

Adventure Game “Rocket”

**Ein Informatik-Projekt im Rahmen des Moduls BTI7301
an der Berner Fachhochschule**



Vorgelegt von:
Martin Käser
Fabian Schwab
Marcel Tschanz

Klasse I3q

Betreuung durch:
Jürgen Eckerle

Inhaltsverzeichnis

1	Startanalyse	1
1.1	Projektbeschreibung	1
1.2	Projektziele	1
1.3	Scoping	1
1.4	Anforderungen	1
2	Requirement Engineering	1
2.1	Use Case Diagramm	1
2.2	Use Cases	1
3	Design Phase	2
3.1	Finale Spielstory	2
4	Technologien / Pattern	4
4.1	Hierarchical State Machine	4
4.2	Speichern und Laden in Unity	4
4.2.1	Alternative: UnitySerializer	4
4.2.2	Speicher- und Ladevorgänge in Rocket	5
5	Realisierung	6
5.1	Verwendete Tools	6
5.2	Arbeitsweise	6
5.2.1	GitHub als Repository	6
5.2.2	Austausch von Komponenten: Das Mergen	6
5.3	Übersicht über die Komponenten des Levels	9
5.3.1	Levelplan	9
5.4	Modellierung der Räume und deren Verhalten	10
5.4.1	Situation beim Spielstart	10
5.4.2	Status der Räume	10
5.4.3	Beleuchtung	11
5.5	GameController	12
5.6	Spielfigur	12
5.6.1	Tastatur-Steuerung	12
5.6.2	Maus-Navigation	13
5.6.3	Inventar & Interaktion mit Objekten	13
5.7	Computer & Computercard	13
5.8	Lift	14
5.9	Gegnerischer Akteur	15
5.9.1	Zustände	15
5.9.2	Zustände in Unity mit HFSM	16
5.9.3	TopLevelState Level 00	17
5.9.4	AkkuState	18
5.9.5	PluggedInState Level 01	19

5.10	Robotereigenschaften und Verhalten.....	20
5.10.1	Der Roboter umgesetzt in Unity	20
5.10.2	Standard Komponente vom Roboter	21
5.10.3	Robot (Script).....	22
5.10.4	NavMesh and NavMesh Agent and.....	22
5.10.5	Script: EnemyAI	22
5.10.6	PathUtils	23
5.10.7	Patrouille.....	23
5.10.8	Sichtradius	24
5.10.9	Batterie	24
5.11	Inventar und Items	24
5.11.1	Generische Itemklasse.....	24
5.11.2	Info-Boxen	24
5.11.3	Auswählen des Items	24
5.12	Darstellung von Text-Meldungen	25
5.12.1	ProximityMessage.....	25
5.12.2	InteractionMessage.....	25
5.13	MessageDispatcher & Telegramm.....	26
5.13.1	Spielerinteraktion mit Game-Objekten.....	26
5.14	EntityManager	26
5.14.1	Einsatz bei Telegramme	26
6	Spielmenü.....	27
6.1	Unity GUI-Elemente.....	27
6.2	Menu mit Spielobjekten und Kamera	28
7	Setup des Projekts	29
7.1	Build & Run	30
8	Erfahrungen.....	31
8.1	Unity	31
8.1.1	Genutzte Features & Funktionen in Unity.....	31
8.1.2	Erwähnenswerte Einstellungen.....	35
8.1.3	Positive Erfahrungen	36
8.1.4	Negative Erfahrungen.....	36
8.1.5	Lessons learnt	36
8.2	Xamarin MonoDevelop-Unity	38
8.2.1	Positive Aspekte	38
8.2.2	Negative Aspekte.....	38
8.3	Fazit zum Projektabschluss	39
8.4	Offenes und Ausblick.....	39
9	Glossar.....	40
10	Referenzen.....	41

1 Startanalyse

Die Ergebnisse des Requirement Engineerings im Rahmen des Moduls BTI7082 so wie auch die Unterlagen zu den Artefakten der Fortschritte in der Designphase sind in separaten Dokumenten angelegt. Für die gedruckte Version, werden die blau unterstrichenen Hyperlinks durch die Angabe des Speicherortes auf dem GitHub Repository erweitert.

Anforderungsdokument:

github.com/Makae/rocket/blob/master/Dokumente/Dokumentation/Anforderungsdokument/Anforderungsdokument_V1_5_3.pdf

Use Cases & Diagramm:

github.com/Makae/rocket/tree/master/Dokumente/Use%20Cases

1.1 Projektbeschreibung

[Anforderungsdokument](#)

1.2 Projektziele

[Anforderungsdokument](#)

1.3 Scoping

[Anforderungsdokument](#)

1.4 Anforderungen

[Anforderungsdokument](#)

2 Requirement Engineering

2.1 Use Case Diagramm

[Diagramm](#)

2.2 Use Cases

[Use Cases](#)

3 Design Phase

Der grösste Teil der Artefakte aus der Designphase ist auf Papier vorhanden. Die wichtigsten Gedanken von uns, wie auch ein Use-Case Diagramm sind im Dokument „[Design Artefacts](#)“ abgefasst worden.

GitHub Repository Link:

github.com/Makae/rocket/blob/master/Dokumente/Dokumentation/Design/Design_Artefacts.docx

3.1 Finale Spielstory

Ältere Ideen von Spielstories sind auf dem Repository github.com/Makae/rocket und im Design Dokument zu finden.

Spielstory

Nach einem kurzen Briefing durch ein Informationsfenster, welches über den groben Spielplan und die Regeln informiert, beginnt das Spiel mit dem Eintreten der Figur in das Level. Der Spieler steuert aus der First-Person-Perspektive eine menschenähnliche Figur durch ein abgeschlossenes Level, eines Bürokomplexes. Die Umgebung des Basislevels wird auf eine Ebene beschränkt bleiben. Um den ersten Raum zu verlassen, muss ein passender Gegenstand gefunden werden, um die Türe für den nächsten Raum öffnen zu können. Der Spieler stellt fest, dass im nächsten Raum kein Licht brennt. Nur die Notbeleuchtung ist aktiv. In der Nähe befindet sich ein schwach beleuchteter Sicherungskasten. Bei Interaktion mit dem Sicherungskasten wird der Spieler darauf hingewiesen, dass etwas kaputt ist. Bei einer erneuten Interaktion findet der Spieler heraus, dass die Sicherung kaputt ist. Die Ersatzsicherung dazu befindet sich im Startraum. Sobald der Spieler die Ersatzsicherung aufgenommen hat, ist diese im Inventar sichtbar. Um den Strom einzuschalten, muss der Spieler die Ersatzsicherung bei sich tragen und erneut mit dem Sicherungskasten interagieren. Der Stromfluss ist nun prinzipiell wieder wiederhergestellt und das Licht im Level kann im Kontrollraum eingeschaltet werden.

Zudem befindet sich im Kontrollraum ein Button, um den über ein Fenster einzusehenden, anliegenden Raum zu öffnen. Dies gibt den Zugang zu einem wichtigen Gegenstand frei, aktiviert aber auch einen weiteren Gegenspieler. Der Roboter bemerkt den Spieler direkt nach seiner Aktivierung noch nicht, aber verlässt den Raum und gibt dabei ein Tonsignal ab.

In einem weiteren Raum findet der Spieler eine Leiche vor, die einen Batch bei sich trägt. Mit dem Batch ist ein Computer bedienbar, der den Lift aktivieren, den Schatzraum öffnen und den Code für die Bombe sichtbar machen kann. Weiter befindet sich versteckt im Level eine Bombe. Um den Sprengsatz zu benutzen ist eine Kombination zur Aktivierung nötig (-> siehe die Beschreibung des Computers).

Ein Lift in der Nähe des Kontrollraumes ist durch ein zerstörbares Hindernis blockiert. Beim Untersuchen des Hindernisses, bemerkt der Spieler, dass das Objekt durch eine Explosion zerstörbar sein könnte. Sobald das Hindernis weg ist, stellt der Spieler fest, dass er für den Lift einen Schlüssel benötigt, um von dieser Ebene wegzukommen.

Der Schlüssel befindet sich in einem Raum, welcher noch nicht zwingend vom Spieler besucht wurde. Hat der Spieler den Schlüssel aufgenommen, das Tor zum Lift geöffnet und die Gegenstände vor dem Lift weggesprengt, kann er die Aufzugstüre öffnen und durch das Aktivieren des Lifts das Spiel gewinnen.

4 Technologien / Pattern

4.1 Hierarchical State Machine

Das Verhalten der Roboter in Rocket wird durch endliche, hierarchische Zustandsautomaten kontrolliert. Im Englischen werden sie „Hierarchical Finite State Machines“, kurz (HFMS) genannt und sind ein in der Spieleentwicklung typisches Konstrukt, um für einen Agenten eine endliche Anzahl Zustände modellieren zu können. Dabei können nicht nur Personen - oder wie bei uns der Roboter – mit dem Pattern der hierarchischen Zustandsautomaten modelliert werden, sondern das ganze Spiel als Solches könnte ein hierarchischer Zustandsautomat sein. Gegenüber einfachen Zustandsautomaten haben sie den Vorteil, dass man sich einiges an Code sparen kann, wenn bestimmte Funktionen auch über mehrere Zustände hinweg kontinuierlich ausgeführt werden müssen,

Nach dem Studium des Kapitel 5.39 Hierarchical State Machines (Millington, 2006) entwarfen wir erste Skizzen für den Einsatz von Automaten im Spiel Rocket.

Evaluiert haben wir den Einsatz der Automaten für die Räume und für den Spieler selber. Wir sind aber zum Schluss gekommen, dass sich der Einsatz einer State Machine auf Grund der geringeren Komplexität von Spielfigur und Umgebung für diese Komponenten nicht empfiehlt. Deshalb basiert in der vorliegenden Alphaversion alleine die Intelligenz der Roboterwesen auf einem hierarchisch aufgebauten Zustandsautomaten.

Die Eigenintelligenz der Räume, der Spielfigur und der Gegenstände (Computer, Lift etc.) sind im Spiel durch Decision Trees implementiert worden.

4.2 Speichern und Laden in Unity

Das Speichern von Spielständen wird im Allgemeinen über den Ansatz der (De)-Serialisierung realisiert. Sich verändernde GameObjekte werden also exportiert und beim Laden wieder eingelesen.

Bei der Serialisierung nach XML, JSON oder als Binary stehen einem aber die Unity-Komponenten im Weg, welche von MonoBehaviour erben -> GameObjects. Diese Unity-Klassen implementieren die notwendigen Interfaces nicht. Typische Fehlermeldungen, bei Exporten in XML und der Binary Serialisierung:

XML: *InvalidOperationException: To be XML serializable, types which inherit from IEnumerable must have an implementation of Add(System.Object) at all levels of their inheritance hierarchy. UnityEngine.Transform does not implement Add(System.Object)*

C# Serialiazation (binary):

SerializationException: Type UnityEngine.GameObject is not marked as Serializable

4.2.1 Alternative: UnitySerializer

Quelle: <http://whydoidoit.com/unityserializer/>

Seit Mitte 2013 wird das Gratis-Tool UnitySerializer leider nicht mehr weiterentwickelt und weist Kompatibilitätsprobleme mit Unity 4.5 auf. Der UnitySerializer basiert auf einem von Mike Talbot geschriebenen Silverlight-Serializer, mit dem er Unity-Komponenten serialisieren kann. Die Daten werden in PlayerPrefs als String Key in die Registry geschrieben: (HKEY_CURRENT_USER\Software\DefaultCompany\SaveLoad)

Um den UnitySerializer verwenden zu können, müssen alle zu serialisierenden Gameobjekte das im Downloadpaket enthaltene Script *StoreInformation.cs* angehängt haben. Das Gameobjekt erhält so zum einen eine Unique ID und zum anderen werden damit die notwendigen Interfaces implementiert. Auch alle Kinder eines Objektes müssen dieses Script angehängt bekommen.

4.2.2 Speicher- und Ladevorgänge in Rocket

Die Alpha-Version des Spiels Rocket bietet keine Speicher-und Ladefunktionen. Sobald aber ein zweites Level hinzugefügt wird, drängt sich die Möglichkeit zur Speicherung auf.

Erste Tests haben aber unbefriedigende Resultate beim Wiederherstellen von Objekten zur Laufzeit geliefert. Besonders die Re-Positionierung der Komponenten im Level scheiterte regelmässig. Ein Code-Refactoring des in 4.2.1 erwähnten UnitySerializer müsste wahrscheinlich durchgeführt werden. Andernfalls könnten auftauchende Probleme manuell behoben werden, in dem man eigene Scripts für das Abspeichern aktueller Attributwerte und für das anschliessende Neuinitialisieren der Objekte und Setzen der gespeicherten Werte erstellt.

5 Realisierung

In diesem Kapitel werden die Gameobjekte mit ihren Komponenten beschrieben, welche unser Spiel hauptsächlich ausmachen. Noch offene Punkte, das heisst entweder nicht realisierte Funktionen oder bekannte Bugs wurden ganz am Ende dieses Dokuments in Punkt 9.4 unter „Offenes und Ausblick“ festgehalten.

5.1 Verwendete Tools

Name	Funktion	Version
Unity 4	Entwicklungsstudio	4.5.4f1
MonoDevelop	Editor	4.0.1
GitHub	Code Repository	Windows Clients: 6.5 Mac OS: Terminal only
MS Visio	Charts & Diagramme	2010
MS Word	Dokumentation	2010

5.2 Arbeitsweise

5.2.1 GitHub als Repository

Als Code- und allgemein als Datei-Repository verwendeten wir den Hosting-Dienst GitHub (github.com). Das dort angelegte Projekt ist für Drittpersonen lesbar, jedoch besitzen nur Mitglieder des Projekts Schreibrechte.

Um mit dem Repository arbeiten zu können verwendeten wir den Windows Client resp. das Terminal unter MacOS. Mit dem Windows Client kam es regelmässig zu nicht lösbaren Konflikten wenn es sich nicht um Binary-Dateien handelte.

