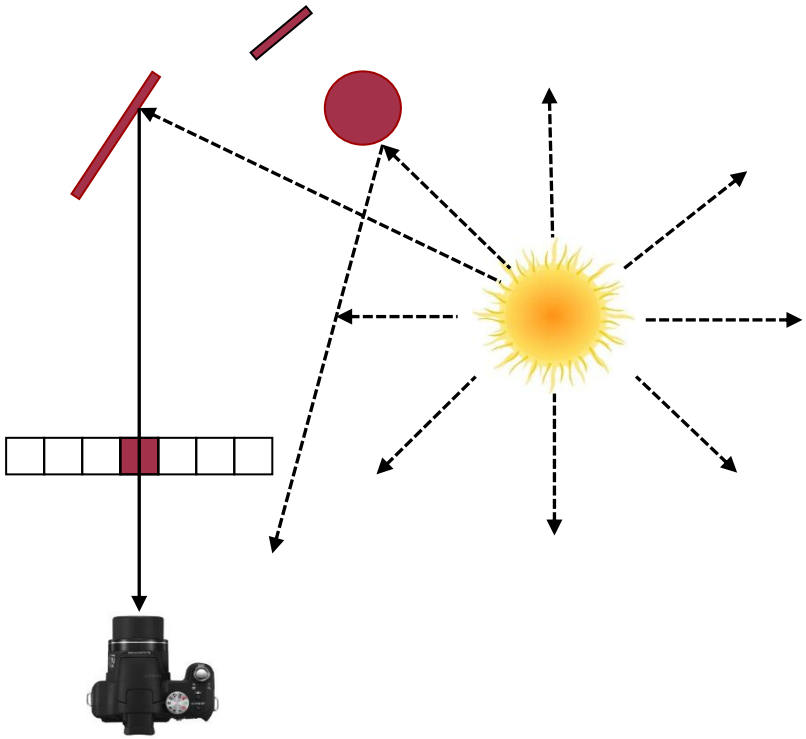
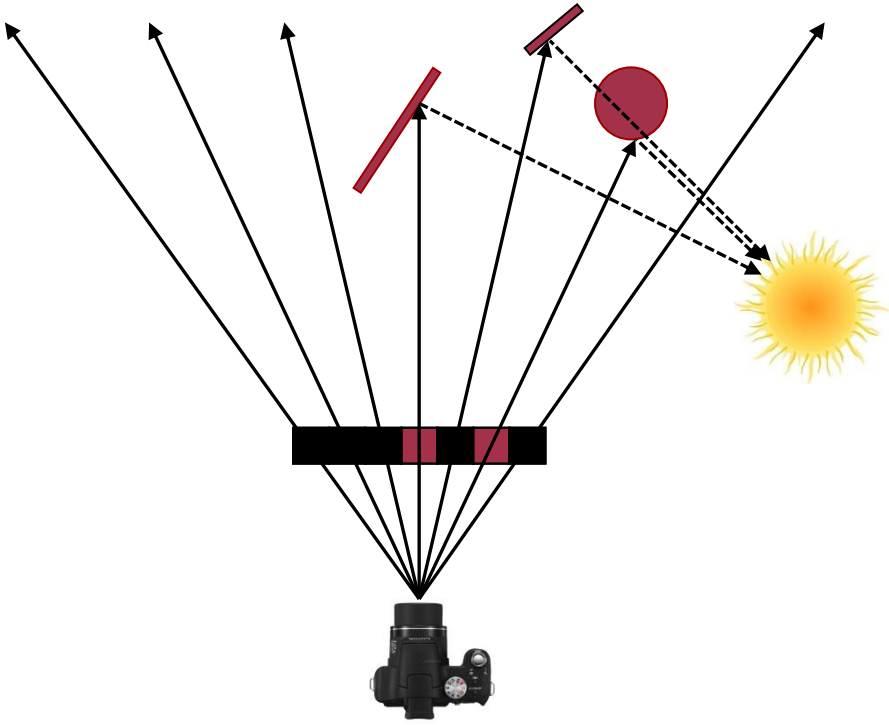
Computergrafik

Universität Osnabrück, Henning Wenke, 2012-07-03

Noch Kapitel XVI





Realtime Ray Tracing

Wiederholung: 2D implizites Rendern

```
kernel void implicit2DRenderer(
global Circle*       circles,  // Siehe Folie 28, gestern
const  int           circleCnt,
write_only image2d_t rederedImage) {
  int2 pixelPos = (int2)(get_global_id(0), get_global_id(1));

  float minDepth = 90000; // > maximale Szenentiefe
  float4 color = (float4) (0.0, 0.0, 0.0, 0.0); // Hintergrundfarbe
  for(int i = 0; i < circleCnt; ++i) {
    Circle c = circles[i];
    if(overlaps_N_Nearer(c, pixelPos, minDepth)) { // Folie 29, gestern
      // Kreis setzt sich durch. Speichere Werte lokal
      minDepth = c.depth;
      color = c.color;
    }
  }
  // Schreibe Farbe des nächsten Kreises. Keiner? Hintergrundfarbe
  write_imagef(rederedImage, pixelPos, color);
}
```

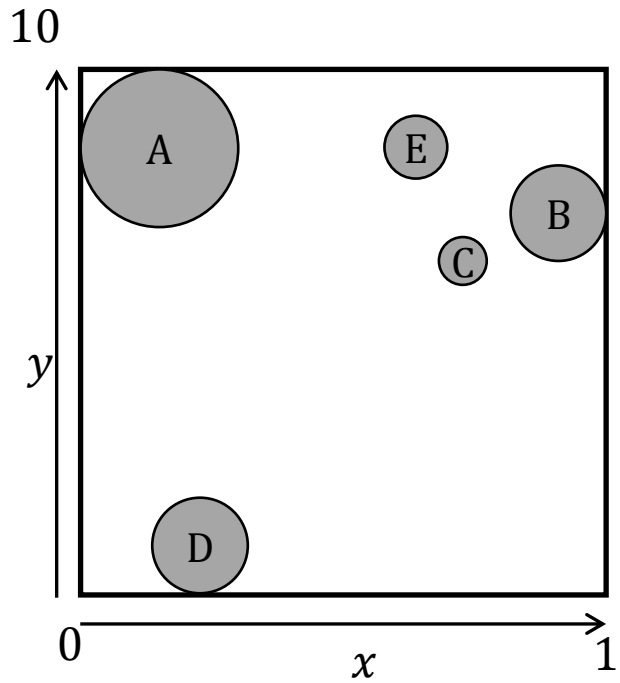
16.2

Implizites 2D Rendern mit KD-Tree

Prinzip KD-Tree I

- K-Dimensionale Baumstruktur, hier:2D
- Für K-dimensionale Suchanfragen geeignet
- Teilt Raum Nodes, die je einen rechteckigen Ausschnitt des Raums repräsentieren und alle zugehörigen Objekte enthalten
- Finde maximale Ausdehnung der Szene. Dies ist Bounding Box der Root Node (Node 0), welche alle Objekte der Szene enthält

