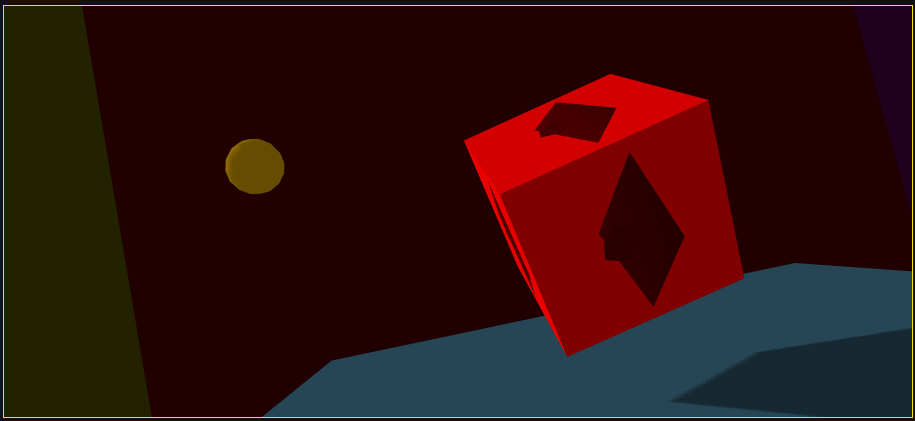
# Graphics programming - home exam 1

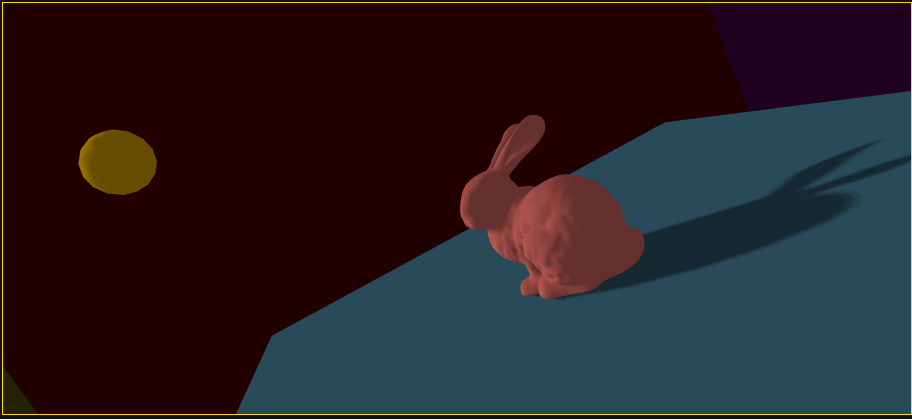
## Description of solution

This report describes how I completed the first PG6200 home exam.

The end result is what I believe to be a functioning implementation of a shadow map algorithm.



A cube rendering the depth texture as its own texture



The stanford bunny casting a shadow

### Shaders

After moving to using depth textures, most of the shaders became pretty simple.

#version 300 es  
  
precision highp float;  
  
out float fragDepth;  
  
void main (void) {  
 fragDepth = gl\_FragCoord.r;  
}

*With depth textures, just pass the r-value along*

The camera/model vertex shader uses the shadow bias matrix to convert coordinates from coordinates in the space of [-1, 1] to coordinates in [0, 1]:

#version 300 es  
in vec4 aVertexPosition;  
  
in vec3 aVertexNormal;  
in vec4 aVertexColor;  
  
uniform mat4 uNormalMatrix;  
  
uniform mat4 uProjectionMatrix;  
uniform mat4 uModelViewMatrix;  
uniform mat4 uLightModelViewMatrix;  
uniform mat4 uLightProjectionMatrix;  
  
uniform vec4 uColorMult;  
uniform vec3 uLightDirection;  
  
out highp vec3 vLighting;  
out highp vec4 vAmbientColor;  
out highp vec4 vColor;  
  
