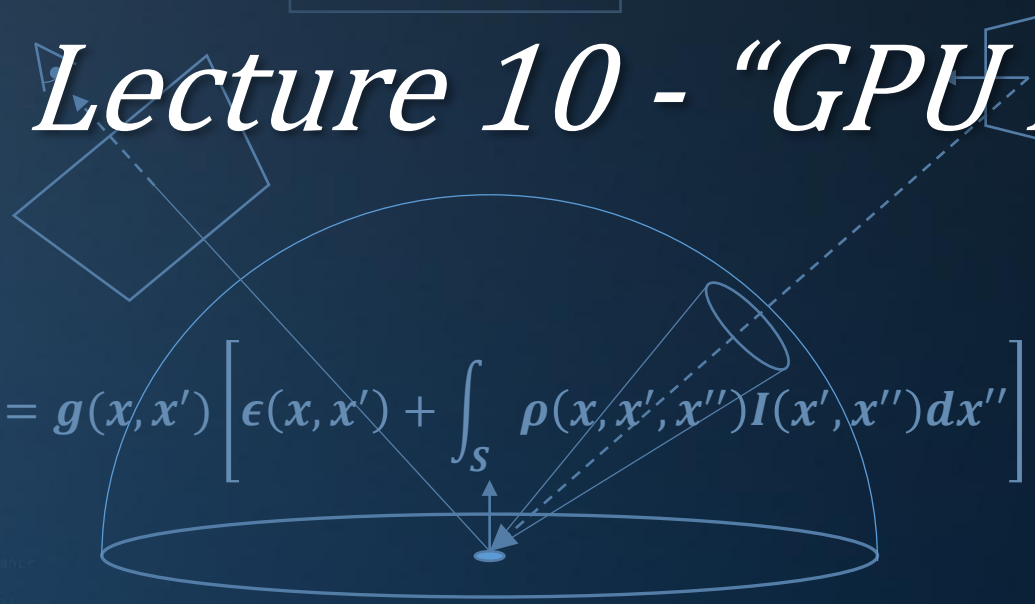
TL;DR version

INFOMAGR – Advanced Graphics

Jacco Bikker - November 2022 - February 2023

Lecture 10 - "GPU Ray Tracing (2)"

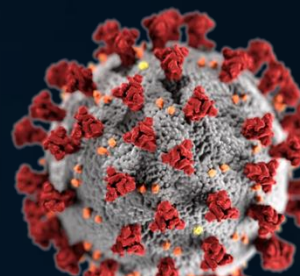


$$I(x,x') = g(x,x')\left[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right]$$

Welcome!

Today's Agenda:

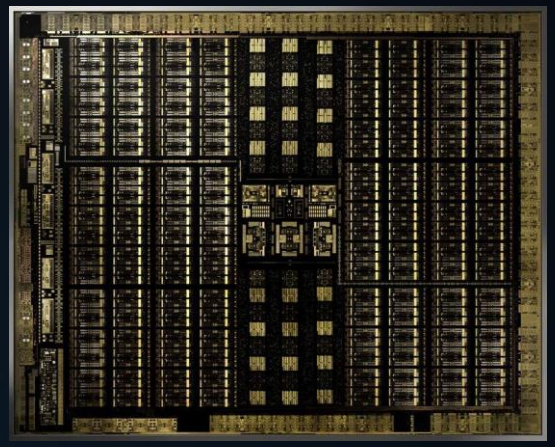
- State of the Art **TL;DR version**
- Wavefront Path Tracing
- Assignment 3 Kick-Off
- What's Next



STAR

Previously in Advanced Graphics

GPU Architecture



https://www.shadertoy.com/view/wdcBW2

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = (fragCoord-.5*iResolution.xy)/iResolution.y;
    uv.y += .355;
    vec2 mouse = iMouse.xy/iResolution.xy;
    uv *= .29;
    vec3 col = vec3(0);
    uv.x = abs(uv.x);
    uv.y += tan(((5./6.)*3.1415))*.68;
    vec2 n = N((5./6.)*3.1415);
    float d = dot(uv-vec2(.5, 0), n);
    uv -= n*max(0., d)*2.;
    n = N((2./3.)*3.1415);
    float scale = 1.;
    uv.x += .5;
    for(int i=0; i < 1; i++) {
        uv *= 3.;
        scale *= 3.;
        uv.x -= 1.5;
        uv.x = abs(uv.x);
        uv.x -= 2.1;
        uv -= n*min(0., dot(uv, n))*1.;
```

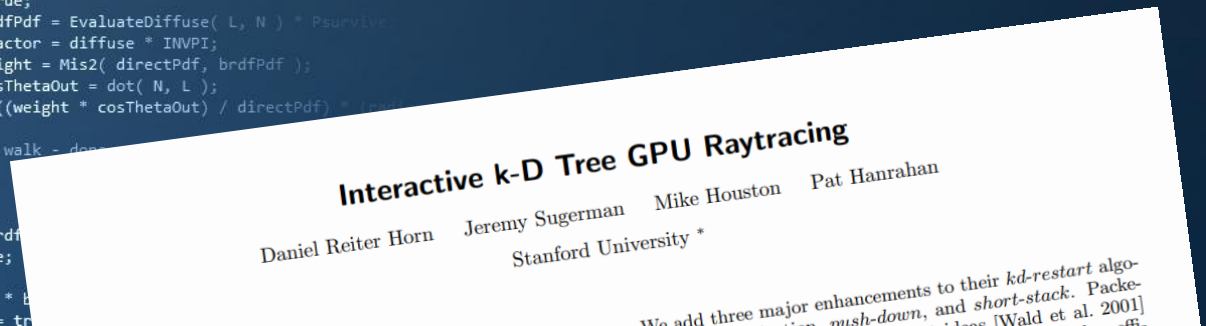
STAR

Previously in Advanced Graphics

A Brief History of GPU Ray Tracing

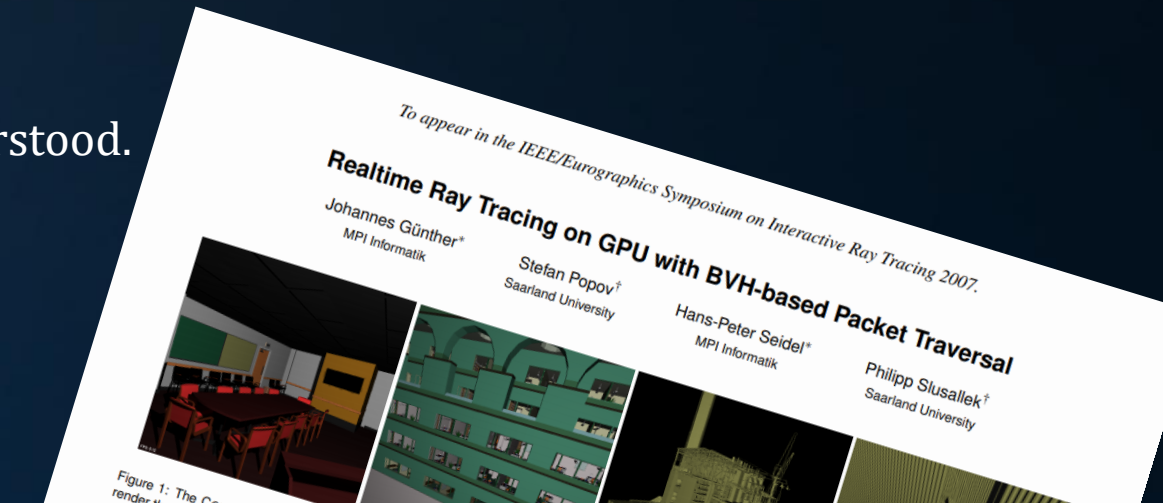
2002: Purcell et al., multi-pass shaders with stencil, grid, low efficiency
2005: Foley & Sugerman, kD-tree, stack-less traversal with kdrestart
2007: Horn et al., kD-tree with short stack, single pass with flow control
2007: Popov et al., kD-tree with ropes
2007: Günther et al., BVH with packets.

- The use of BVHs allowed for complex scenes on the GPU (millions of triangles);
- CPU is now outperformed by the GPU;
- GPU compute potential is not realized;
- Aspects that affect efficiency are poorly understood.


Interactive k-D Tree GPU Raytracing

Daniel Reiter Horn Jeremy Sugerman Mike Houston Pat Hanrahan

Stanford University *


To appear in the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007.

Realtime Ray Tracing on GPU with BVH-based Packet Traversal

Johannes Günther* MPI Informatik

Stefan Popov† Saarland University

Hans-Peter Seidel* MPI Informatik

Philipp Slusallek† Saarland University

STAR

Understanding the Efficiency of Ray Traversal on GPUs*

Observations on BVH traversal:

Ray/scene intersection consists of an unpredictable sequence of node traversal and primitive intersection operations. This is a major cause of inefficiency on the GPU.

Random access of the scene leads to high bandwidth requirement of ray tracing.

BVH packet traversal as proposed by Gunther et al. should alleviate bandwidth strain and yield near-optimal performance.

Packet traversal doesn't yield near-optimal performance. Why not?

*: Understanding the Efficiency of Ray Tracing on GPUs, Aila & Laine, 2009.
and: Understanding the Efficiency of Ray Tracing on GPUs – Kepler & Fermi addendum, 2012.

STAR

Understanding the Efficiency of Ray Traversal on GPUs*

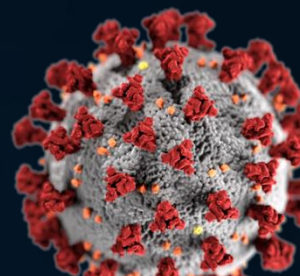
What follows is a wonderful exposition of engineering details, featuring an 'ideal GPU' simulator and a series of simple traversal schemes – which leads to an unlikely culprit. Spoiler: bandwidth is *not* the problem.

Now go an read the paper. ☺

Why not?

Today's Agenda:

- State of the Art **TL;DR version**
- Wavefront Path Tracing
- Assignment 3 Kick-Off
- What's Next



Wavefront

Mapping Path Tracing to the GPU

The path tracing loop from lecture 8 is straight-forward to implement on the GPU.

However:

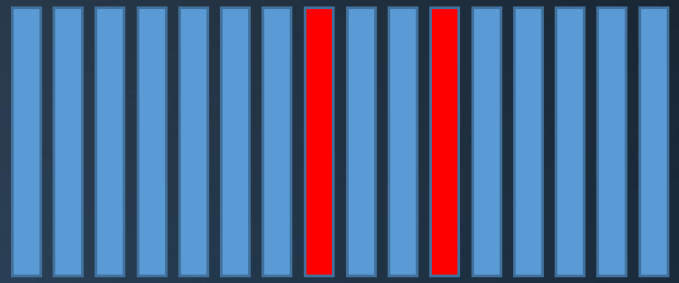
- Terminated paths become idling threads;
- A significant number of paths will not trace a shadow ray.

