Pathtracing Primer

CIS 565 - GPU Programming 2022

Adapted heavily from [Yining Karl Li's 2012 notes] and Austin Eng's Slides from 2017

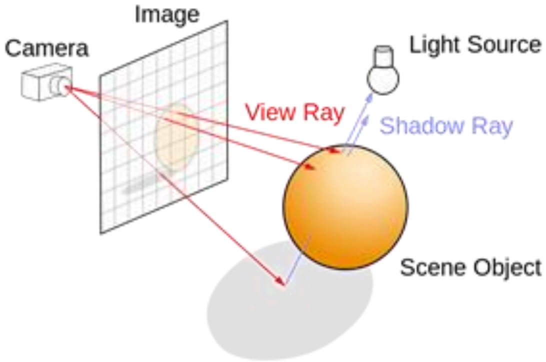
Agenda

- **Raytracing & Pathtracing Basics**
  - Theory TLDR: Raytracing, Pathtracing, Global Illumination
  - Bidirectional Scattering Distribution Functions (BSDFs)
- **Implementing a Pathtracer**
  - Recursive vs. iterative
  - What this means on the GPU
- **Features Overview**
  - Better Intersections: Spatial Hierarchies and Short-stack tree traversal
  - Motion Blur and Depth of Field
  - Multiple Importance Sampling (MIS)
  - Neat-o shaders
- **Inspiration: The Third & The Seventh! (optional viewing)**

Raytracing & Pathtracing Basics

- Techniques for creating images by emulating certain physical properties of light
- IRL, **rays** of light:

  - leave **light sources**
  - bounce around the scene, changing color / intensity based on **material**
  - some rays hit pixels within a **camera**'s view



- Raytracing / Pathtracing: simulate this in **reverse**

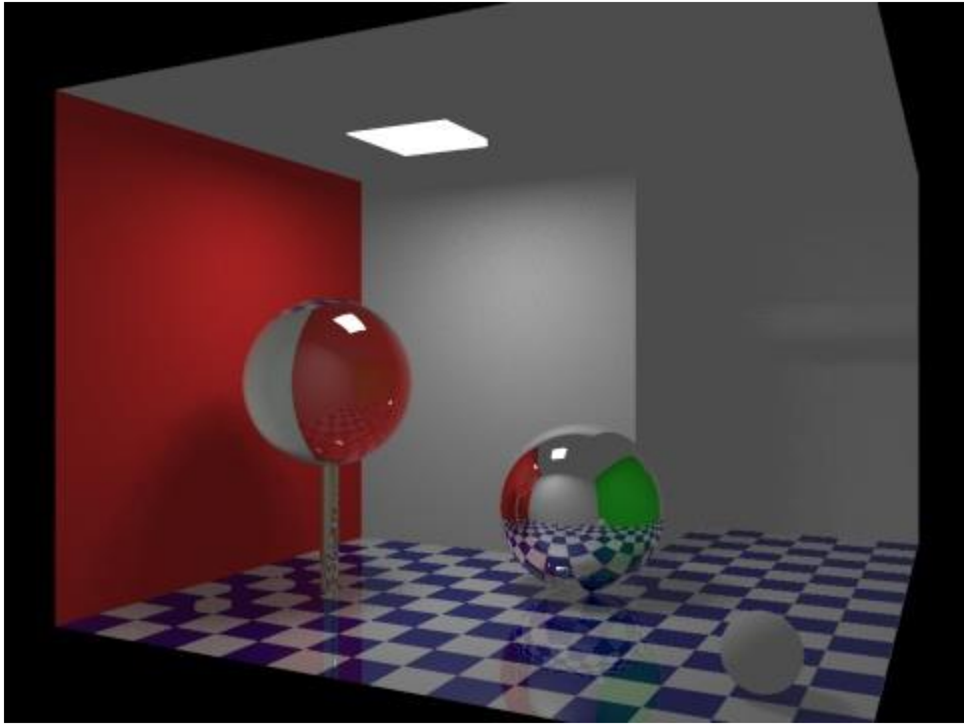
  - fire rays out of camera pixels
  - rays bounce around in scene
  - some hit light source

https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Raytracing Algorithm

1) For each pixel, shoot a ray into the scene
2) For each ray, check what surface it hits
3) If surface is not **reflective**/**refractive**, cast **shadow feeler** ray towards light(s) to determine if this part of surface is in shadow. Return a color for this ray's origin pixel based on surface and shadow status.
4) If surface is **reflective**/**refractive**, generate a **bounce** ray with a new start position and direction, repeat from step 2 with new ray

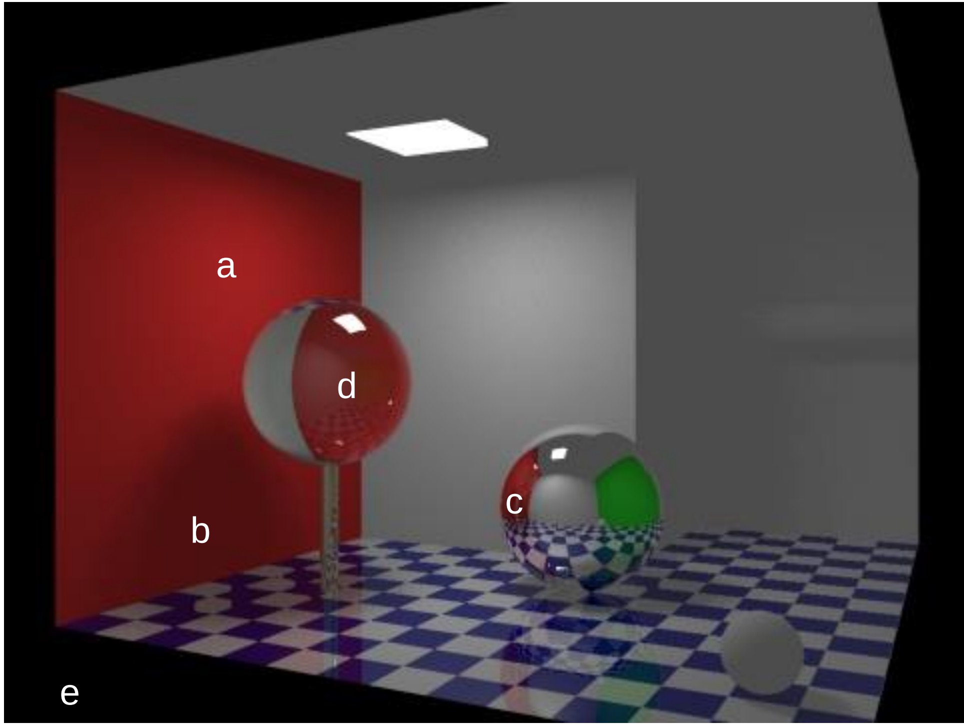
- Sequence of rays from camera to surfaces to light is called a **light path**
- Must limit number of **bounces** a path can take, for infinity-mirror effects



https://en.wikipedia.org/wiki/Global_illumination

Deconstructing a Raytraced Image

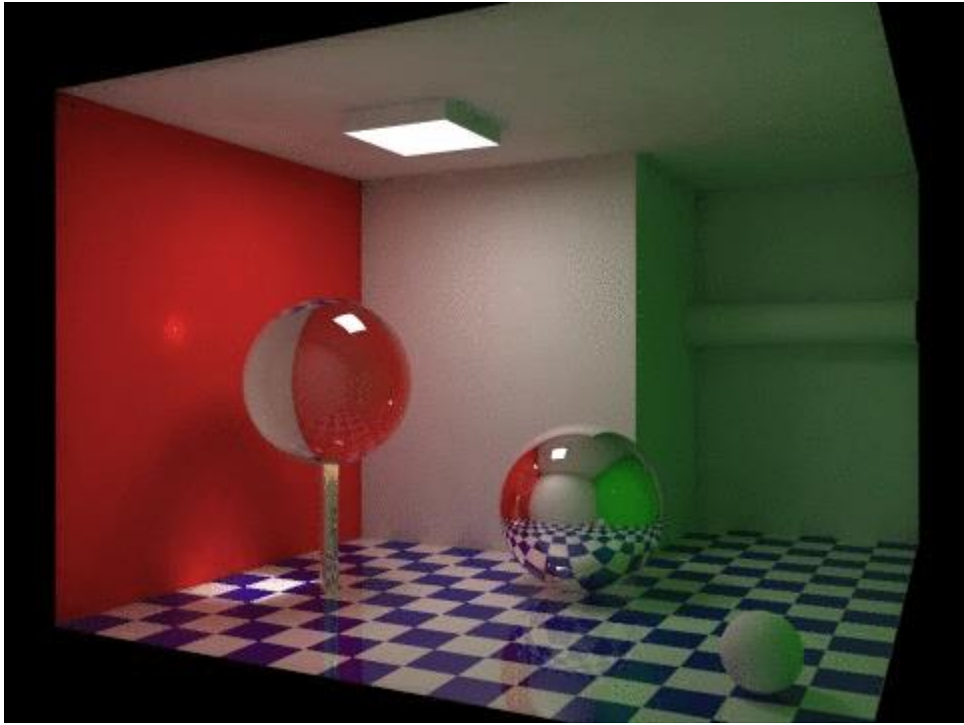
a) Hits wall, casts out another ray towards the light, detects that light is not “occluded.” Return “lit red” color based on angle to light.
b) Hits wall, casts out another ray towards the light, detects that it is blocked by the sphere. Return “shadowed red” color
c) Hits reflective sphere, fires new ray in “reflected” direction. New ray hits red wall, does same thing ray 1 does.
d) Hits refractive sphere, fires new ray out of sphere in “refracted” direction. New ray hits wall, does same thing ray 1 does.
e) Hits nothing, returns “no light energy”



https://en.wikipedia.org/wiki/Global_illumination

Pathtracing & Global Illumination

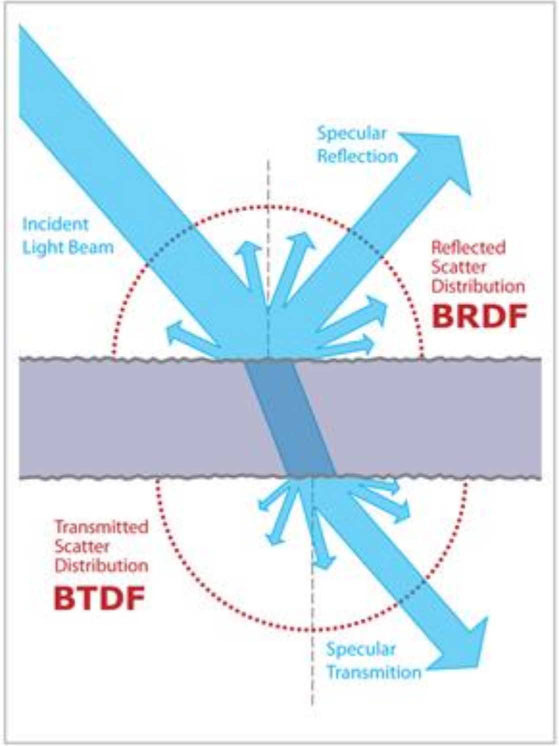
- Real light bounces off everything, not just reflective surfaces
- Ideally all rays hitting non-light surfaces in the scene should spawn **bounce** rays
- limit computation with a **bounce count** - after reaching a certain max length, ray paths “die”
- Describe how new rays should be generated using **Bidirectional Scattering Distribution Functions (BSDFs)**
- Lucky you! A basic pathtracer is algorithmically simpler!



https://en.wikipedia.org/wiki/Global_illumination

Bidirectional Scattering Distribution Functions

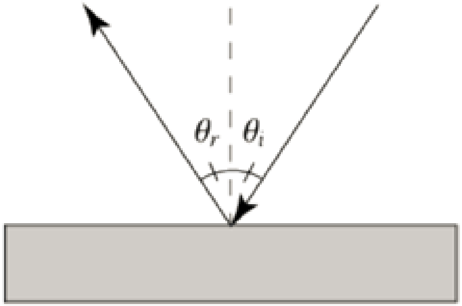
- Technically, combination of **reflection** and **transmission** functions
- **BSDF = BRDF + BTDF**
- defines how a new ray should be generated at an intersection
- Usually dependent on surface normal
- Some models:
    - Ideal Specular (perfect mirror)
    - Ideal Diffuse
    - Glossy/Specular, microfacet, etc.
    - Subsurface Scattering...

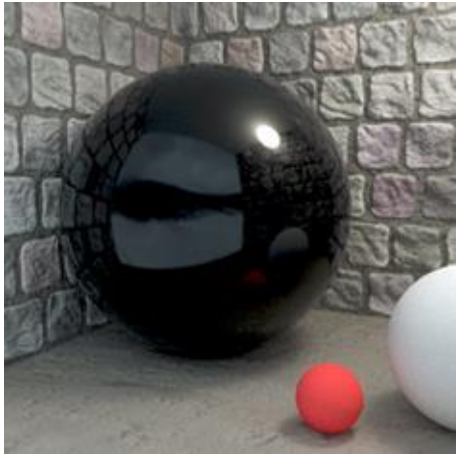


https://en.wikipedia.org/wiki/Bidirectional_scattering_distribution_function

Ideal Specular

- New ray is a perfect reflection of ray that hit this surface
- Use this for perfect mirrors
- Real specular materials can have color, so Ideal Specular material may “color” the ray
- More physically accurate version: approximate Fresnel effects using Schlick’s approximation

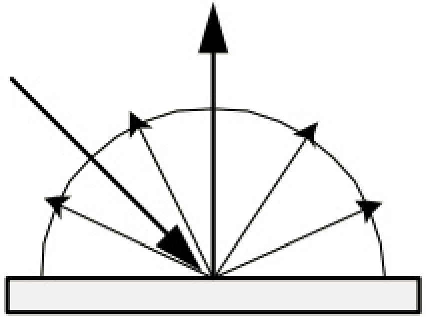


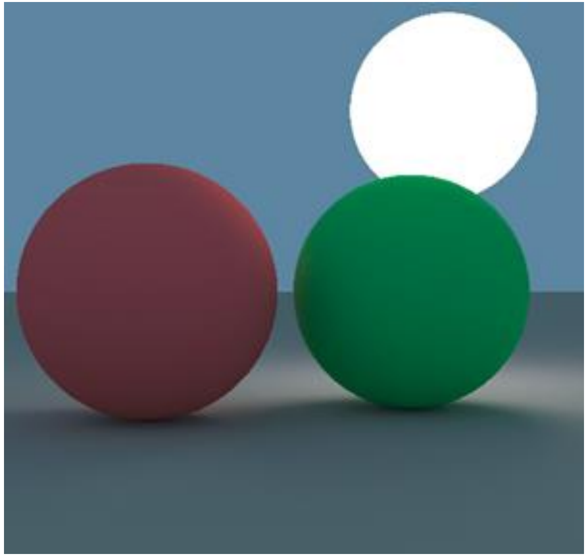


Photorealier, by Peter Kutz 2012

Ideal Diffuse

- Light can be reflected in any direction, based on hemisphere aligned with surface normal
- Usually, new rays spawned randomly over hemisphere
- More complex:
  - Micro-facet models
  - Subsurface reflection

