Abschlussarbeit Sommer 2016

Fachinformatiker für Anwendungsentwicklung

Dokumentation zur betrieblichen Projektarbeit

Charaktersteuerung

Programmierung einer intuitiven Charaktersteuerung im Projekt „Sisyfox“

Abgabetermin: 26.04.2016

Prüfungsbewerber:
Marvin Jung
Weddinger Str. 23
38690 Immenrode

Ausbildungsbetrieb:
ckc ag
Am Alten Bahnhof 13
38122 Braunschweig

Inhaltsverzeichnis

Seite

Abbildungsverzeichnis

1. Einleitung

Die ckc group ist ein von Christian Krentel im Jahr 1989 gegründeter IT- und Business-Consulting-Anbieter in Braunschweig. Sie gehört in dem Bereich zu den führenden Unternehmen Deutschlands. In den Standorten Braunschweig, Berlin, Darmstadt, Hamburg, Dortmund und München beschäftigt sie rund 400 Mitarbeiter[1].

Hauptbranchen des Unternehmens sind die Automobilindustrie samt Zulieferer sowie Banken, Versicherungen, Luft- und Raumfahrt, Retail, Transport und Logistik. Die Kernkompetenz von ckc liegt im Bereich der IT nicht nur auf der Softwareentwicklung, sondern zu Teilen zusätzlich auf der Managementberatung.

1.1. Projektbeschreibung

Das "Sisyfox" Projekt wird entworfen, um für firmeninterne Events (z.B. Seminare, Konferenzen, Sommerfeste, etc.) eine Attraktion zu bieten, die nicht nur mit Spaß verbunden wird, sondern aus der sich auch ein Lernwert ergibt. Das Spiel stellt in einer vereinfachten Form den Mythos des Sisyphos und sein immerwährendes Scheitern gamifiziert[2] dar. Den Mitarbeitern soll dabei der Umgang mit dem Erlebnis des Scheiterns näher gebracht werden. Außerdem soll diese Gamifizierung zu einer Motivationssteigerung führen.

Der Charakter im Spiel wird mit einem Aufbau, auf dem ein 1.20m großer Ball als Steuerkugel[3] befestigt ist, vom Spieler gelenkt. Dafür steht der Aufbau vor dem Spieler auf dem Boden. Die Kugel kann durch den Aufbau mit den Händen auf der Stelle bewegt werden. Die Sensordaten des Aufbaues werden direkt an den Computer übermittelt und müssen in der Steuerung umgesetzt werden.

Die Aufgabe in dem Projekt besteht darin, die Steuerung für den Charakter im Spiel zu programmieren. Diese wird in der Programmiersprache C# geschrieben, damit sie in der Spieleentwicklungsumgebung "Unity" als Skript in das vorhandene Projekt eingebunden werden kann. Hierbei muss auf eine intuitive Steuerung geachtet werden, da die Attraktion einer großen Zielgruppe zur Verfügung gestellt werden soll. Die Intuitivität[4] soll durch eine leicht erlernbare Steuerung sowie eine deutliches visuelles Feedback realisiert werden. Im Projekt vorhanden sind bereits sämtliche Modelle, Animationen sowie Sounds und Hintergrundmusik, aber auch schon einzelne, von mir früher programmierte Skripte, die sich z.B. um das Benutzerinterface kümmern. Die benötigten Daten für die Steuerung des Charakters werden von einer Trackballmaus zum Computer übertragen und müssen dort in die Bewegung umgesetzt werden. Außerdem muss im Rahmen der Projektarbeit bestimmt werden, wie sich der Charakter in der virtuellen Welt

[1] *Unternehmensporträt der ckc.* [2016] [1]
[2] *Gamifizierung Definition.* [2016] [2]
[3] Wie in einem Kugellager.
[4] Je leichter es ist sich die Bedienung der Steuerung anzueignen, desto höher ist die Intuitivität.

verhält, in welchen Situationen der Spieler das Spiel vorzeitig verlieren kann und ab welchem Zeitpunkt das Spiel gewonnen ist.

1.2. Projektziel

Mit dem Projekt soll eine gewisse Hartnäckigkeit, gefolgt von gesteigerter Motivation durch Erfolgserlebnisse bei den Mitarbeitern, sowie Besuchern von Firmenfesten und anderen Events erzielt werden. Der Benutzer wird mit jeder gespielten Runde vor die Aufgabe gestellt, eine scheinbar unerreichbare Hürde zu überwinden. Er wird viele Versuche brauchen, um den Gipfel zu erreichen. Der spielerische Aspekt sorgt unweigerlich dafür, dass der Benutzer leichter mit häufigen Niederlagen umgehen kann, um anschließend den anfangs weit entfernten Berg erneut herauszufordern. Sobald der Spieler am Gipfel angekommen ist, gibt es auditive und visuelle Erfolgsmeldungen. Dazu erscheint eine Anzeige, wie lange der Weg zur Bergspitze gedauert hat, damit der Spieler sich mit anderen messen kann und ein Wettkampfgefühl entsteht.

1.3. Projektabgrenzung

Im Projekt sind bereits ein Spielehauptcontroller sowie eine Animations- und Bewegungssteuerung vorhanden. Der Hauptcontroller sorgt für die Verwaltung der Grundstatus[5] im Spiel. Außerdem speichert er die wichtigsten Variablen, wie unter anderem Spieldauer oder die Höhe abweichend vom Fuß des Berges. Es wird lediglich ein Verweis via Assoziation in der Charaktersteuerung eingebunden, damit z.B. die Höhendaten kommuniziert werden können.

[5] Die vier Status sind in aktiv, spielend, gewonnen und verloren eingeteilt.

2. Projektplanung

2.1. Projektphasen

Für die Umsetzung des Teils des Projekts stehen insgesamt 70 Stunden zur Verfügung. Die Entwicklung wurde vor Beginn des Projekts auf die verschiedene Projektphasen aufgeteilt. In der folgenden Tabelle [1] lässt sich die grobe Zeitplanung ablesen.

Projektphase	**Geplante Zeit**
Ist-Analyse	6 Std
Soll-Konzept	11 Std
Implementierung	24 Std
Integration in das Unityprojekt	8 Std
Testen und Fehlerbehebung	11 Std
Dokumentation	10 Std
Gesamt	**70 Std**

Tabelle 1.: Grobe Zeitplanung

2.2. Ressourcenplanung

Es werden alle verwendeten Ressourcen wie Hard- und Software in einer Übersicht im Anhang in Abbildung A.1 auf Seite i aufgeführt. Um möglichen Aufwand für Software so gering wie möglich zu halten, wurde darauf geachtet, kostenfreie sowie Open Source Software zu benutzen.

3. Analysephase

3.1. Ist-Analyse

In dem Spiel gibt es viele kleine Details, welche über das Einbinden von Scripten und das Platzieren von 3D-Modellen realisiert werden. Dazu gehört zunächst die Spielkarte, auf der sich der Charakter bewegt. Diese wurde von anderen Teammitgliedern erstellt, mit 3D-Modellen wie z.B. Bäumen, Sträuchern oder Gesteinsbrocken bestückt und in das Projekt implementiert. Zusätzlich wurden 3D-Modelle von Wolken und Vögeln erstellt, die sich im Kreis um den Hauptberg bewegen[1]. Die Spielfigur namens „Sisyfox“ gehört zu den schon erstellten 3D-Modellen und wurde ebenfalls schon mit einer vollständigen Animation versehen. Diese Animation hat jedoch nichts mit der eigentlichen Bewegung des Charakters zu tun. Sie zeigt lediglich visuell eine Laufbewegung an, die aber nicht für eine Positionsverschiebung sorgt. Wie in Abschnitt 1.3 Projektabgrenzung bereits erwähnt, ist der Hauptcontroller, als eines von vielen Scripten, schon vorhanden. Weitere wichtige Scripte sind die Kamerasteuerung, sämtliche Animationsscripte, eine Höhenmeteranzeige und ein Arduinokommunikationsscript, welches eine Verbindung zu einer Hardwarekonstruktion namens „Arduino[2]“ aufbaut. Letztes sorgt dafür, dass sich relativ zur Höhe auf der sich der Charakter befindet, ein angeschlossener Ventilator stärker dreht und eine Nebelmaschine kurz vor Erreichen des Gipfels anspringt.

