CG Programming III – Assignment #3 (SSAO)
Due on 06/13/2012 (at the final)

In this assignment you will be required to implement a portion of screen-space ambient occlusion. The implementation wil proceed in three phases. The first phase modifies the rendered scene to produce ambient occlusion effects.

- Render a *lot* more objects to the scene. All of the objects may be the same, but there should be a lot of them. The should be arranged so that, from the camera's point of view, there are a lot of "creases" and interesting topology that will lead to ambient occlusion.
- Disable shadows.

The second phase builds some infrastructure for post-processing. Recall that SSAO is, at its roots, a post-processing effect.

- Render the scene to an FBO. This FBO should (initially) be configured with a color texture attachment and a depth texture attachment. For visualization, the FBO should be copied to the window using `glBlitFramebuffer`.
- Disable the FBO-to-window copy.
- Using the color texture attached to the FBO, draw two triangles that cover the whole screen and copy the texture. The rendered result should be identical to the results produced by the (now removed) `glBlitFramebuffer` call.

You should be able to simply modify assignment #1 to use shadow maps instead of shadow textures.

- Add two extra (color) attachments to the FBO, and modify the rendering shader to output the camera-space surface normal and the camera-space position to these outputs.
- Modify the texture-to-window shader to copy the camera-space surface normals (instead of the rendered colors) to the window. These will need to be remapped from $[-1, 1]$ to $[0, 1]$. Verify that the results look sensible.
- Modify the texture-to-window shader to copy the camera-space positions to the window. These will need to be remapped from the range of possible values in the view frustum to $[0, 1]$. Verify that the results look sensible. You'll have to do some trig to figure out what this range is. See below for partial derivation.
- Implement C code that will calculate the dimensions of region of space at the near and far planes that project to a single pixel. Provide this data as uniforms to your texture-to-window shader.
- Derive a formula to determine the area that projects to a pixel at any position in the view frustum. This is the radius of the occluding sphere for the SSAO calculation. Modify the texture-to-window shader to calculate the radius of each pixels. These will need to be remapped from $[r_{near}, r_{far}]$ to $[0, 1]$. Verify that the results look sensible.
- Modify the texture-to-window shader to calculate the SSAO factor from the lecture notes. A 17 by 17 grid of sample positions should be sufficient for this step.

Recall that the near plane is perpendicular to a view vector through the middle of the screen. Let n be the distance to the near plane. Let θ be the field of view angle. These are used to construct a right triangle. The length of the side opposite the camera is

$$n \cos \theta \tag{1}$$

This is half the height of the screen in camera space. If the screen is h pixels high, each pixel is

$$2n/h \cos \theta \tag{2}$$

high in camera space.

The width of a pixel can be determined by multiplying the result of equation 2 by the aspect ratio. Similar calculations can also be performed for the far plane.

Criteria	**Excellent**	**Good**	**Satisfactory**	**Unacceptable**
Completion	Program correctly implements all required elements in a manner that is readily apparent when the program is executed. User interface is complete and responsive to input. Program documents user interface functionality.	Program implements all required elements, but some elements may not function correctly. User interface is complete and responsive to input.	Program implements most required elements. Some of the implemented elements may not function correctly. User interface is complete and responsive to input.	Many required elements are missing. User interface is incomplete or is not responsive to input.
Correctness	Program executes without errors. Program handles all special cases. Program contains error checking code.	Program executes without errors. Program handles most special cases.	Program executes without errors. Program handles some special cases.	Program does not execute due to errors. Little or no error checking code included.
Efficiency	Program uses solution that is easy to understand and maintain. Programmer has analysed many alternate solutions and has chosen the most efficient. Programmer has included the reasons for the solution chosen.	Program uses an efficient and easy to follow solution (i.e., no confusing tricks). Programmer has considered alternate solution and has chosen the most efficient.	Program uses a logical solution that is easy to follow, but it is not the most efficient. Programmer has considered alternate solutions.	Program uses a difficult and inefficient solution. Programmer has not considered alternate solutions.
Presentation & Organization	Program code is formatted in a consistent manner. Variables, functions, and data structures are named in a logical, consistent manner. Use of white space improves code readability.	Program code is formatted in mostly consistent with occasional inconsistencies. Variables, functions, and data structures are named in a logical, mostly consistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted with multiple styles. Variables, functions, and data structures are named in a logical but inconsistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted in an inconsistent manner. Variables, functions, and data structures are poorly named. Use of white space hurts code reability.
Documentation	Code clearly and effectively documented including descriptions of all global variables and all non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of most global variables and most non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of the most important global variables and the most important local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted.	No useful documentation exists.

This rubric is based loosely on the "Rubric for the Assessment of Computer Programming" used by Queens University (http://educ.queensu.ca/ compsci/assessment/Bauman.html).