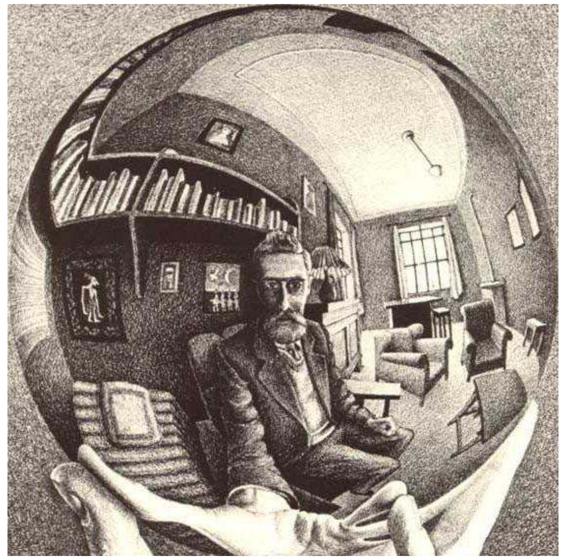
Bremen

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



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

Your Guess: What is the Latency Humans Can Notice?

Please,
don't spoil by
"look-ahead"!

https://www.menti.com/smvndia2ss

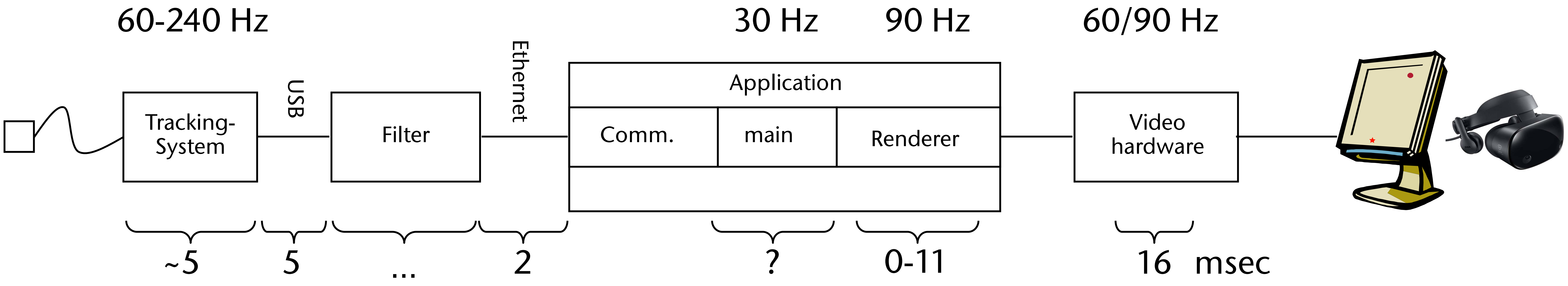
Latency (Lag, Delay)

- 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
> 500	Presence vanishes (and simulation fidelity)

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

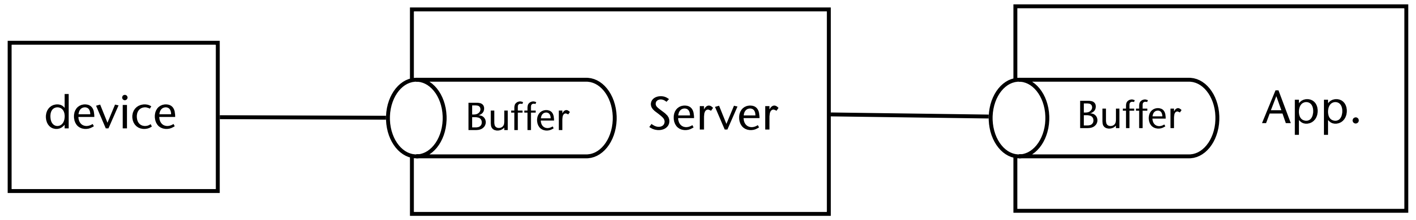
The Latency Pipeline

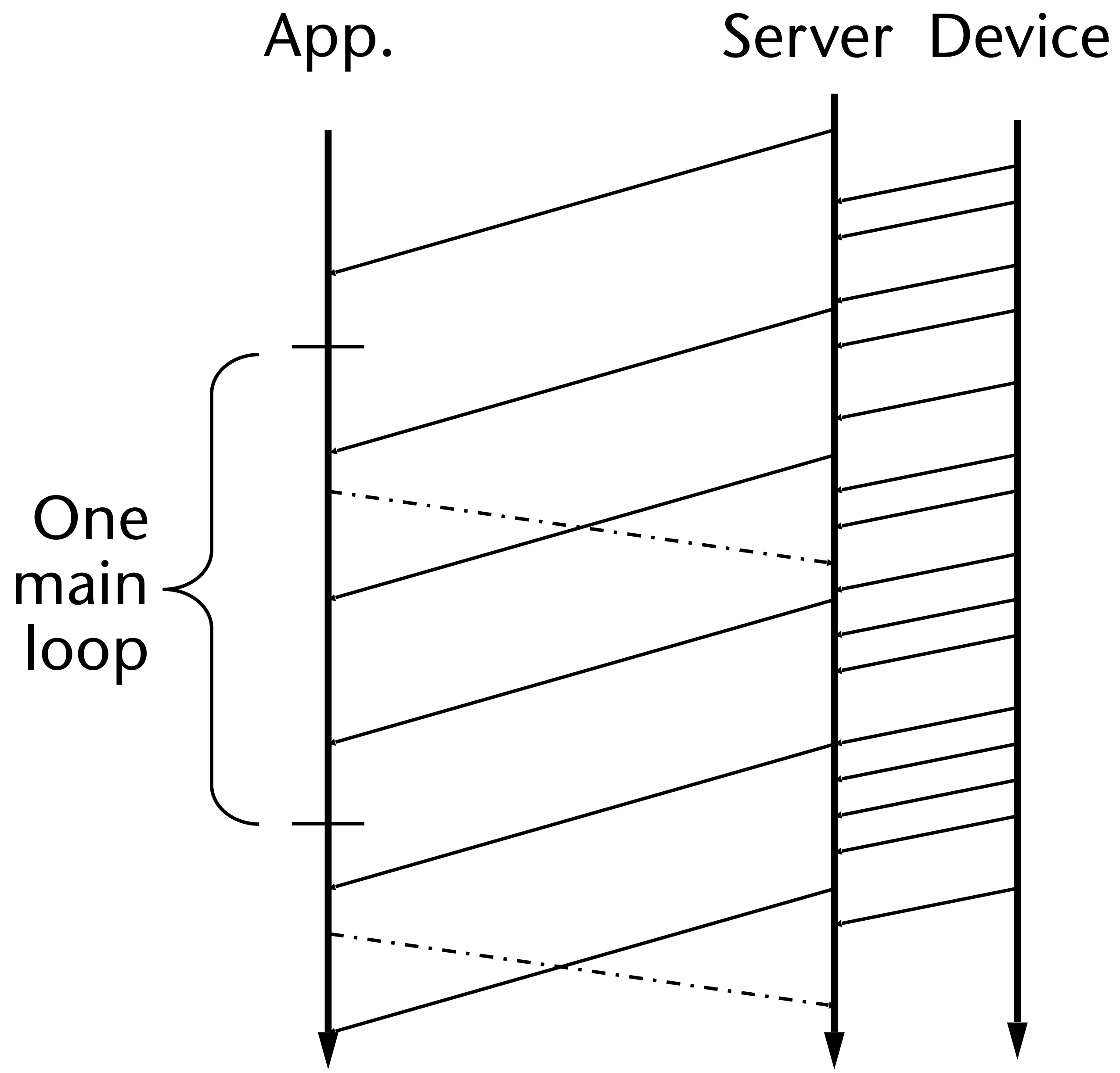


- Types/causes of lag:
  - Internal to devices
  - Transportation of data over communication channel (e.g., Ethernet)
  - Software (time for processing the data)
  - Synchronization delay

General Strategies for Solutions

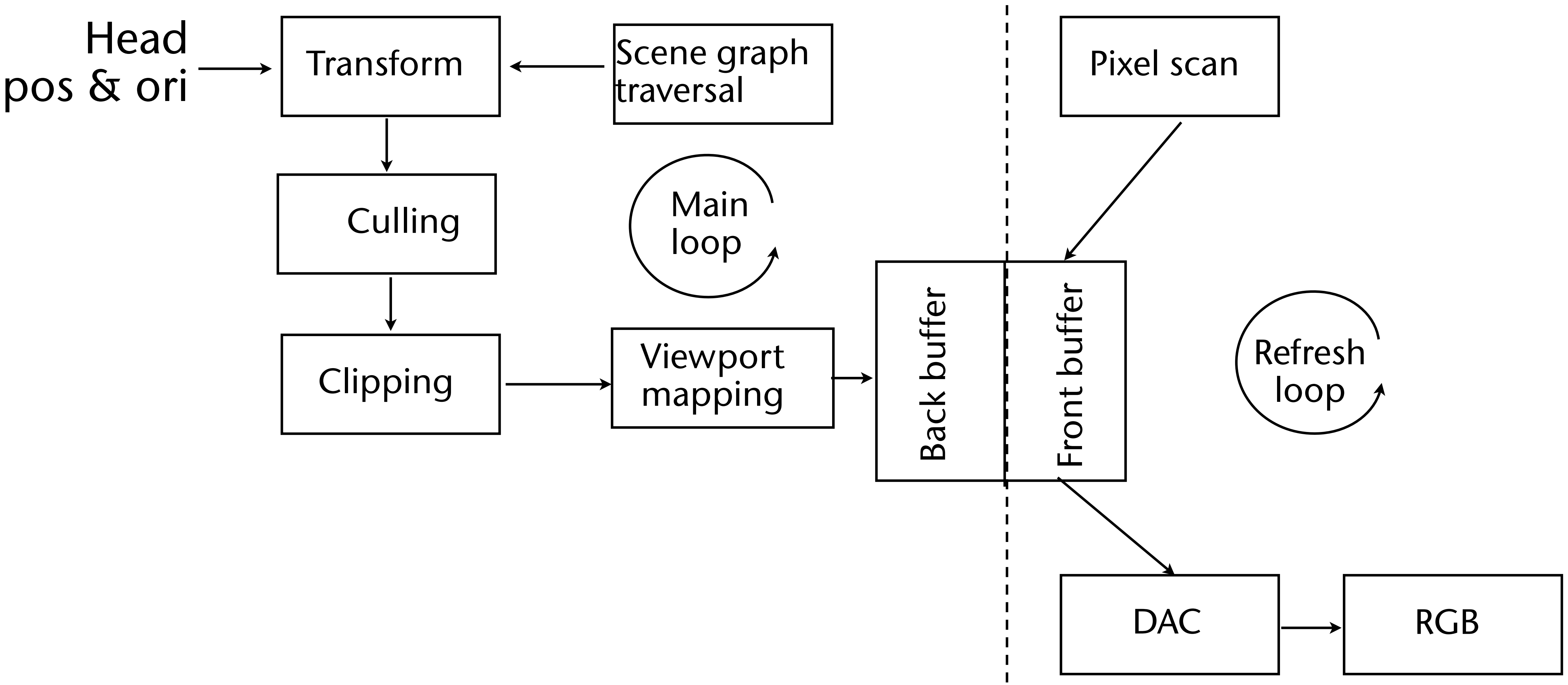
1. Device-server-app communication:
   - Put device and server into continuous mode
   - Send "keep alive" messages from client to server
2. Do time-critical computing:
   - Each and every module of the app receives a specific time budget
   - Module tries to compute a usable(!) partial solution as good as possible within the time budget
   - Stop when time is up
3. Try to predict user/tracker position in *x* msec's time



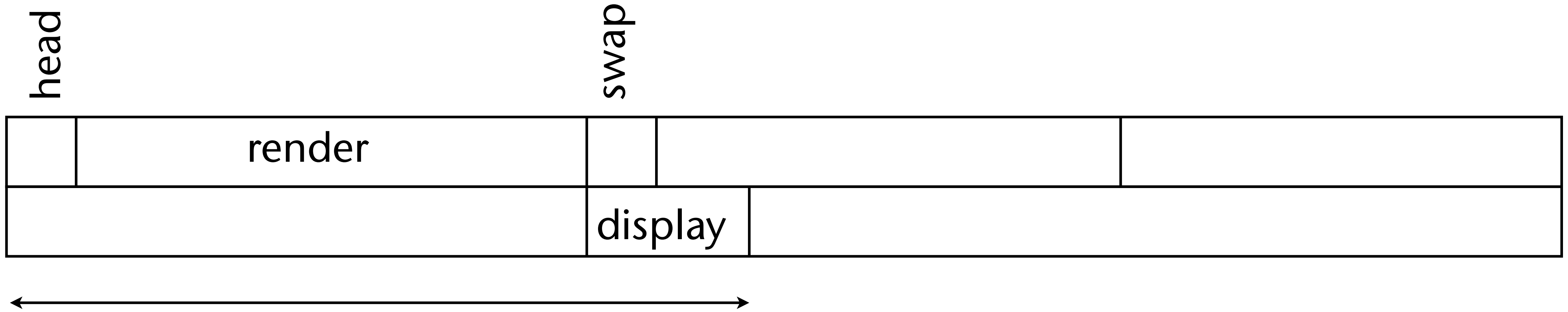


Sources of Latency During Rendering

- The classical graphics pipeline, at least parts of it, visualized as a loop:

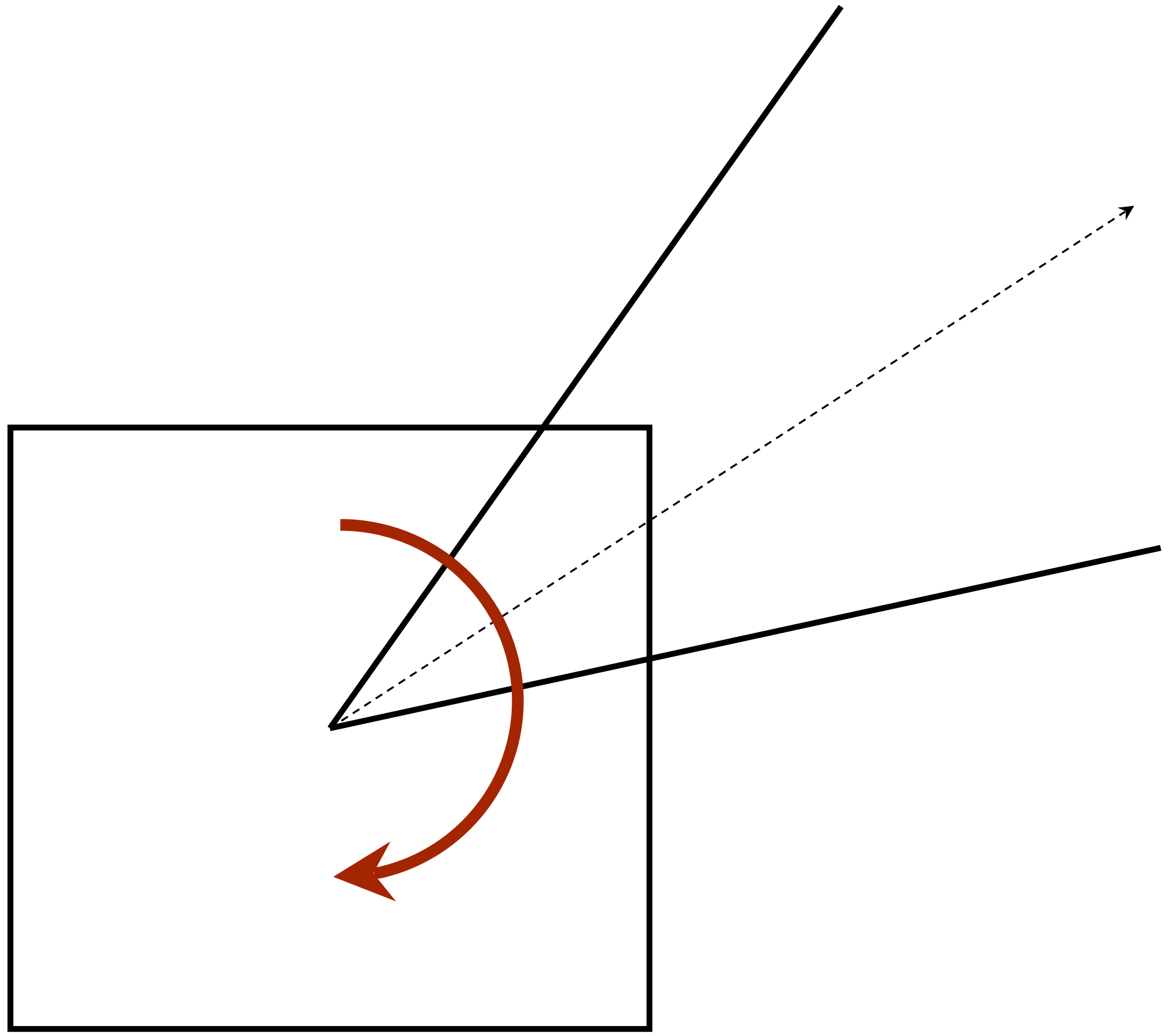


- Latency:

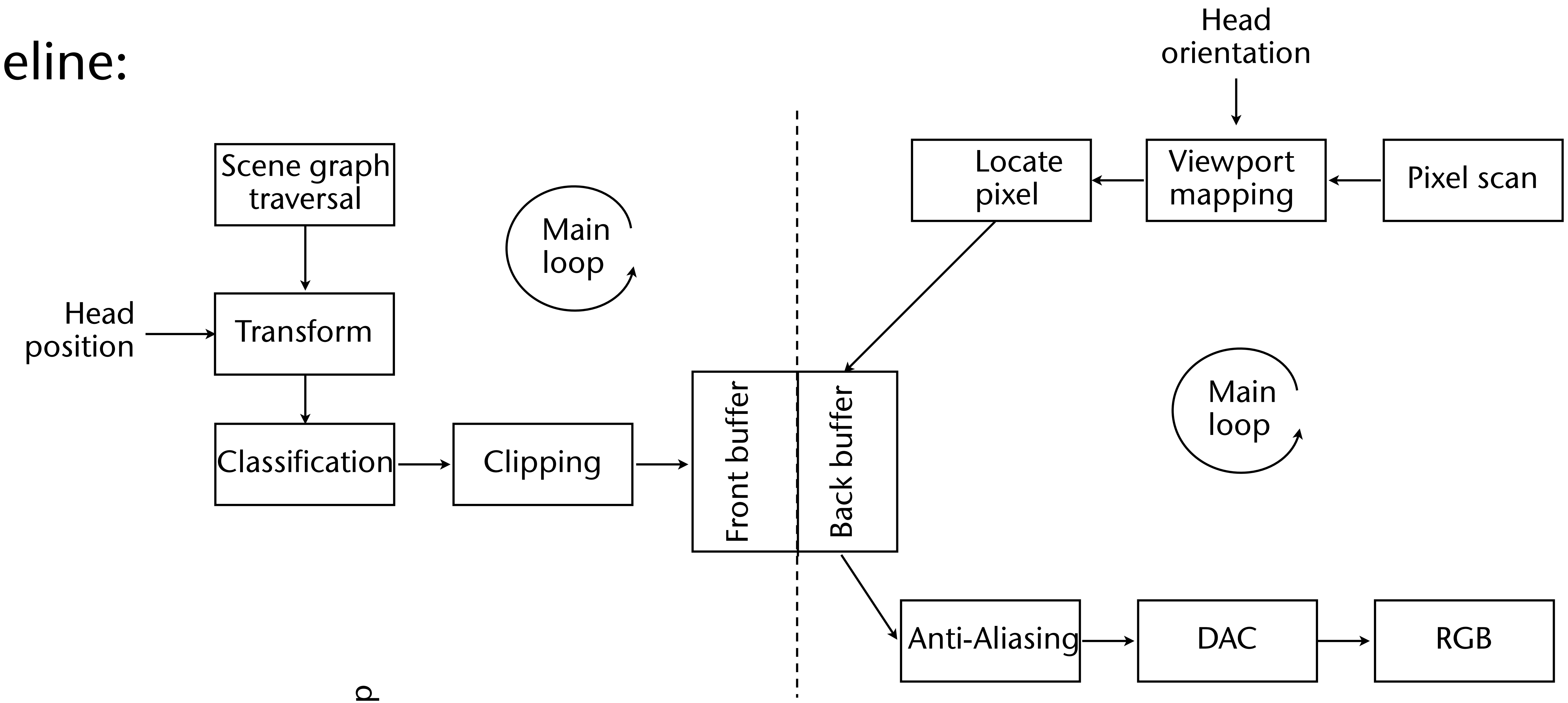


Viewport-Independent Rendering

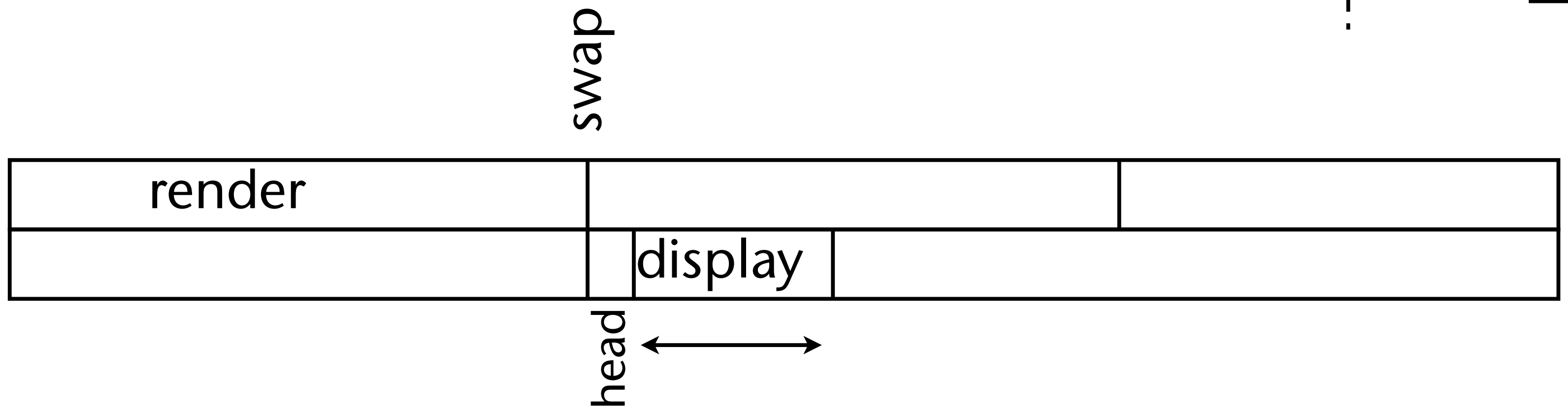
- Conceptual idea:
  - Render the scene onto a *sphere* around the viewer ⟶ spherical viewport
  - If viewpoint rotates: just determine new cutout of the spherical viewport
- Practical implementation:
  - Use a cube as a viewport around user, instead of sphere
  - Remark: this was also one of the motivations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



- New pipeline:

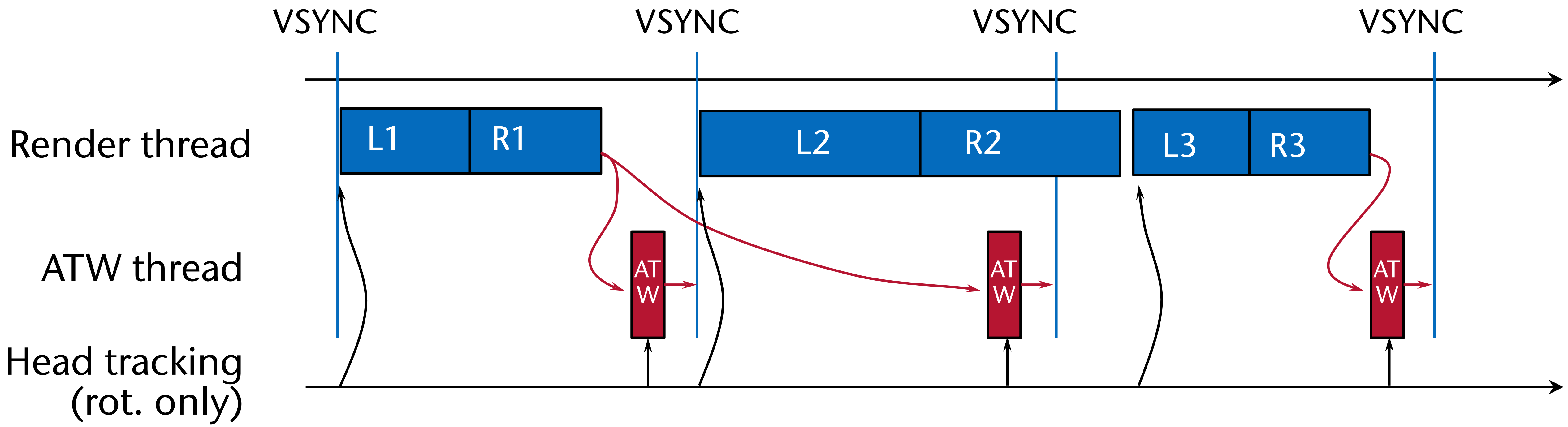


- Latency:



"Asynchronous Timewarp" (Oculus)

- Render a bigger-than-visible viewport (not the whole cube)
- Shift image using current orientation of head
- Do this only in case the renderer is not finished in time:



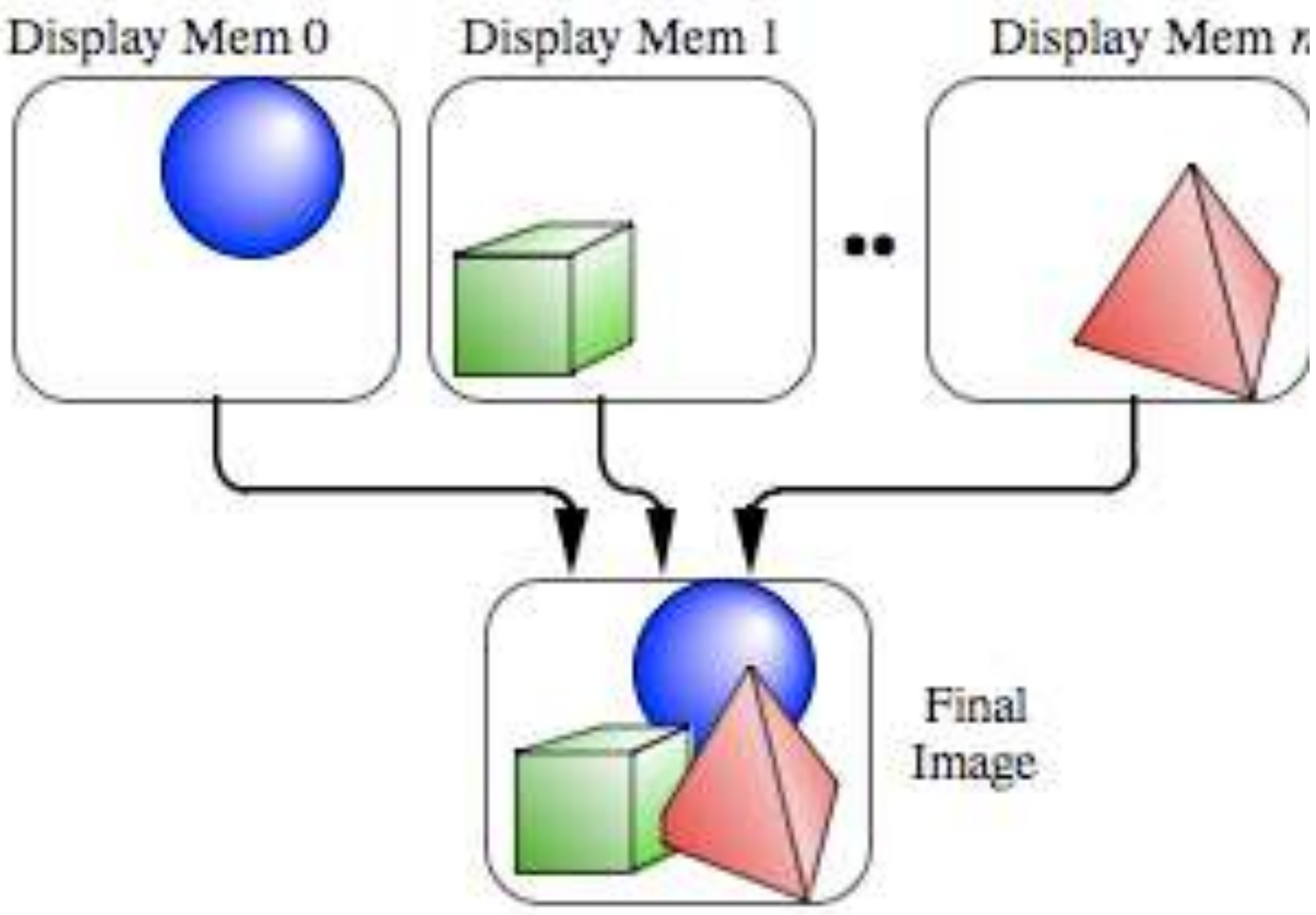
- Requires GPU preemption (i.e., stop GPU's pipeline, including shaders, immediately)

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)

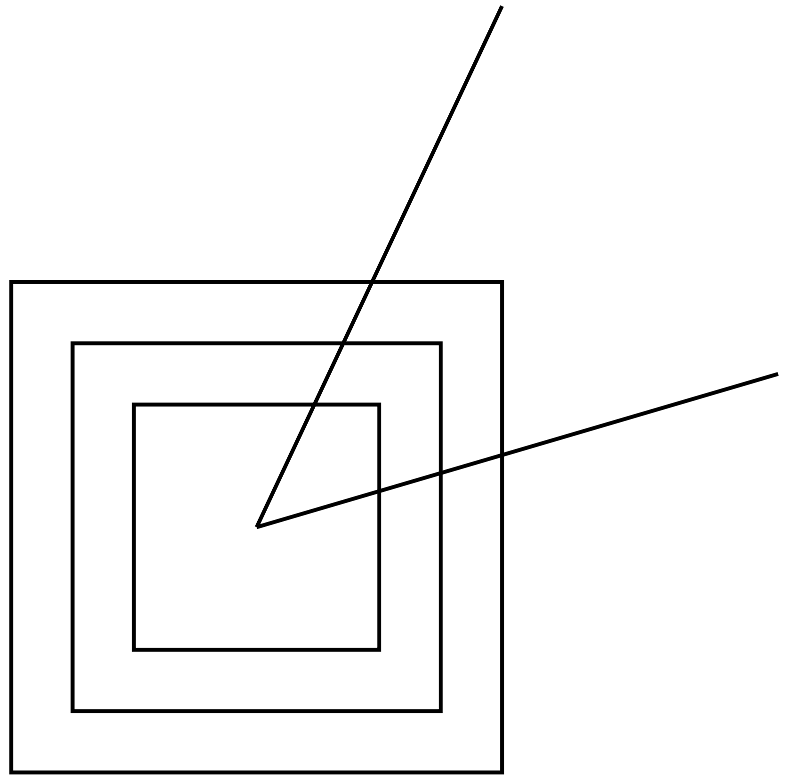
Multi-Threaded Rendering and Image Composition

- 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 → update objects "behind" only rarely
  - Objects being manipulated must have highest priority
  - Objects in peripheral field of vision can be updated less often

What Are Some Good Software-Engineering Practices?

https://www.menti.com/smvndia2ss

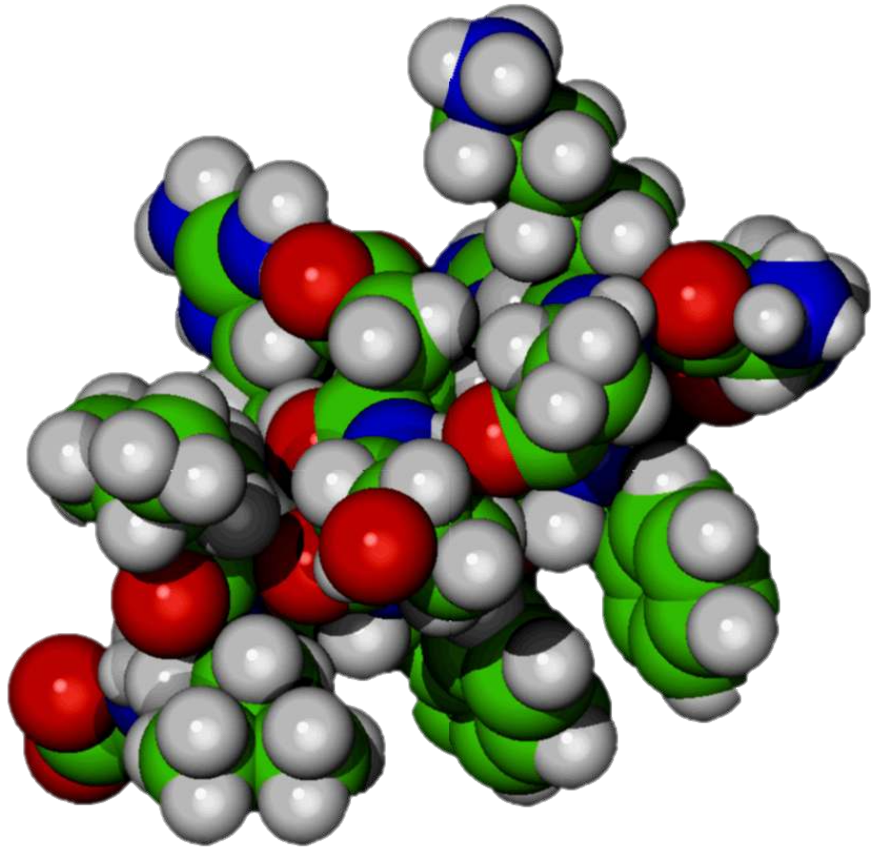
Efficient Memory-Layout for Fast Rendering

- Frequent problem: the elegant way to structure data (from the perspective of software engineering) is inefficient for fast rendering
- Example for illustration: visualization of molecules
  - Following good SE practice, we should design classes like this

```
class Atom
{
public:
    Atom( uint atom_number, Vec3 position, ... );
private:
    Vec3   position_;
    uint   atom_number_;
    Atom * bonds_[max_num_bonds];
    ...
};
```

- And the class for a molecule:

```
class Molecule
{
public:
    Molecule( const std::vector<Atom> & atoms );
private:
    std::vector<Atom> atoms_;
    ...
};
```



- Memory layout of a molecule is now an array of structs (AoS):

pos	num	bonds	pos	num	bonds	pos	num	bonds

- Problem with that: memory transfer becomes slow
- Alternative: Struct of Arrays (SoA)

```
class Molecule
{
private:
    std::vector<Vec3>  position;
    std::vector<uint>  atom_number;
   ...
};
```

pos[0] pos[1] pos[2] . . .	atom_number[0] . . .	

- Terminology: "Array of Structs (AoS)" vs. "Struct of Arrays (SoA)"

Constant Framerate by "Omitting Stuff"

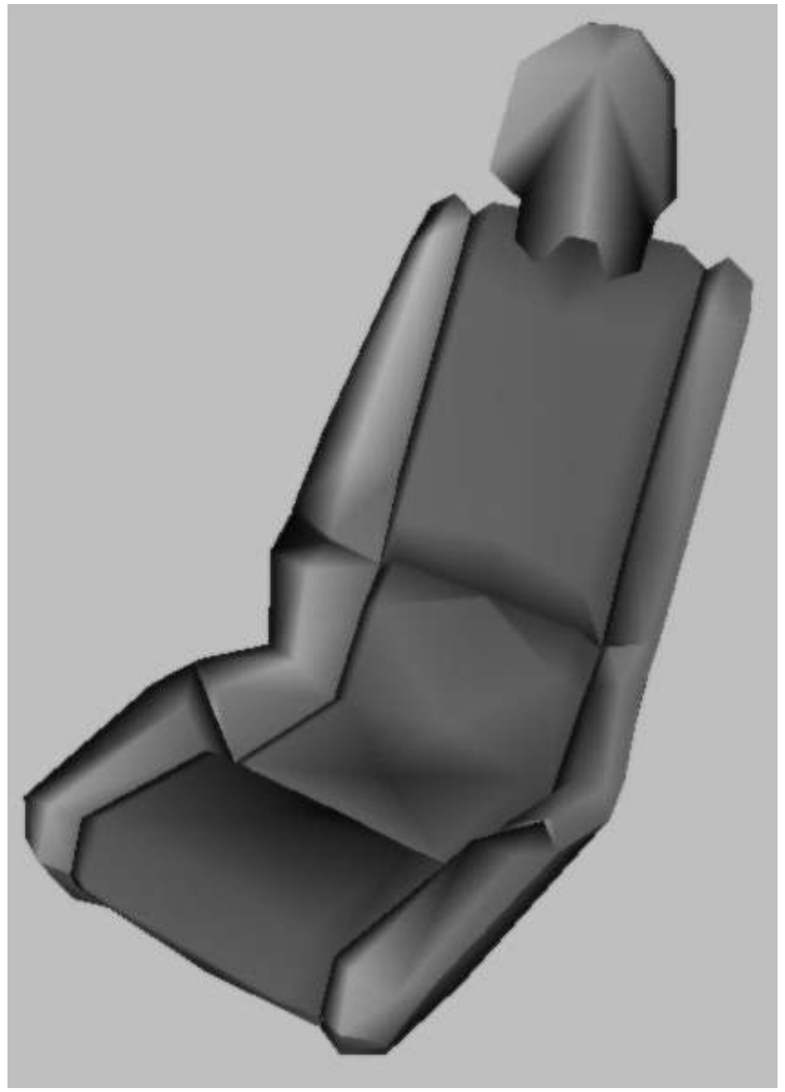
- Reasons for the need of a constant framerate:
  - Prediction in *predictive filtering* of tracking data of head/hands works only, if all subsequent stages in the pipeline run at a known (constant) rate
  - Jumps in framerate (e.g., from 90 to 45 Hz) are very noticeable (stutter/judder)
- Consider rendering as "*time-critical computing*":
  - Rendering gets a certain time budget (e.g., 11 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,

Which Things Could the Renderer Omit in Case of Overrunning the Time Budget?

https://www.menti.com/smvndia2ss

The Level-of-Detail (LoD) Technique

- Example: do you see a difference?

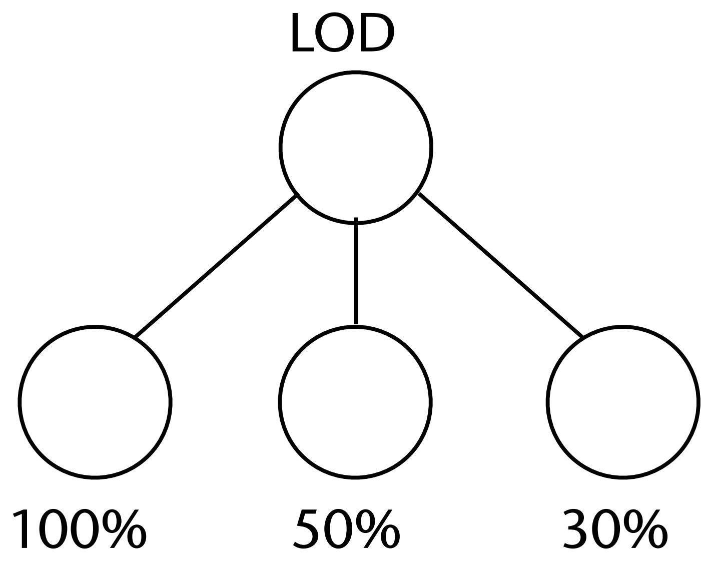


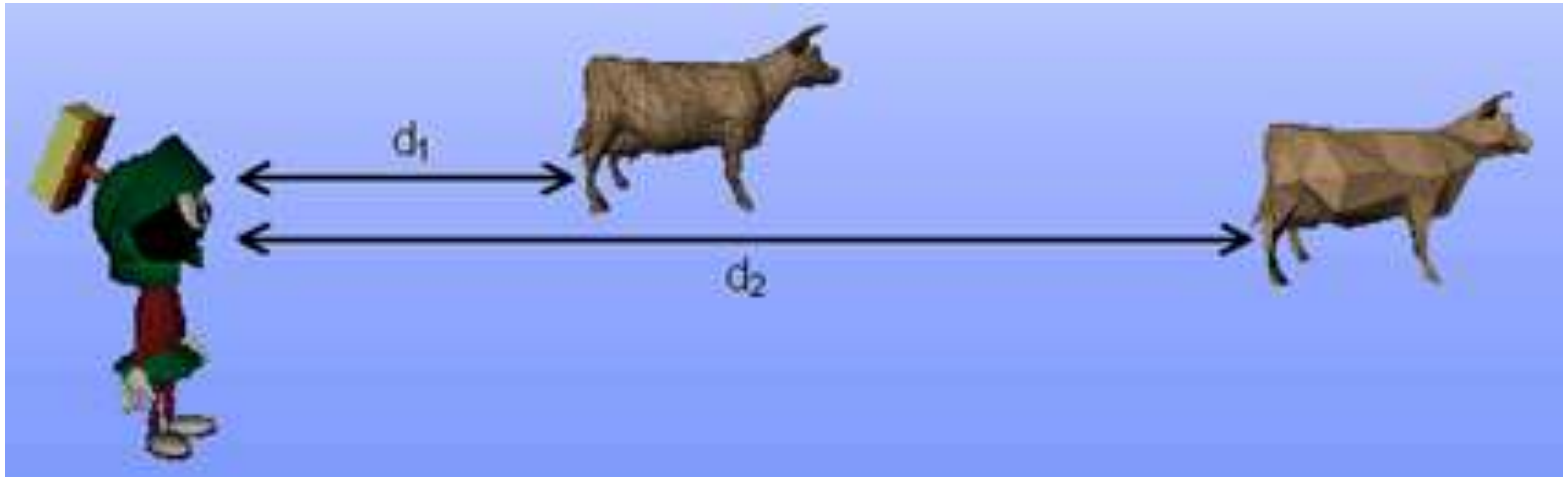
- Idea: render a reduced version of the object, where the amount of reduction is chosen such that users cannot see the difference from the full-resolution version

- Definition:
  A level-of-detail (LOD) of an object is a simplified version,
  i.e., a model 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 the number of polygons of the original model
  2. Runtime: select "right" LOD, make switches between LODs unnoticeable

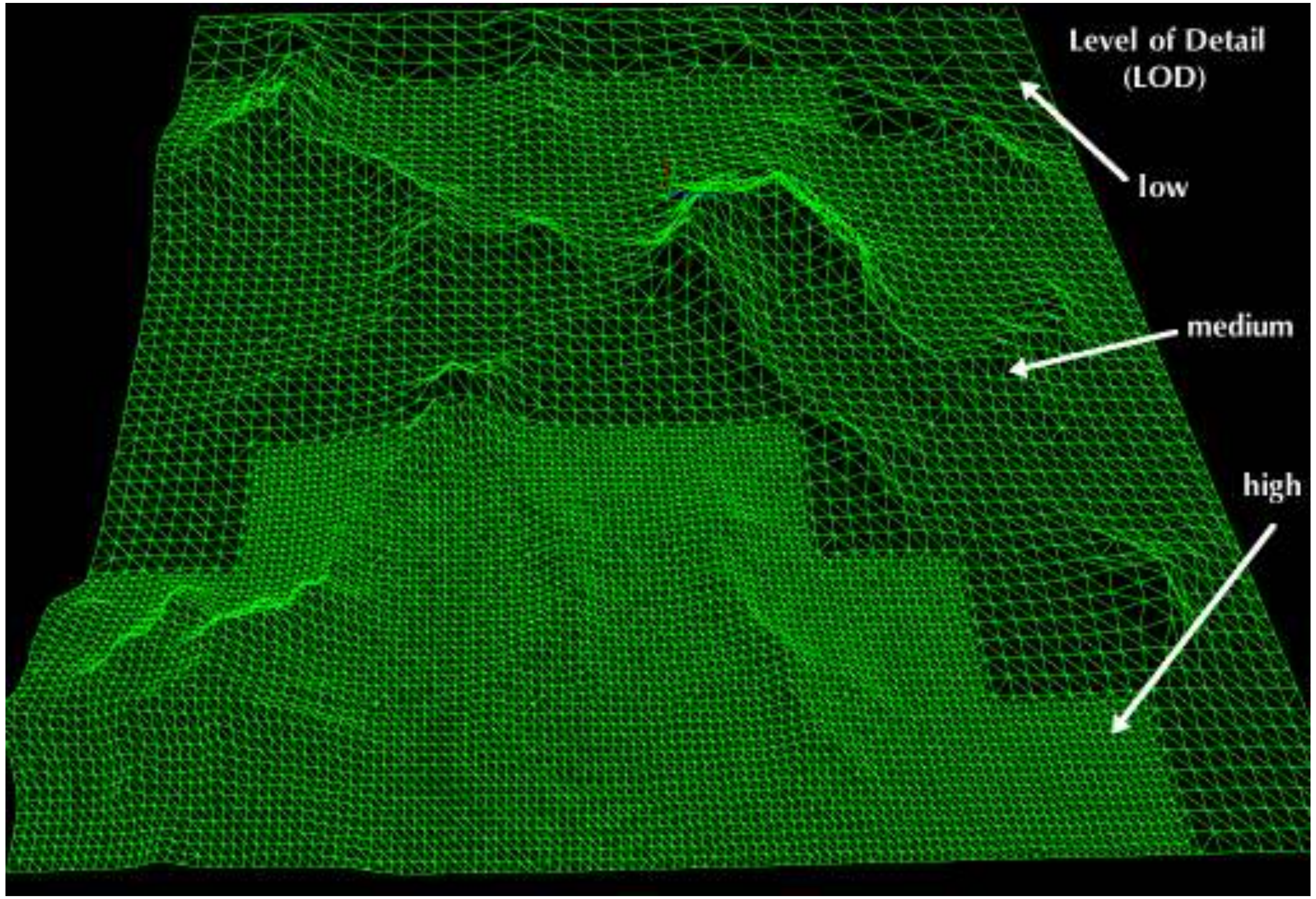
Selection of the LOD

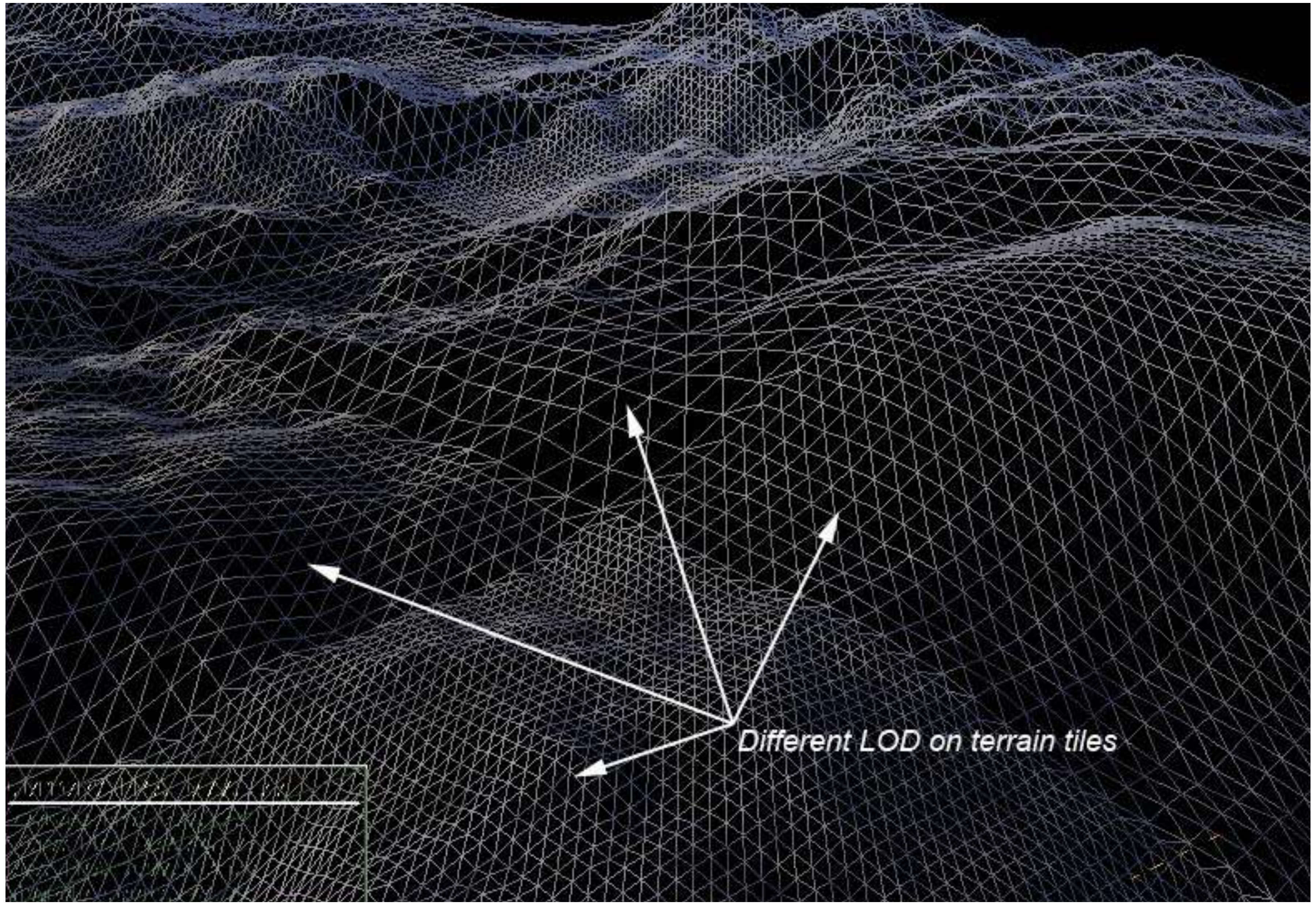
- Balance visual quality against "temporal quality"
- Static selection algorithm:
  - Level i for a distance range (d_i, d_{i+1})
  - Optimal distance ranges depend on FoV
  - Problem: size of objects is not considered





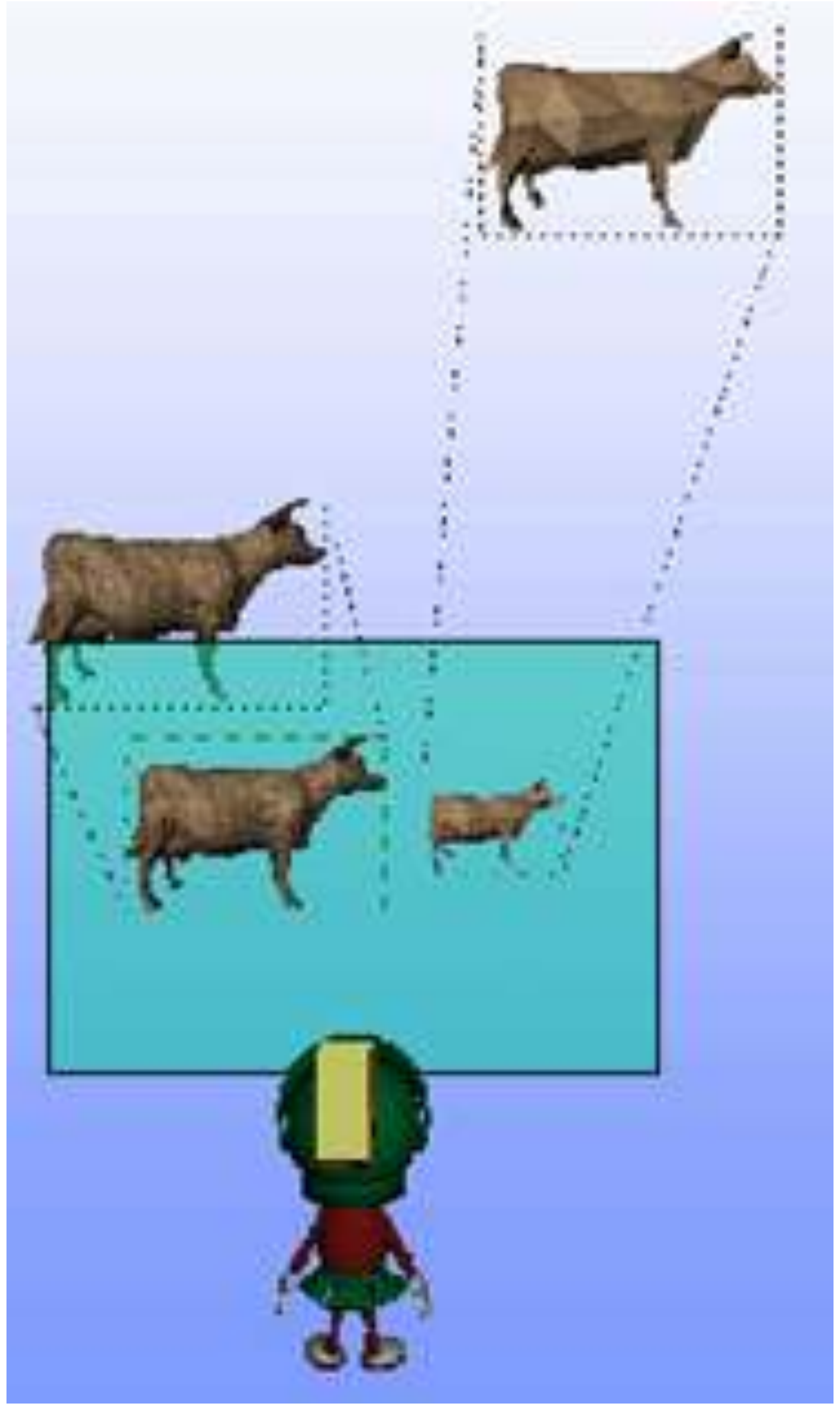
Typical Use Case: Terrain Rendering





Improved Static Selection

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

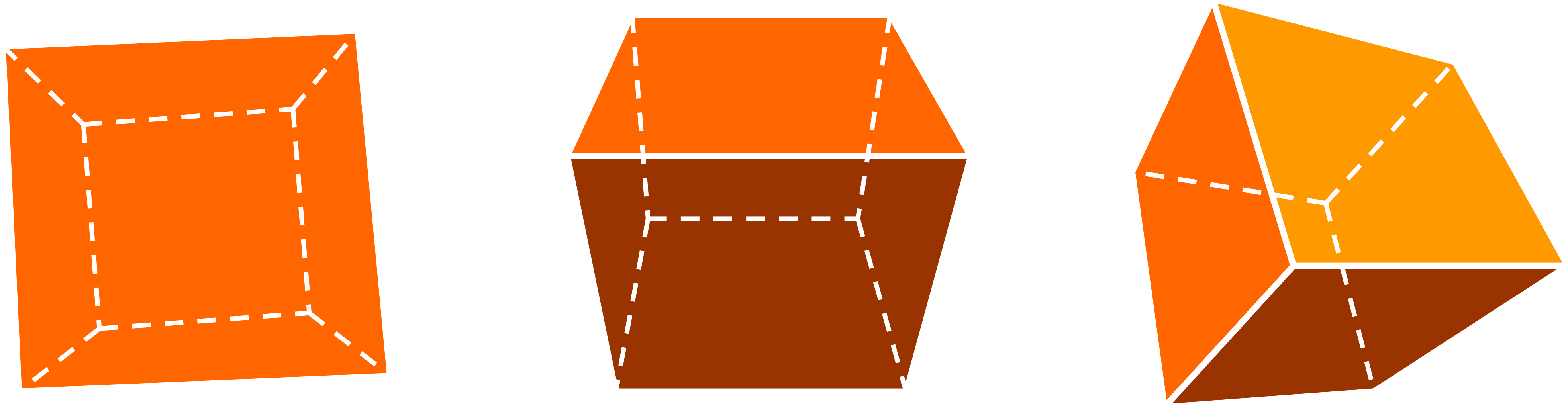


Estimation of the Size of an Object on the Screen

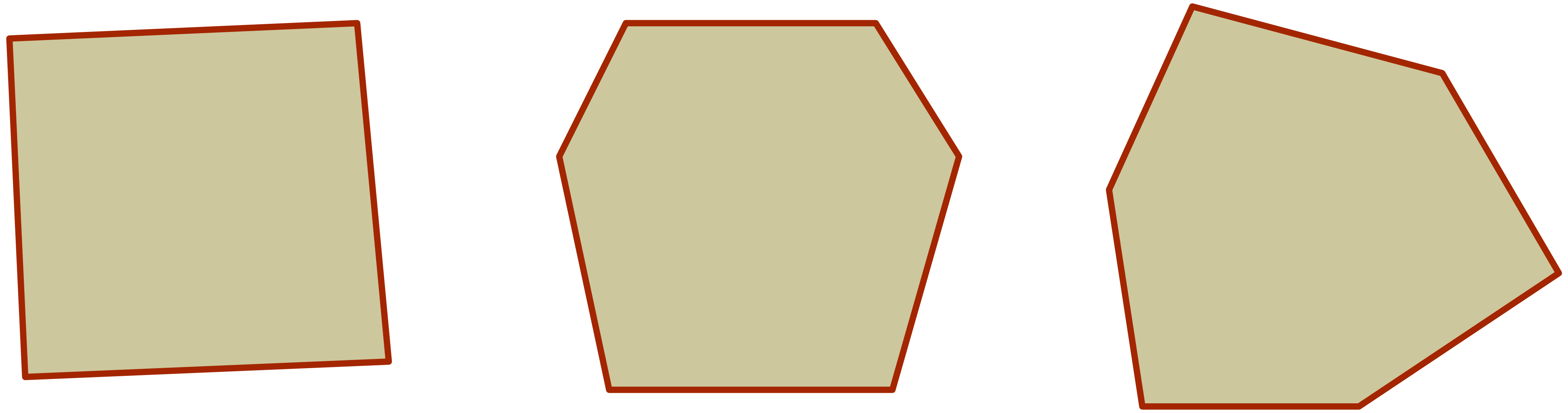
- Naïve method:
  - Compute bounding box (bbox) of object in 3D (probably already known by scenegraph for occlusion culling)
  - Project bbox onto 2D → 8x 2D points
  - Compute 2D bbox (axis aligned) around 8 points
- Better method:
  - Compute true area of projected 3D bbox on screen

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

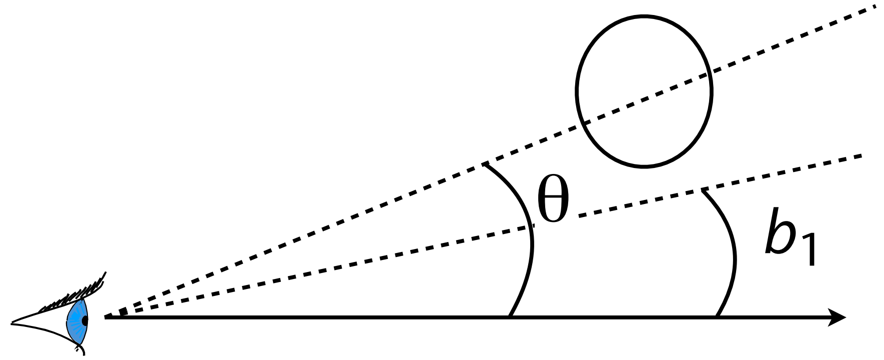
FYI

- 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)
- Each of the 27-1 entries of the LUT lists the 4 or 6 vertices of the silhouette
- Then, project, triangulate (determined by each case in LUT), and accumulate areas

Psychophysiological LOD Selection

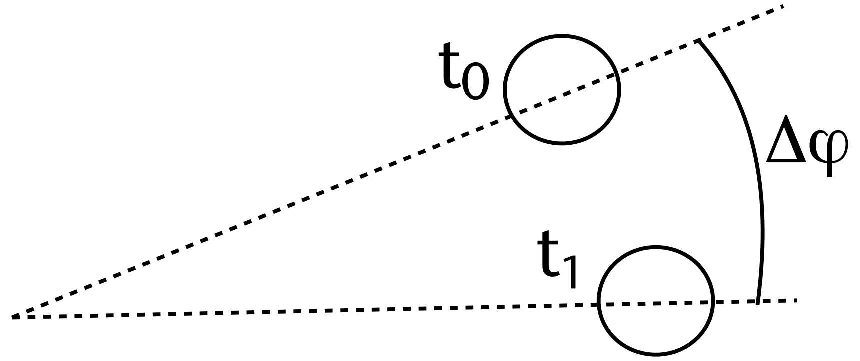
- Idea: exploit human factors with respect to visual acuity
  - Central / peripheral vision:

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



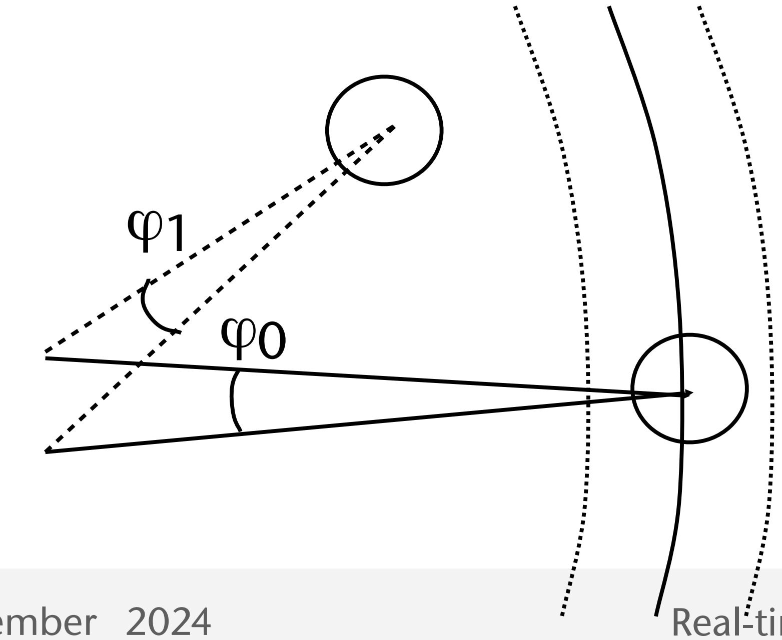
  - Motion of obj (relative to viewpoint):

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



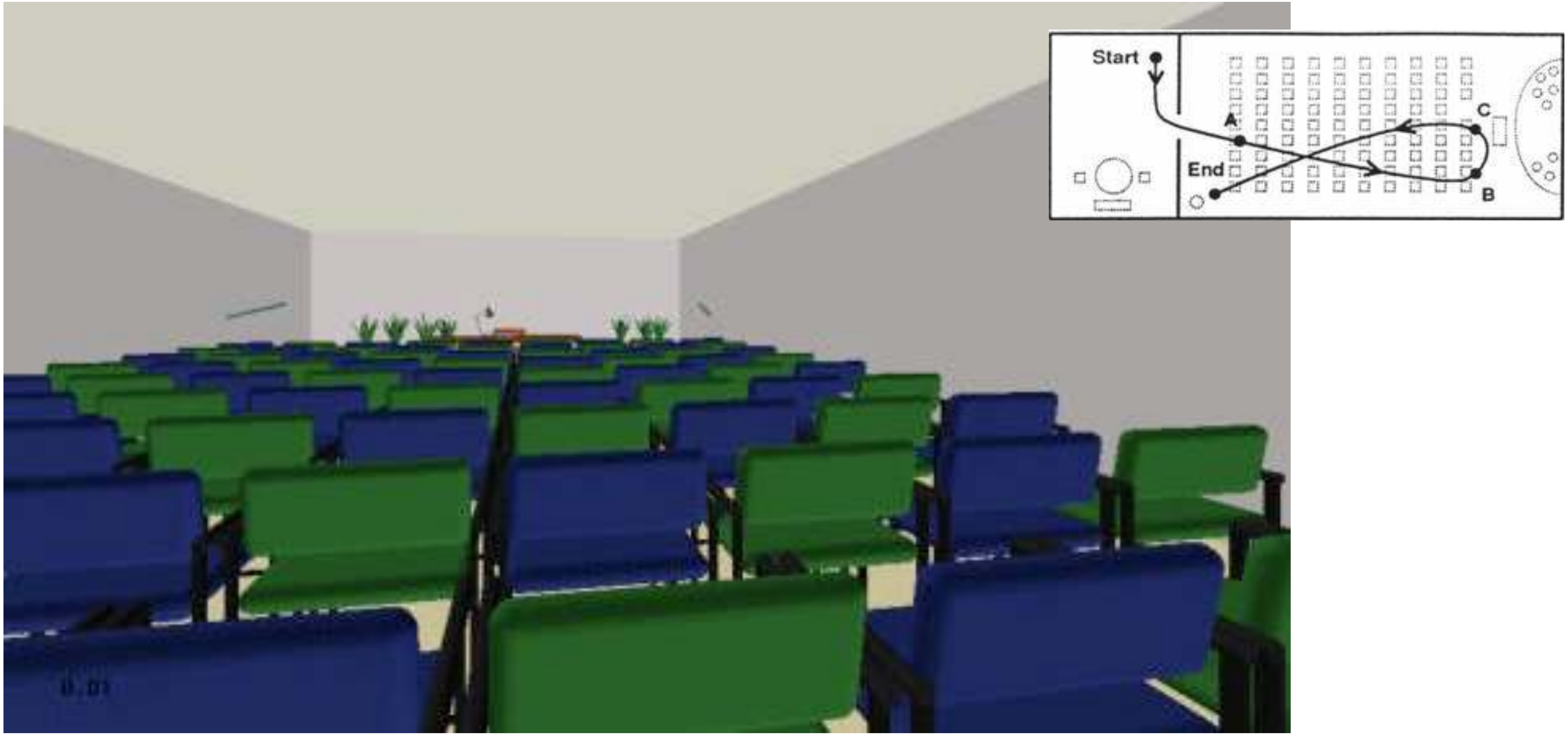
  - Depth of obj (relative to horopter):

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

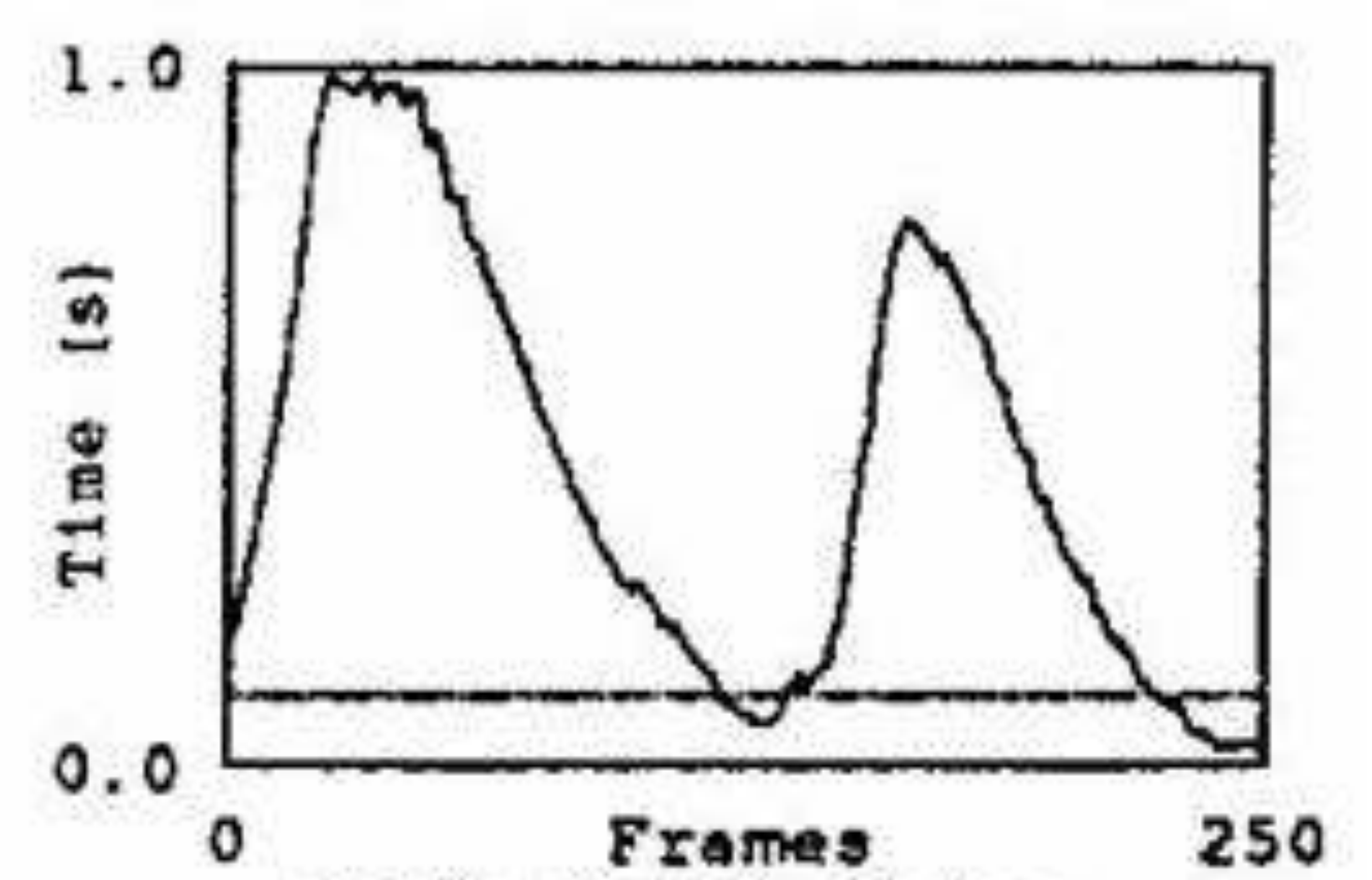


- Determination of LODs:
  1. $k = \min\{k_i\} \cdot k_0$, oder $k = \prod k_i \cdot k_0$
  2. $r_{\min} = 1/k$ (or similar transfer function)
  3. Select level l such that $\forall p \in P_l : r(p) \geq r_{\min}$, where P_l is the set of polygons of level l of an object, and $r(p)$ = radius of polygon p
- Do we need *eye tracking* for this to work?
  - Maybe …
  - Psychophysiology: eyes usually never deviate > 15° from head direction
  - So, assume eye direction = head direction, and choose b_1= 15°

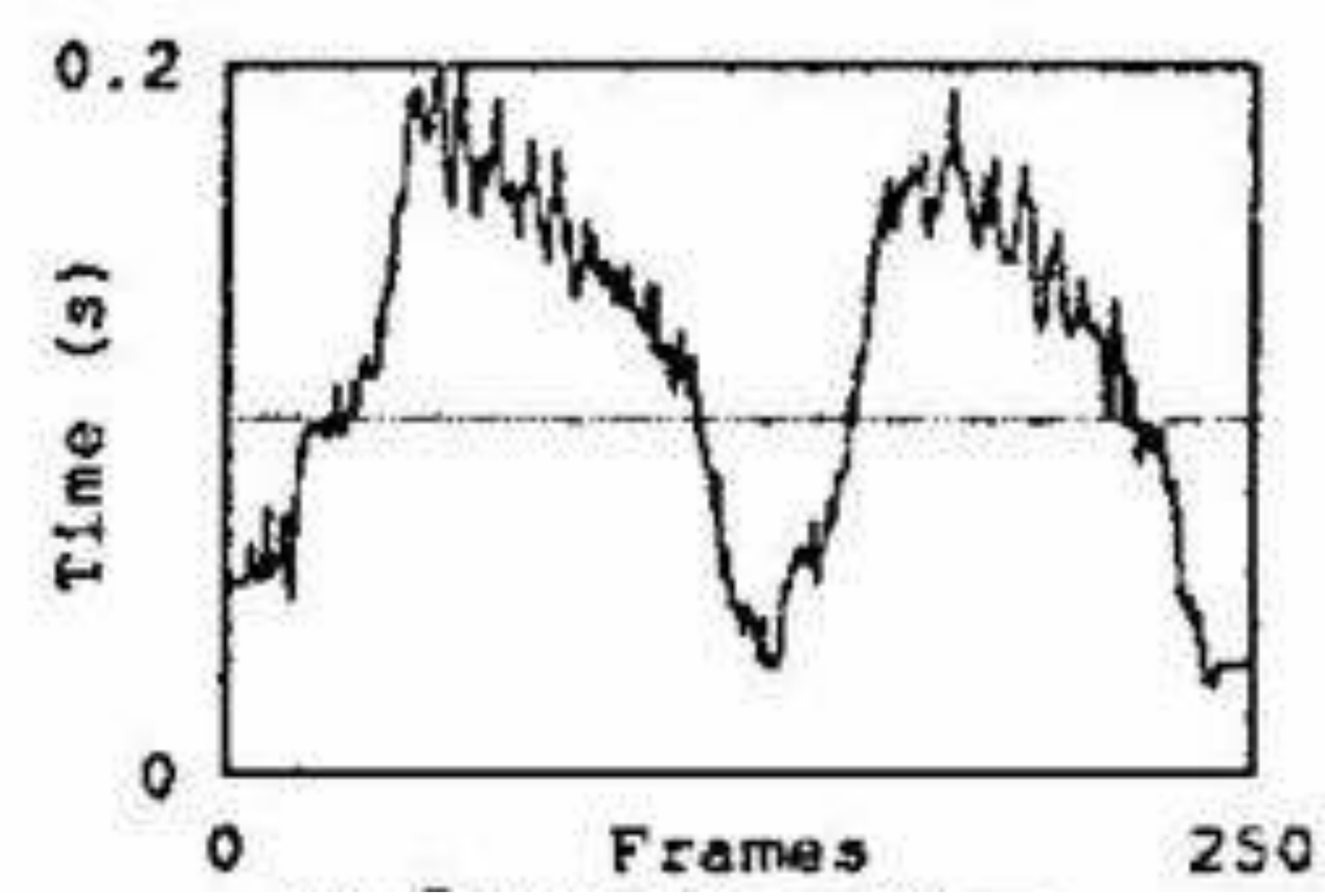
Example Scenario



Problems of Static LoD Selection



a) No LoD's



b) Static LoD selection

Reactive vs. Predictive LOD Selection

- Reactive LOD selection:
  - Keep history of rendering durations
  - *Based on the history*, estimate duration T_r for next frame,
  - Let T_b = time budget that can be spent for next frame
    - Usually constant, e.g., 11 msec for 90 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 actual rendering time in history
- Reactive LOD selection can produce severe outliers

Predictive LOD Selection

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

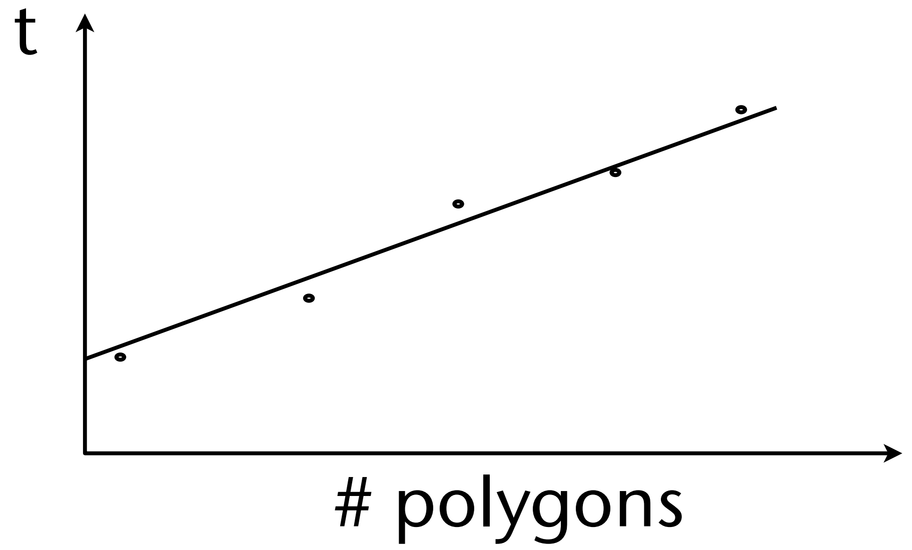
  under the condition $$T_r = \sum_{(O,L,R) \in S'} \text{cost}(O,L,R) \leq T_b$$

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

What Kind of Optimization Problem is This?

https://www.menti.com/smvndia2ss

- 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: $\text{Cost}(O, L, R) = \max \begin{Bmatrix} C_1 \cdot \text{Poly} + C_2 \cdot \text{Vert} \\ C_3 \cdot \text{Pixels} \end{Bmatrix}$
- 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: $\text{Rendering}(O,L,R) = \begin{cases} 1 - \frac{c}{\#\text{pgons}} & \text{, flat} \\ 1 - \frac{c}{\#\text{vert}} & \text{, Gouraud} \\ 1 - \frac{c}{\#\text{vert}} & \text{, per-pixel} \end{cases}$
  - 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: $\text{Hysteresis}(O,L,R) = \frac{c_1}{1+|L-L'|} + \frac{c_2}{1+|R-R'|}$
  - Together: $\text{Benefit}(O,L,R) = \text{Size}(O) \cdot \text{Rendering}(O,L,R) \cdot \text{Importance}(O) \cdot \text{OffCenter}(O) \cdot \text{Vel}(O) \cdot \text{Hysteresis}(O,L,R)$

- Optimization problem = multiple-choice knapsack problem → NP-complete
- Idea: compute sub-optimal solution
  - Reduce it to continuous knapsack problem (see algorithms class)
  - Define

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

  - Solve this greedily:
    - Sort all object tuples by value(O,L,R)
    - Choose the first k tuples until knapsack is full
  - Additional 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

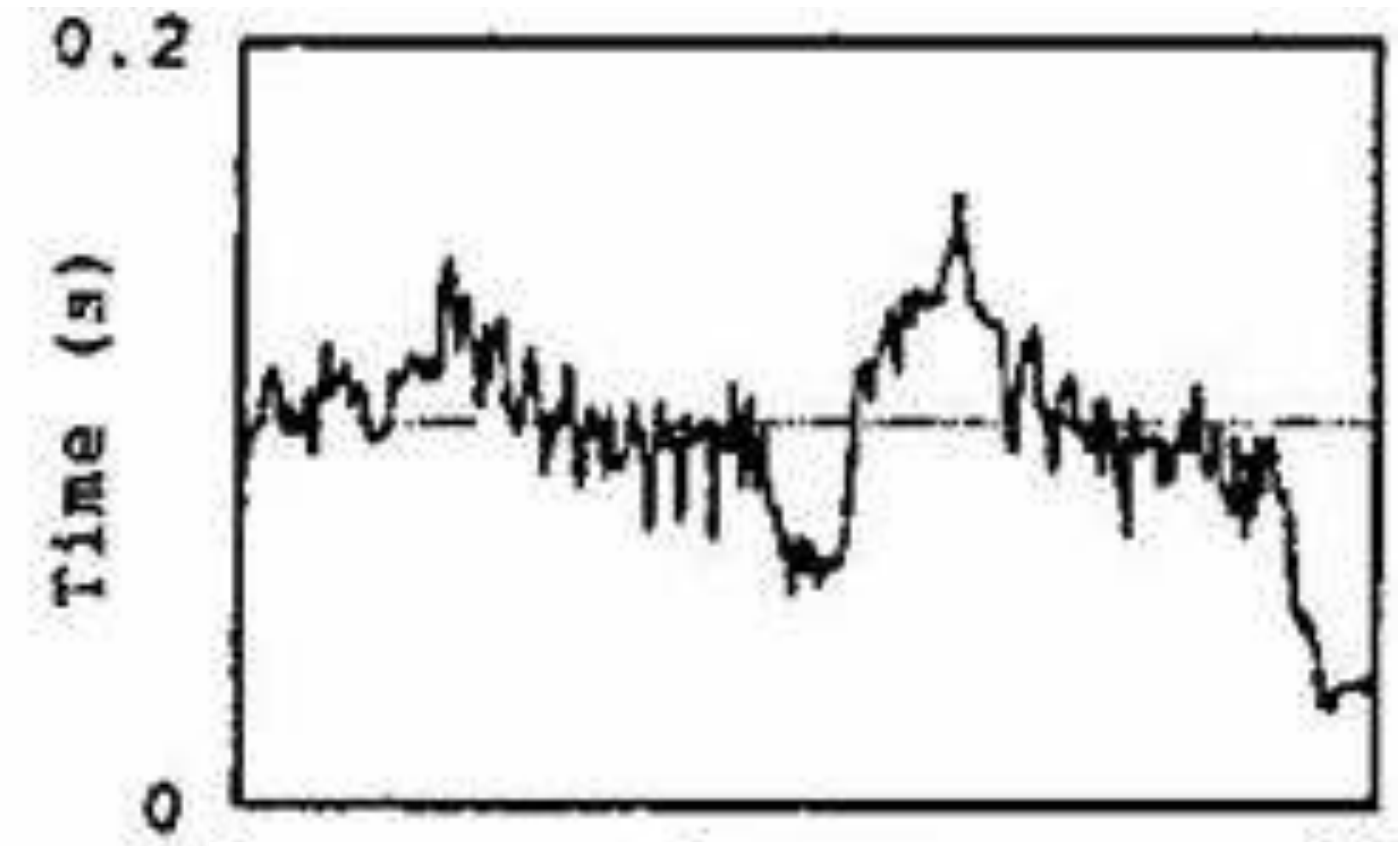
    $$\text{value}(O_k, L_k + a, R_k + b) - \text{value}(O_k, L_k, R_k) = \max$$

    and

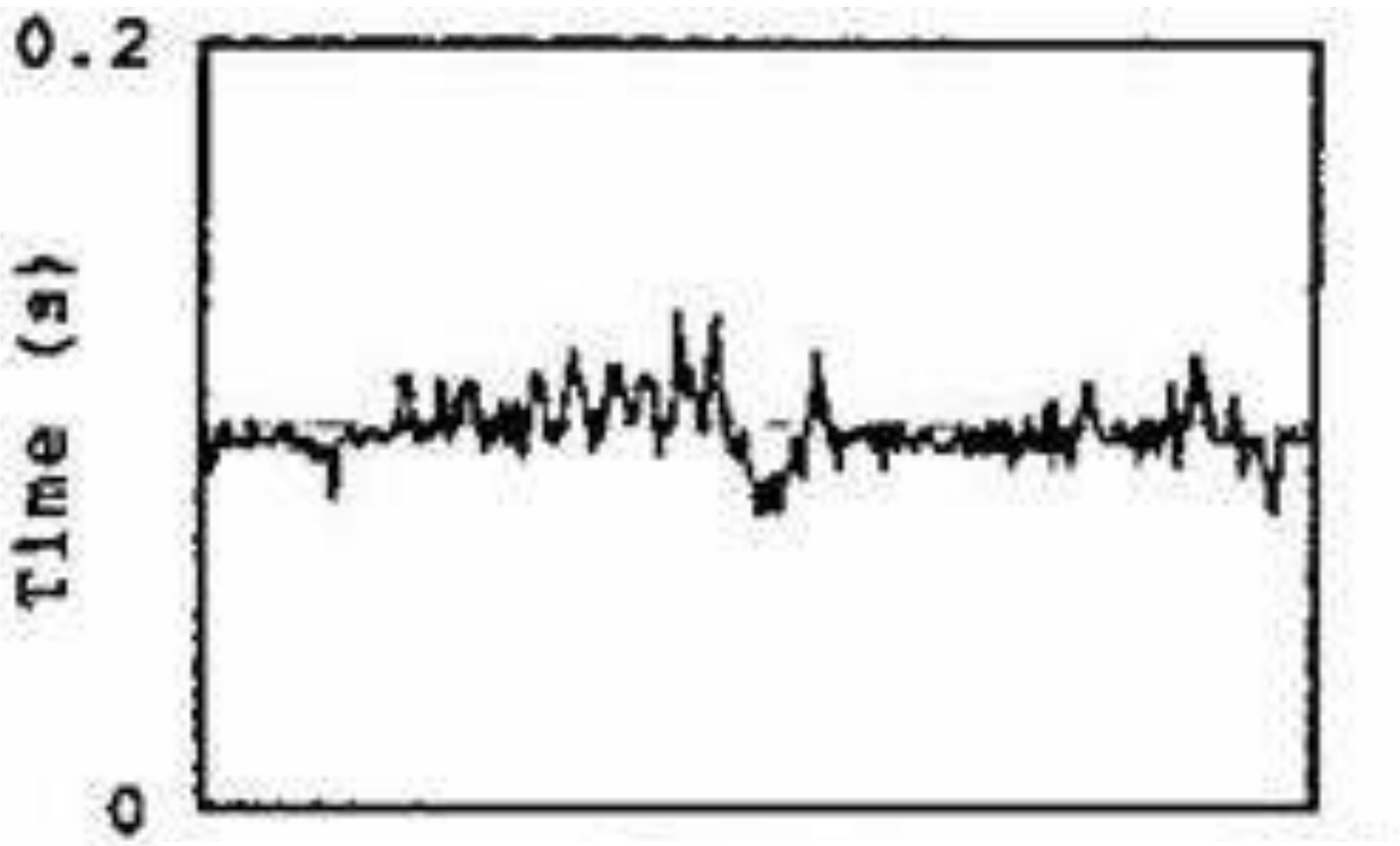
    $$\sum_{i \neq k} \text{cost}(O_i, L_i, R_i) + \text{cost}(O_k, L_k + a, R_k + b) \leq \text{max. frame time}$$

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

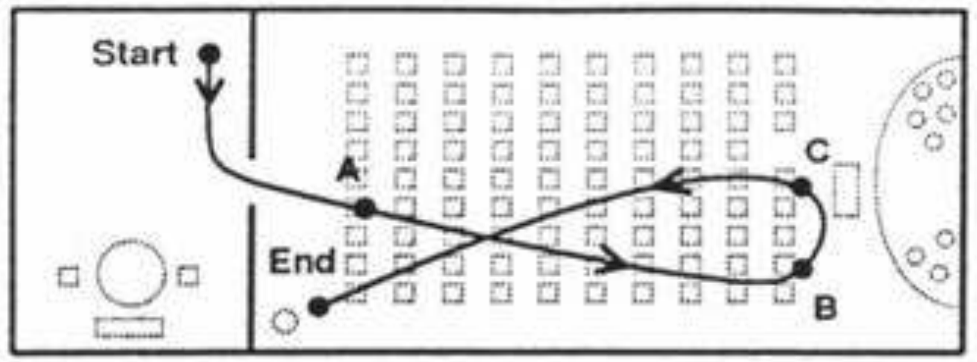
Performance in the Example Scenes



c) Reactive LoD selection



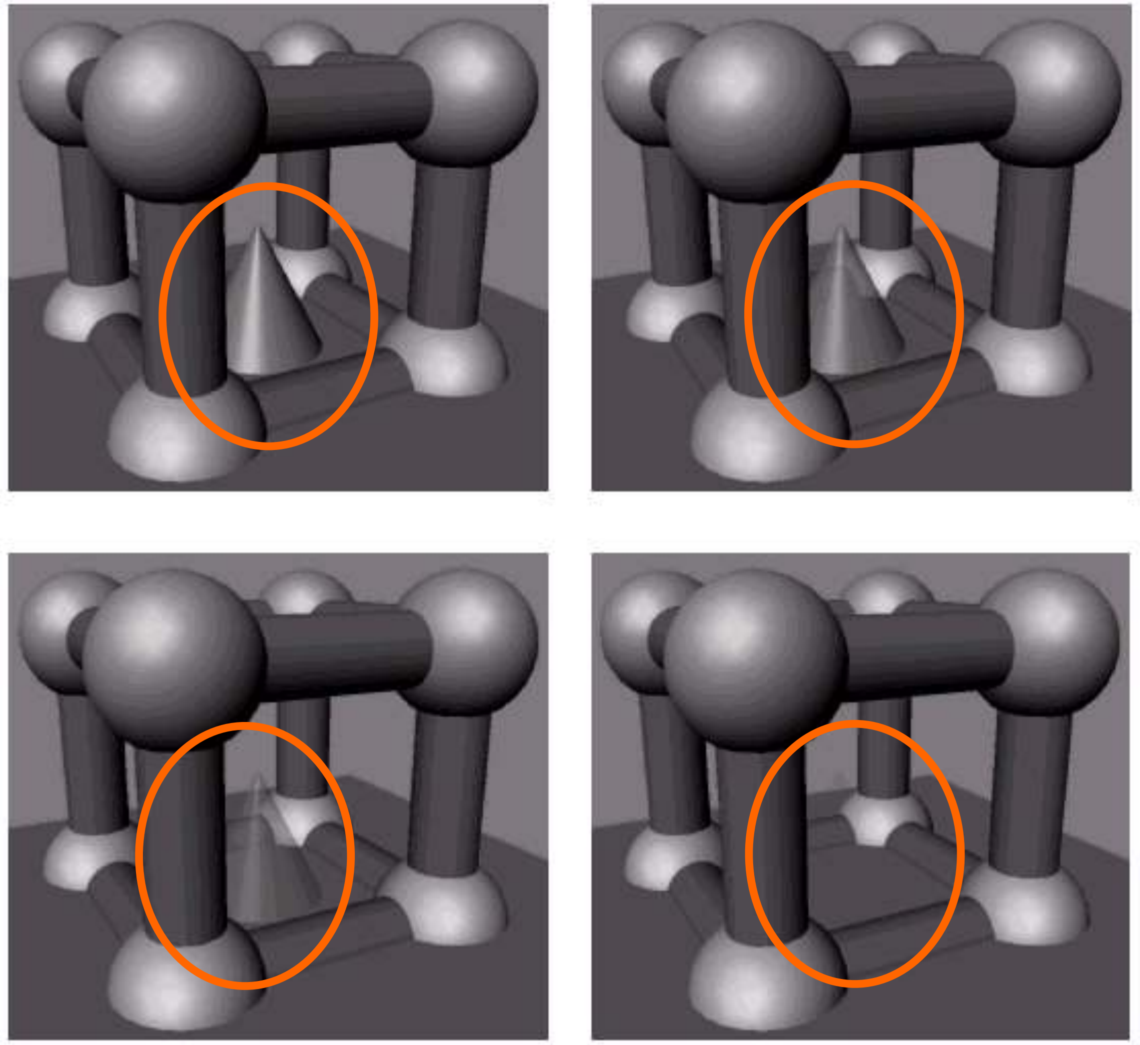
c) Predictive LoD selection



Problem with Discrete LODs

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

1. Simplest solution: temporal hysteresis (reduces frequency of pops, especially filters out short back-and-forth 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



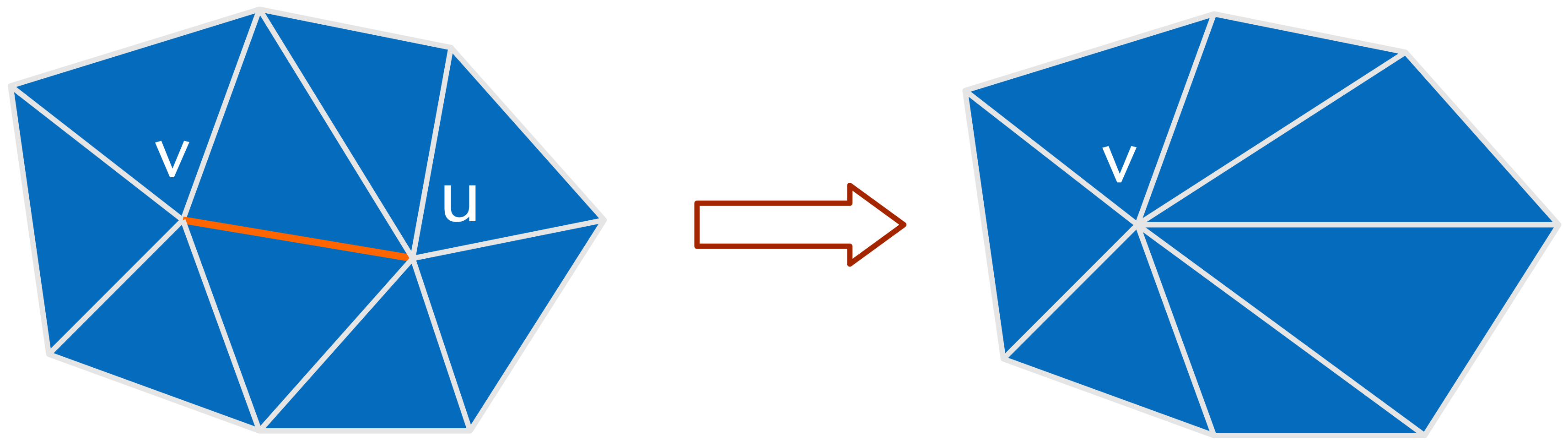
Progressive Meshes

- 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}

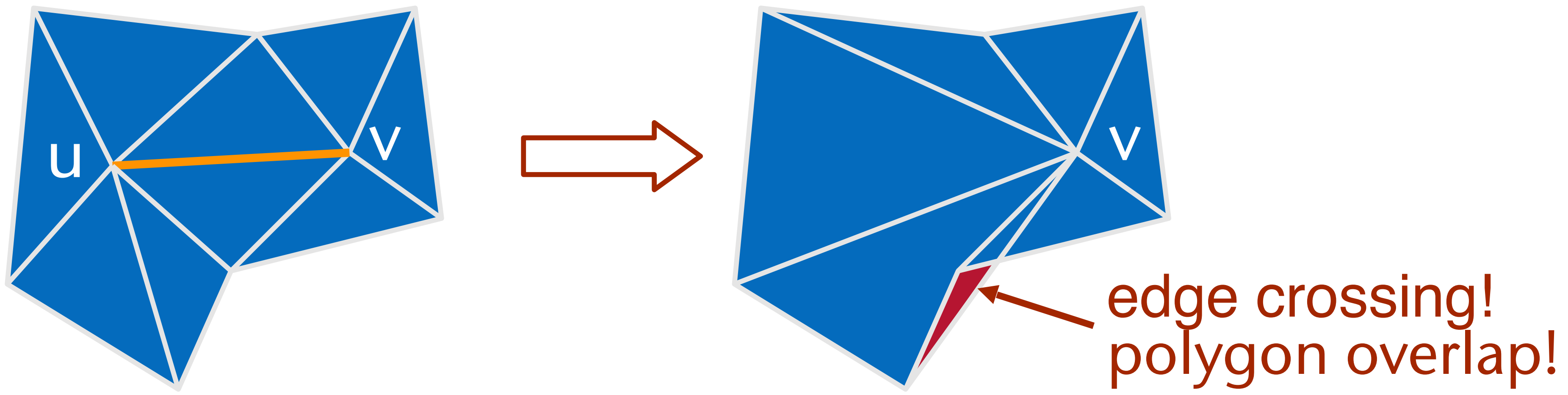
- Definition: progressive mesh = representation of an object, starting with a high-resolution mesh M_0, with which one can continuously (up to the vertex level) generate "in-between" meshes ranging from 1 polygon up to M_0 (and do that extremely fast).

Construction of Progressive Meshes

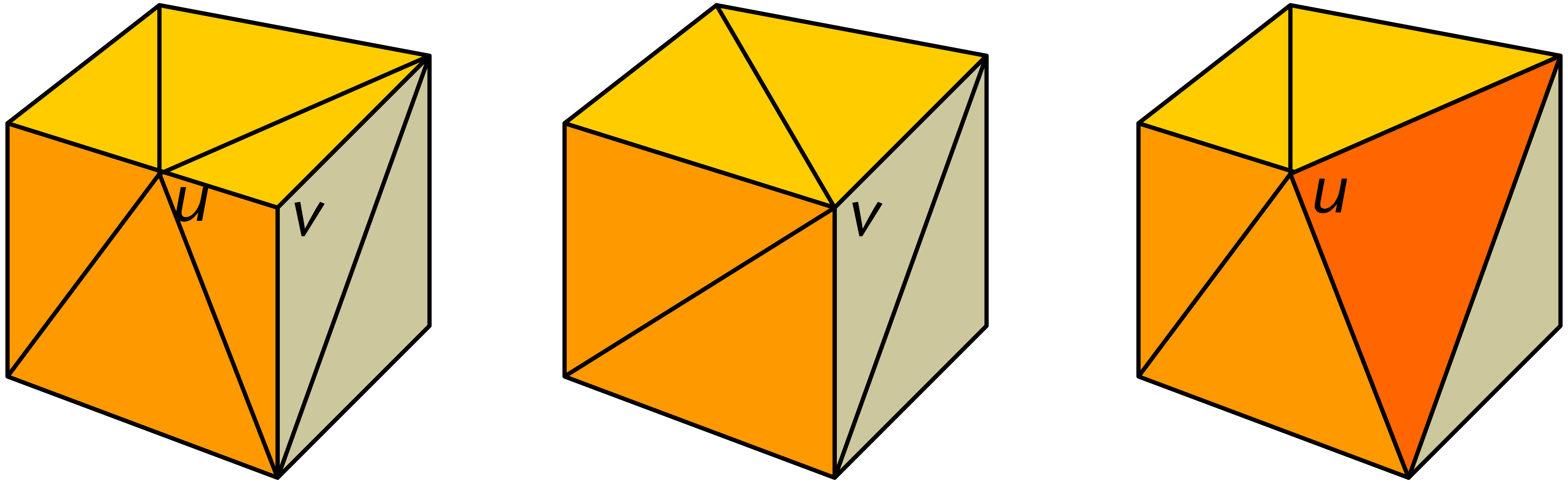
- Approach: successive *simplification*, until only 1 polygon left
- The fundamental operation: *edge collapse*



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



- The direction of edge collapses is important, too:



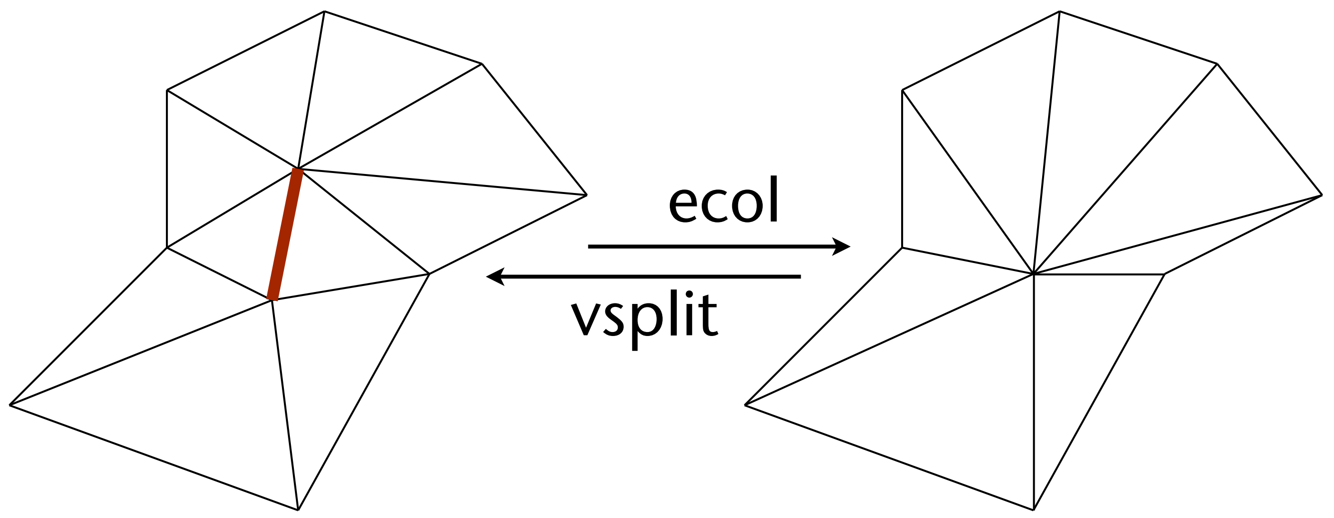
- Introduce measure of edge collapses that evaluates "visual effect"
- Goal: first perform edge collapses that have the least visual effect
- Remark: after every edge collapse, all remaining edges need to be evaluated again, because their "visual effect" (if collapsed) might be different now

- Progressive mesh = sequence of edge collapses / vertex splits:

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

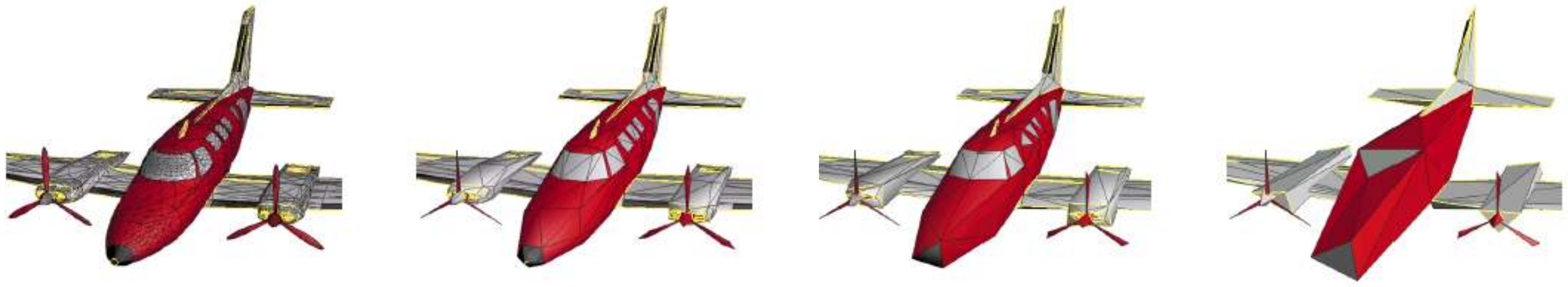
  - M^i = i-th refinement = 1 vertex more than M^{i-1}

- Representation of progressive mesh = list of ecol/vsplit operations



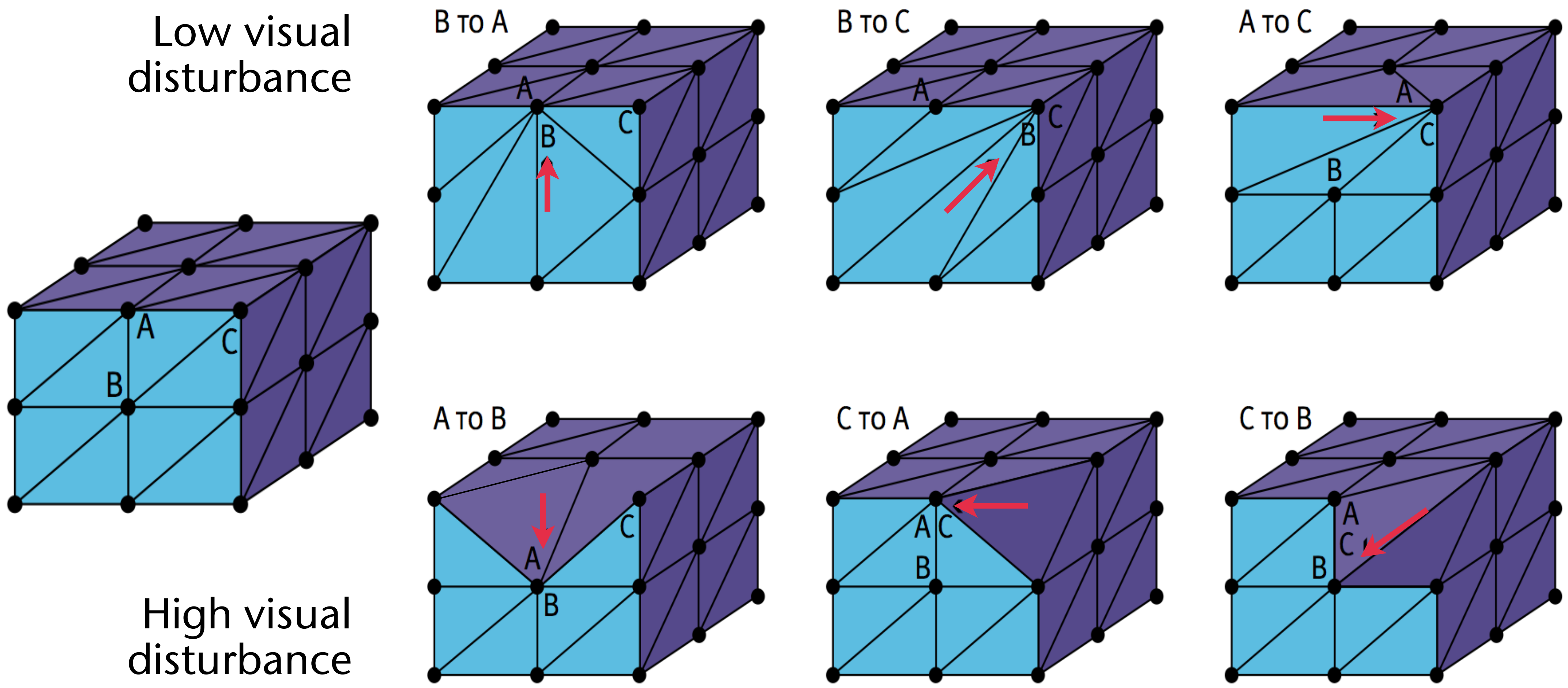
- Representation of an edge collapse / vertex split:
  - Edge (= pair of vertices) affected by the collapse/split
  - Position of the "new" vertex
  - Triangles that need to be deleted / inserted

- 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 (e.g., eyes & mouth are very important in faces)
- Examples of a progressive mesh:



A Simple Edge Evaluation Function

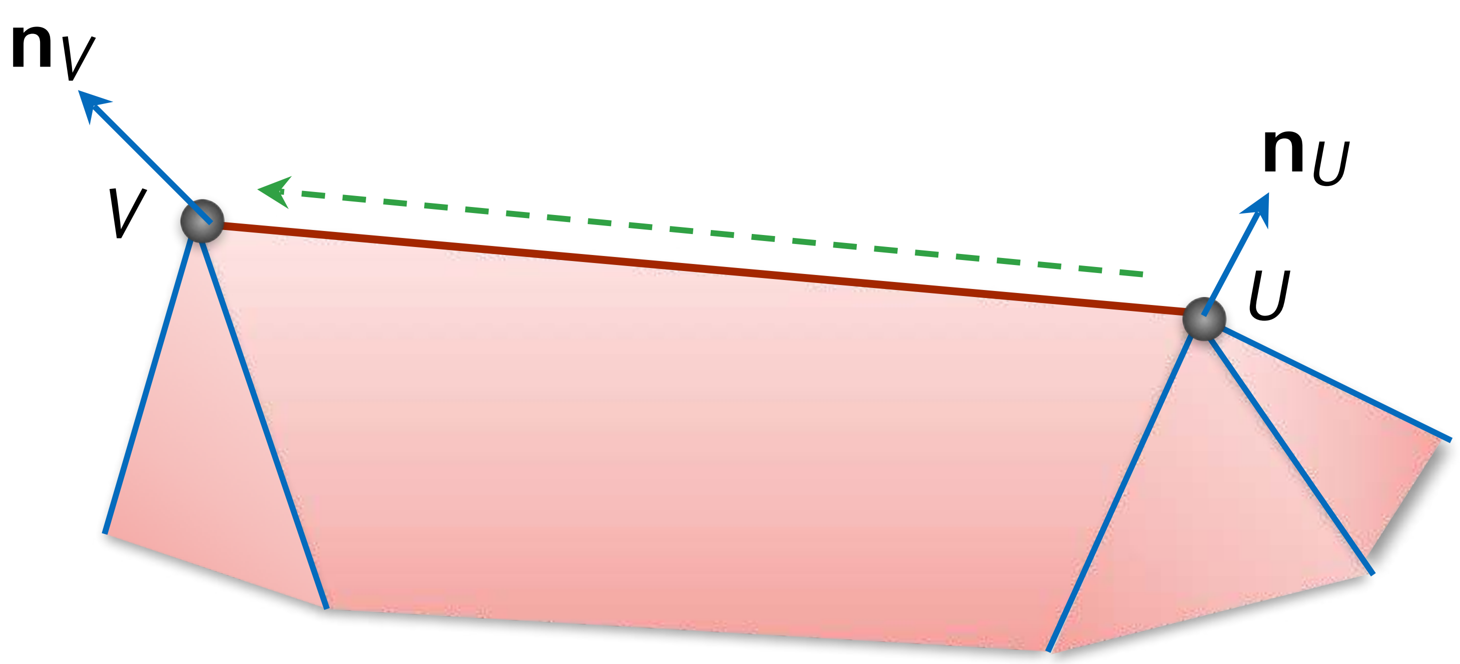
- Motivation:



- Follow this heuristic:
  - Delete small edges first; and,
  - If surface incident to *U* has a smaller (discrete) curvature than surface around *V*, then move vertex *U* onto vertex *V*

- A simple measure for the "costs" of an edge collapse from U onto V:

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



- What is the rationale for this cost function?
- Note: the cost function is *not* symmetric (which is good):

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

Simple Method to Calculate a Rough Estimate of the Discrete Curvature

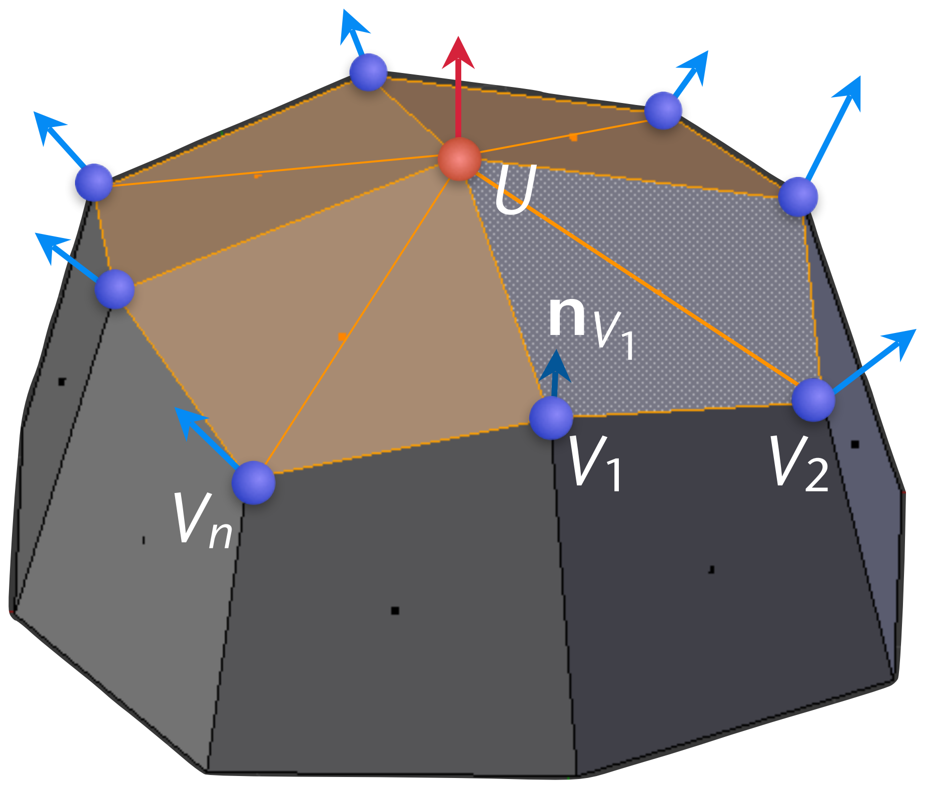
1. Calculate "curvature" *along* each edge $e_i = (U, V_i)$:

$$\text{curv}(e_i) = \frac{(\mathbf{n}_{V_i} - \mathbf{n}_U)\cdot(V_i - U)}{|V_i - U|^2}$$

2. Calculate estimate of "curvature" at U as geometric mean of incident edges:

$$\text{curv}(U) = \left(\prod_{i=1}^{n} \text{curv}(e_i)\right)^{\frac{1}{n}}$$

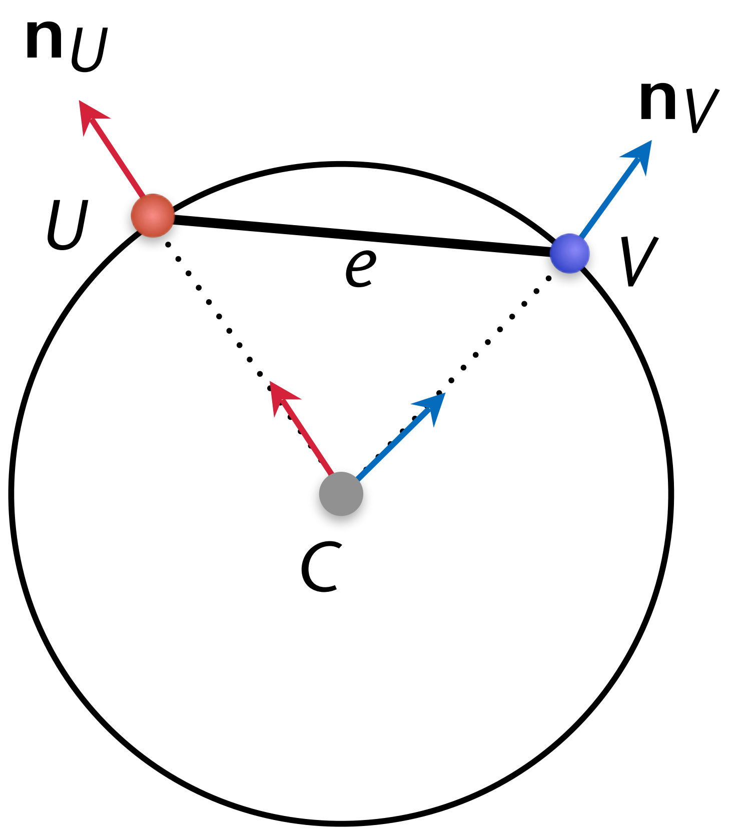
- Alternative to step 2:
  - Find the two edges e_1 and e_2 with minimal and maximal curvature, k_1 and k_2, resp.
  - Set $\text{curv}(U) = \frac{1}{2}(k_1 + k_2)$



Vertex normals must have unit length!

Reasoning Behind the Curvature Formula

- Consider a cross-section through U, one of the V's and the edge e=(U,V)
- Assume a circle through U, V with radius r and center C , and assume the normals are perpendicular to the circle; then



$$V = C + r\mathbf{n}_V \quad U = C + r\mathbf{n}_U$$

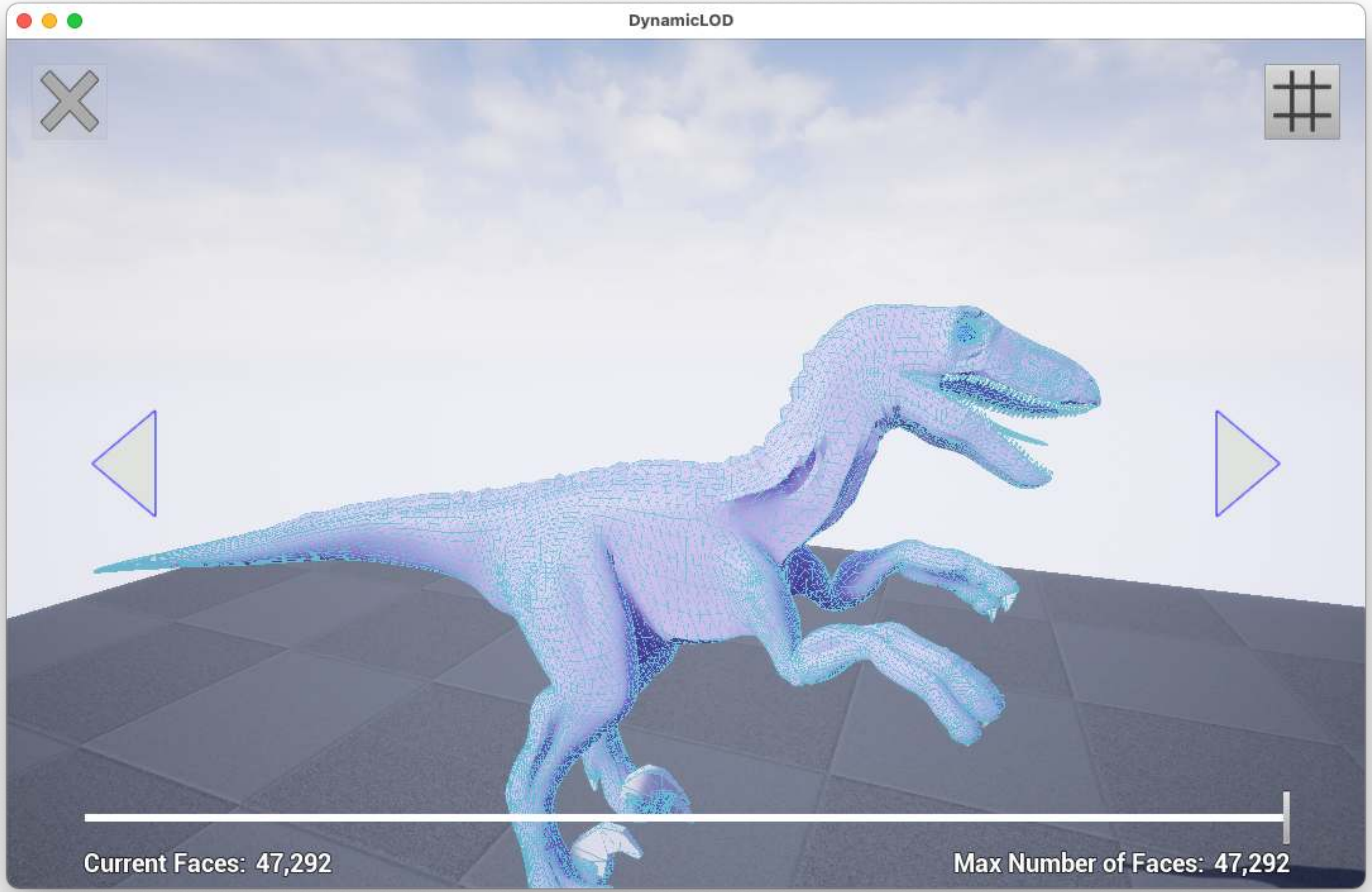
$$V - U = r(\mathbf{n}_V - \mathbf{n}_U)$$

$$\text{curv}(e) = \frac{1}{r} = \frac{\|\mathbf{n}_V - \mathbf{n}_U\|}{\|V - U\|}$$

- Make it more "robust" in 3D by first projecting $(\mathbf{n}_V - \mathbf{n}_U)$ onto the edge:

$$\text{curv} = \frac{(\mathbf{n}_V - \mathbf{n}_U)\cdot(V-U)^0}{\|V-U\|} = \frac{(\mathbf{n}_V - \mathbf{n}_U)\cdot\frac{V-U}{\|V-U\|}}{\|V-U\|} = \frac{(\mathbf{n}_V - \mathbf{n}_U)\cdot(V-U)}{\|V-U\|^2}$$

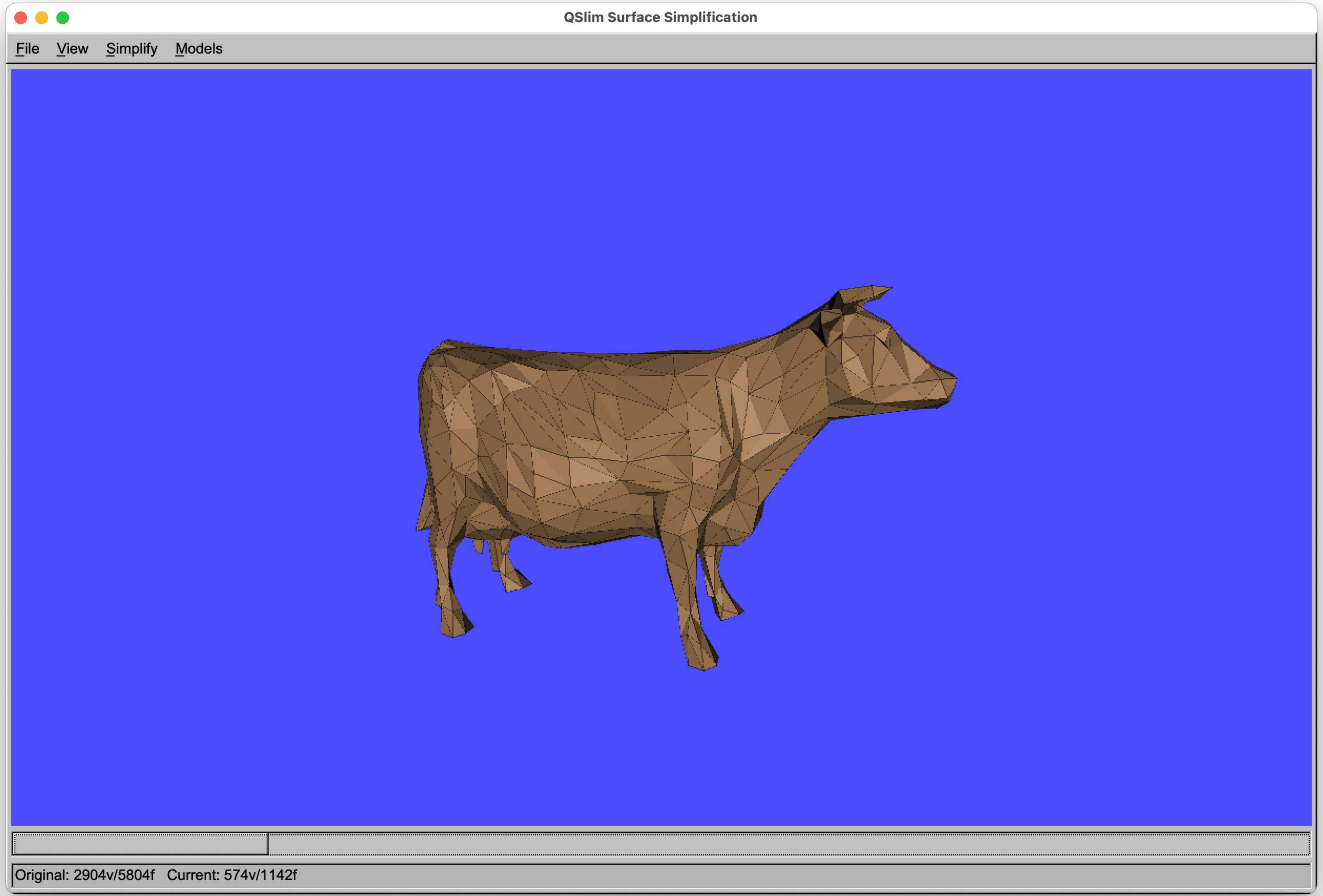
Demo



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

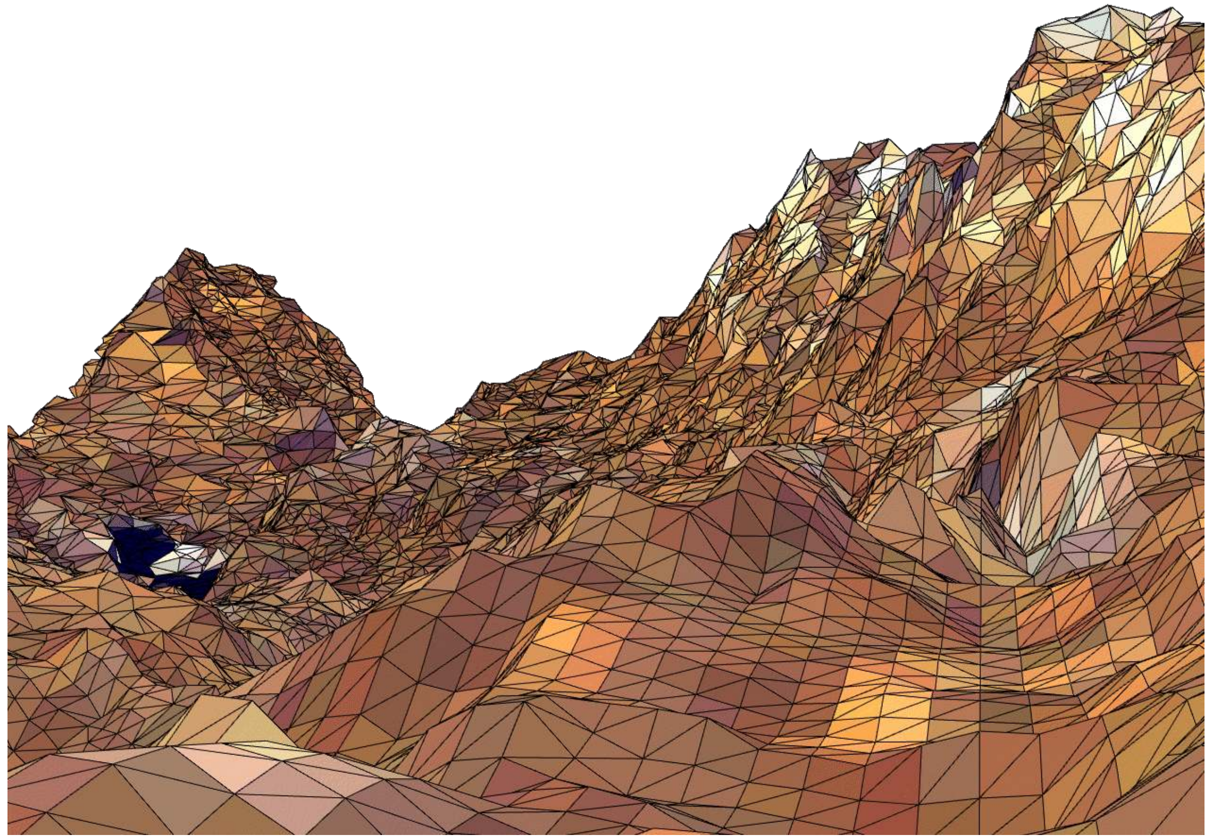
Master Thesis ...

Alternative Demo

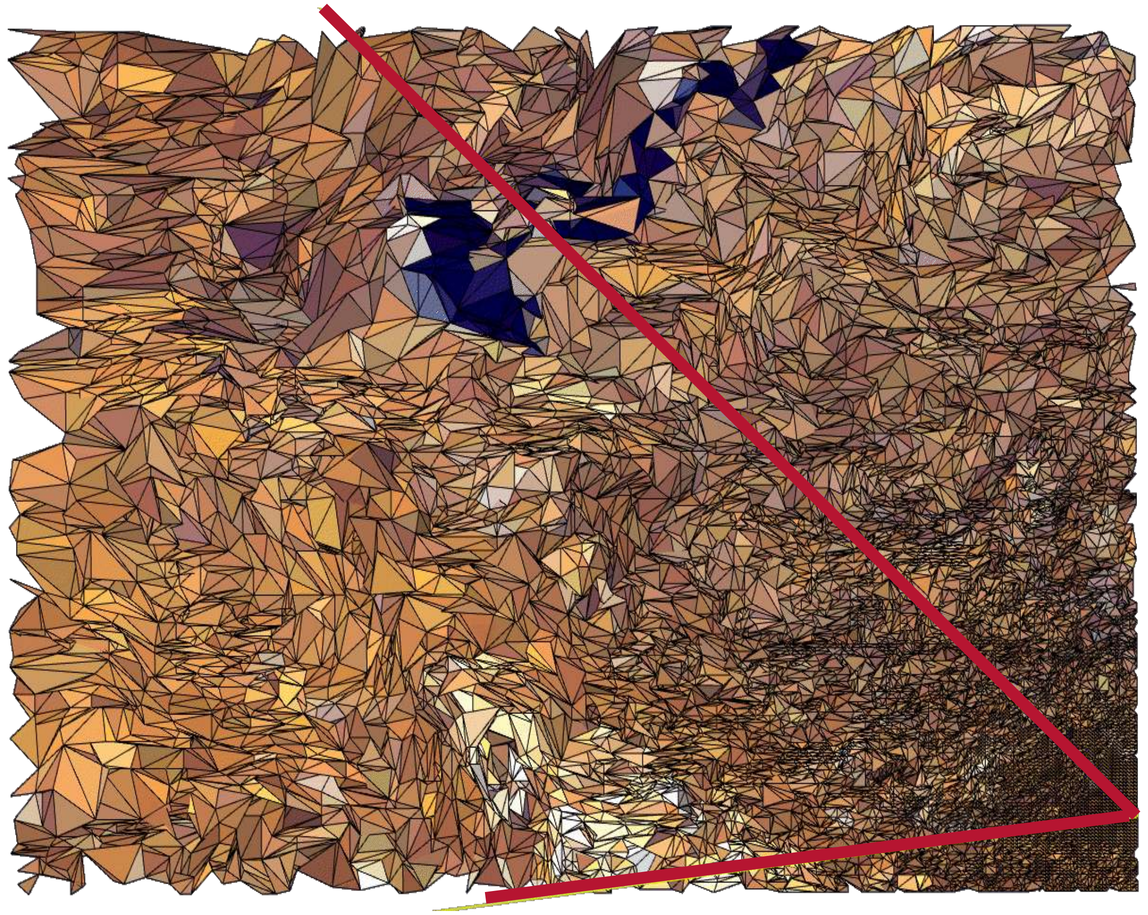


View-Dependent LOD's

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

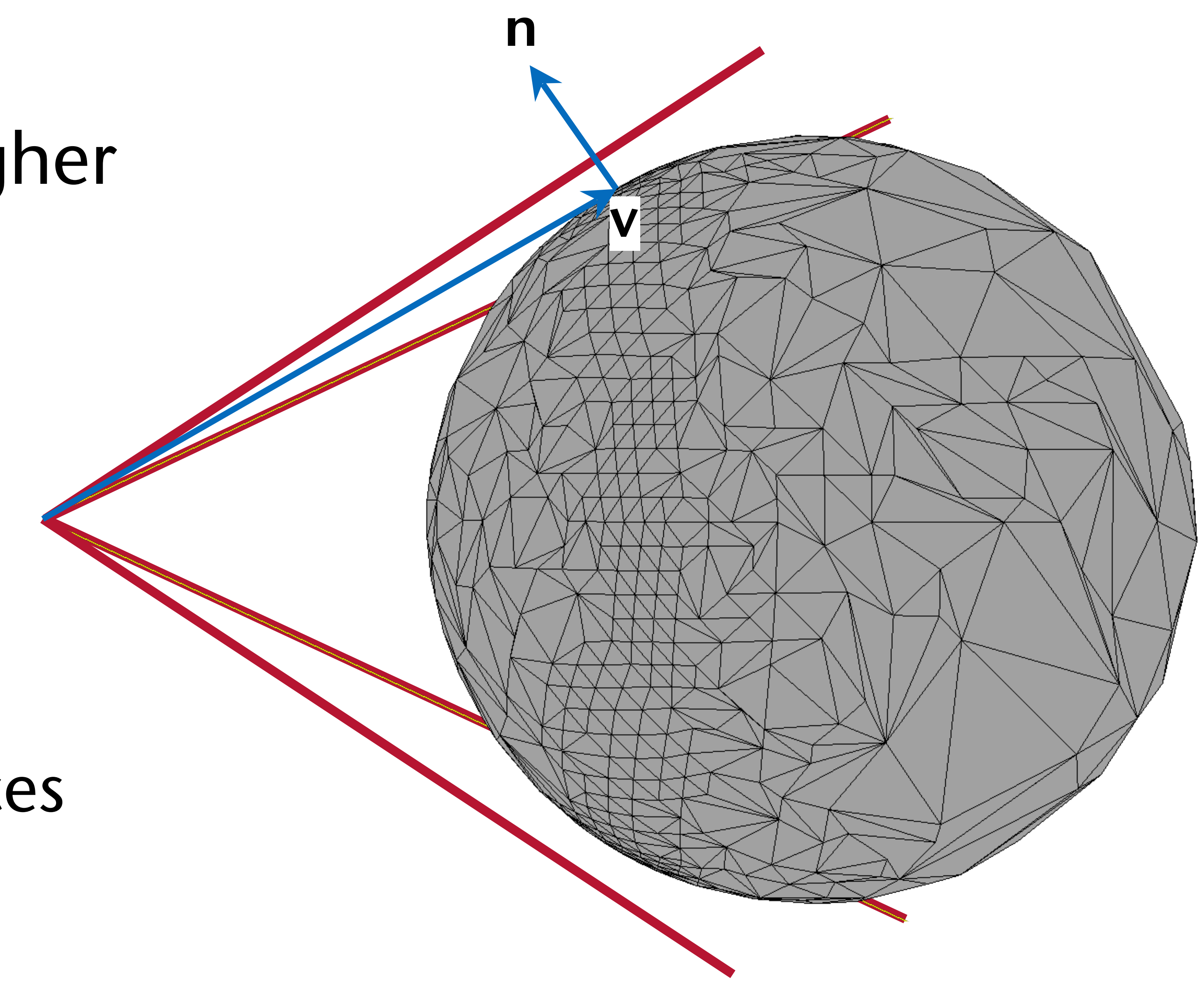


View from eye point



Birds-eye view

- Additional factor: visual importance
- Example: render closed objects with higher resolution near silhouette border
  - Maximal screen space error is modulated by (**v**·**n**)
- Other potential criteria:
  - Specular highlights
  - Salient features, e.g., feature points in faces
- Overall criteria:
  - Triangle budget
  - Time budget (remember *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
    - Simple for the programmer, too, i.e., easy to implement
    - No CPU overhead during rendering
  - Can upload LODs to GPU as vertex buffer objects (VBO)

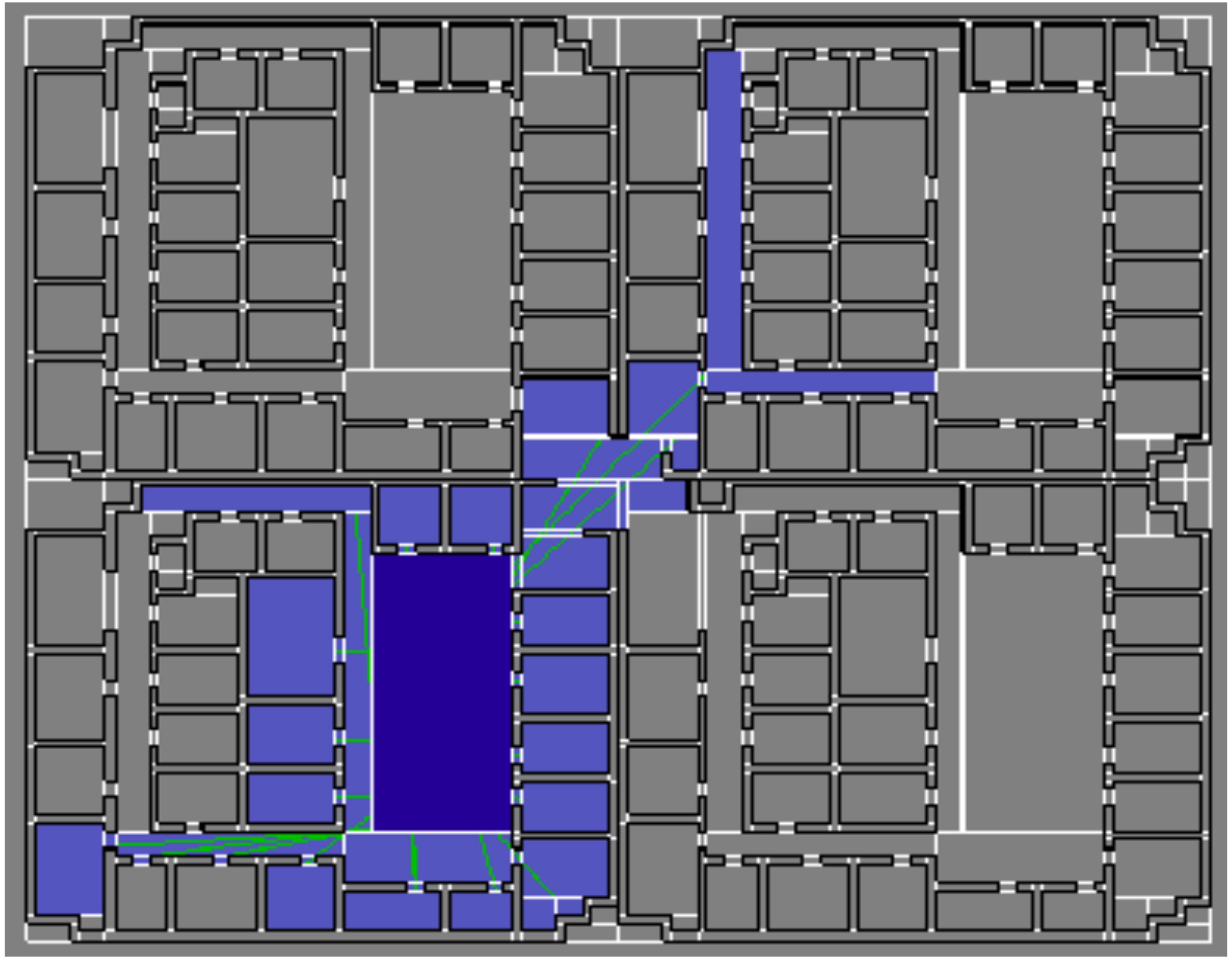
Master's Thesis topic: is it possible to implement progressive meshes (or other kind of dynamic LOD) in the GPU's vertex buffers?

Other Kinds of LODs

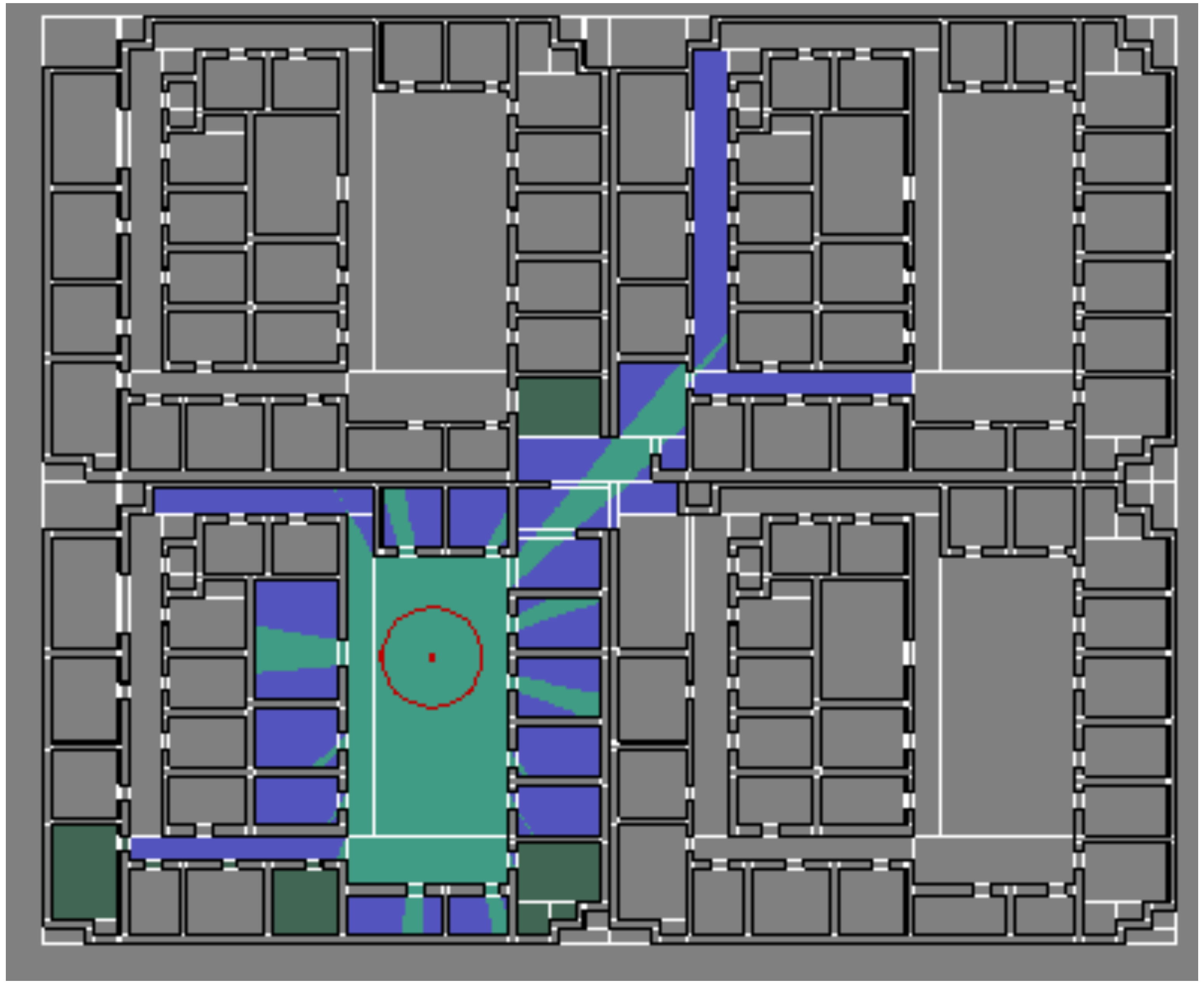
- Idea: apply LOD technique to other, non-geometric content
- E.g. "*behavioral LOD*":
  - If in focus, simulate the behavior of an object exactly, otherwise simulate it only "approximately"

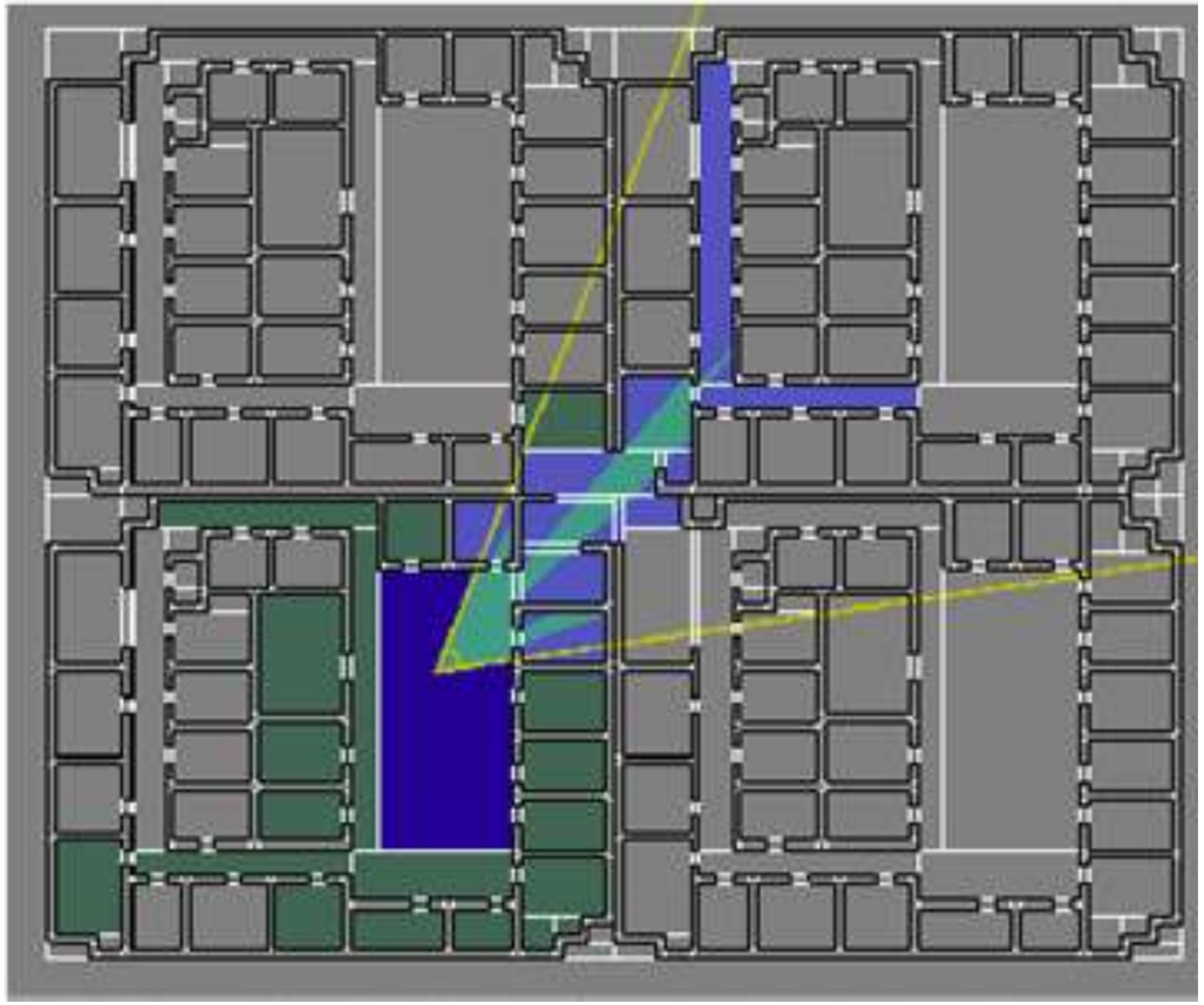
Portal Culling (Culling in Buildings)

- 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





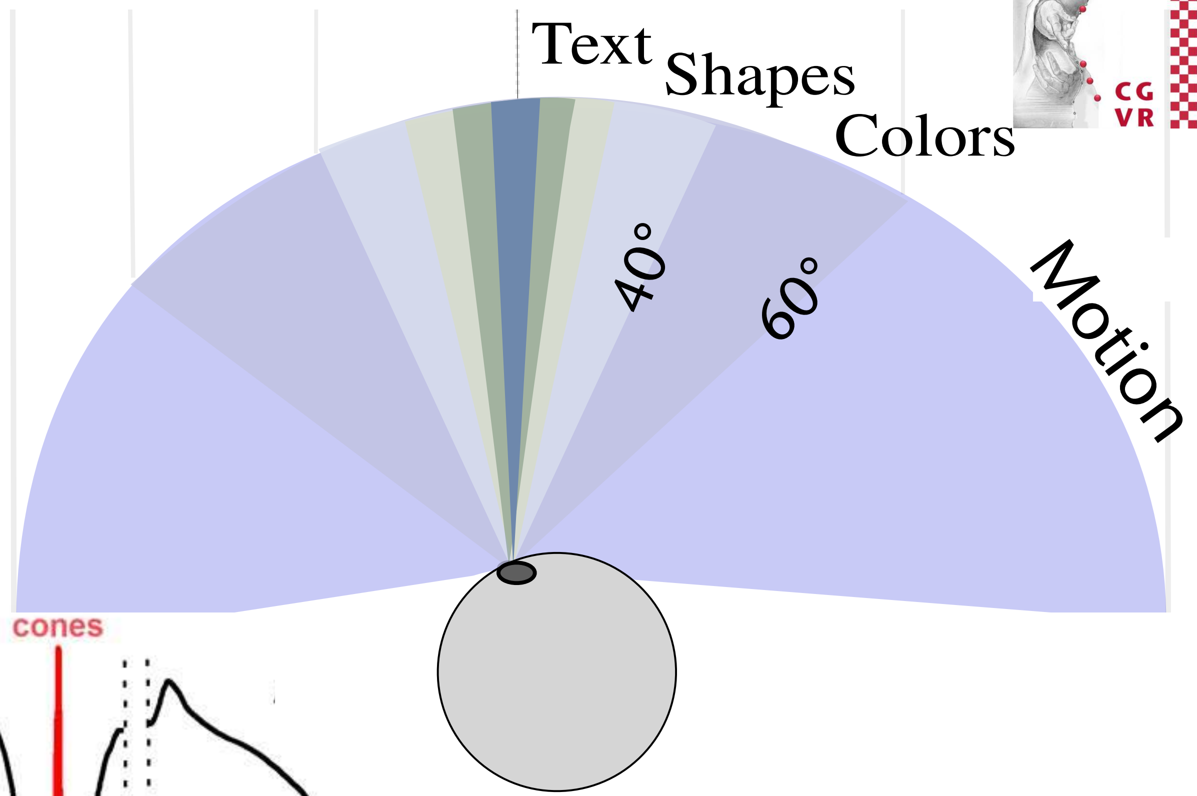
Test Your Knowledge of the Human Visual System

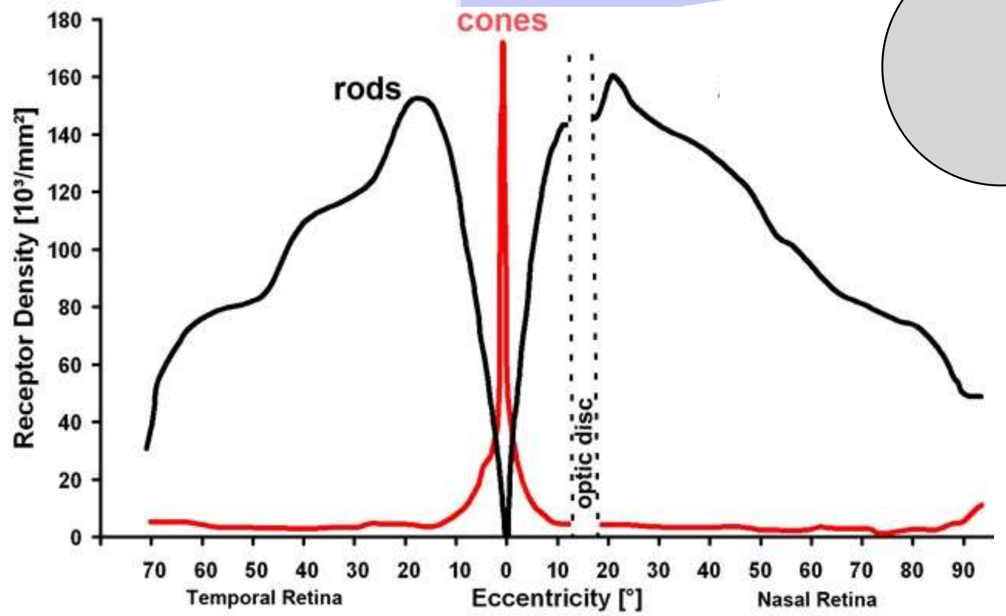
Please,
don't spoil by
"look-ahead"!

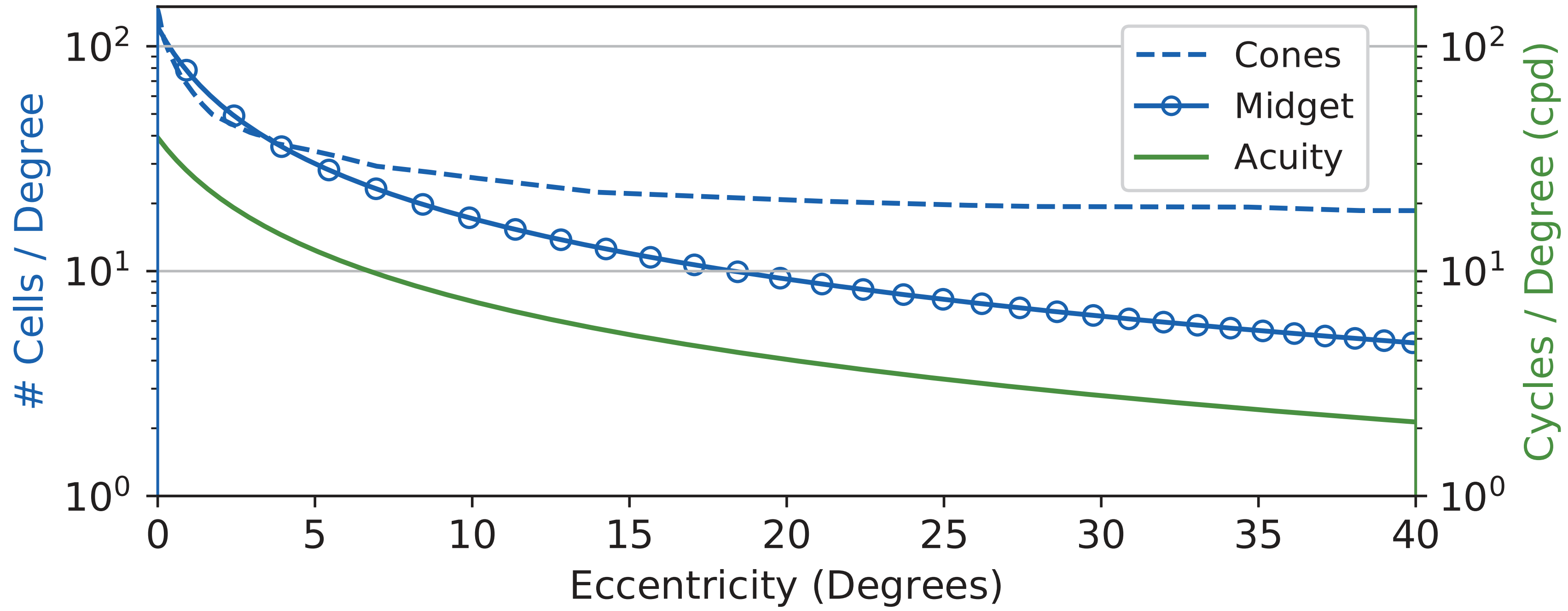
https://www.menti.com/smvndia2ss

Foveated Rendering

- Recap of some factors of our human visual system (HVS):
  - Critical flicker frequ. in periphery ≈ 85 Hz
  - Fovea = area of high visual acuity ≈ 5°
  - Resolution in fovea ≈ 1 arcmin !
  - At 20° eccentricity, spatial res. ≈ 7.5 arcmin
  - Midget (ganglion) cells collect and process cones' signals, then forward to brain ⟶ their density influences our visual acuity
  - Fovea covers ≈ 4% pixels of HMD
- Most pixels in HMD's are wasted!

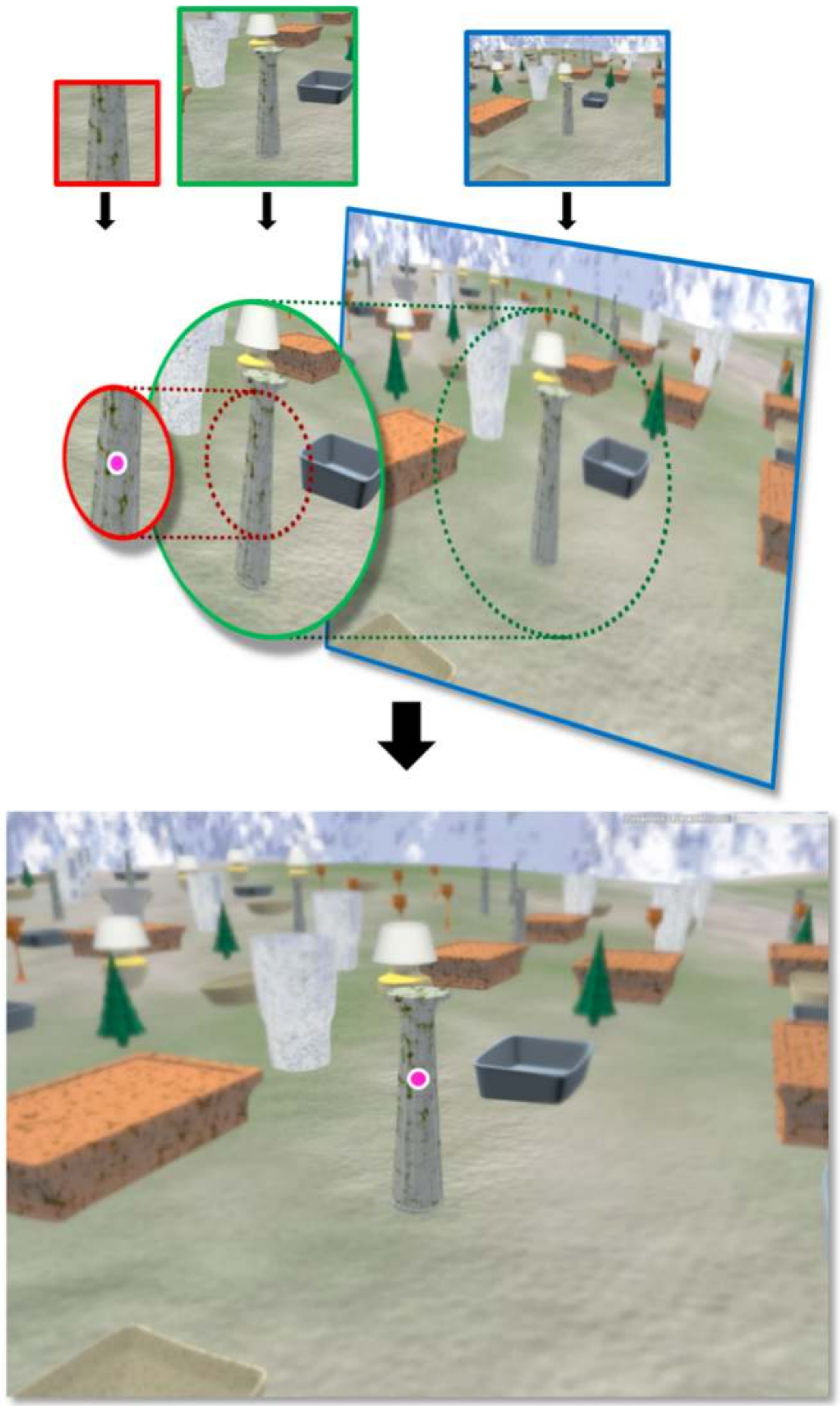






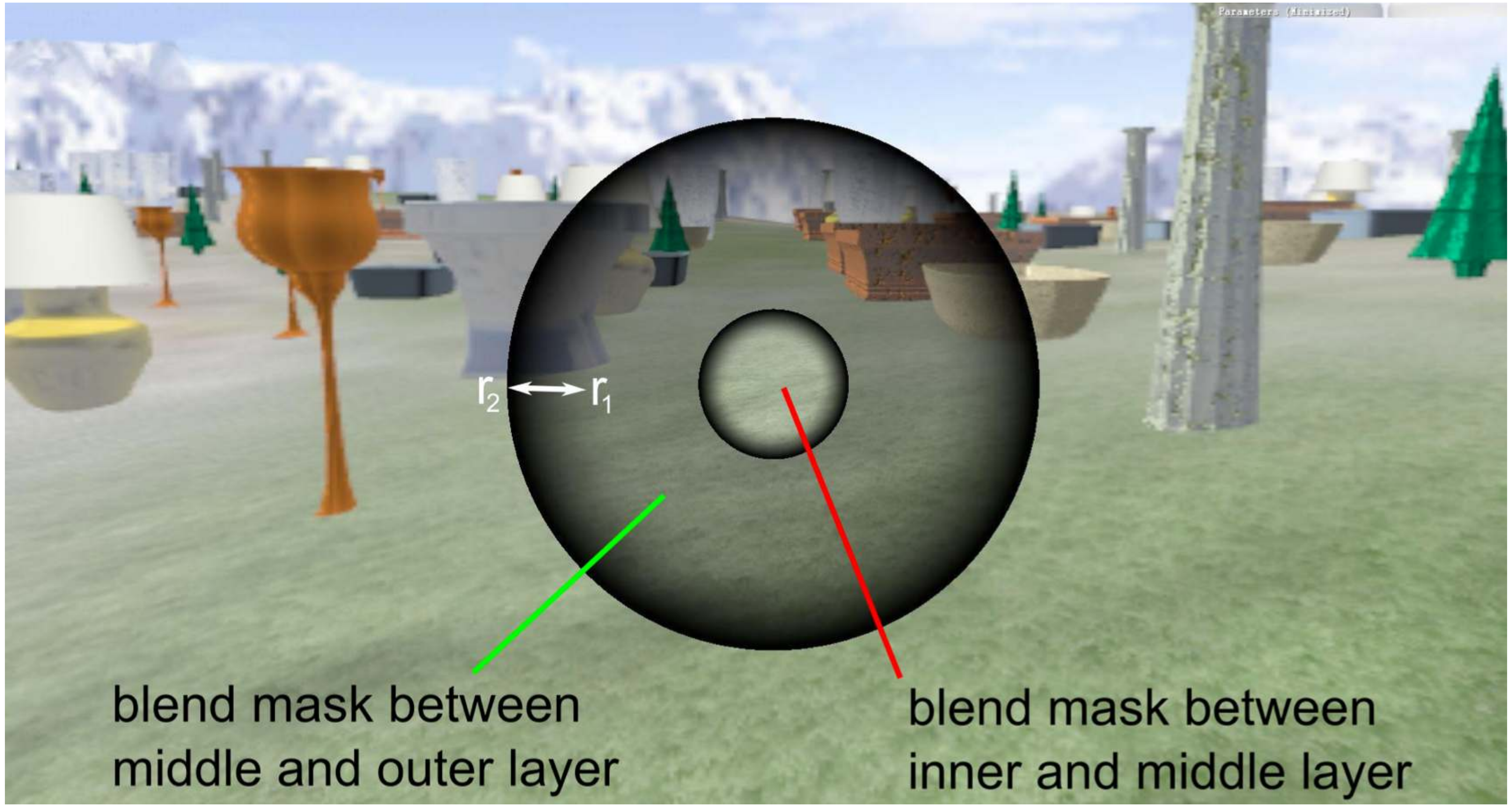
Foveated Rendering Technique

- Prerequisite: eye gaze tracking
- Goal: reduce image resolution towards periphery (*subsampling*)
- Approach:
  - Render 3 overlapping, nested "eccentricity layers" (render targets)
  - Each layer has its own image resolution (and LOD levels) ⟶ different sampling spacing!
  - Interpolate outer layers to final display resolution, then blend together
  - Optionally: update outer layers with lower frame rate



Blending the Layers

- Overlay on top of each other
- Calculate blend weights, depending on radius of pixel from center (i.e., gaze direction)
- Visualization of blending weights:

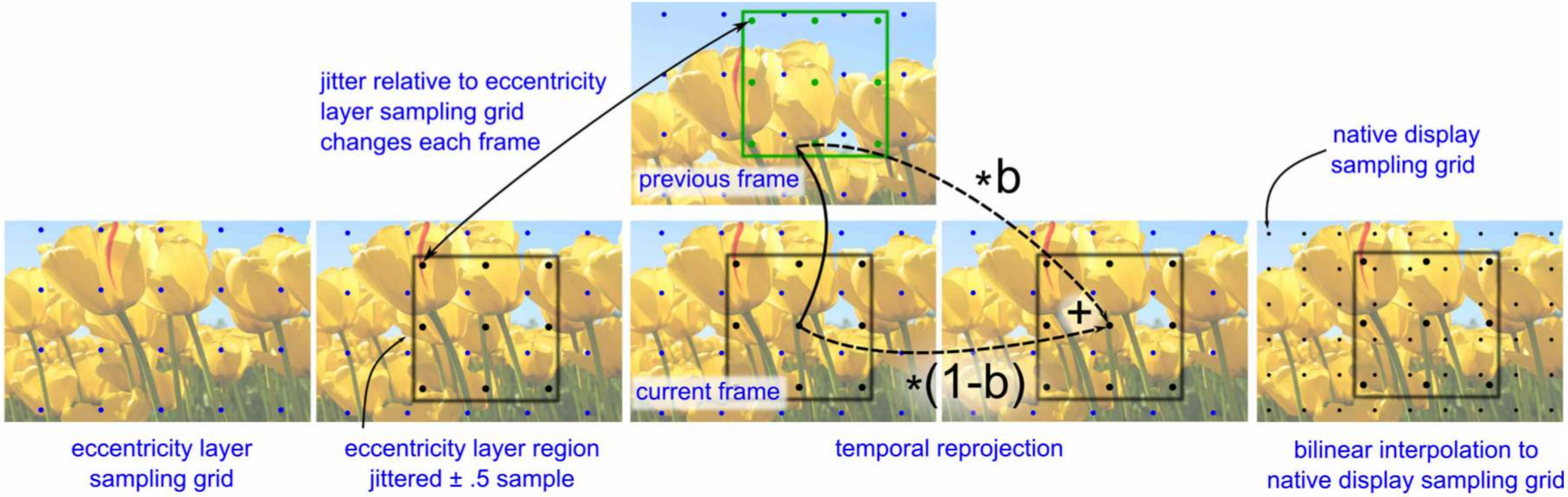


Challenges

- Latency: time elapsed between capturing the eye gaze direction and displaying the corresponding foveated image
- Experience shows:
  - 60 Hz monitor, 50 Hz eye tracker, 35 ms latency $\longrightarrow$ obvious "pop" in image resolution
  - 120 Hz monitor, 300 Hz eye tracker, 10 ms latency $\longrightarrow$ no visible "pop"
- Aliasing:
  - Outer layers have wide "pixel" stride $\longrightarrow$ aggravates aliasing artifacts
  - Periphery is very sensitive to temporal changes $\longrightarrow$ moving aliasing artifacts are extremely distracting / annoying

Anti-Aliasing Methods

- MSAA (Multi-Sample Anti-Aliasing): standard in GPU's, sample each pixel multiple times (e.g., by grid, or other pattern, within each pixel)
- Whole frame jitter sampling plus temporal reprojection:



Blending and Anti-Aliasing at Work

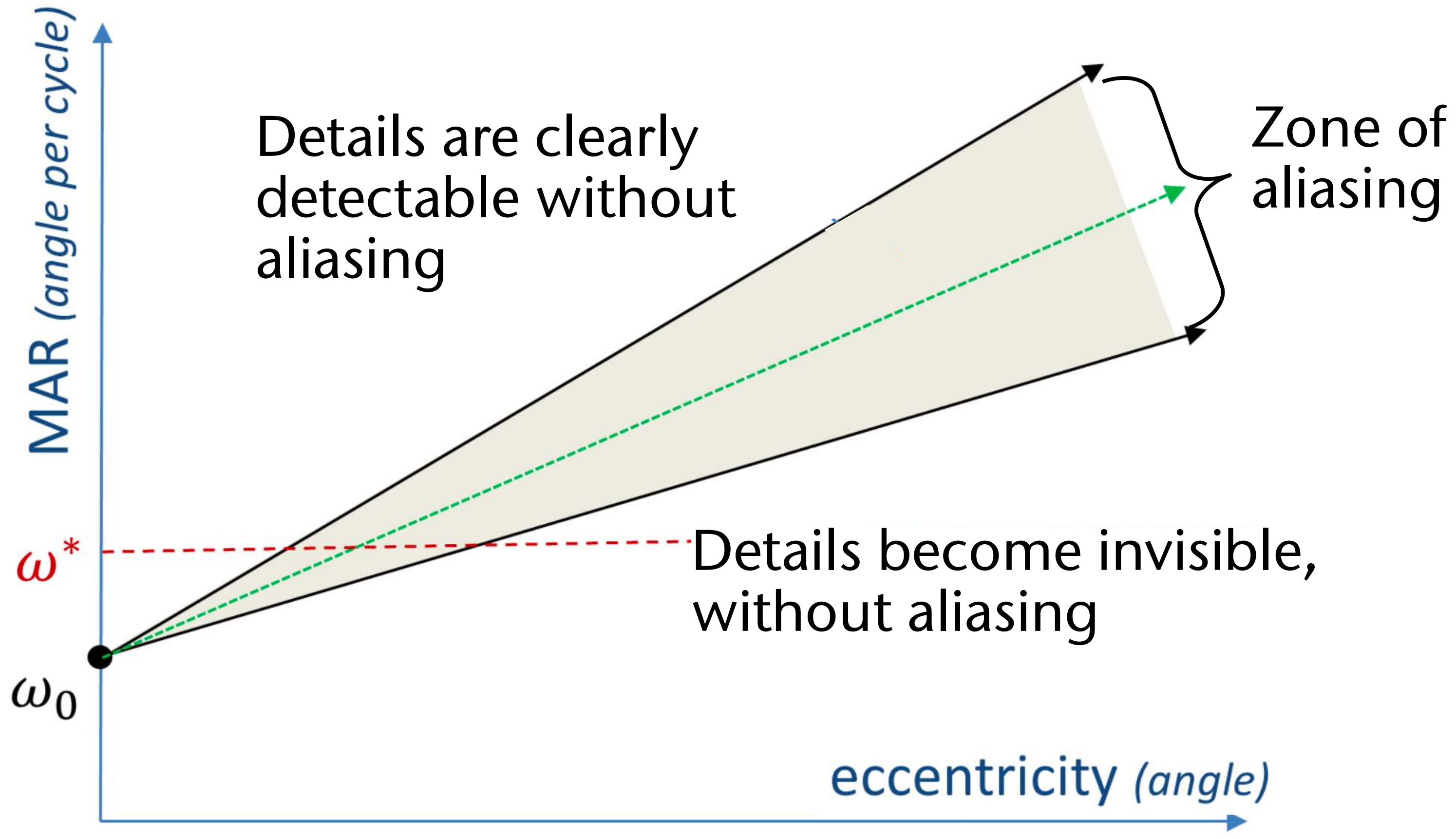
Smooth Composition

More on the Human Visual System

- Definition:
  - Imagine a grating of black and white lines next to each other
  - Minimum angle of resolution (MAR) ω = smallest angle of a cycle of white-black lines still visible
  - Visual acuity $= \dfrac{1}{\text{minimum angle of resolution}}$
  - Units:
    - MAR = degrees (°) = degrees per cycle
    - Acuity = frequency (Hz) = cycles per degree
- Standard model for MAR:

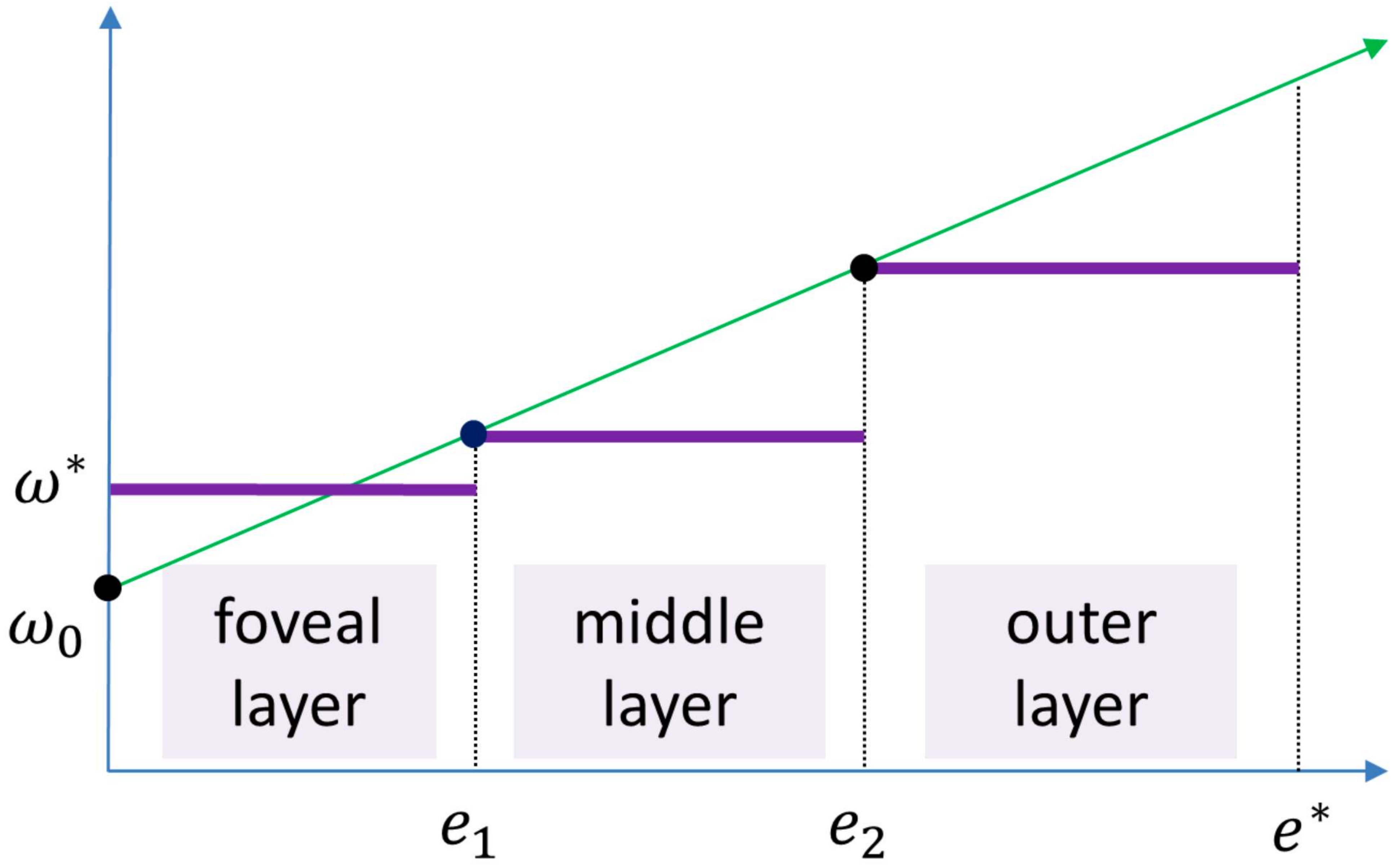
$$\omega = me + \omega^0$$

with e = eccentricity, ω^0 = MAR at fovea



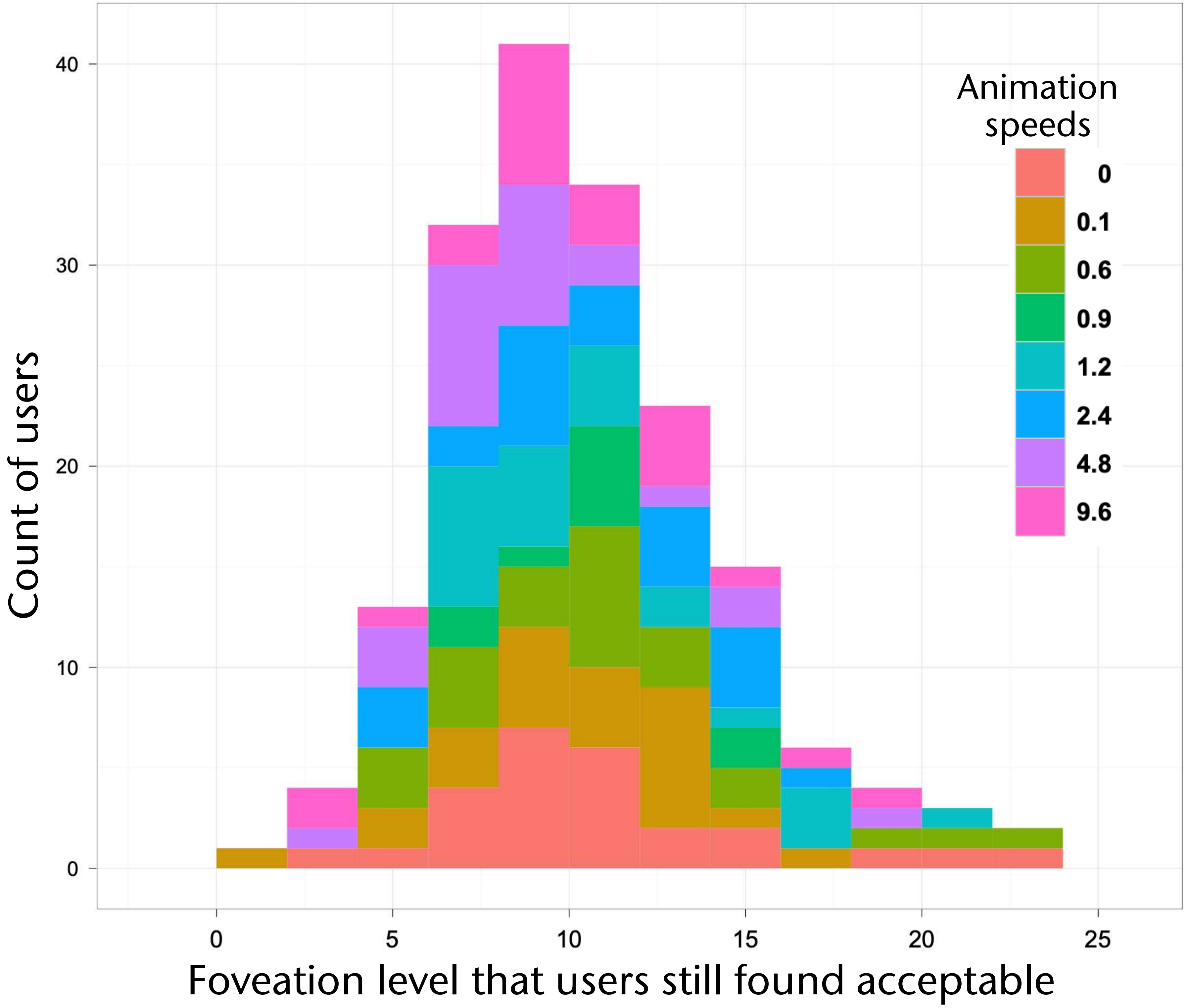
Connection Between Model and Rendering Speed

- Task: given a specific slope in the MAR model, *m*, and the number of eccentricity layers, choose the radii of the layers
  - Radii e_1, e_2 determine the total number of pixels to be rendered
- Determine by optimization
  - E.g.: brute force, choose e_1, e_2, with $0 < e_1 < e_2 < e^*$, then count the number of pixels
- Question: what is the best parameter *m*?
  - Smaller *m* ⟶ larger radii, more pixels to be rendered, less savings



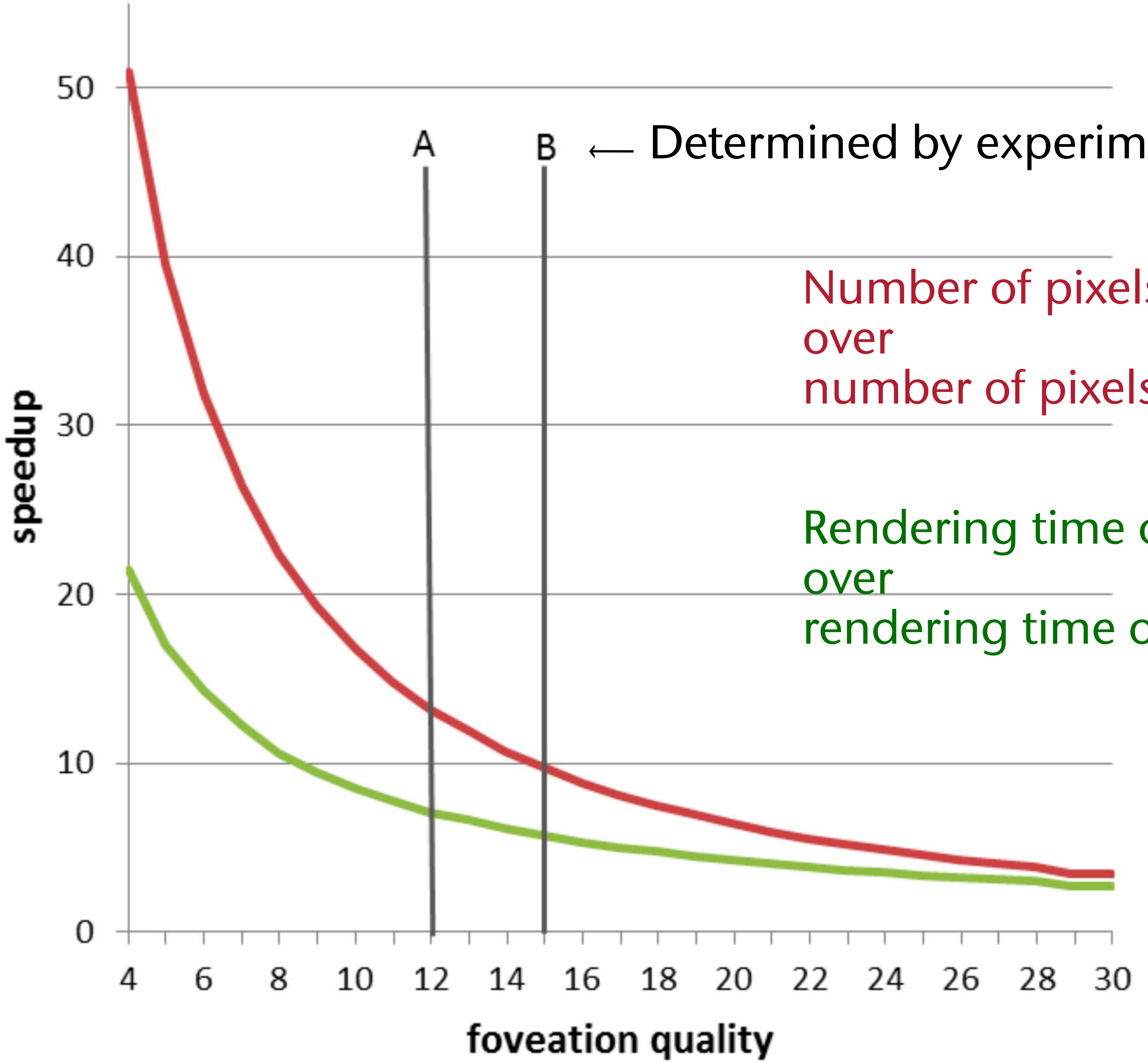
User Study to Determine Parameters

- Slider test:
  - Present participants the non-foveated animation sequence first
  - Then start with low degree of foveation (high rendering quality)
  - Let users increase level of foveation (decrease rendering quality) until just noticeable artifacts appear
  - Conditions: different animation speeds
- Results:



Video of User Study

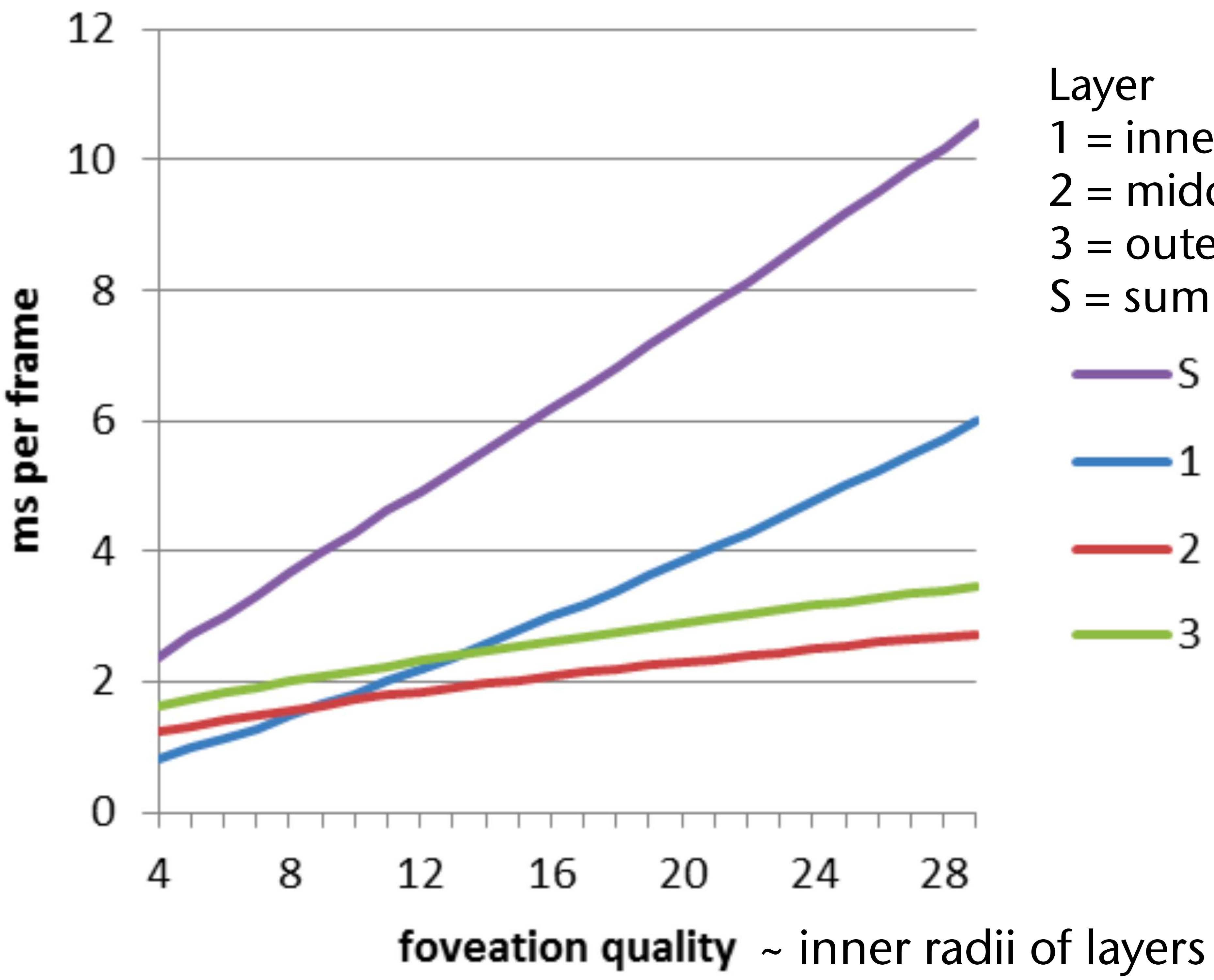
Speedup, Overall Results



← Determined by experiments

Number of pixels on display
over
number of pixels rendered

Rendering time of foveated version
over
rendering time of un-foveated version

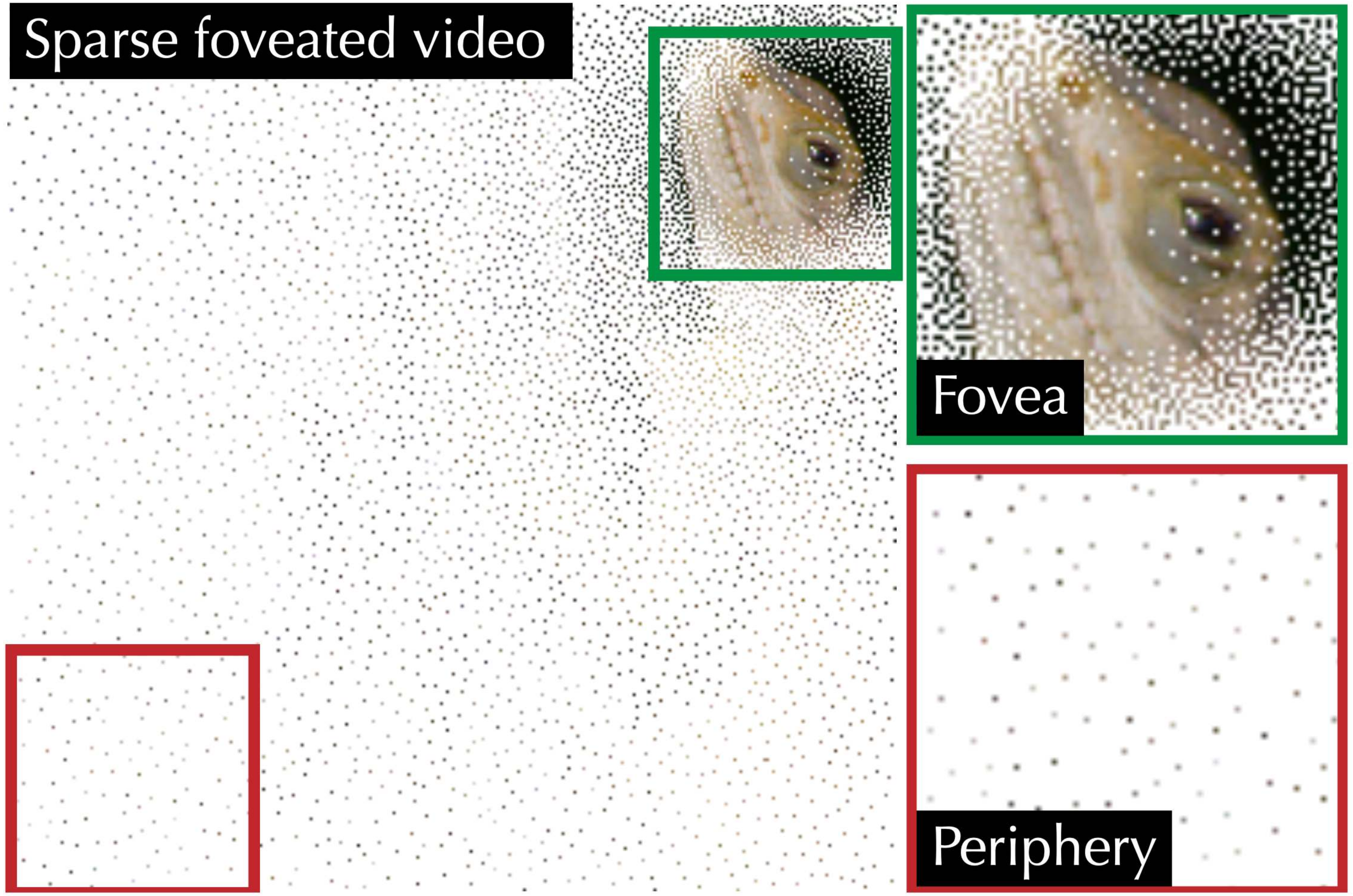


Layer
1 = inner,
2 = middle,
3 = outer.
S = sum

~ inner radii of layers 2 and 3

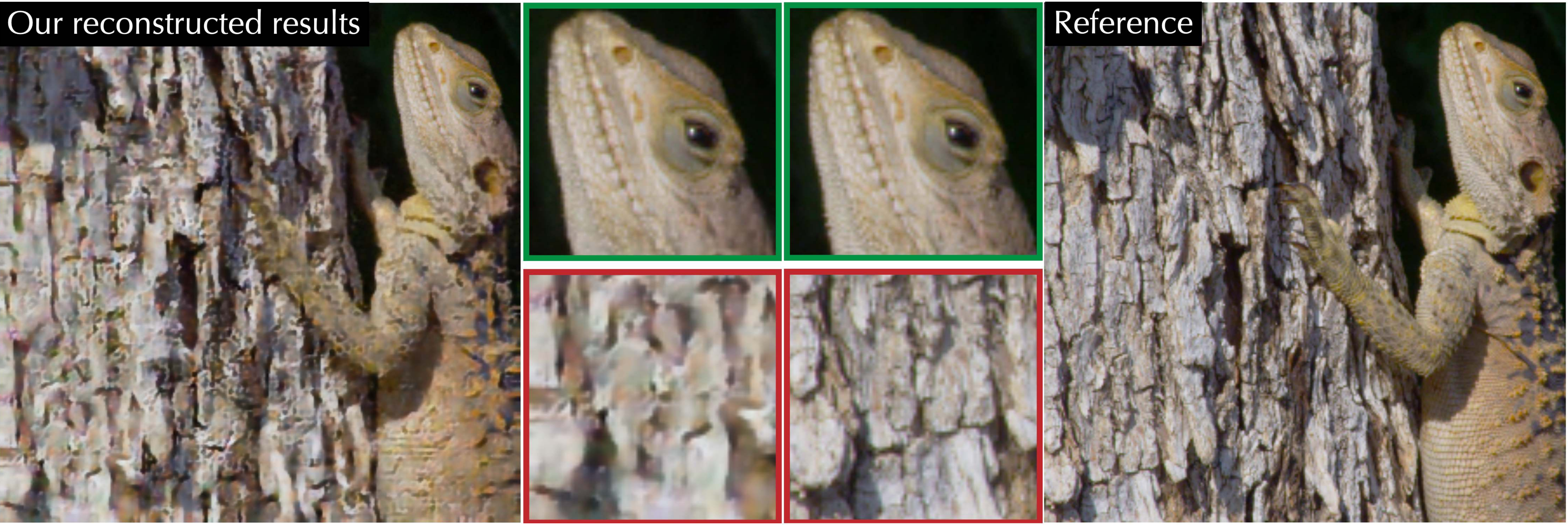
Further Improvements

- In order to reconstruct the whole image, use GANs (generator adversarial networks), instead of layered rendering, followed by anti-aliasing and blending
- Idea:
  - Generate mask with high density at fovea, low density in periphery
  - Render image at mask points
  - Fill in other pixels using GAN
  - Train GAN on large number of frames from video games and natural scene



["DeepFovea ...", 2019]

Comparison with Ground Truth

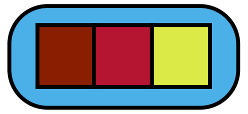


Runtime performance: 9 ms, using 4x NVIDIA Tesla V100 GPUs (2019)

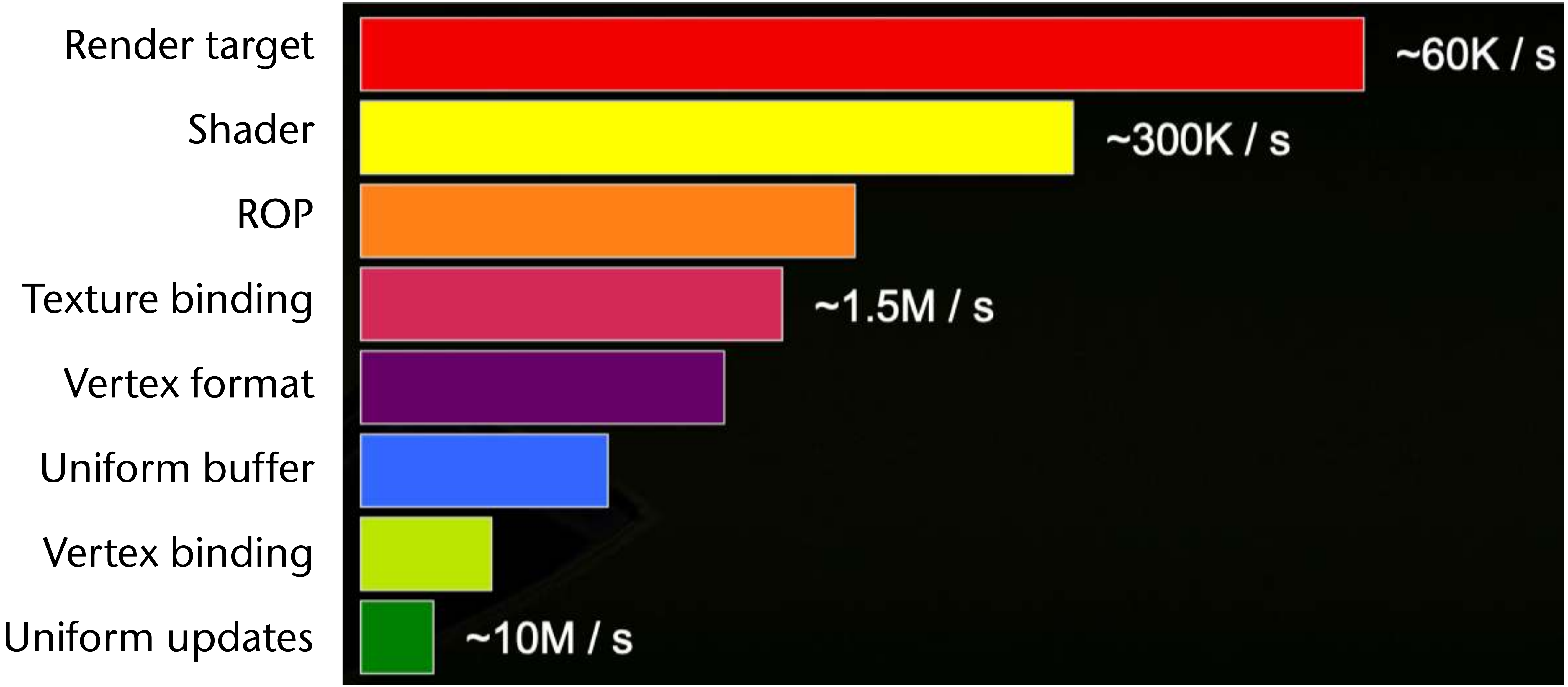
Get Creative: Are You Aware of Any Other Human Factors of the HVS that Might, Perhaps, be Utilized to Improve Rendering Performance?

https://www.menti.com/smvndia2ss

State Sorting

- A 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 of state changes in modern OpenGL [2014]

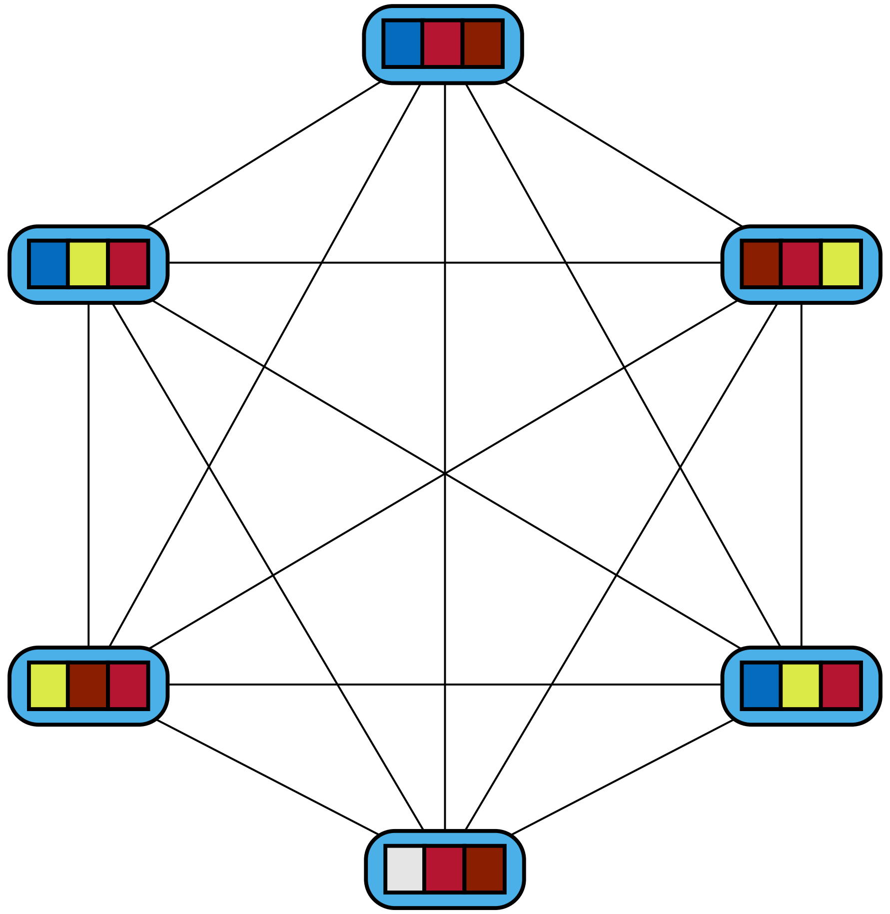


Not to scale!

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

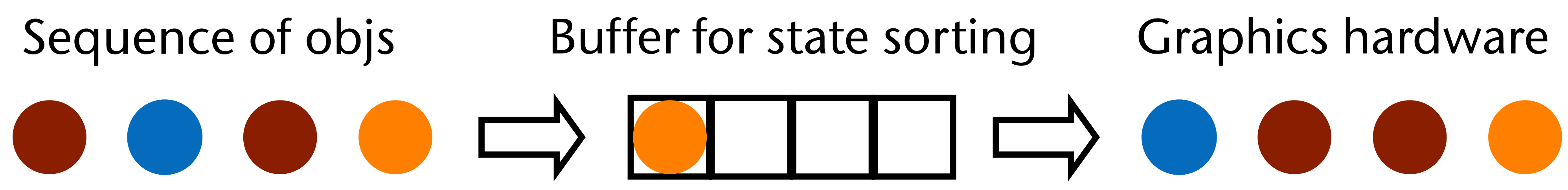
Solution: Sorting by State

- 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

- For the sake of argument: consider only *one* kind of attribute ("color")
- Introduce a buffer between application and graphics card
  - (Could be integrated into the driver, since an OpenGL command buffer already exists)



- Buffer contains *k* elements
- Perform one of 3 operations with each draw call (= app sends a "colored element" to the hardware/buffer):
  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

Interlude: Online Algorithms

- There are 2 categories of algorithms:
  - "Online" algorithms: the algorithm 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 implement a buffer, but it *can* utilize its knowledge of the future to decide whether to store elements)
- In the following, we consider only "lazy" online strategies:
  - Extract elements from the buffer only in case of buffer overflow
  - This is wlog., because every non-lazy online strategy can be converted into a lazy one with the 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?

Interlude: Competitive Analysis

- Definition *c-competitive* :

  Let $C_{\text{off}}(k)$ = costs of *optimal* offline strategy,

  let $C_{\text{on}}(k)$ = costs of *some* online strategy,

  "cost" = number of color changes, k = buffer size.

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

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

  where a must not depend on k (c may 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 $c = c(k)$ 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: this way, 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: $\Omega(\sqrt{k})$
- 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 $(x^k y^k)^{\frac{m}{2}} c_1^m \cdots c_m^m$

The Bounded Waste & the Random Choice Strategy

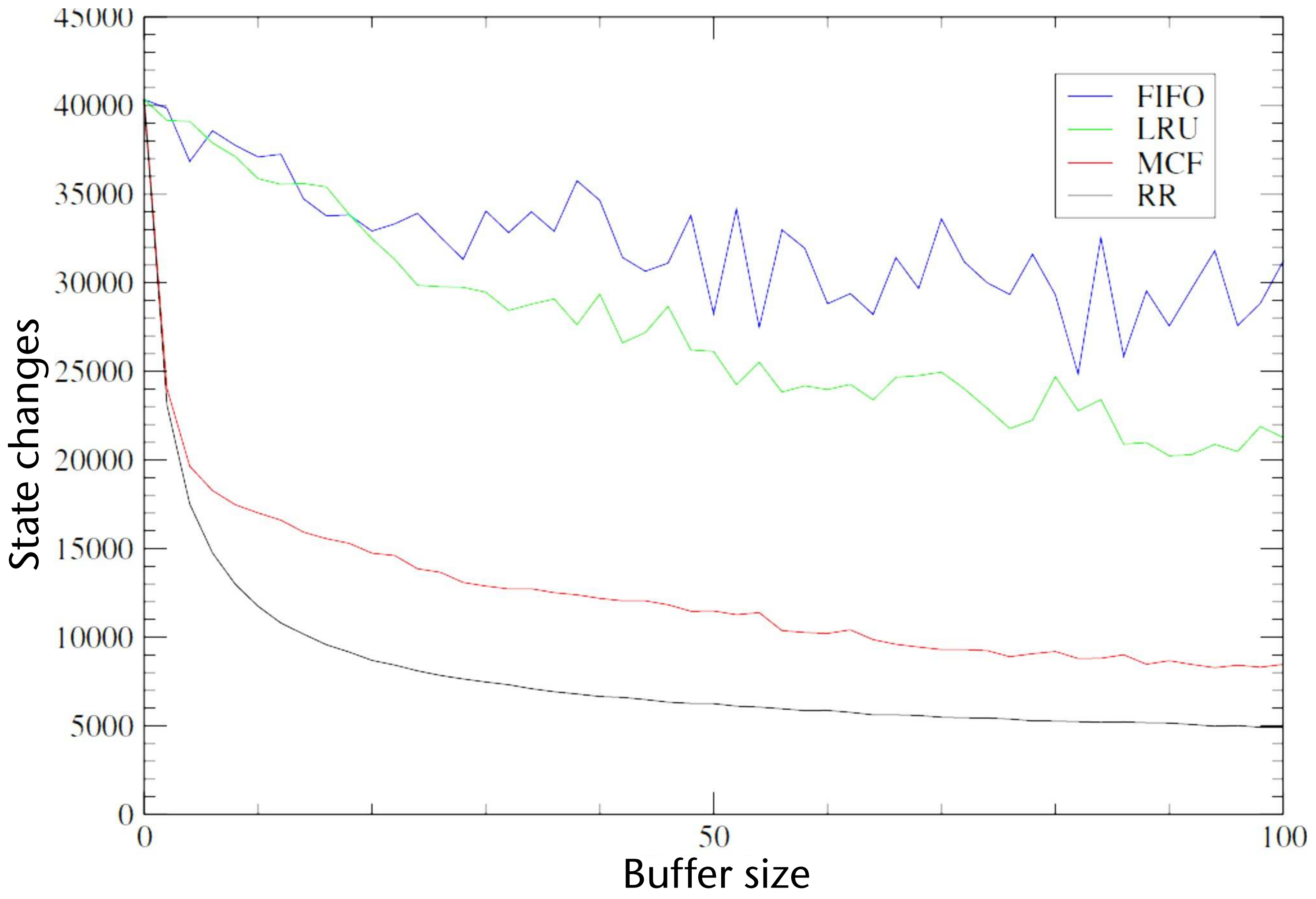
- Idea:
  - Count the number of all elements in the buffer that have the *same* color
  - Extract those elements whose color is most prevalent in the buffer
- Introduce waste counter $W(c)$:
  - With new element on input side: increment $W(c)$, c = color of new element
- Bounded waste strategy:
  - With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

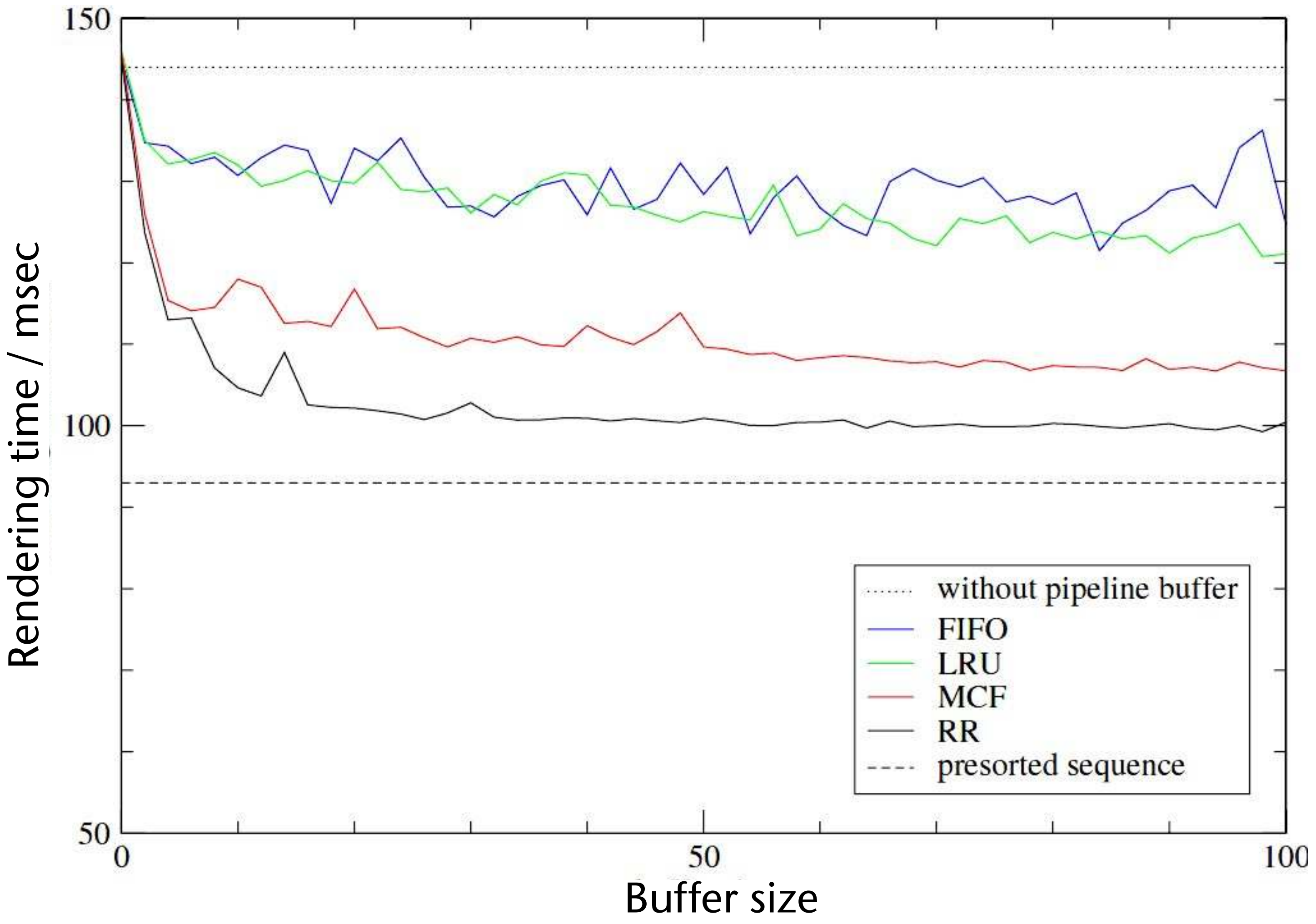
The Round Robin Strategy

- 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 (hence, "round robin")
  - Maintain pointer to current slot, move pointer to next slot every time a slot is chosen

Comparison

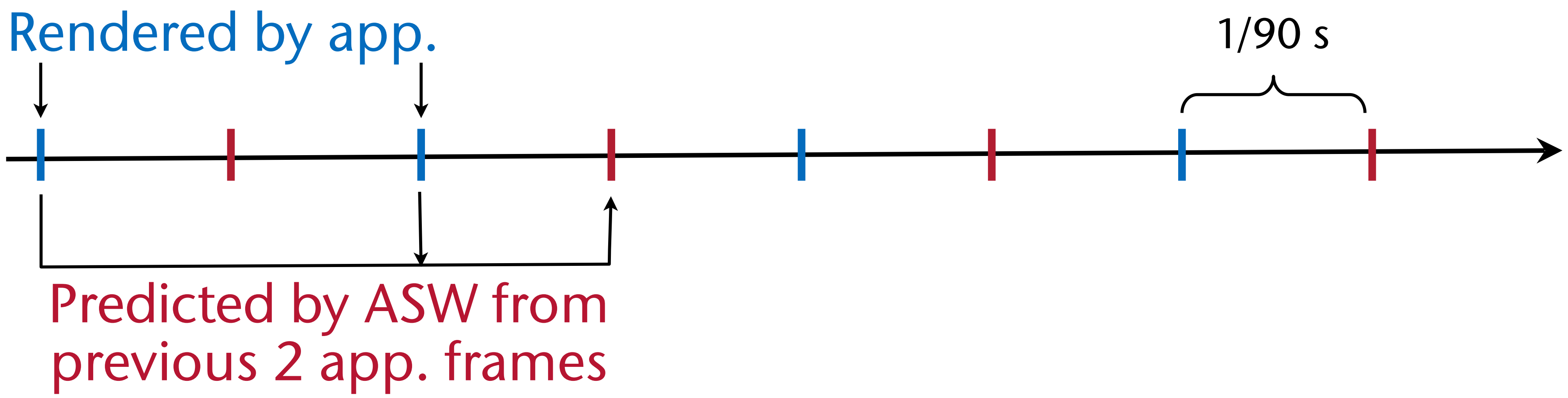
- Take-home message:
  - Round-robin yields very good results (although/and it is very simple)
  - Worst case doesn't say too much about performance in real-world applications

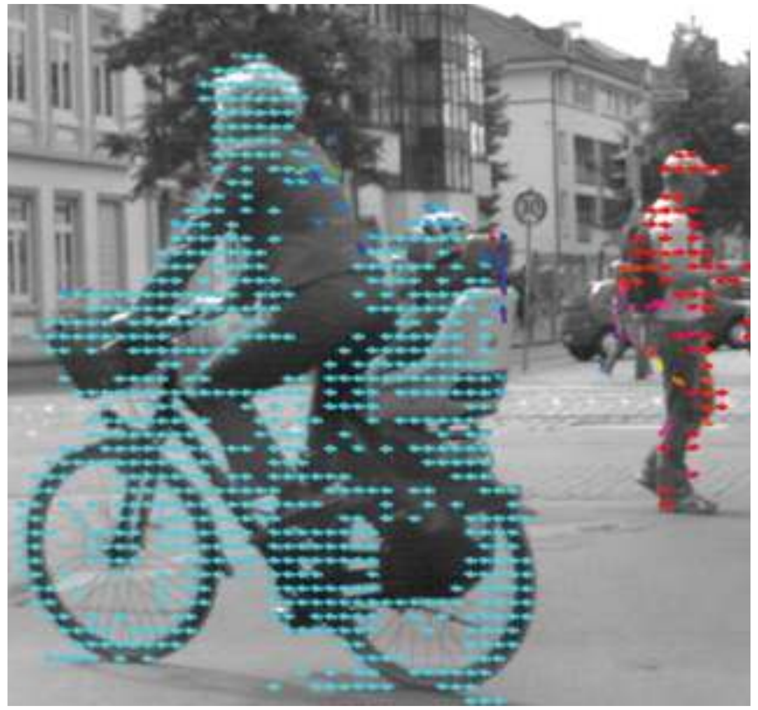




"Asynchronous Spacewarp" (Oculus)

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





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

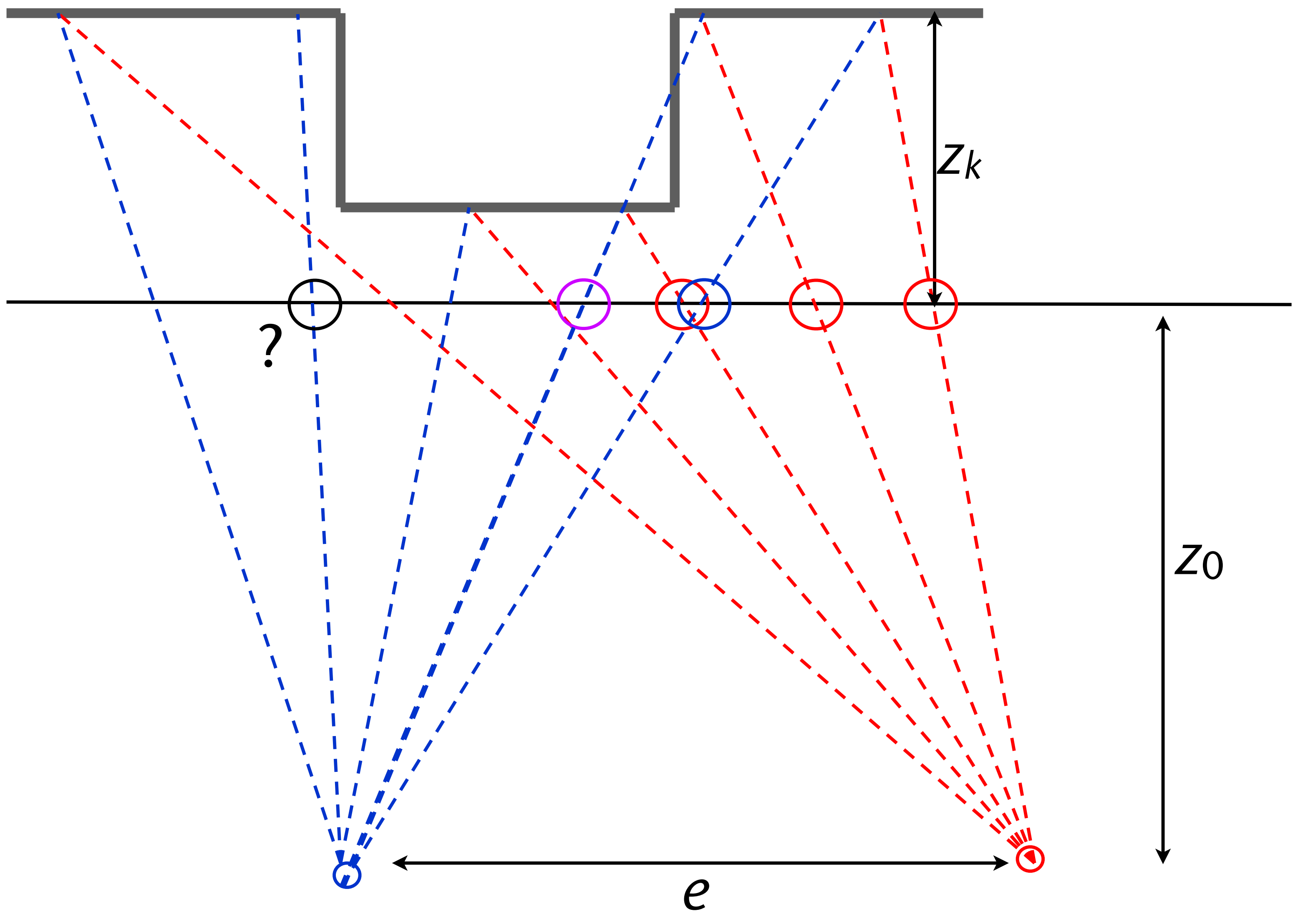
Example Frames (Can You Spot the Artefacts?)

Change in lighting

Disocclusion trail

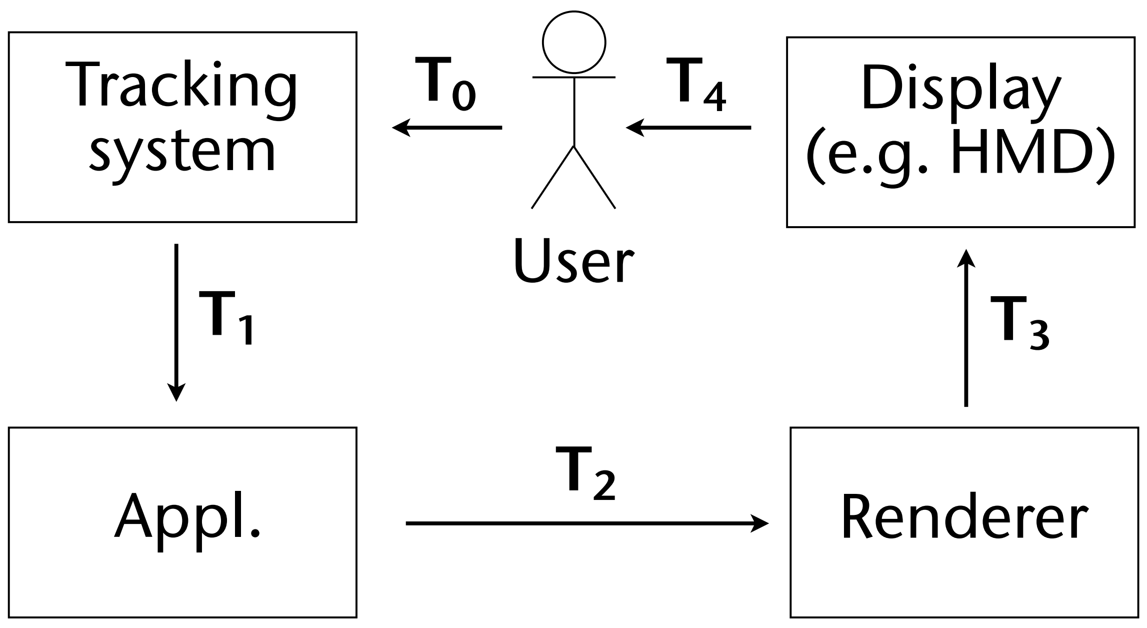
Stereoscopic Image Warping (Stereo Without 2x Rendering)

- 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
- Algoritm: consider all pixels on each scanline *from right to left*, draw each pixel *k* at the new x-coordinate $x'_k = x_k + \frac{e}{\Delta} \frac{z_k}{z_k + z_0}$ where Δ = pixel width
- Problems:
  - Up-vector must be vertical
  - Holes!
  - Ambiguities & aliasing
  - Reflections and specular highlights are at wrong position

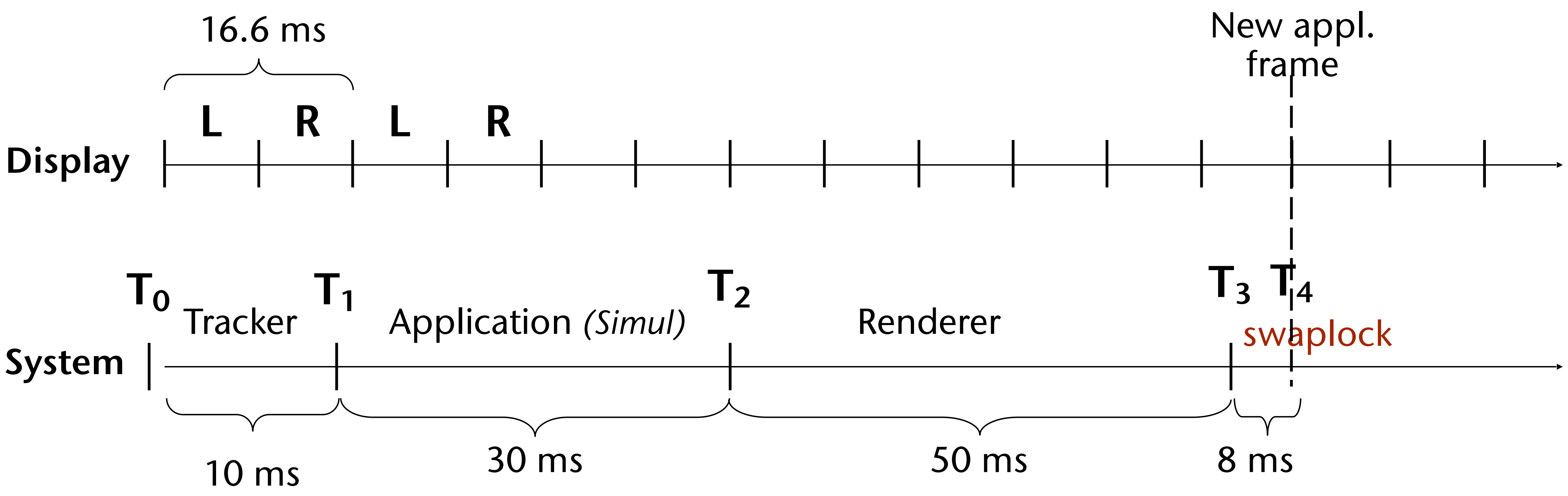


Reducing Latency by 3D Image Warping

- A simple VR system:



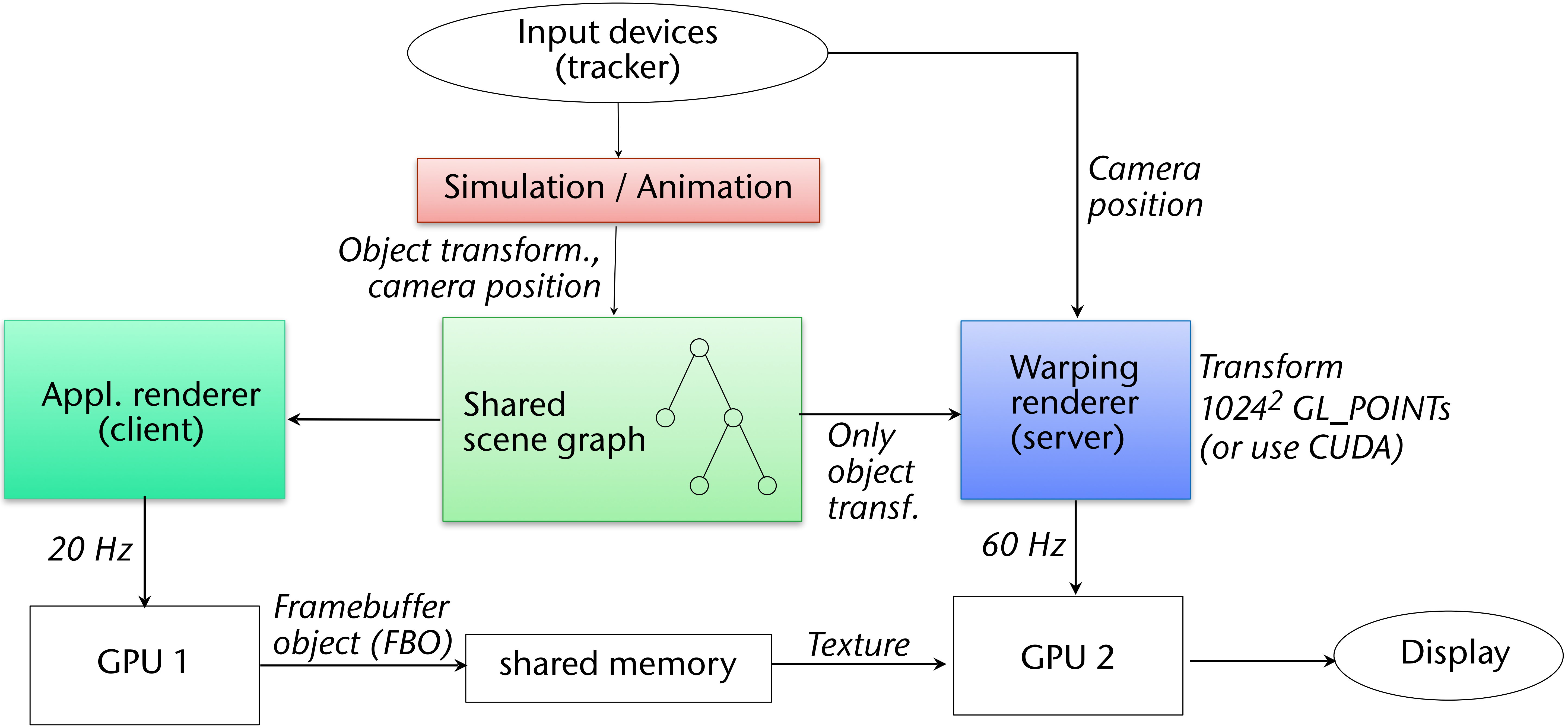
- Latency in this system (stereo with 60 Hz ⟶ display refresh = 120 Hz):



Issues & Observations

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

Idea: 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 (e.g., into separate frame buffer object)
  2. Save camera transformations at time t_1 : $T_{t_1,cam \leftarrow img}$ and $T_{t_1,wld \leftarrow cam}$
  3. With each object i , save its transformation $T^i_{t_1,obj \leftarrow wld}$

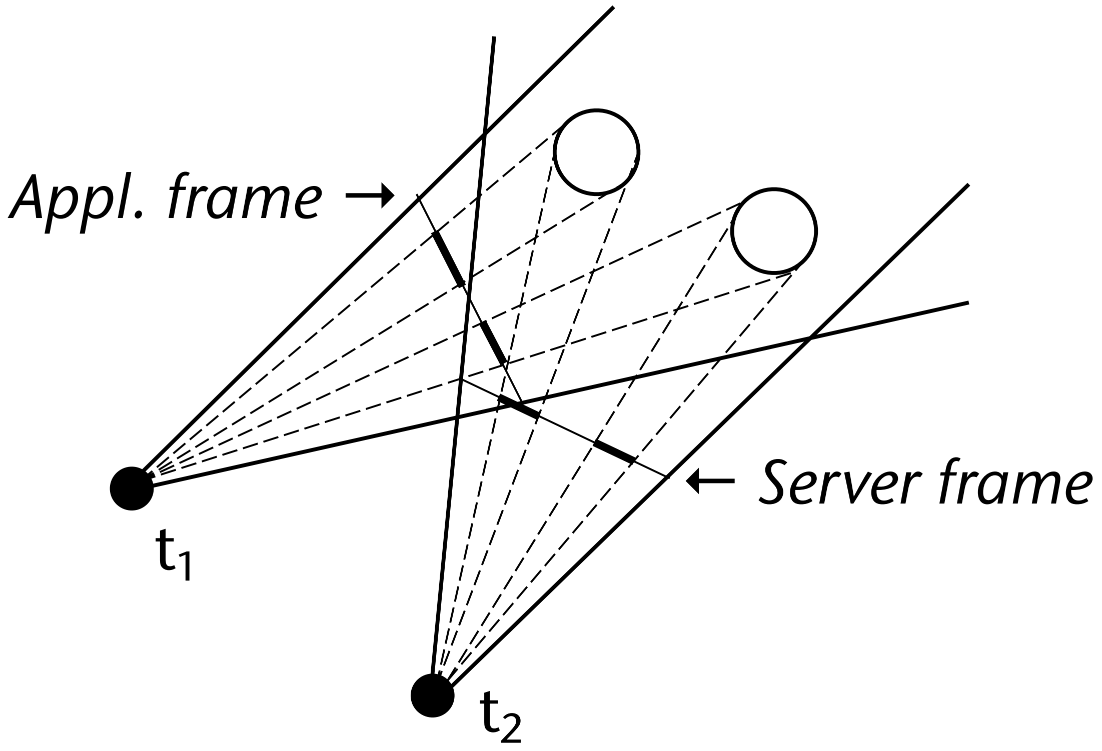
Warping of a Frame (Server)

- At a later time, t_2 , the server generates an image from an application frame by 3D warping
- Transformations known at this time:

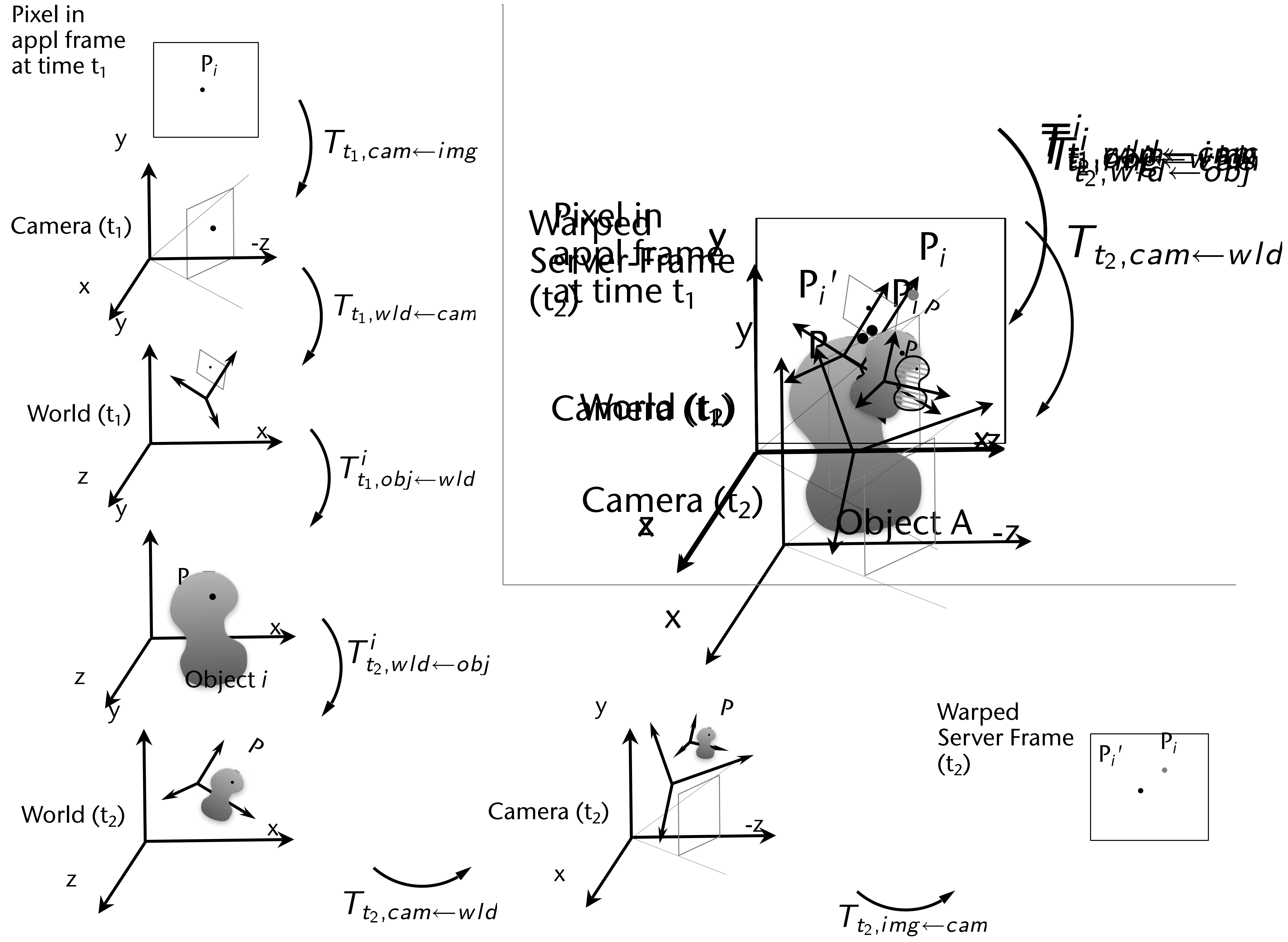
$$T^i_{t_2,wld \leftarrow obj} \qquad T_{t_2,img \leftarrow cam} \qquad T_{t_2,cam \leftarrow wld}$$

- A pixel $P_A = (x, y, z)$ in the application frame will be "warped" (transformed) 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 transformation matrix can be precomputed for each object and each new server frame

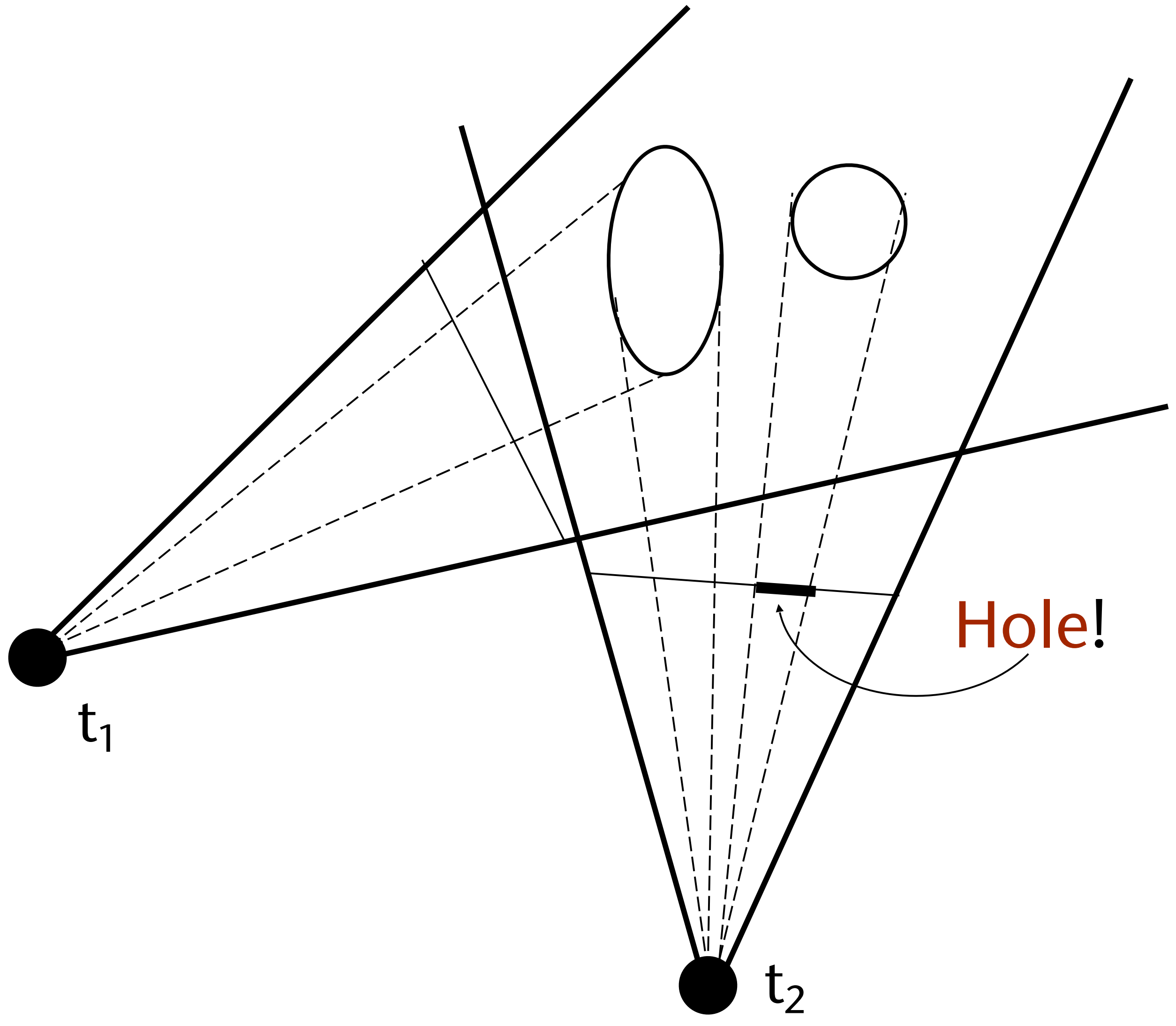

Pixel in
appl frame
at time t1
P_i
$T_{t_1,cam \leftarrow img}$
Camera (t1)
$T_{t_1,wld \leftarrow cam}$
World (t1)
$T^i_{t_1,obj \leftarrow wld}$
Object i
$T^i_{t_2,wld \leftarrow obj}$
World (t2)
$T_{t_2,cam \leftarrow wld}$
Camera (t2)
$T_{t_2,img \leftarrow cam}$
Warped
Server Frame
(t2)
P_i'
Object A

Remarks

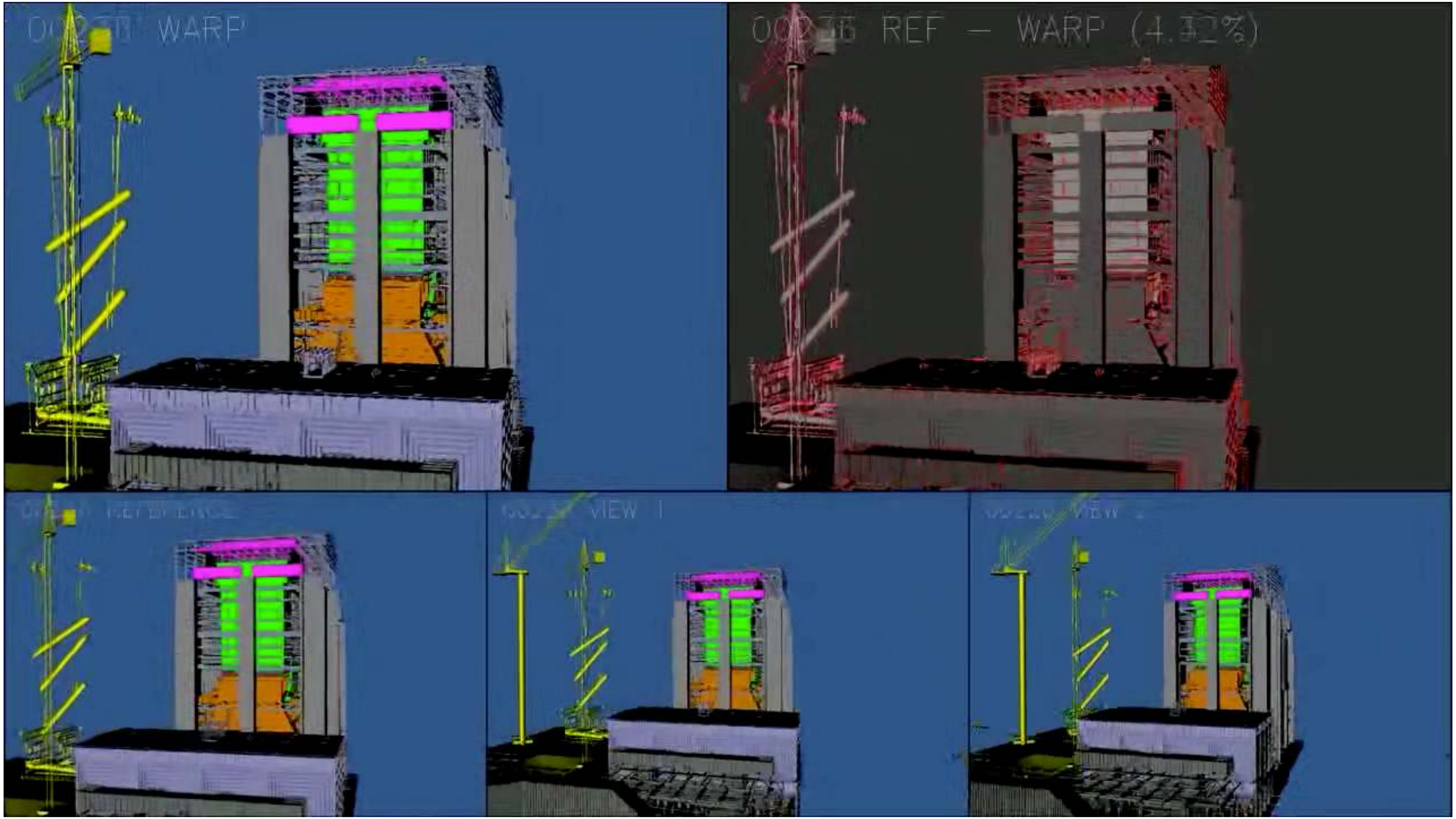
- Implementation of the warping:
  - Could be done in the vertex shader
    - Doesn't work in the fragment shader, because the output (= pixel) position is fixed in fragment shaders!
  - Better do the warping in CUDA, one thread per pixel in the appl frame
- Note: the server (warping) renderer does use current (t_2) positions of animated/simulated objects!
- Advantages:
  - The frames (visible to the user) are now "more current", because of more current camera *and* object positions (i.e., animated objects)
  - Server framerate is independent of number of polygons
  - With additional tricks, re-lighting is possible (to some extent)

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 (4.32%)
VIEW 1