Real-time Graphics

4. Global Illumination

Martin Samuelčík

Rendering equation

$$L_o(\mathbf{x},\omega,\lambda,t) = L_e(\mathbf{x},\omega,\lambda,t) + \int_\Omega f_r(\mathbf{x},\omega',\omega,\lambda,t) L_i(\mathbf{x},\omega',\lambda,t)(-\omega'\cdot\mathbf{n})d\omega'$$

- λ is a particular wavelength of light
- t is time
- $L_o(\mathbf{x},\omega,\lambda,t)$ is the total amount of light of wavelength λ directed outward along direction ω at time t, from a particular position $\mathbf{x}$
- $L_e(\mathbf{x},\omega,\lambda,t)$ is emitted light
- $\int_\Omega \cdots d\omega'$ is an integral over a hemisphere of inward directions
- $f_r(\mathbf{x},\omega',\omega,\lambda,t)$ is the proportion of light reflected from ω' to ω at position $\mathbf{x}$, time t, and at wavelength λ
- $L_i(\mathbf{x},\omega',\lambda,t)$ is light of wavelength λ coming inward toward $\mathbf{x}$ from direction ω' at time t
- $-\omega'\cdot\mathbf{n}$ is the attenuation of inward light due to incident angle

- Global illumination: contribution of neighboring scene points to illumination
- Ambient occlusion, shadows, ray-tracing, radiosity, photon mapping, path tracing, reflections, refractions, caustics, ...

Ambient term

- Simulating light scattered many times by environment
- There are surface points with different number of accumulating rays (plane parts vs. corners)
- Ambient light is NOT constant for all points
- Perceptual clues – depth, curvature, spatial proximity

Ambient occlusion

Ambient occlusion

- For illuminating point P, compute visibility from P in all hemisphere directions

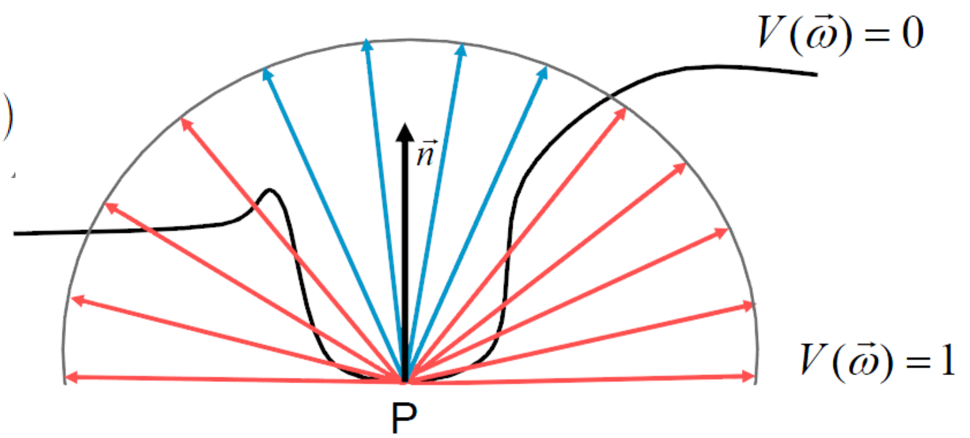
$$AO(P,\vec{n}) = \frac{1}{\pi}\int_{\Omega} V(P,\vec{\omega})\cdot max(\vec{n}\cdot\vec{\omega},0)\,d\vec{\omega}$$

$$P - illuminated\ point$$

$$\vec{n} - normal\ at\ point\ P$$

$$V(P,\vec{\omega}) - visibility\ function$$

$$AO(P,\vec{n}) = \frac{1}{\pi}\sum_{\Omega} V(P,\vec{\omega})\cdot max(\vec{n}\cdot\vec{\omega},0)$$



AO computation

- Monte Carlo – computing integral by sampling hemisphere with random rays

$$AO(P,\vec{n})=\frac{1}{n}\sum_{i=0}^{n-1} V(P,\vec{rnd_i\omega})\cdot max(\vec{n}\cdot\vec{rnd_i\omega},0)$$

$$\vec{rnd_i\omega}=i-th\ random\ vector$$

- Distribution of random rays?
- Still not real-time
- Static geometry & lights – offline precomputation of ambient maps

AO – object space

- Compute intersection with of ray from illuminating point with objects in scene
- Intersection with hierarchical simplified geometry
- Intersection only with close objects
- Usually computation per-vertex
- Performance dependent on geometry complexity

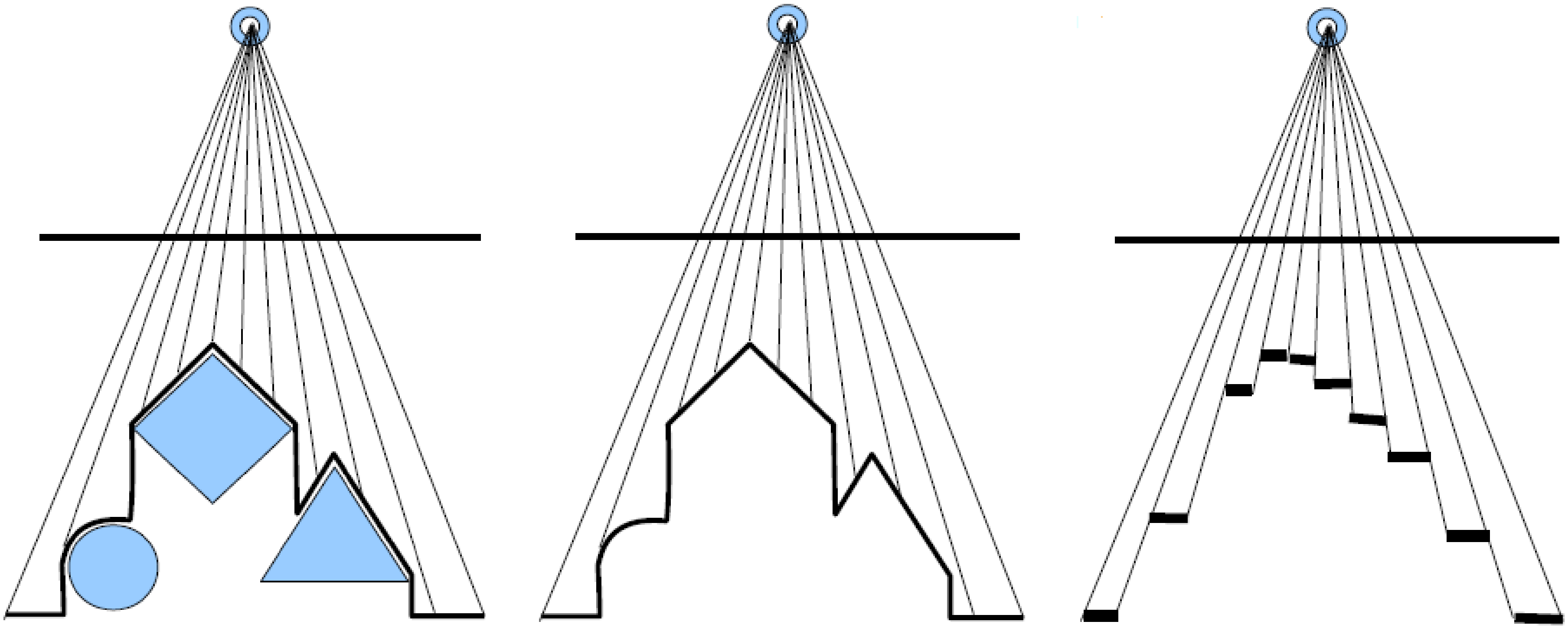
AO – screen space

- Computation of visibility function in screen space, using data about pixels (fragments)
- State of the Art
- Post-processing effect
- Geometry independent
- Requires (1. pass)
  - Pixel depth values
  - Pixel normal values

