Evaluierung des Unity-Frameworks für die Generierung und Visualisierung von 3D Szenen

Studienarbeit

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von:	Franziska Kolbe
geboren am:	10. Januar 1985
in:	Eberswalde
Gutachter:	Dr. Peter Eisert
eingereicht am:	4. Februar 2013

Inhaltsverzeichnis

Abbildungsverzeichnis iv

Verzeichnis der Skripte vii

1. Einleitung 1

2. Konzept 2
   - 2.1. Allgemein . . . 2
   - 2.2. Funktionsweise . . . 3
   - 2.3. Asset Pipeline . . . 5
   - 2.4. Scripting . . . 6
   - 2.5. Tools . . . 8
     - 2.5.1. Terrain Engine . . . 8
     - 2.5.2. Tree Creator . . . 8
     - 2.5.3. Particle Systems . . . 10
     - 2.5.4. ShaderLab . . . 10
     - 2.5.5. Animation Tool . . . 13
   - 2.6. Liste sonstiger Features von Unity . . . 15

3. Benutzeroberfläche 17
   - 3.1. Fensteraufbau . . . 17
   - 3.2. Navigation . . . 20
   - 3.3. Objekte positionieren . . . 22

4. Das GameObject 24
   - 4.1. Aufbau . . . 24
   - 4.2. Komponentenarten . . . 25
   - 4.3. Gruppierung . . . 26
   - 4.4. Prefabs . . . 27

5. Eigene Objekte 28
   - 5.1. Texturen . . . 28

5.2. 3D Objekte 29

6. Szenenaufbau 32
6.1. Szene 32
6.2. Kamera 32
6.3. Grundfläche 34
6.4. Licht 35
6.5. Hintergrund 36
6.6. Objekte 37
6.7. Build Settings 39

7. Interaktion von Objekten 41
7.1. Skripte in Unity 41
7.2. Variablen im Inspector 43
7.3. Objekte adressieren 44
7.4. Andere wichtige Klassen und Funktionen 46
7.5. GUI 49

8. Steuerung von Objekten und Spielern 53

9. Beispiele 56
9.1. Mondfahrzeug fahren 56
9.2. Kugeln 59
9.3. Terrain 63
9.4. Start 66
9.5. Simon Says 68
9.6. Shooter 75

10. Schlussteil 80

Literaturverzeichnis 81

Abbildungsverzeichnis

1.1. Unity Konzept. 1

2.1. Plattformen. 3
2.2. Unity API. 7
2.3. Tools zum Gestalten eines Terrains. 8
2.4. Tree Creator. 9
2.5. Particle System. 10
2.6. Animation View. 14

3.1. Toolbar. 17
3.2. Hierarchy View der Szene „Terrain“ und Project View. 18
3.3. Auswahlmöglichkeiten für den Draw Mode in der Scene View. 18
3.4. Scene View und Game View der Szene „Terrain“. 19
3.5. Play/Pause Button. 20
3.6. Buttons für Position und Ausrichtung der Achsen eines Objektes. 20
3.7. Scene Gizmo. 21
3.8. Scene Gizmo mit verschiedenen Ansichtsmöglichkeiten. 22
3.9. Verschieben. 22
3.10. Rotieren. 23
3.11. Skalieren. 23

4.1. GameObject mit möglichen Komponenten. 24
4.2. Beispiel eines primitiven Fahrzeugs. 27

5.1. Auswahlmöglichkeiten für Texturtyp. 28
5.2. Importeinstellungen für Texturen. 29
5.3. Importmöglichkeiten für 3D Objekte. 30
5.4. Importeinstellungen. 31

6.1. Main Camera. 32
6.2. Main und Back Camera. 33
6.3. First Person Controller. 34

6.4. Grundfläche. 35
6.5. Licht. 36
6.6. Hintergrund. 37
6.7. Ein Material für eine Steinmauer. 38
6.8. GUI Elemente. 39
6.9. Build- und PlayerSettings. 40

7.1. Application Design Model. 41
7.2. Klassendiagramm für Object. 42
7.3. Veränderbare Variablen. 43
7.4. Ergebnis der OnGUI()-Funktion im Spiel. 51
7.5. Ergebnis der if-Anweisung in der OnGUI()-Funktion. 52

8.1. Input Manager. 53

9.1. Die Szene Mondfahrzeug fahren. 56
9.2. Mondfahrzeug mit Schweif. 57
9.3. Script „Fahren“. 58
9.4. Material „Grau“. 58
9.5. Kugelkette. 59
9.6. Hinge Joint. 60
9.7. Kugelkette in Bewegung. 62
9.8. Terrain aus Sicht eines Flugzeugs. 63
9.9. Water Surface Splash. 64
9.10. Terrain aus Sicht eines Flugzeugs. 65
9.11. GUI Terrain. 66
9.12. Szene „Start“. 67
9.13. Simon Says. 75
9.14. Shooter. 76
9.15. Shooter: Feind. 76
9.16. Shooter: Bloodscreen. 77
9.17. Shooter: Tür. 77
9.18. Shooter: Munition. 78
9.19. Shooter: Granate. 78
9.20. Shooter: Explosion der Granate. 79

9.21. Shooter: Explosion des Feinds. 79

Verzeichnis der Skripte

2.1. ShaderLab Syntax für einen Shader 11
2.2. ShaderLab Syntax für einen Pass mit Cg Teil 12

7.1. Objekt bewegen mit Translate 44
7.2. Script adressieren mit GetComponent 44
7.3. GameObject adressieren mit Find 45
7.4. Instatiieren eines neuen Objekts 45
7.5. Objekt entdecken und dorthin drehen 46
7.6. Objekt mit Geschwindigkeit bewegen durch Time.deltaTime 47
7.7. Tür öffnen durch Raycast und Animation 47
7.8. Highscore in PlayerPrefs hinterlegen 48
7.9. Hauptmenu durch LoadLevel aufrufen 49
7.10. GUI Buttons mit Tooltip 50
7.11. GUILayout mit TextField zum Abfragen eines Spielernamens 51

8.1. Fahrzeugsteuerung durch Pfeiltasten 53
8.2. Koordinatenabfrage der Mausposition 54

9.1. Spieler aufs Spielfeld zurücksetzen 60
9.2. GUI einer Startszene 66
9.3. Kugel einfärben und aufleuchten lassen mit OnMouseOver 68
9.4. Simon Says Skript 69

1. Einleitung

Unity 3D ist eine Spiel-Engine für die Entwicklung von 3D Spielen jeder Art. Durch Unity wird der Spielverlauf gesteuert und der visuelle Spielablauf dargestellt. Die Modellierung der 3D Komponenten muss größtenteils in externen Programmen realisiert werden, da in Unity nur einige primitive vorgefertigte 3D Komponenten integriert sind. Es besteht jedoch die Möglichkeit direkt in Unity sogenannte Terrains zu entwerfen, die zu dem sehr effizient sind. Sehr große Pluspunkte von Unity sind die ständige Testbarkeit während der Spielentwicklung und der gute Support durch Unity‘s Webseite. Dort findet man eine sehr gute Dokumentation, ein Handbuch, Tutorials, sowie die Community, in der einem jederzeit Fragen beantwortet werden.

Die Basisversion ist kostenlos und für Windows, sowie Mac OS X erhältlich. Des Weiteren gibt es eine kostenpflichtige Version, UnityPro, die einige Features mehr birgt, wie zum Beispiel Realtime Shadows oder einen Profiler, um ein Spiel zu optimieren. Die derzeit aktuelle Version ist Unity 3.5, mit der es möglich ist Spiele für Windows, Mac OS, Webplayer, Flash, iOS, Android, XBox 360, Playstation 3 und Wii zu entwickeln. Diese Studienarbeit soll Einblicke darüber vermitteln, wie man mit Unity arbeitet.

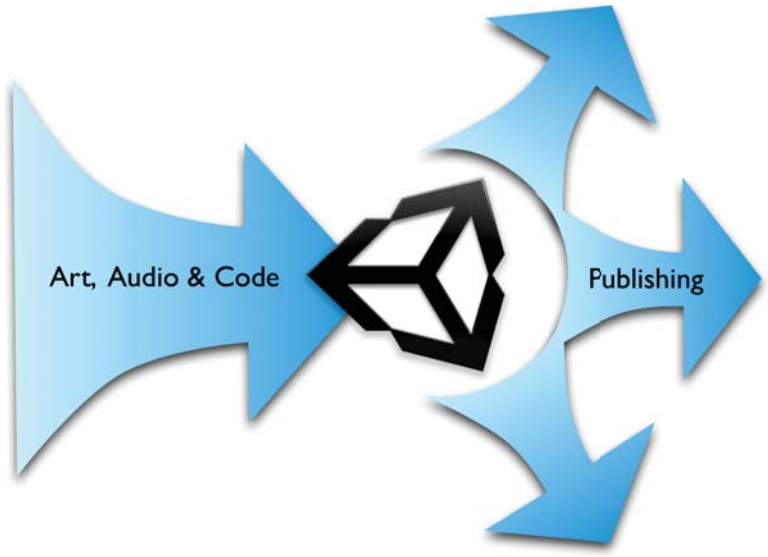


Abbildung 1.1.: Unity Konzept.

2. Konzept

2.1. Allgemein

Das Konzept von Unity beruht darauf mit dem Inhalt, also der Landschaft, den Spielern, dem Sound und weiteren Objekten zu beginnen und nicht mit dem Programmieren. Es besitzt einen Editor, in dem man sich seine Spielwelt zuerst zusammenbauen und währenddessen Skripte hinzufügen kann. Das bedeutet, im Gegensatz zum Normalfall kontrolliert nicht das geschriebene Programm den Inhalt, sondern andersherum: alle geschriebenen Skripte sind Teil der Spielwelt. Die Zusammensetzung der Spielszene erfolgt aus *GameObjects*, also Objekten verschiedenster Art, denen durch Komponenten ihre Eigenschaften zugeordnet werden. Hält Unity keine passende Komponente für einen Zweck bereit, kann man sich diese programmieren und als Skript-Komponente an ein *GameObjects* hängen.

Unity richtet sich an Anfänger, Fortgeschrittene sowie Profis in der 3D Spielentwicklung, wobei diese drei Gruppen auch gleichzeitig an einem Projekt arbeiten können. Da Unity in der Lage ist, Objekte und Modelle aus einer Vielzahl an Programmen zu importieren und in einem Projekt weiter zu verarbeiten, kann jeder Entwickler seine persönlichen Vorlieben ausüben. Das gilt auch für das Scripting: Innerhalb eines Projektes können Skripte in den Sprachen JavaScript, C# und Boo, welche Python ähnelt, verwendet werden.

Mit Unity besteht auch für Anfänger die Möglichkeit, komplette Spiele selbst zu entwickeln und es ist auch nicht unbedingt nötig, sich mit einer 3D-Modellierungssoftware auszukennen, wenn das Spiel keine hohen grafischen Anforderungen erfüllen soll. Einige primitive Objekte sind schon im Unity Editor zu finden. Des Weiteren findet man eine Vielzahl von 3D Objekten zum Download im Internet, die man nur noch in Unity zu importieren und anschließend zu einem Spiel zusammensetzen braucht. Auch Unity selbst hat dazu ein Angebot: den *Asset Store*. Hier kann man neben Spielfiguren auch Texturen und Materialien, Tools zur Landschafts- und Spielgestaltung, Editorerweiterungen in Form von Plugins und Skripte herunterladen oder selbst hochladen. Es gibt im *Asset Store* auch Demos und Tutorials, die einem das Lernen von Unity erleichtern. Einiges findet man dort sogar kostenlos.

Die eingebaute Grafik-Engine basiert je nach Zielplattform auf *OpenGL* oder *Direct3D*. Unity arbeitet mit der NVIDIA® PhysX® Engine. Dadurch sind viele physikalische

Interaktionen von Objekten möglich. Man kann feste Körper erstellen auf die physikalische Kräfte wirken können. Es ist auch möglich Verbindungen zwischen Objekten zu simulieren, die sich wie Sprungfedern, Scharniere oder Körperglieder verhalten [52].

Natürlich kann man mit Unity nicht nur Einzelspieler-Programme gestalten, es gibt auch Mehrspieler- sowie Netzwerkfunktionen [51].

Ein weiterer großer Vorteil von Unity ist die Plattformunabhängigkeit. Hat man ein Spiel einmal entwickelt, kann man es sehr schnell an alle möglichen Plattformen anpassen. Teilweise muss man nur einiges an der Steuerung ändern.

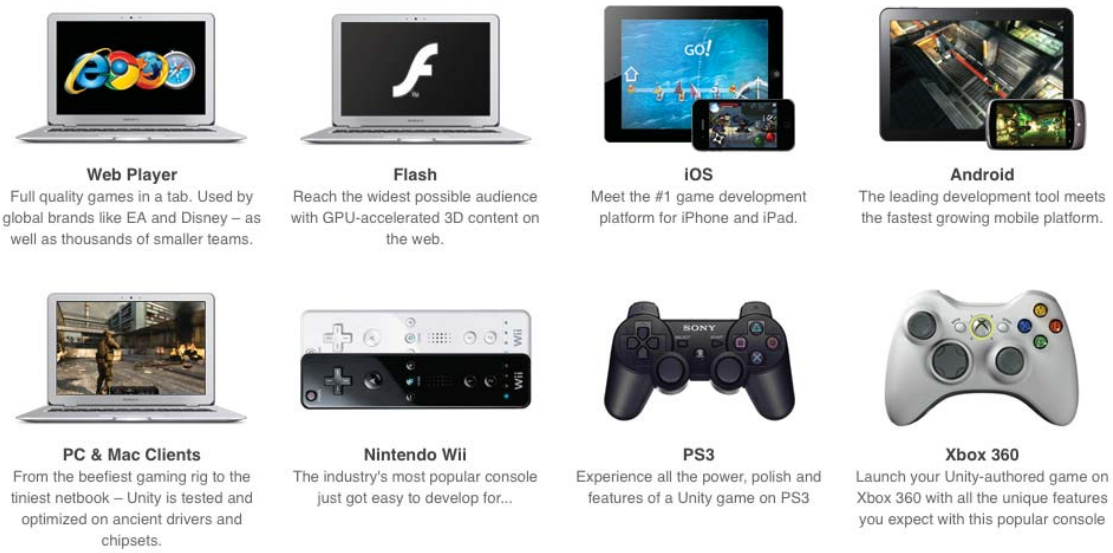


Abbildung 2.1.: Plattformen, für die man mit Unity Anwendungen entwickeln kann [53].

Um Spiele für iOS, Android, Playstation, Xbox und Nintendo Wii zu veröffentlichen, benötigt man jedoch jeweils eine Lizenz von Unity. Nutzt man die kostenlose Variante von Unity, ist es möglich sein Spiel zu veröffentlichen und sogar Geld damit zu verdienen. Allerdings kann man einige Features nicht nutzen, wie zum Beispiel Deferred Shading, Echtzeit-Schatten, Spiegelungen, Postprocessing Effekte, Plugins und Profiler. Außerdem erscheint im Splash Screen des fertigen Spiels immer das Unity Logo.

2.2. Funktionsweise

Für ein Unity Projekt benötigt man viele verschiedene Objekte. Diese werden alle im Projektordner gespeichert und sind über die *Project View* in Unity, aber auch über das

Dateisystem des Betriebssystems (OSX: Finder/ Windows Explorer) zugänglich. Der Hauptbegriff für alle Ressourcen, die in einem Unity Projekt benötigt werden, ist *Assets*. Dies können Texturen, 3D Meshes, Sounds, Skripte, Animationen, Materialien, Prefabs und mehr sein. Unity liefert *Standard Assets* mit, die man sich in jedes neue Projekt laden kann, wodurch man Grundbausteine für sein Spiel erhält. Des Weiteren kann man seine selbst erstellen *Assets* aus anderen Programmen in sein Projekt integrieren. Nicht alle Objekte, die sich im Projektordner befinden, werden am Ende für das fertige Spiel gebraucht. Einige befinden sich dort evtl. nur zu Testzwecken oder sind mit den *Unity Standard Assets* gekommen. Im Built-Vorgang werden von Unity nur *Assets* berücksichtigt, die auch wirklich in den zu kompilierenden Szenen vorkommen.

Hier ist die nächste Begrifflichkeit zu klären. Auch eine Szene ist ein *Asset*, sie ist aber eine Zusammensetzung aus anderen *Assets* und *GameObjects*, sowie Komponenten und kann als ein Level angesehen werden. Das Grundkonzept von Unity ist das GameObject-Komponenten-Modell. *GameObjects* sind zuerst leere Objekte, denen dann Komponenten hinzugefügt werden, wodurch sie ihre spezifischen Eigenschaften, wie Aussehen und Verhalten bekommen und miteinander kommunizieren können. Komponenten sind von Unity schon vorgefertigte Skripte, die die verschiedensten Funktionen haben, zum Beispiel zum Erzeugen von Verhalten, zur Definition von Aussehen und zum Beeinflussen anderer Aspekte einer Objektfunktion im Spiel [3]. So erzeugen Lichtkomponenten Licht, Kamerakomponenten erzeugen ein Bild der dargestellten Szene und die Rigidbody-Komponente lässt physikalische Kräfte auf ein Objekt wirken. Auch selbst geschriebene Skripte sind Komponenten. Um Komponenten gezielter auf seine Bedürfnisse abzustimmen, erhalten diese Variablen. Das können einzelne Zahlen sein, um einen Wert einzustellen, aber auch andere *GameObjects* der Szene oder *Assets*, wie Materialien, Sounds, Animationen etc. Man kann ein *GameObject* samt seiner Komponenten als ein *Asset* im Projektordner speichern, um es mehrmals wieder verwenden zu können. Dazu erstellt man daraus ein Prefab.

Eine neue Szene erzeugt man, indem man seine Spielwelt mit *GameObjects* verschiedener Funktionen füllt. Dies kann man zum einen im Editor tun, indem man Prefabs aus seinem Projektordner in die Szene zieht oder aus den von in Unity schon vorhandenen *GameObjects* (auch das leere) auswählt, woraufhin sie von Unity in die Szene gesetzt werden. Man kann aber auch fast alle *GameObjects* per Script zur Laufzeit erzeugen lassen. Dazu muss die Szene zumindest ein *GameObject* enthalten, da Skripte immer an Objekte gehängt werden. Da sich Unity durch seinen Szeneneditor auszeichnet, ist dies jedoch nicht ratsam und

verkompliziert den Spielentwicklungsablauf, da man sich im Szeneneditor seine Spielwelt nicht anschauen kann. Nach dem Aufbau der Spielwelt mit allen Objekten und Ausrichtung der Kamera und des Lichts beginnt man das Verhalten der einzelnen Objekte festzulegen. Dazu schreibt man Skripte, welche als Komponenten an das entsprechende *GameObject* gehängt werden. Bei einem kleinen Spiel, wie etwa ein Puzzlespiel, für das man nur ein Skript benötigt welches die komplette Handlung steuert, kann man dieses einfach der Hauptkamera (Main Camera) hinzufügen oder ein leeres *GameObject* zur Szene hinzufügen, welches nur dieses Skript als Komponente hat. Um einen Spieler zu steuern, kann man ein eigenes Skript schreiben, jedoch liefert Unity in seinen *Standard Assets* auch schon eines dafür mit, welches man sich noch individuell anpassen kann. Im Unity *Asset Store* sind auch Skripte für alle möglichen Situationen vorhanden. So muss man fast nichts selber machen, außer seine Szene zusammenstellen. Den Rest erledigt Unity bei der Kompilierung.

2.3. Asset Pipeline

Die *Asset Pipeline* von Unity ist ein großer Pluspunkt. Da es fast alle für die Spielentwicklung gängigen Dateiformate unterstützt, kann man diese direkt per Drag&Drop in den Unity Editor ziehen und für sein Spiel verwenden. Möchte man doch noch Änderungen daran vornehmen, kann man die Datei aus Unity heraus öffnen. Das heißt man öffnet die Datei in seinem Unity Projekt per Doppelklick, woraufhin sich das entsprechende Programm zum Bearbeiten der Datei öffnet, wie zum Beispiel Blender bei .blend-Dateien oder Photoshop bei .psd-Dateien. Editiert man diese Dateien nun im entsprechenden Programm und speichert diese anschließend, werden sie in Unity direkt aktualisiert und man muss sie nicht neu einfügen. Möglich ist dies durch die Schritte, die Unity im Hintergrund ausführt. Es überprüft ständig den Assets-Ordner und stellt sofort fest, wenn sich ein Objekt darin befindet, welches es noch nicht kennt. Daraufhin wird dem *Asset* ein eindeutiger ID Wert zugeordnet, wodurch Unity dann auch Objekte mit gleichem Namen unterscheiden kann. Erst dann wird das *Asset* importiert und weiter verarbeitet. Was man also im Projektfenster sieht, ist das durch Unity weiterverarbeitete *Asset*, dessen Inhalt sich vom Original unterscheidet. So werden die oben genannten .psd-Dateien zwar als solche dargestellt, sind aber in Wirklichkeit in ein internes Format konvertiert worden, das zusätzlich Metadaten enthält. Die Metadaten werden im Library-Ordner gespeichert, weshalb man an diesem Ordner keine manuellen Veränderungen vornehmen sollte. Auch die Ordner, die man in seinem Projekt für Ordnungszwecke erstellt, werden von Unity

in das Dateisystem des Betriebssystems gespiegelt. Werden Ordner dort verändert oder verschoben, kann es zu Informationsverlusten kommen, weshalb man sein Projekt nur in Unity organisieren sollte [55].

2.4. Scripting

Innerhalb eines Projektes können Skripte in den Sprachen *JavaScript*, *C#* und *Boo* gleichzeitig verwendet werden und aufeinander zugreifen. Das ist möglich durch die Reihenfolge in der die Skripte kompiliert werden. Als erstes werden alle Skripte im Ordner „Standard Assets“ kompiliert, danach die Skripte in „Standard Assets/Editor“. Anschließend werden alle Skripte kompiliert, die sich nicht in diesen Ordnern befinden, also die selbst erstellten. Diese haben Zugriff auf alle im ersten Schritt kompilierten Skripte, wodurch die Interaktion zwischen verschiedenen Skriptsprachen möglich wird. Schreibt man ein JavaScript, welches auf ein C# Script zugreifen soll, speichert man das C# Skript in „Standard Assets“ und das JavaScript außerhalb. Das JavaScript kann das C# Skript dann direkt referenzieren. Durch diese Reihenfolge kommt es auch dazu, dass die Skripte in „Standard Assets“ länger brauchen um kompiliert zu werden. Denn trifft man eine Veränderung an einem „Standard Assets“-Skript, sind davon auch alle Skripte, die darauf zugreifen, also im dritten Schritt kompiliert werden, davon betroffen und müssen auch wieder kompiliert werden. Um Zeit beim Kompilieren zu sparen, sollte man also alle Skripte, die sich oft ändern, in die dritte Gruppe legen, denn darauf greift kein Skript einer anderen Gruppe zu [63]. Die Spiellogik läuft auf der Open Source .NET Plattform *Mono*. Daher können, je nach .NET Kompatibilitätslevel in den Projekteinstellungen, .NET Klassenbibliotheken für die Skripte in Unity verwendet werden [62].

Alle Skripte werden zu DLLs kompiliert. In der ausführbaren Datei befinden sich diese zusammengefasst im Ordner „Data“ und werden nach der Skriptsprache benannt. Alle Skripte, die wie oben beschrieben im ersten Schritt kompiliert wurden, werden nach Sprache zusammengefasst und in den DLLs „Assembly-CSharp-firstpass.dll“ und „Assembly-UnityScript-firstpass.dll“ gespeichert. Analog werden die Skripte aus dem dritten Schritt in „Assembly-CSharp.dll“ und „Assembly-UnityScript.dll“ zusammengefasst. Hat man die Sprache *Boo* verwendet, kommt entsprechend noch eine DLL dafür hinzu. Man kann seinem Projektordner auch externe Managed .NET DLLs hinzufügen und diese in seinem Unity Projekt verwenden, natürlich nur soweit der Inhalt von Unity unterstützt wird [8]. Die Klassen, die es beinhaltet können dann wie normale Skripte verwendet werden

[61]. Die Vorteile bestehen darin, dass man Mono Code von Dritten für sein Projekt benutzen kann und dass man seinen eigenen Unity Code ohne seine Quelle veröffentlichen kann (z.B. im *Asset Store*). Richtige Plugins, also Libraries, geschrieben in C oder C++ werden leider nur von UnityPro unterstützt [57].

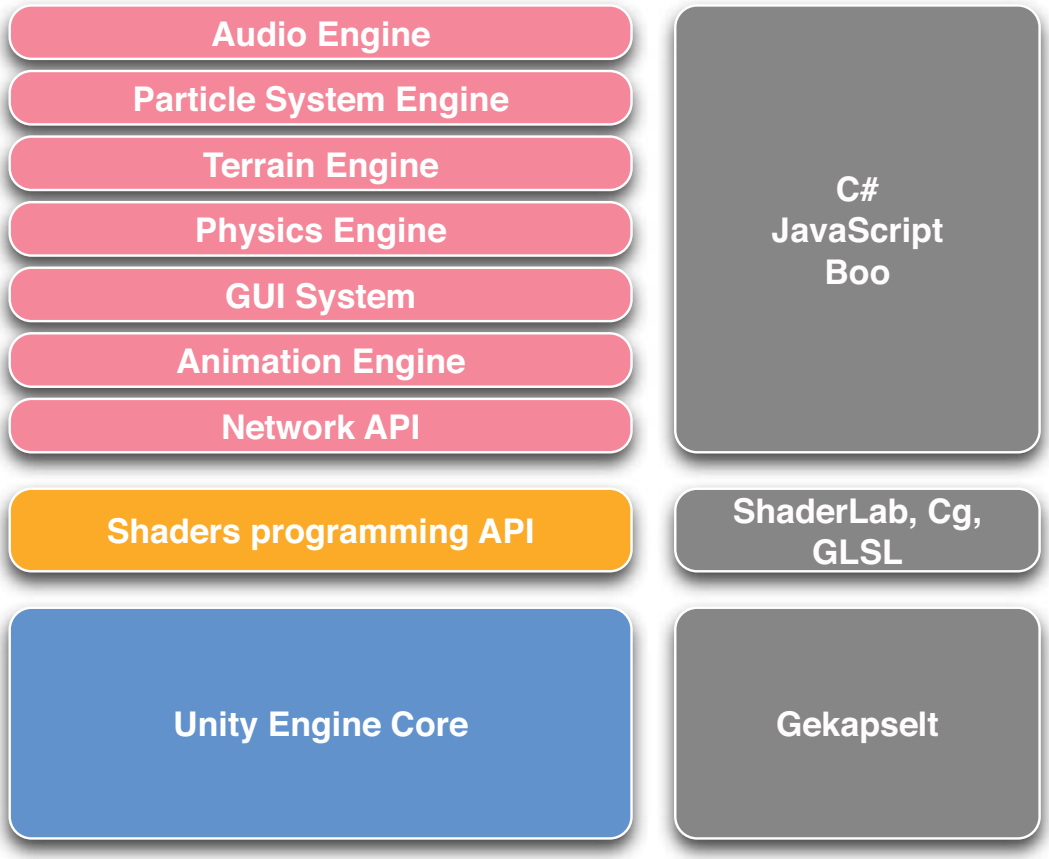


Abbildung 2.2.: Unity API [6].

Unity Dateien oder Teile von Unity außerhalb von Unity zu verwenden, gestaltet sich sehr schwierig bis unmöglich, da da es kein .NET Programm bzw. Bibliothek ist, sondern eine eigenständige Anwendung, in die .NET Mono integriert ist. Auch Plugins in C oder C++ können nicht auf die Unity API zugreifen, weshalb sie nur als Ergänzung und nicht zur Spielprogrammierung selbst nützen [1]. Für ein iOS Projekt besteht jedoch die Möglichkeit die kompilierten Dateien in ein größeres Projekt einzubinden, da Unity ein XCode-Projekt kreiert.

2.5. Tools

2.5.1. Terrain Engine

Neben vielen *Assets* und Komponenten bringt Unity ein Tool zum Erstellen von Terrains mit. So kann man seine komplette Spielwelt direkt in Unity entwickeln, zu der man dann nur noch einzelne spielrelevante Objekte hinzufügen muss. Will man ein Terrain erstellen, beginnt man mit einer Grundebene. Mit den Werkzeugen kann diese dann an einigen Stellen angehoben oder abgesenkt werden. Die Werkzeuge sind immer in Pinselform, wie man es aus Bildbearbeitungsprogrammen kennt. Sie haben eine bestimmte Form, Größe und Deckkraft. Als weiteres Werkzeug gibt es einen Höhenregulator, mit dem man sein Terrain auf eine bestimmte Höhe erheben kann. Um Ecken und Kanten in den Höhendifferenzen zu glätten gibt es einen Weichzeichner. Es besteht die Möglichkeit eine Höhenkarte zu importieren, womit man sich das „Zeichnen“ in Unity spart. So kann man etwa mit Photoshop eine Höhenkarte in Graustufen erstellen und hat somit auch die Möglichkeit echte Landschaften abzuzeichnen, wenn man eine Landkarte als Vorlage benutzt.

