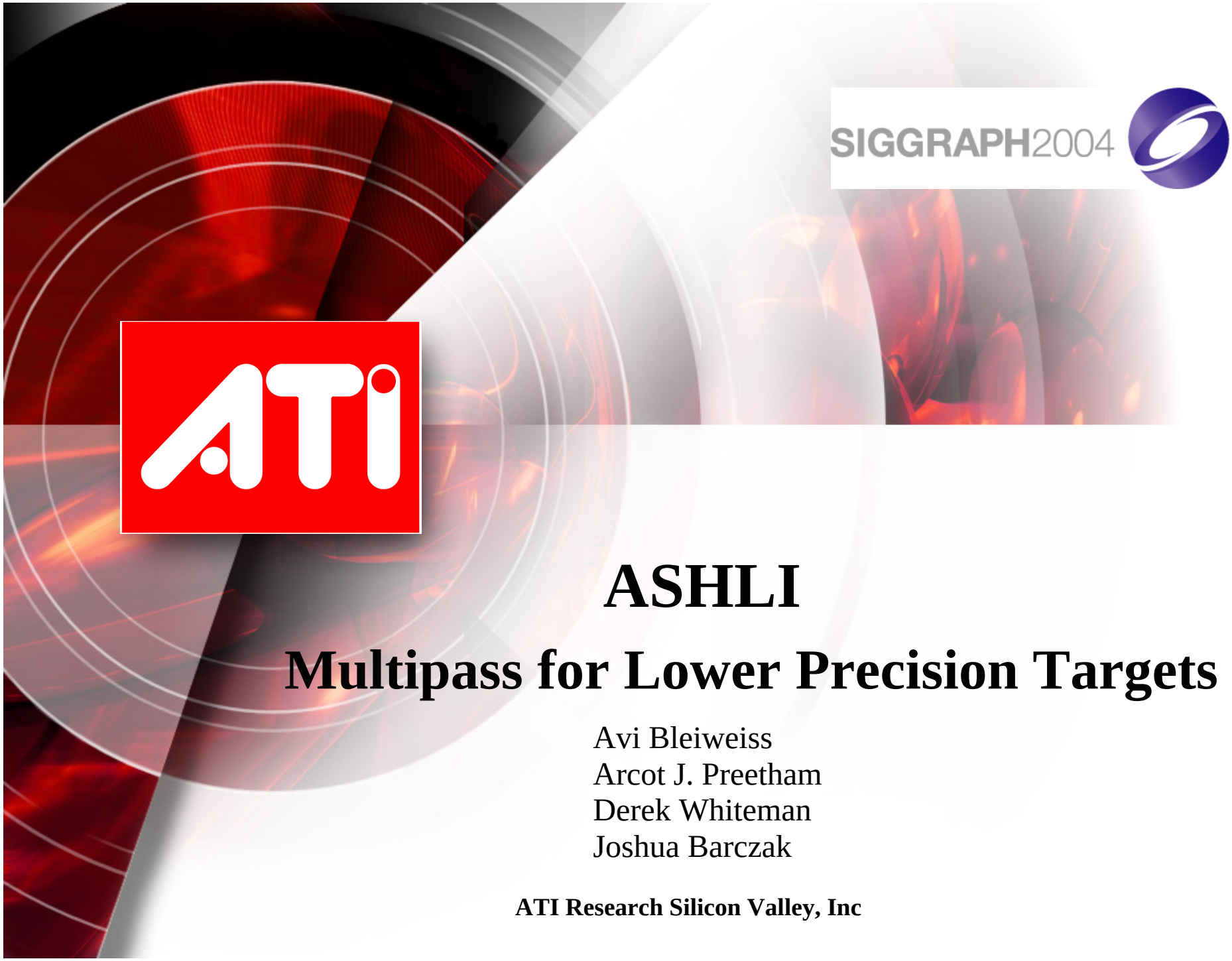


ASHLI

Multipass for Lower Precision Targets

Avi Bleiweiss
Arcot J. Preetham
Derek Whiteman
Joshua Barczak

ATI Research Silicon Valley, Inc

Background

- Shading hardware virtualization is key to Ashli technology
- Segmenting complex shaders still vital
  - Less so for instructions, more for registers and limited vertex iterators
- Multipass rendering incorporates shader partitions
- Intermediate buffer allocation criteria – retain ultimate precision

