A5 Project Documentation
Title: Tank War
Name: Chenyang Zhang
Student ID: 20756699
User ID: c529zhan

1 Purpose

The purpose of the project is to create a 2.5D computer game. In the game, I will implement the following: alpha blending and texture mapping are applied to models; L-system plants are used; shadow volumes are applied to the scene, animations and partical systems are applied to some models.

2 Statement

The main idea of the game is that the player need to control a tank to defend their base and to eliminate all enemy tank. The map will be created by a rectange grid. There are destroyable blocks (destroyed when there is a shell hit) and non-destroyable blocks. The player should be able to move the tank and fire tank shells in 4 directions: front, back, left, right. Enemy tanks also move and fire in those 4 directions. When all enemy tanks are eliminated, the player wins. When the base is destroyed, the player losses.

The first thing to do is to create models for every objects in the scene. The game logic should be implemented. In this part, the data structure of the grid, objects, and the tanks should be created. Firing system and shell impact system need to be designed. Finally, enemy logic should be implemented. For the graphics parts, textures and alpha blending are used on models with shadow effects and particle effects. Skybox for the background is another objective to do.

The game itself should be fun to play with. The enemy will have some intelligence so that it will have some difficulty to play with. The scene contains many objects that make it interesting, such as trees and blocks where a player can see through but can not shoot through. Shadows and textures will make the scene more realistic. Some animations are made, so the actions of the tank seem smoothly. There are some background music and sound effects in this game to provide better gaming experience. There are a few challenges in this project. The particle systems are challenging since the movement patterns of small particles need be designed. How to generate a L-system tree and apply shadow volumes are also difficult.

In this project, there will be some very useful graphic techniques for games, such as shadow volumes, L-system plants, texture mapping, alpha blending, and keyframe systems. Implementing those features will help me have a better understanding of game graphics. I can also learn the process (planning, implementing, and testing) for game development. It can potentially help me to find jobs in the game industry.

3 Technical Outline

The models that need to be implemented are tanks, the base, trees, and blocks. I plan to implement several different type of blocks: destroyable blocks, not destroyable blocks, and semi-transparent blocks. I plan to use lua to create models. The basic shapes should be cube, sphere, and cylinder. The tanks are likely to be made with hierarchical models.

I plan to use the bracketed OL-system, as described by Prusinkiewicz and Lindenmayer (1990), to construct tree trunks. It is an improved L-system that introduces 2 symbols, '[' and ']', to delimit a branch. The symbols between [and] represents a branch. When '[' is passed in, then the current state of the node (position, orientation, width) will be saved to a stack. When ']' is passed in, pop the stack and load that information to the current state.

Shadow volumes will be calculated for the tank, potentially for the trees and the blocks. I plan to use directional light source to simplify the process. The ray will simply has a directional vector, and there will be only 1 light source. The implementation should be similar as described in the lecture slides, i.e. use the

stencil buffer

For texture mapping, I plan to implement it only on floors and blocks. Hence very likely, there is no texture mapping applied to cylindrical or spherical objects. I plan to use simple small patterns to start with, and repeat the patterns. For triangle mesh, I will extend it to a rectange and map the texture to each coordinates. OpenGL has some library that can help me to do this part.

For alpha blending, I will linearly combine the color behind the object C_b with the color of the object C_o with opacity α. The algorithm is simply

$$C_r = C_o * \alpha + C_b * (1 - \alpha)$$

The thickness of the object is not considered. I plan to apply alpha blending to some of the blocks, leaf balls of the trees, and particles.

I plan to implement a particle Class to store information of the a particle. Position, velocity, and acceleration is stored in the class. I will potentially store the scale information as well, so that I can reduce the particle as time passes. There should also be a particle effect Class to store the particles together, initialize particles based on some pattern, such as moving away from a coordinate. Lifespan should be included in that class. The particle effect object will be deleted after time passes its lifespan. Particle effects will be applied on tank destruction.

