Hochschule Aalen

Projektbericht
Studiengang : Informatik

Belt Wars

von

Sebastian Flum, Markus Harder, Daniel Heizenreider, Patrick Kneifel, Tomy Nguyen,

76855, 70747, 70772, 29224, 74750

Betreuender Professor: Prof. Dr. Carsten Lecon

Einreichungsdatum : 10. Februar 2021

Inhaltsverzeichnis

1 Einleitung

Im Rahmen des Vorlesungs-Moduls SSpieleprogrammierungän der Hochschule Aalen sollte in einer Gruppe ein Computerspiel konzipiert und entwickelt werden. Wir haben uns dafür entschieden, ein rundenbasiertes 2D-Spiel zu entwerfen, welches dem bekannten Videospiel "Worms"ähnelt.
Das Setting des Spiels handelt von Raumschifffraktionen, die im Asteroidengürtel des Sonnensystems um Ressourcen streiten. Es wird aus der 2D-Vogelperspektive gespielt und die Spielwelt wird aus den Schiffen und Asteroiden gebildet. Pro Runde darf nur mit einem Schiff interagiert werden d.h. das Schiff bewegen und einmal schießen. Die Schiffe besitzen als Waffen eine Machine-Gun, einen Rocket-Launcher oder einen Laser-Beam verwenden. Während des Spiels kann zwischen den Runden zufällig das Support-Ship-Event auftreten. Dieses platziert auf der Map ein Item, welches von den Schiffen aufgesammelt werden kann. Zudem existieren als Spielmodi ein Singleplayer mit einer KI als Gegner, ein Single-Device-Multiplayer, bei dem zwei Spieler an einer Gerät spielen, sowie ein Multiplayer in einem lokalen Netzwerk.
Ziel des Spieles ist es, alle gegnerischen Raumschiffe zu zerstören.
Als Game-Engine für unser Spiel benutzten wir Unity und die Zielplattform sind PC.

2 Grundlagen

2.1 C#

Die Programmiersprache C# verwendet Konzepte der Programmiersprachen Java, C, Haskell, C sowie Delphi und zählt zu den objektorientierten Programmiersprachen. Sie ist eine universelle Programmiersprache, die ursprünglich entwickelt wurde, um die Entwicklung von Windows-Anwendungen zu vereinfachen. Vor allem die Ähnlichkeiten zwischen C#, C sowie Java erleichtern einen Einstieg, wenn man einen Hintergrund in objektorientierten Sprachen besitzt. Durch das .NET Framework verfügt C# über eine umfangreiche Klassenbibliothek, die zum Beispiel die Erstellung grafischer Oberflächen unterstützt. Weitere Eigenschaften sind Exception Handling und verschiedene Arten von Polymorphismus[1].

2.2 Unity

Unity ist eine Laufzeit- und Entwicklungsumgebung für Spiele und zu Ihren Zielplattformen gehören PCs, Spielekonsolen, mobile Geräte und Webbrowser. Im Unity Editor ist ein Hauptfenster zu sehen, der die 3D- oder auch 2D-Szene darstellt. Menüs und verschiedene Formulare erlauben die Manipulation von Kamera und Szene. Teile dieser Szene(GameObjects) können skaliert, gedreht sowie verschoben werden und diese GameObjects können Komponenten z.B. Materialien, Skripte und physikalische Eigenschaften zugeordnet werden. Skripte sind notwendig, um Spielablauf und -logik zu beschreiben. Komplexe Komponenten (Assets) z.B. 3D-Modelle, Texturen, Sounds und Animationen, die in anderen Programmen erstellt wurden, können per Drag&Drop importiert werden. Die Skripte und Assests können in Prefabs zusammengefasst werden, um gleichartige Objekte mehrfach in einer Szene verwenden. In der Game-View werden Verhalten des Spiels und die grafische Darstellung simuliert. Unity kann durch eingebundene Plug-ins um weitere Funktionen und Assets erweitert werden[7].
Zum Beispiel:

- Werkzeuge für Partikeleffekte
- Terrain-Modellierer

- EditorVR

2.3 Mirror Networking

Die Multiplayer High Level API (HLAPI) von Mirror ist ein System zum Aufbau von Multiplayer-Funktionen für Unity-Spiele. Es basiert auf der top of the lower level transport real-time communication layer und erledigt viele der allgemeinen Aufgaben, die für Multiplayer-Spiele erforderlich sind. Während die Transportschicht jede Art von Netzwerktopologie unterstützt, ist das HLAPI ein serverautorisierendes System. Obwohl einer der Teilnehmer gleichzeitig Client und Server sein kann, ist kein dedizierter Serverprozess erforderlich. In Verbindung mit den Internet Services können Multiplayer-Spiele mit wenig Aufwand über das Internet gespielt werden[2].

2.3.1 Server und Host

Multiplayer-Spiele mit einem Mirror-System umfassen:

- **Server:** Der Server ist eine Instanz des Spiels, mit der sich alle anderen Spieler verbinden, wenn sie zusammen spielen möchten. Dieser verwaltet häufig verschiedene Aspekte des Spiels, z. B. das Halten der Punktzahl und das Zurücksenden dieser Daten an den Client.
- **Clients:** Die Clients sind Instanzen des Spiels, die eine Verbindung von verschiedenen Computern zum Server herstellen. Clients können eine Verbindung über ein lokales Netzwerk oder online herstellen.

Der Server kann entweder ein “Dedicated Server” oder ein "Host-Serverßein:

- **Dedicated server:** Dies ist eine Instanz des Spiels, die nur als Server ausgeführt wird.
- **Host server:** Wenn kein dedizierter Server vorhanden ist, spielt einer der Clients auch die Rolle des Servers. Dieser Client ist der "Host-Server". Dieser erstellt eine einzelne Instanz des Spiels, die sowohl als Server als auch als Client fungiert.

2.3.2 Instanzierung und Spawn

Bei diesem Multiplayer-System muss der Server selbst Spielobjekte ßpawnen", damit sie im vernetzten Spiel aktiv sind. Wenn der Server Spielobjekte erzeugt, wird die

Erstellung von Spielobjekten auf verbundenen Clients ausgelöst. Das Spawning-System verwaltet den Lebenszyklus des Spielobjekts und synchronisiert den Status des Spielobjekts basierend darauf, wie man das Spielobjekt eingerichtet hat[2].

