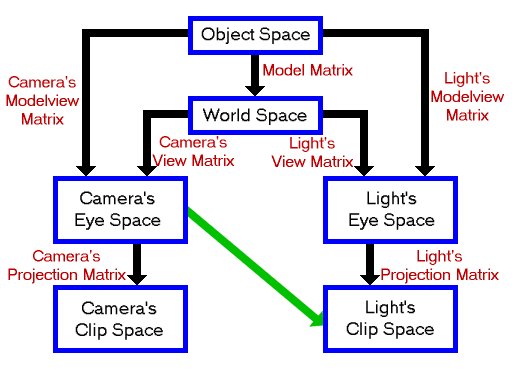
Shadow Mapping: image - based shadowing technique

uses texturing and depth buffering

First render from the light's point of view

light's view is rendered, storing the depth of every surface it sees (the shadow map)

The regular scene is rendered comparing the depth of every point drawn (as if it were being seen by the light, rather than the eye) to this depth map. (hence the following image)



**Paul's Project Shadow Mapping Tutorial**

http://www.paulsprojects.net/tutorials/smt/smt.html

**Shadow Mapping by Cass Everitt**

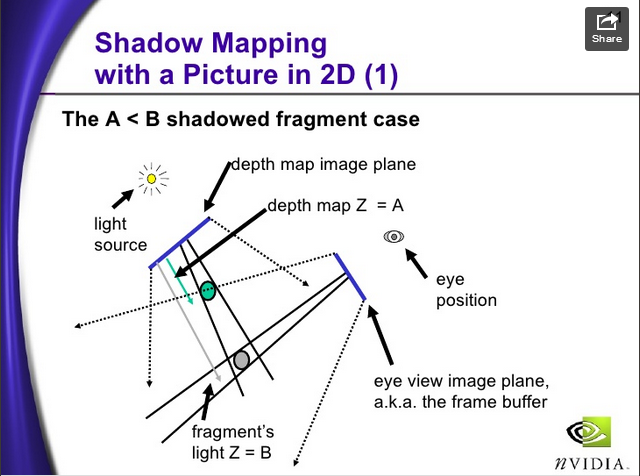
http://www.cs.berkeley.edu/~ravir/6160/papers/shadow\_mapping.pdf

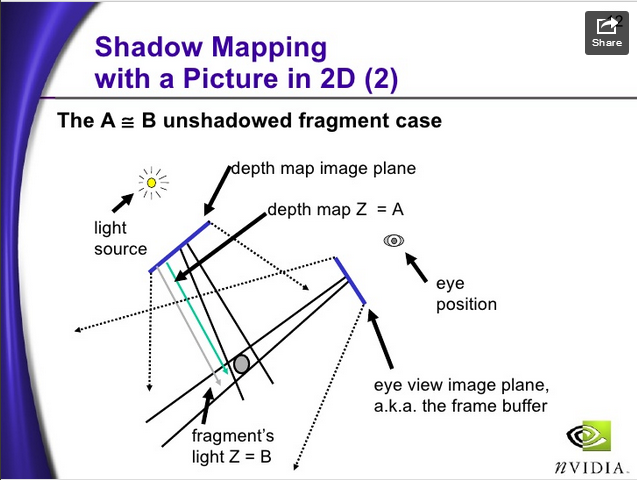
**More explanations in here (very good explanations)**

http://takinginitiative.wordpress.com/2011/05/15/directx10-tutorial-10-shadow-mapping/

**Shadow Mapping Explained From Nvidia**

https://developer.nvidia.com/sites/default/files/akamai/gamedev/docs/ShadowMaps\_CEDEC\_E.pdf?download=1





**C Specification**

|  |  |  |
| --- | --- | --- |
| void **glGetTexGendv**( | GLenum | *coord*, |
|  | GLenum | *pname*, |
|  | GLdouble \* | *params*); |
| void **glGetTexGenfv**( | GLenum | *coord*, |
|  | GLenum | *pname*, |
|  | GLfloat \* | *params*); |

|  |  |  |
| --- | --- | --- |
| void **glGetTexGeniv**( | GLenum | *coord*, |
|  | GLenum | *pname*, |
|  | GLint \* | *params*); |