Blizzard Entertainment

Depth buffer

- Approximation of visible geometry



SSAO

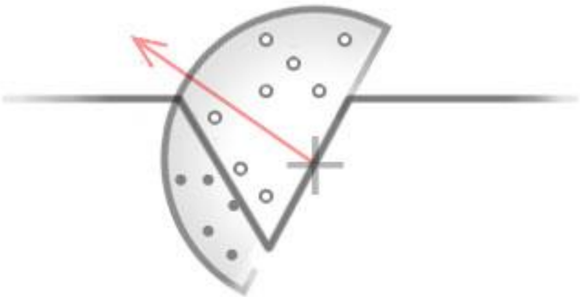
- Screen-space ambient occlusion
- Computation of visibility functions from depth values and from normals
- For illuminating point P, sample depth values in P neighborhood and approximate visibility function
- Sampling first in given direction and given region of interest

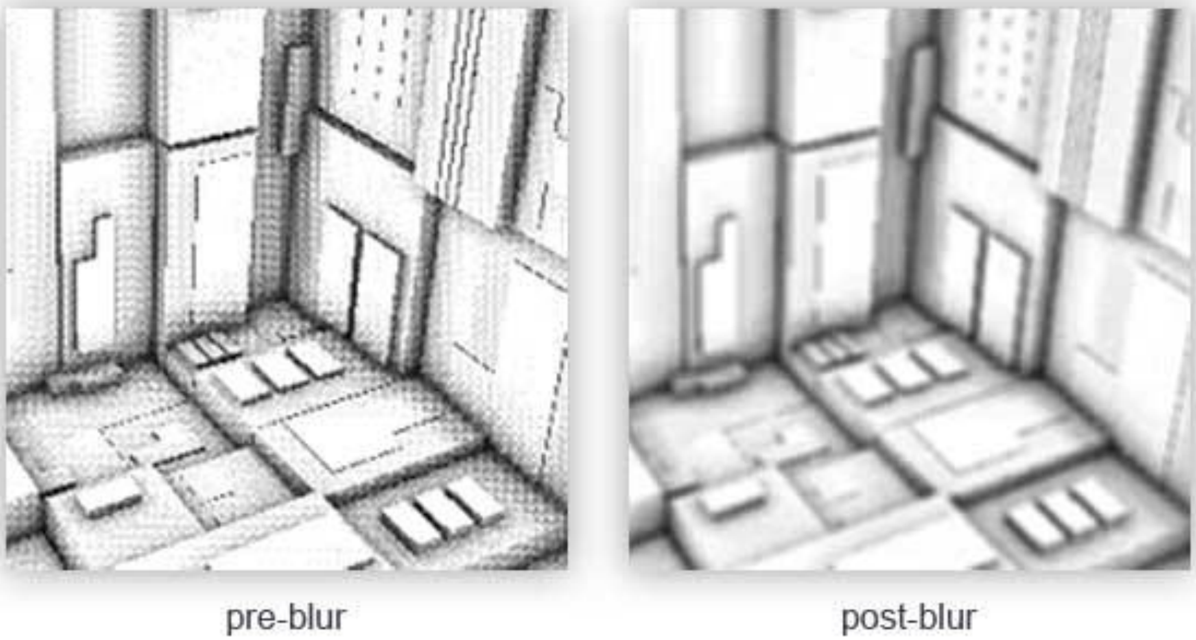
Basic SSAO

- Normal-oriented Hemisphere algorithm
- In each fragment, get eye position of fragment, then place there hemisphere oriented by eye-space normal in fragment
- Create samples inside hemisphere
- For each sample, project it and get depth of sample from depth buffer
- If computed depth of sample is larger than depth of sample from depth buffer, then sample is behind some geometry and fragment is occluded by that geometry

Basic SSAO

http://john-chapman-graphics.blogspot.co.uk/2013/01/ssao-tutorial.html





Basic SSAO

- 1.pass – rendering of eye-space normal and linear depth to textures from camera
- 2.pass – rendering full screen quad with binded normal and depth textures

```
// SSAO – 1.pass - vertex shader
uniform float near, far;
varying vec3 normal_eye;
varying float linear_depth;

void main(void)
{
   // compute normal of vertex in eye coordinates
   normal_eye = gl_NormalMatrix * gl_Normal;
   // compute position of vertex in eye coordinates
   vec4 pos_eye = gl_ModelViewMatrix * gl_Vertex;
   // compute normalized linear depth of vertex ine eye coordinates
   linear_depth = (-pos_eye.z - near) / (far - near);
   // compute position of vertex in clip coordinates
   gl_Position = gl_ProjectionMatrix * pos_eye;
}
```

```
// SSAO – 1.pass - fragment shader
varying vec3 normal_eye;
varying float linear_depth;

void main(void)
{
   // store normal and depth in result texture
   gl_FragColor.rgb = 0.5 * normal_eye + 0.5;
   glFragColor.a = linear_depth;
}
```

Basic SSAO

- 1.pass – rendering of eye-space normal and linear depth to textures from camera
- 2.pass – rendering full screen quad ([0,0,-1], [1,0,-1], [1,1,-1], [0,1,-1]) with binded normal and depth texture

```
for (int i = 0; i < sampleKernelSize; ++i)
{
   sampleKerne[i] = vec3( random(-1.0f, 1.0f),
                          random(-1.0f, 1.0f),
                          random(0.0f, 1.0f));
   sampleKernel[i].normalize();
   sampleKernel[i] *= random(0.0f, 1.0f);
   float scale = i / float(sampleKernelSize);
   scale = lerp(0.1f, 1.0f, scale * scale);
   sampleKernell[i] *= scale;
}
```

```
// SSAO  – 2.pass – vertex shader
varying vec2 vTexCoord;

void main(void)
{
   vTexCoord = vec2(gl_Vertex);
   gl_Position = 2 * gl_Vertex - 1;
}
```

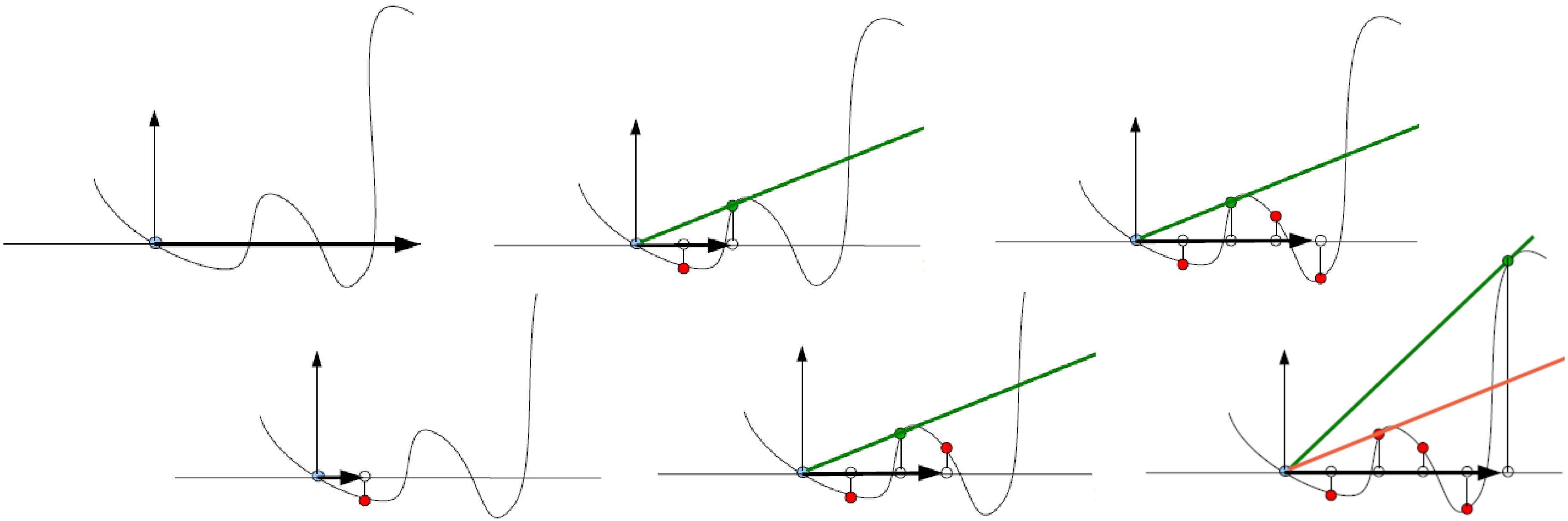
```
// SSAO – 2.pass - fragment shader
varying vec2 vTexCoord;
uniform sampler2D normalDepthTexture;
uniform sampler2D randomTexture;
uniform mat4 projMatrix;
uniform mat4 invProjMatrix;
uniform float near, far, checkRadius;

void main(void)
{
   // reconstruct eye-space position of fragment from linear depth
   float T1 = projMat[2][2];
   float T2 = projMat[2][3];
   float E1 = projMat[3][2];
   float linear_depth = -(near + (far - near) * texture2D(normalDepthTexture, vTexCoord).a);
   vec4 pos_clip;
   pos_clip.w = E1 * linear_depth;
   pos_clip.x = (2 * vTexCoord.x – 1) * pos_clip.w;
   pos_clip.y = (2 * vTexCoord.y – 1) * pos_clip.w;
   pos_clip.z = (T1 * linear_depth + T2);
   vec4 pos_eye = invProjMat * pos_clip;
```

