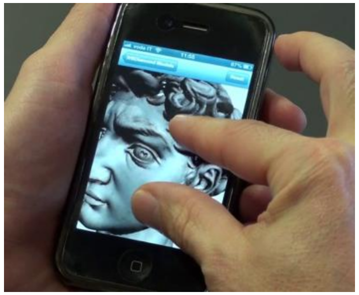
Part 5

Scalable mobile visualization



Mobile platforms scenario

- **Mobile hardware is continuously improving at impressive paces.**
- **Screen resolutions are often extremely large. (2 – 6 Mpix)**
- **Mobile 3D graphics hardware is powerful but still constrained**
- **Major limiting factors wrt desktop counterparts**
  - low computing powers
  - low memory bandwidths
  - small amounts of memory
  - limited power supply.

- **Try to circumnavigate these limitations**
  - In order to achieve scalable mobile rendering

Mobile rendering scenario

- **Requirements**
  - Hi quality interactive images
- **Constraints**
  - Limited GPU, RAM and Bandwidth
- **No brute force method applicable**
  - Need for “smart methods” to perform interactive rendering
  - Exploit at best reduced rendering power
- **Proposed solutions**
  - Render only necessary data: adaptive multiresolution
  - Data not already available on device: streaming approach
  - Exploit at best available bandwidth: data compression

Related Work on mobile visualization

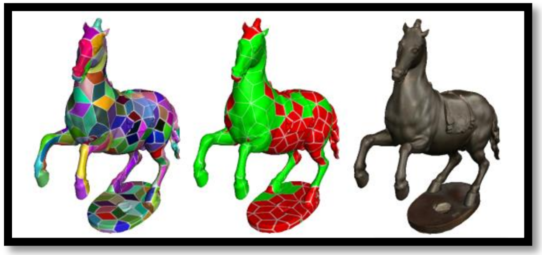
- ***remeber previous session for details***
- **Remote Rendering**
  - …..
- **Local Rendering**
  - Model based
    - Original models
    - Multiresolution models
    - Simplified models
      - Line rendering
      - Point cloud rendering
  - Image based
    - Image impostors
    - Environment maps
    - Depth images
  - Smart shading
  - Volume rendering

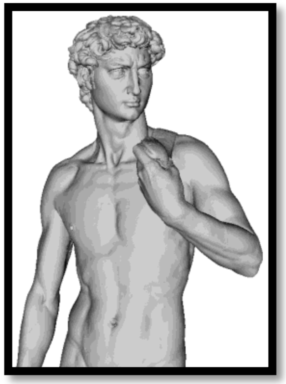
Related Work on mobile visualization

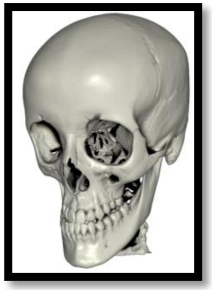
- ***remeber previous session for details***
- **Remote Rendering**
  - …..
- **Local Rendering**
  - Model based
    - Original models
    - **Multiresolution models**
    - Simplified models
      - Line rendering
      - Point cloud rendering
  - Image based
    - Image impostors
    - **Environment maps**
    - Depth images
  - **Smart shading**
  - **Volume rendering**

Scalable Mobile Visualization

- **Big/complex models:**
  - Detailed scenes from modeling, capturing..
    - Output sensitive: adaptive multiresolution
    - Compression / simple decoding
- **Complex rendering**
  - Global illumination
    - Pre-computation
    - Smart shading
  - Volume rendering
    - Compression / simple decoding







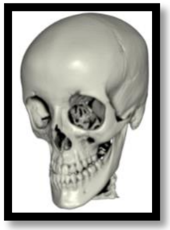
Scalable Mobile Visualization. Outline

Large meshes

High quality illumination: full precomputation

High quality illumination: smart computation

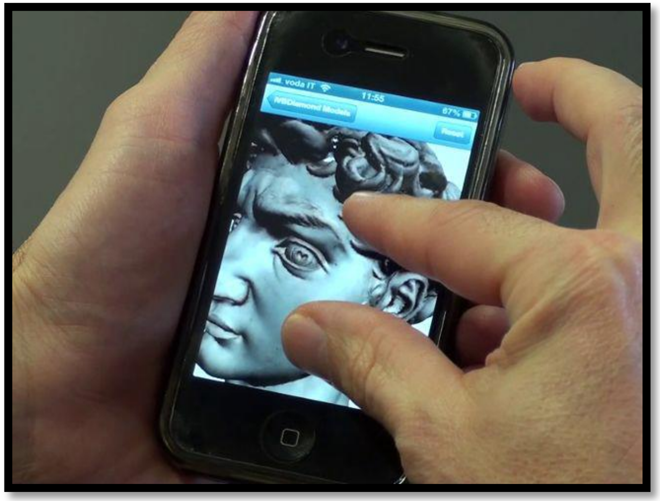
Volume data



Scalable Mobile Visualization

LARGE MESHES

St. Matthew 374M Tri



David 1 G Tri

Scalable Mobile Visualization

Phenomenal

Cosmic

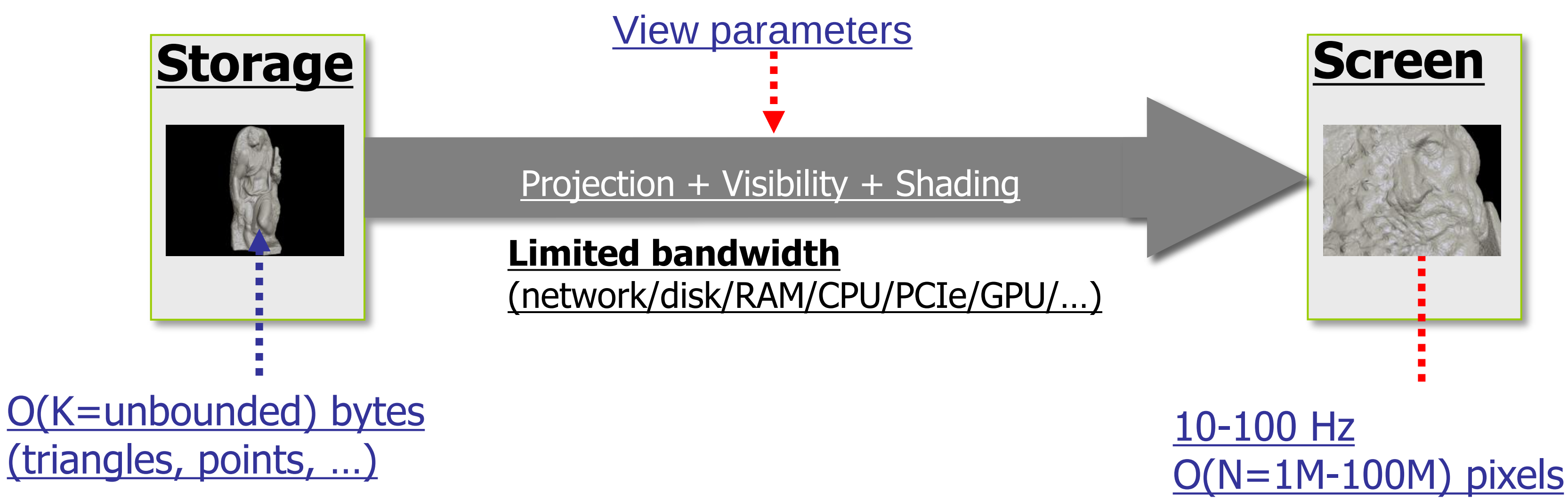
Models

Scalable Mobile Visualization

Itty bitty living space!

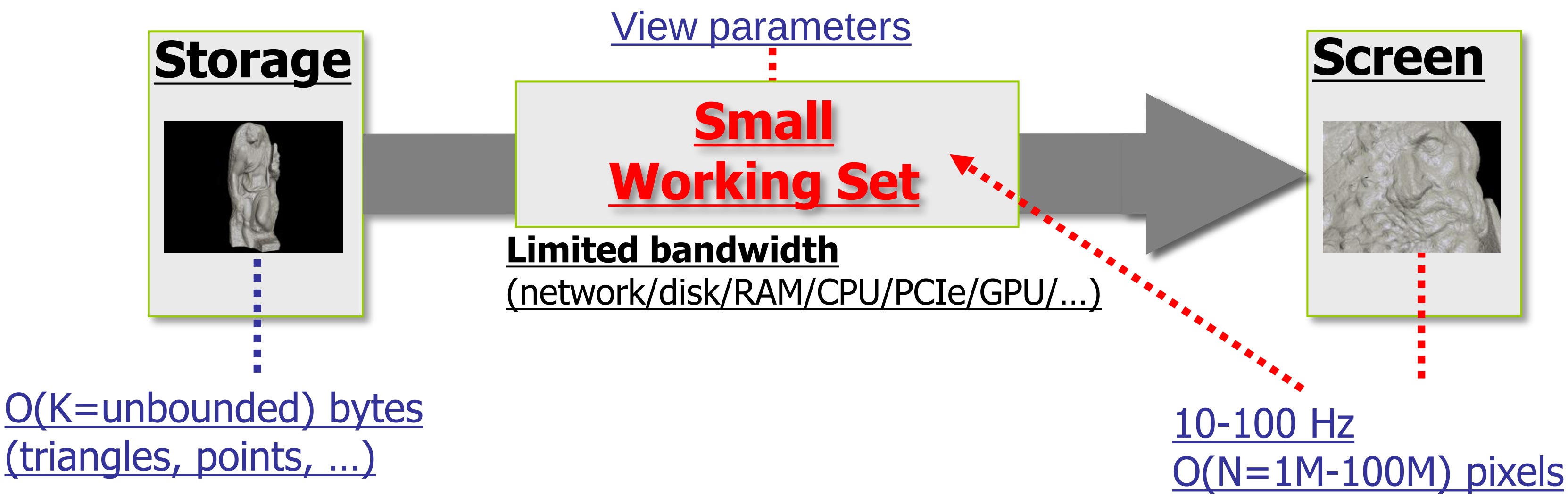
A real-time data filtering problem!

- **Models of unbounded complexity on limited computers**
  - Need for output-sensitive techniques (O(N), not O(K))
    - We assume less data on screen (N) than in model (K →∞)



A real-time data filtering problem!

