Lecture 18:

Mapping Shading Languages to GPU Hardware
ML Framework Discussion
A Quick Lecture on Rendering for VR

Visual Computing Systems

Stanford CS348K, Fall 2018

Shading system implementation

(Efficiently mapping shading computations to GPU hardware)

Shading often has very high arithmetic intensity

```
sampler mySamp;
Texture2D<float3> myTex;
float3 ks;
float  shinyExp;
float3 lightDir;
float3 viewDir;

float4 phongShader(float3 norm, float2 uv)
{
  float result;
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  float spec = dot(viewDir, 2 * dot(-lightDir, norm) * norm + lightDir);
  result = kd * clamp(dot(lightDir, norm), 0.0, 1.0);
  result += ks * exp(spec, shinyExp);
  return float4(result, 1.0);
}
```

Image credit: http://caig.cs.nctu.edu.tw/course/CG2007

3 scalar float operations + 1 exp()
8 float3 operations + 1 clamp()
1 texture access

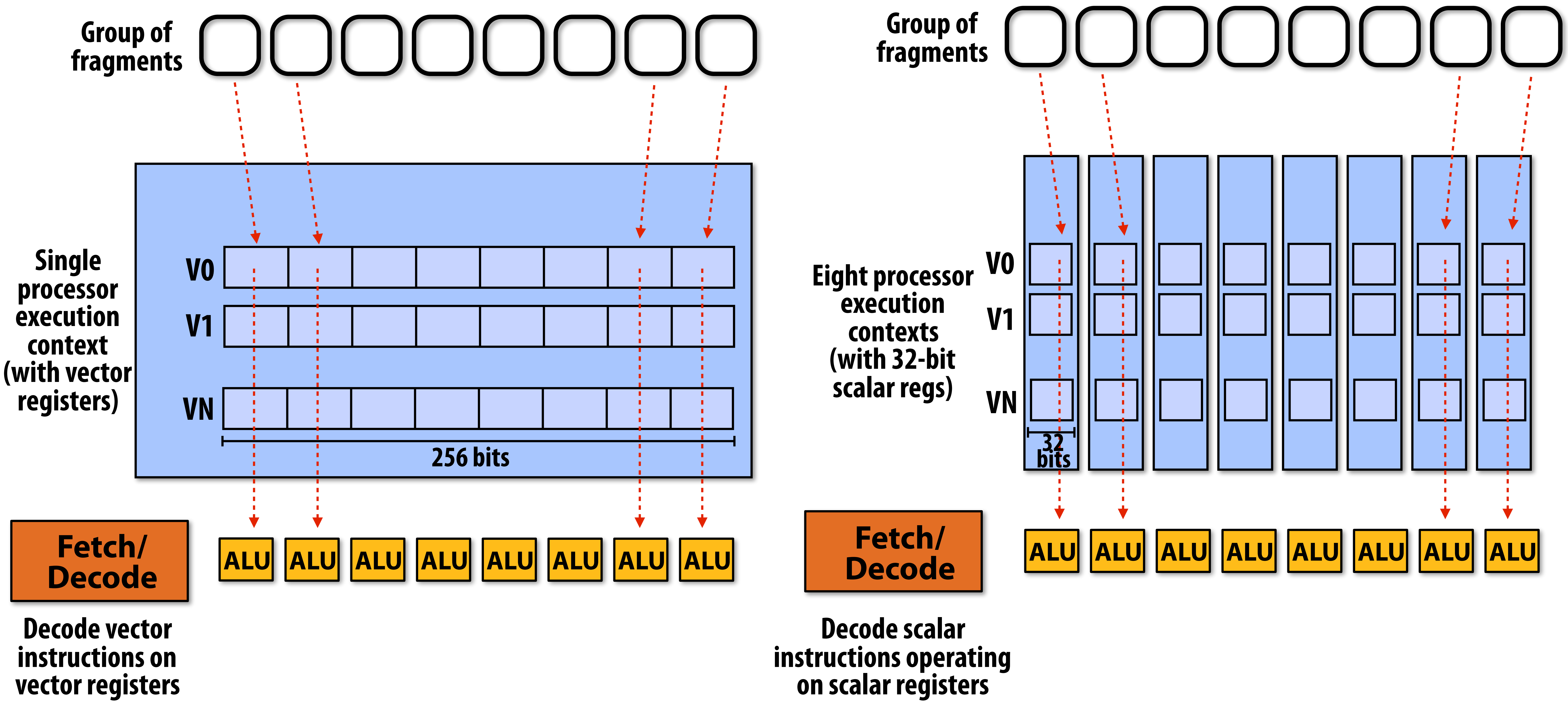
Vertex processing often has even higher arithmetic intensity than fragment processing (less use of texturing)

Review: fictitious throughput processor

- Processor decodes one instruction per clock
- Instruction controls all eight SIMD execution units
  - SIMD = "single instruction multiple data"
- "Explicit" SIMD:
  - Vector instructions manipulate contents of 8x32-bit (256 bit) vector registers
  - Execution is all within one hardware execution context
- "Implicit" SIMD (SPMD, "SIMT"):
  - Hardware executes eight unique execution contexts in "lockstep"
  - Program binary contains scalar instructions manipulating 32-bit registers

Mapping fragments to execution units:

Map fragments to "vector lanes" within one execution context (explicit SIMD parallelism) or to unique contexts that share an instruction stream (parallelization by hardware)



GLSL/HLSL shading languages adopt a SPMD programming model

- SPMD = single program, multiple data
  - Programming model used in writing GPU shader programs
    - What's the program?
    - What's the data?
  - Also adopted by CUDA and ISPC

- How do we implement a SPMD program on SIMD hardware?

Example 1: shader with a conditional

```
sampler mySamp;
Texture2D<float3> myTex;

float4 fragmentShader(float3 norm, float2 st, float4 frontColor, float4 backColor)
{
  float4 tmp;
  if (norm[2] < 0)  // sidedness check (direction of Z component of normal)
  {
     tmp = backColor;
  }
  else
  {
     tmp = frontColor;
     tmp *= myTex.sample(mySamp, st);
  }
  return tmp;
}
```

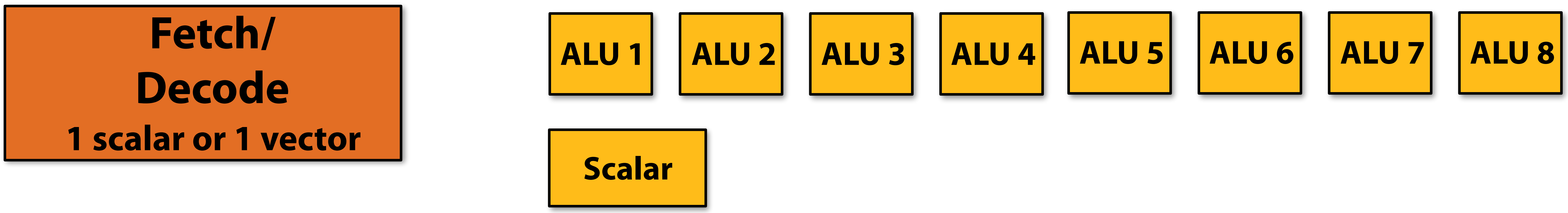
Example 2: predicate is uniform expression

```
sampler mySamp;
Texture2D<float3> myTex;
float myParam;      // uniform value
float myLoopBound;

float4 fragmentShader(float3 norm, float2 st, float4 frontColor, float4 backColor)
{
   float4 tmp;
   if (myParam < 0.5)
   {
       float scale = myParam * myParam;
       tmp = scale * frontColor;
   }
   else
   {
      tmp = backColor;
   }
   return tmp;
}
```

Notice:
predicate is uniform expression
(same result for all fragments)

Improved efficiency: processor executes uniform instructions using scalar execution units

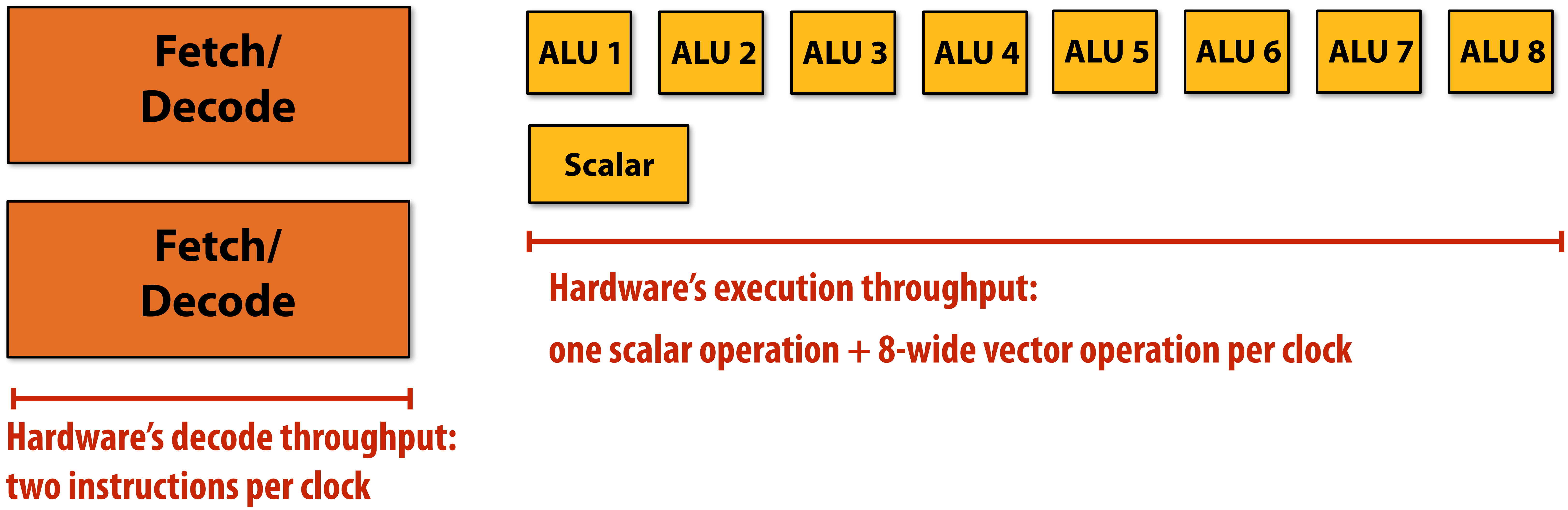


Logic shared across all "vector lanes" need only be performed once (not repeated by every vector ALU)

Scalar logic identified at compile time (compiler generates different instructions)

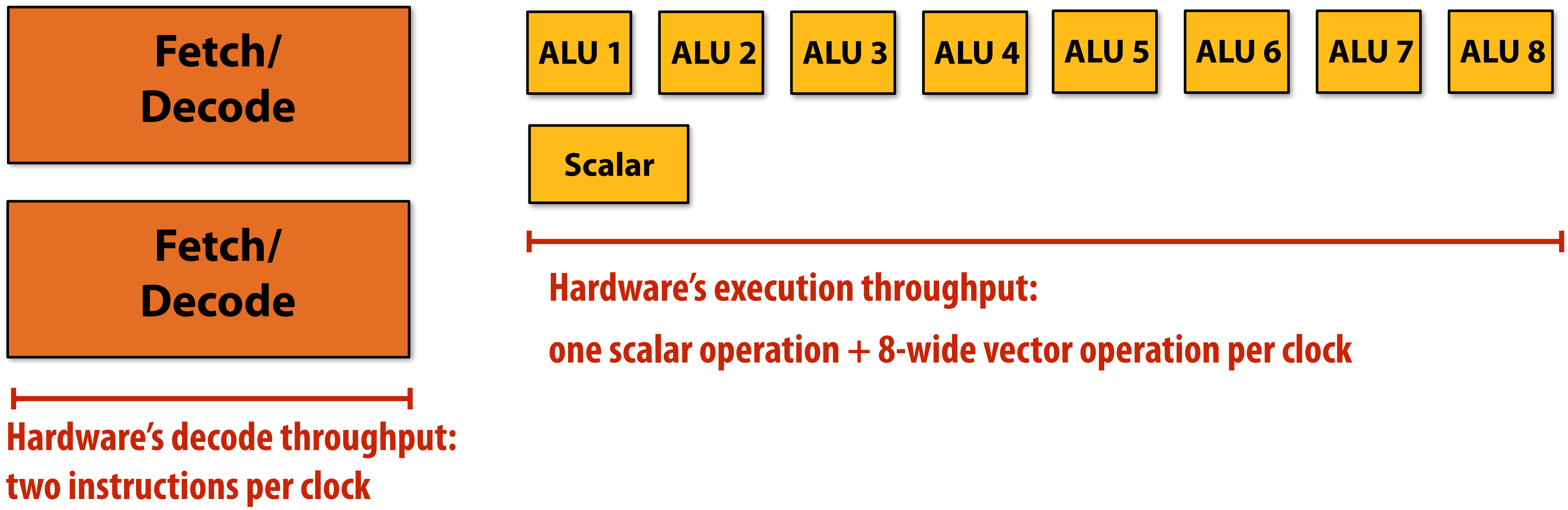
```
float3 lightDir[MAX_NUM_LIGHTS];
int numLights;
float4 multiLightFragShader(float3 norm, float4 surfaceColor)
{
   float4 outputColor;
   for (int i=0; i<num_lights; i++) {
     outputColor += surfaceColor * clamp(0.0, 1.0, dot(norm, lightDir[i]));
   }
}
```

Improving the fictitious throughput processor



- Now decode two instructions per clock
  - How should we organize the processor to execute those instructions?

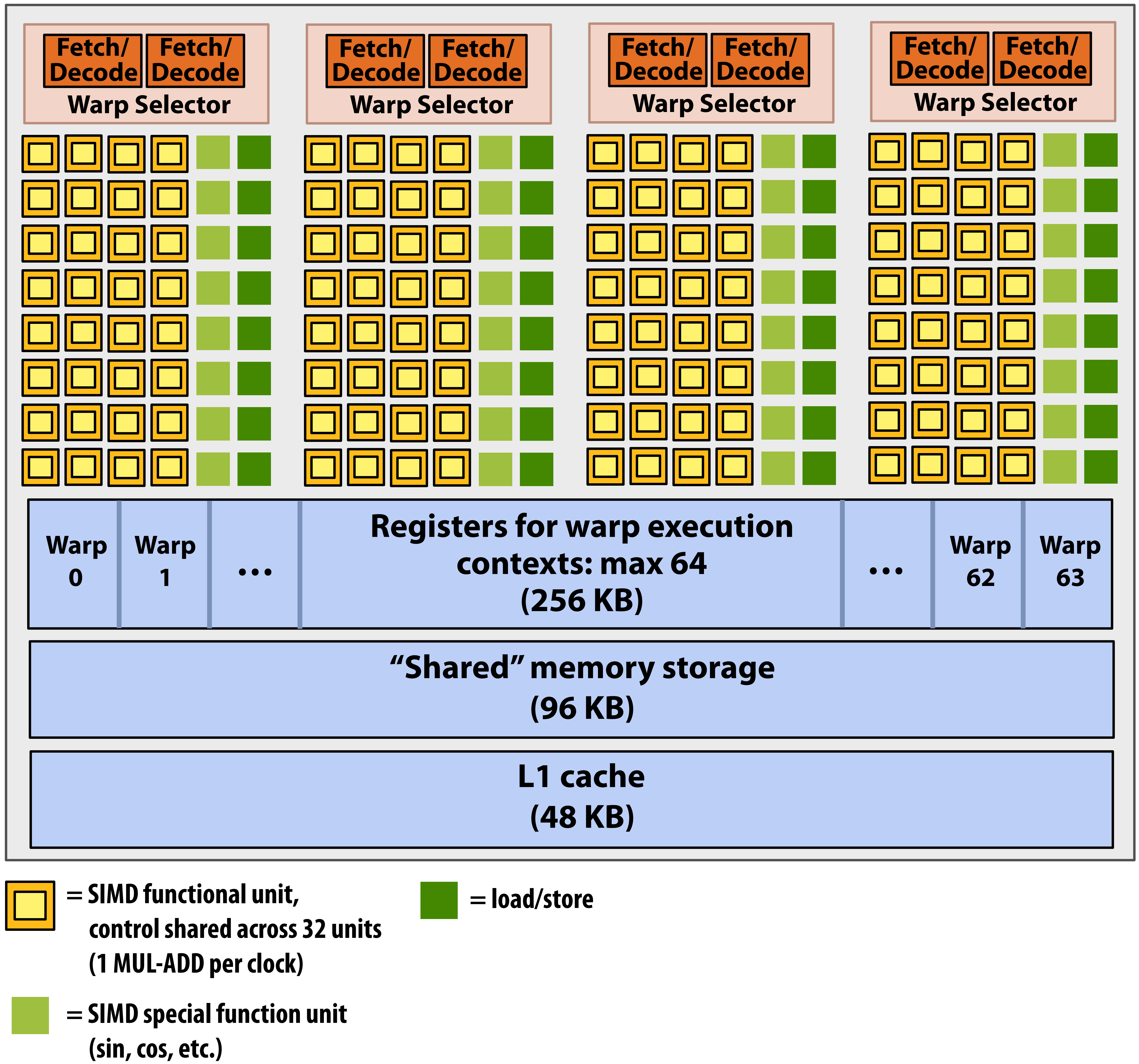
Three possible organizations



- Execute two instructions (one scalar, one vector) from same execution context
  - One execution context can fully utilize the processor's resources, but requires instruction-level-parallelism in instruction stream
- Execute unique instructions in two different execution contexts
  - Processor needs two runnable execution contexts (twice as much parallel work must be available)
  - But no ILP in any instruction stream is required to run machine at full throughput
- Execute two SIMD operations in parallel (e.g., two 4-wide operations)
  - Significant change: must modify how ALUs are controlled: no longer 8-wide SIMD
  - Instructions could be from same execution context (ILP) or two different ones

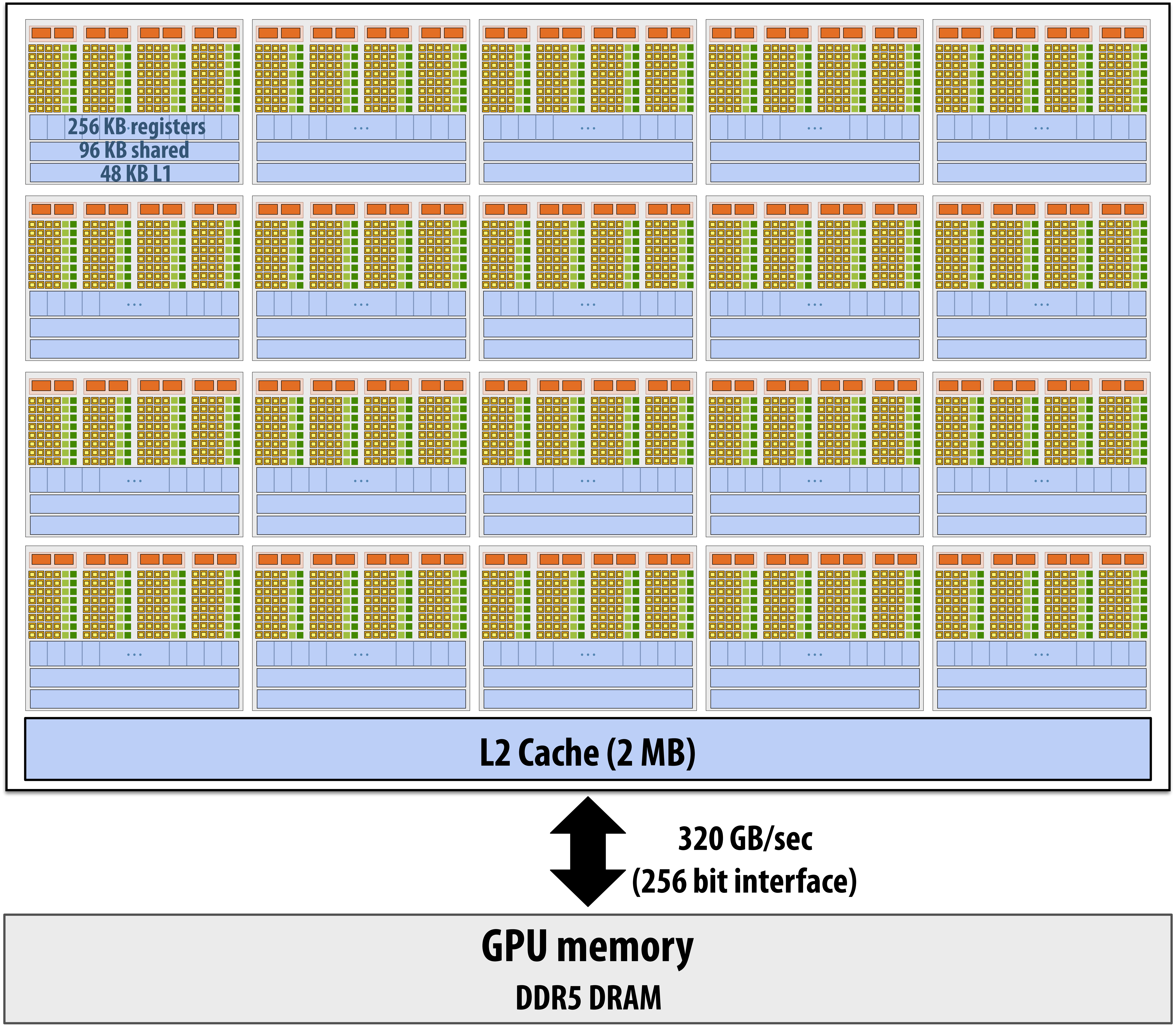
NVIDIA GTX 1080 (2016)

