**Report:**

**Motivation:**

The motivation for this game was to produce a game with high replay value, challenging gameplay and a clear goal. AttackGame meets two of these criteria but does not fulfil the other as well as it could.

The goal of the attack game is to survive ever escalating numbers of enemies to gain more and more points. The game provides a high replay value by having an extensive upgrade system and varying levels of difficulty, that makes it more interesting to play again and see how the other upgrades improve the ship or how hard or easy the other difficulties are.

The place I feel the game falls short is in providing a clear goal. Beyond the acquisition of points and the complete upgrading of all systems there is nothing specific the player is aiming for. To improve this if given more time I would add a goal, possible to get to a (very large) number of points, as this would create another decision for the player, whether they want to spend their points on ways to survive longer or if they want to save them towards their goal.

**Class Overview:**

**Class: AttackGame**

**Inherits From: Microsoft.Xna.Framework.Game**

**Overview:** AttackGame is responsible for keeping track of the game state, the menu selections and difficulty of the game, for setting up the initial game parameters and keeping track of important information like the current score of the player. It is also responsible for keeping several lists of current content, namely a list of objects to draw while the game is running, a list of objects that must be updated each tick when the game is running and a list of enemies.

AttackGame is responsible for drawing all these items and the HUD when the game is running and for drawing the correct menu when it is not.

**Method Overview:**

**Initialise:** Sets up any constructs that will be kept the same whenever the game is reset, or should not be touched by the game resetting, for example difficulty or the List of objects used to load models.

**Reset:** Sets up structures that will be reinitialised when the game resets, for example game state being splash, the avatar.

**LoadContent:** Loads all content necessary for the game. Outside of this method Content.Load<> is never used. This looks through the LoadList created in initialise and assigns the specified models to all of the given objects, as well as loading some other important content (Sound effects, HUD textures etc.)

**Update:** Updates the game based on the current state, contains all logic for the menus, in the case the game is running just calls update on all the objects in the updateableobjects list, checks if it needs to spawn a new enemy, and looks for any input not covered by the Craft class (changing state, disabling/enabling music)

**addPoints:** A public method to allow enemies to add their point value to the players score when they die. Adds the specified number of points \* the time alive multiplier \* the difficulty multiplier

**spawnRandomEnemy:** Spawns a random enemy type at a random location.

**loseGame:** a public method allowing other classes to tell the game the player has lost and change the state

**Draw:** If the game is running draws the models, if not routes the player through to the correct method to draw their desired menu.

**drawHUD:** Draws the game HUD if the game is running.

**drawSplash, drawDifficultyMenu, drawControlScreen, drawUpgradeMenu, drawGameOver:** Based off FuelCell example drawing of splash screen. Draws a specific menu. Due to time constraints these are all rather inefficient and reuse a lot of code. With more time I would refactor these more.

calculateBoundingSphere: taken from FuelCell and used unmodified. Calculates a merged bounding sphere based off the model bounding spheres

collides: checks if 2 object’s bounding spheres intersect

listCollides: looks at a list and generates a list consisting of all objects in that list that collide with another given object

findNearest: looks at a list and finds the nearest object in that list to a given object, used avoid ambiguity in collision detection

addObject, addUpdatingObject, removeObject, removeUpdatingObject, addEnemy, enemyDestroyed: List management, adds and removes objects from lists. Important to note that updating objects in not directly affected to prevent issues with the update list being modifier as it is iterated over.

**Class: Camera**

**Overview:** Handles camera stuff. Taken fully from chasecam and modified to not use the spring.

**Class: PlayerStats**

**Overview:** Provides a set of calls to get different stat values as well as a list of all stats. Adding a stat here will automatically update the update menu.

**Class: Stat**

**Overview:** The lowest element of the level system, stores the base value of a stat, how much the stat should be increased/decreased per level, the name of it, the current level, the max level and the cost of upgrading the stat at level 1.

**Methods:**

getNextLevelCost: Works out the cost of upgrading to the next level by: baseCost \* currentLevel

getValue: works out the current value by: baseVal + (increment \* (currentLevel - 1))

upgradeStat: public method to allow the incrementing of the level of the stat.

**Class: GameObject**

**Overview:** GameObject is used in polymorphism to allow the storing of all game objects in the AttackGame class to a list of objects to draw. GameObjects need basic model requirements, position, bounding sphere, model. Any GameObject created is automatically added to the object list in its constructor. The Model variable is static as it will be the same for all objects of this type. All subclasses override this to allow different models for different classes.