2.3.3 Players und Local Players

Wenn ein neuer Spieler dem Spiel beitritt, wird das Spielobjekt dieses Spielers zu einem "lokalen SpielerSpielobjekt auf dem Client dieses Spielers, und Mirror verknüpft die Verbindung des Spielers mit dem Spielobjekt des Spielers. Mirror ordnet jeder Person, die das Spiel spielt, ein Spieler-Spielobjekt zu und leitet Netzwerkbefehle an dieses einzelne Spielobjekt weiter. Ein Spieler kann keinen Befehl für das Spielobjekt eines anderen Spielers aufrufen, sondern nur für sein eigenes[2].

2.3.4 Nützliche Funktionen

Von Mirror im Projekt genutzte Komponenten sind folgende:

- **NetworkIdentity:** Damit ein Objekt im Netzwerk überhaupt existiert benötigt es diese Komponente. Diese wird vollständig von Mirror verwaltet und dient zur eindeutigen Identifikation. Zudem wird hierüber die Authorität über ein Objekt verwaltet. [3]
- **NetworkTransform:** Positionen von Objekten im Netzwerk werden hiermit von Mirror synchronisiert. [4]
- **SyncVars:** Variablen deren Werte von Mirror synchronisiert werden. Um direkt auf eine Änderung zu reagieren bietet Mirror hier einen speziellen Hook um eine gewünschte Funktion aufzurufen.[6]
- **Commands:** Ein Client sollte nicht in der Lage sein, alle Aktionen durchzuführen. Möchte er also eine Aktion durchführen, die nur der Server durchführen kann, so muss er ein Command verwenden. [5]
- **ClientRPCs:** Eine Funktion, welche auf jedem Client ausgeführt wird, aber nur vom Server aufgerufen werden darf.[5]

3 Problemanalyse, Lösungskonzepte und Implementierung

3.1 Map Creation

Problemanalyse

Da das Spiel von der 2D-Variante der Videospielreihe „Worms“ inspiriert ist, sollte das Spielfeld, ähnlich wie in der Vorlage, zufällig generiert werden und zerstörbar sein. Gleichzeitig sollte die Karte Deckung für die Figuren, in diesem Fall die Raumschiffe, bieten.

Die Nachbildung eines Worms-ähnlichen Terrains stellte sich nach einiger Recherche als sehr aufwändig dar, vor allem auch im Hinblick auf den geplanten Netzwerk-Spielmodus. Die Herausforderung einer zufällig generierten zusammenhängenden Karte, bei der einzelne Fragmente unterschiedlicher Größe je nach der Wirkung des einschlagenden Waffenprojektils zerstört werden, schien hinsichtlich der ohnehin herausfordernden Synchronisationsproblematik im Netzwerkspiel als kaum lösbar innerhalb eines Studiensemesters.

Lösungskonzept und Implementierung

Aus diesem Grund wurde entschieden, für die Map-Creation eine pragmatischere Lösung zu finden, die der Vorlage dennoch möglichst nahekam. Anstatt eines zusammenhängenden Terrains besteht die Spielkarte nun aus einzelnen Asteroiden, die ein Asteroidenfeld bilden und somit als Nebeneffekt die Hintergrundgeschichte für das Spiel „Belt Wars“ liefern, nämlich im Groben der Kampf von Erde und Mars um die Ressourcen des Asteroidengürtels unseres Sonnensystems (siehe Abbildung 3.1).

Die Asteroiden werden zufällig aus unterschiedlichen existierenden Prefabs (insgesamt sechs Asteroiden-Varianten) ausgewählt und zufällig auf der Karte platziert. Dabei wird darauf geachtet, dass ein Asteroiden-Objekt nicht an derselben Stelle eines bereits auf der Karte existierenden Objekts platziert wird. Zudem wird auch die Rotation des Asteroiden in Z-Richtung zufällig generiert. Um die Map zerstörbar zu machen, können die Asteroiden zwar zerstört werden, allerdings sind sie in der Lage, je nach Waffenstärke, mehrere Treffer auszuhalten, bevor sie in einer

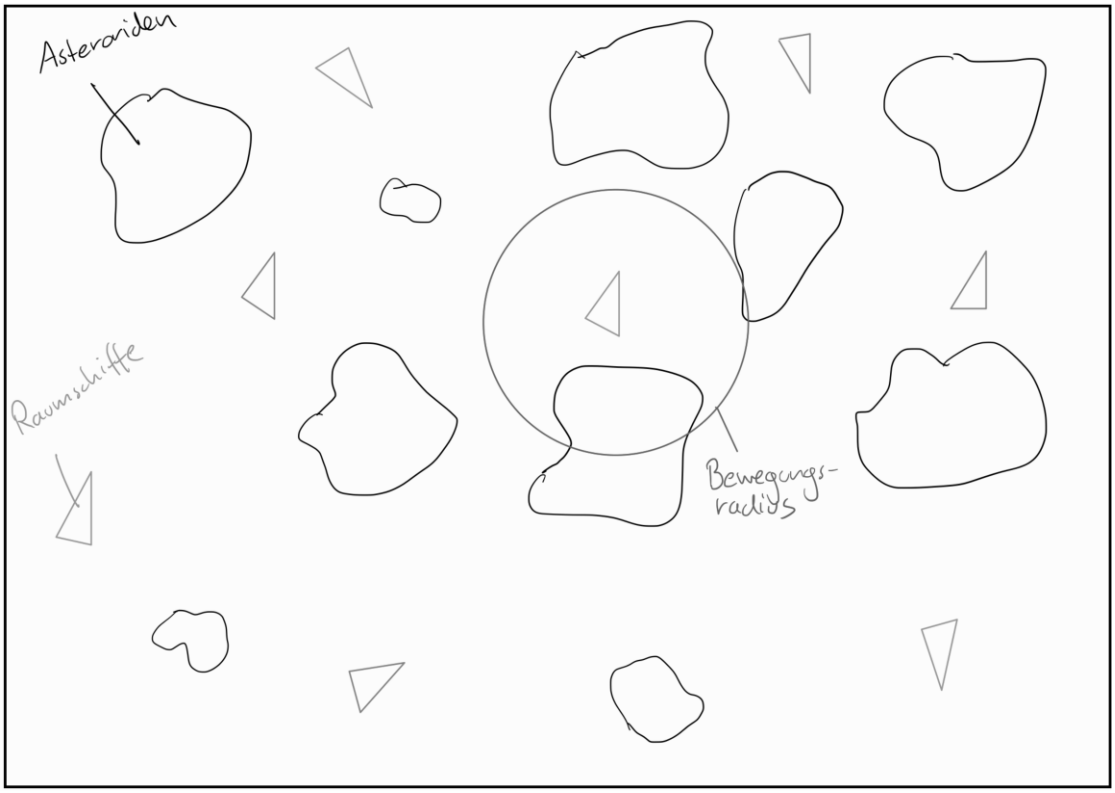

Abbildung 3.1: Erste Skizze Map

Explosionsanimation zerstört werden.

