Part 4.5

Scalable Mobile Visualization: Smart precomputation for complex lighting

Pere-Pau Vázquez, UPC
High quality illumination

- Consistent illumination for AR
- Soft shadows
- Deferred shading
- Ambient Occlusion
Consistent illumination for AR

• High-Quality Consistent Illumination in Mobile Augmented Reality by Radiance Convolution on the GPU [Kán, Unterguggenberger & Kaufmann, 2015]

• Goal
  – Achieve realistic (and consistent) illumination for synthetic objects in Augmented Reality environments
Consistent illumination for AR

• Overview
  – Capture the environment with the mobile
  – Create an HDR environment map
  – Convolve the HDR with the BRDF’s of the materials
  – Calculate radiance in realtime
  – Add AO from an offline rendering as lightmaps
  – Multiply with the AO from the synthetic object
Consistent illumination for AR

- Capture the environment with the mobile
  - Rotational motion of the mobile
    - In yaw and pitch angles to cover all sphere directions
    - Images accumulated to a spherical environment map
- HDR environment map constructed while scanning
  - Projecting each camera image
    - According to the orientation and inertial measurement of the mobile
  - Low dynamic range imaging is transformed to HDR
    - Camera uses auto-exposure
      - Two overlapping images will have slightly different exposure
    - Alignment correction based on feature matching
  - All in the device
Consistent illumination for AR

- **Convolve the HDR with the BRDF’s of the materials**
  - Use MRT to support several convolutions at once
  - Assume distant light
  - One single light reflection on the surface
  - Scene materials assumed non-emissive
  - Use a simplified rendering equation

- **Weight with AO (obtained offline)**
  - Built for real and synthetic objects
  - Use the geometry of the scene
    - Use a proxy geometry for the objects of the real world
    - Cannot be simply done on the fly
Consistent illumination for AR

• Results

Without AO

With AO

Images courtesy of Peter Kán
Consistent illumination for AR

• Performance

<table>
<thead>
<tr>
<th>3D model</th>
<th># triangles</th>
<th>Framerate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reflective cup</td>
<td>25.6K</td>
<td>29 fps</td>
</tr>
<tr>
<td>Teapot</td>
<td>15.7K</td>
<td>30 fps</td>
</tr>
<tr>
<td>Dragon</td>
<td>229K</td>
<td>13 fps</td>
</tr>
</tbody>
</table>

• Limitations
  – Materials represented by Phong BRDF
  – AO and most shading (e.g. reflection maps) is baked
Soft shadows using cubemaps

• Efficient Soft Shadows Based on Static Local Cubemap [Bala & Lopez Mendez, 2016]

• Goal
  – Soft shadows in realtime

Taken from https://community.arm.com/graphics/b/blog/posts/dynamic-soft-shadows-based-on-local-cubemap
Mobile Graphics Tutorial – 3DV 2018

Soft shadows using cubemaps

• Overview
  – Create a local cube map
    • Offline recommended
    • Stores color and transparency of the environment
    • Position and bounding box
      – Approximates the geometry
    • Local correction
      – Using proxy geometry
  – Apply shadows in the fragment shader
Soft shadows using cubemaps

• Generating shadows
  – Fetch texel from cubemap
    • Using the fragment-to-light vector
    • Correct the vector before fetching
      – Using the scene geometry (bbox) and cubemap creation position
        » To provide the equivalent shadow rays
  – Apply shadow based on the alpha value
  – Soften shadow
    • Using mipmapping and addressing according to the distance
Soft shadows using cubemaps

• Conclusions
  – Does not need to render to texture
    • Cubemaps must be pre-calculated
  – Requires reading multiple times from textures
  – Stable
    • Because cubemap does not change

• Limitations
  – Static, since info is precomputed
Physically-based Deferred Rendering

- Physically Based Deferred Shading on Mobile [Vaughan Smith & Einig, 2016]

