### 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);
```

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)



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

## **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

### ock Ition units

### it (256 bit) vector registers Itext

### xts in "lockstep" pulating 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)</pre>
  {
    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

Fetch/ Decode **1 scalar or 1 vector** 



### 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++) {</pre>
     outputColor += surfaceColor * clamp(0.0, 1.0, dot(norm, lightDir[i]));
   }
```

## Improving the fictitious throughput processor



two instructions per clock

### Now decode two instructions per clock

- How should we organize the processor to execute those instructions?

## Three possible organizations



two instructions per clock

### **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 multithreading)
- Up to 64 warps are interleaved on the SM (interleaved multithreading)
- **Over 2,048 fragments/** vertices/etc can be processed concurrently by a core

## **NVIDIA GTX 1080 (20 SMs)**

NVIDIA GTX 1080 (2016)	NVIDIA GTX 1080 (2016)	NVIDIA GTX 1080 (2016)	NVIDIA GTX 10		
256 KB-registers 96 KB shared NVIDIA GTX 408 (P.016)	NVIDIA GTX 1080 (2016)	NVIDIA GTX 1080 (2016)			
NVIDIA GTX 1080 (2016)	NVIDIA GTX 1080 (2016)	NVIDIA GTX 1080 (2016)			
NVIDIA GTX 1080 (2016)	NVIDIA GTX 1080 (2016)	NVIDIA GTX 1080 (2016)			
L2 Cache (2 MB)					



**DDR5 DRAM** 



## 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!
- **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

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

# 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/ <u>MX.net/PyTorch</u>

```
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:

- 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.

## evel languages orks like TensorFlow/



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

Name	e 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
<b>C</b> 8	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	C K, S
D1	depthwise conv	v2d 112,	112 3	2 3, 1
D2	depthwise conv	v2d 112,	112 6	4 3, 2
D3	depthwise conv	v2d 56,	56 12	28 3, 1
D4	depthwise conv	v2d 56,	56 12	28 3, 2
D5	depthwise conv	v2d 28,	28 25	<b>56 3</b> , 1
D6	depthwise conv	v2d 28,	28 25	56 3, 2
D7	depthwise conv	v2d 14,	14 51	2 3, 1
D8	depthwise conv	v2d 14,	14 51	3, 2
D9	depthwise conv	v2d 7,	7 10	24 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



## **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]

