TDD SP4

Contents

[1.0 Technical Documentations 2](#_Toc381920153)

[1.1 Artificial Intelligence Programming 3](#_Toc381920154)

[General AI Programming 3](#_Toc381920155)

[State Transition 3](#_Toc381920156)

[AI Waypoint: 3](#_Toc381920157)

[AI Shooting at Player: 3](#_Toc381920158)

[1.2 Collision Programming 4](#_Toc381920159)

[General Collision: 4](#_Toc381920160)

[Continuous Collision Detection 5](#_Toc381920161)

[Broad-phase and Narrow-phase checking 6](#_Toc381920162)

[Components 6](#_Toc381920163)

[Code Flow 6](#_Toc381920164)

[Class Diagrams 7](#_Toc381920165)

[1.3 User Interface 10](#_Toc381920166)

[Main Menu 10](#_Toc381920167)

[glPrint() include, able to print text and edit its design in the BitFont.h 10](#_Toc381920168)

[1.4 Player Controls 10](#_Toc381920169)

[Key Binding 10](#_Toc381920170)

[1.5 Sounds 10](#_Toc381920171)

[1.6 Ship 11](#_Toc381920172)

[Movement 11](#_Toc381920173)

[Weapons 11](#_Toc381920174)

[Weapon Shooting 12](#_Toc381920175)

[2.0 Scripting Language 15](#_Toc381920176)

# 1.0 Technical Documentations

## 1.1 Artificial Intelligence Programming

### General AI Programming

The A.I that we will be creating will be controlled by switch cases and functions that will allow the enemy ships to act on their own. Switch cases indicate what they will be doing, and functions that allow the ship to fire bullets and maneuver around. The cases will switch based on timers that each of the enemy ships have upon spawning. To keep track of every enemy ship we will use vector list to store and to update their informations. When the enemy ships’ X position are close enough to the player ship’s X position, they will attempt to fire and hit the player.

### State Transition

For the types of enemies that can be encountered, each wave will already be assigned which enemy ships to spawn, based on the level of which the player is on. Normally when enemy ships spawn and fly in a single file formation, all of the enemy ships are the same kind. On certain levels, the enemy ships won’t be the same, and will have different attributes from the rest, like being able to hit more than 1 hit from the player’s bullets or moving in a different way. How this will be done, is based on Lua reading. The Lua file will indicate how many levels, what enemy ships to create on which levels, how many enemy ships to create on which levels, where do the enemy ships spawn from, indicate waypoints to fly through, and destinations for the enemy ships to stop at.

### AI Waypoint:

X = cos θ \* r

Y = sin θ \* r

Where r is the radius (how big) we want the waypoint to be. This enables us to make the AI move in a circle. By setting theta’s interval to 10 degrees, this means that for every 10 degrees based on this formula, a waypoint will be placed.

### AI Shooting at Player:

The approach we will be doing is to set a horizontal speed for the missile. The missile contains the player’s position and has a variable that limits it’s movement to only one side of the axis E.g. negative x-axis or positive x-axis. If the player moves in the opposite direction where the missile cannot move to, it will continue its flight path as per normal.

Basic waypoint logic

Basic waypoint function that is given to the enemy AI ships, so that the enemy ships will fly through each waypoint.

Created logic for enemy ships to fly in a circle

I helped create the basic function that creates waypoints that forms a circle for the enemy ships to fly through

Created logic for enemy ships to fly to their assigned positions

I helped create the basic function that is assigned to each enemy ship when they are released from their spawn points, which is a final waypoint that the enemy ships will fly to.

## 1.2 Collision Programming

### General Collision:

The game makes use of AABB and Sphere bounding values since it’s a 2D game. There is no environmental collision except the boundaries of the screen hence the Ship’s movement will be limited by the screen width.

For AABBs, the bounding volumes will be determined by the width and height of the sprite being used hence it can be dynamically resized according to the sprite.

### Continuous Collision Detection

There will be continuous collision detection as the enemies and bullets move at high speeds hence making sure that collision is detected properly.

CCD will be coded out by adding a variable to keep track of the projectile’s last pos. Before every frame, the program will store the projectile’s current position into the last position. The program will then check the distance the bullet has travelled per frame and using that information will interpolate the bullet from the last position to its current position while using the distance travelled as the limit.

