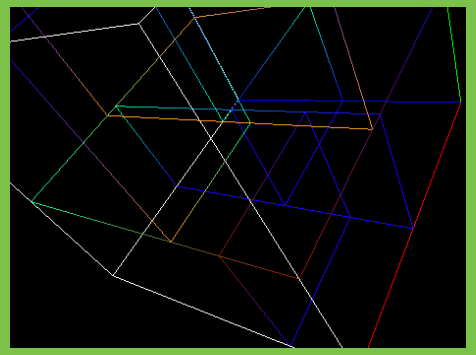
Visual Computing – Sichtbarkeit & Verdeckung

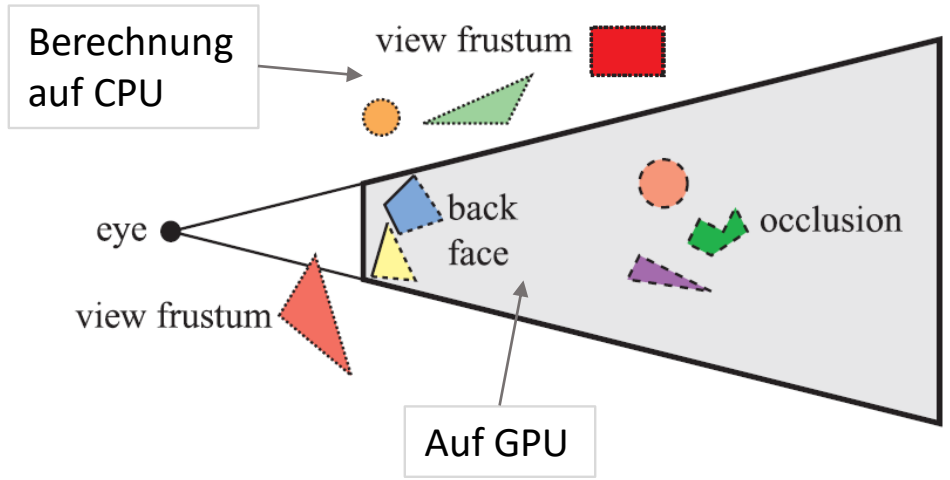


Yvonne Jung

Sichtbarkeitsbestimmung

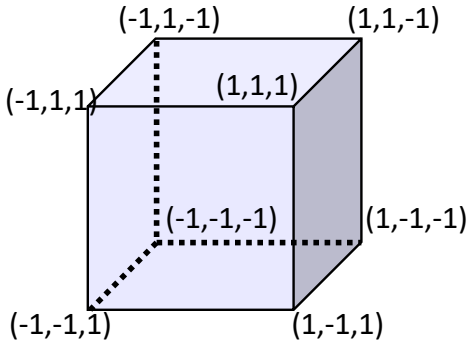
h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Gute Bildqualität und schnelle Darstellung erfordern korrekte Ermittlung der sichtbaren und Beseitigung unsichtbarer Bildteile
- Entfernen von Szenenteilen, bei denen man Sichtbarkeit durch einfache Tests feststellen kann
  - Vorzugsweise komplette Objekte bzw. zumindest ganze Polygone
- Arbeitsvermeidung durch Culling
  - Backface Culling
  - Viewfrustum Culling
  - Occlusion Culling
- Culling-Techniken im Vergleich (s. Abb.):
  - Nur Frustum Culling auf Anwendungsebene

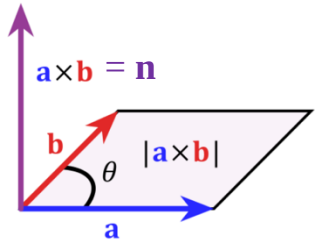


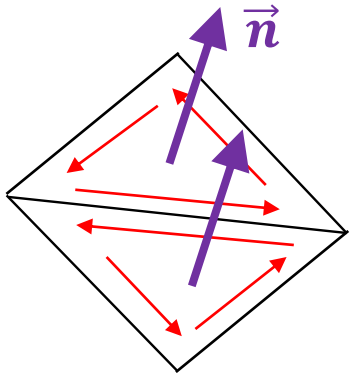
Akenine-Möller et al., 2008. Real-Time Rendering

Backface Culling

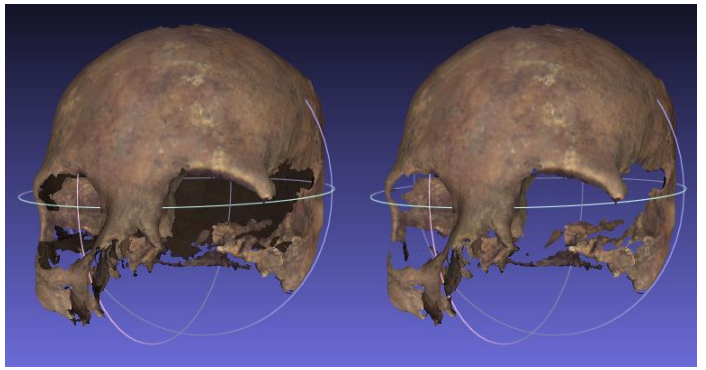


- Grafikkarte geht von geschlossenen, nicht-transparenten Körpern aus
- Rückseitige Flächen, d.h. vom Betrachter abgewandte Seiten, sind nicht sichtbar
  - Müssen also nicht gezeichnet werden
    - Ca. 50% der in Szene vorkommenden Flächen
  - Dazu Rückseitenentfernung
- Erkennung erfordert einheitliche Orientierung der Polygonnetze
  - Kanten benachbarter Polygone haben gegenläufige Orientierung und damit gleiche Normalenrichtung

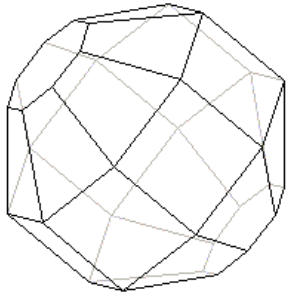


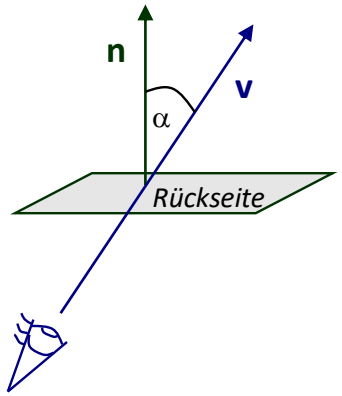


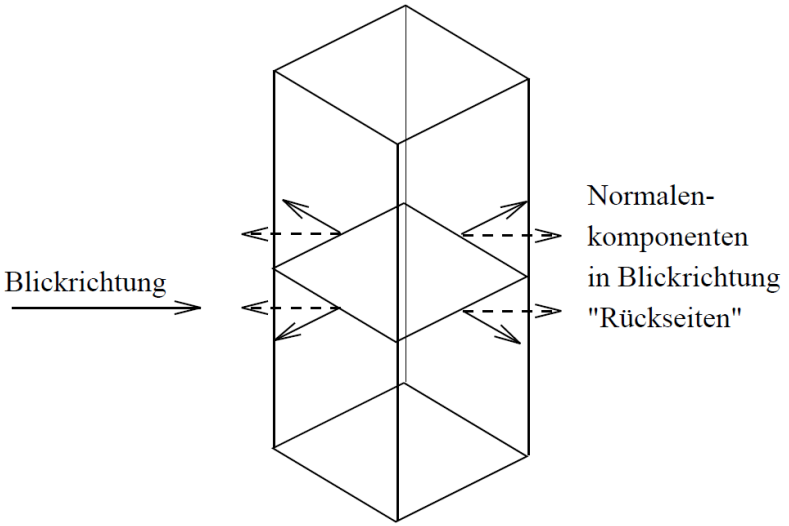
Backface Culling



- Vom Betrachter abgewandte Polygone vom Rendering ausschließen
  - Polygon heißt „front-facing“, falls Skalarprodukt von Normale $\vec{n}$ und Blickrichtung $\vec{v}$ negativ: $\vec{n} \cdot \vec{v} < 0$
    - Sonst abgewandt, d.h. „back-facing“: $\vec{n} \cdot \vec{v} \geq 0$ (identifiziert so rückseitige Flächen)
  - In APIs meist sehr einfach aktivierbar
    - Bsp. WebGL: `gl.enable(gl.CULL_FACE);`

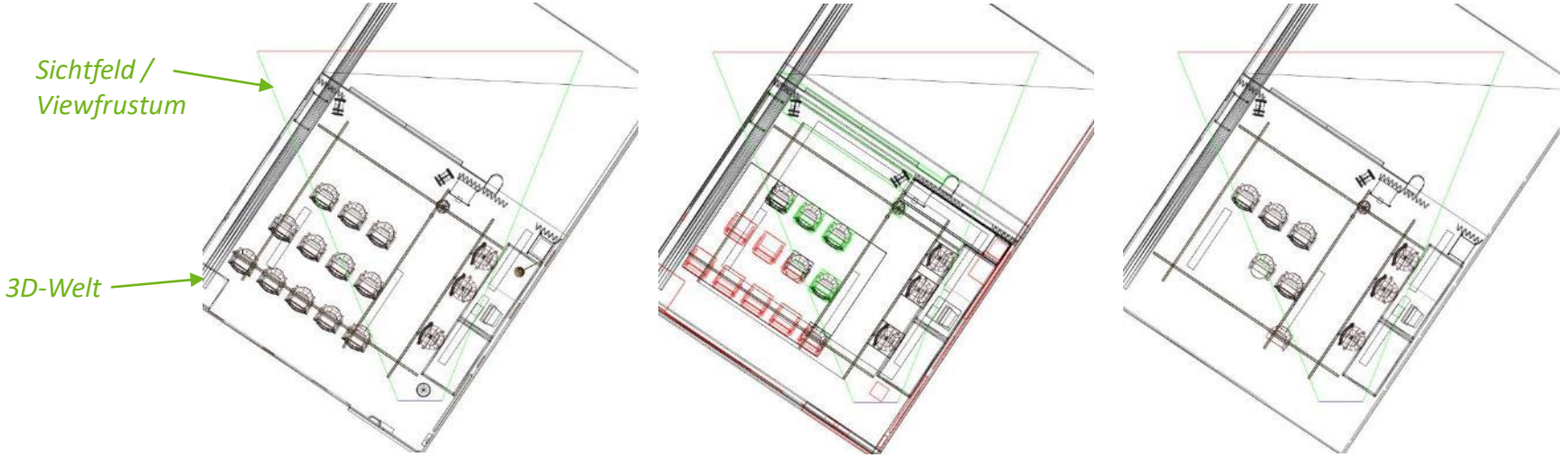


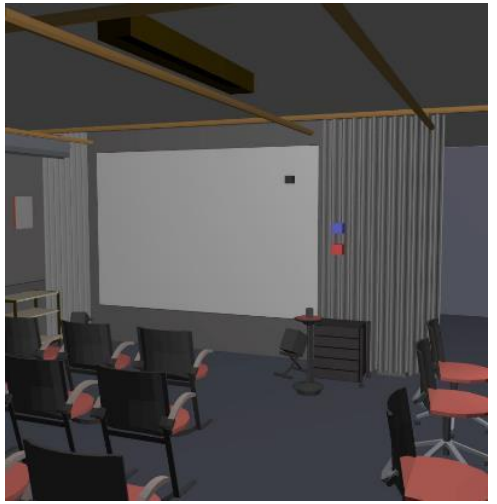




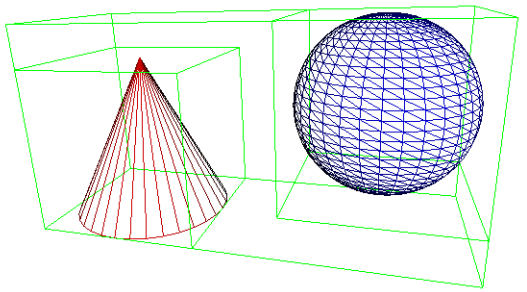
Viewfrustum Culling

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES





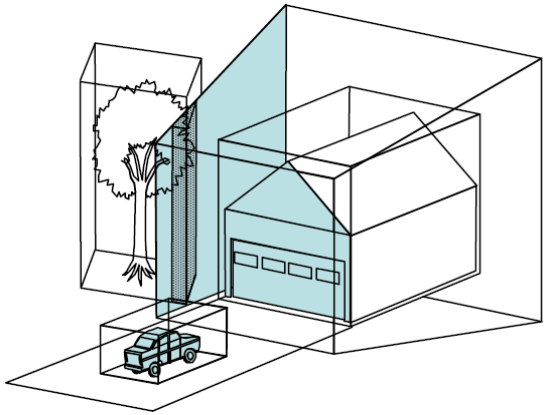
- Alles, was außerhalb des Sichtfeldes ist und im Bild nicht auftaucht, muss nicht gerendert werden
- GPU kann Polygone schneller zeichnen als CPU Test machen kann
  - Nicht einzelne Polygone, sondern konservativ ganze Objekte testen
- Anzahl Tests durch Objekthierarchie (z.B. Szenengraph) minimieren
  - Komplett unsichtbare Teilbäume ignorieren (outside)
  - Komplett sichtbare Teilbäume nicht weiter testen (inside)