Ähnlich wie mit den Asteroiden wird mit den Raumschiffen verfahren, die vom Spieler gesteuert werden. Auch Sie werden zufällig auf der Karte an noch freien Stellen platziert und erhalten eine zufällige Rotation in Z-Richtung (siehe Abbildung 3.2).

```
103    private void spawnShipRandom(string path, int index)
104    {
105        GameObject ship = Resources.Load(path) as GameObject;
106        CameraMeasurements camera = new CameraMeasurements();
107
108        if (ship == null) Debug.Log("Ship resource is null!");
109
110        bool collision;
111        Vector3 spawnLocation;
112
113        do
114        {
115            float x = Random.Range(camera.getHorizontalMin() + borderDistance, camera.getHorizontalMax() - borderDistance);
116            float y = Random.Range(camera.getVerticalMin() + borderDistance, camera.getVerticalMax() - borderDistance);
117
118            spawnLocation = new Vector3(x, y, 0);
119
120            collision = checkCollision(spawnLocation, ship);
121        } while (collision);
122
123        float rotation = Random.Range(0, 360);
124        GameObject clone = (GameObject) Instantiate(ship, spawnLocation, Quaternion.Euler(0, 0, rotation));
```

Abbildung 3.2: Zufällige Platzieung von Schiffen

Um den Eindruck eines im Weltraum befindlichen Areals zu verstärken, werden auch Hintergrundbilder, welche unterschiedliche Weltraumszenarien darstellen, in die Szene eingefügt. Dabei wird aus einer Ansammlung von insgesamt acht Hintergrundbild-Prefabs zufällig eines ausgewählt. Zusätzlich wird zu Beginn vor jedem Spiel eine Abbildung eines Planeten oder Mondes aus einer Auswahl von neun Abbildungen eine ausgewählt und zufällig auf der Karte platziert und zufällig skaliert. Ebenso wird mit dem Planetenschatten, des platzierten Planeten verfahren. Er wird an die Skalierung und die Position des Planeten angepasst. Nur die Größe des Schattens, als Abbildung, wird hier zufällig ausgewählt.

Als Resultat wird zu jedem Spielbeginn ein einzigartiges Spielfeld generiert (siehe Abbildung 3.3), wodurch zahllose unterschiedliche Spielszenarien und Geschichten entstehen können.

3.2 Singleplayer und KI-Gegner

Problemanalyse

Zu Projektbeginn war es nicht geplant, einen Singleplayer-Modus mit einem KI-Gegner zu implementieren, sondern lediglich einen Single-Device-Multiplayer- und einen Netzwerk-Multiplayer-Modus. Im Laufe des Projekts stellte sich jedoch die Erkenntnis heraus, dass für ein komplettes Spielerlebnis auch ein Singleplayer

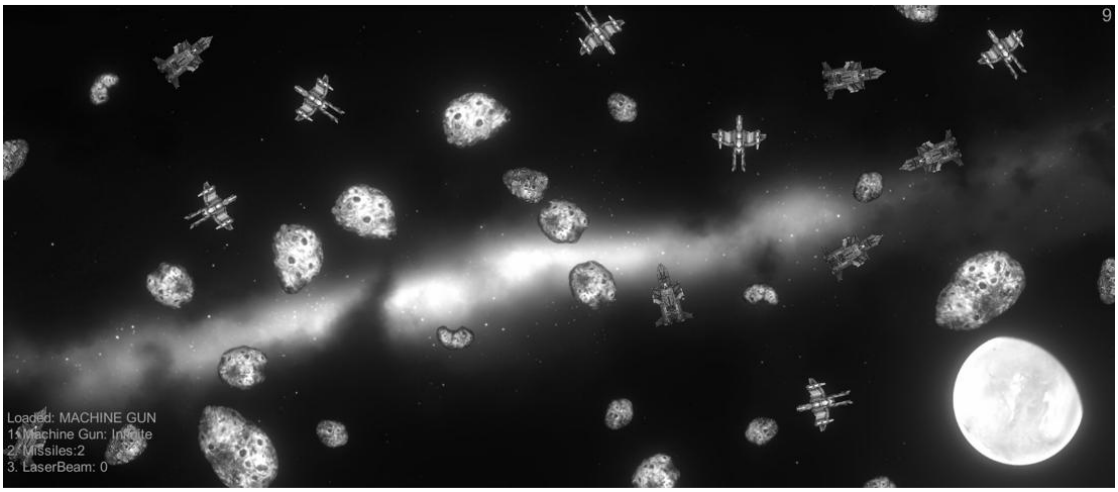

Abbildung 3.3: Zufällig generierte Map

vorhanden sein sollte, weshalb spät im Projektablauf mit der Implementierung eines KI-Gegners begonnen wurde.

Im Wesentlichen sollte der KI-Gegner in der Lage sein, während jeder Spielrunde ein gegnerisches Schiff zum Bekämpfen auszuwählen, sich in eine Schussposition zu bewegen, die ein freies Schussfeld ermöglicht und selbstständig zu entscheiden, wann gefeuert wird. Die größte Herausforderung bei der Umsetzung des KI-Gegners war das autonome Manövrieren des KI-Schiffes in eine freie Schussposition. Problem dabei war vor allem, dass die KI erkennen muss, ob sich zwischen dem eigenen Schiff und dem gegnerischen Schiff ein Asteroid befindet oder ein eigenes Schiff. Außerdem sollte die KI beim Manövrieren nicht an einem Asteroiden „hängen bleiben“.

Zunächst wurde erwogen, mit dem Pathfinding-System von Unity der KI einen Weg zu einem gegnerischen Schiff zu bahnen bzw. in eine freie Schussposition zu bringen. Allerdings bestand das Problem darin, dass das Pathfinding-System bei einer sich dynamisch veränderten Spielwelt an seine Grenzen stößt bzw. sehr aufwändig umzusetzen ist. So können sich Asteroiden in Belt-Wars bewegen, wenn auch nur leicht aufgrund ihrer hohen Masse, oder zerstört werden und somit verschwinden.

Lösungskonzept und Implementierung

Als Lösung wurde wieder ein pragmatischer Ansatz gewählt. Anstatt dass sich die KI an Asteroiden und eigenen Schiffen herum manövriert, bekommen KI-Schiffe die Fähigkeit Asteroiden beiseitezuschieben, indem kurzfristig die Masse des Rigidbody’s des Schiffs, das am Zug ist, erhöht wird. Mit der Erhöhung der Masse, die höher, als diejenige der Asteroiden ist, können diese leicht beiseitegeschoben werden. Ist die Runde wieder zu Ende, wird die Masse des Schiffs wieder auf ihren

ursprünglichen Wert zurückgesetzt.

