

Last Mech Standing

Gameplay

The goal of the game is to defeat as many enemies as possible before your life points reach zero. The player can freely move around the arena and take hide behind the containers or jump on them to get a better overview of the scene. The positions, dimensions and appearances of the containers are generated randomly on startup so that the arena looks different every time.

The enemies spawn randomly around the arena and try to defeat the player. They walk around by themselves and as soon as the player comes into their viewing range, they start to chase and shoot him. When being stuck in front of an obstacle (like a container), they automatically try to jump over it. During the game the enemies respawn time is lowered and their initial health points are raised, which makes it increasingly harder over time to survive.

When the player's health points reach zero, the final score (which is the number of enemies the player defeated during the game) is shown.

Controls

W, A, S, D ... Movement of the mech

Mouse ... View control

Left mouse button ... Shooting

Space ... Jumping

Project structure:

- **EXTERNAL/*** ... all external include and library files
- **GAME/*** ... the main classes that manage the game's behavior
 - Display.cpp ... wrapper object which manages the glfw window
 - Gameloop.cpp ... sets up the initial game state and runs the game loop
 - ShortKeys.cpp ... manages the short keys (F1 – F9)
- **RESOURCES/*** ... contains all files that are not source code (3D models, texture files, shader code files, fonts)
- **SCENE/*** ... all the objects the scene contains
 - **ASSIMP/*** ... Model and Mesh classes that contain the 3D data which is loaded via the Assimp library
 - **GUIELEMENTS/*** ... all the parts of the GUI that are displayed on the screen (player's and enemy's health bar, crosshair, ...)
 - **SCENEOBJECTS/*** ... all the specific objects that can be placed in the scene (Player, Enemy, Arena, ...)
 - **CAMERA.CPP** ... the camera which is used to display the scene
 - **GUI.CPP** ... wraps all GUIElements and allows easy access to the GUI parts
 - **LIGHTSOURCE.CPP** ... a directional or point light source
- **UTIL/***
 - **BOUNDINGBOX.CPP** ... wraps the bounding boxes of the SceneObjects; is used for collision detection and view frustum culling

- **FONT.CPP** ... loads and wraps a font via FreeType and makes it available to be written on the screen (for the GUI)
- **FRAMEBUFFER.CPP** ... wraps the frame buffer object which is used for the shadow mapping and bloom effects
- **QUAD.CPP** ... represents a 2D rectangle which can be used to draw textures or simple colors
- **SHADOWMAP.CPP** ... wraps a depth texture which is used for shadow mapping
- **SHADER.CPP** ... loads and compiles a shader code file and wraps the shader program
- **COMMONHEADER.H** ... contains all include's and constants that are used in every class
- **MAIN.CPP** ... initializes the external libraries and starts the game loop

Lighting, materials and models

There is one directional light source (sunlight) which illuminates the scene from above.

The objects (player, enemies and arena) are all modeled in Blender and use different materials. The player and enemy objects use diffuse maps only. The arena uses both diffuse and specular maps.

The enemies are hierarchically animated as their feet rotate independently from their bodies.

Effects

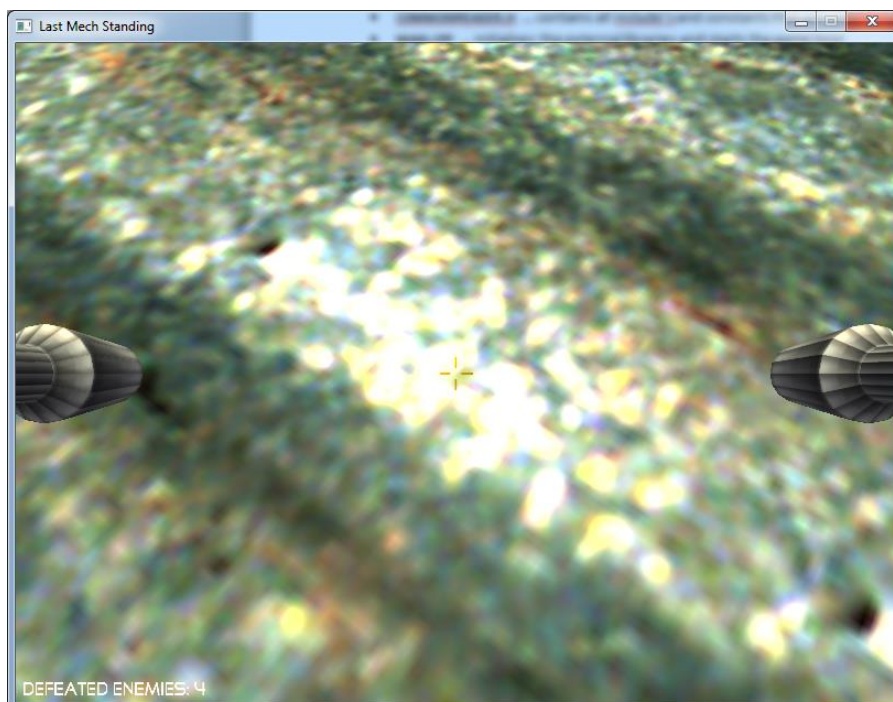
Shadow Maps with PCF (1.5 pts)

