

1, 2, Shoot: Documentation

A 3D FPS-shooter programmed in C++ using OpenGL

TU Wien Computer Graphics Course 2020S

11704336 Emanuel Weingartner

11810909 Samo Kolter

Gameplay

Our game is a First-Person-Shooter, which features 11 levels/rooms, steadily increasing in difficulty and introducing the game-mechanics in a step-by-step fashion. As each shooter needs a camera, we implemented our own version of a **FPS-camera** with help of [this](#) tutorial (and some additional modifications). We also integrated the [Bullet](#) physics engine into our game, which allowed us to implement **advanced gameplay/physics features** more easily.

The game features the usual, **intuitive**, FPS-shooter **controls**. In order to “clear” a room and advance to the next one, all targets in it must be destroyed. The player can move around the scene, jump, shoot targets, throw bombs (the velocity/distance of the throw can be controlled, the bombs can destroy targets) and also pick up other game objects and use them to reach the next room (by using them as “stepping stones” for getting over bigger obstacles, which cannot simply be jumped over).

Of course, our game features proper **collision detection**, improving the level of immersion significantly. There are also **moving objects/platforms**, which the player sometimes has to use in order to advance to the next level. The player can also walk up/down stairs to reach the targets which have to be shot.

Once all rooms are “cleared”, the player gets notified that he/she has won. The player has to clear the rooms in a certain amount of time: If the timer reaches zero, the player loses (a fitting overlay is also shown). The game finishes as soon as **the player wins or loses**.

The game can be started with **adjustable parameters** (modifiable in settings.ini file).

Physics

As Bullet isn't well documented, we combined various sources from the internet with old-school trial-and-error methods. All the visible objects in our game have fitting counterparts in the “physics world”. We implemented our own version of a player controller. All static and dynamic objects in our game are simulated in Bullet (the walls, moving platforms, holdable objects, even the lights). Bullet takes care of collision detection (we use `btDynamicsWorld` to add objects to the world on the fly - of course we also remove them if they should disappear). Bullet's ray-tracing is used for the shooting: We shoot a ray from the view of the camera, if one of the targets intersects the ray, it is deleted from the scene. Throwing bombs is also implemented with the help of Bullet.

Playability

We made sure our game runs consistently at 60 FPS (as required). Therefore we made some optimizations in the render loop. We also make sure that only the part of the game which is currently needed is rendered to improve performance.

3D Geometry

We implemented the loading of complex 3D models created in modelling software with the [Assimp Model Loading library](#). This task was again made easier by referring to a [tutorial](#). Additionally, the object's lighting was also improved. We added Phong-shading as well as normal maps. A comparison of the object without any lighting (diffuse textures only) and with Phong-shading and normal mapping can be seen in the beginning ('waiting room').

HUD

The game features a HUD displaying a crosshair and information like the game progress, time left to complete the game and (if a bomb is being thrown) the velocity with which the bomb will be thrown. For the implementation, we used the [freetype](#) library (with help of another helpful [tutorial](#)). We cheated a bit for the crosshair: It is just a character/glyph of a special font we found [here](#), which we then simply put on the center of the screen.

Illumination Model, Textures

Each room in our game features a **point light** in the center of the room which is enabled as long as there are still targets to "clear". All our game objects feature **textures** (with **UV coordinates mipmapping and trilinear filtering**) and are illuminated with **Phong shading** (the only exception being one of the two models in the first room, which is rendered with diffuse texture only for comparison reasons).

We use **textures** which get added/combined into a **material**. For each created material, properties like **ambient, diffuse and specular lighting** can be set. Furthermore, **specular maps** and/or **normal maps** can also be added to materials. Those materials are then used to render all the simple **geometry** objects in our game (cuboids, spheres and cylinders), which make up most of the scene (except for the **complex models** which are loaded from an external file). We also added functionality to change how textures should be scaled for each object.

Effects

CPU Particle System

Particle systems try to emulate effects like explosions, fire, confetti etc. by drawing many small moving objects onto the screen. We created a **CPU particle system with instancing** (multiple copies of the same mesh with some modified properties like position, size and color). We used [this](#) tutorial as a starting point, however it took quite some time to make it work, as things weren't really explained well.