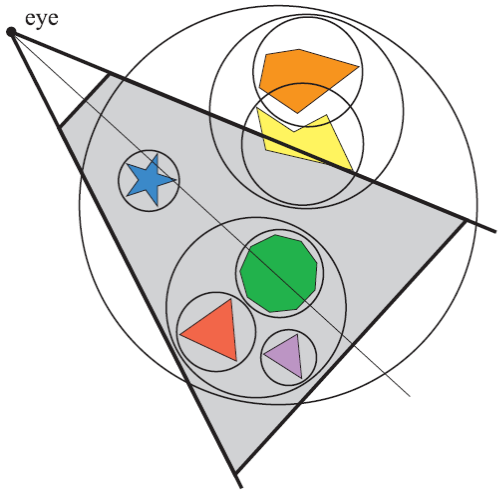


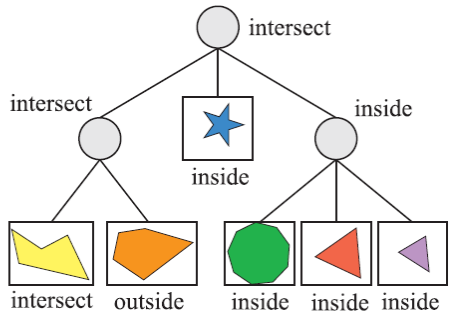
Viewfrustum Culling

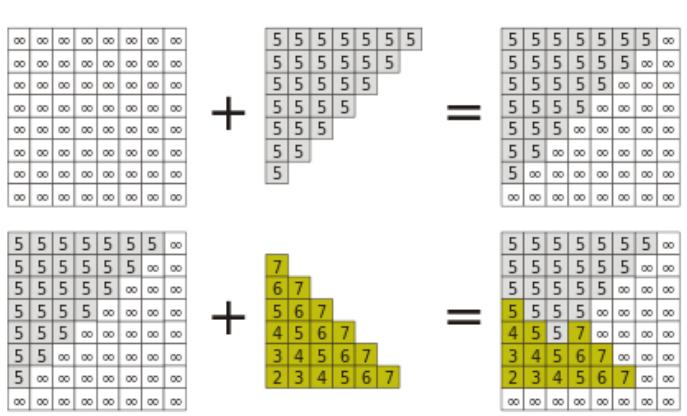
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- 3D-Objekt in Sichtvolumen sichtbar?
  - Anzahl Tests durch hierarchische Struktur reduzieren
  - Jeder SG-Knoten hat einfachen Hüllkörper, der Teilbaum/Mesh einschließt
- Traversiere Hüllkörper-Hierarchie (Bounding Volume Hierarchy)
  - Teste Hüllkörper (i.d.R. Bounding Box) der Knoten gegen Frustum
  - Bounding Box schneidet Viewfrustum?
    - → Traversiere Kinder des Szenengraphknotens
    - Sonst in-/outside: Rendern, wenn nicht outside
- Vorteil: Beschleunigung der Darstellung durch Arbeitsvermeidung
  - Wichtig: Tests u. Hüllkörper müssen einfach sein









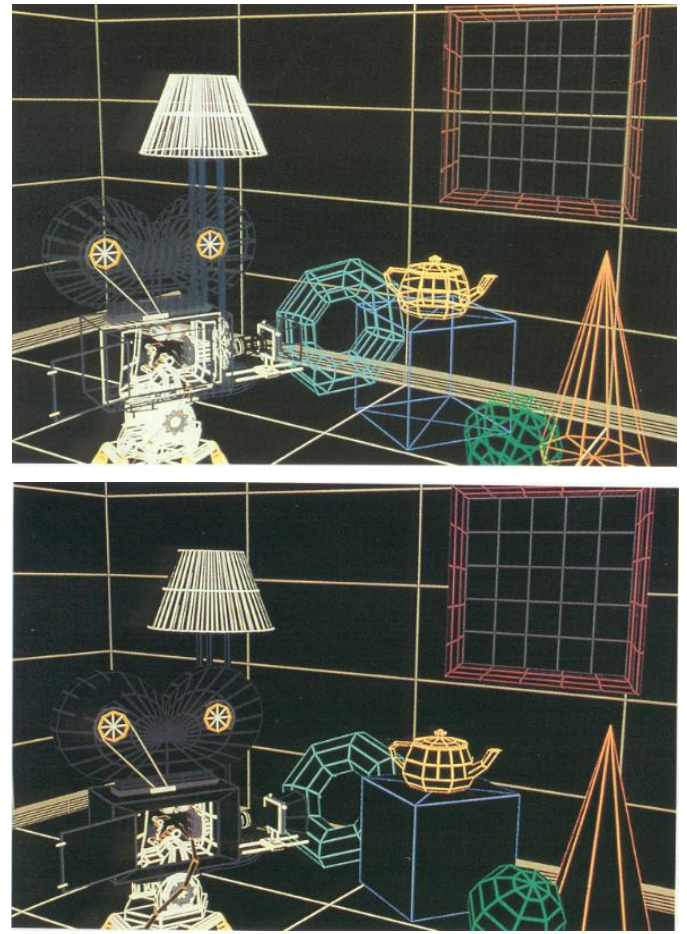
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Das Sichtbarkeitsproblem

Problem gegenseitiger Verdeckungen

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Mehrere 3D-Objekte erhalten durch Projektion gleiche Bildkoordinaten
  - Trotz unterschiedlicher Distanz
- Intuitiv: Objekte werden vom gleichen Sehstrahl getroffen
  - Gedachter Strahl geht von Auge durch Pixelposition
  - Sichtbar ist der dem Auge am nächsten liegende Punkt
- Ist Objekt durchsichtig, werden dahinterliegende Punkte auch sichtbar
  - I.d.R. Sonderbehandlung nötig



Gegenseitige Verdeckungen

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Quelle: Settlers 7, Ubisoft

Gegenseitige Verdeckungen

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Quelle: Settlers 7, Ubisoft

Gegenseitige Verdeckungen

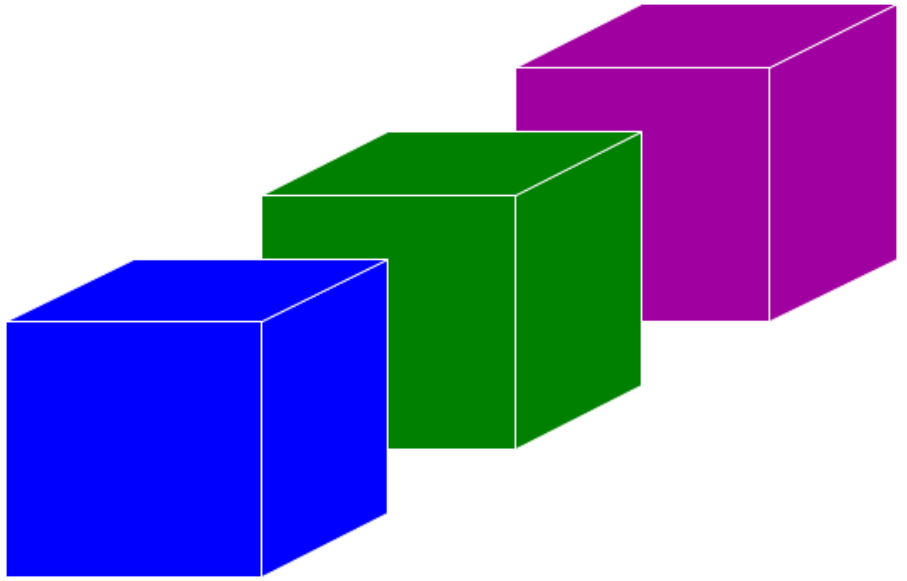
h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Quelle: Settlers 7, Ubisoft

Verdeckungen durch Überzeichnen

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Lösungsidee: Zeichnen aller 3D-Objekte von hinten nach vorne
  - Vorne: Nahe beim Betrachter liegende Objekte
  - Hinten: Entfernt vom Betrachter liegende Objekte
- Typische Vorgehensweise zur Behandlung (semi-) transparenter Objekte
- Ansatz ähnlich wie bei Gemälde mit mehreren Farbschichten
  - Vordere Schichten überlagern hintere Schichten



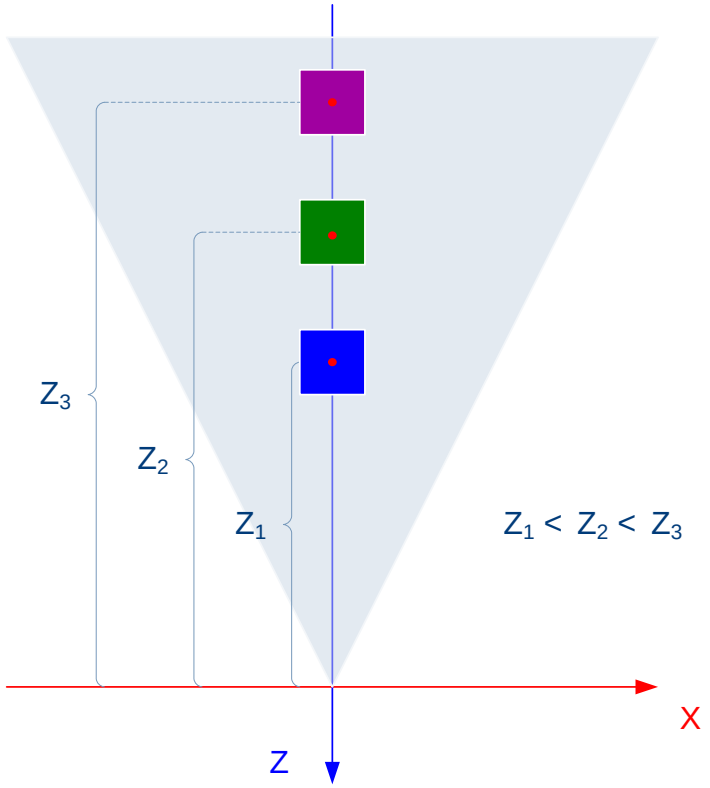
Verdeckungen durch Überzeichnen

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Sortierung anhand der z-Werte im Eye Space (Betrachterraum)

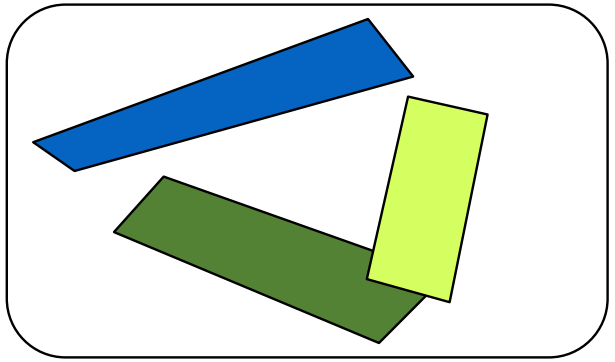
- Nach Multiplikation mit Modelview-Matrix
- Kamera bzw. Augpunkt im Ursprung mit Blickrichtung entlang negativer z-Achse
- Auswahl des Objekt-Mittelpunkts als Bezugspunkt der Sortierung

Problem: Sortierung nicht immer eindeutig möglich

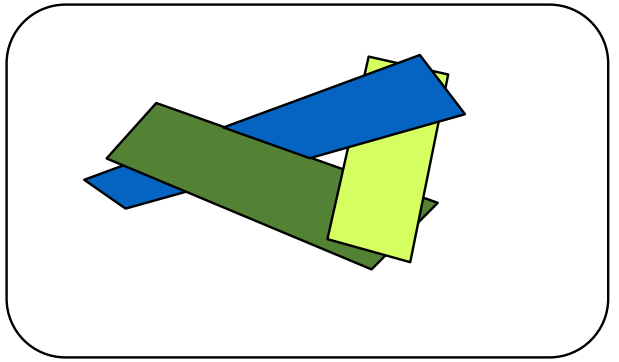


Maleralgorithmus

- Zeichne Polygone wie ein Maler: das am weitesten entfernte zuerst
- Ursprüngliches Verfahren: Sortiere Polygone nach z-Wert: $[z_{min}, z_{max}]$
  - Falls z-Intervalle überlappen, berechne Schnittpolygone
  - Heutzutage höchstens für transparente Objekte, aber nicht auf Polygonebene

