CIS 565: Assignment 3: Advanced GLSL
Spring 2011
Due Wednesday, 02/09, using Blackboard by 11:59pm

Part 1: Questions (10%)
Include answers to the following questions in your readme file.

1. Explain why OpenGL immediate mode is not an efficient way to transfer vertices from
   system to GPU memory. (4%)

2. Explain the performance implications of using the OpenGL glReadPixels function to
   read data from GPU to system memory. (4%)

3. Explain how an application or the OpenGL driver can utilize a multicore CPU to
   minimize the performance impact of OpenGL driver CPU overhead. (2%)

Part 2: Globe Rendering (30%)
This part is based off the Globe starter project (different than the Sphere from the last
assignment). Most of your work should take place inside of globe.frag. In this assignment, we
will layer a number of effects using many texture maps to build up a rich appearance.
Starting with a simple diffuse-shaded globe with a texture map, we will build up night and
day sides, animated clouds, and specular highlights.
The images for this part were downloaded from http://planetpixelexemporium.com/
earth.html.

Controls
Hold left mouse and drag to rotate the globe
Hold right mouse and drag up and down to zoom
+ and - to change the speed of rotation

Night and Day (10%)
Add diffuse lighting, based on a coefficient of $\text{diffuse} = \max(\text{dot}(-\text{DirectionalLight}, \text{Normal}), 0)$. Note that this is a directional light source, as opposed to a point light source on the last assignment. Your output color should linearly interpolate between the day and night textures based on the diffuse coefficient, instead of fading to black on the night (unlit) side. That is, it should show the day texture with coefficient $\text{diffuse}$ and the night texture with coefficient $(1.0 - \text{diffuse})$.

Completed Video: [Night And Day](#)

**Clouds** (10%)
We will now layer clouds on top of the globe. We will use a transparency map to blend between showing the cloud texture where they exist and the globe from the previous step where they don’t. Moreover, we’ll animate the clouds so they move independently of the Earth’s surface.

To animate the clouds, add a uniform time parameter that you advance every frame. Offset the s dimension of the cloud texture coordinate by this time parameter to swirl the clouds around the globe. (Note: This works because the cloud textures are set to repeat the texture if you access outside of the 0-1 range. Thus, no modulus operations are necessary). Make sure to offset the look-up value for both the cloud texture and the cloud transparency map.

Like the Earth, the clouds will be subject to diffuse lighting. Just multiply the cloud color by the diffuse coefficient. This time, we will fade to black on the dark side of the Earth, instead of blending with a darker texture. This will make the clouds appear as dark shadows as they pass over the lights on the dark side.

Finally, we need to composite the clouds over the globe. The cloud transparency map is
dark where the clouds are visible and light where the clouds are not. Show the day-night globe with coefficient \texttt{cloud\_transparency} and the clouds with coefficient \(1.0 - \texttt{cloud\_transparency}\).

Completed video: \texttt{Clouds}

\textbf{Specularity (10\%)}

Calculate a basic specular highlight as in the Blinn-Phong examples from the Sphere project from the last assignment. This time we will modulate the strength of the highlight by a specularity value that says how specular is the surface at that point. We have a specular map describing how specular the Earth’s surface is; oceans are reflective and landmasses are not. Notice how in the image above, the specular highlight stops at the edge of North Africa.

However, clouds are also reflective, so there should be a specular highlight where clouds are present. To compute the overall specularity, blend between the Earth’s specular map and a constant \texttt{vec3(1.0f)} representing the clouds with a parameter of \texttt{cloud\_transparency}. Multiply the specular highlight you computed by this specularity to compute a final specular component, which you can add to the output color from the previous task.
Completed Video: Specularity

Extra Credit (10%)
Add bump mapping or displacement mapping using the provided displacement texture. These techniques fake geometric surface detail based on a texture describing height offsets from the baseline geometry.

Part 3: Screen Space Ambient Occlusion (60%)

Background
In this part, we explore Screen Space Ambient Occlusion (SSAO), a new but already ubiquitous technique in real-time graphics. First introduced in Crysis in 2007, SSAO approximates global illumination using basic scene information captured in normal, depth, and/or position buffers, yielding a decent visual appearance at a fraction of the cost.

Global Illumination is one of the most general and expensive illumination models. Unlike simple diffuse and specular lighting discussed in class, in Global Illumination indirect sources of light are fully considered (example, light bouncing around the inside of a room). Global Illumination might be done offline through radiosity or photon mapping.

