Software Architecture Document

AMBroSIA

Version 1.0

Nikolaos Bukas, Anthony Chin, Meong Hye Seo, Michael Smith, Haisin Yip

Ambrosia Games Ltd.

March 1, 2013

Table of Contents

1.0 System Overview

1.1 Purpose

The purpose of the document is to present a detailed description of all classes, objects and associated methods used in the game. The document is intended for current developers working on the project, future developers and the client. It is to be used as a point of reference when writing the game itself.

It should be noted that this document is subject to change, and may be modified by any intended party at any time if circumstances require it (i.e. it is discovered that intended functionality is impossible to implement as described in this document).

1.2 Scope

The piece of software to be written will be a game called “AMBroSIA”, modeled after the arcade-style game “Asteroids”. The game can be played by one or two people, and will feature a 2D graphical user interface in which the player will be able to control a spaceship and destroy approaching asteroids. As a game, the software will have an entertainment value to those who play it, with the development team’s goal being to maximize the entertainment value possible.

This design document presents an overview of how the game will be structured, without delving deeply into implementation details. It follows from the Software Requirements Specification

2.0 Views

2.1 UML Diagram

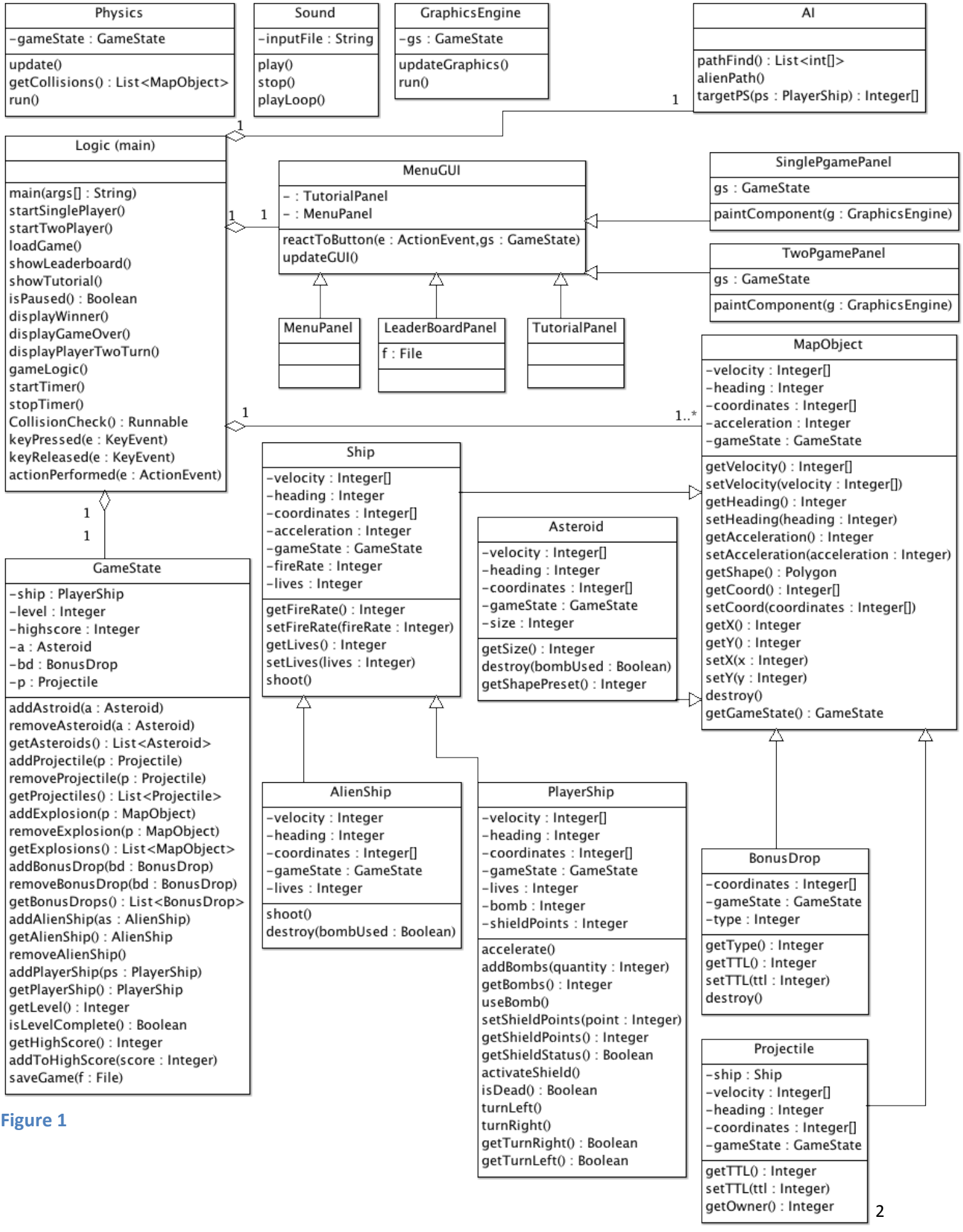
Physics
-gameState : GameState
update()
getCollisions() : List<MapObject>
run()
Sound
-inputFile : String
play()
stop()
playLoop()
GraphicsEngine
-gs : GameState
updateGraphics()
run()
AI
pathFind() : List<int[]>
alienPath()
targetPS(ps : PlayerShip) : Integer[]
Logic (main)
main(args[] : String)
startSinglePlayer()
startTwoPlayer()
loadGame()
showLeaderboard()
showTutorial()
isPaused() : Boolean
displayWinner()
displayGameOver()
displayPlayerTwoTurn()
gameLogic()
startTimer()
stopTimer()
CollisionCheck() : Runnable
keyPressed(e : KeyEvent)
keyReleased(e : KeyEvent)
actionPerformed(e : ActionEvent)
MenuGUI
- : TutorialPanel
- : MenuPanel
reactToButton(e : ActionEvent,gs : GameState)
updateGUI()
SinglePgamePanel
gs : GameState
paintComponent(g : GraphicsEngine)
TwoPgamePanel
gs : GameState
paintComponent(g : GraphicsEngine)
MenuPanel
LeaderBoardPanel
f : File
TutorialPanel
MapObject
-velocity : Integer[]
-heading : Integer
-coordinates : Integer[]
-acceleration : Integer
-gameState : GameState
getVelocity() : Integer[]
setVelocity(velocity : Integer[])
getHeading() : Integer
setHeading(heading : Integer)
getAcceleration() : Integer
setAcceleration(acceleration : Integer)
