Projekt im Kurs “Entwicklung von Multimediasystemen”

Lukas Hodel Robert Kasseck Richard Remus

30. Januar 2015

Contents

1 Vorbereitung des Projektes

1.1 Idee

Um die Größe unserer Projektgruppe zu legitimieren, galt es ein Projekt mit hinreichender Komplexität zu finden. Bald hatten wir uns für ein selbst gestaltetes Spiel entschieden. Da wir uns bis zu diesem Zeitpunkt noch nicht mit Echtzeitspielen beschäftigt hatten, fanden wir diesen Bereich recht interessant. Auch wollten wir ein bisschen Physik in den Spielablauf einbringen. Wir einigten uns bald darauf, einen 2d-Weltraum-Shooter zu entwickeln. Darin sollte es darum gehen gegnerische in einem Spielfeld zu bekämpfen, in welchem Planeten und Sonnen die Schussbahn der Projektile beeinflussen können. Ziel des Spiels sollte es sein, als erster einen zuvor vereinbarte Zahl von Abschüssen zu erreichen. Zusätzlich sollte das Spiel über das Netzwerk spielbar sein und auch über optionale, vom Computer gesteuerte Gegner beinhalten. Wichtig war uns auch, die Logik und die GUI stark zu entkoppeln.

1.2 Anforderungen

1.2.1 Ziel

Implementierung eines Mehrspieler-Weltraum-Shooters (Gravitron) in C++.

1.2.2 Basic Features

- (3) Objekte verfügen über eine Gravitation relativ zu ihrer Masse.
- (5) Das Spiel soll über das Netzwerk im Multiplayer spielbar sein, wobei ein Spieler der Host ist.
- (4) Es soll eine KI mit 3 Schwierigkeitsstufen beinhalten.
- (2) Einstellungen werden lokal persistent gespeichert.
- (2) Jedem Spieler wird die Sicht relativ zu seinem Raumschiff gerendert.
- (3) Es soll zwischen den Sprachen Englisch und Deutsch gewechselt werden können.

1.2.3 Spielfeld

(5) Das Spielfeld ist eine Fläche. jeder Spieler steuert sein eigenes Schiff und bewegt sich auf der Fläche.

1.2.4 Spielmodus

(2) Es handelt sich um ein klassisches Deathmatch, d.h. jeder Spieler erzielt Punkte durch das Abschießen gegnerischer Schiffe. Wer auf diese Weise eine vorher definierte Punktzahl erreicht, gewinnt und das Spiel wird beendet.

1.2.5 Gravitationssystem

(5) Alle im Spielfeld befindlichen Objekte (Raumschiffe, Planeten, Schwarze Löcher...) besitzen eine Gravitation relativ zu ihrer Masse. Diese beeinflusst die Flugbahn von Geschossen und Schiffen.

1.2.6 Nicht-Spieler-Objekte

- (1) Planeten
- (1) Schrott
- (1) Asteroiden

1.2.7 Arten von Waffen

- (1) Laser
- (1) Rakete
- (3) Zielsuchende Rakete

1.2.8 Kollisionen

(2) Es wird für alle möglichen Kollisionen eine eine spezifische Behandlung definiert. Beispielsweise führt die Kollision mit einem Planeten zu einer stärkeren Beschädigung des Schiffes, als die Kollision mit einem Asteroiden.

1.2.9 PowerUps

(2) Das Abschießen eines gegnerischen Schiffes kann ein PowerUp erzeugen, welches von allen Spielern aufgesammelt werden kann.

1.2.10 Mögliche Arten von PowerUps:

(1) Waffenaufwertung

1.3 Plattformen

(1) Linux, OSX, Windows

1.4 Einarbeitung

Wir wollten unser Projekt so gut und professionell wie möglich umsetzen. Dementsprechend einigten wir uns darauf, nicht gleich los zu programmieren, sondern hinreichend viel Zeit zunächst in Recherche und anschließend in durchdachtes Design zu investieren.

Wir informierten uns über ähnliche Spiele, versuchten deren Designentscheidungen zu verstehen. Auch beschäftigten wir uns mit der allgemeinen Theorie der Spielentwicklung und stießen so auf das Buch *Game Programming Patterns* von Robert Nystrom. Dieses Buch sollte für uns bald eine wichtige Inspiration und Wissensquelle werden.