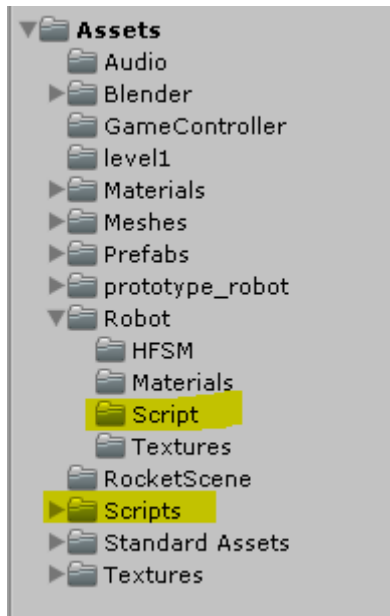
Für ein nächstes Projekt würden wir uns nach einem anderen Versionsverwaltungssystem umsehen, welches auf die Spieleentwicklung zugeschnitten ist. Unser Hauptproblem beim Arbeiten mit dem GitHub Client lag darin, dass jedes geänderte File komplett synchronisiert resp. hochgeladen wurde. Wenn dies nicht blockweise für eine Datei geschieht, ist es kaum realistisch mit langsameren Internetanschlüssen ein Projekt von einer Online-Ablage gemeinsam zu nutzen.

5.2.2 Austausch von Komponenten: Das Mergen

Anfangs Dezember begannen wir damit, die einzeln entwickelten Komponenten zusammen zu führen um ein erstes spielbares Level zu gestalten. In dieser Vorabversion sollten alle bis zu diesem Zeitpunkt erarbeiteten Einzelkomponenten in einem Projekt vereint werden. Dieses Zwischenprodukt wurde an alle Gruppenmitglieder verteilt und bildete die Basis für den weiteren Verlauf unserer Arbeit. Änderungen wurden von da an die ganze Gruppe kommuniziert und die manipulierte Komponente wurde als Prefab an die restlichen Teammitglieder verteilt. Als Beispiel ist dieses Vorgehen im nächsten Abschnitte am First-Person-Controller beschrieben.

FirstPersonController verändert (exakt: dessen Child die „Main Camera“)

1. Erstellen der benötigten Scripts oder Materialien in den dafür vorgesehenen Ordnern.



Anmerkung: Eine einheitliche Projektstruktur (Ordner-Hierarchie) wurde von uns leider zu spät definiert, so dass nicht alle Scripts in dem dafür vorgesehenen Scripts-Ordner liegen.

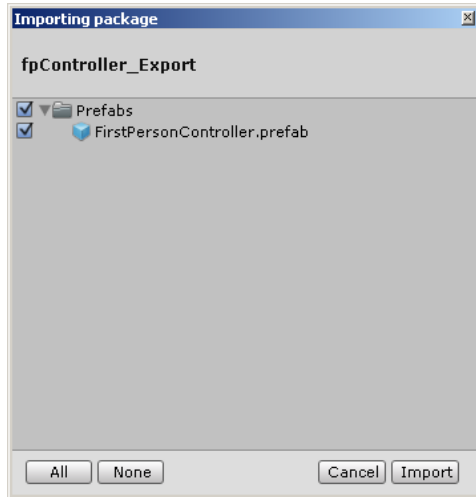
Eine Reorganisation der Assetstruktur hatte aber speziell bei Fabian häufig negative Auswirkungen auf den Editor MonoDevelop, da dieser scheinbar seine Referenzen zu den Scripts nicht anzupassen vermochte. Deshalb wurde die Hierarchie nicht mehr neu geordnet.

2. Änderungen am Objekt in der Hierarchie vornehmen
3. Auf Parent des veränderten Objekts (wenn nicht selbst Parent) klicken und im Inspector „apply“ wählen.
4. In der Project-View Rechtsklick auf das Prefab und „Export Package“ wählen
5. Wenn möglich alles abwählen, was sicherlich in keinem Zusammenhang mit den Änderungen steht. Auf jeden Fall aber das Prefab selbst und unbedingt benötigte Komponenten markieren.
6. Änderungen kommunizieren, dass von Prefab X eine neue Version bereit steht. Vorzugsweise diese Prefab-Datei mit dem Erstellungsdatum und einer Versionsnummer versehen auf das Repository legen.

Import auf den Clients der anderen Teammitglieder

Nachdem die übrigen Teammitglieder von der Person die die Komponente manipuliert hat informiert wurde, dass Änderungen verfügbar sind, laden sie sich das Paket mit den neusten Daten vom Repository herunter. Alle Exporte tragen die Dateiendung *.unitypackage. Um den Import zu starten, öffnet man die Unity Applikation und folgt den Schritten 1-3 in der folgenden Anleitung:

1. Rechtsklick auf Prefabs Folder und „Import Package“ wählen
2. Package auswählen und Frage nach Import bestätigen:

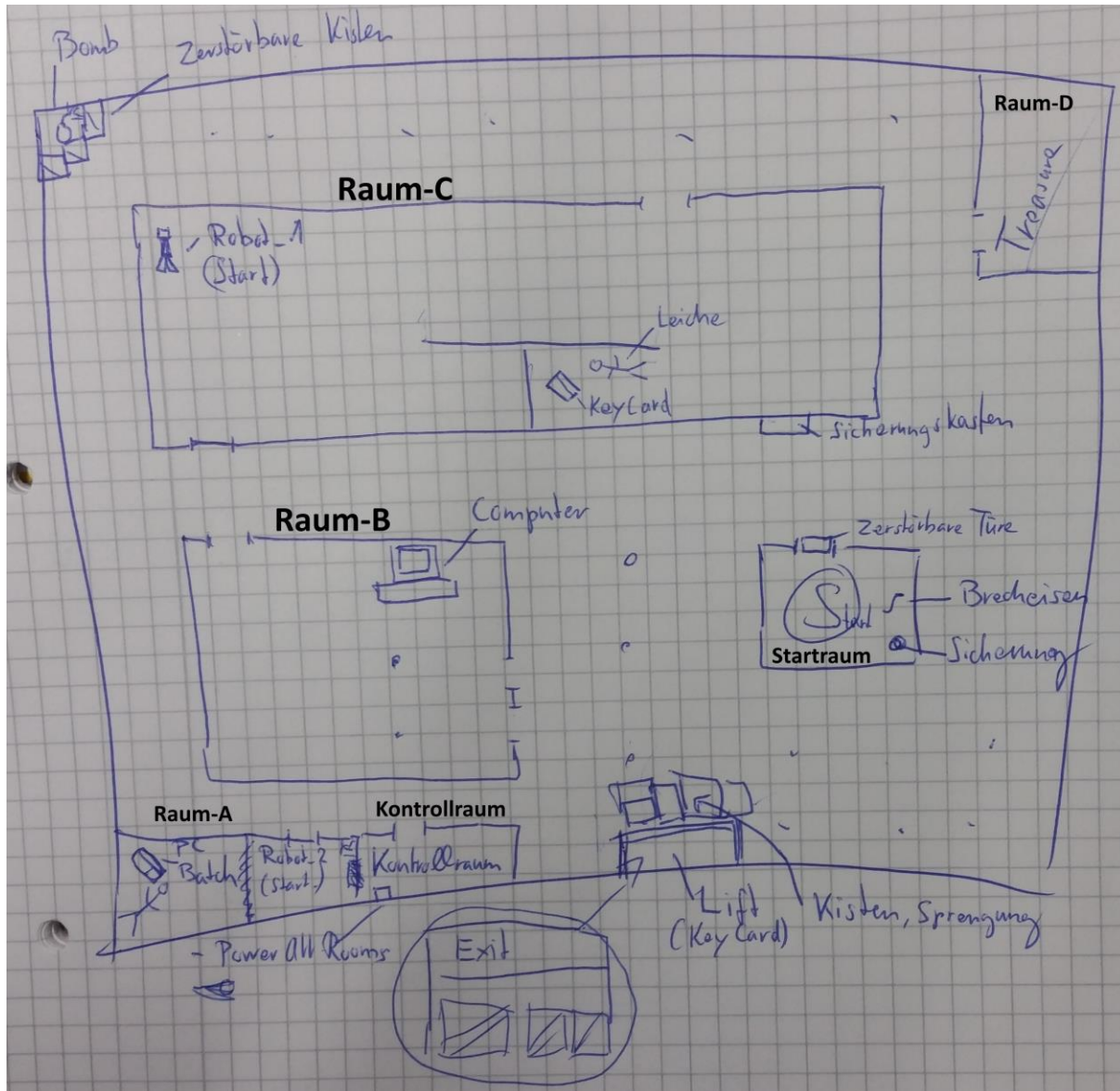


3. Nun sollte auch die dazugehörige Instanz in der Hierarchie bereits aktualisiert worden sein. Falls dies nicht der Fall sein sollte, eventuell die „Revert“ Option auf dem fraglichen Prefab auswählen.

5.3 Übersicht über die Komponenten des Levels

5.3.1 Levelplan

Eine in der Design-Phase erstellte Skizze, die den groben Aufbau des Basislevels beschreibt. Die Infrastruktur wurde so praktisch 1:1 umgesetzt und dient in diesem Dokument als Orientierung wenn in den folgenden Abschnitten auf die einzelnen Räume und Komponenten eingegangen wird.



5.4 Modellierung der Räume und deren Verhalten

Da es sich bei den Räumen um relativ statische und auch eher passive Elemente im Spiel handelt, wurden die Aktionen und Zustände der Räume nicht durch eine State Machine modelliert. Ihre Verhaltensmuster basieren auf einem Entscheidungsbaum und lassen sich mit einem Guard-Diagramm abbilden.

5.4.1 Situation beim Spielstart

Nachdem der Spieler aus dem Startraum entkommen ist, findet er das Level in Dunkelheit vor. Bis auf die rot beleuchtete Kontrolleinrichtung sind zu diesem Zeitpunkt alle Räume dunkel und abgeschlossen. In unmittelbarer Nähe befindet sich ein schwach beleuchteter Sicherungskasten. In diesen kann eine Sicherung eingelegt werden. Dies lässt es anschließend zu, dass im Kontrollraum die Levelbeleuchtung und die Stromversorgung der einzelnen Räume eingeschaltet werden kann.

Im Kontrollraum sind folgende Schalter zu finden:

- CORRIDOR POWER -> Default: OFF | Aktivierung bedingt die eingelegte Sicherung
Wird dieser Schalter eingeschaltet (und wurde die Sicherung vorher in den Kasten eingelegt, wird das Level durch die LevelLights1-7 beleuchtet
- POWER IN ROOMS -> Default: OFF | Aktivierung bedingt die eingelegte Sicherung
Wird dieser Schalter eingeschaltet, erhalten alle Räume Strom und es öffnen sich deren Türen. Ausnahme ist dabei Raum D (der Treasure-Raum) und der an den Kontrollraum angrenzenden Raum A. Licht brennt aber nach wie vor nicht in allen Räumen, sondern muss teilweise über die Lichtschalter manuell eingeschaltet werden.
- DO NOT PUSH -> Default OFF
Dieser Schalter öffnet die Türe zum angrenzenden Raum A. Zwar aktiviert diese Aktion den darin gefangenen Roboter frei, doch die in Raum A versteckte Liftkarte wird für den Abschluss des Levels zwangsläufig benötigt.

5.4.2 Status der Räume

Jeder Raum besitzt ein Objekt des Typs Environment, welches entscheidet, ob der Raum mit Strom versorgt wird oder nicht. Die Idee dahinter ist, für die Räume individuelle Zustände modellieren zu können wie z.B. Sauerstoffmangel beim Ausbruch eines Feuers oder Ähnliches.

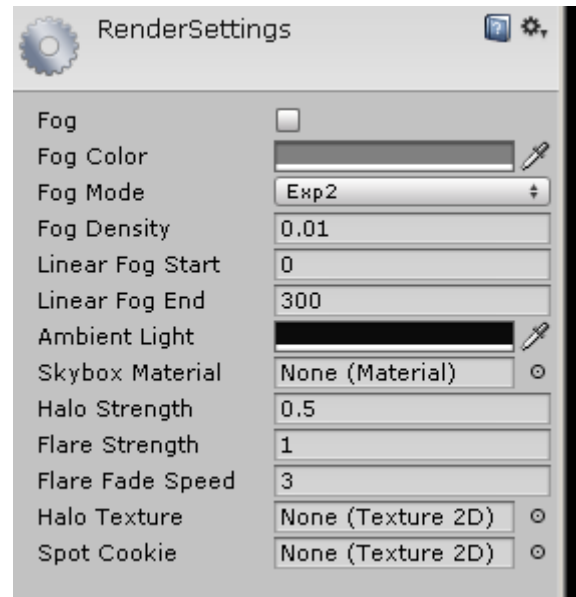
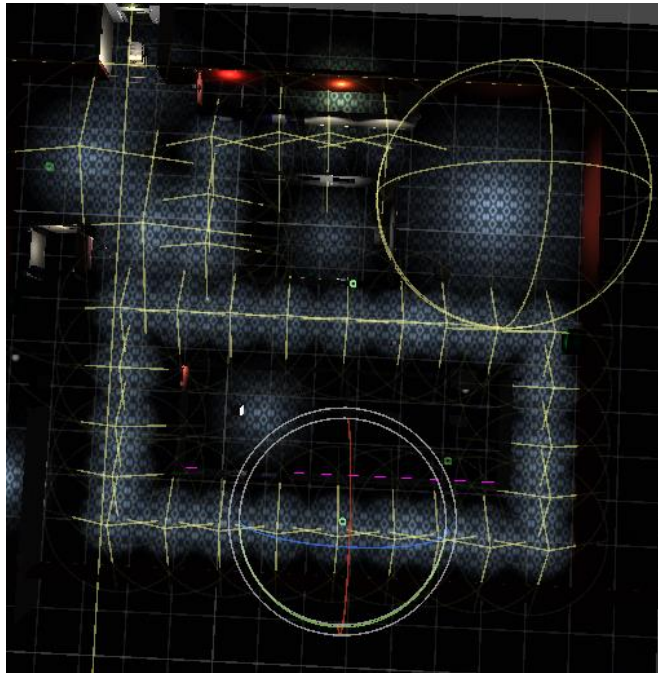
Die *Hauptstromversorgung der Räume* ist zu Beginn unterbrochen, wenn der Hauptschalter im Kontrollraum nicht auf ON ist. Dies bedeutet, dass zu Beginn des Spiels *das Attribut powerState* für die einzelnen Environment Objekte, mit Ausnahme jenes im Kontrollraum auf false geschaltet ist.