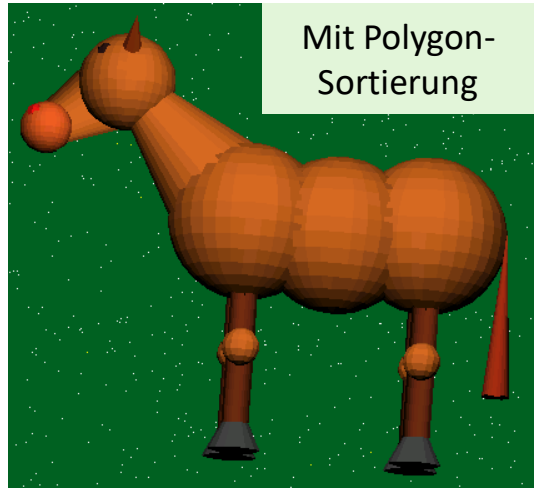


Von hinten nach vorne zeichnen

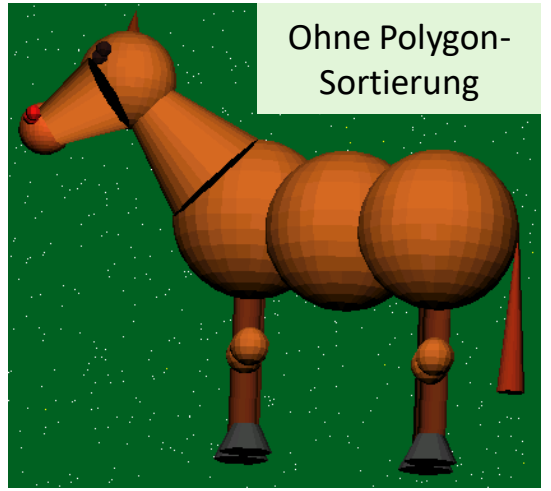


…ist nicht immer eindeutig

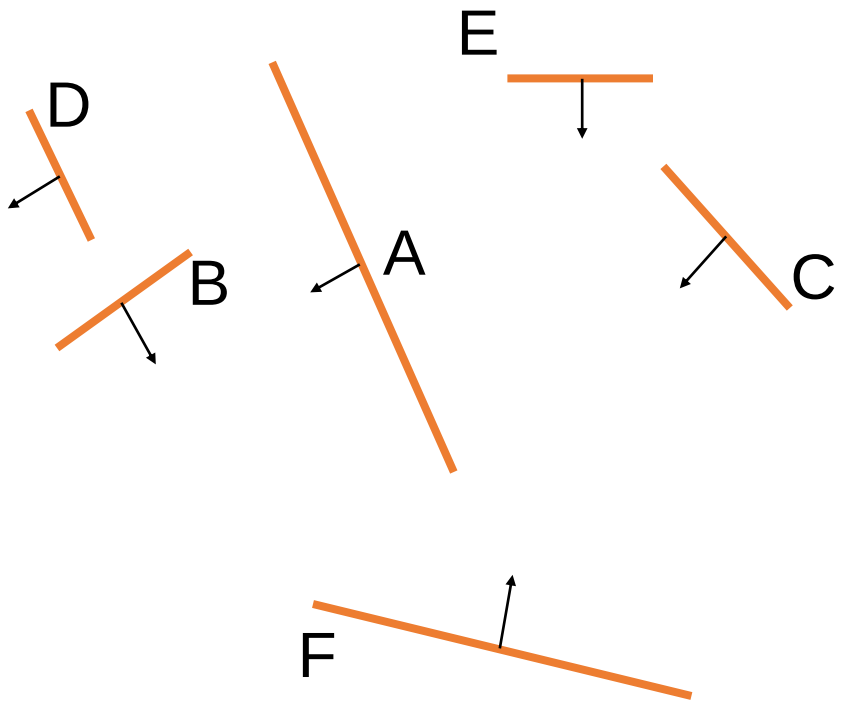
BSP-Baum



- Hierarchische, räumliche Datenstruktur
  - Geeignet zur Auflösung von Verdeckungen und zur räumlichen Sortierung
    - Wurde für Performance u.a. genutzt in den Spielen Quake und Doom
  - Beginne Zeichnen von Polygon mit kleinstem *z*-Wert
- Rekursive Unterteilung des *n*-dimensionalen Raumes in zwei konvexe Teilmengen
  - Trennung mittels sog. Hyperebene: (*n*–1)-dimensionaler Unterraum des IR^n
  - Im 2D: an durch Linien gegebenen Geraden teilen
  - Im 3D: an durch Dreiecke gegebenen Ebenen teilen
- Jeder Knoten entspricht einer Unterteilungsebene
  - BSP-Baum unterteilt Raum in zwei Halbräume: Binary Space Partitioning
  - Also binäre Unterteilung des Raums in zwei Teilbäume
  - Je links und rechts einer Unterteilungsebene