3.2. Anforderungsanalyse

Um die Positionsverschiebung des Spielecharakters zu implementieren, muss eine Steuerung geschrieben werden. Diese soll von dem Nutzer kontrolliert werden. Das bedeutet, dass der Nutzer Eingaben tätigt und die Steuerung dafür sorgt, dass es ein visuelles Feedback gibt. Hierbei muss innerhalb der Steuerung auf vorher festgelegte Niederlage- und Gewinnbedingungen reagiert und zusätzlich eine möglichst realistische und damit intuitive Bewegung ermöglicht werden. In der Entwurfsphase wird das genauere Vorgehen einschließlich einer Programmarchitekturauswahl hinsichtlich der gestellten Anforderungen erläutert.

3.3. Qualitätsanforderungen

Das Spiel und der dazu geschriebene Code muss folgenden Qualitätsanforderungen entsprechen.
Im Code muss ein sauberer Schreibstil gepflegt werden, um bei Wartungen oder Veränderungen den Aufwand gering zu halten. Dafür werden für jede Methode Kommentare verfasst, die die grundlegende Funktion beschreiben. Des Weiteren werden die einzelnen Funktionsweisen in Methoden von einander getrennt. Variablenamen müssen selbstsprechend benannt werden und bei Abweichung für Hilfsvariablen, darüber ein Kommentar mit einer kurzen Beschreibung geschrie-

[1] Die Bewegung wird von einem Script gesteuert.
[2] *Arduino Beschreibung.* [2016] [3]

ben werden. Nach dem Fertigstellen einzelner Methoden wird ein manueller Test der implementierten Funktion durchgeführt. Das Spiel wird gestartet und die Funktion ausprobiert.

4. Entwurfsphase

4.1. Zielplattform

Zur Auswahl standen die IDEs[1] „Unity 3D Engine“ und „Unreal Engine 4“. Gewählt wurde die Unity Engine, da in dieser gegenüber der Unreal Engine deutliche Vorkenntnisse vorhanden sind, die dann im Projekt für einen flüssigen Workflow sorgen. Ein weiterer Grund ist das sehr einfache und intuitive Einbinden von Gameobjekten[2] und deren angebundenen Scripten.

4.2. Architekturdesign

Für den Architekturaufbau standen das „Model-View-Controller-“ (MVC) und das „Presentation-Abstraction-Control-Muster“ (PAC) zur Auswahl.

Das MVC Muster (Siehe Schema Abbildung A.2 auf Seite i) besteht grundsätzlich aus drei Strukturen, die unterschiedliche Ebenen im Programmaufbau darstellen. Diese unterteilen sich in „Modell-“, „View-“ und „Controller-Ebene“. Die Unterteilung dient hierbei dem Zweck, dass jede Ebene flexibel und unabhängig von den Anderen verändert, ersetzt oder erweitert werden kann. Das Modell beinhaltet die Daten, die für die Anzeige nötig sind. Über die View-Ebene werden die Daten aus dem Modell dem Benutzer angezeigt und des Weiteren mögliche Benutzereingaben entgegengenommen. Wenn sich Daten im Modell ändern wird im Regelfall die View-Ebene mithilfe eines Beobachters[3] benachrichtigt, sodass sich die View aktualisieren kann. Die entgegengenommen Benutzereingaben werden von dem Controller übernommen und verarbeitet. In dem objektorientierten Ansatz führt der Controller Methoden im Modell aus, damit die Aktionen des Benutzers wirksam werden. Außerdem kann der Controller die View direkt manipulieren, um z.B. die angezeigten Eingabemöglichkeiten zu verändern.

Für den alternativen Ansatz, in dem man ein System in verschiedenen Einzelteilen erstellt, benötigt man ein Muster, das jede Aufgabe in einem anderen Teil des System, abbildet. Dadurch wird hohe Flexibilität ermöglicht, die sich auch im Wartungsaufwand widerspiegelt. Im PAC Muster (Siehe Schema Abbildung A.3 auf Seite ii) teilt man das System in zwei Richtungen auf, zunächst in drei Einheiten, die grafische Oberfläche, eine Kommunikationsschnittstelle und ein Datenmodell. Diese Einheiten ähneln dem MVC-Muster.

Darüber hinaus teilt sich das Architekturmuster hierarchisch auf sogenannte „Agenten“ auf. Diese Agenten werden auf drei Schichten verteilt, auf *Top-Level-*, *Intermediate-Level-* und *Bottom-Level-*Agenten. Den *Top-Level-*Agenten gibt es nur einmal im System, er übernimmt die globalen

[1] eng: integrated development environment, de: Integrierte Entwicklungsumgebung
[2] „Gameobjekte“ sind die Grundbausteine von jedem Unity Spiel und können verschiedene Komponenten aufnehmen, um damit den unterschiedlichen Aufgaben eines Spiels gerecht zu werden.
[3] Der Beobachter meldet sich bei der View-Ebene, sobald sich Daten im Modell ändern.

Aufgaben. Für die *Bottom-Level*-Agenten ist eine möglichst in sich abgeschlossene Aufgabe vorgesehen. Die *Bottom-Level*-Agenten können über implementierte Methoden in den *Intermediate-Level*-Agenten mit dem *Top-Level* kommuniziert.

In dem Projekt wird sich für eine einfache Variante des PAC-Musters entschieden. Der schon vorhandenen Hauptcontroller wird als Top-Level-Agent eingestuft und die Charaktersteuerung als eine Mischung aus Intermediate- und Bottom-Level-Agent. Die Steuerung wird über eine einfache Assoziation direkt mit dem Hauptcontroller kommunizieren können. Im nächsten Abschnitt wird der Entwurf eines Klassendiagrammes erläutert, in dem diese Herangehensweise umgesetzt wird.

4.3. Entwurf der Benutzeroberfläche

An der Benutzeroberfläche wird in dem Projekt nichts verändert. Es werden jedoch einige Daten von der Charakter Steuerung an den Hauptcontroller übermittelt, welche auf dem Bildschirm angezeigt werden. Im Anhang befindet sich eine Skizze (Siehe Anhang Abbildung A.4 auf Seite iii) sowie ein Anzeigebild aus dem laufendem Spiel (Siehe Anhang Abbildung A.5 auf Seite iii), in dem die wichtigsten Anzeigen eingetragen sind. Auf der Skizze befindet sich Punkt „A“ oben links auf dem Bildschirm lokalisiert. An der Position wird eine weiße Bergsilhouette gezeigt, die sich relativ zur Charakterhöhe am Berg füllt. Die Charakterhöhe wird von der Charaktersteuerung ermittelt und an den Hauptcontroller für die Anzeige der Höhe und der Bergfüllanzeige übergeben. Gleich rechts daneben - Punkt „B“ - zeigt die vergangene Zeit für den bisherigen Versuch an. In der Mitte des Bildschirms ist eine Höhenanzeige in Metern eingebaut (siehe Punkt „C“). Diese erscheint alle 150 Höhenmeter und gibt den derzeitigen Zwischenstand an. Anschließend verschwindet sie wieder. Als letzten Punkt auf der Skizze ist die Charakterposition eingezeichnet.

In der Abbildung A.5 auf Seite iii sind alle auftretenden Benutzerinterface Anzeigen im Spiel dargestellt.

4.4. Geschäftslogik

Zunächst werden die Anforderungen an die Charaktersteuerung festgelegt, damit daraus ein Klassendiagramm entstehen kann. Das Klassendiagramm kann im Anhang in Abbildung A.6 auf Seite iv eingesehen werden. Die Steuerung muss als Hauptaufgabe die Mausinformationen sowie deren Bewegungsrichtung ermitteln und in eine Bewegung der virtuellen Kugel übersetzen. Um das zu realisieren, muss die aktuelle Mausbewegungsrichtung gespeichert werden. Daraus wird ein Richtungsvektor ermittelt. Der Vektor muss anschließend mit der einer spezifischen Geschwindigkeit multipliziert werden, damit die Beschleunigung entsteht. Nun ist ein Problem entstanden. Die Richtung, in die die Maus bewegt wurde, kann nicht direkt auf die Bewegungs-

richtung der Kugel addiert werden, da die Kugelbewegung auf dem 3D Weltkoordinatensystem[4] liegt und die Mausbewegung auf dem Bildschirmkoordinatensystem.

