1 Basic Tasks

1.1 One 3D object with at least 8 individual faces

The object chosen was a marble bench. The bench consists of three cuboids, namely the sitting surface of the bench and the two legs that support it. The `cuboid` function was created to allow easy specification of a cuboid in space. Using the arguments `left, right, bottom, top, back, front`, the bounds of the cuboid were defined. Using these bounds, a vertex could be placed at each corner of the cuboid and each face be defined through two triangles.

1.2 One texture applied to the object

A marble texture is applied to the object. To ensure this wraps properly, a standard cube wrapping strategy is used to calculate the texture coordinates for each part of the bench. Cube wrapping takes the full texture and defines six smaller areas within it, each representing one face of the cube. By having faces be adjacent on the source texture, proper wrapping is ensured. Cube wrapping can be extended to any cuboid shape by stretching mapping areas to match each face of the cuboid. Figure 1 shows an example of a die texture that can be wrapped onto a cube.

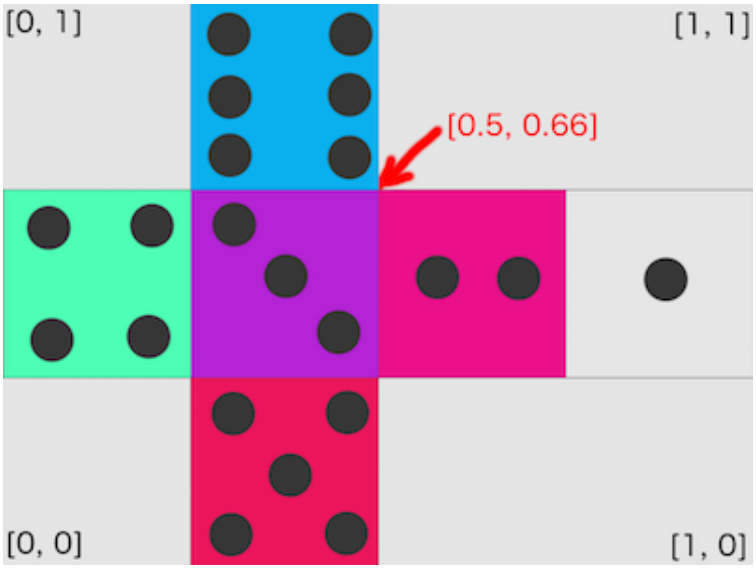


Figure 1: Wrapping of a die texture onto a cube [1]

1.3 Ambient shading

Ambient shading can be defined as every part of an object having the same lighting. This simplifies calculation of fragment colour in the shader as the texture can be applied directly.

1.4 Simple animation

To give the object a simple animation, the object is rotated around the y-axis each frame. This requires calculating a rotation matrix that can be multiplied with the

bench's model matrix to get a new model matrix. This new model matrix is then passed to the shader, updating the position of the bench with each frame

1.5 One camera, using oblique perspective

Oblique perspective can be applied through shearing matrix 1 [2], [3].

$$H(\theta, \phi) = \begin{bmatrix} 1 & 0 & -\cot\theta & 0 \\ 0 & 1 & -\cot\phi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

As glMatrix stores matrices in column-major order (Figure 2), the transpose of shearing matrix 1 is needed.

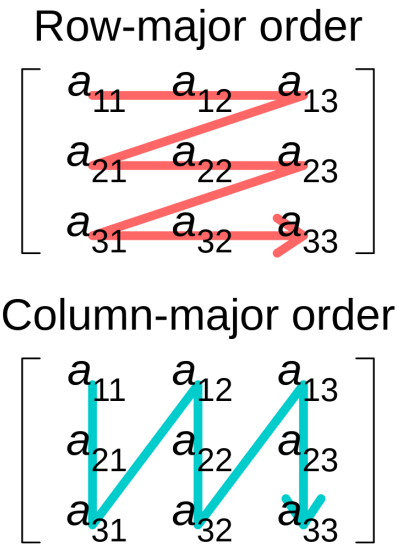


Figure 2: Two matrix orders. glMatrix uses column-major order. Image: [4]

Result

Figure 3 shows a render, including all the requirements listed in previous subsections.

Figure 3: The bench in Part 1 with oblique projection

2 Standard Tasks

2.1 Perspective projection

To use perspective projection, the camera had to be placed and projection applied. glMatrix functions `lookAt` and `perspective` help with this. Placing the camera far enough away and using a reasonable field of view ensured a good view of the scene.

2.2 Three or more objects using more than one texture

A lamppost and flower were created to form a set of three objects. The lamppost consists of a cuboid base, pole and lamp.

