Mathias Steinbach

Interactive *k*-D Tree GPU Raytracing

(Horn et al. I3D 2007)

Seminar Computergrafik

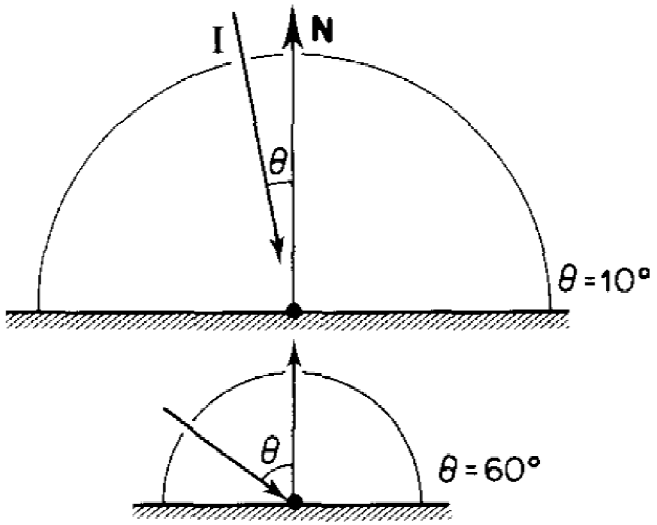
Gliederung

- Motivation
- Traversieren von *k*-d-Bäumen
- Traversieren auf der GPU
  - notwenige Anpassungen
  - Optimierung
- Umsetzung
- Auswertung