Ambient Occlusion is a cheaper offline approximation of Global Illumination, first used in Pearl Harbor. Ambient occlusion assumes that all light sources are ambient; that is, incoming in uniform strength from all directions. Think of the illumination of an overcast day. Ambient Occlusion measures how much ambient illumination is occluded, or blocked, by nearby geometry. To calculate ambient occlusion, rays are cast outwards into the scene from world-space sample points and collided with nearby geometry. For static or semi-static objects (for
example, mildly deforming characters) ambient occlusion can be precomputed and included into
real-time lighting calculations.

SSAO takes advantage of the fact that much of the information needed to calculate
Ambient Occlusion is present in screen-space buffers of position, normals, and depth, that
might be present in, for example, a deferred renderer. To create such buffers, we render the
scene geometry with a shader program that outputs the desired attributes for each fragment.
To estimate the ambient occlusion, instead of casting rays out into the scene to collide with
scene geometry, we sample in the neighborhood of points from these stored screen-space
buffers.

This implementation will be comparable with many common post-processing effects
such as motion blur, depth of field, or HDR lighting.

This project is inspired by writeups from:
http://www.gamerendering.com/2009/01/14/ssao/ and
http://www.gamedev.net/page/resources/ /reference/programming/140/lighting-and-shading/
a-simple-and-practical-approach-to-ssao-r2753 and the Poisson disk samples from http://
developer.download.nvidia.com/whitepapers/2008/PCSS_Integration.pdf:

Don’t take code directly from these sites, but feel free to read them if you feel lost.

Tasks
You will complete one function to estimate occlusion and three different sampling
schemes to use to gather samples. All code changes you need to make should be isolated to
ssao.frag in the Street project.

Starter Code
The provided starter code in the Street project creates a mock city scene. You may
look around the world by holding and dragging the mouse, and walk around using WASD. Most
of your work will take place in ssao.frag, which already has uniforms for textures containing
position, normal, and depth information about the scene from an earlier pass.

There are a number of different display modes available that can be switched between
using the keyboard:
6: Display Depth
   Linearized depth. Note that the background is white here and black in the
   position view (since nothing was draw here, but depth is set to its maximum value when its
   cleared)
7: Display Normals
   Absolute value of view-space normals
8: Display Position
   Absolute value of view-space position, scaled by the far clipping plane
9: Display Occlusion
   Total occlusion subtracted from white
0: Display Total
   Diffuse Lighting and occlusion subtracted from ambient lighting
In Total and Occlusion, there are a number of different sampling schemes to switch between:
1: No Occlusion
2: Regular Grid Samples
3: Poisson Screen Space Samples
4: World Space Samples

**Occlusion Function** (15%)

The first task to complete is

```c
float gatherOcclusion(vec3 pt_normal,
                      vec3 pt_position,
                      vec3 occluder_normal,
                      vec3 occluder_position)
```

which is a function to estimate the occlusion of point by a new sample occluder. This function has several desired qualities; one is occlusion should fall off with large distances to prevent samples from foreground objects from spuriously occluding background points.

Another quality is that points should not be occluded much by samples in the same or similar plane. This prevents false self-occlusion on a flat surface. Co-planarity can be determined by looking at the normals at the point and the sampled occluder.

The last quality is that occluding samples that are located overhead a point, with respect to its normal, are more important than samples that are located obliquely. This is because, in a basic diffuse lighting model, the luminance due to a light source is proportional to dot(Normal, Incident). Thus ambient light sources blocked out by an occluder overhead are more important.

Design a function that has that roughly fits these criteria and any more you can think of. It should look roughly like the examples, but doesn’t have to be an exact replica. You may change the function signature if you need additional information. Very inspired estimators will receive extra credit up to 5% of the total assignment.

Note: You may wish to start this part to get something workable, then do some of the sampling tasks before coming back to tune it. Designing a function is easier with a good sampling scheme to view to the results.

**Implementation:** (10%) 

Answer the following question in your readme:

**Discussion** (5%): Explain why you chose the occlusion function you did and what the various terms do.

**Regular Samples** (10%)
Armed with your occluding function, you can now estimate ambient occlusion by sampling a number of occluders around each fragment and average the results. The first sampling scheme we'll try is using a regular grid in screen-space around each fragment. To keep performance and appearance roughly consistent with the next two parts, you should use 16 total samples in a 4x4 grid, and use the step size provided in ssao.frag. At each screen-space grid sample, access the position and normal buffers and call your occlusion function to estimate the occlusion on the fragment from that sample. Average the sampled occlusions to get your result.

**Implementation (5%)**

Answer the following question in your readme:
**Discussion** (5%): This sampling scheme will intentionally have artifacts. It is included because it is a natural first step, of which you should learn the limitations. Describe the sampling artifacts you observe.