```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront

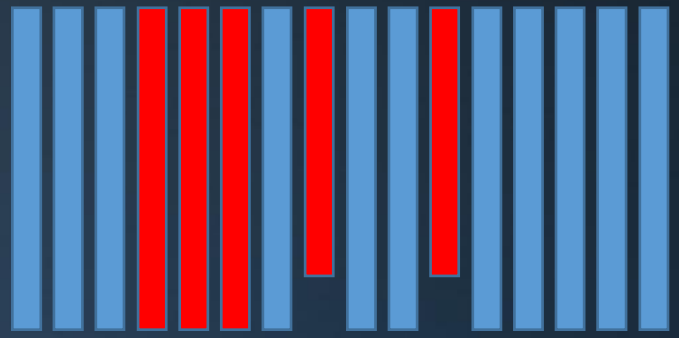
```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront



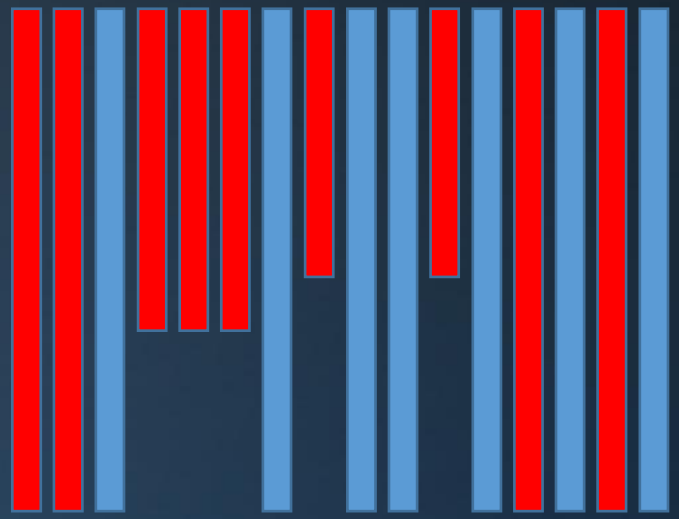
```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront



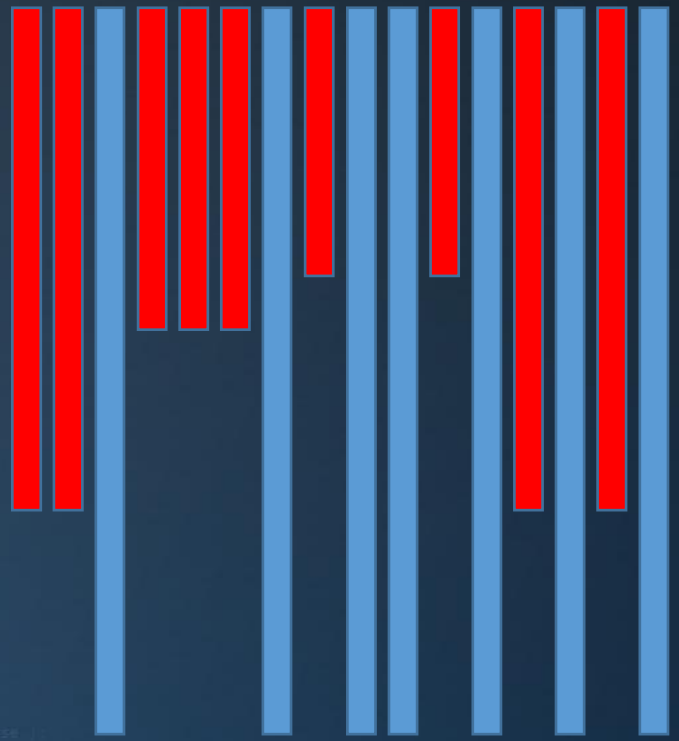
```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront



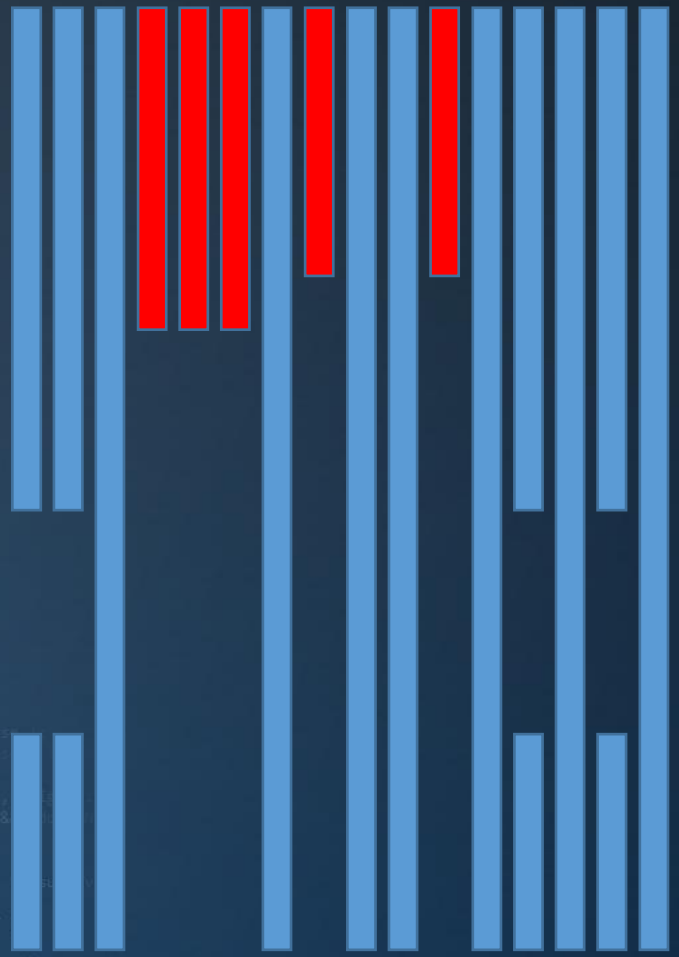
```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront



```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront

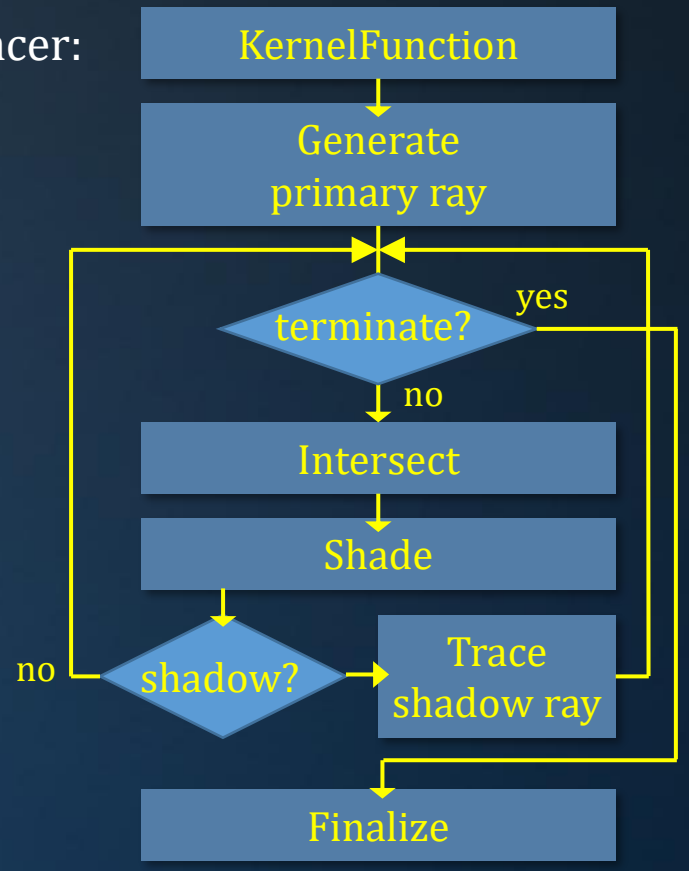


