CS562 Project 1: Deferred Shading

Synopsis

Implement deferred shading with many local lights as a first step toward a full semester project. The local light's brightness should fall off smoothly to zero with no sharp edge at the end of its range.

Instructions

Assumed existing code organization:

The assumption is that you already have an interactive graphics application with at least a minimal lighting calculation organized similar to this:

- **Application** Sets viewport to the screen size, depth-testing on, blending off. The draw-loop passes the transformations to the shader, then draws all the scene's objects, each one proceeded by sending object specific parameters (colors, textures, model transformation) to the shader.
- **Vertex shader:** Transforms the vertex to screen (NDC) coordinates. For lighting purposed, transforms the vertex and normal to world coordinates. Passes the world position and normal to the pixel shader.
- **Pixel shader:** Receives eye position, light position and brightness and surface properties from the application. Calculates lighting vectors N, L, and V. Calculates and outputs the pixel's color.

Basic deferred shading organization:

Start with this rewrite of the basic drawing and lighting code for the first required portion of this project.

G-buffer pass:

- **Application:** Same as above with the addition that a G-buffer (multiple render target, or FBO with multiple color attachments), is set up to receive the output.
- **Vertex shader:** Same as above.
- **Pixel shader:** Same start as above to retrieve the values needed for the lighting calculation, but not including the actual lighting calculation. Instead those values are output to the G-buffer. For instance:

```
out vec4 FragData[];
FragData[0].xyz = WorldPosition;
FragData[1].xyz = N;
FragData[2].xyz = Kd;
FragData[3]     = vec4(Ks, alpha);
```

Lighting pass:

- **Application:** Passes each of the render target's attachments to the shaders as textures. Draws a full-screen quad rectangle with vertices at $(\pm 1, \pm 1, 0)$.
- **Vertex shader:** No transformations necessary. For instance, in OpenGL, the full shader might be:
  in vec4 vertex; void main() {gl_Position=vertex;}
- **Pixel shader:** Receives eye position, light position and brightness from the application. Works out it's pixel position, for instance **gl_FragCoord.xy/vec2(width,height),** and recovers the G-buffer values at that pixel. Calculates lighting vectors N, L, and V. Calculates and outputs the pixel's color.

Many local lights:

The second required portion of this project adds a many-local-lights pass after the above passes.

G-buffer pass: Same as above.
Lighting pass: Same as above.
Local lights pass:

- **Application:** ***Sets depth-testing off, blending on for additive blending, and face culling on.*** Loops through all the many local lights, sending each light's *position*, *range*, and *color* to the shaders, and drawing a sphere with a modeling transformation that places it in world coordinates bounding the spherical range of the light.
- **Vertex shader:** Applies the sphere's modeling transformation, as well as the usual viewing and perspective transformations to the vertex (which comes from the sphere being drawn).
- **Pixel shader:** Works out it's pixel position, and reads quantities from the G-buffer. Checks if the world-position from the G-buffer is within the range of the light. If not outputs (0,0,0). If it is in range, uses the same lighting equation as above to calculate and output this light's contribution to the pixel. (Reminder: The additive blending choice implies this output is ADDED to the output image.) The light's brightness should be attenuated smoothly to zero at the full extend of the light's range. An easy attenuation that approximates the physically accurate $1/d^2$ falloff but guarantees a zero at the light's max range r is $1/d^2 - 1/r^2$.

Preserving a shadow map algorithm:

If your code contains a shadow-map algorithm that you wish to preserve through this project, insert the shadow-map generating pass before the lighting pass (before or after the G-buffer pass doesn't matter)

G-buffer pass: Same as above
Shadow map pass: Previous project's shadow map code
Lighting pass:

- **Application:** Sets the viewport to screen size (since the shadow-map pass certainly set it to the shadow-map size). Passes the shadow-map and the transformations used in its creation to the shaders. Otherwise the same as above.
- **Vertex shader:** Same as above .
- **Pixel shader:** Same as above with the addition that the shadow test can be performed and its results can affect the lighting calculation.

Local lights pass: Same as above.

Framework

I don't supply a framework specific to this class. I do provide a very simple interactive graphics framework that was developed for other classes (CS541, CS300, CS251) and may serve as a starting point for this class.

What to display

To demonstrate that the g-buffer is created correctly, produce a displayable image of each of the quantities stored in the g-buffer.

