Pixel Synchronization:

Solving Old Graphics Problems with New Data Structures

Marco Salvi

Advanced Rendering Technology

Intel - San Francisco

My Background

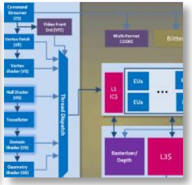
- 7 yrs as Gfx Engineer on PC and two generations of Sony & MS consoles
  - High performance 3D engines
  - Exponential shadow maps & deferred shadowing
  - HDR rendering & MSAA with LogLuv buffers (aka nao32 ☺)

My Background

- 7 yrs as Gfx Engineer on PC and two generations of Sony & MS consoles
  - High performance 3D engines
  - Exponential shadow maps & deferred shadowing
  - HDR rendering & MSAA with LogLuv buffers (aka nao32 ☺)

- Intel R&D – Tech Lead in Advanced Rendering Technology team (2008 – present)
  - Shadow map filtering & partitioning schemes
  - OIT, anti-aliasing, volumetric shadows
  - Stochastic rasterization & shader caches
  - New graphics architectures





Talk Outline

- Introduction and Problem Statement
- Pixel Synchronization
- Applications & Demos
- Performance Tips & Tricks
- Summary
- Q&A

Problem Statement

- Programmable shaders had (and continue to have) huge impact
  - Spurred the development of countless new rendering techniques

*3D pipeline stages post pixel/fragment shading

Problem Statement

- Programmable shaders had (and continue to have) huge impact
  - Spurred the development of countless new rendering techniques
- Pipeline back-end* still not programmable
  - Can only order color, z & stencil operations from a fixed menu..
  - ..but very fast and power efficient

*3D pipeline stages post pixel/fragment shading

Problem Statement

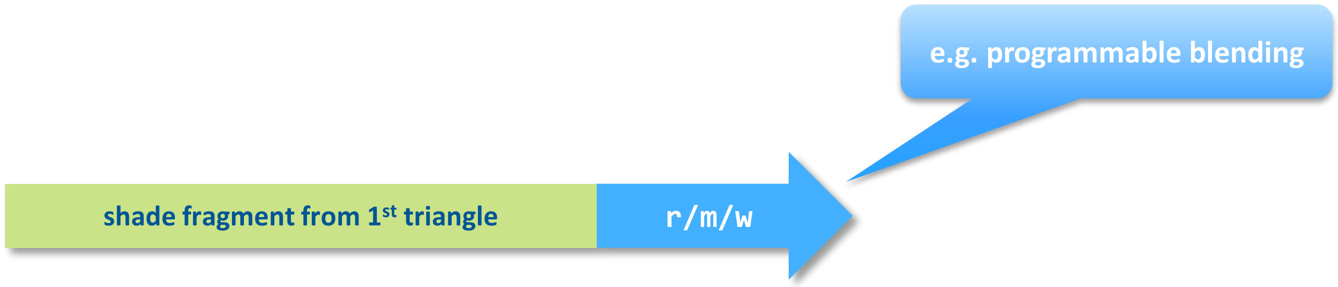
- Programmable shaders had (and continue to have) huge impact
  - Spurred the development of countless new rendering techniques
- Pipeline back-end* still not programmable
  - Can only order color, z & stencil operations from a fixed menu..
  - ..but very fast and power efficient
- Add new programmable back-end?
  - Let it coexist side by side with fixed function HW to leverage respective strengths

Programmable Back-End

- DX11/OGL 4.2 enable arbitrary R/W memory ops from a pixel shader but..

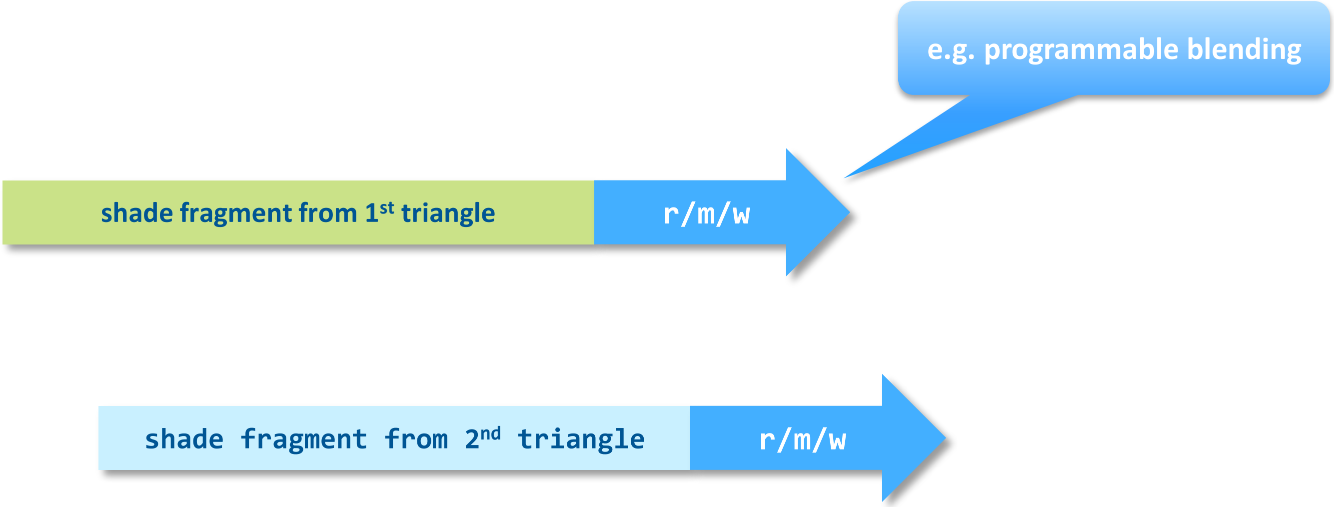
Programmable Back-End

- DX11/OGL 4.2 enable arbitrary R/W memory ops from a pixel shader but..



