Real-Time Global Illumination for Dynamic Scenes

Screen Space Techniques

Carsten Dachsbacher
Visualization Research Center
University of Stuttgart

Jan Kautz
University College London

Overview

- What are screen space techniques?
- Screen Space Ambient Occlusion
- Screen Space Directional Occlusion
  - indirect illumination from nearby geometry
- Reflective Shadow Maps
  - indirect illumination using gathering and shooting
  - multi-resolution splatting
- Conclusions, Limitations

In this part of the course we will be talking about a class of recently popular methods to approximate global illumination, or at least produce fake GI effects.

These techniques have in common that they operate in screen space, and thus they are easy to implement, and very GPU-friendly.

We will start with a brief introduction to screen space techniques in general, and will then start with our overview over popular approaches for

computing ambient occlusion, and indirection illumination at interactive or real-time frame rates.

Screen Space Techniques

- screen space or image space techniques
  - use information obtained from a rasterization pass
  - fragment depth, position, normal, …
  - closely related to deferred shading [Deering]
  - enable a variety of (lighting) effects
  - compute output based on these attributes and pixel neighborhoods

A screen space, or image space technique, only takes information into account which can be obtained from a rasterization pass that takes place before computing the global illumination approximation.

As such, this information comprises fragment depths, positions, normals, tangent spaces etc.

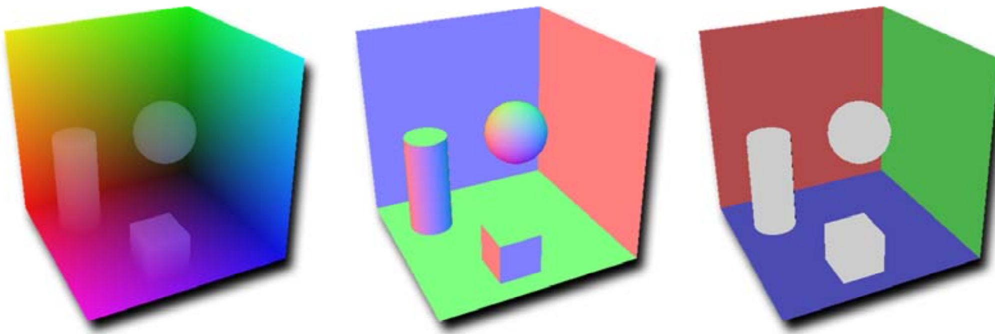
Obviously it is closely related to deferred shading, which is the forefather of image space lighting techniques.

Based only on screen space data, such techniques then can compute a variety of illumination effects.

In most cases, these algorithms do not only consider a single pixel, but also the neighborhood of a pixel to compute its shading.

Deferred Shading

- for each pixel store surface data that is required for lighting computation
  - deferred shading buffers: position, normal, material
- use pixel/fragment shaders for lighting computation
  - render textured quadrilateral
  - good for complex shaders, many small light sources



I just mentioned deferred shading, which is an important basis for the techniques that I will describe in the following.

As its name says, the shading of the surfaces is not immediately done when rasterizing the geometry.

Instead we render the information required for the lighting computation, that is position, normals and material parameters, into textures and we defer the lighting computation to a later render pass.

So after rendering the scene geometry, we render a quadrilaterial covering the entire screen with the deferred shading textures mapped onto it.

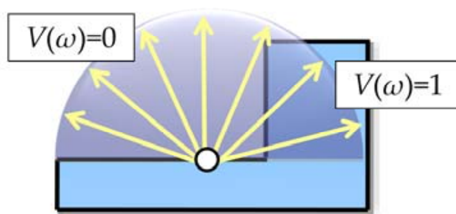
A fragment shader reads the surface information from the textures and computes the lighting.

Deferred shading is particularly good for rendering with very costly shaders, and in scenes with many small light sources.

Ambient Occlusion

- soft shadows by darkening surfaces that are partially visible to the environment
- compute the surface accessibility
  - used to modulate lighting
  - portion of the hemisphere above each surface point not occluded by geometry

$$A = 1 - \frac{1}{\pi}\int_{\Omega^+} V(\omega) W(\omega) (\omega \cdot \mathbf{n}) d\omega$$



The by far most popular technique for „global“ lighting effects that can be computed in screen space – I intentionally said effect – is ambient occlusion.

Ambient occlusion provides soft shadowing of surfaces, by computing how much light from the environment would reach a surface point, which is also called accessibility.

This scalar values is all that we need to compute for AO.

In principle – we would evaluate the hemispherical integral over a surface point, and test whether another surface is blocking light from the environment using the visibility function V which is 1 if there‘s no blocking surface, and 0 otherwise.

The straightforward way, also used in an offline AO computation is to compute the visibility function is using ray tracing.

The W(omega) term is an optional attenuation function, the reduce the blocking impact on the accessibility value for distant surfaces.

And finally, depending on the AO implementation, there‘s also the cosine of the angle between the light direction and the surface normal.

Obviously, approximating the integral with Monte-Carlo integration and ray tracing is not the way to go for real-time rendering.

Screen Space Ambient Occlusion

- depth buffer = discrete approximation of the scene
- compute accessibility from the pixel neighborhood
- Crytek's method [Mittring07][Kajalin 2008]
  - use random samples inside sphere

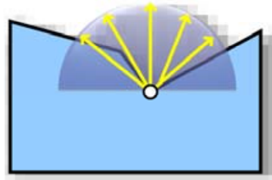


Image: Crytek GmbH

A clever observation is that the depth buffer of a rendered scene is nothing else but a discrete approximation of the scene geometry.

It does not capture all surfaces, only the front most, but this is sufficient for many GI effects.

All screen space AO techniques have in common that they rely on this information to compute the accessibility of a surface.

The most well-known implementation, where the term SSAO probably stems from, is Crytek's method.