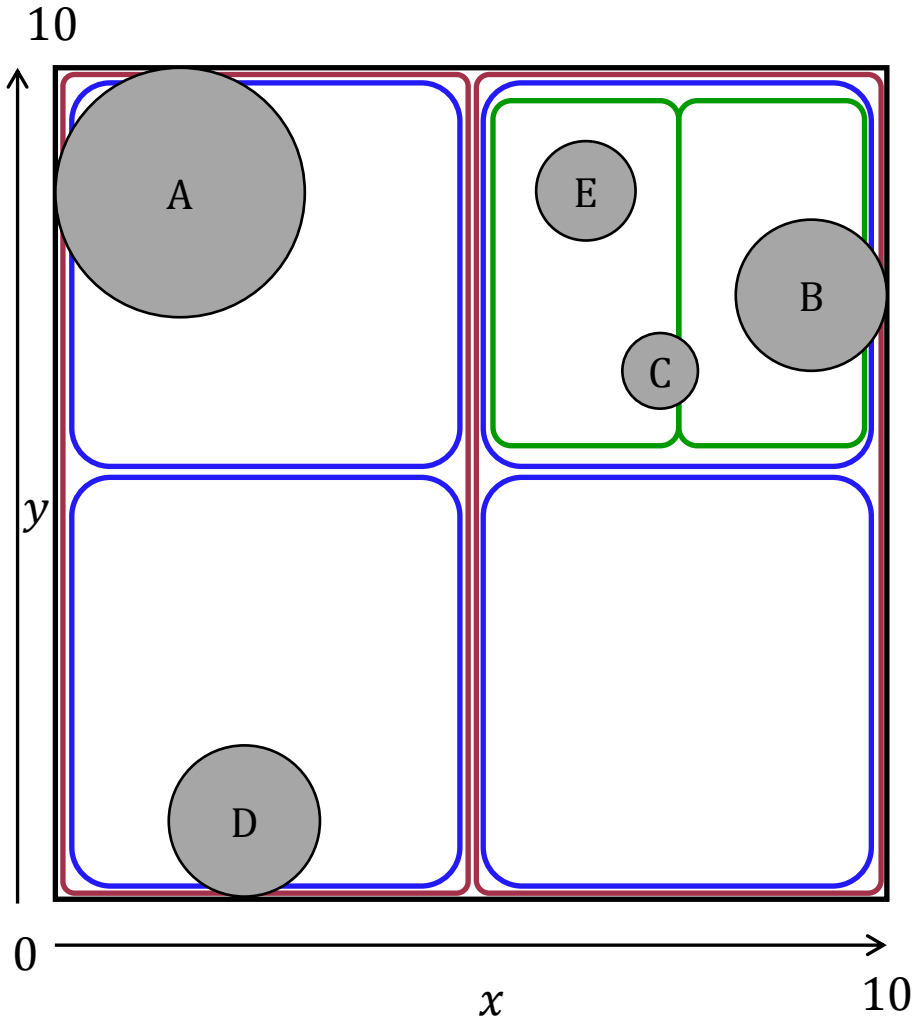
Node 0
A,B,C,D,E
x [0, 10]
y [0, 10]



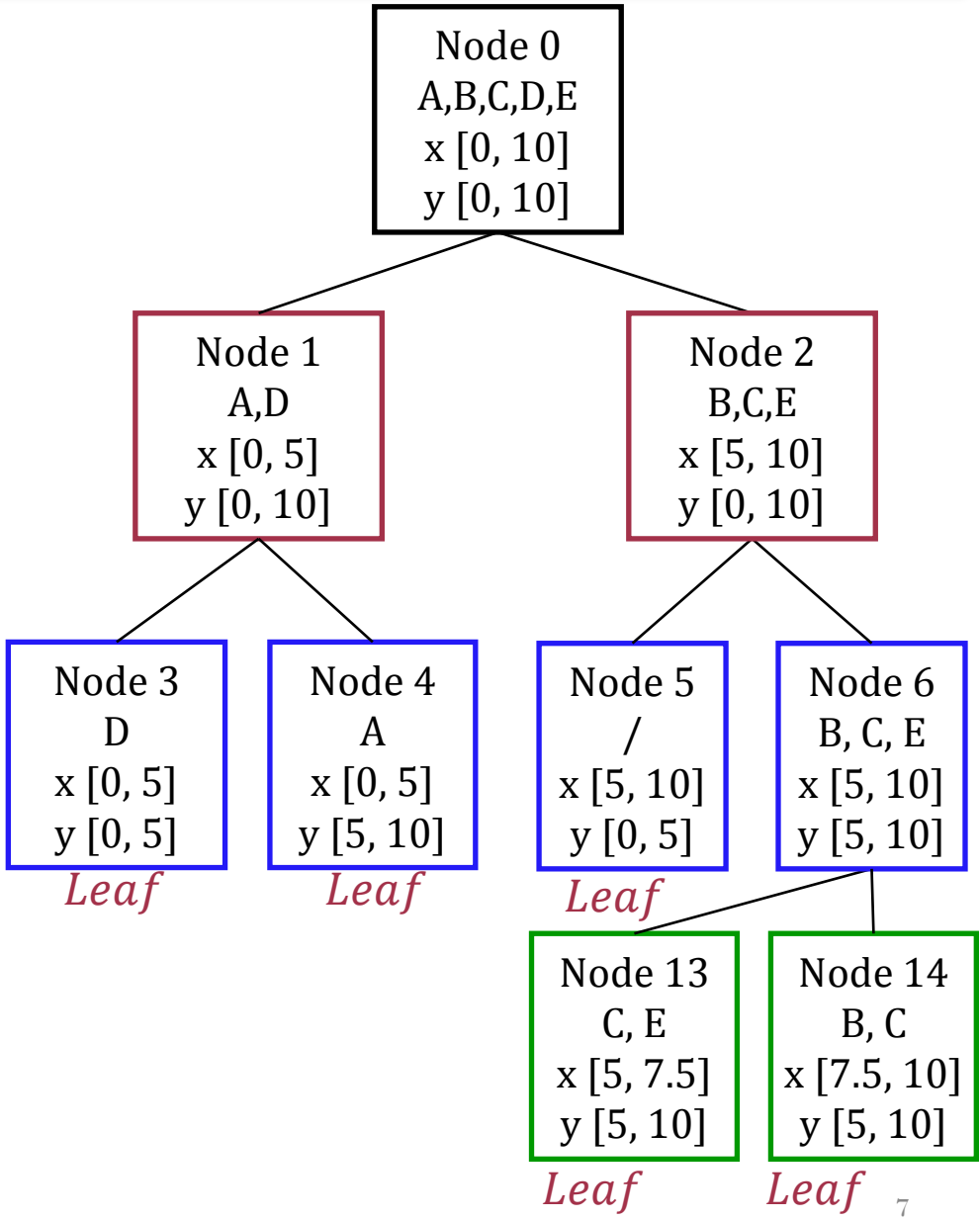
Prinzip KD-Tree II

- Definiere Abbruchbedingungen:
  - Maximale Rekursionstiefe erreicht
  - Objektzahl ≤ ideale Objektzahl pro Node
- Beginne mit Root Node
- Wenn Abbruchbedingungen nicht erfüllt:
  - Teile Raum in Dimension mit längster Ausdehnung der Node mittig
  - Alternative: Spatial Area Heuristic
  - Erzeuge linke Child Node und rechte Child Node, welche die beiden Raumhälften repräsentieren
  - Teile Objekte auf Child Nodes gemäß Überlappung auf
  - Werte Abbruchbedingung für Child Nodes aus
- Nodes ohne Childs sind Leafs
- Nur diese werden durchsucht
- Aufwand Aufbau: Linear von Elementzahl abhängig

