

Computergrafik

Vorlesung im Wintersemester 2020/21

Kapitel 5: Räumliche Datenstrukturen

Prof. Dr.-Ing. Carsten Dachsbacher

Lehrstuhl für Computergrafik

Karlsruher Institut für Technologie



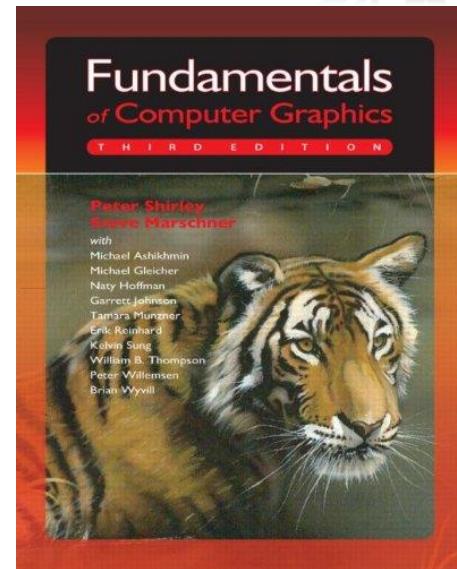
Inhalt: Räumliche Datenstrukturen

- ▶ Analyse der Kosten bei Raytracing
- ▶ Ansätze zur Beschleunigung von Raytracing
- ▶ Bounding Volumes (Hüllkörper)
- ▶ Räumliche Datenstrukturen
 - ▶ **Bounding Volume-Hierarchies**
 - ▶ **reguläre und adaptive Gitter**
 - ▶ **BSP-Bäume und kD-Bäume**



- ▶ Randnotiz: dieselben Datenstrukturen werden auch eingesetzt für
 - ▶ Kollisionserkennung und Bahnplanung in der Robotik
 - ▶ Szenengraphen, Culling und Physiksimulationen
 - ▶ als Suchbäume (z.B. Nächster Nachbar in (hochdim.) Räumen)

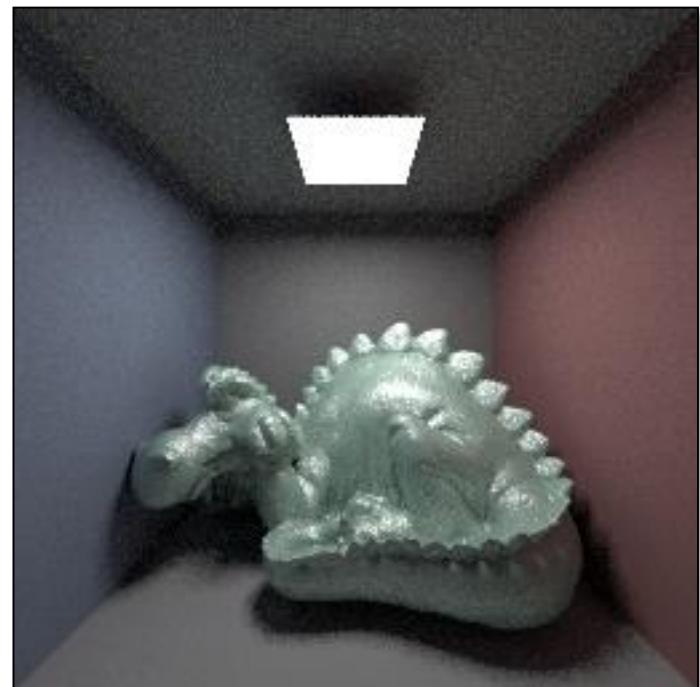
- **Fundamentals of Computer Graphics,**
P. Shirley, S. Marschner, 3rd Edition, AK Peters
→ Kapitel 12 (Data Structures for Graphics)
→ insb. Kapitel 12.3 (Spatial Data Structures)



Was haben wir bisher gelernt...

- ▶ Farbe, Bilder und ein wenig Perzeption
- ▶ Raytracing: Schnittpunktberechnung, Beleuchtungsberechnung, ...
- ▶ Texturierung

- ▶ und wie lange dauert es ein Bild mit diesem Dreiecksnetz zu berechnen?
- ▶ Path Tracing (verwandt mit Distributed Raytracing)
 - ▶ 32× stochastisches Supersampling
 - ▶ 256^2 Pixel, 480000 Dreiecke
 - ▶ Brute Force: 176462sec \approx 49h
 - ▶ nach diesem Kapitel: <80sec
(BVH 18.3sec, Rendering 59sec)
 - ▶ hier: alte Performanz-Messungen!



Was haben wir bisher gelernt...



Wie schnell kann ein Raytracer denn sein? Aktuelle Performance-Zahlen

- ▶ Datenstrukturen optimiert für besonders schnellen Strahlschuss
 - ▶ 1 bis 10 Milliarden (kohärente/inkohärente) Strahlen pro Sekunde (moderne GPU)
 - ▶ Aufbau: Sekunden für ca. 10 Mio. Dreiecke
 - ▶ CPU-GPU schwer zu vergleichen, generell etwa gleich
- ▶ Datenstrukturen optimiert für besonders schnellen Aufbau
 - ▶ Aufbau: ca. 100 Mio. Dreiecke pro Sekunde, 1 Milliarde „refit“
 - ▶ Raytracing-Performanz ca. 50%-80% von optimalen Datenstrukturen
- ▶ Disclaimer: rule of thumb!
 - ▶ alleine die Abhängigkeit von der Beschaffenheit der 3D-Szene macht genaue Aussagen unmöglich

Raytracing Pseudocode

```
class Ray {
    vec3  origin, direction; // Startpunkt und Richtung des Strahls
    float t;                // Strahlparameter
    void * object;          // Zeiger auf evtl. getroffenes Objekt
    ...
};

for ( y = 0; y < height; y++ ) {
    for ( x = 0; x < width; x++ ) {
        generateRay( ray, x, y );
        color = raytrace( ray, ... );
    }
}
```

Raytracing Pseudocode

```
vec3 raytrace( Ray *ray, ... ) {
    vec3 color = 0.0f;

    if ( !cast( ray, FLOAT_MAX ) )
        return color; // optional hier: Environment Mapping mit ray->direction

    for ( jede Lichtquelle )
        color += computeDirectLight( ... ); // -> Schattenstrahlen

    if ( Fläche ist spiegelnd ) {
        // berechne Reflexionsstrahl
        Ray reflect = ...;
        color += i.kr * raytrace( reflect, ... );
    }

    if ( Fläche ist transparent ) {
        // berechne Transmissionsstrahl
        Ray refract = ...;
        color += i.kt * raytrace( refract, ... );
    }

    return color;
}
```

Raytracing Pseudocode

```
class Ray {
    vec3 origin, direction; // Startpunkt und Richtung des Strahls
    float t;                // Strahlparameter
    void * object;          // Zeiger auf evtl. getroffenes Objekt
    ...
};

bool cast( Ray *ray, float maxDist )
{
    intersection = NULL;
    float t = maxDist;

    for ( each object ) {

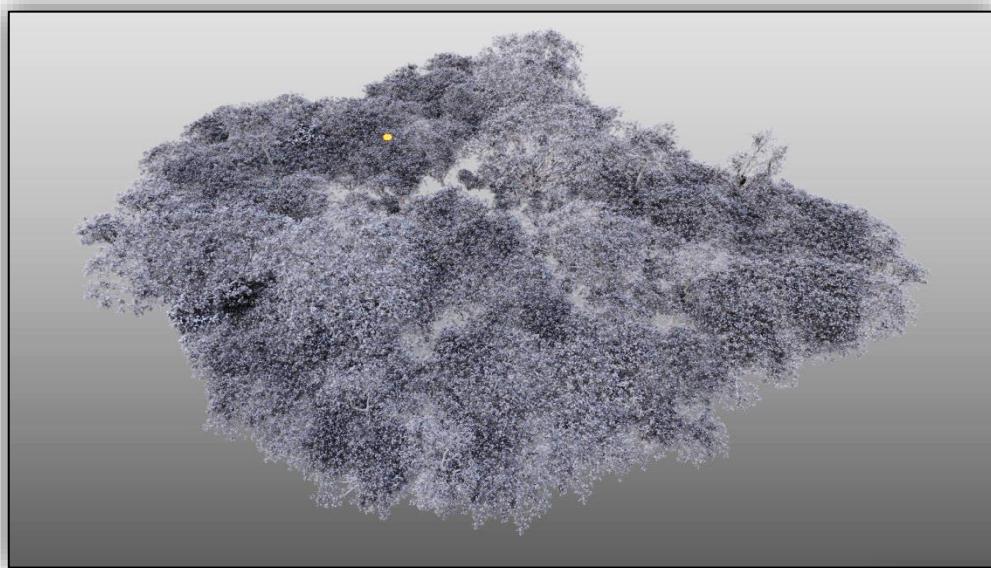
        t' = intersect( object, ray->origin, ray->direction );

        if ( t' > 0 && t' < t ) {
            intersection = object;
            t = t';
        }
    }

    ray->t = t;
    ray->object = intersection;
    return intersection != NULL;
}
```

Räumliche Datenstrukturen

Avatar (2009): 1 Petabyte Daten für die Bildsynthese



Optimierung des Raytracing

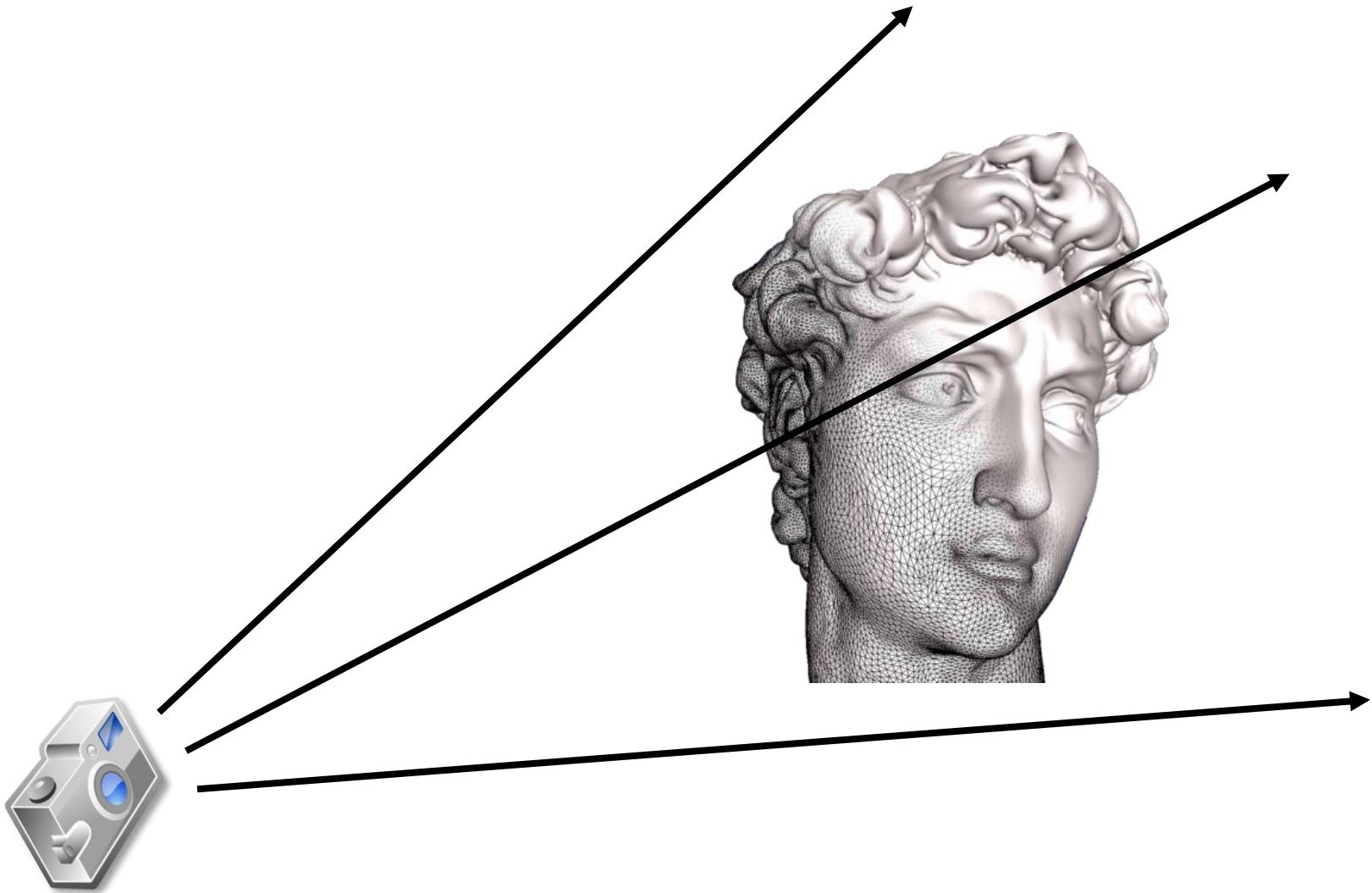
- der wesentliche Teilalgorithmus von Raytracing für die Laufzeit, so wie wir es bisher kennengelernt haben, ist:
 - ▶ finde Schnittpunkt des Strahls mit nahstem Primitiv der Szene
 - ▶ Schnitt mit allen Primitiven nimmt beinahe die gesamte Rechenzeit in Anspruch: Aufwand $O(n)$ bei n Primitiven
- Optimierungsansätze
 - ▶ schnellere Schnittalgorithmen? ... *wenig Spielraum zur Optimierung*
 - ▶ weniger Primär- und/oder Sekundärstrahlen? ... *wenig Spielraum*
 - ▶ Schnittpunkte mit „dicken“ Strahlen (Strahlenkegel/-bündel): Cone- und Beam-Tracing (nur in Verbindung mit speziellen Repräsentationen sinnvoll, behandeln wir an dieser Stelle nicht)
- Lösung: weniger Schnittberechnungen
 - ▶ vermeide Berechnungen mit weit vom Strahl entfernten Objekten
 - ▶ Raumunterteilung um potentiell geschnittene Geometrie schneller zu finden

Raumunterteilung

- Idee/Ziel:
 - ▶ teste keine Objekte, die nicht in Frage kommen
 - ▶ finde potentiell geschnittene Objekte schneller
- Datenstrukturen zur **Unterteilung von Objektgruppen und Räumen**
 - ▶ Hüllkörper/Bounding Volumes
 - ▶ Hüllkörper-/Bounding Volume-Hierarchien (BVH)
 - ▶ Gitter und verschachtelte Gitter
 - ▶ Oktalbäume (Octrees)
 - ▶ Binary-Space-Partition (BSP) Trees oder kD-Bäume
- wir gehen im Folgenden der Einfachheit halber davon aus, dass alle Objekte/Primitive in Weltkoordinaten platziert sind/sein können:
eine Datenstruktur für alle Objekte

Beschleunigung des Raycasting

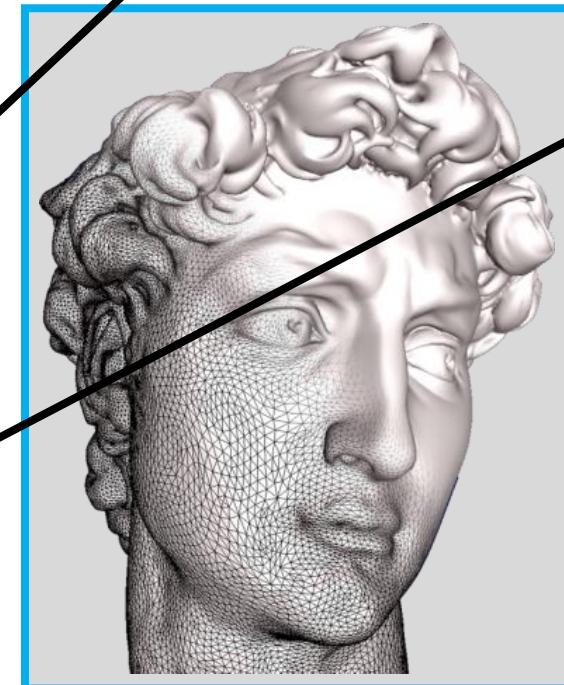
Reduziere die Anzahl der Strahl-Primitiv-Schnitttests



Beschleunigung des Raycasting

Reduziere die Anzahl der Strahl-Primitiv-Schnitttests

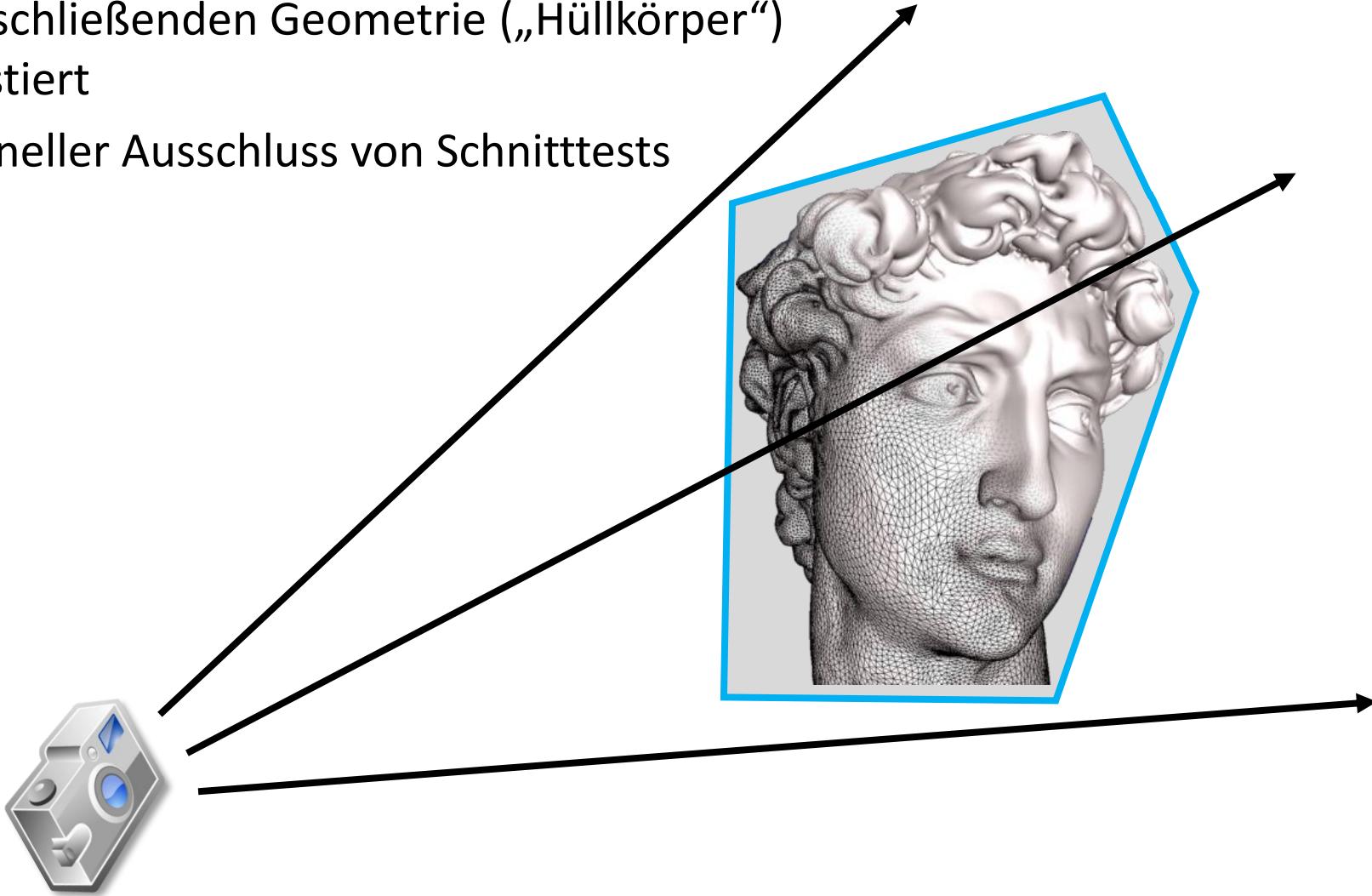
- ▶ überprüfe zuerst, ob ein Schnitt mit einer (konservativ-großen) einschließenden Geometrie („Hüllkörper“) existiert
- ▶ schneller Ausschluss von Schnitttests



Beschleunigung des Raycasting

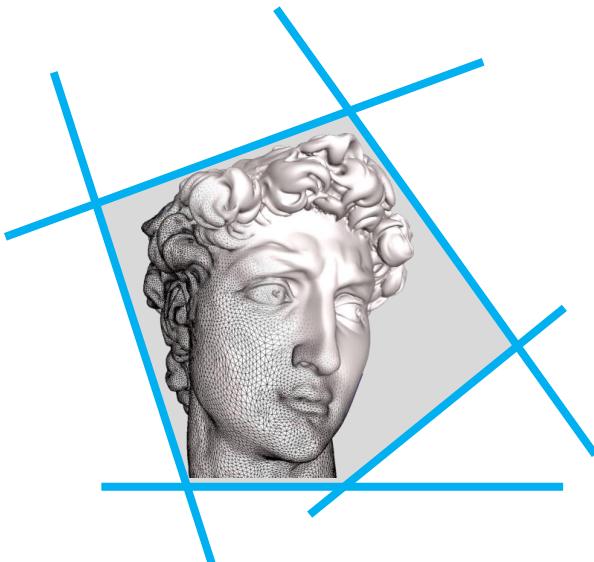
Reduziere die Anzahl der Strahl-Primitiv-Schnitttests

- ▶ überprüfe zuerst, ob ein Schnitt mit einer (konservativ-großen) einschließenden Geometrie („Hüllkörper“) existiert
- ▶ schneller Ausschluss von Schnitttests

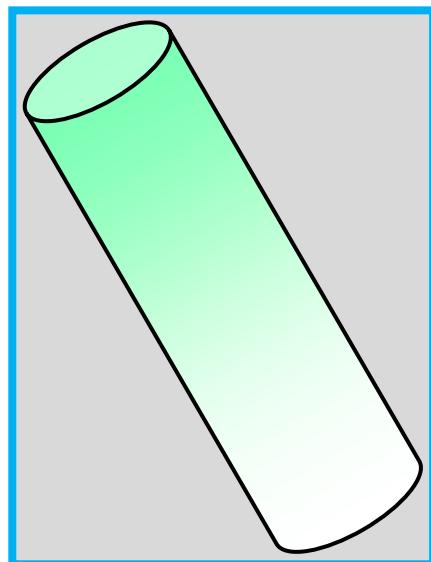


Konservative Hüllkörper

- Hüllkörper (engl. Bounding Volumes)
 - sollen möglichst enganliegend/klein sein, um wenige Falschpositive zu erzeugen
 - schnelle Schnittberechnung
- lohnt nur, wenn die enthaltene Geometrie entsprechend detailliert ist

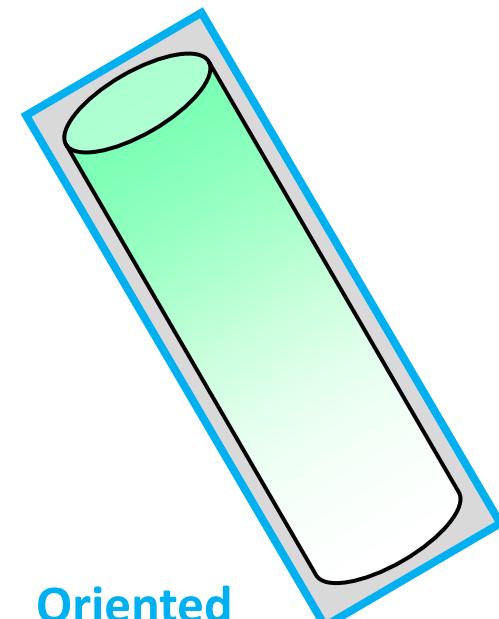


allgemeine konvexe Region (Kombination von begrenzenden Halbräumen und Spezialfall „Slabs“)



Axis-Aligned
Bounding
Box (AABB)

Bounding
Sphere

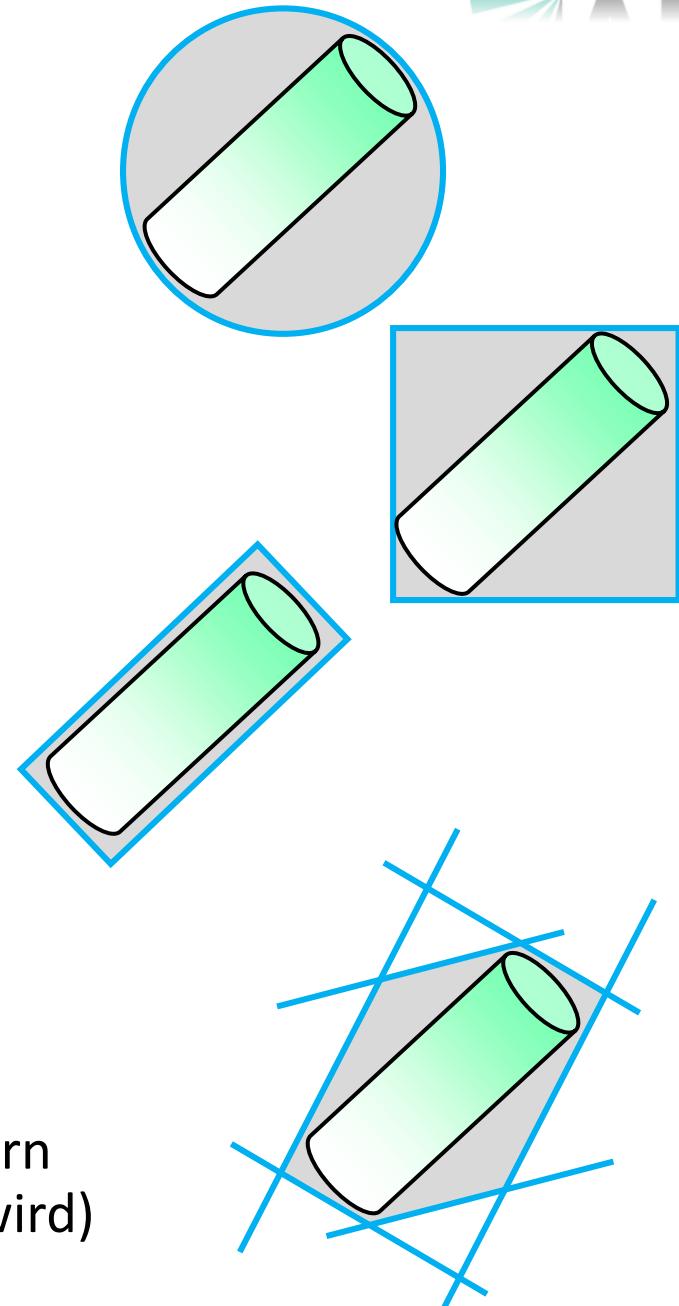


Oriented
Bounding Box (OBB)

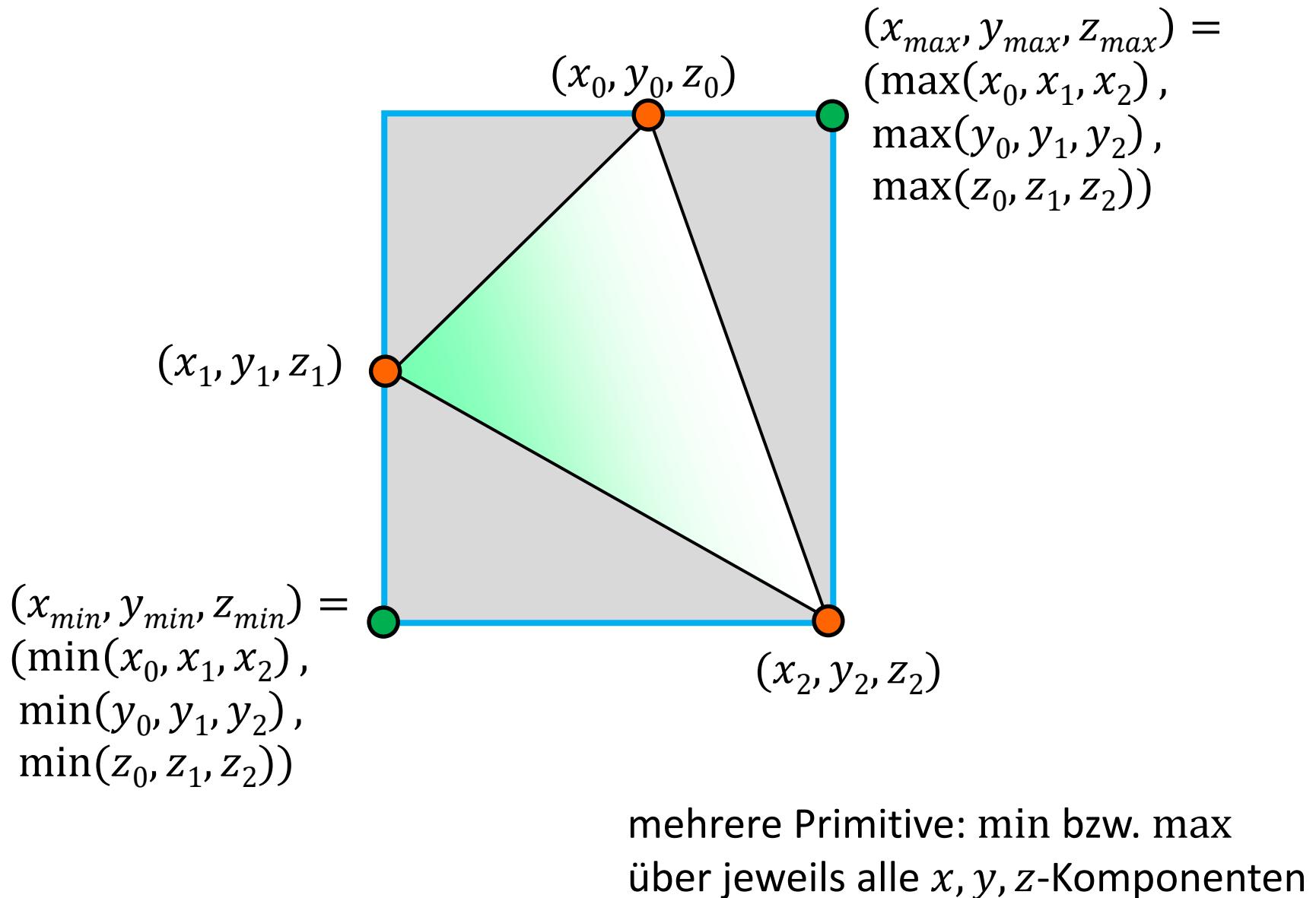
Optimierung: Bounding Volumes

Hüllkörper, Bounding Volumes

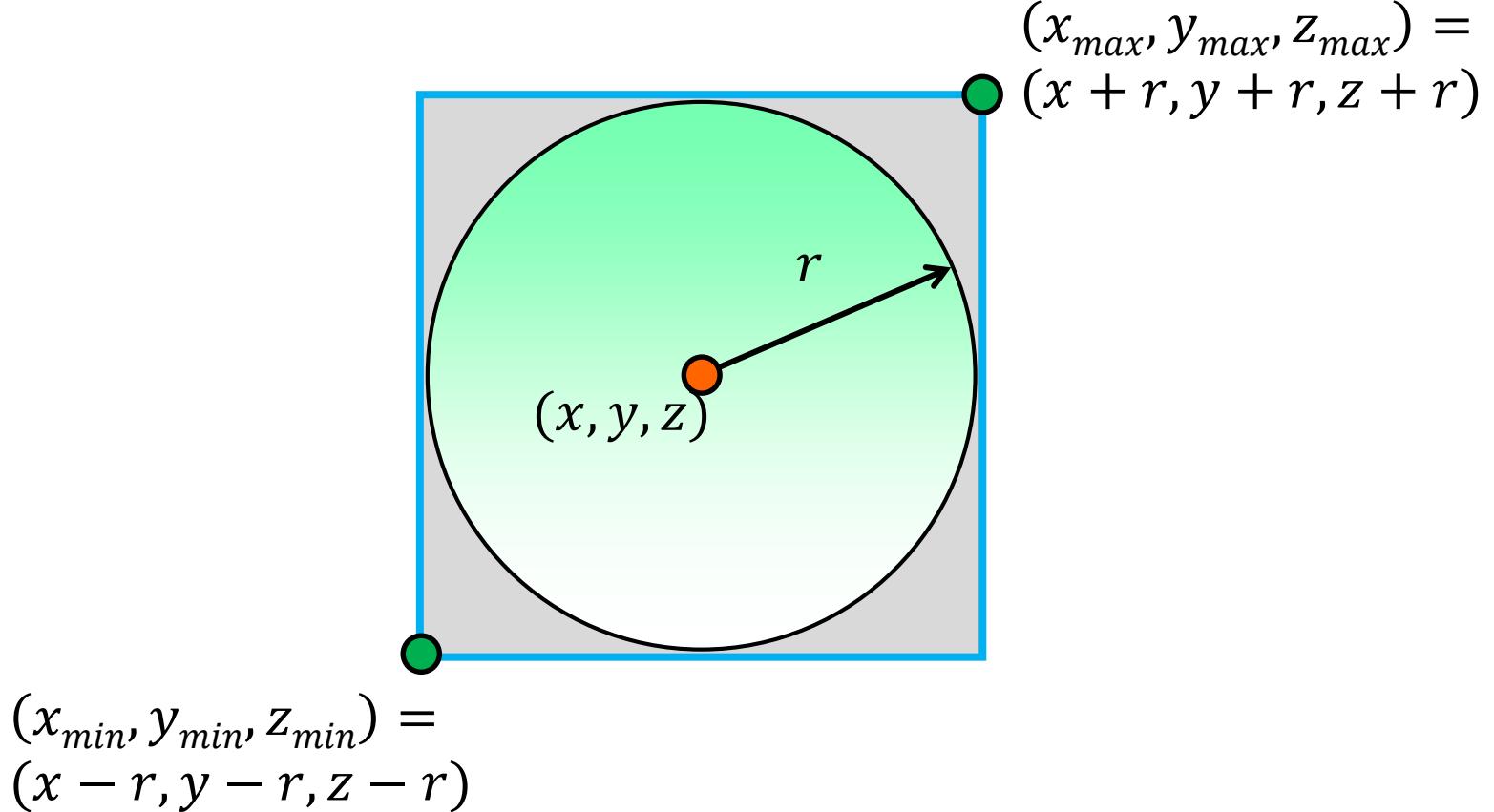
- ▶ Kugel
 - ▶ schneller Schnittalgorithmus
 - ▶ oft schlechte Effizienz, da zu groß
- ▶ achsenparallele Box (AABB)
 - ▶ einfache Berechnung (min/max)
 - ▶ wichtigster Hüllkörper
- ▶ orientierte Bounding Box (OBB)
 - ▶ aufwändiger Berechnung
(Hauptkomponentenanalyse der Menge von Eckpunkten)
- ▶ Slabs
 - ▶ Spezialfall: Schnitt von Paaren paralleler Halbebenen
 - ▶ gute Effizienz, schnelle Berechnung (sofern nicht die global beste Lösung gefordert wird)



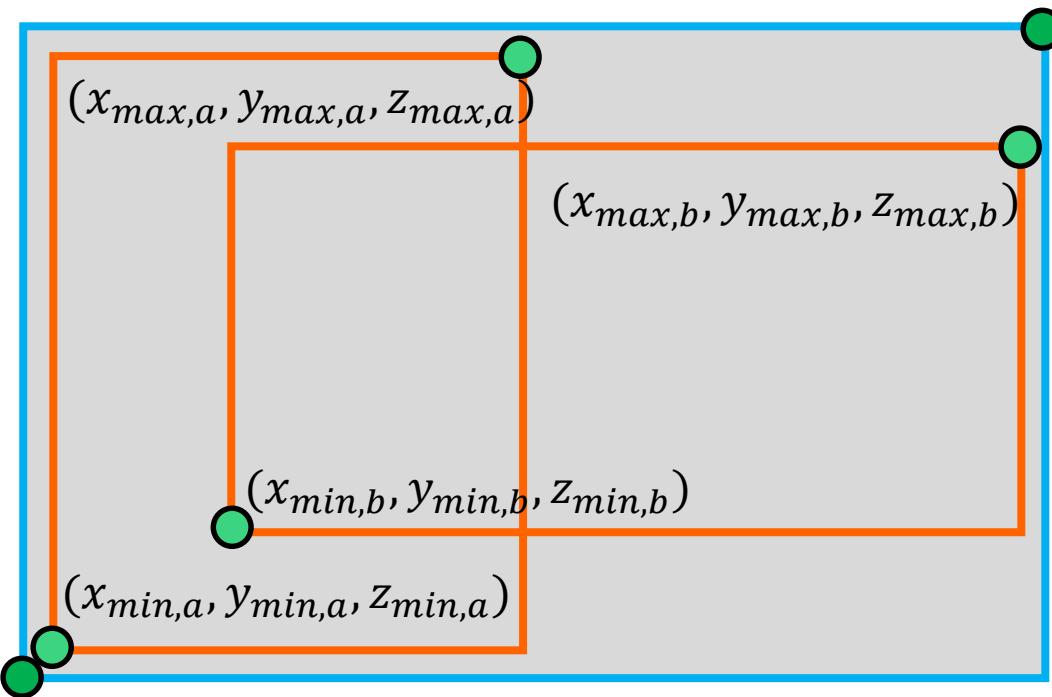
Axis-Aligned Bounding Box eines Dreiecks



Axis-Aligned Bounding Box einer Kugel



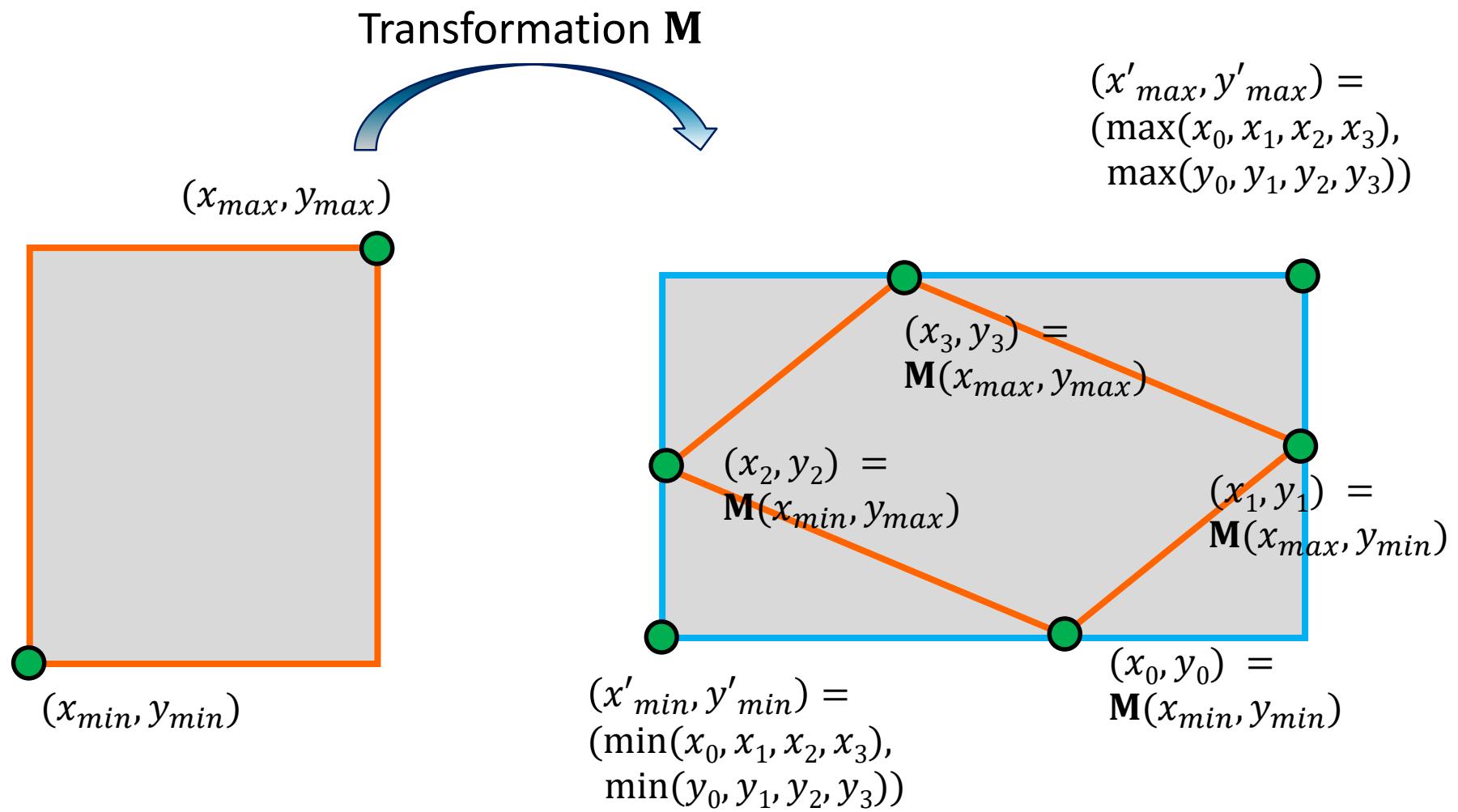
AABB einer Gruppe von AABBs



$$(x_{max}, y_{max}, z_{max}) = (\max(x_{max,a}, x_{max,b}), \max(y_{max,a}, y_{max,b}), \max(z_{max,a}, z_{max,b}))$$

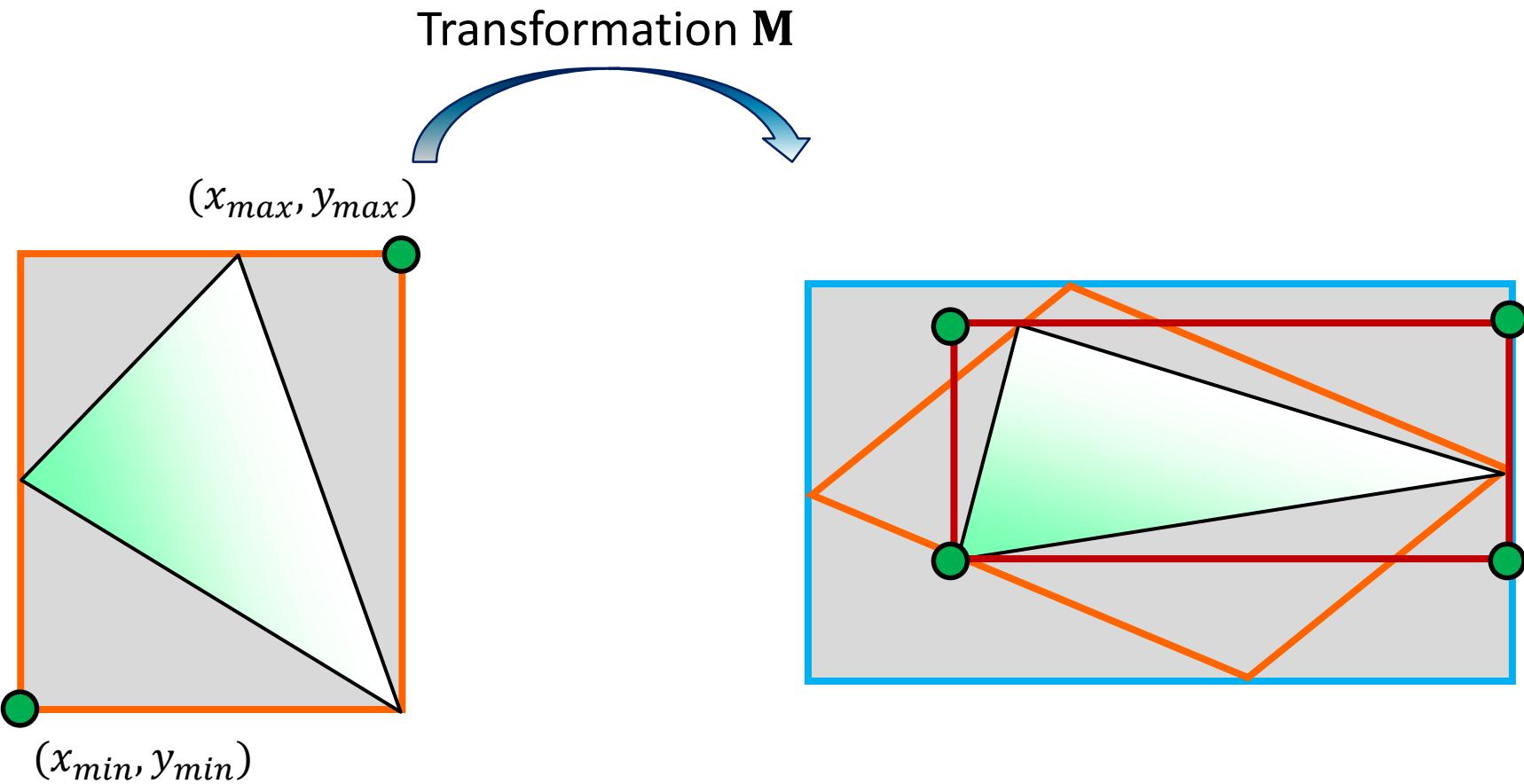
$$(x_{min}, y_{min}, z_{min}) = (\min(x_{min,a}, x_{min,b}), \min(y_{min,a}, y_{min,b}), \min(z_{min,a}, z_{min,b}))$$

AABB einer transformierten AABB



AABBs transformierter Dreiecksnetze

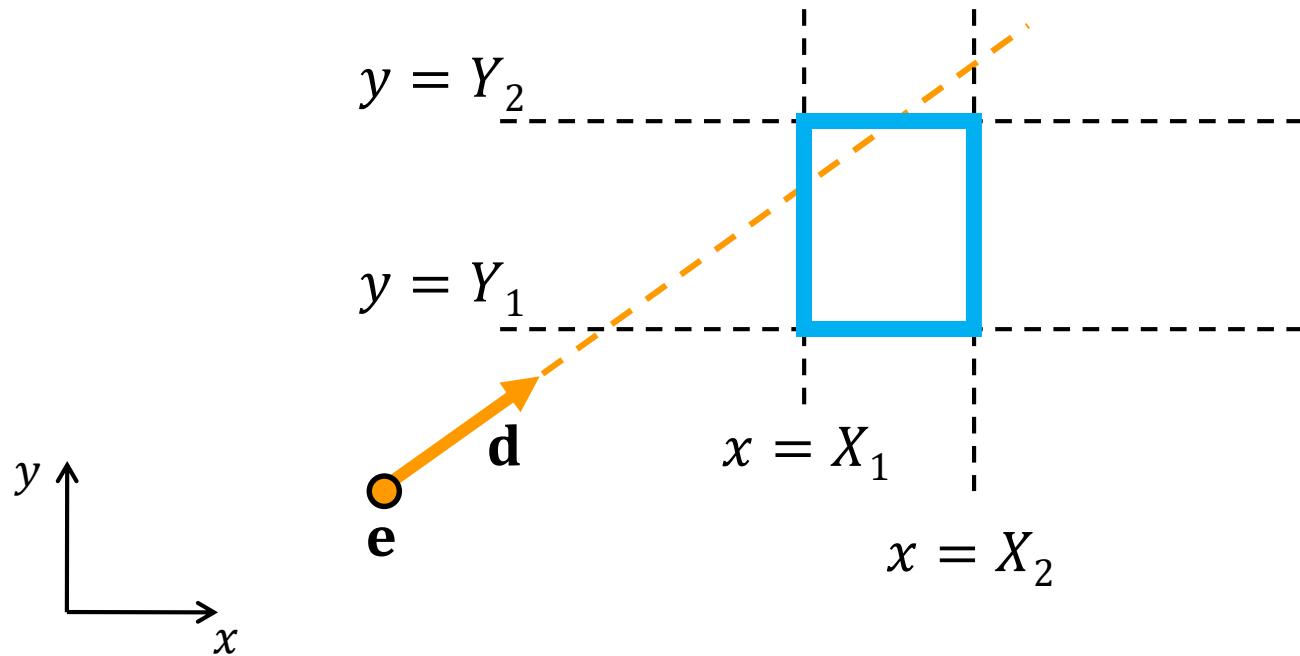
- ▶ kleinere AABB durch Transformation der Eckpunkte des Dreiecks(netzes) und Neubestimmung der Minima und Maxima
- ▶ erhöhte Kosten für Bestimmung der AABB, aber i.d.R. deutlich kleinerer Hüllkörper und somit weniger Falschpositive