For example:

Bullet current position is at (0, 5) and is moving upwards at 2 units per second. By storing this current position at our last position and then updating the bullet we now have the following information:

Current Pos = (0,7); (after update)

Last Pos = (0, 5)

The CCD code will then be applied as follows:

Distance between Current Pos and Last Pos = 2 (Scalar value)

Then, using the last pos, add a lerp value until it reaches the current position;

Along the way, check collision after every addition of the lerp value.

This solves the problem with collision between a slow and fast object.

### Broad-phase and Narrow-phase checking

The enemy’s bullets will only be checked for against the player’s ship when it reaches the bottom of the screen to reduce computing costs thus optimizing the game.

When the enemy’s bullet reaches the bottom half of the screen, only then will collision detection be enabled for that bullet. This is to save on the computational cost of this operation.

### Components

1. Collider Class

This class contains position and width and height of the bounding box and all other relevant information.

1. Collider Class

This class contains position and width and height of the bounding box and all other relevant information.

### Code Flow

Step 1:

A collider is attached to a game object. The game object’s respective collider is then passed into the Collision Manager which is a Singleton Class.

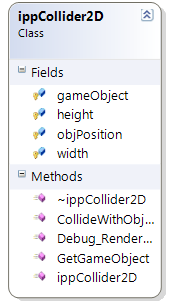
Step 2:

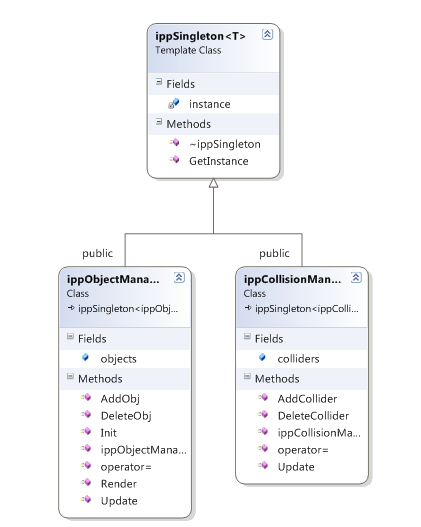
The Collision Manager checks for collision between each and every collider that is being stored. Broad phase and Narrow phase is implemented here for optimization

Step 3:

When there is a collision, the manager will call the callback function OnCollisionEnter for the collided object. Collision Handling is then handled by the respective GameObjects.

### Class Diagrams





## 1.3 User Interface

### Main Menu

The main menu will consist of button that the player will be able to scroll through using the Controller (XBOX360) that will be implemented into the game.

In the main game screen, players will see their score, highscore, stage number displayed in integers where as their lives are not only shown in numbers, but in graphical sprites resembling the player-controlled ships as well.

### glPrint() include, able to print text and edit its design in the BitFont.h

AddMenu(“name”,x,y) allows user to add a new page to a vector list, navigated by searching for its user defined names

Cursor tracks position of button it is currently pointed at

AddButton(“name”,x,y) allows users to add buttons to any pages also using a vector list, navigation and selection also using the buttons’ names.

Intergrated to use switch case with other parts of the game

## 1.4 Player Controls

### Key Binding

Xbox controller to replace all keyboard usage. (But can be used)

JoystickFunc used.

Keys are binded to an Xbox controller, entire game can be controlled by the controller

## 1.5 Sounds

Music and Sounds from the classic Galaga may be remixed and not sound 100% the same from the classic game, but will have enough familiar tones that the players will recognise it. Also, Irkklang will be used to emulate said sounds in above statement.

SoundEngine has been globalized, allowing other parts of the game to call it freely without re-initializing the engine to prevent overriding of engines.

## 1.6 Ship

### Movement

The ship will be able to move left or right along the X-axis. When the player press the left command, the X value of the vector3D Position will be minus off by a value times the time delta for frame independent movement (Position->x -= 10 \* timeDelta). When the player press the right command, the x value of the vector3D Position will be added by a value times the time delta for frame independent movement (Position->x += 10 \* timeDelta).