**Parameters**

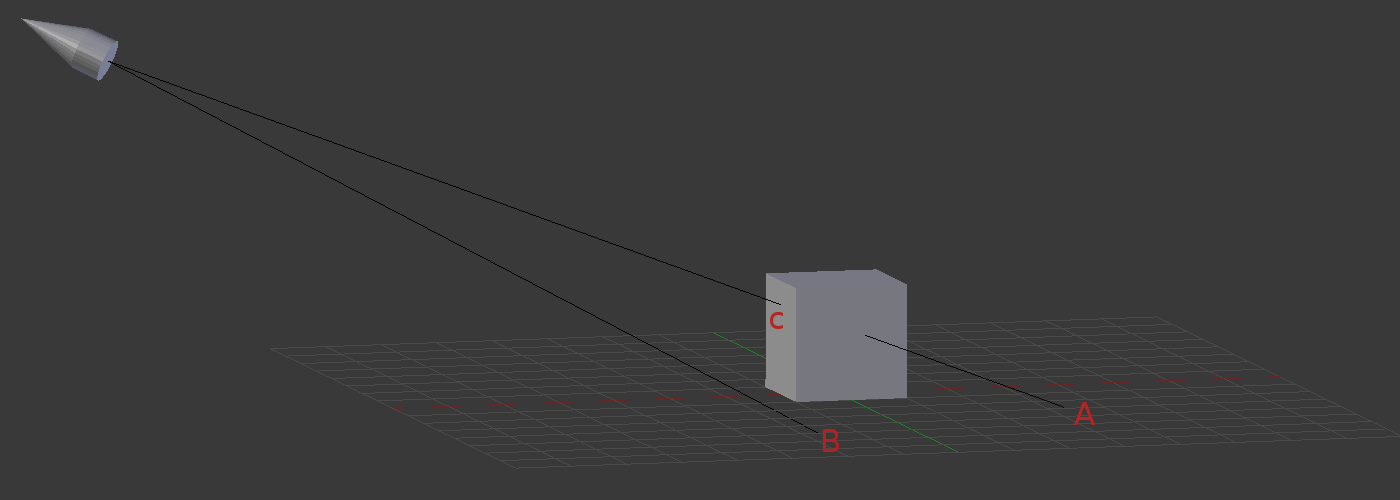
*coord*

Specifies a texture coordinate. Must be GL\_S, GL\_T, GL\_R, or GL\_Q.

## Description

glGetTexGen returns in *params* selected parameters of a texture coordinate generation function that was specified using [glTexGen](https://www.opengl.org/sdk/docs/man2/xhtml/glTexGen.xml). *coord* names one of the (s, t, r, q) texture coordinates, using the symbolic constant GL\_S, GL\_T, GL\_R, or GL\_Q.

BackGround



how depth buffer works

when point B is rendered, its depth value goes into the depth buffer.

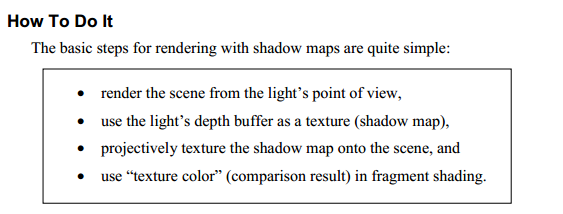
For point C and A, after the perspective projection the rasterizer finds out both points need to go to the same pixel on the screen. This is the depth test and point C wins it, so his depth value gets written to the depth buffer

We're comparing the depth from the clip space coordinate and the stored depth. So all that remains is how do we fetch the stored depth value?

we first convert the position/vertex into light's clip space. Then once we have the clip space coordinate, we need to find a way to get the depth value from the shadow map.