- **Models of unbounded complexity on limited computers**
  - Need for output-sensitive techniques (O(N), not O(K))
    - We assume less data on screen (N) than in model (K →∞)



Proposed approaches

- **Output sensitive techniques: adaptive multiresolution**
  - Algorithm complexity proportional to pixel count, not to model size
- **Chunk-based multiresolution structures**
  - Amortize selection costs over groups of primitives
  - Same structure used for visibility and detail culling
- **Seamless combination of chunks**
  - Dependencies ensure consistency at the level of chunks
- **Data compression**
  - Fast GPU decompression or compression domain rendering
- **Chunk-based external memory management**
  - Streaming, compressed data, caching
- **Minimize CPU workload**
  - Move computation to GPU
- **Complex rendering primitives**
  - GPU programming features (curvilinear patches)

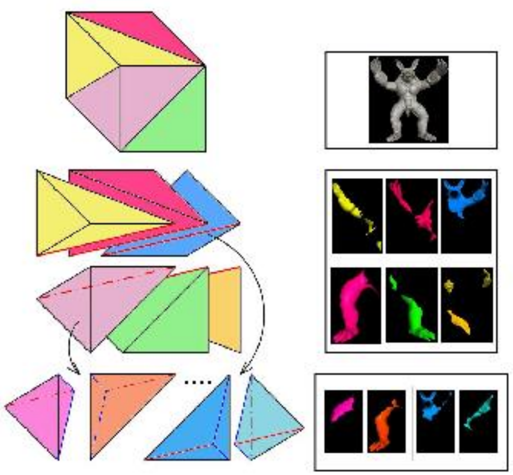
Mobile mandatory requirements

- **Output sensitive techniques: adaptive multiresolution**
  - Algorithm complexity proportional to pixel count, not to model size
- **Chunk-based multiresolution structures**
  - Amortize selection costs over groups of primitives
  - Same structure used for visibility and detail culling
- **Seamless combination of chunks**
  - Dependencies ensure consistency at the level of chunks
- **Data compression**
  - Fast GPU decompression or compression domain rendering
- **Chunk-based external memory management**
  - Streaming, compressed data, caching
- **Minimize CPU workload**
  - Move computation to GPU
- **Complex rendering primitives**
  - GPU programming features (curvilinear patches)

Chunked multiresolution structures

- **Two surface representation approaches integrated in a common framework with**
  - Compression, Streaming, Rendering

  - **Fixed coarse subdivision**
    - Multiresolution inside patch
  - **Adaptive coarse subdivision**
    - Global multiresolution

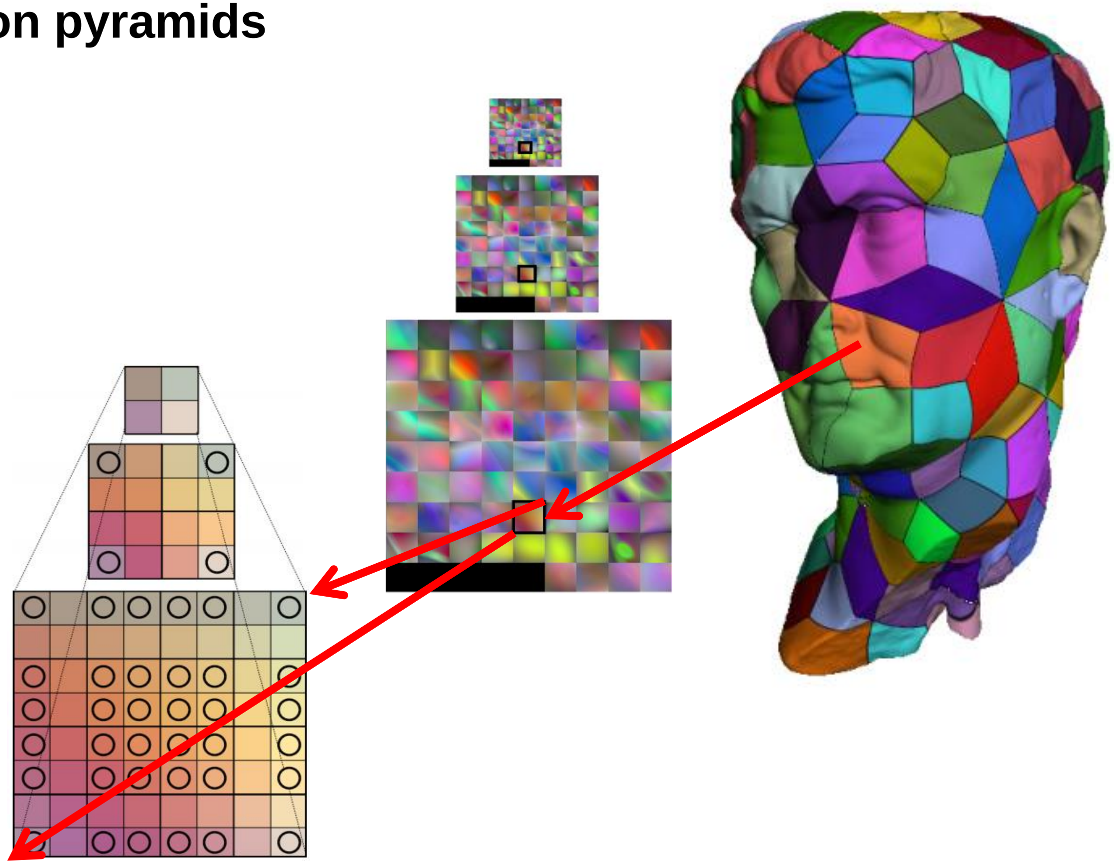


Generic approach for simple 3D models

- **Predefined structure with fixed number of quad patches**
- **Multiresolution structure per patch**
- **Compressed data loaded incrementally on demand**
- **Reuse components: compressed images.png**
- **Adaptive rendering handled almost totally in GPU**
- **Works both on Mobile and WebGL**
- **Works with topologically simple clean manifold meshes**

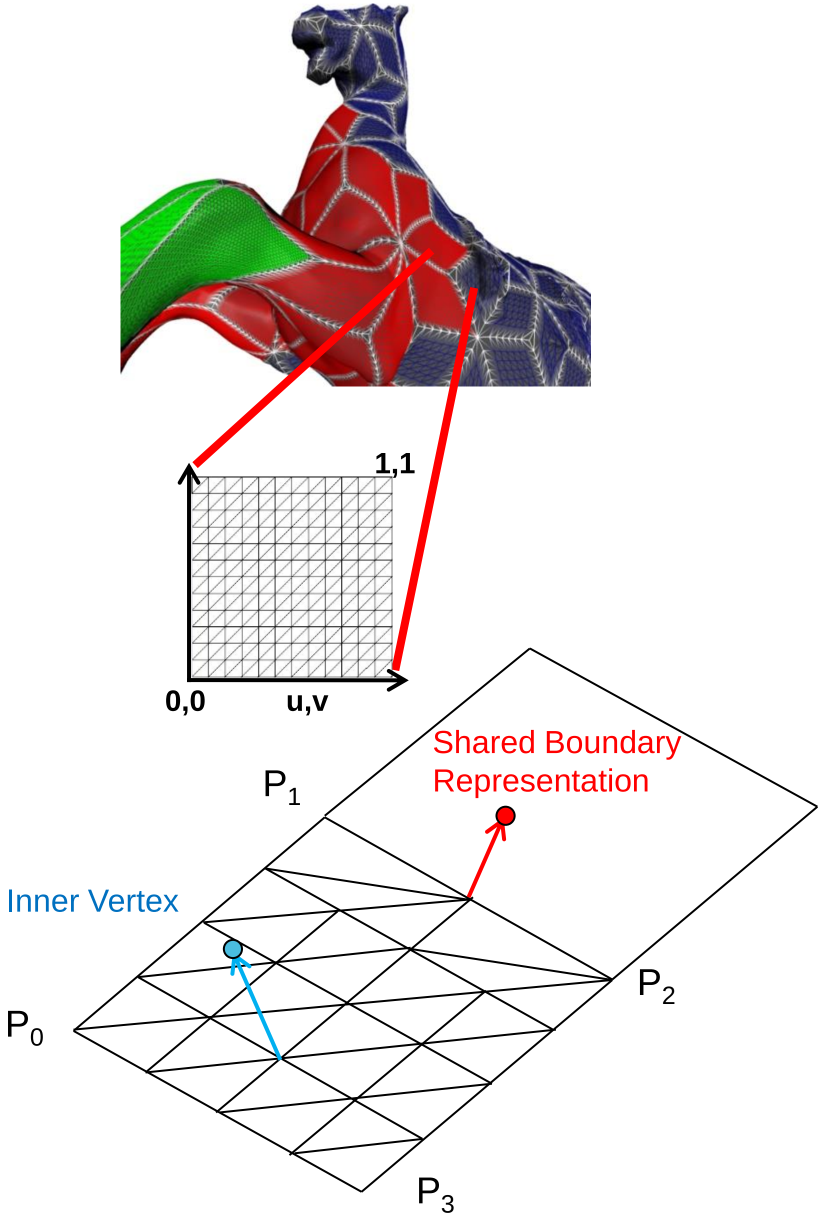
Adaptive Quad Patches:
simplified streaming & rendering for mobile & web

- **Models partitioned into fixed number of quad patches**
  - Geometry encoded as detail with respect to the 4 corners interpolation
- **For each quad: 3 multiresolution pyramids**
  - Detail geometry
  - Normals
  - Colors
- **Data encoded as images**
  - Exploit .png (lossless compression)
- **Ensure connectivity**
  - Duplicated boundary information



Adaptive rendering

- **1. CPU LOD Selection**
  - Find edge LODs
  - Quad LOD = max edge LODs
  - If data available use it, otherwise
    - Query data for next frames
    - Use best available representation
  - Send VBO with regular grid (1 for each LOD)
- **2. GPU: Vertex Shader**
  - Snap vertices on edges (match neighbors)
  - Base position = corner interpolation (u,v)
  - Displace VBO vertices
    - normal + displacement (dequantized)
- **3. GPU: Fragment Shader**
  - Texturing & Shading



Results

St. Matthew	374 M Tri
Avg bps (geo + col + norm)	24.3 (6.3 + 9.5 + 8.5)
Pixel Accuracy	1
FPS avg	37
FPS min	13
ADSL 8Mbps refine time	2s for model from scratch

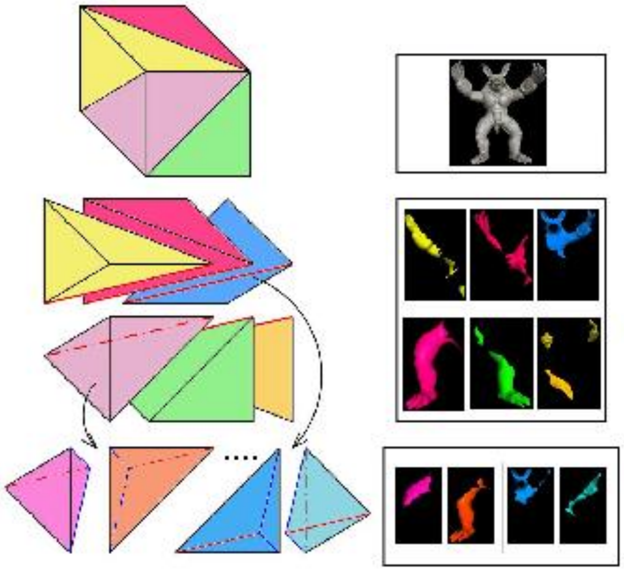
Conclusions: Adaptive Quad Patches

- **Effective creation and distribution system**
  - Fully automatic
  - Compact, streamable and renderable 3D model representations
  - Low CPU overhead → GPU adaptive rendering
  - Mobile, WebGL
- **Limitations**
  - Closed objects with large components (i.e,3D scanned objs)
- **Next ? More general method**

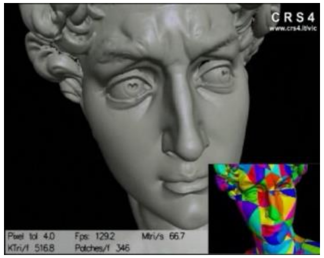
Compact Adaptive Tetra Puzzles

Efficient distribution and rendering for mobile

- **Built on Adaptive TetraPuzzles [CRS4+ISTI CNR, SIGGRAPH'04]**
- **More general models**
  - Regular conformal hierarchy of tetrahedra
  - Spatially partition input mesh
    - Mesh fragments at different resolutions
    - Associated to implicit diamonds

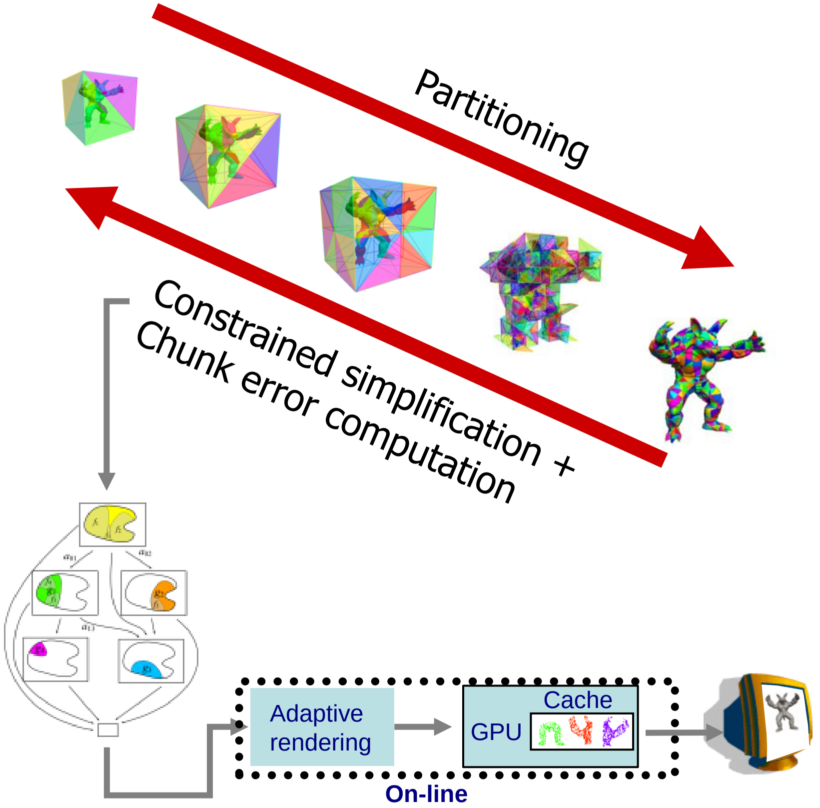


- **Objective**
  - Mobile
    - Limited resources / performance
  - Compact GPU representation
    - Good compression ratio (maximize resource usage)
    - Low decoding complexity (maximize decoding/rendering performance)



