Der Multiplayer

1. Allgemeines Konzept

Die generelle Struktur des Server/Client Systems ist überwiegend dezentral. Der Server selber fungiert über weite Teile nur als Überbringer von Nachrichten. Die Nachrichten selber bestehen aus Strings, die bei key-, upgrade-, oder Todesevents versendet werden. Stellt man ein "!" seiner Nachricht voran, so behandelt der Server die Nachricht gesondert. Verschiedene Nachrichten werden zB an alle Spieler, andere wie Bewegungsübermittlungen jedoch an alle außer dem eigenen Spieler gesendet. Jedoch werden alle Methoden ausschließlich lokal (vom Client) ausgeführt. Dies war meiner Meinung nach die am simpelsten zu implementierende Möglichkeit, birgt jedoch, im Gegensatz zur serverseitigen Implementierung, verstärkt die Gefahr der Desynchronisation.

2. Konzepte zur Prävention eines asynchronen Spielverlaufs

2.1 Schon bei der Initialisierung einer Partie werden alle Spieler, die über das Netzwerk gesteuert werden, so modifiziert, dass sie nicht mehr durch Kollision mit einem tödlichen Objekt sterben können. In der initComponent()-Methode der MultiplayerUnit wird für diese einfach die remotePlayer Variable auf true gesetzt, welche dann in der update()-Methode des Playerobjekts das Sterben unterbindet. Getötet werden kann nur der eigene Spieler, welcher dann über das Netzwerk an die anderen Clients die Nachricht seines Todes verschickt.
Auf diese Weise wird ausgeschlossen, dass der Spieler aufgrund von Latenzen auf einigen Clients nicht getötet wird.
2.2 Durch Latenzen kommt es zu einem leicht versetzen Spielverlauf, was besonders die Bewegung der Spieler betrifft. Das größte Problem ist jedoch, dass je nach Karte, die langsame update()-Methode der Karte den Spielfluss stark einschränken kann. Dadurch wird die Schleife, die die Frames aktualisiert nicht mehr rechtzeitig fertig, was wiederum dazu führt, dass das Spiel bei dem betroffenen Spieler langsamer läuft.
Bei jedem key event werden Koordinaten mit gesendet, um das Spiel möglichst synchron zu halten, jedoch kann dies zu heftigen Positions-Rucklern der remote gesteuerten Spieler führen.
2.3 Der meiste Synchronisationsaufwand muss jedoch bei den Upgrades betrieben werden. Da diese per Chance gespawned werden, wird kurzerhand Spieler 1 (zufälligerweise auch der Host) als einziger Spieler dazu ermächtigt, Upgrades zu spawnen. Spawned ein Upgrade, erfährt die MultiplayerUnit dies über ein listener interface. Diese sendet nun eine Nachricht an den Server, der alle anderen Clients benachrichtigt, welche dieses Upgrade nun auch auf der Karte spawnen. Bleibt noch zu erwähnen, dass der Server die Nachricht vorher analysiert und die mitgesendete ID dieses Upgrades in eine Liste einträgt.
Läuft ein Spieler (nur der selbst gesteuerte Spieler) nun über eines der

Upgrades, nimmt er dieses nicht direkt auf, sondern benachrichtigt zunächst den Server. Dieser vergleicht die ID des Upgrades mit den IDs in seiner Liste und falls diese vorhanden ist, löscht er das entsprechende Element aus der Liste und echot an alle Clients zurück, dass das Upgrade vom entsprechenden Spieler aufgenommen werden soll. Liefe nun ein anderer Spieler über das selbe Upgrade noch bevor der Server Bescheid sagen kann, dass das Upgrade entfernt und oben erwähntem Spieler gegeben werden soll, so würde der Server allerdings die ID des Upgrades nicht mehr in seiner Liste vorfinden und dementsprechend die Nachricht nicht weiterleiten. Um hier Konflikte zwischen den verschiedenen Threads zu unterbinden, ist diese spezielle Methode handleUpgradeMsg synchronized.

3. Aufbau des Servers

Der Server wird von der NetworkConnectorUnit aufgerufen, welche ihm den Port, den Namen der zu spielenden Karte und die maximale Spieleranzahl übergibt. In seiner run()-Methode versucht dieser nun eingehende TCP-Verbindungen zu akzeptieren. Ist die Anzahl maximaler Spieler erreicht oder das Spiel gestartet worden, bricht der Server aus der Schleife aus und schließt den Server Socket. Die akzeptierten Sockets werden direkt einem an dieser Stelle erzeugtem Thread, zusammen mit der entsprechenden Spielernummer und der Referenz auf diesen Server, als Parameter übergeben und in einer Liste abgespeichert. Diese "ToClientSocket"-Threads lauschen permanent auf eingehende Nachrichten des ihnen zugewiesenen Clients und übergeben diese dann einer Server Methode, die in der Mehrheit der Fälle einfach nur die Nachricht an die restlichen Clients weitergibt.
Die hierzu erforderliche Methode distributeMessage nimmt einen String oder eine Zahl und einen String entgegen. Im ersten Fall sendet der Server den String an alle Clients, im zweiten hingegen an alle außer an den Client, dessen Index man angegeben hat. Hierzu läuft die Methode die Array List toClientSockets durch. In der Schleife holt sich die Methode das ReentrantLock und den OutputStream vom entsprechenden Objekt. Danach kann die Nachricht geschrieben werden.

4. Aufbau des Clients

Wie bereits erwähnt, werden nur key events versendet. Der Netzwerktraffic ist dadurch extrem gering, jedoch müssen bei jeder Eingabe Koordinaten mitgeschickt werden, um die Positionen der Spieler synchronisieren zu können. Im ungünstigsten Fall würde sonst zB eine Bombe auf ein falsches Feld gesetzt. Auch der Client besitzt einen eigenen Thread, der den Socket hält (der ReadFromHost-Thread). Jedoch ist dieser eine äußere Klasse, die separat von der eigentlichen MultiplayerUnit bereits von der NetworkConnectorUnit gestartet wird, da für die Lounge schon Netzwerkkommunikation bestehen muss. Die Lounge bringt für die MultiplayerUnit die Anzahl an Mitstreitern, die eigene Spielernummer und die zu spielende Map in Erfahrung. Natürlich sorgt sie auch

dafür, dass die Spieler gleichzeitig ins eigentliche Spiel gelangen. Die Anzahl der Mitstreiter ist wichtig, da eine Mulitplayer Map mehr Spieler enthalten kann, als eigentlich spielen möchten. Die überflüssigen Spieler werden dann beim Initialisieren entfernt.

5. Exception Handling

Selbstverständlich ist der Netzwercode absturzsicher. Jeder der erstellten Sockets hat einen Socket-Timeout, die Timeout Exception wird vor der IOException abgefangen (dort wird nichts unternommen, es soll einfach der normale Programmablauf weiter geführt werden). Wenn dann eine "richtige" IOException auftritt, wird der Thread sanft beendet, indem aus der Schleife ausgebrochen wird. Fehler beim Schreiben werden gar nicht behandelt, es wird vollstens auf die korrekte Behandlung der Exceptions in der read-Methode vertraut.
Zu den Threads im Server-Programm ist noch zu sagen, dass diese den Spieler bei den anderen Clients töten, bevor sie sich selbst aus der Liste im Server löschen und dann terminieren.