Wird der Hauptschalter für Räume („POWER IN ROOMS“ vgl. 5.3.1) aktiviert, wird eine Message an die Gruppe „Rooms“ geschickt, und die Räume reagieren auf den Empfang, in dem sie den powerState auf true schalten. Es funktionieren bis zum Versand der Message nur jene elektronischen Anlagen, die an der Notstromversorgung angeschlossen sind, was anfänglich nur für den Kontrollraum zutrifft. Weitere Räume, die an der Notversorgung angeschlossen sind,

sind nicht geplant. Falls diese doch realisiert werden sollten, käme am einfachsten ein weiteres Attribut: emergencyPower zur Environment Klasse hinzu, um die Steuerung dieser Einrichtungen zu vereinfachen.

5.4.3 Beleuchtung

Das Level wird insgesamt von 7 Lichtquellen beleuchtet. Es sind dies die Prefabs Lightning1-7.



Bei Spielstart präsentiert sich das Level in völliger Dunkelheit. Mit Hilfe der RenderSettings wurde die Umgebungsbeleuchtung auf ein Minimum reduziert, was ein Navigieren innerhalb des Levels ohne eingeschaltete Beleuchtung verunmöglicht. So wird der Spieler dazu gebracht, sich in den Kontrollraum zu begeben und von dort aus die weiteren Abläufe im Spiel zu durchlaufen.

Auf die Problematik mit einer zu hohen Anzahl Lichtquellen wird im Abschnitt ‚Erfahrungen‘ eingegangen.

5.5 GameController

Der GameController ist eine Instanz ohne physikalische Repräsentation im Spiel. Er soll der Überwachung des Spiels dienen und bei Bedarf Informationen zum Zustand des Spiels liefern und globale Auswirkungen steuern können, wie zum Beispiel das Aufrufen eines Menus während des laufenden Spiels. Auf dem GameController liegen folgende Scripts:

Scriptname	Funktion
MyUnitySingleton	Wird verwendet, damit ein Spielobjekt nicht gelöscht wird, wenn eine neue Szene geladen wird.
Skynet	(De)-Aktivierung der Robotergegner
MessageDispatcher	Zuständig für das Verteilen der Messages die bei den Empfängern unterschiedliche Aktionen auslösen.
Countdown	Am oberen linken Bildrand läuft ein Countdown von 400 Sekunden an abwärts. Ist das Level nicht vor dem Erreichen der 0 abgeschlossen, gilt das Spiel als verloren.
PauseMenu	Pausiert das Spiel
MouseCursor	Das MouseCursor-Script zeigt einen Pfeil für die Navigation innerhalb des GUIs dar.
TextDisplay	Mit Hilfe des TextDisplayer Scripts werden die Info-Boxen unterhalb der Objekte angezeigt, mit denen eine Interaktion stattfinden kann.
LastPlayerSighting	Wird vorwiegend zur Audio-Steuerung verwendet.

5.6 Spielfigur

5.6.1 Tastatur-Steuerung

Klassisch wird mit der WASD-Tastenbelegung gespielt. Daneben stehen dem Spieler folgende Steuermöglichkeiten zur Verfügung:

Funktion	Tastenbelegung	Wirkung
Gehen	W	Die Spielfigur bewegt sich vorwärts
Laufen	Shift+W	Die Spielfigur bewegt sich schneller vorwärts
Rückwärts gehen	S	Die Spielfigur bewegt sich rückwärts
Seitwärts gehen (l)	A	Die Spielfigur bewegt sich seitwärts nach links
Seitwärts gehen (r)	D	Die Spielfigur bewegt sich seitwärts nach rechts

Ducken	C	Die Spielfigur duckt sich – wichtig um einige Items aufnehmen zu können.
Lean-Look Links	Q	Der Sichtwinkel verändert sich, damit auf die linke Seite hin „um die Ecke“ geschaut werden kann.
Lean-Look Rechts	W	Der Sichtwinkel verändert sich, damit auf die rechte Seite hin „um die Ecke“ geschaut werden kann.
Pause	ESC	Das Spiel wird pausiert. Der Countdown stoppt und auch die Bewegungsmöglichkeiten der Spielfigur sind alle blockiert

5.6.2 Maus-Navigation

Mit der Maus wird auf zwei Varianten navigiert:

1. Mit dem Fadenkreuz, das während dem normalen Spielablaufs zu sehen ist
2. Mit dem klassischen Mauszeiger (modifiziert, so dass er als grüner Pfeil erscheint), der verwendet wird, um mit GUI Elementen zu interagieren (anklicken von Buttons etc.)

5.6.3 Inventar & Interaktion mit Objekten

Im Inventar werden alle gefundenen Gegenstände abgelegt. Mit Hilfe des Mausekzes kann durch das Inventar navigiert werden. Momentan aktive Gegenstände sind rot markiert und werden bei der nächsten Interaktion mit einem Objekt verwendet. Ein Klick mit der linken Maustaste auf ein Objekt gibt bei möglicher Interaktion eine Textbox aus und informiert über den Gegenstand und/oder über Aktionsmöglichkeiten. Diese Funktionalität ist in der Alphaversion noch nicht für alle Gegenstände umgesetzt worden.

5.7 Computer & Computercard

In Raum B befindet sich ein Computer, der vom Spieler unter der Bedingung des Besitzes der Computercard benutzt werden kann. In Raum C findet der Spieler neben dem Soldaten diese Computerkarte, mit der die Verwendung des Computers möglich wird.

Befindet sich die Computercard bei der Interaktion mit dem Computerobjekt im Inventar und ist sie ausgewählt (rot markiert), erscheint beim Anklicken des Computers ein Menu, welches folgende Optionen zur Auswahl stellt:

Optionsliste des Computers in Raum B:

a) Enable elevator

Aktiviert den Lift. Nebst dem Beseitigen des Hindernisses und dem Auffinden der Lift-Karte ist dies die dritte Bedingung um ein Verlassen des Levels zu ermöglichen.

b) Bomb code

Um die Bombe zu verwenden und mit ihr das Hindernis vor dem Lift wegsprengen zu können, bedarf es eines Codes. Bei jedem Reset des Menus wechselt dieser Code.

c) Open treasure room

Öffnet die Türe zum Schatzraum und gibt die wertvollen Items frei.

d) Paralyze robots

Weil die Wahrscheinlichkeit, die Bombe zu platzieren und dabei vom Roboter nicht erwischt zu werden sehr klein ist, kann mit dieser Option der/die Roboter paralysiert werden. In der Alpha-version ist das setzen der Bombe ohne Verzögerung möglich, was dieser Option weniger Bedeutung zukommen lässt. Wenn aber eine bestimmte Zeit für das Anbringen der Bombe benötigt würde und die Geschwindigkeit der Roboter erhöht wird, wäre diese Option spielentscheidend.

+) EXIT (verlässt das Menu)

+) REST (schliesst den Schatzraum wieder, reaktiviert die Roboter, ändert den Bomben-Code und sperrt den Lift)

Durch dieses Schema ist es fast zwingend, folgendermassen vorzugehen (wenn man die Roboterkonfrontation vermeiden will):

1. Auswahl Treasure Room + disable Robots
(-> Roboter danach manuell ausschalten)
2. Reset
3. Lift und Bombencode
4. Level verlassen

Anmerkung: Während dem das Menu geöffnet ist, pausiert das Spiel und auch die Navigation mit Maus und Tastatur wird blockiert. Der Computer fährt herunter (resp. das Menu wird ausgegraut), nachdem man 2 Optionen ausgewählt hat und er meldet, dass 2 Eingaben getätigt wurden.

5.8 Lift

Das Level wird vom Spieler über den Lift verlassen, der zu Beginn aber durch ein Gatter unerreichbar ist. Zudem blockieren mehrere Kisten den Zugang zum Lifteingang. Ein Schild oberhalb des Liftes informiert den Spieler, dass eine Liftkarte benötigt wird.

Mit Hilfe des Computers in Raum B, kann das Gatter entfernt und der Bombencode in Erfahrung gebracht werden. Die Bombe wird dazu verwendet, die Kisten weg zu sprengen. Ist dies erledigt, benötigt der Spieler nur noch die Liftkarte um die Lift-Türe öffnen zu lassen. Anschliessend kann er die Spielfigur in den nun zugänglichen Teil hinter dem Lift begeben, was die Schlusssequenz des Spiels einleitet.

Die Schlusssequenz liefert dem Spieler eine Statistik, die ihm in der Alphaversion Angaben zu seiner Spielzeit und den gefundenen Schätzen liefert. Aus dem Statistikbildschirm heraus hat man die Möglichkeit, das Spiel neu zu starten.

Der Spieler hat das Basis-Level damit erfolgreich abgeschlossen.

5.9 Gegnerischer Akteur

Unser Spiel soll gegnerische Akteure enthalten, die mit einer Hierarchical State Maschine in verschiedene Zustände überführt werden können und so unterschiedliche Tätigkeiten ausführen können.

Das Ziel ist hier, eine Grundlage zu schaffen, um später weitere Gegnertypen oder Gegnervarianten mit relativ wenig Aufwand erstellen zu können.

Bei unserem Spiel wird der gegnerische Akteur ab sofort als „Roboter“ bezeichnet. Der Roboter gilt in unserem Level grundsätzlich als Gegenspieler. Er versucht das Spiel zu beenden, in dem er den Spieler berührt, sobald er den Spieler sieht.

Dieses Vorgehen bei einem Roboter bietet sich an, da dieser ja intern auch nur „Routinen“ kennt und je nach Situation anwendet. Natürlich sind auch andere Gegnertypen denkbar, sind aber in der vorliegenden Version noch nicht umgesetzt.

5.9.1 Zustände

Der Roboter kann verschiedene Zustände einnehmen, die durch eine Hierarchical State Maschine, kurz HSM, geregelt werden. Die Zustände „States“ sind Verhaltensmuster des Roboters, um eine künstliche Intelligenz zu simulieren. Die States, die aktiv ausgeführt werden sind:

Nr.	Zustandsname	Beschreibung
01	TopLevelState	Bildet den „Root-Knoten“ aller States
02	AkkuState	Alle Aktionen, die der Roboter während der Akkulaufzeit (Batterielaufzeit) macht, sind unterhalb dieser State
03	PluggedInState	Alle Aktionen, die während des Aufladens passieren.
04	IdleState	Zwischen den Wegpunkten ist der Roboter untätig.
05	OnDutyState	Alle Protokolle wie ChasingState oder Patrol State laufen darunter.
06	RechargingState	Enthält das Verhalten, um den Roboter wieder aufzuladen, nachdem der Batteriestand zu niedrig ist. Der Roboter berechnet selbstständig die benötigte Restenergie, die er braucht, damit er die nächste Ladestation erreicht.
07	PatrolState	Jeder Roboter wird mit einer Wegpunkt-Route bei der Initialisierung ausgestattet. Die Patrouille ist sein „Standardverhalten“ und der Roboter kehrt immer wieder zur Patrouille zurück, nachdem er sich aufgeladen hat.
08	ChasingState	Sobald ein Roboter den Spieler sieht, werden alle Roboter über die aktuelle Spielerposition informiert. Die Roboter begeben sich zu dieser Position und halten nach dem Spieler Ausschau. Das Spiel ist beendet, sobald ein Roboter den Spieler erreicht.
09	SearchState	Wurde nicht mehr implementiert. Angedacht war, dass der Roboter weiter nach dem Spieler sucht, wenn er die Spur vom Spieler verloren hat.

Zustände („States“) sind ein sehr gutes Konzept, um einen Gegner zu steuern. Die grössten Vorteile sind die Wiederverwendbarkeit der Komponenten und die relativ einfache Erweiterung von bestehenden Zuständen. Für eine Weiterentwicklung könnten zum Beispiel verschiedene Robotertypen, menschliche Wächter oder Tiere wie Wachhunde zum Einsatz kommen.

5.9.2 Zustände in Unity mit HFSM

Die Zustände werden in der HFSM.cs Klasse initialisiert:

```
public HFSM (string userName)
{
    userObjectName = userName;

    public HFSM(string userName)
    {
        userObjectName = userName;
        //ACHTUNG REIHENFOLGE, initialisiere immer zuerst die States, die dann zu
        //level 0
        TopLevel = new TopLevel (this, this.TopLevel);
        AkkuState = new AkkuState (this, this.TopLevel);
        PluggedInState = new PluggedInState (this, this.TopLevel);

        //level 2
        OnDutyState = new OnDutyState (this, this.AkkuState);
        IdleState = new IdleState (this, this.AkkuState);
        InitState = new InitState (this, this.AkkuState);

        //level 3
        PatrolState = new PatrolState (this, this.OnDutyState);
        ChasingState = new ChasingState (this, this.OnDutyState);

        //level 2
        RechargingState = new RechargingState (this, this.PluggedInState);
        SearchState = new SearchState (this, this.AkkuState);

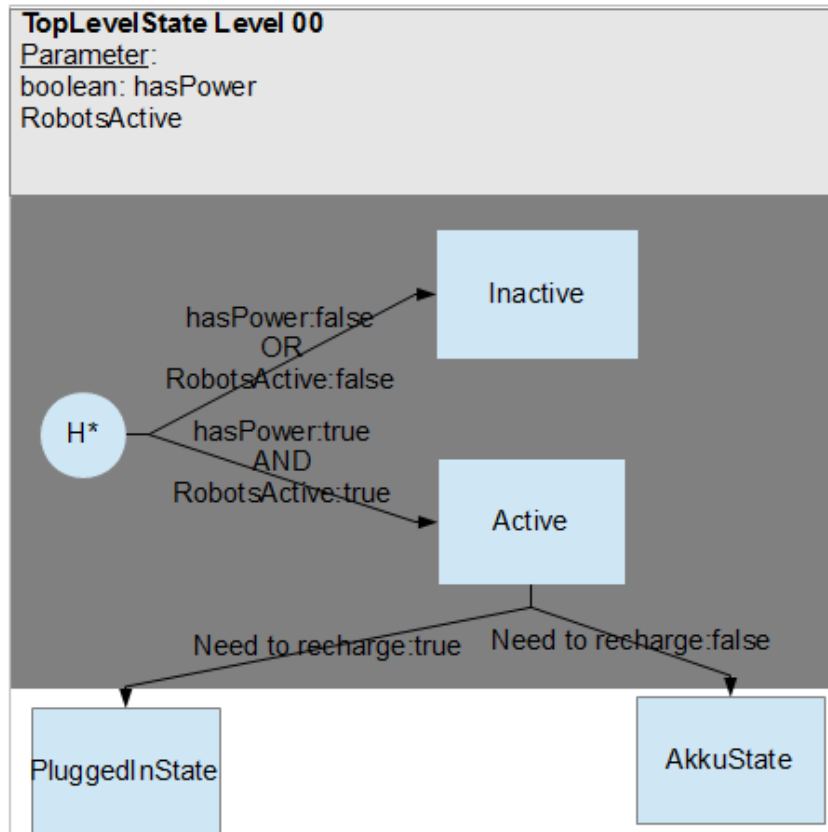
        this.currentState = TopLevel;
        this.previousState = TopLevel;

        this.currentState.entry ();
    }
}
```

Dabei muss die richtige Reihenfolge bei der Initialisierung beachtet werden. Es gilt, zuerst immer die oberen States zu aktivieren und anschliessend die darunterliegenden States zu aktivieren.