Als Ziel wählt ein KI-Schiff immer das gegnerische Schiff, das sich zu ihm am nächsten befindet, außer in einem bestimmten Radius um das KI-Schiff befindet sich ein Schiff mit geringerer Gesundheit. Dann wird das Schiff mit geringerer Gesundheit als Ziel zum Angriff ausgewählt. Auf dem Weg zum Ziel ermittelt ein unsichtbarer Raycast an der Spitze der Schiffe, ob ein gegnerisches in das Schussfeld gelangt (siehe Abbildung 3.4). Ist dies der Fall, wählt das KI-Schiff seine stärkste noch vorhandene Waffe und feuert auf das gegnerische Schiff.

```
62      MoveTowards(enemyToAttack.position);
63
64      Debug.DrawRay(spawnPointProjectile.position, spawnPointProjectile.TransformDirection(Vector2.up) *10f, Color.red);
65      RaycastHit2D hit = Physics2D.Raycast(spawnPointProjectile.position, spawnPointProjectile.TransformDirection(Vector2.up), Mathf.Infinity);
66
67      if (hit)
68      {
69          if (hit.collider.name.StartsWith("Ship_Earth"))
70          {
71              shoot();
72              alreadyShot = true;
73              timer = 1f;
74          }
75      }
```

Abbildung 3.4: KI-Schiessen und Raycast

Ist der Angriff beendet, bewegt sich das KI-Schiff auf eine zufällig gewählte Position auf der Karte zurück. Für den Fall, dass sich allerdings ein von einem Support-Schiff hinterlassenes Item auf der Karte befindet, wird dieses eingesammelt (siehe Abbildung 3.5).

```
91      private void moveToFinalPosition()
92      {
93          if (!finalPositionDetermined)
94          {
95              //Chose Drop Item if exists or random Position
96              CameraMeasurements camera = new CameraMeasurements();
97              float x = Random.Range(camera.getHorizontalMin() + 0.5f, camera.getHorizontalMax() - 0.5f);
98              float y = Random.Range(camera.getVerticalMin() + 0.5f, camera.getVerticalMax() - 0.5f);
99              Transform dropItem = dropItemPosition();
100             finalPosition = dropItem ? dropItem.position : new Vector3(x, y, 0);
101             finalPositionDetermined = true;
102         }
103
104         MoveTowards(finalPosition);
105         RotateTowards(finalPosition);
106
107         if(transform.position == finalPosition)
108         {
109             finalPositionDetermined = false;
110             roundFinished = true;
111             timer = 3f;
112             alreadyShot = false;
113         }
114     }
```

Abbildung 3.5: Finale Position KI

Im Groben befolgt ein KI-Schiff in jeder Runde die folgenden Schritte:

1. Gegnerisches Schiff als Angriffsziel wird ausgewählt.
2. Es erfolgt eine Drehung in Richtung des gegnerischen Schiffes

3. Das KI-Schiff bewegt sich solange auf das gegnerische Schiff zu, bis der Raycast eine freie Schussbahn meldet.
4. Die stärkste vorhandene Waffe wird ausgewählt.
5. Das gegnerische Schiff wird einmal beschossen.
6. Falls ein Item auf der Karte vorhanden ist, bewegt sich das KI-Schiff an die entsprechende Position, um es einzusammeln. Ist kein Item vorhanden, bewegt sich das Schiff an eine zufällig ausgewählte Position.

3.3 Spielmenü

Als Gamer wissen wir, dass die Immersion ein unglaublich wichtiges Element ist. Wenn man in das Spiel eingetaucht ist, verliert man das Zeitgefühl und wird in das Spielgeschehen einbezogen. Ein wichtiger Faktor für das Eintauchen ist, wie einfach es für den Spieler ist, sich im Spiel zurecht zu finden. Das heißt, wie flüssig die User Experience (UX) und wie gut die Benutzeroberfläche (UI) gestaltet ist. Ein Spiel schadet sich selbst, wenn es zu wenig oder zu viele Informationen bereitstellt, zu viele Eingaben erfordert, den Spieler mit nicht hilfreichen Aufforderungen verwirrt oder es einem neuen Spieler schwer macht, zu interagieren. Schlechtes UI-Design kann das Spiel sogar komplett zerstören.

Problemanalyse

Um die User Experience möglichst intuitiv zu gestalten, wird ein konsistentes, einfach gehaltenes Spielmenü benötigt, welches den User so klar wie möglich durch die verschiedenen Funktionen des Spiels führt. Ein weiteres wichtiges Feature des Menüs stellt die Übernahme von User-Settings in andere Szenen und Spielinhalte dar. Kein Nutzer möchte die gleichen Einstellungen mehrmals tätigen müssen. Somit soll es beispielsweise möglich sein, Einstellungen von Musik und Sound im Main Menü vorzunehmen, um dann genau diese Einstellungen so im Pause Menüs des Spiels wiederzufinden (und umgekehrt).

Lösungskonzept und Implementierung

Mit Berücksichtigung der oben genannten Herausforderungen, wurde folgendes Lösungskonzept für das Spielmenü entworfen:

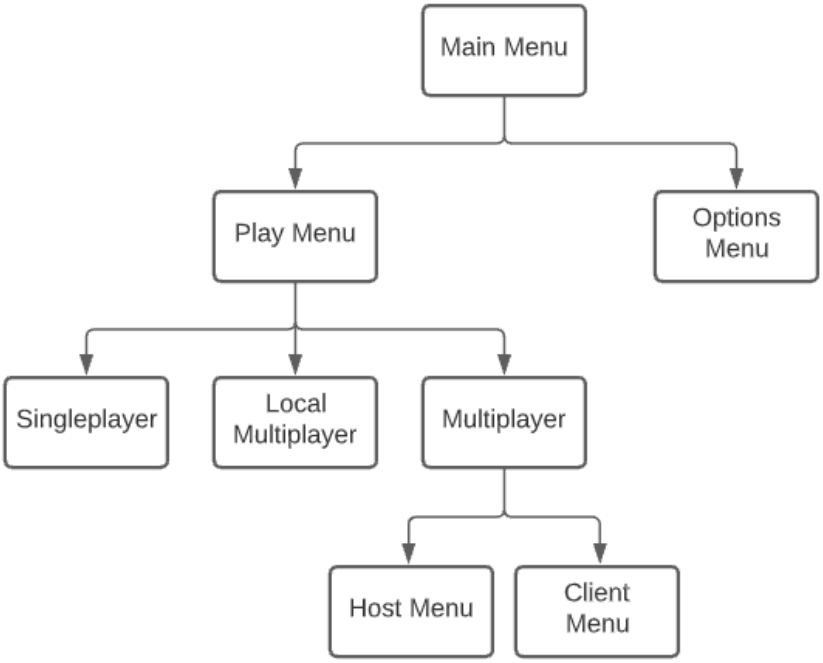

Abbildung 3.6: Main Menü Struktur

Zu Beginn befindet sich der User im Main Menü. Von dort aus hat er die Möglichkeit, Einstellungen an der Musiklautstärke oder der Soundeffektlautstärke vorzunehmen indem er sich ins Optionsmenü begibt. Möchte er jedoch direkt mit dem Spielen beginnen, kommt er vom Mainmenü durch einen einfachen Klick in das Spielmenü. Dort kann er sich zwischen drei verschiedenen Spielmodi entscheiden. Zur Auswahl stehen der Singleplayermodus gegen den Computer, der lokale Multiplayermodus und der Multiplayermodus über das Netzwerk. Wählt der User nun den Multiplayer, muss er sich nun entscheiden, ob er das Spiel hosten möchte, oder ob er die Seite des Clients übernimmt und der andere Spieler das Spiel hostet.

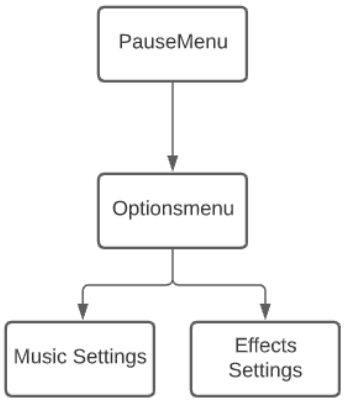

Abbildung 3.7: Pause Menü Struktur

Neben dem MainMenu war es wichtig, ein zweites Menü in das laufende Spiel zu integrieren. Hier hat der User die Möglichkeit, während des laufenden Spiels Änderungen an Musik- und Effekteinstellungen zu tätigen. Sollte dem User beispielsweise erst beim Spielen auffallen, dass Ihn die Musik stört, kommt er durch das Drücken einer Taste in das Pausemenü. Von dort aus kann er, neben dem Verändern der Einstellungen, auch das Spiel beenden.

Im nachfolgenden Bild wird anschaulich, wie das erreichen des Pausemenü umgesetzt wurde:

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.P))
    {
        if (GameIsPaused)
        {
            Resume();
        }
        else
        {
            Pause();
        }
    }
}
```

Abbildung 3.8: Pause Menü Code

Eine einfache Abfrage (pro Frame) genügt bereits um durch das Betätigen der Taste "Pïn das Menü zu gelangen und auf gleichem Wege wieder herauszufinden.

3.4 Sounds

Was wäre wohl Tetris ohne seine ikonische Melodie? Längst hat sich das Stück verselbstständigt und hat, ebenso wie das Spiel, Kultstatus erreicht. Videospielmusik gehört so selbstverständlich zum Spielerlebnis, dass wir sie oft gar nicht mehr bewusst wahrnehmen. Denn ist sie gut gemacht, fügt sie sich perfekt in das Spielgeschehen ein.

Mehr noch: Sie ist sogar in der Lage die Stimmung zu beeinflussen, die eine Spielszene erzeugen soll. Das gelingt, indem sie die anderen audiovisuell transportierten Informationen unterstreicht, oder sogar konterkariert.

Problemanalyse

Die nicht triviale Herausforderung dieses Features war nicht nur das Auswählen und Abspielen der richtigen Sounds und Musik zum passenden Zeitpunkt, sondern diese über das gesamte Spiel und die verschiedenen Szenen konsistent und generisch zu halten.

So empfanden wir es als sehr wichtig, eine Lösung zu entwerfen, die die Verwendung des Sounds auf gleiche Weise in verschiedenen Szenen zur Verfügung stellt. Ein weiterer wichtiger Punkt waren die vom User getätigten Einstellungen zu Musik und Soundeffekte in die verschiedenen Szenen zu übernehmen. Entscheidet sich ein User beispielsweise dafür, die Musik im Spielmenü auszustellen, wäre es ein sehr ernüchterndes Erlebnis, wenn er sie dann innerhalb der Spielszene ein zweites Mal ausstellen muss.

Lösungskonzept und Implementierung

Um die Verwendung der Sounds möglichst einfach zu gestalten, wurde ein Audio-Manager entworfen, welcher die Verwaltung und das Abspielen der einzelnen Sounds übernimmt. Somit können alle Sounds an einer Stelle gehalten werden und bei Benötigen eines bestimmten Soundeffekts einfach vom Audio-Manager aus abgespielt werden. Dieser Audio-Manager greift dabei auf globale Variablen zu, welche die Einstellungen des Users wiederspiegeln. Auf diese Variablen hat der AudioManager sowohl im Menü als auch während des Spielens Zugriff, um die vom Nutzer übernommenen Einstellungen in allen Szenen zur Verfügung zu stellen.

Zur Verdeutlichung folgt ein grobes Klassendiagramm, welches den Audio-Manager, die dazugehörigen Sounds und die Nutzung der globalen Variablen darstellen soll.

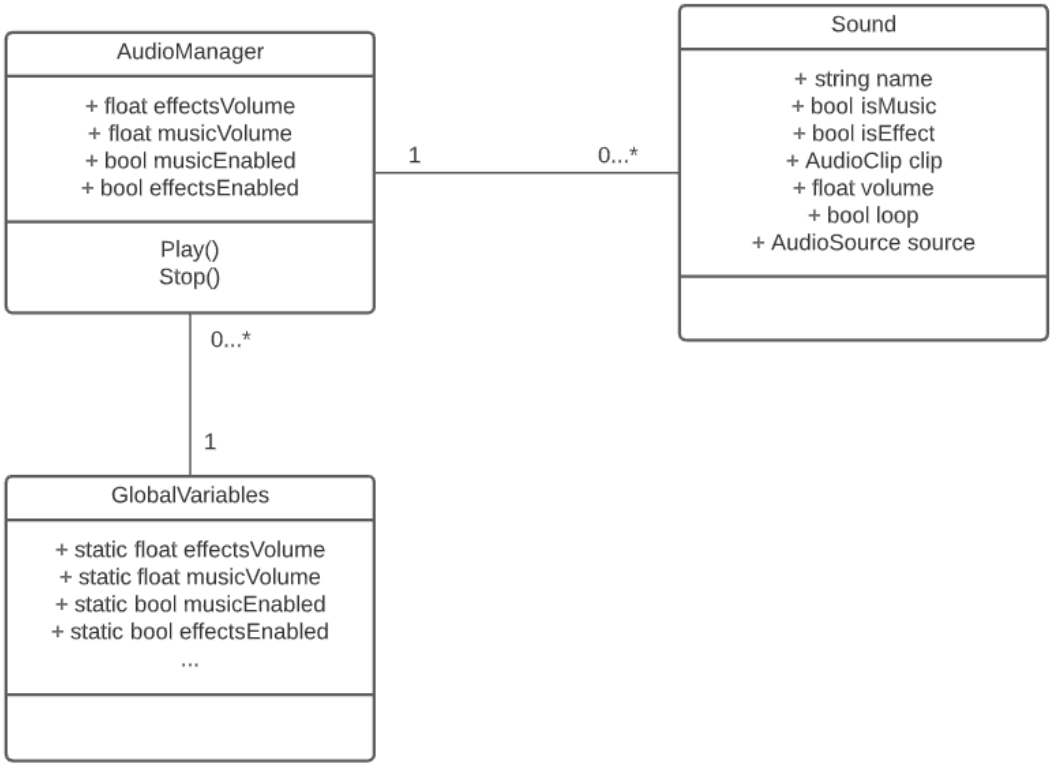

Abbildung 3.9: AudioManager, Sound und globale Variablen

Der Audio-Manager findet sich sowohl als GameObject in der Szene des Menüs, als auch in der Szene des Spiels (unter Verwendung eines Prefabs) wieder. Alle verwendeten Sounds befinden sich als Instanzen innerhalb des Audio-Managers, welcher deren Verwaltung tätigt. Nimmt ein User nun Einstellungen am Sound vor, werden die globalen Variablen automatisch aktualisiert. Wechselt er die Szene - beispielsweise vom Menü zum Spiel - ruft der Audio-Manager aktuellen der Spielszene die globalen Variablen auf und übernimmt dessen Einstellungen.

Soll nun ein Sound abgespielt werden, kann dies durch eine einzige Zeile Code getätigt werden, ohne dass man sich dabei um die einzelnen Lautstärken oder ähnliches kümmern muss. Um diesen Prozess zu verdeutlichen folgt ein Code Snippet aus dem Audio-Manager.

