

# Submission 2 Report

## Armored Combat Group

### 1. Gameplay

#### a. 3D Geometry

Our game incorporates only non trivial 3D geometry which all have been externally loaded with the open asset import library Assimp [1]. All 3D models were created by us using Blender [2]. The models are exported using the .obj or .dae file format. Using an Assimp importer, the files are read and the meshes are processed separately. After the meshes have been initialized, they will receive a vao and the corresponding vbos that are used to load the vertex data into the vertex shader.

#### b. Playable

The game is playable to the extent that it does include gameplay mechanics like:

- avoiding enemy projectiles
- shooting at a target
- movement in 3D space

The game can also start via .exe and runs error free.

#### c. Advanced Gameplay

Our game incorporates an enemy that has advanced and sophisticated behaviours based on its current healthpoints. Its pool of behaviours contain 4 different moves and 2 scripted events.

Furthermore mechanics like limited player ammo, a secondary weapon, breaking the enemy shield and avoiding homing missiles are considered to raise the complexity and enjoyment of the game.

#### d. Min. 60 FPS and Framerate Independence

Our game can run with 60 FPS (16ms) at all time and a max. of 1200 FPS (0.8ms). All movements are framerate independent and the framerate can be adjusted in game by pressing "p" following up with a number for fps and pressing "p" again.

#### e. Win-Lose condition

Win condition:

Using primary (left-mouse-button) and secondary (right-mouse-button) projectiles, the player has to hit the enemy and decrease its health bar. If the enemy hp bar reaches 0, the player wins.

Lose condition:

The player needs to avoid enemy projectiles. If the players hp bar reaches 0, the player loses.

Scene stops moving and Win-Lose screen is showed.

- f. **Intuitive Controls:**
  - WASD for horizontal movement in space (polling)
  - Space and C/Left-Shift for vertical movement in space (polling)
  - Left mouse button for shooting primary projectile (callback)
  - Right mouse button for shooting secondary projectile (callback)
  - Continuous player movements are using own polling methods that triggers booleans instead of using the conventional `glfwGetKey()` function in order to have more control on handling movement inputs like disabling certain keys etc.
- g. **Intuitive Camera**
  - Our type of game doesn't allow a free moving camera. Instead the camera strictly follows behind the player and dynamically changes distance to player based on position of the player within its movable space.
  - For a free moving camera a pause/debug mode has been implemented and can be toggled with "p". Followed learnOpenGL Tutorial[3].
  - In pause mode WASD Space and C are used for moving and mouse position polling for rotating the camera. Holding down Left-Shift speeds up movement.
  - Conventional `glfwGetKey()` is used for polling movement.
- h. **Illumination Model**
  - Every 3D-Model loaded with assimp[4] has material coefficients that are loaded into the fragment shader. There is a directional light that is shining on the back of the player character and a pointlight that is highlighting and moving along the boss. Every model has normal vectors and uv values.
- i. **Textures**
  - The landscape and projectiles are textured with different textures. Because they are loaded with assimp they also have uv values.
  - Texture for particles are loaded with ECG Framework provided loader.
  - Textures for any other objects are loaded with imported library `stb_image`[19].
  - Mipmap and trilinear filtering are enabled.
- j. **Moving Objects**
  - Our whole scene is automatically moved along a track by updating the transformation of our physics-rigidbodies and mapping the transformation onto the rendered model, while the track is rendered static.
  - Player and enemy can also move within a specified bounding box.
  - Projectile movements, however, are handled by the physics engine `pyBullet` [5].
- k. **Adjustable Parameters**
  - Parameters in `settings.ini` can be adjusted.
  - Includes:
    - Screen Resolution, Fullscreen-Mode, Refresh-Rate, Brightness

## 2. Optional

### a. Collision Detection (Basic Physics)

Physics is covered by Bullet3 [5].

Player and enemy are modeled as kinematic rigid bodies and collision detection with dynamic rigid bodies (projectiles) are handled automatically by the engine.

Collision detection against the invisible bounding box are handled manually by enabling static-static/kinematic-kinematic collisions and looping through the contact manifolds and triggering collision events. Tutorials: bullet doc [6], pybullet forum [7], Book: Learning game physics with bullet physics and opengl [8]

### b. Advanced Physics

All projectiles are dynamic objects and thus also controlled by bullet.

Projectiles will bounce against other Projectiles and the enemy shield.

Getting hit by projectiles can trigger events like decreasing hp bar.

At win/lose the enemy/player turns from kinematic into a dynamic rigidbody with torque impulse and gravity. The World gravity is set to -2 towards the vertical axis.

### c. Heads-Up Display

A 2D overlay used for showing HP bars and current ammo.

The HUD also incorporates textured objects (symbol besides ammobar and win/lose screen) and is implemented using blending following a learnOpengl tutorial[9].

F2 to toggle HUD.