```
#include <THC/THC.h>
Vinclude <iostream:
Vinclude "math.h"
extern THCState *state:
 __device__ float diff_abs(float x) (
float eps = 1e-8;
  return sqrt(x*x+eps);
  _device__ float d_diff_abs(float x) {
  float eps = 1e-8;
  return x/sqrt(x*x+eps);
  device float weight_z(float x) {
  float abx = diff_abs(x);
  return max(1.0f-abx, 0.0f);
  _device__ float d_weight_z(float x) {
  float abx = diff_abs(x);
  if(abx > 1.0f) (
    return 0.0f;
    // return abx;
  } else (
   return d_diff_abs(x);
 __global___ void BilateralSliceApplyKernel(
    int64_t nthreads,
    const float* grid, const float* guide, const float* input,
    const int bs, const int h, const int w,
    const int gh, const int gw, const int gd
     const int input_chans, const int output_chans
    float* out)
  // - Samples centered at 0.5.
  // - Repeating boundary conditions
  int grid_chans = (input_chans+1)*output_chans;
  int coeff_stride = input_chans+1;
  const int64_t idx = blockIdx.x*blockDim.x + threadIdx.x;
  if(idx < nthreads)
    int x = idx % w;
    int y = (idx / w) % h;
    int out_c = (idx / (w*h)) % output_chans;
int b = (idx / (output_chans*w*h));
    float gx = (x+0.5f)*gw/(1.0f*w);
    float gy = (y+0.5f)*gh/(1.0f*h);
    float gz = guide[x + w*(y + h*b)]*gd;
    int fx = static_cast<int>(floor(gx-0.5f));
    int fy = static_cast<int>(floor(gy-0.5f));
    int fz = static_cast<int>(floor(gz-0.5f));
    // Grid strides
    int sx = 1
    int sy = gw;
    int sz = gw≭gh;
    int sc = gw*gh*gd;
int sb = grid_chans*gd*gw*gh;
    float value = 0.0f;
    for (int in_c = 0; in_c < coeff_stride; ++in_c) {</pre>
      float coeff_sample = 0.0f;
for (int xx = fx; xx < fx+2; ++xx) (</pre>
         int x_ = max(min(xx, gw-1), 0);
float wx = max(1.0f-abs(xx+0.5-gx), 0.0f);
for (int yy = fy; yy < fy+2; ++yy)</pre>
           int y_ = max(min(yy, gh-1), 0);
           float wy = max(1.0f-abs(yy+0.5-gy), 0.0f);
           for (int zz = fz; zz < fz+2; ++zz)
              int z_ = max(min(zz, gd-1), 0);
             float wz = weight_z(zz+0.5-gz);
             int grid_idx =
               sc*(coeff_stride*out_c + in_c) + sz*z_ + sx*x_
                                                 + sy*y_ + sb*b;
             coeff_sample += grid[grid_idx]*wx*wy*wz
       } // Grid trilinear interpolation
       if(in_c < input_chans) {
        int input_idx = x + w*(y + input_chans*(in_c + h*b));
value += coeff_sample*input[input_idx];
      } else { // Offset term
         value += coeff_sample
     out[idx] = value;
 ____global____void BilateralSliceApplvGridGradKernel(
```

gload.\_\_\_\_void bilateralslicexpplyuriduradvermel( int64\_t nthreads, const float\* grid, const float\* guide, const float\* input, const float\* d\_output, const int bs, const int h, const int w, const int gh, const int gw, const int gd, const int input\_chans, const int output\_chans, float\* out)

int grid\_chans = (input\_chans+1)\*output\_chans; int coeff\_stride = input\_chans+1;

const int64\_t idx = blockIdx.x\*blockDim.x + threadIdx.x; if(idx < nthreads) ( int gx = idx % gw; int gy = (idx / gw) % gh; int gz = (idx / (gm\*gw)) % gd; int c = (idx / (gd\*gh\*gw)) % grid\_chans;

int c = (idx / (gd\*gh\*gw)) % grid\_chans; int b = (idx / (grid\_chans\*gd\*gw\*gh)); float scale\_w = w\*1.0/gw;

float scale\_h = h\*1.0/gh;

int left\_x = static\_cast<int>(floor(scale\_w\*(gx+0.5-1))); int right\_x = static\_cast<int>(ceil(scale\_w\*(gx+0.5+1))); int left\_y = static\_cast<int>(floor(scale\_h\*(gy+0.5-1))); int right\_y = static\_cast<int>(ceil(scale\_h\*(gy+0.5+1)));

// Strides in the output
int sx = 1;

