Project 2: Moment Shadow Map

Name: Minsuk Kim(m.kim)
Instructor: Dr. Gary Herron
Class: CS562
Semester: Spring 2022
Date: 2/13/2022

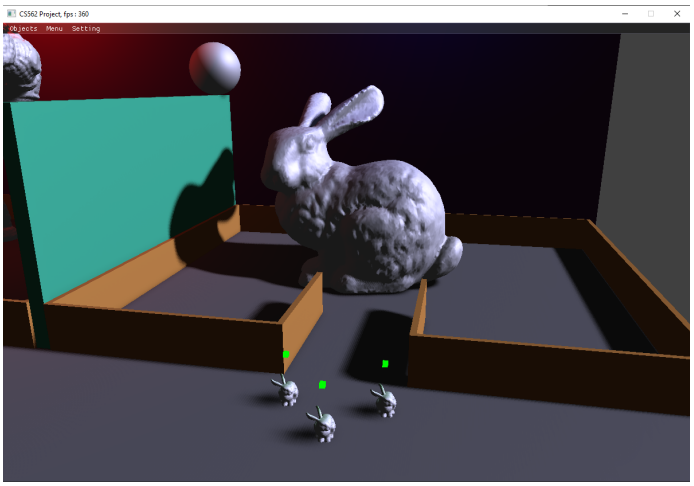


Table of Contents

Introduction **2**
- Overview 2
- Implementation 3

Result Images **4**
- Images with MSM algorithm 5
- Depth texture images 6
- Images with filtering 7

Notes **8**
- Quantization 8
- Gaussian blur 11
- Thread group shared memory 12
- Moment Shadow Mapping 13
- Debugging 13

Introduction

Overview

The main purpose of the project is to implement moment shadow rendering. The moment shadow rendering technique is the most efficient shadow calculation technique for the 64 bits size texel map. It stores depth values in 4 channels for each texel. It uses the 4 channel depth data to compute the sharpest possible lower bound as an approximation to the shadow intensity.

In this project, I will cover the Hamburger four-moment shadow mapping algorithm which works best among the MSM algorithms. The program will have three passes for the shadow rendering: shadow mapping pass, blur pass and handle shadow pass. The shadow mapping pass uses the closest depth value to store in four-channel texels with depth, depth squared, depth cubed and depth to the power of four. Then, the program does quantization texels with the range 0 to 1. The blur pass uses both vertical and horizontal Gaussian blur to filter the image. After filtering the image, the handle shadow pass reads the data and de-quantization the depth values to calculate the shadow values of a certain point. After calculating the shadow value, the program multiplies the diffuse and specular color of the scene by the shadow value.

I started the project with Vulkan API and GLFW. I also used an external library Assimp that read the object files.

Implementation

The shadow mapping algorithm is saving the distance value between objects and lights that are casting the shadow to a depth texture. It reads the depth texture to determine if there is a shadow or not in each point by comparing the distance between lights and pixels with the depth value. The MSM algorithm filtered the depth texture and got the shadow value for soft shadows. To implement the moment shadow rendering, the rendering sequence for MSM will be the following: mapping depth texture, filtering the depth texture and calculating the shadow values.

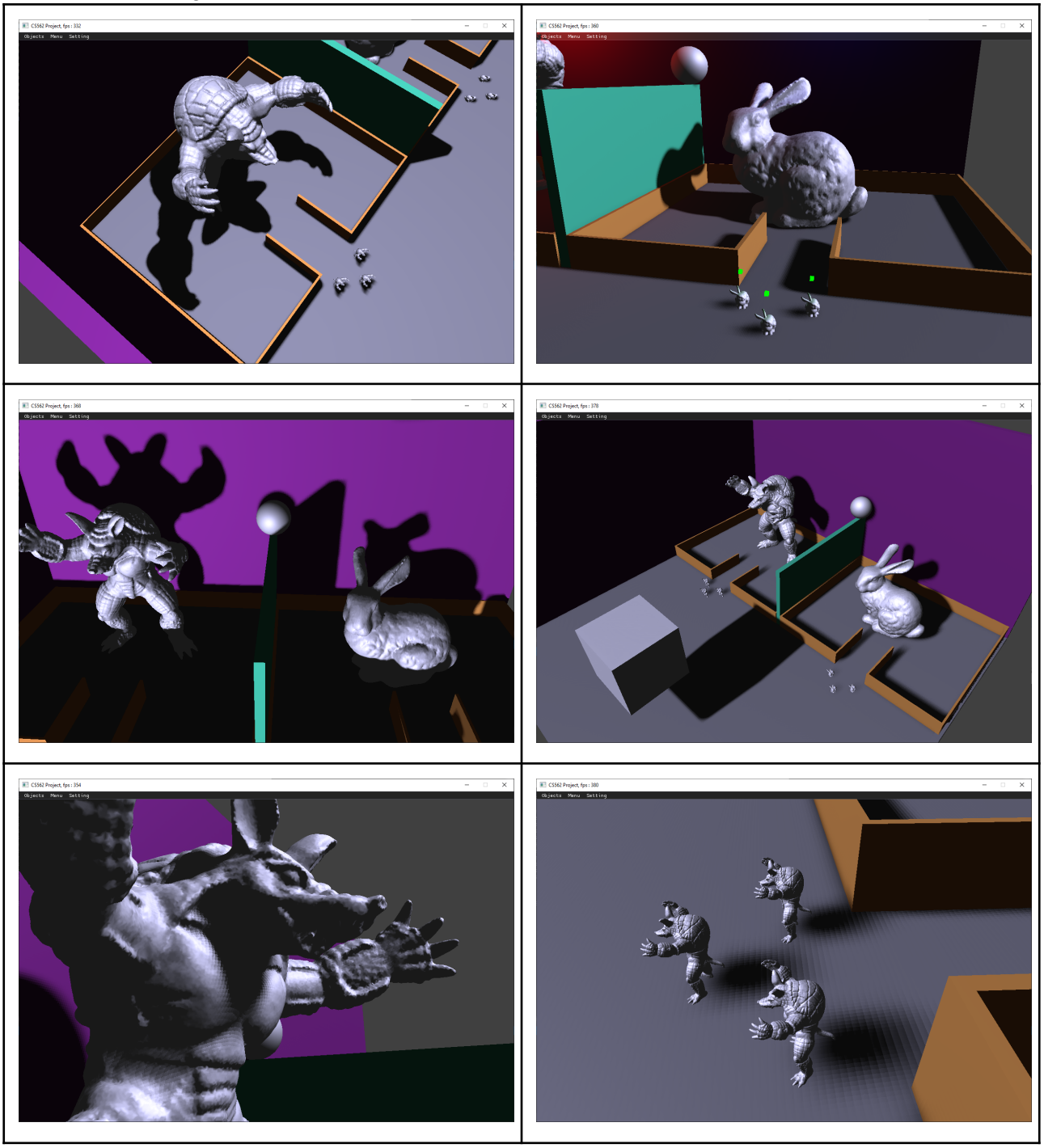
First, the program will calculate the depth value in light space and store it on a four-channel texture. The shader uses a projection matrix to transform vertices from local space to light space. The w value of transformed vertices will be the distance between light and vertices(same as depth in light space). The program gets the depth value by normalizing the w value to range 0 to 1(divide by far_plane of light projection matrix). With the depth value, it stores the value in a four-channel texture by d, d^2, d^3, d^4 where d is depth value. A four-channel texture has 16 bits for each channel so it has 64 bits total for one texel. To use 16 bits efficiently, the program transforms the depth values with quantization so that there will be less data loss on filtering depth maps.

