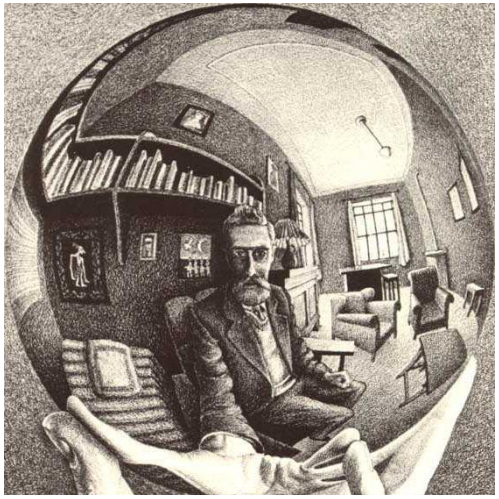
Virtual Reality & Physically-Based Simulation Techniques for Real-time Rendering



G. Zachmann

University of Bremen, Germany

cgvr.cs.uni-bremen.de

Simulator Sickness

- Simulator sickness = more or less of the following symptoms (can sometimes occur with prolonged stay in flight simulators / virtual environments):
  - Nausea (including vomiting), eye strain, dizziness, drowsiness, blurred vision, headache, fatigue
- Cause is not entirely clear
- Common hypothesis: inconsistent sensory input to brain, e.g., mismatch between vision and vestibular organ (organ for equilibrium)
  - E.g., when staying below deck for a prolonged time
  - In flight simulators with latency between motion of platform and rendering
- Frequency: occurs with 20-40% of jet pilots
  - Occurs more frequently with experienced pilots than novices [sic]
- Other observations with mismatching sensory inputs:
  - In a rotating field when walking forward, people tilt their heads and feel like they are rotating in the opposite direction
  - If a person is walking on a treadmill holding onto a stationary bar and you change the rate the visuals are passing by, it will feel to the person like the bar is pushing or pulling on their hands

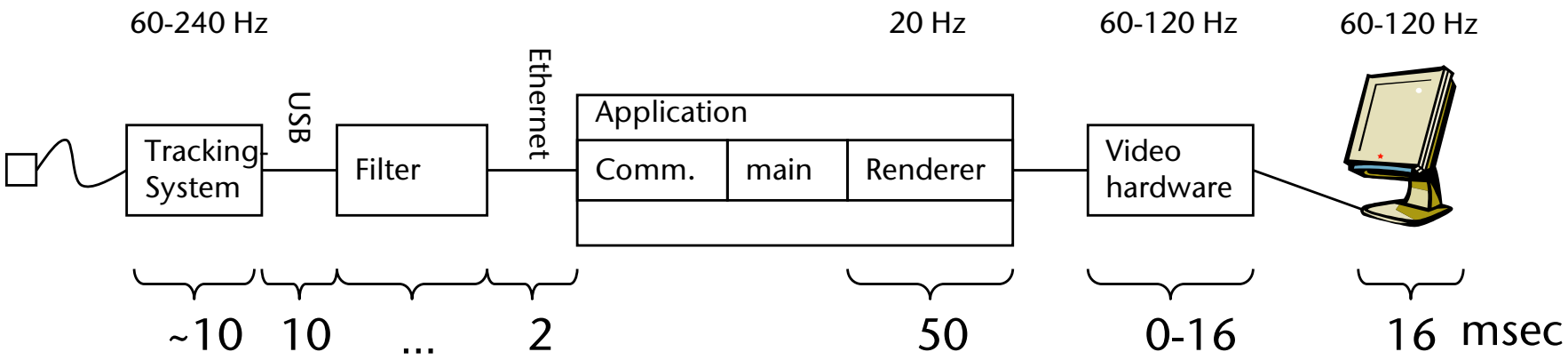
Latency (Lag, Delay)

- Definition: Latency = duration from a user's action (e.g., head motion) until display shows a change caused by the user's action ("from motion to photons")
- Some *human factors* (here for visual displays):

Latency (msec)	Effect on the user
> 5	Noticeable
> 30	*User performance* decreases (and possibly *simulator sickness*)
> 500	Immersion vanishes (and probably presence)

Note: a user's head can rotate by as much as 1000 degrees/sec !

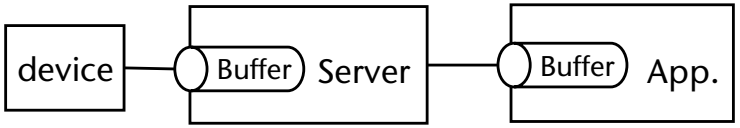
- The latency pipeline:

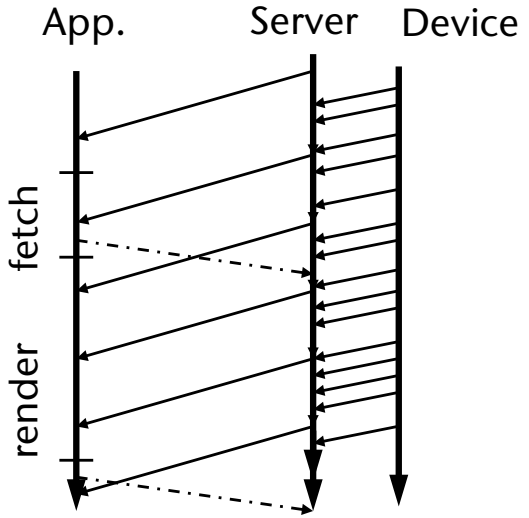


- Types/causes of lag:
  - Internal to devices
  - Transportation of data over communication channel (e.g., Ethernet)
  - Software (time for processing the data)
  - Synchronization delay

General Strategies for Solution

- Gerät-Server-App-Kommunikation:
  - Put device and server into continuous mode
  - Send "keep alive" messages from client to server

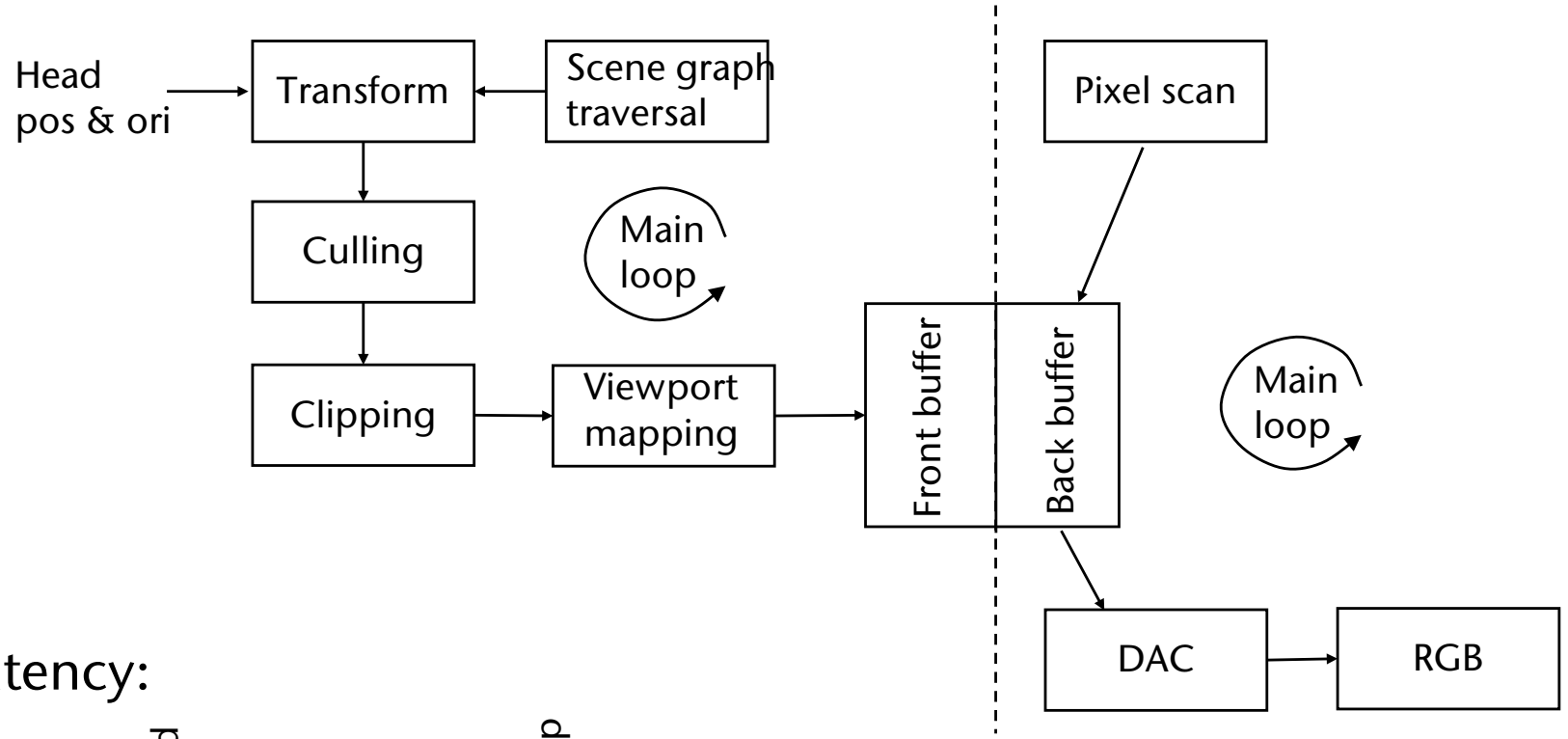




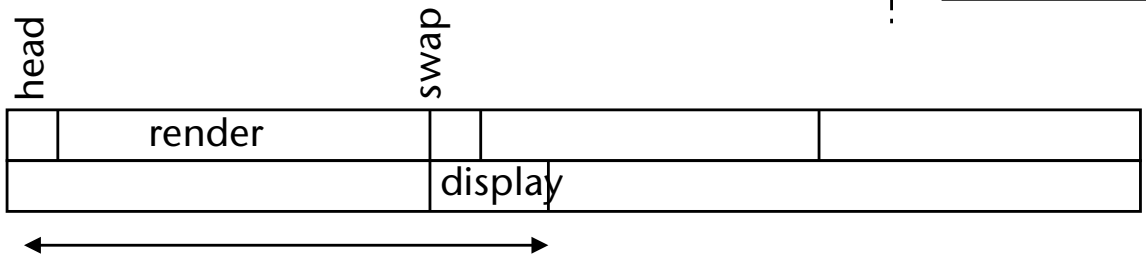
- Do time-critical computing:
  - Each and every module of the app receives a specific time budget
  - Module tries to compute a usable(!) partial solution as good as possible within the time budget
  - Stop when time is up
- Try to predict user/tracker position in *x* msec's time

Sources of Latency During Rendering

- Classical graphics pipeline (parts of it):



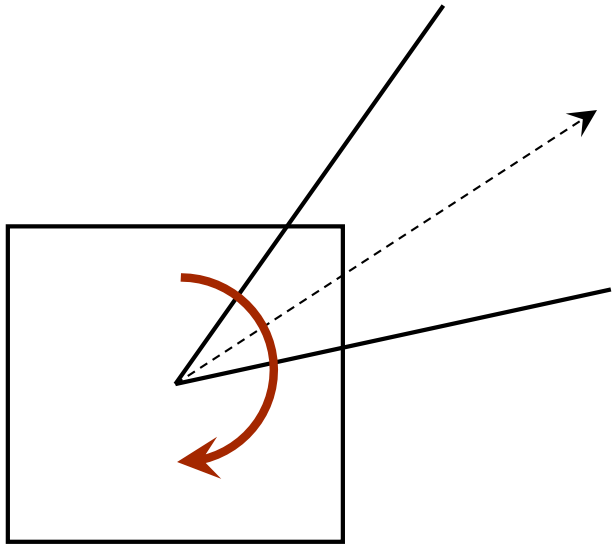
- Latency:



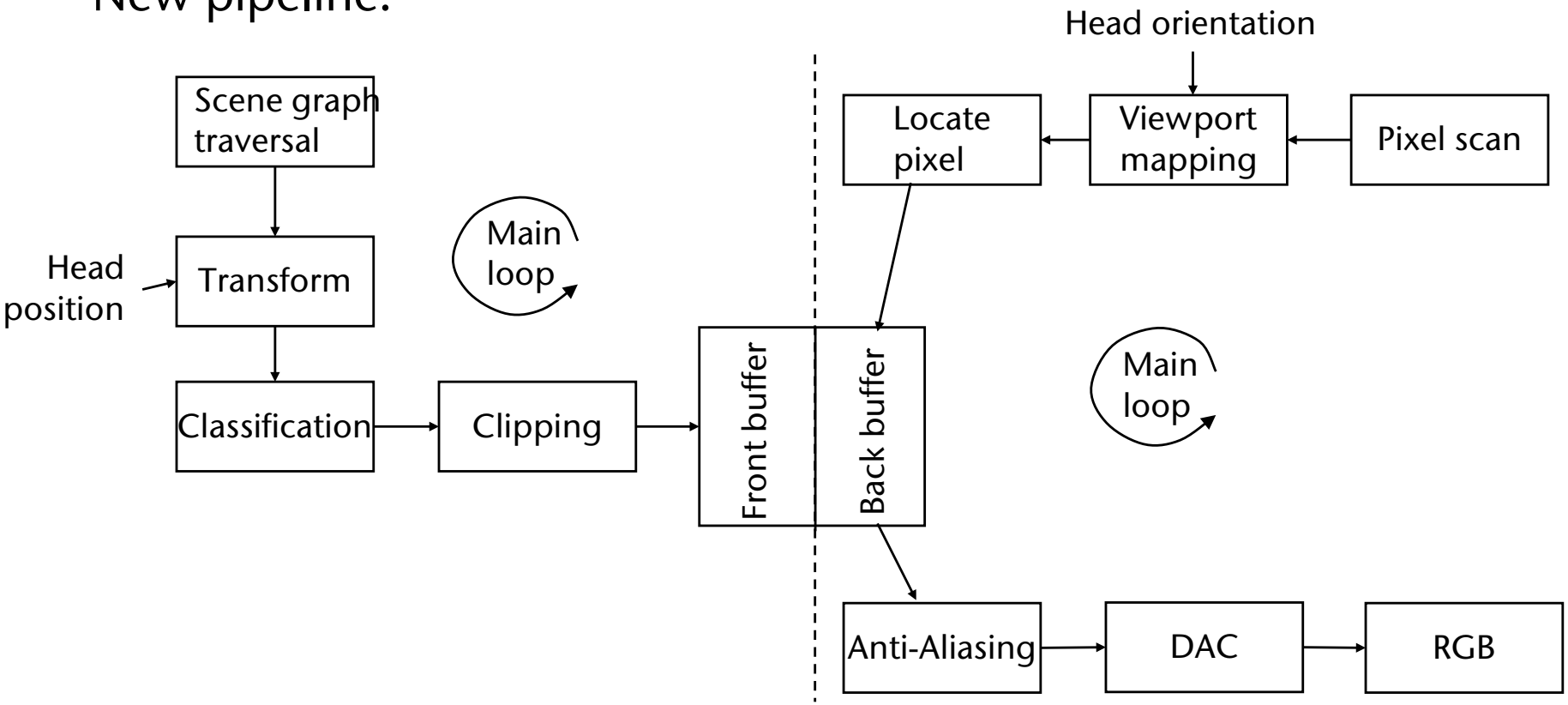
- Idea for improvement: render more than just the viewport

