Projekt im Kurs “Entwicklung von Multimediasystemen”

Lukas Hodel Robert Kasseck Richard Remus

29. 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

1.3 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. Bevor wir mit der Entwicklung beginnen konnten, war eine umfangreiche Recherche notwendig.

1.3.1 GameDesignPatters

2 Qt/Qml

Als GUI sprache haben wir uns für QML und gegen die QWidgets entschieden. Dies 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 mit der Menüführung. Wie ist es möglich in QML zwischen verschiedenen Views zu switchen? Zuerst versuchten wir es indem wir die einzelnen Ebenen mit einfach mit dem Visibility parameter sichtbar und unsichtbar machten. Diese Lösung fühlte sich jedoch nicht sehr elegant an. Da

viele die QML verwenden nicht das standard qml verwenden, sonder bereits 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, inder wir im main.qml eine **ScrollView** element haben welches eine **ListView** beinhaltet. Dieser List view wiederum geben wir als Model ein **VisualItemModel**. Ein VisualItemModel kann wiederum andere QML objekte beinhalten. Dabei nutzen wir nun in dem VisualItemModel eine **Loader** objekt, welches QML objekte nachladen kann. Wenn wir nun also in der Menustruktur 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 Menus 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 Eebene verfügbar machen. Eine Möglichkeit ist es indem man beim Laden des QMLs, dem Kontekt Referenzen

zu C++ objekten registriert. 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 überal im QML auf das Settings objekt zugegriffen werden. So können nun auch QML-Controlls gebunden werden:

```
// Hier wird der playerName von den registrierten Settings geholt.
// sobald der Benutzer den Focus vom Feld weg nimmt, wird
// ebenfalls über dasSettings 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 sonder bei onTextChanged aufrufen, hätten wir einen Loop.

3 Spiellogik

Ein sehr kritischer Teil des Projektes besteht 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.

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.

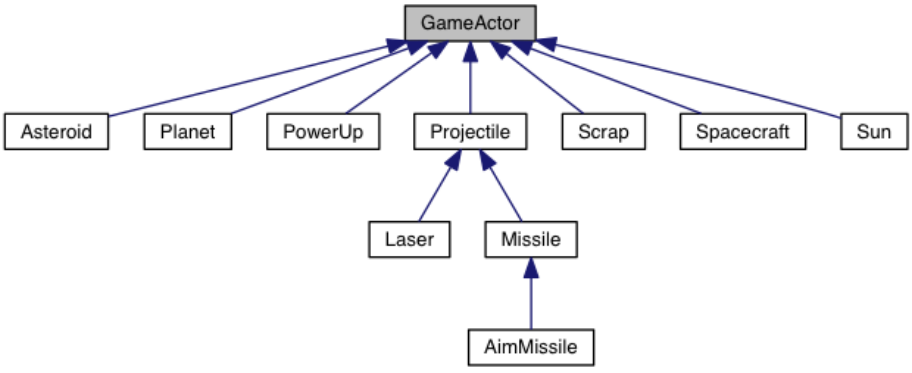

Figure 1: GameActor Klassenhierarchie

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.

3.2 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.

4 Programmlogik

4.1 Game

Am Anfang des Projektes war es zu erst mal wichtig herauszufinden wie ein Echtzeit spiel überhaupt aufgebaut ist. Um einwehnig das Gefühl dafür zu erhalten haben wir das Buch www.gameprogrammingpatterns.com "duchgelesen". Dabei sind wir auf den GameLoop gestossen. http://gameprogrammingpatterns.com/game-loop.html.

4.2 GameLoop

Dabei haben wir uns vorallem auf folgende implementierung geeinigt:

```
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 Keyborad input über das lokale keyboard oder übers netzwerk. Dann wird der spielstatus neu berechnet indem alle Aktoren sich aktuallisieren (update). Wenn der neue Spielstand berechnet wurde, wird dann alles gerendert. Wenn dann noch zeit überig blieb, wird diese geschlaffen. So garantieren wir eine konstante spielgeschwindigkeit, egal ob ein rechner schneller ist oder nicht.

4.2.1 Rendern in einem anderen thread.

Der GameLoop ist ein eigener Thread, das effektive rendern wird nicht im GameLoop gemacht. Im GameLoop werden nur die Daten vorbereitet, damit sie universell gerendert werden können. Dabei werden die vorbereiteten Views über QT 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 Key-Value liste. Er beinhaltet 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 Game::render gelesen. Dann werden für jeden View ein QQuickItem generiert. Welche QML Datei verwendet werden soll, wird ebenfalls im GameActorView definiert. So kann einfach pro GameActor eine 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 davo erstellte davo erstellt
QQmlComponent component(engine, QUrl(path));
QQuickItem *childItem = qobject_cast<QQuickItem*>(component.create());

// Dieser muss dann einen Parent gegeben werden. In unserem Fall ist
// der qmlParent dem Game.cpp 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 Updates

4.4 Multi-Key InputHandling

Beim input handling 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. Dabei Haben wir einen input vector, 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.

4.4.1 Mutex für thread savety

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.

4.5 Netzwerk

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

Dann war es wichtig zu definieren, was für verschiedene Pakete wir übermitteln wollen. So zusagen ein eigenes Protokoll. Dabei sind wir auch folgende Typen gekommen.

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

Die **Inputs** sind die Keyboad inputs vom Netzwerkspieler welcher auf dem Server sein Spacecraft steuern möchte. Input Pakete starten immer mit einem “i” und enden mit einer *newline*

Die **Controls** sind steuerbefehle vom Server zu client. z.B. dass das spiel begonnen hat, oder setzen des Lebensbalken, Hintergrund usw. Diese Pakete starten immer mit einem “c” und enden mit einem *newline*

Grundsätzlich Enden alle Pakete mit einer *newline*. Dies da wir das Problem hatten, dass Pakete gesplitted versendet wurden und zum Absturtz des Spiels führten. Nun können wir auf der Ebene des TCP Sockets überprüfen ob ein Packet 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.

4.5.1 Remote Rendering

Da wir bereits das Rendern komplett vom Gameloop getrennt haben, können wir nun den Render mechanismus relativ einfach netzwerkfähig machen. Um das zu machen, können wir den TcpClient mit einer remoteRender methode verbinden. Diese remoteRender Methode überprüft nun, ob das erhaltene Paket einen view ist. Startet mit "v". Wenn dies der fall ist, generiert die Methode daraus wieder eine liste von GameActorViews und ruft die gleiche render methode auf wie der Server. Die render Methode rendert nun aus den GameActorView wieder QML objetke und zeigt sie an.

4.6 Player

4.6.1 HumanPlayer

4.6.2 HumanNetworkPlayer

4.6.3 AIPlayer

4.6.4 AINetworkPlayer

5 Tests

5.1 Physics

5.2 Vec3f

6 Speichertests

6.1 Sanitize

7 Probleme beim Entwicklungsprozesses

7.1 Qt verstehen

7.2 Settings persistent speichern

8 Ergebnisse und Einschätzung

9 Fazit

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