Abbildung 2.3.: Tools zum Gestalten eines Terrains. Terrain erhöhen/absenken, Höhenregulator, Weichzeichner, Texturen zeichnen, Bäume platzieren, Details zeichnen, Terrain Einstellungen

Um der erstellten Landschaft nun ein natürliches Aussehen zu verleihen, kann man wieder mit einem Pinsel Texturen darauf verteilen. Die erste gewählte Textur ist die Grundtextur und wird automatisch dem gesamten Terrain hinzugefügt. Alle weiteren kann man dann nach belieben verteilen. Des Weiteren gibt es ein Werkzeug, mit dem man Bäume zu seinem Terrain hinzufügen kann. Das besondere daran ist, dass diese Bäume nur als 3D Objekte gerendert werden, wenn sich die Kamera in der Nähe befindet. Alle Bäume in der Ferne werden als Billboards gerendert. Auf die gleiche Weise, kann man auch noch Gras und Detail Meshes (z.B. Steine) zum Terrain hinzufügen [47].

2.5.2. Tree Creator

Mit dem Tree Creator Tool kann man Bäume erstellen, die man anschließend auf seinem Terrain verteilen kann oder auch als *GameObject* in einem Spiel verwenden kann. Dazu

beginnt man an der Wurzel und fügt dann hierarchisch Gruppen von Ästen und Zweigen hinzu. Dabei kann man die Anzahl von Ästen einer Hierarchieebene verändern. Die Äste werden automatisch nach einem bestimmten, wählbaren Muster hinzugefügt. Man kann diese aber auch selbst noch verändern. Als letzte Hierarchieebene fügt man Blätter hinzu, die als Endknoten wirken, man kann daran nichts mehr anhängen. Das Anhängen von Blättern erfolgt analog zum Anhängen der Äste und Zweige. Für jede Gruppe von Zweigen und Blättern kann man ein Material auswählen, welches jedoch einen bestimmten Shader für Blätter bzw. Zweige haben muss.

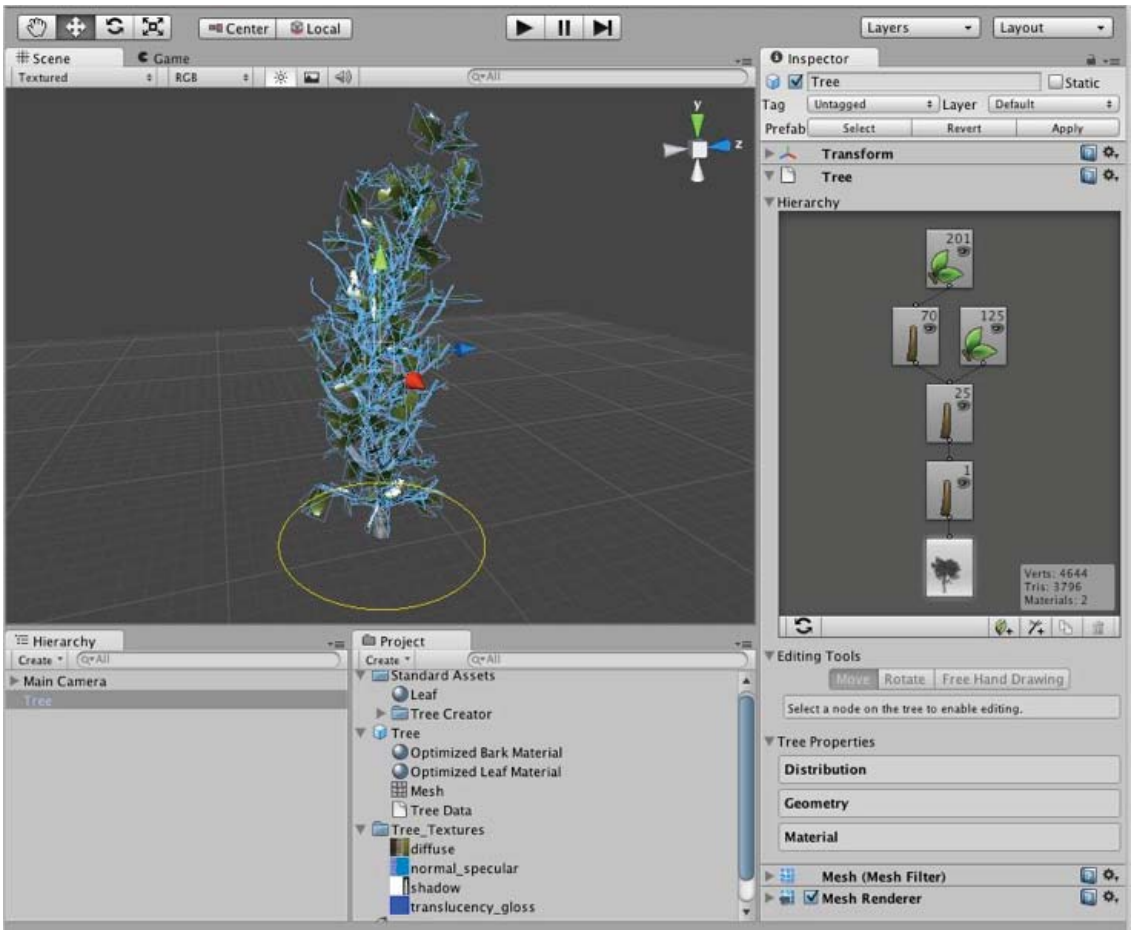


Abbildung 2.4.: Tree Creator. In der Scene View sieht man den Baum in 3D Ansicht. Im Inspector befindet sich die Hierarchieansicht des Baumes. Die Icons sind Knoten der Struktur und Stellen je eine Gruppe von Ästen oder Bättern dar. Die Zahl im Icon zeigt die Anzahl der Objekte in der Gruppe. Eine Gruppe kann sichtbar oder unsichtbar sein (offenes oder geschlossenes Auge) [48].

2.5.3. Particle Systems

Ein weiteres sehr vielseitiges Tool dient dem Erstellen von Particle Systems mit denen man Feuer, Rauch, Laub, Wasserbewegungen, Staub und mehr simulieren kann. In Unity gibt es zwei Arten von Particle Systems, zum einen das *Legacy*, was vor Version 3.5 zum Einsatz kam und das *Shuriken*, welches in der aktuellen Version genutzt wird und intuitiver zu bedienen ist. Das Besondere am *Shuriken Particle Emitter* ist, dass sich alle individuellen Werte als Kurve generieren lassen und so sehr präzise einstellbar sind.

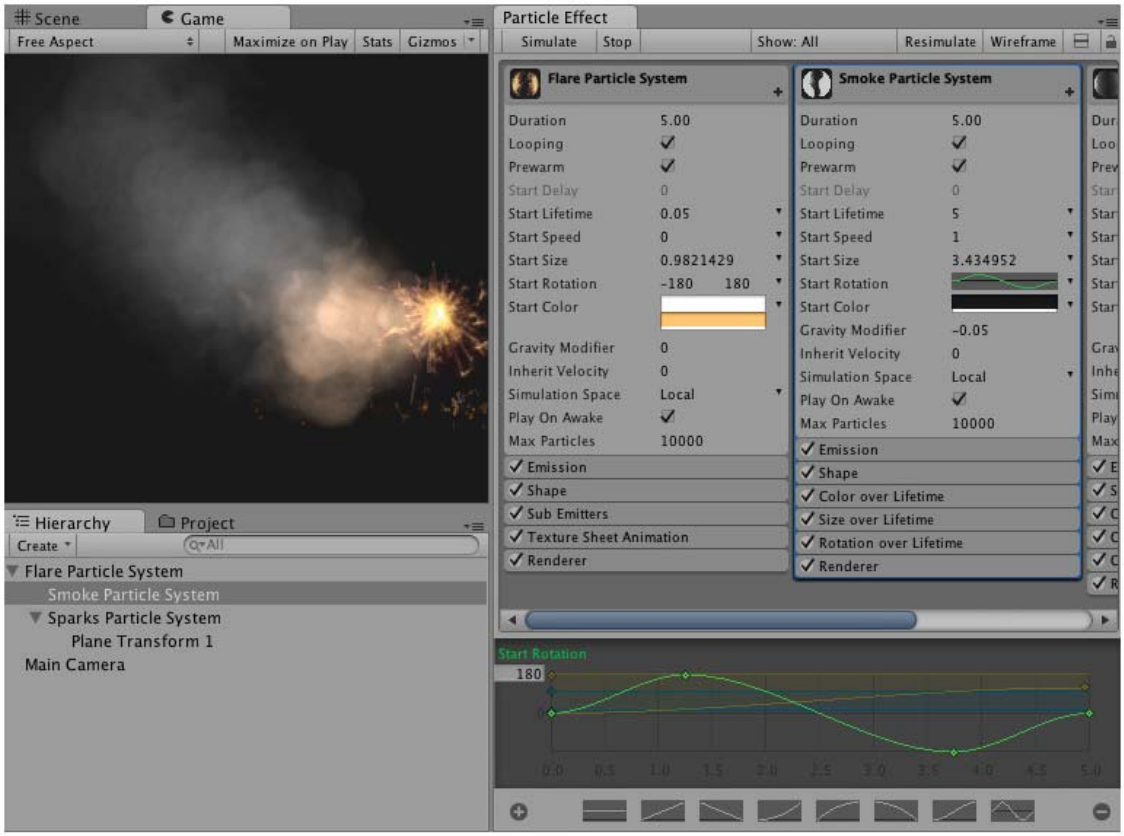


Abbildung 2.5.: Particle System. In der Game View sieht man das erzeugte Particle System in Echtzeit. Statt das System im Inspector anzupassen, kann man dies auch im Particle Effect Editor tun. Die Particle System Kurven kann man dort auch anpassen [45].

2.5.4. ShaderLab

Sehr gut eignet sich Unity auch für das Gestalten von Materialien für Mesh-Oberflächen, Skyboxen etc., da es viele Shader mitliefert. Shader sind kleine Skripte, die der Grafik-

Hardware vermitteln, wie Oberflächen gerendert werden sollen [58]. Diese reichen von einfach, wie *Diffuse* und *Glossy*, über anspruchsvoll, wie *Bumped Specular* und *Transparent Diffuse*, bis hin zu sehr fortgeschritten, wie *Reflective Normal Mapped Vertex-lit*.

Wem die mehr als 60 von Unity mitgelieferten Shader nicht ausreichen, kann sich mit Hilfe der Sprachen *ShaderLab* und *Cg/HLSL* eigene generieren. Ein Shader ist in Unity, wie die meisten Objekte, ein Asset und man erstellt ihn über das Menü: Assets -> Create -> Shader. Öffnet man diesen per Doppelklick über die Project View, kann man ihn editieren.

Die *ShaderLab* Syntax für einen Shader sieht folgendermaßen aus: Shader "name"{ [Properties] Subshaders [Fallback] }. Man erkennt hier, dass ein Shader aus mehreren *SubShadern* besteht. Diese sollen sich ähneln, aber in ihrer Komplexität abnehmen. Die Reihenfolge muss so gewählt werden, dass der anspruchsvollste Shader ganz oben steht, da Unity beim Rendern eines Shaders den ersten SubShader nimmt, den die Hardware unterstützt.

Wie auch im Beispielcode [46] unten zu erkennen, gibt es drei Arten von Shadern, die man erstellen kann:

- Surface Shaders: für Shader, die von Licht und Schatten beeinflusst werden,
- Vertex And Fragment Shaders: Shader, die nicht mit Licht interagieren müssen, für exotische Effekte,
- Fixed Function Shaders: für alte Hardware, die keine programmierbaren Shader unterstützt, werden komplett in der Sprache ShaderLab geschrieben [58].

```
Shader "MyShader" {
  Properties {
    _MyTexture ("My Texture", 2D) = "white" { }
    // definiert den Shader als 2D Textur
    // hier folgen andere Eigenschaften wie z.B. colors oder
       vectors
  }
  SubShader {
    // Hier wird ein SubShader definiert, welcher von folgender
       Art sein kann:
    //  - surface shader
    //  - vertex and program shader
```

```
    // - fixed function shader
    Pass {
      // Pass Befehle
      // Cg Teile
    }
    // Weitere Passes, falls benoetigt
  }
  SubShader {
  // hierhin kommt eine einfachere Version des vorherigen
     SubShaders, der auf aelteren Grafikkarten laeuft
  }
  // Optionaler Fallback: wird genutzt, falls kein SubShader auf
      der Grafikkarte des Nutzers laeuft
  // Genauer gesagt, wird hier befohlen, die SubShader eines
     anderen Shaders zu versuchen
  FallBack "VertexLit"
}
```

Skript 2.1: ShaderLab Syntax für einen Shader

Im Properties-Block werden alle Eigenschaften definiert, die später im Material-Inspector editiert werden können, wie beispielsweise die Grundfarbe oder die Helligkeit. Als nächstes werden *Passes* definiert, da jeder *SubShader* eine Kollektion aus diesen ist. Für jeden *Pass* wird die Objektgeometrie auf spezielle Art gerendert, weshalb es mindestens einen geben muss. Innerhalb des *Pass* kann man verschiedene Eigenschaften des Shaders festlegen, wie zum Beispiel „Lighting On“, was das *Standard Vertex Lighting* aktiviert. Außerdem wird das *SetTexture* Kommando gesetzt, welches festlegt, welche Texturen auf welche Weise für das Rendern genutzt werden [59]. Innerhalb der Passes kann die Programmiersprache Cg benutzt werden, welche von Unity in „low-level shader assambly“ für *OpenGL* und *Direct3D* kompiliert wird. Cg Teile in den Passes müssen immer durch „CGPROGRAM“ und „ENDCG“ gekennzeichnet werden [60].

```
Pass {
  Material {
    Diffuse [_Color]
    // ... mehr Material-Eigenschaften ...
  }
```

```
  Lighting On
  SetTexture [_MainTex] {
    constantColor [_Color]
    Combine texture * primary DOUBLE, texture * constant
      // wird fuer jeden Pixel, der auf dem Bildschirm gerendert
          wird, ausgefuehrt
  }
  // ... mehr des ueblichen pass state setups ...
    CGPROGRAM
      // Kompilationsanweisungen fuer diesen Teil, z.B.:
      #pragma vertex vert
      #pragma fragment frag
      // der Cg code selbst
    ENDCG
    // ... der Rest des pass setup ...
}
```

Skript 2.2: ShaderLab Syntax für einen Pass mit Cg Teil

Unity kompiliert die Cg Teile für Foward- und Deferred-Renderer, lässt es mit Lightmaps funktionieren und konvertiert es automatisch zu GLSL für mobile Geräte [54]. Benutzt man sehr fortgeschrittene Shadereffekte, kann man mit Unity‘s Grafikkartenemulation testen, wie diese in einem System mit älterer Hardware aussehen. In der Regel sind alle Shader abwärtskompatibel und Unity versucht auf jedem System das bestmögliche Ergebnis zu erzielen [42].

2.5.5. Animation Tool

Hat man in seinem 3D Programm eine Animation für ein Objekt entwickelt, kann man diese in Unity importieren. Sollte man nachträglich noch eine Animation für sein Objekt benötigen, kann man dies auch mit Unity‘s Animation Tool bewerkstelligen. Mit Unity ist es nicht nur möglich Bewegungen zu animieren, man kann auch die Variablen von Materialien und Komponenten animieren. Um eine Animation für ein Objekt zu erstellen, muss es sich in der Scene bzw. Hierarchy View befinden und ausgewählt sein. Beim Wechsel in die Animation View sieht man dort die Animationskurven des Objekts, falls schon welche vorhanden sind. Durch Drücken auf den Button mit dem roten Punkt, erstellt

man eine neue Animation, die man anschließend bearbeiten kann. In der Liste neben den Kurven sieht man alle Komponenten, Materialien und untergeordneten Objekte des ausgewählten Objekts. Die angegebenen Parameter der jeweiligen Komponente können per Animationskurve verändert werden. Um ein Objekt beispielsweise innerhalb von 1,5 Sekunden um 180° um die y-Achse zu drehen, wählt man für den ersten Frame bei 0 Sekunden eine y-Rotation von 0, wählt dann auf der Zeitleiste den Punkt für 1,5 Sekunden und ändert dort die y-Rotation auf 180. Unity erstellt nun die Rotation in einem Animation Clip. Man kann natürlich mehrere Werte in einem Animation Clip verändern. Außerdem kann man für jedes Objekt mehrere Animationen erstellen.

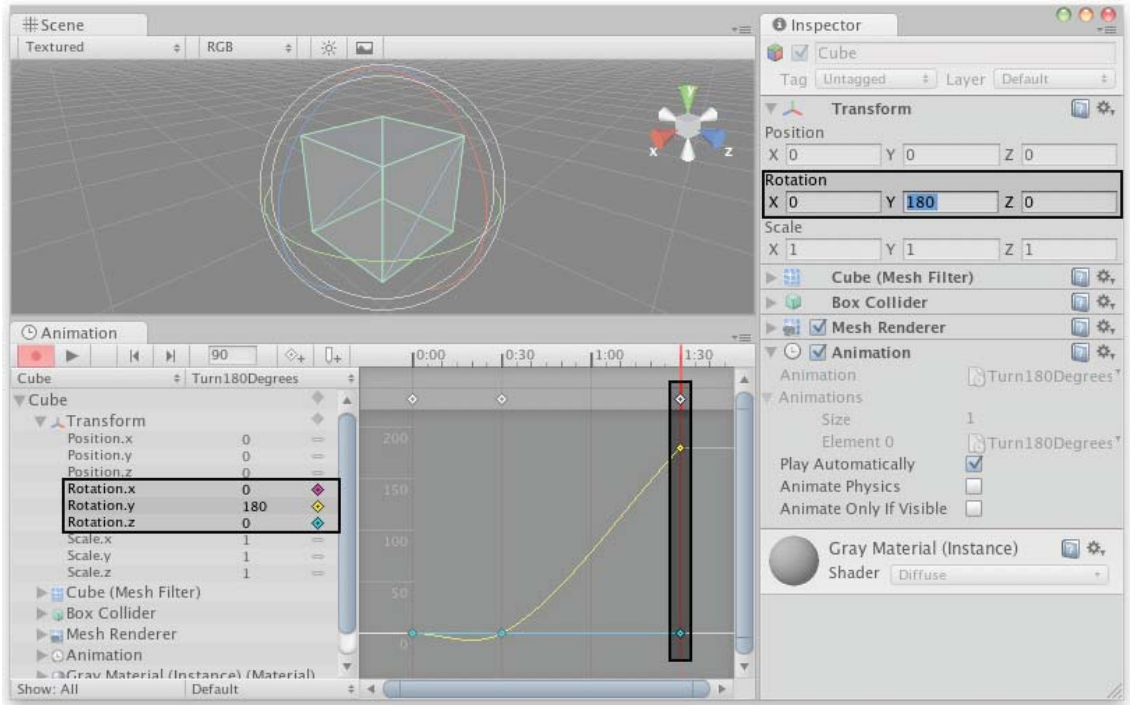


Abbildung 2.6.: Animation View. In der Scene View sieht man das zu animierende Objekt in der aktuellen Position für den markierten Frame. Da es rotiert werden soll, sind die Rotationsparameter der Transform Komponente markiert. Die Animationskurve für die Rotation um die y-Achse ist gelb dargestellt. Die rote vertikale Linie kennzeichnet den aktuellen Frame [49].

2.6. Liste sonstiger Features von Unity

- Deferred Rendering Pipeline: Unbegrenzt viele dynamische Lichter können dargestellt werden.
- Post-Processing Effekte für Bilder/Texturen(UnityPro): Sonnenstrahlen, hochqualitative Tiefenunschärfe, Linseneffekte, Chromatische Aberration und Farbkorrekturen sind nur einige Beispieleffekte, die erzielt werden können.
- Occlusion Culling mit Umbra (UnityPro): Es wird nur gerendert, was auch tatsächlich zu sehen ist.
- Realtime Shadows (UnityPro): Echtzeitschatten können von jeder Lichtquelle in der Spielszene geworfen werden.
- Ambient Occlusion (UnityPro): Durch geringen Rechenaufwand können sehr echt wirkende Schatten erzielt werden.
- Sun Shafts / Lenseffects (UnityPro): Eine Sonnenstrahlenkomponente kann der Kamera hinzugefügt werden, um einen bestimmt Effekte zu erzielen.
- Lightmapping: Helligkeitskarten für statische Objekte und Terrains, die direkt im Editor erzeugt werden können.
- Allegorithmic Substances: Hybride aus prozeduralen Texturen, Vektordateien und Bitmaps für hohe Detaillierung von Levels mit minimalem Datenaufwand.
- Cloth (Physic Engine): Physikalisch korrekte Animation von Kleidung.
- Ragdoll Wizard: Sehr schnelle Erstellung einer Ragdoll (Animation eines unbelebten meschlichen Körpers) eines animierten Charakters.
- Built-in Pathfinding durch NavMeshes (UnityPro): Automatische Navigation Meshes speichern alle Grenzen der Spielfläche und nutzen diese für das Pathfinding.
- Audio Engine FMOD: Führende Audio Engine und Bearbeitungswerkzeuge.
- Audio Filter (UnityPro): Durch Unterstützung von DSP Filtern können Stimmungen und Audioeffekte ohne Scripting hinzugefügt werden.
- Debugger: Integrierte Möglichkeit zum Finden von Fehlern in Skripten.

- Profiler (UnityPro): Spieloptimierung, indem die Szene Bild für Bild durchgegangen und der Zeitverbrauch überprüft werden kann.
- Anpassbarer Unity Editor: Eigene Editor Tools entwerfen durch GUI Scripting.
- Mip Map Generator: Antialiasing für Texturen.

3. Benutzeroberfläche

3.1. Fensteraufbau

Der Unity Editor besteht aus mehreren Fenstern mit Tabs, sogenannten Views, die man sich per Drag&Drop beliebig anordnen kann. Am oberen Rand des Editors befindet sich die Toolbar.

Abbildung 3.1.: Toolbar.

In der Project View sieht man alle Dateien bzw. Assets, die im Projektordner abgespeichert sind. Das sind Szenen, Texturen, Materialien, Skripte, 3D Modelle und mehr. Um Dateien hinzuzufügen, zieht man diese einfach per Drag&Drop in die Project View oder importiert sie mit Assets -> Import New Asset. Über das Assets-Menü kann man auch vorgefertigte Assets, wie Ordner, Animationen, Skripte und weiteres, erstellen, welche auch gleich automatisch im Projekt abgespeichert werden und somit in der Project View erscheinen.

Die Hierarchy View zeigt hierarchisch alle Objekte an, die sich aktuell in der Szene befinden, stellt also den Szenengraph dar. Sie gelangen dorthin, indem man Objekte aus der Project View direkt in die Szene oder in die Hierarchy View zieht. Wenn man durch das GameObject-Menü ein Objekt erstellt, befindet sich dieses auch direkt in der Szene und in der Hierarchy View.

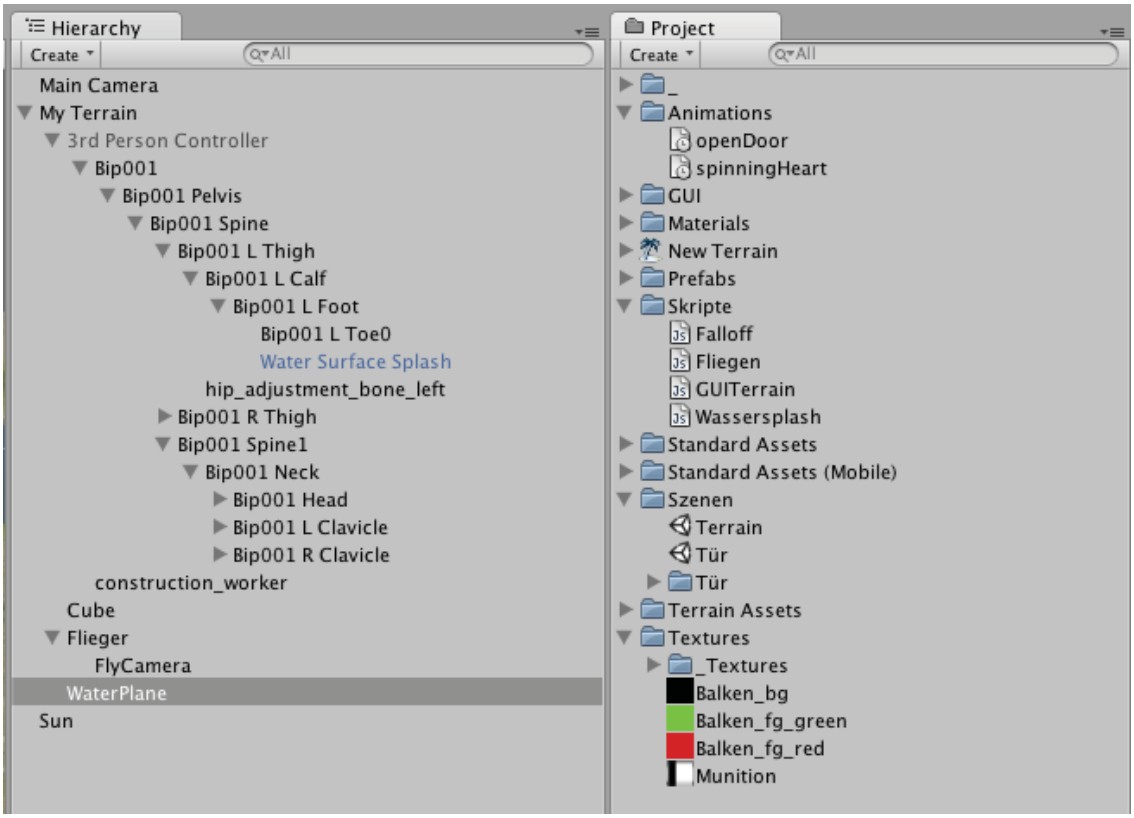


Abbildung 3.2.: Hierarchy View der Szene „Terrain“ und Project View.

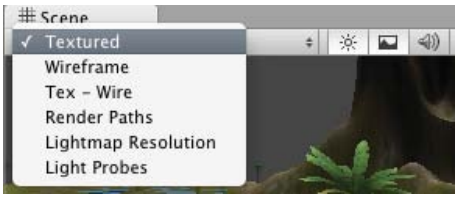


Abbildung 3.3.: Auswahlmöglichkeiten für den Draw Mode in der Scene View.

In der Szene View werden alle Objekte, die sich in der Szene befinden, grafisch dargestellt. Man kann Objekte platzieren, verschieben, skalieren und drehen. Oben in der Scene View befinden sich einige Kontrollelemente mit denen man verschiedene Ansichten generieren kann. So kann man beispielsweise die Szene nur als Meshes, ohne Texturen, anzeigen lassen oder die Texturen zusammen mit den Meshes. Falls man seiner Szene noch keine Lichter hinzugefügt hat, kann man sie hier auch erst einmal durch aktivieren des Lichtknopfes ausleuchten lassen.

Die Game View ist dazu da, das momentane Aussehen des fertigen Spiels zu simulieren. Dazu ist mindestens eine Kamera nötig, da das Spiel aus deren Sicht gerendert wird. Man kann in dieser View nichts bewegen, dies muss immer in der Scene View oder über den

Inspector geschehen.

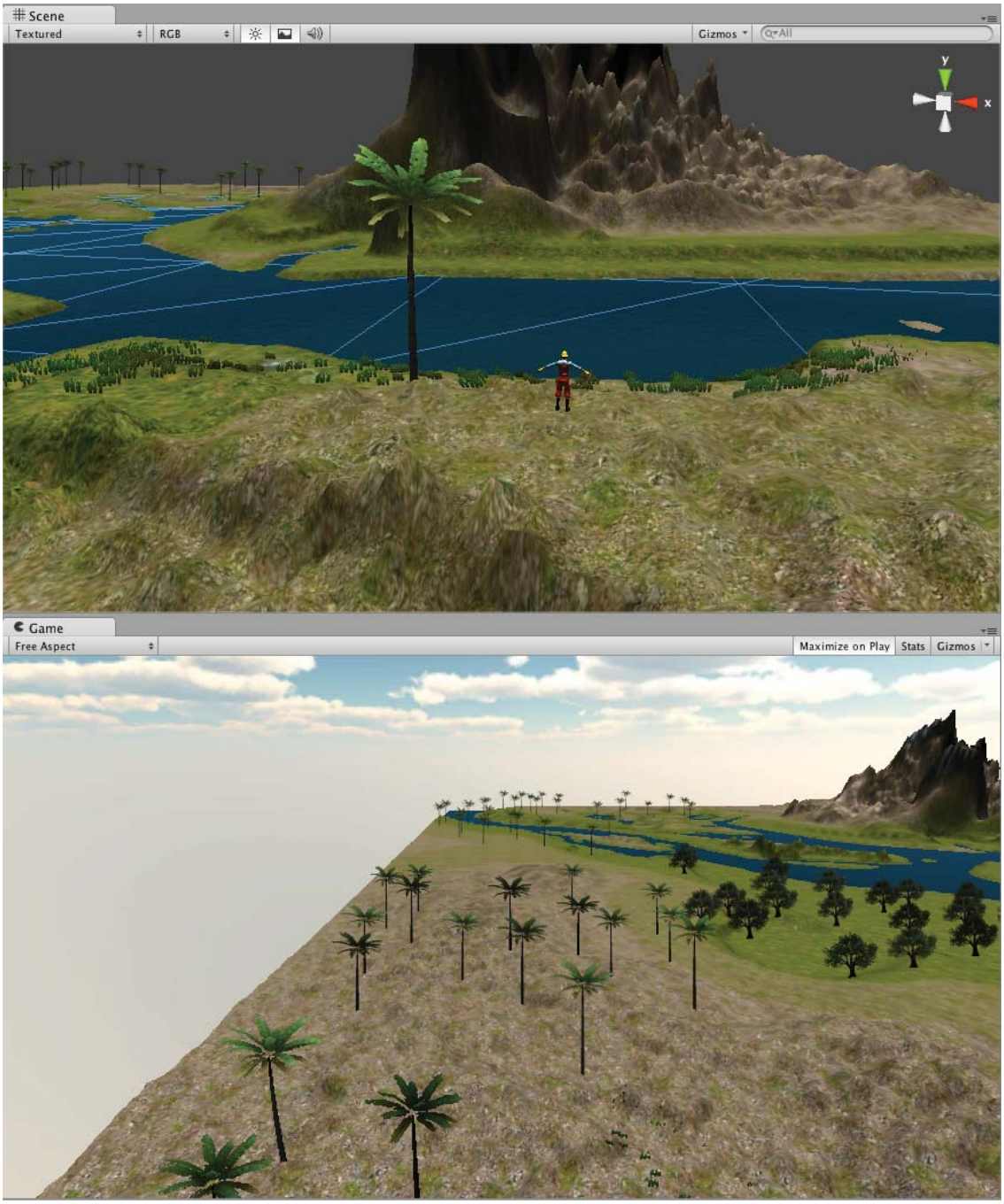


Abbildung 3.4.: Scene View und Game View der Szene „Terrain“.

