DirectX 11 Rendering in Battlefield 3
Johan Andersson, Rendering Architect, DICE
Agenda

Overview

Feature:
- Deferred Shading
- Compute Shader Tile-Based Lighting
- Terrain Displacement Mapping
- Direct Stereo 3D rendering

Quality:
- Antialiasing: MSAA
- Antialiasing: FXAA
- Antialiasing: SRAA
- Transparency Supersampling

Performance:
- Instancing
- Parallel dispatch
- Multi-GPU
- Resource Streaming

Conclusions

Q & A
Battlefield 3

› FPS

› Fall 2011

› DX11, Xbox 360 and PS3

› Frostbite 2 engine
Frostbite 2

› Developed for Battlefield 3 and future DICE/EA games

› Massive focus on creating simple to use & powerful workflows

› Major pushes in animation, destruction, streaming, rendering, lighting and landscapes
DX11 API only
 › Requires a DX10 or DX11 GPUs
 › Requires Vista SP1 or Windows 7
 › No Windows XP!

Why?
 › CPU performance win
 › GPU performance & quality win
 › Ease of development - no legacy
 › Future proof

BF3 is a big title - will drive OS & HW adoption
 › Which is good for your game as well! 😊
Options for rendering

Switched to Deferred Shading in FB2
› Rich mix of Outdoor + Indoor + Urban environments in BF3
› Wanted lots more light sources

Why not Forward Rendering?
› Light culling / shader permutations not efficient for us
› Expensive & more difficult decaling / destruction masking

Why not Light Pre-pass?
› 2x geometry pass too expensive on both CPU & GPU for us
› Was able to generalize our BRDF enough to just a few variations
› Saw major potential in full tile-based deferred shading

See also:
› Nicolas Thibieroz’s talk ”Deferred Shading Optimizations”
Deferred Shading

Weaknesses with traditional deferred lighting/shading:

- Massive overdraw & ROP cost when having lots of big light sources
- Expensive to have multiple per-pixel materials in light shaders
- MSAA lighting can be slow (non-coherent, extra BW)
FEATURES
1. Divide screen into tiles and determine which lights affects which tiles

2. Only apply the visible light sources on pixels
   › Custom shader with multiple lights
   › Reduced bandwidth & setup cost

How can we do this best in DX11?
Lighting with Compute Shader

Tile-based Deferred Shading using Compute Shaders

Primarily for analytical light sources
  › Point lights, cone lights, line lights
  › No shadows
  › Requires Compute Shader 5.0

Hybrid Graphics/Compute shading pipeline:
  › Graphics pipeline rasterizes gbuffers for opaque surfaces
  › Compute pipeline uses gbuffers, culls lights, computes lighting & combines with shading
  › Graphics pipeline renders transparent surfaces on top
CS requirements & setup

1 thread per pixel, 16x16 thread groups (aka tile)

**Input:** gbuffers, depth buffer & list of lights

**Output:** fully composited & lit HDR texture

Texture2D<float4> gbufferTexture0 : register(t0);
Texture2D<float4> gbufferTexture1 : register(t1);
Texture2D<float4> gbufferTexture2 : register(t2);
Texture2D<float4> depthTexture : register(t3);

RWTexture2D<float4> outputTexture : register(u0);

#define BLOCK_SIZE 16
[numthreads(BLOCK_SIZE,BLOCK_SIZE,1)]
void csMain(
uint3 groupId : SV_GroupID,
uint3 groupThreadId : SV_GroupThreadID,
uint groupIndex: SV_GroupIndex,
uint3 dispatchThreadId : SV_DispatchThreadID)
{
    ...
}

normal

Diffuse Albedo

Roughness

Specular Albedo
1. Load gbuffers & depth

2. Calculate min & max z in threadgroup / tile
   › Using InterlockedMin/Max on groupshared variable
   › Atomics only work on ints 😞
   › Can cast float to int (z is always +)

```c
// *** globals above, function below ******
float depth = depthTexture.Load(uint3(texCoord, 0)).r;
uint depthInt = asuint(depth);
minDepthInt = 0xFFFFFFFF;
maxDepthInt = 0;
GroupMemoryBarrierWithGroupSync();
InterlockedMin(minDepthInt, depthInt);
InterlockedMax(maxDepthInt, depthInt);
GroupMemoryBarrierWithGroupSync();
float minGroupDepth = asfloat(minDepthInt);
float maxGroupDepth = asfloat(maxDepthInt);
```
CS step 3 – Culling

