Architektur und Programmierung von Grafik- und Koprozessoren

Rendering Algorithmen

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

Beschleunigungsdatenstrukturen

Offensichtlich: höherer Realismus $\Rightarrow$ viel mehr Strahlen. Raue Reflektion etwa benötigt mindestens $16 - 32$ Sample Strahlen. Für diese entstehen neue Strahlbäume mit erneuten Interaktionen mit rauen Oberflächen etc.

Offensichtlichste Optimierung: Suchdatenstruktur, um Schnitttestzahl zu reduzieren.

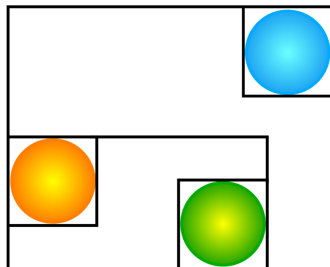
Beschleunigungsdatenstrukturen

Prinzip: Gruppe von möglichst vielen Primitiven umhüllen. Test gegen Hülle einfach (z. B. Schnitttest Strahl/Box oder Schnitttest Strahl/Ebene).

Falls Schnitt mit Hüllobjekt: teste gegen Primitive, sonst: breche ab.

Hierarchische Datenstrukturen, indem kleinere Hüllkörper von größeren Körpern umhüllt werden $\Rightarrow$ logarithmische Komplexität in Anzahl Primitive.

Objekt- und Raumaufteilung



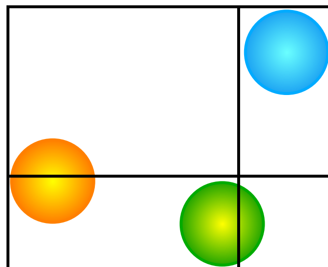
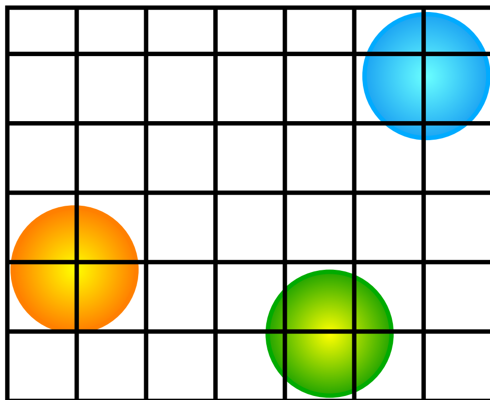


Abbildung: links: *Objekte* werden hierarchisch unterteilt, rechts: *Raum* wird hierarchisch unterteilt.

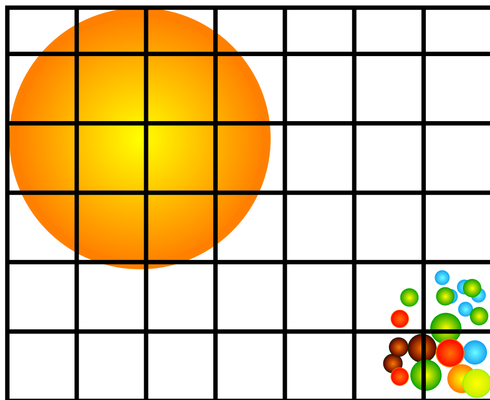
Uniforme Gitter



Vorteile:

- Kaum zusätzlicher Speicherbedarf.
- Strahl kann einfach in Gitterzellen *projiziert* werden.

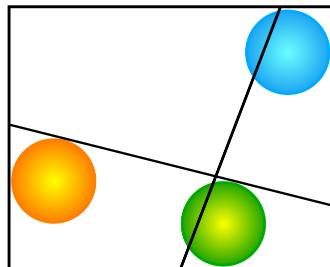
Uniforme Gitter

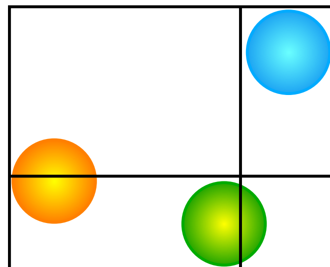


Nachteile:

- “Teapot in a stadium” Problem: viele kleine Primitive fallen in eine einzelne Zelle.

