# Anmerkungen zur Stackless BVHs

Kai

Last built: 9. Juli 2012

## 1 Intro

Die normale Stackless-Sache siehe Thrane and Simonsen [1] zeigt entweder richtungsabhängige Performance oder ist langsamer als sie sein könnte.

Der erste Fall tritt auf, wenn beim Erstellen der Struktur die Knoten die jeweils den entsprechenden "negativen" Teilbaum darstellen einheitlich als linker oder rechter Teilbaum einsortiert werden. Dadurch ergbit sich, dass die Struktur von einer Richtung aus passend aufgebaut ist (d.h. die Knoten näher am Strahl werden auch zuerst traversiert), von der invertierten Richtung aus ist diese Struktur sehr ungünstig.

Um das zu vermeiden sortieren Thrane and Simonsen [1] die Knoten zufällig um, so dass der Baum kein direktes Bias mehr har. Das führt aber natürlich dazu, dass nicht nur der Worst-Case verschwindet, sondern auch der Best-Case.

### 1.1 Vergleich BVH/SBVH

### 1.2 Außen vs. Innen

Die für die Richtungsabhängigkeit aufgeführte Argumentation ist sehr anschaulich, wenn man sich die Strahlen von außerhalb der Szene stammend vonstellt. Wie verhält es sich, wenn die Strahlen z.B. vom Szenenmittelpunkt (z.B. in Sponza) geschossen werden?

*These.* Es wird sehr schnell erkannt dass Teilbäume nicht traversiert werden müssen weil sie hinter der Kamera liegen.

Das muss aber noch überprüft werden.

### 1.3 Relation zu naivem BVH-Traversal

Beim Traversieren von BVHs werden üblicherweise die Boxen der Kinder eines Knotens mit dem Strahl geschnitten. Aus dieser Berechnung geht auch die Distanz der jeweiligen Konten entlang des Strahls hervor, und gemäß dieser werden die Kindknoten korrekt sortiert auf den Stack abgelegt (so, dass der näheste zuerst traversiert wird).

Eine naivere Traversionsmethode ist den betrachteten Knoten selbst zu schneiden und die Kindknoten in beliebiger Reihenfolge auf den Stack zu legen. In dem Fall ergibt sich eine ähnliche Situation wie bei der SBVH: werden die Knoten nach einem bestimmten Musten auf

den Stack abgelegt schlägt sich das Richtungsabhängig nieder, wird randomisiert (was an der Stelle (aka inner loop) per se keine gute Idee ist) erhält man nur Average-Case Performance.

### 1.4 Traversion ohne Skip-Pointer

| | |
|---|---|
| $L(k,d)$ | left child of $k$ (when the ray direction is in octant $d$) |
| $R(k,d)$ | right child of $k$ (ditto) |
| $P(k)$ | parent of $k$ |
| $N(k,d)$ | $\begin{cases} L(k,d) & k \text{ inner node} \\ S(k,d) & k \text{ leaf node} \end{cases}$ |
| $S(k,d)$ | $\begin{cases} S(P(k),d) & R(P(k),d) = k \\ R(P(k),d) & \text{otherwise} \end{cases}$ |

### 1.5 Ray Codes

Wie oben angedeutet ist die Entscheidung immer relativ zur Strahlrichtung, z.B. Skip von Knoten $k$ bei Strahlrichtung $d$, was jeweils auf $L(k,d)$ bzw. $R(k,d)$ herunterkocht. Praktisch heißt das, dass die Kindknoten abhängig von der Strahlrichtung vertauscht werden müssen. Die Achse nach der das Volumen des Knotens unterteilt wurde spielt auch eine Rolle, weil eben diese Komponente der Strahlrichtung ausschlaggebend ist.

**Pro Knoten.** Für jeden Knoten wird deshalb gespeichert, für welche der 8 Grundrichtungen ein Vertauschen der Kindknoten nötig ist (das sind die Richtungen die eine negativen Wert in der Komponenten der Split-Ebene haben). Die Lookup-Tabelle $S$ ist wiefolgt aufgebaut:

$$\left[\begin{pmatrix}-\\-\\-\end{pmatrix}\ \begin{pmatrix}-\\-\\+\end{pmatrix}\ \begin{pmatrix}-\\+\\-\end{pmatrix}\ \begin{pmatrix}-\\+\\+\end{pmatrix}\ \begin{pmatrix}+\\-\\-\end{pmatrix}\ \begin{pmatrix}+\\-\\+\end{pmatrix}\ \begin{pmatrix}+\\+\\-\end{pmatrix}\ \begin{pmatrix}+\\+\\+\end{pmatrix}\right]$$

Für jede der 8 so indizierten Position der Tabelle wird gespeichert ob für diese Richtung die Kindknoten vertauscht werden müssen. Interpretiert man + als 0 und − als 1 in der Tabelle ergeben sich folgende Codes für die Split-Achsen:

| | |
|---|---|
| X | 10101010 |
| Y | 11001100 |
| Z | 11110000 |

**Pro Strahl.** Zu Begin der Traversion der BVH mit einem Strahl $\mathbf{o} + d\mathbf{t}$ wird für diesen ein Richtungs Code, der *Ray Code*, bestimmt. Dazu wird + und − wie oben auf Zahlen abgebildet und die so erhaltene Richtung (z.B. $(d_z\ d_y\ d_x) = (-+-) \to (1\ 0\ 1) = (c_z\ c_y\ c_x)$) als Binärzahl interpretiert: $c = 4c_z + 2c_y + c_x$. Diese gibt nun den Index in die Lookup-Tabellen der Knoten.

**Traversion.** Bei der Traversion wurde zu Beginn der Ray Code $c$ berechnet. Zum Bestimmen ob die Kinder eines Knoten vertauscht werden müssen wird nun einfach $s = S \sqcap 2^c$ geprüft. Ist diese Verknüpfung $s \geq 0$ müssen die Knoten getauscht werden.

**Details** Ein paar Eigenschaften:

- Der Vergleich läuft auf 1 bei Vertauschen heraus da die beiden Knoten sequentiell im Speicher liegen, also einfach $s' = s/2^c$ auf den Index des Knotens addiert werden muss um den zweiten zu erhalten.
- Die Lookup-Tabelle kann in einem Byte abgespeichert werden. Das ist besonders wichtig da so vermieden werden kann, dass die BVH größer als eine normale BVH ist.
- $10101010_2 = \mathrm{aa}_{16}$, $11001100_2 = \mathrm{cc}_{16}$, $11110000_2 = \mathrm{f0}_{16}$.

**Beispiel** Knoten mit Splitachse X, Strahlrichtung $(- + -)$.

- $S = 10101010$, $c = 5$,
- $s = S_c \sqcap 2^c = 10101010_2 \sqcap 00100000 = 32$.
- $s \geq 0 \rightarrow$ vertauschen nötig.
- Das passt auch, weil die Strahlrichtung in X negativ ist.

# Literatur

[1] Niels Thrane and Lars Ole Simonsen. A Comparison of Acceleration Structures for GPU Assisted Ray Tracing. Master's thesis, University of Aarhus, Denmark, August 2005.