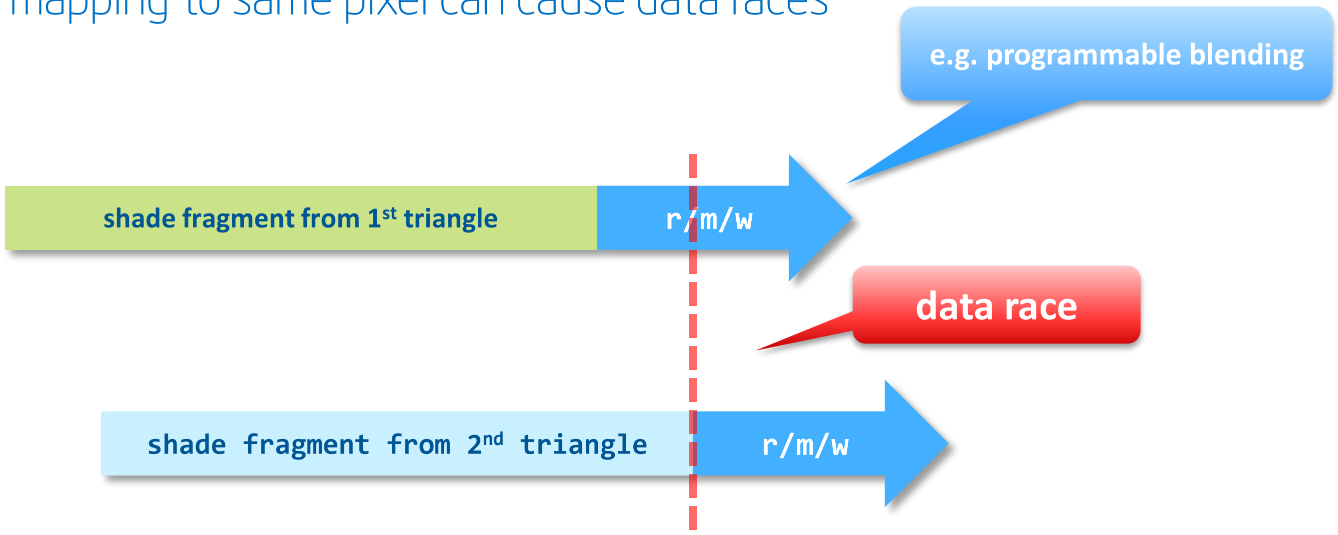
Programmable Back-End

- DX11/OGL 4.2 enable arbitrary R/W memory ops from a pixel shader but..



Programmable Back-End

- DX11/OGL 4.2 enable arbitrary R/W memory ops from a pixel shader but..
  - Fragments mapping to same pixel can cause data races



Programmable Back-End

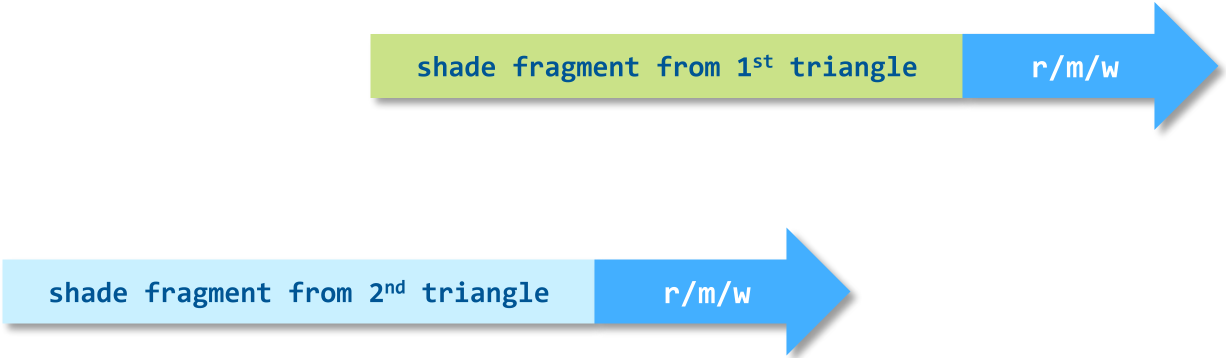
- DX11/OGL 4.2 enable arbitrary R/W memory ops from a pixel shader but..
  - Fragments mapping to same pixel can cause data races

Programmable Back-End

- DX11/OGL 4.2 enable arbitrary R/W memory ops from a pixel shader but..
  - Fragments mapping to same pixel can cause data races