Overview

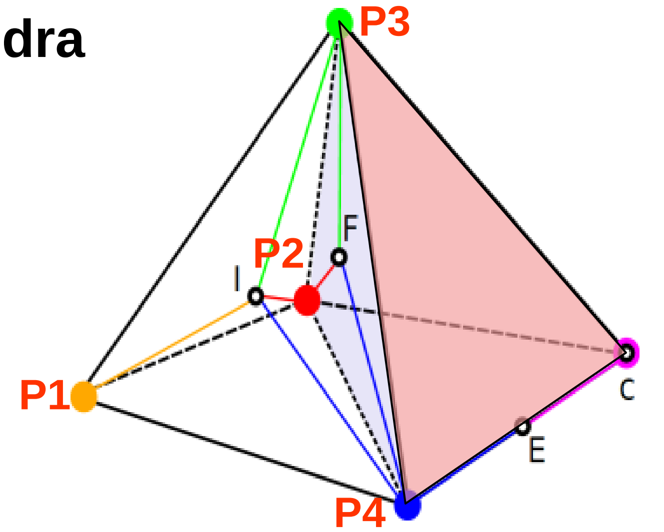
- **Construction**
  - Start with hires triangle soup
  - Partition model
  - Construct non-leaf cells by bottom-up recombination and simplification of lower level cells
  - Assign model space errors to cells
- **Rendering**
  - Refine graph
  - Render selected precomputed cells



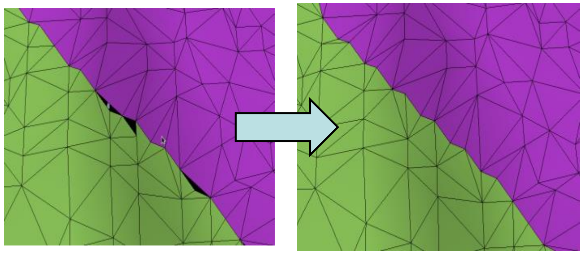
Ensure continuity → Shared information on borders

Our approach

- **Geometry clipped against containing tetrahedra**

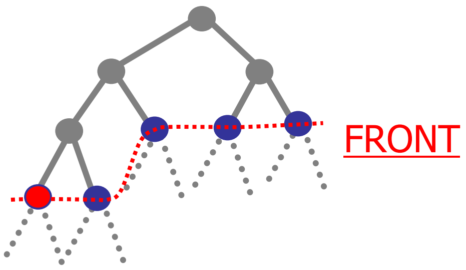


- **Vertices: tetrahedra barycentric coordinates**
  - $P_{barycentric} = \lambda_1 * P_1 + \lambda_2 * P_2 + \lambda_3 * P_3 + \lambda_{4*} P_4$
- **Seamless local quantization**
  - Inner vertices (I): 4 corners
  - Face vertices (F): 3 corners
  - Edge vertices (E): 2 corners
- **GPU friendly compact data representation**
  - 8 bytes = position (3 bytes) + color (3 bytes)+ normal(2 bytes)
  - Normals encoded with the octahedron approach [Meyer et al. 2012]

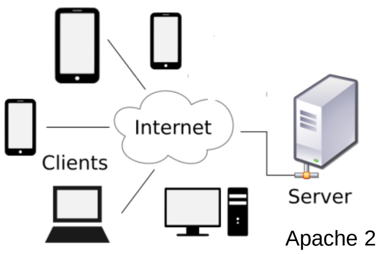


- **Further compression with entropy coding**
  - exploiting local data coherence

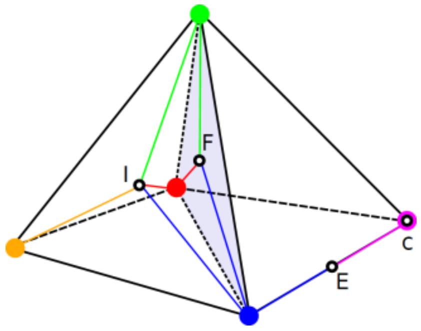
Rendering process



- **Extract view dependent diamond cut (CPU)**
- **Request required patches to server**
  - Asynchronous multithread client
  - Apache 2 based server (**data repository**, no processing)



- **CPU entropy decoding of each patch**
- **For each node (GPU Vertex Shader):**
  - VBO with barycentric coordinates, normals and colors (64 bpv)
  - Decode **position** : P = MV * [C0 C1 C2 C3] * [Vb]
    - Vb is the vector with the 4 barycentric coords
    - C0..C3 are tetrahedra corners
  - Decode **normal** from 2 bytes encoding [Meyers et al. 2012]
  - Use **color** coded in RGB24



Results

Rendering	iPad 3° gen	iPhone 4
Pixel tolerance	3	3
Triangle throughput	30 Mtri/s	2.8 Mtri/s
FPS avg	35	10
FPS refined views	15	2.8
Triangle Budget	2 M	1 M

- **Input Models**
  - St. Matthew 374 MTri
  - David 1GTri
- **Compression:**
  - 40 to 50 bits/vertex
- **Streaming full screen view**
  - 30s on wireless,
  - 45s on 3G
  - David 14.5MB (1.1 Mtri)
  - St. Matthew 19.9MB (1.8 Mtri)

Conclusions: Compact ATP

- **Generic gigantic 3D triangle meshes on common handheld devices**
  - Compact, GPU friendly, adaptive data structure
    - Exploiting the properties of conformal hierarchies of tetrahedra
    - Seamless local quantization using barycentric coordinates
  - Two-stage CPU and GPU compression
    - Integrated into a multiresolution data representation
- **Limitations**
  - Requires coding non-trivial data structures
  - Hard to implement on scripting environments

Conclusions: large meshes

- **Various solutions for large meshes**

- **Constrained solution: Adaptive Quad Patches**
  - Simple and fast
  - Good compression
  - Works on topologically simple models

- **General solution: Compact Adaptive Tetra Puzzles**
  - Compact data representation
  - More complex code

Complex scenes

- **We have seen how to deal with complex models O(Gtri)**
- **How to deal with real time mobile complex illumination?**
- **Two options:**
  - Full precomputation
  - Smart computation

Scalable Mobile Visualization

COMPLEX LIGHTING: FULL PRECOMPUTATION

Ubiquitous exploration of scenes with complex illumination

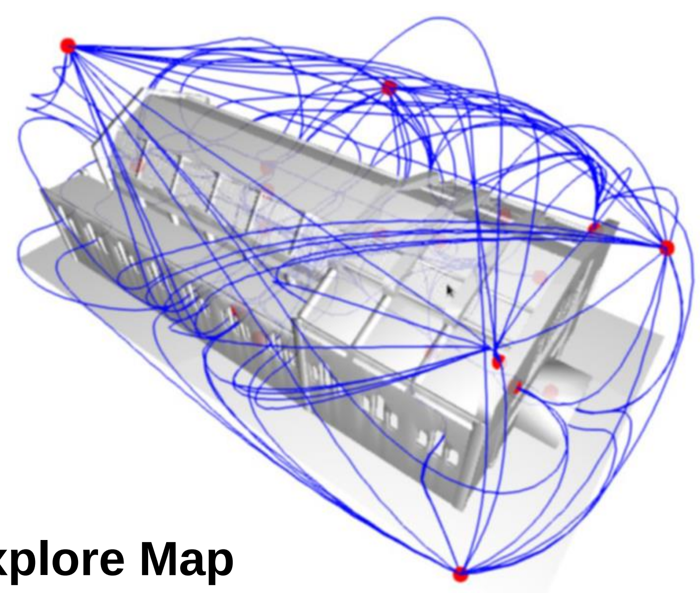
- **Real-time requirement: ~30Hz**
  - Difficulties handling complex illumination on mobile/web platforms with current methods
- **Image-based techniques**
  - Constraining camera movement to a set of fixed camera positions
  - Enable pre-computed photorealistic visualization
- **Explore-Maps: technique for**
  - Scene representation as set of probes and arcs
  - Precomputed rendering for probes and transitions

Scene Discovery

- **ExploreMaps: Automatic method for generating**
  – Set of probes providing full model coverage
    - Probe = 360° panoramic point of view
  – Set of arcs connecting probes
    - Enable full scene navigation

Gobbetti et al. Eurographics 2014

ExploreMaps: Efficient Construction and Ubiquitous Exploration of Panoramic View Graphs of Complex 3D Environments.

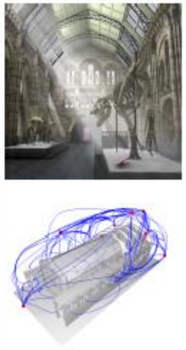
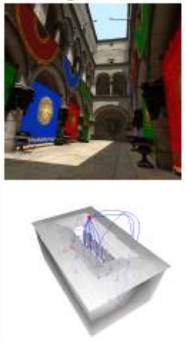
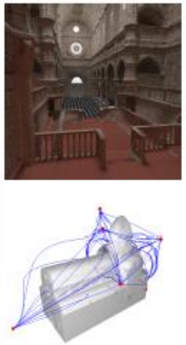
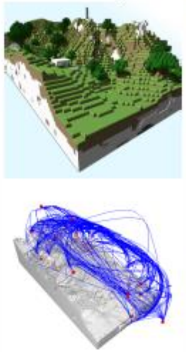
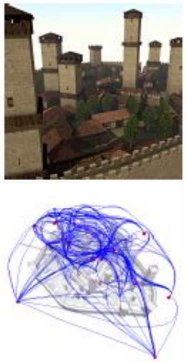
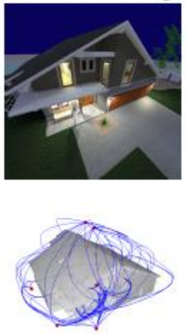
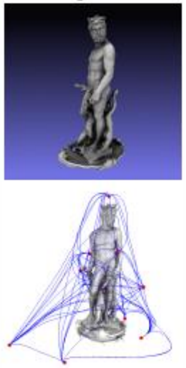


Explore Map

Dataset Creation (rendering)

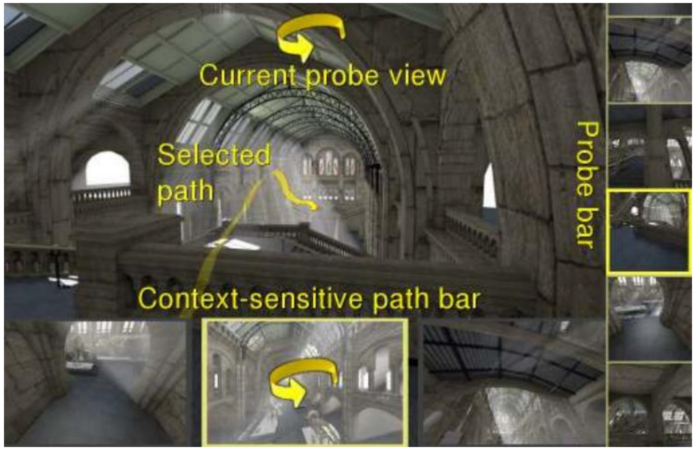
- **Input: Explore Map**
  - Probes with full scene coverage
  - Transitions between “reachable” probes
- **Pre-processing**
  - Photorealistic rendering (using Blender 2.68a)
    - panoramic views both for probes and transition arcs
  - 1024^2 probe panoramas
  - 256^2 transition video panoramas
  - 32 8-core PCs,
  - Rendering times ranging from 40 minutes to 7 hours/model

