Schiffe-Versenken

Dies ist die komplette Dokumentation der Schiffe-Versenken Anwendung.

Beschreibung des Spiels

Im Spiel "Schiffe-Versenken" spielen zwei Spieler gegeneinander und versuchen jeweils die Schiffe des Gegners zu zerstören.

Zu Beginn wählen die Spieler jeweils aus, wo sie ihre Schiffe auf dem Feld platzieren wollen. Natürlich kann man nicht zwei Schiffe übereinander platzieren. Danach wechseln sich die Spieler ab und wählen immer eine Koordinate aus mit dem Ziel, die Schiffe des Gegners zu treffen. Der Spieler, welcher als Erstes alle Schiffe des Gegners zerstört, gewinnt das Spiel.

Starten des Spiels

Um das Spiel mit zwei Spielern zu spielen, müssen beide Spieler ein Command Prompt öffnen. Nun müssen beide Spieler die Pfade für das Programm einfügen. Mit dem ersten Pfad lässt sich Python aufrufen und mit dem zweiten Pfad können wir die Main des Schiffe-Versenken-Spiels aufrufen.

Als nächstes muss ein Spieler einen beliebigen Spielernamen und eine Portnummer eingeben und dann auf `Enter` drücken um das Spielfeld für diesen Spieler zu initialisieren. Der zweite Spieler gibt einen anderen Spielernamen und auch eine andere Portnummer ein.

Außerdem muss dieser auch die IP-Adresse des ersten Spielers eingeben und dessen Portnummer. Falls beide Spieler auf dem gleichen Rechner sind, verwendet man für die IP-Adresse `127.0.0.1` oder `localhost`.
Das Ganze würde dann so aussehen:

```
python.exe main.py <player_name> <local_port> [<remote_ip> <remote_port>]
```

Nun werden beide Spieler dazu aufgefordert, ihre verschiedenen Schiffe auf dem Spielfeld zu platzieren. Es gibt folgende Schiffarten:

```
Titanic, 4 Felder lang, ein Schiff verfügbar
Cruiser, 3 Felder lang, zwei Schiffe verfügbar
Yacht, 2 Felder lang, drei Schiffe verfügbar
Boot, 1 Feld lang, vier Schiffe verfügbar
```

Nachdem alle zehn Schiffe platziert wurden, werden die Spieler miteinander verbunden und können anfangen, gegeneinander zu spielen.

Aufbau

Unsere Anwendung enthält insgesamt drei Python-Dateien:
eine `main.py`, `board.py` und `ship.py`.

Main.py

Eine der wichtigsten Bestandteile dieser Anwendung ist die Interprozesskommunikation, denn nur durch diese ist es möglich, eine Verbindung zwischen zwei Spielern zu erstellen und diese gegeneinander spielen lassen zu können. Dies wurde in der Main mit Hilfe von Sockets gelöst. Sockets ermöglichen die Interprozesskommunikation in verteilten Systemen. Hierbei kann ein Benutzerprozess ein Socket vom Betriebssystem anfordern, mit dem er dann Daten - wie beispielsweise die Bomben in unserem Spiel - verschicken und empfangen kann.

Es werden hierfür zwei Prozesse benötigt - ein Client- und ein Server-Prozess. Deshalb haben wir auch eine Client- und eine Server-Klasse erstellt.

Hiermit wird ein Server-Socket erstellt, an einen Port gebunden, empfangsbereit gemacht und die Verbindungsanforderung akzeptiert (Zeile 23-27):

```
self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.socket.bind((str(self.context.ip), int(self.context.port)))
self.socket.listen()

(conn, addr) = self.socket.accept()
```

In der Client-Klasse muss ebenfalls ein Socket erstellt werden, welches dann mit dem Server-Socket verbunden wird. Nun können der Client- und Server-Prozess Daten senden und empfangen.

In den Zeilen 118-129 kann man dann sehen, wie im Spiel mit Hilfe des Clients und des Servers die Spieler die Schiffe angreifen. Mit `client.send(bomb)` wird die Koordinate des Angriffs vom Client an den Server geschickt. `result = server.receive()` empfängt den Angriff und `print(result)` informiert den Spieler, ob sein Angriff ein Schiff getroffen hat oder ob ein Schiff verfehlt wurde. `board_info = server.receive()` empfängt die Koordinaten des Angriffs auf dem Spielfeld und `print(board_info)` zeigt das Spielfield an mit dem Feld, das angegriffen wurde.