The next step is to filter the depth map. The program uses compute shaders for filtering the depth texture. In this project, the program uses a Gaussian blur algorithm for filtering a depth map. For efficiency, the program executes blur two times: vertical and horizontal. While filtering the depth texture, the program uses threaded group shared memory for memory efficiency. In this project, 128 threads are used.

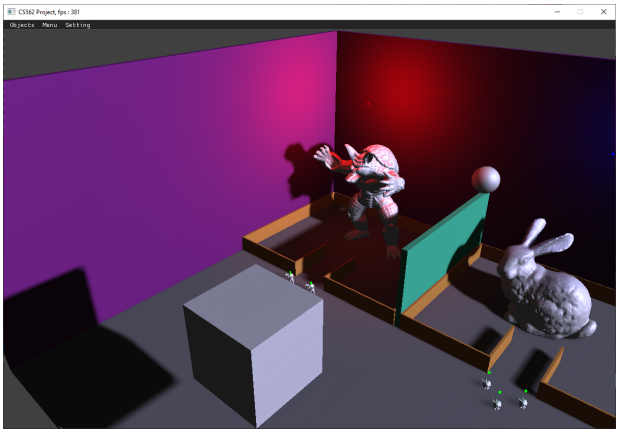
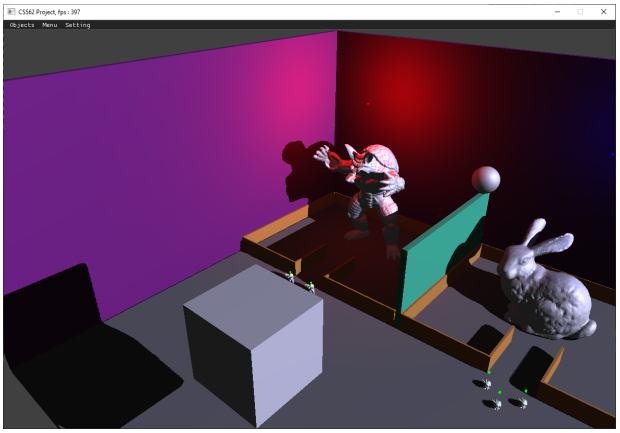
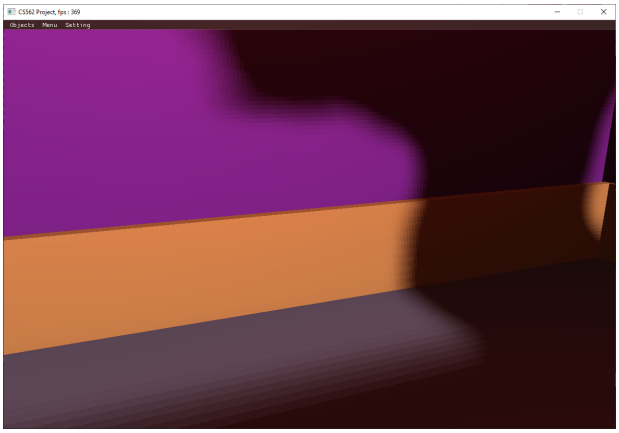
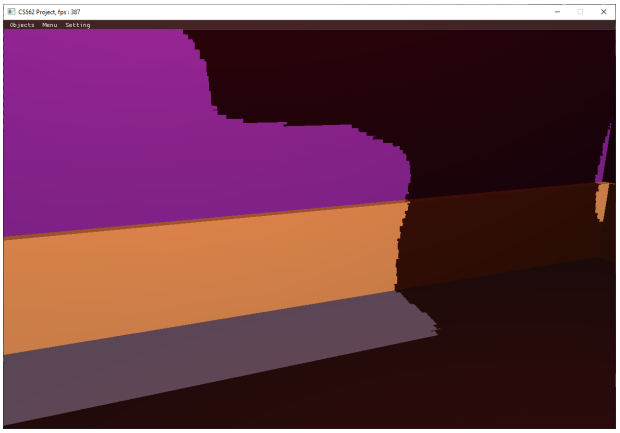
The final step of the shadow mapping algorithm is calculating shadow values in pixel position. The program calculates position in light space. The W value of light space position is the distance between light and the position. The XY is a texture coordinate in depth texture after transforming it to [0, 1] from [-1, 1]. With depth value, the program performs de-quantization of all four data and calculates the value of shadow. The output will be ambient + shadow * (diffuse + specular).

