CS488 Project Proposal

Title: First-Person Shooter Game Engine

Name: Zac Joffe

Student ID: 20711812

User ID: zmjoffe

Contents

1 Project Outline **3**
1.1 Purpose . . . 3
1.2 Background & Motivation . . . 3

2 Technical Outline **4**
2.1 Objective 1: Modeling the Scene . . . 4
2.2 Objective 2: UI . . . 4
2.3 Objective 3: Texture Mapping . . . 4
2.4 Objective 4: Particle System . . . 4
2.5 Objective 5: Synchronized Sound . . . 4
2.6 Objective 6: Static Collision Detection . . . 4
2.7 Objective 7: Dynamic Collision Detection . . . 5
2.8 Objective 8: Physics Engine . . . 5
2.9 Objective 9: Shadow Mapping . . . 5
2.10 Objective 10: Keyframe Animation . . . 5

References **6**

A Objectives **7**

1 Project Outline

1.1 Purpose

To create a 3D environment that can be explored via a first-person perspective.

1.2 Background & Motivation

I've always been interested in creating a 3D game engine using OpenGL. First-person shooters have been one of my favourite game genres for well over a decade [1]. This project is primarily focused on the graphics engine; the gameplay will be simple and serves as an interesting context for the implementation of the graphics component. The player controls a character equipped with a firearm in a first-person point of view. The gun can be used to shoot dynamically-placed enemies in the environment. The player can freely move about the world on the ground-level and look around the environment, just as one would expect from a first-person shooter.

This project is challenging since it employs a wide range of graphics concepts. Nearly all concepts in the pre-ray tracing part of the course will need to be employed, and implementation of most objectives are non-trivial. The project is also highly extensible; the ten objectives provide a solid foundation for further graphics and gameplay extensions. An example of possible extensions [2] include more complex scenes, improved physics modeling, and hierarchical modeling of enemies for variable damage based on hit location. The foundation of this game engine could also be used to implement clever enemy AI, such as that found in *F.E.A.R.* [1].

Through this project, I will learn how to create a basic 3D game engine which can be used to create a simple first-person shooter. It will also aid in further developing my understanding of the raster graphics pipeline taught in lectures and through assignments 1, 2, and 3.

[1] Standout titles of the genre include *Doom*, *Quake*, *Blood*, *Half-Life*, *Portal 2*, and *Metroid Prime*.

[2] I don't necessarily plan to implement any of these for the subjective marks. The point is to illustrate that the project has great expansion potential beyond the specification outlined by the ten objectives.

2 Technical Outline

This section outlines technical details associated with each of the ten objectives. See appendix A for the list of the ten objectives.

2.1 Objective 1: Modeling the Scene

This objective is about modeling the area that the player moves around in. This will be a simple scene with a flat, rectangular floor and four surrounding walls.

2.2 Objective 2: UI

The game will launch in a main menu that is visually distinct from normal gameplay. The menu will have a button that lets players start the game. Implementation of this feature can be done with a finite state machine containing two states — one for the game, and one for the main menu. This state machine can be implemented with the State design pattern by leveraging polymorphic rendering methods, as described in the Gang of Four design patterns chapter of *Game Programming Patterns* [2].

2.3 Objective 3: Texture Mapping

Texture mapping will be employed to give surfaces more detail and realistic appearances. The approach to implementing texture mapping in OpenGL largely the same as that first described by Blinn and Newell [3]. 2D texture rasters are loaded from disk onto the GPU. Texture coordinates u, v are passed as input to the vertex shader, which outputs it to the fragment shader. The fragment shader uses the input texture coordinates to determine the fragment's color. With the `GL_LINEAR` texture filtering option, OpenGL performs bilinear interpolation of the texels surrounding texture coordinates to calculate the color.

2.4 Objective 4: Particle System

A particle system will be implemented to model muzzle flash of the gun after gun it is fired. An implementation of particle systems is outlined in *Learn OpenGL: Learn Modern OpenGL Graphics Programming in a Step-by-step Fashion.* [4]. Particles are implemented using a particle emitter, which spawns a myriad of particles stochastically in a region with given colors, lifetimes, and velocities. Particles are killed after reaching a certain height in the world or after a set amount of time.