Bounding Volume: AABB

Schnittberechnung Strahl-AABB

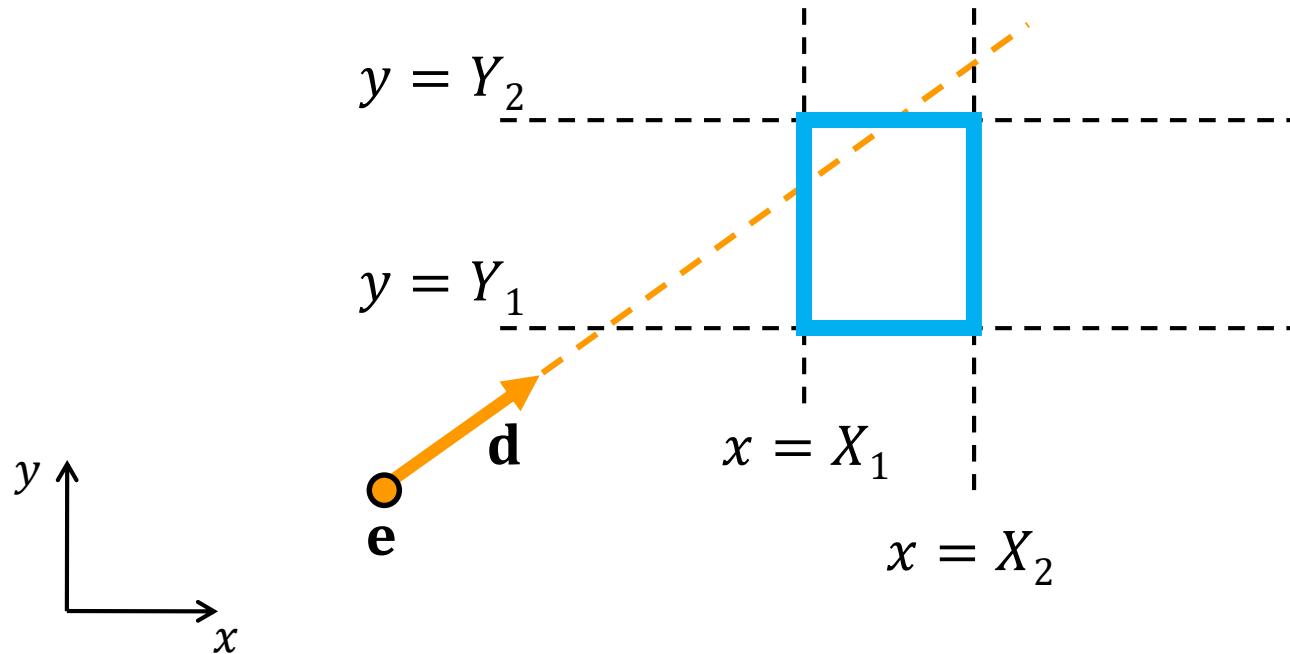
- ▶ AABB: $(X_1, Y_1, Z_1) \rightarrow (X_2, Y_2, Z_2)$
- ▶ Strahl: $\mathbf{r}(t) = \mathbf{e} + t\mathbf{d}$
- ▶ wichtigster Hüllkörper, später sind wir nicht nur interessiert, **ob** es einen Schnitt mit der AABB gibt, sondern auch wo sie liegen



Bounding Volume: AABB

Schnittberechnung Strahl-AABB

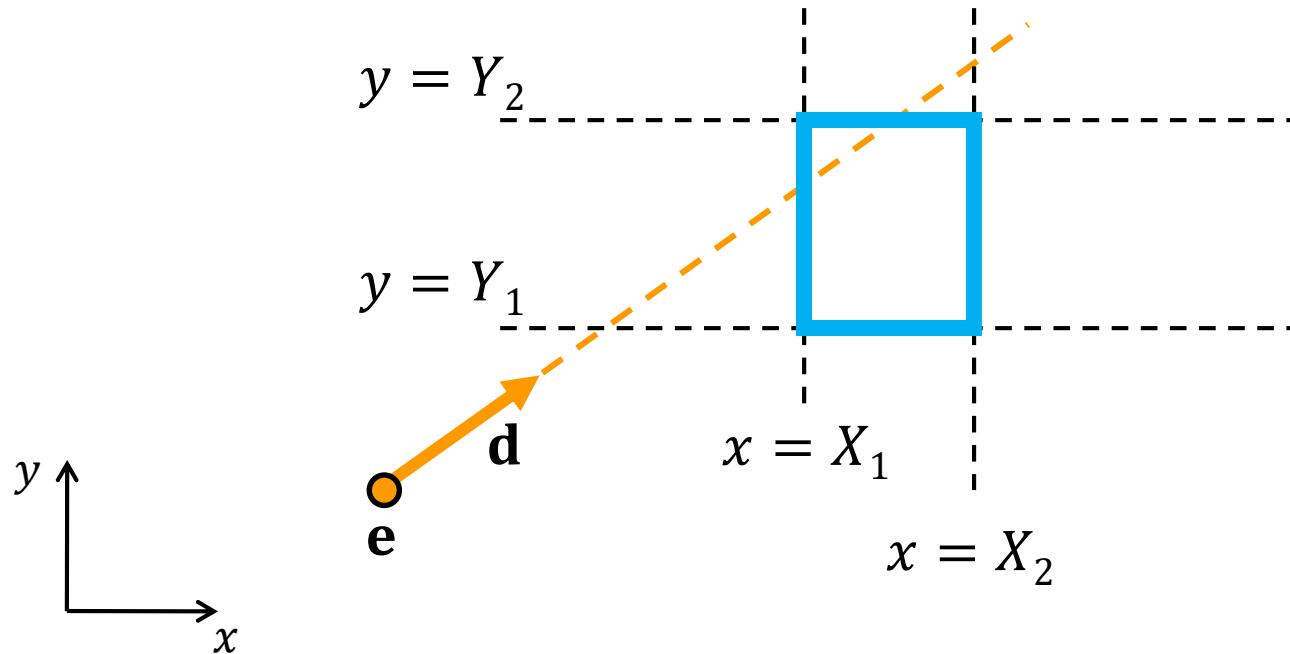
- ▶ AABB: $(X_1, Y_1, Z_1) \rightarrow (X_2, Y_2, Z_2)$
- ▶ Strahl: $\mathbf{r}(t) = \mathbf{e} + t\mathbf{d}$
- ▶ naiver Ansatz:
 - ▶ 6 Ebenengleichungen: berechne alle Schnittpunkte
 - ▶ teste, ob einer der Schnittpunkte *auf* der Box liegt



Bounding Volume: AABB

Optimierungen

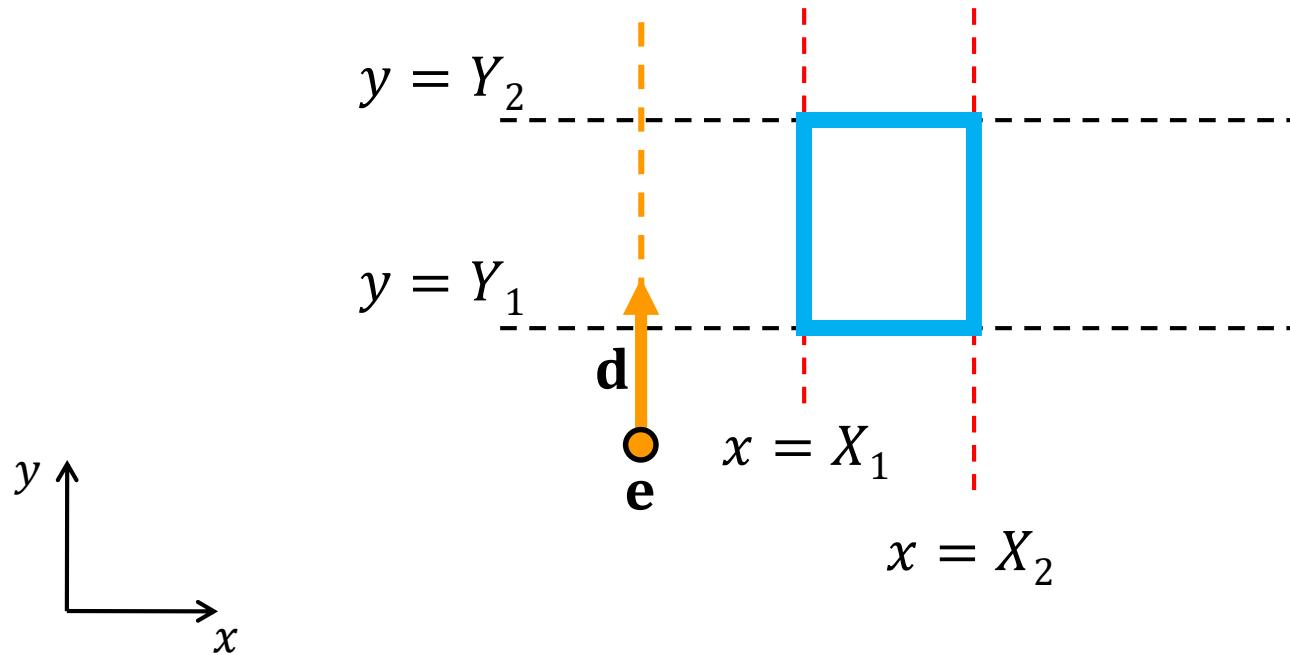
- ▶ der Schnitttest Strahl-AABB wird i.d.R. sehr oft benötigt
→ hier lohnt es sich zu optimieren!
- ▶ Beobachtung:
 - ▶ jeweils 2 Ebenen besitzen dieselbe Normale → führe Berechnungen für jede Dimension jeweils paarweise aus
 - ▶ Normalen sind achsenparallel: nur eine Komponente ungleich 0



Bounding Volume: AABB

Schnittberechnung Strahl-AABB: Test, ob Strahl und Box parallel

- ▶ vor den Ebenenschnitten zunächst Ausschlusstests...
- ▶ wenn $d_x = 0$ (Strahl parallel zu yz -Ebene) $\wedge (e_x < X_1 \vee e_x > X_2)$
⇒ kein Schnitt
- ▶ analog für d_y und d_z



Bounding Volume: AABB

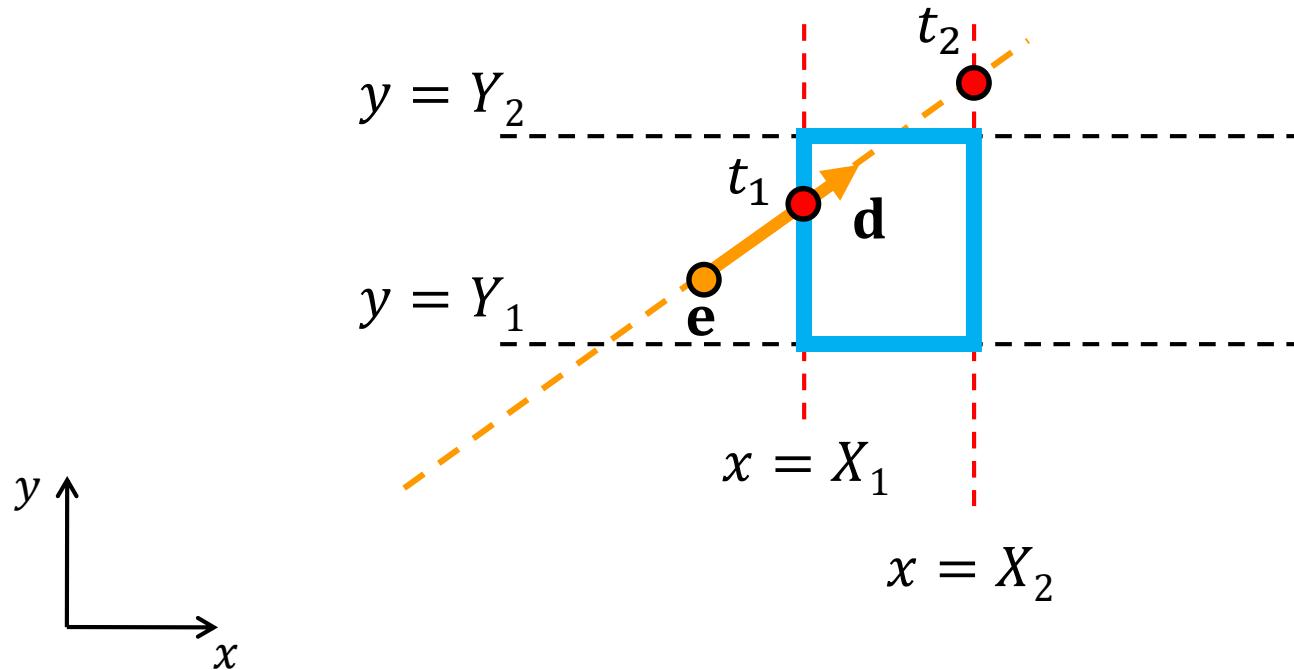
Schnittberechnung für jede Dimension

► berechne Entfernung zu den Ebenenschnitten t_1 und t_2

$$\blacktriangleright t_1 = (X_1 - e_x)/d_x$$

$$\blacktriangleright t_2 = (X_2 - e_x)/d_x$$

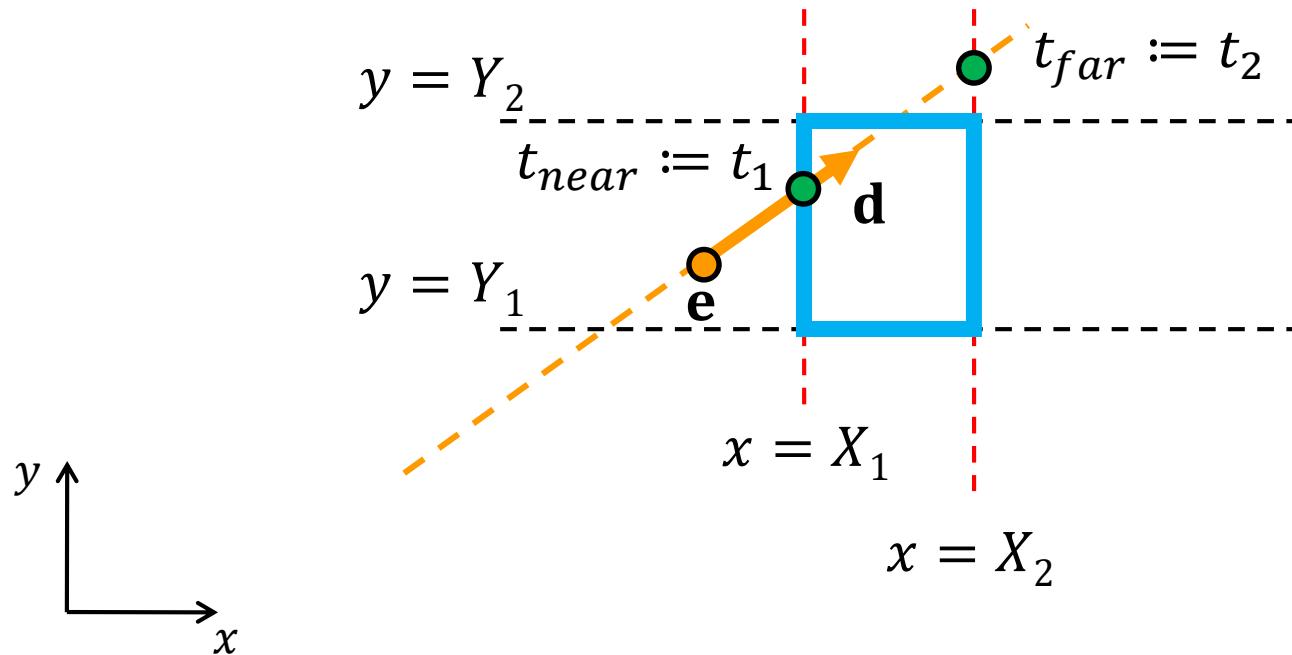
► diese Berechnung ist Spezialfall des Strahl-Ebenen-Schnitt
(Normalen entlang der Achsen), allgemein: $t = \frac{d - \mathbf{e} \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$



Bounding Volume: AABB

Schnittberechnung für jede Dimension

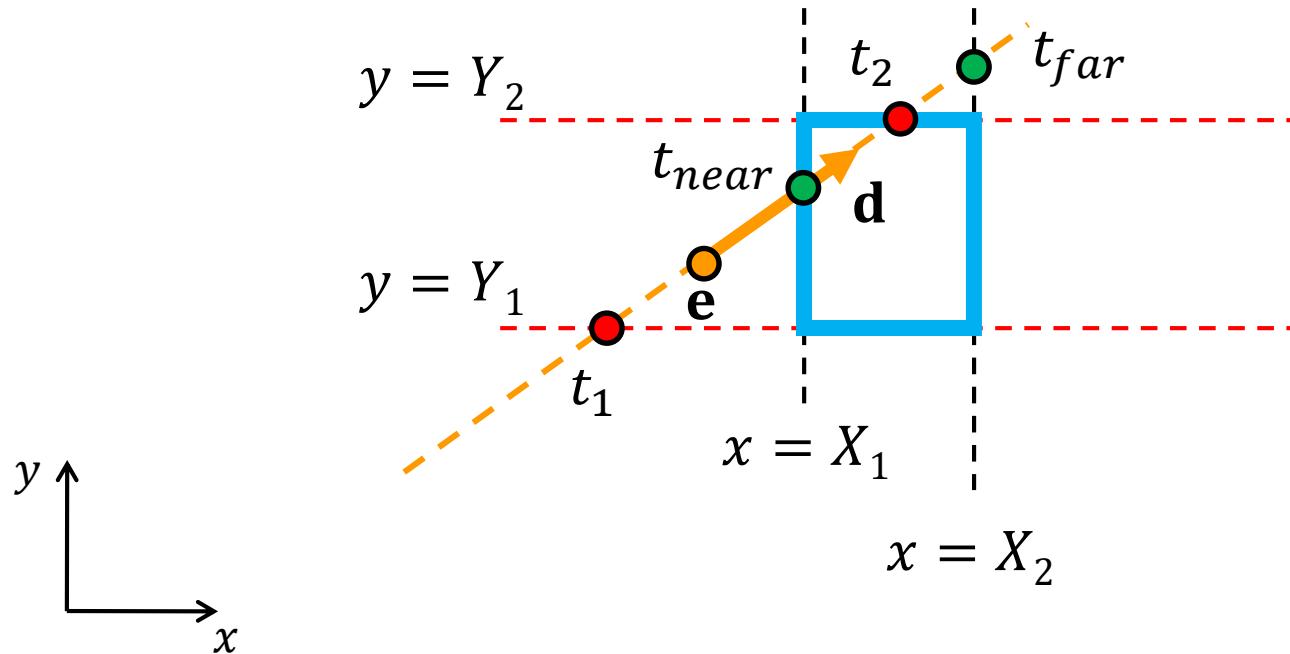
- berechne Entfernung zu den Ebenenschnitten t_1 und t_2
 - Annahme $t_1 \leq t_2$ (ansonsten werden die Werte getauscht)
 - $t_{near} := t_1$ und $t_{far} := t_2$
 - $[t_{near}; t_{far}]$ soll am Ende das Intervall des Strahls beschreiben, das mit der AABB überlappt



Bounding Volume: AABB

Schnittberechnung Strahl-AABB: t_{near} und t_{far} bestimmen

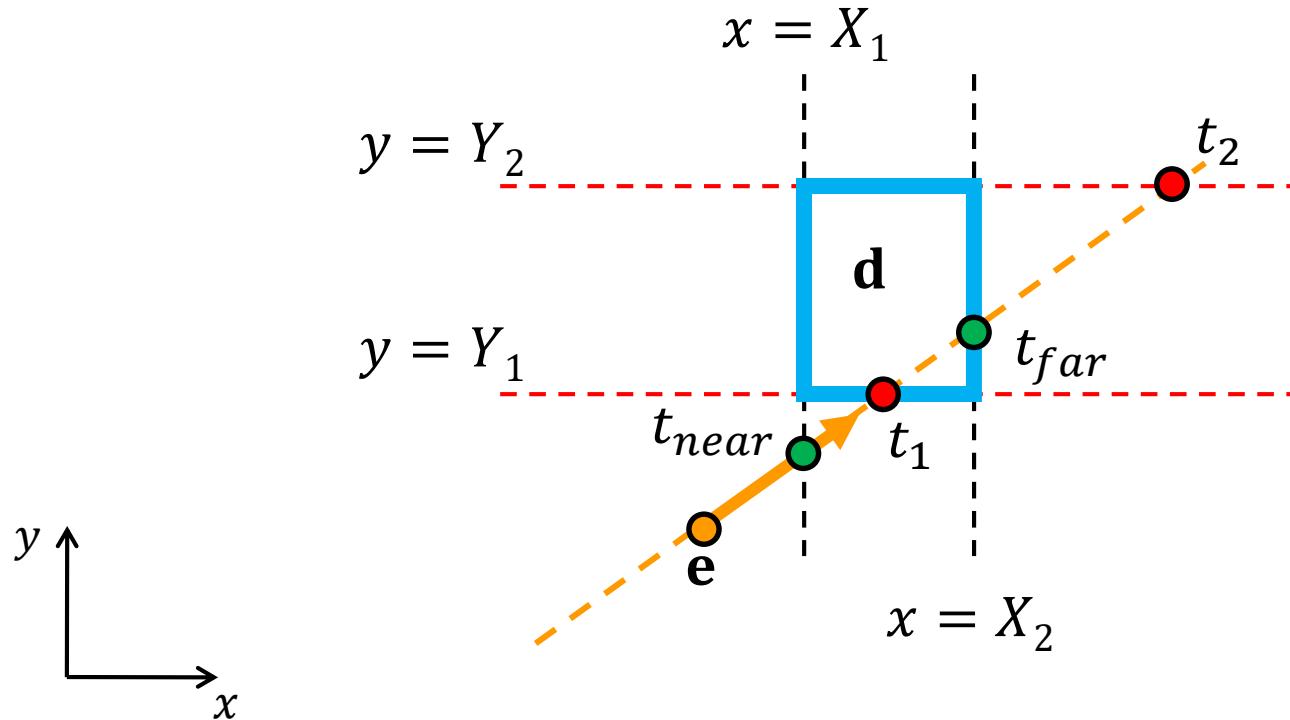
- ▶ Berechnung der Schnitte mit den weiteren Ebenenpaaren und Mitführen des nahsten und entferntesten Schnitts *mit der AABB*
 - ▶ Annahme $t_1 \leq t_2$
 - ▶ wenn $t_1 > t_{near}$, dann $t_{near} = t_1$ (hier nicht der Fall)
 - ▶ wenn $t_2 < t_{far}$, dann $t_{far} = t_2$



Bounding Volume: AABB

Schnittberechnung Strahl-AABB: t_{near} und t_{far} bestimmen

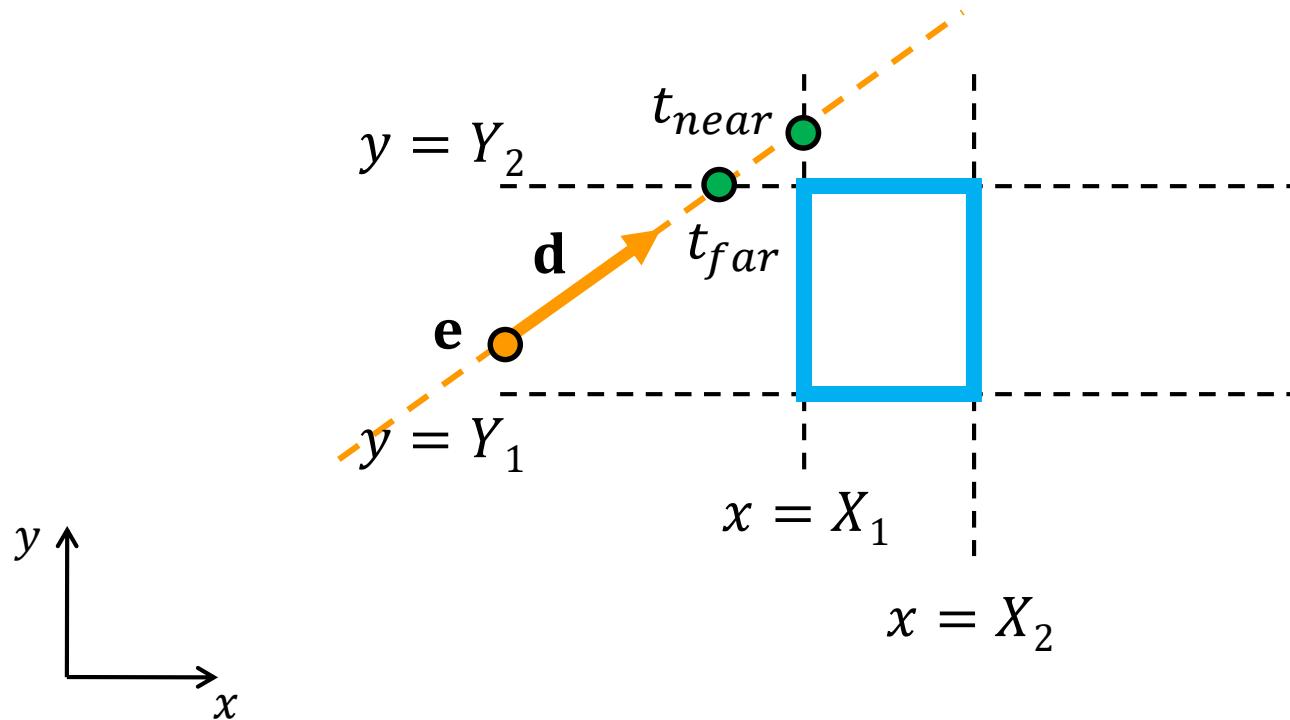
- ▶ Berechnung der Schnitte mit den weiteren Ebenenpaaren und Mitführen des nahsten und entferntesten Schnitts *mit der AABB*
 - ▶ wenn $t_1 > t_{near}$, dann $t_{near} = t_1$
 - ▶ wenn $t_2 < t_{far}$, dann $t_{far} = t_2$ (hier nicht der Fall)
- ▶ am Ende beschreibt $[t_{near}; t_{far}]$ den Überlapp des Strahls mit der AABB



Bounding Volume: AABB

Schnittberechnung Strahl-AABB: existiert ein Schnittpunkt?

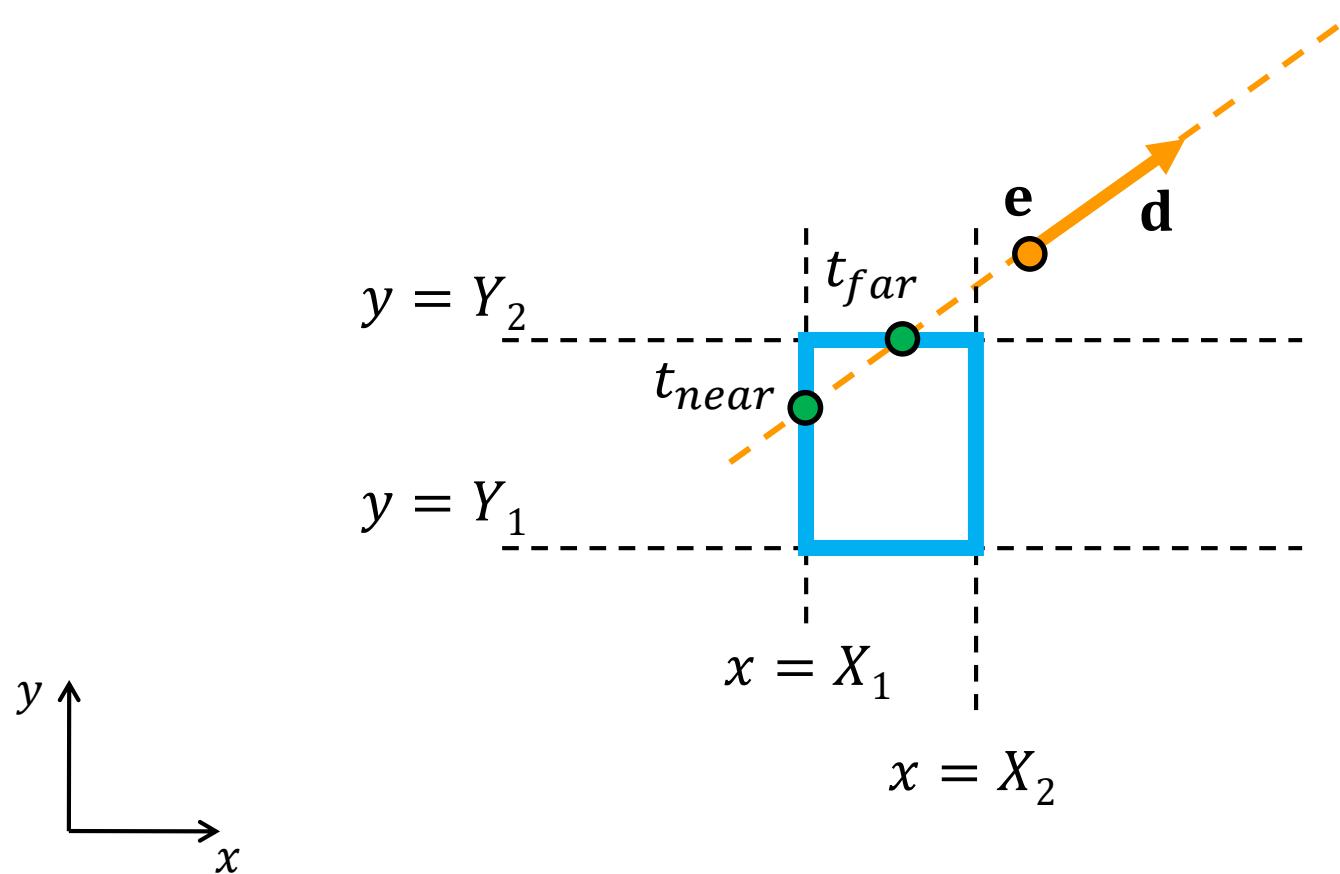
- ▶ am Ende beschreibt $[t_{near}; t_{far}]$ den Überlapp des Strahls mit der AABB, außer in den folgenden Fällen...
- ▶ wenn $t_{near} > t_{far}$ wird die Box nicht getroffen
- ▶ hier: der Strahl verlässt das Intervall $[Y_1; Y_2]$ bevor er $[X_1; X_2]$ erreicht



Bounding Volume: AABB

Schnittberechnung Strahl-AABB: Box hinter dem Strahlursprung?

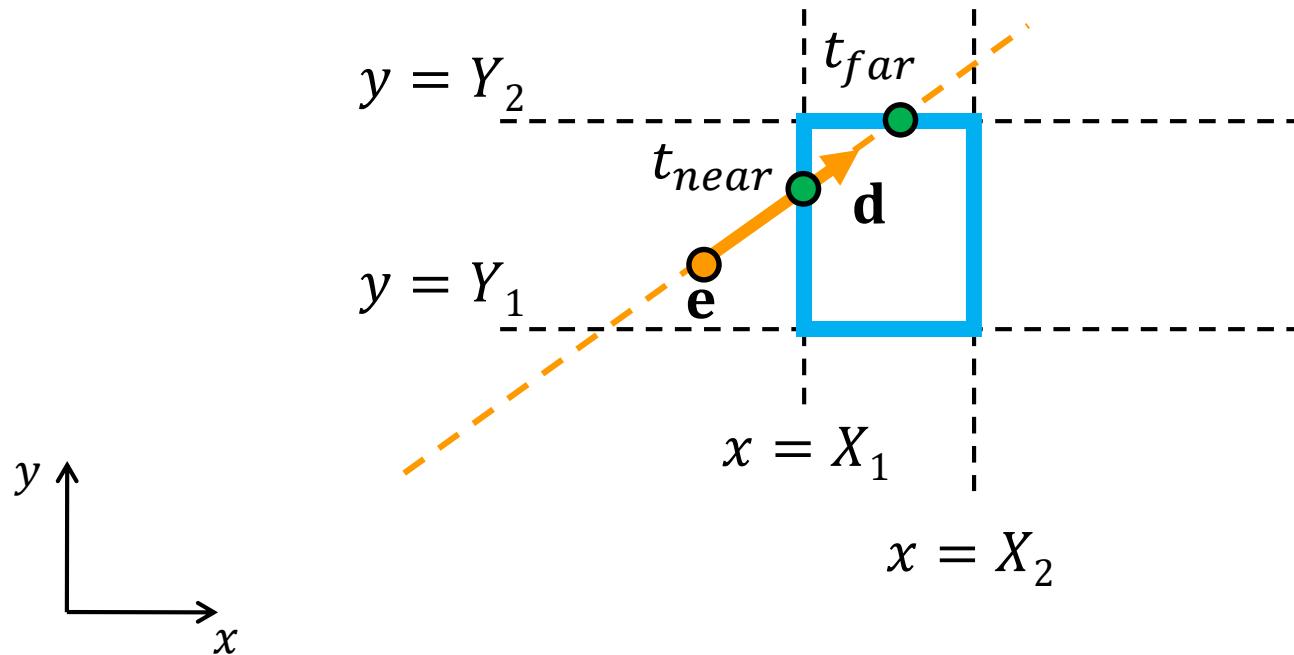
- wenn $t_{far} < 0$ ist die Box hinter dem Strahl



Bounding Volume: AABB

Richtigen Schnittpunkt zurückliefern

- ▶ wenn $t_{near} > 0 \rightarrow$ nahster Schnittpunkt bei t_{near}
- ▶ sonst: Schnittpunkt bei t_{far} (Strahl beginnt innerhalb der AABB)
- ▶ t_{near} und t_{far} helfen uns also die Fälle elegant zu unterscheiden
- ▶ auch wenn wir gerade nur daran interessiert zu testen, **ob** es einen Schnitt mit der AABB gibt, brauchen wir das Strahlsegment später noch



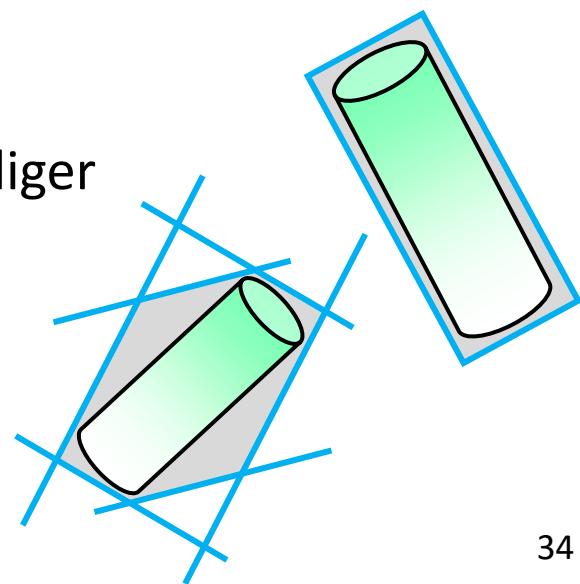
Bounding Volume: AABB

Zusammenfassung Schnittberechnung Strahl-AABB

- ▶ für jede Dimension (hier für x gezeigt):
 wenn $d_x = 0$ (Strahl parallel zu yz -Ebene) $\wedge (e_x < X_1 \vee e_x > X_2)$
 \Rightarrow kein Schnitt
- ▶ für jede Dimension: berechne t_1 und t_2
 - ▶ $t_1 = (X_1 - e_x)/d_x$ $t_2 = (X_2 - e_x)/d_x$
 - ▶ wenn $t_1 > t_2$ tausche t_1 und t_2 (benötigt wenn $d_x, d_y, d_z < 0$)
 - ▶ initialisiere bzw. aktualisiere t_{near} und t_{far}
 (nahster und entferntester Schnitt bis zu diesem Zeitpunkt)
 - ▶ wenn $t_1 > t_{near}$, dann $t_{near} = t_1$
 - ▶ wenn $t_2 < t_{far}$, dann $t_{far} = t_2$
- ▶ wenn $t_{near} > t_{far}$ \rightarrow Box nicht getroffen
- ▶ wenn $t_{far} < 0$ \rightarrow Box hinter dem Strahl
- ▶ wenn $t_{near} > 0$ \rightarrow nahster Schnitt bei t_{near}
- ▶ sonst \rightarrow nahster Schnitt bei t_{far}

Einfache Optimierungen

- ▶ $1/d_x, 1/d_y$ und $1/d_z$ können einmal pro Strahl berechnet werden und für alle AABB-Tests verwendet werden
- ▶ Loop-Unrolling, Schleifen sind teuer
- ▶ vermeide t_{near} und t_{far} Vergleich für die erste Dimension
- ▶ **SIMD: teste mehrere Strahlen/AABBS parallel**
- ▶ Verallgemeinerung des Schnitttests auf OBBs und Slabs möglich
 - ▶ parallele Ebenenpaare
 - ▶ konvexe Form des Hüllkörpers
 - ▶ nur Berechnung von t_1 und t_2 ist etwas aufwändiger (wegen der beliebigen Orientierung der Ebenen)



Wie viele Hüllkörper sollen es denn sein?

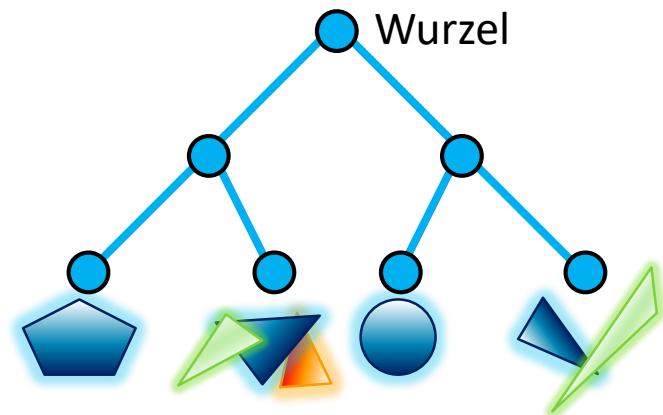
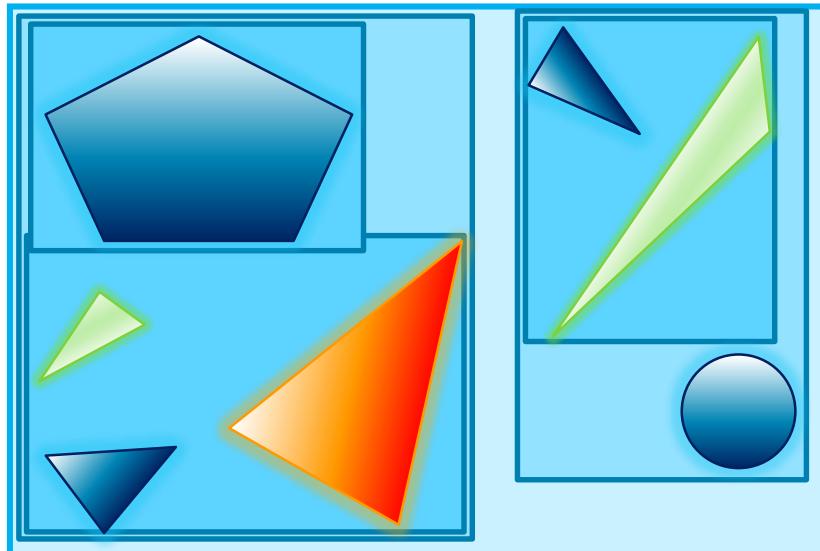
... und wie viel Geometrie soll jeweils eingeschlossen werden?



Optimierung: Bounding Volume-Hierarchien

Grundidee: Hierarchie von einschließenden Hüllkörpern

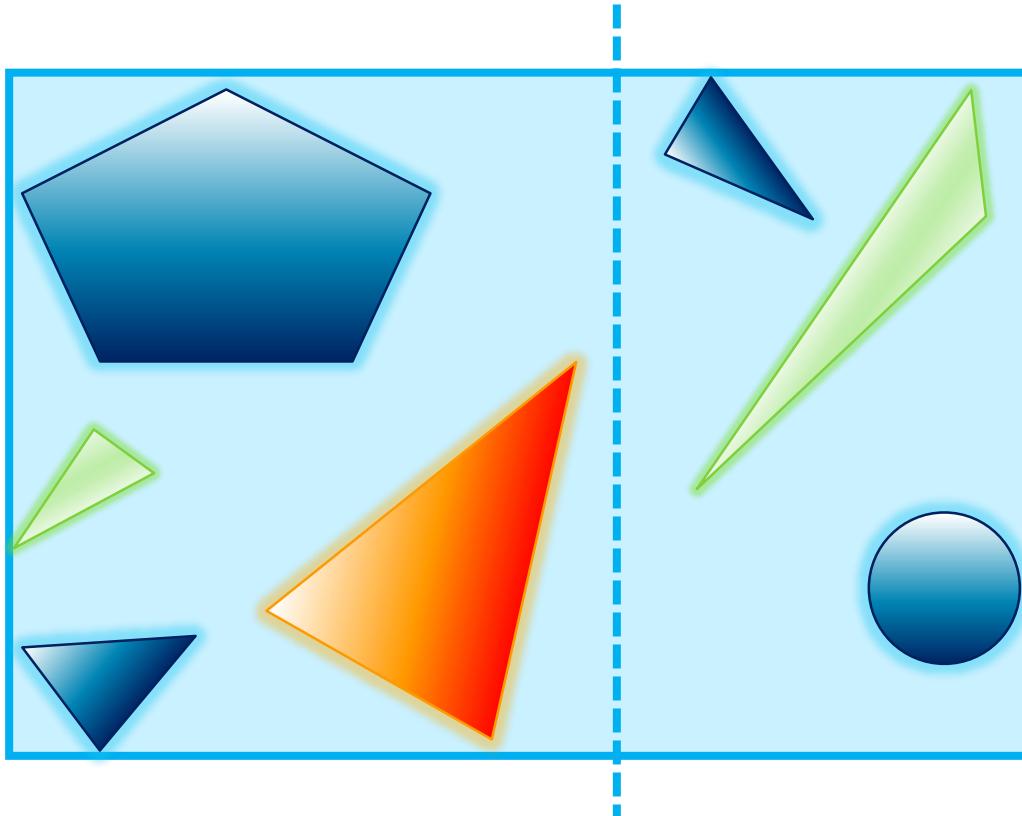
- ▶ automatisches Zusammenfassen von (Gruppen von) Objekten/Primitiven, oder rekursiv Unterteilen der Primitive in Gruppen
- ▶ Vorteile:
 - ▶ Adaptivität
 - ▶ schnellere Suche (ganze Gruppen können ausgeschlossen werden)
- ▶ Bsp. Hierarchie mit AABBs (generell auch andere Hüllkörper möglich)



Bounding Volume-Hierarchie

Aufbau der Hierarchie (top-down)

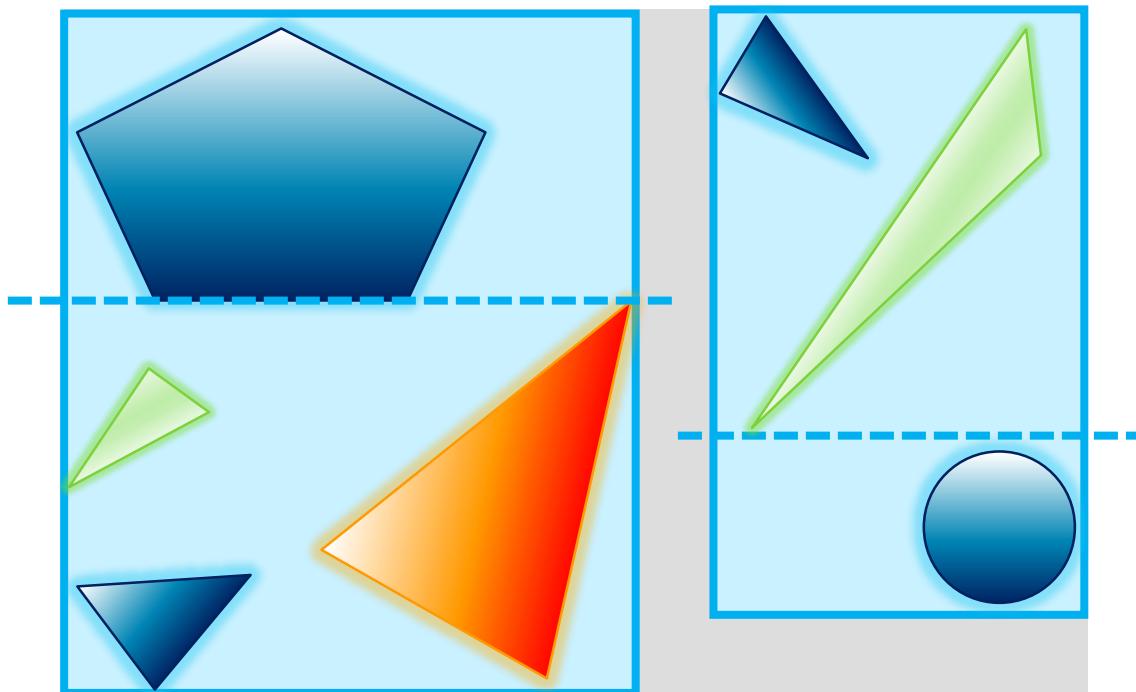
- ▶ bestimme die gemeinsame Bounding Box aller Objekte
- ▶ teile die Objekte in **zwei Gruppen** auf
- ▶ die Linie ----- dient zur Illustration der Aufteilung



Bounding Volume-Hierarchie

Aufbau der Hierarchie

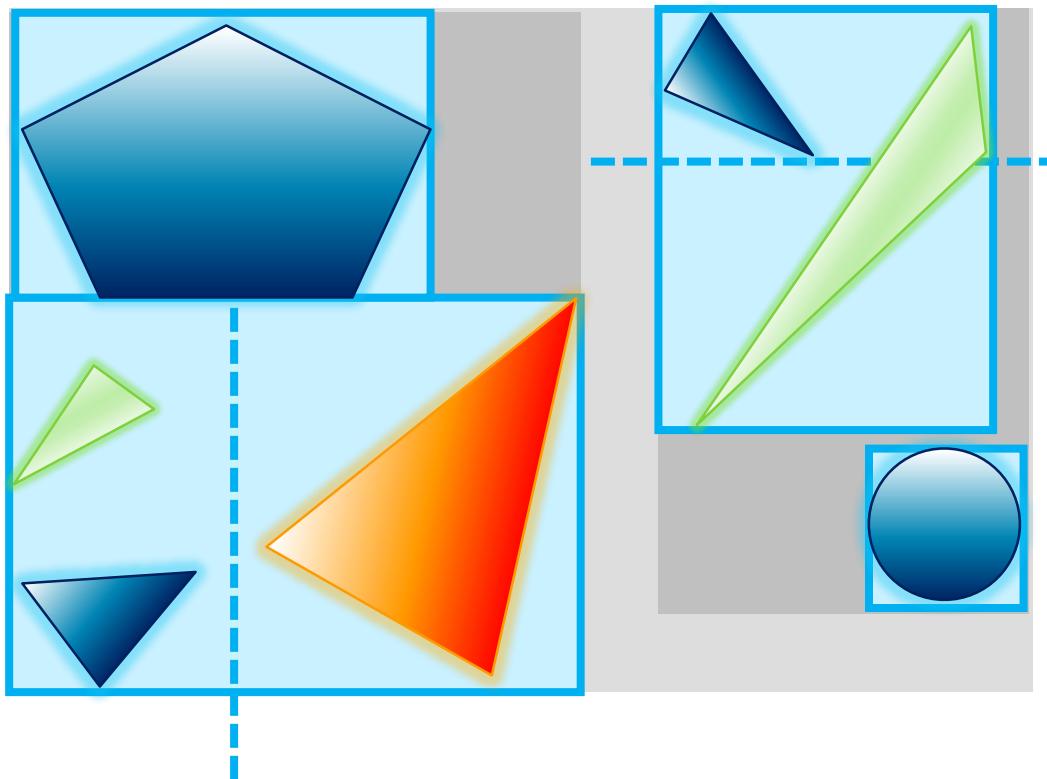
- ▶ bestimme die Bounding Box der Objekte jeder Gruppe
- ▶ teile die Objekte jeweils wieder in **zwei Gruppen** auf
- ▶ verfahre so rekursiv weiter



Bounding Volume-Hierarchie

Aufbau der Hierarchie

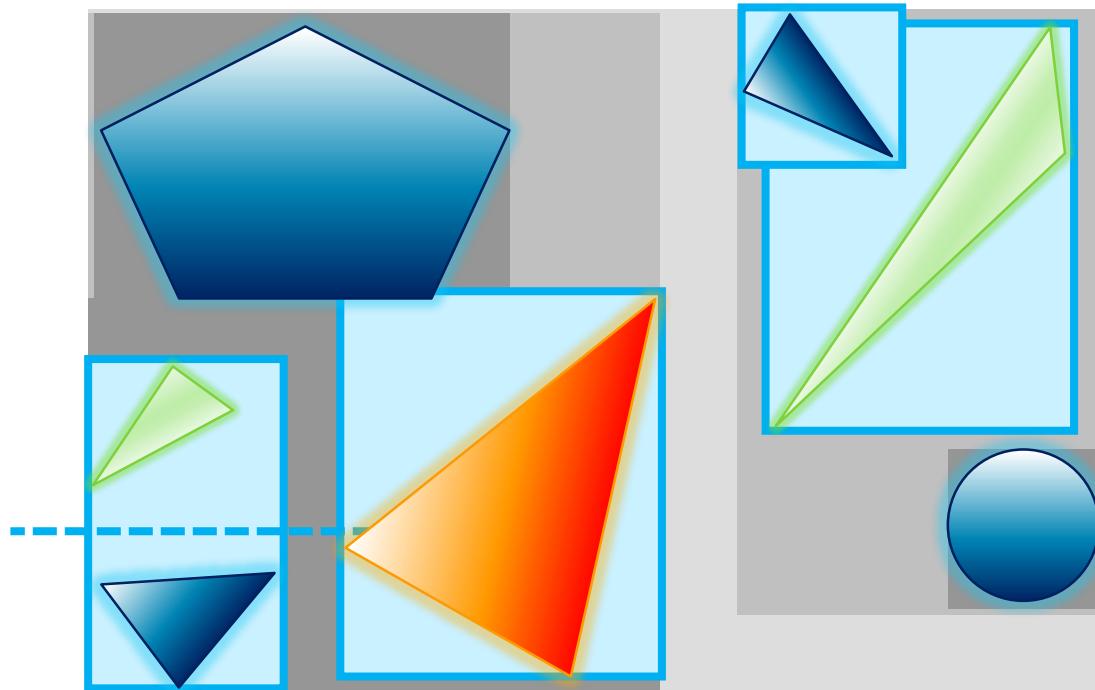
- ▶ bestimme die Bounding Box der Objekte
- ▶ teile die Objekte jeweils wieder in **zwei Gruppen** auf
- ▶ verfahre so rekursiv weiter