Result Images

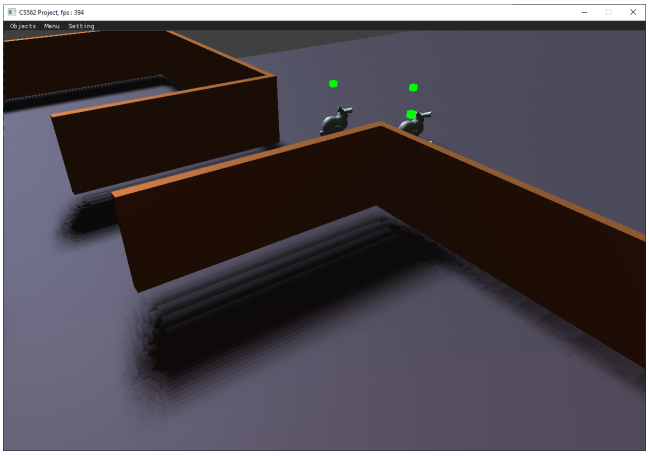
some sample images



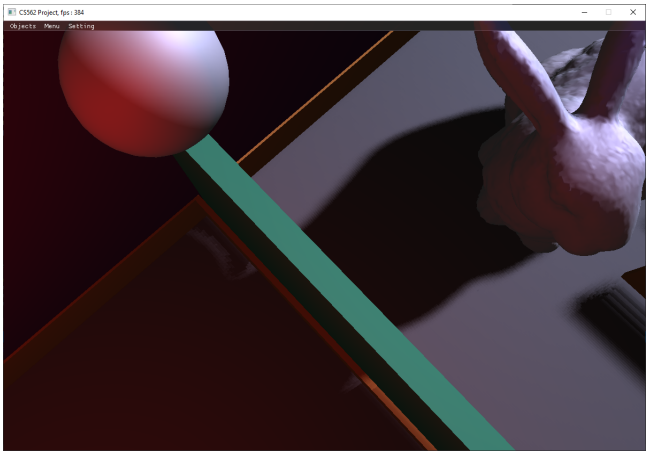
Images with MSM algorithm

MSM algorithm	No MSM algorithm
	
	

some errors occurred with MSM



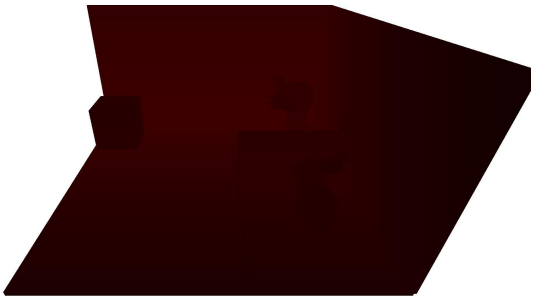
shadows in short distance may not work properly



other shadows may interrupt other shadow

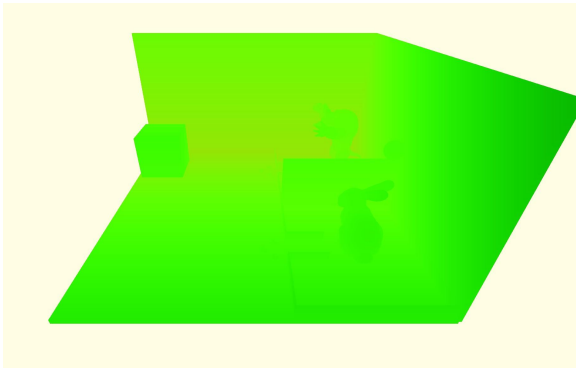
Depth texture images

depth texture with only depth



depth texture with (depth, depth^2, depth^3, depth^4)

