Ambient Occlusion

Pere-Pau Vázquez

ViRVIG – UPC

Outline

- Concept
- Derivation
- Algorithms
- Practical issues
- Implementation in mobile

Outline

- **Introduction**
- Derivation
- Algorithms
- Practical issues
- Implementation in mobile

Introduction

- https://www.youtube.com/watch?v=1MuCGwoAH-U

Introduction. Concept

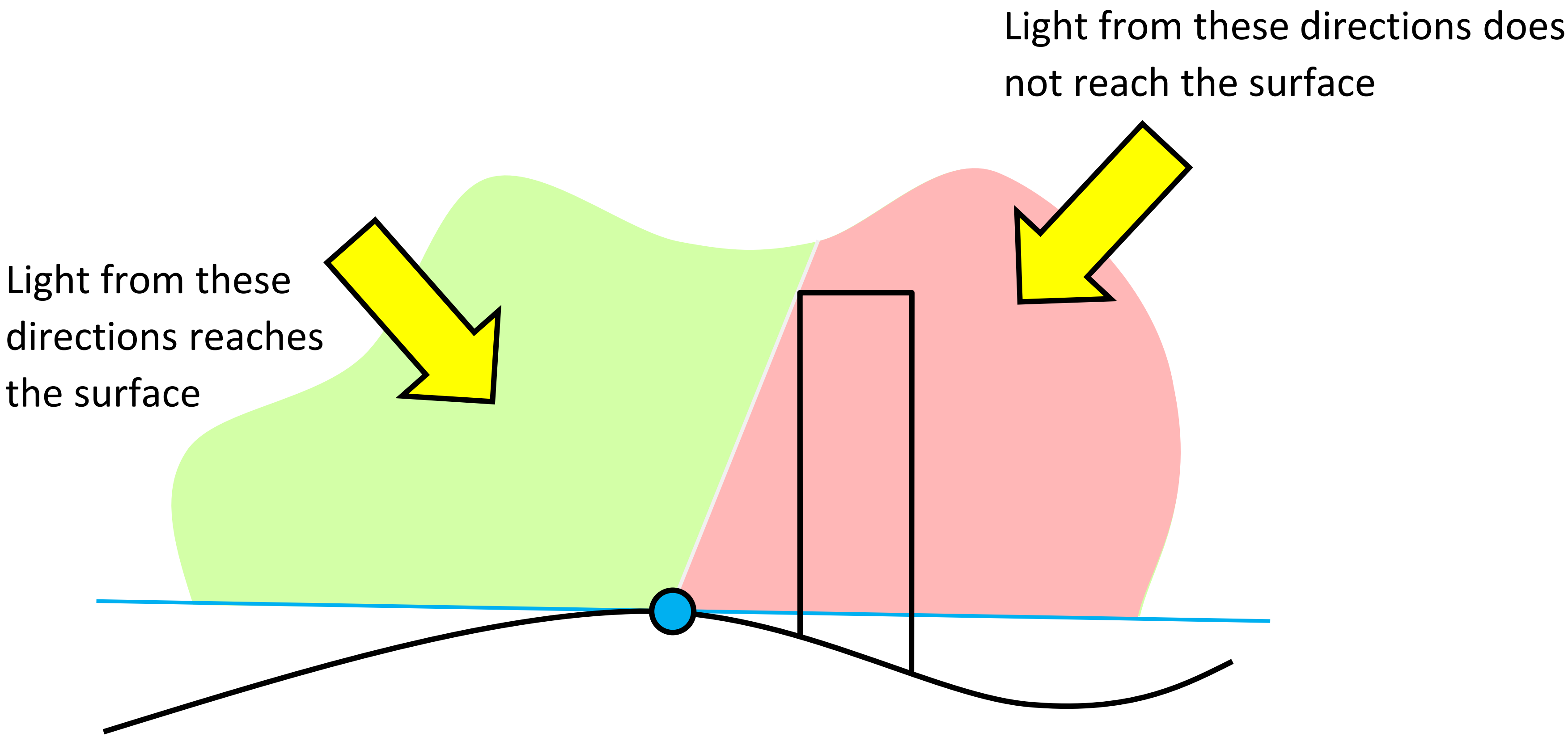
- Ambient occlusion (AO) approximates global illumination
  - Determining how exposed a point on a surface is to ambient lighting
    - Relatively cheap approximation of rendering equation
  - Can be easily combined with other shading calculations (e.g. Phong shading, shadows…)

Introduction. Concept

- Global illumination
  - Light reaching to your eye (or surface) from any part of the scene (world)
    - Especially interesting on non-directly lit surfaces

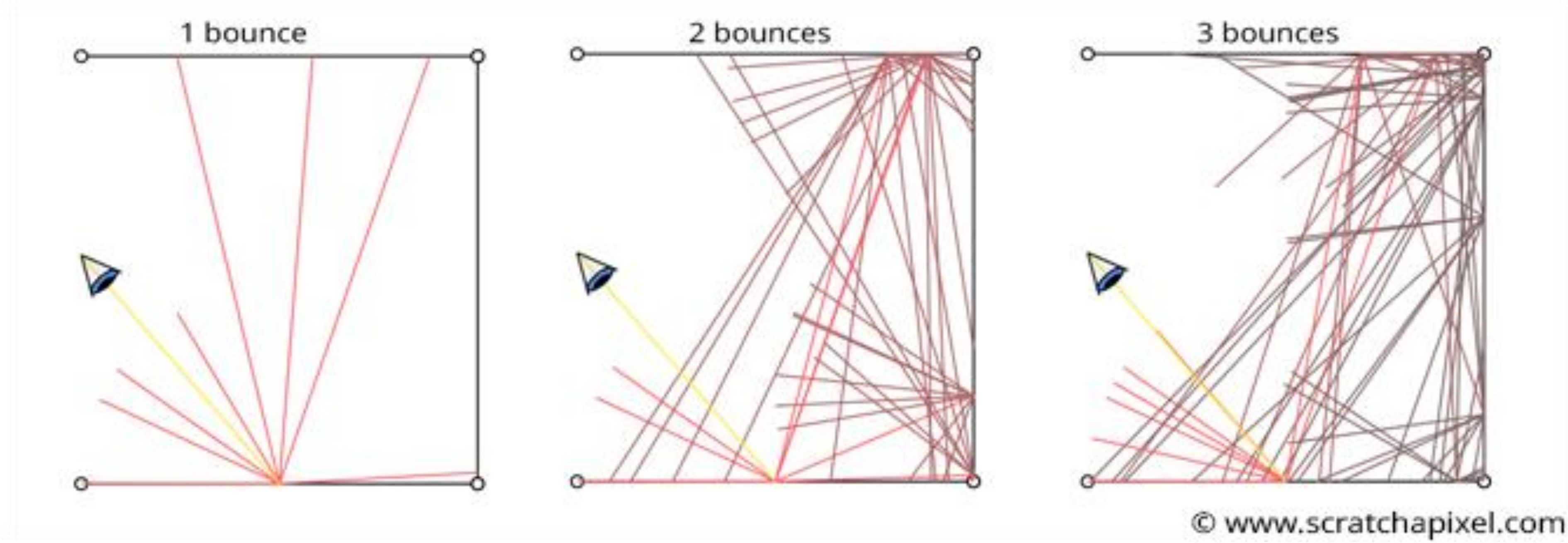
Introduction. Concept

- Occlusion of incoming ambient light



Introduction. Concept

- Global illumination
  - Result of multiple bounces of light across the objects in the scene



Introduction. Concept

- Ambient occlusion

Introduction. The Rendering Equation

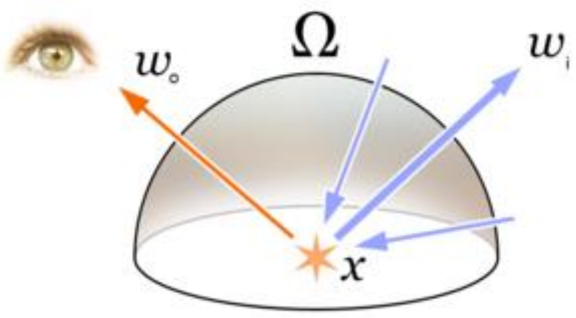
- The interaction between light and objects is modelled by what is known as the rendering equation.
  - Describes how light reflects off a surface

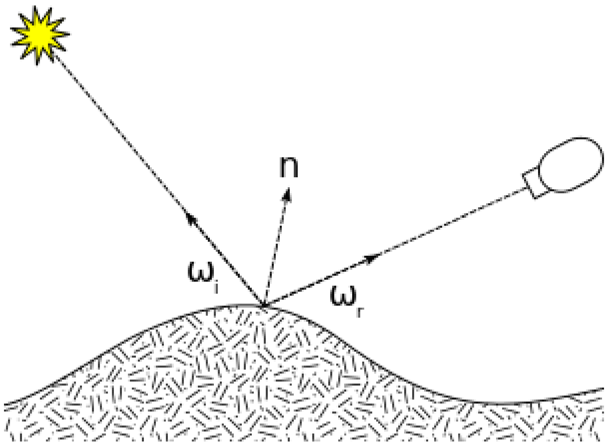
$$\underbrace{L_r(x,\omega_r)}_{\text{Light reflected towards eye}} = \int_{H^2} \underbrace{f_r(x,\omega_i \to \omega_r)}_{\text{Surface reflectance}} \underbrace{L_i(x,\omega_i)}_{\text{Incoming light}} \underbrace{\cos\theta_i'}_{\text{Angle weighting}} \, d\omega_i$$

Introduction. The Rendering Equation

- The Rendering Equation

$$L_o(p,\omega_o) = \int_\Omega f(p,\omega_o,\omega_i)\, L_i(p,\omega_i)\, \cos\theta_i \, \mathrm{d}\omega_i$$





Introduction. The Rendering Equation

- Every beam of light incident at *p* in direction ω_i
  - Multiplied by the BRDF of the material
  - And added up
    - Yields the reflected light at *p* in direction ω_0

$$L_o(p, \omega_o) = \int_\Omega f(p, \omega_o, \omega_i)\, L_i(p, \omega_i)\, \cos\theta_i \,\mathrm{d}\omega_i$$

Introduction. The Rendering Equation

- Every beam of light incident at *p* in direction ω_i
  - May come from many different points in space
    - Light (which is not absorbed) continuously bounces off the surfaces in scene

- L_i is expressed as a function of L_0
  - Recursive formulation!!!

$$L_o(p,\omega_o) = \int_\Omega f(p,\omega_o,\omega_i)\, L_i(p,\omega_i)\, \cos\theta_i \,\mathrm{d}\omega_i$$

Introduction. The Rendering Equation

- L_i is expressed as a function of L_0
  - Recursive formulation!!!

$$L_o(p, \omega_o) = \int_\Omega f(p, \omega_o, \omega_i)\, L_i(p, \omega_i)\, \cos\theta_i \,\mathrm{d}\omega_i$$

- Means we need to apply recursivity for every ray, every bounce…

Outline

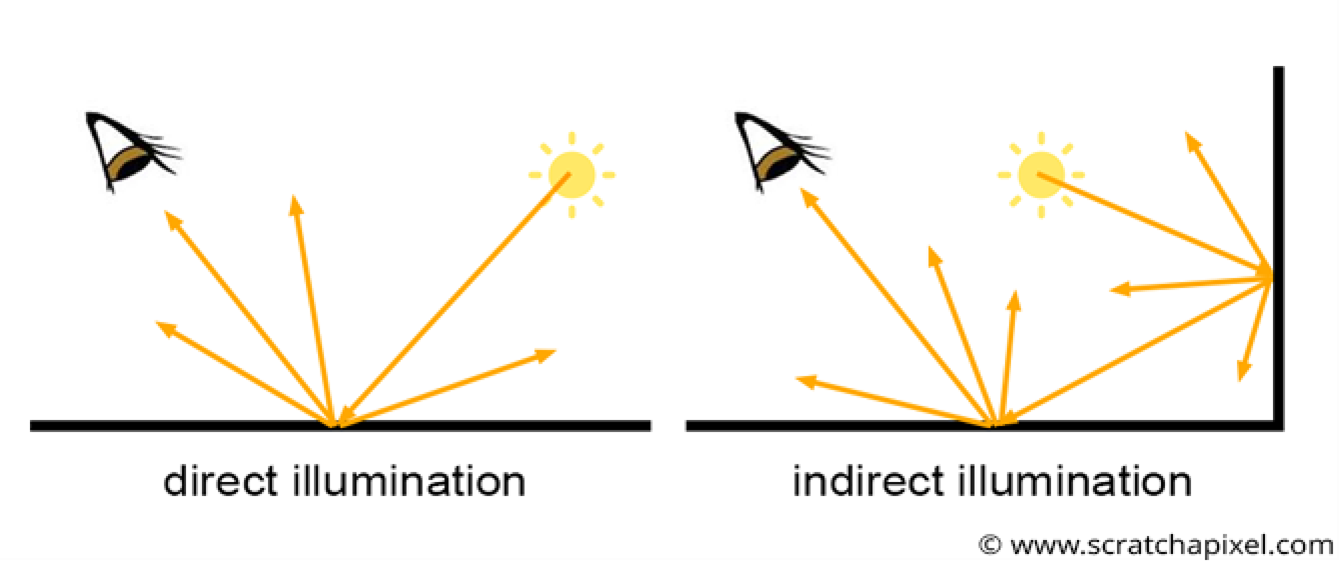
- *Introduction*
- **Derivation**
- Algorithms
- Practical issues
- Implementation in mobile

Derivation

- Many CG algorithms approximate the Rendering Equation
  - To accelerate its calculation
  - Otherwise it would be too costly

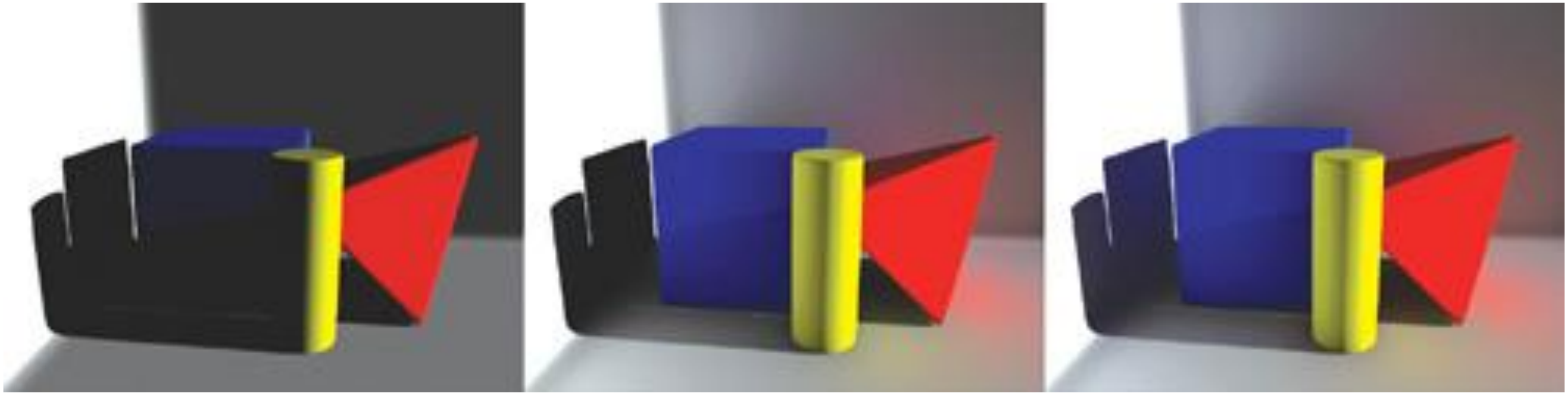
Derivation

- Direct lighting:
  - The simplest approach
  - Take into account a single bounce



Derivation

- 1 bounce vs 2 bounces vs 3 bounces
  - The larger the number of bounces the better the approximation

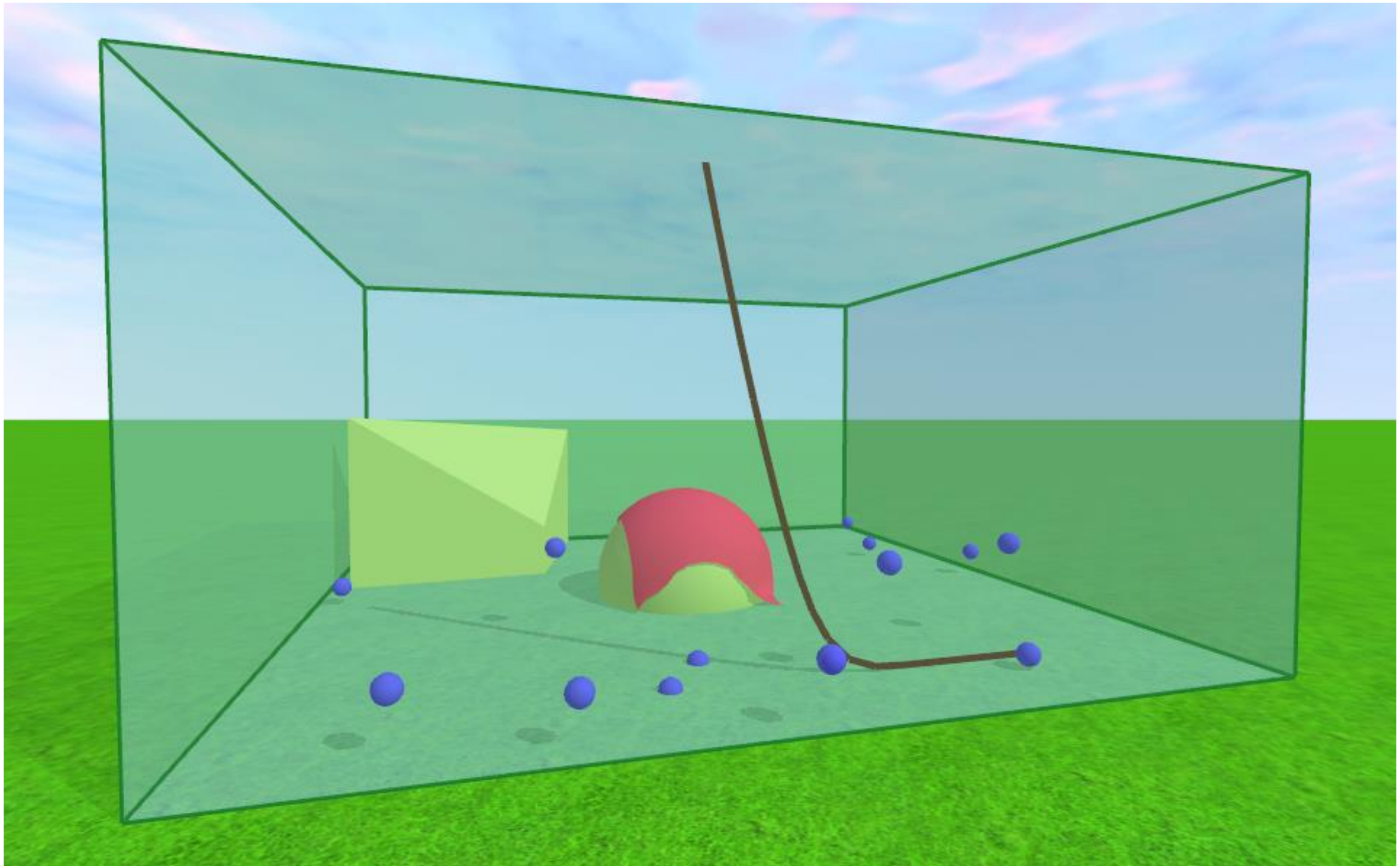


Derivation

- Ambient light
  - Replace indirect lighting with a constant
  - We assume that light reaches every point in every direction and with equal intensity
    - Ignoring the point's surrounding objects
- Traditional and computationally-efficient
- Rough approximation