2.5 Objective 5: Synchronized Sound

Sounds will be played to give feedback for specific player actions. For instance, the action of firing the gun may be accompanied by a synchronized gunshot noise. These sounds are implemented as sound files on disk that are loaded into the game engine. Many C++ high-level audio libraries could be utilized to play these sounds, such as `SDL_mixer` and `SoLoud`.

2.6 Objective 6: Static Collision Detection

Static collision detection will be employed to prevent the player from moving out of the bounds defined by scene. The player's location will be surrounded by an invisible axis-aligned bounding box (AABB). The size of this box will be make to enclose the imaginary player character model. All collidable objects will have an AABB. Upon player input for movement, the updated position of the player will be calculated. Then, prior to rendering the update on said frame, every collidable object in the scene will be tested for collision with the player by testing for intersection of AABBs. If a collision occurs, player locations are not updated to the updated value that was calculated prior to the collision tests.

2.7 Objective 7: Dynamic Collision Detection

Enemies will move around the scene dynamically, thus a dynamic collision detection algorithm is necessary to determine if a bullet shot by the player hits the enemy. The implementation of this will be done with hitscanning, which is very similar to the ray casting algorithm as described in the ray tracing chapter of *Fundamentals of Computer Graphics* [5]. When the player shoots their gun, a ray will be casted from the camera in the direction of the gunshot. Each enemy will have an AABB, similar to the static scene geometry in objective 6, which will be used to compute ray-enemy intersection. The nearest enemy hit will then take damage in some visible way.

2.8 Objective 8: Physics Engine

The physics engine will govern player movement with its simulation of friction and gravity. When a player releases a movement key, friction will cause the character to slow to a stop rather than stopping instantaneously. In classical mechanics, friction can be modeled as a function of material properties (coefficient of friction) and the normal force of the moving object. This model is simplified to a function that reduces the magnitude of the velocity linearly to 0 at a convincing rate.

Gravity simulation will allow for realistic jumping mechanics. In classical mechanics, gravity (on earth) is represented as a constant force acting down on an object at a rate proportional to its mass. This model is simplified by ignoring the mass of the player (or rather setting the gravitation acceleration to account for the player's mass accordingly). Upon jumping, the player's height will increase by some initial velocity. The force of gravity will cause this velocity to decrease over time by a constant acceleration value. This will cause the player to rise and fall in the same way one would if they were to jump in the real world.

2.9 Objective 9: Shadow Mapping

Shadows are implemented in the world using shadow mapping. This algorithm was first described by Williams in 1978 [6], and involves a two-pass rendered where the world is rendered from the light's point of view, then rendered from the camera's point of view. The second rendering pass uses z-buffer values computed from the first rendering pass to determine if a pixel is visible from the light source, and draws shadows accordingly.

2.10 Objective 10: Keyframe Animation

Players will be able to see a model of the gun they wield in their first-person perspective. This gun will be animated using keyframes, where linear interpolation is used to render the intermediate frames between keyframes. An example of gun animation is recoil after the gun is shot; another effective way to give visual feedback to the player after shooting.

References

[1] Jeff Orkin. *Three States and a Plan: The A.I. of F.E.A.R.* 2006. URL: https://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf.

[2] Robert Nystrom. *Game Programming Patterns*. State. Chap. II. URL: https://gameprogrammingpatterns.com/state.html.

[3] Martin E. Newell James F. Blinn. "Texture and reflection in computer generated images". In: *Communications of the ACM* 19.10 (Oct. 1976), pp. 542–547. URL: https://doi.org/10.1145/360349.360353.

[4] Joey de Vries. *Learn OpenGL: Learn Modern OpenGL Graphics Programming in a Step-by-step Fashion.* Kendall & Welling, 2020.

[5] Peter Shirley Steve Marschner. *Fundamentals of Computer Graphics*. 2016. Chap. 4.

[6] Lance Williams. "Casting curved shadows on curved surfaces". In: *ACM SIGGRAPH Computer Graphics* 12.3 (1978), pp. 270–274. URL: https://doi.org/10.1145/965139.807402.

A Objectives

1: Model the scene for the player to move around in.

2: Main menu user interface that the player interacts with to start the game.

3: Texture mapping.

4: Particle system.

5: Synchronized sound corresponding to player actions.

6: Static collision detection of surrounding environment.

7: Dynamic collision detection of bullets fired by the player with enemies in the environment.

8: Physics engine with friction and gravity.

9: Shadows using shadow mapping.

10: Keyframe animation using linear interpolation.