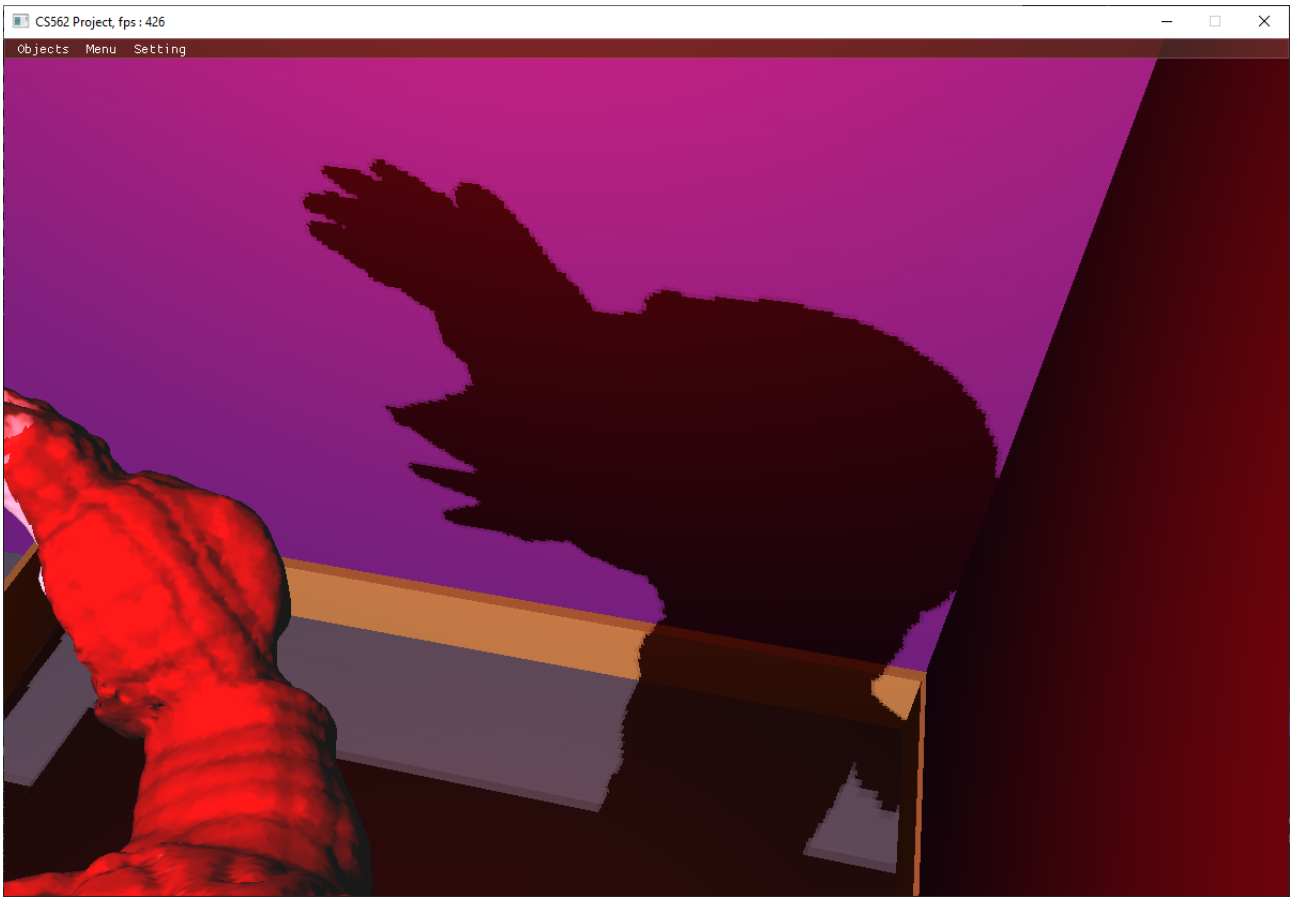
after quantization depth texture



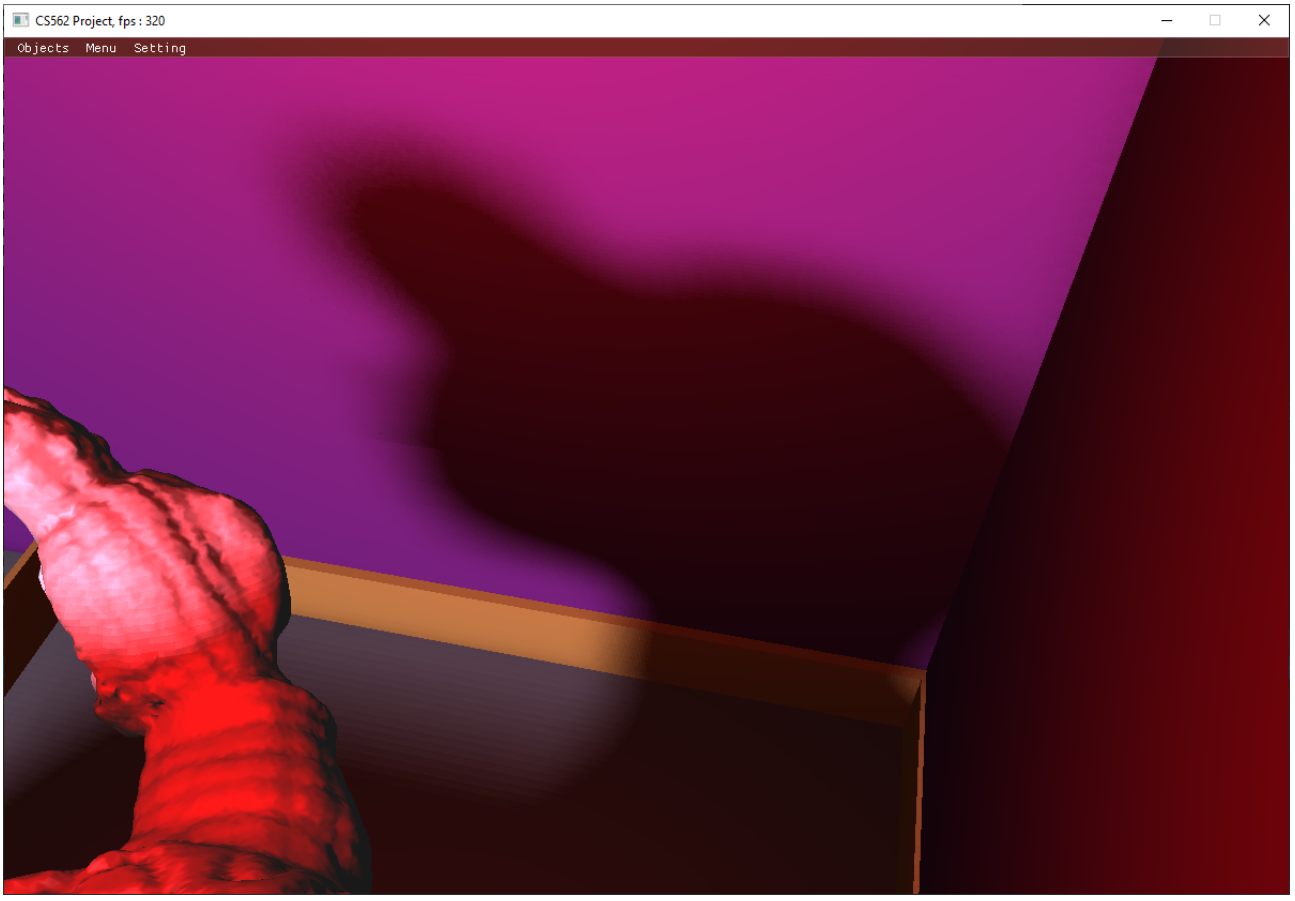
depth texture with other position

Images with filtering

gaussian blur with kernel size 3



gaussian blur with kernel size 43



Notes

Quantization

In this project, we used 16 bits for each channel which is not enough to calculate depths. To prevent data loss from filtering depth images, I have to transform the depth data(depth, depth squared, depth cubed, and depth to the power of four). The program