ECSE 446/546: Realistic/Advanced Image SynthesisAssignment 3: Ambient Occlusion

Due Wednesday, October 17th, 2018 at 11:59pm EST on myCourses
17% (ECSE 446 / COMP 598) *or* **12%** (ECSE 546)

You will begin by implementing Monte Carlo estimators for two integration problems in the offline component of this assignment: one for ambient occlusion (AO) and another for reflection occlusion (RO). For the real-time component, you will implement a screen-space ambient occlusion (SSAO) algorithm to approximate AO at interactive framerates.

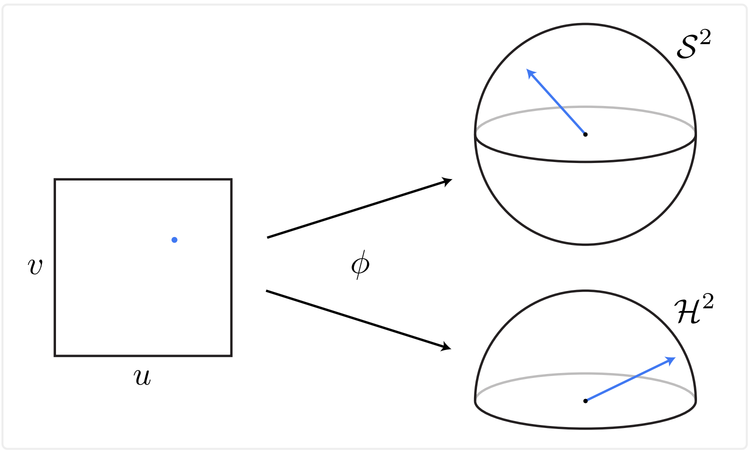
Contents

Before you begin, we recommend that you review the course material, especially slides on random sampling, Monte Carlo estimators and AO.

Afterwards, start by copying your rendering loops from A1 to `src/core/renderer.cpp` and **ensure that you initialize your random sampler** object with your student ID, e.g., `Sampler sampler(123456789)`.

1 Offline Rendering (*40 pts*)

1.1 Canonical Sample Warping for Various Distributions (*20 pts*)



You will begin by implementing methods to draw samples on several domains, and with different distributions, of interest: namely, distributions of directions over the unit sphere and hemisphere. The methods you implement here will serve as building blocks later on in this (and future) assignments.

For each such distribution, you will be responsible for implementing two routines: one to *draw* a sample proportional to the distribution, and another to *evaluate* the value of the PDF at a sample location. The `Warp` namespace in `src/core/math.h` encapsulates a set of sampling methods that expect 2D uniform canonical random variables $(u, v) \in [0, 1)^2$ as input and yield 2D (or 3D) warped samples $\phi(u, v)$ in an alternative sampling domain of interest. Corresponding PDF evaluation methods are also associated to each of these sampling routines.

There are a total of four (4) distributions we will expect you to support, and so eight (8) warping and evaluation methods you will have to implement. Each warping/PDF method pair is worth **5 points**. For more information on sample warping, we suggest you consult Chapter 13 of [PBRTe3](http://www.pbrt.org).

Uniform Spherical Direction Sampling

Implement `Warp::squareToUniformSphere()` and `Warp::squareToUniformSpherePdf()`: the former should transforms uniform 2D canonical random numbers (i.e., on the unit square) into uniform points on the surface of a *unit sphere* (centered at the origin; i.e., unit spherical directions); the latter should implements a probability density evaluation function for this warping.

Uniform Hemispherical Direction Sampling

Implement `Warp::squareToUniformHemisphere()` to transform uniform 2D canonical random numbers into uniformly distributed points on the surface of a unit hemisphere **aligned about the z-axis (0,0,1)**. Implement this function's probability density evaluation function, `Warp::squareToUniformHemispherePdf()`, too.

Cosine-weighted Hemispherical Direction Sampling

Next, implement `Warp::squareToCosineHemisphere()` to transform 2D canonical random numbers to directions distributed on the unit hemisphere (aligned about the z-axis) according to a cosine-weighted density. Then implement this PDF's evaluation method, `Warp::squareToCosineHemispherePdf()`.

