CAB201 - Assignment, Exercise 1: Assignment

Return to Exercise List »

Results so far:

Latest Score: n/a marks, out of n/a (n/a%).

Best Score: n/a marks, out of n/a (n/a%).

Assessed weight: 0%. This item is non-assessed. You may continue to attempt this item until you reach 10 submissions. So far, you have made 0 submissions.

Requirements:

CAB201 Assignment - Tank Battle

Assignment Weight: 40%

Tank Battle is a game in which each player controls a tank and attempts to hit the other player(s) by adjusting the angle to aim at and the amount of power to use.

You are provided with an existing Visual Studio project to use as a framework. Use of this framework is not optional. Note that **you will be doing Windows Forms programming**, which means that Visual Studio for Mac and MonoDevelop will likely be insufficient. Ensure that you have access to Visual Studio in order to work on this assignment.

(/CAB201/Downloads/53179305.zip)

Download the Tank Battle Visual Studio project here. (/CAB201/Downloads/53179305.zip)

You will develop this assignment as a **bottom-up implementation exercise**. This means that you will work based off a specification that has already been prepared and implement classes and methods, one by one, first implementing those classes and methods that don't rely on other classes to operate, then implementing the classes and methods that rely on those and so forth. This means that it will take quite some time before a functioning tank game emerges out of this exercise. It also means you won't be able to test that you've implemented various parts of the program correctly except by doing your own testing.

Table of Contents

1. TankBattle Unit Testing Framework
2. Frequently Asked Questions
3. Changes
4. Class Specification
    - Gameplay
    - Map
    - Opponent
    - PlayerController
    - BattleTank
    - Blast
    - Effect
    - Shell
    - GameplayForm
    - MainMenuForm
    - AIPlayer
5. Additional Functionality
6. Marking Criteria
7. Optional Groups

TankBattle Unit Testing Framework

When you download the above Visual Studio project, then compile and run it, you will be greeted with a console displaying this:

```
Checking classes for public methods...
All public methods okay.
[failed] Gameplay.Gameplay(): System.NotImplementedException
[      ] Gameplay.NumPlayers(): -> Gameplay.Gameplay()
[      ] Gameplay.GetMaxRounds(): -> Gameplay.Gameplay()
[      ] Gameplay.CreatePlayer(): -> Gameplay.Gameplay()
[      ] Gameplay.GetPlayer(): -> Gameplay.Gameplay()
[failed] Gameplay.GetColour(): System.NotImplementedException
[failed] Gameplay.GetPlayerLocations(): System.NotImplementedException
[failed] Gameplay.Shuffle(): System.NotImplementedException
[      ] Gameplay.BeginGame(): -> Gameplay.Gameplay()
[failed] Map.Map(): System.NotImplementedException
[      ] Gameplay.GetArena(): -> Map.Map()
[      ] Gameplay.GetCurrentPlayerTank(): -> Gameplay.Gameplay()
[failed] TankModel.GetTank(): System.NotImplementedException
[      ] TankModel.DisplayTankSprite(): -> TankModel.GetTank()
[failed] TankModel.LineDraw(): System.NotImplementedException
[      ] TankModel.GetTankArmour(): -> TankModel.GetTank()
[      ] TankModel.WeaponList(): -> TankModel.GetTank()
[      ] Opponent.PlayerController(): -> TankModel.GetTank()
[      ] Opponent.GetTank(): -> TankModel.GetTank()
[      ] Opponent.Identifier(): -> TankModel.GetTank()
[      ] Opponent.GetColour(): -> TankModel.GetTank()
[      ] Opponent.AddScore(): -> TankModel.GetTank()
[      ] Opponent.GetScore(): -> Opponent.AddScore()
[      ] PlayerController.CommenceRound(): -> TankModel.GetTank()
[      ] PlayerController.NewTurn(): -> Gameplay.BeginGame()
......
```

(note that there are quite a few lines, so you may have to scroll up to see this)

These are unit tests and there are 61 in total. Each unit test corresponds to one method of one class. After you have finished programming a particular method of a class, if you implemented the method correctly, the unit test for it should pass.

With a couple of exceptions, the methods you are asked to implement have been provided in the Visual Studio project, but with the body of the method replaced with the line: `throw new NotImplementedException();` This means that, if any of these methods is called, they will simply throw this exception. The unit test driver is designed to handle exceptions and as a result shows the exception name as the reason these test cases failed.

The left-most column of the unit test console gives the result of the test: passed, failed or blank. A blank unit test result means the unit test was not run due to this unit test relying, either directly or indirectly, on a previous unit test. Because many methods cannot be tested by themselves, many of the unit tests rely on some methods other than the one it is testing. As a result, these unit tests will not run until all prerequisite unit tests have

passed. For example, at the moment the Gameplay.NumPlayers(), Gameplay.GetMaxRounds(), Gameplay.CreatePlayer() unit tests cannot be run as they are waiting for Gameplay's constructor to be implemented and pass its own unit test first- because instance methods cannot be called before the constructor, and if the constructor isn't working properly it doesn't make much sense to check the instance variables.

What this means for you is that you should focus on implementing methods that are currently failing unit testing. Once you have fixed those methods, other methods will become available for testing.

From the start, these five cases are available for testing: **Gameplay.Gameplay(), Gameplay.GetColour(), Gameplay.GetPlayerLocations(), Gameplay.Shuffle(), Map.Map(), TankModel.GetTank()** and **TankModel.LineDraw()**.

It is therefore recommended that you implement those methods first, which will then open up more methods for you to implement, until you have implemented everything and all your unit tests pass.

Note that there are other public methods and classes that you will need to implement, but don't have unit tests attached. Most of these public methods are used by other public methods that are themselves tested, meaning that you will still need to implement them to be able to have a successfully-passing unit test.

FrequentlyAskedQuestions

So far there are no frequently asked questions about this assignment. Watch this space.

Changes

- 26/09/2017 - Changed "Creates a numPlayers size array of Opponent:Opponent" to "Creates a numPlayers size array of Opponent"

Class Specification

public class Gameplay

public Gameplay(int numPlayers, int numRounds)

Gameplay's constructor. This is called with the number of players in the game and the number of rounds that will be played. This method:

- Creates a numPlayers size array of Opponent (which is stored as a private field of Gameplay)
- Sets another private field to the number of rounds that will be played
- Creates an array or list collection of Effect (if it's an array, a suitably large array should be created for it, e.g. 100).

No objects of type Opponent or Effect should be created here; only the arrays.