used ‘quantization’ to depth data to maximize entropy for efficiency. The quantization is performed with

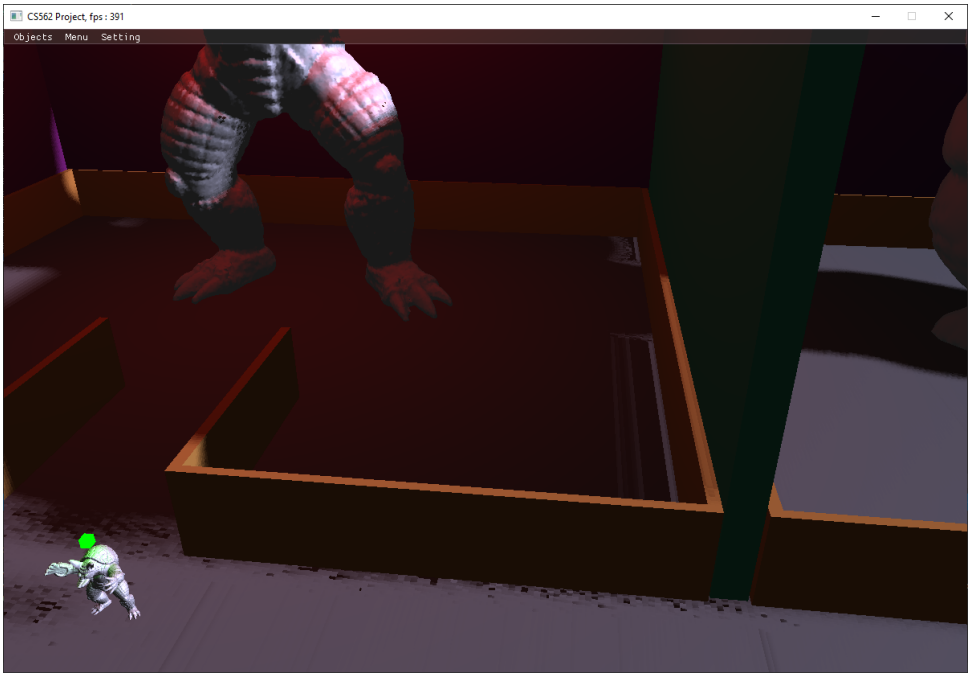
$$b' = C + M \cdot b$$

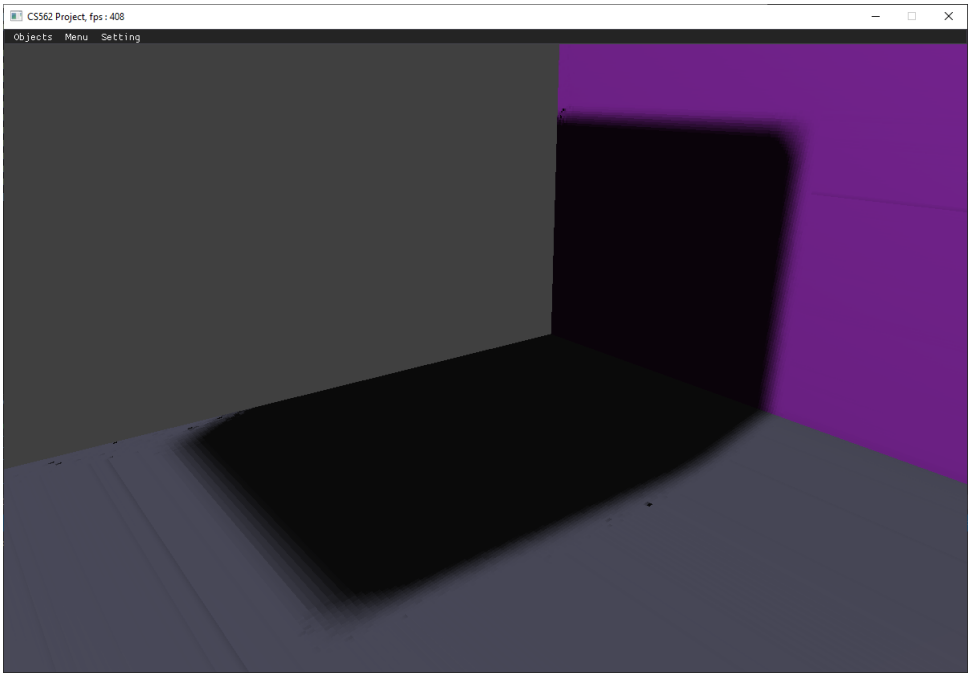
$$(where\ b\ is\ depth\ data\ and\ b'\ is\ quantization\ data)$$

the C is (0.0359558848,0,0,0) and M is (-2.07224649, 32.2370378, -68.5710746, 39.3703274, 13.7948857, -59.4683976, 82.035975, -35.3649032, 0.105877704, -1.90774663, 9.34965551, -6.65434907, 9.79240621, -33.76521106, 47.9456097, -23.9728048).