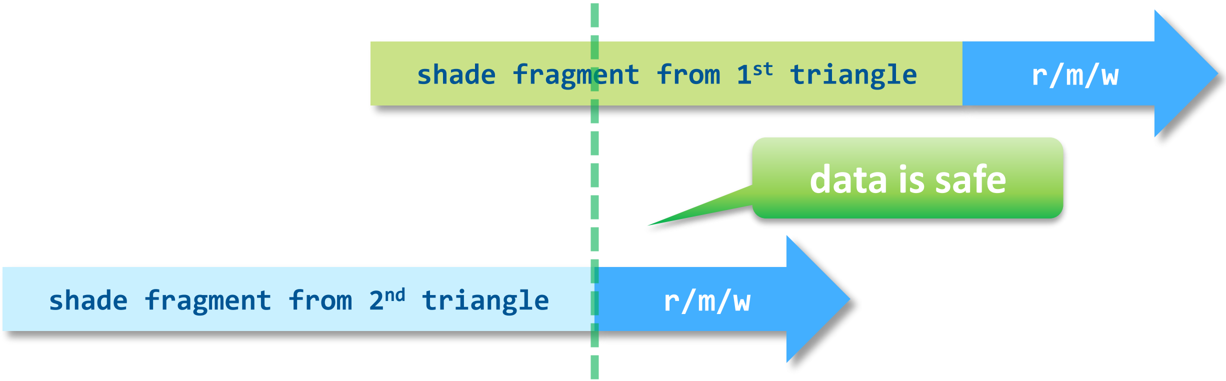
Programmable Back-End

- DX11/OGL 4.2 enable arbitrary R/W memory ops from a pixel shader but..
  - Fragments mapping to same pixel can cause data races



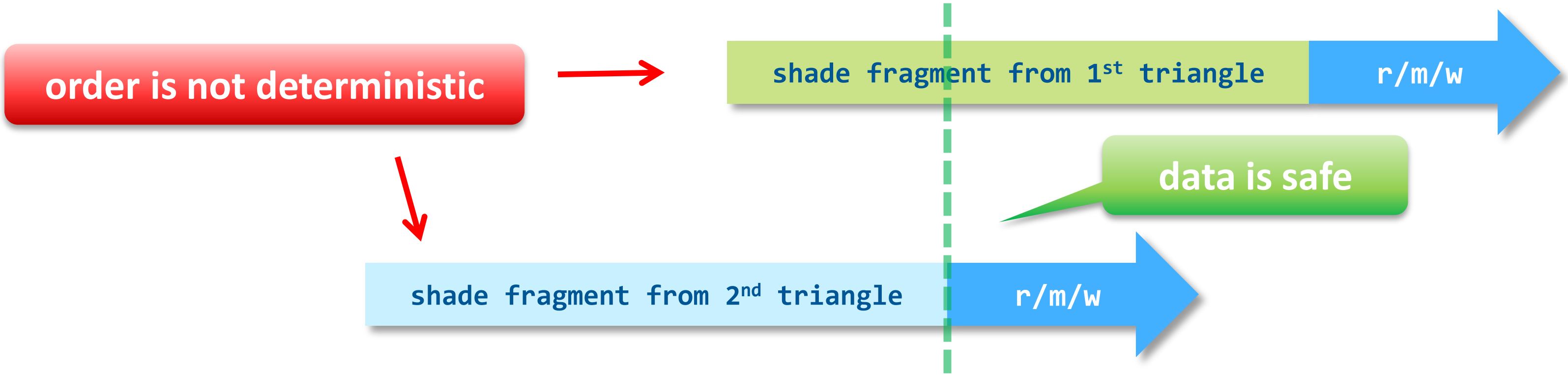
Programmable Back-End

- DX11/OGL 4.2 enable arbitrary R/W memory ops from a pixel shader but..
  - Fragments mapping to same pixel can cause data races



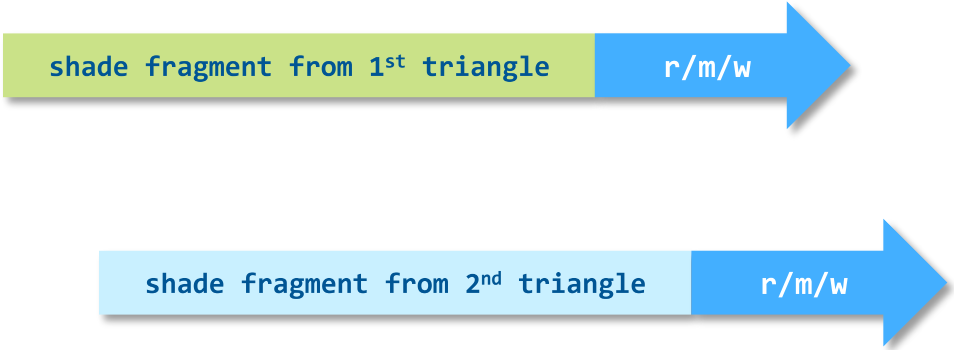
Programmable Back-End

- DX11/OGL 4.2 enable arbitrary R/W memory ops from a pixel shader but..
  - Fragments mapping to same pixel can cause data races
  - Fragments can be shaded out-of-order, can't support order-dependent algorithms