This is one NVIDIA Pascal GP104 streaming multi-processor (SM) unit



- Instructions operate on 32 pieces of data at a time (instruction streams called "warps").
- Different instructions from up to four warps can be executed simultaneously (simultaneous multi-threading)
- Up to 64 warps are interleaved on the SM (interleaved multi-threading)
- Over 2,048 fragments/ vertices/etc can be processed concurrently by a core

NVIDIA GTX 1080 (20 SMs)

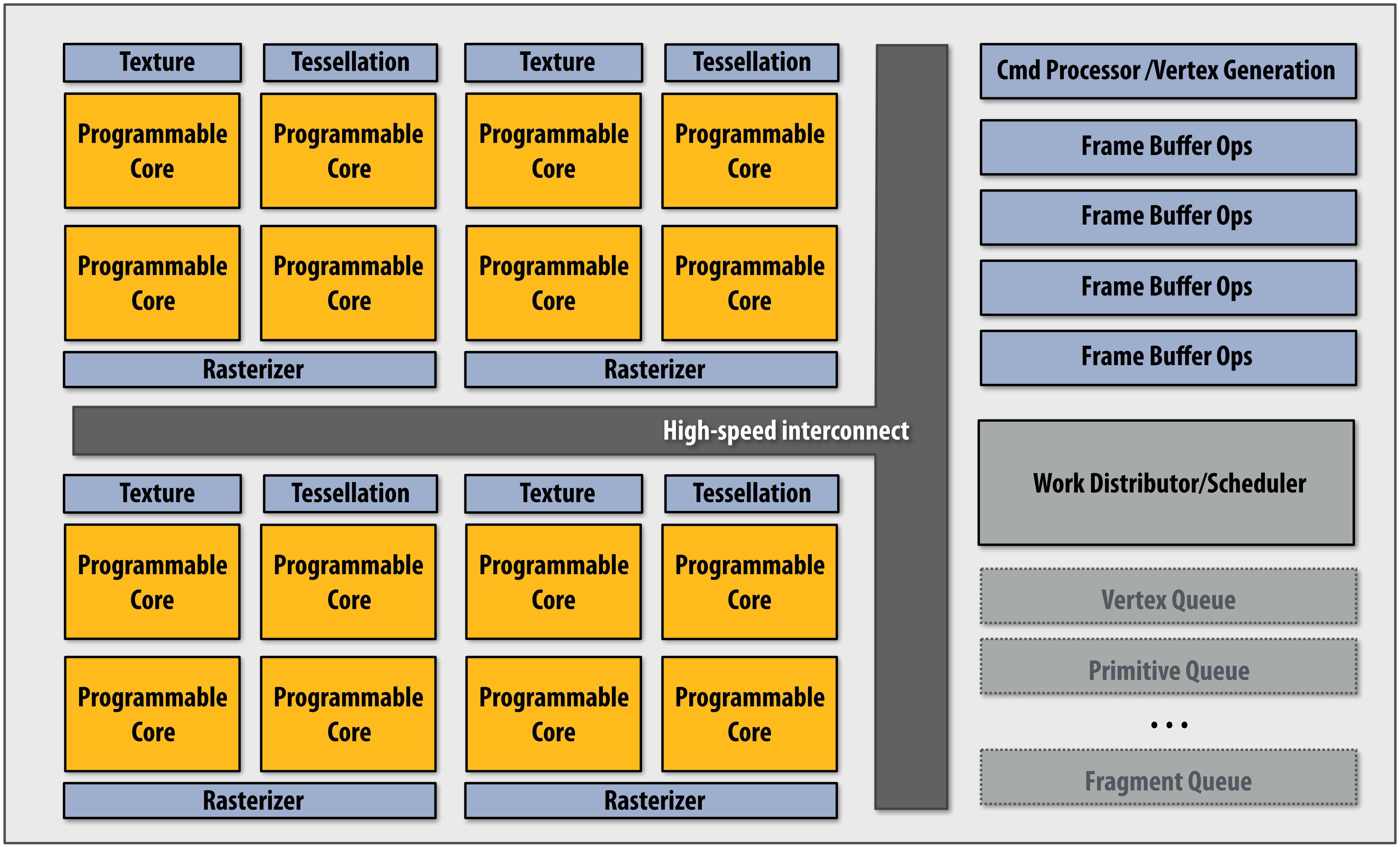


Shading languages summary

- **Convenient/simple abstraction:**
  - Wide application scope: implement any logic within shader function subject to input/output constraints.
  - Independent per-element SPMD programming model (no loops over elements, no explicit parallelism)
  - Built-in primitives for texture mapping
- **Facilitate high-performance implementation:**
  - SPMD shader programming model exposes parallelism (independent execution per element)
  - Shader programming model exposes texture operations (can be scheduled on specialized HW)
- **GPU implementations:**
  - Wide SIMD execution (shaders feature coherent instruction streams)
  - High degree of multi-threading (multi-threading to avoid stalls despite large texture access latency)
    - e.g., NVIDIA GPU: 16 times more warps (execution contexts) than can be executed per clock
  - Fixed-function hardware implementation of texture filtering (efficient, performant)
  - High performance implementations of transcendentals (sin, cos, exp) -- common operations in shading

One final thought

Recall: modern GPU is a heterogeneous processor



An unusual aspect of GPU design (when running graphics pipeline)

- Fixed-function components on a GPU control the operation of the programmable components
  - Fixed-function logic generates work (input assembler, tessellator, rasterizer generate elements)
  - Programmable logic defines how to process generated elements

- Application-programmable logic forms the inner loops of the rendering computation, not the outer loops!

Contrast this design to video decode/tensor core interfaces on a SoC

- Ongoing debate: can we flip this design around?
  - Maintain efficiency of heterogeneous hardware implementation, but give software control of how pipeline is mapped to hardware resources

Discussion: what are the key components of a DL framework?

Concept 1

- Defining operations and graphs

- Recall words of wisdom from Bill Mark
  - The reason to use accelerators is for performance
  - So high-productivity programming language better not prevent you from getting good performance

Operators written in lower-level languages

- Common design choice in major frameworks like TensorFlow/ MX.net/PyTorch

```
tf.nn.conv2d(
    input,
    filter,
    strides,
    padding,
    use_cudnn_on_gpu=True,
    data_format='NHWC',
    dilations=[1, 1, 1, 1],
    name=None
)
```

Defined in generated file: `tensorflow/python/ops/gen_nn_ops.py`.

Computes a 2-D convolution given 4-D `input` and `filter` tensors.

Given an input tensor of shape `[batch, in_height, in_width, in_channels]` and a filter / kernel tensor of shape `[filter_height, filter_width, in_channels, out_channels]`, this op performs the following:

1. Flattens the filter to a 2-D matrix with shape `[filter_height * filter_width * in_channels, output_channels]`.
2. Extracts image patches from the input tensor to form a *virtual* tensor of shape `[batch, out_height, out_width, filter_height * filter_width * in_channels]`.
3. For each patch, right-multiplies the filter matrix and the image patch vector.

In detail, with the default NHWC format,

```
output[b, i, j, k] =
    sum_{di, dj, q} input[b, strides[1] * i + di, strides[2] * j + dj, q] *
                    filter[di, dj, q, k]
```

Must have `strides[0] = strides[3] = 1`. For the most common case of the same horizontal and vertices strides, `strides = [1, stride, stride, 1]`.

Args:

- `input`: A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`. A 4-D tensor. The dimension order is interpreted according to the value of `data_format`, see below for details.

Operation implementations in low-level language like CUDA, or performance library like cuDNN or MKL-DNN

Challenge many parameters to existing operators, researchers create new types of operators

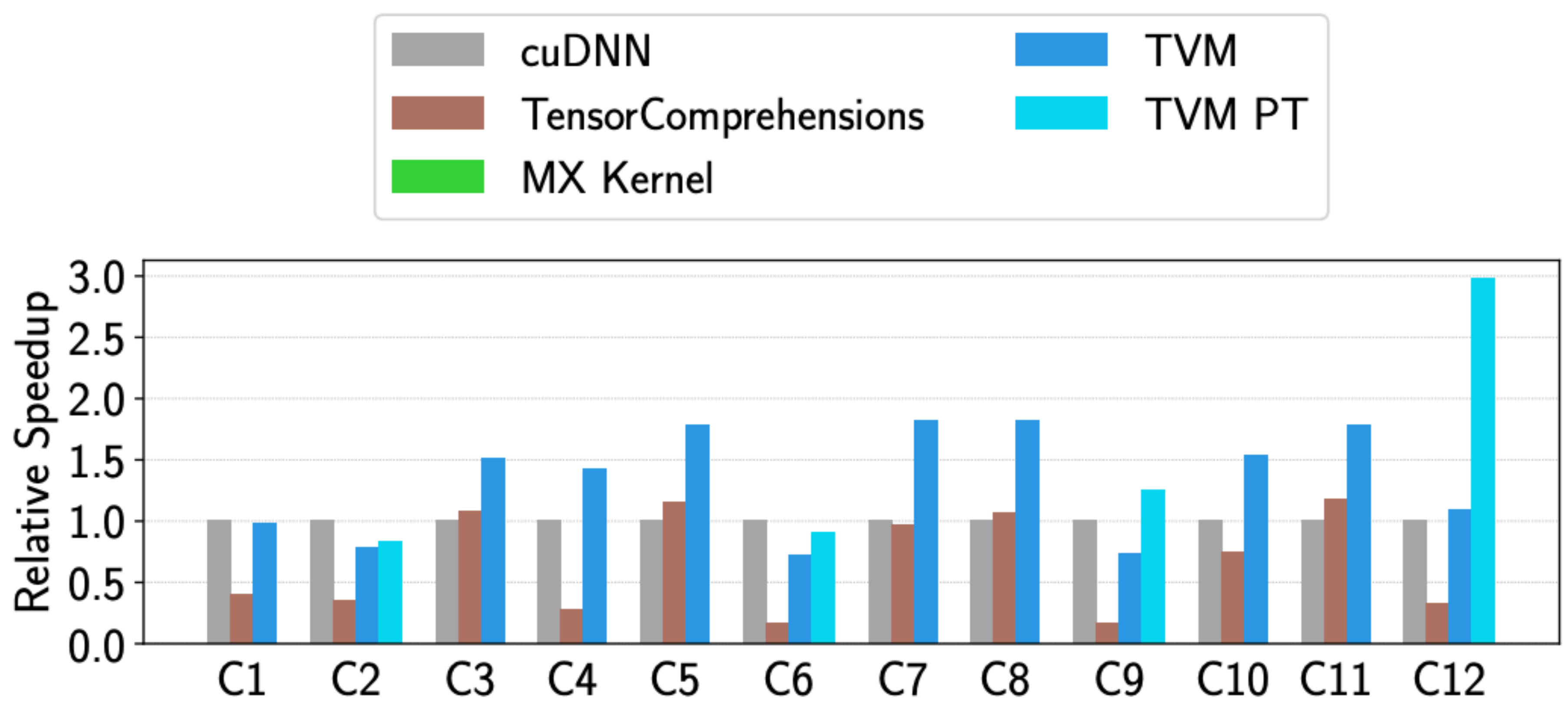
Name	Operator	H,W	IC,OC	K,S
C1	conv2d	224, 224	3,64	7, 2
C2	conv2d	56, 56	64,64	3, 1
C3	conv2d	56, 56	64,64	1, 1
C4	conv2d	56, 56	64,128	3, 2
C5	conv2d	56, 56	64,128	1, 2
C6	conv2d	28, 28	128,128	3, 1
C7	conv2d	28, 28	128,256	3, 2
C8	conv2d	28, 28	128,256	1, 2
C9	conv2d	14, 14	256,256	3, 1
C10	conv2d	14, 14	256,512	3, 2
C11	conv2d	14, 14	256,512	1, 2
C12	conv2d	7, 7	512,512	3, 1

Name	Operator	H,W	IC	K,S
D1	depthwise conv2d	112, 112	32	3, 1
D2	depthwise conv2d	112, 112	64	3, 2
D3	depthwise conv2d	56, 56	128	3, 1
D4	depthwise conv2d	56, 56	128	3, 2
D5	depthwise conv2d	28, 28	256	3, 1
D6	depthwise conv2d	28, 28	256	3, 2
D7	depthwise conv2d	14, 14	512	3, 1
D8	depthwise conv2d	14, 14	512	3, 2
D9	depthwise conv2d	7, 7	1024	3, 1

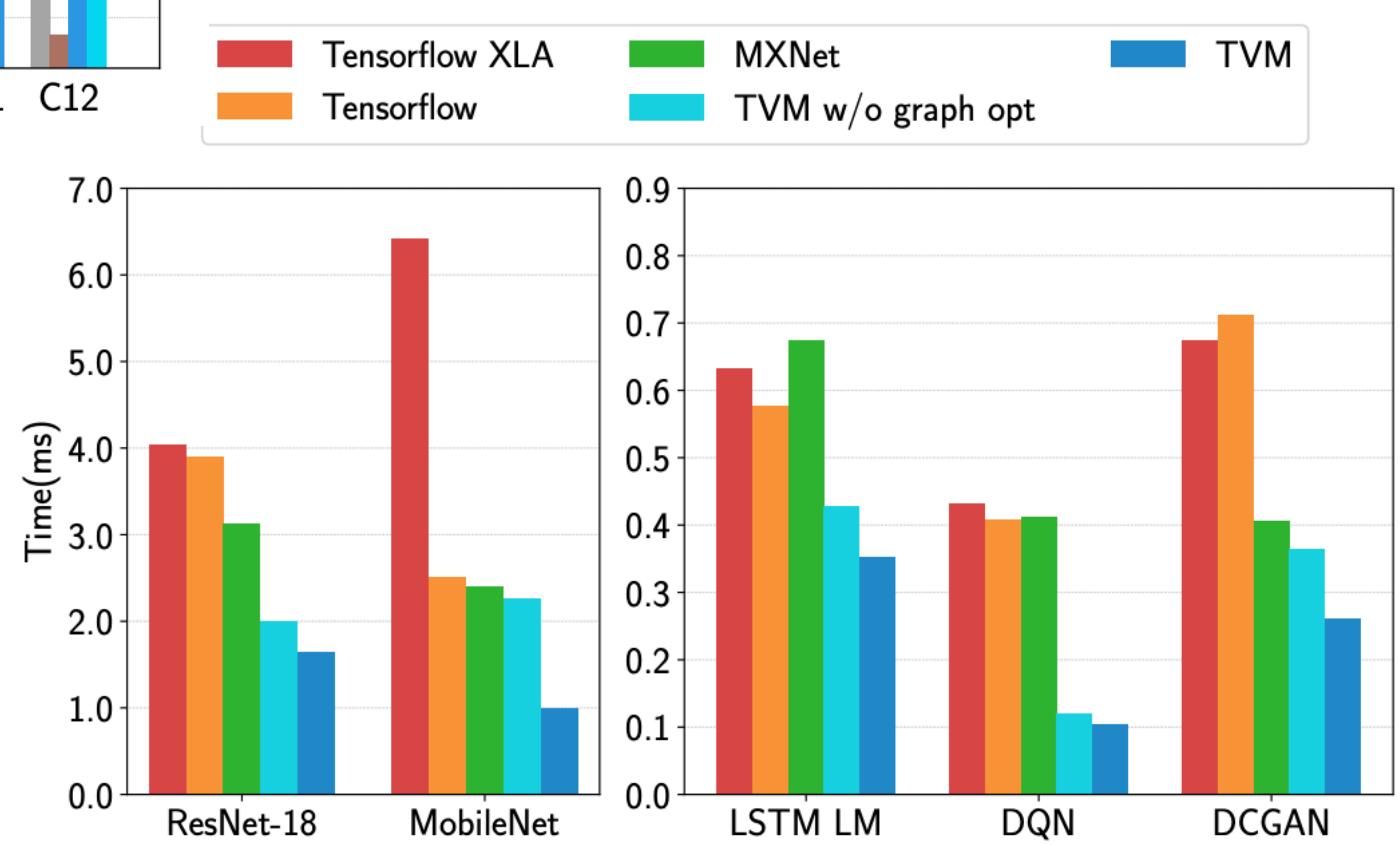
Increasing use of neural architecture search is leading to increasing number of layer parameterizations.

Interest in compiler support for generating implementations

- **Example: AutoTVM**
  - Simulated annealing based search over schedule space
  - Variant of Halide scheduling language where programmer defines parameterized space space, not a specific schedule



ResNet-18 convlayers



Concept 2

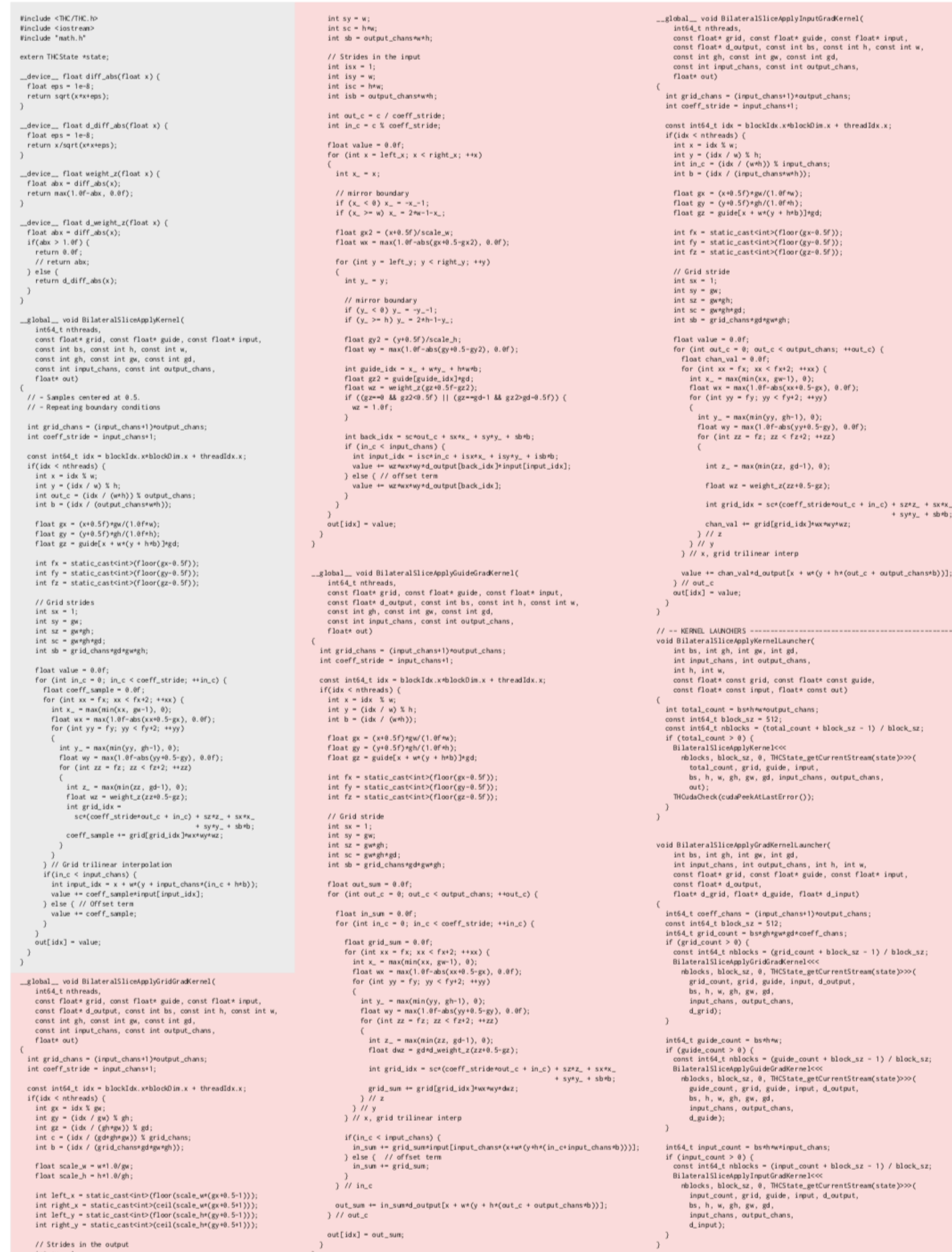
- Eager vs. lazy evaluation

- Lazy = construct entire computation graph (IR), then execute computation
  - Traditional TensorFlow/mx.Net
  - PyTorch JIT