Instead of tracing rays, they generate a set of random sample points in a sphere centered around the surface point in question.

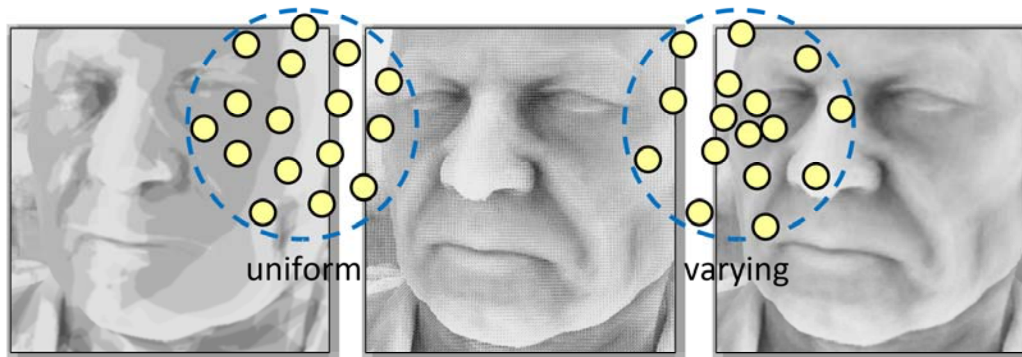
The accessibility is then estimated from the ratio of sample points closer to the camera than the surface, to the total number of sample points.

This is very simple to compute and properly darkens concave regions, but convex edges appear lighter than planar surfaces.

Despite this problem, it apparently mimics the real accessibility quite well and is acceptable in many applications.

Screen Space Ambient Occlusion

- Crytek's method [Mittring07][Kajalin 2008]
  - distance attenuation: varying sampling density
  - taking few samples of a fixed pattern results in banding
  - randomly rotating the pattern results in noise
  - geometry-aware filter as a post-process



Images: Crytek GmbH

Distance attenuation, i.e. reducing the blocking of distant geometry, can be easily integrated by generating more sample points closer to the center of the sphere, and less further away.

To keep the computation cost low, only a small number of sample points (~16) is used in their implementation.

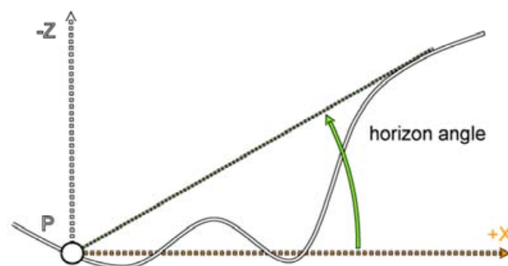
When taking only few samples of a fixed pattern the results will exhibit banding artifacts.

A common solution to this is to trade noise for banding, and this is done by using a single sampling pattern, but randomly rotating it per pixel (actually they use 16 different orientations in a 4x4 block).

The noise is then reduced by applying a geometry-aware filter as a post-process.

Screen Space Ambient Occlusion

- Horizon-Based Ambient Occlusion [Bavoil08]
  - depth buffer = height field
  - cast rays into oriented hemisphere (per-pixel normals)
  - step along rays to determine horizon angles
  - expensive, but high-quality results



A more costly method, but also providing higher quality, has been described by Louis Bavoil and Miguel Sainz.

The starting point is almost the same: this method relys on the depth buffer, but also on per-pixel normals of the image.

However, they are actually pursuing the monte-carlo approach and they cast rays into the hemisphere above a surface (thus the per-pixel normals) and step along them in order to determine whether a ray intersects a surface.

The goal of the ray casting is to determine the horizon angles of the surrounding of a surface point, and then compute the accessibility from it.

Although this is much more costly to compute, the method still runs at interactive frame rates and produces much nicer results.

Just as a side note, this method still relies on a blur filter to reduce noise.

Screen Space Ambient Occlusion

- ⊕ coarse approximation, but plausible
- ⊕ no precomputation, fully dynamic scenes
- ⊕ simple implementation
- ⊕ independent of scene complexity
- ⊕ real-time on recent GPUs
- ⊖ no global illumination!
- many variants, e.g.
  - Luft et al., *Image Enhancement by Unsharp Masking the Depth Buffer,* ACM Transactions on Graphics (Proc. of SIGGRAPH), 2006
  - Shanmugam et al., *Hardware Accelerated Ambient Occlusion Techniques on GPUs*, Symposium on Interactive 3D Graphics and Games (I3D) 2007
  - Filion et al., *StarCraft II Effects & Techniques*, Advances in Real-Time Rendering in 3D Graphics and Games Course, SIGGRAPH 2008

The advantages of SSAO techniques are pretty obvious.

Although most of them – this slide lists a few more – compute coarse approximations only, the results are plausible and sufficient in many cases.

All of them require no precomputation at all, and fully support dynamic scenes.

Most SSAO techniques are simple to implement, typically it only requires few additional render passes with rather simple shaders.

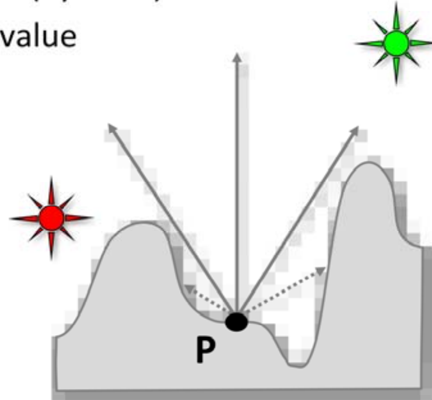
Since they work in screen space their cost is also independent from the scene complexity.

They are also fast enough to be recomputed every frame and thus they are the preferred way to get AO effects in real-time nowadays.

