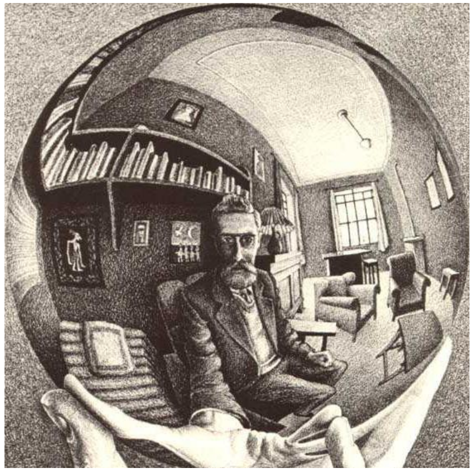
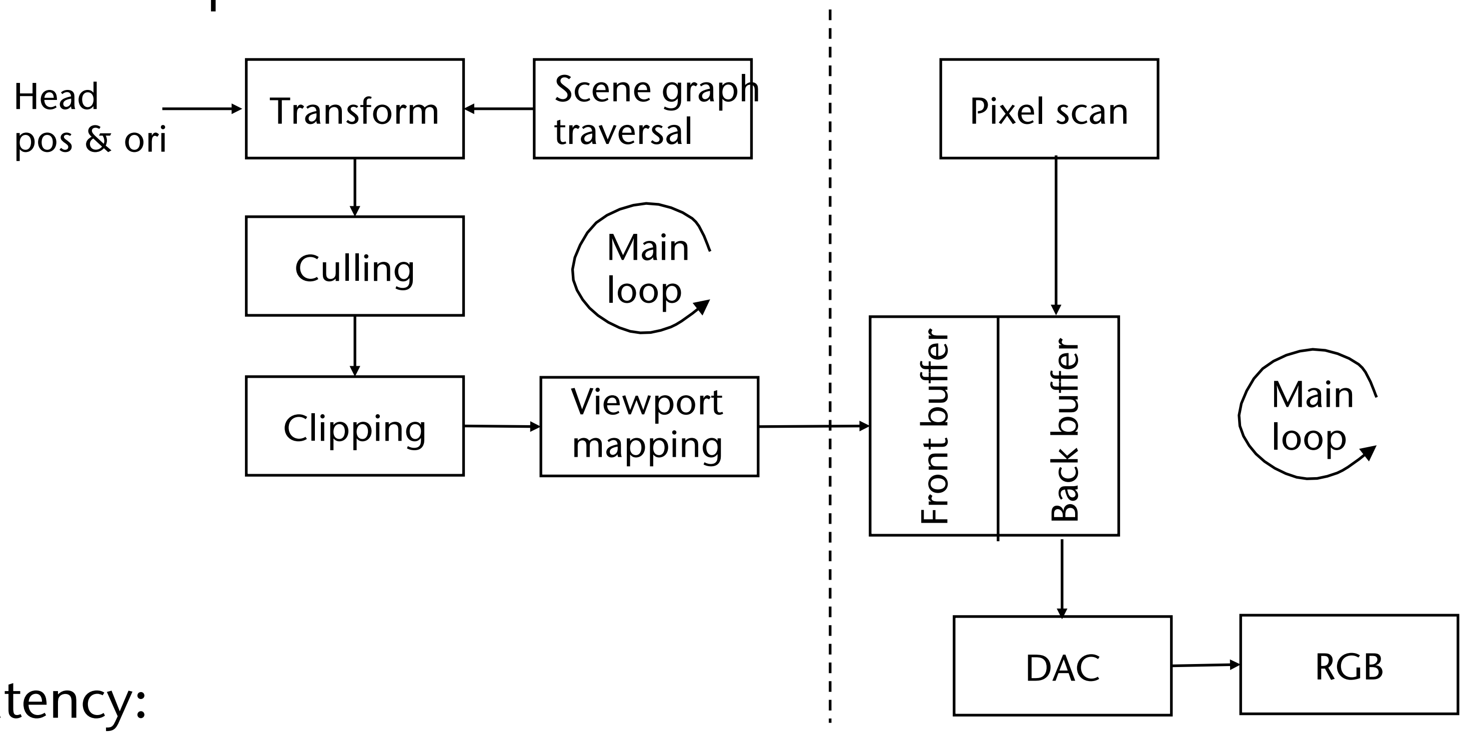
Virtual Reality
Real-time Rendering



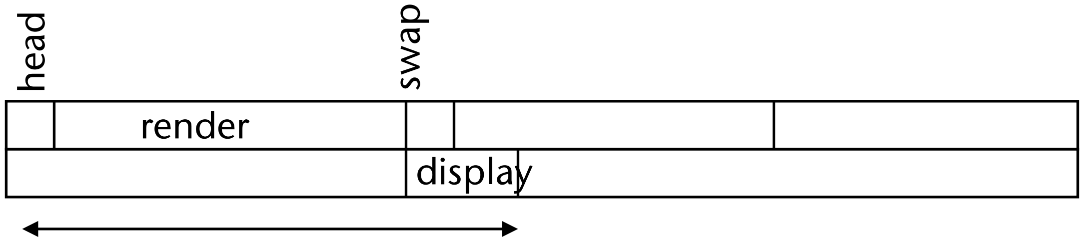
G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

Sources of Latency During Rendering

- Classical Pipeline:



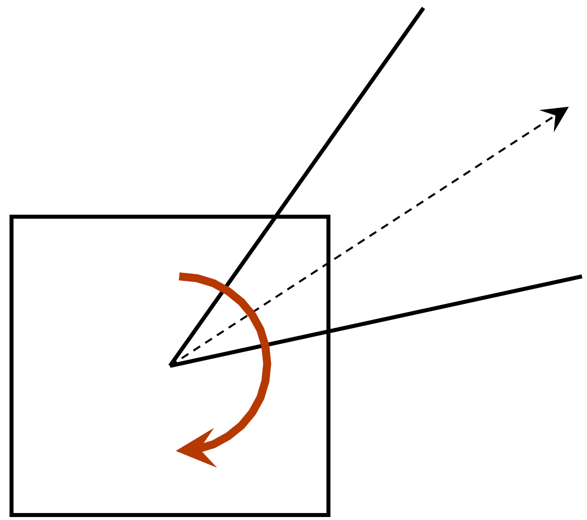
- Latency:



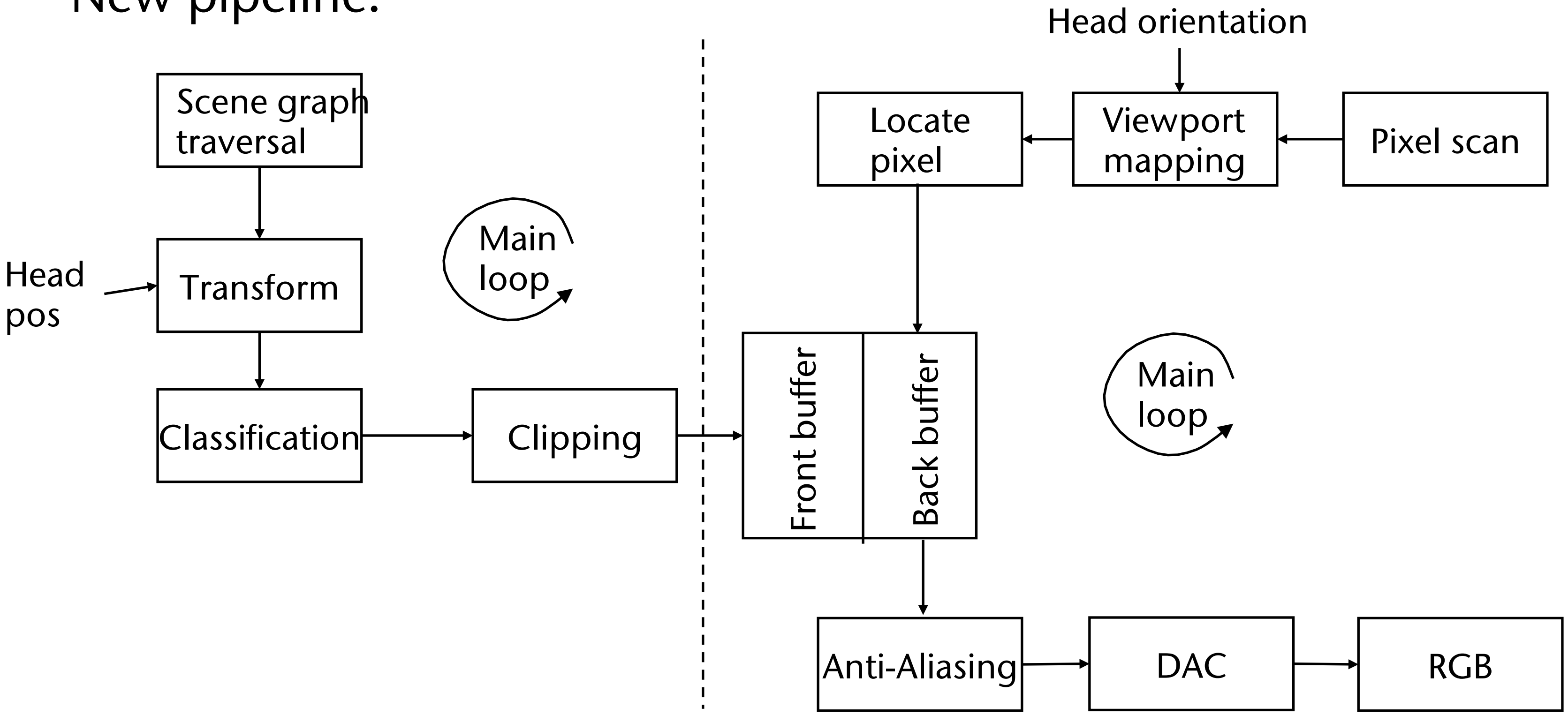
- Idea: render more than one viewport

Viewport Independent Rendering

- Conceptional idea:
  - Render the scene onto a *sphere* around the viewer
  - If viewpoint rotates: just determine new cutout of the spherical viewport
- Practical implementation: use cube as a viewport around user, instead of sphere (see also Cave)



- New pipeline:



- Latency:

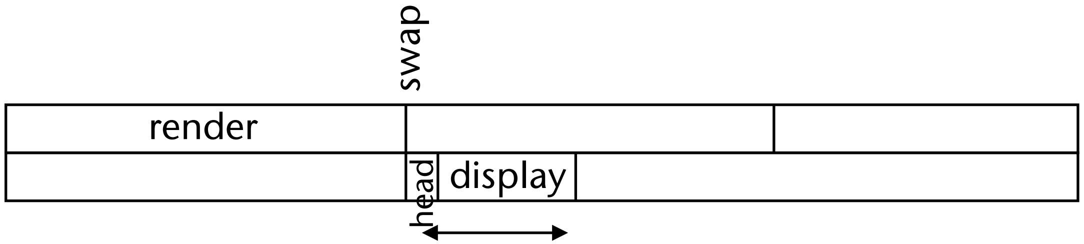
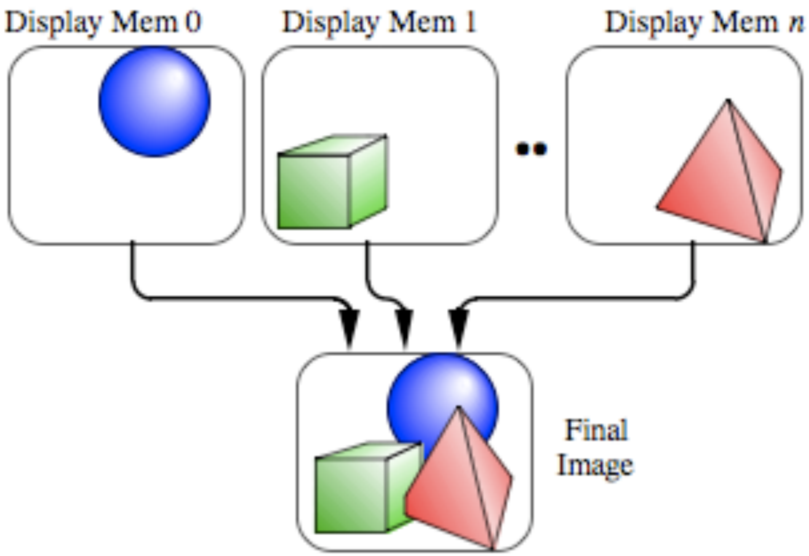