The flower consists of a stem, one leaf on each side and two petals above the stem. The stem and one of the two petals is created using the `cuboid` method, as before. The two leaves and second petal are created using the custom `line` method. This takes in a start point (`op1`) and an end point (`op2`). These points are used to determine the width of the object. The additional `height` and `depth` arguments define the remaining dimensions. Together, this information is used to draw a cuboid with a custom width, height and depth, directly from one point to another. By taking the cross product of the difference between the points with the y and z-axis, the vertices of the cuboid can be determined. Finally, taking the cross product of the other two cross products completes the full set of normals to use for shading. One remaining issue with this approach is that if the object is aligned with the y or z-axis, then the cross product will be zero.

To apply a different texture, each object is given a texture slot. When rendering an object, its texture is activated and bound to show the correct texture. All the textures used were provided in lecture materials. Alternative textures for the lamppost and flower would be more fitting, such as steel for the lamppost and greenery for the flower. Additionally, the code lacks the ability to give different parts of a model a different texture. This hinders giving the flower petals a different texture to the stem and leaves.

2.3 Flat shading with a coloured, direction light source

To give the objects flat shading, each object must be given normals that are passed to the vertex shader. The normal matrix is multiplied by the normal of each vertex to give a transformed normal. This must be normalised to ensure that any changes to the magnitude of normals from transformations are negated.

The transformed normals can be used with a directional light source to determine a lighting value for each vertex. The directional light is represented by a direction and colour vector. The colour vector is set to a blue colour to make the light coloured. The lighting value for each vertex is determined by taking the dot product of the transformed normal with the light's direction and adding this to an ambient light value. The lighting value for each vertex is passed to the fragment shader, where it is multiplied with the base colour of the texture to complete shading.

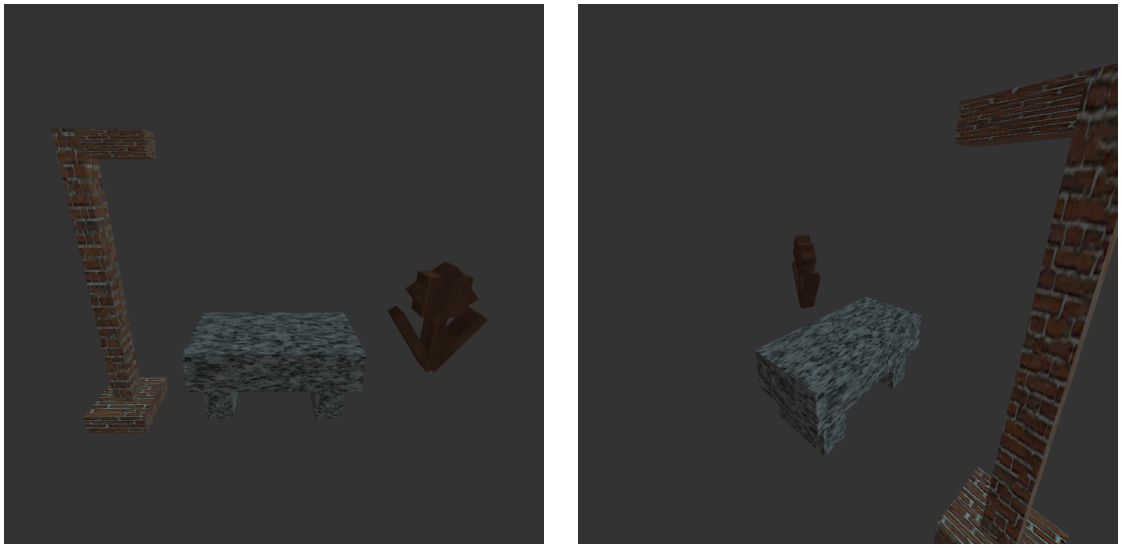
2.4 Animation of objects around an arbitrary point at different speeds

To rotate the objects around a point that is not the origin, they are translated in the negative y direction. Furthermore, a scene hierarchy is created, with the bench serving as the parent for the lamppost and flower. The `createM` and `returnM` methods provided with lecture material create and multiply model matrices of each object with any parents it may have. This creates a combined model matrix that takes into account the hierarchy.

Using a scene hierarchy lets the position of the lamppost and flower be defined in relation to the bench, simplifying the transforms needed. All objects are rotated around the y-axis, with the lamppost and flower rotating at higher speeds than the bench.

Result

Figure 4 shows a render, including all the requirements listed in previous subsections.