```
int sy = w:
     int sc = h≉w
    int sb = output_chans*w*h;
    // Strides in the input
    int isx = 1;
    int isy = w;
int isc = h*w;
    int isb = output_chans*w*h
    int out_c = c / coeff_stride;
    int in_c = c % coeff_stride;
    float value = 0.0f;
     for (int x = left_x; x < right_x; ++x)</pre>
      int x_{-} = x;
      // mirror boundary
       if (x_{-} < 0) x_{-} = -x_{-}1;
      if (x_{-} \ge w) x_{-} = 2 + w - 1 - x_{-};
      float gx2 = (x+0.5f)/scale_w;
      float wx = max(1.0f-abs(gx+0.5-gx2), 0.0f);
      for (int y = left_y; y < right_y; ++y)</pre>
        int y_{-} = y;
        // mirror boundary
         if (y_{-} < 0) y_{-} = -y_{-}1
        if (y_ >= h) y_ = 2*h-1-y_;
        float gy2 = (y+0.5f)/scale_h;
         float wy = max(1.0f-abs(gy+0.5-gy2), 0.0f);
        int guide_idx = x_{-} + w \star y_{-} + h \star w \star b;
         float gz2 = guide[guide_idx]*gd;
        float wz = weight_z(gz+0.5f-gz2);
         if ((gz=0 && gz2<0.5f) || (gz==gd=1 && gz2>gd=0.5f)) {
          wz = 1.0f;
        int back_idx = sc*out_c + sx*x_ + sy*y_ + sb*b;
        if (in_c < input_chans) (
    int input_idx = isc*in_c + isx*x_ + isy*y_ + isb*b;</pre>
        value += wz*wx*wy*d_output[back_idx]*input[input_idx];
} else ( // offset term
           value += wz*wx*wy*d_output[back_idx];
     out[idx] = value;
__globa1__ void BilateralSliceApplyGuideGradKernel(
   int64_t nthreads,
    const float* grid, const float* guide, const float* input
    const float* d_output, const int bs, const int h, const int w,
   const int gh, const int gw, const int gd,
const int input_chans, const int output_chans
    float* out)
 int grid_chans = (input_chans+1)*output_chans;
 int coeff_stride = input_chans+1;
 const int64_t idx = blockIdx.x*blockDim.x + threadIdx.x;
 if(idx < nthreads) {
    int x = idx % w;</pre>
    int v = (idx / w) \% h;
    int b = (idx / (w+h));
   float gx = (x+0.5f)*gw/(1.0f*w);
float gy = (y+0.5f)*gh/(1.0f*h);
   float gz = guide[x + w*(y + h*b)]*gd;
    int fx = static_cast<int>(floor(gx-0.5f));
   int fy = static_cast<int>(floor(gy-0.5f));
int fz = static_cast<int>(floor(gz-0.5f));
    // Grid stride
    int sx = 1;
    int sy = gw;
int sz = gw*gh;
    int sc = gw*gh*gd;
    int sb = grid_chans*gd*gw*gh;
    float out_sum = 0.0f;
    for (int out_c = 0; out_c < output_chans; ++out_c) (</pre>
      float in_sum = 0.0f;
      for (int in_c = 0; in_c < coeff_stride; ++in_c) {</pre>
        float grid_sum = 0.0f;
        for (int xx = fx; xx < fx+2; ++xx) {
           int x_ = max(min(xx, gw-1), 0);
float wx = max(1.0f-abs(xx+0.5-gx), 0.0f);
           for (int yy = fy; yy < fy+2; ++yy)
              int y_ = max(min(yy, gh-1), 0);
             float wy = max(1.0f-abs(yy+0.5-gy), 0.0f);
for (int zz = fz; zz < fz+2; ++zz)</pre>
                int z_ = max(min(zz, gd-1), 0);
               float dwz = gd*d_weight_z(zz+0.5-gz);
                int grid_idx = sc*(coeff_stride*out_c + in_c) + sz*z_ + sx*x_