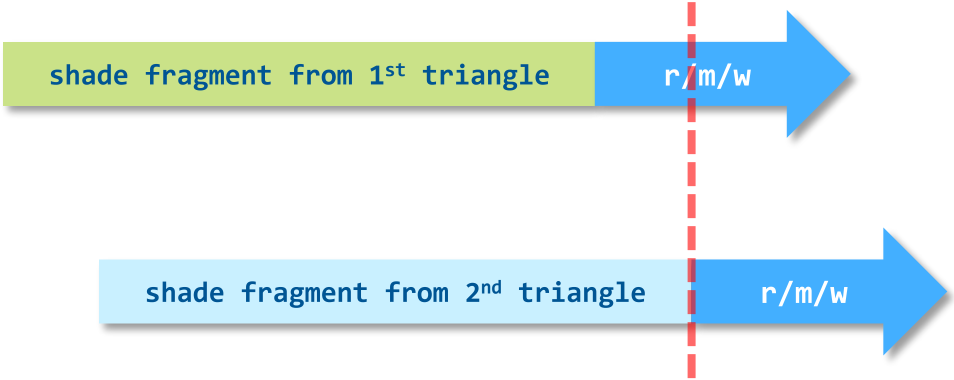


Programmable Back-End

Programmable Back-End

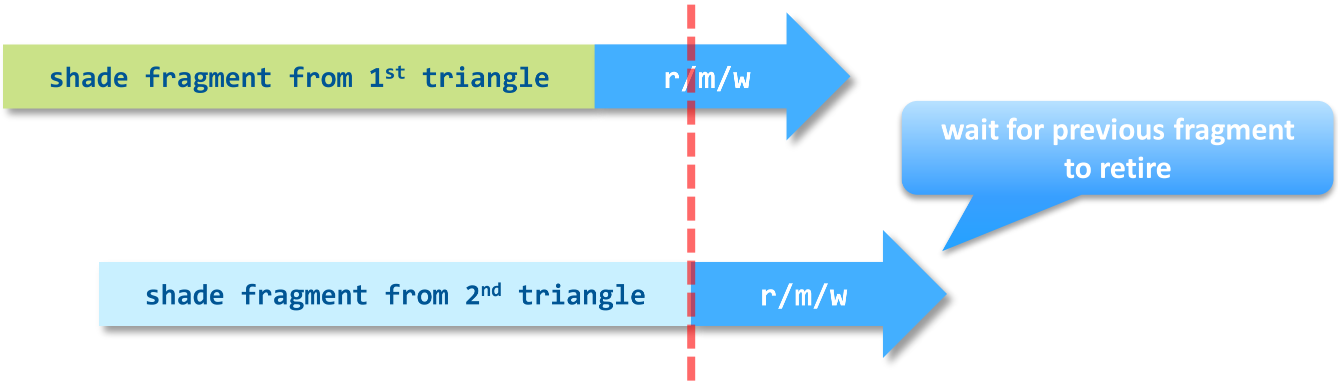


Programmable Back-End



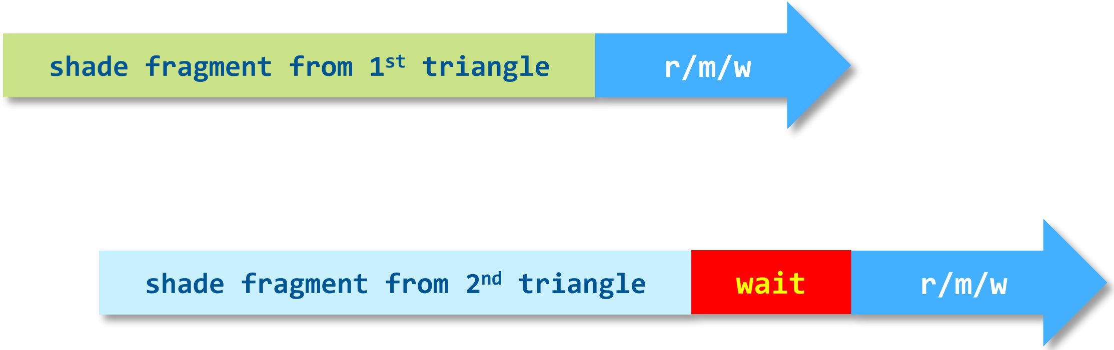
Programmable Back-End

- Haswell can detect dependencies among fragments and..



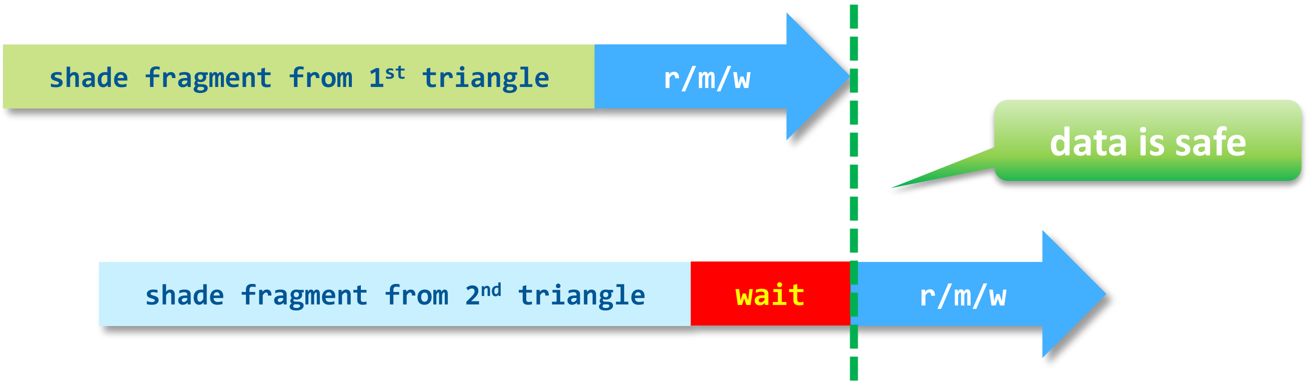
Programmable Back-End

- Haswell can detect dependencies among fragments and..