(a) Front View (b) View With Some Rotation

Figure 4: The objects in Part 2 with perspective projection and a coloured, directional light source

3 Advanced Task

The advanced task chosen was to implement environment / reflection mapping. This includes a skybox and surfaces in the scene that reflect the skybox texture. As guides, two online articles by WebGL Fundamentals [5], [6] and one YouTube video, *Environment Mapping* [7] by C. Yuksel, were used.

3.1 Skybox

The skybox is created using cube mapping. This maps six images onto the face of a cube. In the context of the skybox, the cube surrounds the entire scene and has one face per axis. The same cube wrapping principle explained in Section 1.2 is used. However, each face is stored as an individual file and then combined in code. Figure 5 shows the unwrapped skybox texture.

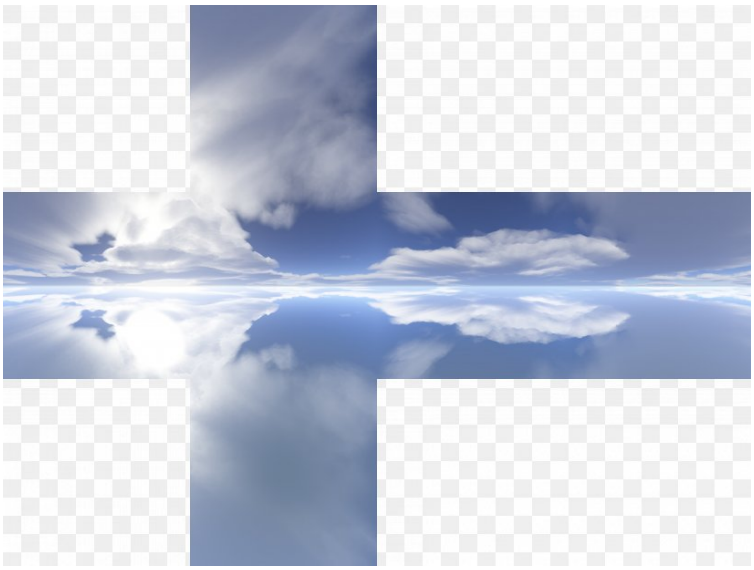


Figure 5: The unwrapped skybox

The skybox uses different shader code to the objects. The skybox's vertex shader takes the original position of a vertex and sets both the z and w values to 1. This ensures that the skybox is always rendered behind objects. If the skybox were instead rendered as a cube in normal z space, then orienting it to stay within the clip space could result in objects being obscured that should be visible (Figure 6).

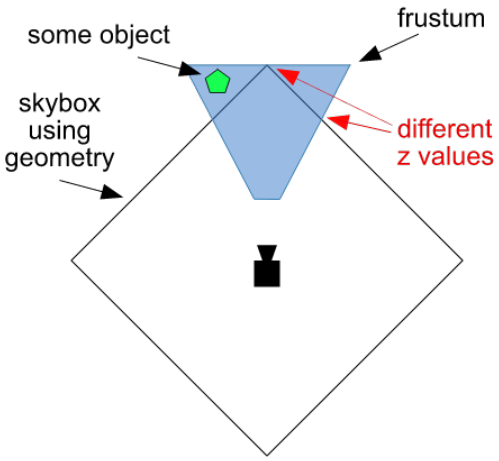


Figure 6: A skybox in normal z space could obscure objects that should be in view of the camera [5]

To render the skybox properly, the camera's viewing direction must be taken into account. A series of matrix multiplications can give a matrix that allows this. First, a view direction matrix is initialised from the view matrix. The translation components of this view direction matrix are set to 0 as the skybox is assumed to be very far away and unaffected by camera movement. Second, the projection matrix is multiplied by the view direction matrix to take projection into account. Finally, the inverse of the view direction projection matrix is calculated to get the direction of the camera to the skybox. This inverse matrix can then be passed to the fragment shader. In the fragment shader, the matrix is used to get the texture coordinate to sample from the skybox cubemap.

After rendering each object, the depth test is adjusted to render the skybox at the edge of the clip space. The shader program is switched to the skybox shader program and buffers bound before rendering the skybox.

3.2 Environment / Reflection Mapping

Making the objects in the scene reflect the skybox requires loading the skybox texture onto them and then blending this with the original texture of each object. The skybox texture is created using cube mapping as before. However, the texture must be bound to a different texture unit than that of each object's main texture. To satisfy this requirement, the main texture is bound to texture unit 0 and the skybox texture to texture unit 1.

