

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 1.0f - nt)
    {
        nt = nt / nc; ddn = ddn * nc;
        rands2t = 1.0f - nnt * nnt;
        D, N );
    }
    else
    {
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (R0 + (1 - R0) * rands2t);
        (Tr) R = (D * nnt - N * (ddn * rands2t));
    }
    E * diffuse;
    = true;
    refrl + refr) && (depth < MAXDEPTH)
    {
        D, N );
        refrl * E * diffuse;
        = true;
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, R, Align );
    e.x + radiance.y + radiance.z );
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following Small's
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}
```

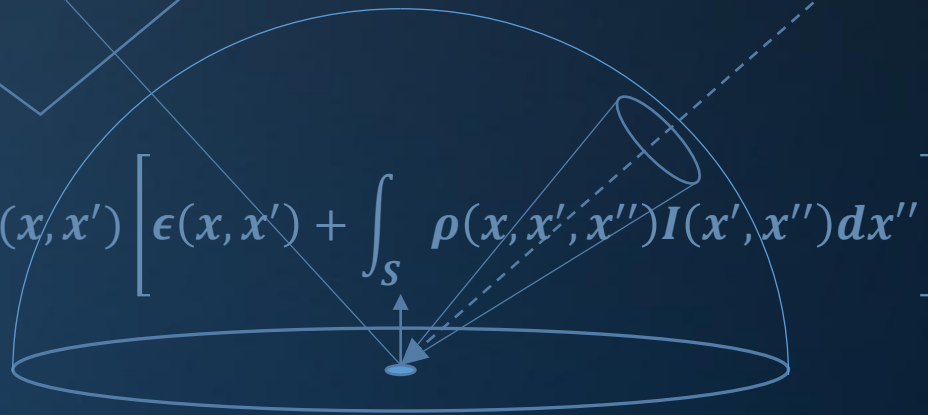


INFOMAGR – Advanced Graphics

Jacco Bikker - November 2022 - February 2023

Lecture 7 - “GPU Ray Tracing (1)”

Welcome!



Today's Agenda:

- Introduction
- Survey: GPU Ray Tracing
- Practical Perspective



Introduction

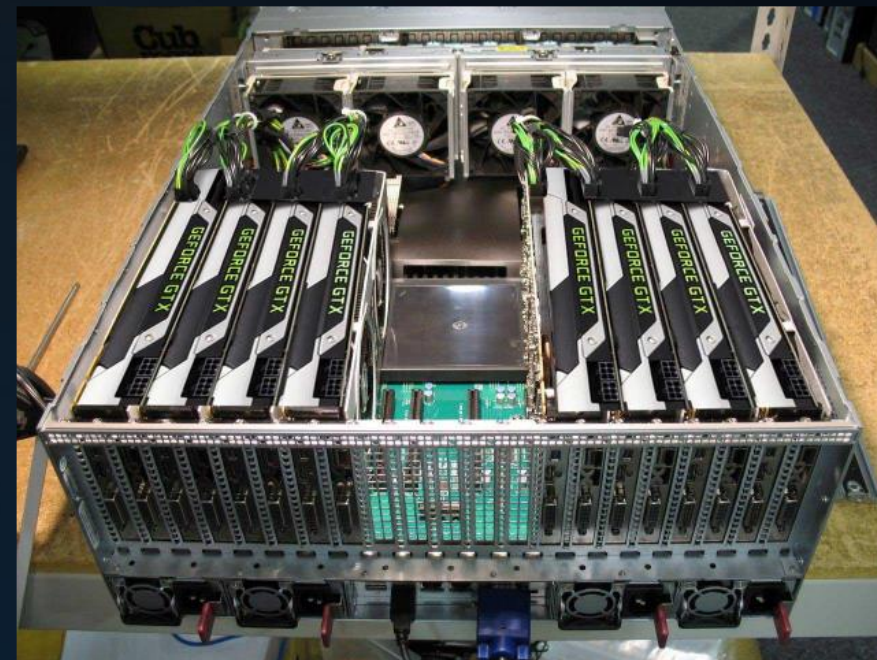
Transferring Ray Tracing to the GPU

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

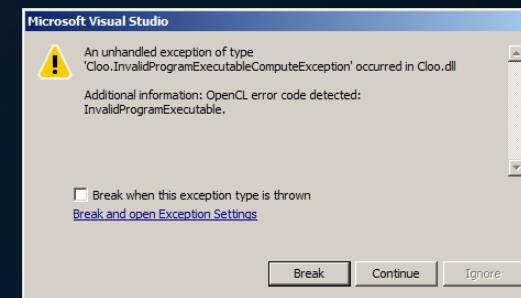
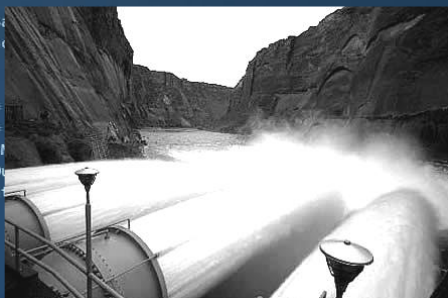
Survey

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

```

ics
& (depth < MAXDEPTH)
{
    if (inside & !isRefr)
    {
        nt = nt / nc; ddn = dot(N, R);
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
    else
    {
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (R0 + (1 - R0) * ddn);
        Tr) R = (D * nnt - N * (ddn > 0 ? 1 : -1));
    }
    E * diffuse;
    = true;
    {
        refl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, close);
    if;
    radiance = S;
    e.x + radiance;
    w = true;
    at brdfPdf =
    at3 factor =
    at weight =
    at cosTheta0
    E * ((weight
    random walk -
    ve)
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, pdf);
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



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:

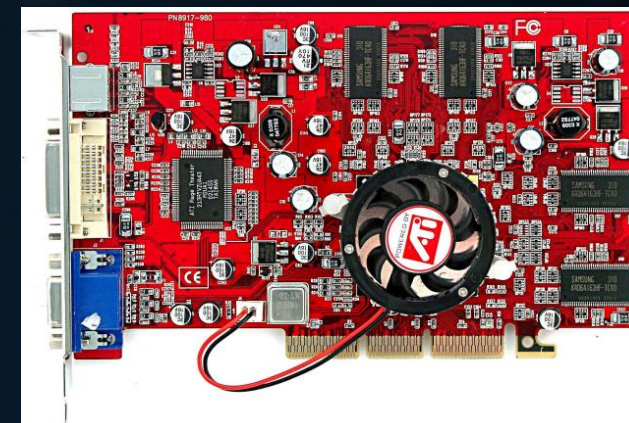
- Floating point fragment shaders
- Improved instruction sets
- Multiple outputs per fragment shader

No branching

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



NVidia GeForce 3



ATi Radeon 8500



Survey

2002

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = dot(N, D);
        float r = 1.0f - nnt * ddn;
        float D, N );
    }
    else
    {
        at a = nt - nc, b = nt + nc;
        float Tr = 1 - (R0 + (1 - R0) * r);
        float R = (D * nnt - N * (ddn * nnt - 1));
        E * diffuse;
        = true;
        refl + refr)) && (depth < MAXDEPTH)
        {
            D, N );
            refl * E * diffuse;
            = true;
        }
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &align );
    e.x + radiance.y + radiance.z ) > 0) && (depth <
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following SampleLight
    survive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
    
