Shadow Mapping

Oh, divide by W, you so crazy

Ben Kenwright

Abstract

Shadow maps are the current technique for generating high quality real-time dynamic shadows. This article gives a 'practical' introduction to shadow mapping (or projection mapping) with numerous simple examples and source listings. We emphasis some of the typical limitations and common pitfalls when implementing shadow mapping for the first time and how the reader can overcome these problems using uncomplicated debugging techniques. A scene without shadowing is life-less and flat - objects seem decoupled. While different graphical techniques add a unique effect to the scene, shadows are crucial and when not present create a strange and mood-less aura.

1 Introduction

1.1 Simple but Beautiful Shadows

Dynamic shadows can make lifeless, flat, uninteresting scenes more realistic and attractive. Furthermore, dynamic shadows enable a scene to possess the ability to display the passing of time (e.g., night and day).

A basic, straightforward, uncomplicated implementation of shadow mapping can be performed in just a few dozen lines of code. However, new students to graphics are often presented more complex implementations with little understanding to the true underlying meaning of what is accurately happening and what to expect when things go adrift or misbehave.

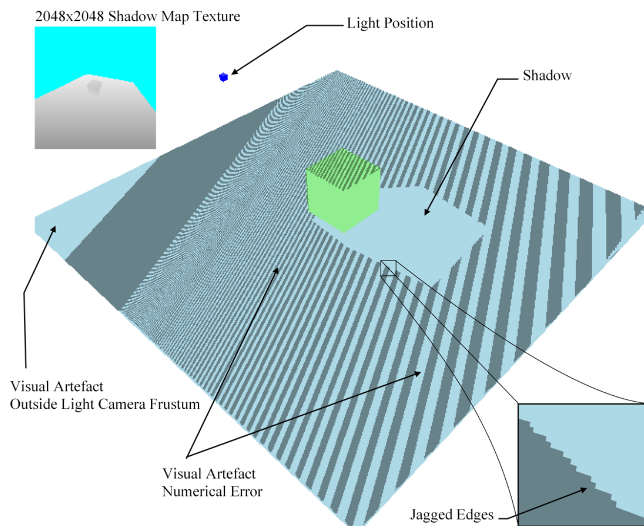


Figure 1: **Artifacts** - Shadow maps are an elegant technique that exploits the depth buffer and the graphical processing hardware to achieve detailed shadows for real-time scenes, such as, in games. However, everything isn't all peaches and cream - since the technique possesses a number of limitations and inherent artifacts which must be handled in a working implementation.

The aim of this article is to give a practical, step-by-step introduction to shadow mapping that will enable the reader to have a solid unquestionable understanding of exactly what shadow maps are. Most importantly how the theoretical and practical aspects differ and how limiting factors such as numerical inaccuracies can be overcome with engineering solutions. Furthermore, this paper gives tips and tricks on how the reader can enhance shadow mapping using out of the box thinking. This paper focuses on the real-time practical aspects to attain visually pleasing even though the approach might be less physically accurate. Towards the end of this article, we introduce some novel methods to expand the basic functioning of shadow maps to produce improved visual effects.

Motivation The goal of this article is to provide a clear, visually interesting, practical perspective on shadow mapping to give the reader a solid unquestionable understanding of shadow mapping. We

aim to present a practical no-frills implementation with a step-by-step account of what is happening, why it happens, and the problems associated with each step. We then go onto introducing numerous enhancements and improvements that can be used to build a usable shadow map system that can be integrated into any system without complications.

2 Related Work

The three fundamental main methods for generating shadows are:

✓ Planar Projection Shadows
✓ Shadow Volumes
✓ Shadow Mapping

Shadows are an important visual effect for both offline and real-time graphical systems. Hence, shadow maps is an active area of research; whereby numerous techniques are constantly being investigated to take advantage of improved hardware advances or novel algorithm enhancements.

We give a short list of some of the novel and interesting papers on shadow mapping that were been presented over the past few decades. Before the evolution of graphical hardware to make shadow maps a viable real-time options you can read about other alternative methods. Such as the initial ground breaking work by Crow on shadow mapping [CROW77] or Weilers work on generating geometric shadowing based on clipping [WEIL77].

Similarly, around the same time, shadow maps were presented by Williams [WILL78] in a paper titles, Casting Curved Shadows on Curved Surfaces.

Multiple shadow map of varying resolution to improve quality and performance [TAQJ01][FEFB01]. [STAM02] presents a paper on perspective shadow maps. Deep shadow maps for hair [LOKO00]. Variance shadow maps [DONN06]. Light space perspective shadow maps [WISC04]. Sample distributed shadow maps [LASA11]. Alternatively, books and the web offer a plethora of resources and tutorials on shadow mapping. While some focus on a specific API or graphical libraries (e.g., OpenGL, DirectX) they still present an interesting read and help diversity a persons understanding while also presenting an alternative practical point of view that you might not have thought of. Gives a broad introduction to shadows with source code samples [LILA12]. [RAND03] book presents an enjoyable read and provides a well-written introduction to shadow mapping. [FANZ07] gives clear diagrams and a good introduction to more advanced topics. [BOBB11] gives a programmers practical perspective on implementing shadow mapping with Directx10. [CLOS01] discusses shadow maps in virtual environments and demonstrates a simple application for varying parameter values. [SANG11] this is an enjoyable tutorial on shadow mapping with GLSL (OpenGL Shading Language) with example source code and systematic examples.

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}_{\text{Homogeneous}} \Rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}_{\text{Cartesian}}$$

Figure 2: **Homogeneous Coordinates** - Homogeneous coordinates or projective coordinates (i.e., w-component). Essentially, they are a neat extension of standard three dimensional vectors and allow us to simplify various transforms.

Trade-offs As with most things, there is no single amazing solution. You have to choose what is more important - producing extremely high quality accurate shadows at the expense of more memory and longer render times.

3 Theory

Shadow mapping is a two-pass technique. Firstly, the scene is rendered from the location of the light source into a depth texture. This pass generates the shadow map texture. Secondly and finally, the scene is rendered from the location of the camera as usual but we use the shadow map texture to identify which parts of the scene are in shadow from the light source. Therefore, for each pixel

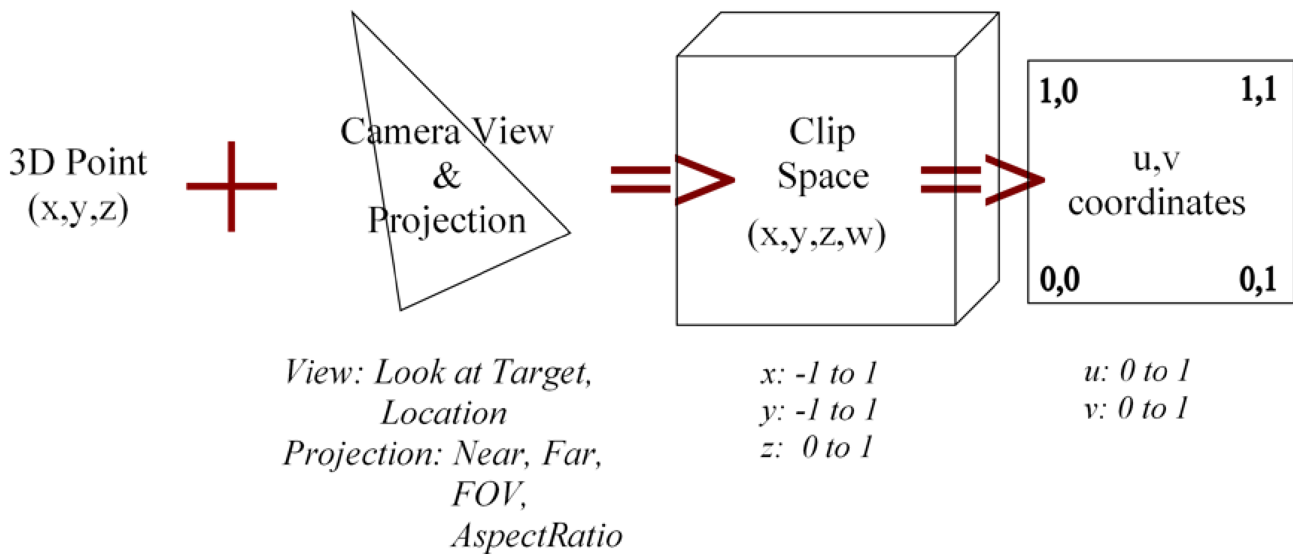


Figure 3: **Pipeline** - 3D to 2D conversion process.