Derivation

Ambient light



Derivation

- Ambient Occlusion
  - Indirect lighting approximation halfway between ambient light and true indirect illumination
  - Instead of using a constant ambient term, ambient occlusion produces an ambient term for every point in the scene

Derivation

- Ambient Occlusion makes two assumptions
  - The surface is a perfect diffuse surface
    - The BRDF becomes a constant

$$L_o(p, \omega_o) = k \int_\Omega L_i(p, \omega_i) \cos\theta_i \, \mathrm{d}\omega_i$$

Derivation

- Ambient Occlusion makes two assumptions (ii)
  - Light potentially reaches a point *p* equally in all directions
    - But takes into account point's visibility

$$L_o(p, \omega_o) = k \int_{\Omega} V(p, \omega_i) \cos\theta_i \, \mathrm{d}\omega_i$$

$$V(p, \omega_i) = \begin{cases} 0 & p \text{ occluded in direction } \omega_i \\ 1 & \text{otherwise} \end{cases}$$

Derivation

- Ambient Occlusion
  - Accounting for all visible directions
    - Upper part of the surface

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

Derivation

- Ambient Occlusion. Problem:
  - Indoor scenes would yield pitch black AO
    - Not the same occlusion comes from all the objects
- Solution:
  - Assume the closer objects occlude at a higher rate
    - Ambient Obscurance

Derivation

- Ambient Obscurance
  - Applies a fall-off function to the distance of intersection

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

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

Derivation

- Ambient Obscurance
  - Fall-off function serves three purposes
    - Makes AO work for closed scenes
    - Cuts the occlusion to zero for far away points
      - AO relatively local computation
    - Gives nearby occluders greater occlusion value

- Ambient Occlusion and Ambient Obscurance commonly synonyms in literature

Outline

- *Introduction*
- *Derivation*
- **Algorithms**
- Practical issues
- Implementation in mobile

Algorithms

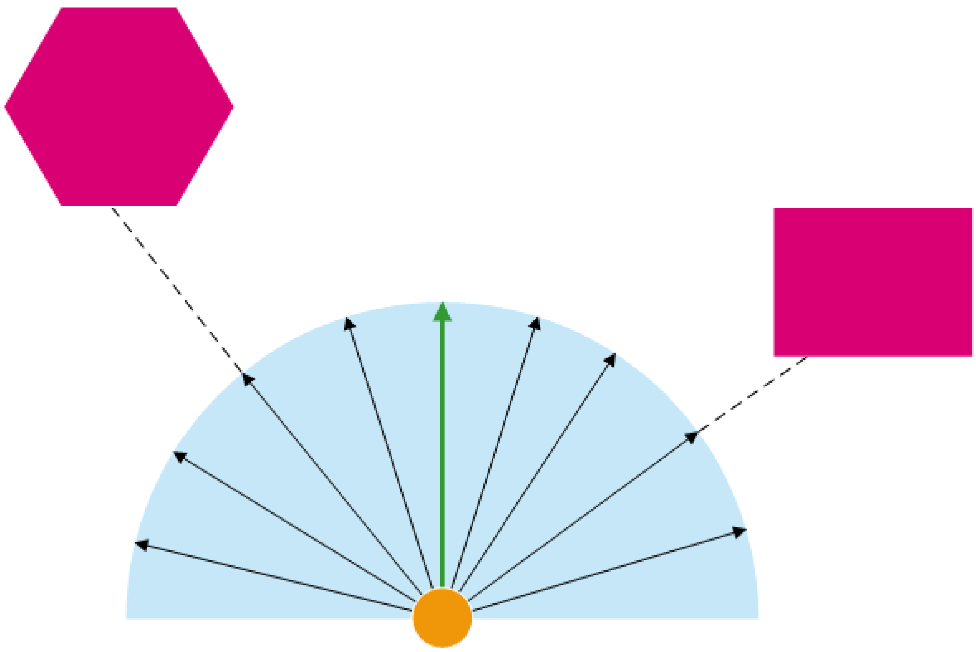
- Four main approaches
  - Ray-based
  - Geometry-based
  - Volume-based
  - Screen-space

Algorithms

- Four main approaches
  - **Ray-based**
  - Geometry-based
  - Volume-based
  - Screen-space

Algorithms. Ray-based

- Trace rays from the point to shade against the scene

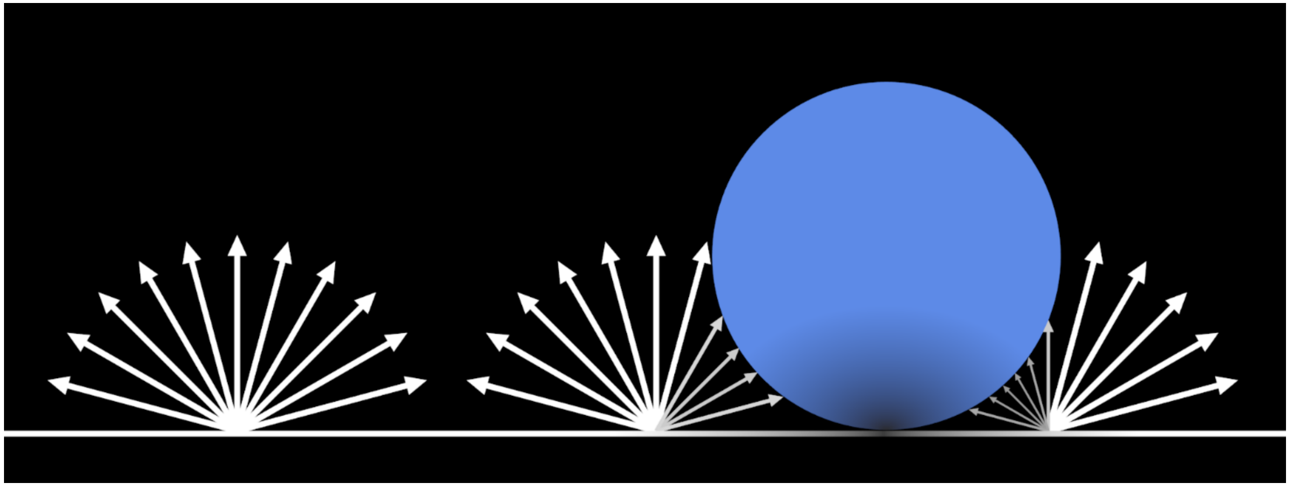


Algorithms. Ray-based

- In practice
  - Use limited range for occlusion
    - Otherwise everything would be occluded in indoor scenes
    - Also faster to calculate because of finite radius
    - Use falloff function to smooth the transition
  - Do not solve analytically
    - Theoretically doable, but would be ridiculously expensive
    - Solution: Monte Carlo sampling
    - 256 – 1024 samples is usually enough

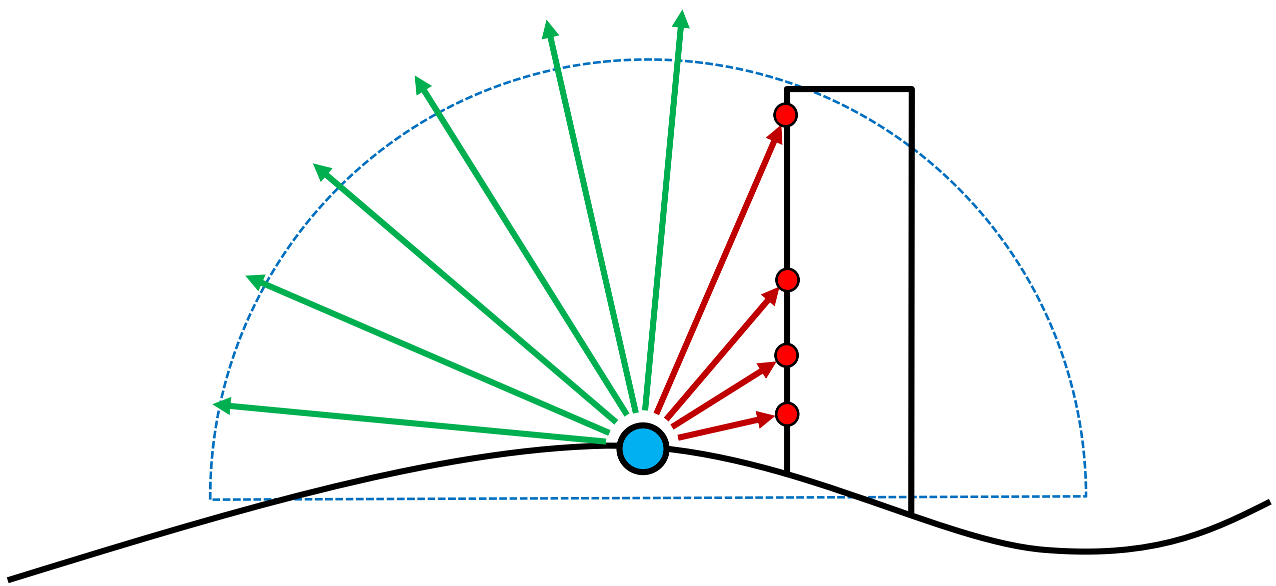
Algorithms. Ray-based

- Ray-based
  - Trace rays from the point to shade against the scene



Algorithms. Ray-based

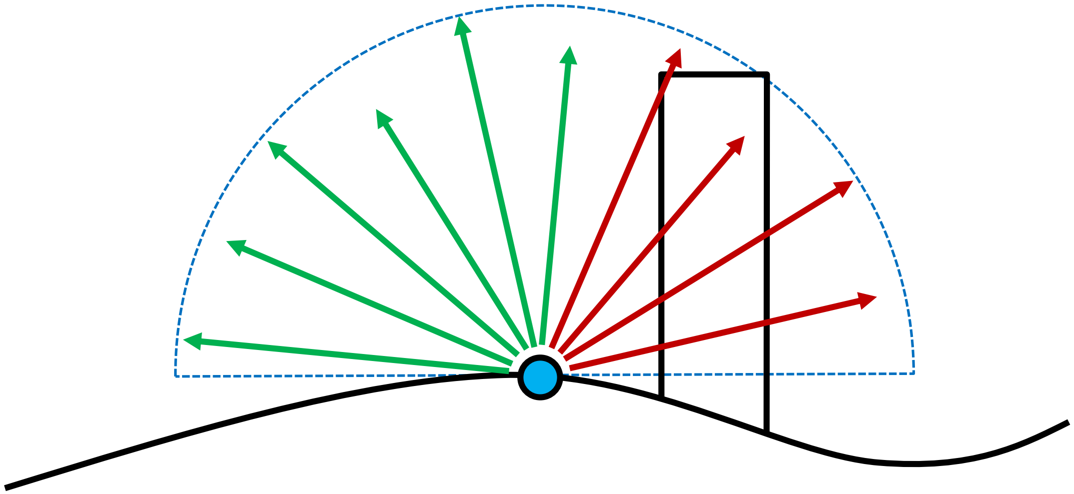
- Cast a number of rays from the point to be shaded
- Determine occlusion distance, apply falloff, sum



- In reality use a fancy low-discrepancy sampling pattern

Algorithms. Ray-based

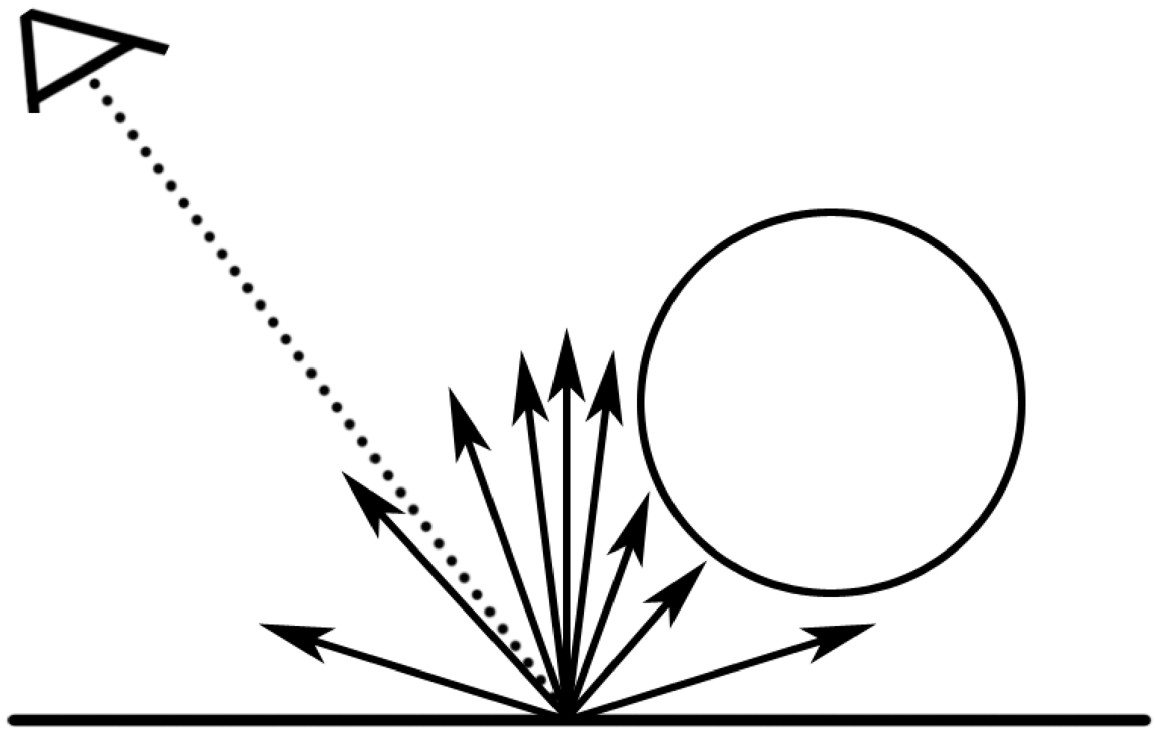
- Shadow rays
  - Usually faster than ordinary rays
    - Enough to detect any intersection, not necessarily closest one
  - Can do if we bake falloff function into the rays



Algorithms. Ray-based

- The amount of illumination a surface point is likely to receive in a scene, is calculated.
- A ray is sent from the camera to a surface point.
- This ray is split up into multiple rays that shoot off, into the scene, in a hemispherical fashion.
- These rays are biased to the normal of the surface.

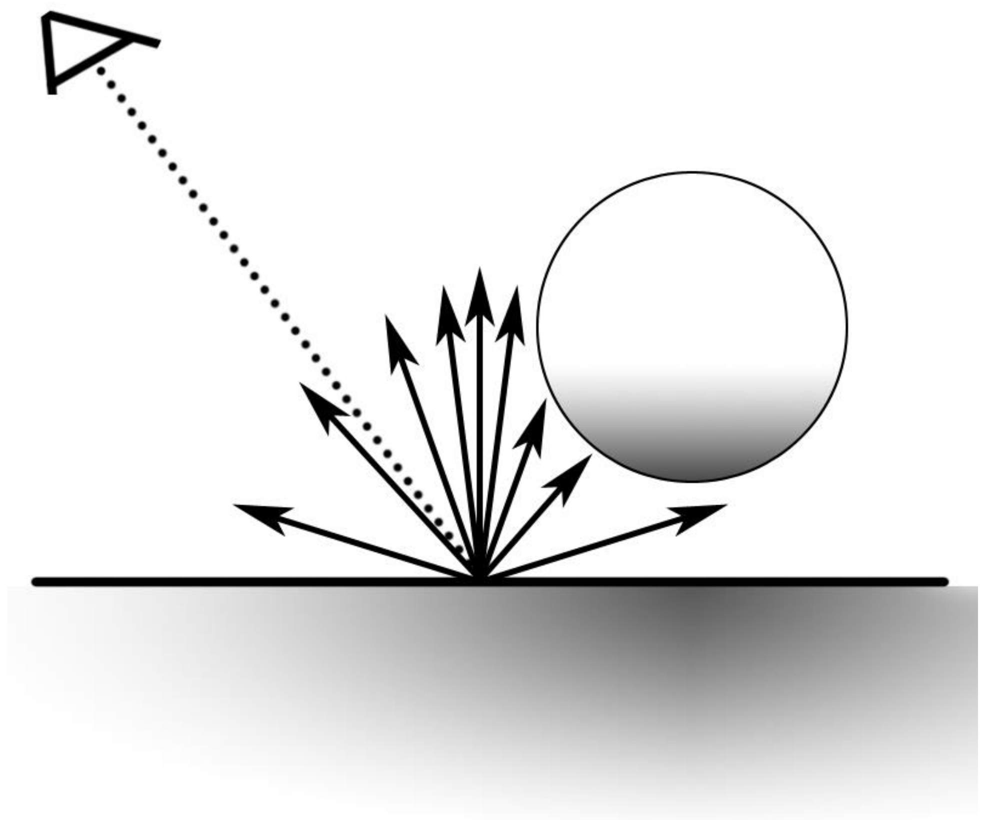
Algorithms. Ray-based



Algorithms. Ray-based

- A calculation of the ratio of intersections and the number of cast rays is made, to produce the final illumination value.
- The less intersections we have, the brighter the point is.
- This is repeated again for the next surface point.