```

Ray Tracing on Programmable Graphics Hardware

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

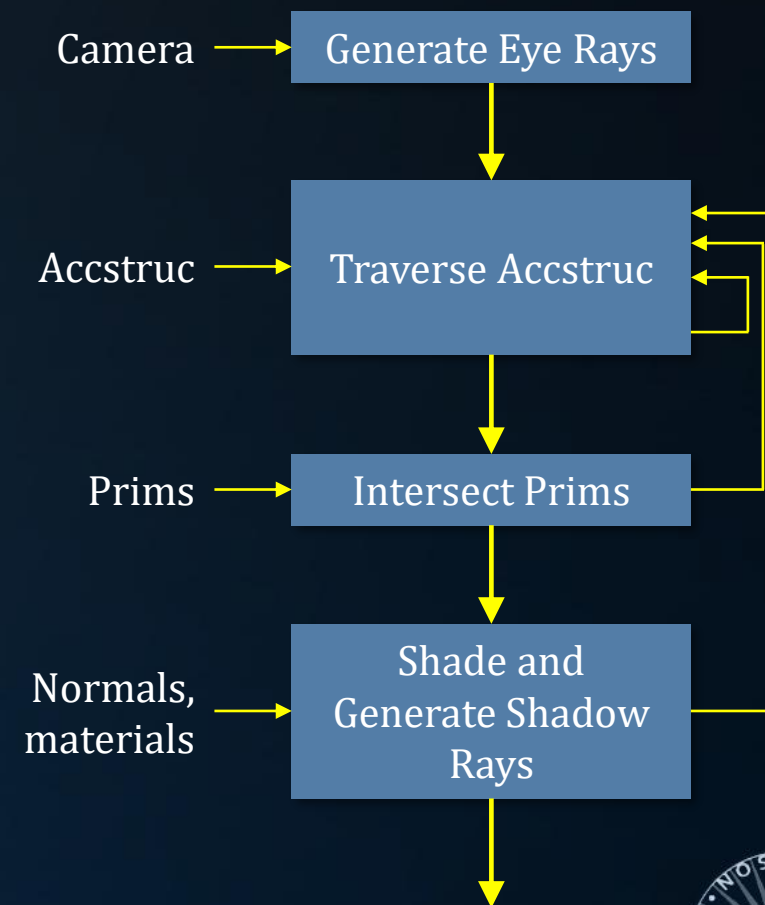
Stage 1: Produce a stream of primary rays.

(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

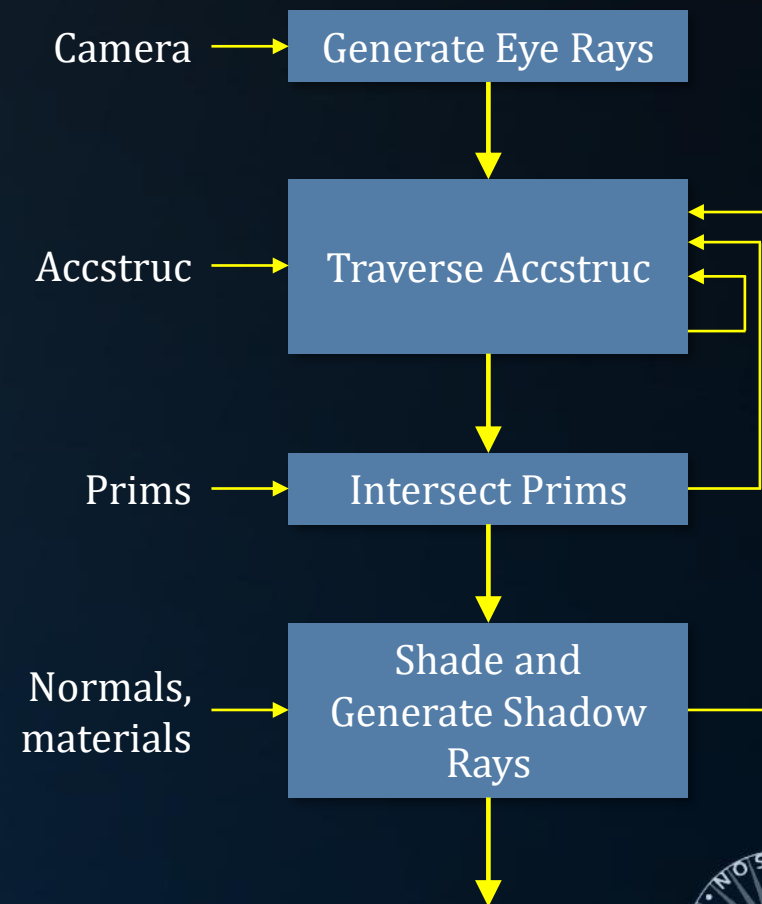
Stream computing without flow control:

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.

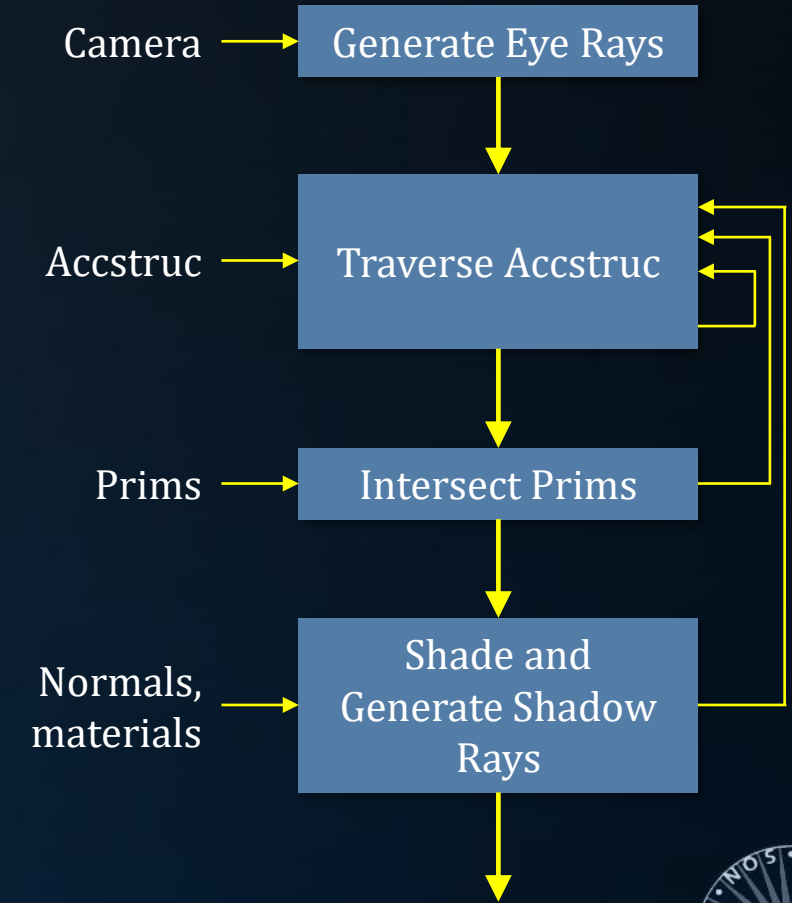
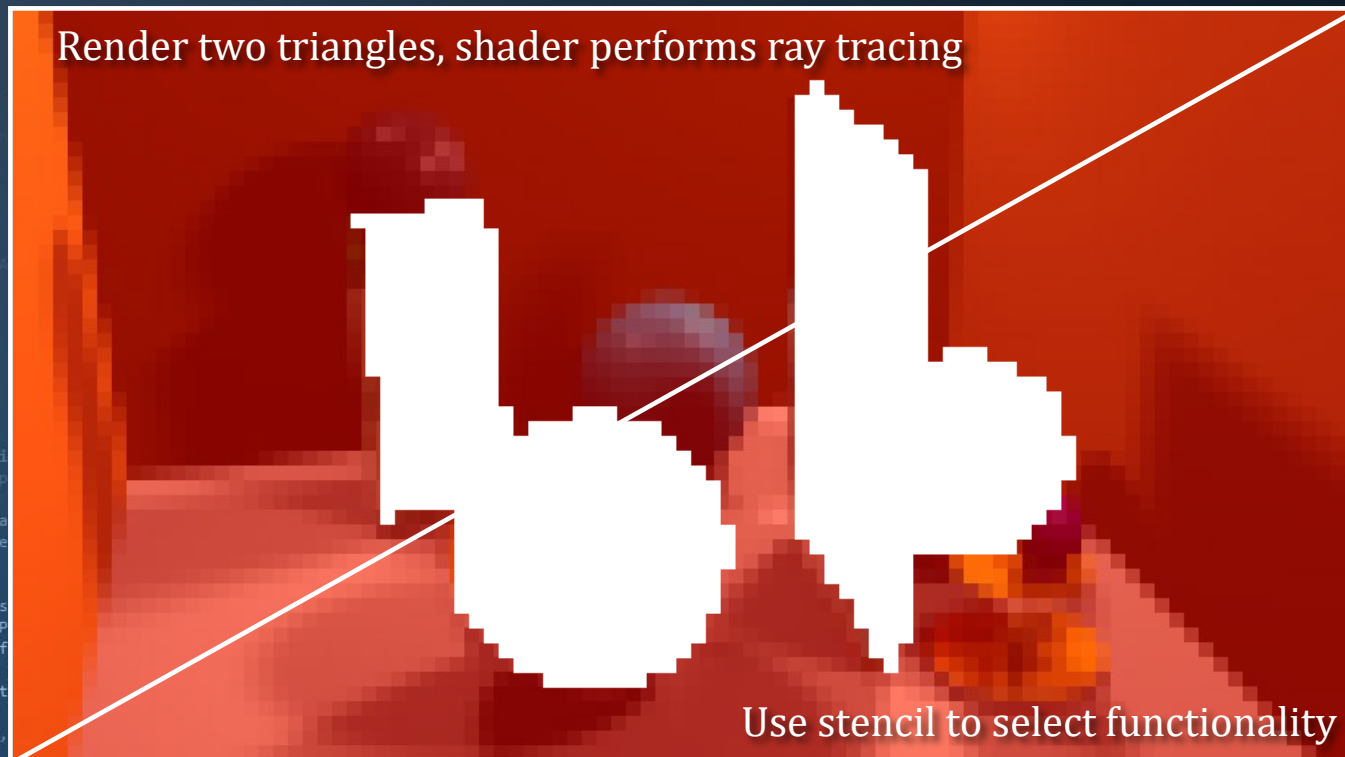


Survey

2002

Ray Tracing on Programmable Graphics Hardware

Stream computing without flow control:



Survey

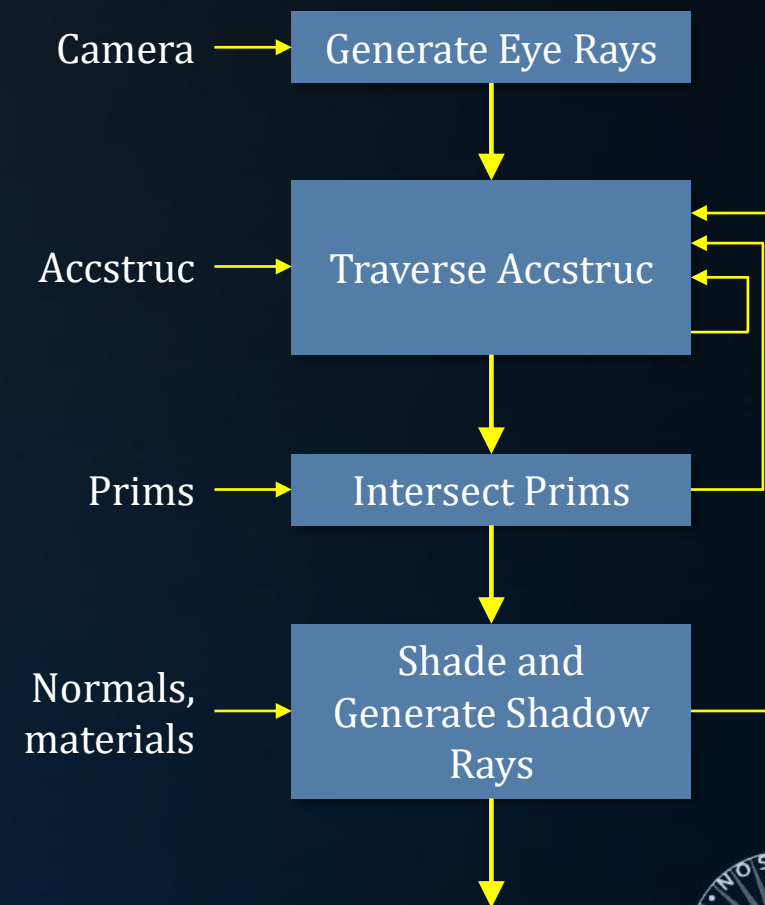
2002

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*.