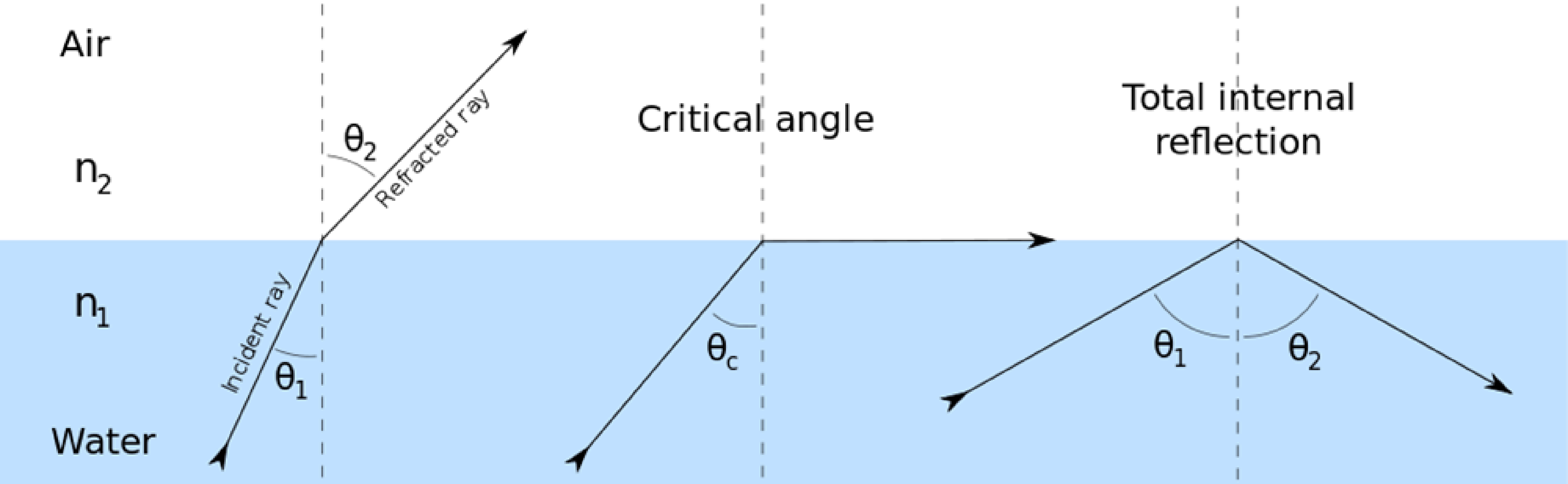




GPU Pathtracer, by Peter Kutz and Yining Karl Li

Ideal Refraction

- New ray generated based on surface normal, index of refraction of materials
- New angles computed using [Snell's law]
- More accurate implementation involves fresnel effects



Implementing a Pathtracer

1. For each pixel, shoot a ray into the scene
2. For each ray, compute the first surface it hits. Sample the emittance (if it's a light) or sample the material and BSDF for the surface and generate a new, **shaded** ray
3. Continue bouncing ray around until a path length limit is reached or it hits a light
4. Repeat steps 1-3 and continuously average result until an image forms

- Steps 1-3 are referred to as an **iteration**
- Each ray path (extending back through space-time to a pixel) referred to as a **sample**
- Since many common BSDFs generate new rays semi-randomly, may take many iterations to generate a clean image
- Some pathtracers save full ray paths, not just samples along the ray – allows for interesting post-processing possibilities


TAKUA Renderer
Yining Karl Li
2015

1 spp


TAKUA Renderer
Yining Karl Li
2015
8 spp


TAKUA Renderer
Yining Karl Li
2015
64 spp


All paths that hit the lights terminate
White walls appear colored due to color bleed from floor
TAKUA Renderer
Yining Karl Li
2015
64 spp
Soft shadows and reflections are “free”

Pathtracing on the GPU: Motivation, Caveats

- Individual ray paths are “embarrassingly parallel” - massive parallelism allows rapid processing of a single iteration (at least for “brute force”)
- rendering for movies uses **renderfarms** for massive CPU parallelism
- We can also work in parallel algorithms to take advantage of CUDA architecture (more in a bit)
- Obstacles:
    - Large scenes - may gigabytes or terabytes of geometry and textures – less of a problem now
    - Scene traversals and material evaluation – potential for a lot of branching

Encore: Implementing a Pathtracer

Basic (CPU) Pathtracing Algorithm:

1) For each pixel, shoot a ray into the scene
2) For each ray, compute the first surface it hits. Sample the emittance and return (if it’s a light) or sample the BSDF for the surface and generate a new ray
3) Continue bouncing the ray around until a path length limit is reached or it hits a light
4) Repeat steps 1-3 and continuously accumulate result until an image forms

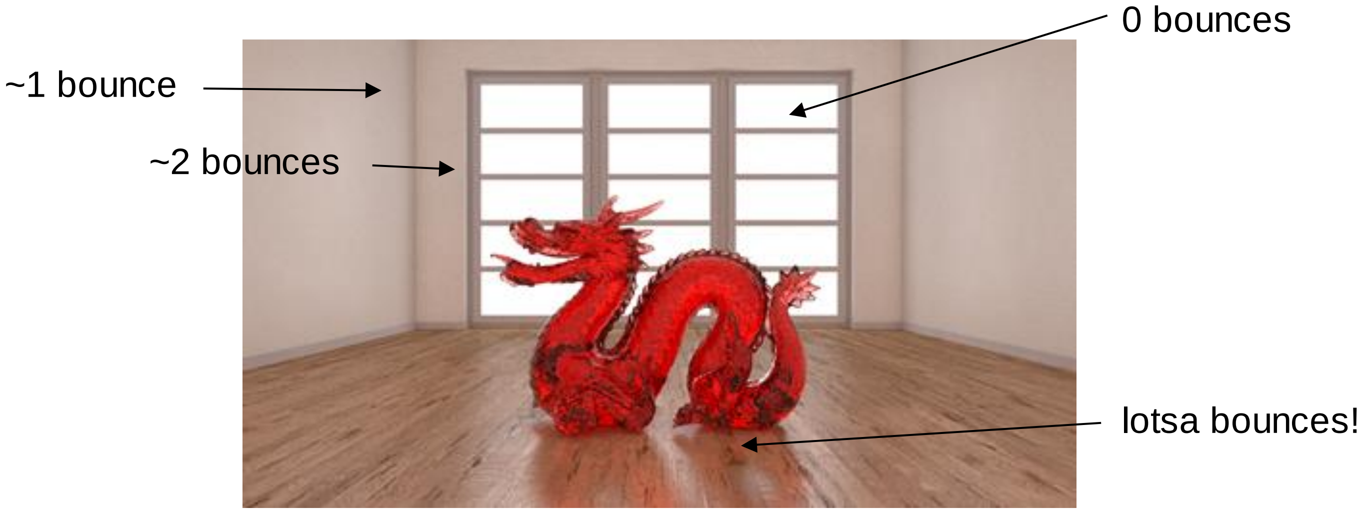
- Seems like this could be a very recursive algorithm
- So, parallelize by 1), each pixel?

Recursive Pathtracing

```
color3 rayTracePixel(int depth, ray currentRay, vector<geom> objects) {
  // Determine closest intersected object material mat,
  // intersection normal norm, intersection point pt.
  // Return "black" if no intersection is detected

  // Terminate path if the ray hits a light
  if (mat.isEmissive) {
    return currentRay.color * mat.color;
  }
  if (depth > 0) {
    newRay = computeNewRay(mat.bsdf, currentRay, norm, pt);
    newRay.color = currentRay.color * mat.color;
    return rayTracePixel(depth - 1, newRay, objects);
  }
  else {
    return black; // Bottomed out without hitting a light
  }
}
```

What’s wrong with this implementation?

- CUDA doesn’t support recursion, except on Fermi or newer
  - even on Fermi and newer, recursion is Considered Harmful™ (slow)
- Threads can be in flight for a long time
- Threads will almost certainly finish at different times -> divergence



Pathtrace iteratively instead?

- Use a loop and cache current ray for use in next iteration?

```
color3 rayTracePixelIterative(int depth, ray startRay, vector<geom> objects) {
  ray currentRay = startRay;
  for (int i = 0; i < depth; i++) {
    // Determine closest intersected object material mat,
    // intersection normal norm, intersection point pt.
      // Return "black" if no intersection is detected

    // Terminate path if the ray hits a light
    if (mat.isEmissive) {
      return currentRay.color * mat.color;
    }

              // Otherwise, launch new ray.
    newRay = mat.bsdf.computeNewRay(currentRay, norm, pt);
    newRay.color = currentRay.color * mat.color;
    currentRay = newRay;
  }
  return currentRay.color;
}
```

What’s wrong with this (iterative) implementation?

- ~~CUDA doesn’t support recursion except on Fermi or newer~~ (solved!)
- Threads can be in flight for a long time
- Threads will almost certainly finish at different times -> divergence

0 bounces

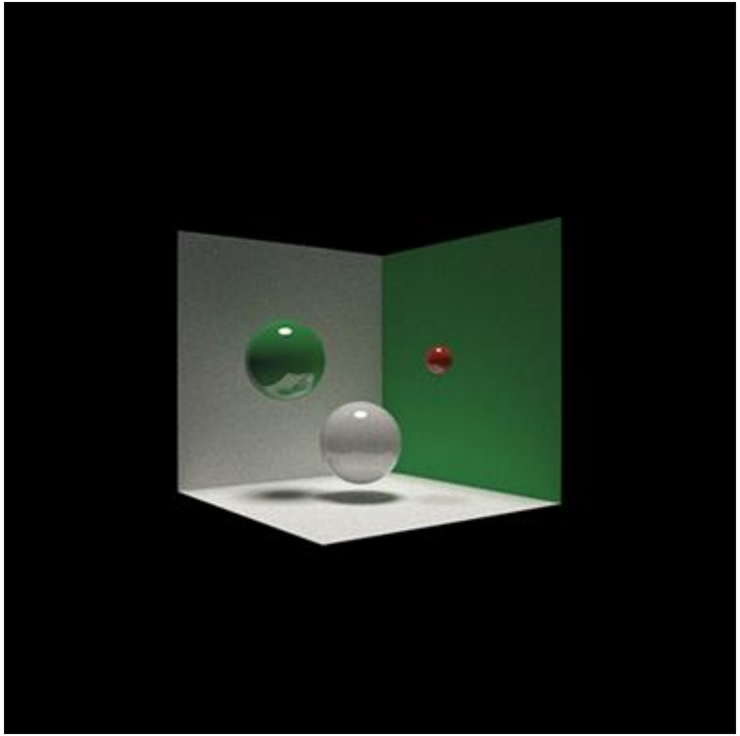
~1 bounce

~2 bounces

lotsa bounces!

A closer look at divergence - wasted cycles

- How many bounces does each path take before returning?
- Remember: diffuse materials usually involve random directions on new rays!

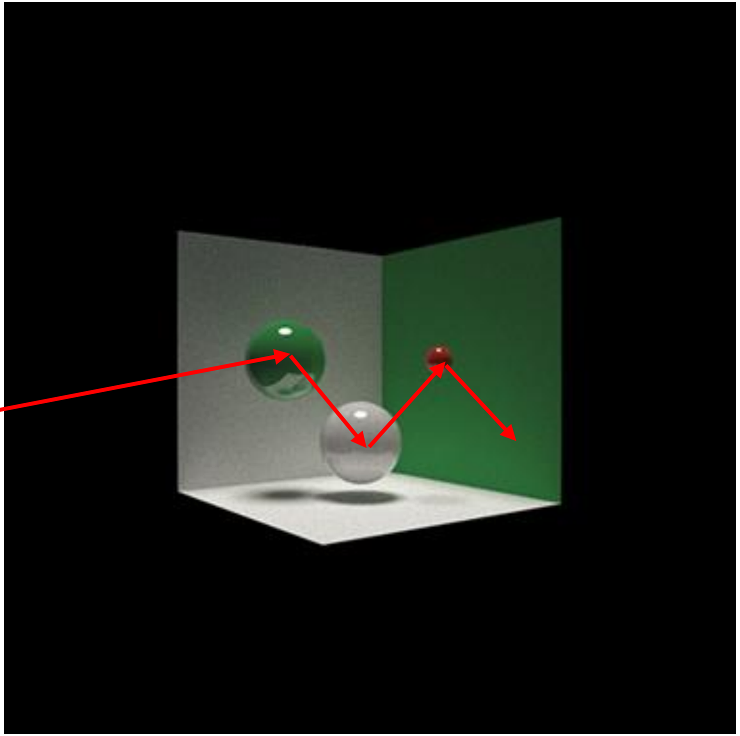


TAKUA renderer, Yining Karl Li, 2012

A closer look at divergence - wasted cycles

- How many bounces does each path take before returning?
- Remember: diffuse materials usually involve random directions on new rays!

4 bounces?

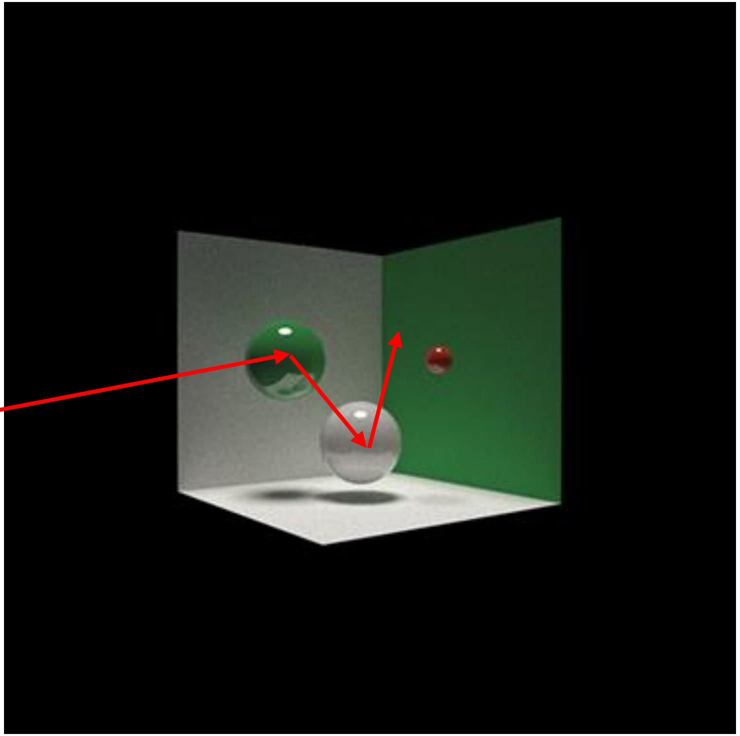


TAKUA renderer, Yining Karl Li, 2012

A closer look at divergence - wasted cycles

- How many bounces does each path take before returning?
- Remember: diffuse materials usually involve random directions on new rays!

3 bounces?

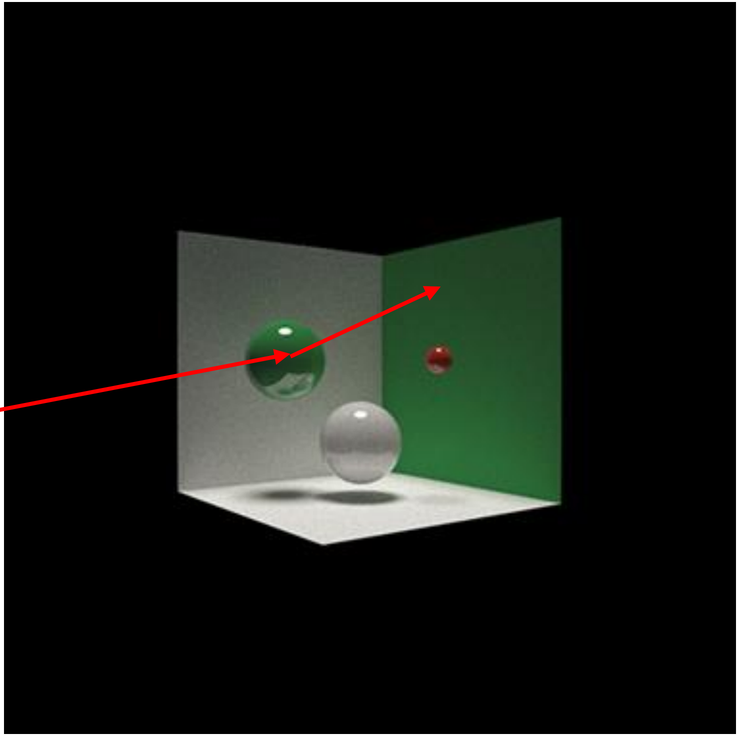


TAKUA renderer, Yining Karl Li, 2012

A closer look at divergence - wasted cycles

- How many bounces does each path take before returning?
- Remember: diffuse materials usually involve random directions on new rays!

