

Implementation

Gameplay

Description

In this game, the player controls a cube by using the "awsd" keys to fold it through the level. The levels are made up of cubes and the goal is to fold the player cube onto a certain "target" cube of the level.

If the player made a wrong move, he/she can use the "u" key to fold the player cube back up to its starting position. Some levels can't be solved with folding out the cube just once. For those levels, the player can press the "e" key once the player cube is fully folded out to fold it up to the position it moved to last.

Levels

The single levels are stored in .xml files and look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Level>
  <Blocks>
    <Start>  <!-- Position, where the player starts out -->
      <x>0.0</x>
      <y>1.0</y>
      <z>0.0</z>
    </Start>
    <NormalBlock> <!-- Standard level block -->
      <x>0.0</x>
      <y>0.0</y>
      <z>0.0</z>
    </NormalBlock>
    <NormalBlock>
      <x>1.0</x>
      <y>-1.0</y>
      <z>0.0</z>
    </NormalBlock>
    <NormalBlock>
      <x>1.0</x>
      <y>-1.0</y>
      <z>-1.0</z>
    </NormalBlock>
    <Finish> <!-- Target cube -->
      <x>2.0</x>
```

```
<y>0.0</y>
<z>-1.0</z>
</Finish>
</Blocks>
</Level>
```

This XML defines the third level that is currently in the game. The levels are loaded by reading the xml and placing the cubes defined in it relative to a "start" offset of the level.

Movement

The movement can be broken down into two parts. Checking if the player can move in a certain direction and animating the cube correctly.

Can it move?

The PlayerCube object stores the players current position. When "a", "w", "s" or "d" are down, the according direction is added to that position to determine the target cube.

```
if (glfwGetKey(window, GLFW_KEY_W)) {
    direction = vec3(0, 0, -1);
}
else if (glfwGetKey(window, GLFW_KEY_S)) {
    direction = vec3(0, 0, 1);
}
else if (glfwGetKey(window, GLFW_KEY_A)) {
    direction = vec3(-1, 0, 0);
}
else if (glfwGetKey(window, GLFW_KEY_D)) {
    direction = vec3(1, 0, 0);
}
vec3 targetCube = player->currentPosition - vec3(0, 1, 0) + direction;
```

Then, I check if that target cube is available in the current level. If so, I additionally check, if the player cube has the necessary sides to move onto that cube.

Move it.

Before the cube is animated, the sides which have to be left behind have to be determined. Usually, this is only the side which is currently on the bottom.

```
for (it = availableSides.begin(); it != availableSides.end(); it++) {
    if (round(it->get()->cubeSide.y) == -1) { // is bottom side
        placedSides.splice(placedSides.end(), availableSides, it);
        break;
    }
}
```

```
}
```

Then, depending on the direction, the rotation point, axis and degree are calculated.

```
vec3 offset = vec3(
    direction.x / 2,
    direction.y == 1 ? 0.5 : -0.5,
    direction.z / 2);

vec3 rotationAxis = vec3(
    (direction.x == 0 ? 1 : 0),
    0,
    (direction.z == 0 ? 1 : 0));

float degrees = ((direction.x != 0)
    ? -90.0f * direction.x
    : 90.0f * direction.z) * (direction.y != 0 ? 2 : 1);
```

The animation is then executed using an Ease-In animation function to calculate the animation steps.

Effects

Shadow Maps

These are implemented as usual. In a first pass, a shadow map is rendered from the point of view of the light. During the actual render, the distance of the rendered points to the light are compared with the distance values stored in the shadow maps. If they are farther from the light than the stored values, they are interpreted as "in the shadow".

The Implementation with PFC was done following the tutorials <https://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping>, <http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/> and <http://ogldev.atspace.co.uk/www/tutorial23/tutorial23.html>.

Particles

The particles are rendered using Instancing. For the instances, two buffers are used. One defining the mesh (a quad) and color of the particles and one which stores their current location. Updating the instances is done on the CPU side with very simple physics simulating gravity and a bit of air drag.

```
float dragFactor = 0.005;
float gravity = -0.3;
```

```
for (GLint i = 0; i < numParticles; i++)
{
    Particle p = particles[i];
    p.speed = p.speed + vec3(0, gravity, 0);
    p.speed.x = p.speed.x - p.speed.x * dragFactor;
    p.speed.z = p.speed.z - p.speed.z * dragFactor;
    p.x = p.x + p.speed.x * 0.001;
    p.y = p.y + p.speed.y * 0.001;
    p.z = p.z + p.speed.z * 0.001;
    particles[i] = p;
}
```

When the player reaches the target cube, the particles are launched with an initial upward velocity.

Complex Objects

A model is loaded from an .obj file and displayed in the target cube of each level. Since these files store the vertices of the object in plain text, I only needed to read in the file line by line and translate the values into a format, that OpenGL can use to render the object.

For this, I followed the tutorial

https://en.wikibooks.org/wiki/OpenGL_Programming/Modern_OpenGL_Tutorial_Load_OBJ.

Animated Objects

Since the requirements asked for hierarchical animation, I added a small green cube inside the player cube. It rotates around its y-axis while following the rotation and movement of the player cube around it.

View Frustum Culling

To improve performance, I used bounding-sphere View Frustum Culling, since most of my objects are cubes of size 1 and can thus be easily fitted into a sphere with radius ~0.0710. Using the MVP-Matrix read out from OpenGL I was able to check if the bounding spheres were within the Viewing Frustum.

The implementation was done following the tutorial <https://r3dux.org/2011/02/how-to-perform-manual-frustum-culling/>

Game Features

See Implementation -> Gameplay

Illumination: How and what

There is a directional light used to create and render a shadow map. The light and the shaders used to implement the shadow mapping were created by following the tutorials <https://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping>, <http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/> and <http://ogldev.atspace.co.uk/www/tutorial23/tutorial23.html>, which also helped me to build a shader with ambient, diffuse, specular and texture color. Each block of the level and the player cube sides are assigned the same Shader. To differentiate them, the player cube sides and the normal level blocks use a 1x1 white texture which is multiplied by the color handed to the shader for each block/side. The textured target block on the other hand uses a 512x512 texture and the shader is handed white as a color. This way, one Shader can be used to create both simply colored blocks as well as textured ones which are all under the effect of the lighting and shadow map.

Which Effects

See Implementation -> Effects

Tools used for Models

As a complex model, I used Suzanne, which I exported from Blender.