Tank Combat Game: OpenGL Implementation

Cosmina Mihoreanu

POLITEHNICA University of Bucharest

In this paper I will describe my approach to a project assignment for my second year Computer Graphics course. I implemented a simple World Of Tanks-like game in C++, over the OpenGL graphics framework. In this game, the player controls its own tank through a randomly generated landscape of buildings and enemy tanks. The goal is to shoot down as many enemy tanks as possible in the allotted time, while avoiding getting hit back.

1 Game Logic

In this section I aim to lay out the core logic of the game, to set the scene for further detailing of the implementation. The game starts with the player (pink tank) at the center of the game window (see fig.1). The surrounding landscape consists of flat terrain, buildings (lime green structures) and enemies (red tanks).

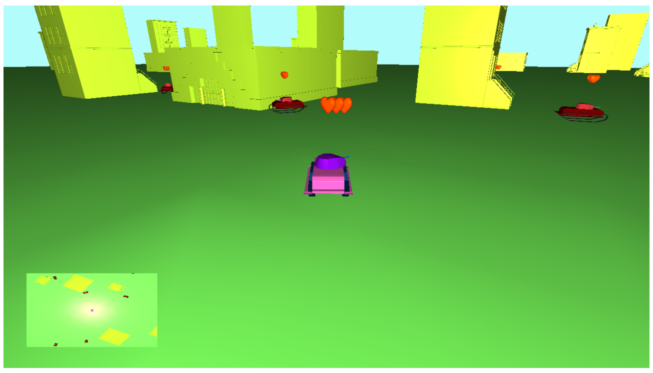
fig 1: Starting Game View

The player tank can move around the map freely, but it cannot pass through other objects. All tanks have 3 HP (Health Points) initially. By moving the mouse left to right, the player can aim the tank's turret and shoot cannonballs by clicking the left button of the mouse. Cannonballs cannot pass through buildings, and have a limited lifespan. When hit by a cannonball, an enemy tank loses one HP. If left with no HP left, a tank remains static for the rest of the game. When the player gets close to an enemy tank, the latter aims its turret at the former and fires cannonballs. If the player loses all HP, the game ends. If the player survives until the end of the game (initially set at the 2 minute mark; can be changed), the scene freezes and the console displays the final score (the number of defeated enemies).

2 Project Structure

I structured the code in 4 main object classes:

1. Homework2.cpp - main game logic (init, mesh loading, updates, rendering);
2. Tank.cpp - models a tank's properties and interactions with the game environment;
3. Building.cpp - models a building's properties;
4. Projectile.cpp - models a projectile's properties and behavior.

The camera.h file implements a simple third-person camera and its movements and rotations.

3 Blender Meshes

The meshes I used in this project are assembled in Blender 4.0, the open-source 3D creation tool. I imported several models as .obj files from free online databases like Free3D and then I centered and scaled them using Blender. I then loaded them into the program in the Init() function call directly from the asset folder. There are 3 different building meshes with varying sizes:

- storefront.obj;
- brownstone.obj;
- residence.obj.

The tank mesh is created from 4 different components imported separately and assembled in Blender:

- tracks.obj;
- body.obj;
- hatch.obj;

- cannon.onj.

The hearts representing HP above each tank are also imported 3D models:

- heart.obj.

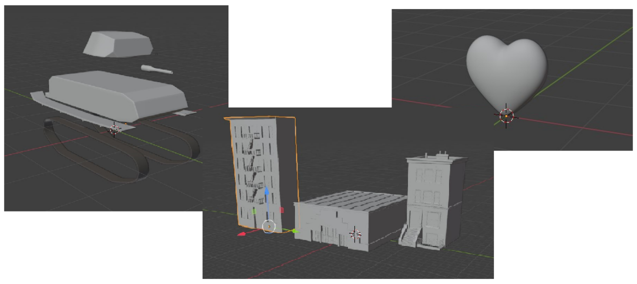

fig 2: Blender View Objects

4 Custom Shading

To add dimension to the game's graphics I wrote my own VertexShader and FragmentShader programs to run on the graphics processor. Given the many details of the chosen 3D models, I implemented a simple version of the Phong reflective model. Phong's model consists of 3 different lighting computations done for each vertex, representing:

- the ambient component: overall light in the scene (here I chose a constant value all over):

  $ambient\_light = 0.25$

- the diffuse component: light scattered by hitting surfaces:

  $diffuse\_light = Kd \times max(dot(N, L), 0)$

  where Kd = material diffusion constant

  N = surface normal

  L = light source direction

- the specular component: light reflected off a surface in a specific direction:

  $spec\_light = Ks \times pow(max(dot(V, R), 0), S)$

  where Ks = material specular constant

  V = visualization direction

  R = light reflection direction

  S = shine coefficient

The compound light on each vertex is simply the sum of all three components.

Another feature obtained through shader code is the progressive deformity and darkening of the tanks as they lose HP.

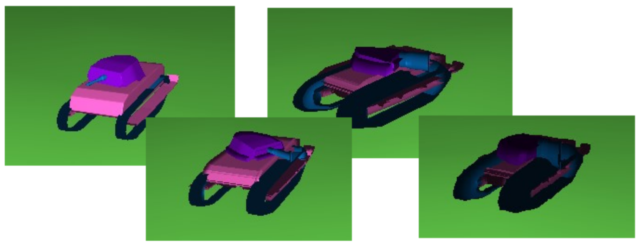

fig 3: Progression of Tank Shading Stages

More specifically, the render call sends the HP value of each object as a uniform value to the shader program (for objects where HP does not make sense, -1 is sent by default). This value is then used to compute a position displacement for each vertex (in the vertex shader), and a color displacement (in the fragment shader) based on the following functions:

- position:

  $delta = -0.1 \times damage \times position \times normal$

- color:

  $delta = -0.2 \times damage \times color$

The light source is positioned at the camera's coordinates for all computations.

5 Scene Generation

As previously stated in the first section of this paper, the playfield is randomized. The Homework2 class contains the **buildings** and **enemies** vectors, which are populated with the corresponding objects during the Init() phase. However, the number of scene elements is fixed.

- For enemy generation, their initial coordinates are assigned randomly within the boundaries of the world (modelled using the **planeLim** class variable).
- For building generation, there are 3 different randomized parameters: position, mesh type and orientation (north, south, east, west), as illustrated in the following code snippet, where the length and width of each type are defined in Blender:.

```
Mesh* mesh = nullptr;
switch (type) {
    case 0:
        mesh = meshes["brownstone"];
        length = 14; width = 14;
        break;
    case 1:
        mesh = meshes["storefront"];
        length = 20; width = 19;
        break;
        case 2:
                mesh = meshes["residence"];
                length = 13; width = 16;
}
int orientation = rand() % 4;
```

6 Controls

In this section I will enumerate the available game controls for the player tank:

Key	Movement
W	forward translation
S	backward translation
A	in-place left rotation
D	in-place right rotation

Mouse Event	Action
left to right moves	turret aiming
left click	firing cannonballs
right click and move	camera rotation

7 Cameras and Minimap

I used two different third-person cameras for this game: one for the player-perspective view, and one for the map view.

The player-perspective camera (the **camera** object) is initially set at a fixed distance behind the tank, and higher on the vertical axis. This camera enables a perspective view, so it has an associated projection matrix computed by glm::perspective().

Similarly, the **orthoCamera** is used to render the map at the bottom-left corner of the screen, and is initially placed right above the player at a fixed distance. It makes use of an orthogonal view, thus its projection matrix is generated by glm::ortho().

Whenever the tank moves, both cameras translate forward or backward with the same displacement, or rotate left to right with the same angle (around the vertical axis).

An extra feature of the player-perspective camera is its rotation around the tank, mentioned in the last section about game controls. Upon pressing the right button of the mouse, the left-to-right movement of the mouse rotates the camera around the stationary tank. This way, the player can then view its tank's movements from any angle of their choosing.

