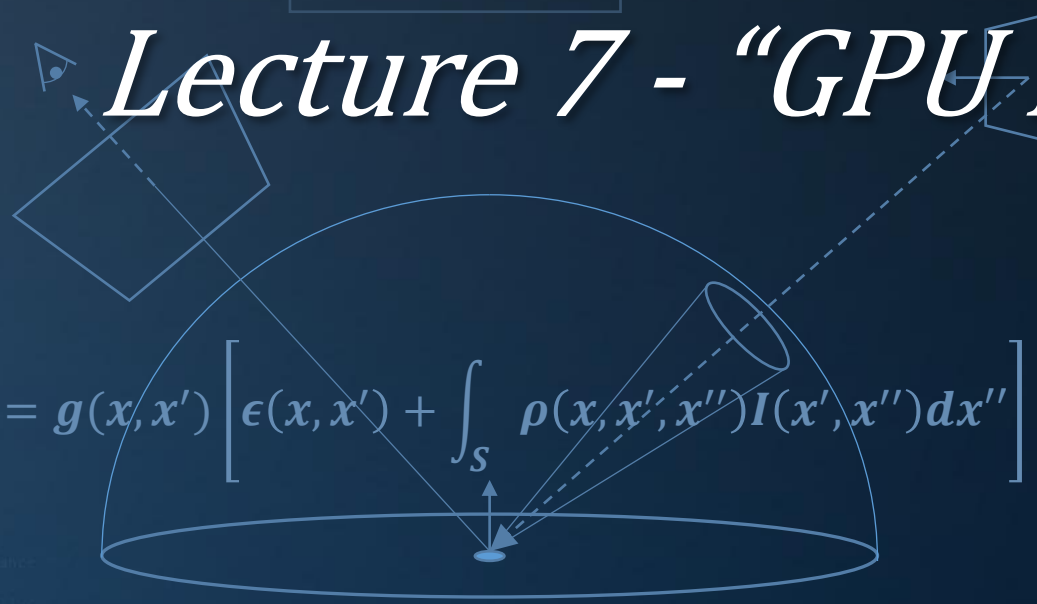
INFOMAGR – Advanced Graphics

Jacco Bikker - November 2022 - February 2023

Lecture 7 - "GPU Ray Tracing (1)"

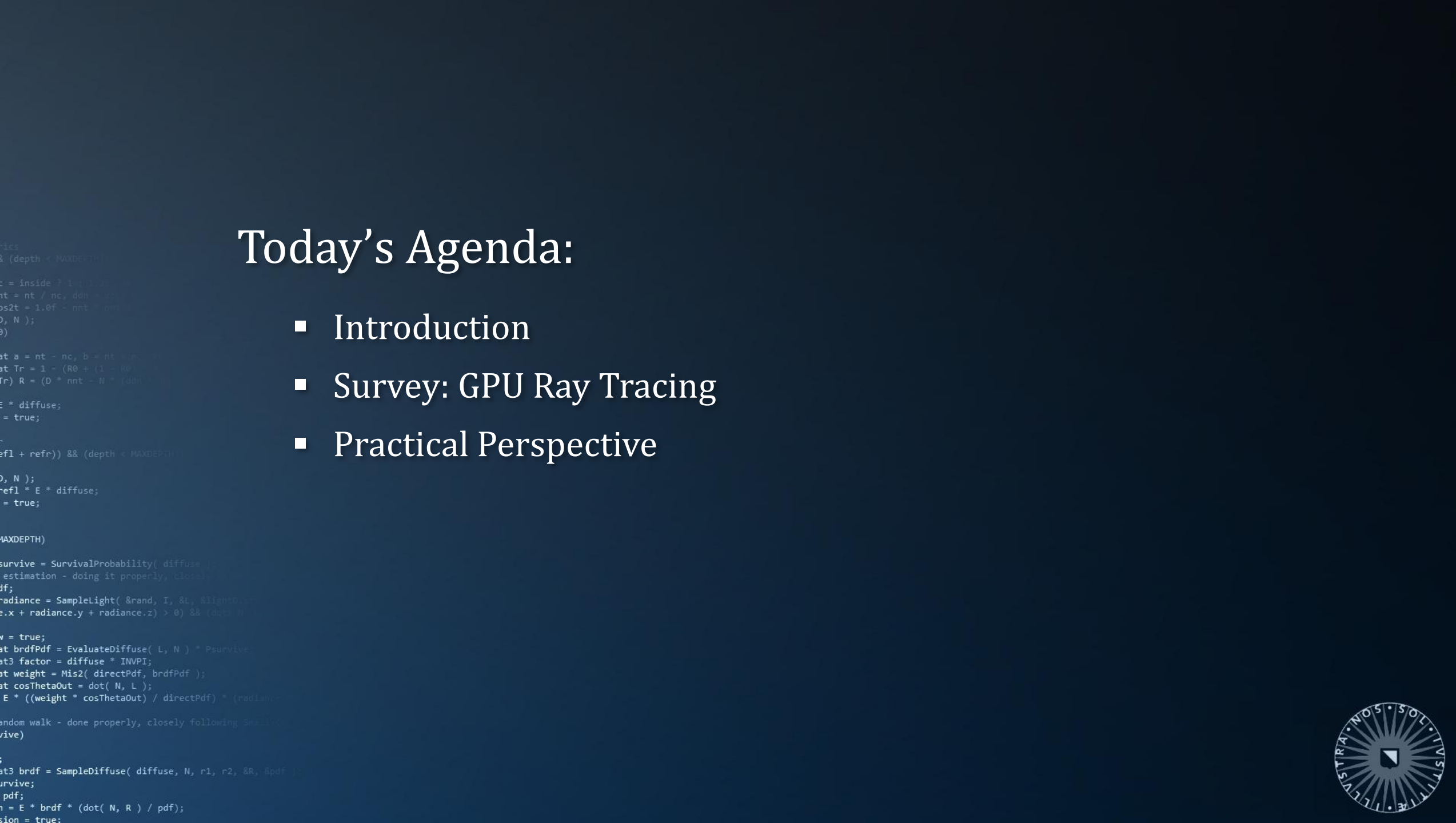


$$I(x,x') = g(x,x')\left[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right]$$

Welcome!

Today’s Agenda:

- Introduction
- Survey: GPU Ray Tracing
- Practical Perspective



Introduction

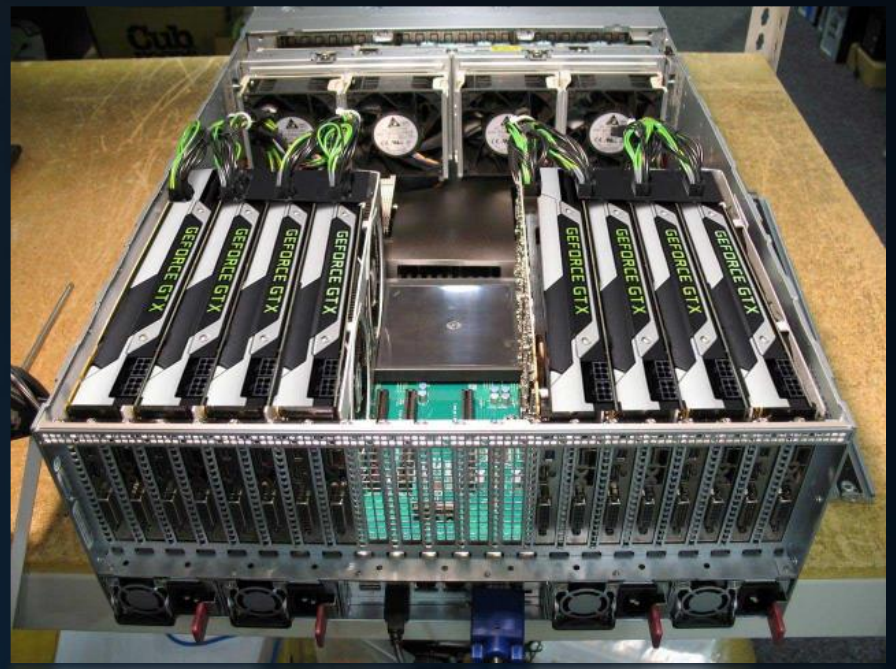
Transferring Ray Tracing to the GPU

Platform characteristics:

- Massively parallel
- SIMT
- High bandwidth
- Massive compute potential
- Slow connection to host

Challenges:

- Thread state must be small
- Efficiency requires coherent control flow

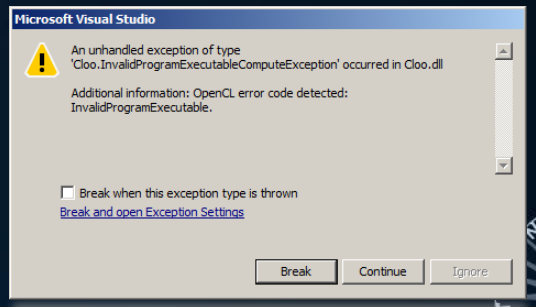


Introduction

Transferring Ray Tracing to the GPU

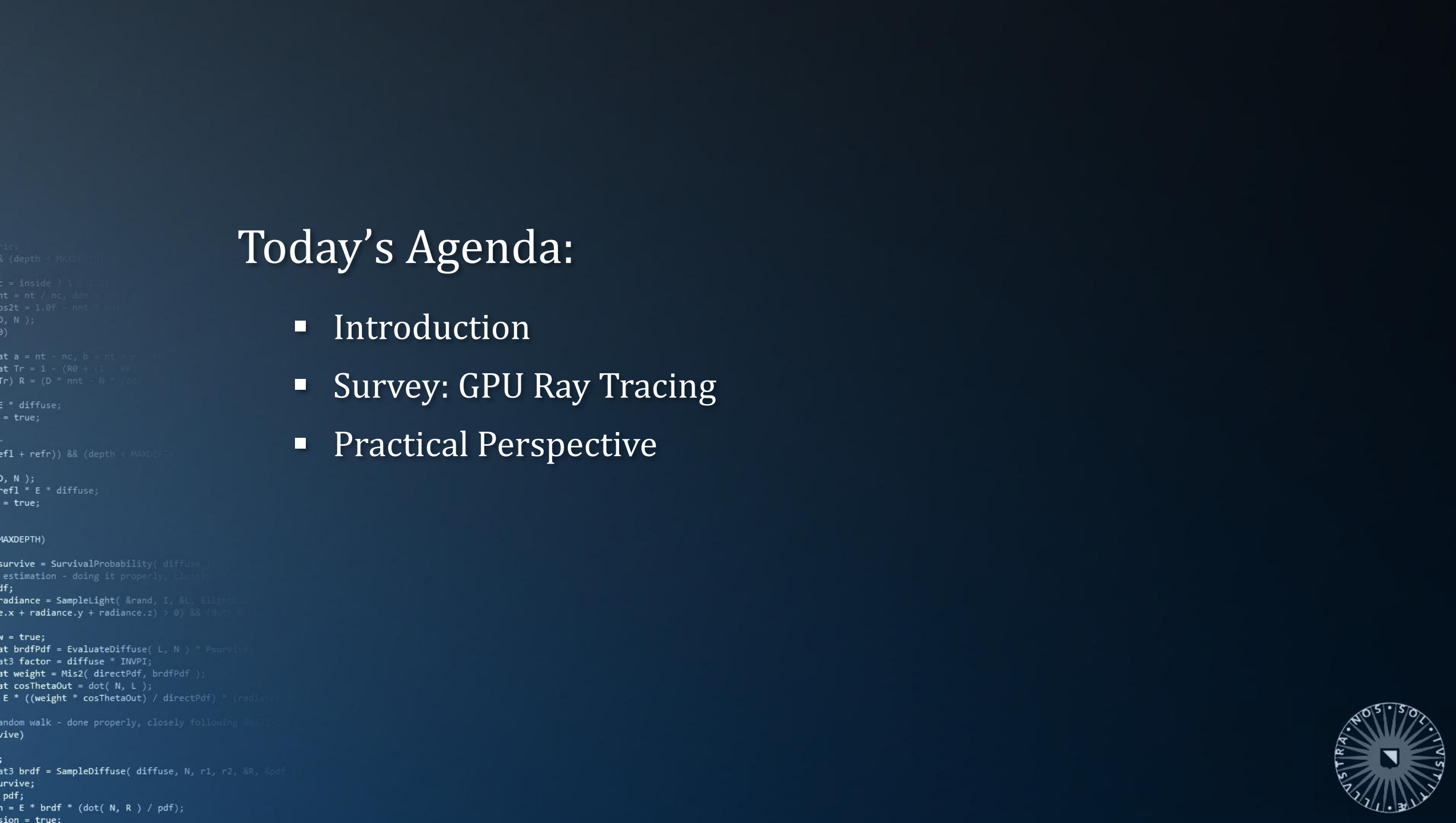
Survey

- Understand evolution of graphics hardware
- Understand characteristics of modern GPUs
- Investigate algorithms designed with these characteristics in mind



