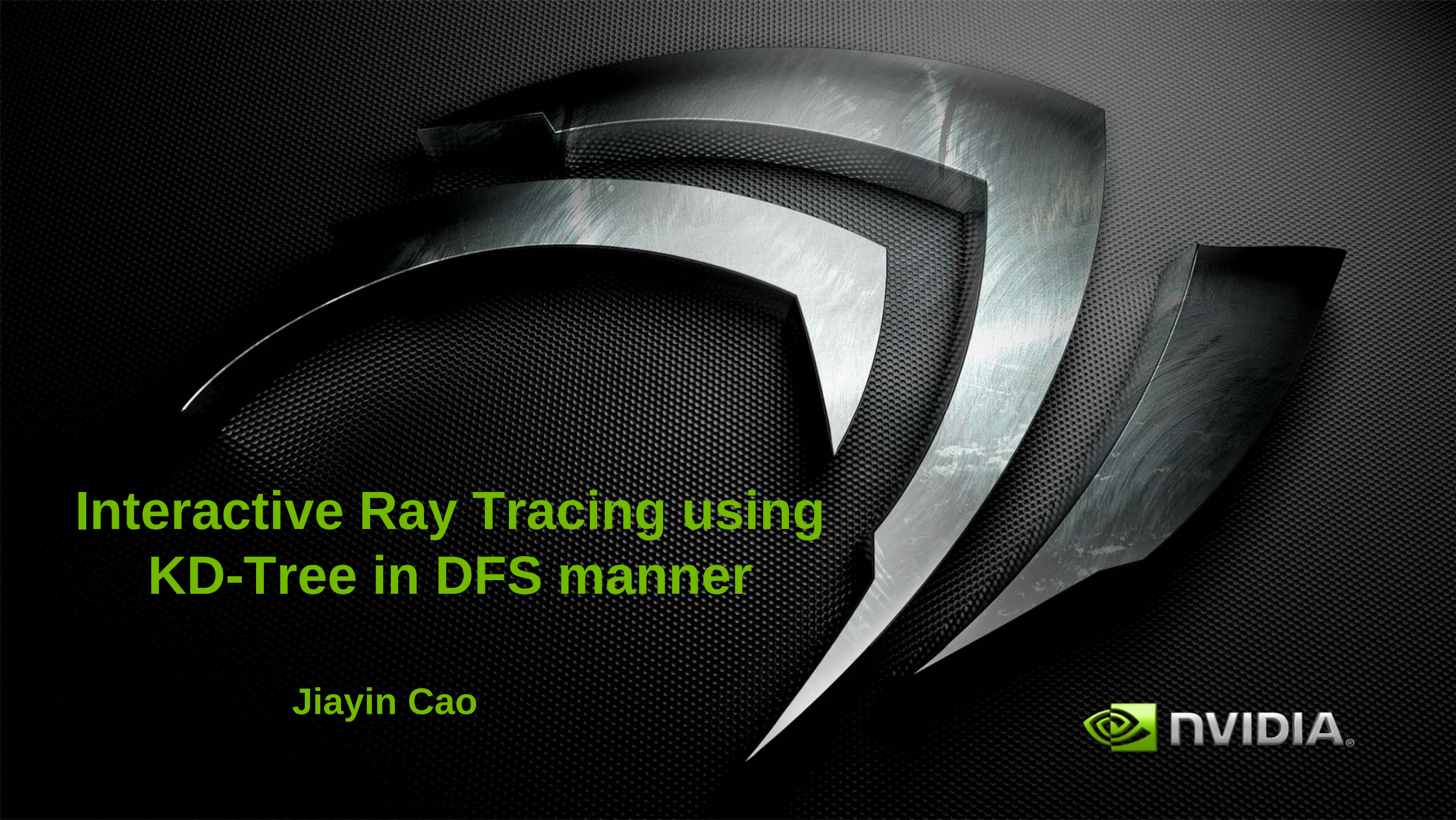

Interactive Ray Tracing using
KD-Tree in DFS manner
Jiayin Cao
NVIDIA

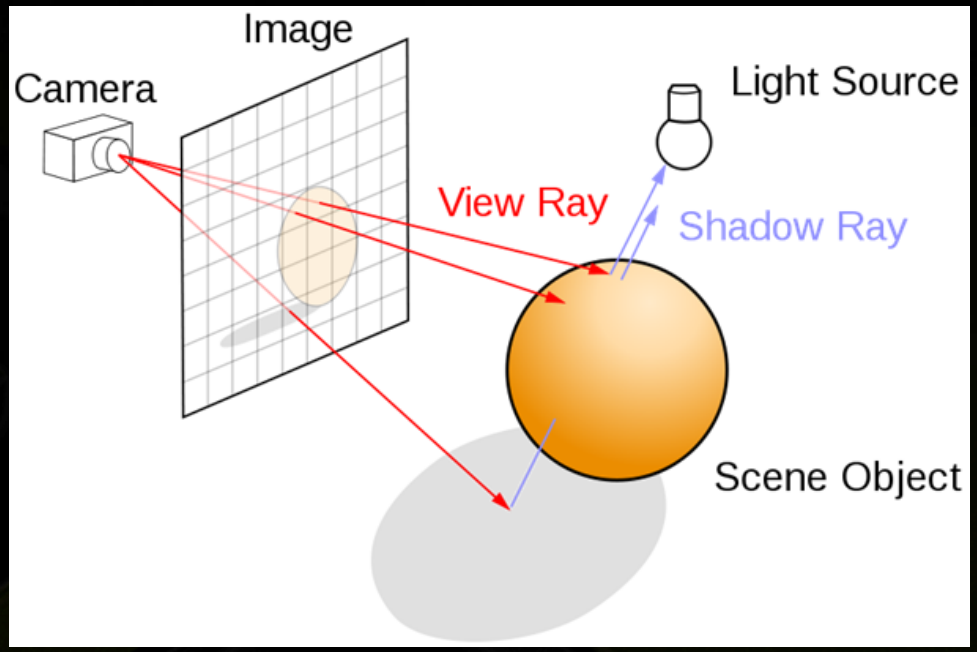
# Agenda


- The highlight of my work.
- What’s ray tracing.
- How do I map ray tracing algorithm on CUDA hardware.
- The performance of the algorithm.

# The highlight of my work

- Mapping ray tracing algorithm on CUDA hardware well , support texture , shadow , reflection , refraction and multiple lights.
- A depth first search algorithm in GPU kernel without hardware stack.
- Achieving interactive performance for complex scenes.