Viewport-Independent Rendering

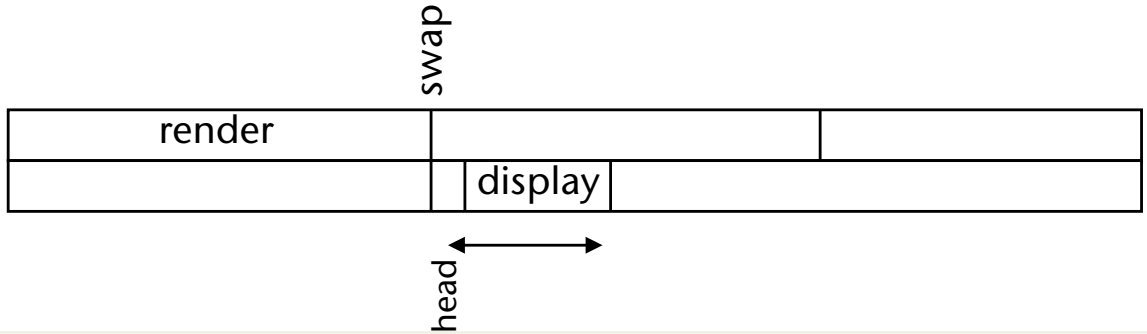
- Conceptual idea:
  - Render the scene onto a *sphere* around the viewer ⟶ spherical viewport
  - If viewpoint rotates: just determine new cutout of the spherical viewport
- Practical implementation:
  - Use cube as a viewport around user, instead of sphere
  - This was also one of the motivations to build Cave's



- New pipeline:

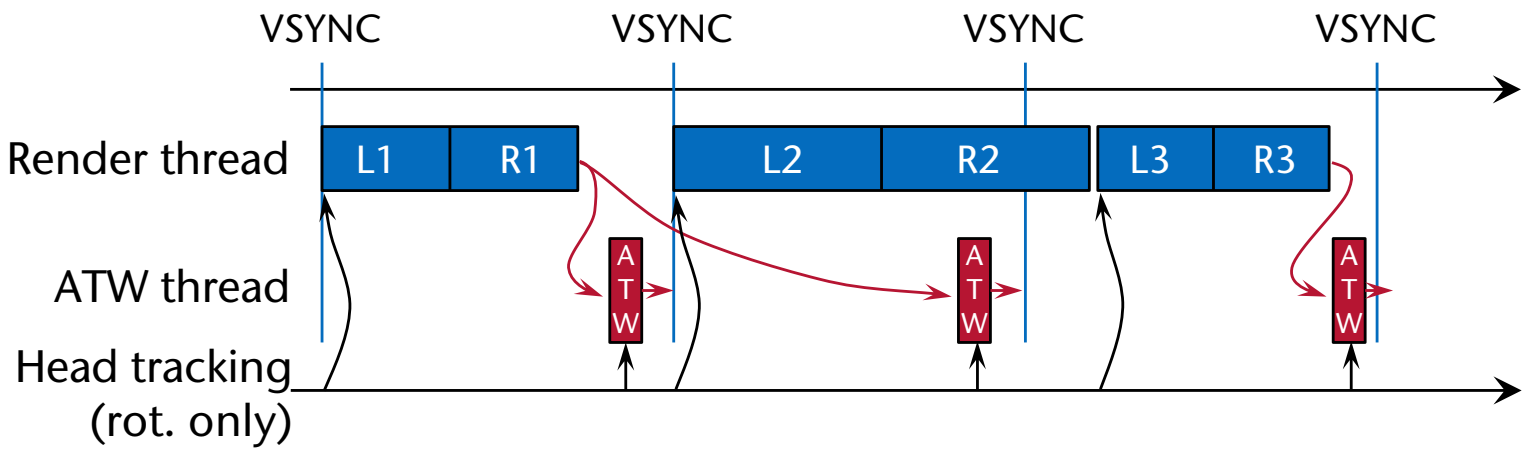


- Latency:



"Asynchronous Timewarp" (Oculus)

- Shift image using current orientation of head
- Do this only in case the renderer is not finished in time:

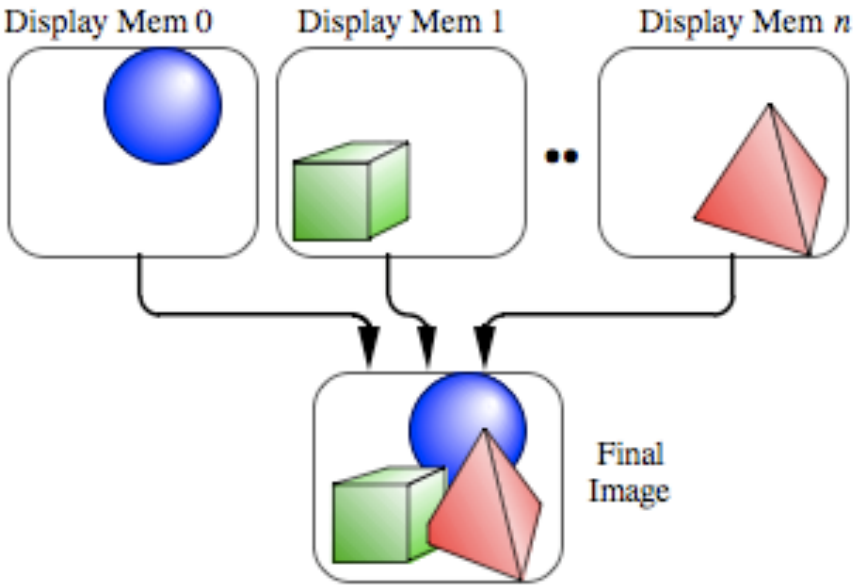


- Requires GPU preemption (i.e., stop GPU's pipeline, including shaders, immediately)

- Limitations:
  - Judder of animated objects
  - Incorrect positions of highlights and specular lighting
  - Head rotation also changes position of the viewpoint, but the image is shifted only according to rotation of viewing direction ⟶ judder for near objects (even static objects)

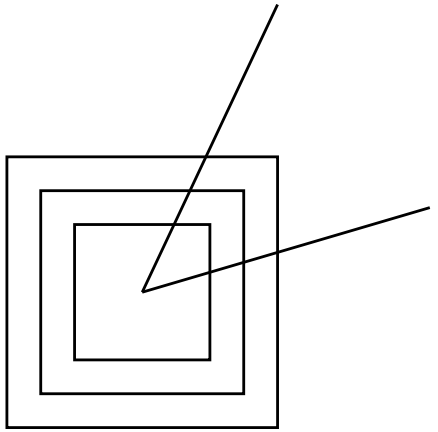
Multi-Threaded Rendering and Image Composition

- Conceptual 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 the Z values of corresponding pixels
- In practice:
  - 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



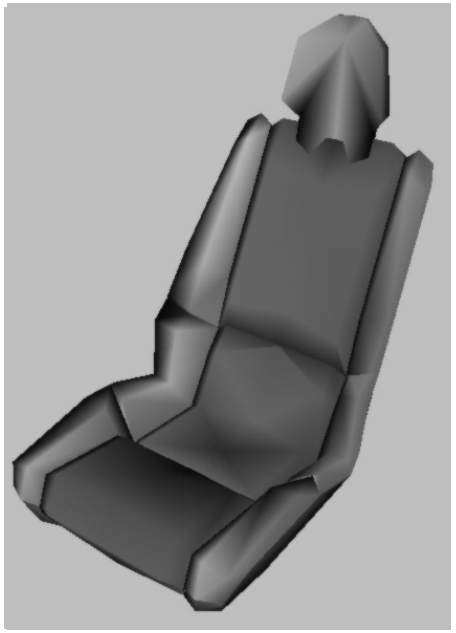
Constant Framerate by "Omitting"

- 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 (called stutter/judder)
- 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

- Example: do you see a difference?





- Idea: render a reduced version of the object, where the amount of reduction is chosen such that users can't see the difference from the full-resolution version

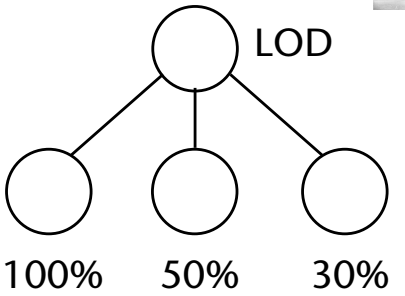
- Definition:

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

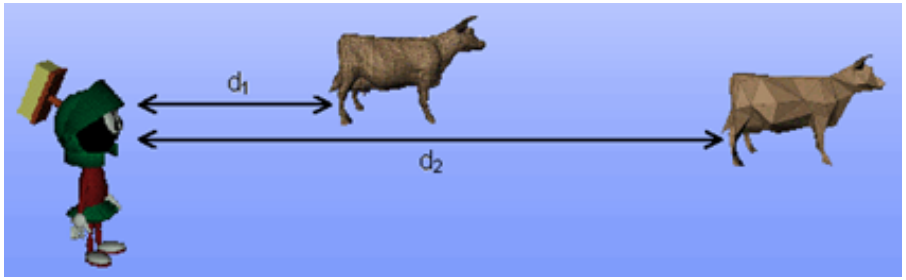
- The technique consists of two tasks:

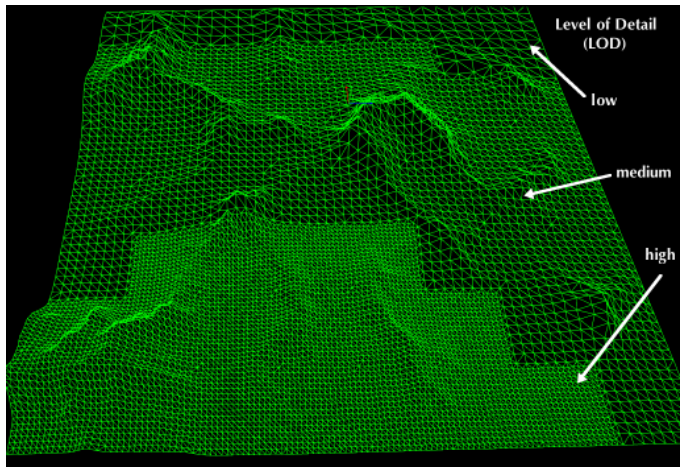
  1. Preprocessing: for each object in the scene, generate *k* LODs
     - For instance, we generate LODs at 100%, 80%, 60%, ..., of number of polygons of original model
  2. Runtime: select "right" LOD, make switches between LODs unnoticeable

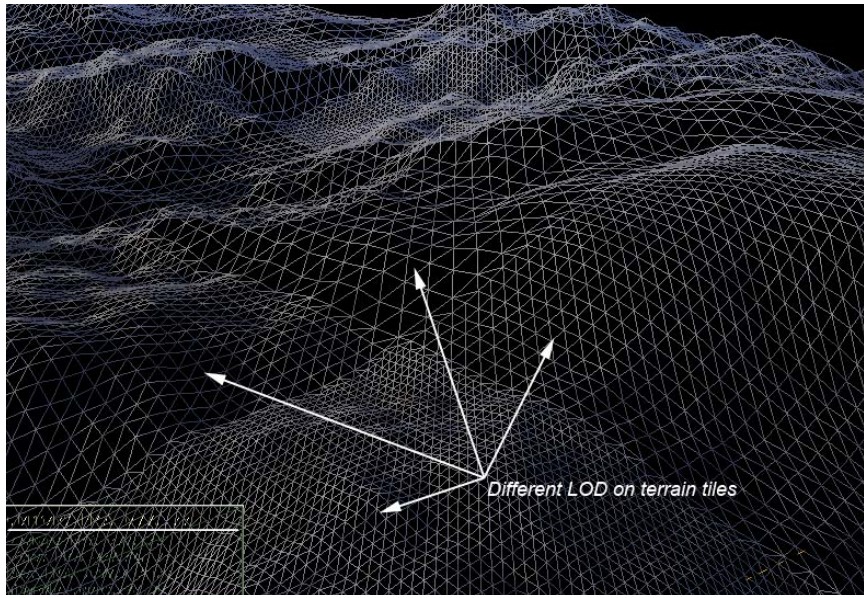
Selection of the LOD



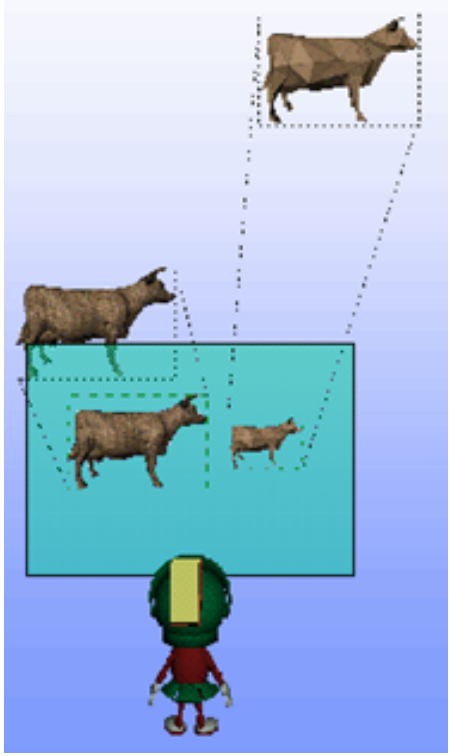
- Balance visual quality against "temporal quality"
- Static selection algorithm:
  - Level i for a distance range (d_i, d_{i+1})
  - Depends on FoV
  - Problem: size of objects is not considered
- For some desktop applications, e.g. terrain rendering, this can be sufficient:







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

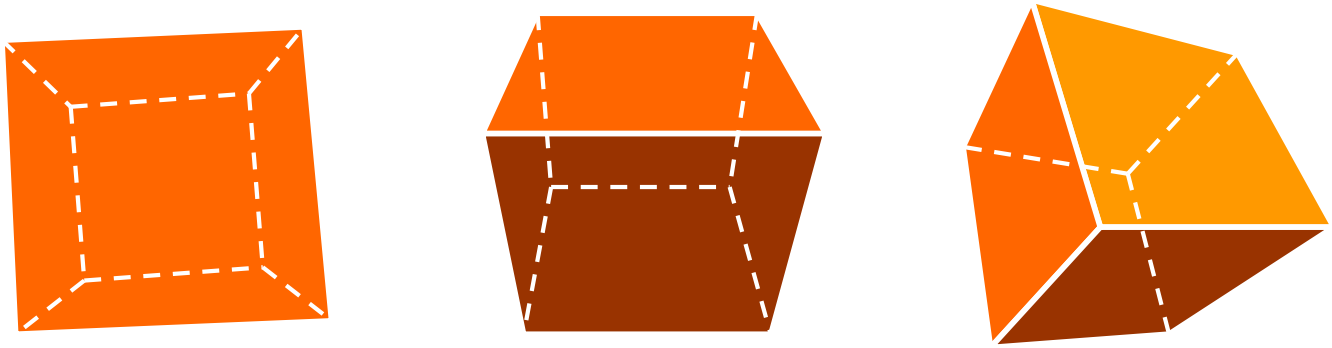


Estimation of Size of Object on the Screen

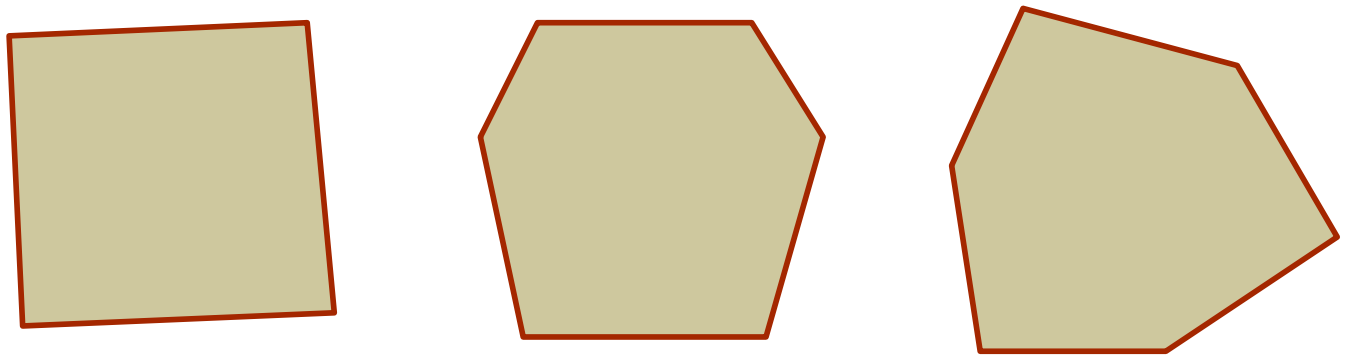
- Naïve method:
  - Compute bounding box (bbox) of object in 3D (probably already known by scenegraph for occlusion culling)
  - Project bbox onto 2D → 8x 2D points
  - Compute 2D bbox (axis aligned) around 8 points
- Better method:
  - Compute true area of projected 3D bbox on screen

Idea of the Algorithm

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



- Project only points on the silhouette (4 or 6) onto 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 case in LUT), accumulate areas

Psychophysiological LOD Selection

- Idea: exploit human factors with respect to visual acuity:
  - Central / peripheral vision:

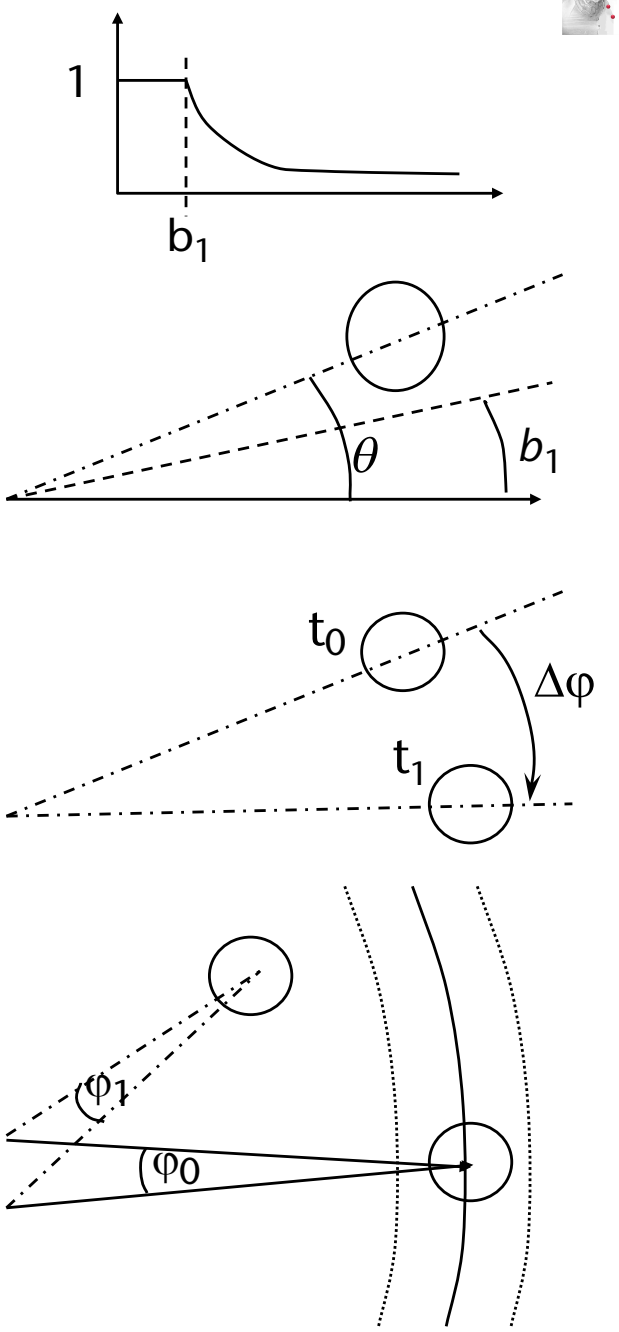
$$k_1 = \begin{cases} e^{-(\theta - b_1)/c_1} & , \theta > b_1 \\ 1 & , \text{sonst} \end{cases}$$

  - 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. $r_{\min} = 1/k$
  3. Select level l such that

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

where P_l is the set of polygons of level l of an object, and $r(p)$ = radius of polygon p

- 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, based on history
  - Let T_b = time budget that can be spent for next frame
    - Usually constant, e.g., 16 msec for 60 Hz framerate
  - 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

Predictive LOD Selection

[Funkhouser &Sequin]

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

  O = object, L = level,

  R = rendering quality (#textures, #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 $\max_{S' \subset S} \sum_{(O,L,R) \in S'} \text{benefit}(O, L, R)$

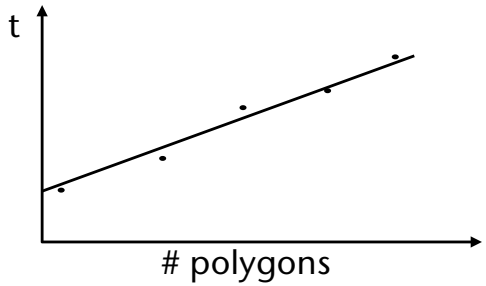
  under the condition $T_r = \sum_{(O,L,R) \in S'} \text{cost}(O, L, R) \leq T_b$

  where $S = \{$ all possible object tuples in the scene $\}$

- Cost function depends on:
  - Number of vertices (≈ *# coord. transforms + lighting calcs + clipping*)
  - Setup time per polygon
  - Number of pixels (*scanline conversions, alpha blending, texture fetching, anti-aliasing, Phong shading*)
  - Theoretical cost model:

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

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



- Benefit function: "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 (similar to psychophysiological LOD factors)
  - 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
  → NP-complete
- Idea: compute sub-optimal solution:
  - Reduce it to continuous knapsack problem (see algorithms class)
  - Solve this 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
  - Add'l constraint: no 2 object tuples must represent the same object

- Incremental solution:
  - Start with solution $(O_1, L_1, R_1), \ldots, (O_n, L_n, R_n)$ 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

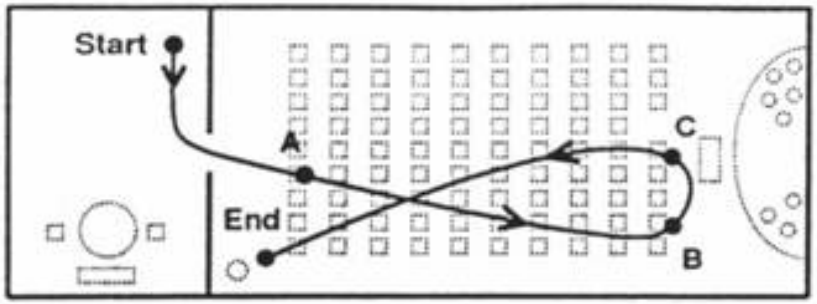
$$\text{value}(O_k, L_k + a, R_k + b) - \text{value}(O_k, L_k, R_k) = \text{max}$$

and

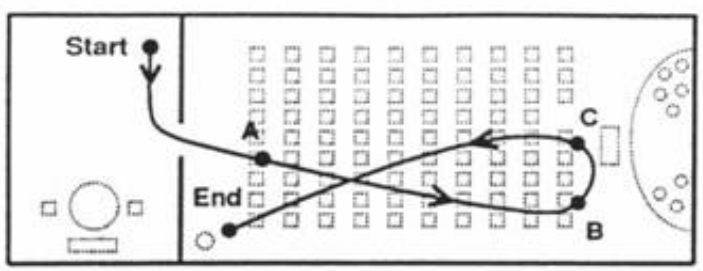
$$\sum_{i \neq k} \text{cost}(O_i, L_i, R_i) + \text{cost}(O_k, L_k + a, R_k + b) \leq \text{max. frame time}$$

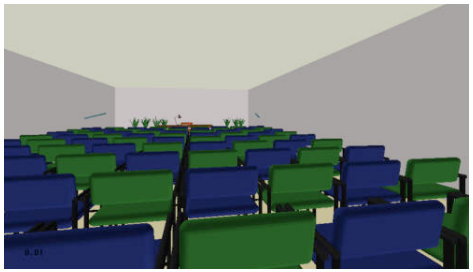
- Analog, if $\sum_i \text{cost}(O_i, L_i, R_i) > \text{max. frame time}$

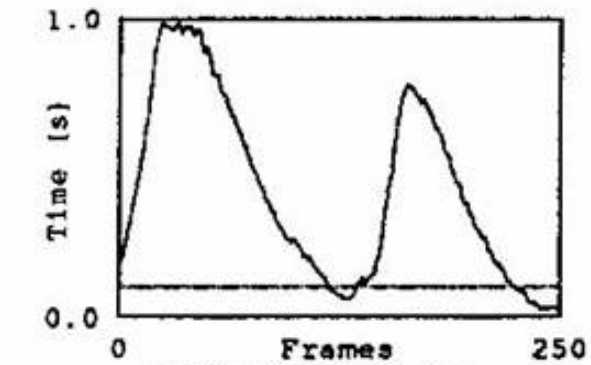
Example Scenario



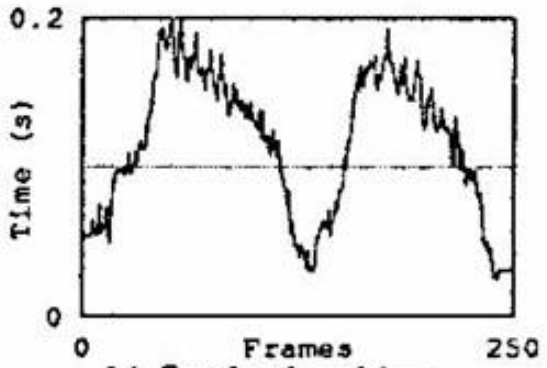
Performance in the example scenes



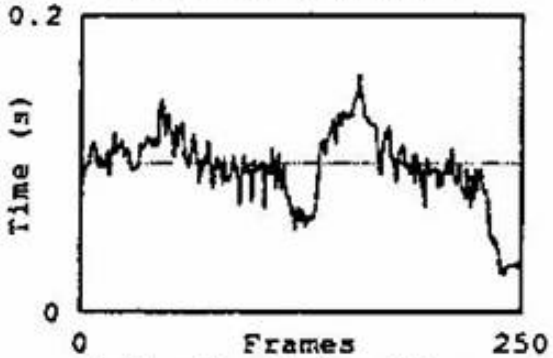




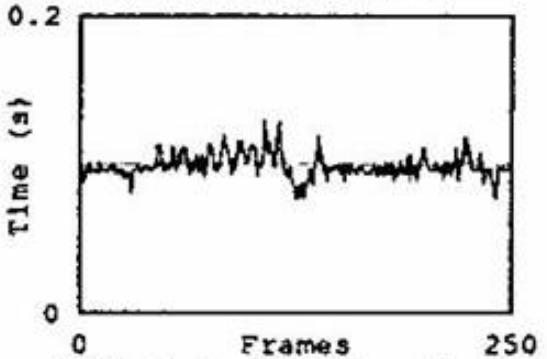
a) No detail elision.



b) *Static* algorithm.



c) *Feedback* algorithm.



d) *Optimization* algorithm.

Screenshots from Another Example Scene

No detail elision, 19,821 polygons

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

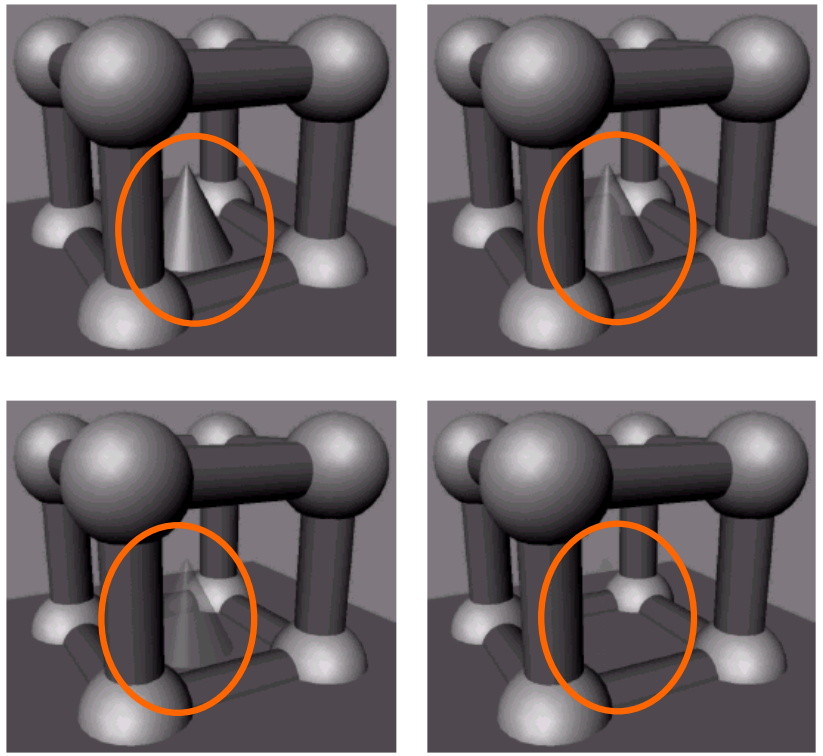
Level of detail: darker gray means more detail

Problem with Discrete LODs

- "Popping" when switching to next higher/lower level

1. Simplest solution: temporal hysteresis (reduces frequency of pops)

2. Alpha blending of the two adjacent LOD levels ("Alpha-LODs"):
   - Instead of switching from level *i* to *i*+1, fade out level *i* until gone, at the same time fade in level *i*+1
   - "Man kommt vom Regen in die Traufe"
   - Don't use them!



