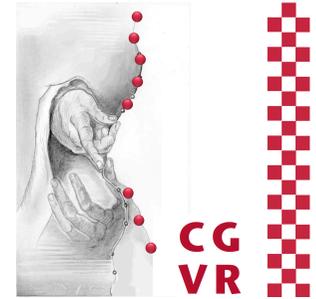
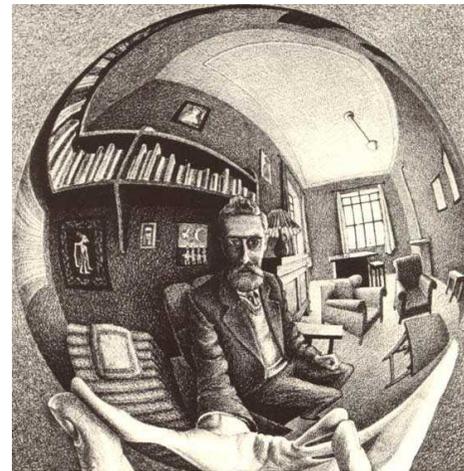


Bremen



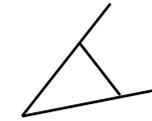
Virtuelle Realität Real-time Rendering



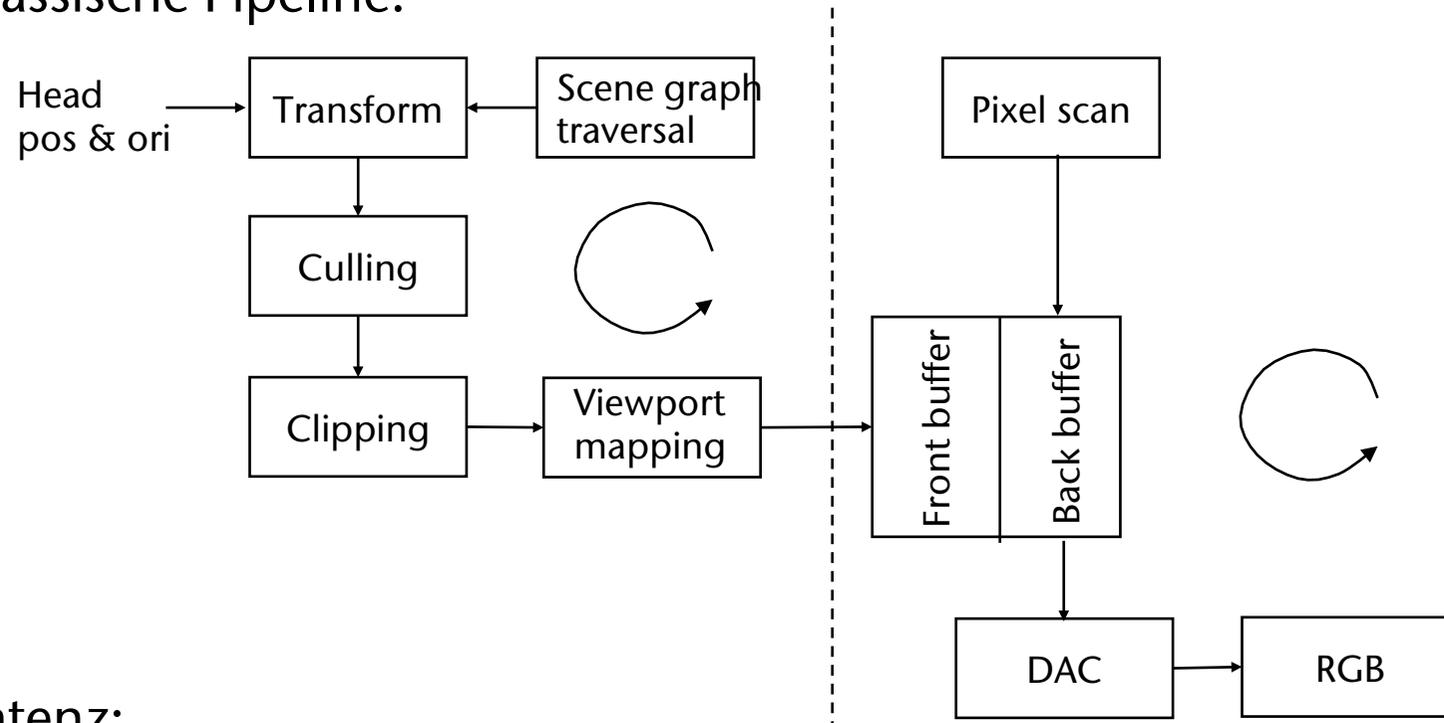
G. Zachmann

University of Bremen, Germany

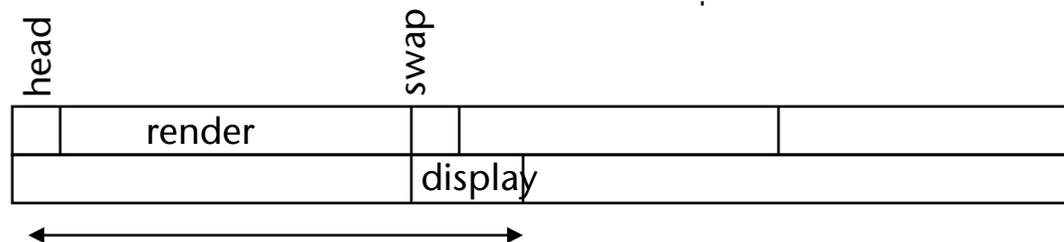
cgvr.cs.uni-bremen.de



- Klassische Pipeline:

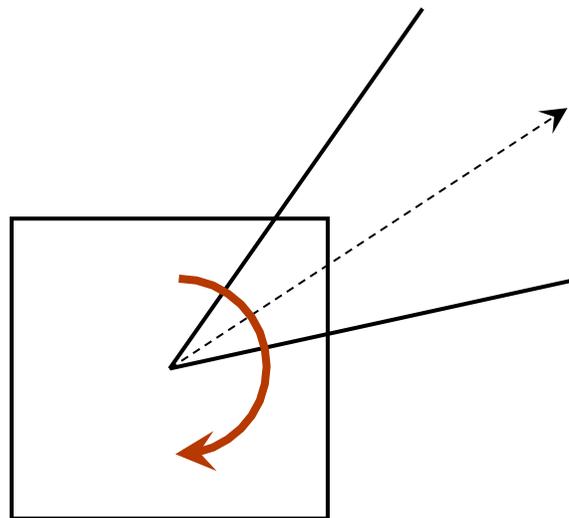


- Latenz:

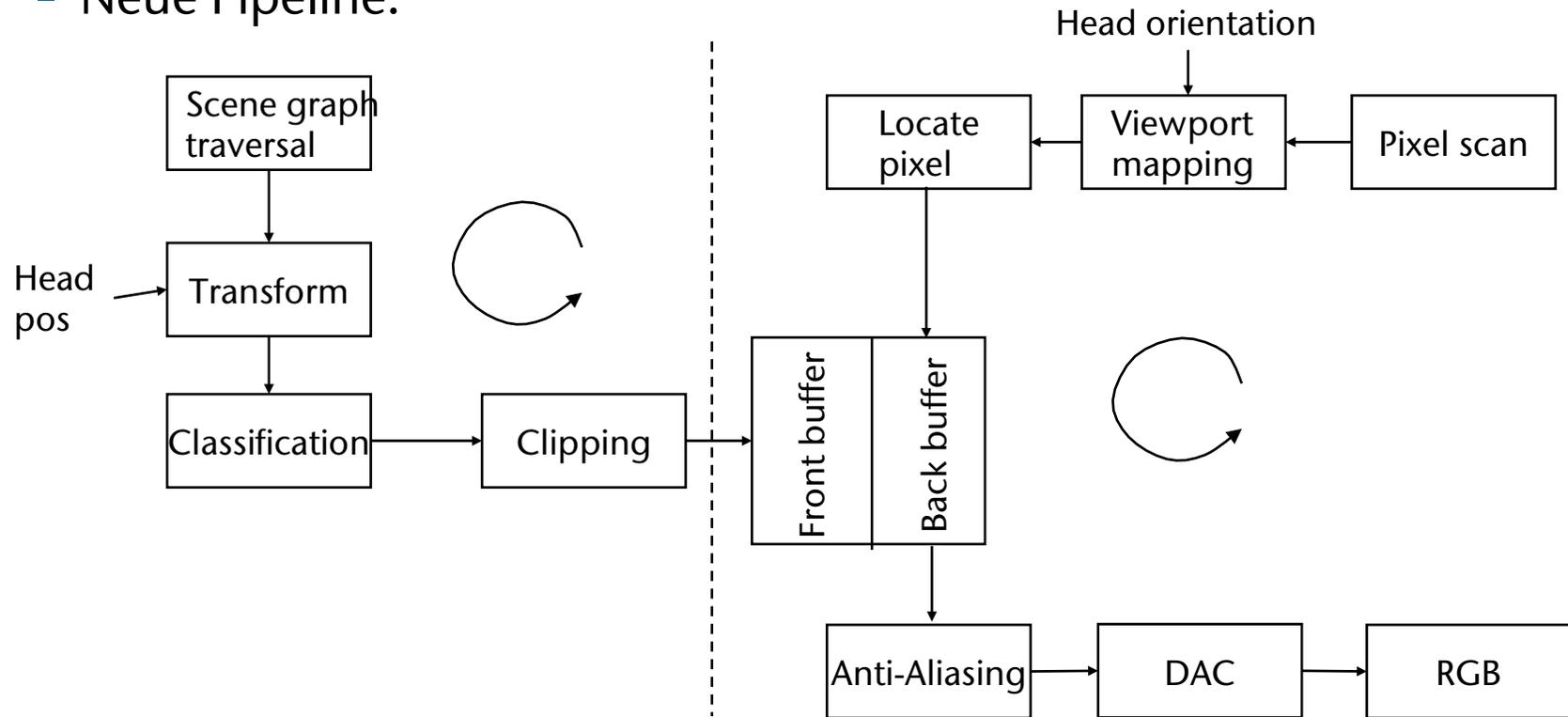


- Idee: rendere mehr als den Viewport

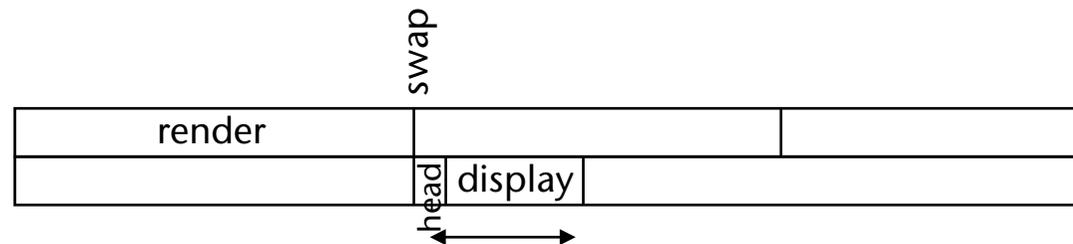
- Konzeptionelle Idee:
 - Rendere Szene auf Kugel um Betrachter
 - Bei Rotation des Viewpoints: Ausschnitt des Viewports neu wählen
 - Wähle statt Kugel einen Würfel (s.a. Cave)



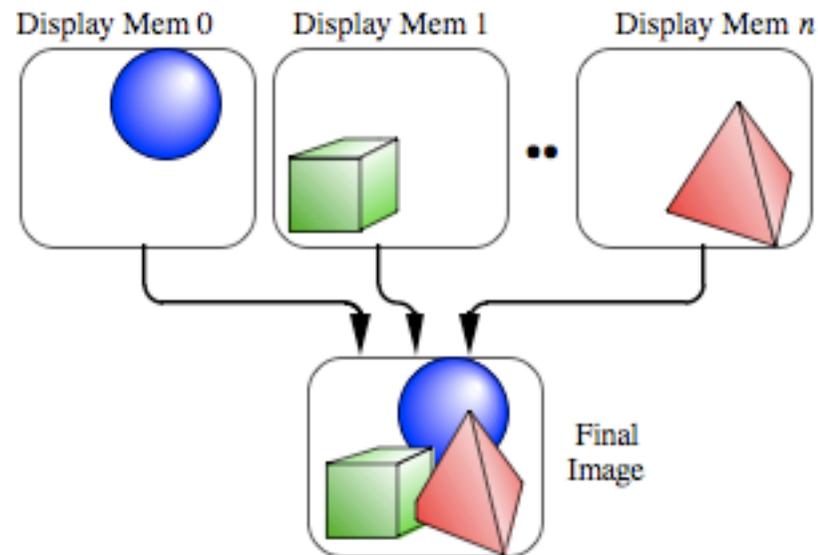
■ Neue Pipeline:



■ Latenz:

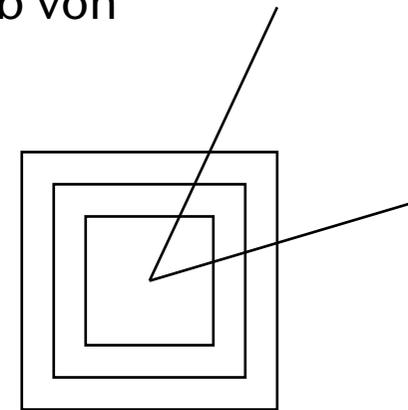


- Konzeptionelle Idee:
 - Jeder Thread rendert nur "sein" Objekt in einen eigenen Frame-Buffer
 - *Video-Hardware* liest *Frame-Buffer* inkl. Z-Buffer aus
 - Image Compositor kombiniert die Einzelbilder durch Z-Vergleich pro Pixel
 - In praxi: teile Menge aller Objekte in der Szene in eine kleine Anzahl von Teilmengen auf; rendere jede Teilmenge durch einen PC mit eigener Grafikkarte



- *Prioritized Rendering:*

- Weit entfernte / langsame Objekte bleiben länger auf dem Bildschirm "gleich"
- Idee: rendere auf mehrere (gedachte) Schalen
- Wieviele Schalen neu gerendert werden, hängt ab von
 - Geforderter Framerate
 - Szenenkomplexität
 - Viewpoint-Bewegung
 - Objekt-Bewegung



- Human factors beeinflussen Priorität:

- Keine Kopfdrehung um 180° → Objekte "hinten" nur selten updaten
- Manipulierte Objekte haben hohe Priorität
- Objekte in der Peripherie seltener updaten

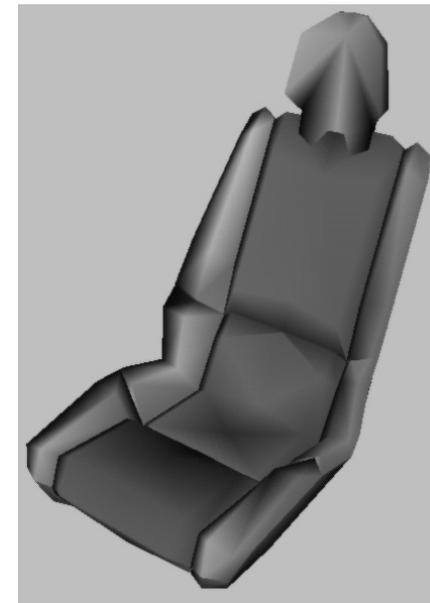
Konstante Framerate durch "Weglassen"

- Gründe für eine konstante Framerate:
 - Vorhersage bei *predictive filtering* des Trackings von Kopf / Hände funktioniert nur, wenn alle nachfolgenden Stufen der Pipeline (d.h., alle Stufen) immer gleich schnell arbeiten
 - Wahrnehmung von Framerate-Sprüngen (z.B. zwischen 60 und 30 Hz)
- Rendering als "*time-critical computing*":
 - Rendering bekommt bestimmtes Zeitbudget (z.B. 17 msec)
 - Rendering-Algorithmus muß Bild produzieren "so gut wie möglich"
- Techniken des "*Weglassens*":
 - *Levels-of-Details (LOD)*
 - Unsichtbare Geometrie weglassen (*Culling*)
 - *Image-based rendering*
 - *Lighting-Model* reduzieren, Texturen reduzieren,
 - ... ?

- Definition:

Ein **Level-of-Detail (LOD)** eines Objektes ist eine **reduzierte Version** (d.h., weniger Polygone) einer (hoch-aufgelösten) Geometrie

- Beispiel:

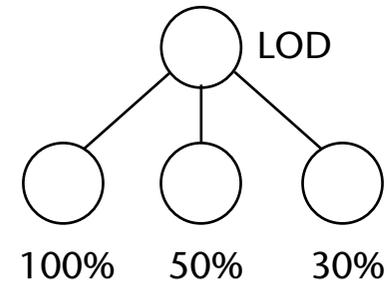


- Idee: verwende den LOD "passend" zur Entfernung vom Viewpoint

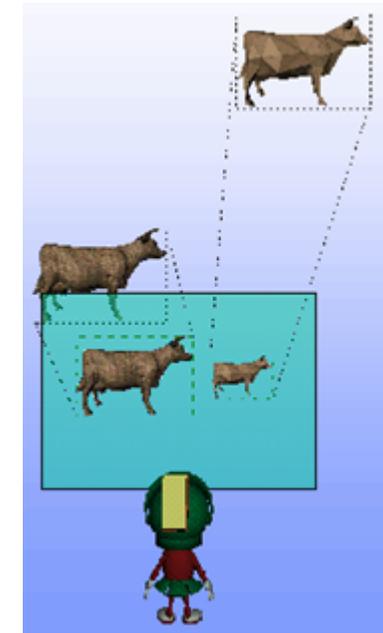
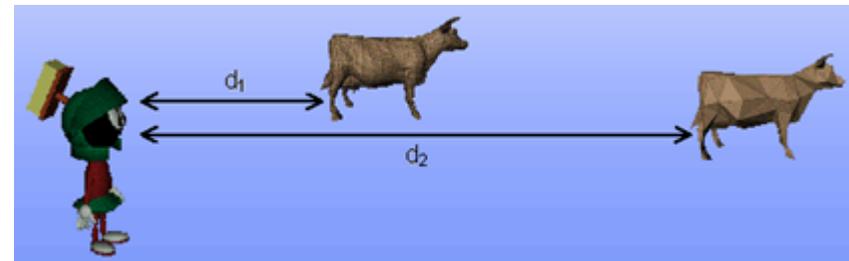


- Zwei Aufgaben:
 1. Preprocessing: Generierung von LODs
 2. Runtime: Auswahl des "richtigen" LODs, und ggf. "Blending"

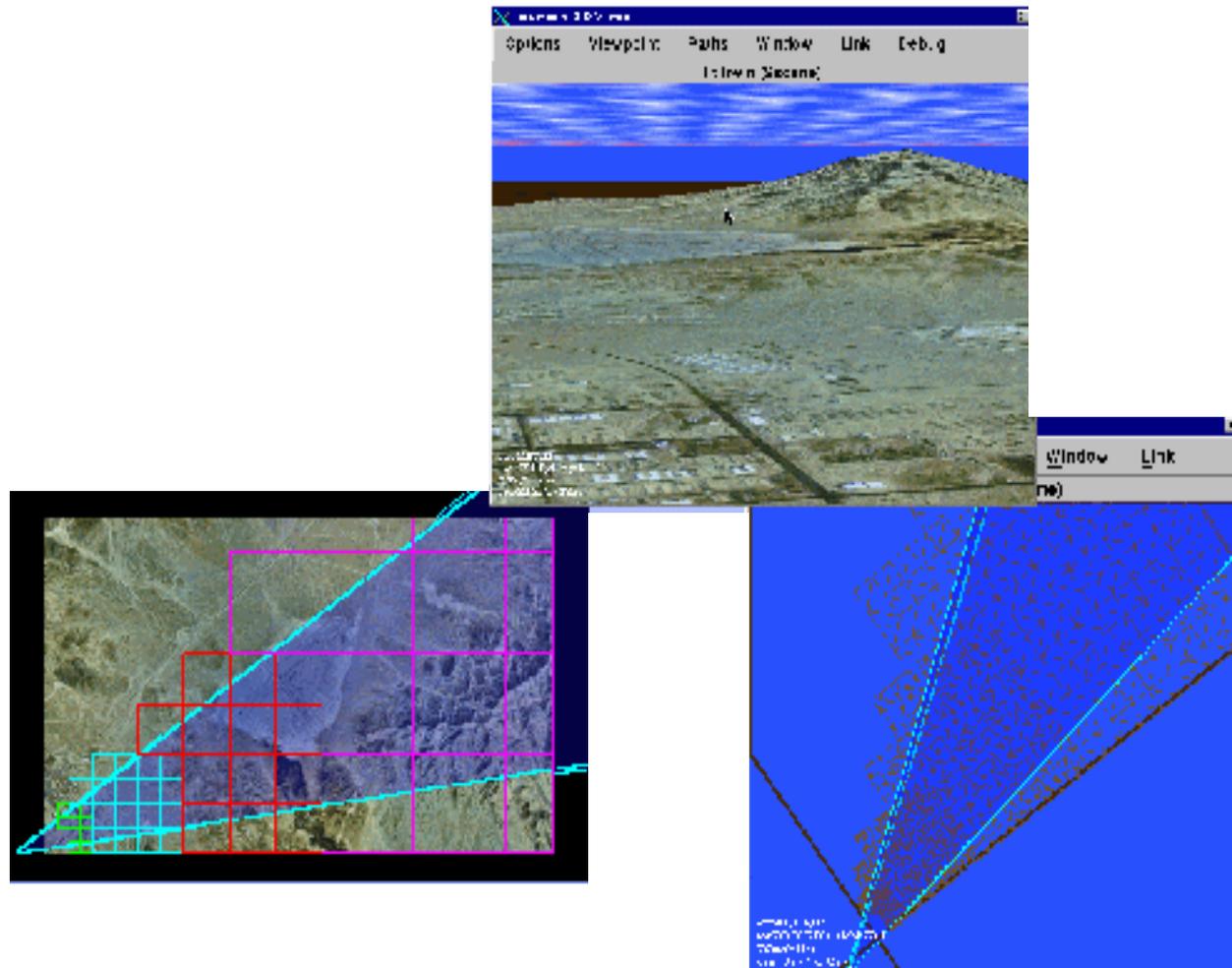
LOD-Selektion



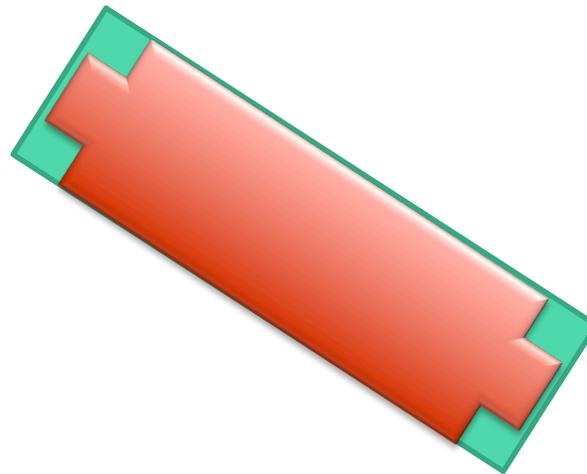
- Visuelle Güte gegen zeitliche Güte
- Wie selektiert man den "richtigen" Level?
- Statisch:
 - Level i für Entfernungsbereich (d_i, d_{i+1})
 - Abhängig von FOV
 - Größe des Obj unberücksichtigt
- Dynamisch:
 - Schätze Größe des Objektes auf dem Bildschirm
 - Unabhängig von Bildschirmauflösung, FOV, Größe des Obj
 - Ist automatisch entfernungsabhängig



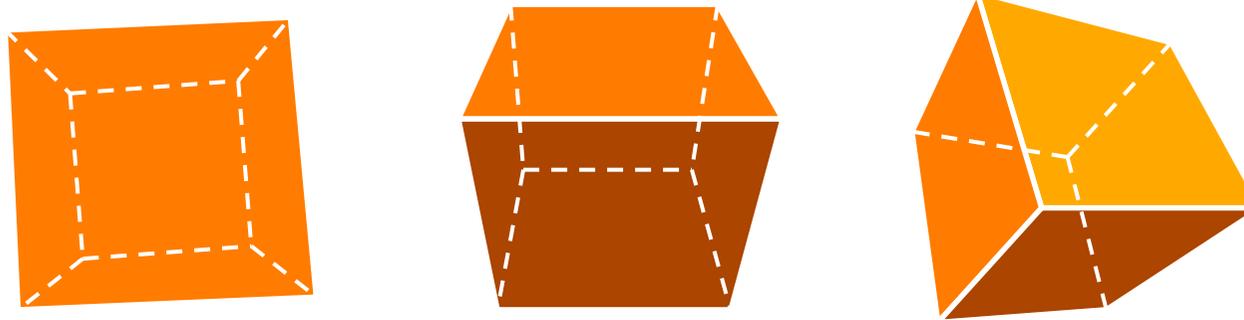
- Für Desktop-Applikationen i.A. schon ausreichend:



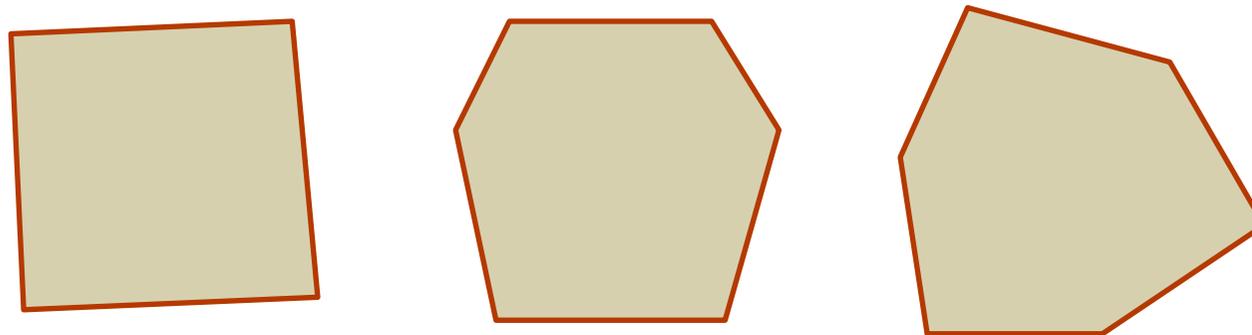
- Naive Methode:
 - Berechne Bounding-Box (bbox) des Objektes in 3D
 - Projiziere Bbox nach 2D → 8x 2D-Punkte
 - Bestimme 2D-Bbox (achsenparallel)
- Genauere Methode:
 - Bestimme wahre Fläche der projizierten 3D-Bbox auf dem Screen



- Bestimme die Anzahl der Seiten der Box, die in 2D sichtbar sind:



- Projiziere die Punkte auf der Silhouette (4 oder 6) nach 2D:



- Berechne den Betrag dieser (konvexen!) Fläche

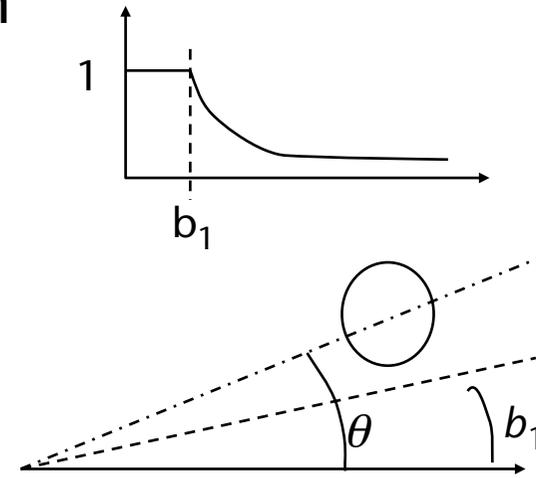
- Für jedes Paar von (parallelen) Seitenflächen (jeden *Slab*): klassifiziere Viewpoint bzgl. dieses Paares als "darunter", "darüber", oder "dazwischen"
- Ergibt $3 \times 3 \times 3 = 27$ Möglichkeiten
 - M.a.W.: die Ebenen der Seitenflächen eines Würfels unterteilen den Raum in 27 Teilräume
- Verwende Bit-Codes (à la Out-Codes) und eine Lookup-Table
 - Ergibt 2^6 theoretische Einträge
- 27-1 Einträge der LUT listen die 4 oder 6 Vertices der Silhouette
- Dann nur noch projizieren, triangulieren, Flächen aufsummieren

Psychophysiological LOD-Selektion

- Faktoren bzgl. der Sehschärfe:

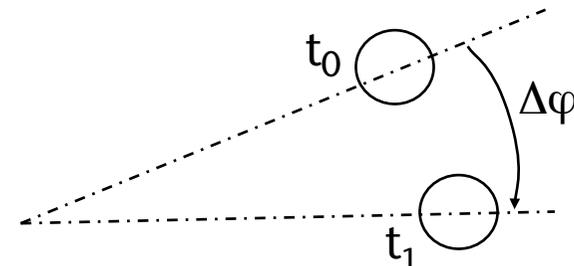
- Zentrum/Peripherie:

$$k_1 = \begin{cases} e^{-(\theta - b_1)/c_1} & , \theta > b_1 \\ 1 & , \text{sonst} \end{cases}$$



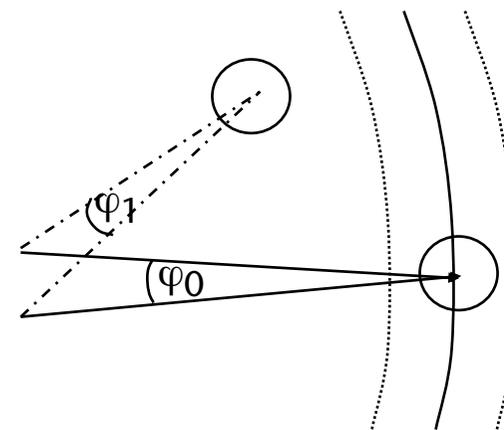
- Bewegung:

$$k_2 = e^{-\frac{\Delta\varphi - b_2}{c_2}}$$



- Unschärfe/Tiefe:

$$k_3 = e^{-\frac{|\varphi_0 - \varphi| - b_3}{c_3}}$$



- Bestimmung des LODs:

1. $k = \min\{k_i\} \cdot k_0$, oder $k = \prod k_i \cdot k_0$
2. $r_{\min} = 1/k$
3. Selektiere Level l so, daß

$$\forall p \in P_l : r(p) \geq r_{\min}$$

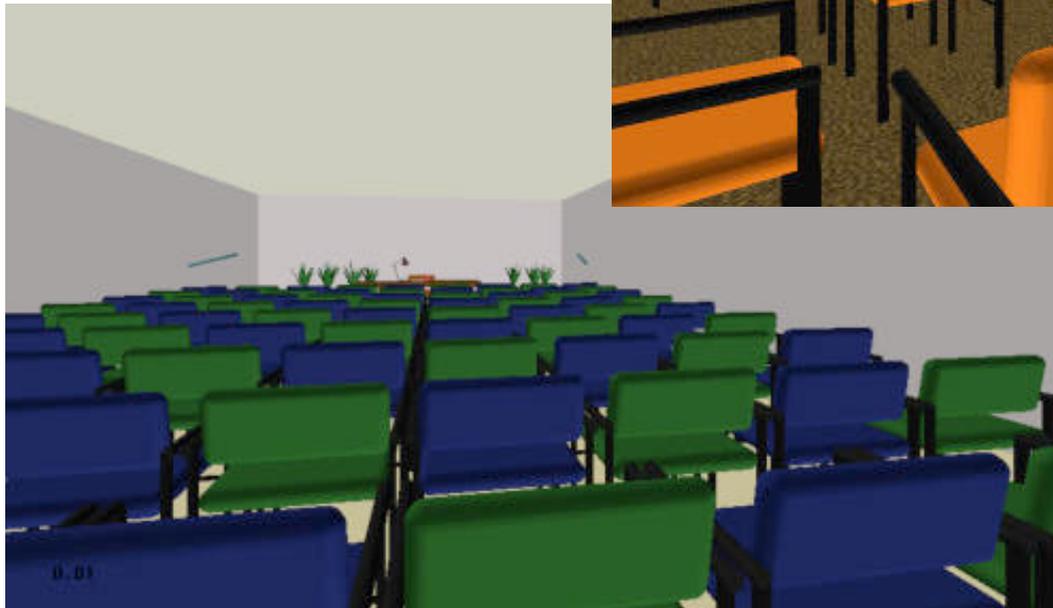
- Braucht man *Eye-Tracking*?

- Teuer, ungenau, "*intrusive*"
- Psychophysiologie: Augen immer $< 15^\circ$ ausgelenkt
- Treffe die Annahme Kopfrichtung=Augenrichtung, und wähle $b_1 = 15^\circ$

- Reaktive LOD-Selektion:
 - Verwalte Historie von Rendering-Zeiten
 - Bestimme daraus die geschätzte Zeit T_r für das kommende Frame
 - Sei T_b = das zur Verfügung stehende Zeit-Budget
 - Falls $T_r > T_b$: erniedrige LODs (größere Levels)
 - Falls $T_r < T_b$: wähle feinere Levels
 - Dann rendern und tatsächlich benötigte Zeit in die Historie aufnehmen



- Reaktive LOD-Selektion hat Ausreißer
- Beispiel-Szenen:



- Definition **Objekt-Tupel (O,L,R)**:

O = Objekt, L = Level,

R = Rendering Algo (#Texturen, anti-aliasing, #Lichtquellen)

- Bewertungsfunktionen für Objekt-Tupel:

Cost(O,L,R) = Rendering-Zeit

Benefit(O,L,R) = **"Beitrag zum Bild"**

- Optimierungsproblem:

suche
$$\max_{S' \subset S} \sum_{(O,L,R) \in S'} \text{benefit}(O, L, R)$$

unter der Bedingung
$$T_r = \sum_{(O,L,R) \in S'} \text{cost}(O, L, R) \leq T_b$$

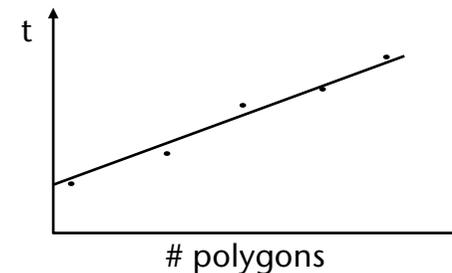
wobei
$$S = \{ \text{mögliche Objekt-Tupel in der Szene} \}$$

- Cost-Funktion hängt ab von:

- Anzahl Eckpunkte (\sim # Koord. Trafos + *Lighting-Calcs* + *Clipping*)
- Setup pro Polygon
- Anzahl Pixel (*Scanline-Konvertierung*, *Alpha-Blending*, *Textur*, *Anti-Aliasing*, *Phong*)
- Theor. Kostenmodell:

$$\text{Cost}(O, L, R) = \max \left\{ \begin{array}{l} C_1 \cdot \text{Poly} + C_2 \cdot \text{Vert} \\ C_3 \cdot \text{Pixels} \end{array} \right\}$$

- Besser experimentell bestimmen:
 Rendere versch. Objekte mit allen
 möglichen verschiedenen Parametern



- Benefit-Funktion: in den "Beitrag" zum Bild gehen ein

- Größe
- Shading-Methode: $\text{Rendering}(O, L, R) = \begin{cases} 1 - \frac{c}{\text{pgons}} & , \text{flat} \\ 1 - \frac{c}{\text{vert}} & , \text{Gouraud} \\ 1 - \frac{c}{\text{vert}} & , \text{Phong} \end{cases}$

- Distanz vom Zentrum (Peripherie, Tiefe)

- Geschwindigkeit

- Semantische "Wichtigkeit"

- Hysterese:

$$\text{Hysterese}(O, L, R) = \frac{c_1}{1 + |L - L'|} + \frac{c_2}{1 + |R - R'|}$$

- Zusammen:

$$\begin{aligned} \text{Benefit}(O, L, R) = & \text{Size}(O) \cdot \text{Rendering}(O, L, R) \cdot \\ & \text{Importance}(O) \cdot \text{OffCenter}(O) \cdot \\ & \text{Vel}(O) \cdot \text{Hysteresis}(O, L, R) \end{aligned}$$

- Optimierungsproblem: ist ein "*multiple-choice knapsack problem*"
→ NP-vollständig
- Idee: berechne sub-optimale Lösung :
 - Rückführung auf kontinuierliches Rucksackproblem
 - Dann greedy lösen mit zusätzlichem Constraint
 - Definiere
$$\text{value}(O, L, R) = \frac{\text{benefit}(O, L, R)}{\text{cost}(O, L, R)}$$
 - Sortiere alle Objekt-Tupel nach Value(O,L,R)
 - Wähle die ersten Objekt-Tupel bis Rucksack voll
 - Constraint: keine 2 Objekt-Tupel dürfen dasselbe Objekt darstellen

- Inkrementelle Lösung:

- Starte mit der Lösung $(O_1, L_{1,1}), \dots, (O_n, L_n, R_n)$ vom letzten Frame

- Falls

$$\sum_i \text{cost}(O_i, L_i, R_i) \leq \text{max. Frame-Zeit}$$

dann suche Objekt-Tupel (O_k, L_k, R_k) ,

so daß

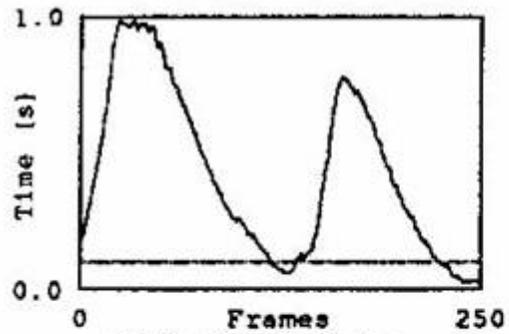
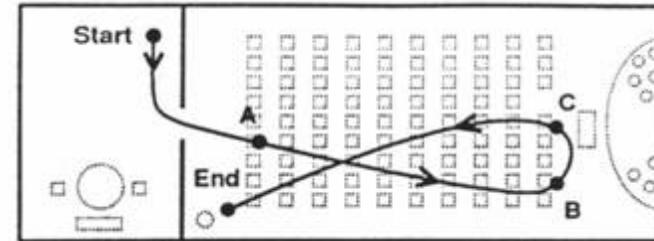
$$\text{value}(O_k, L_k + a, R_k + b) - \text{value}(O_k, L_k, R_k) = \text{max}$$

und

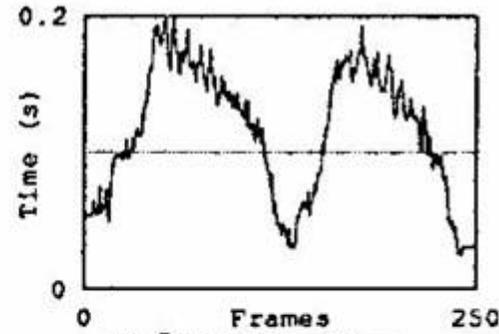
$$\sum_{i \neq k} \text{cost}(O_i, L_i, R_i) + \text{cost}(O_k, L_k + a, R_k + b) \leq \text{max. Frame-Zeit}$$

- Analog, falls $\sum_i \text{cost}(O_i, L_i, R_i) > \text{max. Frame-Zeit}$

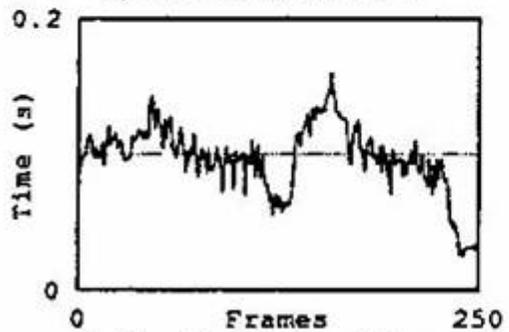
- Performance in der Beispielszene:



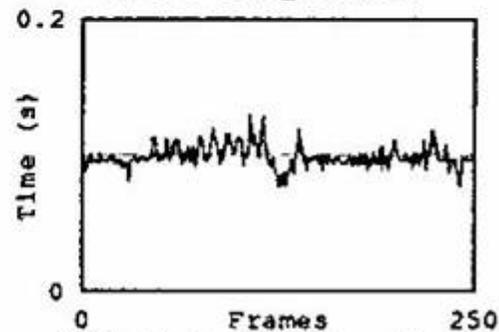
a) No detail elision.



b) *Static* algorithm.



c) *Feedback* algorithm.



d) *Optimization* algorithm.



Screenshots aus der Beispielszene



No detail elision, 19,821 polygons



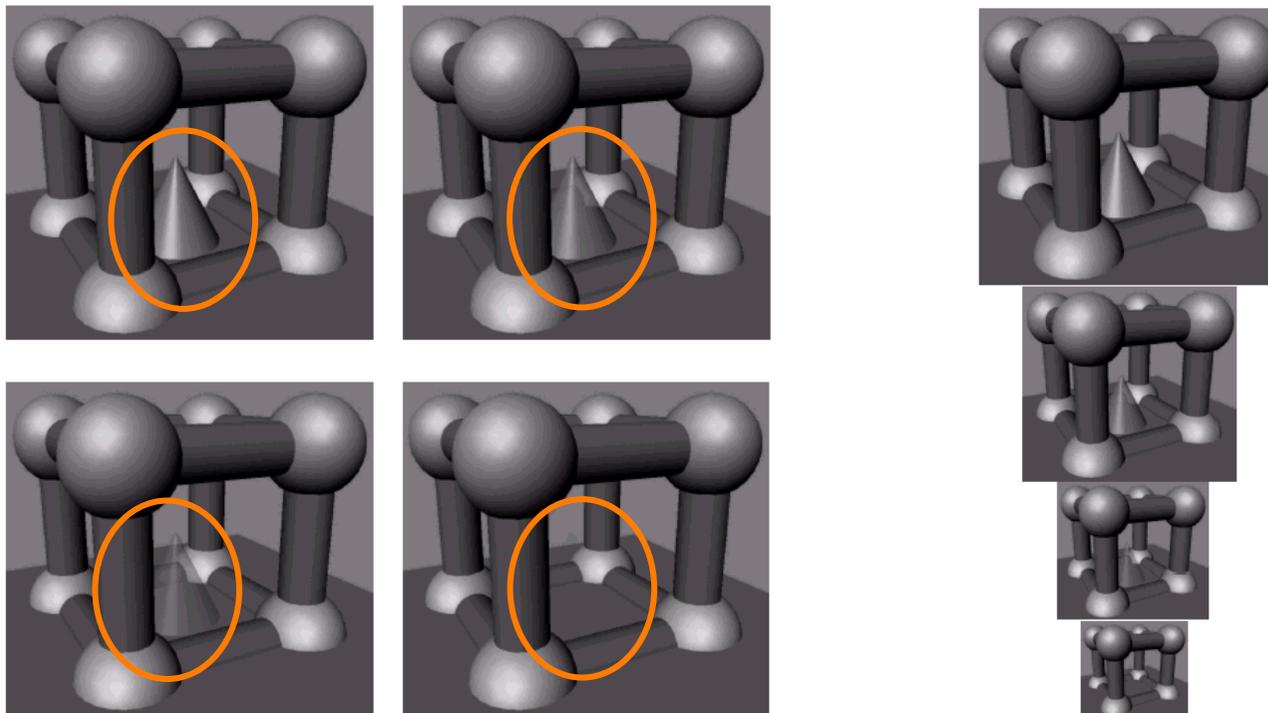
Optimization, 1,389 polys,
0.1 sec/frame target frame time

Level of detail: darker
gray means more detail



- "Popping" beim Umschalten zwischen Levels
- Maßnahmen gegen "Popping":
 - Hysterese
 - Alpha-Blending der beiden benachbarten LOD-Stufen
 - Man kommt vom Regen in die Traufe ;-)
 - Kontinuierliche, view-dependent LODs

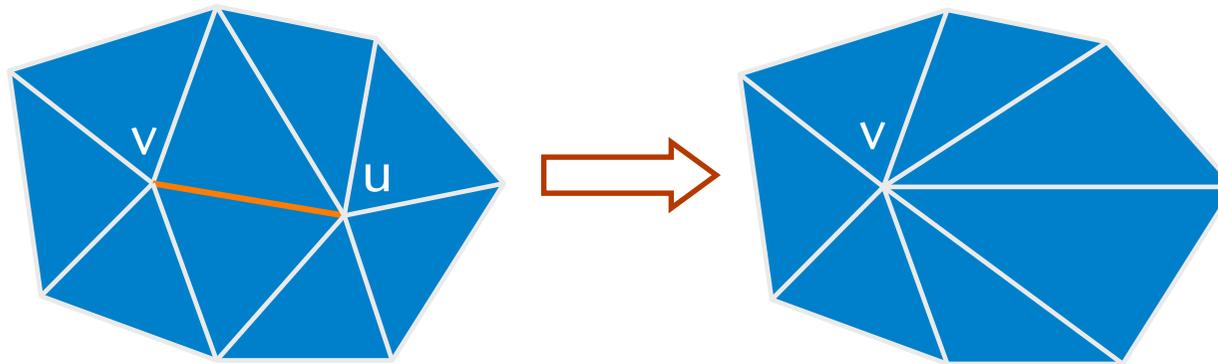
- Einfache Idee, um Popping zu vermeiden:
when beyond a certain range, fade out object until gone



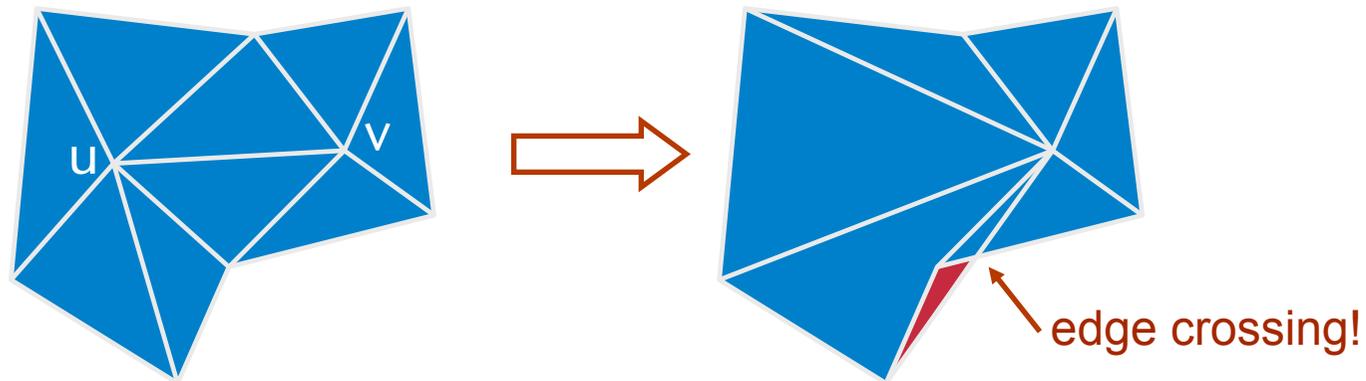
- A.k.a. **Geomorph-LODs**
- Idee:
 - Gegeben zwei Meshes (LODs desselben Objektes) M_i und M_{i+1}
 - Erzeuge ein Mesh M' "zwischen" diesen beiden
- Definition: **Progressive Mesh** = Repräsentation eines Objektes, ausgehend von einem hoch-aufgelösten Mesh M_0 , mit Hilfe derer man (fast) stufenlos zwischen 1 Polygon und M_0 "Zwischen-Meshes" generieren kann (möglichst schnell).