Here, you may wish to implement `Warp::squareToUniformDiskConcentric()` first to help you for this task, if you choose to apply Nusselt's Analog.

Phong Cosine-power Lobe Hemispherical Direction Sampling

Finally, implement `Warp::squareToPhongLobe()` to transform 2D canonical random numbers to directions distributed about a cosine-power distribution (i.e., a Phong BRDF lobe), again aligned about the z-axis. As usual, implement the corresponding PDF evaluation, `Warp:squareToPhongLobePdf()`, too.

1.2 Ambient Occlusion Monte Carlo Estimators (*10 pts*)

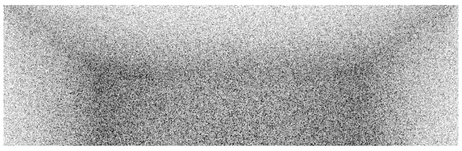
Your next task is to implement three (3) Monte Carlo estimators for ambient occlusion according the following importance sampling schemes: uniform spherical direction sampling, uniform hemispherical direction sampling, and cosine-weighted hemispherical direction sampling. You can find the API you need to implement for the `AOIntegrator` in `src/integrators/ao.h`, and the algorithm should roughly follow the structure below:

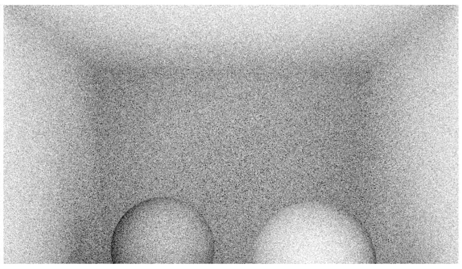
1. Intersect your eye rays with the scene geometry.
2. If there's an intersection `i`, solve for the appropriate Monte Carlo AO estimate at this shade point: you can sample directions in your MC estimator using the sampling routines you developed earlier.
3. When evaluating the visibility in the AO integrand of your MC estimator, take care when specifying relevant positions and directions; remember, all intersection routines expect world-space coordinates. Here, you will need to compute the parameters of a shadow ray based on `i.p` and `i.wi`.
4. When computing the contribution of each MC sample to the final integral estimate, don't forget to evaluate all the integrand terms and to divide by the appropriate PDF evaluation.

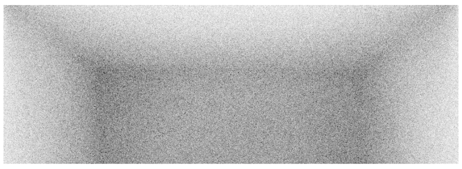
Note that you will have to limit the length of the shadow rays using a scene bounding sphere heuristic. Think about what would happen if you rendered an indoor scene with AO: any shadow ray you trace would hit the scene and the final image would be completely black. To avoid this problem, set the maximum shadow ray length to half of the bounding sphere radius. Use the `scene.aabb.getBSphere()` to retrieve this sphere and its corresponding radius.

The albedo is commonly set to 1.0 for AO (and RO).

Visually benchmark your integrator on the Cornell Box scene, using each of the three sampling methods. Be sure to compare your output with the test images below, using 16 spp.


Uniform Spherical Direction Sampling


Uniform Hemispherical Direction Sampling


Cosine-weighted Hemispherical Uniform Sampling

If you did not implement multiple pixel samples in A1, you can control the MC sampling rate as a parameter of the integrator, looping and averaging over your samples directly in the integrator. In this case, you will shade rays passing through the pixel centers, but you will sample multiple directions per pixel in `AOIntegrator` (and `ROIntegrator`).

1.3 Reflection Occlusion Monte Carlo Estimator (*10 pts*)

Reflection Occlusion is a direct illumination effect that is very similar to ambient occlusion except, instead of each surface having a diffuse BRDF, each one has a Phong BRDF.

Concretely, you will solve the direct illumination equation with $L_e = 1$ and $f_r = \frac{n+2}{2\pi}\max(\cos^n \alpha, 0)$. Here, α is the angle formed by the mirror reflection of the view vector about the normal with the incident direction (i.e., the integration directions you will be sampling). Don't forget, there is another (clamped) cosine about the normal for the projected solid angle measure (i.e., foreshortening).