Ein Beispiel (Siehe auch Abbildung 4.1: Das Eingabemedium wird vom Benutzer nach links vorne geschoben, daraus entsteht ebenfalls eine Mausbewegung nach links vorne. Das ist der Vektor „a“ in Abbildung 4.1 bei Punkt [1]. Vektor „b“ bezeichnet die aktuelle Kugelbewegung innerhalb des Weltkoordinatensystems. Für den Benutzer rollt die Kugel nach vorne, da die Kamera immer hinter dem Betrachter in Bewegungsrichtung schauend steht. Im Weltsystem ist die Rollrichtung jedoch ein Vektor, der von der Weltrichtung „vorne“ abweicht. Das heißt, wenn jetzt wie in Punkt [2] die beiden Vektoren ohne Vorbehandlung addiert werden, die Kugel nicht wie gewünscht nach links, sondern nach rechts rollt. Um die unbeabsichtigte Richtung zu korrigieren, muss der Mausbewegungsvektor „a“ um den Winkel α aus Punkt [1], der die Abweichung des Welt „vorne“ Vektors von der Kugelbewegungsrichtung darstellt, gedreht werden. Anschließend kann der „neue a“ Vektor wie in Punkt [3] auf den Kugelvektor „b“ addiert werden. Daraus resultiert jetzt die erwartete Bewegungsrichtung.

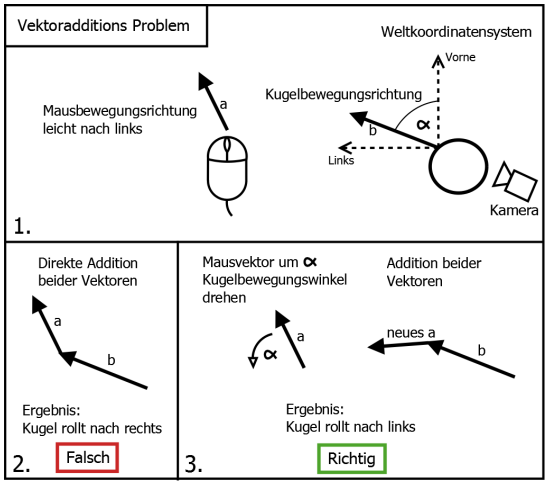


Abbildung 4.1.: Koordinatensystem Problem

Ergänzend sollen vorher festgelegte Mechanismen überprüfen, in welchen Situationen der Spieler das Spiel gewinnt oder verliert. Dieser Zustand oder auch Status des Spiels wird vom Hauptcontroller verwaltet (Siehe Statemachine Anhang Abbildung A.7 auf Seite v). Der Status geht vom *inaktiv*[5] Status über in den *laufend* Status, welcher je nach Spielweise in *gewinnen* oder *verlieren* endet. Gewinnen wird der Spieler, wenn er am Gipfel des Berges angelangt ist und dort in eine Triggerbox[6] (Siehe auch Abbildung A.8 auf Seite v) hineinläuft und somit den *gewonnen* Status erreicht. Der Niederlage-Mechanismus funktioniert ähnlich. Hierfür wurde festgelegt, dass bei Berührung eines Baumes (Siehe auch Abbildung A.9 auf Seite vi) oder Felsbrockens das Spiel verloren sein soll. Dafür werden diese ebenfalls mit einem Trigger versehen, der den *verloren* Status auslöst.

Zusätzlich soll der Kugel eine möglichst realistische Physik gegeben werden. Das heißt, dass der Charakter nicht in der Lage sein soll, die Kugel seitlich an einem Abhang entlang zu rollen, ohne dass sie ihm zur Seite weg rollt und er die Runde verliert. Diese Mechanik ist in Abbil-

[4] Das Koordinatensystem ist statisch. Die Kugelbewegung ist, obwohl sie visuell nach vorne rollt, möglicherweise abweichend davon.
[5] Im inaktiv Status verweilt das Spiel solange, wie es keine Benutzerinteraktion gibt.
[6] Ein Trigger ist in dem Fall eine unsichtbare Box mit Kollisionserkennung an den Außenseiten, die ein Kollisionsevent erzeugt. Das Event löst eine Funktion beim kollidierenden Objekt aus.

dung 4.2 visualisert. Der Benutzer kann als Reaktion versuchen den Charakter unter die Kugel zu manövrieren, sodass die Steigung vor ihm liegt und nicht seitlich.

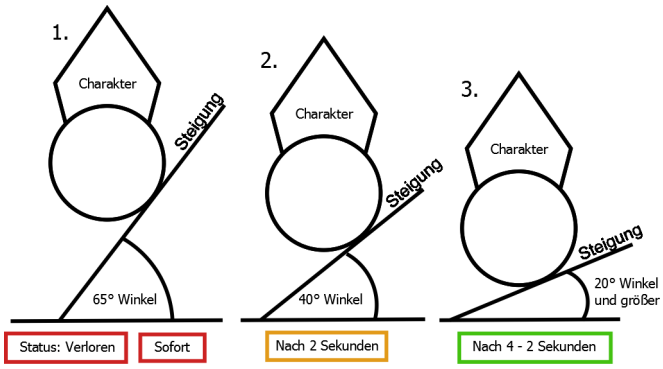


Abbildung 4.2.: Bedingungen bei seitlichem Neigungswinkel

Dann kann der Charakter die Kugel festhalten. Bei einem seitlichen Neigungswinkel von über 65°, wie bei Punkt [1], verliert der Spieler sofort das Spiel. Liegt der Winkel, wie in Punkt [2], zwischen 40°und 65°, dann hat der Spieler 2 Sekunden, um seine Spielfigur zu richten. Auf einem Hang, wie in Punkt [3], mit einem Neigungswinkel zwischen 20°und 40°hat der Spieler abhängig vom Winkel vier bis zwei Sekunden Zeit die Spielfigur zu richten. Wie angesprochen kann die Spielfigur die Kugel an beliebig steilen Steigungen festhalten, solange sie sich unter[7] der Kugel befindet. Befindet sich der Charakter auf der anderen Seite („über“ der Kugel), dann ist die Kugel ebenfalls bei einem Winkel von 65°verloren.

Als weitere Funktion der Steuerung soll der Charakter, wenn er durch einen Busch läuft, verlangsamt werden. Hierzu muss für jeden Busch ein weiterer Trigger (Siehe auch Abbildung A.10 auf Seite vi) hinzugefügt werden.

4.5. Maßnahmen zu Qualitätssicherung

Um die Qualität der umzusetzenden Steuerung zu gewährleisten, wird während der Implementierung nach jeder Einbindung einer neuen Funktion, ein manueller Test durchgeführt. Die neue spezifische Funktion wird anhand einer augenscheinlichen Ausführung und Debug-Informationen[8] begutachtet. Daraus wird geschlussfolgert, ob die Implementierung erfolgreich eingesetzt wurde.

[7] „Unter“ bedeutet, dass der Spieler gerade auf die Steigung zuläuft.
[8] Die Debug-Informationen werden „hard-coded“. Das bedeutet, dass die Informationen zum erfolgreichen Erreichen des Codeabschnittes direkt in den Code geschrieben werden und dort nach erfolgreichem Test gelöscht werden.

5. Implementierungsphase

5.1. Implementierung der Klassenstruktur

Am Anfang werden die beiden Klassen „SphereController“ und „stSphere“ erstellt. Die Steuerung des Kugel wird hier „SphereController“ genannt, da es bereits einen „CharacterController“ gibt. Dieser ermittelt den aktuellen Stand der Animation der Spielfigur und deren Position hinter der Kugel. Die Bezeichnung ist daher passend, weil sich die Steuerung im Endeffekt um die Bewegung der Kugel oder auch „Sphere“ kümmert.