- Goal:
  - Adapt deferred shading pipeline to mobile
  - Bandwidth friendly
  - Using Framebuffer Fetch extension
    - Avoids copying to main memory in OpenGL ES
Physically-based Deferred Rendering

- **Overview**
  - Typical deferred shading pipeline
Physically-based Deferred Rendering

- **Main idea**: group G-buffer, lighting & tone mapping into one step
  - Further improve by using Pixel Local Storage extension
    - G-buffer data is not written to main memory
    - Usable when multiple shader invocations cover the same pixel
  - Resulting pipeline reduces bandwidth

![Diagram](image_url)
Physically-based Deferred Rendering

- Two G-buffer layouts proposed
  - Specular G-buffer setup (160 bits)
    - Rgb10a2 highp vec4 light accumulation
    - R32f highp float depth
    - 3 x rgba8 highp vec4: normal, base color & specular color
  - Metallicness G-buffer setup (128 bits, more bandwidth efficient)
    - Rgb10a2 highp vec4 light accumulation
    - R32f highp float depth
    - 2 x rgba8 highp vec4: normal & roughness, albedo or reflectance metallicness
Physically-based Deferred Rendering

• Lighting
  – Use precomputed HDR lightmaps to represent static diffuse lighting
    • Shadows & radiosity
  – Can be compressed with ASTC (supports HDR data)
    • PVRTC, RGBM can also be used for non HDR formats
  – Geometry pass calculates diffuse lighting
  – Specular is calculated using Schlick’s approximation of Fresnel factor
Physically-based Deferred Rendering

• **Results (PowerVR SDK)**
  – Fewer rendering tasks
    • meaning that the G-buffer generation, lighting, and tonemapping stages are properly merged into one task.
    • reduction in memory bandwidth
      – 53% decrease in reads and a 54% decrease in writes

• **Limitations**
  – Still not big frame rates
Ambient Occlusion in mobile

- Optimized Screen-Space Ambient Occlusion in Mobile Devices [Sunet & Vázquez, Web3D 2016]

- Goal: Study feasibility of real time AO in mobile
  - Analyze most popular AO algorithms: Crytek’s, Alchemy’s, Nvidia’s Horizon-Based AO (HBAO), and Starcraft II (SC2)
  - Evaluate their AO pipelines step by step
  - Design architectural improvements
  - Implement and compare
Ambient Occlusion in mobile

- Ambient Occlusion. Simplification of rendering equation
  - The surface is a perfect diffuse surface (BRDF constant)
  - Light potentially reaches a point \( p \) equally in all directions
    - But takes into account point’s visibility

\[
L_o(p, \omega_o) = \frac{1}{\pi} \int_\Omega \rho(d(p, \omega_i)) \cos \theta_i \, d\omega_i
\]

\[
\rho(d) = \begin{cases} 
  f(d) & \text{if } d \in [0, 1] \\
  0 & \text{otherwise}
\end{cases}
\]
Ambient Occlusion in mobile

- **AO typical implementations**
  - Precomputed AO: Fast & high quality, but static, memory hungry
  - Ray-based: High quality, but costly, visible patterns…
  - Geometry-based: Fast w/ proxy structures, but lower quality, artifacts/noise…
  - Volume-based: High quality, view independent, but costly

- **Screen-space:**
  - Extremely fast
  - View-dependent
  - [mostly] requires blurring for noise reduction
  - Very popular in video games (e.g. Crysis, Starcraft 2, Battlefield 3…)
Ambient Occlusion in mobile

- **Screen-space AO:**
  - Approximation to AO implemented as a screen-space post-processing
    - ND-buffer provides coarse approximation of scene's geometry
    - Sample ND-buffer to approximate (estimate) ambient occlusion instead of shooting rays
 Ambient Occlusion in mobile