By using the quantization, we need a higher bias for shadow bias but it works on 16 bits texture well.

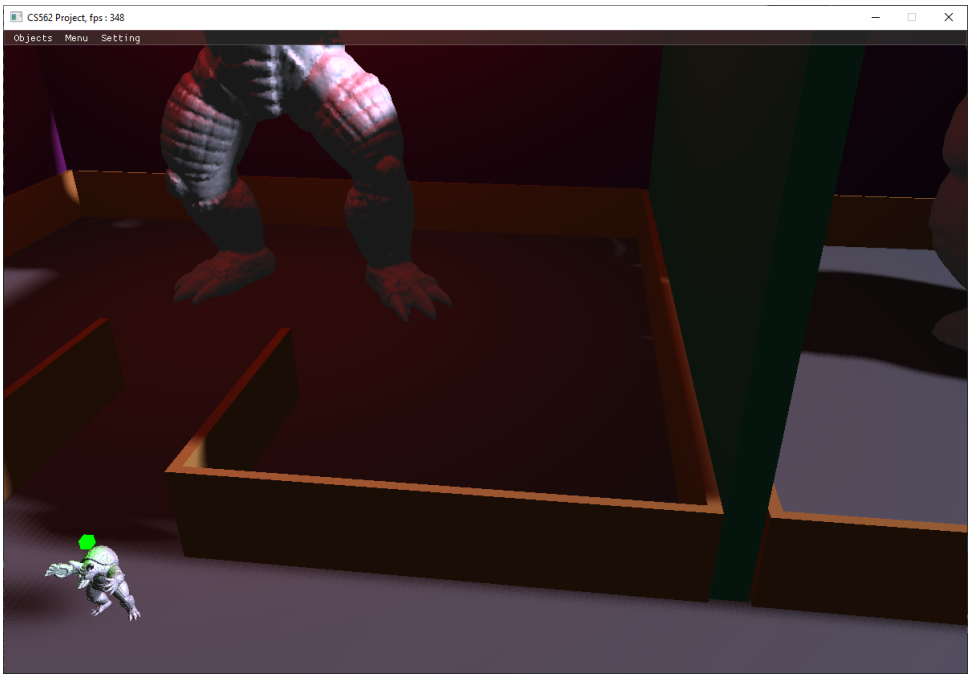
image of not using quantization with 16 bits

