Assassin's Creed 4: Black Flag
Road to next-gen graphics

Bartłomiej Wronski
3D Programmer, Ubisoft Montreal
Presentation overview

- Deferred Normalized Irradiance Probes
- Volumetric Fog
- Screen Space Reflections
- Next-gen Performance and Optimizations
Goals

- Improve AC3 ambient lighting – flat, uniform
- Partially baked solution
- Work on current gen (~1ms / < 1MB for GPU)
- Dynamic weather / time of day
- Small impact on art pipelines
Background

- One key light
- Weather has no influence on light direction
- Small amount of local lights

Deferred Normalized Irradiance Probes
Data storage

- **8-bit** RGB normalized irradiance
- **8** key-framed values a day
- **4** basis vectors (FC3 basis)
- Uniform grid **2m x 2m**
- Only one layer
  - **2.5D** world layout / structure
Deferred Normalized Irradiance Probes

**Offline**
- On GPU bake sunlight bounce irradiance
- Store irradiance at 8 different hours
- Compute 2D VRAM textures (many lightprobes)

**Runtime**
- De-normalize and blend irradiances
- Blend out bounced lighting with height
- Combine with indirect sky lighting and AO
Ambient cube
Sun light bounce
Ambient cube – comparison
Benchmarks and summary

<table>
<thead>
<tr>
<th>Description</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU performance cost</td>
<td>1.2ms fullscreen pass - PS3</td>
</tr>
<tr>
<td>Memory cost (probe data)</td>
<td>600kb (VRAM only)</td>
</tr>
<tr>
<td>Memory cost (render targets)</td>
<td>56kb</td>
</tr>
<tr>
<td>CPU cost</td>
<td>0.6ms (amortized)</td>
</tr>
<tr>
<td>Num probes in Havana bruteforce</td>
<td>~110 000</td>
</tr>
<tr>
<td>Num probes in Havana trimmed</td>
<td>~30 000</td>
</tr>
<tr>
<td>Full baking time for Havana</td>
<td>8 minutes (nVidia GTX 680, one machine)</td>
</tr>
</tbody>
</table>
Volumetric Fog
No light scattering
Light scattering

In-scattering

Out-scattering
Light scattering

- Intensity of effect depends on media, distance, light angle, weather conditions
- Problem difficult to solve (integration)
- Approximations used by the industry since 90s

<table>
<thead>
<tr>
<th>Post-process god-rays</th>
<th>Distance based fog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Billboard light-shafts</td>
<td>Volumetric shadows</td>
</tr>
</tbody>
</table>
Inspiration

- Kaplanyan, "Light Propagation Volumes", Siggraph 2009
Shadow cascades → CS: Shadowmap downsample & blur → ESM

CS: Density estimation and volume lighting

CS: Solving scattering equation

Depth buffer
Color buffer

CS: Density & in-scattering
Accumulated scattering

PS: Apply fog

Final color buffer
Volume shadowing technique

- 4 shadow cascades 1k x 1k
  - Too much detail
  - Shadowing above volume Nyquist frequency
  - Lots of aliasing, flickering
  - Needed to apply low-pass filter
  - Naïve 32-tap PCF = unacceptable performance
Volume shadowing technique

- Exponential Shadow Maps
  - Do not compare depths for testing
  - Estimate shadowing probability
  - Efficient to compute shadowing test
  - Code snippets in bonus slides!

Source: Annen et al, “Exponential Shadow Maps”
Volume shadowing technique

- Exponential Shadow Maps
  - Can be down-sampled!
  - 256x256 R32F cascades
  - Can be filtered (separable blur)
  - One disadvantage – shadow leaking
    - Negligible in participating media

Source: Annen et al, “Exponential Shadow Maps”
Shadow cascades → CS: Shadowmap downsample & blur → ESM

CS: Density estimation and volume lighting

CS: Solving scattering equation

Depth buffer → PS: Apply fog → Final color buffer

Color buffer

Density & in-scattering

Accumulated scattering
Volume data layout

RGB = in-scattered light color, A = media density
Volume resolution – too low?