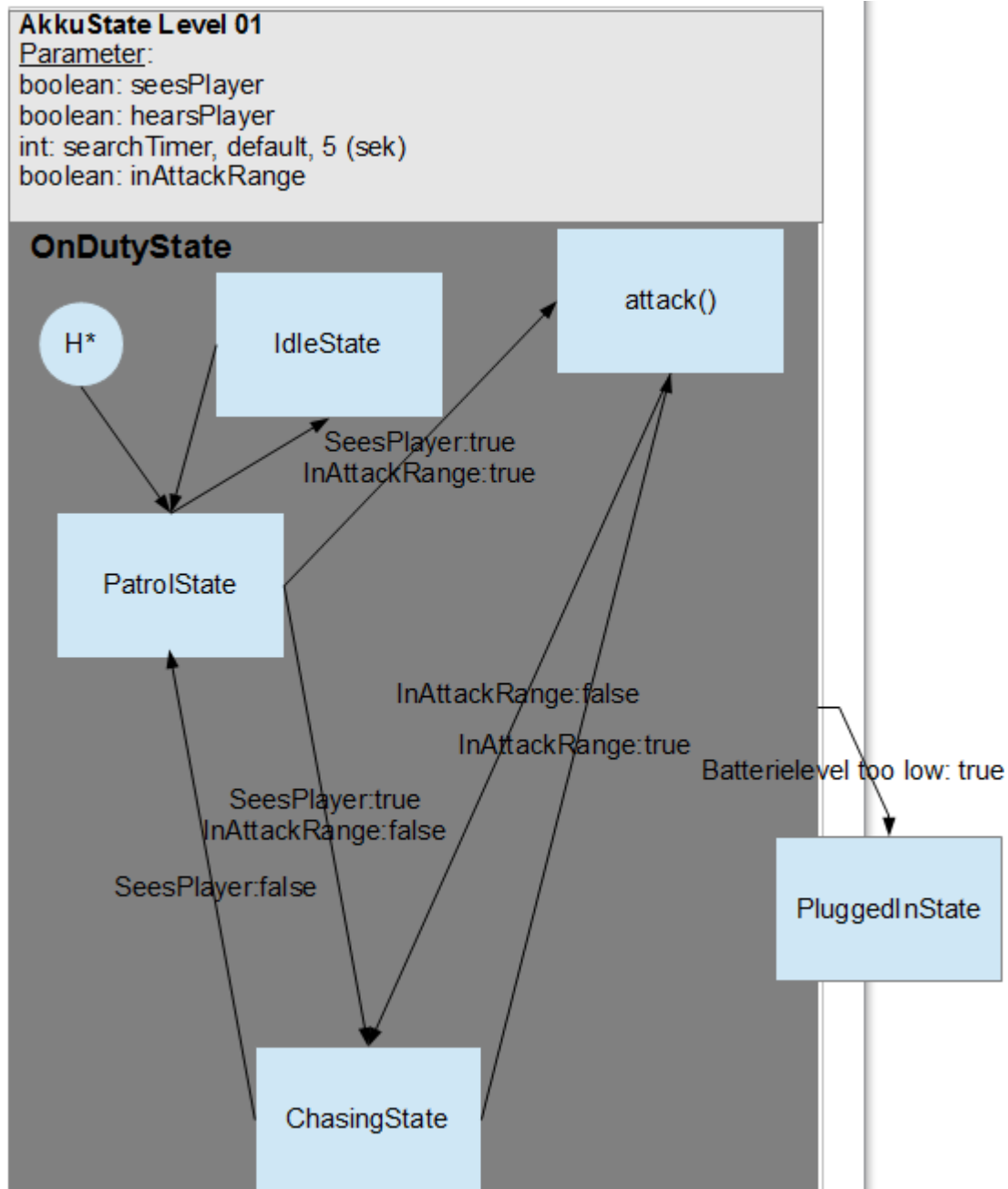
Das muss so sein, da jeweils der „Parent-State“ dem State bei der Initialisierung mitgegeben werden muss.

5.9.3 TopLevelState Level 00



Die TopLevelState stellt die grundsätzliche Entscheidung dar, ob der Roboter überhaupt aktiv ist. Dazu werden Variableninhalte verwendet, um festzustellen, ob ein Roboter aktiv ist. RobotsActive liegt auf dem GameController-Objekt und kann am Computer verändert werden. Der zweite Roboter wird ausserdem erst aktiv, wenn die Räume mit Strom versorgt werden und die Türen deaktiviert werden.

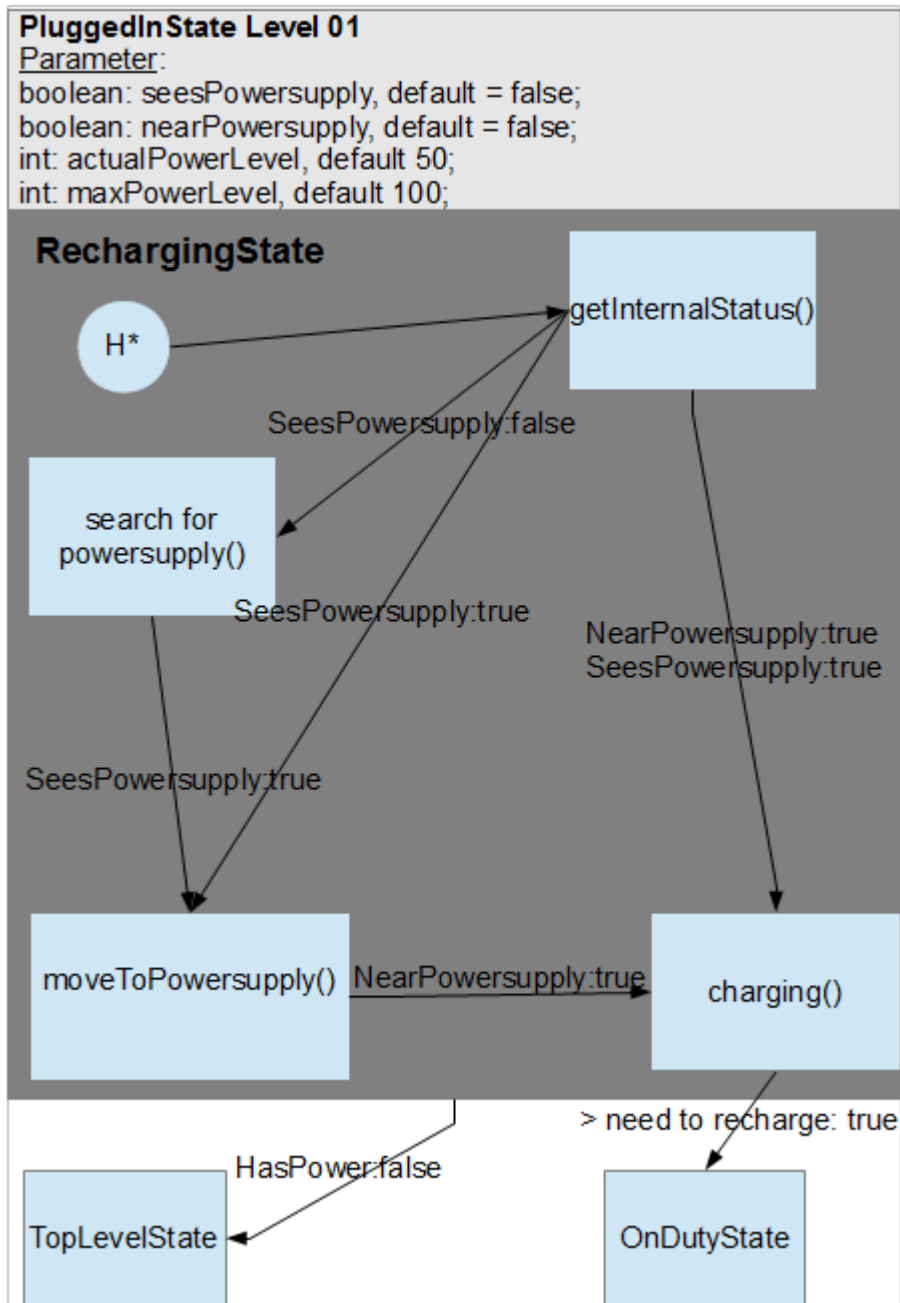
5.9.4 AkkuState



Der Startmodus ist der AkkuState Level 01 Modus. Dabei prüft der Robot sein „BatteryLevel“ und versucht immer frühzeitig zu einer PowerSupply-Station zu fahren und sich wieder aufzuladen. Dabei berechnet der Roboter, wie weit er noch „fahren“ kann mit seinem BatteryLevel und seiner Geschwindigkeit. Dem gegenüber steht die Distanz zur nächsten PowerSupply-Station, die der Roboter während seiner Bewegung immer wieder neu berechnet. Wenn also der BatteryLevel zu tief sinkt, wechselt der Roboter automatisch in die PluggedInState:RechargingState.

Sobald der Spieler in Reichweite kommt, werden die entsprechenden anderen States ausgelöst, um den Spieler zu verfolgen und anzugreifen.

5.9.5 PluggedInState Level 01



Das PluggedInState Level 01 fasst alle Aktionen zusammen, die der Roboter ausführen kann oder muss, um seinen Batteriestand (Powerlevel oder auch BatteryLevel) wieder aufzuladen. Das beinhaltet, die nächste Ladestation ausfindig zu machen, sich zu dieser zu bewegen und anschliessend so lange auszuharren, bis die Batterie vollständig aufgeladen ist. Anschliessend kehrt der Roboter wieder zu seiner letzten Aufgabe in der OnDutyState zurück. Sollte der Roboter nicht mehr aktiv sein, wechselt er auf TopLevelState::Inactive.

5.10 Robotereigenschaften und Verhalten

Es existieren in unserem Prototyp mehrere Verhaltensmuster des Roboters, die durch eine HSM (Hierarchical State Maschine) verwaltet werden.

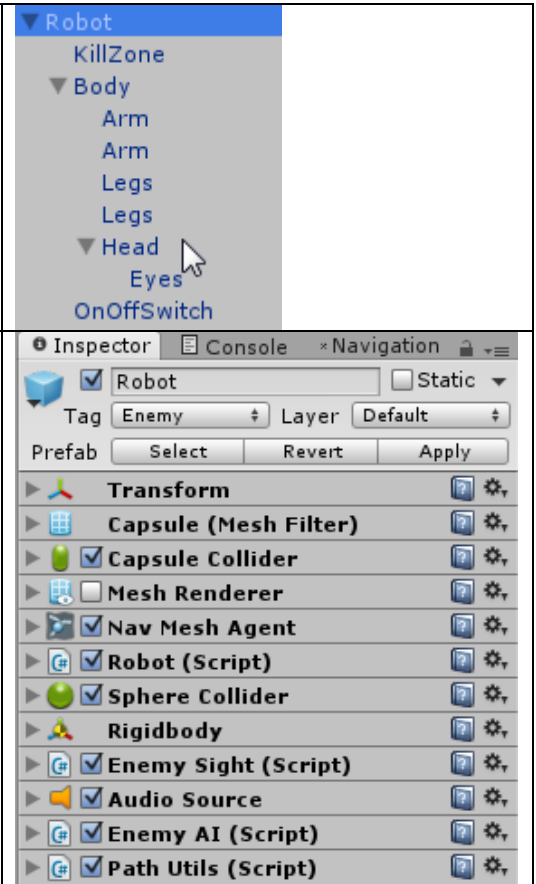
5.10.1 Der Roboter umgesetzt in Unity

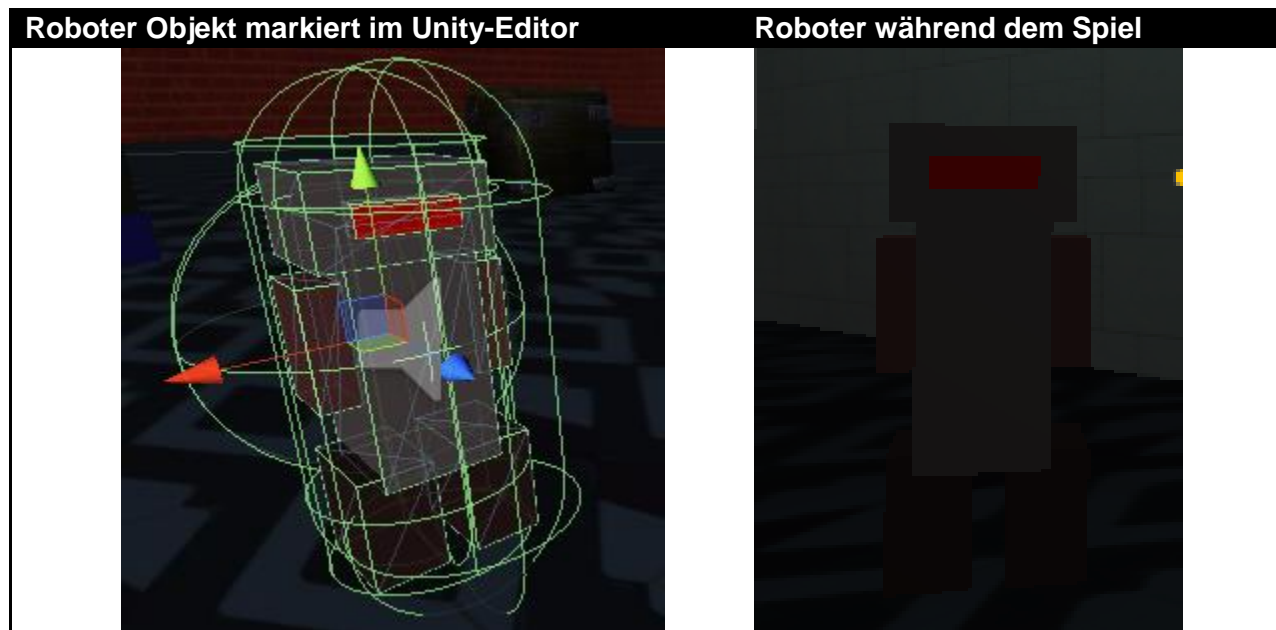
In Unity wird zuerst ein Modell gebraucht, welches den Roboter darstellen soll. Das ist bei uns sehr einfach gehalten: Der Roboter besteht aus rechteckigen Blöcken, die als Beine, Arme und Kopf dienen.