For skybox, I plan to extend the map and apply texture mapping on it. Then I will put some background images on the sides far away from the map.

Animation will be implemented with interpolated keyframes. It will be applied only on the tanks: tank movement, tank self rotation, and tank firing a shell. An action Class should be implemented. The locations (vec3), and rotations (vec3) of the action is stored seperately. The time for how long the action will take and the start time are stored as well. The locations and rotations are linearly interpolated. For example, the location L_{now} at time t, given start time s, duration d, initial location L_i, final location L_f, is calculated as the following:

$$L_{now} = \frac{t-s}{d} * L_f + (1 - \frac{t-s}{d}) * L_i$$

The calculation for rotation is the same as above.

For an animation, there can be multiple actions. An animation Class stores a list of actions and decides which action to perform based on time delta.

In enemy logic part, how the enemies move, when an enemy shoots, and when to spawn enemies will be implemented. The enemy will have tedency to move toward the player's base.

Some suitable background music is chosen for this game. There could be sound effects on several places: firing shells, shells on impact, tanks on destruction, and game over scenes.

4 Manual

In this section, I will be describing how to run the program. First please follow the readme file to compile the program. Simply do './A5' will run this program. No arguments should be supplied.

There will be no useful command line outputs.

For game interaction:

- Pressing up/down/left/right key on your keyboard will move the tank. If the tank's current orientation is different from the direction you want it to move, the tank will change its orientation but it won't move. You can only move once in a small time period.

- Pressing Space key will fire a bullet. You can only fire once in a small time period.

NOTE: Before pressing start or S, you can not move or fire, and enemy will not be generated.

- Each enemy and destoryable walls has 2 hp. The player has 4 hp. The puppet has 1 hp. If a bullet hits a tank/the puppet, the tank/the puppet decrements its hp by 1. Enemy has NO friendly fire demage (but you can attack your puppet)

- If your tank has 0 hp or the puppet has 0 hp, you loss.

For UI interaction:

- Pressing Quit Application button or Q will quit the program.

- Pressing Reset button or R will reset the view and the game. (will not start the game)

- Pressing retry button or N will reset the map and put the player to default position. (will not start the game)

- Pressing start button or S will start the game.

- Toggle Show Shadow will draw shadows. Default value for it is true.

- Toggle Show Shadow Volume will draw the shadow volume in green (only if Show Shadow is toggled). Default value is false.

- Toggle Texture mapping will map textures on blocks and the floor. NOTE: textures for skybox cannot be disabled by this checkbox.

- Drag using the left mouse will rotate the view.

- Scroll the middle mouse will zoom in and out.

- You can change the volume by sliding the volume bar.

- You can change the opacity of the tree leaves to see the tree trunks by sliding the opacity bar.

5 Implementation

For the game code structures, I tried to follow a standard OOP design. A base class *Entity* is designed. Position and health are stored. Every game object (Blocks, Tanks, Enemies, and Shells) is extended from the base class. *World* is the class to store all the game entities and update every entity for each frame. A *EnemySpawner* is designed to spawn enemy tanks

For L-system tree trunks, there is a *Grammar* class that will generate a string that can be used to generate the tree based on given rules. For each iteration, it will randomly select a rule and write to the string. Then when generating the trees (in class *LSystem*), I keep track of the transformation matrix of the current state for the turtle. There is a stack to store the states for the turtle. When the turtle sees a character that performs rotation, the transformation matrix will be rotated around the given axis. If it sees character 'F', it will generate a new transformation matrix that can be used to render the tree trunk and store that transformation matrix. If it sees character '[', it will push the current state to a stack, and calculate the new scale and length for the following tree trunks. If is sees character ']', then it will pop from the top of the stack to its current state. After generating the tree trunks, all transformation matrices are stored for further rendering.