Für die Umsetzung physikalischer Gesetzmäßigkeiten bedienten wir uns dem Buch *The Nature of Code* von Daniel Shiffman, in welchem die Umsetzung von Naturgesetzen mithilfe von in **Processing** geschriebenen Codebeispielen zu erläutern.

Auch ging einige Zeit in das gedankliche durchdringen des Qt-Frameworks im Bereich GUI. Die Projektwebsite bot dabei immerhin in den meisten Fällen Hilfe und Beispiele. Ein gelegentliches Anecken an der Logik von Qt und auch entsetztes Unverständnis derer ließen sich jedoch manchmal nicht Vermeiden.

Darüber hinaus nutzten wir auch eine kurze Zeit um unserer Kenntnisse von C++ ein wenig aufzufrischen.

2 Qt/Qml

Als GUI-Sprache haben wir uns für QML und gegen die QWidgets entschieden, weil durch QML eine saubere Trennung von Code und Design möglich ist und weil es in Zukunft der Standard von QT sein wird.

2.1 Scene Management

Das erste Problem stellte sich in der Menüführung. Wie ist es möglich in QML zwischen verschiedenen Views zu schalten? Zuerst versuchten wir es indem wir die einzelnen Ebenen einfach mit dem Visibility-Parameter sichtbar und unsichtbar machten. Diese Lösung fühlte sich jedoch nicht sehr elegant an. Da in den meisten Fällen in denen QML verwendet wird, nicht das Standard-QML verwendet wird, sondern GUI-Libraries von Firmen wie Blackberry oder Nokia, gibt es nicht all zu viel Hilfe im Internet. Es gibt sozusagen keine *best practices*.

Wir haben am Ende eine Lösung gefunden, in der wir im **main.qml** ein **ScrollView**-Element einfügten, welches eine **ListView** beinhaltet. Dieser ListView wiederum geben wir als Model ein **VisualItemModel**. Ein VisualItemModel kann wiederum andere QML-Objekte beinhalten. Dabei nutzen wir nun in dem VisualItemModel ein **Loader**-Objekt, welches QML-Objekte nachladen kann. Wenn wir nun also in der Menüstruktur in eine andere Ebene gelangen, müssen wir nur noch dem **Loader** eine neue *source* (Pfad zu einer QML-Datei) geben. Mit dieser Struktur können wir nun ziemlich elegant geschachtelte Menüs erstellen.

```
VisualItemModel {
    id: theModel
    Column {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        Loader {
            id: modelLoader
            source: "menus/Main.qml"
        }
    }
}

ScrollView {
    z: 1
    id: scrollView
    anchors.fill: parent

    ListView {
        id: main_list
        model: theModel
        header: Item { height: 70 }
        footer: Item { height: 50 }
        anchors.fill: parent
    }
}
```

2.2 Property Binding

Ein weiteres Problem bestand darin QML-Controls an C++-Objekte zu *binden*. Dafür muss man zuerst ein C++-Objekt auf QML-Ebene verfügbar machen. Eine Möglichkeit ist es, beim Laden des QMLs, dem Objekt Referenzen zu C++-Objekten registrieren. Dies scheint nicht sehr elegant, ist jedoch ziemlich Effizient.

```
// Hier wird ein Settins-Objekt unter dem
// Namen "Settings" in QML verfügbar gemacht.
GravitronSettings settings;
engine.rootContext()->setContextProperty("Settings", &settings);
```

Sobald man dies gemacht hat, kann überall im QML auf das Settings-Objekt zugegriffen werden. So können nun auch QML-Controls gebunden werden:

```
// Hier wird der playerName von den registrierten Settings geholt.
// sobald der Benutzer den Focus vom Feld weg nimmt, wird
// ebenfalls über das Settings-Objekt der neue Spielername gesetzt.
TextField {
    id: txt_playerName
    height: Global.textFieldHeight
    width: Global.textFieldWidth
    placeholderText: qsTr("Name")
    text: Settings.playerName
    onEditingFinished: Settings.setPlayerName(txt_playerName.text)
}
```

Es muss dabei beachtet werden, dass keine Loops entstehen. Wenn wir z.B. die **setPlayerName()**-Methode nicht bei **onEditingFinished**, sondern bei **onTextChanged** aufrufen, würde ein solcher Loop entstehen.