Erzeugung von Progressive Meshes

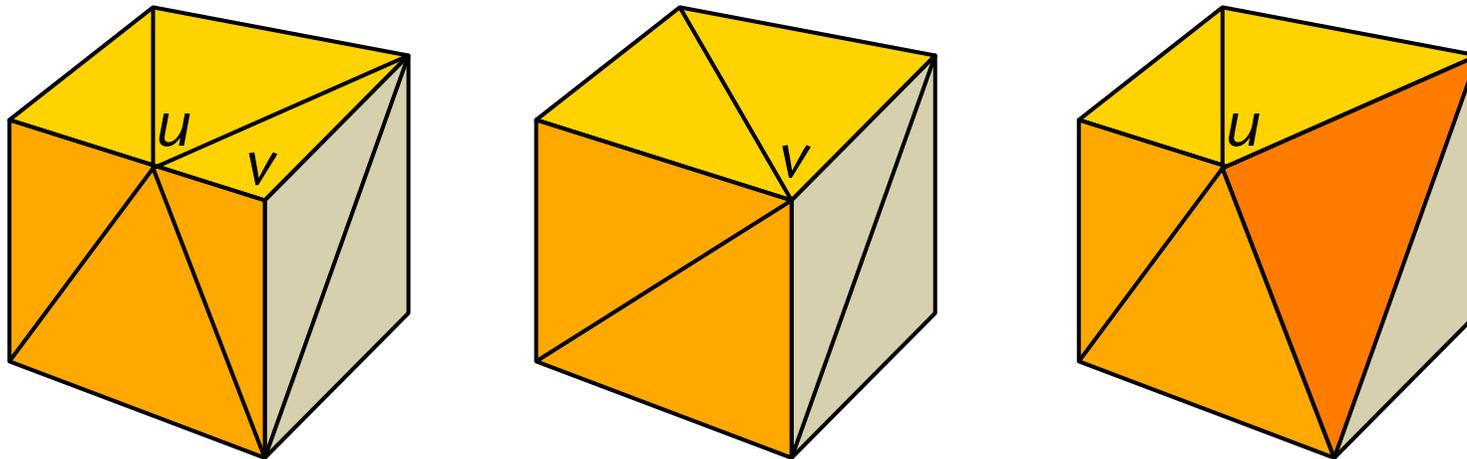
- **Simplifizierung** (*simplification*)
- Grundlegende Operation: *edge collapse*



- **Bad Edge Collapses:**

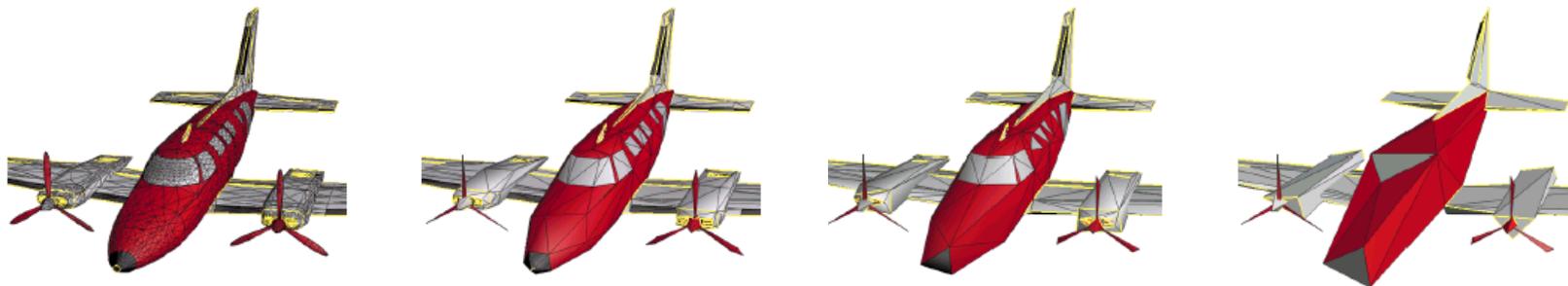


- Reihenfolge der Edge Collapses:
 - Führe Bewertung eines Edge-Collapses ein, die visuellen Effekt "misst"

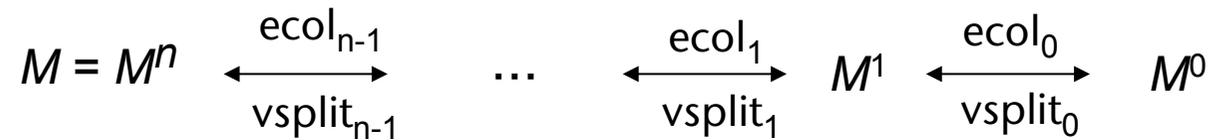


- Führe zunächst Edge Collapses mit kleinster Auswirkung aus
- Nach jedem Schritt müssen im Prinzip alle Kanten neu bewertet werden

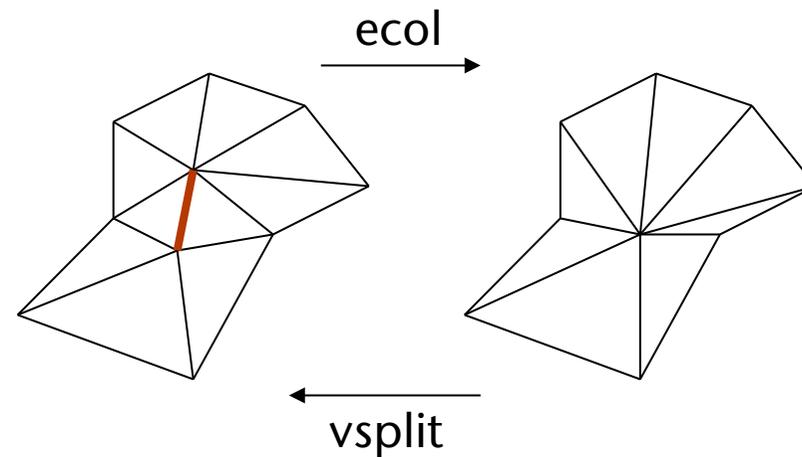
- Bewertungsfunktion für Edge-Collapses ist nicht trivial und, vor allem, wahrnehmungsbasiert!
- Beeinflussende Faktoren:
 - Krümmung der Kanten / Fläche
 - Beleuchtung, Texturierung, Viewpoint (Highlights!)
 - Bedeutung der Geometrie (Augen & Mund sind besonders wichtig)
- Beispiel eines Progressive Mesh:



■ Repräsentation eines Progressive Meshes:



- $M^{i+1} = i$ -te Verfeinerung (*refinement*) = 1 Vertex mehr als M^i



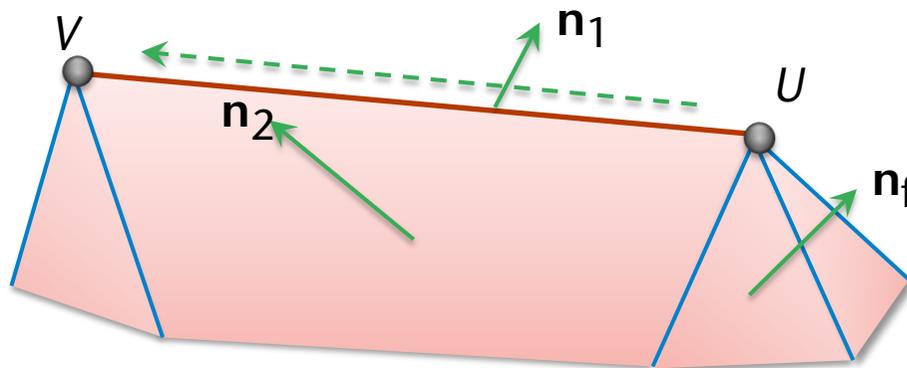
■ Repräsentation eines Edge Collapse / Vertex Split:

- Betroffenes Paar von Vertices (Kante)
- Position des "neuen" Vertex
- Dreiecke, die gelöscht / eingesetzt werden

- Heuristik:
 - Kleine Kanten zuerst entfernen
 - Einen Vertex U auf einen Vertex V ziehen, falls die Fläche um U eine geringe (diskrete) Krümmung hat
- Ein einfaches Kostenmaß für einen Edge-Collapse von U auf V :

$$\text{cost}(U, V) = \|U - V\| \cdot \text{curv}(U)$$

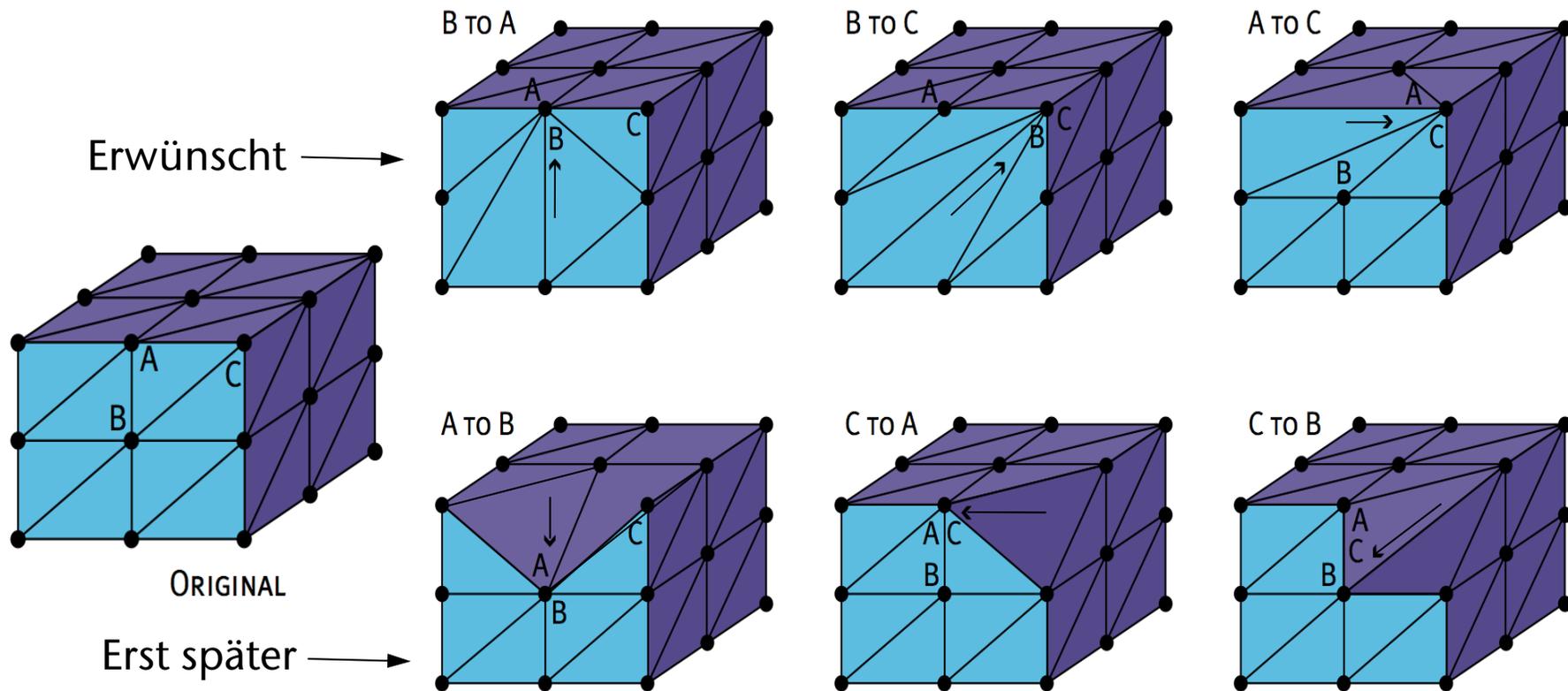
$$\text{curv}(U) = \frac{1}{2} \left(1 - \min_{f \in T(U) \setminus T(V)} \max_{i=1,2} \mathbf{n}_f \cdot \mathbf{n}_i \right)$$

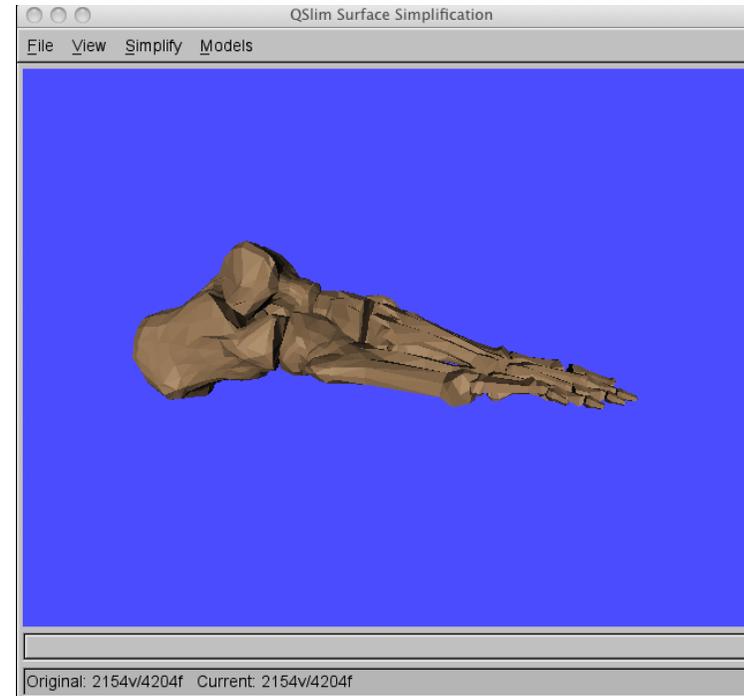
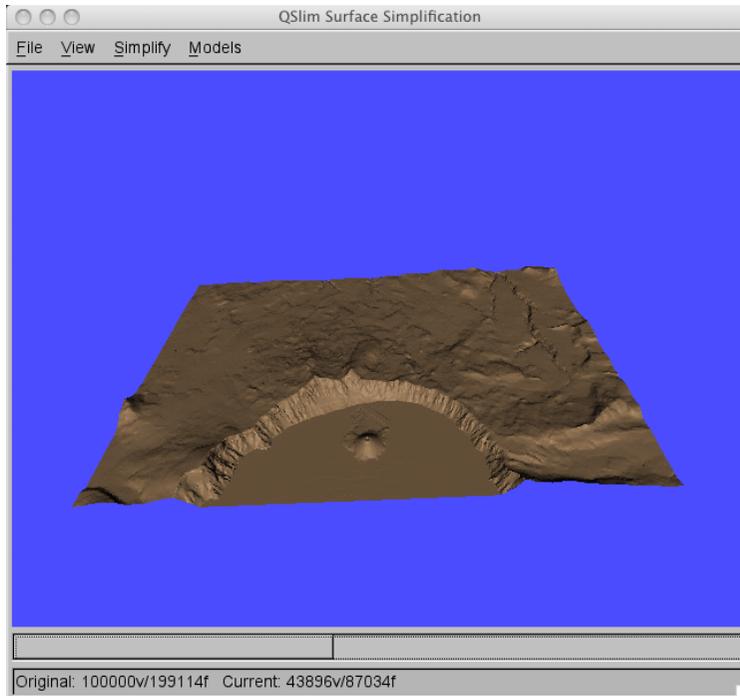