- numPlayers will always be between 2 and 8 (inclusive)
- numRounds will always be between 1 and 100 (inclusive)

public void CreatePlayer(int playerNum, Opponent player)

This method takes a player number (guaranteed to be between 1 and the number of players) and sets the appropriate field in Gameplay's Opponent array to player. Note that the value of playerNum is indexed from 1 while arrays are indexed from 0, so CreatePlayer() will need to account for this.

public Opponent GetPlayer(int playerNum)

This method takes a player number (between 1 and the number of players) and returns the appropriate Opponent from the Opponent array. Again, note that arrays are indexed from 0 while playerNum is indexed from 1.

public static Color GetColour(int playerNum)

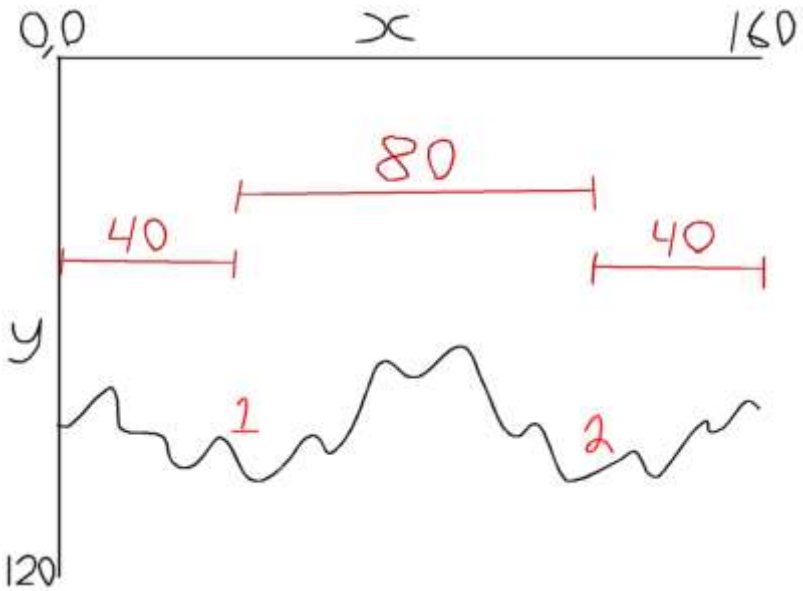
This static method takes a player number (between 1 and the number of players) and returns an appropriate colour to be used to represent that player. Note that you can access a number of predefined colours by using Color.[colour name], e.g. `Color c = Color.LightGreen;` Simply typing in `Color.` in Visual Studio should give you a list.

- Each player should get a bright, visually distinct colour. e.g. GetColour(1) and GetColour(2) should not return colours that look similar.
- Each time this method is called with a particular number it must return the same colour. For example, if GetColour(1) returns Color.Red it must return Color.Red at every subsequent invocation.

public static int[] GetPlayerLocations(int numPlayers)

Given a number of players, this static method returns an array giving the horizontal positions of those players on the map. For example, if GetPlayerLocations(2) was called, it would return something like {39, 119}. Note that the total width of the map (160) is stored in the Map.WIDTH constant so use this in your calculations.

- Player positions in the array are to be strictly from left to right: e.g. {40, 120} is okay, but {120, 40} is not okay.
- The array size must equal numPlayers; GetPlayerLocations(4) must return an array of length 4, for example.
- Each player is to be placed equally close to its neighbours- for four players, if the first player is 40 tiles away from the second player, the second player should be (approximately) 40 tiles away from the third player and the third player should be (approximately) 40 tiles away from the last player.
- The leftmost and rightmost players should be placed a distance away from their respective horizontal border that is half the size of the distance between players.



- To ensure that tanks are attractively positioned, take into account that a TankModel is **4 x 3** and these locations are based on the leftmost position of the tank.

For example, on a 10x5 battlefield, a TankModel placed in the horizontal centre:

```
..........
..........
..........
...####...
...####...
```

...actually has a horizontal position of 3 (positions are base 0). Note that you can get the TankModel width and height from the TankModel.WIDTH and TankModel.HEIGHT constants.

- Because this method returns integers, it is not necessary that the distances described be exact as in many situations this may be impossible. It may be preferable to work with floats, then round them with Math.Round and cast them to ints before putting them in the array. This is not necessary, however.
- numPlayers will never be less than 2; however, it won't necessarily be limited to the maximum number of players. As a result you should calculate these positions rather than having predefined arrays for each player count. That said, if your method works for up to 8 players and doesn't work for a greater number of players you should still receive most of the marks for this method.

public static void Shuffle(int[] array)

This method, given an array of at least 1, randomises the other of the numbers in it:

e.g. {0, 1, 2, 3} might become {0, 3, 1, 2} or {1, 2, 0, 3} or {3, 1, 2, 0} etc.

It is important that the array has the exact same numbers as it did before the call to Shuffle; only the order the numbers appear in will change.

public void BeginGame()

This method begins a new game, which involves doing the following.

- A private field in Gameplay representing the **current round** is initialised to 1.
- A private field in Gameplay representing the **starting Opponent** is initialised to 0. A variable is used to hold this because the starting Opponent changes with each round.
- The CommenceRound() method is called. As a result you will be unable to fully test this method until CommenceRound() is implemented. (The test case may run but will throw a NotImplementedException).

public void CommenceRound()

This method begins a new round of gameplay. A game consists of multiple rounds, and a round consists of multiple turns. Starting a new round involves the following:

- Initialising a private field of Gameplay representing the **current player** to the value of the **starting Opponent** field (see BeginGame).
- Creating a new Map, which is also stored as a private field of Gameplay.
- Creating an array of Opponent positions by calling GetPlayerLocations with the number of Opponents playing the game (hint: get the length of the Opponents array)
- Looping through each Opponent and calling its CommenceRound method.
- Shuffling that array of positions with the Shuffle method.
- Creating an array of BattleTank as a private field. There should be the same number of BattleTanks as there are Opponents in the Opponent array.
- Initialising the array of BattleTank by finding the horizontal position of the BattleTank (by looking up the appropriate index of the array returned by GetPlayerLocations and shuffled with the Shuffle method), the vertical position of the BattleTank (by calling TankYPosition() on the Map with the horizontal position as an argument), and then calling BattleTank's constructor to create that BattleTank (passing in the appropriate Opponent, the horizontal position, the vertical position and a reference to **this**).
- Initialising the wind speed, another private field of Gameplay, to a random number between -100 and 100.
- Creating a new GameplayForm and Show()ing it.

public Map GetArena()

This method returns the current Map used by the game. This is stored in a private field and is initialised by CommenceRound().

public BattleTank GetCurrentPlayerTank()

This method returns the BattleTank associated with the current player. Both the current player and an array of BattleTank are private fields of Gameplay and are also initialised in CommenceRound().

public void DrawPlayers(Graphics graphics, Size displaySize)

This method tells all the BattleTanks to draw themselves. The graphics and displaySize arguments are not important here; they are simply passed along to the METHOD_DRAW_ methods.

- Loop over each BattleTanks in the array.

- Check if a BattleTank is still around by calling its Exists() method.
- If it is, call its Display() method, passing in graphics and displaySize.

public void AddWeaponEffect(Effect weaponEffect)

This method adds the given Effect to its list/array of Effects. If you used an array, you will want to do something like this:

- Loop over the Effect array.
- Find the first blank spot by comparing each Effect to `null`.
- Once the first blank spot has been found, set it to weaponEffect and break out of the loop.

If you used a list or similar collection, you can just Add() it to the list.

Finally, call ConnectGame on the newly-added Effect, passing in a reference to **this**. This allows the Effect to access this Gameplay for the purposes of checking for collisions and inflicting damage on players.

public bool ProcessWeaponEffects()

This method loops through all Effects in the array, calling Tick() on each.

Remember, if you are using an array, that `null` is used for slots in the array that do not contain Effects, so make sure you check that before calling Tick().