                                                                     + sy*y_ + sb*b;
                grid_sum += grid[grid_idx]*wx*wy*dwz;
             } // z
        } // y
} // x, grid trilinear interp
        if(in_c < input_chans) {
        in_sum += grid_sum*input[input_chans*(x+w*(y+h*(in_c+input_chans*b)))];
) else ( // offset term
in_sum += grid_sum;
      } // in_c
       out_sum += in_sum*d_output[x + w*(y + h*(out_c + output_chans*b))];
   } // out_c
    out[idx] = out_sum;
```

global void BilateralSliceApplvInputGradKernel( int64\_t nthreads. const float\* grid, const float\* guide, const float\* input, const float\* d\_output, const int bs, const int h, const int w, const int gh, const int gw, const int gd, const int input\_chans, const int output\_chans, float\* out) int grid\_chans = (input\_chans+1)\*output\_chans; int coeff\_stride = input\_chans+1; const int64\_t idx = blockIdx.x\*blockDim.x + threadIdx.x; if(idx < nthreads) { int x = idx % w: int y = (idx / w) % h; int in\_c = (idx / (w\*h)) % input\_chans; int b = (idx / (input\_chans\*w\*h)); float gx =  $(x+0.5f) \pm gw/(1.0f \pm w)$ float gy =  $(y+\theta.5f)*gh/(1.0f*h)$ float gz = guide[x + w\*(y + h\*b)]\*gd; int fx = static\_cast<int>(floor(gx-0.5f)); int fy = static\_cast<int>(floor(gy-0.5f)); int fz = static\_cast<int>(floor(gz-0.5f)); // Grid stride
int sx = 1; int sy = gw; int sz = gw\*gh; int sc = gw\*gh\*gd; int sb = grid\_chans\*gd\*gw\*gh float value = 0.0f; for (int out\_c = 0; out\_c < output\_chans; ++out\_c) ( float chan\_val = 0.0f; for (int xx = fx; xx < fx+2;  $\leftrightarrow xx$ ) ( int x\_ = max(min(xx, gw-1), 0);
float wx = max(1.0f-abs(xx+0.5-gx), 0.0f); for (int yy = fy; yy < fy+2; ++yy) int y\_ = max(min(yy, gh-1), 0);
float wy = max(1.0f-abs(yy+0.5-gy), 0.0f); for (int zz = fz; zz < fz+2; ++zz) int z\_ = max(min(zz, gd-1), 0); float wz = weight\_z(zz+0.5-gz); int grid\_idx = sc\*(coeff\_stride\*out\_c + in\_c) + sz\*z\_ + sx\*x\_ + sy\*y\_ + sb\*b; chan\_val ← grid[grid\_idx]\*wx\*wy\*wz; } // z } // x, grid trilinear interp value += chan\_val\*d\_output[x + w\*(y + h\*(out\_c + output\_chans\*b))]; } // out\_c out[idx] = value; // -- KERNEL LAUNCHERS void BilateralSliceApplyKernelLauncher( int bs, int gh, int gw, int gd, int input\_chans, int output\_chans, int h, int w, const float\* const grid, const float\* const guide, const float\* const input, float\* const out) int total\_count = bs+h+w+output\_chans; const int64\_t block\_sz = 512; const int64\_t nblocks = (total\_count + block\_sz - 1) / block\_sz; if (total\_count > 0) { BilateralSliceApplyKernel<<< nblocks, block\_sz, 0, THCState\_getCurrentStream(state)>>>( total\_count, grid, guide, input, bs, h, w, gh, gw, gd, input\_chans, output\_chans, THCudaCheck(cudaPeekAtLastError()); void BilateralSliceApplyGradKernelLauncher( int bs, int gh, int gw, int gd, int input\_chans, int output\_chans, int h, int w, const float\* grid, const float\* guide, const float\* input, const float\* d\_output. float\* d\_grid, float\* d\_guide, float\* d\_input) int64\_t coeff\_chans = (input\_chans+1)\*output\_chans; const int64\_t block\_sz = 512; int64\_t grid\_count = bs\*gh\*gw\*gd\*coeff\_chans; if (grid\_count > 0) ( const int64\_t nblocks = (grid\_count + block\_sz - 1) / block\_sz; BilateralSliceApplyGridGradKernel<</pre>
