Assassin’s Creed 4: Black Flag

Lighting, Weather and Atmospheric Effects

Bart Wronski, 3D Programmer, Ubisoft Montreal
Background

- Next-gen consoles launch title!
- ...but done for previous generation as well
- Current-gen was dominating install-base and we cared a lot about those players
- Constraints on assets duplication and extra artist work
- Next-gen improvements had to be procedural
Presentation overview

- Lighting and Global Illumination
  - Improvements over Assassin’s Creed 3
  - Global illumination solution
  - Ambient Occlusion: temporal-supersampled SSAO
  - Multi-resolution ambient occlusion
- Atmospheric and weather effects
  - Rain ripples and wet materials
  - Screenspace reflections
  - GPU simulated rain
  - Volumetric fog
Notice how flat and uninteresting ambient lighting looks in shadowed areas. We lose not only sense of positional relations between objects, scene composition but also don’t see almost any normal mapping.
Lighting improvements

- Key problems
  - Overall flatness
  - Lack of sense of position and directionality
  - Loss of normal mapping information

- Requirements
  - Prototyped multiple real-time GI solutions, none of them looked good enough
  - Partially baked solution
  - Vast open world with sparse dense areas
  - Dynamic weather / time of day
  - Work on current gen (~1ms / < 1MB for GPU)
  - Small impact on art pipelines
We had already developed good and working technology suitable for open world games with dynamic time of day – Far Cry 3 Deferred Radiance Transfer Volumes. We decided to give it a try and integrated it quickly despite engine differences (around 1-2 man-weeks)
Our early prototypes confirmed all benefits of Far Cry 3 solution. However technique that was perfect for very sparse, forest and open areas was not high enough quality for urban environments. We observed too low resolution and variance in GI information, bounced indirect lighting was not giving us enough contrast and didn’t change enough during day. It was mostly because extremely low precomputed radiance transfer resolution – second order SH are not capable of storing complex visibility function.
Key observations:
- Under sunny weather usually most perceived indirect lighting comes from main light source
- We can easily split indirect lighting from sky and its bounces from main light component – techniques existing in our engine (like World Ambient Occlusion and SSAO) are enough to provide shadowing information
- Main light trajectory not dependent on weather preset – only light color and strength depends on it
- For a single GI diffuse-only bounce we can precompute and store “some” information based only on:
  - Albedo
  - Normals
  - Shadowing information

New solution - background

- One key light
- Weather has no influence on light direction
- Small amount of local lights
We developed a new solution - **Deferred Normalized Irradiance Probes**
Storing « normalized » irradiance for neutral color - WHITE, but take shadowing into account.
No multi bounce (was possible, but artists didn’t want to increase iteration times) -> use constant ambient term to fake it and avoid pitch-black areas.
First, let’s have a look at our offline rendering / baking part.

1. We use GPU cubemap capture and GPU convolution to calculate irradiance for four basis vectors.
2. GPU baker was very efficient as albedo and normals do not change with dynamic time of day, only shadowmaps. We reused big resolution shadowmaps for whole sector of 64x64 meters.
3. World has 2.5D structure so we stored information in 2D textures that could be compressed and later blitted using GPU
Use navmesh: trim the ones in unreachable interiors and reuse neighboring information -> Solve interpolation issue.
Data storage

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

Times at which we captured GI:
Midnight, 3AM, 6AM, 9AM, 12PM, 3PM, 6PM, 9PM
1. For each probe, 8 cubemaps, every 3 hours
   1. We use a neutral, white sun in order to be able to apply weather at runtime
   2. We have an optional ambient term because we don’t have multi bounce GI
1. For each cubemap we compute 4 irradiance colors by integrating the cubemap in 4 directions, indicated by the arrow (same as FC3 basis)
2. Faster to do it this way to get the final ambient color (PS3 swizzling vector registers performance)
3. Red arrow on upper right side of the screen -> One normalized irradiance color
4. Put this color in Red channel of 3 different texels
5. Repeat the process for Green direction, blue direction, grey direction
6. We obtain 4 colors x 8 times of day.
7. We pack each channel R,G,B of the 4 directions together, into 3 RGBA channels
8. And we store everything in a texture
- Group this data in sectors -> 16x16 probes
- Blinking dot represent data for one probe
- Texture that is used in engine
- Embedded in entity, streamed in like other resources
To provide interactive iteration times, we needed to modify brute-force baking **A LOT**. @ 60 FPS / 30 000 probes, 6 cube faces x 8 time of day = 1 440 000 renders -> 400 mn (6h40) Unacceptable! (notes: before navmesh spawning optimization, we had 110 000 probes, so we would have needed ~ 26 hours to compute everything ...)