3 Spiellogik

Ein sehr kritischer Teil des Projektes bestand in der inneren Spiellogik. Sie bestimmt die Regeln, den Ablauf und auch jeden möglichen Zustand des Spiels. Ihre Aufgabe ist es auch, den Zustand der einzelnen Akteure des Spiels konsistent zu halten. Um die Umsetzung der Logik unabhängig von Qt zu halten, haben wir uns entschieden die Spiellogik völlig von den anderen Bestandteilen des Spiels zu entkoppeln. Im folgenden werden wir auf die wichtigsten und interessantesten Bestandteile der Spiellogik eingehen.

3.1 GameActor

Die Klasse GameActor repräsentiert jedes denkbare Objekt, das Teil des Spielgeschehens ist. Dazu zählen die Raumschiffe der Spieler, Planeten, Sonnen, Weltraumschrott, Asteroiden aber auch jegliche Geschosse der Waffen über die die Schiffe verfügen.

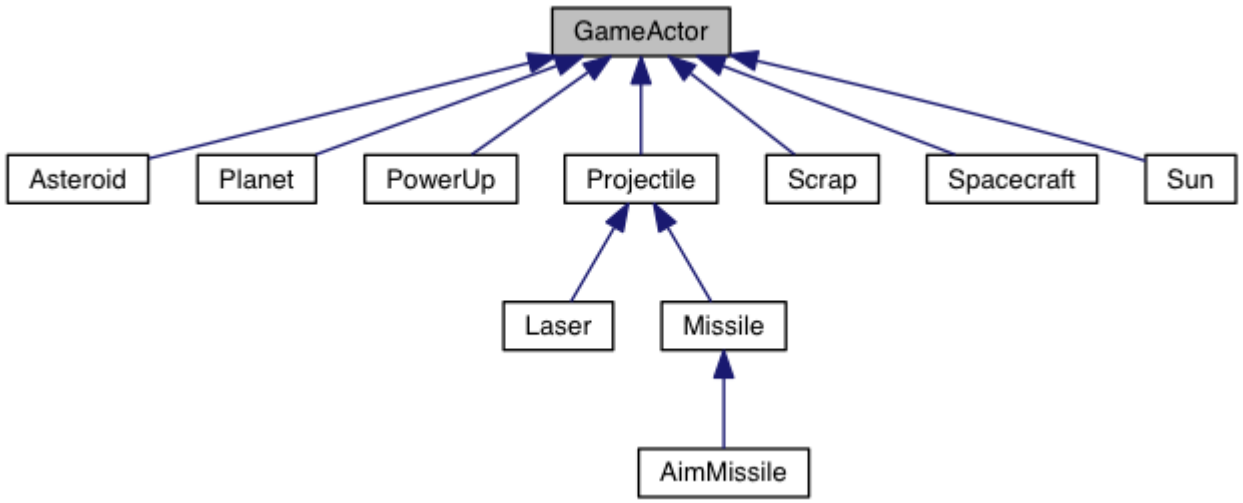

Figure 1: GameActor-Klassenhierarchie

Die wichtigsten Eigenschaften eines GameActors sind seine Position, Geschwindigkeit, Beschleunigung, Masse, Lebenspunkte, Gravitationskraft sowie die Reichweite, über welche seine Gravitation andere GameActors beeinflussen kann. Seine Positionierung und Bewegung werden mit elementarer Vektorrechnung realisiert. Wann immer eine Kraft auf einen GamActor wirken soll, wird die Methode **applyForce()** mit einem entsprechenden Kraftvektor verwendet. Die angewendete Kraft wirkt sich zunächst nur auf seinen Beschleunigungsvektor addiert, erst beim Aktualisieren des GameActors über **update()** wird der Beschleunigungsvektor mit dem Geschwindigkeitsvektor verrechnet und mit diesem dann schließlich die neue Position bestimmt. Falls GameActors miteinander Kollidieren wird dies mithilfe der **collisionDetection()** aus der **Physics**-Bibliothek ermittelt. Durch Implementation der virtuellen Methode **handleCollision()** kann festgelegt werden, wie der jeweilige GameActor mit einem Zusammenstoß umgehen soll. Sollte er schaden nehmen, kann dies mit **dealDamage()** realisiert werden, diese Methode bestimmt dabei auch, ob der zugewiesene Schaden die maximale Zahl von Lebenspunkten überschreitet und aktualisiert das Feld **killed** dementsprechend. **addHealth()** agiert in analog entgegengesetzter Weise zu **dealDamage()** wobei wir jedoch davon ausgehen, dass ein getöteter oder zerstörter GameActor nicht “wiederbelebt” werden sollte. Dies sollte, falls nötig über einen anderen Weg geregelt werden. Zusätzlich definieren wir mit einer Lebenspunktzahl von -1 einen GameActor, der nicht durch das Zuteilen von Schaden durch **dealDamage()** sterben kann. Falls ein GameActor durch seinen Tod einen Effekt auf das Spielgeschehen haben, zum Beispiel eine Explosion oder das setzen eines PowerUps, kann dies in der virtuellen Methode **handleKill()** definiert werden.