The effect can be seen in use at the start of the game (in the waiting room), when shooting targets during the game, and at the very end (CONFETTI!).

Vertex Shader Animation

For this effect, we used [slides](#) from the Computer Graphics course of the Vienna University of Technology as a starting point. The **vertex shader animation** can be seen in action in the ‘waiting room’ (the “pool” with animated water/waves). We animate vertices in the vertex shader with a sine wave animation, setting a time offset in the vertex shader via uniforms. The surface **normals** are also computed in real-time with some more math (derivatives).

Specular and Normal Mapping

Specular maps make specular highlights of our game objects a lot more realistic. The specular map is just a second texture which can be added to a material and be used when calculating the specular highlight of an object. It is a simple grayscale texture which “encodes” whether a given part (UV-coordinate) of an object’s texture should have a specular highlight and “how strong” it should be. For our implementation of this feature, we built on the explanations from [this](#) tutorial.

Normal maps allow one to add a whole lot of detail to each face of a mesh by encoding information on the “structure” of a surface in another texture, called a normal map. The surface normal is modified by the sampled normal from the texture. The math behind is a bit more complicated, all lighting calculations have to happen in the so-called “tangent space”. For the implementation of this feature, we built on the explanations from [this](#) tutorial. We extended the Geometry, GeometryData and (Texture)Material classes of the provided in the ECG-Framework, which is mentioned at the end of this document.

A comparison of objects with vs. without specular/normal maps can be seen in the first room: the “container”/cube with steel frame is rendered twice: once with simple phong shading and once with a specular map, which makes only the “steel part” of the container shine. The “wood part” has no specular highlight. The effects of specular mapping can also be compared looking at the two spheres (one features specular mapping, one doesn’t). Lastly, the two cubes with brick textures show the difference between simple phong shading with a texture and material params vs. phong shading with diffuse texture, normal mapping and specular mapping. The effect of specular and normal mapping becomes very apparent when looking at the brick walls on the left and right side of the room: Thanks to normal/specular mapping, the bricks look “3D” despite being part of a regular, flat surface.

Bloom/Glow

The **Bloom effect** can be used to make light sources and very bright objects “glow”. For this effect, the scene is rendered into a separate framebuffer with two color buffers (and dept buffers). The first color buffer gets the whole image, the second only the “bright parts”. The image with the bright pixels then gets blurred and as a last step the blurred “bright parts” are

put on top of the regular image, creating a texture, which is then simply rendered onto the screen as a quad. We added bloom to our project with the help of [this](#) tutorial.

In our game, bloom is applied to the each room's **light**: it "glows" as long as it is active.

Controls

- Move Mouse: Look Left/Right/Up/Down
- Mouse Left: Shoot
- Mouse Right: Throw Bomb (hold to increase velocity)/Pick up objects
- Scroll Up/Down: Zoom In/Out
- F1: Toggle Wireframe
- F2: Toggle Backface Culling (Results visible with wireframe)
- F3: Toggle Debug Info (FPS etc.)
- F4: Toggle HUD
- F5: Toggle Bloom
- F6: Toggle Normal Mapping
- F7: Toggle Mouse Cursor
- N: Start game (when in 'waiting room')

References/Resources

Libraries used

- **Basics (OpenGL, textures, lighting, shaders, input handling):** ECG Framework
We used the updated version (ca. march 2020) of the framework for the "Introduction to Computer Graphics (ECG)" course as a starting point for our game. The framework includes libraries for using OpenGL and handling user input (GLEW and GLFW) as well as some utility/helper functions for loading textures and shaders.
- **Physics (Collision Detection, Ray-Tracing for Shooting):** Bullet 2.81
<https://github.com/bulletphysics/bullet3>
- **Font Rendering, Graphics Overlay:** freetype2 (<https://www.freetype.org/index.html>)
- **Model Loading:** assimp <https://www.assimp.org/>

Assets

- "Target Shooting" Font: <https://www.dafont.com/de/search.php?q=target+shooting>
- "Nanosuit" Model:
<https://sketchfab.com/3d-models/crysis-nano-suit-2-ca311b94a0c249a1abc6697d105253e5>
- Textures: [texturehaven](#), [cc0textures](#), [learnopengl](#)