2 bounces?

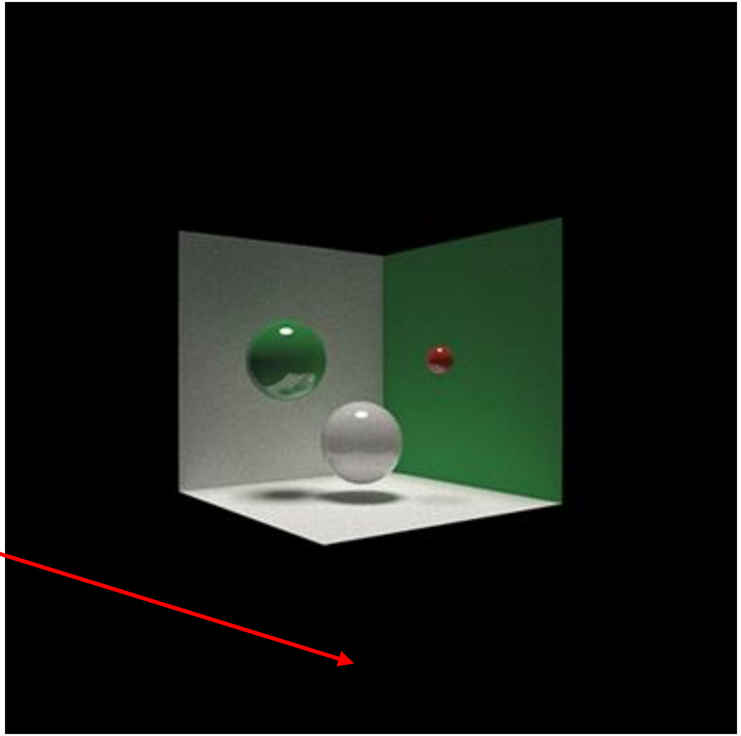


TAKUA renderer, Yining Karl Li, 2012

A closer look at divergence - wasted cycles

- How many bounces does each path take before returning?
- Remember: diffuse materials usually involve random directions on new rays!

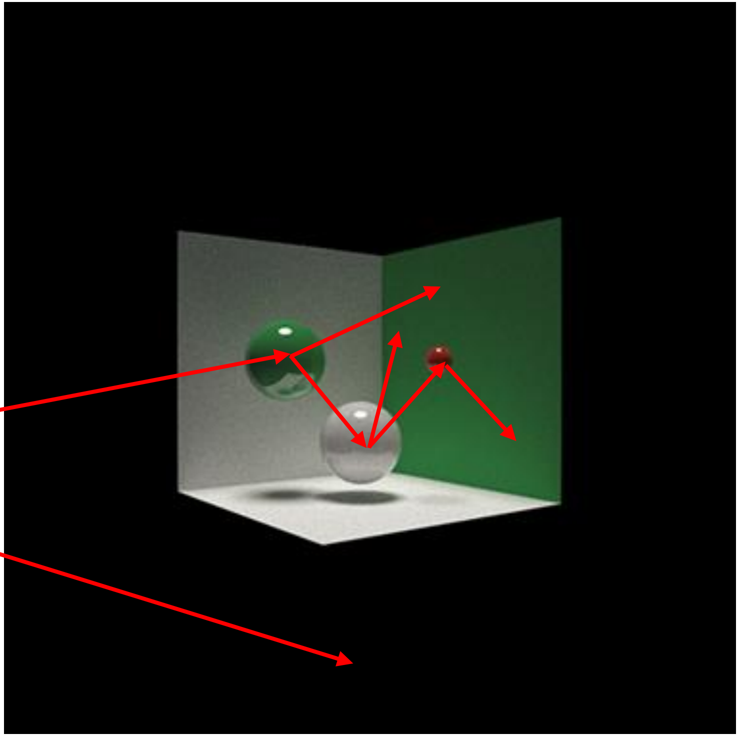
No bounces?



TAKUA renderer, Yining Karl Li, 2012

A closer look at divergence - wasted cycles

- How many bounces does each path take before returning?
- Remember: diffuse materials usually involve random directions on new rays!



- Uncertain how long paths for different pixels/different iterations will be
- So parallelizing by pixel is a bad idea...

TAKUA renderer, Yining Karl Li, 2012

A closer look at divergence - wasted cycles

- Recall in CUDA: can only launch a finite number of blocks at a time
- If some threads are tracing more bounces and some are only tracing a few, can end up with a lot of idling threads

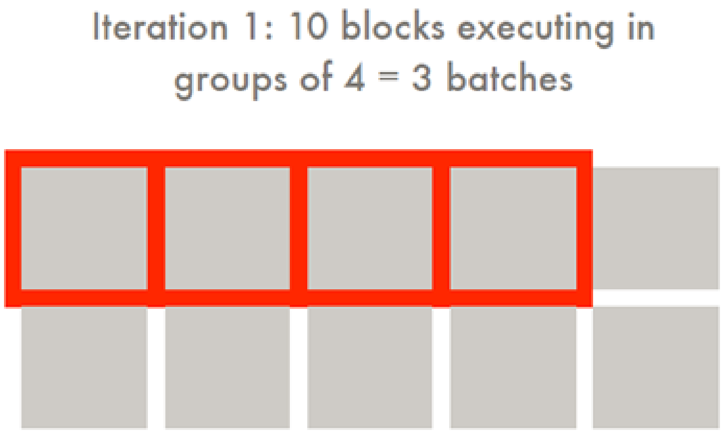
Thread 1	Thread 2	Thread 3	Thread 4	Thread 5
Bounce 1	Bounce 1	Done!	Bounce 1	Bounce 1
Bounce 2	Done!	idling...	Bounce 2	Done!
Bounce 3	idling...	idling...	Done!	idling...
Bounce 4	idling...	idling...	idling...	[illegible]ing...
Done!	idling...	[illegible]	[illegible]	[illegible]ing...

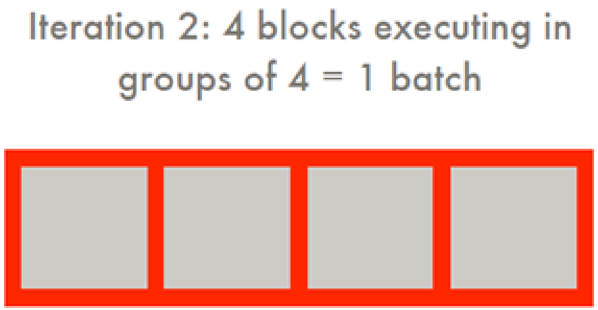
Solution: Parallelize by Rays!

- In any iteration, each ray in a path must be:
  - Checked against scene geometry
  - “Shaded” -> check for termination, generate new ray using BSDF, compute color change
- Instead of doing an entire path at once, maintain a pool of rays
- Launch a kernel that traces ONE bounce for every ray in the pool, updates the results
- Remove terminated rays from the ray pool with stream compaction
- Rinse and repeat

More on Ray Parallelization

- Ray pool can only stay the same size or get smaller with each iteration
- Stream compaction lets us reduce threads needed -> each iteration should generally execute faster than previous





Ray Parallelization: Super Simple Example

1st Kernel Launch

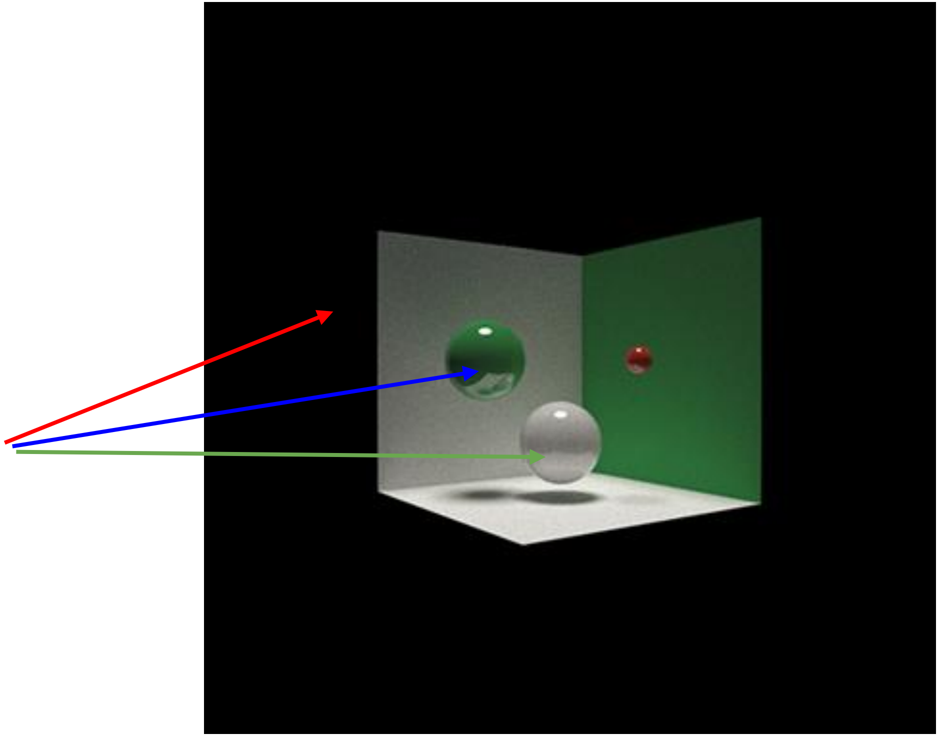
Ray Pool:

Ray 1, Ray 2, Ray 3

Threads Needed: 3

Terminated Rays:

Ray 1



TAKUA renderer, Yining Karl Li, 2012

Ray Parallelization: Super Simple Example

2nd Kernel Launch

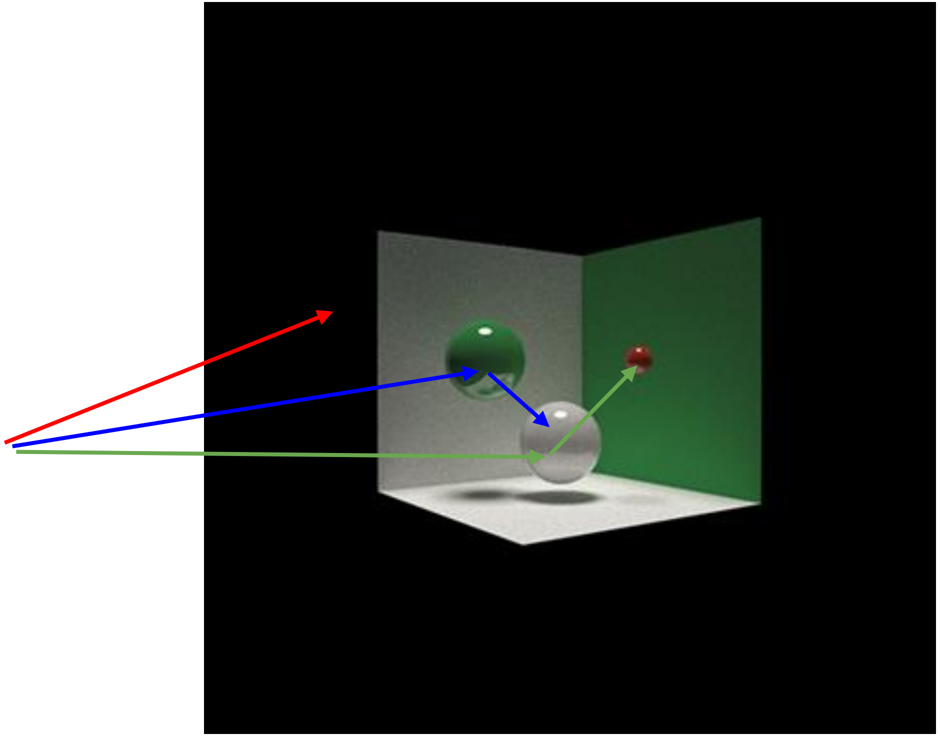
Ray Pool:

Ray 2, Ray 3

Threads Needed: 2

Terminated Rays

Ray 1



TAKUA renderer, Yining Karl Li, 2012

Ray Parallelization: Super Simple Example

3rd Kernel Launch

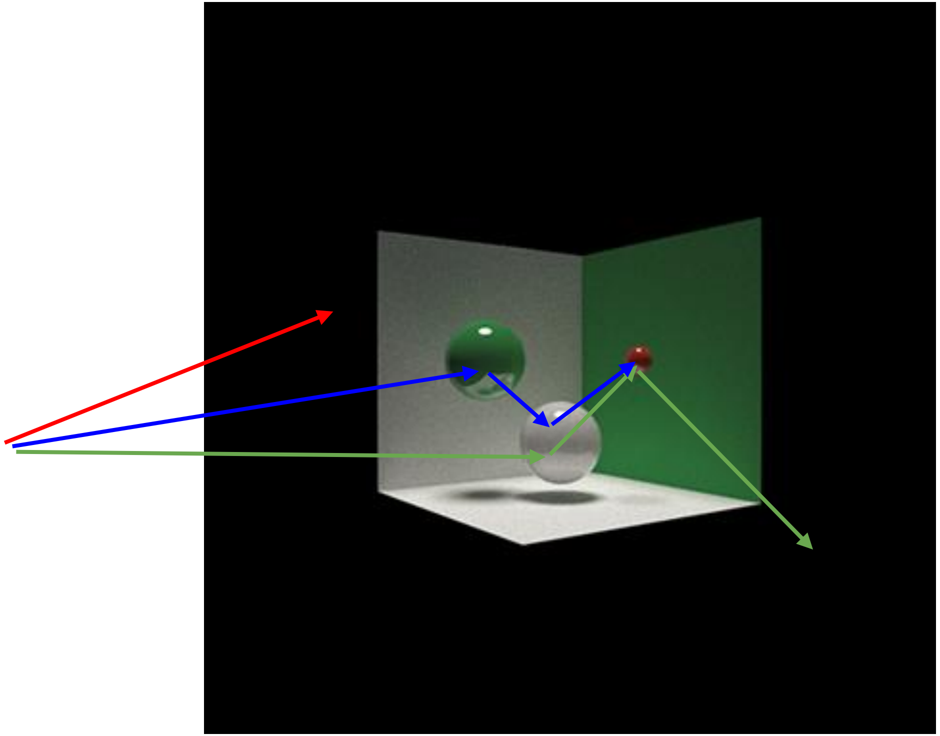
Ray Pool:

Ray 2, Ray 3

Threads Needed: 2

Terminated Rays

Ray 1, Ray 3



TAKUA renderer, Yining Karl Li, 2012

Ray Parallelization: Super Simple Example

4th Kernel Launch

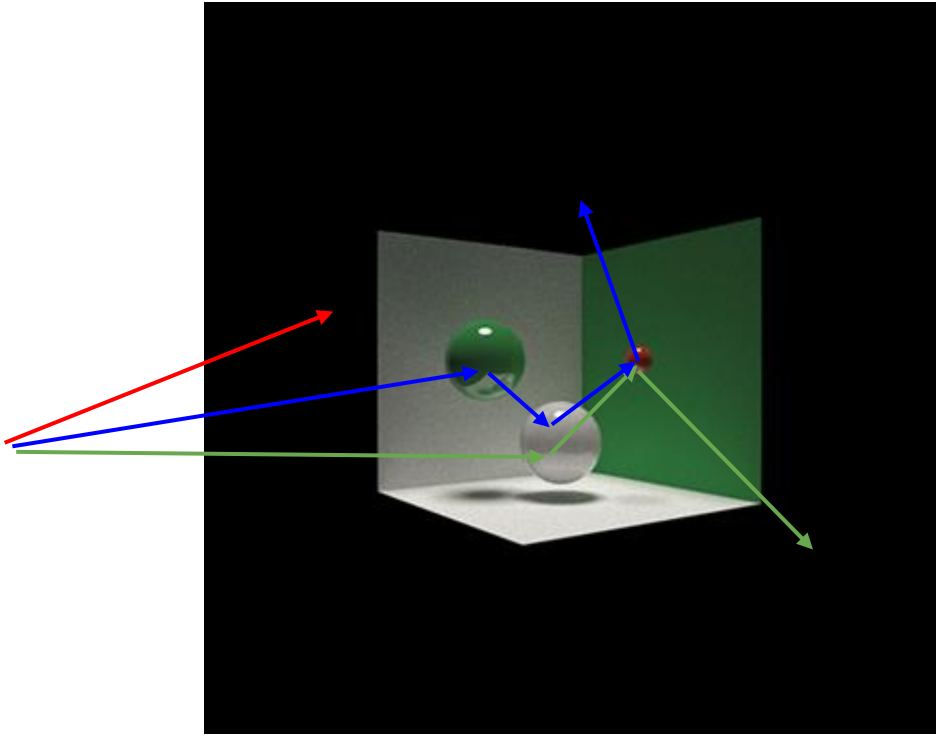
Ray Pool:

Ray 2

Threads Needed: 1

Terminated Rays

Ray 1, Ray 3, Ray 2



TAKUA renderer, Yining Karl Li, 2012

Ray Parallelization: Super Simple Example

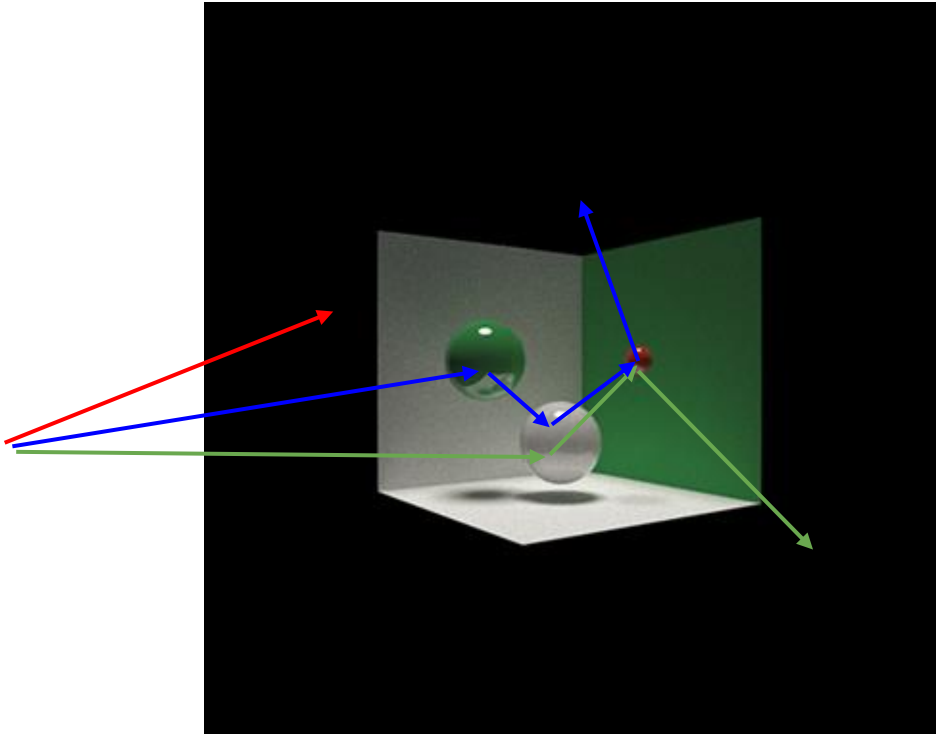
4th Kernel Launch

Ray Pool:

Threads Needed: 1

Terminated Rays

Ray 1, Ray 3, Ray 2



TAKUA renderer, Yining Karl Li, 2012

One more time: What’s wrong with this implementation?

- Threads can be in flight for a long time
- Threads will almost certainly finish at different times
- Why are these still problems??!
- Expensive BSDF computations lead to longer computation for some rays

Blue Sky Animation Studios

Solution: Sort by material type

- Continue parallelizing by ray
- Perform intersection testing and shading/BSDF evaluation in separate kernels
- Referred to as a “wavefront” of intersection testing, then shading/evaluation
- Use parallel radix sort to batch by material type

Ray:	a	b	c	d	e	f	g
Material:	Mirror	Glass	Picture	Glass	Glass	Mirror	Picture

Radix Sort by Material ID

Ray:	a	f	c	g	e	b	d
Material:	Mirror	Mirror	Picture	Picture	Glass	Glass	Glass

Base Code Intro

Overview: Highest Level

Initialize Scene

↓

Pathtrace (a couple of times)

↓

Save Image

Base Code

```
parse in scene file
initialize buffers
```

```
shoot rays from camera
for i in [0...iterations]:
    Compute intersections

    Shade the ray
    Bounce it off

Collect ray colors
```

```
fill pixel buffer
write image to disk()
```

What you have right now.

```
Scene = new Scene(file);
PathtraceInit();
```

pathtrace

```
generateRayFromCamera()
for i in [0...iterations]:
    computeIntersection()

    shadeFakeMaterial()

finalGather()
```

```
sendImageToPBO()
saveImage()
```

What they're called in the code.

```
parse in scene file
initialize buffers

shoot rays from camera
for i in [0...iterations]:
    Compute intersections

    Shade the ray
    Bounce it off

Collect ray colors

fill pixel buffer
write image to disk()
```

Back to a little higher level...

```
parse in scene file
initialize buffers
```

↓

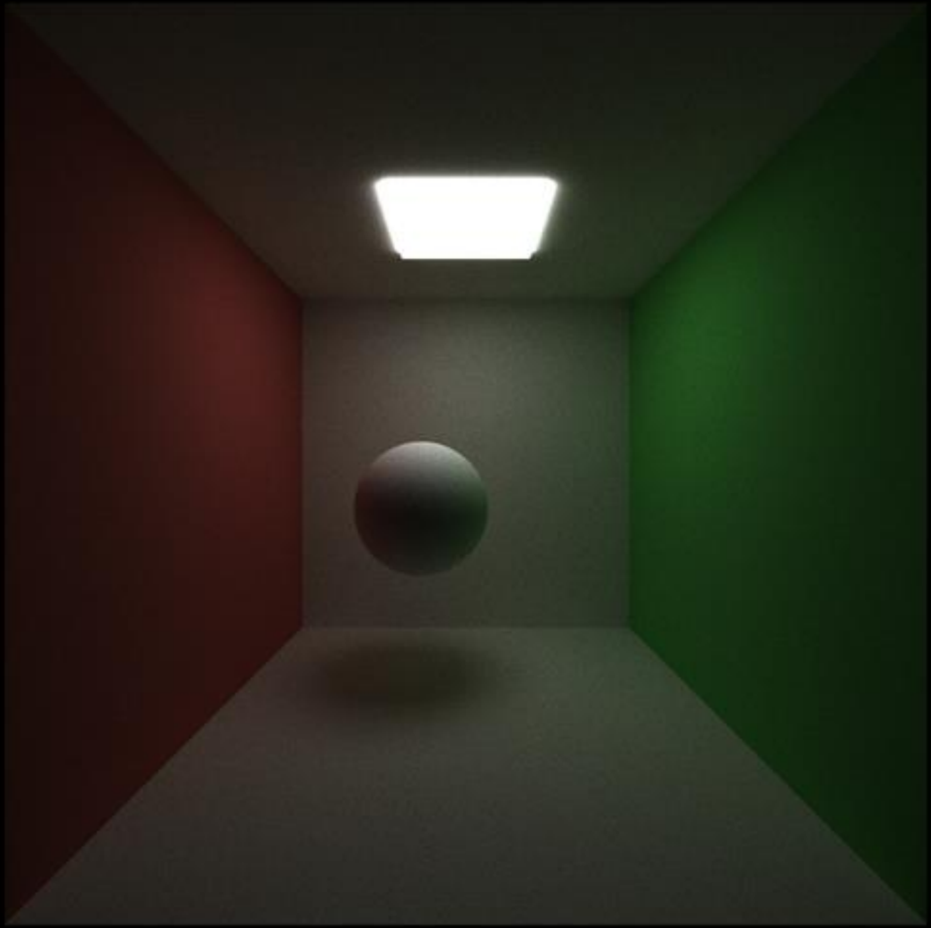
```
shoot rays from camera
for i in [0...iterations]:
    Compute intersections
    Sort rays by material
    Shade the ray
    Bounce it off
    Stream Compact
Collect ray colors
```

↓

```
fill pixel buffer
write image to disk()
```

What You have to implement for **Part 1:**

- Ray sorting by material
- Ideal Diffuse Shading & Bounce
- Perfect Specular Reflection
- Stream Compaction
- Cache first bounce



```
parse in scene file w/ mesh?
initialize KD Tree?
```

```
shoot rays from camera
for i in [0...iterations]:
    Compute Refraction?
    Sort rays & rebounce?
    Shade the ray (MIS? Direct?)
    Bounce it off (Subsurface?)
    Stream Compact (Shared Mem?)
Collect ray colors
```

```
fill pixel buffer
write image to disk()
```

Features for Part 2:

- Arbitrary Mesh Loading (obj, gltf, etc..)
- Hierarchical Data Structure
- Procedural Shapes & Textures
- Motion Blur (with Motion)
- Wavefront Path Tracing
- Two of: (DOF, Anti-aliasing, Refraction)
- Direct Lighting Shading
- Multiple Importance Sampling
- Texture Mapping & Bump Mapping
- Subsurface Scattering
- Work Efficient Stream Compaction

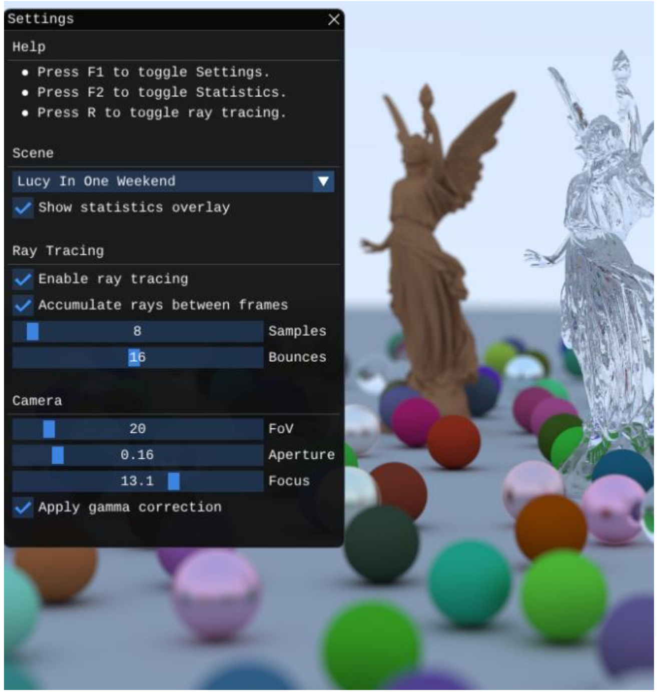
```
parse in scene file w/ mesh?
initialize KD Tree?

shoot rays from camera
for i in [0...iterations]:
    Compute Refraction?
    Sort rays & rebounce?
    Shade the ray (MIS? Direct?)
    Bounce it off (Subsurface?)
    Stream Compact (Shared Mem?)
Collect ray colors

fill pixel buffer
write image to disk()
```

The more you add, the more points you will get.

The cooler/harder the feature the higher the points!

ImGui

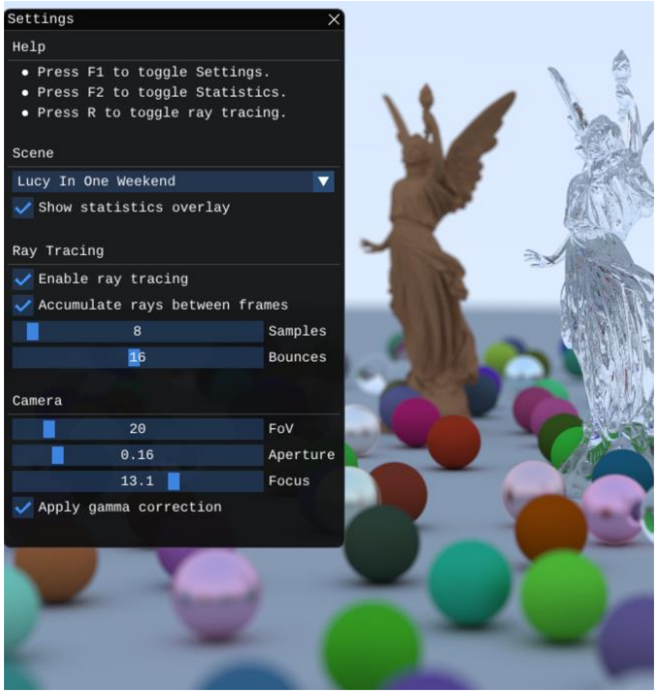
- Ability to tweak parameters for example camera FOV, aperture, Focal distance.
- A dialog box to easily select scene file, instead of boring command line.
- Analytics data such number of alive rays, kernel time, material shader time etc.
- Just go crazy, make your path tracer application more informative and versatile!



ImGui

ImGui

- Check out preview.cpp RenderImGui() method. Examples on usage.
- Check out GuiDataContainer class in utilities.h. A container for sharable ImGui Data between pathtrace.cu and preview.cpp.
- Check out pathtrace.cu pathtrace() method to see how depth variable is being updated and sent to preview.cpp RenderImGui().