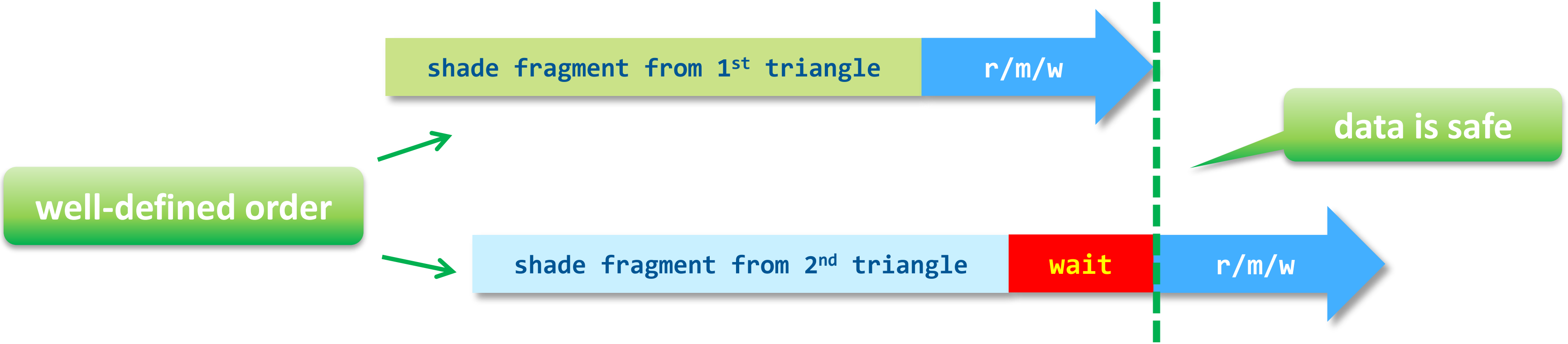
Programmable Back-End

- Haswell can detect dependencies among fragments and..
  - Avoid data races



Programmable Back-End

- Haswell can detect dependencies among fragments and..
  - Avoid data races
  - Guarantee primitive submission order for R/M/W memory operations



Pixel Synchronization

- Simple extension for pixel/fragment shaders
  - Enable ordering for R/W memory accesses (i.e. same order as alpha-blending)
  - Just a function call in your shader: `IntelExt_BeginPixelOrdering()`

Pixel Synchronization

- Simple extension for pixel/fragment shaders
  - Enable ordering for R/W memory accesses (i.e. same order as alpha-blending)
  - Just a function call in your shader: `IntelExt_BeginPixelOrdering()`
- Very good performance
  - Little to no performance impact in most cases
  - R/W memory accesses are backed by the full SoC cache hierarchy

Pixel Synchronization

- Simple extension for pixel/fragment shaders
  - Enable ordering for R/W memory accesses (i.e. same order as alpha-blending)
  - Just a function call in your shader: `IntelExt_BeginPixelOrdering()`
- Very good performance
  - Little to no performance impact in most cases
  - R/W memory accesses are backed by the full SoC cache hierarchy
- More powerful than reading back the frame buffer from a pixel shader
  - Build and access data structures of arbitrary size/type/dimensionality (including voxels ☺)
  - Decoupled from MSAA, can work with per-pixel and/or per-sample data structures

Example: Blending on a RGBE color buffer

Example: Blending on a RGBE color buffer

```
void PS_RGBE_Blend (...)
{
    IntelExt_Init();
```

Initialize shader extensions

Example: Blending on a RGBE color buffer

Compute fragment color & alpha

```
void PS_RGBE_Blend (...)
{
    IntelExt_Init();

    float3 rgb   = ...
    float  alpha = ...
```

Initialize shader extensions

Example: Blending on a RGBE color buffer

```
void PS_RGBE_Blend (...)
{
    IntelExt_Init();

    float3 rgb   = ...
    float  alpha = ...

    IntelExt_BeginPixelOrdering();
```

Compute fragment color & alpha

Initialize shader extensions

Enable pixel synchronization

Example: Blending on a RGBE color buffer

Compute fragment color & alpha

Read RGBE buffer & convert to RGB

Initialize shader extensions

Enable pixel synchronization

```
void PS_RGBE_Blend (...)
{
    IntelExt_Init();

    float3 rgb   = ...
    float  alpha = ...

    IntelExt_BeginPixelOrdering();

    uint   rgbe   = gRGBEBuffer[xy];
    float3 dstRGB = RGBE_to_RGB(rgbe);
```

Example: Blending on a RGBE color buffer

```
void PS_RGBE_Blend (...)
{
    IntelExt_Init();

    float3 rgb   = ...
    float  alpha = ...

    IntelExt_BeginPixelOrdering();

    uint   rgbe   = gRGBEBuffer[xy];
    float3 dstRGB = RGBE_to_RGB(rgbe);

    dstRGB  = alpha * rgb + (1 - alpha) * dstRGB;
```

Compute fragment color & alpha

Initialize shader extensions

Enable pixel synchronization

Read RGBE buffer & convert to RGB

Alpha-blending in RGB space

Example: Blending on a RGBE color buffer

```
void PS_RGBE_Blend (...)
{
    IntelExt_Init();

    float3 rgb   = ...
    float  alpha = ...

    IntelExt_BeginPixelOrdering();

    uint   rgbe   = gRGBEBuffer[xy];
    float3 dstRGB = RGBE_to_RGB(rgbe);

    dstRGB  = alpha * rgb + (1 – alpha) * dstRGB;

    gRGBEBuffer[xy]  = RGB_to_RGBE(dstRGB);
}
```