- We store information for whole view ray
- And for every depth along it – tex3D filtering
- Every 1080p pixel gets proper information
- No edge artifacts!
- Soft result
Density estimation and volume lighting

- Fog density estimation
  - Procedural Perlin noise animated by wind
  - Vertical attenuation

- Lighting in-scattering
  - ESM shadowing for the main light
  - Constant ambient term
  - Loop over point lights
Density estimation and volume lighting

- Lighting in-scattering phase function
  - Not physically based (art driven instead) – 2 colors (sun direction, opposite direction)
Shadow cascades

CS: Density estimation and volume lighting

CS: Solving scattering equation

CS: Shadowmap downsample & blur

ESM

Density & in-scattering

Accumulated scattering

Depth buffer

Color buffer

PS: Apply fog

Final color buffer
Solving scattering equation

**Beer-Lambert Law**

\[ T_{(A \rightarrow B)} = e^{-\int_A^B \text{density}(x)dx} \]

RGB = in-scattering

A = out-scattering multiplier
Solving scattering equation

- 2D compute shader
- Brute-force, numerical integration
- Marching through depth slices and accumulating
- Using UAV writes
- Front to back order
  - More scattering with distance
Solving scattering equation

- Apply equation from Beer-Lambert’s law

```c
// One step of numerical solution to the light scattering equation
float4 AccumulateScattering(float4 colorAndDensityFront, float4 colorAndDensityBack)
{
    // rgb = light in-scattered accumulated so far, a = accumulated density
    float3 light = colorAndDensityFront.rgb + saturate(exp(-colorAndDensityFront.a)) * colorAndDensityBack.rgb;
    return float4(light.rgb, colorAndDensityFront.a + colorAndDensityBack.a);
}
```

One step of iterative numerical solution to the scattering equation

```c
// Writing out final scattering values
void WriteOutput(in uint3 pos, in float4 colorAndDensity)
{
    // final value rgb = light in-scattered accumulated so far, a = scene color decay caused by out-scattering
    float4 finalValue = float4(colorAndDensity.rgb, 1.0f - exp(-colorAndDensity.a));
    g_outputUAV[pos].rgba = finalValue;
}
```

Writing out final scattering values
## Performance

**On Microsoft XboxOne**

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total cost</td>
<td>1.1ms</td>
</tr>
<tr>
<td>Shadowmap downsample</td>
<td>0.163ms</td>
</tr>
<tr>
<td>Shadowmap blur</td>
<td>0.177ms</td>
</tr>
<tr>
<td>Lighting volume and building densities</td>
<td>0.43ms</td>
</tr>
<tr>
<td>Solving scattering equation</td>
<td>0.116ms</td>
</tr>
<tr>
<td>Applying on screen (can be combined)</td>
<td>0.247ms</td>
</tr>
</tbody>
</table>
Summary

- Robust and efficient
- Compatible with deferred and forward
  - Dependent only on shadowmaps, not on scene
  - Only last step depends on final screen information
- Multiple extensions possible
  - Every component can be swapped separately!
  - Artist authored / particle injected densities
  - Density maps
  - Physically based phase functions
Screen Space Reflections
Screen-space reflections

- Any 3D oriented point can be reflector
- No additional pass
  - No CPU / GPU per-object cost
  - Can be easily integrated in the engine
- Animated and dynamic objects
- Glossy / approximate reflections
- Good occlusion source for specular cube maps
Screenspace reflections

CS: Find “interesting” areas and compute the reflection mask

Half resolution buffers
- Color and depth buffer
- Reflection mask

CS: Do a precise raymarching in masked areas

PS: Perform a separable blur according to glossiness

Raytracing result
Blurred reflections
Screenspace reflections
Creating reflection mask

Sampling pattern for 64x64 block
Screenspace reflections

CS: Find “interesting” areas and compute the reflection mask

CS: Do a precise ray marching in masked areas

PS: Perform a separable blur according to glossinessness

Half resolution buffers

- Color and depth buffer
- Reflection mask
- Raytracing result
- Blurred reflections
 Screenspace reflections

**CS:** Find “interesting” areas and compute the reflection mask

**CS:** Do a precise raymarching in masked areas

**PS:** Perform a separable blur according to glossiness

**Half resolution buffers**
- Color and depth buffer
- Reflection mask
- Raytracing result
- Blurred reflections
Screenspace reflections blur and “push-pull” pass
# Performance

On Microsoft XboxOne

<table>
<thead>
<tr>
<th>Description</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total (worst case, fully reflective scene)</td>
<td>~2ms</td>
</tr>
<tr>
<td>Total (average scene)</td>
<td>~1ms</td>
</tr>
<tr>
<td>PS: Downsampling</td>
<td>0.1ms</td>
</tr>
<tr>
<td>CS: Rays mask</td>
<td>0.16ms</td>
</tr>
<tr>
<td>CS: Raytracing</td>
<td>0.29ms</td>
</tr>
<tr>
<td>PS: Separable blur</td>
<td>0.28ms</td>
</tr>
<tr>
<td>PS: Apply on screen</td>
<td>0.21ms</td>
</tr>
</tbody>
</table>
PS4 & XboxOne GPU Optimizations
PS4 and XboxOne GPUs