**Poisson Disk Screen Space Samples** (15%)  

This sampling scheme seeks to eliminate many of the coherent artifacts from the regular grid. It will use a two-pronged approach to do so. First, we will no longer sample on regularly, evenly spaced grid points. Instead, we will sample on an irregular Poisson disk, an irregular but well-spaced distribution. Second, will not use the exact same sampling mask at each fragment, but will instead rotate the mask based on a random angle per fragment.

We will again use 16 samples per fragment. The disk is provided as a 16-element array of vec2’s in ssao.frag called poissonDisk. For each fragment, grab a random 0-1 float from a random texture with getRandomScalar(), linearly scale it to a random angle between 0 and 2*PI,
and rotate the disk points around the origin by that angle. Then sample around the screen-
space coordinates of the fragment with offsets from rotated disk and average the computed
occlusions. To keep the appearance consistent with the other sampling schemes, you should
scale the disk radius by the provided radius.

**Implementation** (10%)

Answer the following questions in your readme:

**Discussion** (2%): Why would high-frequency white noise such as this scheme may
produce be preferable to coherent larger-scale artifacts like the Regular Sampling may
produce?

**Discussion** (3%): We only used a single random number per fragment to break up
artifacts. Why didn’t we simply collect all 16 samples per fragment uniformly at random
(ie use 16*2 random floats)?

**World Space Samples** (20%)
One problem with both of the previous schemes is that the apparent radius of occlusion is the same on foreground and background objects. This is because the sampling is done irrespective of the scene geometry. We can patch this by scaling the sampling radius based on distance, but instead we'll present an entirely different approach.

Instead of sampling occluders from a screen-space disk, we will sample occluders on a view-space hemisphere around the fragment. This gives us accurate scaling of the effect with distance, better importance sampling (since we only sample points in front of the fragment), and a sampling that better represents the neighborhood of the fragment. As a reminder, View-space is the coordinate system after the model and view transformations, but before the perspective transformations, where points exists in 3-dimensions but up and forward are aligned to the camera.

As before, we will use a single set of 16 sample points in a Poisson Sphere. To break up artifacts, we will again randomly adjust the points per-fragment. In this case, we will get a
random unit vector using getRandomNormal(), and reflect all the sample points across the plane that unit vector defines. This is done per-fragment.

Once we have a somewhat-random sphere of 16 points, some of points on that sphere may be “behind” the normal of the fragment, and couldn’t contribute to occlusion. We want to reflect them across the plane of the normal, so that we now have a hemisphere of points above the normal. To keep the appearance consistent with the other schemes, you should use scale the hemisphere by the provided radius.

Now that we have a proper hemisphere, our occlusion samples have to be taken in 0-1 screen space. Thus, we need to transform our view-space points into screen-space. This is similar to what happens in most vertex shaders and then later during clipping.

First, the points should then be transformed to perspective space by multiplying with the provided projection matrix u_Persp. After this step, you need to perform homogenous division and divide all the points by their w coordinate. The points are now in clipping space, which ranges from -1 to 1. Since screen-space ranges from 0 to 1, we need to perform a last affine remapping of clipping/2 + 0.5 to place the points in screen-space. We can now sample occluders and average as before.

**Implementation:** (12%)

Answer the following questions in your readme:

**Discussion:** (3%) What is the primary cost of performing world-space sampling instead of screen-space sampling?

**Discussion:** (5%) Compare/Contrast the appearance of the world-space sampling scheme with the Poisson disk screen-space sampling from before.

**Addendum: Helpful Debugging Hints!**

- Self-shadowing, Flat planes are darker than they should be:
  - Check occlusion function, either attenuate occlusion based on the angle between the point’s and the sample’s plane or include a term that favors overhead occluders.
- Streaking, “Brushstrokes”
  - Sampling schemes, For 2 and 3 check that you are randomizing properly
  - Check that random textures are working
- Glowing edges
  - Check self-shadowing, make sure you don’t have negative terms causing negative “occlusion”
- Bold dark outlines or, alternatively, Halos
  - Make sure you attenuate occlusion from distant occluders, i.e. foreground objects on background objects
- Color “Banding”, Posterization
  - Insufficient storage and computation precision
- Wild all dark or all white flashes
- Out of range values
- Negative values
- Divide by zero/NotANumber
- Numerical precision problems
- Poles/Singularities
- Degenerate textures/texture reads, i.e. everything samples the same texcoord

- Fails to run
  - Check shader compile log
  - Right DLL’s for platform
  - Linking errors, SxS assembly problems

- Black screen
  - Texture setup, Framebuffer setup, Camera Setup
  - This is the hardest thing to fix

- Poor framerate
  - OpenGL setup
  - Excessive texture reads
  - Incoherent branches
  - Excessive trigonometric and/or matrix operations