For RO, you will only be graded on the implementation of a single (1) Monte Carlo estimator using a **cosine-power lobe** hemispherical direction sampling scheme. That being said, you can validate the correctness of your estimator by implementing other sampling schemes and seeing if images from different estimators converge to the same result (they should!).

The high-level structure of this estimator is similar to the AO estimators, with the exception of the difference in the BRDF terms. Be mindful of the various (clamped) cosine terms, and the coordinate frames the directions you will be taking scalar products need to be in.

Implement routines to evaluate and perform cosine-power importance sampling for the Phong BRDF. These functions should call your `Warp` sampling methods. Complete the implementation of `ROIntegrator` in `src/integrators/ro.h` and render the test scene below:

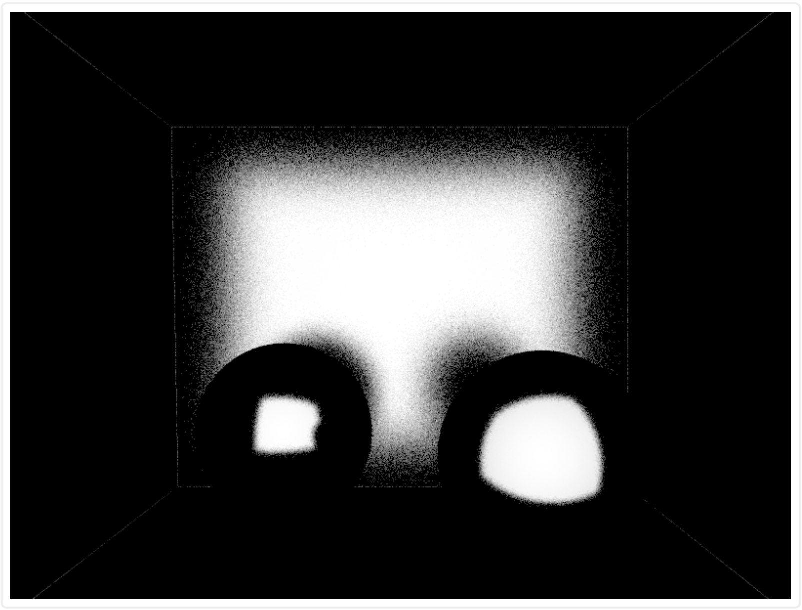


Figure 1: *Reflection Occlusion*

Your images will not perfectly match the test images due to random noise, but there shouldn't be any noticeable intensity differences overall.

2 Real-time Rendering (*60 pts*)

Screen Space Ambient Occlusion (SSAO)

Ambient occlusion is a compelling visual effect, but it is also typically too expensive to compute accurately at interactive rates. SSAO is a rendering method that approximates AO at interactive speeds.

We will see that, to do so, SSAO will make many simplifying assumptions in order to increase performance. As such, you should not expect renderings generated with SSAO to match “ground truth” computed using your more physically-based MC estimators, above.

SSAO is the first example of a *deferred rendering algorithm* you will have encountered in the course. So far for interactive rendering we have relied on a more traditional *forward* rendering structure: you submit draw calls for the geometry you want to display and bind shaders that act directly on this geometry (e.g., transforming its geometry and computing shading values, either at vertices or pixels or both).

A *deferred* rendering structure involves a more complicated draw loop, with the benefit of enabling more advanced interactive rendering effects (such as SSAO).

The idea behind deferred shading is straightforward: instead of computing the shade directly from the geometry before populating to the screen buffer, all in a single pass (i.e., a single submission of the scene geometry, bound to a single pair of shaders), you will first render auxilliary data about the scene out to temporary buffers. In secondary passes, you will use this auxilliary data to perform shading, populating the screen buffer.

This auxilliary data you typically render during the first pass of a deferred rendering loop can include, for example, normals and positions (each of which in whatever coordinate system you like), texture coordinates, or even per-object material IDs.

The set of all these temporary, off-screen buffer outputs is referred to as the **G-buffers** generated by your deferred renderer. To generate the G-buffers, you submit scene geometry for drawing, but now you bind special shaders that will output the properties you want (instead of more traditional forward rendering shaders, that compute and output the final shade).