Wird ein Objekt in der Hierarchy oder Scene View ausgewählt, zeigt der Inspector alle möglichen Informationen über dieses an.

In der Animation View können, wie in Kapitel 2.5.5 beschrieben, Animationen erstellt oder abgespielt werden. Die Animation des Objekts ist dann in der Scene View zu erkennen, deshalb sollte man die Fenster, während man eine Animation erstellt, so anordnen, dass beide zu sehen sind und nicht hintereinander liegen.

Um ein Spiel während der Entwicklung in Unity zu testen, betätigt man den Play-Button in der Toolbar. Unity wechselt zur Game View und simuliert den Spielablauf. Drückt man währenddessen den Pause-Button, hält das Spiel an und man kann Änderungen vornehmen, die jedoch temporär sind und rückgängig gemacht werden, sobald man den Spielmodus verlässt.

Abbildung 3.5.: Play/Pause Button.

In der Toolbar befinden sich auch Schalter für die Position und Ausrichtung der Achsen eines Objekts. Diese können sich entweder in der Mitte oder im Pivotpunkt des Objekts befinden. Hat das Objekt keine untergeordneten Objekte, befinden sich Mitte und Pivotpunkt in derselben Position. Wird Local für die Achsenrotation ausgewählt, sind die Achsen relativ zum Objekt ausgerichtet. Wird Global gewählt orientieren sich die Achsen an der Ausrichtung der Szene. Wurde das Objekt nicht gedreht, sind beide Ausrichtungen gleich.

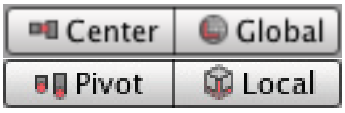


Abbildung 3.6.: Buttons für Position und Ausrichtung der Achsen eines Objektes.

3.2. Navigation

Um in der Scene View zu navigieren eignet sich am besten eine Zweitastenmaus mit Scrollrad. Mit der Maus alleine ist es möglich die Szene vor der Kamera zu bewegen oder die Kamera zu drehen. Mit Kamera ist hier nicht die Kamera in der Szene gemeint, von der aus die Spielansicht gerendert wird, sondern der Punkt von dem man momentan auf die aktuelle Szene schaut. Nimmt man zusätzlich die Tastatur zur Hilfe, bewegt man nicht die Szene, sondern die Kamera. Im Endeffekt kann man auch ohne Tastatur die gewünscht Kameraposition erreichen, beim geübten Umgang mit der Szenennavigation ist man mit Tastatur jedoch schneller.

	Maus	Tastatur	Toolbar
Szene vor der Kamera bewegen			
Kamera bewegen			
Kamera drehen (um sich selbst)			
Kamera drehen und bewegen (um einen Pivotpunkt)			
Zoomen			
Zoomen			

Ein weiterer wichtiger Punkt bei der Navigation ist das Finden von Objekten, da diese manchmal außer Sichtweite sind. Dazu markiert man das gesuchte Objekt in der Hierarchy View, bewegt den Mauszeiger in die Scene View und drückt dann F auf der Tastatur. Das gesuchte Objekt wird dann zentriert. Dies ist ebenfalls per Doppelklick auf das Objekt in der Hierarchy View möglich.

Abbildung 3.7.: Scene Gizmo.

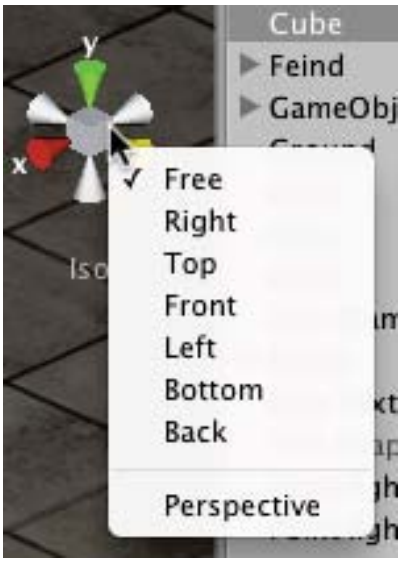


Abbildung 3.8.: Scene Gizmo mit verschiedenen Ansichtsmöglichkeiten.

In der rechten oberen Ecke der Scene View befindet sich das sogenannte Scene Gizmo. Es zeigt die momentane Orientierung der Kamera an. Klickt man einen der Arme an, wechselt die Kamera in den isometrischen Modus und man kann die gesamte Szene von oben, unten, vorne, hinten und von den Seiten betrachten. Um wieder in den perspektivischen Modus zu wechseln, klickt man auf das Quadrat in der Mitte des Gizmos. Man kann alle Ansichten auch direkt über das Klicken mit der rechten Maustaste auf das Scene Gizmo auswählen.

3.3. Objekte positionieren

Um Objekte in der Szene zu positionieren, kann man ihre Position, Drehung und Skalierung direkt in ihrer Transform-Komponente im Inspektor eingeben. Wenn man sich mit den genauen Werten nicht sicher ist, muss man dabei viel herum probieren. Intuitiver geht es in der Scene View, in der man einzelne Objekte direkt mit der Maus verschieben, drehen und skalieren kann.

Zum Verschieben eines Objekts klickt man dieses an, wobei in der Toolbar gleichzeitig der Verschiebe-Modus aktiviert wird. Klickt man auf einen der Zeiger und zieht die Maus, kann man das Objekt nur entlang der Achse dieses Zeigers bewegen. Wählt man beispielsweise den grünen Zeiger für die y-Achse, kann das Objekt nach oben und unten verschoben werden. Klickt man auf das Feld zwischen zwei Zeigern, kann man das Objekt in der entsprechenden Ebene bewegen. Hält man während des Verschiebens die Befehlstaste (OS X, für Windows: Ctrl) gedrückt, wird das Objekt immer um eine bestimmte Schrittweite verschoben,

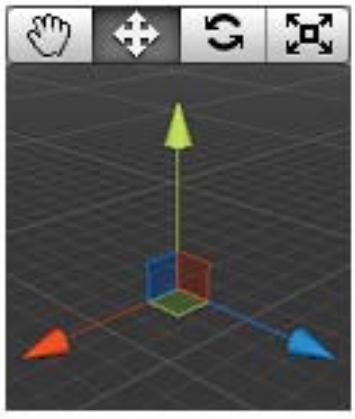


Abbildung 3.9.: Verschieben.

die festgelegt werden kann (Edit -> Snap Settings).

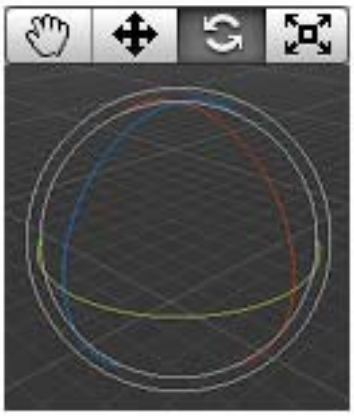


Abbildung 3.10.: Rotieren.

Um ein Objekt zu drehen, muss es aktiviert werden und der Rotationsmodus muss in der Toolbar ausgewählt werden. Die Zeiger sind jetzt ringförmig angeordnet und in der entsprechenden Farbe der Achse dargestellt, um welche das Objekt damit gedreht werden kann. Wie beim Verschieben wird, bei Anklicken eines Zeigers und ziehen der Maus, nur um die entsprechende Achse rotiert. Klickt man zwischen die Zeiger, kann man das Objekt gleichzeitig in alle Richtungen drehen.

Aktiviert man den Skalierungsmodus, kann man ein Objekt durch Ziehen an den Zeigern der entsprechenden Achse, entlang dieser vergrößern oder verkleinern. Durch Klicken auf den Würfel im Ursprung der drei Zeiger und Ziehen der Maus vergrößert bzw. verkleinert sich das Objekt in alle Richtungen und die Seitenverhältnisse bleiben proportional.

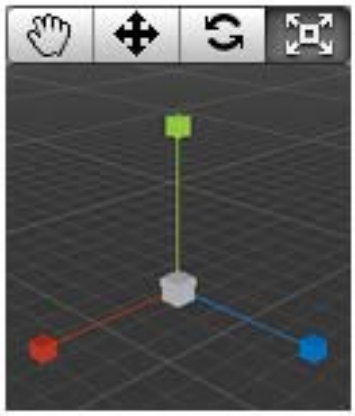


Abbildung 3.11.: Skalieren.

4. Das GameObject

4.1. Aufbau

Alle Objekte, die man in Unity seiner Spielszene hinzufügt, sind *GameObjects*. Diese beinhalten erstmal nichts als einen Namen, eine Position, Rotation und Größe. Erst durch das Hinzufügen von Komponenten bekommt das *GameObject* seine spezifischen Eigenschaften, es ist also eine Art Container. Der Inhalt dieses Containers ist jeweils im Inspector zu sehen.

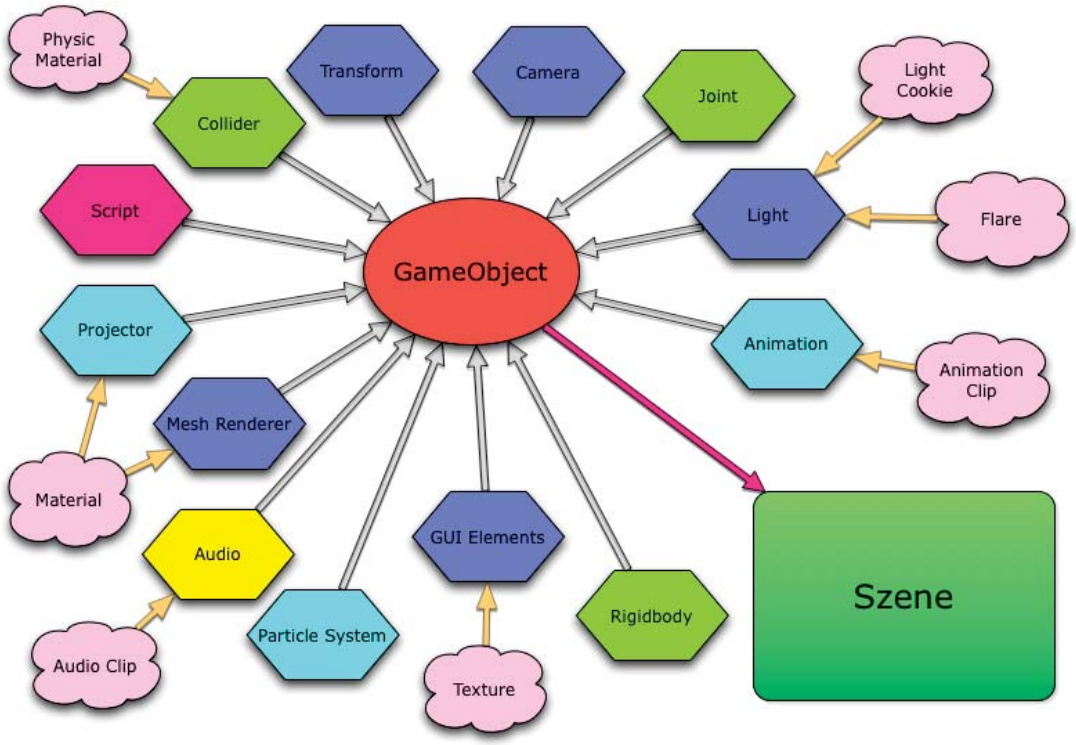


Abbildung 4.1.: GameObject mit möglichen Komponenten (nicht vollständig). Mesh- und Renderkomponenten (lila), physikalische Komponenten (hellgrün), Audiokomponenten (gelb), Skriptkomponente (pink), Effektkomponenten (hellblau). Assets, die zu Komponenten hinzugefügt werden können, um diese zu definieren (rosa).

Außerdem können GameObjects erst durch die einzelnen verschiedenen Komponenten überhaupt miteinander interagieren und kommunizieren, da man die Komponenten durch

Skripte, die ja ebenfalls Komponenten sind, ansprechen und verändern kann.

Unity stellt auch schon einige vorgefertigte GameObjects zur Verfügung. Diesen wurden schon eine oder mehrere Komponenten hinzugefügt. Ein Beispiel ist das Kamera-GameObject, dem unter anderen schon eine Kamera-Komponente hinzugefügt wurde. Man kann diesen vorgefertigten GameObjects, so wie auch den leeren, selbst noch Komponenten hinzufügen.

4.2. Komponentenarten

Die wichtigsten Gruppen dieser Komponenten sind Meshkomponenten, Renderkomponenten, physikalische Komponenten, Audiokomponenten, sowie Skripte.

Meshkomponenten sind dazu da, einem Objekt sein Aussehen zu geben. Die Komponenten enthalten Meshes und Materialien. Renderingkomponenten sind Werkzeuge um die Szene darzustellen, also Kameras, Licht, Skyboxes etc. Eine Skybox sollte immer zum gleichen GameObject gehören wie die Kamera, da sie sonst nicht zu sehen ist.

Um physikalische Interaktionen zwischen Objekten zu ermöglichen nutzt man die physikalischen Komponenten. Durch eine Collider-Komponente wird ein Objekt undurchdringlich und für andere Objekte wahrnehmbar. Falls der Collider nur als Trigger fungieren soll, ist das Objekt trotzdem durchdringlich, sendet aber Nachrichten aus, wenn ein anderes Objekt eindringt, sich innerhalb befindet oder es verlässt. Man muss bei Collidern beachten, dass diese nur einseitig funktionieren. Befindet sich der Spieler beispielsweise innerhalb eines Würfel-Colliders, kann er hinausgehen, jedoch nicht hinein. Soll ein Objekt massiv werden und sollen physikalische Kräfte darauf angewendet werden, fügt man die Komponente *Rigidbody* dazu. Dadurch können Kräfte auf das Objekt wirken. Positioniert man ein GameObject mit einer Rigidbody-Komponente beispielsweise irgendwo in der Luft, wirkt während des Spielverlaufs Schwerkraft darauf und es fällt nach unten, bis es auf einen Collider trifft. Außerdem kann man auf das Objekt via Skript Kräfte wirken lassen und es somit bewegen.

Eine Alternative zur Rigidbody-Komponente ist der *Character Controller*. Für viele Spielfiguren würde ein Rigidbody „zu echt“ wirken, wodurch das gewünscht Ergebnis nicht erzielt werden könnte. Wären zum Beispiel die Füße der Spielfigur zu klein, würde sie durch die Schwerkraft, die bei Bewegungen auf sie wirkt, oft hinfallen. Der Character Controller ist ein einfacher Collider, der um einige Funktionen ergänzt wurde. So kann man ihm via Skript mitteilen, dass er sich in bestimmte Richtungen bewegen oder andere

Rigidbodys verschieben soll [43].

Die letzten wichtigen physikalischen Komponenten sind die *Joints*, unsichtbare Verbindungen zwischen Objekten mit Rigidbody-Komponente. Diese Verbindung kann fest und unbeweglich oder elastisch sein und als Gelenk wirken.

Die beiden wichtigen Audio-Komponenten sind *Audio Source* und *Audio Listener*. Die Audio Source Komponente platziert man mit ihrem GameObject an die Stelle, wo Klänge erzeugt werden bzw. abgespielt werden sollen. Um diese Klänge überhaupt zu hören, benötigt man eine Audio Listener-Komponente. Während man sehr viele Audio Sources in seiner Szene haben kann, empfiehlt es sich, nur einen Audio Listener einzusetzen. Diese Komponente sollte an den Spieler gebunden sein, der sich durch die Szene bewegt oder besser noch an die Kamera, die ihn verfolgt.

Eine weitere, sehr vielseitige Komponente ist ein Skript. Es deckt alle Bereiche ab, die nicht von Unity in Form von Komponenten vorgegeben sind. Man kann mehrere davon an ein GameObject hängen.

4.3. Gruppierung

Zusätzlich zur Definition eines GameObjects durch Komponenten kann man mehrere GameObjects auch zu einem neuen gruppieren. So kann man beispielsweise eine Spielfigur aus Kopf, Rumpf, Armen und Beinen zusammensetzen und die einzelnen Objekte noch weiter unterteilen. Der Vorteil besteht darin, dass man die einzelnen Objekte unabhängig voneinander gestalten kann, wenn man das übergeordnete Objekt bewegt, rotiert und skaliert, sich aber alle untergeordneten Objekte mitbewegen und ihre Position im Gesamtobjekt beibehalten.

Um Objekte so zu gruppieren, zieht man in der Hierarchy View ein Objekt, welches gruppiert werden soll auf ein weiteres, welches dann das übergeordnete Objekt wird. Die Positionsangaben des untergeordneten Objektes ändern sich und werden relativ zum übergeordneten Objekt angegeben. Fügt man ein leeres GameObject zur Szene hinzu, in das man ein oder mehrere GameObjects gruppieren will, sollte das leere GameObject die Position (0,0,0) besitzen. Ist das nicht der Fall, ändern sich die Koordinaten des untergeordneten Objekts, was zu erheblichen Problemen beim weiteren Spielentwicklungsverlauf führen kann. Andererseits bietet es sich auch an, das neue GameObject auf die Koordinaten des zukünftigen untergeordneten Objekts zu setzen und die, des dann untergeordneten Objekts, in Abhängigkeit auf (0,0,0).

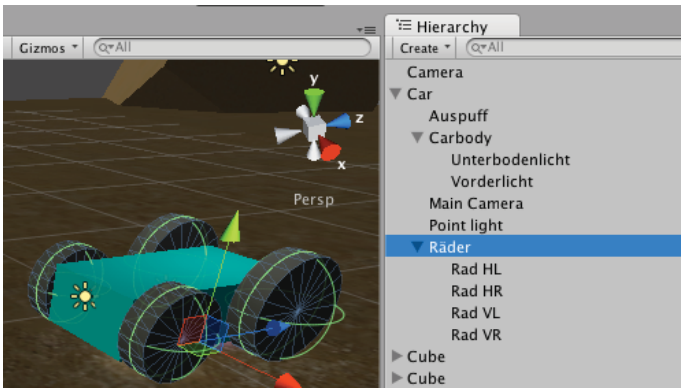


Abbildung 4.2: Beispiel eines primitiven Fahrzeugs. Das GameObject Car hat mehrere untergeordnete Objekte, wie Auspuff und Räder. Bewegt man „Car“, bewegen sich alle „Autoteile“ mit. Markiert man das GameObject Räder, werden alle vier einzelnen Räder markiert und man könnte beispielsweise alle proportional vergrößern oder verkleinern. Dadurch, dass die Räder einzelne Objekte sind, hätte man ihnen auch verschiedene Farben geben können.

4.4. Prefabs

Ein Prefab ist ein wiederverwendbares GameObject, welches im Projekt gespeichert ist. Ein einfaches GameObject ist nur in der Szene zu finden, also in der Scene und Hierarchy View, wohingegen ein Prefab im Projekt abgespeichert ist und somit in der Project View erscheint und außerdem von außen zugänglich ist. Ein Prefab zu konstruieren bietet sich an, wenn man ein GameObject in einer Szene mehrfach benutzen will. Man könnte dies ebenso durch Copy&Paste erreichen, was aber von Nachteil ist, wenn man nachträglich noch Änderungen am Objekt vornimmt. Erstellt man aus einem GameObject ein Prefab und fügt es anschließend mehrmals in die Szene ein, braucht man nur an einer Instanz dieses Prefabs Änderungen vornehmen und kann diese dann auf alle anderen Instanzen übertragen. Bei einem durch Copy&Paste eingefügten GameObject würden die Änderungen nur an diesem erfolgen und wären nicht auf die anderen Objekte übertragbar.

Der weitere Vorteil eines Prefabs ist, dass sich keine Instanz davon in der Szene befinden muss, man dieses Objekt aber per Skript während des Spiels instanziieren kann. GameObjects alleine können nicht instanziiert werden, da diese ja nicht explizit im Projekt abgespeichert sind. Als gutes Beispiel für notwendige Prefabs dienen Explosionen oder Kanonenkugeln, aber auch Spielfiguren, da man diese in jedem Level neu verwendet.

Um ein eigenes Prefab zu erstellen wählt man im Menü Assets -> Create -> Prefab. Dieses erscheint in der Project View, man kann es benennen und anschließend mit einem GameObject verknüpfen, indem man dieses aus der Hierarchy View auf das Prefab zieht.

5. Eigene Objekte

Da die Assets und 3D Objekte in Unity ja relativ beschränkt sind, gibt es natürlich die Möglichkeit eigene Assets, wie Meshes, Texturen, Materialien, Animationen, Skripte, Sounds und Kombinationen daraus, einzufügen. Dies geschieht sehr einfach per Drag&Drop in den Projektordner oder über die Funktion Assets -> Import New Asset.

5.1. Texturen

Um Texturen in Unity einzufügen, kann fast jedes beliebige Bildformat, wie JPEG, PNG, GIF, BMP, TGA, IFF, PICT und mehr, verwendet werden. Als besonderes Feature können Photoshop .psd und .tiff Dateien, die verschiedene Ebenen (Layers) beinhalten, direkt eingefügt werden. Dabei werden die Ebenen in Unity auf eine Ebene reduziert, die Originaldatei bleibt aber unverändert und kann in Photoshop weiter bearbeitet werden.

Die Seitenlängen der Textur sollten immer eine Zweierpotenz sein, also 64x64 oder 128x256. Wählt man eine Textur in der Project View aus, erscheinen ihre Importeinstellungen im Inspector. Hier kann man unter anderem den Typ der Textur auswählen, wie beispielsweise Normal Map. Dabei wird aus einer schwarz/weiß Textur eine Normal Map generiert. Zusammen mit der einfachen dazugehörigen farbigen Textur kann man daraus ein sehr plastisch wirkendes Material erstellen, z.B. für eine Steinmauer.

Weitere Typen sind GUI, Reflection (zum Erstellen einer Cube Map), Cookie (für Licht Cookies) und Advanced. Wählt man letzteren Typ, kann man viele spezifische Einstellungen für seine Textur vornehmen, wie das Generieren einer passenden Normal Map, Mip Map und Cubemap.

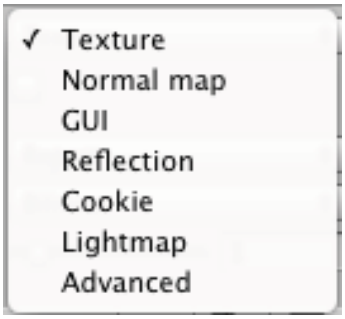


Abbildung 5.1.: Auswahlmöglichkeiten für Texturtyp.

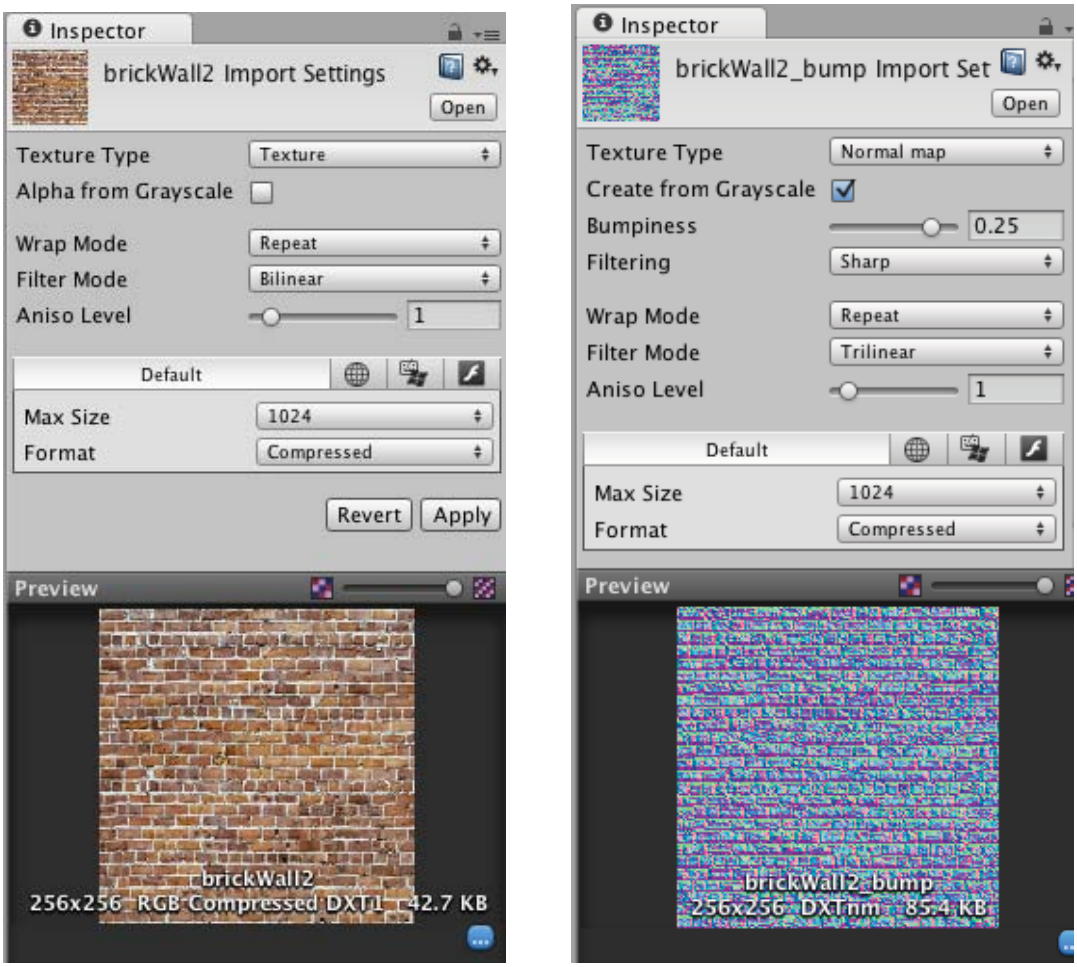


Abbildung 5.2.: Importeinstellungen für Texturen. Links: Texturtyp ist eine einfache Textur, rechts: Texturtyp ist Normal Map. Diese wird aus der Grausstufenabbildung der Originaltextur generiert.

5.2. 3D Objekte

Unity kann 3D Objekte aus fast allen 3D Programmen importieren. Dabei werden folgende Objekte mit importiert: alle Knoten mit ihrer Position, Rotation und Skalierung, Pivot-Punkte, Namen, Meshes, Normalen, Materialien mit Texturen und Animationen. Die folgende Tabelle zeigt, was aus welchem Programm importiert werden kann:

Werden die 3D Objekte nachträglich noch in Maya, 3ds Max, Cinema 4D, Cheetah3D oder Blender bearbeitet und anschließend gespeichert, werden alle Änderungen auch gleichzeitig in Unity vollzogen.

3D Package Support

	Meshes	Textures	Anims	Bones
Maya .mb & .ma[1]	✓	✓	✓	✓
3D Studio Max .max[1]	✓	✓	✓	✓
Cheetah 3D .jas[1]	✓	✓	✓	✓
Cinema 4D .c4d[1 3]	✓	✓	✓	✓
Blender .blend[1]	✓	✓	✓	✓
modo .lxo[2]	✓	✓	✓	
Autodesk FBX	✓	✓	✓	✓
COLLADA	✓	✓	✓	✓
Carrara[1]	✓	✓	✓	✓
Lightwave[1]	✓	✓	✓	✓
XSI 5.x[1]	✓	✓	✓	✓
SketchUp Pro[1]	✓	✓		
Wings 3D[1]	✓	✓		
3D Studio .3ds	✓			
Wavefront .obj	✓			
Drawing Interchange Files .dxf	✓			

[1] Import uses the application's FBX exporter. Unity then reads the FBX file.
[2] Import uses the application's COLLADA exporter. Unity then reads the COLLADA file.
[3] Cinema4D 10 has a buggy FBX exporter. Please see here for workarounds.

Abbildung 5.3.: Importmöglichkeiten für 3D Objekte [50].

Sobald man ein importiertes Mesh in der Project View auswählt, erscheint der FBXImporter im Inspector. Hier kann man alle möglich Einstellung für den Import des Meshes vornehmen. Falls es sich nicht anbietet für sein Objekt einen primitiven Collider zu verwenden, kann man hier durch ein Häkchen bei „Generate Colliders“ gleich einen komplexeren Collider in der Form seines Meshes generieren lassen. Dies ist allerdings nur für feste Objekte möglich. Bei Objekten, die bewegt werden sollen, wie Autos oder Spieler, muss immer ein primitiver Collider benutzt werden [44].