mblocks, block\_sz, 0, THCState\_getCurrentStream(state)>>( grid\_count, grid, guide, input, d\_output, bs, h, w, gh, gw, gd, input chans, output chans, d\_grid); int64\_t guide\_count = bs\*h\*w; if (guide\_count > 0) { nst int64\_t nblocks = (guide\_count + block\_sz - 1) / block\_sz; BilateralSliceApplyGuideGradKernel<<< nblocks, block\_sz, 0, THCState\_getCurrentStream(state)>>>( guide\_count, grid, guide, input, d\_output, bs, h, w, gh, gw, gd input\_chans, output\_chans d\_guide); int64\_t input\_count = bs\*h\*w\*input\_chans; inteq\_t input\_count > 0) (
 const int64\_t nblocks = (input\_count + block\_sz - 1) / block\_sz; BilateralSliceApplyInputGradKernel<<< nblocks, block\_sz, 0, THCState\_getCurrentStream(state)>>>( input\_count, grid, guide, input, d\_output, bs, h, w, gh, gw, gd, input\_chans, output\_chans,

### Stanford CS348K, Fall 2018

d\_input);

3

### Value of auto-differentiation

#include <THC/THE.ho #include <liostream> #include "math.hf

extern THEState \*state;

\_\_device\_\_ float diff\_abs(float x) { float eps = le-8; return sgrt(x%+eps);

\_dwvice\_\_ float d\_weight\_x(float x) {
 float abx = diff\_sbn(x);
 if(abx > 1.0f) {
 return Ab;
 // return abx;
 else {
 return d\_diff\_sbn(x);
 }

intol\_t relationship and any property intol\_t related to the second relation of the seco

- Samples centered at 0.5. - Repeating boundary conditions

int grid\_chans = {input\_chans+1}\*output\_chan
int coeff\_stride = input\_chans+1;

const int64\_t idx = blockldk.xebloc if(idx < nthreads) { int x = idx X x; int y = {idx / x; X h; int dx \_ c = {idx / {xeh}} X output int b = {idx / {xeh}} X output in

float gx = {x+0.5f}\*gs/{1.0f\*s}; float gy = {y+0.5f}\*gh/{1.0f\*h}; float gz = gside(x + s\*{y + h\*b})\*gd;

// Grid strikes int sx = 1; int sy = gs; int sz = gsegh; int sc = gsegh; int sb = grid\_chana\*gi

 $\begin{array}{l} \label{eq:constraints} \mbox{finst value = 0.0f;} \\ \mbox{for (int ing = 0; ing, c coneff_string: ++in_sc) } \\ \mbox{finst coneff_sample = 0.0f;} \\ \mbox{finst ac = naplin(a, ge13, 0);} \\ \mbox{finst ac = naglin(a, ge13, 0);} \\ \mbox{finst ac = naglin(3-dap(cod-pc), 0, 0^2);} \\ \mbox{finst ac = naglin(3-dap(cod-pc), 0, 0^2);} \\ \mbox{finst (int ay = fy; ay < fy2; ++ay) } \\ \mbox{j} \end{array}$ 

nt y\_ = max(min(yy, gh-1), 0); lnat sy = max(1.0f-abn(yy+0.5-gg), 0.0f) or {int zz = fz; zz < fz+2; ++zz} t x\_ = max{min\$zx, gd-1}, 0}; ad mz = meight\_z(zz+0.5-gz);

/grid\_idx =
ac\*(coeff\_strid=sut\_c + in\_c) + sz\*z\_ + + sy\*y\_ + s seff\_sample += grid(grid\_idx)%ex\*sy\*sg;

ub minute, ithwadt, nate grid, const float\* guide, const float\* inp nat\* d\_output, const int bs, const int h, const int gh, const int ge, const int gd, nt imput\_charm, const int output\_charm,