const mat4 shadowBiasMatrix = mat4(  
 0.5, 0.0, 0.0, 0.0,  
 0.0, 0.5, 0.0, 0.0,  
 0.0, 0.0, 0.5, 0.0,  
 0.5, 0.5, 0.5, 1.0  
);  
  
out vec4 vShadowPosition;  
  
void main(void) {  
 gl\_Position = uProjectionMatrix \* uModelViewMatrix \* aVertexPosition;  
  
 vShadowPosition = shadowBiasMatrix \* uLightProjectionMatrix \* uLightModelViewMatrix \* aVertexPosition;  
  
 vColor = aVertexColor \* uColorMult;  
  
 highp vec3 ambientLight = vec3(0.6, 0.6, 0.6);  
 vAmbientColor = vec4(ambientLight.rgb \* vColor.rgb, vColor.a);  
  
 highp vec3 directionalLightColor = vec3(1, 1, 1);  
 highp vec3 directionalVector = normalize(uLightDirection);  
  
 highp vec4 transformedNormal = normalize(uNormalMatrix \* vec4(aVertexNormal, 1.0));  
  
 highp float directional = max(dot(transformedNormal.xyz, directionalVector), 0.0);  
 vLighting = directionalLightColor \* directional;  
}

… and then calculates the shadow based on the light value of its nearest neighbors. To avoid areas in shadows to become completely blackened, we add the ambient light values to the visibility values:

#version 300 es  
precision highp float;  
  
in vec4 vShadowPosition;  
in vec4 vColor;  
in vec3 vLighting;  
in vec4 vAmbientColor;  
  
uniform sampler2D uDepthTexture;  
  
out vec4 fragColor;  
  
void main(void) {  
 vec3 shadowCoord = vShadowPosition.xyz;  
 float acneThreshold = 0.0015;  
 shadowCoord.z -= acneThreshold;  
  
 float visibility = 0.0;  
  
 float texelSize = 1.0 / 1024.0;  
  
 for (int x = -1; x <= 1; x++) {  
 for (int y = -1; y <= 1; y++) {  
 if (shadowCoord.z < texture(uDepthTexture, shadowCoord.xy + vec2(x, y) \* texelSize).r) {  
 visibility += 1.0;  
 }  
 }  
 }  
  
 visibility /= 9.0;  
 fragColor = vAmbientColor + visibility \* vec4(vColor.rgb \* vLighting, vColor.a);  
}

### GL general

The render loop is split into three stages:

1. Map the model data (positions, faces, etc) for the shadow casting objects into model and normal matrices
2. Do the first render pass for depths, using the light's projection matrix. Render to the allocated frame buffer object.
3. Do a second render (map and include the remaining models that don't affect shadows) using the depth texture's values to calculate what parts of the scene should be in shadow and not.

The light projection matrix uses a pretty simple lookAt function to always face in towards the center of the world and can easily be controlled through the UI



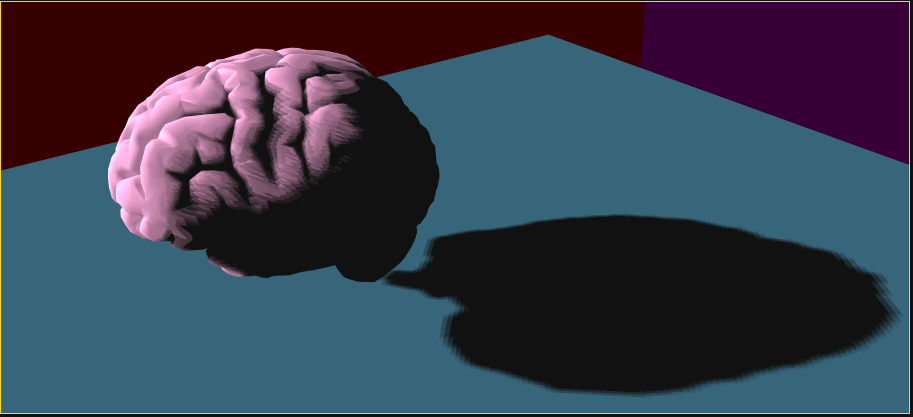
The main UI controls, including model selection, light sliders, and individual model controls (different models may have different shader programs available, for instance)

### Known limitations and where to go next

Given more time, I would have liked to look more into shading and shadow mapping. In particular, it would have been interesting to explore other options to 'percentage closer filtering' or how that could be improved. I would also have liked to look more into diffuse and specular shading and how to implement these. Furthermore, some more time looking into lighting scenes and models and how the shaders interact would likely yield fruitful results.