Es ist möglich Normal- und Lightmaps mit dem Mesh zu importieren, aber auch diese durch Unity berechnen zu lassen. Dazu setzt man Häkchen bei „Generate Lightmap UVs“ bzw. „Normals Calculate“. Die Normalen werden dann anhand des *Smoothing Angle* berechnet.

Da man in seinem 3D Programm oftmals das komplette 3D Objekt, inklusive Texturen, erstellt, bietet Unity die Möglichkeit diese gleich mit zu importieren, oder aber auch bewusst wegzulassen. Falls man sie importieren möchte, setzt man ein Häkchen bei „Import Materials“, woraufhin Unity die Textur in ein Material umwandelt und ebenfalls im Projekt speichert. Neben den Texturen können auch vorgefertigte Animationen mit importiert werden. Um zu verhindern, dass diese die Spielperformance verringern, sollte beim Import immer angegeben werden, dass sie komprimiert werden, was Unity dann zuverlässig erledigt [44].

Mitunter kann es Probleme mit der Rotation der importierten Modelle geben. Während in Unity die y-Achse nach oben zeigt, ist dies in manchen 3D Programmen die z-Achse. Für die Ansicht im Spiel ist das kein Problem, da die Rotation automatisch angepasst wird, das heißt das Objekt liegt nicht etwa auf der Seite, nur die Rotation liegt bereits bei 90 Grad um die x-Achse. Ein Problem entsteht erst wenn man über ein Skript auf das Objekt zugreift, da man beim Zugriff auf die Achsen ständig umdenken muss. Es empfiehlt sich daher das 3D Objekt in einem weiteren leeren GameObject, welches sich genau in der Mitte des 3D Objekts befindet, abzulegen. Dieses hat nun die richtige Orientierung und kann im Skript angesprochen werden. Außerdem sollte man in seinem 3D Programm immer darauf achten, sein Objekt im Koordinatenursprung zu modellieren, um es in Unity leichter platzieren zu können.

Abbildung 5.4.: Importeinstellungen für ein 3D Objekt im Inspector. Die Materialien und Animationen werden mit importiert, außerdem werden Collider für das Objekt generiert.

6. Szenenaufbau

6.1. Szene

Um ein Spiel zu erstellen, beginnt man als erstes ein neues Projekt. In dieses Projekt werden gleich einige Standard Assets von Unity geladen, die man sich nach Belieben auswählen kann. Anschließend erzeugt man eine neue Szene über File -> New Scene, welche dann im Projekt unter einem gewählten Namen abgespeichert wird. Eine Szene in Unity könnte man als ein Level betrachten, die Menge aller Szenen ergibt dann das Spiel. Eine Szene kann aber auch nur ein Startbildschirm sein oder eine Zwischensequenz.

6.2. Kamera

In jeder neuen Szene befindet sich schon ein GameObject: die Main Camera. Diese enthält die Komponenten Camera, GUILayer, Flare Layer und den Audio Listener. Durch den GUILayer können 2D GUI Elemente gerendert werden, die sich aus jeder Kameraansicht an der gleichen Stelle des Bildschirms befinden. Wenn man nun in der Szenenansicht beginnt das Spiel zu modellieren, kann man dem entsprechend die Main Camera ausrichten. Wählt man eine Kamera aus, so wird in der Ecke der Scene View eine Kamera Vorschau gezeigt.

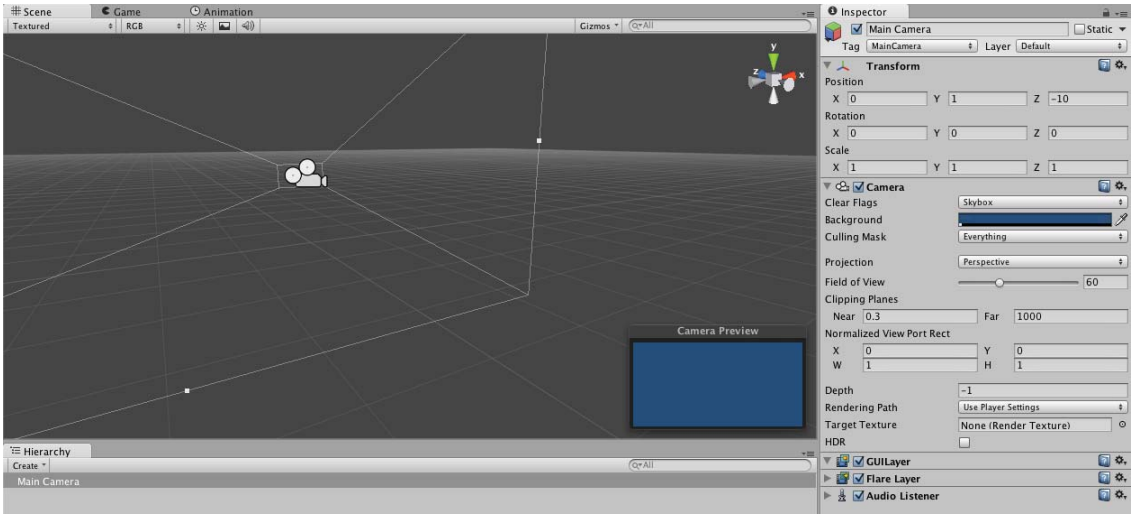


Abbildung 6.1.: Eine Main Camera in der Scene und Hierarchy View, sowie ihre Komponenten im Inspector.

Das Spiel wird immer aus Sicht der Kamera gerendert, deren Tiefe am geringsten ist. Man kann dies über „Depth“ im Inspector einstellen. Außerdem kann man dort einstellen, wie groß das Kamerabild im Verhältnis zum ganzen Bild sein und wo auf dem Bildschirm es dargestellt werden soll (via „Normalized View Port Rect“). Auf diesem Weg kann man einen Split Screen erstellen, der einmal zeigt was sich vor dem Spieler befindet und auf einem kleineren Bild, was sich hinter ihm befindet. Man sollte darauf achten, die Audio Listener Komponente bei weiteren Kameras zu entfernen, da es sonst zu Fehlern führt.

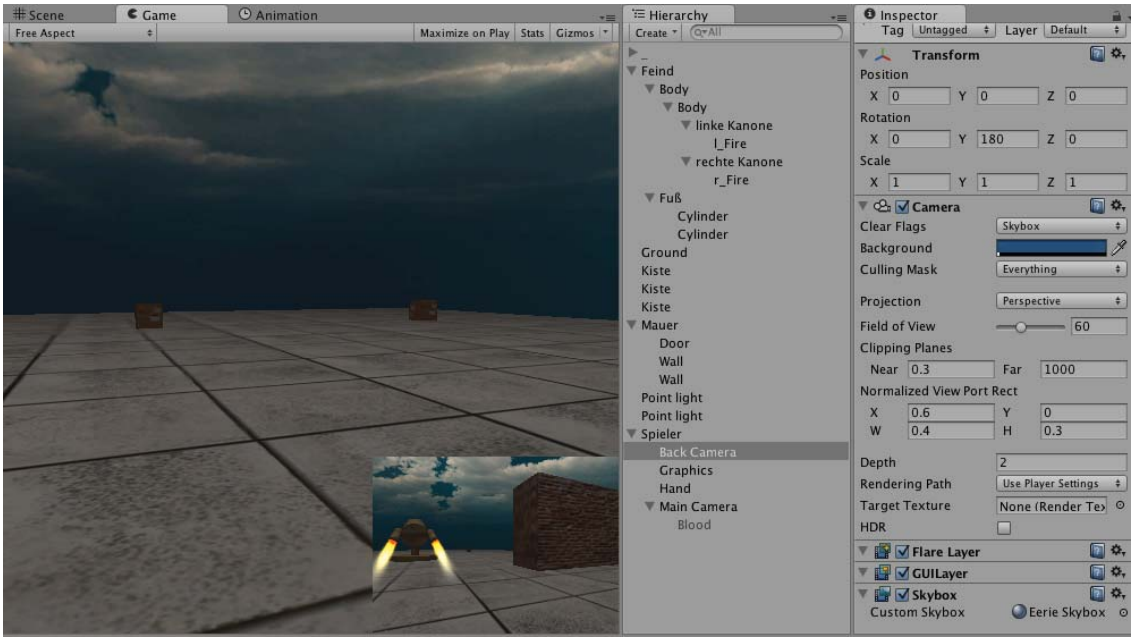


Abbildung 6.2.: Main Camera und Back Camera in der Game View. Das große Bild zeigt die Sicht nach vorne, das kleine Bild in der unteren rechten Ecke zeigt was sich hinter dem Spieler befindet. Beide Kameras besitzen die gleiche Skybox Komponente. Im Inspector sieht man die Eigenschaften der Back Camera. Die Werte für X und Y in der Kamera-Komponente geben die Position des Kamerabilds an, W und H die Größe in Relation zur Bildschirmgröße.

Möchte man ein 2D- oder ein isometrisches Spiel mit Unity kreieren, welches nicht perspektivisch erscheint, gibt es die Möglichkeit, die Kameraprojektion auf orthografisch umzuschalten.

Eine Kamera rendert die Sicht des Spielers, wie in einem First Person Shooter, wenn diese dem Spieler in der Hierarchie untergeordnet und schließlich auf Augenhöhe platziert wird. Soll die Kamera dem Spieler folgen, wie etwa in einem (Auto-) Rennspiel, kann sie ebenfalls mit dem zu verfolgenden Objekt gruppiert werden. Eine schönere Lösung für

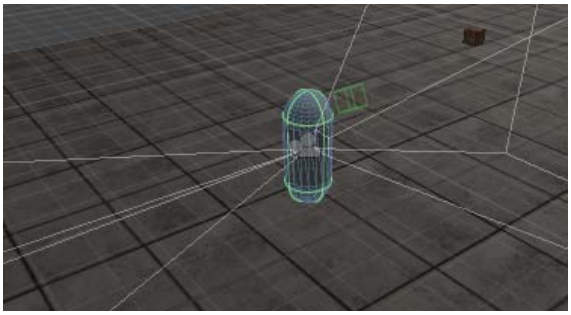


Abbildung 6.3: First Person Controller. Die Kameras nach vorn und hinten befinden sich innerhalb des Spielers, weshalb man ihn nicht sieht.

diesen Fall ist es, ein Skript für die Kamera anzuwenden. Einige Kamera-Skripte, wie das „Smooth Follow“ Skript, welches erwirkt, dass ein Objekt einem Zielobjekt folgt, werden schon von Unity mitgeliefert. Für Puzzle-, Logik- oder sonstige Spiele, bei denen man die gesamte Spielfläche abbilden muss, platziert man die Kamera statisch.

6.3. Grundfläche

Da man Dank der Kamera nun seine Spielumgebung sehen kann, kann man beginnen diese aufzubauen. Will man kein Puzzlespiel oder ähnliches mit fester Kamera bauen, hat man in der Regel vor, ein Spiel mit physikalischen Interaktionen und einer Spielfigur zu entwickeln. Dazu benötigt man eine Grundfläche, auf der das ganze stattfindet. Man hat sehr viele Möglichkeiten eine Grundfläche für sein Spiel auszuwählen. Das Wichtigste ist jedoch immer, dass diese eine Collider-Komponente enthält. Wenn man vergisst seiner Grundfläche diese hinzuzufügen, werden alle Objekte mit Rigidbody-Komponente einfach hindurch fallen.

Die einfachste Art einer Grundfläche, ist eine der in Unity vorhandenen primitiven GameObjects, die Fläche (Plane). Man kann jedoch auch die anderen primitiven GameObjects benutzen und so beispielsweise aus mehreren Würfeln eine unebene Fläche zusammenstellen oder diese als einzelne Plattformen anordnen.

Des Weiteren kann man natürlich seine eigenen Objekte benutzen, welche man in einem 3D Programm erstellt und in Unity importiert hat. Das können ein kompletter Raum, eine Landschaft oder auch nur einzelne Plattformen sein. Beim Import sollte man dann beachten, gleich Collider für die Objekte zu erstellen. Alternativ kann man auch passende primitive Collider aus Unity auswählen und diese zu den Objekten hinzufügen. Will man ein Spiel entwickeln, was in einer Landschaft spielt, bietet es sich an die Terrain Engine von Unity zu nutzen, da dieses sehr effektiv ist.

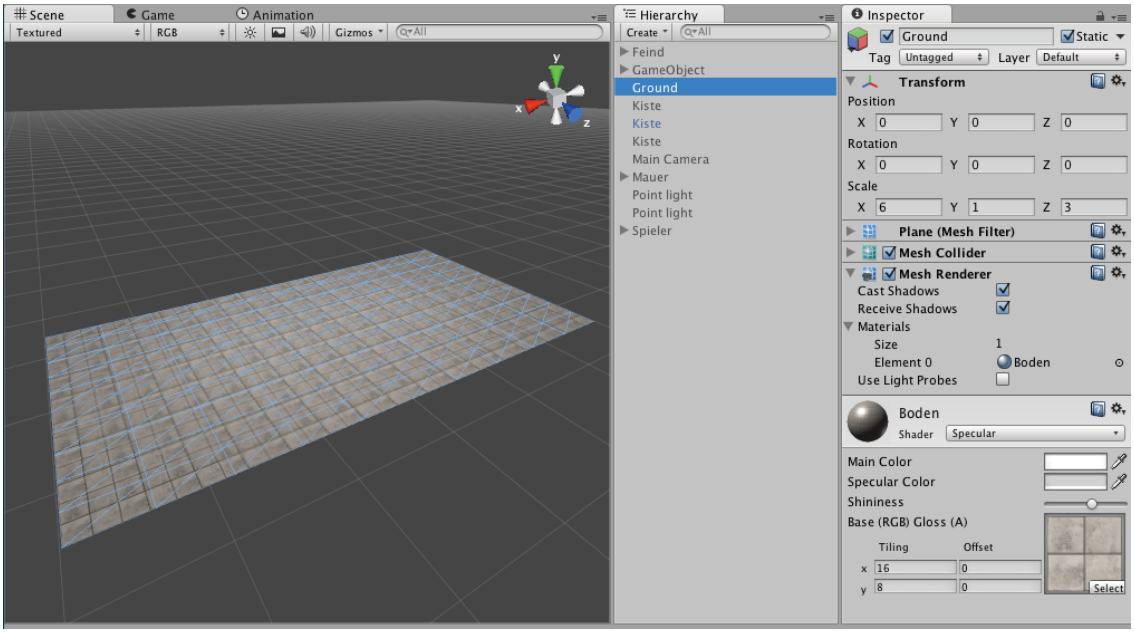


Abbildung 6.4.: Eine Grundfläche, bestehend aus der Form „Plane“. Dem GameObject wurde neben dem Mesh Renderer, der die Fläche sichtbar macht und dem Mesh Filter, der die Form beinhaltet, eine Mesh Collider Komponente hinzugefügt, dass Objekte nicht hindurchfallen. Dem Renderer wurde ein Material hinzugefügt, welches einen einfachen Shader besitzt, so dass diesem nur eine Textur hinzugefügt werden musste.

6.4. Licht

Um die Spielfläche in der Game View nun überhaupt zu sehen, sollte man ein paar Hauptlichter einsetzen. Diese kann man im Spielentwicklungsverlauf dann immer weiter anpassen. In der Scene View wird die gesamte Szene, wie in der Abbildung 6.4 zu sehen, automatisch ausgeleuchtet, was man mit Aktivieren des Lichtknopfes ausschalten kann. Lichter findet man, schon von Unity vorgefertigt, im GameObjects-Menü. Man kann aber auch jedem GameObject eine Lichtkomponente hinzufügen. So lässt sich beispielsweise eine Decken- oder Stehlampe realisieren. Wenn ein Licht einem Objekt folgen soll, sollte man das GameObject mit der Lichtkomponente dem zu folgenden Objekt unterordnen.

Da Licht für viele verschiedene Dinge, zum Beispiel zur Simulation der Sonne oder von Lampen, bei Explosionen und weiteren, eingesetzt werden kann, muss es natürlich auch verschiedene Arten geben. In Unity stehen das Spotlight, das Point Light und ein gerichtetes Licht, das Directional Light zur Verfügung. Jedes Licht kann in seiner Intensität und Farbe variiert werden, um verschiedene Stimmungen zu erzeugen. Diese Veränderungen

können auch während des Spielverlaufs durch Skripte vorgenommen werden.

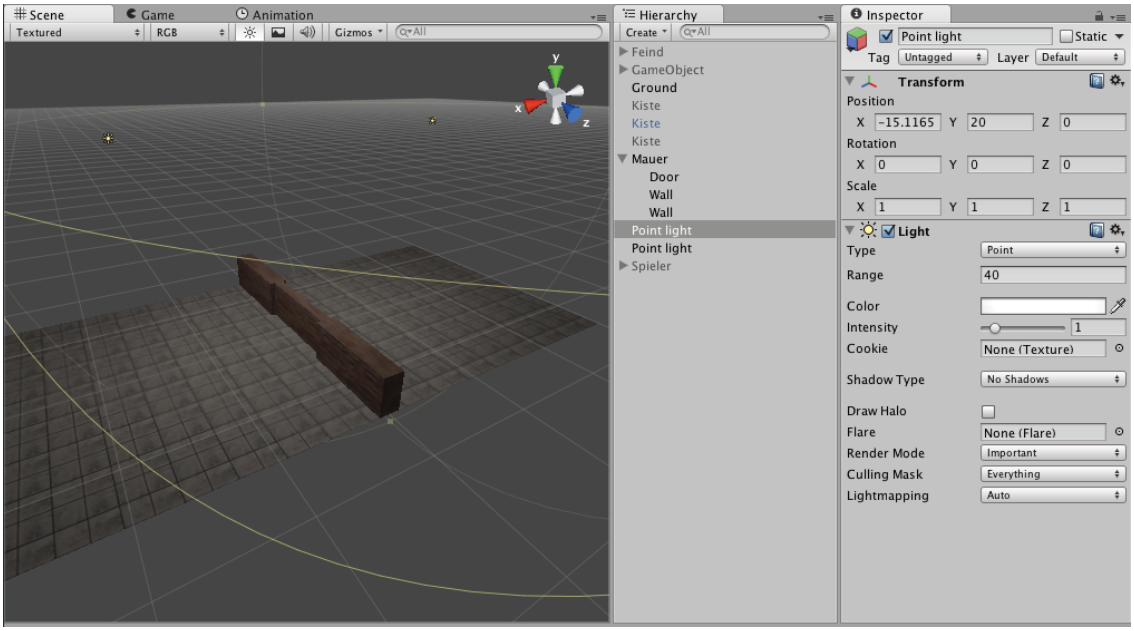


Abbildung 6.5.: In dieser Szene sind zwei Point Lights zu sehen, die den Boden, sowie eine schon hinzugefügte Mauer, beleuchten. Es wurde für jede Hälfte der Grundfläche ein Point Light eingesetzt. So wird die Mauer, welche aus drei Cubes besteht, von beiden Seiten beleuchtet.

Um mit einem einfachen Licht noch mehr Effekte zu erzielen, kam man Light Cookies einsetzen. Besonders gut eignet sich dies, wenn man simulieren möchte, dass ein Licht durch ein Fenster oder eine andere Öffnung in der Wand scheint. Unity ermöglicht ebenfalls den Einsatz und die Generierung von *Light Probes* und *Lightmaps*. Das Generieren von Schatten ist leider nur mit UnityPro möglich. Als Alternative kann man mit einen Projektor (Component -> Effects -> Projector) einen einfachen *Blob Shadow* erzeugen.

6.5. Hintergrund

Nachdem man seine Grundfläche nun ein wenig ausgeleuchtet hat, stellt sich die Frage nach dem Hintergrund. Hat man als Spielfläche einen geschlossenen Raum gewählt, erübrigt sich diese Frage. Falls dieser Raum aber aus mehreren Teilen zusammengesetzt wurde, ist es möglich, dass zwischen den einzelnen Teilen Nahtstellen entstanden sind. Unterscheidet sich die Hintergrundfarbe zu sehr von der Farbe des Raums, sind diese sichtbar. Man sollte dann die Farbe für den Hintergrund so wählen, dass sie der des Raums ähnelt.

Bei einer offenen Spielfläche kann man eine einzelne Farbe als Hintergrund wählen, vielfältiger ist allerdings die Skybox. Damit die Kamera eine Skybox rendern kann, benötigt sie eine Skybox-Komponente, zu welcher man dann ein entsprechendes Material hinzufügen kann. Dieses kann man importieren, aus den Unity Standard Assets auswählen oder selbst mit Unity erstellen. Dazu erzeugt man ein neues Material (Create -> Material) mit einem Skybox Shader (RenderFX -> Skybox). Dort kann man alle sechs Texturen für seine Skybox einsetzen.

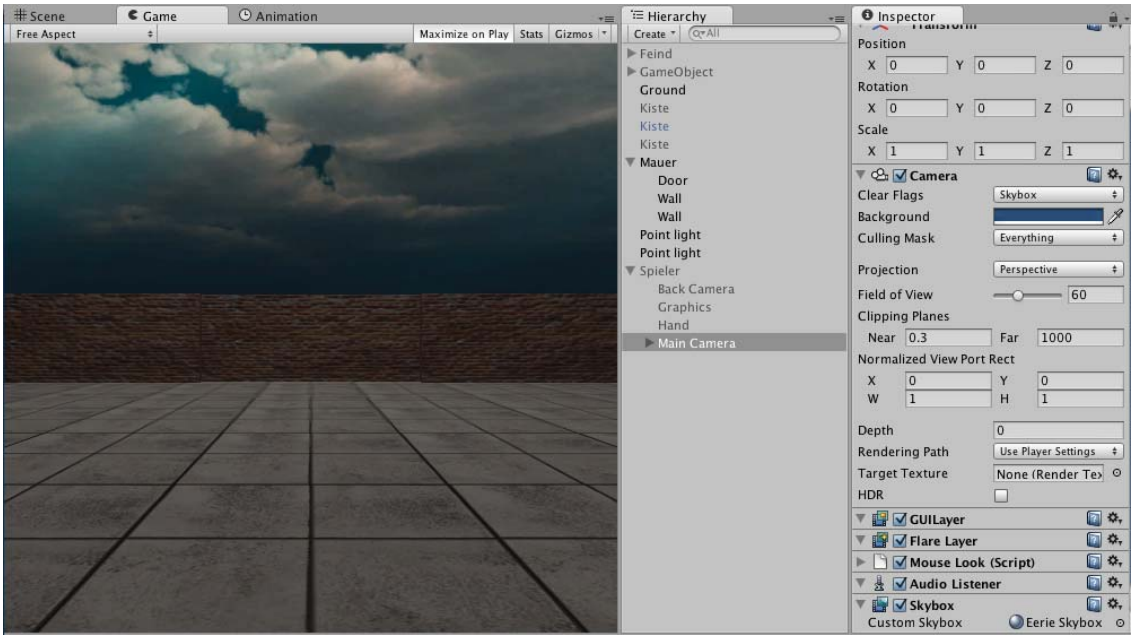


Abbildung 6.6.: Die Main Camera rendert die Szene aus der Sicht des Spielers. Eine Skybox Komponente wurde hinzugefügt. Deaktiviert man diese, wird der Hintergrund einfarbig, in der dafür in der Kamera-Komponente festgelegten Farbe, dargestellt.

6.6. Objekte

Da nun die Grundlagen geschaffen sind, um eine Szene zu gestalten, kann man beginnen, alle weiteren Spielobjekte in die Szene einzusetzen und zu positionieren. Dazu benutzt man vorher entwickelte Prefabs oder importierte Meshes, man kann aber auch noch neue Objekte mit Unity erstellen, sowie bereits vorhandene durch neue Komponenten erweitern. Um einem Mesh eine Farbe oder Textur zu geben, muss man ein Material erstellen, dem man entweder eine Farbe zuordnet oder eine Textur. Man kann die Farbe oder die Textur

nicht direkt auf das Mesh legen. Die Mesh Renderer Komponente jedes Objekts hat zwar erstmal ein Standardmaterial, wenn man dieses verändert, verändert es sich jedoch überall wo es eingesetzt wurde.

Für das Erstellen von Materialien gibt es in Unity viele vorgefertigte Shader. Falls der Richtige nicht zu finden ist, kann man mit dem eingebauten ShaderLab auch eigene entwickeln.

Abbildung 6.7: Ein Material für eine Steinmauer. Als Shader wurde Bumped Diffuse eingesetzt, dem eine einfache Textur und eine Normal Map hinzugefügt werden muss. Für dieses Material wurden die in Abbildung 5.2 gezeigten Texturen verwendet.

Zur Anzeige von Spielständen oder Ähnlichem, kann man GUI Objekte einsetzen, die durch die Kamera gerendert werden, wenn sie einen GUILayer besitzt. Zur Auswahl hat man GUI Texture und GUI Text, welche man beide nicht wie 3D Objekte in der Szene bewegen kann, sondern nur im zweidimensionalen Feld auf dem Bildschirm. Die GUI Objekte sind dadurch aus jeder Kameraperspektive immer an der selben Stelle auf dem Bildschirm zu sehen. Die Textur und der Text der Objekte kann, wie andere GameObjects auch, durch Skripte verändert werden. Man hat die Möglichkeit die GUI Objekte permanent im Bildschirm einzublenden, oder erst bei einem bestimmten Ereignis aufzurufen. Eine einfachere Variante für GUIs ist UnityGUI (siehe Kapitel 7.5), wobei GUI-Elemente komplett während der Laufzeit per Skript erzeugt werden und in der Scene und Hierarchy View nicht als Objekte auftauchen.

Möchte man einen Text im dreidimensionalen Raum darstellen, kann man dafür ein GameObject mit einer Text Mesh Komponente erstellen. Der Mesh Renderer muss dann ein Material mit einem Text Shader beinhalten, um den Text farbig darzustellen.

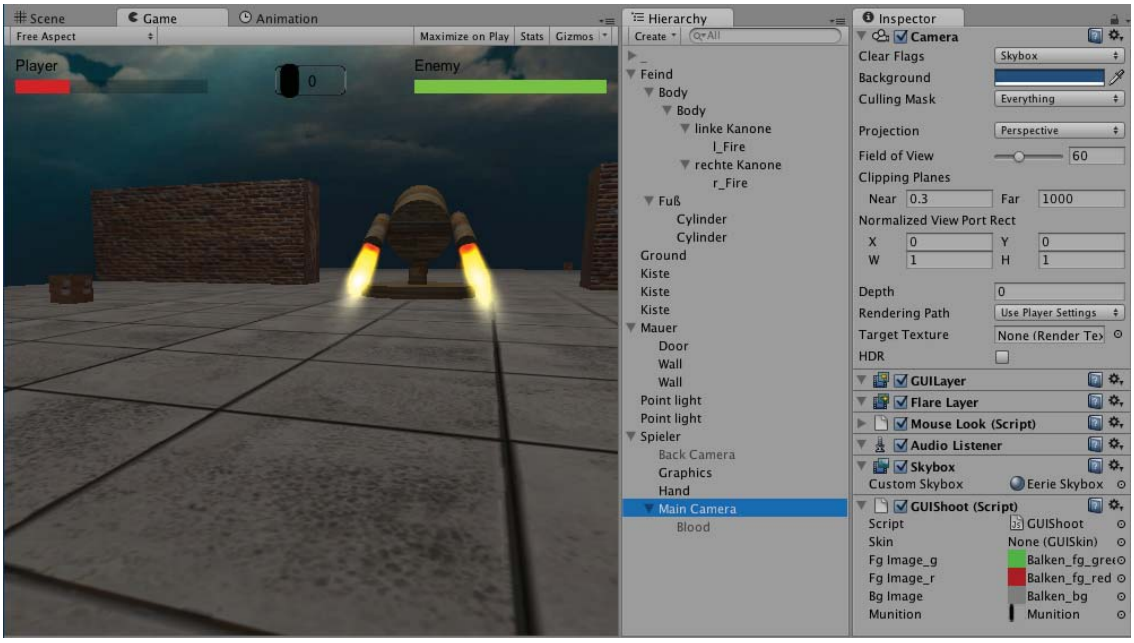


Abbildung 6.8.: Am oberen Rand der Game View sieht man die GUI Elemente, die während des Spiels gerendert werden. Ein Script, welche diese generiert (GUIShoot) wurde als Komponente zur Main Camera hinzugefügt. Die Variablen vom Typ Texture sieht man im Inspector und kann sie dort verändern.

6.7. Build Settings

Nachdem man seine Szene fertig erstellt und in Unity ausgiebig getestet hat, kann man sich nun ansehen, wie sie außerhalb, in einer eigenen Anwendung aussieht und funktioniert. Dazu öffnet man die Build Settings, wählt eine, mehrere oder alle Szenen aus und entscheidet für welche Plattform man sein Spiel entwickeln möchte. Nachdem man Speicherort und Namen für sein Spiel gewählt hat, wird es von Unity kompiliert und man kann es anschließend auf der entsprechenden Plattform testen.

Zusätzlich kann man in den PlayerSettings ein Icon, sowie andere Einstellungen, beispielsweise die Standardauflösung, für seine ausführbare Anwendung festlegen.

Man kann sein Spiel größtenteils ohne jegliche Veränderungen für Windows, OS X, Webplayer und Flash kompilieren lassen. Wählt man ein Smartphone Betriebssystem oder eine Konsole als Plattform, müssen die Eingabemöglichkeiten verändert werden, da dies hier nicht über Maus und Tastatur gehandhabt wird, sondern über Touchscreens und diverse Controller.

