MONSTER SLAYER

A top down shooter made in OpenGL

Sohan Anisetty

Problem statement

Development of a 3D top-down shooter, wherein the camera rotation is fixed and follows the player around. The objective is to shoot randomly spawning monsters around the player. The player has 3 degrees of freedom and can only move in a 3D world. The game also has a modular level building system as well as easy to tweak settings by the user. The user can design and create the levels by entering model details in a JSON file.

Game Rules:

- The user can move around the world using the WASD keys as well as jump by pressing the spacebar.
- The user can aim the gun using the mouse and shoot using the left mouse button.
- The monsters spawn randomly around you in a fixed radius at fixed intervals.
- Killing a monster gives you 10 points and colliding with a monster results your score to 0.
- All parameters can be easily changed in the settings file.
- You need to gain 50 points under 1 minute to win. These can changed in the settings file.

Implementation overview

I have used a completely OOP approach and used concepts like inheritance, polymorphism, abstraction and encapsulation. The code is divided into the following classes:

Core mechanics:

1. Window
2. Shader
3. Camera
4. Settings
5. GameLevel
6. Game

Model Loading and rendering:

1. Mesh
2. Model
3. Animation
4. Texture
5. TextRenderer
6. SpriteRenderer

GameObjects:

1. GameObject
2. Player
3. Enemy
4. Bullet
5. Terrain

Core Mechanics

1. **Window**: The window class handles creation and destruction of windows and its resizing, initialising GLFW and GLAD as well handling user input from keyboard, mouse and recording cursor position.
2. **Shader**: Reads vertex and fragment shader information from files, initialises and compiles shaders, as well as links them to a program. Handles setting and getting uniform variables in the shader.
3. **Camera:** Handles camera movement position and speed as well as well as projection models
4. **Settings** : Reads settings from a JSON file and converts it into a form usable by the rest of the code.
5. **GameLevel**: Handles reading of environment information from JSON file and loading them into memory. It also handles rendering of the objects in the world.
6. **Game:** Initialises all shaders, models, level objects, textures, animations, lighting information, game state, score and spawns enemies at regular intervals. Handles collision checks between bullet and enemy as well as between player and environment. Handles bullets and enemy information currently on the screen.

Model Loading and rendering

1. **Mesh:** Converts vertice information (position, normals, texture coordinates), face indices, and textures to VAO and Texture units. Handles drawing these objects to the screen and destroying them when not needed.
2. **Model**: Uses the **Assimp** library to read a 3D model made in modelling software. It loads vertex information, normals, texture coordinates, texture maps, animation bone information. It then creates a mesh object using this information.
3. **Animation:** Handles reading bone data and animation key frames and interpolates between them.
4. **Texture:** Uses the **stb_image** library to load a image and convert it to the OpenGL texture format. Handles creation of mipmaps and tilling.
5. **TextRenderer:** Loads a font and converts it to OpenGL textures for each character using the **FreeType** library. Takes in text as input and renders texture version of the text on screen.
6. **SpriteRenderer:** Handles rendering of 2D objects on the screen such as the crosshair. Maintains coherency with the 3D objects on the screen.

GameObjects

1. **GameObject**: Base class for all 3D objects. Stores position, rotation, scale, bounding box, speed, direction vectors, mesh and collision check information. Responsible for updating object information every frame.
2. **Player:** Derived from GameObject class. Stores additional information about turn and jump speed, processing keyboard and mouse command to movements, environment collision detection, and updating height based on terrain.
3. **Enemy:** Derived from GameObject class. Handles player follow movement and health.
4. **Bullet:** Derived from GameObject class. Stores additional information about enemy collision, range, and movement direction.
5. **Terrain:** Handles creation of 3D mesh from heightmaps. Heightmaps are grayscale images where each pixel value corresponds to a height. Uses barycentric interpolation to figure out the height of the terrain at any coordinate on the terrain.

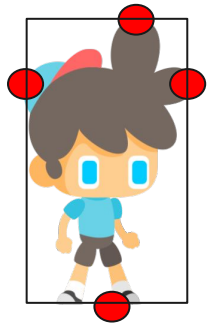
Notable achievements of your work

1. Implemented a modular level building system from a file.
2. Easy to tweak settings from the JSON file.
3. Heightmap and AABB based environmental collision detection. Ability to easily arrange terrain tiles (a combination of mesh and heightmap) to expand the level.
4. Modularised all components for easy debugging and reuse of functionality. Abstraction of complex model importing, linking textures, animation and VAO linking code.
5. Separating animations from base mesh. Doing so allows us to save only bone information instead of entire mesh. It also allows to enable/disable animations in runtime.