```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * nc;
        cos2t = 1.0f - nnt * nnt;
        D, N );
    }
}

at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) *
Tr) R = (D * nnt - N * (ddn *
    E * diffuse;
    = true;
    -
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &light;
    e.x + radiance.y + radiance.z > 0) && (depth <
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following Shell's
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2 * R0;
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
    
```

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



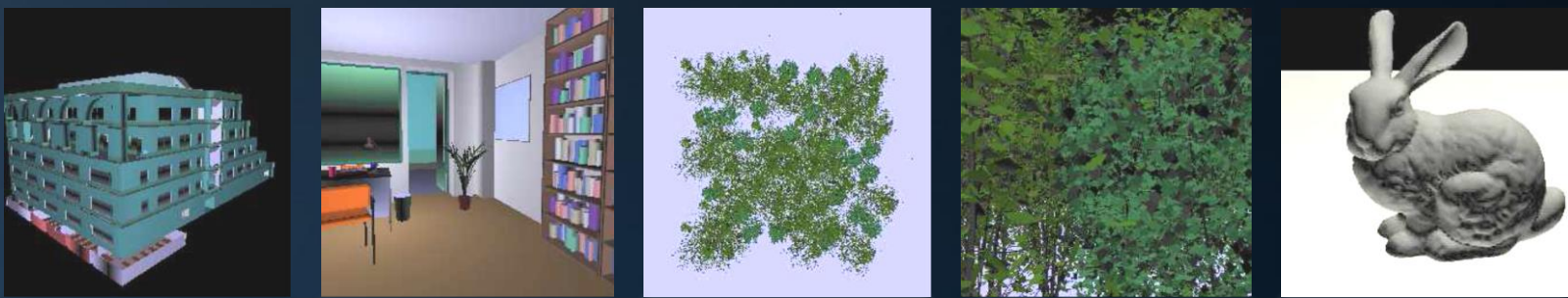
Survey

2002

Ray Tracing on Programmable Graphics Hardware

Results

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = dot(N, N);
        cos2t = 1.0f - nnt * ddn;
        D, N );
        0);
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (R0 + (1 - R0) *
        Tr) R = (D * nnt - N * (ddn
        E * diffuse;
        = true;
        -
        refl + refr)) && (depth < MAXDEPTH)
        D, N );
        -refl * E * diffuse;
        = true;
        MAXDEPTH)
        survive = SurvivalProbability
        estimation - doing it properly,
        df;
        radiance = SampleLight( &rand, I, &L, &lightP
        e.x + radiance.y + radiance.z) > 0) && (depth <
        w = true;
        at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        at3 factor = diffuse * INVPI;
        at weight = Mis2( directPdf, brdfPdf );
        at cosThetaOut = dot( N, L );
        E * ((weight * cosThetaOut) / directPdf) * (radiance
        random walk - done properly, closely following Small's
        vive)
        ;
        at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
        survive;
        pdf;
        n = E * brdf * (dot( N, R ) / pdf);
        sion = true;
```



passes
efficiency

2443	1198	1999	2835	1085
0.009	0.061	0.062	0.062	0.105

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



2002

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 Raytracer

Tim Foley and Jeremy Sugerman [†]

Stanford University

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

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors I.3.1 [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 scenes with nonuniform distributions of geometry.

