CG Programming III – Assignment #4 (SSAO)
Due on 03/23/2011

In this assignment you will be required to implement screen space ambient occlusion (SSAO). To show SSAO in action, render a scene with a large number of objects in close proximity to each other.

- Draw *multiple* objects in the scene. These can either be simple solids (e.g., the torus or sphere) or models loaded from disk. There should be a large number of object, and they should be in close enough proximity to exhibit ambient occlusion effects.
- Include a single light source in the scene.
- Render the scene to an off-screen rendering target (FBO).
- Implement SSAO as a post-processing pass. Use the depth buffer from the FBO as an input to determine AO values. Modulate the AO values with data read from the color buffer from the FBO. Write the resulting colors to the screen.

You should be able to modify assignment #2 as the basis for this assignment. You may want to remove the shadow mapping code.

The SSAO implementation may be simplified from a "production quality" implementation in a couple of ways.

- Instead of sampling the screen geometry using a rotated random grid, select samples using an ordered grid.
- Skip the application of a geometry-aware filter to remove noise caused by the sampling pattern.

One of the more difficult parts of SSAO is correctly implementing the back-projection to calculate eye-space positions from screen coordinates and depth buffer values. It may difficult to determine the origin of implementation bugs. Is the problem in the projection? Is the problem in the AO calculate (that uses the back-projected location)? Is the problem elsewhere? During development, it may be beneficial to write the eye-space position of each fragment to a separate floating point rendering target. This can then be used to develop the AO calculations.

One drawback of this approach is that most hardware cannot render to floating point and fixed point render targets at the same time. This means that two rendering passes will be required. The first pass will generate the color data to which the AO values will be applied. The second pass will generate the eye-space position values that will be used to calculate the AO values.

There is no requirement to use this approach during development. However, it may make debugging a lot easier along the way. In the final version, you *must* use the standard back-projection technique.

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).