CS324 Report

Lair de La Lune by Rowan Mather (2100495)

January 2024

1 How to Win

1. Click to begin.
2. Using the arrow keys, weave to dodge the cuboid obstacles. For near guaranteed success, I recommend using single left/right taps and otherwise letting the current carry you.
3. Once you reach the back wall, ensure you are in line with and enter the white archway.
4. Wait for the second level to load.
5. Again, using the arrow keys dodge the moving obstacles.
6. Aim for the back right, just behind the dragon.
7. You will see a red target on the floor. Move onto this.
8. The camera will re-orient itself and a sniper-like view will be shown.
9. Simply click before you are hit by a boulder.
10. The dragon will be shot and the game is won.

Figure 1: Aim for the red target.

2 Controls and Camera

The game uses an entirely first person camera based on the `PointerLockControls` module. Upon clicking to lock the pointer to the window, one may drag the camera orientation around. Movement is possible in every direction using the arrow keys or ASDW.

3 Movement

Working from the template given in the example trackball implementation lab, the movement code was modified to change the camera x-position and z-position explicitly, rather than with the module helper functions. This was in order to easily implement the rebounding from walls and natural water flow in the first level.

Each input key enables/disables a flag for its corresponding direction. This means that if the player holds down both the forwards and backwards key, no resulting movement will occur. Movement is calculated in the `inputVelocity` function, relative to the direction of the camera.

Some natural deceleration is applied throughout, but the key input movement is designed to be fast so as to make the game more challenging.

In the first level, if one does not actively move backwards, one is pulled along at a slow speed through the obstacle course. Upon hitting a side wall, the player will bounce off it for a short period, continuing to move in the same z-direction (parallel to the wall), but the reverse x-direction (perpendicular to the wall). Since the first level is otherwise stagnant, this physics increases the difficulty.

The second level is mechanically different - walls simply stop the player and the water only appears to control that of the flowing rocks.

4 Lighting and Fog

The lighting code is organised within the lightWorld function.

Set in a cave system environment, the complete application is dimly lit. A latern (point-light) follows the camera throughout. In the first level there is a little ambient light, but in the second there appears to be a skylight. The texture in the roof crevice is of the emission type, and this is reinforced by a hemisphere light and additional spot-light to illuminate the dragon clearly.

In the first level, there is a thick fog. This obscured obstacles close to the camera so the player has to dodge and weave in the moment.

5 Overlays

Game updates such as the instructions (Figure 2), loading screen and death screen are created with separate html elements and displayed/hidden from the main code. Each one has a corresponding function (e.g. `failureScreen`) which will release the pointer lock where appropriate.

Figure 2: Menu overlay.

6 Three.js objects

All scene object creation/importing is done within the `buildWorld` function.

The rocks and obstacles are native to `Three.js`. A fixed number of them are always generated and stored in the `rocks` global array.

Along the z-axis (the length of the cave in level 1), blocks are evenly distributed. There are two to each row and they are pseudo-randomly positioned along the x-axis. Each cuboid mesh is generated using the `addMesh` and stored in `rocks`.

Level 2 operates on the basis of polar co-ordinates. There is one block at each unit distance from the origin, but its initial rotation is pseudorandomly generated and then updated by a constant amount every frame (`moveScene2()`). The objects stored in the `rocks` array contains the mesh, the current rotation, and the set distance from the origin. This means its Cartesian co-ordinates can easily be calculated from the stored polar ones using the `positionRock()` function.

Otherwise, the `Three.js` components consist of the cuboid floor and edges, and in scene 2, the torus border and red target made from two ring meshes.

7 Textures

Both water textures for the river (Figure 3) and whirlpool (Figure 4) are drawn and animated bespoke, so that they appear to move. The first level uses a video texture - the floor itself is stagnant with a flat animation. Conversely, level 2 uses a stagnant whirlpool image with a rotating floor. They are both then shaded using the `PhongMaterial` present in order to give the impression of being deep within the cave.

Figure 3: One of the river texture frames.

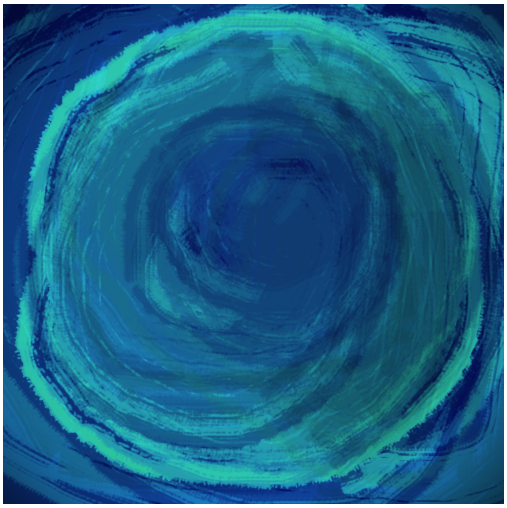

Figure 4: Whirlpool texture.