Algorithms. Ray-based

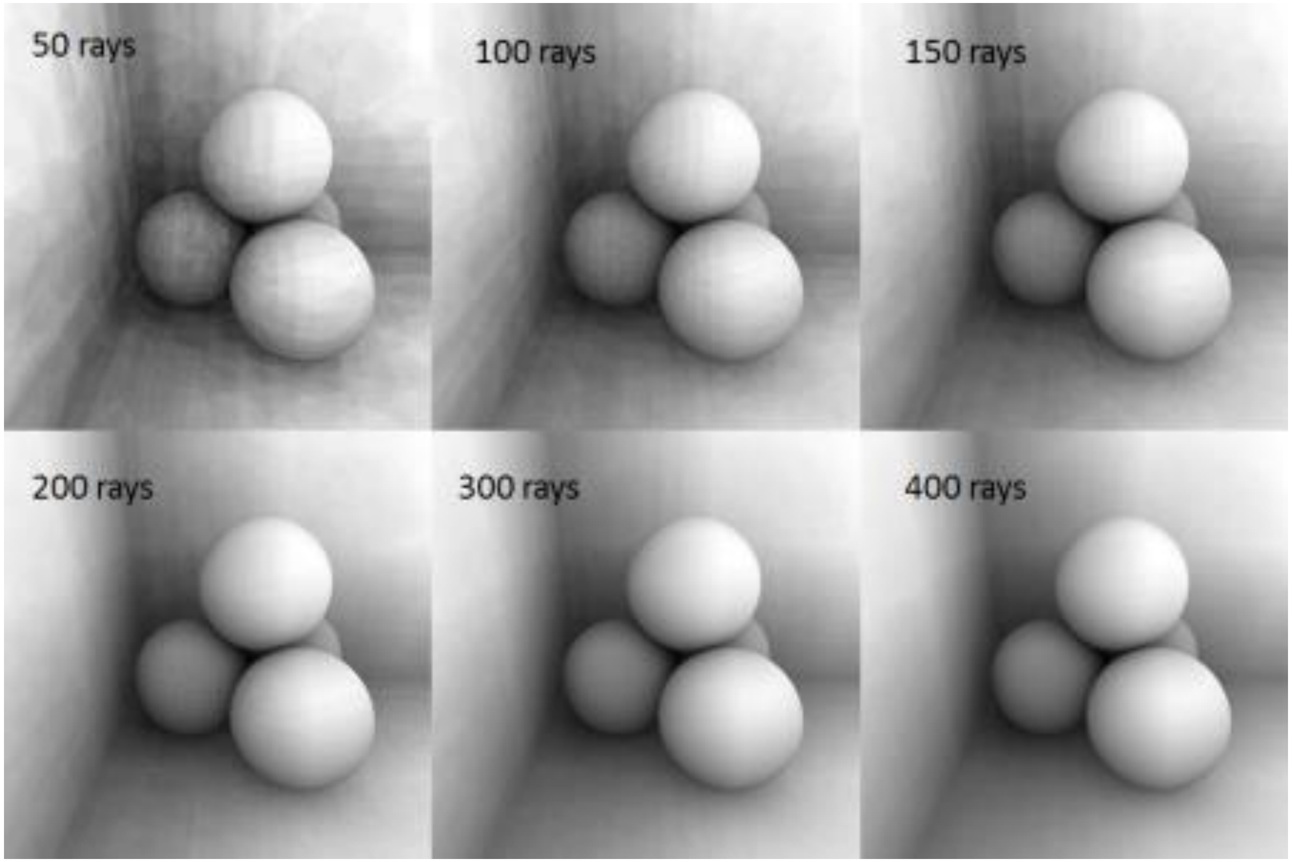


Algorithms. Ray-based

- Ray-based
    - High quality
    - Costly
    - Ray patterns
    - Sampling

Algorithms. Ray-based

- Ray-based



Algorithms

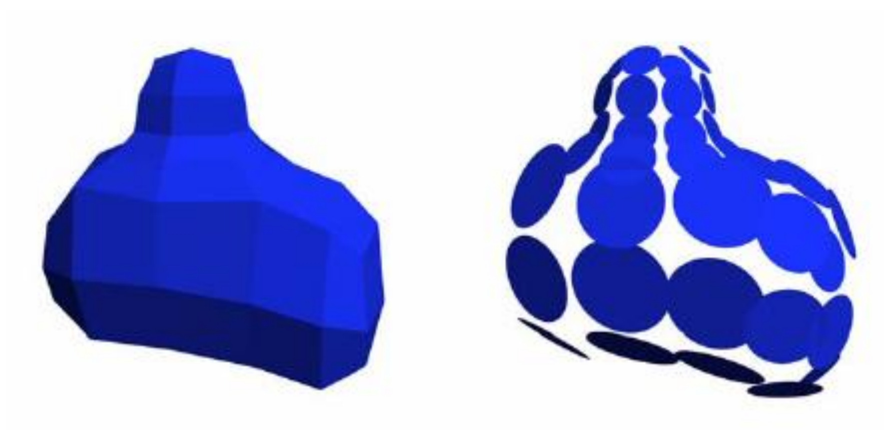
- Four main approaches
  - *Ray-based*
  - **Geometry-based**
  - Volume-based
  - Screen-space

Algorithms. Geometry-based

- RC is costly
  - Offline
- Simplify the query geometry
  - 2 layers: close geometry
  - Simplify objects

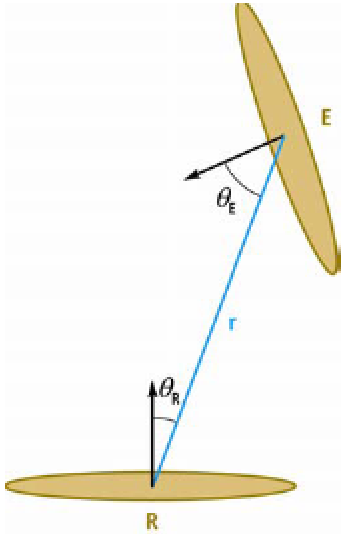
Algorithms. Geometry-based

- Simplify Geometry: Use polygon meshes as disk-shaped elements [Bunnell, 2005]
  - One element per vertex
  - Elements defined by position, normal, and area
  - Simplifies form factor calculation



Algorithms. Geometry-based

- Percentage of the hemisphere above a point occluded by an element (Solid Angle)
- Like radiosity form factor with 100% visibility



Emitter element E occludes receiver element R based on distance r and angles θ_E and θ_R

Algorithms. Geometry-based

- Calculate occlusion at a receiver by summing form factors:

  Occlusion = 0

  For each element E

      Occlusion += form factor of E;

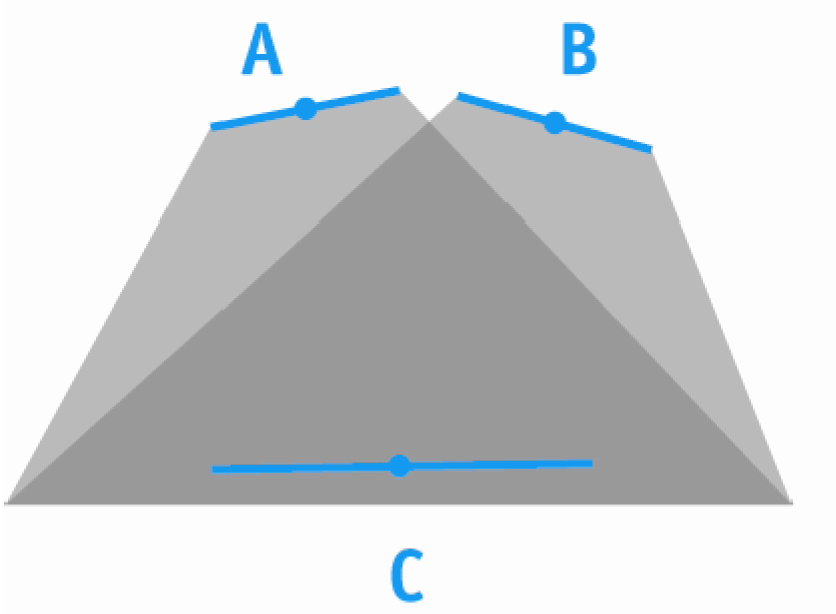
Algorithms. Geometry-based

- May simplify geometry further:

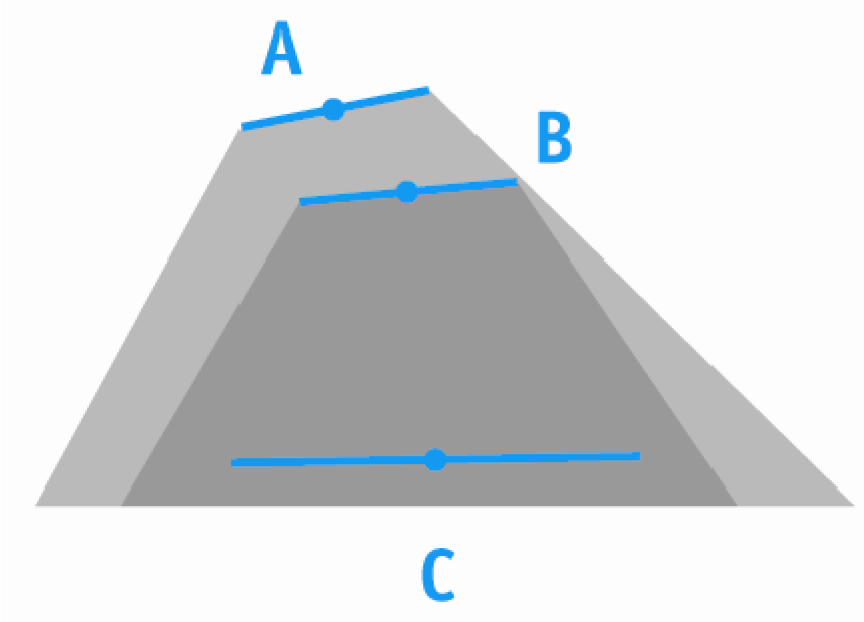
Algorithms. Geometry-based

- Must avoid double shadowing:

Ok

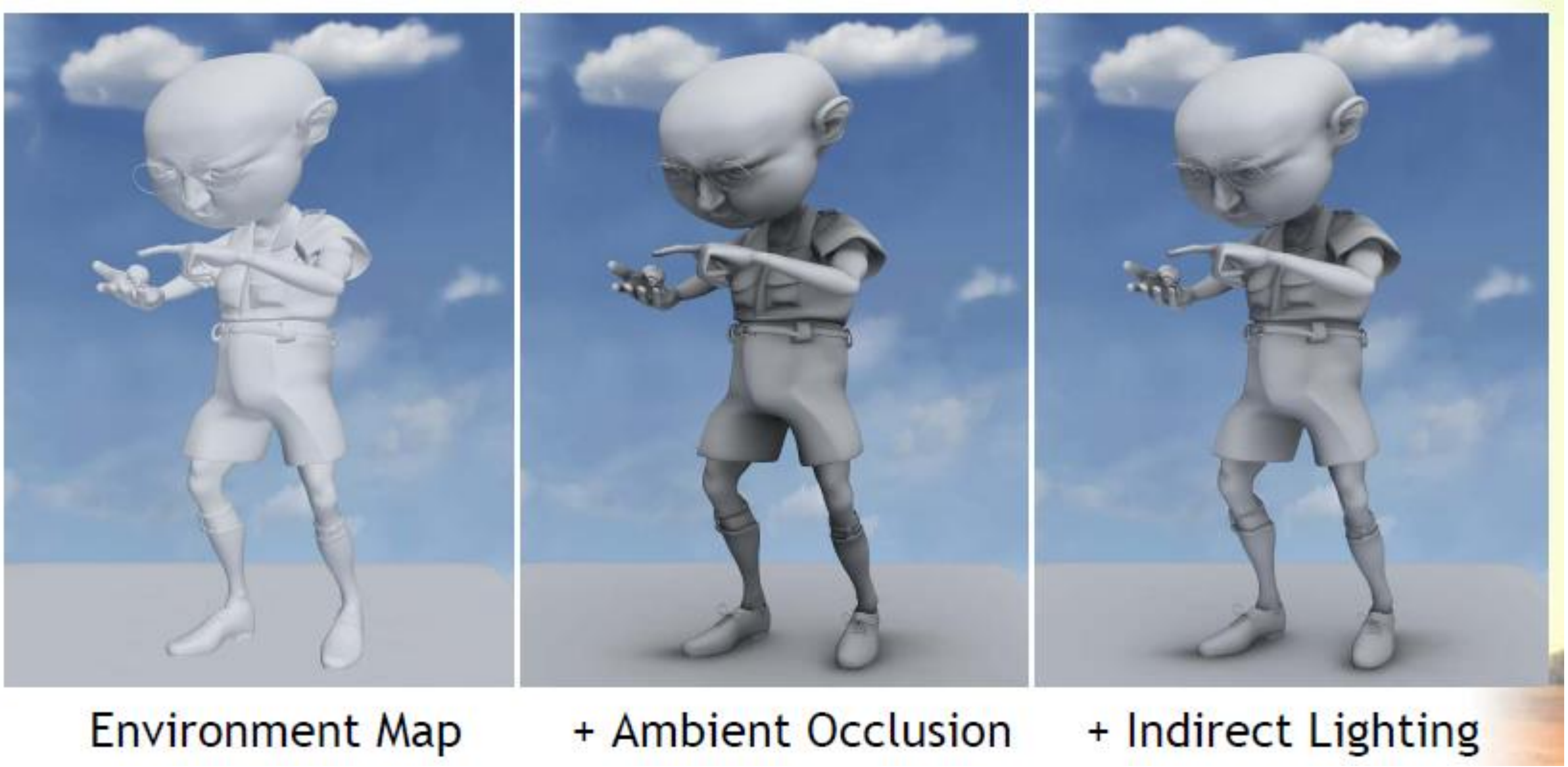


No Ok

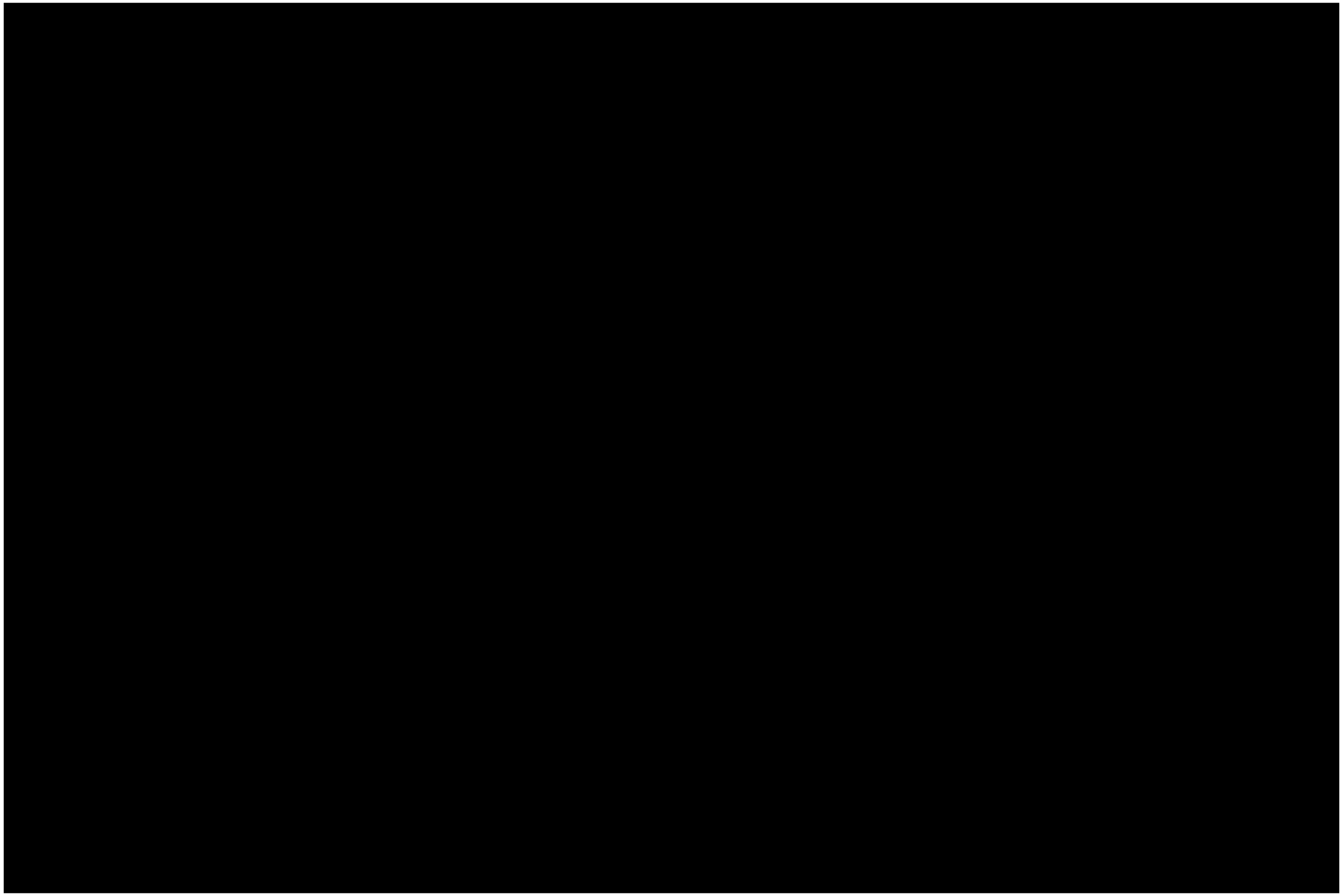


Algorithms. Geometry-based

- Results



Algorithms. Geometry-based

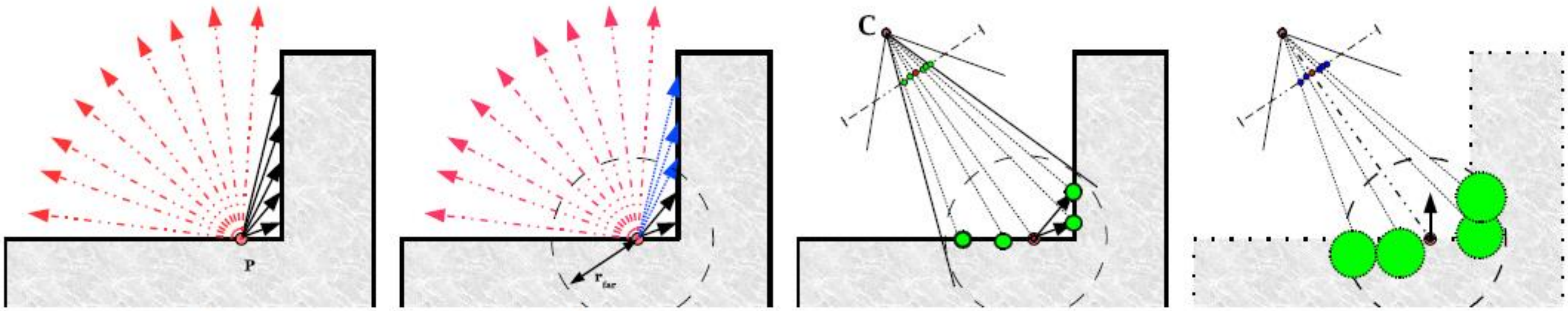


Algorithms. Geometry-based

- Other implementations:
  - Some proxy of geometry
  - Use multiple passes with shadow maps

Algorithms. Geometry-based

- Main idea: Approximating the geometry with spheres [Shanmugam and Arikan, 2007]:
  - Ray intersection is easy
  - Can be tricky to represent planar shapes