Beispiel: 2D-Tree Aufbau



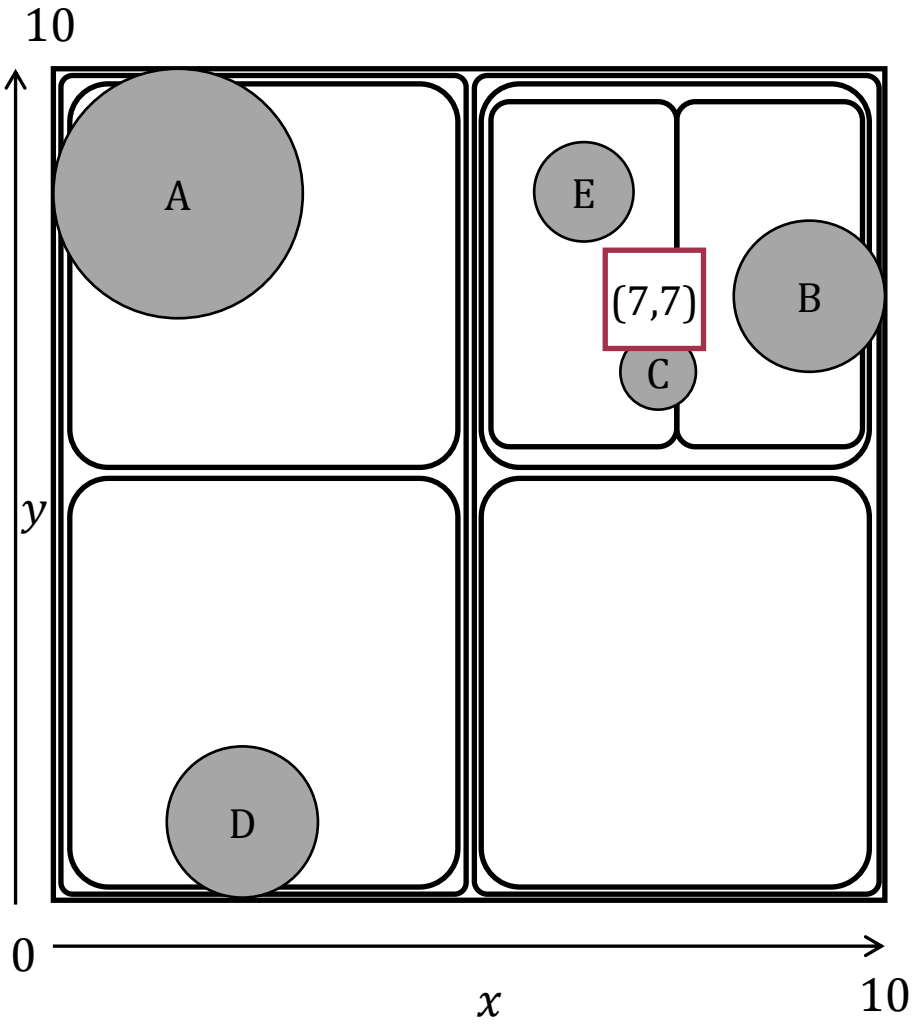
targetElementsPerNode 1
maxDepth 4

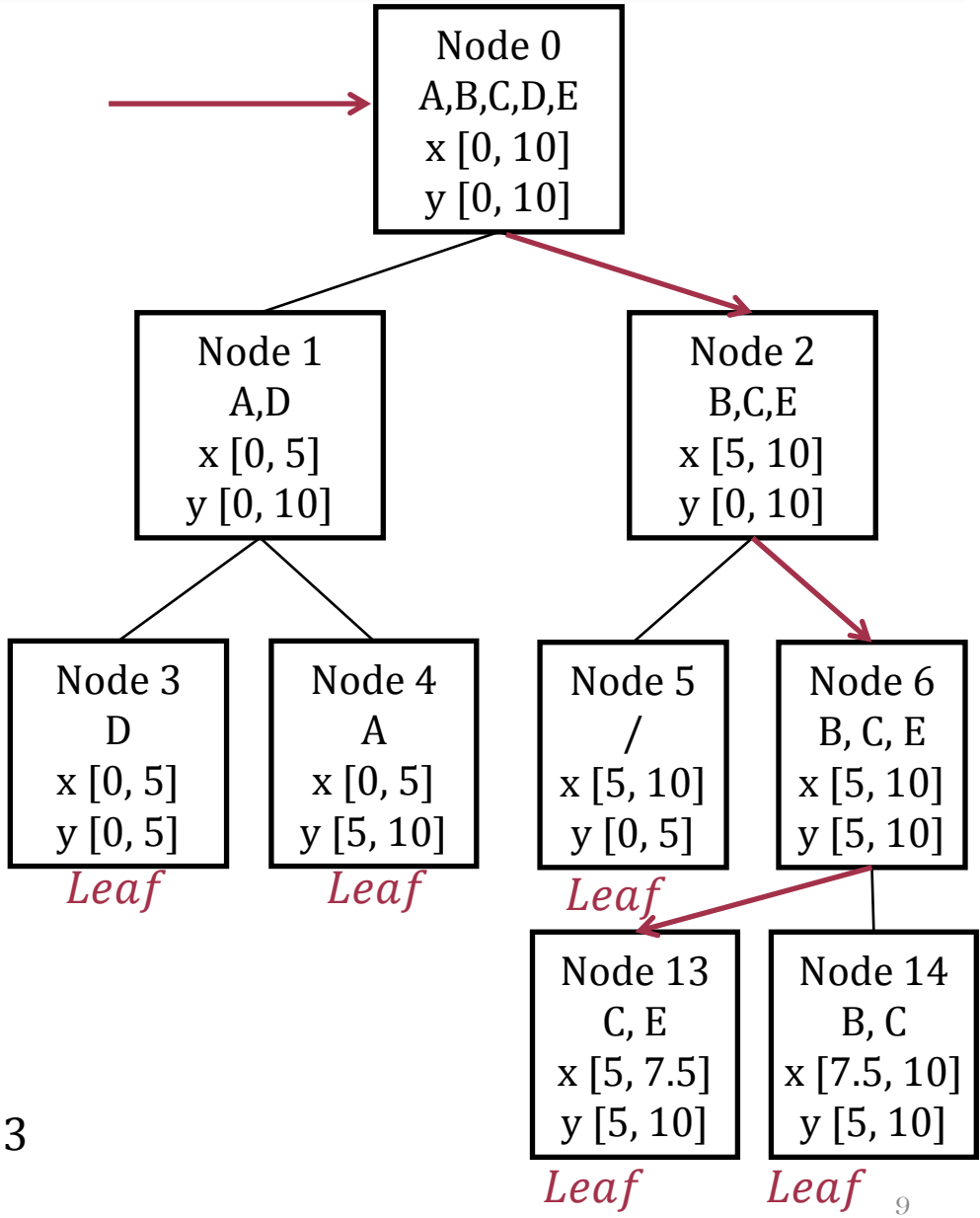


Suche in KD-Tree

- Gegeben: Pixel mit Koordinaten (x, y)
- Folge Baum, ausgehend von der Wurzel, durch Knoten, die (x, y) enthalten
- Durchsuche Leaf-Nodes linear nach Objekten, die (x, y) überlappen
  - Linear, wie beim naiven Ansatz