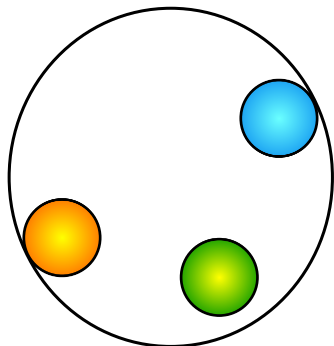
BSP Bäume

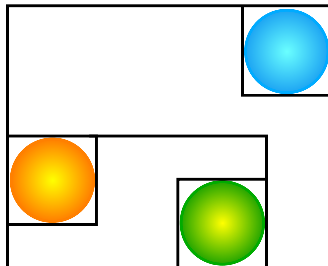




Teile Raum hierarchisch in *Halbräume* auf ("binary space partitioning trees"). Variante: (rechts) k-d Bäume – Ebenen parallel zu karthesischen Hauptebenen.

Bounding Volume Hierarchien





Axis-Aligned Bounding Box (AABB) BVHs am gebräuchlichsten.
BVHs derzeit die populärste Beschleunigungsdatenstruktur für Strahlverfolgung.

k-d Bäume vs. BVHs

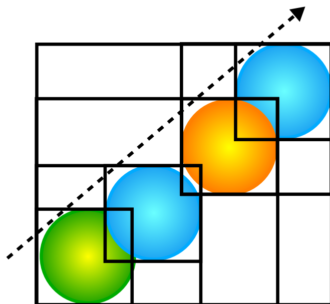
- Effizienz beim *Traversieren*: AABBs überlappen. Überlapp Gütekriterium für BVHs. k-d Bäume i. d. R. etwas im Vorteil.
- Effizienz beim *Konstruieren*: BVHs für hunderdtausende Dreiecke können mit modernen Algorithmen in Sekundenbruchteilen aufgebaut werden.
- BVH: Speicherbedarf a priori bekannt – Anzahl Blattknoten beschränkt durch Anzahl Primitive. (k-d Bäume: Primitive können in zwei Halbräume fallen $\Rightarrow$ zwei Referenzen speichern.)
- BVHs können auch in Teilen neu aufgebaut werden, k-d Bäume werden komplett neu konstruiert ("ymmv..").

k-d Bäume vs. BVHs

- Forschung: verbessere Konstruktionszeiten, ohne dass Traversierungsgeschwindigkeit zu stark in Mitleidenschaft gezogen wird.
- Hybride Ansätze: "SBVH": sind BVHs, falls beim Konstruieren Überlapp lokal sehr hoch, führe einen räumlichen Split ein.
- Rekonstruiere nur Teilbäume: "Two-level BVHs" – *Szenengraphtransformationen* für oberes Level, unten exakte BVHs.

Axis-Aligned Bounding Boxes

Konzeptionelles Problem mit AABBs: bei komplexen Szenen schneiden *tangential zur Geometrie verlaufende Strahlen* hunderte Bounding Boxen und kein Primitiv.



Axis-Aligned Bounding Boxes

Konzeptionelles Problem mit AABBs: bei komplexen Szenen schneiden *tangential zur Geometrie verlaufende Strahlen* hunderte Bounding Boxen und kein Primitiv.

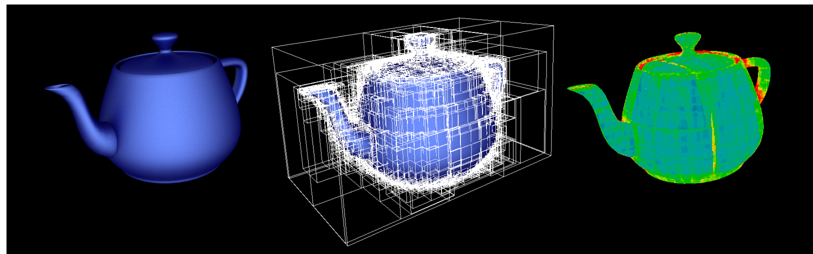


Abbildung: Modell: *Utah Teapot*, Rendering: Stefan Zellmann

Konstruktion von Hierarchien

Top-Down Konstruktion von Binärbäumen (z. B. BVHs)

```
function BaueHierarchie(Knoten)
    if Split(Knoten,AABB_L,AABB_R) then
        L ← Gruppiere(AABB_L,Knoten.Primitive)
        R ← Gruppiere(AABB_R,Knoten.Primitive)
        BaueHierarchie(L)
        BaueHierarchie(R)
    else
        Blatt(Knoten.Primitive)
    end if
end function
```

Konstruktion von Hierarchien

Funktion SPLIT(Knoten,AABB_L,AABB_R) entscheidet über Güte der BVH.

- Einfache Heuristiken wie “Median Split”.
- BVHs mit hoher Güte mittels “Surface Area Heuristik”.
  - Kostenfunktion, um zu entscheiden, ob Split oder nicht.
  - Top-Down Konstruktion: *lokale* Optima.