Motivation

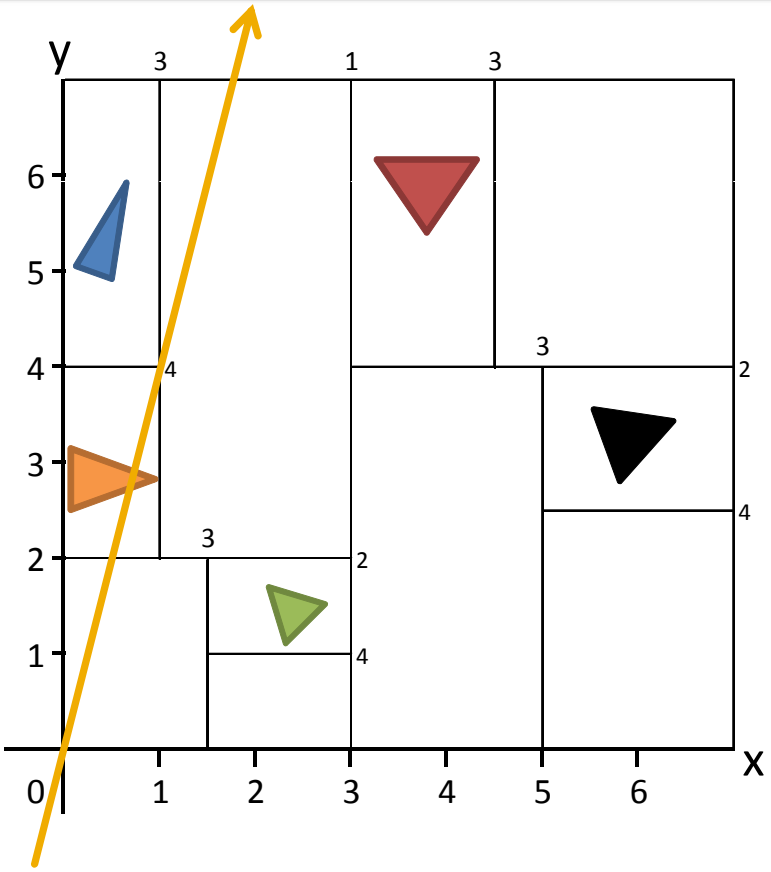
diffuse Reflexionen

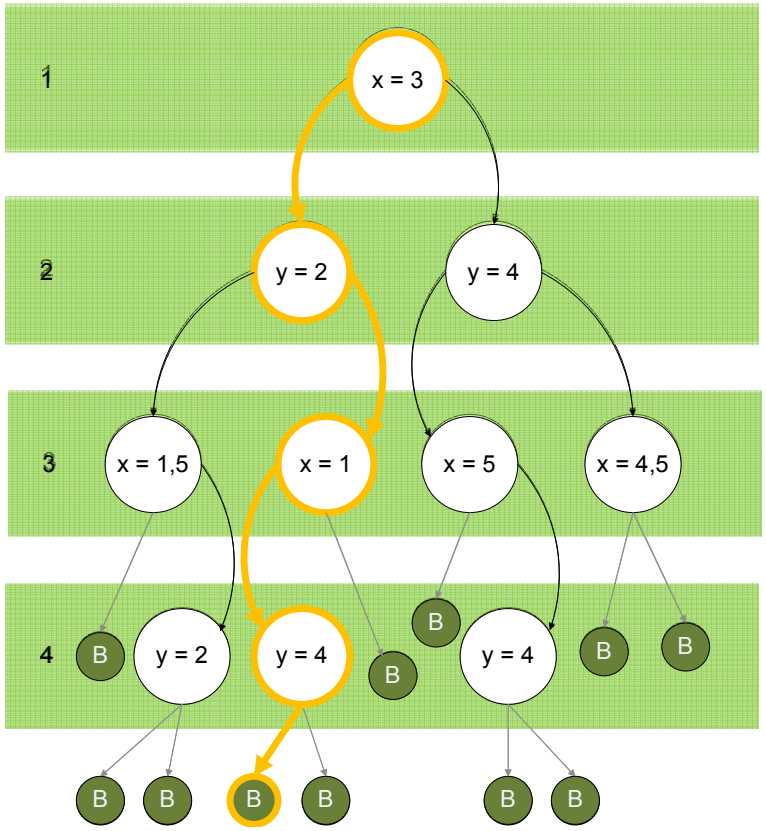
spiegelnde Reflexionen



Reflexionen:
Einfallswinkel = Ausfallswinkel

Aufbau von *k*-d-Bäumen



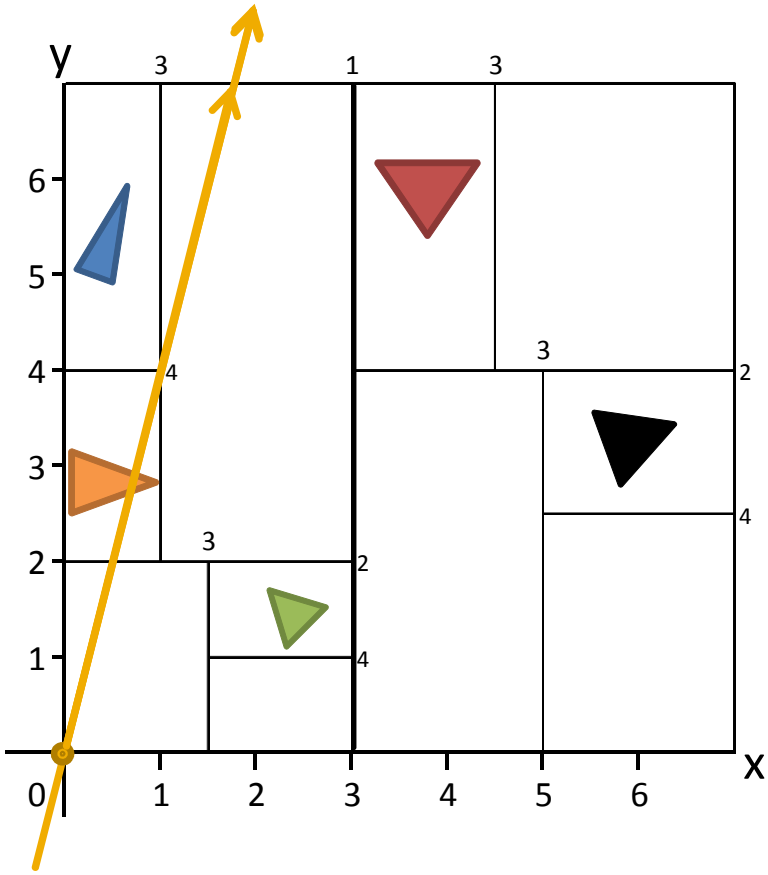


- schnelle Suche von Polygonen die ein Strahl schneidet
- beste Beschleunigungsstruktur für GPU-Raytracer
  (Foley et al. 2005)

Traversieren von *k*-d-Bäumen

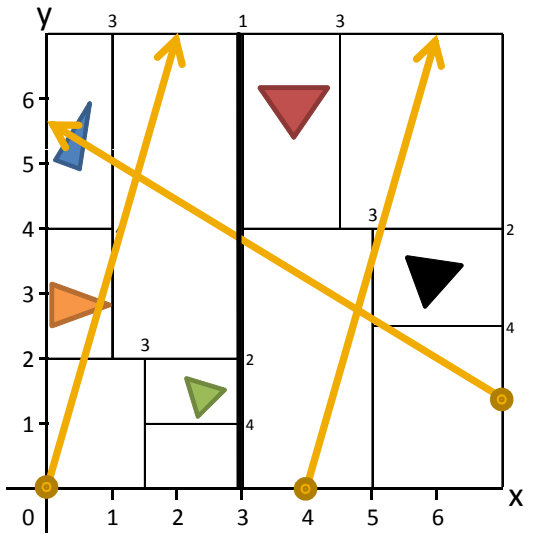
- Wie lang muss ein Strahl sein, damit er eine Schnittebene trifft?
- benötigt:
  - Strahl-Richtung (normiert)
  - Strahl-Ursprung
  - Weg durch die Szene
  - Weg zur Szene
- Weg zur Schnittebene:

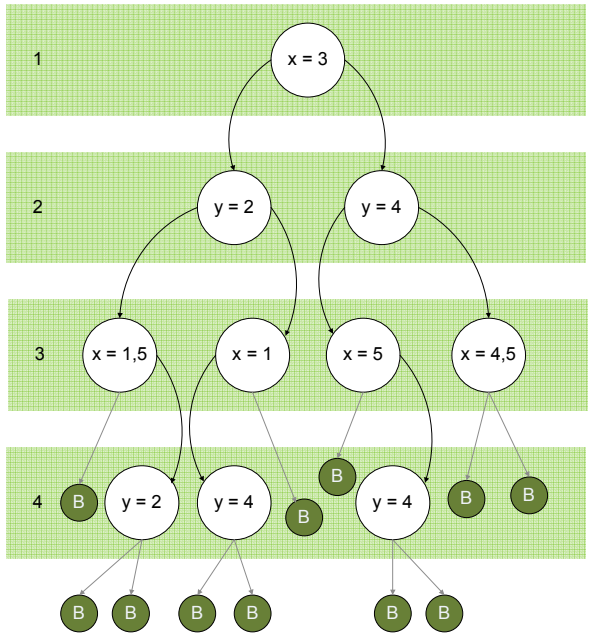
$$Weg| = \frac{(Grenze - Ursprung[a])}{Richtung[a]}$$



Berechung und Auswertung

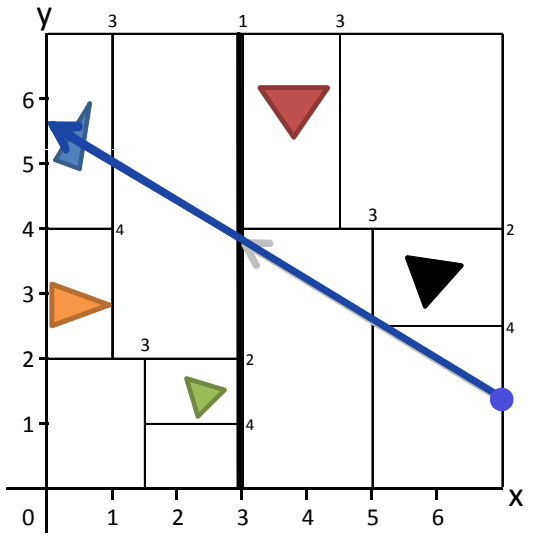
- $Weg| = \frac{(Grenze - Ursprung[a])}{Richtung[a]}$
- bestimme nahe und ferne Seite
- wenn: $Weg| \geq max.Weg\ (oder\ Weg| < 0)$
  - § suche auf der nahen Seite weiter
- sonst:
  - § wenn: $Weg| \leq min.Weg$
    - suche auf der fernen Seite weiter
  - § sonst
    - suche auf beiden Seiten weiter
    - nah: $max.Weg = Weg|$
    - fern: $min.Weg = Weg|$

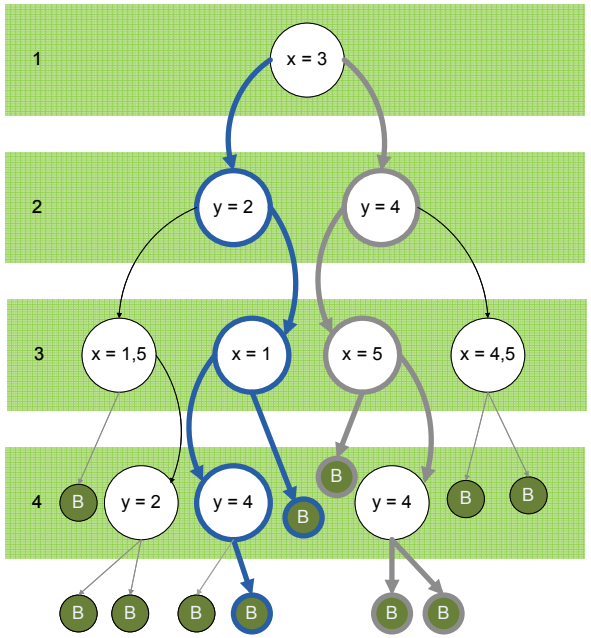




Algorithmus

```
Stack.push( Wurzel, min_Weg, max_Weg)
while( Stack.nicht_leer() )
{
    [Knoten, min_Weg, max_Weg] = Stack.pop()

    while( Knoten.ist_kein_Blatt() )
    {
        [Weg_zur_Ebene, Knoten_Nah, Knoten_Fern] = bereche( Strahl, Knoten )

        //Auswerten:
        //Auf der nahen Seite weitersuchen:
            Knoten  = Knoten_Nah
        //Auf der Fernen Seite weitersuchen:
            Knoten  = Knoten_Fern
        //Auf beiden Seiten weitersuchen:
            Stack.push( Knoten_Fern, Weg_zur_Ebene, max_Weg )
            max_Weg = Weg_zur_Ebene
            Knoten  = Knoten_Nah
    }

    for( Polygon : Knoten.Polygone() )
        if ( schneidet( Strahl, Polygon ) )
            return Polygon

}
return NULL
```