The relative performance of different acceleration structures has been widely studied. Havran [Hav00] compared a large number of acceleration structures across a variety of scenes and determines that the kd-tree is the best general-purpose acceleration structure for CPU raytracers. GPU raytracers seem natural, therefore, to try to use a kd-tree to accelerate raytracing. As we will describe in section 3, the standard algorithm for kd-tree traversal relies on a ray dynamic stack. Ernst et al. [EVG04] demonstrate that this data structure can be built on the GPU, and in fact a stack-based kd-tree traversal. However, their approach requires storage proportional to the maximum stack size multiplied by the number of rays, which may limit the number of rays that can be traced in parallel. Also, parallel operation the stack requires additional render passes with a ray operation.

Our work instead presents kd-tree traversal algorithms, *kd-restart* and *kd-backtrack* that run without a stack. We show that these new algorithms maintain the exponential performance of kd-tree traversal. We also present a GPU raytracer that incorporates these algorithms and demonstrates that, as on CPUs, they outperform uniform grid acceleration structures with scenes of sufficient complexity. Finally,

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

Survey

2005

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * nc;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) * cos2t);
    Tr) R = (D * nnt - N * (ddn * cos2t));
    E * diffuse;
    = true;
    refl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse, I, &L, &align);
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &align);
    e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
    {
        w = true;
        at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        at3 factor = diffuse * INVPI;
        at weight = Mis2( directPdf, brdfPdf );
        at cosThetaOut = dot( N, L );
        E * ((weight * cosThetaOut) / directPdf) * (radiance
    }
    random walk - done properly, closely following SurvivalProbability
    survive)
    {
        at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
        survive;
        pdf;
        n = E * brdf * (dot( N, R ) / pdf);
        ion = true;
    }
}

```

KD-Tree Acceleration Structures for a GPU Ray Tracer

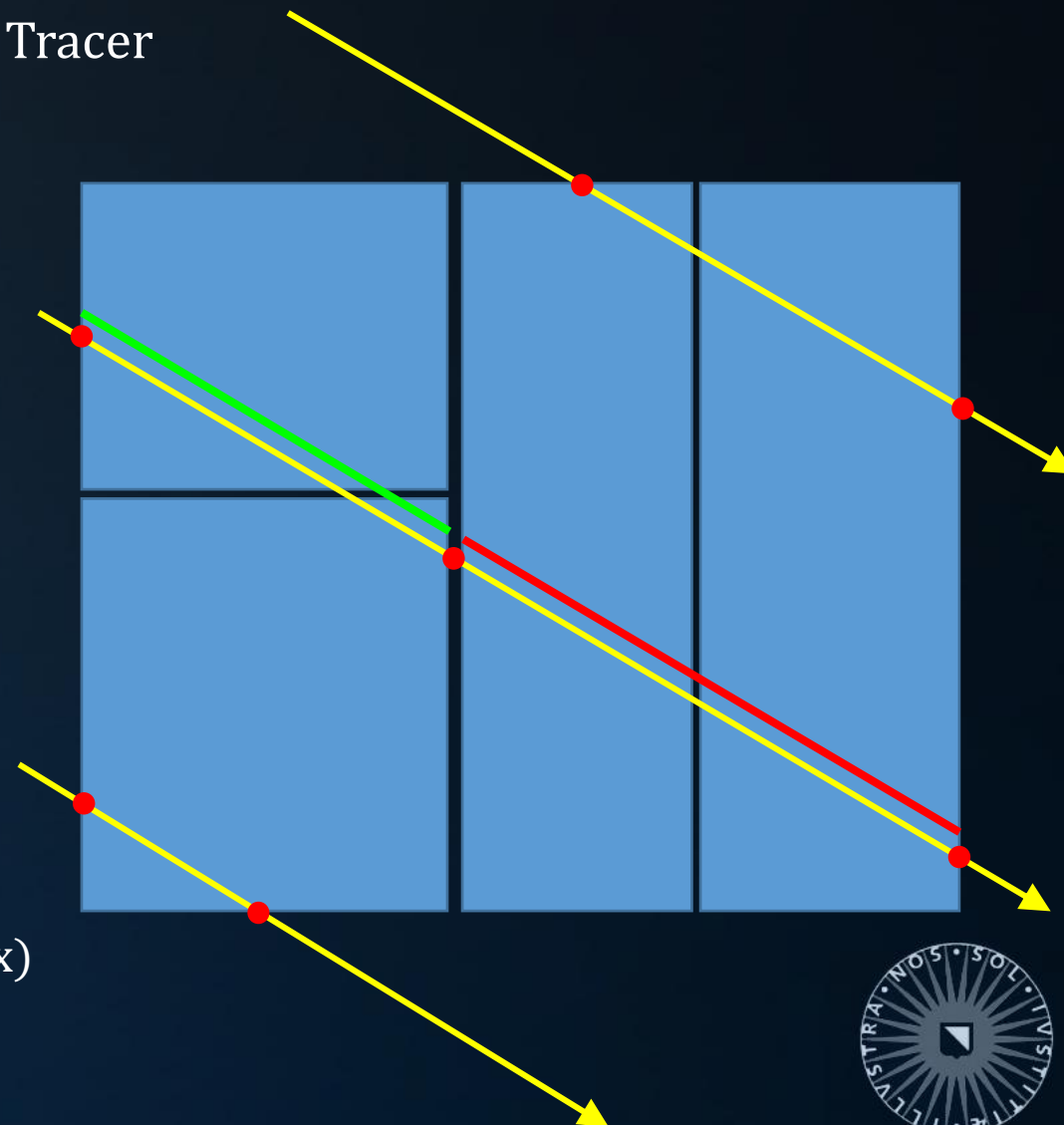
Recall standard kD-tree traversal:

Setup:

1. $t_{max}, t_{min} = \text{intersect}(\text{ray}, \text{root bounds})$;

Root node:

2. Find intersection t with split plane
3. If $t_{min} \leq t \leq t_{max}$:
 - Process near child with segment (t_{min}, t)
 - Process far child with segment (t, t_{max})
4. else if $t > t_{max}$:
 - Process left child with segment (t_{min}, t_{max})
5. else
 - Process right child with segment (t_{min}, t_{max})



Survey

2005

KD-Tree Acceleration Structures for a GPU Ray Tracer

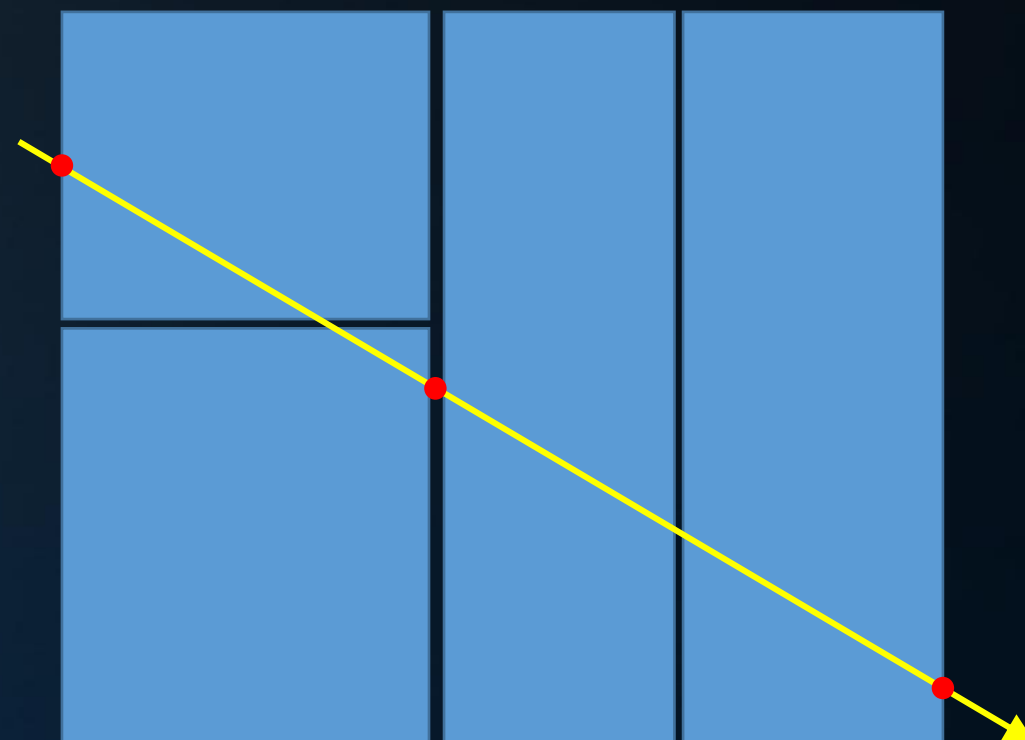
Recall standard kD-tree traversal:

Setup:

1. $t_{\max}, t_{\min} = \text{intersect}(\text{ray}, \text{root bounds})$;

Root node:

2. Find intersection t with split plane
3. If $t_{\min} \leq t \leq t_{\max}$:
 - **Push far child**
 - **Continue with near child**
4. else if $t > t_{\max}$:
 - Process left child with segment $(t_{\min}, t_{\max})$
5. else
 - Process right child with segment $(t_{\min}, t_{\max})$



Survey

2005

KD-Tree Acceleration Structures for a GPU Ray Tracer

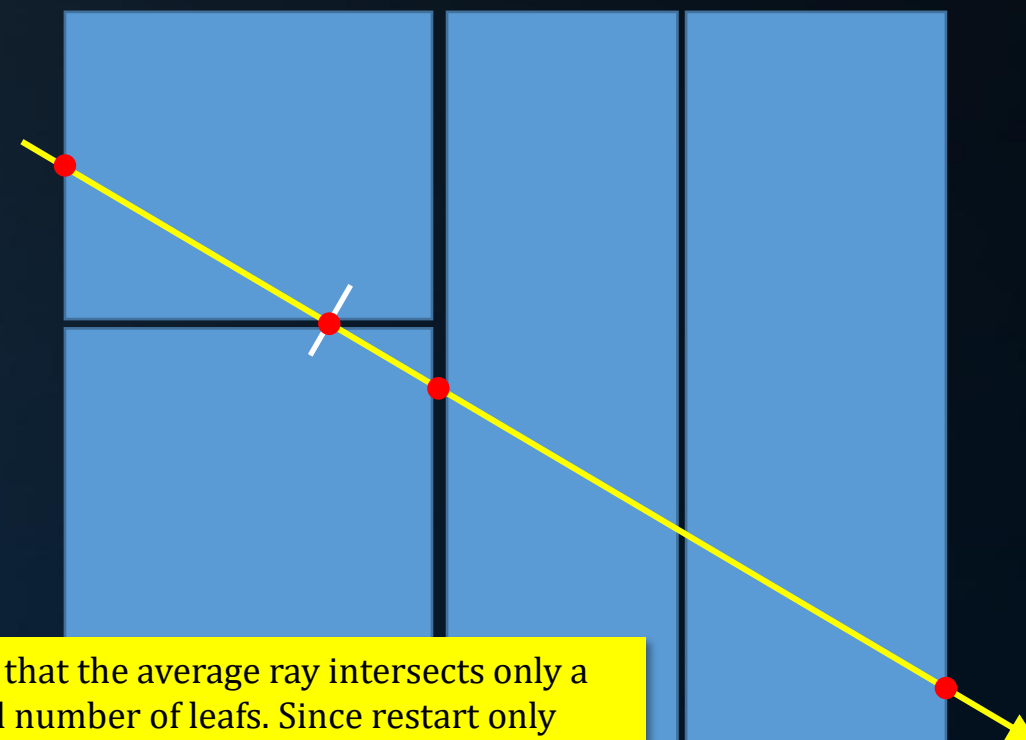
Traversing the tree without a stack:

If we always pick the nearest child, the only value that will change is t_{max} .

Setup:

1. $t_{max}, t_{min} = \text{intersect}(\text{ray}, \text{root bounds})$;
2. Always pick the nearest child.
3. Once we have processed a leaf, restart with:
 - $t_{min} = t_{max}$
 - $t_{max} = \text{intersect}(\text{ray}, \text{root bounds})$

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*.

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * nc;
        pos2t = 1.0f - nnt * ddn;
        D, N );
    }
}

at a = nt - nc; b = nt * nc;
at Tr = 1 - (R0 + (1 - R0) * a);
Tr) R = (D * nnt - N * (ddn * a));

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
{
    D, N );
    refrl * E * diffuse;
    = true;
}

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &lightPos );
e.x + radiance.y + radiance.z) > 0) && (dot( N, L ) > 0)
{
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance.x + radiance.y + radiance.z);
}

random walk - done properly, closely following Small's
vive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



Survey

2005

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * nt;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (ddn
    E * diffuse;
    = true;
    efl + refr)) && (depth < MAXDEPTH)
    D, N );
    efl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse,
    estimation - doing it properly, closely
    if;
    radiance = SampleLight( &rand, I, &L, &light;
    e.x + radiance.y + radiance.z) > 0) && (depth <
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following Small's
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
    
```