3.1.1 Spacecraft

Das Raumschiff ist ein spezieller GameActor, welcher von einem Spieler oder dem Computer gesteuert wird. Ihm haben wir spezielle Methoden zur Umsetzung der Inputs des Spielers beigefügt, die **shoot**- und **force**-Methoden. Darüber hinaus gaben wir dem Spacecraft einen Counter für die erfolgreichen Abschüsse anderer Schiffe durch den Spieler. Auch verfügt das Spacecraft über das Feld **weapon**, welches die gerade ausgerüstete Waffe beschreibt.

3.2 Projectile

Mit Projectile sollen alle möglichen Arten von Geschossen realisiert werden. Das könnten Laser und Raketen, aber auch Bomben, Minen oder andere Dinge sein, welche durch eine Waffe im Spielfeld platziert werden können. Projectile erbt von GameActor, bringt aber ein paar Erweiterungen. Zum einen kann es den Punktestand von befreundeten Raumschiffen beeinflussen, wobei "befreundet" bedeutet, dass dieses Projectile sie nicht in negativer weise beeinflussen kann. Zum anderen kann für jedes Projectile eine maximale Lebensdauer (time to live), gemessen in Update-Zyklen. Dadurch soll simuliert werden, dass die meisten Geschosse keine unendliche Reichweite haben, eine Rakete könnte zum Beispiel eine begrenzte Menge Treibstoff haben. Erreicht ein Projectile eine **timeToLive** von null, wird es als **killed** behandelt. In analoger Weise zu den Lebenspunkten des GameActors definieren wir, dass ein Projectile unbegrenzte Lebenszeit hat, wenn es eine **timeToLive** von -1 hat. Darüber hinaus gehorcht das Projectile allen durch GameActor definierten Regeln, sofern diese nicht überschrieben werden.

3.2.1 AimMissile

Die Implementierung der Zielsuchrakete war besonders interessant, da ein Algorithmus gefunden werden musste, der ein sinnvolles Ziel für die Rakete selektiert. Dabei muss geprüft werden, ob das Ziel ein feindlicher GameActor ist und auch ob es ein Spacecraft ist. Zeitweise erlaubten wir der Rakete auch GameActors zu wählen, die keine Raumschiffe sind, jedoch hat sich die Waffe dann sehr unpräzise angefühlt. Die Zielsuchrakete sollte einen höheren Wert als die anderen Waffen haben, und so entschieden wir uns, dass sie nur andere Spacecrafts angreifen darf. Zusätzlich haben wir auch eine Methode implementiert, welche ein zufälliges Ziel wählt. Dies könnte zum Beispiel eine Wirkung eines eingesammelten PowerUps sein, ein Störsender beispielsweise. Der Zielwahlalgorithmus der Zielsuchrakete wurde in teilen bei der Implementation der KI wiederverwendet.

4 Programmlogik

4.1 Game

Wie schon erwähnt, fanden wir im Rahmen der Recherche das Buch *Game Programming Patters*. Dies gab uns ein gutes Verständnis der Umsetzung und Aufbau eines Echtzeitspieles. In ihm stießen wir auch auf eine Vorlage unseres GameLoops.