```
23 references
public void Play(string name)
{
    if (!GlobalVariables.local && (FindObjectOfType<NetworkAudioHandler>() != null)) {
        FindObjectOfType<NetworkAudioHandler>().handlePlayClientSound(name);
        return;
    }
    Sound s = Array.Find(sounds, sound => sound.name == name);
    if (s == null)
    {
        Debug.LogWarning("Sound: " + name + "not found!");
        return;
    }

    // set volume
    if (s.isMusic)
    {
        s.source.volume = s.volume * musicVolume;
        currentMusic = s;
    }
    else
    {
        s.source.volume = s.volume * effectsVolume;
    }

    if(!s.source.isPlaying)
    {
        // check whether it is allowed to play the music or the effect
        if (s.isMusic && musicEnabled)
        {
            s.source.Play();
        }
        else if (s.isEffect && effectsEnabled)
        {
                s.source.Play();
        }
    }
}
```

Abbildung 3.10: AudioManager Play()

Ein Sound wird durch die hier dargestellte "Play-Methodeäbgespielt. Es wird ausschließlich der für den Sound festgelegten Namen verwendet, um auf die entsprechende Sound Instanz zugreifen zu können. Zu beginn wird die Liste der Sounds nach dem gefragten Sound durchsucht. Wurde er gefunden, wird überprüft, ob es sich dabei um Musik oder einen Soundeffekt handelt. Da immer nur eine Musik gleichzeitig abgespielt wird, und der User während des Abspielvorgangs die Möglichkeit hat, dessen Lautstärke zu verändern, müssen wir sicher stellen, dass wir die aktuell abgespielte Musikinstanz speichern, um dessen Lautstärke während des

Abspielens verändern zu könnnen. Anschließend wird überprüft ob der gesuchte Sound bereits abgespielt wird. Ist dies nicht der Fall, wird der jeweilige Sound abgespielt, wenn es die vom Nutzer getätigten Einstellungen zulassen.

3.5 Netzwerk

Problemanalyse

Die größte Problematik in jedem Netzwerkspiel ist die Zustandssynchronisation. Spielobjekte sollten sich bei allen Mitspieler gleich und am besten auch gleichzeitig bewegen. Lebenspunkte und Munition sollten in beiden Spielinstanzen gleich sein, ebenso die Spielzeit. Welchem Spieler ist es gerade überhaupt erlaubt, eine Aktion auszuführern? Ein Event muss auch bei beiden Spielern gleichzeitig eintreten. Optionsmenü und die Lautstärke sind davon aber nicht betroffen, im Gegenteil. Diese sollten "nur"lokale Auswirkungen haben.
Im Laufe der Entwicklung jeder Spielkomponennte musste man sich darüber im klaren sein, ob diese für das Netzwerkspiel relevant war, oder nicht. Falls dies zutraf, galt es die Komponente zu verstehen und ins Netzwerk zu überführen.
Ein weiterer Punkt sind die Rollen der Spielteilnehmer aus Netzwerksicht und die generelle Verwaltung der Verbindung. Git es einen dedizierten Server auf den sich zwei Clients verbinden können, oder übernimmt ein Spieler die Rolle des Hosts und ist somit Server und Client zugleich? Gibt es eine Lobby?
Zuletzt stellt sich noch die Herausforderung, lokale Komponenten richtig zu verwenden. Die Netzwerkkomponente soll also die lokale um Funktionalitäten erweitern und nicht diese doppeln. Würde man einfach zwei unterschiedliche Komponenten verwenden wäre es um einiges aufwändiger kleine Änderungen durchzuführen, da man diese immer an zwei Stellen müsste.

Grob lassen sich die erwähnten Probleme in folgende Kategorien unterteilen:

- Erweiterung der lokalen Komponenten
- Zustandssynchronisation
- Aktionen
- Netzwerkmanager

Die API Mirror bietet umfangreife Funktionnen, um sich diesen Problemen zu stellen. Jedoch braucht es etwas Zeit und Übung, bis man diese versteht und auch richtig einsetzten kann.

Lösungskonzepte und Implementierung

Anhand der Kategorien der Probleme werde ich einige erarbeitete Löungen aufzählen.

Erweiterung der lokalen Komponente

Das Grundspiel ist für ein Gerät entwickelt worden. Die meisten Spielkomponenten müssen daher um Netzwerkfunktionen erweitert werden. Jedoch ist es unsinnig, bspw. zwei separate Klassen für die Erstellung des Spielsfelds zu entwickeln. Unser Lösungsansatzt hier war es, eine Netzwerkkomponente zu entwickeln, welche die benötigten Netzwerkfunktionen implementiert und die Funktionen der zugrundeliegenden Komponente verwendet. In der Single-Devices Komponennte muss an entsprechender Stelle überprüft werden, ob es sich um ein lokales oder Netzwerkspiel handelt, also ob diese selbst ihre Funktion ausführt, oder ob die Netzwerkkomponente entscheidet, wann welche Funktion aufgerufen wird.