Determine visible light sources for each tile
 › Cull all light sources against tile frustum

Input (global):
 › Light list, frustum & SW occlusion culled

Output (per tile):
 › # of visible light sources
 › Index list of visible light sources

<table>
<thead>
<tr>
<th>Lights</th>
<th>Indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global list</td>
<td>1000+</td>
</tr>
<tr>
<td>Tile visible</td>
<td>~0-40+</td>
</tr>
</tbody>
</table>

Per-tile visible light count (black = 0 lights, white = 40)
CS step 3 – Impl

3a. Each thread switches to process lights instead of pixels
   › Wow, parallelism switcharoo!
   › 256 light sources in parallel
   › Multiple iterations for >256 lights

3b. Intersect light and tile
   › Multiple variants – accuracy vs perf
     Tile min & max z is used as a ”depth bounds” test

3c. Append visible light indices to list
   › Atomic add to threadgroup shared memory
     ”inlined stream compaction”

3d. Switch back to processing pixels
   › Synchronize the thread group
   › We now know which light sources affect the tile

```c
struct Light {
    float3 pos; float sqrRadius;
    float3 color; float invSqrRadius;
};
int lightCount;
StructuredBuffer<Light> lights;

groupshared uint visibleLightCount = 0;
groupshared uint visibleLightIndices[1024];

// --- globals above, cont. function below ---

uint threadCount = BLOCK_SIZE*BLOCK_SIZE;
uint passCount = (lightCount+threadCount-1) / threadCount;
for (uint passIt = 0; passIt < passCount; ++passIt) {
    uint lightIndex = passIt*threadCount + groupIndex;

    // prevent overrun by clamping to a last ”null” light
    lightIndex = min(lightIndex, lightCount);
    if (intersects(lights[lightIndex], tile))
    {
        uint offset;
        InterlockedAdd(visibleLightCount, 1, offset);
        visibleLightIndices[offset] = lightIndex;
    }
}
GroupMemoryBarrierWithGroupSync();
```
4. For each pixel, accumulate lighting from visible lights
   › Read from tile visible light index list in groupshared memory

5. Combine lighting & shading albedos
   › Output is non-MSAA HDR texture
   › Render transparent surfaces on top

```cpp
float3 color = 0;
for (uint lightIt = 0; lightIt < visibleLightCount; ++lightIt) {
    uint lightIndex = visibleLightIndices[lightIt];
    Light light = lights[lightIndex];

    color += diffuseAlbedo * evaluateLightDiffuse(light, gbuffer);
    color += specularAlbedo * evaluateLightSpecular(light, gbuffer);
}
```
Only edge pixels need full per-sample lighting
› But edges have bad screen-space coherency! Inefficient

Compute Shader can build efficient coherent pixel list
› Evaluate lighting for each pixel (sample 0)
› Determine if pixel requires per-sample lighting
› If so, add to atomic list in shared memory
› When all pixels are done, synchronize
› Go through and light sample 1-3 for pixels in list

Major performance improvement!
› Described in detail in [Lauritzen10]
Terrain rendering

GDC 2011 pre-alpha
Terrain rendering

Battlefield 3 terrains
› Huge area & massive view distances
› Dynamic destructible heightfields
› Procedural virtual texturing
› Streamed heightfields, colormaps, masks
› Full details at a later conference

We stream in source heightfield data at close to pixel ratio
› Derive high-quality per-pixel normals in shader

How can we increase detail even further on DX11?
› Create better silhouettes and improved depth
› Keep small-scale detail (dynamic craters & slopes)
GDC 2011 pre-alpha
Normal mapped terrain
Terrain Displacement Mapping

Straight high-res heightfields, no procedural detail
› Lots of data in the heightfields
› Pragmatic & simple choice
› No changes in physics/collisions
› No content changes, artists see the true detail they created

Uses DX11 fixed edge tessellation factors
› Stable, no swimming vertices
› Though can be wasteful
› Height morphing for streaming by fetching 2 heightfields in domain shader & blend based on patch CLOD factor

