Beerpocalypse

Documentation – Submission 2 (Final Game)

Marc Putz	**Fabian Holzschuster**
12024709	12025338

The Game

The game takes place in a Zombie apocalypse where the player needs to find beer collectibles to finish the level. Zombies will attack the player as soon as they see him, so the player must shoot them beforehand. The player needs to explore the map and find all beer collectibles, which they need to finish the game. The game is won after the player has found and collected all beer bottles. The game is lost when the Zombies attack the player too much and they lose all their health.

Some bottles may be hidden from the player and aren't accessible immediately. This is the puzzle element of our game, where the player needs to find a hidden button in order to progress to new, previously hidden, areas.

All ways the player can interact in the game are listed in the "Controls" section. The game also contains various sound effects (shooting, walking, zombie grunts, etc.).

Gameplay Hint: *At one part, the player needs to find a wooden crate to jump onto in order to reach a higher platform and progress. Another part requires the player to find a button to make a fake wall disappear.*

Controls

The game's main controls are all explained in the Pause Menu, which is accessed by pressing the ESC key. They are:

• WASD	Move Around
• L SHIFT	Sprint
• Mouse	Look Around
• Scroll wheel	Increase/Decrease FOV
• Spacebar	Jump
• LMB	Shoot
• E	Interact
• R	Reload
• F	Toggle Flashlight On/Off
• ESC	Pause/Unpause
• +/- (Numpad)	Increase/Decrease Brightness (Exposure)
• Page Up/Down	Increase/Decrease Gamma
• Arrow Up/Down	Increase/Decrease Volume

Furthermore, the game has various debug features that can also be toggled on or off. The key bindings can be seen in the Debug HUD (accessible by pressing F10). These are:

• F10	Toggle Debug HUD Menu
• F1	Toggle Wireframe
• F2	Toggle Backface Culling
• F3	Toggle Game HUD GUI
• F4	Toggle Normal Mapping
• F5	Show Normals
• F6	Toggle Bloom

• F7	Enable/Disable Zombie Pathfinding
• F8	Enable/Disable Infinite Jumping

Also, a third-person camera is implemented for debug purposes (not meant for normal gameplay as this is a first-person game). It can be toggled by pressing:

• C	Switch First/Third Person Camera

Implementation of Gameplay and Effects

Most of our implementation was done by following online tutorials and reference links provided by the LVA team, PhysX calls were mostly done by taking a deep look at the online NVIDIA PhysX API.

Compulsory Gameplay

- **3D Geometry (6 Points)**
  The game is implemented in OpenGL as a 3D game and can render all types of 3D geometry. This does not only include simples like Cubes, Spheres, etc., but it's also possible to load complex 3D meshes and display them in-game. Loading Meshes is done by using the "assimp" library.
- **Playable (3 Points)**
  As mentioned above, all basic gameplay mechanics are implemented. These include moving around, looking around, jumping, shooting enemies, etc.
- **Advanced Gameplay (3 Points)**
  The term "*sophisticated gameplay*" leaves much room for interpretation; however we have tried our best to include complex game-logic and gameplay elements. These include button interactions, fake walls, zombie pathfinding, physics-based wooden crates, etc.
- **Min. 60 FPS and Framerate Independence (3 Points)**
  We tried to optimize our code and third-party assets to keep the game at a stable and high framerate. On a M1 MacBook Pro running Windows in a Virtual Machine, we normally reach around 300~400 frames per second.
  When the Debug Menu is enabled, the framerate may drop to under 60, because the current on-screen text rendering is admittedly not very efficient and is only really used to read off some currently relevant data for debugging. *However*, this only applies to debug mode and does not happen at all during normal gameplay.
  All update methods and physics callbacks are implemented using deltaTime as a time reference, which means the game runs completely framerate independent.
- **Win/Lose Condition (3 Points)**
  The player can win the game by collecting all beer bottles and lose the game by being defeated by a zombie.
- **Intuitive Controls (2 Points)**
  Regarding our control scheme we looked at other common games of the genre and similar games and stuck to their controls. Only our debug controls are quite specific, but they are explained in the debug HUD and should not be needed for normal gameplay anyway.
- **Intuitive Camera (2 Points)**
  As our game is in first-person, we have implemented a first-person camera which shows the game from the player character's point of view. The player can move in all 3 dimension and the camera follows this movement precisely. For debug purposes, we also implemented a third-person camera which you can toggle by pressing the C key. However, the third-person camera is not meant for normal gameplay as the player model itself should not be seen.
- **Illumination Model (2 Points)**
  The game uses the Blinn-Phong shading model for illumination and supports diffuse and specular textures. All three main types of lights are implemented: Directional Lights, Spotlights and Point

Lights. The shading uses normal vectors, which are either implicitly given or provided by a separate normal map for some complex geometries and textures.

- **Textures (2 Points)**
  All our used game objects use textures. Mipmapping and trilinear filtering are also enabled.
- **Moving Objects (2 Points)**
  The game uses static and dynamic elements. Dynamic elements are controlled either manually by a script or controlled by the physics engine and can move around the map. The player and zombies are moved both manually and by physics, other objects like wooden crates can be shoved around by the players or zombies using the physics engine.
- **Adjustable Parameters (1 Point)**
  While most parameters can be adjusted in-game, most of them have default values stored in an INI-file. It sets the default values for the screen resolution (width/height), Fullscreen mode, the refresh rate, brightness parameters (exposure and gamma), camera parameters, audio volume, etc.

Optional Gameplay

- **Collision Detection (4 Points)**
  The game uses the PhysX physics engine to provide collision detection and prevent objects from walking into or through another.
- **Advanced Physics (6 Points)**
  The game contains dynamic objects which are solely controlled by physics, e.g., wooden crates. Furthermore, every physics object can have a layer-mask attached to it, which will determine when it uses a collision-callback instead of the standard behavior. Using this, the collider will act like a trigger and custom code can be implemented. This is used e.g., for beer collectibles and zombie attacks.
- **Heads-Up Display (4 Points)**
  The game has two HUDs: one for normal gameplay (health, ammo, etc.) and one for debug purposes. The gameplay HUD uses pictures, the debug HUD uses text.

Effects

- **CPU Particle System (8 Points)**
  The particle system works by creating a 2D quad/billboard which is used for drawing the particle texture(s). Every frame of the game the position, lifetime and other values of each particle are updated on the CPU to determine if and where the particle should be drawn. These per-particle values are then written into buffers and passed to the GPU where via instancing we can draw one particle per chunk of data we provide in the buffers all within a single draw call.
  For this effect we used the suggested tutorial on http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/
- **GPU Vertex Skinning (20 Points)**
  First the game loads models and animations into memory with the assimp-library, traverses assimp's data structure and extracts data about the meshes and animations and saves this information in our own data structures. The vertex skinning uses the extracted animation to determine at which point in the animation we currently are and interpolates the required transformation matrices via the closest previous and next keyframe transformations. The resulting list of matrices is then passed to the vertex shader where the required transformation for the current vertex is then extracted from them via the ID associated with the vertex of the mesh. The transformations are then multiplied with the corresponding weights saved in the vertex and summed up according to the maximum amount of influence the bones have on each other. The resulting final transform is then used to transform the vertex of the model to the position where it should be in the current animation.

For this effect we used the tutorial on https://learnopengl.com/Guest-Articles/2020/Skeletal-Animation which, after checking the suggested tutorial by “ogldev”, seemed very similar.

- **Specular Map (4 Points)**
  By sampling the object’s specular map, just like a diffuse texture, in the fragment shader via texture/UV coordinates we can extract a specific color (brightness) value to add to the diffuse color of the object and add object-specific highlight areas. The specular part of the lighting calculations is done via the Blinn Phong reflection model.
  For this effect we used the suggested tutorial on https://learnopengl.com/Lighting/Lighting-maps
- **Simple Normal Mapping (4 Points)**
  In our normal mapping implementation we create the TBN (tangent, bitangent, normal) matrix in the vertex shader, which is then used in the fragment shader to transform the sampled normal of the normal map from its tangent-space into world-space, allowing us to use this transformed normal in the lighting calculations instead of the per-vertex normals of the object.
  For this effect we used the suggested tutorial on https://learnopengl.com/Advanced-Lighting/Normal-Mapping
- **Bloom/Glow (8 Points)**
  We create our bloom effect via multiple render passes. In the first pass we render the scene into a texture, while also extracting points in the scene of high brightness and saving those into a separate texture. We then blur the image containing the bright spots of the scene in the second render pass repeatedly, creating a glowing effect around those bright spots. In the third render pass we combine the original texture of the scene with the texture containing the blurred bright spots into one and render it onto a quad/billboard which is then shown on the screen.
  For this effect we used the suggested tutorial on https://learnopengl.com/Advanced-Lighting/Bloom

Third-Party Libraries

- Assimp – Model Loading
  - Used for loading complex 3D models and geometries
  - http://assimp.org/
- FreeType – Loading fonts as bitmaps
  - Used for on-screen text (debug HUD, pause menu, etc.)
  - https://freetype.org/index.html#
- Stb – Image file loading
  - Loads image files (textures, HUD elements, etc.)
  - https://github.com/nothings/stb
- NVIDIA PhysX 4.1 – Physics engine
  - Used for physics-based interactions of game elements
  - https://github.com/NVIDIAGameWorks/PhysX
- irrKlang – Sound library
  - Plays sounds in-game
  - https://www.ambiera.com/irrklang/