getShape() : Polygon
getCoord() : Integer[]
setCoord(coordinates : Integer[])
getX() : Integer
getY() : Integer
setX(x : Integer)
setY(y : Integer)
destroy()
getGameState() : GameState
Ship
-velocity : Integer[]
-heading : Integer
-coordinates : Integer[]
-acceleration : Integer
-gameState : GameState
-fireRate : Integer
-lives : Integer
getFireRate() : Integer
setFireRate(fireRate : Integer)
getLives() : Integer
setLives(lives : Integer)
shoot()
Asteroid
-velocity : Integer[]
-heading : Integer
-coordinates : Integer[]
-gameState : GameState
-size : Integer
getSize() : Integer
destroy(bombUsed : Boolean)
getShapePreset() : Integer
GameState
-ship : PlayerShip
-level : Integer
-highscore : Integer
-a : Asteroid
-bd : BonusDrop
-p : Projectile
addAstroid(a : Asteroid)
removeAsteroid(a : Asteroid)
getAsteroids() : List<Asteroid>
addProjectile(p : Projectile)
removeProjectile(p : Projectile)
getProjectiles() : List<Projectile>
addExplosion(p : MapObject)
removeExplosion(p : MapObject)
getExplosions() : List<MapObject>
addBonusDrop(bd : BonusDrop)
removeBonusDrop(bd : BonusDrop)
getBonusDrops() : List<BonusDrop>
addAlienShip(as : AlienShip)
getAlienShip() : AlienShip
removeAlienShip()
addPlayerShip(ps : PlayerShip)
getPlayerShip() : PlayerShip
getLevel() : Integer
isLevelComplete() : Boolean
getHighScore() : Integer
addToHighScore(score : Integer)
saveGame(f : File)
AlienShip
-velocity : Integer
-heading : Integer
-coordinates : Integer[]
-gameState : GameState
-lives : Integer
shoot()
destroy(bombUsed : Boolean)
PlayerShip
-velocity : Integer[]
-heading : Integer
-coordinates : Integer[]
-gameState : GameState
-lives : Integer
-bomb : Integer
-shieldPoints : Integer
accelerate()
addBombs(quantity : Integer)
getBombs() : Integer
useBomb()
setShieldPoints(point : Integer)
getShieldPoints() : Integer
getShieldStatus() : Boolean
activateShield()
isDead() : Boolean
turnLeft()
turnRight()
getTurnRight() : Boolean
getTurnLeft() : Boolean
BonusDrop
-coordinates : Integer[]
-gameState : GameState
-type : Integer
getType() : Integer
getTTL() : Integer
setTTL(ttl : Integer)
destroy()
Projectile
-ship : Ship
-velocity : Integer[]
-heading : Integer
-coordinates : Integer[]
-gameState : GameState
getTTL() : Integer
setTTL(ttl : Integer)
getOwner() : Integer
1
1..*

Figure 1

3.0 Software Units

3.1 Classes

3.1.1 Logic (main class)

3.1.1.1 Purpose

The purpose of the logic class is to manage all other classes and methods in such a way as to produce a playable game. This includes (but is not limited to) calling for the creation of the main menu, displaying the leaderboard or starting the game in either single or two player mode as appropriate, depending on player actions.