4.2 GameLoop

Für die Umsetzung des GameLoops einigten wir uns auf die folgende Implementierung:

```
while (true)
{
  double start = getCurrentTime();
  processInput();
  update();
  render();
  sleep(start + MS_PER_FRAME - getCurrentTime());
}
```

Es wird sequenziell zuerst den Benutzerinput verarbeitet (**processInput()**), dabei handelt es sich um Inputs über das lokale Keyboard oder über auch über das Netzwerk. Anschließend wird der Spielstatus neu berechnet indem alle GameActors aktualisiert werden (**update()**). Wenn der neue Spielstand berechnet wurde, wird der View gerendert. Wenn dann noch Zeit übrig bleibt, wird diese wird der Prozess schlafen gelegt. So garantieren wir eine konstante Spielgeschwindigkeit, unabhängig von der Hardware des jeweiligen Systems.

4.2.1 Rendern in einem anderen Thread.

Der GameLoop ist ein eigener Thread, das tatsächliche rendern wird nicht im GameLoop bewerkstelligt. Dort werden lediglich die Daten vorbereitet, damit sie universell gerendert werden können. Dabei werden die vorbereiteten Views über QTs eigene Signale weitergereicht.

4.2.2 C++ Qml Mapping

Um den Spielstand universell rendern zu können und somit die Logik vom Design zu trennen, mussten wir ein Datenobjekt erstellen, welches die Eigenschaften eines Spielobjektes einfach darstellen kann. Dabei haben wir den GameActorView entwickelt. Der GameActorView ist streng genommen nur eine Liste mit *Key-Value-Pairs*. Er speichert die Eigenschaften als Keys und deren Values. Jeder GameActor, kann nun einen GameActorView von sich und seinem aktuellen Status erstellen. Dieser View kann dann serialisiert werden und an einer beliebigen anderen Stelle wieder gelesen werden. Ob dann aus dem View ein QML-Objekt generiert wird oder mit OpenGL ein Objekt gerendert wird ist egal.

In unserem Fall werden diese GameActorViews im **Game.cpp** von der Methode **render()** gelesen. Dann wird für jeden View ein QQuickItem generiert. Welche QML-Datei verwendet werden soll, wird ebenfalls im GameActorView definiert. So kann einfach für jeden GameActor eine eigen GameActor.qml erstellt werden.

Der Schwierige Punkt in diesem Szenario war, wie man aus C++ heraus QML-Objekte generieren kann:

```
// Zuerst muss der QML-Pfad vom GameActorView gelesen werden.
// Dieser definiert welche .qml-Datei für
```

```
// den View verwendet werden soll.
QString path = QString::fromStdString((*view)->getQmlPath());

// Dann wird eine neue Komponente davon erstellt
QQmlComponent component(engine, QUrl(path));
QQuickItem *childItem = qobject_cast<QQuickItem*>(component.create());

// Dieser muss dann ein Parent gegeben werden. In unserem Fall ist
// der qmlParent dem Game-Objekt bekannt.
childItem->setParent(qmlParent);
childItem->setParentItem(qmlParent);

...

// Dann können die einzelnen Eigenschaften dem QQuickItem
// übergeben werden.
for(pit = props.begin(); pit != props.end(); pit++) {
    childItem->setProperty(pit->first.c_str(), pit->second.c_str());
}
```

4.3 Multi-Key InputHandling

Beim *Inputhandling* handelt es sich um die Aufnahme und Verarbeitung von Keyboard-Inputs durch den Spieler. Das erste Problem stellte sich in diesem Bereich darin auch mehrere Keys parallel zu erkennen. Wenn man einfach auf den KeyDown Input-Event hört kommen die parallel gedrückte Tasten hintereinander. Und noch viel schlimmer ist, dass das Drücken der einen Taste, die anderen Tasten blockiert. Um dieses Problem zu lösen hören wir nicht nur auf den KeyDown-Event sondern auch auf den KeyUp-Event. Dazu nutzen wir einen Input-Liste (eigentlich ein Vektorobjekt), welcher bei KeyDown den Keycode speichert. Bei KeyUp vom gleichen Code wird dieser wieder von der Liste gelöscht. Um herauszufinden, welche Keys gerade gleichzeitig gedrückt werden, kann nun einfach über die input-Liste iteriert werden, egal ob eine Taste die anderen blockiert. Eine Taste ist gedrückt, bis ein KeyUp-Event der gleichen taste wieder kommt.