Basic SSAO

```
// SSAO – 2.pass - fragment shader - continuing
  // get TBN matrix for transforming samples from TBN to eye-space
  vec3 normal = texture(normalDepthTexture, vTexcoord).xyz * 2.0 - 1.0;
  normal = normalize(normal);
  vec3 randomVec = texture(randomTexture, vTexCoord * 10).xyz * 2.0 - 1.0;
  vec3 tangent = normalize(randomVec - normal * dot(randomVec, normal));
  vec3 bitangent = cross(normal, tangent);
  mat3 TBN = mat3(tangent, bitangent, normal);

  float occlusion = 0.0;
  for (int i = 0; i < sampleKernelSize; ++i)
  {
    // get sample position in eye space
    vec3 sample_eye = TBN * sampleKernel[i];
    sample_eye = sample_eye * checkRadius + pos_eye.xyz;

    // project sample position to NDC
    vec4 sample_ndc = projMatrix * vec4(sample_eye, 1.0);

    // get depth of sample from depth texture
    float sampleDepth = texture2D(normalDepthTexture, 0.5 * (sample_ndc.xy / sample_ndc.w) + 0.5).a;
    sampleDepth = -(near + (far - near) * sampleDepth);

    // range check & accumulate
    float rangeCheck= abs(pos_eye.z - sampleDepth) < checkRadius ? 1.0 : 0.0;
    occlusion += (sampleDepth <= sample_eye.z ? 1.0 : 0.0) * rangeCheck;
  }
  gl_FragColor.r = occlusion / sampleKernelSize;
}
```

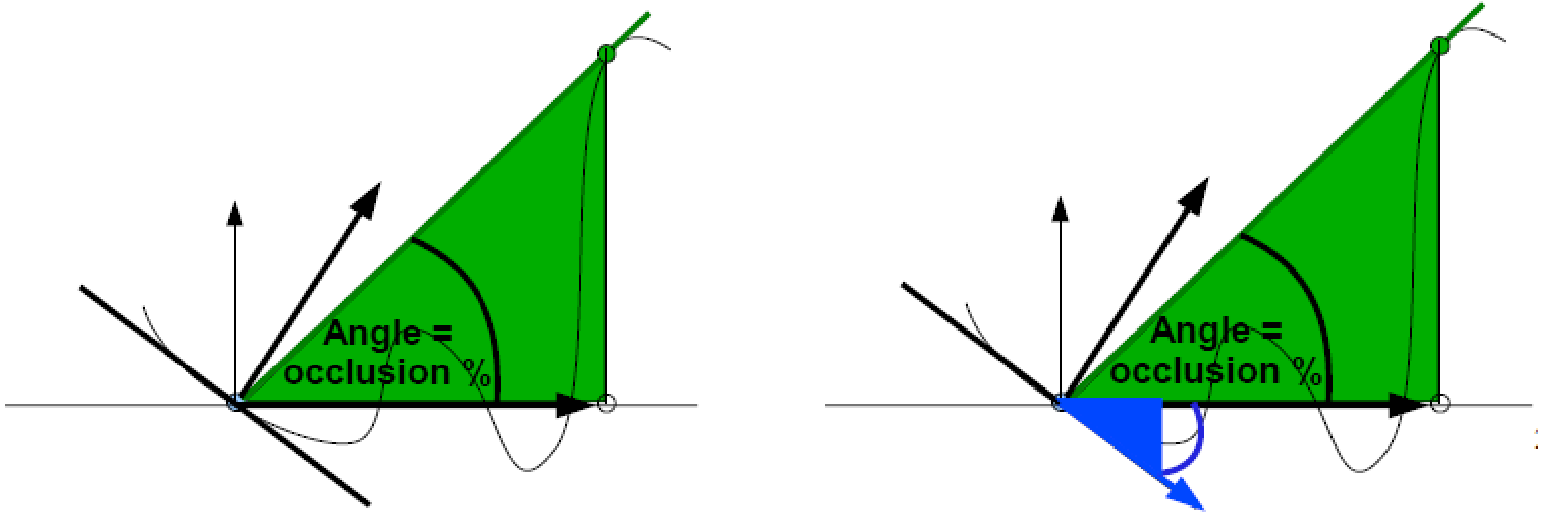
Horizon-based SSAO

- Presented by NVIDIA at SIGGRAPH 2008
- Occlusion in height field
- Sampling height field along ray



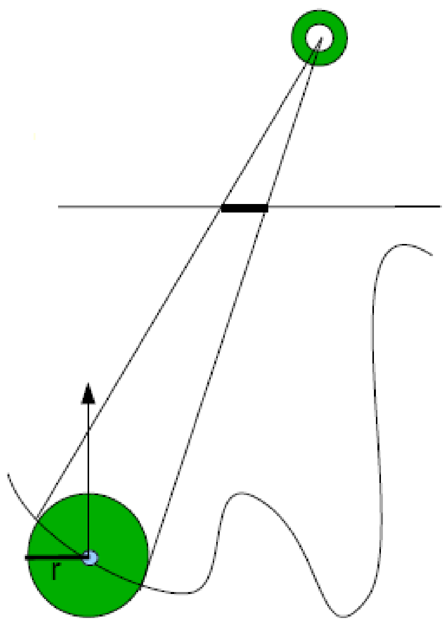
Horizon-based SSAO

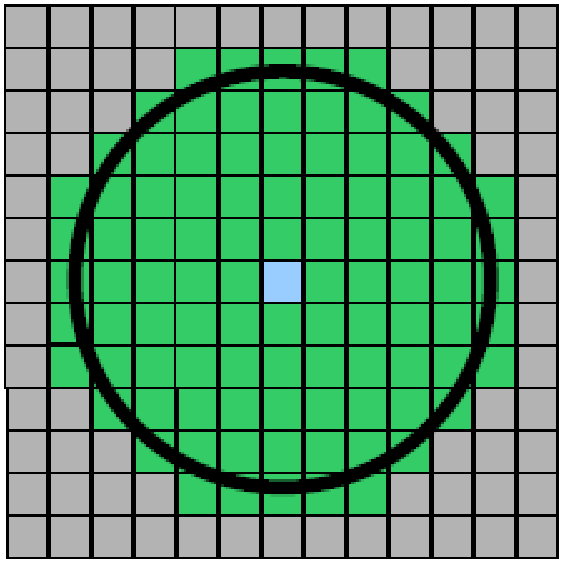
- Getting horizon angle $h(\Theta)$ for direction Θ
- Given normal in P -> tangent vector in P -> signed tangent angle $t(\Theta)$
- $AO = \sin(h(\Theta)) - \sin(t(\Theta))$



SSAO parameters

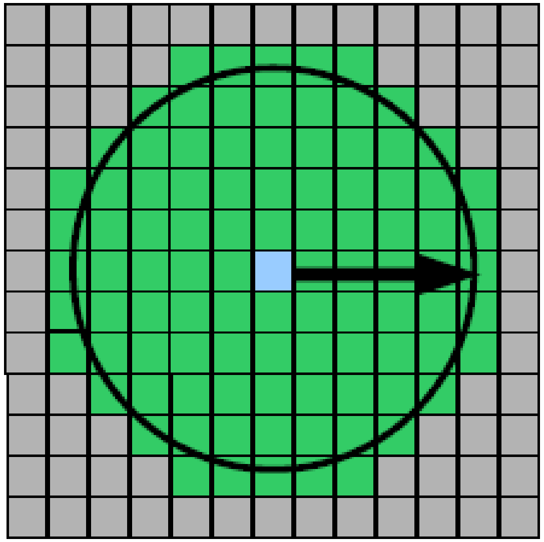
- Number of samples per ray = area of interest
- Radius r is constant and defined in eye space
- Calculate projection of sphere in screen space