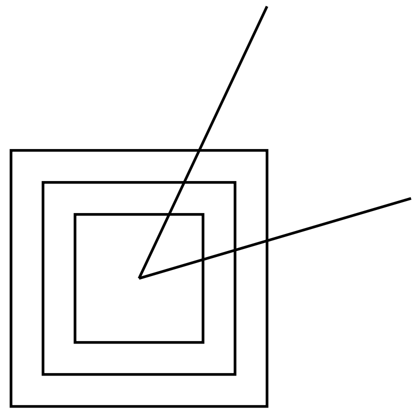
Image Composition

- Conceptional idea:
  - Each thread renders only "its own" object in its own framebuffer
  - Video hardware reads framebuffer *including* Z-buffer
  - Image compositor combines individual images by comparing Z per pixel
- In praxi:
  - Partition set of objects
  - Render each subset on one PC



Another technique: *Prioritized Rendering*

- Observation: images of objects far away from viewpoint (or slow relative to viewpoint) change slowly
- Idea: render onto several cuboid viewport "shells" around user
  - Fastest objects on innermost shell, slowest/distant objects on outer shell
  - Re-render innermost shell very often, outermost very rarely
- How many shells must be re-rendered depends on:
  - Framerate required by application
  - Complexity of scene
  - Speed of viewpoint
  - Speed of objects (relative to viewpoint)
- Human factors have influence on priority, too:
  - Head cannot turn by 180° in one frame → objects "behind" must be updated only rarely
  - Objects being manipulated must have highest priority
  - Objects in peripheral field of vision can be updated less often



Constant Framerate by "Omitting"

- Reasons for a constant framerate:
  - Prediction in *predictive filtering* of tracking data of head/hands works only, if all subsequent stages in the pipeline run at a known (constant) rate
  - Jumps in framerate (e.g., from 60 to 30 Hz) are very noticeable
- Rendering is "*time-critical computing*":
  - Rendering gets a certain time budget (e.g., 17 msec)
  - Rendering algorithm has to produce an image "as good as possible"
- Techniques for "*Omitting*" stuff:
  - Levels-of-Detail (LODs)
  - Omit invisible geometry (Culling)
  - *Image-based rendering*
  - Reduce the *lighting model*, reduce amount of textures,
  - … ?

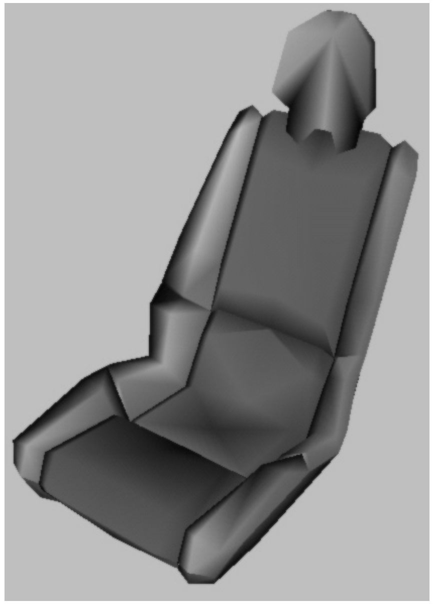
The Level-of-Detail Technique

- Definition:

  A level-of-detail (LOD) of an object is a reduced version, i.e. that has less polygons.

- Example:

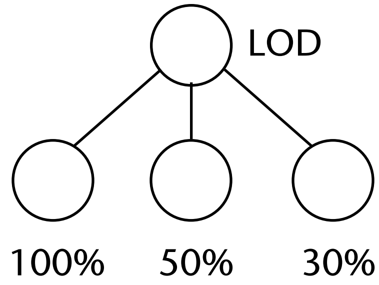




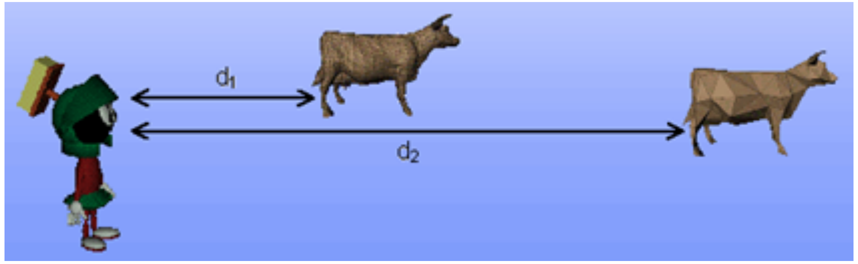
- Idea: render that LOD that fits the distance from the viewpoint, i.e., where users can't see the difference from the full-res. Version

- The technique consists of two tasks:
  1. Preprocessing: for each object in the scene, generate *k* LODs
  2. Runtime: select the "right" LODs, make switch unnoticeable

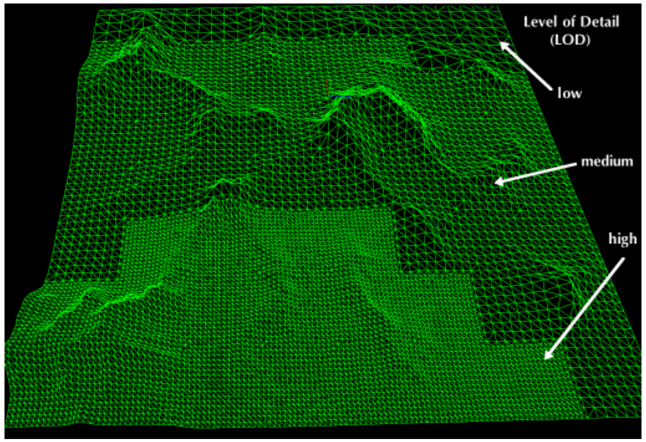
Selection of LOD

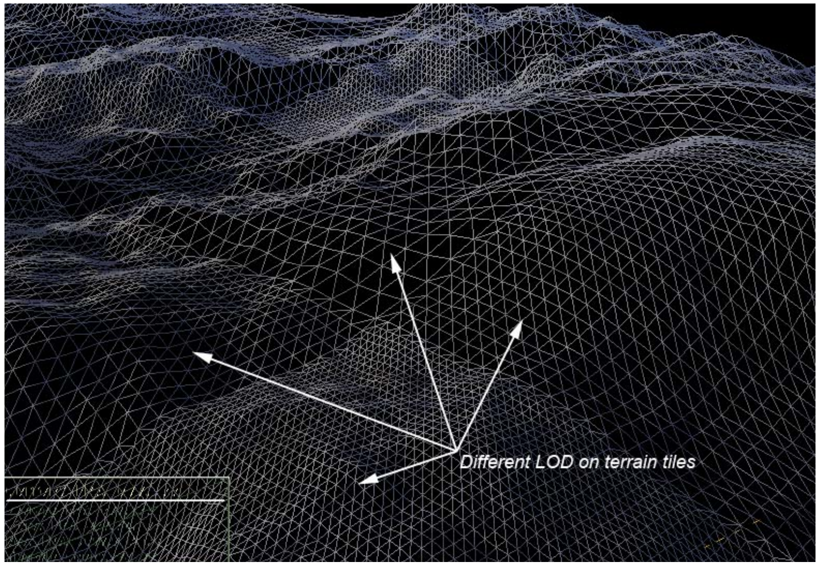


- Visual quality against "temporal quality"
- Static selection algorithm:
  - Level i for a distance range (d_i, d_{i+1})
  - Depends on FoV
  - Problem: size of objects is not considered

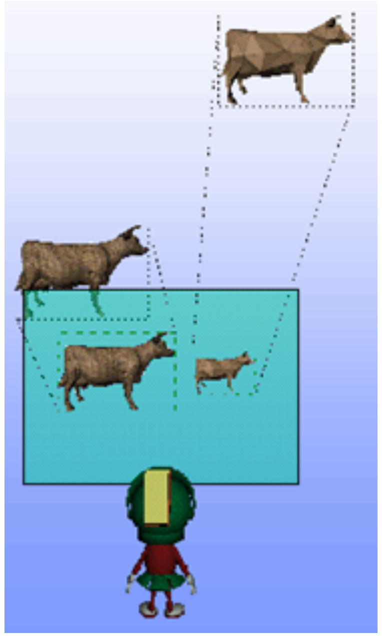


- For some desktop applications, e.g. terrain rendering, this is already sufficient:



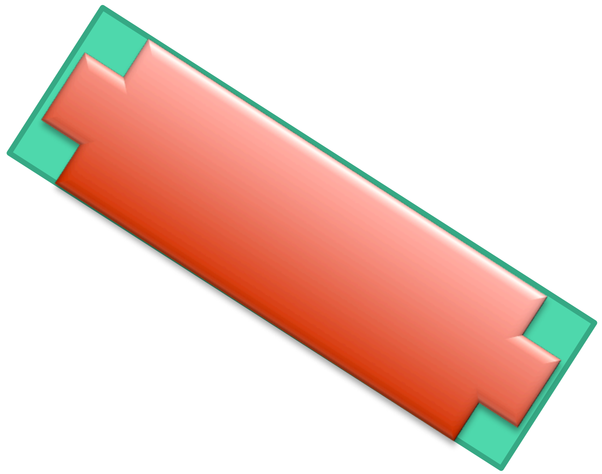


- Dynamic selection algorithm:
  - Estimate size of object on the screen
  - Advantage: independent from screen resolution, FoV, size of objects
  - LOD depends on distance *automatically*



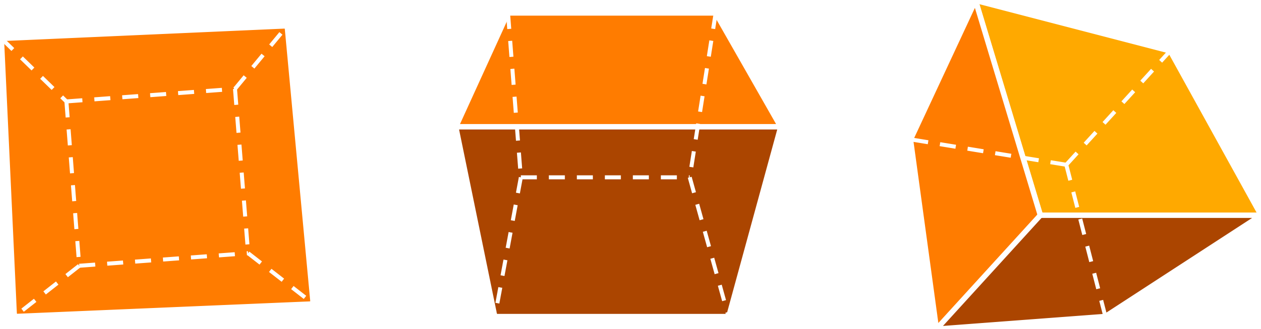
Estimation of Size of Object on the Screen

- Naïve method:
  - Compute bounding box (bbox) of object in 3D (probably already known)
  - Project bbox in 2D → 8x 2D points
  - Compute 2D bbox (axis aligned) around 8 points
- Better method:
  - Compute true area of projected 3D bbox on screen

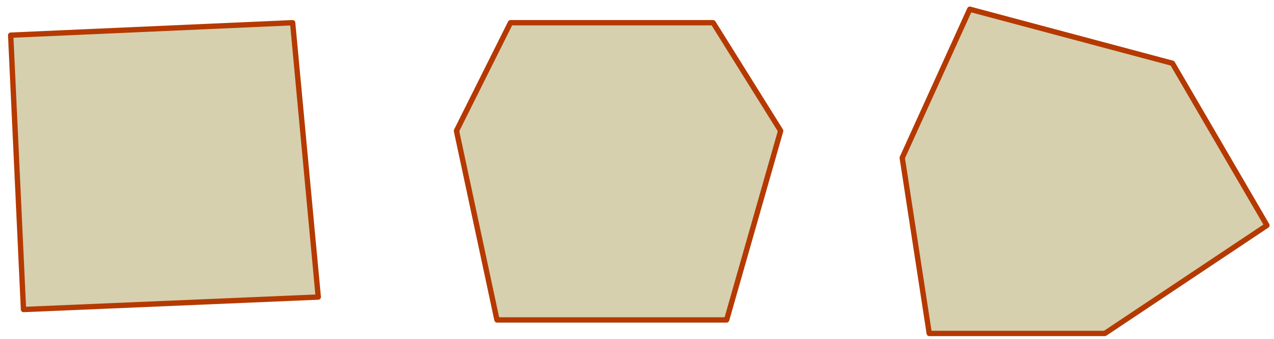


Idea of the Algorithm

- Determine number of sides of 3D bbox that are visible:



- Project only points on the silhouette (4 oder 6) in 2D:

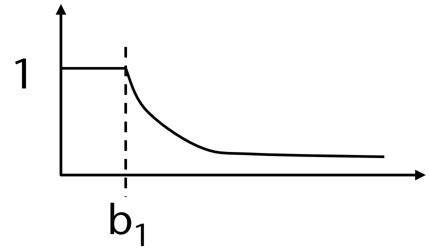


- Compute area of this (convex!) polygon

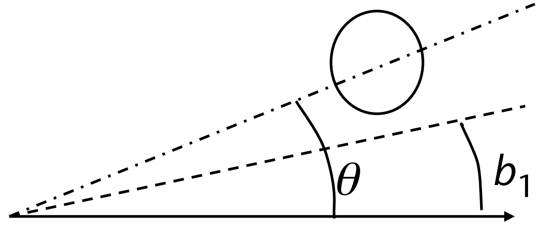
Implementation

- For each pair of (parallel) box sides (i.e., each *slab*):
  classify viewpoint with respect to this pair into "below", "above", or "between"
- Yields 3x3x3 = 27 possibilities
  - In other words: the sides of a cube partition space into 27 subsets
- Utilize bit-codes (à la out-codes from clipping) and a lookup-table
  - Yields LUT with 2^6 entries (conceptually)
- 27-1 entries of the LUT list each the 4 or 6 vertices of the silhouette
- Then, project, triangulate (determined by each casein LUT), accumulate areas

Psychophysiological LOD Selection

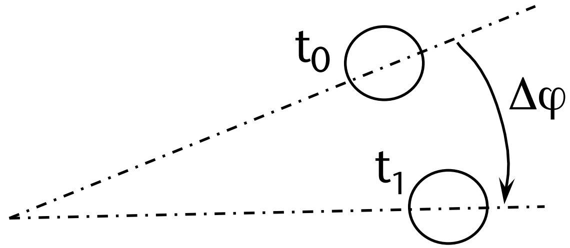


- Idea: exploit human facors with respect to visual accuity:
  - Central / peripheral vision:



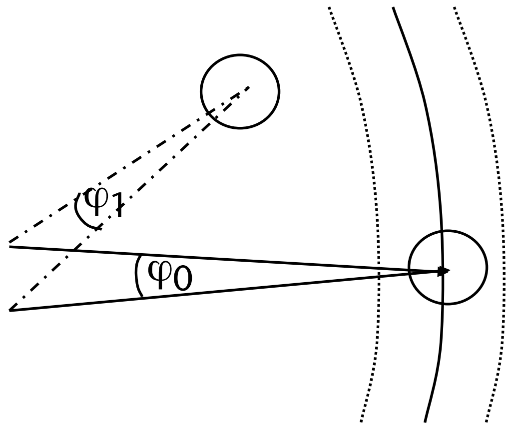
$$k_1 = \begin{cases} e^{-(\theta - b_1)/c_1} & , \theta > b_1 \\ 1 & , \text{sonst} \end{cases}$$

  - Motion of obj (relative to viewpoint):



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

  - Depth of obj (relative to horopter):



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

- Determination of LODs:
  1. $k = \min\{k_i\} \cdot k_0$,   oder $k = \prod k_i \cdot k_0$
  2. $r_{\text{min}} = 1/k$
  3. Select level l such that

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

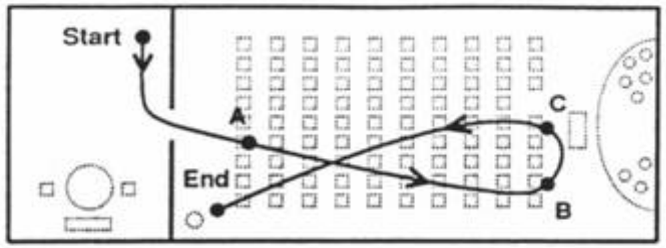
  where P_l is the set of polygons of level l of an object

- Do we need *eye tracking* for this to work?
  - Disadvantages of eye tracking: expensive, imprecise, "*intrusive*"
  - Psychophysiology: eyes always deviate < 15° from head direction
  - So, assume eye direction = head direction, and choose $b_1 = 15°$

Reactive vs. Predictive LOD Selection

- Reactive LOD selection:
  - Keep history of rendering durations
  - Estimate duration T_r for next frame
  - Let T_b = time budget that can be spent for next frame
  - If $T_r > T_b$: decrease LODs (use coarser levels)
  - If $T_r < T_b$: increase LODs (finer levels)
  - Then, render frame and record time duration in history

- Reactive LOD selection can produce severe outliers
- Example scenario:



Predicitive LOD Selection

[Funkhouser und Sequin]

- Definition object tuple (O,L,R):

  O = object, L = level,

  R = rendering algo (#textures, anti-aliasing, #light sources)

- Evaluation functions on object tuples:

  Cost(O,L,R) = time needed for rendering

  Benefit(O,L,R) = "contribution to image"

- Optimization problem:

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

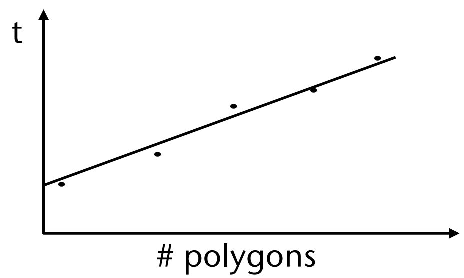
  under the condition $$T_r = \sum_{(O,L,R) \in S'} \text{cost}(O, L, R) \leq T_b$$

  where S = { mögliche Objekt-Tupel in der Szene }

- Cost-Funktion depends on:
  - Number of vertices (~ *# coord. transforms* + *lighting calcs* + *clipping*)
  - Setup per polygon
  - Number of pixels (*scanline conversions, alpha blending, textur fetching, anti-aliasing, phong shading*)
  - Theoretical cost model:

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

- Better determine the cost function by experiments: Render a number of different objects with all different parameter settings possible



- Benefit funktion: "contribution" to image is affected by
  - Size of object
  - Shading method:

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

  - Distance from center (periphery, depth)
  - Velocity
  - Semantic "importance" (e.g., grasped objects are very important)
  - Hysteresis for penalizing LOD switches:

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

  - Together:

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

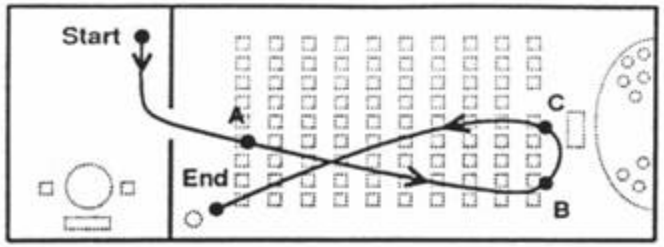
- Optimization problem = "*multiple-choice knapsack problem*"
  $\rightarrow$ NP-complete
- Idea: compute sub-optimal solution:
  - Reduce it to continuous knapsack problem (see algorithms class)
  - Solve it greedily with one *additional* constraint
  - Define

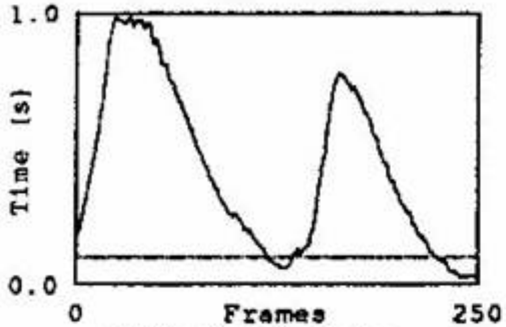
$$\text{value}(O, L, R) = \frac{\text{benefit}(O, L, R)}{\text{cost}(O, L, R)}$$

  - Sort all object tuples by value(O,L,R)
  - Choose the first k tuples until knapsack is full
  - Constraint: no 2 object tuples must represent the same object

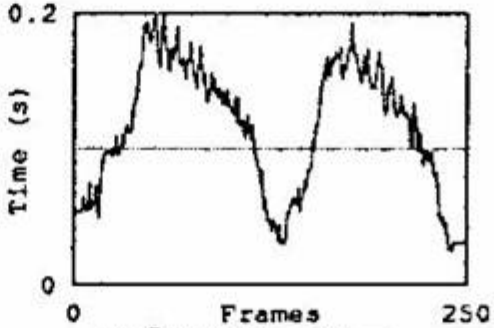
- Incremenal solution:
  - Start with solution $(O_1, L_{1,1}), \ldots, (O_n, L_n, R_n)$ as of last frame
  - If
    $$\sum_i \text{cost}(O_i, L_i, R_i) \leq \text{max. frame time}$$
    then find object tuple (O_k, L_k, R_k) ,
    such that
    $$\text{value}(O_k, L_k + a, R_k + b) - \text{value}(O_k, L_k, R_k) = \max$$
    and
    $$\sum_{i \neq k} \text{cost}(O_i, L_i, R_i) + \text{cost}(O_k, L_k + a, R_k + b) \leq \text{max. frame time}$$
- Analog, falls $\sum_i \text{cost}(O_i, L_i, R_i) > \text{max. frame time}$

- Performance in the example scenes:

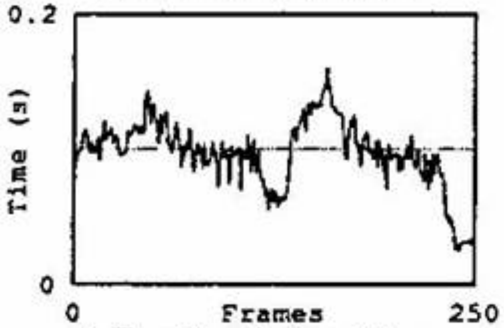




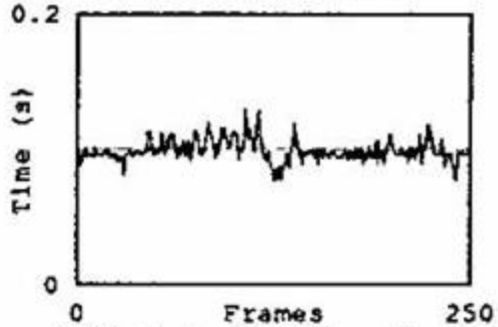
a) No detail elision.



b) *Static* algorithm.



c) *Feedback* algorithm.



d) *Optimization* algorithm.

Screenshots from the Example Scenes

No detail elision, 19,821 polygons

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

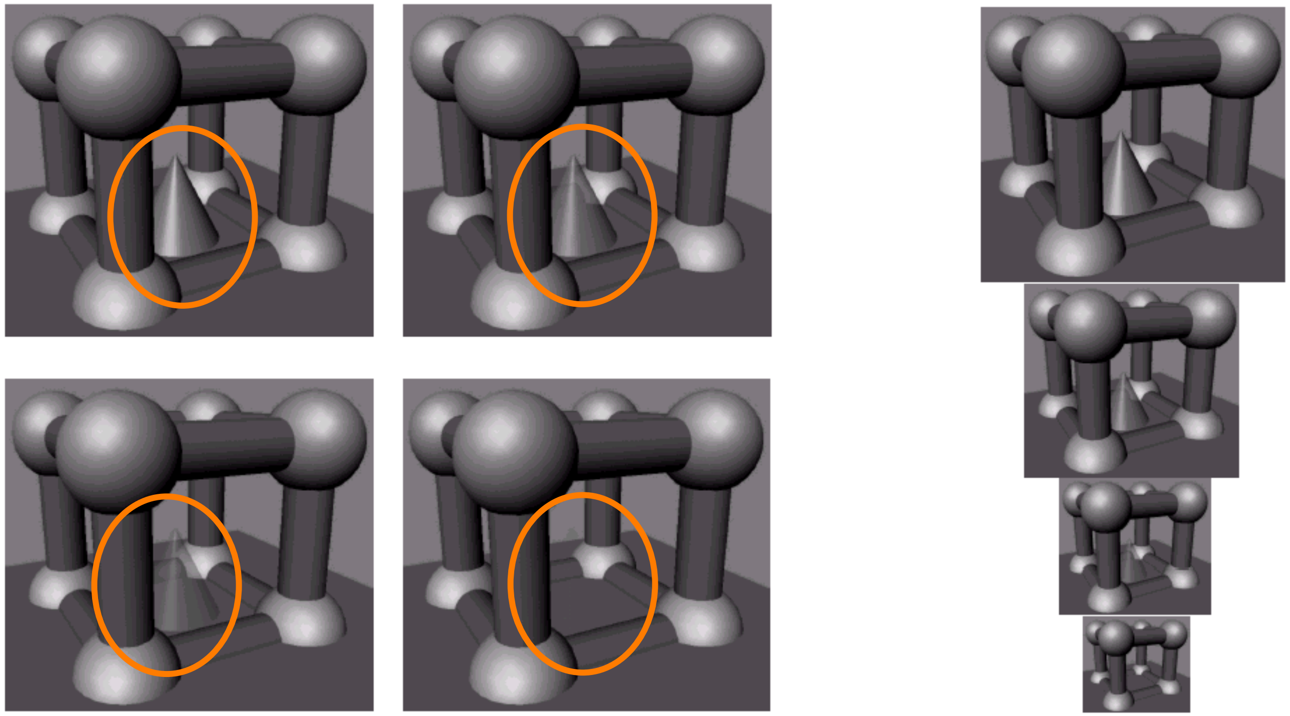
Level of detail: darker
gray means more detail