Compute fragment color & alpha

Initialize shader extensions

Enable pixel synchronization

Read RGBE buffer & convert to RGB

Alpha-blending in RGB space

Conversion to RGBE & buffer write

Example: Blending on a RGBE color buffer

```
void PS_RGBE_Blend (...)
{
    IntelExt_Init();

    float3 rgb   = ...
    float  alpha = ...

    IntelExt_BeginPixelOrdering();

    uint   rgbe   = gRGBEBuffer[xy];
    float3 dstRGB = RGBE_to_RGB(rgbe);

    dstRGB  = alpha * rgb + (1 – alpha) * dstRGB;

    gRGBEBuffer[xy]  = RGB_to_RGBE(dstRGB);
}
```

always run
concurrently with other
fragments

Example: Blending on a RGBE color buffer

```
void PS_RGBE_Blend (...)
{
    IntelExt_Init();

    float3 rgb   = ...
    float  alpha = ...

    IntelExt_BeginPixelOrdering();

    uint   rgbe   = gRGBEBuffer[xy];
    float3 dstRGB = RGBE_to_RGB(rgbe);

    dstRGB  = alpha * rgb + (1 - alpha) * dstRGB;

    gRGBEBuffer[xy]  = RGB_to_RGBE(dstRGB);
}
```

always run concurrently with other fragments

might wait for the retirement of other fragments that map to the same pixel

A Few Programmable Blending Applications

- New blending operators, non-linear color spaces, exotic encodings, etc.
  - e.g. RGBE, LogLuv, etc.

A Few Programmable Blending Applications

- New blending operators, non-linear color spaces, exotic encodings, etc.
  - e.g. RGBE, LogLuv, etc.
- Blending for deferred shaders
  - e.g. Apply decals by blending normals and other material attributes

*Bavoil et al. *"Multi-fragment effects on the GPU using the *k*-buffer".* Proceedings of the 2007 symposium on Interactive 3D graphics and games

K-Buffer

- Generalization of the Z-Buffer*
  - Render N-layers of the image in a single pass

*Bavoil et al. *"Multi-fragment effects on the GPU using the k-buffer"*. Proceedings of the 2007 symposium on Interactive 3D graphics and games

K-Buffer

- Generalization of the Z-Buffer*
  - Render N-layers of the image in a single pass

- Countless applications:
  - Depth-peeling
  - Constructive solid geometry
  - Depth-of-field & motion blur
  - Volume rendering
  - …
  - <insert your idea here ☺>

K-Buffer: Single-Pass Depth Peeling

Compute fragment color, z, etc..

```
void PSMain(...)
{
    IntelExt_Init();
    Fragment frag = {...};
```

K-Buffer: Single-Pass Depth Peeling

Compute fragment color, z, etc..

```
void PSMain(...)
{
    IntelExt_Init();
    Fragment frag = {...};

    IntelExt_BeginPixelOrdering();
```

Enable pixel synchronization

K-Buffer: Single-Pass Depth Peeling

Compute fragment color, z, etc..

Read N fragments from K-buffer

Enable pixel synchronization

```
void PSMain(...)
{
    IntelExt_Init();
    Fragment frag = {...};

    IntelExt_BeginPixelOrdering();

    Fragment fragArray[N] = gBuffer[xy];
```

K-Buffer: Single-Pass Depth Peeling

Compute fragment color, z, etc..

Read N fragments from K-buffer

Enable pixel synchronization

Bubble sort (1 pass)

```
void PSMain(...)
{
    IntelExt_Init();
    Fragment frag = {...};

    IntelExt_BeginPixelOrdering();

    Fragment fragArray[N] = gBuffer[xy];
    for (int i = 0; i < N; i++) {
       if (frag.Z < fragArray[i].Z) {
            Fragment temp = frag;
            frag          = fragArray[i];
            fragArray[i]  = temp;
        }
    }
```

K-Buffer: Single-Pass Depth Peeling

Compute fragment color, z, etc..

Read N fragments from K-buffer

Write N fragments to K-buffer

Enable pixel synchronization

Bubble sort (1 pass)

```
void PSMain(...)
{
    IntelExt_Init();
    Fragment frag = {...};

    IntelExt_BeginPixelOrdering();

    Fragment fragArray[N] = gBuffer[xy];
    for (int i = 0; i < N; i++) {
       if (frag.Z < fragArray[i].Z) {
           Fragment temp = frag;
           frag          = fragArray[i];
           fragArray[i]  = temp;
        }
    }
    gBuffer[xy] = fragArray;
}
```

Order-Independent Transparency

- Why order-independent transparency?
  - Correct compositing, rendering foliage & fences with zero aliasing ☺, etc..

Order-Independent Transparency

- Why order-independent transparency?
  - Correct compositing, rendering foliage & fences with zero aliasing ☺, etc..
- DX11-style order-independent transparency has significant drawbacks
  - Requires unbounded memory (per-pixel lists)
  - Not so great performance due to global atomics, fragments sorting, etc.

Order-Independent Transparency

- Why order-independent transparency?
  - Correct compositing, rendering foliage & fences with zero aliasing ☺, etc..
- DX11-style order-independent transparency has significant drawbacks
  - Requires unbounded memory (per-pixel lists)
  - Not so great performance due to global atomics, fragments sorting, etc.
- Pixel Synchronization enables new methods
  - Single geometry pass and fixed memory requirements
  - Stable and predictable performance
  - Scalable: easily trade-off image quality for performance/memory

A Recipe for Order-Independent Transparency

A Recipe for Order-Independent Transparency

- Step 1: Improve alpha-blending
  - Use depth to decide whether to composite incoming fragment over or under
  - Much better than vanilla alpha-blending but in some cases not quite correct

A Recipe for Order-Independent Transparency

- Step 1: Improve alpha-blending
  - Use depth to decide whether to composite incoming fragment over or under
  - Much better than vanilla alpha-blending but in some cases not quite correct

- Step 2: Make it even better by distributing the error over multiple terms
  - Store N layers per pixel & pick the "best" one when compositing incoming fragment
  - Use full screen pass to resolve data and blend resulting color over opaque color buffer

A Recipe for Order-Independent Transparency

- Step 1: Improve alpha-blending
  - Use depth to decide whether to composite incoming fragment over or under
  - Much better than vanilla alpha-blending but in some cases not quite correct

- Step 2: Make it even better by distributing the error over multiple terms
  - Store N layers per pixel & pick the “best” one when compositing incoming fragment
  - Use full screen pass to resolve data and blend resulting color over opaque color buffer

- Step 3: Use more layers to trade-off image quality for perf/memory

*Salvi et al. *"Adaptive Volumetric Shadow Maps"*. Computer Graphics Forum (Proceedings of EGSR 2010), vol. 29(4), pp. 1289-1296, June 2010.