```
Color Sample( Ray ray )
{
   T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
   while (1)
   {
      I, N, material = Trace( ray );
      BRDF = material.albedo / PI;
      if (ray.NOHIT) break;
      if (material.isLight) break;
      // sample a random light source
      L, Nl, dist, A = RandomPointOnLight();
      Ray lr( I, L, dist );
      if (N·L > 0 && Nl·-L > 0) if (!Trace( lr ))
      {
         solidAngle = ((Nl·-L) * A) / dist²;
         lightPDF = 1 / solidAngle;
         E += T * (N·L / lightPDF) * BRDF * lightColor;
      }
      // continue random walk
      R = DiffuseReflection( N );
      hemiPDF = 1 / (PI * 2.0f);
      ray = Ray( I, R );
      T *= ((N·R) / hemiPDF) * BRDF;
   }
   return E;
}
```

Wavefront

Megakernels Considered Harmful*

Naïve path tracer:



Translating this to CUDA or OpenCL code yields a single kernel: individual functions are still compiled to one monolithic chunk of code.

Resource requirements (registers) - and thus parallel slack - are determined by 'weakest link', i.e. the functional block that requires most registers.

Conditional code leads to idling threads that wait until others are done.

*: Megakernels Considered Harmful: Wavefront Path Tracing on GPUs, Laine et al., 2013

Wavefront

Megakernels Considered Harmful

Solution: *split the kernel*.

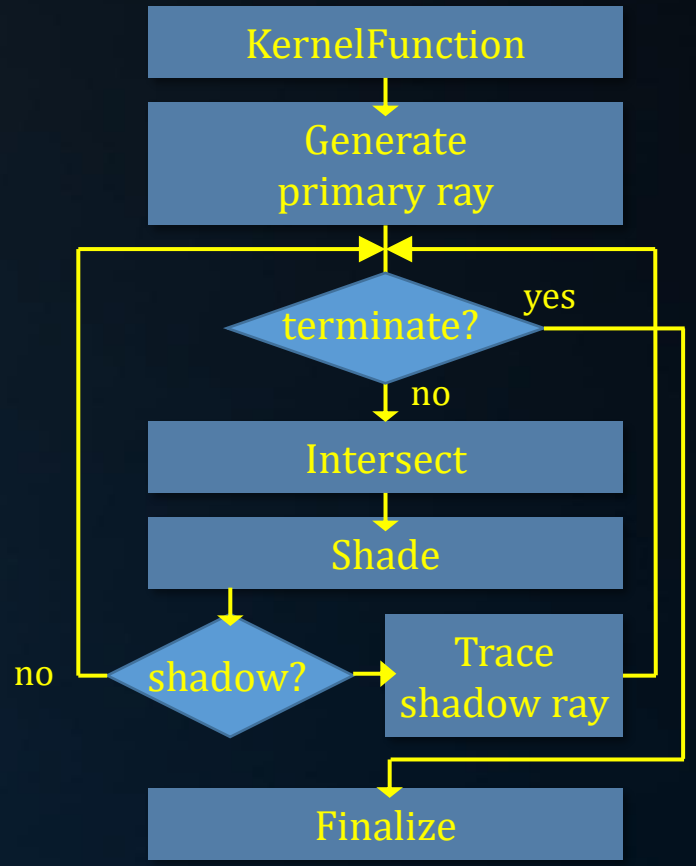
Example:

Kernel 1: Generate primary rays.
Kernel 2: Trace paths.
Kernel 3: Accumulate, gamma correct, convert to ARGB32.

Consequence:

Kernel 1 generates *all* primary rays, and stores the result.
Kernel 2 takes this buffer and operates on it.

➔ *Massive memory I/O.*



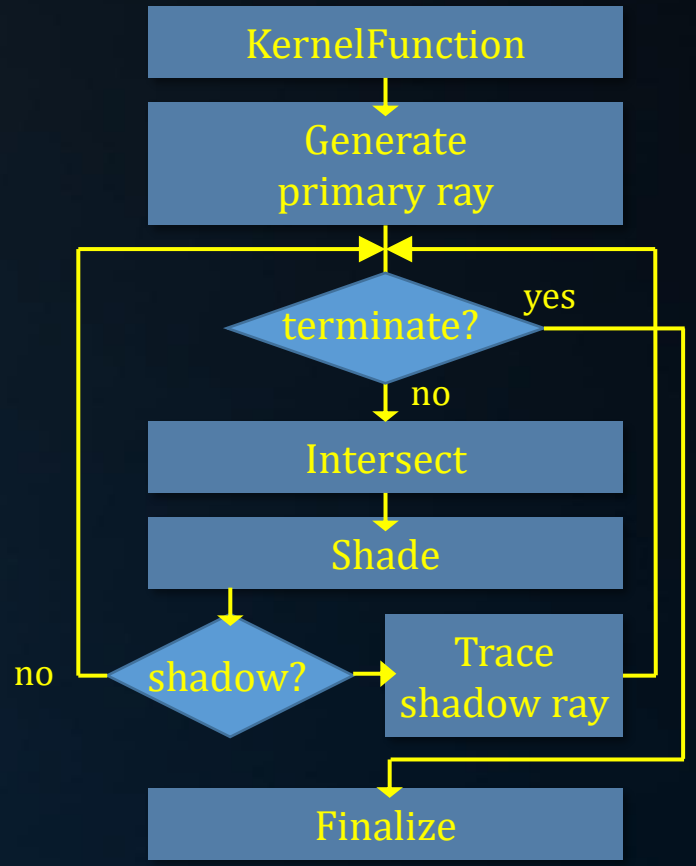
Wavefront

Megakernels Considered Harmful

Taking this further: streaming path tracing*.

Kernel 1: generate primary rays.
Kernel 2: extend.
Kernel 3: shade.
Kernel 4: connect.
Kernel 5: finalize.

Here, kernel 2 traces a set of rays to find the next path vertex (the random walk).
Kernel 3 processes the results and generates new path segments and shadow rays (2 separate buffers).
Kernel 4 traces the shadow ray buffer.
Kernel 1, 2, 3 and 4 are executed in a loop until no rays remain.



*: Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU, van Antwerpen, 2011

Wavefront

Megakernels Considered Harmful

Zooming in:

The `generate` kernel produces *N* primary rays:

0, 1, …	…, N-1

Buffer 1: path segments (*N* times O,D,t,primIdx)

The `extend` kernel traces extension rays and produces intersections*.
The `shade` kernel processes intersections, and produces new extension paths as well as shadow rays:

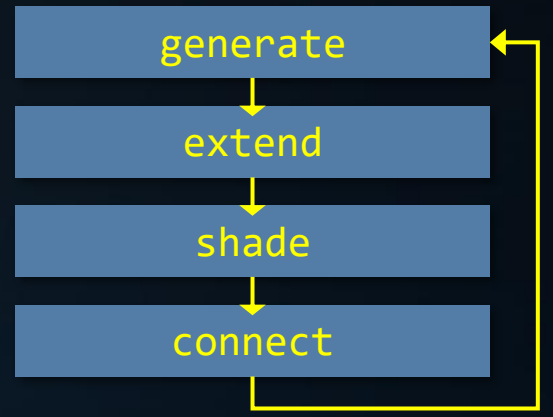
0, 1, …	…, N-1

Buffer 2: generated path segments (*N* times O,D,t,primIdx)

0, 1, …	…, N-1

Buffer 3: generated shadow rays (*N* times O,D,t, E,pixelIdx)

Finally, the `connect` kernel traces shadow rays.



Note: here, the loop is implemented on the host. Each block is a separate kernel invocation.

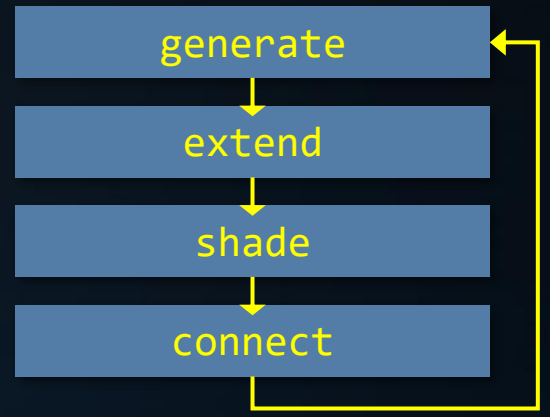
*: An intersection is at least the t value, plus a primitive identifier.

Wavefront

Megakernels Considered Harmful

Generate:

```
for each screen pixel i
{
    O,D = GenerateRayDirection(i)
    rayBuffer[i] = Ray( O, D, infinity, -1 )
}
```

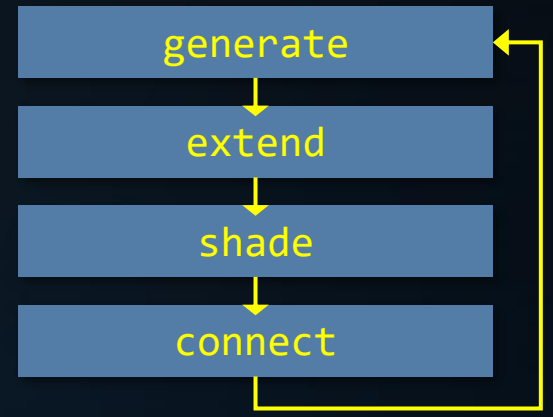


Wavefront

Megakernels Considered Harmful

Extend:

```
for each buffered ray r
{
    O,D,dist = rayBuffer[i]
    dist, primIdx = FindNearestIntersection( O, D, dist )
    rayBuffer[i].dist = dist
    rayBuffer[i].primIdx = primIdx
}
```

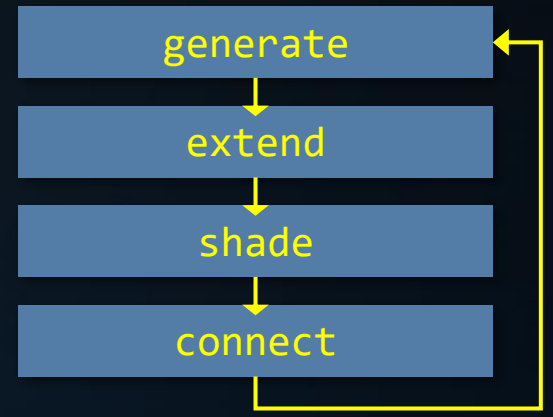


Wavefront

Megakernels Considered Harmful

Shade:

```
for each buffered ray r
{
    O,D,dist,primIdx = rayBuffer[i]
    I = IntersectionPoint( O, D, dist )
    N = PrimNormal( primIdx, I )
    if (NEE) {
        si = atomicInc( shadowRayIdx )
        shadowBuffer[si] = ShadowRay( … )
    }
    if (bounce) {
        ei = atomicInc( extensionRayIdx )
        newRayBuffer[ei] = ExtensionRay( … )
    }
}
```

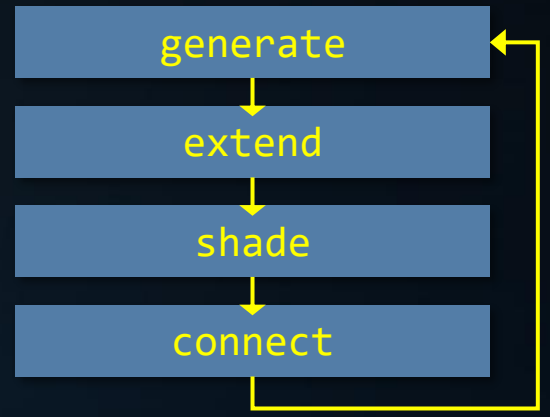


Wavefront

Megakernels Considered Harmful

Connect:

```
for each buffered shadowRay r
{
   O,D,dist,E, pixelIdx = shadowBuffer[i]
   if (!Occluded( O, D, dist ))
   {
      accumulator[pixelIdx] += E;
   }
}
```



Wavefront

Megakernels Considered Harmful

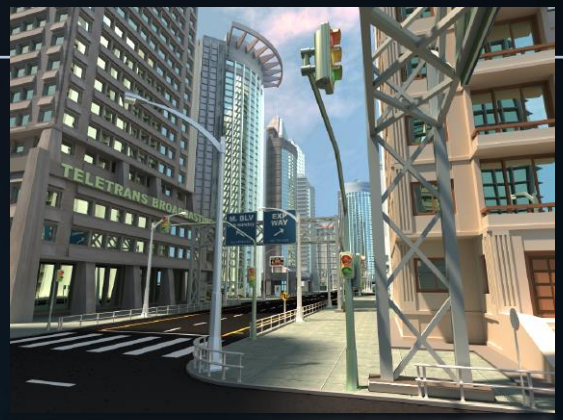
Digest:

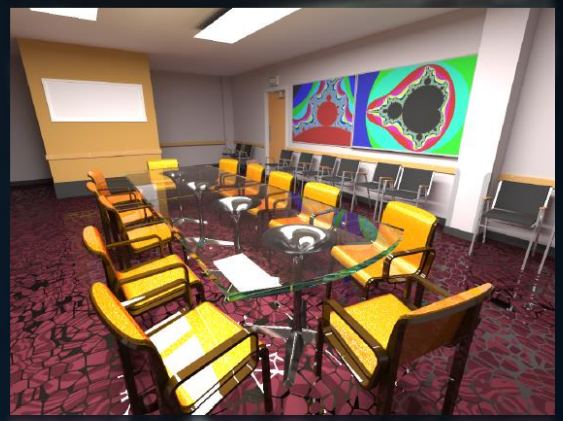
Streaming path tracing introduces seemingly costly operations:

- Repeated I/O to/from large buffers;
- A significant number of kernel invocations per frame;
- Communication with the host.

The Wavefront paper claims that this is beneficial for complex shaders. In practice, this also works for (very) simple shaders.

Also note that the megakernel paper (2013) presents an idea already presented by Dietger van Antwerpen (2011).



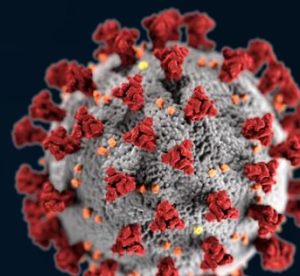


Today's Agenda:

- State of the Art **TL;DR version**
- Wavefront Path Tracing
- Assignment 3 Kick-Off
- What's Next

Today's Agenda:

- State of the Art **TL;DR version**
- Wavefront Path Tracing
- 2023 Update: SER
- Assignment 3 Kick-Off
- What's Next



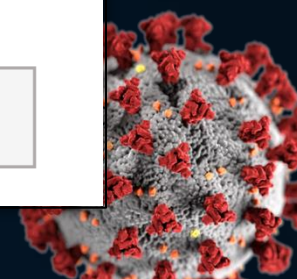
Shader Execution Reordering (SER) is a new scheduling technology introduced with the Ada Lovelace generation of NVIDIA GPUs. It is highly effective at simultaneously reducing both execution divergence and data divergence. SER achieves this by on-the-fly reordering threads across the GPU such that groups of threads perform similar work and therefore use GPU resources more efficiently. This happens with minimal overhead: the Ada hardware architecture was designed with SER in mind and includes optimizations to the SM and memory system specifically targeted at efficient thread reordering. Using SER, we observe speedups of up to 2x in raytracing regimes of real-world applications, achieved with only a small amount of developer effort. To applications, SER is exposed through a small API that gives developers new flexibility and full control over where in their shaders reordering will happen. This API is detailed in the following sections.

API overview

ReorderThread

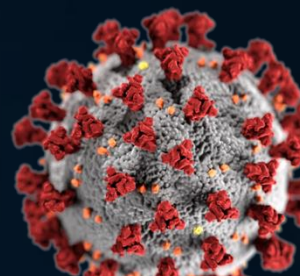
The main functionality of SER is encapsulated in a single function:

```
void ReorderThread( key )
```



Today's Agenda:

- State of the Art **TL;DR version**
- Wavefront Path Tracing
- 2023 Update: SER
- Assignment 3 Kick-Off
- What's Next

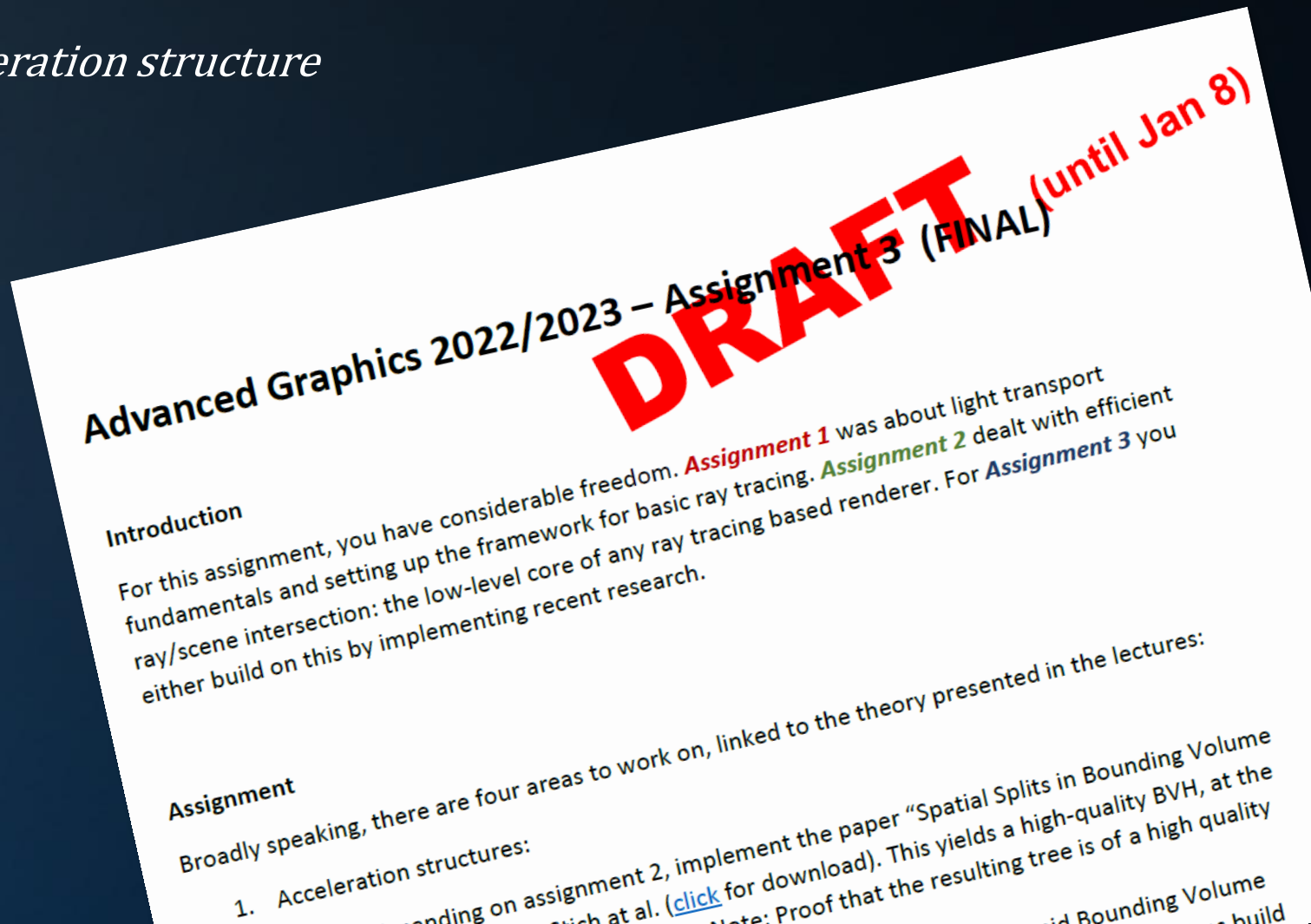


Assignment 3

Assignment 1: *Build a framework*

Assignment 2: *Add an acceleration structure*

Assignment 3: Freestyle.



Advanced Graphics 2022/2023 – Assignment 3 (FINAL) DRAFT (until Jan 8)

Introduction

For this assignment, you have considerable freedom. **Assignment 1** was about light transport fundamentals and setting up the framework for basic ray tracing. **Assignment 2** dealt with efficient ray/scene intersection: the low-level core of any ray tracing based renderer. For **Assignment 3** you either build on this by implementing recent research.

Assignment

Broadly speaking, there are four areas to work on, linked to the theory presented in the lectures:

1. Acceleration structures:

Assignment 3

Freestyle

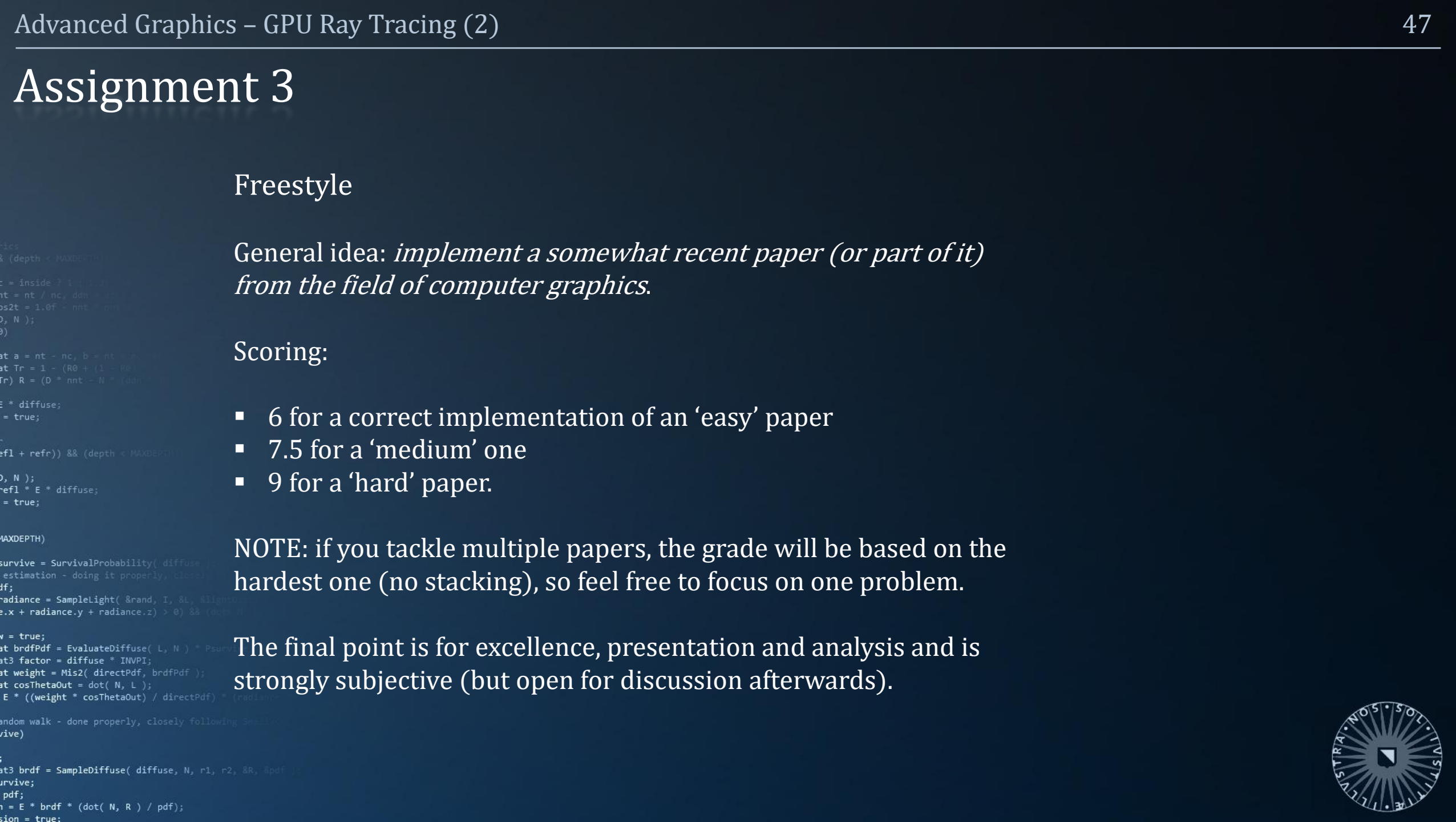
General idea: *implement a somewhat recent paper (or part of it) from the field of computer graphics*.

Scoring:

- 6 for a correct implementation of an ‘easy’ paper
- 7.5 for a ‘medium’ one
- 9 for a ‘hard’ paper.

NOTE: if you tackle multiple papers, the grade will be based on the hardest one (no stacking), so feel free to focus on one problem.

The final point is for excellence, presentation and analysis and is strongly subjective (but open for discussion afterwards).



Assignment 3

Freestyle

Topics – part 1 – Acceleration Structures