Konstruktion von Hierarchien

Surface Area Heuristic

Kostenfunktion setzt Fläche der Primitive im Parent Knoten zur Außenfläche der Umbox ins Verhältnis. Gegeben: N Dreiecke in einem bereits konstruierten Teilbaum mit Volumen V. Für Partitionierung L und R mit jeweils N_L und N_R Dreiecken sowie Volumen V_L und V_R, bestimme Kosten:

$$C := C_T + C_I \left(\frac{SA(V_L)}{SA(V)} N_L + \frac{SA(V_R)}{SA(V)} N_R \right). \tag{37}$$

C_T und C_I sind jeweils konstante, applikationsspezifische Kosten, um das Traversieren eines Strahls nach unten in der Hierarchie, bzw. den Schnitttest Strahl / Dreieck zu bewerten. Mit $SA()$ wird die Umfläche von Volumen V ermittelt.

Konstruktion von Hierarchien

Surface Area Heuristic

Greedy Heuristik:

1. Bestimme Split Ebenen Kandidaten (heuristisch, z. B. *Sweeping* entlang der Hauptachsen; diskrete Positionen entlang Hauptachsen ("*Binning*")).
2. Für alle Kandidaten, bestimme SAH-Kosten C.
3. Wähle Split Ebene mit niedrigstem C und bestimme AABB_L und AABB_R als Umboxen um die N_L und N_R Dreiecke.

Konstruktion von Hierarchien

Surface Area Heuristic

- **Sweeping**: bewege Kandidatenebene *stetig* durch Box.
- Wenn kostenoptimale Ebene gefunden, *partitioniere* ($O(n)$, vgl. Quicksort) Dreiecke *in place* in linken und rechten Teilbaum.
- Besonders aufwendig, wenn obere Levels des Baums aufgebaut werden, da viele Dreiecke zu partitionieren.
- Offensichtliche Vereinfachung: kein *stetiges* Sweeping, sondern orientiere Ebene an Dreieckseckpunkten.
- Weitere Vereinfachung: **Binning** - unterteile AABB in Bins und orientiere Kandidatenebenen daran. Projiziere *Mittelpunkte der Dreiecksumboxen* in Bins.

Konstruktion von Hierarchien

Surface Area Heuristic

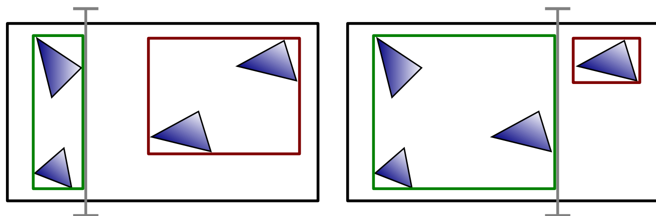


Abbildung: Zwei Kandidatenebenen. Berechne Umboxen für L und R und bewerte Kandidatenebene mit C. Kandidatenebene mit niedrigstem C determiniert Split.

Konstruktion von Hierarchien

Surface Area Heuristic

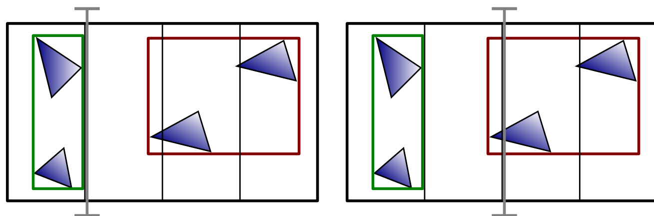


Abbildung: *Binning* diskretisiert das Verfahren. Auf höheren Ebenen weniger Dreiecke zu partitionieren.

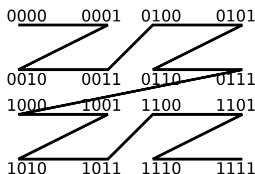
Konstruktion von Hierarchien

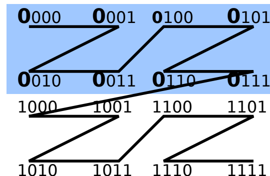
Surface Area Heuristic - Bemerkungen

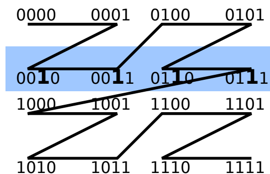
- Greedy Heuristik bestimmt nur lokale Optima. Refinement Algorithmen existieren, die nachträglich den Baum umstrukturieren (z. B. *Tree Rotations*, um Surface Area Kosten weiter zu senken ($\Rightarrow$ höhere Konstruktionskosten).
- SAH-basierte BVH Konstruktion derzeit State-of-the-Art bzgl. Traversal.