Deep Shadow Maps

- DSMs encode per-pixel visibility function from light point-of-view
  - Typically used to render volumetric shadows
  - Developed by Pixar for off-line rendering, require unbounded memory

*Salvi et al. *"Adaptive Volumetric Shadow Maps"*. Computer Graphics Forum (Proceedings of EGSR 2010), vol. 29(4), pp. 1289-1296, June 2010.

Deep Shadow Maps

- DSMs encode per-pixel visibility function from light point-of-view
  - Typically used to render volumetric shadows
  - Developed by Pixar for off-line rendering, require unbounded memory

- Adaptive Volumetric Shadow Maps*
  - Like DSMs but designed for real-time rendering
  - Lossy compression of the visibility data

*Salvi et al. *"Adaptive Volumetric Shadow Maps"*. Computer Graphics Forum (Proceedings of EGSR 2010), vol. 29(4), pp. 1289-1296, June 2010.

Deep Shadow Maps

- DSMs encode per-pixel visibility function from light point-of-view
  - Typically used to render volumetric shadows
  - Developed by Pixar for off-line rendering, require unbounded memory
- Adaptive Volumetric Shadow Maps*
  - Like DSMs but designed for real-time rendering
  - Lossy compression of the visibility data
- Pixel synchronization enables first fixed memory implementation of AVSM
  - Demo ☺

Voxelization

- Build complex per-voxel data structures on the GPU at voxelization time
  - e.g. direction-dependent representations (anisotropic voxels, etc.)

Voxelization

- Build complex per-voxel data structures on the GPU at voxelization time
  - e.g. direction-dependent representations (anisotropic voxels, etc.)

- Voxelization via 2D rasterization projects triangles to XY, YZ or XZ plane
  - But global atomic ops are slow and pose significant restrictions on struct size, type, etc.

Voxelization

- Build complex per-voxel data structures on the GPU at voxelization time
  - e.g. direction-dependent representations (anisotropic voxels, etc.)
- Voxelization via 2D rasterization projects triangles to XY, YZ or XZ plane
  - But global atomic ops are slow and pose significant restrictions on struct size, type, etc.
- Use pixel synchronization to build 3D data structures at voxelization time
  - Problem: fragment dependencies cannot be tracked over multiple 2D planes

Voxelization

- Build complex per-voxel data structures on the GPU at voxelization time
  - e.g. direction-dependent representations (anisotropic voxels, etc.)
- Voxelization via 2D rasterization projects triangles to XY, YZ or XZ plane
  - But global atomic ops are slow and pose significant restrictions on struct size, type, etc.
- Use pixel synchronization to build 3D data structures at voxelization time
  - Problem: fragment dependencies cannot be tracked over multiple 2D planes
- Easy fix: voxelize onto one 2D plane at time
  - 3 draw calls per mesh, one per 2D plane (i.e. reject triangles that map to other planes)
  - Number of generated voxels doesn't change & more flexible than using global atomics

Advanced Anti-Aliasing

- Use pixel synchronization to improve or replace multi-sampling anti-aliasing
  - Higher image quality vs. lower memory requirements vs. better performance

*Jouppi et al. *"Z³: an economical hardware technique for high-quality antialiasing and transparency"*. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware

Advanced Anti-Aliasing

- Use pixel synchronization to improve or replace multi-sampling anti-aliasing
  - Higher image quality vs. lower memory requirements vs. better performance

- Z^3 anti-aliasing* (1999)
  - Originally developed as HW based high-quality anti-aliasing algorithm
    - Store N fragment per pixel (z, $\partial z/\partial x$, $\partial z/\partial y$, color, coverage)
    - Merge fragments (lossy)

*Jouppi et al. *"Z^3: an economical hardware technique for high-quality antialiasing and transparency"*. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware

Advanced Anti-Aliasing

- Use pixel synchronization to improve or replace multi-sampling anti-aliasing
  - Higher image quality vs. lower memory requirements vs. better performance

- Z^3 anti-aliasing* (1999)
  - Originally developed as HW based high-quality anti-aliasing algorithm
    - Store N fragment per pixel (z, $\partial z/\partial x$, $\partial z/\partial y$, color, coverage)
    - Merge fragments (lossy)

- Analytic methods
  - Render scene using conservative rasterization
  - Build per-pixel spatial subdivision structure using primitive edges (per-pixel BSP? ☺)
  - Compute fragment weights from fraction of pixel area covered by leaf cells and resolve

Performance Tips & Tricks

- Don't clear large buffers. Clear a small buffer and use it as a clear mask.

Performance Tips & Tricks

- Don't clear large buffers. Clear a small buffer and use it as a clear mask.

Read clear mask

```
bool clear = gClearMask[xy];
```

Performance Tips & Tricks

- Don't clear large buffers. Clear a small buffer and use it as a clear mask.