nut int64\_t idx = blockldk.x4t Kidx ≤ nthreadx} ζ

int y\_ = y; // mirror boundary if  $(\gamma_- \leq 0) \gamma_- = -\gamma_--1;$ if  $(\gamma_- >= h) \gamma_- = 24h-1-\gamma_-;$ int guide\_ide = x\_ + sey\_ + theets; flmat gr2 = guide[guide\_ide]tgi; flmat sr = swight\_z[gr0.5f+gr2]; fl (gr==0 dd gr20.5f) || (gr==gd-1 dd gr20gd+0.5f)) { sr = 1.0f;  $\begin{array}{l} int \ hack_i \ ids = \ scient_i c + \ scient_i + \ sgieg_i + \ sgieg_i$ 

int sp = s; int sc = h\*s; int sb = output\_chans\*s\*h;

// Strides in the input int inx = 1; int iny = s; int inc = h\*s; int inb = output\_chans\*s\*h;

int out\_c = c / coeff\_stride; int in\_c = c % coeff\_stride;

 $\begin{array}{l} \label{eq:constraint} \label{eq:constraint} & \text{if } \{x_- \leq 0\} \ x_- = \neg x_- 1; \\ & \text{if } \{x_- \coloneqq s\} \ x_- = 2^s s \neg 1 \neg x_-; \end{array}$ 

int x\_ = x;

finat value = 0.0f; for {int x = left\_x; x < right\_x; ++x}

flmat gs2 = {x+0.5f}/scale\_s; flmat sx = max{1.0f-abs(gx+0.5-gs2), 0.0f};

for {int y = left\_y; y < right\_y; ++y}</pre>

int grid\_charm = {input\_charm+1}\*output\_charm; int coeff\_stride = input\_charm+1;  $\begin{array}{l} \mbox{const int64,t idx = blocklds.x+blocklis.x + threadids.x;} \\ \mbox{if(idx < nthreads)} \\ \mbox{int } x = idx \ X =; \\ \mbox{int } y = (idx \ / x \in X) X ;; \\ \mbox{int } b = (idx \ / (w^b)); \\ \end{array}$ 

float  $g_X = (x+0.5f)^{a}g_X/(1.0f^{a}_X);$ float  $g_Y = (y+0.5f)^{a}g_V/(1.0f^{a}_X);$ float  $g_Z = g_{21} de[x + s^{a}(y + h^{a}_X)]^{a}g_Z;$ int fx = static\_cast<int>{floor(gx=0.5f)) int fy = static\_cast<int>{floor(gy=0.5f)) int fx = static\_cast<int>{floor(gz=0.5f)}

// Grid stride int xx = 1; int xy = gx; int xz = gx\*gh; int xc = gx\*gh\*gd; int xb = grid\_chanx\*g

float out\_sum = 0.0f; for {int out\_c = 0; out\_c < output\_chans; ++out\_c) float in\_sum = 0.0f; for (int in\_c = 0; in\_c < coeff\_stride; ++in\_c) {

 $\begin{array}{l} \mbox{finat grid_sum = 0.0f;} \\ \mbox{for (int xx = fx; xx < fxX; ++xx) ( \\ \mbox{int x_{-} = max(inf(xx, ge-1), 0);} \\ \mbox{finat xx = max(1.0f+unf(xe0.5+go), 0.0f);} \\ \mbox{for (int yy = fy; yy < fyx2; ++yy)} \\ \mbox{.} \end{array}$ 

 $\begin{array}{l} \label{eq:linear_state} \inf_{x \in [x,y]} \left\{ y_{1}, y_{2}, y_{2}, y_{3}, y_{$ 

. int z\_ = max(min(zz, gd-1), 0); float dwz = gd+d\_weight\_z(zz+0.5-gz);

int grid\_idx = sc+(coeff\_stride+sut\_c + in\_c) + sz+z\_ + su+z\_ grid\_num += grid[grid\_idx]+sortsy+dsz; } // x, grid trilinear interp

if(in\_c < input\_chann) {
 in\_um += grid\_um\*inpu
} else { // offset term
 in\_um += grid\_um;</pre>

} // in\_c

out\_sum += in\_sum\*d\_output[x + m\*{y + h\*{out\_c + output\_chans\*b}}] } // out\_c out[idx] = out\_sum;

} // out\_c out[idx] = value; t total\_count = bs=h=w=output\_chans;

rid(t\_puids\_court = hubbs; (guids\_court) = 0 { errort int6(t\_hild(secont + hind(sec - 1) / hind(se militaterallic)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(second)(