# The way ray tracing works



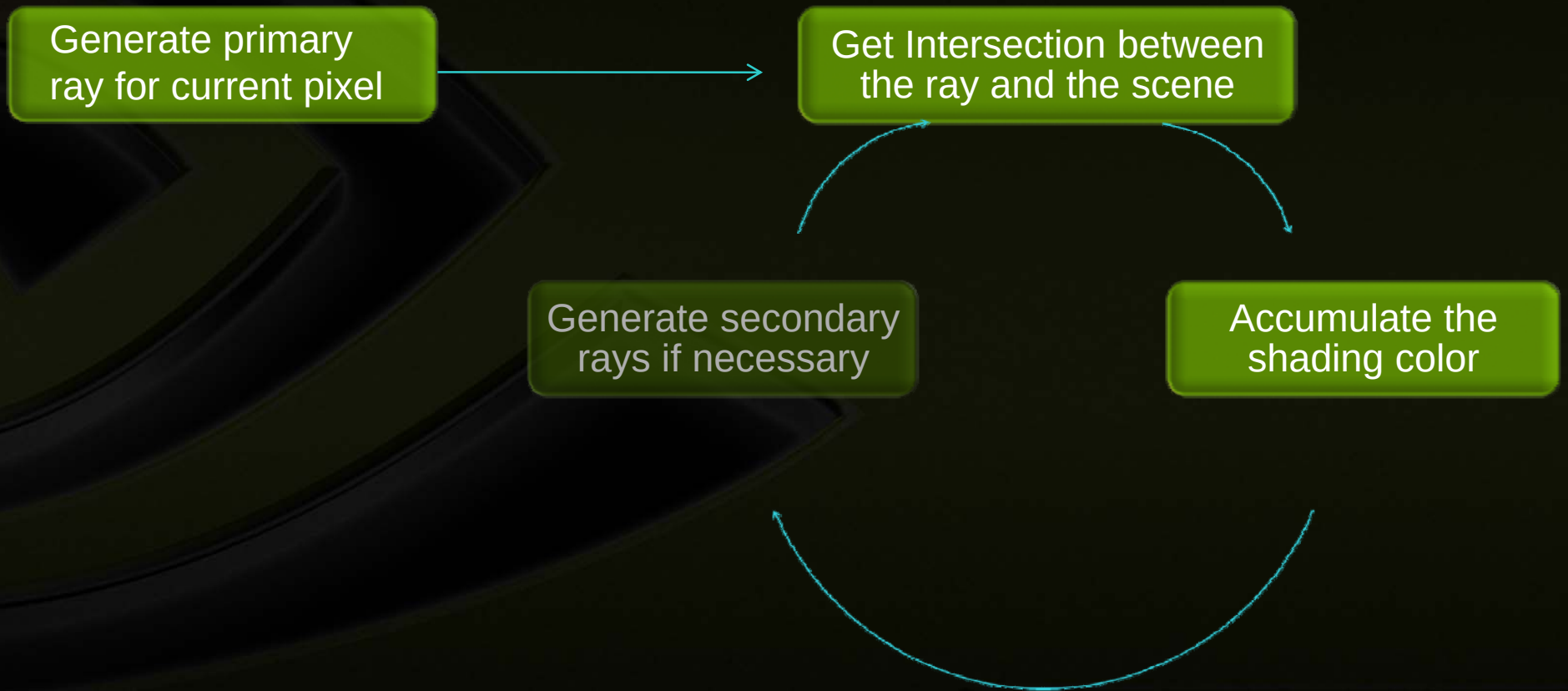

**In computer graphics, ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane.**

# Why ray tracing

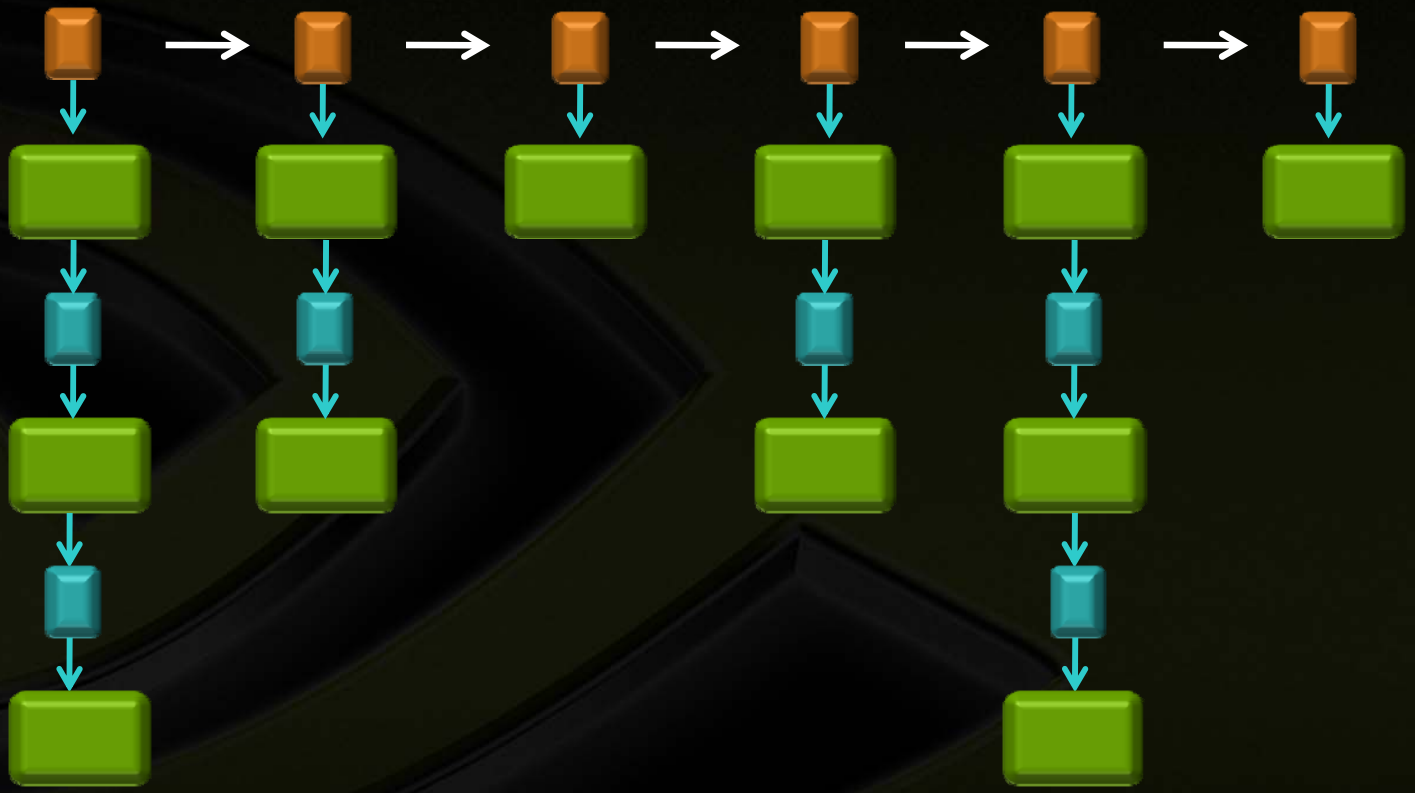

- Ray tracing can generate photo realistic images.( reflection, refraction )
- Ray tracing is software rendering , which means that every pass is programmable.
- Ray tracing could be mixed with other algorithm, such as photon mapping , radiosity.