3. Continuous, view-dependent LODs using progressive meshes

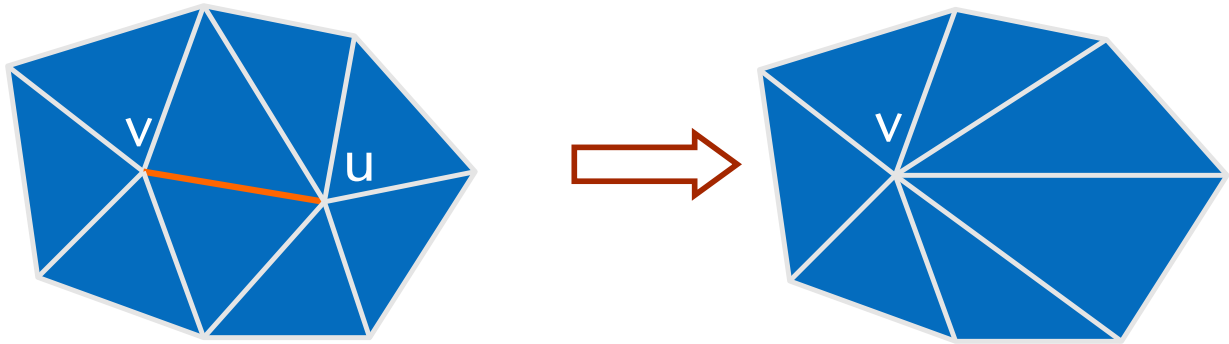
Progressive Meshes

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

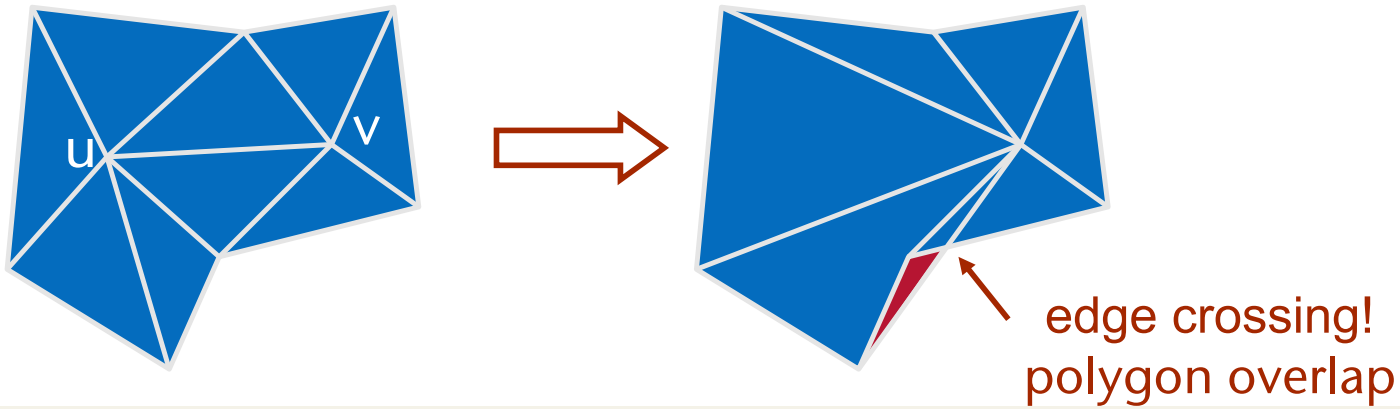
- Definition: progressive mesh = representation of an object, starting with a high-resolution mesh M_0, with which one can continuously (up to the vertex level) generate "in-between" meshes ranging from 1 polygon up to M_0 (and do that extremely fast).

Construction of Progressive Meshes

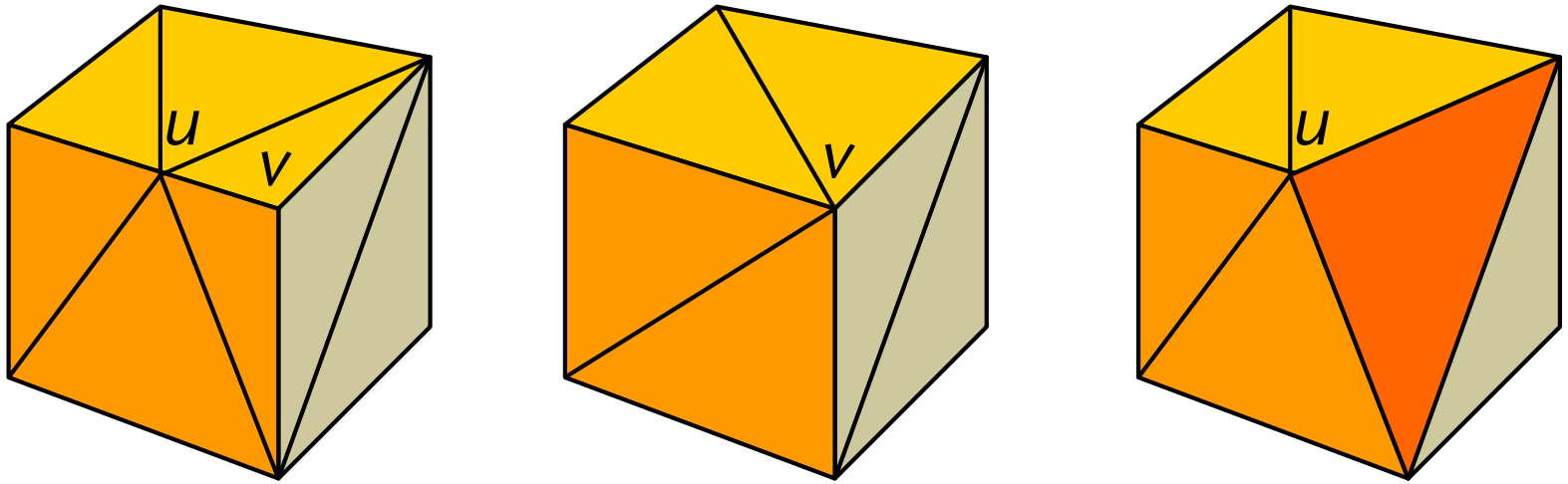
- Approach: successive *simplification*, until only 1 polygon left
- The fundamental operation: *edge collapse*



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



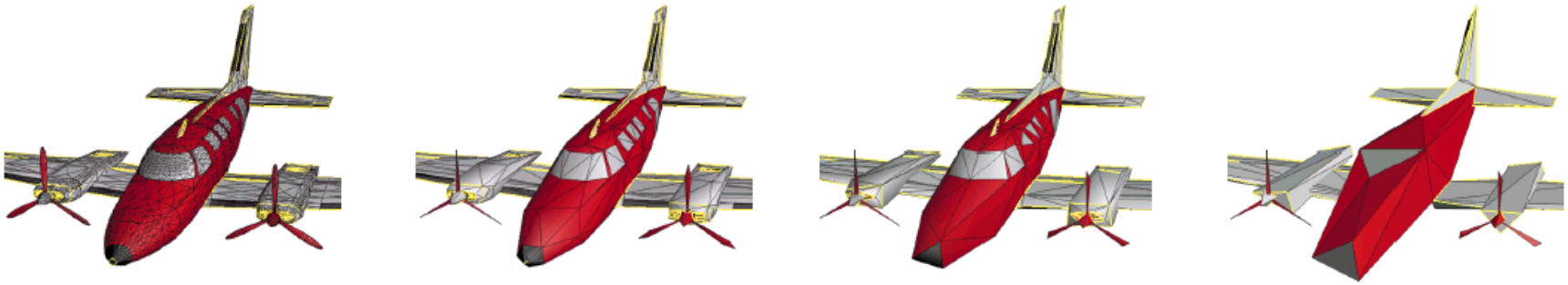
- The direction of edge collapses is important, too:



- Introduce measure of edge collapses that evaluates "visual effect"
- Goal: first perform 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)

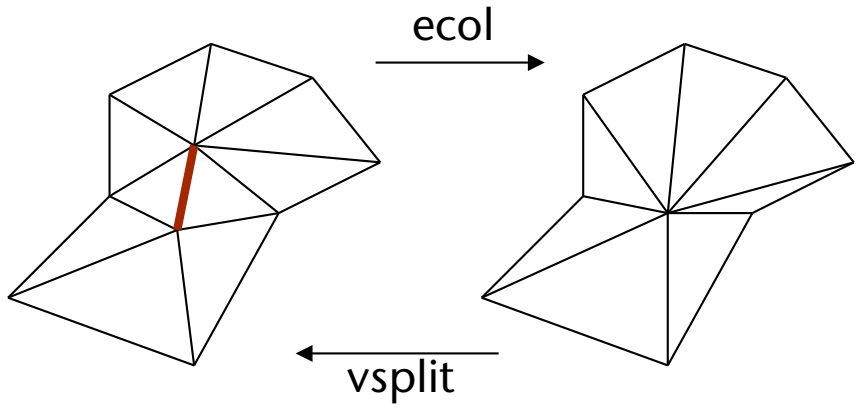
- Examples of a progressive mesh:



- Representation of a progressive mesh:

$$M = M^n \xleftrightarrow[\text{vsplit}_{n-1}]{\text{ecol}_{n-1}} \cdots \xleftrightarrow[\text{vsplit}_1]{\text{ecol}_1} M^1 \xleftrightarrow[\text{vsplit}_0]{\text{ecol}_0} M^0$$

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



- 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

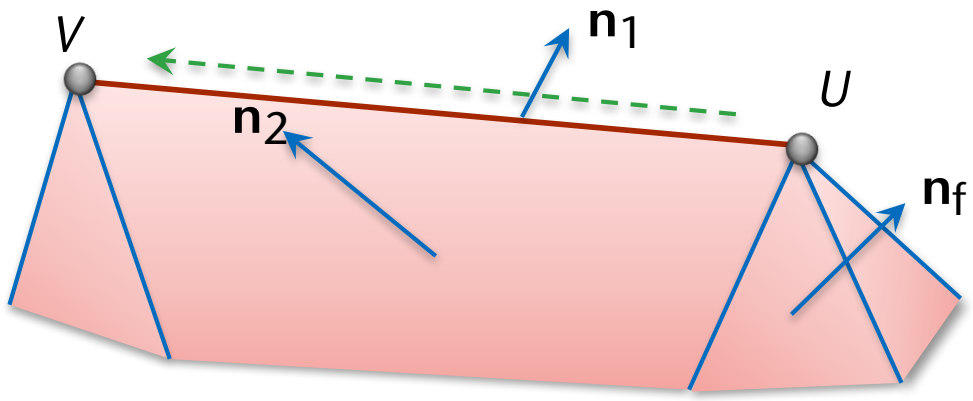
Example for a Simple Edge Evaluation Function

- 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*:

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

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

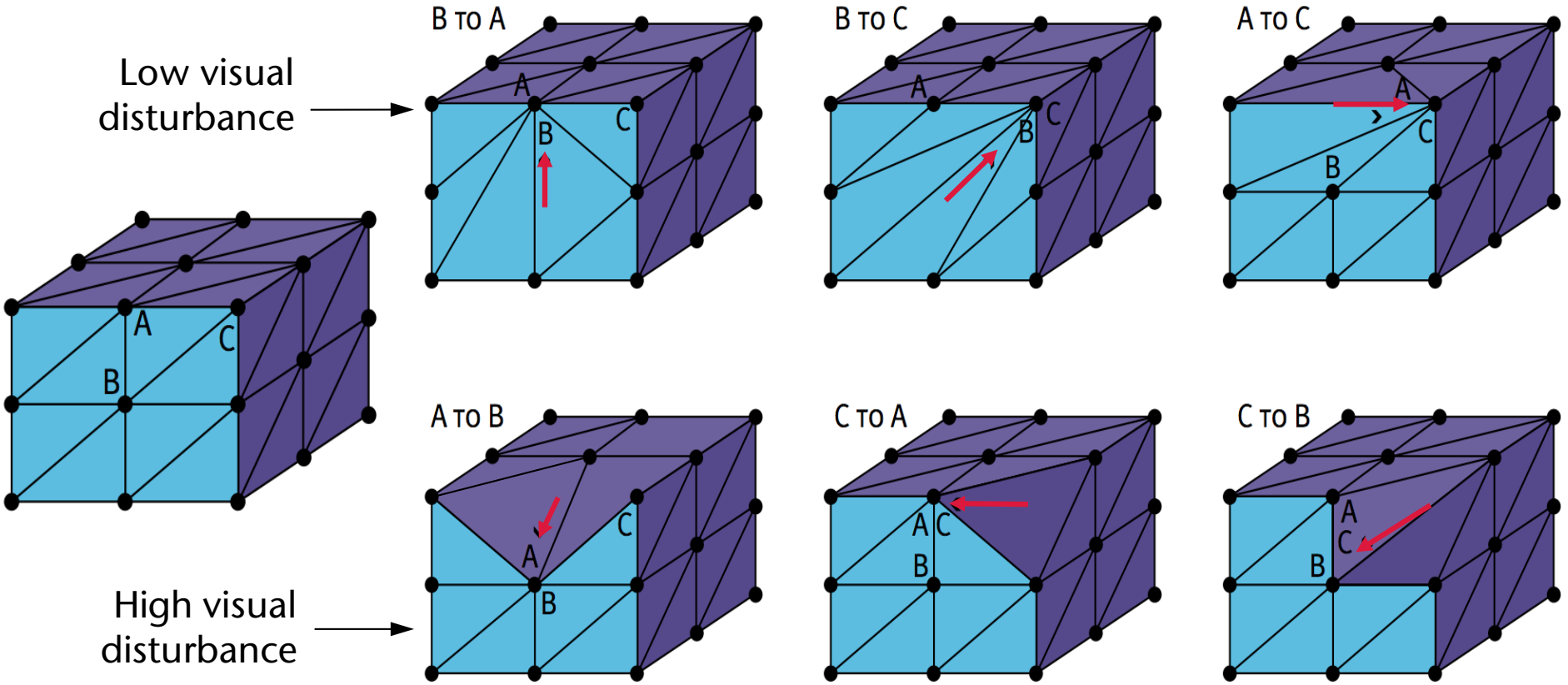
F(*U*) = set of all faces incident to U, but not to V