Motivation

- Multipass is memory demanding
  - Expensive per-pass per-output, window sized, IEEE float component buffer
- Video memory a scarce resource after all
  - Temporary buffers shared with textures and state
- Make multipass pervasive for mobile and hand held platforms
- Challenge unified ultimate precision quest
  - Use lower precision targets (LPT), when possible

Shader Range

- Pre-compiled shader value range essential for determining LPT usage
- Both intermediate and final results delimited by range
- Range calculation, off and in line options
  - Apply numerical analysis based on inputs
    - Use generated Cpp from any of the languages
  - Examine IEEE float buffer for each pass cumulatively
- Inline range computation
  - Resolve pass data into a single min/max buffer
  - Use reduction operation for final range values

Shader Range (cnt'd)

- Per-buffer sample shaders:

```
struct VertexInput {
    float4 position         : POSITION;
    float2 texcoords        : TEXCOORD0;
};

struct VertexOutput {
    float4 position         : POSITION;
    float2 texcoords        : TEXCOORD0;
};


VertexOutput
main(VertexInput vi)
{
    VertexOutput vo;

    vo.texcoords = vi.texcoords;
    vo.position = vi.position;

    return vo;
}
```

```
struct PixelInput {
    float2 texcoords        : TEXCOORD0;
}

struct PixelOutput {
    float2 clr              : COLOR0;
}

uniform sampler2D new;
uniform sampler2D current;

PixelOutput
main(PixelInput pi)
{
    // get value of new pixel from intermediate buffer
    float4 new = tex2D(new, pi.texcoords);

    // get current min/max values from min/max buffer
    // r component is min, g component is max
    float2 cur = tex2D(current, pi.texcoords);

    // test each component of the pixel against the current min and max
    float2 output = float2(0,0);
    output[0] = min(cur[0], min(new[0], min(new[1], min(new[2], new[3]))));
    output[1] = max(cur[1], max(new[0], max(new[1], max(new[2], new[3]))));

    // store the result.
    PixelOutput po;
    po.clr = output;

    return po;
}
```

Vertex Pixel

- First implementation caters to a single color output pixel shader

API for LPT

- Shader value range
  - A program shader optionally takes in delimiter parameters e.g. addShader(processor, entry, *min*, *max*)
- Compile time flags for normalizing temporary and final results
  - Into either an unsigned (0 to 1) or a signed (-1 to +1) buffer
- Shader source invariant of final render target format

Normalization

- LPT normalization pairs derived from shader range
  - Each composed of a scale and an offset
- Apply normalization to each shader segment, right before writing output result
- De-normalize LPT stored result, before being used, for all partitions but the first
- LPT normalization mad's part of subdivision cost metrics

Normalization (cnt'd)