Some optimizations notes:
- Less objects -> camera Far @ 15 meters
- Reuse Gbuffer for 8 times of day
- Same shadow maps shared by sector of probes
- One Cascade -> contains all sector
- Frustum Culling bottleneck -> custom traversal, 2d position and distance based
- Make sure CPU won’t stall when locking GPU textures -> use round robin of multiple queries
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
Denormalization done on GPU (pixel shader taking 0.1ms on ps3)

We use the irradiance texture generated offline
Based on the current time of day, and camera location, sample the appropriate pixels in each probe area
Interpolate to a neutral value based on XY distance of the pixel from camera
We use these runtime irradiance textures to compute our indirect lighting: the 4 irradiance get interpolated based on the pixel normal, and we have a height texture that allows us to fadeout vertically to a neutral color.

Right side of the slide: sky lighting – ambient cube combined with normals + world ambient occlusion data simulating occlusion from high buildings and other “big” geometric features
Center: combined ambient (without SSAO)
Image decomposition:

Direct lighting
Sky lighting + world AO
Indirect lighting
Combined, with SSAO (PS3 SSAO version!)
Combined, SSAO, albedo applied
Benchmarks and summary

<table>
<thead>
<tr>
<th>Performance cost</th>
<th>1.2ms fullscreen pass - PS3</th>
</tr>
</thead>
<tbody>
<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>

Performance cost:
PS3&X360 Without stencil marking optimization (sky, ocean) 1.2ms for fullscreen pass
When combined with calculating screen space sun shadow mask (cascade selection, PCF filtering) in the same pass, around 1.8ms – some cost got amortized
On next gen and PC cost of bounce calculations is negligible compared to shadow mask that it is computed with

Memory cost:
We load 25 sectors around the player, all of them store data as 16x16 probes for 8 times of day and irradiance in 4 directions:
25 RGBA textures sized 48x128 = 600kb of VRAM
We thought about DXT5 compression or streaming just required 2 times of day, but cost is almost negligible so we didn’t need to pursue it

CPU performance:
We needed to uncompress world height data available for gameplay and placed in RAM memory. We optimized it a lot using vector intrinsics, but still cost was non-negligible.
In the end it cost us ~0.6ms for full update. Fortunately, some parts we didn’t have to do – as we supported texel snapping, we could update height data only when it was needed.

Number of probes:
When using bruteforce placement on highest available spot in the world aligned to grid, we had around 110 000 probes per typical game world.
When we used navigation mesh to place probes only on areas accessible to the player, it got down to 30 000.

Full baking time:
On single machine equipped with nVidia GTX 680, full baking time for whole world and all 8 times of day was around 8 minutes. Therefore it didn’t require any distribution over whole world, technical art directors and lighting artists were able to do it on demand.
We also added option to re-bake some probes or have continuous baking in background in editor (probes visible in the viewport were constantly rebaked in the background – almost no performance impact)
We wanted to change irradiance storage basis to something more accurate one.

HL2 basis with added “wrap-around” vector pointing down - didn’t work very well. The weights were not always normalized, resulting in brighter color in some directions. There was lots of ground color bleeding to the sides.
Increase probe density / Have multiple layers in height axis instead of blending it out with height

Instead of storing “normalized” irradiance, use multiple weather presets

Handle multiple bounces. Had some support but didn’t use it in the final result – iteration times. We were also not being completely energy conserving (storage basis) – adding light to the scene instead of diffusing it.

Update some probes on GPU -> cache Gbuffer and light env map @ runtime -> distribute probes on multiple frames
Ambient occlusion
Scalable Ambient Obscurance

- Extensions to algorithm by McGuire et al
- Great performance - ~1.6ms full res on consoles with filtering and applying
- Large radius ~1.5m, takes normal mapping into account
- Our extension - **temporal supersampling**
- Different than existing approaches by changing sampling pattern every frame
- Rotate AO samples spiral in 3 distinct patterns every frame (120 degrees)
- Effectively the triple number of samples!
- Blend and accumulate frames, reject on depth difference
Example pattern (not final one!) rotated 3 times and accumulated over time to triple number of effective samples
No temporal
Temporal
No temporal supersampling / smoothing
Ambient occlusion

- SSAO is not enough to represent sky lighting occlusion...
- Even multi-res approaches won’t catch all occlusion
- Idea - separate ambient occlusion into multiple frequency bands
- Every frequency band calculated with a different algorithm!
Great write-up and example to existing technique by Inigo Quilez.
World Ambient Occlusion

- Needed large scale sky occlusion for direct sky lighting from buildings and trees
- World Ambient Occlusion
- Technique developed for Assassin’s Creed 3
- See Jean-Francois St-Amour talk from GDC 2013!
Looks of “World ambient occlusion” in Assassin’s Creed 3
Algorithm very approximate and requires artists to tweak some “magic values”, but extremely cheap to evaluate and quite robust.