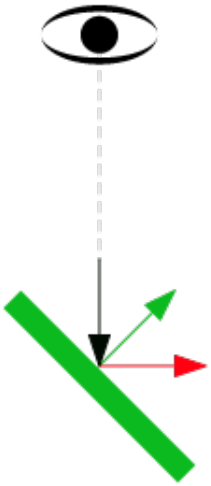
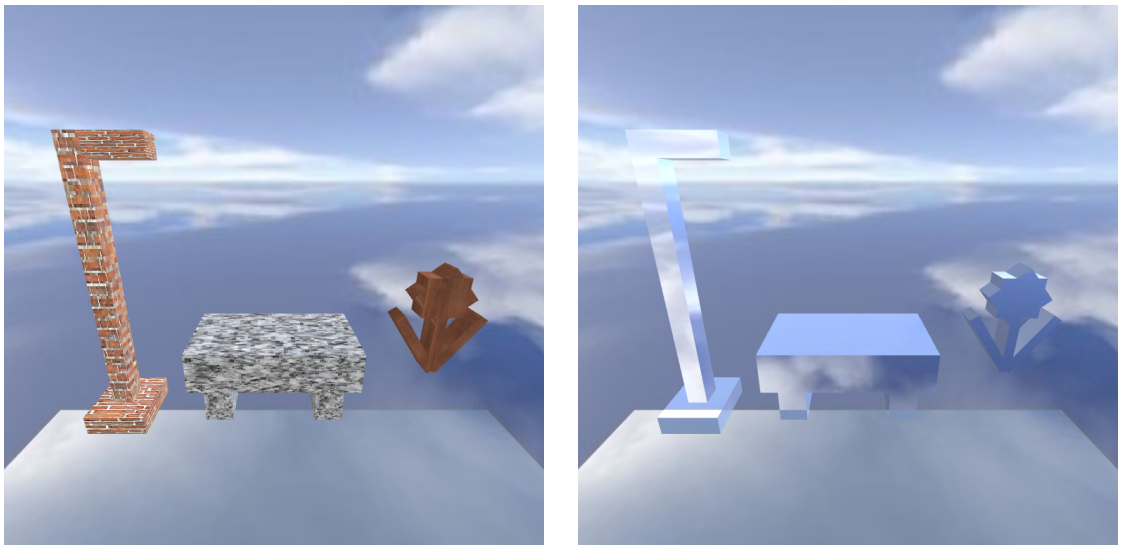


Figure 7: The reflection direction (red) can be calculated from the camera to surface direction (black) and oriented normal (green) [6]

To render the correct part of the skybox texture on each object, the direction from the camera and object normals must be used to calculate the reflection direction towards the skybox (Figure 7). First, the view position and oriented normals are calculated and passed to the fragment shader. Passing in the camera's position in world space and subtracting this from the view position gives a vector from the camera to the surface. Normalising this vector from the camera to the surface and using it in GLSL's reflect function together with the oriented normals gives the reflection direction. The reflection direction can then be used to sample the skybox texture. Finally, the base and skybox texture samples can be blended together with GLSL's mix function, giving a final fragment colour. Figure 8 shows what the objects would look like without proper blending.

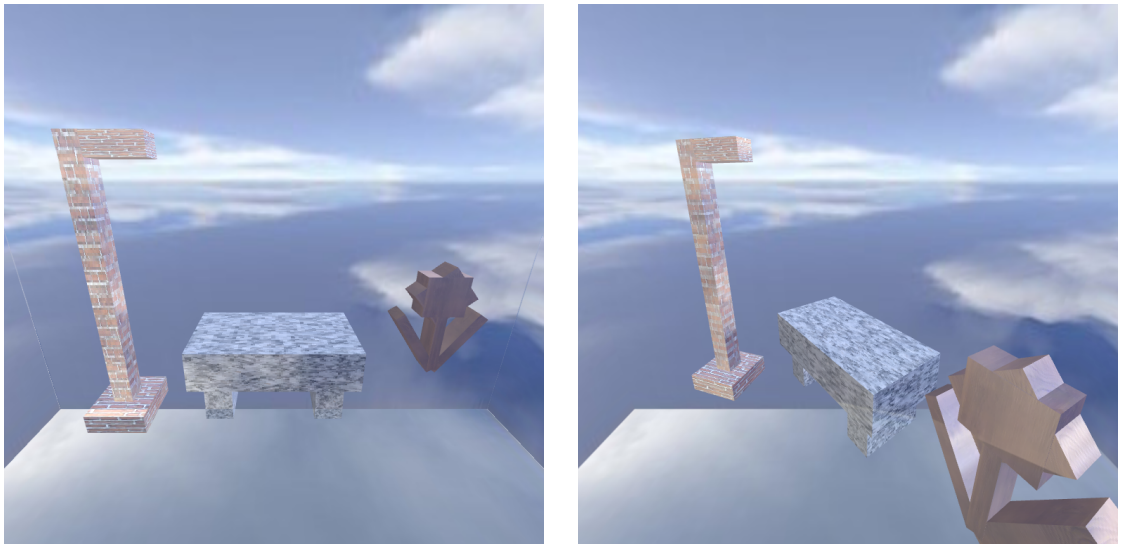


(a) Base Texture (b) Skybox Texture

Figure 8: The objects in Part 3 with improper blending

Result

Figure 9 shows a render, including all the requirements listed in previous subsections.