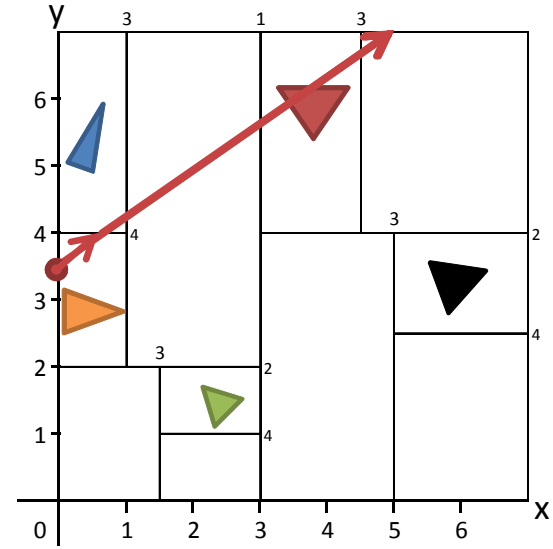
Traversieren auf der GPU

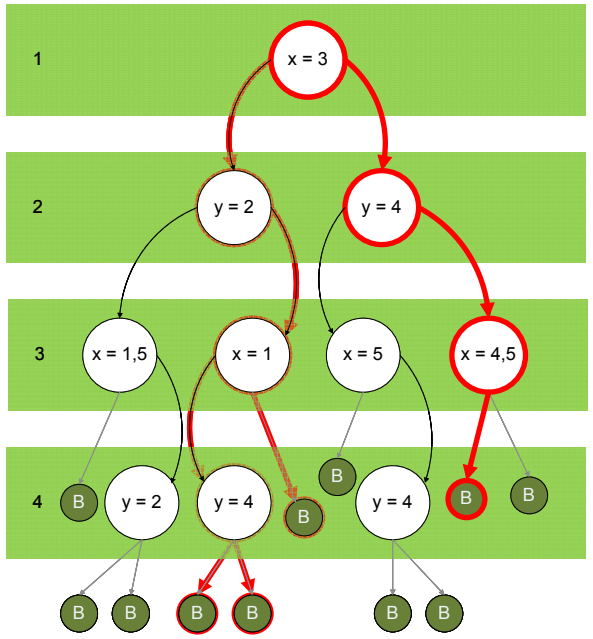
- Problem:
  - jeder Strahl benötigt einen eigenen Stack
  - nur wenig schneller Speicher auf der GPU
  - rekursionen sind nicht möglich
- Lösung:
  - Foley & Sugerman: kd-restart

kd-restart

(Foley, Sugerman 2005)

```
//Stack.push( Wurzel, min_Weg, max_Weg)
while( max_Weg < Weg_durch_die_Szene ) //Stack.nicht_leer() )
{
    //[Knoten, min_Weg, max_Weg] = Stack.pop()
    Knoten  = Wurzel
    min_Weg = max_Weg
    max_Weg = Weg_durch_die_Szene

    while( Knoten.ist_kein_Blatt() )
    {
        [Weg_zur_Ebene, Knoten_Nah, Knoten_Fern] = bereche( Strahl, Knoten )

        //Auswerten:
        //Auf der nahen Seite weitersuchen:
            Knoten  = Knoten_Nah
        //Auf der Fernen Seite weitersuchen:
            Knoten  = Knoten_Fern
        //Auf beiden Seiten weitersuchen:
            //Stack.push( Knoten_Fern, Weg_zur_Ebene, max_Weg )
            max_Weg = Weg_zur_Ebene
            Knoten  = Knoten_Nah
    }

    for( Polygon : Knoten.Polygone() )
        if ( schneidet( Strahl, Polygon ) )
            return Polygon

}
return NULL
```