ImGui

Project Structure

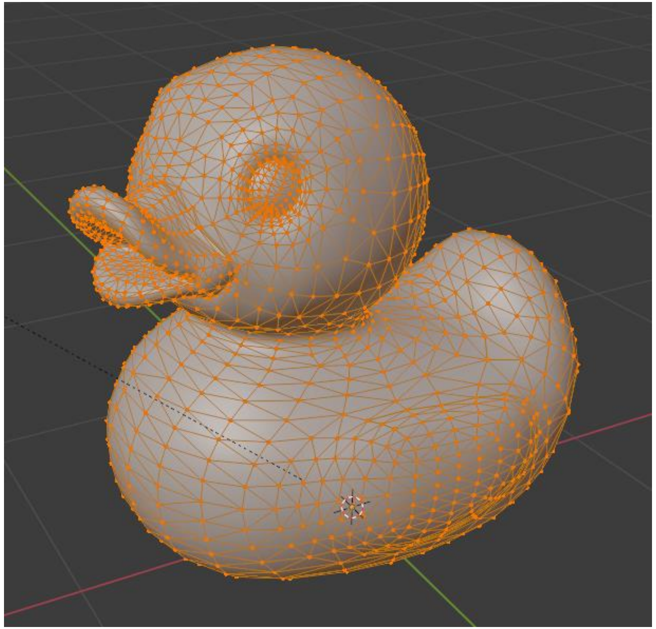
Project Structure

- Project is divided into two parts:
- Required Core – Basic Parallel Path Tracer with BSDF etc
- “Choose Your Own Adventure” – Pick features you want to implement
  - Each feature has a “Feature Score” associated with it
  - You are required to implement features totaling up to at least 10 feature score points
  - Feature Score Points beyond the required 10 will be counted as extra credit
    - Approx. 5 project grade points per 1 Feature Score Point
    - For example: implementing 14 feature score points = 10 required + 4 eligible for Extra Credit
  - Feature ideas present in the INSTRUCTION.md file
  - If you have additional ideas, you can propose and get approval via Piazza

Features Intro
(Non-Exhaustive)

Mesh Loading

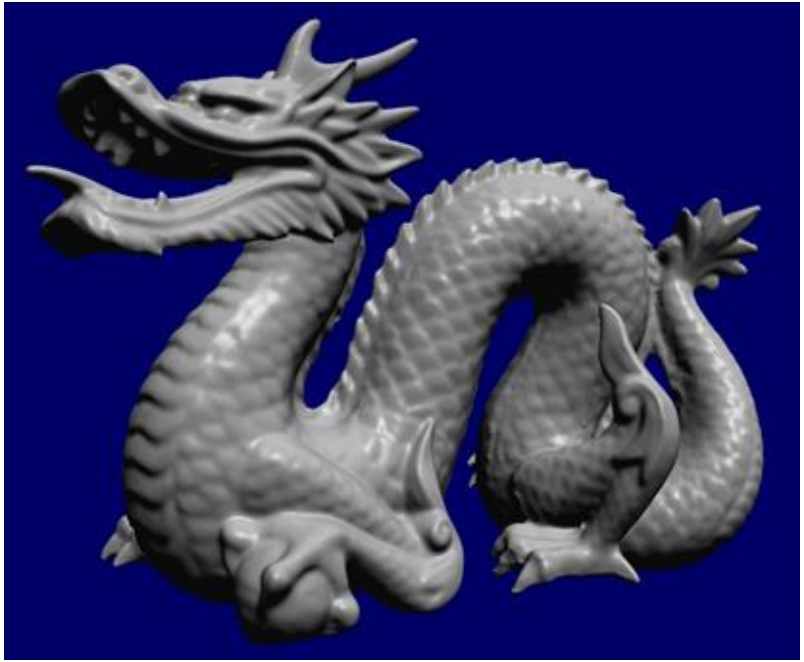
- Ability to load a triangle mesh from glTF or OBJ from scene file
- Quick way to make your scene more impressive and unique – load a model in the scenefile!
- Only triangle mesh loading required – additional glTF features may take considerable effort in the project time
- Naive way to raytrace a triangle mesh:
  In each thread, check if the ray intersects each triangle



[GlTF Sample models – Duck]

Intersection Testing (1) - cache first intersections

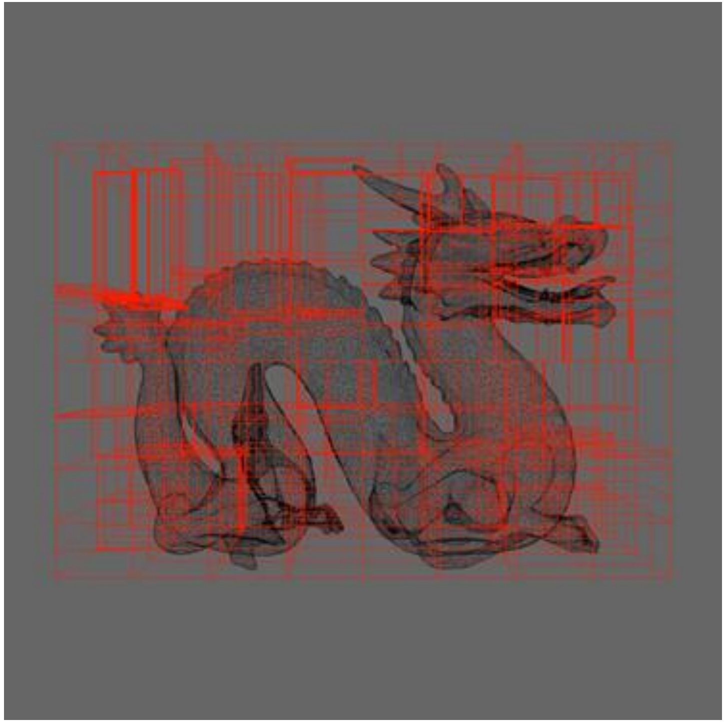
- Naive intersection testing: check ray against each primitive
  - OK for Cornell Box - sphere, cube primitives
  - But for triangle meshes?
  - Stanford Dragon? ~5 million triangles?
- Can cache intersections for first bounce, which will be the same across all iterations
- Diminishing returns as bounce counts increase
- Combine with other effects by shooting many rays per pixel on the first pass



Stanford 3D Scanning Repository

Intersection Testing (2) - Spatial Hierarchies

- Similar to Uniform Grid: preprocess primitives into a spatial data structure for coarse-level intersection culling
- Something like an Octree, KD Tree, or Bounding Volume Hierarchy
- Build on the CPU, or if you want a challenge, build on the GPU
- Depth-limited for GPU iterative traversal
- “Short Stack KD Tree Traversal”



[Yining Karl Li - Short-Stack KD Tree Traversal]

Motion Blur

- Iterations for a frame happens over a “timestep”
- Rays cast over the timestep
- So model (or camera!) may have a slightly different transformation for each ray

https://en.wikipedia.org/wiki/Motion_blur

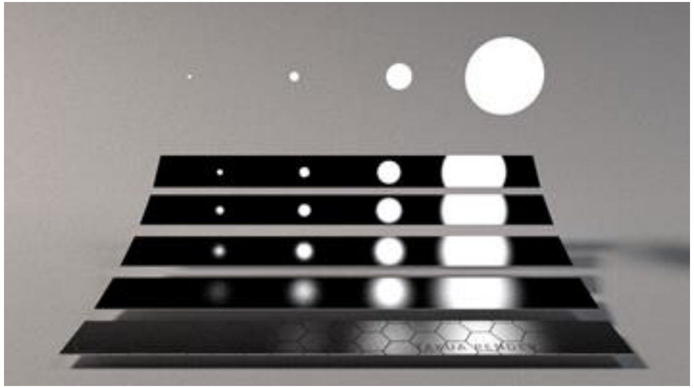
Depth of Field

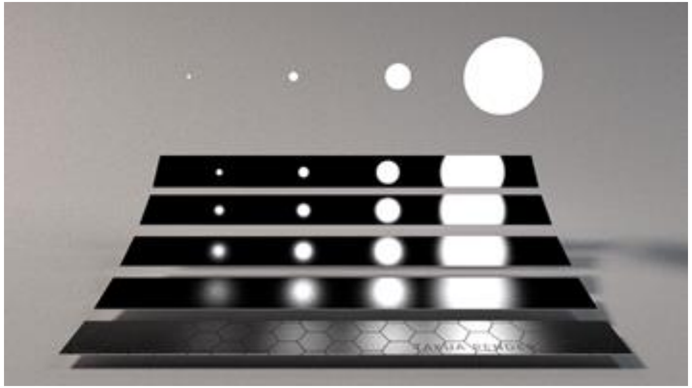
- Up to this point, assumed all rays originate from a single point
- For Depth of Field, can offset ray origin on a lens

https://en.wikipedia.org/wiki/Depth_of_field

Multiple Importance Sampling (MIS)

- Weight contributions of additional sampling methods
- Typical example: sample lights directly - for semi-reflective materials and small lights, why is this better with the same number of samples?
    - Below: semi-glossy surfaces and varying size lights, w/out MIS
    - Without MIS: highlights from the small light are poorer

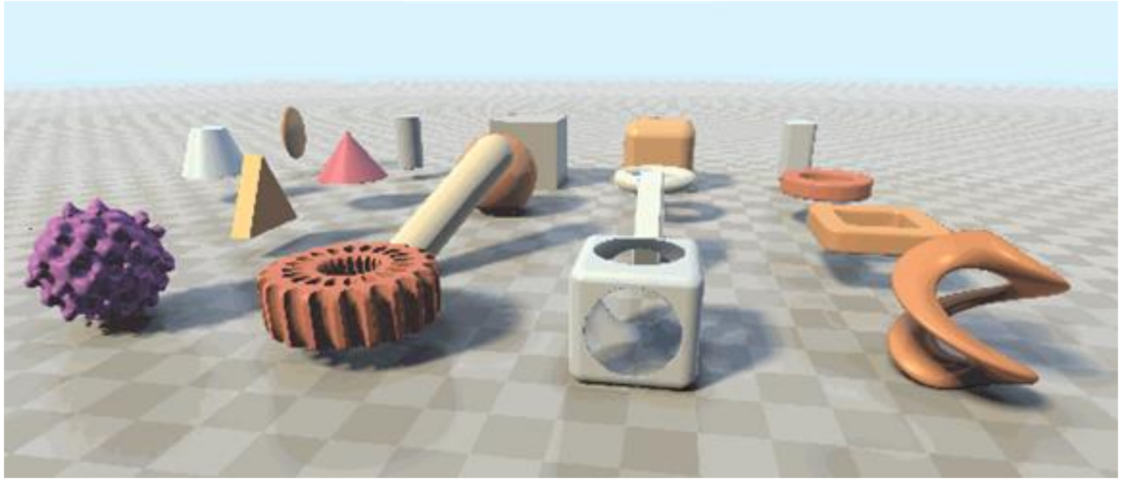




[Yining Karl Li - Multiple Importance Sampling]

Procedural Shapes

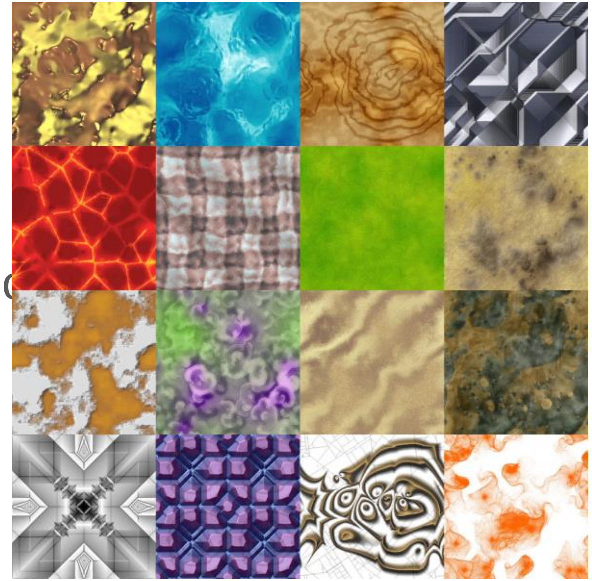
- Proceduralism is cool!
- Generate and shade shapes using just code?
- Sometimes take the form of "signed distance functions," which may require raymarching



Inigo Quilez - https://www.shadertoy.com/view/Xds3zN

Procedural Textures

- Proceduralism is cool!
- Color, shade, texture materials based on code
- Can make for really unique images
- Examples for procedural textures are everywhere



wikipedia.org/wiki/Procedural_texture

Open Image Denoiser (or any other “smart” denoiser)

- Not. Simple.
- CPU-based open source denoiser
- Takes in raw path tracer output buffer from 1spp to inf
- Applies a filter to the image
- Can also take in other buffers to preserve detail
- To get full credit, you must add an additional buffer

Open Image Denoiser

- Example :

Scene courtesy of Frank Meinl, downloaded from Morgan McGuire's Computer Graphics Archive.

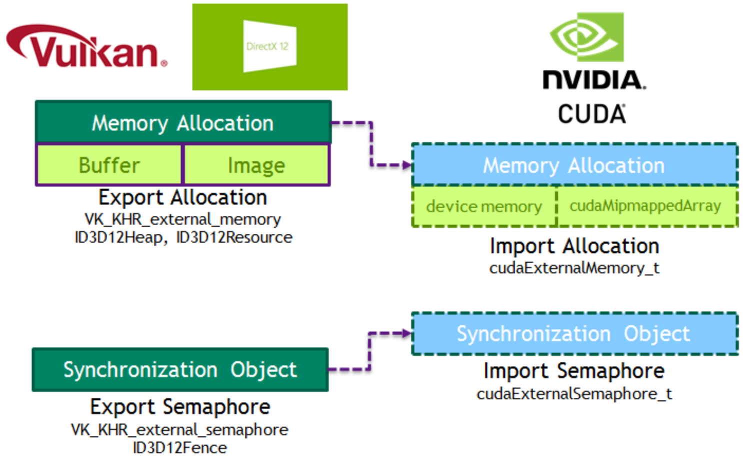
Neat-o shaders: Subsurface Scattering

- Subsurface Scattering: simulation of light bouncing around inside translucent materials
  - Skin
  - Milk
  - Marble
  - Many other things!
- Interesting problem in rendering
  - Bounces within a substance: potentially very computationally expensive
  - Approximations also welcome
- Check out Peter Kutz's notes

Peter Kutz - Multiple Scattering in Photorealizer

CUDA-Vulkan Interop

- The Path Tracer (as well as Project 0 and 1) use CUDA-OpenGL Interop
  - This API allows you to copy CUDA memory into OpenGL buffers without copying to CPU
  - https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__OPENGL.html
- Change the CUDA-OpenGL Interop to use CUDA-Vulkan Interop
  - ie. Switch to Vulkan based calls
- Why should you do this?
  - If you plan to use Vulkan for final project, this would be a great start
  - High feature score points
  - Immortality! This will become part of future Path Tracer projects
- Talk to Janine if interested
  - Some of the work has already been done



Inspiration: The Third & The Seventh

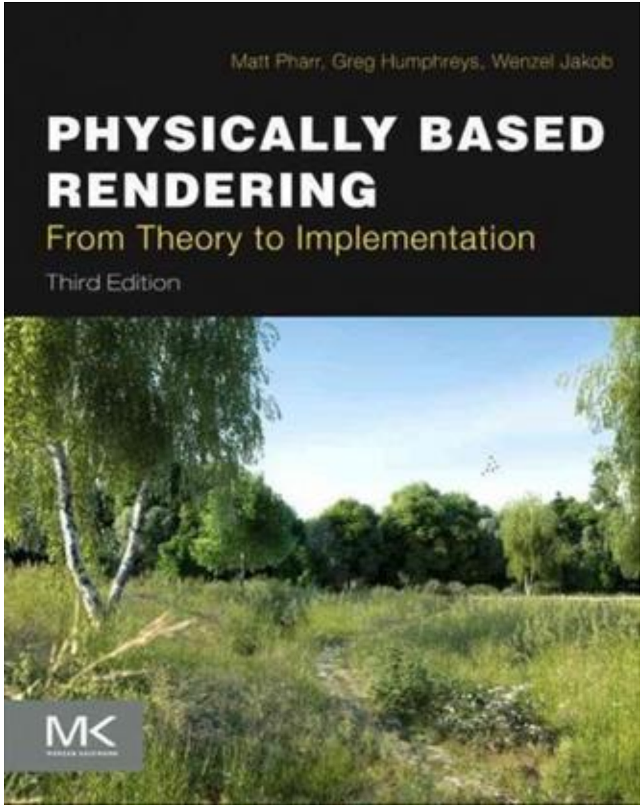
- Optional Viewing:
- By Alex Roman
- And ONLY Alex Roman!
- Rendered using Vray, a commercial renderer with a path tracer + some other Global Illumination techniques
- This is from 2009!!!

[Alex Roman - The Third & The Seventh]

For More Information:

- Check out:
- Physically Based Rendering: From Theory to Implementation!