Bounding Volume-Hierarchie

Aufbau der Hierarchie

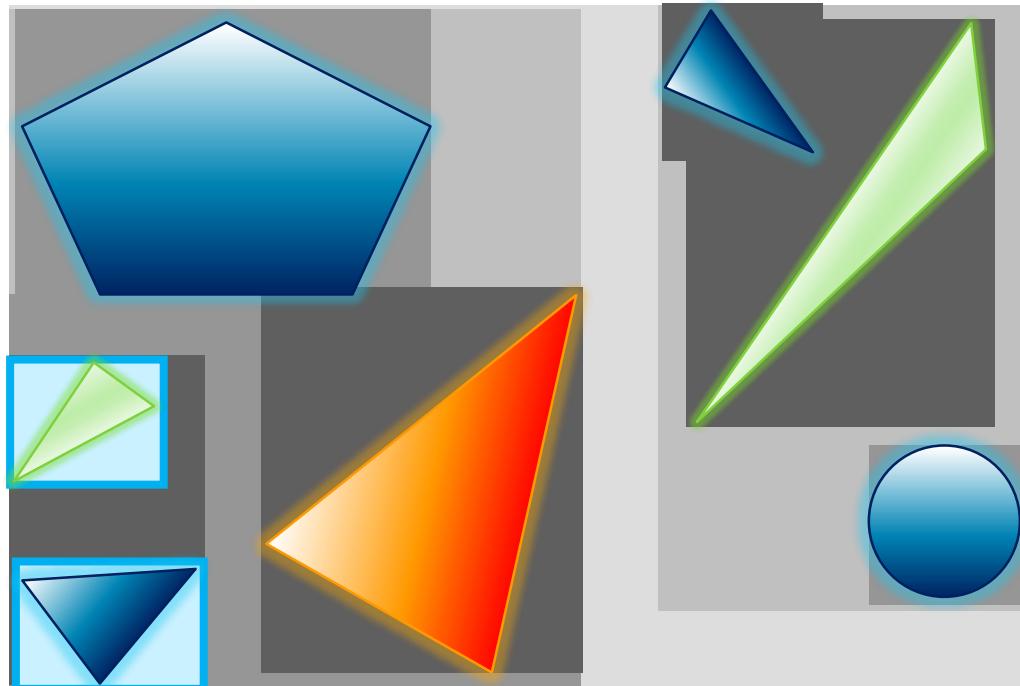
- ▶ bestimme die Bounding Box der Objekte
- ▶ teile die Objekte jeweils wieder in **zwei Gruppen** auf
- ▶ verfahre so rekursiv weiter



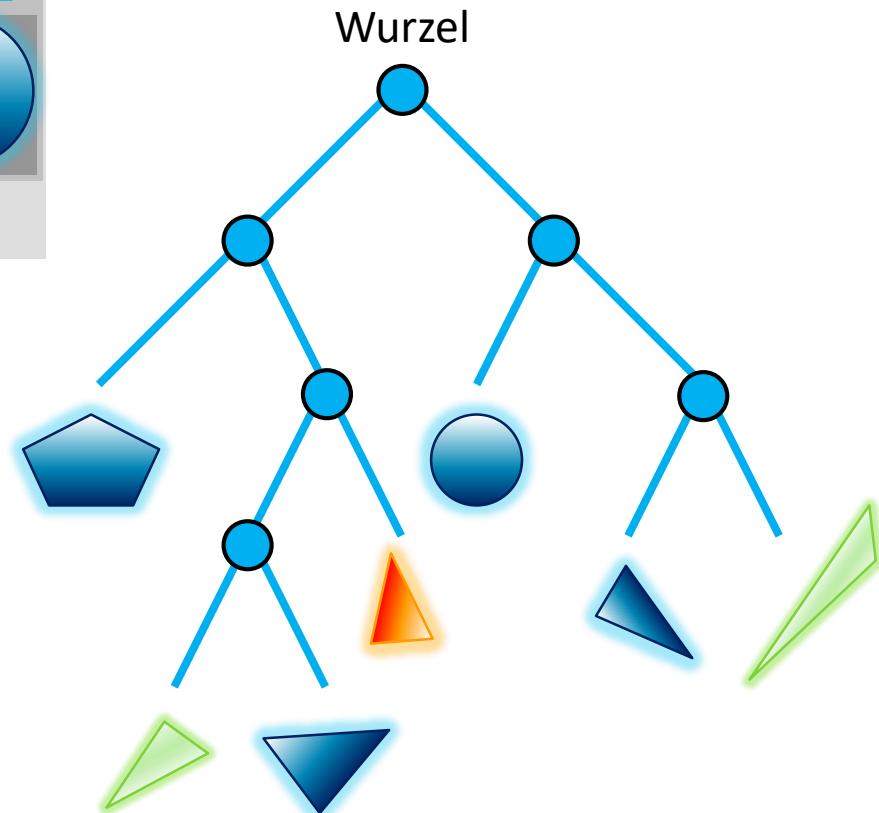
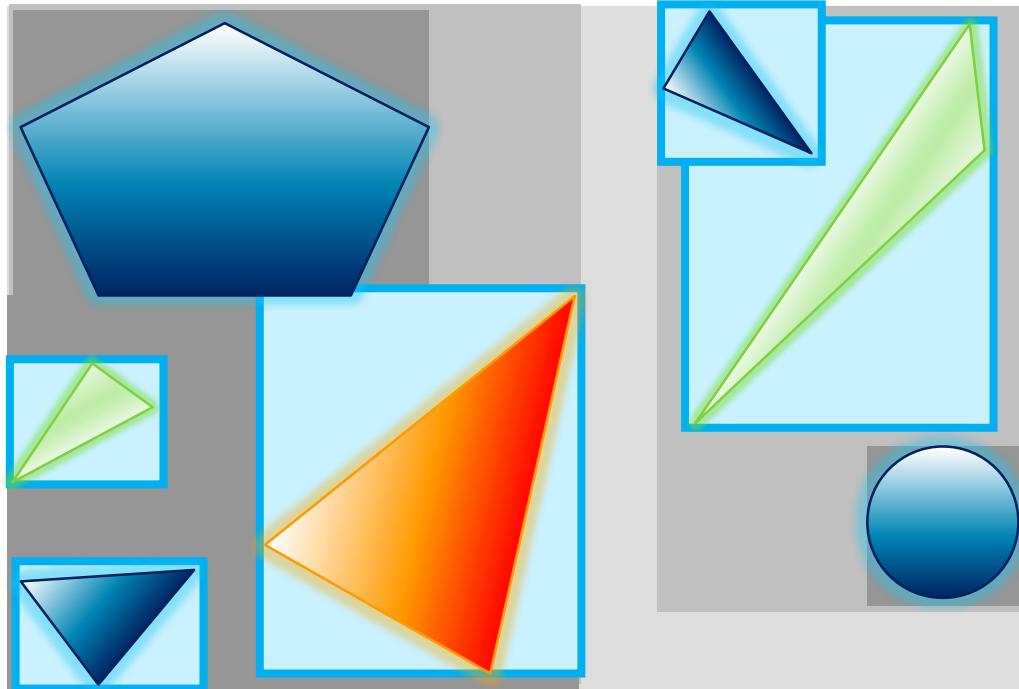
Bounding Volume-Hierarchie

Aufbau der Hierarchie

- ▶ bestimme die Bounding Box der Objekte
- ▶ teile die Objekte jeweils wieder in **zwei Gruppen** auf
- ▶ verfahre so rekursiv weiter (z.B. bis nur noch m Objekte in jeder Gruppe enthalten sind, hier $m = 1$)

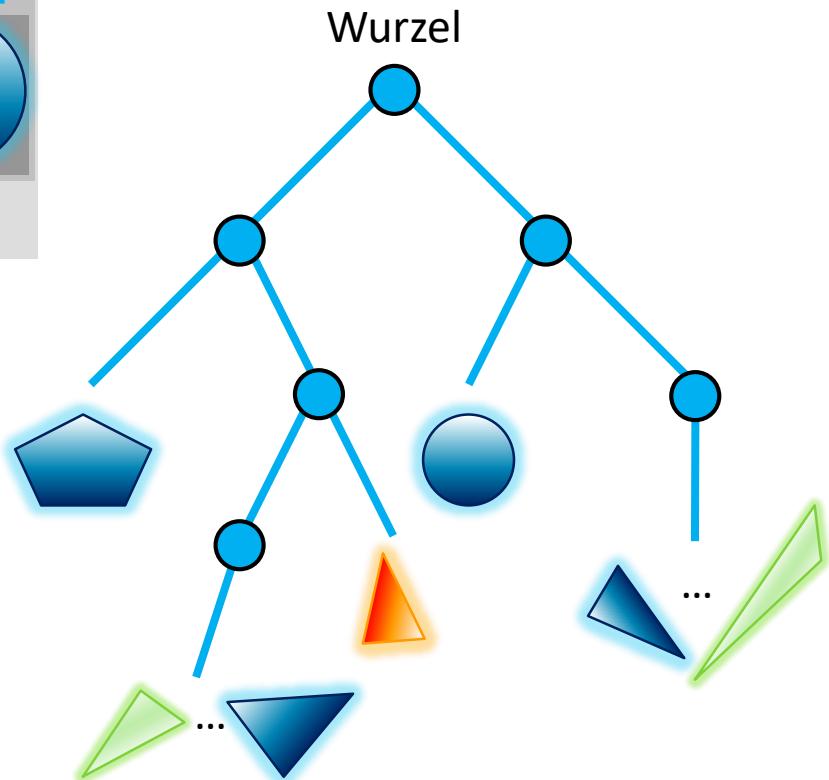
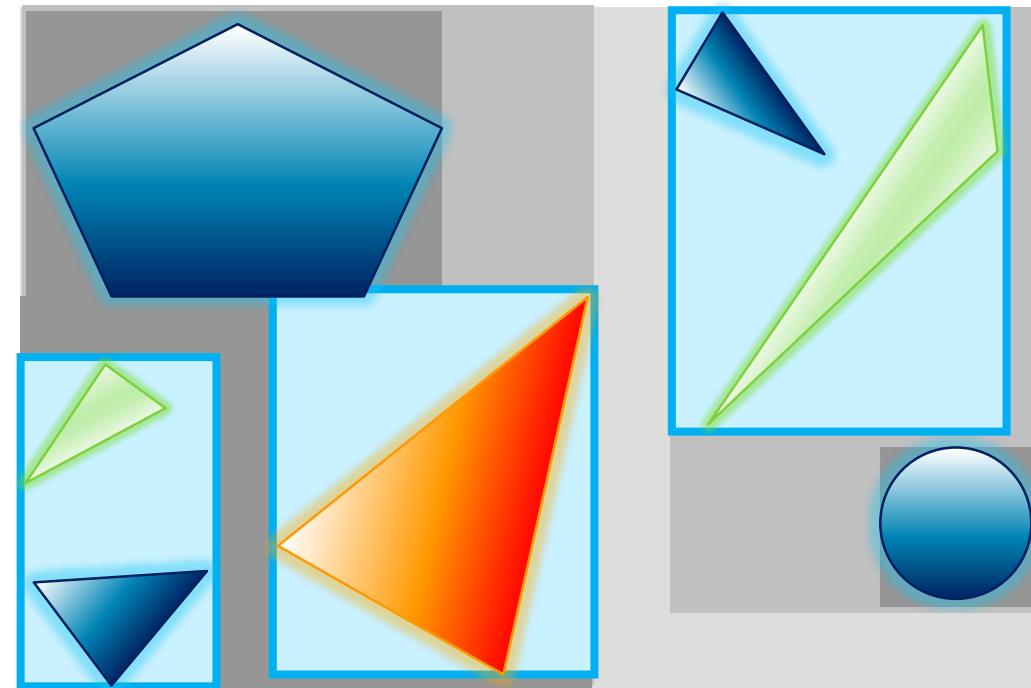


Bounding Volume-Hierarchie



jeder Knoten speichert eine AABB
und Verweis auf seine Kindknoten
bzw. auf Objekte/Primitive (oder auf
eine Liste der Primitive)

Bounding Volume-Hierarchie

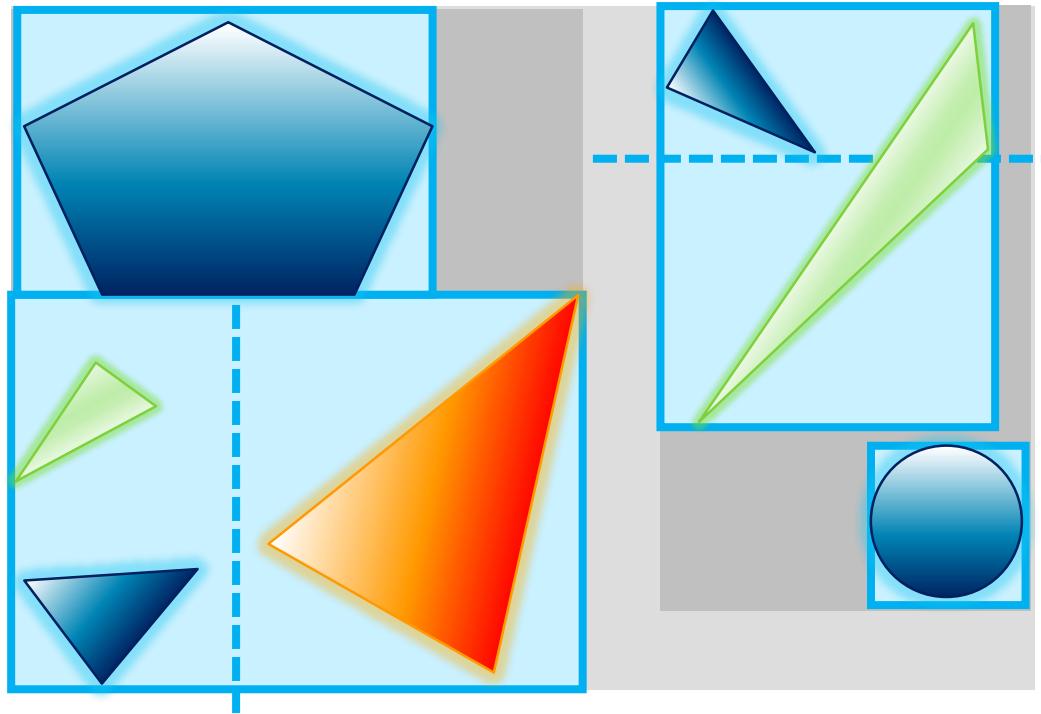


jeder Knoten speichert eine AABB
und Verweis auf seine Kindknoten
bzw. auf Objekte/Primitive (oder auf
eine **Liste der Primitive**)

Bounding Volume-Hierarchie

Unterschiedliche Unterteilungskriterien (später mehr)

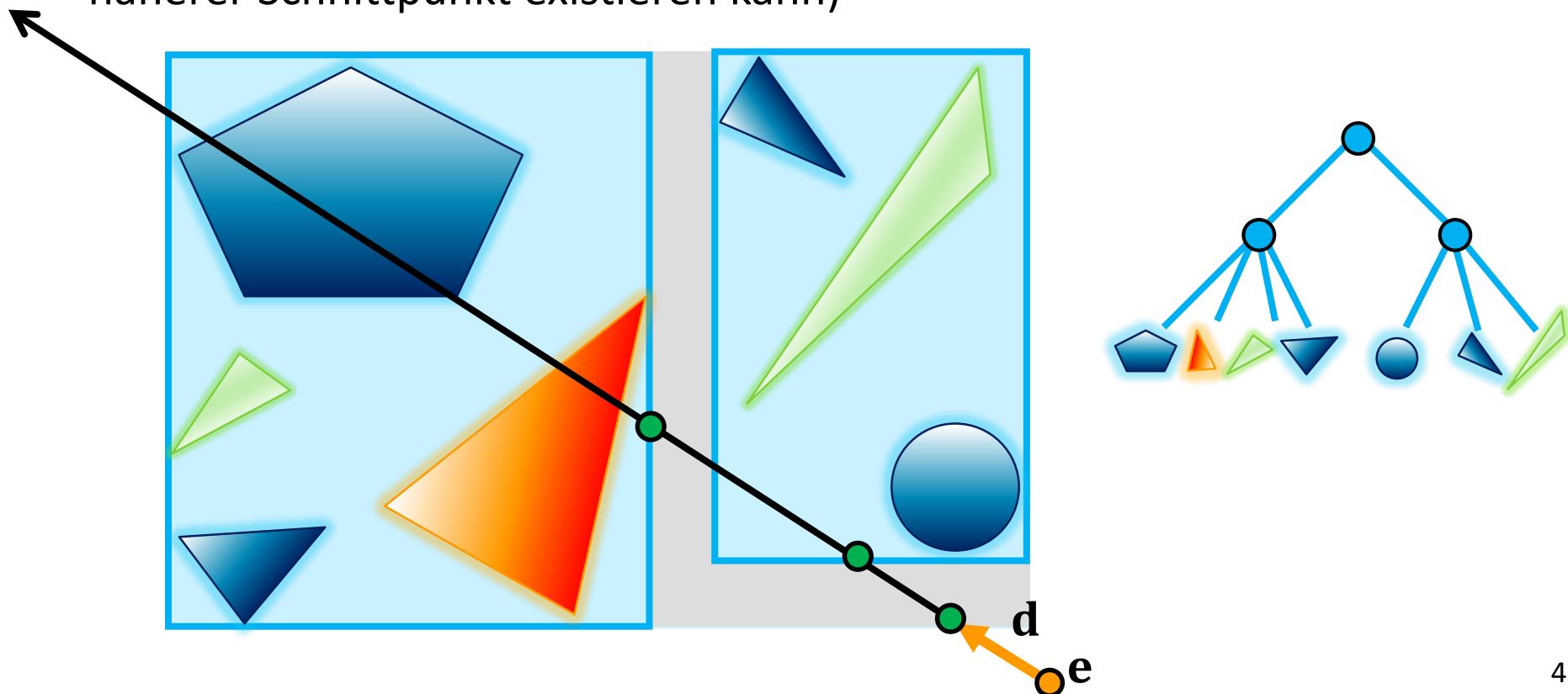
- ▶ unterteile in der Mitte senkrecht zur Achse der größten Ausdehnung
- ▶ unterteile einmal entlang der x -Achse, dann y -Achse, dann z -Achse, ...
- ▶ unterteile so, dass in jeder Teilmenge etwa die gleiche Anzahl Objekte ist
- ▶ verwende Szenengraph bzw. Modellhierarchie
- ▶ minimiere eine Kostenfunktion (z.B. Surface Area Heuristics)



Schnittberechnung mit BVH

Beschleunigung der Schnittberechnung mit BVHs

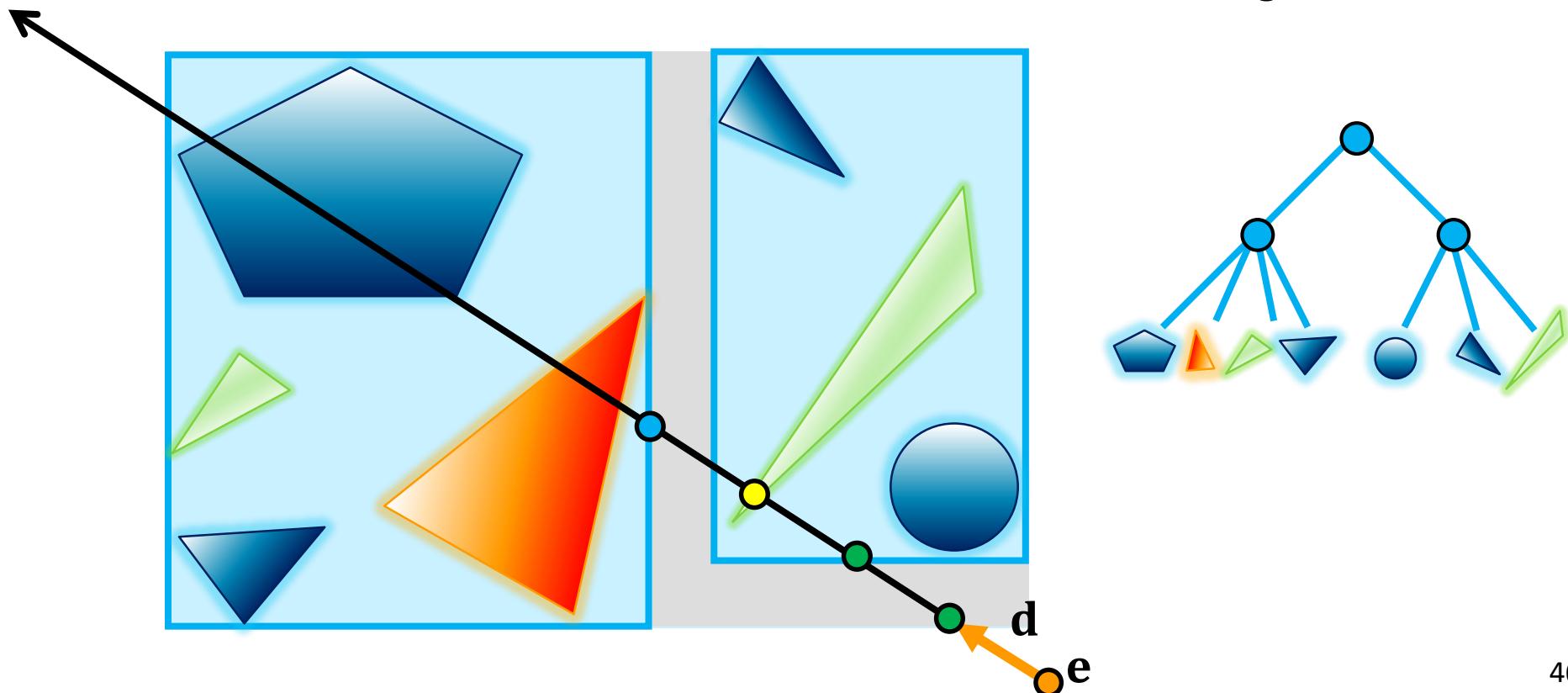
- ▶ überprüfe auf Schnitt mit der Wurzel (= AABB der gesamten Szene)
- ▶ dann steige rekursiv ab (beginnend mit dem näheren Kindknoten)
- ▶ teste auf Schnitt mit Objekten im Kindknoten (steige evtl. weiter ab)
- ▶ prüfe anschließend noch die weiter entfernten Knoten (sofern ein näherer Schnittpunkt existieren kann)



Schnittberechnung mit BVH

Beschleunigung der Schnittberechnung mit BVHs

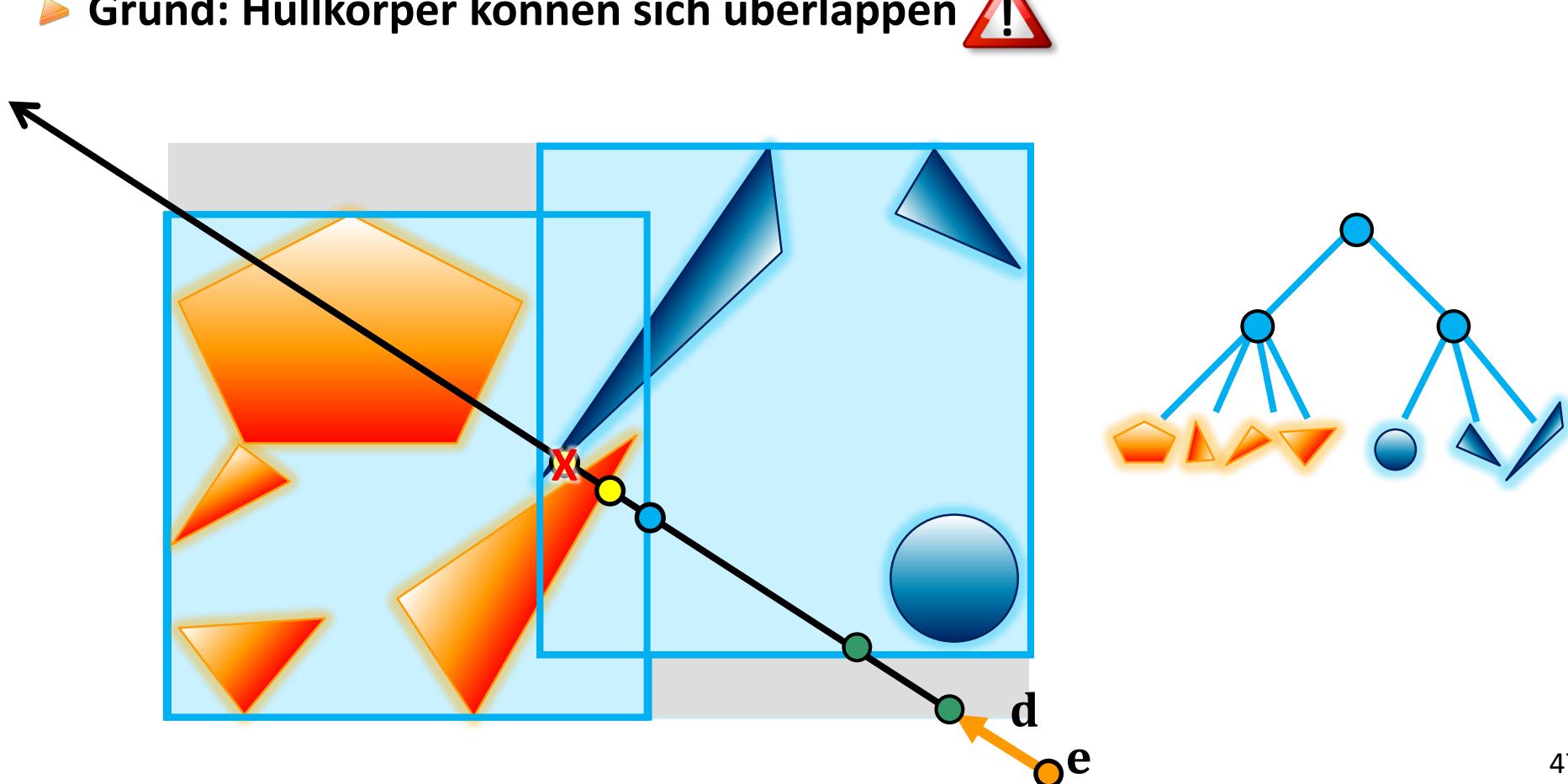
- ▶ teste auf Schnitt mit Objekten im Kindknoten
(hier kein weiterer Abstieg, wir sind schon in einem Blattknoten)
- ▶ Schnittpunkt ● wurde gefunden
- ▶ der Schnittpunkt ist näher als die hintere AABB (Schnittpunkt ○)
→ keine weiteren Tests in diesem Ast des Baums notwendig



Schnittberechnung mit BVH

Beschleunigung der Schnittberechnung mit BVHs

- Vorsicht: liefere einen gefundenen Schnittpunkt nicht sofort zurück!
(es sei denn, wir wollen nur testen, ob es irgendeinen Schnitt gibt)
- es könnte einen näheren in einem anderen Knoten geben
- **Grund: Hüllkörper können sich überlappen** !



Fazit Bounding Volume-Hierarchie



Vorteile

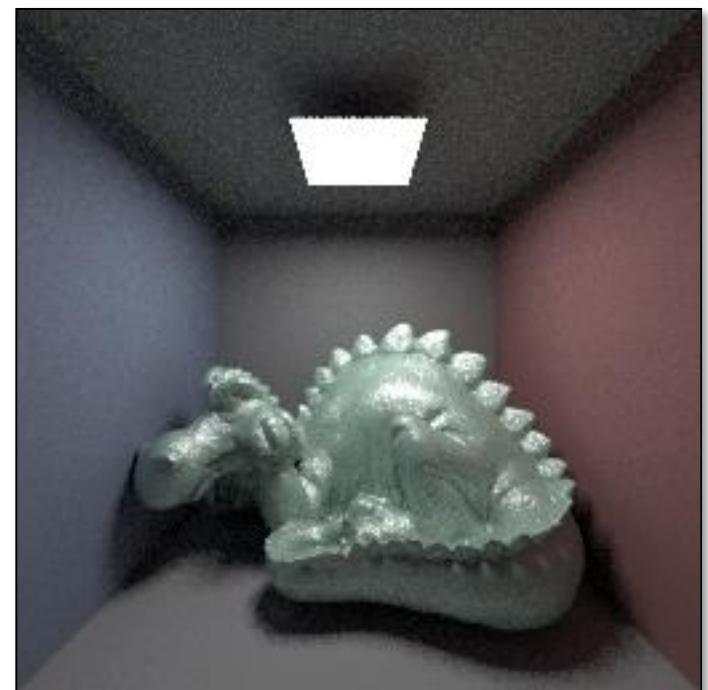
- ▶ Konstruktion und Traversierung ist einfach
- ▶ resultiert in einem Binärbaum mit fixer, geringer Verzweigung
(in der Praxis populär: 1-zu-4 Verzweigung, also jeweils 4 Kindknoten)
- ▶ Komplexität
 - ▶ im Mittel $O(\log n)$ Schnitttests für n Objekte/Primitive in der Szene
(vgl. ohne BVH werden $O(n)$ Schnitttests benötigt)
 - ▶ Aufbau $O(n \log n)$ wenn in der Mitte der AABBs unterteilt wird
 - ▶ Aufbau $O(n \log^2 n)$, wenn die Objekte in einer AABB in zwei (etwa) gleich große Gruppen geteilt werden und eine $O(n \log n)$ Sortierung verwendet wird

Herausforderungen

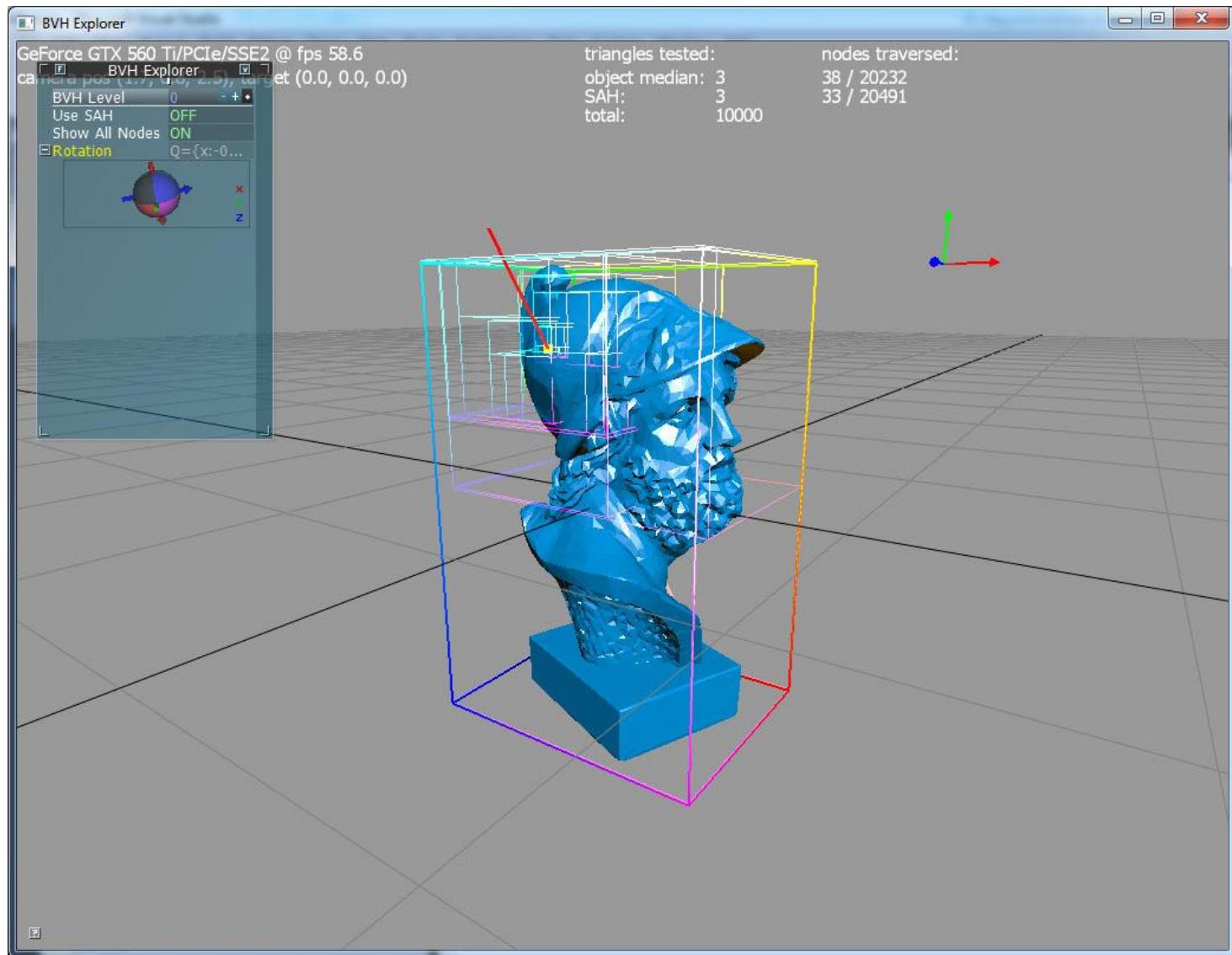
- ▶ Finden einer guten Unterteilung ist schwierig, weil nicht unbedingt klar ist, was eigentlich gut ist: wir wissen nicht vorab, welche Strahlen beim Raytracing verschlossen werden
- ▶ ungeschickte Unterteilung kann zu schlechterer Performanz führen

Performance von BVHs

- ▶ Path Tracing
 - ▶ 32x stochast. Supersampling, Auflösung 256^2 Pixel, 480000 Dreiecke
 - ▶ insgesamt ca. $6.3 \cdot 10^6$ Primär-, Sekundär- und Schattenstrahlen
- ▶ Brute Force (176462sec \approx 49h):
 - ▶ ca. $3 \cdot 10^{12}$ Schnitttests
- ▶ Bounding Volume-Hierarchie (unoptimiert)
 - ▶ Strategie:
 - Unterteilung in 2 gleich große Gruppen
 - ▶ Aufbau 18.3sec, Rendering 59sec
 - ▶ 262143 AABB Knoten
 - ▶ $7.5 \cdot 10^8$ AABB Tests
 - ▶ $1.3 \cdot 10^8$ Schnitttests mit Dreiecken
- ▶ Vergleich
 - ▶ Brute Force: $3431 \times$ Schnitttests
 - ▶ BVH Beschleunigung: $2200 \times$
(BVH Aufbau berücksichtigt)



BVH Demo



Inhalt: Räumliche Datenstrukturen

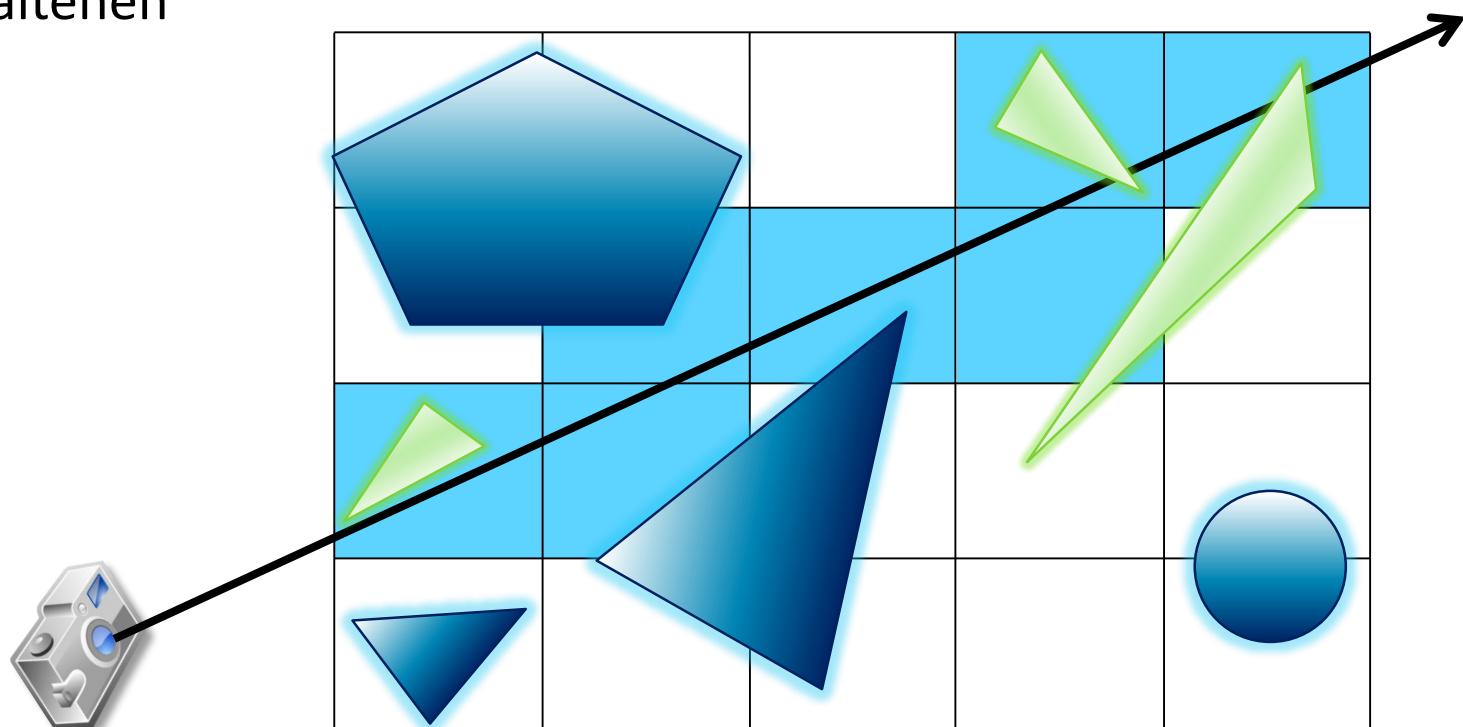
- ▶ Analyse der Kosten bei Raytracing
- ▶ Ansätze zur Beschleunigung von Raytracing
- ▶ Bounding Volumes (Hüllkörper)
- ▶ Räumliche Datenstrukturen
 - ▶ Bounding Volume-Hierarchies
 - ▶ **reguläre und adaptive Gitter**
 - ▶ BSP-Bäume und kD-Bäume



Raumunterteilung durch reguläre Gitter

Grundidee, Prinzip

- ▶ Unterteilung des Raums in Zellen gleicher Größe und Form (daher die Bezeichnung „regulär“), ausgehend von der Bounding Box der Szene
- ▶ Eintrag der Objekte in Zellen, die von ihnen geschnitten werden
- ▶ Traversierung der vom Strahl getroffenen Zellen und Schnittberechnung mit den enthaltenen Objekten

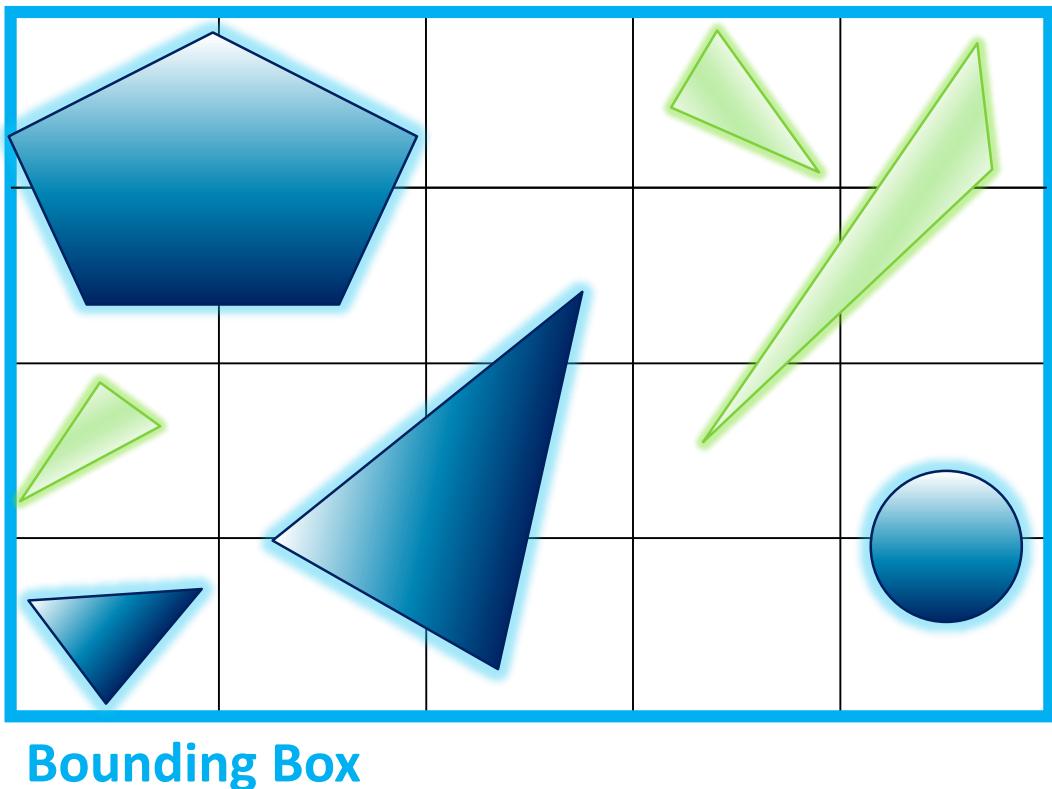


Raumunterteilung durch reguläre Gitter

Größe und Dimension des Gitters

- bestimme Bounding Box der gesamten Szene
- wähle Gitterauflösung n_x, n_y, n_z (n_x muss nicht gleich n_y sein etc.), d.h. die Zellen sind immer gleich große Quader

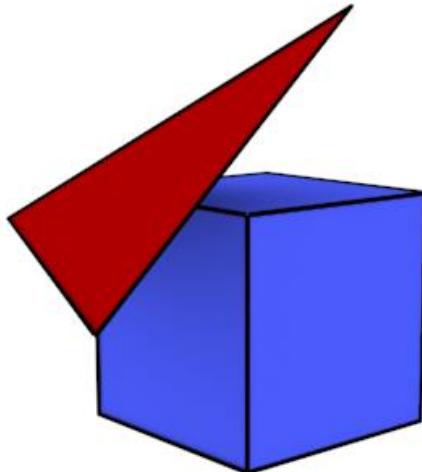
- hier: $n_x = 5, n_y = 4$



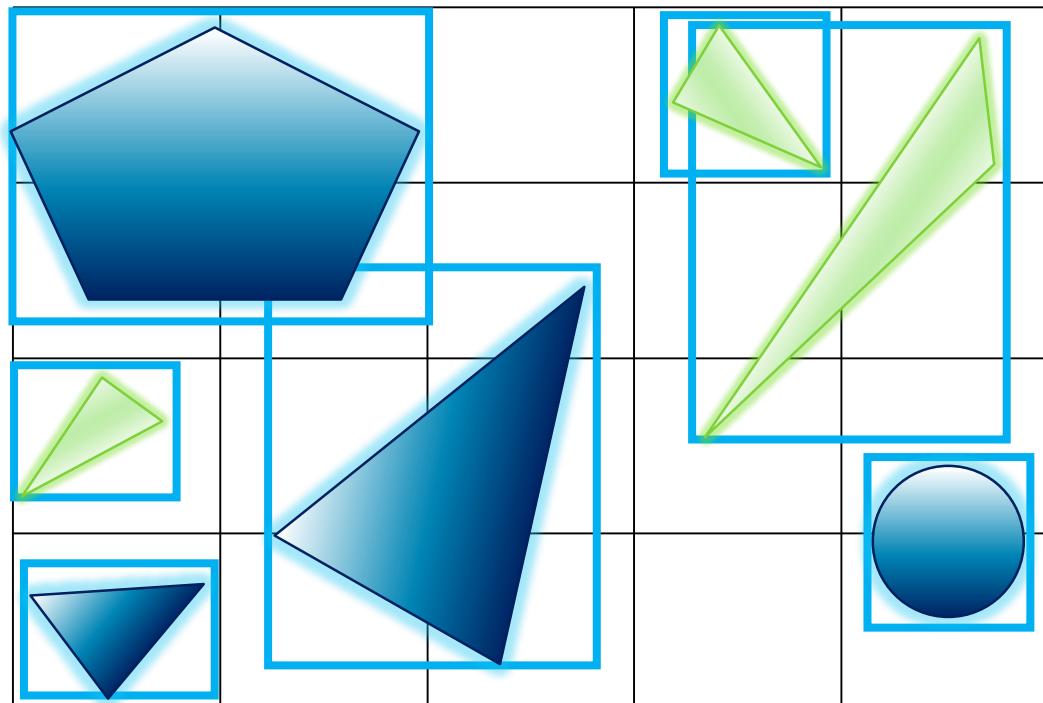
Raumunterteilung durch reguläre Gitter

Eintragen der Primitive in das Gitter

- bestimme die Gitterzellen, die ein Objekt/Primitiv schneidet
 - ▶ einfach und konservativ: alle Zellen, die die AABB des Obj. überlappt
 - ▶ besser: AABB-Test zur Vorauswahl der Zellen vor einem exaktem Test
- überlappt ein Objekt mehrere Zellen, so wird es in jede Zelle eingetragen (verwende Zeiger)



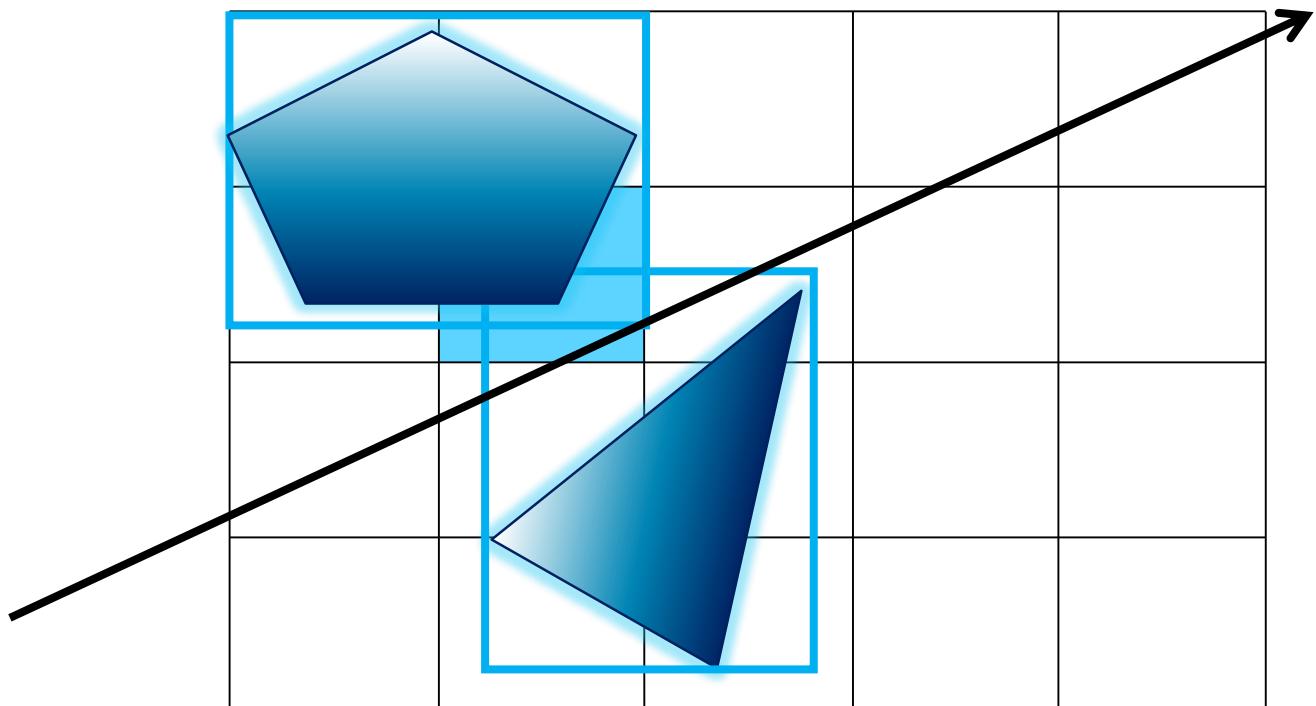
Schnitt Dreieck-Zelle?