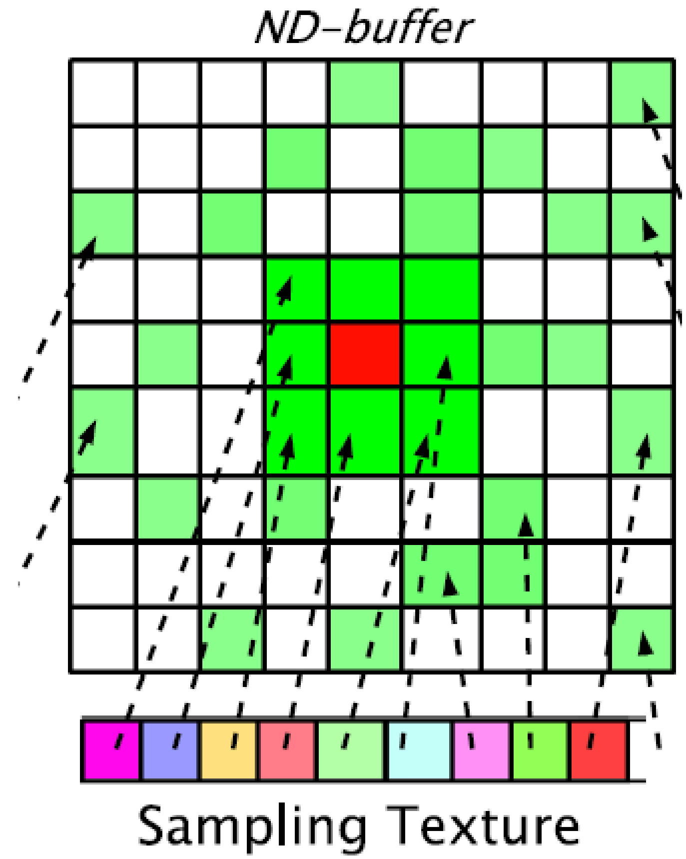
(a) (b) (c) (d)

Algorithms. Geometry-based

- Two-levels:
  - High-frequency shadows – closer geometry
    - Querying the Normal-Depth (ND) map
  - Low-frequency shadows – distant objects
    - Sphere approximation with ray-length limitation

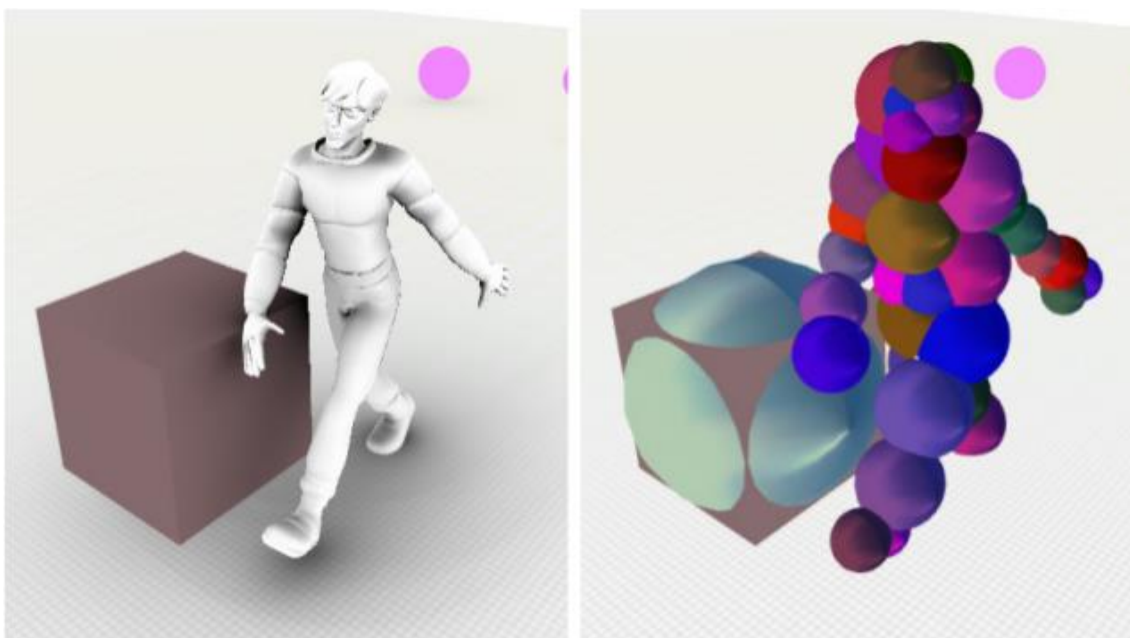
Algorithms. Geometry-based

- High-frequency shadows



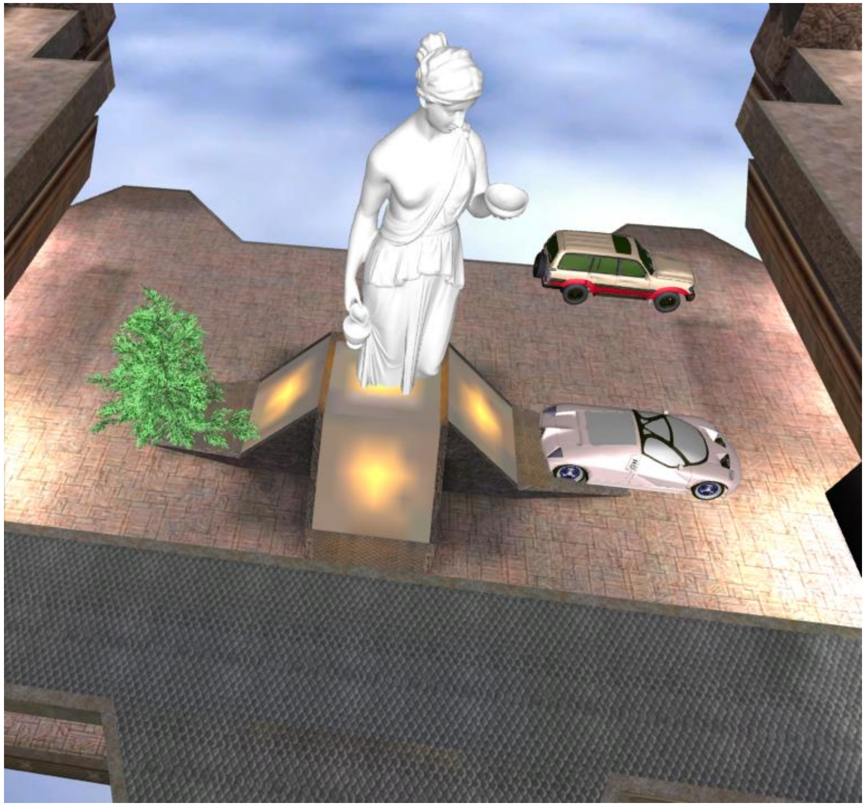
Algorithms. Geometry-based

- Low-frequency shadows:
  - Approximating the geometry with spheres:



Algorithms. Geometry-based

- Geometry-based



Algorithms

- Four main approaches
  - *Ray-based*
  - *Geometry-based*
  - **Volume-based**
  - Screen-space

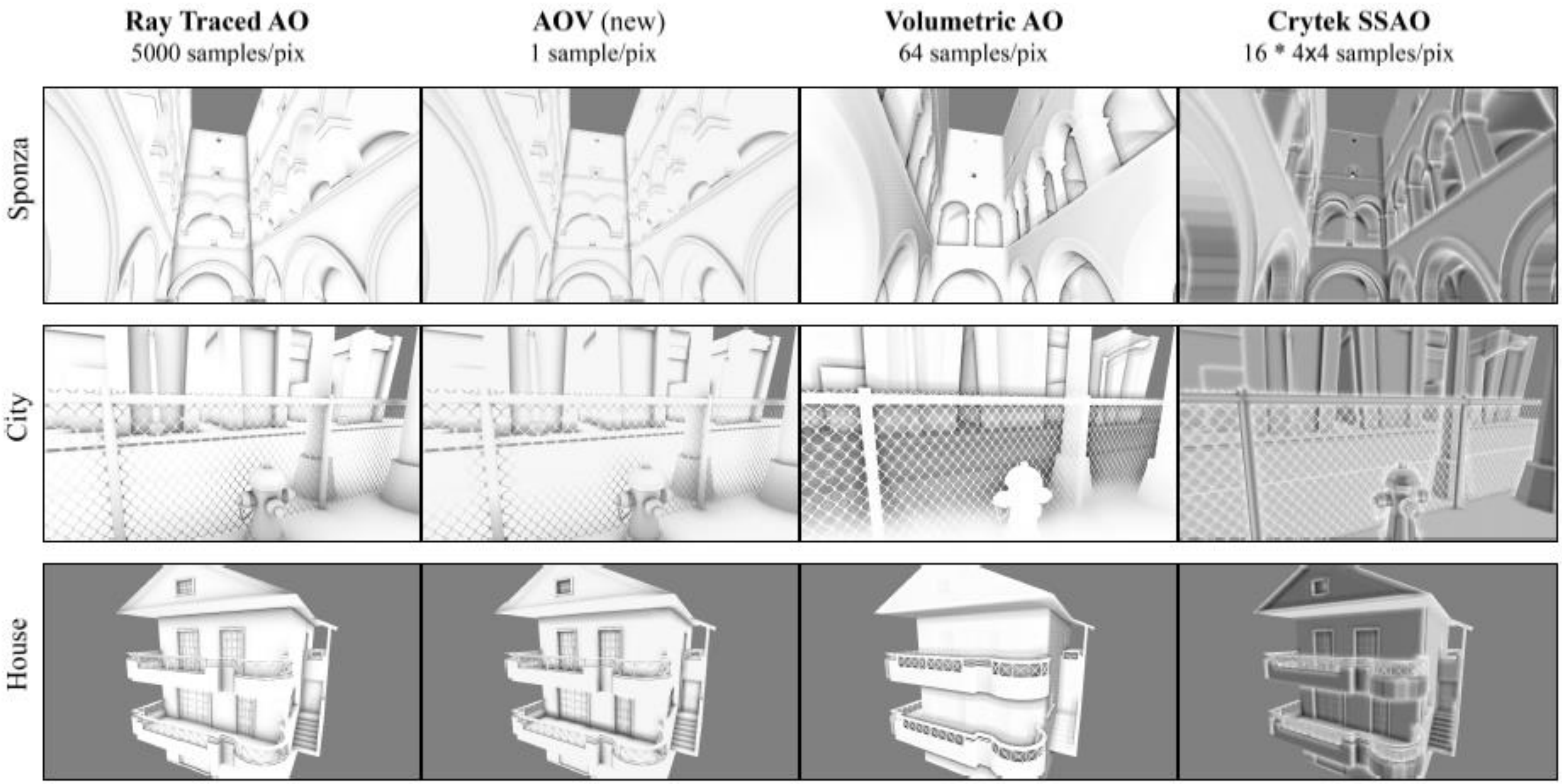
Algorithms. Volume-based

- Similar to geometry-based but
  - Scene is 3D
    - 3D texture encoding space occupancy
    - Voxels
    - Fragmented voxels

Algorithms. Volume-based

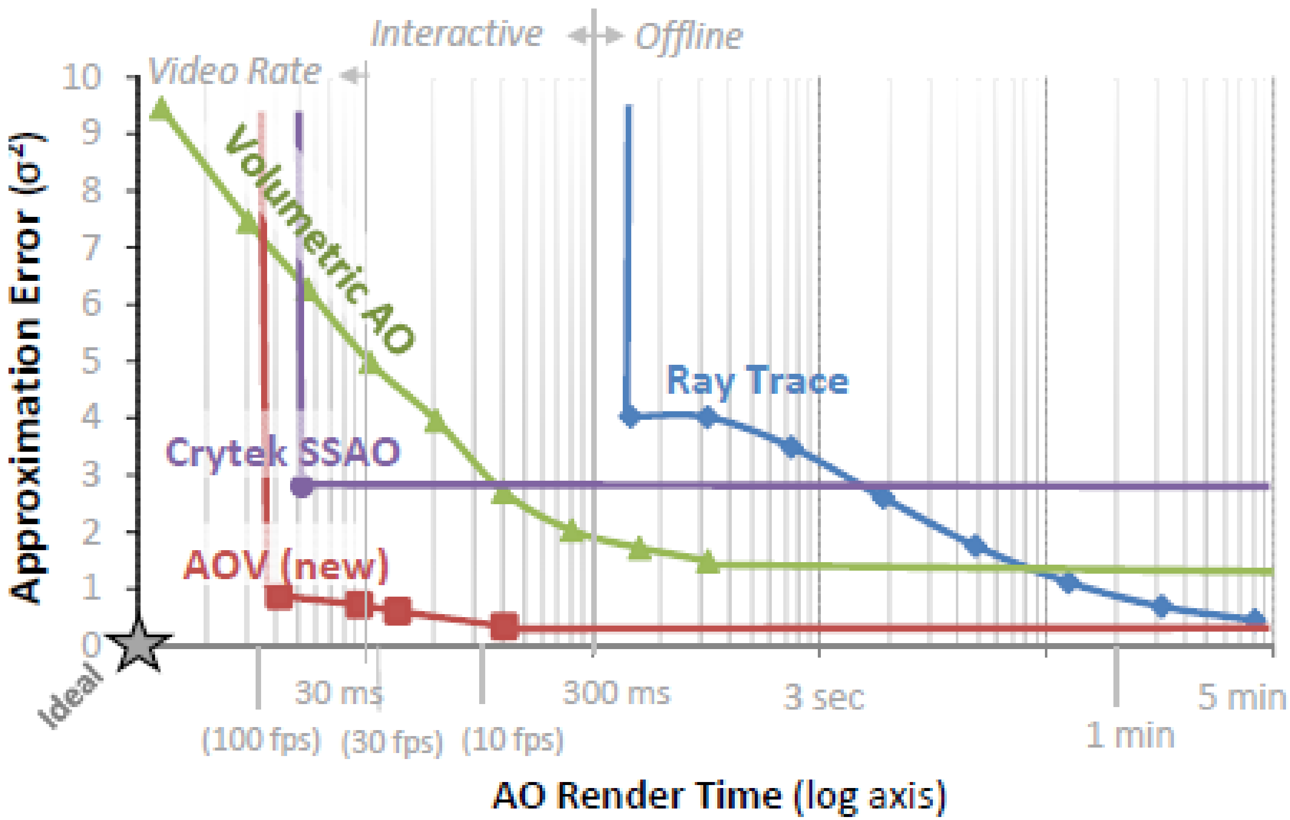
- Ambient Occlusion Volumes (McGuire 2010)
  - It computes the analog of a shadow volume for ambient light around each polygon
  - Applies a tunable occlusion function within the region it encloses

Algorithms. Volume-based



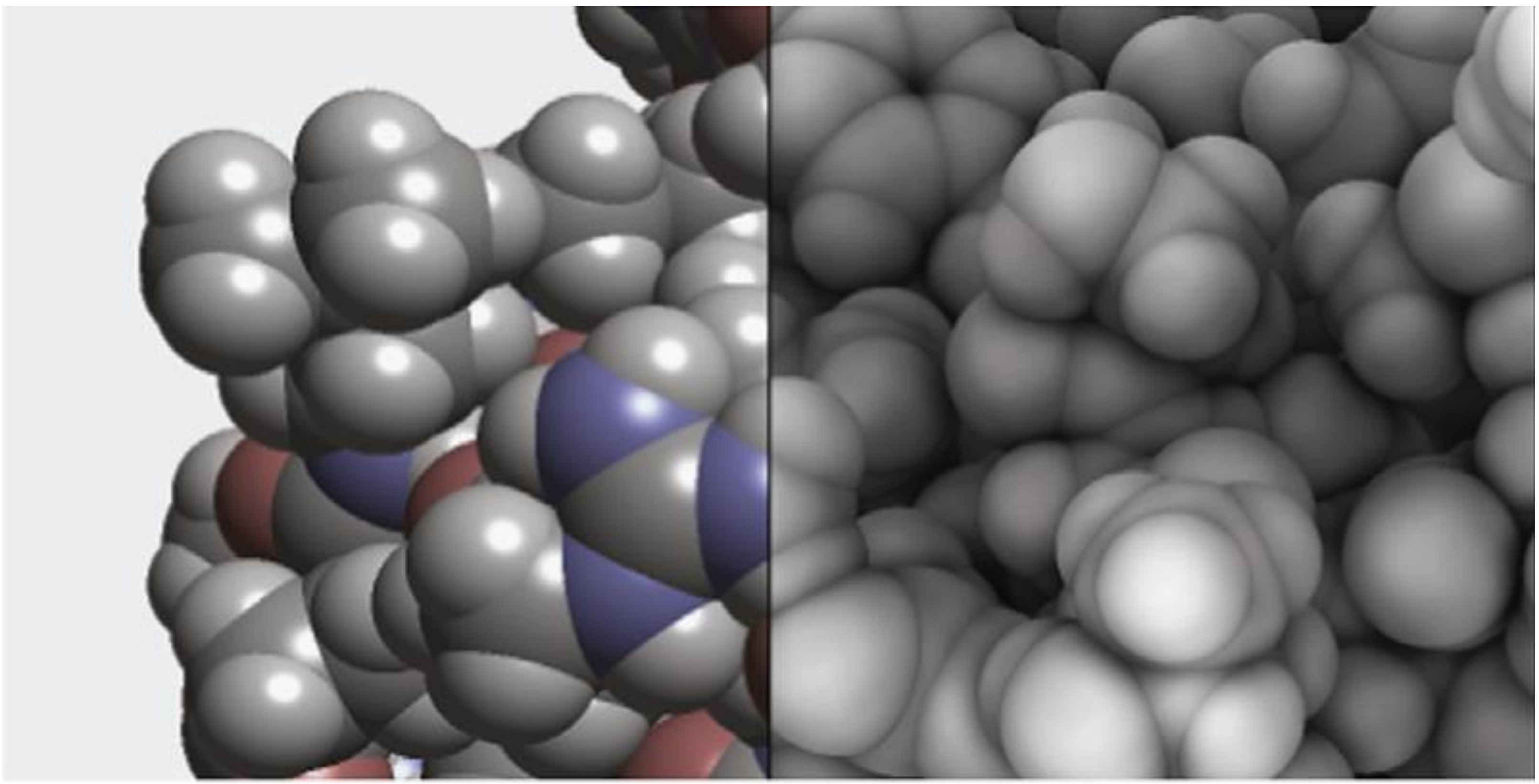
Algorithms. Volume-based

- Ambient Occlusion Volumes (McGuire 2010)