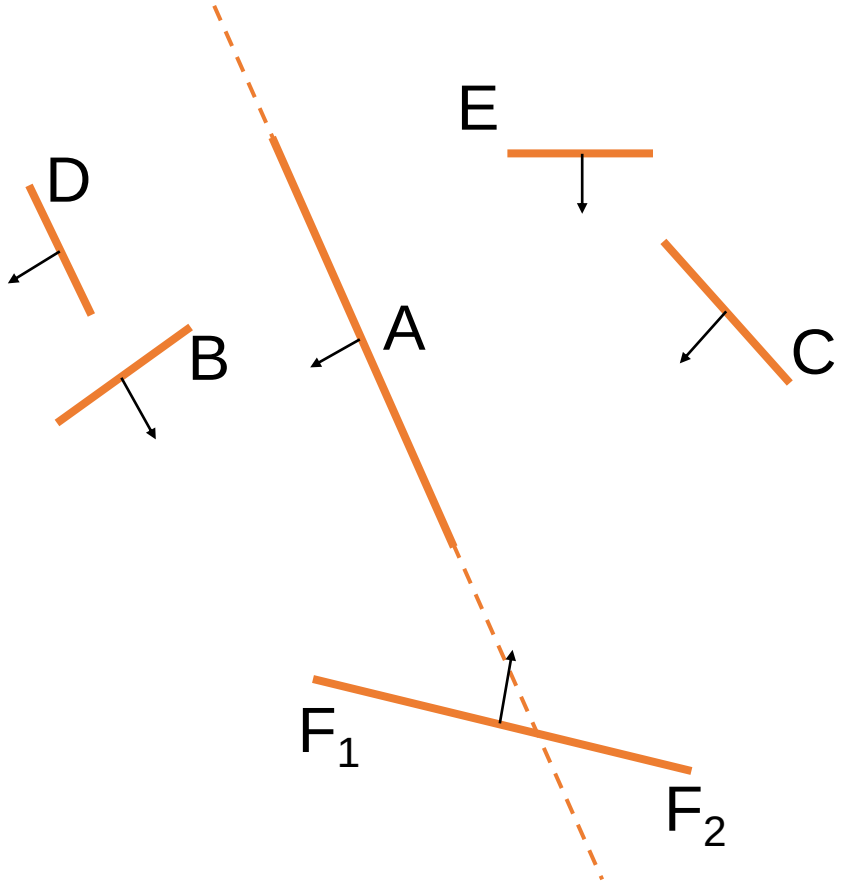


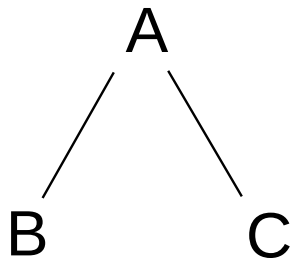
Binary Space Partitioning



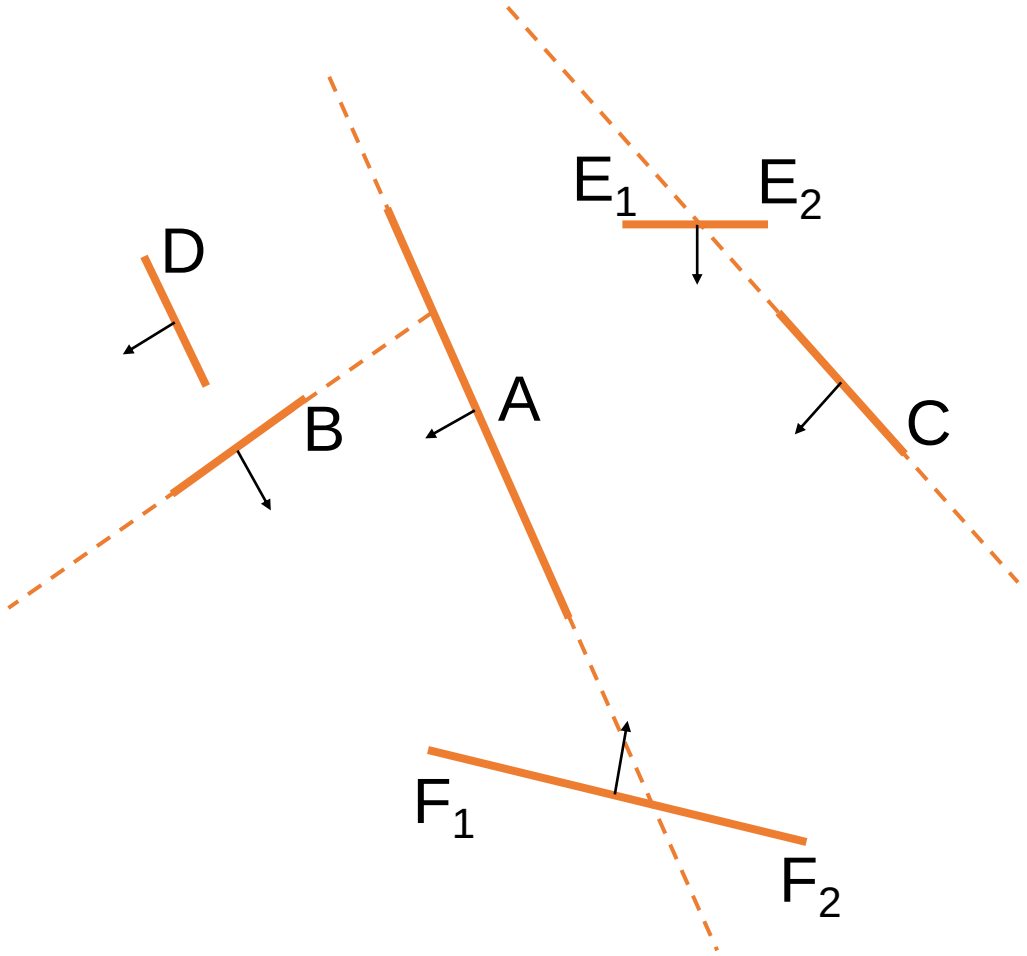
A

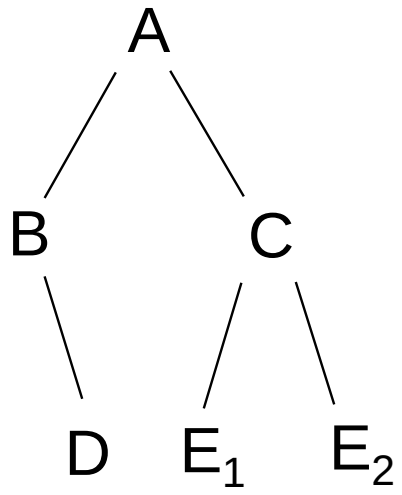
Binary Space Partitioning



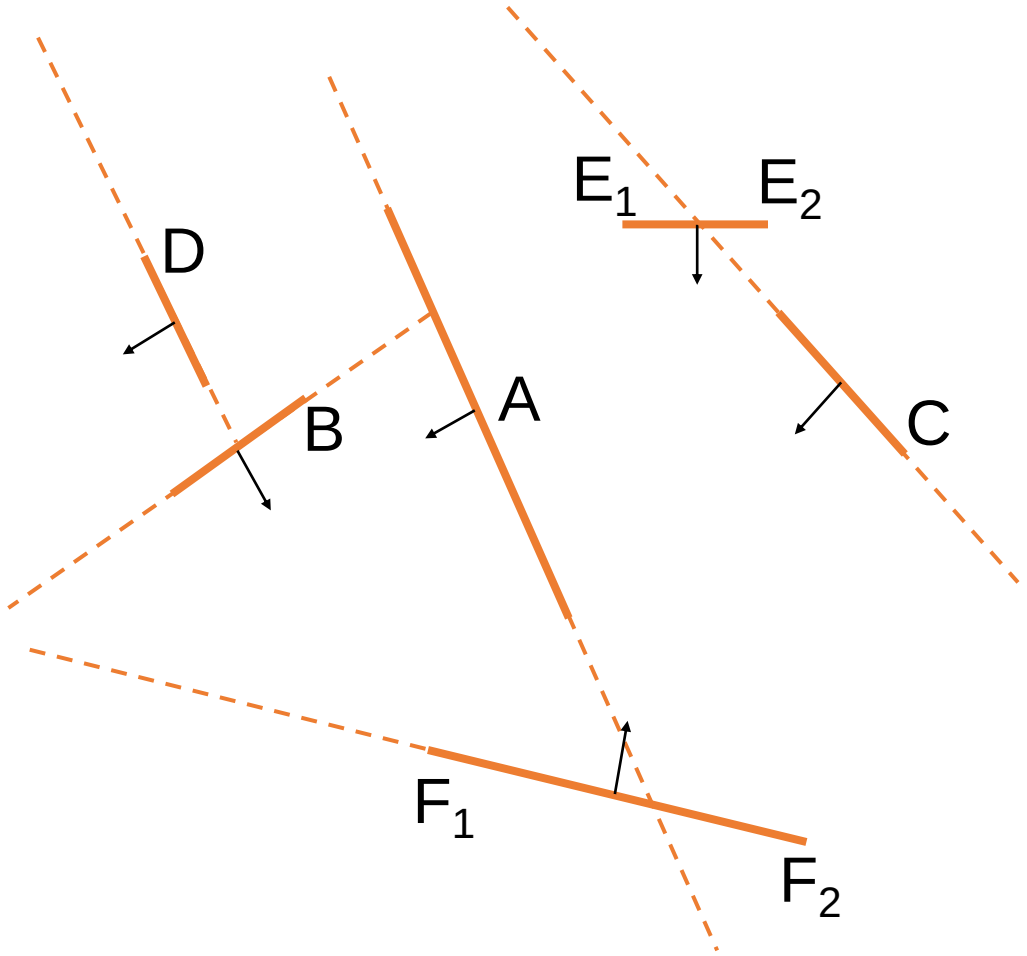


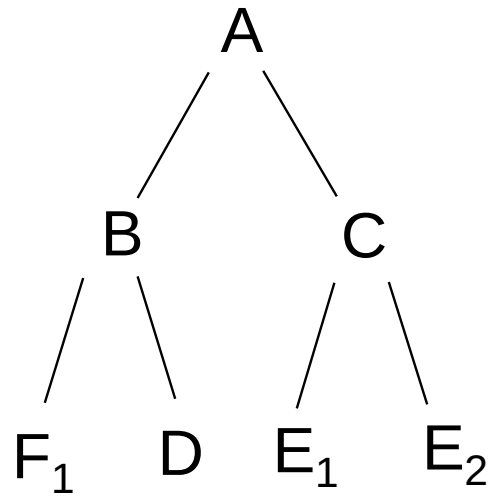
Binary Space Partitioning



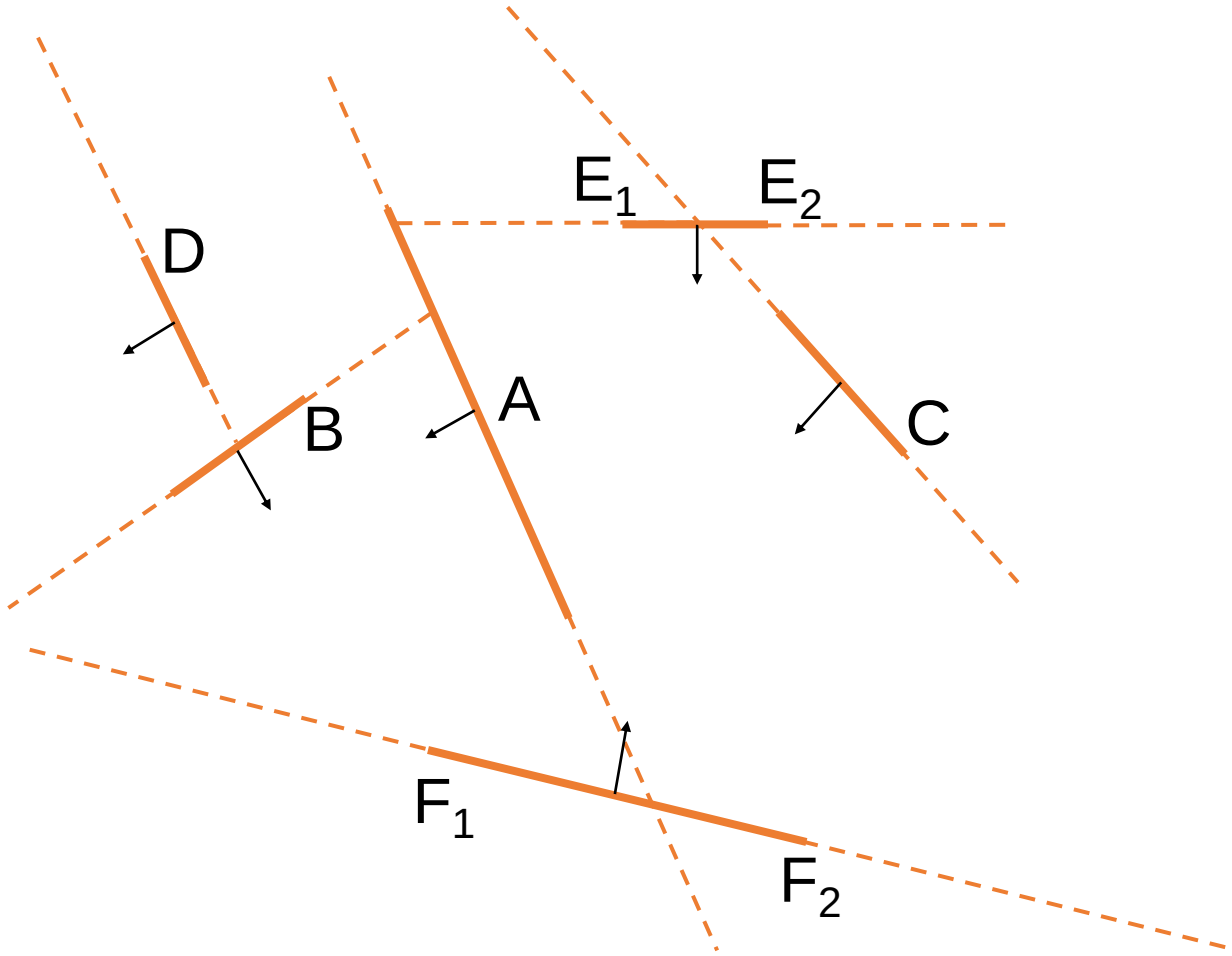


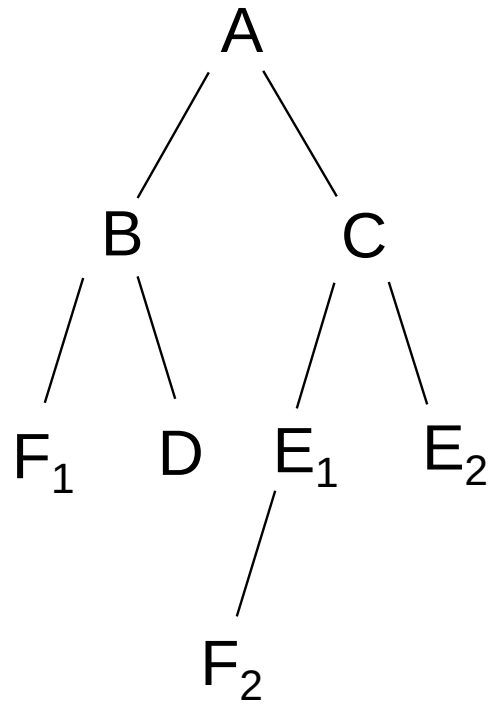
Binary Space Partitioning





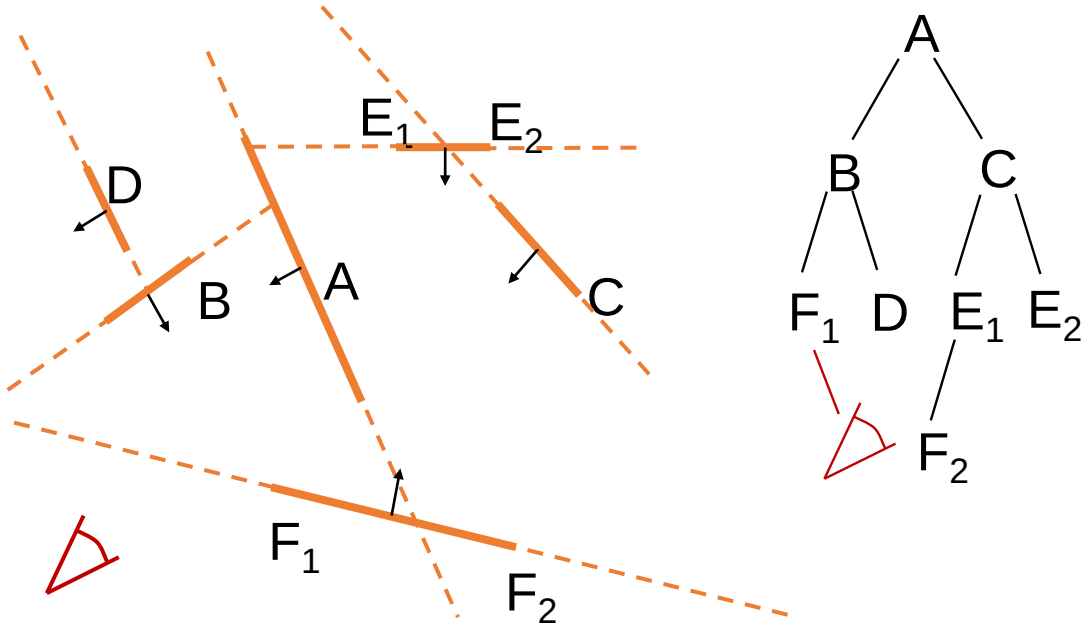
Binary Space Partitioning





BSP: Reihenfolge für Maleralgorithmus

- Sortierte Liste von Polygonen durch Traversierung erstellen
- Identifiziere dazu Halbraum **H_{v*}** in dem Augpunkt *nicht* liegt
- (Inorder-) Traversierungs-Reihenfolge:
  1. H_{v*}
  2. Knoten (Polygon)
  3. H_v
- Lsg. im Bsp.: E_2, C, E_1, F_2, A, D, B, F_1
- Problem
  - Berechnung der Hierarchien aufwendig
  - Daher nur für statische Szenen geeignet
  - Ungeeignet für sehr dynamische Szenen



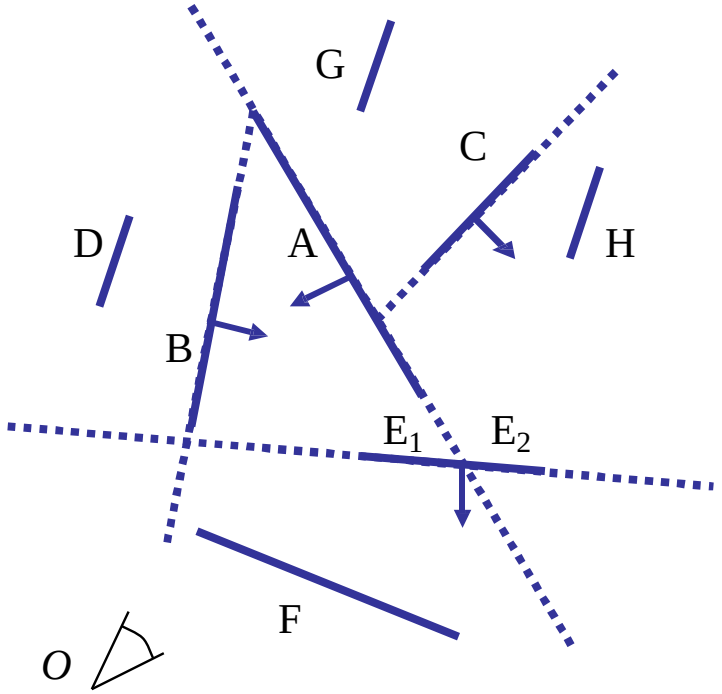
Zusammenfassung BSP-Baum

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

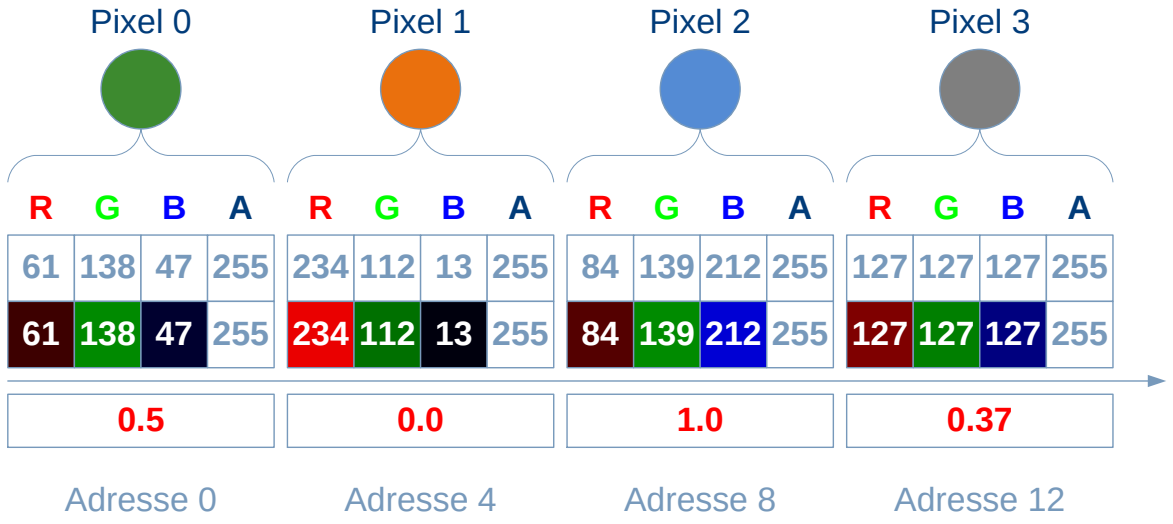
- Binärer Suchbaum, der aus Polygonliste (Polygon A, Polygon B usw.) erstellt wird
  - Aufbau eines solchen Baumes recht teuer, daher nur für statische Szenen geeignet
  - Einfügekriterium nicht "<" wie bei Zahlenfolge, sondern ob neu einzufügendes Polygon vor oder hinter dem Polygon liegt, das im betrachteten Knoten abgespeichert ist
  - Richtung ergibt sich aus Normalenrichtung, wobei jedes Polygon den Raum durch die Ebene, die es aufspannt, immer weiter unterteilt
- Traversiert man Baum in-order von links nach rechts (Schema left - node - right), erhält man aufsteigend sortierte Folge
  - Maleralgorithmus löst Verdeckungsproblem jedoch dadurch, dass man (vom Betrachter aus) von hinten nach vorne sortiert (d.h. back-to-front)
  - Befindet sich Betrachter / Kamera bezogen auf Wurzel des Baumes im vorderen Bereich, sortiert man stattdessen absteigend, weshalb man von hinten anfängt zu traversieren (right - node - left)
  - Befindet sich Betrachter mitten in Szene, kann man Kamera (nur gedacht) an entsprechende Position im Knoten hängen und Traversieren abbrechen, ist man an diesem Knoten angelangt

Übung 1

- Beschreiben Sie die Sichtbarkeit der Kanten *A* bis *H* (back-to-front) in Bezug auf den Betrachter *O* mit Hilfe eines BSP-Baums!