```
void Start()
{
    GlobalVariables.local = false;
    shipNumber = GlobalVariables.numOfShips;
    asteroidDensity = GlobalVariables.asteroidDensity;

}

// Update is called once per frame
[Server]
Unity-Nachricht | 2 Verweise
override protected void Update()
{
    if(!start) { return; }
    base.Update();

}
```

Abbildung 3.11: Verwendung von lokaler Funktion im Netzwerk

Im Netzwerkspiel soll der Spielfluss nur vom Server gesteuert werden, daher führt auch nur dieser die Update Funktion aus. Außerdem soll diese erst zu einem späteren Zeitpunkt ausgeführt werden, die Variable start wird aus dem NetzwerkManager heraus gesetzt. Die Start Funktion musste auch angepasst werden, da diese in der lokalen Version bereits das Spiel erstellt, was im Netzwerk aber an anderer Stelle geschehen muss.

Zustandssynchronisation

Hier kann man wieder zwischen der Position und Spielzuständen(Leben, Munition, ...) unterscheiden, da Mirror unterschiedliche Komponenten bietet, diese umzusetzten. Zunächst braucht jedes Objekt im Netzwerkspiel eine Identität (Network Identity). Mittels dieser wird auch die Authorität festgelegt. Genauer gehe ich darauf in dem Lösungskonzept für Aktionen ein.

Um die Position im Netzwerkspiel für beide Parteien synchron zu halten, muss man lediglich dem Spielobjekt die Komponente Network Transform von Mirror hinzufügen. Wird ein solches Spielobjekt nun vom Server gespawned, so ist dessen Position für alle Teilnehmer die gleiche.
Zustände die für den Spielfluss relevant sind, bspw. Leben, Rundenzeit oder der Drop des Events werden mit Hilfe von SyncVars umgesetzt. Der Vorteil hier ist, dass diese Variablen sich automatisch über alle Instanzen aktualisiert.

```
// Update is called once per frame
[Server]
Unity-Nachricht | 0 Verweise
void Update()
{
    if(syncHealth != dest.health)
    {
        syncHealth = dest.health;
    }
}
```

Abbildung 3.12: NetworkkShoot Update

Zu beachten ist hier aber, die Werte der Variablen sinnvoll zu setzten, sprich in den meisten Fällen vom Server aus. Unter Umständen können auch unangenehmen Raceconditions autreten. Der neue Lebenspuktestand wird hier vom Server errechnet.

```
1-Verweis
private void setHP(float oldHP, float newHP)
{
    if(syncHealth <= 0)
    {
        dest.health = 0;
        dest.Explosion();
        if (isServer)
        {
            NetworkServer.Spawn(dest.debrisInstance);
        }

    }
    else
    {
        dest.health = newHP;
    }
}
```

Abbildung 3.13: Setzen der HP eines Schiffes für beides Spieler

Zudem bietet sie einen praktischen Hook, bei dem man auf eine Änderung sofort reagieren kann. Hier werden die verbleibenden Lebenspunkte eines Schiffs für beide Parteien auf den gleichen Wert gesetzt, nachdem der Server einen neuen Lebenstatnd festgelegt hat.

Aktionen

Aufgrund der rundenbasierten Natur unseres Spiels ist es wichtig manche Aktionen nur zu bestimmten Zeitpunkten zu erlauben. Ein Spieler darf sich nur dann bewegen, wenn er am Zug ist und es sich auch um sein Schiff handelt, also ob er die Authorität über das Schiff bestitzt. Schießen während seines Zuges, mit seinem Schiff und wenn er noch nicht geschossen hat. Das Optionsmenü sollte aber jederzeit zugänglich sein.
Um dies ins Netzwerk zu überführen ist es zunächst wichtig die lokale Komponente zu verstehen und festzulegen, wer denn letztendlich die Aktion ausführt. Es gibt hier unterschiedliche Ansätze. Zum einen kann man alles über den Server ausführen. Möchte der Client eine Aktion ausführen, so teilt er dies dem Server mit einer Commandfunktion mit. Der Server führt die Aktion aus und ggf. werden mittels eines ClientRPC calls auf weitere Funktionen nur auf den Clients ausgeführt. Abgesehen von der Bewegung, die der Client selbst durchführt. Zum anderen kann man auch den Client selbst Aktionen ausführen lassen und anschließend den Server darüber informieren. In unserem spiel werden viele Funktionen die den Spielfluss betreffen aber nur vom Server ausgeführt und der Client wird über Callbacks oder Hooks aktualisiert.

```
void Update()
{
    if(isServer)
        checkShoot();
    if (active && hasAuthority)
    {
        evalInput();
    }
}
```

Abbildung 3.14: NetworkShoot Update

Ein Beispiel ist die Klasse NetworkShoot. Hier wird zunächst vom Server geprüft, ob bereits ein Schuss gefeuert wurde. Anschließend, ob das Schiff am Zug ist, und ob es Schiff auch dem Spieler gehört. Danach wird der Input evaluiert. Die ob ein Schiff aktiv ist (Variable active) wird mit einer SyncVar bestimmt.