- Eager = perform computations as NN library calls are evaluated
  - PyTorch
  - TensorFlow Eager

Concept 3

- **Challenge of writing gradients**



black = algorithm code in CUDA
red = gradient code in CUDA

[Figure credit: Li et al 18]

Value of auto-differentiation

```python
xx = Variable(th.arange(0, w).cuda().view(1, -1).repeat(h, 1))
yy = Variable(th.arange(0, h).cuda().view(-1, 1).repeat(1, w))
gx = ((xx+0.5)/w) * gw
gy = ((yy+0.5)/h) * gh
gz = th.clamp(guide, 0.0, 1.0)*gd
fx = th.clamp(th.floor(gx - 0.5), min=0)
fy = th.clamp(th.floor(gy - 0.5), min=0)
fz = th.clamp(th.floor(gz - 0.5), min=0)
wx = gx - 0.5 - fx
wy = gy - 0.5 - fy
wx = wx.unsqueeze(0).unsqueeze(0)
wy = wy.unsqueeze(0).unsqueeze(0)
wz = th.abs(gz-0.5 - fz)
wz = wz.unsqueeze(1)
fx = fx.long().unsqueeze(0).unsqueeze(0)
fy = fy.long().unsqueeze(0).unsqueeze(0)
fz = fz.long()
cx = th.clamp(fx+1, max=gw-1);
cy = th.clamp(fy+1, max=gh-1);
cz = th.clamp(fz+1, max=gd-1)
fz = fz.view(bs, 1, h, w)
cz = cz.view(bs, 1, h, w)
batch_idx = th.arange(bs).view(bs, 1, 1, 1).long().cuda()
out = []
co = c // (ci+1)
for c_ in range(co):
  c_idx = th.arange((ci+1)*c_, (ci+1)*(c_+1)).view(\
           1, ci+1, 1, 1).long().cuda()
  a = grid[batch_idx, c_idx, fz, fy, fx]*(1-wx)*(1-wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, fy, fx]*(1-wx)*(1-wy)*(  wz) + \
          grid[batch_idx, c_idx, fz, cy, fx]*(1-wx)*(  wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, cy, fx]*(1-wx)*(  wy)*(  wz) + \
          grid[batch_idx, c_idx, fz, fy, cx]*(  wx)*(1-wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, fy, cx]*(  wx)*(1-wy)*(  wz) + \
          grid[batch_idx, c_idx, fz, cy, cx]*(  wx)*(  wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, cy, cx]*(  wx)*(  wy)*(  wz)
  o = th.sum(a[:, :-1, ...]*input, 1) + a[:, -1, ...]
  out.append(o.unsqueeze(1))
out = th.cat(out, 1)

out.backward(adjoints)
d_input = input.grad
d_grid = grid.grad
d_guide = guide.grad
```

CUDA 308 lines

430 ms (1M pix)

2270 ms (4M pix)

PyTorch 42 lines

1440 ms (1M pix)

out of memory (4M pix)

red: gradient code

[See Gradient Halide for alternative]

[Figure credit: Li et al 18]

Barrage of systems/frameworks

- GLOW (FB): https://github.com/pytorch/glow
- PyTorch JIT (FB)   (compiles to XLA)
- Swift for TensorFlow / DLVM (UIUC project, embedded in Swift, adds Autodiff)
- Flux (library in Julia built on top of Julia AutoDiff, compiles to TPU via XLA) (https://github.com/FluxML/Flux.jl)
- Google XLA (large tensor ops, some basic fusion of ops)
- TVM (Halide-like, has auto scheduling of basic tensor ops)
- …
- Facebook Tensor Comprehensions (Polyhedral, emits Halide schedules for codegen)
- ONNX (https://github.com/onnx/onnx), framework for graph definition and extensible optimization passes
  - Halide implementation of most ONNX ops "exists"
- Gradient Halide (adds reverse-mode Autodiff to Halide)

VR hardware

VR headsets

Oculus Rift

HTC Vive

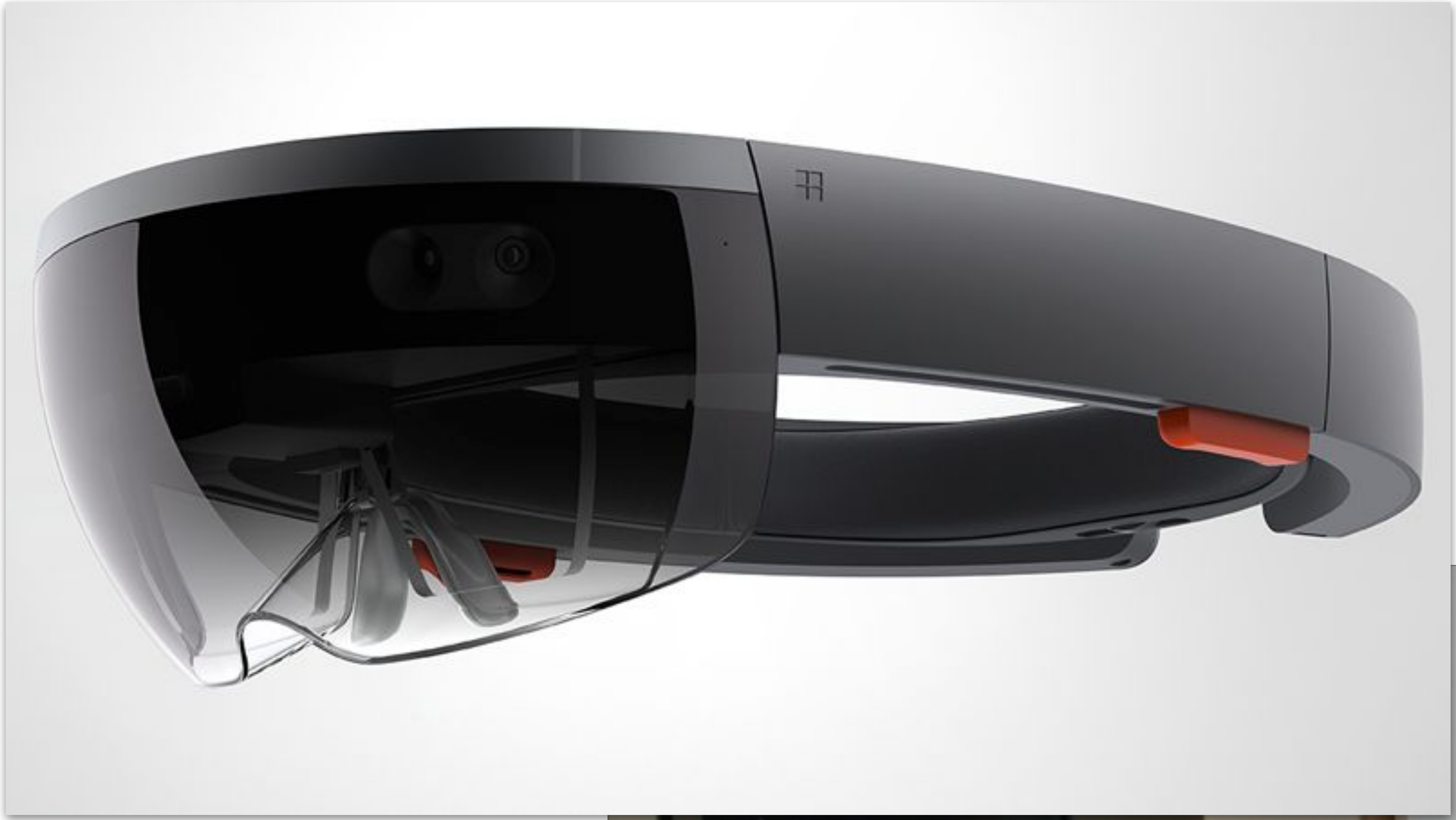
Sony Morpheus

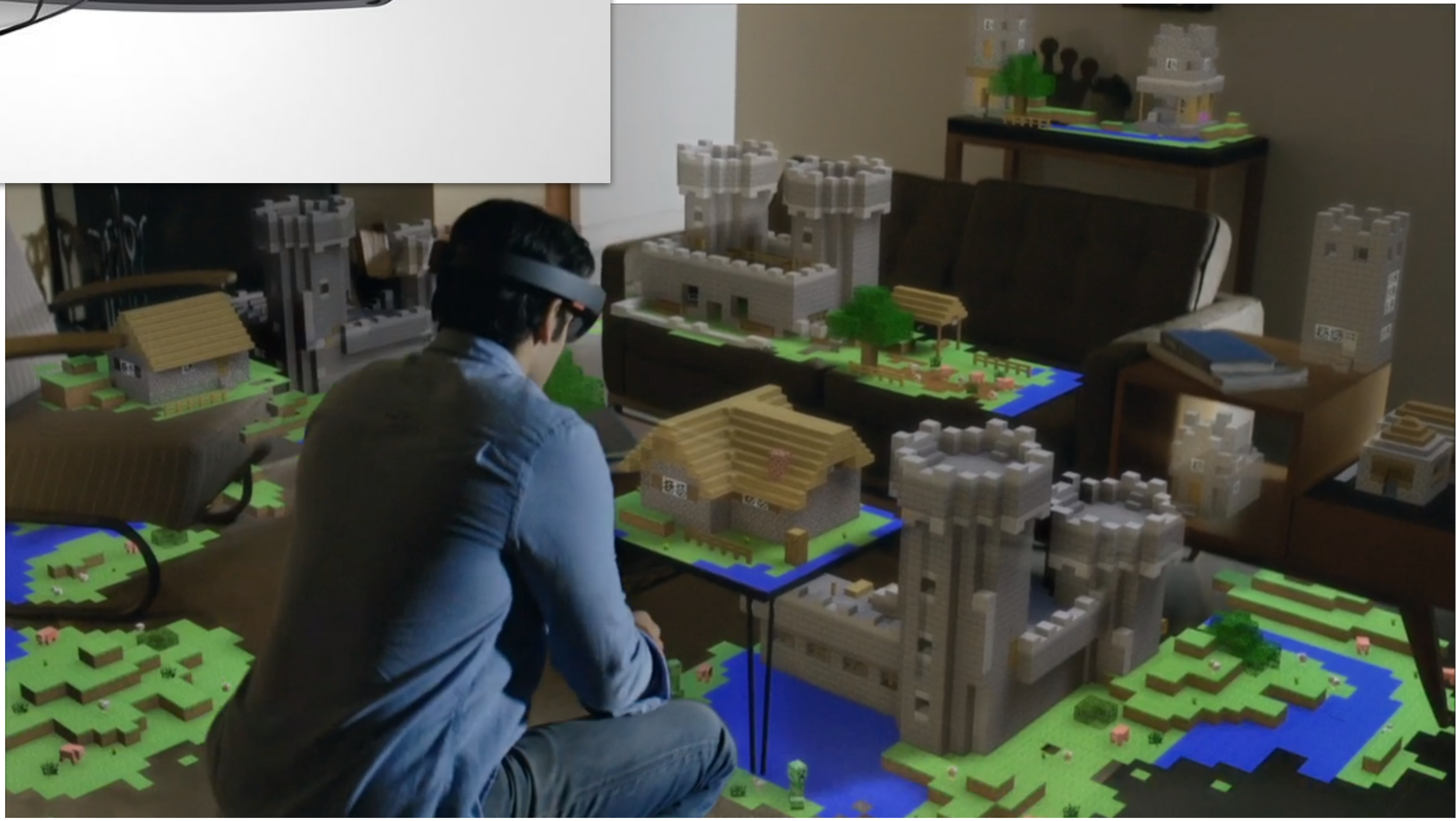
Oculus Go

Google Daydream

Google Cardboard

AR headset: Microsoft Hololens





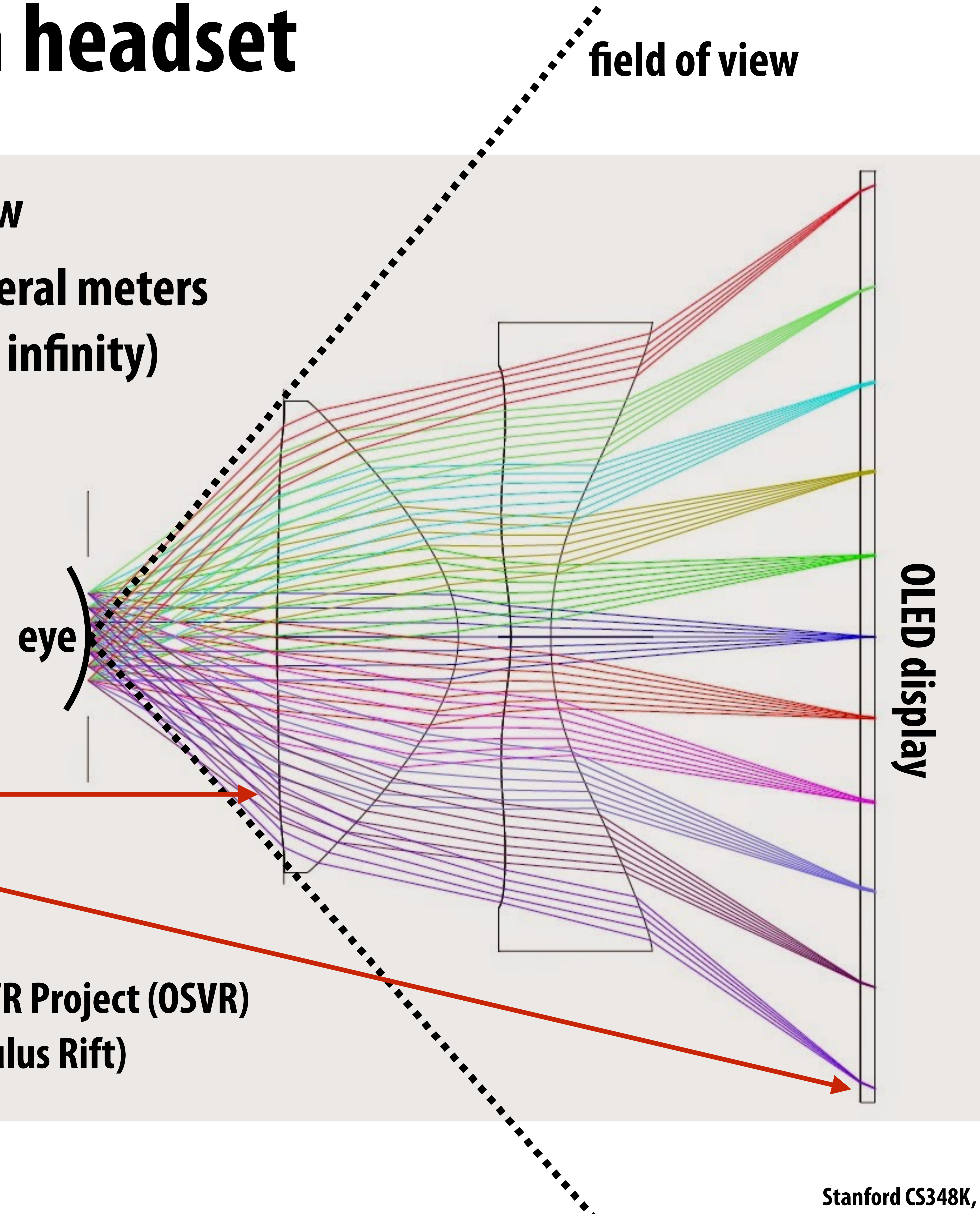
Oculus Rift CV1 headset

Image credit: ifixit.com

Role of optics in headset

1. Create wide field of view
2. Place focal plane at several meters away from eye (close to infinity)

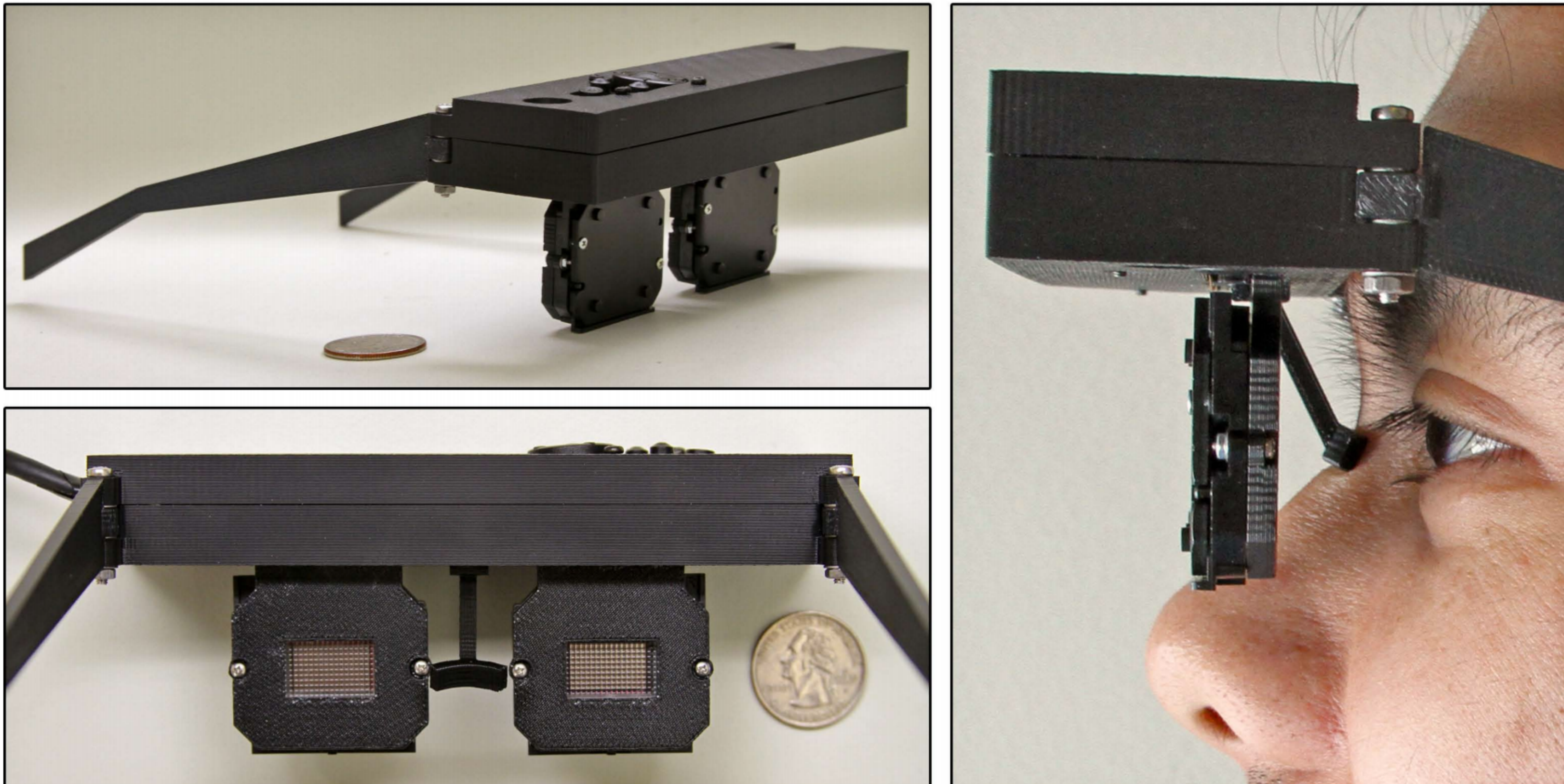
Note: parallel lines reaching eye converge to a single point on display (eye accommodates to plane near infinity)



Lens diagram from Open Source VR Project (OSVR)
(Not the lens system from the Oculus Rift)
http://www.osvr.org/

Aside: near-eye "light field" displays

Attempt to recreate same magnitude and direction of rays of light as produced by being in a real world scene.