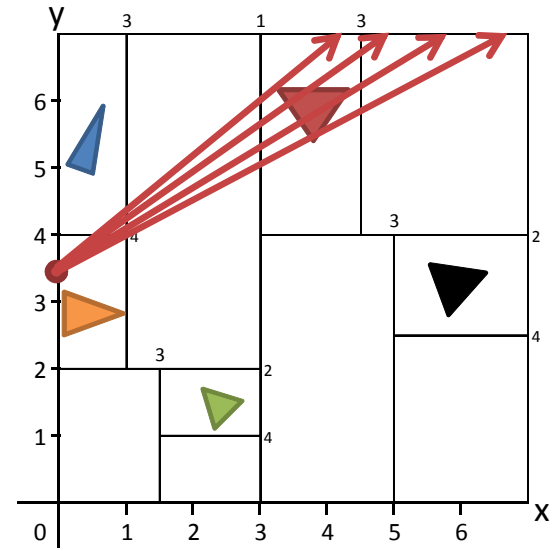
Optimierungen

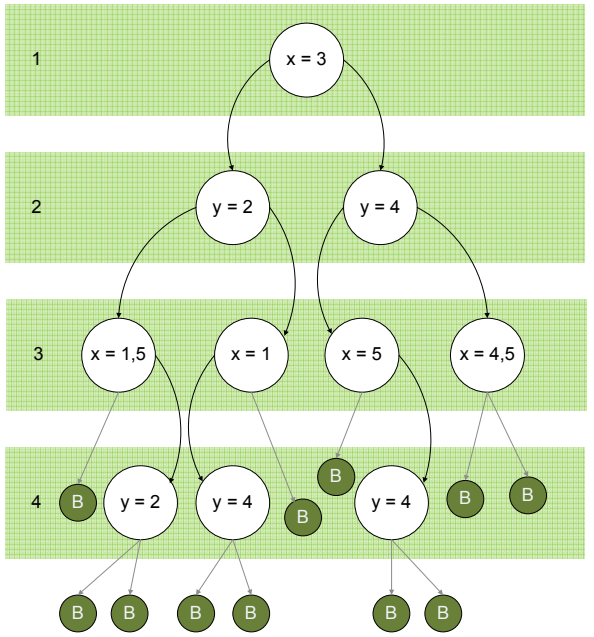
(Horn, Sugerman, Houston & Hanrahan I3D 2007)

- **Packets**
  - „fast" parallele Strahlen in Pakete gruppieren
- **Push-Down**
  - Neustart nicht bei der Wurzel sonder tiefer beginnen
- **Short-Stack**
  - einfügen eines begrenzten Stacks fester Größe

Packets

- Annahme: „fast“ parallele Strahlen benutzen den gleichen Pfad durch den Baum
- SIMD: gleiche Operationen auf unterschiedliche Daten sind Hardware beschleunigt
- weniger Datentransfer
- unterschiedliche Verzweigungen erst in der Nähe der Blätter
- notwendig: Maske um überflüssige Strahlen auf unterschiedlichen Pfaden auszublenden

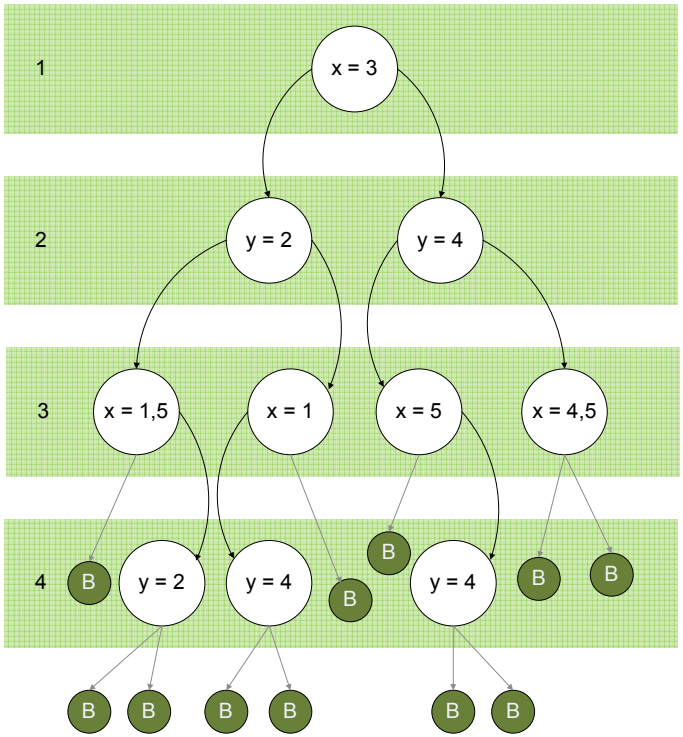




Push-Down

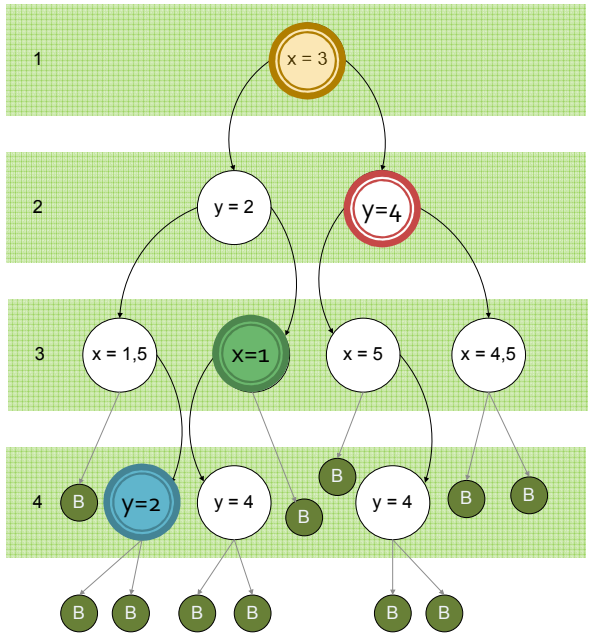
```
while( max_Weg < Weg_durch_die_Szene)
{
    Knoten   = Wurzel
    min_Weg = max_Weg
    max_Weg = Weg_durch_die_Szene

    push     = TRUE

    while( Knoten.ist_kein_Blatt() )
    {
        [Weg_zur_Ebene, Knoten_Nah, Knoten_Fern] = bereche( Strahl, Knoten )
        //Auswerten:
        //Auf der nahen Seite weitersuchen:
            Knoten   = Knoten_Nah
        //Auf der Fernen Seite weitersuchen:
            Knoten   = Knoten_Fern
        //Auf beiden Seiten weitersuchen:
            max_Weg = Weg_zur_Ebene
            Knoten   = Knoten_Nah
            push     = FALSE

        if( push )
            Wurzel = Knoten
    }
    for( Polygon : Knoten.Polygone() )
        if ( schneidet( Strahl, Polygon ) )
            return Polygon
}
return NULL
```