More work left to figure optimal tessellation scheme for our use case
Stereo 3D rendering in DX11

*Nvidia’s 3D Vision* drivers is a good and *almost* automatic stereo 3D rendering method
› But only for forward rendering, doesn’t work with deferred shading
› Or on AMD or Intel GPUs
› Transparent surfaces do not get proper 3D depth

We instead use *explicit* 3D stereo rendering
› Render unique frame for each eye
› Works with deferred shading & includes all surfaces
› Higher performance requirements, 2x draw calls

Works with *Nvidia’s 3D Vision* and *AMD’s HD3D*
› Similar to OpenGL quad buffer support
› Ask your friendly IHV contact how
Instancing

Draw calls can still be major performance bottleneck
› Lots of materials / lots of variation
› Complex shadowmaps
› High detail / long view distances
› Full 3D stereo rendering

Battlefield have lots of use cases for heavy instancing
› Props, foliage, debris, destruction, mesh particles

Richard Huddy: “Batch batch batch!”

Batching submissions is still important, just as before!
Instancing in DirectX

DX9-style stream instancing is good, but restrictive
› Extra vertex attributes, GPU overhead
› Can’t be (efficiently) combined with skinning
› Used primarily for tiny meshes (particles, foliage)

DX10/DX11 brings support for shader Buffer objects
› Vertex shaders have access to SV_InstanceID
› Can do completely arbitrary loads, not limited to fixed elements
› Can support per-instance arrays and other data structures!

Let’s rethink how instancing can be implemented..
Instancing data

Multiple object types
› Rigid / skinned / composite meshes

Multiple object lighting types
› Small/dynamic: light probes
› Large/static: light maps

Different types of instancing data we have
› Transform: float4x3
› Skinning transforms: float4x3 array
› SH light probe: float4 x 4
› Lightmap UV scale/offset: float4

Let’s pack all instancing data into single big buffer!
Buffers are allocated for storing arrays of data that are to be processed as a single unit. In this example, we allocate a buffer for storing vector data associated with each instance:

```
Buffer<float4> instanceVectorBuffer : register(t0);
```

A constant buffer (cbuffer) is then defined to hold the start vector and the number of vectors per instance:

```
cbuffer a
{
    float g_startVector;
    float g_vectorsPerInstance;
}
```

In the `VsOutput` function, we use the `uint instanceId : SV_InstanceId)` to access the instance vector buffer.

```
VsOutput main(
    // ....
    uint instanceId : SV_InstanceId)
{
    uint worldMatrixVectorOffset = g_startVector + input.instanceId * g_vectorsPerInstance + 0;
    uint probeVectorOffset       = g_startVector + input.instanceId * g_vectorsPerInstance + 3;

    float4 r0 = instanceVectorBuffer.Load(worldMatrixVectorOffset + 0);
    float4 r1 = instanceVectorBuffer.Load(worldMatrixVectorOffset + 1);
    float4 r2 = instanceVectorBuffer.Load(worldMatrixVectorOffset + 2);
    float4 lightProbeShR = instanceVectorBuffer.Load(probeVectorOffset + 0);
    float4 lightProbeShG = instanceVectorBuffer.Load(probeVectorOffset + 1);
    float4 lightProbeShB = instanceVectorBuffer.Load(probeVectorOffset + 2);
    float4 lightProbeShO = instanceVectorBuffer.Load(probeVectorOffset + 3);

    // ....
}
```
half4 weights = input.boneWeights;
int4 indices = (int4)input.boneIndices;

float4 skinnedPos = mul(float4(pos,1), getSkinningMatrix(indices[0])).xyz * weights[0];
skinnedPos += mul(float4(pos,1), getSkinningMatrix(indices[1])).xyz * weights[1];
skinnedPos += mul(float4(pos,1), getSkinningMatrix(indices[2])).xyz * weights[2];
skinnedPos += mul(float4(pos,1), getSkinningMatrix(indices[3])).xyz * weights[3];

// ...