However, although AO looks really cute, this is no global illumination – but we get closer in the next minutes…

Screen Space Directional Occlusion

- SSAO: compute accessibility and lighting independently, directionally-varying light is ignored
- example
  - surface at P should be green: only red light is blocked
  - AO computes illumination first (=yellow)
  - then scales by a single scalar value
  - → surface at P is brown



With SSAO techniques we compute the accessibility and the lighting of a surface independently.

After having computed both, we modulate the lighting or surface color by the AO term.

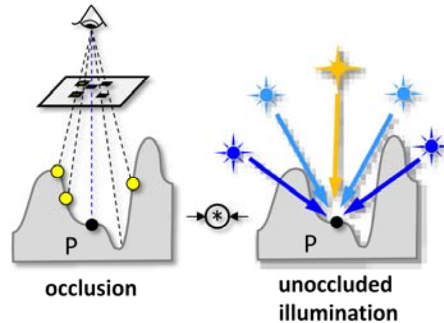
The example on the slide illustrates a scenario where this procedure generates wrong results.

In this setting there are two light sources, a red and a green one, and the surface at P should be greenish, as the red light is blocked.

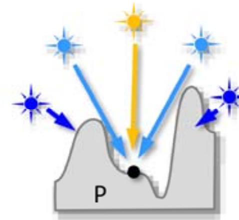
With AO, we first compute the lighting, which would be yellowish here, then scale it using the scalar AO term, and end up with a brown, instead of green surface.

Screen Space Directional Occlusion

- ambient occlusion
  - when looping through neighbor pixels, why not integrate the incoming radiance for each direction?
  - [Ritschel et al. 2009]



- screen space directional occlusion (SSDO)
  - approximate visibility for each sample
  - accumulate only non-blocked illumination



In Ritschel et al's paper at this year's I3D a simple yet practical observation is described:

If we recap, what we're actually doing with SSAO is, that we sample a pixel's neighborhood, determine the location of the surfaces, and estimate the accessibility which is then multiplied with the illumination.

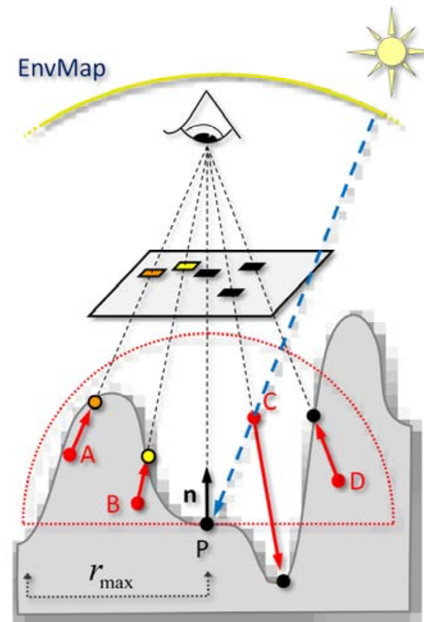
Sampling the neighborhood is the major cost in SSAO and a practical and clever observation is:

why only compute the blocking, why not immediately integrate the incident radiance?

This procedure, called SSDO, does not only approximate visibility, but also provides directional radiance information by effectively approximating the integral over visibility times radiance, e.g. taken from an environment map.

SSDO Visibility Test

- compute N samples (A – D) in the upper hemisphere of P with user-defined radius (typically N = 8..16)
- for each sample
  - backproject sample to image
  - look-up surface position
  - if surface is closer:
    sample is occluded (A,B,D)
  - so far similar to Crytek's method
  - otherwise: P is illuminated from this direction (C)



The rendering procedure in SSDO works as follows:

In the upper hemisphere of a surface point to be lit, we choose a number of random sample points.

Project each of them into image space, and then test the computed depth value of the sample against the depth stored in the z-buffer.

This is very similar to Crytek's SSAO.

The difference is however, that every sample that is not occluded, immediately contributes to the incident radiance.

Directional Occlusion

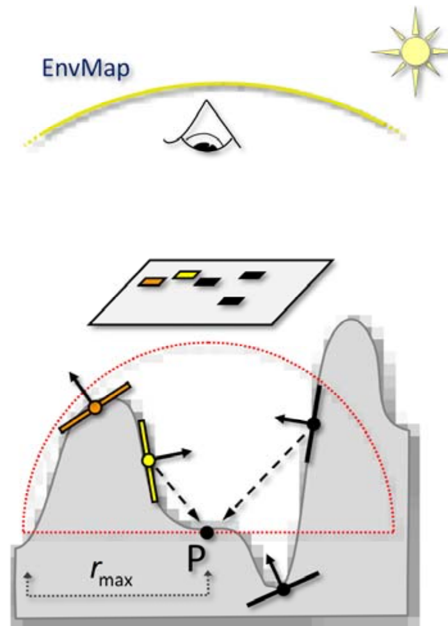
- SSDO shadows: oriented and colored
- SSAO shadows: no orientation, gray
- only little overhead of 2%-9% [Ritschel et al. 2009]

This image shows a SSDO rendering. In contrast to SSAO you can see oriented and colored shadowing from the surface details.

Ritschel reported a comfortable small overhead of SSDO compared to SSAO of only a few percent render time depending on the resolution and number of samples.

Indirect Illumination

- every pixel is a small surface patch reflecting light
  - oriented around pixel normal
  - compute reflected radiance from direct lighting
- accumulate contributions from neighboring pixels
- one indirect bounce from nearby geometry



Now this can be even taken one step further:

Each pixel in the image represents a small surface that reflects light.

That is, every surface sample in the neighborhood can be considered as a small reflector, for which we compute the reflected radiance and

then accumulate the lighting from those pixels by computing a form factor approximation, effectively yielding one bounce indirect illumination from (only) nearby surfaces.