Explore Maps – Processing Results

	Museum	Sponza	Sibenik	Lighthouse	Lost Empire	Medieval Town	German Cottage	Neptune
								
Input								
#tri	1,468,140	262,267	69,853	48,940	157,136	14,865	79,400	2,227,359
Output								
#probes	70	36	92	57	74	78	140	79
#clusters	17	10	21	17	25	30	23	19
#paths	127	29	58	81	206	222	102	93
Time (s)								
Exploration	154	23	63	15	41	34	163	38
Clustering	17	3	27	8	13	14	118	14
Synthesis	144	35	449	453	284	395	427	279
Path	7	1	31	12	22	80	23	13
Path smoothing	3,012	122	81	89	482	199	185	150
Thumbn.	11	3	7	5	8	10	7	6
Thumbn. pos	2	2	1	1	4	4	2	1
Total	3,347	189	659	583	854	736	925	501
Storage (MB)								
Probes	59	28	72	59	86	103	79	43
Paths	248	146	113	159	371	376	390	120

Interactive Exploration

- UI for Explore Maps
  - WebGL implementation + JPEG + MP4
  - Panoramic images: probes + transition path
- Closest probe selection
  - Path alignment with current view
- Thumbnail goto
  - Non-fixed orientation



Conclusion: Interactive Exploration

- **Interactive exploration of complex scenes**
  - Web/mobile enabled
  - Pre-computed rendering
    - state-of-the-art Global Illumination
  - Graph-based navigation → guided exploration
- **Limitations**
  - Constrained navigation
    - Fixed set of camera positions
  - Limited interaction
    - Exploit panoramic views on paths → less constrained navigation
- **Next part of the talk:**
  - A dynamic solution for complex illumination with smart computation

Scalable Mobile Visualization

COMPLEX LIGHTING: SMART COMPUTATION

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
  - Nee 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**

Taken from [Kán et al., 2015]

Consistent illumination for AR

- **Performance**

3D model	# triangles	Framerate
Reflective cup	25.6K	29 fps
Teapot	15.7K	30 fps
Dragon	229K	13 fps

- **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

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

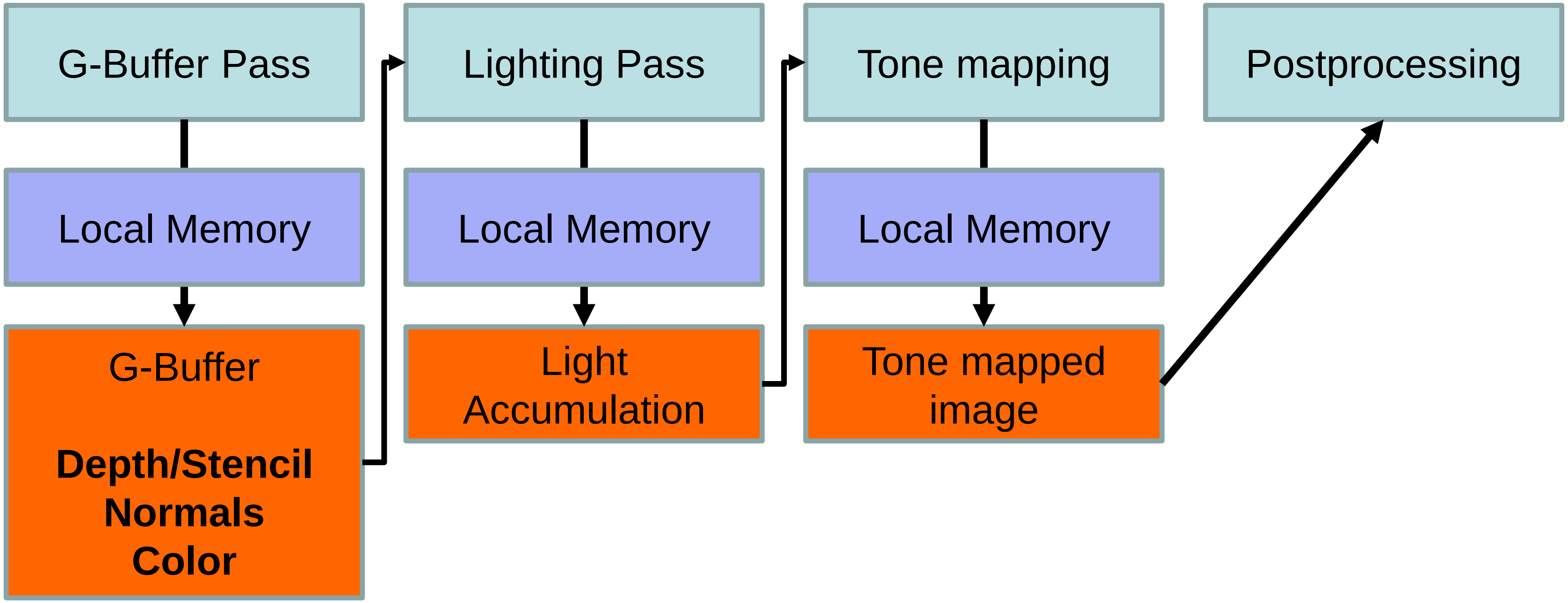
- **Conclusions**
  - Does not need to render to texture
    - Cubemaps must be pre-calculated
  - Requires reading multiple times from textures
  - Stable
    - Because cubemap does not change
- **Limitations**
  - Static, since info is precomputed

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

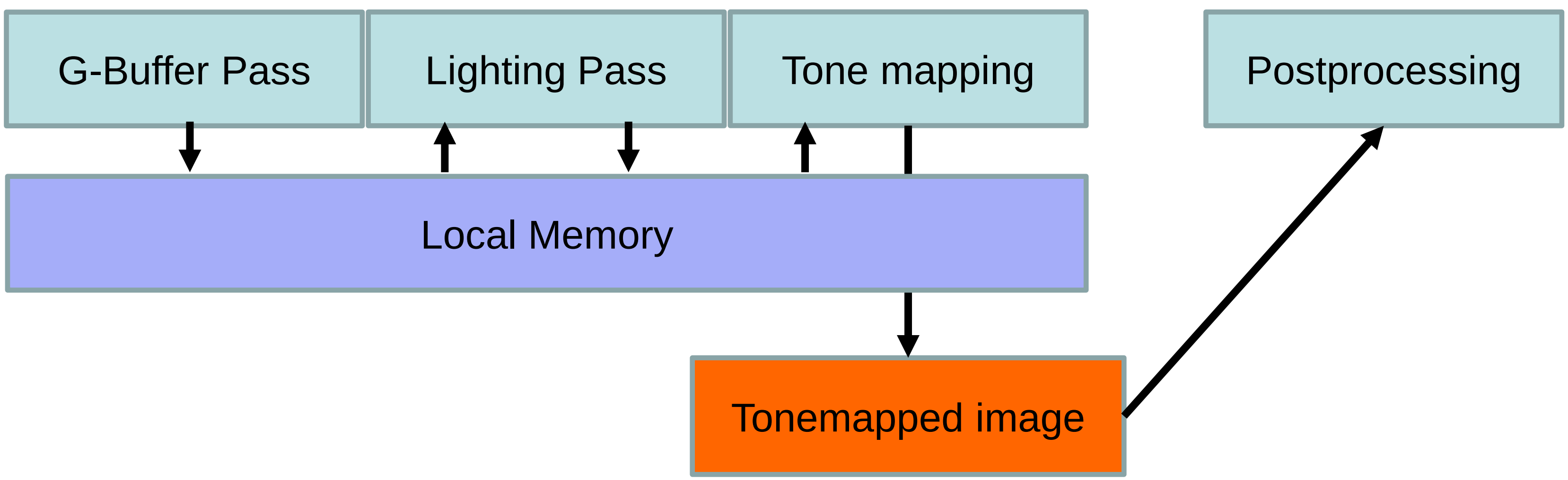
Physically-based Deferred Rendering

- **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



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

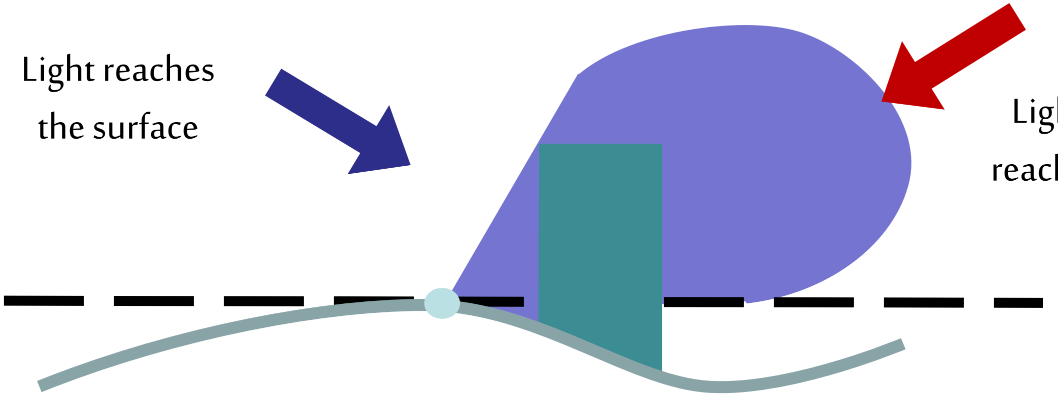
Ambient Occlusion in mobile

- **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



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

$$\rho(d) = \begin{cases} f(d) \in [0,1] & d < \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