After generating your G-buffers (in what's often referred to as the **geometry pass** of your draw loop), you perform an image post-processing computation that uses them as input and computes the output based on a (hopefully more advanced) shading algorithm. This last step is referred to as the **shading pass** of your deferred renderer.

In the case of SSAO, implemented in `src/renderpasses/ssao.h`, the two passes need to perform the following tasks:

1. *Geometry pass*: populate G-buffers with *screen-space* normals and positions.
2. *Shading pass*: render a full-screen quad to access the auxilliary G-buffer data (e.g., passing them in as input textures) before approximating AO using an MC estimator of a simplified form of the AO integral. This occurs in the SSAO shader.

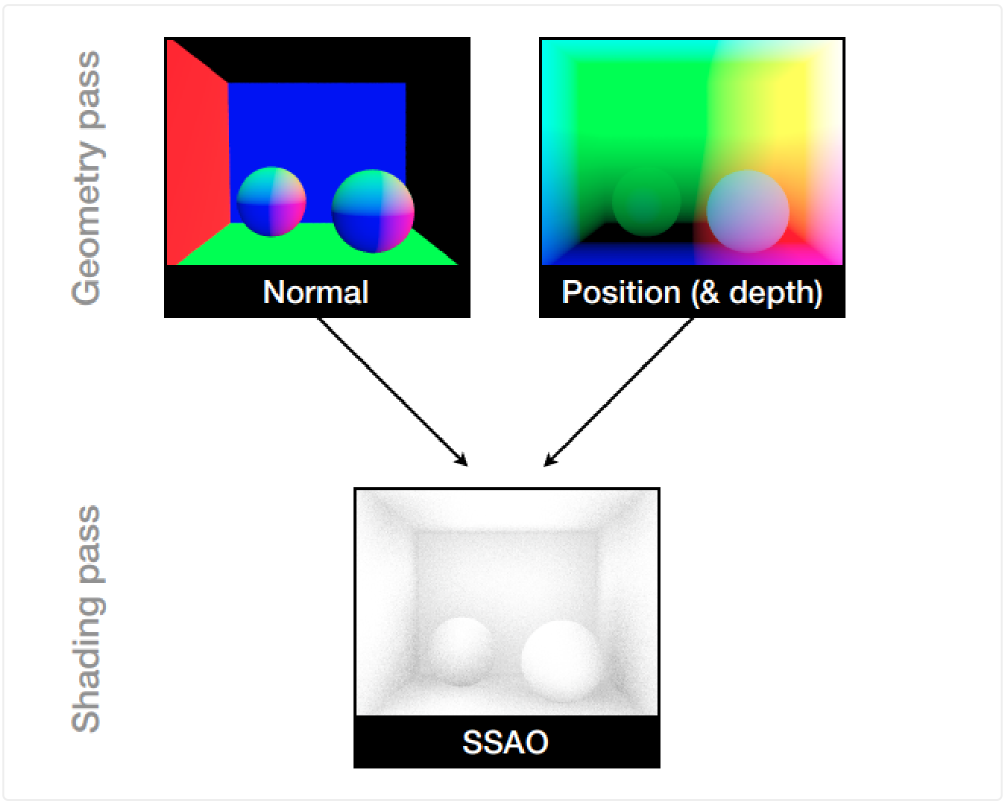


Figure 2: *SSAO Pipeline Diagram: in a first pass (the geometry pass), you will output screen-space normals and positions; then, in a second pass, you'll use this data to devise an approximate visibility and implement an MC estimator to compute the shading.*

G-buffers are supported by OpenGL using the Frame Buffer Object (FBO) facility, and they can be associated to one or many individual textures. GPUs have a fixed number of texture units, each of which can be thought of as a 2D buffer optimized for storing texture data. To access a texture OpenGL GLSL shaders, you need to first activate the texture unit and then bind a texture to it. The TAs will cover the basics of setting up a deferred renderer in the tutorial session, but you will be expected to perform independent reading and experimentation in order to set up the appropriate render loop logic.

2.1 Geometry Pass (*10 pts*)

You will implement a geometry pass that populates the G-buffer data. The shaders `shaders/geometry.vs` and `shaders/geometry.fs` already perform the brunt of the logic for you, but you need to complete the draw call and texture/FBO/shader bindings in `renderpasses/ssao.h`.