This method returns true if there were any Effects to call Tick() on, and false otherwise. This is used to determine if there are any tank shells or explosions still being animated, because the animation doesn't stop until they are all gone.

public void DrawAttacks(Graphics graphics, Size displaySize)

This method loops through all Effects in the array, calling Display() on each. graphics and displaySize are not used by this method itself, but are passed to Effect's Display() method.

Once again, if you are using an array, `null` is used for slots in the array that do not contain Effects, so make sure you check that before calling Display() on it.

public void RemoveEffect(Effect weaponEffect)

This method removes the Effect referenced by weaponEffect from the array or list used by Gameplay to store active Effects.

If you are using an array, this involves finding the appropriate Effect and setting it to `null`. If you are using a list collection, just remove it from the list.

public bool CheckHitTank(float projectileX, float projectileY)

This method returns true if a Shell at projectileX, projectileY will hit something.

- If the coordinates given are outside the map boundaries (less than 0 or greater than Map.WIDTH or Map.HEIGHT respectively), return false.

- If the Map contains something at that location (hint: use Get), return true.
- If there is a BattleTank at that location, return true.
- To detect collisions against BattleTanks, loop through the array of BattleTanks and check if the point described by projectileX, projectileY is inside the rectangle occupied by BattleTank. The position of the BattleTank can be found using the GetX() and Y() methods, while the width and height are stored in TankModel.WIDTH and TankModel.HEIGHT.
- Note that collisions can never occur against the current player's BattleTank. Otherwise shots fired by a tank would instantly hit that same tank.

public void InflictDamage(float damageX, float damageY, float explosionDamage, float radius)

This method inflicts up to explosionDamage damage on any BattleTanks within the circle described by damageX, damageY and radius.

- Loop through the array of BattleTanks.
- If the BattleTank is still alive (check this with BattleTank's Exists() method), calculate the distance between it and damageX/damageY.
- For this calculation, first work out the position of the centre of the BattleTank. (Use GetX() and Y(), then add on half of TankModel.WIDTH and TankModel.HEIGHT. Calculate and store this position as a float for more accurate calculations.
- Calculate the distance between damageX, damageY and the BattleTank's position.
- Hint 1: Use Pythagoras' theorum. Math.Sqrt(Math.Pow(x1 - x2, 2) + Math.Pow(y2 - y2, 2))
- Hint 2: Math methods return a double, so you'll have to cast it to a float.
- Once you have the distance between the damage point and the BattleTank, compare it to radius.
- If the distance is greater than radius, no damage is done to the BattleTank.
- If the distance is between radius and radius/2, calculate the damage done by multiplying explosionDamage by the difference between radius and dist, then dividing that by radius.
- If the distance is less than radius/2, explosionRadius damage is done.
- Call BattleTank's METHOD_DAMAGE_() method with the amount of damage to be inflicted.

public bool GravityStep()

This method is called after all Effect animations have finished and moves any terrain and/or BattleTanks that are floating in the air down. Similarly to ProcessWeaponEffects, this method returns false once there is nothing left to move, and true until then. It will be called in a loop by the GameplayForm.

- Use a bool to keep track of whether anything moved. It should start out as false, but may be set to true based on what the other methods called by this method return.
- First, call the GravityStep() method of Gameplay's Map. This moves any loose terrain down, and it also returns true if anything moved, so if Map's GravityStep() returns true, set the bool to true.
- Next, loop through all BattleTanks in the array, calling their GravityStep() methods. Once again, if any of them returned true, set the bool to true.
- If the bool keeping track of whether anything moved is set to true, return true. Otherwise return false.

public bool FinishTurn()

This method is called once the current turn is over. It checks how many BattleTanks are still in battle, makes a determination as to whether the round is over or not, and if not changes the current player to the next player that's still in the running. This method returns true if the round is still going, and false if it's over.

- Loop through the array of BattleTanks, calling Exists() on each to check if it's still around.
- If there's 2 or more BattleTanks still around, the round continues. Increment the private field representing the current player, then check if the new current player is Exists().
- If it is, that player is the new current player. Otherwise, keep looping, going back to 0 if you run out of players.
- Adjust the wind speed (increasing it by a random number between -10 and 10). If it falls below -100 or 100, set it to -100 or 100 respectively.
- Return true, indicating that the round is still going.
- If there are 0 or 1 BattleTanks still around, call FindWinner() and return false, indicating that the round is over.

public void FindWinner()

This method is called by FinishTurn() after it decides that the current round is over. It finds out which player won the round and rewards that player with a point.

Loop through each BattleTank in the array, calling Exists() on each. If it returns true, call AddScore() on the appropriate Opponent (you can either look up the appropriate index in the Opponent array or call GetPlayer() on the BattleTank to get a Opponent to call AddScore() on.

public void NextRound()

This method is called by GameplayForm after the round is over (GameplayForm decides whether the round is over based on whether FinishTurn() returned true or false).

- Increment the current round (which is a private field of Gameplay that was initialised in BeginGame()). If the current round is less than or equal to the number of rounds a match is supposed to go for, the game continues. Switch to the next starting player (increment the private starting player field by 1, then reset it back to 0 if it reached the number of players in the game) and call CommenceRound().
- If the current round is greater than the number of rounds the game goes for, the game is over. If you have a leaderboard, now is the time to show that. Alternatively, just show the MainMenuForm again.

public int GetWindSpeed()

This method returns the current wind speed, which is a private field of Gameplay and ranges between -100 and 100.

public class Map

This class represents the landscape, the arena on which the tanks battle. The terrain is randomly generated and can be destroyed during the round. A new Map with a newly-generated terrain is created for each round.

public Map()

This constructor randomly generates the terrain on which the tanks will battle. It should create a two-dimensional array of bools to be used for representing the terrain (where 'true' means there is terrain at that location). The size of the array is in the constants Map.WIDTH and Map.HEIGHT (160 x 120). It's up to you whether to put the y coordinate first (new bool[HEIGHT,WIDTH][) or the x coordinate first (new bool[WIDTH,HEIGHT]) as long as you use the same notation consistently.

It's up to you how to go about generating the random landscape- however there are some restrictions on how you generate it.

- There must always be room at the top for a BattleTank. This means the first TankModel.HEIGHT rows of the map must be left empty.
- In any given column of the map, there must not be any locations containing terrain that lie above a location that doesn't contain terrain (in other words, no floating ground).

In other words, this is okay:

```
............................
............................
............................
............................
...........#................
........#####....#..........
###...########.#####.......##.
######################...#####
##############################
```

But this is not:

```
............................
............................
............................
.................###........
...................##.......
.....................###....
.....###...........######....
..########.....############..
##############################
```

- There cannot be any bottomless pits (i.e. locations on the bottom row of the map that don't contain any terrain.) These may emerge during gameplay as a result of land destruction, where they will serve as a hazard, but the map cannot be generated with them.

- The map must actually be random, and not (always) completely flat. The approach to use is up to you as long as the above restrictions are followed.
- Hint: To get an idea of what your Map looks like before you have the GameplayForm hooked up to it, create a Main() method to instantiate a Map, call Get() in a loop to draw the map to the console, using `Console.Write()` and `Console.WriteLine()`.

public bool Get(int x, int y)

This returns whether there is any terrain at the given coordinates. If there is, it returns true. Otherwise, it returns false.