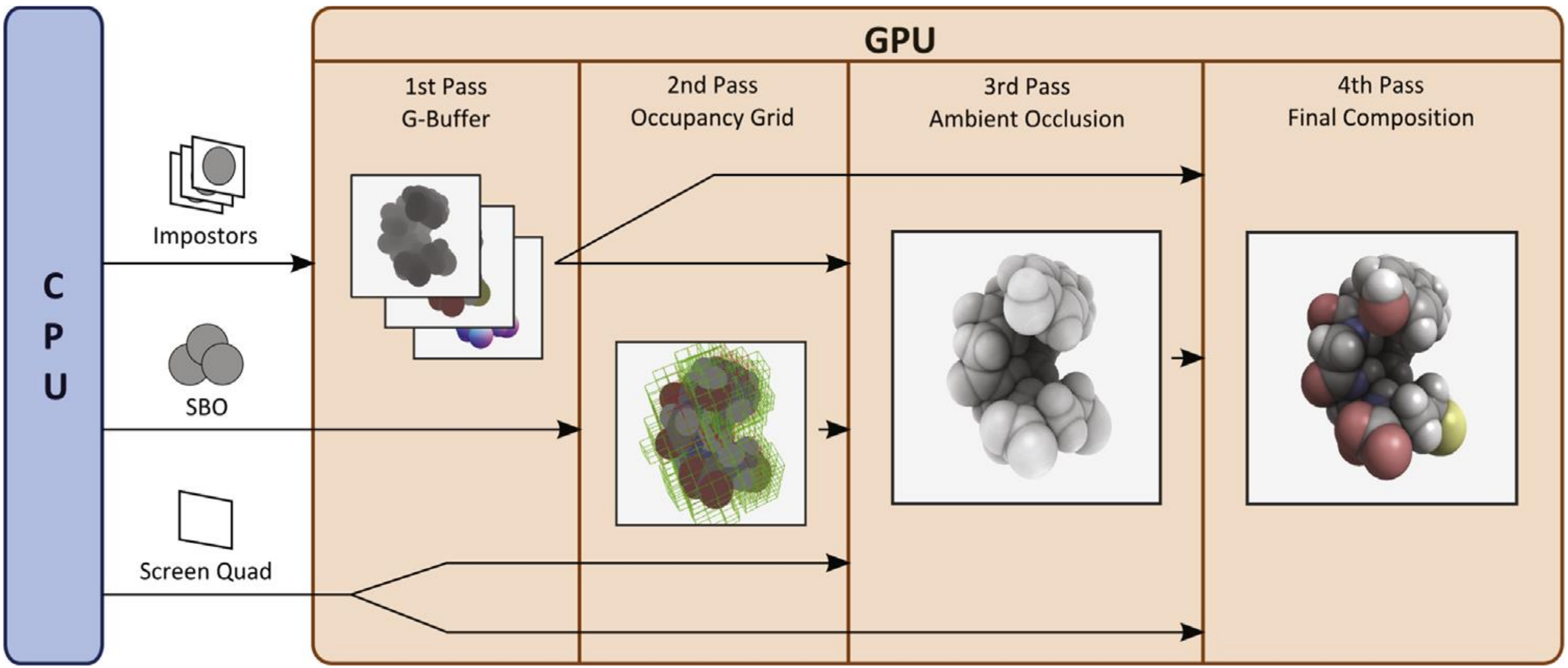


Algorithms. Volume-based

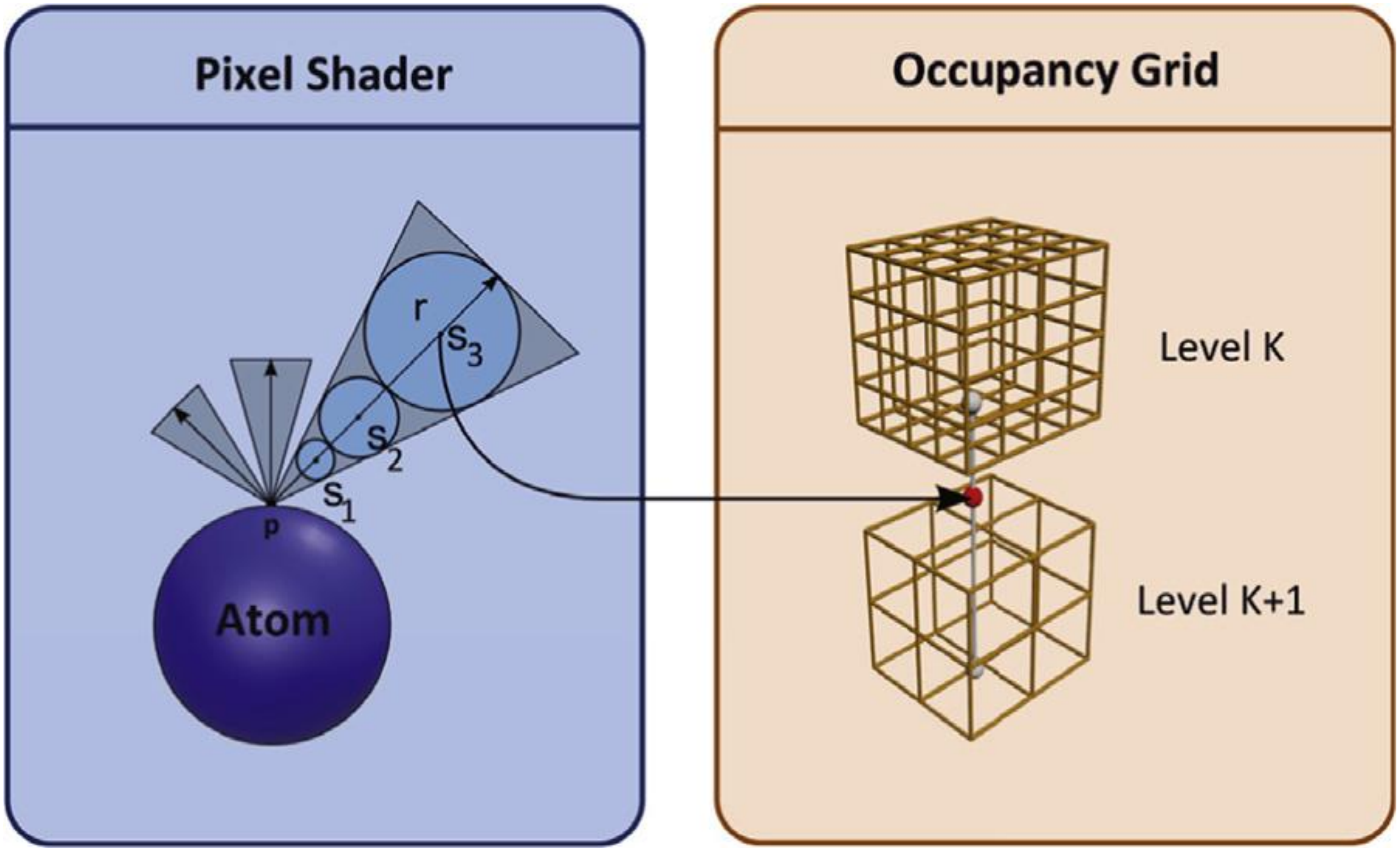
- AO for large molecular models [Hermosilla et al., 2015]



Algorithms. Volume-based

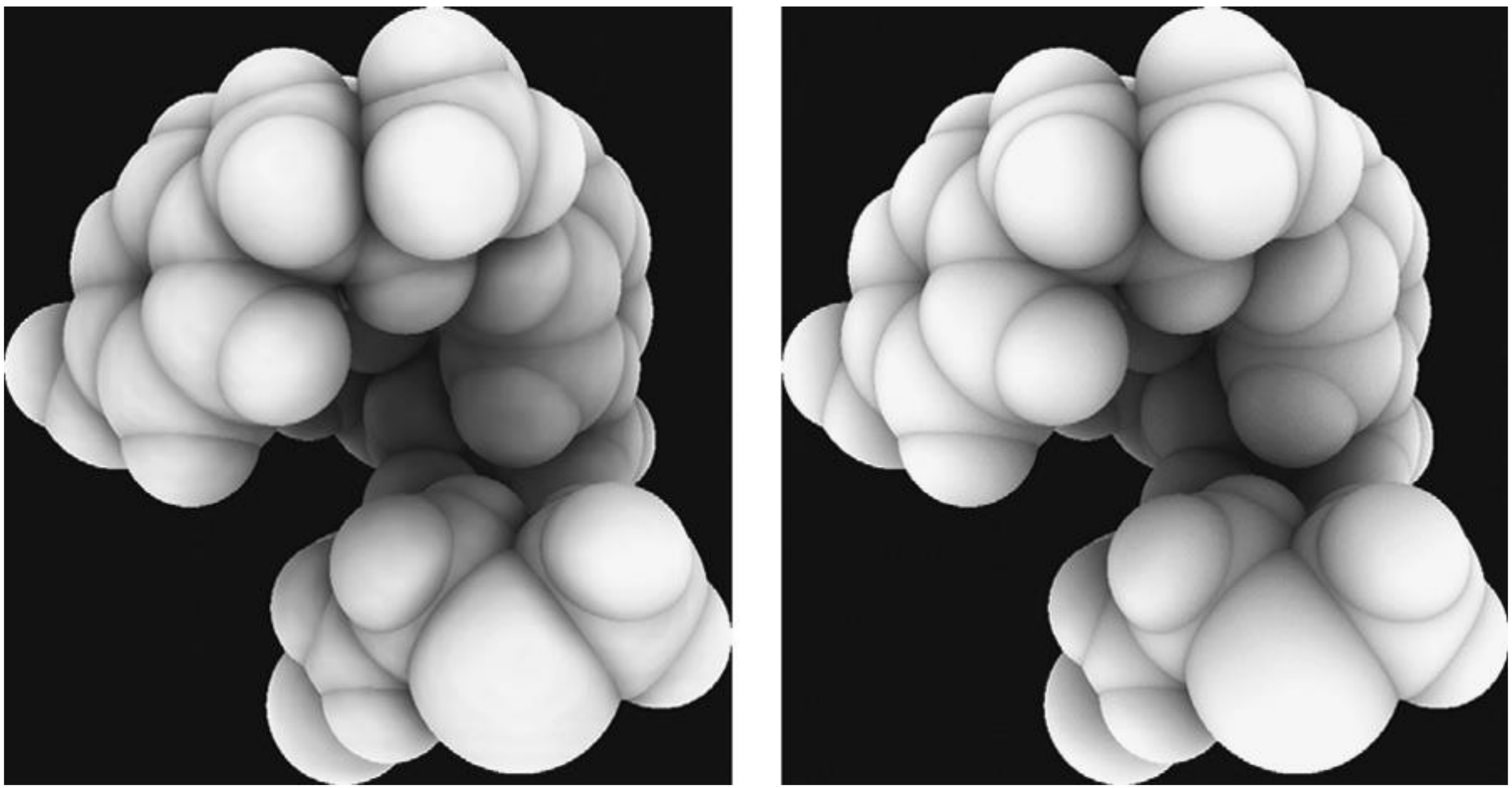


Algorithms. Volume-based



Algorithms. Volume-based

- AO vs ground truth



Algorithms. Volume-based

- 1.36M atoms

Algorithms

- Four main approaches
  - *Ray-based*
  - *Geometry-based*
  - *Volume-based*
  - **Screen-space**

Algorithms. Screen-space

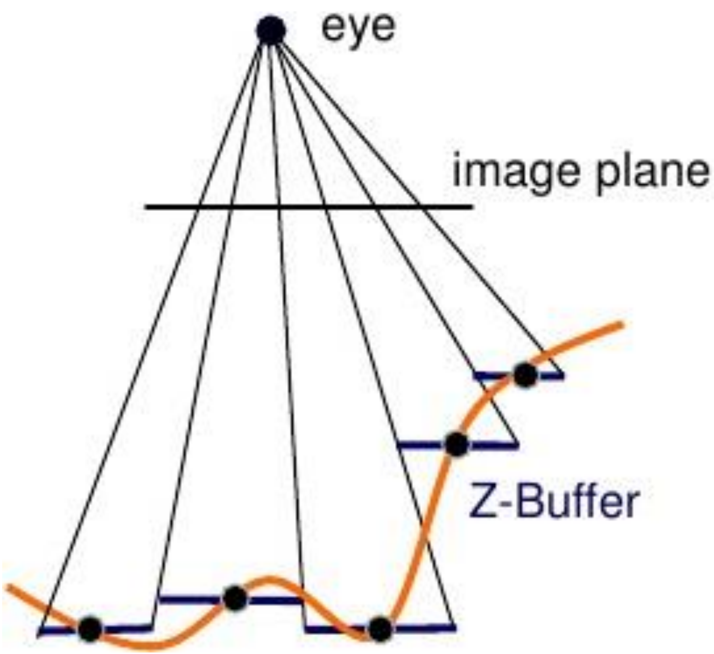
- Reduce calculation complexity
  - Use depth buffer to approximate scene geometry
  - Can also use Depth + Normal (aka ND buffer)
  - Calculation in screen-space $\rightarrow$ reduced complexity

Algorithms. Screen-space

- Uses depth buffer to approximate scene geometry

Algorithms. Screen-space

- Screen-space post-processing effect
  - Z-buffer data might be already available in a texture
    - Use it to estimate AO



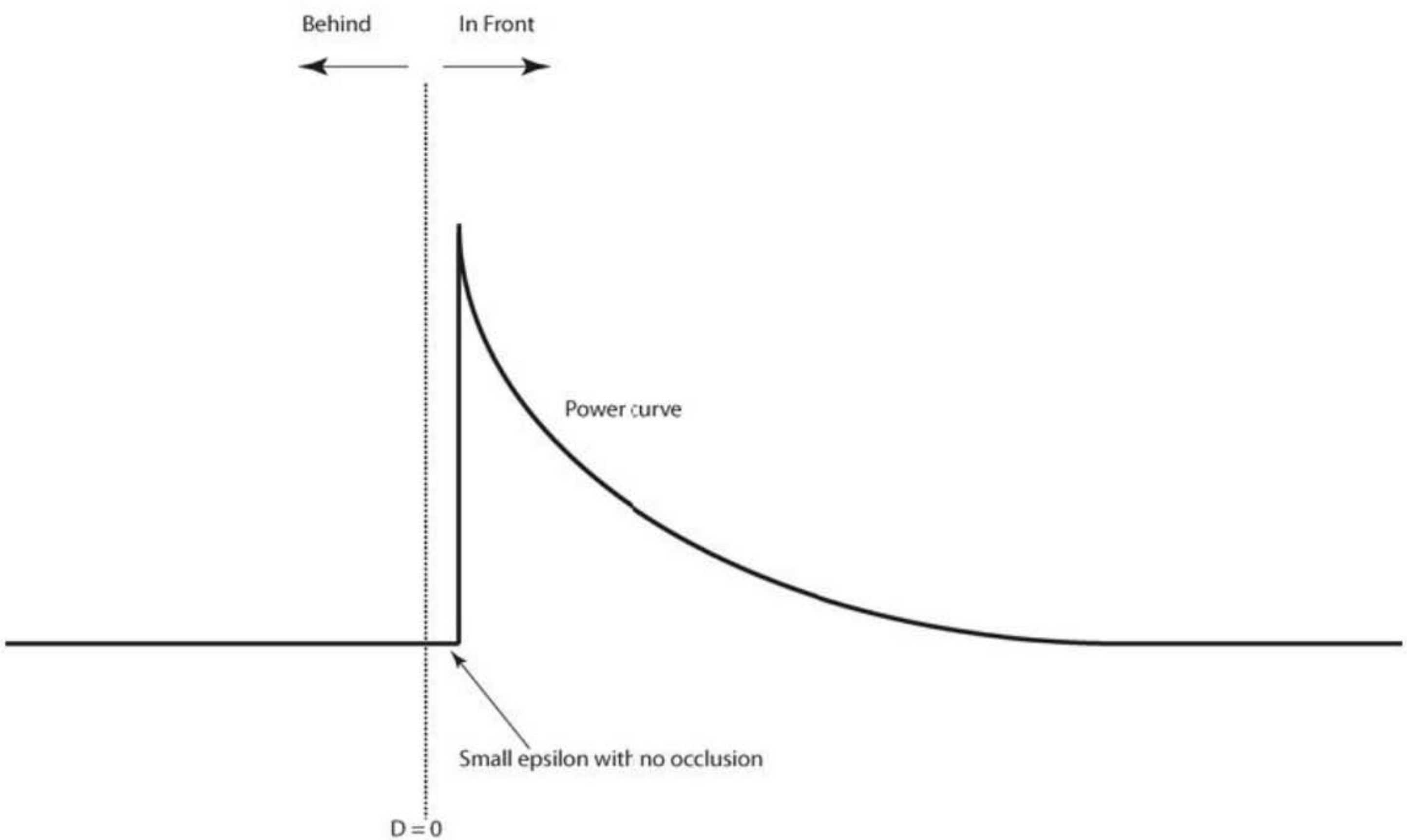
Algorithms. Screen-space

- AO estimation
  - Sample neighbouring pixels
  - The calculated term is used to attenuate incoming light

Algorithms. Screen-space

- Use random samples inside a sphere centered at a surface point
  - Prevents banding
- Occlusion function used to relate sample depth delta and distance from the central point
  - Negative deltas give zero occlusion
  - Small positive depth delta produces high occlusion term
  - Large positive depth delta tend to zero
    - Because this calculation happens in screen space
  - Simple exponentials or look-ups used