- Bemerkung:

$$\text{cost}(U, V) \neq \text{cost}(V, U)$$

- Beispiel:





[Michael Garland: Qslim]

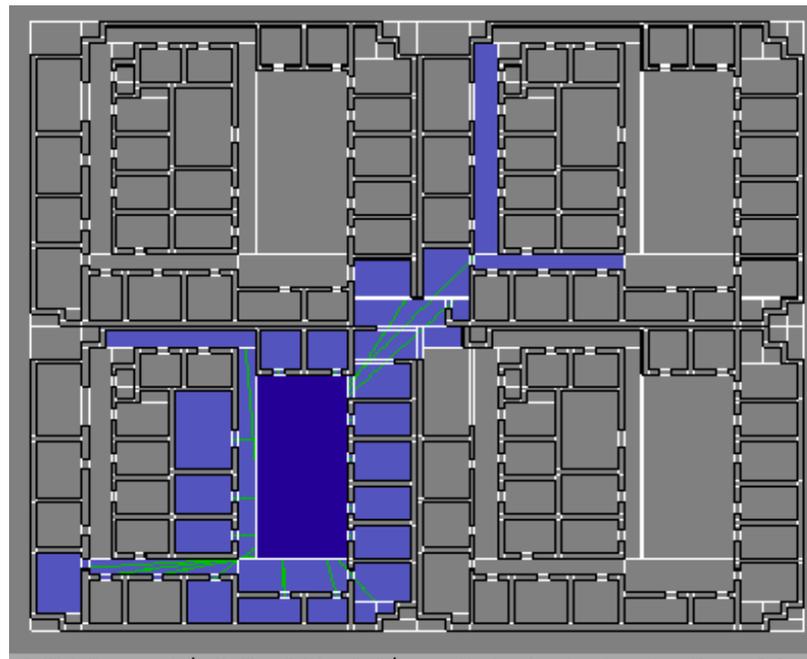
Wie funktioniert der Funkhouser-Sequin-Algo mit progressiven Meshes?



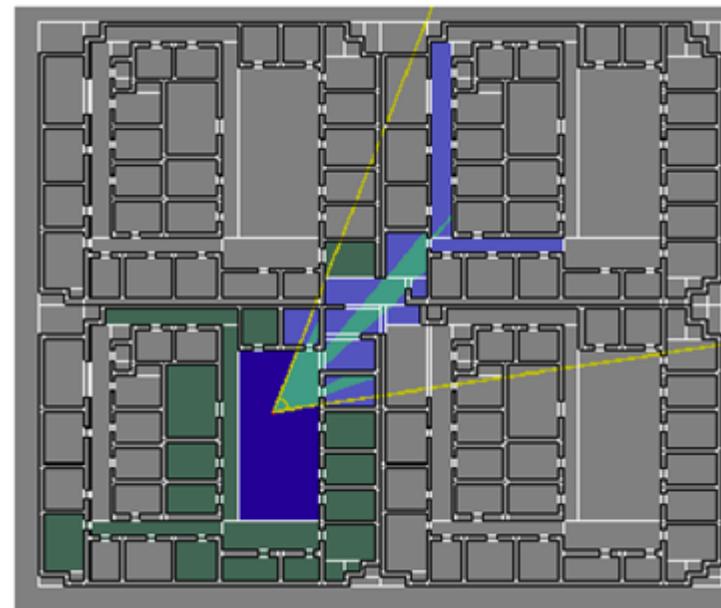
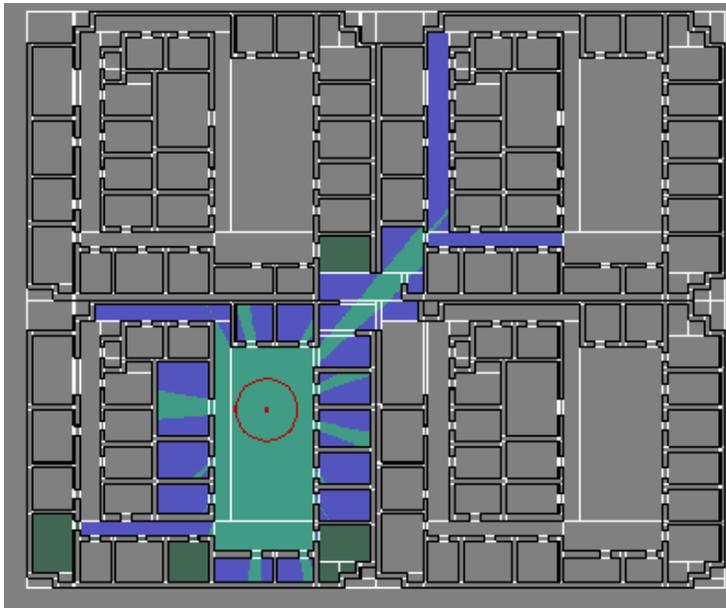
- Idee: LOD-Technik auf nicht-geometrische Inhalte anwenden
- Z.B. "*Behavioral LOD*":
 - Simuliere Verhalten eines Objektes exakt, wenn im Fokus, sonst nur "grob"

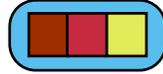
Culling in Gebäuden (portal culling)

- Beobachtung: viele Räume innerhalb des Viewing-Frustums sind nicht sichtbar
- Idee:
 - Unterteile Raum in "Zellen"
 - Berechne *Cell-to-Cell-Visibility* im voraus



- Von bestimmtem Viewpoint aus ist innerhalb der Zelle noch weniger sichtbar:



- State = Zustand = 
 - Zusammenfassung aller Attribute
 - Beispiele für Attribute: Farbe, Material, Lighting-Param., Textur, Blend-Fkt., Shader-Programm, etc.
 - Jedes Attribut hat zu jedem Zeitpunkt genau 1 Wert aus einer endlichen Menge
- State-Wechsel sind einer der schlimmen Performance-Killer
- Kosten: 

Matrix-Stack- Modifikation	Lighting- Modifikation	Textur- Modifikation	Shader-Programm- Modifikation
-------------------------------	---------------------------	-------------------------	----------------------------------
- Ziel: kompletten Szenengraphen mit minimaler Anzahl State-Wechsel rendern
- "Lösung": Pre-Sorting

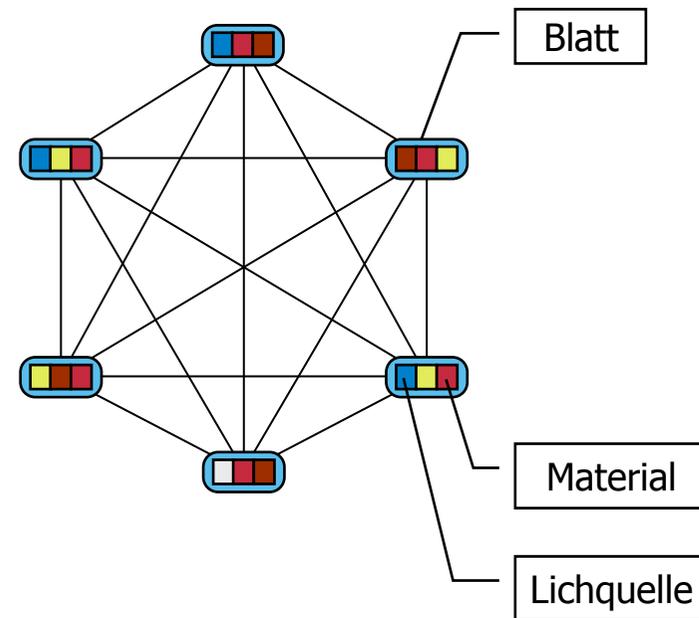
- Problem: die optimale Lösung ist NP-vollständig

- Denn:

- Jedes Blatt ist ein Knoten in einem vollständigen Graphen,
- Kosten einer Kante = Kosten der State-Wechsel (verschiedene State-Wechsel kosten verschieden viel, z.B. sind neue Trafos relativ billig),
- Gesucht: kürzester Weg

→ *Traveling Salesman Problem*

- Weiteres Problem: klappt nicht bei dynamischen Szenen und Occlusion Culling



- Idee & Abstraktion:
 - Betrachte nur ein Attribut ("Farbe")
 - Schalte Buffer zwischen App. und Graphikkarte
 - Buffer enthält Elemente mit verschiedenen Farben
 - Pro Schritt eine von drei Operationen:
 - Element direkt an Graphikkarte weiterreichen
 - Element lesen und in Buffer schreiben
 - Teilmenge aus Buffer löschen und an Graphikkarte schicken



- Zwei Algorithmen-Klassen:
 - "Online"-Algorithmen: Algo kennt *nicht* zukünftige Elemente!
 - "Offline"-Algorithmen: Algo kennt *alle* Elemente, muß aber trotzdem Buffer verwenden
- Betrachte nur "lazy" online-Strategie:
 - Elemente werden nur bei Buffer-Overflow aus Buffer entfernt
 - Jede nicht-lazy Online-Strategie läßt sich in eine lazy Strategie mit gleichen Kosten umwandeln
- Frage: welche Elemente muß man bei Buffer-Overflow auswählen, damit minimale Anzahl Farbwechsel auftritt?

- Definition *c-competitive* :

Sei $C_{off}(k)$ = Kosten (Anzahl Farbwechsel) der optimalen Offline-Strategie, k = Buffer-Größe.

Sei $C_{on}(k)$ = Kosten der Online-Strategie.

Dann heißt diese Strategie "*c-competitive*" gdw.

$$C_{on}(k) = c \cdot C_{off}(k) + a$$

wobei a von k unabhängig ist.