- x will never be less than 0 or greater than Map.WIDTH - 1.
- y will never be less than 0 or greater than Map.HEIGHT - 1.

public bool CheckTankCollide(int x, int y)

This is similar to Get, but it returns whether there is room for a tank-sized object (a tank is TankModel.WIDTH wide and TankModel.HEIGHT tall) at the given coordinates. The coordinates refer to the top left of the tank position, which means that if our map looks like this:

```
............................
............................
........x...................
............................
...........#................
........#####....#..........
###...########.#####.......##.
######################...#####
##############################
```

...then calling CheckTankCollide(8, 2) (where the 'x' is) will return true because there is some terrain in the bottom right corner of the 4x3 window.

- If there is any terrain within the tank area, this method returns true. Otherwise it returns false.
- Neither x nor y will ever be less than 0.
- x will never be greater than Map.WIDTH - TankModel.WIDTH.
- y will never be greater than Map.HEIGHT - TankModel.HEIGHT.

public int TankYPosition(int x)

This method takes the x coordinate of a tank and, using that, finds the largest (lowest) y coordinate where a tank can go. Keeping in mind the tank dimensions of TankModel.WIDTH by TankModel.HEIGHT (4x3), if this method was called with a value of 15 on this map:

```
.............................
.............................
..............TTTT...........
..............TTTT...........
...........#...TTTT...........
........#####....#............
###...########.#####.......##.
######################...#####
##############################
```

...the method would return 2, as these are the upper-left coordinates of the lowest position on that x coordinate the tank can reach without colliding with the terrain. The position of the tank is indicated by a block of Ts. If the x value was further over (for example, 20) the tank would be placed in a lower position.

- Due to the restrictions on terrain generation outlined in Map's constructor, this method can never fail. There is always room for a tank somewhere at every x position.
- x will never be less than 0 or greater than Map.WIDTH - TankModel.WIDTH.
- Hint: Use CheckTankCollide() to check if the tank can fit in a particular location.

public void DestroyTerrain(float destroyX, float destroyY, float radius)

This method destroys all terrain within a circle centred around destroyX, destroyY. This method is called after a Shell explodes, destroying the terain

- Loop over every y coordinate of the map:
- Within that, loop over every x coordinate of the map:
- Calculate the distance between the x and y coordinate and destroyX, destroyY. (Once again, use Pythagoras- see Gameplay.InflictDamage().
- If the distance is less than radius, set the terrain at that position to false.

Here is a demonstration of what DestroyTerrain() is supposed to do. Take the following map:

```
.............................
.............................
..............TTTT...........
..............TTTT...........
...........#...TTTT...........
........#####....#............
###...########.#####.......##.
######################...#####
##############################
```

The following block shows the area that would be affected by a call to DestroyTerrain(10, 4, 3.5)

```
.............................
.........XXX.................
........XXXXX................
.......XXXXXXX...............
.......XXXXXXX...............
.......XXXXXXX...#...........
###...##XXXXX#.#####.......##.
#########XXX##########...#####
##############################
```

Destroying the terrain in this area will leave the map looking like this:

```
.............................
.............................
.............................
.............................
.............................
................#............
###...##.....#.#####.......##.
#########...##########...#####
##############################
```

public bool GravityStep()

This method moves any loose terrain (that is, terrain that is directly above an empty space) down one tile. Like the other GravityStep methods, this returns true if any terrain was moved and false otherwise. This has the effect of compacting the terrain down after damage from combat leaves holes in mountains and other unstable formations.

Before:

```
.............................
.............................
.............................
.................###.........
..................##.........
..................###........
.....###..........######.....
..########.....#############..
##############################
```

After:

```
.............................
.............................
.............................
.............................
.................####........
..................####.......
.....###..........######.....
..########.....#############..
##############################
```

As you can see, every tile of terrain that was above an empty space was moved down. If there are multiple tiles of terrain in a column above an empty space, those tiles will move down as a group, as the following demonstrates in two calls to GravityStep():

```
.###.     .....     .....
.###.     .###.     .....
..... ->  .###. ->  .###.
.....     .....     .###.
#####     #####     #####
```

The two calls to GravityStep() depicted above would both return true as terrain moved in both of those calls. If GravityStep() was called a third time it would return false as there is no longer any floating terrain to move down.

public abstract class TankModel

TankModel is an abstract class representing a generic tank model. You will need to create at least one concrete class inheriting from it for this game to work. Different tanks can have different graphics, weapons, armour etc.

public static TankModel GetTank(int tankNumber)

This is a factory method, used to create a new object of a concrete class that inherits from TankModel and return it. This way different parts of the program can create a variety of different tanks without having to know anything about your concrete class.

If you only have one type of tank your method can simply be something like:

```
return new MyTank();
```

If you have multiple varieties of tank, you will want to return a specific type of tank based on the value of tankNumber. The unit tests assume that tank numbers start at 1.

public abstract int[,] DisplayTankSprite(float angle)

This method draws the tank into an array and returns it. This is an abstract class and will therefore be implemented by each tank that inherits from TankModel.

DisplayTankSprite() returns an int[12,16] array containing a 1 for each pixel that makes up the tank's shape. The code for translating this array into a bitmap and applying colouring and outlines is provided (see the CreateBMP method), but you will need to draw the tank. The angle parameter is the angle of the tank's turret--90 is straight left, 0 is straight up and 90 is straight right.

Here is an example of declaring an int[12,16] array containing a basic image of a tank (without the turret):

```
int[,] graphic = { { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                   { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                   { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                   { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                   { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                   { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                   { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                   { 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0 },
                   { 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0 },
                   { 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0 },
                   { 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0 },
                   { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } };
```

Once you have this tank graphic, you can draw the turret by calling LineDraw starting from 7, 6 (the position directly above the centre of the row of three 1s at the top) to whatever the correct location is given the angle.

- For example, if the angle is 0 and the tank is aiming straight up, you might call LineDraw(graphic, 7, 6, 7, 1); to get this: 
- If it's -90, you might LineDraw(graphic, 7, 6, 2, 6); to get this:
- If it's 90, you might LineDraw(graphic, 7, 6, 12, 6); to get this:
- -45, LineDraw(graphic, 7, 6, 3, 2); 
- 45, LineDraw(graphic, 7, 6, 1, 2); 
- ...and so forth.

There's a couple of ways you can handle drawing the turret at the correct angle:

- Have a series of 'if' statements to check various angles and draw a different line based on whether the angle falls between a certain set of points. For example, you might draw the -90 degree line if the angle is less than -67.5, the -45 degree line if the angle is between -67.5 and -22.5 and so forth.
- Use Math.Sin() and Math.Cos() to determine where to draw the line to. You will need to convert from degrees to radians and move the angle into the right quadrant for this to work, but it will allow you to draw a turret at any angle. This is the best option if you can pull it off.

Finally, make sure that you don't write any numbers to the outermost border of the array. This is because the outlining code wants to create a 16x12 bitmap and the outline will be cut off, which will look weird.