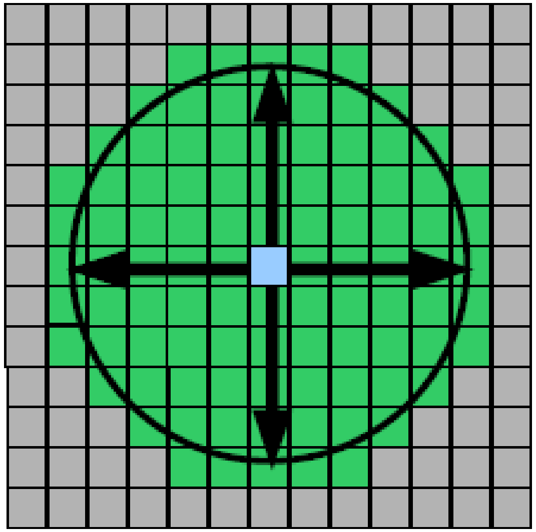


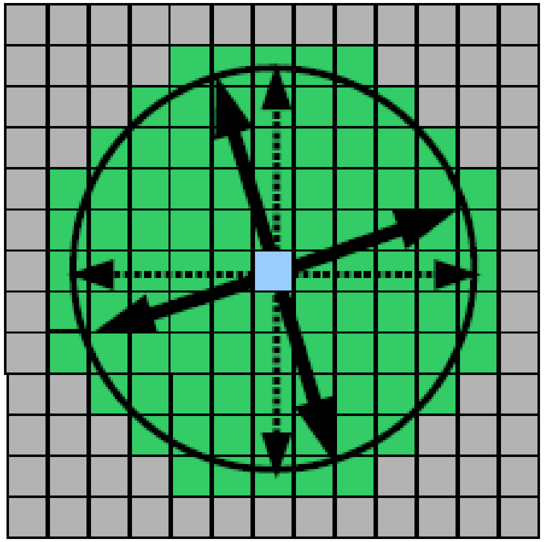


SSAO parameters

- Sampling rays – user defined number
- Random rotation of rays
- Jitter samples along ray

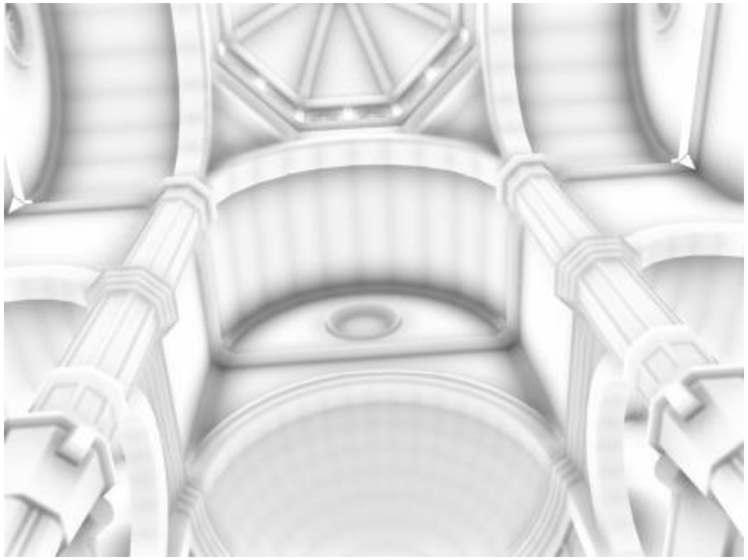




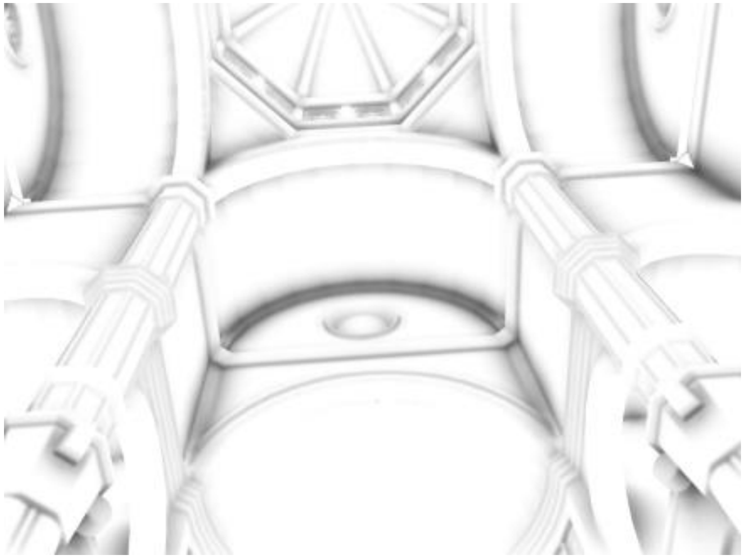


SSAO angle bias

- Ignore occlusion near the tangent plane
- Remove low tessellation artifacts
- Horizon angle is at least some value



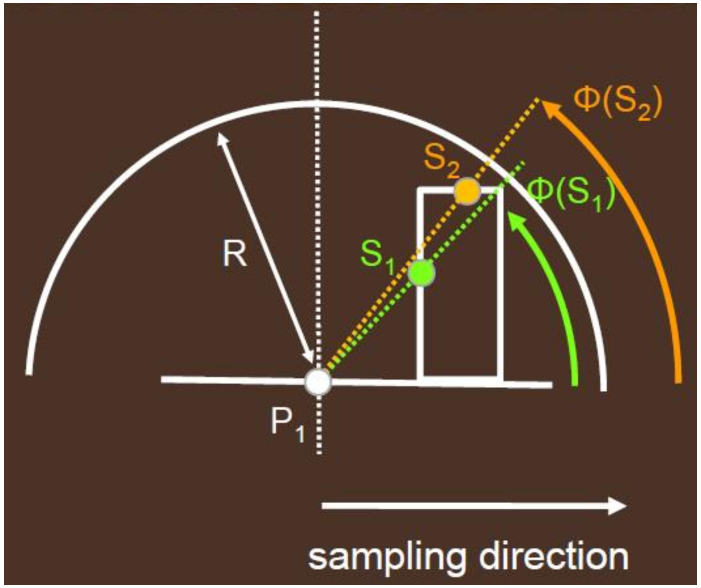
No angle bias



With 30 deg angle bias

Distance attenuation

- Solving large differences of SSAO values for neighboring pixels
- Using weights for each sample, $W(r) = 1-r^2$, …
- Cumulating AO while sampling along ray
  - Initialize WAO = 0
  - After sample S_1
    - $AO(S_1) = \sin(\Phi(S_1)) - \sin t$
    - $WAO \mathrel{+}= W(S_1)\, AO(S_1)$
  - After sample S_2
    - If $\Phi(S_2) > \Phi(S_1)$
    - $AO(S_2) = \sin(\Phi(S_2)) - \sin t$
    - $WAO \mathrel{+}= W(S_2)(AO(S_2)-AO(S_1))$



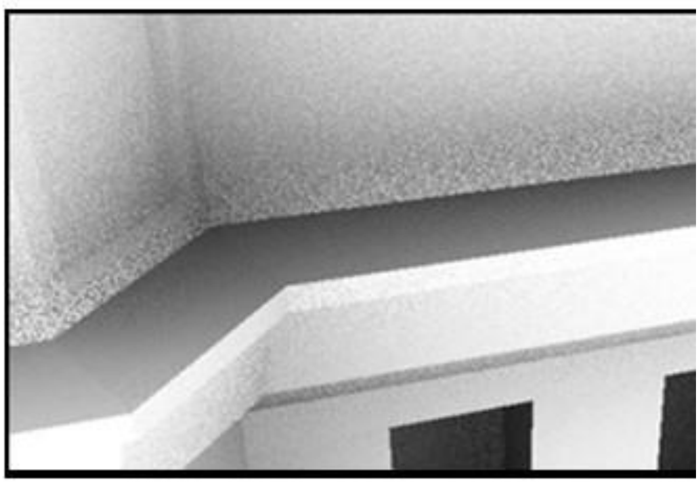
Distance attenuation

No attenuation

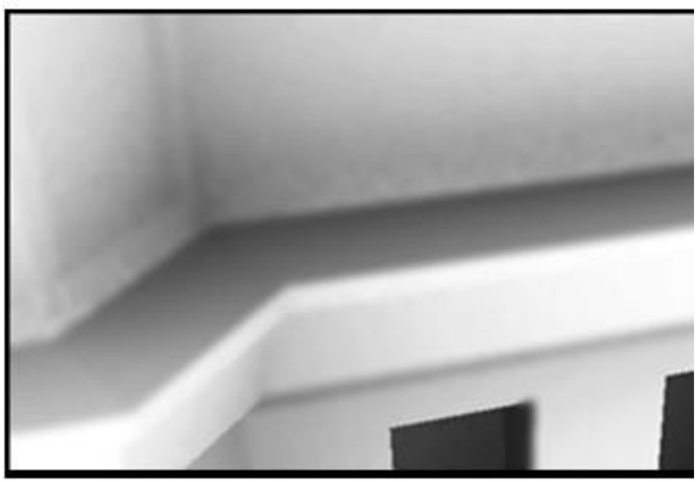
With attenuation, $W(r)=1-r^2$

Noise reduction

- Sampling only few values -> noise, alias
  - Process downscaled depth/normal buffers
  - Blur AO values (remove high frequencies), use depth dependent Gaussian blur



Without Blur

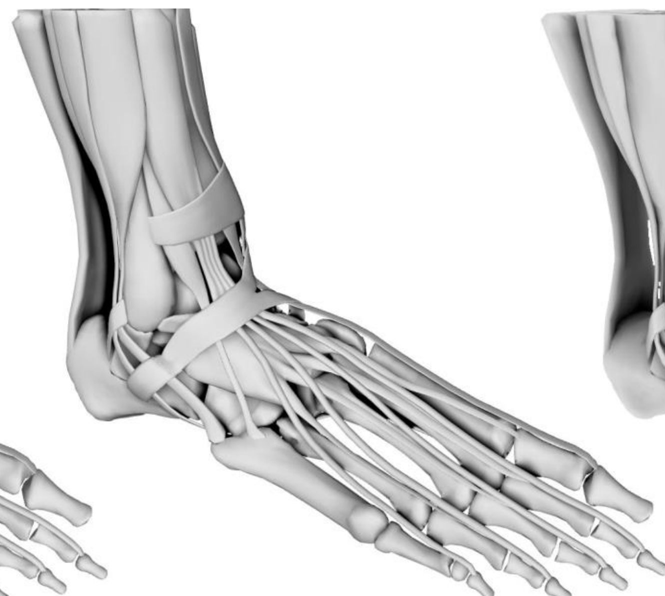


With 15x15 Blur

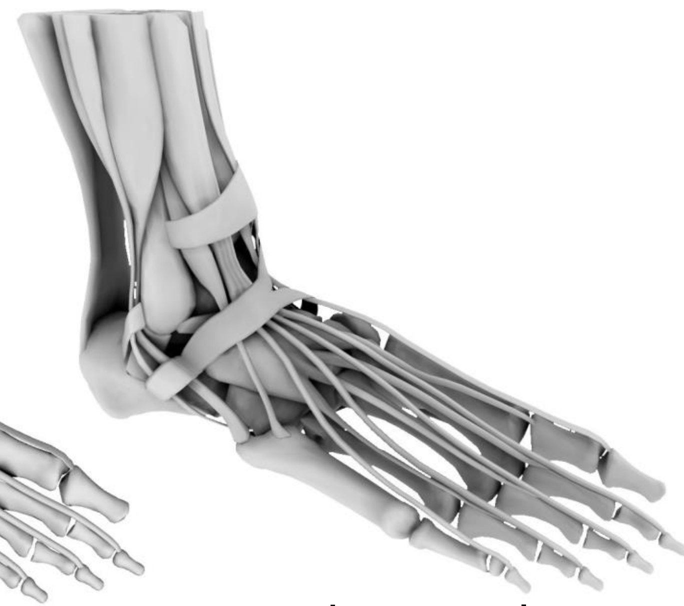
Depth buffer masking

- Depth buffer is blurred, then subtracted from the original depth buffer
- Difference is added to color channels

Original rendering



Depth darkening



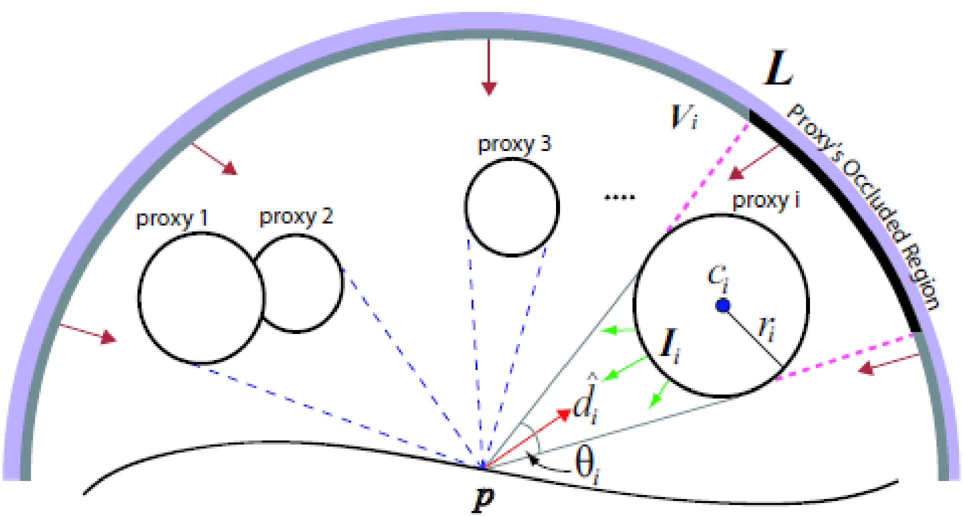
Ambient occlusion

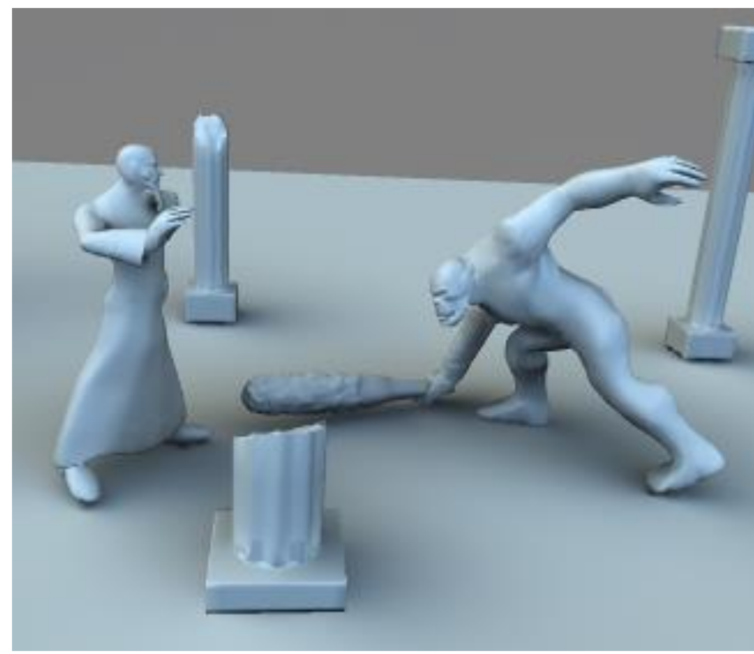
Real-time GI

- Simulating indirect lighting with high number of small direct lights
- Using deferred rendering

Real-time GI

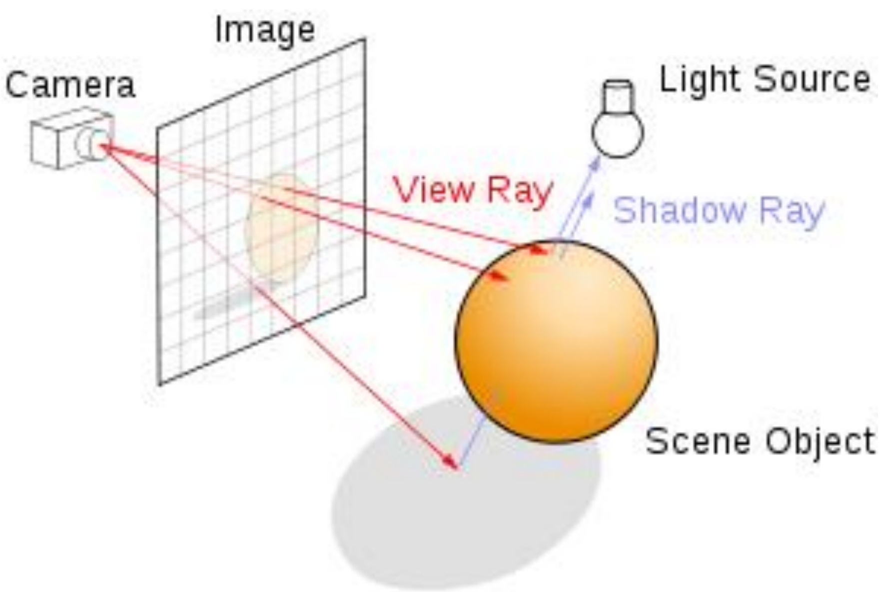
- Sloan at al.
- Using spherical proxies – simplification of dynamical geometry





Ray tracing

- Backtracking ray that comes to eye
- On surface – multiple bounces – Monte Carlo



Ray tracing

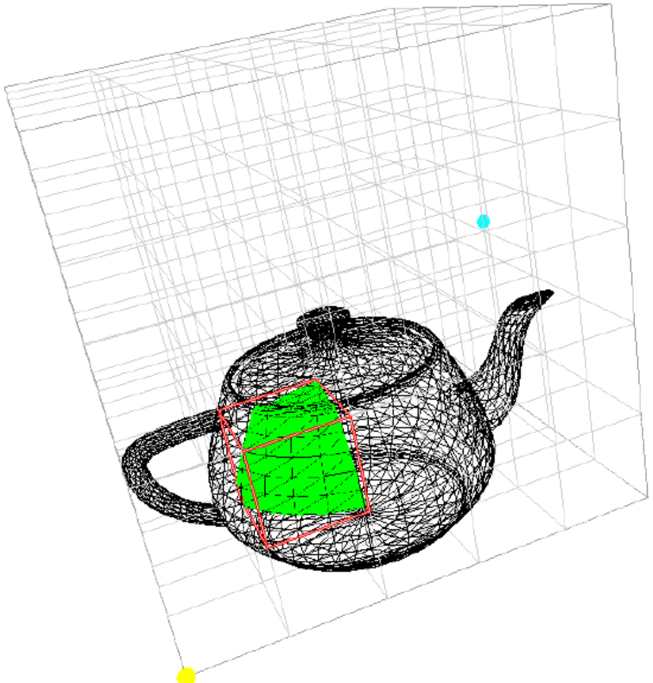
- Crucial – intersection of ray and object in scene
- Intersection speed up
  - Data structures – BVH trees, uniform grids, kD trees, octrees, …
  - Efficient algorithms
- Local illumination in intersection
- Handle absorption, reflection, refraction and fluorescence in intersection

Ray tracing

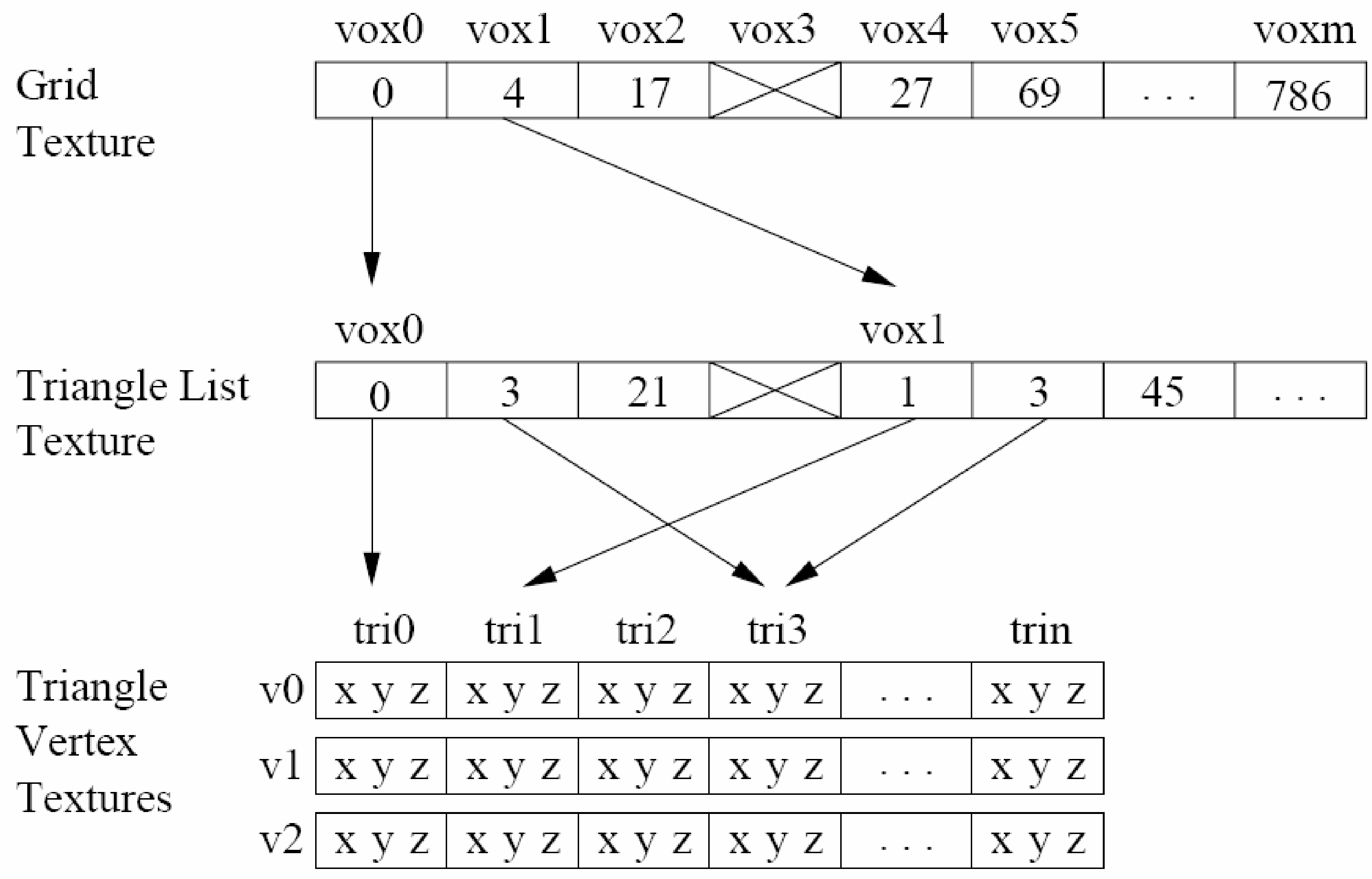
- Traverse acceleration structure
  - Cull away parts that ray cannot hit
  - Leaf nodes contain primitives
- Primitive intersection
  - Intersect ray directly
  - Return hit status to traversal
- Generate secondary rays from hit

Uniform grid

- GPU friendly → 3D texture
- Dependent fetches for lookup
- Each voxel → several primitives (parts)
- Precomputed on CPU
- Static scene
- Resolution?



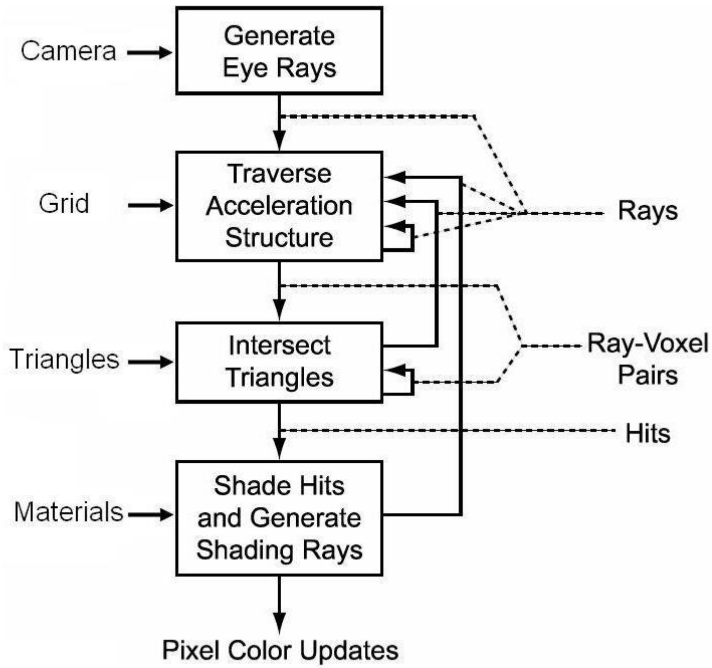
Uniform grid - GPU

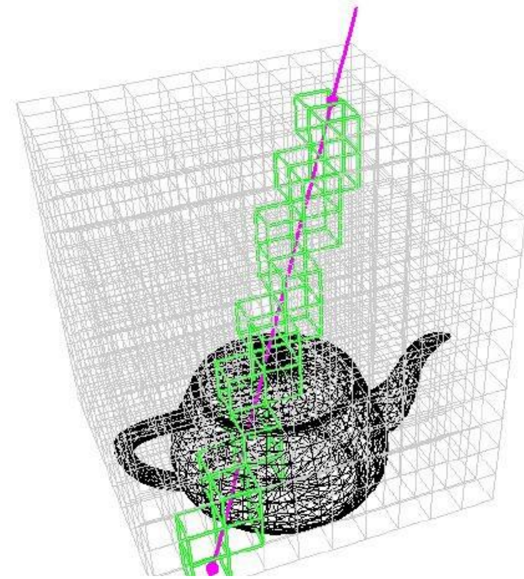


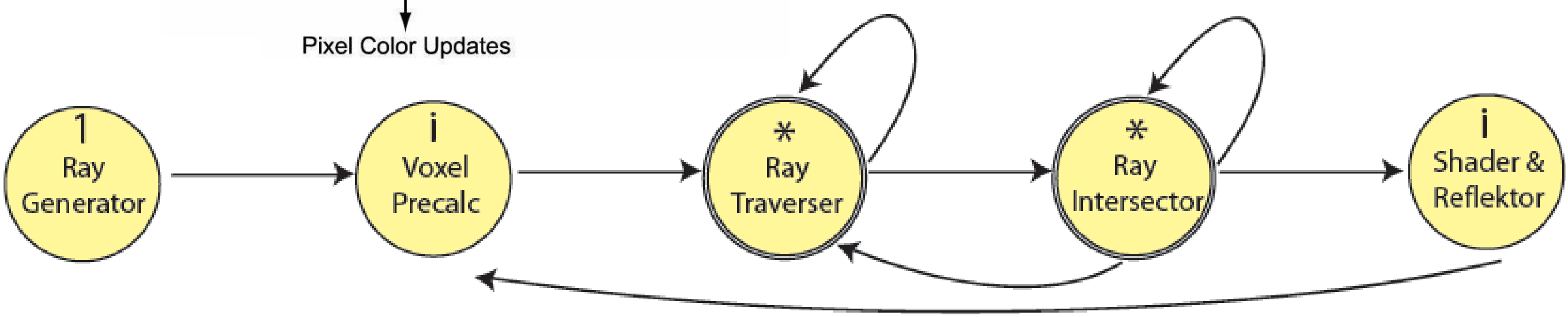
GPU ray tracing

- Rendering quads with screen size
- Using fragment shaders for computation
- Storing data in textures
- Textures for rays, intersections,
- Using 3D DDA for uniform grid traversal
- Computing exact intersection of ray and triangle
- http://www.clockworkcoders.com/oglsl/rt/

GPU ray tacing

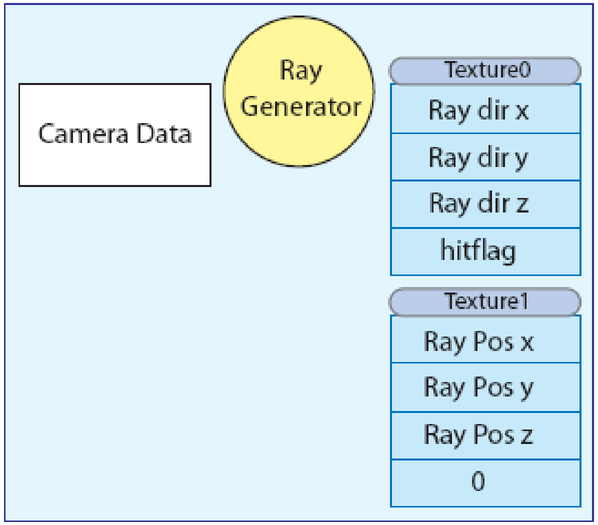






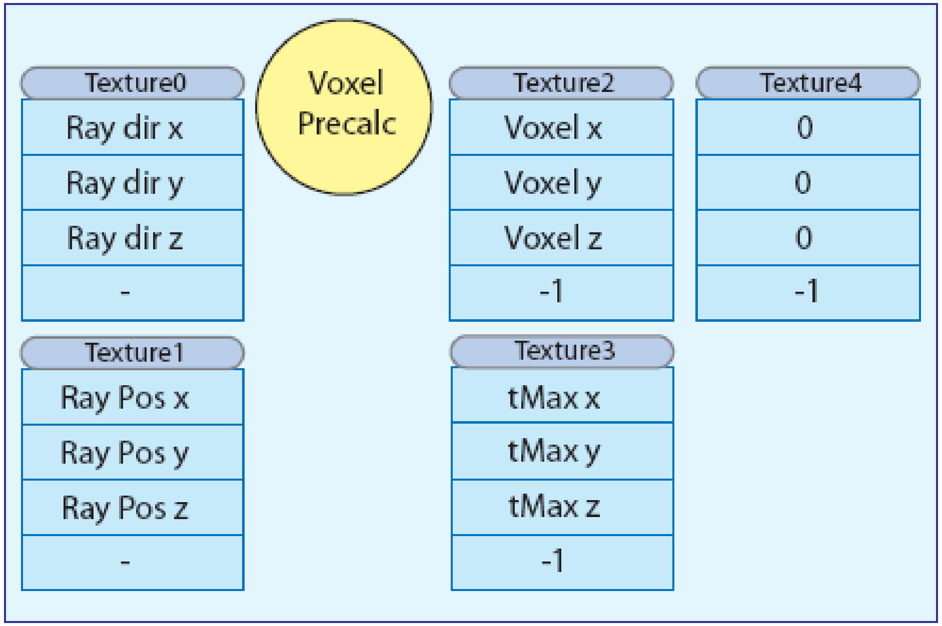
Ray generation

- Generation of ray parameters for each pixel
- In – camera data
- Out – ray starting point and direction, flag if ray hit bounding box of scene



Voxel Precalc

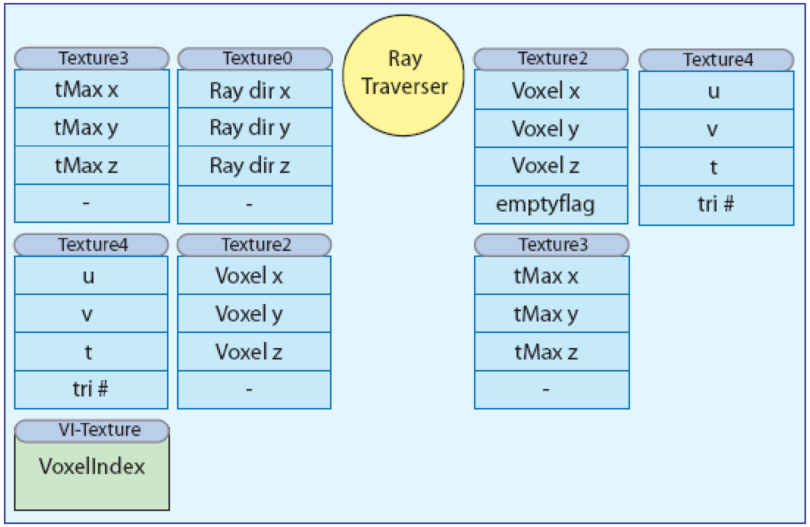
- First voxel of grid along ray
- In – ray vector, position (world coord)
- Out – in/out position (grid coord)



Ray Traverser

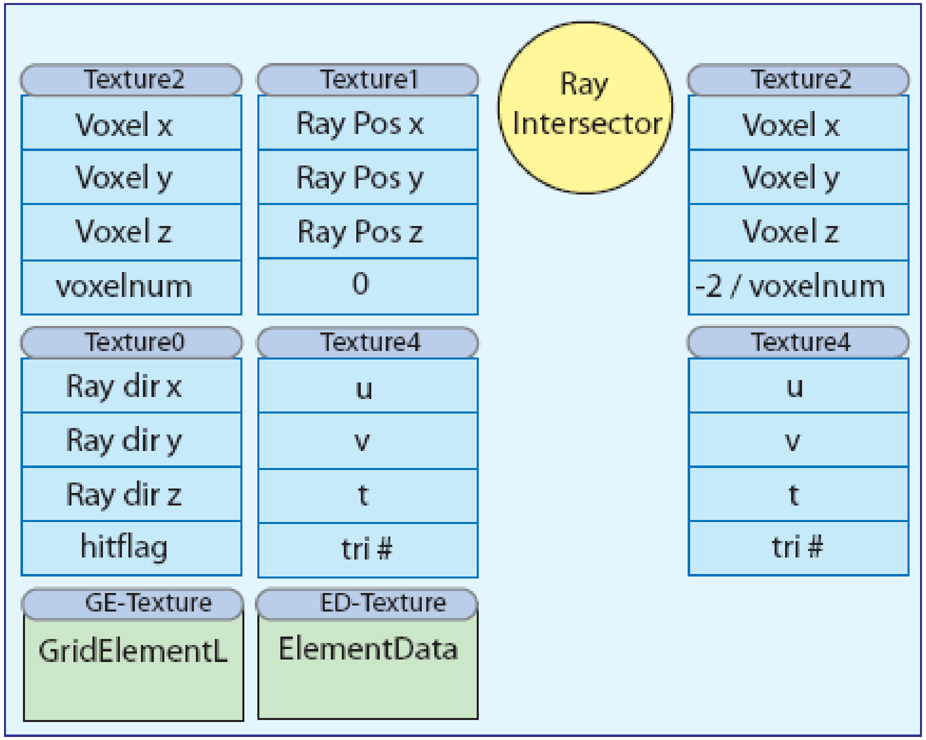
- Traversing grid along ray, setting state of ray based on voxel position and voxel triangles
- In – current voxel
- Out – next voxel

active	traverse Grid
wait	ready to check intersections
dead	ray doesn't hit grid (was already rejected in voxel precalculation)
inactive	a valid hit point was found
overflow	traversal left voxel space (no valid hits)

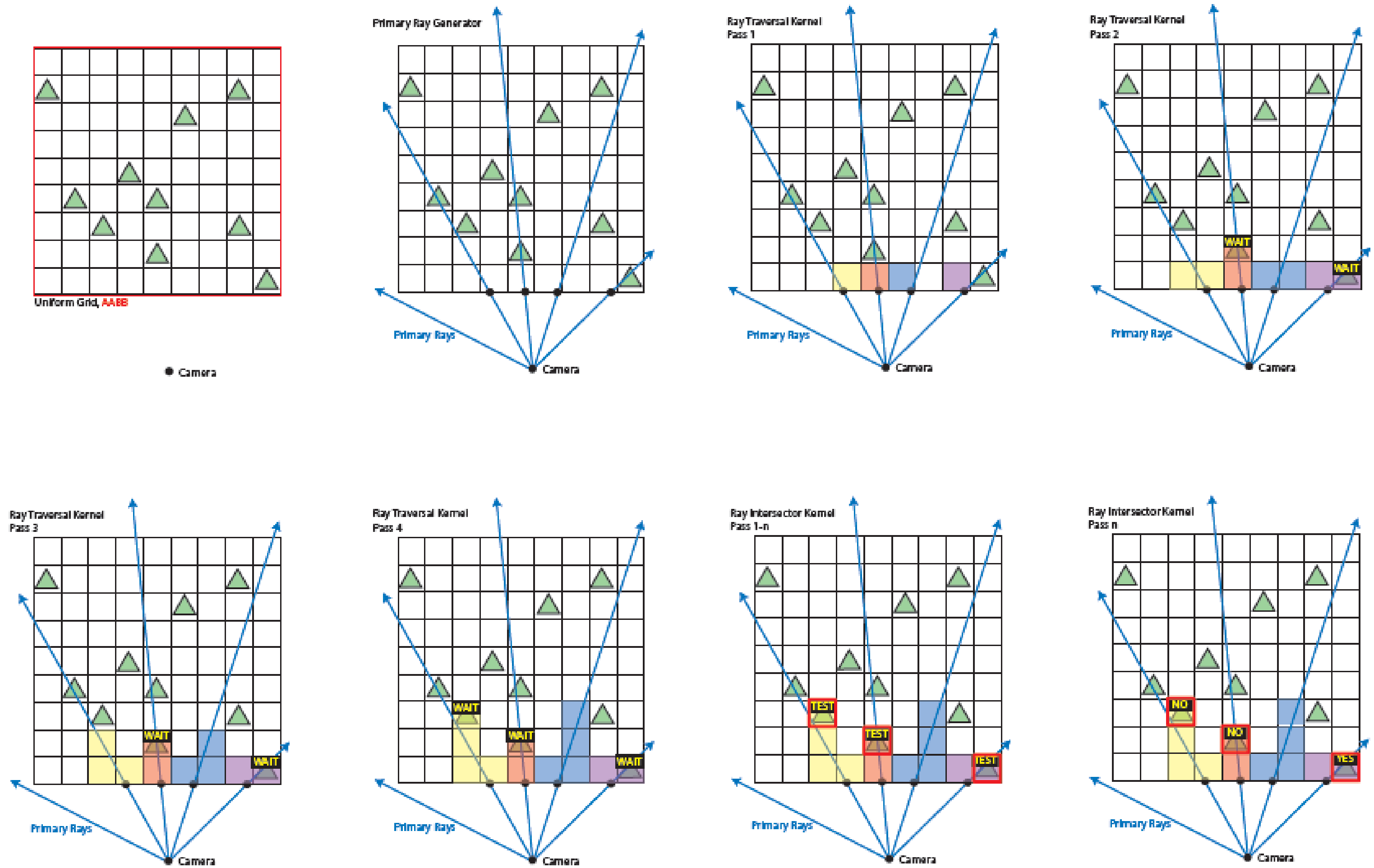


Ray Intersector

- For wait-state rays, computing intersection of ray and triangles in current ray’s voxel
- In – current voxel
- Out – intersection



Loop


Uniform Grid, AABB
Camera
Primary Ray Generator
Primary Rays
Camera
Ray Traversal Kernel
Pass 1
Primary Rays
Camera
Ray Traversal Kernel
Pass 2
WAIT
WAIT
Primary Rays
Camera
Ray Traversal Kernel
Pass 3
WAIT
WAIT
Primary Rays
Camera
Ray Traversal Kernel
Pass 4
WAIT
WAIT
WAIT
Primary Rays
Camera
Ray Intersector Kernel
Pass 1-n
TEST
TEST
TEST
Primary Rays
Camera
Ray Intersector Kernel
Pass n
NO
NO
YES
Primary Rays
Camera

GPU ray tracing

Other GI sources

- Using GPGPU capabilities, CUDA, OpenCL
- Path tracing
  - http://igad.nhtv.nl/~bikker/ (CPU)
- Ray tracing:
  - http://www.nvidia.co.uk/object/optix_uk.html
  - http://graphics.stanford.edu/papers/i3dkdtree
  - http://graphics.cs.uiuc.edu/geomrt/
  - http://www.mpi-inf.mpg.de/~guenther/BVHonGPU/index.html

Questions?