- Advanced GPU architectures...
- Lots of custom extensions
- Capabilities not available on PCs
- ...but both based on AMD GCN architecture!
- AMD Southern / Sea Islands ISA publicly available
“Usual” optimizations

- Current gen optimizations are still important
  - Reduce amount of total work - **resolution**
  - Reduce work done - **instructions**
  - Reduce used bandwidth - **resources**
  - Maximize instruction pipelining – **micro-optimizations**
PS4/XboxOne specific

- All of those still apply...
- ...but GPU is not an array of huge number of simple processors
- AMD GCN architecture is way more complicated!
AMD GCN GPU block diagram

Source: “Southern Islands Series Instruction Set Architecture”, AMD
AMD GCN GPU Compute Unit

Source: “Southern Islands Series Instruction Set Architecture”, AMD
Wavefronts / waves

- **Up to** 10 running on a SIMD on CU
- 64 work items
- Pixels or compute threads
- Simplest operations take 4 cycles
- But with 4 SIMDs you get 1 cycle per op
Wavefront occupancy

- Only 1 vector ALU operation on 1 wave on a SIMD, no parallel ALU operations
- Why do we need bigger occupancy?
- Scalar operations in parallel
- ...but a wave can also be stalled
- ...and wait for the results of a memory (texture / buffer / LDS) operation!
Wavefront pipelining

- Big latency of memory operations
- Possibly up to 800 cycles! (L2 cache miss)
- Much higher occupancy needed to hide it
  - One wave waits for results of a texture / buffer fetch...
  - ...other waves can be at different instruction pointer and do some ALUs!
  - ...you need to have proper ALU to MEM operations ratio though
  - Can achieve perfect pipelining and parallelism
Wavefront pipelining

- Number of active waves per SIMD 1 to 10
- Determined by available resources
- All waves must share
  - 512 Scalar GPRs, 256 Vector GPRs
  - Over 64 VGPRs used = occupancy under 4!
  - 16kB L1 cache, 64kB Local Data Storage (LDS)
  - Texturing units etc.
Scalar vs vector registers

**Vector register**

- Is not “float4 vectorVariable;”!
  - float4 is 4 vector registers!
- “Superscalar” architecture
- One vector per wavefront
- Vector register = 64 values
- Potentially different value for each work item
- Used for regular ALU operations
Scalar vs vector registers

**Scalar register**

- Is not “float variable;”
  - which is 1 vector register!
- Everything common to whole wavefront
- Uniforms, buffers, constants
- Samplers, texture objects
- Sampler states
- Program counter and flow instruction control
Shader resource bottleneck effect

● Wave occupancy is global for whole instruction buffer of a shader invocation
● So only “worst” spots of your code matter
● They affect performance of whole shader
● Even simple parts / loops will run slow (worse latency hiding)

```cpp
[numthreads(8, 8, 1)]
void ComputeShader()
{
    float outValue;
    ComplexLogicExecutedJustOnce(outValue);  /// VGPRs: 100

    [loop]
    for(int i = 0; i < 128; ++i)
    {
        float loopContext;
        SomeTexFetches(outValue, loopContext);   /// VGPRs: 10
        VerySimpleLogic(loopContext);
    }
}
```

Whole shader occupancy limited by 100 VGPRs
Maximize Compute Unit Wave Occupancy

- Crucial to reduce used “temporary” shader resources
  - LDS, registers, samplers...
- Minimize shader register usage – both vector and scalar!
  - See instruction set
  - Check code disassembly for reference
  - Minimize temporary variable lifetime
- Re-use samplers (separate sampler/texture objects)
  - Refactor existing DX9 material/texture systems
  - Texture2D Load or operator[] can be cheaper than Sample
    - Memory import cost is the same
    - Uses less registers
Maximize Compute Unit Wave Occupancy

- Common X360/PS3 optimizations can be counter-productive
  - Combining passes / too much unrolling
  - Pipelining can be achieved by better wave occupancy instead
  - Split some compute passes
  - Removes “bottleneck effect” of local small occupancy
- Avoid unnecessary use of LDS
• Use “simple” numerical values instead of uniforms
  • Uniforms get loaded to scalar and then vector register
  • Instructions can use constants like 1, -1, 2 directly!

```c
float2 TexcoordToScreenPos(float2 inUV)
{
    float2 p = inUV;
    p.x = p.x * 2.0 + (-1.0);
    p.y = p.y * -2.0 + 1.0;
    return p;
}
```

```
float2 TexcoordToScreenPos(float2 inUV)
{
    float2 p = inUV;
    p.x = p.x * cFov.x + cFov.z;
    p.y = p.y * cFov.y + cFov.w;
    return p;
}
```

```
v_mad_f32 v0, v0, 2.0, -1.0
v_mad_f32 v1, v1, -2.0, 1.0
```
HLSL Optimizations