Indirect Illumination

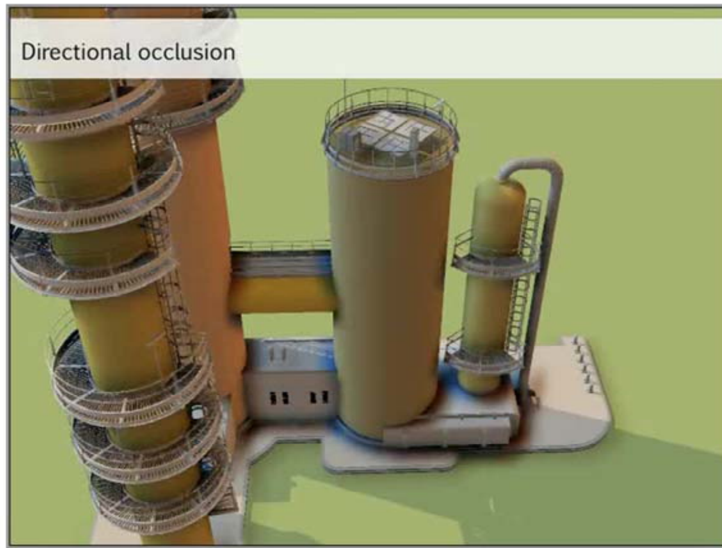
- direct lighting computed using prefiltered environment maps or point lights
- good temporal coherence (many nearby samples)
- overhead: ~30%

This video shows an example, where the direct lighting is computed from environment maps and/or point lights and the indirect lighting has been computed in screen space.

The high sampling density in a pixel neighborhood provides a good temporal coherence and plausible color bleeding.

Again, this is achieved with a comparatively small overhead of 30% against SSAO.

Complex Scene



GeForce 8800 GTX

Resolution:	1024 x 768
Polys:	~300k
SSDO:	55 – 80 fps
Indirect Light:	40 – 65 fps

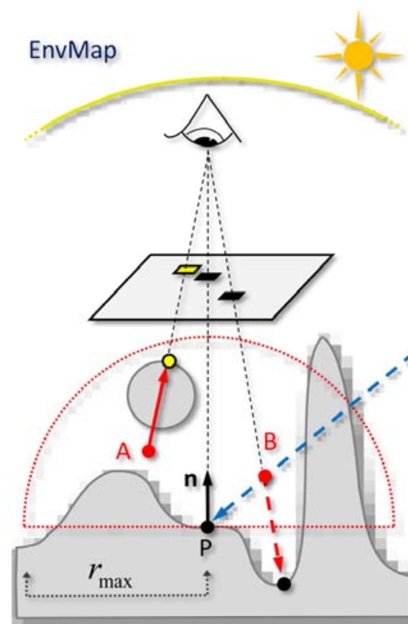
3D Model from
Dosch Design

This allows rendering of complex scenes, at high-resolution at real-time frame rates.

However, if you watch closely, then you will be able to identify halo artifacts and notice that indirect illumination has a relatively local effect only, but this is to be expected with such rendering techniques.

SSDO – Limitations

- approximate visibility due to missing information
  - wrong visibility
    - sample A → occluder
    - sample B → visible
  - results in halos
  - depth peeling? ray marching?
- only nearby geometry: no occluders outside sphere



The reasons for the limitations of SSDO are common to all other screen space techniques.

The visibility for the incident radiance obtained from the depth buffer is approximate, and can lead to wrong results, as the depth buffer only captures the first visible surface.

The slide shows two such wrong classifications that in principle can be resolved by using depth peeling and ray marching, however, at a significantly increased rendering cost of course.

And as just mentioned, only contribution from surfaces that are nearby is considered, this is simply to limit the number of texture accesses and to keep the method at interactive speed.

Reflective Shadow Maps

- basic approach to screen space indirect illumination
  - motivation: one-bounce indirect illumination “officially” approved to be sufficient
  - *An Approximate Global Illumination System for Computer Generated Films*, Tabellion and Lamorlette 2004
- basic idea: quickly retrieve information about surfaces creating indirect illumination

After SSAO and SSDO which operator very locally in screen space we will talk about one of the older or classic techniques for computing indirect illumination in screen space.

We’ll start with the basic principles first and then briefly discuss recent improvements.

The original motivation for this work was that one-bounce indirect illumination is sufficient in many cases, and e.g. it has been sufficient for offline file production rendering as well.

For example, Tabellion and Lamorlette created a texture atlas for a scene containing direct lighting and when rendering the final image, they gathered the first bounce of indirect light

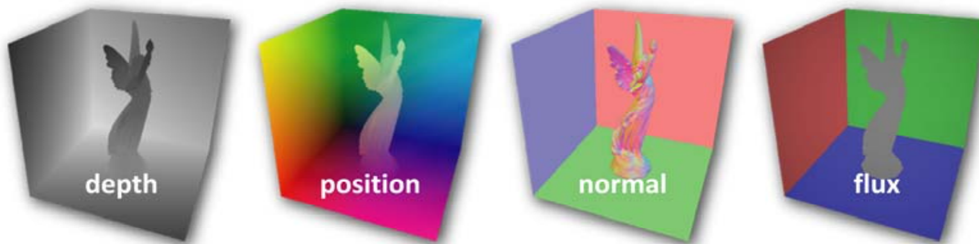
with raytracing from this texture.

For real-time rendering, the tricky part is to quickly retrieve the information about those surfaces that create indirect illumination.