- Code example:
  - Range <-1000,+1000>, into an unsigned buffer

```
// Instructions (Alu):      20
// Instructions (Tex):      0
// Registers    (Temp):     2
// Registers    (Constant): 3
// Registers    (Color):    0
// Registers    (TexCoord): 1
// Registers    (Texture):  0
// Registers    (Output):   1

dcl t0.rgba
def c0, 0.5, 0.0005, 2, 0.5
def c2, 1, 0, 0, 0

rcp r0.r, c1.g
mul r0.r, t0.g, r0.r
frc r0.g, r0.r
add r0.r, r0.r, -r0.g
mul r0.r, c0.a, r0.r
frc r0.r, r0.r
mov r1, c0
mad r0.r, r0.r, r1.b, -c2.r
cmp r0.r, r0.r, -c2.r, c2.r
mad r0.r, r0.r, c0.a, c0.a
mov r1, c1
mad r0.g, r1.r, c0.a, t0.r
lrp r1.r, r0.r, r0.g, t0.r
rcp r0.r, c1.r
mul r0.r, r0.r, r1.r
frc r0.r, r0.r
mad r0.r, r0.r, c0.g, c0.r
mov r0, r0.r
mov oC0, r0
```

```
// Instructions (Alu):      40
// Instructions (Tex):      7
// Registers    (Temp):     5
// Registers    (Constant): 6
// Registers    (Color):    0
// Registers    (TexCoord): 2
// Registers    (Texture):  3
// Registers    (Output):   1

dcl t0.rgba
dcl t1.rgba
dcl_2d s0
dcl_2d s1
dcl_volume s2
def c0, 0.5, 0.0005, -1000, 2000
def c1, 0, 1, 0.1, 0
def c5, 2, 1, 0, 0

texldp r0, t1, s0
mad r0.r, r0.r, c0.a, c0.b
mov r1, c1.a
mul r1.r, r0.r, c1.b
mov r1.a, c5.g
rcp r0.g, c4.a
mul r0.b, t0.g, r0.g
frc r0.a, r0.b
. . . . . .
mad r0.g, c5.r, r2.r, -c5.g
mul r0.g, c2.r, r0.g
mad r0, c3, r0.r, r0.g
mad r0, r0, c0.g, c0.r
mov oC0, r0
```

```
// Instructions (Alu):      52
// Instructions (Tex):      6
// Registers    (Temp):     4
// Registers    (Constant): 7
// Registers    (Color):    0
// Registers    (TexCoord): 2
// Registers    (Texture):  4
// Registers    (Output):   1

dcl t0.rgba
dcl t1.rgba
dcl_2d s0
dcl_2d s1
dcl_2d s2
dcl_volume s3
def c0, 0.5, 0.0005, 0, 1
def c1, -1000, 2000, 0, 1
def c4, 0.1, 0, 2, 0.5
def c6, 1, 0, 0, 0

texldp r0, t1, s1
mad r0.r, r0.r, c1.g, c1.r
mov r1, c4.g
mul r1.r, r0.r, c4.r
. . . . . .
cmp r0.r, r0.r, r3.r, c4.g
add r0.r, r0.r, -r0.a
mul r0.r, r0.g, r0.r
mad r2, r2, c0.a, c0.b
lrp r2, r0.r, r2, r1
mad r0, r2, c0.g, c0.r
mov oC0, r0
```

Pass # 0 | Pass # 1 | Pass # 2

Footprint

- Target format choices for one, two, and four components:
  - 32-bit floating point (IEEE)
  - 16-bit float (10-bit mantissa, 5-bit exponent)
  - 16-bit unsigned integer
  - 8-bit unsigned integer
- Consider a 1K by 1K pixels window, per-pass per-output buffer cost (MB):

Format	Single Component	Two Components	Four Components
32 bits	4	8	16
16 bits	2	4	8
8 bits	1	N/A	4

Footprint (cnt'd)

- Multiple output partitions may increase single output mate footprint
  - Lesser passes, more buffers per-pass
- Use one or two component intermediate buffers, when possible
- Reclaim inactive output buffer(s) from previous passes
- Reorder pass execution to avoid buffer dependencies across passes

Performance

- LPT performance impact is of interest for pixel shading limited scenes
- Single (SRT) and multiple (MRT) render targets evaluated for memory behavior
- SRT:
  - Memory bandwidth lesser factor for compute intensive passes:
    - Figures are in frames-per-second (32 bit relative in parenthesis)

Instructions per Pass	32 bits	16 bits	8 bits
8	10.7 (1.00)	16.5 (1.54)	20.2 (1.88)
16	21.8 (1.00)	29.8 (1.36)	33.1 (1.52)
32	27.0 (1.00)	34.5 (1.27)	35.9 (1.33)
64	44.1 (1.00)	45.7 (1.04)	45.7 (1.04)

Performance (cnt'd)

- MRT:
  - A more involved memory access pattern with lesser locality
  - Scene performance model:
    - Evaluated in increments of 5 distant lights
    - Consistent four outputs per pass, a four component output buffer
    - Figures are in frames-per-second (32 bit relative in parenthesis)