- Unroll partially/manually
  - Sometimes better to [loop] than [unroll]
  - Still, batch/group 4 memory/texture reads together

- Float4 operations can be suboptimal
  - Use 4 vector registers and 4 operations!
  - Check which variables really need float4, avoid unnecessary work
  - Especially if you know that alpha channel is not used
  - Check if you need 4x4 or 4x3 transform matrices!
GCN Summary

- Very powerful and efficient architecture
- But you need to understand it...
- ...and think very low level!
- Analyze your final ISA assembly constantly
- Great tools available to help you
- Potential speed-up factors of 2-10x with exactly same algorithm!
## Credits – AC4 rendering team

<table>
<thead>
<tr>
<th>Alexandre Lahaise</th>
<th>Michel Bouchard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benjamin Goldstein</td>
<td>Mickael Gilabert</td>
</tr>
<tr>
<td>Benjamin Rouveyrol</td>
<td>Nicolas Vibert</td>
</tr>
<tr>
<td>Benoit Miller</td>
<td>Thierry Carle</td>
</tr>
<tr>
<td>John Huelin</td>
<td>Typhaine Le Gallo</td>
</tr>
<tr>
<td>Lionel Berenguier</td>
<td>Wei Xiang</td>
</tr>
<tr>
<td>Luc Poirier</td>
<td></td>
</tr>
</tbody>
</table>
Special thanks

- Reviewers: Christina Coffin, Michal Drobot, Mickael Gilabert, Luc Poirier, Benjamin Rouveyrol
- Rest of the GI Team: Benjamin Rouveyrol, John Huelin and Mickael Gilabert
- Lionel Berenguier, Michal Drobot, Ulrich Haar, Jarkko Lempiainen for help on code / maths
- Again - whole AC4 rendering team and everyone who helped us
Contact

- Email: bartlomiej.wronski@ubisoft.com
- Twitter: @BartWronsk
- Slides will be available
- www.bartwronski.com
Questions?
Bonus slides
Deferred Normalized Irradiance Probes

Limitations of the technique

- Lack of side bounce
- Ground color bleeding
- Basis not orthonormal
Deferred Normalized Irradiance Probes

Future work

- Change basis to more accurate one
- Add indirect specular
- Increase probe density in X/Y/Z
- Use real HDR irradiance with sky lighting
- Multiple bounces
- Update closest probes in the runtime
Exponential Shadow Maps use in Volumetric Fog

1. Shadowmap downsampling / transform to exponent space

```cpp
float4 accum = 0.0f;
accum += exp(InputTextureShadowmap.GatherRed(pointSampler, samplingPos, int2(0, 0)) * EXPONENT);
accum += exp(InputTextureShadowmap.GatherRed(pointSampler, samplingPos, int2(2, 0)) * EXPONENT);
accum += exp(InputTextureShadowmap.GatherRed(pointSampler, samplingPos, int2(0, 2)) * EXPONENT);
accum += exp(InputTextureShadowmap.GatherRed(pointSampler, samplingPos, int2(2, 2)) * EXPONENT);
OutputTextureESMShadowmap[pos] = dot(accum, 1/16.0f);
```

2. Separable 11-pixel wide box filter (2 trivial passes)

3. Applying shadowmap

```cpp
float receiver = exp(shadedPointShadowSpacePosition.z * EXPONENT);
float occluder = InputESM.SampleLevel(BilinearSampler, shadedPointShadowSpacePosition.xy, 0);
shadow = saturate(occluder / receiver);
```
Screen Space Reflections Optimizations

- We didn’t use hierarchical acceleration structures
  - Decreased shader wave occupancy
  - Added fixed cost – hierarchy construction (~0.4ms on XboxOne)
  - Will investigate more in the future
- Brute force worked better in our case
  - Loop and initialization code must be extremely simple
- Redoing some work was better than syncing group
- Raymarching in lower resolution (2-texel steps in half res)
  - You can do an additional “refinement” step to check for missed collision at earlier texel
Screen Space Reflections Optimizations – Raytracing code

```c
while(1)
{
    // xy = texture space position, z = 1 / scaled linear z
    pos.xyz += ray.xyz;

    float depth_compare = InputTextureDepth.SampleLevel(pointSampler, pos.xy, 0).x * pos.z;

    bool is_offscreen = dot(pos.xy-saturate(pos.xy), 1) != 0;
    bool collision = (depth_compare < depth_threshold.x && depth_compare > depth_threshold.y);

    if(is_offscreen || collision)
        break;
}
```
Parallax Occlusion Mapping
Parallax Occlusion Mapping Optimizations