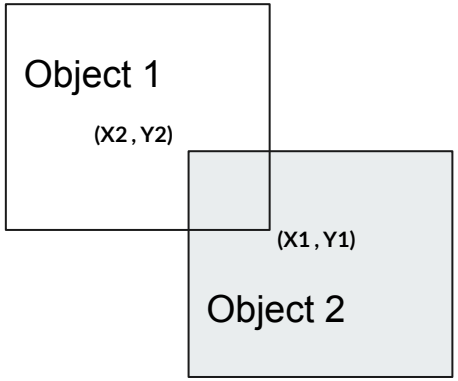
Collision detection

I have implemented 2 different types of collision detection techniques. One is the AABB-AABB collision detection and the other utilizes the heightmap.

AABB collision detection: AABB stands for axis-aligned bounding boxes. The bounding box information for each mesh is computed during loading of the mesh. The bounding box vertices are the maximum and minimum x,y and z coordinates of the vertices in the mesh. A collision is detected when all the axis overlap as shown below.



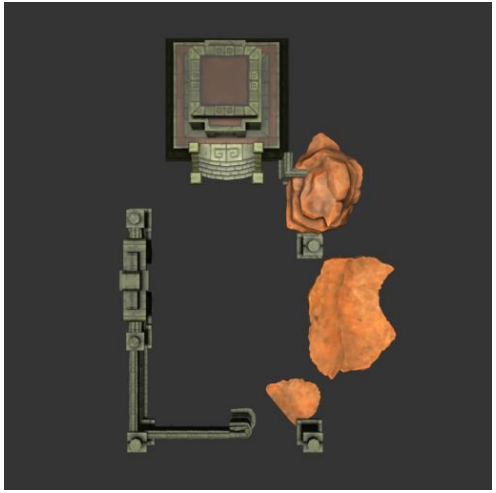
Bounding box computation

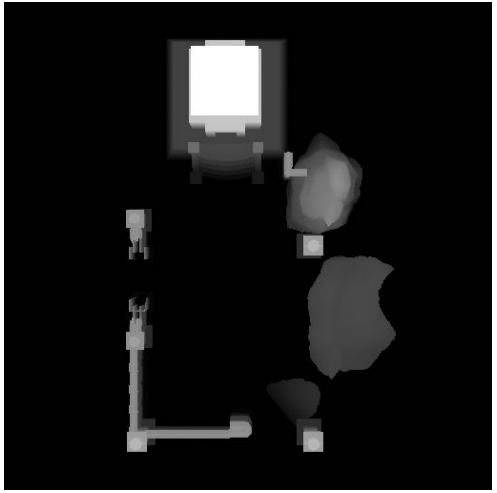


Collision detection

Heightmap collision detection: The previous approach can be used for collision detection between simple objects but can not be used for terrain as it cannot represent the undulations of the terrain with a single bounding box. Increasing the number of bounding boxes is not a viable option and doesn't work in doors and arches. Thus I opted to create a heightmap of the environment using Blender and that is used to determine player height in the environment.

Current altitude of the player is equal to the interpolated height from the heightmap. When the user moves the player I check the gradient between new position and current position. If the gradient is too high the player doesn't move otherwise the player climbs to the new position. However the player can attempt to jump to the new position.





Notable challenges experienced in the course projects

1. Structuring the game was a challenge. Once the scale started increasing I had to rewrite the code into separate classes and connect them using objects. Pointer memory leaks were an issue during this stage.
2. Animation using OpenGL was a challenge as it required knowledge about bone based animations and required complex interpolations between keyframes and synchronisation as well as quaternions for rotation interpolation.
3. Collision detection using AABB was not working for complex terrain as well as for doorways and other similar structures. After trying various other methods I decided to devise my own and built the heightmap based approach.
4. Syncing the 2D components with 3D objects was a challenge as well, due to 2D using orthographic projection while 3D uses perspective projection and a view matrix.

Areas of further improvements

1. **Multi threading**: Currently model rendering, collision detection between all objects and user input processing all happen sequentially in every loop. Distributing these processes to multiple threads would make it more efficient and faster.
2. **Adaptive rendering**: Currently all models part of the scene are loaded into memory and placed on the 3D environment even if it not currently seen in the camera. This takes up a lot of memory and computation power when more objects are added. Thus, only rendering those objects visible in the current field of view should be loaded into memory.
3. **Smarter collision detection**: In every loop collision detection checks happen between the player and all objects even if the object is far away from the player. This wastes resources. Thus, only detecting collisions between the nearest objects would be optimal.
4. **Advanced effects**: Effects like explosion and gunfire would enhance the gameplay. Similarly sound would make the game more enriching and interesting.