In die Klasse „stSphere“ werden alle spezifischen Daten der Sphere abgelegt. Im Anhang ist die vollständige Klasse in ?? auf Seite viii abgebildet. Für sie wurde die statische Variante gewählt, da es im Spiel nur eine einzige Kugel gibt und sich deren Eigenschaften nicht verändern. So muss im „SphereController“ kein Objekt der Kugel erstellt werden. In der Steuerung kann einfach via „stSphere.Variablename“ auf die Attribute der Kugel zugegriffen werden. Alle Variablen der „stSphere“ Klasse sind nur mit *gettern*[1] versehen, damit es nicht die Möglichkeit gibt durch einen anderen Script die Werte zu verändern[2]. Die eingesetzten Variablen sind direkt initialisiert. Die Werte ergeben sich aus mehreren manuellen Tests und anschließender Optimierung an die subjektiv beste Spielerfahrung. In der Klasse finden sich, die in Abschnitt 4.4 Geschäftslogik festgelegten Winkelbedingungen unter „InstantLossAngle“, „MiddleLossAngle“ und „LowLossAngle“ wieder.

Für die „SphereController“ Klasse werden zunächst die Verweise und Abhängigkeiten zu den anderen Scripten geschaffen. Der „GameController“ Script übergibt sich selbst bei der Instanzierung[3] des Spielerobjektes an den Sphere Script. In der Start Methode des „SphereController“ wird der Verweis zu dem *Transform*[4] sowie zu dem *Rigidbody*[5] der Kugel erstellt. Im folgenden ist ein Ausschnitt der Startfunktion zu sehen.

Codeausschnitt 5.1: Startfunktion

```
1  //Transform und Rigidbody Verweise zusammensuchen
2  traSphere = this.gameObject.GetComponent<Transform> ();
3  rigSphere = this.gameObject.GetComponent<Rigidbody> ();
```

[1] Ein *getter* ermöglicht eine Anpassung der Rückgabefunktion der zugehörigen Variable.
[2] Der Wert könnte mithilfe eines *setters* von außerhalb verändert werden. Der wurde hier weggelassen.
[3] Instanzierung heißt bei Unity, dass ein Objekt als Kopie eines *Prefabs* in der 3D-Welt erstellt wird. Ein *Prefab* ist ein zuvor erstelltem Spielobjekt mit angebundenen Komponenten, in dem Fall die Spielefigur mit Kameraobjekt und Kugelobjekt. An das Kugelobjekt ist der „SphereController“ angebunden. Die Unity Ansicht kann in Abbildung A.11 auf Seite vii gesehen werden.
[4] Das Transform ist eine Komponente eines Gameobjektes und enthält dazu die Positions-, Rotations- und Skalierungsdaten.
[5] Das Rigidbody ist eine Komponente eines Gameobjektes und sorgt für Physikbeachtung des Objektes. Dazu gehört z.B. das Reagieren auf die Schwerkraft oder Kollidieren mit anderen Gameobjekten. Zum Kollidieren wird zusätzlich eine „Collider“-Komponente benötigt.

5.2. Implementierung der Geschäftslogik

Im Anschluss an die Implementierung der Klassenlogik wird sich der Geschäftslogik gewidmet. Der „SphereController“ Script hat eine Update Funktion, die von der Engine[6] vor jedem Frame[7] aufgerufen wird. Dort wird sich je nach Hauptsteuerungsstatus entschieden, welche Funktionen ausgeführt werden. Außerdem wird in jedem Update Aufruf die Mausbewegungsrichtung[8] zwischengespeichert. Die Methode „GetAxis“ gibt einen Wert zwischen -1 und 1 für die angegebene Achse zurück, der, wenn *mouseX* und *mouseY* zu einem Vektor zusammengefasst werden, den Richtungsvektor symbolisiert.

Codeausschnitt 5.2: Updatefunktion

```
1  void Update () {
2     //Je nach Spielstatus wird hier entschieden was gemacht wird
3     switch ( GameControllerScript.GameState ) {
4        case 0:
5           //Status inaktiv ausfuehren
6           State0_Idle ();
7           break;
8        case 1:
9           //Status laufend ausfuehren
10          State1_Running ();
11          break;
12       case 2:
13          break;
14       case 3:
15          break;
16       default:
17          break;
18    }
19
20    //Mausposition X und Y jeden Frame neu speichern
21    mouseX = Input.GetAxis ( "Mouse X" );
22    mouseY = Input.GetAxis ( "Mouse Y" );
23 }
```

Die in der Updatefunktion aufgerufene *State0_Idle* Methode beinhaltet das Verhalten der Kugel im inaktiven Zustand. Dort wird abgefragt, ob die *mouseX* oder *mouseY* Variablen einen Wert anders als 0 hat, um eine Bewegung des Eingabemediums festzustellen. Sobald die Kugel vom Benutzer bewegt wird, gibt der „SphereController“ dem „GameController“ Bescheid, dass das Spiel nun in den nächsten Zustand (Siehe auch Abbildung A.7 auf Seite v) übergehen kann.

Codeausschnitt 5.3: „Idle“ Funktion

```
1  void State0_Idle () {
2     //Wenn die Maus bewegt wird, startet das Spiel
3     if ( (mouseX != 0 || mouseY != 0) ) {
4        //Dem Hauptcontroller Bescheid sagen, dass das Spiel startet
5        GameControllerScript.GameStarted ();
6
7        //...
8     }
9  }
```

Im Anschluss geht es an die Umsetzung der in Abschnitt 4.4 Geschäftslogik genannten Problemlösung der Vektorberechnung. Es wird die *State1_Running* Funktion erstellt. Am Anfang

[6] Die Engine arbeitet im Hintergrund des Spiels und ruft unter anderem die Start und Update Funktionen in Scripten auf. Außerdem kümmert sie sich auch um die grafische Darstellung vom
[7] Ein Frame ist ein angezeigtes Bild während das Spiel läuft. Das Spiel läuft mit ca. 60 Frames pro Sekunde.
[8] Die Mausbewegung wird durch das physische Eingabemedium erzeugt.

steht die Berechnung des Winkels für die Drehung des Mausvektors. Dieser ist in der Variable *angle* abgelegt. *Vector.Angle* ermittelt den Winkel zwischen *Vector.forward* und der aktuellen Bewegungsrichtung *new Vector(velocity.x, 0, velocity.z)*.

Codeausschnitt 5.4: „Running“ Funktion Teil 1

```
1  void State1_Running () {
2      //...
3      //Richtung berechnen, in die Geschoben wird
4      //-sign- bezeichnet 1 oder -1 je nachdem die Rechnung über oder
           gleich 0, oder unter 0 ist
5      float sign = Mathf.Sign ( Vector3.Dot ( new Vector3 ( velocity.x, 0,
           velocity.z ), Vector3.right ) );
6      //-angle- beinhaltet den Winkel der Beschleunigungsrichtung
           abweichend von dem Welt "vorne"
7      float angle = Vector3.Angle ( Vector3.forward, new Vector3 (
           velocity.x, 0, velocity.z ) ) * sign;
8      //...
```

Danach kann aus *mouseX* und *mouseY* der Mausvektor bestimmt werden. Solange die Geschwindigkeit der Kugel kleiner dem, in der Klasse „stSphere“ festgelegten Wert ist, wird dem Vektor die *mouseY* oder auch Schubkraftvariable hinzugefügt. Ansonsten kann im Endeffekt nur gelenkt und nicht geschoben werden, da die *mouseX* nur für die Richtungsänderung zuständig ist. Die Variable *angle* wird mit Hilfe der vorgegebenen *Quaternion.AngleAxis* Methode auf die Mausbewegung multipliziert damit diese korrekt gedreht ist.