- Expanding on assignment 2, implement the paper “Spatial Splits in Bounding Volume Hierarchies” by Stich at al. This yields a high-quality BVH, at the expense of construction time. Note: Proof that the resulting tree is of a high quality is required. Difficulty: medium/hard.
- Or, go the opposite route: implement the paper “Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees” by Ganestam et al., which maximizes build performance. Note: build performance must be roughly in line with the numbers in the paper. Difficulty: medium/hard.
- Starting from a TLAS/BLAS builder from the tutorial series, add ‘partial rebraiding’, based on the paper “Improved two-level BVHs using partial re-braiding” by Benthin et al. Prove that the resulting TLAS/BLAS performs in line with the findings in the paper. Difficulty: easy.
- Other options exist. There is a paper by NVIDIA that explains how to efficiently traverse a (CPU-built) 8-wide BVH on the GPU for state-of-the-art #RTXoff performance (hard). There exist other heuristics than SAH; an implementation and an in-depth analysis may provide you with an interesting project (easy).
- Feel free to propose something interesting!

Assignment 3

Freestyle

Topics – part 2 – Physically-based Rendering

- Implement a basic but correct bidirectional path tracer, as described by Veach in his Ph.D. thesis (as well as in many other sources). Proof the correctness of your implementation by comparing energy levels. Warning: hard.
- Implement photon mapping, as described by Henrik Wann Jensen (“Global Illumination using Photon Maps”, 1996). Your implementation should correctly handle caustics. Verify your result against ground truth produced by a path tracer. Warning: must produce correct energy levels. Difficulty: easy – medium.
