Edinburgh Napier UNIVERSITY

Coursework Report

Marco Moroni
402138730@napier.ac.uk
Edinburgh Napier University - Computer Graphics (SET08116)

Abstract

This computer graphics project aims to show a scene that replicates the art style used in the game *Monument Valley* by using OpenGL. The main characteristic of the game – and this project – is the clever use of the clever use of a camera with an orthographical projection that allows to create impossible geometries.

Keywords – OpenGL, Computer Graphics, GLSL, Monument Valley, Games Development

1 Introduction

Monument Valley (figure 1) is a game where the player guides a character through architectures made by impossible geometries. The goal of this scene (figure 2) is to recreate this same illusion with a very similar art style. The illusions are made possible through the use of an orthographical projection that eliminates the sense of perspective from the user (or the player).
The core techniques to archive such effects are explained in one development videos made by the creators of the game [1], and are here in part reproduced with OpenGL.

Last thing to note is that the aim of the project is not to build a foundation for a game. This means that the visual design of the project is not limited by the practical considerations of implementing gameplay.

2 Related work

This is a coursework made for a Computer Graphics module, which provides a workbook that covers most of the techniques used here and slides explaining the mathematical principles.

3 Implementation

For simplicity, this report will use the terms left block, central block and right block to refer to the 3 distinct part of the scene (figure 2).

3.1 Camera

Probably the most important aspect of this project is the orthographical projection, which makes the impossible geometries possible (see figure 2: it has a perspective projection).
The orthographical projection is a parallel planar projection. Because of the position of the camera (at (20.0, 20.0, -20.0) pointing at the origin (0.0, 0.0, 0.0)) the projection is also isometric, which means that the 3 axis *x*, *y* and *z* have the same length.
The projection of a point $P = (p_x, p_y, p_z)^T$ onto a projection plane is

$$\tilde{P} = (p_x, p_y, 0)^T$$

which means that the equivalent transformation matrix A is the following:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

therefore

$$\tilde{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

From the matrix is evident that the perspective has no part in the calculation.
The code for the projection `P` is the following:

```
1 P = glm::ortho(-screen_width, screen_width, -screen_height, ↩
      screen_height, near, far);
```

There is a second free camera, that can be used to navigate around the scene.
Because of the differences between the orthogonal and perspective projections, some of the control had to be tweaked. For example to move left from it original position the camera would move (20.0 + left, 20.0, 20.0 + left). It is also possible to use a zoom by changing the code for `P`:

```
1 P = glm::ortho(-screen_width / zoom, screen_width / zoom, -↩
      screen_height / zoom, screen_height / zoom, near, far);
```

3.2 Lights

In the scene a simple cube would be rendered with only 3 visible faces. It follows that the most convenient way to control the colours would be to implement 3 directional lights, each orthogonal to one to one of the faces. The left block contains a spot light, used to show a water effect on the surface of the semi-transparent box.

3.3 Phong shading and materials

the blocks on the scene are rendered through a Phong shader without taking in consideration the reflection com-

Figure 1: **Monument Valley** - A screenshot from the game

ponent. This is most important in order to create the correct illusion because the colours of a surface need to be completely uniform.

3.4 Textures

In order to apply a different texture to each mesh there is a `map<string, string> texture_link`. The first string is the name of a mesh, while te second is the name of the corresponding texture. Note that meshes and textures are stored in a `map<string, mesh>` and `map<string, texture>` respectively.
Almost every mesh in the scene is made by cubes or planes. This is important for the art style used, because the texture are applied in a tile-based manner. The boxes are generated with a size of `1 x 1 x 1`, while quads are generated with a size of `2 x 2` and then scaled by 0.5. This is necessary in order to render the textures correctly on every surface.
The left and the central blocks are made by brown tx-

