Deferred Rendering

Bruno Deo Vergilio

The Project

The main goal of my study was to make use of a rendering technique called Deferred Rendering. It consists in rendering the objects properties, such as colors, normals, depth and others, into different GPU buffers, and later applying lighting and / or shadow effects by using all these buffers. The reason why it is called Deferred Rendering is because shader calculations are postponed and done at a later stage. There are a few advantages to this approach over Forward Rendering:

- Affected pixels are processed once per light, not per object
- Only the front most objects get lit up
- Can use an indefinite amount of lights
- No need for arrays of constant buffers in the shaders
- Properties such as depth, specular values can be reused in post-processing effects

There are also trade-offs, the most common being:

- Transparency is simply not feasible (no previous pixel state is saved)
- Multisampling now becomes difficult to implement (and very expensive)
- Increased fill-rate (shouldn't be a problem for newer graphics cards)

As time allowed, I've also managed to implement shadows through a technique called Shadow Mapping, which uses its own depth buffers, one for each light that casts shadows (in this implementation, could be done with one depth buffer per light type). There are different ways to implement this technique, and they vary according to the type of light source. An attempt was made to get directional shadows working through the cascaded shadow maps technique, but shadows flicker.

Lastly, I also tried a demo with hardware instancing, used both in the rendering of the objects as well as the light sources.

Support Systems

I've also developed supporting systems to help me create my demos; one of these systems, the math engine, is used extensively throughout my demos. Other systems that I have

created (but am not using in the current demos) are the memory system, the file system, and a renderer system, which as of now, serves as a wrapper to the DirectX 11 API.