3.1.1.2 Interface

This class implements ActionListener and extends KeyAdapter.

void main(String args[])
Main method; creates the main menu and acts as appropriate depending on player input.

void startSinglePlayer()
This method will start the single player game.

void startTwoPlayer()
This method will start the game in 2 player mode.

void loadGame()
This method will prompt the player to select a save game file. The file will be read and the original game state will be loaded allowing the player to resume his game.

void showLeaderboard()
This method will display the leaderboard to the user. The leaderboard file (*.txt) will be retrieved when method is called to display stored score information.

void showTutorial()
This method will show the tutorial information to the user.

boolean isPaused()
This method checks if the game is paused

void displayWinner()
The output displays text stating that the player has won (as appropriate for single or two player).

void displayGameOver()
The output display a “Game Over” message.

void displayPlayerTwoTurn()
The method will display information relevant to player two's turn.

void gameLogic()
The method will provide the entire gameflow.

void startTimer()
This method will start the global timer responsible for keeping all game elements up to date. If possible, the timer will use some form of multithreading to execute update tasks concurrently.

void stopTimer()
This method will stop the aforementioned timer.

Runnable collisionCheck()
This method will provide a Runnable class that can be used to check for collisions between game objects.

void keyPressed(KeyEvent e)
This method will handle events caused by user key presses.

void keyReleased(KeyEvent e)
Same as above, except is only used when a key is released.

void actionPerformed(ActionEvent e)
Handles events relating to the user clicking on menu buttons, such as "Single Player" or "Tutorial".

3.1.2.1 MenuGUI

3.1.2.1.1 Purpose

The purpose of the GUI class is to provide the frame of the game. This frame will output the main menu screen, tutorial screen, and the gameplay.

3.1.2.1.2 Interface

MenuGUI()
This is a constructor that initializes the basic GUI and draws the menu. The menu may feature a background animation, if time permits.

void reactToButton(ActionEvent e, GameState gs)
This method handles all button presses as appropriate.

void updateGUI()
Redraws the screen based on information present in the GameState at the time the method is called.

3.1.2.2 MenuPanel

3.1.2.2.1 Purpose

The purpose of the MenuPanel class is to provide the basic panel on which the menu is drawn.

3.1.2.2.2 Interface

MenuPanel()
This constructor draws the panel in question.

3.1.2.3 TutorialPanel

3.1.2.3.1 Purpose

The purpose of the TutorialPanel class is to provide the player with instructions on how to play the game

3.1.2.3.2 Interface

TutorialPanel()
This constructor draws the panel in question.

3.1.2.4 LeaderBoardPanel

3.1.2.4.1 Purpose

The purpose of the LeaderBoardPanel class is to draw a leaderboard displaying the scoring information of past players.

3.1.2.4.2 Interface

LeaderBoardPanel(File f)
This constructor draws the panel in question, displaying the information contained in the file.

3.1.2.5 SinglePgamePanel

3.1.2.5.1 Purpose

The purpose of the SinglePgamePanel class is to display the game itself - the map, all objects, information like highscore, lives remaining, etc. for the single player mode.

3.1.2.5.2 Interface

SinglePgamePanel(GameState gs)
This constructor draws the panel in question, using the GameState to retrieve necessary information about the game (highscore, polygon for the player's ship, etc.)

paintComponent(Graphics g)
This method is responsible for the actual drawing of shapes (asteroids, aliens, player's ship) on the screen.

3.1.2.6 TwoPgamePanel

3.1.2.6.1 Purpose

The purpose of the TwoPgamePanel class is to display the game itself - the map, all objects, information like highscore, lives remaining, etc. for the two player mode.

3.1.2.6.2 Interface

TwoPgamePanel(GameState gs)
This constructor draws the panel in question, using the GameState to retrieve necessary information about the game (highscore, polygon for the player's ship, etc.)

paintComponent(Graphics g)
This method is responsible for the actual drawing of shapes (asteroids, aliens, player's ship) on the screen.

3.1.3 GameState

3.1.3.1 Purpose

The purpose of the GameState class is to keep the state of the game in memory. This is done by holding references to all the Asteroid, Projectile, Bonus Drop, and Ship objects currently active in the game. Information like the high score and level is also stored.

3.1.3.2 Interface

GameState(int level, int highScore)
This is the constructor for the GameState class. It creates empty lists for asteroids, projectiles, bonus drops, and explosions. The high score and level are set as defined in the arguments.

void addAsteroid(Asteroid a)
Will add the object a into the list of asteroids.

void removeAsteroid(Asteroid a)
Will remove the object a from the list of asteroids.

List<Asteroid> getAsteroids()
Returns a list of Asteroids.

void addProjectile(Projectile p)
Will add the object p into the list of projectiles.

void removeProjectile(Projectile p)
Will remove the object p from the list of projectiles.

List<Projectile> getProjectiles()
Returns a list of Projectiles.

void addExplosion(MapObject p)
Will add the object p into the list of explosions.

void removeExplosion(MapObject p)
Will remove the object p from the list of explosions.

List<MapObject> getExplosions()
Returns a list of MapObjects representing explosions.

void addBonusDrop(BonusDrop bd)
Will add the object bd to the list of bonus drops.

void removeBonusDrop(BonusDrop bd)
Will remove the object bd from the list of bonus drops.

List<BonusDrop> getBonusDrops()
Returns a list of Bonus Drops.

void addAlienShip(AlienShip as)
Will make the AlienShip variable reference object as.

AlienShip getAlienShip()
Returns the AlienShip object in game state.

void removeAlienShip()
Will remove the AlienShip from the game state.

void addPlayerShip(PlayerShip ps)
Will make the PlayerShip variable reference object ps.