- Verringert Zugriffe auf Knoten auf den oberen Ebenen
- Neustart wird auf den Knoten vor der 1. Verzweigung verschoben

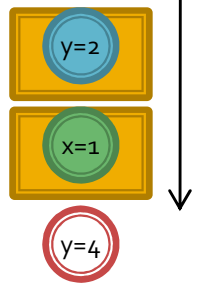


Short-Stack

- verhindert Neustart bei Verzweigungen auf den unteren Ebenen
- push() auf vollen Stack:
  - das unterste Element fällt heraus
- Stack leer:
  - normaler kd-restart

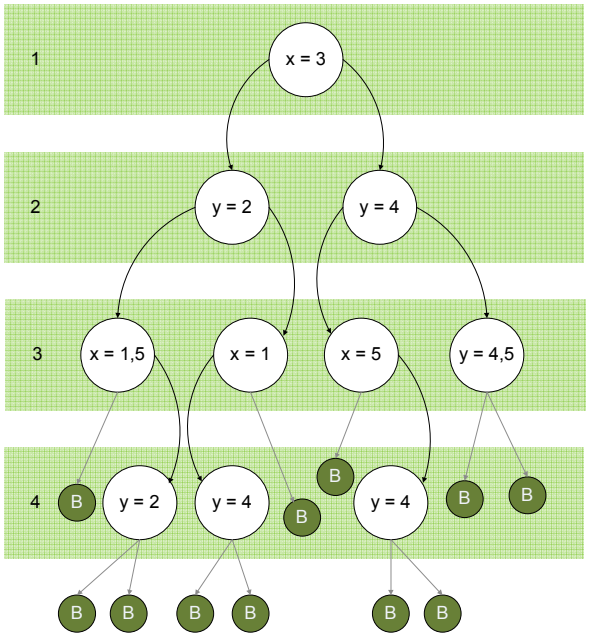


Short-Stack(2):



Short-Stack


```
while( max_Weg < Weg_durch_die_Szene)
{
    if( Stack.ist_leer() )
    {   Knoten   = Wurzel
        min_Weg = max_Weg
        max_Weg = Weg_durch_die_Szene
        push    = TRUE
    }
    else
    {   [Knoten, min_Weg, max_Weg] = Stack.pop()
        push    = FALSE
    }

    while( Knoten.ist_kein_Blatt() )
    {   [Weg_zur_Ebene, Knoten_Nah, Knoten_Fern] = bereche( Strahl, Knoten )
        //Auswerten:
        //Auf der nahen Seite weitersuchen:
            Knoten   = Knoten_Nah
        //Auf der Fernen Seite weitersuchen:
            Knoten   = Knoten_Fern
        //Auf beiden Seiten weitersuchen:
            Stack.push( Knoten_Fern, Weg_zur_Ebene, min_Weg)
            max_Weg = Weg_zur_Ebene
            Knoten   = Knoten_Nah
            push    = FALSE

        if( push )
            Wurzel = Knoten
    } ...
```

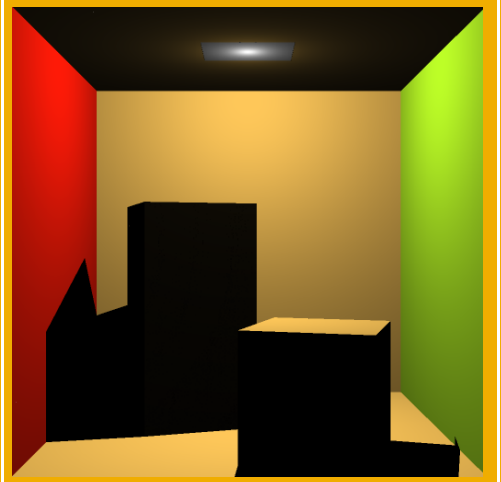
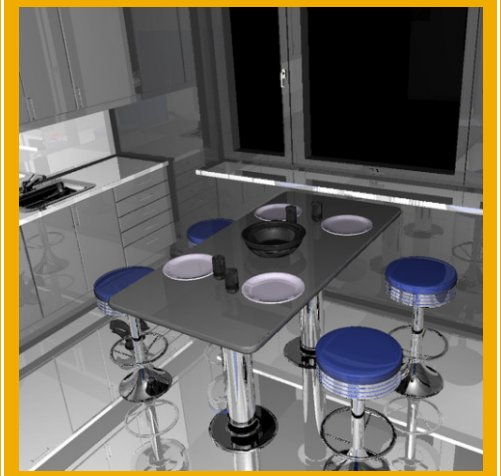
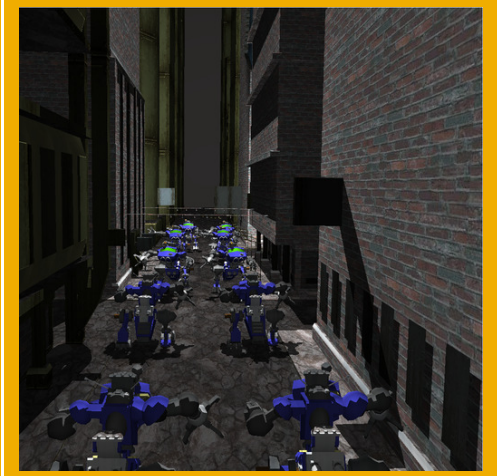