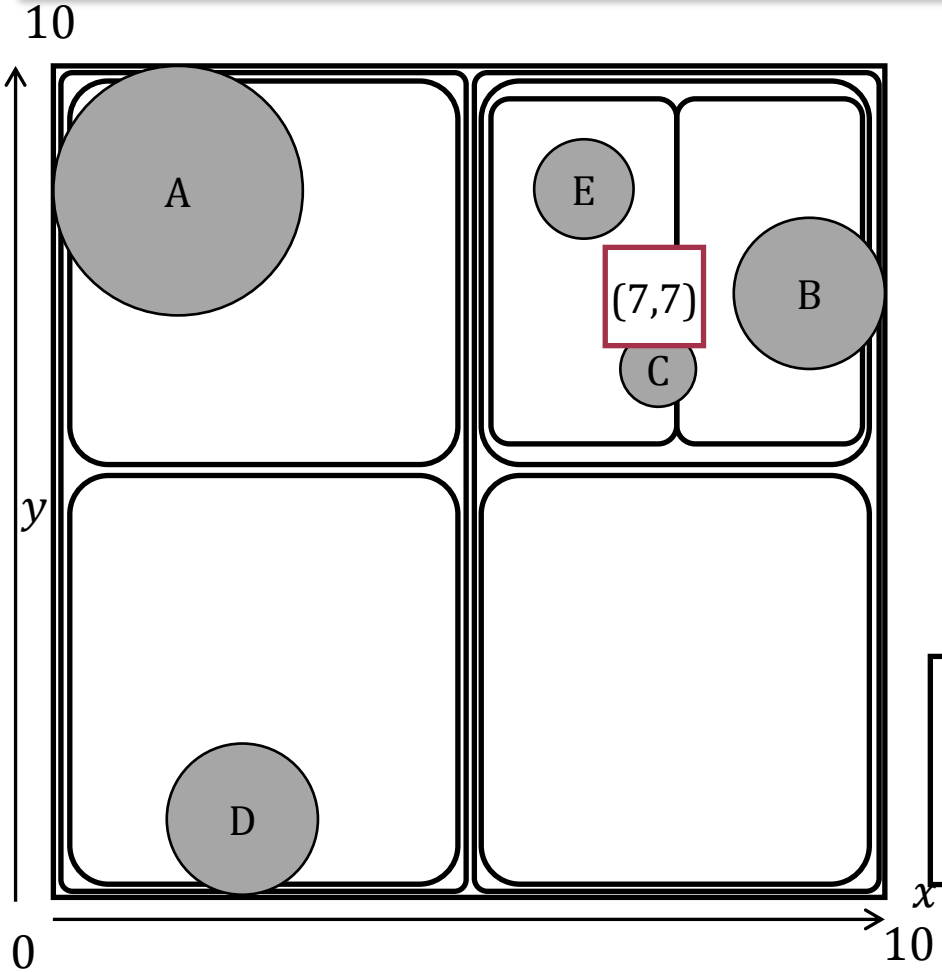
I. Suche Pixel (7,7) enthaltende Leafs

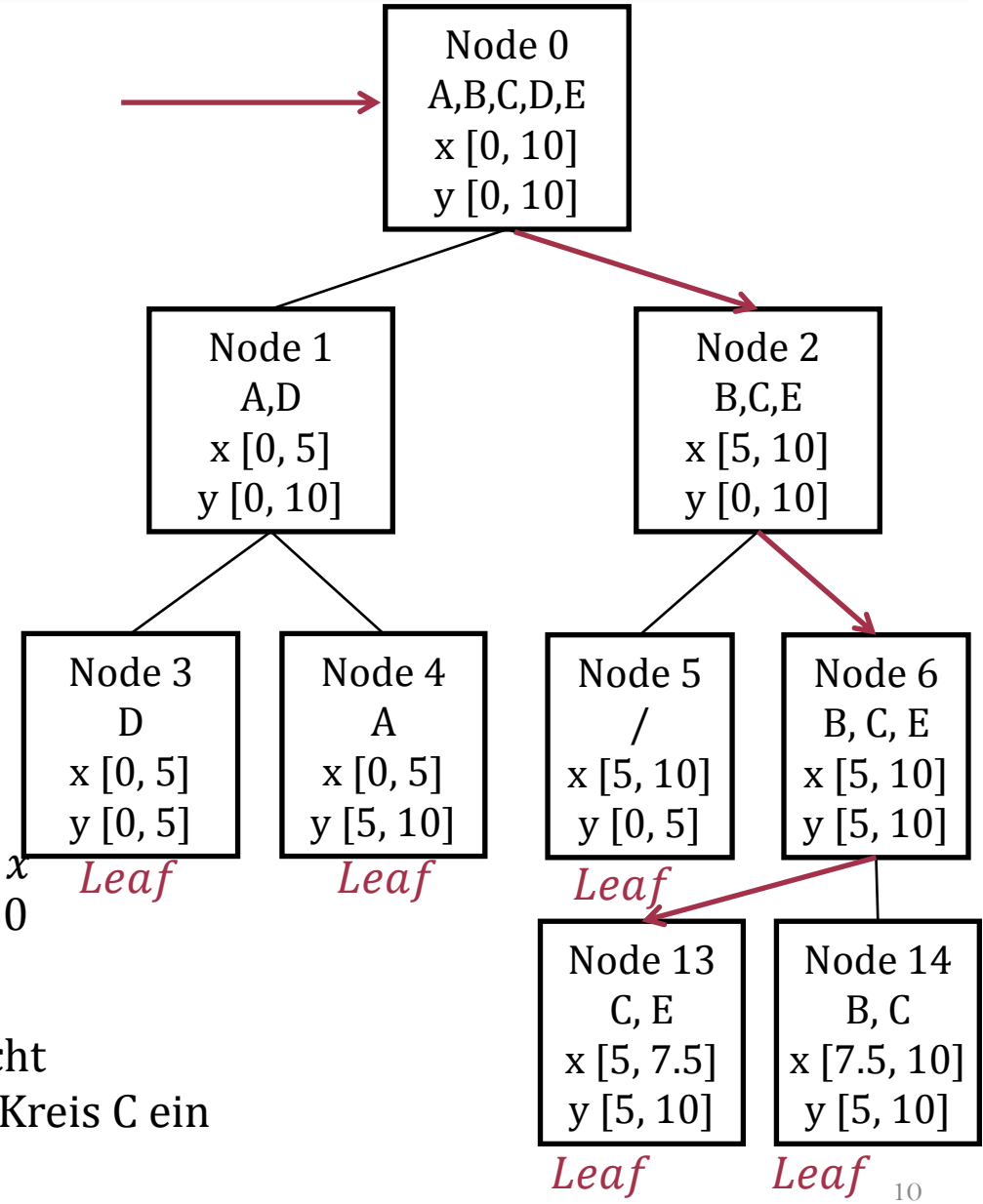




- Pixel (7,7) enthalten in Leaf-Node 13

II. Durchsuche Objekte Node 13





- Node 13: C überlappt Pixel
- Node 13: E überlappt Pixel nicht
- Also: Färbe Pixel (7,7) gemäß Kreis C ein

Implementation: Daten

```
// Modelliert Beziehung einer Node zu Child Nodes
// und enthaltenen Objekten
typedef struct NodeRel {
  int id;           // Node-Id. ids entsprechen Position im Array
                    //   (Diese hier nicht unbedingt nötig)
  int lChild;       // id der linken Child-Node
  int rChild;       // id der rechten Child Node
  int circleMapOffset; // offset zum Zugriff in „mapNode2Circles“
                       //  nötig, um zugehörige Circles zu finden
} NodeRel;
```

```
// Bounding Box einer Node
// Gibt Bereich an, den diese repräsentiert
typedef struct BBox2D {
  float minX; // minimaler und
  float maxX; //   maximaler x-Wert
  float minY; // minimaler und
  float maxY; //   maximaler y-Wert
} BBox2D;
```

Finde enthaltende Child Node

```
// Liefert eine Child Node, welche Pixel mit pixelPos enthält
// Es gilt: Pixel mit in Parent Node enthalten.
//   Folglich muss er auch in mindestens einem Child enthalten sein
NodeRel getContainingChild(BBox2D bBoxLeft, // Bbox des linken Childs
                           NodeRel cL,      // Linke Child Node
                           NodeRel cR,      // Rechte Child Node
                           int2 pixelPos    // Position des Pixels
){
  if(   bBoxLeft.minX <= pixelPos.x // Teste,
     && bBoxLeft.maxX >  pixelPos.x // ob im
     && bBoxLeft.minY <= pixelPos.y // linken Child
     && bBoxLeft.maxY >  pixelPos.y // enthalten
  )
     return cL; // Ja, liefere dieses zurück
  else
     return cR; // Sonst muss es im anderen enthalten sein
}
```