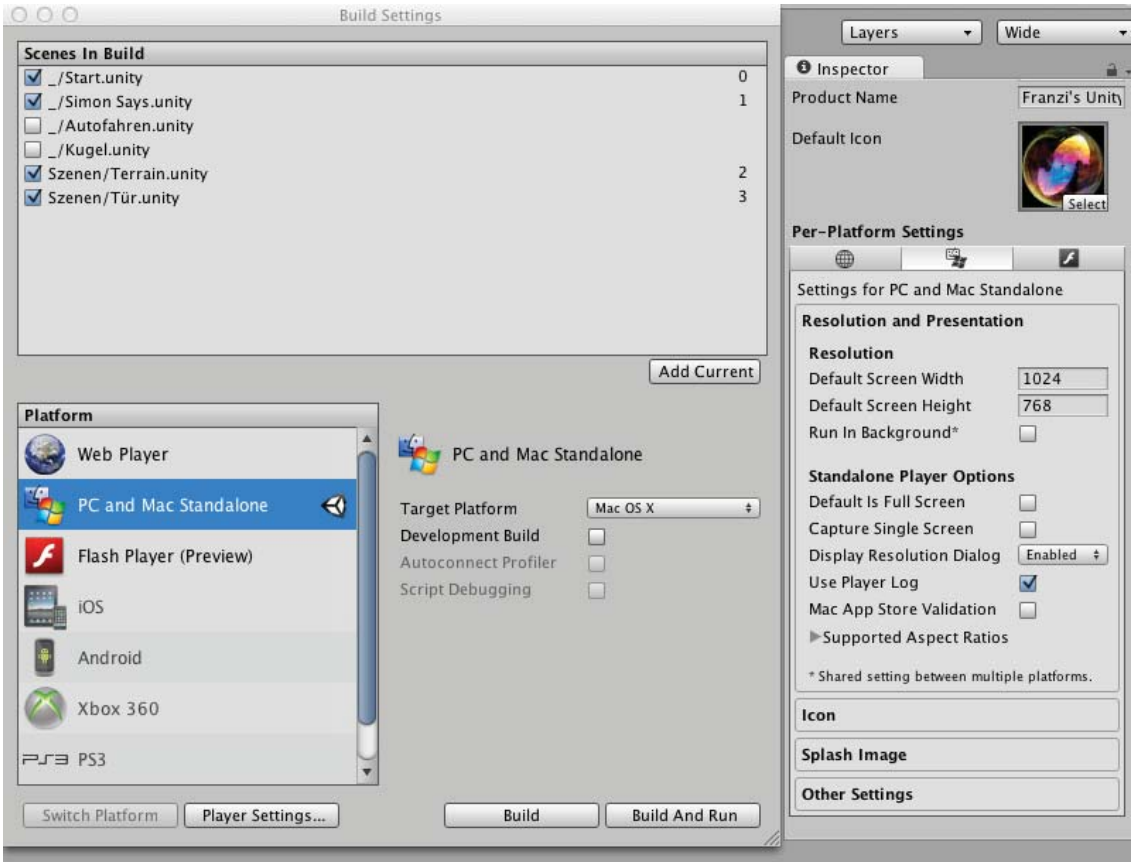


Abbildung 6.9.: Build Settings im eigenen Fenster und PlayerSettings im Inspector. Die Szenenreihenfolge ist veränderbar. Die Indizes der Szenen kann man auch nutzen, um diese in Skripten zu adressieren. Hier wurde in den PlayerSettings ein Icon, sowie eine Auflösung von 1024 x 768 festgelegt. Außerdem wurde angegeben, dass beim Öffnen der Anwendung ein Dialogfenster erscheint, in dem der Benutzer die Bildschirmauflösung selbst verändern kann.

7. Interaktion von Objekten

7.1. Skripte in Unity

Skripte in Unity sagen Objekten, wie sie sich in bestimmten Situationen verhalten sollen. Sie werden als Komponenten zu GameObjects hinzugefügt. Für Unity kann man, wie schon erwähnt, Skripte in C#, Boo und JavaScript verfassen, wobei sich die JavaScript-Variante Unity's etwas von der normalen unterscheidet. Von Vorteil ist, dass man alle Skriptarten in einem Projekt verwenden kann und man so eventuell vorgefertigte Skripte nicht umschreiben muss.

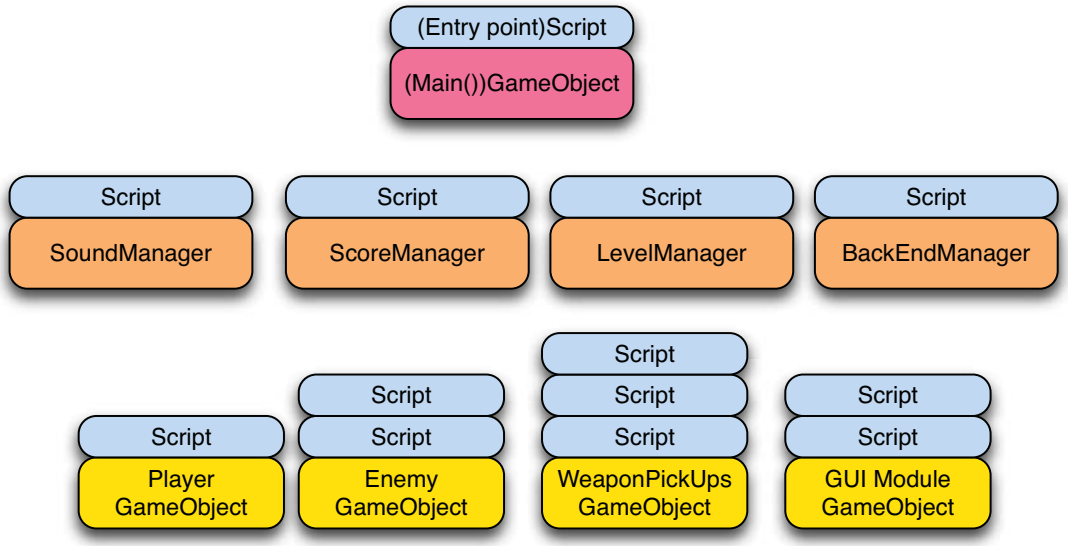


Abbildung 7.1.: Application Design Model. Es gibt ein Main()GameObject (pink), welches wie in einem C-Programm die Main() Funktion als Skript enthält. Außerdem gibt es Manager Objekte (orange), die Skripte für Sound, Score, Level und BackEnd enthalten. Diese werden für jedes neue Level übernommen. Man kann Manager Objekte für alle beliebigen Belange erstellen, aber auch ganz ohne arbeiten. In der unteren Ebene befinden sich die normalen GameObjects (gelb) mit ihren Skripten [7].

Da in der Szene fast jedem GameObject ein oder mehrere Skripte angehängt werden, kann man schon erahnen, dass dies mitunter recht unübersichtlich wird, wenn man ein großes Projekt gestaltet. Es ist daher hilfreich, wenn man sich für ein Spiel Manager

GameObjects erstellt, die man in jeder Szene wieder findet. Es gibt dazu eine Funktion aus der Klasse *Object*[30] namens *DontDestroyOnLoad()*[31], die verhindert, dass ein GameObject beim Laden einer neuen Szene zerstört wird, was standardmäßig der Fall ist.

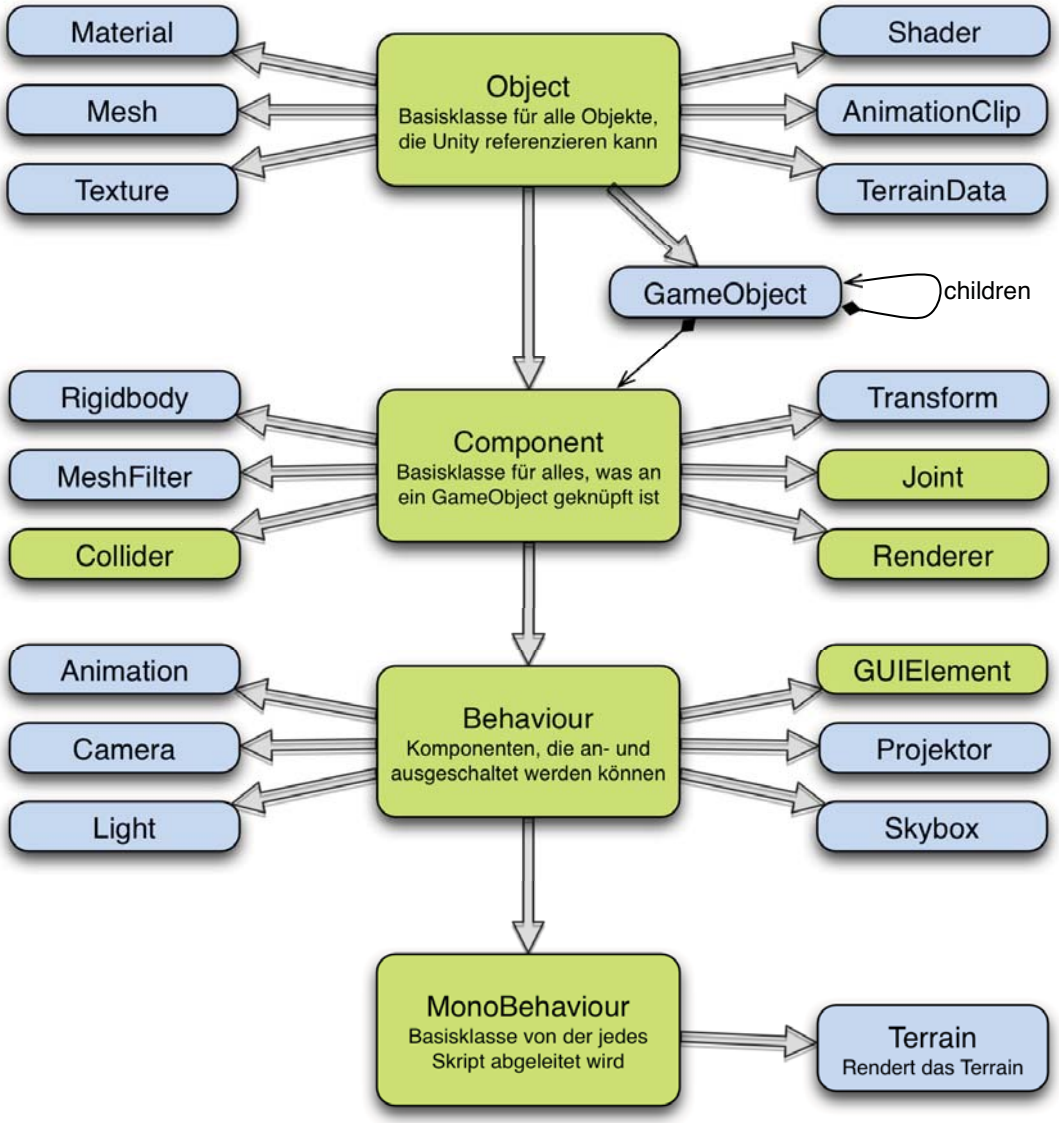


Abbildung 7.2.: Klassendiagram für die Klasse *Object* (nicht vollständig) mit Basisklassen (grün) und abgeleiteten Klassen (blau).

Unity stellt viele Klassen und Funktionen bereit, wie zum Beispiel die *MonoBehaviour*-

Klasse[23], von der jedes Skript abgeleitet wird. Sie beinhaltet die wichtigen Funktionen *Update()*[29], die vor jedem Frame aufgerufen wird, *Start()*[28], die nur einmal beim Laden des Objekts aufgerufen wird und *Reset()*[27]. Außerdem findet man hier die *Event Handler*, die automatisch bei bestimmten Ereignissen aufgerufen werden. Beispiele dafür sind: *OnCollisionEnter()*[24] (wird aufgerufen, wenn der Collider des Objekts auf einen anderen trifft), *OnMouseUp()*[26] (wird aufgerufen, wenn der Benutzer die Maustaste loslässt) und *OnGUI()*[25] (wird aufgerufen um GUI Elemente zu rendern).

7.2. Variablen im Inspector

Wenn man den Wert einer Variablen ändern möchte, müsste man theoretisch für jeden Versuch das Skript öffnen, den Wert ändern, neu kompilieren und die Szene neu spielen. Ist man sich mit dem Wert nicht sicher, dauert dies womöglich sehr lange, was sich durch Unity allerdings verhindern lässt.

Man kann Werte, für die man im Skript eine Variable deklariert hat, während des Spielmodus im Inspector ändern und somit den richtigen durch Testen ermitteln. Beendet man den Spielmodus, ist der im Inspector veränderte Wert wieder zurückgesetzt. Jedoch kann man nun den vorher ermittelten Wert in das Skript einsetzen.

Soll eine Variable nicht im Inspector erscheinen, muss sie als *private var* deklariert werden. Wird sie als *static var* deklariert, kann man von anderen Skripten auf diese Variable zugreifen und sie verändern, jedoch sieht man sie ebenfalls nicht im Inspector.

Abbildung 7.3.: Das GameObject „Body“ im Inspector. In der Skript-Komponente „Feind“ sieht man veränderbare Variablen.

7.3. Objekte adressieren

Skripte ermöglichen die Interaktionen von Objekten. Hierbei gibt es zwei Varianten: Zum einen will man bei bestimmten Bedingungen das Objekt verändern, dem auch das Skript zugehörig ist, zum anderen sollen Veränderungen an anderen GameObjects vorgenommen werden.

Für die erste Variante muss man die gewünschte, zu verändernde Komponente mit dem Skript erreichen. Viele der von Unity mitgelieferten Komponenten sind dazu bereits als Variablen hinterlegt und man kann diese direkt ansprechen und vordefinierte Funktionen aufrufen. Will man beispielsweise ein Objekt bewegen, falls ein bestimmtes Ereignis eintritt (z.B. wurde eine Pfeiltaste gedrückt), so muss man die Transform-Komponente, die jedes Objekt besitzt, ansprechen und ihre Bewegungsfunktion (*Translate()*) aufrufen.

```
function Update() {
  transform.Translate(0, 0, 2);
}
```

Skript 7.1: Objekt bewegen mit Translate

Hierbei ist darauf zu achten, dass die zugehörige Variable jeder Komponente klein geschrieben werden muss, da die gewünschte Funktion sonst nicht ausgeführt wird und man oft die Fehlermeldung „NullReferenceException: Object reference not set to an instance of an object“ bekommt. Bei Großschreibung versteht Unity die Komponente immer als eine Klasse oder einen Skriptnamen und nicht als Referenz auf eine Komponente. Erhält man oben genannte Fehlermeldung, hat man sich entweder vertippt oder man hat vergessen einer Variable, die als ein Objekt bzw. Komponente eines Objekts deklariert ist, ein entsprechendes GameObject bzw. Prefab zuzuordnen. Möchte man eine Komponente ansprechen, der keine Variable durch Unity zugeteilt ist, wie etwa ein selbstgeschriebenes Skript, dann nutzt man die Funktion *GetComponent()*[14] und kann diese dann über ihren Namen erreichen. Mit der folgenden Funktion erreicht man ein anderes Skript, welches zum gleichen GameObject gehört und ruft eine Funktion (TuEtwas()) darin auf:

```
function Update () {
  var anderesScript: AnderesScript = GetComponent(AnderesScript)
      ;
  anderesScript.TuEtwas();
}
```

Skript 7.2: Script adressieren mit GetComponent

Um ein GameObject zu adressieren, dem das aufrufende Skript nicht zugehörig ist, stellt Unity die *Find()*-Funktion[15] zur Verfügung. Dazu sollte eine Variable als GameObject deklariert werden und dieser anschließend das entsprechende GameObject zugeordnet werden. Man kann auf die *Find()*-Funktion auch verzichten, wenn man der Variable im Inspector ein GameObject zuordnet. Um die Spielperformance nicht leiden zu lassen, sollte die Funktion nicht in jedem Frame, wie beispielsweise in der *Update()*-Funktion, aufgerufen werden. Eine Erweiterung der *Find()*-Funktion ist *FindWithTag()*[15], bei der ein Objekt mit einem bestimmten Tag, z.B. „Player“ adressiert wird.

Hat man so nun das gewünschte Objekt gefunden und gegebenenfalls als Variable abgespeichert, kann man wie oben beschrieben, dessen Komponenten adressieren und ihre Funktionen aufrufen.

```
var anderesObjekt : GameObject;
var spieler : GameObject;
anderesObjekt = GameObject.Find("Anderes");
spieler = GameObject.FindWithTag("Player");
function Update () {
    anderesObjekt.GetComponent(VerhaltensSkript).TuEtwas();
    spieler.transform.Translate(0,0,3);
}
```

Skript 7.3: GameObject adressieren mit Find

In manchen Situationen möchte man nicht ein vorhandenes Objekt ändern, sondern ein neues erstellen. Das ist zum Beispiel der Fall, wenn Munition abgefeuert oder eine Explosion hervorgerufen werden soll. Für diesen Fall speichert man, wie in 4.4 beschrieben, einen Prefab des Objekts in seinem Projektordner. Um diesen nun ins laufende Spiel zu bekommen, muss via Skript ein Klon instanziiert werden. Dazu stellt Unity in der Klasse *Object* die Funktion *Instantiate()*[32] bereit. Der Funktion muss das Originalobjekt, die Position, an der es auftauchen soll und eine Rotation übergeben werden.

```
//Skript Shoot
var speed= 5;
var projectilePrefab : Rigidbody;

function Update () {
  // Pruefen, ob Munition vorhanden ist und der Spieler nicht
```

```
      tot ist.
  // beide Variablen werden im Skript Collisions abgespeichert.
  if(Collisions.munition>0 && Collisions.playerhealth>0){
    if (Input.GetButtonDown ("Fire1")) {
        //Granate erzeugen, an der Position der Hand, keine
           Drehung
      var projectile = Instantiate (projectilePrefab, transform.
         position, Quaternion.identity);
        //erzeugte Granate nach vorne schleudern
        projectile.rigidbody.AddForce(transform.forward*2000);
        //vorhande Munition um 1 verringern
        Collisions.munition--;
    }
  }
}
```

Skript 7.4: Instatiieren eines neuen Objekts

7.4. Andere wichtige Klassen und Funktionen

Um 3D-Vektoren anzugeben, wird in Unity die Klasse *Vector3*[41] benutzt. Damit kann man Positionen und Richtungen ausdrücken. Jeder *Vector3* besteht aus den drei Komponenten x, y und z. für einige wichtige Vektoren gibt es auch Abkürzungen. Beispielsweise steht *up* für *Vector3(0, 1, 0)* und *one* für *Vector3(1, 1, 1)*.

Die Klasse *Quaternion*[36] wird genutzt um Rotationen anzugeben. Wer sich nicht mit Quaternionen auskennt, sollte sie nicht selbst entwerfen, sondern stattdessen die von Unity vorgefertigten Rotationen benutzen.

```
//Skript Feind (Auszug)
//Wenn der Spieler gesehen wird
if (CanFind()){
  targetDirection=target.position - transform.position;
  targetRotation=Quaternion.LookRotation(targetDirection,Vector3
     .up);
  //in Richtung des Spielers drehen, zeitversetzt
```

```
    transform.rotation=Quaternion.Slerp(transform.rotation,
        targetRotation, Time.deltaTime *2.0);
}
```

Skript 7.5: Objekt entdecken und dorthin drehen

Will man sein Spiel nicht von Frameraten abhängig machen, sondern auf „echte“ Zeiten setzen, kann man die Klasse *Time*[38] verwenden. Diese zählt die Zeit in Sekunden. Insbesondere wird hier die Variable *Time.deltaTime* [39] benutzt, die angibt wie viele Sekunden es dauerte den letzten Frame zu beenden. Multipliziert man diese mit einem Wert x, bedeutet dies, dass man etwas x mal pro Sekunde, statt x mal pro Frame tun möchte.

```
function Update () {
  // Das Objekt 10 Meter pro Sekunde bewegen
  var translation : float = Time.deltaTime* 10;
  transform.Translate (0, 0, translation);
}
```

Skript 7.6: Objekt mit Geschwindigkeit bewegen durch Time.deltaTime

Die *Physics*-Klasse[33] wird genutzt um globale physikalische Einstellungen für eine Szene zu machen. So kann man etwa die Stärke und Richtung der Schwerkraft bestimmen. Außerdem hält sie einige Hilfsfunktionen für Kollisionen bereit. Eine sehr hilfreiche Funktion daraus ist *Raycast()*[34], welche einen Strahl von einem Punkt in eine bestimmte Richtung aussendet und mittels der Klasse *RaycastHit*[37] Informationen über dessen Kollisionen zurückgibt. Die *Physics*-Klasse kann statt *Raycasts* auch *Casts* in anderen Formen, wie *Line*, *Sphere* und *Capsule*, senden.

```
//Skript Collisions (Auszug)
function Update () {
  var hit: RaycastHit;
  //Erzeuge Raycast vom Spieler, 10 Meter nach vorne
  if (Physics.Raycast (transform.position,transform.forward,hit,
      10)){
    //Wenn Raycast das Objekt "Door" trifft
    if(hit.collider.gameObject.tag =="door"){
      //oeffne die Tuer (Spiele Animation ab)
      hit.collider.gameObject.animation.Play("openDoor");
```

```
    }
  }
}
```

Skript 7.7: Tür öffnen durch Raycast und Animation

Eine weitere sehr hilfreiche Klasse in Unity ist *PlayerPrefs*[35], mit der man wichtige Einstellungen in einer Systemdatei abspeichern kann. Dabei kann man für einen *key* (String) eine Zahl (Integer oder Float) und einen String speichern. Am häufigsten wird *PlayerPrefs* dafür genutzt, eine Highscoreliste mit Namen und Punktzahlen zu speichern. Man kann jedoch auch den Spielstand speichern, damit man beim nächsten Öffnen des Spiels an dieser Stelle weiterspielen kann.

```
//Skript Simon Says (Auszug)
//Ein Name und eine Punktzahl wird dem Highscore hinzugefuegt
function AddScore(name : String, score : int){
  var newScore : int;
  var newName : String;
  var oldScore : int;
  var oldName : String;
  newScore = score;
  newName = name;

  //Es gibt 10 Plaetze im Highscore.
  for(i=0;i<10;i++){
    //wenn es den Platz im Highscore schon gibt
    if(PlayerPrefs.HasKey("HScore"+i)){
      //wenn die neue Punktzahl hoeher ist
      if(PlayerPrefs.GetInt("HScore"+i)<newScore){
        //die alte Punktzahl wird zwischengespeichert
        oldScore = PlayerPrefs.GetInt("HScore"+i);
        oldName = PlayerPrefs.GetString("HScoreName"+i);
        //die neue Punktzahl wird auf diesem Platz eingetragen
        PlayerPrefs.SetInt("HScore"+i,newScore);
        PlayerPrefs.SetString("HScoreName"+i,newName);
        //fuer die alte Punktzahl wird ein neuer Platz gefunden
```

```
            newScore = oldScore;
            newName = oldName;
          }
        }
        //wenn es den Platz im Highscore noch nicht gibt
        else{
          PlayerPrefs.SetInt("HScore"+i,newScore);
          PlayerPrefs.SetString("HScoreName"+i,newName);
          newScore = 0;
          newName = "";
        }
      }
      PlayerPrefs.Save();
}
```

Skript 7.8: Highscore in PlayerPrefs hinterlegen

Die Klasse *Application*[11] nutzt man, um Daten zur Laufzeit des Spiels zu erhalten oder zu ändern. So kann man zum Beispiel in Erfahrung bringen, in welchem Level man sich befindet oder auf welcher Plattform man spielt. Wichtiger sind in dieser Klasse allerdings die Funktionen *Quit()*[13], um das Spiel zu beenden und *LoadLevel()*[12], um eine andere Szene zu laden.

```
//Skript Hauptmenu
function OnGUI () {
  if (GUI.Button (Rect ((Screen.width)-100,10,80,20),"Hauptmen\"
     u")) {
    Application.LoadLevel ("Start");
  }
}
```

Skript 7.9: Hauptmenu durch LoadLevel aufrufen

7.5. GUI

Das in Unity eingebaute System zum Erstellen eines *Graphic User Interface* ist *UnityGUI*. Im Gegensatz zur wie in 6.6 beschriebenen Möglichkeit GUI-Elemente als Objekte in die

Szene einzusetzen und ihnen anschließend per Skript eine Funktion zuzuteilen, kann man die GUI-Elemente gleich per Skript erstellen. Alle GUI-Objekte müssen in der Funktion *OnGUI()*[25] beschrieben werden, da es sonst zu Fehlermeldungen kommt und sie nicht gerendert werden.

Die Klasse *GUI*[16] stellt einige Funktionen bereit, um GUI-Objekte zu erzeugen, die auch in ihrer Farbe verändert werden können. So können Boxes, Buttons, Slider, Textfelder, Scrollbars, popup-Fenster und Texturen erstellt werden. Die gleichen Elemente stehen auch in der *GUILayout*-Klasse[17] zur Verfügung, in der ihnen aber ein automatisches Layout zugeordnet wird. Der Vorteil hierbei ist, dass man nur einmal eine Größen- und Positionsangabe für die gesamte GUI machen muss und anschließend alle Elemente der Reihe nach darin angeordnet werden. Nutzt man die Klasse *GUI*, muss diese Angabe für jedes Element einzeln gemacht werden.

Um ein GUI-Element zu erzeugen, wird dieses mit der entsprechenden Klasse, gefolgt von Position, Größe, Inhalt oder Werte und gegebenenfalls einem Stil aufgerufen. Möchte man nicht die von Unity gestalteten Standard GUI-Elemente nutzen, kann man mit einem *GUI Style* deren Aussehen verändern. Ebenso kann man eine Sammlung solcher *GUI Styles* anlegen, in der man Stile für die verschiedenen Elemente, wie Buttons, Slider etc. festlegt. Diese Sammlungen sind Assets, werden also im Projekt gespeichert und nennen sich *GUI Skin*. Erzeugt werden sie, wie andere Assets, über den Menüpunkt Assets -> Create -> GUI Skin und angewendet indem der Variable *GUI.skin* der Name des erstellten GUI Skins zugeteilt wird.

Ruft man ein GUI-Element, wie etwa einen Button, innerhalb einer if-Anweisung auf, wird dieser erzeugt, gleichzeitig wird ihm eine Funktion zugewiesen, die beim Drücken ausgeführt wird.

```
//Skript GUI Terrain
var laufen: GameObject;
var fliegen: GameObject;
var flycam: GameObject;

function OnGUI () {
  if (GUI.Button (Rect (10,10,80,20),GUIContent("Laufen","Rennen
     : Shift, Springen: Leertaste"))) {
    laufen.active=true;
    fliegen.active=false;
```

```
    flycam.active=false;
  }
  if (GUI.Button (Rect (10,40,80,20),GUIContent("Fliegen","Nach
      oben mit x, nach unten mit y"))) {
    laufen.active=false;
    fliegen.active=true;
    flycam.active=true;
  }
  GUI.Label (Rect (100,40,120,120), GUI.tooltip);
}
```

Skript 7.10: GUI Buttons mit Tooltip

Abbildung 7.4.: Ergebnis der *OnGUI()*-Funktion im Spiel. Der Mauszeiger befindet sich auf dem Laufen-Button, daher wird ein *Tooltip* im festgelegten Rechteck angezeigt. Durch vorheriges Drücken des Laufen-Buttons wurde die Kamera, die dem Spieler folgt und die Steuerung für den *Third Person Controller* aktiviert.

```
//Skript Simon Says (Auszug)
//Spielernamen abfragen
if (spielername) {
  GUI.Box (Rect ((Screen.width/2)-130, (Screen.height/2)-70,260,
```

```
        140),"");
    GUILayout.BeginArea (Rect ((Screen.width/2)-120, (Screen.
        height/2)-60, 240, 120));
    GUILayout.BeginVertical();
    GUILayout.Label ("Du hast Level "+ (counter-1) +" geschafft
        und bist im Highscore!");
    GUILayout.Label ("Gib deinen Namen ein:");
    pname = GUILayout.TextField (spname, 25);
    if (GUILayout.Button ("OK")) {
      AddScore(spname, counter-1)
    }
    GUILayout.EndVertical();
    GUILayout.EndArea();
}
```

Skript 7.11: GUILayout mit TextField zum Abfragen eines Spielernamens

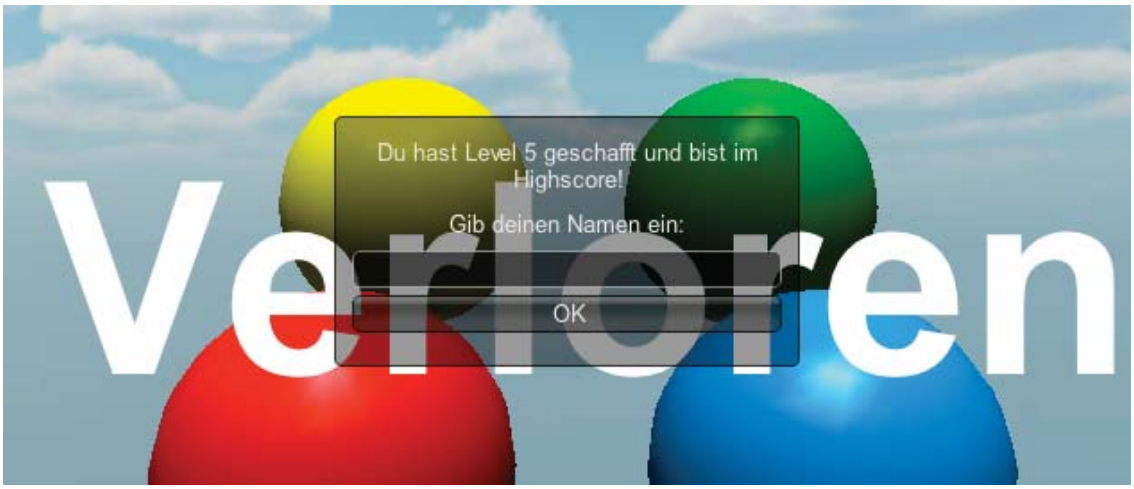


Abbildung 7.5.: Ergebnis der if-Anweisung in der *OnGUI()*-Funktion im Spiel *Simon Says* für den Wert „true“. Der eingegebene Name wird nach Drücken des OK-Buttons in der Variable „pname“ gespeichert und zusammen mit dem Wert des Rundenzählers an die Funktion AddScore() übergeben, welche den Namen im Highscore speichert.