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.

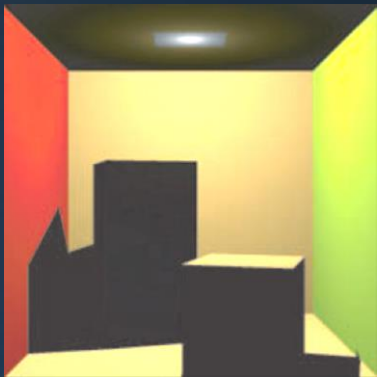


Survey

2005

KD-Tree Acceleration Structures for a GPU Ray Tracer

Results (ms*):



brute force
grid
kd-restart
kd-backtrack

23
63
80
84

4620
357
701
690

4770
8344
968
946

7350
2687
992
857

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



2007

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
- **₂: Stackless KD-tree traversal for high performance GPU ray tracing, Popov et al., 2007

******: Stackless KD-tree traversal for high performance GPU ray tracing, Popov et al., 2007

Over the past
high number
can explore
CPU load
from making
using more
introduction
techniques
the most, as
converting to
12-18 sec.

CR Category
Architectural
and Structural

1. Intro

* plantar

Universal attention should be given to the 50 demonstrated prototype systems to ensure that this is not viewed as a

Deployment without an first response

that we go
d from giv
China the
fragment of
the GPU, I
increased to
veloped a f
ray interme
and Care of
a replacement
the previous
tantly out
the world.

3 Algorithm

much greater
also indicate
while case 4
results after
while case
B. 1. 1.

and
 along
 and
 along
 was
 and
 the
 the rest
 along
 of the
 was
 around the



The way
longer late
under a ray
the way is to
under the
from a case

2.2 Short
Toby et al. also reported a high proportion of short rapids. However, of the rapids that were short, only 10% were the only short rapids in the reach.

an empty
triggers a
that can b
various lea
short, ab
more valu
like witho

We add three major measurements to those of earlier algorithms: parallelization, push-reduce, and short-circuit. Parallelization combines CPU and GPU engines (Wald et al. 2003) with contract, and takes advantage of the GPU's higher off-chip memory than on-chip (push-reduce), push-reduce, broadcast, and the technique of using contract instead of push-reduce. From the tree root, short-circuit adds a second, final state stack to maintain the last N element, and falls back to broadcast on unfindable. The net impact is more than an order of magnitude performance improvement over Foley et al. and a 10-fold performance improvement over our own earlier and a 25-fold improvement over the best conventional stack-based approach.

[illegible]

There has been a significant interest in studying systems using parallel architectures. Boudrias et al. (2006) designed

while an *adversary* formula to hide the identity on a node basis. Their experiment performs nearly as well per core as per clock as a commodity x86, resulting in 40 million instructions per second and 21 million primary plus shadow rays per second on the conference scene. This gives them clock-to-clock a 3.6x performance advantage over a single core x86. Sugrue et al. (2008) implemented a full tree raytracer on the Cell with a software managed cache, but without feature culling or software threads. This version achieved 17.4 million primary rays per second on the Helix test scene, the same performance as the CPU reference without loss.