- Gesucht: Online-Strategie mit möglichst kleinem c
(im worst-case, und – wichtiger noch – im average case)

Beispiel: LRU (least-recently used)

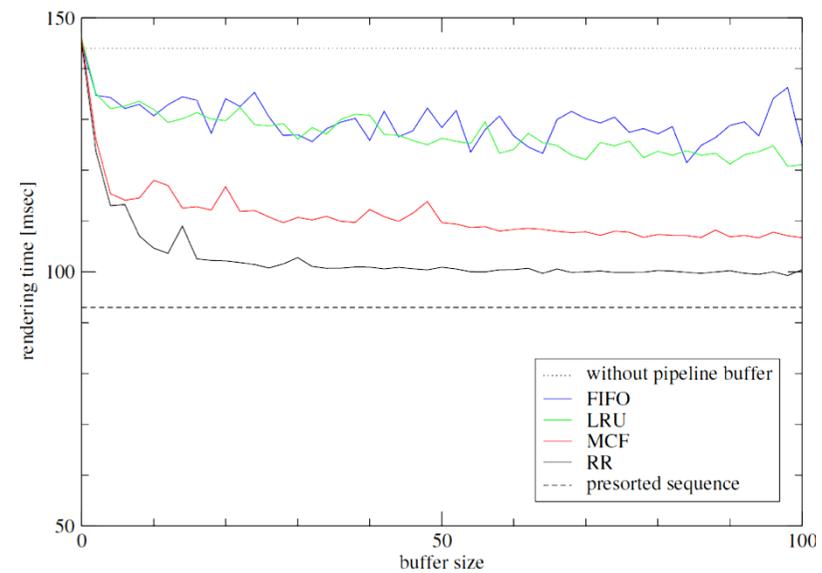
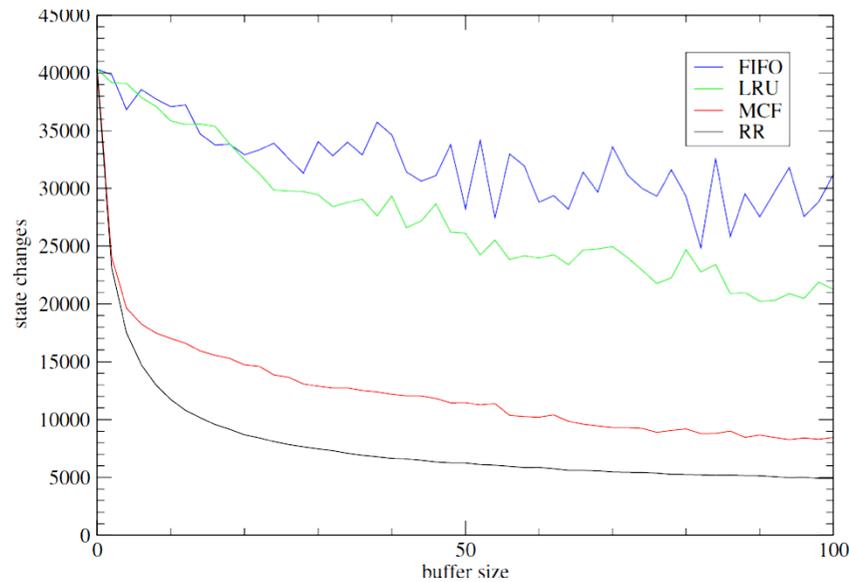
- Strategie:
 - Pro Farbe ein Timestamp (**nicht pro Element!**)
 - Element wird in Buffer geschrieben → Timestamp seiner Farbe wird auf aktuelle Zeit gesetzt
 - Achtung: dabei können die Timestamps anderer Elemente im Buffer auch aktualisiert werden
 - Buffer-Overflow → entferne Elemente, deren Farbe ältesten Timestamp hat
- Untere Schranke für die Competitive-Ratio: $\Omega(\sqrt{k})$
- Beweis durch Beispiel:
 - Setze $m = \sqrt{k - 1}$, oBdA m gerade
 - Wähle die Eingabe $(c_1 \cdots c_m x^k c_1 \cdots c_m y^k)^{\frac{m}{2}}$
 - Kosten der **Online**-LRU-Strategie: $(m + 1) \cdot 2 \cdot \frac{m}{2}$ Farbwechsel
 - Kosten der **Offline**-Strategie: $2m$ Farbwechsel, da Ausgabe = $(x^k y^k)^{\frac{m}{2}} c_1^m \cdots c_m^m$

Bounded-Waste- & Random-Choice-Strategie

- Idee: zähle Platzbedarf jeder Farbe im Buffer über die gesamte bisherige Zeit aufsummiert
- Führe Waste-Zähler $W(c)$ ein:
 - Bei Farbwechsel: erhöhe $W(c)$ um Anzahl Elemente im Buffer mit Farbe c
- Bounded-Waste-Strategie:
 - Bei Buffer-Overflow entferne alle Elemente der Farbe c' , wobei $W(c')$ maximal
- Competitive Ratio (o.Bew.): $O(\log^2 k)$
- Random-Choice-Strategie:
 - Randomisierte Version von Bounded-Waste
 - Wähle uniform zufälliges Element aus Buffer, entferne alle Elemente mit derselben Farbe
 - Folge: häufige Farbe wird häufiger ausgewählt, über die Zeit ergibt sich gerade $W(c)$

- Problem: Generierung guter Zufallszahlen dauert rel. lange
- RR-Strategie:
 - Variante von Random-Choice
 - Wähle nicht zufälligen Slot im Buffer, sondern den gemäß eines Pointers
 - Dieser Pointer wird in Round-Robin-Manier weitergeschaltet

- Fazit:
 - Round-Robin ist sehr gut (obwohl sehr einfach)
 - Worst-Case sagt sehr wenig über prakt. Perf.



Stereo *ohne* 2x rendern (einfaches Image-Warping)

- Beobachtung: linkes und rechtes Bild unterscheiden sich wenig
- Idee: 1x für rechts rendern, dann Pixel verschieben
- Algo: betrachte alle Pixel auf jeder Scanline *von rechts nach links*, zeichne Pixel *k* an neuer x-Pos.

$$x'_k = x_k + \frac{i}{\Delta} \frac{z_k}{z_k + z_0}, \quad \Delta = \text{Pixelbreite}$$

- Probleme:
 - Löcher!
 - Up-Vektor muß senkrecht sein
 - Reflexionen und specular highlights sind an falscher Pos
 - Aliasing

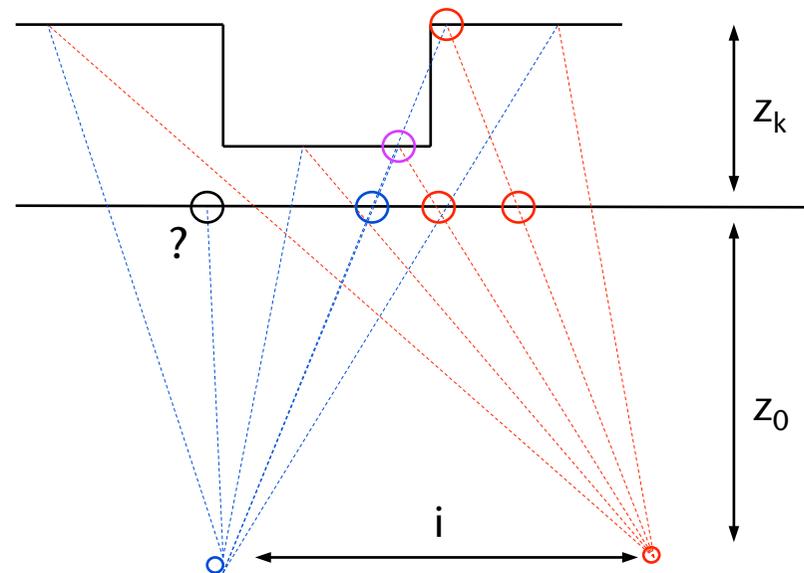
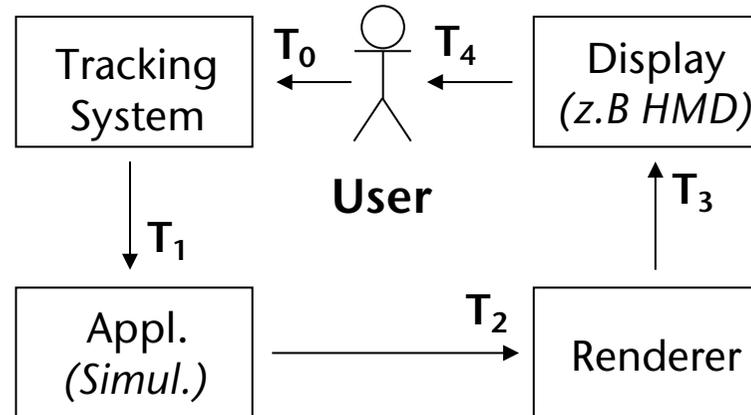
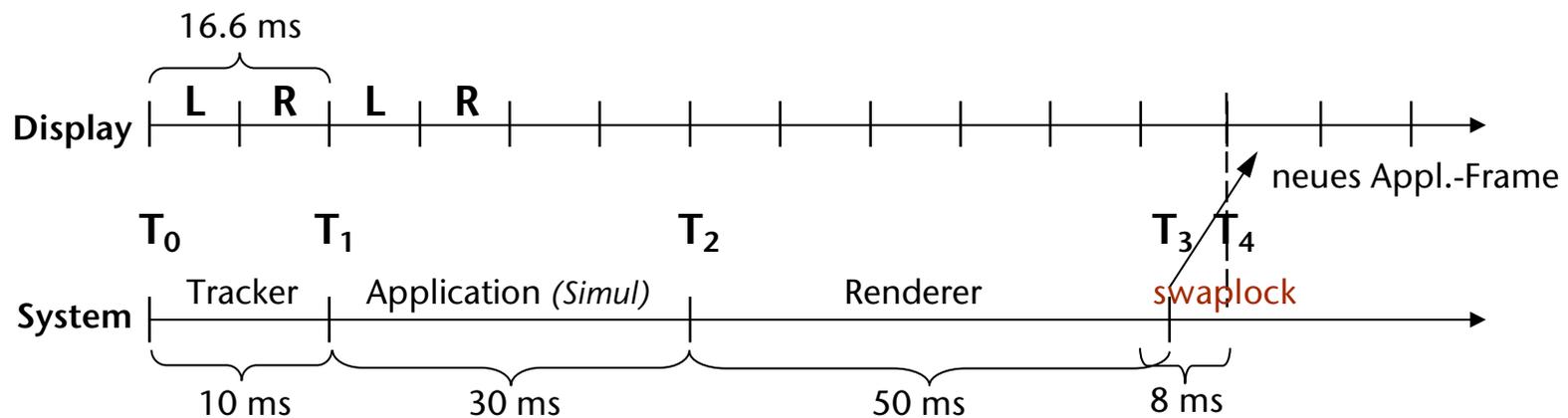


Image Warping

- Ein naives VR-System:

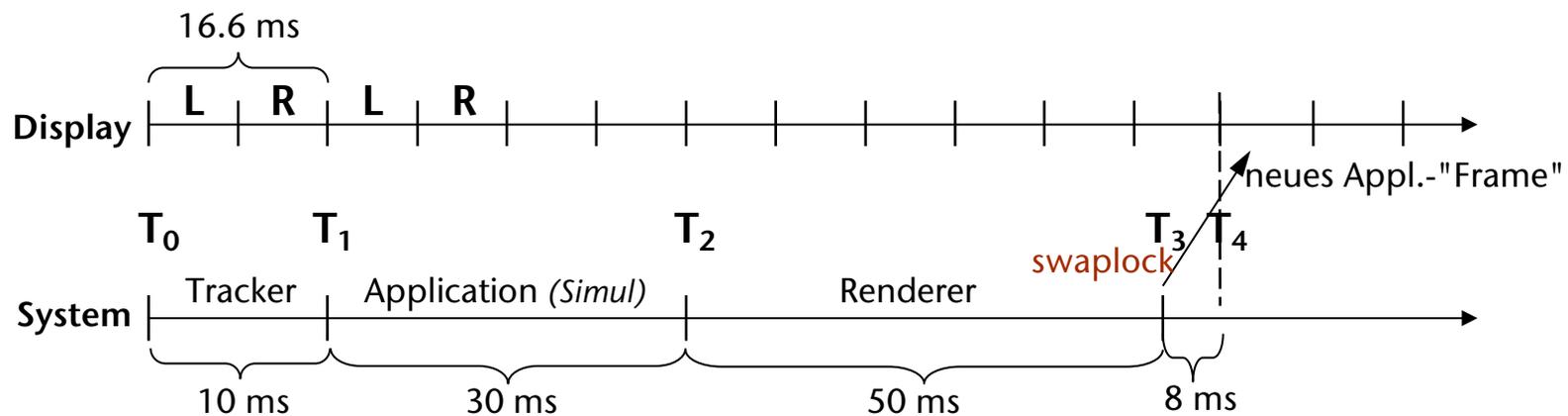


- Latenz in diesem System (Stereo mit 60 Hz \rightarrow Display-Refresh = 120 Hz):