Des Weiteren wurde in der `main.py` umgesetzt, dass einige Informationen zu Beginn des Spiels angezeigt werden. Wenn der erste Spieler sich verbindet, wird ihm angezeigt, was für Schiffsarten er platzieren kann und wie viele er jeweils platzieren kann:

```
    if first_deployment:
        print("Place your Ships. You have 4 different types of ships:"
              "\n"
              "\n- Titanic. 4 units long. 1 available."
              "\n- Cruiser. 3 units long. 2 available."
              "\n- Yacht. 2 units long. 3 available"
              "\n- Boat. 1 unit long. 4 available. \n")
```

Damit die Spieler entscheiden und eingeben können, wo sie ihre einzelnen Schiffe auf dem Spielfeld platzieren wollen, haben wir eine `place_ships` Methode (Zeile 68) implementiert. Hier wird der Spieler dazu aufgefordert eine Koordinate einzugeben und dann eine Richtung, in die die Länge des Schiffs erweitert werden soll. Nach der Richtung wird nur gefragt, wenn das Schiff länger als eine Einheit ist, da Schiffe mit der Länge 1 logischerweise nicht erweitert werden müssen (Zeile 77-80):

```
    if health > 1:
        direction = input("Type a direction. (N: North, E: East, S: South, W: West):
")
    else:
        direction = 'S'
```

Die Schiffe werden dann jeweils für den ersten Spieler, mit Hilfe von Zeile 170-179, und für den zweiten Spieler, mit Hilfe von Zeile 214-223, platziert. Nachdem die Schiffe der beiden Spieler platziert werden, werden sie miteinander und können anfangen, das Spiel zu spielen.

Während des Spiels werden Angriffe mit der `handle_outgoing_move()` Methode gesendet, welche dann mit der `handle_incoming_move()` Methode verarbeitet und ausgewertet werden. Somit ist in der `handle_outgoing_move()` Methode zum Beispiel der Input implementiert worden, mit dem die Spieler die Koordinaten ihres Angriffs eingeben können (Zeile 117):

```
bomb_ = input("Send bomb (e.G. A4): ")
```

Wenn sich die eingegebene Koordinate auf dem Spielfeld befindet, wird diese vom Client an den Server geschickt (Zeile 119-121):

```
if 0 <= int(letter_to_num[bomb_[0]]) < 10 and 0 <= int(bomb_[1]) < 10:
    bomb = (letter_to_num[bomb_[0]] + "," + bomb_[1])
    client.send(bomb)
```

In der `handle_incoming_move()` Methode wird diese Koordinate vom Server empfangen und ausgewertet (Zeile 98-100):

```
bomb = server.receive()
bomb_coordinates = bomb.split(",")
result = board.bomb(int(bomb_coordinates[0]) - 1, int(bomb_coordinates[1]) - 1)
```

Anschließend wird die Antwort mit dem entsprechenden Ergebnis vom Client zurück an den Server geschickt, vom Server des anderen Spielers empfangen und das Ergebnis für alle angezeigt (Zeile 103-104; Zeile 126-129):

```
client.send(result)
client.send(board.info())

result = server.receive()
print(result)
board_info = server.receive()
print(board_info)
```

Zum Schluss befinden sich in den Methoden die Nachrichten, die die Spieler jeweils am Ende des Spiels informieren, ob sie gewonnen oder verloren haben (Zeile 107-109; Zeile 132-134):

```
if board.game_over():
    print("Game over, you lose :-(")
    sys.exit()

if "Over" in result:
    print("Game over, you win!")
    sys.exit()
```

Board.py

`Board.py` enthält eine Spielfeld-Klasse, in der das Spielfeld initialisiert wird und in der sich die Methoden befinden, welche alles durchführen, was mit dem Spielfeld zu tun hat.

In Zeile 25 befindet sich eine Methode namens `add_ship`, welche ein neues Schiff platziert und hierbei auch zum Beispiel überprüft, ob die Koordinate, wo das Schiff platziert werden soll, schon von einem anderen Schiff besetzt ist oder ob die Koordinate sich überhaupt auf dem Spielfeld befindet.

Eine weitere wichtige Methode ist die `bomb` Methode, die in Zeile 47 beginnt. Diese ermöglicht, dass die von einem Spieler ausgewählte Koordinate angegriffen wird und überprüft, ob sich auf dieser Koordinate ein Schiff befindet oder nicht (Zeile 51-57):

```
if status == EMPTY:
    # Vorbei am Schiff
    self.board[row][col] = MISS
    result = num_to_letter[int(row) + 1] + str(col + 1) + " -> Miss"
elif status == SHIP:
    # Schiff getroffen
    self.board[row][col] = HIT
```

Wenn der Angriff kein Schiff trifft, werden die Spieler mit einer `Miss` Nachricht informiert. Wenn jedoch ein Schiff getroffen wird, muss erstmal dieses Schiff auf dem Spielfeld gefunden werden. Danach muss überprüft werden, ob das Schiff bereits zerstört wurde. Falls nicht, muss untersucht werden, ob jetzt alle Schiffe zerstört sind, ob das eine Schiff zerstört ist oder ob nur ein einziger Treffer gelandet wurde. Je nachdem welche der Situationen zutrifft, werden die Nachrichten `Destroyed, Game Over`, `Destroyed` oder `Hit` für die Spieler angezeigt. Eine letzte Nachricht `Already bombed` wird herausgegeben, wenn die Koordinate bereits zuvor angegriffen wurde.