4.3.1 Mutex für Thread Safety

Da der GameLoop in einem eigenen Thread existiert, die Inputs aber vom Hauptthread kommen, verwenden wir beim schreiben und lesen der Input-Liste einen Mutex. So kann der GameLoop von der Liste lesen und der Hauptthread schreiben ohne dass es zu Kollisionen kommt.

4.4 Netzwerk

Um das Netzwerk zu ermöglichen haben wir eine **TcpServer**- und **TcpClient**-Klasse geschrieben. Diese werden beim Programmstart initialisiert. Es besitzen also alle Spieler einen Server und einen Client, egal ob sie der Server oder Client im Spiel sind.

Auch war es wichtig zu definieren, welche verschiedenen Pakete wir übermitteln wollen. Sozusagen ein eigenes Protokoll. Dabei sind wir auf folgende Typen gekommen.

Die **Views** sind die zu übertragenen GameActorViews, welche vom Client gerendert werden sollen. Pakete welche Views sind, starten immer mit einem *v* und enden mit einer *newline*.

Die **Inputs** sind die Keyboadinputs vom Netzwerkspieler welcher auf dem Server sein Raumschiff steuern möchte. Input-Pakete starten immer mit einem *i* und enden mit einer *newline*.

Die **Controls** sind Steuerbefehle vom Server zum Client. z.B. der Spielbeginn , das Aktualisieren des Lebensbalkens, die Position des Hintergrundbildes usw. Diese Pakete starten immer mit einem *c* und enden mit einer *newline*.

Grundsätzlich Enden alle Pakete mit einer *newline*. Dieser Entscheidung liegt die Tatsache zugrunde, dass wir das Problem hatten, dass Pakete *gesplitted* versendet wurden und so zum Absturz des Spiels führten. Mit dieser Lösung können wir auf der Ebene des TCP-Sockets überprüfen ob ein Paket vollständig angekommen ist. Wenn dies nicht der Fall ist (*newline* fehlt) buffern wir das Packet und fügen es mit dem nächsten Paket zusammen, bis es wieder vollständig ist.

4.4.1 Remote Rendering

Da wir bereits das Rendern komplett vom Gameloop getrennt hatten, konnten wir nun den Rendermechanismus relativ einfach netzwerkfähig machen. Um das zu realisieren, haben wir den TcpClient mit einer **remoteRender()**-Methode verbunden. Diese **remoteRender()**-Methode überprüft, ob das erhaltene Paket einen View beinhaltet, also mit *v* beginnt. Wenn dies der Fall ist, generiert die Methode daraus wieder eine liste von GameActorViews und ruft die gleiche **render()**-Methode auf wie der Server. Diese Methode rendert nun aus den GameActorViews wieder QML-Objekte und zeigt sie an.

4.4.2 Remote Inputs

Um die Inputs des Netzwerkspielers zu empfangen, wird auf der Seite des Netzwerkspielers ein *Inputhandler* erstellt wie auch für den lokalen Spieler. Dabei wird nun aber die Inputliste serialisiert über das Netzwerk versendet, sobald diese sich ändert. Auf der Serverseite ist nun ein **NetworkInputHandler** über *signals* und *slots* an den TCP-Socket der Servers gebunden. Dieser hört auf jedes erhaltene Paket und prüft ob es sich dabei um Inputs handelt. Wenn dies der Fall ist, wird das Paket deserialisiert und analog zum InputHandler in einem durch Mutex geschützten Inputvektor gespeichert. Dieser kann nun vom NetworkPlayer-Objekt bei jedem Durchlauf des Gameloops auf Inputs überprüft werden.

4.5 Player

Die Klasse Player ist die Vorlage für die Implementation der Repräsentanten für menschliche Spieler sowie die KI. Sie beinhaltet alle Information über die Spieler die nach außen weitergegeben werden müssen, wie z.B. der Punktestand des Spielers, die Lebenspunkte seines Raumschiffes oder auch die Zeit für die der Spieler aussetzen muss, nachdem sein Schiff zerstört wurde.