- Implement a basic but correct volumetric path tracer for non-homogeneous media, e.g. a cloud. Provide a test scene that clearly demonstrates the functionality. Difficulty: medium.
- Other options: feel free to propose something interesting.

Assignment 3

Freestyle

Topics – part 3 – Filtering & Reprojection

- Implement a filter that combines reprojection and spatial filtering to improve image quality. Suggestion: use a simple scene to make reprojection worthwhile. Difficulty: easy.
- Implement Adaptive Sampling (see e.g. “A Survey of Adaptive Sampling in Realistic Image Synthesis”, M. Šik. Difficulty: easy.
- Add the “Open Path Guiding Library” to your existing renderer and report on the integration process. Difficulty: easy, I think.
- Implement a recent paper on the topic of filtering (see 4).

Assignment 3

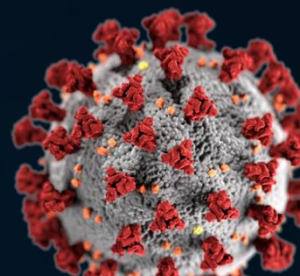
Freestyle

Topics – part 4 – Other Options

- "Path Guiding in Production", by Vorba et al., 2019. Path guiding is a class of 'learning algorithms', which estimate importance of directions over the hemisphere based on earlier transport results. Difficulty: medium.
- "Direct Ray Tracing of Smoothed and Displacement Mapped Triangles", Smits et al., 2000. The paper describes a method for directly rendering displacement maps in a ray tracer. The method is computationally expensive, which made it impractical in 2000. Now, in 2020, things have changed. Difficulty: I think hard. (2023 update: yes, hard).