(a) Front View (b) View With Some Rotation

Figure 9: The objects in Part 3 with environment mapping and a skybox

4 Champion Level Task

The champion level task chosen was to implement a mirror surface showing other objects (two-pass rendering). A mirror is placed below the objects from the previous scene. The mirror shows a reflection of the objects through use of a framebuffer. This framebuffer holds a separate render of the objects from an inverted camera angle, which can be shown on the mirror surface as a render texture. As guides, an online article by WebGL Fundamentals [8] and one YouTube video, *Reflections* [9] by C. Yuksel, were used.

4.1 First Pass Render

First, the mirrored view is rendered to a framebuffer. The framebuffer is initialised and attached to an empty target texture to render to. Furthermore, a depth buffer is initialised to take depth into account when creating the texture. With the framebuffer initialised, it can be bound to the GPU, the scene cleared and the viewport set to match the texture dimensions. Now the scene is rendered with a view matrix that has its y coordinate inverted to emulate the mirrored view. The mirror is not rendered in the first pass. This fills the render texture with a mirrored image of the scene (Figure 10).

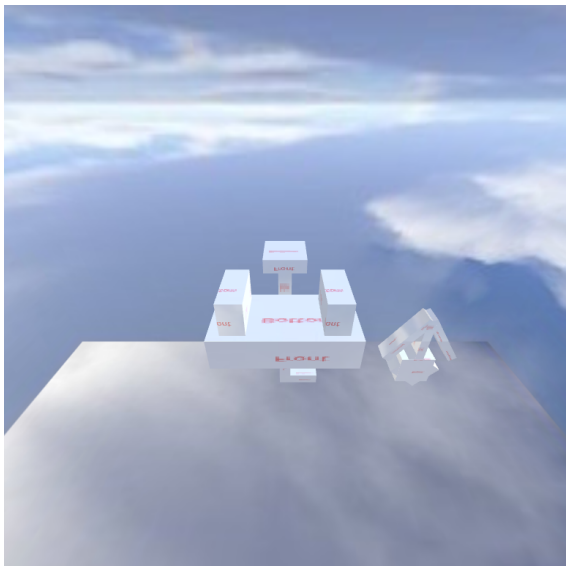
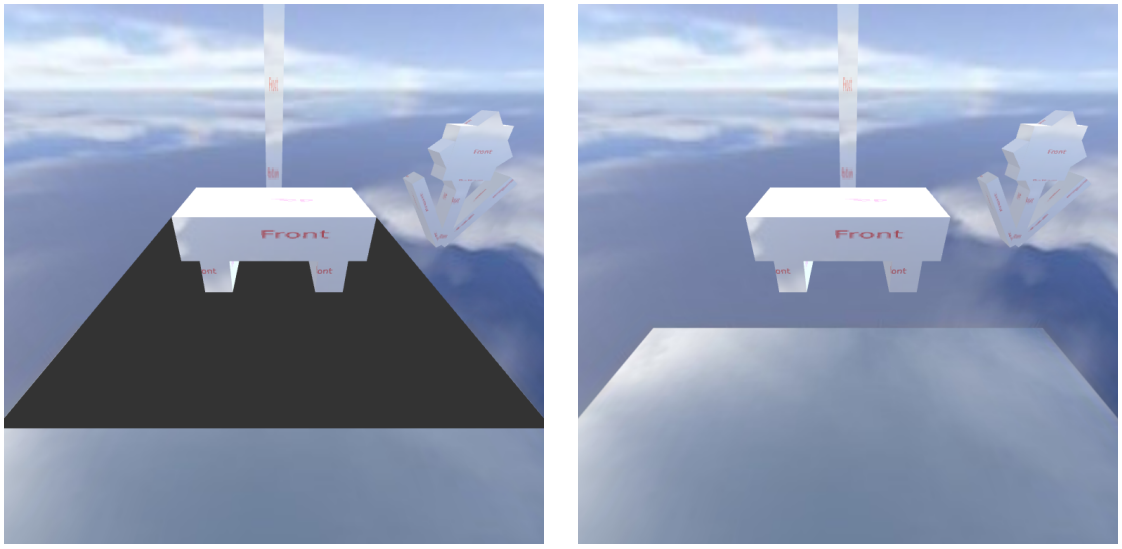


Figure 10: The render produced from the first pass views the scene from the bottom. Textures have been replaced with a special diagnostics texture that marks the faces of each object.