(ippt\_cont > 0) {
 cont int(i, relations);
 interprise (input\_cont + block\_ur - 1) / block\_ur
 histeral SiloskphyloputGradieredcox
 block, block\_ur, 0; HSChute\_petCarentSitema(state)co>(
 input\_cont, grid, guide, input, d\_output,
 h., h., sp. ge, gd.

out.backward(adjoints) d\_input = input.grad d\_grid = grid.grad d\_guide = guide.grad

out = []

**CUDA** 308 lines 430 ms (1M pix) 2270 ms (4M pix)

red: gradient code

[Figure credit: Li et al 18]

// Grid stride int sx = 1; int sy = gs; int sz = gs\*gh; int sz = gs\*gh\*gd; int sb = grid\_charas\*gd\*gs\*gh; float value = 0.0f; for (int out\_c = 0; out\_c < output\_charm; ++out\_c) { float char\_val = 0.0f; nat char\_val = 0.0f;  $\sigma$  (int xx = fx; xx < fx=2; +=xx) { int x\_ = max[in(xx, p=-1), 0]; float sx = max[1.0f-abn[xx=0.5-gx], 0.0f]; for { int yy = fy; yy < fy=2; +=yy} ; int y\_ = max[min[yy, gh-1], 0]; float my = max[1.0f-abx(yy+0.5-gy), 0.0f]; for {int xx = fx; xx < fx+2; ++xx} int z\_ = max(min(zz, gd-1), 0); float sz = seight\_z(zz+0.5-gz); int grid\_idx = sc\*(coeff\_stride\*out\_c + in\_c) + sz\*z\_ + sz\*z\_ + su\*z\_ + sb\*b; chan\_val += grid[grid\_idx]+sor\*sy+sz; } // y } // x, grid trilinear interp value += chan\_val\*d\_surput[x + m\*[y + h\*[surt\_c - RINEL LANDERS d BilderalDioopplyCorrellancter( to b, int g, int gs, int gs, int input\_chem, int aniput\_chem, cont flash\* cornt grid\_correl cont flash\* cornt grid\_correl cont flash\* cornt input, flast\* corut su()

total\_coart = bortweakupt\_down; indf, i black\_u = S12; indf, i placks = {total\_coart + black\_uz = 1} / black\_uz black\_coart > 0; indexdow, black\_uz, 0; TECStatk\_getCurrentStream[state]000{ total\_coart, grid\_plack\_input, b\_n, n, s, gh, gd, Jopt\_down, sutput\_down, BilateralSiloskepijdradkorvelLaucher( ti bs. int gk. int gs. int gd. nt input\_charm. int output\_charm. int h. int s. rent floats grid, const floats guide, const floats rent floats d\_output. loats d\_grid, floats d\_guide, floats d\_input)

intic, t nitwadt, until, t nitwadt, comt flaat\* grid, comt flaat\* guide, const flaat\* input, comt flaat\* guidud, comt in he, comt in h, comt in h, comt in he, const in he, comt in t h, comt in h, comt in he, const in he, comt int gd, unst in head, comt int south of gd.

int grid\_chans = {input\_chans+1}\*output\_chans int coeff\_stride = input\_chans+1;

int y = {idx / s} % h; int in\_c = {idx / {s\*h}} % input\_chans int b = {idx / {input\_chans\*s\*h}};

flaat gx = (x+0.5f)\*gw/(1.0f\*s); flaat gy = (y+0.5f)\*gb/(1.0f\*b); flaat gz = guide[x + x\*(y + h\*b))\*gd;

int fx = static\_cast<int>{floor(gx=0.5f)); int fy = static\_cast<int>{floor(gy=0.5f)); int fx = static\_cast<int>{floor(gx=0.5f));

66.1 kinds.um = 512; rid...tml + kongregetgebrandf.chans; rid...tml + kongregetgebrandf.chans; rid.t.thalost.um (prid.gamet + block.um - 1) / black.um al Slinekeysthjöridforddormed/ovi k.s.block.um, P. BCStatu getfarmentistrems[tate]pool 4.cmet.grid.grid.m.impi.d.gamtpat. h. m. dt. m. et.

