CS5625 Programming Assignment 4 (PA4): SSAO
Due Monday April 8th, 11:59pm
Work in groups of 2.

Instructor: Kavita Bala

1 Overview

In this assignment, you will implement a simple version of Screen Space Ambient Occlusion (SSAO). Note that you have less time than usual to complete this assignment.

This assignment is a bit different than what we went over in lecture. You will only sample on the surface of a hemisphere instead of within a sphere. Sticking to the surface gives less accurate results for large radii but is much faster. We also get a speed boost by only sampling within the hemisphere normal to the surface.

We have added another intermediate screen-space buffer which you will fill with the SSAO visibility value at each pixel. This buffer functions in exactly the same way as the silhouette buffer from PA2: it is filled in its own rendering pass between the material properties pass and the ubershader pass. Note that we have added a line of code to the ubershader that scales the output color by the visibility factor in the SSAO buffer.

The sample rays used to compute the visibility factor are passed to the SSAO shader in an array of uniform vectors. You will compute these rays in the Renderer class and store them in an array; we will handle the boring OpenGL code to pass them to the shader.

Here is a brief summary of the keyboard controls for this assignment. The 's' key toggles SSAO; by default SSAO will be turned off. The 'r' and 'R' keys decrease and increase the radius of the sample rays. The 'e' and 'E' keys decrease and increase the number of sample rays (and trigger ray regeneration); the number of rays defaults to half of the maximum number of rays as set in the SSAO shader. To view the SSAO visibility buffer alone hit '6'.

2 Sample Ray Generation

You must fill out the createNewSSAORays method in Renderer.java such that mSampleRays is filled with exactly numRays sample rays. Usually this is done by generating random unit vectors, but if you figure out a way to generate them using a deterministic algorithm that will work just as well. We require that your rays satisfy the following properties.

1. They should be uniformly distributed over a hemisphere If you're not sure how to do this, check: http://mathworld.wolfram.com/SpherePointPicking.html. We recommend either using equations 1 and 2 to compute spherical coordinates or equations 6, 7, and 8 to compute x, y, and z coordinates. Note that these equations give you uniform points over a sphere, not a hemisphere. One easy way to fix this is to take the absolute value of the z component.

2. They should be normalized. The sampling radius will be applied in the SSAO shader.

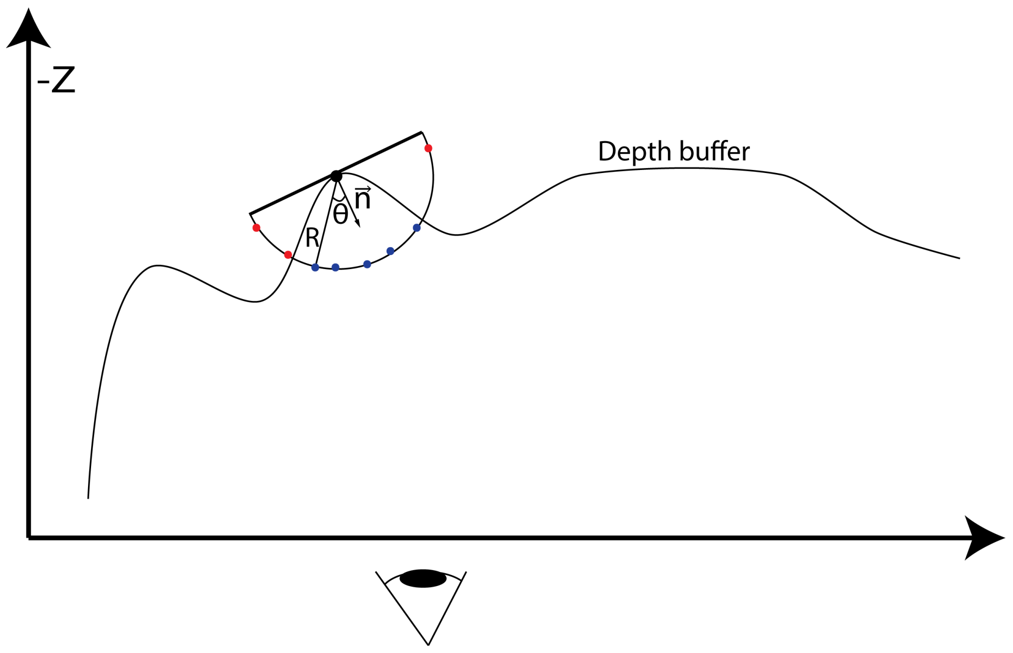

Figure 1: The camera's view of the SSAO algorithm. The colored points represent sample points on the surface of a hemisphere normal to the current fragment (the black point). The red points are blocked points because their depth is more negative than the depth in the z-buffer. Note that the contribution of each blocked point must be scaled by $cos(\theta)$.

3 The SSAO Shader

This shader takes in the diffuse buffer, the position buffer, sample rays, ray count, ray radius, the projection matrix, and the size of the screen in pixels. Your task is to turn these inputs into a gray-scale color that represents the visibility of the current fragment where black means completely blocked and white means completely visible.