Weiter sieht man bereits zwei Roboterfunktionalitäten: Die KillZone und der OnOffSwitch.

Wie die meisten Objekte besitzt auch der Roboter einige Unity-Scripts, die die Aktionen im Spiel sammeln interpretieren und auswerten. Rechts eine Übersicht über alle Unity-Komponenten, die der Roboter besitzt:

Weiter unten gehe ich auf die wichtigsten Komponenten genauer ein.





Im Spiel wurde der Fokus auf Funktionalität gelegt. Es gibt bereits fix fertige Modells die auch Animationen enthalten. Wir wollten aber wo möglich unsere eigenen Items und Roboter erschaffen um alle Zusammenhänge zu verstehen.

5.10.2 Standard Komponente vom Roboter

Die folgenden Komponenten sind alles Standard-Unity Komponente und werden nur kurz erleutert:

Komponent Name	Beschreibung
Transform	Positionsdaten vom Roboter für x-,y- und z-Achse, die Drehung des Objekts und die Skalierung
Capsule	Das Mesh der Objekte
Capsule Collider	Der Collider des Roboters, der für physischen Kontakt in der Spielwelt gebraucht wird.
MeshRenderer	Das eigentliche Mesh des Roboters wird nicht gerendert, sondern nur die „Unterkomponente“ aus, denen der Roboter besteht.
Sphere Collider	Ist der generelle „Interaktions“-Radius des Roboters. Ein eingeschränkter Teil wird dazu verwendet, um die Sicht zu simulieren, wenn der Sphere Collider im vorderen Teil vom Spieler Objekt betreten wird.
Rigit Body	Wird gebraucht, um dem Roboter physikalische Attribute wie Gewicht zu geben. Wird später vom Nav Mesh Agent gebraucht für die Bewegung.
Audio Source	Damit der Roboter Geräusche erzeugen kann, muss er eine Audio Source besitzen. Dann sind seine Geräusche nur bis zu einem gewissen Abstand hörbar. Der Roboter erzeugt ein Peep-Geräusch, immer wenn er den State wechselt. EnemyAI:: changeState()

5.10.3 Robot (Script)

Enthält einzelne Roboterkomponente, die nur zum GegnerTyp „Roboter“ gehören. Das ist hier zum Beispiel „Batterielevel“ oder „RechargingSpeed“.

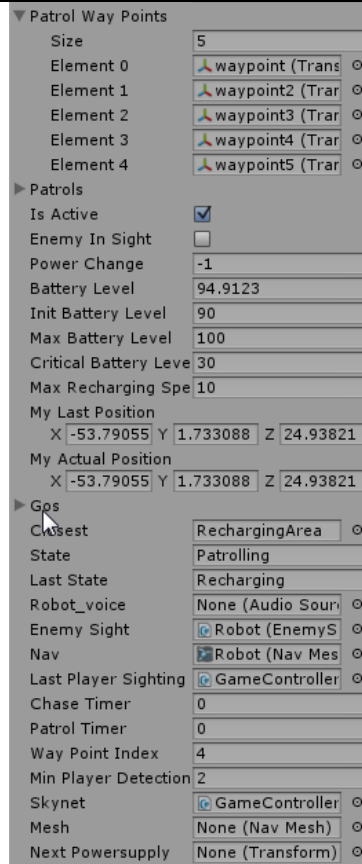
5.10.4 NavMesh and NavMesh Agent and

Damit sich der Robot mehr oder weniger frei im Level bewegen kann, braucht er eine „NavMesh Agent“ Komponente. Alle Objekte mit diesem „NavMesh Agent“ können sich auf dem NavMesh bewegen. Das NavMesh ist eine vordefinierte „Ebene“ im Level, die als „begehrbar“ definiert wurde. Grundsätzlich sind das alle Orte, die einen Untergrund besitzen und nicht von einem zu hohen statischen Objekt versperrt werden. Das sieht in unserem Projekt dann so aus:

5.10.5 Script: EnemyAI

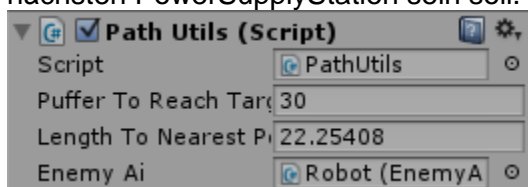
```
EnemyAI ▶ initIsActiv
1 using UnityEngine;
2 using System;
3 using System.Linq;
4 using System.Collections;
5
6 /* Here is a copy and an update from http://unity3d.com/learn/tutorials/projects/stealth/enemy-ai
7 Functions as the "Brain" of the Robot*/
8 public class EnemyAI : MonoBehaviour
9 {
10     public float patrolSpeed = 4f; // The nav mesh agent's speed when patrolling.
11     public float chaseSpeed = 6f; // The nav mesh agent's speed when chasing.
12     public float chaseWaitTime = 5f; // The amount of time to wait when the last sighting is reached.
13     public float patrolWaitTime = 2f; // The amount of time to wait when the patrol way point is reached.
14     public Transform[] patrolWayPoints; // An array of transforms for the patrol route.
15     public GameObject[] patrols; // An gameobject array to hold all patrol points
16
17     public bool isActive = false; //is the robot activ?
18     public bool enemyInSight = false; //is the enemy in sight?
19     public static bool initIsActiv = true; // what is the start value of the robot isActive-State
20     public static float normalPowerUse = -1.0f; // battery use under normal conditions
21     public float powerChange; // active powerChange, can change
22     public float batteryLevel = 0f; //batteryLevel - changes with every update
23     public float initBatteryLevel = 90f; //Starting value for BatteryLevel
24     public float maxBatteryLevel = 100f; //Maximum possible BatteryLevel
25     public float criticalBatteryLevel = 30f; //Level of Battery to change state to recharge
26     public float maxRechargingSpeed = 10f; //Maximum recharging speed - to limit the recharge Duration per Rob
27     public Vector3 myLastPosition; // an 3d position of the robot, last known position, so he can retu
28     public Vector3 myActualPosition; // an 3d position of the robot, actual position
29
30     public GameObject[] gos; //tagged objects
31     public GameObject closest; //closest object found
32     public string state; //state of robot
33     public string lastState; //last state, save for later use
```

Fast sämtliche Variablen werden in diesem Script initialisiert und verwaltet. Hier wird auch die Hierarchical State Maschine initialisiert sowie interne Roboterfunktionen bereitgestellt, die mehrmals von unterschiedlichen Scripts aufgerufen werden. (RoboterIsActive(), etc.)



5.10.6 PathUtils

Das Script ist eines der Roboter-Utility Scripts. Es berechnet den Weg zur nächsten PowerSupplyStation im Game und definiert auch, wie hoch der Puffer für das Erreichen der nächsten PowerSupplyStation sein soll.



5.10.7 Patrouille

Der Roboter läuft mehrere Wegpunkte ab, die bei der Initialisierung geladen wurden. Das ist sein Standardverhalten und wird ausgeführt, wenn kein Ereignis (Batteriestand, Spieler in der Nähe, etc.) dieses Verhalten überschreibt.

5.10.8 Sichtradius

Der Roboter hat einen Sichtradius von 260 Grad und eine Reichweite von 20 Meter. Sobald der Spieler diesen Bereich betritt und sich nicht hinter einem Objekt (Wand, Kiste, etc.) versteckt, wird der Status von "SpielerInSicht" auf "true" gesetzt. Das löst gleichzeitig den State "Chasing" aus, in dem der Roboter den Spieler einfangen will.

Auch denkbar ist hier eine Erweiterung der Sensoren um Geräusche wahrzunehmen.

5.10.9 Batterie

Das Energiemanagement des Roboters fügt ein weiter Faktor zum Verhalten des Roboters hinzu. Bei der Initialisierung startet der Roboter mit 70% der Maximalenergie.

Das Energiekonzept ist auch erweiterbar. Es ist denkbar, dass jedes Verhalten unterschiedlich viel Energie während der Aktivität benötigt. Kämpfen und Verfolgen würde demnach mehr Energie kosten als nur von Punkt A zu Punkt B zu patrouillieren.

5.11 Inventar und Items

Die Interaktion mit manchen Spiel-Objekten verlangt vom Spieler dass er ein bestimmtes Item in der Hand hält. Damit dies auch mit unterschiedlichen Items möglich und leicht erweiterbar bleibt, wurde ein Inventar implementiert.

5.11.1 Generische Itemklasse

Die Itemklasse ermöglicht es schnell und unkompliziert ein neues Item der Spielwelt hinzuzufügen. Man fügt dabei einfach das Items.cs Script auf das gewünschte Game-Objekt und gibt Label so wie das Icon für die Darstellung im Inventar an. Beim Interagieren wird dann automatisch erkannt dass es sich um ein aufsammelbares Game-Objekt handelt.

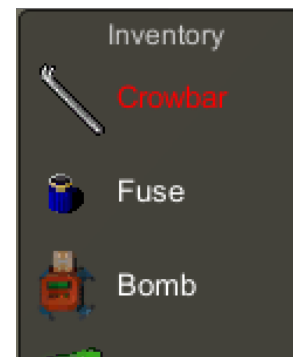


Abbildung Inventar

5.11.2 Info-Boxen

Damit der Spieler genauer erkennen kann was er betrachtet, wird ihm eine kleine Info-Box dargestellt mit dem Namen des Items. Diese Info-Boxen sind vom Typ "ProximityMessage" und werden von den Items bei deren Instanzierung automatisch generiert. Näheres dazu unter dem Abschnitt „Darstellung von Text-Meldungen“.

5.11.3 Auswählen des Items

Der Spieler hat die Möglichkeit das gewünschte Item durch Drehen des Mausekkrades auszuwählen. Das aktive Item wird rot dargestellt und wird bei einer Interaktion mit einem anderen Game-Objekt mitgeschickt. Näheres dazu unter dem Abschnitt MessageDispatcher.

5.12 Darstellung von Text-Meldungen

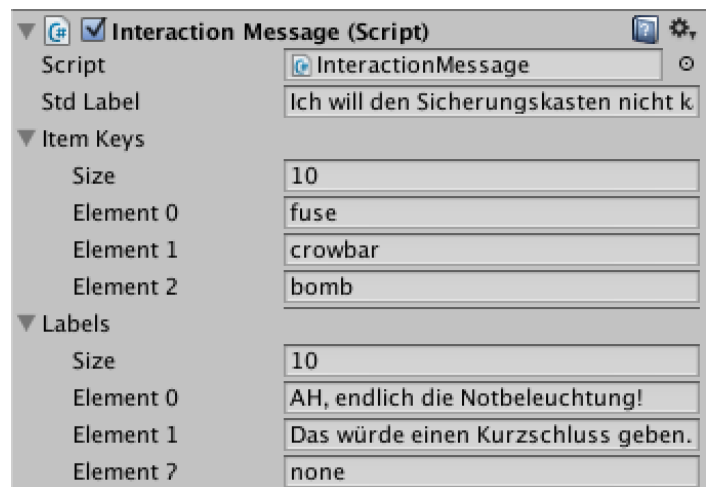
Für die Darstellung von Text-Meldungen wird die Klasse „TextDisplayer“ verwendet. Sie wählt unter Berücksichtigung von Priorität und Anzeigedauer die aktuell passendste Nachricht aus und stellt sie dar. So hat zum Beispiel die „InteractionMessage“ eine höhere Priorität als die „ProximityMessage“ und wird daher bevorzugt bei der Darstellung. In diesem Kapitel sind die Klassen vorgestellt welche den TextDisplayer verwenden.

5.12.1 ProximityMessage

Damit der Spieler erkennen kann, was er im Moment betrachtet wurde die Klasse „ProximityMessage“ erstellt. Sie kann zusammen mit einer gewünschten Meldung einem Game-Objekt zugewiesen werden. Die ProximityMessage verwendet dann bei der Betrachtung den TextDisplayer für das darstellen der Meldung.

5.12.2 InteractionMessage

Bei der Interaktion mit einem Game-Objekt kann eine InteractionMessage verwendet werden. Dies soll dem Spieler Hinweise zum verwenden des Game-Objekts geben. Zum Beispiel kann man über den TextDisplayer die Meldung ausgeben „Diese Türe versperrt mir den Weg!“, sobald der Spieler mit einer geschlossenen Türe interagieren will. Dabei kann je nach Item, welches der Spieler in der Hand hält, eine andere Meldung dargestellt werden. Die „InteractionMessage“ ist generisch aufgebaut, somit kann man über das GUI die unterschiedlichen Rückmeldungen pro Item definieren.



5.13 MessageDispatcher & Telegramm

Die Klassen „MessageDispatcher“ und „Telegramm“ werden verwendet um ein generisches Interface für die Kommunikation zwischen den einzelnen Game-Objekten zu bieten.

Möchte man eine Nachricht von einem Game-Objekt ans nächste schicken so benutzt man den „MessageDispatcher“. Dieser nimmt die Nachricht entgegen und verpackt sie in ein Telegramm. Das Telegramm beinhaltet nebst der Nachricht eine Empfänger- und Senderadresse, so wie eine Liste mit zusätzlichen Daten. Somit kann dann der Empfänger selber entscheiden wie er z.B. auf eine Interaktion mit dem Spieler reagieren soll. Weiter kann man dem „MessageDispatcher“ mitteilen er soll die Nachricht mit einer Verzögerung verschicken. Dies wird zum Beispiel bei der Bombenplatzierung verwendet für den Countdown der Bombe.

