EEE2041: 3D Tank Game Assignment
Computer Vision and Graphics
Spring 2017

Ho Man Herman Yau

November 10, 2017

Abstract

This report outlines the objective of the 3D Tank Game Assignment, gives an overview of OpenGL and its supporting libraries and frameworks. Techniques involved in finishing the assignments are also discussed in subsequent sections. The finished game are demonstrated with screenshots.

1 Introduction

1.1 Objectives

The objective of this assignment is to develop a 3D 3rd person maze action game using Modern (Shader-based) OpenGL and FreeGLUT. The goal is to navigate a hamvee around a maze using keyboard a mouse input and collect all coins within time limit while avoiding falling off the edges.

1.2 Techniques Used

In general, the following techniques are used in the development of the game:

1. **Transformation** - Translation of entities by 4x4 Matrices.
2. **Texture and Object Importation** - Importing pre-existing objects and texture images from folders with functions provided.
3. **Pseudo-physics** - Implementation of simple physics using Newton's laws of motion with custom values.
4. **Lightings** - Implementation of Phong reflection model.
5. **Shaders** - Use of shaders for raw texture and object data processing, as well as colours, lighting and transformation computation.

2 Method

2.1 Shaders

Shaders are the basis of modern OpenGL programming. In a nutshell, shaders are pieces of program which are written in OpenGL Shading Language (GLSL), enabling direct access to the GPU by the programmer. This is known as programmable 'shader-based' graphics pipeline.

In this assignment, shaders can be divided into vertex shader and fragment shader. Vertex shader applies all kinds of transforms to 3D vertex positions, while fragment shader applies colour, shading and lighting to the output pixels. The function *glUniform* is used to pass uniforms into shaders for computation.

2.2 Maze Environment

Maze data for the program are represented by 10 x 9 integer matrix where 0 indicates an empty space, 1 indicates a cube surface, 2 indicates a cube with a target and 3 indicates a trap cube. The level data are read from text files with two different levels.

After the maze data are successfully stored into a a static 2D global integer array defined in the main program, it proceeds to terrain generation. Three functions are responsible for this task: *CubeRenderer*, *CoinRenderer* and *MapRenderer*.

- **void CubeRenderer(int x, int z)**
  This function generates one single cube and place it into the world in the X-Z plane. All cubes are represented by a separate Matrix4x4 constructor defined in *Matrix.cpp* and declared as *Matrix4x4 CubeMVMatrix*. They are translated to their appropriate position with respect to integer matrix in text files, which are then passed to the GPU for computation in shaders with *glUniform* function.

- **void CoinRenderer(int x, int z)**
  Likewise, this function generates one token and place it into the world in the X-Z plane. The tokens are represented by *Matrix4x4 CoinMatrix*, and are also translated to their appropriate position with respect to integer matrix in text files which are then passed to GPU.

- **void MapRenderer(int Maze[MAX_LENGTH][MAX_LENGTH])**
  This function is where the placement of the generated object takes place. It utilises two nested for-loops, each representing X and Z coordinates respectively. Inside the inner for loop, a switch case is used where *CubeRenderer* and *CoinRenderer* are called to place the objects at the designated coordinates corresponding to the level's integer matrix.

  Note: The maze is scaled 8 times larger to allow higher degree of freedom for tank movement.

2.3 Entities

Data for all movable entities are encapsulated into a structure named *Entity* in the header file. The structure contains the following parameters:

Parameters	Descriptions
PosX	X-coordinate position of the entity
PosY	Y-coordinate position of the entity
PosZ	Z-coordinate position of the entity
VelX	X-coordinate velocity of the entity
VelY	Y-coordinate velocity of the entity
VelZ	Z-coordinate velocity of the entity
Rotation	Angle of rotation in degrees of the entity

The sections below will discuss methods used to load object files and textures, as well as movement for separate models implemented in the program.

2.3.1 Models

The loading of models for the program can be broken down into two parts: **Object Loading** and **Texture Loading**.

- **Object Loading**
  Object loading is done by one of the provided C++ codes in lab exercise: *Mesh.cpp*. The code contains *Mesh* class which has a method to read OBJ file format which contains vertex coordinates, vertex texture coordinates (optional) and vertex normals with ID numbers in front to distinguish each face. To declare a new object, *Mesh* is declared with an object name. The method *LoadOBJ* is then called to read the OBJ file that the program point

towards. In this assignment, we have a total of seven *Mesh*: cube, chassis, FrontWheel, BackWheel, coin, turret and ball.
To draw the object, simply call the *Draw* function in the *Mesh* class.