Today’s Agenda:

- Introduction
- Survey: GPU Ray Tracing
- Practical Perspective



Survey

2002

Ray Tracing on Programmable Graphics Hardware*

Graphics hardware in 2002:

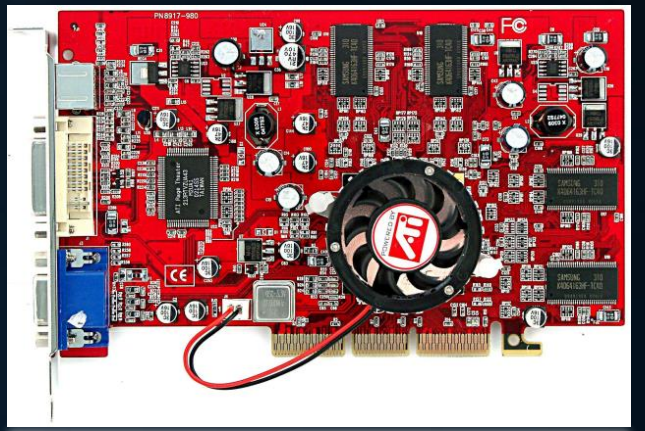
- Vertex and fragment shaders only:
- Simple instruction sets
- Integer-only (fixed-point) fragment shaders
- Limited number of instructions per program
- Limited number of inputs and outputs
- No loops, no conditional branching

Expectations:

- Floating point fragment shaders
- Improved instruction sets
- Multiple outputs per fragment shader

No branching

NVidia GeForce 3



ATi Radeon 8500

*: Ray tracing on programmable graphics hardware, Purcell et al., 2002.

Survey

2002

Ray Tracing on Programmable Graphics Hardware

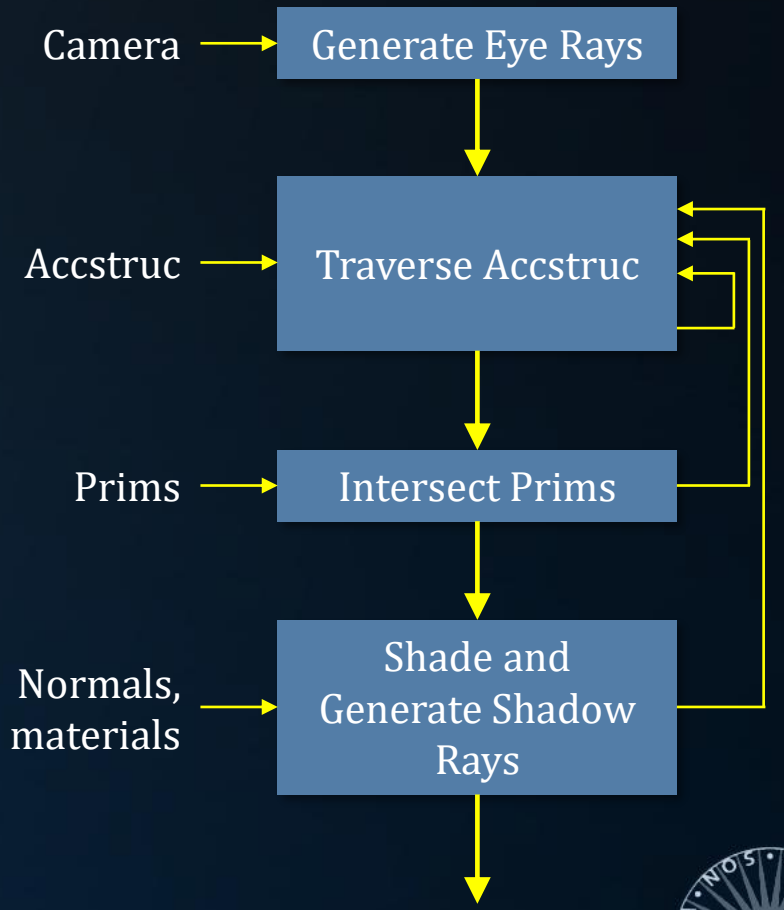
Challenge: to map ray tracing to *stream processing**.

Stage 1: Produce a stream of primary rays.
(a shader, executed for each pixel of the quad, sets up 1 ray)

Stage 2: For each ray in the stream, find a voxel containing geometry. (a shader, ...)

Stage 3: For each voxel in the stream, intersect the ray with the primitives in the voxel.

Stage 4: For each intersection point in the stream, apply shading and produce a new ray.



*: https://en.wikipedia.org/wiki/Stream_processing

Survey

2002

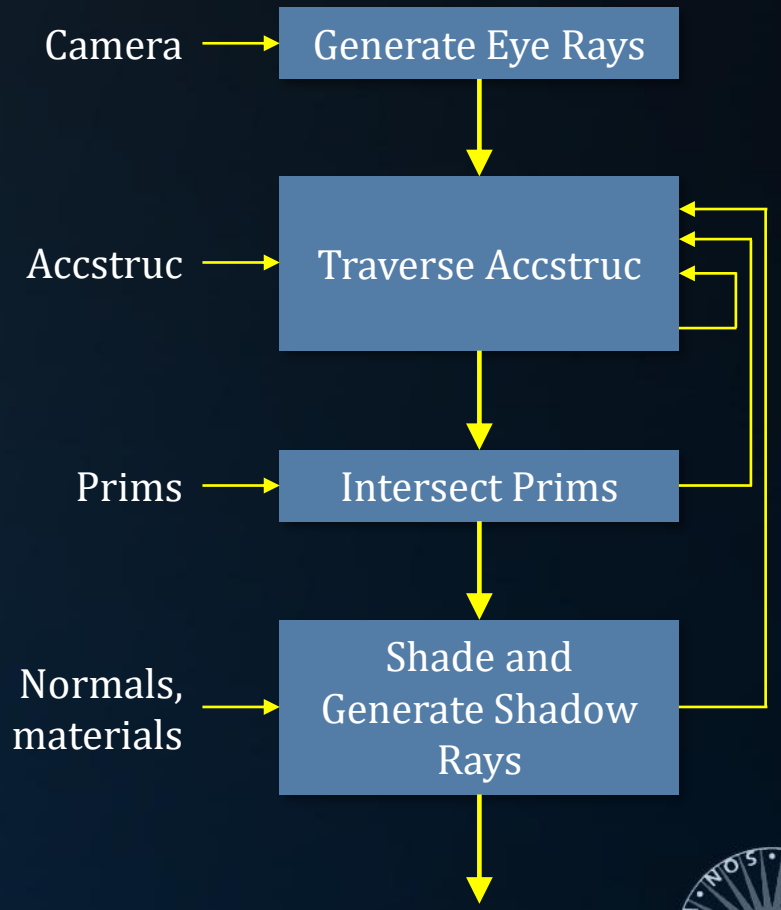
Ray Tracing on Programmable Graphics Hardware

Stream computing without flow control:

Assign a state to each ray.

1. Traversing;
2. intersecting;
3. shading;
4. done.

Now, for each program render a quad using a *stencil* based on the state; this enables the program only for rays in that state*.

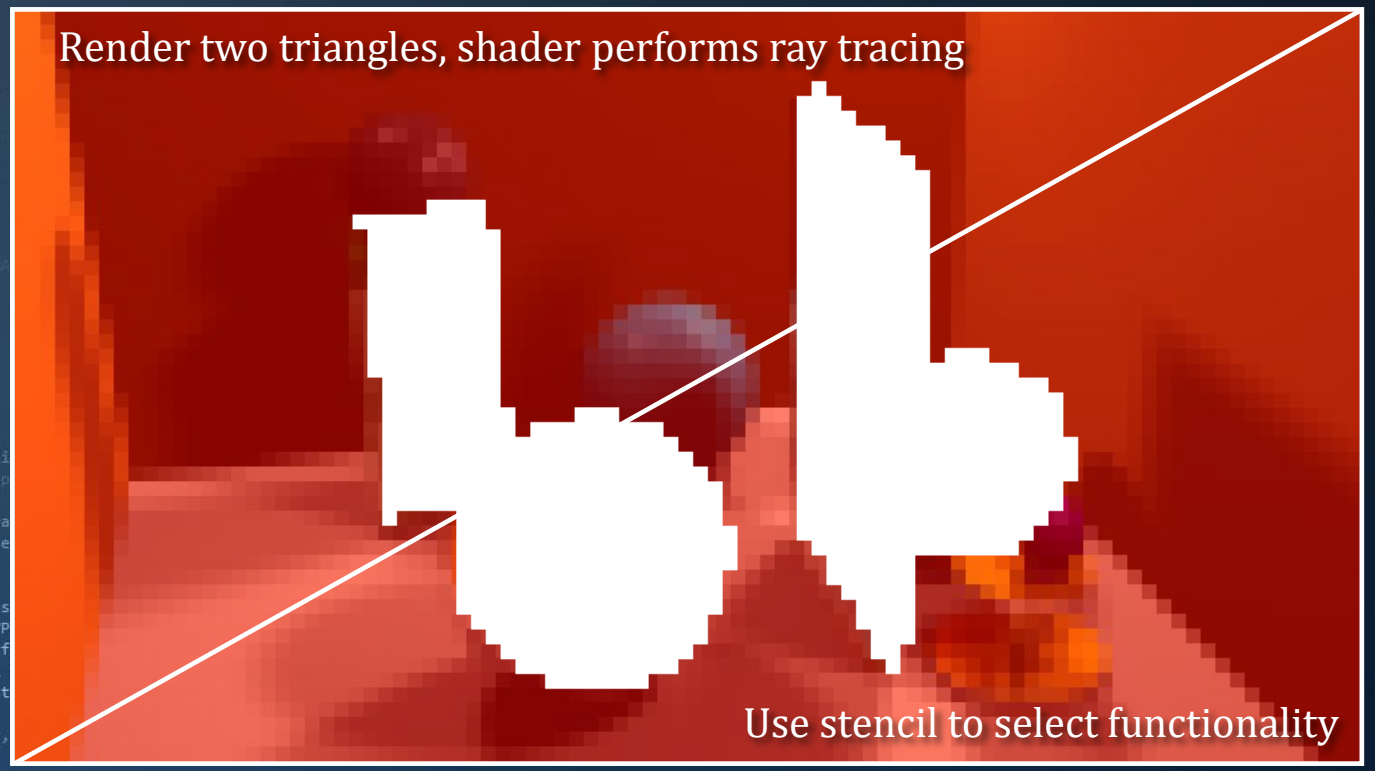


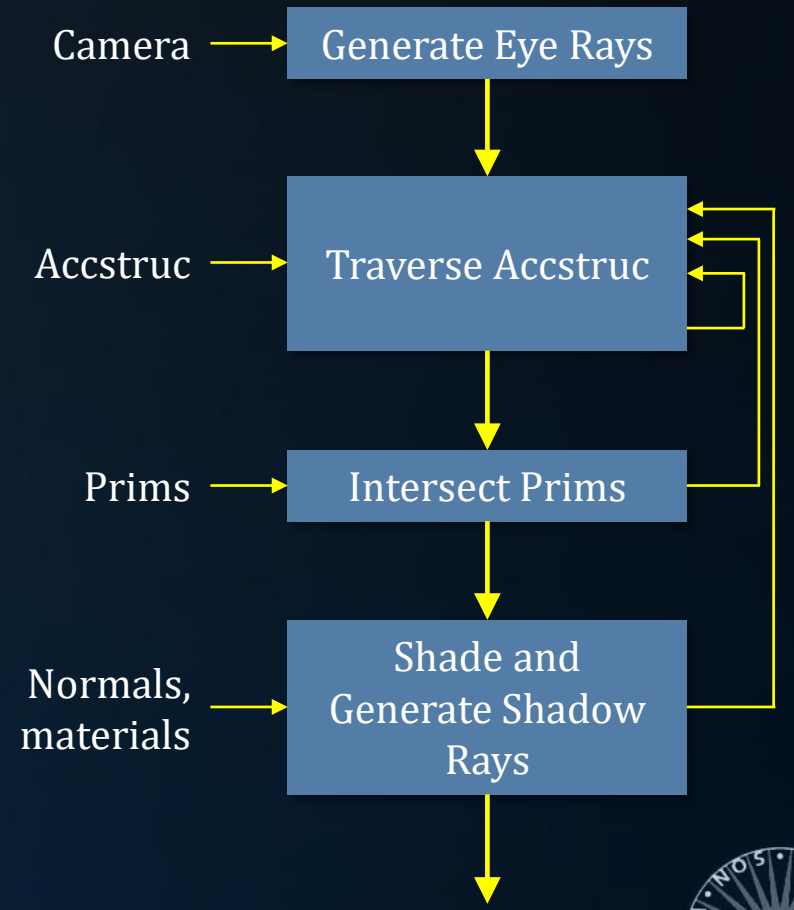
*: Interactive multi-pass programmable shading, Peercy et al., 2000.

