MTAT.03.015 Computer Graphics (Fall 2013)
Exercise session IX: Shadows

Konstantin Tretyakov, Ilya Kuzovkin

November 4, 2013

The standard graphics pipeline is a very efficient algorithm for rendering complex scenes – this is the reason why it is the primary (if not the only) method used nowadays in high-performance graphics, such as games or interactive visualization.

However, as the scene in this algorithm is rendered on a strictly "local" polygon-by-polygon basis, implementing any effects related to global light propagation, such as shadows, reflections, transparency and various kinds of light scattering from one object to another, is increasingly hard or cumbersome. In this exercise session we shall study the three most popular algorithms for implementing shadows in the standard graphics pipeline framework. While working on them, think how each of them solves the problem of introducing "global" light interaction into an otherwise purely local approach in its own way.

As usual, the code base is provided in `practice09.zip` archive on the course website or via the course github page. Download, unpack and open it. You will need to submit your solution as a zipped archive file.

1 Projective Shadows

Open the project `1_ProjectiveShadows` and study the code. The main file of the project is `projective_shadows.cpp`, which presents the (now hopefully very familiar to you) structure of a GLUT application. The `display` function renders a simple scene with three "walls" (a floor and two walls, to be more precise), two "objects" (which are actually cubes) and two light sources. To simplify the code, some of the functionality has been separated into the file `scene_primitives.cpp`. You should not need to look much into that file, in principle, but you should at least study the `scene_primitives.h` header to understand what it provides.

Exercise 1 (0.5pt). Complete the `display` function by adding code that renders projection shadows for both cubes from both light sources onto the walls and the plane. An example is provided to get you started. Note that your main tool here is the `shadowMatrix(light, plane, offset)` function (provided in `scene_primitives.h/.cpp`). This function constructs a projection matrix with `light` as the center of projection, and `plane` as the projection plane. Multiplication by this matrix "squishes" all primitives onto the projection plane. The projection plane is specified as a four-component vector (a, b, c, d). This vector provides the coefficients of the implicit plane equation

$$ax + by + cz + d = 0,$$

For example, for the floor plane $(z = 0)$ the implicit equation is

$$0x + 0y + 1z + 0d = 0,$$

i.e. the corresponding vector is `{0, 0, 1, 0}`. You will need to provide plane equations for the right wall $(x = 4)$ and the back wall $(y = 4)$ as well.

Finally, the `offset` parameter adds a tiny offset to the "squished" object shadow along the normal to the plane (a, b, c). This prevents depth-buffer related artefacts (set offset to 0 to observe them).

Note that despite their overall conceptual simplicity, projective shadows are quite limited in what you can achieve with them: objects do not cast shadows on each other, and the computation scales linearly with the number of planes that should receive shadows.

A popular version of the same idea is that of *projective textures*[1], where some texture image is projected onto geometry from the light source center (like a projector would)[2]. If the texture contains pre-rendered shadow silhouettes this can be used as a simple shadow rendering algorithm (sometimes referred to as *projective texture shadows*).

2 Stencil Shadows

A much better algorithm for real-time shadows is the *stencil shadow* algorithm, most well known for its use in the game Doom III. The idea of the algorithm is quite well described in the corresponding Wikipedia article[3].

Exercise 2 (1pt). Open the project `2_StencilShadows`. It has the same structure and uses the same geometry as the previous exercise. Modify the `display` function to implement the stencil shadow algorithm. The main complication of the algorithm is the rendering of the shadow volumes, but this functionality is already provided for you in the `Cube::draw_shadow_volume()` function. Follow the instructions provided in the comments in the code to complete the algorithm.

[1] `http://en.wikipedia.org/wiki/Projective_texture_mapping`

[2] Note that the `GL_EYE_LINEAR` texture coordinate generation mode with default parameters, that you had a chance to briefly see in the last practice session, is essentially an orthogonal projection of the texture from the viewer onto the objects in front of him.

[3] `http://en.wikipedia.org/wiki/Shadow_volume`

Exercise 3* (0.5pt). You should note that the shadows you obtained in the last exercise are actually incorrect: shadows from two different light sources are “merged” into an equally gray area, whereas in reality the region that is in shadow with respect to both light sources must be darker than the one where just one light source is seen. Indeed, a correct implementation of stencil shadows requires preparing a separate stencil buffer for each light source, rendering just the lighting component for that particular light source, and aggregating the resulting pixel values using blending or accumulation buffer. Implement this.

Hint: You might want to sacrifice sRGB-conversion for the sake of simplicity of implementation. In other words, a solution without proper gamma correction is acceptable for sake of grading in this exercise.

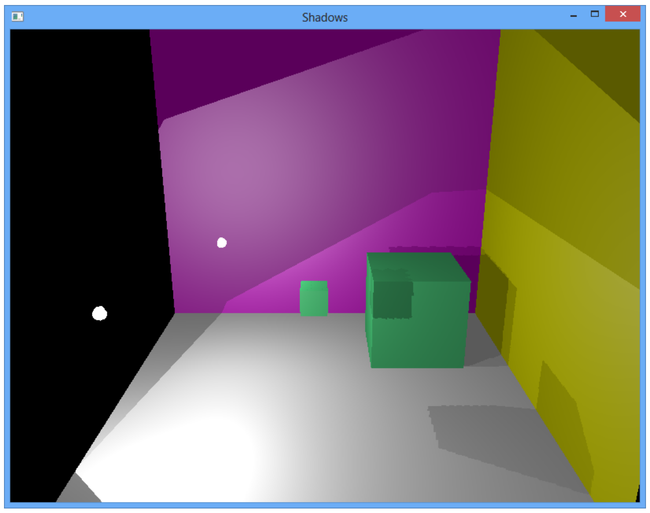
3 Shadow Mapping

Finally, probably the most popular way of implementing real-time shadows nowadays is *shadow mapping*. The idea is to render the scene from the “viewpoint of the light source”, saving the resulting depth buffer into a texture. That texture can then be used to check for any pixel whether it is occluded from the light source or not. Again, a fairly clear description of the technique is provided in Wikipedia[4].

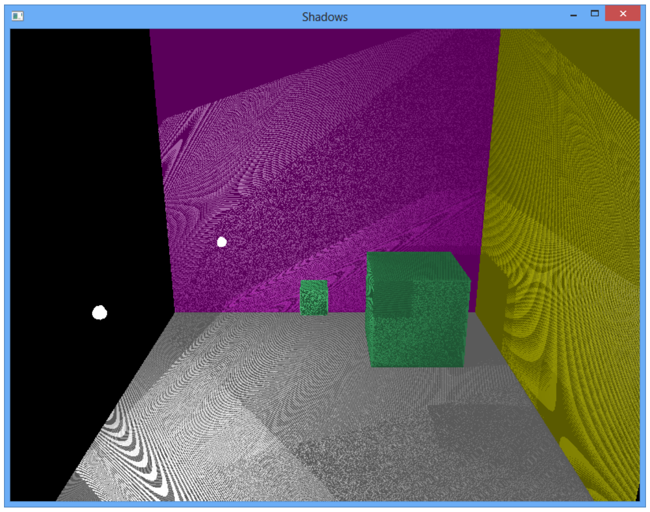
Exercise 4 (2pt). Open the project `3_ShadowMaps` and study the `shadow_maps.cpp` file (in particular the `display` function). The core shadow mapping algorithm logic is already implemented there. Your task is to complete the implementation by providing the appropriate fragment shader code in `shadowmap.frag.glsl`. The comments in the fragment shader will hopefully provide enough guidance.

Note that due to the logic of the algorithm your light sources will act as a kind of “square” directional lights. You should expect to get a result like the following:

[4] `http://en.wikipedia.org/wiki/Shadow_mapping`



Most probably your first working implementation will produce an artifact known as "shadow acne"[5]:



It stems from having code like the following somewhere in your shader:

```
if (current_depth_value <= texture_depth_value)
    c += blinn(gl_LightSource[i]); // The fragment is lit
```

The artifact is caused by insufficient precision of the depth buffer. Whenever the surfaces are lit, the values of `current_depth_value` and `texture_depth_value` should in principle be strictly equal. However, due to the limited precision of

[5]`http://msdn.microsoft.com/en-us/library/windows/desktop/ee416324(v=vs.85).aspx`

floating point values, the equality is rarely exact: for some pixels one value is slightly greater, for other slightly less, which leads to the observed effects.

The fix is simple: add a small offset to the pixel's depth.

```
if (current_depth_value - offset <= texture_depth_value)
    c += blinn(gl_LightSource[i]); // The fragment is lit
```

When the offset is too large, however, gaps will start to appear between the objects and their shadows, and you don't want this to happen either.

Coming up with a solution to this exercise is a good indicator that you have mastered the basic concepts behind the standard graphics pipeline.

Exercise 5* (0.5pt). For even nicer shadow mapping, it is possible to use GPU-s built-in capabilities for working with shadow maps. This will result in both cleaner code and better texture sampling, somewhat reducing aliasing artifacts. Try doing it:

1. Change the type of `shadowMapTexture`k variables from `sampler2D` to `sampler2DShadow`.

2. Now, rather than accessing them as textures via the `texture2D` function, you will need to use the `shadow2D` function, which takes as input the shadow map texture and a three-dimensional point. It performs the shadow test (using (x, y) to look up the depth value in the texture and z to compare against), and returns nonzero value(s) if the shadow test passes, i.e. the corresponding point is not in shadow.

3. Finally, there is also a `shadow2DProj` function, which directly accepts four-dimensional homogeneous coordinates (so that you do not need to perform perspective division). You cannot immediately feed clip-space coordinates to it, however. As all other texture access functions, it expects coordinates scaled to texture-space range, i.e. x, y, z must be in range $[0, 1]$ rather than $[-1, 1]$. The rescaling from clip-space homogeneous coordinates into the texture-space range can be done using the matrix

$$\begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Have the shader use the `shadow2DProj` function for shadow mapping to get points for this task.