Codeausschnitt 5.5: „Running“ Funktion Teil 2

```
1      //...
2      //Solange die maximale Geschwindigkeit nicht erreicht ist ->
3      if ( rigSphere.velocity.magnitude < stSphere.MaxSpeed *
           speedMultiplier ) {
4          //...
5          pushDirectionWithoutAngle = new Vector3 ( mouseX, 0, mouseY );
6      } else {
7          pushDirectionWithoutAngle = new Vector3 ( mouseX, 0, 0 );
8      }
9      //Finale Schubrichtung berechnen - Den Schubvektor um den -angle-
           drehen
10     pushDirection = Quaternion.AngleAxis ( angle, Vector3.up ) *
           pushDirectionWithoutAngle;
11     //Finalen Schub auf die Kugel anwenden
12     applySphereMovement ( pushDirection );
13
14     //Überprüfe Niederlagebedingungen bei Steigungswinkel
15     SphereGradientCheck ();
16 }
```

Im nächsten Abschnitt wird die Neigungswinkel-Überprüfung betrachtet und umgesetzt. Anfangs werden die Winkel die sich im Moment unterhalb der Kugel befinden bestimmt. Dafür werden Raycasts[9] benutzt, die ein Gameobjekt zurückgeben. Das Objekt beinhaltet eine Transformkomponente, die ihre genaue Position kennt. Und anschließend kann man mit jeweils zwei Raycasts, einerseits vor und hinter und andererseits rechts und links der Kugel, die Winkel zwischen den Treffern bestimmen. Im folgenden Codeausschnitt ist beispielhaft die Ermittlung des Winkels zwischen dem vorderen und hinterem Raycasttreffer dargestellt.

Codeausschnitt 5.6: Neigungswinkel Überprüfung Teil 1

[9] „Rays“ sind Strahlen, die von der Engine in bestimmte Richtungen ausgesandt werden können und das Objekt, auf das sie treffen, zurückgeben.

```
1   void SphereGradientCheck () {
2
3       RaycastHit hit;
4       //-Vor- der Kugel einen Raycast auf den Boden casten
5       if ( Physics.Raycast ( traSphere.transform.position +
            rigSphere.velocity.normalized * 0.075f, Vector3.down, out hit ) ) {
6           Vector3 frontHit = hit.point;
7
8           //-Hinter- der Kugel eine Raycast auf den Boden casten
9           if ( Physics.Raycast ( traSphere.transform.position -
                rigSphere.velocity.normalized * 0.075f, Vector3.down, out hit )
                ) {
10              Vector3 backHit = hit.point;
11
12              //Steigung oder Gefälle zwischen beiden Raycast Hits berechnen
13              //sin(α) = h/b
14              winkelSteigungFB = Mathf.Asin ( (frontHit.y - backHit.y) /
                    (frontHit - backHit).magnitude ) / 0.0174532925199433f;
15          }
16      }
```

6. Abnahme- und Einführungsphase

6.1. Abnahme

6.2. Einführung

6.3. Dokumentation

7. Fazit

7.1. Soll-/Ist-Vergleich

7.2. Lessons Learned

7.3. Ausblick

Literaturverzeichnis

[1] *Unternehmensporträt der ckc.* 2016. URL: http://www.ckc-group.de/index.php?id=709.

[2] *Gamifizierung ist die Übertragung von spieltypischen Elementen und Vorgängen in spielfremde Zusammenhänge mit dem Ziel der Verhaltensänderung und Motivationssteigerung bei Anwenderinnen und Anwendern.* 2016. URL: http://wirtschaftslexikon.gabler.de/Definition/gamification.html.

[3] *Die Arduino Hardware ist ein Entwicklerboard zum Basteln von eigenen Schaltkreisen und zum Ausführen von Steuerungsprogrammen für die Verwaltung von angeschlossener Elektronik.* 2016. URL: https://www.arduino.cc/en/Guide/Introduction.

A. Anhang

Abbildung A.1.: Verwendete Ressourcen

Hardware:

- Büroarbeitsplatz mit Standrechner

Software:

- Windows 7 Professional Service Pack 1 - Betriebssystem
- Unity 3D Version 5.3.3f1 - 3D Entwicklungsumgebung
- Visual Studio 2015 - Code Entwicklungsumgebung
- MiKTEX- Distribution des Textsatzsystems TEX
- TEXMaker - LATEXSchreibprogramm
- Dia Version 0.97.2 - Anwendung zum Zeichnen strukturierter Diagramme
- Entwickler - Implementierung der Scripte / Realisierung
- Anwendungsentwickler - Code Begutachtung