8. Steuerung von Objekten und Spielern

Um Objekte in einem Spiel zu steuern, werden in der Regel Maus- und Tastatureingaben in das Spiel und auf Spielfiguren übertragen.

In Unity ist es auch möglich Eingaben eines Touchscreens, sowie verschiedener Controller weiterzuleiten.

Mit dem *Input Manager* (Edit->Project Settings->Input) kann man die Standardeinstellung für die Belegung verschiedener *Axes* verändern oder erweitern. Dabei kann man jedem *Axis* einen Namen geben, um ihn von einem Skript aus zu erreichen, einen Eingabetyp festlegen und gegebenenfalls je zwei Buttons für positive und negative Richtungen vergeben [56]. Durch die Klasse *Input*[18] kann man Eingaben abfragen. Mit der Funktion *GetAxis()*[19] kann man Richtungsangaben auslesen, die mit der Maus oder Tastatur gemacht werden.

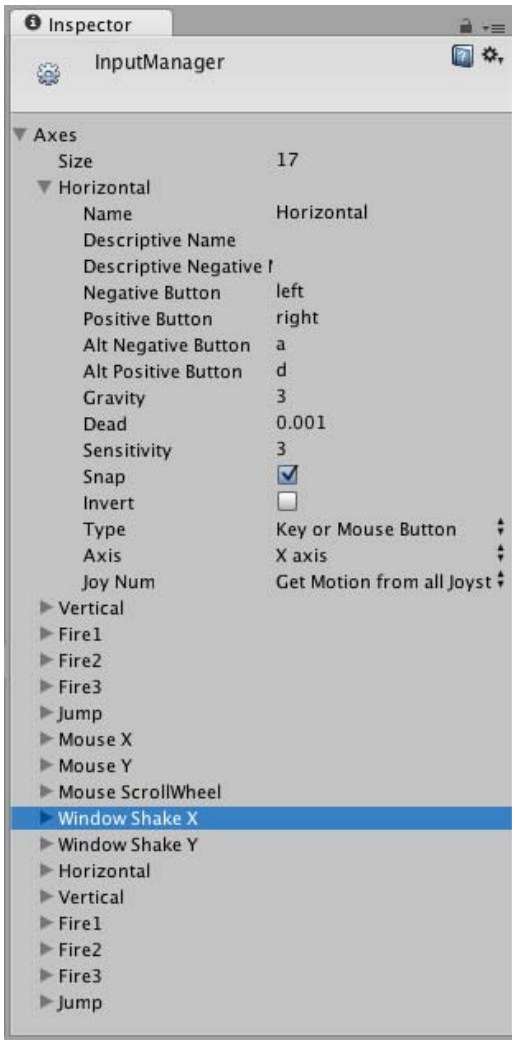


Abbildung 8.1.: Input Manager.

```
//Skript Fahren

var forwardSpeed: float=30;
var turnSpeed: float=2;
private var forwardMove: float;
private var turnMove: float;


function Update () {
  //Eingabe fuer vertikale Achse abfragen (-1 oder 1) und mit
     Geschwindigkeit multiplizieren
  forwardMove = Input.GetAxis ("Vertical")*forwardSpeed;
```

```
    //Eingabe fuer horizontale Achse
    turnMove = Input.GetAxis ("Horizontal")*turnSpeed;
    //lenken
    transform.Rotate(0,turnMove,0);
    //fahren
    rigidbody.AddRelativeForce (forwardMove,0,0);
}
```

Skript 8.1: Fahrzeugsteuerung durch Pfeiltasten

Will man Axes für Ereignisse abfragen, die keine Bewegungen darstellen, wie etwa *Fire* oder *Jump*, verwendet man die Funktion *GetButton()*[20], bzw. *GetMouseButton()*[21] für Eingaben mit der Maus. Unterstützend zur Steuerung mit der Maus können auch deren Koordinaten abgefragt werden, um diese beispielsweise als Schussposition zu verwenden.

```
var particle : GameObject;
function Update () {
  if (Input.GetButtonDown ("Fire1")) {
    // Konstruiere einen Strahl von der aktuellen Mausposition
  var ray : Ray = Camera.main.ScreenPointToRay(Input.
      mousePosition);
    if (Physics.Raycast (ray)) {
      // Kreiere einen Partikel wenn etwas den Strahl trifft
      Instantiate (particle, transform.position, transform.
          rotation);
    }
  }
}
```

Skript 8.2: Koordinatenabfrage der Mausposition

Zur Steuerung eines Spielers mit Tastatur und Maus, stellt Unity schon zwei vorgefertigte Skripte bereit, einen First Person und einen Third Person Controller. Die Steuerung per Smartphone oder Tablet mit den Betriebssystemen iOS und Android ist sehr viel umfangreicher, da man diese nicht durch Knöpfe und Mausbewegungen steuern kann. Um Zugriff auf die Bildschirmberührungen zu bekommen, bietet *Input* die Funktion *GetTouch()*[22]. Die Position und Positionsveränderungen eines Fingers werden durch die Struktur *Touch*[40] beschrieben. Des Weiteren kann Unity die Orientierung des Geräts

mittels *Input.deviceOrientation* auslesen, wodurch man herausfinden kann, ob das Gerät im Hoch- oder Querformat gehalten wird oder mit der Oberseite nach unten bzw. oben. Ebenso kann der Beschleunigungssensor ausgelesen werden, wodurch die Spielsteuerung mittels Bewegung des Geräts im dreidimensionalen Raum möglich ist. Die Variablen hierfür sind *Input.acceleration* und *Input.accelerationEvents*.

Durch die Variablen *Input.compass* und *Input.location* sind auch Zugriffe auf den Kompass und den Standort des Geräts möglich.

Eine weitere Möglichkeit zur Steuerung bietet der Festplattenschutz von Laptops, bei Apple der *Sudden Motion Sensor*. Dadurch kann man mit einem MacBook ein Spiel ähnlich wie mit einem Smartphone steuern. Leider ist es nicht möglich diesen durch eine von Unity bereitgestellte Klasse abzufragen. Es gibt allerdings schon Plugins dafür, welche man aber nur mit UnityPro nutzen kann [2].

9. Beispiele

Im Rahmen dieser Studienarbeit habe ich einige kleine Spiele und Anwendungen entwickelt, um die Grundlagen und Features von Unity kennenzulernen und zu testen. Sie sind alle nicht sehr umfangreich, da es sehr zeitintensiv ist ein Spiel zu entwickeln und ich in kurzer Zeit möglichst vieles kennenlernen wollte, was Unity bietet.

9.1. Mondfahrzeug fahren

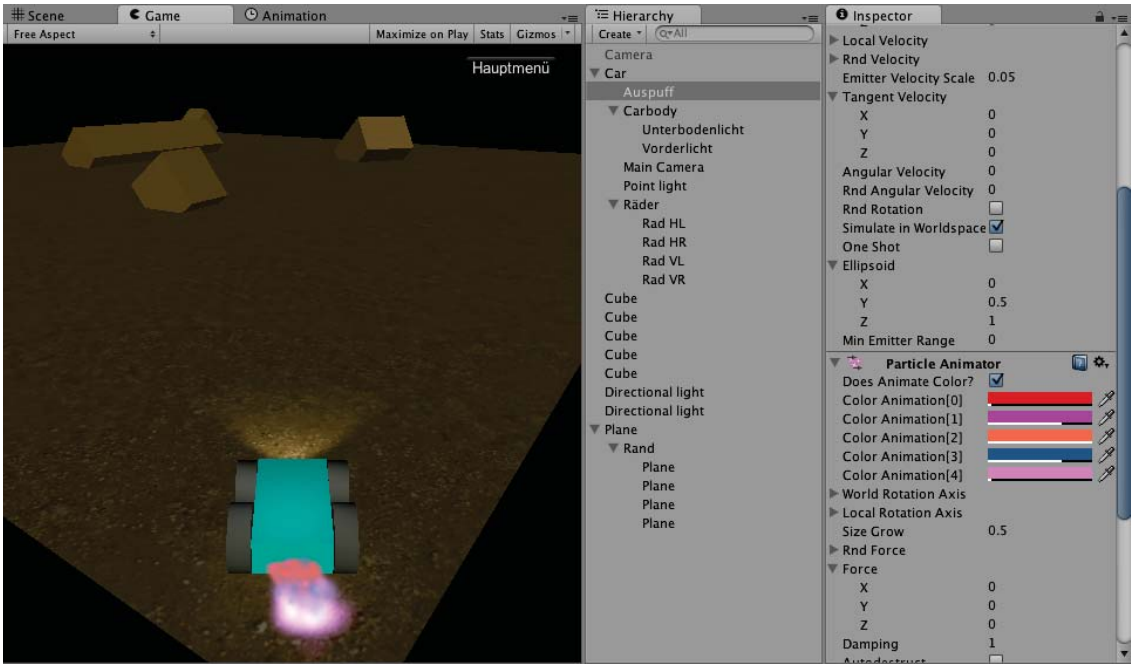


Abbildung 9.1.: Die Szene „Mondfahrzeug fahren“. In der Game View sieht man die gesamte Spielfläche, sowie das Fahrzeug samt Beleuchtung und Abgase. An den quaderförmigen Hindernissen ist das gerichtete Licht gut erkennbar. In der Hierarchy View ist das GameObject „Auspuff“, welches ein *Particle System* ist, angewählt. Im Bildausschnitt sieht man die gewählten Farben für die Partikel, sowie die Form (Ellipsoid).

Diese erste Szene dient ersten physikalischen Versuchen. Eine einfache Grundfläche mit einem *Collider* wurde erstellt und anschließend mit *Directional Lights* beleuchtet. Anschließend wurde ein kleines, einfaches Fahrzeug entwickelt, welches nur aus primitiven Formen und zwei Lichtern besteht. Damit das Fahrzeug nicht von der Grundfläche fällt,

befinden sich drum herum unsichtbare Flächen mit *Collidern*. Um es dem Fahrzeug nicht ganz so leicht zu machen, befinden sich auf der Grundfläche einige Hindernisse in Form von Quadern, ebenfalls mit *Collidern* ausgestattet.

Die ersten Versuche mit einem *Particle System* äußern sich in einer Art Abgas, welches das Fahrzeug während es sich bewegt ausstößt. Realisiert wurde dies, indem die Partikel nur auf der Stelle austreten und sich in keine Richtung bewegen. Da die Teilchen eine *Energy* (Lebensdauer) von 3-5 haben, sind diese noch kurze Zeit sichtbar, wodurch ein Schweif entsteht, wenn sich das Fahrzeug bewegt.

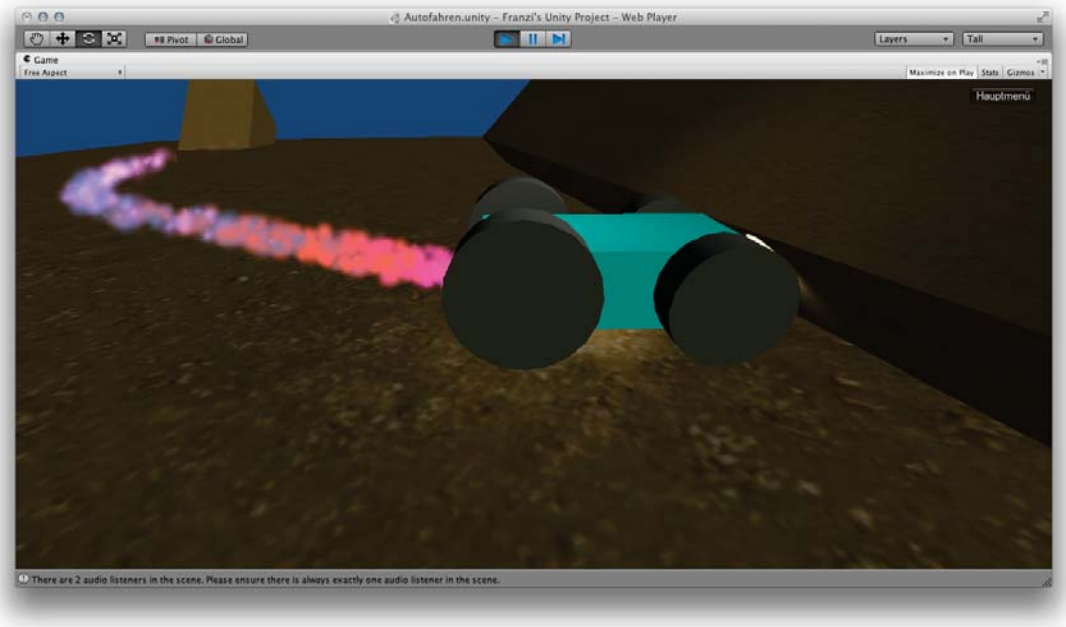


Abbildung 9.2.: Mondfahrzeug mit Schweif.

Um Kräfte auf das Fahrzeug wirken zu lassen, dass es bewegt werden kann, wurde die *Rigidbody*-Komponente hinzugefügt. Im einzigen Skript des Spiels wurde die Steuerung des Fahrzeugs implementiert (siehe Skript 8.1). Die Main Camera des Spiels wurde als untergeordnetes Objekt dem Fahrzeug hinzugefügt. Die Kamera folgt diesem dadurch starr.

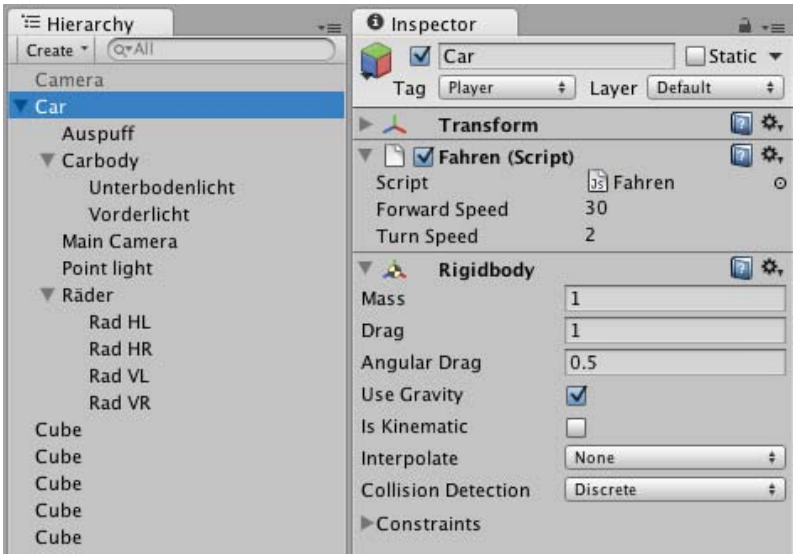


Abbildung 9.3: Script „Fahren“. Im Inspector ist das Skript „Fahren“, welches als Komponente zum GameObject „Car“ hinzugefügt wurde, mit zwei veränderbaren Geschwindigkeitsvariablen zu sehen.

Die farbliche Gestaltung der Szene erfolgte durch meine ersten drei eigenen Materialien, alle mit dem Shader *Diffuse* ausgestattet. Für die Grundfläche und die Hindernisse wurde eine wüstenartige Textur verwendet, für das Fahrzeug die Farben Türkis und Grau.

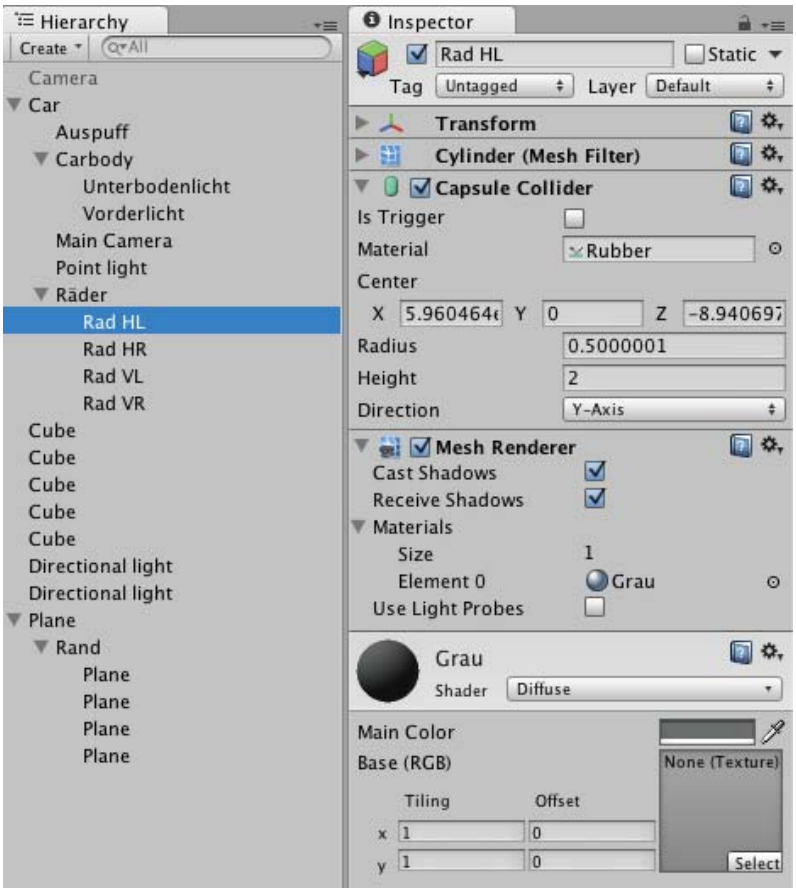


Abbildung 9.4.: Material „Grau“. Im Inspector ist in der *Mesh Renderer*-Komponente das Material „Grau“ zu sehen, welches auf alle Räder angewandt wurde.

9.2. Kugeln

Das Nächste Spiel beschäftigt sich ebenfalls mit physikalischen Aspekten, diesmal stehen jedoch die *Joints* und *Character Controller* im Vordergrund. Als Grundfläche dient wieder eine Fläche mit *Collider*. Auf der Fläche befinden sich viele „Bretter“, Quader mit einer *Rigidbody*-Komponente ausgestattet. Diese sind als Stapel oder wie eine Reihe Dominosteine angeordnet. Eine Kugelkette dient als Spieler.

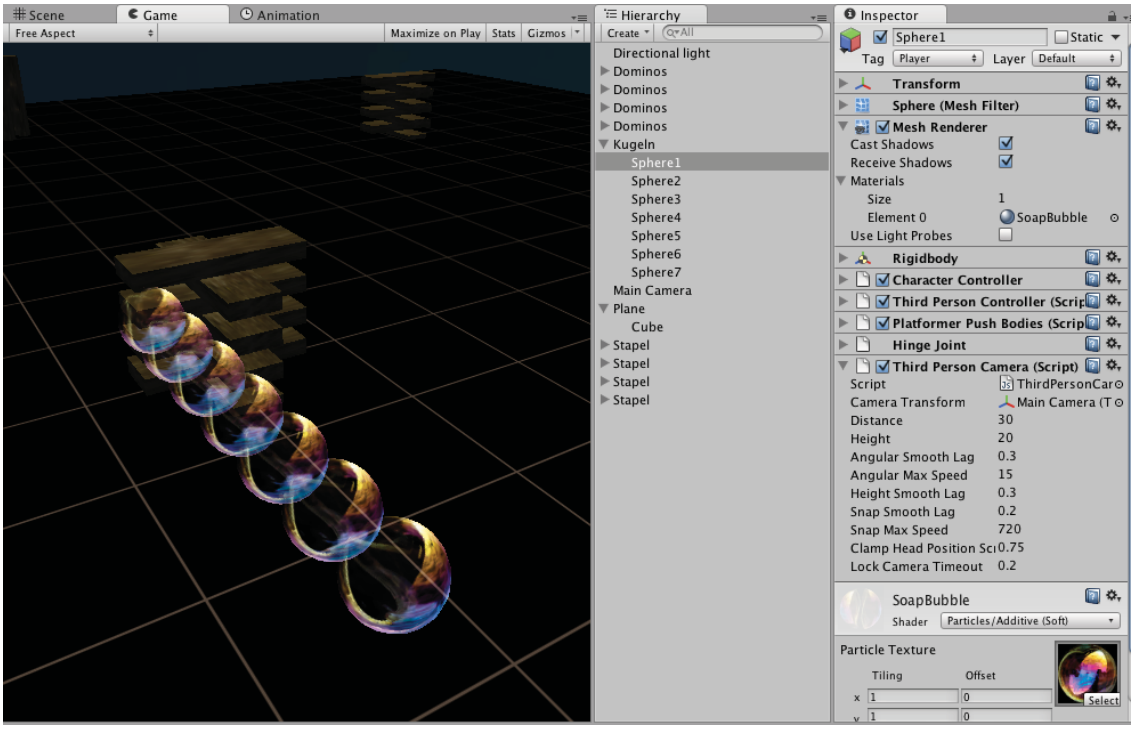


Abbildung 9.5.: Kugelkette. Im Inspector sind die Komponenten der ersten Kugel in der Kette zu sehen.

Um ohne viele Skripte auszukommen, wurde der ersten Kugel der Kette ein *Character Controller* zusammen mit einem *Third Person Controller*-Skript, welches in den *Unity Standard Assets* mitgeliefert wird, angehängt. Dadurch ist die Kugel mit den Pfeiltasten steuerbar und kann sogar springen. Das *Third Person Camera*-Skript lässt eine ausgewählte Kamera dem Objekt folgen.

Mit einem *Hinge Joint* ist die erste Kugel mit der zweiten verbunden. Die Verbindung ist dadurch nicht starr, sondern beweglich, wie bei einer Kette.

Um den *Hinge Joint* überhaupt anwenden zu können, benötigt die Kugel eine *Rigidbody*Komponente. Diese erfüllt gleich einen weiteren Zweck, da bei der Kollision zweier Rigidbodys Kräfte auf diese einwirken und man dadurch mit den Kugeln die Holzstapel umstoßen kann. Da die erste Kugel durch das *Third Person Controller*-Skript bewegt wird, welches auf die *Character Controller*-Komponente zugreift, wirken bei der Bewegung keine Kräfte auf die *Rigidbody*-Komponente. Die Kugel kann also dennoch keine Kraft auf die Bretter ausüben. Die Lösung dazu liegt im Skript *PlatformerPushBodys* aus einem Demoprojekt von Unity. Ein *Character Controller* kann dadurch Kräfte auf Rigidbodys wirken lassen.

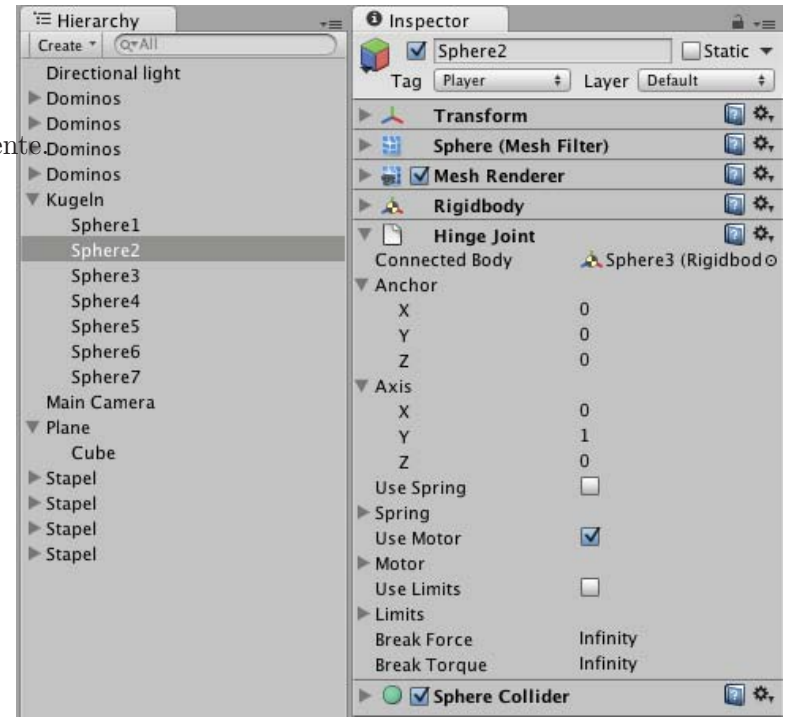


Abbildung 9.6.: Hinge Joint. Im Inspector ist die *Hinge Joint*-Komponente der zweiten Kugel zu sehen. Sie ist mit der dritten Kugel verbunden. Analog ist die dritte Kugel mit der vierten verbunden usw.

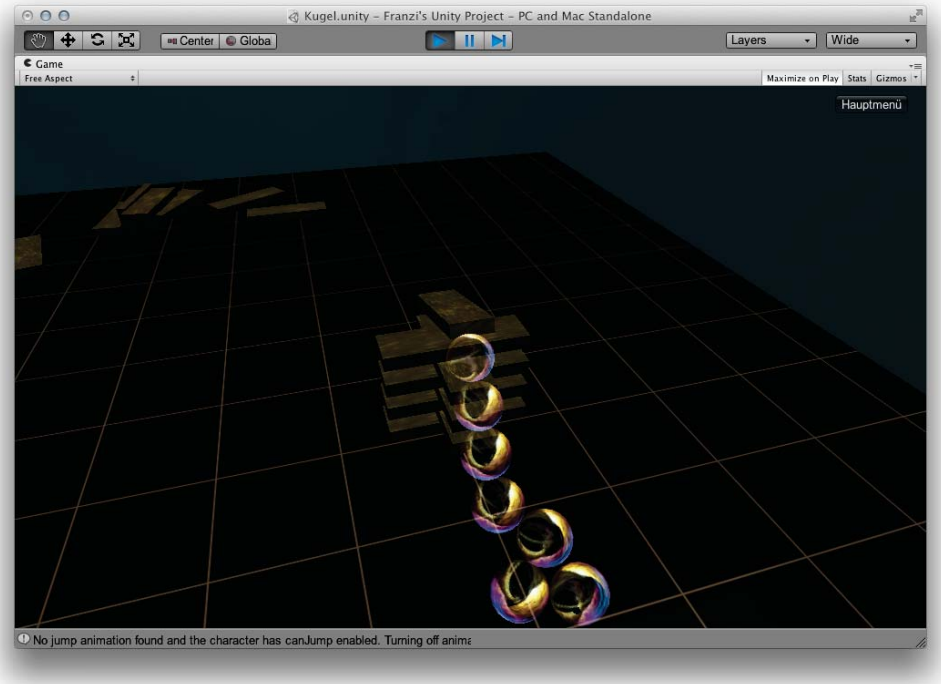
Das einzige selbstgeschriebene Skript in dieser Szene dient dazu, das Verlassen der Kugeln von der Spielfläche zu steuern. Das Skript ist an einen großen *Box Collider* um die Spielfläche gehängt. Sobald der Spieler diese verlässt, wird er an eine bestimmte Stelle auf das Spielfeld zurück gesetzt.

```
//Script: Falloff
var player: GameObject;
var position: Vector3;

function OnTriggerExit (player : Collider) {
  yield WaitForSeconds (2);
```

```
    player.transform.position=position;
}
```

Skript 9.1: Spieler aufs Spielfeld zurücksetzen



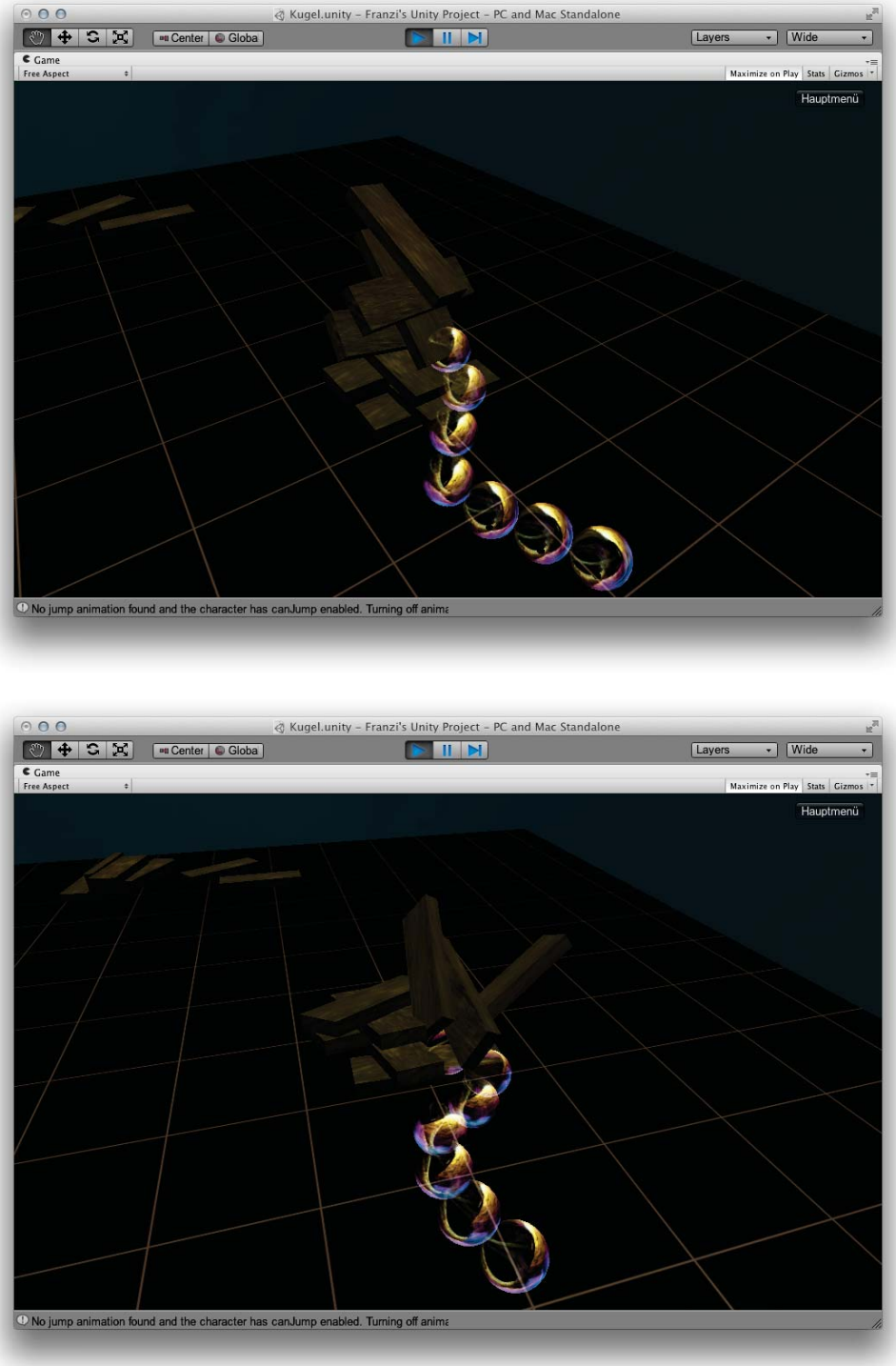


Abbildung 9.7.: Kugelkette. Die Kugelkette springt auf einen Holzstapel. Durch die Schwerkraft, die auf die Kugeln beim Herunterfallen wirkt, werden die Bretter nach oben geschleudert. Die Fehlermeldung am unteren Rand bemerkt, dass für das Springen keine Animation hinterlegt wurde, da das *Third Person Controller*-Skript eine Animation für den Sprung erwartet.

