

# Documentation

## Game description

The player spawns in the center of the map. You have to collect 8 gas cans to win the game. When you collect the first gas can, the monster spawns. It chases the player and becomes more and more aggressive depending on how many cans you have collected. The monster can only be seen with Night Vision. If you see the monster with the Night Vision, you scare it away and it has more distance to the player again. However, the Night Vision has a rechargeable battery (10s). If you collect a canister you get 30% battery back. If the monster catches you, you lose the game. To help you find your way, there is a compass (top center of the window) and 4 sculptures which look different. All 4 figures look in the direction of the spawn (center of the level). The point light in the middle is also used for orientation to find the center faster.

### Hints:

The game is very difficult, so we have built in debug options to cheat your way through the game. For example, the player's position can be displayed with F9 and the positions of the canisters can be found in the console. Furthermore, the monster script can be deactivated by pressing F10. These options ensure that you win the game without much effort.

## Steuerung

- WASD: moves the character
- Mouse: moves the camera
- Space: jump
- ESC: Quit
- Left mouse button: Toggle Nightvision
- Arrow key up / down: Change brightness

### Debug Features:

- F8: Deactivate/activate view frustum culling
- C: writes the number of rendered objects to the console
- F9: Displays player position and writes canister positions to console
- F10: deactivates/activates monster script

## Additional libraries

**lua54.lib with sol2 wrapper:** Our game uses the scripting language lua54 with the Lua API wrapper Sol2. These libraries are used for the movement of the monster. The monster runs towards the player at a certain speed, which is increased each time gas canisters are collected, and stops when the player catches sight of the monster. The script is integrated in the `initLua()` function in `Monster.cpp`.

**assimp.lib:** The `assimp.lib` library makes it possible to import models from different formats into a standardized data structure, which can then be used in graphical applications or, as in our case, in a game engine. Assimp takes care of parsing the files, applying transformations and processing material and texture information.

**freetype.lib:** FreeType is an open source software library that makes it possible to load and render fonts from files such as TrueType (TTF), OpenType (OTF), etc. It is used to generate texts in graphical user interfaces and games. In our case, this library is used to generate the text in the HUD for e.g. the number of gas canisters collected and required.

**sfml-audio.lib:** This library was specially developed for handling audio. SFML Audio offers simple functions for loading, playing and processing audio files in formats such as WAV, OGG or FLAC. The library also supports the playback of music tracks and the control of real-time sound effects. In our game, this library was used to create the soundscape and sound effects.

**soil2.lib:** Soil2 is a library for simple loading, saving and manipulation tasks of textures in OpenGL applications. It supports various image formats and makes it possible to load textures directly into OpenGL. This library is no longer used in our final version of the game, but was used in the early days of our game engine.

**BulletCollision\_vs2010\_x64.lib:** BulletCollision\_vs2010\_x64 is a static library from the Bullet Physics Engine, which is specially designed for handling collision detection in 3D applications. By using this library, all collision queries were implemented. By adding rigid bodies to the various objects such as cameras, trees, etc., the player is prevented from running through models. The library is integrated as debug and release.

**BulletDynamics\_vs2010\_x64.lib:** BulletDynamics\_vs\_2010\_x64 is another central component of the Bullet Physics Engine and focuses on calculating the dynamics of bodies. The library therefore includes functions for simulating forces, impulses and movements of objects in compliance with the laws of physics. The library is used in the character header, for example, where a linear velocity is applied to the character when the camera moves and a central impulse is applied when the character jumps. The library is integrated as debug and release.

**LinearMath\_vs2010\_x64.lib:** LinearMath\_vs2010\_x64 is also a basic library of the Bullet Physics Engine, which provides the mathematical basis of the operations and data structures required for the physical calculations. This library contains vectors, matrices, quaternions, etc., which are essential for the calculation of movements and transformations in 3D spaces. The library is integrated as debug and release.

## Implementierte Features

### Gameplay

#### Compulsory

3D Geometry	There are many Objects in the game which are non-trivial (Trees, Sculptures, Flashlight, Gastanks, ...).
Playable	The game is playable and there are various mechanics such as the Nightvision with which you can see and chase the monster, the collection of gas tanks and the built-in collision detection that allows you to jump on objects.

Min. 60 FPS and Framerate Independence	All movements are frame rate dependent (implemented with delta time). The game always runs above 60fps on a rtx 4070, with weaker GPUs the view frustum culling must be activated to achieve 60 fps.
Intuitive Controls	See Controls!
Illumination Model	A PBR illumination model was chosen, which means that each object (except the water and the L-system bushes) has a material and texture (albedo, metallic, roughness, normal). There are three light sources: The flashlight, a pointlight over the water (blue) and a directional light from the moon.
Textures	Each model has textures.
Intuitive Camera	A classic first person camera has been implemented.
Moving Objects	The flashlight, the monster and the petrol cans move. There is also a ball located near the spawn which can be moved.
Adjustable Parameters	The parameters can be set in the config.ini file.
Win/Lose Condition	If you collect 8 gas cans you win the game and get a winning screen, from there you can end or restart the game. If the monster catches you, you get a losing screen and you can restart or end the game.