```
1-Verweis
public void evalInput()
{
    if (Input.GetKeyDown(KeyCode.Keypad1) || Input.GetKeyDown(KeyCode.Alpha1))
    {
        cmdSetWeapon(0);
    }
    if (Input.GetKeyDown(KeyCode.Keypad2) || Input.GetKeyDown(KeyCode.Alpha2))
    {
        cmdSetWeapon(1);
    }
    if (Input.GetKeyDown(KeyCode.Keypad3) || Input.GetKeyDown(KeyCode.Alpha3))
    {
        cmdSetWeapon(2);
    }
    if (Input.GetKeyDown(KeyCode.Space)) { cmdShoot(); }
}
```

Abbildung 3.15: Verarbeitung des Userinputs

Hier werden dann abhängig vom Input des Spielers entsprechende Commandfunktionen aufgerufen, welche der Server ausführt. Letztendlich teilt der Client dem Server nur mit was er machen möchte. Nur der Server setzt die Waffen entsprechend und schießt.

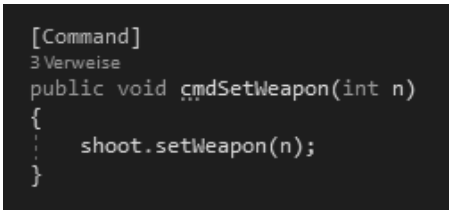

Abbildung 3.16: Command um Waffe zu setzten

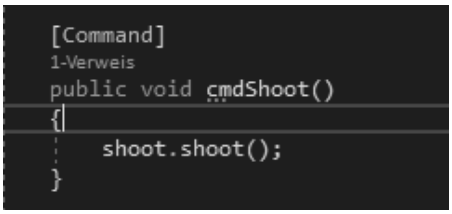

Abbildung 3.17: Command um die Waffe auszuwählen

In den Commands werden lediglich die Funktionen der Schusskomponente aus dem lokalen Spiel aufgerufen.

Natürlich ist jede Komponente etwas anders im Netzwerk umzusetzten. Generell kommt man entweder mit Commands, ClientRPCs und SyncVars an das Ziel.

Netzwerkmanager

Der Netzwerkmanager ist für die Rollenverteilung im Netzwerk, die Verwaltung der Verbindungen/Spieler und den Start des Spiels verantwortlich. Mirror bietet hierfür eine Basisklasse, welche man überschreiben und anpassen kann. Wird der Aufbau/Abbruch einer Verbindung festgestellt, so werden entsprechende Funktionen ausgeführt.

```
11 Verweise
public override void OnClientConnect(NetworkConnection conn)
{
    base.OnClientConnect(conn);
    OnClientConnected?.Invoke();
}
```

Abbildung 3.18: Überschreibung der Basisfunktion von Mirror

Die Funktion der Basisklasse wurde um ein Event erweitert welches mit die Aktivierung/Deaktivierung bestimmter Buttons im UI auslöst.

```
10 Verweise
public override void OnServerAddPlayer(NetworkConnection conn)
{
    Transform start = numPlayers == 0 ? startPos1.transform : startPos2.transform;
    GameObject player = Instantiate(playerPrefab, start.position, start.rotation);
    player.name = "Player" + count++;
    NetworkServer.AddPlayerForConnection(conn, player);
    foreach (var it in NetworkServer.connections)
    {
        if (it.Value != NetworkServer.localConnection)
        {
            cl = it.Value;
        }
    }

    if (numPlayers == 2)
    {
        GlobalVariables.local = false;
        OnGameStarted?.Invoke();
        menuObject.SetActive(false);
        NetworkSceneCreator nsc = new NetworkSceneCreator();
        nsc.createNetworkGameScene(cl);
        ngc.conn = conn;
        ngc.elements = nsc.game;
        ngc.start = true;
        FindObjectOfType<AudioManager>().Play("ingame_music");

    }
}
```

Abbildung 3.19: Spieler hinzugefügen

Spieler müssen auch gespawned werden, sonst können diese am Spielvergnügen nicht teilhaben. Es gibt unterschiedliche Spawnpunkte, was für das Spiel an sich irrelevant ist, da der Spieler nur die Schiffe steuert, aber es ist deutlich übersichtlicher.

Um die Authorität später richtig zu verwalten wird die des Clients in einer Variable gespeichert und später an den NetzwerkGameControlller weitergegeben. Sind zwei Spieler verbunden, so wird das Spiel gestartet. Das Menü wird hierfür deaktiviert, das Spielfeld erstellt, die Objekte gespawned und die Musik wird gestartet.

4 Evaluierung/Zusammenfassung/Ausblick

Ziel unseres Projekts war die Entwicklung eines rundenbasiertes Netzwerk-Multiplayer-Spiel. Bei dem Spiel mit einer 2D-Vogelperspektive, geht es um den Kampf zwischen zwei Raumschifffraktionen. Die Karte wird zufällig generiert. Die Raumschiffe können sich fortbewegen und können pro Runde einmal schießen. Ein zufällig auftauchendes Supportschiff soll zwischen den Runden nützliche Gegenstände fallen lassen, die von den Schiffen aufgelesen werden können.

Bei der Umsetzung haben wir unsere Vorstellungen größtenteils umgesetzt. Ein Single-Device-Multiplayer sowie ein Netzwerk-Multiplayer-Modus wurden erfolgreich implementiert. Zudem haben wir einen Singleplaye-Modus mit einer KI in unser Spiel aufgenommen.

Für die weitere Entwicklung des Spiels, wäre es möglich einen Storymodus zu erstellen. Des weiteren ist das Spiel durch zusätzliche Raumschiffe, Waffen und Gegenstände erweiterbar. Mit der Verbesserung der KI könnte man in Zukunft auch den Schwierigkeitsgrad im Einzelspielmodus anpassen.

In unserem Projekt gab es einige Herausforderungen. Da die einzelnen Komponenten isoliert entwickelt wurden, war die anschließende Integration oft sehr herausfordernd und aufwändiger als die ENtwicklung der Komponente selbst. Bei der Netzwerkkomponente bestand die Herausforderung darin, dass der Verantwortliche die gesamte Spiellogik kennen musste, wobei die Synchronisation der Gameobjects der schwierigste Punkt war.

Durch das Projekt konnten wir wieder Neues dazulernen und haben uns neue Kenntnisse in der Game Engine Unity angeeignet bzw. vertieft.

Literatur

[1] URL: https://www.dev-insider.de/was-ist-c-a-846162/.

[2] *General Overview*. URL: https://mirror-networking.com/docs/Articles/General/index.html.

[3] *Network Identity*. URL: https://mirror-networking.com/docs/Articles/Components/NetworkIdentity.html.

[4] *Network Transform*. URL: https://mirror-networking.com/docs/Articles/Components/NetworkTransform.html.

[5] *Remote Actions*. URL: https://mirror-networking.com/docs/Articles/Guides/Communications/RemoteActions.html?q=clientrpc.

[6] *SyncVars*. URL: https://mirror-networking.com/docs/Articles/Guides/Communications/RemoteActions.html?q=clientrpc.

[7] *Über Unity*. URL: https://de.wikipedia.org/wiki/Unity_(Spiel-Engine).