PlayerShip getPlayerShip()
Returns the PlayerShip object in game state.

int getLevel()
Returns the level.

boolean isLevelComplete()
Return true if level is completed.

int getHighScore()
Returns the high score.

void addToHighScore(int score)
Adds to high score the value of score.

void saveGame(File f)
Saves the current state of the game to the specified file.

3.1.5 MapObject

3.1.5.1 Purpose

The purpose of the MapObject is to provide a general representation of game objects (such as asteroids and aliens). It includes functionality present in all objects (to some extent).

3.1.5.2 Interface

MapObject(int[] velocity, int heading, int[] coordinates, int acceleration, GameState gameState)
The constructor, which takes all essential values: a velocity (magnitude and direction), heading, coordinates, acceleration and the GameState, used to add the object to the game when created or remove it when destroyed.

int[] getVelocity()
Returns an array representing the velocity of the object.

void setVelocity(int[] velocity)
Sets the velocity to the given value.

int getHeading()
Returns the heading in which the object is facing (not to be confused with the direction related to velocity).

void setHeading(int heading)
Sets the heading to the specified value.

int getAcceleration()
Returns the acceleration of the object.

void setAcceleration(int acceleration)
Sets the acceleration to the specified value.

Polygon getShape()
Returns the polygon representing the shape.

int[] getCoord()
Returns an array of length 2 containing the coordinates of the object.

void setCoord(int[] coordinates)
Sets the coordinates to those given in the *coordinates* array.

int getX()
Returns the x coordinate of the object.

int getY()
Returns the y coordinate of the object.

void setX(int x)
Sets the x coordinate to the specified value.

void setY(int y)
Sets the y coordinate to the specified value.

void destroy()
The destroy method is used to removed the MapObject from the game state. Necessary animations, sound effects or other effects of the destruction are also taken care of (destroy() overridden in inheriting classes when appropriate).

GameState getGameState()
Returns the gameState object initially passed as a constructor argument.

3.1.5.1 BonusDrop

3.1.5.1.1 Purpose

The purpose of the BonusDrop is to provide a general representation of the all the bonus drop objects. It inherits the MapObject class. In addition, it contains integer properties for the type and Time to Live (TTL).

3.1.5.1.2 Interface

BonusDrop(int[] coordinates, GameState gameState, int type)
The constructor of BonusDrop takes coordinates, the gamestate and type as parameters. The type indicates what type of bonus drop it is (extra life, extra bomb, extra shield points). It adds the BonusDrop object to the GameState.

int getType()
Returns the integer value of the type property (represents whether its a life or a bomb).

int getTTL()
Returns the int value of the TTL property.

void setTTL(int ttl)
Sets the TTL to the int parameter.

void destroy()
Removes BonusDrop object from the gamestate.

3.1.5.2 Projectile

3.1.5.2.1 Purpose

The purpose of the Projectile is to provide the representation for the projectile objects. It inherits properties and methods from the MapObject class.

3.1.5.2.2 Interface

Projectile(Ship ship, int[] velocity, int heading, int[] coordinates, GameState gameState)
The Projectile constructor takes the same parameters as the MapObject constructor with the addition of the TTL. It adds the Projectile object to the game.

int getTTL()
Returns the integer value of the TTL property.

void setTTL(int ttl)
Sets the TTL to the int parameter.

int getOwner()
Returns an int value that represents the owner of this Projectile. The owner is the one that fired/created the projectile.

3.1.5.3 Asteroid

3.1.5.3.1 Purpose

The purpose of the Asteroid class is to represent asteroids in the game. Most parameters and methods are inherited from MapObject, with one exception.

3.1.5.3.2 Interfaces

Asteroid(int[] velocity, int heading, int[] coordinates, GameState gameState, int size)
Constructor takes the same parameters as the MapObject constructor but in addition it takes size. It adds the Asteroid object to the gamestate. The shapePreset number is randomly generated at this time.

int getSize()
Returns the size of the asteroid.

destroy(boolean bombUsed)
Removes the asteroid from the gamestate. It increases the player's high score if the boolean parameter is set to false. It adds new asteroids into the game depending on its size as determined in the SRS.

int getShapePreset()
Returns a number representing what basic shape this asteroid is to take.

3.1.5.4 Ship

3.1.5.4.1 Purpose

The Ship object is used as a generic representation of a ship, either player or alien. It inherits from MapObject.

3.1.5.4.2 Interfaces

ship(int[] velocity, int heading, int[] coordinates, int acceleration, GameState gameState, int fireRate, int lives)
The constructor takes the same parameters as the MapObject constructor in addition to fireRate and lives.

int getFireRate()
Returns the firing rate.

void setFireRate(int fireRate)
Sets the firing rate.

int getLives()
Returns the number of lives.

void setLives(int lives)
Sets the number of lives.

void shoot()
Creates a projectile.

3.1.5 PlayerShip

3.1.5.1 Purpose

The PlayerShip class defines all the properties and methods appropriate to the player ship that are not included in Ship. It inherits from the Ship class.