Implementation: Datenlayout

```
// Pointer mit Structs des Typs Circle, genau wie bei naiver Variante
global Circle*       circles

// Repräsentieren zusammen die Nodes des KD-Trees. Erster Eintrag: Rootnode
// Nach Erzeugung dichtgepackt. nodes_Relations[nIndex], nodes_flag[nIndex] und
// nodes_BBox[nIndex] ergeben zusammen alle Daten der Node mit id nIndex
global nodeRel*      nodes_Relations // Beziehungen zu anderen Nodes & Circles
global int*          nodes_flag      // Left, Right oder Leafnode
global BBox2D*       nodes_Bbox      // Ausdehnung der Node

// Enthält Integer Einträge variabler Anzahl.
// Ein Eintrag ist organisiert:
//   -Erstes Wert: Gibt Anzahl cCnt der Circles einer Node an
//   -Folgende cCnt Werte: Ids der Circles der Nodes.
//      Jeweils Position im Buffer circles"
global int* mapNode2Circles

// Beispiel, gegeben:
//   -Node mit "circleMapOffset" 4
//   -mapNode2Circles{3, 7, 78, 1, 2, 90, 90000, ..};
// Die Node enthält 2 Kreise
// Deren Ids sind 90 und 90000
```

Kernel

```
constant int LEAF = 0;
kernel void implicit2DRenderer_KD(
global NodeRel*    nodes_Relations,
global int*          nodes_flag,
global BBox2D*       nodes_BBox,
global Circle*       circles,
global int*          mapNode2Circles,
write_only image2d_t rederedImage) {
  int2 pPos = (int2)(get_global_id(0), get_global_id(1));
  NodeRel node = nodes_Relations[0];   // Beginne mit Root Node
  while(nodes_flag[node.id] != LEAF) { // Traversiere KD-Tree bis LEAF gefunden
    NodeRel cL = nodes_Relations[node.lChild];
    NodeRel cR = nodes_Relations[node.rChild];
    node = getContainingChild(nodes_BBox[cL.id], cL, cR, pPos); // Folie 12
  }
  float minDepth = 100000; float4 color = (float4) (0.0, 0.0, 0.0, 0.0); // Hintergrund
  int nodeCircleCnt = mapNode2Circles[node.circleMapOffset];
  for(int i = 1; i <= nodeCircleCnt ; ++i) {
    Circle c = circles[mapNode2Circles[node.circleMapOffset + i]];
    if(overlaps_N_Nearer(c, pPos, minDepth)) {  // Folie 29, gestern
      minDepth = c.depth;
      color = c.color;
    }
  }
  write_imagef(rederedImage, pPos, color);
}
```

Vergleich

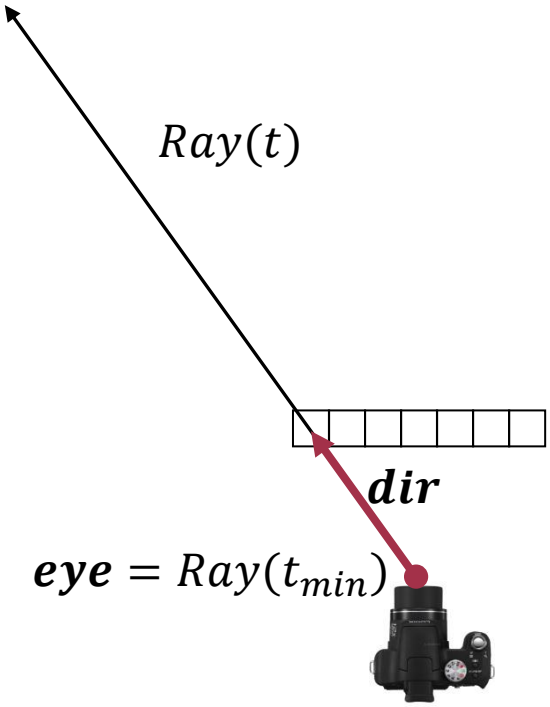
- Gegeben:
  - 1024 x 1024 Bildpunkte
  - 8192 Kreise
  - GTX 470
- Naiver Ansatz:
  - Ca. 8 000 000 000 Pixel-Kreis Vergleiche pro Bild
  - 2,5 FPS
- KD-Tree
  - Vermutlich << 8 Milliarden Pixel-Kreis Vergleiche pro Bild
  - 311 FPS

16.3

KD-Tree Raytracing

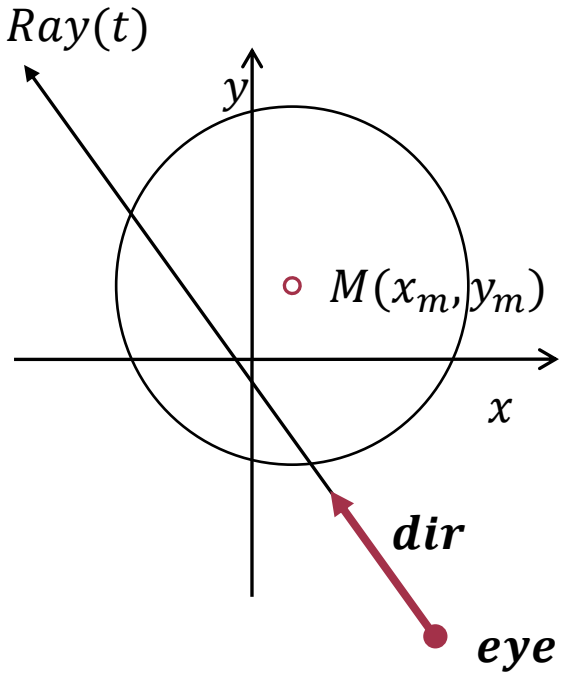
Viewing

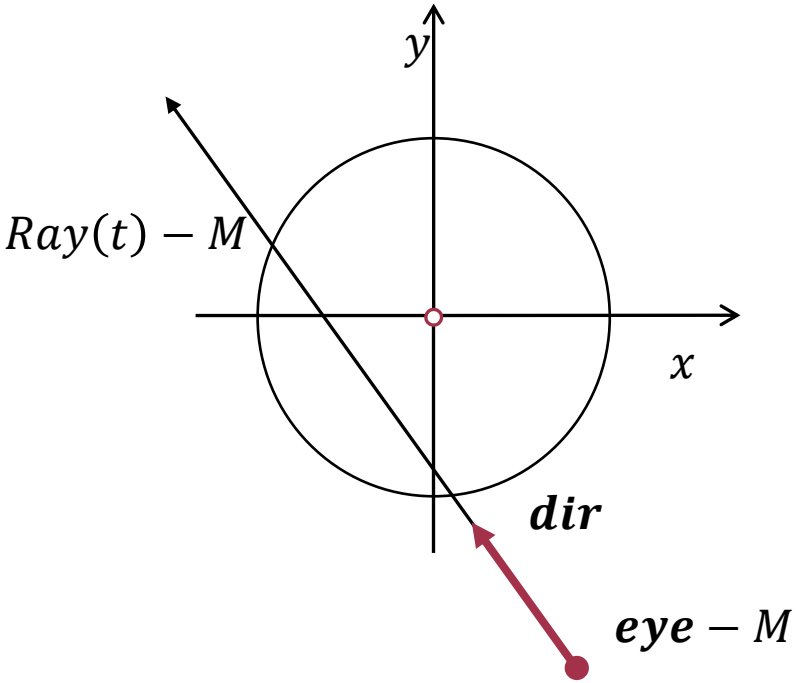
- Erzeuge Strahl aus Betrachterposition und Vektor zur Pixelposition
- $Ray(t) = \boldsymbol{eye} + t \cdot \boldsymbol{dir}$
- Definiere $t_{\min} = 0$
- Ausblenden von Dingen hinter dem Betrachter?
  - Teste, ob $t_{hit} > t_{\min}$
- Field of View?
  - Verhältnis aus Displaybreite und Distanz dazu
  - Im Strahl hinterlegt
- Wende anschließend View Transformation auf Stahl an