Survey

2002 Ray Tracing on Programmable Graphics Hardware

Stream computing without flow control:





Survey

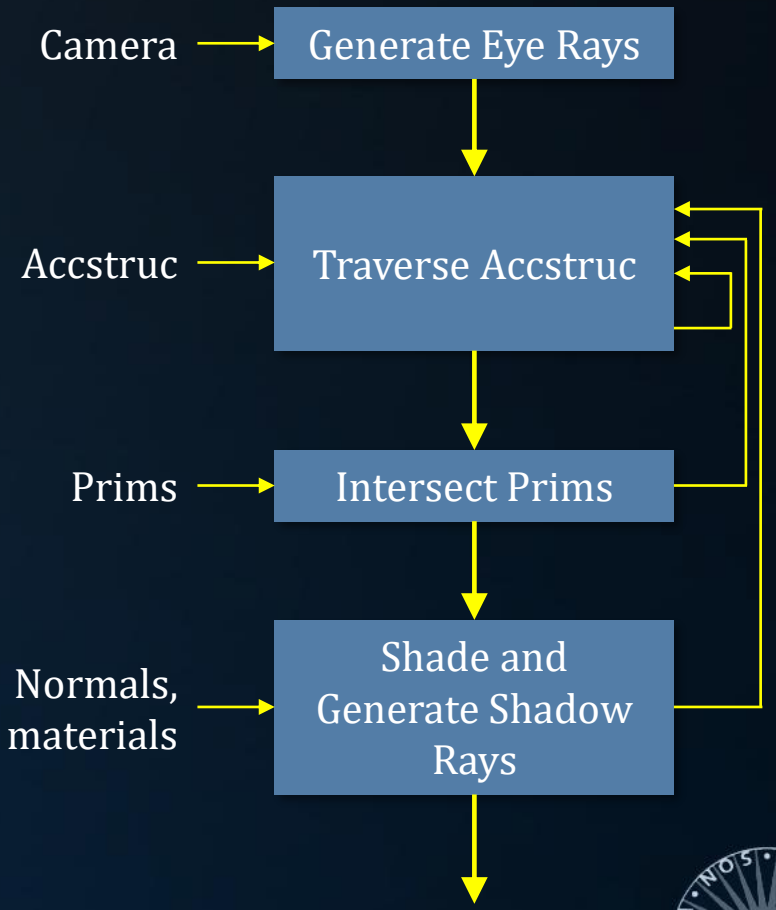
2002

Ray Tracing on Programmable Graphics Hardware

Acceleration structure (grid) traversal:

1. setup traversal;
2. one step using 3D-DDA*.

Note that *each step* through the grid requires *one pass*.



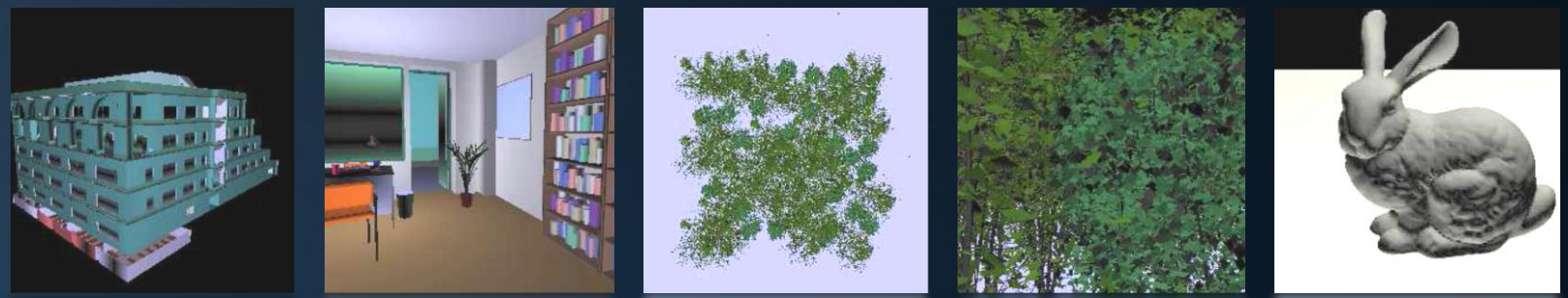
*: Accelerated ray tracing system. Fujimoto et al., 1986.

Survey

2002

Ray Tracing on Programmable Graphics Hardware

Results



passes	2443	1198	1999	2835	1085
efficiency	0.009	0.061	0.062	0.062	0.105

Here, ‘efficiency’ is the average ratio of active fragments during each pass.

Survey

2002

Ray Tracing on Programmable Graphics Hardware

Conclusions

- Ray tracing can be done on a GPU
- GPU outperforms CPU by a factor 3x (for triangle intersection only)
- Flow control is needed to make the full ray tracer efficient.

Survey

2005

KD-Tree Acceleration Structures for a GPU Raytracer*

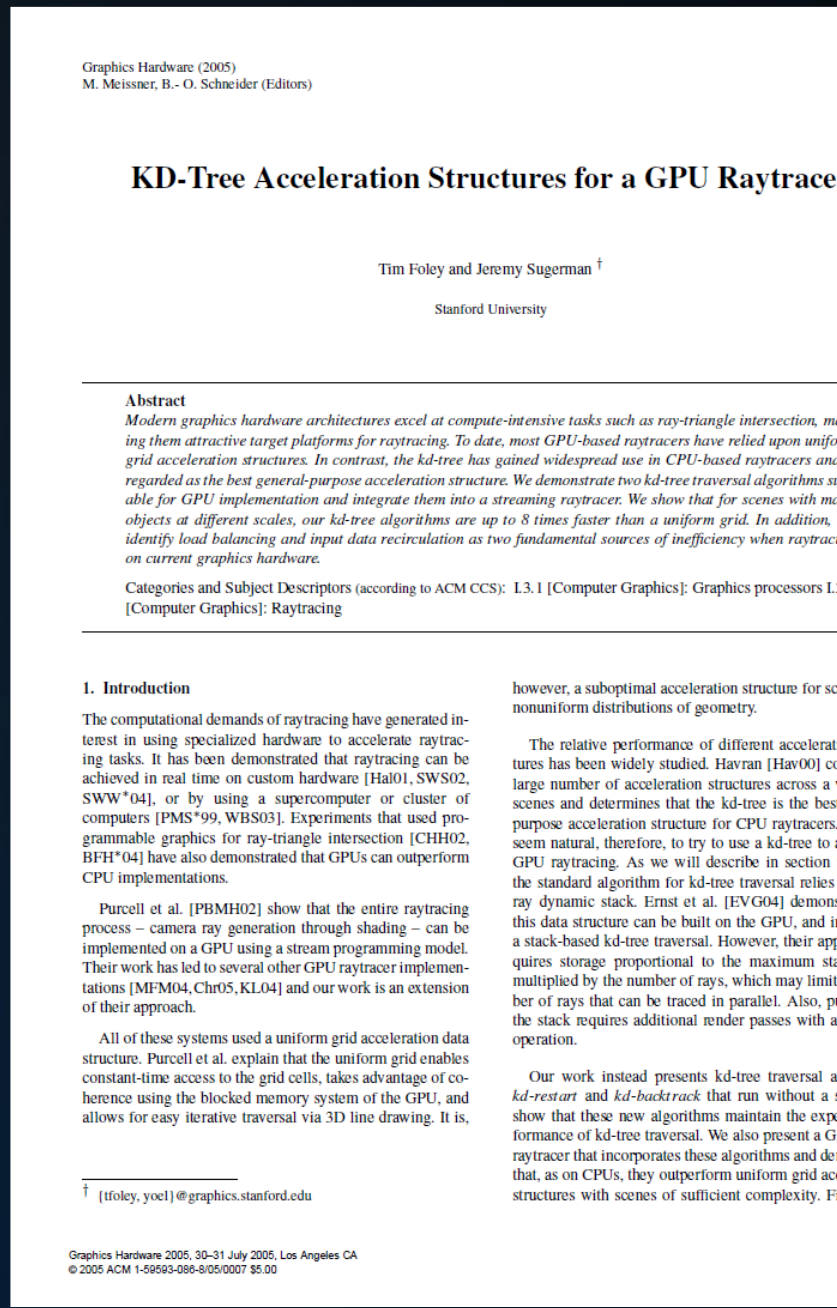
Observations on previous work:

- Grid only: doesn't adapt to local scene complexity
- kD-tree traversal could theoretically be done on the GPU, but the stack is a problem.

Goal:

- Implement kD-tree traversal without stack.

*: KD-Tree Acceleration Structures for a GPU Raytracer, Foley & Sugerman, 2005



Graphics Hardware (2005)
M. Meissner, B.- O. Schneider (Editors)

KD-Tree Acceleration Structures for a GPU Raytrace

Tim Foley and Jeremy Sugerman †

Stanford University

Abstract

Modern graphics hardware architectures excel at compute-intensive tasks such as ray-triangle intersection, m ing them attractive target platforms for raytracing. To date, most GPU-based raytracers have relied upon unifo grid acceleration structures. In contrast, the kd-tree has gained widespread use in CPU-based raytracers and regarded as the best general-purpose acceleration structure. We demonstrate two kd-tree traversal algorithms su able for GPU implementation and integrate them into a streaming raytracer. We show that for scenes with mo objects at different scales, our kd-tree algorithms are up to 8 times faster than a uniform grid. In addition, identify load balancing and input data recirculation as two fundamental sources of inefficiency when raytraci on current graphics hardware.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors I. [Computer Graphics]: Raytracing

1. Introduction

The computational demands of raytracing have generated interest in using specialized hardware to accelerate raytracing tasks. It has been demonstrated that raytracing can be achieved in real time on custom hardware [Hal01, SWS02, SWW*04], or by using a supercomputer or cluster of computers [PMS*99, WBS03]. Experiments that used programmable graphics for ray-triangle intersection [CHH02, BFH*04] have also demonstrated that GPUs can outperform CPU implementations.

Purcell et al. [PBMH02] show that the entire raytracing process – camera ray generation through shading – can be implemented on a GPU using a stream programming model. Their work has led to several other GPU raytracer implementations [MFM04, Chr05, KL04] and our work is an extension of their approach.

All of these systems used a uniform grid acceleration data structure. Purcell et al. explain that the uniform grid enables constant-time access to the grid cells, takes advantage of coherence using the blocked memory system of the GPU, and allows for easy iterative traversal via 3D line drawing. It is, however, a suboptimal acceleration structure for sc nonuniform distributions of geometry.

The relative performance of different accelerat tures has been widely studied. Havran [Hav00] co large number of acceleration structures across a scenes and determines that the kd-tree is the bes purpose acceleration structure for CPU raytracers. seem natural, therefore, to try to use a kd-tree to GPU raytracing. As we will describe in section the standard algorithm for kd-tree traversal relies ray dynamic stack. Ernst et al. [EVG04] demons this data structure can be built on the GPU, and i a stack-based kd-tree traversal. However, their app quires storage proportional to the maximum st multiplied by the number of rays, which may limit ber of rays that can be traced in parallel. Also, p the stack requires additional render passes with a operation.