# A naïve ray tracing algorithm

- **The following operations are performed for each pixel:**



# A naïve ray tracing algorithm



Accumulate shading result ( get intersection , brdf )

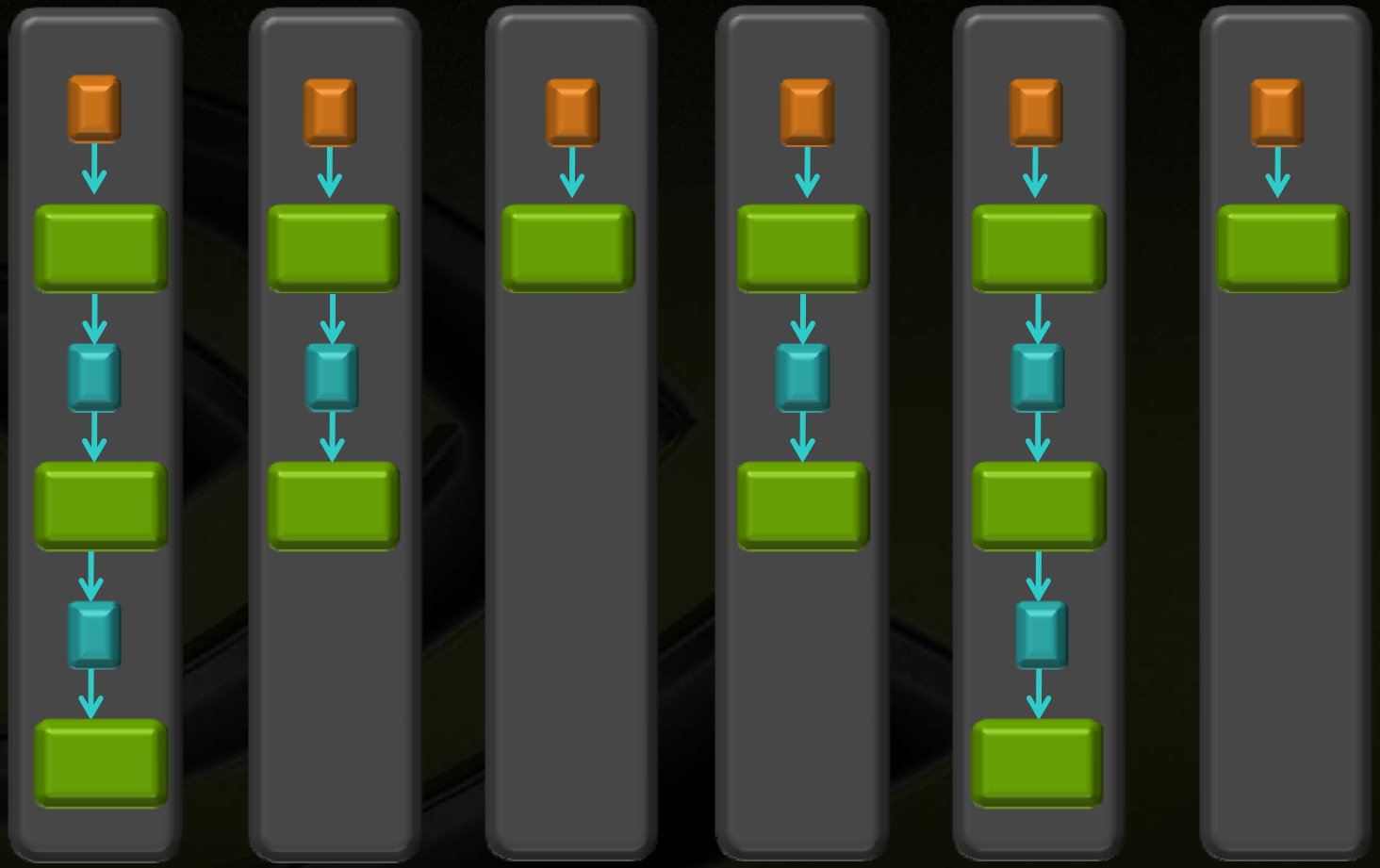
Generate next level ray

Generate primary ray

# The problems need to be solved

- How to map the algorithm on CUDA hardware.
- How to traverse KD-Tree to get the intersection result.

# A simple way of doing GPU ray tracing



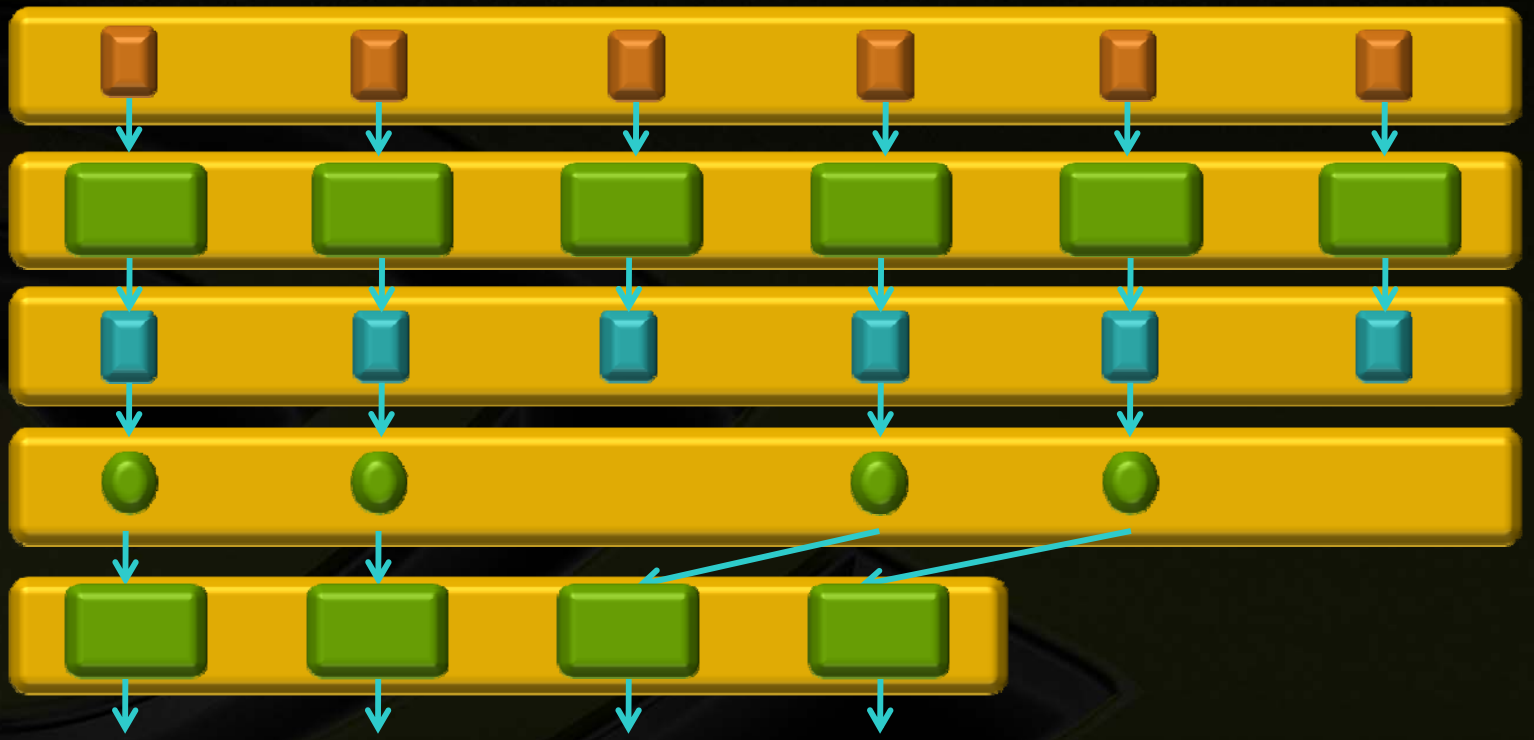
Each pixel is processed by one single thread.