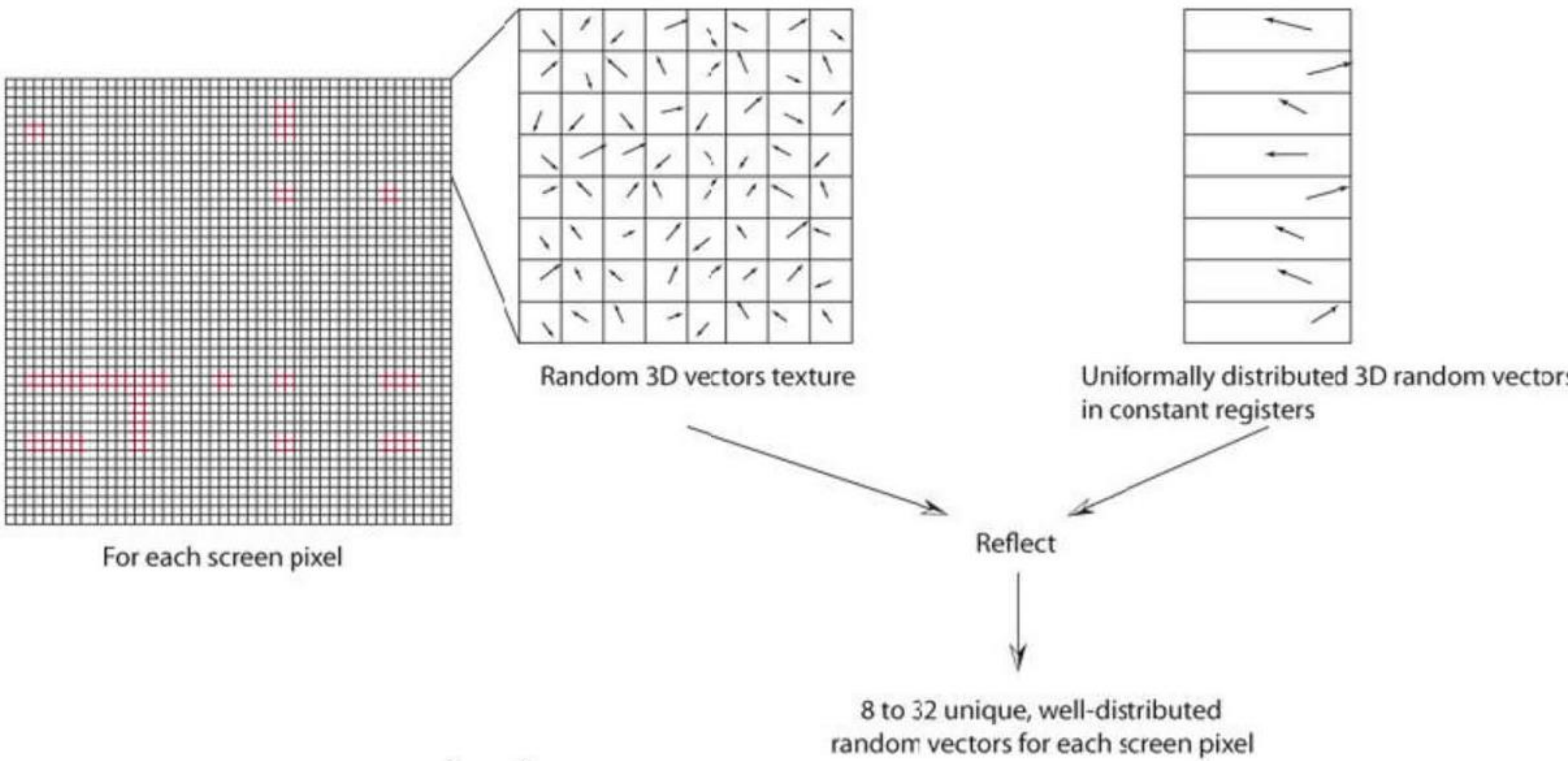
Algorithms. Screen-space



Algorithms. Screen-space

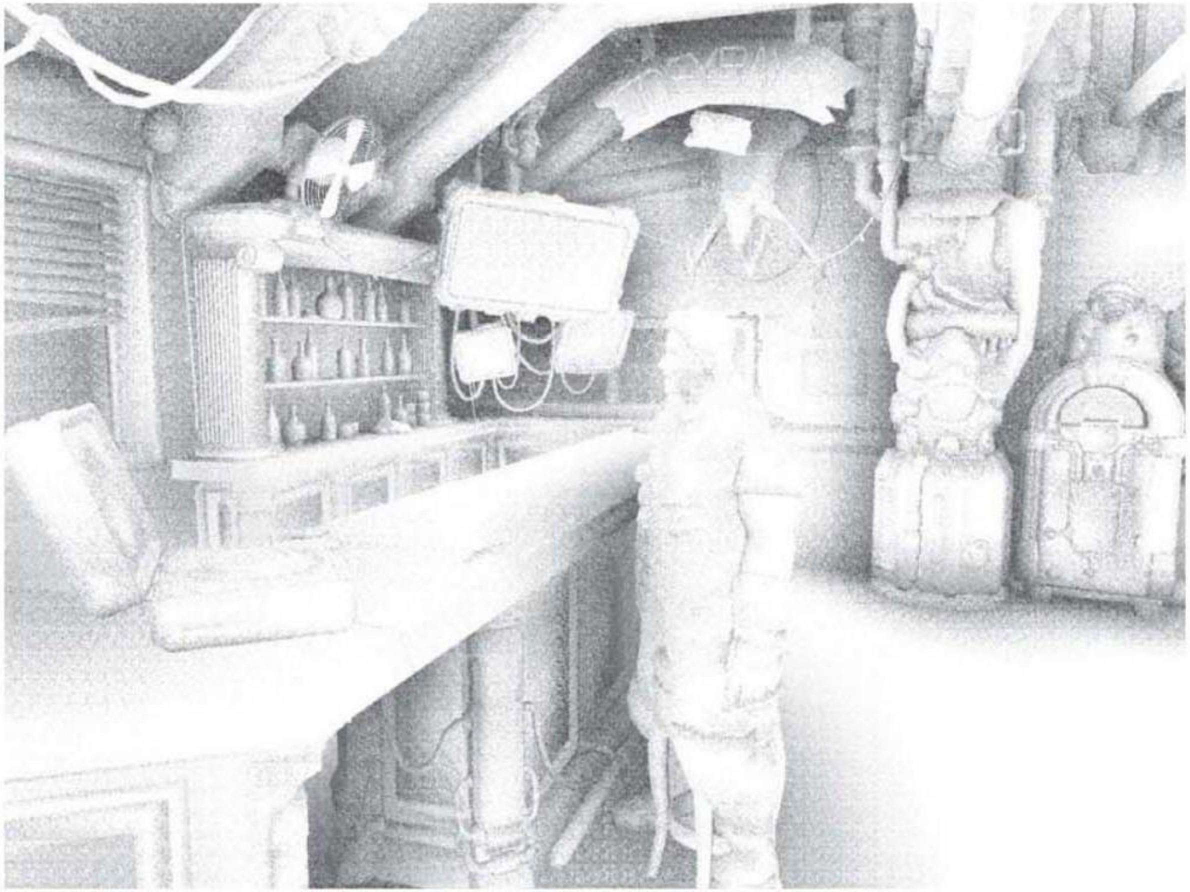
- Calculation:
  - Generate some number of random normal vectors per pixel:
    - 8-32: Starcraft II, 16: Cryengine II
  - Reflect random vectors off another set of varying length vectors with uniform distribution in solid sphere
    - Range of length is scaled by some artistic parameter
  - Samples then passed through the occlusion function

Algorithms. Screen-space



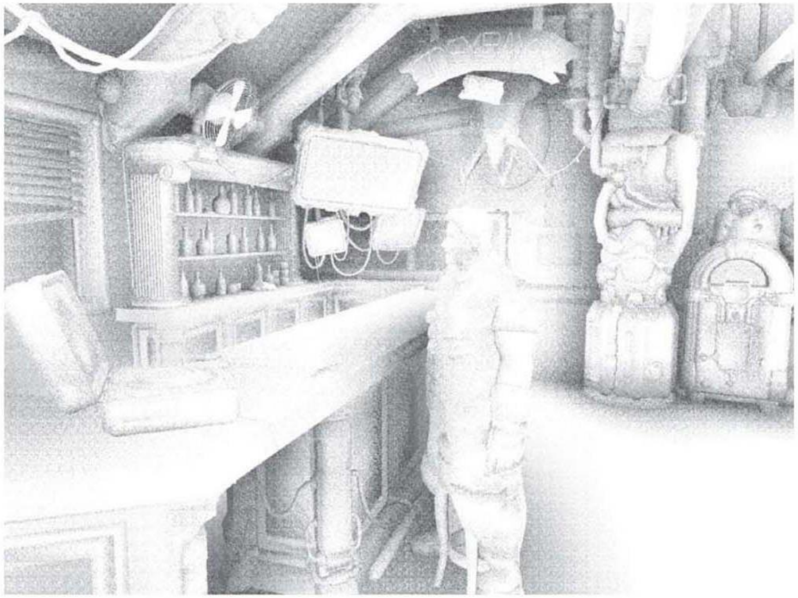
Algorithms. Screen-space

- The result is a noisy image



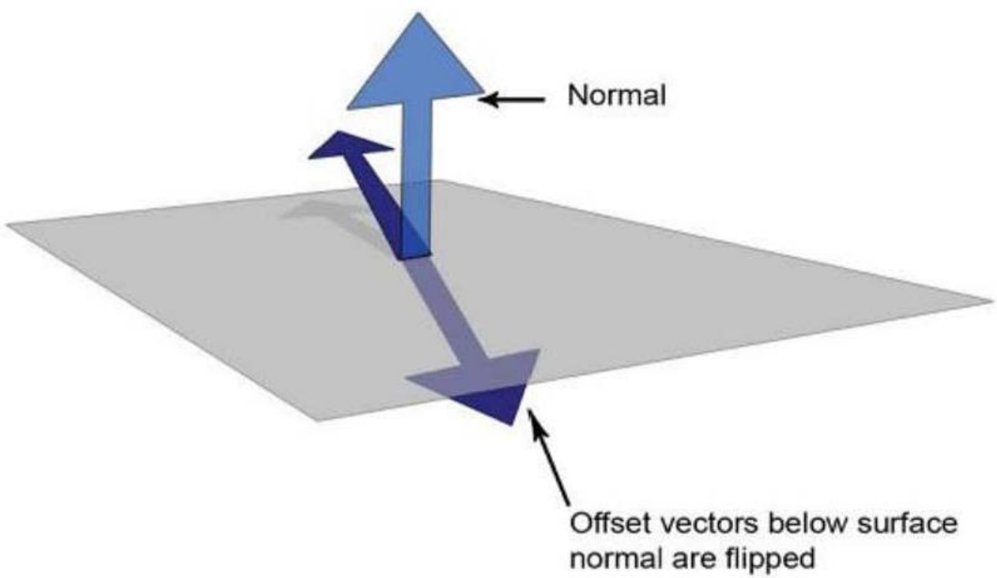
Algorithms. Screen-space

- The result is a noisy image
  - Solution: Blur the resulting image



Algorithms. Screen-space

- Self occlusion:
  - A sample may occlude itself
    - Flip offset vectors



Algorithms. Screen-space

- Issues:
    - No sample outside a certain target:
        - No shadow
    - Camera close to an object: Noise more noticeable
        - Increase the number of samples based on view proximity
        - Constrain the area in which the samples are taken

Algorithms. Screen-space

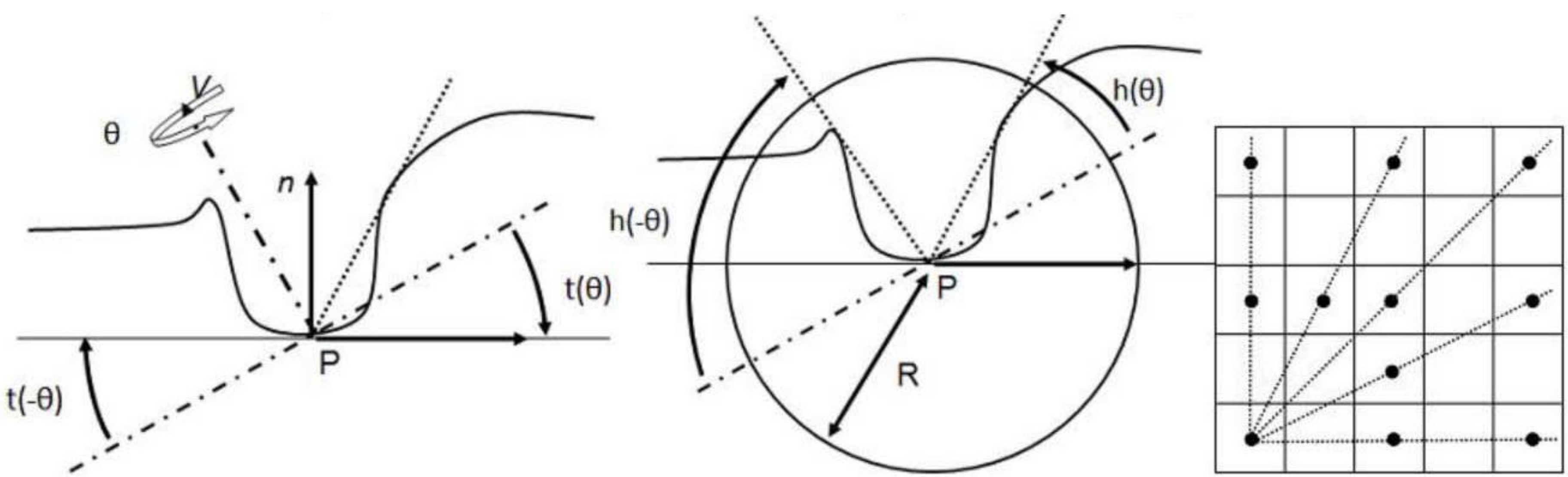
- Performance:
  - Stable, since it is a SS technique
  - Bottleneck at the random sampling
  - Tends to over illuminate solid object edges
  - Lower screen resolution for depth buffer often sufficient
    - ¼ size of the original depth render
  - Multiple SSAO functions can be used together (with different sampling constraints) to model different effects
    - Take the largest occlusion of all occlusion averages

Algorithms. Screen-space

- Horizon-Based AO [Bavoil et al., 2008]
  - For some radius around sample point:
    - Step through depth buffer in some number of randomized directions for a fixed number of samples
    - Find highest altitudes from center within radius (horizon)
    - Average the weighted samples over the directions
    - Intergrade radiance over through visible angle

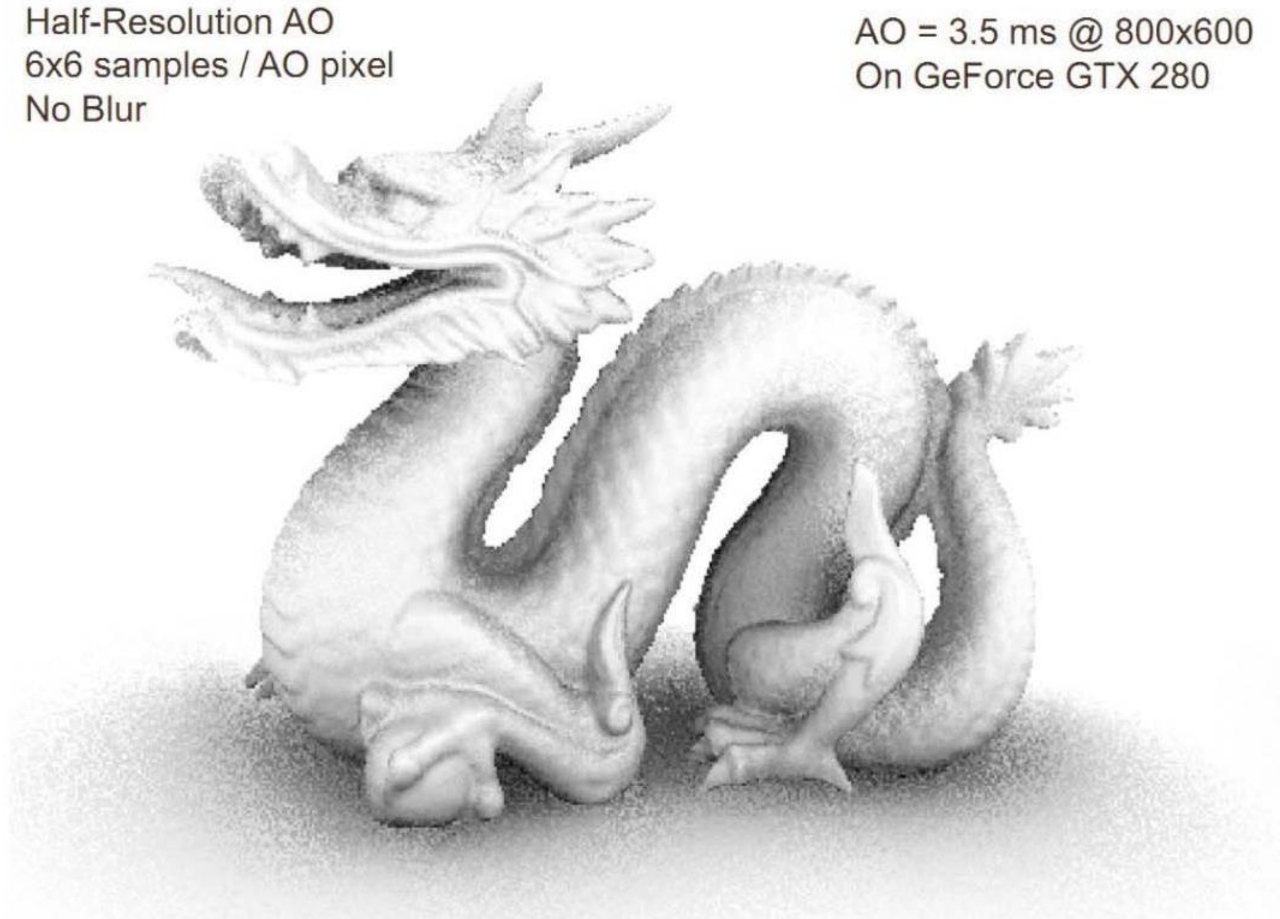
Algorithms. Screen-space

- HBAO:
  - Step through depth buffer in some number of randomized directions for a fixed number of samples
  - Find highest altitudes from center within radius (horizon)



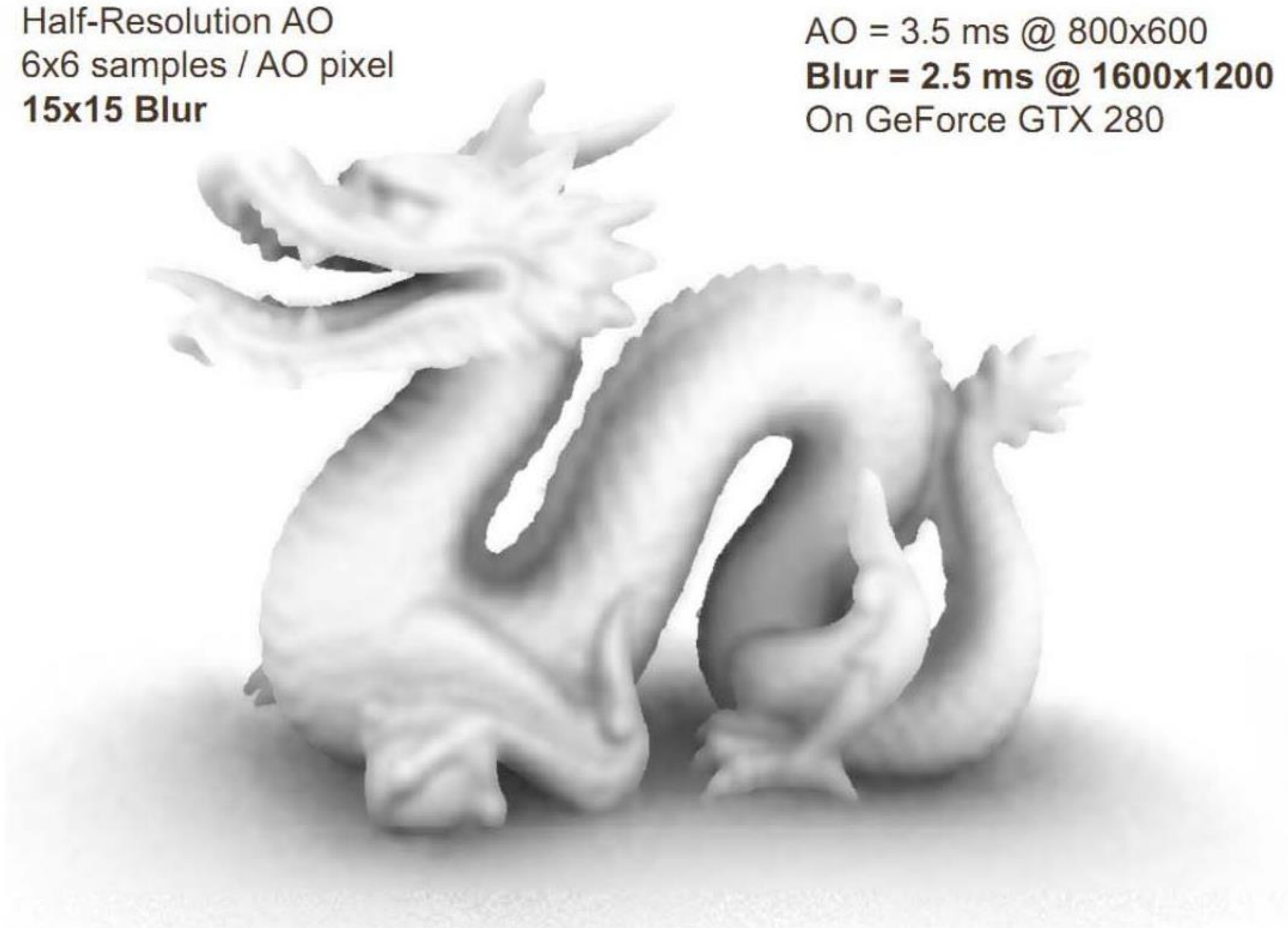
Algorithms. Screen-space

- AO vs HBAO



Algorithms. Screen-space

- AO vs HBAO



Algorithms. Screen-space

- HBAO
  - Interactive frame rates (2008)
  - Also relies on blurring
  - Might require some biasing in horizon angle
  - Per sample falloff (radial falloff function)
  - Jitter samples
  - More accurately simulates ray casting AO