To populate these G-buffers, you need to consider what geometry you will be submitting to draw, which shaders need to be bound when drawing it, and what buffers should be active when rendering. Our code leverages a **multiple render target** (MRT) facility that allows us to output to more than three (3) “color” channels (and, so, to more than one texture) using a single call; the `shaders/geometry.fs` shader renders screen-space positions and normals into such an MRT, all from a single set of geometry draw calls.

Deferred Geometry Pass — High-level Steps

1. Bind the appropriate G-buffer FBO for output, bind the G-buffer generation shaders and their uniforms.
2. For all objects: bind the vertex array object for the geometry, draw the object, and unbind its vertex array.

2.2 Shading Pass: Full-screen Quad Rendering (*20 pts*)

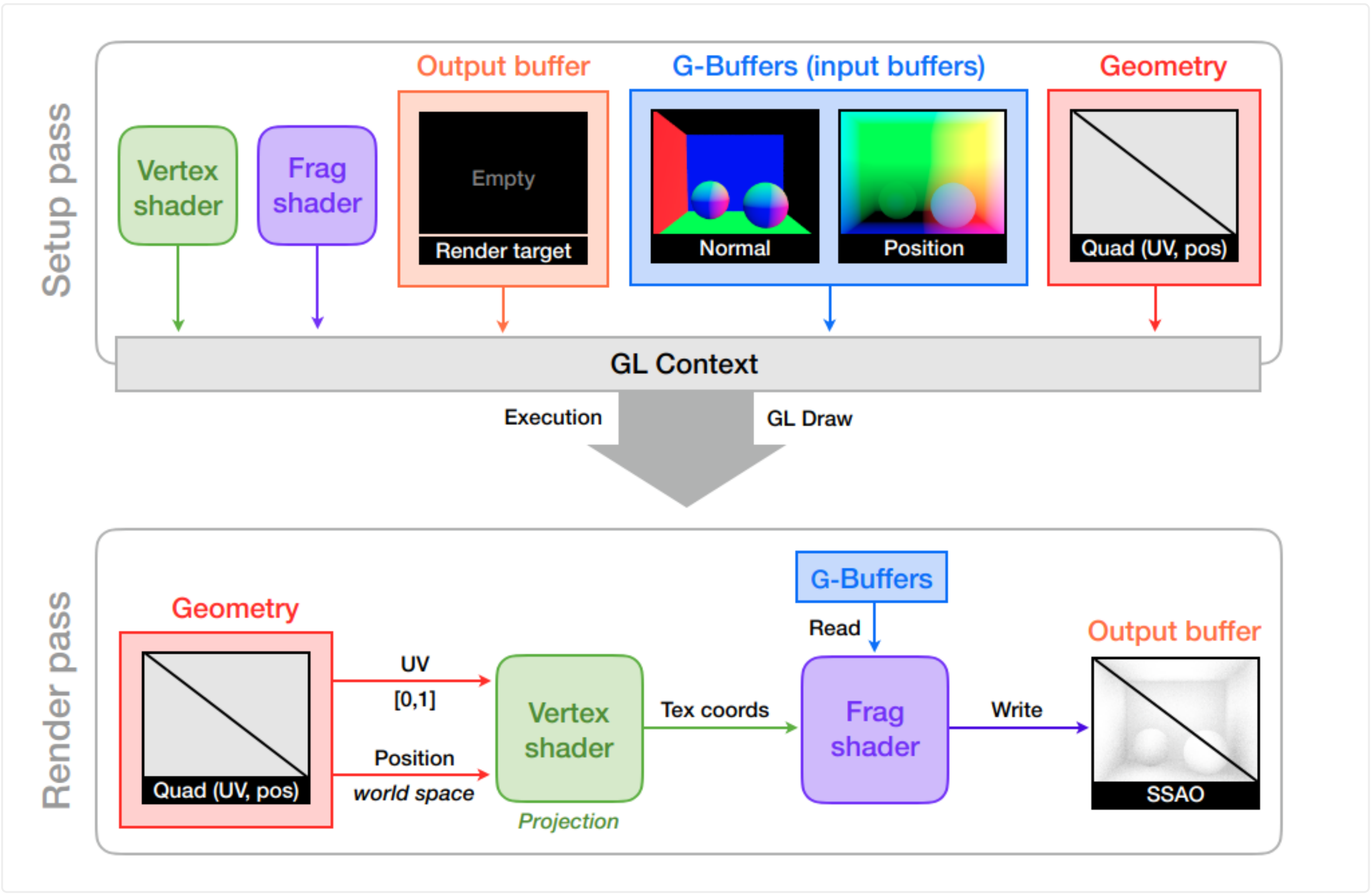


Figure 3: *Overview of the full-screen quad rendering pipeline.*

Vertex Shader (*5 pts*)

The shading pass does not operate directly on the scene geometry but, instead, performs full-screen image post-processing using the G-buffer data as input. As such, you want to render a "scene" that will allow you to perform image processing operations using a fragment shader.

This requires what's commonly referred to as a **full-screen quad** render pass, where we render a quad (or triangles) that is perpendicular to our view and that covers the entire screen. We already provide the

`SSAOPass::quadVAO` and `SSAOPass::quadVBO` data needed to create this geometry, and you need to implement a vertex shader (`shaders/quad.vs`) with the following input/output behavior:

Attributes (`in`)

- Position (in *world space*)
- UV (2D)

Outputs (`out`)

- Screen-space texture coordinates
- *World-space* positions

Fragment Shader — SSAO Computation (*15 pts*)

This final step of the SSAO method is where all the shading magic happens. Here, you will access the G-buffer data (using the texture coordinates you outputted in the previous full-scree vertex stage!) and use their data to approximate AO using an MC estimator and a rudimentary screen-space visibility test.