Abbildung A.2.: Model View Controller

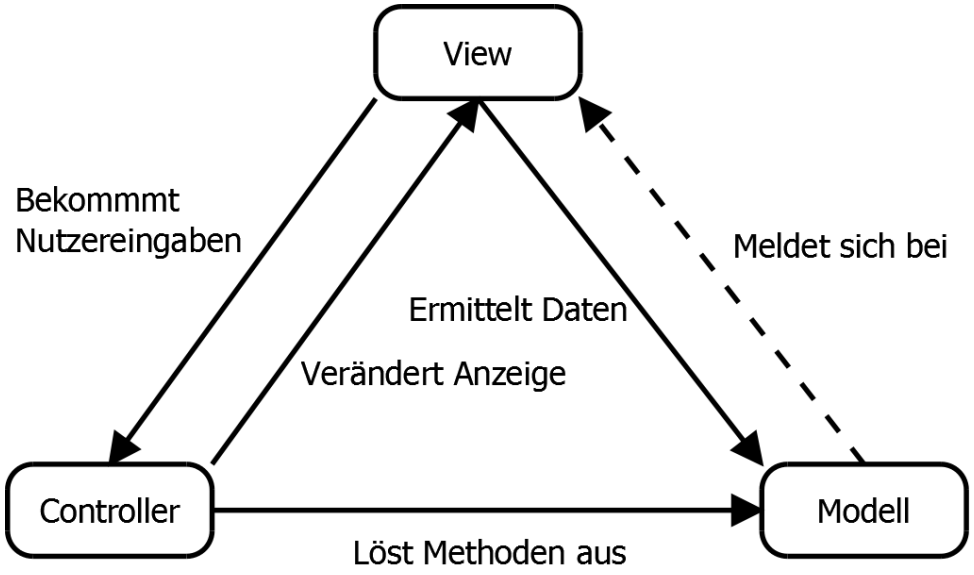


Abbildung A.3.: Presentation Abstraction Control

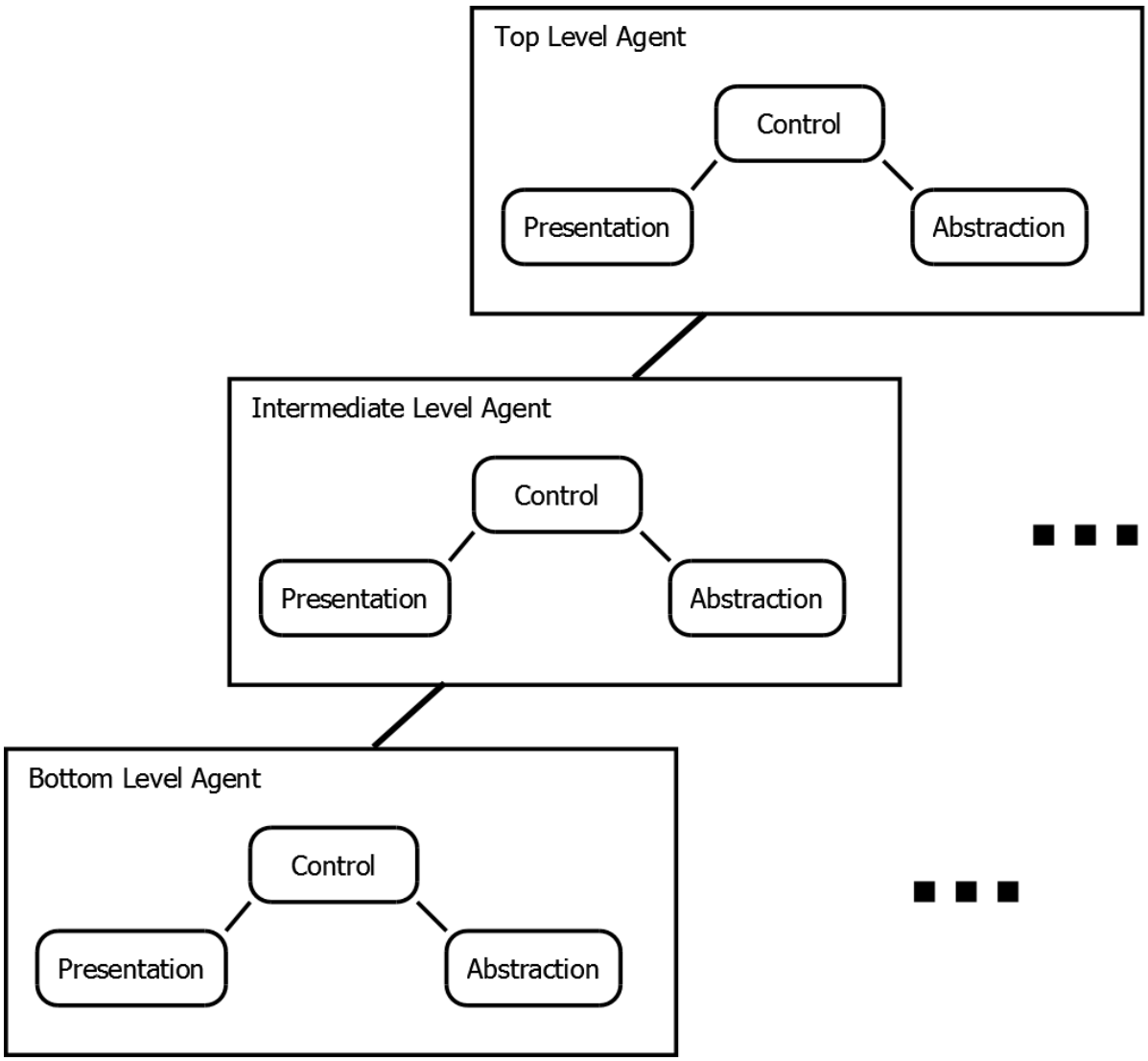


Abbildung A.4.: Benutzer Oberfläche Skizze

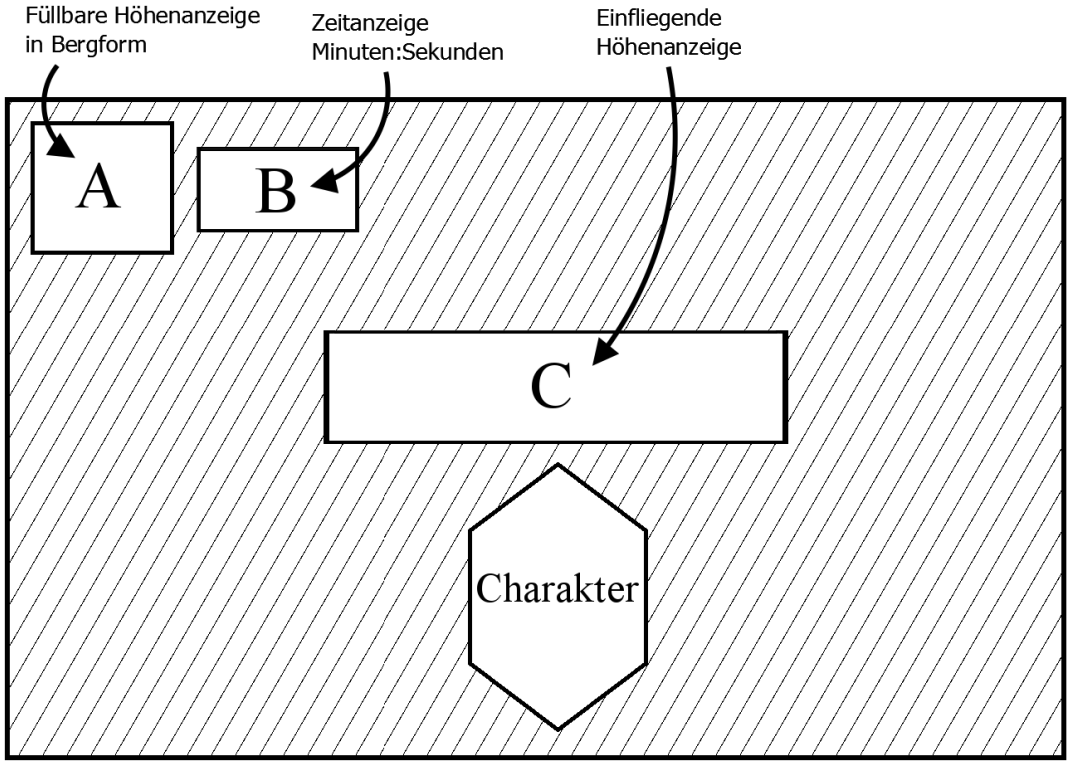


Abbildung A.5.: Benutzer Oberfläche Im Spiel

Abbildung A.6.: Klassendiagramm Charaktersteuerung mit Sphere

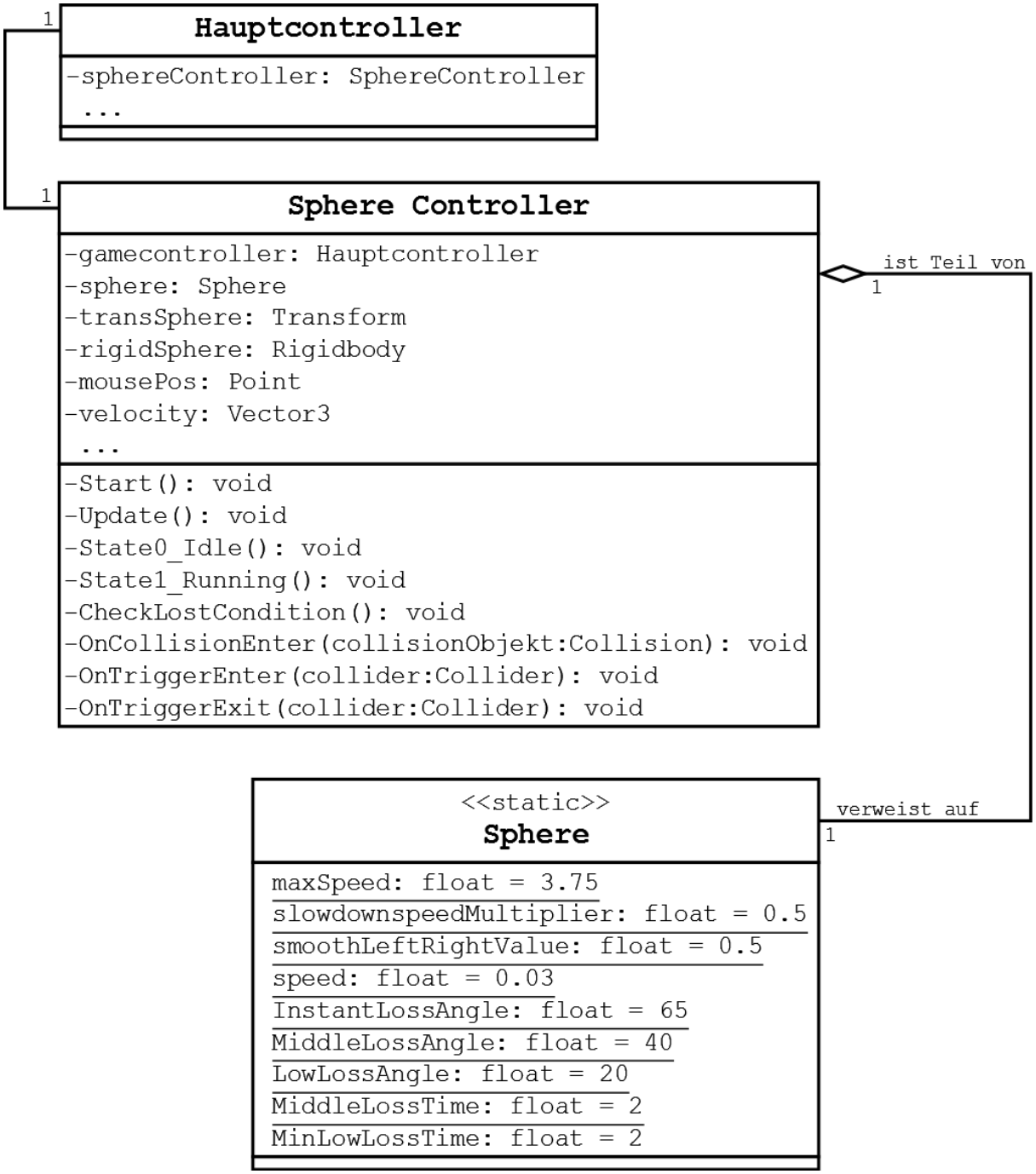


Abbildung A.7.: Zustandsdiagramm Statusverwaltung

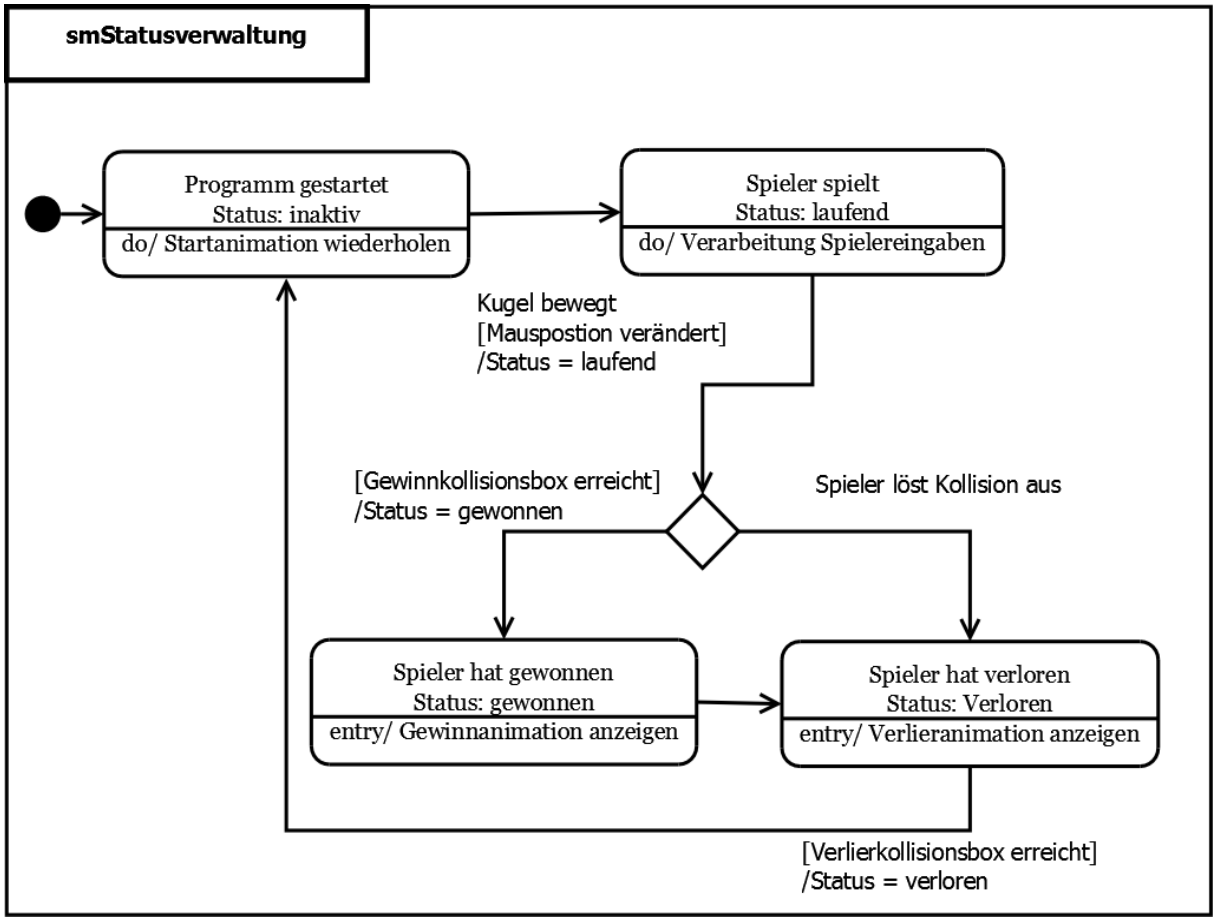


Abbildung A.8.: Gewinn Triggerbox am Gipfel des Berges

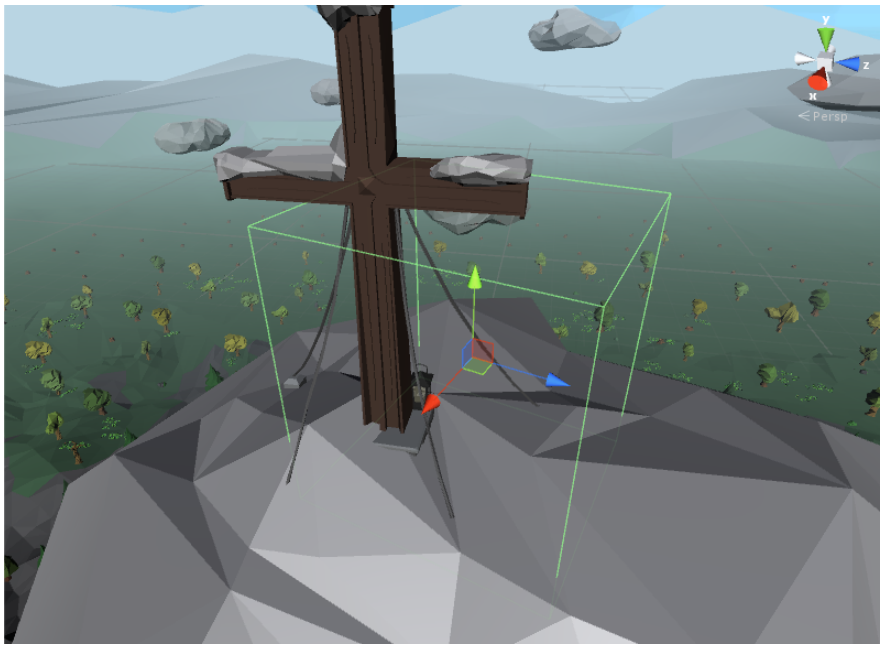


Abbildung A.9.: Baum Triggerbox

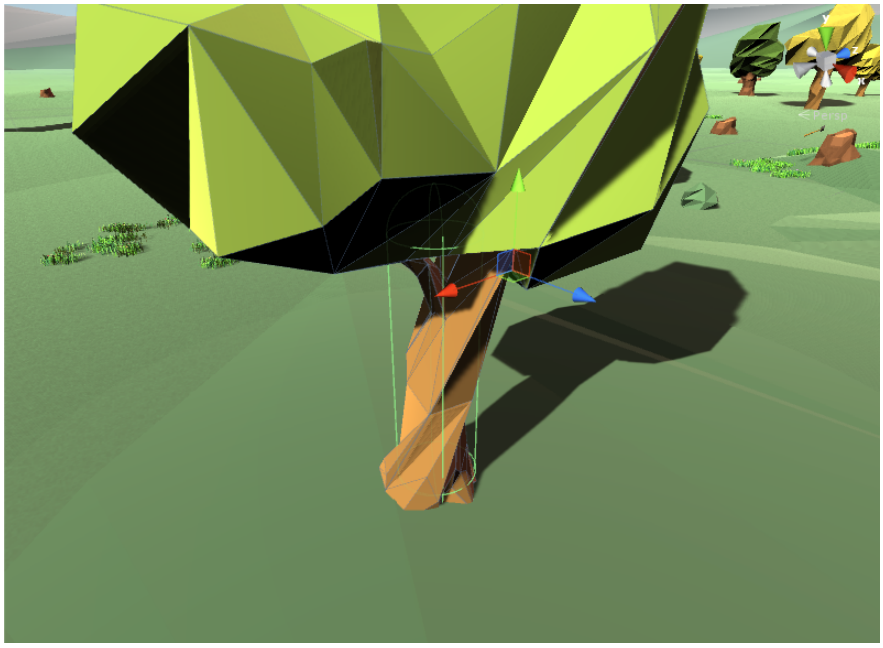


Abbildung A.10.: Busch Triggerbox am Rand des Berges

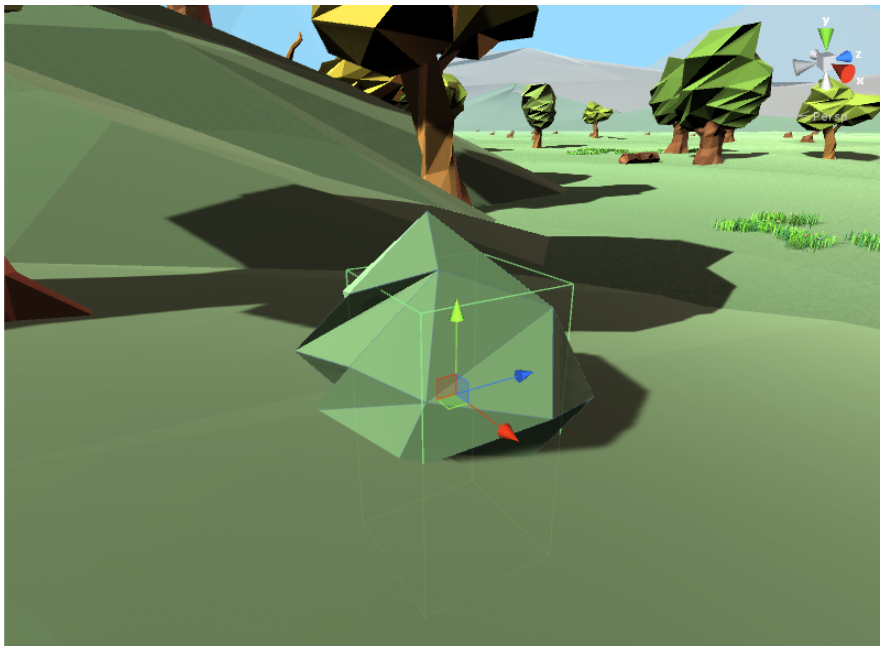