### Weapons

There will be virtual update and virtual render function as this is a base class for the different kind of weapons. The bullet class and missile class will then derive from the weapon class.

Turret class stores bullets in a vector ( bulletList.push\_back ( new ippBullet ( ) ) ) and the object-manager. It uses a circular buffer to cycle through the bullets to let it fire. Bullets set to not alive when it reaches the border of the screen. While the bullet is not alive, render and update will not be called.

Missilelauncher class stores missile in a vector ( missileList.push\_back ( new ippMissile ( ) ) ) and the object-manager. The missile will not be able to fire as long as one set of missile is on the screen. Missile set to not alive when it reaches the border of the screen. While the missile is not alive, render and update will not be called.

### Weapon Shooting

When the fire bullet function ( FireBullet () ) is called, a bullet that is not alive in the circular buffer will be set to alive. Once the bullet is alive, it will render and update. Bullets will not be able to fire when there are 2 sets of bullets on screen. The bullet will travel straight forward in the y-Axis from the ship’s position. This is done by adding the speed value to the bullet's y-axis each frame (Position->y += speed)

When the fire missile function ( FireMissile () ) is called, a pair of missile will be set to alive if it was originally not alive. Once the missile is alive, it will render and update. The ship is not able to fire the missile if there are already missiles on the screen. The missiles will appear from the side of the ship, shooting out to opposite directions ( direction = Vec3D ( -1.0f, 0.0f, 0.0f ) and direction = Vec3D ( 1.0f, 0.0f, 0.0f)). The missile will take the

## 1.7 Power Up

Power up

- Power up manager stores a pointer to a ippPowerUp object. Adds the object into object manager to be updated and rendered.

- Random power up will spawn at a set interval. ( (curTime - prevTime) > delay)

- The type of power up is determined by the type ( string type ).

- Ship will collect the power up and read what type of power up it is ( HandlePowerUp ( tempPowerUp->GetType () ) ).

- There are 2 power ups ( Missile charge and speed up )

## 1.8 Particles and Background

Able to create particles

Particles are created on certain events in the program, such as enemy ships being destroyed.

Each particle have its own lifespan and will disappear over time

Particles are given random lifespans, and fade out over its own lifespan.

- StarManager stores a list of star objects into a vector. ( vector<ippstar\*> starList; )

- The init of StarManager ( Init () ) randomize the size, speed and position on the screen.

Size of star: float size = rand () % 2 + 2;

Speed of star: float speed = rand () % 4 + 2;

Position:

tempVec.x = rand () % ( Global::WIDTH\_RESOLUTION );

tempVec.y = rand () % ( Global::HEIGHT\_RESOLUTION );

- Stars' position will be set on the top of the screen ( position.y = -10.0f ).

## 1.9 Save files and Encryption

**Save text file**

Creates new text file if the file doesn’t exist

When saving, the program will check if the file exists first, before storing information into the file. However when the file doesn’t exist, a file is created on the spot, and store the information into it.

Save game information into text file, like shop upgrades and high scores and is able to retrieve saved information

The program is able to store a lot of different information, and is also able to retrieve the information from the file.

**Encrypt text file**

All information on text file is encrypted can only be read inside the program

Whenever the file is being saved, it will be encrypted first, before being stored into the file

Encrypted using base64, credit to René Nyffenegger

Credits to René Nyffenegger, for his base64.h and base64.cpp files at this website [**http://www.adp-gmbh.ch/cpp/common/base64.html**](http://www.adp-gmbh.ch/cpp/common/base64.html)

## 2.1 Drones

- Drone class contains a Vec3D pointer that will store the position of the ship so it can keep track of the ship's position. Allows the drone to follow the ship.

- Bullet drone class is derived from the drone class

- Bullet drone stores a list of bullets ( ippBullet ) in a vector to be used

- Bullet drone rotates around the ship with an offset ( Rotate2D ( angle, true ) )

- Bullet drone fires bullet at a set interval based on the level the drone is at.

# 3.0 Scripting Language

Lua will be THE scripting language in the remake.

The uses of Lua will consist of but not limited to:

Scripting of enemy behaviors, spawn positions, spawn timings and timers and any code or variables that will be frequently changed to prevent the need for recompilation.