rendered we compare the depth sample with the shadow map depth. If the shadow map depth is greater than the cameras depth then the pixel is closer to the light source. Alternatively, if the pixel is not in shadow we would expect both depth values to be almost identical.

When people are shown Figure 4 it does not really explain HOW. We understand that the first phase will generate the shadow map texture with the depth information. Furthermore, the second phase will use the shadow map to resolve if the pixel is being shadowed.

In the pixel shader we determine the colour of each pixel. We can extrapolate the 3D world coordinates for that pixel. Then we determine if the 3D location of that point is closer than what the shadow map calculated. If another point was in front of it then the shadow map would have only stored the closer z-distance due to the z-buffer.

The reader should understand that the view matrix and a projection matrix are combined to transform a 3D world position into 2D screen coordinates relative to the cameras view matrices location.

We are given two view camera matrices. The first is the view matrix for the light source and the second is the view matrix for the camera. When we render the scene for the shadow map or the camera we are given the 3D world location for every point that will be converted and squashed onto a flat surface.

4 3D World Position to 2D + Depth

Combining the world, view, and projection matrix together gives the necessary information for converting a 3D world position into the 2D screen position plus the z-depth distance from the screen surface. When we render the scene we pass the 3D information to the vertex and pixel shader. The vertex shader simply converts the 3D vertex position to the screen coordinates using the world, view and projection. Both the shadow map and the main scene camera render phase will use the same light view and light projection matrix to gather depth information.

1. Create the light world-view-projection matrix
2. Multiply the 3D world position by the light world-view-projection matrix to produce the 2D plus depth information
3. The full screen is a texture so instead of placing a colour at the 2D location you place the depth information
4. The automatic z-buffer will ensure only the nearest 3D world positions are remembered

Remember that two 3D points can occupy the same pixel position when multiplied by the world-view-projection matrix. However, the z-buffer ensures that only the closest depth information is stored in our shadow map texture. Alternatively, if we know the location of a 3D point we can calculate its depth information.

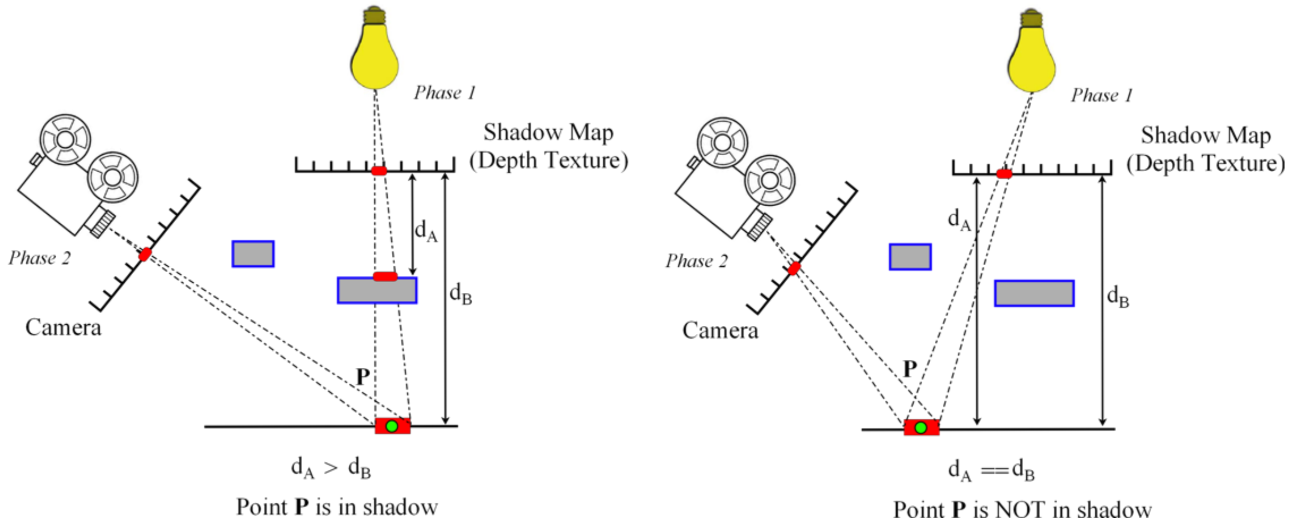


Figure 4: **Two Passes** - Illustrate the two phase process of generating a distance from the shadow map Two possible outcomes either the z-distance will be the same as the camera or it will be less due to an object being closer to the light source view.

Step-by-Step The five steps to implementing a simple shadow mapping technique:

Step 1 You pass your model and light matrix information e.g.:

```
1 Matrix mvp = World * View * Projection;
```

Step 2 You render your vertices. Each vertex is passed to the vertex shader in the form of an x,y,z coordinate. You multiply this by your mvp matrix.

```
1 float4 pos = vertex * mvp;
```

The resulting multiplication has given you a Homogeneous coordinate. To convert this to a Cartesian coordinate you simply divide it by the w component. e.g.:

```
1 pos /= pos.w;
```

Step 3 However, we pass the four float homogeneous coordinate to the pixel shader to take advantage of the interpolation. Hence, at the pixel shader we are given the 3D world position (in homogeneous coordinates) for every point.

Step 4 This is where the two-render phases differ. Since the scene rendered from the light will have object behind other objects culled due to the z-buffer. However, the same objects rendered from the main camera might not be behind another object.

Step 5 You perform the same operations for the shadow map shader and the main scene shader. However, the main scene will compare the z depth from the shadow map and if it is different then the vertex point is in shadow. We know after we have converted the 3D point into Cartesian space that it will be within the clip space region. It is simply a matter of scaling and converting to texture coordinates to find the shadow maps depth value for the comparison.

4.1 Depth Comparison

Due to numerical errors and approximations the depth from the shadow map texture and the depth calculated at that instant using the main camera can have small differences. Depending upon the

size of the texture the distance from the camera and the number of bytes being allocated to store the depth information it can range from anywhere between 0.0001 to 0.1.

4.2 Linear Depth Buffer

The depth buffer by default is non-linear. The precision of the z-buffer can be limited. Majority of the precision is focused towards close to the eye and very little precision is towards the far distance. The principle is this, we have our vertex coordinate, e.g., v(x,y,z,1). Then we take our lights projection matrix. When this is multiplied out, we get our transformed coordinate, e.g., v(x,y,z,w).

```
1 v' = v * ProjectionMatrix
```

At this point, v is still linear. The conversion from homogeneous coordinates by dividing by the w component results in the non-linear problem. If we neglecting the near plane and remember that the z value should be between 0.0 to 1.0. We simply replace the w division with the z far distance. e.g.:

```
1 non-linear :  pos /= pos.w
2 linear     :  pos /= zFar
```

This simple approximation can be replaced with a more accurate one to incorporate the z-near (see Brabec et al. for details [BRAS02]). In addition, the excellent article by Dunlop [DUNL00] gives a very clear explanation of how the z-buffer is created and how it can be linearized in the shader. Again, the technique of linearizing the depth buffer to gain more uniform shadow maps is explained by Brabec et al. [BRAS02].