1. we fetch it in the fragment shader. The rasterizer maps the X and Y of the NDC coordinate to screen space and uses them to store the depth. So we can get our current pixel's depth.

then for stored depth, we transform the X and Y of the current NDC coordinate into texture coordinate. But also remember, we need to map [-1, 1] to [0, 1] between NDC coordinate and texture coordinate.



**Steps (referencing the CplusplusGuy's tutorial)**

**1. Render the Scene from the light's point of view into a texture**

- From light's point of view, render the scene to the depth texture with GL\_DEPTH\_TEST enabled.

- we get the closest depth values in the depth buffer, and we get our shadow map.

- by Enabling GL\_DEPTH\_TEST, it will leave the pixels closest to the screen(your current perspective) in the Depth buffer.

**Code walkthrough**

1. rendering from light's point of view is just like rendering from a camera's perspective, you set to (VIEW\_MATRIX), and rotate/translate to light's position (Adobe Hebrew)

m\_pipeline.matrixMode(VIEW\_MATRIX);

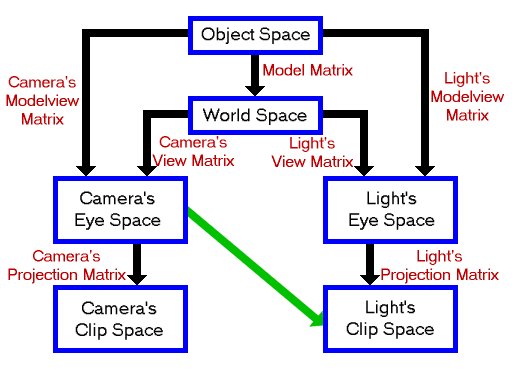
m\_pipeline.loadIdentity();

m\_pipeline.rotateX(lightDirection.x);

m\_pipeline.rotateY(lightDirection.y);

m\_pipeline.translate(lightPosition.x,lightPosition.y, lightPosition.z);

we also want to save the modelViewMatrix of the light's perspective now, because we will transform all vertices according to the light's modelViewProjection Matrix in the 2nd pass.



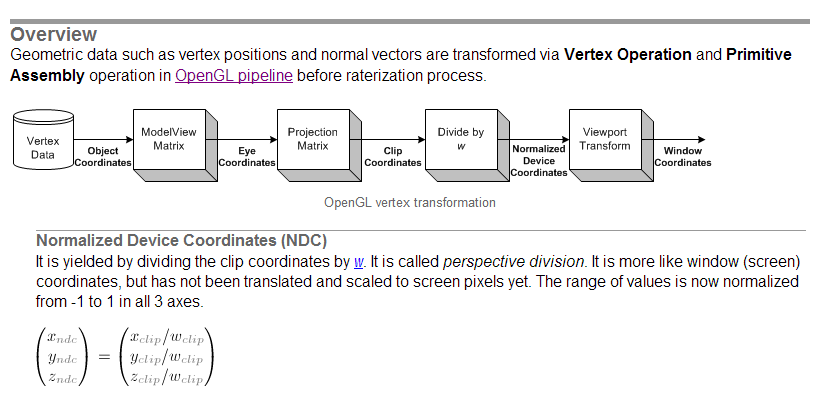
Light\_ModelMatrix = m\_pipeline.getModelMatrix();

Light\_ViewMatrix = m\_pipeline.getViewMatrix();

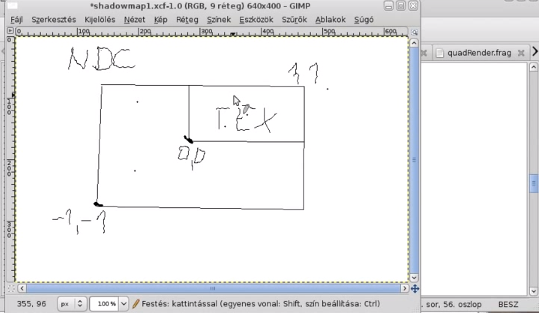
Light\_ProjectionMatrix = m\_pipeline.getProjectionMatrix();