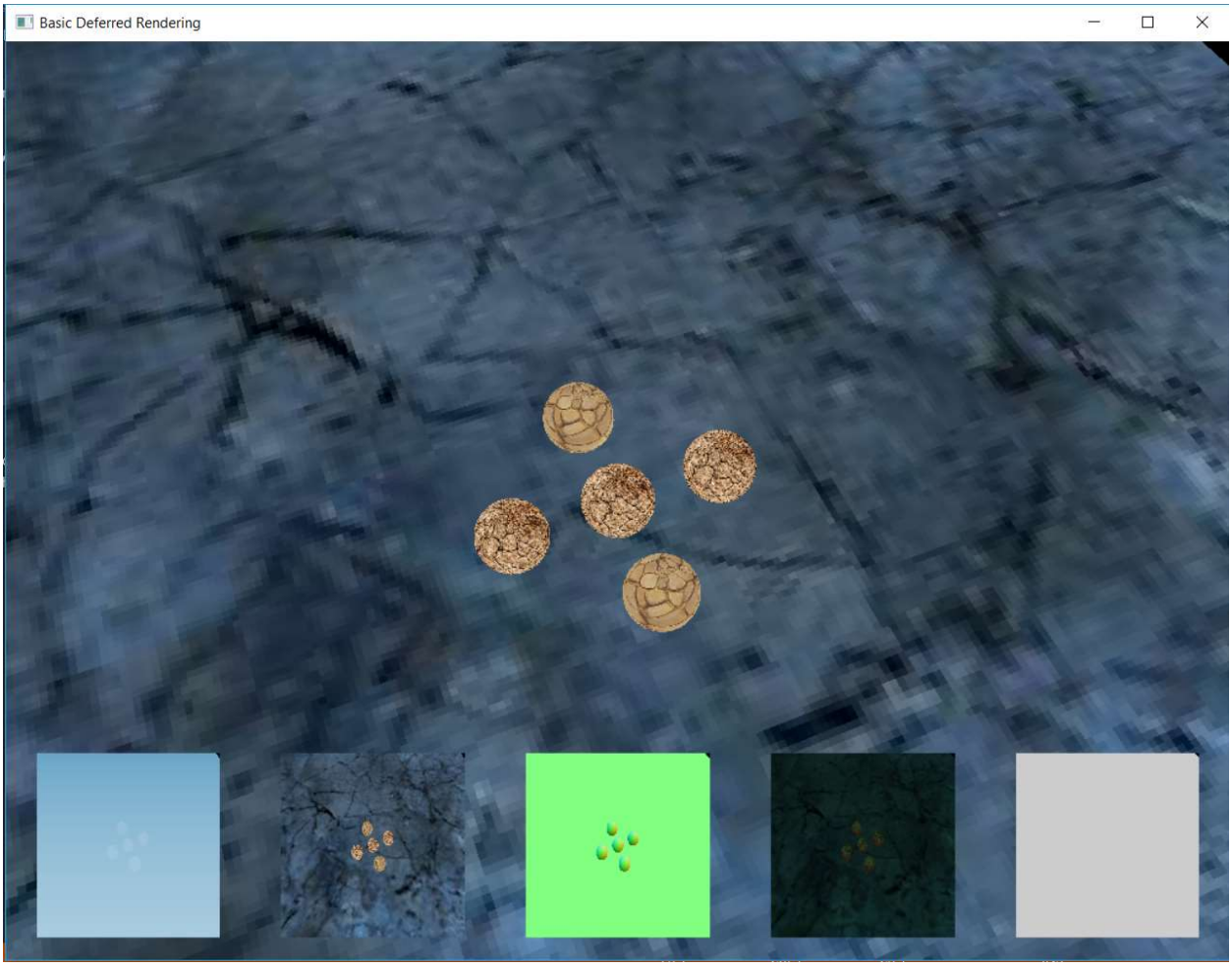
The Engine

The Engine provides classes for setting up a Deferred Renderer object, along with a GBuffer class that will be used to set up the necessary resources to render the objects to different GPU buffers. Structures used for the GBuffer are Texture2D objects, for the memory regions, Render Targets, which will allow these regions to be rendered to, and Shader Resource Views to later access these same buffers in the shaders, for lighting calculations. The light sources are now drawn as regular objects (light volumes), with screen quads representing ambient / hemispheric and directional lights, geospheres for point lights, and cones for spotlights.

The Demos

For all the demos, basic input is provided. If you hold control and mouse the mouse, the camera acts as a fps camera, and the keys W and S move it forward and backwards, while the keys A and D strafe the camera left and right.