Umsetzung

- ATI X1900XTX
- Brook & Direct3D 9.0c
- Assembler optimierte Schleifen
- 4 Strahlen pro Paket
- Short-Stack für 3 Elemente
- Angepasste Baumhöhe
- GPU Rasterung für 1. Schnittpunkte

Auswertung

Geschwindigkeitszuwachs:

			
kd-restart	38.3	8.6	7.7
+ Packets	88.8	12.5	14.7
+ Short-Stack	91.3	16.3	17.9

Short-Stack

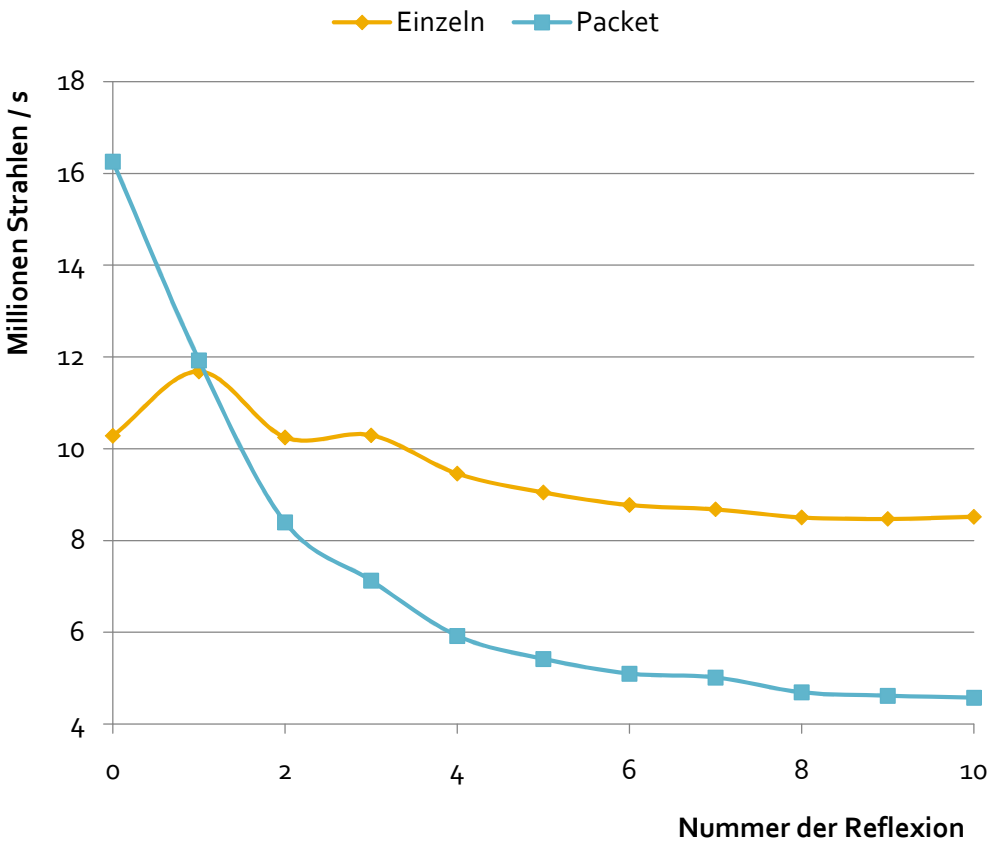
- kd-restart besucht 166% mehr Knoten als die Stack-Implementierung
  - Short-Stack mit einer Kapazität von 1 besucht nur 25% mehr Knoten

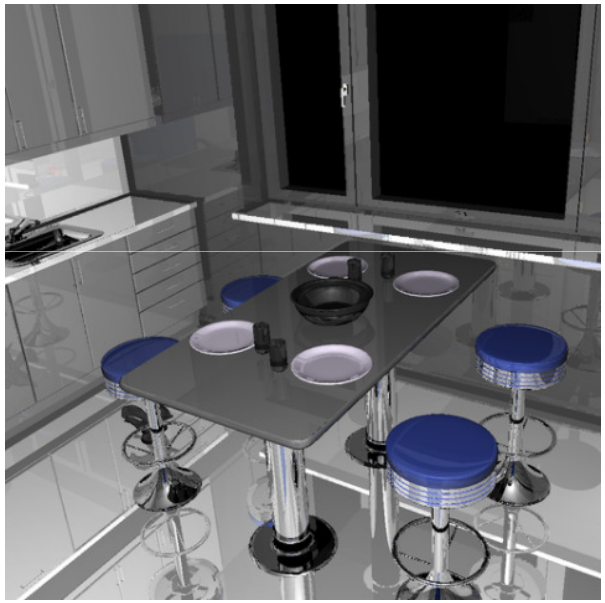
    - Speicherbedarf:
      - 36 Byte für Pakete
      - 12 Byte für einzelne Strahlen
  - Short-Stack mit einer Kapazität von 3 besucht nur noch 3% mehr Knoten