Trennungssatz, siehe http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/tribox_tam.pdf

Schnittpunktberechnung

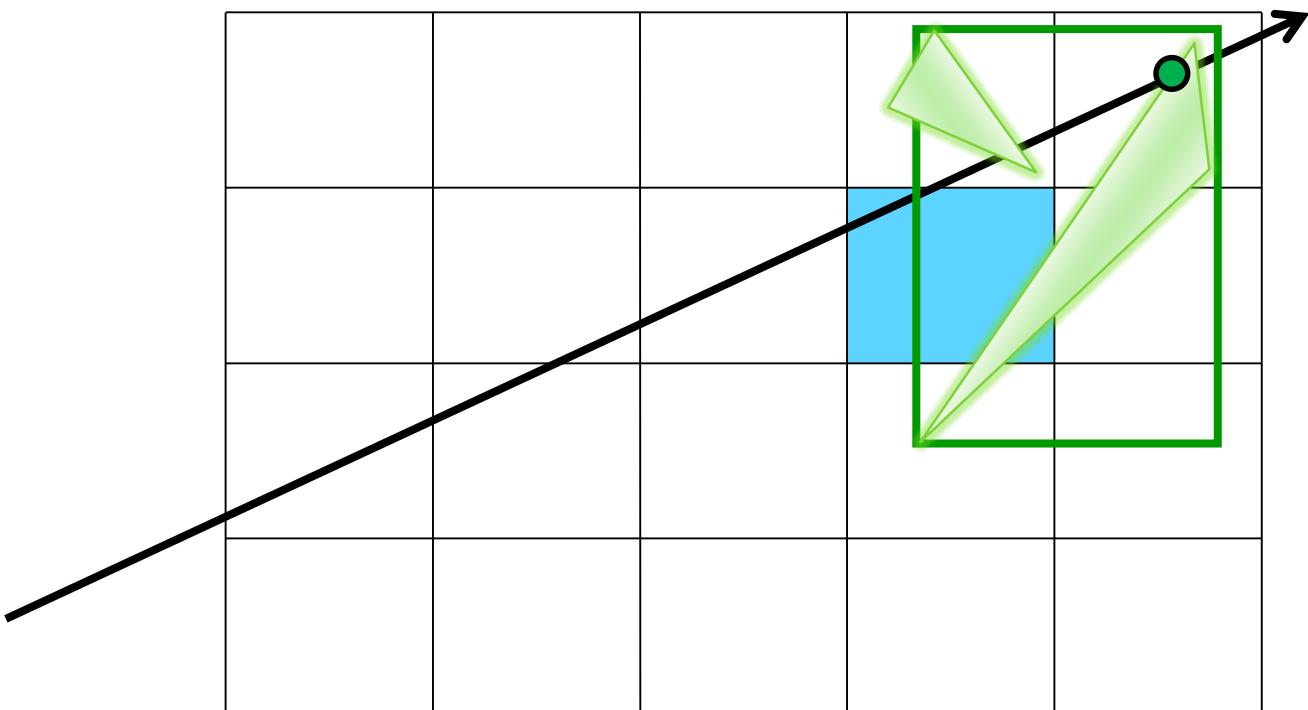
- ▶ teste für jede Zelle entlang des Strahls:
gibt es **in dieser Zelle** Schnitte mit Objekten?
 - ▶ ja: gebe nahsten dieser Schnittpunkte zurück
 - ▶ nein: weiter mit der Traversierung



Raumunterteilung durch reguläre Gitter

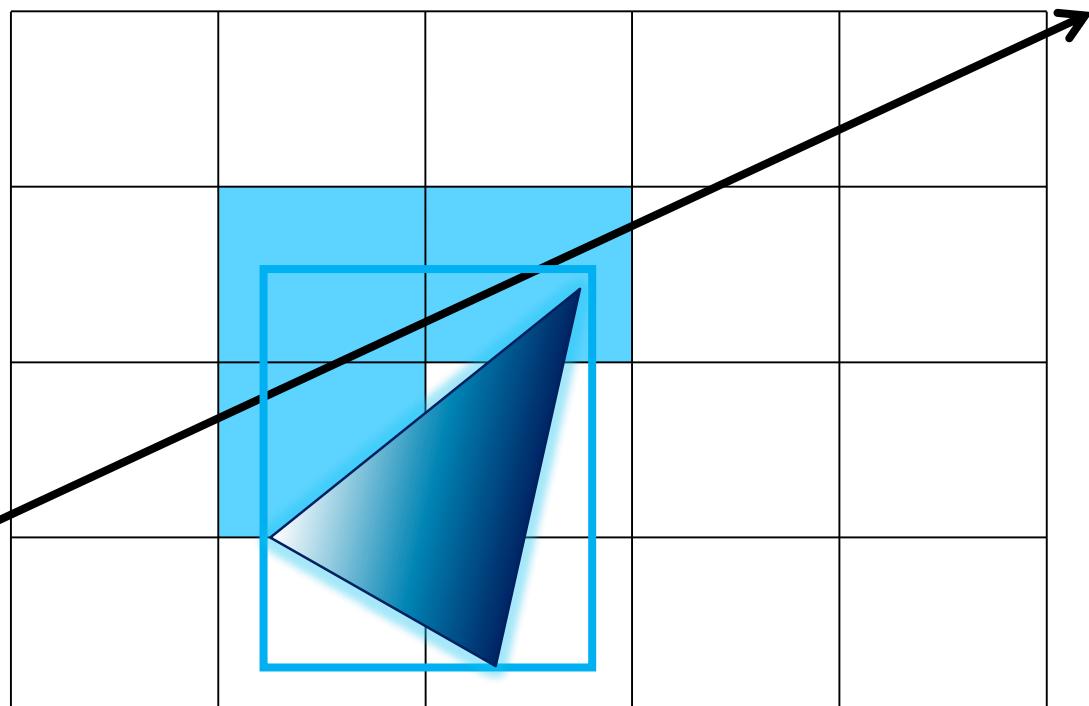
Ignoriere Schnitte außerhalb der aktuellen Zelle

- ▶ verwerfe Schnittpunkt(e) außerhalb der aktuellen Zelle
- ▶ es könnte nähere Schnittpunkte mit anderen Objekten geben (hier ⚡)



Mailboxing: vermeide mehrfache Schnitttests mit einem Objekt

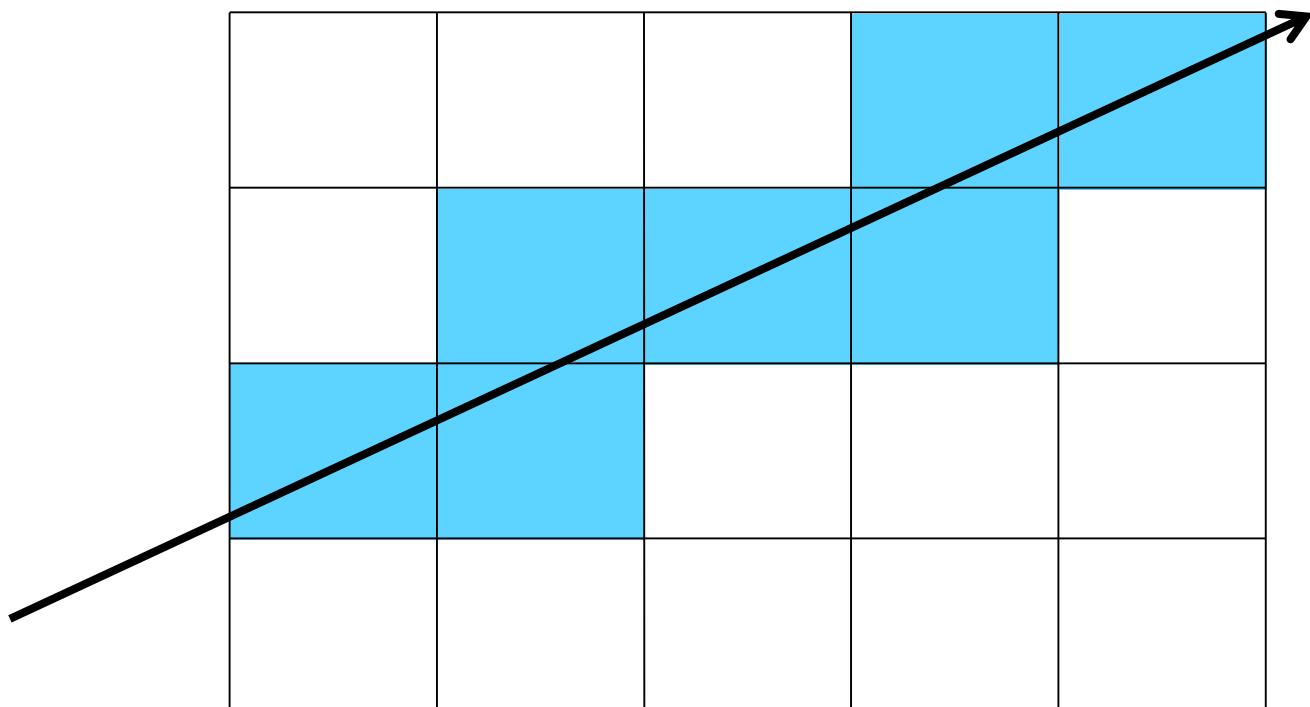
- ▶ führe Schnitttest nur einmal durch und markiere das Objekt als getestet
 - ▶ Ziel: kein nochmaliger Schnitttest mit einem bereits markierten Objekt
 - ▶ wenn ein Schnittpunkt gefunden wurde (und er außerhalb der Zelle lag): speichere den Schnittpunkt, um ihn später nicht nochmal berechnen zu müssen
-
- ▶ Mailboxing ist optional, Trade-off zwischen weniger Schnitttests und zusätzlichem Overhead



Raumunterteilung durch reguläre Gitter

Traversierung des Gitters

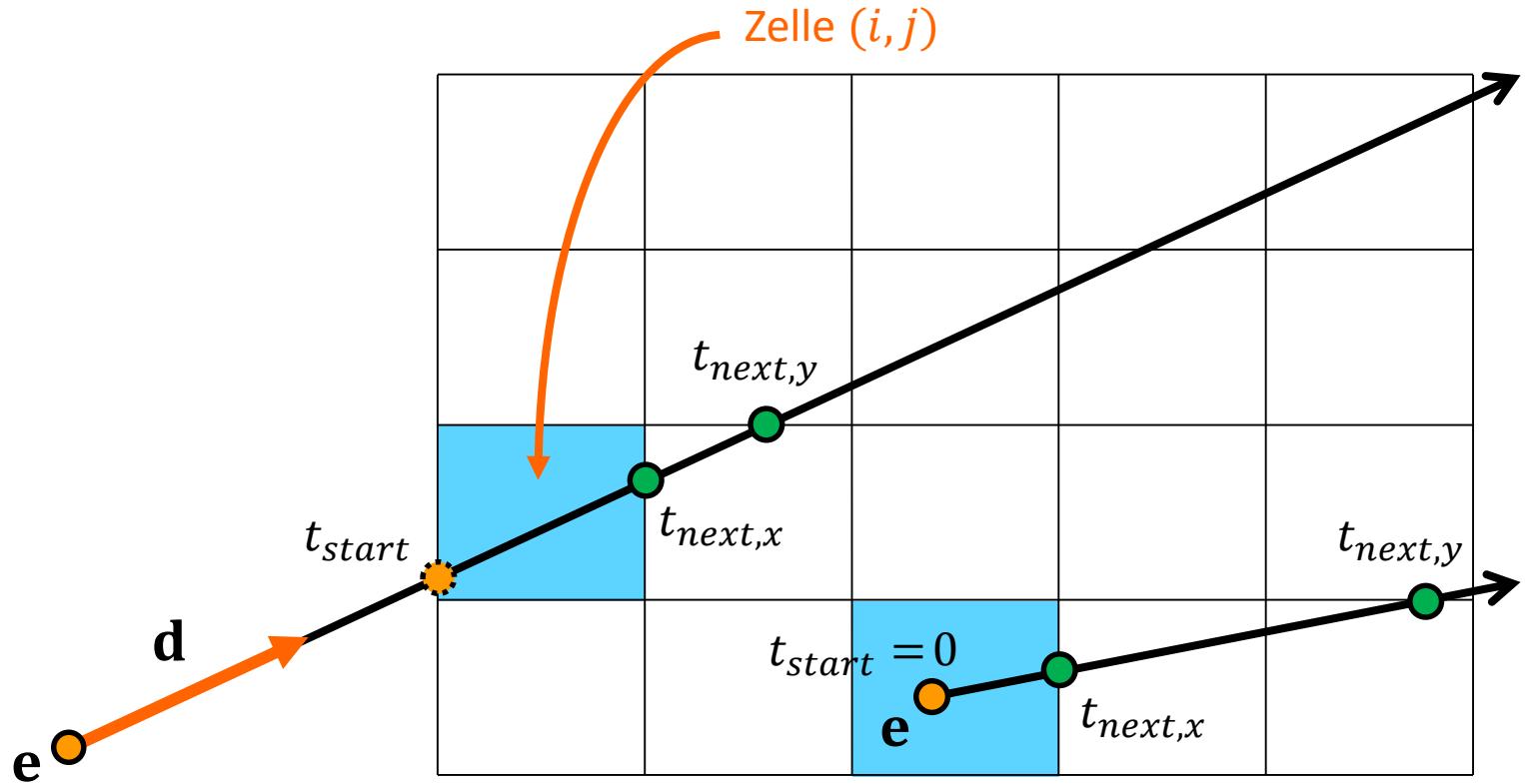
- ▶ wie findet man die Zellen die ein Strahl durchläuft?
(nur Objekte/Primitive in diesen Zellen möchte man testen)
- ▶ Ziel: jede Zelle genau einmal, in der richtigen Reihenfolge, betrachten



Raumunterteilung durch reguläre Gitter

Bestimmung der Start-Zelle

- schneide Strahl mit der Bounding Box der Szene, liefert t_{start}
- aus den Schnittkoordinaten lässt sich der Zellenindex (i, j) und (zusammen mit \mathbf{d}) dann $t_{next,x}$ bzw. $t_{next,y}$ berechnen
- Strahlursprung kann auch in der Bounding Box der Szene liegen

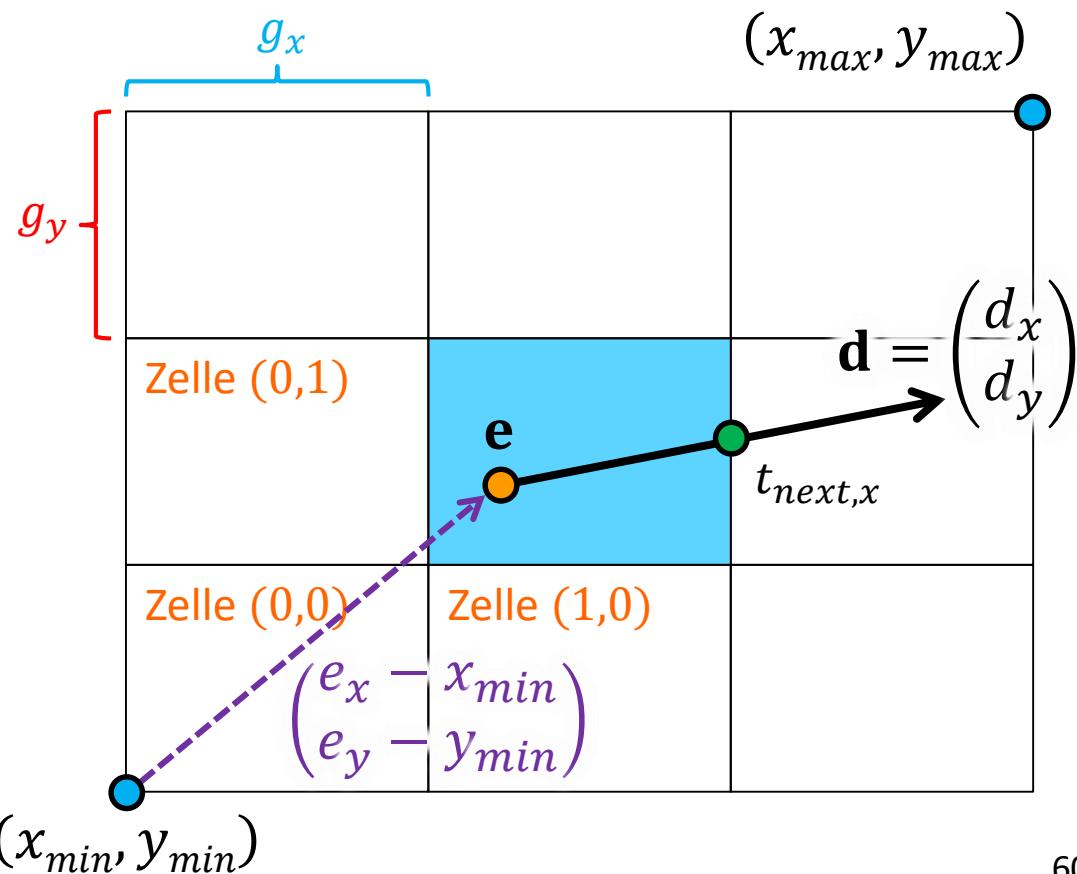


Raumunterteilung durch reguläre Gitter

Bestimmung der Start-Zelle

- Zellenindex (i, j) und $t_{next,x}$ bzw. $t_{next,y}$ aus Koordinaten berechnen
- $i = \lfloor (e_x - x_{min})/g_x \rfloor$ bzw. $j = \lfloor (e_y - y_{min})/g_y \rfloor$
- $e_x + t_{next,x}d_x = (i + 1)g_x + x_{min}$ (für $d_x > 0$)
 $\Rightarrow t_{next,x} = \dots$

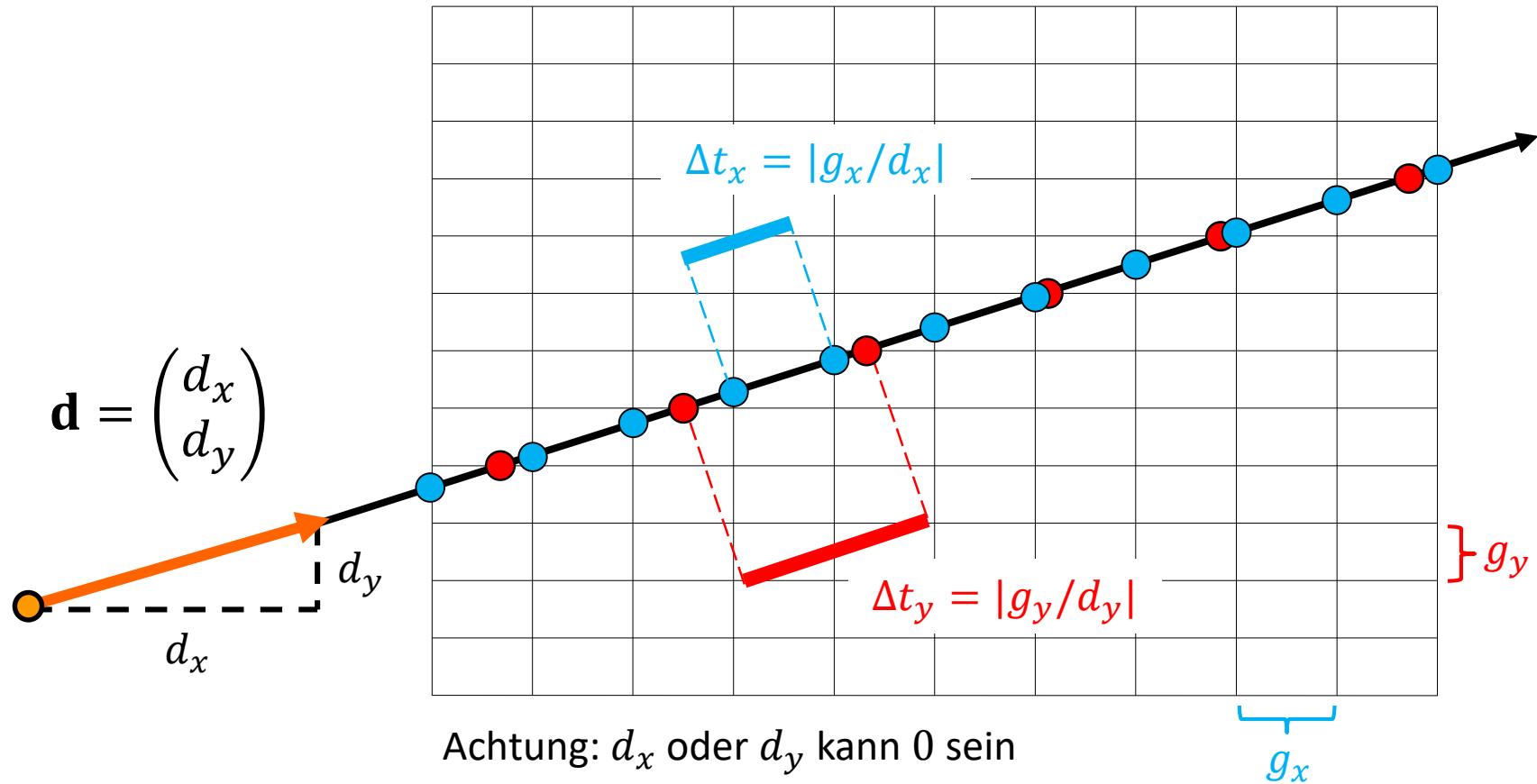
(Achtung: d_x oder d_y kann auch 0 sein)



Raumunterteilung durch reguläre Gitter

Sprung zur nächsten Zelle

- horizontale und vertikale Sprünge zur nächsten Spalte bzw. Zeile haben immer denselben Abstand



Raumunterteilung durch reguläre Gitter

Sprung zur nächsten Zelle (digital differential analyzer, DDA)

-  funktioniert wie folgt auch für $d_{x,y} < 0$
- Zelle auf (i, j) folgend (wir betrachten gerade die Stelle $t = t_{min}$):

```
if ( $t_{next,x} < t_{next,y}$ )
```

```
     $i += \text{sign}_x$ 
```

```
     $t_{min} = t_{next,x}$ 
```

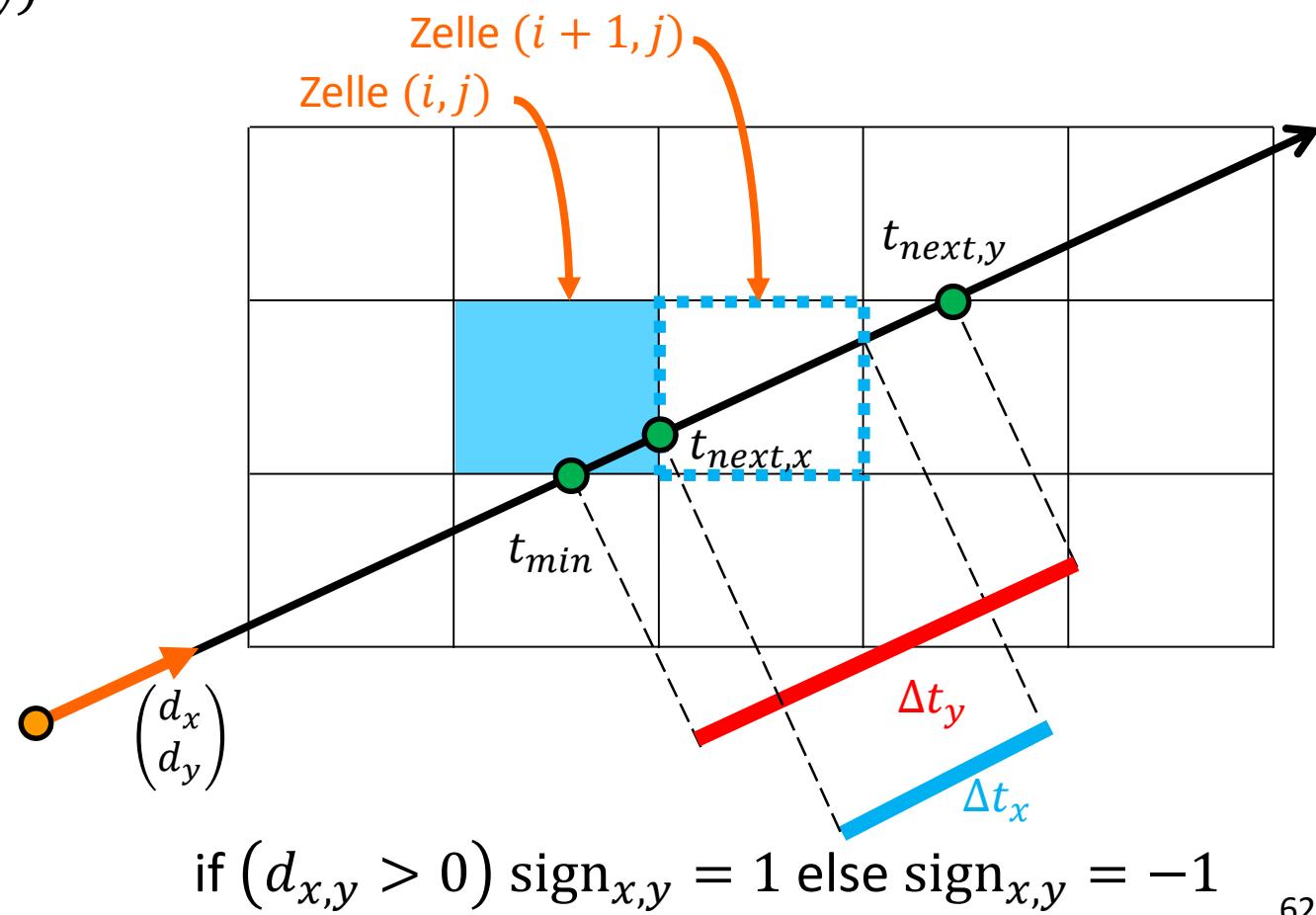
```
     $t_{next,x} += \Delta t_x$ 
```

```
else
```

```
     $j += \text{sign}_y$ 
```

```
     $t_{min} = t_{next,y}$ 
```

```
     $t_{next,y} += \Delta t_y$ 
```



Raumunterteilung durch reguläre Gitter

Sprung zur nächsten Zelle (digital differential analyzer, DDA)

-  funktioniert wie folgt auch für $d_{x,y} < 0$
- Zelle auf (i, j) folgend (wir betrachten gerade die Stelle $t = t_{min}$):

```
if ( $t_{next,x} < t_{next,y}$ )
```

```
     $i += \text{sign}_x$ 
```

```
     $t_{min} = t_{next,x}$ 
```

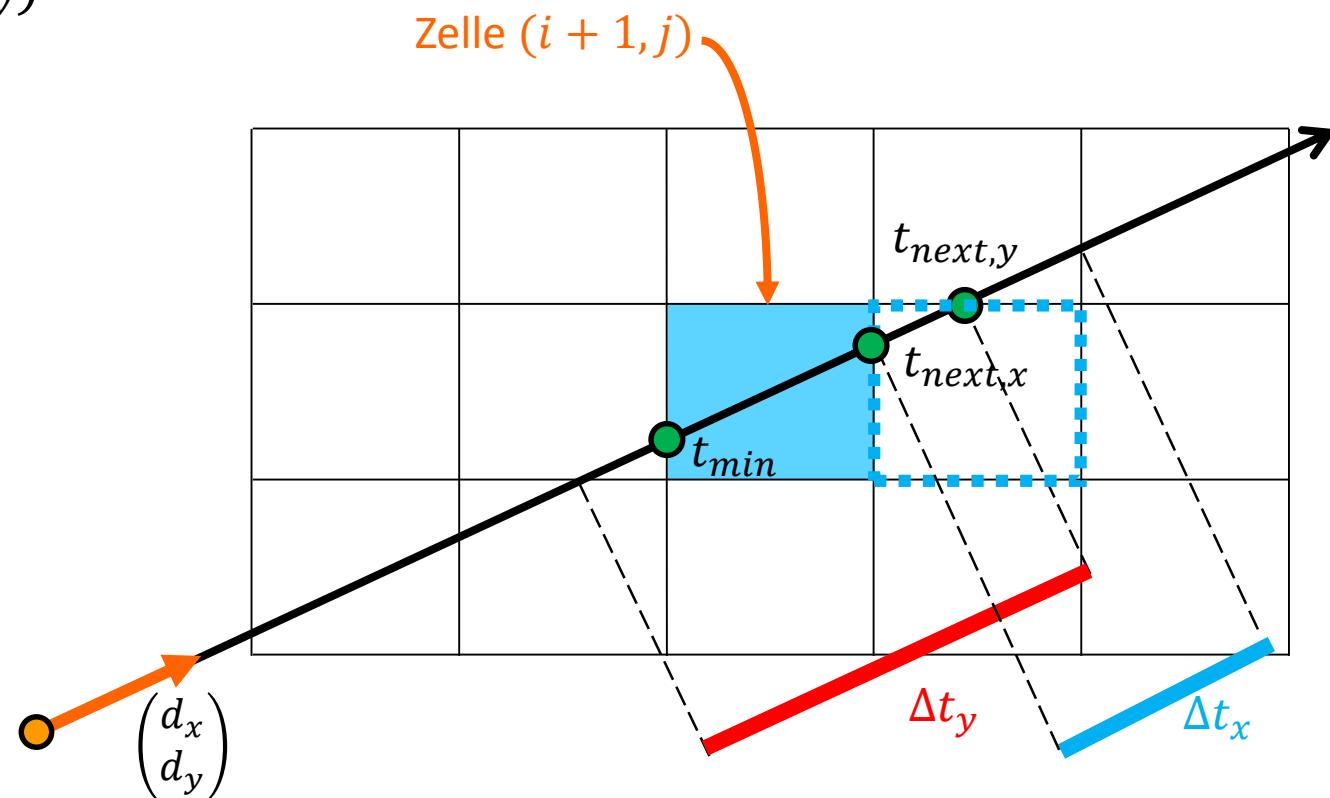
```
     $t_{next,x} += \Delta t_x$ 
```

```
else
```

```
     $j += \text{sign}_y$ 
```

```
     $t_{min} = t_{next,y}$ 
```

```
     $t_{next,y} += \Delta t_y$ 
```



$\text{if } (d_{x,y} > 0) \text{ sign}_{x,y} = 1 \text{ else sign}_{x,y} = -1$

Raumunterteilung durch reguläre Gitter

Sprung zur nächsten Zelle (digital differential analyzer, DDA)

-  funktioniert wie folgt auch für $d_{x,y} < 0$
- Zelle auf (i, j) folgend (wir betrachten gerade die Stelle $t = t_{min}$):

```
if ( $t_{next,x} < t_{next,y}$ )
```

```
     $i += \text{sign}_x$ 
```

```
     $t_{min} = t_{next,x}$ 
```

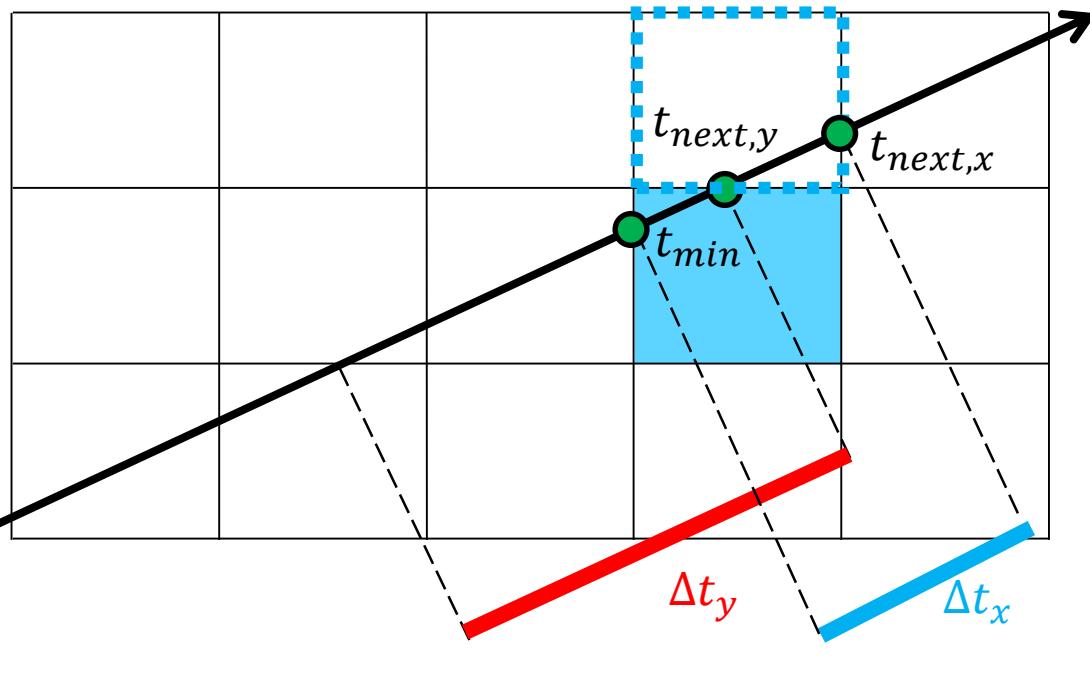
```
     $t_{next,x} += \Delta t_x$ 
```

```
else
```

```
     $j += \text{sign}_y$ 
```

```
     $t_{min} = t_{next,y}$ 
```

```
     $t_{next,y} += \Delta t_y$ 
```



$\text{if } (d_{x,y} > 0) \text{ sign}_{x,y} = 1 \text{ else sign}_{x,y} = -1$

Zusammenfassung

- ▶ bestimme AABB der Szene und erzeuge Gitter
- ▶ trage Objekte/Primitive in das Gitter ein
- ▶ für jeden Strahl $\mathbf{r}(t)$
 - ▶ bestimme Startzelle $z(i, j, k)$, t_{start} , $t_{next,x}$, $t_{next,y}$ und $t_{next,z}$ (in 3D)
 - ▶ bestimme $sign_x$, $sign_y$ und $sign_z$ aus d_x , d_y , d_z
- ▶ solange $0 \leq i < n_x \wedge 0 \leq j < n_y \wedge 0 \leq k < n_z$
 - ▶ für jedes Primitiv P in $z(i, j, k)$
 - ▶ teste auf Schnitt von $\mathbf{r}(t)$ und P
 - ▶ wenn Schnitt(e) in aktueller Zelle gefunden, dann gebe nahsten zurück
 - ▶ sonst: gehe weiter zu nächster Zelle
 - ▶ optional: Mailboxing

Fazit: Raumunterteilung durch reguläre Gitter



Vorteile

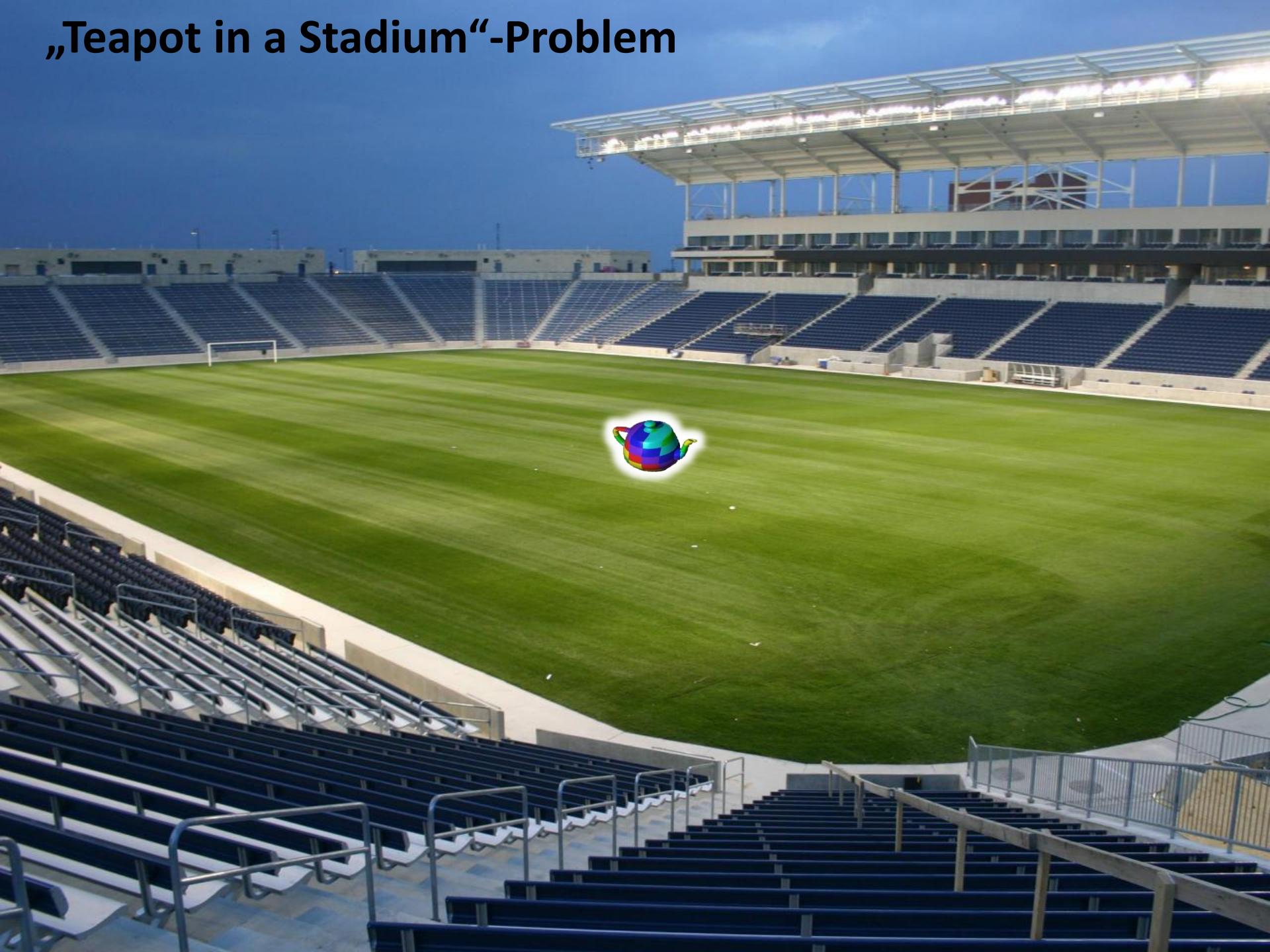
- ▶ einfach zu konstruieren
- ▶ einfach zu traversieren

Nachteile

- ▶ in der Regel nur wenige Zellen belegt
- ▶ u.U. viel Geometrie in wenigen Zellen konzentriert
(siehe nächste Folie)

→ einfaches Traversieren und Aufbau der Datenstruktur
... aber lohnt sich der Verzicht auf Adaptivität?

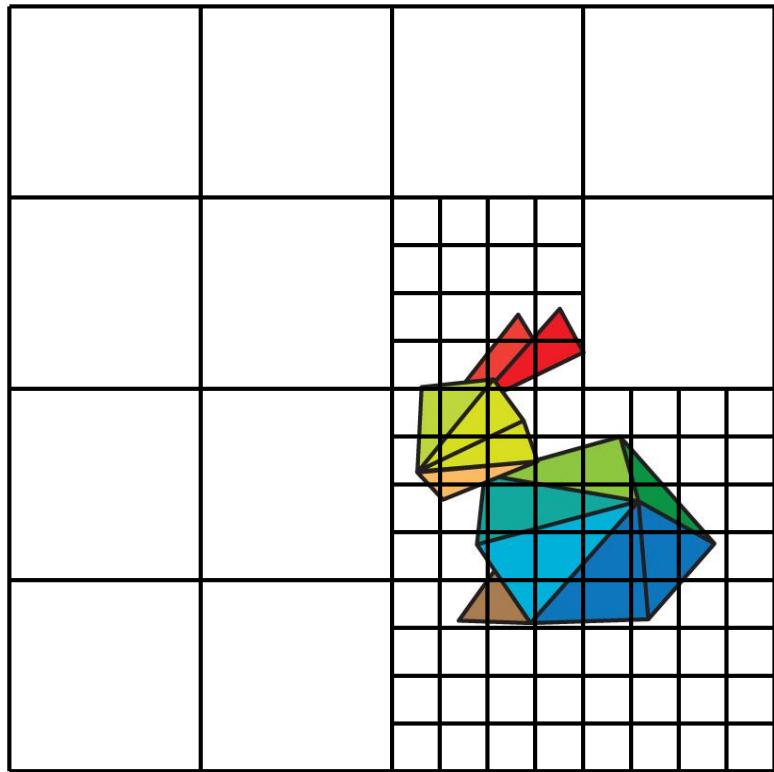
„Teapot in a Stadium“-Problem



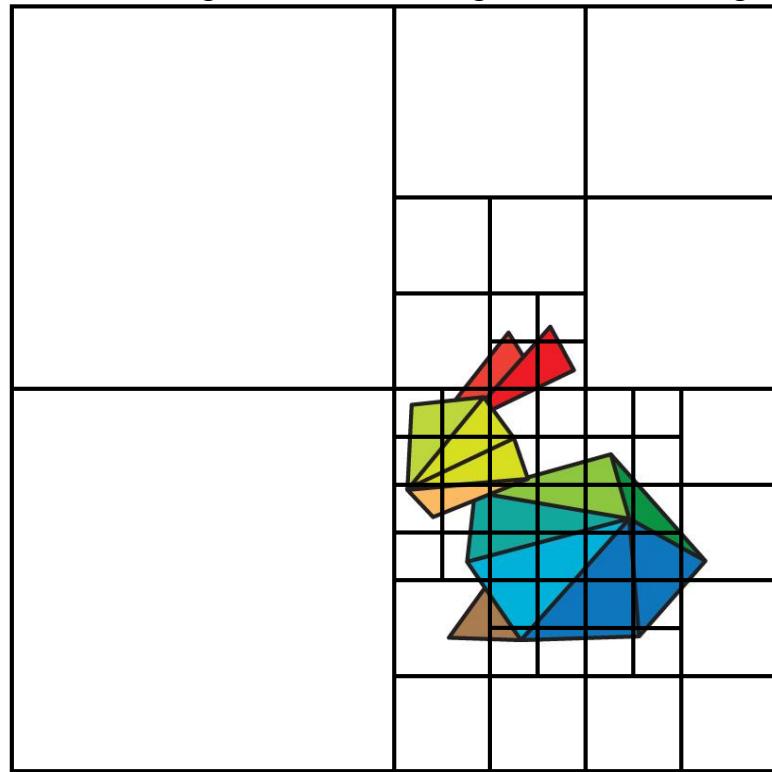
Raumunterteilung durch adaptive Gitter

Grundidee

- rekursive Unterteilung einer Zelle bis
 - ▶ sie nur noch maximal eine vorgegebene Anzahl Primitive enthält, oder
 - ▶ eine maximale Anzahl Unterteilungen durchgeführt wurde



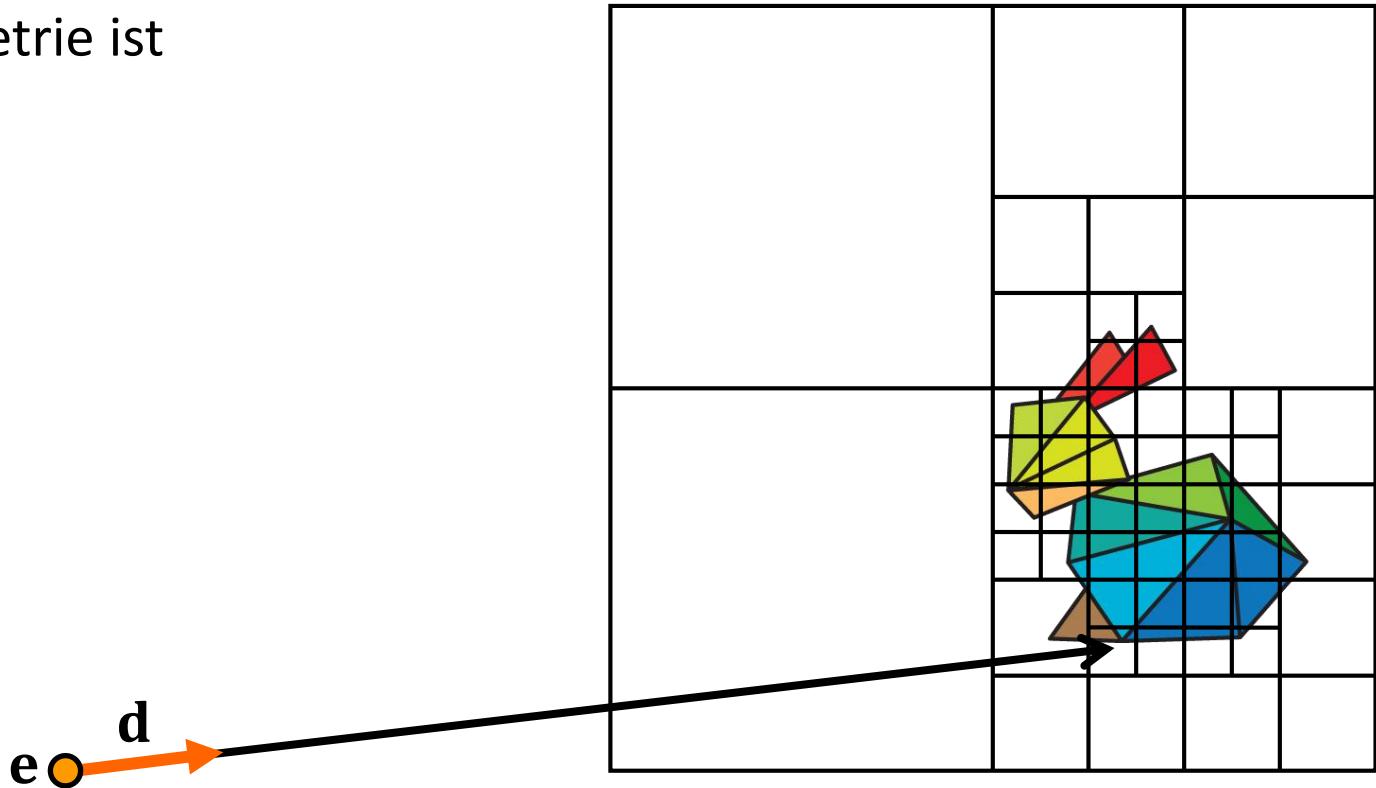
Verschachtelte Gitter (Nested Grids)



Oktalbaum (Octree), bzw. hier in 2D ein Quadtree (jeder Knoten hat 4 Kinder)

Hierarchische Raumunterteilung mit Octrees

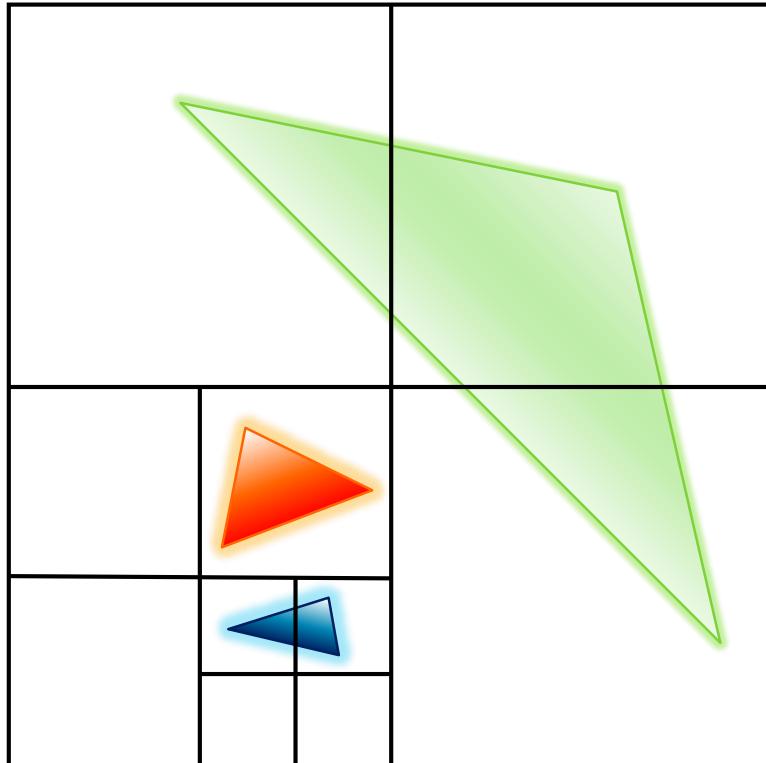
- ▶ beginne wieder mit der Bounding Box für die ganze Szene
- ▶ rekursive 1-zu-8 (daher „Oktal“) Unterteilung jeweils in der Mitte in jeder Richtung, falls noch zu viele Primitive in einer Zelle sind
 - ▶ adaptive Unterteilung erlaubt große Schritte im leeren Raum
 - ▶ feine Unterteilung nur dort, wo Geometrie ist



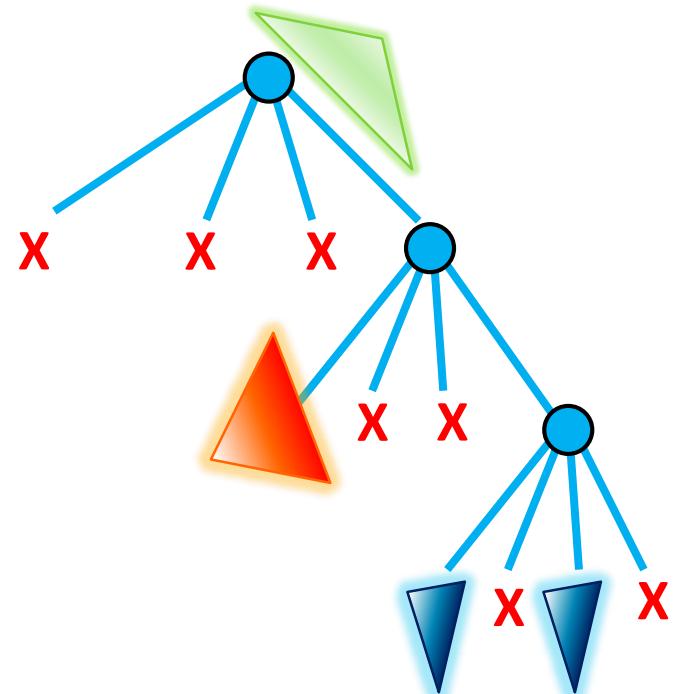
Oktalbaum/Octree

Hierarchische Raumunterteilung mit Octrees

- ▶ Verweise auf Primitive werden in inneren Knoten gespeichert (grün) oder als Blätter des Baums (orange, blau), dann u.U. mehrfach
- ▶ das grüne Dreieck könnte also auch anstelle der 3 leeren Kindknoten der Wurzel referenziert sein (**nur dann echte Raumunterteilung**)



