INTRO

In diesem Workshop geht es darum, ein kleines Spiel zu programmieren in dem der Spieler Gegner abschießen muss bevor sie den unteren Bildschirmrand erreichen. Das Spiel hat vier Bestandteile:
Den Spieler (`Player.cs`) selbst, den Gegner (`Enemy.cs`), Projektile die der Spieler schießt (`Projectile.cs`) und den Game Manager (`GameManager.cs`), der den Spielablauf bestimmt.

Probier das Spiel am besten nach jeder Aufgabe direkt aus, um zu schauen wie dein Code das Spiel verändert hat!

TEIL 1 – DER SPIELER

Im ersten Teil werden wir alles programmieren, was mit dem Spieler zu tun hat. Wenn du mit diesem Teil fertig bist sollte der Spieler sich mit den Pfeiltasten hin- und herbewegen und Projektile schießen können.

AUFGABE 1: SPIELERBEWEGUNG

Wo: `Player.cs`, `Update()`

Als erstes müssen wir den Spieler dazu bringen, sich zu bewegen wenn eine Taste gedrückt wird. Die `bool` Variablen `left` und `right` sind schon gegeben. Sie sind `true` wenn die jeweilige Taste gerade gedrückt wird und ansonsten `false`.
Die Position des Spielers wird in der `Vector3` Variable `position` gespeichert. Um den Spieler horizontal zu bewegen musst du den X-Wert von `position` erhöhen oder verringern.
Schreibe zwei Bedingungen, die den Spieler nach links bzw. rechts bewegen wenn die jeweilige Taste gedrückt wird!

Tipp: Verändere den X-Wert von `position` um `10.0f * Time.deltaTime`.
`Time.deltaTime` ist eine spezielle Variable von Unity. Sie enthält die Zeit seit dem letzten `Update()`. Wenn du damit multiplizierst ist es egal, wie schnell der PC ist auf dem das Spiel läuft – der Spieler wird sich immer gleich schnell bewegen.

AUFGABE 2: BESCHRÄNKUNG DER BEWEGUNG

Wo: `Player.cs`, `Update()`

Unser Spieler kann sich jetzt bewegen. Allerdings ist es dadurch auch möglich, dass der Spieler an der Seite aus dem Bildschirm fliegt. Das sollten wir verhindern.
Verhindere mithilfe von zwei weiteren Bedingungen, dass der Spieler links bzw. rechts aus dem Bildschirm fliegen kann!

Tipp:
Denke wie ein Computer – zum Beispiel „Wenn der X-Wert größer als 10 ist, setze den Wert zurück auf 10“

AUFGABE 3: SCHIEßEN

A)

Wo: `Player.cs`, `Update()` und `Shoot()`

Wenn die Pfeiltaste “hoch” gedrückt wird, soll der Spieler schießen. Hier ist wieder eine `bool` Variablen up gegeben, die dir sagt ob die Taste gerade gedrückt wurde.
Schreibe ein Bedingung ähnlich zu Aufgabe 1, so dass die `Shoot()`Methode aufgerufen wird, wenn der Spieler nach oben drückt.

Allerdings ist die `Shoot()` Methode im Moment noch leer – es würde also noch gar nichts passieren. Deshalb müssen wir darin ein Projektil erzeugen, welches der Spieler schießt. Verwende dazu die spezielle Unity-Methode `Instantiate`. Diese Methode braucht drei Inputs: 1. Was erzeugt werden soll, 2. Wo es erzeugt werden soll und 3. Wie es gedreht sein soll. Übergebe daher die folgenden Werte:

1. `Projectile`
2. `position` (Da wir wollen, dass das Projektil dort erscheint wo der Spieler gerade ist)
3. `Quaternion.identity` (Das bedeutet, dass das Projektil nicht gedreht sein soll)

Verwende die `Instantiate` Methode um ein Projektil zu erzeugen, wenn `Shoot()` aufgrufen wird!

B)

Wo: `Projectile.cs`, `Update()`

Der Spieler kann jetzt schießen. Aber wir haben noch gar nicht programmiert, wie sich die Projektile verhalten sollen!
Verändere deshalb die `position` Variable des Projektils, so dass es sich immer nach oben bewegt (Y-Wert).

Tipp: Verändere den Y-Wert von `position` um `Speed * Time.deltaTime`. Speed ist eine Variable, so dass wir später leicht verändern können wie schnell Projektile fliegen.

C)

Wo: `Projectile.cs`, `Explode()` und `Update()`

Wenn wir beim Schießen dauernd neue Projektile erzeugen, müssen sie auch irgendwann wieder verschwinden. Dafür verwenden wir die `Explode()` Methode.
Lösche das Projektil in `Explode()` indem du die Unity-Methode `Destroy(gameObject)` aufrufst! Wenn du magst kannst du mit `Instantiate` außerdem noch einen `explosionEffect` erzeugen.