- **Texture Loading**
  Similarly, texture loading is done by another provided C++ codes: *Texture.cpp*. The code contains *Texture* class which has a method to load a bitmap (BMP) image pointed by the program upon calling the function. The textures are defined in an integer array **GLUint Texture[4]**, with hamvee.bmp, Crate.bmp, coin.bmp and ball.bmp loaded into the array respectively.

  Activating texture mapping is slightly more complicated. *glEnable(GL_TEXTURE_2D)* has to be called to enable texture mapping in OpenGL. The desired texture is then bound to an object by *glBindTexture* in which we specify the dimension of the texture and the assigned texture ID with respect to it.

2.3.2 Tank Movement

The tank is able to move in all directions along the X-Z plane using keyboard keys, as well as rotating its orientation according to direction of movement. To achieve acceleration and gravitation, pseudo-physics is implemented throughout the program, following physics equations in general.
The physics parameter of the tank is determined by *Entity* structure *TankInfo* and the transformation are represented and declared by *Matrix4x4 TankMVMatrix*. It can be controlled by W, A, S, D where W, S are forward and backward movement respectively and A, D are rotation movement.

Rotation for the tank is easy done by calling *Rotate* function defined in *Matrix.cpp* along the Y-axis of the it. Velocity of the tank is given by

$$v_{new} = v_{prev} + \Delta t \tag{1}$$

where v_{new} is the magnitude of new velocity;
v_{prev} is the magnitude of previous velocity;
Δt is change of time.
Change of time and new physics parameters can be obtained from the *glutTimer* function which is called every 10 millisecond by the program. The X and Z component of the tank's velocity are then calculated by simple trigonometry, where the the components are given by

$$v_x = v \times \sin\theta \tag{2}$$

$$v_z = v \times \cos\theta \tag{3}$$

where θ is the angle of rotation by the tank in radian.
Then, the position of the tank can be updated with the equation below:

$$P_{new} = P_{prev} + v_{new} \tag{4}$$

where P_{new} is the new position of the tank, P_{prev} is the old position of the tank. Likewise, the position can be broken down into X and Z position and the same formula applies with velocity. If the tank is out of boundary , it falls with gravity with the equation

$$v_{new_y} = v_{old_y} + \Delta t \cdot G \tag{5}$$

where G is the gravity constant defined in the program.

2.3.3 Turret Movement

The turret of the tank is a constituent part of the tank. It is allocated to another *Entity* structure *TurretInfo* and a separate Matrix4x4 *TurretMVMatrix* to allow independent movement and control from the main body of the tank. As you may have guessed, the position of the turret can be directly copied from *TankInfo* since it simultaneously moves and is stationary relative to its body. The mechanism of rotation for the turret is the same as above for the tank. It rotates in the X-Z plane along the Y-axis by R and T, where R rotates the turret to the left and T to the right.

In addition to independent rotation, the turret is also capable of shooting projectiles, which can be shot by pressing spacebar. The projectiles have a constant velocity of 50 defined as *PROJ_VEL*, and are affected by gravity using the equation (1) and (4).

2.4 Camera and Text Display

2.4.1 Camera

The camera is programmed such that it moves around the maze from a fixed distance from the tank, with its rotation following the turret's orientation. The position of the camera can be calculated by

$$x_{new} = x_{prev} - d \times \sin\theta \tag{6}$$

$$z_{new} = z_{prev} - d \times \sin\theta \tag{7}$$

where x, z are the X and Z positions of the tank respectively;
θ is the angle of rotation of the turret;
d is the magnitude of distance, which is an incorporated function given in *Vector.cpp* and calculated by

$$d = \sqrt{x^2 + y^2 + z^2} \tag{8}$$

2.4.2 Text Display

2D text display for information are implemented using *render2dText* function. The function is called after shadesr are unused as it does not require GPU computation. It renders 2D text into the window with colours specified with R, G, B parameters and coordinates specified in the range of -1 and 1.

2.5 Collision Detection

Collision detection is achieved by tracking real-time position of an entity with the array index of the maze with the aid of *glutTimerFunc*. After trial-and-error and some derivations, the equation to achieve it is found to be

$$X = \frac{P_x + s}{L} \tag{9}$$

$$Z = \frac{P_z + s}{L} \tag{10}$$

where X, Z are the array index of the maze;
P_x and P_z are the position of the concerned entity;
s is the scale factor of the maze;
L is the length of each crate after scaling. For projectiles, an extra Y-axis check is implemented so that collision detection with tokens activates only when they are positioned above surface, i.e. $y > 0$.