[Dehennin et al., 2002]. This study has some limitations: it is

As explained in previous work by Foley et al. (2003), the stack requirement of the standard list tree algorithm depends poorly on CPUs. The recently added ability for CPUs to perform disjunction within a currency system as caching need a vastly slower than needed 4 times on a CPU. Here if a stack were fast enough, the resource requirement would be impractical, enough memory for the deepest $\text{exp}(\text{size multiplied by the total number of eggs traversing})$ is possible. It is possible to create a subset of the eggs at the time and create a stack buffer, but CPUs only can afford with the thousands of eggs traversing at once.

signature assignment function, but requires only a single time-step. The algorithm is an *in situ* word boundary task, and is similar to the single word addition, deletion, and correspondence (ADD, DEL, and COR) models described in [1]. The algorithm is implemented as a neural network with one hidden layer and one output layer. The output layer has two nodes, one for the addition and one for the deletion. On the output, the *ADD* node is always active, and the *DEL* node is active when the current time-step is a word boundary. The algorithm is implemented as a neural network with one hidden layer and one output layer. The output layer has two nodes, one for the addition and one for the deletion. On the output, the *ADD* node is always active, and the *DEL* node is active when the current time-step is a word boundary. The algorithm is implemented as a neural network with one hidden layer and one output layer. The output layer has two nodes, one for the addition and one for the deletion. On the output, the *ADD* node is always active, and the *DEL* node is active when the current time-step is a word boundary.

3.1 Packets

One of the most significant innovations in interactive ray tracing was the introduction of packets (Wald et al. 2001). In these simplest form, packets trace four rays at a time to take advantage of the four-order SIMD instructions on modern CPUs. Beyond that, however, adjusting the packet size allows the programmer to trade-off accelerating the cost of

One final constraint modification is the introduction of a *gender* effect. The significant composition-by-gender interaction indicates that most respondents are female, so focus on the modifying effect when individuals are so inclined. To state the my forward at the current point, we set the new *W* to be the previous step's *W* plus *W*. With gender, we add a *W* term to each *my* so that the mask is always added based on whether that *my* should remove a given child. Only one outstanding is the *W* term obtained from *W* by

3.2 Push-Down

Often the intersection of a ray with the scene volume can pass through a subset of the entire k-d tree. Instead of intersecting at the root, such rays only need to back up to the node that is the root of the lowest subtree containing them. Thus, as a ray descends from the root, as long as it can



The, Robotic, Kitchen, Conference Room.

4 Implementation

We implemented the optimizations from Section 3 in a re-compiler for the ATLASIRK in CPU. While the compiler on one side, it can also optimize and share like a traditional MD CPU application. Our implementation uses the Boost [2006] container and stack. Dmccom 0.0.6, and the CPU kernel program for optimizing are written in Boost to parallelize 32 threads and run with version 0.7 of the ATLASIRK.

We selected the parameters of our implementation to best suit our hardware. The best tree is built using a random number generator (Rosen and Blake 2002), but with an additional

Our systems resemble as follows: we construct a vector buffer and an ivary containing the same geometry, and use

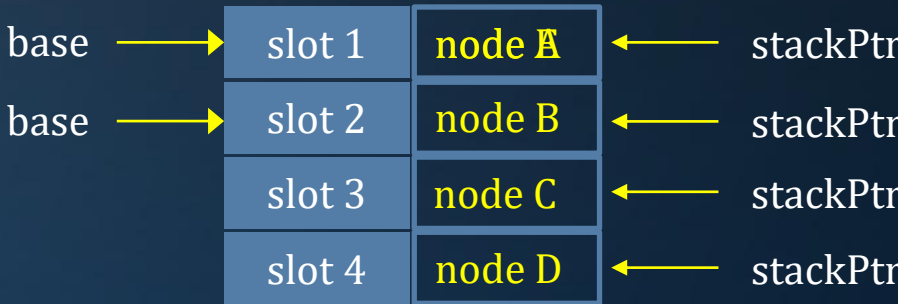
Survey

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

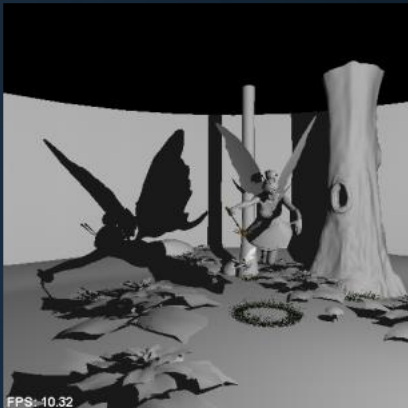
2007

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

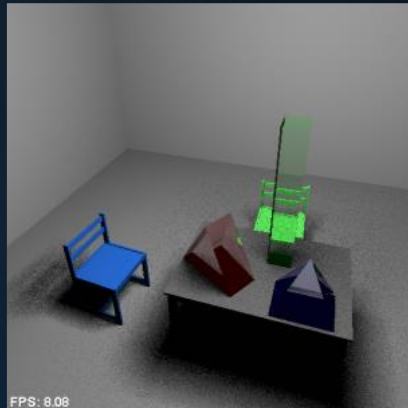
Results*:



FPS: 18.77



FPS: 10.32



FPS: 8.08



FPS: 19.61

GPU
CPU (1 core)

12.7

-

10.6

3.6

36.0

6.6

16.7

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. $1/3^{\text{rd}}$ 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...

```

ics
& (depth < MAXDEPTH)
{
    if ( ! inside ) return 0;
    nt = nt / nc; ddn = dot(N, D);
    cos2t = 1.0f - nnt * ddn;
    if ( cos2t < 0 ) return 0;
    D, N );
    )
    {
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (R0 + (1 - R0) * a);
        (Tr) R = (D * nnt - N * (ddn *
    )
    E * diffuse;
    = true;
    -
    efl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        -refl * E * diffuse;
        = true;
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &lightPdf );
    e.x + radiance.y + radiance.z) > 0) && (dot(N, L) > 0)
    {
        w = true;
        at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        at3 factor = diffuse * INVPI;
        at weight = Mis2( directPdf, brdfPdf );
        at cosThetaOut = dot( N, L );
        E * ((weight * cosThetaOut) / directPdf) * (radiance
    }
    random walk - done properly, closely following Small's
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    ion = true;
}

```



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

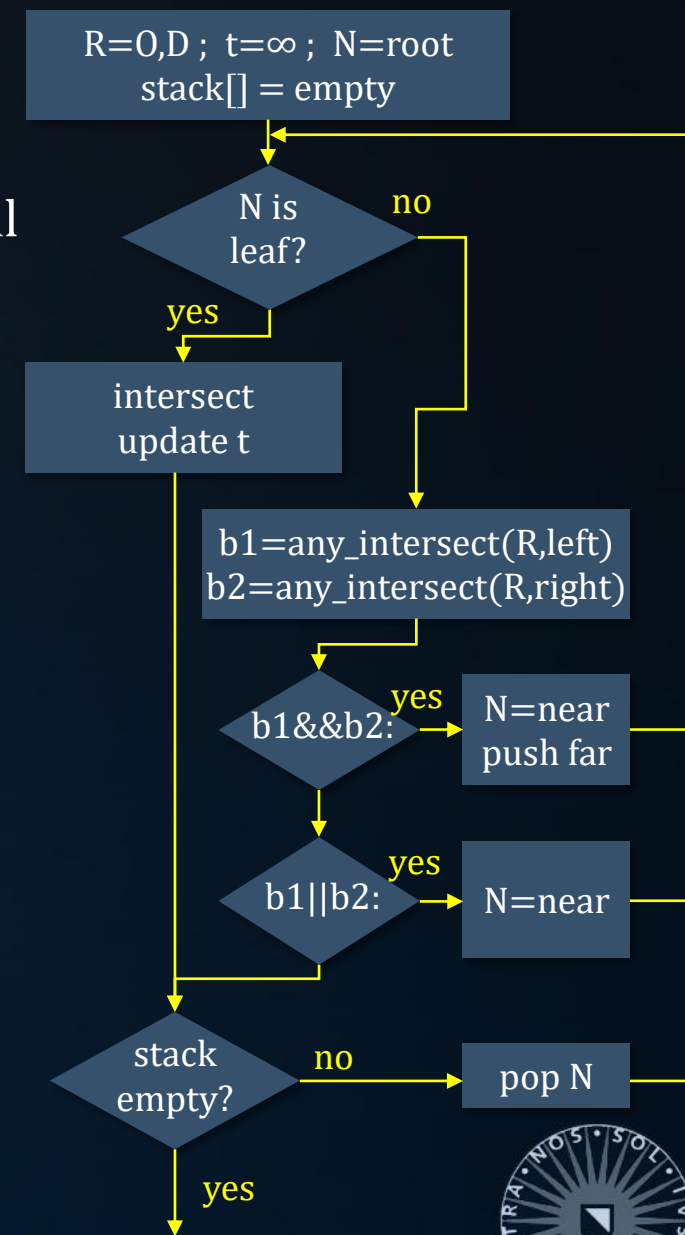
```

ics
& (depth < MAXDEPTH)
{
    if (nt < inside ? 1.f : 0.f)
    {
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
    else
    {
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (R0 + (1 - R0) * ddn);
        Tr) R = (D * nnt - N * (ddn *
    )
    E * diffuse;
    = true;
    -
    refl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &light );
    e.x + radiance.y + radiance.z > 0) && (depth <
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following Small's
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
    
```