Konstruktion von Hierarchien

Bottom-Up Konstruktion







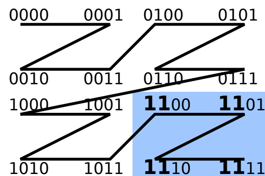


Abbildung: Morton Codes implizieren Hierarchien (vgl. Pharr, Jakob, Humphreys: Physically Based Rendering (2017)).

Konstruktion von Hierarchien

Bottom-Up Konstruktion

- Familie von Konstruktionsalgorithmen in linearer Zeit.
- *Linear BVH* (LBVH) Algorithmus:
  1. Berechne Umbox für jedes Dreieck.
  2. Berechne Liste mit 30-bit 3D Morton Codes bzgl. der *Mittelpunkte der Umboxen*.
  3. Lineares Sortieren der Mittelpunkte mit Radix Sort (z. B. auf GPU).
  4. Median Split: bestimme Split an der ersten Stelle, an der sich die signifikantesten Bits der Morton Codes unterscheiden. (Finde mittels Binärsuche).
- Niedrige Qualität, dafür sehr schnelle Konstruktion.
- LBVH Algorithmus Grundlage für Reihe von State-of-the-Art Konstruktionsalgorithmen (HLBVH, TR-BVH).

Konstruktion von Hierarchien

Bemerkungen

- Top-Down Konstruktion $\Rightarrow$ langsam, wenig Parallelismus auf Root-Level.
- Bottom-Up Konstruktion $\Rightarrow$ Bäume mit niedriger Qualität.
- Häufig hybride Verfahren, um Konstruktions- und Traversierungskosten zu balancieren:
  - Bottom-Up Konstruktion von Teilbäumen auf unterer BVH-Ebene ("*Treelets*").
  - Auf oberer Ebene, sortiere Treelets mit Surface Area Heuristik.
  - Gegenstand aktueller Forschung. Suchbegriffe, um Forschungsaufsätze zu finden: "Bonsai-BVH", "HLBVH", "TR-BVH", "ATR-BVH".

Real-Time Ray Tracing

- Erste CPU Real-Time Ray Tracer Anfang der 2000er.
- Etwa die Zeit, als SIMD Vektor Units Standard in CPUs wurden (MMX, SSE).
- Hochoptimierte Programme, zahlreiche Einschränkungen ggü. regulären Ray Tracern (damals z. B. PovRay etc.)

Real-Time Ray Tracing

1.) Objektorientierung

```
class Primitive {
    virtual bool intersect() = 0;
    virtual vec3 get_normal() = 0;
};
class Triangle {
    bool intersect() { /* ... */ }
    vec3 get_normal() { /* ... */ }
};

class Material {
    virtual vec3 shade() = 0;
    virtual vec3 sample() = 0;
};
class Metal {
    vec3 shade() { /* ... */ }
    vec3 sample() { /* ... */ }
};
```

Real-Time Ray Tracing

1.) Objektorientierung

- Problem: *Late Binding* $\Rightarrow$ kein Optimierungspotential für Compiler.
- Real-Time Ray Tracing: Optimierungspotential durch *Inlining* kleiner Funktionen.
  - Lineare Algebra Funktionen.
  - Material Shading Funktionen / BRDFs.
  - ...
- Handoptimierte Codes können um Größenordnung schneller sein als objektorientierte Codes.

Real-Time Ray Tracing

2.) Dynamisches Branching

```
class Primitive {
    virtual bool intersect() = 0;
    virtual vec3 get_normal() = 0;
};
class Triangle {
    bool intersect() { /* ... */ }
    vec3 get_normal() { /* ... */ }
};
```

Polymorphe Vererbung $\Rightarrow$ Branching.

Bei Strahl/Dreiecksschnitttest: Branching in innerster Schleife.

Real-Time Ray Tracing: eliminiere Branch, unterstütze nur Dreiecke!

Real-Time Ray Tracing

3.) SIMD

- **Strategie 1**: Traversiere *Bündel* (auch: *Strahlpakete* von N (=SIMD-Breite) Strahlen durch 3-D Szene. Problem: Divergenz - problematischer, je zufälliger einzelne Strahlen im Bündel verzweigen.
  - Ideal für Algorithmus Primaerstrahlverfolgung.
  - Whitted Algorithmus recht kohärent.
  - Problematisch: stochastisches Sampling.
- **Strategie 2**: Traversiere Einzelstrahlen, dafür BVHs mit N Bounding Boxen pro innerem Knoten und N Dreiecken pro Blatt.
- **Hybride Strategien**: Bündel für obere BVH Ebenen, Einzelstrahlen für untere Ebenen, auf denen Divergenz höher ist.