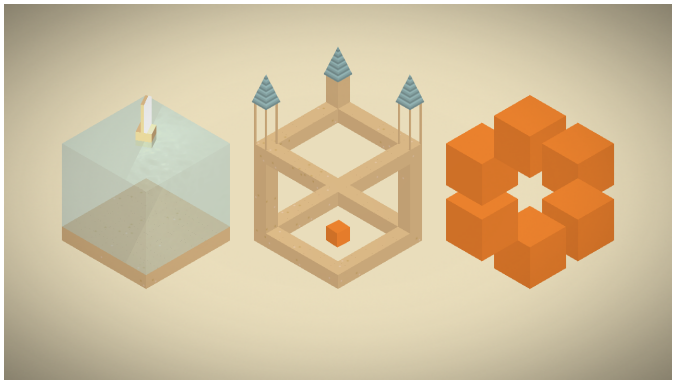

Figure 2: **The entire scene**

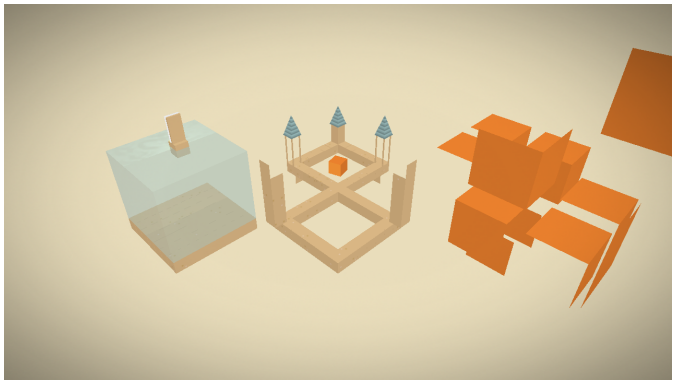

Figure 3: **The scene with a perspective projection**

tures and some of them have bricks in it (figure 4). The

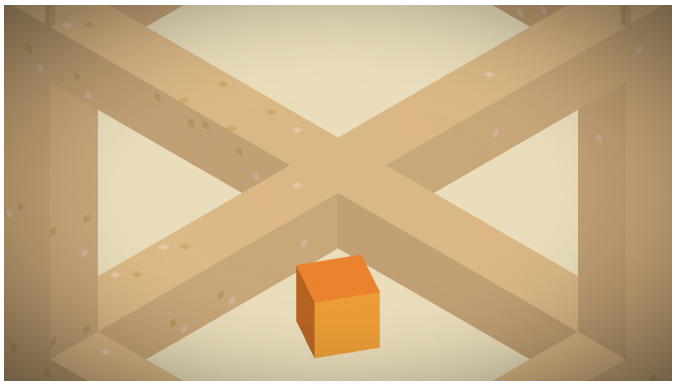

Figure 4: **The textures in detail**

textures used are 4: one plain brown texture and three different textures with bricks. To randomly use one of them a function is called every time a texture is linked to a mesh through the `texture_link`. The pseudo-code of this function is the following: It's possible to make the bricks appear less often by increasing the number which in the first line is now `10`.
The skybox uses the same texture for all its six sides.

```
wall_tex_number = random_number % 10;
if wall_tex_number == 1 then
    return "wall_brick_1";
end
else if wall_tex_number == 2 then
    return "wall_brick_2";
end
else if wall_tex_number == 3 then
    return "wall_brick_3";
end
else
    return "wall";
end
```

3.5 Normal Maps

The three blocks are made with 2 maps of meshes: `meshes` and `water_meshes`. `meshes` does not have normal maps. `water_meshes` does render a normal map in order to create the waves of the water (in order to make it more visually pleasing it is only affected by the spot light). This normal map is made by the multiplication two identical normal maps that move in different directions (figure 5).