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

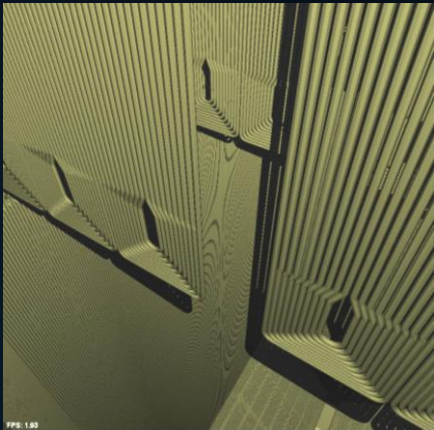


Survey

2007

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

Results*:



primary
+shadow

19.0
6.1

16.2
5.7

6.4
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

```
ics
& (depth < MAXDEPTH)
{
    if (nt < nc)
    {
        nt = inside ? 1.0f : 0.0f;
        nt = nt / nc; ddn = dot(N, D);
        cos2t = 1.0f - nnt * nnt;
        D, N );
    }
    else
    {
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (R0 + (1 - R0) * sqrt(cos2t));
        R = (D * nnt - N * (ddn < 0 ? 1 : -1));
        E * diffuse;
        = true;
    }
    if (refl + refr) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse,
    estimation - doing it properly, close
    df;
    radiance = SampleLight( &rand, I, &L,
    e.x + radiance.y + radiance.z) > 0) &&
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N );
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / direct
    random walk - done properly, closely following
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}
```



5 Faces

2013

Pragmatic GPU Ray Tracing*

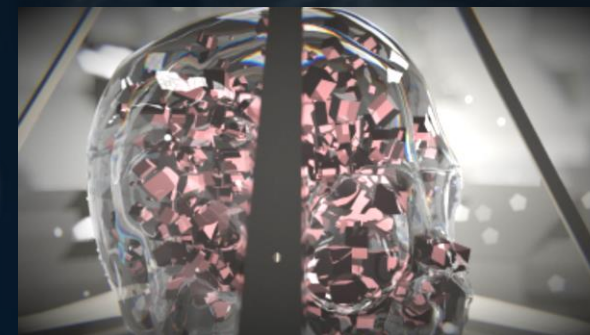
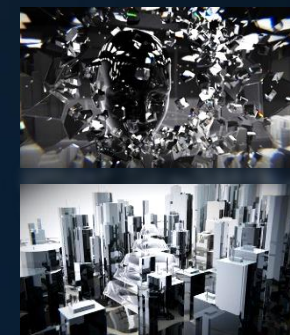
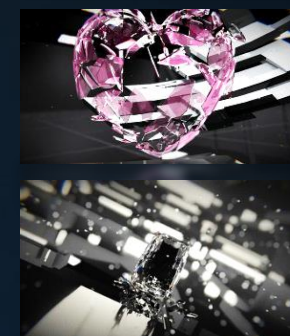
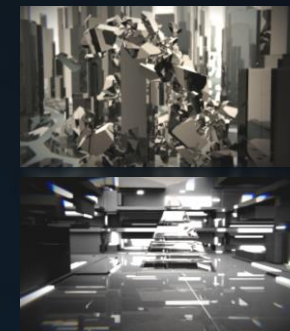
Context:

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

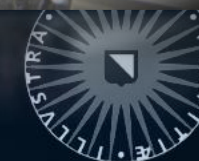
➔ Rasterize primary hit

➔ 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

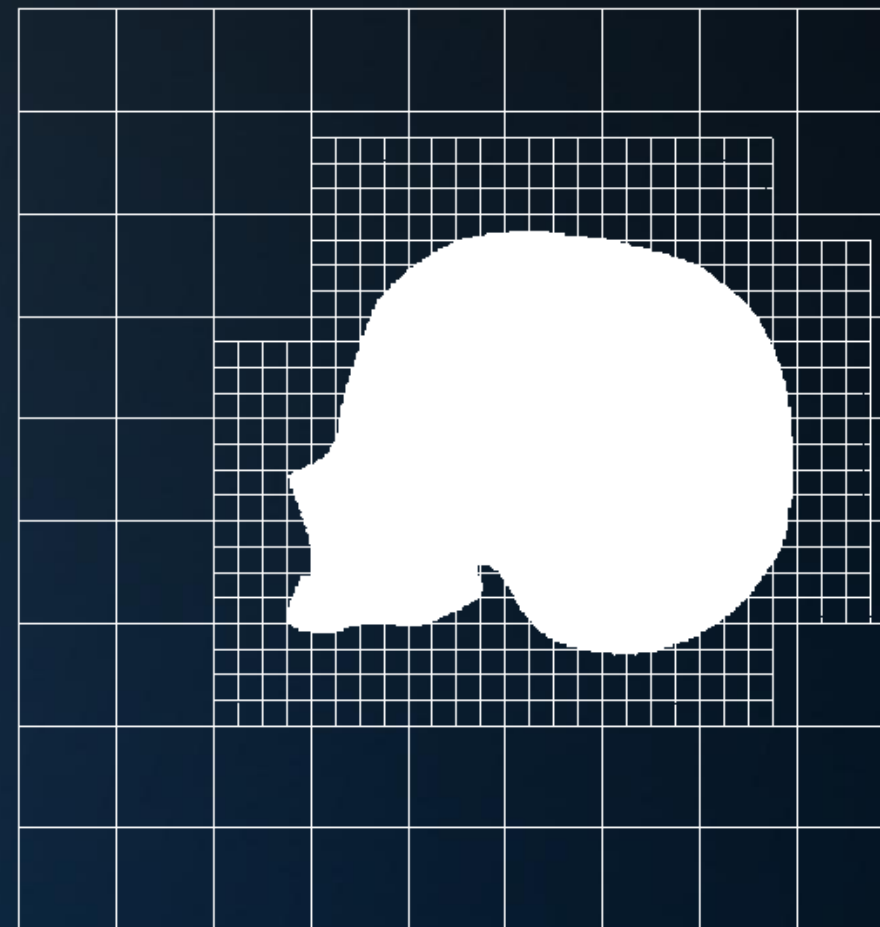
2013

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.