LineDraw(int[,] graphic, int X1, int Y1, int X2, int Y2)

This method draws a line on the row-major two-dimensional array 'graphic' connecting X1,Y2 to X2,Y2. The line will be drawn by setting 1s in the array in the places where the line graphic is supposed to go.

For example:

```
int[,] graphic = { {0, 0, 0, 0, 0, 0, 0},
                   {0, 0, 0, 0, 0, 0, 0},
                   {0, 0, 0, 0, 0, 0, 0},
                   {0, 0, 0, 0, 0, 0, 0},
                   {0, 0, 0, 0, 0, 0, 0} };

LineDraw(graphic, 2, 1, 4, 3);
```

LineDraw will alter the array 'graphic' so that its contents are now this:

```
{ {0, 0, 0, 0, 0, 0, 0},
  {0, 0, 1, 0, 0, 0, 0},
  {0, 0, 0, 1, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0},
  {0, 0, 0, 0, 0, 0, 0} }
```

Another example on the same blank array as before:

```
LineDraw(graphic, 6, 1, 0, 4);
```

...would result in something like:

```
{ {0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 1, 1},
  {0, 0, 0, 1, 1, 0, 0},
  {0, 0, 1, 0, 0, 0, 0},
  {1, 1, 0, 0, 0, 0, 0} }
```

- Note that you shouldn't expect the array passed in to necessarily be [12,16] like the tank graphic is- if you need to read the array dimensions use GetSize(). You shouldn't need to, however.
- You can use whatever line drawing approach you like, as long as it draws reasonable-looking lines.

- Due to the nature of drawing a monochrome line, the line won't necessarily look perfect. Approximations are fine.

  Some references you may find helpful:

- https://en.wikipedia.org/wiki/Line_drawing_algorithm (https://en.wikipedia.org/wiki/Line_drawing_algorithm)
- https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm (https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm)

public abstract int GetTankArmour()

This abstract method gets the starting durability of this type of tank. This method must be overrided by the concrete class to return the durability rating for this type of tank. Your first/default tank should return 100 when this is called, which is also the default amount of damage inflicted by a standard weapon- this means an accurate hit will instantly destroy the tank, but an inaccurate shot will inflict partial damage. Other models of tank (represented by other classes inheriting from TankModel) may have larger or smaller values to compensate for increased/reduced weaponry.

Note that this value is distinct from the amount of durability remaining a tank on the battlefield has at the moment- this is because a TankModel represents a model of tank while a BattleTank represents an actual tank in combat, and it keeps track of its own durability- it just uses GetTankArmour() to initialise this value.

public abstract string[] WeaponList()

This abstract method returns an array containing a list of weapons that this tank has. This should be overrided in the deriving classes to return the weapons associated with a particular tank. Every tank must have at least one weapon. Each weapon is just a string naming the weapon; the code for handling the specific

Example implementation: `return new string[] { "Standard shell" };`

public abstract void ActivateWeapon(int weapon, BattleTank playerTank, Gameplay currentGame)

This abstract method is used to handle firing the specified weapon from the tank playerTank. This should be overrided in the deriving classes to enable those tanks to use weapons.

The weapon parameter is an int and it is based on the strings returned from WeaponList(). For instance, if WeaponList() returns an array with 2 strings in it, valid values for weapon are 0 and 1. If WeaponList() returns { "Standard shell", "Mining charge" } and the player selects "Standard shell" before firing, the value of weapon passed in will be 0.

The process for handling a standard implementation of ActivateWeapon() is:

- Use GetX() and Y() on the BattleTank to get the tank's coordinates.
- Convert the coordinates into floats and add on half the values of TankModel.WIDTH and TankModel.HEIGHT respectively to them, to get the position at the centre of the tank.
- Get the Opponent associated with the BattleTank passed to ActivateWeapon by using GetPlayer.

- Create a new Blast to reflect the payload of the weapon. Reasonable values to pass in are 100 (for damage), 4 (for explosion radius) and 4 (for earth destruction radius).
- Create a new Shell for the projectile itself. Pass in the X and Y coordinates of the centre of the tank, the angle and power (from BattleTank's GetTankAngle() and GetCurrentPower() respectively), a reasonable value for gravity (e.g. 0.01f), then the Blast and Opponent references we just got.
- Call Gameplay's AddWeaponEffect(), passing in the newly-created Shell.

The process may differ slightly when making different sorts of weapons- you may want to create two Shells with slightly different angles, for instance- but this is sufficient for a basic weapon.

public abstract class Opponent

This abstract class represents either a computer or human player. A player has an associated name, TankModel and colour, and also keeps track of the number of rounds won by that player. The AIPlayer and PlayerController inherit from Opponent.

public Opponent(string name, TankModel tank, Color colour)

This constructor passes in this Opponent's name, TankModel and colour. These three values are to be stored as private members of Opponent. In addition to those three private variables there is a fourth private variable keeping track of the number of rounds this player has won. This variable should be initialised to 0.

public TankModel GetTank()

This method returns the TankModel associated with this Opponent.

public string Identifier()

This method returns this Opponent's name.

public Color GetColour()

This method returns the colour associated with this Opponent.

public void AddScore()

This method increments the number of rounds won by this player by 1.

public int GetScore()

This method returns the number of rounds that have been won by this player.

public abstract void CommenceRound()

This abstract method is implemented by the AIPlayer and PlayerController classes, and is called each new round of battle. This is mainly for use by the AIPlayer, which may like to know that this is a new round and all the tanks have different locations now.

public abstract void NewTurn(GameplayForm gameplayForm, Gameplay currentGame)

This abstract method is implemented by the AIPlayer and PlayerController classes, and is called when it's this player's turn to move.

public abstract void HitPos(float x, float y)

This abstract method is implemented by the AIPlayer and PlayerController classes, and is called when the Shell launched by this player lands somewhere. This is mainly for the AIPlayer to use to adjust its aim for the next shot.

public class PlayerController : Opponent

This is a concrete class that extends the Opponent class, providing functionality specific to human-controlled Opponents.

PlayerController(string name, TankModel tank, Color colour) : base(name, tank, colour)

Constructor for the PlayerController class. All functionality is handled by Opponent so this method doesn't need to do anything.

public override void CommenceRound()

This method is called each round, but it doesn't need to do anything here.

public override void NewTurn(GameplayForm gameplayForm, Gameplay currentGame)

This method is called when it's this player's turn. Because this player is human-controlled, all this method should do is call EnableControlPanel() on the GameplayForm passed to this method.

public override void HitPos(float x, float y)

This method is called each time a shot fired by this player lands, but it doesn't need to do anything here.

public class BattleTank

This class represents a tank on the battlefield, as distinct from TankModel which represents a particular model of tank.

public BattleTank(Opponent player, int tankX, int tankY, Gameplay game)

This constructor stores player, tankX, tankY and game as private fields of BattleTank. It then gets the TankModel by using the Opponent's GetTank() method, then calls GetTankArmour() on it and stores this as the BattleTank's current durability. This will go down as the tank takes damage.

The constructor also initialises the angle, power and current weapon private variables, which start at 0, 25 and 0 respectively. Angle should be stored as a private float, while power and current weapon should be stored as private ints.