Schnittpunktberechnung mit Kugel I

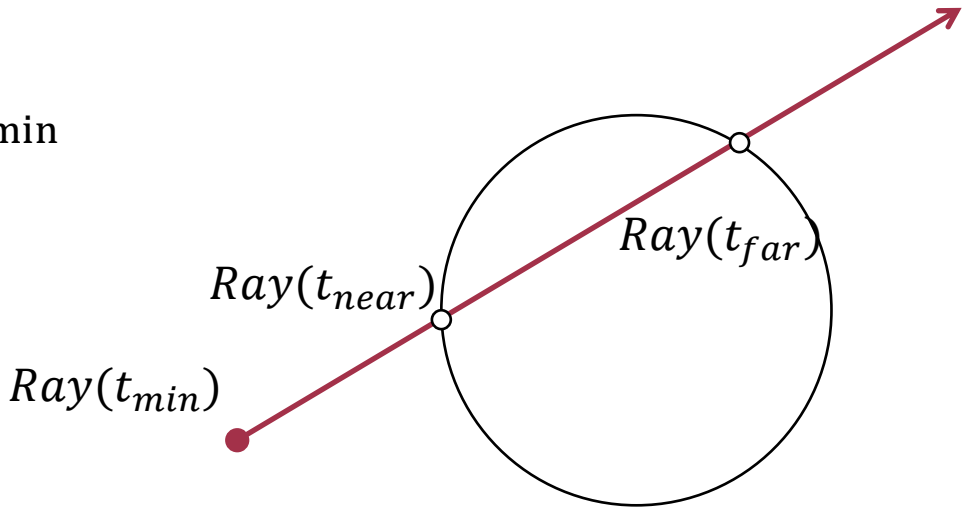
- Verschiebe Kugel zunächst in Ursprung
- Wende gleiche Transformation auf Ray an
- Nachfolgende Berechnungen dadurch vereinfacht





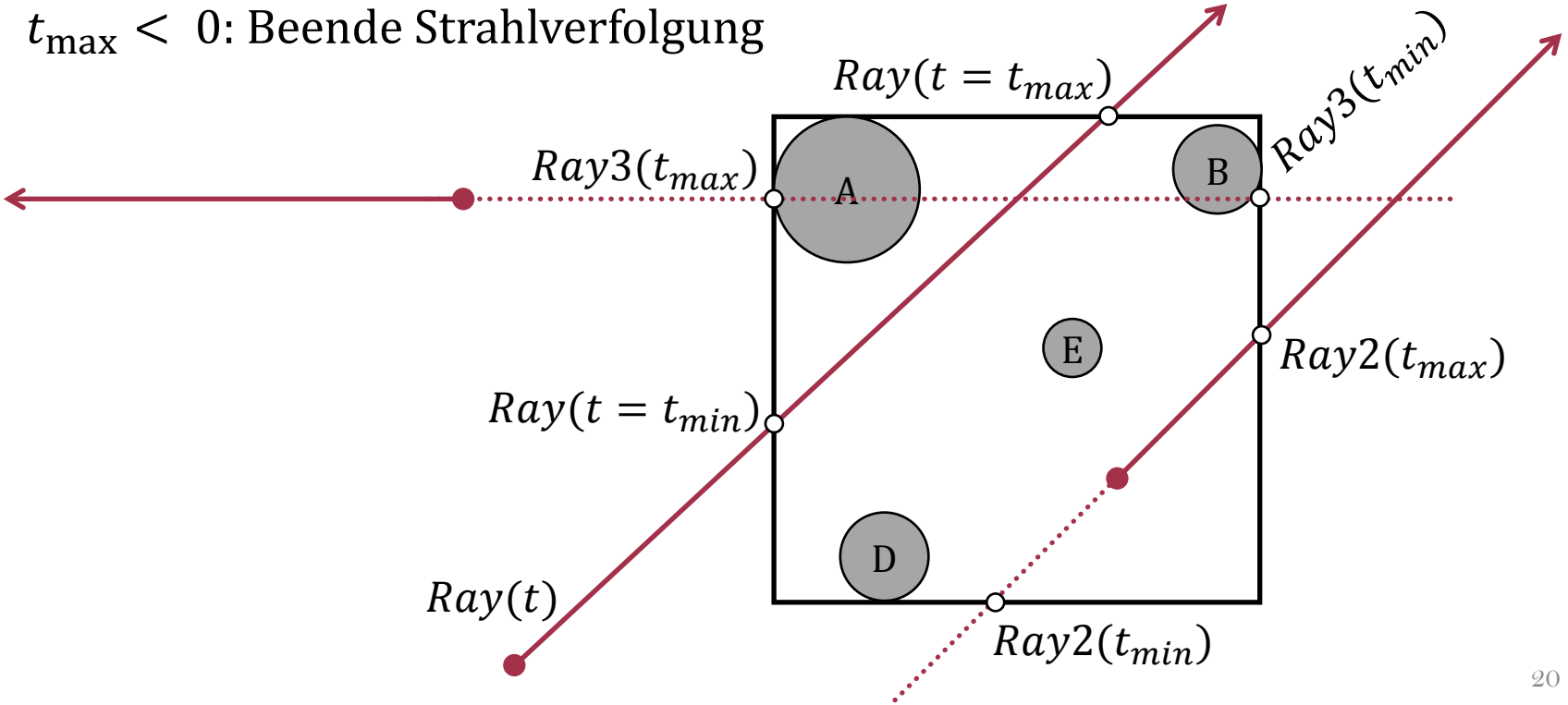
Schnittpunktberechnung mit Kugel II

- $Ray(t) = \boldsymbol{eye} + t \cdot \boldsymbol{dir}$
- Implizite Kugelgleichung:
  - $K_{r,M} = \{(x, y, z) \mid r^2 \geq (x - x_m)^2 + (y - y_m)^2 + (z - z_m)^2\}$
- Bei Ursprungskugel:
  - $K_{r,M} = \{(x, y, z) \mid r^2 \geq x^2 + y^2 + z^2\}$
- Für Schnittpunkt(e) muss gelten:
  - $(eye_x + t \cdot dir_x)^2 + (eye_y + t \cdot dir_y)^2 + (eye_z + t \cdot dir_z)^2 = r^2$
- Quadratische Gleichung von t, hat 0, 1, oder 2 reelle Lösungen
- Falls 2 gilt: $t_{\min} < t_{near} < t_{far}$
- Liefere kleineren Wert, falls $> t_{\min}$