float4x3 getSkinningMatrix(uint boneIndex)
{
    uint vectorOffset = g_startVector + instanceId * g_vectorsPerInstance;
    vectorOffset += boneIndex*3;
    float4 r0 = instanceVectorBuffer.Load(vectorOffset + 0);
    float4 r1 = instanceVectorBuffer.Load(vectorOffset + 1);
    float4 r2 = instanceVectorBuffer.Load(vectorOffset + 2);
    return createMat4x3(r0, r1, r2);
}
Instancing benefits

Single draw call per object type instead of per instance
  › Minor GPU hit for big CPU gain

Instancing does not break when skinning parts
  › More deterministic & better overall performance

End result is typically 1500-2000 draw calls
  › Regardless of how many object instances the artists place!
  › Instead of 3000-7000 draw calls in some heavy cases
Great key DX11 feature!
› Improve performance by scaling dispatching to D3D to more cores
› Reduce frame latency

How we use it:
› DX11 deferred context per HW thread
› Renderer builds list of all draw calls we want to do for each rendering ”layer” of the frame
› Split draw calls for each layer into chunks of ~256
› Dispatch chunks in parallel to the deferred contexts to generate command lists
› Render to immediate context & execute command lists
› Profit! *

* but theory != practice
Parallel Dispatch in Practice

Still no performant drivers available for our use case 😞
› Have waited for 2 years and still are
› Big driver codebases takes time to refactor
› IHVs vs Microsoft quagmire
› Heavy driver threads collide with game threads

How it should work (an utopia?)
› Driver does not create any processing threads of its own
› Game submits workload in parallel to multiple deferred contexts
› Driver make sure almost all processing required happens on the
draw call on the deferred context
› Game dispatches command list on immediate context, driver does
absolute minimal work with it

Still good to design engine for + instancing is great!
Resource streaming

Even with modern GPUs with lots of memory, resource streaming is often required
› Can’t require 1+ GB graphics cards
› BF3 levels have much more than 1 GB of textures & meshes
› Reduced load times

But creating & destroying DX resources in-frame has never been a good thing
› Can cause non-deterministic & large driver / OS stalls 😞
› Has been a problem for a very long time in DX
› About time to fix it
DX11 Resource Streaming

Have worked with Microsoft, Nvidia & AMD to make sure we can do stall free async resource streaming of GPU resources in DX11

› Want neither CPU nor GPU perf hit

› Key foundation: DX11 concurrent creates

Resource creation flow:
› Streaming system determines resources to load (texture mipmaps or mesh LODs)
› Add up DX resource creation on to queue on our own separate low-priority thread
› Thread creates resources using initial data, signals streaming system
› Resource created, game starts using it

Enables async stall-free DMA in drivers!

Resource destruction flow:
› Streaming system deletes D3D resource
› Driver keeps it internally alive until GPU frames using it are done. NO STALL!
Efficiently supporting Crossfire and SLI is important for us
› High-end consumers expect it
› IHVs expect it (and can help!)
› Allows targeting higher-end HW then currently available during dev

AFR is easy: Do not reuse GPU resources from previous frame!
› UpdateSubResource is easy & robust to use for dynamic resources, but not ideal

All of our playtests run with exe named AFR-FriendlyD3D.exe
› Disables all driver AFR synchronization workarounds
› Rather find corruption during dev then have bad perf
› ForceSingleGPU.exe is also useful to track down issues
Reducing aliasing is one of our key visual priorities
› Creates a more smooth gameplay experience
› Extra challenging goal due to deferred shading

We use multiple methods:
› MSAA – Multisample Antialiasing
› FXAA – Fast Approximate Antialiasing
› SRAA – Sub-pixel Reconstruction Antialiasing
› TSAA – Transparency Supersampling Antialiasing
Our solution:
  › Deferred geometry pass renders with MSAA (2x, 4x or 8x)
  › Light shaders evaluate per-sample (when needed), averages the samples and writes out per-pixel
  › Transparent surfaces rendered on top without MSAA

1080p gbuffer+z with 4x MSAA is 158 MB
  › Lots of memory and lots of bandwidth 😞
  › Could be tiled to reduce memory usage
  › Very nice quality though 😊

Our (overall) highest quality option
  › But not fast enough for more GPUs
  › Need additional solution(s)...
"Fast Approximate Antialiasing"
› GPU-based MLAA implementation by Timothy Lottes (Nvidia)
› Multiple quality options
› ~1.3 ms/f for 1080p on Geforce 580

