Features:
Controls:
W Up
S Down
A Left
D Right

ESC closes the Window

Basic Game Logic:

Upon touching the Goal, the Game is finished and the Final Score is displayed in the Command window.

Colliding WIth any Objects Except for the big grey Goal Wall at the end of the Tunnel, will result in the Player taking damage. After the receiving too much damage, the player dies and the game is over.

The Score increases steadily as long as the Player is not dead or at the goal.
Or the player can earn 10 points by collecting the yellow coin blocks.

Implementation (Abgabe 2)

The Camera Follows the “Player Object”.

The Movement/ Player work through physics and applying forces.
There is a Moving Obstacle, that is also powered by physics, using constraints and applying force.

We use std::vectors to keep track of the different Game Objects and to draw / render them.
There is a pool functionality for Physics shapes, to improve performance.
Debug Mode:
F1: Toggles Debug mode
F2: Toggle Wire Frame (Only in Debug mode)
F3: Toggle Backface Culling

F6: Toggle Forward Movement on/off
F7: Toggle Hud
F8: Toggle View Frustum Culling
F9: Toggle Player Death/Damage on/off
F10: Toggle Physics Debug Drawing (bounding boxes)
F11: Toggle Free Form Camera

Effects:

Cell shading:
Implementation based on the slides [1] from last year. The intensity calculation is based on the phnog calculation. Once the intensity was calculated, it is used to sample from a 1D texture to get the intensity from said texture. The texture color is then multiplied by the intensity and added to the output color.

Contours via Edge Detection
For the Edge Detection both the Depth Buffer and the normals rendered to an FBO are used.
The implementation was created using these slides [1], these tutorials [2, 3] for rendering to the FBO, these [3, 4] for the Depth buffer and this wikipedia article [5] for the Sobel Filter.
The Sobel filter was adapted for the normal view to work for color as well.
After running Edge-Detection on the Depth and Normals image, both are combined over the Textured view of the Scene to create the View with the Edges.

Particles using Compute Shader
Implementation based on the slides [6, 7, 8].
Compute Shader:
The Particles can be spawned around the origin within a given range, this range is given by a uniform vec3.
The size can be random, this is done by giving the compute shader a vec2 for min and max size, this size value will then calculated be at the spawn time of a particle, and saved as the fourth value of the velocity vector. It stays constant throughout the particles life.
The initial Velocity can be set.
Particle Shader:
In the particle Shader there is the Possibility of setting different start and end of life sizes. This will make the particle change its relative size over the course of its lifetime.
The particles are lit by the point light, meaning that they will change colour depending on how far they are from it. Also making use of sampling from the Cell shade texture.
The base Fragment Shader of the particle also generates the Circle to display by itself(no use of texture).

Vertex Shader Animation:
Vertex shader animation is used to render the water. A plain object is first distorted, then the new normals are calculated to replace the old ones, as explained in the slides of the CG VO.[4]

Procedural Textures:
In addition to Vertex shader animation, the water shader now also consists out of Procedural Textures which are created and constantly change at runtime, to create a “Watery” feel. We used a Worley Noise Function to do so.

Gameplay:

Compulsory:
We included all compulsory gameplay points except Adjustable parameters.

Collision Detection:
The bullet physics engine is used for collision detection. When creating a Game Object with a collision shape, it is first looked up, if such a shape already exists so that it can be reused to improve performance.

Advanced Physics:
This was again made through bullet. The Players movement is done through applying force to it. There is a Moving Obstacle powered by physics, using constraints and applying force.

Model Loading:
We can load .obj files and their textures. We used Assimp as explained by learn OpenGl [9] for the model and stb_image to load textures.

Heads-Up Display:

The Hud is first rendered onto a quad and into a seperate FBO and then cut out and transferred to the main FBO via blitFrameBuffer. It is used to display your health.

View Frustum Culling:
Was implemented using bounding spheres for the objects. It can be en-/disabled through F8. And the Frustum works through a sphere with the radius slightly larger than the far plane, or through extracting the planes from the camera.
[10]

Other:

Ship rotation:
The Ship tilts when moving from side to side to indicate direction and speed.

Intuitive Camera:
Camera changes position when gaining speed and when flying to close to the ground or ceiling.

Player Particles:
The Particle systems of the ship follow it and their origin also changes on the ships rotation, by calculating it with the ships model matrix.

Additional libraries and Sources:
Physics: Bullet Physics (https://pybullet.org/)
Model Loading: Assimp (https://www.assimp.org/)
Image Loading: stb_image(https://github.com/nothings/stb)
Alongside all additional libraries used in the ECG Solution.

References:

[1]
https://tuwel.tuwien.ac.at/pluginfile.php/1721131/mod_page/content/37/CelShading_SS19.pdf
[2]
https://learnopengl.com/Advanced-OpenGL/Framebuffers
[3]
http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-14-render-to-texture/
[4]
https://tuwel.tuwien.ac.at/pluginfile.php/1721131/mod_page/content/37/Animation_SS18.pdf
[5]
https://de.wikipedia.org/wiki/Sobel-Operator
[6]
https://www.cg.tuwien.ac.at/courses/Realtime/repetitorium/rtr_rep_2016_particles
[7]
https://tuwel.tuwien.ac.at/pluginfile.php/1721131/mod_page/content/38/GPU_Particles_SS18.pdf
[8]
https://tuwel.tuwien.ac.at/pluginfile.php/1721131/mod_page/content/38/ComputeShader_SS18.pdf
[9]
https://learnopengl.com/Model-Loading/Assimp
[10]
http://www.lighthouse3d.com/tutorials/view-frustum-culling/