Telegram
Verzögerung
Sender
Empfänger
Nachricht
Daten

Abbildung Aufbau
Telegramm

5.13.1 Spielerinteraktion mit Game-Objekten

Klickt der Spieler während des Spieles, so wird entschieden ob man mit dem Game-Objekt interagieren kann welches man gerade betrachtet. Wenn ja, wird diesem Game-Objekt ein Interaktions-Telegramm verschickt. Dieses Interaktions-Telegramm wird über den MessageDispatcher verschickt.

5.14 EntityManager

Der EntityManager bietet ein globales Interface für die Adressierung von Game-Objekten. Ein Game-Objekt kann sich bei dieser Klasse mit einer ID anmelden um direkt angesprochen zu werden. Ausserdem bietet der EntityManager die Möglichkeit, dass ein Game-Objekt sich auch mehreren Gruppen registrieren kann.

5.14.1 Einsatz bei Telegramme

Wie das Kapitel „MessageDispatcher und Telegramm“ beschreibt, können Nachrichten zwischen Game-Objekten ausgetauscht werden indem man ein „Telegramm“ verschickt. In diesem Telegramm werden Sender und Empfänger über die im EntityManager definierten IDs definiert. Es ist also zwingend notwendig, dass der Empfänger sich als Objekt im EntityManager registriert hat.

6 Spielmenü

Zu jedem Spiel braucht es mindestens ein minimales Menü, um Einstellungen vorzunehmen oder um das Spiel zu starten. In Unity gibt es einen eigenen „Werkzeugkasten“, mit dessen Hilfe man einfach GUI – Elemente erstellen und für den Spieler sichtbar machen kann. Ein eigenes Menü zu entwerfen hat aber den Vorteil, dass die eigenen Komponenten der Schrift, Farbe und Geräuscheffekten verwendet werden kann. Wir verwenden in unserem Projekt beide Methoden:

6.1 Unity GUI-Elemente

Die Unity GUI-Elemente können Rahmen, Bilder und Beschriftungen erstellt werden. Mit der Funktion „onGUI“ werden diese dann über den eigentlichen Bildausschnitt gelegt. Das kommt zum Beispiel beim Timer oder beim Anzeigen des Inventars zum Einsatz. Das erlaubt dem Entwickler, Werte oder Bilder einfach und dynamisch zu gestalten und zu verwalten. Diese Eigenschaften unterstützt das Unity von Haus aus und die Elemente können bei jedem Unity-Frame neu gezeichnet werden.



Abbildung Inventar In-Game

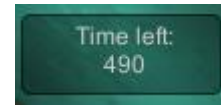


Abbildung Timer

6.2 Menü mit Spielobjekten und Kamera



Abbildung Startmenu Ansicht mit Maus auf "Load Level"

Das Spielmenü und Menüpunkte wie „Start Game“, „Load Game“ oder „Exit“ sind als Objekte in der Spielwelt modelliert, die durch eine fixe Kameraansicht wie ein Menü erscheinen. So können alle Effekte, Animationen und Tools von Unity genutzt werden, um ein ansprechendes Menü zu erstellen. Das Menü in der Grundstruktur besteht aus verschiedenen 2D-Objekten, auf welche die Kamera in einem senkrechten Winkel gerichtet ist. Im Editor sieht das dann so aus:

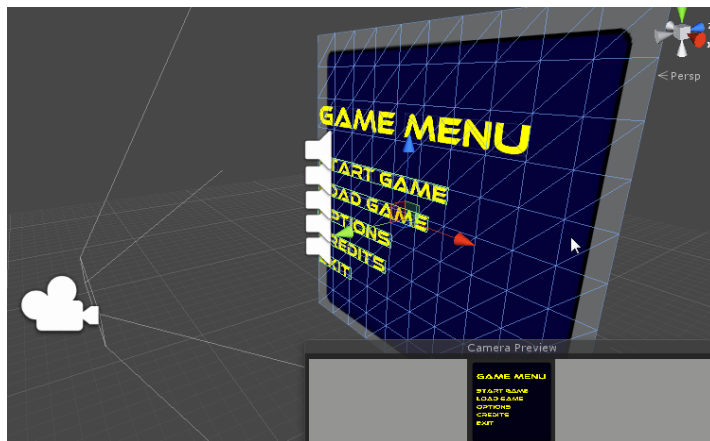


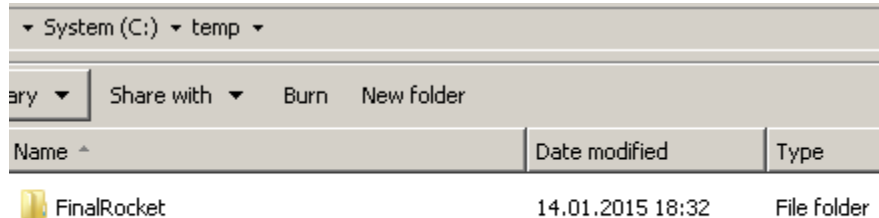
Abbildung Game Menu im Unity Editor

Das auch dieses Menü ein Prototyp sein soll, wurde für jeden Menüeintrag das gleiche *Prefab* verwendet. Darin ist definiert, dass ein Objekt auf eine neue *Szene* zeigt.

7 Setup des Projekts

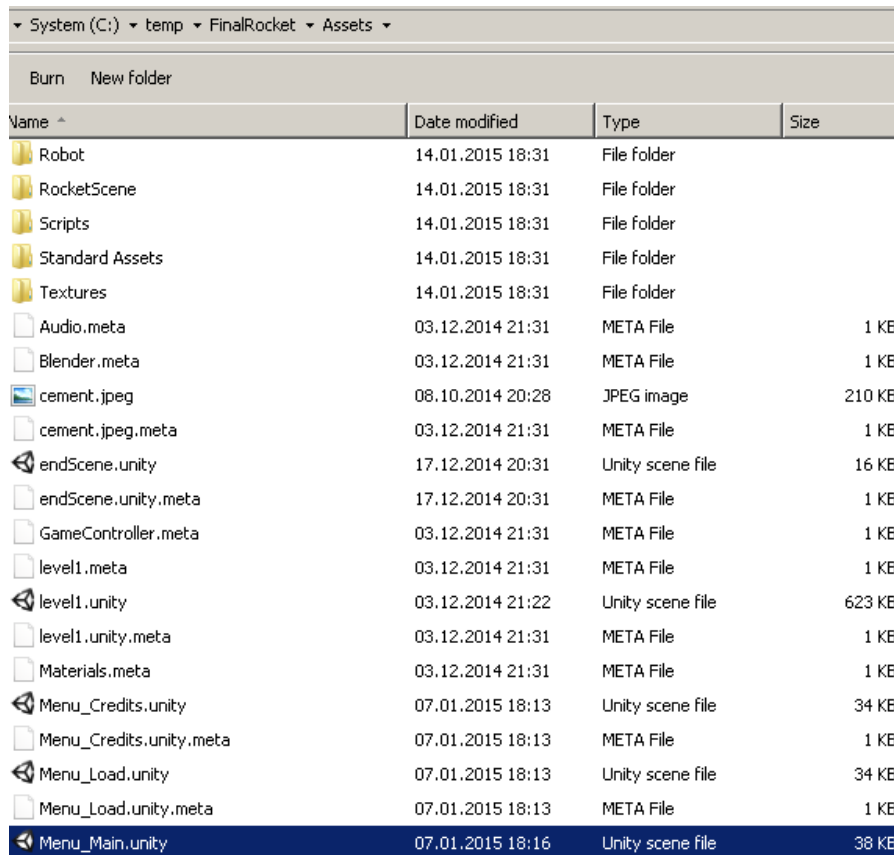
Das Projekt Rocket wurde mit Unity Version 4.5.4 erstellt. Wir empfehlen deshalb die Installation von 4.5 und höher, um unser Projekt zu laden und zu spielen.

Unter [LINK \(Rocket.zip\)](#) steht das Projekt Rocket zum Download bereit. Nach dem Entpacken des Archivs befindet sich nun der Ordner „FinalRocket“ auf Ihrem Rechner.

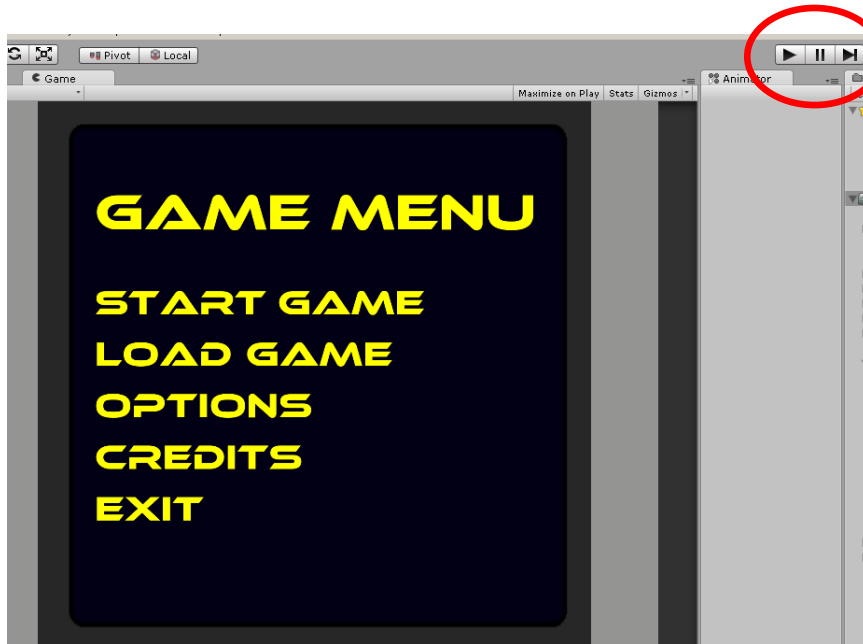


System (C:) > temp		
Share with	Burn	New folder
Name	Date modified	Type
FinalRocket	14.01.2015 18:32	File folder

Um direkt ins Spielmenu zu gelangen, wechselt man nach ..\FinalRocket\Assets und öffnet die Datei Menu_Main.unity per Doppelklick. Ist das Dateiformat .unity mit der Unityapplikation verknüpft, lädt dies nun die Szene des Main Menus. Ansonsten muss das .unity Format mit „öffnen mit“ mit der Unity Applikation verknüpft werden. Alternativ kann man die Menu_Main.unity-Datei aus der Applikation via Datei – Open Scene öffnen.



System (C:) > temp > FinalRocket > Assets			
Burn New folder			
Name	Date modified	Type	Size
Robot	14.01.2015 18:31	File folder	
RocketScene	14.01.2015 18:31	File folder	
Scripts	14.01.2015 18:31	File folder	
Standard Assets	14.01.2015 18:31	File folder	
Textures	14.01.2015 18:31	File folder	
Audio.meta	03.12.2014 21:31	META File	1 KB
Blender.meta	03.12.2014 21:31	META File	1 KB
cement.jpeg	08.10.2014 20:28	JPEG image	210 KB
cement.jpeg.meta	03.12.2014 21:31	META File	1 KB
endScene.unity	17.12.2014 20:31	Unity scene file	16 KB
endScene.unity.meta	17.12.2014 20:31	META File	1 KB
GameController.meta	03.12.2014 21:31	META File	1 KB
level1.meta	03.12.2014 21:31	META File	1 KB
level1.unity	03.12.2014 21:22	Unity scene file	623 KB
level1.unity.meta	03.12.2014 21:31	META File	1 KB
Materials.meta	03.12.2014 21:31	META File	1 KB
Menu_Credits.unity	07.01.2015 18:13	Unity scene file	34 KB
Menu_Credits.unity.meta	07.01.2015 18:13	META File	1 KB
Menu_Load.unity	07.01.2015 18:13	Unity scene file	34 KB
Menu_Load.unity.meta	07.01.2015 18:13	META File	1 KB
Menu_Main.unity	07.01.2015 18:16	Unity scene file	38 KB



Über den **Play Button** lädt es das GameMenu.

Unter Umständen wird in der Voransicht der Scene noch nicht wie links das Bild des Game-Menus eingeblendet, sondern erst nach dem man auf den Play Button gedrückt hat. Das hängt davon ab, wie man die geladene Szene betrachtet. Bitte sicherstellen, dass man sich in der richtigen Szene, Menu_Main.unity befindet.



Man erkennt das laufende Spiel nun daran, dass oben links der Countdown Timer zu sehen ist und man mit der Maus das Spiel über „Start Game“ starten kann.

7.1 Build & Run

Auch ein Build (Im Unity ist die Option über File – Build&Run zu erreichen) wurde von der Alphaversion erstellt und als .exe auf seine Spielbarkeit getestet. Konfigurierte Einstellungen zum Build:

Plattform: PC, Mac & Linux Standalone
Target Platform: Windows
Architecture: x86_64

Folgende Bugs wurden im Build festgestellt, aber auf Grund von Zeitmangel nicht behoben:

- Die TextMessage-Funktion, die Beschreibungen von Objekten und Interaktionsmöglichkeiten am Bildschirm darstellt, funktioniert im Build nicht.
- Die Beleuchtung in Raum A funktioniert nicht.
- Auf den Texturen der Wände sind teilweise fehlerhafte, schattenartige Muster zu sehen

Da ein Build (d.h. eine fehlerfrei spielbare *.exe Datei) nicht das Ziel des Projekts war, haben wir für dessen Debugging kaum Zeit aufgewendet. Es muss aber auch erwähnt werden, dass sich ein Debugging einer kompilierten .exe Datei als sehr schwierig gestalten würde, da man kaum Möglichkeiten hat mit einer Konsole-Ausgabe oder ähnlichen Tools zu arbeiten.