8 Collisions

Upon hitting an obstacle (or the back wall of the first cave system), the game terminates. Obstacle objects are stored in the `rocks` array and their relative distance to the user is checked every frame in order to determine collisions. This methodology is due to the limited number of obstacles - it is far more efficient to calculate intersection with meshes using formulae for distance to a sphere/cuboid than to use a ray-caster.

9 Custom Objects and Animation

Both caves systems are created and coloured in Blender using a similar process of subdividing the mesh and randomisation vertex positions to create stalagmites and stalactite (see figure 5). Some manual modification and accentuation was then performed and the latter cave was shaded smooth (see figure 6).

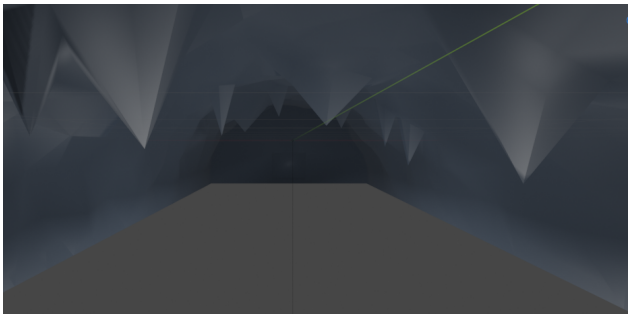

Figure 5: View inside level 1 cave.

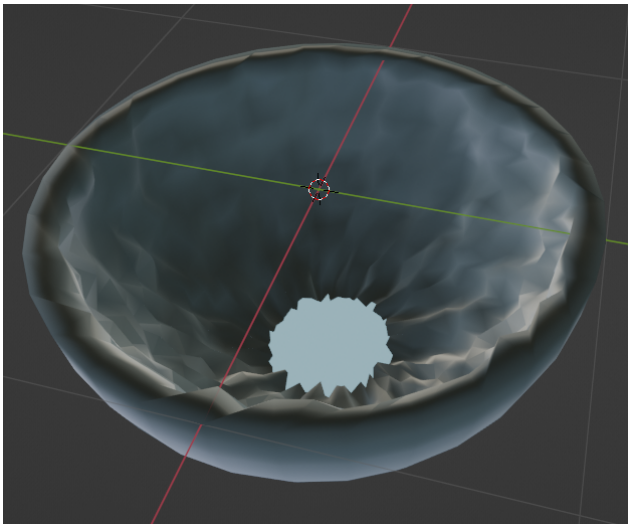

Figure 6: Level 2 cave inverted.

The water dragon was created originally from a cylinder then extruded and scaled into the main tubular body. The mirror modifier was used to ensure symmetry. Having drawn up a rough design which was imported in as a reference (and is still available in the Blender project), proportions were simple to determine.

The head detailing was predominantly crafted used a combination of insets and edge slides to create the spines. Taking the knife tool to a plane, the wings were manually cut out and certain points translated up into the third dimension. This is often effective for lamina surfaces or when tracing a reference. Finally, the solidify modifier was applied to give them thickness. Joining the wings with the main object, faces were

manually added to connect them up to the body. A similar process was performed for the barbels on the nose.

To decorate the dragon, simple block colour materials were used. This combined with the flat shading gives it a striking appearance.

The dragon is intended to survey the area, so an armature was created with a bone for each of the many neck segments, and the head (see figure 9). Three of these joints were then key-framed at different rotations to create the left and right motion. It would be trivial to add further animation to the rest of the dragon, but this was sufficient for demonstration.