Finally, it should also call TankModel's CreateBMP method, passing in the colour (retrieved from Opponent's GetColour()) and current angle. The return value should then be stored as yet another private field.

public Opponent GetPlayer()

This returns the Opponent associated with this BattleTank.

public TankModel GetTank()

This returns the TankModel associated with this BattleTank. Hint: call GetTank() on the Opponent.

public float GetTankAngle()

This returns the BattleTank's current aiming angle. -90 means the turret is pointing to the left, 90 means the turret is pointing to the right, while 0 means the turret is pointing straight up.

public void SetAngle(float angle)

This method sets the BattleTank's current aiming angle.

public int GetCurrentPower()

This returns the BattleTank's current turret velocity. 5 is minimum power. 100 is maximum power.

public void SetPower(int power)

This method sets the BattleTank's current turret velocity.

public int GetWeaponIndex()

This returns the index of the current weapon equipped by the BattleTank.

public void SetWeapon(int newWeapon)

This method sets the BattleTank's current weapon.

public void Display(Graphics graphics, Size displaySize)

This method draws the BattleTank to graphics, scaled to the provided displaySize. The BattleTank's durability will also be shown as a percentage.

Here is an implementation you can use. Replace x, y and tankBmp with the names of the variables used to store the tank's x and y coordinates and the bitmap returned from CreateBMP. You will also need to set the startingArmour variable to the value initially retrieved from TankModel.GetTankArmour().

```
int drawX1 = displaySize.Width * x / Map.WIDTH;
int drawY1 = displaySize.Height * y / Map.HEIGHT;
int drawX2 = displaySize.Width * (x + TankModel.WIDTH) / Map.WIDTH;
int drawY2 = displaySize.Height * (y + TankModel.HEIGHT) / Map.HEIGHT;
graphics.DrawImage(tankBmp, new Rectangle(drawX1, drawY1, drawX2 - drawX1, drawY2 - drawY1));

int drawY3 = displaySize.Height * (y - Tank.HEIGHT) / Map.HEIGHT;
Font font = new Font("Arial", 8);
Brush brush = new SolidBrush(Color.White);

int pct = armour * 100 / startingArmour;
if (pct < 100)
{
    graphics.DrawString(pct + "%", font, brush, new Point(drawX1, drawY3));
}
```

public int GetX()

Returns the current horizontal position of the BattleTank.

public int Y()

Returns the current vertical position of the BattleTank.

public void Fire()

This causes the BattleTank to fire its current weapon. This method should call its own GetTank() method, then call ActivateWeapon() on that TankModel, passing in the current weapon, the **this** reference and the private Gameplay field of BattleTank.

public void InflictDamage(int damageAmount)

This inflicts damageAmount damage, reducing this BattleTank's durability by the given amount.

public bool Exists()

This returns true if this BattleTank's durability is greater than 0; otherwise it returns false. If the BattleTank is less than or equal to 0, it is considered destroyed and will not receive turns or be drawn to the screen.

public bool GravityStep()

Calling this method calls the BattleTank to fall down one tile, if possible. If the BattleTank moves as a result of this method, the method will return true. Otherwise it will return false.

- First, if Exists() returns false, this method returns false.

- Next, a reference to the Map is retrieved by calling the Gameplay method GetArena.
- The CheckTankCollide method of Map is then used to test the location one below the current BattleTank's position (in other words, CheckTankCollide(x, y + 1))
- If CheckTankCollide returns true, this method returns false.
- Otherwise, the BattleTank's vertical position is incremented by 1 and the durability is decreased by 1. This represents falling damage.
- If the BattleTank is now at the bottom of the screen (that is, its vertical position = Map.HEIGHT - TankModel.HEIGHT), its durability is reduced to 0. This is because reaching the bottom of the screen is considered to be equivalent to falling down a bottomless pit. This also allows tanks that are more geared to land destruction to be more effective in combat.
- Finally, the method returns true.

public class Blast : Effect

The Blast class is a type of Effect that represents the payload attached to a Shell. An Blast will inflict damage on tanks and destroy terrain within a radius.

public Blast(int explosionDamage, int explosionRadius, int earthDestructionRadius)

The Blast takes the explosion damage, explosion radius and earth destruction radius values it is passed and stores them as private fields.

Ignite(float x, float y)

This method detonates the Blast at the specified location. The x and y values are stored as private fields in Blast and another private field representing the Blast's lifespan is initialised to 1.0f. The reason this is performed in a method and not the constructor is so a Blast can be created and stored in a Shell before it is known where the Blast will actually appear.

public override void Tick()

This method reduces the Blast's lifespan by 0.05, and if it reaches 0 (or lower), does the following:

- Calls the Gameplay's InflictDamage() method with the Blast's x and y coordinates, explosion damage and explosion radius.
- Calls the Gameplay's GetArena() to get a reference to the Map and then call DestroyTerrain() on it, this time passing in Blast's x and y coordinates and the earth destruction radius.
- Calls the Gameplay's RemoveEffect(), passing in the **this** reference to remove the Blast from the list of active Effects.

Note that, to have access to the current Gameplay, you will need to add a **protected** reference to Gameplay in Effect.

public override void Display(Graphics graphics, Size displaySize)

This method draws one frame of the Blast. The idea is to draw a circle that expands, cycling from yellow to red and then fading out.

Here is an implementation you can use, although you will need to change the references to private fields to match the names you used:

```
float x = (float)this.x * displaySize.Width / Map.WIDTH;
float y = (float)this.y * displaySize.Height / Map.HEIGHT;
float radius = displaySize.Width * (float)((1.0 - lifetime) * explosionRadius * 3.0 / 2.0) / Map.
WIDTH;

int alpha = 0, red = 0, green = 0, blue = 0;

if (lifetime < 1.0/3.0)
{
    red = 255;
    alpha = (int)(lifetime * 3.0 * 255);
} else if (lifetime < 2.0 / 3.0)
{
    red = 255;
    alpha = 255;
    green = (int)((lifetime * 3.0 - 1.0) * 255);
} else
{
    red = 255;
    alpha = 255;
    green = 255;
    blue = (int)((lifetime * 3.0 - 2.0) * 255);
}

RectangleF rect = new RectangleF(x - radius, y - radius, radius * 2, radius * 2);
Brush b = new SolidBrush(Color.FromArgb(alpha, red, green, blue));

graphics.FillEllipse(b, rect);
```

public abstract class Effect

This abstract class represents a generic effect created by a BattleTank's attack. Both Blast and Shell come under this umbrella.

ConnectGame(Gameplay game)

This is called in Gameplay's AddWeaponEffect. The value of 'game' should be assigned to a **protected** field in Effect so that methods in Blast and Shell can use it.

public class Shell : Effect

The Shell class is a type of Effect that represents the a projectile or shell launched by a BattleTank. A Shell is launched at a certain angle and velocity and is affected by gravity and wind.

public Shell(float x, float y, float angle, float power, float gravity, Blast explosion, Opponent player)

This method constructs a new Shell. The x, y, gravity, explosion and player fields should all be stored in private fields of Shell.

Two more private fields are also initialised in this constructor: x velocity and y velocity. This refers to how much the Shell moves in 1/10 of a frame.