Light\_ModelViewProjectionMatrix = Light\_ProjectionMatrix \* Light\_ViewMatrix \* Light\_ModelMatrix;

However, when you multiply vertex with the modelviewProjection Matrix, it will place the vertices in NDC coordinates (Normalized Device coordinates)



NDC coordinates as explained are in range (-1, 1), however our shadow Texture is in range (0,1), See the comparison below



Which is why we need to somehow map the vertex's position in the NDC coordinate into the texture coordinate using a bias Matrix. Essentially we are moving everything in range (-1, 1) to something in range (0,1)

static MATRIX4X4 Light\_biasMatrix  
(0.5f, 0.0f, 0.0f, 0.0f,  
0.0f, 0.5f, 0.0f, 0.0f,  
0.0f, 0.0f, 0.5f, 0.0f,  
0.5f, 0.5f, 0.5f, 1.0f);

so our Conversion Matrix(shadowMatrix) here is, what it does it transforms vertices in object space in to texture coordinates. This will be used later on in the 2nd rendering

shadowMatrix = Light\_BiasMatrix \* Light\_ModelViewProjectionMatrix;

2. since we are rendering into our shadow map, we will first enable the Frame buffer and pass our "shadow\_depthTexture" (Our texture object) in. Also we have to enable (GL\_DEPTH\_TEST) and clear the depth buffer

glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, GL\_TEXTURE\_2D, shadow\_depthTexture, 0);

glEnable(GL\_DEPTH\_TEST);

glClear(GL\_DEPTH\_BUFFER\_BIT);

then we just render it with our "shadow\_FirstRender" shader. Besure to remember the turn off the framebuffer after the rendering.

shadow\_FirstRender->useShader();

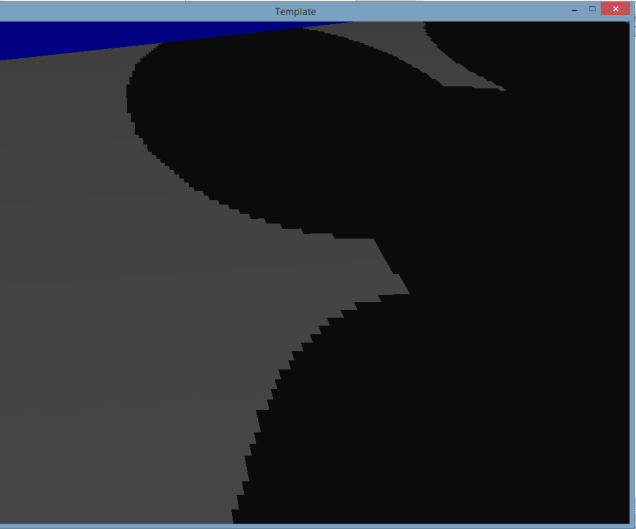
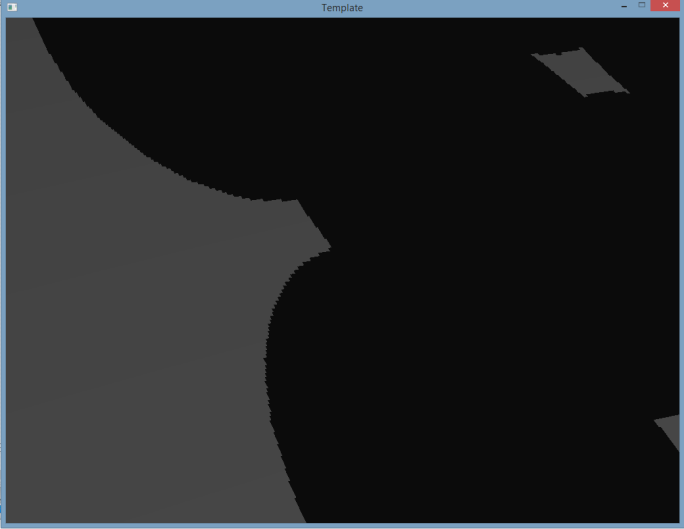
m\_pipeline.updateMatrices(shadow\_FirstRender->getProgramId());

ground->draw(shadow\_FirstRender->getProgramId());

shadow\_FirstRender->delShader();