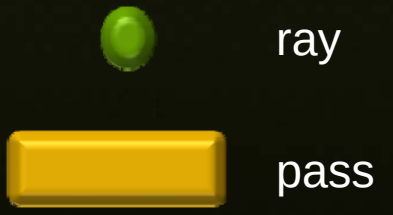
## Problem

Divergence in warp will cause bad performance.

thread

# The way my algorithm works

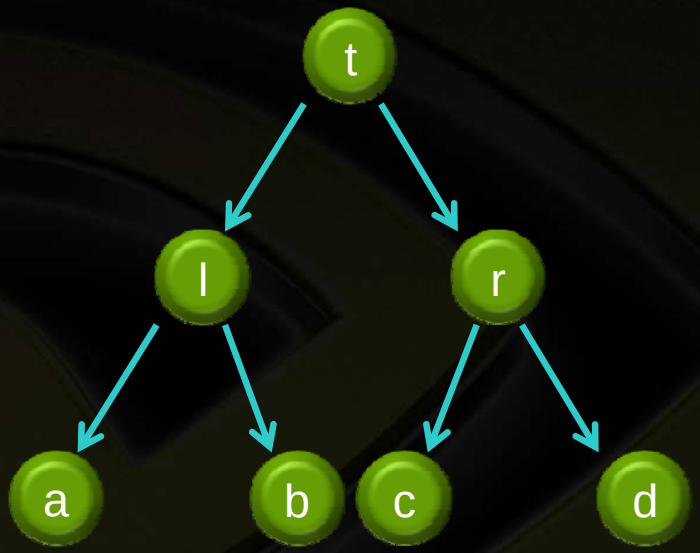


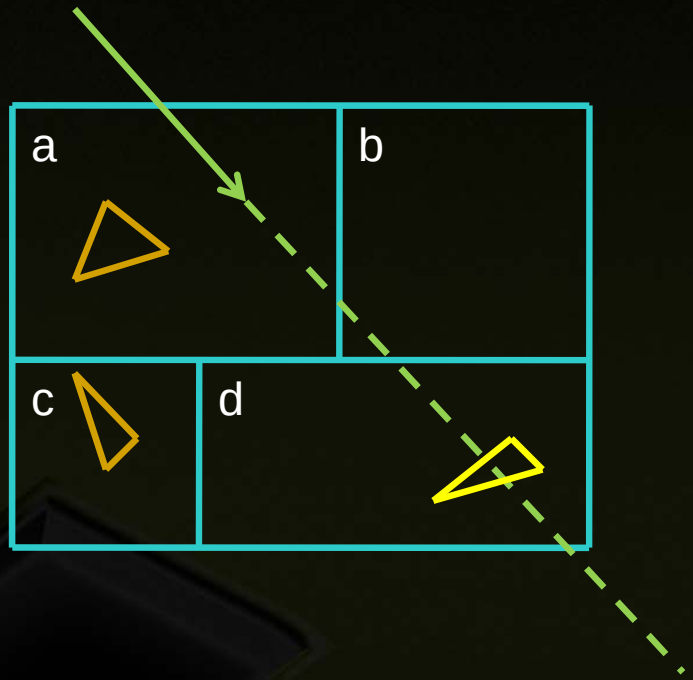


# Depth first search in KD-Tree

- Problem:
  - There is no hardware stack in GPU kernel.
- The way it is solved:
  - Using 32-bits unsigned integer as stack. ( fast , low cost )