Name of the game, part 1: low latency

- The goal of a VR graphics system is to achieve “presence”, tricking the brain into thinking what it is seeing is real

- Achieving presence requires an exceptionally low-latency system
  - What you see must change when you move your head!
  - End-to-end latency: time from moving your head to the time new photons hit your eyes
    - Measure user’s head movement
    - Update scene/camera position
    - Render new image
    - Transfer image to headset, then to transfer to display in headset
    - Actually emit light from display (photons hit user’s eyes)
  - Latency goal of VR: 10-25 ms
    - Requires exceptionally low-latency head tracking
    - Requires exceptionally low-latency rendering and display

Thought experiment: effect of latency

- Consider a 1,000 x 1,000 display spanning 100° field of view
  - 10 pixels per degree

- Assume:
  - You move your head 90° in 1 second (only modest speed)
  - End-to-end latency of graphics system is 33 ms (1/30 sec)

- Therefore:
  - Displayed pixels are off by 3° ~ 30 pixels from where they would be in an ideal system with 0 latency

Example credit: Michael Abrash

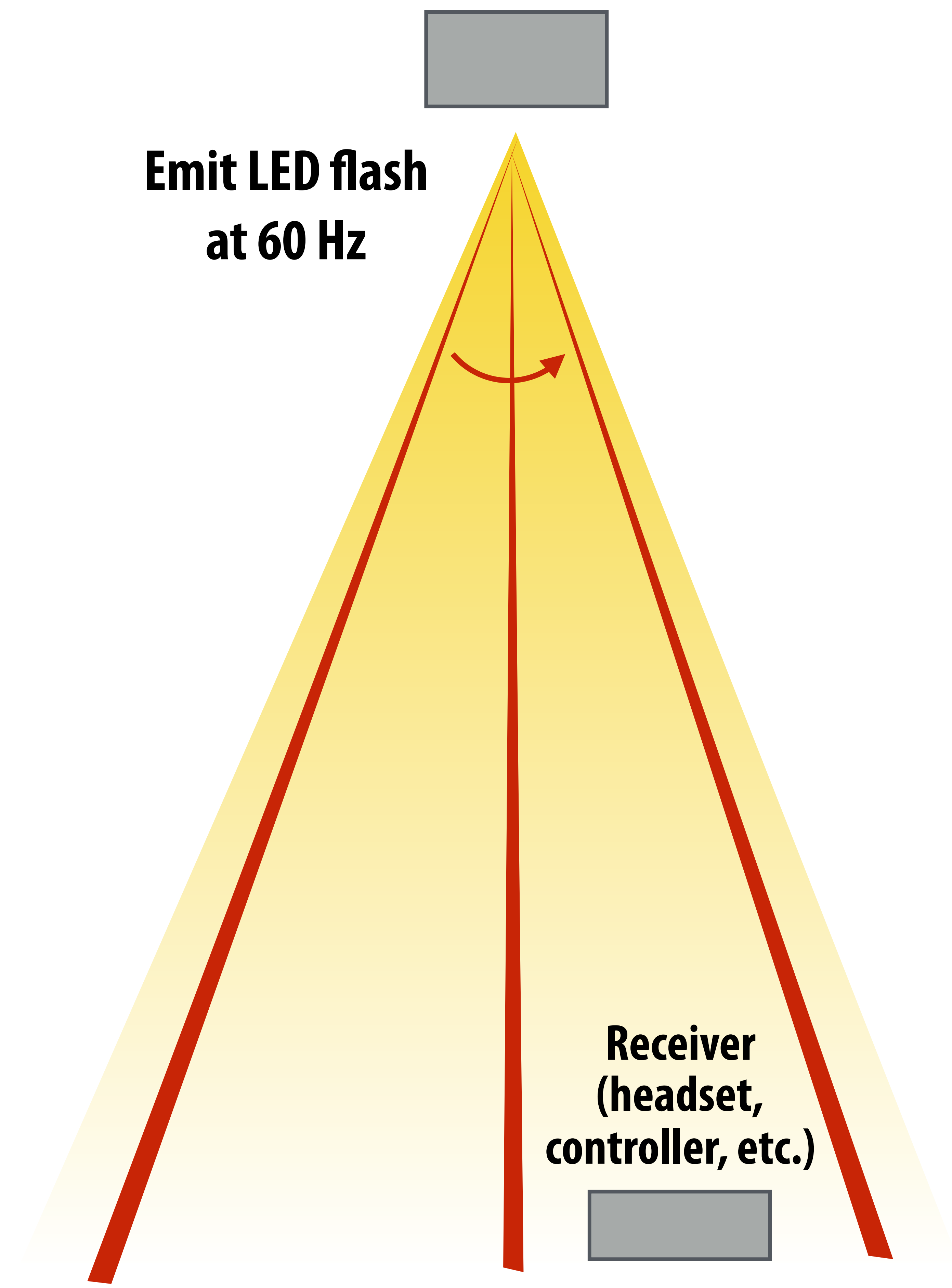
Oculus CV1 IR camera and IR LEDs

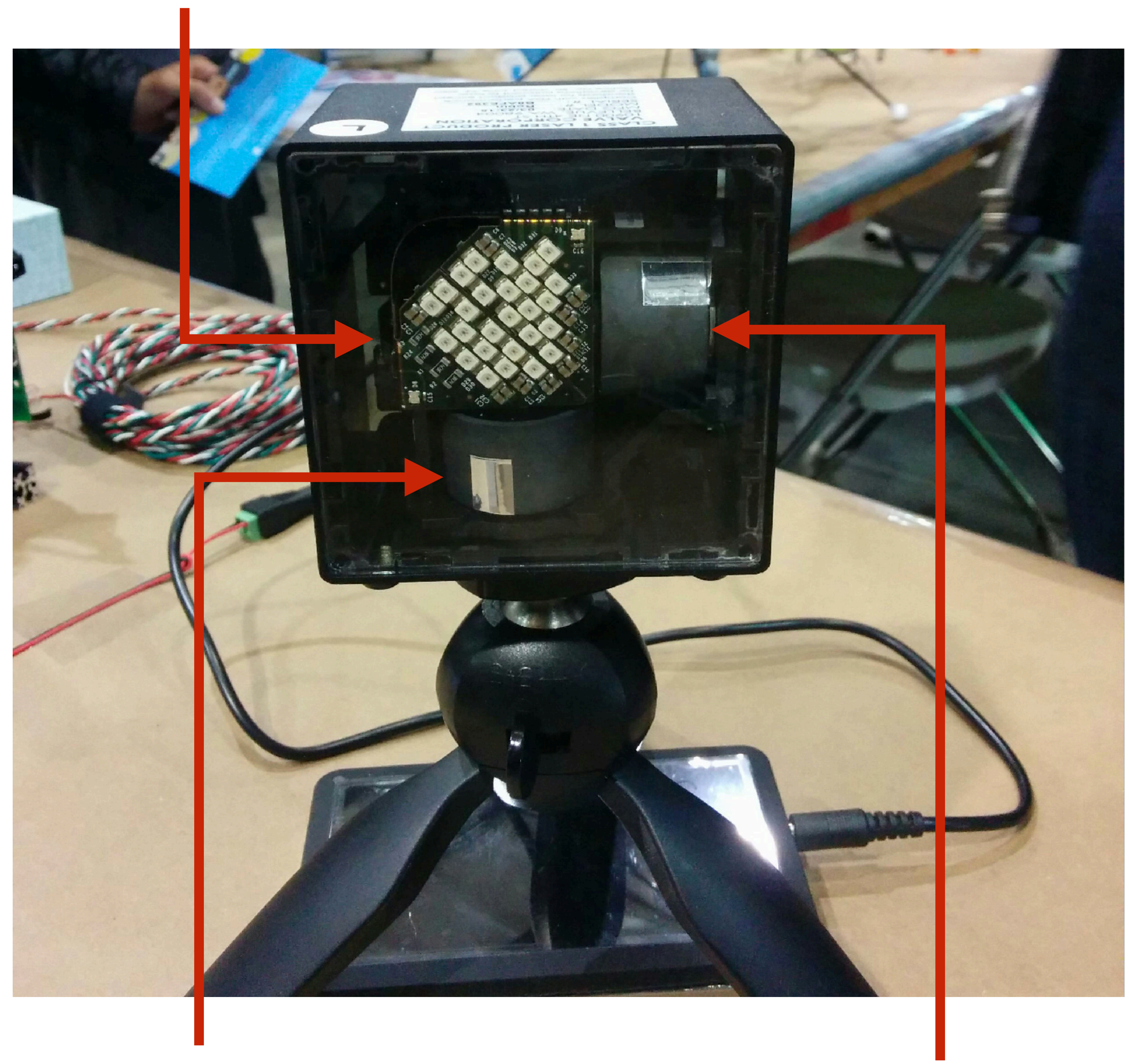
Headset contains:
IR LEDs (tracked by camera)
Gyro + accelerometer (1000Hz). (rapid relative positioning)

60Hz IR Camera
(measures absolute position
of headset 60 times a second)

Image credit: ifixit.com

Valve's Lighthouse: cameraless position tracking



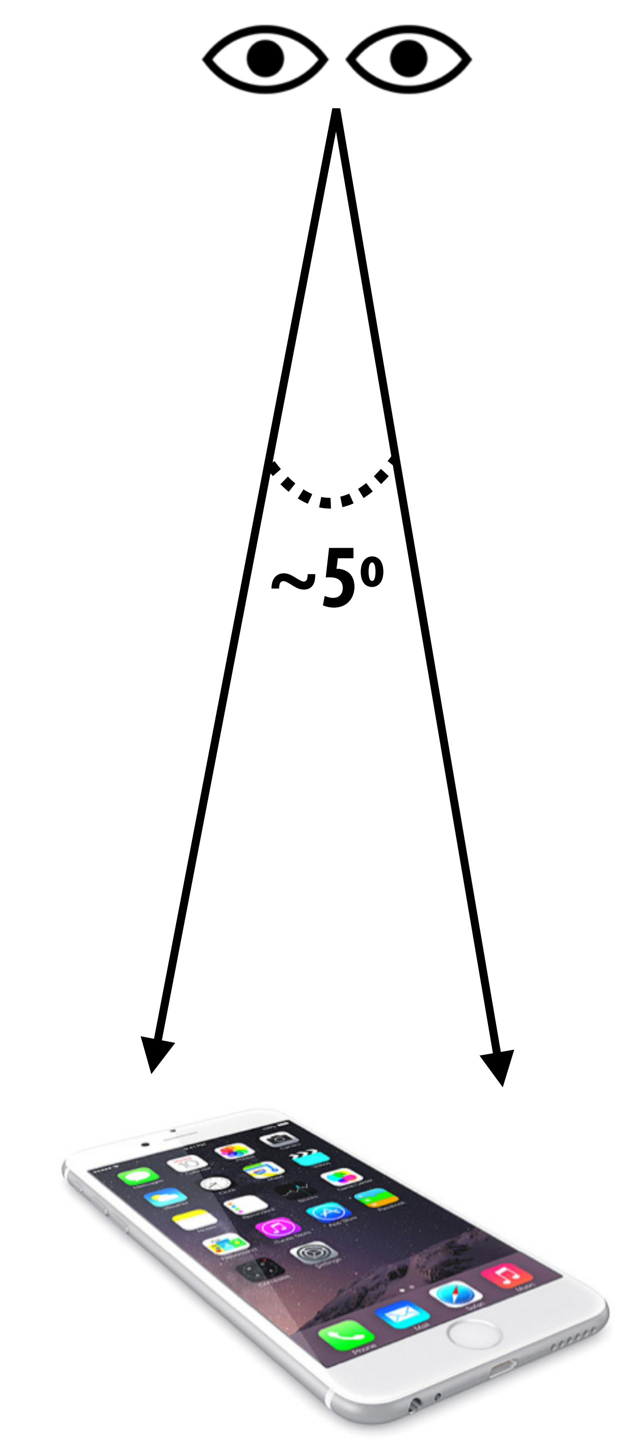


No need for computer vision processing to compute position of receiver: just a light sensor and an accurate clock!

Image credit: Travis Deyle
http://www.hizook.com/blog/2015/05/17/valves-lighthouse-tracking-system-may-be-big-news-robotics

Accounting for resolution of eye

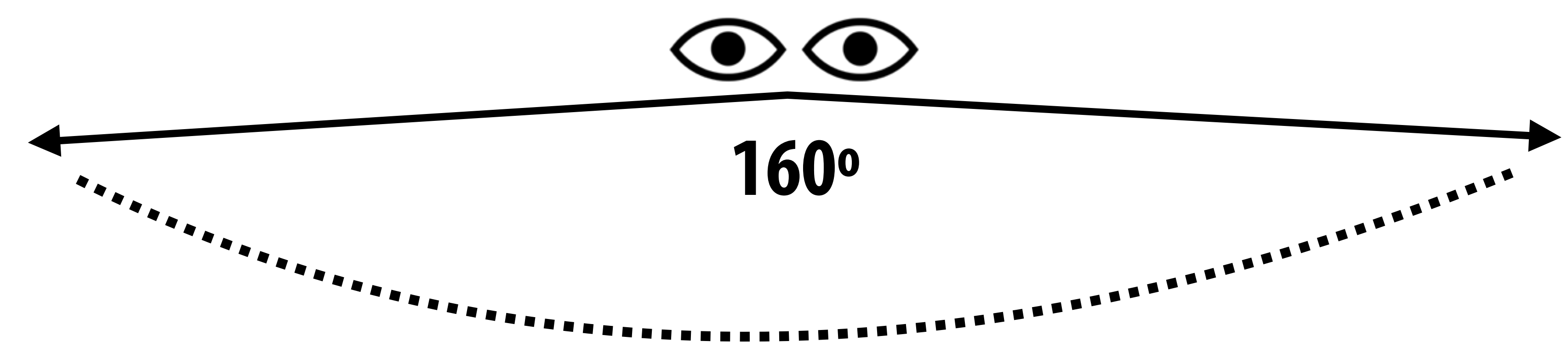
Name of the game, part 2: high resolution



iPhone 6: 4.7 in "retina" display:

1.3 MPixel

326 ppi → 57 ppd



Human: ~160° view of field per eye (~200° overall)

(Note: this does not account for eye's ability to rotate in socket)

Future "retina" VR display:

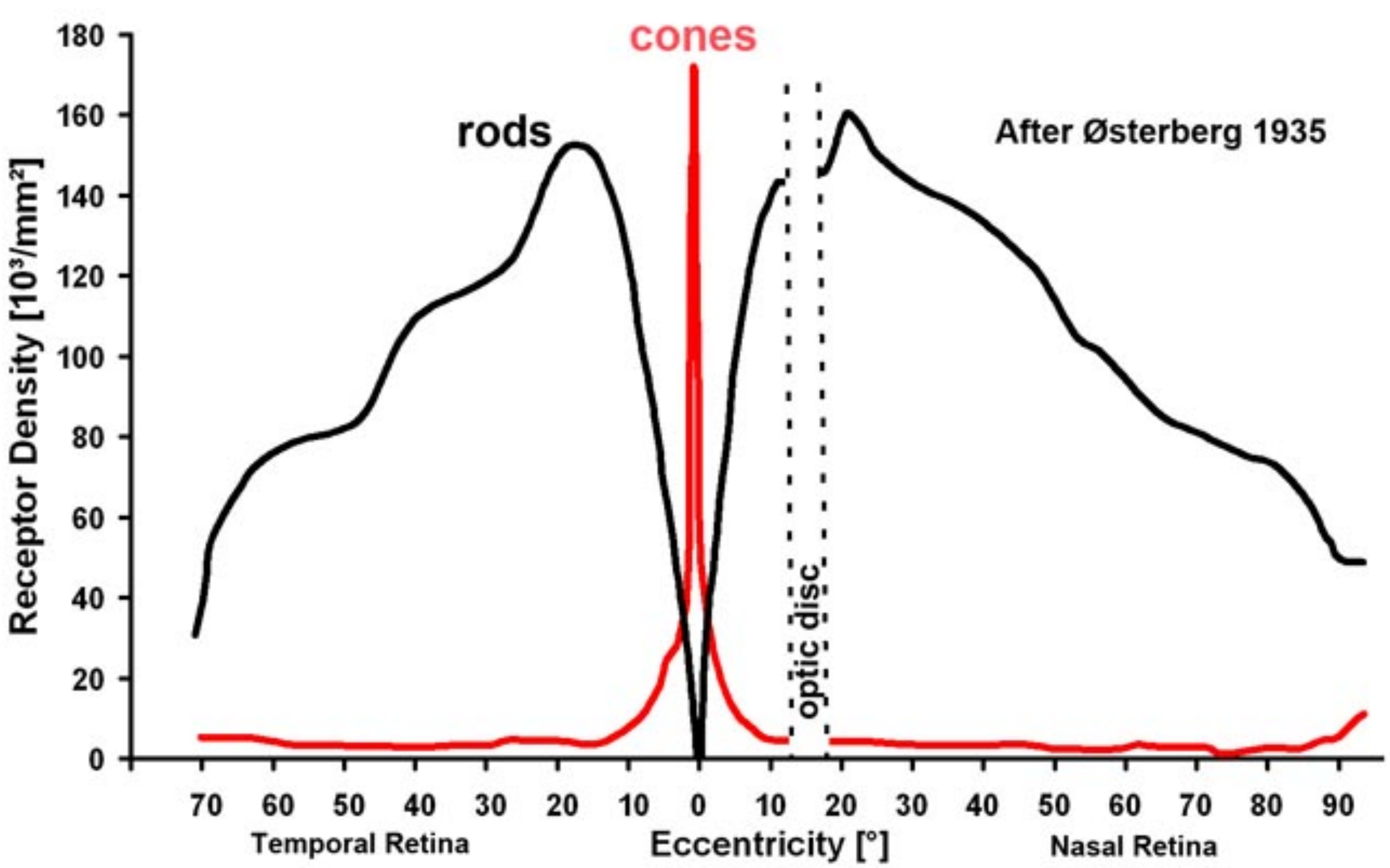
57 ppd covering 200°

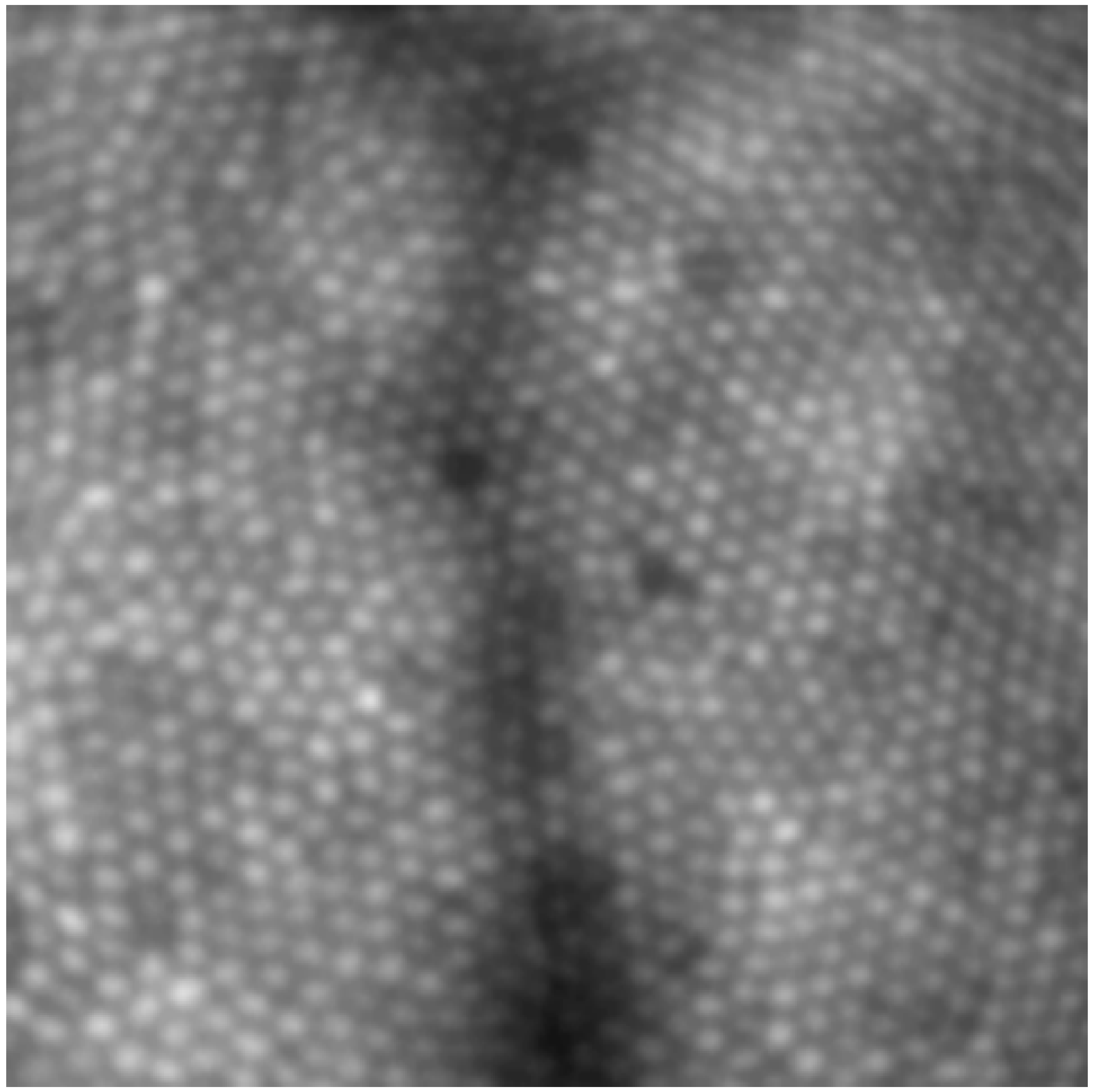
= 11K x 11K display per eye

= 220 MPixel

Strongly suggests need for eye tracking and foveated rendering (eye can only perceive detail in 5° region about gaze point