3.1.5.2 Interface

PlayerShip(int[] velocity, int heading, int[] coordinates, GameState gameState, int lives, int bomb, int shieldPoints)
The constructor initializes the object much like MapObject, with the addition of the lives, bomb and shieldPoints.

void accelerate()
Sets the PlayerShip to accelerate.

void addBombs(int quantity)
Adds quantity to bombs.

int getBombs()
Returns the amount of bombs.

void useBomb()
Detonates (uses) the bomb.

void setShieldPoints(int points)
Sets the amount of shield points.

int getShieldPoints()
Returns the amount of shield points.

boolean getShieldStatus()
This method checks if shield if activated

void activateShield()
This method activates the shield for the ship.

boolean isDead()
This method checks if the player or the ship is dead or not.

void turnLeft()
Sets the PlayerShip to turn left.

void turnRight()
Sets the PlayerShip to turn right.

boolean getTurnRight()
Returns true if the ship is turning right.

boolean getTurnLeft()
Returns true if the ship is turning left.

3.1.5 AlienShip

3.1.5.1 Purpose

The AlienShip class defines all the properties and methods appropriate to the alien ships that are not included in Ship. It inherits from the Ship class.

3.1.5.2 Interfaces

AlienShip(int[] velocity, int heading, int[] coordinates, GameState gameState, int lives)
The constructor initializes the object much like MapObject, with the addition of the lives and adds it to the GameState.

shoot()
The method defines the weapon mechanic of the AlienShip and does so by creating projectiles.

destroy(boolean bombUsed)
Removes the AlienShip object from the GameState and increases the high score of the player if the boolean parameter is set to false.

3.1.6 Physics

3.1.6.1 Purpose

The purpose of the physics class is to provide the game physics. It calculates velocities and displacements and is able to detect collisions. Implements Runnable.

3.1.6.2 Interfaces

Physics(GameState gameState)
The constructor takes the GameState as a parameter.

void update()
Updates the physical properties (velocity, heading, xy coordinates, and acceleration) of every in-game object.

ArrayList<MapObject> getCollisions()
This method returns a list of MapObjects that have been detected in a collision.The list will always an even number of entries. Each sequential pair of MapObjects indicates that the two of them collided.

void run()
Because Physics implements Runnable the run() method is used to call the update() method.

3.1.7 Sound

3.1.7.1 Purpose

This purpose of the Sound class is to output the sound needed for the game.

3.1.7.2 Interfaces

Sound(String inputFile)
The constructor takes as String input the path of the sound file.

void play()
Plays the sound only once.

void stop()
Stops the current sound from playing.

void playLoop()
Continuously plays the song.

3.1.8 GraphicsEngine

3.1.8.1 Purpose

The GraphicsEngine class is responsible for defining the shape of all in-game objects, and placing said shapes at the appropriate locations based on the position of the object. It implements Runnable.

3.1.8.1 Interfaces

Graphics(GameState gs)
The constructor takes the GameState as input.

void updateGraphics()
Updates the shapes of all in-game objects by placing them at the new position of the object (note: asteroid shapes are defined by their size and the number returned by the getShapePreset() method; see section 3.1.5.3)

void run()
Because GraphicsEngine implements Runnable the run() method is used to call the update() method.

3.1.9 AI

3.1.9.1 Purpose

The AI class serves a controller for the Alien ship. In other words, it will control how the Alien ship moves and reacts.

3.1.9.2 Interfaces

ArrayList<int[]> pathfind()
Finds and returns the a set of coordinates representing the path it should follow.

void alienPath()
The alien will traverse a certain path, before disappearing.

int[] targetPS(PlayerShip ps)
Returns a set of coordinates of where the player ship might be next and shoot the projectile at that point.

3.2 Categories

AI	Physics	Sound	Graphics	GUI	Game Objects	Logic
AI	Physics	Sound	GraphicsEngine	MenuGUI MenuPanel TutorialPanel LeaderBoardPanel SinglePGamePanel TwoPGamePanel	MapObject BonusDrop Projectile Asteroid Ship PlayerShip AlienShip	Logic (main class) GameState

4.0 Analysis

The following traceability matrix demonstrates the coherency of the SAD and SRS.

4.1 Traceability Matrix

Requirement	AI	Physics	Sound	GraphicsEngine	GUI	Game Objects	Logic
R3.1.1.3.1	-	-	-	-	MenuPanel()	-	startSinglePlayer()
R3.1.1.3.2	-	-	-	-	LeaderboardPanel()	-	showLeaderboard()
R3.1.1.3.3	-	-	-	-	startTwoPlayer()	-	startTwoPlayer()
R3.1.1.3.4	-	-	-	-	TutorialPanel()	-	showTutorial()
R3.1.1.3.5	-	-	-	-	quit()	-	-
R3.1.2.3.1	-	-	-	-	drawSidePanel()	-	-
R3.1.2.3.2	-	-	-	updateGraphics()		-	gameLogic()