Deferred Rendering Shading Pass (Fragment Shader) — High-level Steps

1. Bind the output frame buffer of this pass `SSAOPass::postprocess_fboScreen`.
2. Bind the appropriate SSAO shaders and their uniforms.
3. Bind the input textures from your G-buffer, containing position and normal from the GBuffer.
4. Bind the vertex array for the full-screen quad geometry.
5. Draw the quad geometry.
6. Unbind the vertex array.
7. Unbind the textures.

All that's left is to actually implement the shading logic, in what amounts to a screen-space post-process in a fragment shader.

2.3 SSAO Fragment Shader (*30 pts*)

You need to complete the implementation of the SSAO algorithm in the `shaders/ssao.fs`. This shader is responsible for computing the approximation of AO, relying exclusively on the data present in the G-buffers.

You will compute SSAO, at each pixel, with a 32-sample Monte Carlo estimator of the ambient occlusion equation that uses uniform hemispherical direction samples. We provide you with shader code for generating canonical uniform random numbers.

One fundamental deviation from your ray-tracer implement here is how visibility will be computed (really, approximated).

Given a shading point $\mathbf{x}$ at a pixel, and an MC sampling direction ω_i, you will approximate visibility as follows: if a point $\mathbf{y} = \mathbf{x} + \beta\omega_i$ a fixed distance β away from $\mathbf{x}$ (in direction ω_i) is closer to the viewer than the point on the actual geometry that $\mathbf{y}$ projects onto (in screen-space), then $\mathbf{x}$ and $\mathbf{y}$ are mutually-visible otherwise the visibility is zero. A value of $\beta = 0.15$ is set for you in the `RADIUS` constant.

This aforementioned visibility logic provides a very crude approximation of mutual visibility that you can substitute for $V(\mathbf{x}, \omega_i)$ in your Monte Carlo estimator for (SS)AO. Keep in mind that screen-space depth is stored the z-component of the screen-space positions stored in the `texturePosition` texture of your G-buffer inputs. Be sure to offset by an epsilon, defined in the `BIAS` constant, to help mitigate some of the shadow acne artifacts.