Density of rod and cone cells in the retina



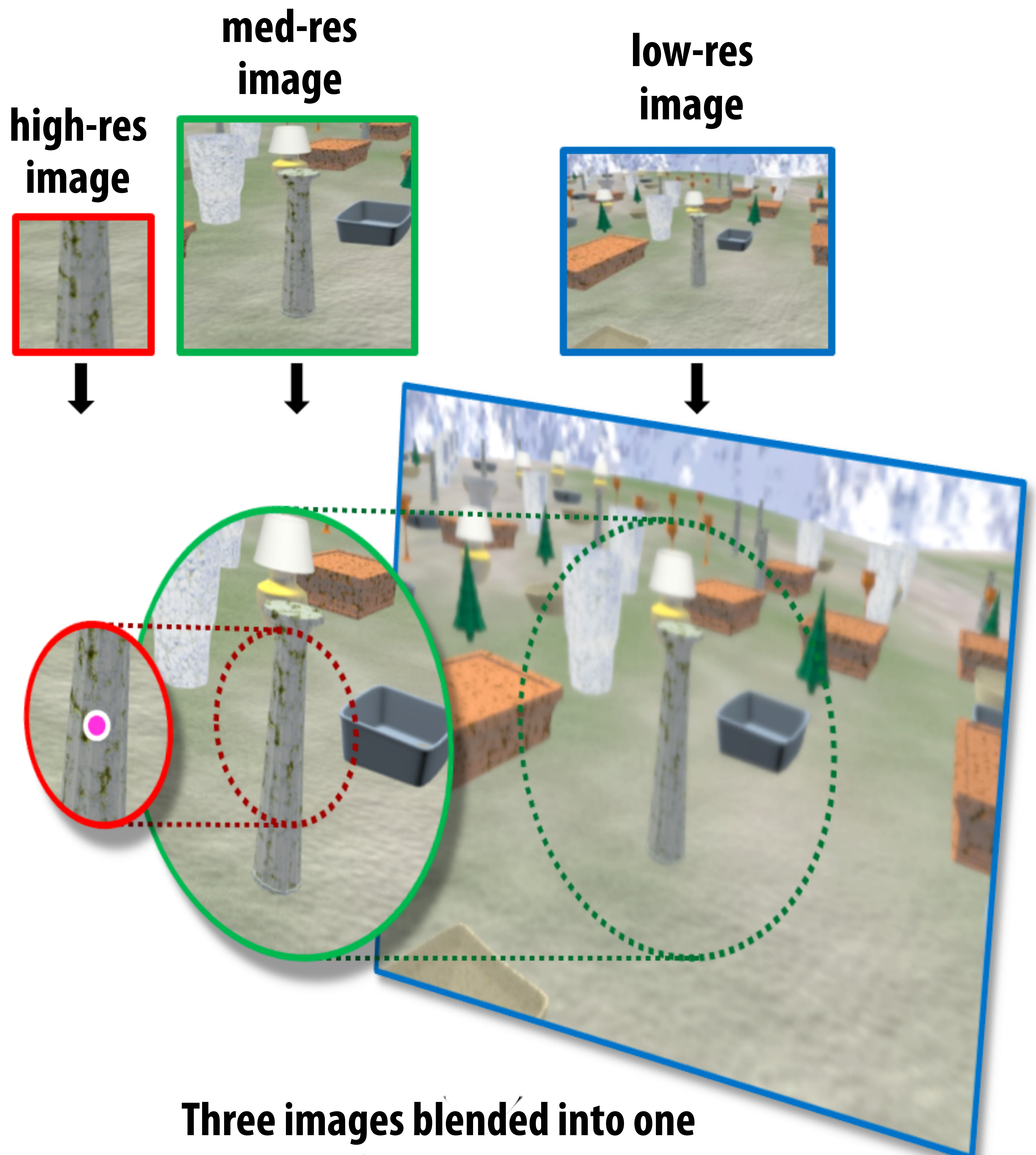


[Roorda 1999]

- Cones are color receptive cells
- Highest density of cones is in fovea
  (best color vision at center of where human is looking)

Addressing high resolution and high field of view: foveated rendering

Idea: track user's gaze, render with increasingly lower resolution farther away from gaze point



Three images blended into one for display

Traditional rendering (uniform screen sampling)

[Patney et al. 2016]

Low-pass filter away from fovea

In this image, gaussian blur with radius dependent on distance from fovea is used to remove high frequencies

[Patney et al. 2016]

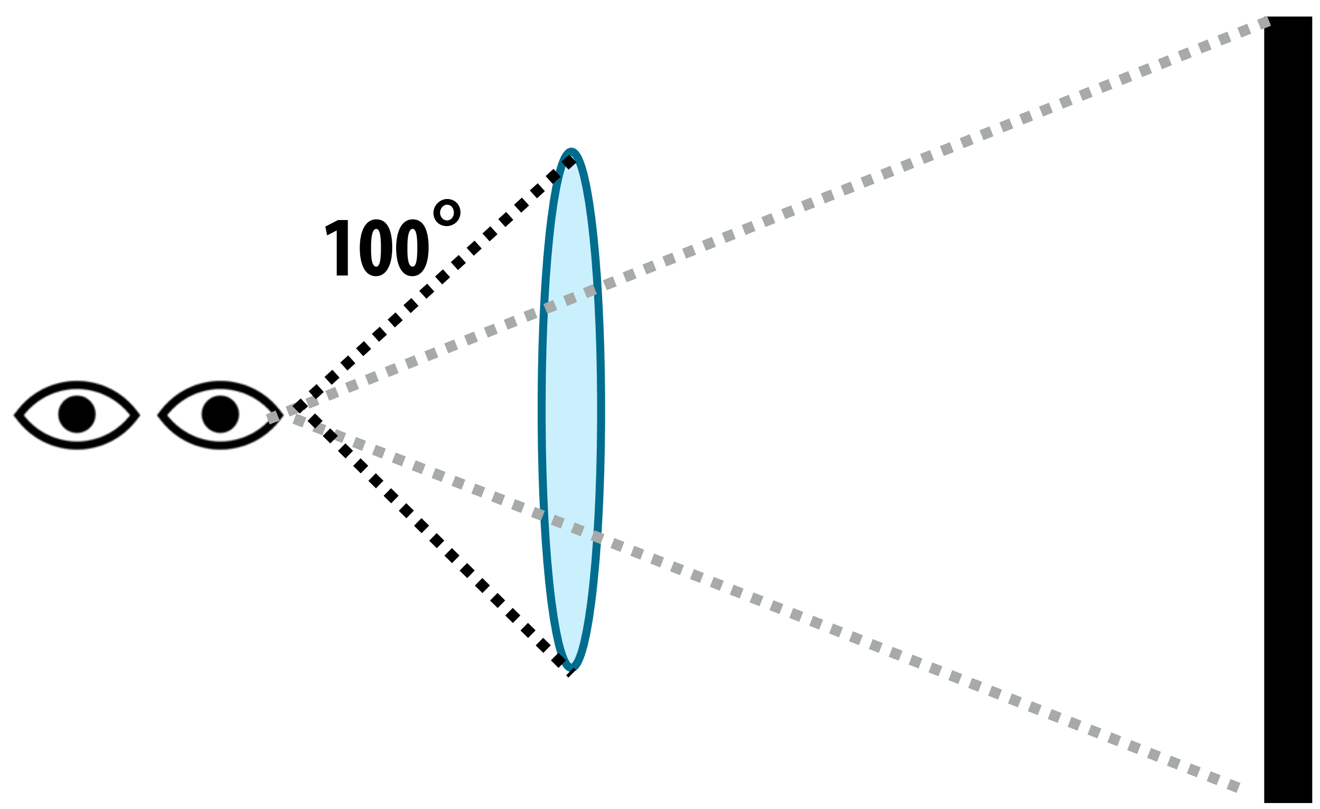
Contrast enhance periphery

Eye is receptive to contrast at periphery

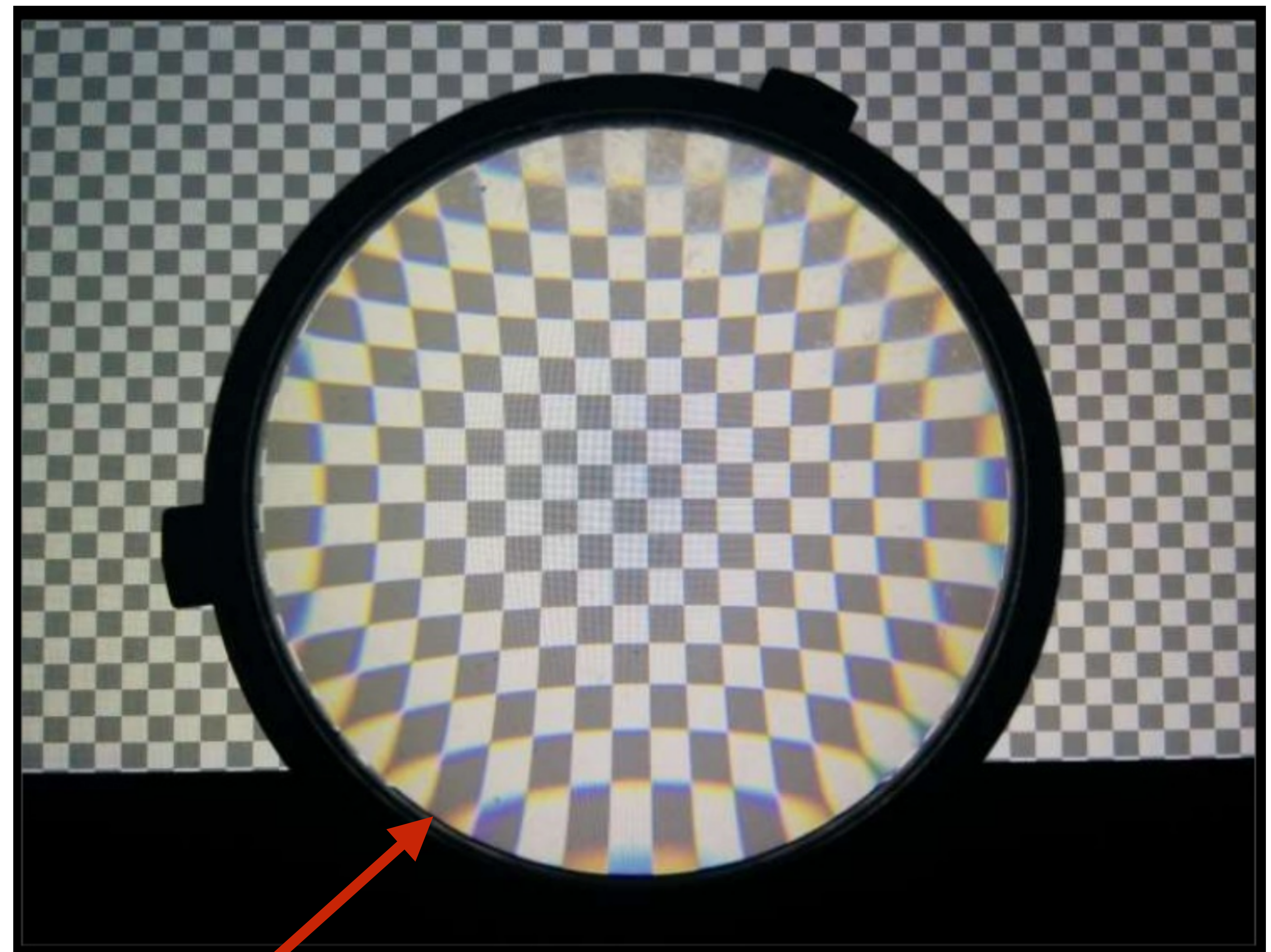
[Patney et al. 2016]

Accounting for distortion due to design of head-mounted display

Requirement: wide field of view



View of checkerboard through Oculus Rift lens

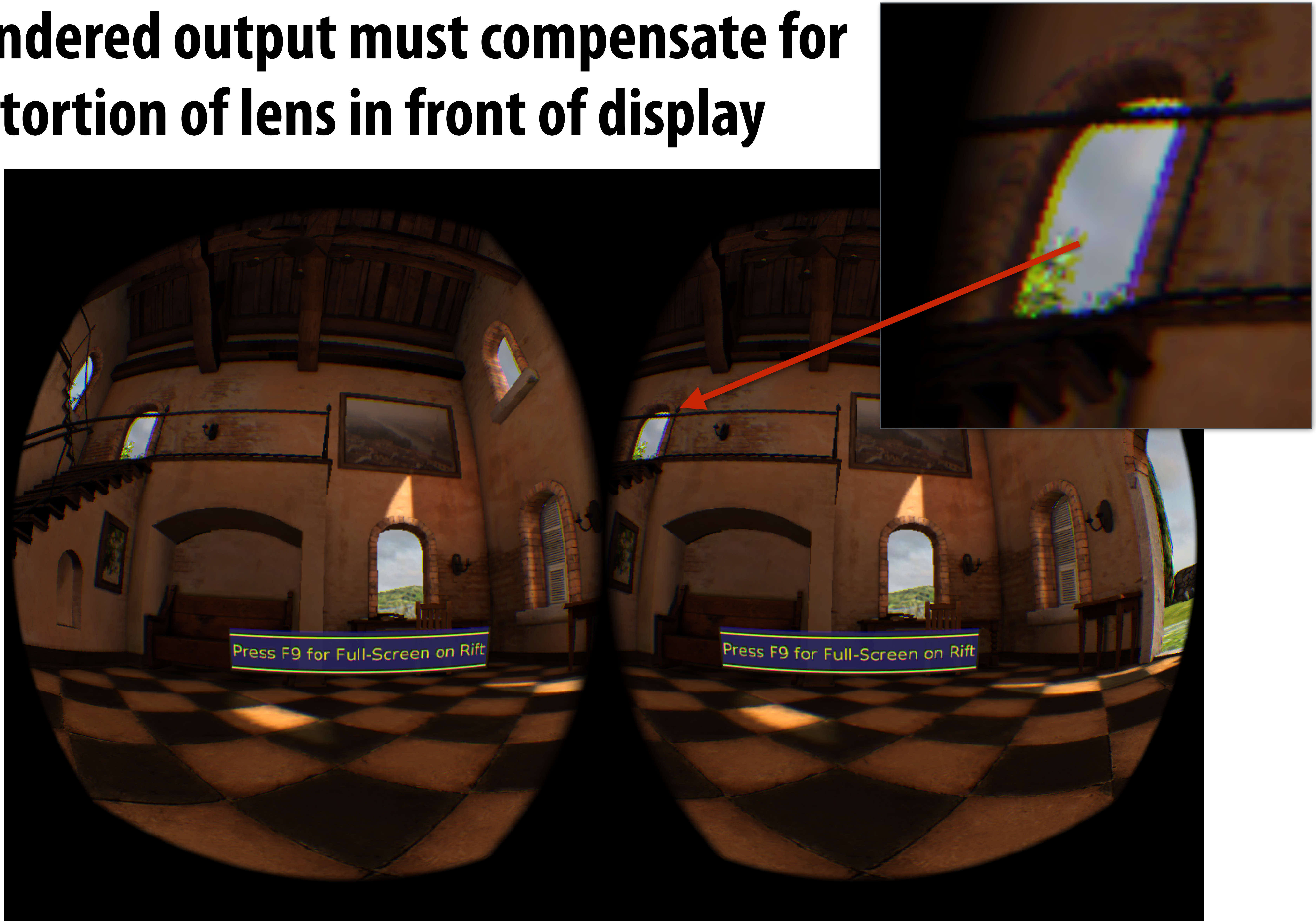


Lens introduces distortion

- Pincushion distortion
- Chromatic aberration (different wavelengths of light refract by different amount)

Icon credit: Eyes designed by SuperAtic LABS from the thenounproject.com
Image credit: Cass Everitt

Rendered output must compensate for distortion of lens in front of display



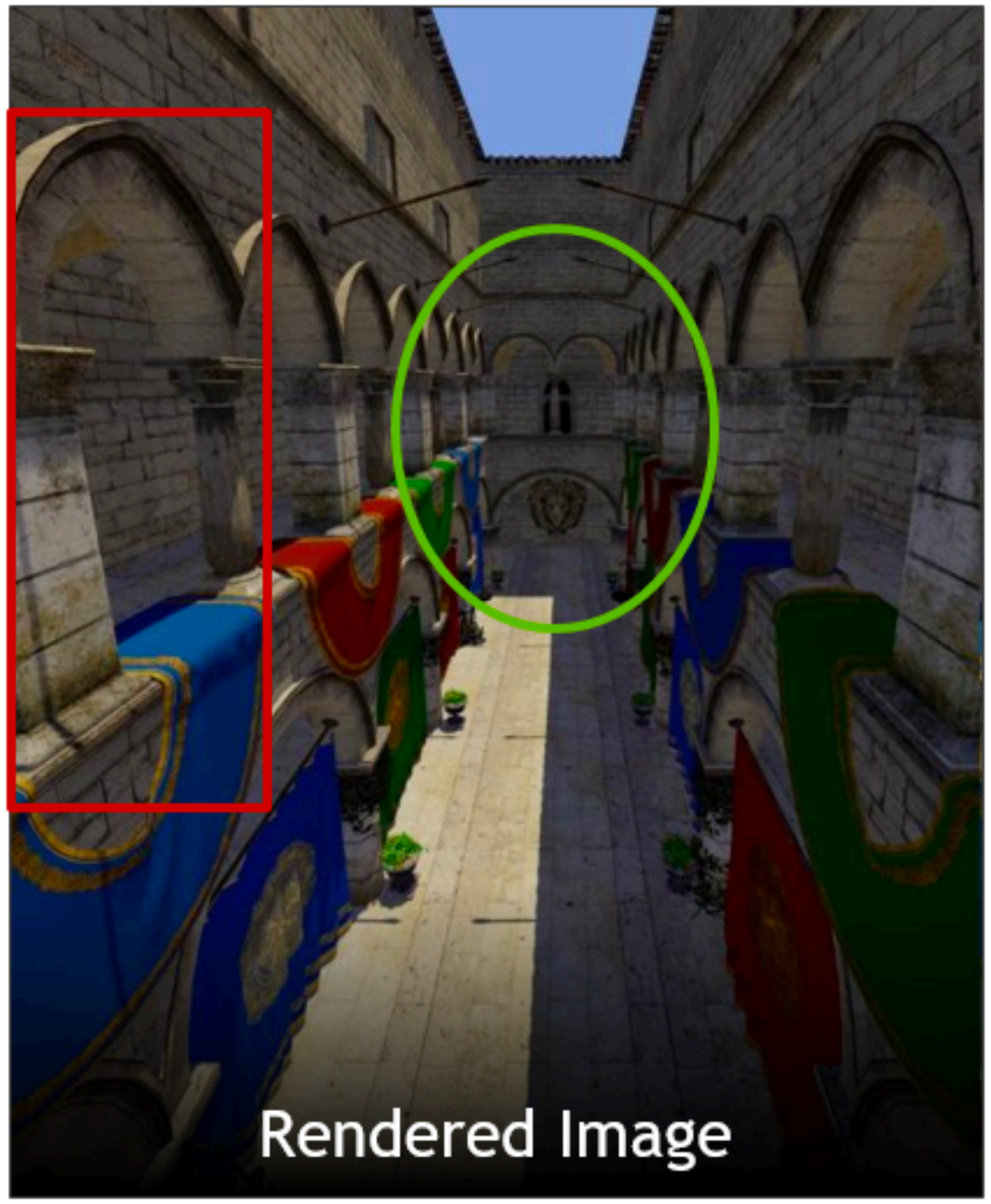
Step 1: render scene using traditional graphics pipeline at full resolution for each eye