4.2 Second Pass Render

For the second pass, the framebuffer is unbound to the GPU, the scene cleared and the viewport reset to the canvas. The scene is then rendered as in previous parts with the standard view matrix. Now, the mirror can be rendered with the render texture from the first pass. The span of the mirror is shown in Figure 11(a), with the remaining issue of texture mapping discussed in Section 4.3.



(a) View with blank mirror (b) View without mirror

Figure 11: The second pass views the scene from the top, with a mirror surface to map the first pass onto

4.3 Texture Coordinate Mapping

The final remaining task is to map the texture from the first pass so that it fits into the scene of the second pass. Using the full texture from the first pass results in a poor reflection as the texture contains more than should be reflected (Figure 12).

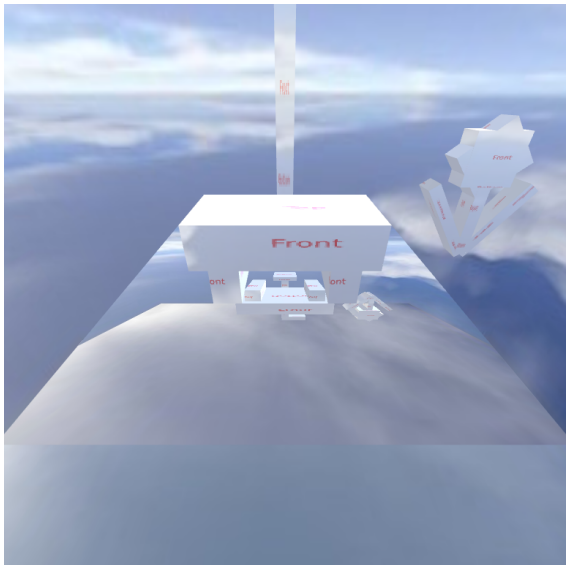
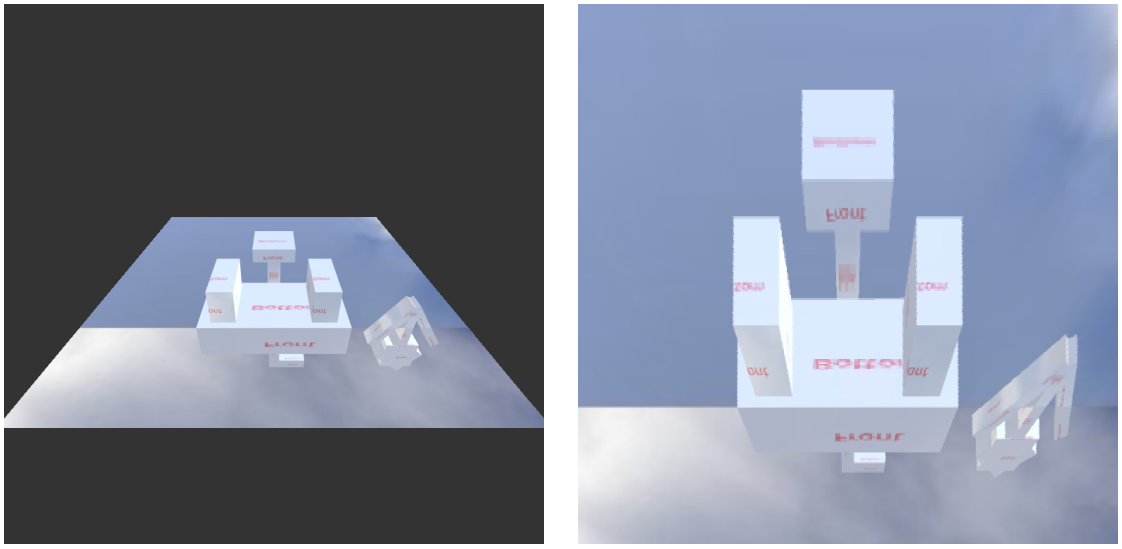