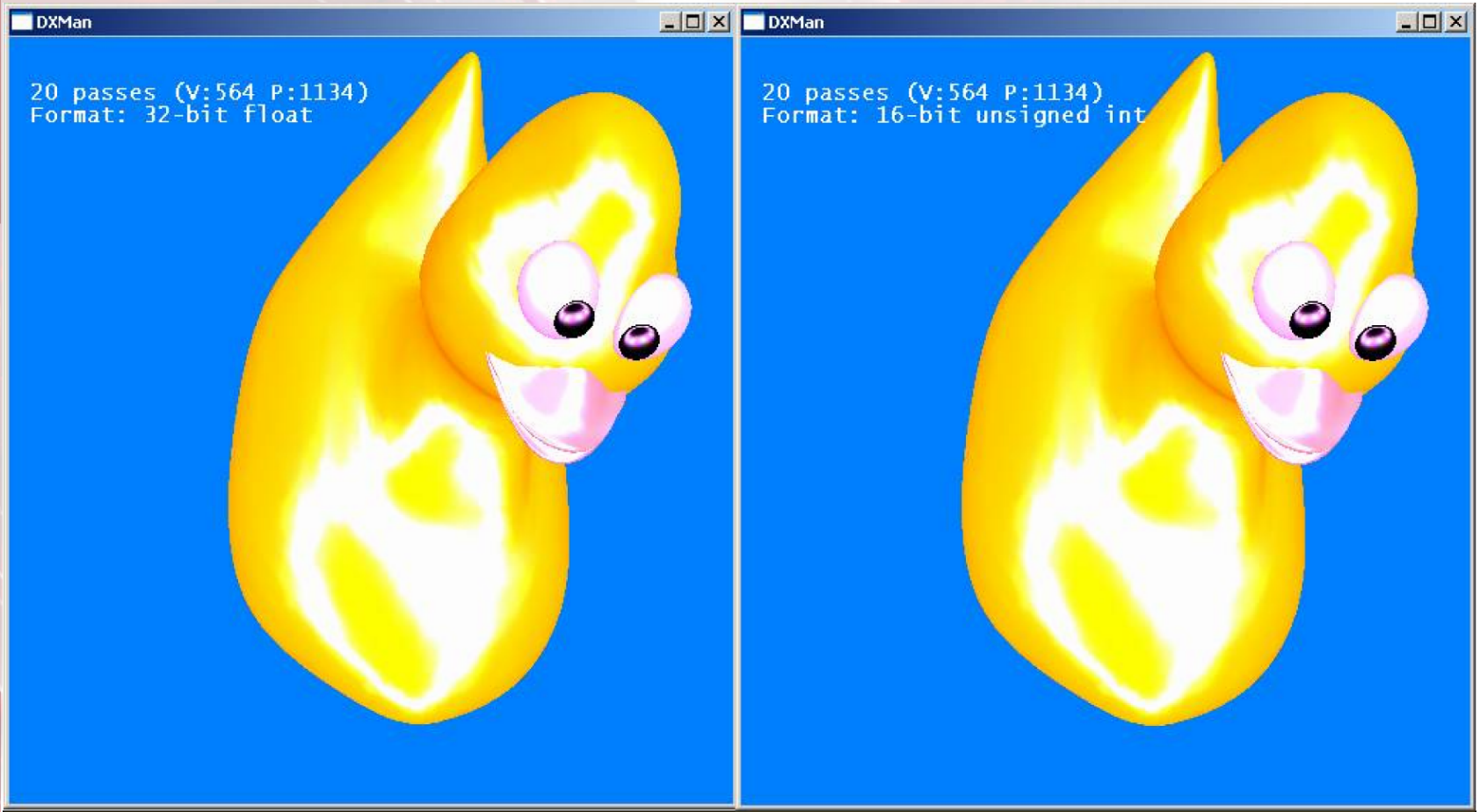
Lights	Passes	Buffers	32 bits	16 bits	8 bits
10	14	24	11.90 (1.00)	18.45 (1.55)	25.74 (2.16)
15	20	34	7.76 (1.00)	11.86 (1.52)	18.35 (2.39)
20	27	44	5.97 (1.00)	8.88 (1.48)	13.69 (2.29)
25	34	54	4.86 (1.00)	7.08 (1.46)	10.98 (2.26)
30	40	64	3.73 (1.00)	6.13 (1.64)	9.29 (2.49)

Quality

- A shader range maps onto a *minimal* LPT format
- Perception quantization error determining factor
- Any LPT format above the lowest match expects to yield comparable render quality
- Following samples depict normalization to a shader range of <0, 40>
  - 16 bit unsigned integer minimal format

Quality (cnt'd)