- "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting", Bitterli et al., 2020. As discussed in the slides: a method for importance sampling thousands of lights. Difficulty: medium, but potentially a lot of work.
- Implement Wavefront Path Tracing on the GPU. Difficulty: medium, but rewarding.

Today's Agenda:

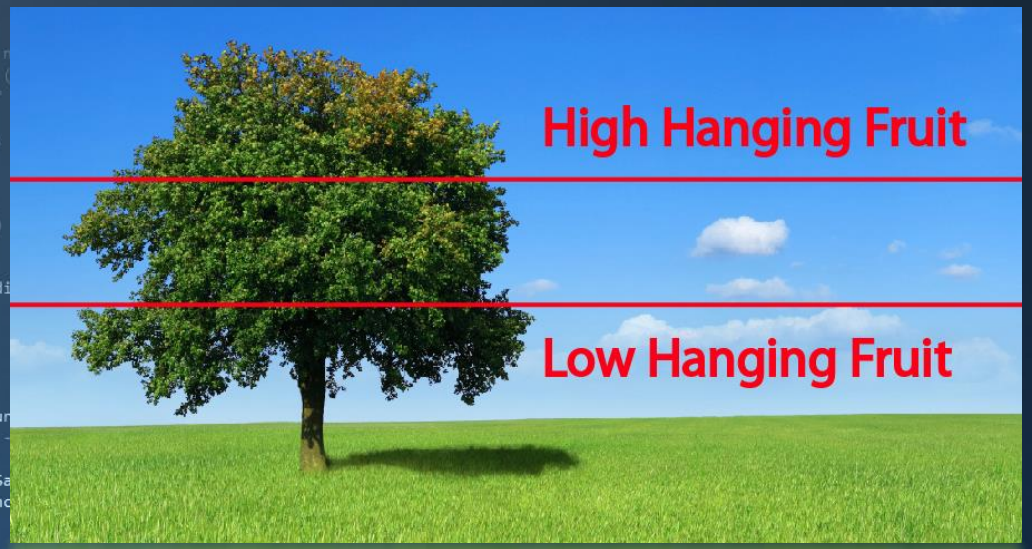
- State of the Art **TL;DR version**
- Wavefront Path Tracing
- 2023 Update: SER
- Assignment 3 Kick-Off
- What's Next

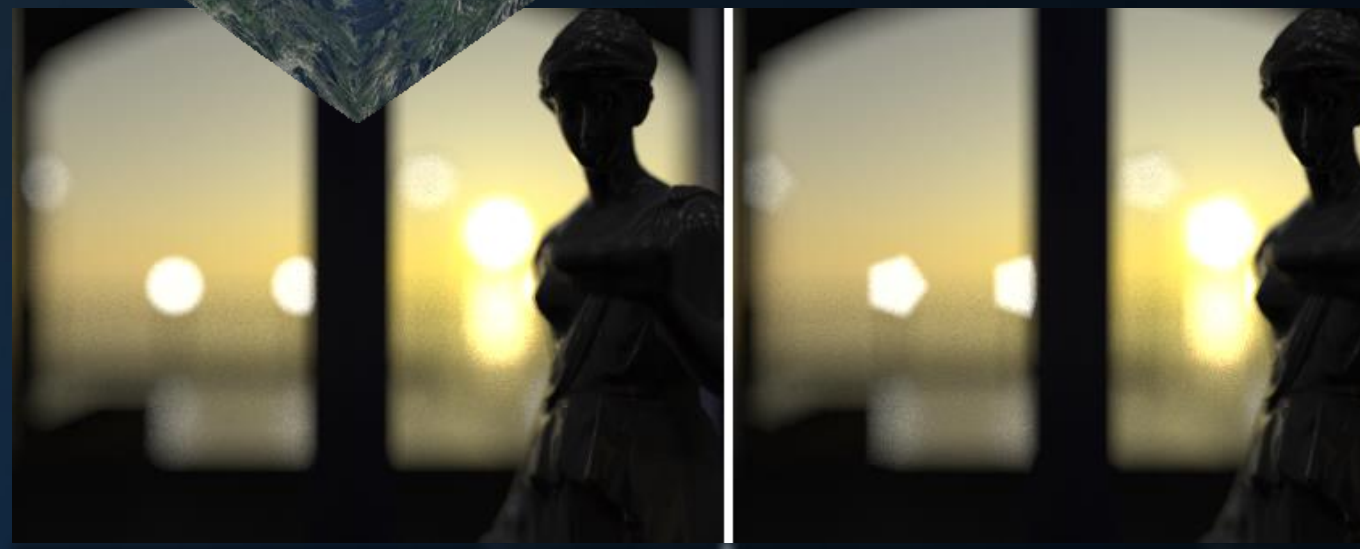


MAGR – What's Next

Upcoming Topics

Thursday: "Various"





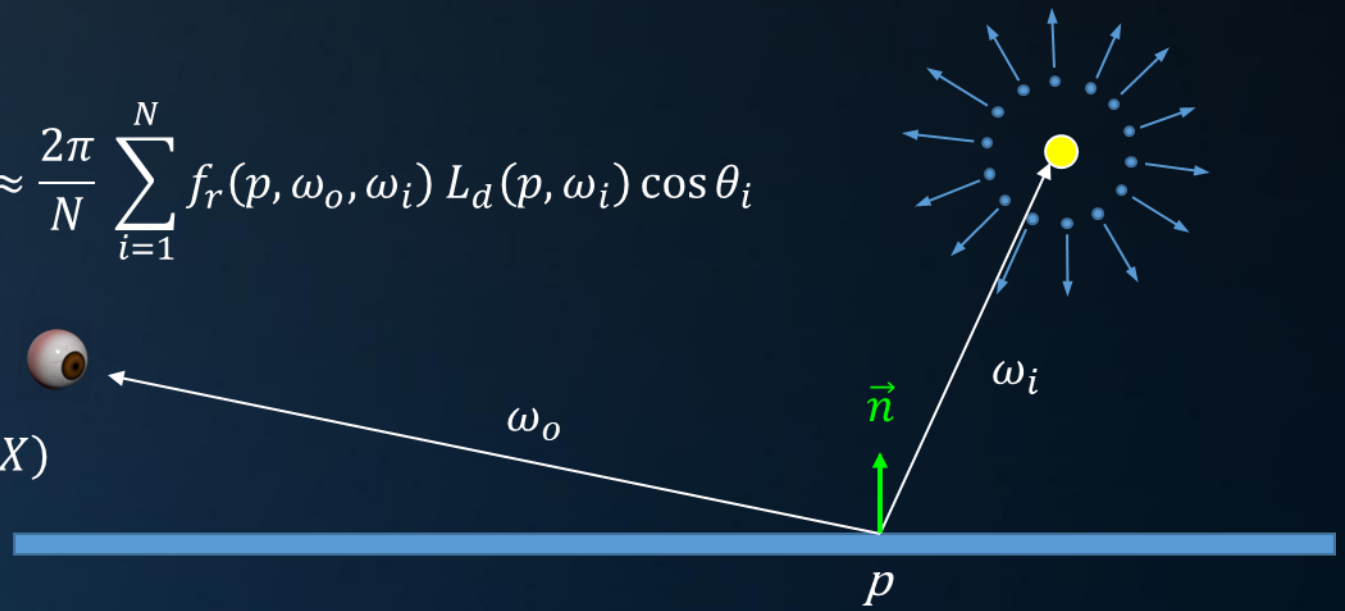
MAGR – What's Next

Upcoming Topics

Next week: "Probability"

$$L_o(p,\omega_o) \approx \frac{2\pi}{N}\sum_{i=1}^{N} f_r(p,\omega_o,\omega_i)\, L_d(p,\omega_i)\cos\theta_i$$

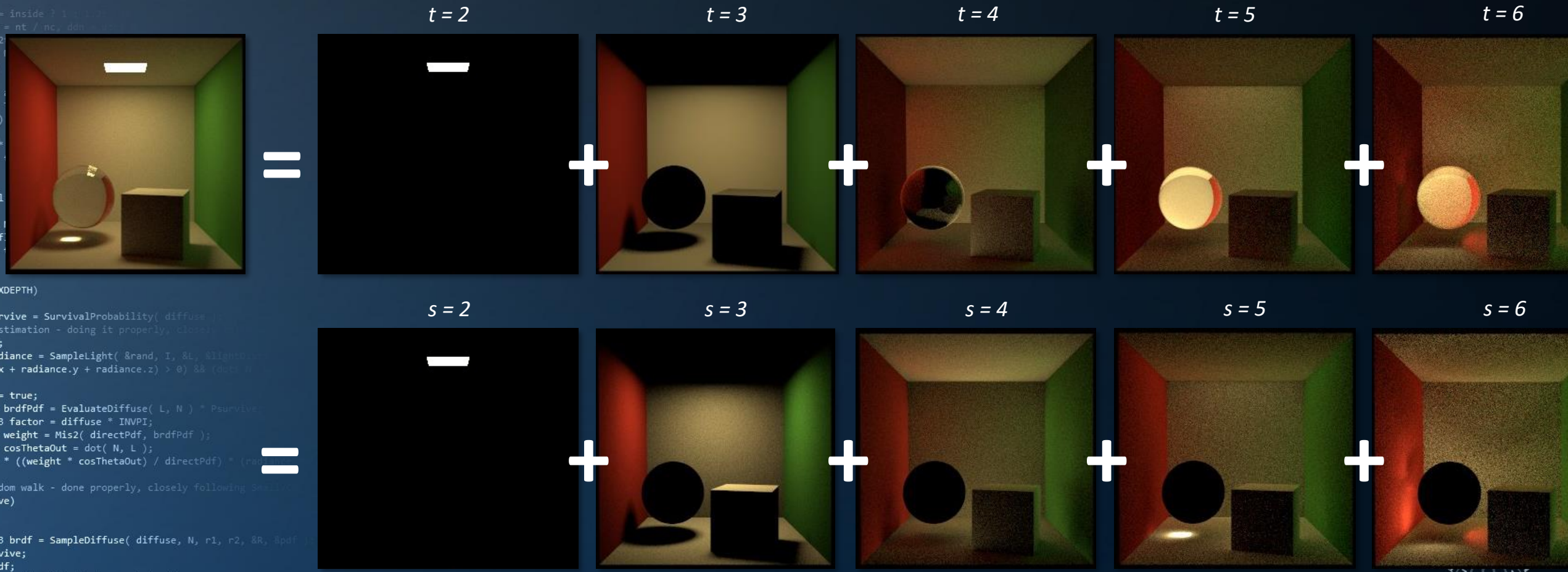
$$E\big(f(X)\big) \approx \frac{B-A}{N}\sum_{i=1}^{N} f(X)$$



MAGR – What's Next

Upcoming Topics

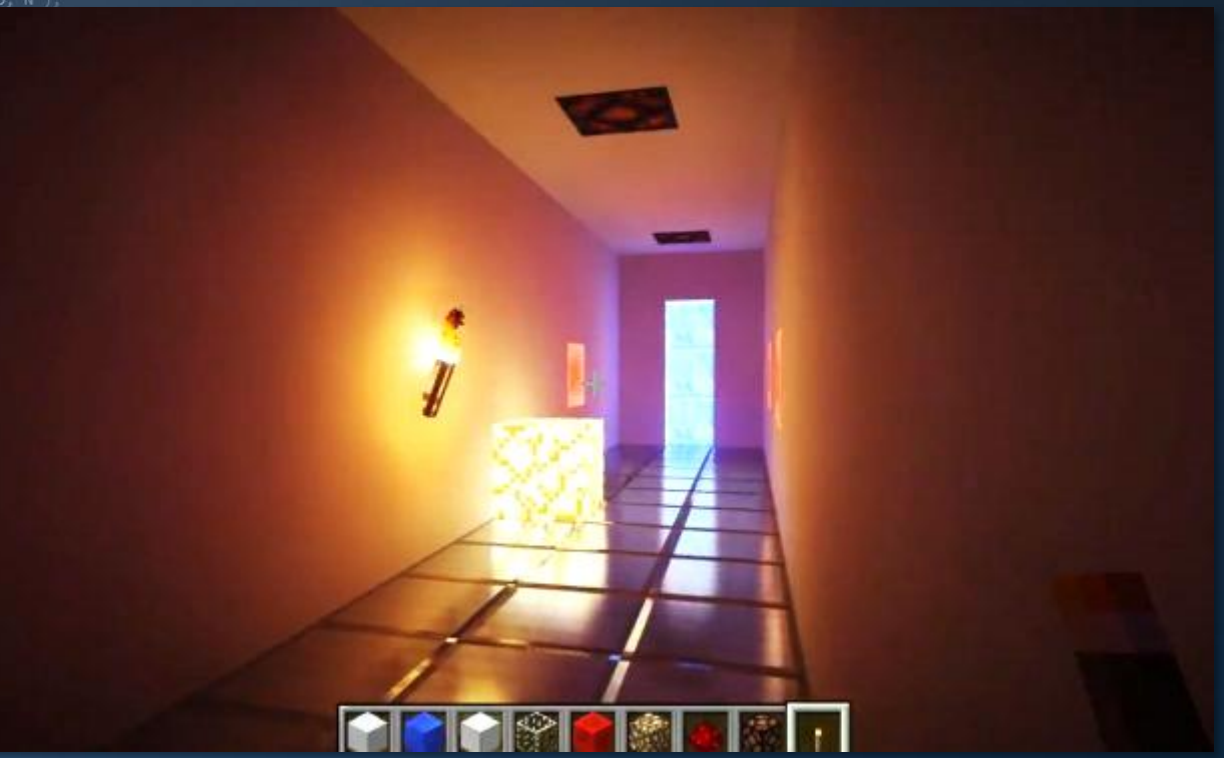
Next week: "Bidirectional"

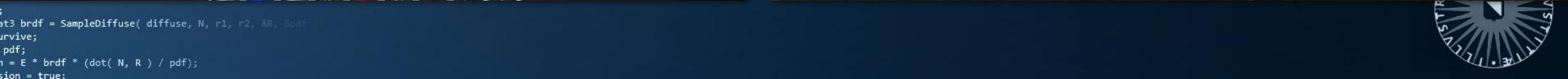


MAGR – What's Next

Upcoming Topics

Later: "ReSTIR"

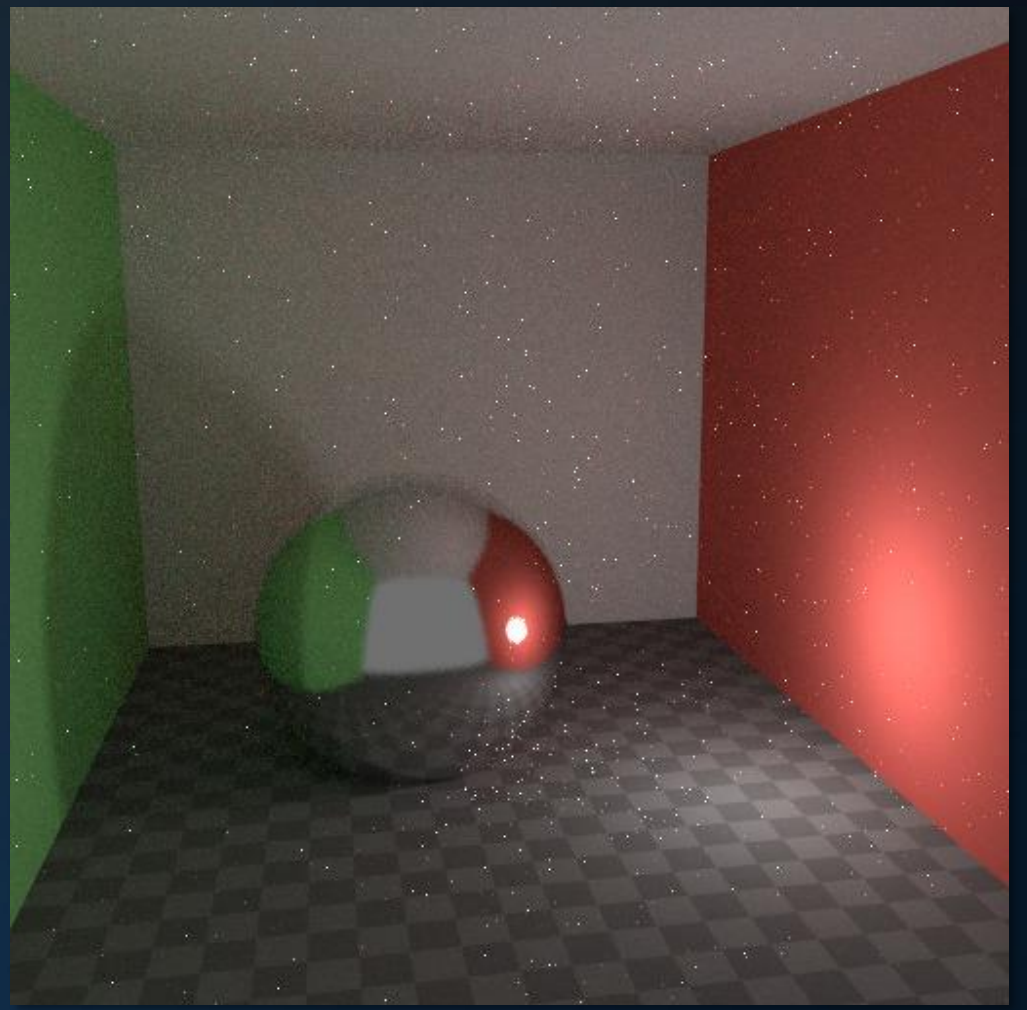




MAGR – What's Next

Upcoming Topics

Later: "Filtering"



MAGR – What's Next

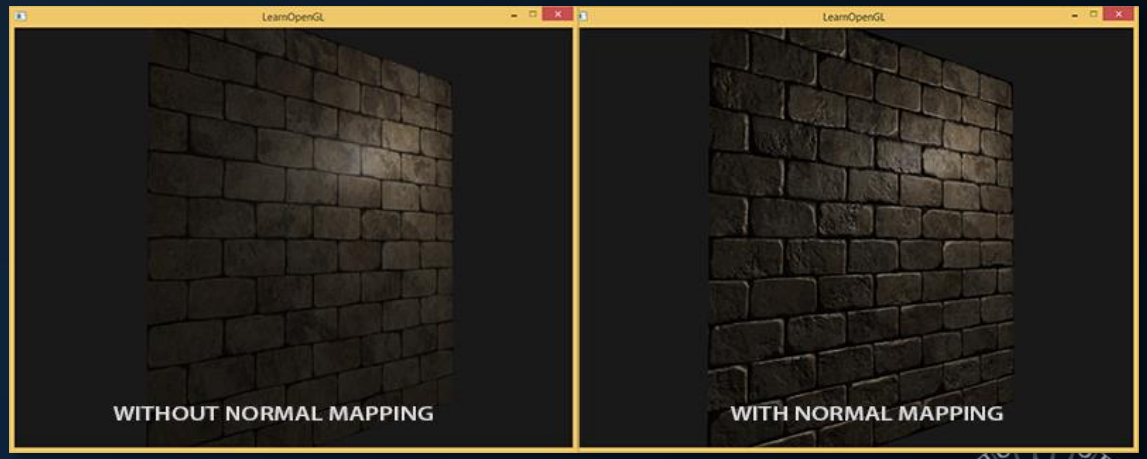
Upcoming Topics

Later: "Bit's & Pieces"

+

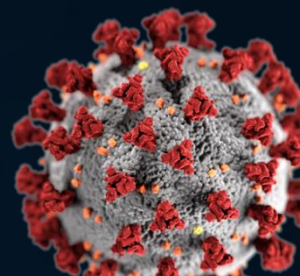
EXAM

PRACTICE



Today's Agenda:

- State of the Art **TL;DR version**
- Wavefront Path Tracing
- 2023 Update: SER
- Assignment 3 Kick-Off
- What's Next



INFOMAGR – Advanced Graphics

Jacco Bikker - November 2022 - February 2023

END of “GPU Ray Tracing (2)”

next lecture: “Variance Reduction (2)”