Read clear mask

```
bool clear = gClearMask[xy];

if (clear) {
    gClearMask[xy] = false;
    myLargeStruct  = ...
```

Mark pixel as "used" and initialize large struct

Performance Tips & Tricks

- Don't clear large buffers. Clear a small buffer and use it as a clear mask.

Read clear mask

```
bool clear = gClearMask[xy];

if (clear) {
    gClearMask[xy] = false;
    myLargeStruct  = ...
} else {
    myLargeStruct  = gLargeDataStruct[xy];
    ...
}
```

Mark pixel as "used" and initialize large struct

If pixel is not in clear state load large struct and update it

Performance Tips & Tricks

- Don’t clear large buffers. Clear a small buffer and use it as a clear mask.

Read clear mask

```
bool clear = gClearMask[xy];

if (clear) {
    gClearMask[xy] = false;
    myLargeStruct  = ...
} else {
    myLargeStruct  = gLargeDataStruct[xy];
    ...
}

gLargeDataStruct[xy] = myStruct;
```

Mark pixel as “used” and initialize large struct

If pixel is not in clear state load large struct and update it

Write large struct data back to memory

Performance Tips & Tricks

- Don't clear large buffers. Clear a small buffer and use it as a clear mask.

Read clear mask

Clear this!

Mark pixel as "used" and initialize large struct

If pixel is not in clear state load large struct and update it

Write large struct data back to memory

```
bool clear = gClearMask[xy];

if (clear) {
    gClearMask[xy] = false;
    myLargeStruct  = ...
} else {
    myLargeStruct  = gLargeDataStruct[xy];
    ...
}

gLargeDataStruct[xy] = myStruct;
```

Performance Tips & Tricks

- Don't clear large buffers. Clear a small buffer and use it as a clear mask.

Read clear mask

Clear this!

Mark pixel as "used" and initialize large struct

```
bool clear = gClearMask[xy];

if (clear) {
    gClearMask[xy] = false;
    myLargeStruct  = ...
} else {
    myLargeStruct  = gLargeDataStruct[xy];
    ...
}

gLargeDataStruct[xy] = myStruct;
```

If pixel is not in clear state load large struct and update it

Write large struct data back to memory

Not this!

Performance Tips & Tricks

- Small(er) data structures can improve performance
  - Use more instructions to pack/unpack data
  - Balance data structure size and amount of packing/unpacking code

Performance Tips & Tricks

- Small(er) data structures can improve performance
  - Use more instructions to pack/unpack data
  - Balance data structure size and amount of packing/unpacking code

- Address 1D structured buffers as tiled to better data exploit locality
  - e.g. 1x2 or 2x2 (2D textures), 2x2x2 (voxels), etc..

Performance Tips & Tricks

- Small(er) data structures can improve performance
  - Use more instructions to pack/unpack data
  - Balance data structure size and amount of packing/unpacking code

- Address 1D structured buffers as tiled to better data exploit locality
  - e.g. 1x2 or 2x2 (2D textures), 2x2x2 (voxels), etc..

- Prefer inserting the synchronization point in the second half of the shader
  - Increase likelihood of concurrently shading fragments that map to the same pixel
  - Corollary: use HW z-test when possible for better performance (Hi-Z is fast!)

Summary

- Programmable shading revolutionized real-time rendering
  - ..but the revolution did not include the tail of the pipeline

Summary

- Programmable shading revolutionized real-time rendering
  - ..but the revolution did not include the tail of the pipeline

- Pixel synchronization is a new tool that injects new life in the 3D pipeline

Summary

- Programmable shading revolutionized real-time rendering
  - ..but the revolution did not include the tail of the pipeline

- Pixel synchronization is a new tool that injects new life in the 3D pipeline
  1. Pick the per-pixel data structure that can better solve your rendering problem

Summary

- Programmable shading revolutionized real-time rendering
  - ..but the revolution did not include the tail of the pipeline

- Pixel synchronization is a new tool that injects new life in the 3D pipeline
  1. Pick the per-pixel data structure that can better solve your rendering problem
  2. Draw geometry to build your data in a streaming fashion

Summary

- Programmable shading revolutionized real-time rendering
  - ..but the revolution did not include the tail of the pipeline

- Pixel synchronization is a new tool that injects new life in the 3D pipeline
  1. Pick the per-pixel data structure that can better solve your rendering problem
  2. Draw geometry to build your data in a streaming fashion
  3. Use the data & enjoy your results (sip tea or coffee ☺)

Summary

- Programmable shading revolutionized real-time rendering
  - ..but the revolution did not include the tail of the pipeline

- Pixel synchronization is a new tool that injects new life in the 3D pipeline
  1. Pick the per-pixel data structure that can better solve your rendering problem
  2. Draw geometry to build your data in a streaming fashion
  3. Use the data & enjoy your results (sip tea or coffee ☺)

- DX11+ extension available now (download demos), OpenGL extension in development.

Q&A

- Acknowledgements
  - The ART team
  - Tom Piazza, Chuck Lingle, Tomasz Janczak , Prasoon Surti, Mike Dwyer, Andy Dayton, Mike Apodaca, Aaron Lefohn, Larry Seiler, Leigh Davies, Filip Strugar, Matthew Fife, Steve Hughes, Axel Mamode, Richard Huddy and many others

- Source code
  - Programmable Blending: bit.ly/pixelsync_pb
  - Order-Independent Transparency: bit.ly/pixelsync_oit
  - Adaptive Volumetric Shadow Maps: bit.ly/pixelsync_avsm

- Contacts
  - e-mail: marco.salvi@intel.com
  - twitter: @marcosalvi