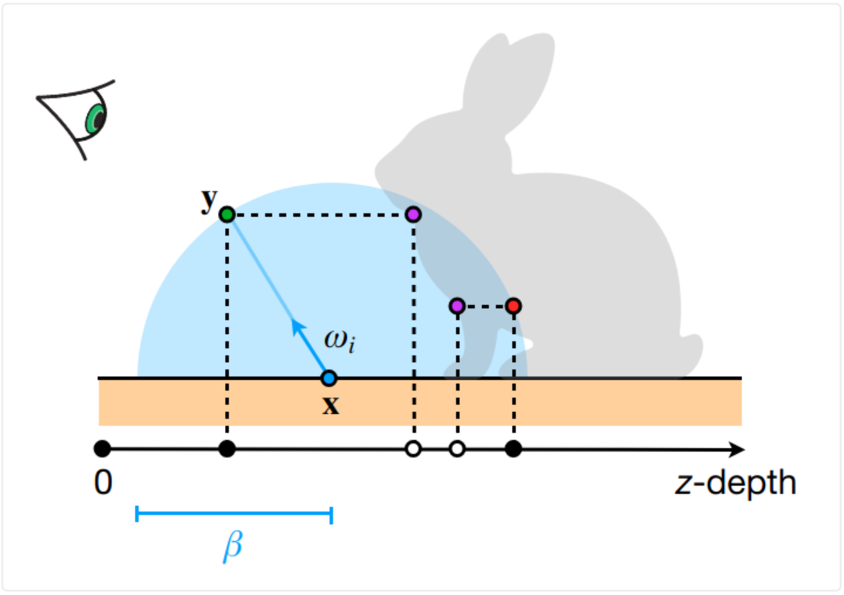


Figure 4: *Approximating (finite) visibility test for SSAO.*

You can consult the OpenGL documentation for technical details about the API.

2.4 Bonus: More Accurate SSAO Visibility (*20 pts*)

The visibility computation for SSAO is, put mildly, a huge hack. Differences in screen-space depth between points at a fixed distance away from your shading point introduces many opportunities for false-positive **and** false-negative visibility results.

Instead of comparing the depth between the geometry rasterized into the G-buffers and a *single* point sampled a distance β away from your pixel shade point $\mathbf{x}$, sample **many** points along the direction ω_i (i.e., for $\epsilon \leq \beta \leq \infty$) until the "ray" in direction ω_i "escapes" off-screen. When *marching* this ray over screen-space, test for visibility using something more sophisticated than screen-space depth-based discrimination, e.g., by treating the cosines at $\mathbf{x}$ and each sampled $\mathbf{y}$, for example.

Submit your most visually pleasing AO approximation, limiting yourself to 32 samples per pixel. You have full latitude over the complexity of your improved visibility approximation.

What to submit

Render all the scenes in `a3/spaceship/tinyrender`. Note that you will need to render `spaceship_ao.toml` multiple times, once for each setting of the importance sampling schemes you implement in your `AOIntegrator`. Finally, edit your `config.json` to include your credentials. Your configuration file should be similiar to the following; **only lines 2–4 should be modified.**

```
{
  "firstlast": "John Doe",
  "id": 123456789,
  "course": 446,
  "assignment": 3,
  "renders": [
    { "scene": "Ambient occlusion with spherical uniform sampling",
      "render": "offline/spaceship_ao_sphere.exr"
    },
    { "scene": "Ambient occlusion with hemispherical uniform sampling",
      "render": "offline/spaceship_ao_hemisphere.exr"
    },
    { "scene": "Ambient occlusion with cosine-weighted hemispherical sampling",
      "render": "offline/spaceship_ao_cosine.exr"
    },
    { "scene": "Reflection occlusion",
      "render": "offline/spaceship_ro.exr"
    },
    { "scene": "Screen space ambient occlusion",
      "render": "realtime/cbox_ssao.exr"
    }
  ]
}
```

Submit this file with all your code in a `.zip` or `.tar` archive file. Include your raw `.exr` files in separated folders (see structure below).

```
a3_first_last.zip
   config.json
   src
   offline/
      spaceship_ao_sphere.exr
      spaceship_ao_hemisphere.exr
      spaceship_ao_cosine.exr
      spaceship_ro.exr
   realtime/
      spaceship_ssao.exr
```

Make sure your code compiles and runs for both the interactive and offline components before submitting! **You will obtain a score of zero if your submission does not follow this exact structure.** You can use the `tree` command to verify it.