**SSAO pipeline**

1. Generate ND (normal + depth, OpenGL ES 2) or G-Buffer (ND + RGB…, OpenGL ES 3.+)  
2. Calculate AO factor for visible pixels  
   a. Generate a set of samples of positions/vectors around the pixel to shade.  
   b. Get the geometry shape (position/normal…)  
   c. Calculate AO factor by analyzing shape…  
3. Blur the AO texture to remove noise artifacts  
4. Final compositing
Ambient Occlusion in mobile

- **Optimizations.**
  - G-Buffer storage (less precision)
    - 8 not enough
    - 16 and 32 similar quality
  - Normal storage (RGB vs RG)
    - RGB normals are faster
  - Samples generation (offline)
    - Poisson disc (2D) and 8-point cosine-weighted hemisphere (3D)
  - Geometry recovery
    - Similar triangles instead of inverse tf
  - Geometry storage
    - Store depth instead of 3D positions
      - Trades bandwidth for memory

- **Optimizations (cont)**
  - Banding and noise
    - Reduce noise using bilateral (separable) filter instead of Gaussian
  - Progressive improvement
    - Amortize AO through multiple frames

![Diagram of Ambient Occlusion process: Frame i-1 and Frame i](image)
Ambient Occlusion in mobile

- **Optimizations**
  - Naïve improvement: Reduce the calculation to a portion of the screen
    - Mobile devices have a high PPI resolution
    - Reduction improves timings dramatically while keeping high quality
  - Typical reduction:
    - Offscreen render to 1/4th of the screen
    - Scale-up to fill the screen

- **Results**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Optimized (not progressive)</th>
<th>Optimized + progressive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starcraft 2</td>
<td>17.8%</td>
<td>38.5%</td>
</tr>
<tr>
<td>HBAO</td>
<td>25.6%</td>
<td>39.2%</td>
</tr>
<tr>
<td>Crytek</td>
<td>23.4%</td>
<td>35.0%</td>
</tr>
<tr>
<td>Alchemy</td>
<td>24.8%</td>
<td>38.2%</td>
</tr>
</tbody>
</table>
Part 4.5

Scalable Mobile Visualization: Volumetric Data

Pere-Pau Vázquez, UPC
Rendering Volumetric Datasets

- Introduction
- Challenges
- Architectures
- GPU-based ray casting on mobile
- Conclusions
Rendering Volumetric Datasets

Capturing

Rendering

3D texture

GPU-based ray casting

Output
Rendering Volumetric Datasets

• Introduction
  – Volume datasets
    • Sizes continuously growing (e.g. $>1024^3$)
      – Complex data (e.g. 4D)
  – Rendering algorithms
    • GPU intensive
    • State-of-the-art is ray casting on the fragment shader
  – Interaction
    • Edition, inspection, analysis, require a set of complex manipulation techniques
Rendering Volumetric Datasets

- **Desktop vs mobile**
  - Desktop rendering
    - Large models on the fly
    - Huge models with the aid of compression/multiresolution schemes
  - Mobile rendering
    - Standard sizes (e.g. $512^3$) still too much for the mobile GPUs
    - Rendering algorithms GPU intensive
      - State-of-the-art is GPU-based ray casting
    - Interaction is difficult on a small screen
      - Changing TF, inspecting the model...
Rendering Volumetric Datasets

- **Challenges on mobile:**
  - **Memory:**
    - Model does not fit into memory
      - Use client server approach / compress data
  - **GPU capabilities:**
    - Cannot use state of the art algorithm (e.g. no 3D textures)
      - Texture arrays
  - **GPU horsepower:**
    - GPU unable to perform interactively
      - Progressive rendering methods
  - **Small screen**
    - Not enough details, difficult interaction
Rendering Volumetric Datasets

- **Mobile architectures**
  - Server-based rendering
  - Hybrid approaches
  - Pure mobile rendering

- Server-based and hybrid rely on high bandwidth communication
Rendering Volumetric Datasets

• **Pure mobile rendering**
  – Move all the work to the mobile
  – Nowadays feasible

• **Direct Volume Rendering on mobile. Algorithms**
  – Slices
  – 2D texture arrays
  – 3D textures
Rendering Volumetric Datasets

- **2D texture arrays + texture atlas [Noguera et al. 2012]**
  - Simulate a 3D texture using an array of 2D textures
  - Implement GPU-based ray casting
    - High quality
    - Relatively large models
    - Costly
    - Cannot use hardware trilinear interpolation
Rendering Volumetric Datasets

- 2D texture arrays + texture atlas
Rendering Volumetric Datasets

• 3D textures [Balsa & Vázquez, 2012]
  – Allow either 3D slices or GPU-based ray casting
  – Initially, only a bunch of GPUs sporting 3D textures (Qualcomm’s Adreno series >= 200)
  – Performance limitations (data: $256^3$ – screen resol. 480x800)
    • 1.63 for 3D slices
    • 0.77 fps for ray casting
Rendering Volumetric Datasets

2D slices comparison

- Nexus ONE
- HTC Desire
- HTC Desire HD
- HTC Desire Z
- Samsung Galaxy S2
- Advent Vega
- LG Optimus 2X
- Samsung Galaxy S

FPS

Number of Slices

- 64
- 128
- 256
- 512
Rendering Volumetric Datasets

- 2D slices vs 3D slices vs raycasting
Rendering Volumetric Datasets

- Using Metal on an iOS device [Schiewe et al., 2015]

Taken from [Schiewe et al., 2015]
Volume data. GPU ray casting on mobile

• Using Metal on an iOS device [Schiewe et al., 2015]
  – Standard GPU-based ray casting
  – Provides low level control
  – Improved framerate (2x, to a maximum of 5-7 fps) over slice-based rendering
  – Models noticeably smaller than available memory (max. size was $256^2 \times 942$)
Volume data. GPU ray casting on mobile

• Progressive Ray Casting for Volumetric Models on Mobile Devices [Díaz et al., 2018]
  – Two algorithms for progressive ray casting adapted to smartphones
  – Tested on Android
  – Using OpenGL
Volume data. GPU ray casting on mobile

• Overview
  – Common core: Progressive ray-casting
    • Low-level resolution
  – Two methods for progressive refinement
    • FBSlabs: Refine front to back
      – Budget-based rendering (slab sizes)
      – Requires lower level OpenGL ES/WebGL (3.0 [or lower], no compute shaders)
      – Progressive refinement more noticeable
    • STiles: Refine tile-based
      – Budget-based rendering (number/size of tiles)
      – Less visible update changes
      – Requires higher level OpenGL (3.1 or higher, compute shaders to sort tiles)
Volume data. GPU ray casting on mobile

- FBSlabs

1) Low Resolution RC (during interaction)

Ray entry/exit position textures

Ray Input
Ray Output

Low resolution raycasting

RC color (RGB)
Volume data. GPU ray casting on mobile

- FBSlabs
Volume data. GPU ray casting on mobile

- STiles

1) Low Resolution RC (while interacting)

- Ray entry/exit position textures
- Low resolution raycasting
  - RC color (RGB)
  - RC cost (A)

2) Tile Sorting (after interaction finished)

OUTPUT

- Sorted space
- Original space

Coordinate mapping

Maps to original space
Maps to sorted space
Volume data. GPU ray casting on mobile

- STiles

3) Progressive High Resolution RC

INPUT

Frame 1
Tile Raycasting
Composition

Frame 2
Tile Raycasting
Composition

Frame N
Tile Raycasting
Composition

...
Volume data. GPU ray casting on mobile

- **Results. Completion time**

  ![Completion time of several RC methods on different devices](image)
  
  **Avg. completion time (seconds)**
  
  **Device Options**: Huawei Nexus 6P, Motorola Nexus 6
  
  **Methods**: VIX, Head, Obelix, Chameleon, Melanix
Volume data. GPU ray casting on mobile

- Results. Framerate

<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>FBSlabs</th>
<th>STiles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Avg</td>
</tr>
<tr>
<td>Vix</td>
<td>8.03635</td>
<td>11.6122</td>
<td>8.84522</td>
</tr>
<tr>
<td>Head</td>
<td>8.16621</td>
<td>11.1880</td>
<td>9.73117</td>
</tr>
<tr>
<td>Obelix</td>
<td>7.85773</td>
<td>10.2932</td>
<td>8.83859</td>
</tr>
<tr>
<td>Chamaleon</td>
<td>7.90516</td>
<td>10.8366</td>
<td>9.75879</td>
</tr>
<tr>
<td>Melanix</td>
<td>7.92802</td>
<td>10.2918</td>
<td>8.65489</td>
</tr>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Avg</td>
</tr>
<tr>
<td>Vix</td>
<td>8.04296</td>
<td>10.8118</td>
<td>9.65829</td>
</tr>
<tr>
<td>Head</td>
<td>7.8058</td>
<td>10.5197</td>
<td>9.65324</td>
</tr>
<tr>
<td>Obelix</td>
<td>8.65025</td>
<td>12.5418</td>
<td>9.68395</td>
</tr>
<tr>
<td>Chamaleon</td>
<td>8.82493</td>
<td>12.9486</td>
<td>9.59793</td>
</tr>
<tr>
<td>Melanix</td>
<td>9.18801</td>
<td>13.3794</td>
<td>10.1308</td>
</tr>
</tbody>
</table>

Nexus 6P

<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>FBSlabs</th>
<th>STiles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Avg</td>
</tr>
<tr>
<td>Vix</td>
<td>8.03635</td>
<td>11.6122</td>
<td>8.84522</td>
</tr>
<tr>
<td>Head</td>
<td>8.16621</td>
<td>11.1880</td>
<td>9.73117</td>
</tr>
<tr>
<td>Obelix</td>
<td>7.85773</td>
<td>10.2932</td>
<td>8.83859</td>
</tr>
<tr>
<td>Chamaleon</td>
<td>7.90516</td>
<td>10.8366</td>
<td>9.75879</td>
</tr>
<tr>
<td>Melanix</td>
<td>7.92802</td>
<td>10.2918</td>
<td>8.65489</td>
</tr>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Avg</td>
</tr>
<tr>
<td>Vix</td>
<td>8.04296</td>
<td>10.8118</td>
<td>9.65829</td>
</tr>
<tr>
<td>Head</td>
<td>7.8058</td>
<td>10.5197</td>
<td>9.65324</td>
</tr>
<tr>
<td>Obelix</td>
<td>8.65025</td>
<td>12.5418</td>
<td>9.68395</td>
</tr>
<tr>
<td>Chamaleon</td>
<td>8.82493</td>
<td>12.9486</td>
<td>9.59793</td>
</tr>
<tr>
<td>Melanix</td>
<td>9.18801</td>
<td>13.3794</td>
<td>10.1308</td>
</tr>
</tbody>
</table>

Nexus 6
Volume data. GPU ray casting on mobile

- Results. Perceptual differences vs previous frame

![Perceptual dissimilarity against previous frame](chart)

- Chart showing perceptual dissimilarity against previous frame for different methods:
  - FBSlabs
  - STiles
  - Simple (random)
  - Simple (structured)
Rendering Volumetric Datasets

- Challenges: Transfer Function edition
Rendering Volumetric Datasets

• Challenges: Transfer Function edition
Rendering Volumetric Datasets

• Conclusion
  – Volume rendering on mobile devices possible but limited
    • Can use daptive rendering (half resolution when interacting)
  – 3D textures in core GLES 3.0
    • Requires progressive raytracing for not so large models
  – Interaction still difficult
  – Client-server architecture still alive
    • Can overcome data privacy/safety & storage issues
    • Better 4G-5G connections
    • …
Next Session

MOBILE METRIC CAPTURE AND RECONSTRUCTION