_OCL_Csharp_2018.zip
Frameworks for GPGPU via CUDA and OpenCL.

**Lighthouse 2**
Open source experimentation framework for real-time ray tracing.
Assignments

1. (weight: 1):
   Ray tracing framework or Lighthouse 2 core
   
   For this assignment, you prepare a testbed for subsequent assignments.

2. (weight: 1):
   Acceleration structures
   
   In this assignment, you expand your testbed with efficient acceleration structure construction and traversal. This enables you to run Whitted-style ray tracing in real-time.

3. (weight: 2):
   Final assignment

   In this assignment, you either implement an interactive path tracer, or a rendering algorithm you chose, using CPU and/or GPU rendering.
Exam

One final exam at the end of the block.

Materials to study:

- Slides
- Notes taken during the lectures
- Provided literature
- Assignments
Grading & Retake

Final grade for assignments  \[ P = \frac{(P_1 + P_2 + 2 \times P_3)}{4} \]

Final grade for INFOMAGR  \[ G = \frac{(2P + E)}{3} \]

Passing criteria:

- \[ P \geq 4.50 \]
- \[ E \geq 4.50 \]
- \[ G \geq 5.50 \]

Repairing your grade using the retake exam or retake assignment:

- only if \( 5.50 > G \geq 4.00 \)
- you redo P1 or P2 or P3 or E
- this replaces the original P1, P2, P3 or E grade.
Today's Agenda:

- Advanced Graphics
- Recap: Ray Tracing
- Lighthouse 2
- Assignment 1
Ray

A ray is an infinite line with a start point:

\[ P(t) = O + t\vec{D}, \text{ where } t \geq 0. \]

The ray direction \( \vec{D} \) is usually normalized: this way, \( t \) becomes a distance along the ray.
Scene

The scene consists of a number of primitives:

- Spheres
- Planes
- Triangles

...or anything for which we can calculate the intersection with a ray.

We also need:

- A camera (position, direction, FOV, focal distance, aperture size)
- Light sources
Recap

Ray Tracing

*World space*

- Geometry
- Eye
- Screen plane
- Screen pixels
- Primary rays
- Intersections
- Point light
- Shadow rays

Light transport

- Extension rays

Light transport
Recap

Ray setup

A ray is initially shot through a pixel on the screen plane. The screen plane is defined in world space, e.g.:

Camera position: \( E = (0,0,0) \)

View direction: \( \vec{V} = (0,0,1) \)

Screen center: \( C = E + d\vec{V} \)

Screen corners: \( P_0 = C + (-1,1,0), \ P_1 = C + (1,1,0), \ P_2 = C + (-1,-1,0) \)

From here:

- Change FOV by altering \( d \);
- Transform camera by multiplying \( E, P_0, P_1, P_2 \) with the camera matrix.
Ray setup

Point on the screen:

\[ P(u, v) = P_0 + u(P_1 - P_0) + v(P_2 - P_0) \]

\[ u, v \in [0, 1] \]

Ray direction (normalized):

\[ \vec{D} = \frac{P(u, v) - E}{\| P(u, v) - E \|} \]

Ray origin:

\[ O = E \]
Recap

Ray setup

Alternatives:

- Taking into account HMD lens distortion
Recap

Ray setup

Alternatives:

- Taking into account HMD lens distortion
- Fisheye lens
Recap

Ray setup

Alternatives:

- Taking into account HMD lens distortion
- Fisheye lens
- Complex lens system
Ray Intersection

Given a ray $P(t) = O + t\vec{D}$, we determine the closest intersection distance $t$ by intersecting the ray with each of the primitives in the scene.

Ray / plane intersection:

Plane: $P \cdot \vec{N} + d = 0$
Ray: $P(t) = O + t\vec{D}$

Substituting for $P(t)$, we get

$$ (O + t\vec{D}) \cdot \vec{N} + d = 0 $$

$$ t = -(O \cdot \vec{N} + d) / (\vec{D} \cdot \vec{N}) $$

$$ P = O + t\vec{D} $$

Math reminder, dot product:

$A \cdot B = A_x B_x + A_y B_y + A_z B_z$

$A \cdot B = \cos \theta$

$A \cdot B$ is: the length of the projection of $A$ on $B$.

$A \cdot B$ is a scalar.
Ray Intersection

Ray / sphere intersection:

Sphere: \((P - C) \cdot (P - C) - r^2 = 0\)

Substituting for \(P(t)\), we get

\[(O + t\vec{D} - C) \cdot (O + t\vec{D} - C) - r^2 = 0\]
\[\vec{D} \cdot \vec{D} t^2 + 2\vec{D} \cdot (O - C) t + (O - C)^2 - r^2 = 0\]

\[at^2 + bt + c = 0 \quad \Rightarrow \quad t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}\]

Recap

Negative: no intersections
Ray Intersection

Efficient ray / sphere intersection:

```cpp
void Sphere::IntersectSphere(Ray ray) {
    vec3 C = this.pos - ray.O;
    float t = dot(C, ray.D);
    vec3 Q = C - t * ray.D;
    float p2 = dot(Q, Q);
    if (p2 > sphere.r2) return; // r2 = r * r
    t -= sqrt(sphere.r2 - p2);
    if ((t < ray.t) && (t > 0)) ray.t = t;
}
```

Note:
This only works for rays that start outside the sphere.
Observations

Ray tracing is a *point sampling process*:

- we may miss small details;
- aliasing will occur.

Ray tracing is a *visibility algorithm*:

- For each pixel, we find the nearest object (which occludes objects farther away).
Observations

Note: rasterization (Painter’s or z-buffer) is also a visibility algorithm.

Rasterization:
- loop over objects / primitives;
- per primitive: loop over pixels.