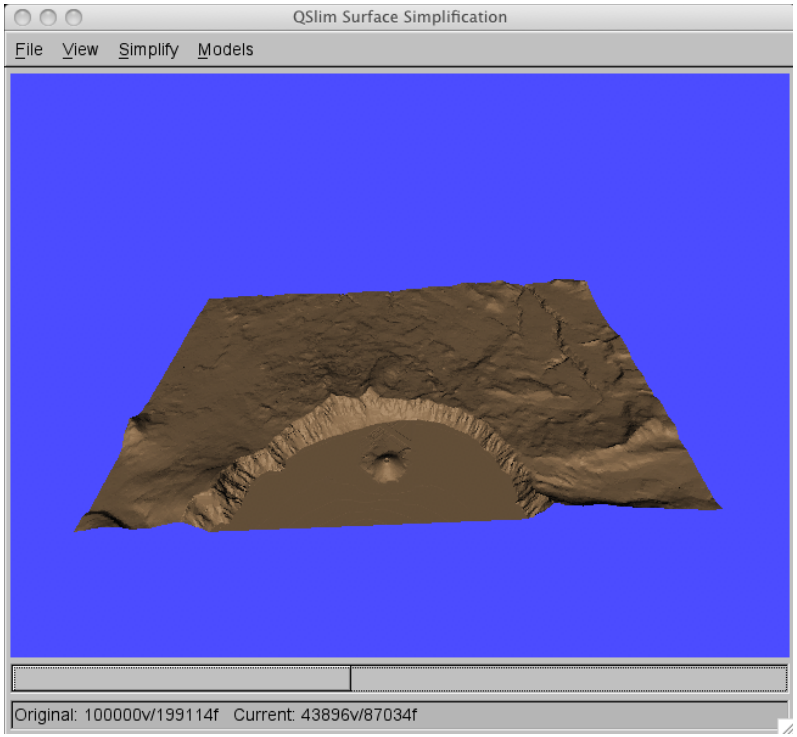
- Remark:

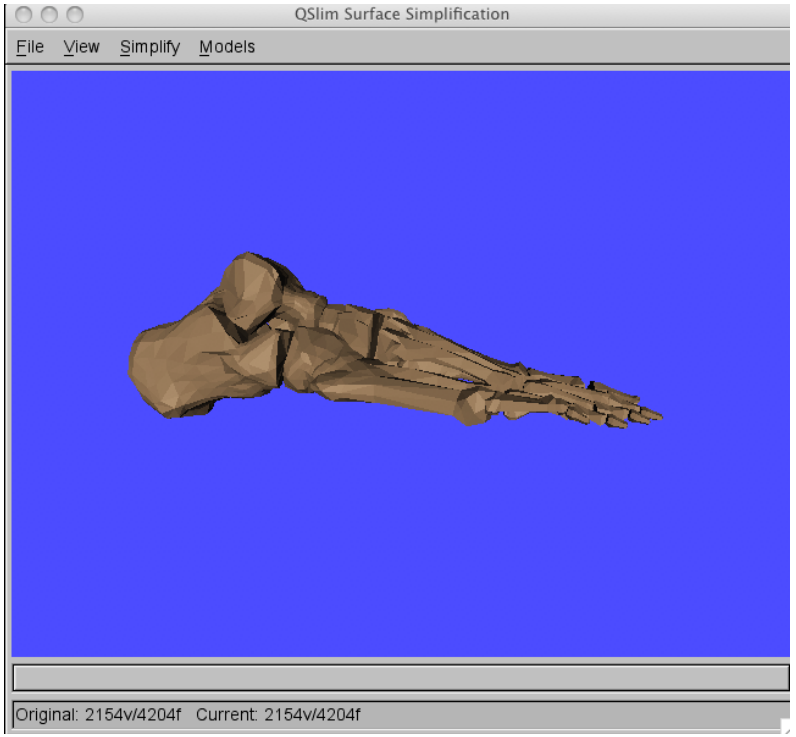
$$\text{cost}(U, V) \neq \text{cost}(V, U)$$

- Example:



Demo





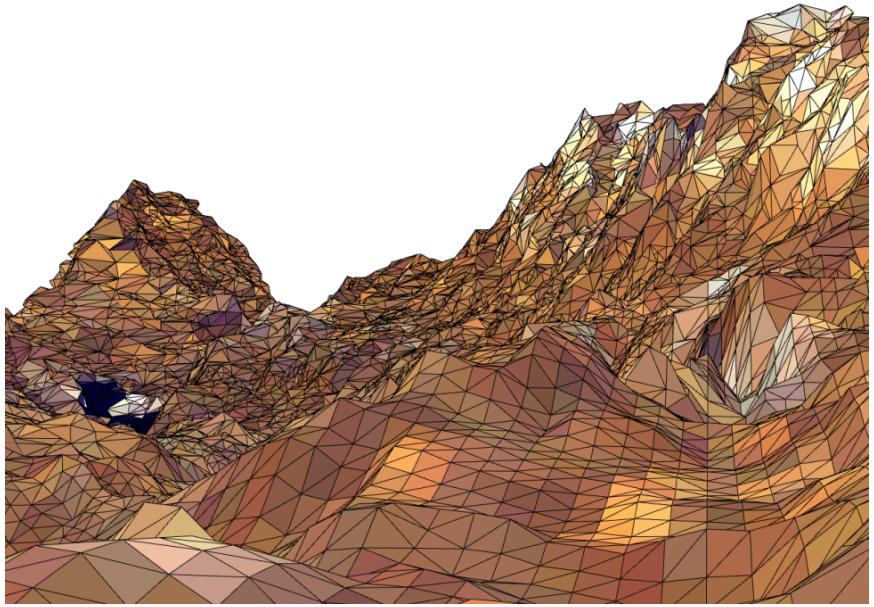
[Michael Garland: Qslim]

How can the Funkhouser-Sequin algorithms be combined with progressive meshes? And implemented on the GPU??

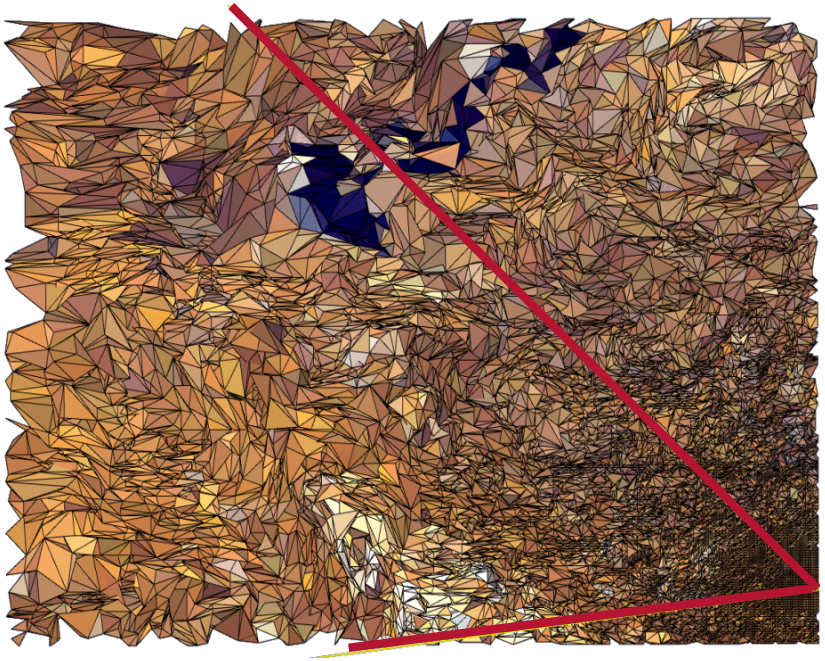
Master Thesis ...

View-Dependent LOD's

- Select *different* resolution within the *same* object, depending on the view point, i.e., different parts of one object are rendered at different resolutions
- Defining metric: screen space error (measured in pixels)
- Example: terrain – choose resolution according to projected area

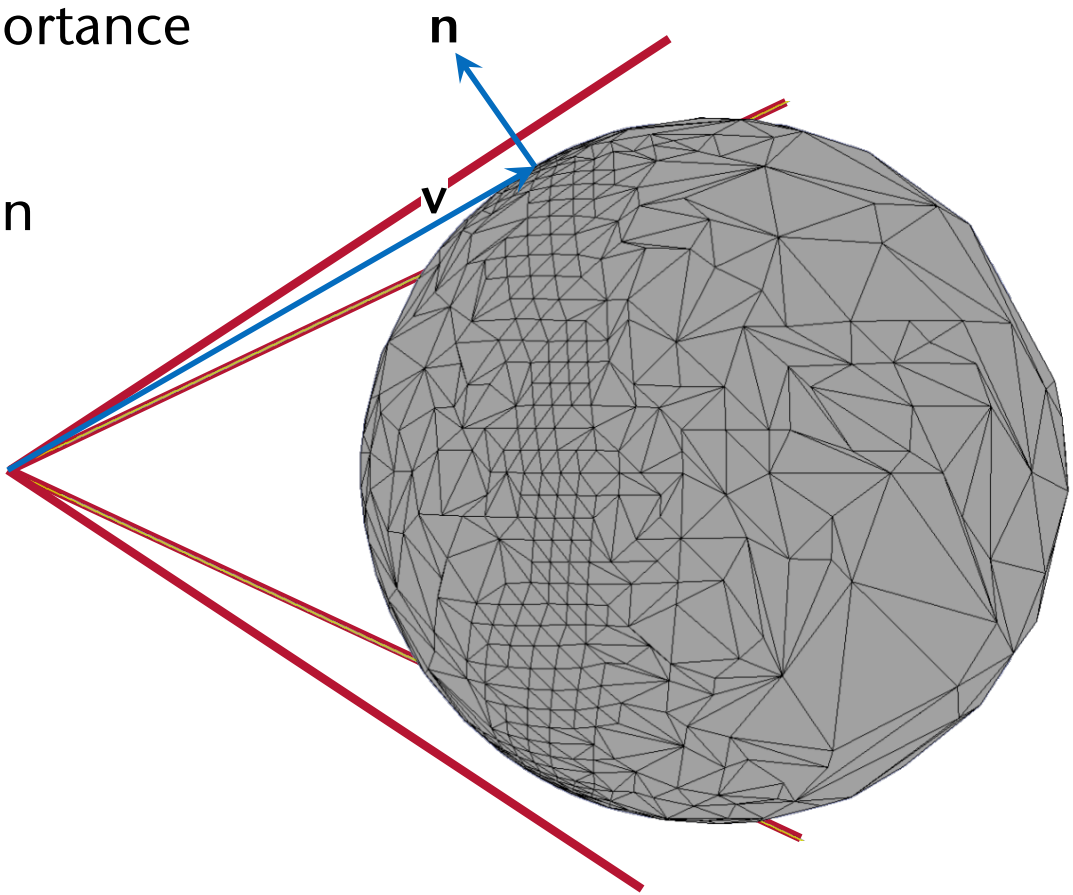


View from eye point



Birds-eye view

- Additional factor: visual importance
- Example: closed objects – render with higher resolution near silhouette border
  - Maximal screen space error is modulated by (**v**·**n**)



- Other possible criteria:
  - Specular highlights
  - Triangle budget
  - Time budget (time critical computing)

Pros and Cons

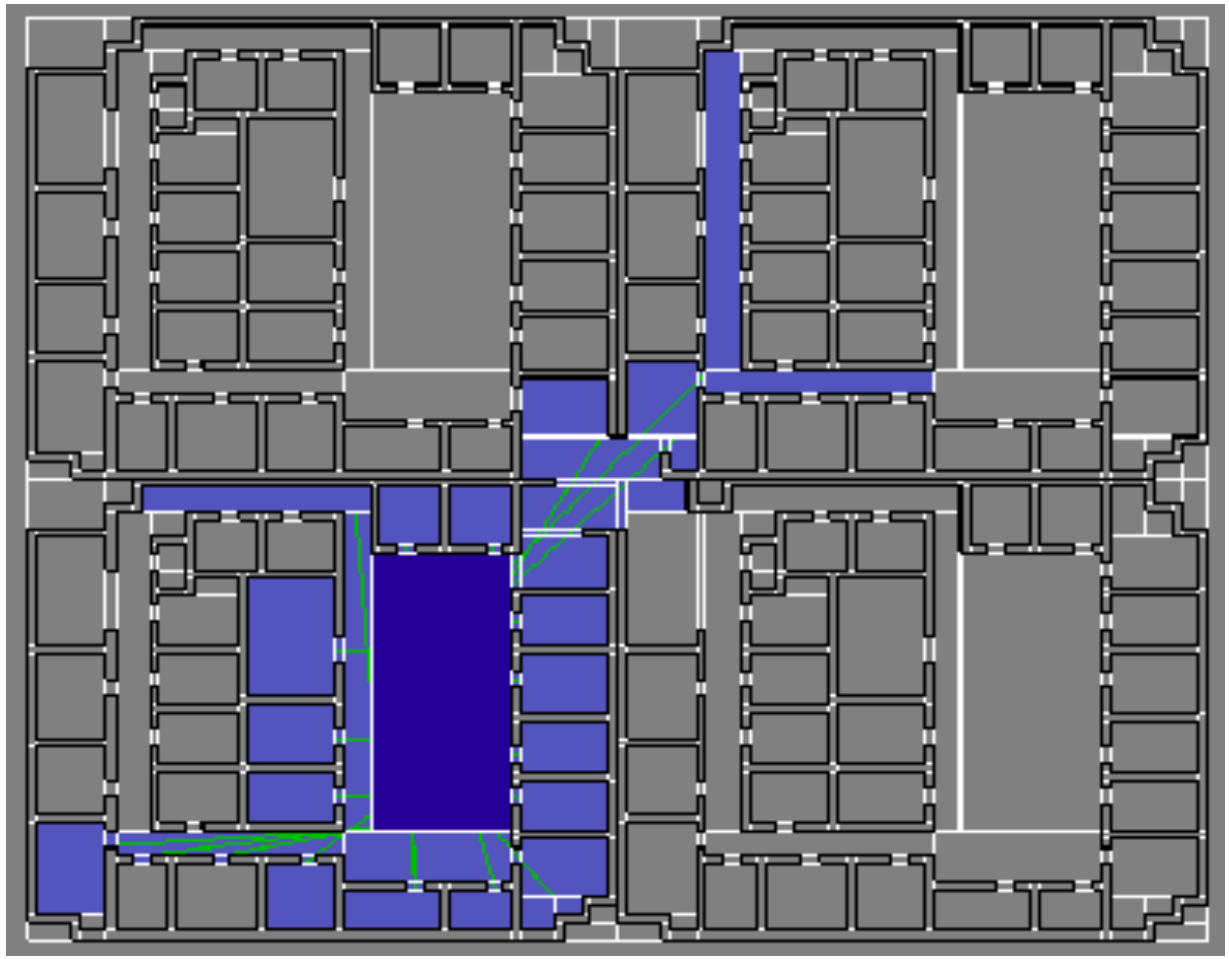
- Advantages of Dynamic LODs (e.g., progressive meshes):
  - No popping artefacts
  - Can be turned into view-dependent LOD
  - Better rendering fidelity for given polygon count
- Advantages of Static LODs:
  - Extremely simple for the renderer
    - Easy to implement in the renderer
    - No CPU overhead during rendering
  - Can upload LODs to GPU as vertex buffer objects (VBO)