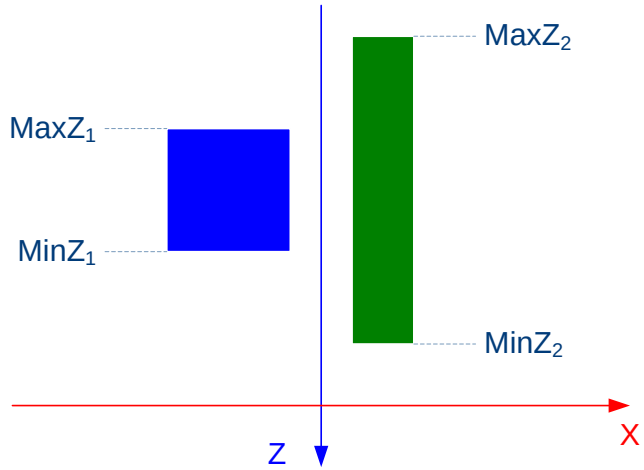


Der Tiefenpuffer



Sortierprobleme revisited

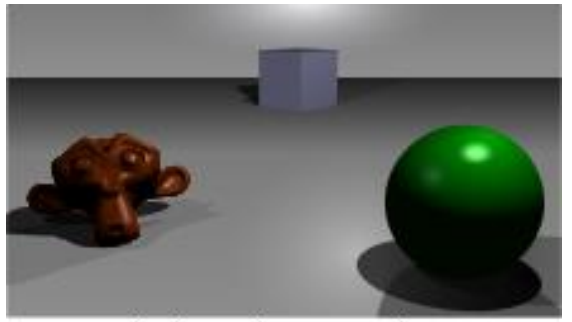
- Unterschiedliche Form, gleicher Mittelpunkt
- Was ist der richtige Bezugspunkt für Sortierung?
  - Bei Wahl des minimalen z-Wertes liegt Blau hinter Grün
  - Bei Wahl des maximalen z-Wertes liegt Blau vor Grün
  - Bei Wahl des mittleren z-Wertes nicht entscheidbar
- Sortierung führt zu nicht eindeutigen Ergebnissen
  - Erzeugt damit Darstellungsfehler
  - Korrektur durch Unterteilung der Objekte und Polygone
  - → Verfahren aufwendig
  - Gesucht: modernerer Ansatz!



Punktorientierte Algorithmen

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Problem bisher
  - Linien- und Flächentests liefern Sichtbarkeit nur implizit
  - Durch wenige Punkttests und Ausnutzung von Kohärenz
- Lösung
  - Untersuchung einzelner Bildpunkte auf Verdeckung bzw. Sichtbarkeit
- Einfachste Art der Verdeckungsrechnung
  - Punkttests liefern Sichtbarkeit explizit
  - Größte Anzahl von einfachen Tests
- Punktorientierte Verfahren zweckmäßig für Rasterdisplays
  - Bildraumverfahren: Testelement und Ausgabeelement stimmen überein
  - Geräteabhängig: Genauigkeit entspricht Auflösung des Ausgabegeräts



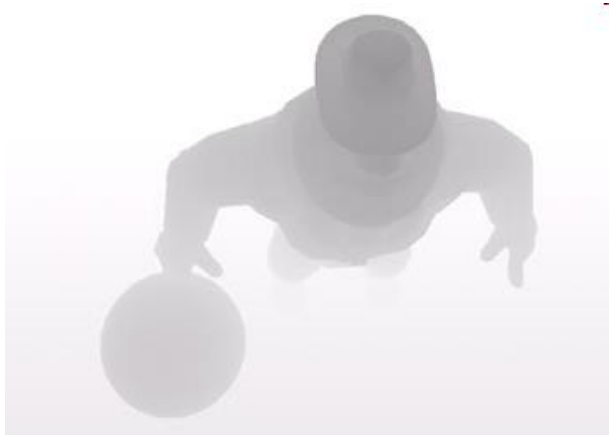
A simple three-dimensional scene

Z-buffer representation

Tiefenpuffer (Depth Buffer)

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Für jeden Bildpunkt wird auch z-Wert gespeichert: Tiefenbild
  - Bereits '74 vorgeschlagen, aber aus Kostengründen damals nicht realisiert
  - Ist weiterer Puffer mit einem Tiefenwert pro Pixel
  - Tiefenwerte entsprechen z-Wert im Eye Space (z-Buffer)
- Initialisierung
  - Bildspeicher mit Hintergrundfarbe füllen (z.B. Schwarz)
    - Farbwerte bilden zusammenhängendes Array (pro Pixel 4 Byte für RGBA)
  - Tiefenspeicher mit maximalem z-Wert (1.0 entspricht max. Distanz)
    - $float$ mit Wertebereich von 0.0 (Near Plane) bis 1.0 (Far Plane)
    - Früher bzw. mobile Geräte 16 – 24 Bit, heute bis zu 32 Bit
- Alle Objekte der Szene werden nacheinander mit Tiefe gerastert
  - Besondere Reihenfolge nicht erforderlich ☺



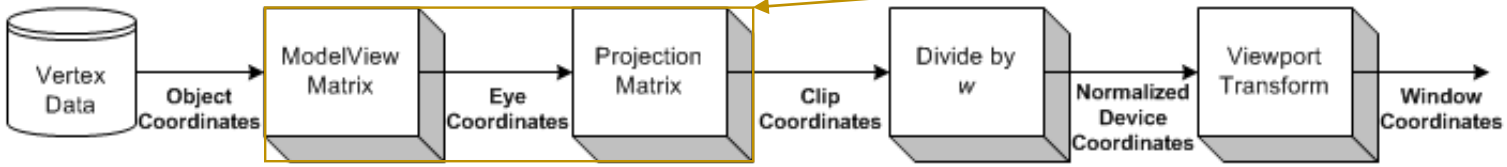
Algorithmus

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

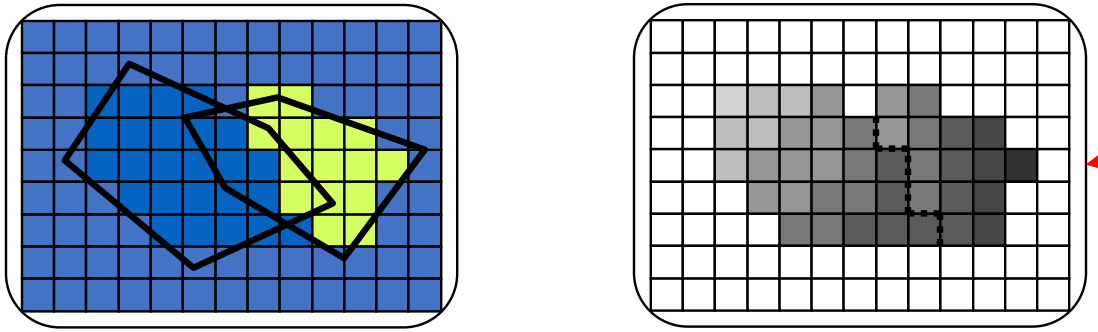
- Szene wird zunächst perspektivisch transformiert
  - Perspektivische Abbildung erhält Tiefe (→ keine reine Projektion)
  - Tiefenberechnung erfolgt automatisch bei Transformation in (4D) Clip Space bzw. (3D) NDC
- Bestimmung des sichtbaren Punktes ist Tiefenvergleich
  - Bei Projektion auf xy-Ebene müssen nur z-Werte verglichen werden
  - Nach Behandlung aller Objekte steht im Framebuffer gewünschtes Bild der sichtbaren Flächen
- Für jeden Bildpunkt (x, y) eines darzustellenden Dreiecks tue:
  1. Berechne Tiefe $z(x, y)$
  2. Neue Tiefe $z(x, y)$ kleiner als der unter (x, y) gespeicherte Wert?
     - Schreibe Tiefe $z(x, y)$ an Stelle (x, y) in z-Buffer
     - Schreibe zugehörige Farbe c an Stelle (x, y) im Bildspeicher

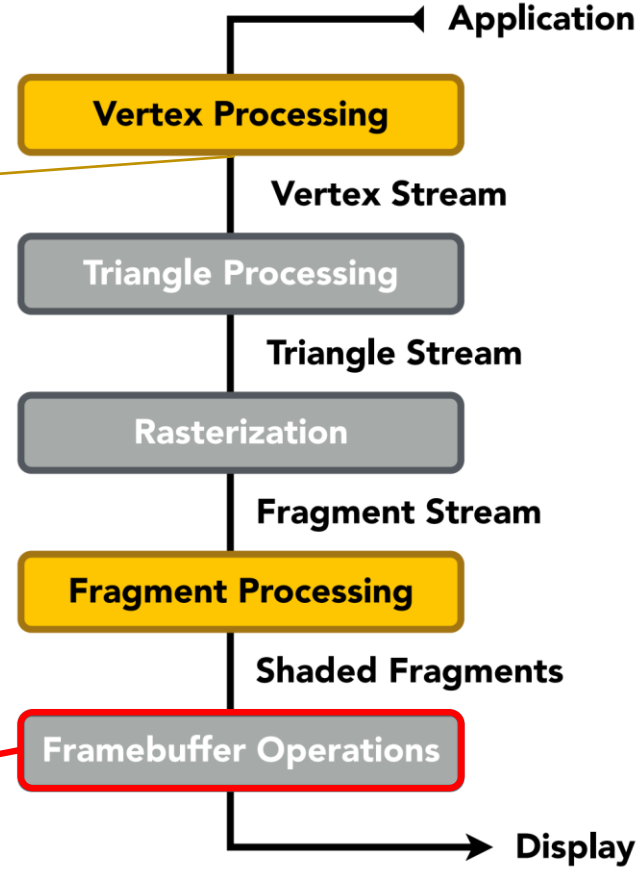
Einordnung in 3D-Graphik-Pipeline

- Jeder Vertex durchläuft in Graphik-Pipeline verschiedene Koordinatentransformationen
  - Fragmente bei Rasterisierung durch Interpolation über Fläche



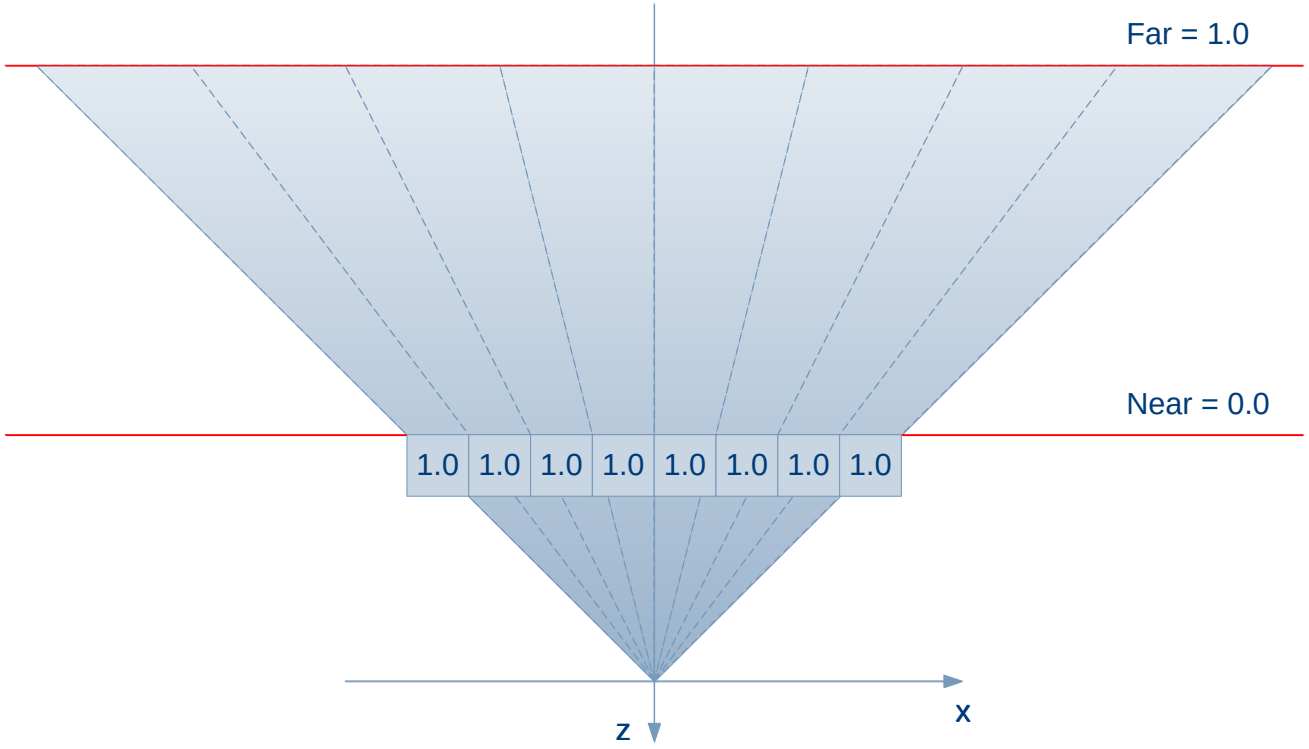
- Zu jedem Pixel wird aktuelle Entfernung zu Auge gespeichert
  - Nur wenn Tiefe des aktuellen Pixels kleiner als in Depth Buffer gespeicherte ist, wird es geschrieben