One possibility to do this are texture atlases plus ray tracing, another one are the so-called reflective shadow maps…

Reflective Shadow Maps (RSM)

- rasterize scene from light source
  - *extended shadow map* captures directly lit surfaces
  - reflection off these surfaces generates indirect light
    → reflective shadow map
  - screen space = shadow map projection
- store for every pixel
  - depth, position, normal, and reflected radiant flux



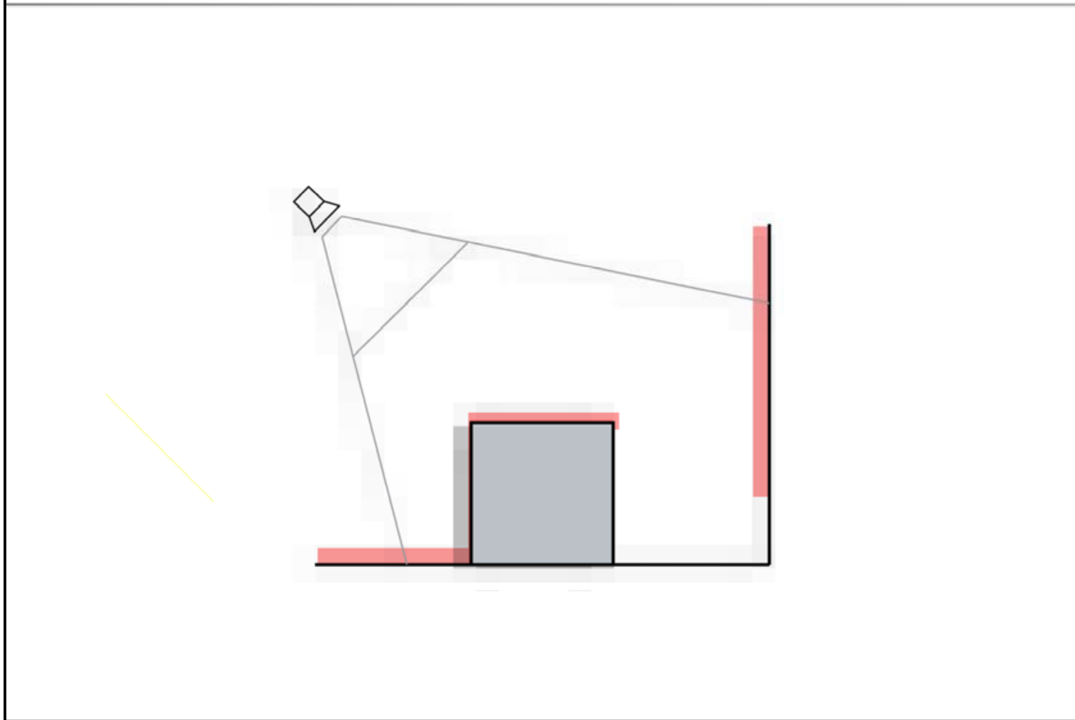
As a normal shadow map, a reflective shadow map rasterizes the scene, as seen from the light source.

The observation is: all surfaces that are visible in a shadow map are directly lit, and thus only these surfaces generate indirect illumination.

In addition to the depth value such an extended shadow map also stores the world space position, the surface normal and the reflected radiant flux for every pixel, in order to compute the indirect lighting off these surfaces.

The position tells use where the indirect light is starting from, the flux defines the brightness and color, and the normal defines the spatial emission characteristics.

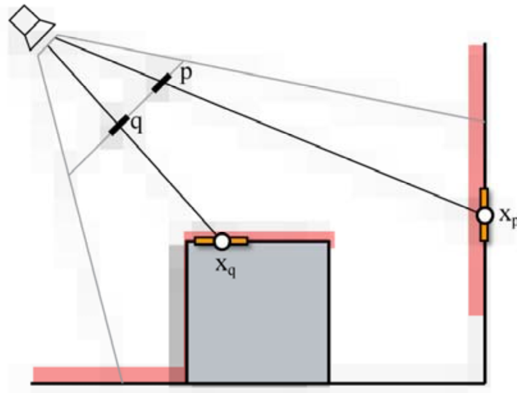
Of course the position can be reconstructed from light source positioning and the depth value, so it’s an implementation choice whether to store or to compute this.



Here you can see a simple setting with a spot light, and a simple geometry… for which we compute a reflective shadow map

Reflective Shadow Maps (RSM)

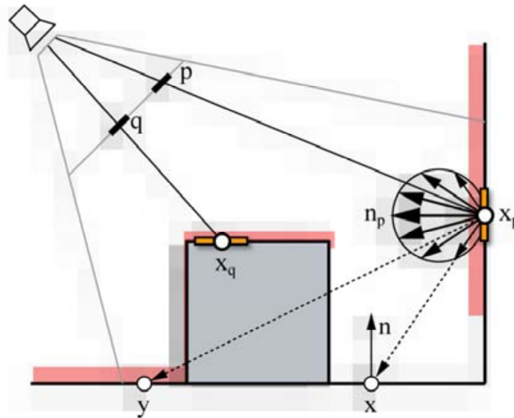
- each pixel is a small light source



And if we take two pixels of that RSM, then we can interpret the piece of geometry that they represent as a small light source again.

Reflective Shadow Maps (RSM)

- each pixel is a small light source
- sum up contributions $E(x,n) = \sum_{\text{pixels}\, p} E_p(x,n)$



⇒ no occlusion or ray marcing

And this means, we can approximate the total indirect irradiance at a surface point by summing up the illumination due to all pixel lights.