# Depth first search in KD-Tree

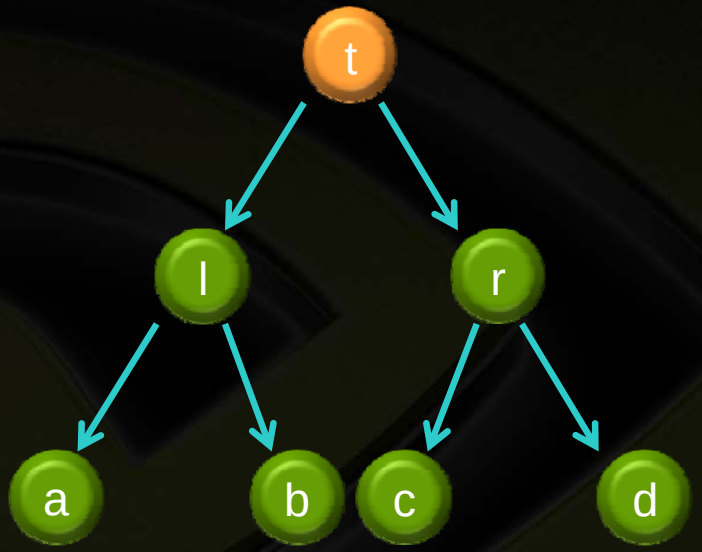


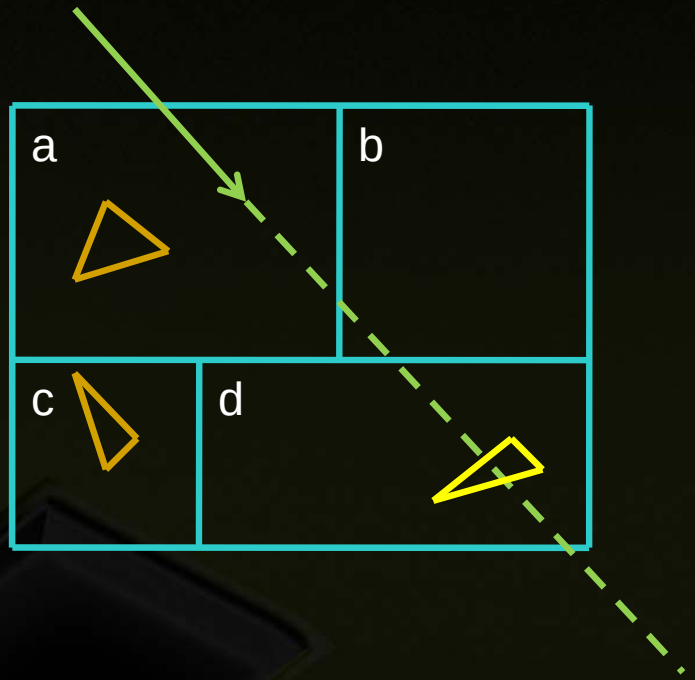




**KD-Tree Traverse**

t → l
l → a
a → l
l → b
b → l
l → t ( important )
t → r
r → d

# Virtual stack using bits



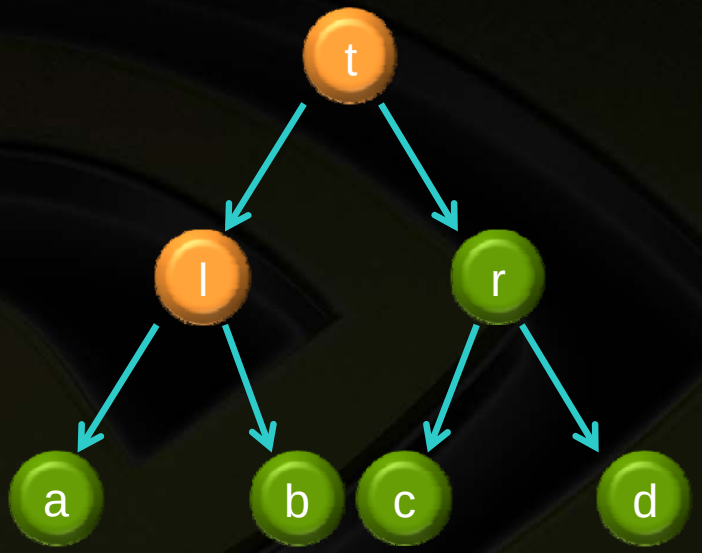



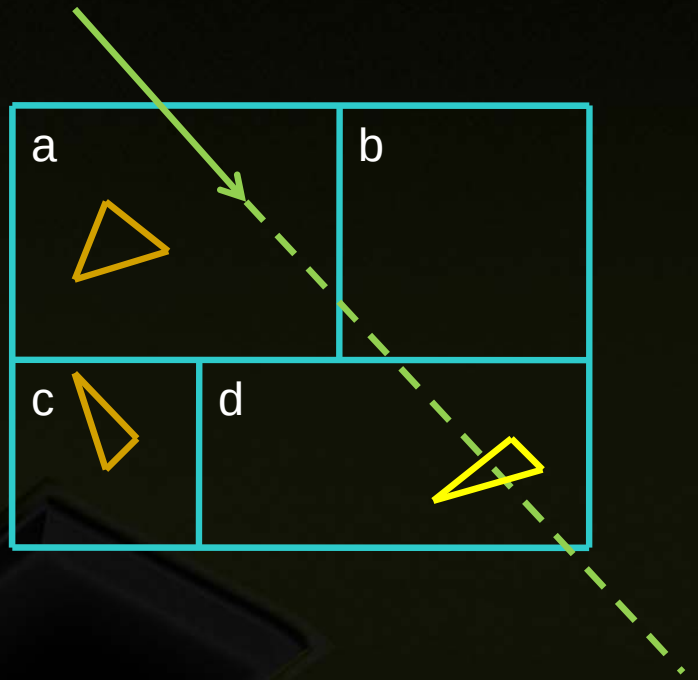

Virtual stack

0
0
0
0
…

Traverse state

# Virtual stack using bits



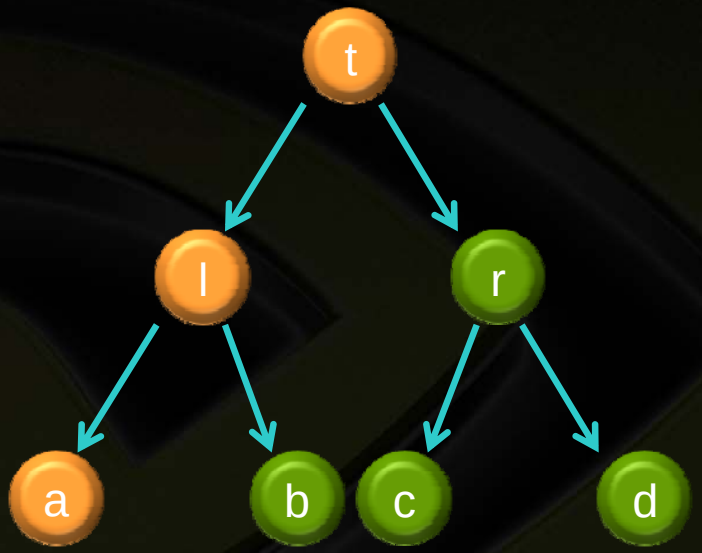



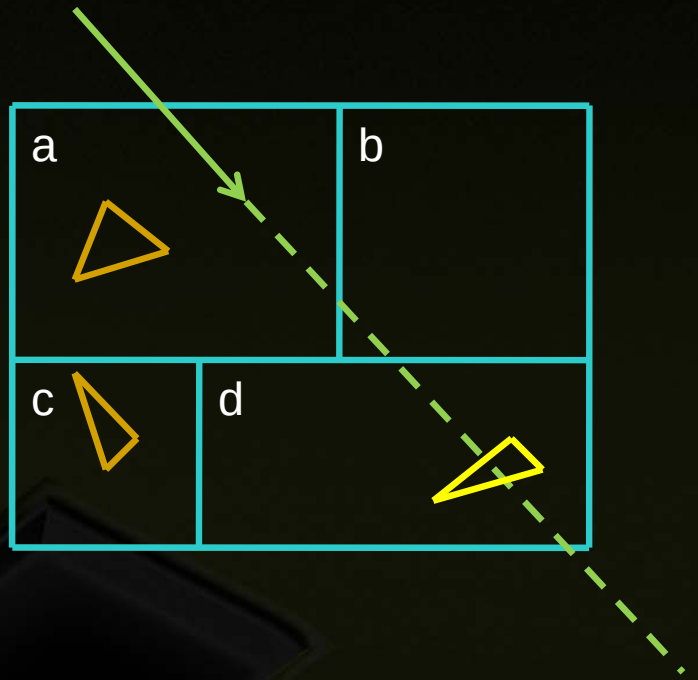

Virtual stack