Problem with Discrete LODs

- "Popping" when switching to next higher/lower level
- Measures against "popping":
  - Hysteresis (just reduces the frequency of pops a little bit)
  - Alpha blending of the two adjacent LOD levels
    - Man kommt vom Regen in die Traufe ;-)
  - Continuous, view-dependent LODs

Alpha-LODs

- Simple idea to avoid popping:
  when beyond a certain range, fade out level *i* until gone,
  at the same time fade in level *i*+1



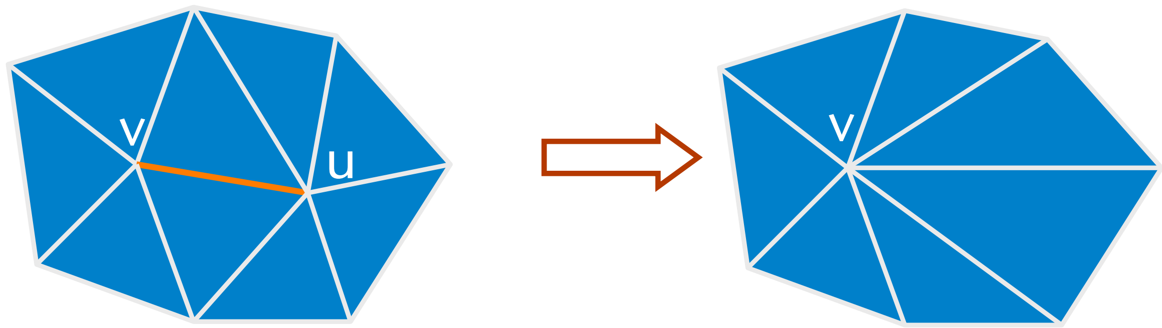
Progressive Meshes

- A.k.a. Geomorph-LODs
- Initial idea / goal:
  - Given two meshes M_i and M_{i+1} (LODs of the same object)
  - Construct mesh M' "in-between" M_i and M_{i+1}
- In the following, we will do more

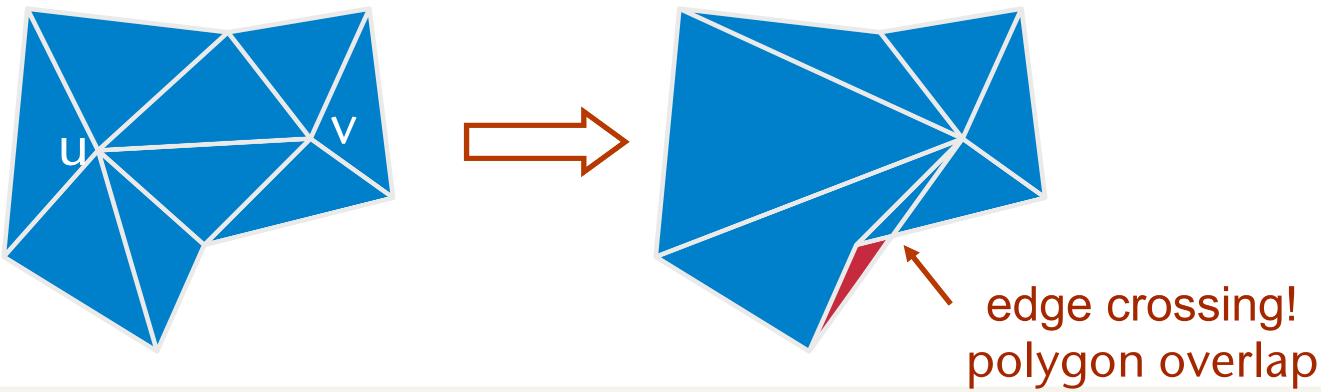
- Definition: Progressive Mesh = representation of an object, starting with a high-resolution mesh M_0, with which one can continuously (up to the edge level) generate "in-between" meshes ranging from 1 polygon up to M_0 (and do that extremely fast).

Construction of Progressive Meshes

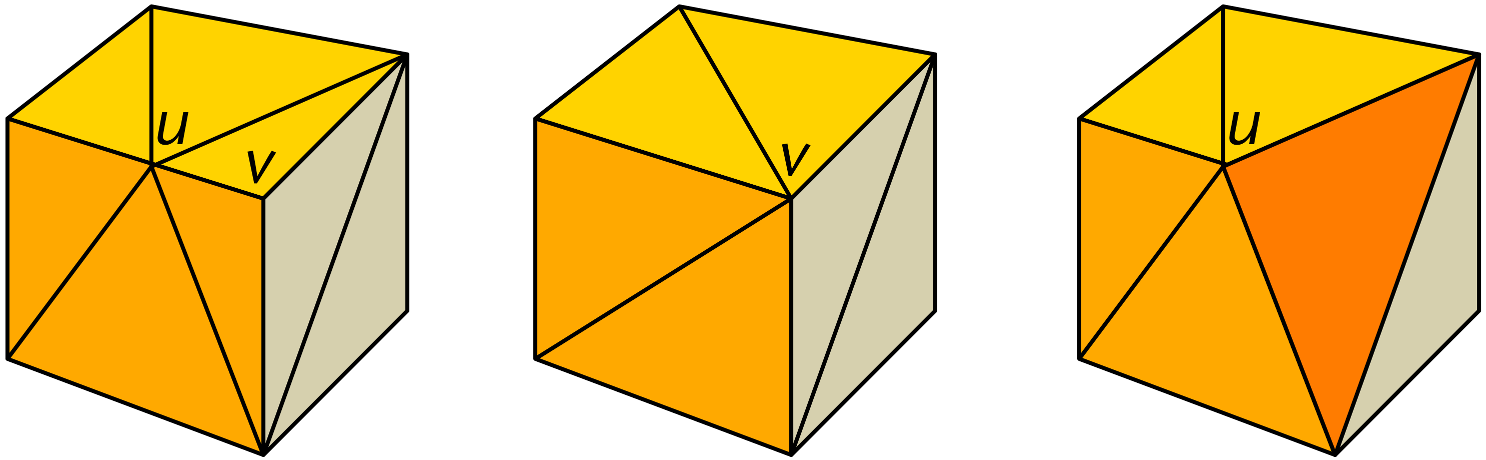
- Approach: successive *simplification*, until only 1 polygon left
- The fundamental opetration: *edge collapse*



  - Reverse operation = vertex split
- Not every edge can be chosen: bad edge collapses



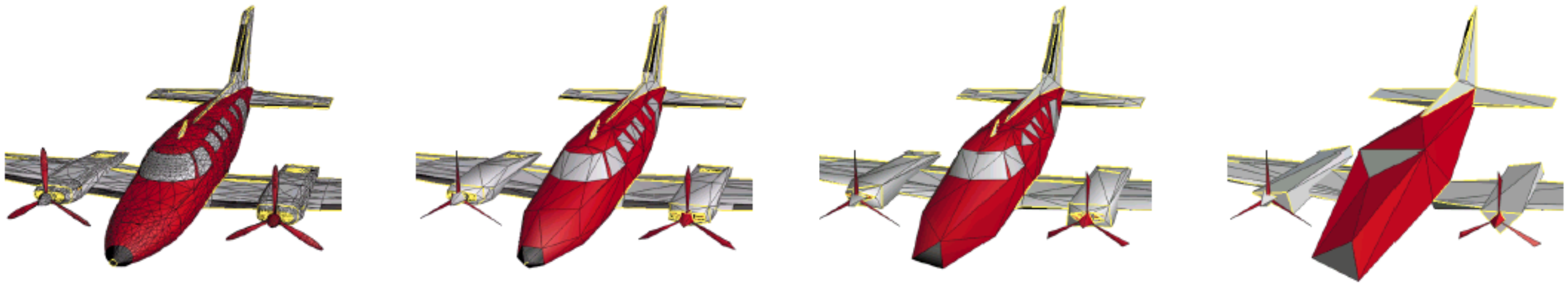
- The order of edge collapses is important:



  - Introduce measure on edge collapses, in order to evaluate "visual effect"

- Goal: perform first edge collapses that have the least visual effect
- Remark: after every edge collapse, all remaining edges need to be evaluated again, because their "visual effect" (if collapsed) might be different now

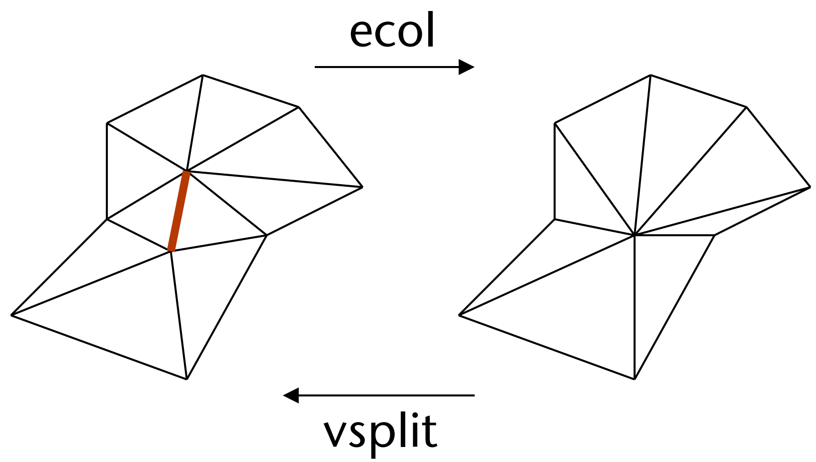
- Evaluation function for edge collapses is not trivial and, more importantly, perception-based!
- Factors influencing "visual effect":
  - Curvature of edge / surface
  - Lighting, texturing, viewpoint (highlights!)
  - Semantics of the geometry (eyes & mouth are very important in faces)
- Examples of a progressive mesh:



- Representation of a progressive meshes:

$$M = M^n \underset{\text{vsplit}_{n-1}}{\overset{\text{ecol}_{n-1}}{\longleftrightarrow}} \cdots \underset{\text{vsplit}_1}{\overset{\text{ecol}_1}{\longleftrightarrow}} M^1 \underset{\text{vsplit}_0}{\overset{\text{ecol}_0}{\longleftrightarrow}} M^0$$

  - M^{i+1} = i-th refinement = 1 vertex more than M^i



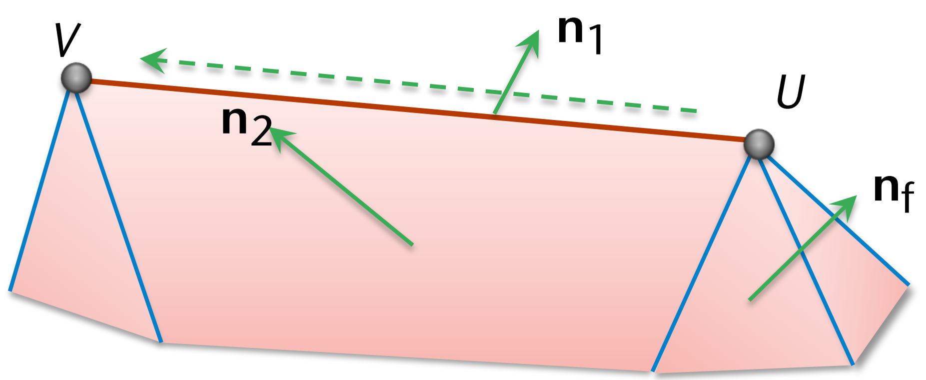
- Representation of an edge collapse / vertex split:
  - Edge (= pair of vertices) affected by the collapse/split
  - Position of the "new" vertex
  - Triangles that need to be deleted / inserted

Example for a Simple Edge Evaluation Function

- Follow this heuristic:
  - Delete small edges first
  - Move vertex *U* onto vertex *V*, if surface incident to *U* has smaller (discrete) curvature than surface around *V*
- A simple measure for an edge collapse from *U* onto *V*:

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

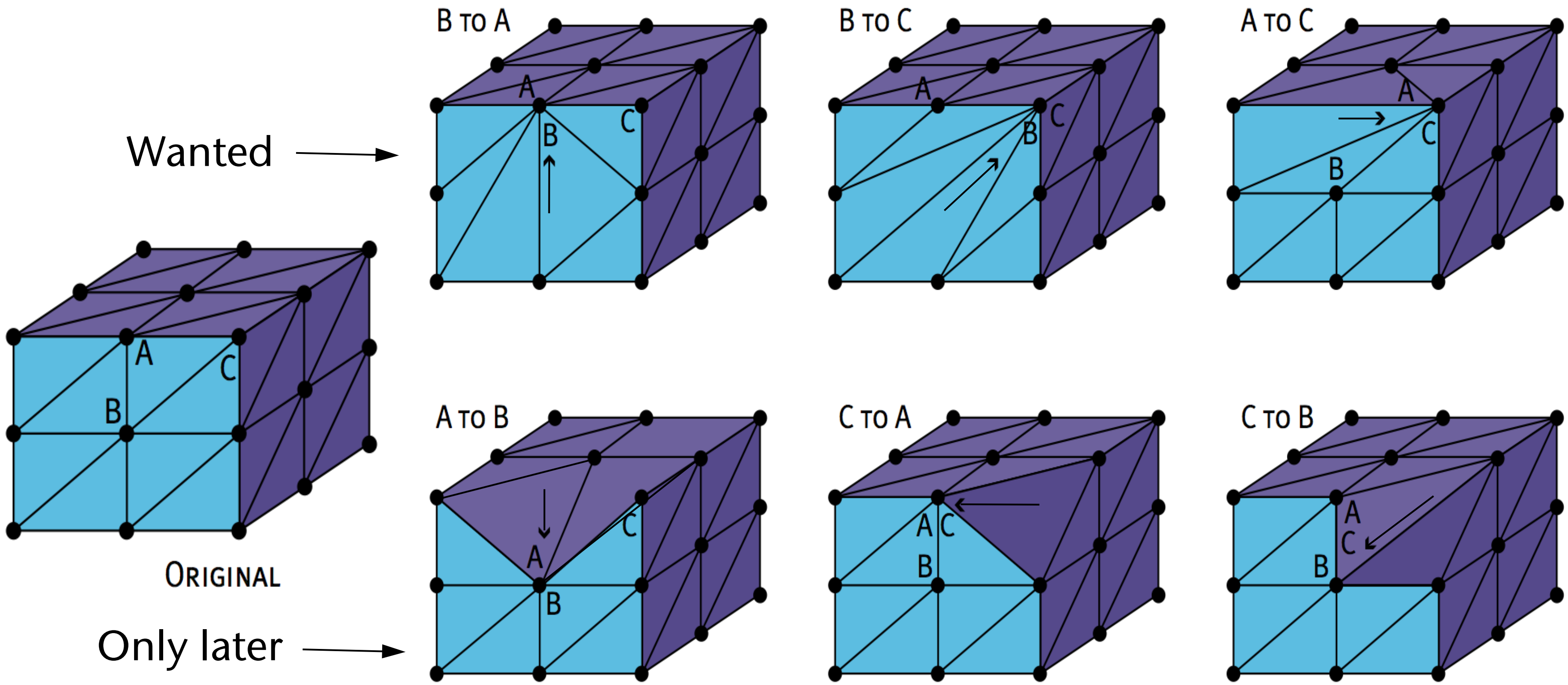
$$\text{curv}(U) = \frac{1}{2}\left(1 - \min_{f \in T(U) \setminus T(V)} \max_{i=1,2} \mathbf{n}_f \mathbf{n}_i\right)$$



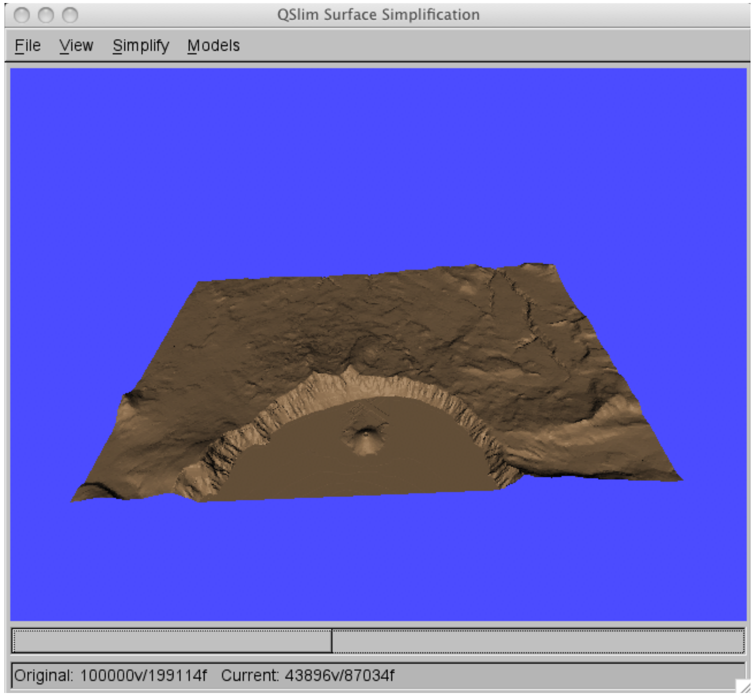
- Remark:

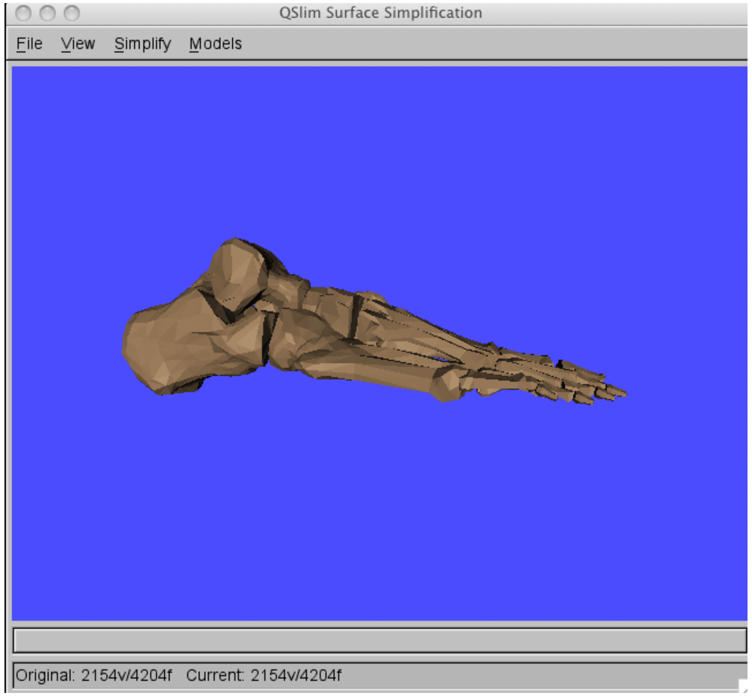
$$\text{cost}(U, V) \neq \text{cost}(V, U)$$

- Example:



Demo





[Michael Garland: Qslim]

How can the Funkhouser-Sequin algorithms be combined with progressiven meshes?

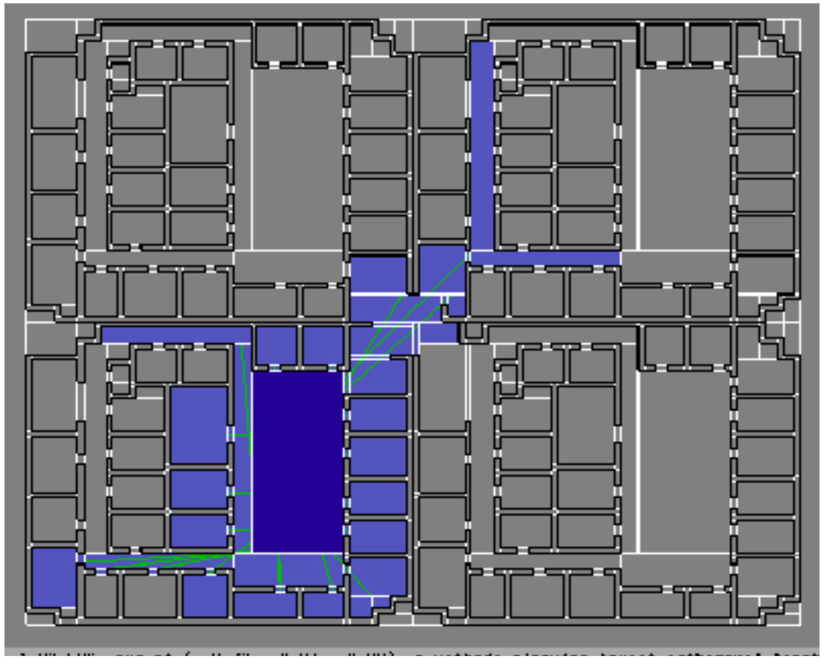
Diplomarbeit ...

Digression: other Kinds of LODs

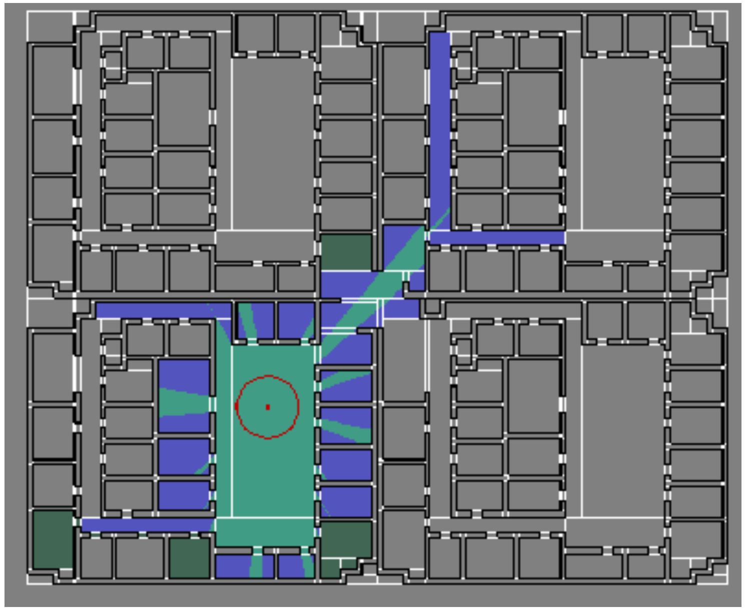
- Idea: apply LOD technique to other non-geometric content
- E.g. "*behavioral LOD*":
  - Simulate the behavior of an object exactly if in focus, otherwise simulate it only "approximately"

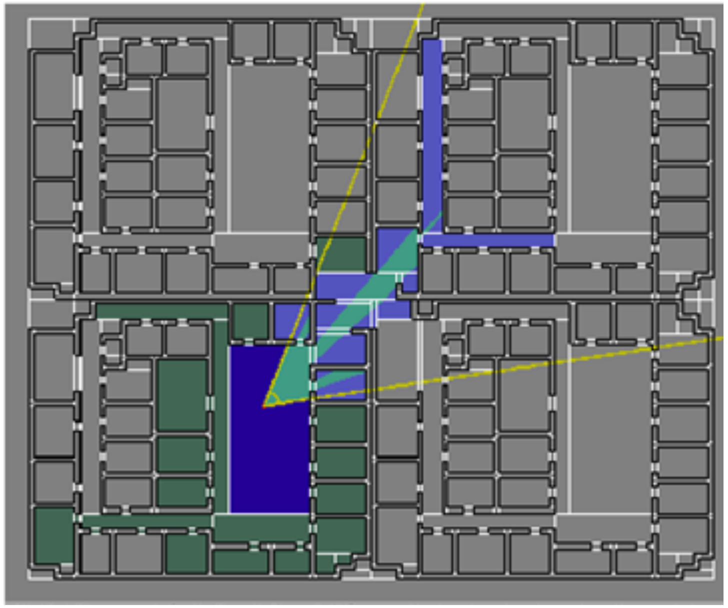
Culling in Buildings (Portal Culling)

- Observation: many rooms within the viewing frustum are not visible
- Idea:
  - Partition the VE into "cells"
  - Precompute *cell-to-cell-visibility* → visibility graph



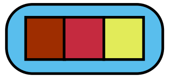
- During runtime, filter cells from visibility graph by viewpoint and viewing frustum:



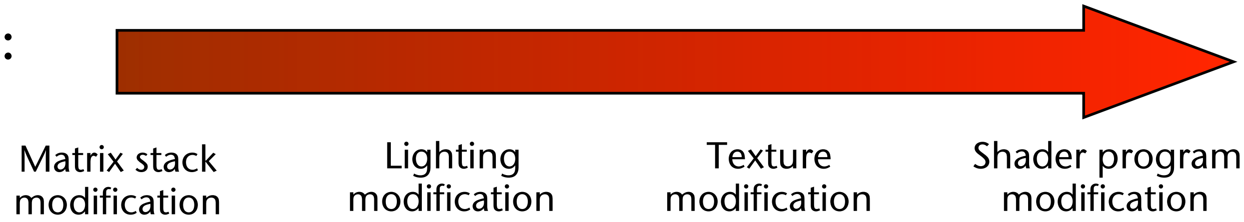


State Sorting

- State in OpenGL rendering =
  - Combination of all attributes
  - Examples for attributes: color, material, lighting parameters, number of textures being used, shader program, etc.
  - At any time, each attribute has exactly 1 value out of a set of possible attributes (e.g., color ∈ { (0,0,0), ..., (255,255,255) }



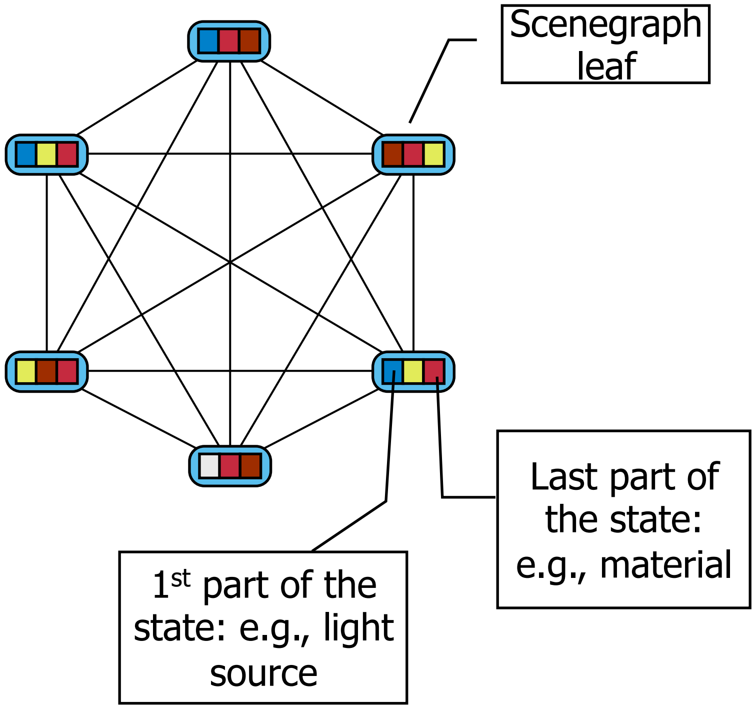
- State changes are a serious performance killer!
- Costs:



- Goal: render complete scene graph with *minimal* number of state changes
- "Solution": pre-sorting

- Problem: optimal solution is NP-complete
- Reason:
  - Each leaf of the scene graph can be regarded a node in a complete graph
  - Costs of an edge = costs of the corresponding state change (different state changes cost differently, e.g., changing the transform is cheap)
  - Wanted: shortest path through graph

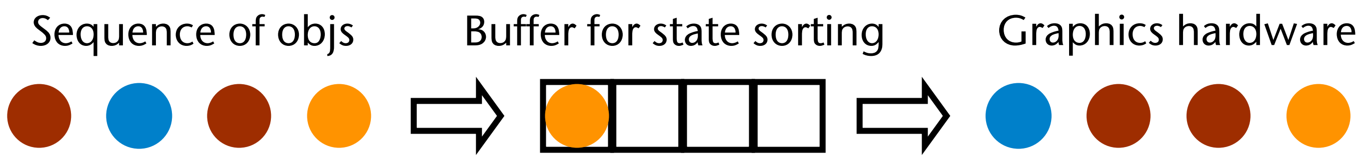
  → *Traveling Salesman Problem*



- Further problem: precomputation doesn't work with dynamic scenes and occlusion culling

Introducing the Sorting Buffer

- Idea & abstraction:
  - For sake of argument: just consider 1 attribute ("color")
  - Introduce buffer between application and graphics card
    - (Could be incorporated into driver / hardware, since an OpenGL command buffer is already in place)
  - Buffer contains elements with different colors
  - With each rendering step (=app sends "colored element" to hardware/ buffer), perform one of 3 operations:
    1. Pass element directly on to graphics hardware
    2. Store element in buffer
    3. Extract subset of elements from buffer and send them to graphics hardware



A Special Class of Algorithms

- There are 2 categories of algorithms:
  - "Online" algorithms: algo does *not* know elements that will be received in the future!
  - "Offline" algorithms: Algo *does* kow elements that will be received in the future (for a fair comparison, it still has to store/extract them in a buffer, but it can utilize its knowledge of the future to decide whether to store it)
- In the following, we consider w.l.o.g only the "lazy" online strategy:
  - Extract elements from the buffer only in case of buffer overflow
  - Because every non-lazy online strategy can be converted into a lazy online strategy with same complexity (= costs)
- Question in our case: which elements should be extracted from the buffer (in case of buffer overflow), so that we achieve the minimal number of color changes?

Competitive Analysis

- Definition *c-competitive* :

  Let $C_{\text{off}}(k)$ = Costs (= number of color changes) of optimal offline strategy, k = buffer size.

  Let $C_{\text{on}}(k)$ = costs of some online strategy.

  Then, this strategy is called "c-competitive" iff

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

  where a must not depend on k.

  The ratio

  $$\frac{C_{\text{on}}(k)}{C_{\text{off}}(k)} \approx c$$

  is called the competitive-ratio.

- Wanted: an online strategy with a c as small as possible (in the worst-case, and — more importantly — in the average case)

Bremen

Example: LRU strategy (least-recently used)

- The strategy:
  - Maintain a timestamp per color (not per element!)
  - An element gets stored in buffer → timestamp of its color is set to current time
    - Notice: timestamps of other elements in buffer can change, too
  - Buffer overflow → extract elements, whose color has oldest timestamp
- The lower bound on the competitive-ratio: $\Omega\left(\sqrt{k}\right)$
- Proof by example:
  - Set $m = \sqrt{k-1}$, w.l.o.g. m even
  - Choose the input $\left(c_1 \cdots c_m x^k c_1 \cdots c_m y^k\right)^{\frac{m}{2}}$
  - Costs of the online LRU strategy: $\left(m+1\right) \cdot 2 \cdot \frac{m}{2}$ color changes
  - Costs of the offline strategy: $2m$ color changes, because its output is = $\left(x^k y^k\right)^{\frac{m}{2}} c_1^m \cdots c_m^m$

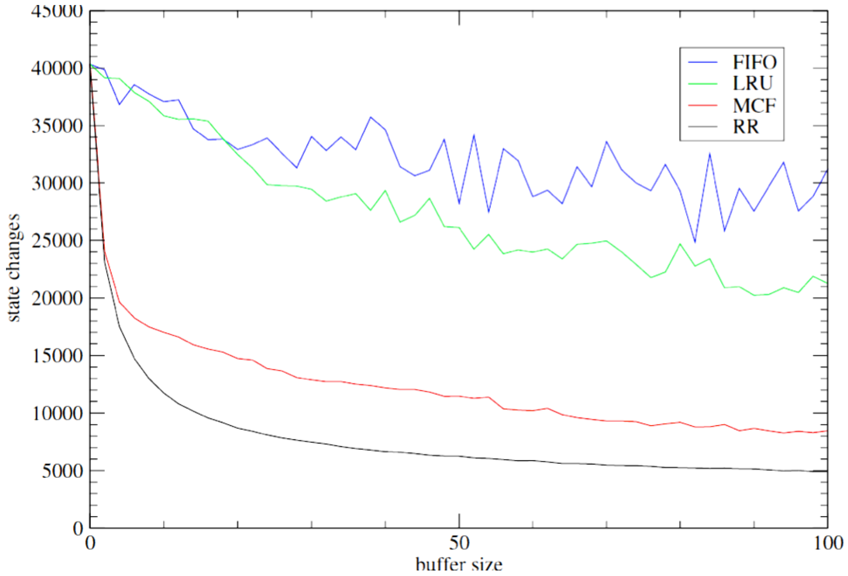
The Bounded Waste & the Random Choice Strategy

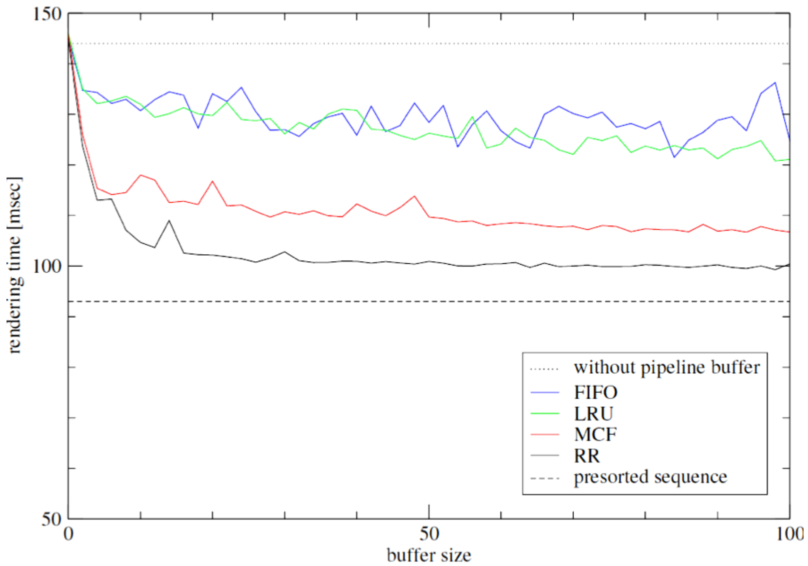
- Idea:
  - Count the number of all elements that have the same color
  - Extract those elements whose color is most prevalent in the buffer
- Introduce waste counter W(c) :
  - With color change on input side: increment W(c)
- Bounded waste strategy:
  - With buffer overflow, extract all elements of color c', whose W(c') = max
- Competitive ratio (w/o proof): $O\left(\log^2 k\right)$
- Random choice strategy:
  - Randomized version of bounded waste strategy
  - Choose uniformly a random element in buffer, extract all elements with same color (most prevalent color in buffer has highest probability)
  - Consequence: more prevalent color gets chosen more often, over time each color gets chosen W(c) times

The Round Robin Strategy

- Problem: generation of good random numbers is fairly costly
- Round robin strategy:
  - Variant of random choice strategy
  - Don't choose a random slot in the buffer,
  - Instead, every time choose the *next* slot
  - Maintain pointer to current slot, move pointer to next slot every time a slot is chosen

- Take-home message:
  - Round-robin yields very good results (although very simple)
  - Worst case doesn't say too much about performance in real-world applications





Stereo *without* rendering 2x (simple image warping)

- Observation: left & right image differ not very much
- Idea: render 1x for right image, then move pixels for left image
- Algo: consider all pixels on each scanline *from right to left*, draw each pixel k at the new x-position

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

- Problems:
  - Holes!
  - Up vector must be vertical
  - Reflections and specular highlights are at wrong position
  - Aliasing