Our work instead presents kd-tree traversal a *kd-restart* and *kd-backtrack* that run without a show that these new algorithms maintain the exp formance of kd-tree traversal. We also present a G raytracer that incorporates these algorithms and de that, as on CPUs, they outperform uniform grid ac structures with scenes of sufficient complexity. F

† {tfoley, yoel}@graphics.stanford.edu

Graphics Hardware 2005, 30–31 July 2005, Los Angeles CA

Survey

2005

KD-Tree Acceleration Structures for a GPU Ray Tracer

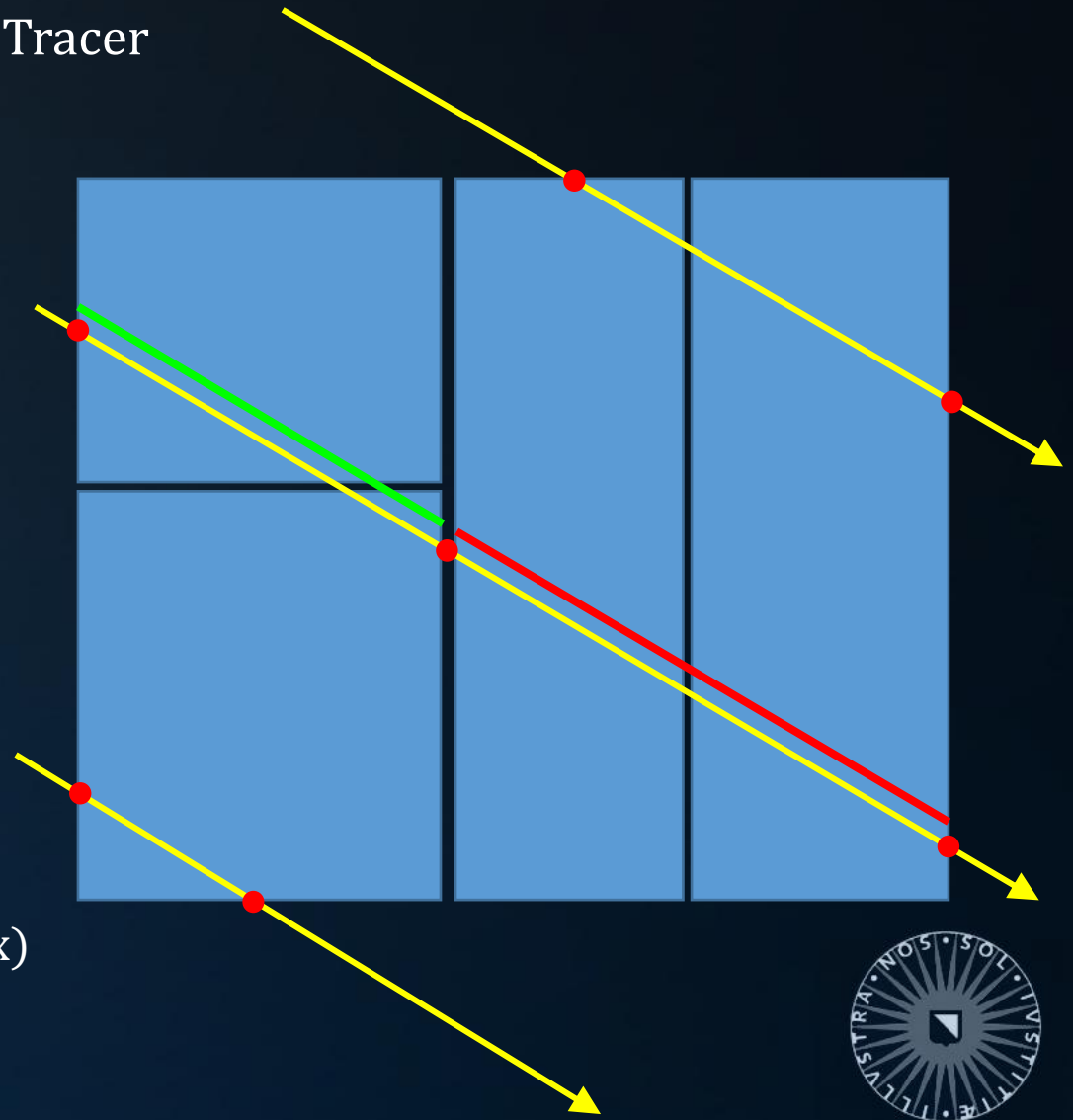
Recall standard kD-tree traversal:

Setup:

1. tmax, tmin = intersect(ray, root bounds);

Root node:

2. Find intersection t with split plane
3. If tmin <= t <= tmax:
   - Process near child with segment (tmin, t)
   - Process far child with segment (t, tmax)
4. else if t > tmax:
   - Process left child with segment (tmin,tmax)
5. else
   - Process right child with segment (tmin,tmax)



Survey

2005

KD-Tree Acceleration Structures for a GPU Ray Tracer

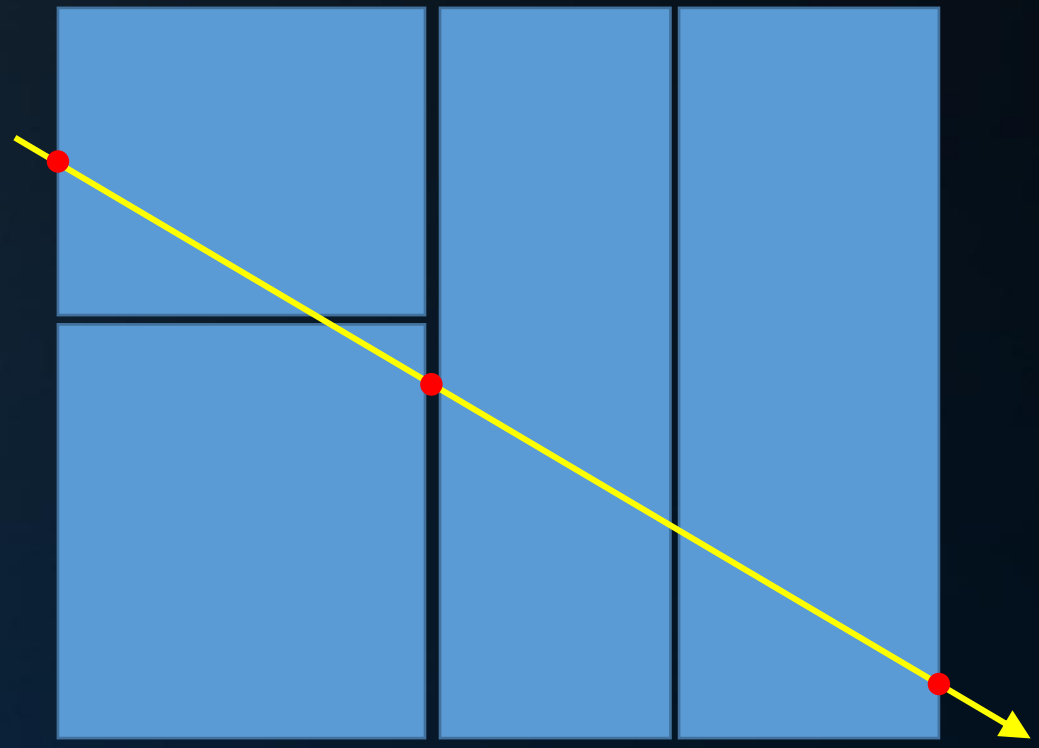
Recall standard kD-tree traversal:

Setup:

1. tmax, tmin = intersect(ray, root bounds);

Root node:

2. Find intersection t with split plane
3. If tmin <= t <= tmax:
   - **Push far child**
   - **Continue with near child**
4. else if t > tmax:
   - Process left child with segment (tmin,tmax)
5. else
   - Process right child with segment (tmin,tmax)



Survey

2005

KD-Tree Acceleration Structures for a GPU Ray Tracer

Traversing the tree without a stack:

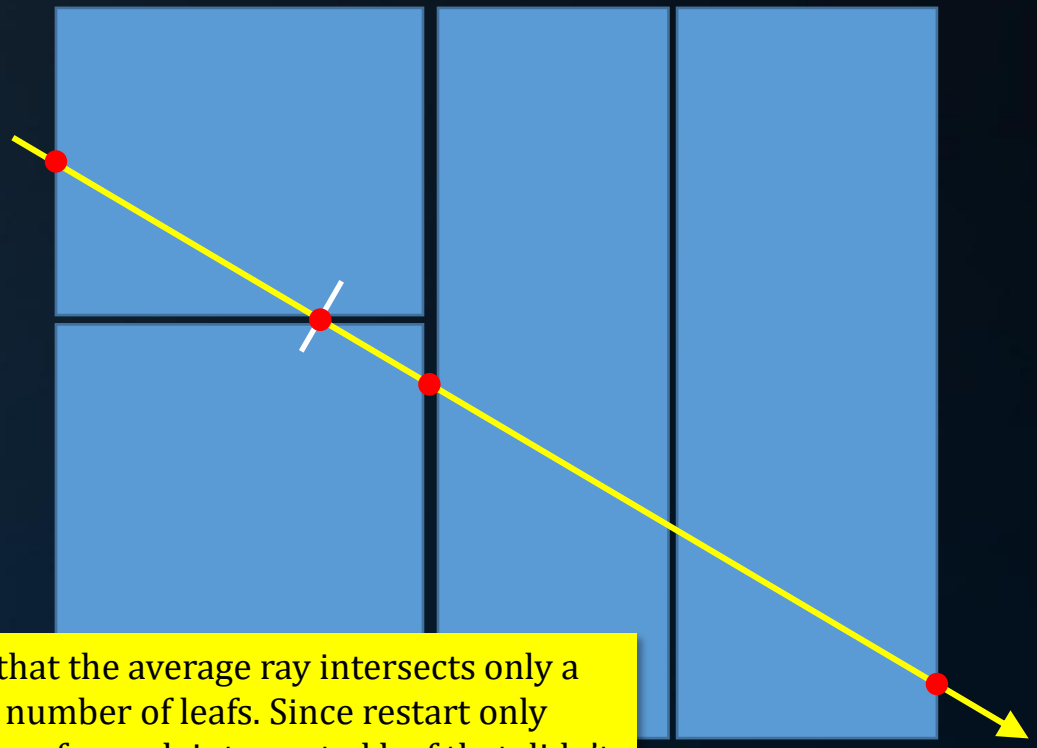
If we always pick the nearest child, the only value that will change is tmax.

Setup:

1. tmax, tmin = intersect(ray, root bounds);

2. Always pick the nearest child.
3. Once we have processed a leaf, restart with:
   - tmin=tmax
   - tmax= intersect(ray, root bounds)

This algorithm is referred to as *kd-restart*.



Note that the average ray intersects only a small number of leafs. Since restart only happens for each intersected leaf that didn't yield an intersection point, the expected cost is still $O(\log n)$.

Survey

2005

KD-Tree Acceleration Structures for a GPU Ray Tracer

We can reduce the cost of a restart by storing node bounds and a parent pointer with each node.

Instead of restarting at the root, we now restart at the first ancestor that has a non-empty intersection with (tmin,tmax).

This algorithm is referred to as *kd-backtrack*.

Survey

2005

KD-Tree Acceleration Structures for a GPU Ray Tracer

Implementation: each ray is assigned a state:

1. Initialize: finds tmin,tmax for each ray in the input stream
2. Down: traverses each ray down by one step
3. Leaf: handles ray/leaf intersection for each ray
4. Intersect: performs actual ray/triangle intersection
5. Continue: decides whether each ray is done or needs to restart / backtrack
6. Up: performs one backtrack step for each ray in the input stream.

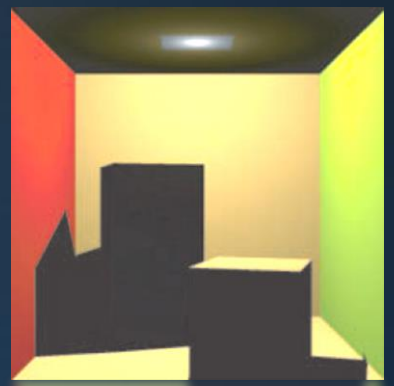
As before, the state is used to mask rays in the input stream when executing each of the 6 programs.

Survey

2005

KD-Tree Acceleration Structures for a GPU Ray Tracer

Results (ms*):




brute force	23	4620	4770	7350
grid	63	357	8344	2687
kd-restart	80	701	968	992
kd-backtrack	84	690	946	857

*: Hardware: 256MB ATI X800 XT PE (2004), rendering @ 512x512, time in milliseconds.