### Other comments

I personally really like the effect of the shadow mapping when the parts that are in shadow are completely obscured, but I realize that it's not always appropriate. However, because it looks really good and it was part of the process, I think it's worth including this image from before the ambient light was added back in to the model:



A brain with shadows and no ambient light.

## Technical challenges met during implementation

The main challenge met during the implementation–which is likely to pop up other places in the report–was my lack of familiarity with WebGL as a whole. While I have made an effort to stay up to date and follow along with all the coursework, I recognize that I may not have succeeded completely, and as such, I must take at least part of the blame for this shortcoming.

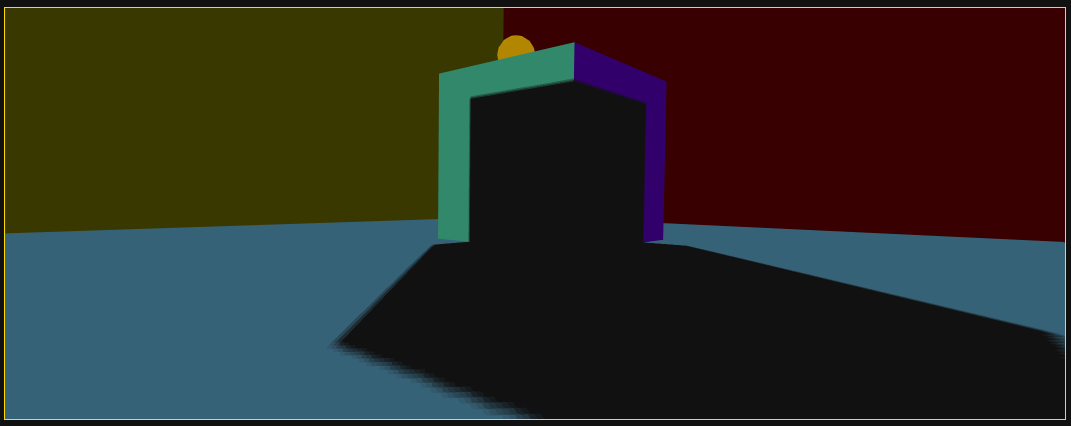
The assignment states to use the data type gl.UNSIGNED\_INT\_24\_8 for the texture. However, despite spending a lot of time looking for resources I was unable to find any that showed how to use this. The data type was mentioned several times, including in [MDN's WebGL docs](https://developer.mozilla.org/en-US/docs/Web/API/WEBGL_depth_texture) and on [WebGLFundamentals](https://webgl2fundamentals.org/webgl/lessons/webgl-data-textures.html), but neither those nor any other resources I came across have any examples on how to use it.

Once I had managed to render to the texture and could confirm that it looked correct by rendering to the canvas instead of to the texture, the next thing was displaying the shadows correctly. In this I faced several challenges, of which the one that took me the longest to solve was how to make the shadows not be affected by the position and rotation of the camera. In the end I realized I had to multiply the lightModelViewMatrix by the camera's rotation matrix to make the shadows appear in the right spot.