Digression: Other Kinds of LODs

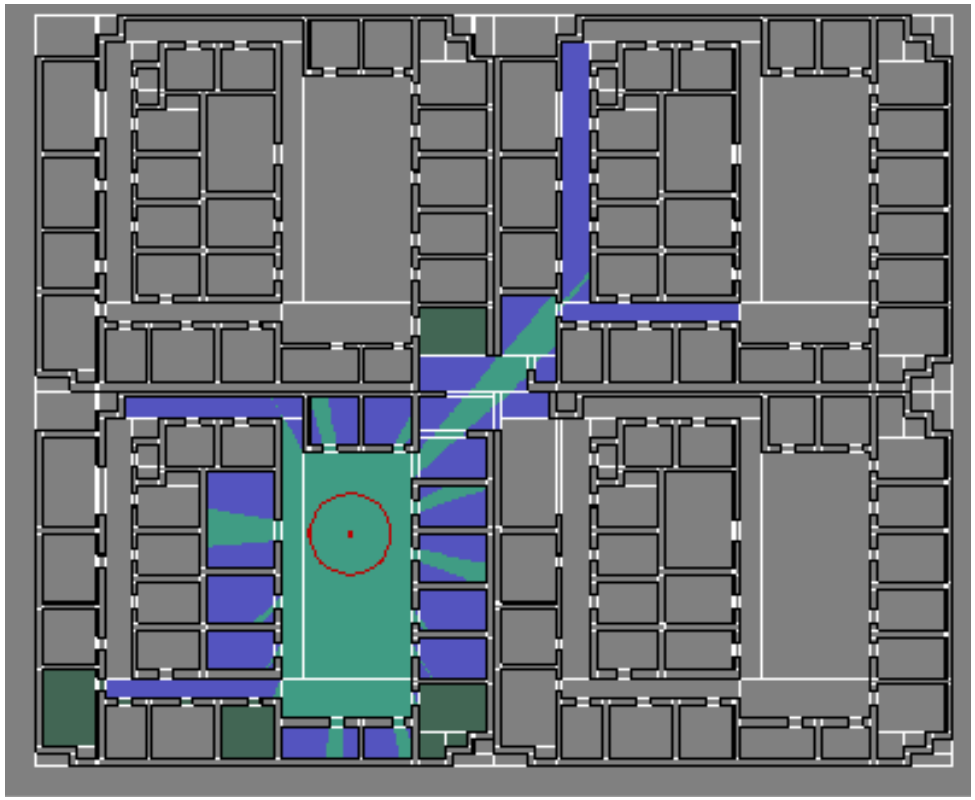
- Idea: apply LOD technique to other non-geometric content
- E.g. "*behavioral LOD*":
  - Simulate the behavior of an object exactly if in focus, otherwise simulate it only "approximately"

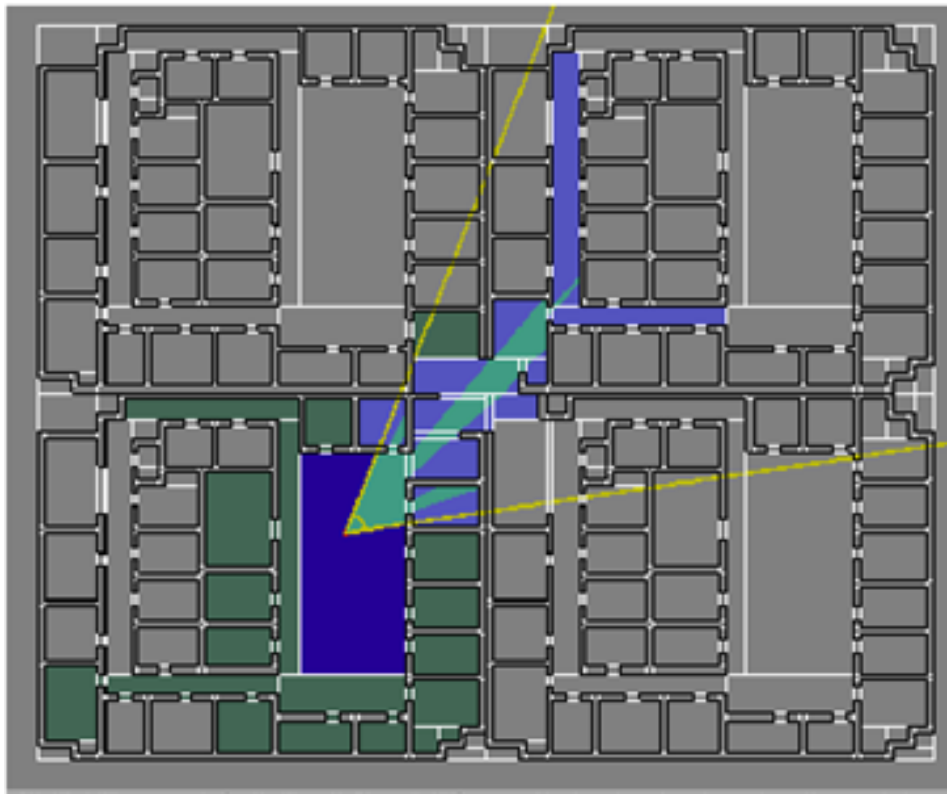
Culling in Buildings (Portal Culling)

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



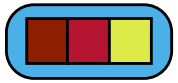
- During runtime, filter cells from visibility graph by viewpoint and viewing frustum:



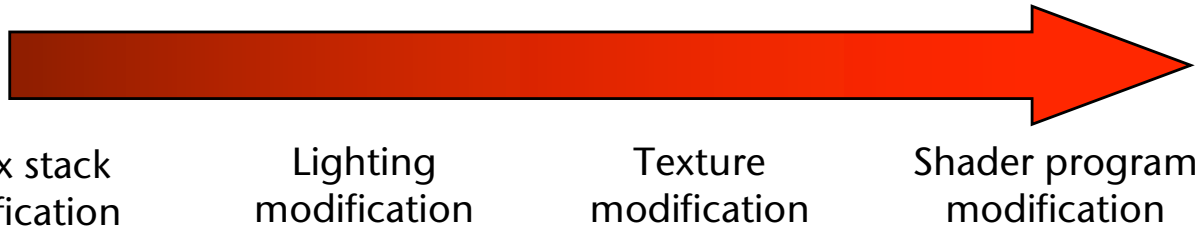


State Sorting

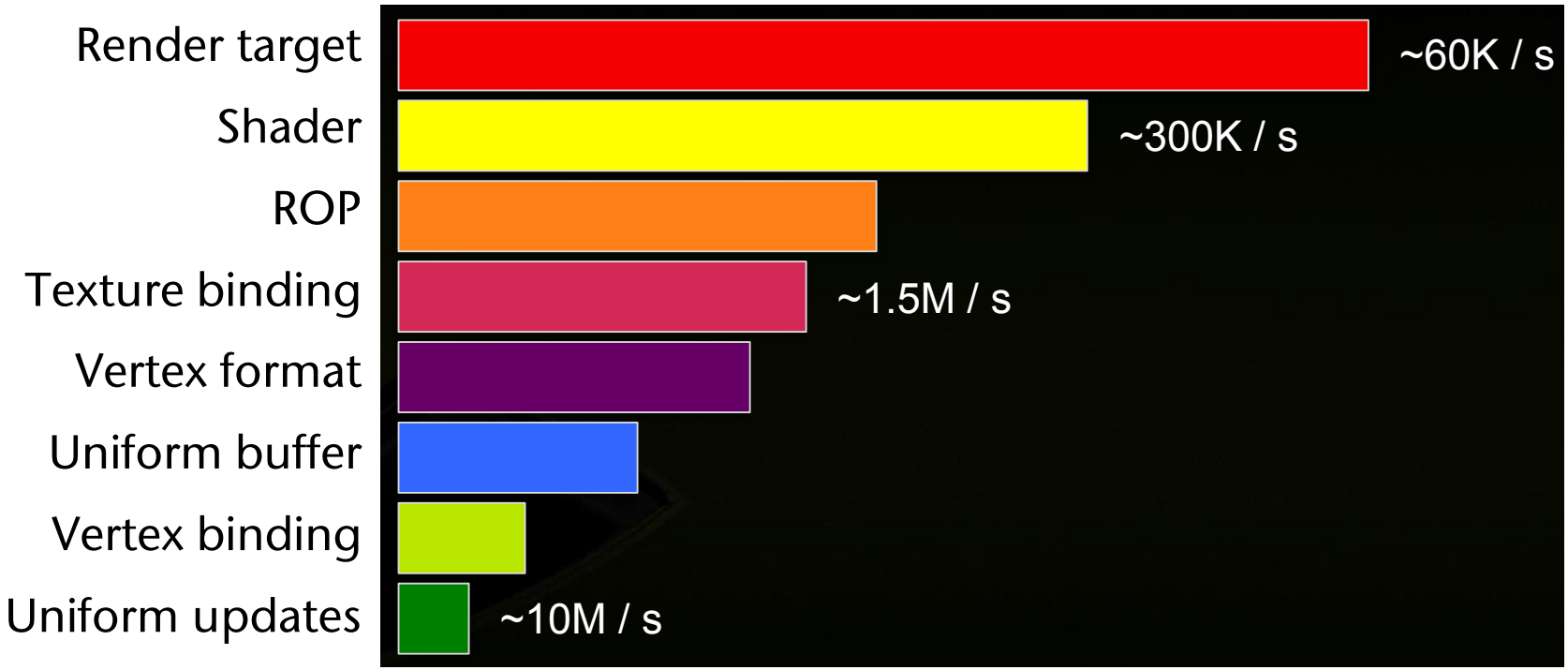
- State in OpenGL rendering =
  - Combination of all attributes
  - Examples for attributes: color, material, lighting parameters, textures being used, shader program, render target, etc.
  - At any time, each attribute has exactly 1 value out of a set of possible attributes (e.g., color ∈ { (0,0,0),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



- State changes are a serious performance killer!
- Costs in old OpenGL:



- Costs of state changes in modern OpenGL [2014]:



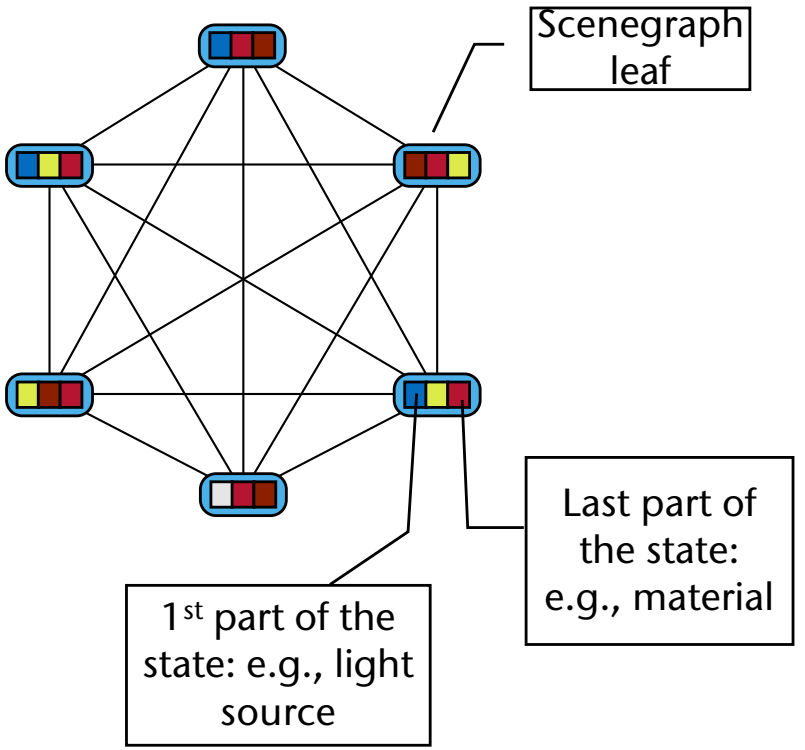
Not to scale!

- Goal: render complete scene graph with *minimal* number of state changes

Solution: Sorting by State

- Problem: optimal solution is NP-complete
- Proof:
  - Each leaf of the scene graph can be regarded as 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

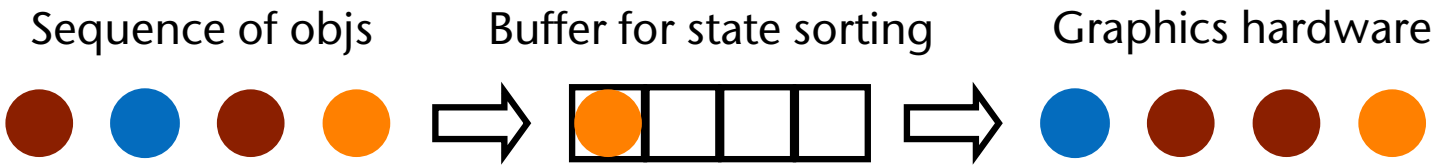
  →*Traveling Salesman Problem*



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

Introducing the Sorting Buffer

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



Online Algorithms

- There are 2 categories of algorithms:
  - "Online" algorithms: algo does *not* know which elements will be received in the future!
  - "Offline" algorithms: algo *does* know 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 wlog. 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 $C_{off}(k)$ = costs of *optimal* offline strategy,
  let $C_{on}(k)$ = costs of *some* online strategy,
  cost = number of color changes, k = buffer size.

  Then, the online strategy is called "c-competitive" iff

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

  where a must not depend on k (c may depend on k).

  The ratio

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

  is called the competitive-ratio.