- Can be enabled/disabled with F6
- The purpose of shadow mapping is so that objects cast shadows on each other, which results in a much more realistic looking scene.
- The scene is first rendered as a depth map from the perspective of the light source. After that everything that is not directly visible by the light source is determined to be a shadow area.
- Since the light source in the game is directional and therefore has no specific location, the depth map is rendered using an orthographic projection. There is still a light source location required for the rendering process, but this location is adjusted dynamically to the player's position. This results in the player being always in the center of the depth map so that around him the shadows are displayed correctly.
- To avoid shadow acne, a bias value was added in the shader.
- PCF (percentage-closer filtering) is used to make the edges of the shadows smoother. This is achieved in the shader due to sampling the surrounding texels of the depth maps and averaging the result.
- The following tutorial was used as a guide for implementing shadow maps with PCF:
<https://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping>



Bloom (1 pt)

- Can be enabled/disabled with F7
- Bloom makes bright areas on the screen “shine” by adding a blurred glow effect on top.
- The scene is rendered twice: one time as usual, the other time with only the bright areas. The second texture is then blurred with a Gaussian blur and combined with the normal render.
- The two parallel render processes are implemented with MRT (multiple render targets) which means that a single fragment shader can calculate multiple fragment colors and store them independently.
- The following tutorial was used as a guide for implementing shadow maps with PCF:
<https://learnopengl.com/#!Advanced-Lighting/Bloom>



Advanced OpenGL

- **FRAME BUFFER OBJECTS**

- FBOs Are used to store render outputs as textures so that they can be processed further (e.g. blurred for the Bloom effect)

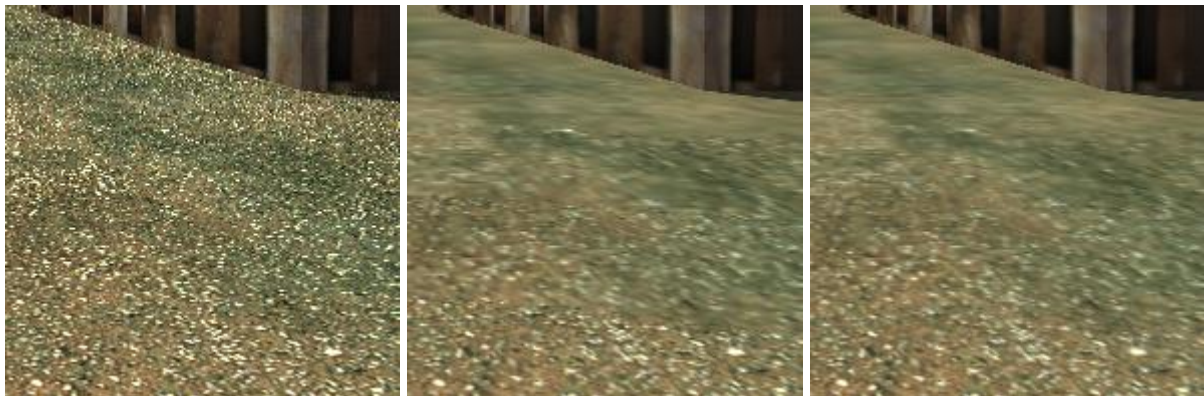
- **BLENDING**

- Can be enabled/disabled with F9
- Blending is used to display textures that are partly transparent. Is used for the crosshair texture and the FreeType fonts



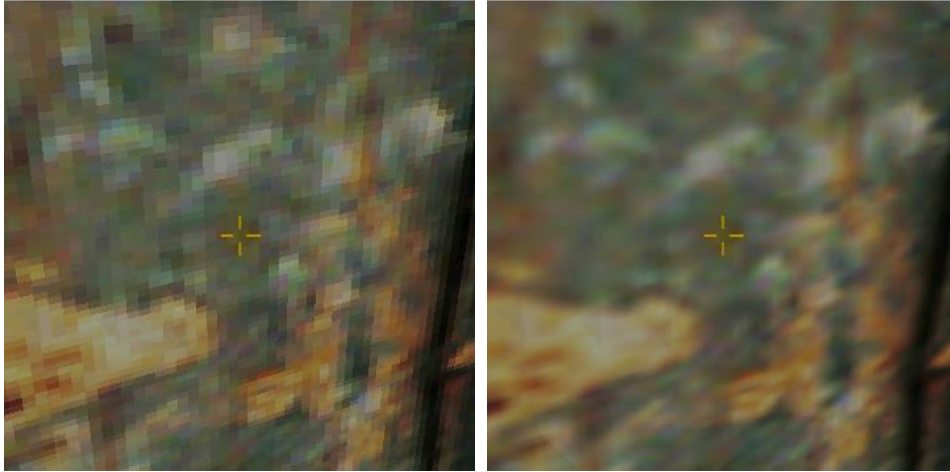
- **MIP MAPPING**

- Can be switched with F5
- Mip maps are generated when loading a texture. They are downsampled versions of the image so that when they are displayed smaller than their actual size is, a lesser detail version is used.
- Off ... no mip mapping is applied
- Nearest neighbor ... there is no interpolation between the different versions
- Linear ... linear interpolation between the different versions results in a smooth look