...As long as your scene can be approximated by heightfield!
Looks of “World ambient occlusion” in Assassin’s Creed 3
Again some results screenshots.
Old AC3 technique – just the ambient cube, without the WorldAO.
Screenshot from around 7-8PM, because of very low sun elevation this part of town is completely in shadow.
Effect – uninteresting, artificial look. Loss of almost all normal mapping and lighting variety.
It is also too dark, but brighter values would result in even flatter look.
This is how our bounced lighting looks like. We get light bounce from proper direction with proper shadowing. Lighting varies not only with direction, but also with world position.
We start to see some normal mapping on the ground.
Composed scene of bounced lighting and regular sky light represented by lighting cube. Scene rich, colorful with lots of light variation and visible normal maps.
Final scene with World AO and SSAO applied.
To read World AO technique description, see GDC 2013 presentation by Jean-Francois St-Amour called “Rendering of Assassin’s Creed 3”

We present multiple probes and their placement (every 2m). Probes are placed automatically using nav-mesh – green lines represent probes missing due to collision – they get replaced by the probe connected to them. We found no errors using such interpolation and probe replacement.

Also this screenshot shows our irradiance vector basis. This is unmodified Far Cry 3 basis – three vectors pointing up (like in Half Life 2 basis) and additional, fourth wrap-around vector responsible for lighting bounced from the ground.
Atmospheric and weather effects
Atmospheric effects

- Caribbean tropical climate extremely unpredictable
- Goes from dusty and dry to showers and storms within minutes
- Already had coherent and robust weather system from AC3
- Need for new cool, next-gen procedural atmospheric effects!
GPU Procedural Rain
Rain ripples
No ripples
Ripples
Rain ripples

CS: Spawn and evolve rain ripples
PS: Draw rain ripples into signed heightfield texture
PS: Calculate heightfield derivatives

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>float2</td>
</tr>
<tr>
<td>Life</td>
<td>float</td>
</tr>
<tr>
<td>Max life</td>
<td>float</td>
</tr>
<tr>
<td>Strength</td>
<td>float</td>
</tr>
<tr>
<td>Max radius</td>
<td>float</td>
</tr>
</tbody>
</table>

Updated rain ripple structure
256x256 signed R8 texture
256x256 signed R8G8 texture
Drawing rain ripples - geometry multiplication using geometry shaders
Rain ripples

- Single pass to apply it on screen and perturb existing normals
- Wrap texture around in world space
- World AO is sky occlusion...
- ... so use it for rain occlusion = no additional runtime cost
- Rain ripples update and texture generation cost ~0.2ms
- Perturbing normals can be a separate pass (~0.4ms) or combined with lighting (pipelined well and “free”!)
Wet materials

- Surface wetness stored in G-buffer
- “Baked” for wet areas or modified dynamically by weather
- Use it during lighting pass to update gloss and albedo
- Increase the gloss, darken the albedo
- Same technique used in Assassin’s Creed 3
- ...but enhanced using screenspace reflections!
No screen-space reflections
Notice how screen-space reflections add wetness impression and again improve composition of the whole scene. Especially interesting is how on-screen objects provided proper sky occlusion.
On the next gen consoles, we wanted to use compute and geometry shaders to achieve extremely high volume 3d rain effect. It was supposed to be fully procedural, but art-driven effect.
Rain drop system – simulation and rendering
Due to rather simple nature of rain simulation (but massive particle count), we decided to keep whole simulation on GPU.
Rain behavior and physics were simulated using compute shaders and then later expanded from points to target rain geometry using geometry shaders.
To avoid “popping” of rain drops but still have decent performance, we have 9 active rain clusters (3x3 grid), centered around the camera. Rain is simulated in all of them, but only ones that are intersecting the camera frustum are being rendered.
We take multiple factors into account during the simulation

- Multiple factors taken into account
  - Random rain drop mass and size
  - Wind and gravity
  - Rain-map for simple sky occlusion
    - Top-down close range 128x128 “shadowmap”
Notice how extremely low resolution was the rain map and it lacked any dynamic objects or even ground!
...but its rendering cost was negligible.
Procedural rain simulation

- Rain map used together with WorldAO for rain occlusion
  - (different range and precision)
- Screen-space collisions with depth buffer
- Spawning new particles on collision
  - Simulating bounced rain drops
Rain update process:
1. Using rain map as occlusion information, we spawn some new rain drops at random positions on top of each cluster and append it to previous frames buffer with rain drops using a compute shader.
2. Update information about rain drops using their velocity, mass and wind information. We also calculate screen-space precise collision to determine which rain drops are colliding with the scene and spawn splashes instead. Every splash is composed from 6 wider rain drops that die out very quickly.
3. We draw rain drops using geometry shaders to expand points to sprites.
4. Rain point sprite buffers from given frame are used in the following frame.

