Fifth Assignment Report

Guided by Professor Michael Manzke

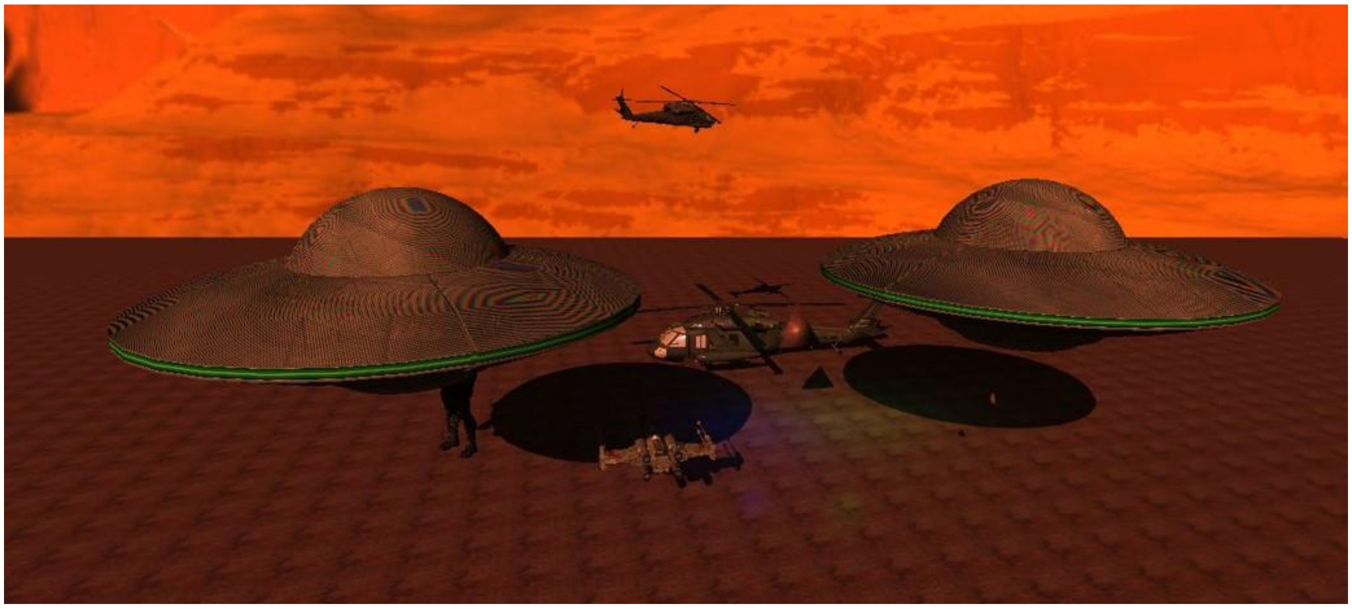
Implemented by: Yuzhou Shao

Student id: 19322035

Youtube Demo: https://youtu.be/Rljm68Xr00w

Abstract:

In this project, I have implemented direction light. I loaded a set of static or rotating objects in the scene. Below those objects, I have created a vast brick ground with repeated texture mapping of a brick.png file. In order to make the performance more photorealistic, a beautiful skybox is implemented. Everything is inside that skybox.



Therefore, the shadows of all the objects are clearly mapped on the brick ground. The shadows are static or rotating according to the status of the objects, respectively. I have used different techniques mentioned in the next sections to solve the defects and issues that occurred on Shadow mapping. And evaluating their performances.

Background:

I have mainly followed the paper, M. Stamminger and G. Drettakis 2002, ‘Perspective Shadow Maps’ in ACM Transactions on Graphics. Perspective shadow maps have been introduced in this paper. It is a creative parameterization for shadow maps.

Perspective Shadow Maps

Marc Stamminger and George Drettakis

REVES - INRIA Sophia-Antipolis, France *

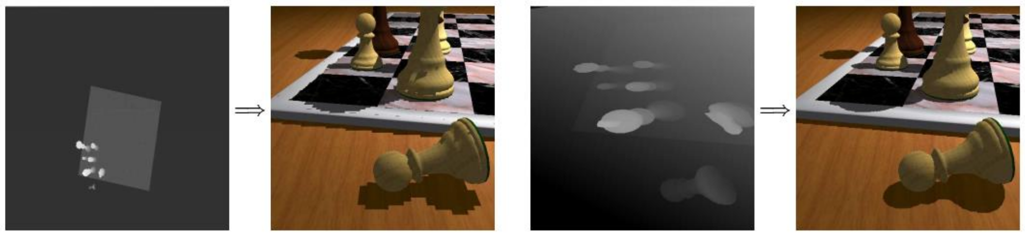


Figure 1: (Left) Uniform 512x512 shadow map and resulting image. (Right) The same with a perspective shadow map of the same size.