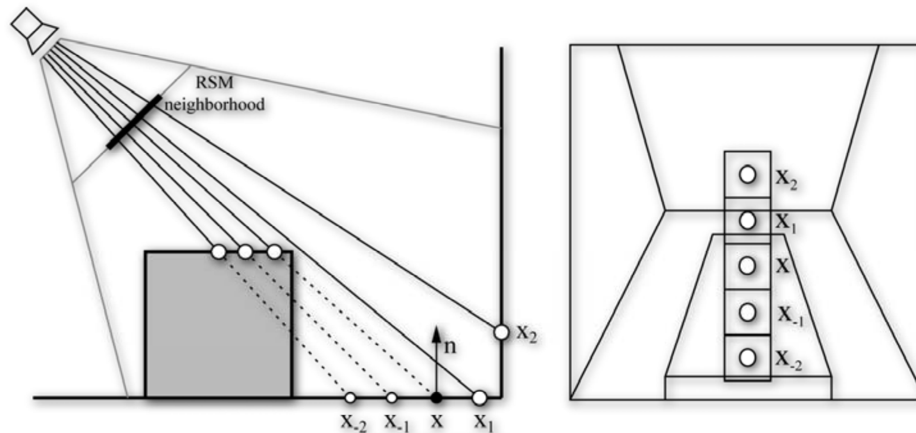
Of course, only the front most surface seen from the light source is captured, i.e. we can either try to compute occlusion for indirect lighting by ray marching, or simple ignore it.

The latter has done in the original approach. Of course, this is a severe approximation, and can lead to wrong results.

However, in many cases it is better to have indirect lighting effects at all, accept imprecise results, and maybe compensate with AO.

Gathering

- gather from RSM: too many pixel lights
  - concentrate on potentially important samples
  - pixels close in world space are close in RSM
  - not vice versa (depth!)



In principle we can approximate the indirect illumination by looping over all pixels in the RSM, but for a typical shadow map resolution the number of pixels if way too large to consider all of them for computing indirect illumination.

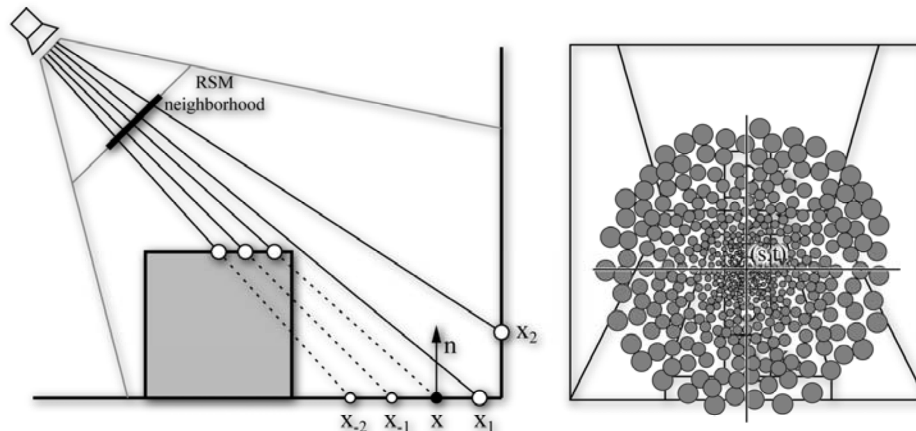
Instead we reduce the sum to a restricted number of a few hundred samples, concentrating on the most relevant pixel lights.

In general those pixel lights are relevant which are close in world space, because of the distance attentuation.

And if they are close in world space, they are also close in the shadow map projection – in reverse this is of course not true.

Gathering

- gather from RSM: too many pixel lights
  - concentrate on potentially important samples
  - pixels close in world space are close in RSM
  - not vice versa (depth!)

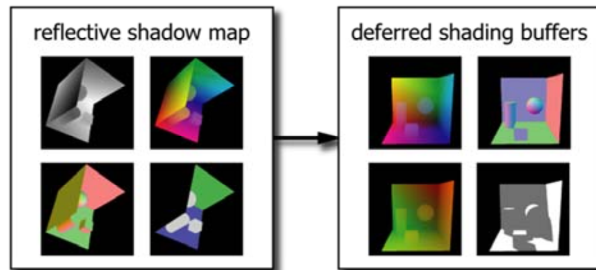


For this, we precompute a sampling pattern with a sampling density decreasing with the distance from the center for selecting pixel lights.

And we center it around the projection of the surface point for which we want to compute the indirect lighting.

And by this obtain the few hundred pixel lights that we want to consider for the computation of the indirect lighting.

Rendering with RSMs



The generation of the final image of this basic RSM technique consists of multiple render passes.

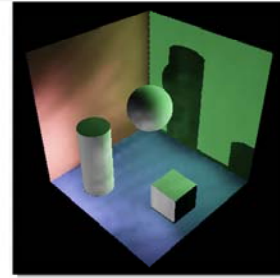
First the RSM and the deferred shading buffers are rendered.

The next step is the gathering of the indirect illumination, which can be done for a sub-sampled low-resolution image first, since indirect illumination is often very smooth.

And finally this sub-sampled indirect light is interpolated where possible, and additional computations of indirect illumination are spawned where no interpolation is possible, which is the case for discontinuities.

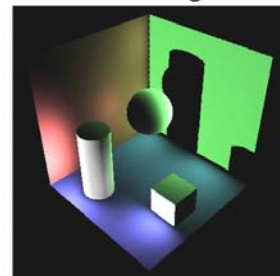
Gathering vs. Shooting

- gathering indirect light
  - many incoherent texture look-ups
  - noise, sampling artifacts
  - non-diffuse reflectors?
  - omni-directional light sources?



gathering

shooting



- shooting approach
  - create/select a set of light sources
  - accumulate contributions

Unfortunately this gathering procedure has several disadvantages, such as many incoherent texture lookups that reduce the rendering performance, artifacts due to the limited number of samples, and the sampling patterns for non-diffuse reflectors and other types of light sources are problematic as well.