- Brute-force approach worked well (like screenspace reflections)
- Calculate mip level manually
- Quickly fade the effect out with distance
- Batch texture reads together
- Artists should turn off aniso filtering on heightmaps! 😊
[loop]
while(numIter < 24)
{
    numIter += 1;

    float4 textureCoords[2];
    textureCoords[0] = result.xyxy + float4(1,1,2,2)*tangentSpaceEyeVector.xyxy;
    textureCoords[1] = result.xyxy + float4(3,3,4,4)*tangentSpaceEyeVector.xyxy;

    float4 compareVal = height.xxxx + float4(1,2,3,4)*tangentSpaceEyeVector.zzzz;

    float4 fetchHeight;
    fetchHeight.x = texObject.SampleLevel(texSampler, textureCoords[0].xy, mipLevel).r;
    fetchHeight.y = texObject.SampleLevel(texSampler, textureCoords[0].zw, mipLevel).r;
    fetchHeight.z = texObject.SampleLevel(texSampler, textureCoords[1].xy, mipLevel).r;
    fetchHeight.w = texObject.SampleLevel(texSampler, textureCoords[1].zw, mipLevel).r;

    bool4 testResult = fetchHeight >= compareVal;
    [branch]
    if (any(testResult))
    {
        float2 outResult=0;
        [flatten]
        if(testResult.w)outResult = textureCoords[1].xy;
        [flatten]
        if(testResult.z)outResult = textureCoords[0].zw;
        [flatten]
        if(testResult.y)outResult = textureCoords[0].xy;
        [flatten]
        if(testResult.x)outResult = result;
        result = outResult;
        break;
    }
    result = textureCoords[1].zw;
    height = compareVal.w;
}
Procedural Rain

- Fully GPU-driven – compute and geometry shaders
- Simulate 3x3 grid of rain clusters around the camera
  - Avoids “popping” of new rain drops and guarantees uniform distribution
- Render only visible clusters (CPU culling)
Rain simulation

- Multiple factors taken into account
  - Random rain drop mass and size
  - Wind and gravity
  - Rain-map for simple sky occlusion
    - Top-down 128x128 "shadowmap"
- Screen-space collisions using depth buffer
- Simulating bounced rain drops
Frame N

CS: Spawn point sprites
CS: Update/simulate point sprites
VS/GS/PS: Expand point sprites to particles and draw

read/write structured buffers

Frame N+1

CS: Spawn point sprites
CS: Update/simulate point sprites
VS/GS/PS: Expand point sprites to particles and draw
Geometry Shaders Optimizations

- Minimize memory processed and generated by GS
  - Minimize number of generated vertices
  - Minimize input/output vertex size
  - Implement GPU frustum/occlusion culling in GS
  - Don’t be afraid of reasonable branching
- Investigate if it’s better to simulate four vertices in CS (possibly better pipelining/wave occupancy)
Summary

- CS Particle update cost negligible
- Possible to implement complex update logic
- Some features ("true" `random()`) are tricky
- Move more particle systems to the GPU
- Didn’t need to optimize any of CS shaders
- Geometry Shaders were the performance bottleneck

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS: Update rain drops (up to 320k particles)</td>
<td>&lt;0.1ms</td>
</tr>
<tr>
<td>CS: Screenspace collision</td>
<td>0.2ms</td>
</tr>
<tr>
<td>CS: Update bounced drops</td>
<td>&lt;0.05ms</td>
</tr>
<tr>
<td>GS/VS/PS: Draw rain drops</td>
<td>0.4-4.0ms</td>
</tr>
</tbody>
</table>
References

- Gilabert and Stefanov “Deferred Radiance Transfer Volumes – Global Illumination in Far Cry 3”, GDC 2012
- St-Amour, “Rendering Assassin's Creed III”, GDC 2013
- Hoffman, “Rendering Outdoor Light Scattering in Real Time”, GDC 2002
- Myers, “Variance Shadow Mapping”, NVIDIA Corporation
- Annen et al, “Exponential Shadow Maps”, Hasselt University
References

- Harris et al, “Parallel Prefix Sum (Scan) with CUDA”, GPU Gems 3
- “Southern Islands Series Instruction Set Architecture”, AMD
- Valient, “Killzone Shadowfall Demo Postmortem”, Guerilla Games
- Tatarchuk, “Practical Occlusion Mapping”, ATI Research/AMD
- Drobot, “Quadtree Displacement Mapping”, Reality Pump