0
0
0
0
…

Traverse state

# Virtual stack using bits



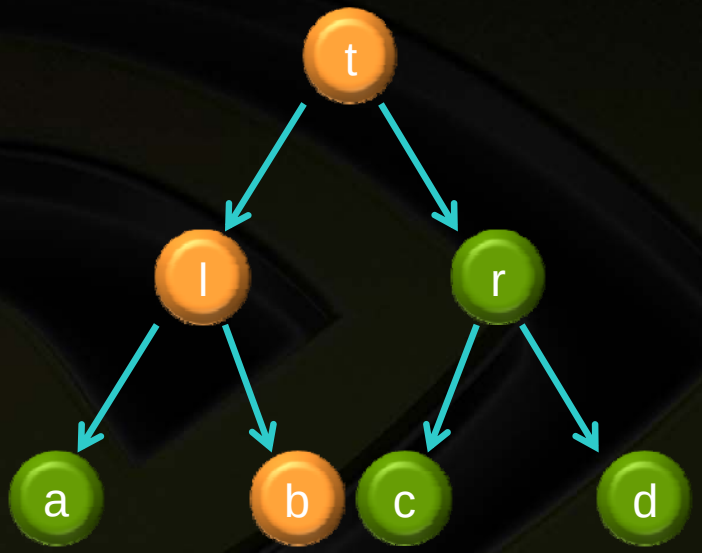



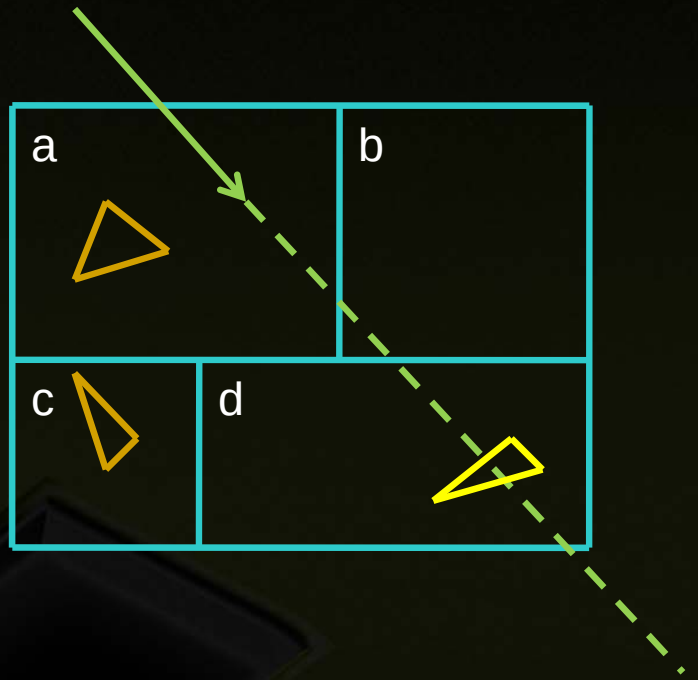

**Virtual stack**

0
0
0
0
…

**Traverse state → Back track state**

# Virtual stack using bits



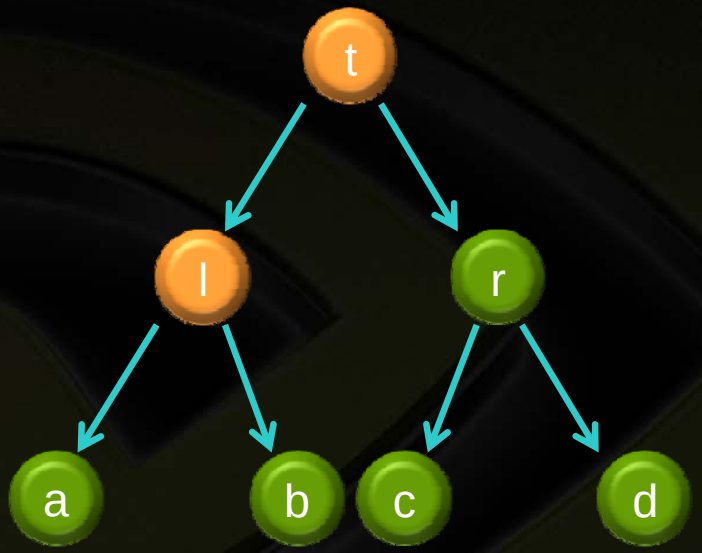



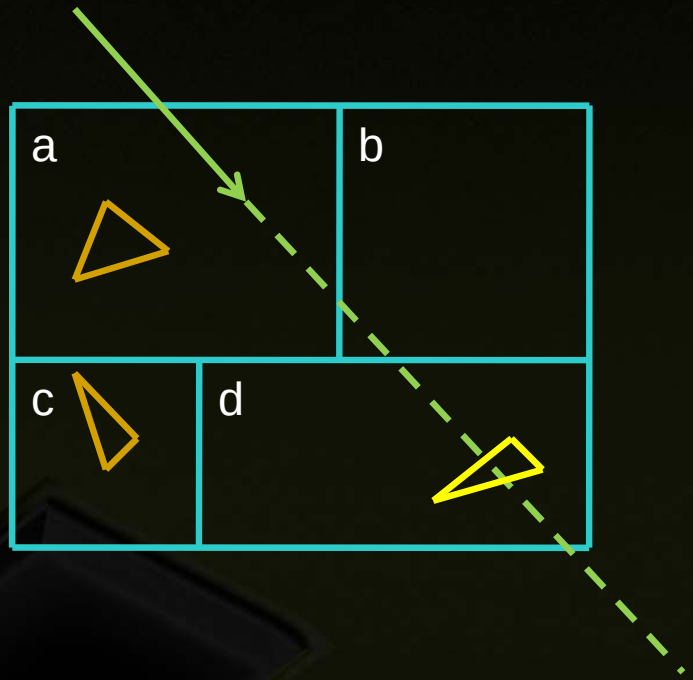

**Virtual stack**

0
0
1
0
…

**Traverse state → Back track state**

# Virtual stack using bits



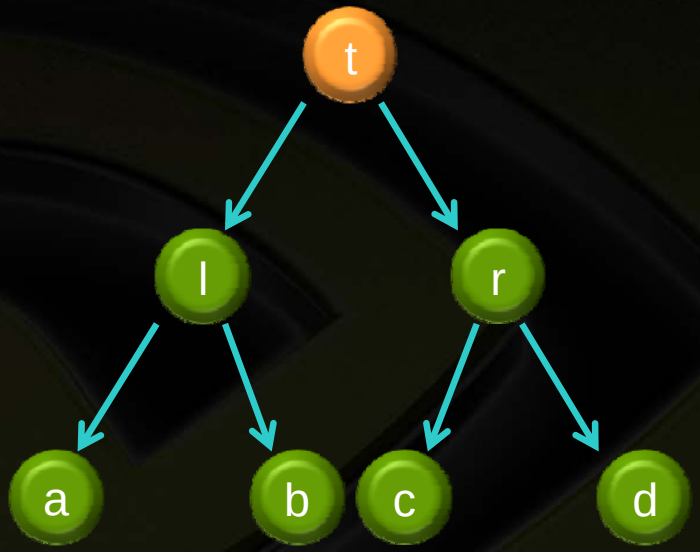