Information from Nth frame will be used in the N+1th frame rain drop system update to keep the simulation continuous.
Video of bounced rain particles – exaggerated
It turned out that moving particles update to the GPU was a good decision:

- **Compute shaders** are capable of updating huge amount of particles in negligible time. We were updating even not visible particles (clusters behind the camera) and the performance was very good. We avoided multiple CPU/GPU sync points like fetching the rainmap, spawning rain splashes on the CPU, updating the dynamic resources etc.
- You can implement even very complex logic in compute shader, with branches etc. but some features are tricky. For example spawning new particles on particle death requires implementing additional CS passes. Also getting good random numbers in CS is not easy – we prepared a buffer with precomputed random numbers.
- It is definitely worth investigating moving more particle system simulation to GPU to offload the CPU.
- We didn’t need to optimize any of our compute shaders
- **Geometry shaders** turned out to be a performance bottleneck. While we saved lots of memory bandwidth by simulating rain drops as points in the CS, the need to expand them in GS turned out to consume lots of GPU time and we needed to optimize it heavily. Rain without those optimizations was taking up to 20ms!
Geometry shaders increase bandwidth usage a lot - all generated data must pass through memory.

We found out that it is beneficial to:

- Minimize both maximum and actual number of generated vertices from a GS.
- Minimize output and output vertex size. Some fields that are redundant can be safely removed and calculations moved to the PS. If they are relatively simple they will be scheduled better on the per-pixel level. Sometimes it is even better to fetch such value from global memory (constant/instance buffer).
- It is usually trivial to implement some sort of frustum/occlusion culling in the GS. We found even doubled performance on this stage just by doing it.
- Early out branches seem to not add any significant performance overhead and can save a lot of generated bandwidth.
- As AMD recommends, it would still be much better to avoid any GS when not necessary... And do manual, vertex shader based instancing and instance fetching.
Technique we called internally a “volumetric fog” is in fact simulation of various and different atmospheric phenomena:

- Fog, mist, haze
- “God rays”
- Light shafts
- Dusty/wet air
- Volumetric shadows

But all of them happen because of single physical phenomenon - light in- and out-scattering!
It worked coherently, correctly and perfectly with multiple light sources
Summary

- Effect compatible with deferred and forward – can be computed even asynchronously (PS4) as soon as shadow maps are available – doesn’t depend on scene geometry
- Any number of transparent layers or particles – cost of applying it in final shader is just cost of one tex3D fetch + one lerp function!
- Very cheap with fixed cost for accumulation and scattering part – the same on 720p as well as 4k!
- Density and lighting can be tweaked independently and changed completely
- For implementation details, see my GDC 2014 talk!
# Credits

<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>Benoît 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

- Benjamin Rouveyrol for materials for GI slides
- Rest of GI Team: Benjamin Rouveyrol, John Huelin and Mickael Gilabert
- Wei Xiang for initial implementation of rain
- Assassin's Creed 4 technical art directors: Danny Oros, Guillaume Lefebvre, Philippe Ringuette-Angrignon, Philippe Trairieux, Sebastien Larrue
Questions?
Contact me!

- Email: bartlomiej.wronski@ubisoft.com
- Twitter: @BartWrons
- Slides on my blog
- www.bartwronski.com
References

- Gilabert and Stefanov “Deferred Radiance Transfer Volumes - Global Illumination in Far Cry 3”, GDC 2012
- Sloan et al., “Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments”, SIGGRAPH 2002
- Quilez, post about Multi-resolution ambient occlusion
  http://www.iquilezles.org/www/articles/multiresaocc/multiresaocc.htm
- 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
- Annes et al., “Exponential Shadow Maps”, Hasselt University
- Andersson and Bavoil, “Stable SSAO in Battlefield 3 with Selective Temporal Filtering”
- Smedberg and Wright, “Rendering Techniques in GEARS OF WAR 2”
- Bilodeau, “Vertex Shader Tricks”
- Wronski, “Assassin’s Creed 4 - road to next-gen graphics”, GDC 2014