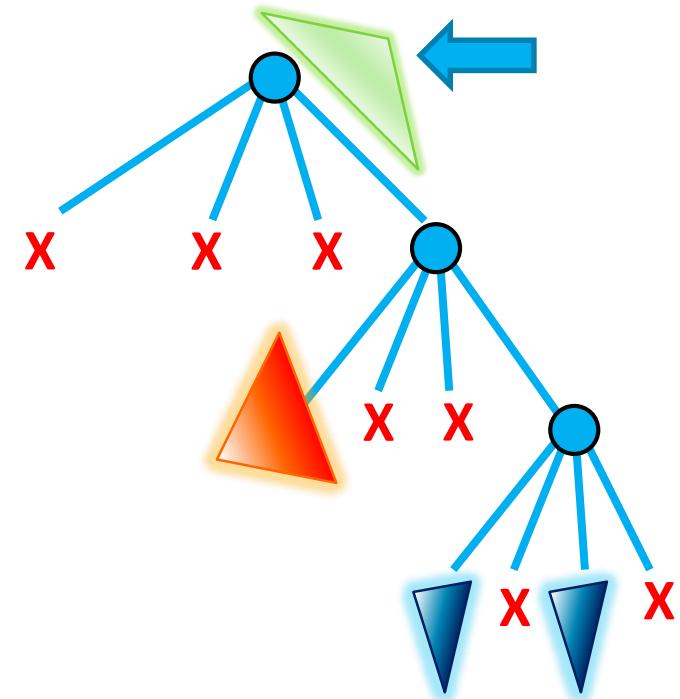
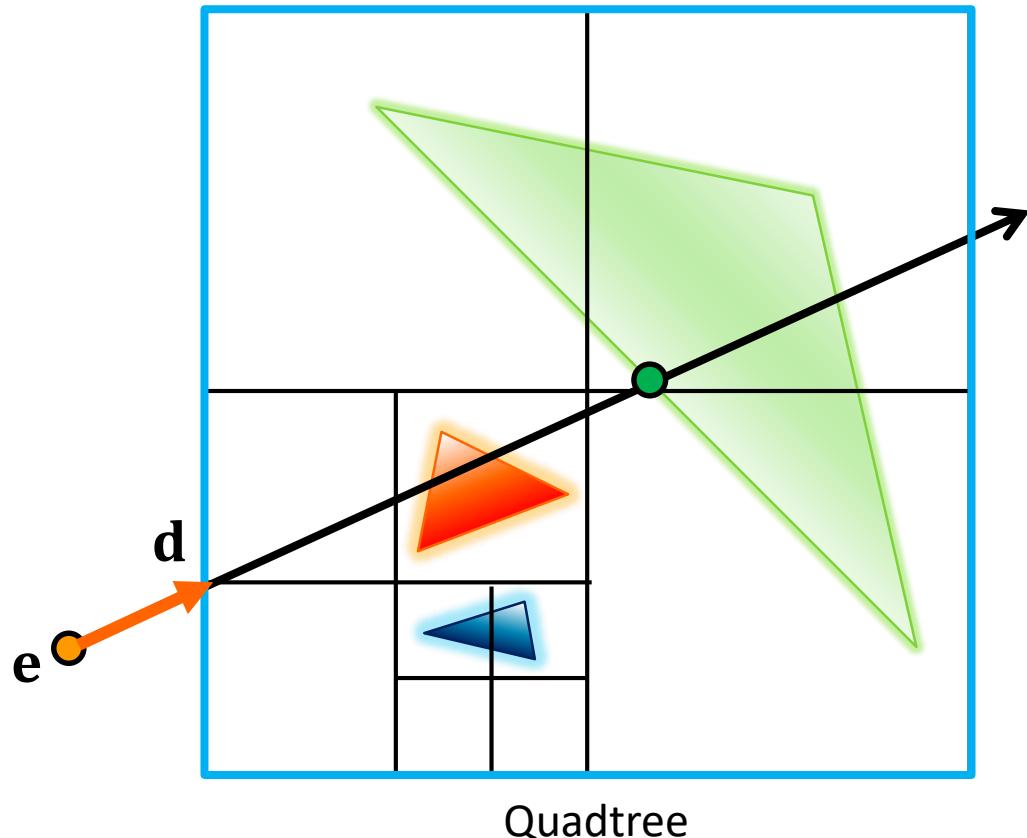
Oktalbaum (Octree), hier in
2D dargestellt als Quadtree



Oktalbaum/Octree

Traversierung

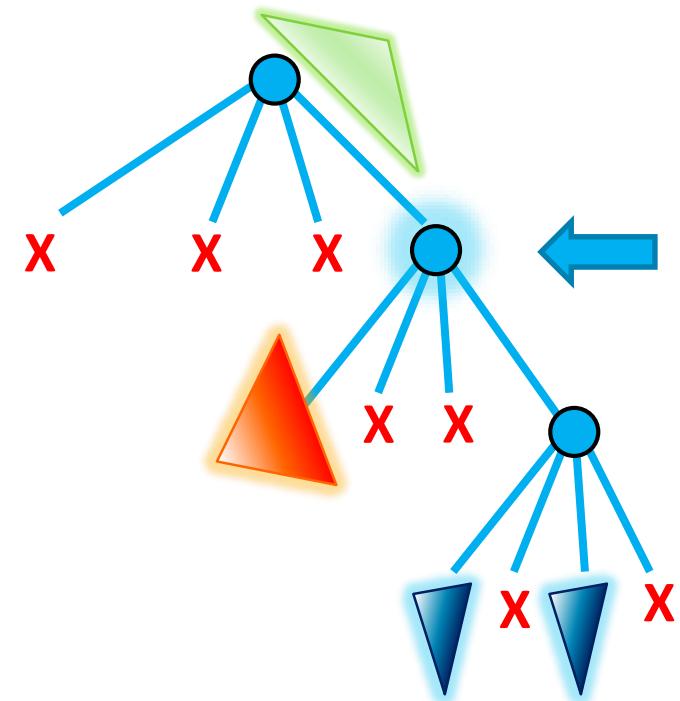
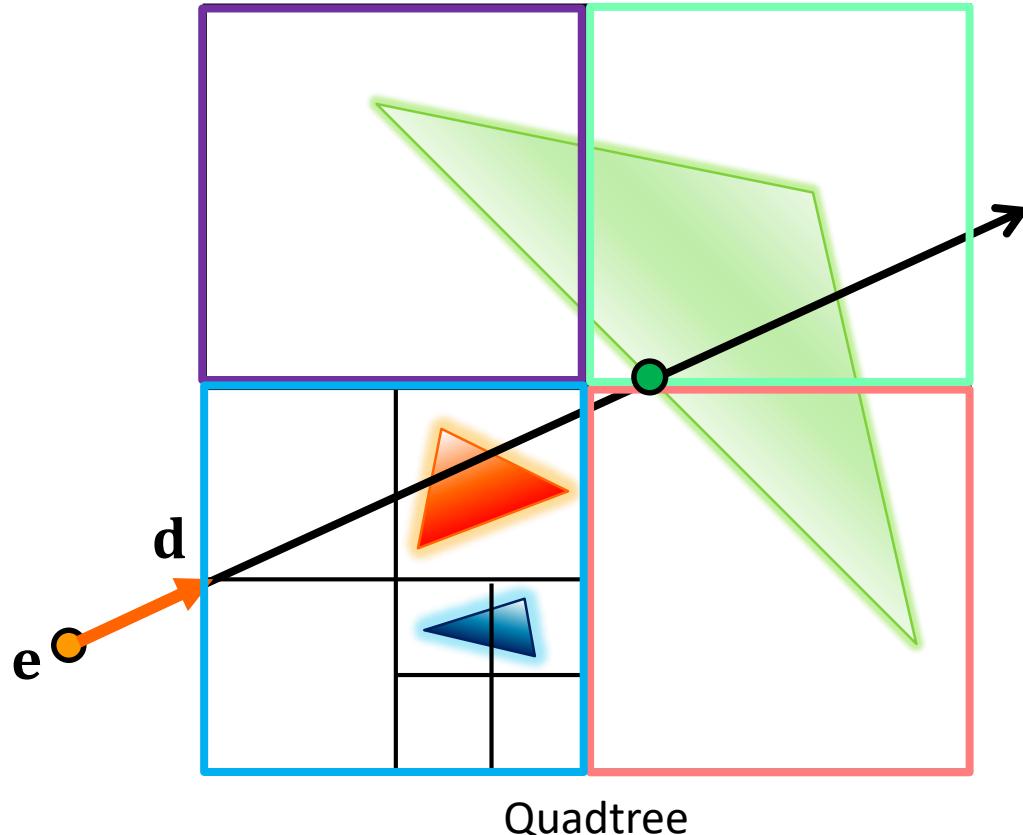
- ▶ Test gegen AABB der Wurzel (= AABB der gesamten Szene)
- ▶ wenn Schnitt mit AABB → Test mit Objekten gespeichert in der Wurzel
(Achtung: auch dieser Schnitt ● muss nicht der naheste sein weil der Wurzelknoten mit anderen überlappt!)



Oktbaum/Octree

Traversierung

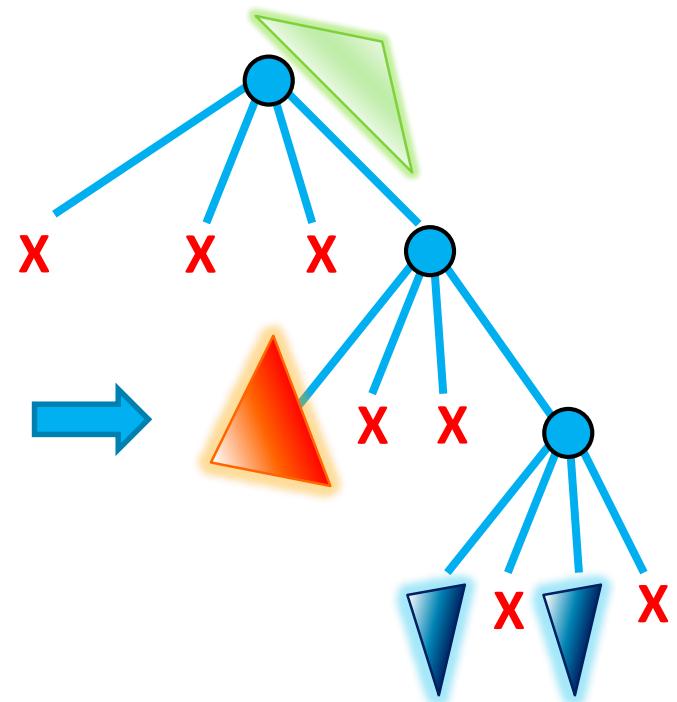
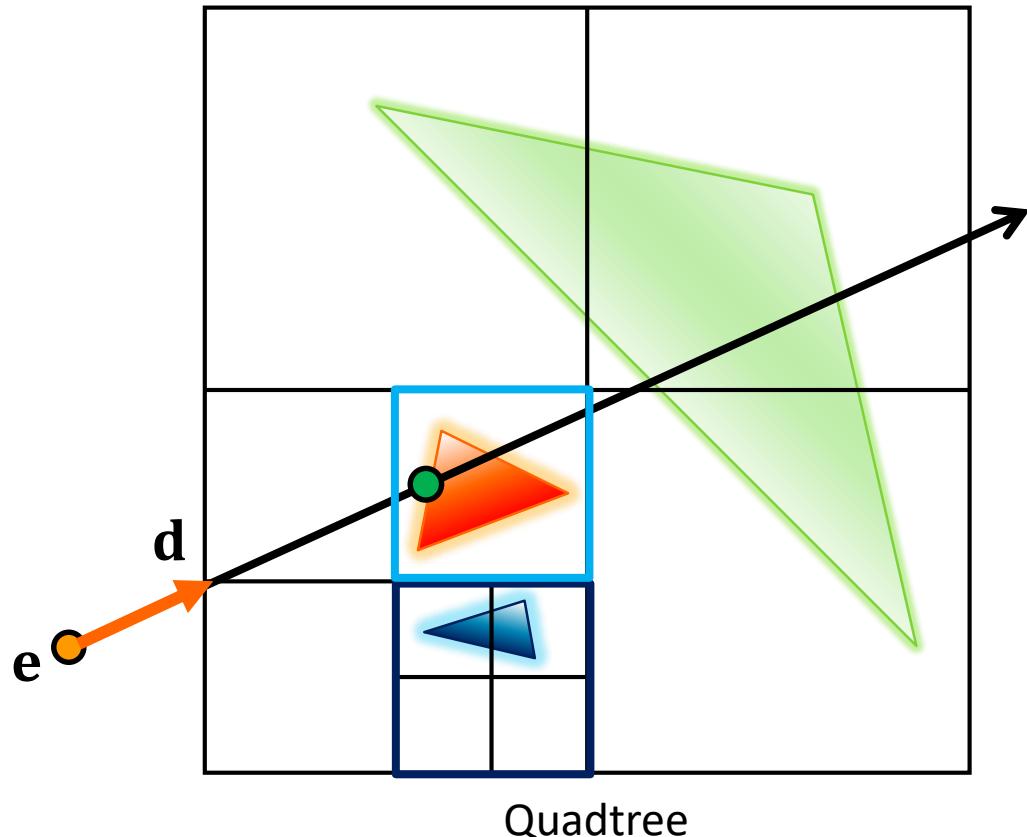
- überprüfe alle Kindknoten 
- ob sie näher als der gefundene Schnittpunkt sind → 
- nur  ist hier näher und nicht leer



Oktbaum/Octree

Traversierung

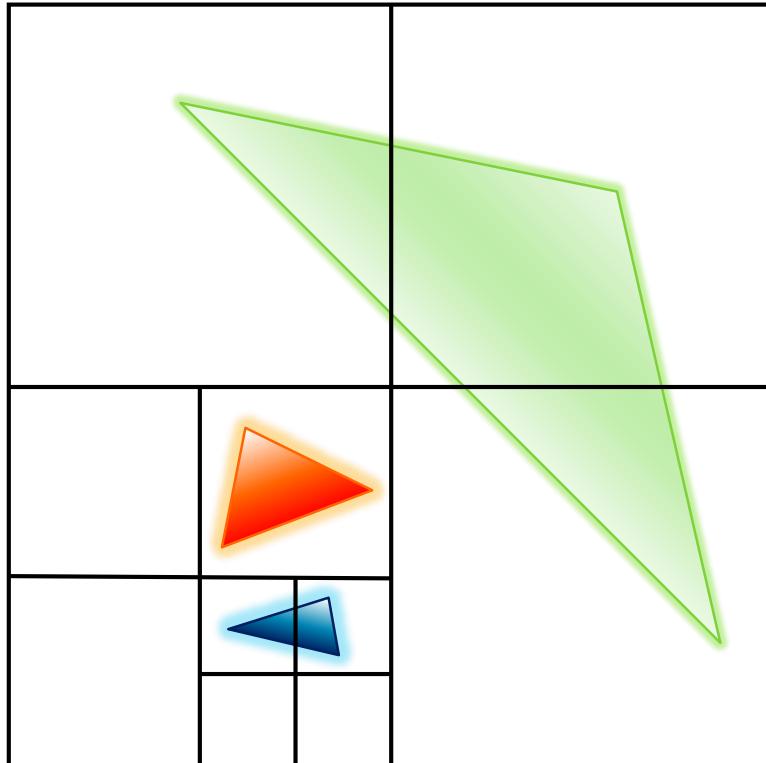
- teste wiederum alle näheren, nicht-leeren Kindknoten
 - nur einer wird geschnitten und ist nicht leer → darin neuer Schnittpkt.
 - hier sind wir fertig – hätten wir nichts geschnitten, dann müssten wir u. U. wieder in der Hierarchie aufsteigen und weitersuchen



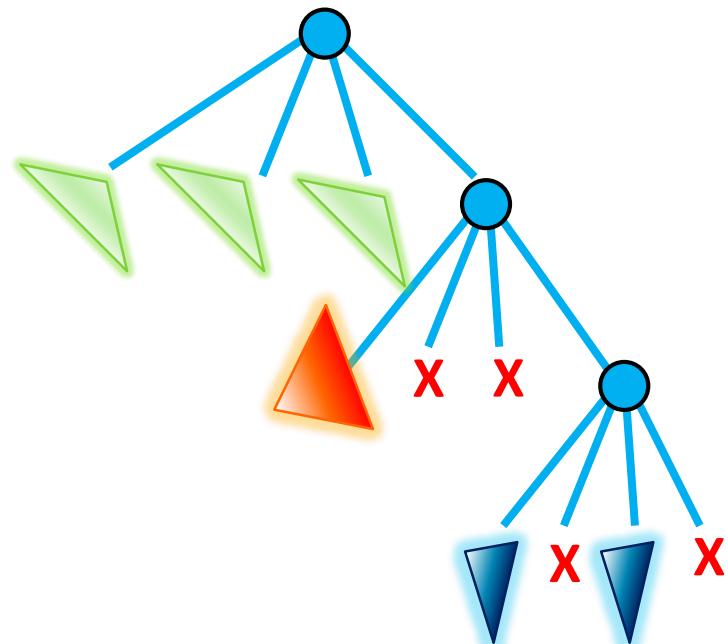
Oktalbaum/Octree

Traversierung (Primitive nur in Blättern)

- hier: Verweise auf Primitive nur in Blättern des Baums, ggf. mehrfach
- Schnittpunkte mit einem Primitiv werden nur berücksichtigt, wenn sie in der gerade betrachteten Zelle liegen



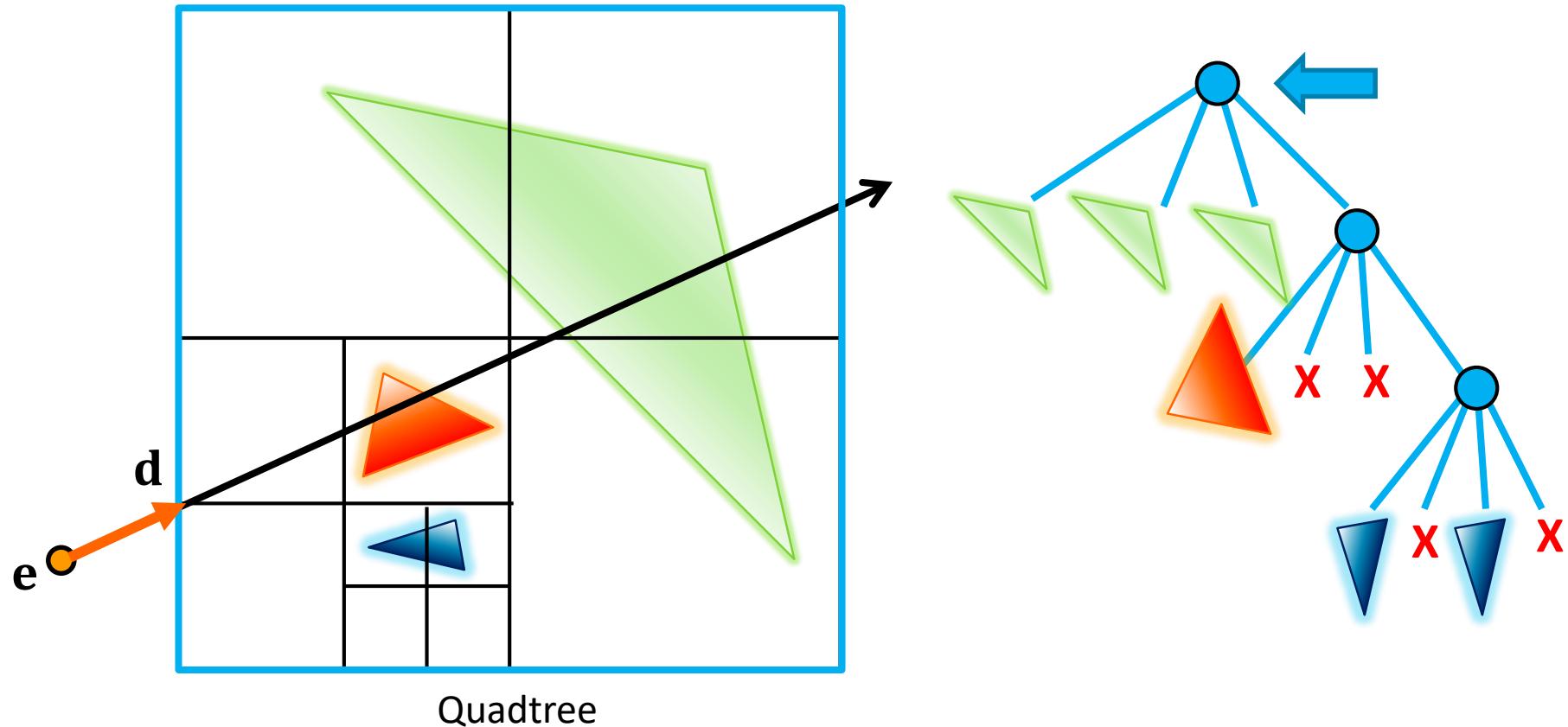
Oktalbaum (Octree), hier in
2D dargestellt als Quadtree



Oktbaum/Octree

Traversierung (Primitive nur in Blättern)

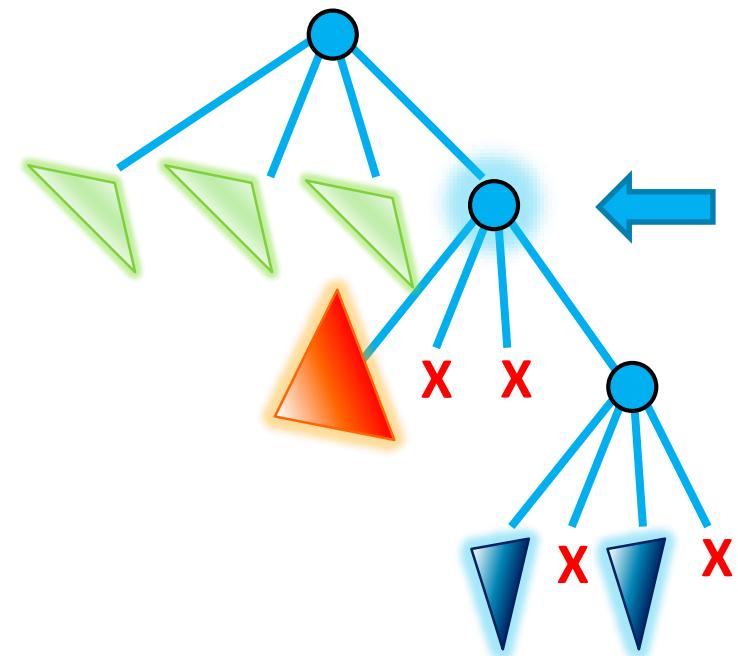
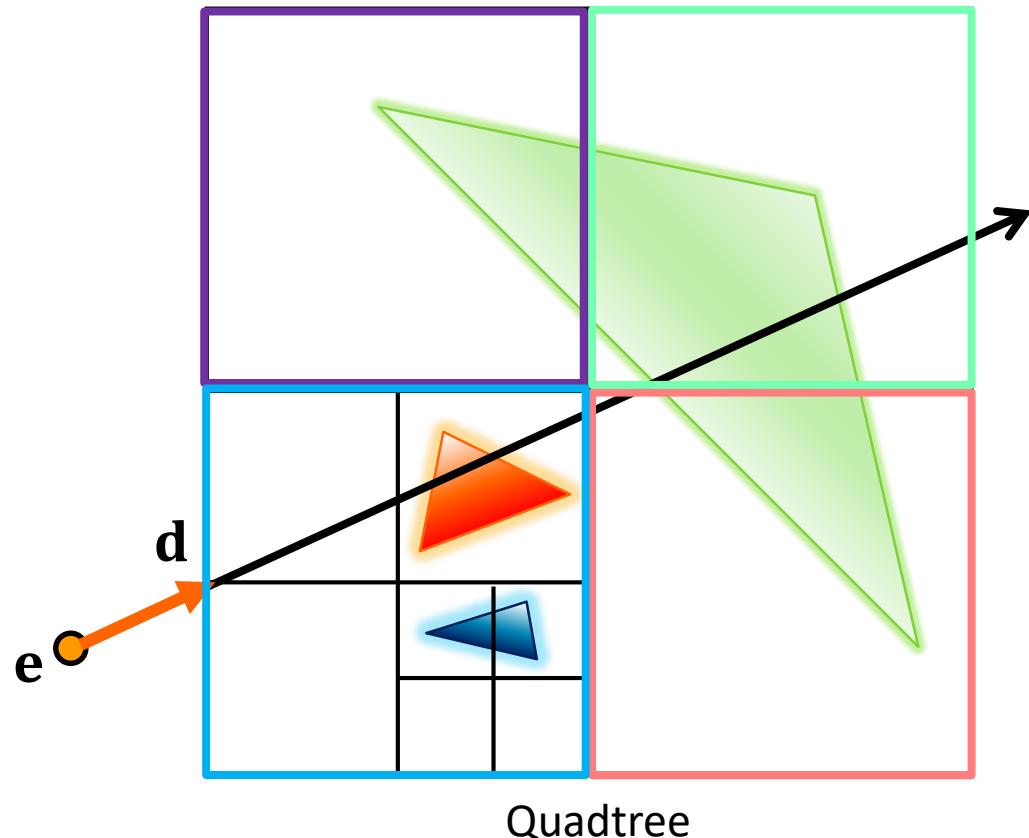
- ▶ Test gegen AABB der Wurzel (= AABB der gesamten Szene)
- ▶ wenn Schnitt mit AABB → weiter mit Kindknoten



Oktalbaum/Octree

Traversierung (Primitive nur in Blättern)

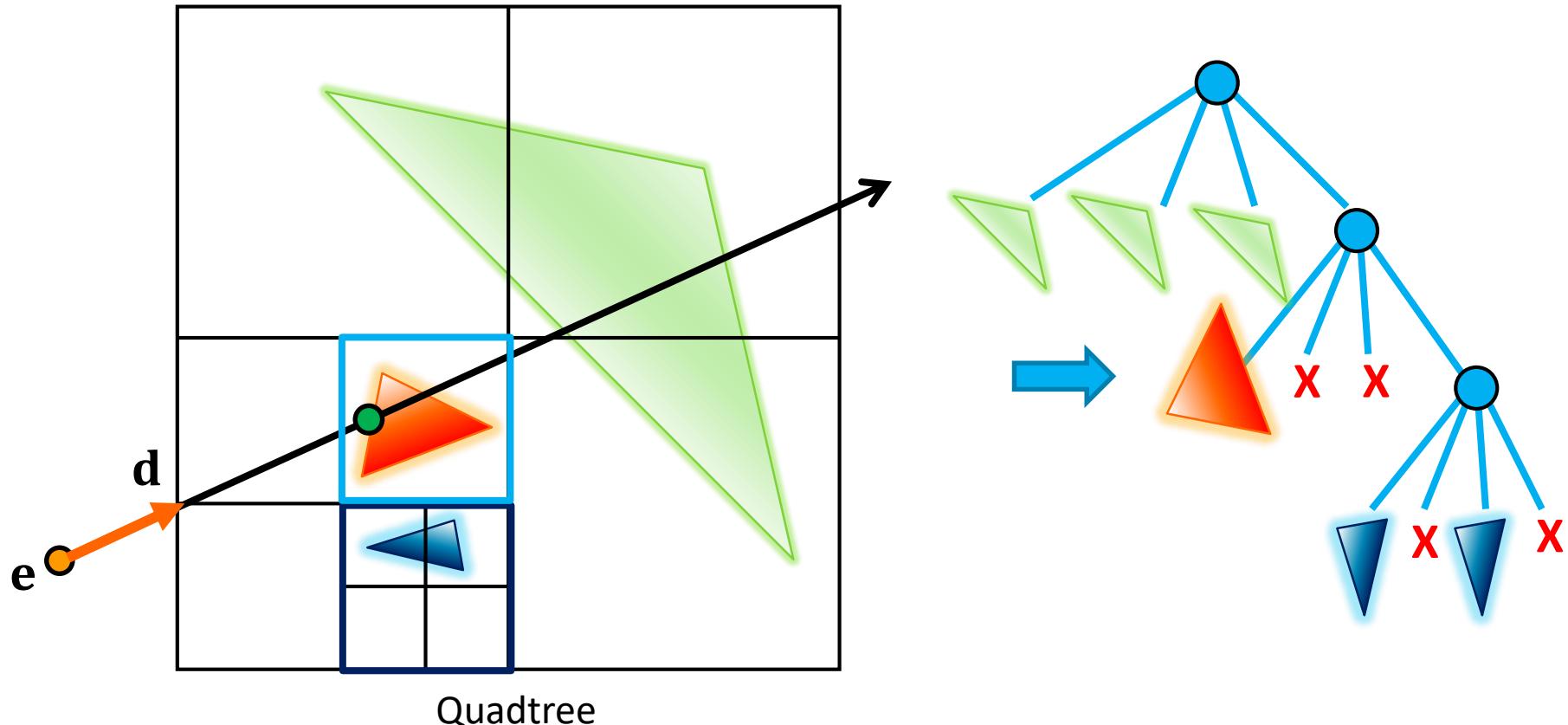
- ▶ überprüfe alle Kindknoten □□□□
- ▶ traversiere die nicht-leeren von vorne nach hinten: □□□



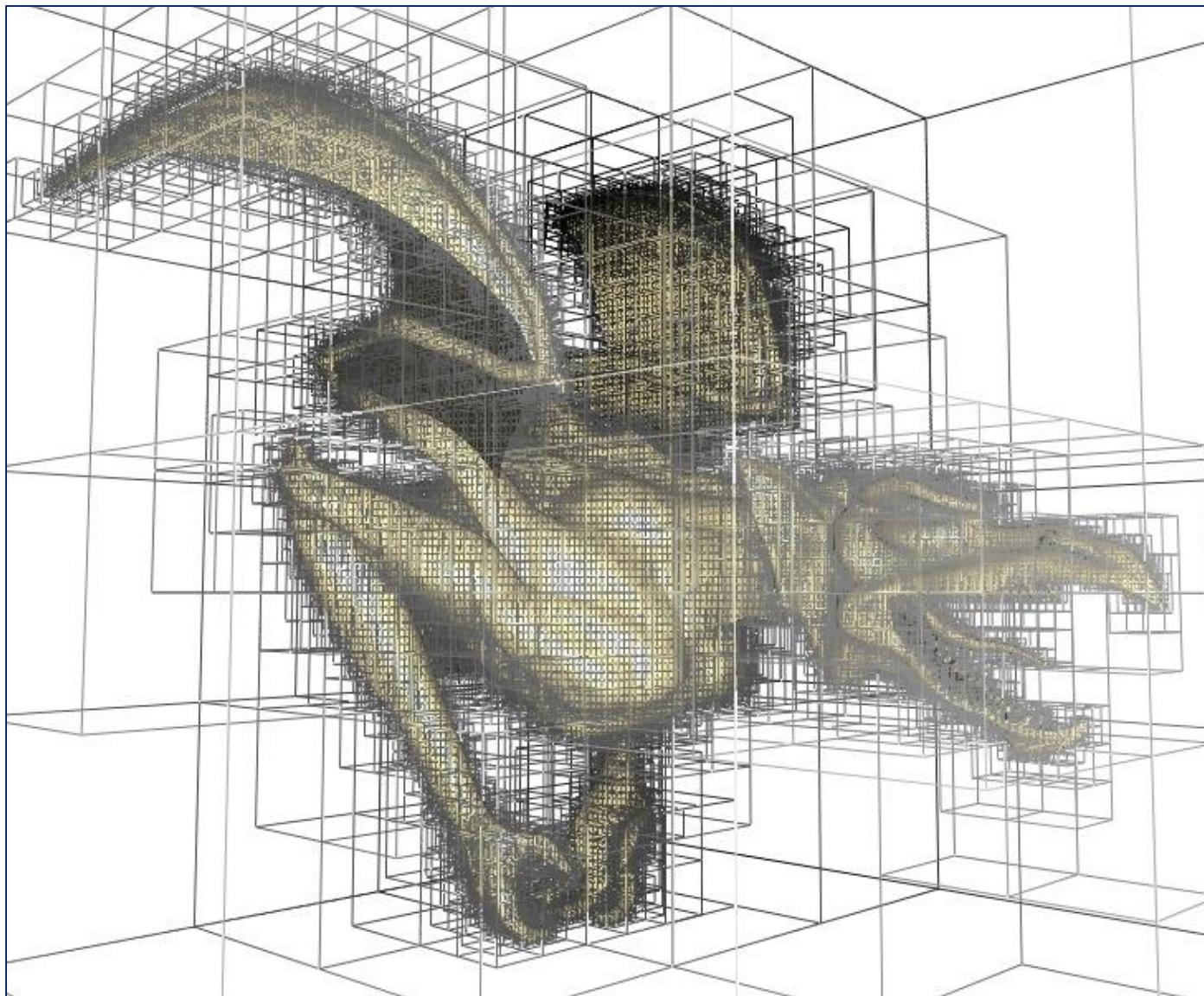
Oktalbaum/Octree

Traversierung (Primitive nur in Blättern)

- teste wiederum nicht-leeren Kindknoten von vorne nach hinten
 - ▶ nur einer wird geschnitten → in dieser Zelle existiert ein Schnittpunkt
 - ▶ damit sind wir fertig – hätten wir nichts geschnitten, dann müssten wir u. U. wieder in der Hierarchie aufsteigen und weitersuchen



Oktalbaum/Octree



[Lefebvre 2004]

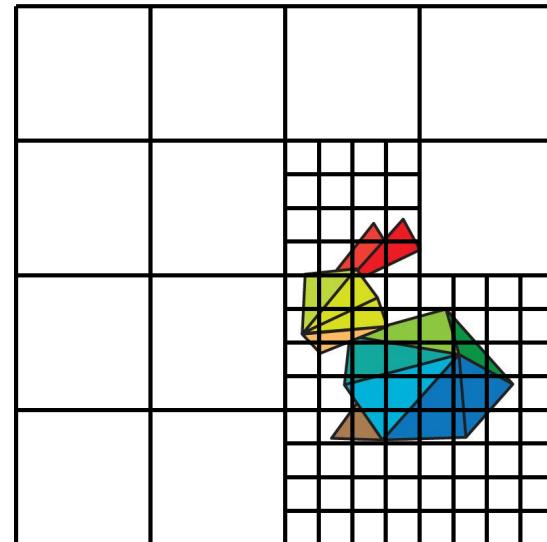
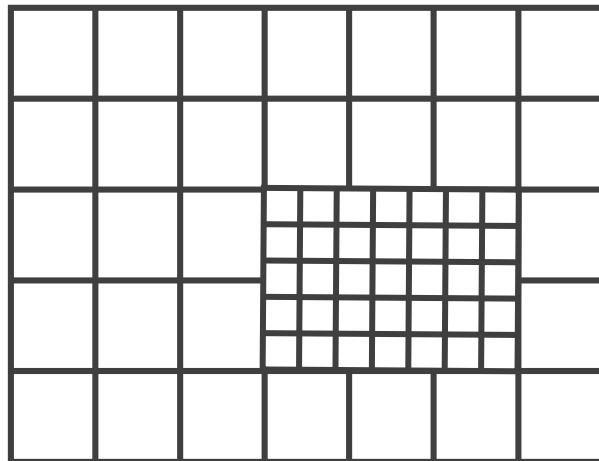
Verschachtelte Gitter

Kombination aus regulären Gittern und weiterer Unterteilung

- ▶ nutze effiziente Traversierung von regulären Gittern
- ▶ nutze Adaptivität einer hierarchischen Repräsentation

Aufbau

- ▶ grobes Gitter für Bounding Box der Szene
- ▶ Zellen mit dicht gepackten Objekten werden durch neues Gitter ersetzt
- ▶ benötigt Heuristiken: wann Gitter welcher Auflösung?



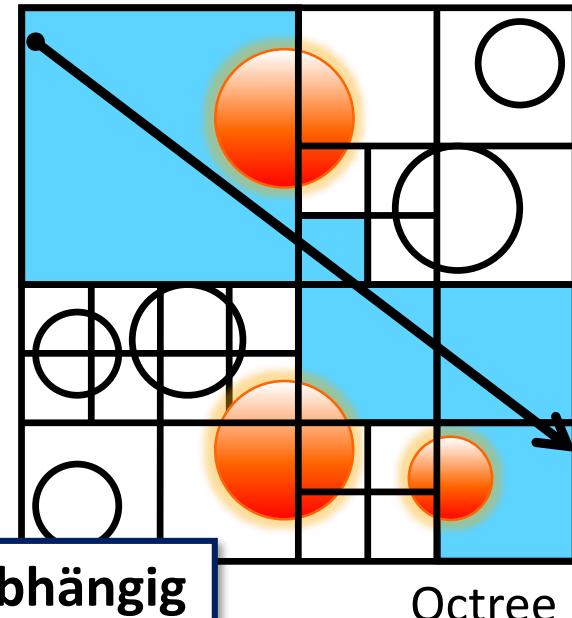
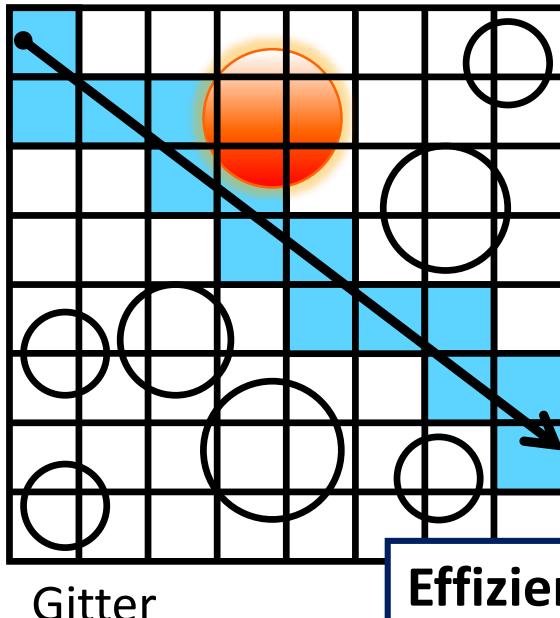
Vergleich Gitter vs. Octree

Gitter

- einfacher Algorithmus
 - schlecht bei ungleich verteilter Geometrie: kein effizientes Überspringen von leerem Raum
 - adaptiv nur durch verschachtelte Gitter
 - ähnlicher Aufwand für ähnliche Strahlen

Octree

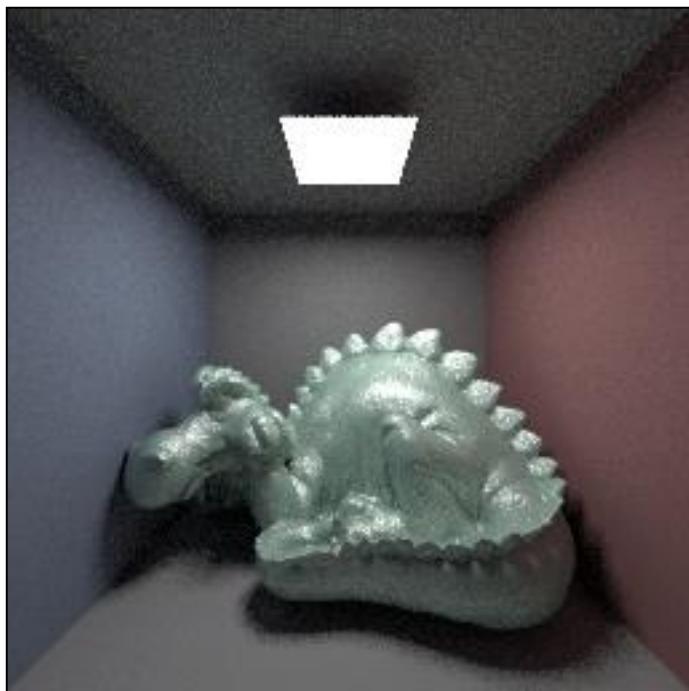
- Adaptivitt mit feiner Granularitt
 - hufige Vertikalbewegung
(auf und ab in der Hierarchie)
 - Traversierungskosten fr zwei
hnliche Strahlen knnen strker
variieren (schwierig bei guter
Ausnutzung paralleler Hardware)



Effizienz ist immer szenenabhängig

Inhalt: Räumliche Datenstrukturen

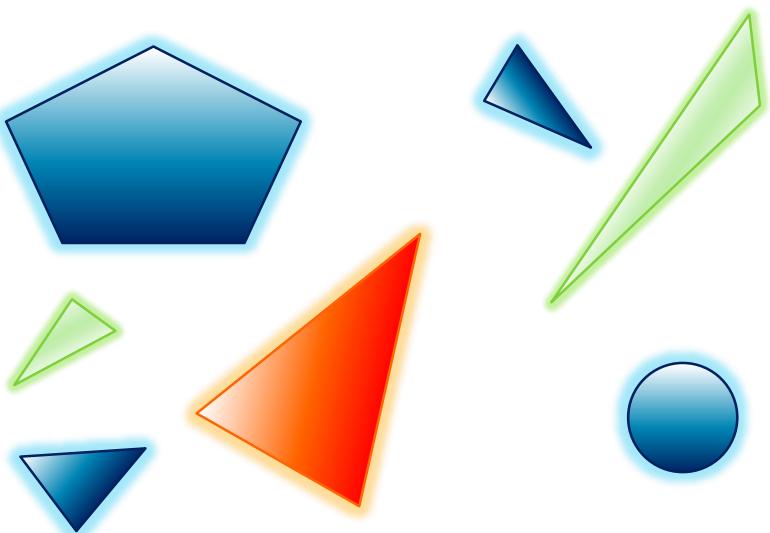
- ▶ Analyse der Kosten bei Raytracing
- ▶ Ansätze zur Beschleunigung von Raytracing
- ▶ Bounding Volumes (Hüllkörper)
- ▶ Räumliche Datenstrukturen
 - ▶ **Bounding Volume-Hierarchies**
 - ▶ **reguläre und adaptive Gitter**
 - ▶ **BSP-Bäume und kD-Bäume**



BSP-Baum und kD-Baum

Grundidee (Binary Space Partitioning Tree)

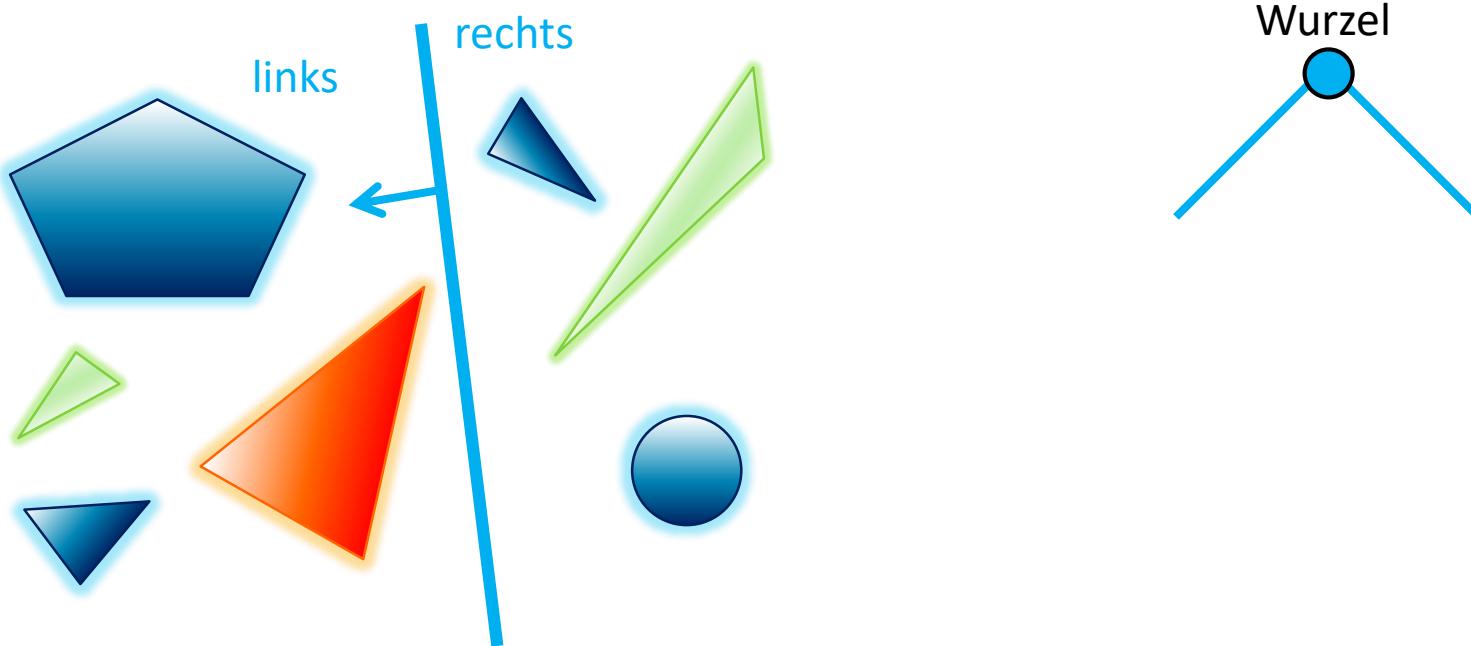
- ▶ Erweiterung von Binärbäumen auf k Dimensionen
- ▶ verwende Ebenen um den Raum rekursiv zu unterteilen
→ ergibt Binärbaumstruktur



BSP-Baum und kD-Baum

Grundidee (Binary Space Partitioning Tree)

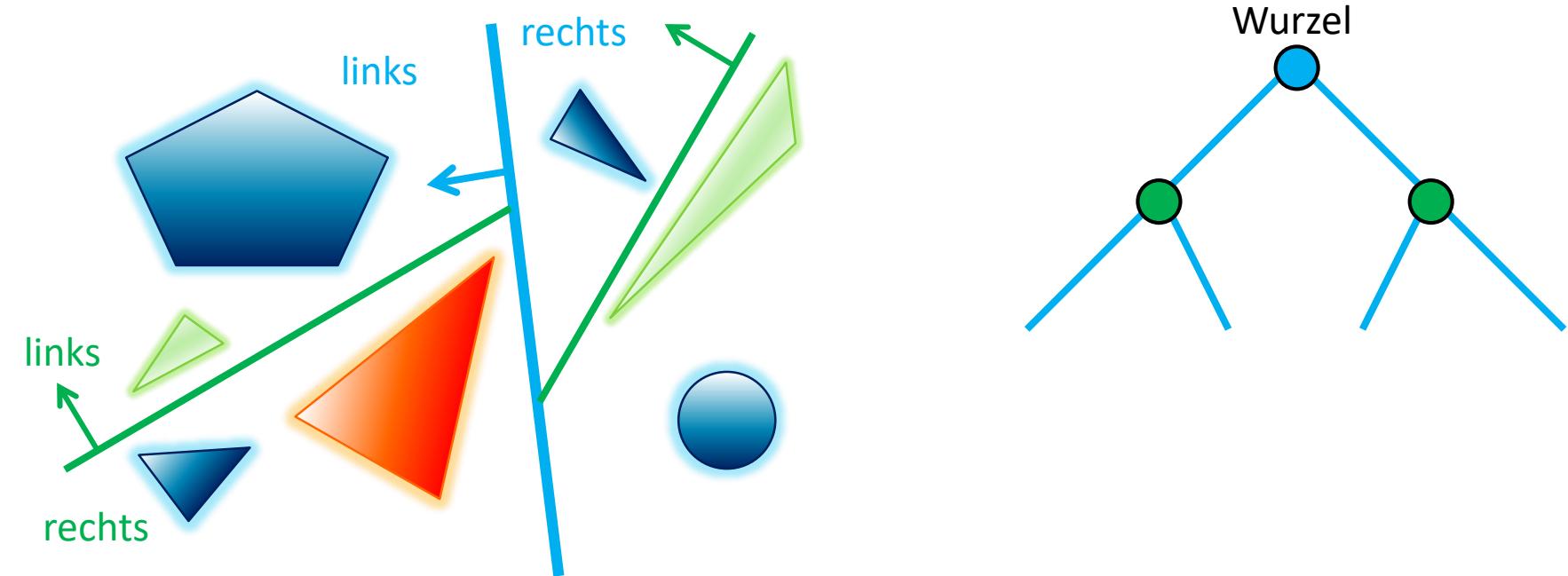
- Erweiterung von Binärbäumen auf k Dimensionen
- verwende Ebenen um den Raum rekursiv zu unterteilen
→ ergibt Binärbaumstruktur



BSP-Baum und kD-Baum

Grundidee (Binary Space Partitioning Tree)