4.5.1 HumanPlayer

Der **HumanPlayer** besitzt einen Inputhandler mit dessen Hilfe er die aktuellen eingaben lesen kann und so das **Spacecraft** gesteuert wird.

4.5.2 HumanNetworkPlayer

Der **NumanNetworkPlayer** unterscheidet sich nur durch den Typen des InputHandlers vom HumanPlayer. Der NetworkPlayer überprüft in der **processInput()**-Methode den Inputvektor des **NetworkInputHandler**s. Diese connected er in seinem Konstruktor mit den Server-Events.

Ebenfalls wird auf das Paket “cname” connected. Zu Beginn des Spiels schickt der Netzwerkspieler seinen Namen übers Netzwerk. Der **HumanNetworkPlayer** hört darauf und setzt seinen Namen dementsprechend.

4.5.3 AIPlayer

Der AIPlayer oder auch die KI war entgegen der Erwartung in kurzer Zeit umsetzbar, da wir Teile der Logik der Zielsuchrakete wiederverwenden konnten. So sucht sich die KI recht zuverlässig ein Schiff oder ein PowerUp zum Ziel und verfolgt dieses. Jedoch schießt die KI, zum Zeitpunkt des Verfassens, in zufällige Richtungen. Die KI haben wir eher als ein *proof of concept* betrachtet und daher keine Zeit in ein sinnvolles Kampfverhalten oder Taktiken gesteckt. Im aktuellen Zustand schießt die zufällig, in gelegentlichen Testspielen haben haben wir die KI aber doch auch alles andere als wehrlos empfunden.

5 Tests

Für die Tests schrieben wir eine eigene Klasse. Diese führt die Tests mithilfe des QT-Testframeworks durch. Die beiden Klassen Physics und Vec3f bilden die Grundlage jeder Bewegung im Spiel. Dementsprechend war es von hoher Bedeutung, dass diese beiden Kassen auch immer wie erwartet arbeiten.

5.1 Physics

Für alle Test wurde zuerst einmal ein einheitlicher Ausgangszustand geschaffen in dem zwei GameActors auf eine feste Postion gesetzt wurden. Ausgehend von diesen Positionen und Gravitationseigenschaften wurde ein erwarteter Wert errechnet und dann in dem Test für alle Methoden der Physics-Klasse geprüft ob auch diese Werte ausgerechnet werden.

5.2 Vec3f

Analog zu den Tests der Physics-Klasse, wurde auch hier ein einheitlicher Ausgangsvektor geschaffen. Auch das weitere Vorgehen war wie bei der Phyics-Klasse.

6 Speichertests

Das Projet wurde in C++ geschrieben und so ist es wichtig Speicherzugriffsfehler und Memoryleaks zu finden und zu beheben. Zunächst probierten wir das Tool **valgrind** zu verwenden. Allerdings gab es einige Probleme, die die Analyse der Fehler unmöglich machten. Ein vermutliche Ursache ist die Speicheroptimierung von QT. Das Testen mit **sanitize** war deutlich erfolgreicher.

6.1 Sanitize

Der Speichertest mit **sanitize** ist mit ein paar einfachen Compiler- und Linker-Kommandos zu nutzen. Die entsprechenden Flags wurden über die QT-Projektdatei dem Compiler und Linker übergeben:

```
QMAKE_CXXFLAGS_DEBUG += -fsanitize=address -O1 -fno-omit-frame-pointer
QMAKE_LFLAGS_DEBUG += -fsanitize=address
```

Der Vorteil den wir bei Gegenüberstellung von **sanitize** und **valgrind** war, dass **sanitize** die Programmausführung bei einem Fehler abbricht. Dies hatte zur Folge, dass das Programm immer ohne Speicherfehler ist. Auch lief die Programmausführung mit **sanitize** wesentlich schneller als mit **valgind**.