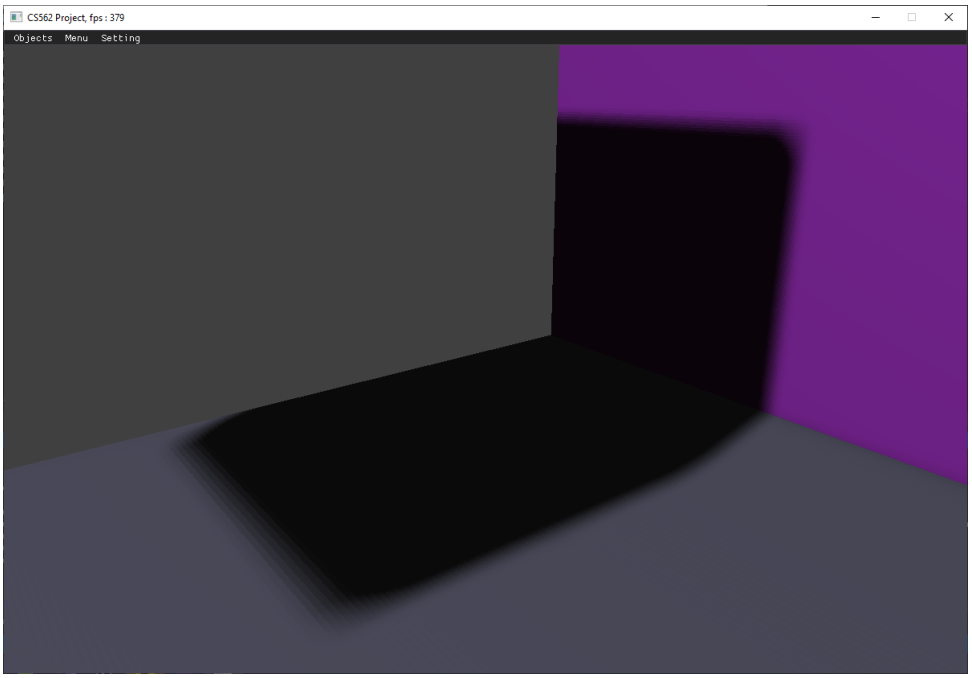




we can see lots of noise and error shadow even though frame rate is high

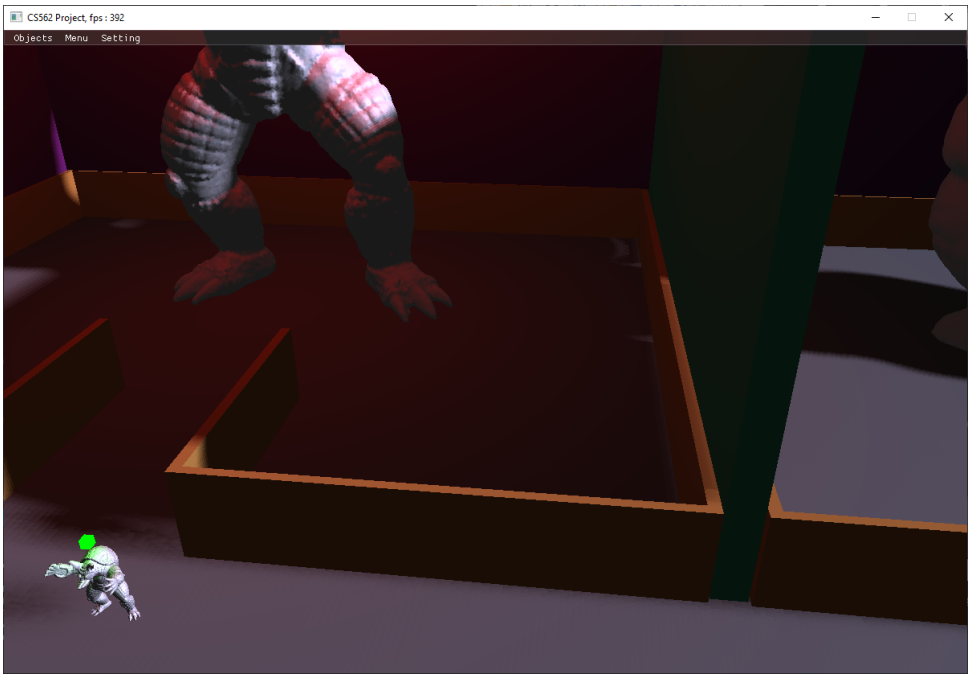
image of not using quantization with 32 bits

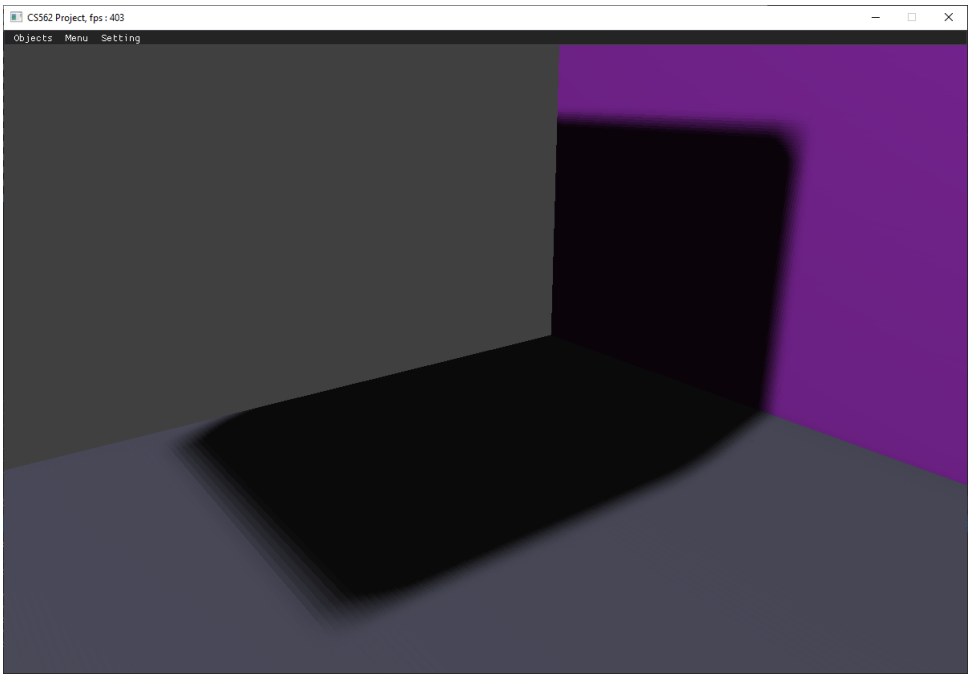




we can see accurate, soft shadows on it. but performance and memory efficiency is not good

image of using quantization with 16 bits





soft, accurate shadow with less memory occupancy with high performance

Gaussian blur

The program used a depth map filtering algorithm called Gaussian Blur with compute shaders. Gaussian blur performs blurring depth values with weight

$$weight(i) = \beta e^{-\frac{2 \cdot i^2}{(w^2)}}$$

where 2w + 1 is kernel size and β is the constant value that makes the sum of weights to be 1. For pixel position i, the program gets the value i - w to i + w and multiplies it with weights to get filtered data. The gaussian blur will be performed vertically and horizontally for efficiency.
This is implemented on blurshadow_vertical.comp and blurshadow_horizontal_comp

Original image before filtering(depth texture)

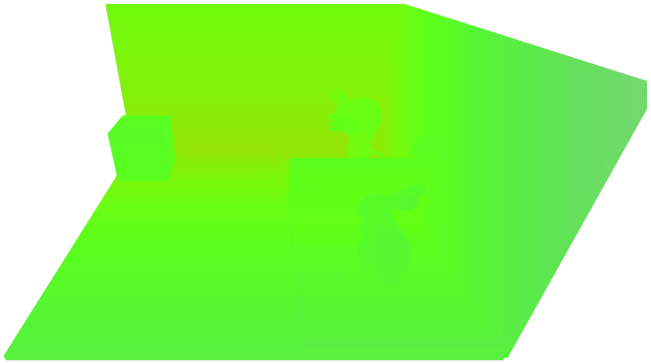
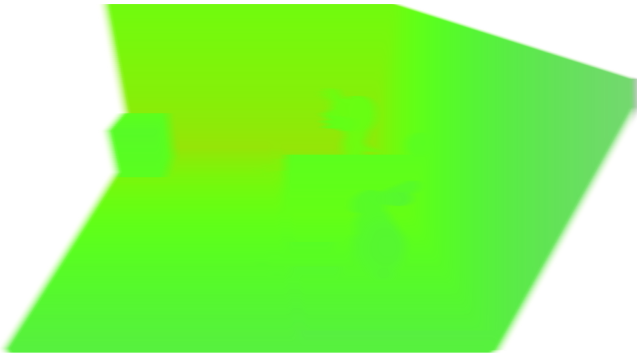