4.3 Orthographic or Perspective Projection Matrix

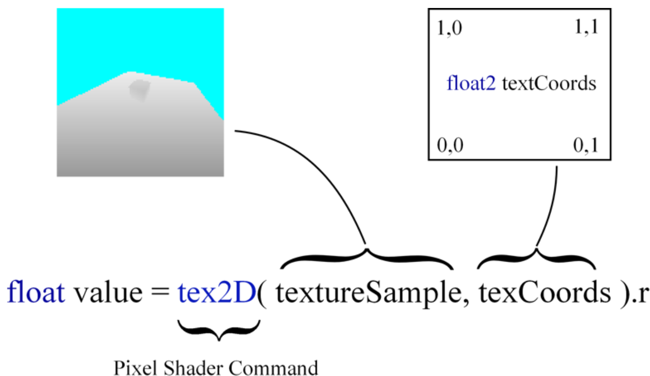


Figure 5: **Screen and Texture Coordinates** - Understanding the texture mapping command argument for extracting correct depth information from the shadow map texture.

The shader implementation details remain the same for both an orthographic matrix or a perspective matrix. Since both the generation of the shadow map and the extrapolation of the shadow pixels are done with the same projection matrix. Perspective projection simulates a point light, while an orthographic matrix mimics a directional light.

4.4 Visual Artefacts

The initial implementation with no-bells-or-whistles exhibits various undesirable visual artefacts. Understanding where these artefacts come from and how we can overcome them is crucial if you want shadow maps to be a viable option for generating quality shadows.

- Incorrect Texture Mode (i.e. Clamping)
- Jagged Edges
- Moir like line artefacts
- Comparison Tolerance
- Toggle Culling
- Light Views Frustum Edges

5 Discussion

Disadvantages

- Each point light needs its own shadow map

- Requires a depth buffer and texture projection
- Must render the scene twice from two different locations (the rendering of the shadow map can use a highly reduce (low-poly) resolution scene)
- Limited resolution (problem of aliasing and jagged looking edges since a single can pixel can represent a large region of space)
- Needs fragment hardware programming capabilities to achieve real-time frame-rates
- Is problematic when a light source is in the middle of the scene with objects all around it. Whereby, shadows will be cast in a full 360 degrees.

Advantages

- No need to update the shadow map if the light or objects dont move
- To take advantage of shadow mapping the scene can be rendered as usual with textgen command to project the screen space coordinate z back to the light space
- Implementation is logical and simple to understand
- Render complex shadow silhouettes
- Automatically disregard and exclude objects outside the clip region

6 Limitations

The fundamental implementation suffers from numerous flaws. For example, a 2048x2048 texture is usually employed to ensure that sufficiently detailed shadows are presented (consuming approx 16mb of memory for a 32bit texture). Furthermore, to gain sufficiently smooth and realistic looking shadows for complex scenes numerous advanced techniques must be employed (e.g., multi-resolution shadow mapping) that can be computationally expensive.

7 Source Code

To simplify the source code we moved the settings for culling type, opaqueness, depth state and so on to the shader (fx) file. Alternatively, the settings can be excluded from the shader and manually in on the renderer in the source code dynamically.

8 Filtering, Smoothing and Soft-Shadows

Simply looping over the surrounding pixels in the shadow map texture and averaging them produces more softer shadow edges.

```
1  #if 1 // Non-blurred default method
2  float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
3  #else // Blurring method
4  // Average filtering on a 4 x 4 texel neighbourhood
5  float depthStoredInShadowMap = 0;
6  for (float y = -1.5; y <= 1.5; y += 1.0)
7  {
8  for (float x = -1.5; x <= 1.5; x += 1.0)
9  {
10 float2 shadowMapSize = 1.0f / float2(2048,2048);
11 float2 texOffset =float2(x*shadowMapSize.x,y*shadowMapSize.y);
12
13 depthStoredInShadowMap += tex2D(ShadowMapSampler, ProjectedTexCoords.xy + texOffset );
14 }
15 }
16 depthStoredInShadowMap = depthStoredInShadowMap / 16.0;
17 #endif
```

Listing 1: Blurring shadow edges.

9 Transparency

The uncomplicated shadow map implementation does not reflect any object transparency by producing shadows of different density. For example, if you have a dirty window in your scene with varying patches of dirt it would be nice if you could emphasis this in your shadow.

One approach that emulates the effect for simple scenes in a hacky sort of way is to embedding the material type of the closest object in the shadow map texture. Therefore, when you come to extracting your shadow information from the shadow map you can extract material information for the object that is occluding the current position. For example, we could visualize this by altering the lightness of the shadow.

10 Debugging

To ensure your shadow map phase is working correctly you should render it in the top left of your screen. This will confirm that your shadow maps view and projection matrix are set up correctly and it will confirm that it is pointing in the correct direction and what parts of the scene will be lit by your lights location. However, if this is blank or is viewing an unseen part of the scene then it needs correcting.

It is vital that you understand each step of the process. For example, the fundamental conversion of Homogeneous to Cartesian coordinates. Do not just accept that it is necessary it should be clear that it is unavoidable due to the way to transform coordinates using the projection matrix.

11 Image Resolution

Experimentation is the road to understanding. Modifying the texture size as expected introduces blocky edges due to loss of information (see Figure 11).

11.1 Typical Problems

Shadow maps are not perfect. However, they do provide a straightforward, uncomplicated, and visually pleasing result. The majority of typical problems comes from a poor understanding of how shadow maps and the render pipeline works (i.e., how the view, projection, and homogeneous coordinate system work). Typically, students who implement shadow mapping for the first time are given an ambiguous explanation along the lines of, the shadow map is in the light space coordinates that we extract and compare. Furthermore, an optimised shader is presented to the student that contains numerous undefined tricks and overhead that does not make it clear to the student what is happening.

The problem is further exasperated when a problem occurs or the student attempts to modify or extend the shadow mapping system. Due to their limited understanding, it results in many, many hours of trial and error experimentation. Always implement the simple, uncomplicated, un-optimized version initially so that you completely understand every line without question.

12 Tips and Tricks

Once you completely understand the theory and have a basic implementation working you can move onto integrating additional improvements to enhance the visual result.

12.1 Multiple Shadow Maps

This requires two shadow maps to be rendered. It is highly expensive but generates high detailed shadow maps at the focus point. First, we render a smaller more focused area using a high detailed shadow map. Secondly, the rest of the viewable scene is rendered using a wider view of the entire scene. Finally, when we come to render our scene we add an additional bounds check to determine which shadow map to use. This technique can exploit the dynamic nature of the fallback shadow map.

As our main camera moves in closer to view shows closer up the more detailed shadow map is able to also move in closer without losing shadows in the distance.

Extending the two-shadow map idea even further you can attain even further detailed shadows. Conversely, be careful that you do not add a shadow to a scene that cannot exist due to the location of another light source.

12.2 Filtering, Smoothing and Soft-Shadows

To enable your shadows to be softer and less jagged you can apply a simple averaging kernel filter to each pixel point. Whereby, each point is combined with its neighbours and averaged to produce a blurring effect.

12.3 Clipping and Soft Fall-Off

Shadows outside the view area of your light should be clipped either by checking the w component or checking the xyz clip space (see Figure 1). You can identify the clip area by highlighting the region outside the cameras view area (see Figure 10). For shapes on the edge of the lights viewing frustum the shadows are sliced abruptly and present a visually unpleasing and unnatural result. However, while it is impossible to remove this artefact without enlarging the viewing frustum we can alternatively reduce the ugly visual unpleasantness. This can be achieved by applying a sharp linear fall-off at the edges of the xyz clip space. Whereby, shows that are close to the edge of the clip space fall off in density exponentially so that you achieve a less noticeable and distasteful visual artefact.

12.4 Scale, Offset Stored Z-Depth

You can offset or scale the depth information in the shadow map texture. Square root it will ensure the magnitudes are shrunk over a smaller range. While exponentially multiplying or squaring the depth to a power will ensure that more of the 32-bit float range is taken. Alternatively, you can offset the depth to try to gain better resolution and take advantage of floating point precision (e.g., subtract 1.0).

13 Conclusion

In conclusion, Shadow Maps are a valuable approach for generating complex shadows in reasonable time-frames (i.e., real-time compared to off-line methods, such as, ray-tracing). They do suffer from limitations. However, the technique has multiple engineering work arounds to solve these short-comings, such as, multi-resolution maps and bluring.

References

[BENK13] Ben Kenwright (2012). March, Shadow Maps, What they are, How they work, and How to implement them. Online Technical Article.

[BOBB11] Bobby Anguelov. (2011). DirectX10 Tutorial 10: Shadow Mapping Part 1. Web: Blog. Retrieved from http://takinginitiative.net/2011/05/15/directx10-tutorial-10-shadow-mapping/

[BRAS02] Brabec, S., Annen, T., & Seidel, H.-P. (2002). Practical shadow mapping. Journal of Graphics Tools, 7(4), 9-18. doi:10.1080/10867651.2002.10487567

[CLOS01] Closson, E. (2001). Real-Time Dynamic Shadow Generation. Web. Retrieved from http://www.evanclosson.com/projects/realtimeshadows

[CROW77] Crow, F. (1977). Shadow algorithms for computer graphics. ACM SIGGRAPH Computer Graphics, 242-248.

[DONN06] Donnelly, W. (2006). Variance shadow maps. Proceedings of the 2006 symposium on, 161-166.

[DUNL00] Dunlop, R. (2000). Linearized Depth using Vertex Shaders. Retrieved January 2, 2011, from http://www.mvps.org/directx/articles/linear_z/linearz.htm

[FANZ07] Fan Zhang, Hanqiu Sun, O. N. (2007). Chapter 10. Parallel-Split Shadow Maps on Programmable GPUs. GPU Gems 3. Retrieved from http://http.developer.nvidia.com/ GPUGems3/ gpugems3 _ch10.html

[FEFB01] Fernando, R., Fernandez, S., & Bala, K. (2001). Adaptive shadow maps. Proceedings of the 28th, (August), 387-390.

[LASA11] Lauritzen, A., & Salvi, M. (2011). Sample distribution shadow maps. on Interactive 3D Graphics and Games.

[LILA12] Liland, . (2012). Shadow. Web. Retrieved from http://www.ia.hiof.no/ borres/cgraph /explain/shadow /p-shadow.html

[LOKO00] Lokovic, T. (2000). Deep shadow maps. Proceedings of the 27th annual conference on, 385-392.

[RAND03] Randima Fernando. (2003). Chapter 9. Advanced Topics. The Cg Tutorial: The definitive Guide to Programmable Real-Time Graphics. Retrieved from http://http.developer.nvidia.com/ CgTutorial/cg_tutorial_chapter09.html

[SANG11] Sanglard, F. (2011). ShadowMapping with GLSL. Web: Blog. Retrieved from http://fabiensanglard.net /shadowmapping/index.php

[STAM02] Stamminger, M. (2002). Perspective shadow maps. ACM Transactions on Graphics (TOG).

[TAQJ01] Tadamura, K., Qin, X., & Jiao, G. (2001). Rendering optimal solar shadows with plural sunlight depth buffers. The Visual Computer, 166-173. Ieee. doi:10.1109/CGI.1999.777935

[WEIL77] Weiler, K. (1977). Hidden surface removal using polygon area sorting. ACM SIGGRAPH Computer Graphics, 214-222.

[WILL78] Williams, L. (1978). Casting curved shadows on curved surfaces. ACM SIGGRAPH Computer Graphics, 270-274.

[WISC04] Wimmer, M., & Scherzer, D. (2004). Light space perspective shadow maps. Symposium on Rendering, (2004).

```
1
2  #include <glew.h> // glUseProgram
3  #include <glfw3.h>
4
5  #pragma comment(lib, "glfw3.lib") // glfwCreateWindow, glfwMakeContextCurrent
6  #pragma comment(lib, "opengl32.lib") // glClear, glGetString
7  #pragma comment(lib, "glew32.lib") // glewUseProgram
8
9
10 using namespace std;
11 #include <vector>
12
13 #include "debug.h"
14 #include "vector3.h"
15 #include "cube.h"
16
17
18 #define SCREEN_WIDTH    1280
19 #define SCREEN_HEIGHT   720
20 #define SHADOW_SCALE_FACTOR 4
21 float g_fSpinX = 0;
22 float g_fSpinY = 0;
23
24
25
26 const char*
27 s_vertexShaderShadow =
28 "                          \n\
29 #version 420 core               \n\
30 uniform mat4 MVP;                  \n\
31 layout(location = 0) in vec3 inPosition;    \n\
32 void main()                 \n\
33 {                          \n\
34 gl_Position  = MVP * vec4(inPosition,1);  \n\
35 }                          \n\
36 \n\
37 ";
38
39 const char*
40 s_fragmentShaderShadow =
```

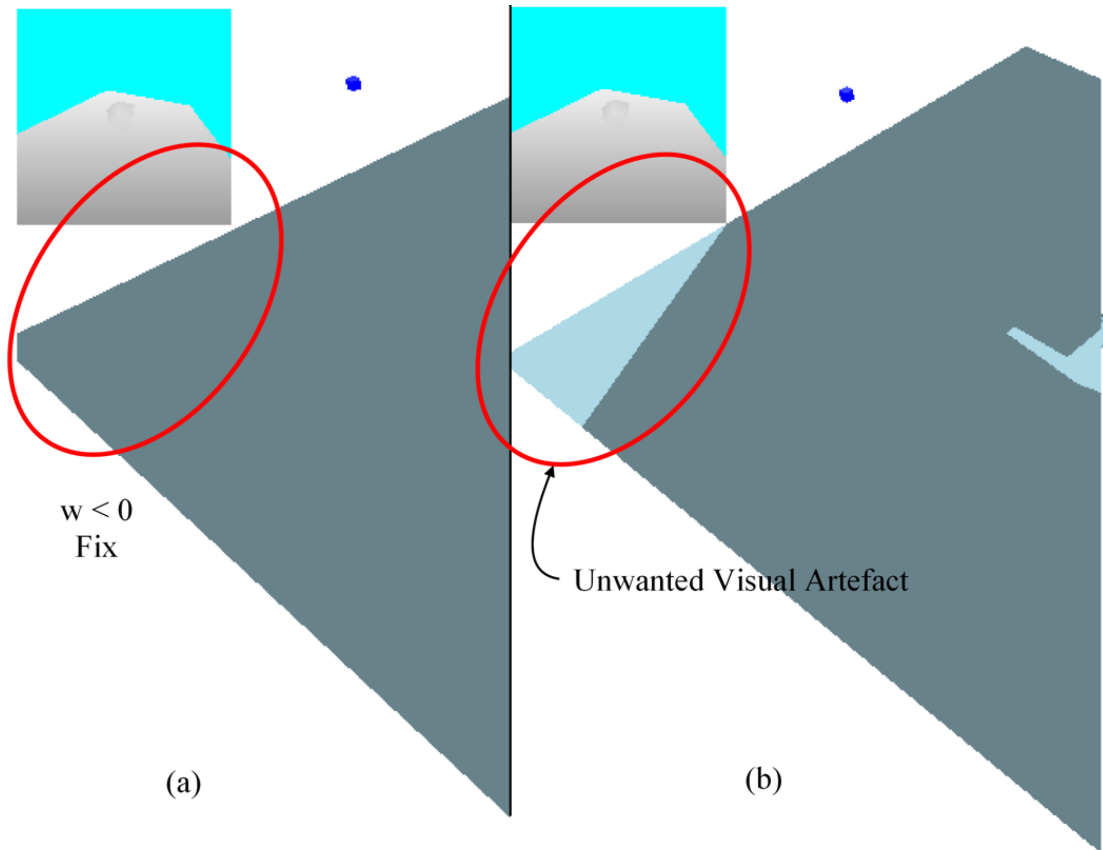


Figure 6: **W Component** - The w component in the main scene can be checked to be negative (e.g. lightPoint.w ¡ 0) to indicate it is outside the camera viewing frustum.

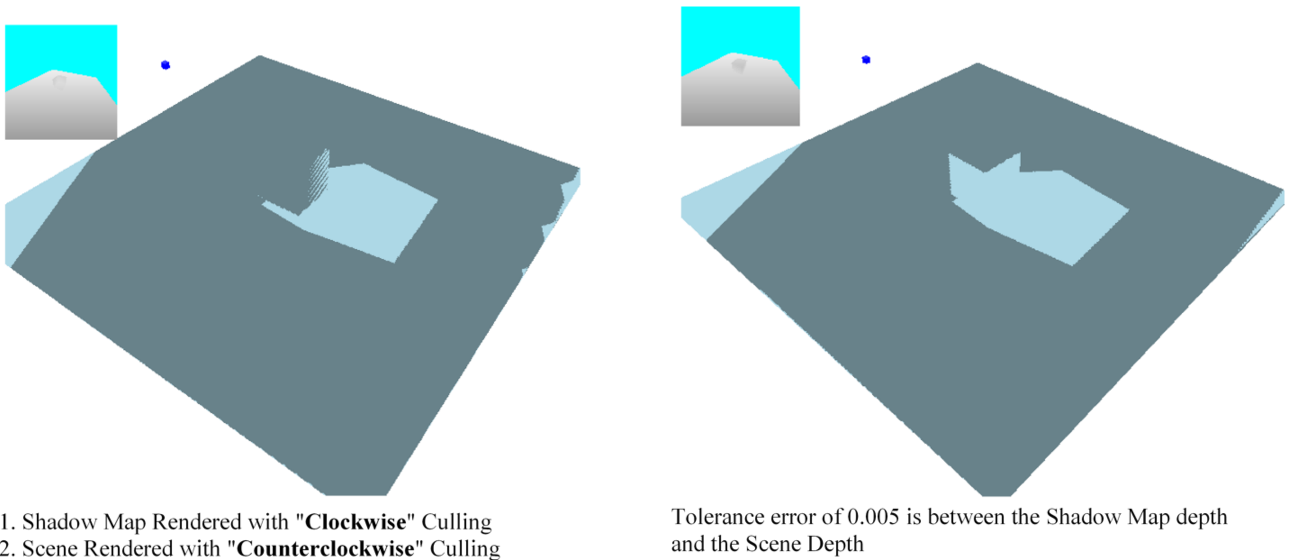
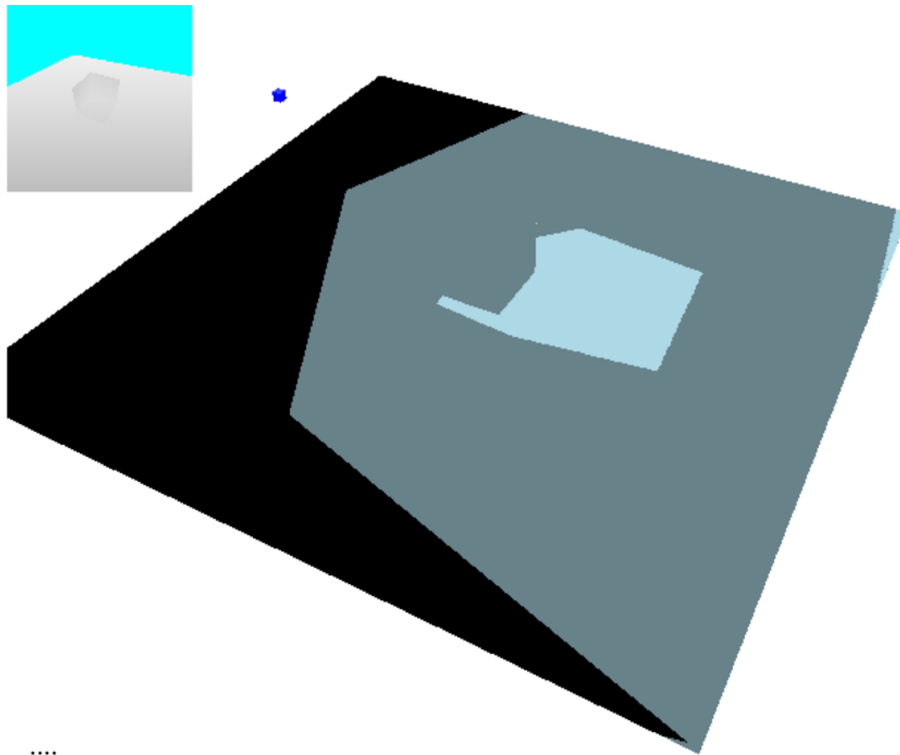


Figure 7: **Clockwise or Counter-clockwise** - Methods to reduce the moir like line artifacts presented due to numerical inaccuracies.



```
....
if ( input.Pos2DAsSeenByLight.x < -1.0f || input.Pos2DAsSeenByLight.x > 1.0f ||
     input.Pos2DAsSeenByLight.y < -1.0f || input.Pos2DAsSeenByLight.y > 1.0f ||
     input.Pos2DAsSeenByLight.z <  0.0f ||  input.Pos2DAsSeenByLight.z > 1.0f )
{
        return float4(0,0,0,0);
}
....
```

Figure 8: **Clamping** - To identify the view frustum of the camera at the light source you can add a simple bounds check.

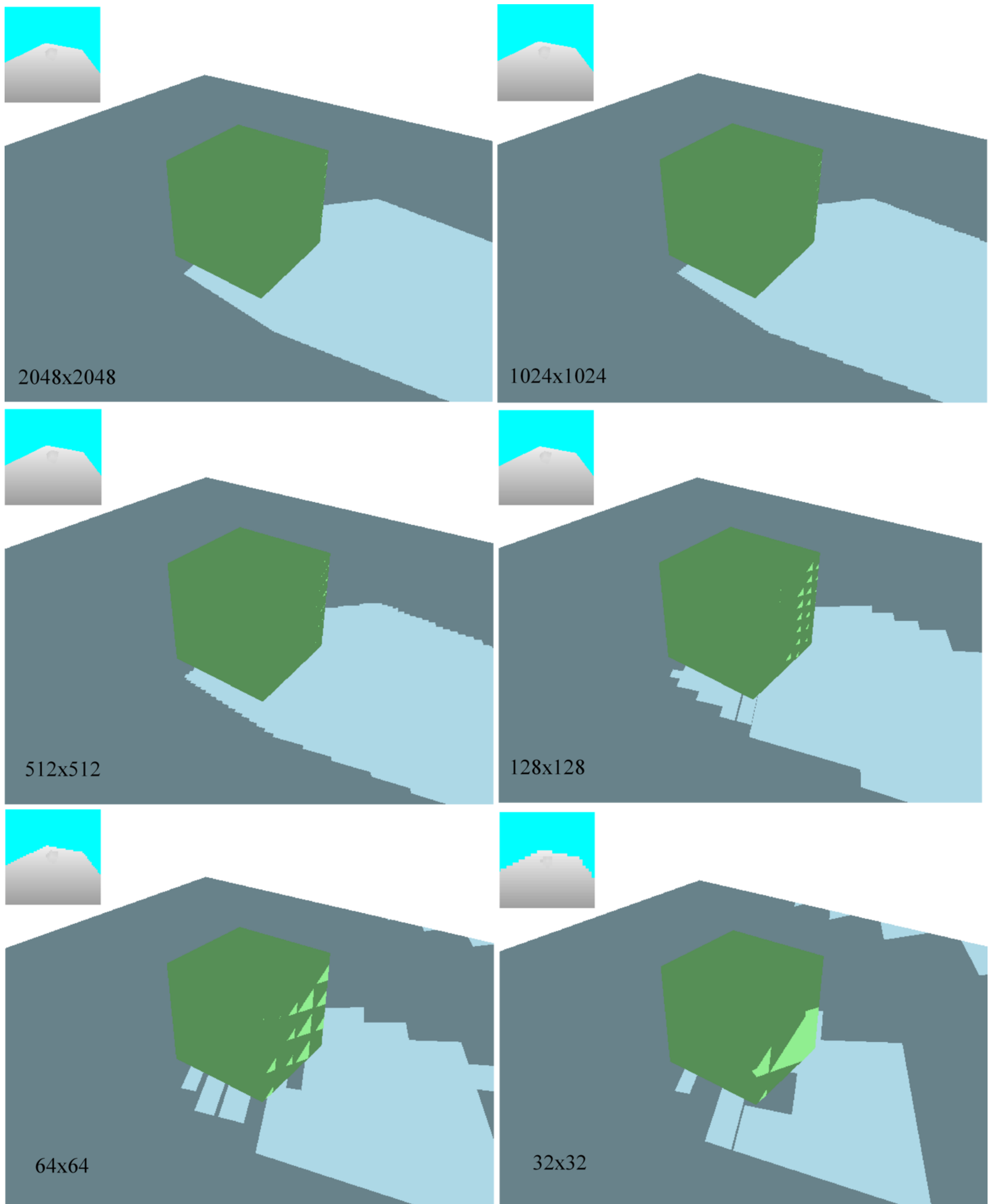


Figure 9: **Resolution** - Reducing the texture map size from 2048x2048 down to 32x32 in steps.

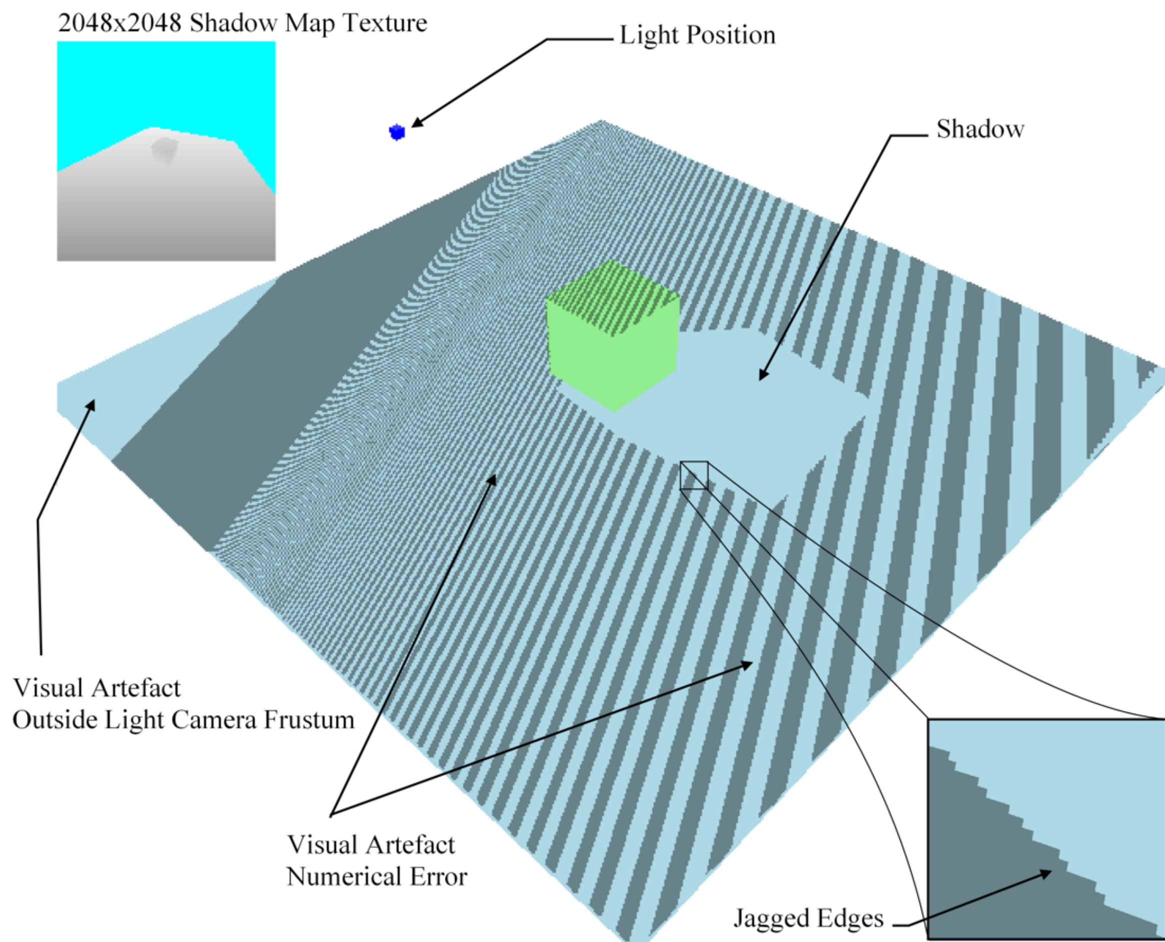


Figure 10: **Moire** - Initial implementation of a shadow-map with typical unwanted visual artifacts.

```
41 "                          \n\
42 #version 420 core                \n\
43 layout(location = 0) out vec4 outColour;     \n\
44 void main()                      \n\
45 {                        \n\
46 outColour = vec4(1); // 0, 0, 1, 1);         \n\
47 }                        \n\
48 ";
49
50
51 const char*
52 s_vertexShaderScene =
53 "                          \n\
54 #version 420 core                \n\
55 uniform mat4 MVP;                \n\
56 uniform mat4 LightMVP;               \n\
57 uniform sampler2D shadowMapTex;          \n\
58 layout(location = 0) in vec3 inPosition;     \n\
59 layout(location = 1) in vec3 inNormal;       \n\
60 layout(location = 0) out vec4 outCPosition;    \n\
61 layout(location = 1) out vec4 outLPosition;    \n\
62 layout(location = 2) out vec3 outNormal;     \n\
63 void main()                      \n\
64 {                        \n\
65 gl_Position  = MVP * vec4(inPosition,1);  \n\
66 outCPosition = gl_Position;          \n\
67 outLPosition = LightMVP * vec4(inPosition,1); \n\
68 outLPosition.z += 2.4;  \n\
69 //outLPosition = outLPosition*0.5 + vec4(0.5,0.5,0.5,0); \n\
70 outNormal  = normalize(inNormal);          \n\
71 }                        \n\
72 \n\
73 ";
74
75
76 const char*
77 s_fragmentShaderScene =
78 "                          \n\
79 #version 420 core                \n\
80 uniform sampler2D shadowMapTex;          \n\
81 uniform mat4 LightMVP;               \n\
82 layout(location = 0) in vec4 inCPosition;    \n\
83 layout(location = 1) in vec4 inLPosition;    \n\
84 layout(location = 2) in vec3 inNormal;       \n\
85 layout(location = 0) out vec4 outColour;     \n\
86 void main()                      \n\
87 {                        \n\
88 //vec4 abc = inLPosition*0.5 + vec4(0.5,0.5,0.5,0); \n\
89 vec4 ProjectionCoords = inLPosition / inLPosition.w; \n\
90 vec2 UVCoords;                 \n\
91 //UVCoords = ProjectionCoords.xy;// * 1.0 + vec2(0.05,0.05);                \n\
92 UVCoords.x = 0.5 * ProjectionCoords.x + 0.5; \n\
93 UVCoords.y = 0.5 * ProjectionCoords.y + 0.5; \n\
94 float depth = texture2D( shadowMapTex, UVCoords.st).z; \n\
95 float shadow = 1.0;                \n\
96 ProjectionCoords.z += 0.0001;        \n\
97 if ( inLPosition.w > 0 )           \n\
98 if ( depth < ProjectionCoords.z )      \n\
99 shadow = 0.5;              \n\
100 outColour = vec4(0, 1, 0, 1) * shadow;     \n\
101 float diff = dot(inNormal, vec3(0.1,0.9,0)); \n\
102 outColour *= diff;               \n\
103 outColour.a = 1.0;               \n\
104 }                        \n\
105 ";
106
107
108
109 GLuint LoadShaders(const char* vShader, const char* fShader)
110 {
111 // Create the shaders
112 GLuint vertexShaderID   = glCreateShader(GL_VERTEX_SHADER);
113 GLuint fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);
114
115 GLint result = GL_FALSE;
116 int infoLogLength;
117
118 // Compile Vertex Shader
119 glShaderSource(vertexShaderID, 1, &vShader, NULL);
120 glCompileShader(vertexShaderID);
121
122 // Check Vertex Shader
123 glGetShaderiv(vertexShaderID, GL_COMPILE_STATUS, &result);
124 glGetShaderiv(vertexShaderID, GL_INFO_LOG_LENGTH, &infoLogLength);
125 std::vector<char> vertexShaderErrorMessage(infoLogLength);
126 glGetShaderInfoLog(vertexShaderID, infoLogLength, NULL, &vertexShaderErrorMessage[0]);
127 printf("%s\n", &vertexShaderErrorMessage[0]);
128 DBG_ASSERT(result==1);
129
130
131 // Compile Fragment Shader
```

```
132 glShaderSource(fragmentShaderID, 1, &fShader, NULL);
133 glCompileShader(fragmentShaderID);
134
135 // Check Fragment Shader
136 glGetShaderiv(fragmentShaderID, GL_COMPILE_STATUS, &result);
137 glGetShaderiv(vertexShaderID, GL_INFO_LOG_LENGTH, &infoLogLength);
138 std::vector<char> fragmentShaderErrorMessage(infoLogLength);
139 glGetShaderInfoLog(fragmentShaderID, infoLogLength, NULL, &fragmentShaderErrorMessage[0]);
140 printf("%s\n", &fragmentShaderErrorMessage[0]);
141 DBG_ASSERT(result==1);
142
143 // Link the program
144 printf("Linking program\n");
145 GLuint programID = glCreateProgram();
146 glAttachShader(programID, vertexShaderID);
147 glAttachShader(programID, fragmentShaderID);
148 glLinkProgram(programID);
149
150 // Check the program
151 glGetProgramiv(programID, GL_LINK_STATUS, &result);
152 glGetProgramiv(programID, GL_INFO_LOG_LENGTH, &infoLogLength);
153 std::vector<char> programErrorMessage(infoLogLength + 1);
154 glGetProgramInfoLog(programID, infoLogLength, NULL, &programErrorMessage[0]);
155 printf("%s\n", &programErrorMessage[0]);
156
157 glDeleteShader(vertexShaderID);
158 glDeleteShader(fragmentShaderID);
159
160 return programID;
161 }
162
163
164 void GenerateShadowFBO(GLuint& outDepthTextureId, GLuint& outFboId)
165 {
166 float shadowMapWidth  = SCREEN_WIDTH  * SHADOW_SCALE_FACTOR;
167 float shadowMapHeight = SCREEN_HEIGHT * SHADOW_SCALE_FACTOR;
168
169 GLenum FBOstatus;
170
171 // Try to use a texture depth component
172 glGenTextures(1, &outDepthTextureId);
173 glBindTexture(GL_TEXTURE_2D, outDepthTextureId);
174 // FIX***FIX***FIX*** - Certain graphics cards - ATI/NVidia won't work unless you explicity create
175 // texture - safe to create it and know it works for all
176 glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32F, (int)shadowMapWidth, (int)shadowMapHeight, 0, ↩
        GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
177
178
179 // GL_LINEAR does not make sense for depth texture. However, next tutorial shows usage of GL_LINEAR and PCF
180 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
181 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
182
183 // Remove artefact on the edges of the shadowmap
184 glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
185 glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP );
186
187 // create a framebuffer object
188 glGenFramebuffersEXT(1, &outFboId);
189 glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, outFboId);
190
191 glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT, GL_TEXTURE_2D, ↩
        outDepthTextureId, 0);
192
193 // check FBO status
194 FBOstatus = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