Pros & cons:
› Superb antialiased long edges! 😊
› Smooth overall picture 😊
› Reasonably fast 😊
› Moving pictures do not benefit as much 😞
› "Blurry aliasing" 😞

Will be released here at GDC’11
› Part of Nvidia’s example SDK
"Sub-pixel Reconstruction Antialiasing"

Presented at I3D’11 2 weeks ago [Chajdas11]
Use 4x MSAA buffers to improve reconstruction

Multiple variants:
- MSAA depth buffer
- MSAA depth buffer + normal buffer
- MSAA Primitive ID / spatial hashing buffer

Pros:
- Better at capturing small scale detail 😊
- Less "pixel snapping" than MLAA variants 😊

Cons:
- Extra MSAA z/normal/id pass can be prohibitive 😞
- Integration not as easy due to extra pass 😞
No antialiasing
MSAA Sample Coverage

None of the AA solutions can solve all aliasing
› Foliage & other alpha-tested surfaces are extra difficult cases
› Undersampled geometry, requires sub-samples

DX 10.1 added `SV_COVERAGE` as a pixel shader `output`
DX 11 added `SV_COVERAGE` as a pixel shader `input`

What does this mean?
› We get full programmable control over the coverage mask
› No need to waste the alpha channel output (great for deferred)
› We can do partial supersampling on alpha-tested surfaces!
Shade per-pixel but evaluate alpha test per-sample
- Write out coverage bitmask
- MSAA offsets are defined in DX 10.1
- Requires shader permutation for each MSAA level

Gradients still quite limited
- But much better than standard MSAA! 😊
- Can combine with screen-space dither

See also:
- DirectX SDK ’TransparencyAA10.1
- GDC’09 STALKER talk [Lobanchikov09]
Alpha testing

4x MSAA + Transparency Supersampling
Conclusions

DX11 is here – in force
› 2011 is a great year to focus on DX11
› 2012 will be a great year for more to drop DX9

We’ve found lots & lots of quality & performance enhancing features using DX11
› And so will you for your game!
› Still have only started, lots of potential

Take advantage of the PC strengths, don’t hold it back
› Big end-user value
› Good preparation for Gen4
Thanks

› Christina Coffin (@ChristinaCoffin)
› Mattias Widmark
› Kenny Magnusson
› Colin Barré-Brisebois (@ZigguratVertigo)
› Timothy Lottes (@TimothyLottes)
› Matthäus G. Chajdas (@NIV_Anteru)
› Miguel Sainz
› Nicolas ThibieroZ

› Battlefield team
› Frostbite team
› Microsoft
› Nvidia
› AMD
› Intel
## Battlefield 3 & Frostbite 2 talks at GDC’11:

<table>
<thead>
<tr>
<th>Time</th>
<th>Title</th>
<th>Speaker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 1:45</td>
<td><strong>DX11 Rendering in Battlefield 3</strong></td>
<td>Johan Andersson</td>
</tr>
<tr>
<td>Wed 10:30</td>
<td><strong>SPU-based Deferred Shading in Battlefield 3 for PlayStation 3</strong></td>
<td>Christina Coffin</td>
</tr>
<tr>
<td>Wed 3:00</td>
<td><strong>Culling the Battlefield: Data Oriented Design in Practice</strong></td>
<td>Daniel Collin</td>
</tr>
<tr>
<td>Thu 1:30</td>
<td><strong>Lighting You Up in Battlefield 3</strong></td>
<td>Kenny Magnusson</td>
</tr>
<tr>
<td>Fri 4:05</td>
<td><strong>Approximating Translucency for a Fast, Cheap &amp; Convincing Subsurface Scattering Look</strong></td>
<td>Colin Barré-Brisebois</td>
</tr>
</tbody>
</table>

Email: repi@dice.se  
Blog: http://repi.se  
Twitter: @repi  

For more DICE talks: http://publications.dice.se
References

› [Lobanchikov09] Igor A. Lobanchikov, ”GSC Game World’s STALKER: Clear Sky – a showcase for Direct3D 10.0/1” GDC’09. http://developer.amd.com/gpu_assets/01gdc09ad3ddstalkerclearsky210309.ppt