First, the angle needs to be converted into radians in the correct quadrant. Here is some code to do that:

```
float angleRadians = (90 - angle) * (float) Math.PI / 180;
```

Next, the magnitude of the movement vector needs to be calculated based on the power:

```
float magnitude = power / 50;
```

Finally, x velocity and y velocity are calculated using `(float) Math.Cos(angleRadians) * magnitude;` and `(float) Math.Sin(angleRadians) * -magnitude;` respectively.

public void Tick()

This method moves the given projectile according to its angle, power, gravity and the wind. The exact way this happens is rather detailed:

- Perform these steps 10 times:
  - Increase the Shell's x and y coordinates by the x and y velocity values respectively.
  - Increase the Shell's x coordinate by the wind speed divided by 1000.0f. The wind speed can be obtained by calling GetWindSpeed() on the Gameplay. (If you don't have a reference to Gameplay, you will need to add a **protected** one to the Effect class)
  - If the Shell has gone off the left, right or bottom sides of the screen (remember that the dimensions are in Map.WIDTH and Map.HEIGHT), call Gameplay's RemoveEffect() on **this** and return
  - Otherwise, if Gameplay's CheckHitTank(), when called with the Shell's x and y coordinates, detects a collision, do the following:
    - Call the Opponent's HitPos() method, passing in the x and y coordinates of the Shell.
    - Call the Blast's Ignite() method, again passing in the x and y coordinates.
    - Call the Gameplay's AddWeaponEffect() method, passing in the Blast.
    - Call the Gameplay's RemoveEffect() method, passing in **this**.
    - Return.
  - Increase the y *velocity* by gravity.

Display()

This method draws the Shell as a small white circle.

Here is an implementation you can use, although you will need to change the references to private fields to match the names you used:

```
float x = (float)this.x * size.Width / Map.WIDTH;
float y = (float)this.y * size.Height / Map.HEIGHT;
float s = size.Width / Map.WIDTH;

RectangleF r = new RectangleF(x - s / 2.0f, y - s / 2.0f, s, s);
Brush b = new SolidBrush(Color.WhiteSmoke);

graphics.FillEllipse(b, r);
```

public class GameplayForm : Form

This form is where most of the actual gameplay happens. You are provided with the outline, but to use this form you will need to add some elements to it.

First, add a Timer (System.Windows.Forms.Timer) to the form. Give it an interval of 20ms but leave it disabled for now.

Next, you will note that there are 2 Panels in the form- one named 'controlPanel' and one named 'displayPanel'. Leave the 'displayPanel' blank, but add these items to the 'controlPanel':

- A Label for holding the current player name (e.g. "Player 1")
- A Label to indicate that the label below it is displaying the wind speed ("Wind")
- A Label to display the current wind speed and direction (e.g. "0 W")
- A Label with the text "Weapon:"
- A Label with the text "Angle:"
- A Label with the text "Power:"
- A Label displaying the current power level (e.g. "20")
- A ComboBox used for selecting weapons
- A NumericUpDown used for selecting the angle. Set the Minimum and Maximum to -90 and 90 respectively, and set the Increment to 5.
- A TrackBar used for selecting the power level. Set the Minimum and Maximum to 5 and 100 respectively and set the LargeChange to 10.
- A Button with the text "Fire!"

These controls can be styled as necessary, but for now all you need is for the controls to be on the form and functional.

public GameplayForm()

The constructor for GameplayForm contains some existing code; first, some style options for reducing flicker and tearing, then the call to InitializeComponent() (which initialises everything added to the form in the Designer).

The following tasks need to be performed **before** the InitializeComponent() call:

- Set the private currentGame field to the Gameplay argument passed to the GameplayForm constructor.

- Set the backgroundImage and landscapeColour fields to a random image and colour respectively.

For best results, choose a random backgroundImage and then set a colour based on that image to ensure that you get a good level of contrast. These pairings were found to work well (select one random number from 0-3 and use it to select both the filename and colour):

```
string[] imageFilenames = { "Images\\background1.jpg",
                            "Images\\background2.jpg",
                            "Images\\background3.jpg",
                            "Images\\background4.jpg"};
Color[] landscapeColours = { Color.FromArgb(255, 0, 0, 0),
                             Color.FromArgb(255, 73, 58, 47),
                             Color.FromArgb(255, 148, 116, 93),
                             Color.FromArgb(255, 133, 119, 109) };
```

Once you have your image filename (those 4 images are the images provided with the Visual Studio project) call Image.FromFile() on it to create an Image that you can store in backgroundImage.

The following tasks need to be performed **after** the InitializeComponent() call:

- Initialise the graphic buffers for 'backgroundGraphics' and 'gameplayGraphics' by calling InitialiseBuffer() twice, setting backgroundGraphics to the return value of the first call and gameplayGraphics to the return value of the second call.
- Call the provided private method DrawBackground()
- Call DrawGameplay(). It doesn't exist yet, but add it to GameplayForm afterwards (private void DrawGameplay();)
- Call NewTurn(). It doesn't exist yet, but add it to GameplayForm afterwards (private void NewTurn();)

public void EnableControlPanel()

This method is used to enable the control panel so the (human) controller can control their tank. Called from PlayerController.NewTurn(). All it does is set controlPanel's Enabled property to true.

public void SetAngle(float angle)

This method alters the value of the UpDownNumeric used to control the angle, setting it to the provided value.

public void SetPower(int power)

This method alters the value of the TrackBar used to control the power level, setting it to the provided value.

public void SetWeapon(int weapon)

This method changes the selected item in the ComboBox, setting it to the provided value.

public void Fire()

This method is called both externally (by the computer player when it wants to fire) and by the 'Fire!' button when it is clicked. It does the following:

- Calls currentGame's GetCurrentPlayerTank() method to get a reference to the current player's BattleTank, then calls its Fire() method.
- Disables the control panel
- Enables the Timer

private void DrawGameplay()

This newly-created method is used to draw the gameplay elements of the screen (that is, everything but the terrain). It performs the following:

- Renders the backgroundGraphics buffer to the gameplayGraphics buffer: `backgroundGraphics.Render(gameplayGraphics.graphics);`
- Calls currentGame.DrawPlayers(), passing in gameplayGraphics.graphics and displayPanel.Size
- Calls currentGame.DrawAttacks(), passing in gameplayGraphics.graphics and displayPanel.Size

private void NewTurn()

This newly-created method is used to update form elements to reflect who the current player is. This involves doing a number of things:

- First, get a reference to the current BattleTank with currentGame.GetCurrentPlayerTank()
- Likewise, get a reference to the current Opponent by calling the BattleTank's GetPlayer()
- Set the form caption to "Tank Battle - Round ? of ?", using methods in currentGame to get the current and total rounds.
- Set the BackColor property of controlPanel to the current Opponent's colour.
- Set the player name label to the current Opponent's name.
- Call SetAngle() to set the current angle to the current BattleTank's angle.
- Call SetPower() to set the current turret power to the current BattleTank's power.
- Update the wind speed label to show the current wind speed, retrieved from currentGame. Positive values should be shown as E winds, negative values as W winds. For example, 50 would be displayed as "50 E" while -38 would be displayed as "38 E".
- Clear the current weapon names from the ComboBox.
- Get a reference to the current TankModel with BattleTank's GetTank() method, then get a list of weapons available to that TankModel.
- Add each weapon name in the list to the ComboBox.
- Call SetWeapon() to set the current weapon to the current BattleTank's weapon.
- Call the current Opponent's NewTurn() method, passing in **this** and currentGame.