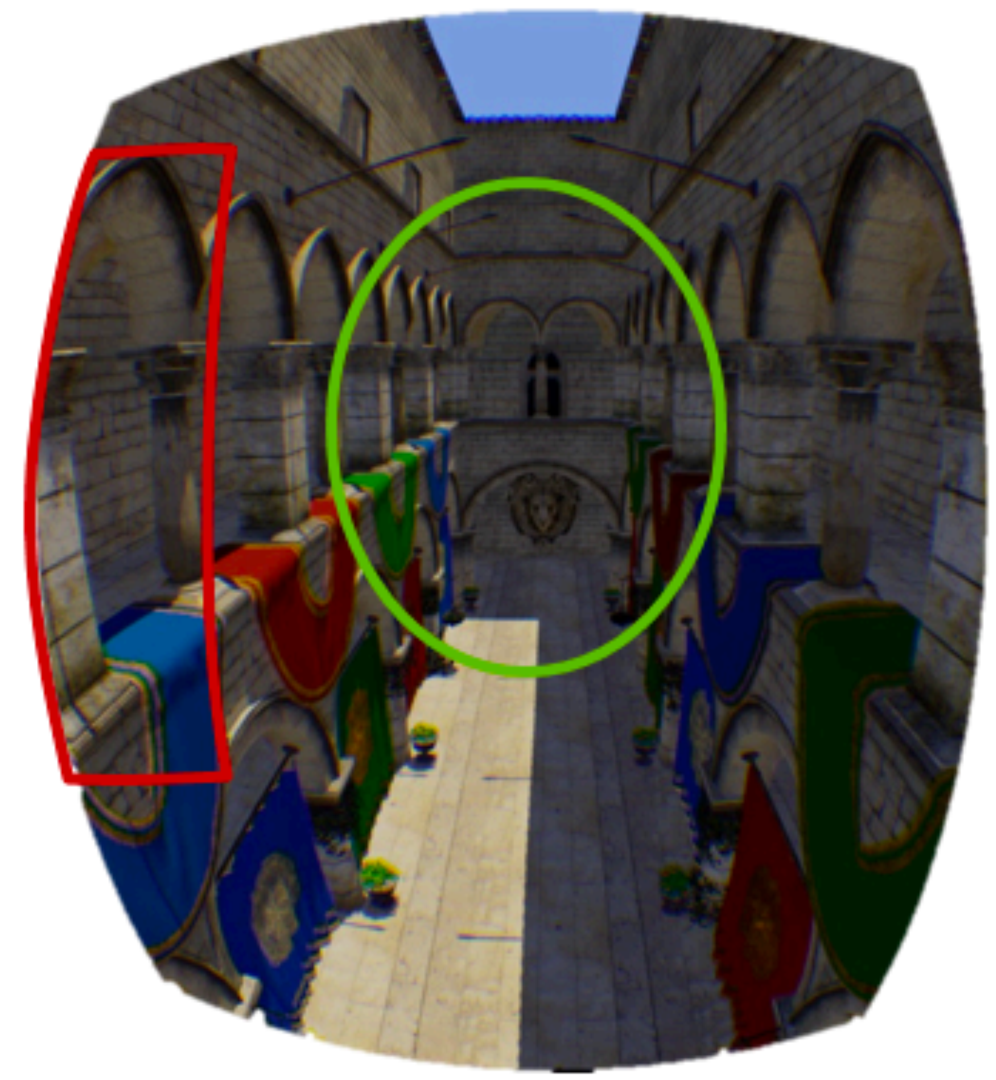
Step 2: warp images and composite into frame so rendering is viewed correctly after lens distortion

(Can apply unique distortion to R, G, B to approximate correction for chromatic aberration)

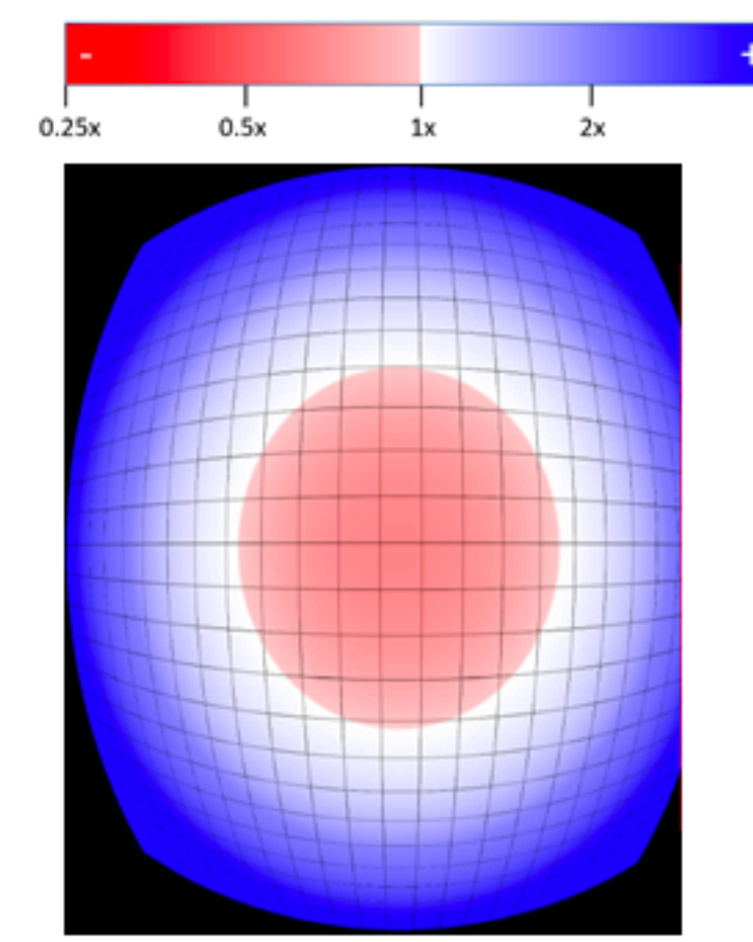
Image credit: Oculus VR developer guide

Problem: oversampling at periphery





Warped Image



Shading Rate After Lens Warp

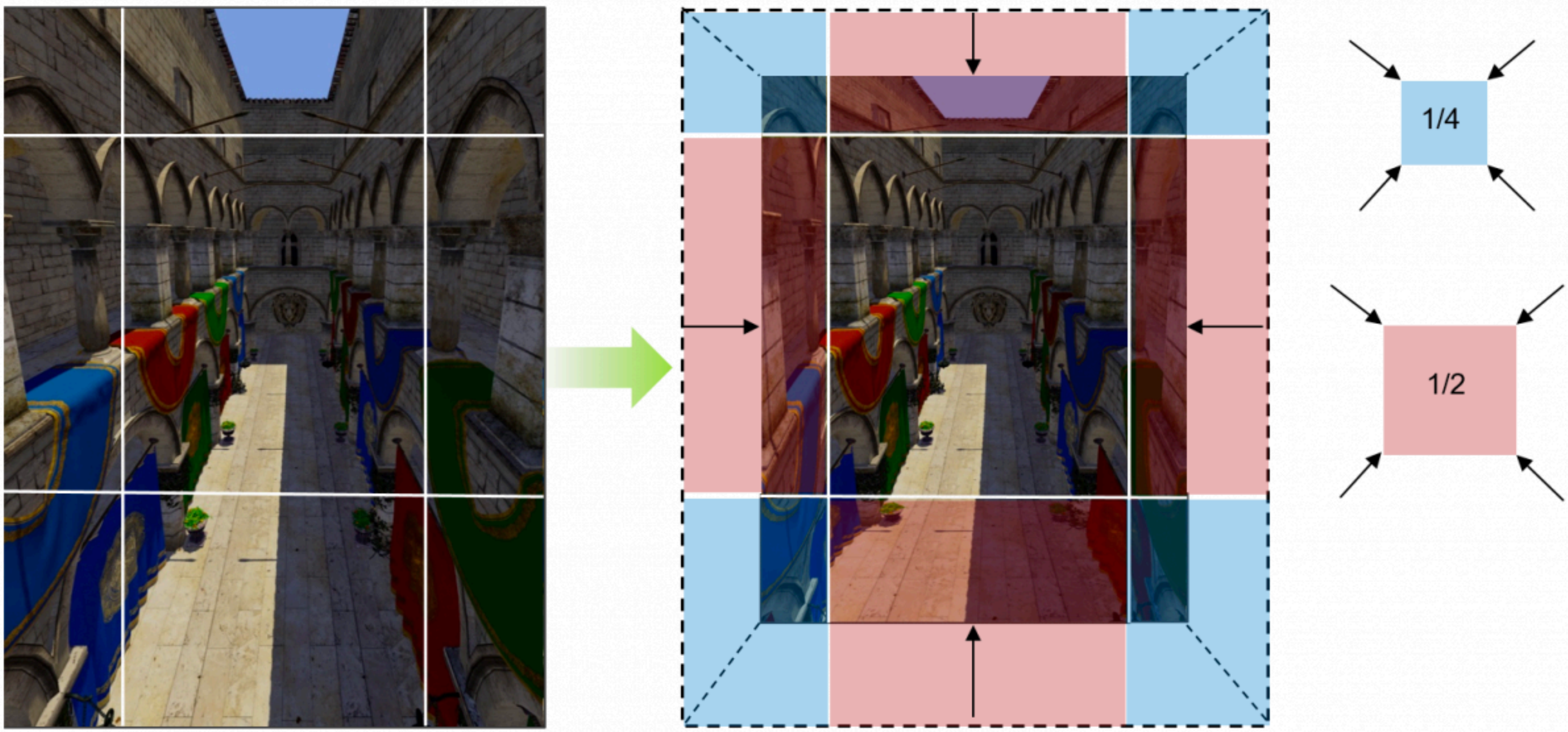
Due to:

Warp to reduce optical distortion (sample shading densely in the periphery)

Also recall eye has less spatial resolution in periphery (assuming viewer's gaze is toward center of screen)

[Image credit: NVIDIA]

Multi viewport rendering



Render the scene once, but graphics pipeline using different sampling rates for different regions ("viewports")

[Image credit: NVIDIA]

Lens matched shading

- Render with four viewports
- "Compresses" scene in the periphery (fewer samples), while not affecting scene near center of field of view

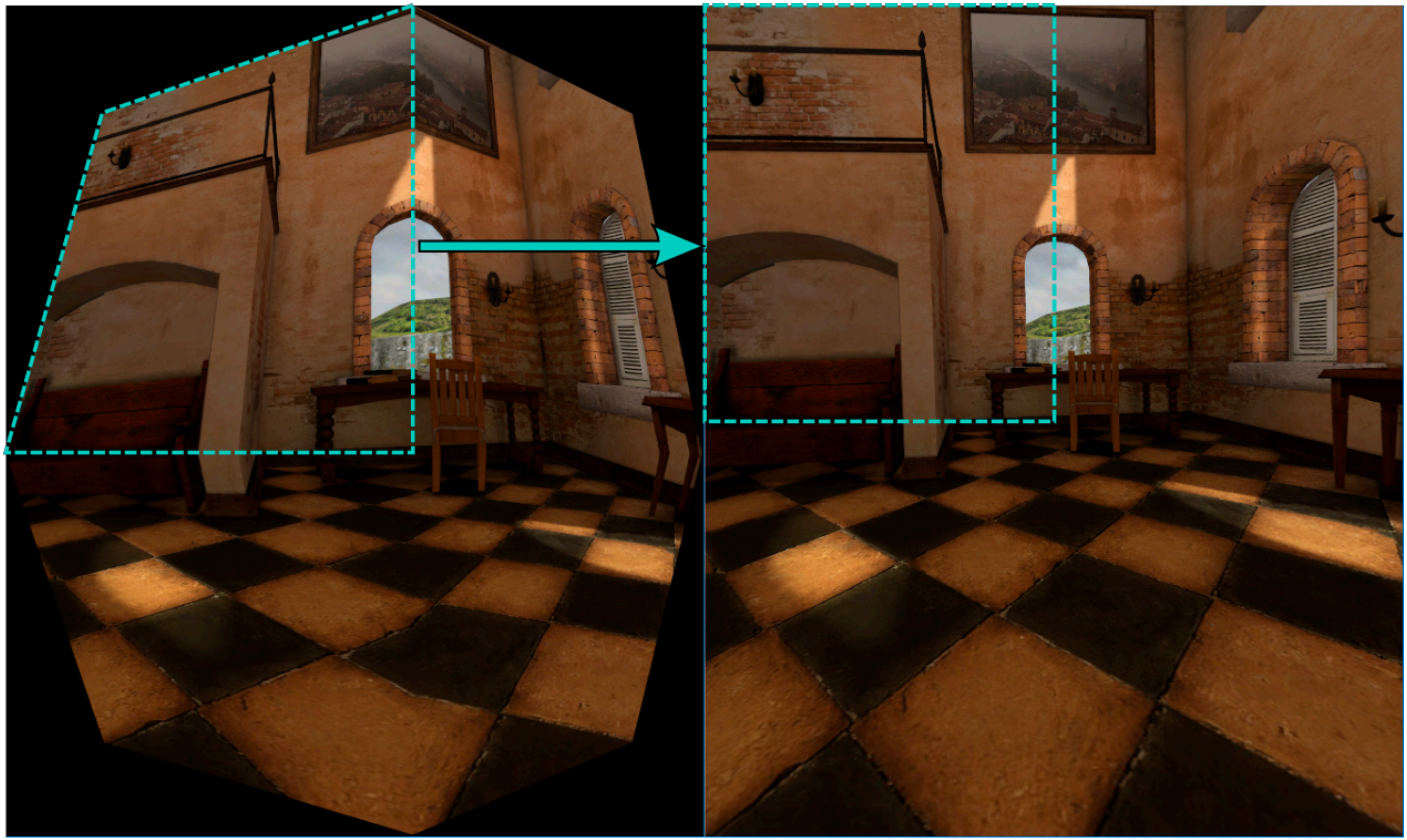
Original Viewport

Enlarged Viewport
Shading Rate Increased

With Modifed W
Periphery Shading Reduced
Center Shading Rate Still Increased
Overall Shading Reduced

[Image credit: NVIDIA]

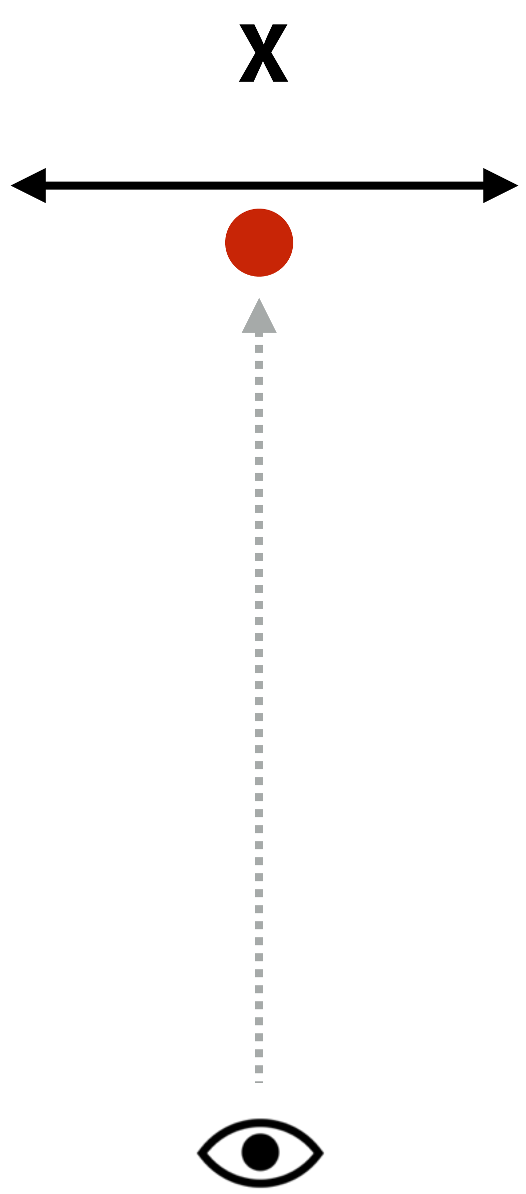
Lens matched shading

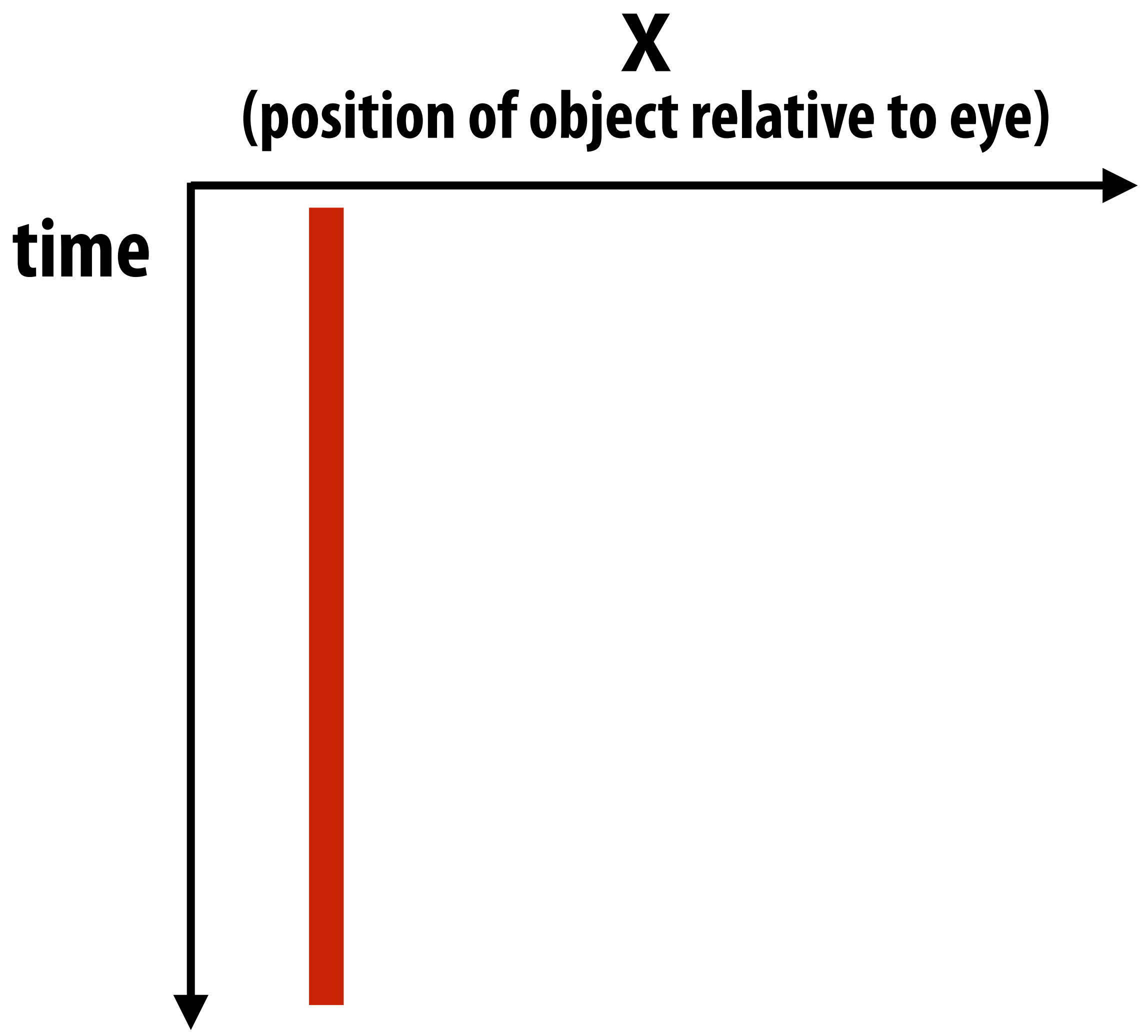


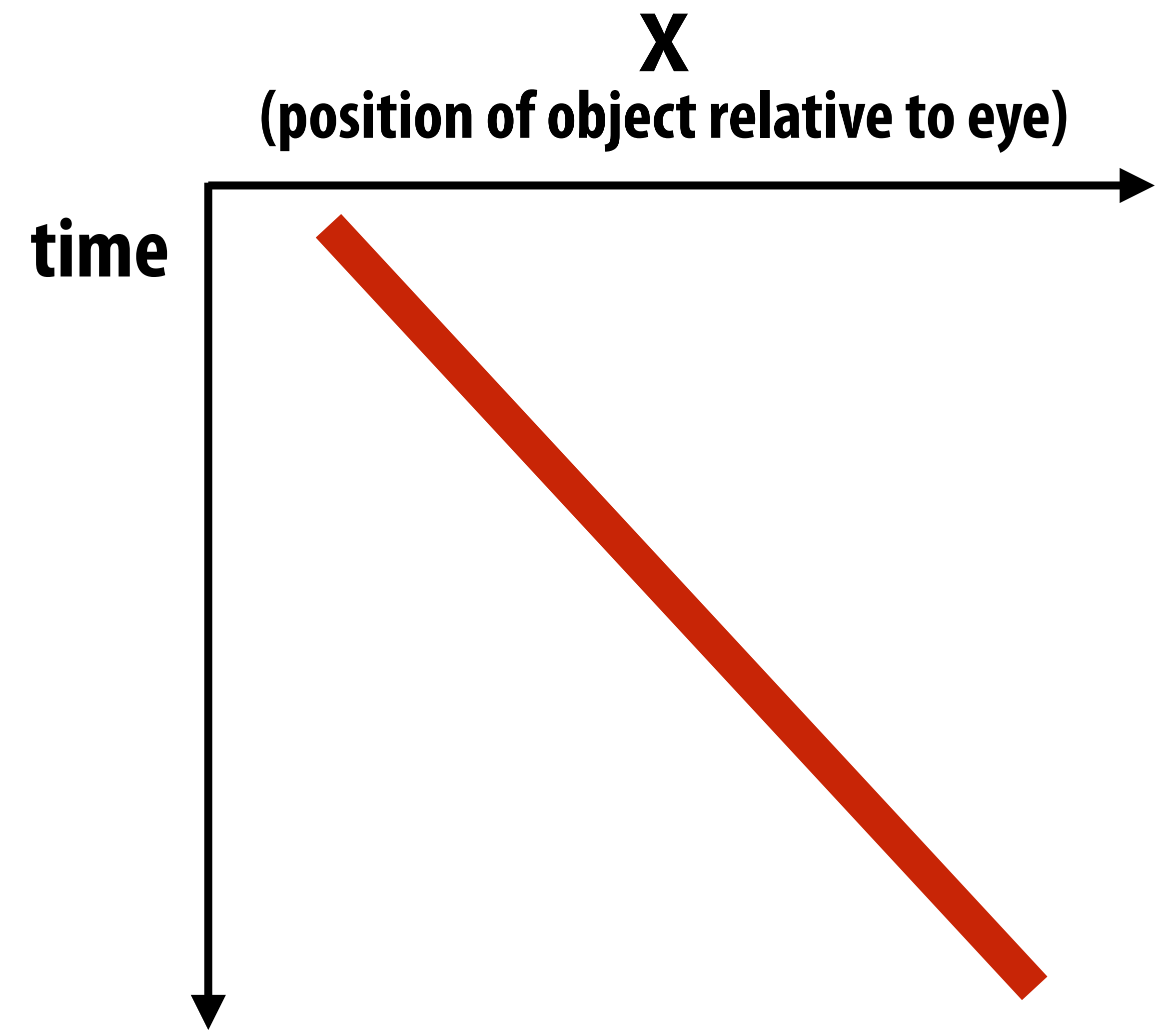
[Image credit: Oculus]