## Optional

**Collision Detection (Basic Physics):** Collision detection was implemented using the Bullet Physics Engine library. For this purpose, the camera (Camera.h) was initialized with a character (Character.h). In the header file Character.h, a RigidBody is created in BoxShape and added to the world simulation. In addition, various attributes such as Friction, Damping and Gravity are set to refine the movement. When the player moves using WASD, the direction of the movement is sent to the camera (Camera.h) and multiplied by the fixed movement speed. The move() function in Character.h is then called with the calculated movement. There, the direction of movement is converted into a velocity and applied to the RigidBody of the character as a linear velocity. The position of the camera (Camera.h) is then synchronized with that of the character (Character.h).

A CollisionMesh (CollisionMesh.h) is created for all static objects. This receives the dynamic world, CollisionShape, position, mass and, if applicable, a position offset during initialization. This allows a rigid body with the desired collision shape, position and mass to be created and added to the world.

**Scripting Language Integration:** The scripting language integration was implemented using the Lua54.lib and the Sol2 Lua API wrapper. These were used to implement the behavior of the monster. The Lua script file monster\_behavior.lua in the Monster.cpp file is initialized in the initLua() function. The updateMonster function in the monster\_behavior.lua file is responsible for the movement of the monster. The monster moves closer and closer to the player's position. If the player looks at the monster with NightVision switched on, it stops in place and looks at the player. With every gas canister collected, the monster's movement

speed increases slightly. If the player looks at the monster for a second using NightVision and then exits NightVision mode or the monster is too far away from the player, the monster respawns at a random position.

**View Frustum Culling:** The View Frustum Culling was implemented in the header file "FrustumCuller.h". Furthermore, the objects in the Game.h file were checked with the FrustumCuller to see if they are rendered. Frustum culling is applied to all objects except for the ground, monster and objects created using the L system. The culling can be toggled with F8 and the number of rendered objects can be written to the console with C.

**Head-Up Display:** The HUD is implemented in the header files "HUDObject.h" and "Font.h". Simple graphics are handled in HUDObject.h and texts (strings) in Font.h. Font.h is implemented according to the tutorial <https://learnopengl.com/In-Practice/Text-Rendering>. The HUD is used in different states of the game (Nightvision, normal, winning screen, losing screen).

## Effects

**L-System:** The L-System was implemented in the header file LSystem.h and the file LSystem.cpp and integrated into Game.cpp. Several L-System instances are created in the initVegetation() function in Game.cpp and the generated models are saved so that they can subsequently be rendered. The axiom (start symbol) and the production rules of the L-system are defined first. The L-System character string is first generated in the LSystem.cpp file and then the geometry is created using the turtleGraphics() function. The CreateModel() function then generates and returns a model of the L-system. In Game.cpp, a random position is set for each LSystem instance and each model is rotated differently. The resulting models can be seen as small two-dimensional bushes or grasses in the game world.

**Animation: Vertex Shader Animation** The vertex shader animation was applied to the water which is located in the center of the level (0, 0, 0). The water uses a mesh of type Mesh.h, and a model of type Model.h. The shader for the water consists of vertex\_water.glsl and fragment\_water.glsl. In vertex\_water.glsl, a displacement is calculated using a sine wave to simulate a wave. Furthermore, the normal vectors are recalculated to ensure correct lighting.

**Texturing: Environment Map:** The environment map was also applied to the water. The skybox (a moon) is reflected in the water. The skybox is implemented in the header file Skybox.h and uses the shaders vertex\_skybox.glsl and fragment\_skybox.glsl. The skybox was implemented according to the tutorial from <https://learnopengl.com/Advanced-OpenGL/Cubemaps>.

The reflection in the water was implemented using a cubemap, which is located in Water.h (Water contains the water model and cubemap as well as the basic texture of the water). As mentioned, the water uses the shaders vertex\_water.glsl and fragment\_water.glsl. The reflection is then calculated in fragment\_water using the cubemap. This was implemented according to the tutorial <https://learnopengl.com/Advanced-OpenGL/Cubemaps>.

**Shading: Physically Based Shading:** Physically Based Shading has been applied to all objects in the game except water, objects created by the L-System and the skybox. The shading uses albedo, metallic, roughness and normal maps. The corresponding shaders are vertex\_PBR.glsl and fragment\_PBR.glsl. The calculation of the PBR models has been realized using the tutorial <https://learnopengl.com/PBR/Lighting>, but has been greatly expanded. The objects in our game (except for the water) are from the class ModelPBRLoaded which is defined in the header file ModelPBRLoaded.h. ModelPBRLoaded uses the ASSIMP library to load obj files. Furthermore, these models must have files for albedo, metallic, roughness and normal to be able to use them with the PBR shader. The lights are defined in Game.h and sent to the shader. The obj files are located in the OBJ folder.

The structure can be summarized as follows:

PBR Object:

- ModelPBRLoaded.h
  - MeshPBR.h
- vertex\_PBR
- fragment\_PBR

An example of a non-metallic object is the floor. As an example of a metallic object, there are two statues which have a metallic texture (see map at the end of the document for position).

**Post Processing: Bloom/Glow:** The Bloom was basically implemented with the help of the tutorial <https://learnopengl.com/Advanced-Lighting/Bloom>. The bloom can be seen on the moon. To obtain the bloom, an HDRframebuffer is used, which is defined in the header file "HDRframebuffer.h". The shaders HDR\_SHADER (hdr\_vertex, hdr\_fragment) and BLUR\_SHADER (vertex\_blur, fragment\_blur) are used. The night vision was implemented in hdr\_fragment. Furthermore, the gamma value, which can be changed in the game, can be used to adjust the brightness.

## Map