195 if(FBOstatus != GL_FRAMEBUFFER_COMPLETE_EXT)
196 {
197 printf("GL_FRAMEBUFFER_COMPLETE_EXT failed, CANNOT use FBO\n");
198 DBG_ASSERT(false);
199 }
200 // switch back to window-system-provided framebuffer
201 glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
202 }
203
204
205
206 void main()
207 {
208 GLFWwindow* window = NULL;
209
210 // Initialize the library
211 if (!glfwInit())
212 {
213 DBG_ASSERT(false); // glwf init failed
214 return;
215 }
216
217 // Create a windowed mode window and its OpenGL context
218 window = glfwCreateWindow(SCREEN_WIDTH, SCREEN_HEIGHT, "Hello Graphics", NULL, NULL);
219 if (!window)
220 {
221 DBG_ASSERT(false); // failed create window
```

```
222 glfwTerminate();
223 return;
224 }
225
226 // Make the window's context current
227 glfwMakeContextCurrent(window);
228
229 if(glewInit() != GLEW_OK)
230 {
231 DBG_ASSERT(false); // Glew failed to init
232 return;
233 }
234
235 GLuint progShadowID = LoadShaders(s_vertexShaderShadow, s_fragmentShaderShadow);
236 GLuint progSceneID  = LoadShaders(s_vertexShaderScene,  s_fragmentShaderScene);
237
238 Vector3 lightPos = Vector3(2,3,3);
239
240 //{
241 glBindAttribLocation(progSceneID, 0, "inPosition");
242 glBindAttribLocation(progSceneID, 1, "inNormal");
243
244 Matrix4 p = Matrix4::Perspective(45,
245 (float)SCREEN_WIDTH/(float)SCREEN_HEIGHT,
246 2.5f,
247 400.0f);
248 Matrix4 veye = Matrix4::LookAt( Vector3(0,8,8),
249 Vector3(0,0,0),
250 Vector3(0,1,0));
251
252 Matrix4 vlight = Matrix4::LookAt(lightPos,
253 Vector3(0,0,0),
254 Vector3(0,1,0));
255 //}
256
257 vector<Cube*> cubes;
258 cubes.push_back( new Cube(Vector3(0,0, 0), Vector3(1,  1.0f, 1) ) );
259 cubes.push_back( new Cube(Vector3(1,-1.0f,0), Vector3(5,  0.1f, 5) ) );
260 cubes.push_back( new Cube(lightPos,       Vector3(0.05f,0.05f,0.05f) ) );
261
262 POINT ptLastMousePosit;
263 POINT ptCurrentMousePosit;
264
265 //{
266 GLuint depthTextureId;
267 GLuint fboId;
268 GenerateShadowFBO(depthTextureId, fboId);
269 //}
270
271 // Loop until the user closes the window
272 while (!glfwWindowShouldClose(window))
273 {
274 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
275 glClearColor(1,0,0,1);
276
277 Matrix4 m = Matrix4::CreateAxisAngleRotation(-g_fSpinX, Vector3(0,1,0)) *
278 Matrix4::CreateAxisAngleRotation(-g_fSpinY, Vector3(1,0,0));
279
280 #if 1
281 {
282 glBindFramebufferEXT(GL_DRAW_FRAMEBUFFER_EXT, fboId);
283
284 // In the case we render the shadowmap to a higher resolution, the viewport must be modified accordingly.
285 glViewport(0,0,(int)(SCREEN_WIDTH*SHADOW_SCALE_FACTOR), (int)(SCREEN_HEIGHT*↩
        SHADOW_SCALE_FACTOR));
286 // Clear previous frame values
287 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
288 glEnable(GL_DEPTH_TEST);
289 // Culling switching, rendering only backface, this is done to avoid self-shadowing
290 glLinkProgram(progShadowID);
291 glUseProgram(progShadowID);
292
293 GLint MVLoc      = glGetUniformLocation(progShadowID, "MVP");
294 DBG_ASSERT(MVLoc>= 0);
295
296 for (int i=0; i<(int)cubes.size()-1; ++i)
297 {
298 glUniformMatrix4fv(MVLoc, 1, FALSE, (float*)&( (cubes[i]->modelMatrix * vlight * p)));
299 glFrontFace(GL_CCW);
300 glCullFace(GL_FRONT);
301
302 cubes[i]->Render();
303 }
304
305 }
306 #endif
307
308 {
309 // Now rendering from the camera POV, using the FBO to generate shadows
310 glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,0);
311 glViewport(0,0,(int)(SCREEN_WIDTH), (int)(SCREEN_HEIGHT));
312 glLinkProgram(progSceneID);
313 glUseProgram(progSceneID);
```

```
314
315 GLint LightMVLoc  = glGetUniformLocation(progSceneID, "LightMVP");
316 DBG_ASSERT(LightMVLoc>= 0);
317 GLint MVPLoc     = glGetUniformLocation(progSceneID, "MVP");
318 DBG_ASSERT(MVPLoc>= 0);
319 GLint shadTexLoc = glGetUniformLocationARB(progSceneID, "shadowMapTex");
320 DBG_ASSERT(shadTexLoc>= 0);
321
322 glActiveTexture(GL_TEXTURE0);
323 glBindTexture(GL_TEXTURE_2D, depthTextureId);
324 glUniform1iARB(shadTexLoc,   0);
325
326 for (int i=0; i<(int)cubes.size(); ++i)
327 {
328 glUniformMatrix4fv(LightMVLoc, 1, FALSE, (float*)&(cubes[i]->modelMatrix * vlight * p));
329 glUniformMatrix4fv(MVPLoc,      1, FALSE, (float*)&(cubes[i]->modelMatrix * m * veye    * p));
330
331 //glUseProgram(progSceneID);
332 glFrontFace(GL_CCW);
333
334
335 glLineWidth(1.5);
336 glEnable(GL_LINE_SMOOTH);
337
338
339 glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
340 cubes[i]->Render();
341
342 glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
343 cubes[i]->Render();
344
345 }
346
347 }
348
349
350
351
352 #if 1
353 glUseProgramObjectARB(0);
354 glViewport(0,0,(int)(SCREEN_WIDTH), (int)(SCREEN_HEIGHT));
355 glMatrixMode(GL_PROJECTION);
356 glLoadIdentity();
357 glOrtho(-SCREEN_WIDTH/2,SCREEN_WIDTH/2,-SCREEN_HEIGHT/2,SCREEN_HEIGHT/2,1,20);
358 glMatrixMode(GL_MODELVIEW);
359 glLoadIdentity();
360 glColor4f(1,1,1,1);
361 glActiveTextureARB(GL_TEXTURE0);
362 glBindTexture(GL_TEXTURE_2D,depthTextureId);
363 glEnable(GL_TEXTURE_2D);
364 glDisable(GL_BLEND);
365 glTranslated(0,0,-1);
366 glBegin(GL_QUADS);
367 glTexCoord2d(0,0);glVertex3f(0,0,0);
368 glTexCoord2d(1,0);glVertex3f(SCREEN_WIDTH/2.0f,0,0);
369 glTexCoord2d(1,1);glVertex3f(SCREEN_WIDTH/2,SCREEN_HEIGHT/2.0f,0);
370 glTexCoord2d(0,1);glVertex3f(0,SCREEN_HEIGHT/2.0f,0);
371 glEnd();
372 glDisable(GL_TEXTURE_2D);
373 #endif
374
375 // Swap front and back buffers
376 glfwSwapBuffers(window);
377
378 // Poll for and process events
379 glfwPollEvents();
380
381 #if 1
382 {
383 POINT ps;
384 GetCursorPos(&ps);
385 if ( !GetAsyncKeyState(VK_LBUTTON) ) // || !( GetFocus() == hwnd ))
386 {
387 ptLastMousePosit.x = ptCurrentMousePosit.x = ps.x;
388 ptLastMousePosit.y = ptCurrentMousePosit.y = ps.y;
389 }
390 else
391 {
392 ptCurrentMousePosit.x = ps.x;
393 ptCurrentMousePosit.y = ps.y;
394
395 if( true )
396 {
397 g_fSpinX -= (ptCurrentMousePosit.x - ptLastMousePosit.x);
398 g_fSpinY -= (ptCurrentMousePosit.y - ptLastMousePosit.y);
399 }
400
401 ptLastMousePosit.x = ptCurrentMousePosit.x;
402 ptLastMousePosit.y = ptCurrentMousePosit.y;
403 }
404 }// End mouse input
405 #endif
406 }
```

```
407
408 glfwTerminate();
409 }// End main(..)
```

Listing 2: Compact shadow mapping implementation in OpenGL 4.4.