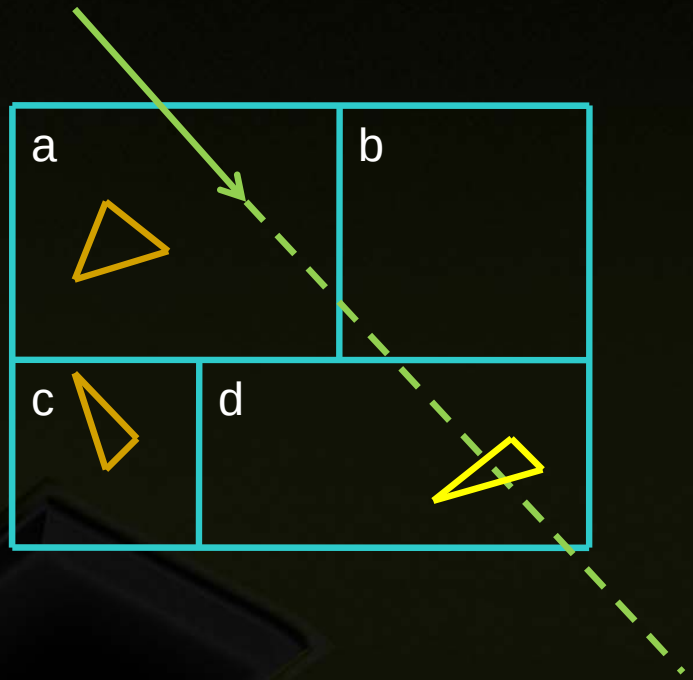

Virtual stack

0
0
0
0
…

Back track state

# Virtual stack using bits



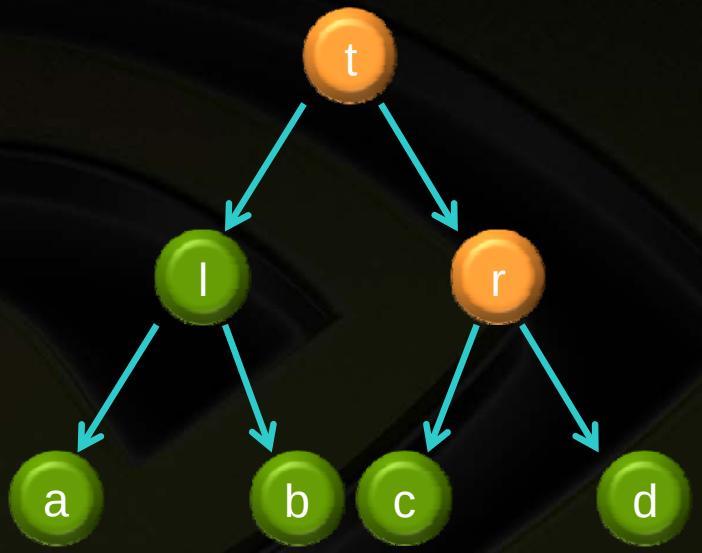



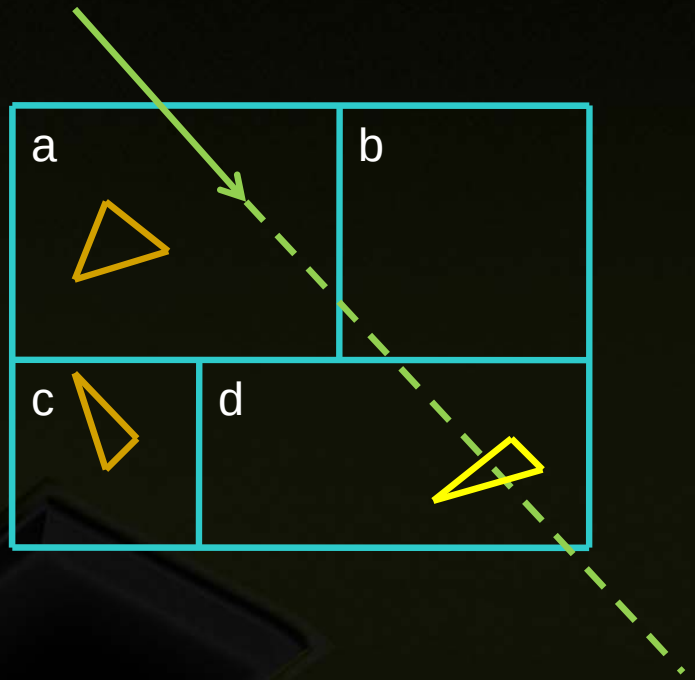

**Virtual stack**

**0**
**0**
**0**
**0**
**…**

**Back track state**

# Virtual stack using bits



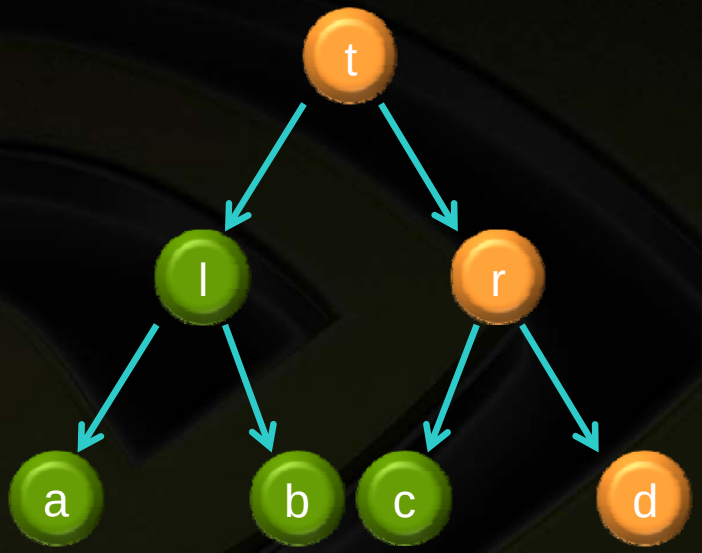



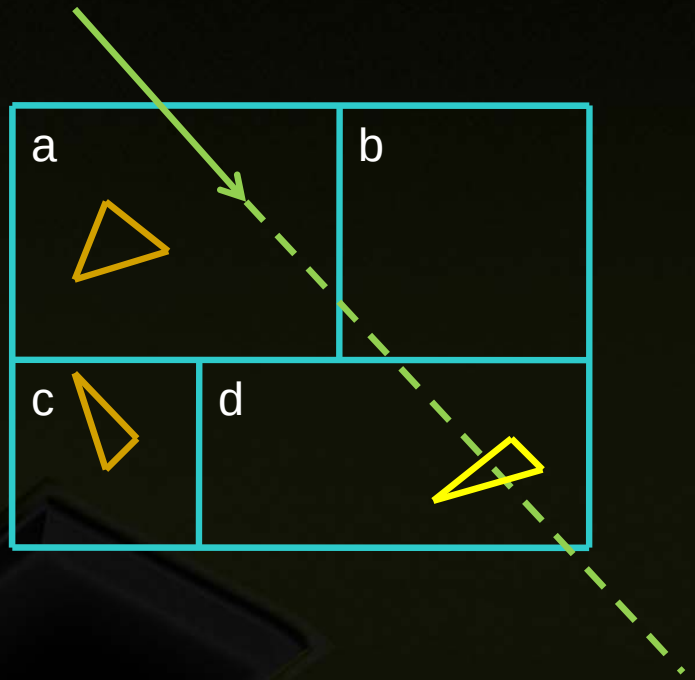

**Virtual stack**

0
0
0
0
…

**Traverse state**

# Virtual stack using bits



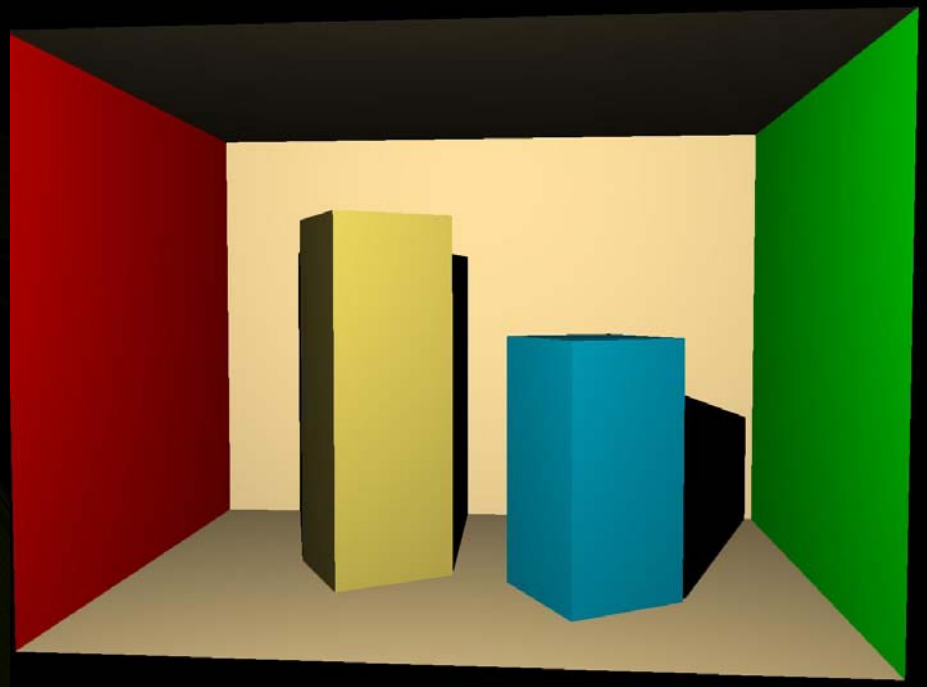