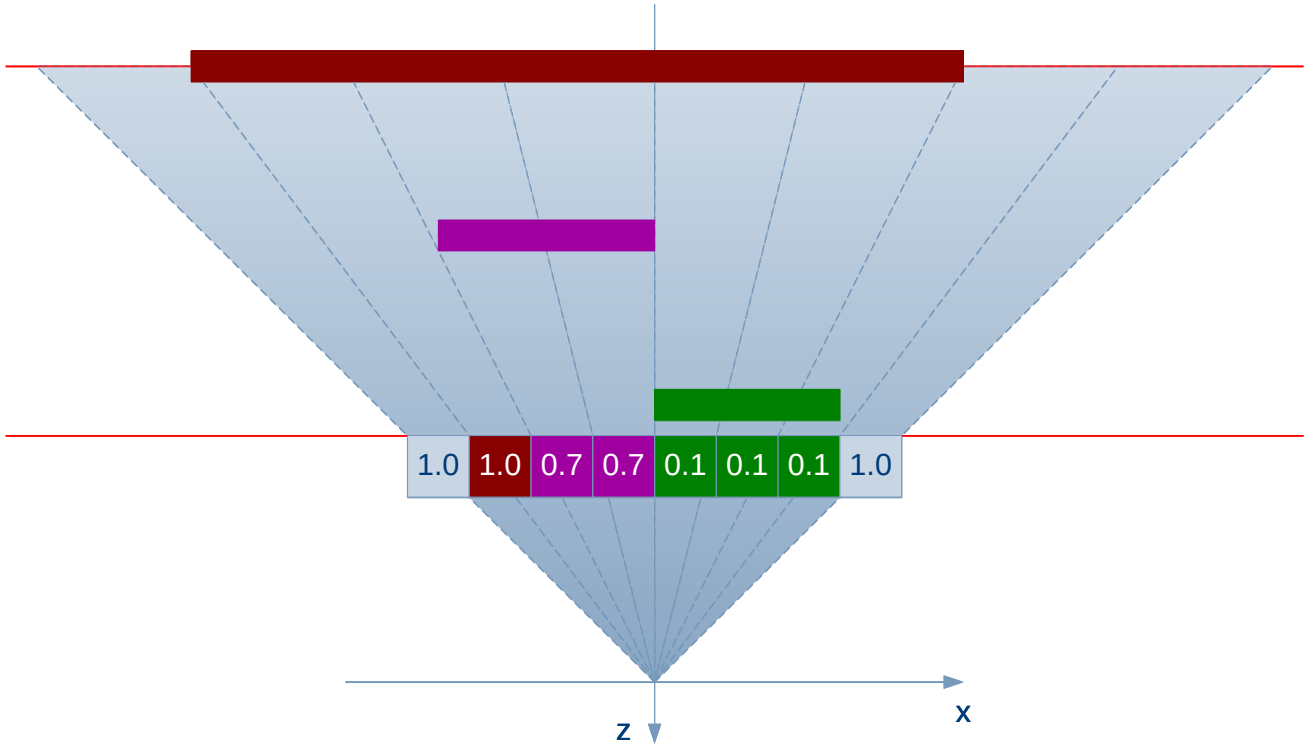
Beispiel

- Betrachte Querschnitt Viewfrustum in *xz*-Ebene



Beispiel

- Objekte nacheinander einfügen bzw. darstellen



Verdeckungsrechnung aktivieren

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Beispiel WebGL (OpenGL im Browser mit JavaScript → OpenGL analog)
  - Initialisierung:

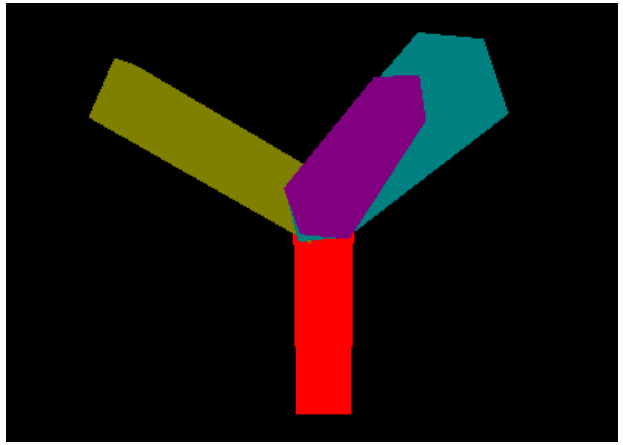
```
gl = canvas.getContext('webgl', {alpha:true, depth:true});
```

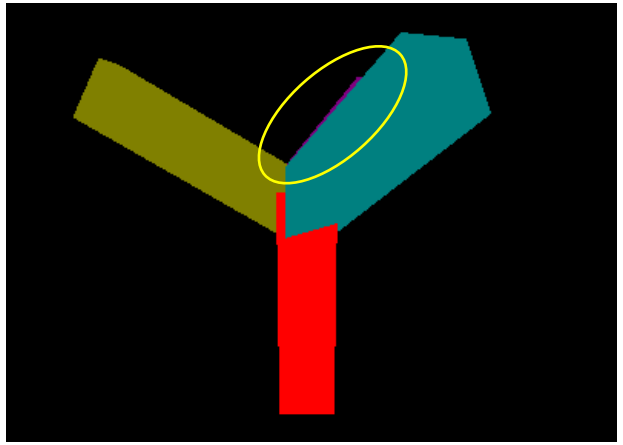
  - Setzen des sog. Tiefenbits:

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

  - Tiefentest einschalten und Vergleichs-Funktion wählen (z.B. LESS, LEQUAL):

```
gl.enable(gl.DEPTH_TEST);    gl.depthFunc(gl.LEQUAL);
```





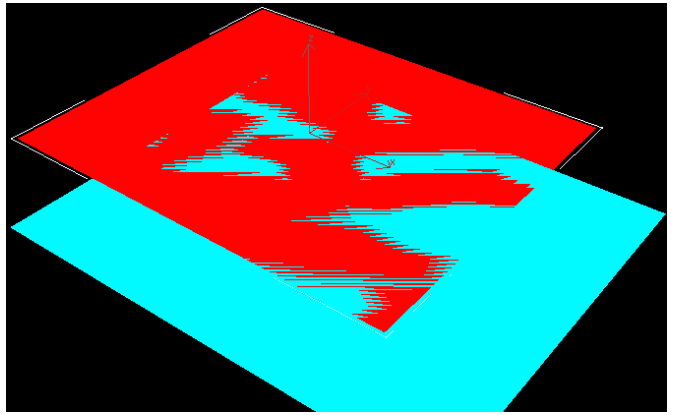
Vorteile

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Skaliert sehr gut: jede 3D-Szene mit jeder Art von Objekten kann behandelt werden
  - Komplexität unabhängig von Tiefenkomplexität
  - Speicherung von z nur für sichtbaren Bildpunkt
  - Suche des minimalen z-Wertes pro Pixel
- In fertige Szene können sogar nachträglich Objekte eingefügt werden
  - Spezielle Zusatzobjekte in 3D-Szene mit korrekter Verdeckung darstellbar
    - Z.B. 3D-Cursor in VR, Annotationen usw.
  - Auch für Kombination mit Kameraaufnahmen interessant
- Sehr leicht in Hardware realisierbar

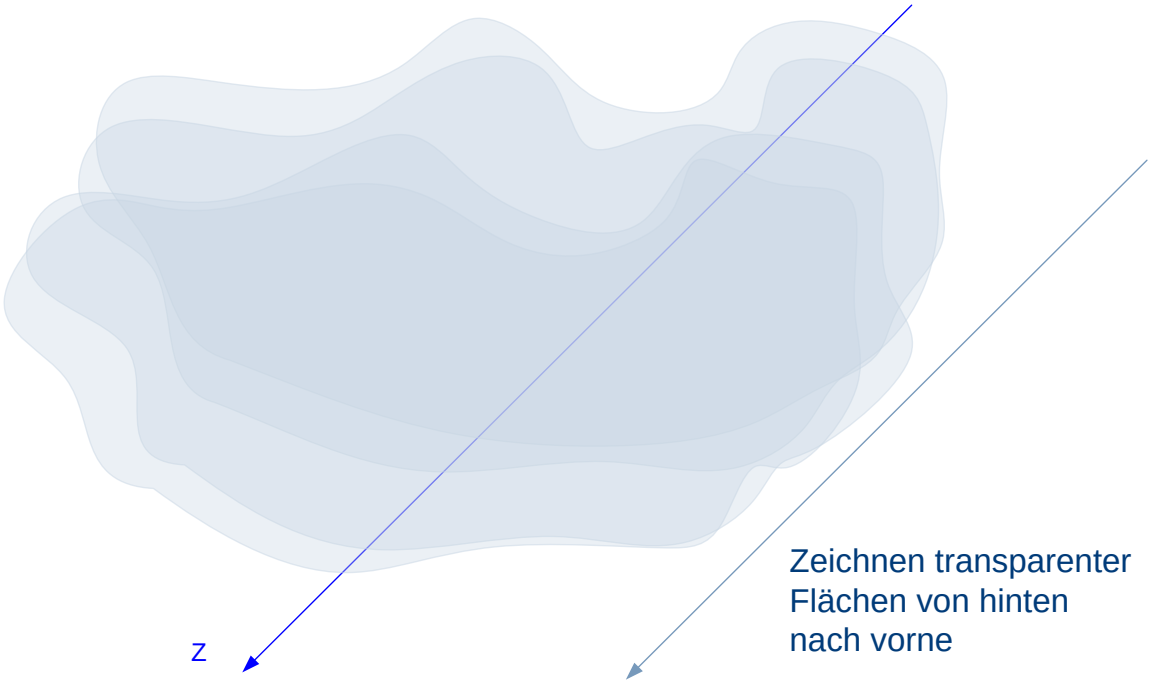
Nachteile

- Pro Bildpunkt wird nur ein Objekt gespeichert
  - Abtastfehler, die durch höhere Auflösung verkleinert aber nicht beseitigt werden
- Genauigkeit des Tiefenpuffers beschränkt
  - Verschiedene Objekte erhalten gleichen z-Wert
    - Mangelnde Genauigkeit führt zu „z-Fighting“ → Near/Far Clipping Plane passend wählen
  - Geringere Genauigkeit mit wachsender Entfernung
    - Tiefenwerte nicht linear verteilt
  - Farbe von Objektreihenfolge bei Rasterisierung bestimmt
    - Probleme bei Animationen
- Transparenzen schwierig ☹



Transparente Flächen

- In transparente und nicht-transparente Objekte trennen
  - Zeichnen der transparenten Objekte erst am Ende
  - Sortierung entlang der z-Achse, dann Zeichnen von hinten nach vorne



Übung 2

- Gegeben seien die drei Objekte A, B und C
- Bestimmen Sie die Sichtbarkeit mit Hilfe des z-Buffer-Algorithmus‘ (Einfüge-Reihenfolge sei A-B-C)
  - Nutzen Sie die Tiefenvergleichsfunktion „LEQUAL“ (≤)
  - Welches Ergebnis würde alternativ „LESS“ (<) liefern?