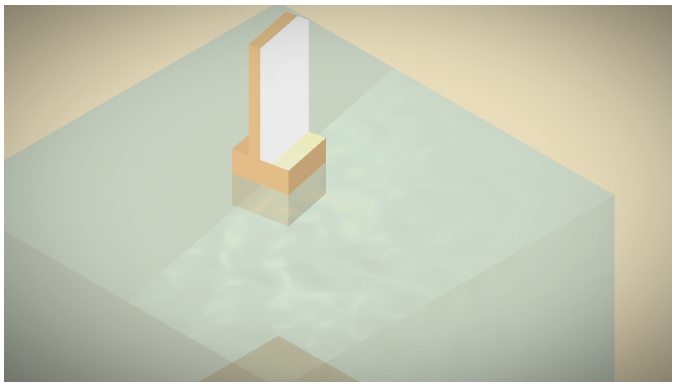

Figure 5: **Waves**

3.6 Movement of meshes

There are two moving elements in the scene: the *floating light* in the left block – whose meshes are organised in an hierarchy – and the orange moving box in the central block. the box is moving in front of the surrounding structure when moving up and behind when moving down. This is another trick that can be archived by using the orthographical projection: every time the box reaches the highest or the lowest point of its path it moves backwards or forward from the point of view of the user. She won't notice it because she can't perceive how far an object is from her.

3.7 Post-processing

There are two post-processing effects: a mask (that gives the scene a vignette effect) (figure 6 show the scene without it) and a simple shader for inverting colours (figure 7).

This second shader can be toggled and can be rendered simultaneously with the first.

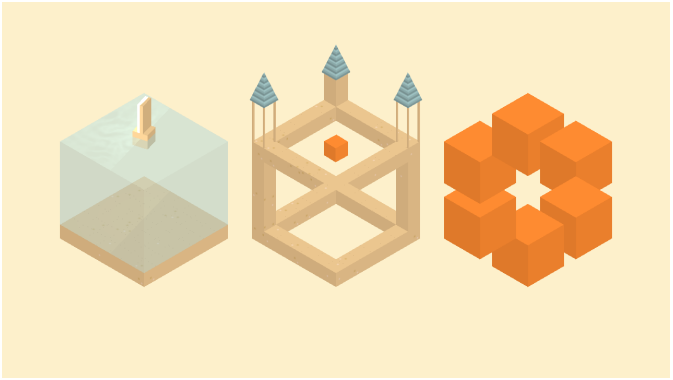

Figure 6: **The scene without the vignette effect**

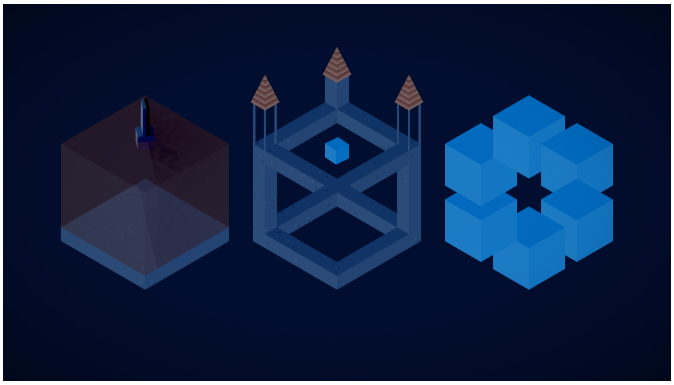

Figure 7: **The scene with the colours inverted (and the vignette effect)**

In order to render both frames at the same time the program uses a temporary frame. The render loop can be summarised as follows:

1. Set render target to `frame`
2. Render all the meshes and the skybox
3. Set render target to `temp_frame`
4. Render inverse screen colour by using `frame` as the texture to modify (only if the toggle is on)
5. Set render target to the screen
6. Render vignette effect by using
    - `temp_frame` if the inverse colour toggle is on *or*
    - `frame` if the toggle is off.

3.8 Optimization

Two methods to optimize the code have been attempted:

- In the render loop the projection `P` and the camera view `V` are calculated as few times as possible;
- The shader to invert the colours is bound only if the toggle is on;
- A future optimization could be made by merging meshes and textures (see *Future work*).

4 Future work

Right now the meshes are made of a big number of boxes and quads, even though the user only needs to see uniform big surfaces. In the *Monument Valley* development video [1] is briefly explained how all the quads (and their relative textures) are merged before exporting a scene of the game. In such a way a level would be rendered faster.

5 Conclusion

References

[1] “ustwo at nordic game 2014: Making of monument valley in unity.”