Since we required that all of your sample rays fit within a hemisphere you must do a bit of work to make sure all sample rays face away from the surface belonging to the current fragment. This comes down to constructing a change-of-basis matrix that takes the sample rays from the space of the fragment to screen space such that the normal of the fragment is equal to the axis of symmetry of your hemisphere. If you took our ray generation advice in Section 2 the axis of symmetry is the positive z axis. You should be able to construct the basis needed for this matrix quite easily (you only need the normal); try to remember how you used cross products to create a camera's basis in CS 4620.

Once that matrix is complete you must compare the z-buffer depth to the sample point depth. If the sample point is further back than the point in the z-buffer we treat it as a blocked sample ray. See Figure 1 for a diagram of what exactly is going on. There are three tricks to getting this right, which we will explain the following sub-sections.

3.1 Accessing the Z-Buffer at a Sample Point

Going from a sample ray to its corresponding z-buffer texture lookup can be confusing. To make matters worse even a small mistake can result in seemingly arbitrary visibility values. Below, we provide the steps you must take to arrive at a pixel-based texture coordinate for a texture2DRect lookup so that you can find the z-buffer depth value at the sample point. Note that when we refer to the z-buffer we simply mean the z values of the positions in the g-buffer.

1. Apply the change-of-basis matrix discussed earlier to get a screen space ray.
2. Scale the transformed ray by the SampleRadius; this is the final sample ray.
3. Offset the screen space position of the current fragment by the scaled ray and project this point into clip space using the ProjectionMatrix.
4. Convert the x and y values of the clip space point into pixel values using a bit of math and ScreenSize. Recall that clip space ranges over [-1, 1] in x and y. Since you've done a projection, remember to divide by w.

3.2 Weighting the Sample Points

We need to weigh the visibility of each ray by its dot-product with the normal. This helps eliminate aliasing artifacts due to the finite resolution of the position buffer and also makes sure that the "important rays" near the normal contribute more towards the visibility factor.

To implement this trick, simply dot the normal against the ray from Step 1 in the previous sub-section and add that value to your contribution instead of the value 1. Keep in mind that you must divide by the sum of all dot products now, not the ray count.

3.3 Treat Background Points as Visible

G-buffer texels with a depth of 0 correspond to the background; the place where no geometry was rendered at all. Since all depth values are negative, you will run into trouble if you do not look out for these texels. To properly handle them you must do two things.

1. If the current fragment is part of the background return a visibility of 1. This will leave the background colors unchanged by SSAO.
2. If one of our sample points is part of the background you should treat it as unblocked. In other words, a sample point can only be blocked if it's z-buffer depth is negative.

An additional step was taken in the Java framework to get this working properly; during the execution of the SSAO shader we disable texture interpolation for the g-buffer. See the appendix at the end of this document for a full explanation.

4 Tips and Tricks

Just like in PA3, there are a number of prerequisite TODOs that you must fill in before the assignment will render the default scene. There are only two TODOs for PA4 material and they are in Renderer.java and ssao.fp.

The second trick mentioned in Section 3 can be ignored at first; if you just add 1 to your counts for each ray you still get pretty reasonable results. It is a good idea to put it off until you get everything else working, and then implement it at the end (but, you must implement it!).

We have added a set of screenshots of the visibility buffer to CMS. The exact parameters used to takes these screenshots are given in their filenames; e.g., "radius0_05" means the sample radius was 0.05 and "rays50" means the ray count was 50.

When testing your code keep in mind the following aspects of the new test scene. We've made the background a bit brighter to make it more obvious that it shouldn't get darker when SSAO is turned on. We've added a Menger Sponge to the scene; it's an excellent object for showing off the fake shadows you get from SSAO. Finally, it's expected that the bottom side of the ground plane has a visibility of zero because it's normal is always pointing "up" and thus the entire sampling hemisphere will be on the wrong side.

5 Submission

Turn in your code, a README explaining what you did, and what files you changed. Mention any problems you might have had. Feel free to change whatever you need in your code base and tell us what you needed to change.

6 Appendix

This section provides the rationale behind the new enableInterpolation method in Texture.java and its use in Renderer's computeSSAOBuffer method. You didn't have to write any of this code but you should still understand how this code works and why we had to write it.

Recall that we must be able to detect background texels so that we can make sure to treat them as visible; this should be fairly easy as all background texels have a depth of 0 whereas all non-background texels have a negative depth. However, if you implement this entire assignment correctly but comment out our use of the enableInterpolation method in Renderer you will see strange artifacts on points near the background. Too see what goes wrong imagine where in texture space an arbitrary sample point falls: by all likelihood it will not land exactly on the center of a texel. This means that you will get a bi-linearly interpolated depth value! Thus any sample points that land on the edge of the object will see a depth value that is closer to zero than it should be, causing the ray to be incorrectly seen as blocked.

There are two ways to fix this in practice: clamp all texture uv's to exact pixel centers or disable texture interpolation. In this case we chose to do the latter because it is much easier to get right. You might expect that disabling interpolation would cause some problems as our g-buffer is now effectively a fancy step function; however, there is no drop in quality because we have exactly one g-buffer texel for each screen pixel.

This problem and it's solution should raise questions about the correctness of our framework in other areas. For example, it's a terrible idea to interpolate things like the material ID because it's not a linear quantity: half-way between materials 2 and 4 gives material 3 which probably isn't what you wanted at all! Keep in mind that this will only be a problem if you sample from the g-buffer between texels; in most cases (like in the ubershader) we always sample at the center of each texel.