Damit Projektile irgendwann explodieren und nicht ewig weiter fliegen, müssen wir nach einer bestimmten Zeit `Explode()` aufrufen. Deshalb musst du jetzt mithilfe von `TimeToLive` einen Timer bauen. `TimeToLive` fängt mit einem Wert von `2.0` an.
Verringere den Timer in `Update()` und wenn er 0 erreicht, rufe `Explode()` auf!

Tipp: `Time.deltaTime` kann hier wieder verwendet werden, um den Timer jedes `Update()` um die Zeit zu verringern, die seit dem letzen `Update()` vergangen ist.

TEIL 2 – GEGNER

Im zweiten Teil kümmern wir uns darum, dass Gegner auftauchen, abgeschossen werden können und dem Spieler Leben abziehen, wenn sie ihn berühren.

AUFGABE 4: GEGNERBEWEGUNG

Wo: `Enemy.cs`, `Update()`

Genau wie das Projektil soll sich auch der Gegner von selbst bewegen.
Verändere deshalb die `position` Variable des Gegners, so dass er sich abhängig von `Speed` immer nach unten bewegt (Y-Wert).

AUFGABE 5: KOLLISIONEN

A)

Wo: `Enemy.cs`, `Explode()` und `OnHitByProjectile(Projectile projectile)`

Gegner sollen auch explodieren können. Das funktioniert genau so wie bei Projektilen (Aufgabe 3 c)).
Fülle die `Explode()` Methode in `Enemy.cs` aus!

Damit wir Gegner abschießen können müssen wir programmieren was passiert, wenn ein Projektil einen Gegner trifft. Wenn das passiert wird automatisch `OnHitByProjectile(Projectile projectile)` aufgerufen.
Lass darin den Gegner und das Projektil explodieren, indem du jeweils ihre `Explode()` Methode aufrufst!

B)

Wo: `Player.cs`, `OnHitEnemy(Enemy enemy)`

Wenn ein Gegner den Spieler berührt, wird automatisch `OnHitEnemy(Enemy enemy)` aufgerufen. Der Spieler muss also in dieser Methode Leben verlieren. Außerdem sollte der Gegner bei der Kollision sterben, damit er dem Spieler nicht mehrere Leben abziehen kann.
Vervollständige die `OnHitEnemy(Enemy enemy)` Methode!

Tipp: Die Variable `Lives` wird benutzt um die Leben zu speichern. Verändere sie, aber pass auf dass sie nicht kleiner als 0 wird – man kann ja schließlich keine negativen Leben haben! Du kannst außerdem wieder einen visuellen Effekt mit `Instantiate` erzeugen. Verwende dafür die vorgefertigte Variable `loseLifeEffect` als ersten Input.

AUFGABE 6: VIELE GEGNER

Wo: `GameManager.cs`, `Update()`

Wir haben jetzt programmiert wie sich ein Gegner verhält. Aber bisher gibt es nur einen davon – das ist ein Bisschen wenig! Der Game Manager soll deshalb jetzt regelmäßig neue Gegner erscheinen lassen. Dafür kannst du wieder einen Timer verwenden (ähnlich wie in Aufgabe 3 c)).
Immer wenn der Timer abläuft, kannst du mithilfe der vorgefertigten Methode `SpawnEnemy` einen Gegner erscheinen lassen. `SpawnEnemy` braucht zwei Inputs: 1. Wo soll der gegner erscheinen? 2. Wie schnell soll der Gegner sein?
Damit die Gegner nicht immer an der selben Stelle auftauchen solltest du die Unity-Funktion `Random.Range(a, b)` verwenden. Sie gibt dir als Ouptut einen zufälligen Wert zwischen a und b. Damit kannst du eine zufällige Position ausrechnen, die du dann an `SpawnEnemy` weitergibst.
Programmiere einen Timer der immer wenn er abläuft einen Gegner an einer zufälligen Position erscheinen lässt!

Tipp: Verwende die Variable `spawnTimer` als Timer und veringere sie wie in 3 c). Wenn `spawnTimer` abgelaufen ist, setze die Variable wieder auf den Anfangswert, damit der Timer erneut ablaufen kann! Verwende beim Zurücksetzen am besten `SpawnCooldown`, damit nachher leicht verändert werden kann, wie schnell Gegner auftauchen.
Für die zufällige Position solltest du eine Variable vom Typ Vector3 erstellen und ihren X-Wert auf die zufällige Zahl setzen. Als Y-Wert kannst du 8 verwenden, damit die Gegner oberhalb des Bildschirms auftauchen.