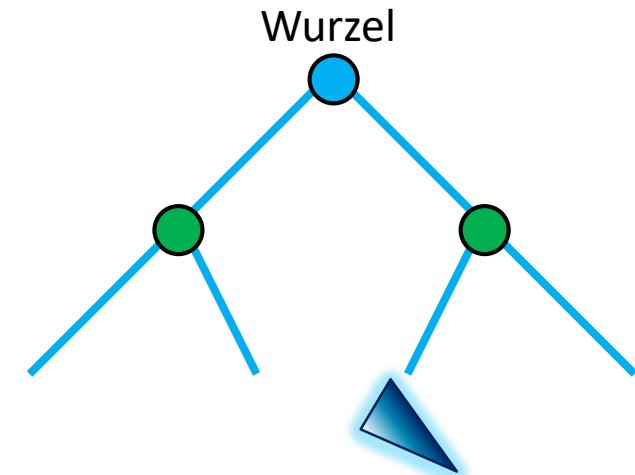
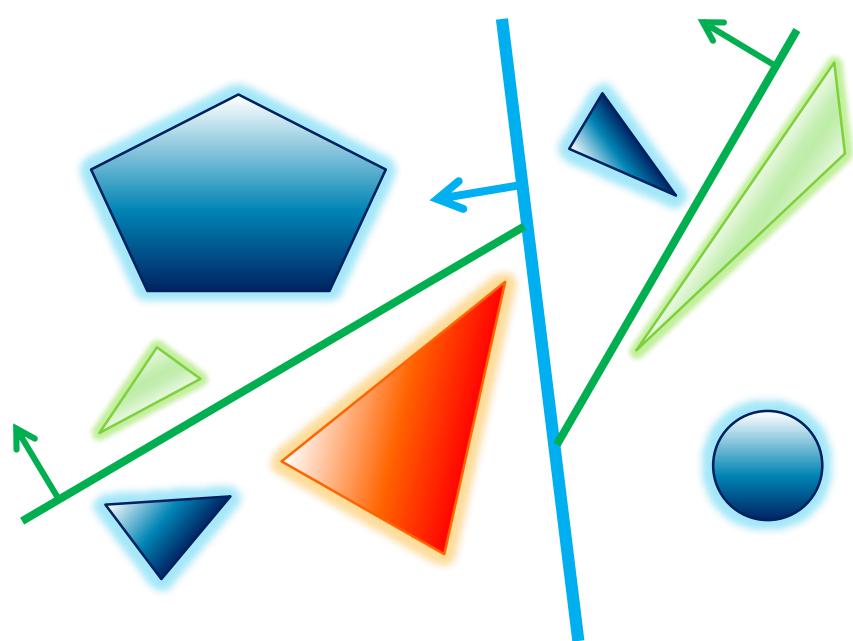
- Erweiterung von Binärbäumen auf k Dimensionen
- verwende Ebenen um den Raum rekursiv zu unterteilen
→ ergibt Binärbaumstruktur



BSP-Baum und kD-Baum

Grundidee (Binary Space Partitioning Tree)

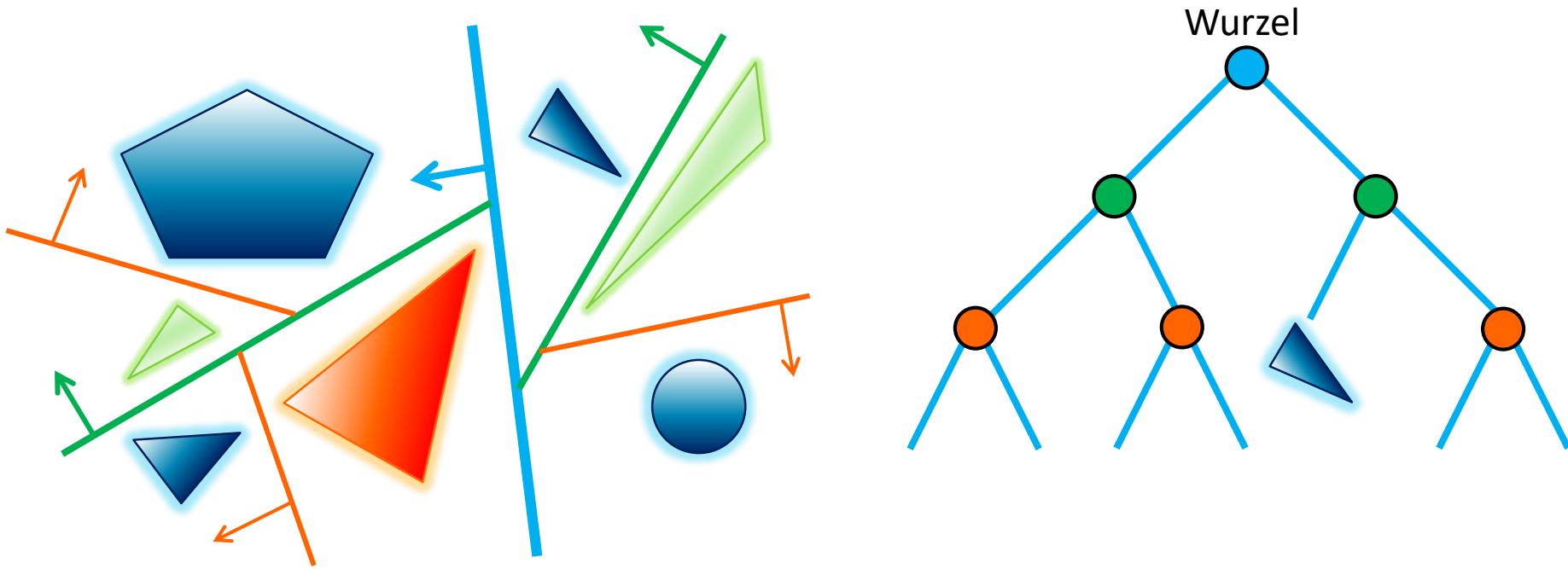
- Erweiterung von Binärbäumen auf k Dimensionen
- verwende Ebenen um den Raum rekursiv zu unterteilen
→ ergibt Binärbaumstruktur



BSP-Baum und kD-Baum

Grundidee (Binary Space Partitioning Tree)

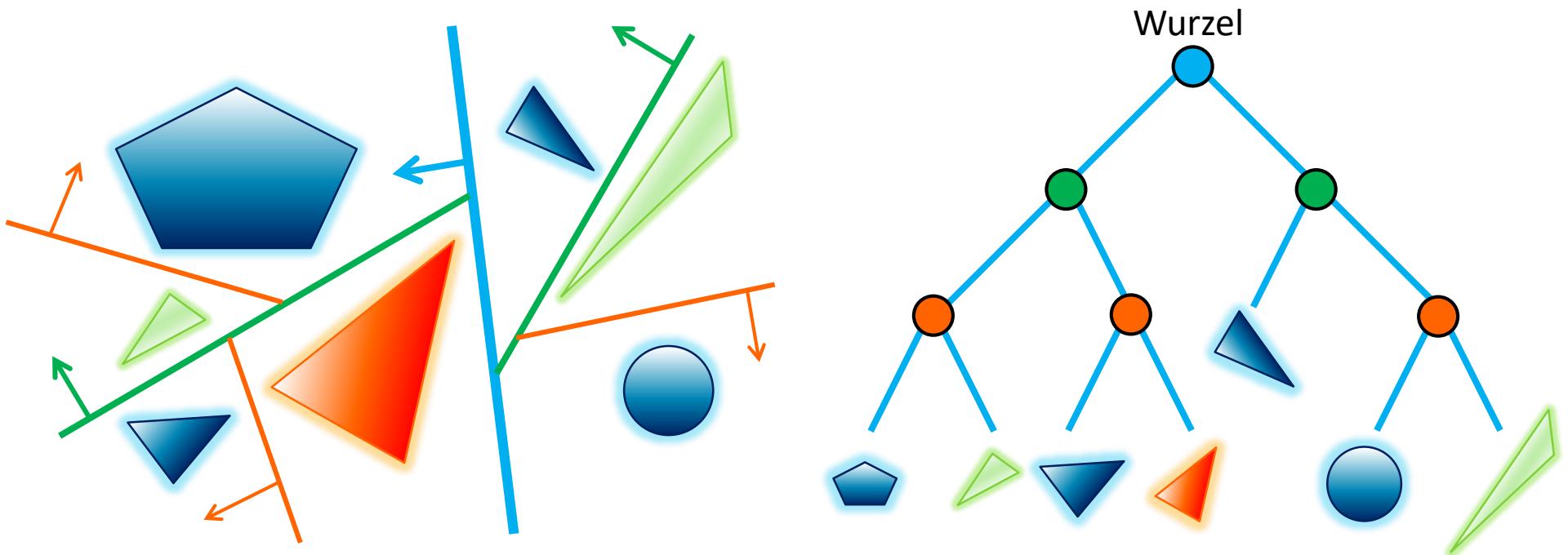
- Erweiterung von Binärbäumen auf k Dimensionen
- verwende Ebenen um den Raum rekursiv zu unterteilen
→ ergibt Binärbaumstruktur



BSP-Baum und kD-Baum

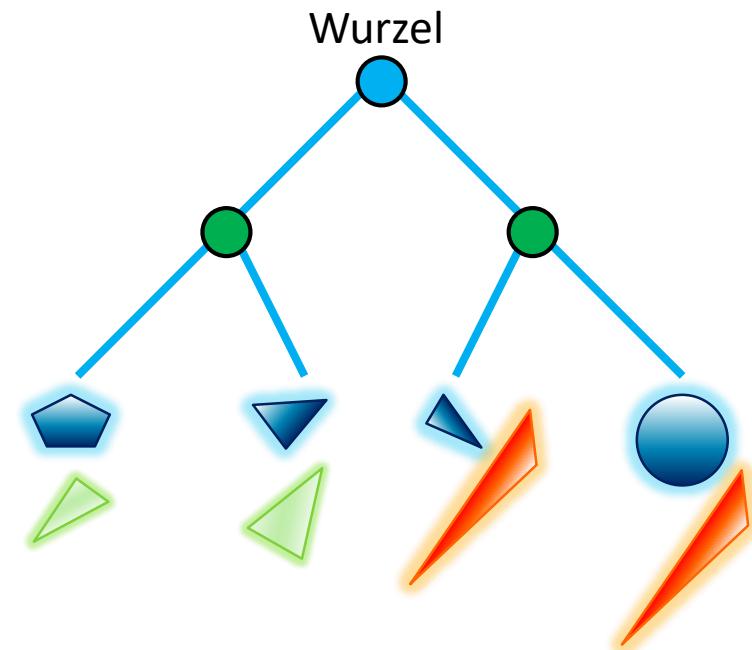
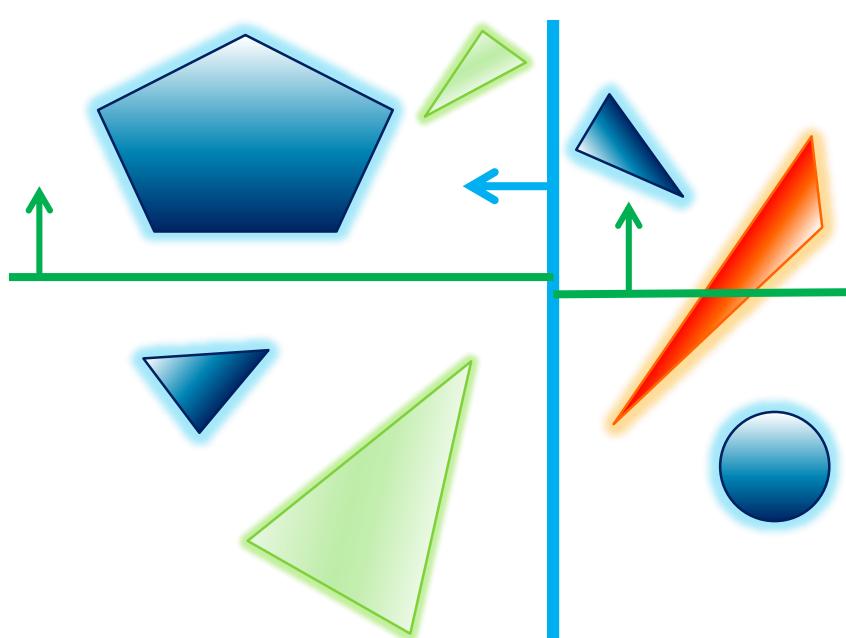
Grundidee (Binary Space Partitioning Tree)

- Erweiterung von Binärbäumen auf k Dimensionen
- verwende Ebenen um den Raum rekursiv zu unterteilen
→ ergibt Binärbaumstruktur
- Unterschied: BSP-Bäume verwenden beliebig orientierte Ebenen, kD-Bäume nur Ebenen, die senkrecht zur x -, y - oder z -Achse sind



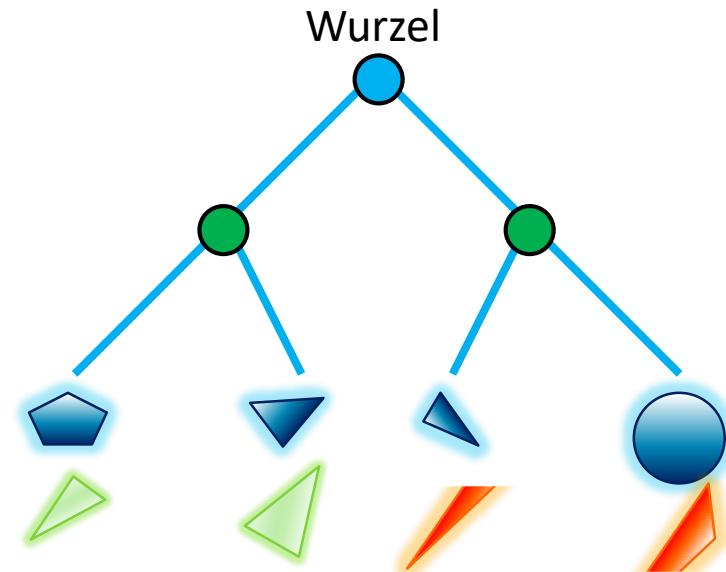
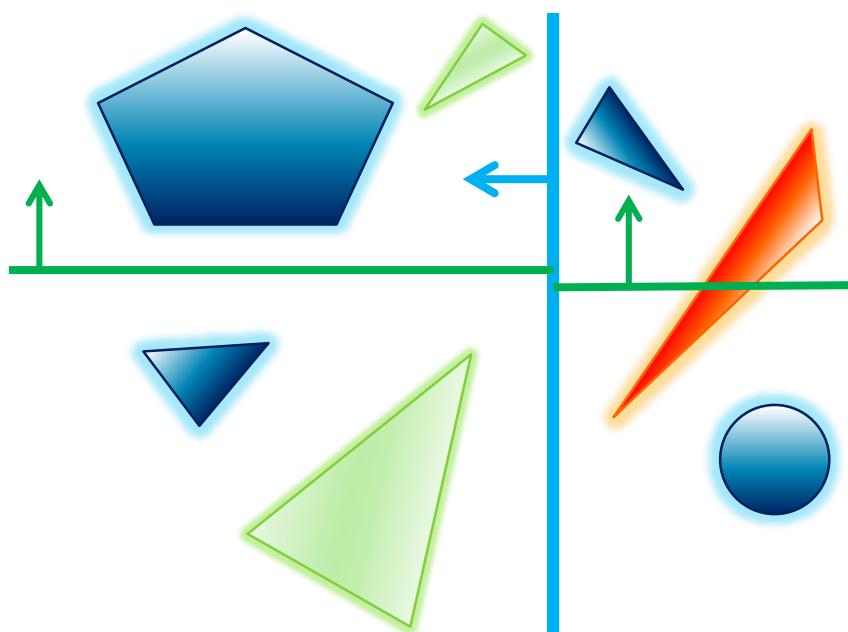
Eigenschaften

- ▶ Split-Ebenen senkrecht zur x -, y - oder z -Achse
- ▶ Primitive die eine Split-Ebene schneiden (gilt ebenfalls für BSP-Bäume) werden meist in beide Kindknoten eingefügt:
ein Primitiv kann also in mehreren Knoten vorkommen
(im Gegensatz zu BVH, Folge: Dreiecke werden mehrfach indiziert)



Eigenschaften

- ▶ Primitive die eine Split-Ebene schneiden (gilt ebenfalls für BSP-Bäume) werden meist in beide Kindknoten eingefügt:
ein Primitiv kann also in mehreren Knoten vorkommen
- ▶ optional: zerschneide das Primitiv
(in der Regel macht man das aber **nicht!**)

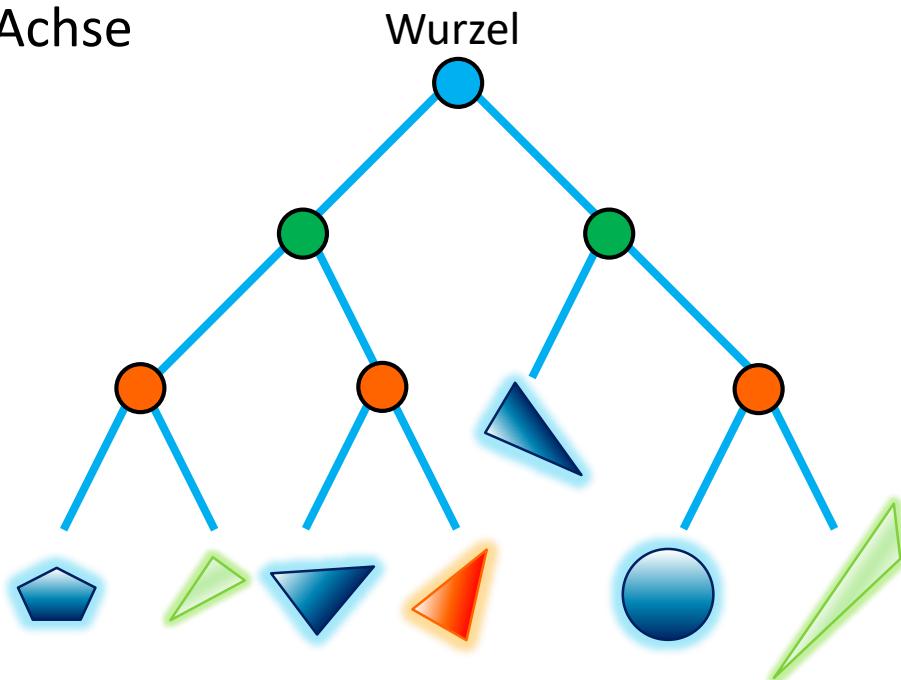


Konstruktion

- ▶ Konstruktion findet rekursiv, ähnlich wie BVH, statt
 - ▶ initialisiere Wurzelknoten: enthält alle Objekte/Primitive der Szene
 - ▶ unterteile den Wurzelknoten rekursiv...
 - ▶ ...bis ein Knoten nur noch eine vorgegebene maximale Anzahl Primitive enthält
 - ▶ ...oder eine maximale Rekursionstiefe erreicht ist
(um die heuristische Natur zu verdeutlichen: ein Erfahrungswert besagte früher einmal Tiefe = $8 + 1.3\log(n)$ für n Primitive)
 - ▶ **unterteile den Raum** und ordne die Primitive einem „linken“ und einem „rechten“ Kindknoten zu
(vgl. BVH: teilt die Primitive auf, die Hüllkörper ergeben sich daraus)

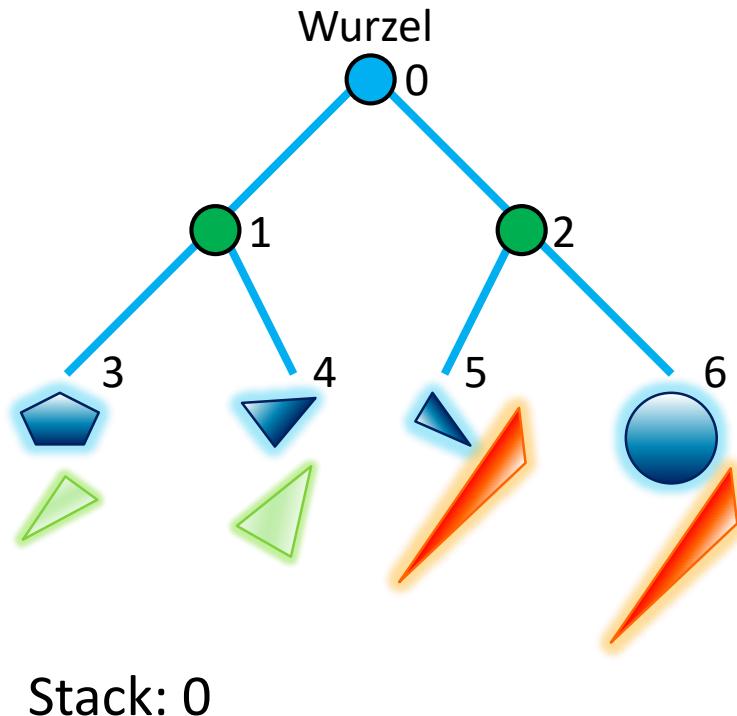
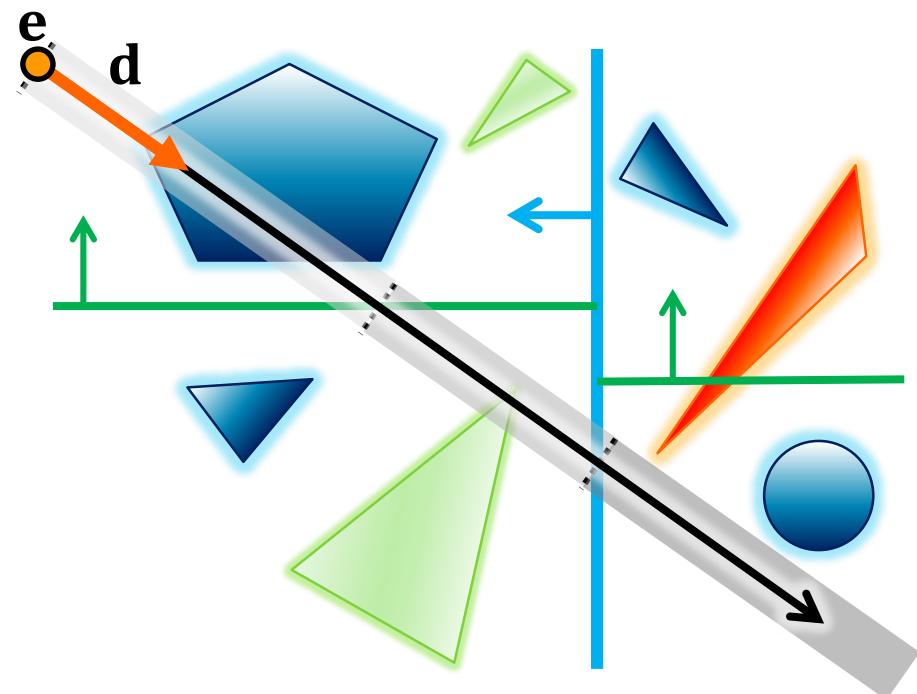
Eigenschaften

- ▶ echte Raumunterteilung: Knoten überlappen sich nicht (vgl. BVH)
- ▶ gute Anpassung an Geometrie möglich (v.a. bei BSP-Bäumen)
- ▶ die Blattknoten speichern die Primitive bzw. Verweise/Indizes darauf
- ▶ innere Knoten speichern die Split-Ebenen
 - ▶ BSP-Baum: Normale der Ebene und Abstand zum Ursprung
 - ▶ kD-Baum: die zur Ebene senkrechte Achse und die Position entlang dieser Achse
 - ▶ Zeiger auf die Kindknoten
- ▶ kD-Bäume sind einfacher zu konstruieren (keine Ebenen mit freier Orientierung) und werden daher häufiger eingesetzt



Traversierung

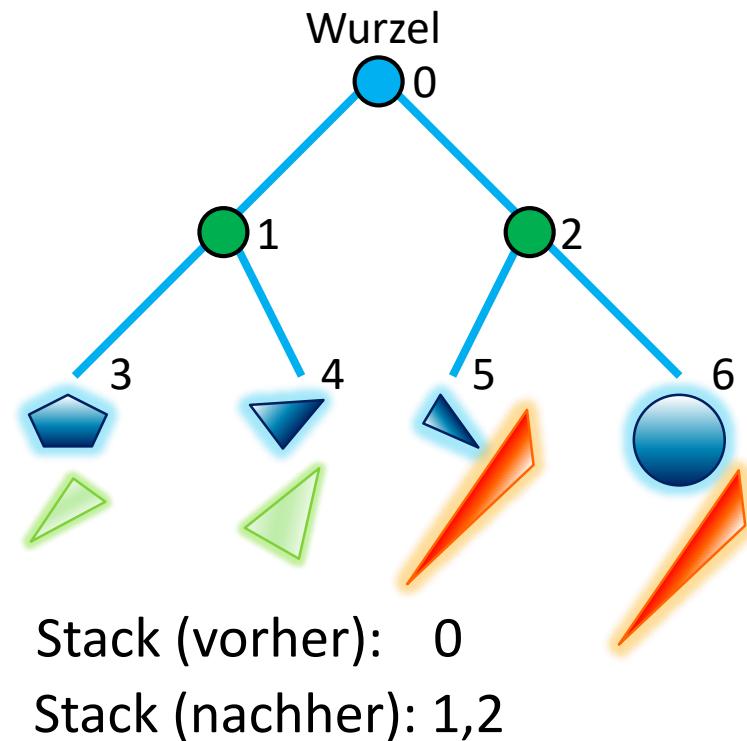
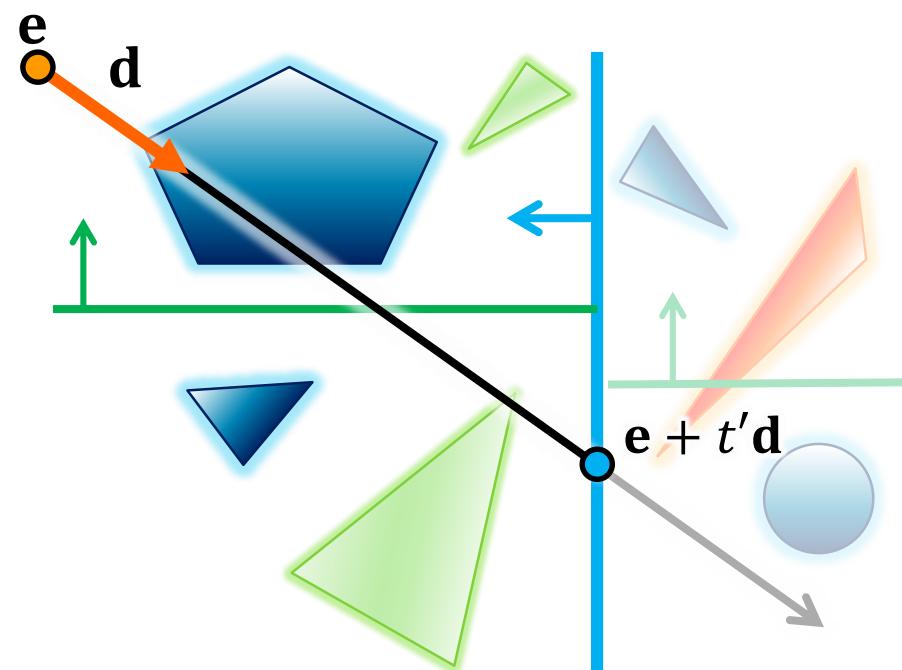
- Strahl $\mathbf{r}(t) = \mathbf{e} + t\mathbf{d}$, mit $t_{min} \leq t \leq t_{max}$ (t_{min}, t_{max} z.B. aus Schnitt mit AABB der ganzen Szene, oder $t_{min} = 0, t_{max} = \infty$)
- Ziel: Traversierung der Knoten „von vorne nach hinten“
- verwende Stack, um die Knoten für die Bearbeitung zu halten
- zu Beginn enthält der Stack den Wurzelknoten (0)



BSP-Baum und kD-Baum

Traversierung

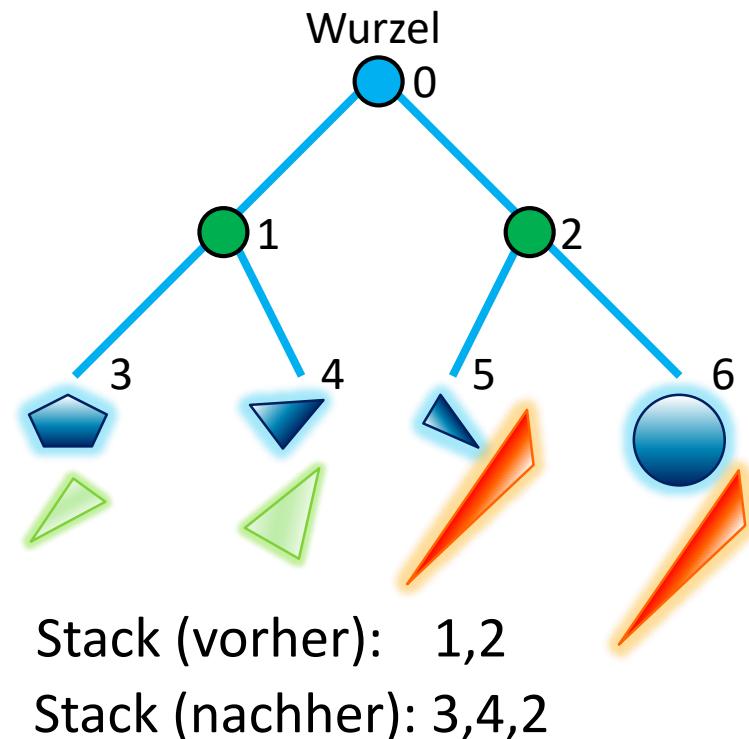
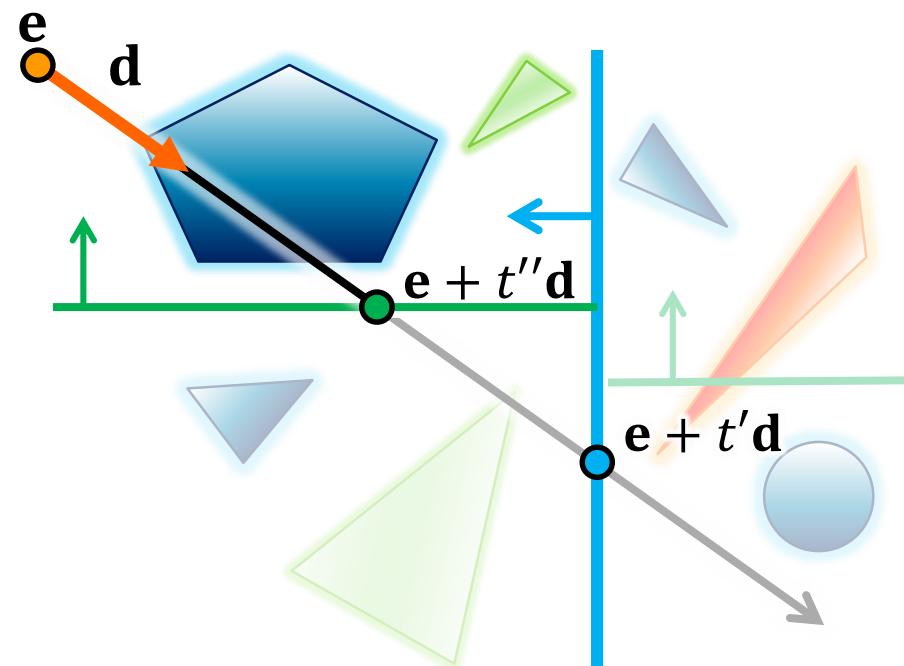
- ▶ Strahl $\mathbf{r}(t) = \mathbf{e} + t\mathbf{d}$, mit $0 = t_{min} \leq t \leq t_{max}$
- ▶ schneide Strahl mit der Split-Ebene (entnehme ersten Knoten vom Stack)
 - ▶ der Schnitt liegt bei t' mit $t_{min} \leq t' \leq t_{max}$
 - ▶ fahre mit beiden Kindern rekursiv fort
 - ▶ zuerst der Kindknoten, der \mathbf{e} enthält



BSP-Baum und kD-Baum

Traversierung

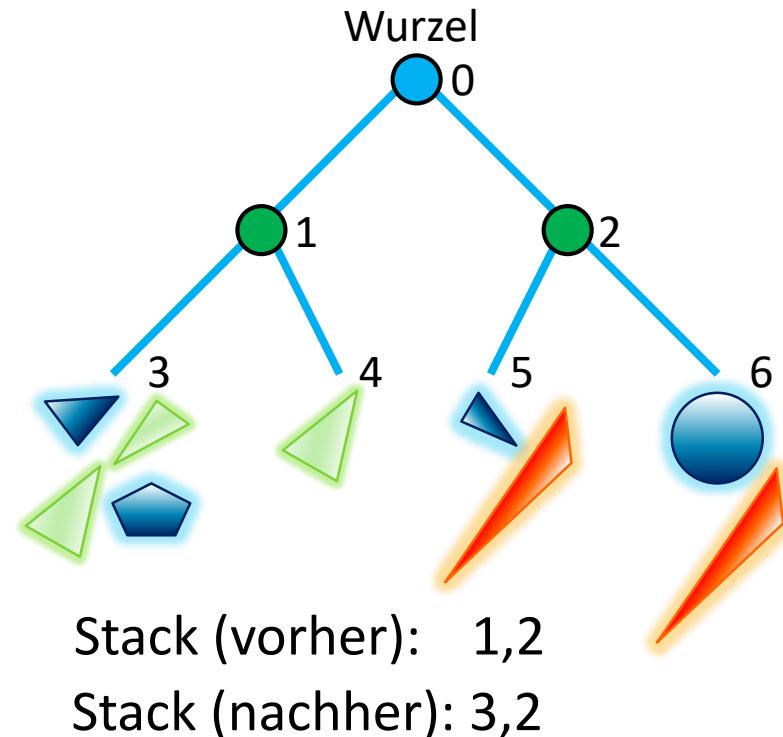
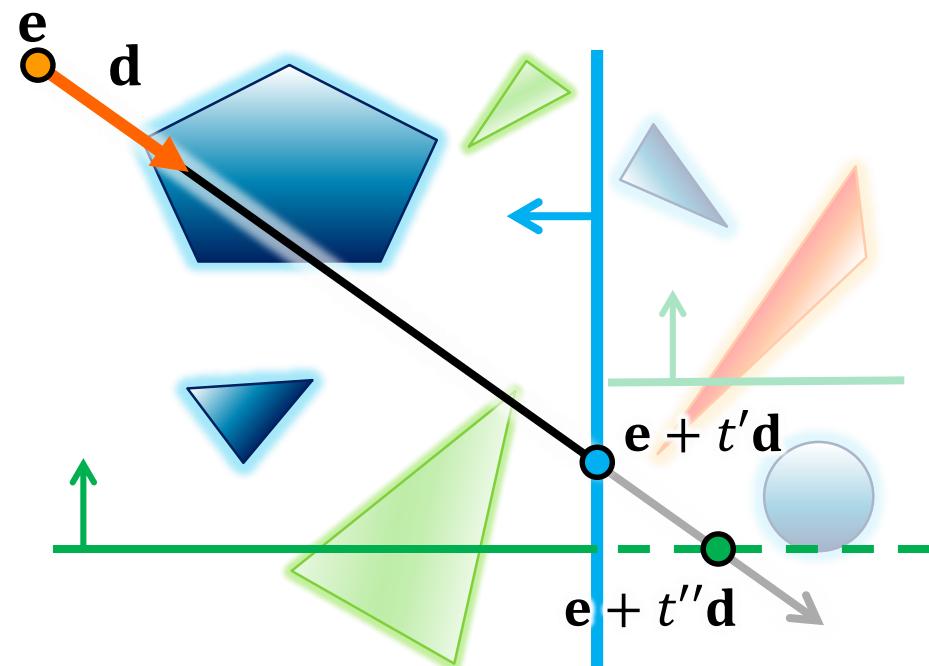
- ▶ schneide Strahl mit nächster Split-Ebene (entnehme Knoten 1 vom Stack)
- ▶ liegt der Schnitt bei t'' mit $t_{min} \leq t'' \leq t'$ (**das muss nicht so sein**)
 - ▶ dann fahre mit beiden Kindern rekursiv fort
 - ▶ zuerst der Kindknoten, der **e** enthält
 - ▶ (**der andere Fall kommt gleich**)



BSP-Baum und kD-Baum

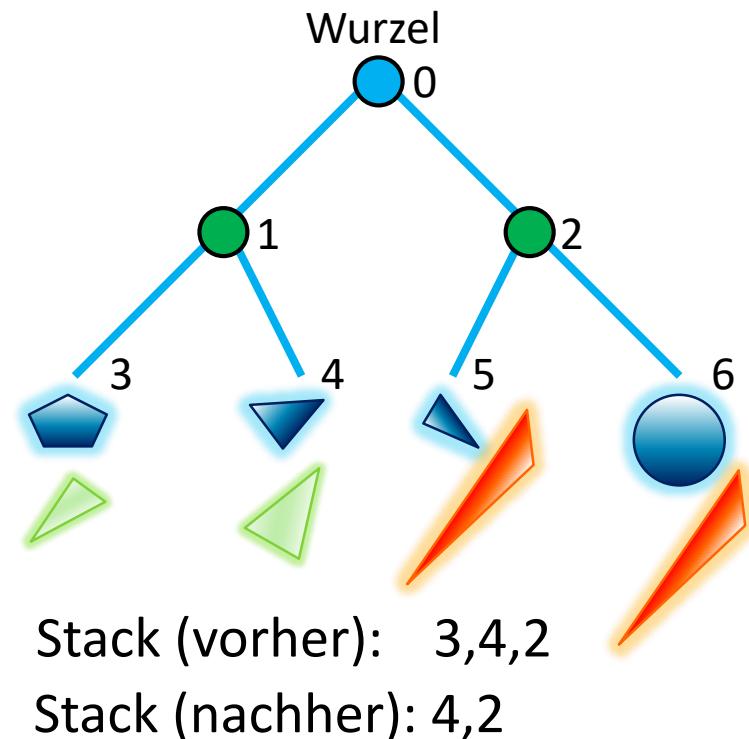
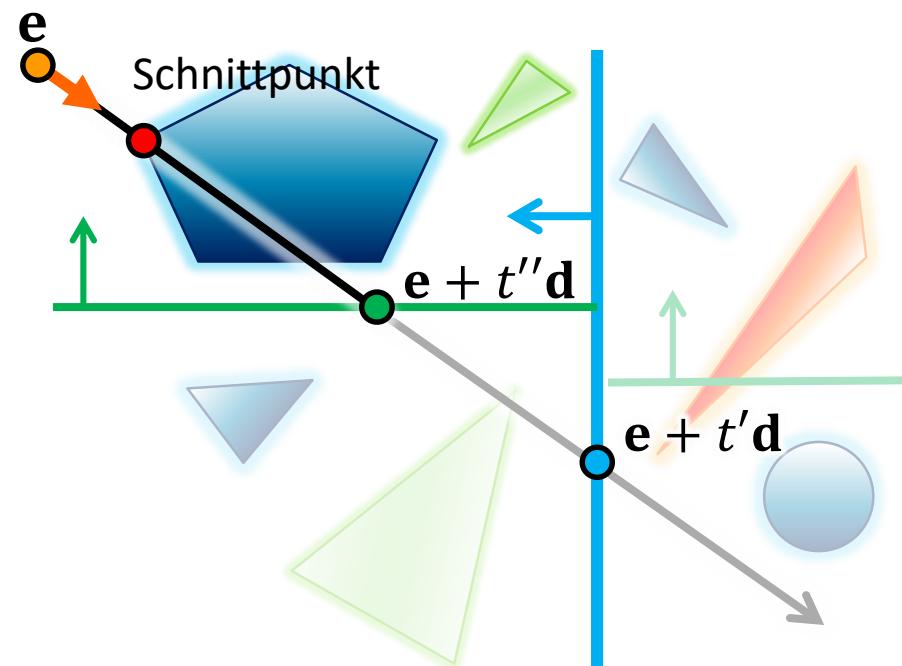
Traversierung: „der andere Fall“

- Achtung: nur zur Illustration, hat nichts mit dem Beispiel eben zu tun!
- schneide Strahl mit der Split-Ebene (entnehme Knoten 1 vom Stack)
- hier liegt der Schnitt bei t'' mit $t'' > t'$
- nur der Kindknoten, der e enthält wird traversiert (hier Knoten 3)
- hier: bevor Knoten 4 erreicht wird, verlassen wir den Teilbaum



Traversierung

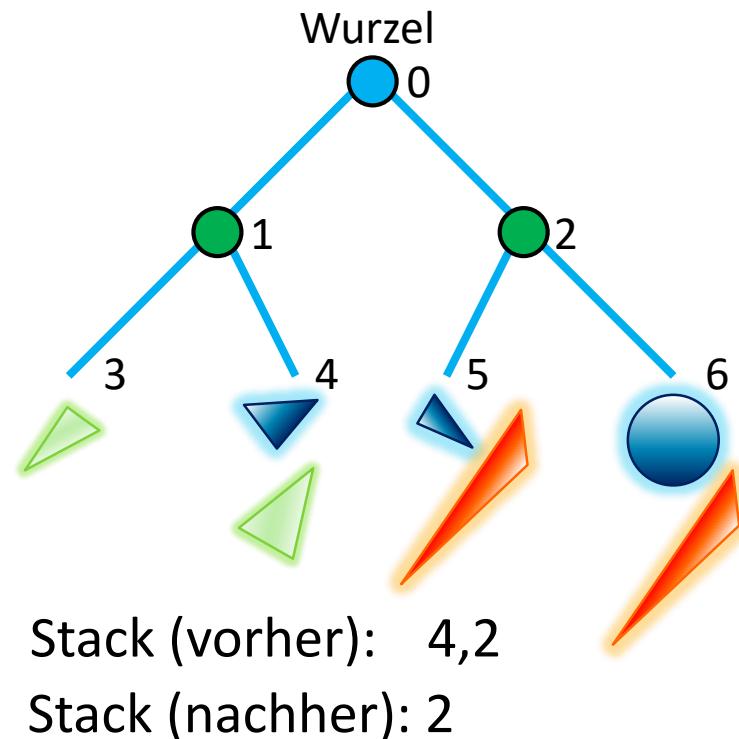
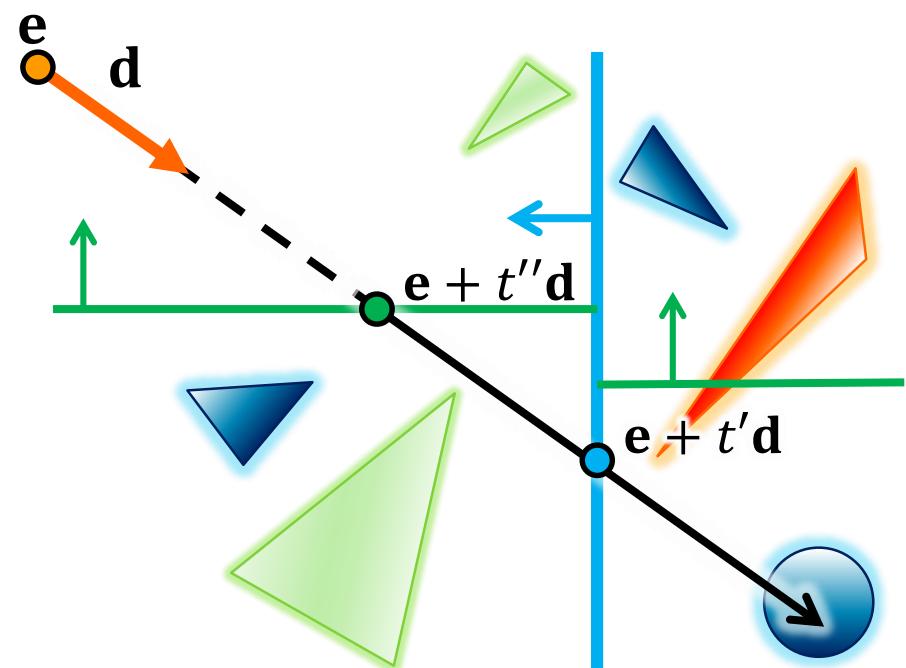
- ▶ wenn Blattknoten erreicht, dann teste Schnittpunkte mit Primitiven
- ▶ gebe nahsten Schnittpunkt zurück, wenn innerhalb $[t_{min}, t'')$
(bzw. $[t'', t')$, wenn Primitive im hinteren Halbraum getestet wurden)
- ▶ Vorteil „echter“ Raumunterteilung: kein möglicher Schnitt weiter hinten
- ▶ in diesem Beispiel sind wir jetzt fertig!