- **TEXTURE SAMPLING QUALITY**

- Can be switched with F4
- The texture sampling determines what happens when a texture is rendered bigger or smaller than their actual size is
- Linear ... results in a pixelated look
- Nearest neighbor ... results in a smoother look



- **Fonts**

- Are used to display parts of the GUI: the number of defeated enemies (always visible), the frame rate (enable/disable with F2), the number of currently displayed objects (enable/disable with F8) and the game over screen

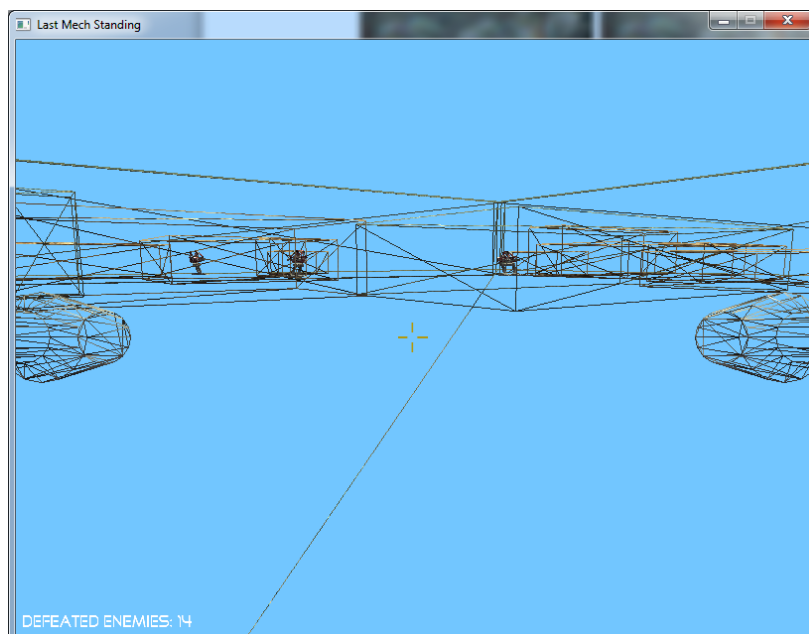


- **VIEWFRUSTUM CULLING**

- Can be enabled/disabled with F8
- Viewfrustum culling prevents objects that are outside of the player's viewing range to not be drawn, which increases performance. It takes advantage of the object's bounding boxes which were implemented for collision detection. If an object is entirely behind the player, it is automatically not considered to be drawn. Otherwise the algorithm checks if part of the object's volume is inside the viewing frustum and then decides whether or not to draw it.
- The number of actually drawn objects is shown on the screen.

- **WIRE FRAME**

- Can be enabled/disabled with F3
- Draws only the wire frame of the scene and displays it on the screen



Shaders

- **MODEL_LOADING.VERT** ... Vertex shader for textured models loaded with Assimp
 - performs the model-view-projection transformation
 - passes the vertex positions, normals and texture coordinates to the fragment shader
- **MODEL_LOADING.FRAG** ... Fragment shader for textured models loaded with Assimp
 - performs the lighting calculations (ambient, diffuse and specular component) and applies the diffuse and specular maps
 - is currently able to handle seven diffuse maps, seven specular maps and one directional light
- **QUAD** ... simple shader for colored or textured 2D elements (like the GUI parts)
- **GAUSS_BLUR** ... performs the Gaussian blur for the Bloom effect
- **BLOOM** ... performs the additive blending of the normal and the blurred render image
- **BULLET / BULLET_ENEMY** ... simple color-only shader for rendering the bullets
- **FREETYPE_GLYPH** ... used for rendering the FreeType fonts
- **SAMPLE_DEPTH_SHADER** ... for rendering the depth map used during the shadow mapping calculations

Used libraries

- **GLFW** ... API for creating windows and contexts as well as receiving input and events from the operation system
<http://www.glfw.org/>
- **GLEW** ... OpenGL extension loader
<http://glew.sourceforge.net/>
- **GLM** ... math library for OpenGL
<http://glm.g-truc.net/0.9.8/index.html>
- **SOIL** ... library for loading different image formats
<http://glm.g-truc.net/0.9.8/index.html>
- **ASSIMP** ... enables loading of different 3D model formats
<http://assimp.sourceforge.net/>
- **FREETYPE** ... enables writing fonts on the screen
<https://www.freetype.org/>

Used tutorials and code snippets

- **LEARN OPENGL** ... extensive OpenGL tutorial for beginners
<https://learnopengl.com/>
- **HSV TO RGB COLOR CONVERSION**
<http://stackoverflow.com/a/8208967>