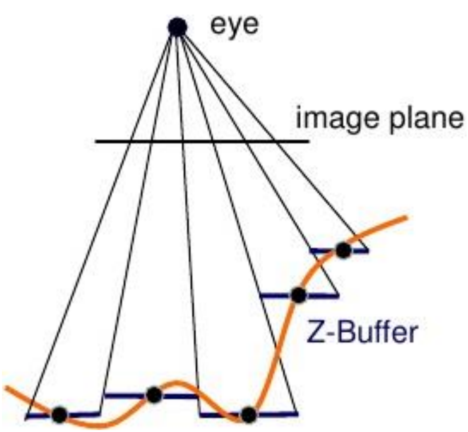
Ambient Occlusion in mobile

- **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



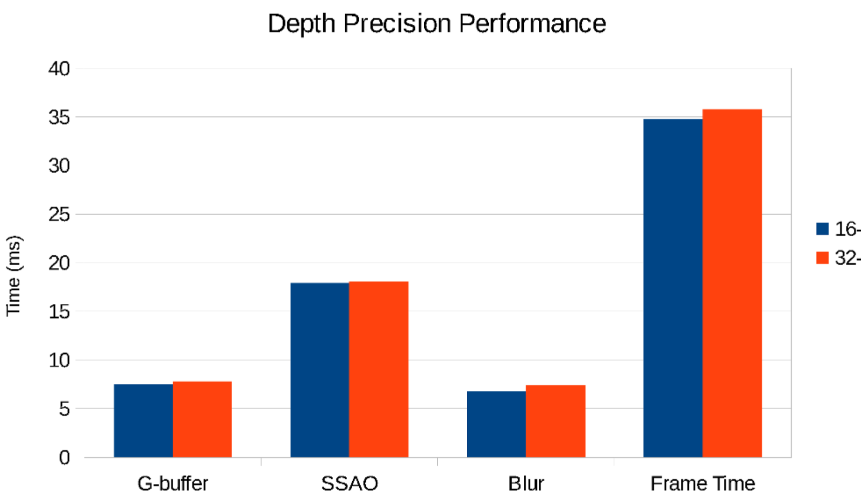
Assassin's Creed Unity

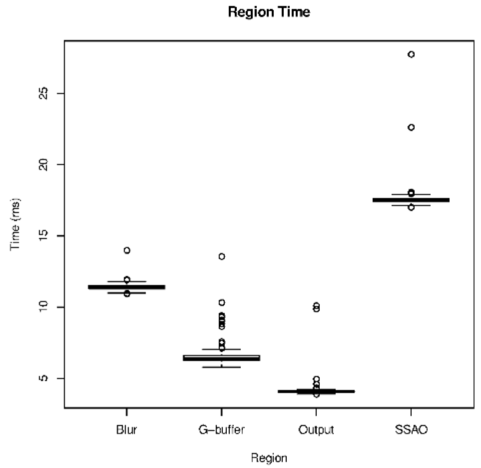
Ambient Occlusion in mobile

- **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

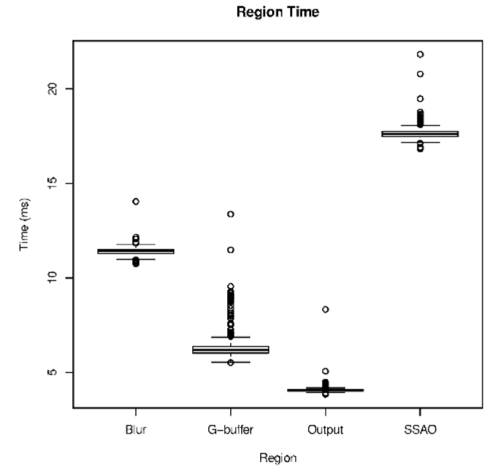
Ambient Occlusion in mobile

- **Optimizations. G-Buffer storage**
  - G-Buffer with less precision (32, 16, 8)
    - 8 not enough
    - 16 and 32 similar quality
  - Normal storage (RGB vs RG)
    - RGB normals are faster





RGB normals.



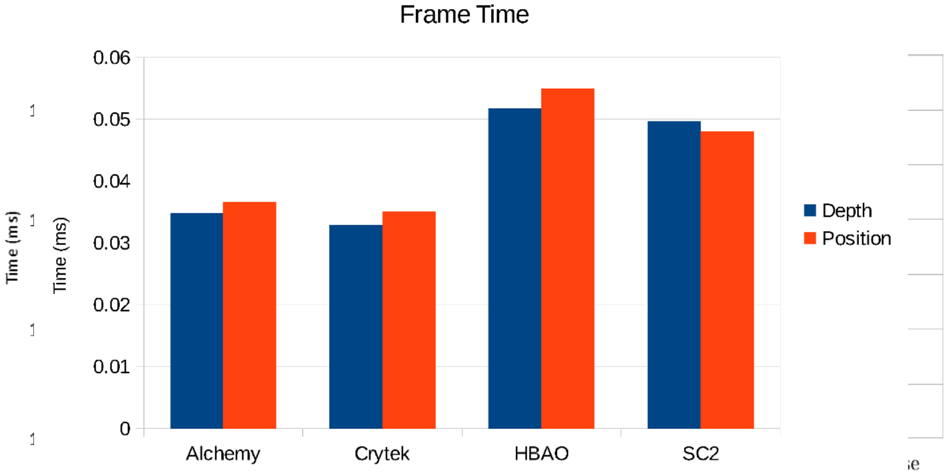
RG normals.

Ambient Occlusion in mobile

- **Optimizations. Sampling**
  - AO samples generation (disc and hemisphere)
    - Desktops use up to 32
    - With mobile, 8 is the affordable amount
      - Pseudo-random samples produces noticeable patterns
  - Our proposed solution
    - Compute sampling patterns offline
      - 2D: 8-point Poisson disc
      - 3D: 8-point cosine-weighted hemisphere (Malley's approach, as in [Pharr and Humprheys, 2010])
    - Scaling and rotating the resulting pattern ([Chapman, 2011])
    - Predictable, reproducible, robust

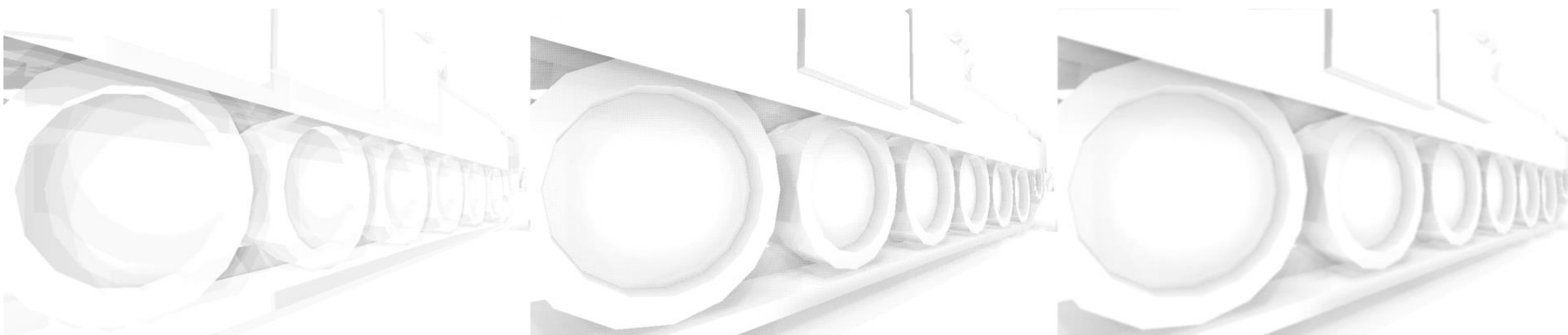
Ambient Occlusion in mobile

- **Optimizations. Getting geometry positions**
  - Transform samples to 3D
    - Inverse transform vs similar triangles
      - Precision for speed
    - Similar triangles are faster
  - Storing depth vs storing 3D positions in G-Buffer
    - Trades bandwidth for memory
    - Depth slightly better
    - Better profile for the application



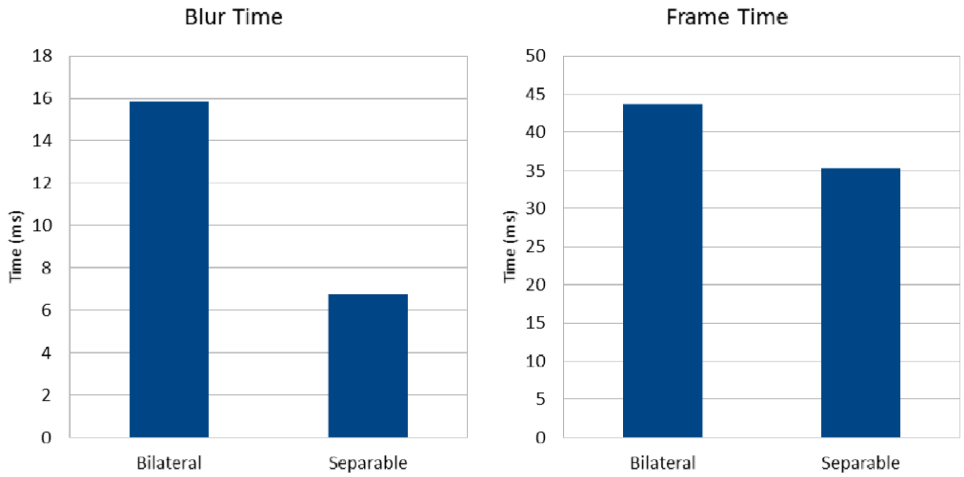
Ambient Occlusion in mobile

- **Optimizations. Banding & Noise**
  - Fixed sampling pattern produces banding (left)
  - Random sampling reduces banding but adds noise (middle)
  - SSAO output is typically blurred to remove noise (right)
    - But blurs edges



Ambient Occlusion in mobile

- **Optimizations. Banding & Noise**
  - User bilateral filter instead
    - Works better
    - Improve timings with separable filter

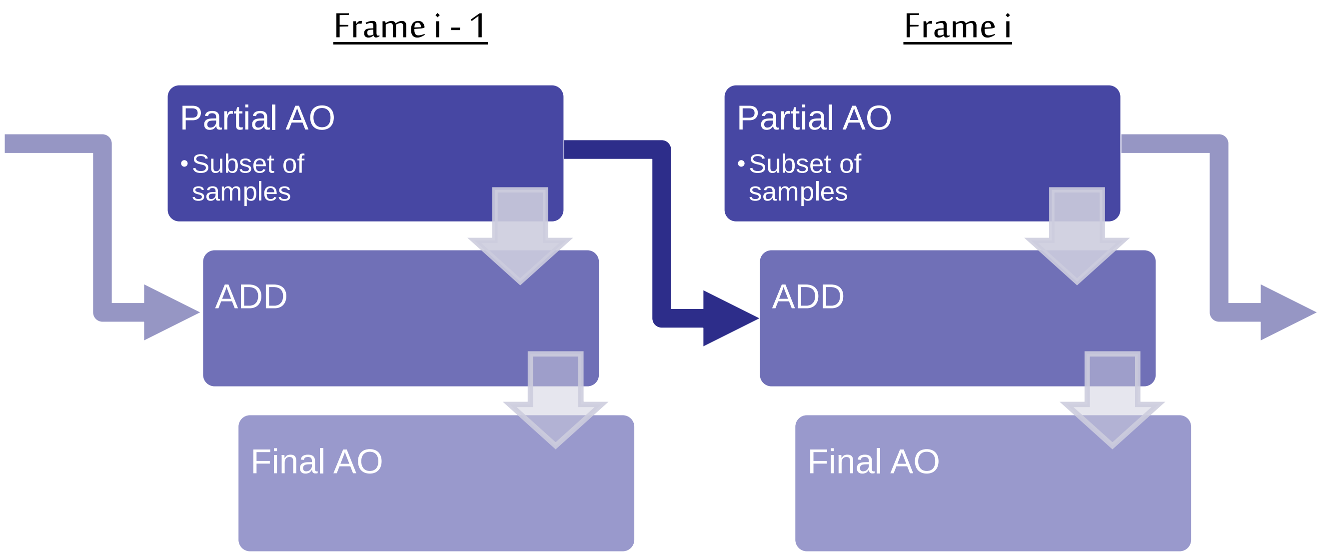


$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(||p-q||)\, G_{\sigma_r}(|I_q - I_p|)\, I_q$$

$$G_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

Ambient Occlusion in mobile

- **Optimizations. Progressive AO**
  - Amortize AO throughout many frames



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

Ambient Occlusion in mobile

- **Results**

Algorithm	Optimized (not progressive)	Optimized + progressive
Starcraft 2	17.8%	38.5%
HBAO	25.6%	39.2%
Crytek	23.4%	35.0%
Alchemy	24.8%	38.2%

Ambient Occlusion in mobile

- **Conclusions**
  - Developed an optimized pipeline for mobile AO
    - Analyzed the most popular AO techniques
      - Improved several important steps of the pipeline
      - Proposed some extra contributions (e.g. progressive AO)
    - Achieved realtime framerates with high quality
    - Developed techniques can be used in WebGL
  - Future Work
    - Further improvement of the pipeline
    - Developing "Homebrew" method
      - With all known improvements
      - Some extra tricks
      - Not ready for prime time yet

