CG Programming III – Assignment #2 (shadow maps)
Due on 05/15/2013

In this assignment you will be required to implement shadow maps. To show shadow maps in action, render a scene with *at least* two objects and a single light source. One of the objects must show self-shadowing. The light source and the objects must move in such a way that object can be both receivers and casters. One way to do this is to have multiple objects arranged in a plane with a light source orbiting them.

Part 1 - Shadow maps

For the most part, you will modify your existing shadow texture implementation to use shadow maps instead.

- Convert the "shadow" FBO to have a single `GL_DEPTH_COMPONENT` texture attachment instead of the current RGB color texture attachment.
- Convert the main rendering shader to use a `sampler2DShadow` sampler instead of a `sampler2D`.
- Use the `GL_DEPTH_COMPONENT` texture as the texture for the scene. Be sure to set the `GL_TEXTURE_COMPARE_MODE` to `GL_COMPARE_REF_TO_TEXTURE`.
- Modify the scene to include more objects, have a moving light source, etc.

At this point, you should have shadow textures working.

Part 2 - PCF

- Add keyboard (or joystick, etc.) controls to your main program for a adjusting a floating point value on the range $[.2, 5]$. Pass this value divided by the width of the shadow map (your shadow map should be square for this to work properly) into your shadow mapping shader as a uniform. I will refer to this variable as `r`.
- Add an array of 36 "random" values that represent points inside a circle of radius 1.0. You may want to write a separate program in C, Javascript, or whatever to generate the set of points. Being able to write programs that generate other programs can be very useful.
- Using the random points and `r`, implement a percentage-closer filter (PCF) in your shadow mapping shader. You will have to manually project the shadow map coordinate (by dividing by w) and bias the position by the random points scaled by `r`. As `r` gets larger, the shadow boundaries should get softer.

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).