TDD SP4

Contents

[1.0 Technical Documentations 2](#_Toc380396263)

[1.1 Artificial Intelligence Programming 2](#_Toc380396264)

[General AI Programming 2](#_Toc380396265)

[State Transition 3](#_Toc380396266)

[AI Waypoint: 3](#_Toc380396267)

[AI Shooting at Player: 3](#_Toc380396268)

[1.2 Collision Programming 3](#_Toc380396269)

[General Collision: 4](#_Toc380396270)

[Continuous Collision Detection 4](#_Toc380396271)

[Broad-phase and Narrow-phase checking 5](#_Toc380396272)

[1.3 User Interface 5](#_Toc380396273)

[Main Menu 5](#_Toc380396274)

[Scores & Stage Levels 5](#_Toc380396275)

[1.4 Player Controls 6](#_Toc380396276)

[Key Binding 6](#_Toc380396277)

[1.5 Sounds 6](#_Toc380396278)

[1.5 Ship 6](#_Toc380396279)

[Movement 6](#_Toc380396280)

[Weapons 7](#_Toc380396281)

[Weapon Shooting 7](#_Toc380396282)

[2.0 Scripting Language 7](#_Toc380396283)

# 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.

## 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.

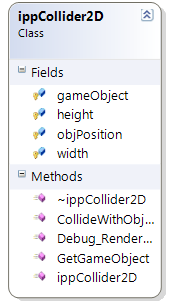
Step 2:

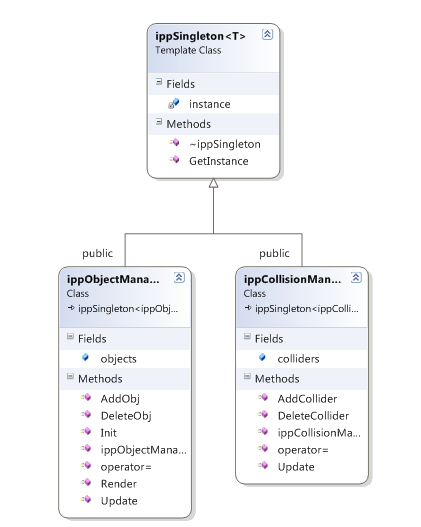
The Collision Manager checks for collision between each and every collider that is being stored. Broad phase and Narrow phase is implemented here for optimization

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 (PS3) tha vct 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.

### Scores & Stage Levels

Stage\_Number progressively adds as the player clears a level

Int Player1\_Lives, Player2\_Lives;

Galaga is able to run 2 players, so different initializations are needed to keep track of both players’ lives.

Int Player1\_Score, Player2\_Score

## 1.4 Player Controls

### Key Binding

In-Game key rebinding using Lua. When the player rebinds the key in-game, the program will check with the Lua config file to see if there are any discrepancies or changes. A Lua config file will use the values from the program and update its settings accordingly.

## 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.

## 1.5 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 (Position->x -= 10). When the player press the right command, the x value of the vector3D Position will be added by a value (Position->x += 10).

### Weapons

The weapon class will contain position of the weapon(vector3D Position), position of the target(vector3D \*tPosition) and speed of the projectile(float speed). 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.

### Weapon Shooting

When the shoot function is called, a new bullet object will be created and stored inside a weapon pointer (Weapon \*weapon = new Bullet(vector3D Position)). The bullet will then be pushed into a vector storing weapons for easier update of each weapon. The constructor of the bullet class will takes in the ship's position so that it knows where to position itself. The bullet will travel straight forward in the y-Axis. This is done by adding the speed value derived from the weapon class to the bullet's y-axis each frame(Position->y += speed)

# 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.