- (Copies available all over in the SIG Lab, any edition is fine - also available online now at: https://www.pbrt.org/)



README

- CUDA Path Tracer gives you immense opportunity to create amazing README
- For 2022, submission is divided into two parts:
- October 3rd: Entire implementation + README with standard expectations
- 2 Additional README Days (October 5th): Make updates to your README and Scenes ONLY
  - Better images, comparisons, performance analysis
  - New scenes
  - Scenes that may take longer to render

- Late days apply to October 3rd deadline. You get 2 additional days from there for README updates
  - Oct 3 + Late Days = Project Submission
  - Oct 3 + Late Days + 2 README Days = README Submission

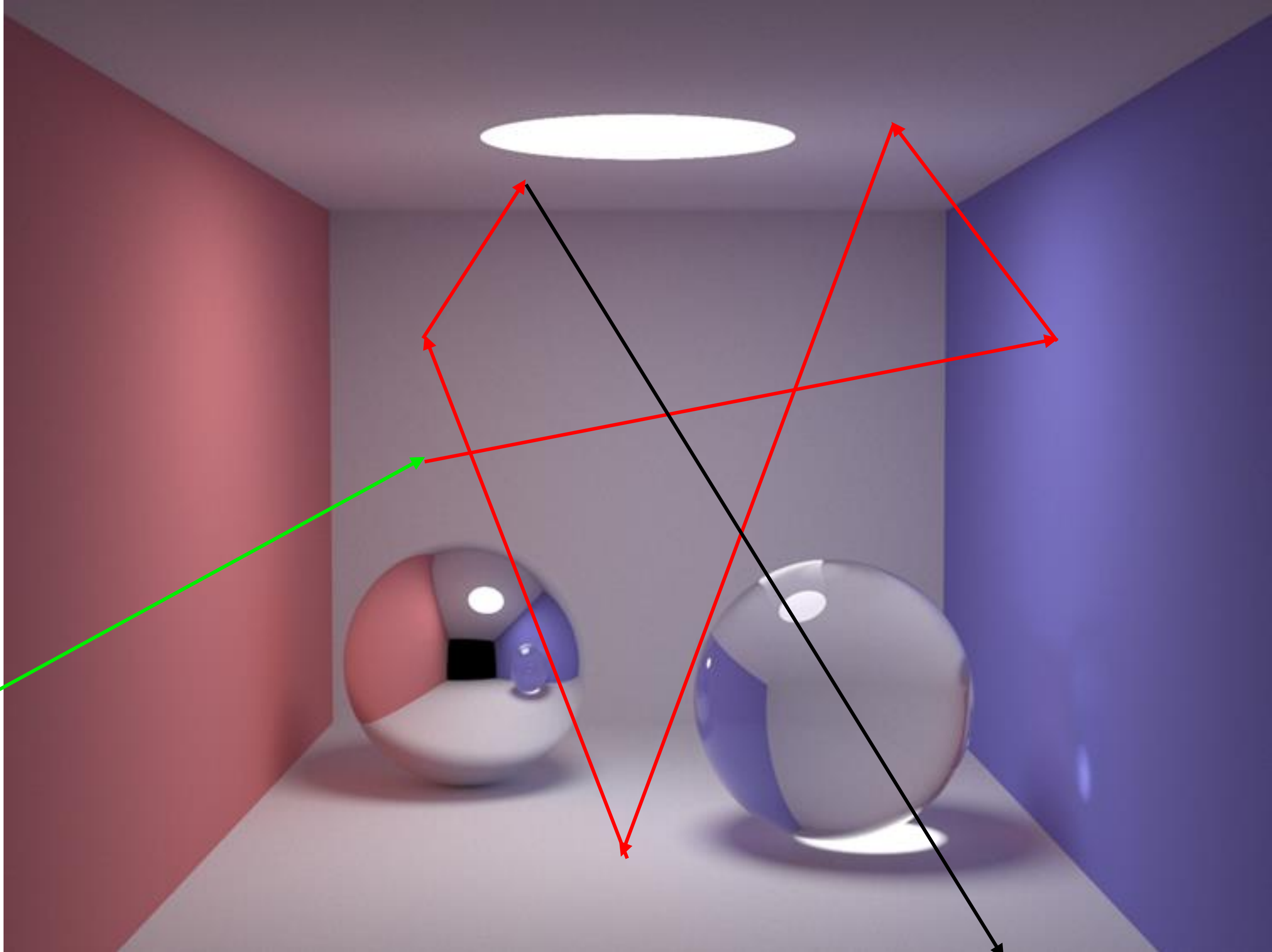
Scenes

- You have been provided with 2 scenes – cornell.txt and sphere.txt
  - These are good for getting started and debugging
  - They make seeing features of Path Tracing easy and recognizable
- However, there are several drawbacks in these scenes:
  - They are generic and not unique
  - They are enclosed – rays do not terminate quickly
  - There is only so much you can do inside a box

Scenes

Cornell Boxes:

+ Lots of bounces so features are easy to spot
- Lots of bounces so takes a long time to resolve
- Either sacrifice iterations or SPP



*Bounces not accurate

Scenes

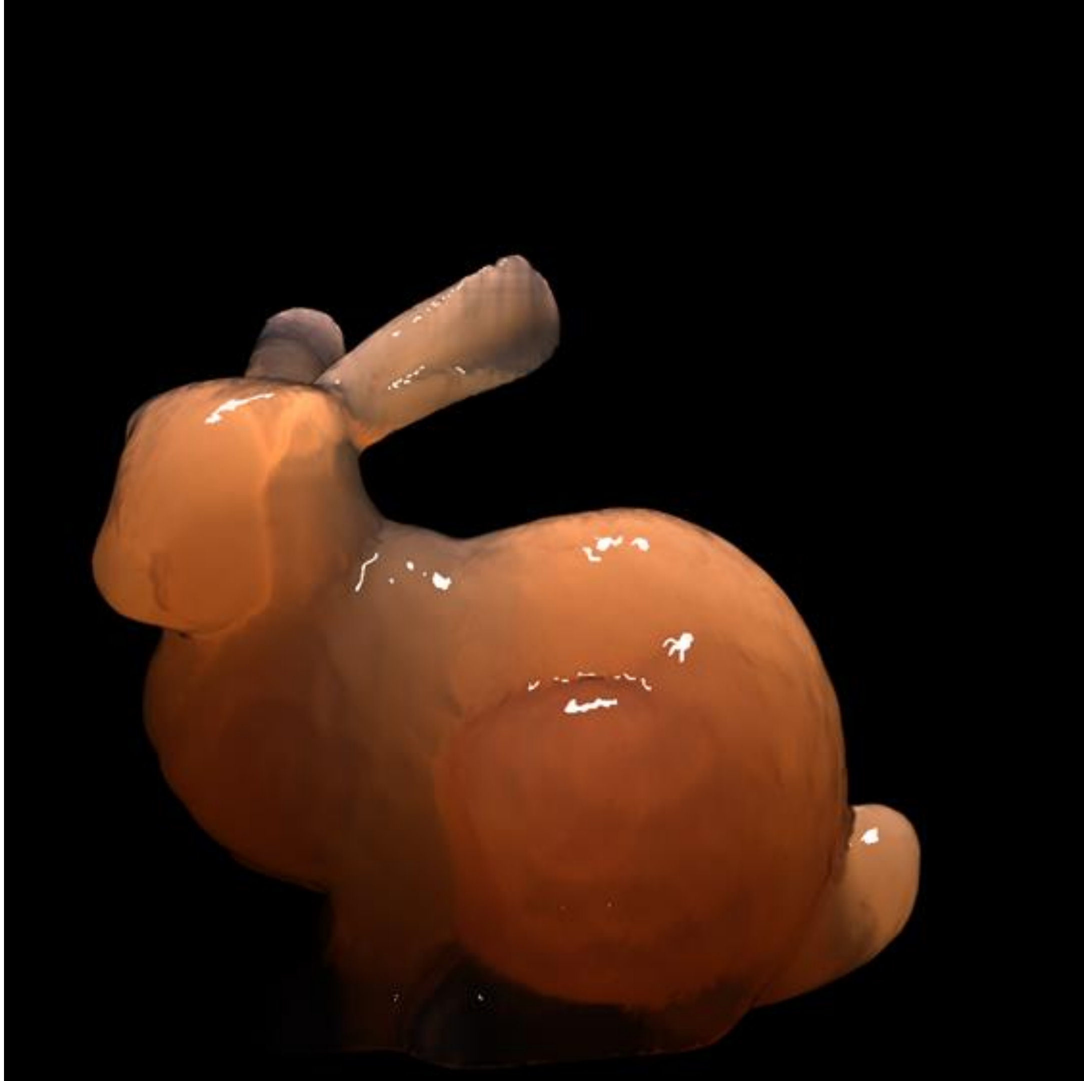
- Use Cornell boxes as debug views or to highlight features
- Create your own scenes from your imagination
  - Loading models is great (but not inside Cornell box)
  - Open up your scenes - Rays will terminate quickly
  - Think creatively
  - Imagine studios or outdoors or real scenes
- Your main image in the readme should not be a Cornell box! Use something more expressive.
- Use scenes that highlight the features even more

Scenes

- **Use of Cornell Box (or variants) as cover images for README is banned**
- You must create your own scene
    - Taking inspiration from scenes of previous years students is acceptable – but don't copy!
- If you significantly modify Cornell Box (see example in slides below), you may request approval on Piazza
- Ok to use for debug and feature images later in the readme

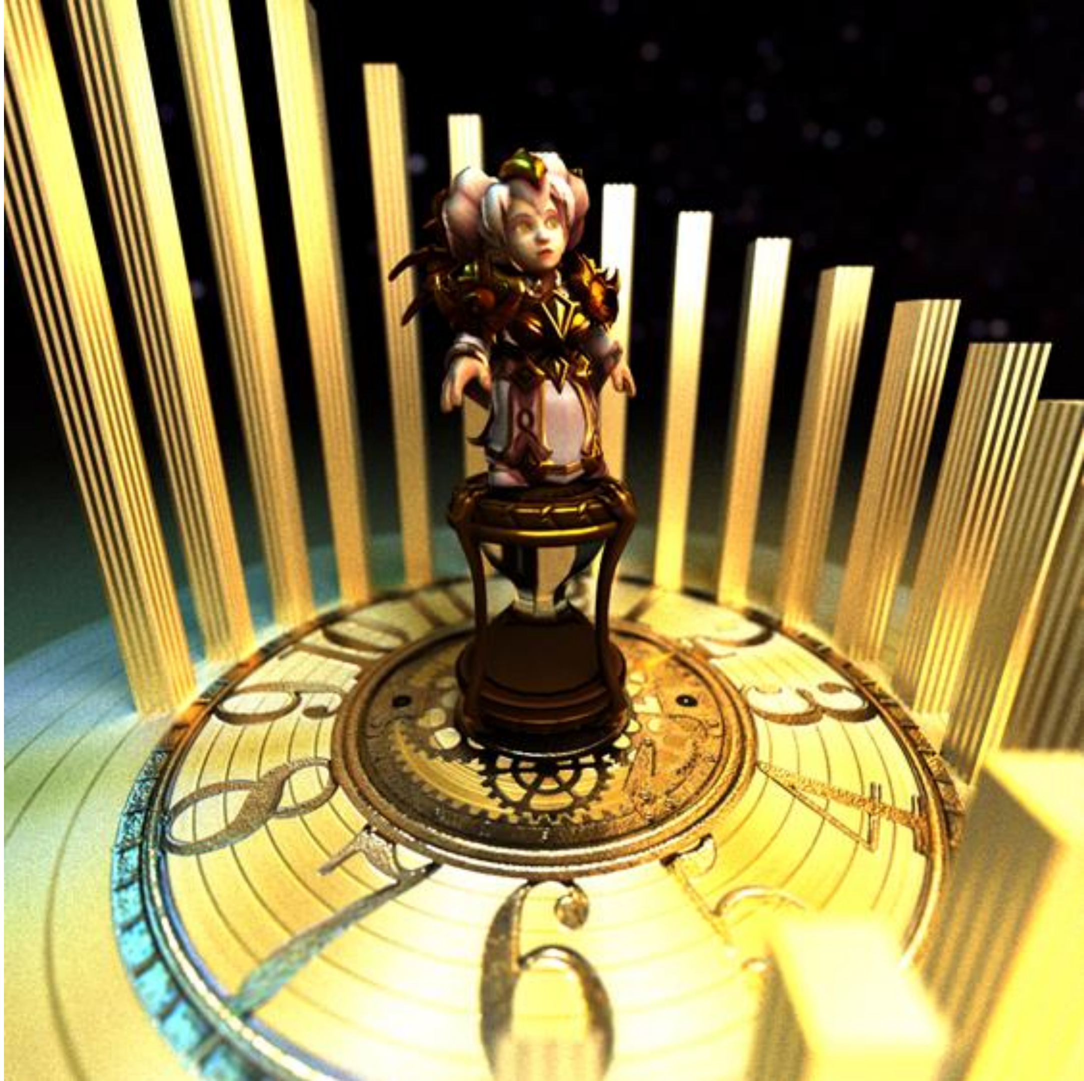
Scenes

https://github.com/loshjawrence/CUDA-Path-Tracer



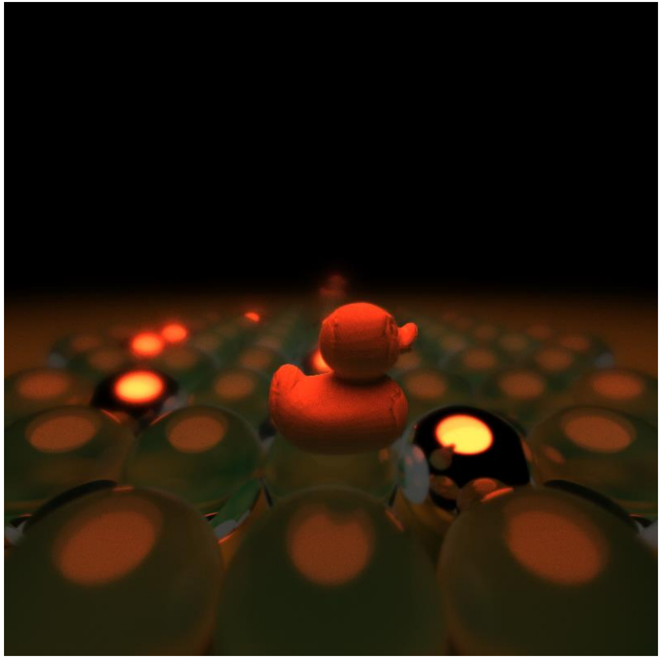
Scenes

https://github.com/byumjin/Project3-CUDA-Path-Tracer



Scenes

[Andrewzhuyx/CUDA-Path-Tracer: Path tracing renderer with support for reflection/refraction/diffusion/depth of field (github.com)]