Ray tracing:
- loop over pixels;
- per pixel: loop over objects / primitives.
Today's Agenda:

- Advanced Graphics
- Recap: Ray Tracing
- Lighthouse 2
- Assignment 1
Lighthouse 2

Application
- Opens an OpenGL window
- Prepares the scene
- Handles user input
- Calls ‘Render’

(Several samples provided)

RenderSystem
- Loads 3D scene data
- Owns & maintains the scene
- Handles animation
- Synchronizes scene data with a ‘Core’
- Invokes a ‘Core’

API overview in:
API ➔ render_api.h

You should not have to change anything in the RenderSystem.

Core
- Receives raw triangle data from RenderSystem
- Performs actual rendering

Several are available:
- Optix7, OptixPrime & Vulkan ray tracing cores;
- A software rasterizer.
The Lighthouse 2 system has been designed to facilitate the development of rendering software based on ray tracing.

The RenderSystem takes care of the boring things, such as model loading and scene management. You get to concentrate on the actual ray tracing.

The current system needs a high-end NVIDIA device for pretty images. Without a recent GPU, only a software rasterizer fallback is available. This is where you come in.
RenderCore

A core implements the low-level rendering functionality.

To build your own: start by copying an existing one.

(project folder ➔ docs ➔ cloning a core.docx)

A core receives raw geometry from the RenderSystem:

```cpp
void RenderCore::SetGeometry(
    const int meshIdx,
    const float4* vertexData,
    const int vertexCount,
    const int triangleCount,
    const CoreTri* triangles,
    const uint* alphaFlags)
```

Note: The core is expected to copy the data it receives. There is no guarantee at all that the RenderSystem leaves the provided data intact after SetGeometry returns.

float4: x, y, z, padding

also contain vertices. raw verts are for intersection; tris are for shading.

RenderSystem makes CoreTris from HostTris, CoreMaterials form HostMaterials and so on.
RenderCore

A core implements the low-level rendering functionality.

A core receives instances from the RenderSystem:

```cpp
void RenderCore::SetInstance(
    const int instanceIdx,
    const int meshIdx,
    const mat4& matrix
)
```

**Note:**
- A mesh that is not instanced is not visible.
- SetInstance always receives a flattened list of instances; there is no hierarchy.
RenderCore

A core implements the low-level rendering functionality.

A core receives a Render command from the RenderSystem:

```cpp
void RenderCore::Render(
    const ViewPyramid& view,
    const Convergence converge,
    const float brightness,
    const float contrast
)
```

**Note:**
- Feel free to ignore brightness and contrast for now.
- Converge will be true when the camera is stationary. This will be useful later for path tracing.
void Init();
CoreStats GetCoreStats();
void SetProbePos( int2 pos );
void SetTarget( GLTexture* target, uint spp );
void Setting( char* name, float value );
void Render( ViewPyramid& view, Convergence converge, float brightness, float contrast );
void Shutdown();
void SetTextures( CoreTexDesc* tex, int textureCount );
void SetMaterials( CoreMaterial* mat, CoreMaterialEx* matEx, int materialCount );
void SetLights( CoreLightTri* arealights, int arealightCount,
    CorePointLight* pointLights, int pointLightCount,
    CoreSpotLight* spotLights, int spotLightCount,
    CoreDirectionalLight* directionalLights, int directionalLightCount );
void SetSkyData( float3* pixels, uint width, uint height );
void SetGeometry( int meshIdx, float4* vertexData, int vertexCount,
    int triangleCount, CoreTri* triangles, uint* alphaFlags = 0 );
void SetInstance( int instanceIdx, int modelIdx, mat4& transform );
void UpdateToplevel();
Today's Agenda:

- Advanced Graphics
- Recap: Ray Tracing
- Lighthouse 2
- Assignment 1
Ray Tracing Testbed, Option #1: Create a Core

Implement a simple ray tracing core based on the software rasterizer. Ingredients:

SetGeometry: Copy the data, either vertices and triangles or just the triangles.
SetInstance: Make sure your geometry has an identity matrix, easier for now.

Ignore textures and materials for now; you don’t need this for a simple render.

Render:

The render method loops over the pixels:
- Initialize a primary ray
- Intersect the ray with each triangle to find the closest intersection
- Visualize the closest intersection (e.g. a greyscale ‘distance’)

From here you can flesh out the other functionality.
Assignment 1

Ray Tracing Testbed, Option #2: Rolling Your Own

Implement an experimentation framework for ray tracing. Ingredients:

Scene
- Primitives: spheres, planes, triangles
- I/O (e.g., obj loader)
- Intersection
- Materials: diffuse color, diffuse / specular / dielectric, absorption

Camera
- Position, target, FOV
- Ray generation

Ray
- Renderer
  - Whitted-style

User interface
- Input handling
- Presentation
Ray Tracing Testbed, Option #2: Rolling Your Own

Regarding the file loading requirement:

- You may want to start with handcrafted scenes
- Don’t build your own OBJ loader, that’s a waste of time
- Use assimp or tinyobjloader (C++) or find a lib if you’re using C#
  
  https://github.com/ChrisJansson/ObjLoader

- Start with small files; minimize your development cycle.
Ray Tracing Testbed, Option #1 and #2

Intersecting triangles:

An easy to implement and quite efficient algorithm is:


...which is explained in elaborate detail by scratchapixel.com:

Today’s Agenda:

- Advanced Graphics
- Recap: Ray Tracing
- Lighthouse 2
- Assignment 1
INFOMAGR – Advanced Graphics

Jacco Bikker - November 2019 – February 2020

END of “Introduction”

next lecture: “Whitted-style Ray Tracing”