    - Speicherbedarf:
      - 108 Byte für Pakete
      - 36 Byte für einzelne Strahlen

Packets und Reflexionen

- Problem: Zuwachs von Entropie nach jeder Reflexion

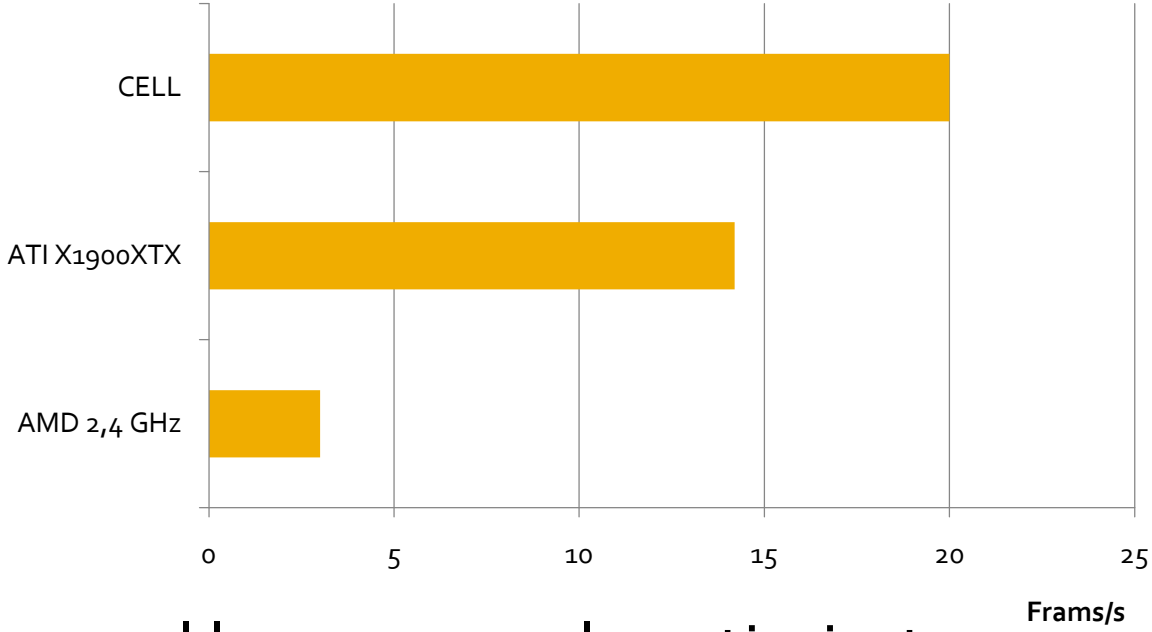




GPU erreicht nur 40-60% ihrer max. Instruktionsrate – Warum?

- Bandbreite und Speicherlatenzen?
  - nein: Verringerung des Speichertacks um 30% hat kaum Auswirkungen.
- Zu wenige unabhängige parallele Threads?
  - nein: Es gibt genügend Strahlen
- Unterschiedliche Ausführungspfade
  - ja: 48 Threads müssen sich den gleichen Programmzähler teilen.
  - Test mit Dummy-Kernel: gleiche Ausführungspfade aber keine Speicherzugriffe und Berechnungen führen zur gleichen Laufzeit!

Fazit



- Der alte kd-restart war 3 mal langsamer als optimierte CPU Raytracer
- Konkurrenzfähige GPU Implementierung
- Steigerung vor allem durch Hardware-Innovationen

Aktuell

- Nvidia Optix-Raytracing-Engine
- 30 Bilder pro Sekunde
  bei Full HD Auflösung
  (2 Mio. Polygone)
- Mit 2 Quadro
  FX 4700 X2
  zu je 2500 €

Interactive K-D Tree

GPU Raytracing

All images rendered at 640x480

Daniel Reiter Horn Jeremy Sugerman

Mike Houston Pat Hanrahan

Stanford University

Fertig!

Fragen?

Quellen:

Paper, Folien und Video:
Horn, D. R., Sugerman, J., Houston, M., Hanrahan, P. 2007. Interactive k-D Tree GPU Raytracing. http://graphics.stanford.edu/papers/i3dkdtree/

Paper:
Foley, T., and Sugerman, J. 2005. Kd-tree acceleration structures for a gpu raytracer. In HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, ACM Press, New York, NY, USA, 15-22.

Folien:
www.elho.net/pub/prosem_slide.ps.gz

Bilder und Inhalt:
http://www.computerbase.de/news/hardware/grafikkarten/nvidia/2008/august/nvidia_raytracing_gpu/