BSP-Baum und kD-Baum

Traversierung

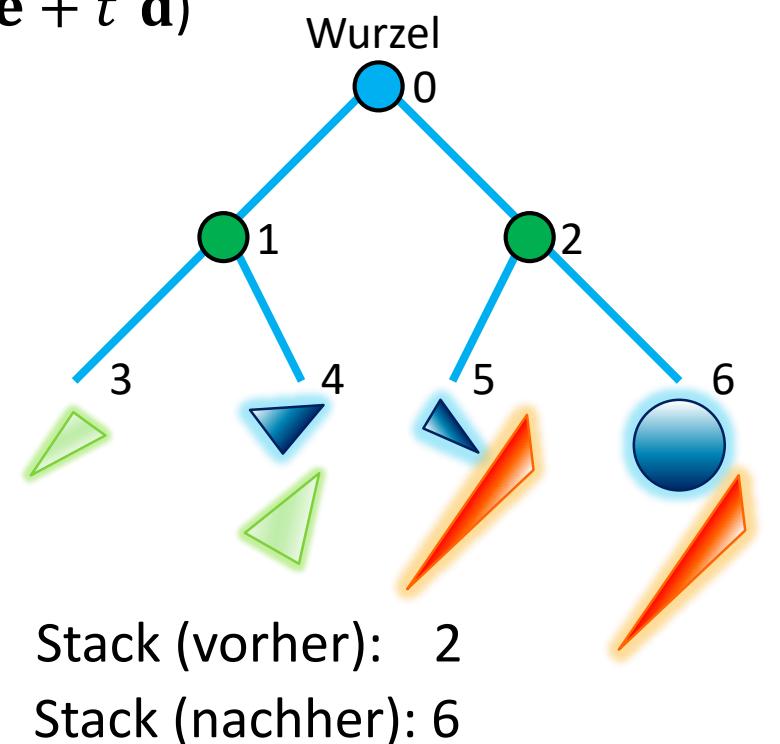
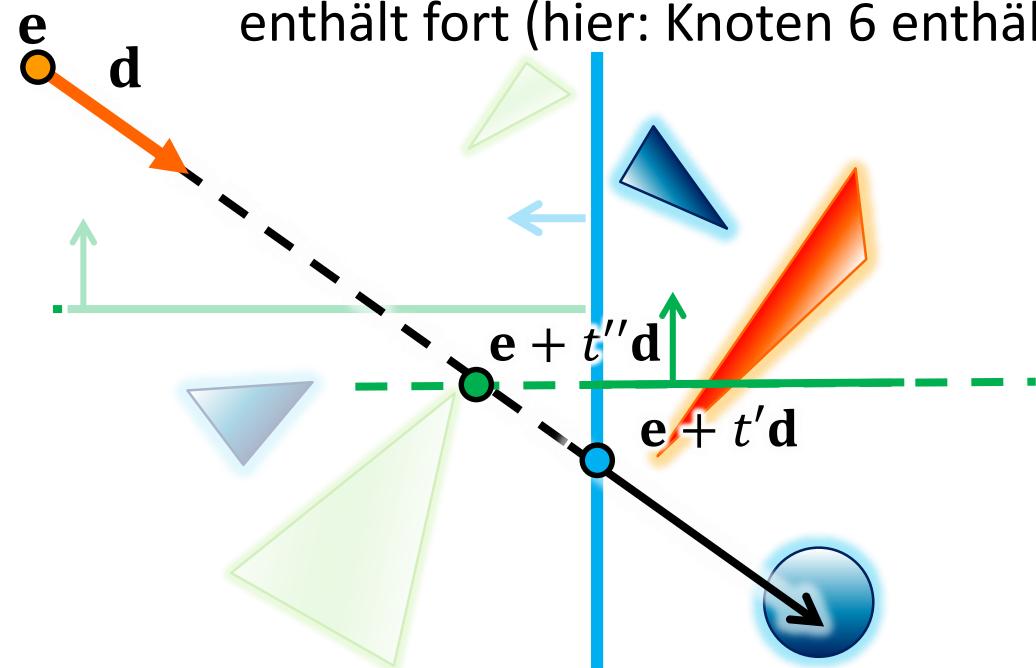
- keine Schnittpunkte im Knoten 3 gefunden
- fahre mit nächstem Knoten vom Stack fort → Knoten 4
- auch dort gibt es keine Schnittpunkte für $t'' \leq t \leq t'$



BSP-Baum und kD-Baum

Traversierung

- ▶ fahre mit nächstem Knoten vom Stack fort → Knoten 2
- ▶ schneide Strahl mit der Split-Ebene (Knoten 2)
 - ▶ liefert den Schnitt bei t''
 - ▶ wenn $t'' \in [t', t_{max}]$, dann wären wir in Knoten 5 (nicht der Fall)
 - ▶ wenn $t'' \notin [t', t_{max}]$, dann fahre mit dem Kind, das den Startpunkt ● enthält fort (hier: Knoten 6 enthält $e + t'd$)

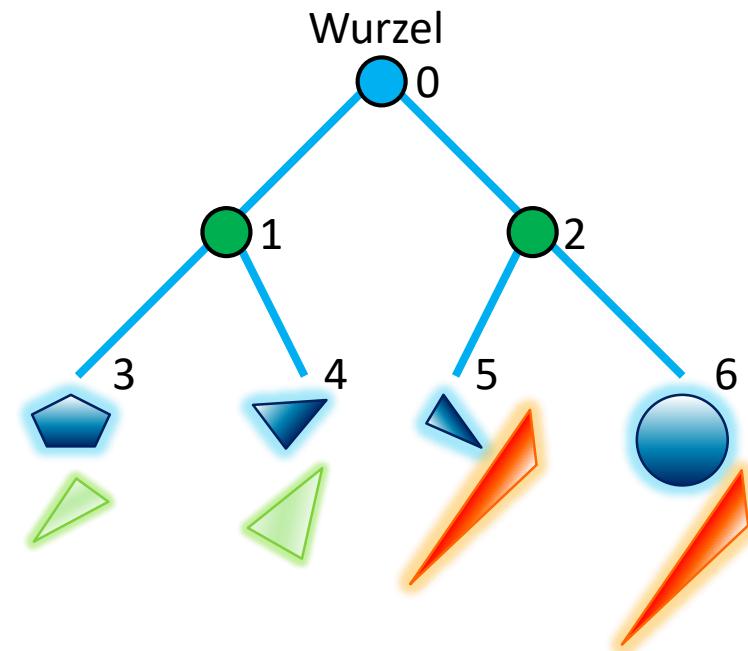
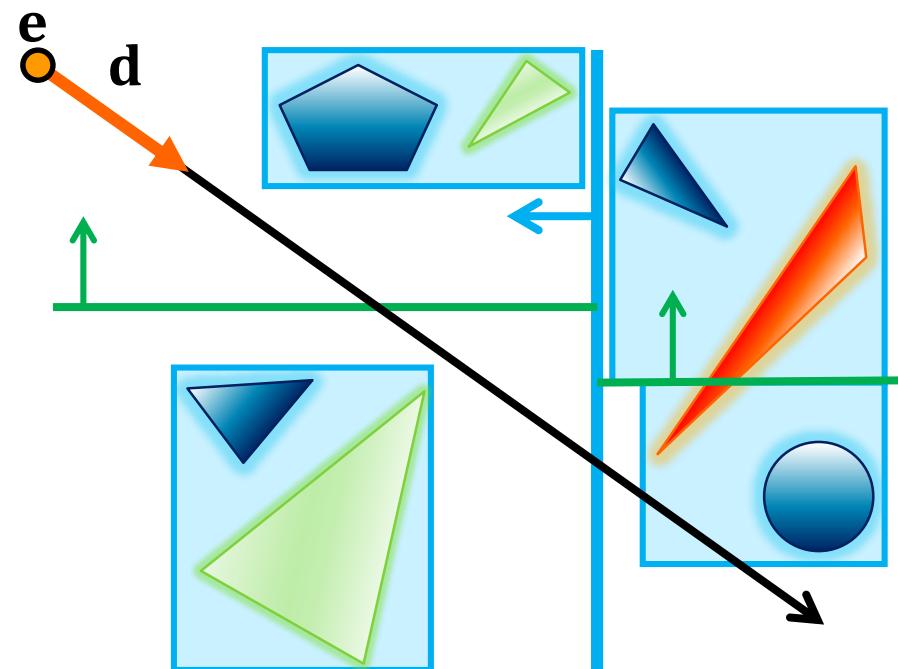


Traversierung

- ▶ Strahl $\mathbf{r}(t) = \mathbf{e} + t\mathbf{d}$, mit $0 \leq t \leq t_{max}$
- ▶ schneide Strahl mit der Split-Ebene (beginnend beim Wurzelknoten)
 - ▶ Schnitt mit der Split-Ebene bei t'
 - ▶ wenn t' auf dem Strahlsegment liegt:
traversiere beide Kinder rekursiv, zuerst das Kind, das \mathbf{e} enthält
 - ▶ liegt t' nicht auf dem Strahlsegment:
dann fahre nur mit dem Kind, das geschnitten wird fort
 - ▶ unterteile das Strahlsegment weiter mit t'', t''', \dots
 - ▶ wenn Blattknoten erreicht:
 - ▶ teste Schnittpunkte mit Primitiven
 - ▶ gebe Schnittpunkt zurück, wenn innerhalb des aktuellen Strahlsegments (= innerhalb des Teilraums)
 - ▶ Reihenfolge der Traversierung der Knoten: von vorne nach hinten!
- ▶ Komplexität: $O(\log n)$ Schnitttests für n Objekte/Primitive in der Szene
- ▶ Anm. Traversierung auch ohne Stack möglich, siehe z.B.
Efficient stackless hierarchy traversal on GPUs with backtracking in constant time,
Binder und Keller, High Performance Graphics'16

Kombination mit AABBs

- manchmal speichert man für die Primitive eines Knotens zus. eine AABB
- hilfreich, wenn Primitive nur einen kleinen Teil des Raums einnehmen
- für Schnitttest pro Knoten bedeutet das: teste nur auf Schnitt mit Primitiven, wenn die AABB geschnitten wird
- (Isolation dünnbesetzter Bereiche daher wünschenswert – gleich mehr)



Aufteilung (Split) eines Knotens für kD-Baum- und BVH-Konstruktion

- Bestimmung der „optimalen“ Split-Ebene
 - ▶ **räumliches Mittel (spatial median):**
teile Knoten in der Mitte entlang der Achse der größten Ausdehnung **oder**
teile erst entlang der x -, dann y -, dann z -, dann wieder x -Achse, usw.
 - ▶ Aufbau $O(n \log n)$: Tiefe des Baums $\log n$, je n Primitive zugeordnet
 - ▶ **Objektmittel (object median):**
teile Knoten so, dass linker und rechter Kindknoten gleich viele Primitive enthalten (typischerweise entlang größter Ausdehnung)
 - ▶ Aufbau $O(n \log^2 n)$, wenn eine $O(n \log n)$ Sortierung verwendet wird
 - ▶ **Kostenfunktion (Surface Area Heuristic)**
 - ▶ Ziel: im Mittel sollen zufällige Strahlen, die den betrachteten Knoten schneiden, den gleichen Aufwand verursachen, egal welcher der Kindknoten weiter traversiert wird
 - ▶ resultiert in einem – **bei der Traversierung** – balancierten Baum
 - ▶ Aufbau $O(n \log^2 n)$ möglich
<http://www.cgg.cvut.cz/members/havran/ARTICLES/ingo06rtKdtree.pdf>
- Anm. die Wahl einer guten Split-Ebene ist bei kD-Bäumen schon schwierig – bei BSP-Bäumen ist der „Suchraum“ noch viel größer

Konstruktion „guter“ kD-Bäume und BVH



- wie unterteilt man Knoten am besten?
 - ▶ so, dass Raytracing schnell geht ☺
 - ▶ Idee: stelle eine Kostenfunktion auf und minimiere
- Kosten für das Traversieren eines unterteilten/inneren Knotens
„wir stehen vor der Entscheidung: einen Knoten nochmal unterteilen, oder bei der Traversierung einfach alle Primitive schneiden“

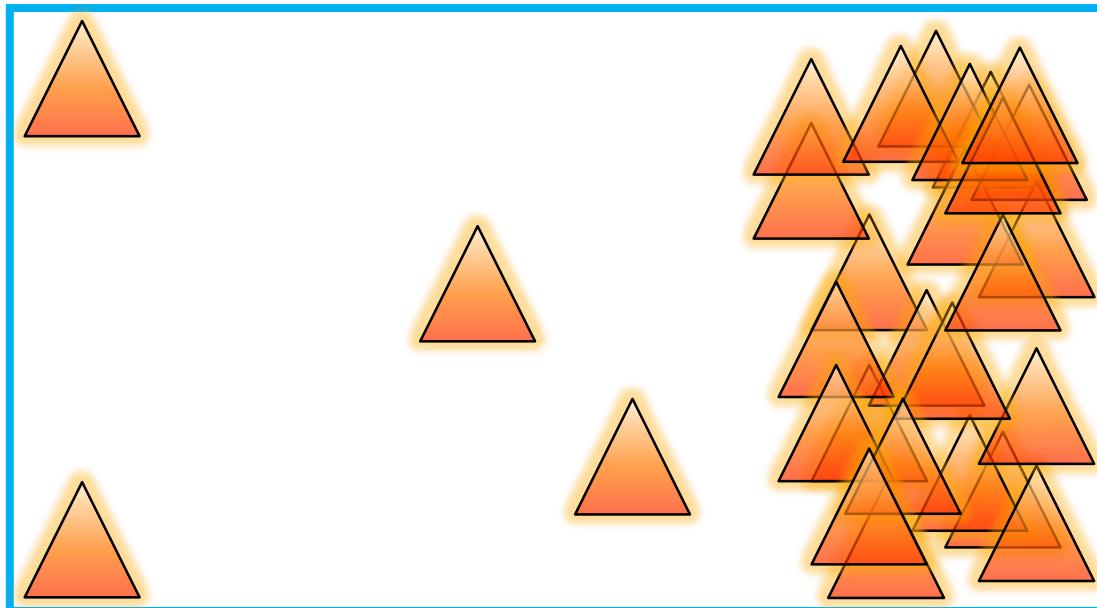
$$C = C_T + P(\text{treffe } L)C(L) + P(\text{treffe } R)C(R)$$

- ▶ C_T : Kosten für das Traversieren des (neuen) inneren Knotens
 (= Entscheidung, ob mit linkem oder rechtem Kind fortgefahrene wird)
- ▶ $P(\text{treffe } L), P(\text{treffe } R)$: Wahrscheinlichkeit, dass der Strahl den linken bzw. rechten Kindknoten trifft
- ▶ $C(L), C(R)$: Kosten für das Traversieren und Schnitttests des linken und rechten Kindknotens
- ▶ wichtig: Unterteilung nur dann, wenn die Kosten dadurch geringer werden, als den Schnitt mit allen Objekten des Knoten zu berechnen

Konstruktion „guter“ kD-Bäume und BVH

Beispiel

- wir betrachten während der Konstruktion Bounding Boxes von Objekten eines Knotens auch wenn wir kD- und BSP-Bäume erzeugen

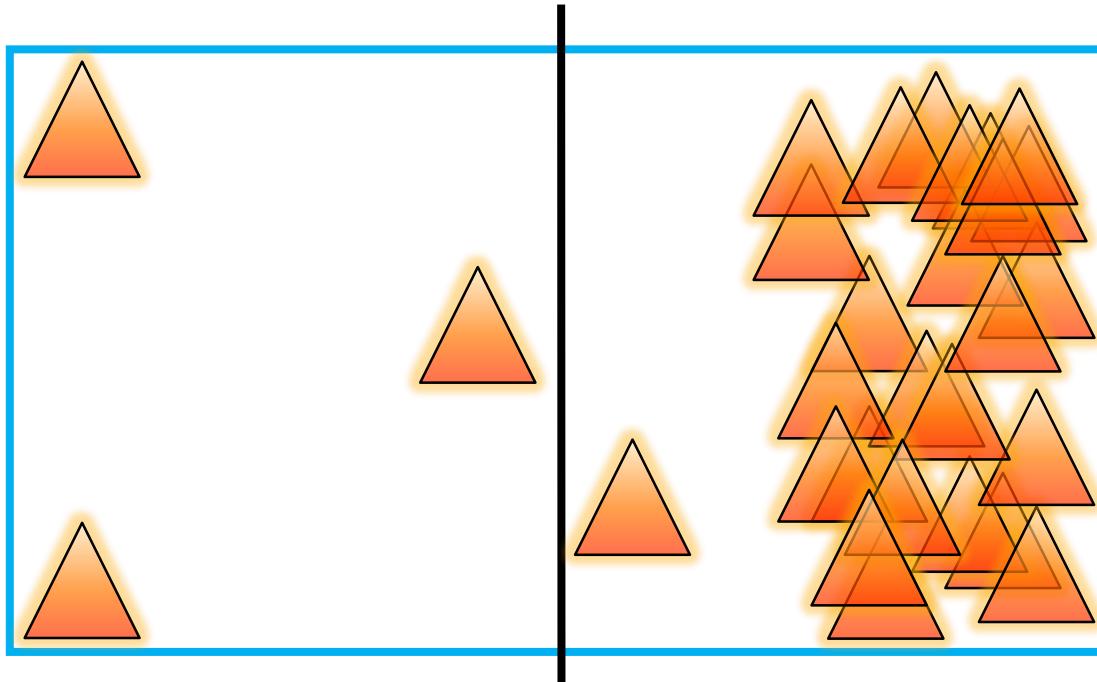


Konstruktion „guter“ kD-Bäume und BVH

Beispiel: Unterteilung in der Mitte

- Motivation: gleiche Wahrscheinlichkeiten für das Treffen des linken bzw. rechten Kindknotens
- aber: keine Berücksichtigung von $C(L)$ bzw. $C(R)$

$$C = C_T + P(\text{treffe } L)C(L) + P(\text{treffe } R)C(R)$$

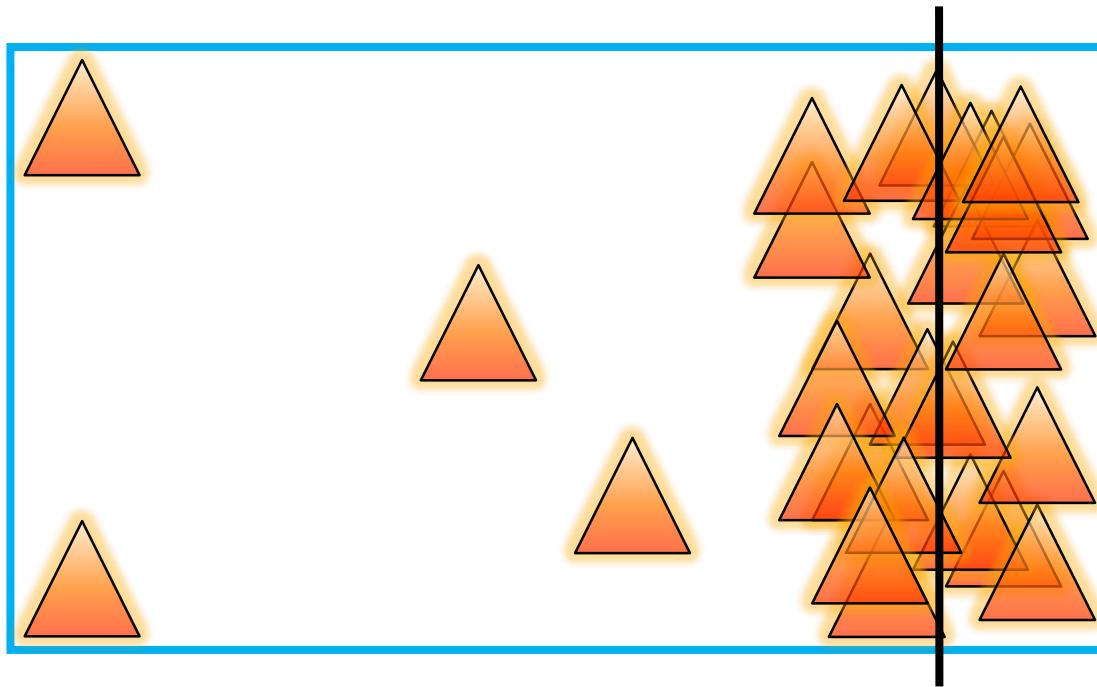


Konstruktion „guter“ kD-Bäume und BVH

Beispiel: Unterteile nach Objekt-Median

- Motivation: gleiche $C(L)$ und $C(R)$, d.h. in etwa genauso viele Dreiecke in den Kindknoten, die geschnitten werden müssen
- aber: keine Berücksichtigung der Wahrscheinlichkeiten für das Treffen des linken bzw. rechten Kindknotens

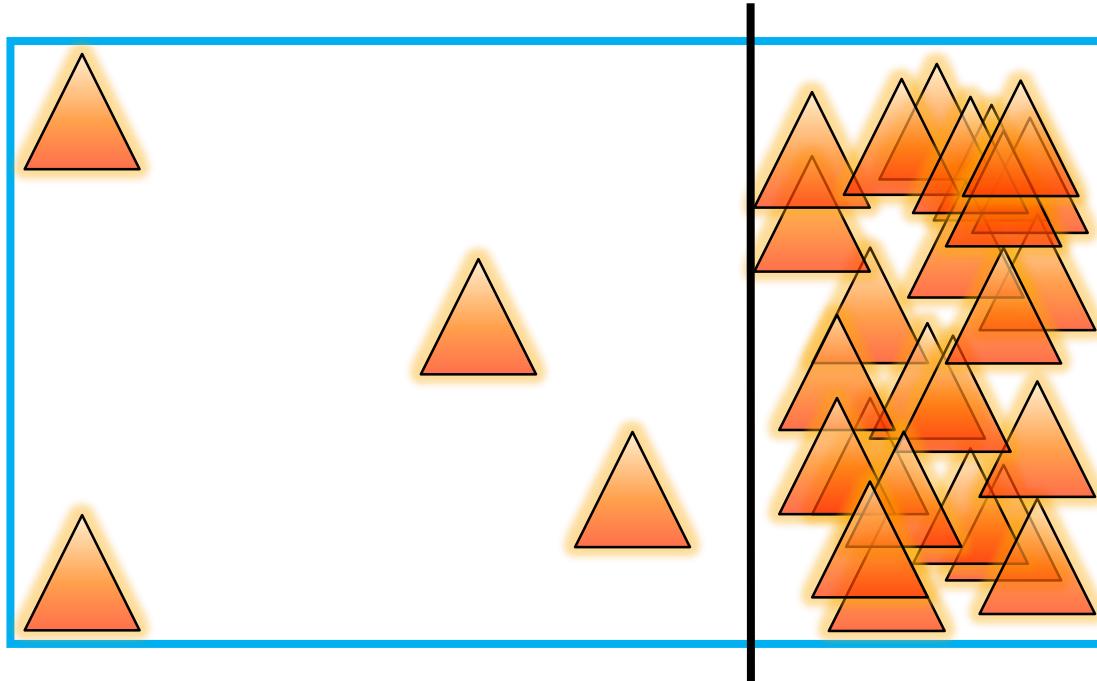
$$C = C_T + P(\text{treffe } L)C(L) + P(\text{treffe } R)C(R)$$



Konstruktion „guter“ kD-Bäume und BVH

Beispiel: kostenoptimierte Unterteilung

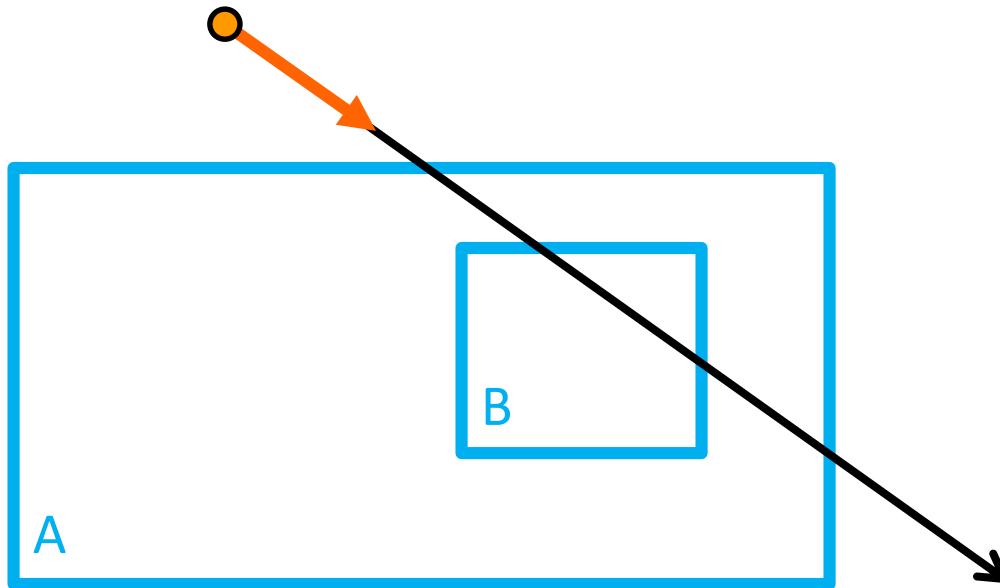
- ▶ Surface Area Heuristic (SAH) zur Minimierung der Kostenfunktion (für kD- und BSP-Bäume, sowie für BVH einsetzbar)
- ▶ führt zu
 - ▶ Isolation von komplexen Szenenteilen
 - ▶ Erzeugung großer dünnbesetzter/leerer Bereiche



Konstruktion „guter“ kD-Bäume und BVH

Eine Konsequenz aus Croftons Theorem (gilt allg. für konvexe Objekte)

- ▶ geg. eine Box B , komplett enthalten in einer Box A
- ▶ die Wahrscheinlichkeit, dass ein zufälliger Strahl von außerhalb, der A schneidet, auch B schneidet ist: $SA(B)/SA(A)$
- ▶ $SA(\cdot)$ ist die Oberfläche (surface area) einer Box



Kostenfunktion

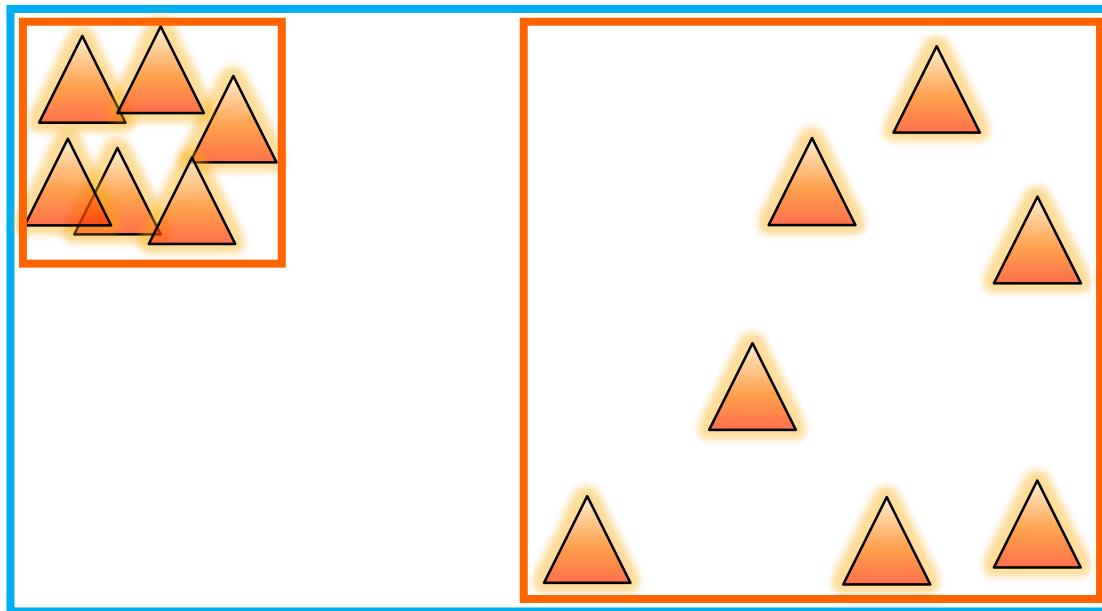
- Kostenfunktion für das Unterteilen eines kD-Baum/BVH Knotens p

$$C = C_T + \frac{SA(B_l)}{SA(B_p)} |P_l|C_i + \frac{SA(B_r)}{SA(B_p)} |P_r|C_i$$

- B_l , B_r und B_p sind die Bounding Boxes der Primitive im linken und rechten Kindknoten, bzw. des betrachteten Knotens p
- $|P_l|$ und $|P_r|$ Anzahl der Primitive im linken und rechten Kindknoten
- C_i sind die Kosten eines Strahl-Primitiv-Schnitttests
- C_T sind die Kosten der Traversierung eines kD-Baum- bzw. BVH-Knoten
 - unterteile, wenn $C < (|P_l| + |P_r|)C_i$
 - durch den konstanten Overhead C_T lohnt sich Unterteilung bei kleineren Knoten irgendwann nicht mehr
- Ziel: finde die Unterteilung, die die Kostenfunktion minimiert

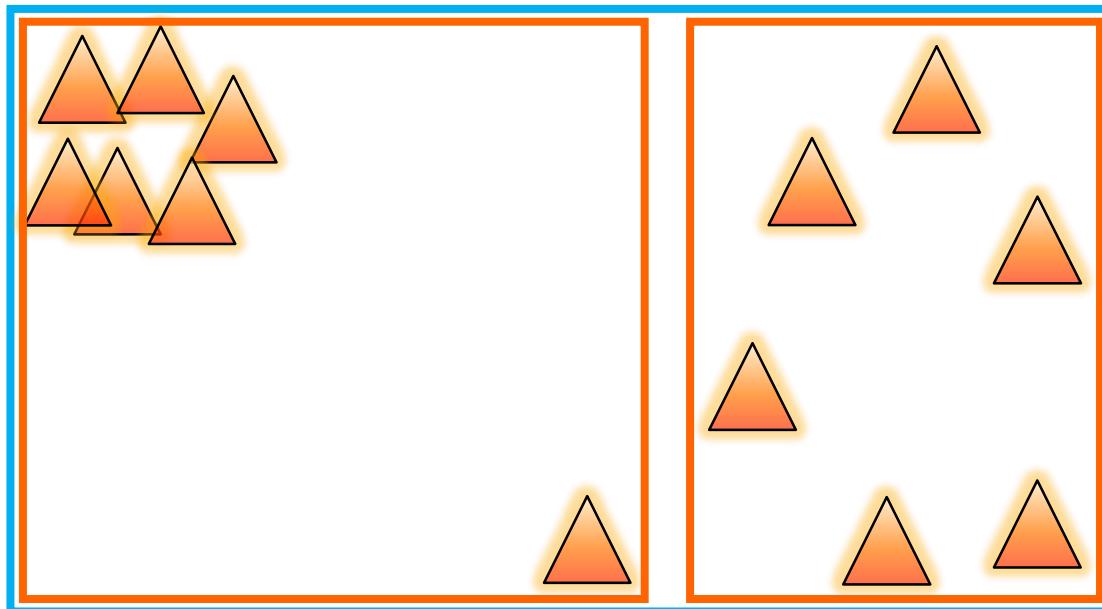
Konstruktion „guter“ kD-Bäume und BVH

Beispiel: richtige Unterteilung spielt eine Rolle



Konstruktion „guter“ kD-Bäume und BVH

Beispiel: richtige Unterteilung spielt eine Rolle



Surface Area Heuristics (SAH)

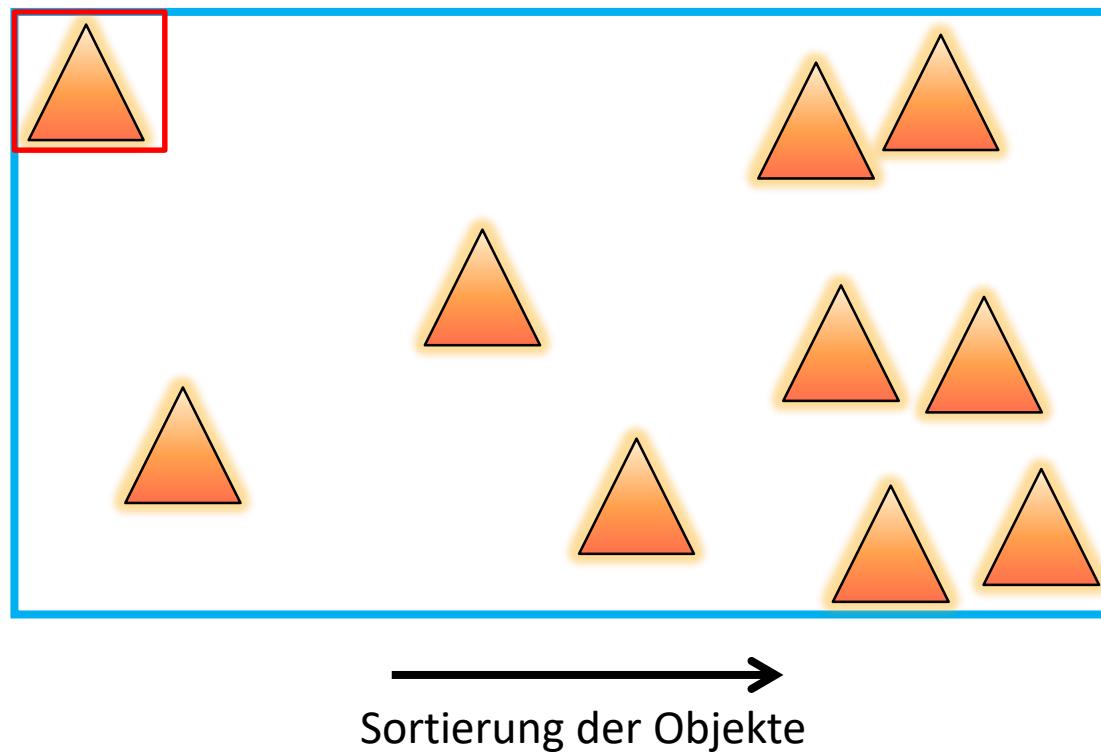
Optimierung/Bestimmung der Unterteilung

- ▶ aufwändige Suche nach der besten Unterteilung (niedrigste SAH-Kosten), da der Suchraum groß sein kann
- ▶ Variante 1: approximative Konstruktion (nicht elegant)
 - ▶ erzeuge einige Unterteilungskandidaten
(Ebenen zur Unterteilung entlang der x -, y - und z -Achse), natürlich innerhalb der Bounding Box B_p
 - ▶ berechne Kostenfunktion für jeden Kandidaten
- ▶ Variante 2: Konstruktion mit inkrementeller Berechnung
 - ▶ sortiere die n Primitive nach x -, y - und z -Achse
 - ▶ teste jede der möglichen $3(n - 1)$ Aufteilungen
 - ▶ inkrementelle Berechnung der AABBs und damit vergleichsweise effiziente Berechnung der SAH möglich
- ▶ Konstruktion mit SAH verursacht nicht vernachlässigbare Kosten!
- ▶ gut konstruierte kD-/BSP-Bäume bzw. BVHs können dafür aber um ein Vielfaches schneller sein, als schlecht konstruierte

Surface Area Heuristics (SAH)

Inkrementelle Berechnung der AABBs

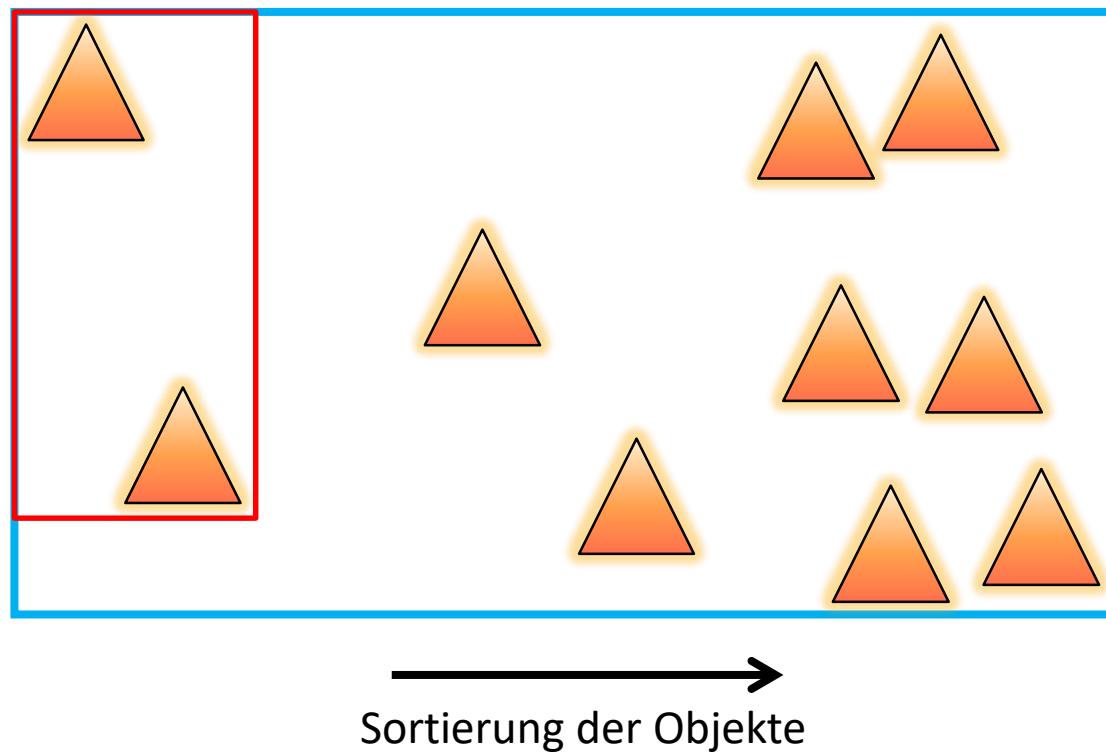
- ▶ AABB für linken Teilbaum für Primitiv 0



Surface Area Heuristics (SAH)

Inkrementelle Berechnung der AABBs

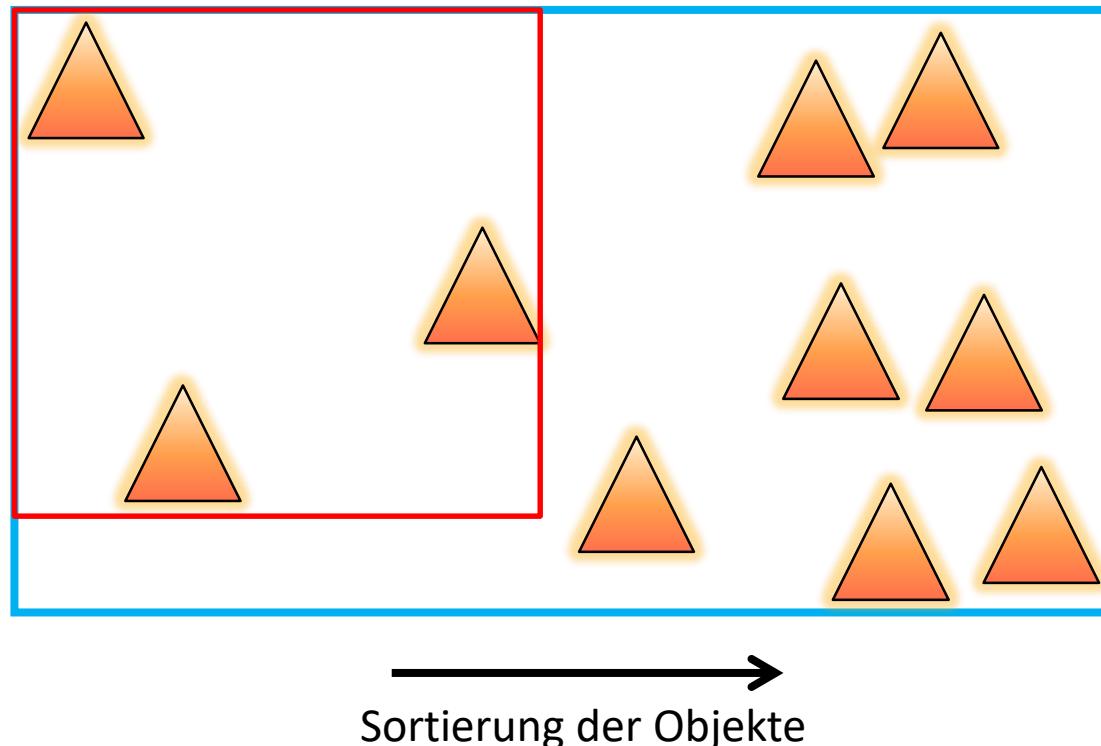
- ▶ AABB für linken Teilbaum für Primitiv 0..1



Surface Area Heuristics (SAH)

Inkrementelle Berechnung der AABBs

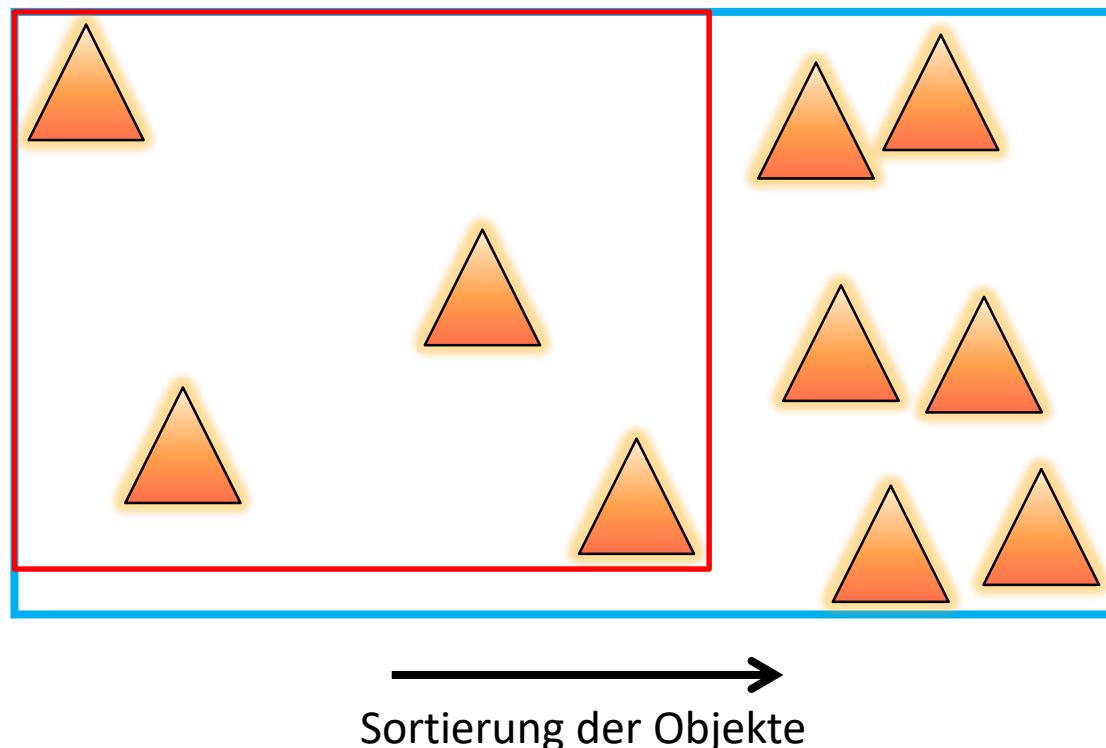
- ▶ AABB für linken Teilbaum für Primitiv 0..2



Surface Area Heuristics (SAH)

Inkrementelle Berechnung der AABBs

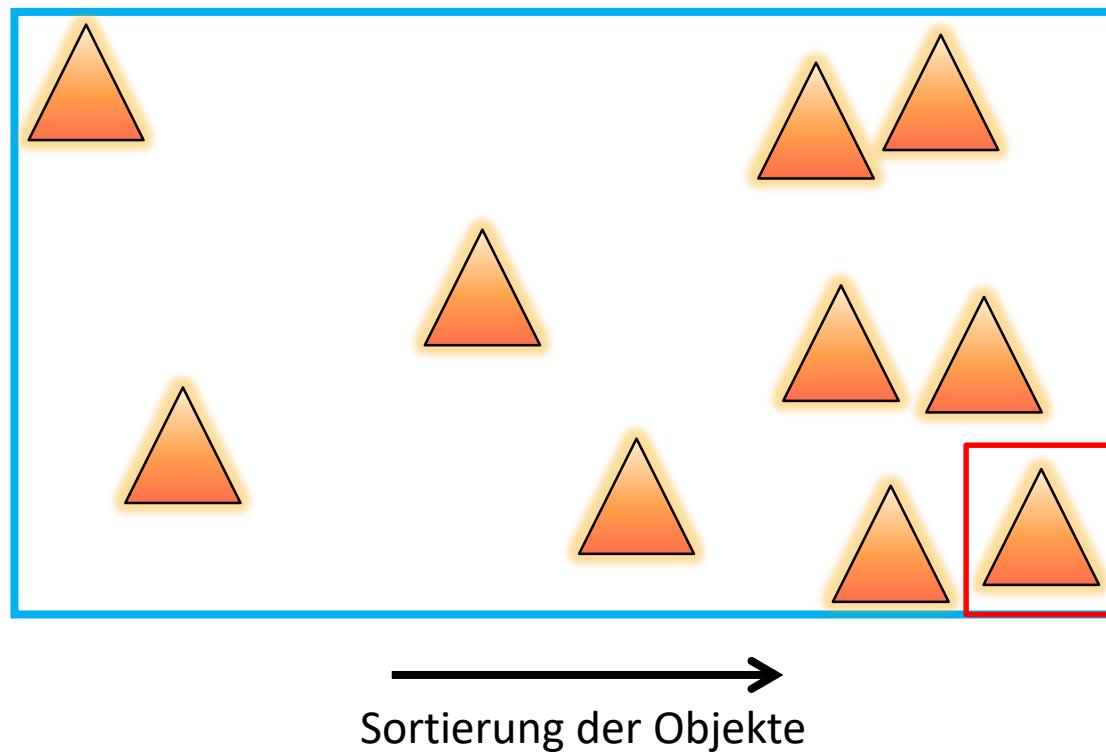
- ▶ AABB für linken Teilbaum für Primitiv 0..3
- ▶ weiter bis 0.. $n - 1$



Surface Area Heuristics (SAH)

Inkrementelle Berechnung der AABBs

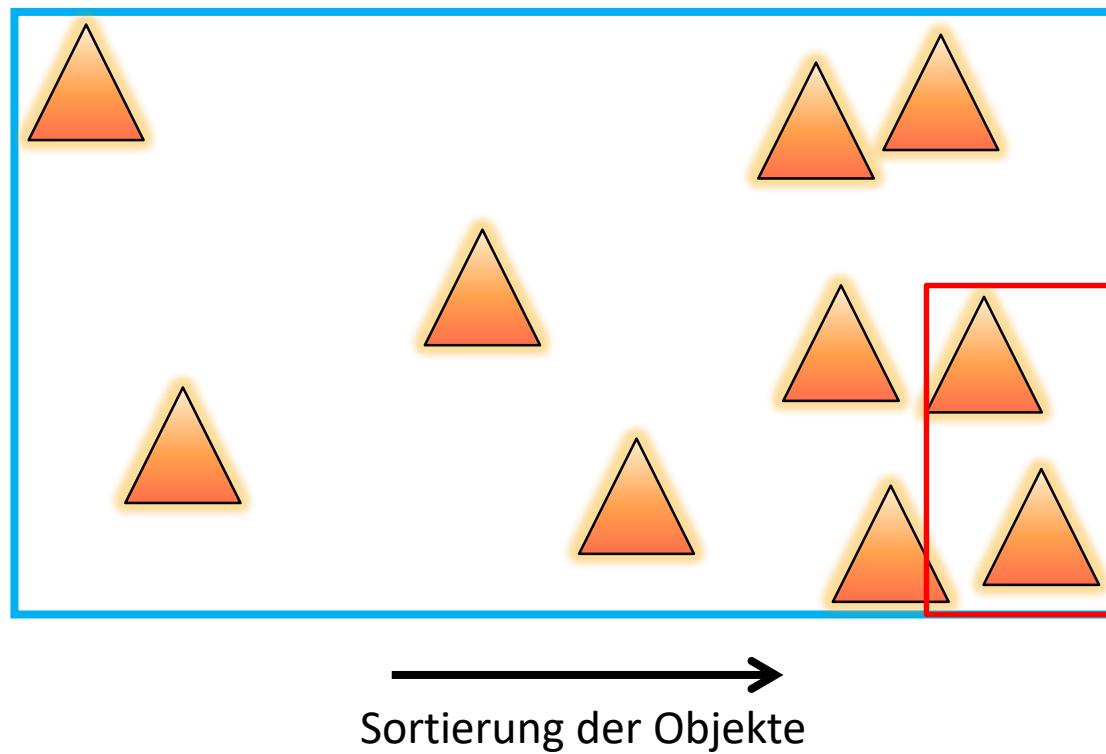
- AABB für rechten Teilbaum für Primitiv n – 1



Surface Area Heuristics (SAH)

Inkrementelle Berechnung der AABBs

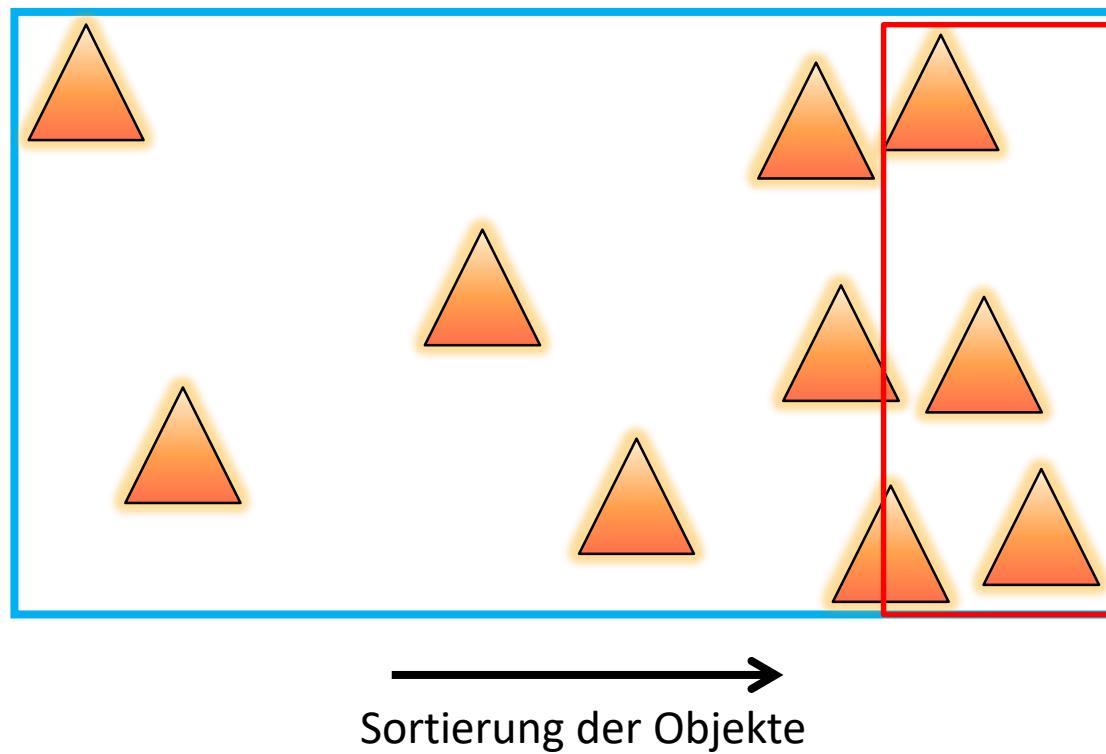
- AABB für rechten Teilbaum für Primitiv n – 1..n – 2



Surface Area Heuristics (SAH)

Inkrementelle Berechnung der AABBs

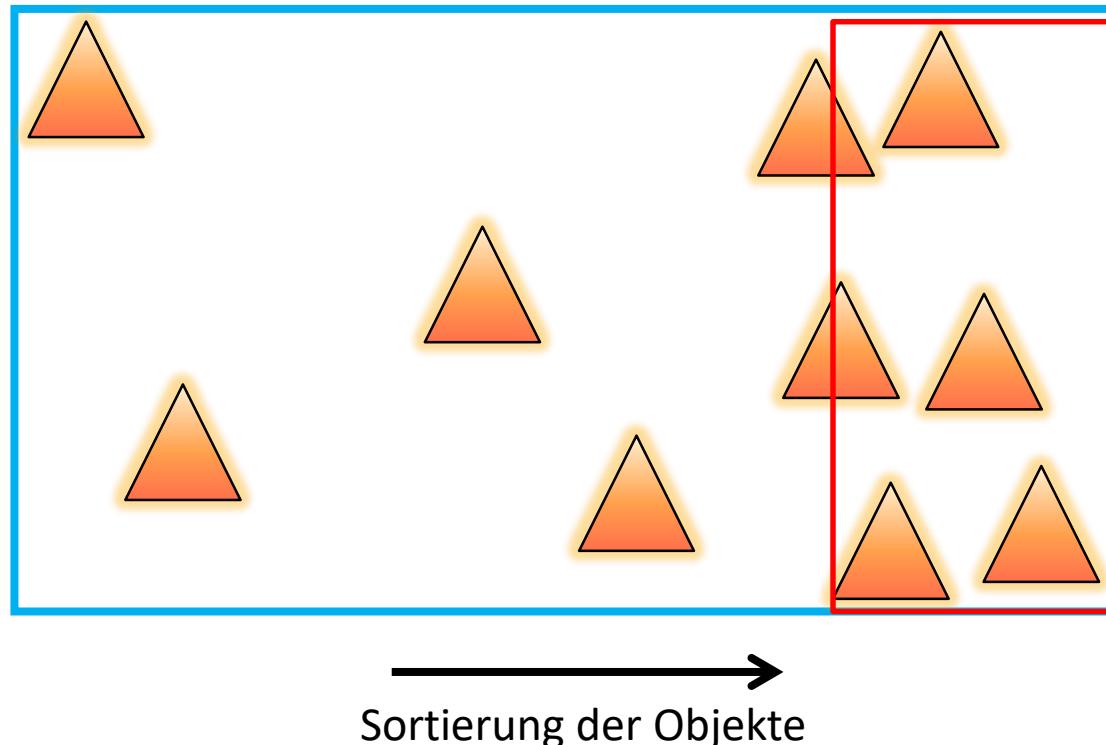
- AABB für rechten Teilbaum für Primitiv n – 1..n – 3