Basic Deferred Rendering

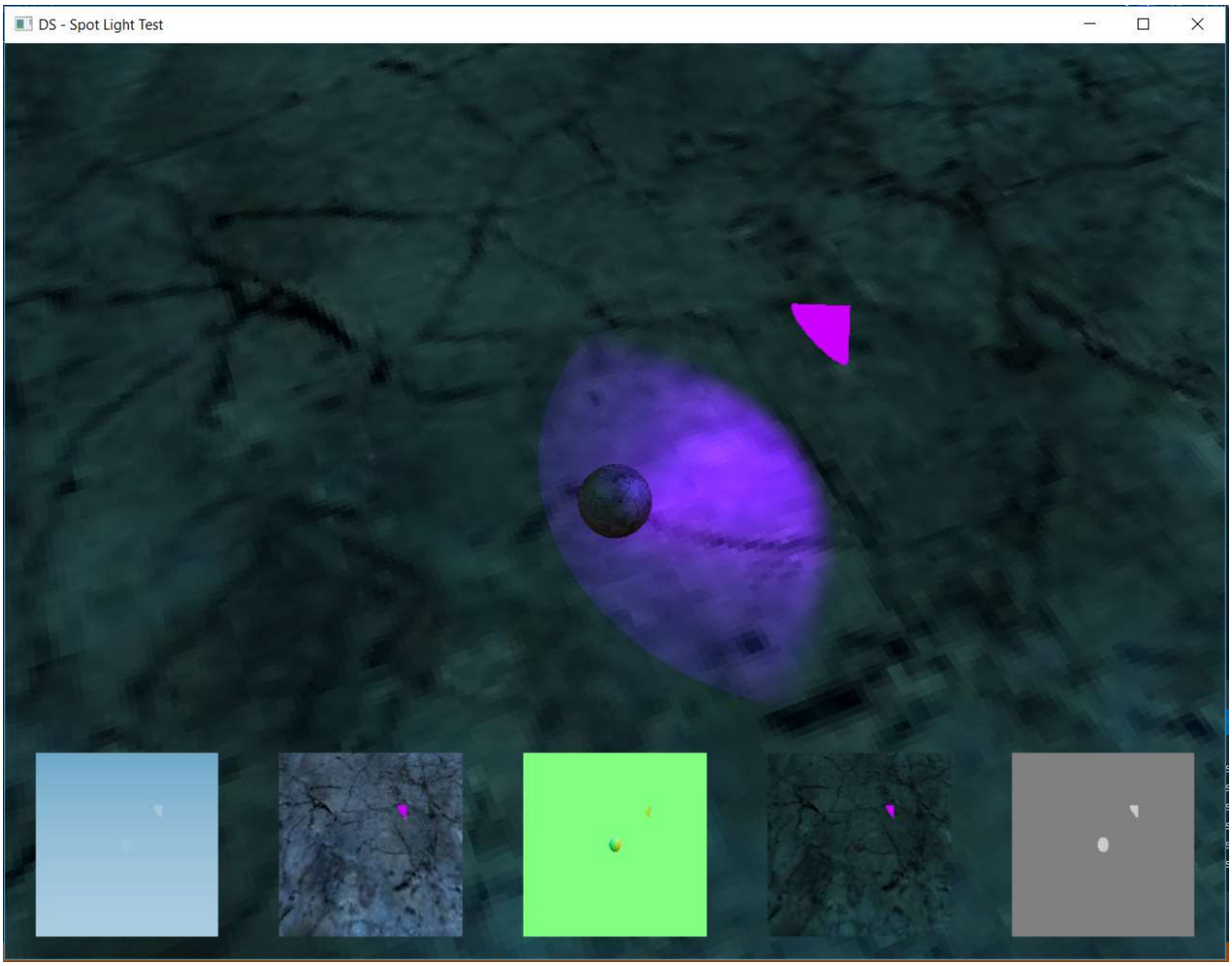


This was the first demo I set up, to demonstrate the separation of properties into multiple buffers (GBuffer creation). I have 4 GBuffers (5 if you count the Depth Buffer), which are used in the lighting pass to apply the shader calculations. These are:

- Depth Buffer
- Diffuse Buffer
- Normals Buffer
- Specular Buffer
- Emissive + Ambient (though no emissive term is being used in my demos)

Buffers can be selected and shown on the screen by pressing the 0-4 keys.