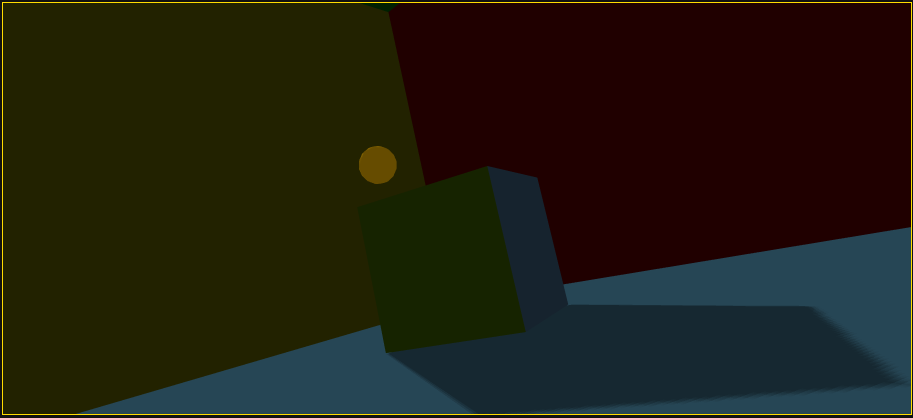
Dimensions of the target texture. This is an area where I struggled to find any good information, but from what I could find, it seems the standard is to make it square. The larger you make it, the more information it can hold and the smoother your shadows will look due to the increased resolution that comes with the larger size. Naturally, this is a point where you trade performance for visual quality, but for keeping downtime to a minimum, I went with 1024x1024, which gives me fairly smooth shadows and still acceptable performance on a mid-range laptop.

There was also a fair amount of confusion relating to the orthographic projection matrix to be used for the light projection. In particular what values would be appropriate for the corners, and the near and far values. In the end it became a case of trial and error: finding something that would be able to capture the whole scene without getting too blurry because it's too far out. The ideal way to solve this would probably be to calculate how much has to be shown on a case by case basis and then rendering just the required amount, so that when the camera approaches an object and gets closer, the level of detail on the shadows would increase.

I had an issue with the shadow acne remover used in my fragment shader. At first it was much too coarse which caused the shadows to be severely reduced creating strange artefacts like the thick outline at the edges of the cube being lit when they were clearly in shadow. By making the acne remover much smaller I was able to greatly reduce the effect, but the issue still persists at a much smaller scale. In the end, adding the ambient light seemed to take care of the issue, as can be seen in the after photo.



Before: Acne issues. Note the thick outline along the edge of the cube, which should be in shadow.



After: Cleared of issues

## Known bugs and probable causes

At the time of writing, I am not aware of any bugs. While not

## Testing methods utilized

While I am very much an avid proponent of testing, there was no formal or rigorous testing used as part of this project. This is due in part to the fact that testing graphics is inherently more difficult as it can often be hard to test visuals, and due in part to the fact that WebGL is a very stateful system, where setting up test cases and assertions can be very difficult.

That said, one point where tests could have been very useful is in matrix multiplication and model transformations. If you knew what you would want to end up with, you could have set up a test case and written functions and refined them until they gave the desired output. However, this would require a very good understanding of exactly where you wanted to go and it sounds more like a tool for making sure your calculations and transformations are correct. That said, broken up into small enough functions, this could have been a good way to assert that transformations are correct and to detect potential regressions.

Furthermore, as my knowledge of WebGL is still very basic, the entire exercise has been in the realm some may describe 'exploratory programming', where you don't really know what you're going to end up with and how you're going to get there, but you do have an idea of the rough outlines of things. At this level, I would argue that proper testing is still more of an overhead than a boon. Once a proper architecture is established and a solid understanding of the system is in place, however, erecting a test suite would be very beneficial.

## Sources

Throughout the assignment I have had to lean heavily on the few sources I could find. The most valuable ones were the [WebGL2 Fundamentals article for rendering to textures](https://webgl2fundamentals.org/webgl/lessons/webgl-render-to-texture.html) from whence I got the use of depth textures, the [opengl-tutorial article on shadow mapping](http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/), which informed my use of the shadow bias matrix, shadow acne removal, and front and back face culling for rendering, and [this CFN shadow mapping tutorial](http://www.chinedufn.com/webgl-shadow-mapping-tutorial/), which influenced the use of the 'percentage closer filtering' algorithm for smoothing the shadows as well as some more basic information around the mapping process itself.