```
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()
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)
```

### **PyTorch** 42 lines 1440 ms (1M pix) out of memory (4M pix)

### [See Gradient Halide for alternative]

## **Barrage of systems/frameworks**

- GLOW (FB): <u>https://github.com/pytorch/glow</u>
- **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) (<u>https://github.com/FluxML/Flux.jl</u>)
- **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 (<u>https://github.com/onnx/onnx</u>), 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**





### **Sony Morpheus**





### AR headset: Microsoft Hololens





### **Oculus Rift CV1 headset**

# ... Uc

Image credit: ifixit.com



## **Role of optics in headset**

- **Create wide field of view**
- Place focal plane at several meters 2. 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/

eye



### **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  $3^{\circ} \sim 30$  pixels from where they would be in an ideal system with 0 latency

## **Oculus CV1 IR camera and IR LEDs**



**60Hz IR Camera** (measures absolute position of headset 60 times a second) **Headset contains:** IR LEDs (tracked by camera) **Gyro** + accelerometer (1000Hz). (rapid relative positioning)



Image credit: ifixit.com

### Valve's Lighthouse: cameraless position tracking



Image credit: Travis Deyle

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

### of receiver: just a light sensor and an accurate clock!

## 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

Eyes designed by SuperAtic LABS from the thenounproject.com



### 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



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



[Roorda 1999]

### Addressing high resolution and high field of view: foveated rendering med-res low-res

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

image

high-res image





### image



### 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**

![](_page_44_Figure_1.jpeg)

### View of checkerboard through Oculus Rift lens

![](_page_44_Picture_3.jpeg)

### 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

![](_page_44_Picture_8.jpeg)

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

![](_page_45_Picture_1.jpeg)

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**

![](_page_46_Picture_1.jpeg)

### **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]

![](_page_46_Picture_5.jpeg)

Warped Image

![](_page_46_Figure_8.jpeg)

**Shading Rate After** Lens Warp

## Multi viewport rendering

![](_page_47_Picture_1.jpeg)

## 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

![](_page_48_Picture_3.jpeg)

Original Viewport

![](_page_48_Picture_5.jpeg)

Enlarged Viewport Shading Rate Increased

[Image credit: NVIDIA]

![](_page_48_Picture_9.jpeg)

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

### Lens matched shading

![](_page_49_Picture_1.jpeg)

[Image credit: Oculus]

## Accounting for interaction of display update + display attached to head

## **Consider object position relative to eye**

![](_page_51_Figure_1.jpeg)

### **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

**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)

## Effect of latency: judder

![](_page_52_Figure_1.jpeg)

(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

![](_page_52_Figure_5.jpeg)

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

### Light from display (image is updated each frame)

## Reducing judder: increase frame rate

![](_page_53_Figure_1.jpeg)

**Case 1: continuous ground truth** 

Light from display (image is updated each frame)

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

Spacetime diagrams adopted from presentations by Michael Abrash

X frame 0 frame 1 frame 2 frame 2 frame 3 frame 4 frame 5 frame 6 frame 7

### Light from display (image is updated each frame)

Higher frame rate results in closer approximation to ground truth

## **Reducing judder: low persistence display**

![](_page_54_Figure_1.jpeg)

**Case 1: continuous ground truth** 

Light from full-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 **Oculus DK2 OLED low-persistence display** 

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

![](_page_54_Figure_9.jpeg)

Light from low-persistence display

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

## **Artifacts due to rolling OLED backlight**

- **Image rendered based on scene state at time t**<sub>0</sub>
- 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 eye moving horizontally relative to display (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.** 

## (position of object relative to eye)

![](_page_55_Figure_15.jpeg)

## **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

### Low-latency image processing for subject tracking

![](_page_58_Picture_2.jpeg)

### Massive parallel computation for high-resolution rendering

![](_page_58_Picture_4.jpeg)

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

![](_page_58_Picture_6.jpeg)

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

### In headset motion/accel sensors + eye tracker

![](_page_58_Picture_9.jpeg)

On headset graphics processor for sensor processing and reprojection