8 Enemy Behavior

The enemy tanks generated in the scene have a randomized movement implemented using a simple state machine logic. I chose 4 possible states, corresponding to the 4 types of movements a tank can do, and I modeled them as an Enum class:

```
enum class MovementState
{
        Forward,
        Backward,
        RotateLeft,
        RotateRight
};
```

Generating a new state for a tank means choosing a random rotation state (RotateLeft, RotateRight) if the previous state was a translation state (Forward, Backward) and vice versa, and setting a random time interval in which to execute that movements continuously. Rotation intervals can range from 0 to 2 seconds, while translation intervals span between 2 and 4 seconds.

At each frame, I do the following checks for all enemy tanks:

- if the current state's total interval has not yet been reached, I decrement it and apply the appropriate movement on the tank;
- if the current state's interval drops to 0, a new state is generated.

The enemy tanks can also attack the player back. When the distance between an enemy and the player tank decreases under 10 units, the enemy's turret rotates so that it aims at the player and it automatically shoots cannonballs at 4 second intervals. The player tank itself can fire every second.

9 Collisions

In this section I will describe the approach I took to in-game object collisions.

When 2 tanks come in contact (overlap), they will each be translated in the direction their centers define, away from each other. Each tank will be displaced with half of the overlap length, as demonstrated on the assignment page (see fig. 4). We check if two tanks collide by comparing the distance between their centers and the sum of their radii.

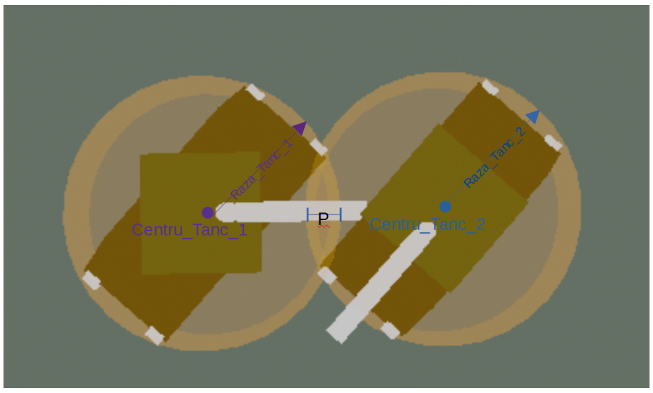

fig 4: Tank Collision

Similarly, when a tank collides with a building, i.e. when the margin of a tank enters the rectangle defined by the building's coordinates and size, the tank is translated directly backwards to avoid overlap.

10 Ending

The game ends by default after 2 minutes. This interval can be changed by manipulating the **gameTime** class variable from Homework2. This variable is decremented each frame with the time elapsed from the last frame. Once the game comes to its natural conclusion this way, the scene freezes and the only movement permitted is the rotation of the player's turret. The console shows the final score accumulated. If the player looses all its HP, no more movements are rendered, and the console prints the losing message.

To make sure these console messages are only printed once, and not for every frame encountered in the Update() function, each of them has an associated boolean variable (**gotScore**, **gotMessage**) initialised to false, and set to true once the message is printed once. The message is only printed if the corresponding variable is false.

The blocking of the moving elements on the scene once the game is over is done the same way, using the boolean class variable **gameOver**, which is false upon initialisation. When the player's HP gets to 0 or the game time has finished, **gameOver** becomes true. All coordinate updates for the tanks and fired projectiles are done only when **gameOver** is false.

11 Summary

To summarize, the tank combat game I implemented made use of several Computer Graphics concepts, such as 3D modelling and movement, importing/exporting meshes as assets, Object Oriented Programming and shading techniques. The code is available in a GitHub repository: https://github.com/cos-mih/world-of-tanks-dupe.

12 Resources

Assignment:
    https://ocw.cs.pub.ro/courses/egc/teme/2023/02
3D Models:
    https://free3d.com/3d-models/
Shader Basics:
    https://ocw.cs.pub.ro/courses/egc/laboratoare/07
State Machine for Random Movements:
    https://wandbox.org/permlink/y9DkfVMLZrhFhg4z