Deferred Shading Spot Light Test

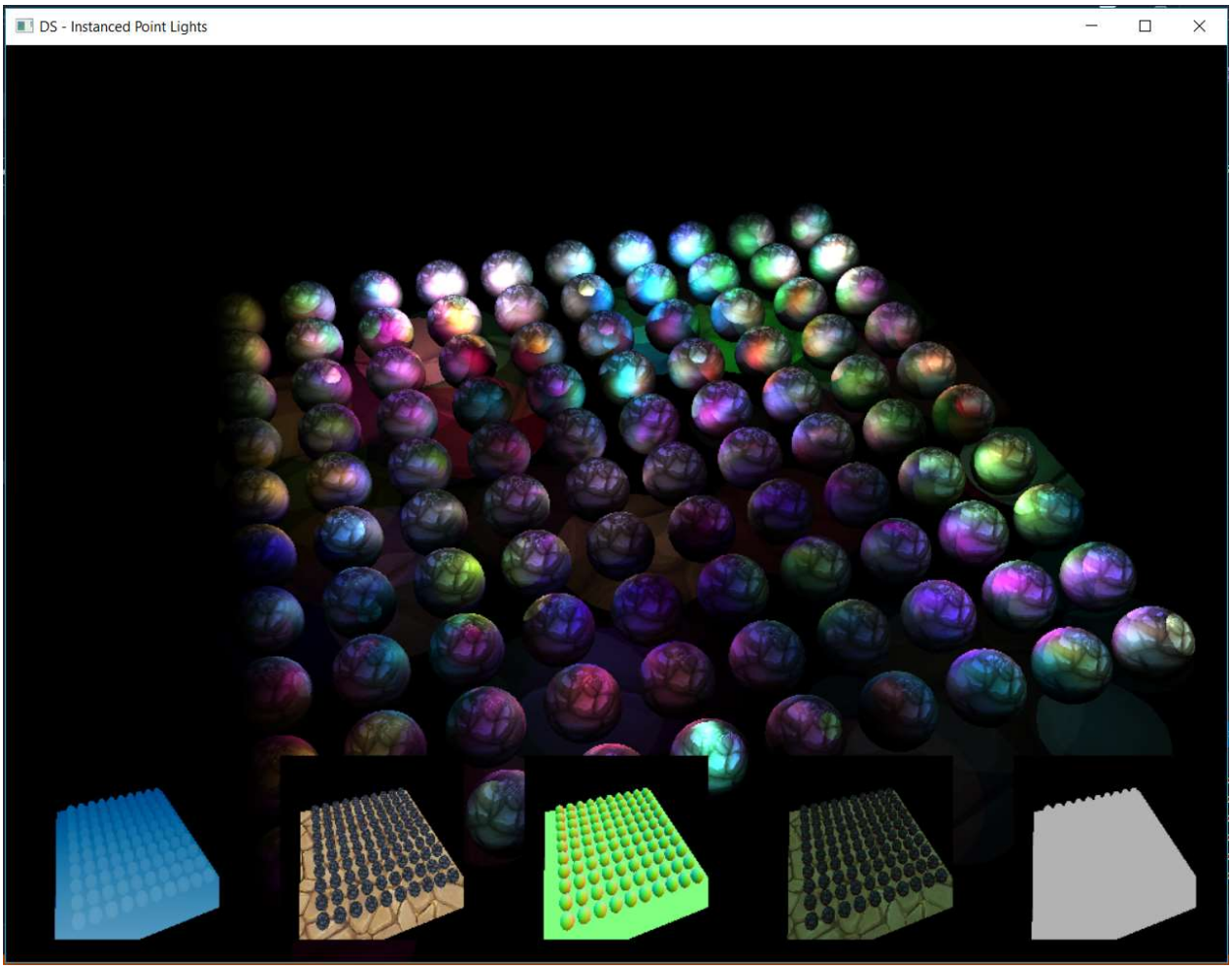


The DSSpotLightTest project uses a single spot light that half-orbits the center of the grid to demonstrate the several properties of the spot light, such as attenuation (constant, linear and quadratic), range, distance and intensity (optional).

By pressing “K”, “L” or “;”, attenuation values can be changed; pressing “I” will change the intensity, “R” the range, and “D” the distance from the light source to the grid. Holding CTRL while pressing one of these keys will do the reverse.

The idea of this demo was to previously debug spot lights, which were more complex to implement and required more time spent on it. This demo extends on the previous one by now applying the light pass, making use of all 5 buffers, which can be seen as thumbnails on the bottom part of the screen.