The approach permits the generation of shadow maps with vastly improved quality. The resolution is considered while appropriated by selecting a suitable projective mapping. The authors have presented the point rendering method, which enables interactive presenting of very complex scenes with high-quality shadows.

In conclusion, as the authors said, perspective shadow maps can be utilized in interactive applications and fully exploit shadow map capabilities of recent graphics hardware. Still, they are also applicable to high-quality software renderers.

Implementation details:

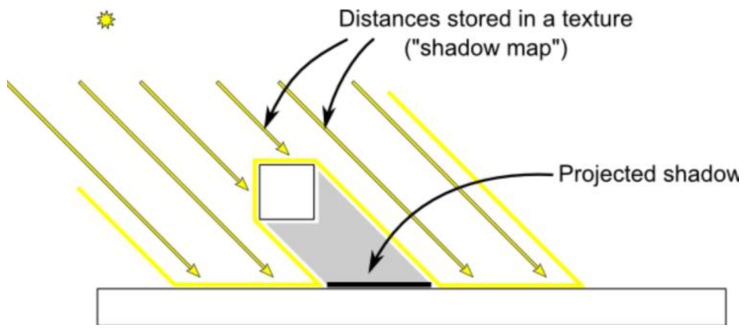
As I briefly mentioned in my PPT presentation, shadows are created by texture maps of depth data. Depth data created by rendering the scene from point of view of light source. I implemented one pass to generate shadow maps and another to render a scene.

Compare the depth of fragment from light’s perspective to value on shadow map texture. I added bias to remove shadow acne, set values from beyond the sampling region to 0, and finally utilize PCF algorithms to fade shadow edges.

In detail, I first need to create a shadows' "map" by a light. Use this map to determine where not to apply light. The map is held as a 2D texture. The map is created using a "Framebuffer," which is then written to texture.

Therefore, a minimum of two rendering passes is required—one for creating a shadow map and the second for drawing scenes.

For the first pass: render the scene from the perspective of a light source.



Shaders do not merely create color outputs. In per-sample operations, depth tests using depth buffer values. A depth buffer is another buffer and color buffer that holds a value between 0 and 1, determining how deep a fragment is to the frustum. 0 is on the near plane to the camera, and 1 is far from the camera.

Frame buffer object extracts depth buffer data. Typically, Framebuffer bound is '0'. This is the default buffer. We can find a separate Frame buffer and draw to that.

The shader follows the below concepts:

1. Apply Projection and View matrices as if the light source is the camera.
2. Apply model matrix on each object.
3. Fragment Shader is not even needed: Depth buffer is written automatically.

Directional Light shadow map works differently to Point/Spot Light shadow maps, and the directional light rays are all parallel but not fan out.

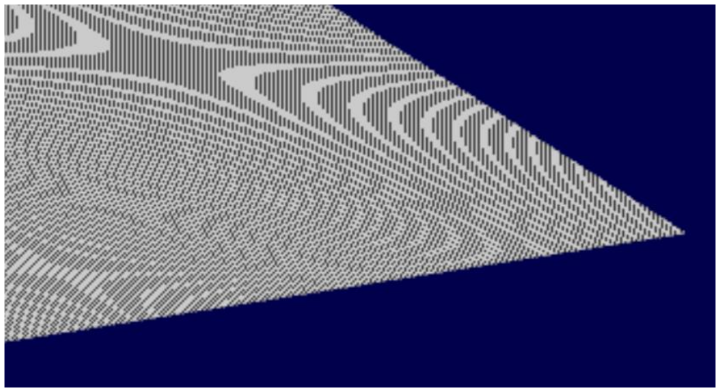
To solve this problem, I utilized Orthographic Projection Matrix.

glm::ortho(-20.0f, 20.0f, -20.0f, 20.0f, 0.01f, 100.0f);

Next, we go to Using Map. Since the texture bound to it is occupied with Shadow Map data. I unbind the Frame buffer used for the shadow map. Then I bound the texture to my main shader and utilized it.

I accessed the View Matrix used in Shadow Map Shader and used the shader to get the current fragment position with the light source. I then created an approach to access points on the shadow Map with the light source perspective's fragment coordinates. Therefore, converting the light source perspective fragment's coordinates to "Normalized Device Coordinates" is necessary.