Survey

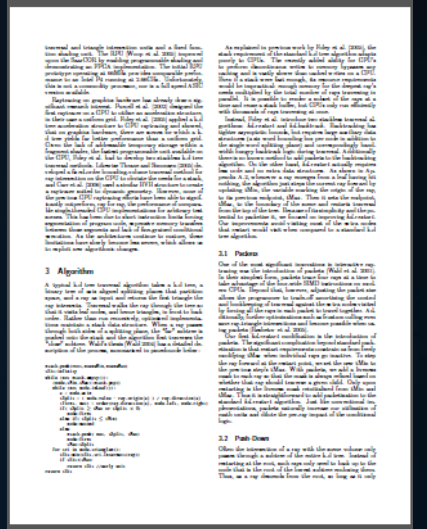
2007

Interactive k-d tree GPU ray tracing*
Stackless KD-tree traversal for high performance GPU ray tracing**

Observations on previous work:

- GPU ray tracing performance can't keep up with CPU
- Kd-restart requires substantially more node visits
- Kd-backtrack increases data storage and bandwidth
- Looping and branching wasn't available, but is now.

*: Interactive k-d tree GPU raytracing, Horn et al., 2007
**: Stackless KD-tree traversal for high performance GPU ray tracing, Popov et al., 2007



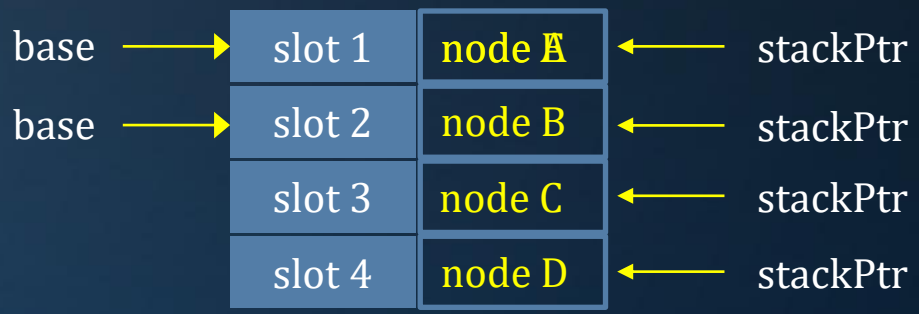
Survey

2007

Interactive k-d tree GPU ray tracing
Stackless KD-tree traversal for high performance GPU ray tracing

Ray tracing with a short stack:

By keeping a fixed-size stack we can prevent a restart in almost all cases.



Survey

2007

Interactive k-d tree GPU ray tracing
Stackless KD-tree traversal for high performance GPU ray tracing

Ray tracing with flow control:

25x performance of the previous paper
1.65x – 2.3x from algorithmic improvements
3.75x from hardware advances

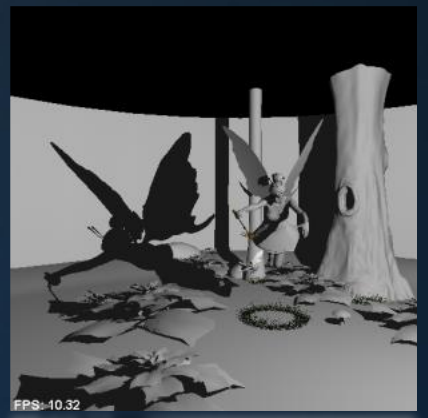
➔ 2.9x from switching from multi-pass to single-pass.

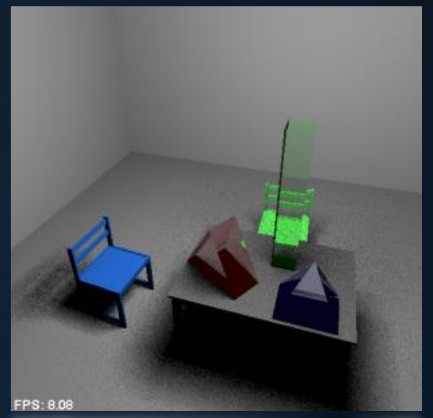
Survey

2007

Interactive k-d tree GPU ray tracing
Stackless KD-tree traversal for high performance GPU ray tracing

Results*:





GPU	12.7	10.6	36.0	16.7
CPU (1 core)	-	3.6	6.6	3.9

*: Hardware: GeForce 8800 GTX / Opteron @ 2.6 Ghz, performance in fps @ 1024x1024.

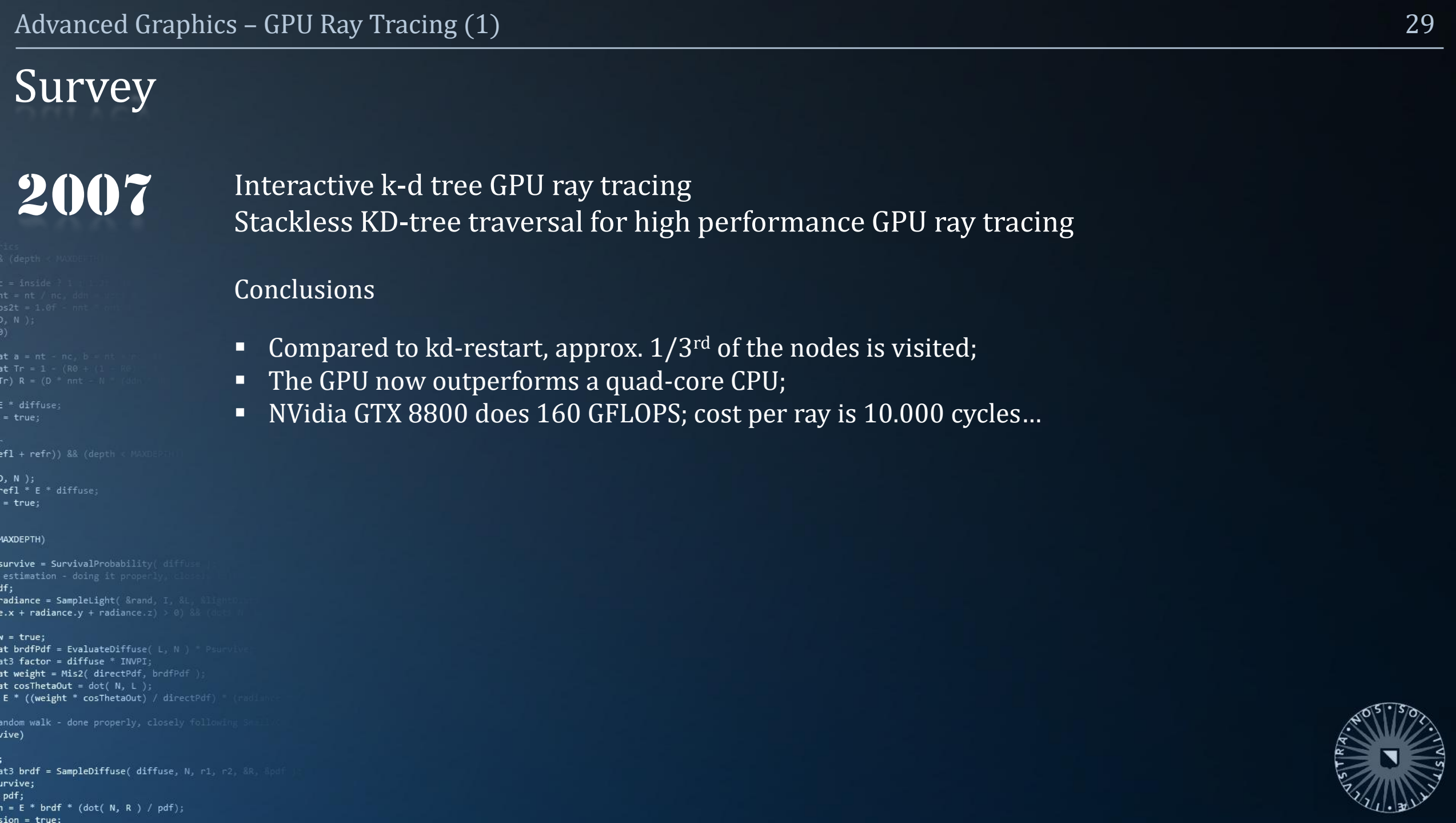
Survey

2007

Interactive k-d tree GPU ray tracing
Stackless KD-tree traversal for high performance GPU ray tracing

Conclusions

- Compared to kd-restart, approx. 1/3rd of the nodes is visited;
- The GPU now outperforms a quad-core CPU;
- NVidia GTX 8800 does 160 GFLOPS; cost per ray is 10.000 cycles…



Survey

2007

Real-time Ray Tracing on GPU with BVH-based Packet Traversal*

Observations on previous work:

- kD-trees limit rendering to static scenes
- kD-trees with ropes are inefficient storage wise
- Popov et al.'s tracer achieves only 33% utilization due to register pressure
- Existing GPU ray tracers do not realize GPU potential
- Existing GPU ray tracers suffer from execution divergence.

Solution:

Use BVH instead of kD-tree.

*: Realtime ray tracing on GPU with BVH-based packet traversal, Günther et al., 2007

Survey

2007

Real-time Ray Tracing on GPU with BVH-based Packet Traversal

Recall: *thread state must be small*.*

> An important difference between kD-tree packet traversal and BVH packet traversal is that kD-tree traversal requires a stack for the packet plus (tmin, tmax) *per ray*, while the BVH packet only requires a stack.

*: To achieve maximum utilization of a G80 GPU, we need 768 threads per multiprocessor (i.e., 24 warps). Each multiprocessor has 16Kb shared memory and 32Kb register space ➔ for 24 warps we have 5 words plus 10 registers per thread available. Beyond that, we are forced to use global memory.

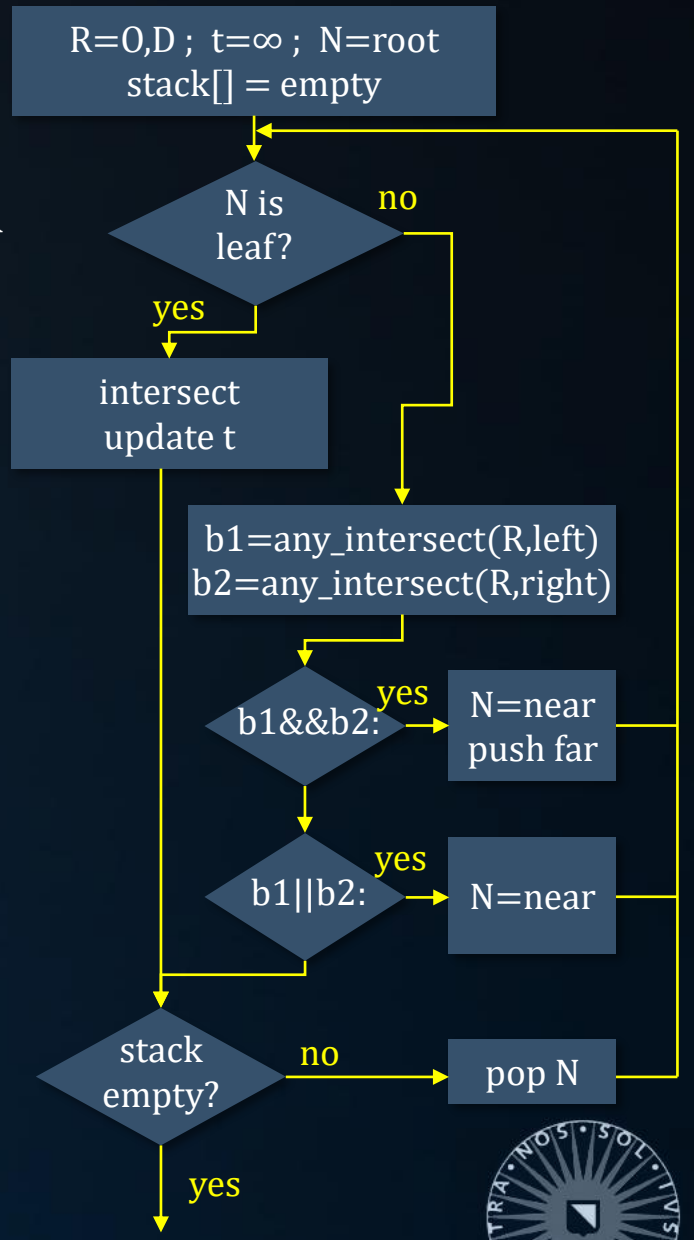
Survey

2007

Real-time Ray Tracing on GPU with BVH-based Packet Traversal

GPU packet traversal for BVH:

1. A packet consists of 8x4 rays, handled by a single *warp*
2. The packet traverses the BVH using *masked traversal* (where t is used as mask)
3. Storage:
   1. Per ray: O, D, t (7 floats)
   2. Per packet: stack

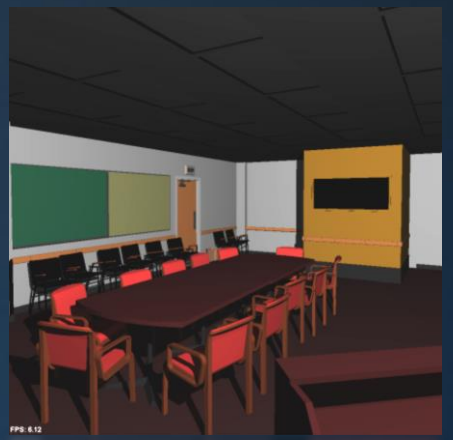


Survey

2007

Real-time Ray Tracing on GPU with BVH-based Packet Traversal

Results*:




primary	19.0	16.2	6.4
+shadow	6.1	5.7	2.9

*: Hardware: GeForce 8800 GTX, rendering at 1024x1024, performance in fps.