Instanced Point Lights

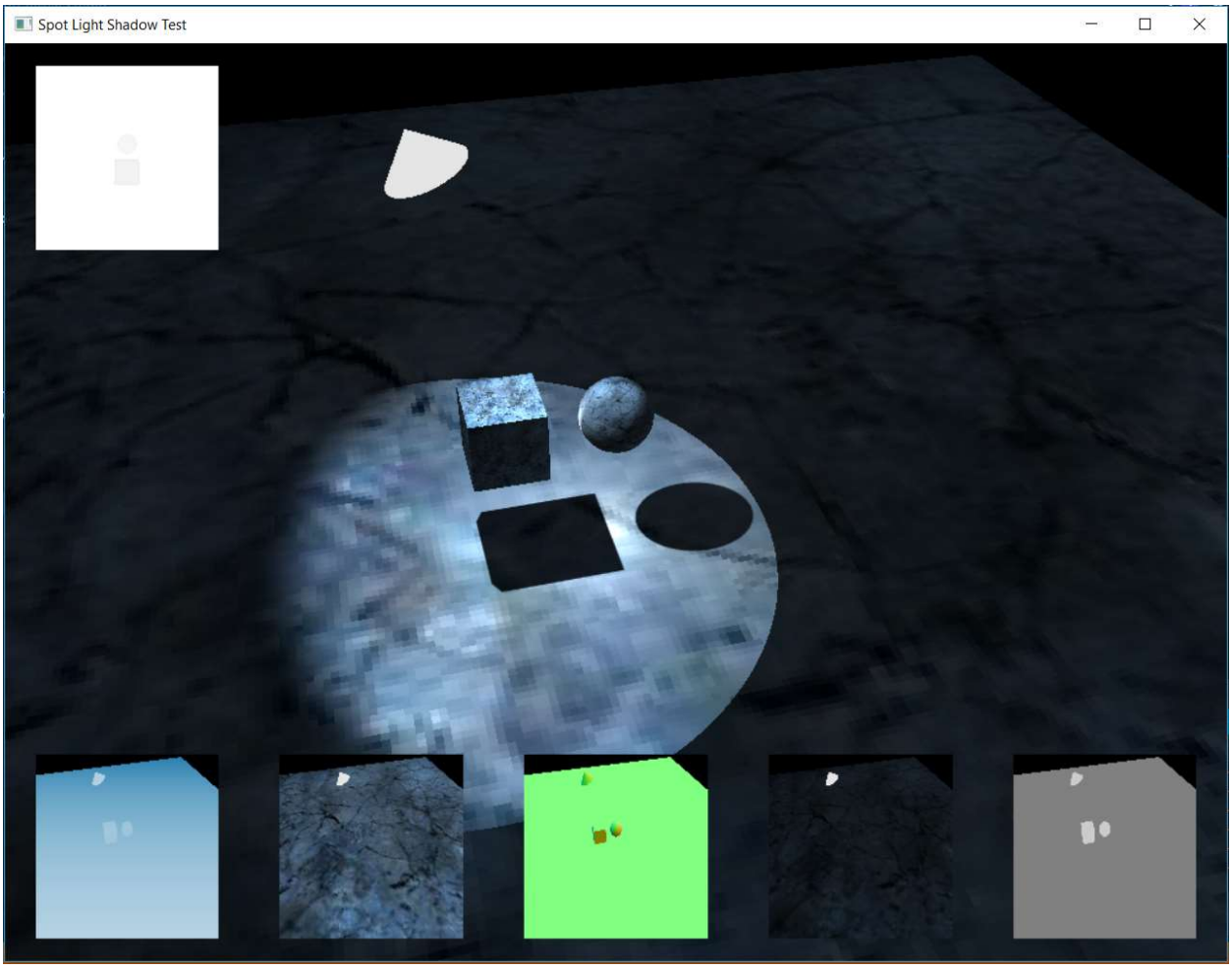


Hardware Instancing was a technique I wanted to not only learn, but also try to apply to light volumes to see if it would work with them as well. It would require a second vertex (or instance) buffer, quite big, due to the number of properties the light structure had, and accounting for the fact that even though not all attributes were used by all the lights, they still shared one single structure to represent all 3 types.

The idea was to render several objects on the screen, and then do the same with the light volumes; however, only a single type of light source would be used, since they are all light volumes, and instancing makes use of the same model – therefore, point lights were used. This demo renders 100 spheres on a grid, through hardware instancing, and then applies the same technique to the light volumes, rendering 200 point lights, with random distances and colors, and intensities going up a down each frame.

I personally liked this demo a lot, and it seemed like the best one to showcase not only showcase the power of hardware instancing, but actually point out the best advantage of deferred rendering: high number of light sources, which allows for more complex scenes.