9.3. Terrain

Im nächsten Schritt wurde das *Terrain Tool* getestet, wofür mit Photoshop eine Höhenkarte erstellt wurde. Als Grundlage diente ein Screenshot einer Flussmündung irgendwo in Mittelamerika aus Google Earth. Da der Fluss am tiefsten lag, wurde er in Schwarz nachgezeichnet und die Umgebung dann in verschiedenen Graustufen. Das Ergebnis im RAW Format abgespeichert, kann man in Unity als Höhenkarte für ein Terrain importieren. Hier wurden dann einige Korrekturen und Glättungen vorgenommen, sowie Berge mit den Unity Terrain Tools hinzugefügt. Anschließend wurde das Ganze mit verschiedenen Texturen der *Unity Standard Assets* „bemalt“ und einige Bäume und Gräser hinzugefügt. Das Wasser ist eine Fläche, die in Höhe des Wasserstands auf das Terrain gelegt und mit dem Material „Nighttime Simple Water“ aus den *Unity Standard Assets* versehen wurde. An allen Stellen, an denen das Terrain höher als der Wasserstand ist, befindet sich die Fläche unterhalb des Terrains und ist nicht zu sehen.

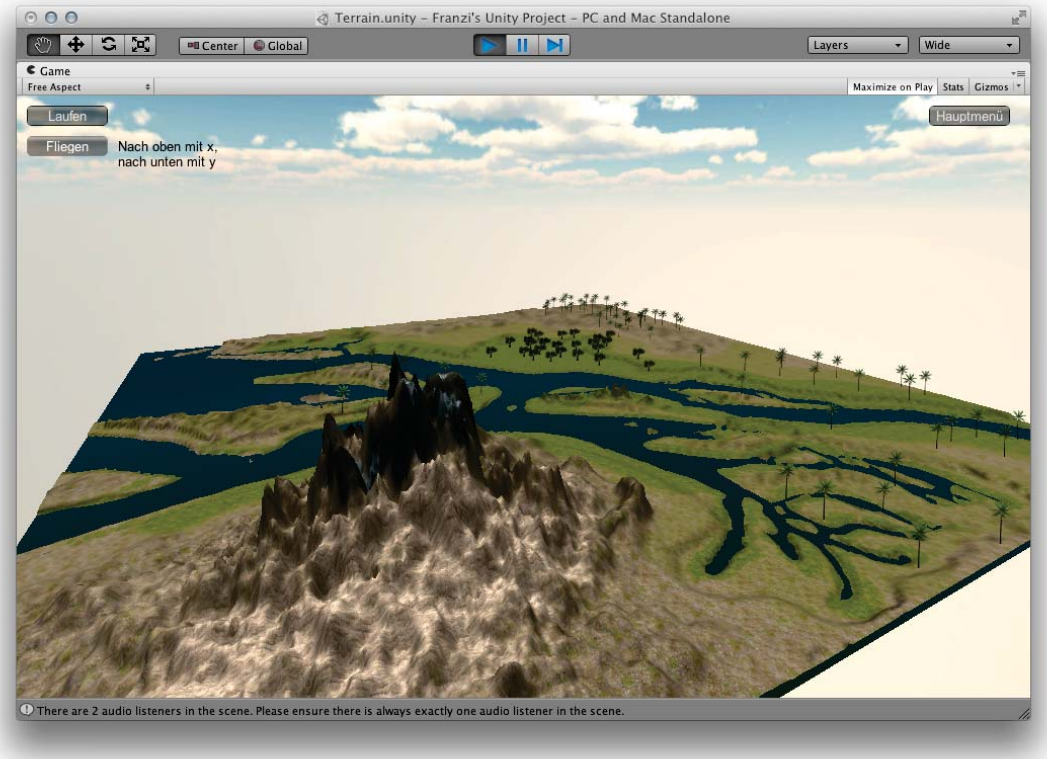


Abbildung 9.8.: Terrain aus Sicht eines Flugzeugs.

Man kann die Landschaft mit einer Spielfigur aus den *Standard Assets* abgehen, der jeweils an den Beinen ein *Particle System* namens „Water Surface Splash“ (*Standard Assets*) hinzugefügt wurde, welches aktiv wird, sobald die Figur die Wasserfläche berührt und sich im Wasser bewegt.

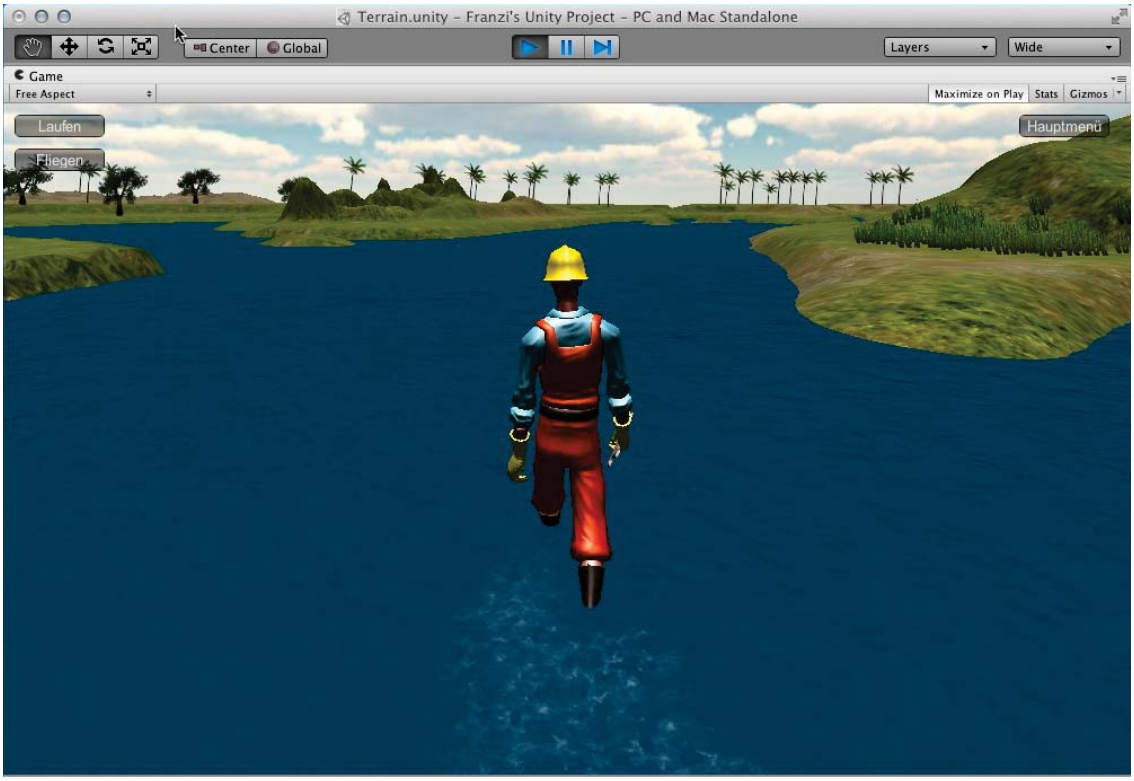


Abbildung 9.9.: Water Surface Splash.

Alternativ kann man das Gelände auch aus Sicht eines Flugzeugs erkunden, welches ähnlich gesteuert wird wie das Mondfahrzeug in der vorherigen Szene.

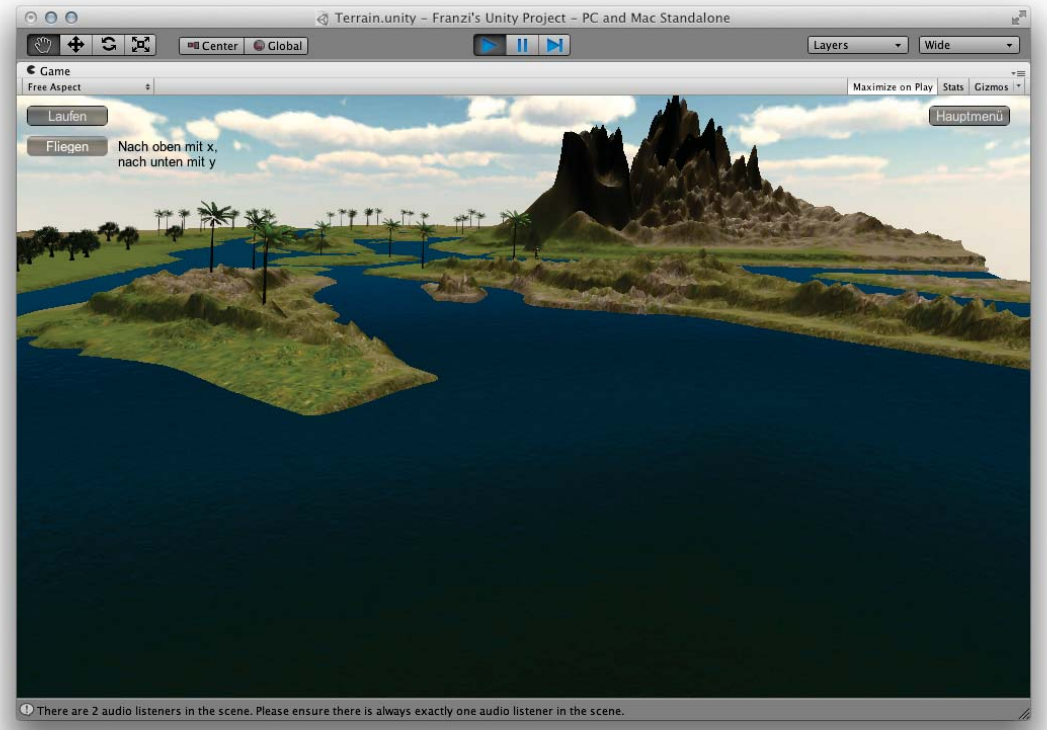


Abbildung 9.10.: Terrain aus Sicht eines Flugzeugs.

Um zwischen Flugzeug und Spieler umzuschalten, sind hier zum ersten Mal GUI-Elemente eingesetzt worden. Diese werden per Skript, welches als Komponente an das GameObject „My Terrain“ gehängt wurde, erzeugt (siehe Skript 7.10).

Einige Screenshots der Szene sind auch schon in den Abbildungen 3.2 und 3.4, sowie 7.4 zu sehen.

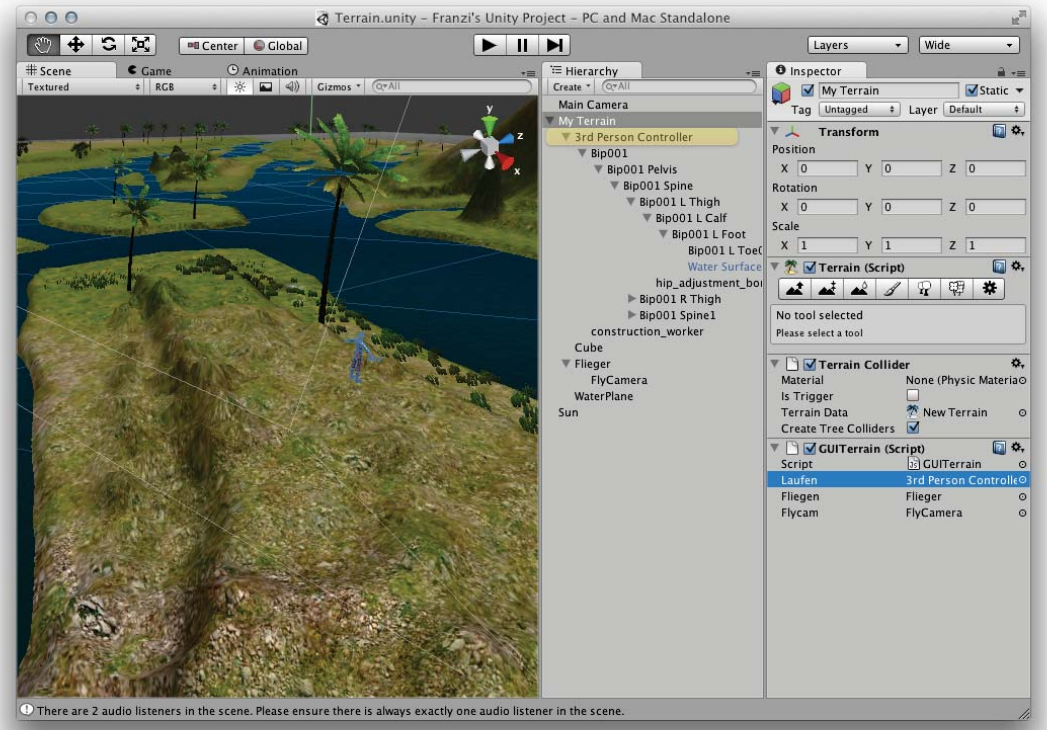


Abbildung 9.11.: GUI Terrain. Den einzelnen Variablen aus dem Skript wurden hier GameObjects zugeteilt. Wird „Laufen“ gedrückt, aktiviert sich das GameObject „3rd Person Controller“, die GameObjects „Flieger“ und „FlyCamera“ werden deaktiviert.

9.4. Start

Um ein wenig mehr mit den GUI-Elementen zu experimentieren und eine Übersicht über die bisherigen erstellten Szenen zu bekommen, wurde die Szene „Start“ erstellt. Sie besteht nur aus einem GameObject, der Main Camera mit den zusätzlichen Komponenten Skybox und dem Skript „Start“, welches die GUI-Elemente erzeugt.

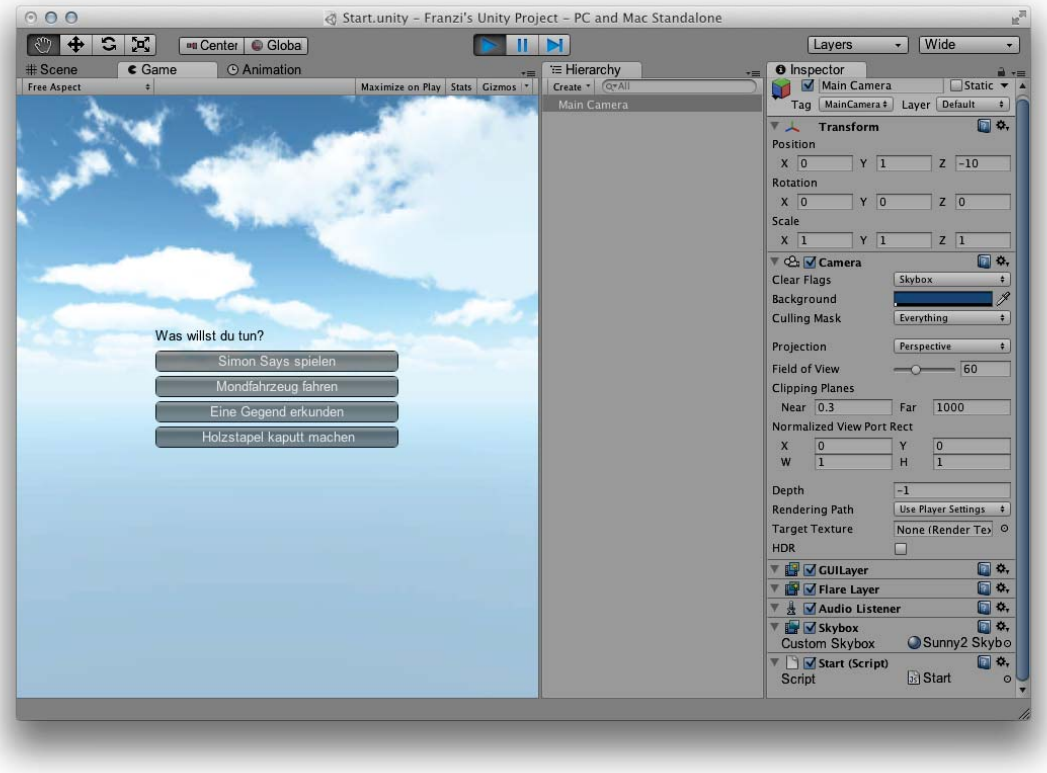


Abbildung 9.12.: Szene „Start“.

```
//Skript Start
function OnGUI () {
  var label = GUI.skin.GetStyle("Label");
  label.fontStyle = FontStyle.BoldAndItalic;
  label.normal.textColor = Color.black;

  GUILayout.BeginArea (Rect ((Screen.width/2)-120, (Screen.
      height/2)-70, 240, 140));
    GUILayout.BeginVertical();
      GUILayout.Label ("Was willst du tun?",label);
      if (GUILayout.Button ("Simon Says spielen")) {
        Application.LoadLevel ("Simon Says");
      }
```

```
        if (GUILayout.Button ("Mondfahrzeug fahren")) {
          Application.LoadLevel ("Autofahren");
        }
        if (GUILayout.Button ("Eine Gegend erkunden")) {
          Application.LoadLevel ("Terrain");
        }
        if (GUILayout.Button ("Holzstapel kaputt machen")) {
          Application.LoadLevel ("Kugel");
        }
    GUILayout.EndVertical();
  GUILayout.EndArea();
}
```

Skript 9.2: GUI einer Startszene

9.5. Simon Says

Da man ab einer bestimmten Stelle kaum noch mit einfachen Skripten auskommt, wurde zu Scripting-Übungszwecken ein Spiel erstellt, welches größtenteils aus Skripten besteht. Es handelt sich um ein bekanntes Merkspiel namens „Simon Says“. Eine Sequenz von farbigen Knöpfen leuchtet auf und man muss diese anschließend selbst in der richtigen Reihenfolge drücken. Die Szene besteht nur aus wenigen GameObjects: vier Kugeln, eine unbewegliche Kamera und ein Directional Light. Die Kugeln sind zunächst farblos und erhalten erst zur Laufzeit per Skript ihre Farbe.

```
//Skript BunteKugel
var farbe: Color; //Farbe ist ueber den Inspector waehlbar
private var highlightMultiply = 3;

renderer.material.shader = Shader.Find ("VertexLit");
renderer.material.color = farbe;

function OnMouseOver() {
  if (Input.GetMouseButtonDown(0)){
      this.SendMessageUpwards("Clicked", this.name);
```

```
    // Ruft die Funktion Clicked() im Skript des uebergeordneten
        GameObjects auf und uebergibt den Namen der Kugel
  }
  if (Input.GetMouseButtonUp(0)){
    NoGlow(); //Kugel hoert auf zu leuchten
    this.SendMessageUpwards("Check");
    // Ruft die Funktion Check() auf
  }
}

// Aufleuchten der Kugel
function Glow() {
  renderer.material.color.r = farbe.r*highlightMultiply;
  renderer.material.color.g = farbe.g*highlightMultiply;
  renderer.material.color.b = farbe.b*highlightMultiply;
}

//Normale Farbe der Kugel
function NoGlow() {
  renderer.material.color = farbe;
}
```

Skript 9.3: Kugel einfärben und aufleuchten lassen mit OnMouseOver

Ein weiteres Skript dieser Szene beinhaltet alle anderen Funktionen und ist an die Main Camera gehangen.

```
//Script Simon Says
var sequence = new Array();  //von Computer generierte Sequenz
var counter: int = 0; //Rundenzaehler
var clickedsequence = new Array();  // vom Spieler geklickte
   Sequenz
var farbnr: int;  //Farbnummer der Kugeln
var vSlider: float= 0.0;  //Geschwindigkeitsslider
var falschmeldung: GameObject;
var spname: String=" "; //Spielername fuer den Highscore
```

```
//Zustaende
enum eState {
  INVALID_STATE,
  THINKING,
  DISPLAYING,
  WAITING,
  NUM_STATES
}
var state: eState = eState.INVALID_STATE;

// Waehrend eine Kugel angeklickt wird, soll diese leuchten
function Clicked (kugel) {
  if (state == eState.WAITING){
    if (kugel == "YellowSphere"){
      farbnr = 1;
      GameObject.Find("YellowSphere").GetComponent("BunteKugel")
          .Glow();
    }
    if (kugel == "GreenSphere"){
      farbnr = 2;
      GameObject.Find("GreenSphere").GetComponent("BunteKugel").
          Glow();
    }
    if (kugel == "RedSphere"){
      farbnr = 3;
      GameObject.Find("RedSphere").GetComponent("BunteKugel").
          Glow();
    }
    if (kugel == "BlueSphere"){
      farbnr = 4;
      GameObject.Find("BlueSphere").GetComponent("BunteKugel").
          Glow();
    }
```

```
    //Die angeklickte Kugel wird der geklickten Sequenz
       hinzugefuegt
    clickedsequence.Push(farbnr);
  }
}

// Nachdem die Kugel angeklickt wurde, wird ueberprueft, ob es
   die richtige war
function Check (){
  if (state == eState.WAITING){
    //Die falsche Kugel wurde angeklickt
    if (Verify() == false){ //Die Funktion Verify wird
       aufgerufen
      falschmeldung.active=true;  //Dem Spieler wird mitgeteilt,
          dass er verloren hat
      if (counter-1 > PlayerPrefs.GetInt("HScore"+9)){
        spielername = true; // Spielername soll abgefragt werden
           , falls es einen neuen Highscore gibt
      }
    }
    else {
    //Die richtige Reihenfolge wurde geklickt
     if (clickedsequence.length == sequence.length){
      yield WaitForSeconds (2);
      clickedsequence.Clear();  //Die angeklickte Sequenz wird
         geloescht
      state = eState.THINKING;  //Der Computer ueberlegt
    }
    }
  }
}

//Wurde die richtige Sequenz angeklickt?
function Verify(){
```

```
  for (i=0; i<clickedsequence.length; i++){
    if (sequence[i]!=clickedsequence[i])
      return false;
  }
  return true;
}

//Der Computer ueberlegt welche Kugel als naechstes aufleuchten
   soll
function NewSequence () {
  if (state == eState.THINKING){
    sequence.Add(Random.Range(1,5));
    state = eState.DISPLAYING;
    ++counter;  //Die Runde wird hochgezaehlt
    DisplaySequence();  //Die Sequenz wird angezeigt
  }
}

//Die Kugeln in bestimmter Reihenfolge aufleuchten lassen
function DisplaySequence () {
  if (state == eState.DISPLAYING ){
    //Alle 3 Runden wird die Geschwindigkeit erhoeht.
    if (counter % 3 ==0 && vSlider<=4.7)
      vSlider=vSlider + 0.3;
    var v: float = 1/vSlider; //Die Wartezeit beim Aufleuchten
       der Kugeln
    for (i=0; i<sequence.length; i++){
      farbnr=sequence[i];
      if (farbnr == 1){
        GameObject.Find("YellowSphere").GetComponent("BunteKugel
            ").Glow();
        yield WaitForSeconds (v);
        GameObject.Find("YellowSphere").GetComponent("BunteKugel
            ").NoGlow();
```

```
            yield WaitForSeconds (v);
        }
        if (farbnr == 2){
            GameObject.Find("GreenSphere").GetComponent("BunteKugel"
                ).Glow();
            yield WaitForSeconds (v);
            GameObject.Find("GreenSphere").GetComponent("BunteKugel"
                ).NoGlow();
            yield WaitForSeconds (v);
        }
        if (farbnr == 3){
            GameObject.Find("RedSphere").GetComponent("BunteKugel").
                Glow();
            yield WaitForSeconds (v);
            GameObject.Find("RedSphere").GetComponent("BunteKugel").
                NoGlow();
            yield WaitForSeconds (v);
        }
        if (farbnr == 4){
            GameObject.Find("BlueSphere").GetComponent("BunteKugel")
                .Glow();
            yield WaitForSeconds (v);
            GameObject.Find("BlueSphere").GetComponent("BunteKugel")
                .NoGlow();
            yield WaitForSeconds (v);
        }
    }
    state = eState.WAITING; //Der Computer wartet auf die
        Spielereingabe
  }
}

//Der Computer ueberprueft, ob er dran ist. Wenn ja fuegt er
    eine neue Zahl/Kugel zur Sequenz hinzu.
```

```
function Update () {
  if (state == eState.THINKING){
    NewSequence();
  }
}

function Start () {
  sequence.Clear();
  clickedsequence.Clear();
  counter = 0;
  farbnr = 0;
  vSlider = 2;
  spname = "  ";
}
```

Skript 9.4: Simon Says Skript

Die Highscoreabfrage der Szene wurde schon in Skript 7.8 und Skript 7.11 behandelt.

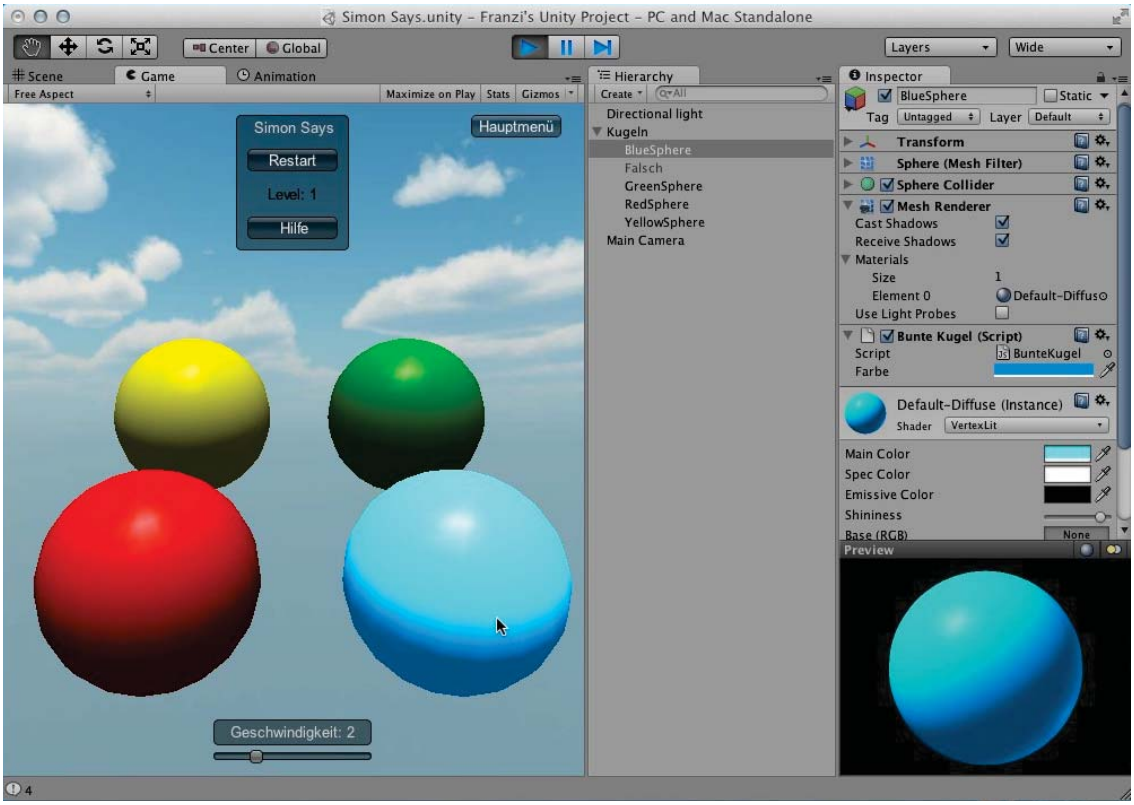


Abbildung 9.13.: Simon Says. Die blaue Kugel wird angeklickt und leuchtet auf. Die Veränderung ist auch im Inspector zu sehen. Die ursprünglich gewählte Farbe ist in der Komponente „Bunte Kugel“ zu sehen, die momentane Farbe als Main Color im Material.