Accounting for interaction of display update + display attached to head

Consider object position relative to eye







Case 1: object stationary relative to eye:

(eye still and red object still
OR
red object moving left-to-right and
eye moving to track object
OR
red object stationary in world but head moving
and eye moving to track object)

Case 2: object moving relative to eye:

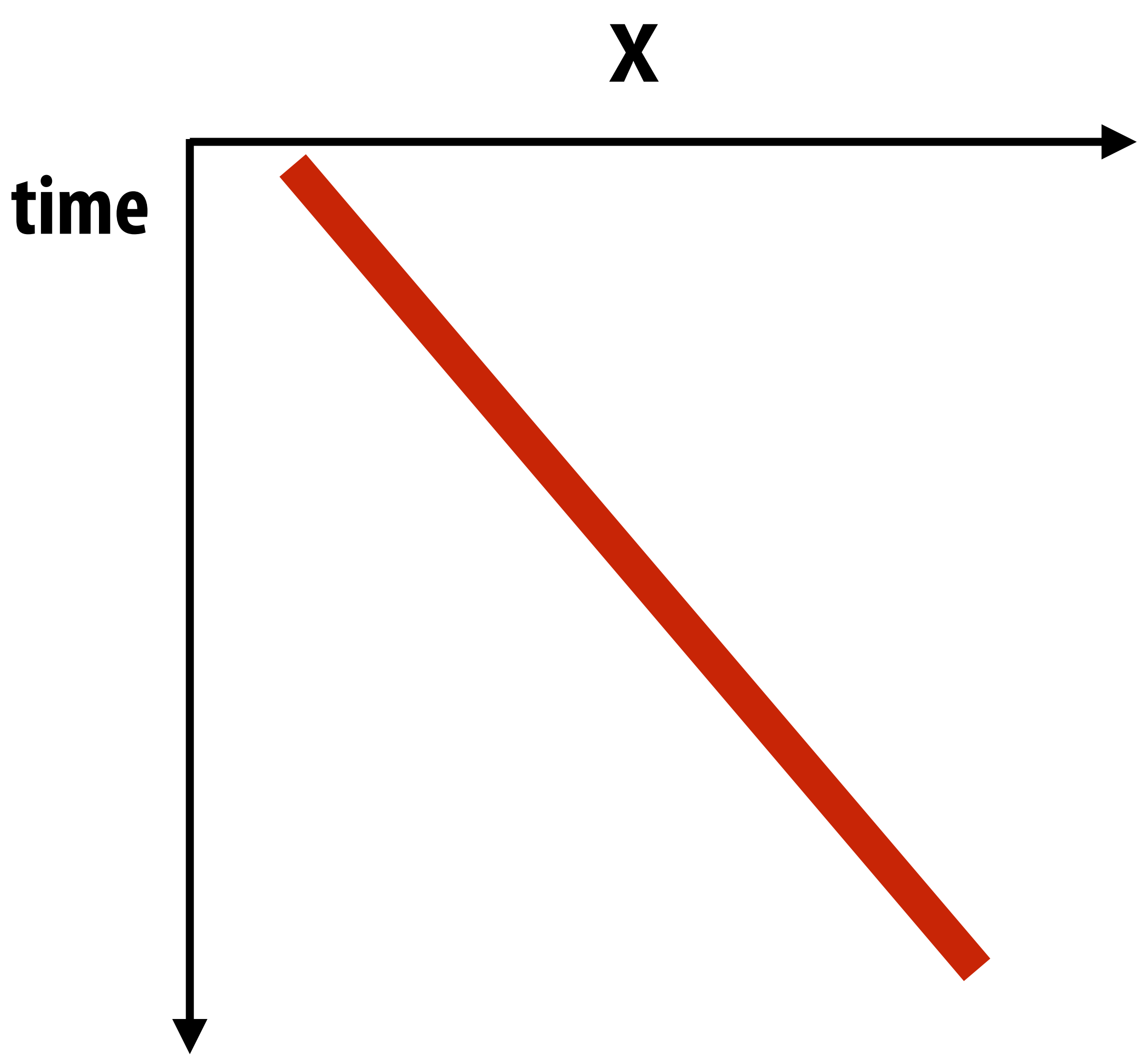
(red object moving from left to right but
eye stationary, i.e., it's focused on a different
stationary point in world)

NOTE: THESE GRAPHS PLOT <u>OBJECT POSITION</u> RELATIVE TO EYE

RAPID HEAD MOTION WITH EYES TRACK A MOVING OBJECT IS A FORM OF CASE 1!!!

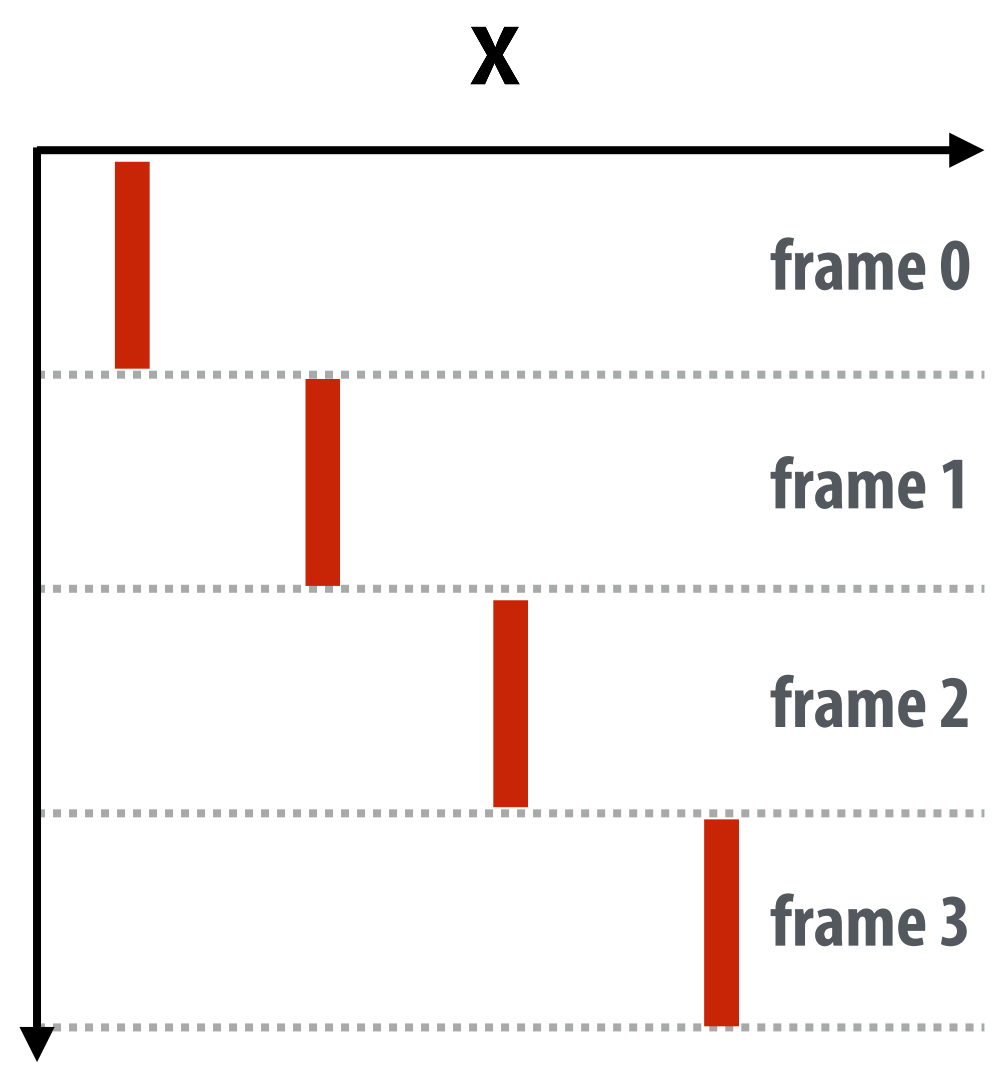
Spacetime diagrams adopted from presentations by Michael Abrash
Eyes designed by SuperAtic LABS from the thenounproject.com

Effect of latency: judder



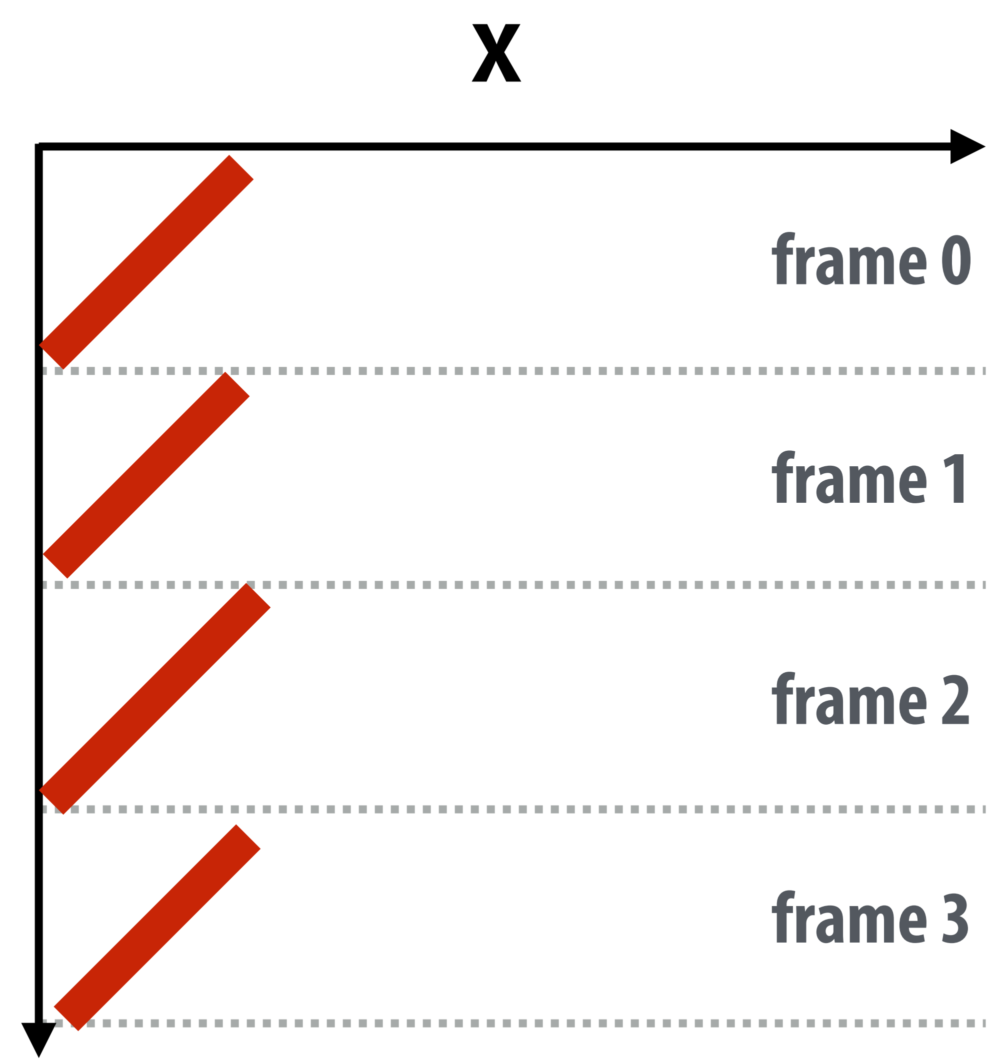
Case 2: object moving from left to right, eye stationary
(<u>eye stationary with respect to display</u>)

Continuous representation.



Case 2: object moving from left to right, eye stationary
(<u>eye stationary with respect to display</u>)

Light from display
(image is updated each frame)



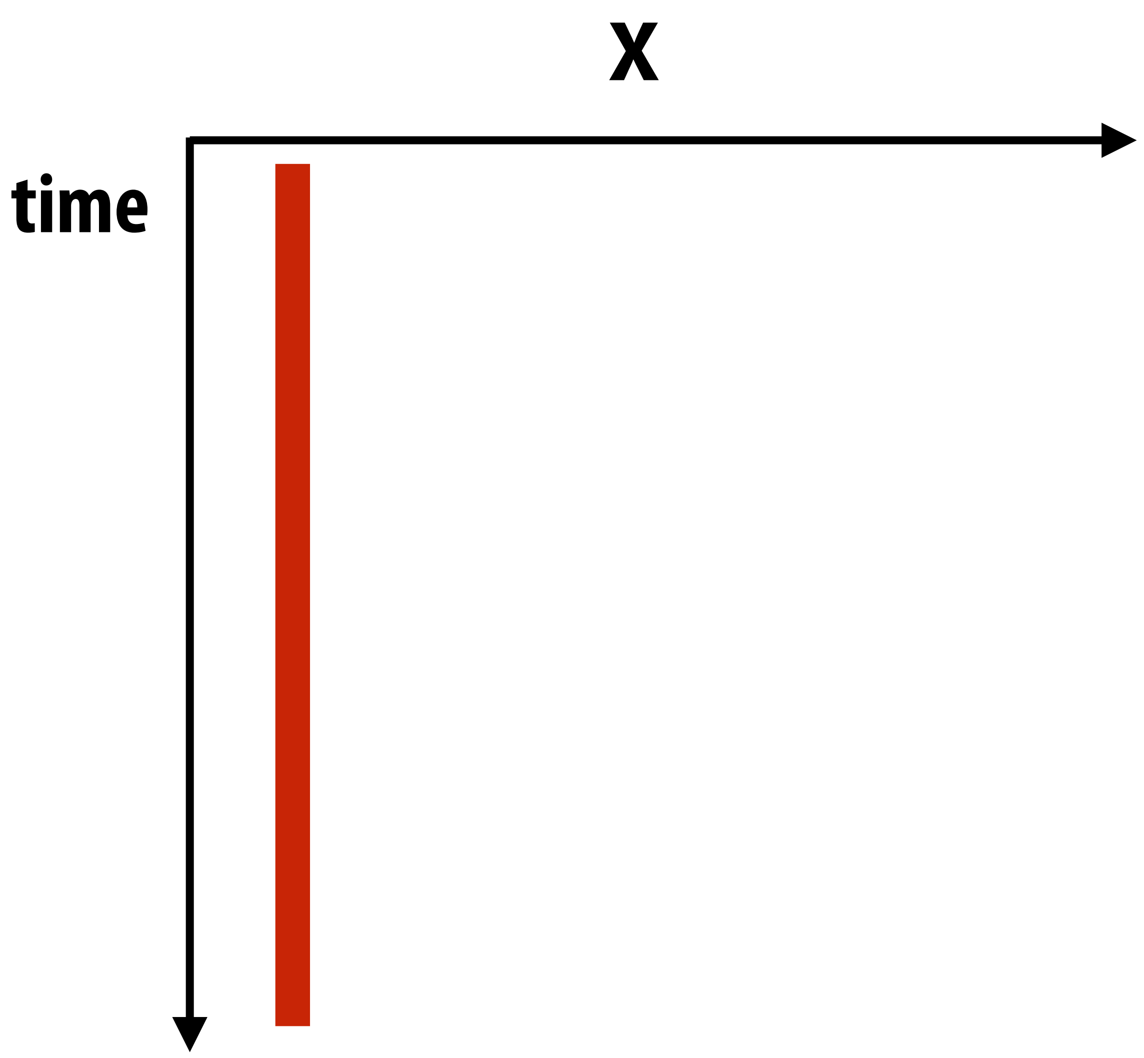
Case 1: object moving from left to right,
eye moving continuously to track object
(<u>eye moving relative to display!</u>)

Light from display
(image is updated each frame)

Case 1 explanation: since eye is moving, object's position is relatively constant relative to eye (as it should be since the eye is tracking it). But due discrete frame rate, object falls behind eye, causing a smearing/strobing effect ("choppy" motion blur). Recall from earlier slide: 90 degree motion, with 50 ms latency results in 4.5 degree smear

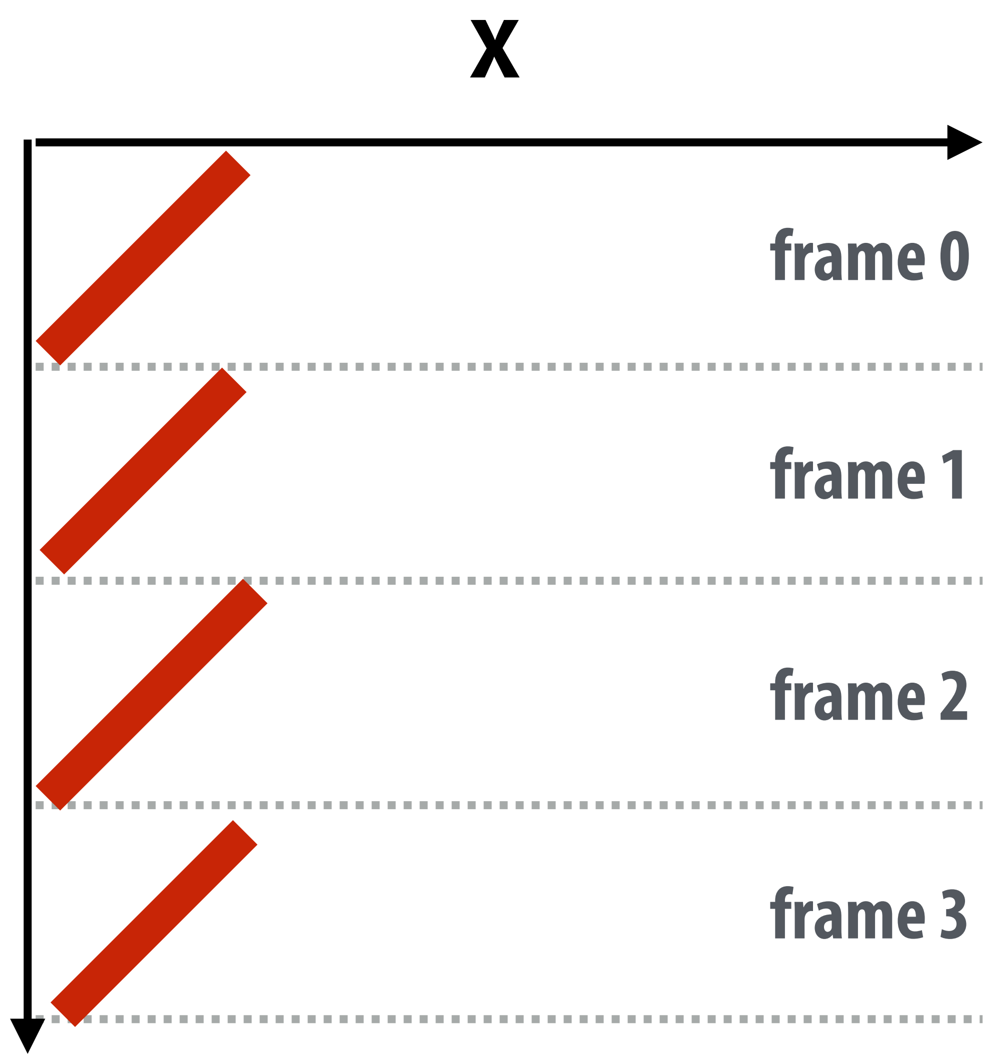
Spacetime diagrams adopted from presentations by Michael Abrash