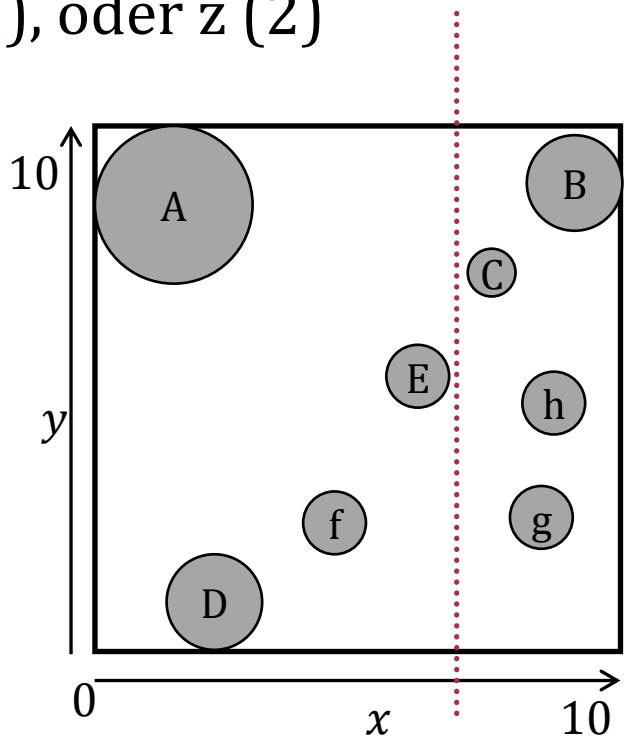
Sichtbarer Teil des Strahls

- Gesamte Szene in Bounding Box enthalten
- Bestimme Parameter $t_{\min}$ und $t_{\max}$ für Ein- und Austrittstpunkt des Strahls aus Bounding Box, mit: $0 < t_{\min} < t_{\max}$
- Berechnete Werte $<$ 0 liegen hinter dem Betrachter
  - $t_{\min} <$ 0: Setze $t_{\min}$ auf 0
  - $t_{\max} <$ 0: Beende Strahlverfolgung



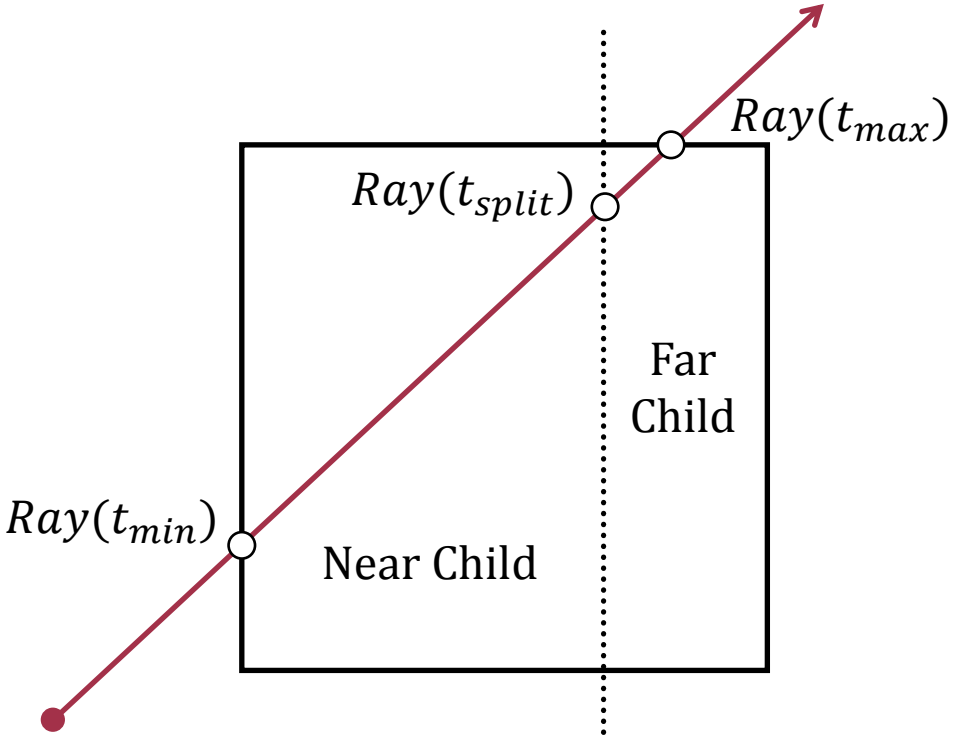
Node im 3D-KD Tree

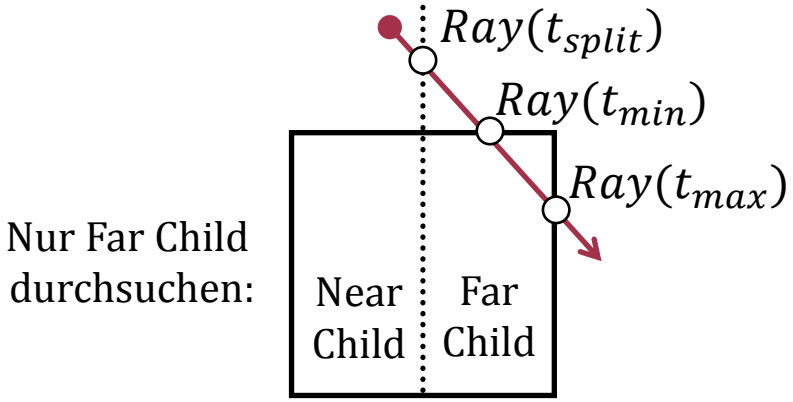
- Ganze KD-Struktur ist Axis Aligned
- Dadurch Tests an Box in jeweils einer Dimension ausführbar
- Nodes enthalten keine Bounding Box, sondern
  - Ganzzahl für Trennachse: x (0), y (1), oder z (2)
  - Float für Position der Trennebene
- Im Beispiel hier:
  - Trennachse: x
  - Splitplane: 7.5

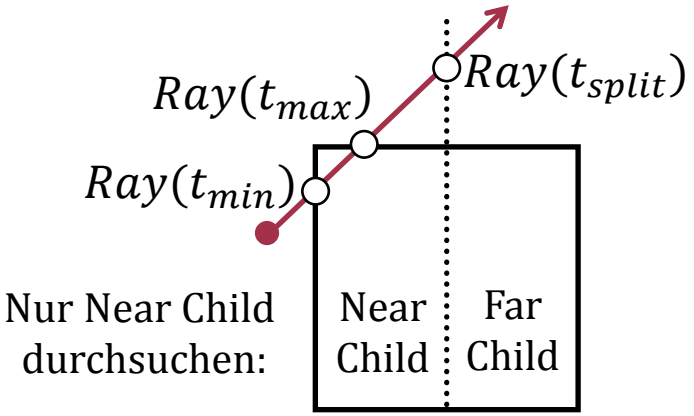


Bestimme zu durchsuchende Childs

- Bestimme t_{split} für Schnittpunkt des Strahls mit Splitplane
- Lege Near / Far Child durch Vergleich v. eye mit $Ray(t_{split})$ fest
  - Führe Suche in Near Child fort und lege Far Child auf Stack
  - Erfolglos? Nur dann (später) in Far Child fortsetzen
  - Manchmal eins ausschließbar