A

1	1	1	1	1	1
1	1	1	1	1	1
1	1	.5	1	1	1
1	1	.5	.5	1	1
1	1	.5	.5	.5	1
1	1	1	1	1	1

B

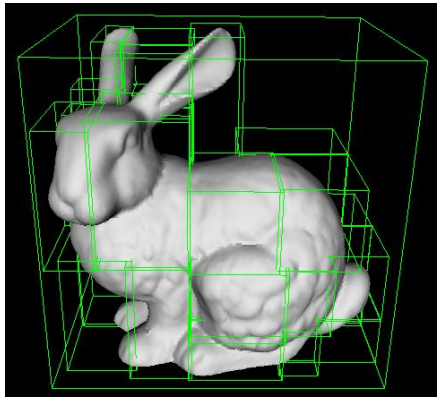
1	1	1	1	1	1
1	.7	.6	.2	.3	1
1	.6	.2	.3	1	1
1	.2	.3	1	1	1
1	.3	1	1	1	1
1	1	1	1	1	1

C

1	1	1	1	1	1
1	1	1	1	1	1
1	1	.2	.2	.2	1
1	1	.2	.2	1	1
1	1	1	1	1	1
1	1	1	1	1	1

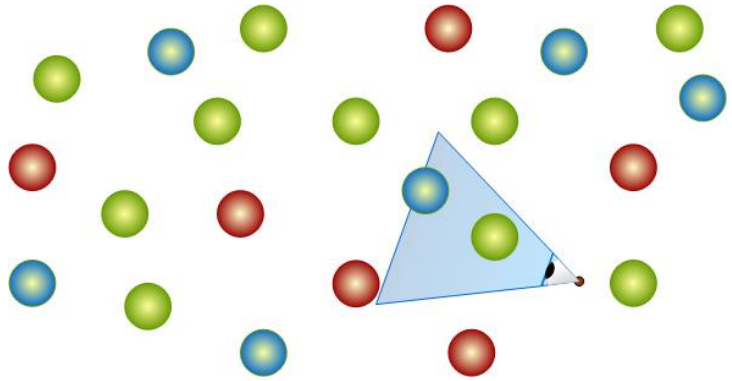
h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Räumliche Datenstrukturen

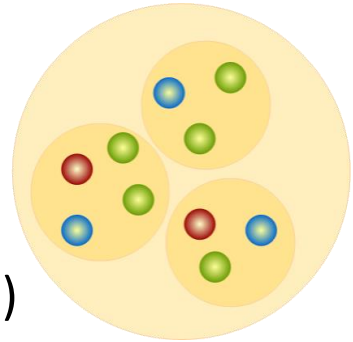


Verwaltung großer Objektmengen

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

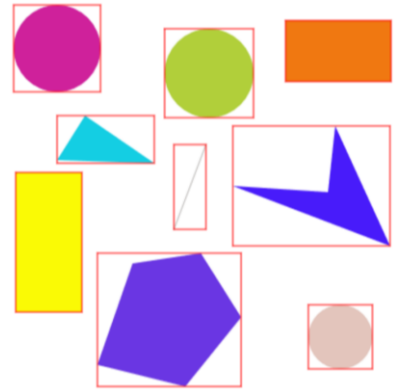


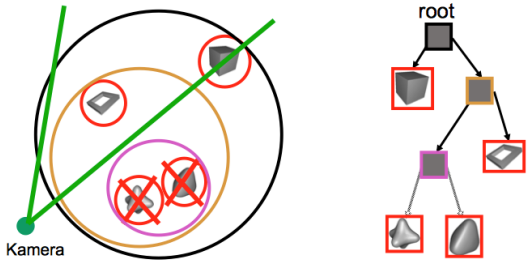
- Hauptproblem großer Datenmengen ist Suche
  - Schnelle Objektselektion?
  - Nächster Nachbar?
  - Alle Objekte innerhalb eines Bereichs?
- Typischer Anwendungsfall: Viewfrustum Culling
  - Welche Objekte liegen innerhalb Sichtpyramide?
- Hierarchische Datenstrukturen reduzieren meist Anzahl der Tests
  - Lösungsansatz also über Gruppierung von Objekten
  - Szenengraph hier gut nutzbar
    - Für jeden Teilgraphen Bounding Volume bestimmen (und Frustum dagegen testen)



Problemstellung

- Räumliche Suchanfragen müssen schnell sein und sehr viele Objekte testen
  - Wichtig für Culling, Picking, Collision Detection etc.
  - Sollte in sublinearer Zeitkomplexität erfolgen
- Lösung:
  - Schnelles Verwerfen
    - Test vereinfachen durch grobe Vorauswahl
      - Umgebe dazu Objekte mit einfachem Hüllkörper, z.B. Bounding Box
    - Exaktere Untersuchung nur für Objekte, die Hüllkörper-Test bestehen
      - Bei linearer Organisation der Daten aber immer noch *n* Tests
  - Divide and Conquer
    - Verwalte graphische Objekte in Baumstruktur
    - Verwerfe komplette Teilbäume mittels einfacher Tests

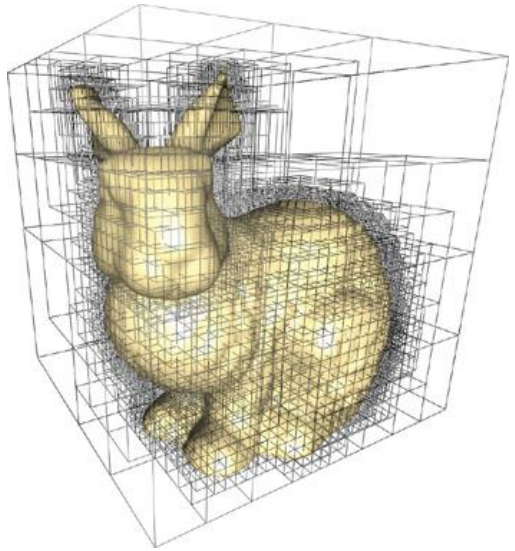


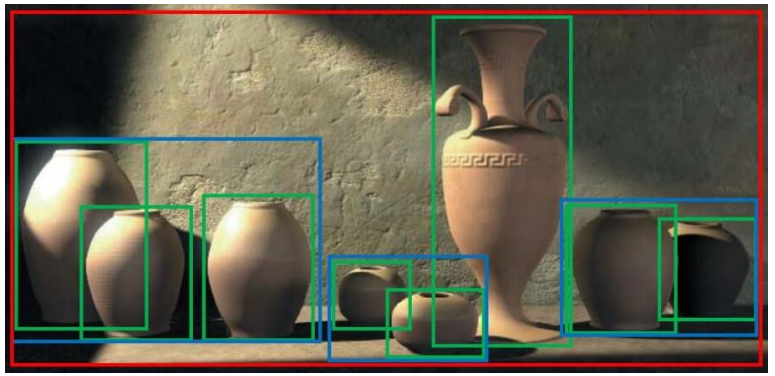


Räumliche Datenstrukturen

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

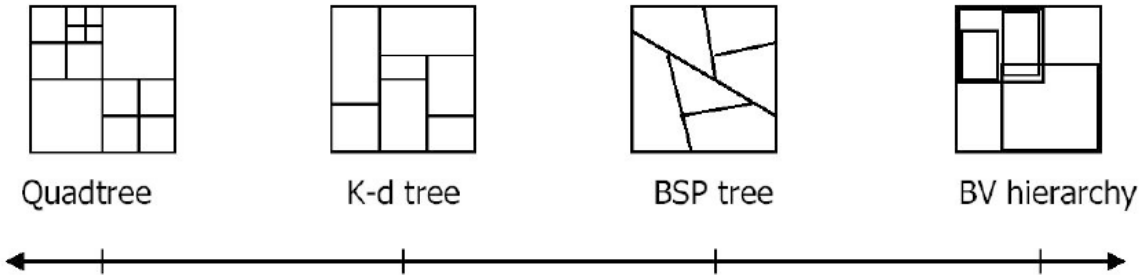
- Zwei Kategorien (Mischformen möglich)
  - Raumunterteilungsschemata (Space Partitioning)
    - Unterteilen Raum in disjunkte Mengen/Zellen
    - Jeder Punkt im Raum gehört zu genau einer Zelle
    - Ein Objekt kann sich aber in mehreren Zellen befinden
  - Objektunterteilungsschemata (Object Partitioning)
    - Unterteilen Objekte in disjunkte Mengen/Teile
    - Jeder Punkt im Raum kann zu mehreren Teilen gehören
    - Aber ein Teilobjekt gehört zu genau einem Teil
- Performance Trade-Offs verschiedener Strukturen
  - Speicher- vs. Zeitverbrauch
    - Zeit für Aufbau der Datenstruktur, Aktualisierung, Abfragen



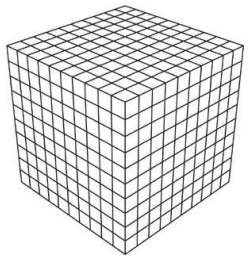


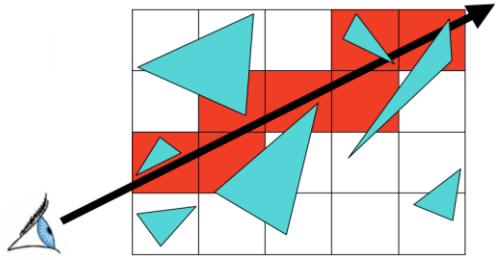
Raum- und Objektunterteilung

- Objektunterteilung
  - Hüllkörperhierarchie
    - Z.B. Bounding Box Hierarchie
- Raumunterteilung
  - Hierarchisch
    - Uniform: Quad-/Octree
    - k-d-Baum
    - BSP-Baum
  - Gitter
    - Rectilinear (Sonderfall regulär)
    - Curvilinear (d.h. gekrümmt)
    - Unstrukturiert



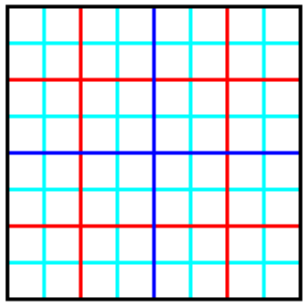
- k-d-Bäume erweitern Konzept binärer Suchbäume
  - Auf mehrere Dimensionen durch k-dimensionale Schlüssel
  - Unterteilungsebenen/-geraden sind an Achsen ausgerichtet
- Teste, ob Punkt in linkem oder in rechtem Teilbaum liegt
  - Abwechselnd z.B. mit x-, y- und z-Koordinaten vergleichen

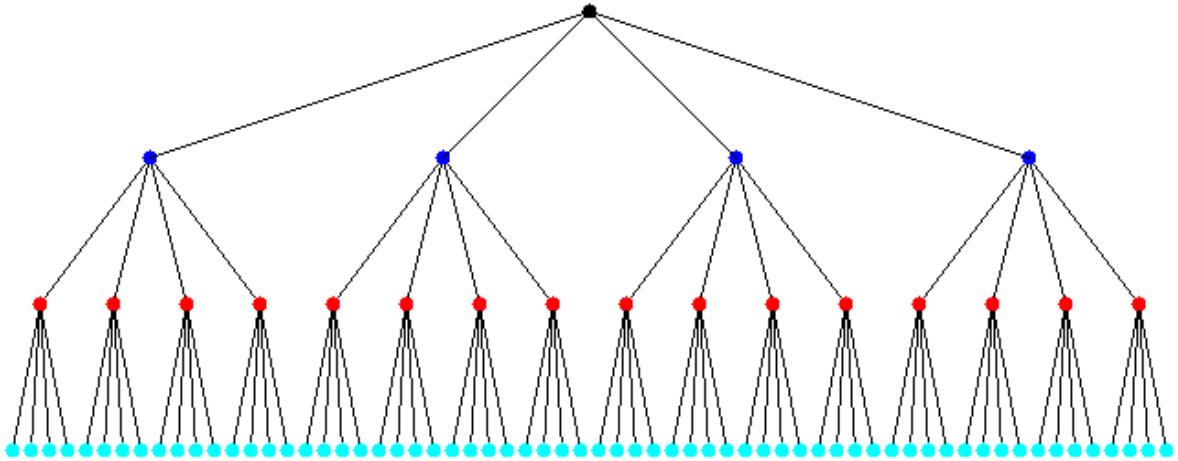




Quadtree

- Baumstruktur (hierarchische Datenstruktur zur Raumunterteilung)
  - Knoten enthalten Daten und weitere Knoten
  - Beispiel:
    Vollständiger Quadtree mit 4 Leveln:





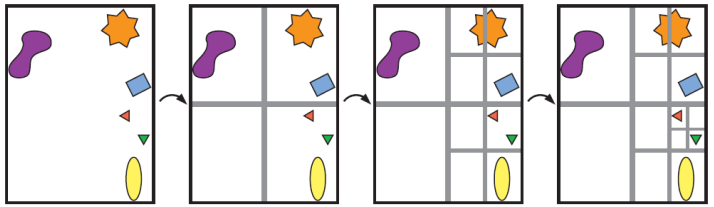
- Jeder Knoten umfasst rechteckigen Bereich
  - Besitzt max. vier Kindknoten, ggfs. Parent, sowie (i.d.R. nur) ein Datenelement
  - Kind-Bereiche entstehen durch Halbieren von jeweils Breite und Höhe

Aufbau

- Rekursive Struktur

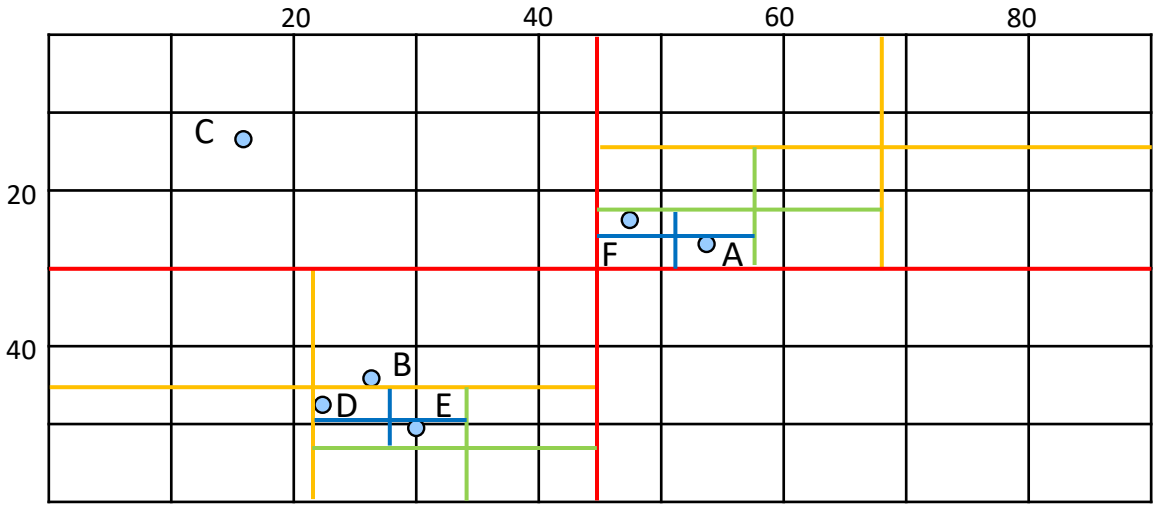
```
Quadtree {
   // Attribute:
   Liste mit Szenenobjekten
   Children (max. 4 Quadtree Nodes)
   Verweis auf Parent
   BoundingBox (Region in Ebene)

   // Wichtige Methoden:
   Konstruktor
   insert(obj);
   find(x, y);
}
```