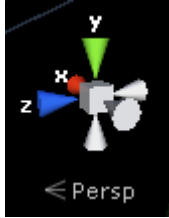
8 Erfahrungen

8.1 Unity

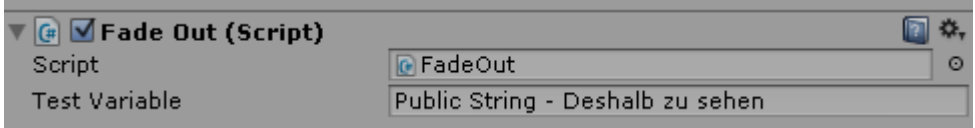
8.1.1 Genutzte Features & Funktionen in Unity

Unity unterstützt die drei Programmiersprachen C#, JavaScript (genannt UnityScript) und Boo. Wir haben für unser Projekt mit einer Ausnahme (der Steuerung) ausschliesslich mit C# gearbeitet. In diesem Kapitel werden die wichtigsten Bestandteile und Komponenten der Entwicklungsumgebung beschrieben, mit denen wir gearbeitet haben und wie diese eingesetzt wurden. In den vorherigen Kapiteln, besonders in Kapitel 5 wo die Spielelemente beschrieben werden, sind ebenfalls bereits einige Erklärungen zu eingesetzten Elementen und Funktionen beschrieben. Diese kurze Zusammenfassung gibt auch Einsteigern einen kleinen Einblick in (einen Bruchteil) der Möglichkeiten, die einem Unity zu bieten hat.

Elemente / Funktionen	Einsatzgebiet, Beschreibung wie es verwendet wurde
MonoBehaviour	<p>Die MonoBehaviour Klasse ist die Parentklasse, wovon alle Scripts erben sollten.</p> <p>Tun sie dies nicht, stehen einem im UI keine proprietären Methoden und Variablen von Unity zur Verfügung. Das bedeutet ausserdem auch, dass man das Script keinem GameObject zuweisen kann. In unserem Projekt gibt es einige Ausnahmen, die für sich eigene Klassen bilden, wie bsp. die Environment Klasse, die im Grunde nur eine Sammlung von Attributen ist. Sobald aber das Script mit anderen Unity Komponenten kommunizieren soll, oder es auf ein GameObject verknüpft wird, muss man die Vererbung von MonoBehaviour deklarieren.</p>
GameObject	<p>Die Klasse wovon alle Objekte in Unity (Cylinder, Texturen, Characters etc.) erben. Instanzen dieser Klasse (eben die Elemente wie Würfel etc.) sind eigentlich Container und enthalten nebst einigen Attributen (bsp. ob es sich um statische Objekte wie Wände handelt) die Components wie bsp. Collider oder Mesh Renderer.</p> <p>Um einzelne GameObjects wie bsp. unseren Roboter zu verwenden, kann man sie entweder über den Tag (frei wählbare Chars) oder über den Namen des Objekts ansprechen.</p>
void Awake() - Funktion	Dient dazu, ein Script zu initialisieren. Es ist so etwas wie der Ersatz für einen Konstruktor, den man auf Grund der Vererbung von der MonoBehaviour Klasse nicht überschreiben darf.
void Start() - Funktion	Unterscheidet sich von der Awake-Funktion dadurch, dass sie nur aufgerufen wird, wenn das Skript aktiviert ist. Wenn dies nicht der Fall ist, wird nur Awake() ausgeführt (die Bedingung ist natürlich, dass das Script einem GameObject zugewiesen wurde).
void Update() - Funktion	Die Update Funktion ist Teil der MonoBehaviour Klasse und wird in jedem Frame aufgerufen. Sie ist DIE zentrale Funktion, in der wohl der grösste

	<p>Teil des Codes eines Script zu finden ist. Alles was laufend ausgeführt werden soll, ist also dort zu finden.</p> <p>Wenn realistische & vor allem fließende Bewegungen realisiert werden sollen, empfiehlt es sich aber die <i>Time.deltaTime()</i> Funktion zu verwendet, die den Aufruf nicht pro Frame, sondern in einem fixen Zeitintervall abhandelt. So kann vermieden werden, dass ein Frame welches länger ausgeführt wird, die Bewegungen unrealistisch erscheinen lässt. Verwendet haben wir dies unter anderem beim Fade-In des Bildschirms zu Beginn, bei der Steuerung und im LeanLook-Script.</p>
GUI-Rendering mit der OnGUI() - Funktion	<p>Prinzipiell kann aus jedem Script heraus ein GUI erzeugt und ausgegeben werden. Um solche Ausgaben wie Buttons oder Textboxen zu erzeugen muss zwangsläufig die <i>OnGUI()</i> Funktion überschrieben werden. Diese wird wie die Update Methode bei jedem Frame aufgerufen.</p> <p>Für die Ausgabe von GUI-Elementen müssen jeweils auch gleich die Position in Form von Koordinaten angegeben werden. Deshalb arbeitet man am besten mit relativen Grössen, da sich die Auflösung resp. die Bildgröße ja ändern kann.</p> <p>Wir haben die <i>OnGUI()</i> Funktion bei unseren Menus verwendet, die während der Laufzeit eingeblendet werden können. Mehr als Buttons, einfache Textboxen und etwas Styling ist aber kaum machbar. Mit den GUI Layouting Optionen kann man die Positionierung und Skalierung der Elemente einigermaßen brauchbar kontrollieren. Es fehlen aber nützliche Elemente wie Tabellen und Graphen. Gerade für die Statistik am Ende hätten wir so etwas gerne verwendet.</p>
Transform	<p>Alle Game Objekte besitzen eine Transform Komponente. Damit wird die Grösse und die Lage des Objektes bestimmt. Will man mit der Position arbeiten, d.h. diese manipulieren, muss man das position-Attribut der Transform-Komponente neu setzen. Bsp.: <i>transform.position.set(1f, 1f, 5f)</i> Die Parameter sind float-Werte und entsprechen den x,y,z-Achsen der Unity-Welt entsprechen. Wie das Objekt zu einem bestimmten Zeitpunkt ausgerichtet ist und in welche Richtung die einzelnen Achsen zeigen, kann zu Beginn etwas verwirrend sein. Am besten orientiert man sich immer am kleinen Kompass oben rechts im Bild der Szene:</p> <p>In Verbänden als Prefabs können die Transform-Komponenten auch Parents besitzen, was zur Folge hat, dass eine Änderung der Skalierung am Parent auch die Kinder im Prefab beeinflusst. Unsere Räume sind Aggregate von mehreren Cubes. Wird der Raum als Ganzes markiert und sein Transform manipuliert, skalieren sich auch alle Kind-Komponenten. In diesem Fall die Wände und Türen.</p>  <p>Auch bei der Umsetzung des "Blicks um die Ecke" wurde schlussendlich mit der Manipulation der Kameraposition gearbeitet um den Eindruck eines sich zur Seite beugenden Spielers zu erwecken. Eine Bewegung des Spielers mit Hilfe der Quaternion-Klasse</p>

	(http://docs.unity3d.com/ScriptReference/Quaternion.html) zu realisieren scheiterte indessen an ungenügendem Verständnis der mathematischen Grundlagen.
Collider	Collider in ihren verschiedenen Formen (Box, Mesh, Sphere) werden auf Objekte gelegt, damit Kollisionen mit anderen Objekten die einen Collider besitzen möglich gemacht werden. Überschneiden sich zwei Collider (meist durch Berührung der Objekte) kann die <i>OnCollision</i> -Funktion genutzt werden. Bei einer Erweiterung des Levels durch die Möglichkeit des Waffengebrauchs und dem Treffen des Ziels würden wir diese Funktion einsetzen. Bei der Sprengung der Kisten, die den Ausgang versperren wurden von Martin auch Tests gemacht, wie herumfliegende Objekte (durch die Explosion weggeschleudert) den Spieler „beschädigen“ könnten. Aus Zeitgründen hat es dieses Feature aber nicht in die Alphaversion geschafft
Getriggerte Collider	<p>Im Gegensatz zu den Kollisionen werden die Collider.OnTrigger-Funktionen in Rocket häufig eingesetzt. In früheren Versionen für die Betätigung der Lichtschalter: Jeder Lichtschalter besitzt einen Collider der sich in einem bestimmten Umfang um den Schalter legt. Tritt der Spieler mit seiner Figur in die Nähe des Schalters, betritt er den Colliderbereich. Da diese Collider getriggert sind, wird die Methode OnTriggerEnter() aufgerufen, die wir so überschrieben haben, dass es prüft, ob es sich beim „Eindringling“ in der Triggerzone um den Spieler handelt und nicht um den Roboter. Ist das Objekt die Spielfigur, konnte der Lichtschalter ein- resp. ausgeschaltet werden.</p> <p>Abgelöst wurde dieses Konzept durch den Versand von Nachrichten – Siehe Kapitel „MessageDispatcher & Telegramm“. Beim Betätigen des Lichtschalters wird ein Telegramm mit dem Absender „Player“ an den Schalter geschickt. Die onMessage() Methode des Lichtschalter-Scripts prüft den Absender & schaltet sich ein, wenn es sich beim Absender wirklich um den Player handelt.</p>
NavMesh	Die NavMesh Komponente dient der Navigation von sich selbstständig bewegendem Objekten, damit diese den richtigen Pfad durch das Level finden und nicht in Wänden oder andere Gegenstände stossen. Umgangen werden von den mobilen Figuren alle als statisch gekennzeichneten Objekte. Nachdem man Wände, Kisten etc. als statisch markiert hat, wird die Navigationskarte (das „NavMesh“) „gebacken“, damit diese dem NavAgent, der auf den Robotern als Komponente vorliegt, als Karte dienen kann.
NavMesh-Agent	Der NavMesh-Agent kontrolliert die Attribute der Bewegung unseres Roboters, wie Geschwindigkeit und Beschleunigung. Über ein Script, welches auf den Robotern liegt, werden dem NavMesh-Agent laufend die neuen Ziele (Waypoints) seiner Bewegungen mitgeteilt.
RigidBody	Dem FirstPersonController, unserem „Player“ wurde die Komponente RigidBody hinzugefügt. Eine RigidBody-Komponente fügt einem Game Objekt eine Masse hinzu und unterwirft es dadurch den „Gesetzen“ der Schwerkraft, d.h. hier der Physik-Engine von Unity. Durch die RigidBody-Komponente hat der Entwickler die Möglichkeit zu bestimmen, wie sich das Objekt bei Kollisionen verhalten soll (um Kollisionen detektieren zu können benötigt es aber zusätzlich noch ein Collider Objekt). Man kann via Script

	<p>Kräfte auf den Körper einwirken lassen. Wir haben dies bsp. bei der Explosion der Bombe ausprobiert, in dem wir nach der Explosion eine Kraft auf den Spieler wirken liessen, wenn er sich in einem bestimmten Radius befindet. Wir bereits erwähnt, wurde dieses Feature aber nicht in die Alphaversion aufgenommen.</p>
Visualisierung der Variablen aus den Skripts im UI von Unity	<p>Wird in einem Script eine Variable als public definiert, ist sie nach erfolgreicher Kompilierung des Scripts im Unity UI zu sehen. Der zugewiesene Wert wird ebenfalls angezeigt, sobald die Variable das erste Mal verwendet wird:</p> 
Time.deltaTime	<p>Diese Funktion verwendet man, wenn man eine Aktion nicht abhängig von der Framerate (d.h. in Abhängigkeit des update() Aufrufs) sondern abhängig der effektiv vergangenen Zeit machen will. Wenn ein Wert mit time.deltaTime multipliziert wird, wird das Produkt jeweils pro Sekunde neu berechnet und nicht bei jedem Frame.</p>
Zugriff auf Variablen anderer Skripts	<p>Option1: Die Variable in einem bestimmten Script kann über die gleiche Methode wie eine Komponente geholt werden: <code>GameObject.FindGameObjectWithTag("TAG").GetComponent<StartRoom>().test = "Hallo";</code> Das bedingt aber, dass die Variable public ist! Ansonsten bleibt einem nur die Möglichkeit mit get/set-Methoden zu arbeiten.</p> <p>Option2: Mit Hilfe von statischen public Variablen kann der ScriptName direkt aufgerufen werden. Für die Klasse StartRoom sieht das so aus: <code>StartRoom.staticTest = „TEST“</code> So wird von einem zweiten Script die Variable staticTest im StartRoom Script verändert und ändert sich somit überall, wo das Script verwendet wird.</p>
Zugriff auf Komponenten	<p>Um eine Komponente, bsp. den BoxCollider eines Objekts zu verändern, fügt man dem Script wo man die Manipulation ausführen will eine Membervariable vom Typ Component hinzu und verweist auf die gewünschte Komponente: <code>GameObject.FindGameObjectWithTag("TAG").GetComponent<BoxCollider> ();</code></p> <p>Anmerkung: Um die HaloKomponente eines GameObjects zu erhalten, muss diese als Behaviour-Instanz gehandhabt werden. Eine Component Variable akzeptiert keine Referenz auf eine HaloKomponente! So passiert beim Ausschalten des Leuchtens der Roboter.</p> <p><i>this.HaloRobi = (Behaviour)this.Robot1.GetComponent("Halo");</i></p>
Debugging Logging	<p>Mit <code>Debug.Log()</code> oder der von der MonoBehaviour Klasse zur Verfügung gestellten statischen Funktion <code>print()</code> können Ausgabewerte in der Konsole erzeugt werden. Wir haben diese Funktion häufig genutzt.</p>
Application.LoadLevel()	<p>Mit dieser Funktion kann ein Level (Scene), die man vorher in die Buildsettings aufgenommen hat, mit ihrem Namen geladen werden. So ist ein Levelwechsel möglich. Wir realisieren damit auch den Wechsel vom</p>

	<p>MainMenu ins Spiel. Im Zusammenhang mit dem Laden eines Levels sind folgende Dinge anzumerken:</p> <ul style="list-style-type: none"> ▪ Unity setzt beim Laden alle Attribute der Objekte einer Szene auf ihre Default-Werte zurück. Es reaktiviert aber keine (während des vorhergegangenen Spielablaufs aus dem das LoadLevel() aufgerufen wurde) deaktivierte GameObjekte. Das führte bei einem ausgewählten Neustart nach Abschluss des Levels zu Problemen, da während des Spiels einige Komponenten und Objekte deaktiviert und / oder zerstört wurden. ▪ Membervariablen von Scripts, die nicht mit GameComponenten verknüpft sind, werden in ihrem originalen Zustand belassen. Aus diesem Grund wurde die Instanz unseres EntityManagers nicht neu gebaut und behielt die Referenzen des vorhergehenden Spiels gespeichert. Diverse Fehler tauchten im Log auf. Durch das manuelle Instanzieren der Entities-Hashtable wurde das Problem behoben. <p><i>EntityManager.Entities = new Hashtable();</i></p>
--	---

8.1.2 Erwähnenswerte Einstellungen

Lichtprobleme: Flackern bei zu hoher Anzahl an Beleuchtungsobjekten

Unser Level wird von mehreren Spotlights erhellt. Wir stellten Graphikfehler fest, nachdem die Anzahl der Beleuchtungsobjekte pro Bereich eine bestimmte Menge überschritten hatte. Kritisch wurden die Werte wenn im Umkreis von 10 Units mehr als eine Lichtquelle installiert wurde. In der Szene waren keine negativen Auswirkungen einer hohen Anzahl an Quellen erkennbar. Wurde das Spiel aber gestartet, stellte man an den Texturen des Bodens und den Wänden ein Flackern fest. Behoben werden kann das Problem durch folgende Einstellung:

a) Ausschalten der geworfenen Schatten der Lichter ->Allg. ein Performancegewinn

b) durch Manipulation der Quality Settings unter „Edit“ – „Project Settings“ – „Quality“ und dem erhöhen der Anzahl Lichter die in der laufenden Szene Einfluss auf das Texturrendern haben. Diese Erhöhung ist aber mit einem Performanceverlust verbunden. Diese Einstellung hat nur einen Einfluss auf die real-time Berechnung der Lichter. Option d) macht diese Massnahme hinfällig.