Methods:

UpdateWorld: Recalculates the world matrix

DrawModel: draws the object at its given position with its given model

destroy: removes the model from the drawn objects list

**Inherits from: GameObject**

**Class: Terrain**

**Overview:** A subclass of terrain to allow a separate model to be stored for the terrain.

**Class: MovingGameObject**

**Overview:** A moving game object, keeps track of its speeds

**Methods:**

Update: Uses speed to update the position of the object in the world

boundaries: keeps the moving object inside the game boundaries

fireBullet: fires a bullet in the direction of the object with a specified speed, damage and range.

destroy: destroys this object, removes it from the list of updating objects then calls the base destroy

damage: abstract

**Inherits from: MovingGameObject**

**Class: Bullet**

**Overview:** A bullet, responsible for its own collision detection and for playing the firing and hit sounds

**Methods:**

**Update:** destroys the bullet if its out of range or hitting anything except its parent or another bullet, then calls base update.

**damage**: override. If the bullet is damaged it is destroyed.

DrawModel: override. Used to provide bright ambient light to make the bullets appear more solidly in the game.

**Class: Enemy**

**Overview:** Enemies keep track of how often they have been hit, how many points they are worth and the location of the avatar

**Methods:**

LookAt: finds the vectors needed for the enemy to be pointing at a given point.

Update: Checks enemy collision, destroys them and damages the craft if they collide with it, if 2 enemies collide bounces them apart.

bounce: moves the enemy away from a given enemy

damage: damages the enemy, reducing its HP if it has none left destroys it

destroy: removes the enemy from the enemy list and gives the player points for its death

**Subclasses of Enemy:**

**Mine:** An extension of enemy that moves towards the player and glows if the player is nearby.

**Orbiter:** An extension of enemy that moves around the player, staying at a desired distance and occasionally pausing and shooting at the player. The main enemy of the game.

**Class: Craft**

**Overview:** The player craft, keeps track of important information

**Methods:**

Update: Updates the ship based on velocity, which increases and decreases gradually based on physics calculations.

Calculate\*\*\*\*Amount: Calculating a certain type of input for the update method

checkOtherInput: Checks for shooting the weapons

damage: Override. Damages shields before hull of the ship

manageShield: Checks if the shield should be recharging and makes it do so if it should.

DrawModel: Overridden to allow a blue glow dependant on shield strength.

**Additional Functionality:**

* Regenerating shield system on top of health system.
* Upgrade system, with different upgrades affecting the ship in different ways
* 2 Different enemy types
* Combat system, with semi complex enemy behaviour
* Difficulty levels, based both on time alive (enemies spawn faster the longer you survive) and selected difficulty level.
* Simple toggleable sound effects for firing and hitting shots
* Coloured lights to represent the states of some entities (Mines glow red when active, Player ship glows blue dependent on shield strength and flashes white or red when damaged with or without shields)
* Working HUD and menu system.
* Complex scoring system (scores are based off both enemy type, how long the player has survived and the difficulty)

**References:**

Code

Heavy use was made of:

<http://rbwhitaker.wikidot.com/>

specifically:

<http://rbwhitaker.wikidot.com/playing-sound-effects> for the sounds in the Bullet class

and <http://rbwhitaker.wikidot.com/basic-effect-lighting> for the lighting systems in the Mine and Craft classes.

The **Fuelcell** example, provided at: <http://msdn.microsoft.com/en-us/library/dd940288.aspx> (As of 25/04/2012)

Formed the base of the menu system and hud.

The GameObject class is a heavily modified version of the one found in the FuelCell example.

The CalculateBoundingSphere method was taken and used unmodified from the FuelCell example.

The **ChaseCam** example, provided at: **//////////////////////ADD THIS**(As of 25/04/2012)

ChaseCamera was used for the camera system, removing the spring and just using the reset() method calls to keep the camera locked behind the spaceship

Ship was used as the base of the spaceship, and was heavily modified to add strafing, shields, shooting etc.

The DrawModel method in the ChaseCameraGame class was used unmodified, except in the cases of the craft and the mines.

The Draw method is also based off the one in ChaseCameraGame

The LookAt method in enemies is taken from: <http://xnawiki.com/index.php?title=Vector_Math#Calculate_the_rotation_for_one_vector_to_face_another_vector> (Up as of 15/04/2012)

Models:

The models not produced by me are:

Ground.x (provided in the chasecam example)

spaceship.x (provided on the course webpage)

bulletCube.x and cube10uR.x (provided on the course webpage)

Textures:

Models/White.png was provided with the FuelCell example

Models/ground.png was provided with the ChaseCam example, and was later edited by me.

Sound Effects:

All sound effects were taken from freesfx.com