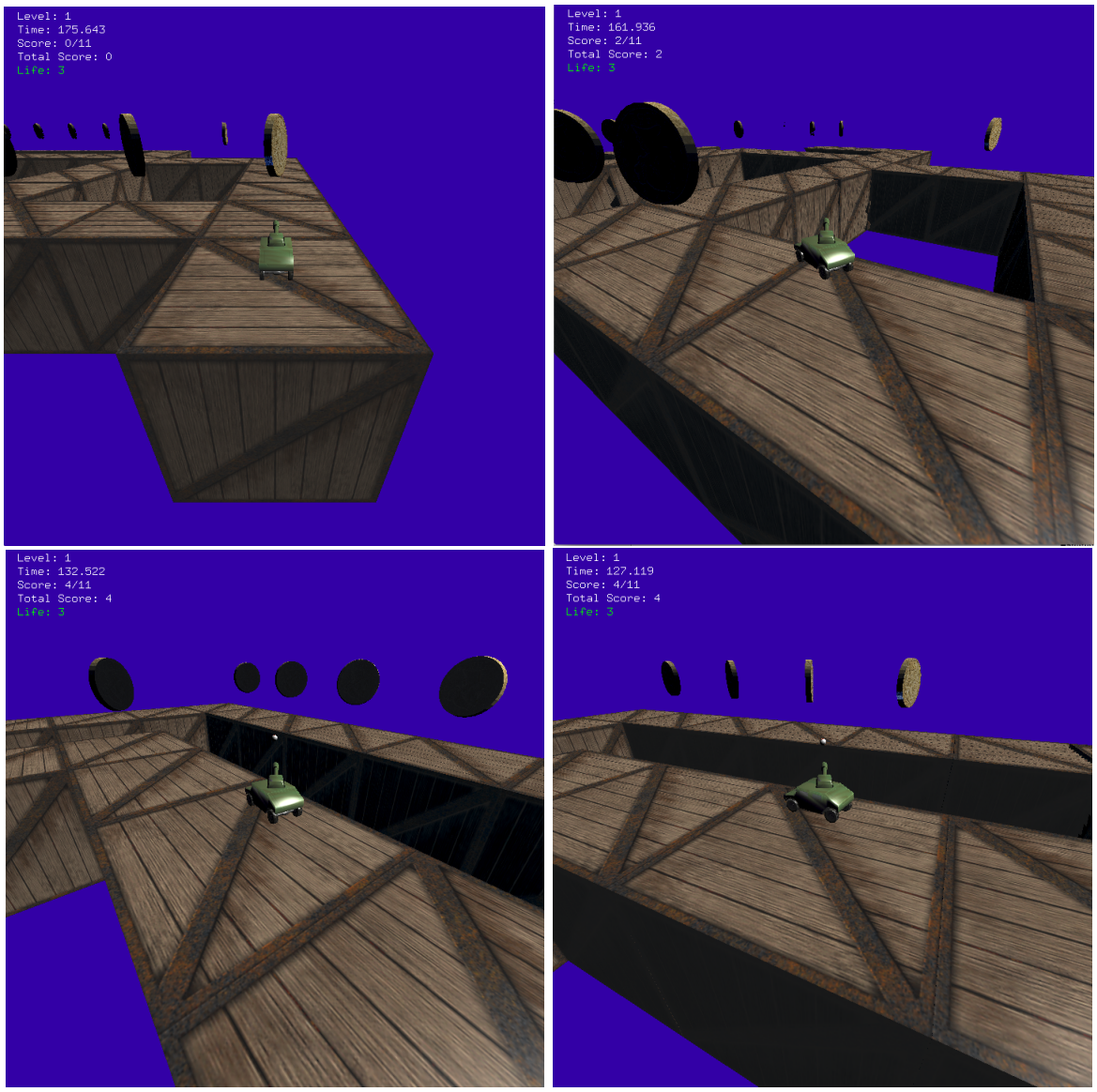
2.6 Lighting

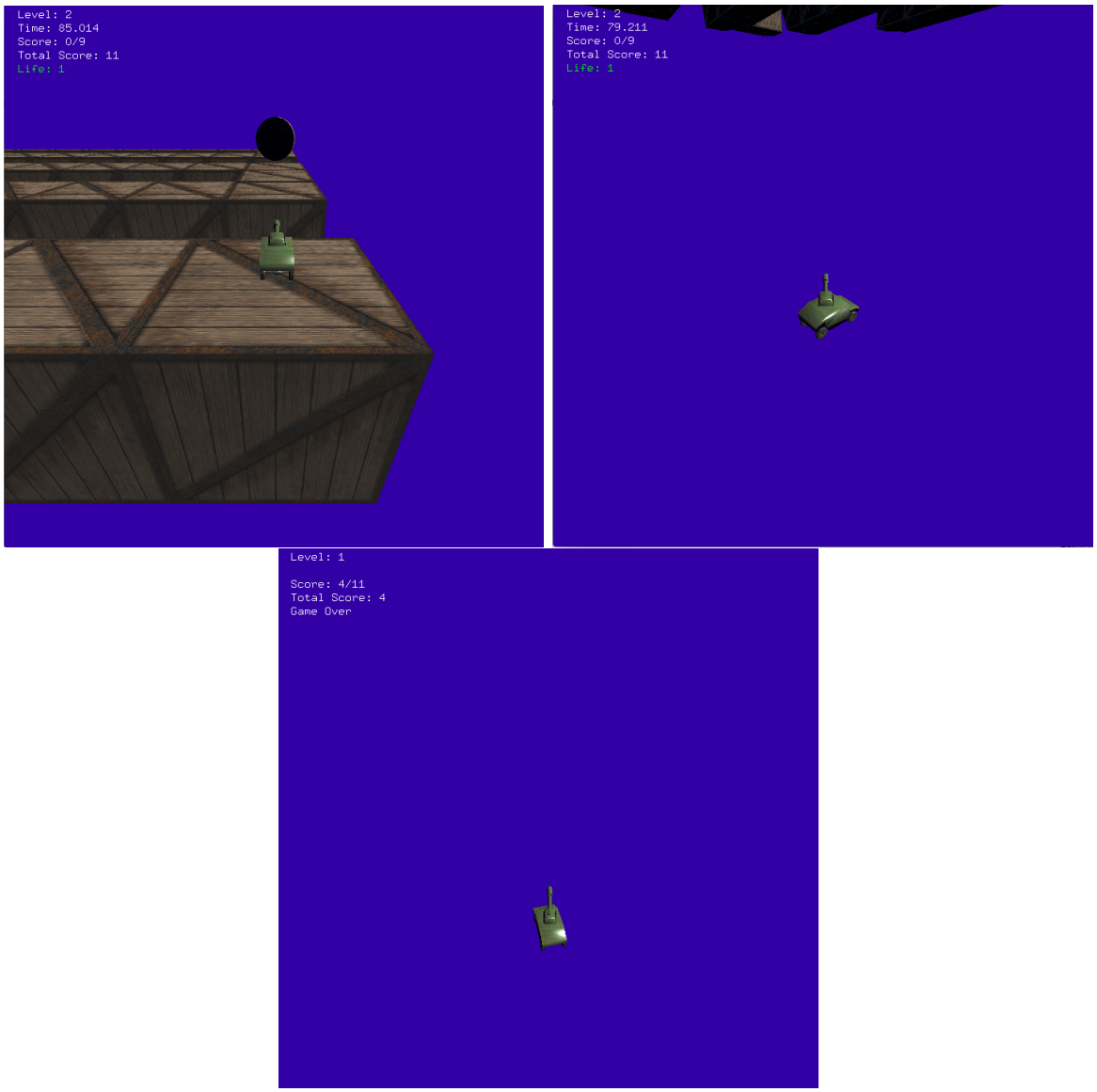
Lightings are mainly implemented and computed in vertex shader and fragment shader instead of the main program in *main.cpp*. In this assignment, Phong reflection model is used. It describes how a surface reflects light using a combination of ambient, diffuse and specular models to simulate a realistic light in software. In addition, vertex normals of object is needed such that shaders can compute the angle between the surface and light source.

The parameters for lighting are specified in *main.cpp*, which defines light position, ambient reflection, diffuse reflection, specular reflection and specular light coefficient.

3 Result

The finished game is demonstrated below with screenshots.





4 Conclusions

This assignment gives an insight into how the OpenGL API can be used to model shapes with polygons, controls of lighting, shading and rendering of objects and textures. Moreover, the mathematics involved in translations and camera positioning are extensively used in the program, which contributes to understanding of matrix math for these operations, as well as methods to model physics in software. Last but not least, it improves proficiency and provides logical thinking in software development.