Algorithms. Screen-space

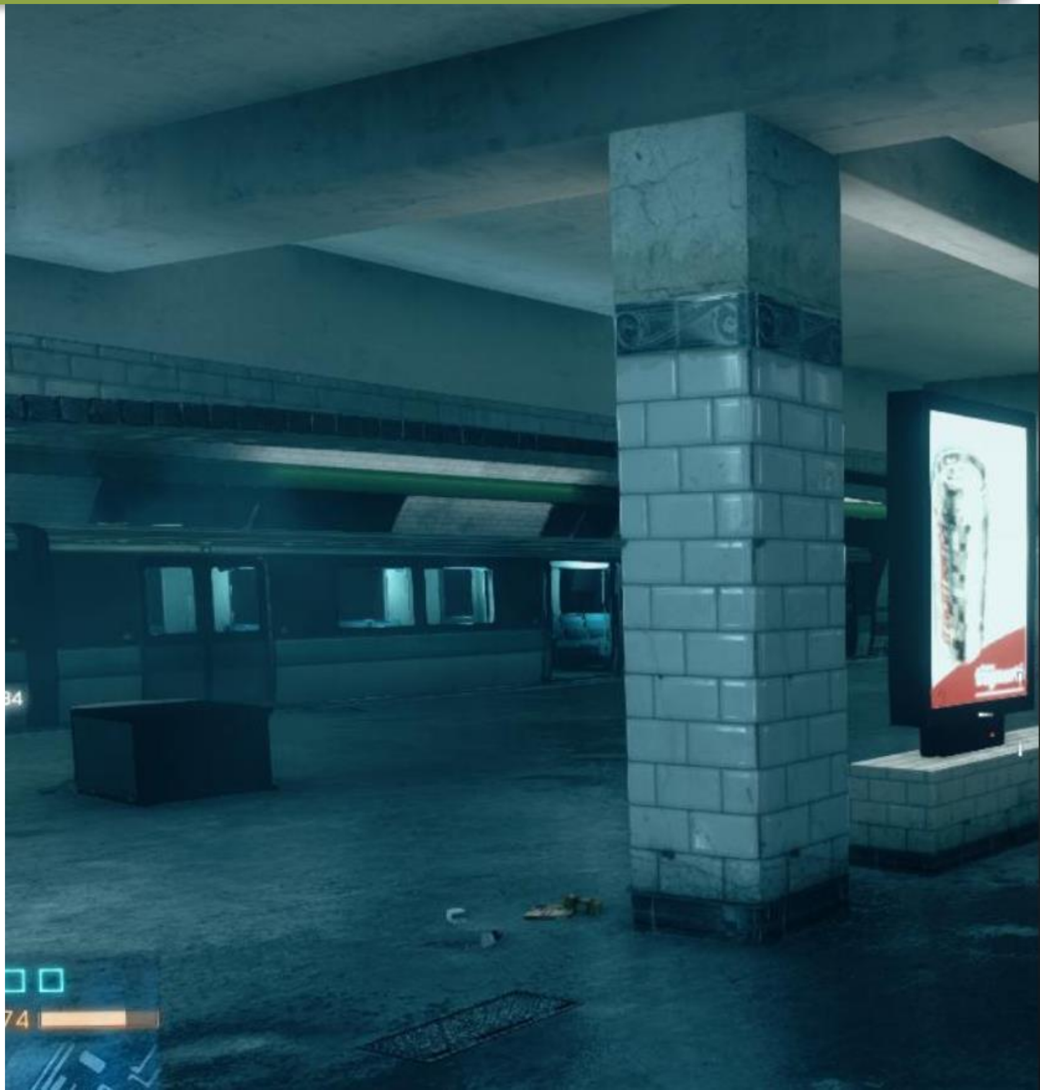
- Battlefield 3.
  - No SSAO

Algorithms. Screen-space

- Battlefield 3.
  - SSAO

Algorithms. Screen-space

- Battlefield 3.
  - HBAO



Algorithms. Screen-space

- Battlefield 3.
  - SSAO

Algorithms. Screen-space

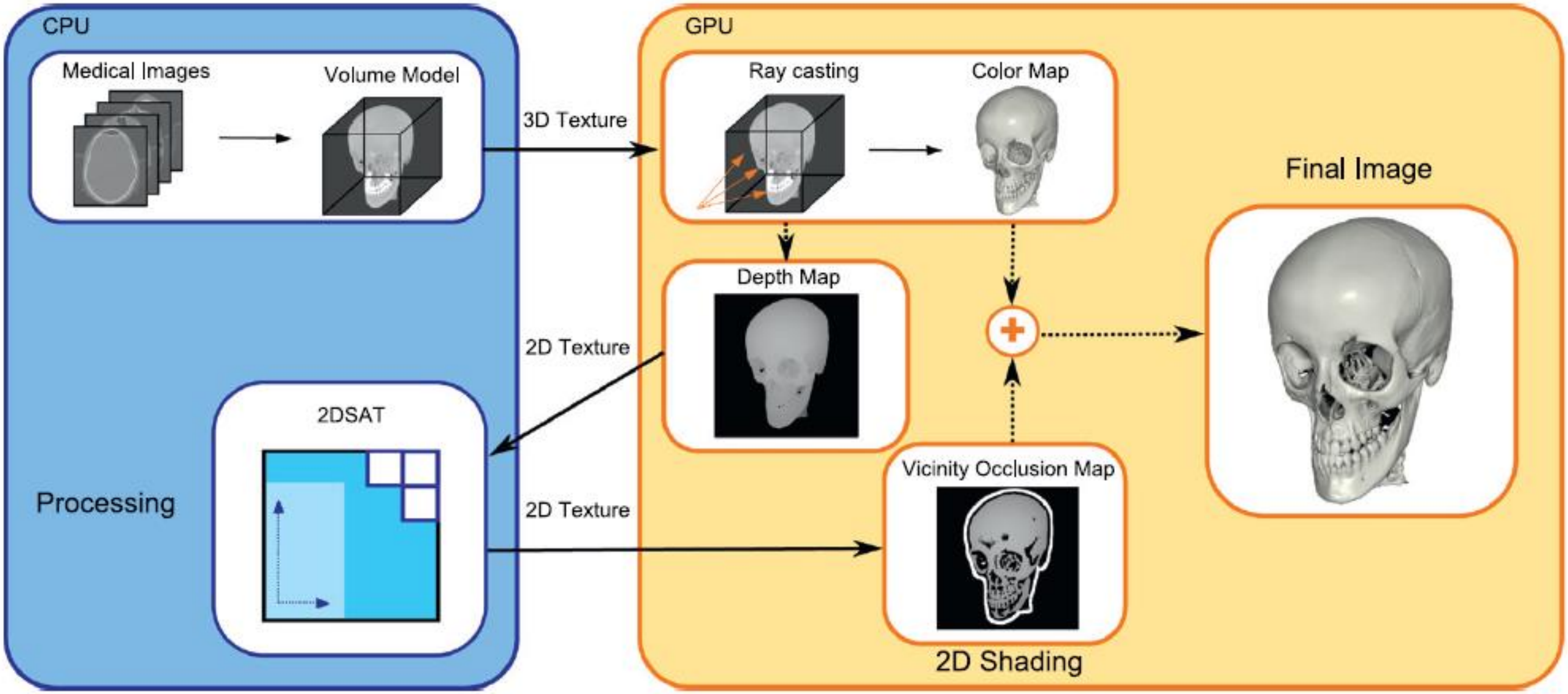
- Battlefield 3. HBAO

Algorithms. Screen-space

- Vicinity Occlusion Maps: AO for volumetric models [Díaz et al., 2008]
  - Approximates ambient occlusion screen-based
  - Uses a data structure based on Summed Area Tables:
    - Reduces sampling cost
    - Provides the blur
    - Halos for free

Algorithms. Screen-space

- Vicinity Occlusion Maps



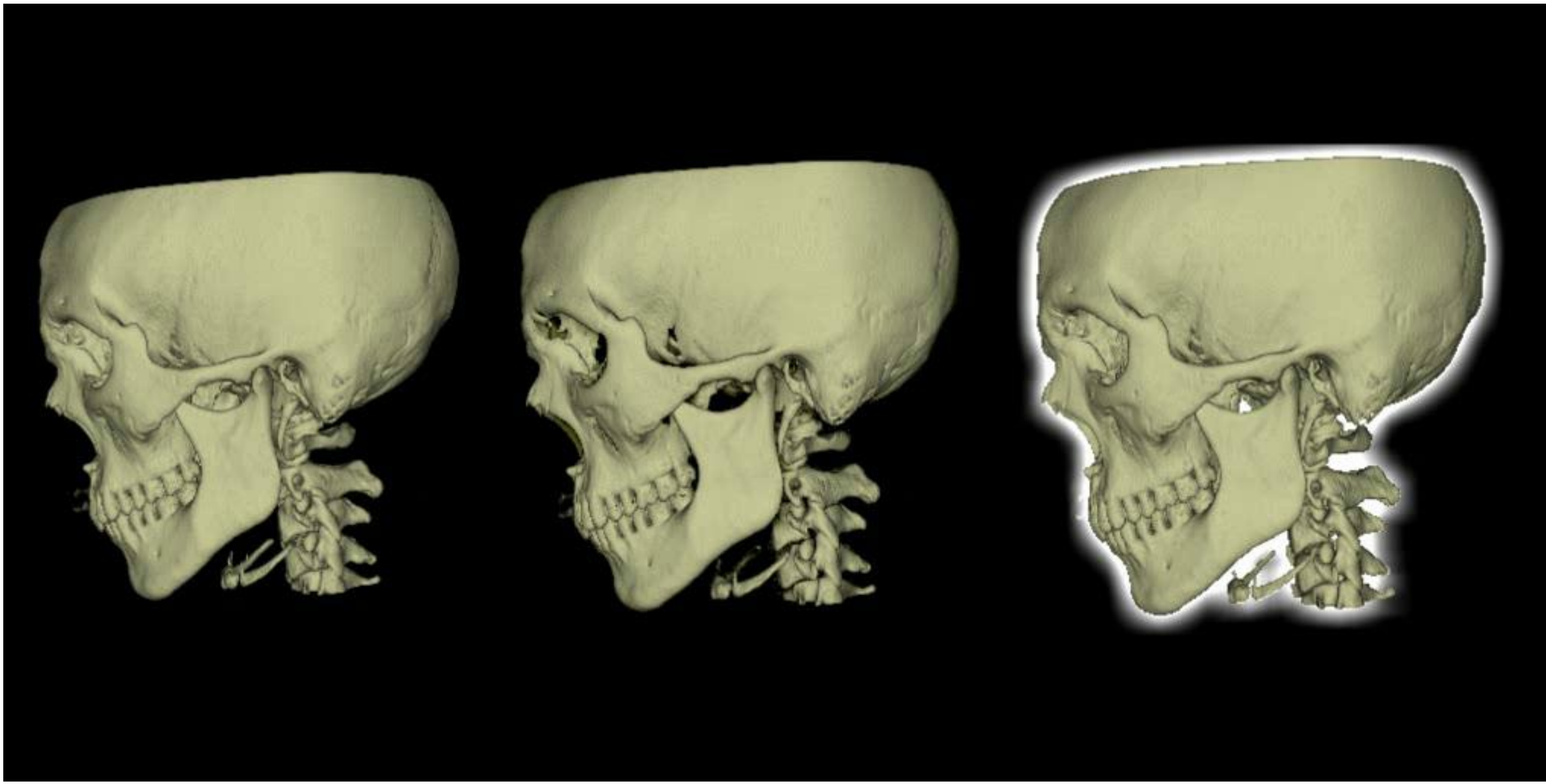
Algorithms. Screen-space

- Vicinity Occlusion Maps

Algorithms. Screen-space

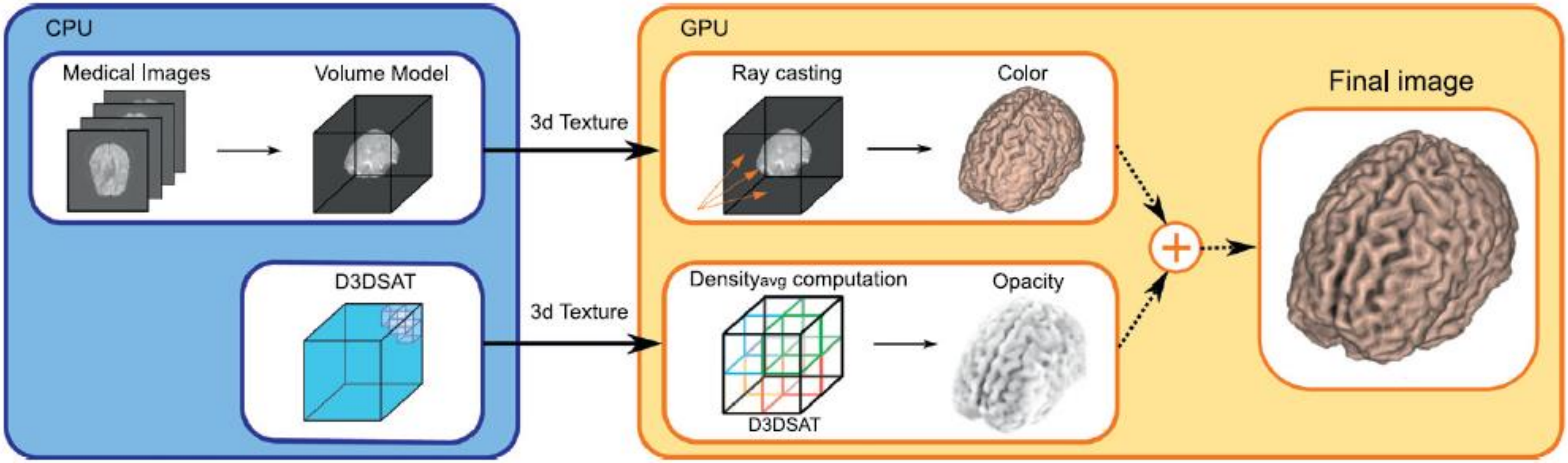
- Vicinity Occlusion Maps

Algorithms. Screen-space



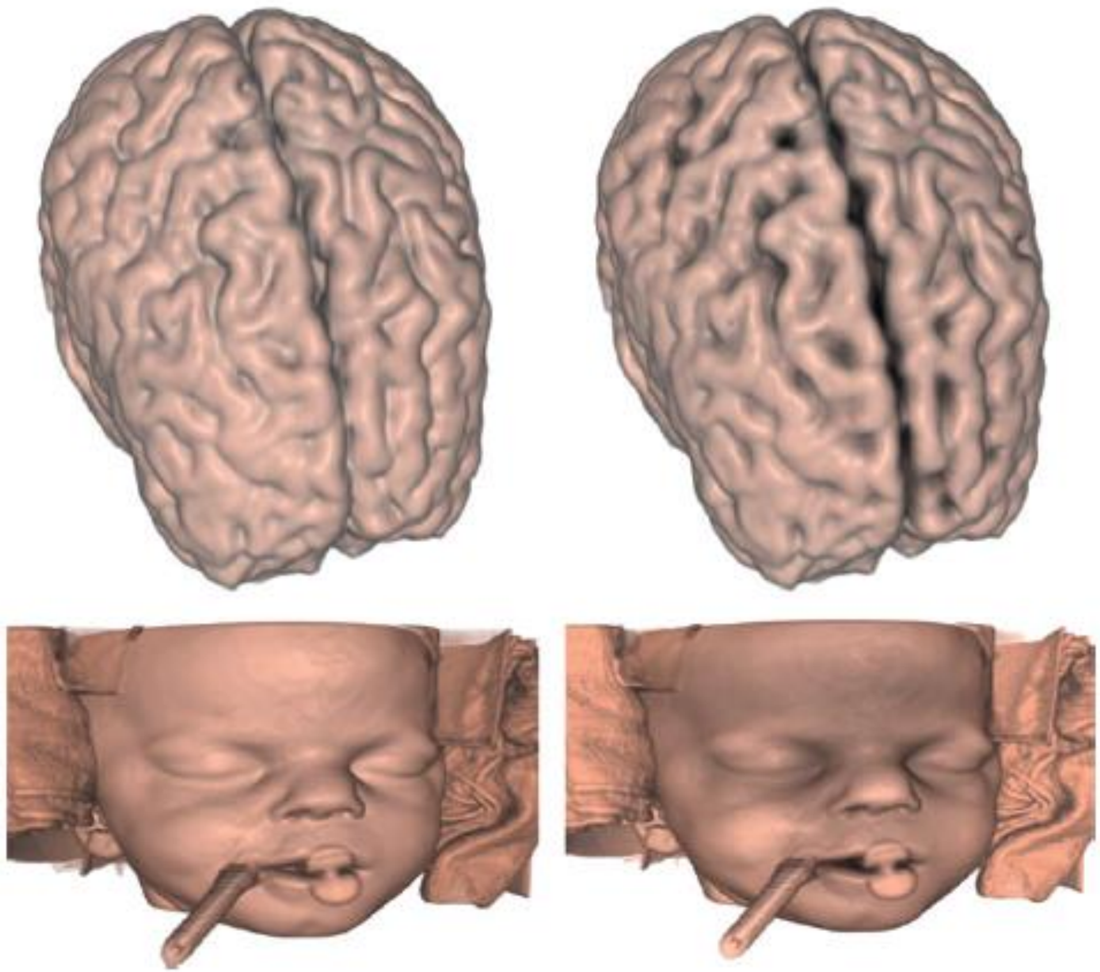
Algorithms. Screen-space

- DSAT3D: View independent AO for volumes [Díaz et al., 2010]
  - Actually more a Volume-based method
  - Uses a 3D SAT for view independent AO calculation