**Virtual stack**

0
0
0
0
…

**Traverse state**

# The performance of the algorithm

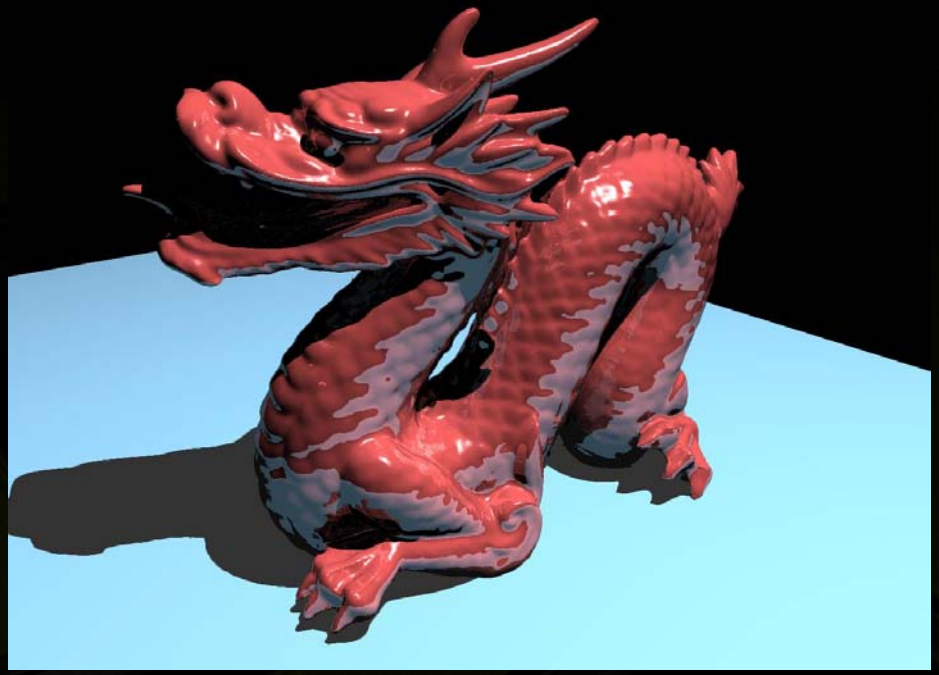


Cornell Box ( 34 triangles )

| GPU | FPS |
|---|---|
| GTX 285 | 30 |
| GTX 460 | 50 |

70 times faster than a naïve CPU ray tracing algorithm.

# The performance of the algorithm



Dragon on table ( 872870 triangles )

| GPU | FPS |
|---|---|
| GTX 285 | 3 |
| GTX 460 | 7 |

18 times faster than a naïve CPU ray tracing algorithm.

# The gallery