					-		
R3.1.2.3.3	-	-	-	-	-	-	gameLogic()
R3.1.2.3.4	-	getCollisions()	-	-	-	destroy()	-
R3.1.2.3.5	-	-	-	-	displayGameOver()	-	-
R3.1.2.3.6	-	-	-	-	displayWinner()	-	-
R3.1.2.3.7	-	-	-	-	updateSide()	-	gameLogic()
R3.1.2.3.8	-	-	-	-	-	-	gameLogic()
R3.1.3.3.1	-	-	-	-	LeaderboardPanel()	-	showLeaderboard()
R3.1.3.3.2	-	-	-	-	-	-	showLeaderboard()
R3.1.3.3.3	-	-	-	-	loadLeaderBoard()	-	-
R3.1.3.3.4	-	-	-	-	loadLeaderBoard()	-	-
R3.1.3.3.5	-	-	-	-	generateMainMenu()	-	-
R3.1.4.3.1	-	-	-	-	drawSidePanel()	-	-
R3.1.4.3.2	-	-	-	updateGraphics()	-	-	startTwoPlayer()
R3.1.4.3.3	-	-	-	-	drawSidePanel()	-	startTwoPlayer()
R3.1.4.3.4	-	-	-	-	UpdateSide()	-	gameLogic()
R3.1.4.3.5	-	-	-	-	displayPlayerTwoTurn()	-	-
R3.1.4.3.6	-	-	-	-	displayPlayerTwoTurn()	-	-
R3.1.4.3.7	-	-	-	-	displayPlayerTwoTurn()	-	-

R3.1.4.3.8	-	-	-	-	displayWinner()	-	-
R3.1.4.3.9	-	-	-	-	UpdateSide()	-	gameLogic()
R3.1.5.3.1	-	-	-	-	TutorialPanel()	-	showTutorial()
R3.1.5.3.2	-	-	-	-	generateMainMenu()	-	-
R3.1.6.3.1	-	update()	-	-	-	-	keyPressed(), keyReleased()
R3.1.6.3.2	-	-	-	-	-	useBomb()	keyPressed(), keyReleased()
R3.1.6.3.3	-	update()	-	-	-	-	keyPressed(), keyReleased()
R3.1.6.3.4	-	update()	-	-	-	-	keyPressed(), keyReleased()
R3.1.6.3.5	-	update()	-	-	-	-	keyPressed(), keyReleased()
R3.1.6.3.6	-	-	-	-	-	shoot()	keyPressed(), keyReleased()
R3.1.6.3.7	-	-	-	-	-	activateShield()	keyPressed(), keyReleased()
R3.1.7.3.1	-	update()	-	-	-	-	keyPressed(), keyReleased()
R3.1.7.3.2	-	-	-	updateGraphics()	-	-	-

R3.1.7.3.3	-	-	-	-	-	-	gameLogic()
R3.1.8.3.1	-	update()	-	-	-	setLives()	gameLogic()
R3.1.8.3.2	-	getCollisions()	-	updateGraphics()	-	-	-
R3.1.8.3.3	-	-	-	-	-	-	-
R3.1.8.3.4	-	-	-	-	-	-	-
R3.1.8.3.5	-	-		-	-	-	-
R3.1.8.3.6	-	update()	-	-	-	-	-
R3.1.9.3.1	-	-	-	-	-	shoot()	-
R3.1.9.3.2	-	-	-	-	-	-	-
R3.1.9.3.3	-	-		-	-	-	-
R3.1.9.3.4	-	-	-	-	-	-	keyPressed()
R3.1.9.3.5	-	-	-	-	-	useBomb()	-
R3.1.9.3.6	-	-	-	-	-	addBomb()	-
R3.1.9.3.7	-	-	-	-	-	useBomb()	-
R3.1.9.3.8	-	-	-	-	-	useBomb()	-
R3.1.9.3.9	-	update()	-	-	-	-	-
R3.1.9.3.10	-	-	-	-	-	-	-
R3.1.9.3.11	-	-	-	-	-	Projectiles()	-
R3.1.9.3.12	-	-	-	-	-	shoot()	-
R3.1.9.3.13	-	getCollisions()	-	-	-	-	-
R3.1.9.3.14	-	update()	-	-	-	setVelocity()	-
R3.1.10.3.1	-	-	-	-	-	setShieldPoints()	-
R3.1.10.3.2	-	getCollisions()	-	-	-	-	gameLogic()
R3.1.10.3.3	-	-	-	updateGraphics()	-	activateShield()	-

R3.1.10.3.4	-	-	-	updatesGraphics()	-	activateShield()	-
R3.1.10.3.5	-	-	-	-	-	setShieldPoints()	-
R3.1.10.3.6	-	-	-	-	-	-	gameLogic()
R3.1.11.3.1	-	getCollisions()	-	-	updateGUI()	-	gameLogic()
R3.1.11.3.2	-	getCollisions()	-	-	updateGUI()	-	gameLogic()
R3.1.11.3.3	-	getCollisions()	-	-	updateGUI()	-	gameLogic()
R3.1.11.3.4	-	getCollisions()	-	-	updateGUI()	-	gameLogic()
R3.1.11.3.5	-	getCollisions()()	-	-	-	-	gameLogic()
R3.1.12.3.1	-	-	-	updateGraphics()	-	-	-
R3.1.12.3.2	-	getCollisions()	-	updateGraphics()	-	getSize()	-
R3.1.12.3.3	-	getCollisions()	-	updateGraphics()	-	getSize()	-
R3.1.12.3.4	-	getCollisions()	-	updateGraphics()	-	getSize()	-
R3.1.12.3.5	-	getCollisions()	-	-	-	-	gameLogic()
R3.1.12.3.6	-	getCollisions()	-	updateGraphics()	-	-	gameLogic()
R3.1.12.3.7	-	-	-	updateGraphics	-	-	gameLogic()
R3.1.13.3.1	-	getCollisions()	-	-	-	-	gameLogic()
R3.1.13.3.2	-	-	-	-	-	-	gameLogic()
R3.1.13.3.3	targetPS()	-	-	-	-	-	-
R3.1.13.3.4	targetPS()	-	-	-	-	-	-
R3.1.13.3.5	-	-	-	updateGraphics()	-	-	-