Figure 12: Using the full render texture gives poor reflections

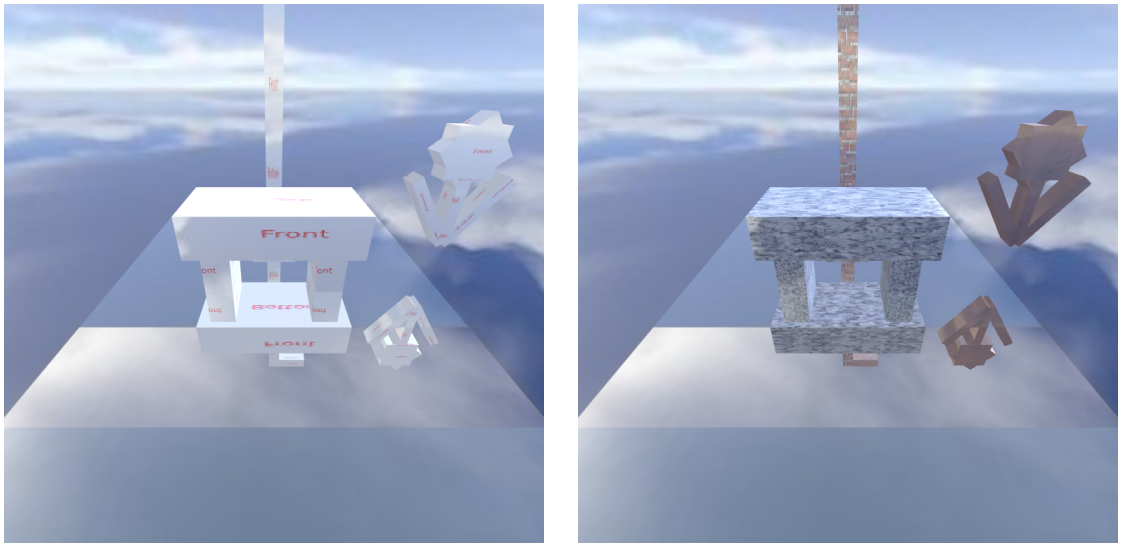
Instead, a cutout from the first pass should be created (Figure 13(a)) that can be stretched into an accurate texture for the mirror (Figure 13(b)). The cutout can be created by converting the vertices of the mirror from model coordinates into clip space coordinates after applying perspective with normalisation. The clip space coordinates can then be converted into texture coordinates by adding one and dividing by two.



(a) Cutout of the desired area from the first pass (b) The texture obtained from the cutout when mapped back onto a square area

Figure 13: Using the texture coordinates of the mirror gives a texture from the first pass that can be mapped onto the mirror in the second pass

When applied to the mirror, the stretched reflection texture should provide accurate reflections after perspective projection (Figure 14).

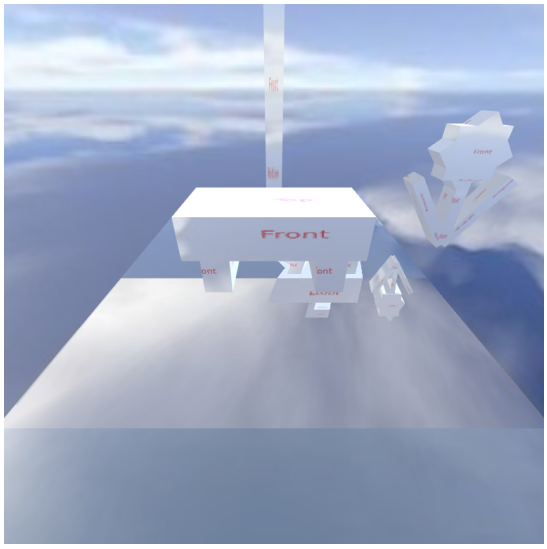


(a) Diagnostics Texture (b) Normal Textures

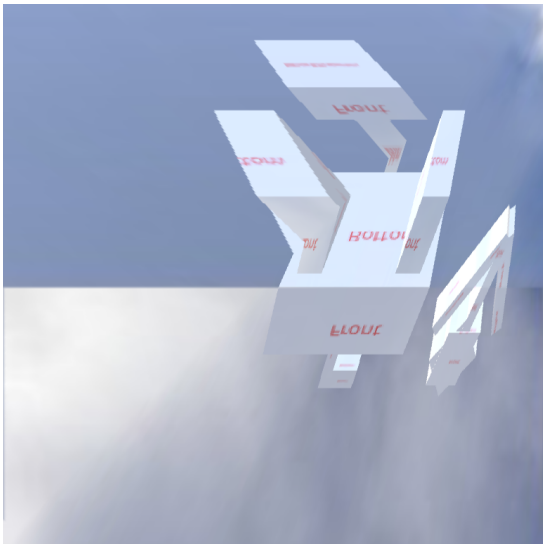
Figure 14: The expected result when mapping the first pass render onto the mirror in the second pass

Result

Unfortunately, the texture coordinate mapping described in 4.3 did not seem to work in WebGL. This is likely due to skewing of a texture via texture coordinates not working as expected by intuition. Figure 15 shows that WebGL skews the texture to the right along a triangular boundary rather than equally in both directions.