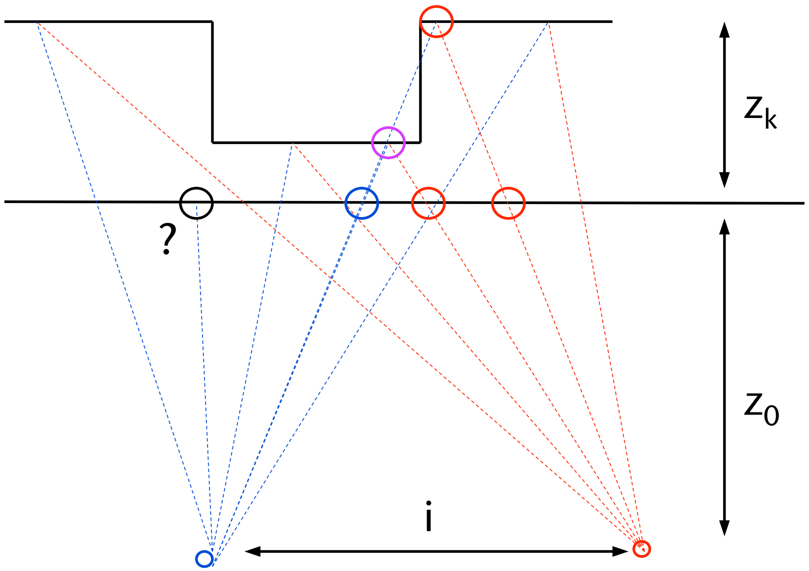
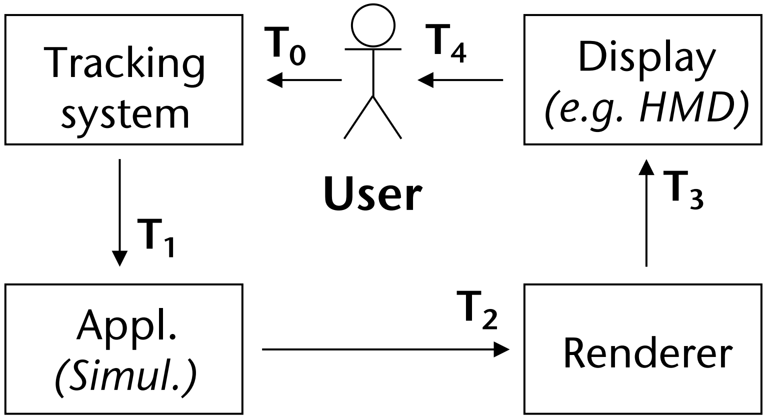
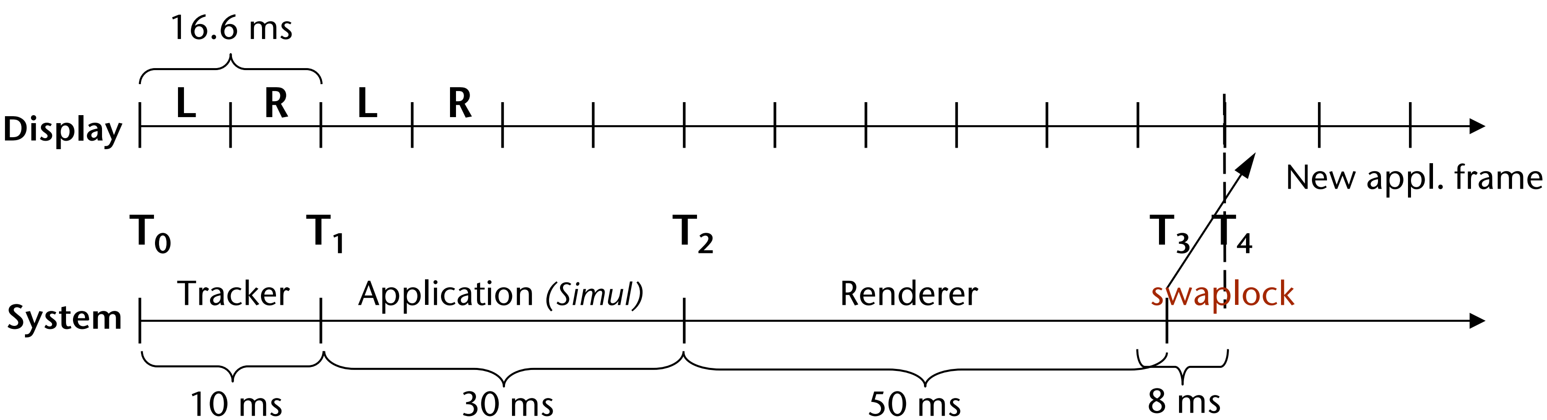


Image Warping

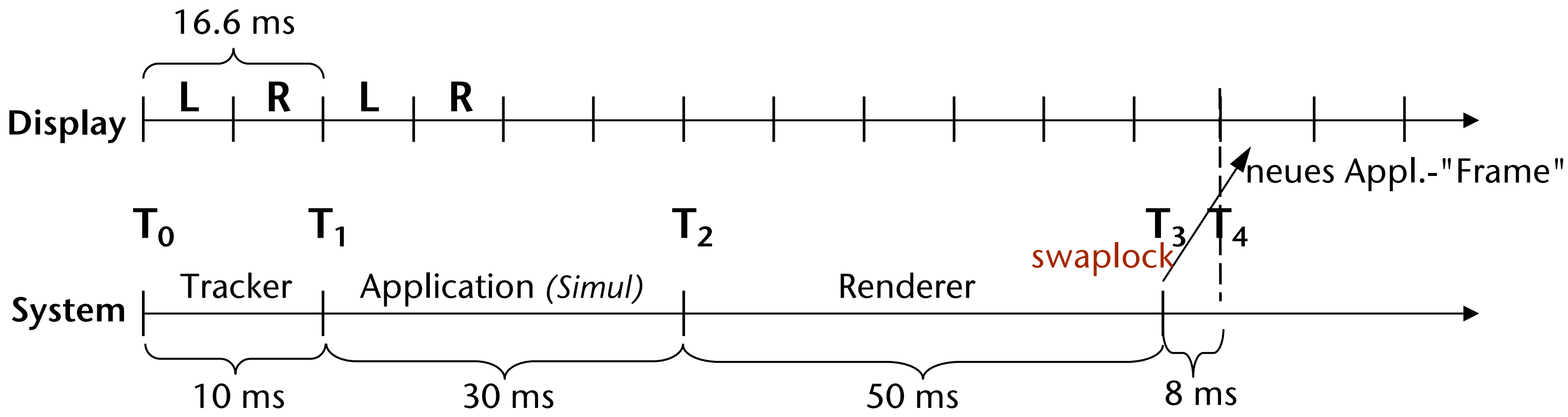
- A naïve VR system:



- Latency in this system (stereo with 60 Hz → display refresh = 120 Hz):



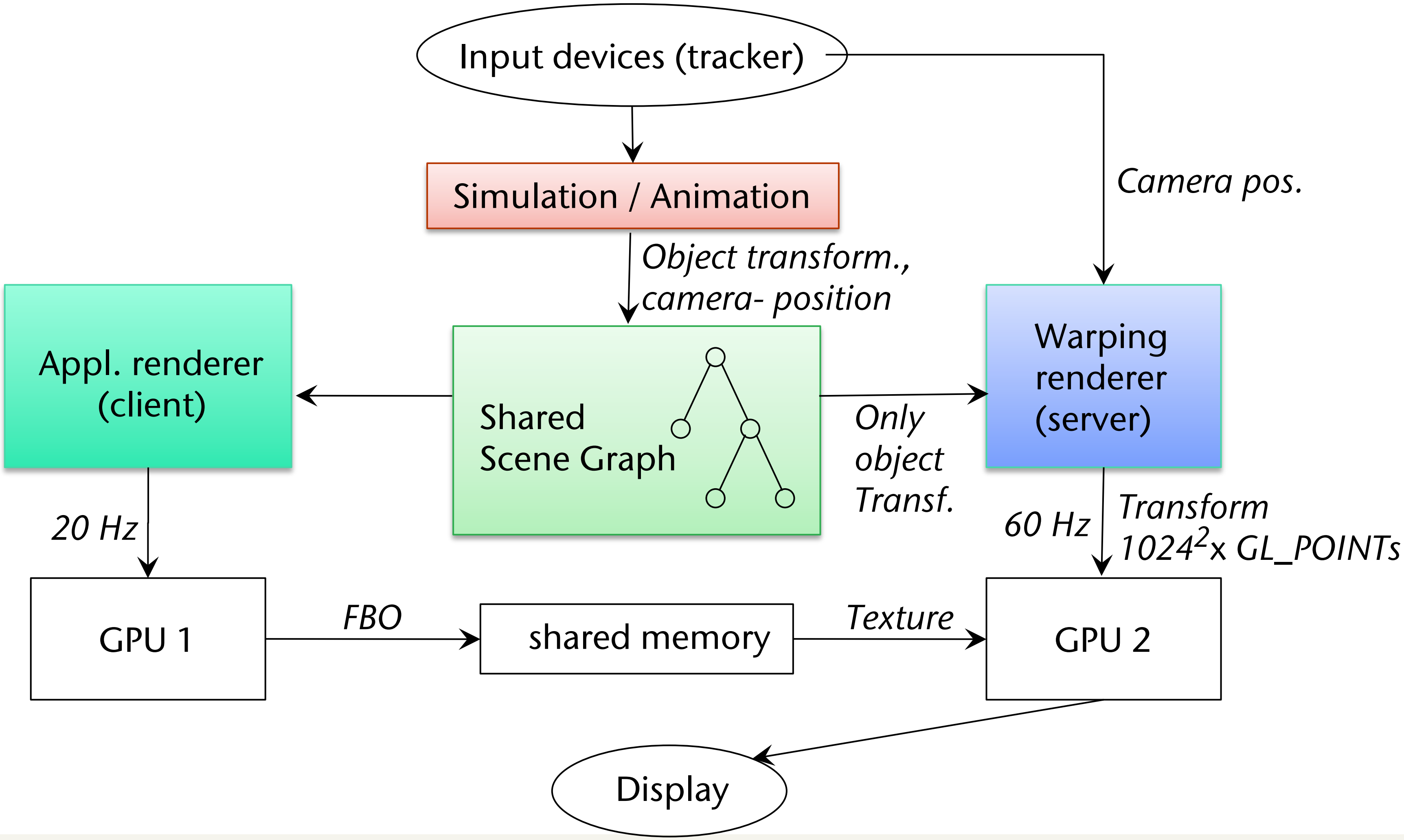
- Problems / observations:
  - The app. framerate (incl. rendering) is typically much slower than the display refresh rate
  - The tracking data, which led to a specific image, were valid in the distant past
  - The tracker could deliver data more often
  - Consecutive frames differ from each other (most of the time) only relatively little (→ temporal coherence)



Idea for a Solution

[2009]

- Decouple simulation/animation, rendering, and device polling:



An Application Frame (Client)

- At time t_1, the application renderer generates a normal frame
  - Color buffer and Z-buffer
- … but additionally saves some information:
  1. With each pixel, save ID of object visible at that pixel
  2. Camera transformations at time t_1

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

  3. With each object i, save its transformation

$$T^i_{t_1, obj \leftarrow wld}$$

Warping of a Frame (Server)

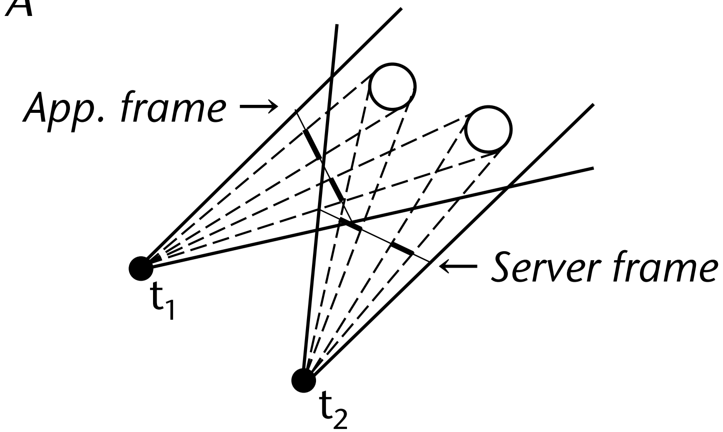
- At a later time t_2, the server generates an image from an application frame by warping
- Transformations at this time:

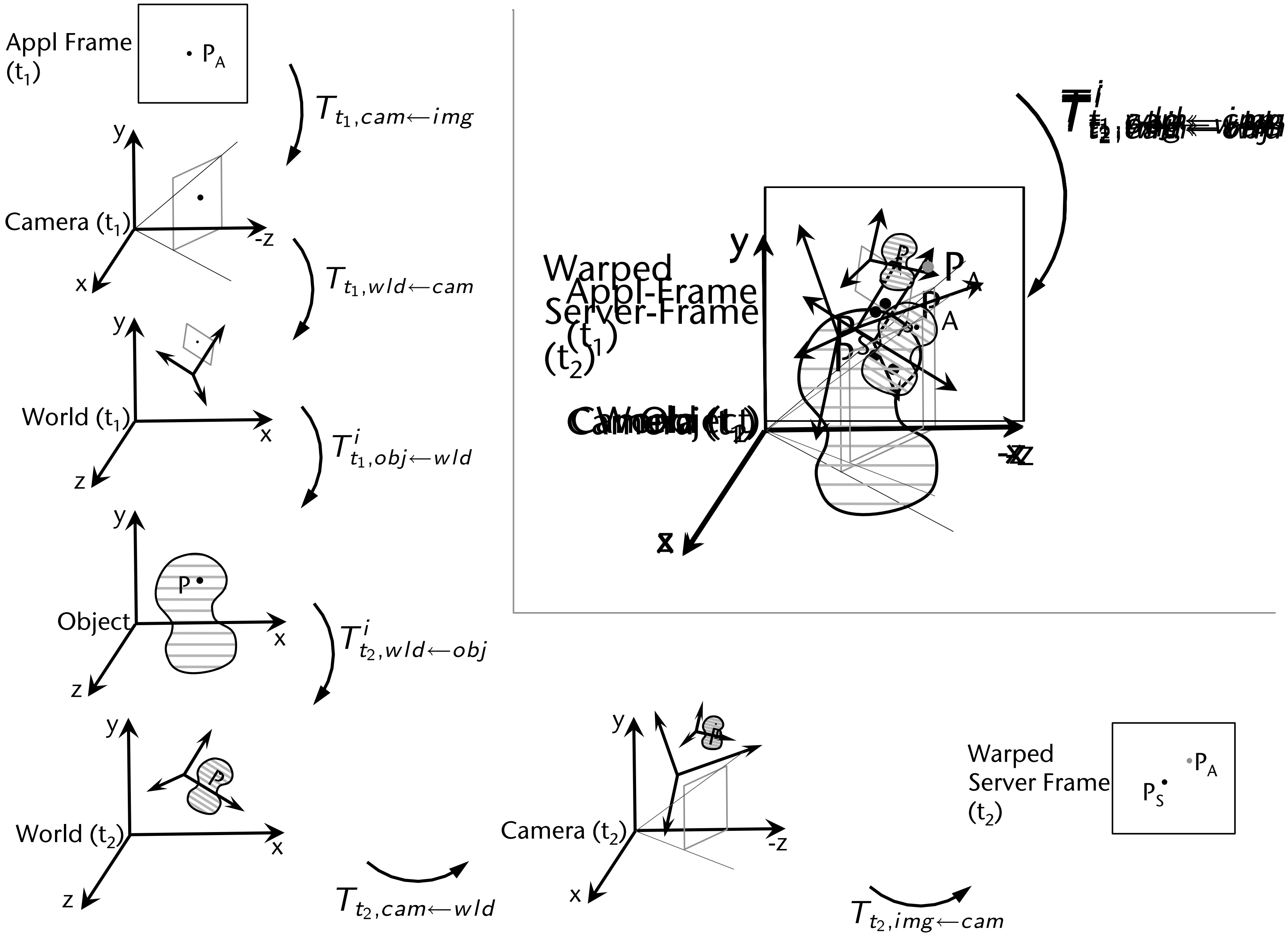
$$T^i_{t_2,wld \leftarrow obj} \qquad T_{t_2,img \leftarrow cam} \qquad T_{t_2,cam \leftarrow wld}$$