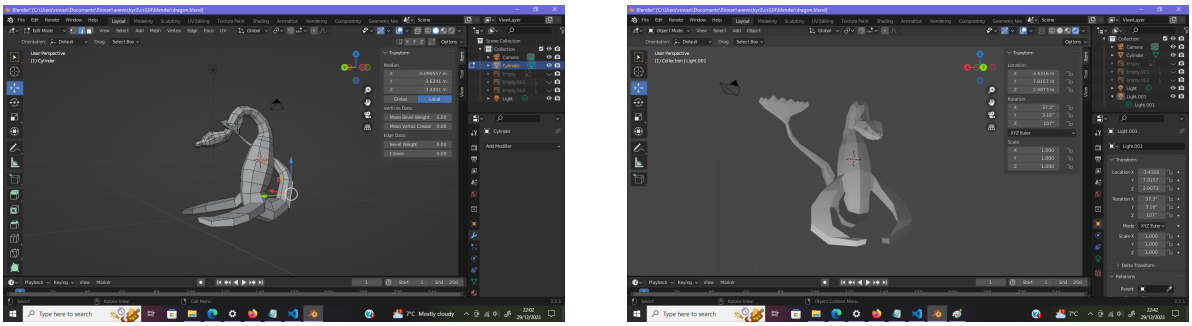

Figure 7: Water dragon in early development stages.

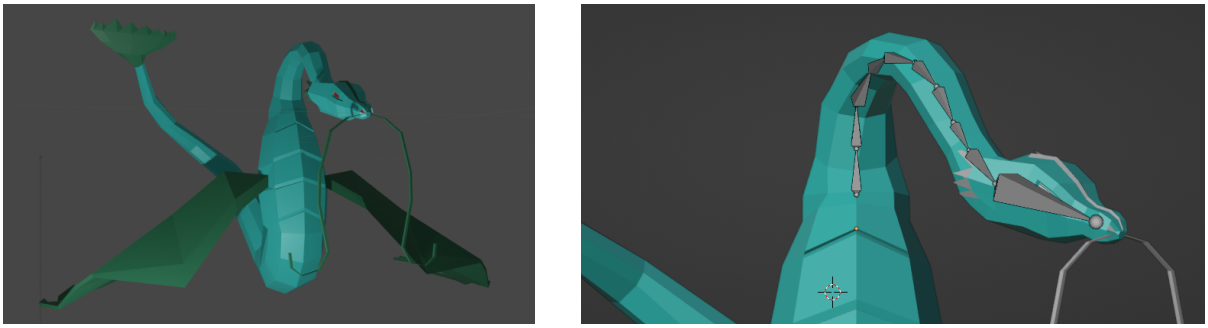

Figure 8: Final water dragon (left). Rigged amateur (right).

10 Particles

The dragon's breath is created with a number of small planes which are randomly coloured (see figure 9). The difficulty in implementation stems from matching the spray to the animation of the head. By using the intended frame-rate, the number of frames and the time delta, one can calculate the duration through the animation cycle. Applying a sinusoidal transformation to this and rotating it round, we align the angles. Each particle has a lifetime - after which it is recycled. This also alters its y-position, falling further as it dies.

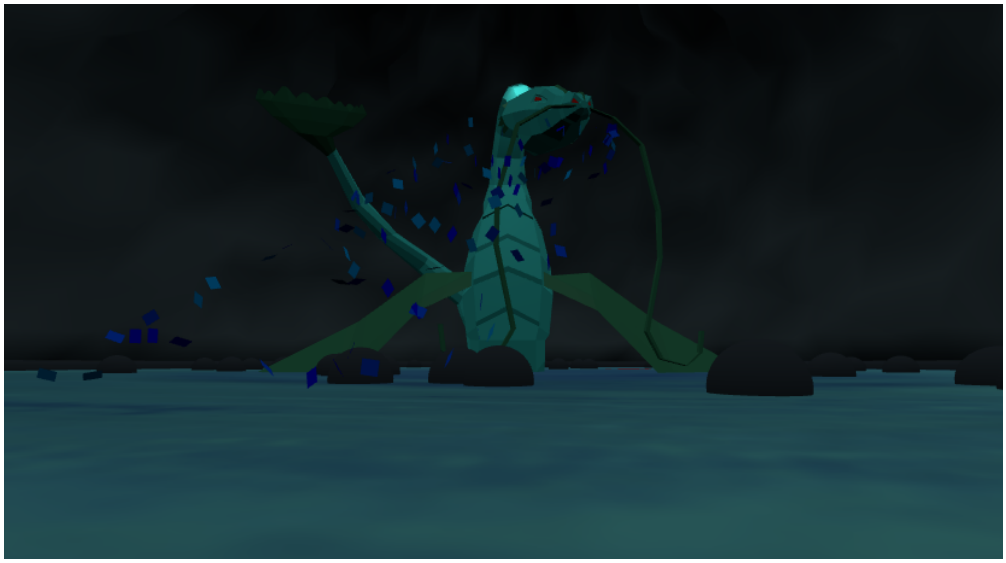

Figure 9: Particle demonstration.