Survey

Digest

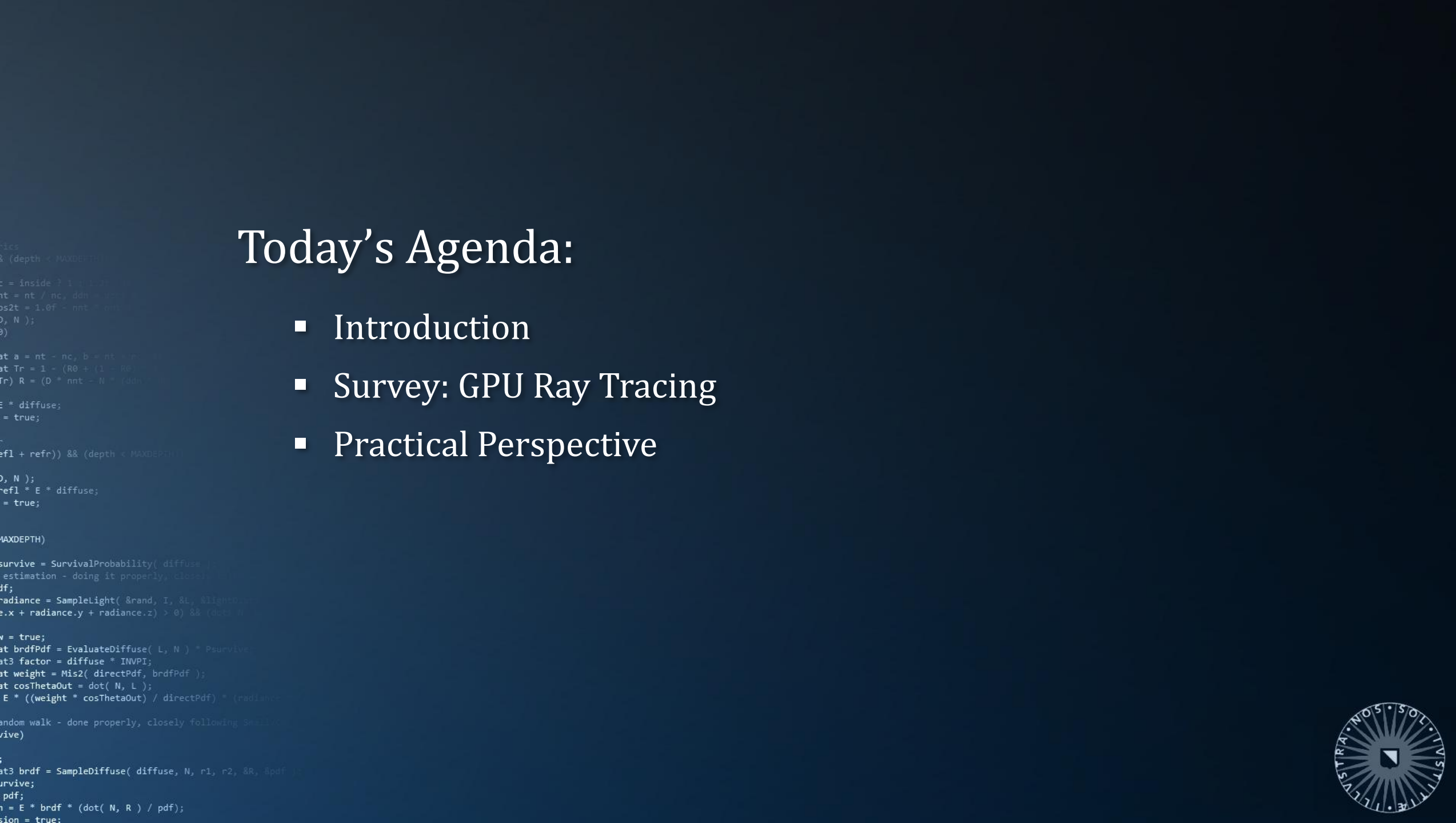
Challenges in GPU ray tracing:

- Utilizing GPU compute potential (getting it to work ➔ beating CPU ➔ efficient)
- Mapping an embarrassingly parallel algorithm to a streaming processor
- Tiny per-thread state (balancing utilization / algorithmic efficiency)
- Freedom in the choice of acceleration structure
- Tracing divergent rays

2002 - 2007

Today’s Agenda:

- Introduction
- Survey: GPU Ray Tracing
- Practical Perspective



5 Faces

5 Faces

2013

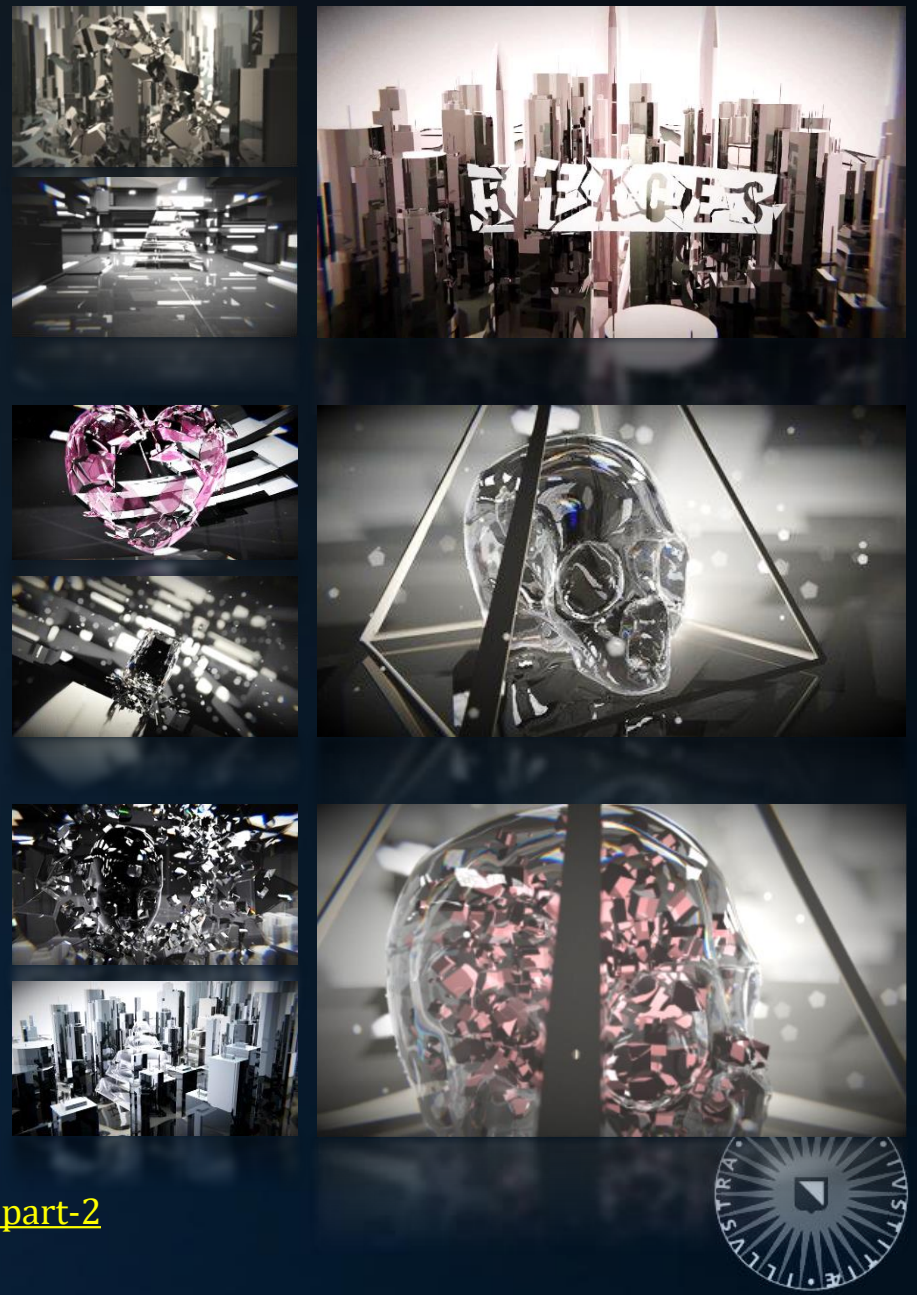
Pragmatic GPU Ray Tracing*

Context:

- Real-time demo
- 50-100k triangles
- Fully dynamic scene
- Fully dynamic camera (no time to converge)
- Must “look good” (as opposed to “be correct”)

➔ Rasterize primary hit
➔ No BVH / kD-tree

➔ Use a grid (or better: sparse voxel octree / brickmap).



*: Real-time Ray Tracing Part 2 – Smash / Fairlight, Revision 2013
https://directtovideo.wordpress.com/2013/05/08/real-time-ray-tracing-part-2

5 Faces

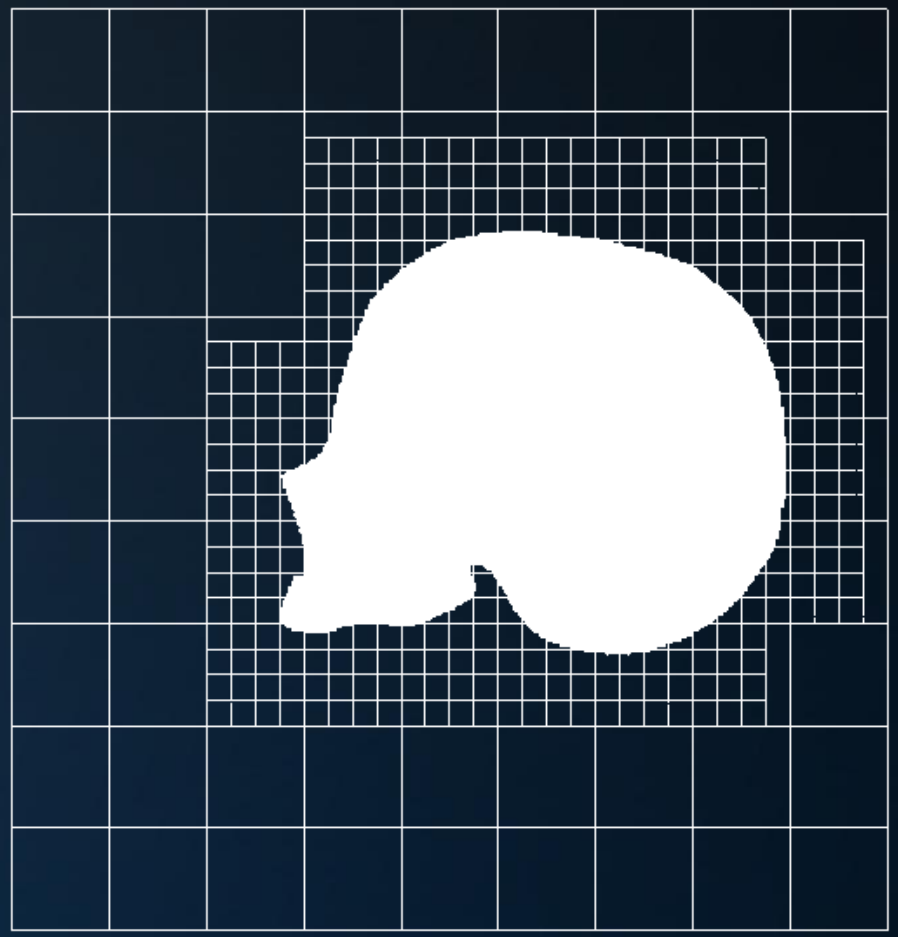
2013

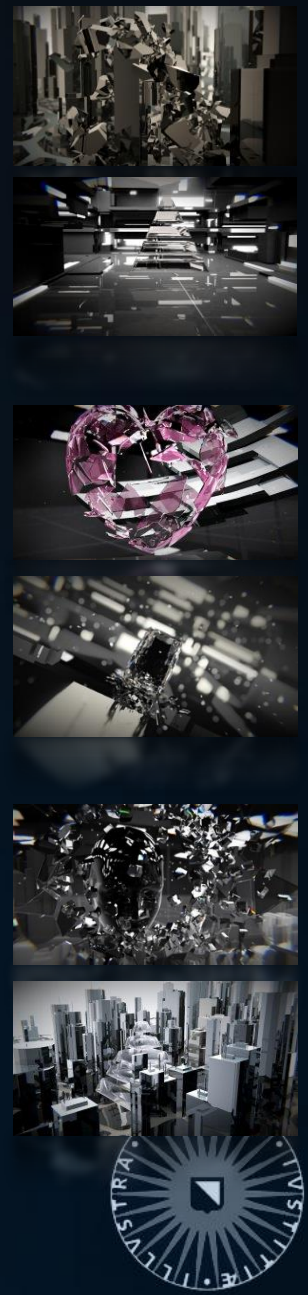
Pragmatic GPU Ray Tracing

Grid traversal: 3D-DDA

Brickmap traversal:

- build in linear time
- locate ray origins in constant time
- skip some open space
- little flow divergence in shader
- simple thread state

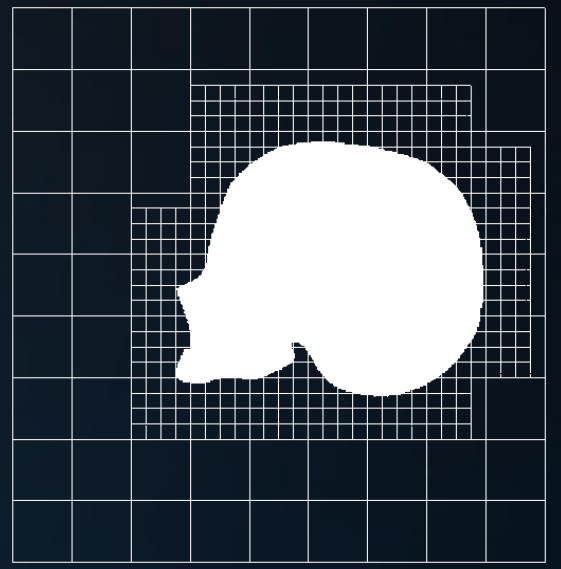




5 Faces

2013

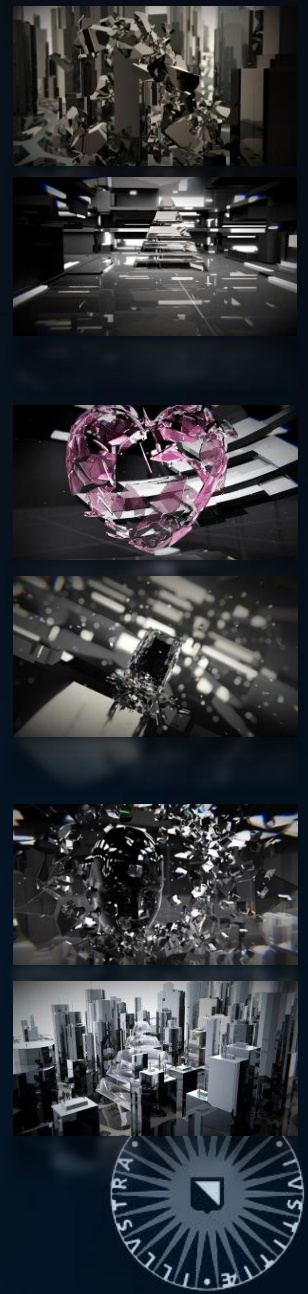
Pragmatic GPU Ray Tracing



Filling the grid: *using rasterization hardware*.
➔ Determine which voxels a triangle overlaps.

Algorithm:

1. Determine for which plane (xy, yz, xz) the triangle has the greatest projected area.
2. Rasterize to that face; use interpolated x, y and depth to determine voxel coordinate.
3. Use conservative rasterization*, **.



*: GPU Gems 2, chapter 42: Conservative Rasterization. Hasselgren et al., 2005.
http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter42.html
**: The Basics of GPU Voxelization, Masaya Takeshige, 2015.
https://developer.nvidia.com/content/basics-gpu-voxelization

5 Faces

2013

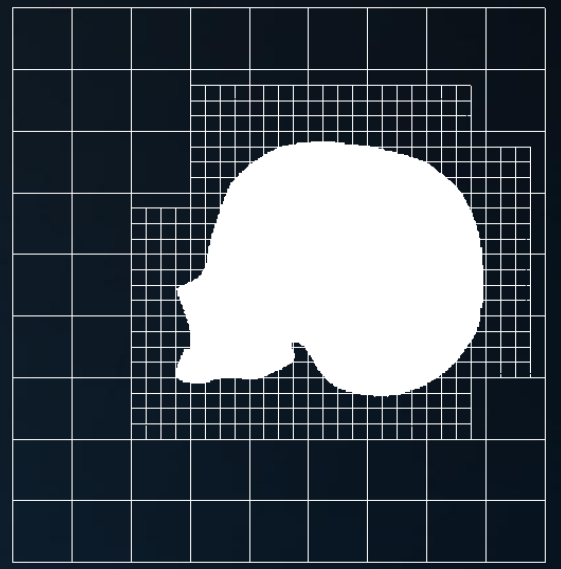
Pragmatic GPU Ray Tracing

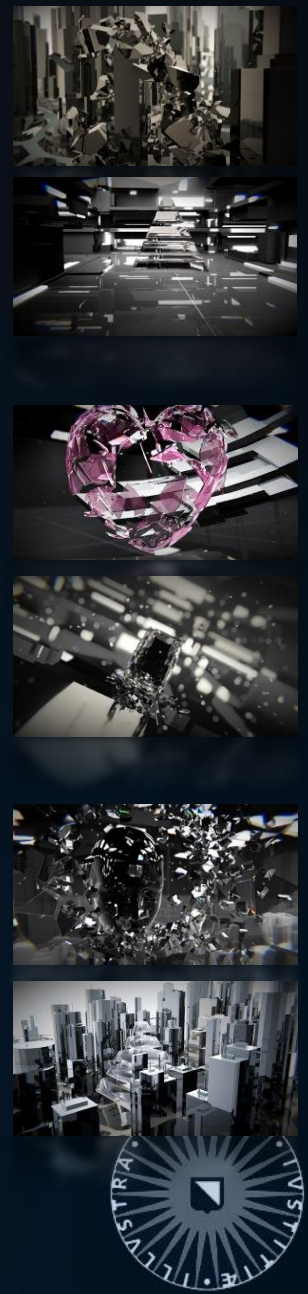
In this case, we are not building a voxel set, but a grid with pointers to the original triangles.

➔ Add each triangle to a pre-allocated list per node.

From grid to brickmap:

- each brick consists of a small grid, e.g. 4x4x4.
- repeat the rasterization process at the higher resolution
- assign each triangle to cells in the fine grid.

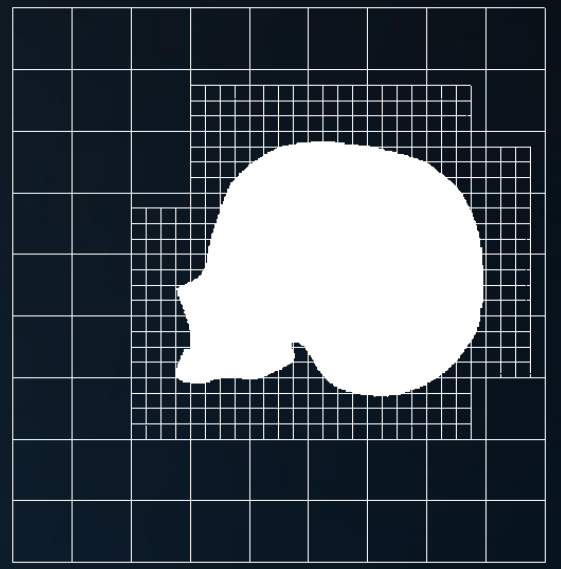




5 Faces

2013

Pragmatic GPU Ray Tracing

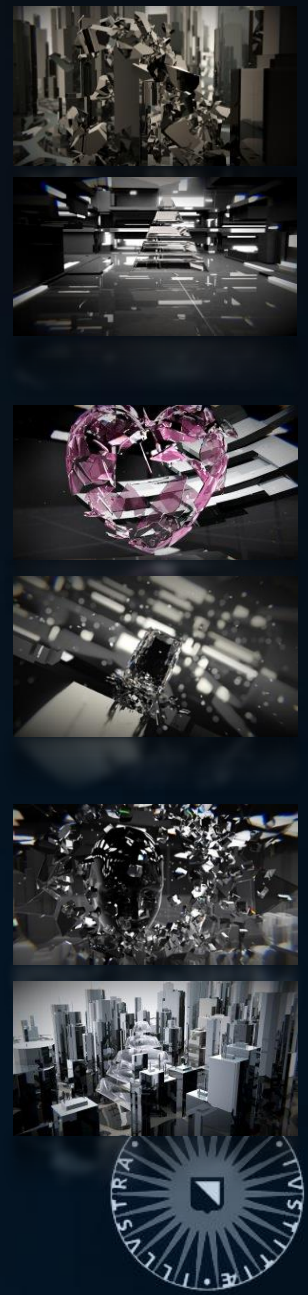


Pragmatic traversal:

- ‘Trace’ primary ray using rasterization
- Determine secondary ray origin from G-buffer

After this:

- Put a maximum on the number of traversal steps, regardless of bounce depth.



5 Faces

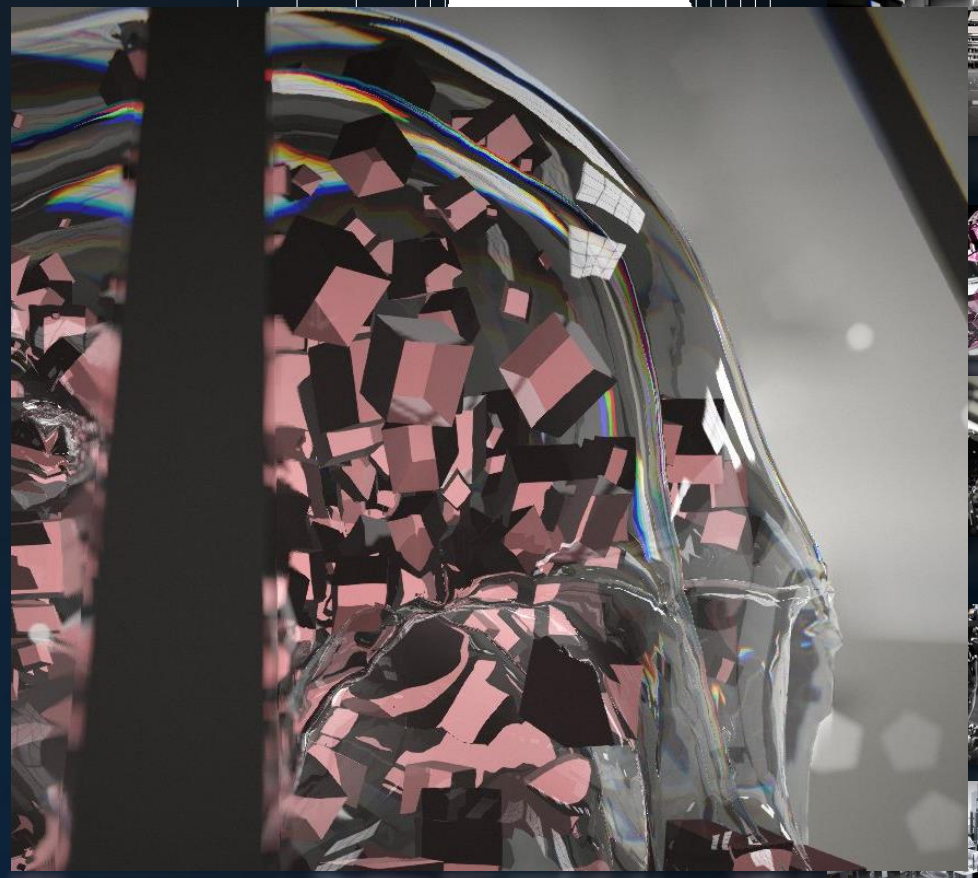
2013

Pragmatic GPU Ray Tracing

Pragmatic diffraction:

Each ray represents 3 'wavelengths', and each results in a different refracted direction. However, only the direction of the first ray is actually used to find the next intersection for the triplet.