Surface Area Heuristics (SAH)

Inkrementelle Berechnung der AABBs

- ▶ AABB für rechten Teilbaum für Primitiv $n - 1..n - 4$
- ▶ die beiden inkrementellen Berechnungen der AABBs „von links“ und „von rechts“ erlauben eine schnelle Berechnung aller $n - 1$ SAH-Werte
 - ▶ eine Richtung (z.B. von $n - 1$ nach 1) berechnen und speichern
 - ▶ andere Richtung berechnen und direkt SAH auswerten



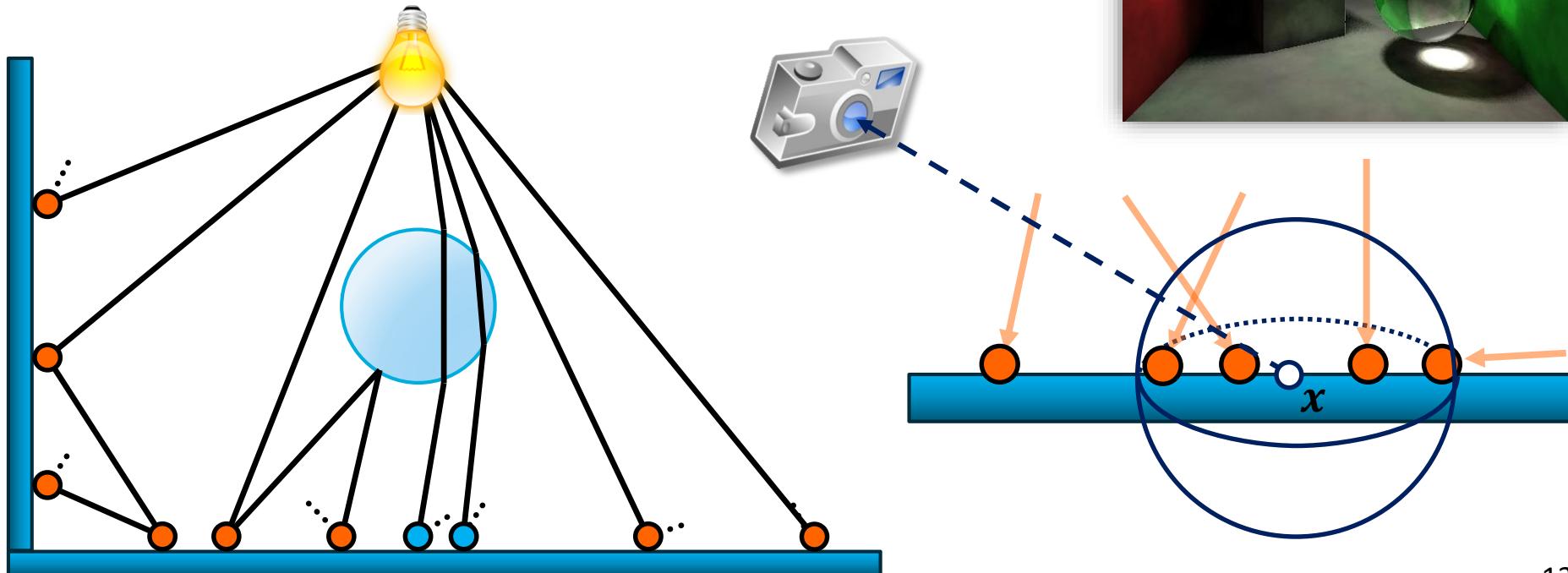
Anwendung: kD-Bäume und Photon Mapping



Exkurs: Photon Mapping

Grundidee

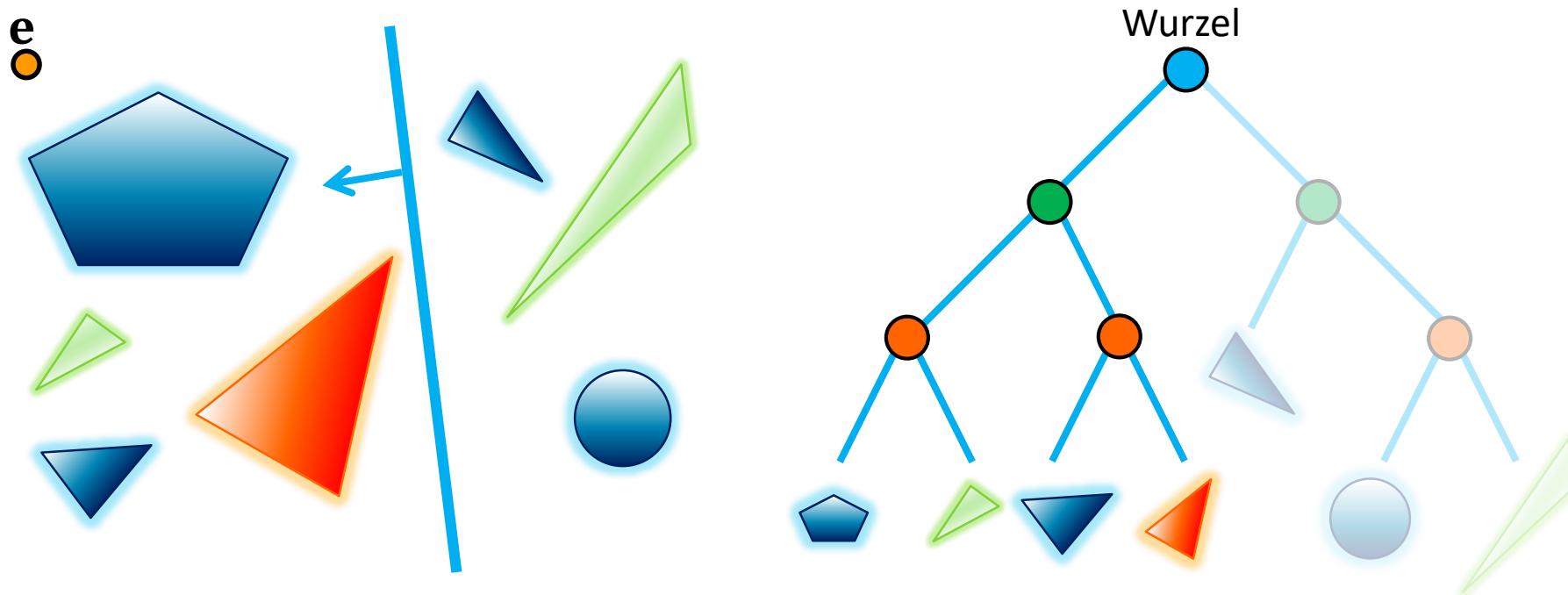
- ▶ sende Photonen (Energiepakete) von den Lichtquellen aus
- ▶ verfolge und speichere Photonen, wenn sie eine diffuse Fläche treffen
- ▶ Helligkeit \propto Photonenergie pro Fläche
 - ▶ Suche die n -nächsten Photonen oder alle Photonen in einer Umgebung mit Radius r



Suche im kD-/BSP-Baum

Suche das nahste Objekt/Primitiv zum Punkt e

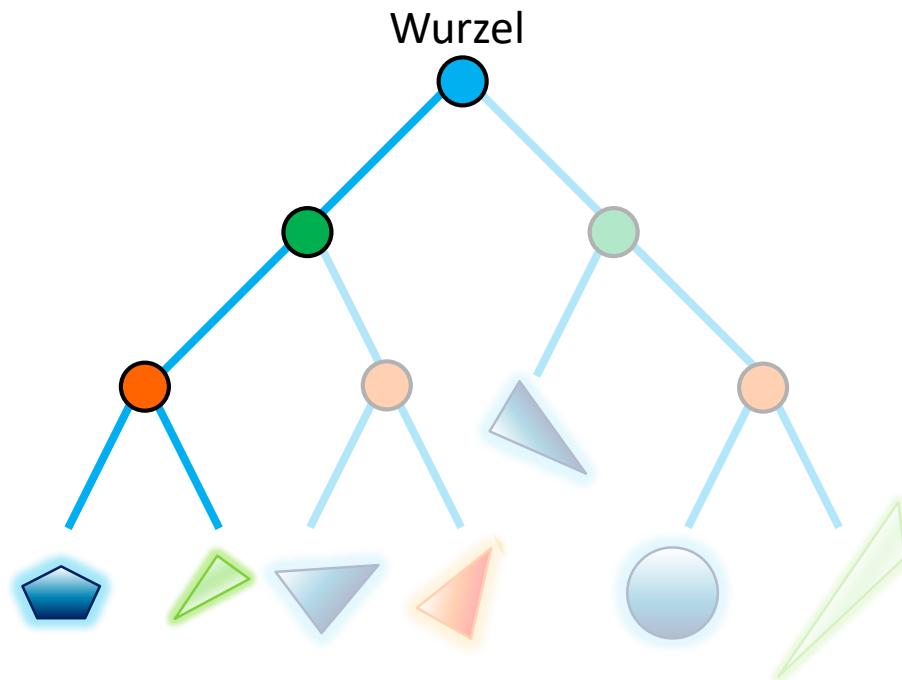
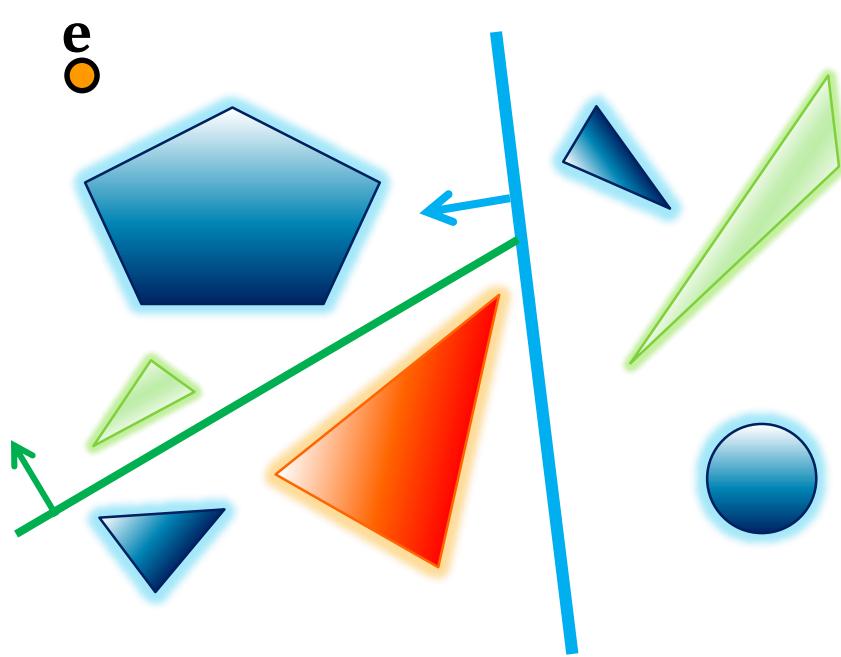
- ▶ e liegt im positiven Halbraum der Split-Ebene der Wurzel
 - ▶ d.h. alle Objekte im negativen Halbraum sind „weiter hinten“



Suche im kD-/BSP-Baum

Suche das nahste Objekt/Primitiv zum Punkt e

- ▶ e liegt im positiven Halbraum der Split-Ebene des Knotens 
- ▶ d.h. alle Objekte im negativen Halbraum sind „weiter hinten“



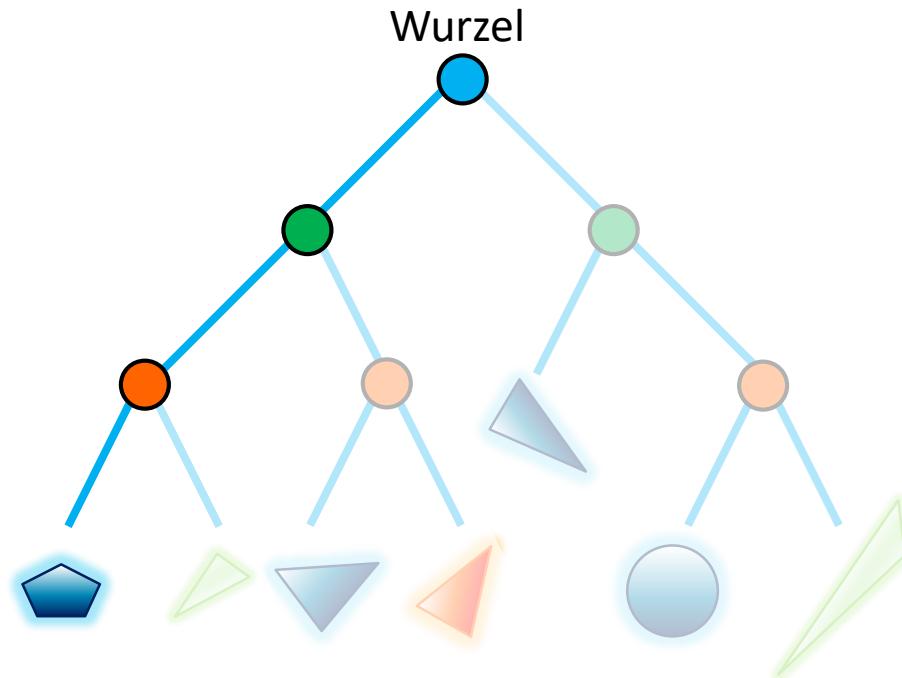
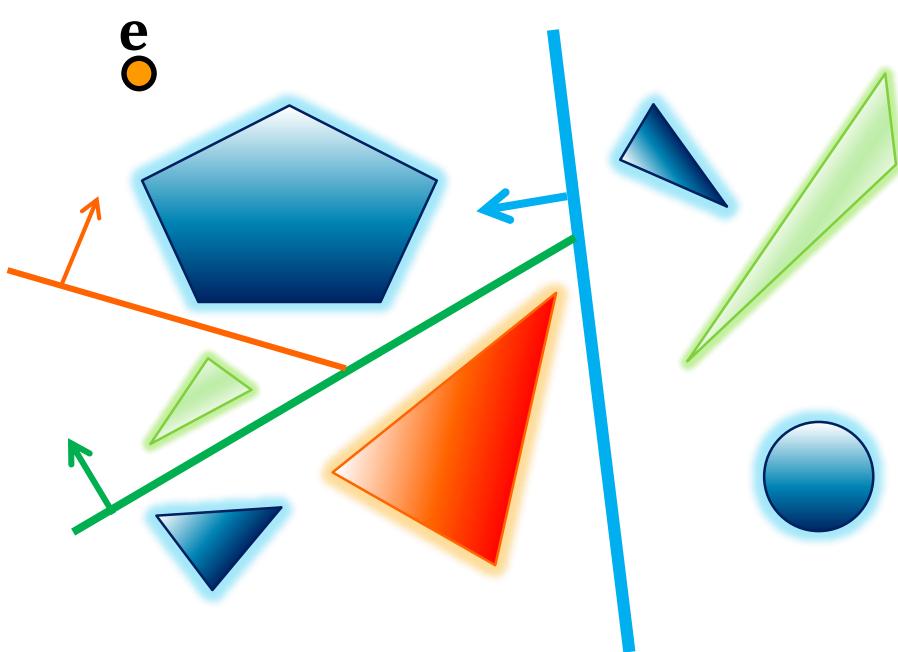
Suche im kD-/BSP-Baum

Suche das nahste Objekt/Primitiv zum Punkt e

► e liegt im positiven Halbraum der Split-Ebene des Knotens

► d.h. alle Objekte im negativen Halbraum sind „weiter hinten“

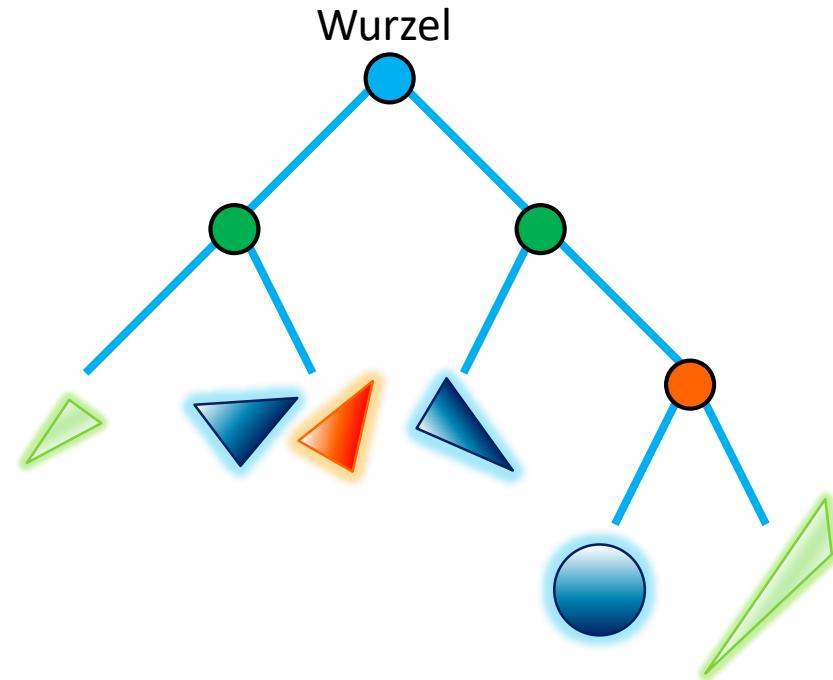
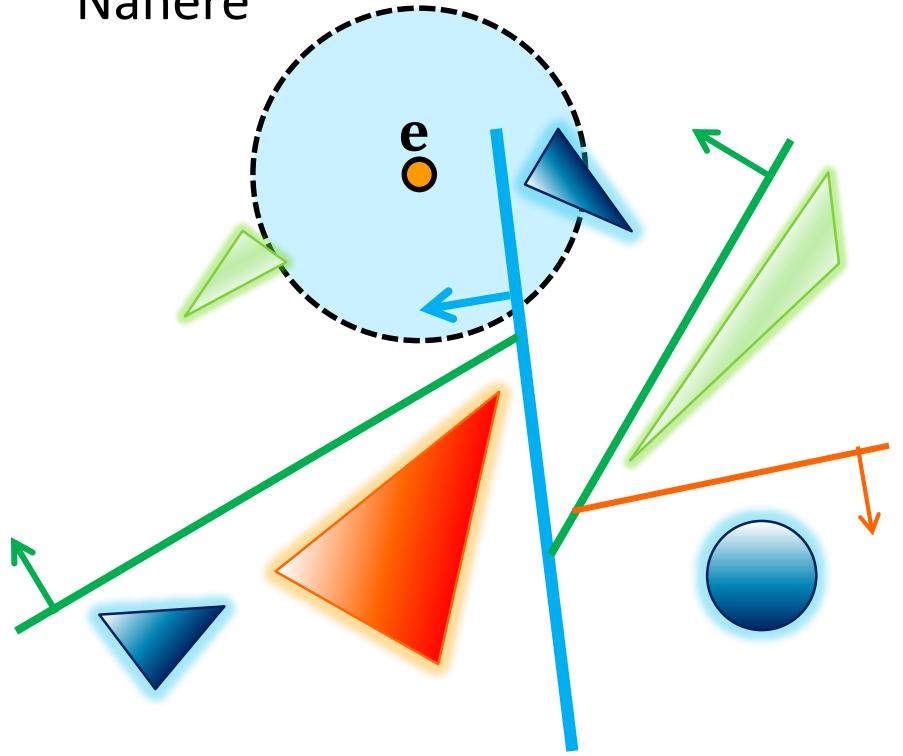
⚠ dieses Prozedere findet den Unterraum in dem sich e befindet, aber nicht immer das nahste Objekt!



Suche im kD-/BSP-Baum

Suche das nahste Objekt/Primitiv zum Punkt e

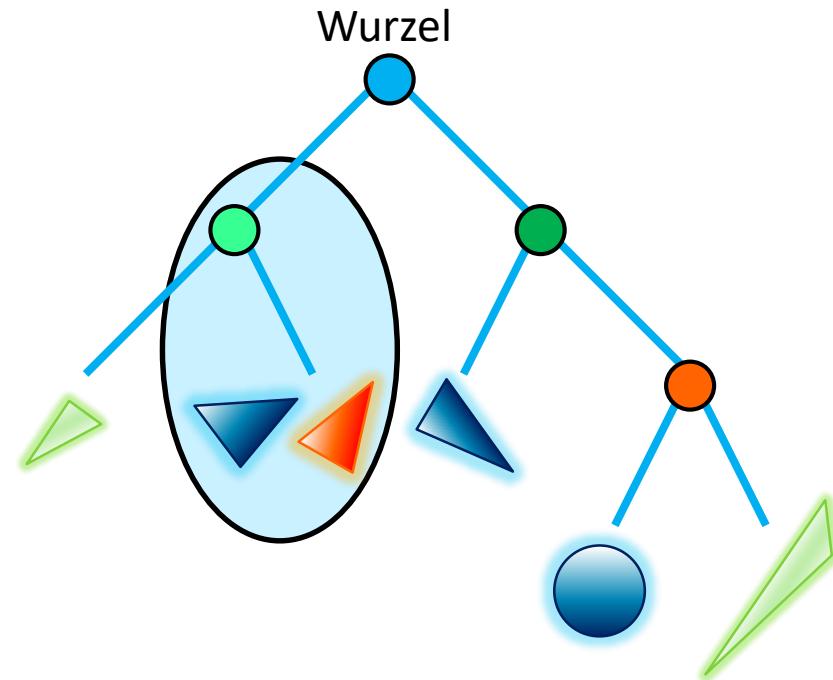
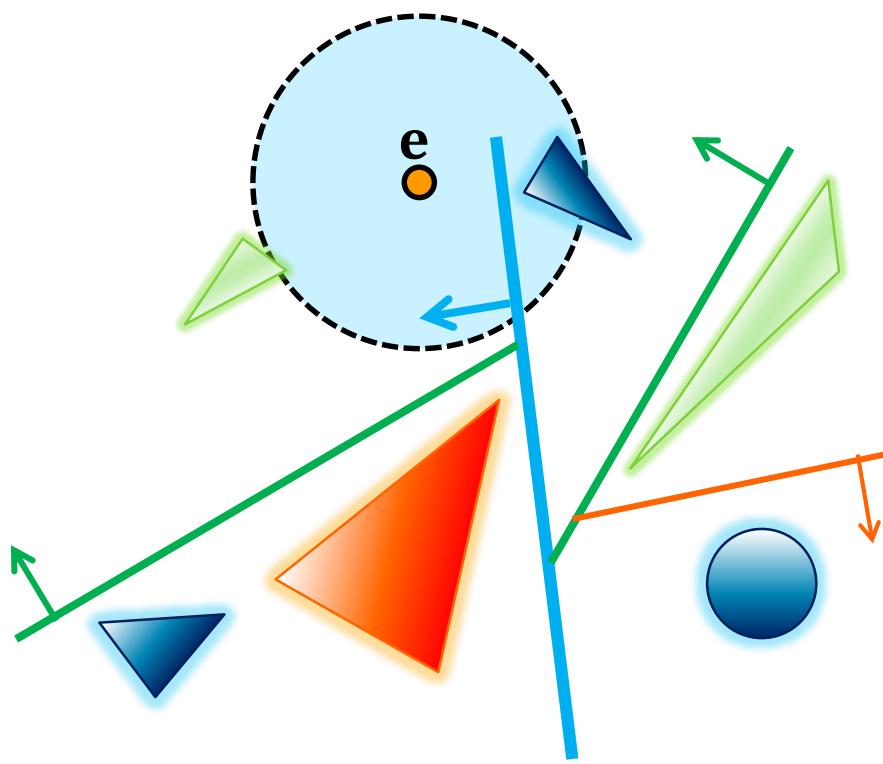
- das Objekt  im selben Unterraum wie e ist nicht das Nahste
- das gesuchte Objekt  kann in jedem Unterraum liegen, der näher an e liegt, als das bisher gefundene Primitiv
- speichere Abstand zum jeweils nahsten gefundenen Objekt und suche Nähere



Suche im kD-/BSP-Baum

Suche das nahste Objekt/Primitiv zum Punkt e

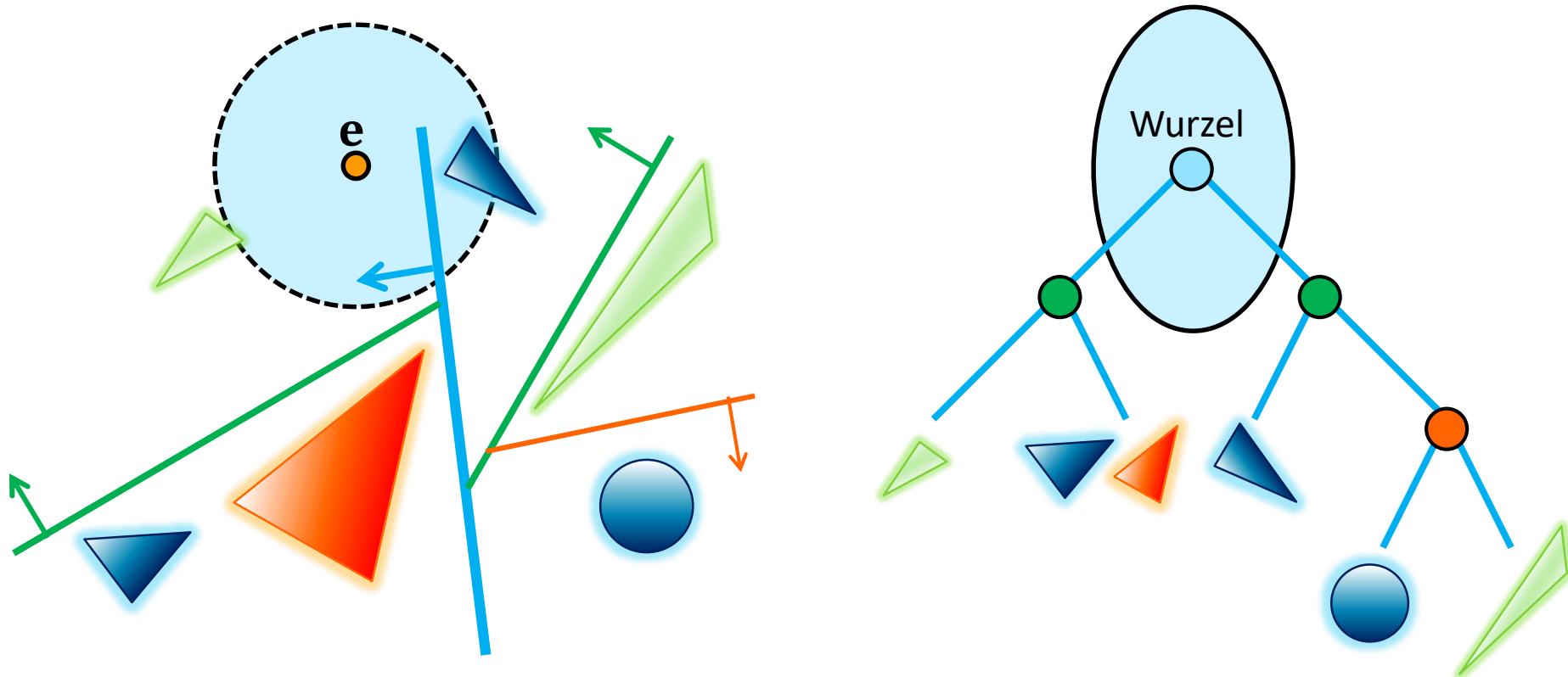
- ▶ das gesuchte Objekt kann in jedem Unterraum liegen, der näher an **e** liegt, als das bisher gefundene Primitiv
- ▶ gehe zum Elternknoten **○** und prüfe ob der andere Kindknoten in Frage kommt (hier nicht der Fall)



Suche im kD-/BSP-Baum

Suche das nahste Objekt/Primitiv zum Punkt e

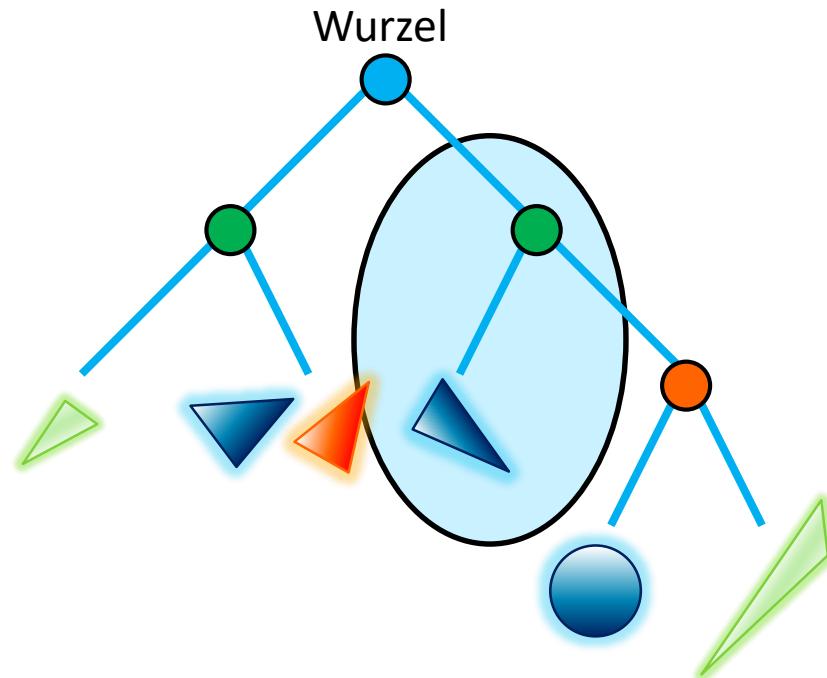
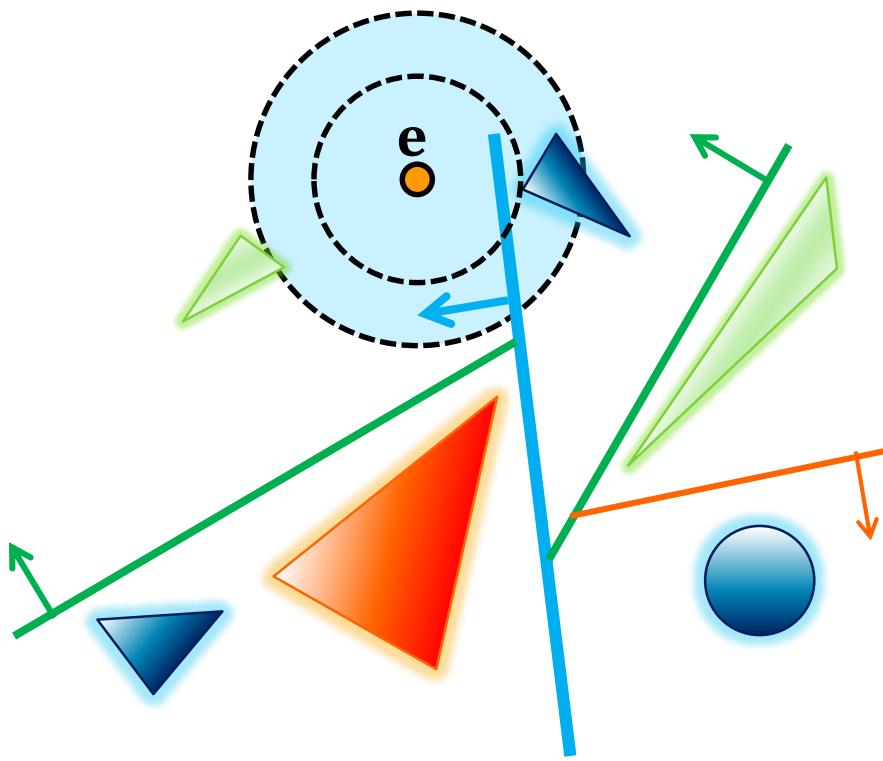
- ▶ das gesuchte Objekt kann in jedem Unterraum liegen, der näher an **e** liegt, als das bisher gefundene Primitiv
- ▶ gehe zum Eltern-Elternknoten **○** (hier schon die Wurzel) und prüfe ob der andere Kindknoten/Teilbaum in Frage kommt (hier der Fall)



Suche im kD-/BSP-Baum

Suche das nahste Objekt/Primitiv zum Punkt e

- steige in diesem Teilbaum wieder durch Tiefensuche ab (depth-first traversal): jeweils den Ast des näheren Teilbaums zuerst
- im Mittel $O(\log n)$, worst case $O(n)$ (n Objekte/Primitive)

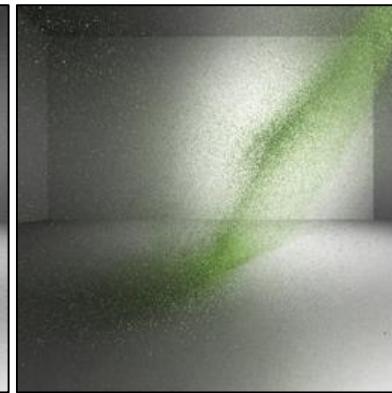


- ▶ SIMD Architekturen und kleine Strahlenpakete (typ. 4-32 Strahlen)
 - ▶ kohärente Strahlenpakete: Gruppen von Strahlen mit ähnlichem Ursprung und Richtung (z.B. Primärstrahlen, Schattenstrahlen etc.)
 - ▶ können oft sehr schnell verarbeitet werden:
oft ähnliche Wege durch die (hierarchische) Beschleunigungsstruktur
 - ▶ Bsp. der Schnitttest Strahl-AABB kann mit SIMD Befehlen einfach auf einen Test 4-Strahlen-AABB erweitert werden
 - ▶ inkohärente Pakete können Parallelität der HW nicht (gut) nutzen
(es gibt sogar Ansätze die Strahlen während des Raytracing sortieren)
- ▶ Geschwindigkeit

siehe auch <http://embree.github.io/> und <https://code.google.com/p/understanding-the-efficiency-of-ray-traversal-on-gpus/>

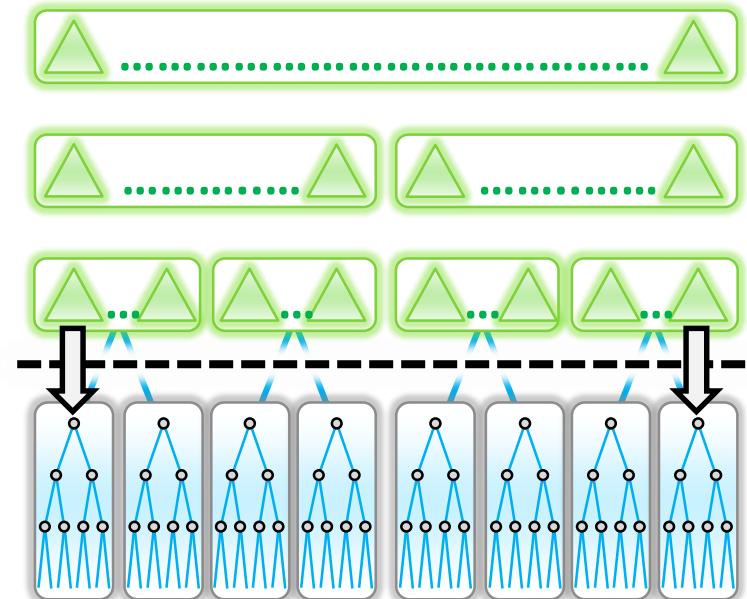
 - ▶ Größenordnung ca. 250 Mio. Strahlen pro Sekunde mit Xeon Platinum (2× 28 Cores, 2.5GHz), Grafikhardware 5-10 Mill. kohärente, ca. 50%-70% inkohärente Strahlen (RTX-GPU Architektur), jeweils ohne Shading
 - ▶ bei komplexen Szenen (Millionen Dreiecke) mit hierarchischen Datenstrukturen (typ. BVH oder kD-Bäume)
 - ▶ paralleler Aufbau der Datenstrukturen ist aktuelles Forschungsthema!

Neuberechnung und inkrementelle Updates in dynamischen Szenen?



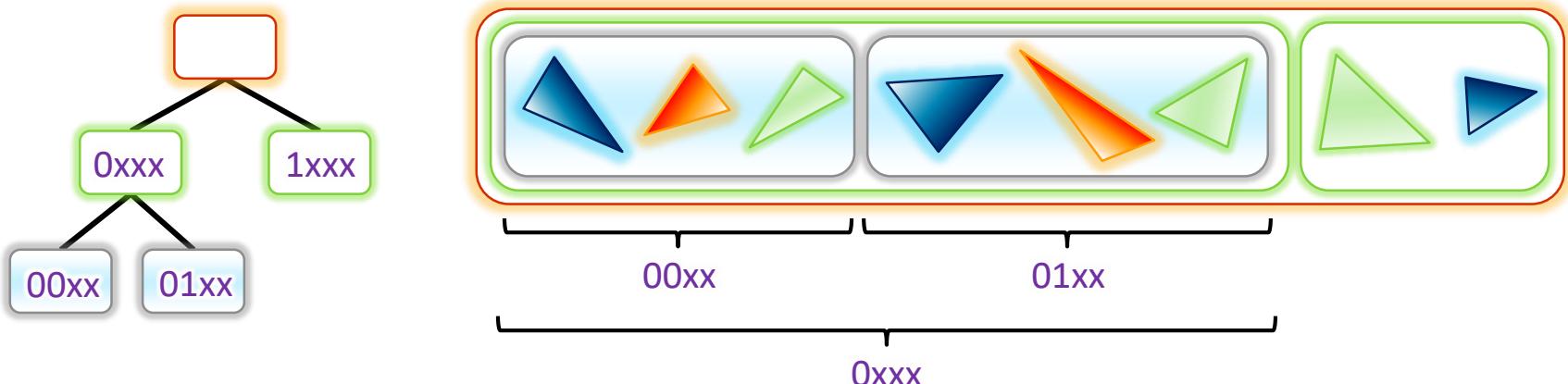
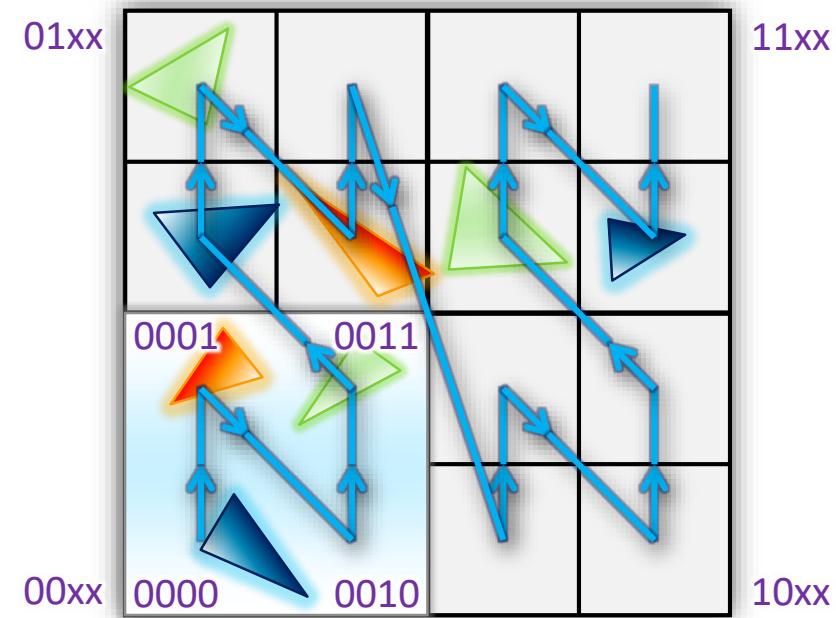
Bilder:
Pantaleoni und Luebke

- ▶ paralleler Aufbau in Millisekunden für Millionen Dreiecke auf CPU und GPU
 - ▶ mehrere hundert Mio. Δ pro Sekunde mit und ohne SAH auf CPUs
 - ▶ Aufbaugeschwindigkeit und Qualität sind widersprüchliche Kriterien
 - ▶ ein weiteres Problem ist der temporäre Speicherbedarf bei der Konstruktion
- ▶ inkrementelle Updates in großen Szenen meist mit „two-level BVHs“



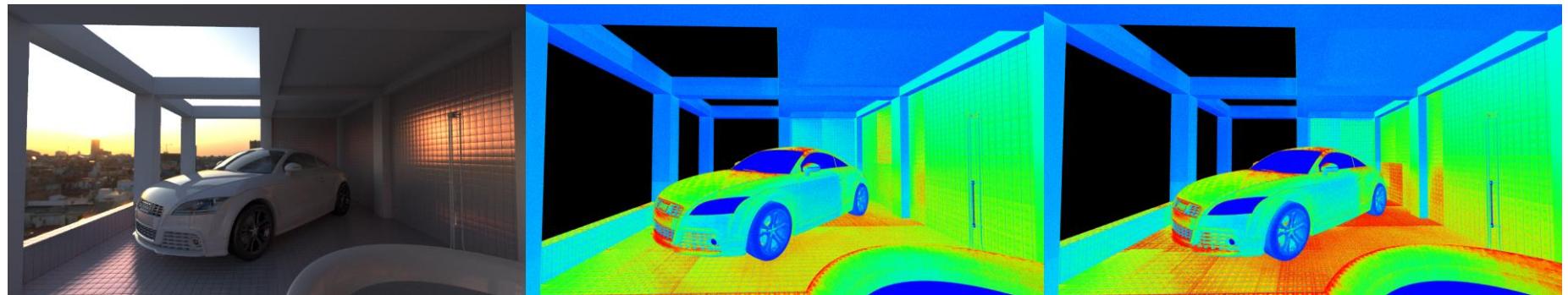
Grundidee: Vorsortierung durch Quantisierung und Morton Codes

- ▶ quantisiere Primitivschwerpunkte
- ▶ Zuweisung Morton Codes zu Zellen
- ▶ sortiere Primitive nach Zellcodes
- ▶ ergibt implizite Baumstruktur



Varianten und aktuelle Forschungsthemen

- ▶ Multi Bounding Volume Hierarchy (mehr als 2 Kindknoten, z.B. 4 AABBS)
- ▶ SATO: Surface Area Traversal Order for Shadow Raytracing, Nah et al. und MBVH Child Node Sorting for Fast Occlusion Test, Ogaki et al.
 - ▶ bei Schattenstrahlen sind wir nur daran interessiert, ob es einen Schnittpunkt gibt – die Reihenfolge kann nach Verdeckungswahrscheinlichkeit priorisiert werden



- ▶ Treelets: schneller paralleler Aufbau durch lokale „Umstrukturierung“
- ▶ siehe z.B. Sessions zu Raytracing z.B. auf der EGSR und HPG

<http://egsr2015.gcc.tu-darmstadt.de>

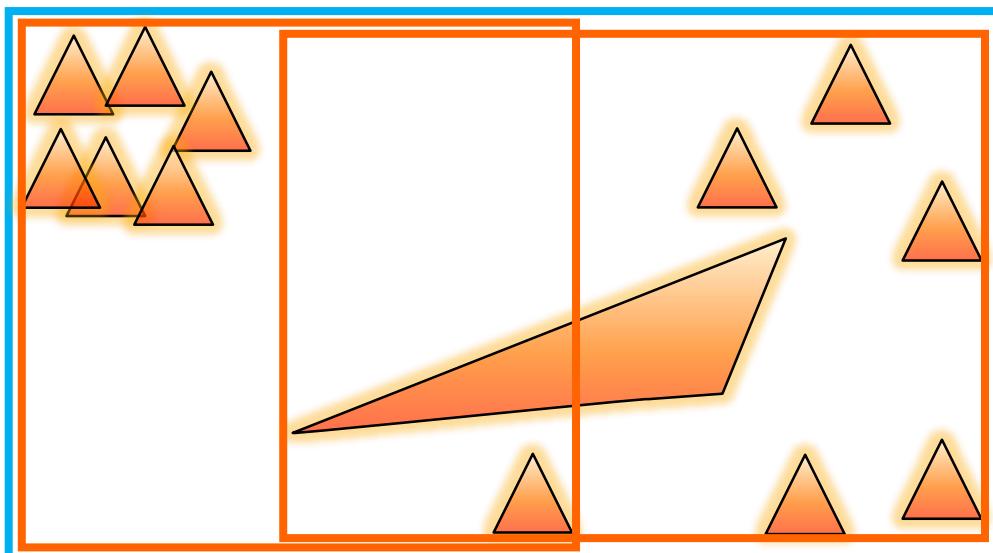
<http://egsr2017.aalto.fi>

<https://egsr2016.scss.tcd.ie>

<https://www.highperformancegraphics.org/2020/program/>

Fazit (Achtung: Erfahrungswerte!)

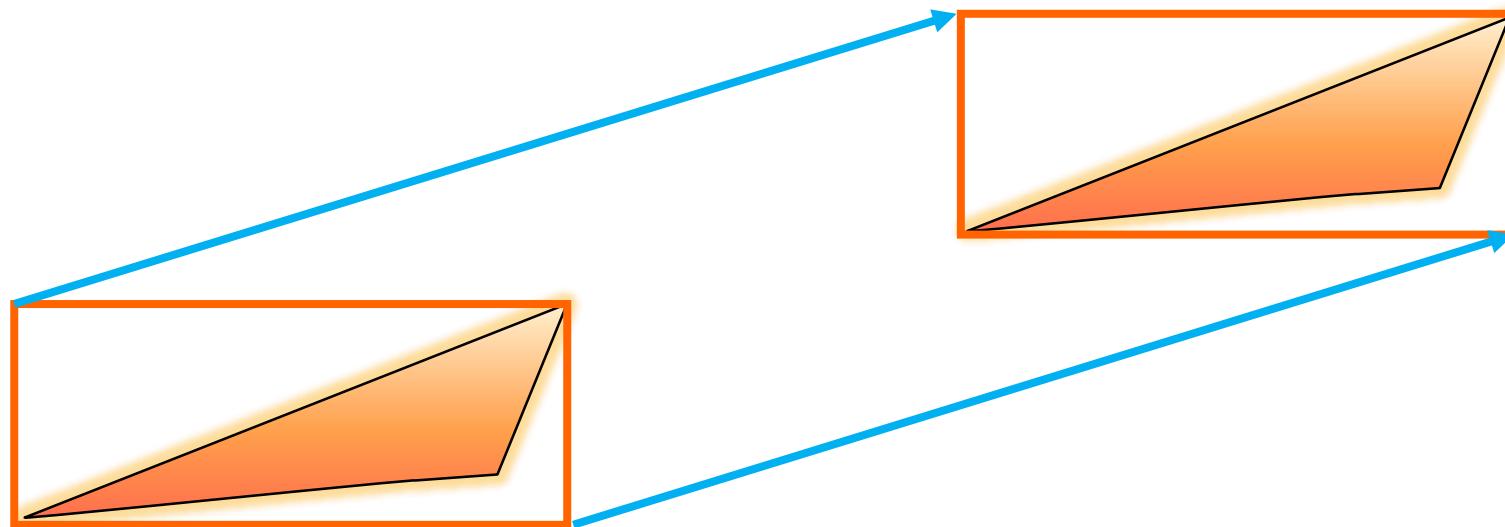
- ▶ Kombination verschiedener Beschleunigungstechniken
 - ▶ oft SIMD-Optimierung zusammen mit BVH oder kD-Baum
 - ▶ BVH mit AABBs: schnelle Erzeugung, gute Hierarchie (tendenziell etwas Probleme mit großen Dreiecken)
 - ▶ kD-Baum: aufwändiger zu konstruieren, manchmal einen Tick besser als BVH, kombiniert mit AABBs pro Knoten
- ▶ Beispiel: große Primitive problematischer bei BVH als kD-Baum (letzterer unterteilt den Raum, Primitiv ist einfach in beiden Teilräumen)



Fazit (Achtung: Erfahrungswerte!)

Beispiel: Bewegungsunschärfe und dynamische Szenen mit BVHs

- ▶ *linear motion*: AABB linear interpolieren
- ▶ schnelle Anpassung an kleine Änderungen:
 - ▶ refitting: nur Aktualisieren der BV $O(n)$
 - ▶ rebuild: $O(n \log n)$ bzw. $O(n \log^2 n)$
- ▶ kD-Baum: bewegte Split-Ebene? schwierig



Fazit (Achtung: Erfahrungswerte!)

- Kombination verschiedener Beschleunigungstechniken
 - ▶ oft SIMD-Optimierung zusammen mit BVH oder kD-Baum
 - ▶ BVH mit AABBs: schnelle Erzeugung, gute Hierarchie (tendenziell etwas Probleme mit großen Dreiecken)
 - ▶ kD-Baum: aufwändiger zu konstruieren, manchmal einen Tick besser als BVH, kombiniert mit AABBs pro Knoten
 - ▶ BSP-Baum: aufwändig zu konstruieren, oft nur ein bisschen besser als ein gut konstruierter kD-Baum
 - ▶ Gitter: eher selten, aber Aufbau auf (Grafik-)Hardware einfacher (als Suchstrukturen oft in Simulation, z.B. SPH)
 - ▶ hierarchische Gitter und Octrees: eher eingesetzt für Simulationen
- weitere Verwendung (nicht nur) in der Computergrafik
 - ▶ Bounding Volumes und BVH: Culling in der Echtzeit-Grafik
 - ▶ kD-/BSP-Bäume: Sortierung, Suche, ...
 - ▶ Octrees: Speicherung von räumlichen Daten, z.B. 3D-Texturen
 - ▶ Kollisionserkennung (Physiksimulation, Robotik)