Um Informationen für das Spielfeld zu generieren und diese anzuzeigen, haben wir die `info` Methode implementiert. Hier wird zum Beispiel angegeben, was man auf dem Spielfeld sehen soll, falls ein Angriff ein Schiff trifft:

```
if self.board[r][c] == HIT:
    row += 'X '
```

Wenn ein Spieler also ein Schiff trifft, wird dies angezeigt, indem ein `X` auf die getroffene Koordinate gesetzt wird. Ein `o` taucht auf dem Spielfeld auf, wenn man ein Schiff verfehlt und noch nicht attackierte Koordinaten werden mit einem `-` dargestellt.

Ship.py

`Ship.py` ist mit Abstand die kleinste der drei Dateien und enthält lediglich eine Schiff-Klasse. Diese Klasse hat die Aufgabe, die Schiffe der Spieler zu initialisieren. Um Schiffe zu platzieren, müssen die Spieler eine Koordinate des Spielfeldes eingeben und danach eine Richtung. Die ursprüngliche Koordinate wird dann um die Länge des Schiffes in die eingegebene Richtung erweitert.

Dies wurde in den Zeilen 9-17 des Codes umgesetzt. Wenn ein Spieler also sein Schiff zum Beispiel in Richtung Norden platzieren will, wird die Koordinate der x-Achse um eine Einheit verkleinert und die Koordinate der y-Achse bleibt bestehen:

```
    for k in range(size):
        if direction == 0:
            self.coordinates.append((origin[0] - k, origin[1]))
```

Das Gleiche wurde für die Richtungen Süden, Westen und Osten implementiert.

Herausforderungen beim Coden

Die erste Herausforderung war es erstmal, Python zu lernen. Dies haben wir zum Beispiel mit Hilfe von verschiedenen Videos und Websiten gemacht. Weiter unten sind ein paar Links zu finden, mit denen wir uns die Grundlagen aneignen konnten.

Grundsätzlich war es nicht das Schwerste, die Programmiersprache Python zu lernen. Die Grundlagen hatten wir relativ schnell gelernt, da wir alle durch das Modul OOP schon mit den Grundprinzipien der Programmierung vertraut waren und somit nur die Syntax lernen mussten, welche uns bei Python sogar viel einfacher fiel, als bei beispielsweise Java.

Als wir dann anfingen, uns langsam an die Entwicklung des Spiels ranzutasten, wussten wir anfangs nicht wirklich, wo wir anfangen sollten und hatten etwas Schwierigkeiten, unsere neu erlernten Python-Kenntnisse direkt im Schiffe-Versenken-Spiel umzusetzen. Nach kurzen Startschwierigkeiten stürzten wir uns einfach ins Geschehen und fingen an, erstmal ohne OOP und Konzept eine 2D-List zu erstellen, und eine Funktion zu programmieren, die diese in einem passenden Format (1-10, A-J) ausgibt. Als das erstmal getan war, ging es auch schneller voran, da man nun einen Anfang hatte und wusste wo man anbinden kann. Schnell kamen dann Platzierungsmethoden für die Schiffe, Überlappungsüberprüfungen, und mehr dazu, sodass wir relativ schnell schon

mal ein Spiel hatten, welches man zwar noch nicht online spielen konnte, aber schon ein wenig gegen den Computer spielen konnte.

Nicht alles funktionierte, aber das war nicht das Hauptproblem: Das Spiel war in katastrophaler Ordnung in Form von verschachtelten If-Abfragen und ohne wirkliche Objektorientierung gecodet worden und das störte uns ziemlich, da wir nicht so einen guten Überblick hatten und vieles im Code sehr repetitiv war. Also beschlossen wir, die Objektorientierung nochmal zu vertiefen und schauten uns wieder etliche Videos an und recherchierten auf Stackoverflow, während wir parallel versuchten, den Code zu optimieren.

Eine der größten Herausforderungen für uns war es, die Interprozesskommunikation mit Hilfe von Sockets in unsere Anwendung zu implementieren, damit überhaupt zwei Spieler gegeneinander spielen können. In den Vorlesungsfolien für "Betriebssysteme und Rechnernetze" befindet sich ein Beispiel für Sockets via TCP, welches allerdings in C geschrieben wurde. Trotzdem konnte dieses Beispiel uns ein wenig weiterhelfen. Um nun herauszufinden, wie wir die Sockets in unsere Python-Anwendung einbauen können, haben wir wieder im Internet recherchiert. Hier haben wir ein paar Beispiele gefunden, wodurch uns das Ganze verständlicher wurde.

Quellen

https://www.youtube.com/watch?v=JeznW_7DlB0&t=1044s

https://www.youtube.com/watch?v=HGOBQPFzWKo&t=38s

https://www.youtube.com/watch?v=rfscVS0vtbw&t=14234s

https://www.w3schools.com/python/default.asp

https://www.youtube.com/watch?v=C7CpfL1p6y0&t=380s

https://www.youtube.com/watch?v=uagKTbohimU

https://docs.python.org/3/library/socket.html

https://docs.python.org/3/howto/sockets.html

https://pythonprogramming.net/server-chatroom-sockets-tutorial-python-3/