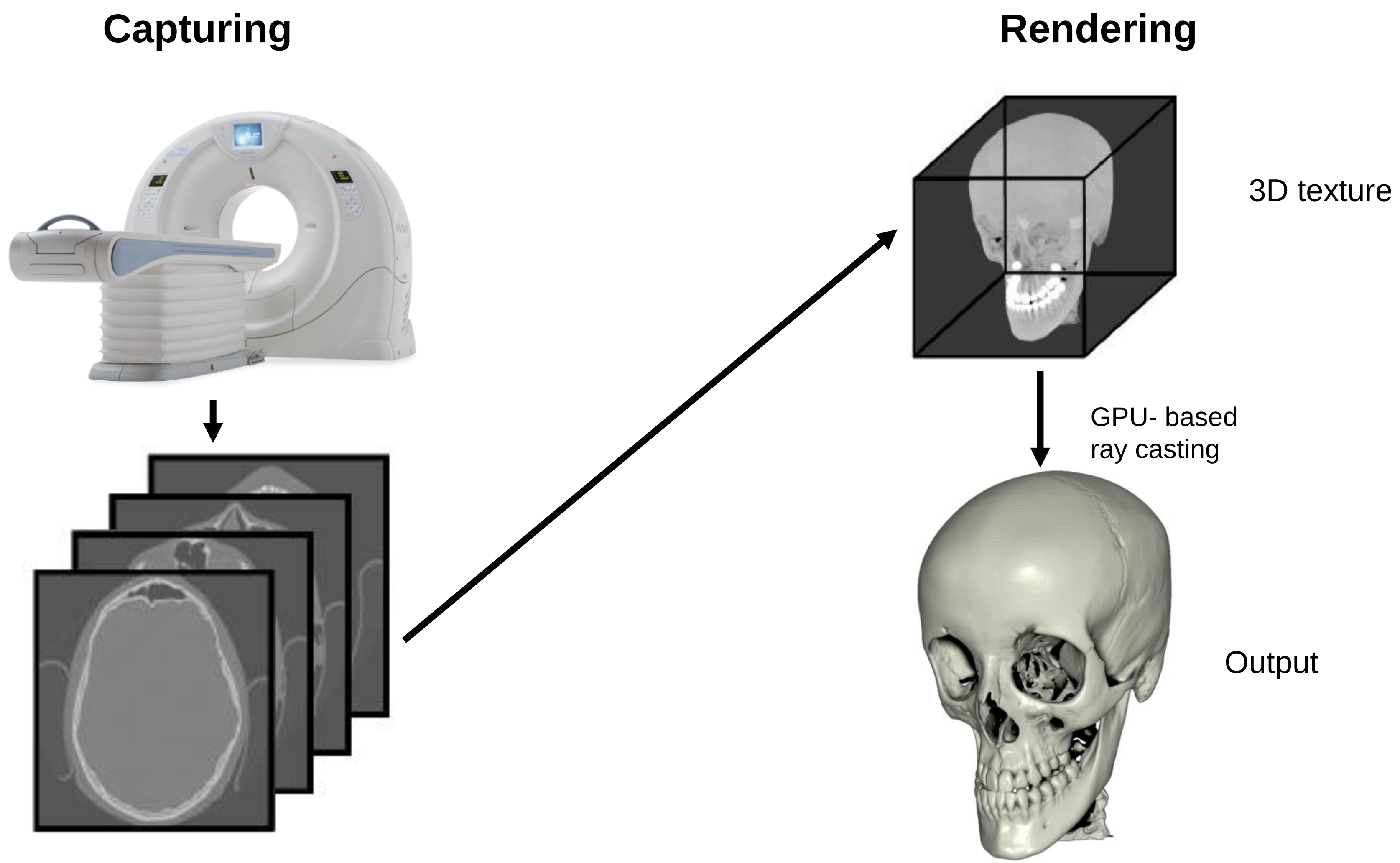
Scalable Mobile Visualization

VOLUMETRIC DATA

Rendering Volumetric Datasets

- **Introduction**
- **Challenges**
- **Architectures**
- **GPU-based ray casting on mobile**
- **Conclusions**

Rendering Volumetric Datasets



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

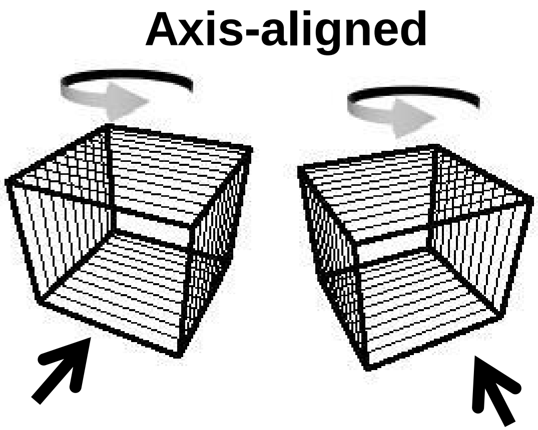
Rendering Volumetric Datasets

- **Pure mobile rendering**
  - Move all the work to the mobile
  - Nowadays feasible

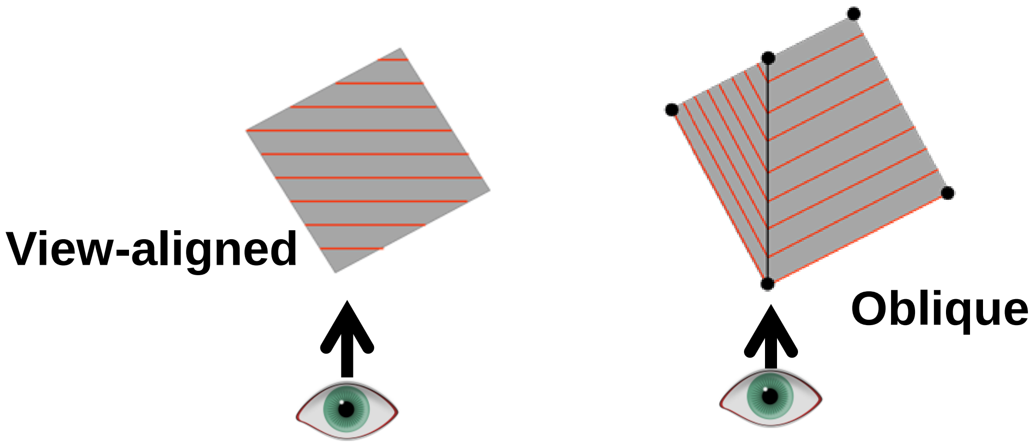
- **Direct Volume Rendering on mobile. Algorithms**
  - Slices
  - 2D texture arrays
  - 3D textures

Rendering Volumetric Datasets

- **Slices**
  - Typical old days volume rendering
    - Several quality limitations
    - Subsampling & view change



  - Improvement: Oblique slices [Kruger 2010]

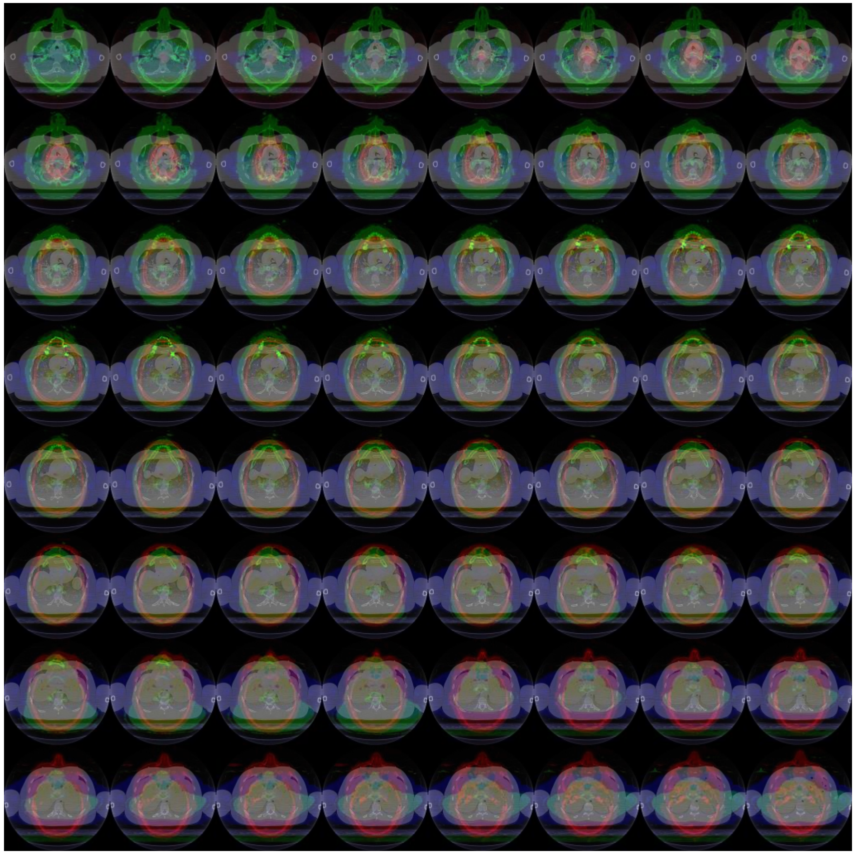


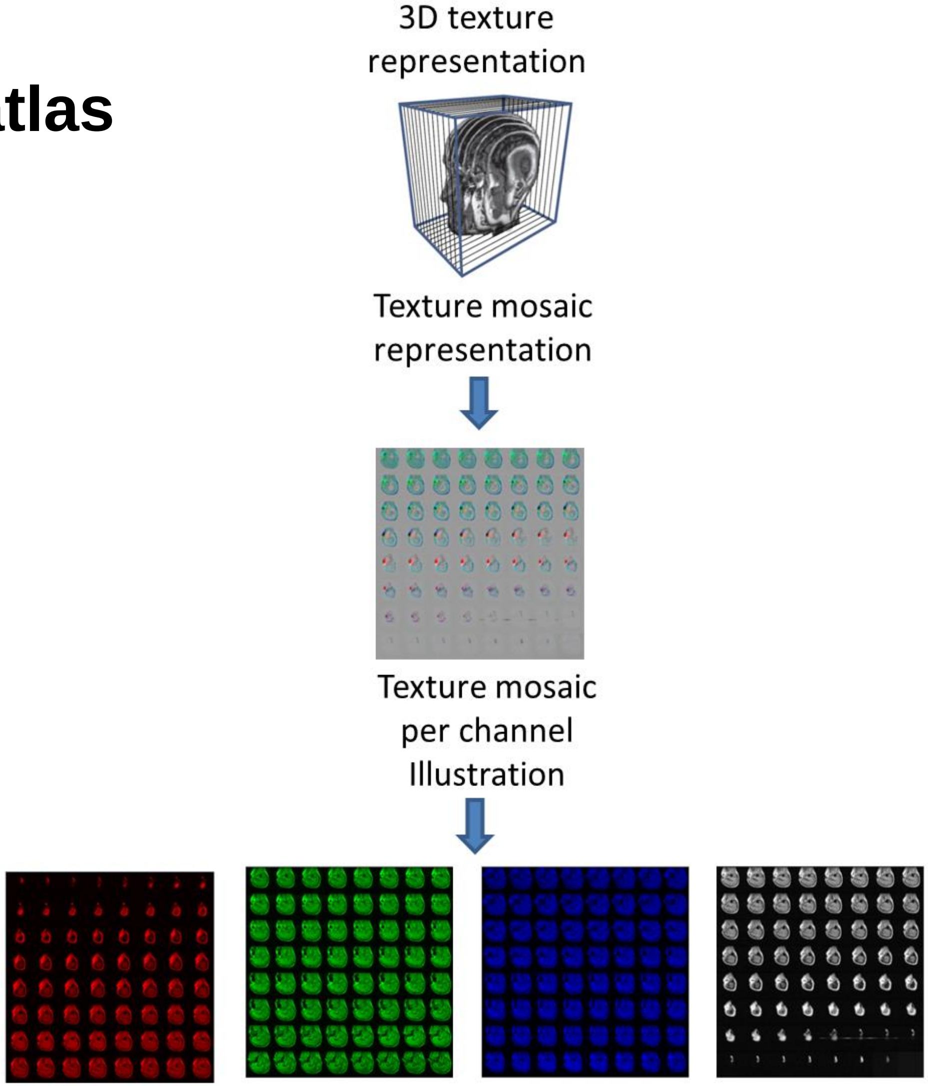
Rendering Volumetric Datasets

- **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

- **2D texture arrays + compression [Valencia & Vázquez, 2013]**
  - Increase the supported sizes
  - Increase framerates

Compression format	Compression ratio	RBA format	RGBA format	GPU support	Overall performance	Overall quality
ETC1	4:1	Yes	No	All GPUs	Good (RC)	Good
PVRTC	8:1 and 16:1	Yes	Yes	PowerVR	Not so good	Bad
ATITC	4:1	Yes	Yes	Adreno	Good (RC)	Good

Rendering Volumetric Datasets

- **2D texture arrays + compression**
  - ATITC: improves performance from 6% to 19%. With an average of 13.1% and a low variance of performance.
  - ETC1(-P): improves performance from 6.3% to 69.5%. With an average of 32.6% and the highest variance of performance.
  - PVRTC-4BPP: improves performance from 4.7% and 36.% and PVRTC-2BPP: from 9,5% to 36,5%. The average performance of both methods is ~15% with high variance.

Rendering Volumetric Datasets

- **2D texture arrays + compression**
  - Ray-casting: gain performance in average of 33%.
  - Slice-based: gain performance in average o f 8%.
  - Ray-casting frame rates are better in all cases compared to slice-based.

Rendering Volumetric Datasets

- **2D texture arrays + compression**

Uncompressed

Compressed with ATI-I

Compressed with ETC1-P

Rendering Volumetric Datasets

- **2D texture arrays + compression**

Uncompressed

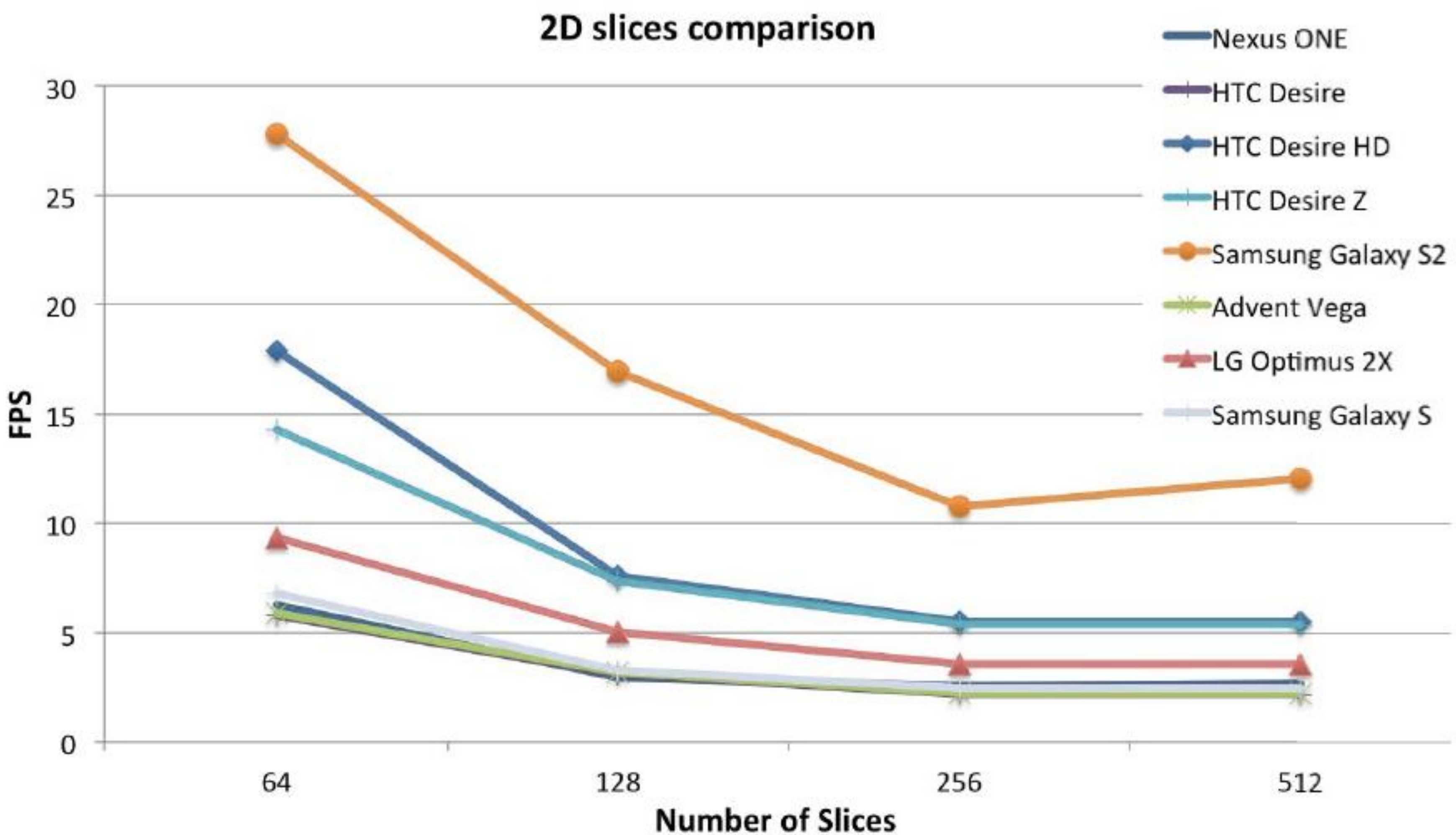
Compressed with PVRTC-4BPP

Compressed with PVRTC-2BPP

Rendering Volumetric Datasets

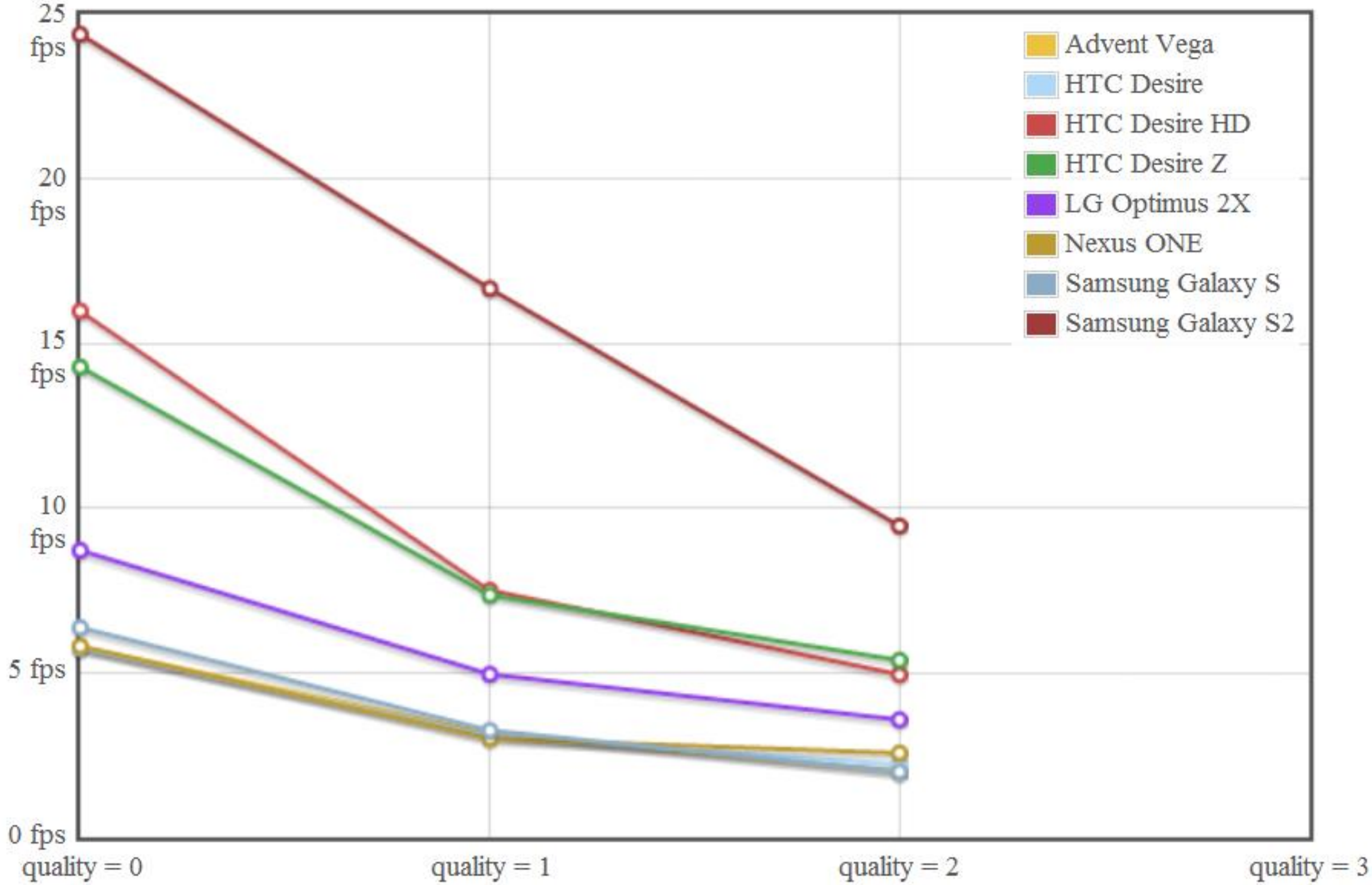
- **3D textures [Balsa & Vázquez, 2012]**
  - Allow either 3D slices or GPU-based ray casting
  - Initially, only a bunch of GPUs sporting 3D textures (Qualcomm's Adreno series >= 200)
  - Performance limitations (data: 256^3 – screen resol. 480x800)
    - 1.63 for 3D slices
    - 0.77 fps for ray casting

Rendering Volumetric Datasets



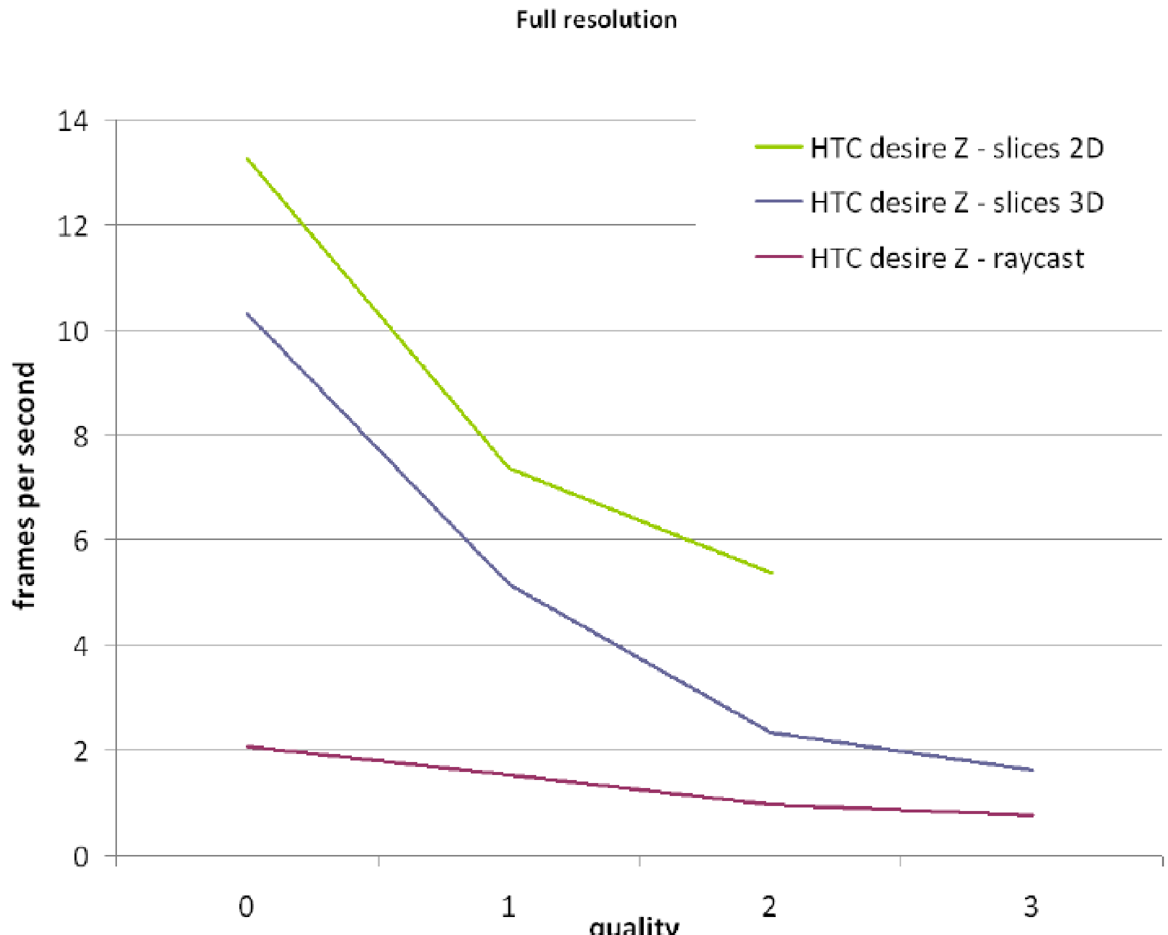
Rendering Volumetric Datasets

- **2D slices**



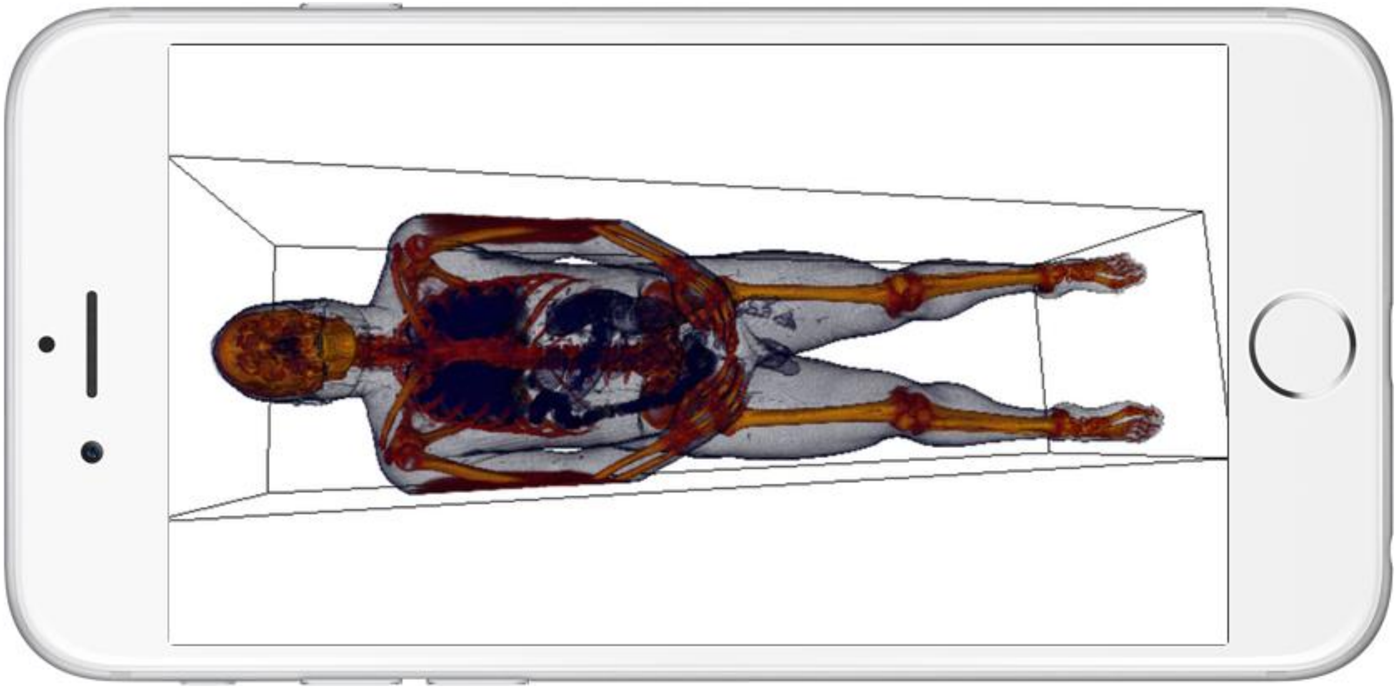
Rendering Volumetric Datasets

- **2D slices vs 3D slices vs raycasting**



Rendering Volumetric Datasets

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



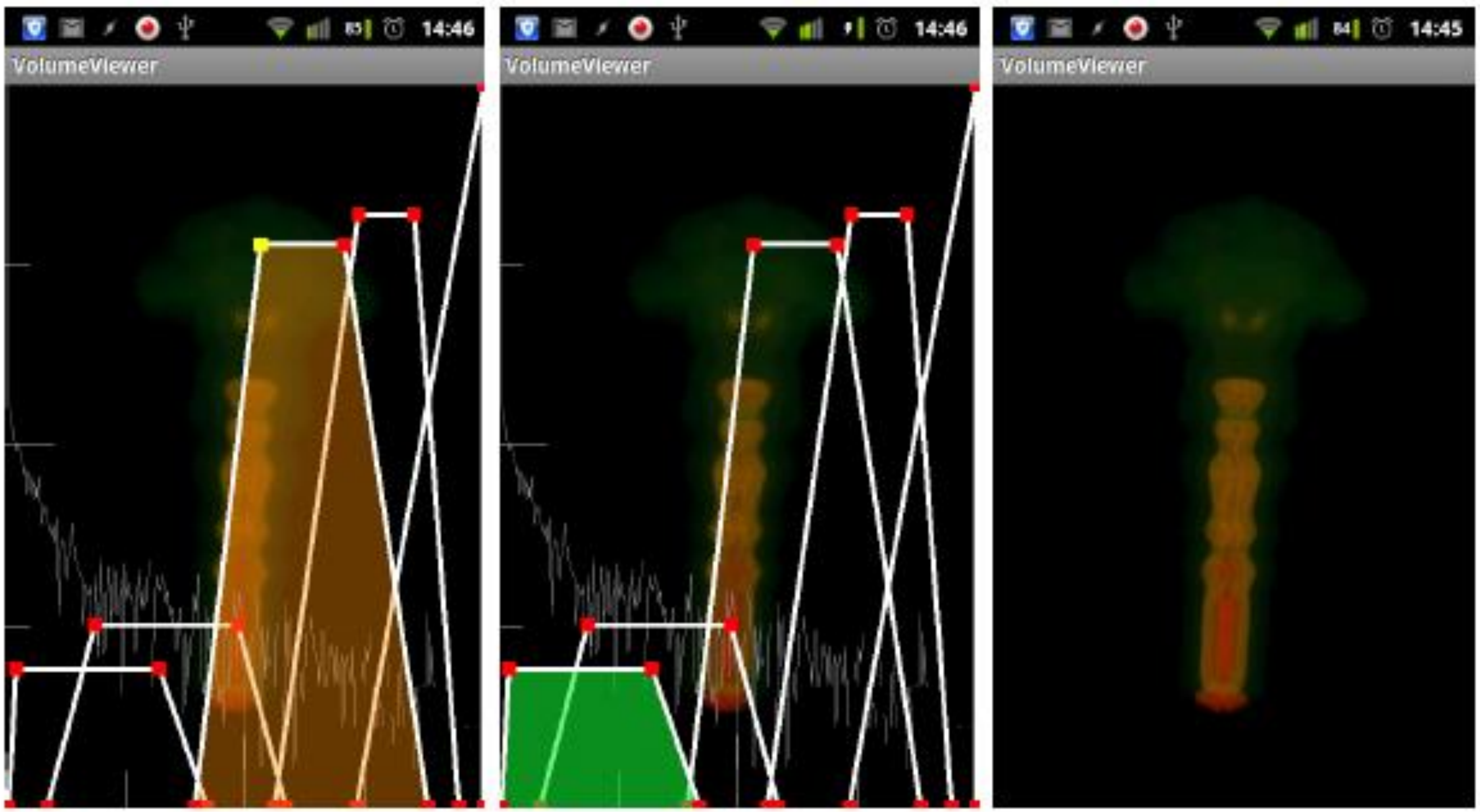
Taken from [Schiewe et al., 2015]

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 256^2x942)

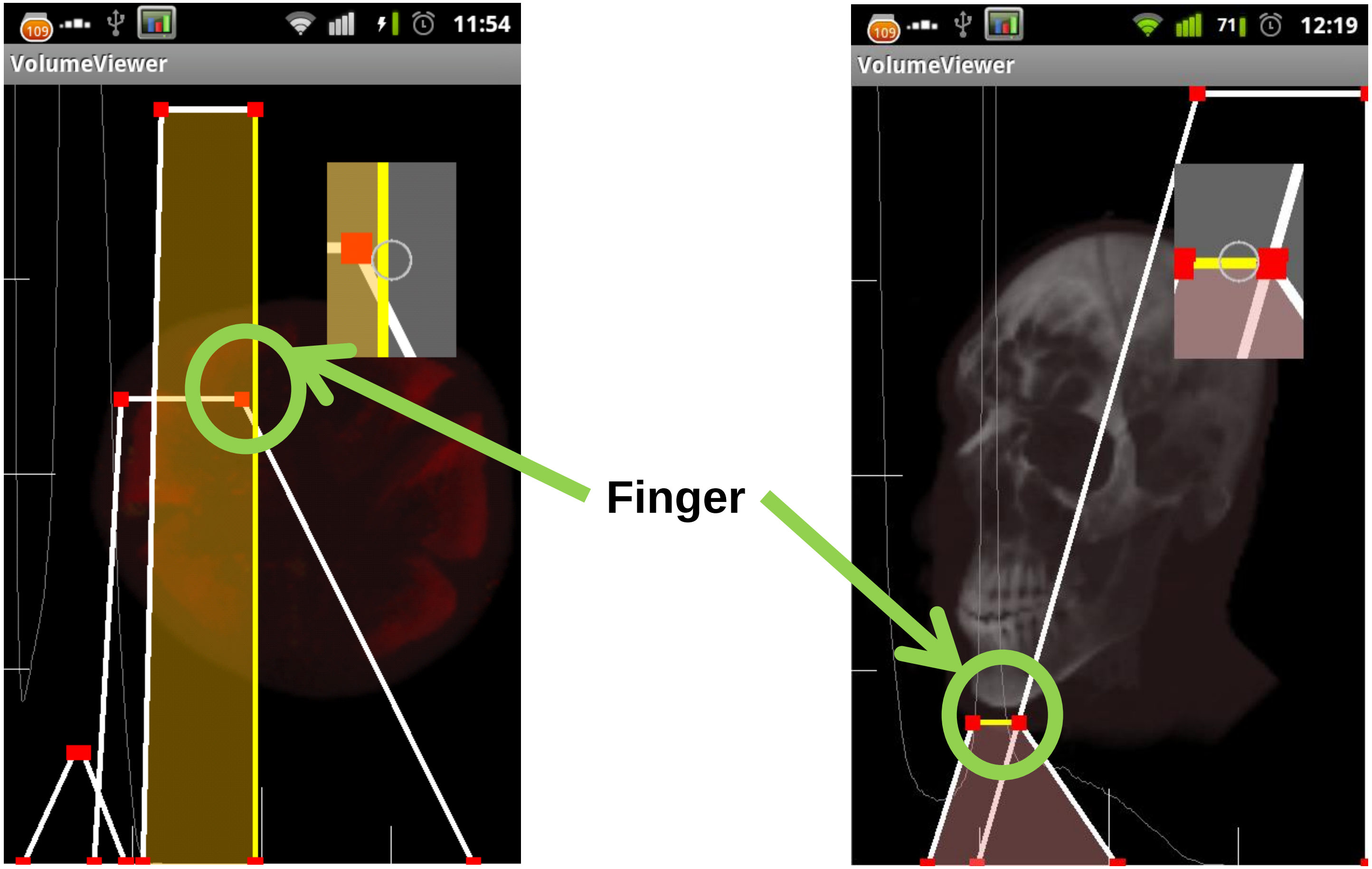
Rendering Volumetric Datasets

- **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
    - Still limited performance (~7fps…)
  - Interaction still difficult
  - Client-server architecture still alive
    - Can overcome data privacy/safety & storage issues
    - Better 4G-5G connections
    - …

Next Session

CLOSING
QUESTION & ANSWERS