R3.1.13.3.6	alienPath()	update()	-	-	-	-	-
R3.1.13.3.7	alienPath()	-	-	-	-	-	-
R3.1.13.3.8	-	-	-	updateGraphics()	-	-	gameLogic()
R3.1.14.3.1	-	getCollisions()	-	updateGraphics()	-	destroy()	-
R3.1.14.3.2	-	getCollisions()	-	updateGraphics()	-	-	-
R3.1.14.3.3	-	getCollisions()	-	-	-	-	gameLogic()
R3.1.14.3.4	-	getCollisions()	-	-	updateGUI()	-	-
R3.1.14.3.5	-	-	-	-	-	-	gameLogic()
R3.1.14.3.6	-	getCollisions()	-	updateGraphics	-	-	gameLogic()
R3.1.15.3.1	-	-	-	-	-	-	gameLogic()
R3.1.15.3.2	-	-	-	-	-	-	gameLogic()
R3.1.16.3.1	-	-	-	updateGraphics()	-	-	-
R3.1.16.3.2	-	-	-	-	-	-	-
R3.1.16.3.3	-	-	-	updateGraphics()	-	isDead()	-
R3.1.16.3.4	-	-	-	updateGraphics()	-	destroy()	-
R3.1.16.3.5	-	-	-	updateGraphics()	-	getAcceleration()	-
R3.1.16.3.6	-	-	-	updateGraphics()	-	-	gameLogic()
R3.1.17.3.1	-	-	play()	-	-	destroy()	-
R3.1.17.3.2	-	-	play()	-	-	isDead()	-
R3.1.17.3.3	-	-	play()	-	-	destroy()	-
R3.1.17.3.4	-	-	play()	-	-	shoot()	-
R3.1.17.3.5	-	getCollisions()	play()	-	-	-	-

R3.1.17.3.6	-	-	play()	-	-	setAcceleration()	-
R3.1.17.3.7	-	-	playLoop()	-	-	-	-
R3.1.17.3.8	-	-	play()	-	-	isDead()	-
R3.1.17.3.9	-	-	play()	-	-	-	-

4.2 Quality Requirements

[R3.2.1.2] The biggest usage of memory will be in loading all the graphical and audio assets of the game into RAM. We are assuming the average size of a sprite is 1 MB each, 1 MB each for the various sound effects, and 4 MB for the background theme song. In total there will be 10 (3x asteroids, 1x player ship, 1x alien ship, 1x projectile, 3x bonus drops, 1x explosion) sprites, 5 sound effects (1x explosion, 1x shooting, 1x acceleration, 1x victory, 1x game over), and 1 background sound. Therefore the total memory usage will be 24 MB (10 MB graphics, 9 MB sound, 5 MB code).

[R3.2.1.3] Assuming the same average sizes as above, and assuming a total size of 2 MB for code, the game will use a maximum of 21 MB of hard drive storage.

[R3.2.1.4] The game will be programmed in Java and therefore is compatible with most modern operating systems that have a Java Virtual Machine.

The follow requirements are currently not applicable. They will be satisfied during the testing phase:

R3.2.1.1
R3.2.1.4
R3.2.1.5
R3.2.2.1

The software architecture was not designed with maintainability in mind. It is specifically specified in the System Requirements Specification that maintainability is not a requirement. For reuse and flexibility, please see our design rationale that uses example cases to show that our architecture was designed taking reuse and flexibility into account.

4.3 Design Rationale

A custom architecture was chosen for this project, as common design layouts were not suited for a project of AMBroSIA's scale. For a small project with only a few group members and limited time to meet, a modular approach was needed. As such, the following model was chosen:

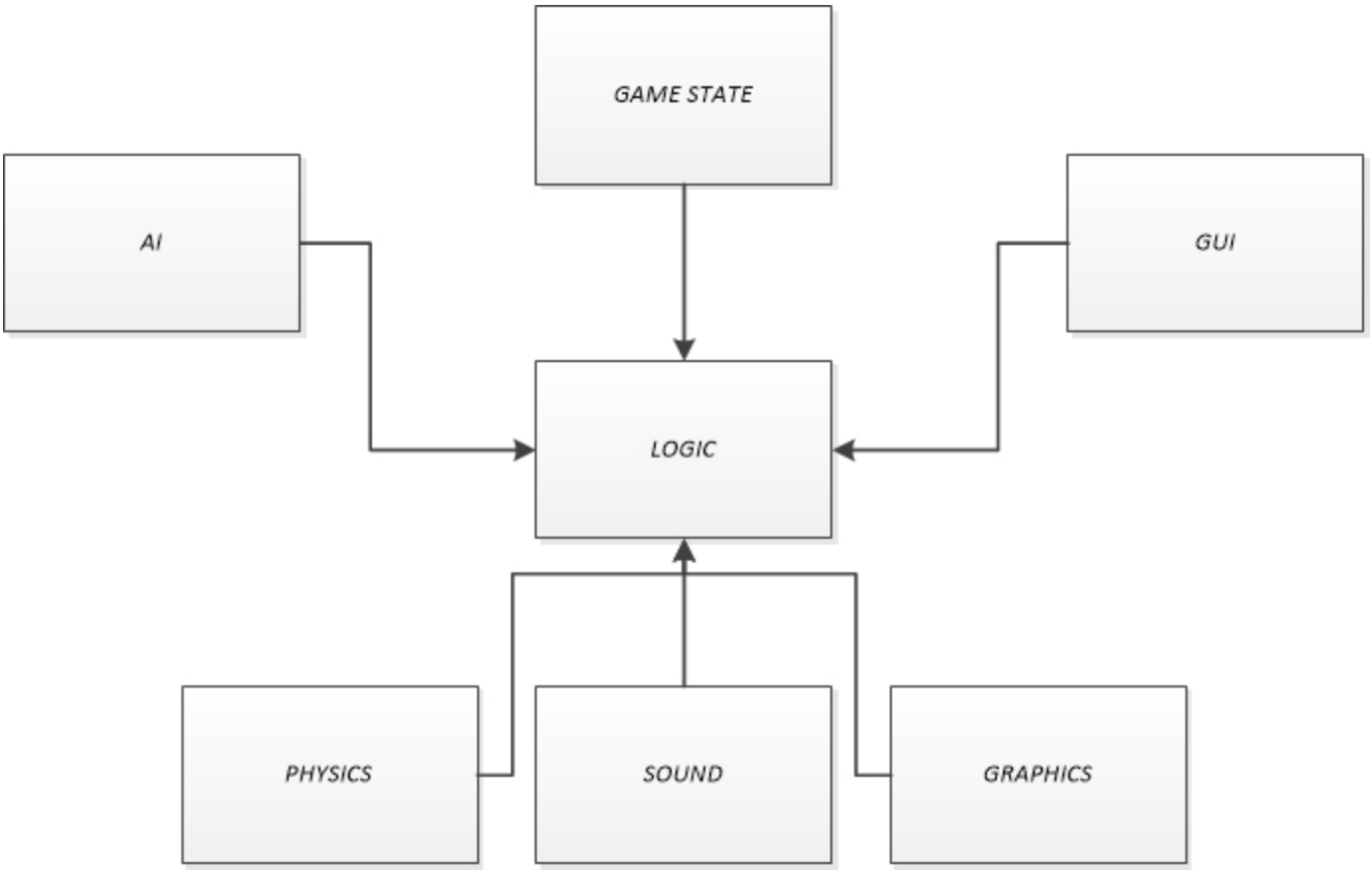

Figure 2

This model allows individual classes to be developed separately, with placeholders for not yet completed classes or methods. For example, objects present on the map - such as the player ship or the asteroids - can use (or be used by) classes such as physics, sound or graphics, without knowing the implementation details involving, for example, trajectory calculations. This also allows code performing similar functionality to be contained within the appropriate class, rather than being split up among many different classes. This reduces interdependencies, improving flexibility.

A good example involves the calculation of the next position of an object for the next frame: objects do not need to know how their position is calculated. Rather, they need only store the data about their current position and some important properties - such as velocity - while the rest is left up to physics. For example, asteroids move at constant velocity, but the player ship has acceleration and momentum. This requires a different calculation, but having both implemented in the physics class reduces the need of code duplication.

With regards to other classes, a MapObject was used (with inheritance) as it allows functionality common to all objects to be grouped together. It also ensures that code that needs to use properties common to all objects (but nothing more) can access anything from the player's ship to an asteroid without using different classes. This simplifies the code needed and improves its readability. Code reuse is also made possible.

In terms of visuals, the Graphics class was created to draw the polygons of each individual game object, while the GUI class was developed to handle the drawing of the actual interface (ie. the game panels). While they are related, they do perform drastically different functions (a simple check of the import java.x statements makes it quite evident) making it best to keep them separate. This keeps the code as flexible as possible, while making testing easier, as bugs in the GUI can be isolated from the polygon calculations, and vice versa.

When it comes to storing game data, a GameState class was created whose sole purpose is to keep track of all objects and data in the game. This allows a central access point from which any piece of code can view and change game information, simplifying function calls and making it easy to keep track of what objects exist. It also ensures changes in data storage do not have an adverse affect on other classes - an example of code flexibility.

Finally, the AI class is used to control the AlienShip. It provides paths for the AlienShip and coordinates for which the AlienShip should direct its weapon fire to (the player ship). Given that the algorithms involved would likely be quite different from the AlienShip objects and other in-game objects, the decision was made to keep it separate to allow for code flexibility.