

Shadow Mapping

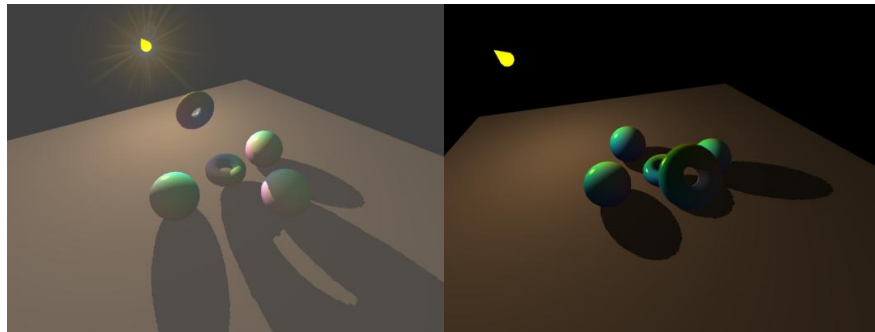
Realtime shadow rendering with OpenGL ES 2.0

Sylwester Bala
Stacy Smith

February 2013

OBJECTIVE

This document describes an approach for rendering shadows in realtime for mobile devices using OpenGL ES 2.0, which does not support depth texture unless the OES_depth_texture extension is available.



OVERVIEW

This document describes a realtime rendering shadows approach using a projective texture mapping technique. The technique does as many rendering passes as the number of lights, plus one final pass to draw the objects with shadows on top of them. Lets call these passes Light and Normal respectively. To keep the document as simple as possible the number of lights has been confined to one. Having only one light, two passes are required: one Light and one Normal pass. The distance from a given fragment to the light is calculated and stored in a texture during the Light pass. It is usually considered as lightweight pass since all rendering features but distance measurement are disabled. The distance is stored in an RGBA texture on the RG components as the two components are sufficient to get enough precision for this example.

Once the distance information has been collected, it is time to do the Normal pass where the scene is rendered from camera position, during which shadow is applied on fragments that are obscured by other fragments on their way to the light.

LIGHT PASS

This section explains how to prepare a shadow texture (distance texture). The most efficient way of creating the texture is to use a Frame Buffer Object, which is available in OpenGL ES 2.0. First, create the texture as follow:

```
glGenTextures(1, &theNameTexture);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, theNameTexture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, theWidth, theHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Now create the FBO and attach the above texture:

```
glGenFramebuffers(1, &theNameFBO);
glGenRenderbuffers(1, &theNameRenderBuffer);
glBindRenderbuffer(GL_RENDERBUFFER, theNameRenderBuffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT16, theWidth, theHeight);
glBindFramebuffer(GL_FRAMEBUFFER, theNameFBO);
glBindTexture(GL_TEXTURE_2D, theNameTexture);
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
                        GL_DEPTH_ATTACHMENT,
                        GL_RENDERBUFFER,
                        theNameRenderBuffer);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, theNameTexture, 0);
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
{ printf("Error: FrameBufferObject is not complete!\n"); }
```

The above creation of FBO as well as texture should be done only once during an application initialization. Once the FBO has been successfully created, there is only one function left to be called to change rendering destination from FBO to framebuffer and vice versa, which is `glBindFramebuffer`:

Binding FBO:

```
glBindFramebuffer(GL_FRAMEBUFFER, theNameFBO);
```

Binding framebuffer - unbinding FBO:

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

NORMAL PASS

Having the FBO set up and ready, the scene can be rendered to it from the Light position in order to collect information about distance of fragments to the Light. This can be done by outputting a distance

value instead of color from a fragment program. Below is an example of how the vertex and fragment programs could be written to output the distance:

Vertex program, which passes the screen position of a vertex to the fragment program:

```
#version 100

attribute vec4 a_v4Position;
uniform mat4 u_m4LightMVP;
varying vec4 v_v4TexCoord;

void main()
{
    v_v4TexCoord = u_m4LightMVP * a_v4Position;
    gl_Position = u_m4LightMVP * a_v4Position;
}
```

The Fragment program receives the position and packs it to RG components in order to increase precision. The assumption has been taken that the range of packing values must be in [-10, 10]. Values outside of this range will cause incorrect rendering results:

```
#version 100

precision mediump float;
varying vec4 v_v4TexCoord;
void main()
{
    /* Generate shadow map - write fragment depth. */
    float value = 10.0 - v_v4TexCoord.z;
    float v = floor(value);
    float f = value - v;
    float vn = v * 0.1;
    gl_FragColor = vec4(vn, f, 0.0, 1.0);
}
```

After the scene has been rendered to the FBO with the above shaders, the distance texture is ready to use. This is enough information to display shadows, which is what the next section is about.

NORMAL PASS

Once the distance texture is ready, the FBO has to be unbound and the scene rendered directly to the framebuffer with all necessary features enabled such as lighting, texturing, blending etc. During this pass the shadows can be applied as well. It is possible to do an extra pass just for applying shadows but we want to avoid it since we have enough room for shadows in the same pass. To render the scene the camera has to be reset to its original position, the distance texture has to be bound as a 2D texture and all shaders must be modified to do shadows since the shadows are being rendered in the same pass as normal rendering. Below is shader code to show how the shaders should be modified:

Vertex shader:

```
#version 100

attribute vec4 a_v4Position;

uniform mat4 u_m4MVP;
uniform mat4 u_m4M;
uniform mat4 u_m4LightV;
uniform mat4 u_m4LightP;

varying vec4 v_v4TexCoord;

void main()
{
    const mat4 biasMat = mat4(0.5, 0.0, 0.0, 0.0,
                              0.0, 0.5, 0.0, 0.0,
                              0.0, 0.0, 1.0, 0.0,
                              0.5, 0.5, 0.0, 1.0);

    /* Calculate vertex position, which is being seen from the light. */
    v_v4TexCoord = u_m4LightP * u_m4LightV * u_m4M * a_v4Position;

    /* Normalize texture coords from -1..1 to 0..1 now, before projection. */
    v_v4TexCoord = biasMat * v_v4TexCoord;

    ...
    /* Here can be done some other computations like: lighting, coordinates, clipping etc. */
    ...

    gl_Position = u_m4MVP * a_v4Position;
}
```

Fragment shader:

```
#version 100

uniform sampler2D u_s2dShadowMap;

varying vec4 v_v4TexCoord;
```

```
void main()
{
    /* Draw main scene - read and compare shadow map. */
    vec2 vfDepth = texture2DProj(u_s2dShadowMap, v_v4TexCoord).xy;
    float fDepth = (vfDepth.x * 10.0 + vfDepth.y);

    /* Unpack the light distance. See how it is packed in the shadow.frag file */
    float fLDepth = (10.0-v_v4TexCoord.z) + 0.1 - fDepth ;
    float fLight = 1.0;
    if(fDepth>0.0 && fLDepth<0.0)
    {
        fLight = 0.2;
        /* Make sure there is no specular effect on obscured fragments */
        intensitySpecular = 0.0;
    }

    ...
    /* Here can be done some other computations like: lighting, texturing, fog etc. */
    ...

    gl_FragColor = vec4(FinalColor.rgb * fLight, FinalColor.a);
}
```

From the above shaders one can see that the vertex position is literally calculated twice; once in lighting space and once in camera space. First the vertex is being calculated in the light space (as the camera would be looking at the scene from the light source). This is the same way of doing transformation as usually it is done for vertices. The vertex is being multiplied by the light transformation matrix instead of a camera.

```
/* Calculate vertex position, which is being seen from the light. */
v_v4TexCoord = u_m4LightP * u_m4LightV * u_m4M * a_v4Position;
```

Where:

u_m4LightP	A projection matrix, which should be exactly the same as the camera's
u_m4LightV	A view matrix from the light position and orientation
u_m4M	A model matrix, transformation (position, rotation, scale) of an object of which the current vertex is being calculated
a_v4Position	A vertex position in the current object

Every time the vertex is calculated according to the above formula, the position is eventually represented in NDC (Normalized Device Coordinates) space, which is in the range [-1, 1]. Since the fragment distance is going to be read from the shadow texture the coordinates should be in the range of [0, 1]. For this reason we have to multiply the X and Y values by 0.5 and then add 0.5 to each. This transformation is shown below:

```
/* Calculate vertex position, which is being seen from the light. */
v_v4TexCoord = u_m4LightP * u_m4LightV * u_m4M * a_v4Position;

/* Normalize texture coords from -1..1 to 0..1 now, before projection. */
const mat4 biasMat = mat4(0.5, 0.0, 0.0, 0.0,
                          0.0, 0.5, 0.0, 0.0,
                          0.0, 0.0, 1.0, 0.0,
                          0.5, 0.5, 0.0, 1.0);
v_v4TexCoord = biasMat * v_v4TexCoord;
```

The final position, which is already in [0, 1] range, is passed to the fragment shader. The fragment shader fetches the distance from the shadow texture and compares it to the actual distance to the light. If the fetched distance is shorter, the fragment is in shadow. Otherwise, it is out in the open and lit by the light. Below is a snippet of code showing the distance comparison in the fragment program:

```
/* Fetch and unpack the light distance from the shadow texture. */
vec2 vfDepth = texture2DProj(u_s2dShadowMap, v_v4TexCoord).xy;
float fDepth = (vfDepth.x * 10.0 + vfDepth.y);

/*Convert the actual distance in the same manner as it is stored in the shadow texture*/
float fLDepth = (10.0-v_v4TexCoord.z) + 0.1 - fDepth ;

/* First assume the fragment is not in shadow */
float fLight = 1.0;
if(fDepth>0.0 && fLDepth<0.0)
{
    /* Now the fragment is definitely in shadow */
    fLight = 0.2;
}
```

REFERENCES:

- Shadow Mapping, <http://www.paulsprojects.net/tutorials/smt/smt.html>
- Shadow Mapping, http://en.wikipedia.org/wiki/Shadow_mapping