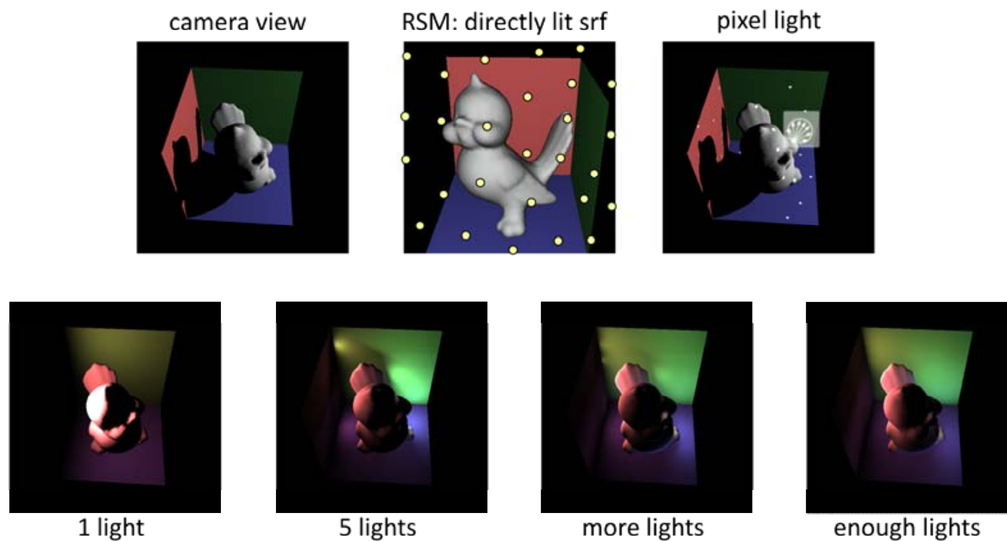
If there’s gathering, then there’s also a shooting approach:

The idea here is to select a fixed set of indirect light sources each frame and then compute and accumulate their contribution to the scene.

And at this point, we‘re getting closer to instant radiosity approaches which will be dicussed later in this course

Shooting

- idea: select a subset of RSM pixel lights
  → virtual point lights (VPLs)



This slide summarizes the idea of the shooting approach: in order to add indirect lighting to a scene, we capture the directly lit surfaces with the RSM, and select some pixels from it.

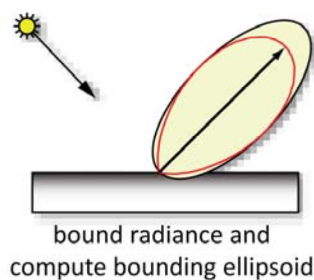
Then we accumulate the contributions from all those pixel light sources using deferred shading, which is perfectly suited for this task, as we will use several hundred light sources (also often called virtual point lights or VPLs).

And the more light sources we use, the smoother the approximation of the indirect lighting becomes.

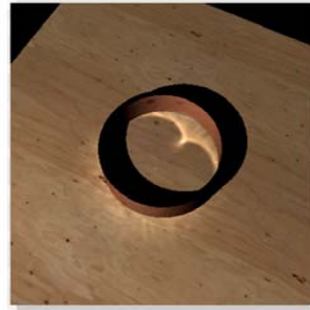
Note that in this work, there‘s still no occlusion for the indirect lighting!

Shooting

- optimizations
  - only significant nearby light source ~$1/d^2$
  - choose radiance threshold to cut-off light contributions
  - 2D AABB or bounding volume of significant region
  - place more VPLs on glossy surfaces
  - no CPU load



bound radiance and compute bounding ellipsoid



Restricting the lighting computation to a certain area of the screen is very simple with deferred shading and comes handy in this case.

The fall-off of a point light source is 1 over d^2 and thus, we can choose a lower threshold of the radiance to cut-off the VPL contributions to the image.

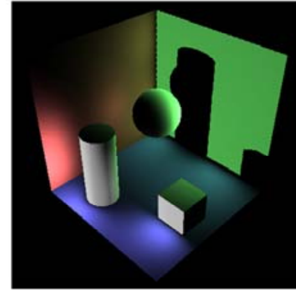
This is allows us to compute a 2D bounding rectangle, or a 3D bounding volume to restrict the deferred shading to regions where a VPL has a significant contribution only.

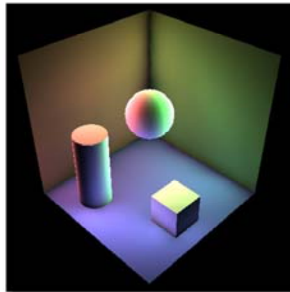
Further possible improvements to this method are importance sampling to place more light sources on glossy surfaces which allows us to render caustics from a metal ring for example.

All these optimizations have been implemented on the GPU, i.e. the CPU is completely available for other tasks.

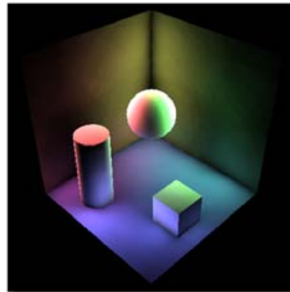
Clamping Artifacts

- impact of radiance threshold: long-range light transport vanishes

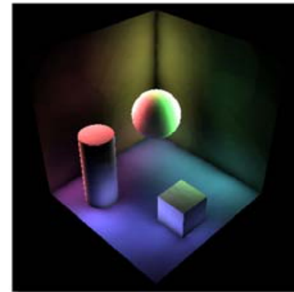




$I=0.01I_0$



$I=0.05I_0$



$I=0.07I_0$

These images here show exaggerated indirect lighting only to demonstrate the effect of cutting off the light sources.

With increasing thresholds – from left to right – the rendering becomes much faster, but the indirect illumination over larger distances gets lost.

Reflective Shadow Maps

- major cost:
  - gathering: lots of incoherent texture lookups
  - shooting: fill rate, hundreds of quads, up to 1M pixels each
- multi-resolution approach
  - observation: upscaled low-resolution image already contains most of the information

So both options that we've seen so far compute indirect lighting from a RSM but suffer from performance problem.

The gathering approach requires lots of incoherent texture lookups, that can only be reduced by subsampling in screenspace, which in turn causes reduced quality.