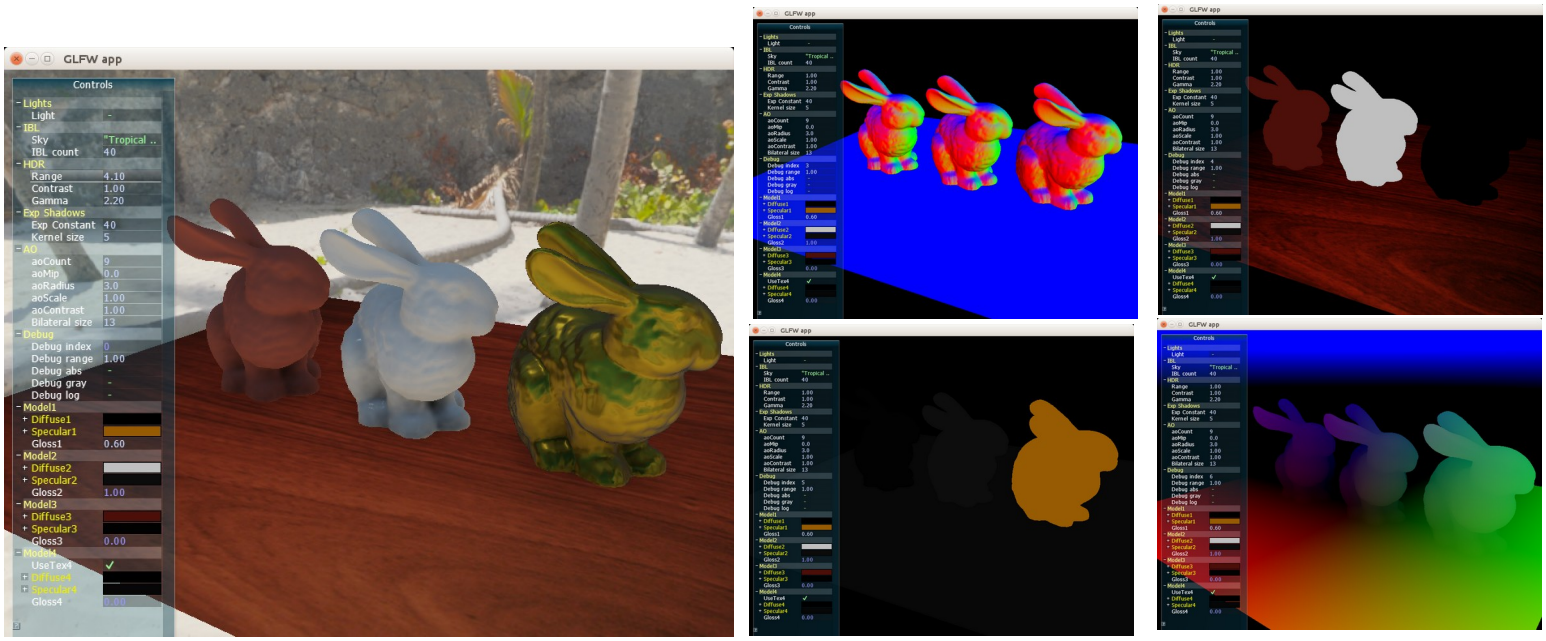
To demonstrate that the full lighting calculation works, display several images of a correctly lit scene with a range of light and eye positions.

To demonstrate that (many) local lights work correctly and efficiently, display images with as many local lights as you can while maintaining a reasonable interactive frame rate.

What to hand in

Turn in the **full code** and a **project report** that includes at least the following:

- An outline of the project, your implementation, and how you verified the results.
- The layout of your g-buffer.
- The images mentioned in the previous section with explanations of what each shows.
- A section that points to the file(s) and perhaps line numbers, where I can find the implementation of this project.
- **Note:** Your grade will be determined completely from the project report. It is important to put some quality effort into it. As an aid in this respect, my Digipen web page has three examples of very good project reports from past semesters.



Some OpenGL hints

A pixel's coordinates: vec2 pos = gl_FragCoord.xy/vec2(WIDTH,HEIGHT);

Additive blending: glBlendFunc(GL_ONE, GL_ONE); glEnable(GL_BLEND); glDisable(GL_BLEND);

Depth testing: glEnable(GL_DEPTH_TEST); glDisable(GL_DEPTH_TEST);

Set viewport: glViewport(0, 0, width, height);

Clearing FBO: glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT| GL_DEPTH_BUFFER_BIT);

Setting 4 draw buffers: After binding the FBO

```
GLenum bufs[4] = { GL_COLOR_ATTACHMENT0_EXT , GL_COLOR_ATTACHMENT1_EXT ,
                   GL_COLOR_ATTACHMENT2_EXT , GL_COLOR_ATTACHMENT3_EXT };
glDrawBuffers(4, bufs);
```

FBO encapsulation class:

```
void FBO::CreateFBO(const int w, const int h)
{
    width = w;
    height = h;

    // Generate an FBO identifier, and bind it
    glGenFramebuffersEXT(1, &fboID);
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboID);

    // Create a render buffer, and attach it to FBO's depth attachment
    unsigned int depthBuffer;
    glGenRenderbuffersEXT(1, &depthBuffer);
    glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, depthBuffer);
    glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_DEPTH_COMPONENT, width, height);
    glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
                                 GL_RENDERBUFFER_EXT, depthBuffer);

    // Create a texture and attach FBO's color 0 attachment.  The
    // GL_RGBA32F and GL_RGBA constants set this texture to be 32 bit
    // floats for each of the 4 components.  Many other choices are
    // possible.
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);
    glTexImage2D(GL_TEXTURE_2D, 0, (int)GL_RGBA32F, width, height, 0, GL_RGBA, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, (int)GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, (int)GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, (int)GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, (int)GL_LINEAR);

    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
                              GL_TEXTURE_2D, textureID, 0);

    // Check for completeness/correctness
    int status = (int)glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
    if (status != int(GL_FRAMEBUFFER_COMPLETE_EXT))
        printf("FBO Error: %d\n", status);

    // Unbind the fbo until it's ready to be used
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
}


void FBO::Bind() { glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboID); }
void FBO::Unbind() { glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0); }
```