TEIL 3 – SPIELLOGIK

Im letzten Teil werden wir das Projekt zu einem Spiel machen. Dafür werden wir den Punktestand des Spielers verändern und einbauen, dass man das Spiel verlieren kann. Außerdem werden wir den Schwierigkeitsgrad langsam ansteigen lassen, wenn der Spieler Punkte bekommt.

AUFGABE 7: PUNKTE

A)

Wo: `GameManager.cs`, `LosePoints(int points)` und `GetPoints(int points)`

Die Methoden `LosePoints` und `GetPoints` sollen immer dann aufgerufen werden wenn der Spieler Punkte bekommt oder verliert.
Verändere in ihnen die Variable `Score` um die Anzahl an Punkten, die in `points` steht!

Tipp: Pass auf, dass `Score` nicht kleiner als 0 wird! Außerdem kannst du in `LosePoints` die vorgefertigte Methode `screenShake.StartShake(0.5f)` aufrufen, damit der Bildschirm wackelt wenn der Spieler Punkte verliert.

B)

Wo: `Player.cs`, `OnHitEnemy(Enemy enemy)` und
`Enemy.cs`, `OnHitByProjectile(Projectile projectile)`

Jetzt müssen wir die beiden Methoden nur noch dann aufrufen, wenn der Spieler Punkte bekommen oder verlieren soll. Man bekommt Punkte wenn man einen Gegner abschießt und verliert Punkte, wenn man von einem Gegner getroffen wird.
Verändere mit `LosePoints` und `GetPoints` die Punktzahl!

Tipp: Da beide Methoden in `GameManager.cs` stehen musst du zum Aufrufen die vorgefertigte Variable `gameManager` benutzen.

C)

Wo: `GameManager.cs`, `Update()`

Es gibt noch einen Fall in dem der Spieler Punkte verlieren soll, nämlich wenn ein Gegner den unteren Bildschirmrand berührt. Dafür musst du alle Gegner anschauen und überprüfen, ob sie zu weit unten sind. `Enemy.Enemies` ist eine Liste aller Gegner, die du benutzen kannst.
Verwende eine Schleife um durch die Liste zu gehen. Wenn ein Gegner unterhalb des Bildschirms ist (Y kleiner als -5.0) zieh dem Spieler 500 Punkte ab und lass den Gegner explodieren!

AUFGABE 8: VERLIEREN UND NEUSTARTEN

Wo: `GameManager.cs`, `Update()`

Wenn man keine Leben mehr hat, sollte das Spiel vorbei sein.
Schreibe eine Bedingung die das Spiel beendet, wenn der Spieler keine Leben mehr hat!

Tipp: Um das Spiel zu beenden musst du die Methode `GameOver()` aufrufen. Speichere dir außerdem am besten in der `bool` Variable `gameOver`, dass das Spiel gerade vorbei ist, damit es neu gestartet werden kann!

Wenn das Spiel gerade verloren wurde, solle man es mit der Leertaste neustarten können. Neustarten bedeutet, dass wir alle Werte wieder auf ihren Startwert zurücksetzen müssen (`Score`, `player.Lives`, `SpawnCooldown`). Außerdem sollten alle Gegner die gerade aktiv sind explodieren (hier kannst du wieder eine Schleife verwenden wie in Aufgabe 7 c)).
Schreibe eine Bedingung die das Spiel neustartet, wenn das Spiel gerade vorbei ist (`gameOver`) und die Leertaste gedrückt wird!

Tipp: Beim Neustarten musst du `Restart()` aufrufen. Pass außerdem auf, dass du `gameOver` wieder auf `false` setzt!

AUFGABE 9: STEIGENDER SCHWIERIGKEITSGRAD

Wo: `GameManager.cs`, `LosePoints(int points)` und `GetPoints(int points)`

Damit das Spiel nicht so langweilig ist sollten mehr Gegner erscheinen, je mehr Punkte man hat. Wenn du bei Aufgabe 6 die Variable `SpawnCooldown` verwendet hast, kannst du sie einfach verändern um Gegner häufiger oder seltener erscheinen zu lassen.
Verändere den Schwierigkeitsgrad indem du `SpawnCooldown` immer ein wenig verringerst wenn der Spieler Punkte bekommt bzw. erhöhst wenn der Spieler Punkte verliert!

Tipp: Du kannst `SpawnCooldown` einfach um einen sehr kleinen Wert wie zum Beispiel 0.05 verändern. Pass dabei auf, dass `SpawnCooldown` nicht kleiner als 0.4 oder größer als 2.0 wird!