However, there is Shadow Acne occurs due to resolution issues. To eliminate Shadow Acne, I have added a slight bias.



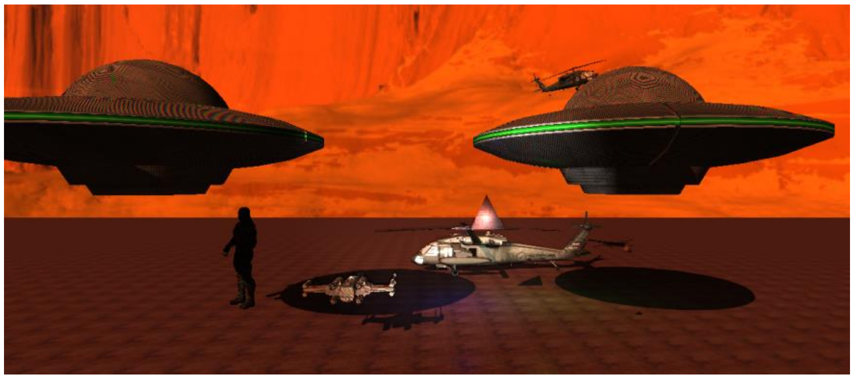
The Bias offset causes areas close to the shadow source to disappear because depth values are close.

However, the areas outside of the projection frustum face the issue of oversampling. My solution was setting texture type to use the border with values all consisting of 0. For values beyond the far plane, which greater than 1, I initialized it to 0. The problem got solved.

The last technique is PCF (Percentage-Closer Filtering). Because the edges of shadows are limited to the resolution of the texture shadow map is written to. It leads to unsightly pixelated edges. My solution is to sample the surrounding Texels and calculate the average. Apply merely partial shadows for shadowed areas.

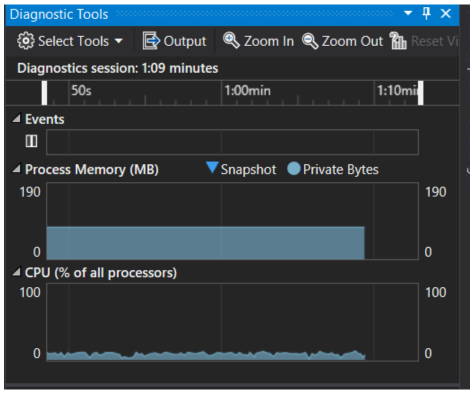
For example, the shadow value calculated as 5, and 8 samples are taken. 5/8=0.625, so apply 63% shadow to that pixel.

Below is my output:



Results/Evaluation:

I can handle it very smoothly in real-time, depending on how sensitive my mouse or keyboard I configurated. On Windows 10, the project averagely hogs up 90MB Process Memory and approximately 10% of the CPU of all processors.

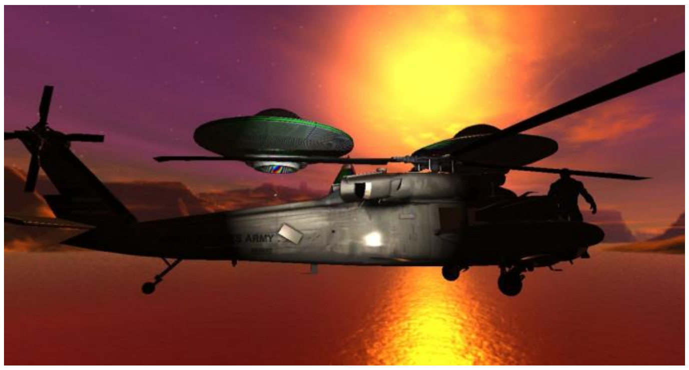


I have implemented shadow maps within an interactive rendering application using OpenGL. The output performance appears on my Youtube demo links given in the beginning.

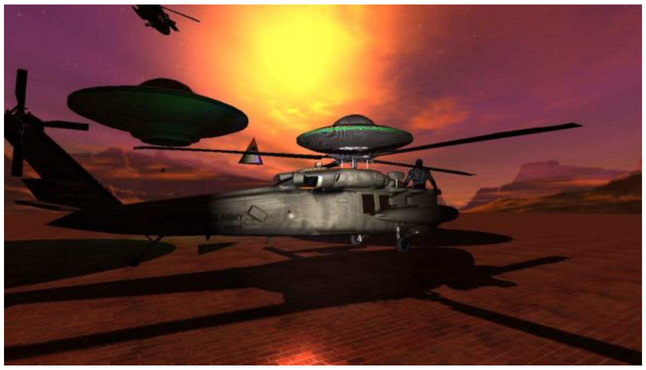
I utilized GL_SGIX_depth_texture and GL_SGIX_shadow. The shadow maps are rendered into p-buffers.