## 3. Effects

### a. Cel-Shading:

the color of a fragment is translated to hsv space, then the z-value is discretized with the levels of cel shading. After the discretization, the color will be translated back to rgb space.[10]

### b. Contours via edge detection:

Using 2 framebuffers the scene is first rendered into a newly created buffer. Then the post-processing shaders will take the rendered scene as a texture and apply a sobel filter to the x and y direction to detect edges. If an edge is detected, the fragment will be black. After this the processed image will be projected on a quad in the standard framebuffer. [11]

### c. GPU particle system using compute shader

The particle system uses 1 compute shader and 1 set of vertex/geometry/fragment shaders. Using SBOs and buffer swapping the particle data is loaded and computed in the compute shader. An atomic counter keeps track of the count of particles. After the particle data has been computed, the data is loaded into a pass through vertex shader. In the geometry shader the point-data will be converted into quad-data and in the fragment shader, a texture will be applied to each fragment. [12]

The particle system is seen at the end of the game, either when the player loses or

the boss is defeated. The defeated character will produce a smoke particle system.

d. GPU Vertex Skinning:

The models are rigged and animated in blender[13] and exported using the .dae file format. The file is loaded and processed with Assimp, the bone data is saved using the struct BoneInfo and the bone data for each vertex is saved using the struct VertexBoneData. In the main loop the final transformation for each bone is calculated and then loaded into the vertex shader. In the vertex shader the position of a vertex is then multiplied with the final transformation of the corresponding bones. [14]

The animations are always seen. The boss moves up and down a bit and wiggles his cannons sometimes. The player moves her legs and floating devices.

## Features of the game:

The game is a 3rd person shooter. The player controls a character and **move** around in 3D-space horizontally with the **W,A,S,D keys** and vertically with the **Spacebar** and the **C or Left-Shift key**. The player can shoot projectiles with the **left mouse button** and a more powerful attack that has a shorter distance with the **right mouse button**.

The **P key** pauses the game and enables a debug camera that can freely move around.

While in **P-mode** input a target refresh rate and toggle **P-mode** again to **change the FPS**.

Pressing **O key** will toggle a **FPS DEBUG** and write into the **console**.

On Game Over the player can press **R** to restart the game.

The goal of the game is to defeat the enemy character that shoots projectiles and missiles at the player with specific patterns. Dodging the projectiles gets harder as the health of the boss gets lower. The boss also can trigger event attacks that must be dodged or a significant amount of damage will be received. He can also summon a shield where the player's projectiles will bounce off.

Player projectiles are limited, but reload automatically. A reload animation in HUD will play.

Movement of player and enemy are non linear. Using the goalPost method the original character object is moving by following after an invisible goalPost to simulate fluidity.[15]

The game includes background music from the game Nier:Automata, if this is a problem due to copyright issues it can also be substituted with a royalty free soundtrack.

The BGM has been loaded with irrklang[17]

### Special feature:

Window size is scalable in game. Viewport, HUDS, postprocessing etc will also update accordingly.

Keys summary:

ESC close game

F1 toggle wireframe

F2 toggle HUD

F3 toggle physics DebugDraw (wireFrame)

F4 toggle physics DebugDraw (Aabb)

F6 toggle fullscreen  
P toggle pause/debug mode  
O toggle FPS Debug output into console  
R to restart the game after game over

## Additional libraries:

- The Open Asset Importer Lib (Assimp) for model loading[18]
- Bulletphysics/Bullet3 for physics[5]
- Irrklang for sound[17]
- stb\_Image for some texture loading[19]

## Quellenverzeichnis:

- [1] Learn opengl: <https://learnopengl.com/Model-Loading/Assimp>
- [2] blender: <https://www.blender.org/>
- [3] learnOpenGL: <https://learnopengl.com/Getting-started/Camera>
- [4] The Open Asset Importer Lib: <https://www.assimp.org/>
- [5] pyBullet: <https://github.com/bulletphysics/bullet3>
- [6] bullet doc <https://github.com/bulletphysics/bullet3/tree/master/docs>
- [7] bullet forum <https://pybullet.org/Bullet/phpBB3/>
- [8] Chris D., Learning Game Physics with Bullet Physics and OpenGL, 1. Auflage, Packt Publishing, 2013
- [9] learnOpenGL: <https://learnopengl.com/Advanced-OpenGL/Blending>
- [10] Cel Shading and Contours from Revision Course 2019 :  
[https://tuwel.tuwien.ac.at/pluginfile.php/1721131/mod\\_page/content/37/CelShading\\_SS19.pdf?time=1553158642339](https://tuwel.tuwien.ac.at/pluginfile.php/1721131/mod_page/content/37/CelShading_SS19.pdf?time=1553158642339)
- [11] OpenGL Postprocessing:  
<https://learnopengl.com/In-Practice/2D-Game/Postprocessing>
- [12] Particle System with Compute Shader:  
[https://tuwel.tuwien.ac.at/pluginfile.php/1721131/mod\\_page/content/37/GPU\\_Particles\\_SS18.pdf](https://tuwel.tuwien.ac.at/pluginfile.php/1721131/mod_page/content/37/GPU_Particles_SS18.pdf)  
<http://web.engr.oregonstate.edu/~mjb/cs575/Handouts/compute.shader.2pp.pdf>
- [13] rigging and skeletal animation in blender:  
<https://www.youtube.com/watch?v=SBYb1YmaOMY>
- [14] GPU Vertex Skinning: <http://ogldev.atspace.co.uk/www/tutorial38/tutorial38.html>  
<https://www.gamedev.net/forums/topic/688121-skeletal-animation-assimp-glm-and-me-in-between33/>
- [15] Basic game physics fluid movement:  
<https://www.genericgamedev.com/general/basic-game-physics-fluid-movement/>
- [16] Title: Alien Manifestation, by Keiichi Okabe. Album: NieR:Automata Original Soundtrack
- [17] irrKlang <https://www.ambiera.com/irrklang/>
- [18] assimp <https://www.assimp.org/>
- [19] stb\_image: <https://github.com/nothings/stb>