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

initially the size of your shadowmap can be set same as (SCREEN\_WIDTH, SCREEN\_HEIGHT), but if you want to increase the resolution of the shadow(less zigzag as shown in the pictures), you can set the texture size to be (SCREEN\_WIDTH x 2, SCREEN\_HEIGHT x 2)

SCREEN\_WIDTH, SCREEN\_HEIGHT  SCREEN\_WIDTH x 2, SCREEN\_HEIGHT x 2

in this case, we want to render to the entire texture, not just the size limited by our screen, which is why we need to set glViewport(0,0,shadowMapWidth, shadowMapheight); so this will render to our " shadow\_depthTexture". Of course, we also need to set it back to glViewport(0,0,SCREEN\_WIDTH, SCREEN\_HEIGHT); so it correctly renders it into the screen after we're done

Vertex Shader

Shader wise it's pretty straight forward

#version 330

in vec3 vertex;

//attribute vec3 vertex;

uniform mat4 modelViewProjectionMatrix;

void main()

{

gl\_Position=modelViewProjectionMatrix\*vec4(vertex,1.0);

}

Note: vertex is passed in from "m\_pipeline.updateMatrices(shadow\_FirstRender->getProgramId());"

Fragment Shader

#version 330

out vec4 FragColor;

void main()

{

FragColor=vec4(1.0,0.0,0.0,1.0);

}

The Color doesn't really matter since we only care about the depth value

after we got our textures, we move the camera then move on to the 2nd rendering

We can also render the texture to the screen to see if our depth map is correct

glDisable(GL\_DEPTH\_TEST);

//render texture to screen

m\_pipeline.loadIdentity();

m\_pipeline.ortho(-1,1,-1,1,-1,1);

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

glClear(GL\_COLOR\_BUFFER\_BIT);

quadRenderShader->useShader();

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_2D,shadow\_depthTexture);

glUniform1i(glGetUniformLocation(quadRenderShader->getProgramId(),"texture"),0);

glUniform2f(glGetUniformLocation(quadRenderShader->getProgramId(),"pixelSize"),1.0/SCREEN\_WIDTH, 1.0/SCREEN\_HEIGHT);

m\_pipeline.updateMatrices(quadRenderShader->getProgramId());

quad->draw(shadow\_FirstRender->getProgramId());

quadRenderShader->delShader();

glEnable(GL\_DEPTH\_TEST);

We set it to ortho mode, so we the texture will occupy the screen completely (which will give us a complete view of the texture). We use the quadRenderShader (since we're rendering a quad/rectangular texture map onto the screen)

**2. Render the Scene from the Camera's point of view**

- we compare the depth value of the vertex (after being transformed using the conversion matrix) and that specific coordinate from light's perspective

- comparison mostly done in the shader

we first get the conversion done in the vertex shader

out vec4 lightVertexPosition; // vertex seen in light's clip space

void main()

{

gl\_Position = modelViewProjectionMatrix\*vec4(vertex,1.0);

lightVertexPosition = lightModelViewProjectionMatrix \* vec4(vertex,1.0);

...

... (the lightVertexPosition, is the converted vertex)

then in the fragment shader, we do the comparison

void main()

{

float shadowValue = shadow2DProj(shadowMap, lightVertexPosition).r;

// r==1.0 not in shadow, r==0.0 in shadow

the shodow2DProj returns 1.0 if not in shadow, 0 if the vertices is in shadow.

Percentage Closer Filtering

OpenGL multiplayer camera

universal coordinate system

explanation of modelViewMatrix

explanation of ProjectionMatrix