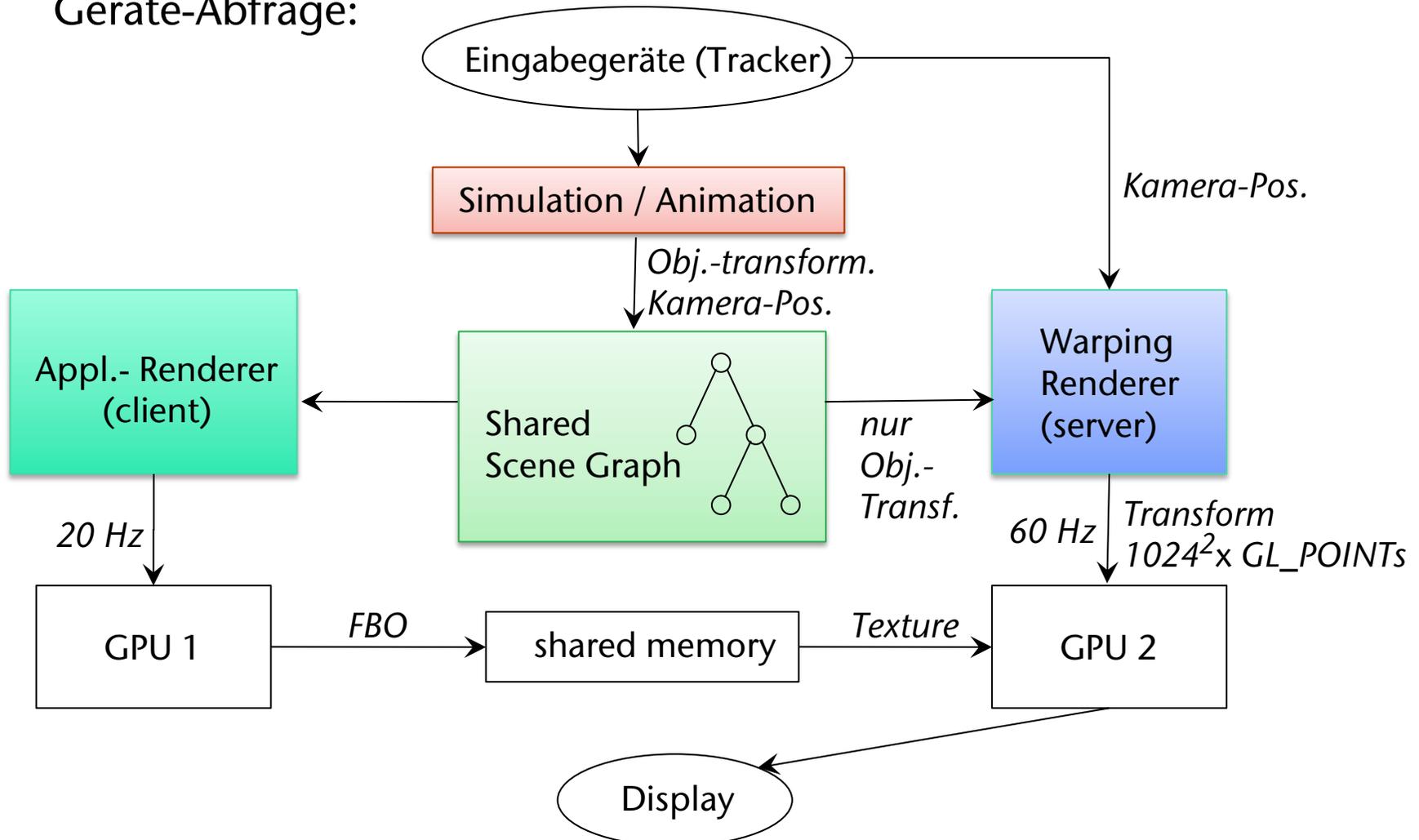


■ Probleme / Beobachtungen:

- Die Appl.-Framerate (inkl. Rendering) ist typischerweise viel langsamer als die Display-Refresh-Rate
- Die Tracking-Werte, die zu einem bestimmten Bild geführt haben, liegen sehr weit in der Vergangenheit
- Der Tracker könnte wesentlich öfter aktuelle Werte liefern
- Die Frames unterscheiden sich (normalerweise) rel. wenig voneinander (**temporal coherence**)



- Entkopple Simulation / Animation, Rendering, und Geräte-Abfrage:



Ein Application-Frame (client)

- Der App.-Renderer rendert zu einem Zeitpunkt t_1 ein ganz normales Frame
 - Color-Buffer und Z-Buffer
- ... und speichert **zusätzlich** einige Informationen:
 - zu jedem Pixel eine Obj-ID, das dort sichtbar ist
 - die Kamera-Transformationen zum Zeitpunkt t_1

$$T_{t_1, cam \leftarrow img} \quad , \quad T_{t_1, wld \leftarrow cam}$$

- zu jedem Obj i die Transformation

$$T_{t_1, obj \leftarrow wld}^i$$

Warping eines Frames (server)

- Zu einem späteren Zeitpunkt t_2 generiert der Server ein Bild aus dem Application-Frame mittels **Warping**

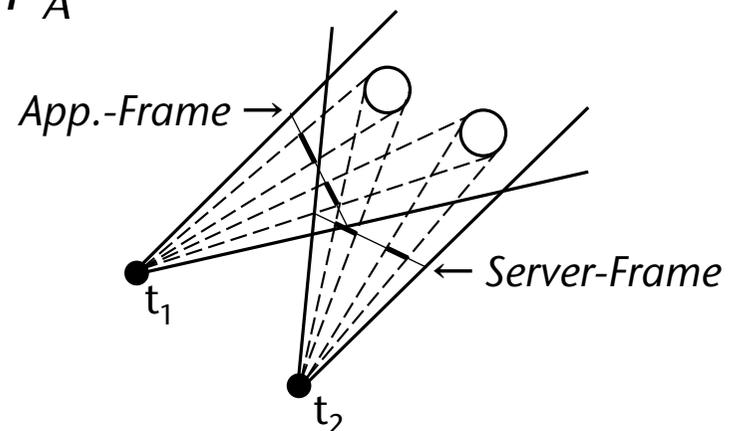
- Transformationen zu diesem Zeitpunkt:

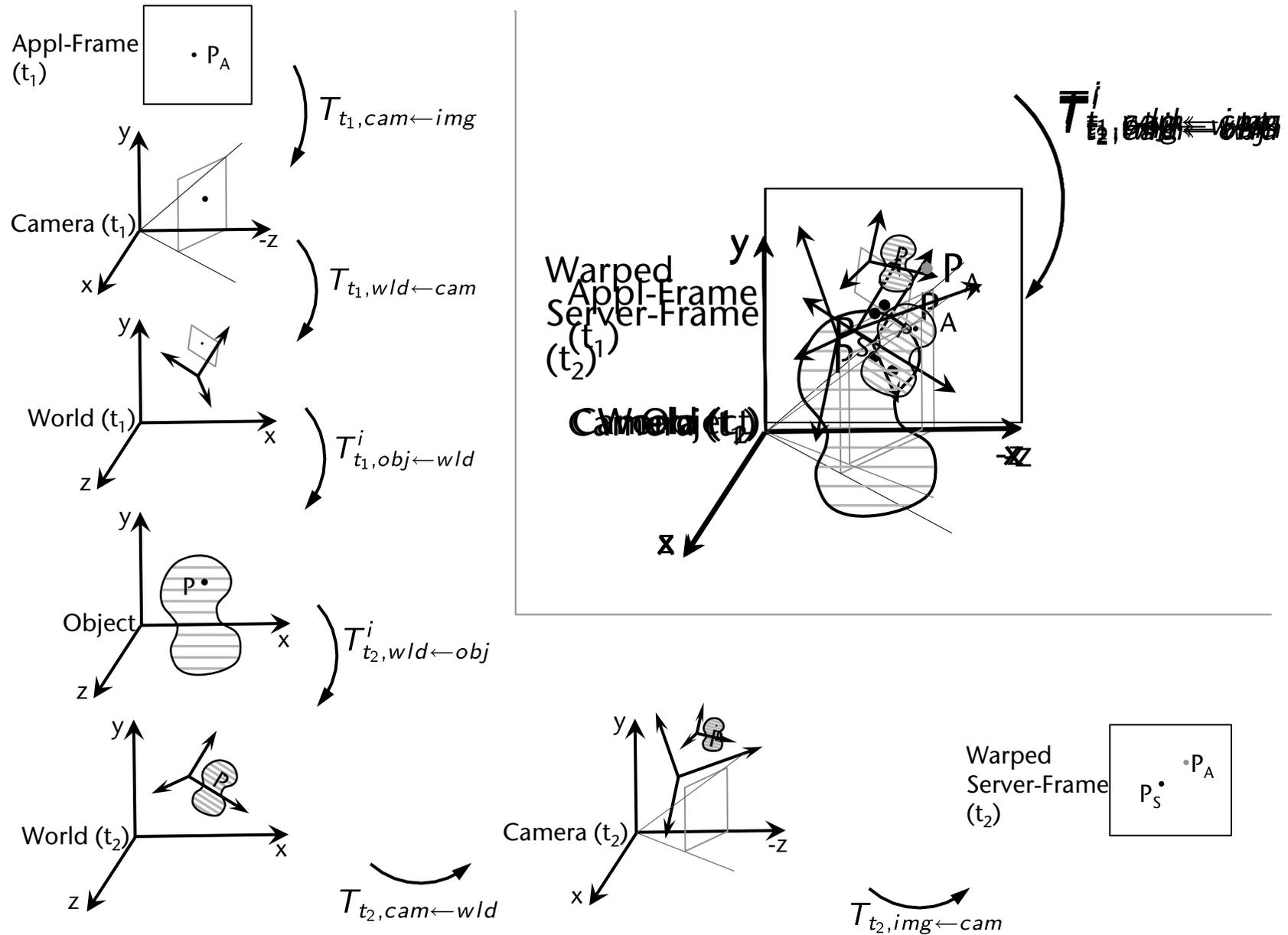
$$T_{t_2, wld \leftarrow obj}^i \quad T_{t_2, img \leftarrow cam} \quad T_{t_2, cam \leftarrow wld}$$

- Ein Pixel $P_A = (x, y, z)$ aus dem App.-Frame wird damit an die richtige Stelle im Server-Frame "gewarped":

$$P_S = T_{t_2, img \leftarrow cam} \cdot T_{t_2, cam \leftarrow wld} \cdot T_{t_2, wld \leftarrow obj}^i \cdot T_{t_1, obj \leftarrow wld}^i \cdot T_{t_1, wld \leftarrow cam} \cdot T_{t_1, cam \leftarrow img} \cdot P_A$$

- Diese Transf.-Matrix kann man zu Beginn eines Server-Frames für jedes Objekt vorberechnen



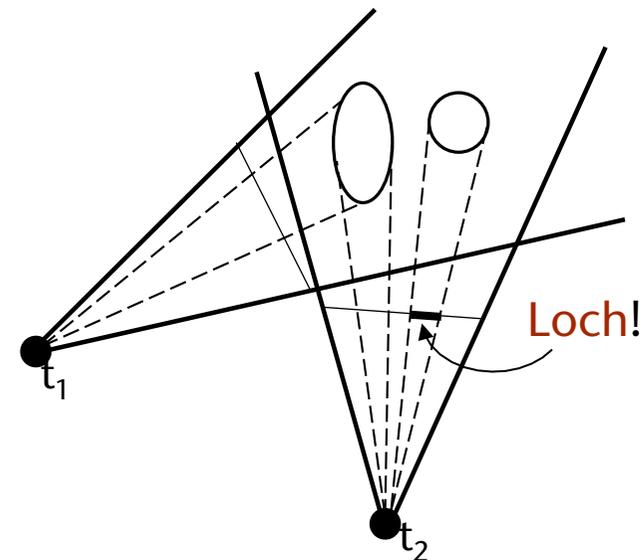


- Implementierung des Warpings:
 - Im Vertex-Shader
 - Geht nicht im Fragment-Shader, da dort Output-Pos. festgelegt ist !
 - Waring Renderer lädt FBO des App.-Frames in Textur, dito alle Ti's
 - Rendere 1024x1024 viele GL_POINTS (sog. Point-Splats)

- Vorteile:
 - Die Frames sind wesentlich aktueller (Kamera und Obj-Pos.!)
 - Server-Framerate ist unabhängig von Anzahl Polygone

- Probleme:

- Löcher im Server-Frame
- Server-Frames sehen unscharf aus (wg. Point-Splats)
- Wie groß sollen die Point-Splats sein?
(kann man zwar abschätzen, aber ...)
- Wenn die App.-Renderer-Framerate zu langsam ist, dann werden die Server-Frames doch zu schlecht
- Leere Stellen an den Rändern des Servers-Frames (evtl. View-Frustum im Client vergrößern)



- Performance-Gewinn:

- 12 Mio Polygone, 800 x 600
- ca.(!) Faktor 20 schneller

An Image-Warping Architecture for VR: Low Latency versus Image Quality

(Single-GPU Implementation)

Submitted to:
IEEE VR 2009

Image-Warping Architecture for VR Low Latency versus Image Quality

(Multi-GPU Implementation)

Submitted to:

IEEE VR 2009