To get the optimized results from shadow maps, the near plane of the perspective camera view should be as far as possible. I implemented it successfully by reading back the depth buffer after each frame and configuring the near plane accordingly.

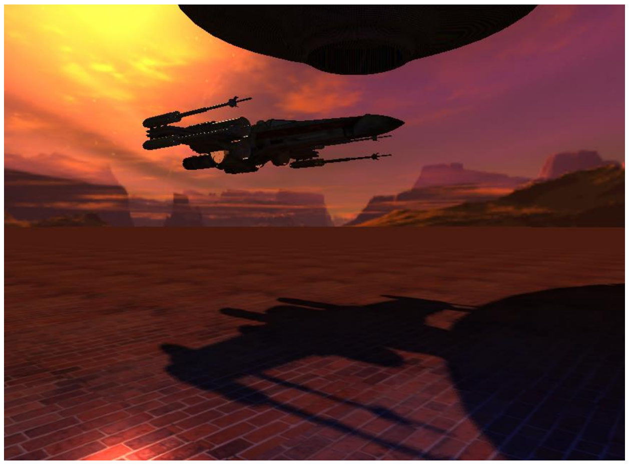
Without Shadow Mapping

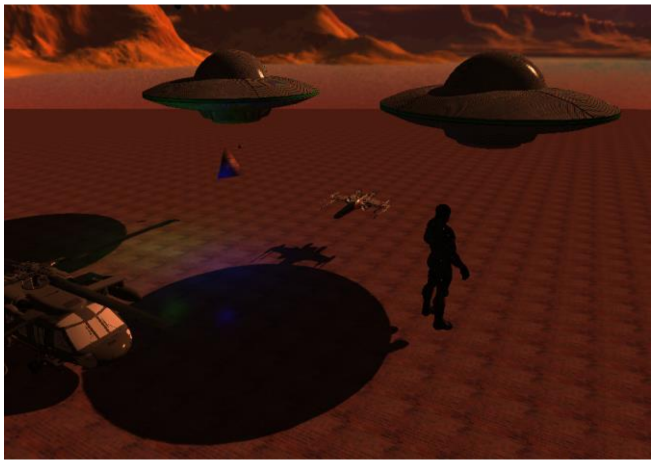


With Shadow Mapping



As down left, the edge of the fighter's shadow fades out, which proves PCF has been implemented successfully.





Shadow Acne has been eliminated as downright shows that prove the Bias Offset has been configurated successfully.

Improvements/Limitations:

No MIP Mapping Implemented on Shadow

There is no MIP Mapping on the shadow class implemented yet. Suppose the object is large, the shadow also becomes large, there will be Moire pattern, which influences the performance, and extra unnecessary calculations will be largely occupied.

Aliasing Problem

Aliasing in shadow maps is due to the synchronization of the two depth maps. Separately rendering the scene for each light source is required. This takes more time for an omnidirectional light point. It requires a 180-degree shadow frustum that must be handled by buffering.

A problem occurs when a light source is located inside the scene, as this condition requires six buffers to handle all shadow cases.

Inappropriate self-shadowing of the shadow maps is another limitation. This is because of the imprecision and under-sampling of the depth values that store a shadow map per texel. As I mentioned in the previous section, biasing is for obtaining robust results.

Proper Citation of papers, Source code, Libraries, text, images and assets used:

Papers:

M. Stamminger and G. Drettakis 2002, ‘Perspective Shadow Maps’ in ACM Transactions on Graphics

D. Weiskopf and T. Ertl 2003, ‘Shadow Mapping Based on Dual Depth Layers’ in University of Stuttgart

H. Kolivand, 2015, Shadow mapping algorithms Applications and limitations in Semantic Scholar

[PDF] Shadow mapping algorithms: Applications and limitations | Semantic Scholar

Tutorial

Computer Graphics with Modern OpenGL and C++ Udemy tutorial by Ben Cook 2017 available from

OpenGL + C++: Modern Graphics for Groundbreaking Games | Udemy

Some of my Source codes, including Texture Class, Camera Class, DirectionalLight Class, PointLight Class, SpotLight Class, Shader Class, and Material Class, are followed and referenced from the above tutorial I modified and wrote in my own approaches.

Libraries:

GLFW GLM GLEW

Assets:

Fighter, Black Hawk Helicopter, UFOs, Nanosuit were downloaded from

Free 3D Ufo Models | TurboSquid