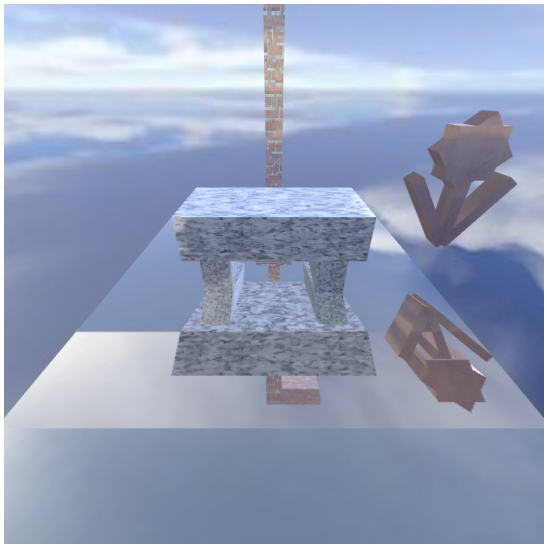
(a) The scene viewed with the poorly skewed mirror texture



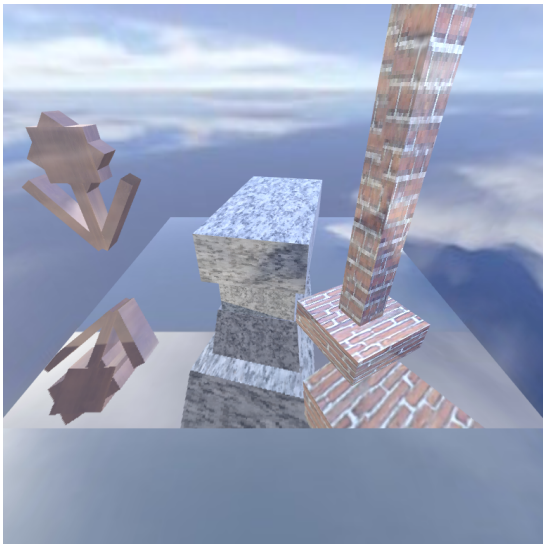
(b) The poorly skewed mirror texture

Figure 15: The result when mapping via the expected texture coordinates. A triangle pattern is visible across the bottom left to top right.

Many other texture coordinate mappings were tried out, but none seemed to produce a fully accurate reflection. Figure 16 shows one set of texture coordinates that matches closely with the configuration of camera and mirror position, but still shows imperfections.



(a) Front View



(b) View With Some Rotation

Figure 16: The objects in Part 4 with a mirror showing their reflection

References

[1] I. Balkanay, *Uv mapping (of a cube)*. [Online]. Available: https://ilkinulas.github.io/development/unity/2016/05/06/uv-mapping.html (visited on 02/04/2024).

[2] E. Angel, *Oblique projections*. [Online]. Available: https://www.cs.unm.edu/~angel/AW/chap05d.pdf (visited on 26/03/2024).

[3] LJ, *Webgl - oblique projection*. [Online]. Available: https://stackoverflow.com/questions/29131729/webgl-oblique-projection (visited on 26/03/2024).

[4] Cmglee, *Row and column major order*. [Online]. Available: https://commons.wikimedia.org/wiki/File:Row_and_column_major_order.svg (visited on 29/03/2024).

[5] WebGL Fundamentals, *WebGL SkyBox*. [Online]. Available: https://webglfundamentals.org/webgl/lessons/webgl-skybox.html (visited on 25/03/2024).

[6] WebGL Fundamentals, *WebGL Environment Maps (reflections)*. [Online]. Available: https://webglfundamentals.org/webgl/lessons/webgl-environment-maps.html (visited on 25/03/2024).

[7] C. Yuksel, *Environment Mapping*. [Online]. Available: https://youtu.be/PeAvKApuAuA (visited on 26/03/2024).

[8] WebGL Fundamentals, *WebGL Rendering to a Texture*. [Online]. Available: https://webglfundamentals.org/webgl/lessons/webgl-render-to-texture.html (visited on 03/04/2024).

[9] C. Yuksel, *Reflections*. [Online]. Available: https://youtu.be/h2RTBs1xl6w (visited on 26/03/2024).