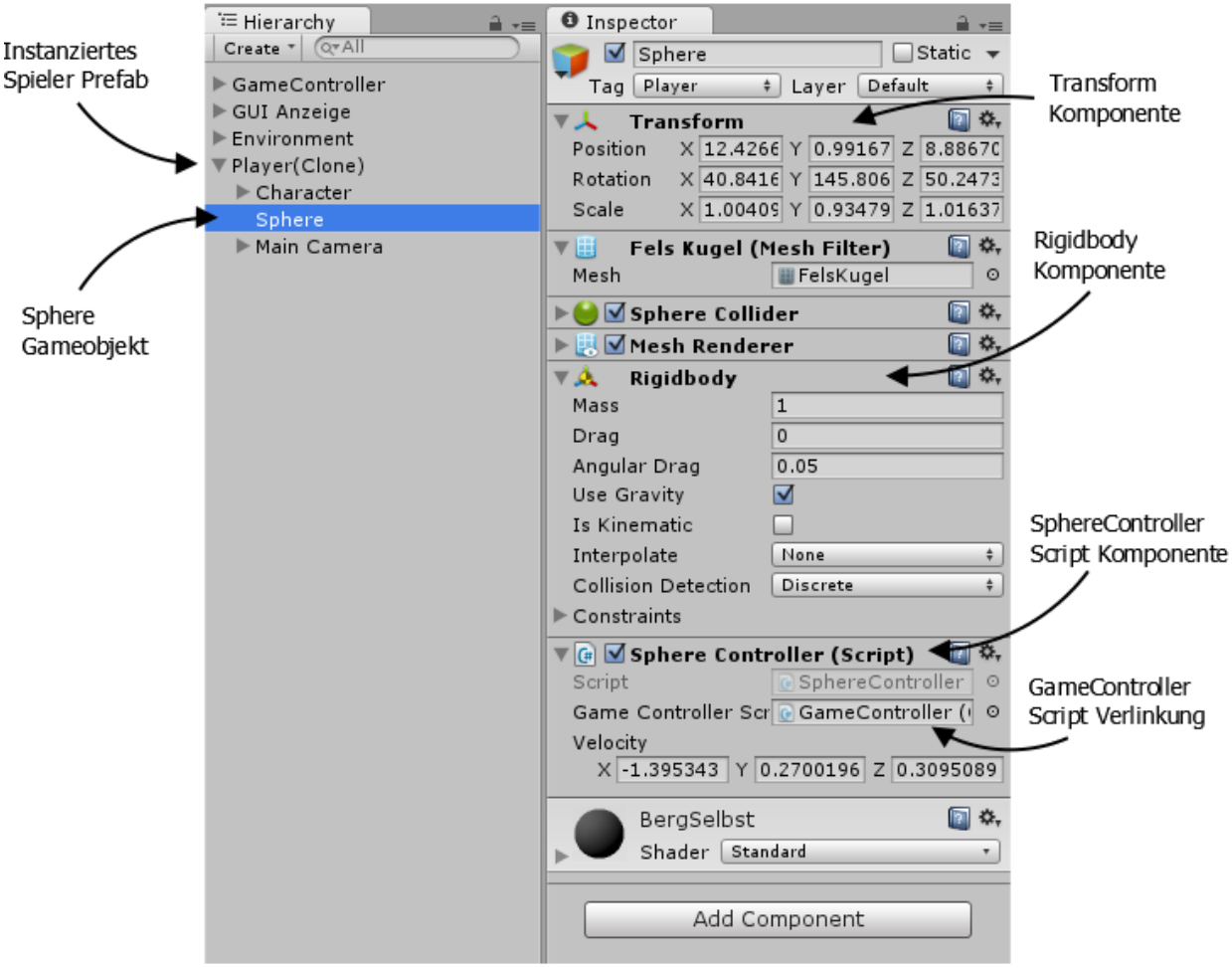


Abbildung A.11.: Ansicht des Inspektors für die Sphere und deren Komponenten in Unity



Codeausschnitt A.1: „stSphere“ statische Klasse

```
1  public static class stSphere {
2
3      //Festgelegte Variabeln
4      //----------------------------
5      //Maximalgeschwindigkeit der Kugel
6      public static float MaxSpeed
7      { get{return 3.75f;} }
8
9      //Geschwindigkeitsmultiplikator, um den die velocity der Kugel
           verlangsamt, wenn sie durch einen Busch rollt
10     public static float SlowDownSpeedTo
11     { get{return 0.5f;} }
12
13     //Wert um den der MausY Wert erweitert wird, damit die Bewegung
           sauberer nach vorne ist
14     public static float SmoothLeftRightValue
15     { get{return 0.5f;} }
16
17     //Wert um den die Kugel beschleunigt wird
18     public static float Speed
19     { get{return 0.03f;} }
20
21     //Winkel Niederlagebedingungen und Timerzeiten
22     //Höchster Winkel bei dem das Spiel sofort verloren ist
23     public static float InstantLossAngle
24     { get{return 65f;} }
25
26     //Mittlerer Winkel bei dem das Spiel nach 2 Sekunden verloren ist
27     public static float MiddleLossAngle
28     { get{return 40f;} }
29
30     //Kleinster Winkel 4 - 2 Sekunden Zeit
31     public static float LowLossAngle
32     { get{return 20f;} }
33
34     //Zeit für mittleren Winkel
35     public static float MiddleLossTime
36     { get{return 2f;} }
37
38     //Endzeit für niedrigsten Winkel
39     public static float MinLowLossTime
40     { get{return 2f;} }
41 }
```