Algorithms. Screen-space

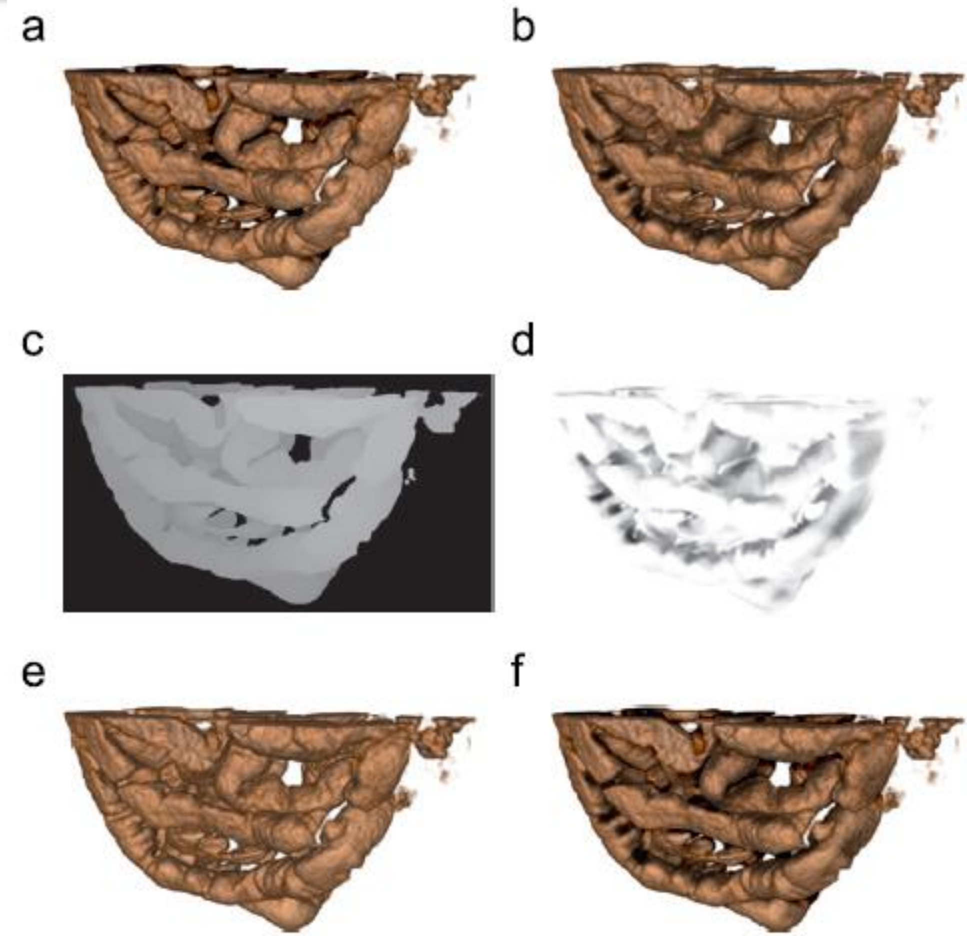
- DSAT3D



Ray casting DSAT3D ambient occlusion

Algorithms. Screen-space

- DSAT3D
  - Combine with 2D VOM



Algorithms. Screen-space

- Conclusions:
  - SSAO methods avoid pre-computation
    - Can be used in dynamic scenes
  - Different techniques concentrate on different aspects of the rendering equation
  - Some techniques may cause (or generate) color bleeding

Algorithms. Screen-space

- Other approaches:
  - Reflective shadow maps
  - Directional occlusion
  - …

Outline

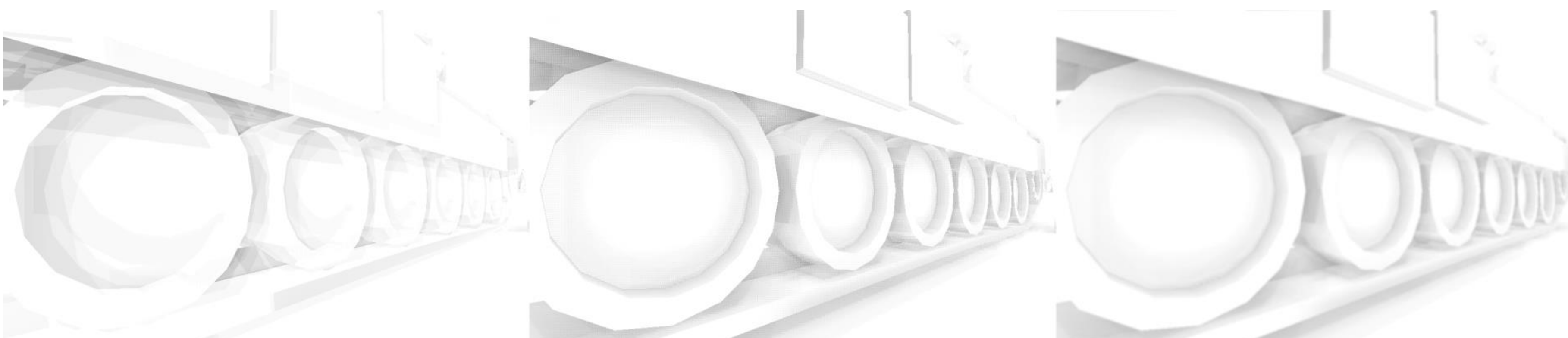
- *Introduction*
- *Derivation*
- *Algorithms*
- **Practical issues**
- Implementation in mobile

Practical issues

- Usually sampling-limited
  - May reduce resolution for depth map
- Distant geometry is tricky
  - Smart fall-off function may help
- Trade noise vs banding

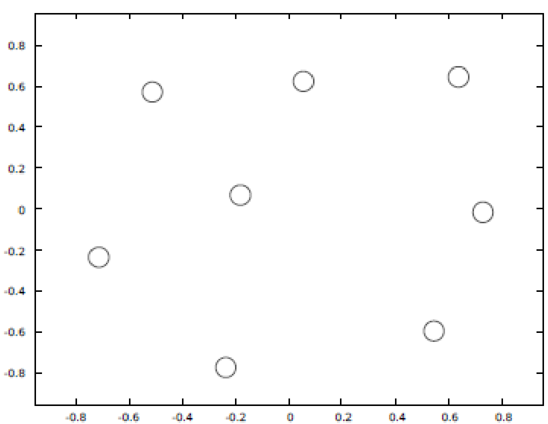
Practical issues

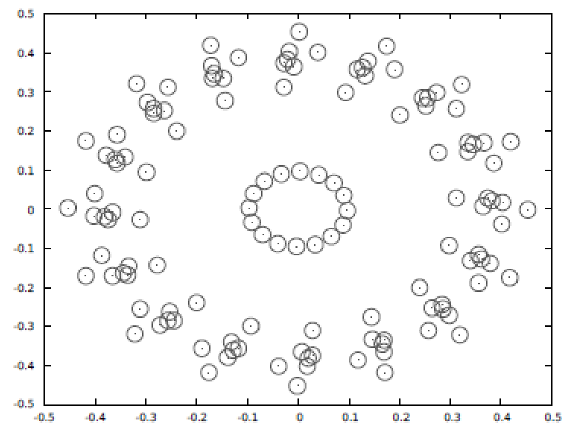
- Banding & Noise
  - A fixed sampling pattern produces banding (left).
  - Random sampling removes banding but introduces noise (middle).
  - SSAO output is typically blurred to remove noise (right).

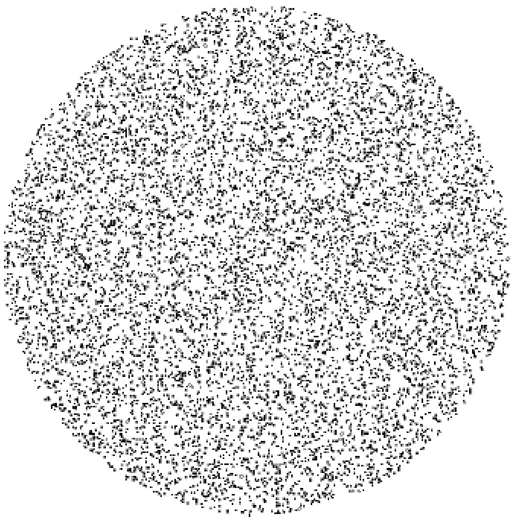


Practical issues

- Sampling
  - Desktop implementations may take 16-32 samples per pixel





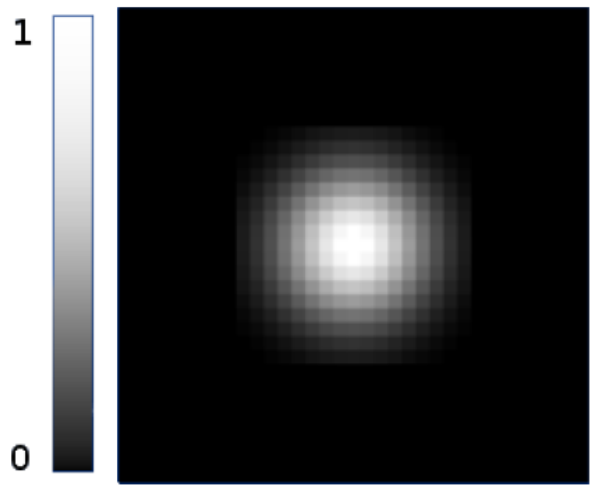


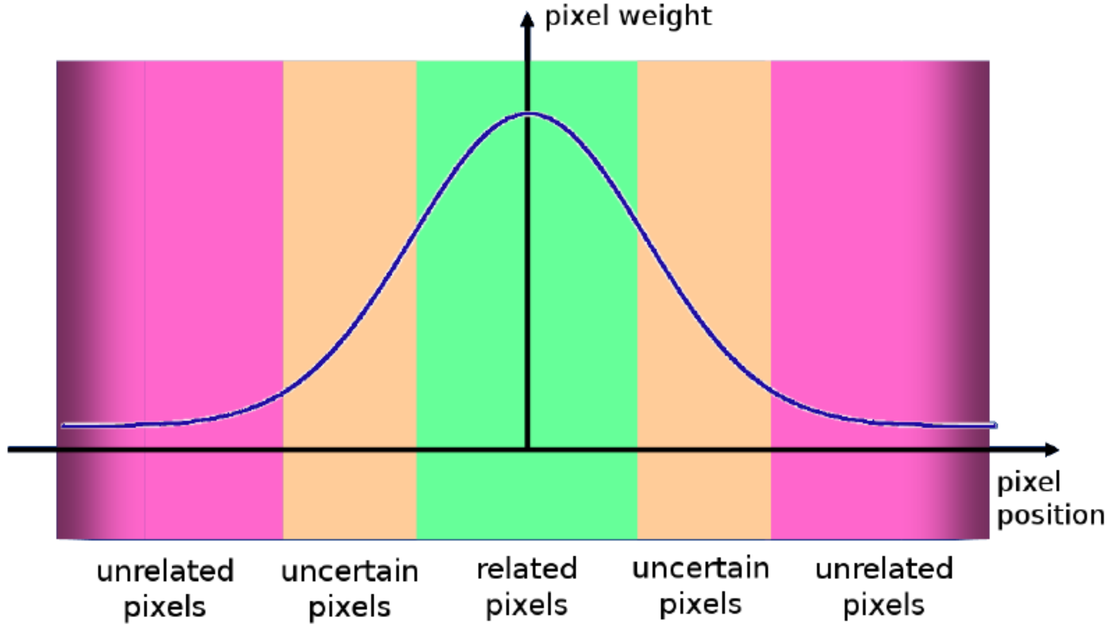
Practical issues

- Reducing resolution
  - AO is usually low frequency, so SSAO/HBAO is performed at half resolution for significant speedup
  - Smart blur is done using full resolution z-buffer to avoid edge bleeding
  - Small high frequency geometry like grass can cause flickering / shimmering due to the half resolution

Practical issues

- Blurring: Typical Gaussian blurring smoothens out edges
  - Use bilateral filtering





Practical issues

- Blurring. Bilateral filtering

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

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

Practical issues

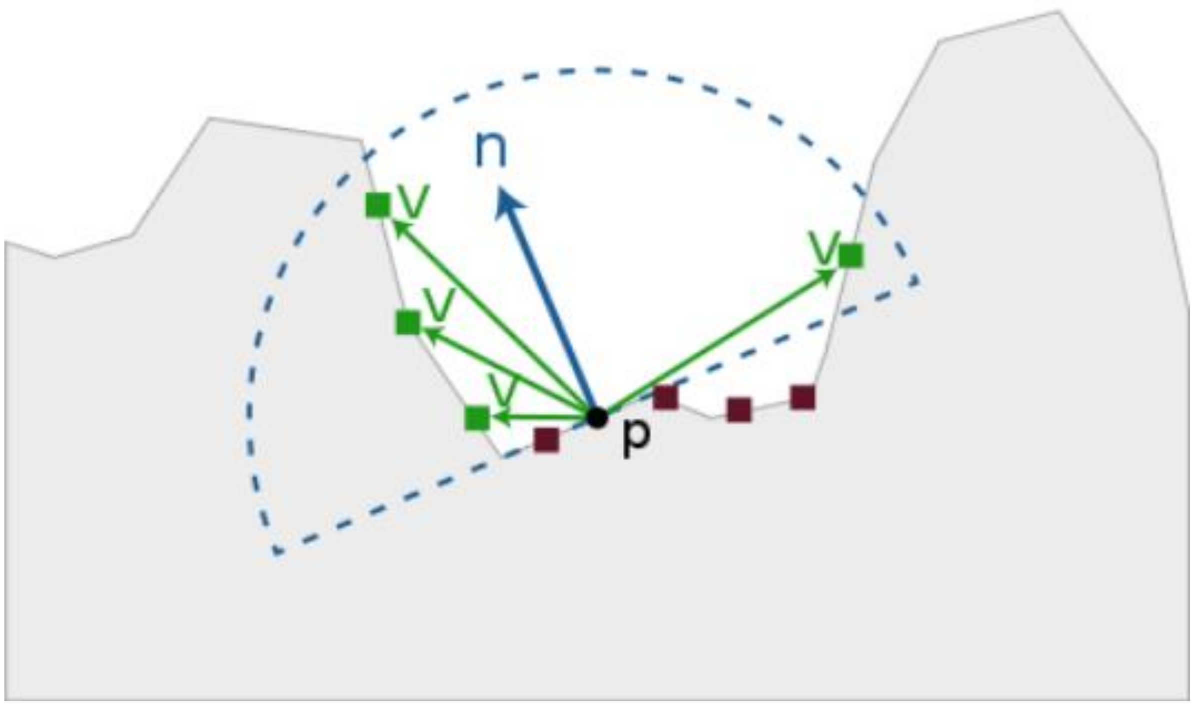
- Solution in Battlefield 3: Temporal filtering
  - AO is dependent only on scene geometry, not camera position
  - Use AO from last frame
    - Reproject it to the current view and interpolate between new and old AO

Outline

- *Introduction*
- *Derivation*
- *Algorithms*
- *Practical issues*
- **Implementation in mobile**

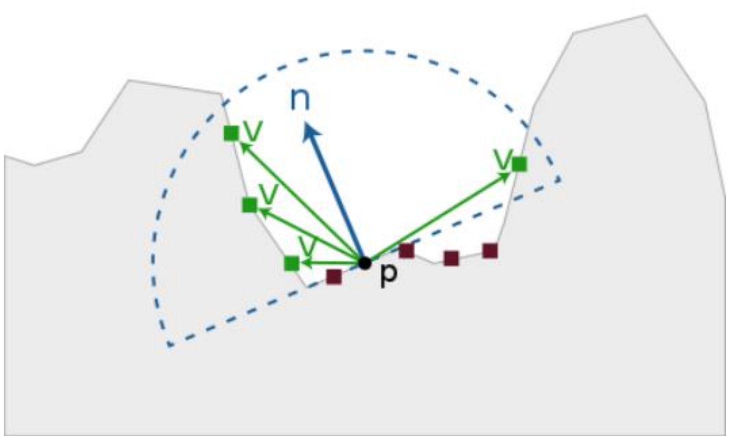
Implementation in mobile

- SSAO: Alchemy-based in OpenGL ES



Implementation in mobile

- Overview of the approach:
  - Sample in a disk (texture space)
    - Unproject to get sample
  - Apply falloff function to map angle (n, sample-p) to AO in [0,1]
  - Use p and s normal to prevent self-occlusion
  - Penalise large depth discontinuities



Implementation in mobile

- Implementation

```
#define NSAMPLES 4
#define NSAMPLESf 4.0

precision mediump float;

uniform sampler2D depth;
uniform sampler2D normal;
uniform sampler2D rotation;
uniform mat4 iProjection;
```

```
uniform vec2 samples[NSAMPLES];
uniform float radius;
uniform vec2 ssaoSize;
uniform float rotationWidth;
uniform vec2 texelSize;

varying vec2 texCoord;
```

Implementation in mobile

- Implementation

```
varying vec2 texCoord;

// Unproject the given point in texture coordinates to view space
   coordinates.
// (s,t, linear depth) -> (x,y,z)
vec3 unproject (vec2 st, float d)
{
    vec2 xy = st*2.0 - 1.0; // x/w and y/w in [-1,1]
    vec4 pfar = (iProjection * vec4(xy, 1.0, 1.0));
    pfar.xyz /= pfar.w;
    return (pfar * d).xyz;
}
```

Implementation in mobile

- Implementation

```
void main ()
{
    vec2  frag    = texCoord; // frag in [0,1]
    float pdepth  = texture2D(depth, frag).r;
    float pradius = radius * (1.0 - pdepth);
    vec3  pnormal = texture2D(normal, frag).rgb*2.0 - 1.0;
    vec3  pview   = unproject(frag, pdepth);

    vec2 rotst  = frag * ssaoSize / rotationWidth;
    vec2 rotvec = texture2D(rotation, rotst).xy*2.0 - 1.0;

    // Alchemy AO
    float AO = 0.0;
    for (int i = 0; i < NSAMPLES; ++i)
    {
```

```
float AO = 0.0;
for (int i = 0; i < NSAMPLES; ++i)
{
     vec2  svec   = pradius * reflect(samples[i].xy, rotvec);
     vec2  qfrag  = frag + svec;
     float qdepth = texture2D(depth, qfrag).r;
     vec3  qview  = unproject(qfrag, qdepth);
     vec3 qnormal = texture2D(normal, qfrag).rgb*2.0 - 1.0;

     // Compute occlusion based on angle
     vec3 v  = normalize(qview - pview);
     float a = smoothstep(0.0, 1.0, dot(v, pnormal));

     // Avoid self-shadowing
     float b = 1.0 - dot(pnormal, qnormal);

     float diff = qdepth - pdepth;

     // Penalise large depth discontinuities
     float dbias = 0.05;
     float c = step(abs(diff), dbias);

     AO += a * b * c;
}
AO = 1.0 - AO/NSAMPLESf;

gl_FragColor.r = AO;
```

Ambient Occlusion

Pere-Pau Vázquez

ViRVIG – UPC