For shadow volume, I implemented it with the zPass algorithm. First draw the scene with only ambient light (with depth information). Then draw the shadow volumes with Stencil test on (with no depth test, doesn't draw to color buffer or depth buffer). Then enable back-face culling, increment the stencil counter if passed. Enable front-face culling, decrement the stencil counter if passed. Then set the stencil function to test for equality to 0. Finally, disable stencil test and render the scene again. For drawing the shadow volumes, I implement a geometry shader to do it (if there is a gpu, then it will be generated in the gpu) instead of generating vertice and send them to the vertex shader.In the geometry shader, when a triangle (3 vertices) is pass in from the vertex shader, it will generate the 6 faces of the shadow volumes using triangle strips, and pass that into the fragment shader.

For particle systems, I implement one to simulate explosions. For each particle, position, direction, rotation, scale, and lifespan are stored (in class *Particle*). For each frame, the particle moves toward the direction with a random direction vector to simulate random movement. The scale of the particle will be smaller each time it updates. To generate a explosion effect, particles with directions that are uniformly distributed to a sphere are generated in class *ParticleSystem*. Scale and lifespan of the particle are uniformly distributed as well. If a particle reaches its life span, it will be deleted. If all particles are deleted for a particle system, then the particle system is deleted. *World* can generate a particle effect on a specific position.

This program uses 'stb_image.h' to load a image resource. A function is written to load a texture from a file path. The return value is the id of the texture that opengl can use to load the texture to a shader.

This program uses irrKlang libraries to load sound file and play sound. This include using a static library 'libIrrKlang.so' (for Linux) and 'libIrrKlang.dll' (for Windows).

NOTE: the zoom in/out interaction is eventually move the model towards/backwards the viewer. Hence there is some slight changes in the shadows.

For how the code is structure:

'Assets/' folder stores the shaders, .obj files, .lua files (models), sound files, and the images for texture mapping.

'common/' folder stores files that will be used to construct scene nodes from lua files, which we used in A3. It also includes a 'common.hpp' for some common functions that I used.

In 'gameobj/' folder, classes for game objects are defined and implemented.

In 'tree/' folder, there is an implementation of a *Grammar* class that generate a string for a L-System and a *LSystem* class used to generate tree trunks.

In 'sound.hpp', a sound player is implemented upon the irrKlang library to play bgm and sound effects.

In 'particle.hpp/.cpp', the particle system is implemented.

The code is written based on the code I wrote in A1.

6 Credit

This section describes resources I used in this project.

Credit to Nathan Ostgard for 'cube.obj' file with texture coordinate info.

Credit to Luke.RUSTLTD for his skybox textures.

Credit to IrrKlang for the sound library and a explosion sound file.

Stardust (Ziggy is coming) by Kraftamt (c) copyright 2020 Licensed under a Creative Commons Attribution Noncommercial (3.0) license. http://dig.ccmixter.org/files/Karstenholymoly/62493 Ft: Platinum Butterfly (for the background music)

Credit to opengl for its tutorial in shadow volume, https://www.opengl.org/archives/resources/code/samples/advanced/advanced97/notes/node102.html.

Credit to Joey de Vries for his tutorial in opengl (mainly about texture mapping and alpha blending), https://learnopengl.com/Introduction

7 Bibliography

The Algorithmic Beauty of Plants, P. Prusinkiewicz & A. Lindenmayer, 1990, pp 21-46
(Bracketed OL-system tree)

GPU Gems, NVIDIA Corporation, 5th Edition, 2007
(Chapter 9 Efficient Shadow Volume Rendering)

Objectives:

Full UserID: c529zhan **Student ID: 20756699**

___ 1: Create models for the tanks and the base

___ 2: Create L-system tree trunk

___ 3: Use shadow volume

___ 4: Apply texture mapping

___ 5: Implement alpha blending

___ 6: Implement particle systems

___ 7: Create a skybox for the background of the scene

___ 8: Implement animations using interpolated keyframes

___ 9: Implement enemy logic

___ 10: Add sound effects and background music.