- Wanted: an online strategy with $c(k)$ as small as possible (i.e., $c(k)$ should be in a low complexity class)

Example: LRU strategy (Least-Recently Used)

- The strategy:
  - Maintain a timestamp per color (not per element!)
  - When 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 $m = \sqrt{k-1}$, wlog. m is 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: $(m+1)\cdot 2 \cdot \frac{m}{2}$ color changes
  - Costs of the offline strategy: $2 \cdot \frac{m}{2} + m = 2m$ 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

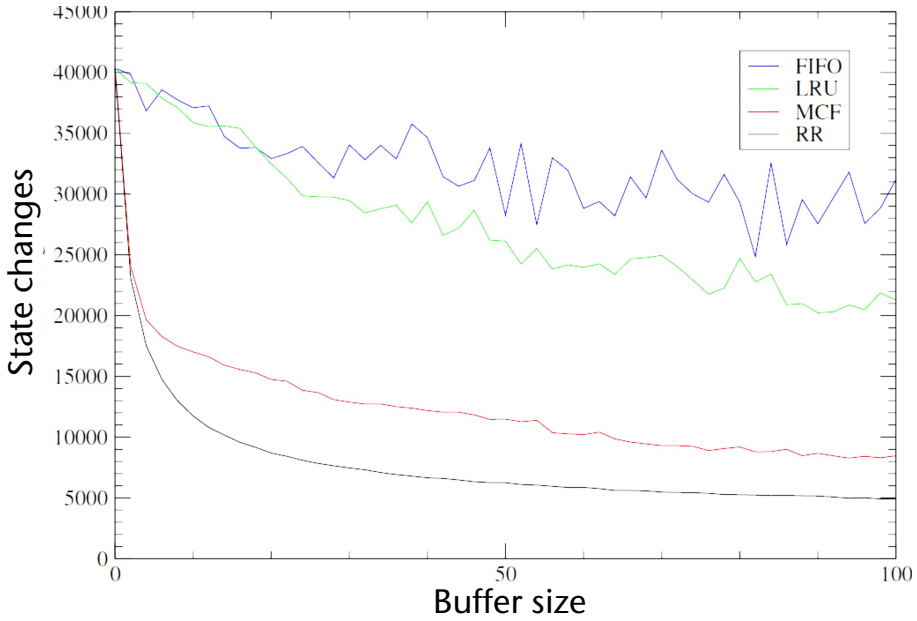
- Idea:
  - Count the number of all elements in buffer that have the *same* color
  - Extract those elements whose color is most prevalent in the buffer
- Introduce waste counter $W(c)$:
  - With new element on input side: increment $W(c)$, c = color of new element
- Bounded waste strategy:
  - With 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

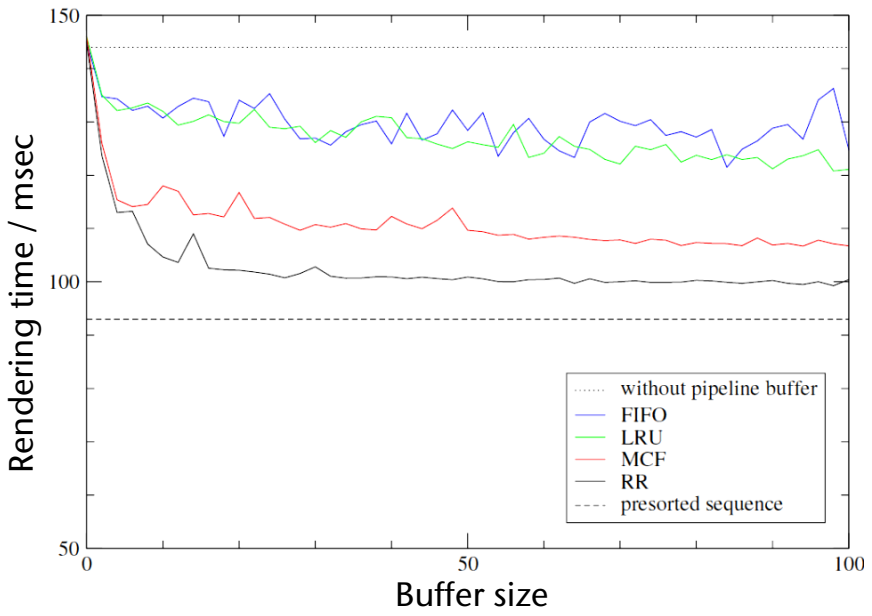
The Round Robin Strategy

- 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

Comparison

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





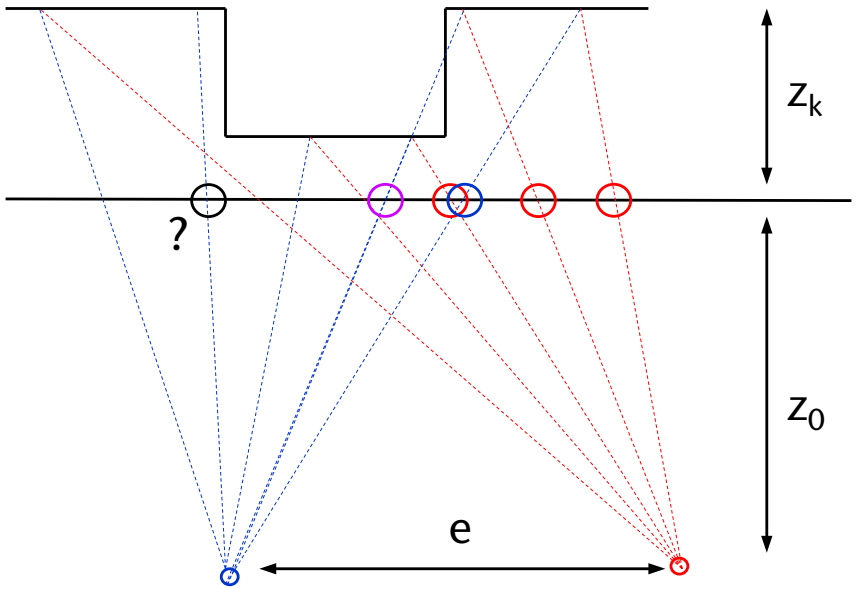
Stereoscopic Image Warping (Stereo *without* 2x rendering)

- Observation: left & right image differ not very much
- Idea: render once for right image, then move pixels to corresponding positions in left image → image warping
- Algo: consider all pixels on each scanline *from right to left*, draw each pixel *k* at the new x-coordinate

$$x'_k = x_k + \frac{e}{\Delta}\frac{z_k}{z_k + z_0}$$

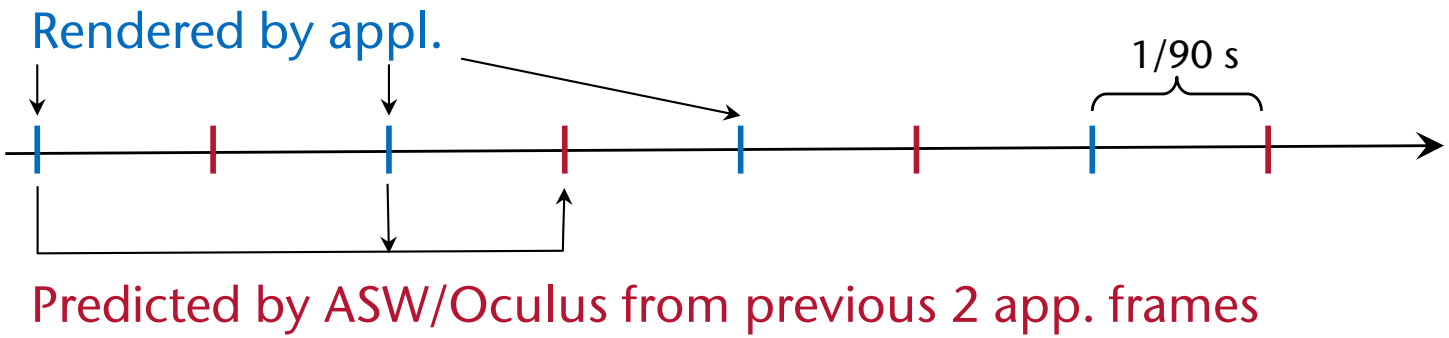
Δ = pixel width

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

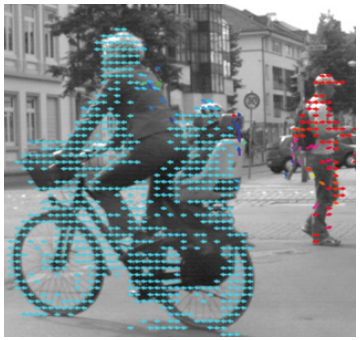


"Asynchronous Spacewarp" (Oculus)

- Oculus display refreshes at 90 Hz; if appl. can render only at 45 Hz, ASW produces frames "in between" by prediction:



- Some details about the method (guessed):
  - Extra thread kicks in if app has not finished rendering in time; stops rendering and graphics pipeline (*GPU preemption*)
  - Take previous two images, predict 2D motion of image parts
    - Optical flow algorithms? use GPU video encoding hardware?
  - Fill holes by stretching neighborhood (image inpainting)

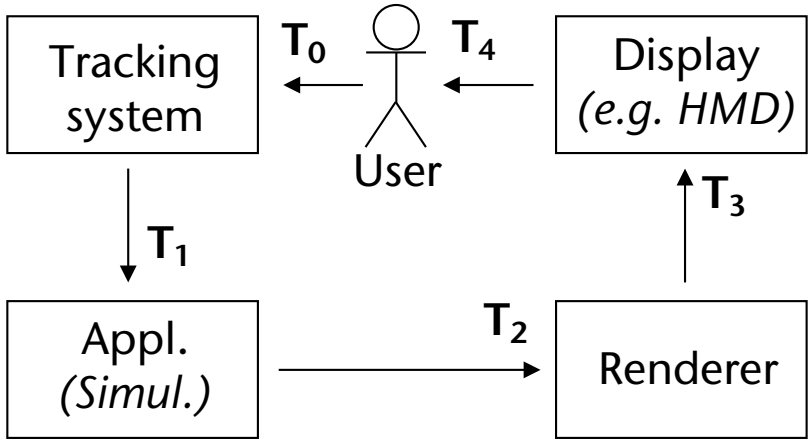


Example Frames

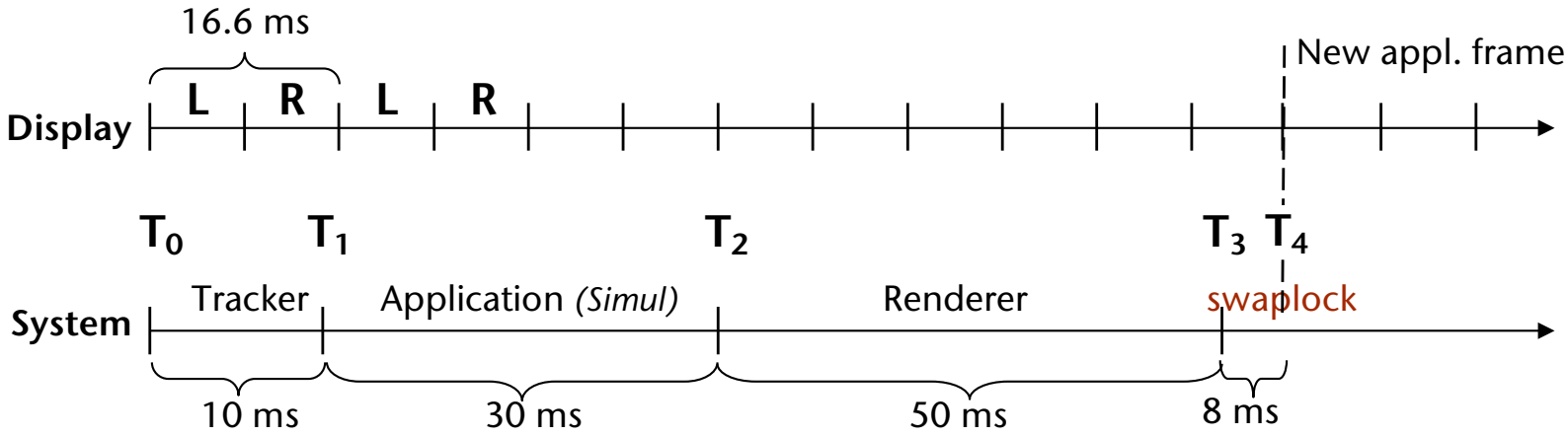
- Can you spot the artefacts?

Reducing Latency by Image Warping

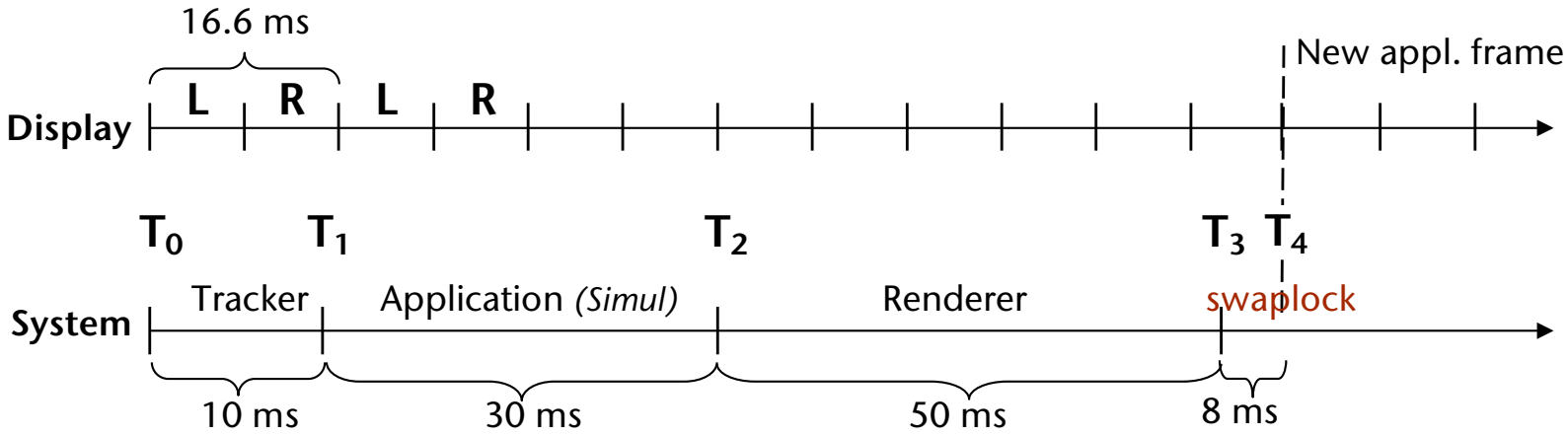
- A naïve VR system:



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

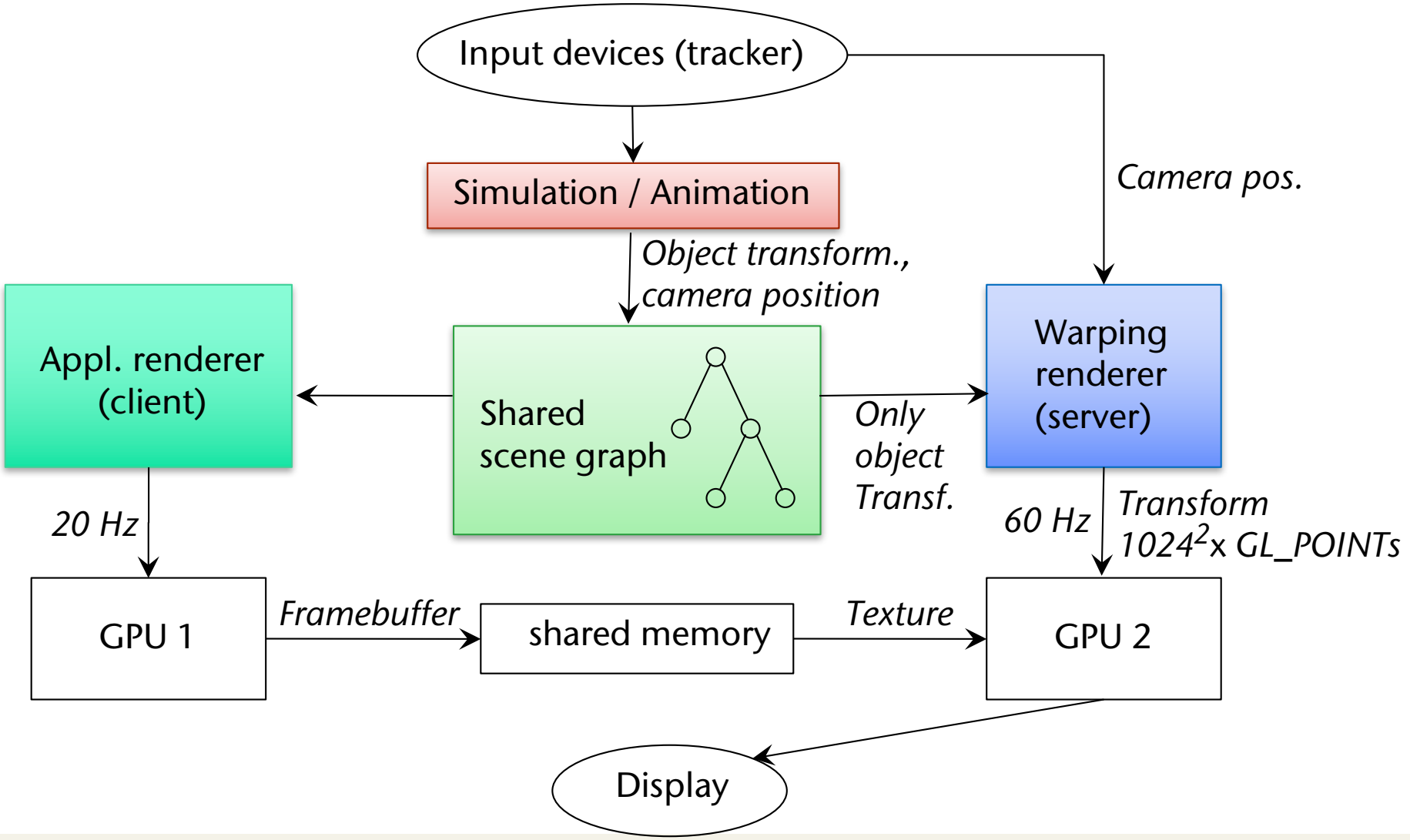


- Problems / observations:
  - The appl. framerate (incl. rendering) is typically much slower than the display refresh rate
  - The tracking data, which led to a specific image, were valid in the "distant" past
  - The tracker could deliver data more often
  - Consecutive frames differ from each other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



Idea of the Solution

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



An Application Frame (Client)

- At time t_1, the application renderer generates a normal frame
  - Color buffer and Z-buffer
  - Henceforth called "application frame"
- ... but also saves additional information:
  1. With each pixel, save ID of object visible at that pixel
  2. Save 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)

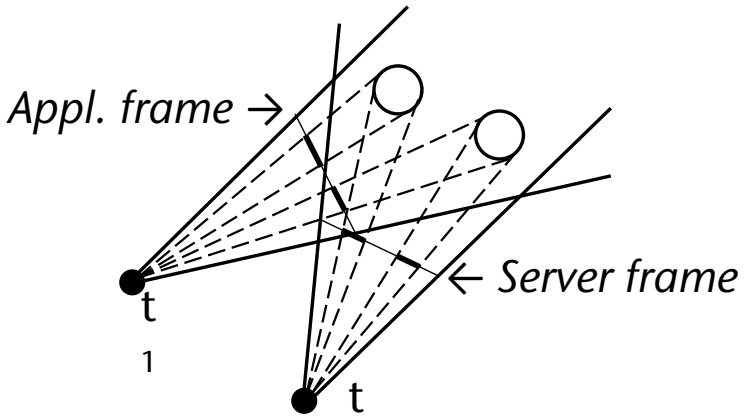
- At a later time t_2 , the server generates an image from an application frame by warping
- Transformations at this time:

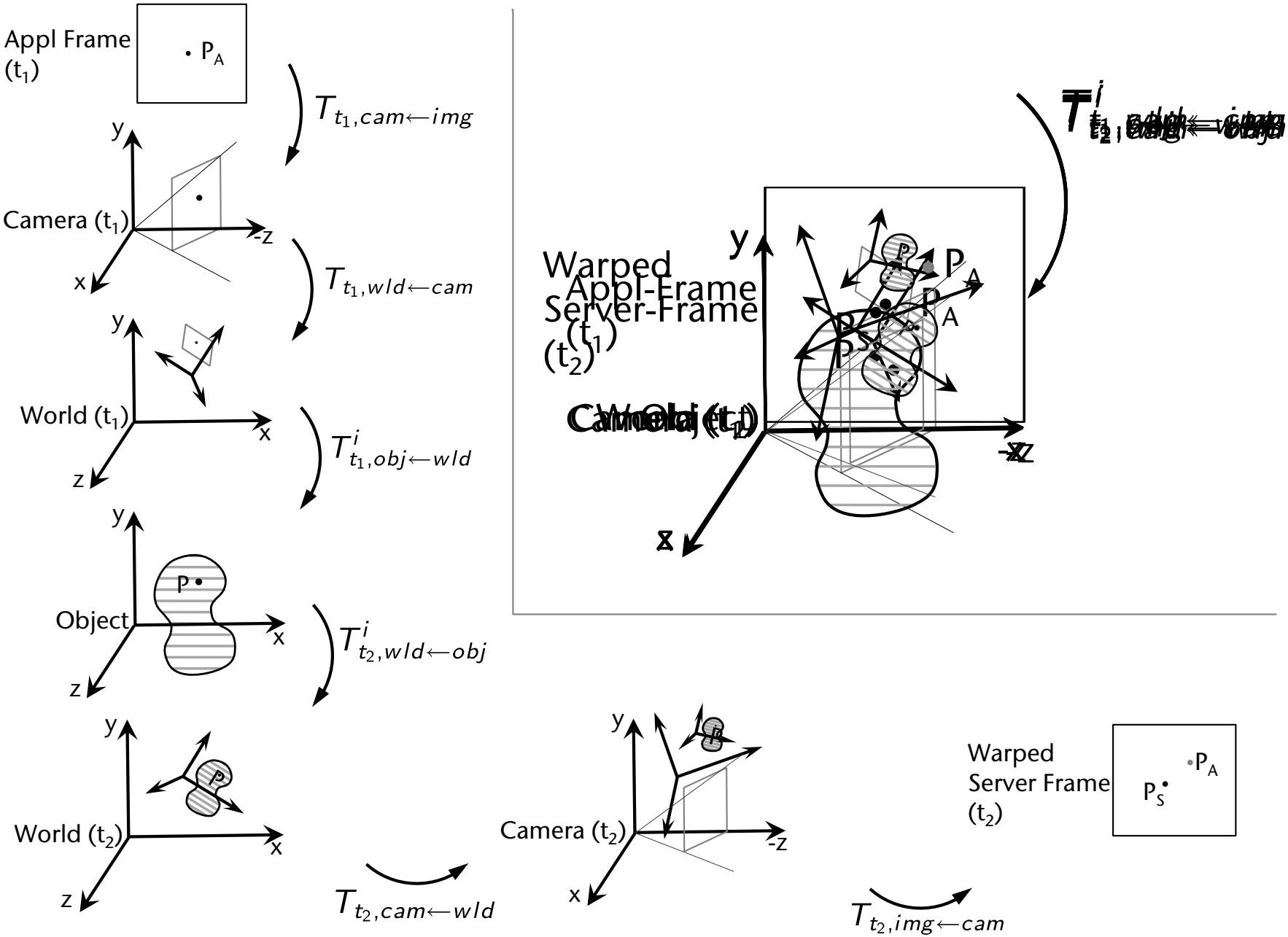
$$T^{i}_{t_2,wld\leftarrow obj} \qquad T_{t_2,img\leftarrow cam} \qquad T_{t_2,cam\leftarrow wld}$$

- A pixel $P_A = (x, y, z)$ in the appl. 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 and each new server frame



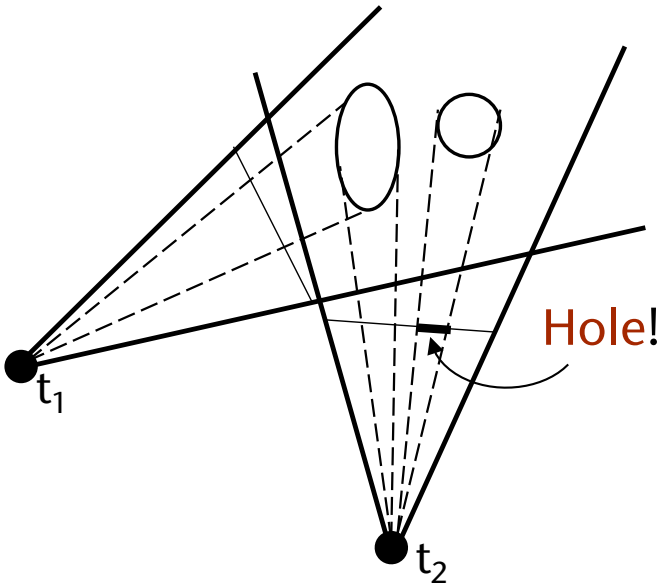

Appl Frame
(t1)
PA
Tt1,cam←img
y
Camera (t1)
-z
x
Tt1,wld←cam
y
World (t1)
x
z
T^i t1,obj←wld
y
P
Object
x
z
T^i t2,wld←obj
y
P
World (t2)
x
z
Tt2,cam←wld
y
Camera (t2)
-z
x
Tt2,img←cam
Warped
Server Frame
(t2)
PA
PS
Warped
Appl-Frame
Server-Frame
(t1)
(t2)
y
PA
PS
x
-z

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 appl. frame as a texture , and it loads all the T_i's
  - Render 1024x1024 many GL_POINTs (called point splats)

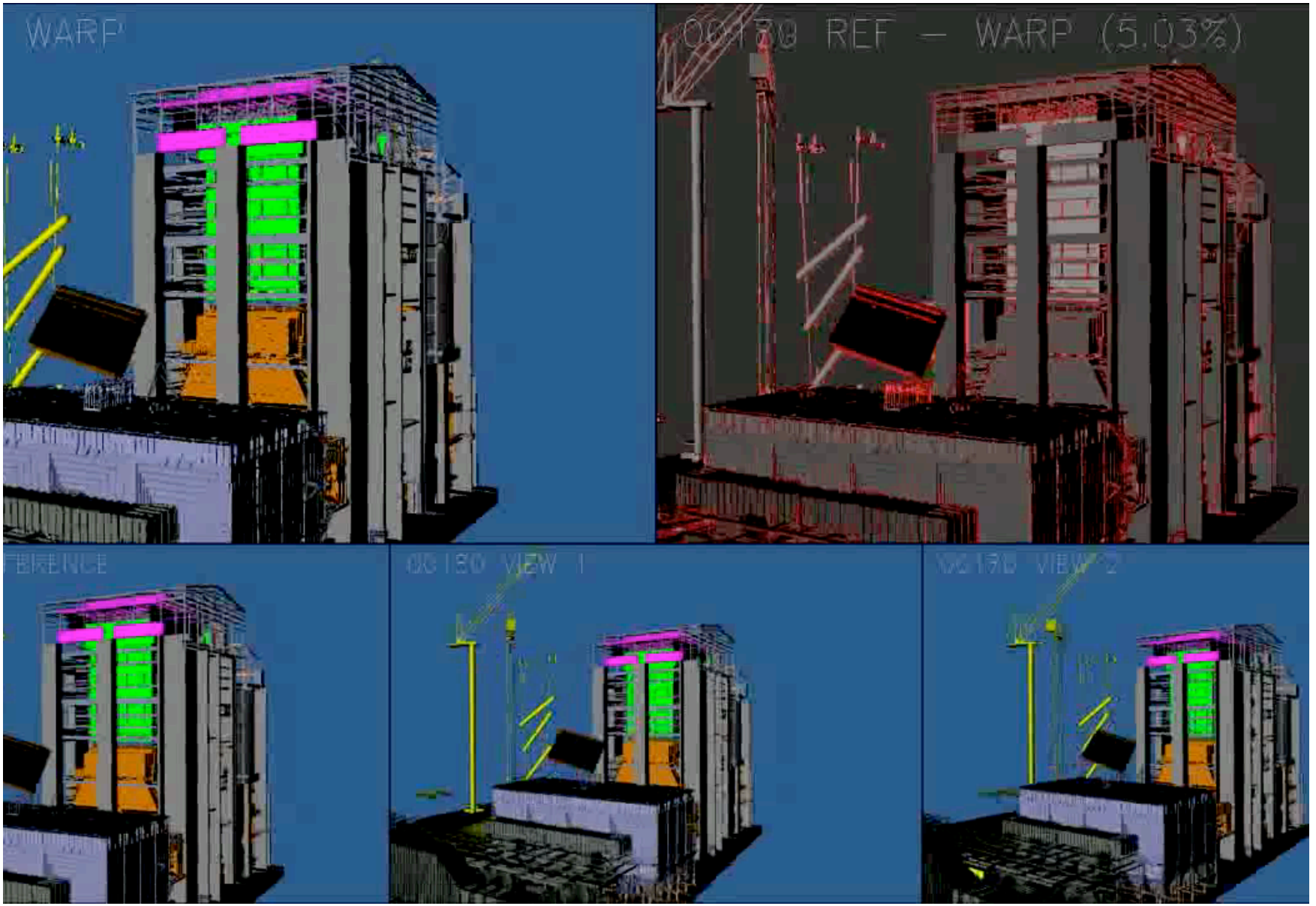
- Advantages:
  - The frames (visible to the user) are now "more current", because of more current camera and object positions
  - Server framerate is independent of number of polygons

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



- Performance gain:
  - 12M polygons, 800 x 600 frame size
  - Factor ~20 faster

Videos


WARP
REF – WARP (5.03%)