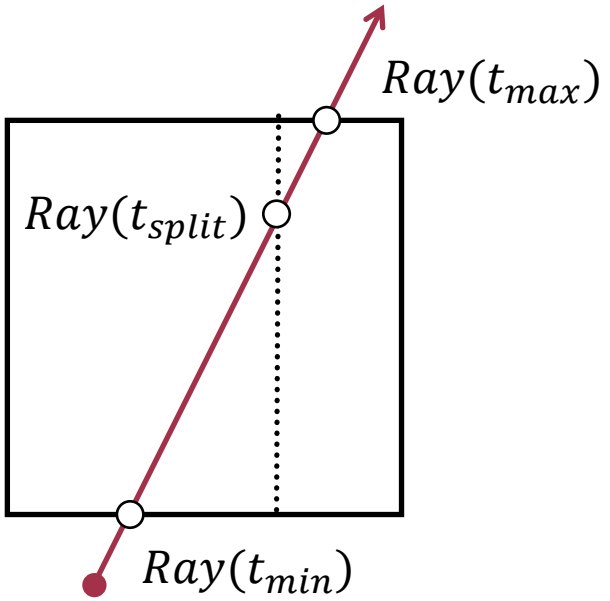


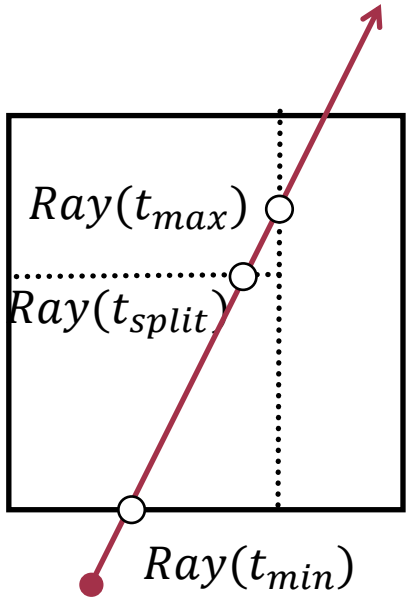


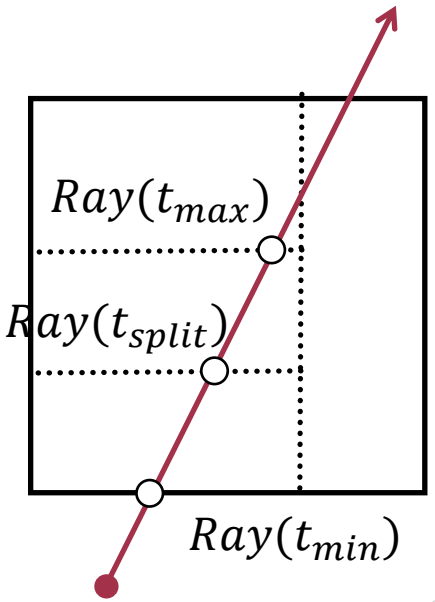


Begrenze Strahl für Child

- Bei Durchsuchung der Childs wird Strahl „verkleinert“ indem $t_{\min}$ oder $t_{\max}$ auf t_{split} gesetzt wird
- $t_{\min} < t_{split}$? Setze $t_{\min}$ auf t_{split}
- $t_{\max} > t_{split}$? Setze $t_{\max}$ auf t_{split}
- Der KD-Tree in Near Node müsste hier jeweils mit t_{min} und t_{split} weiter verarbeitet werden
- Die Far Node müsste hier jeweils mit t_{split} und $t_{\max}$ auf Stack

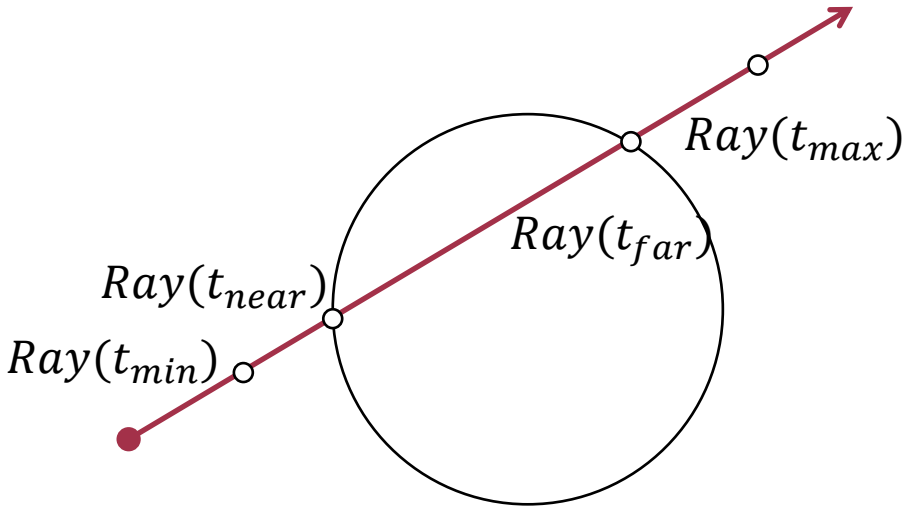






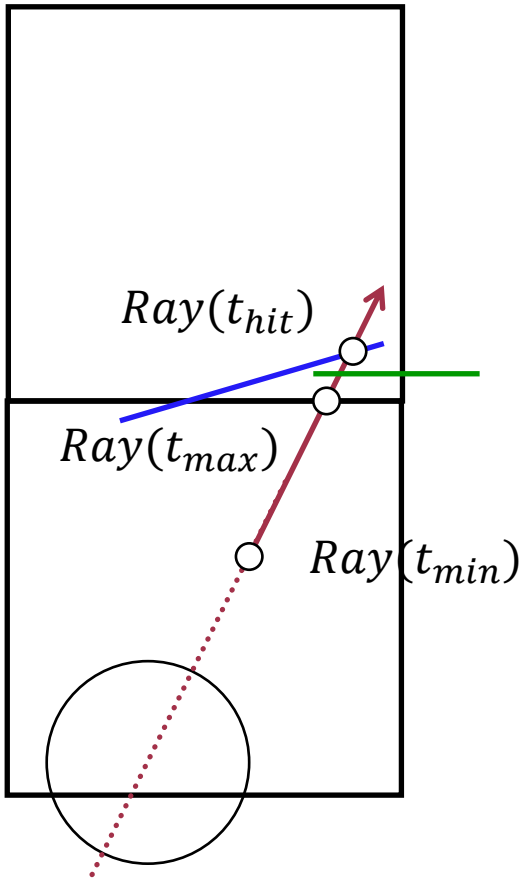
Durchsuche Leaf

- Teste der Reihe nach alle enthaltenen Elemente auf Schnittpunkt(e) t_{near}, t_{far} mit Teilstrahl $(t_{\min}, t_{\max})$ in dieser Node (siehe Folien 18, 19)
- Liefert ggf. den näheren Schnittpunkt (nächste Folie)
- Vergleiche dieses jeweils mit nächstem bisher in der Node gefundenen Schnittpunkt & überschreibe, falls näher
- Da nähere Nodes erst durchsucht werden, kann Traversierung nach vollständigem Durchsuchen der Node beendet werden, falls Schnittpunkt gefunden



Berechne SP mit sichtbarem Teilstrahl

- Berechne Schnittpunkt(e) einer Kugel mit Strahl
- Falls t_{near} und / oder t_{far} gefunden, überprüfe:
  - $t_{min} < t_{near}, t_{far} < t_{max}$
  - Nur dann sichtbar & in dieser Node
  - Andere Node muss dann noch untersucht werden, da nähere Objekte enthalten sein könnten, die nicht in dieser Node sind
- Falls Schnittpunkte existieren, liefere näheren zurück, der diese Bedingung erfüllt



KD-Tree Raytracing (vereinfacht)

```
Nodes todoList  // Stack mit Far Nodes, ggf. noch zu durchsuchen
Hit nearestHit  // Infos zum getroffenen Objekt
Node actNode    // Aktuell zu verarbeitende Node
<Intialisierung des Strahls> // (Folie 17)
actNode <- Root Node // Beginne mit Root Node
While (true)
     Falls keine LEAF-Node
          <Bestimme zu durchsuchende Child Nodes> // (Folie 22)
          <Begrenze Strahl für Child Nodes >      // (Folie 23)
           actNode <- Near Node, falls existent,
           sonst actNode <- Far Node
           Packe, falls existent, weitere Node auf Stack
     Sonst
          <Durchsuche Leaf Node >                 // (Folie 24)
               Treffer? Schreibe nearestHit & Beende Traversierung
          todoList leer? Beende Traversierung
               Sonst: actNode <- pop(todoList)
 End While
 Treffer? <Shade Pixel gemäß nearestHit>
 Sonst: Setze Hintergrundfarbe
```

Ankündigungen

- Klausur: Bitte in Opium anmelden
- Übungen diese Woche
- Montag
- Freitag
- Dienstag
- Übungen nächste Woche