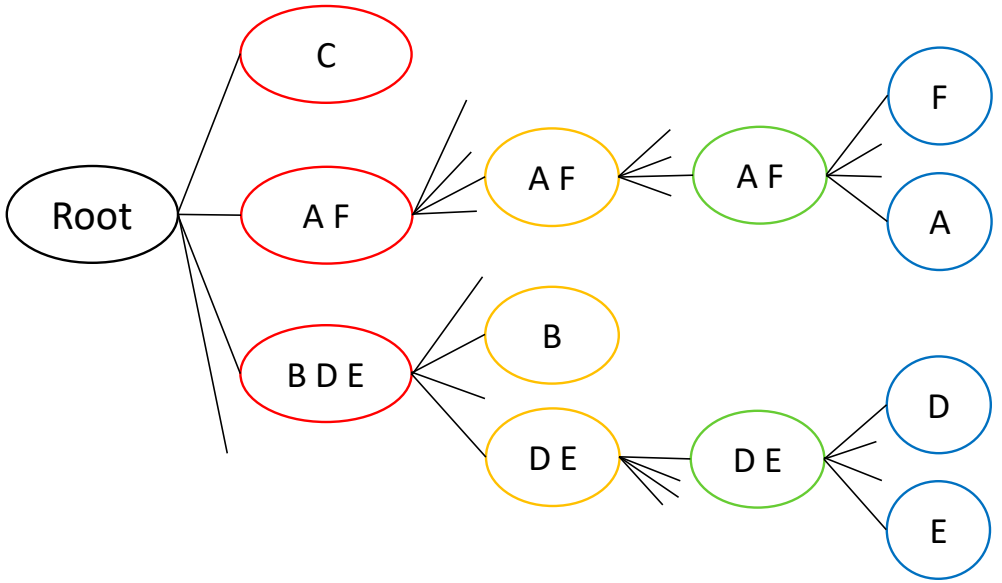


- Baum für 2D-Bereichssuche
- Innere Knoten
  - Haben 4 Kinder (können leer sein), die je einen Quadranten der Region ihres Parents darstellen
  - Halten i.d.R. keine Daten
- Blattknoten
  - Halten i.d.R. einen einzelnen Datenwert (nur z.B. bei Point-Region Quadtree)
  - Bereich wird beim Einfügen unterteilt, so dass nicht mehr als ein Punkt existiert

Beispiel

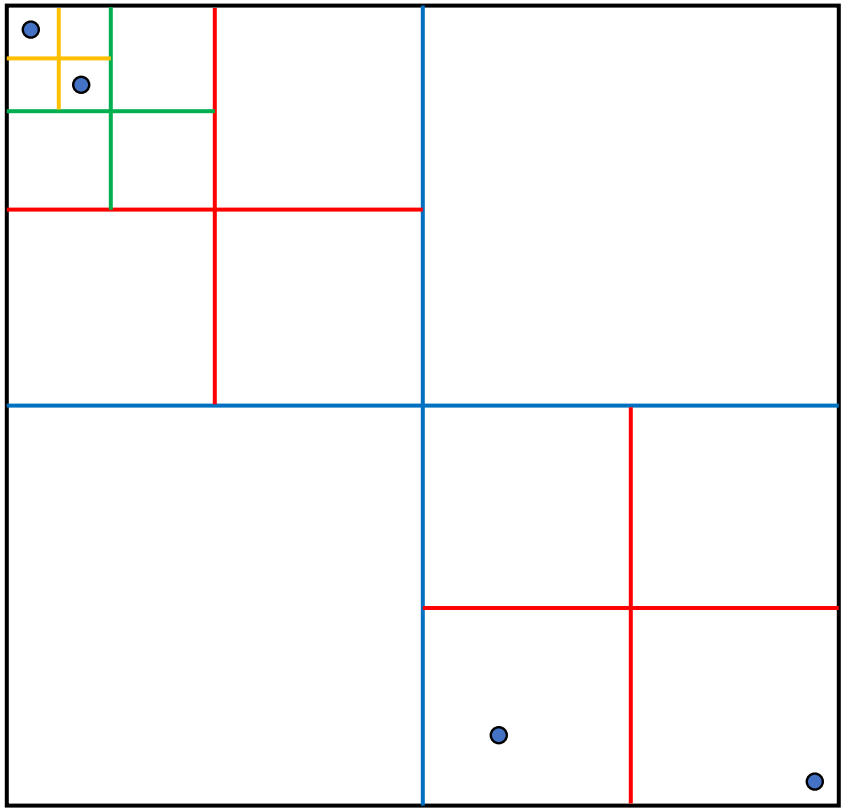


0	1
2	3

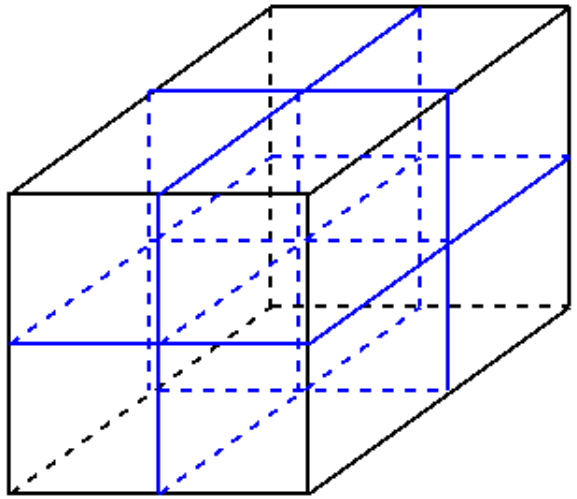


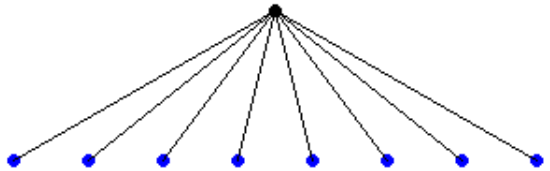
Probleme

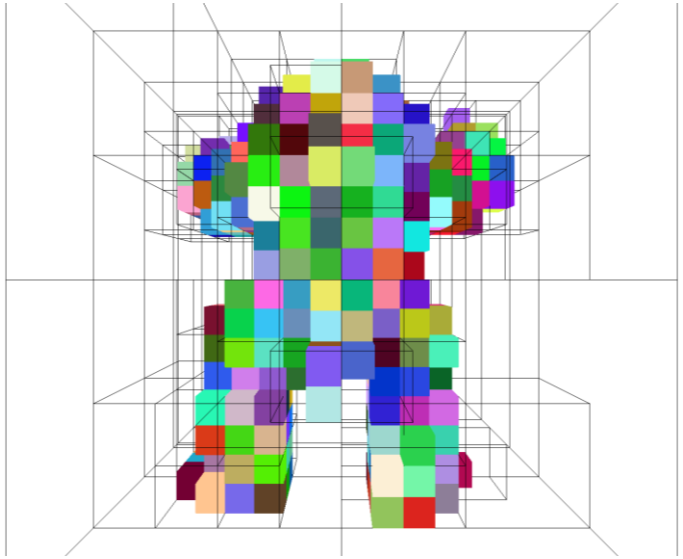
- Baum nicht balanciert, falls Objekte nicht gleichmäßig verteilt
  - Zu viele unnötige Unterteilungen
    - DS passt nicht zu Daten
  - Verschlechtert Performance
- Lösungsmöglichkeiten
  - Nicht gleichmäßig/ regulär unterteilen
  - Loose Quadtree



Octree







- Wie Quadtree, aber in 3D
  - Hat damit je drei Unterteilungsebenen (entlang x, y, z)
  - Jeder Knoten umfasst quaderförmigen Bereich
  - Besitzt max. 8 Kindknoten, ggfs. Parentpointer

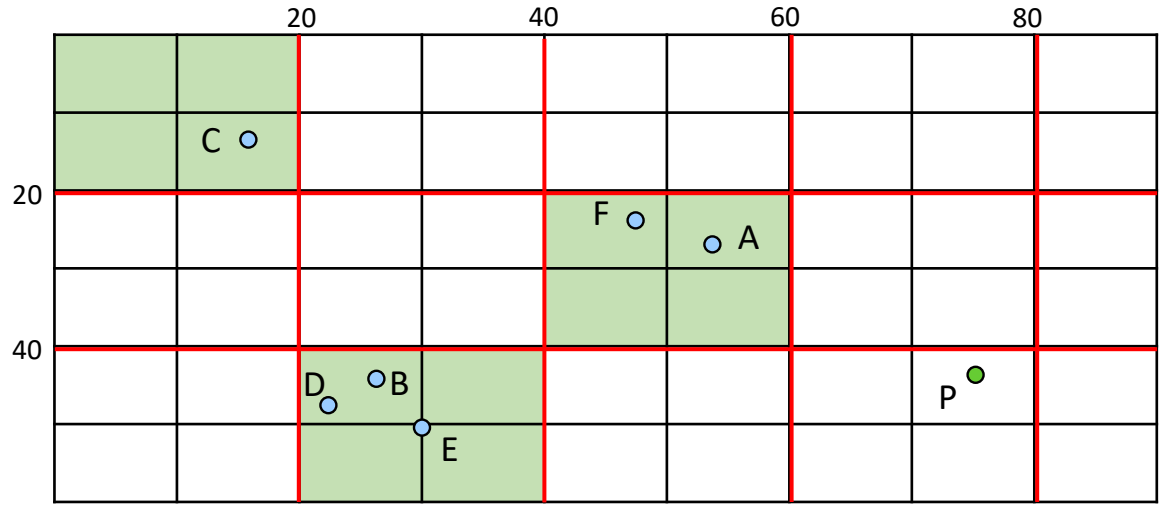
Reguläres Gitter (Grid)

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

- Berechnung von Hierarchien leider oft recht aufwendig, damit ungeeignet für massiv dynamische Szenarien
- Reguläres Gitter ist einfache Datenstruktur zur Raumunterteilung
  - Unterteilt Raum in achsparallele rechtwinklige Bereiche, wobei Kanten entlang einer Achse immer gleiche Länge haben (beim Spezialfall „kartesisches Gitter“ sind alle Kantenlängen gleich)
  - Keine Berechnung einer Hierarchie erforderlich
- Zur Einordnung graphischer Objekten in eine Region (über Bezugspunkte wie Mittel- oder Schwerpunkt), um dynamische Objekte einfach verwalten und suchen zu können
  - Unterteilt Raum in disjunkte Zellen, wobei sich ein Objekt über mehrere Zellen erstrecken kann
  - Gitter ist effizient traversierbare DS mit konstantem Suchaufwand über 2D- (oder 3D-) Position (die unmittelbar Index einer Gitterzelle ergibt), und ermöglicht schnellen Zugriff auf Nachbarn
  - Nachteile: Zelle ist kleinste Einheit (daher lineare Suche der Elemente innerhalb einer Region), speicheraufwendig, kann sich der Geometrie nicht anpassen

Reguläres Gitter (Grid)

- Bsp., Verwaltung von *n* 2D-Punkten: A(54,26), B(26,44), C(15,12), D(22,48), E(30,50), F(48,24)

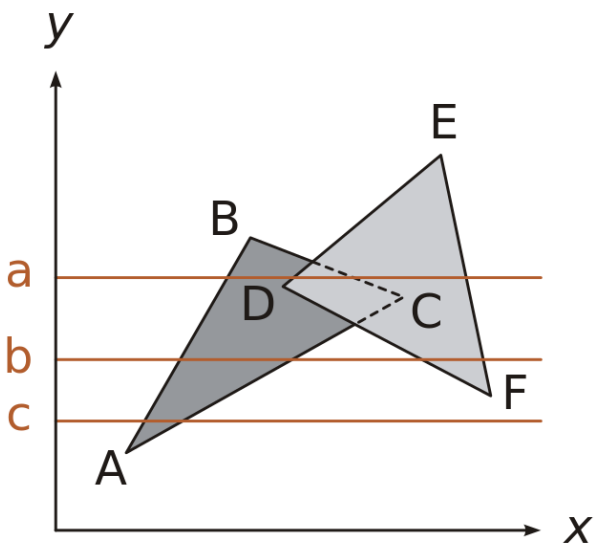


Zellengröße alternativ aus gewünschter Anzahl der Gitterzellen bestimmen (bei 5 x 3 Grid wäre Zellengröße 18 x 20 px)

- Region (*w*=90, *h*=60) mit achsparallelem Gitter überlagern (Zellenbreite: g_x=20, Höhe: g_y=20)
  - 15 Zellen, davon 3 belegt (→ verweisen auf Objekte oder NULL)
- Ges.: Position von P (75, 43) im Gitter (→ Indizes über Teilen durch Zellengröße g_x bzw. g_y)
  - Zellennr. in X-Richtung: 75 / 20 = 3, in Y-Richtung: 43 / 20 = 2

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Vielen Dank!



Noch Fragen?