- LPT format matches range, hardly noticed loss

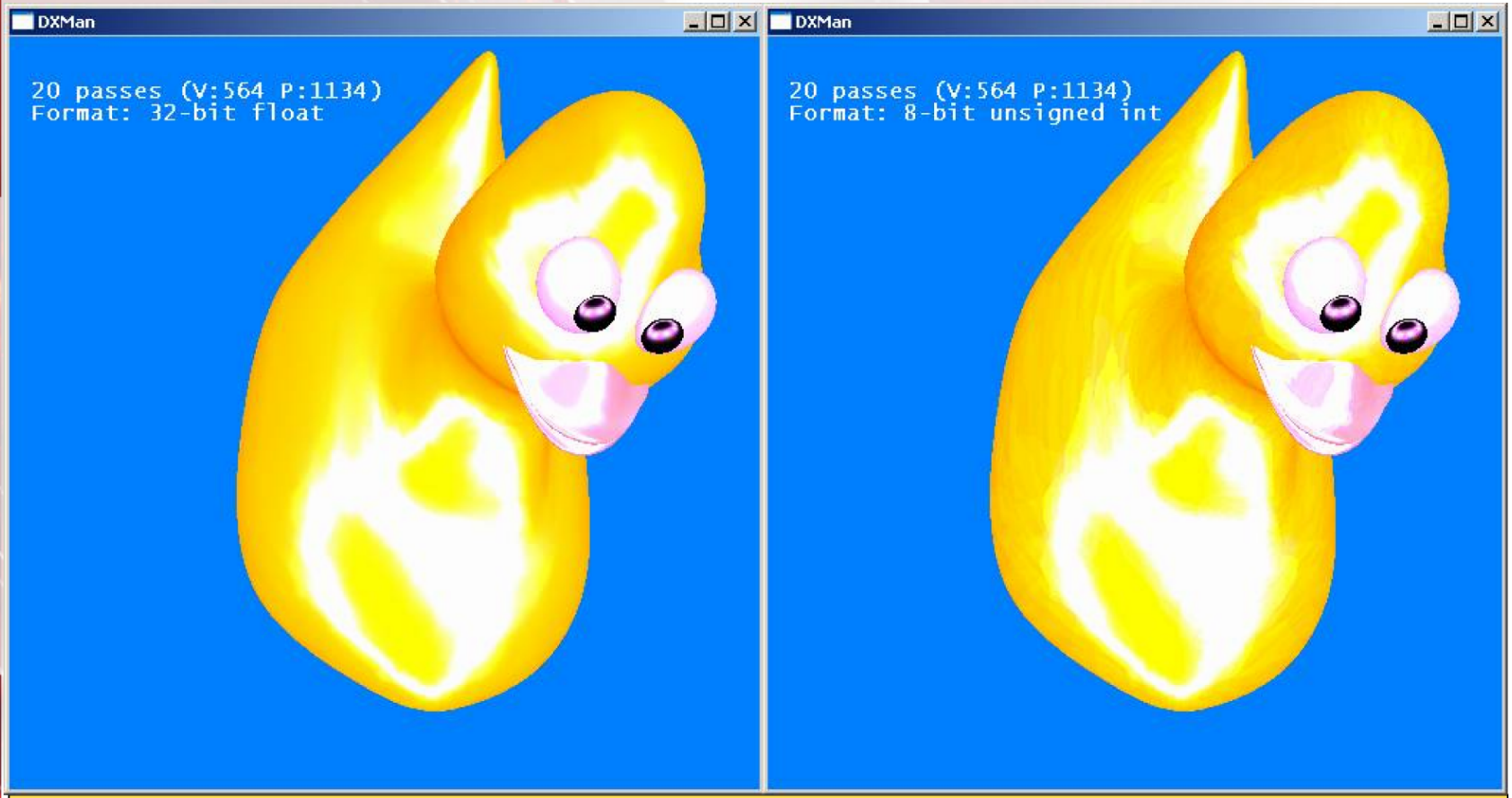


32 bit IEEE float

16 bit unsigned integer

Quality (cnt'd)

- LPT format less than range, significant banding



32 bit IEEE float

8 bit unsigned integer

Summary

- Ashli decouples final render target precision from shader source
- Multipass rendering more affordable on low cost platforms, using LPT
- Ashli main download link and contact:
  - http://www.ati.com/developer/ashli.html
  - devrel@ati.com