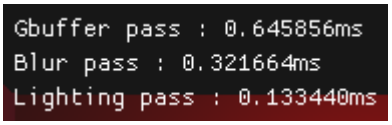
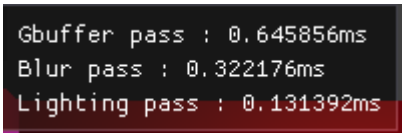
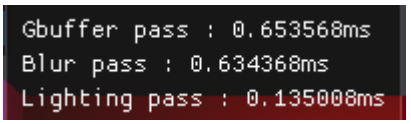
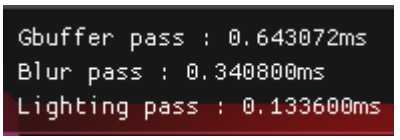
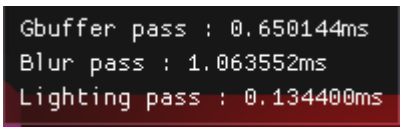
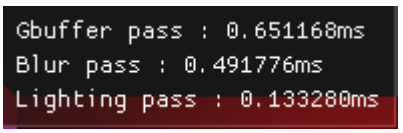
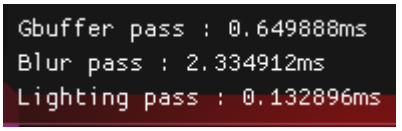
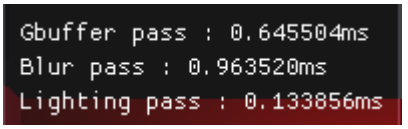
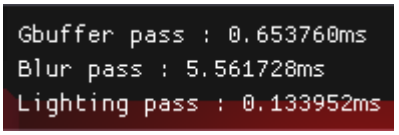
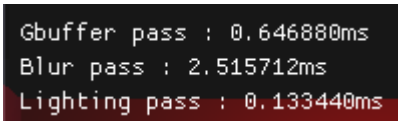
Image after vertical blur with kernel size 43



Final image after blur with kernel size 43

Thread group shared memory

When filtering the depth texture, we can maximize the efficiency of the thread in the compute shader. The thread group shared memory(TGSM) is using shared memory that all threads share and work on it in parallel. In this project, there will be 128 threads that load depth image data and store it to target image data. Each thread reads the one image data from the depth map except for the thread that has an ID less than 2 * w(where 2 w + 1 is kernel size). Threads store it in shared memory. After all loading ending, each thread computes the filtered data from the shared memory.

	Direct Compute(No TGSM)	Use Thread Group Shared memory
kernel = 3		
kernel = 11		
kernel = 19		
kernel = 43		
kernel = 121		

We can see that there is no big difference in low kernel size(In fact, direct computing is slightly faster than using TGSM method in kernel size = 3) but we can improve performance in large kernel size with TGSM.

This is implemented on blurshadow_vertical.comp and blurshadow_horizontal_comp

Moment Shadow Mapping

The program uses the moment shadow mapping(MSM) algorithm. The MSM algorithm is the efficient shadow mapping algorithm that stores the depth value in four channels per one texel and uses it to calculate the shadow value of the point. The depth values are stored with depth, depth squared, depth cubed and depth to the power of four. Set the fragment depth to z_f. The values need an adjustment after filtering. I used a

$$b' = (1 - bias) \cdot b + bias \cdot C$$

$$(where\ b'\ is\ adjusted\ depth\ and\ b\ is\ depth)$$

The C is ((near depth + far depth) / 2). And I need to compute the vector c which satisfies

$$B \cdot c = Z$$

where B is 3x3 matrix with (1, b'.x, b'.y, b'.x, b'.y, b'.x, b'.y, b'z, b'w) and Z is a vector with (1, z_f, z_f^2). I used the Cholesky decomposition method to get the c. The Cholesky decomposition is more efficient than other methods when B is a symmetric matrix. I decompose B into $L \cdot D \cdot L^{-1}$ where L is the left bottom triangle matrix.

With C, solve

$$c.z \cdot z^2 + c.y \cdot z + c.x = 0$$

with a quadratic formula to get the z2 and z3 where z2 < z3.

When z_f <= z2, there will be full shadows on it.

When z_f < depth <= z3, the shadow value is

$$G = \frac{z_f \cdot z_3 - b'.x \cdot (z_f + z_3) + b'.y}{(z_3 - z_2) \cdot (z_f - z_3)}$$

For the rest cases, the shadow value is

$$G = 1 - \frac{z_2 \cdot z_3 - b'.x \cdot (z_2 + z_3) + b'.y}{(z_f - z_2) \cdot (z_f - z_3)}$$

The program will multiply (1 - G) to diffuse and specular, not ambient.

This is implemented on shadow.glsl

Debugging

I made a printing shadow map, turning on/off functions for optimization. I measured time by filtering a shadow map for finding the best algorithm in terms of performance.