- A pixel $P_A = (x, y, z)$ in the app. frame will be "warped" to its correct position in the (new) server frame:

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

- This transform. matrix can be precomputed for each object with each new server frame

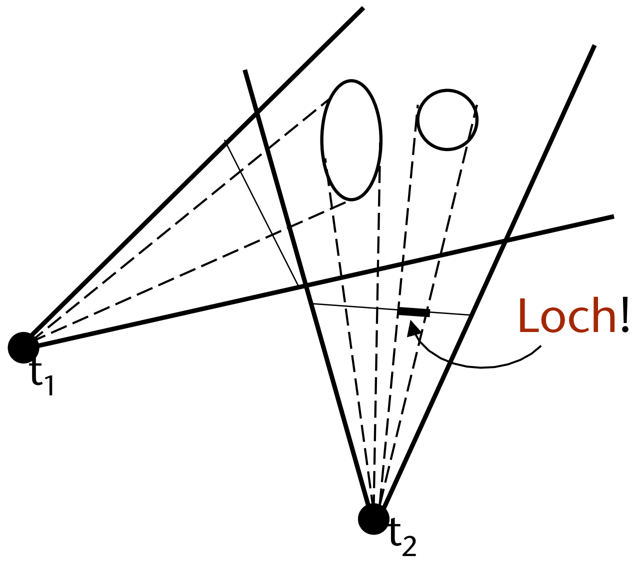



Appl Frame
(t1)
P_A
T_{t1,cam←img}
y
Camera (t1)
-z
x
T_{t1,wld←cam}
y
World (t1)
x
z
T^i_{t1,obj←wld}
y
Object
P
x
z
T^i_{t2,wld←obj}
y
World (t2)
P
x
z
T_{t2,cam←wld}
y
Camera (t2)
-z
x
T_{t2,img←cam}
Warped
Server Frame
(t2)
P_A
P_S

Remarks

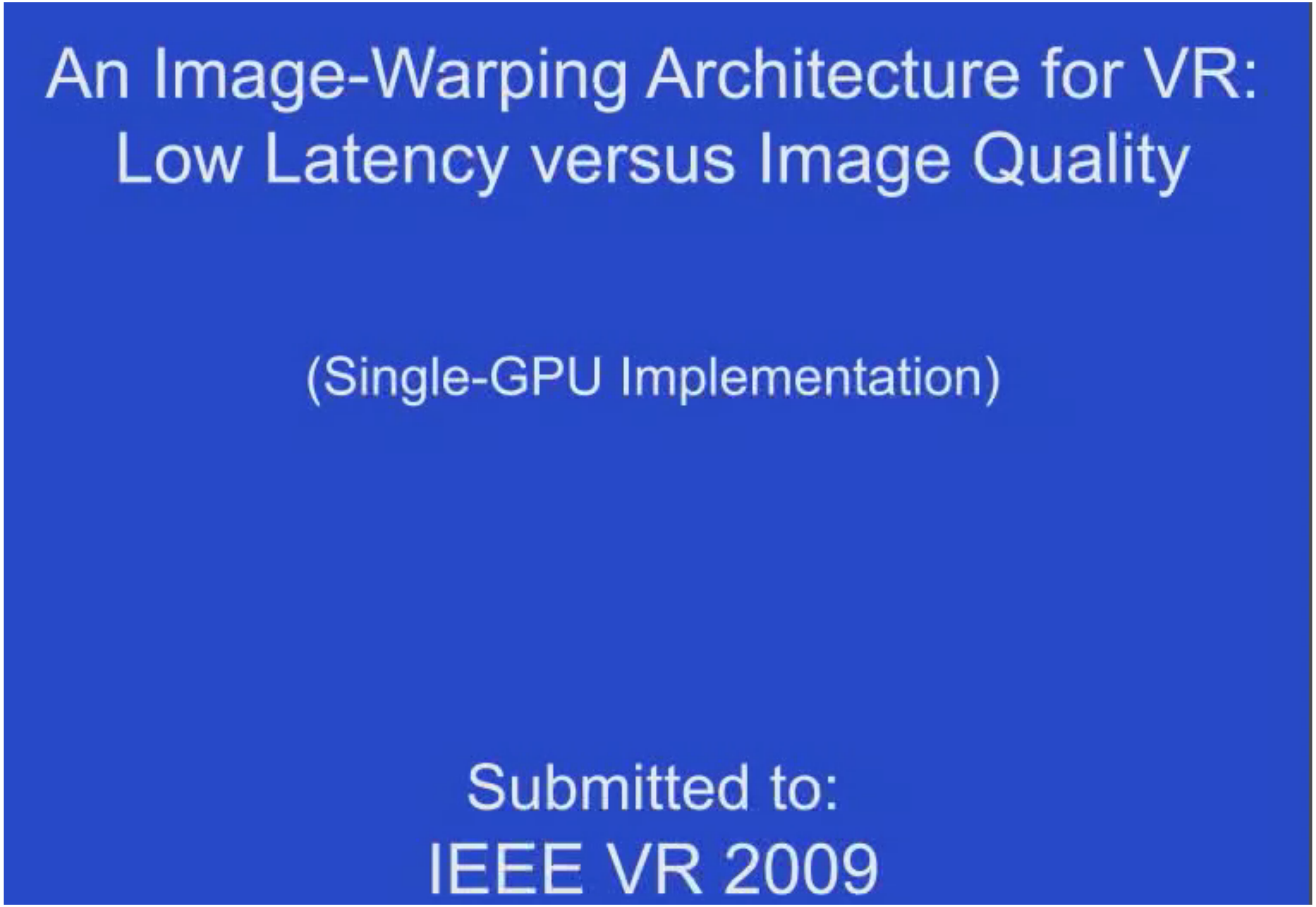
- Implementation of the warping:
  - In the vertex shader
    - Doesn't work in the fragment shader, because the output (= pixel) position is fixed in fragment shaders!
  - Warping renderer treats the image in the FBO containing the app frame as a texture , and it loads all the T_i's
  - Render 1024x1024 many GL_POINTs (called point splats)
- Advantages:
  - The frames (visible to the user) are now "more current", because of more current camera and object positions
  - Server framerate is independent of number of polygons

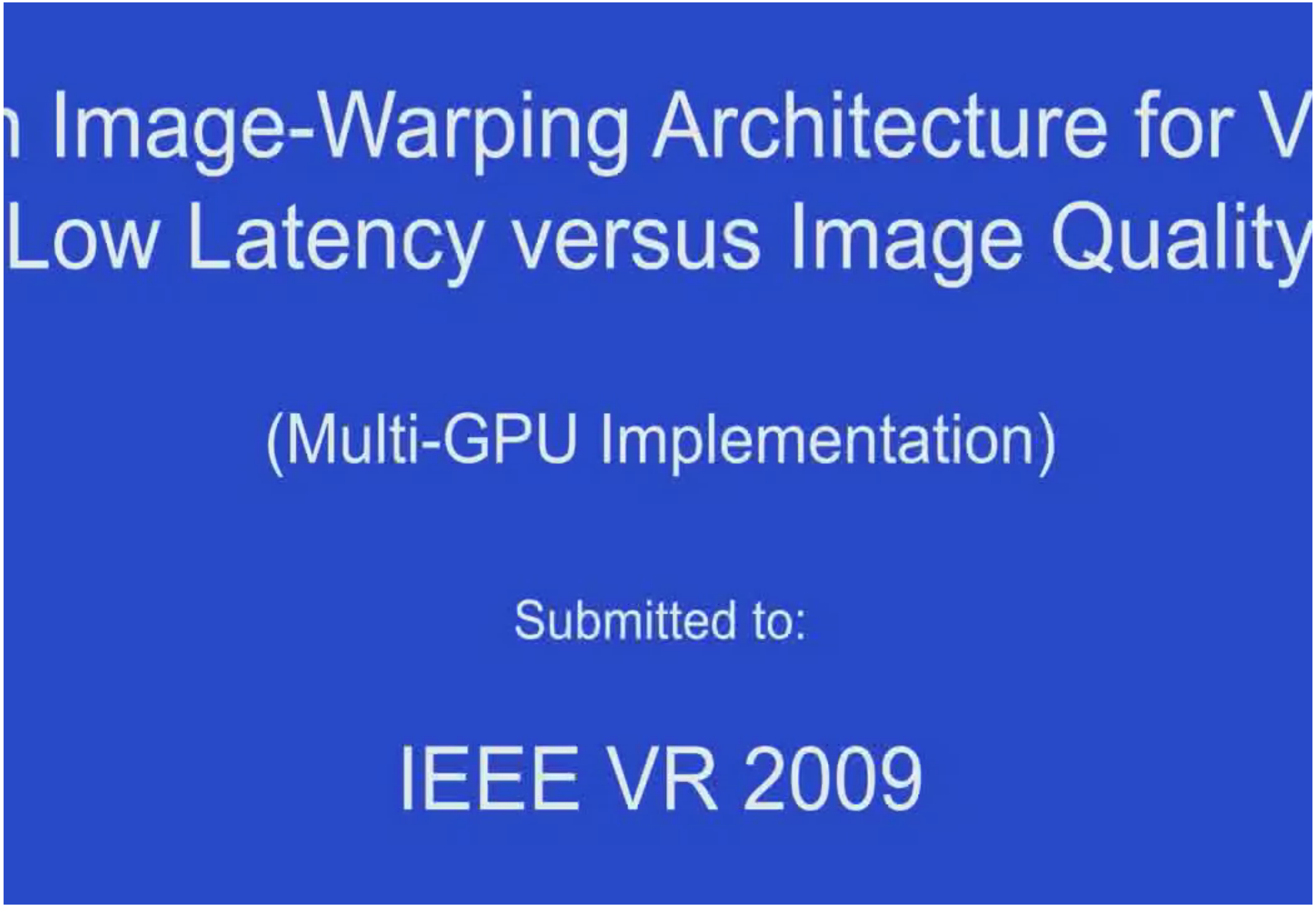
- Problems:
  - Holes in server frame
    - Need to fill them, e.g., by ray casting
  - Server frames are fuzzy (unscharf) (because of point splats)
  - How much should the point splats be?
  - The application renderer (full image renderer) can be only so slow (if it's too slow, then server frames become too bad)
  - Unfilled parts along the border of the server frames
    - Could make the viewing frustum for the app frames larger ...



- Performance gain:
  - 12m polygons, 800 x 600
  - Factor ~20 faster

Videos


An Image-Warping Architecture for VR:
Low Latency versus Image Quality
(Single-GPU Implementation)
Submitted to:
IEEE VR 2009


n Image-Warping Architecture for V
Low Latency versus Image Quality
(Multi-GPU Implementation)
Submitted to:
IEEE VR 2009