Spot Light Shadows



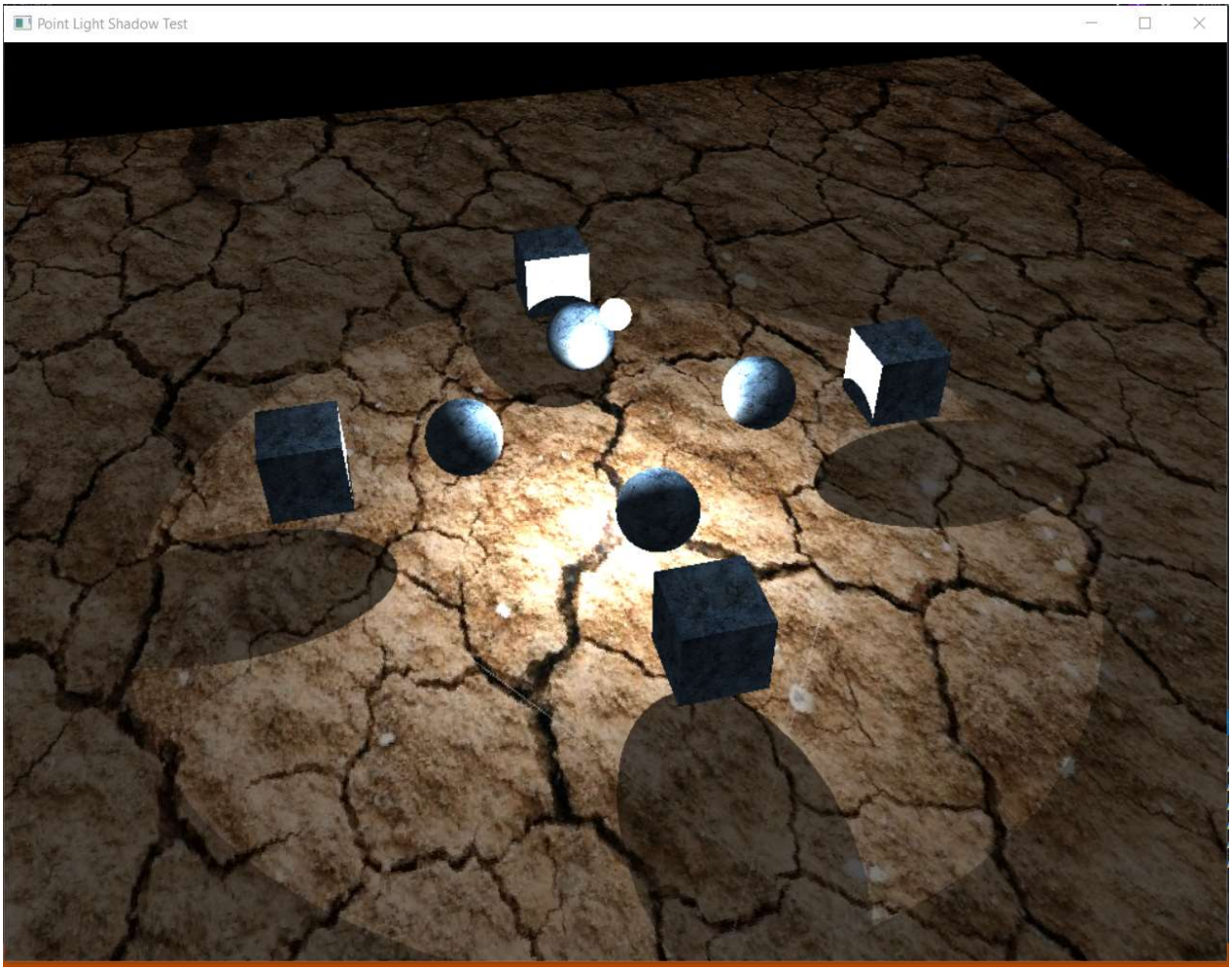
This sample further extends from the spot light test demo, adding shadows to spot light. The keys “M” and “N” will pause and unpause the light source, and “C” and “V” will enable or disable the shadows.

The idea with shadow mapping is that the objects with shadows must be rendered once for in each light that casts shadows in double speed Z mode (DirectX mode with no color writes, which mean binding only a vertex shader and leaving a null pixel shader) to a special depth buffer. These objects, however, will make use of a different View / Projection matrix, which represent the light that will cast the shadows – this basically means the light source is now a camera.

This happens so that in the pixel shader, the depth buffer representing the light source will contain the Z values for the closest objects on the screen; these values will be compared to the camera’s Z values, and if those are greater than the light’s shadow map values, then there is shadow being cast on that given pixel.

The process is straight forward when working with Spot Lights, getting more complex with Point Lights and even further with Directional Lights.

Point Light Shadows



Point light shadows complicates things, because now there is an omni-directional light source, which means light will emanate from the source equally to all directions, while spot lights had a cone-like shape for lighting up objects. One approach is to render the objects six times, using 6 different matrices, all covering a cone of 90 degrees each, however that would be overkill if you had too many spot lights casting shadows.

The approach I took makes use of the Geometry Shader to replicate all the vertices drawn 6 times, in a single draw call. In there, 6 View / Projection matrices are passed in, covering all the directions. The problem is that now a regular texture object won't cover everything, so a cubemap texture object will have to be used, where each face represents a replicated set of the objects drawn, but seen from a different perspective each time. Therefore, 6 view matrices will be created, each one looking at a different direction (+X, -X, +Y, -Y, +Z, -Z), and one 90-degree field-of-view projection matrix, which will be used by all 6 view matrices.

When processing the depth on the shader, the Z values from the world position now must be checked against the correct face of the cube map, since the point light is covering all

directions in a given range. Problem is, they have to be checked against a specific cube face, so in order to correctly choose the face that covers the same area the object is in, a vector that goes from the light source to the pixel's position is created, and used to sample from the cubemap. That will get the correct cubemap's face.