7 Übersetzungen

Um das Programm in mehreren Sprachen zur Verfügung zu stellen, haben wir die QT-Translation-Tools benutzt. Der Text der in mehreren Sprachen angezeigt werden sollte, musste in die Funktion **qsTr()** gesetzt werden. In der Projektdatei mussten wir angeben, für welche Sprachen es eine Übersetzungen geben soll:

```
TRANSLATIONS = gravitron_de.ts \
               gravitron_en.ts
```

Der Name der Übersetzungsdateien kann beliebig sein, da eine Zuordnung manuell vorgenommen werden muss. Um nun die Übersetzungen zu erzeugen sind die drei folgenden Schritte notwendig: 1. **lupdate** in Gravitron.pro: Es werden alle Dateien nach der **qsTr()**-Funktion durchsucht und als *offene* Übersetzung registriert. 2. Die Übersetzungsdateien können nun mit QT Linguist geöffnet und übersetzt werden. 3. **lrelease**: Erzeugt die fertigen Übersetzungsdateien mit der Endung *.qm*.

Bei Programmausführung werden die Übersetzungen mit der Klasse **Locator** geladen.

8 Ergebnisse und Fazit

Nach den ersten paar Wochen der Entwicklung, nagte der Gedanke, dass wir uns vielleicht ein wenig zu viel vorgenommen hatten. Obwohl schon schon viel Code erzeugt worden war, blieben haptische Ergebnisse doch aus. Dazu gab es einige Probleme, welche sich hauptsächlich in den Eigenarten von QML begründeten und auch die persistente Speicherung der Settings war ein größeres Problem. Zusätzlich forderten die anderen Kurse des Semesters ihren Tribut.

Die Feiertage und das Jahresende brachten jedoch die Wendung. Nach und nach klickten die einzelnen Bausteine ineinander, aus vielen Teilen wurde ein Ganzes und plötzlich steuerten wir ein gelbes Quadrat in einem kleinen Fenster und konnten mit diesem andere blaue Quadrate verschießen. Das mag für einen äußeren Betrachter nicht nach viel klingen, doch für uns war klar: Die Engine stand. Unsere Architektur war sinnvoll und funktionierte.

Diese Erkenntnis brachte einen Prozess ins rollen, der sich zurückblickend, am besten als eine Flut von Eindrücken und Erfahrungen beschreiben lässt. Wir machten immer mehr Fortschritte und das Projekt näherte sich mit großen Schritten unseren Vorstellungen. Die erdachte Struktur erwies sich gleichermaßen als flexibel und robust, sodass, wann immer sich ein Problem ergab, dieses schnell gelöst werden konnte. Auch neue Features ließen sich leichter umsetzen als gedacht. Das Spiel wurde täglich deutlich verbessert und die regelmäßigen Commits im Git führten zu einem gesunden gegenseitigen anstacheln.

Es sei auch auf die verwendeten Tools verwiesen, gut, dass Git eine gute Idee ist, braucht hier nun nicht betont werden, jedoch fanden wir großen Nutzen in unseren anderen Tool. Zum einen Trello, mit dem wir ein sehr simples Kanban angelegt haben. Dies hat die Aufgabenverteilung erheblich vereinfacht. Außerdem konnte man gut ablesen, welches Teammitglied gerade welches Feature bearbeitet. Zum anderen lief die restliche (digitale) Kommunikation über Slack. Das aufgeräumte Design und vor allem die Einbindung des Gits und des Trellos über Plugins empfanden wir als angenehm und hilfreich.

Wenn wir nun das Ergebnis betrachten, sehen wir, dass wir fast alles geschafft haben, was wir uns zum Ziel setzten. Die KI hat zum Beispiel keine regulierbaren Schwierigkeitsgrade, dieses Feature blieb aus Zeitgründen aus. Auch haben wir nur ein einziges PowerUp implementiert, aber wir denken, dass seine Existenz ein Indikator für ein funktionierendes Konzept sind. Wenn wir unsere Architektur betrachten, sehen wir, dass sich neue PowerUps, Waffen und andere Dinge und Effekte wie z.B. ein schwarzes Loch oder ein Sonnensturm leicht implementieren lassen. Das Spiel lässt sich von diesem Punkt an gut erweitern und zu größerem ausbauen. Alles in allem sind wir sehr zufrieden.

Dieses Projekt war für uns gleichermaßen eine große Herausforderung und eine äußerst lehrreiche Erfahrung die wir nicht missen wollen.