Reducing judder: increase frame rate

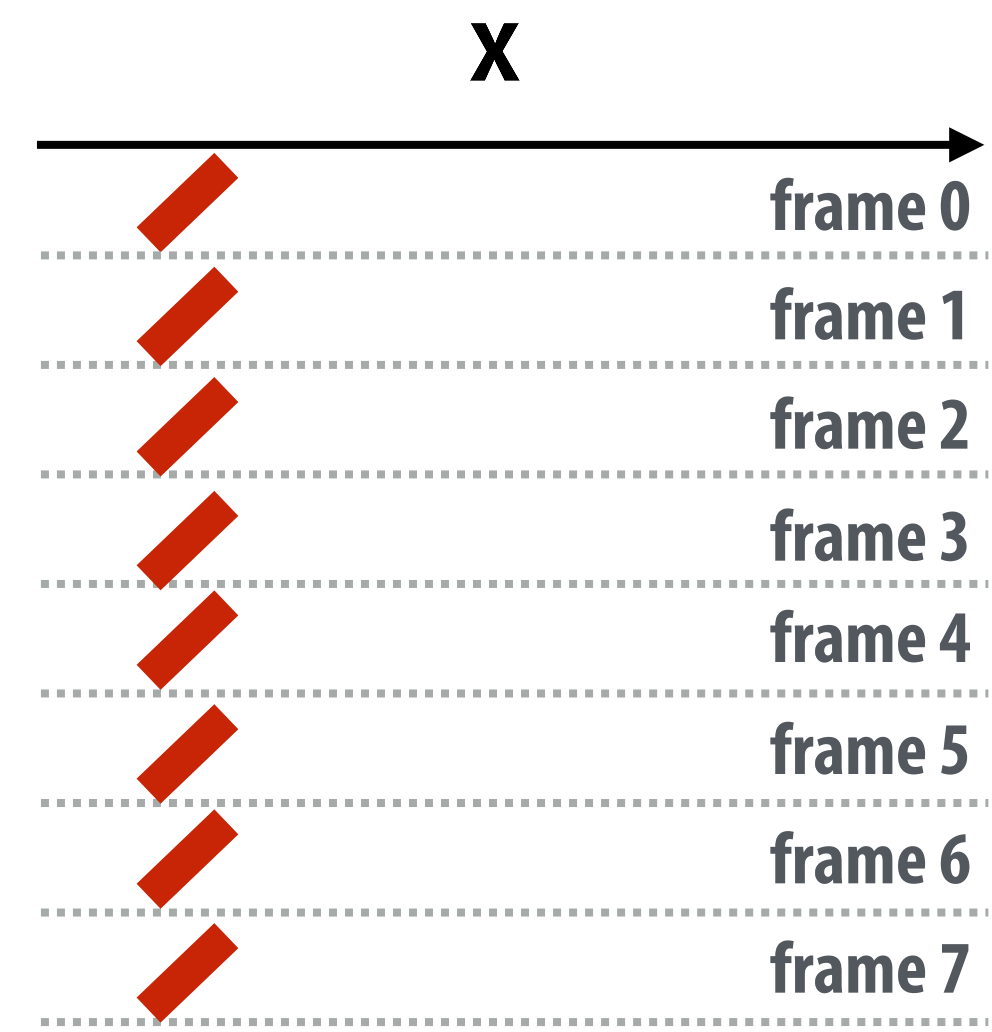


Case 1: continuous ground truth

red object moving left-to-right and eye moving to track object
OR
red object stationary but head moving and eye moving to track object



Light from display (image is updated each frame)

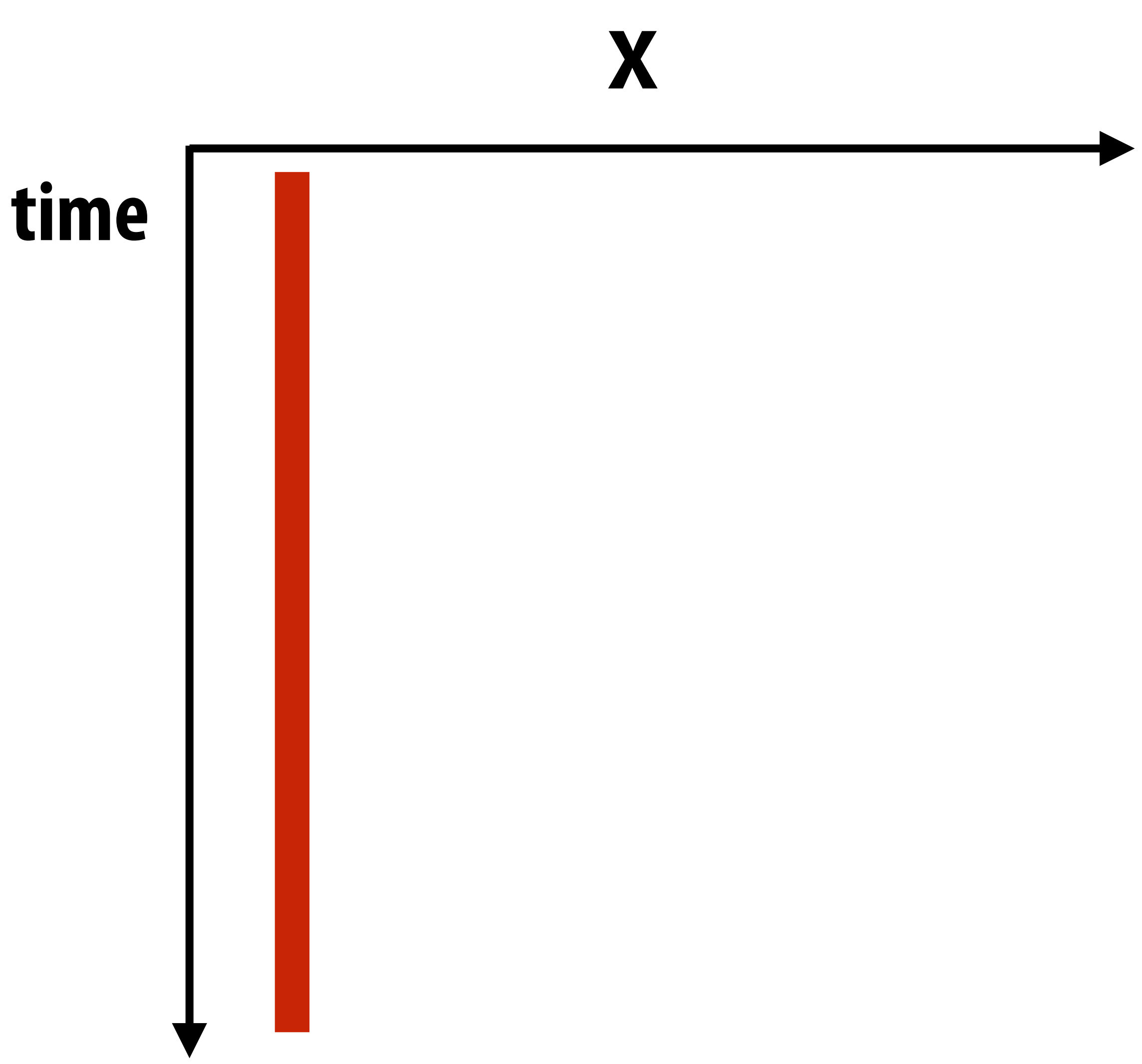


Light from display (image is updated each frame)

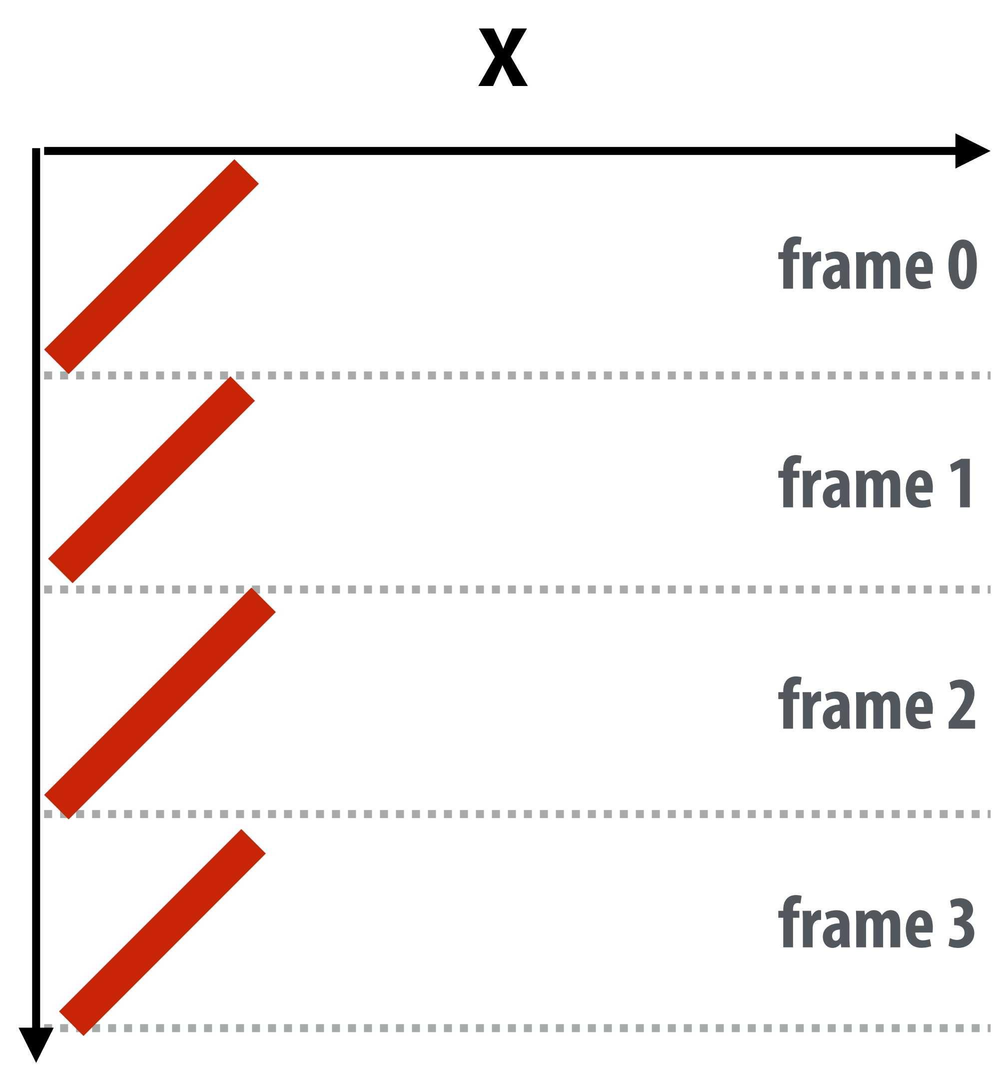
Higher frame rate results in closer approximation to ground truth

Spacetime diagrams adopted from presentations by Michael Abrash

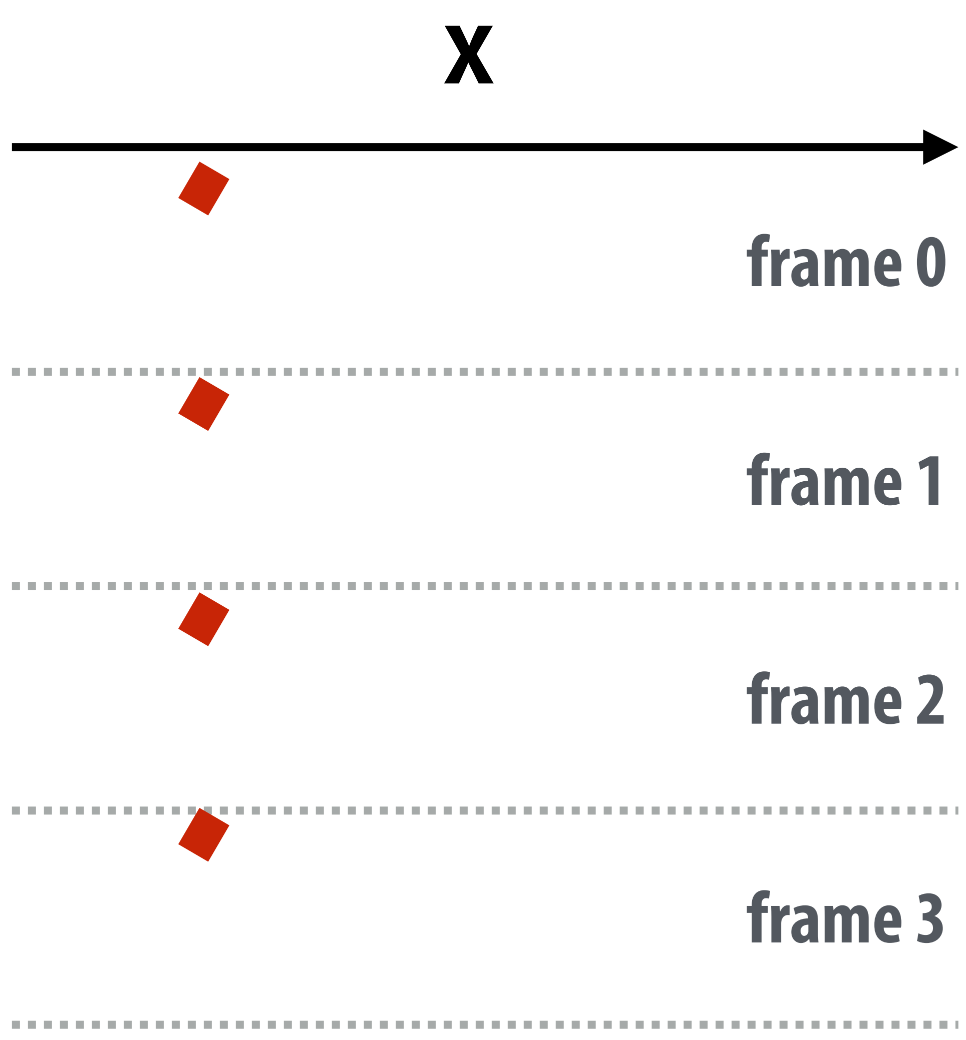
Reducing judder: low persistence display



Case 1: continuous ground truth



Light from full-persistence display



Light from low-persistence display

red object moving left-to-right and
eye moving to track object
OR
red object stationary but head moving
and eye moving to track object

Full-persistence display: pixels emit light for entire frame

Low-persistence display: pixels emit light for small fraction of frame

Oculus DK2 OLED low-persistence display

- 75 Hz frame rate (~13 ms per frame)
- Pixel persistence = 2-3ms

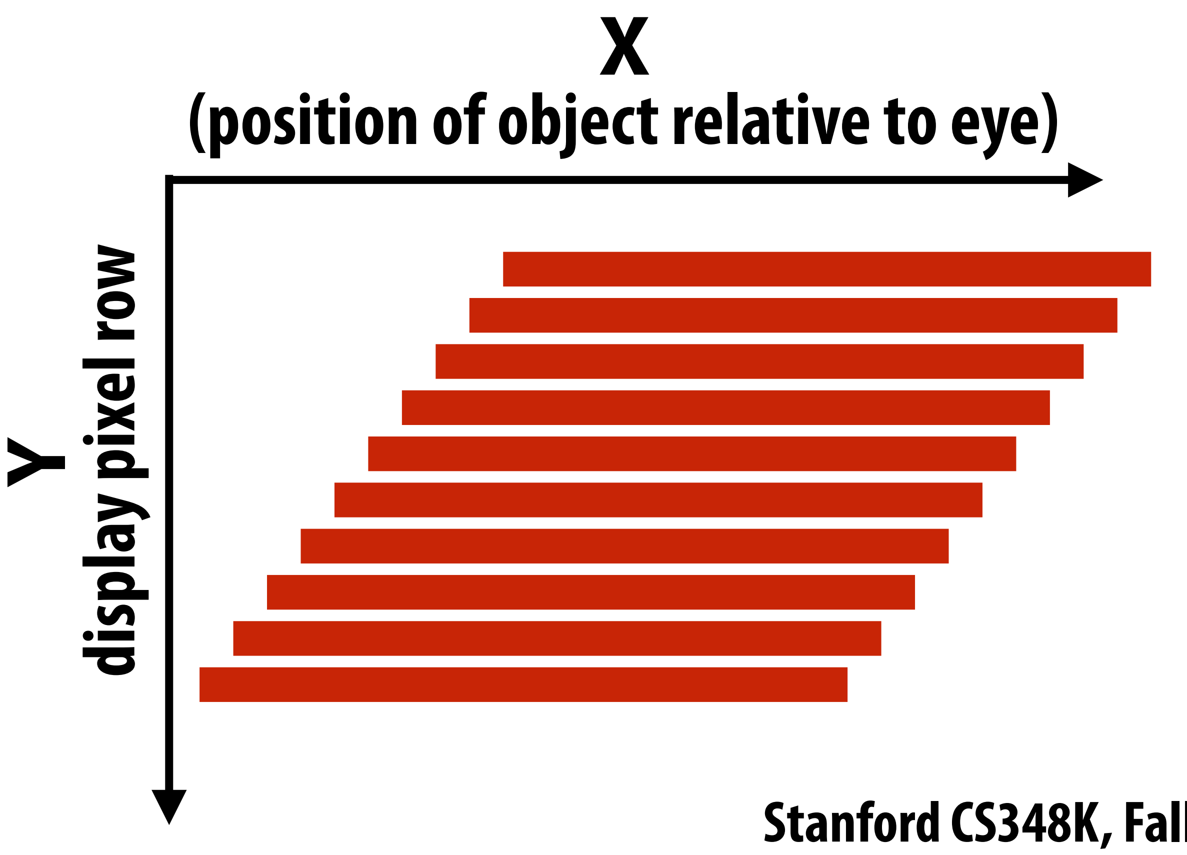
Spacetime diagrams adopted from presentations by Michael Abrash

Artifacts due to rolling OLED backlight

- Image rendered based on scene state at time t_0
- Image sent to display, ready for output at time $t_0 + \Delta t$
- "Rolling backlight" OLED display lights up rows of pixels in sequence
  - Let r be amount of time to "scan out" a row
  - Row 0 photons hit eye at $t_0 + \Delta t$
  - Row 1 photos hit eye at $t_0 + \Delta t + r$
  - Row 2 photos hit eye at $t_0 + \Delta t + 2r$
- Implication: photons emitted from bottom rows of display are "more stale" than photos from the top!
- Consider <u>eye moving horizontally relative to display</u> (e.g., due to head movement while tracking square object that is stationary in world)

Result: perceived shear!

Recall rolling shutter effects on modern digital cameras.



Compensating for rolling backlight

- Perform post-process shear on rendered image
  - Similar to previously discussed barrel distortion and chromatic warps
  - Predict head motion, assume fixation on static object in scene
    - Only compensates for shear due to head motion, not object motion

- Render each row of image at a different time (the predicted time photons will hit eye)
  - Suggests exploration of different rendering algorithms that are more amenable to fine-grained temporal sampling, e.g., ray caster? (each row of camera rays samples scene at a different time)

Increasing frame rate using re-projection

- **Goal: maintain as high a frame rate as possible under challenging rendering conditions:**
  - Stereo rendering: both left and right eye views
  - High-resolution outputs
  - Must render extra pixels due to barrel distortion warp
  - Many "rendering hacks" (bump mapping, billboards, etc.) are less effective in VR so rendering must use more expensive techniques

- **Researchers experimenting with reprojection-based approaches to improve frame rate (e.g., Oculus' "Time Warp")**
  - Render using conventional techniques at 30 fps, reproject (warp) image to synthesize new frames based on predicted head movement at 75-90 fps
  - Interest in image processing hardware on future VR headsets to perform high frame-rate reprojection based on gyro/accelerometer

Summary: near-future VR system components

High-resolution, high-frame rate, wide-field of view display

Low-latency image processing for subject tracking

Massive parallel computation for high-resolution rendering

In headset motion/accel sensors + eye tracker

Exceptionally high bandwidth connection between renderer and display: e.g., 4K x 4K per eye at 90 fps!

On headset graphics processor for sensor processing and re-projection