EXCEPT: when the rays exit the scene and returns a skybox color; only then the three directions are used to fetch 3 skybox colors which are then blended.



5 Faces

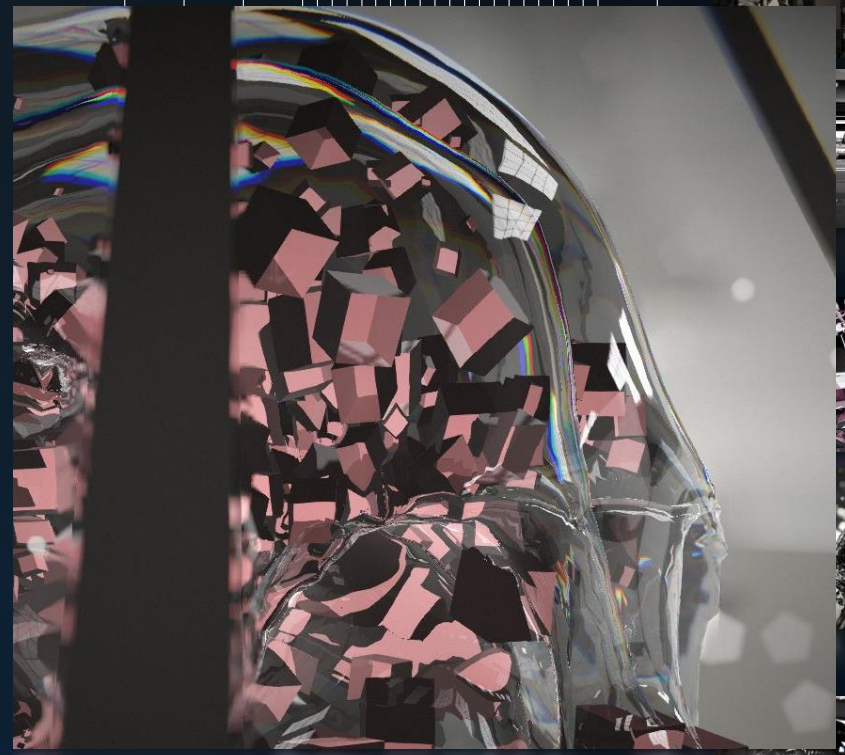
2013

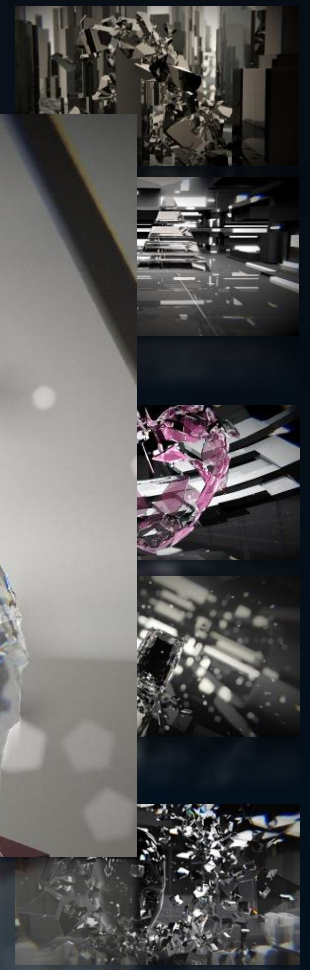
Pragmatic GPU Ray Tracing

Pragmatic depth of field:

Since primary rays are rasterized, the camera used is a pinhole camera.

Depth of field with bokeh is simulated using a postprocess.





See for a practical approach:

Bokeh depth of field – going insane! part 1, Bart Wroński, 2014,
http://bartwronski.com/2014/04/07/bokeh-depth-of-field-going-insane-part-1

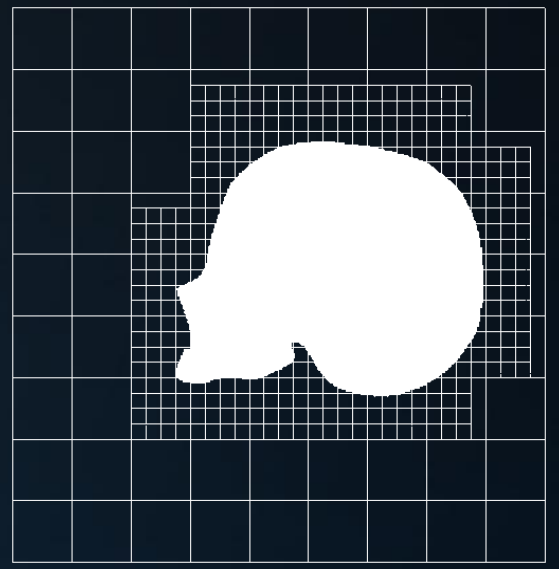
5 Faces

2013

Pragmatic GPU Ray Tracing

Limitations:

- Doesn't work well for 'teapot in a stadium'
- Not suitable for very large scenes (area)
- Manual parameter tweaking



➔ The method is not good for a general-purpose ray tracer, but really clever for a special purpose renderer.

➔ Performance is very good, although hard to estimate:
Demo runs @ 60fps on a high-end GPU;
Traces ~1M primary rays;
Most rays make several bounces (very divergent!);
Guestimate: ~250M rays per second for a fully dynamic scene.

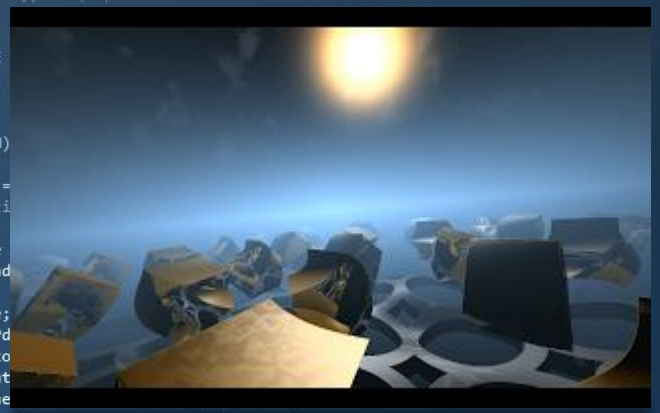
5 Faces

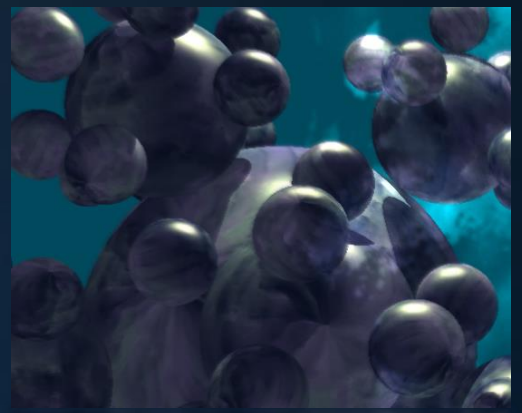
2013

Other Real-time Ray Tracing Demos

For a brief history, see these links:
http://datunnel.blogspot.nl/2009/12/history-of-realtime-raytracing-part-1.html
http://datunnel.blogspot.nl/2009/12/history-of-realtime-raytracing-part-2.html
http://datunnel.blogspot.nl/2009/12/history-of-realtime-raytracing-part-3.html

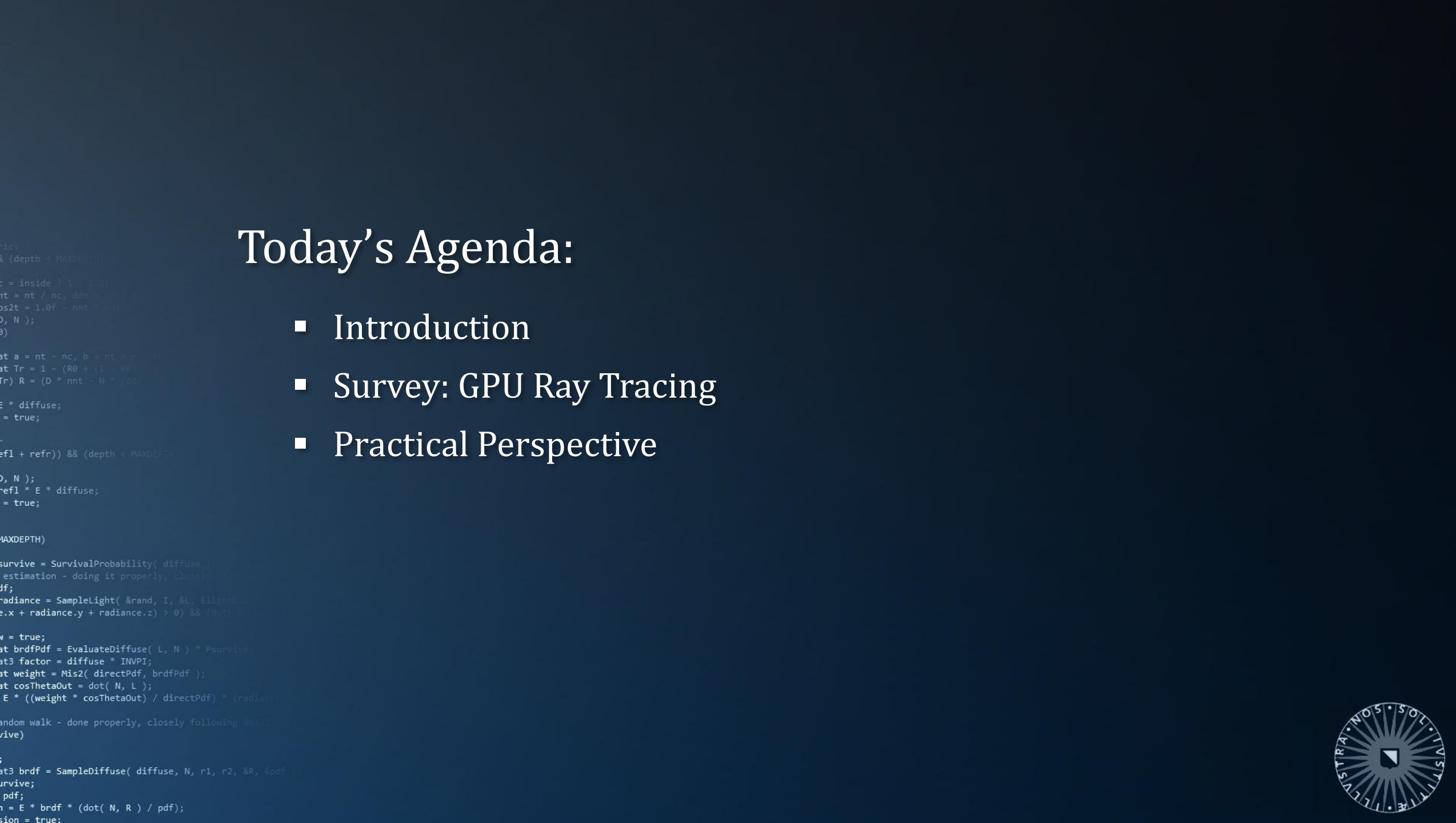
Also check here: http://mpierce.pie2k.com/pages/108.php






Today’s Agenda:

- Introduction
- Survey: GPU Ray Tracing
- Practical Perspective



Next Time

Coming Soon in Advanced Graphics

GPU Ray Tracing Part 2:

- State of the art BVH traversal by Aila and Laine;
- Wavefront Path Tracing
- Heterogeneous Path Tracing: Brigade.

INFOMAGR – Advanced Graphics

Jacco Bikker - November 2022 - February 2023

END of “GPU Ray Tracing (1)”

next lecture: “Variance Reduction”