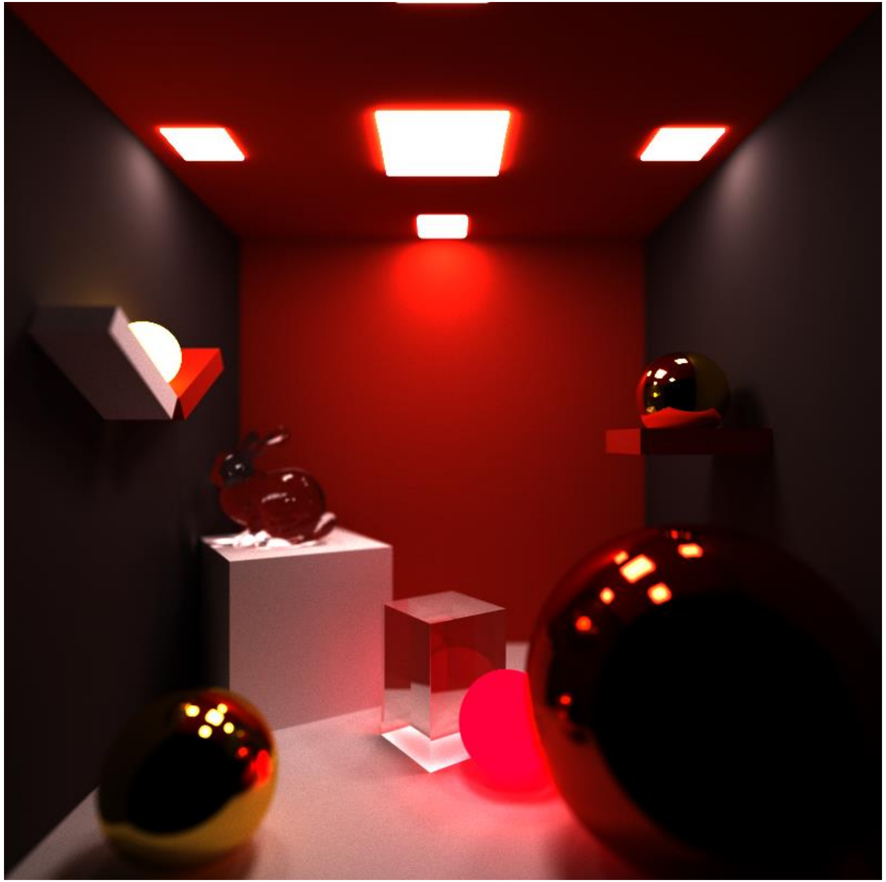


Scenes

[JiyuHuang/Project3-CUDA-Path-Tracer (github.com)]

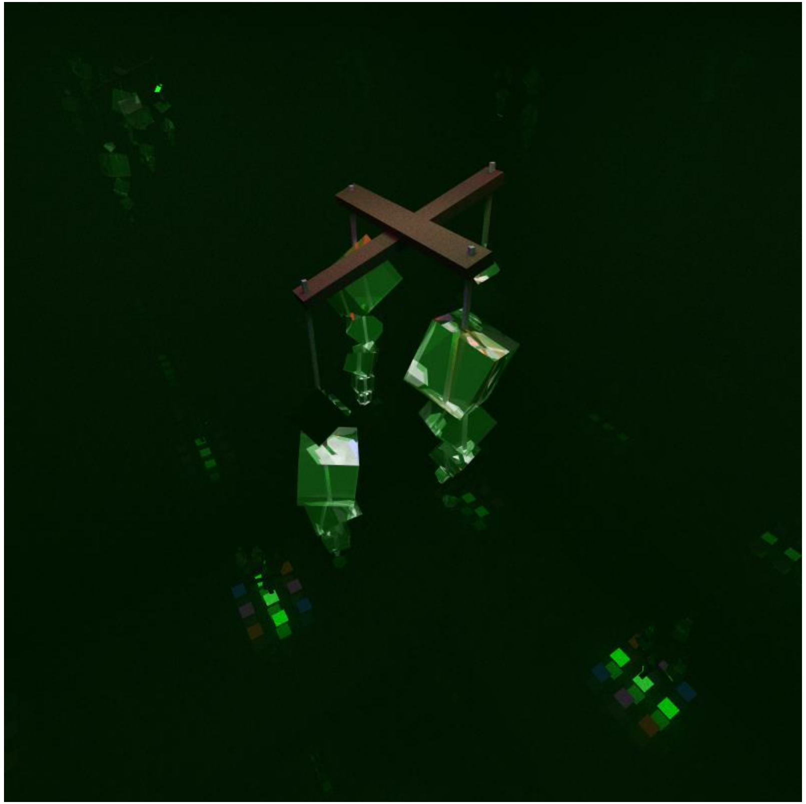
Scenes

[Scoutydren/CUDA-Path-Tracer at submission (github.com)](https://github.com)



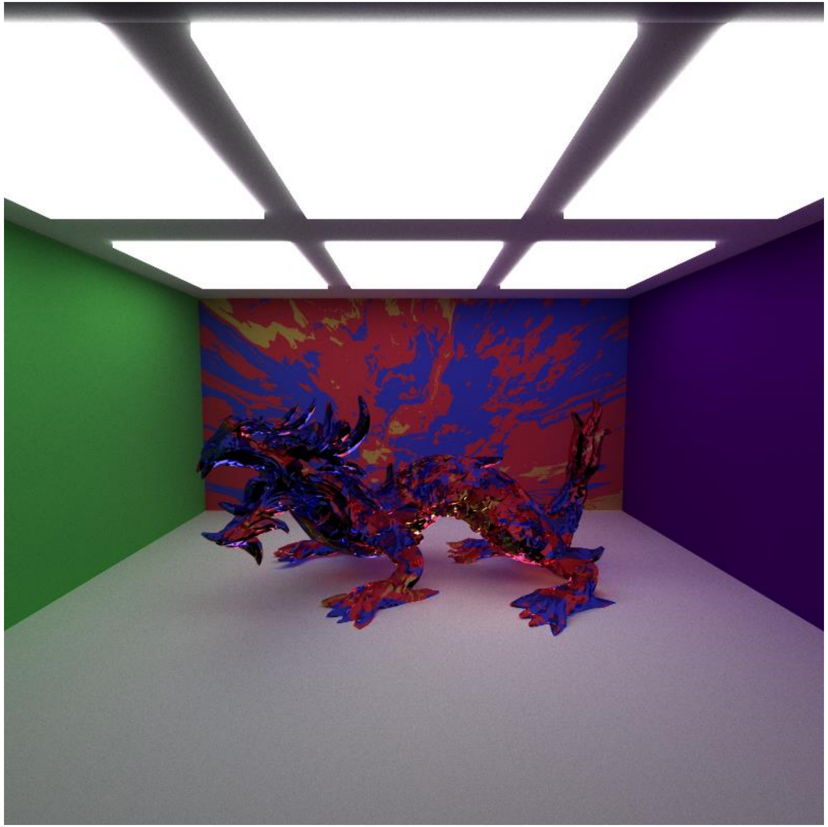
Scenes

[UserRYang/Project3-CUDA-Path-Tracer (github.com)](https://github.com/UserRYang/Project3-CUDA-Path-Tracer)



Scenes

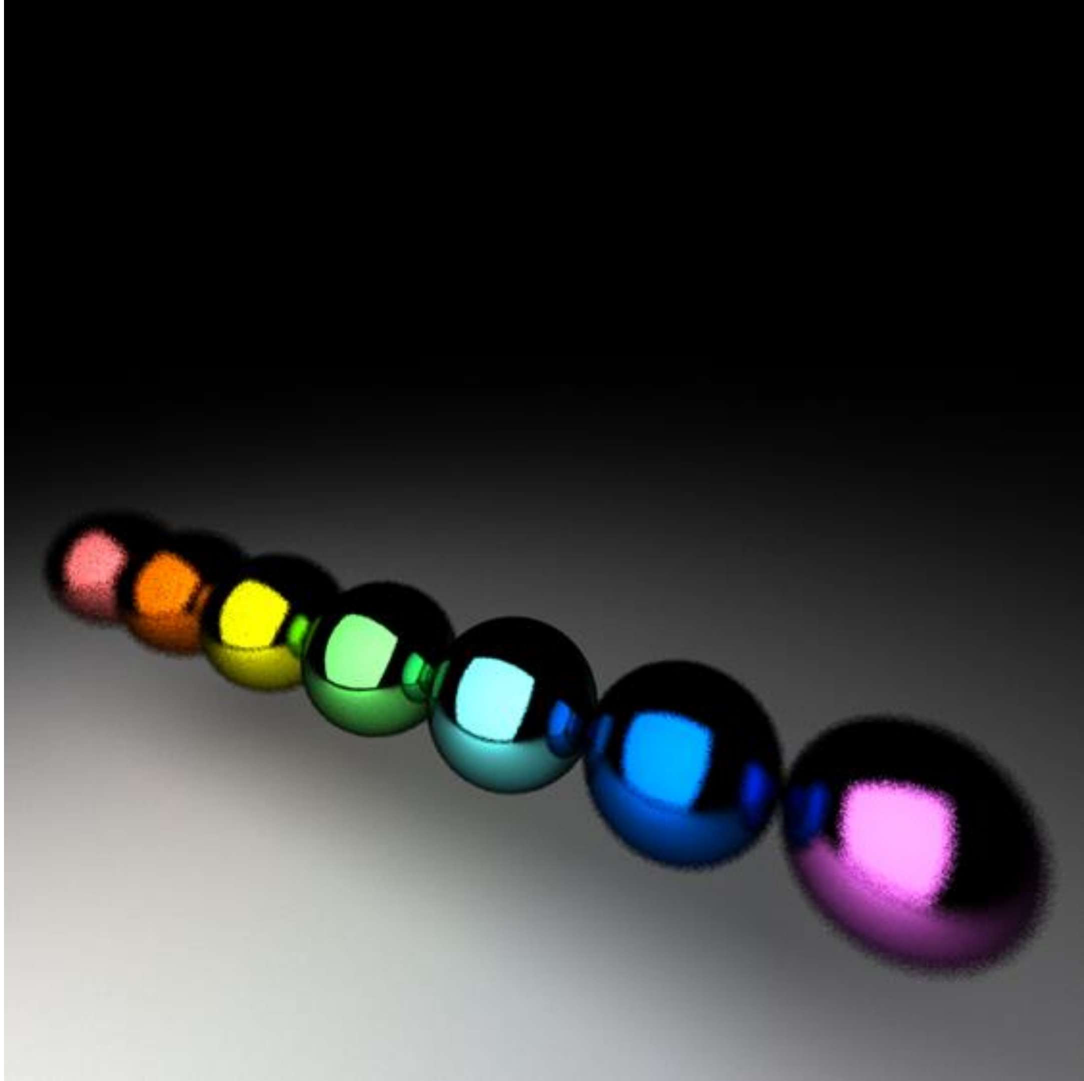
[codeplay9800/Project3-CUDA-Path-Tracer (github.com)]



Scenes

- Use scenes that show off features
- Which features do you see here?

https://github.com/byumjin/Project3-CUDA-Path-Tracer



Scenes

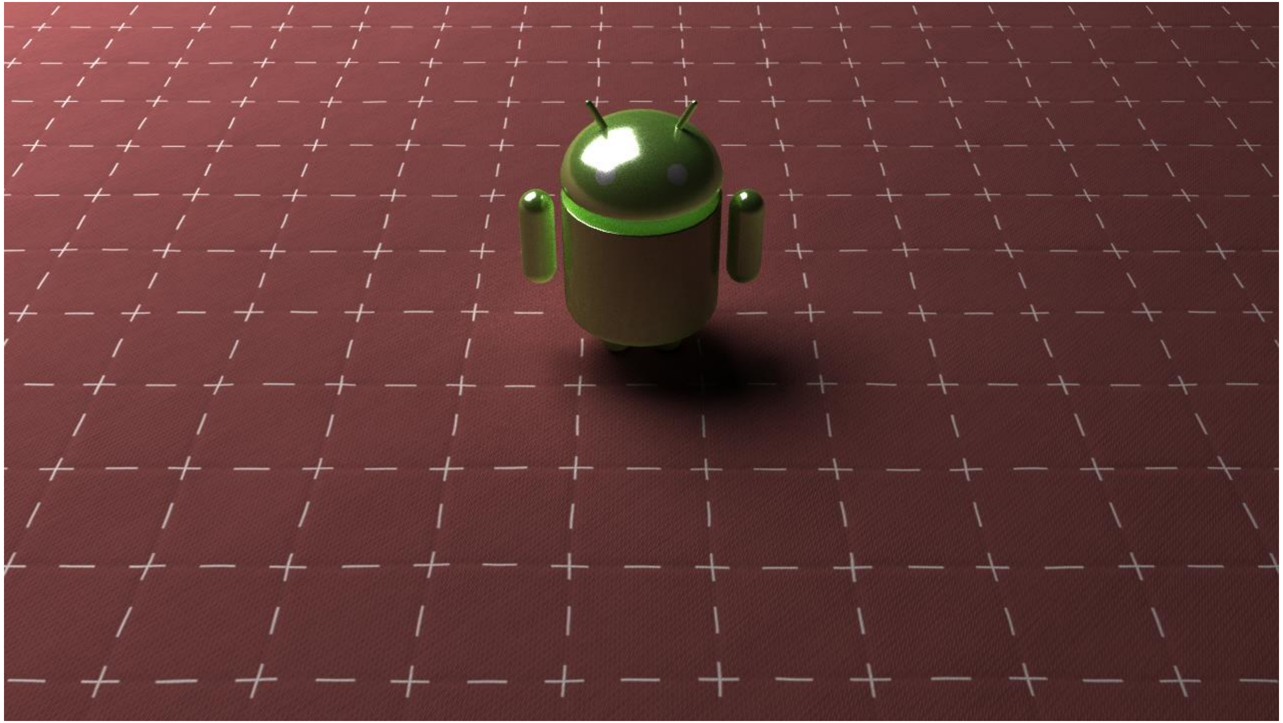
https://github.com/mmerchante/CUDA-Path-tracer