Shooting on the other hand causes significant overdraw because we splat the indirect illumination and waste fill rate.

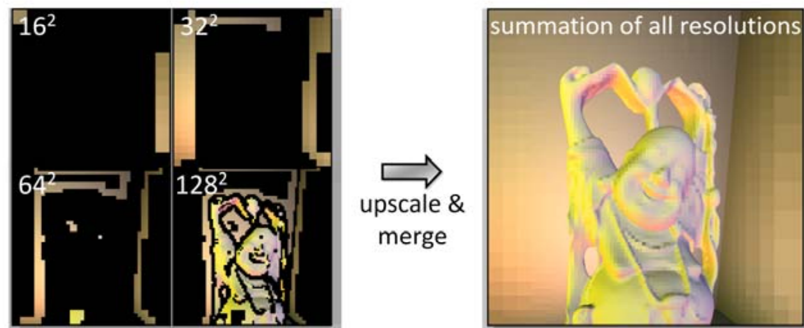
So what can we do?

If you look at the left image you can see a low-resolution image with indirect lighting which is upscaled.

As you can see, most of the information is already there, and this motivates a multi-resolution approach which has been presented recently.

Hierarchical Approach

- multi-resolution splatting [Nichols 2009]
- no clamping: VPL always affects entire screen
  - multi-resolution frame buffer
  - low frequency indirect illumination → low-res buffer
  - high frequency indirect illumination → hi-res buffer
  - always renders small splats



This approach has been presented by Nichols and Wyman at I3D this year.

The do not clamp the contribution of a VPL, i.e. each VPL always affects the entire screen, however, the maintain a multiresolution frame buffer,

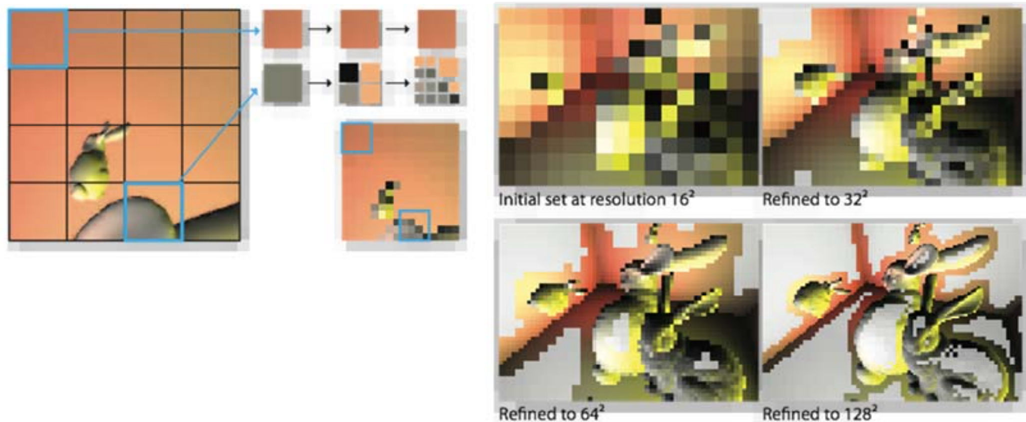
and splat low-frequency indirect illumination into low resolution buffers, and high freq. indirect illumination into high-res buffers.

An example is shown on the left images on this slide.

By this, they always render small splats, and afterwards upscale the multi-resolution levels and merge them to obtain the final rendering, which is shown on the right – here using nearest neighbor sampling for demonstration purposes.

Adaptive Refinement

- try to render VPL into low-res buffer first
- use min-max depth mipmap to detect discontinuities
- if region contains discontinuities →
  refine (split into four higher-res sub-VPLs)



In principle this method starts by trying to splat VPLs into a low-resolution buffer first.

Then the remaining problem is, how to determine if it is accurate enough to splat into the low-res buffer…

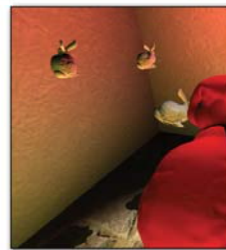
A clever trick is to compute a min-max depth mipmap of the scene. This allows to compute the depth range within an image region, and if this depth range is above a certain threshold, then there are probably discontinuities or steep surfaces, and then the algorithm decides to go to the next higher resolution, where 4 VPLs are created and tested for rendering.

By this, the VPLs itself are refined, and splatting across discontinuities is avoided.

This method is probably the fastest way to compute indirect illumination from a RSM, but it has to be said that it still does not consider blocking for the indirect light.

Summary

- screen-space ambient occlusion (SSAO)
  - **widely used, flexible, easy to implement**
- screen-space directional occlusion (SSDO)
  - directional visibility
  - one-bounce **indirect light of nearby geometry**
  - **little additional cost** compared to SSAO
  - similar to a RSM with a single image space
  - suitable for games (also see ShaderX7)
- reflective shadow maps
  - used to sample directly lit surfaces
  - one-bounce **indirect light without occlusion**
  - hierarchical approach is very efficient
  - can be used for creating VPLs on GPUs



To summarize the methods in this chapter.

SSAO techniques are widely used nowadays, they are flexible, easy to implement, and absolutely feasible for real-time applications.

SSDO is an interesting add-on extending SSAO by directional visibility and indirect illumination from nearby geometry at little additional cost.

SSDO is definitely a technique that is interesting for upcoming games, if not already used in some form (there are similar approaches, e.g. see ShaderX7)

The RSM are quite old already, and should be considered as a basis for many SS techniques. The original approaches are too expensive for games, but the recent improvements make them a perfect candidate.

One final remark about RSMs. The underlying concept of sampling directly lit surfaces provides means to quickly generate VPLs directly on the GPU. This has been used in the Imperfect Shadow Map paper for example, but Jan will talk about this in the next minutes.