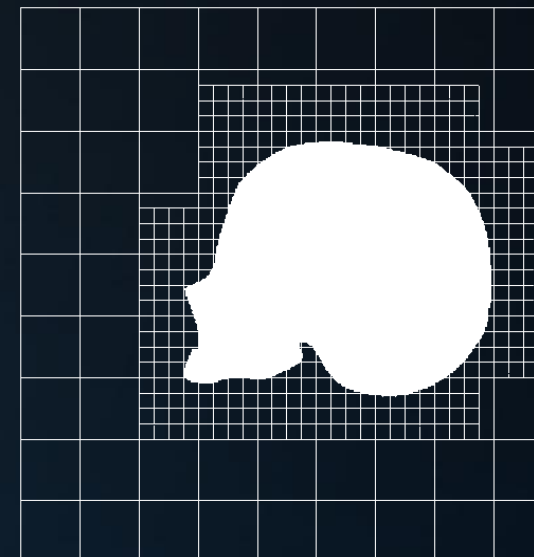
Algorithm:

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

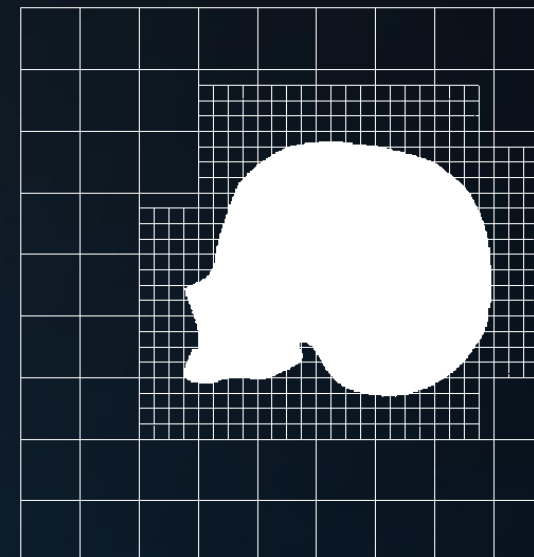
Pragmatic GPU Ray Tracing

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

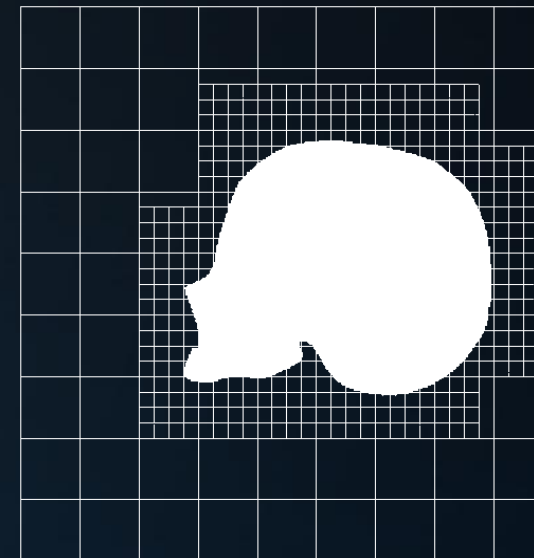
Pragmatic GPU Ray Tracing

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

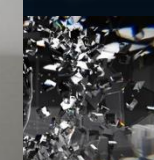
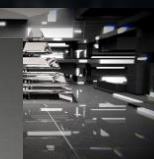
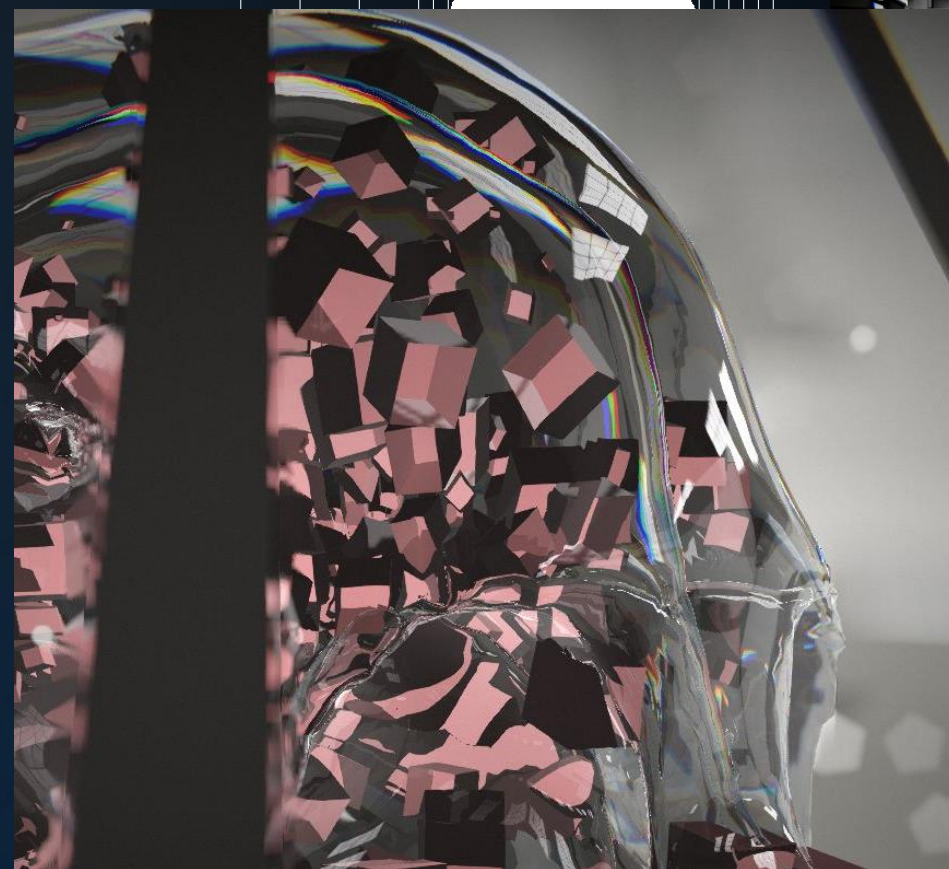
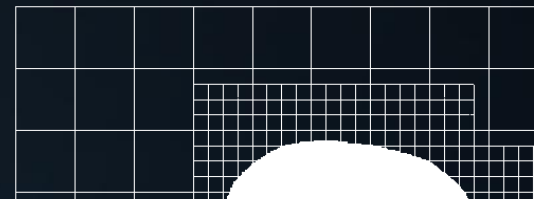
2013

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>



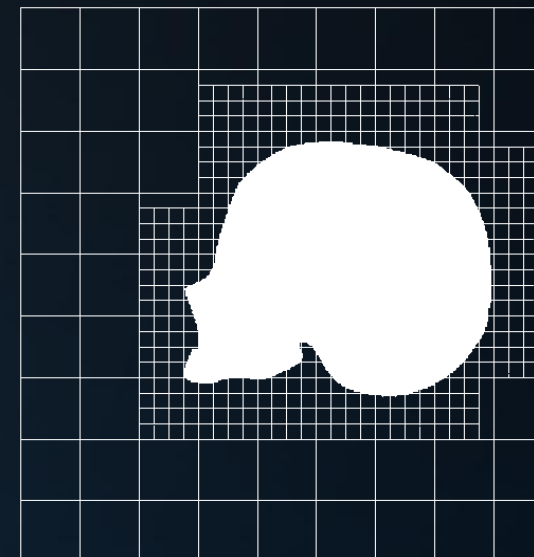
5 Faces

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

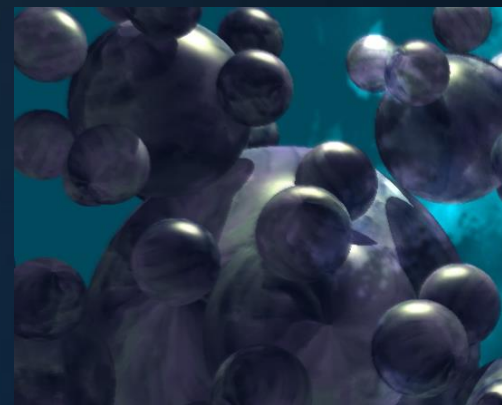
For a brief history, see these links:

<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”