Scenes

https://github.com/emily-vo/cuda-pathtrace

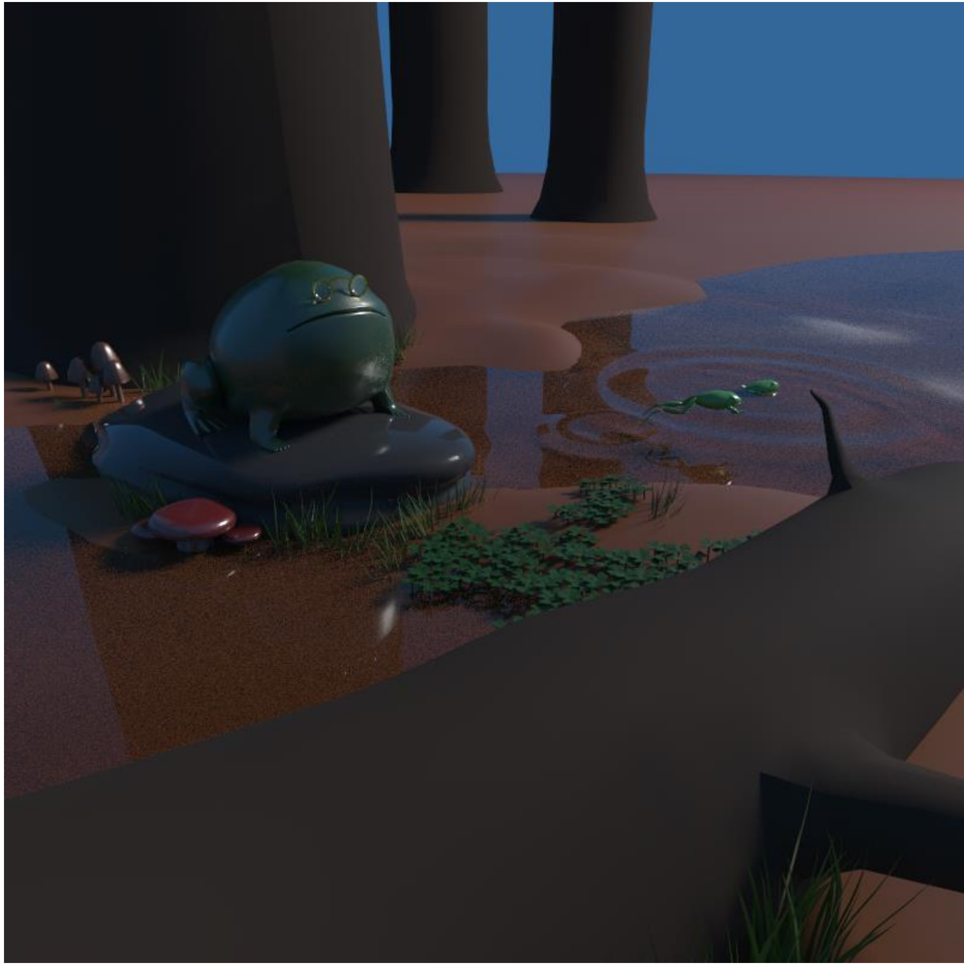
Scenes

https://github.com/vasumahesh1/Project3-CUDA-Path-Tracer



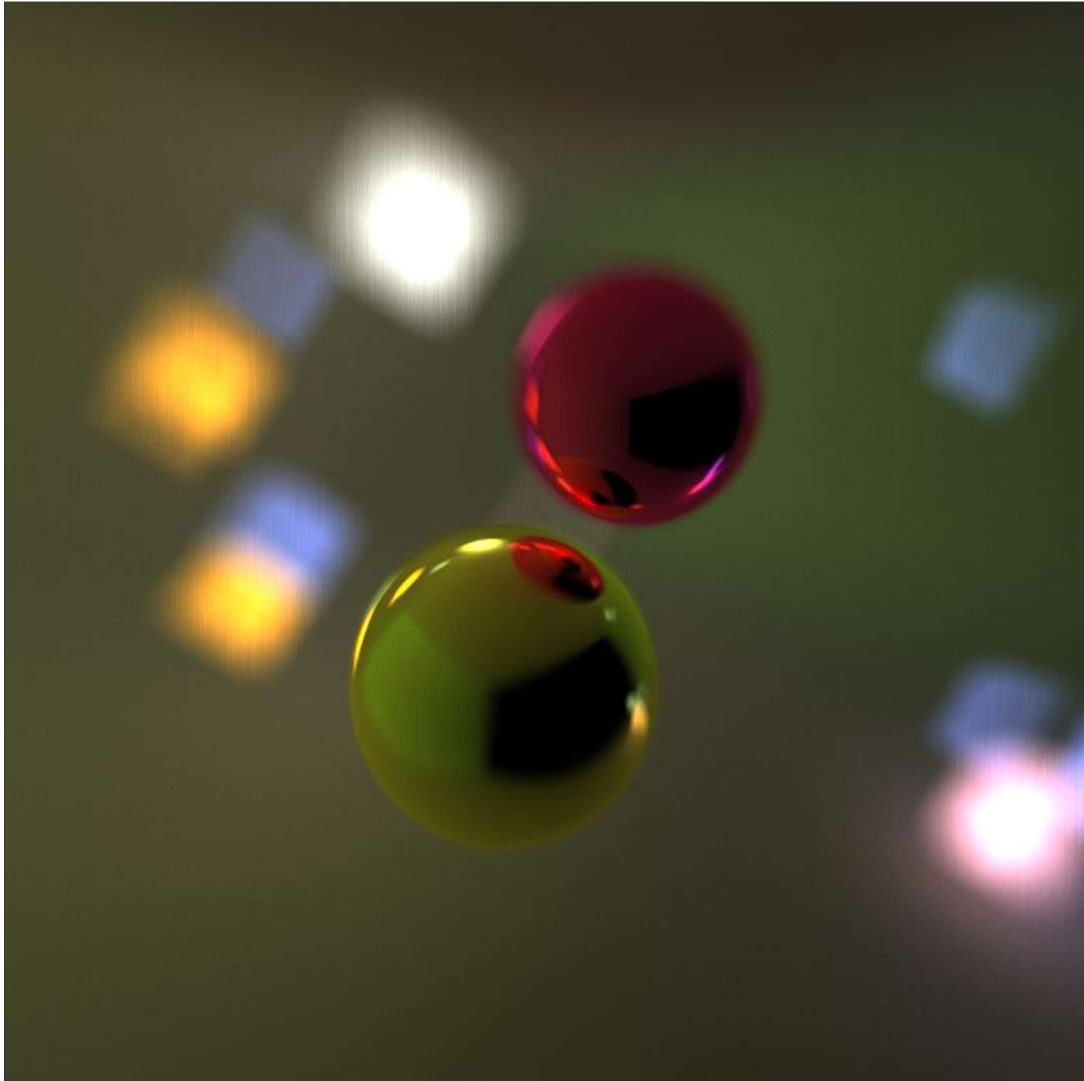
Scenes

https://github.com/lukedan/Project3-CUDA-Path-Tracer

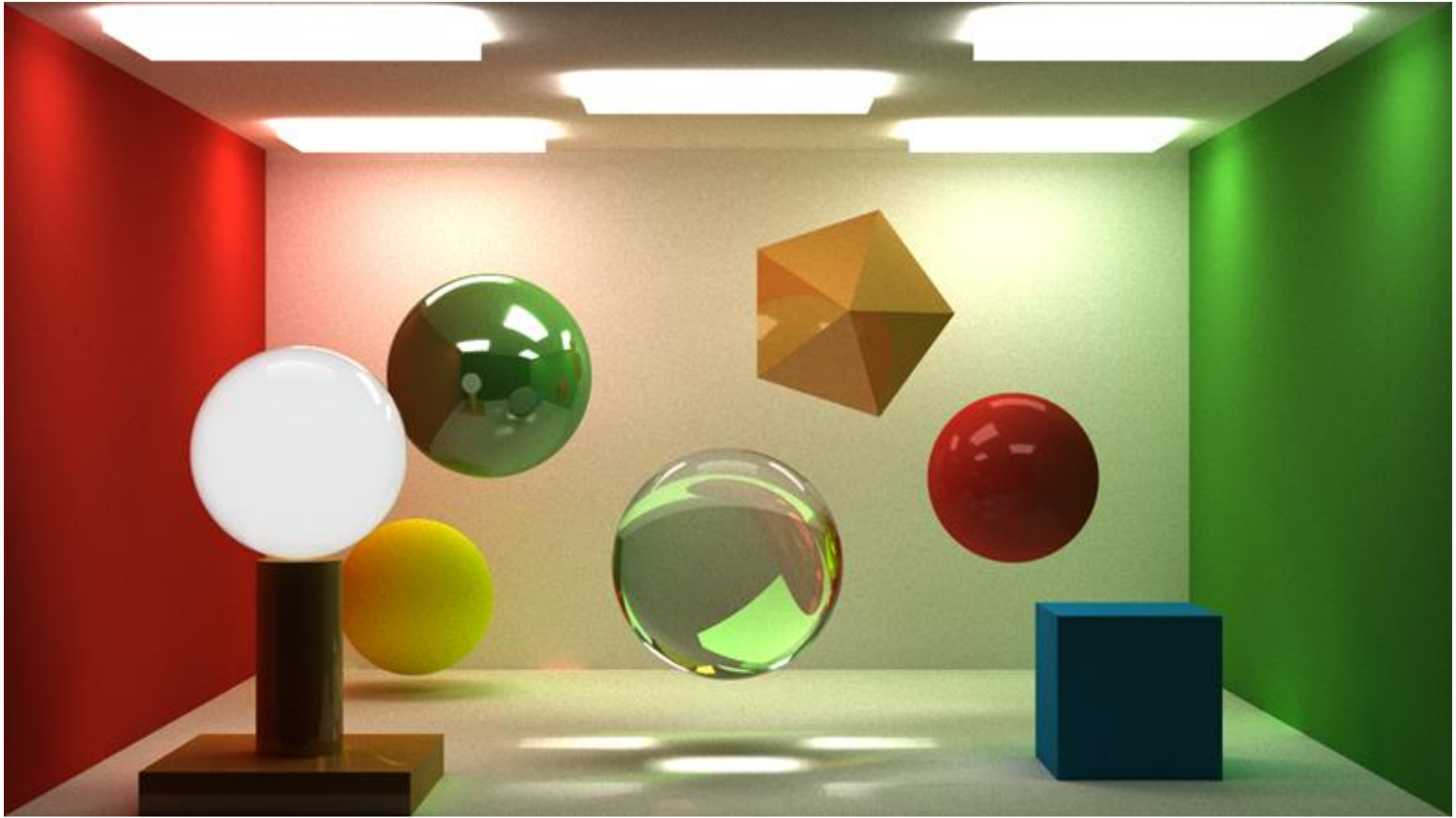


Scenes

https://github.com/Sireesha-Upenn/Project3-CUDA-Path-Tracer

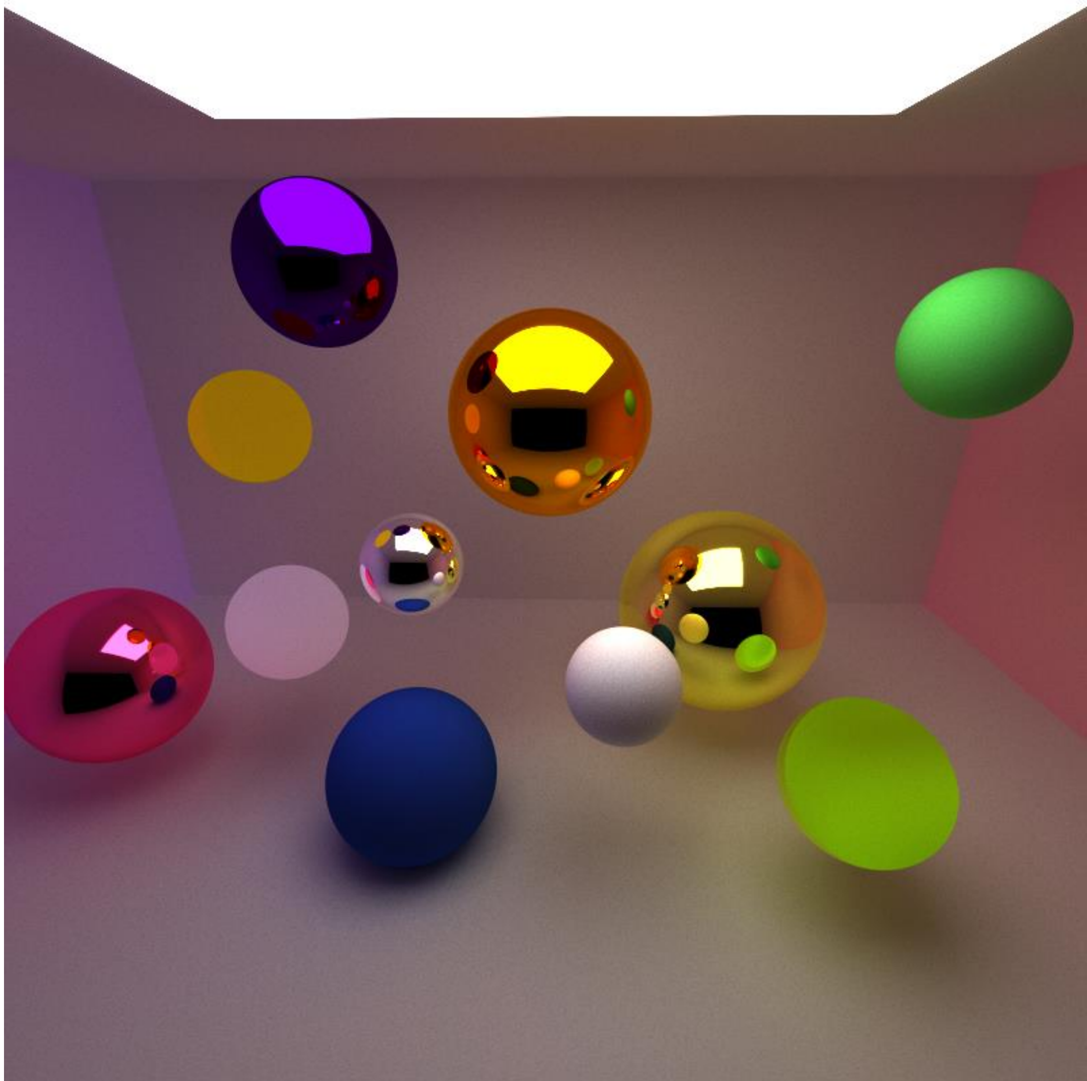


Even when using Cornell Box, Expand It!



Scenes

https://github.com/Sireesha-Upenn/Project3-CUDA-Path-Tracer



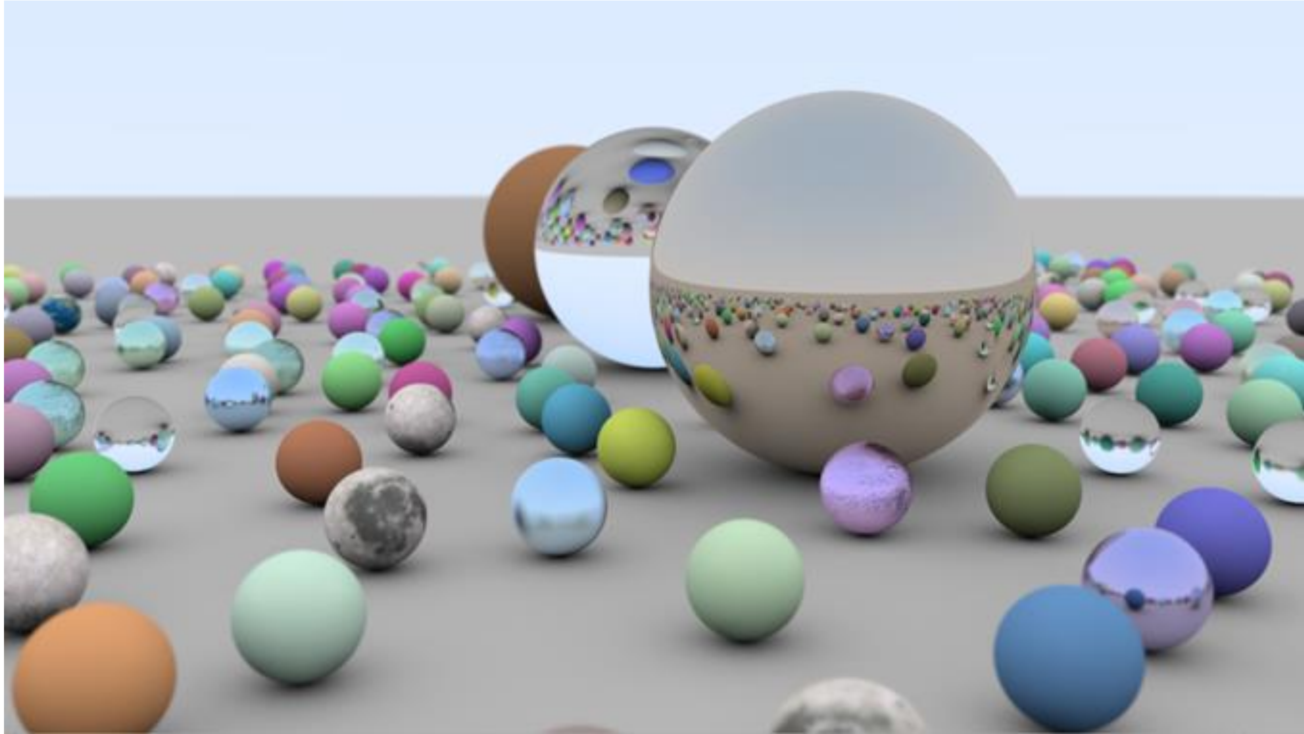
Scenes

http://thume.ca/ray-tracer-site/

(Not 565 student)

Scenes

- https://devblogs.nvidia.com/my-first-ray-tracing-demo/
- Eric Haines



Scenes

- Want even more scenes and models?

- https://casual-effects.com/data/

- https://github.com/KhronosGroup/glTF-Sample-Models/tree/master/2.0
  - (don’t worry about advance features like animations, skinning etc)

- https://sketchfab.com/ (exports gltf)