Trotzdem haben wir von dieser Einstellung am meisten profitiert und versuchten ein Mittelmaß zu finden, dass der Beleuchtung, wie aber auch der Performance Rechnung trug.

c) Vermeiden von Interferenzen der einzelnen Lichtkegeln: Wird ein Mesh von mehreren Objekten gleichzeitig beleuchtet, berechnet Unity jedem Frame dessen Helligkeit auf Grund der verschiedenen Lichtquellen die auf das Objekt fallen. Dies kostet Performance und kann zum Flackern führen.

d) Und schlussendlich durch das Vorberechnen der Beleuchtung mit Hilfe des Lightmappings - dem „baken“ – Dazu wurde das Lightmapping der Beleuchtungsobjekte auf „BakedOnly“ gestellt und es wird bereits während der Entwicklungszeit die Beleuchtung des Levels berechnet und

nicht mehr während dem Spiel in jedem Frame. Dabei werden während des bakens nur statische Objekte berücksichtigt und beleuchtet. Hier ist das Problem, dass dies unser Konzept der Beleuchtung und dem individuellen Ein- und Ausschalten gestört hat und deshalb darauf verzichtet wurde.

8.1.3 Positive Erfahrungen

- Die Navigation von AI gesteuerten Gegner wird von Unity sehr vereinfacht indem die „begehbare“ Fläche berechnet wird. Innerhalb dieses Bereichs können in wenigen Zeilen Code die kürzesten Pfade berechnet werden und die AI erhält neue Zielpositionen, die sie abarbeitet.
- Viele Tutorials zu einfachen Anwendungen und Scripts mit Unity

8.1.4 Negative Erfahrungen

- Beim Exportieren eines Prefabs werden ebenfalls Scripts zum Export vorgeschlagen, die bloss eine Member-Variable auf veränderte Klassen besitzen. Es ist uns keine Möglichkeit bekannt, dieses Verhalten zu beeinflussen. Man sollte aber nur die direkt dem Prefab zugeteilten Skripte exportieren.
- Nette Eigenschaften wie auf bewegliche Objekte zu reagieren stehen nur Unity Pro - Benutzern zur Verfügung.
- GUI Design:
Ansprechende grafische Elemente wie Tabellen oder Menus innerhalb der onGUI() Methode zu bauen ist sehr umständlich und es fehlt an geeigneten Methoden und Klassen. Als Beispiel: Eine Tabelle anzeigen zu lassen, war bis zum Zeitpunkt des Projektendes nicht möglich. Auch das Positionieren von Buttons und anderen GUI Elementen ist sehr umständlich, da keine Werkzeuge dafür zur Verfügung stehen.

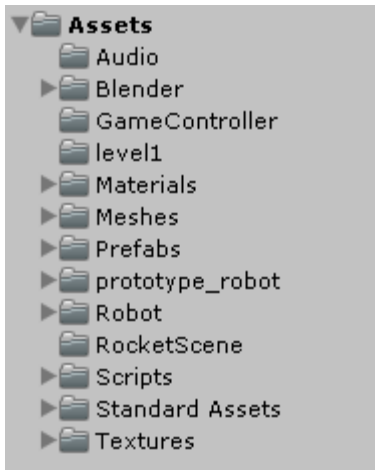
8.1.5 Lessons learnt

- Ein besseres Code Repository mit eleganterer Konfliktbehebung und Versionierung analog einem Subversion würde die Arbeit erleichtern. Dazu müsste man ein Entwicklungsstudio kontaktieren um eine professionelle Empfehlung zu erhalten.
- Ein einheitliches Namenskonzept für das Tagging muss vorgängig deklariert werden.
Bsp. (sehr oberflächlich):
Gegner: enemy_X
Gegenstände: obj_X
Scripts: %nameOfGameObject%_X

Dies erleichtert Einschätzungen bezüglich Auswirkungen auf Komponenten und dient allgemein dem Verständnis.

- Alle Prefabs die erstellt werden, sollten ebenfalls nach einem Namenskonzept benannt und mit einer Versionsnummer versehen werden. Wenn möglich ein separates Dokument erstellen, in dem die einzelnen Versionen mit ihren enthaltenen Änderungen beschrieben sind.

- Die Asset-Hierarchie ebenfalls frühzeitig homogen aufbauen, damit nicht jeder mit einer eigenen Projektstruktur arbeitet:



Unter Robot befinden sich leider auch einige Scripts, welche eigentlich in den Scripts Ordner gehören. Ebenfalls ist der Blender Ordner von der Applikation Blender erstellt worden und auf Grund des heiklen Referenzhandlings (speziell durch den MonoDevelop Client) haben wir darauf verzichtet diese beiden Strukturen anzupassen.

- Das Importieren von GameObjekten will gelernt sein! Blender (3D Modeling Tool) bietet zwar eine Vielzahl an Funktionen und man kann beinahe lebensechte Modelle von Personen, Tieren und Fahrzeugen erstellen und animieren. Doch für unser Projekt hat das Arbeiten mit Blender eher nur Nachteile gehabt. Zwar gibt es eine Import/Export Schnittstelle, aber die erstellten Objekte vom Blender sind dann oft untereinander verlinkt und können nur mit grossem Aufwand oder gar nicht direkt im Unity angepasst werden. Wir würden bei einem nächsten Projekt mit Unity entweder ganz auf den Import von anderen Editoren verzichten und uns ganz auf Unity konzentrieren. Oder wir müssten noch einmal grundlegend das Verständnis aufbauen, um Modelle sauber mit Blender zu erstellen und anschliessend fehlerfrei und ohne Nachteile in Unity importieren zu können. Alles andere bringt aus unserer Sicht beim späteren Projektverlauf nur Mehrarbeit.

8.2 Xamarin MonoDevelop-Unity

8.2.1 Positive Aspekte

Syntaxhervorhebung und Code-Komplettierung für Funktionen und Methoden aus Unity Klassen. Eigentlich ist das der einzige Punkt, weshalb man den MonoDevelop-Editor verwenden sollte.

Die Alternative:

Visual Studio funktionierte in der Version 2010 ganz ordentlich. Die Installation ist aber riesig. Ausserdem steht das Studio nicht für Mac OS zur Verfügung. Auch deshalb haben wir darauf verzichtet um mit einem einheitlichen Editor zu arbeiten.

8.2.2 Negative Aspekte

- Instabilität: Schnelle Klicks auf unterschiedliche Tabs oder das Verschieben von Tabs kann den Editor zum Absturz bringen.
- Kompilierfehler (durch das „Real-Time-Compiling im Unity Editor) können ein ganzes Unity Projekt zerstören. Genauso wie verloren gegangene Referenzen oder fehlende Scripts, die angesprochen werden.
Mehrere Male mussten alle Komponenten eines Unity Projektes exportiert und in ein neues importiert werden, weil beim Start von MonoDevelop der Solution Container automatisch geladen wird und Referenzprobleme meist zu einem Hängen der Applikation führten.
- .NET Runtime musste zwei Mal repariert werden, damit der Editor nach einem Absturz wieder verwendet werden konnte.

8.3 Fazit zum Projektabschluss

Aus den Erfahrungen, die wir bei der Arbeit mit Unity sammeln konnten, erhielten wir einen schönen Einblick in die Welt der Spieleentwicklung. Auch das Nachlesen der Theorie und das Durchsuchen der Manuals und Foren nach Lösungen brachte uns das ein oder andere zusätzliche Thema näher. Weil wir im Rahmen des Moduls BTI7054 die Skriptsprache JavaScript neu kennen lernten, ergänzten sich die beiden Module BTI7301p und BTI7054 sogar ein wenig.

Zu Beginn kämpften wir etwas mit Startschwierigkeiten, was aber in der Natur eines neuen Themas liegt. Dank den vorhandenen Tutorials konnten wir uns aber dann doch zügig in die Materie einarbeiten. Dadurch, dass wir mehrheitlich freie Hand bei der Planung und der Umsetzung erhielten, war auch das Erarbeiten eines realistischen Zeitplans eine Herausforderung. Zumal wir anfangs aufgrund mangelnder Fachkenntnisse nicht die Möglichkeit hatten, abschätzen zu können, wie viel Zeit für welche Komponenten benötigt würde.

Abschliessend können wir behaupten, die Projektanforderungen und die zu Beginn deklarierten Ziele erfüllt zu haben. Nebst dem Endprodukt „Rocket“, einem Adventure Game - entwickelt mit der Unity-Engine - erarbeiteten wir zudem noch folgende Dinge:

- ✓ Konzipierung einer Game Story als Grundlage für die weitere Entwicklung
- ✓ Kennenlernen der Unity Spieleentwicklungs-Umgebung und das Aneignen der notwendigen Kenntnisse in JavaScript und C#
- ✓ Computergesteuerte Gegner mit Eigenintelligenz, die den Spieler am Spielerfolg hindern
- ✓ Evaluation der einzusetzenden Technik für einzelne Problemstellungen. Daraus resultierte weiter:
- ✓ Die Implementation klassischer Techniken der Spiele-Entwicklung, besonders erwähnenswert sind hier Zustandsautomaten und ein zentrales Messaging System

Wir möchten uns bei Herrn Eckerle für sein regelmässiges Feedback, seine Inputs und das Vertrauen, welches er uns entgegenbrachte, danken. Mit der erhaltenen Freiheit wussten wir verantwortungsbewusst umzugehen und brachten das Projekt zu einem erfreulichen Abschluss. Dass wir zum Ende des Semesters ein spielbares Produkt abliefern können und uns somit ebenfalls etwas als Erinnerung bleibt, freut uns natürlich sehr.

8.4 Offenes und Ausblick

Die bestehende Version des Spiels wird von uns nicht mehr weiterentwickelt. Einzelne Codeteile können aber durchaus für spätere Projekte wiederverwendet werden. Beispielsweise die HFSM in C# könnte auch anderen Entwicklern dienen.

Was noch offen ist und wir aus Zeitgründen nicht mehr korrigieren konnten:

IST	SOLL
Inventaritems werden beim Verlassen nicht korrekt angezeigt.	Beim Verlassen des Levels sollten die gefundenen Items gezählt und aufgelistet werden.
Beim Einfangen des Spielers und dem anschliessenden Restart des Spiels werden einige Game Objekte nicht richtig zurückgesetzt. Bsp. der Countdown und oder die Pfade der Roboter. Ein Neustart der Szene (wenn man über den Play-Button aus dem Unity Editor heraus spielt) fällt dies nicht auf. Auch beim erfolgreichen Abschluss des Levels verhält sich der Restart des Levels nicht 100% korrekt.	Für eine finale Build-Version müsste dies korrigiert werden, so dass ein Restart des Levels das Spiel sauber neu initialisiert.
Die Konsole enthält noch diverse Ausgaben. Dies, weil bis zuletzt am Level gearbeitet wurde.	In einer Build-Version wäre die Konsole sowieso nicht zu sehen. Um aber einen ganz sauberen Zwischenstand zu erreichen, müsste das Log wohl auch gesäubert werden.

9 Glossar

Abkürzung / Fremdwort	Bedeutung / Umschreibung
HMS / HFSM	Hierarchical State Maschine, ein Design Pattern, das oft bei der Softwareentwicklung zum Einsatz kommt.
State	Ist ein Zustand, der vom Roboter eingenommen werden kann. Dabei verändert sich das Verhalten und die Reaktionen des Roboters auf die Spielumgebung
PowerSupply	Die Aufladestation des Roboters, an der der Roboter seine Batterie wieder aufladen kann.
Unity-Framerate	Gibt die Frames pro Sekunde an in Unity, siehe auch FPS
FPS	Frames per Second, Hier auch als Unity-FPS bezeichnet. Wie oft pro Sekunde das Bild und der Spielzustand und alle Events berechnet werden.
Prefab	Im Unity kann ein Gegenstand und all seine Eigenschaften quasi als „Bauplan“ gespeichert werden. Das wird im Unity als Prefab bezeichnet. Dieser Gegenstand kann dann beliebig oft in die Spielumgebung kopiert werden. In dem man das Prefab ändert, kann man auch alle daraus erstellten Gegenstände
Scene (im Unity)	Unity lädt immer eine Szene, in der das Spiel sich befindet. Unity unterstützt das Laden von einer Szene zur anderen. Das wird zum Beispiel gebraucht, um vom Game Menü zum eigentlichen Spiel zu wechseln oder von einem Level 1 zu Level 2 zu wechseln beim erfolgreichen Abschluss.

Project (im Unity)	Ein Unity Projekt besteht aus einer oder mehreren Scene und aus einem Projekt kann dann eine .EXE Datei erstellt werden, um das Unity-Projekt zu spielen. Unity unterstützt eine Erstellung auf Mac OS, Windows oder portable Plattformen an.
--------------------	---

10 Referenzen

Webseiten

Community, Unity (2014). *Unity Community* <http://docs.unity3d.com/Manual/index.html>
Community, Unity (2014). *Unity API* <http://docs.unity3d.com/ScriptReference/index.html>

Bücher

Millington, I. (2006). *Artificial Intelligence for Games*. Elsevier.