Next, create ValueChanged (or SelectedIndexChanged) events for the NumericUpDown, TrackBar and ComboBox controls on the control panel. The methods tied to each of these events should call the appropriate BattleTank method (SetAngle(), SetPower(), SetWeapon()).

In addition, the ValueChanged event for the NumericUpDown should call DrawGameplay(); and displayPanel.Invalidate(); to refresh the screen. The ValueChanged event for the TrackBar should also update the label that keeps track of its power, setting it to the current power level.

Finally, create a Tick event for the Timer that you created earlier. A fair bit of logic needs to go into this Tick event as it is responsible for handling much of the animation and physics logic:

- First, call currentGame.ProcessWeaponEffects() to handle all Shells and Blasts.
- If it returned false (all attack animations have ended):
    - Call currentGame.GravityStep() to handle all the after-attack gravity cleanup.
    - Call DrawBackground() and DrawGameplay() to redraw everything after potentially moving terrain.
    - Call the displayPanel's Invalidate() method to trigger a redraw.
    - If it returned true (Some terrain/tanks were moved):
        - Return.
    - If it returned false (No terrain/tanks were moved):
        - Disable the timer.
        - Call currentGame's FinishTurn() method.
        - If it returned true, the round continues. Call NewTurn();
        - Otherwise, close the form by calling Dispose(), then currentGame's NextRound() method.
        - Return.
- Otherwise, attack animations are still ongoing. Call DrawGameplay() and displayPanel's Invalidate() method.
- Return.

public class MainMenuForm : Form

This is a simple title screen / main menu form, and the first thing you will see upon loading the game. Feel free to customise this as you wish, as long as it has a single Button.

A Click event for this button should be created and this should launch a new game of Tank Battle. Exactly what it does will depend on how many additional screens you implement, but the easiest way to get set up with a simple 1-round game for testing is to have the button do this:

```
Gameplay game = new Gameplay(2, 1);
Opponent player1 = new PlayerController("Player 1", TankModel.GetTank(1), Gameplay.GetColour(1));
Opponent player2 = new PlayerController("Player 2", TankModel.GetTank(1), Gameplay.GetColour(2));
game.CreatePlayer(1, player1);
game.CreatePlayer(2, player2);
game.BeginGame();
```

public class AIPlayer : Opponent

This is a concrete class that extends the Opponent class, providing functionality specific to computer-controlled Opponents. It is up to you how much effort you want to put into this AI- the simplest functionality is simply to have the AI set a random angle and power level and then fire, but more advanced functionality is possible.

AIPlayer(string name, TankModel tank, Color colour) : base(name, tank, colour)

Constructor for the AIPlayer class. It mostly exists to pass its parameters to the base constructor, but if the AI needs to do some initialisation work, that can be performed here.

public override void CommenceRound()

This method is called each round, allowing the AIPlayer to refresh its knowledge of the gameplay state, knowing that the tanks have been placed in a different order.

public override void NewTurn(GameplayForm gameplayForm, Gameplay currentGame)

This method is called when it's this player's turn. The player will need to call methods in gameplayForm such as SetWeapon(), SetAngle(), SetPower() and finally Fire() to aim and fire the weapon.

public override void HitPos(float x, float y)

This method is called each time a shot fired by this player hits, allowing the computer to adjust its aim. Note that this method is not always called- if the shot goes off-screen it will never hit anything and this method will not be called.

Additional Functionality

Beyond the basic level of functionality described in the class specification, you can also implement additional functionality to make your Tank Battle game more customisable and enjoyable. Additional functionality, if present and working, gives you extra points that, while they cannot allow your score for this assignment to go above 40%, can make up for potential weaknesses in other areas of your assignment.

It is advised that you do not attempt to add any of this additional functionality until your base game is solid. You will not receive points for having a player configuration screen if the game it leads to is broken. Additional functionality will only be assessed if the game's fundamentals are in sound working order.

Game and Player Setup Screens

Max additional marks: 2

Rather than having the Button on the MainMenuForm immediately start a game with a predefined set of players and tanks, have it load a form that allows you to enter in the number of players, number of rounds, as well as the name and tank chosen by each player, and whether a particular player is controlled by a human or a computer.

End-of-Game Leaderboard

Max additional marks: 1

At the end of a game, instead of simply going straight back to the main menu, show a leaderboard showing each player's score and listing the players in descending order, with a label announcing either a clear winner or that the game ended in a tie.

Multiple Tanks and Weapons

Max additional marks: 2

Create up to 6 different types of tanks, with different graphics, weapons etc. Experiment with different types of weapons. Create weapons that launch multiple projectiles, or launch one projectile that then splits off into multiple in the middle of its journey. This can be done by e.g. creating new classes that inherit from Shell.

AI Opponent

Max additional marks: 3

A very basic AI opponent is part of the standard requirements, but one that puts up a decent fight (while not being too overpowered) will net you some additional marks, depending on how impressive it is.

Keyboard controls

Max additional marks: 1

Using the mouse and entering in angle values for each shot is very tedious. You should be able to add keyboard controls to GameplayForm without breaking the specification.

Better resizing support

Max additional marks: 1

The existing display code is already designed to scale up the graphics it draws without any extra work, but the forms and controls themselves don't respond very well to resizing by default. Make it so that the GameplayForm can be resized and maximised, yet still remain playable.

Marking Criteria

More details about the marking criteria will be announced closer to the due date. As an overview, the total of 40% marks available for this assignment are divided up into automated testing (30%), with the remaining 10% marked with manual assessment. The additional functionality will also be graded through manual assessment, although marks awarded for it are 'extra' marks and not considered part of either the 30% or the 10%.

The automated testing marks are awarded by running your submitted assignment through a set of automated tests. These are not the same as the unit tests that you are provided with - the automated assessment will be far more comprehensive and will test every conceivable facet of your submitted assessment.

Out of the 10% of the marks decided through manual assessment, 5% will be looking at your code quality and commenting, while the other 5% will be making sure your game runs and doesn't have any glaring problems.

Optional Groups

This assignment is designed to be worked on by one person; however, groups of up to two people may be formed. More information on how to register your group will be available later. As each student gets their own version of the Visual Studio project and class specification, it will be up to you two to decide whose project and class specification will be used, and to ensure that you both have copies of them.

Submitted files:

Attach file:

submission.cs

Choose File No file chosen

```
The submission system is not available at this point.
Do not submit anything.
```

Declaration and submission

By submitting this form, I certify that:

- I have read, and understand, QUT Manual Of Policy and Procedures, Section C/5.3, Academic Integrity; and
- This submission is in full compliance with all provisions of QUT Manual of Policy and Procedures, Section C/5.3, Academic Integrity; and
- With the exception of support libraries provided to the class by the CAB201 teaching team, I am the sole author of all source code and attachments included in this submission.

Agree to these conditions: ☐

Submit

Transcript: