Shadow Mapping Algorithm for Android*

"There is no light without shadows" - Japanese proverb

Because shadows in games make them more realistic and interesting, including well-rendered shadows in your games is important. Currently, most games do not have shadows, but this situation is changing. In this paper we will discuss a common method for realizing shadows, called Shadow Mapping.

**Shadow Mapping Theory**

Shadow mapping is one of the most conventional techniques for shadow generation in real-time applications. The method is based on the observation that whatever can be seen from the position of the light source is lit; the rest is in the shade. The principle of this method consists in comparing the depth of the current fragment in the reference system associated with the light source to that which is closest to the light source geometry.

The algorithm consists of just two stages:
1. The shadow map generation
2. The rendering stage

The algorithm’s main advantage is that it is easy to understand and implement. Its disadvantages include that it requires more CPU and GPU resources and calculations to make a picture more real. As a result of these additional resources, the shadow map in the depth buffer could become slower.

**Algorithm Realization**

To create a shadow map, it is necessary to render the scene from the position of the light source. Thus, we obtain the shadow map in the depth buffer, which contains depth values closest to the light source geometry. This approach has the advantage of speed, since the depth buffer generation algorithm is implemented in the hardware.

At the final stage, rendering occurs from the camera position. Each point of the scene is translated into the coordinate system of the light source, and then we calculate the distance from this point to the light source. Calculated distance is compared with the value that is stored in the shadow map. If the distance from the point to the light source is more than the value stored in the shadow map, then this point is in the shadow of any object placed in the path of light.

The code in this article uses the Android SDK (ver. 20) and the Android NDK (ver 8d). It is taken as the basis for a fully native application:

The Android MegaFon Mint* smartphone is based on the Intel® Atom™ processor Z2460:
http://download.intel.com/newsroom/kits/ces/2012/pdfs/AtomprocessorZ2460.pdf
Initialization

The shadow map is stored in a separate texture format GL_DEPTH_COMPONENT, size 512x512 (shadowmapSize.x = shadowmapSize.y = 512), 32 bits per texel (GL_UNSIGNED_INT). In order to optimize, you can use 16 bit textures (GL_UNSIGNED_SHORT). Creating a texture is possible on devices supporting GL_OES_depth_texture [for documentation see http://www.khronos.org/registry/gles/extensions/OES/OES_depth_texture.txt]. The parameters of GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T are set in GL_CLAMP_TO_EDGE. So when you request any value outside the texture sampling mechanism (sampler), a value corresponding to the boundary is returned. This is done to reduce artifacts from the shadows in the final rendering stage. "Tricks with the fields" will be discussed in another blog.

```
//Create the shadow map texture
glGenTextures(1, &m_textureShadow);
glBindTexture(GL_TEXTURE_2D, m_textureShadow);
checkGlError("bind texture");
// Create the depth texture.
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, shadowmapSize.x, shadowmapSize.y, 0, GL_DEPTH_COMPONENT, GL_UNSIGNED_INT, NULL);
checkGlError("image2d");
// Set the textures parameters
glTexParameteri(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_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
// create frame buffer object for shadow pass
glGenFramebuffers(1, &m_fboShadow);
glBindFramebuffer(GL_FRAMEBUFFER, m_fboShadow);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, m_textureShadow, 0);
checkGlError("shadowmaptexture");
status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status != GL_FRAMEBUFFER_COMPLETE) {
   LOGI("init: ");
   LOGI("failed to make complete framebuffer object %x
", status);
}
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

The next initialization phase is the preparation of shaders. Below is the Vertex shader attribute. This is the next step of generating shadow maps:

```
vec3 Position;
uniform mat4 Projection;
uniform mat4 Modelview;

void main(void)
{
   gl_Position = Projection * Modelview * vec4(Position, 1);
}
Pixel shader (step shadow generation):

```glsl
highp vec4 Color = vec4(0.2, 0.4, 0.5, 1.0);

void main(void)
{
    gl_FragColor = Color;
}
```

The main task of shaders is to write the geometry, or in other words, to generate the depth buffer for the main stage.

**Stages of shadow map rendering**

These steps differ from the usual stages of the vectorization scene by the next few points:

1. **FBO**, which acts as our depth buffer, is attached to the texture (shadow map)
   ```c
   glBindFramebuffer(GL_FRAMEBUFFER, m_fboShadow);
   ```
2. You can render shadows using orthographic projection from directional sources (the sun), or from a conical (omni) perspective. In the example, the chosen perspective projection matrix `lightProjectionMatrix` has a wide viewing angle—90 degrees.
3. The color entry in the frame buffer is from the `glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE)`. This optimization can be very useful if you use a complex pixel shader.
4. At this stage, the map is drawn only for the rear surface of the polygons, `glCullFace(GL_FRONT)`. This is one of the most effective and easiest methods to reduce the negative effects of artifacts on the shadow map method. (Note: this is not useful for all geometries.)
5. Area will draw 1 pixel on each side is smaller than the shadow map `glViewport(0, 0, shadowmapSize.x - 2, shadowmapSize.y - 2)`. This is done in order to leave the "field" on the shadow map.
6. After drawing all the elements of the scene, we return to its original value `glCullFace(GL_BACK)`, `glBindFramebuffer(GL_FRAMEBUFFER, 0)` and `glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE)`. 

```c
void RenderingEngine2::shadowPass() {
    GLenum status;
    glEnable(GL_DEPTH_TEST);
    glBindFramebuffer(GL_FRAMEBUFFER, m_fboShadow);
    status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
    if (status != GL_FRAMEBUFFER_COMPLETE) {
        LOGE("Shadow pass: ");
        LOGE("failed to make complete framebuffer object %x\n", status);
    }
    glClear(GL_DEPTH_BUFFER_BIT);
    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

    lightProjectionMatrix = VerticalFieldOfView(90.0,
        (shadowmapSize.x + 0.0) / shadowmapSize.y, 0.1, 100.0);
    lightModelviewMatrix = LookAt(vec3(0, 4, 7), vec3(0.0, 0.0, 0.0), vec3(0, -7, 4));
    glCullFace(GL_FRONT);
}
```
glUseProgram(m_simpleProgram);

glUniformMatrix4fv(uniformProjectionMain, 1, 0,
                   lightProjectionMatrix.Pointer());

glUniformMatrix4fv(uniformModelviewMain, 1, 0,
                    lightModelviewMatrix.Pointer());

glViewport(0, 0, shadowmapSize.x - 2, shadowmapSize.y - 2);

GLsizei stride = sizeof(Vertex);
const vector<ObjModel>& objects = m_Scene.getModels();
const GLvoid* bodyOffset = 0;
for (int i = 0; i < objects.size(); ++i) {
  lightModelviewMatrix = objects[i].m_Transform * LookAt(vec3(0, 4, 7), vec3(0.0,
                                                            0.0, 0.0), vec3(0, -7, 4));
  glUniformMatrix4fv(uniformModelviewMain, 1, 0,
                      lightModelviewMatrix.Pointer());
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, objects[i].m_indexBuffer);
  glBindBuffer(GL_ARRAY_BUFFER, objects[i].m_vertexBuffer);
  glVertexAttribPointer(attribPositionMain, 3, GL_FLOAT, GL_FALSE, stride,
                        (GLvoid*) offsetof(Vertex, Position));
  glEnableVertexAttribArray(attribPositionMain);
  glDrawElements(GL_TRIANGLES, objects[i].m_indexCount,
                 GL_UNSIGNED_SHORT, bodyOffset);
  glDisableVertexAttribArray(attribPositionMain);
}

glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

glBindFramebuffer(GL_FRAMEBUFFER, 0);

glCullFace(GL_BACK);

Rendering scenes with shadows

The first stage of this specific feature is to set textures with the shadow map obtained in the previous step:

```cpp
void RenderingEngine2::mainPass() {    
    glClearColor(0.5f, 0.5f, 0.5f, 1);    
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);    
    modelviewMatrix = scale * rotation * translation * LookAt(vec3(0, 8, 7), vec3(0.0, 0.0, 0.0), vec3(0, 7, -8));    
    lightModelviewMatrix = LookAt(vec3(0, 4, 7), vec3(0.0, 0.0, 0.0), vec3(0, -7, 4));    
}
```

projectionMatrix = VerticalFieldOfView(45.0, (screen.x + 0.0) / screen.y, 0.1, 100.0);
mat4 offsetLight = mat4::Scale(0.5f) * mat4::Translate(0.5, 0.5, 0.5);
mat4 lightMatrix = lightModelviewMatrix * lightProjectionMatrix * offsetLight;
`glUseProgram(m_shadowMapProgram);`    
`glUniformMatrix4fv(uniformLightMatrixShadow, 1, 0, lightMatrix.Pointer());`    
`glUniformMatrix4fv(uniformProjectionShadow, 1, 0, projectionMatrix.Pointer());`    
`glUniformMatrix4fv(uniformModelviewShadow, 1, 0, modelviewMatrix.Pointer());`    
`glViewport(0, 0, screen.x, screen.y);`    
`glActiveTexture(GL_TEXTURE0);`    
`glBindTexture(GL_TEXTURE_2D, m_textureShadow);`    
`glUniform1i(uniformShadowMapTextureShadow, 0);`    
GLsizei stride = sizeof(Vertex);    
`const vector<ObjModel>& objects = m_Scene.getModels();`    
`const GLvoid* bodyOffset = 0;`    
for (int i = 0; i < objects.size(); ++i) {    
    modelviewMatrix = scale * rotation * translation * LookAt(vec3(0, 8, 7), vec3(0.0, 0.0, 0.0, 0.0), vec3(0, 7, -8));    
    `glUniformMatrix4fv(uniformTransformShadow, 1, 0, objects[i].m_Transform.Pointer());`    
}
The most interesting parts of these rendering shadows are the shaders. Here’s the technique.

Vertex shader (draws shadows):

```glsl
attribute vec3 Position;
attribute vec3 Normal;
attribute vec4 SourceColor;
attribute vec2 TexCoord;

varying vec4 fColor;
varying vec3 fNormal;
varying vec2 fTexCoord;
varying vec4 fShadowMapCoord;

uniform mat4 Projection;
uniform mat4 Modelview;
uniform mat4 lightMatrix;
uniform mat4 Transform;

void main(void)
{
    fColor = SourceColor;
    gl_Position = Projection * Modelview * Transform * vec4(Position,
```
The vertex shader, in parallel with its usual work, produces a translation of vertices in the plane of the light source. In this example, the transition to the plane of the light given by the matrix lightMatrix and the result are passed to the pixel shader through fShadowMapCoord.

Pixel shader (draws shadows):

```glsl
uniform highp sampler2D shadowMapTex;

varying lowp vec4 fColor;
varying lowp vec3 fNormal;
varying highp vec2 fTexCoord;
varying highp vec4 fShadowMapCoord;

highp vec3 Light = vec3(0.0, 4.0, 7.0);
highp vec4 Color = vec4(0.2, 0.4, 0.5, 1.0);

void main(void)
{
    const lowp float fAmbient = 0.4;
    Light = normalize(Light);
    highp float depth = (fShadowMapCoord.z / fShadowMapCoord.w);
    highp float depth_light = texture2DProj(shadowMapTex, fShadowMapCoord).r;
    highp float visibility = depth <= depth_light ? 1.0 : 0.2;
    gl_FragColor = fColor * max(0.0, dot(fNormal, Light)) * visibility;
}
```

The pixel shader calculates each pixel value depth based on the relative light source and compares it with the value corresponding to it in the depth map. If the value does not exceed the depth of the depth maps, it is visible from the source position; otherwise, it is in the shade. In this example, we change the visual color intensity using the coefficient visibility, but in general, it is a more difficult technique.

About the Authors
Stanislav works in the Software & Service Group at Intel Corporation. He has 10+ years of experience in software development. His main interest is optimization of performance, power consumption, and parallel programming. In his current role as an Application Engineer providing technical support for Intel® processor-based devices, Stanislav works closely with software developers and SoC architects to help them achieve the best possible performance on Intel platforms. Stanislav holds a Master's degree in Mathematical Economics from the National Research University Higher School of Economics.

Iliya, co-author of this blog, is also a Senior Software Engineer in the Software & Service Group at Intel. He is a developer on the Intel® VTune™ Amplifier team. He received a Master’s degree from the Nizhniy Novgorod State Technical University.
Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel, the Intel logo, Atom, and VTune are trademarks of Intel Corporation in the U.S. and/or other countries. Copyright © 2013 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.