9.6. Shooter

Diese Szene stellt einen kleinen Egoshooter dar. Man muss mit seinem Spieler auf die andere Seite einer Mauer gelangen. Ist man nah genug an der Mauer, öffnet sich eine Tür, was durch eine Animation dargestellt wird. Vor dieser Tür steht jedoch der Gegner, der sich nach uns umdreht und seine Feuerstrahlen aktiviert, sobald wir in seiner Reichweite sind. Wird der Spieler vom Feind getroffen erscheint ein Bloodscreen. Ist man unbeschadet auf der anderen Seite der Mauer, kann man Kisten mit Munition einsammeln und anschließend mit Handgranaten auf den Gegner feuern, bis dieser explodiert.

Skripte siehe Skript 7.4 und Skript 7.5.

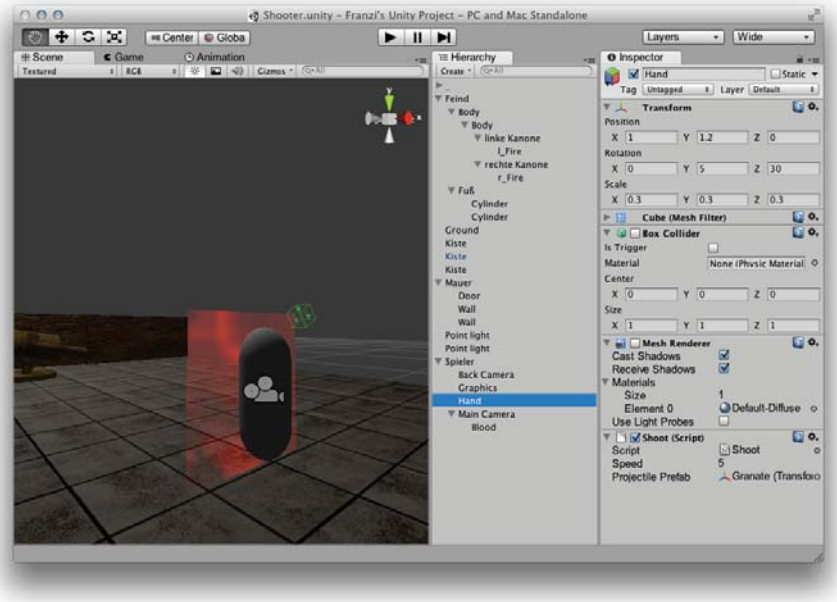


Abbildung 9.14.: Shooter. In der Scene View sieht man den Spieler mit seiner Kamera und dem Bloodscreen, welcher deaktiviert ist, so lang man nicht getroffen wird. Der kleine Würfel oben rechts ist eine imaginäre Hand, von hier werden die Granaten abgefeuert. An das GameObject „Hand“ ist auch das „Shoot“-Skript angehängt.

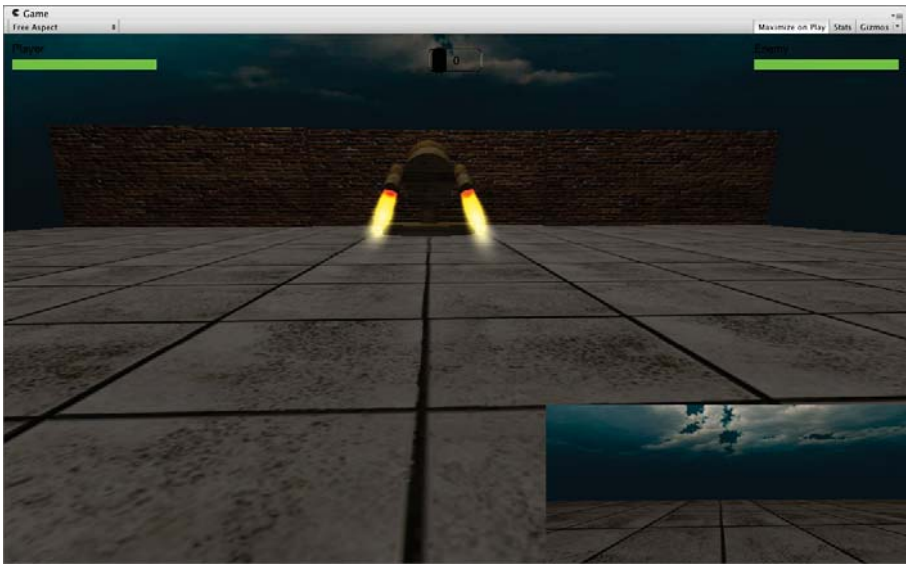


Abbildung 9.15.: Shooter. Der Feind hat den Spieler entdeckt.

Abbildung 9.16.: Shooter. Der Spieler wurde getroffen. Ein Bloodscreen leuchtet auf.

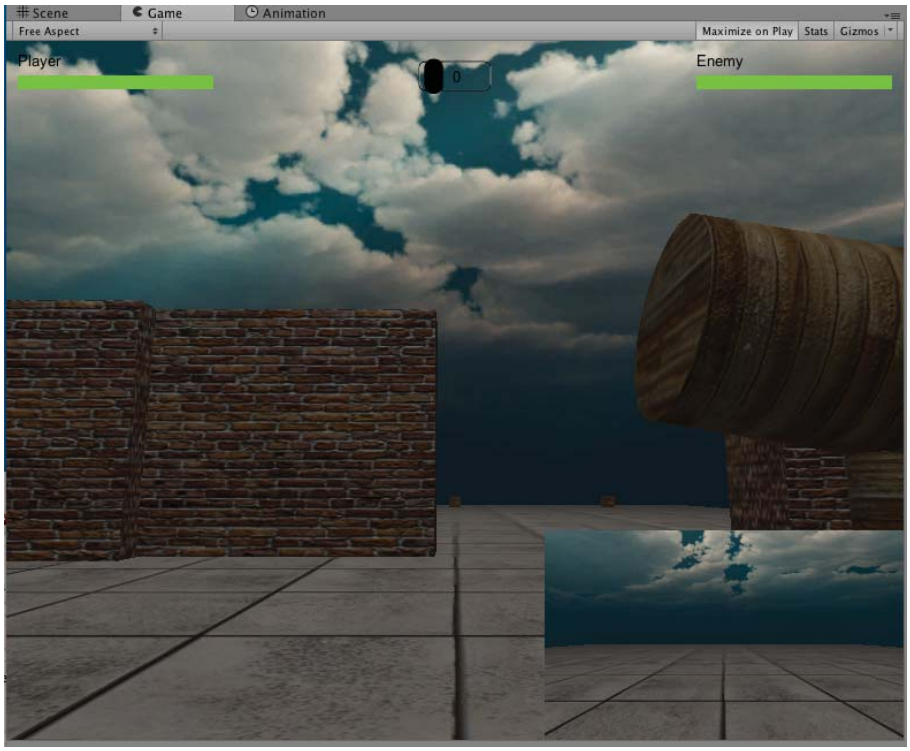


Abbildung 9.17.: Shooter.Der Spieler ist nah genug an der Tür. Diese öffnet sich.

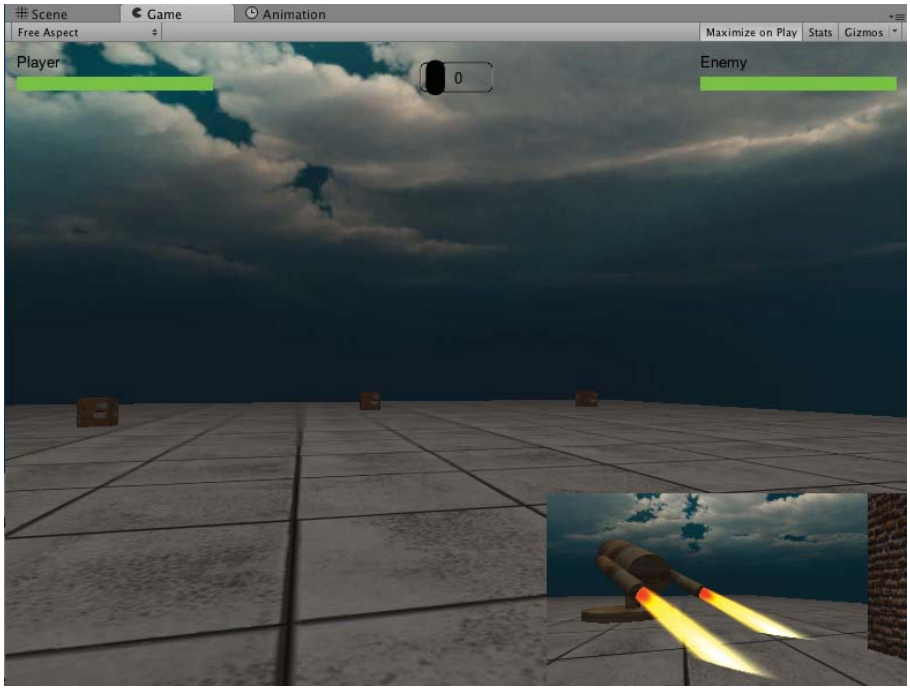


Abbildung 9.18.: Shooter. Der Spieler hat die andere Seite der Mauer erreicht und sieht nun die Munitionskisten. In der Rückensicht des Spielers sieht man, wie sich der Gegner dem Spieler hinterher dreht.

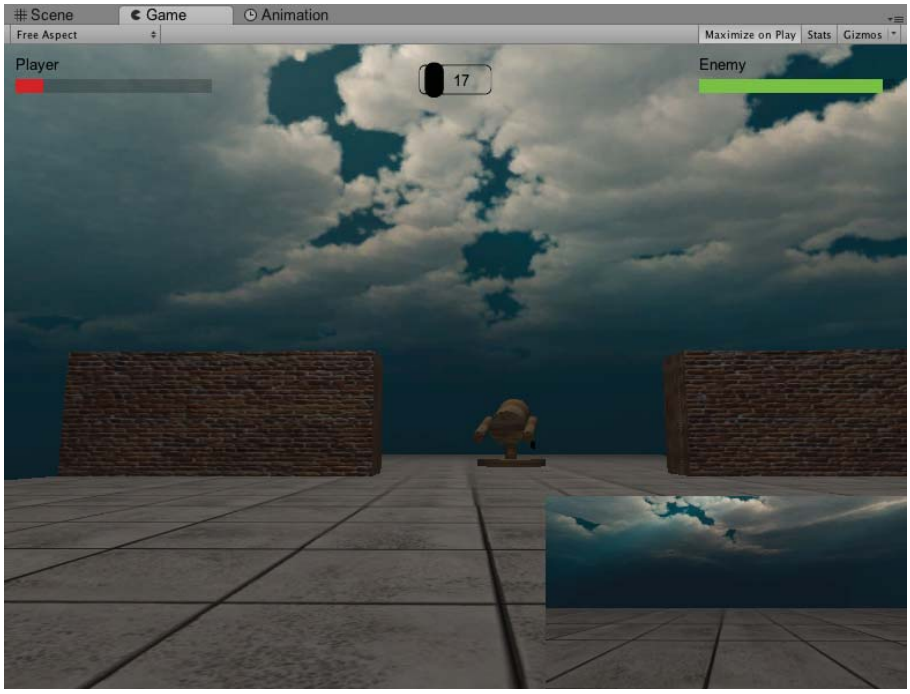


Abbildung 9.19.: Shooter. Der Spieler hat die Munition eingesammelt und eine Granate abgeworfen (schwarzer Punkt beim Gegner). Beim Durchqueren der Mauer wurde er so stark verwundet, dass sein Gesundheitsbalken rot aufleuchtet.

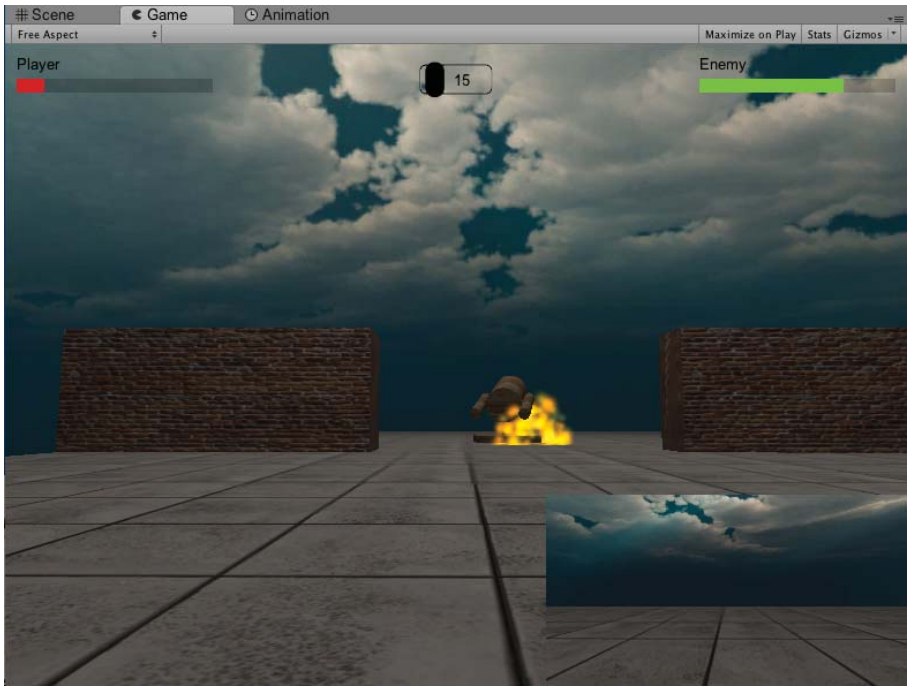


Abbildung 9.20.: Shooter. Die geworfene Granate explodiert.

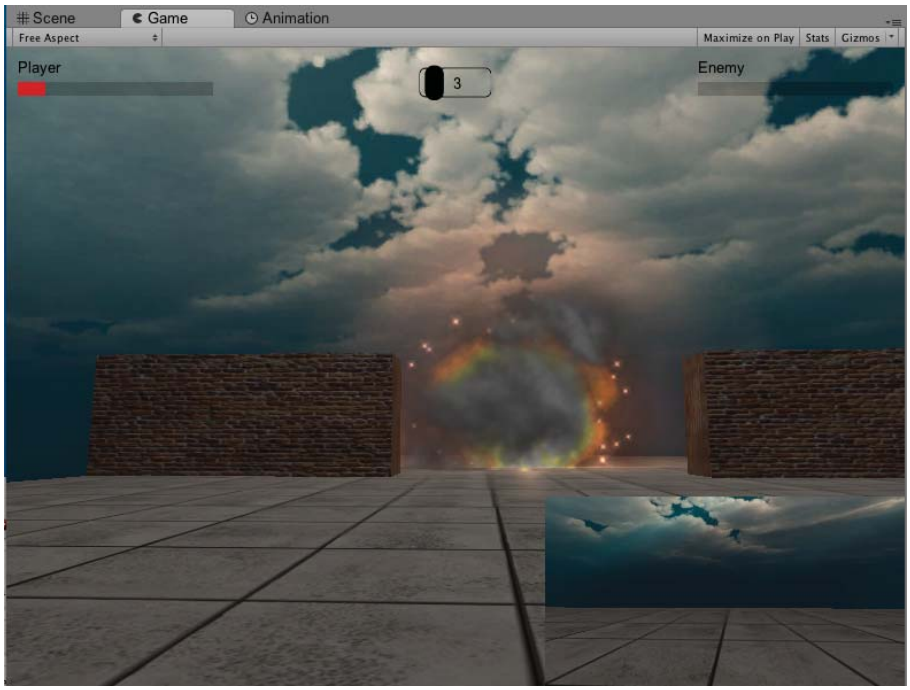


Abbildung 9.21.: Shooter. Der Feind explodiert.

10. Schlussteil

Mit Unity ist es möglich, so gut wie jede 3D Anwendung zu programmieren, die man sich vorstellt. Sollen Objekte im Spiel realistischer sein als nur zusammengesetzte primitive Formen, wie Kugeln, Quader und Zylinder, muss man diese aus einem Programm zur Erstellung von 3D Objekten in Unity importieren.

Es ist sehr einfach mit Unity zu arbeiten und sich in Unity einzuarbeiten, wenn man eine der Programmiersprachen beherrscht. Die mitgelieferten Tools von Unity sind relativ wenig, wodurch man viel Arbeit hat, wenn man eine Anwendung entwickeln will, für die Unity nicht unbedingt gedacht ist. Durch die große Unity Community und den *Asset Store* ist es aber möglich, die benötigten Tools von anderen Benutzern vorgefertigt zu finden. Zusätzlich kann man auch selbst neue Tools zu Unity hinzufügen, was es individuell erweiterbar macht.

Für die meisten großen Hersteller gibt es sicher Engines, die besser auf ihre Bedürfnisse zugeschnitten sind. Wenn man jedoch als Einzelentwickler oder in einer kleinen Gruppe beginnt, ist Unity sehr gut geeignet. Man kann bei sehr komplexen oder außergewöhnlichen Anforderungen natürlich nicht die gleichen Dinge erreichen, als würde man in reinem C++ programmieren. Dafür gehen einige Dinge in Unity aber erheblich schneller. Möchte man als Beispiel eine Anwendung programmieren, die sich durch komplexe physikalische Anforderungen auszeichnet, sollte man dies lieber von Grund auf neu programmieren. Für die normalen physikalischen Anforderungen eines 3D Spiels, reicht die PhysX® Engine, die Unity benutzt, jedoch vollkommen aus und man spart erheblich Zeit, da man nichts neu implementieren muss.

Im Gegensatz zu *Microsofts XNA*, welches wie Unity auf .NET basiert und bei dem man über Visual Studio ebenfalls in C# programmieren kann, bietet Unity den Vorteil des Editors. Dadurch wird die Spielentwicklung intuitiver und ein Teil des Programmieraufwands fällt weg. XNA übernimmt nur die Grundlagen und bietet eine große Klassen- und Toolsammlung [10][9].

Der Vorteil Unity‘s gegenüber dem *Unreal Development Kit (UDK)* von *Epic*, welches auf der *Unreal Engine 3* basiert, ist die Benutzung gängiger Programmier- bzw. Skriptsprachen, wie C# und JavaScript. Beim *UDK* verwendet man die eigene Sprache *UnrealScript*, die an Java angelehnt ist [5]. Andererseits ist hier für Anfänger in der Spielentwicklung ein nicht so hoher Programmieraufwand erforderlich, wie etwa bei Unity, da viele Dinge über

einen entsprechenden Editor realisiert werden können [4].

Unity befindet sich ständig in der Weiterentwicklung und erhält mit jeder Version neue Features, die es noch einfacher und effizienter machen Spiele zu entwickeln. Gerade ist Unity in Version 4 erscheinen. Die wichtigsten neuen Features sind:

- Mecanim: Character Animation
- Ausführbare Programme für Linux erzeugen
- volle Adobe Flash Player Integration
- DirectX 11 Unterstützung
- Schatten für iOS und Android
- Lightmapping mit Normalmaps
- 3D Textur Unterstützung
- Unterstützung von Microsofts Windows 8 und Windows Phone 8

Auch die Unity Community trägt zum Erfolg der Engine bei. Als Anfänger findet man hier Tutorials und Beispielprojekte. Als fortgeschrittener Entwickler trifft man Gleichgesinnte, die einem bei fast jedem Problem bei der Arbeit mit Unity weiterhelfen können.

Ein Nachteil, der auch schon in 7.1 erwähnt wurde, ist die eventuelle Unübersichtlichkeit bei großen Projekten, die anhand der vielen kleinen Skripte entsteht. Durch gute Organisation der Levels kann man dem allerdings entgegenwirken.

Dass man Teile von Unity, wie in 2.4 geschildert, nicht einzeln verwenden kann, sowie fertige Teile und Skripte nicht außerhalb von Unity weiterentwickeln kann, ist ein weiteres Manko.

Abschließend kann man sagen, dass Unity für Anfänger und kleine Gruppen sehr gut geeignet ist. Im professionellen Bereich muss man individuell beurteilen, ob Unity seinen Ansprüchen genügt oder ob man eher eine andere Engine verwenden sollte. Grundsätzlich ist Unity dennoch für Spiele im professionellen Bereich geeignet.

Literaturverzeichnis

[1] Eric5h5, UnityAnswers: *UnityAnswers - General Unity Development*. http://answers.unity3d.com/questions/16808/General-Unity-Development.html. Version: Mai 2012, Abruf: 31.08.2012

[2] Fortress, Holiday: *AppleSMS (Sudden Motion Sensor) plugin*. http://holidayfortress.com/plugins/apple-sms-plugin/. Version: März 2010, Abruf: 31.08.2012

[3] Goldstone, Will: *Unity 3.x Game Development Essentials*. Birmingham : PACKT Publishing Ltd., 2011 (978-1-84969-144-4)

[4] Inc., Epic G.: *Features - Unreal Engine 3 Game Development Tools*. http://www.unrealengine.com/features/. Version: August 2012, Abruf: 31.08.2012

[5] Inc., Epic G.: *Features - UnrealScript*. http://www.unrealengine.com/en/features/unrealscript. Version: August 2012, Abruf: 31.08.2012

[6] Ivanov, Michael: Unity3D Programming. In: *Contech conference* GameIS, 2010, 6. http://de.slideshare.net/sasmaster/unity3d-programming-5725801. Version: November 2010, Abruf: 31.08.2012

[7] Ivanov, Michael: Unity3D Programming. In: *Contech conference* GameIS, 2010, 11. http://de.slideshare.net/sasmaster/unity3d-programming-5725801. Version: November 2010, Abruf: 31.08.2012

[8] Seifert, Carsten: *Neu: Unity 3.1 - Die neuen Features*. http://www.der-softwareentwickler-blog.de/2010/11/17/neu-unity-3-1-dlls-und-der-asset-store/. Version: November 2010, Abruf: 31.08.2012

[9] Wikipedia: *Microsoft XNA — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Microsoft_XNA&oldid=509593163. Version: 2012, Abruf: 31.08.2012

[10] Wikipedia: *XNA (Microsoft) — Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=XNA_(Microsoft)&oldid=107457831. Version: 2012, Abruf: 31.08.2012

Quellen aus der Unity API

[11] UNITY: *Unity API - Application.* http://docs.unity3d.com/Documentation/ScriptReference/Application.html. Version: August 2012, Abruf: 31.08.2012

[12] UNITY: *Unity API - Application.LoadLevel.* http://docs.unity3d.com/Documentation/ScriptReference/Application.LoadLevel.html. Version: August 2012, Abruf: 31.08.2012

[13] UNITY: *Unity API - Application.Quit.* http://docs.unity3d.com/Documentation/ScriptReference/Application.Quit.html. Version: August 2012, Abruf: 31.08.2012

[14] UNITY: *Unity API - Component.GetComponent.* http://docs.unity3d.com/Documentation/ScriptReference/Component.GetComponent.html. Version: August 2012, Abruf: 31.08.2012

[15] UNITY: *Unity API - GameObject.Find.* http://docs.unity3d.com/Documentation/ScriptReference/GameObject.Find.html. Version: August 2012, Abruf: 31.08.2012

[16] UNITY: *Unity API - GUI.* http://docs.unity3d.com/Documentation/ScriptReference/GUI.html. Version: August 2012, Abruf: 31.08.2012

[17] UNITY: *Unity API - GUILayout.* http://docs.unity3d.com/Documentation/ScriptReference/GUILayout.html. Version: August 2012, Abruf: 31.08.2012

[18] UNITY: *Unity API - Input.* http://docs.unity3d.com/Documentation/ScriptReference/Input.html. Version: August 2012, Abruf: 31.08.2012

[19] UNITY: *Unity API - Input.GetAxis.* http://docs.unity3d.com/Documentation/ScriptReference/Input.GetAxis.html. Version: August 2012, Abruf: 31.08.2012

[20] UNITY: *Unity API - Input.GetButton.* http://docs.unity3d.com/Documentation/ScriptReference/Input.GetButton.html. Version: August 2012, Abruf: 31.08.2012

[21] UNITY: *Unity API - Input.GetMouseButton.* http://docs.unity3d.com/Documentation/ScriptReference/Input.GetMouseButton.html. Version: August 2012, Abruf: 31.08.2012

[22] UNITY: *Unity API - Input.GetTouch*. http://docs.unity3d.com/Documentation/ScriptReference/Input.GetTouch.html. Version: August 2012, Abruf: 31.08.2012

[23] UNITY: *Unity API - MonoBehaviour*. http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.html. Version: August 2012, Abruf: 31.08.2012

[24] UNITY: *Unity API - MonoBehaviour.OnCollisionEnter*. http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnCollisionEnter.html. Version: August 2012, Abruf: 31.08.2012

[25] UNITY: *Unity API - MonoBehaviour.OnGUI*. http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnGUI.html. Version: August 2012, Abruf: 31.08.2012

[26] UNITY: *Unity API - MonoBehaviour.OnMouseUp*. http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnMouseUp.html. Version: August 2012, Abruf: 31.08.2012

[27] UNITY: *Unity API - MonoBehaviour.Reset*. http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Reset.html. Version: August 2012, Abruf: 31.08.2012

[28] UNITY: *Unity API - MonoBehaviour.Start*. http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Start.html. Version: August 2012, Abruf: 31.08.2012

[29] UNITY: *Unity API - MonoBehaviour.Update*. http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Update.html. Version: August 2012, Abruf: 31.08.2012

[30] UNITY: *Unity API - Object*. http://docs.unity3d.com/Documentation/ScriptReference/Object.html. Version: August 2012, Abruf: 31.08.2012

[31] UNITY: *Unity API - Object.DontDestroyOnLoad*. http://docs.unity3d.com/Documentation/ScriptReference/Object.DontDestroyOnLoad.html. Version: August 2012, Abruf: 31.08.2012

[32] UNITY: *Unity API - Object.Instantiate*. http://docs.unity3d.com/Documentation/ScriptReference/Object.Instantiate.html. Version: August 2012, Abruf: 31.08.2012

[33] Unity: *Unity API - Physics*. http://docs.unity3d.com/Documentation/ScriptReference/Physics.html. Version: August 2012, Abruf: 31.08.2012

[34] Unity: *Unity API - Physics.Raycast*. http://docs.unity3d.com/Documentation/ScriptReference/Physics.Raycast.html. Version: August 2012, Abruf: 31.08.2012

[35] Unity: *Unity API - PlayerPrefs*. http://docs.unity3d.com/Documentation/ScriptReference/PlayerPrefs.html. Version: August 2012, Abruf: 31.08.2012

[36] Unity: *Unity API - Quaternion*. http://docs.unity3d.com/Documentation/ScriptReference/Quaternion.html. Version: August 2012, Abruf: 31.08.2012

[37] Unity: *Unity API - RaycastHit*. http://docs.unity3d.com/Documentation/ScriptReference/RaycastHit.html. Version: August 2012, Abruf: 31.08.2012

[38] Unity: *Unity API - Time*. http://docs.unity3d.com/Documentation/ScriptReference/Time.html. Version: August 2012, Abruf: 31.08.2012

[39] Unity: *Unity API - Time.deltaTime*. http://docs.unity3d.com/Documentation/ScriptReference/Time-deltaTime.html. Version: August 2012, Abruf: 31.08.2012

[40] Unity: *Unity API - Touch*. http://docs.unity3d.com/Documentation/ScriptReference/Touch.html. Version: August 2012, Abruf: 31.08.2012

[41] Unity: *Unity API - Vector3*. http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html. Version: August 2012, Abruf: 31.08.2012

Quellen aus den Unity Manuals

[42] Unity: *Reference Manual - Built-in Shader Guide*. http://docs.unity3d.com/Documentation/Components/Built-inShaderGuide.html. Version: August 2012, Abruf: 31.08.2012

[43] Unity: *Reference Manual - Character Controller*. http://docs.unity3d.com/Documentation/Components/class-CharacterController.html. Version: August 2012, Abruf: 31.08.2012

[44] Unity: *Reference Manual - Meshes*. http://docs.unity3d.com/Documentation/Components/class-Mesh.html. Version: August 2012, Abruf: 31.08.2012

[45] Unity: *Reference Manual - Particle System (Shuriken)*. http://docs.unity3d.com/Documentation/Components/class-ParticleSystem.html. Version: August 2012, Abruf: 31.08.2012

[46] Unity: *Reference Manual - Shader Reference*. http://docs.unity3d.com/Documentation/Components/SL-Reference.html. Version: August 2012, Abruf: 31.08.2012

[47] Unity: *Reference Manual - Terrain Trees*. http://docs.unity3d.com/Documentation/Components/terrain-Trees.html. Version: August 2012, Abruf: 31.08.2012

[48] Unity: *Reference Manual - Tree Creator Guide*. http://docs.unity3d.com/Documentation/Components/class-Tree.html. Version: August 2012, Abruf: 31.08.2012

[49] Unity: *Reference Manual - Using the Animation View*. http://docs.unity3d.com/Documentation/Components/animeditor-UsingAnimationEditor.html. Version: August 2012, Abruf: 31.08.2012

[50] Unity: *Unity Engine - Importing*. http://unity3d.com/unity/editor/importing. Version: August 2012, Abruf: 31.08.2012

[51] Unity: *Unity Engine - Networking*. http://unity3d.com/unity/engine/networking. Version: August 2012, Abruf: 31.08.2012

[52] UNITY: *Unity Engine - Physics*. http://unity3d.com/unity/engine/physics. Version: August 2012, Abruf: 31.08.2012

[53] UNITY: *Unity Engine - Publishing*. http://unity3d.com/unity/publishing. Version: August 2012, Abruf: 31.08.2012

[54] UNITY: *Unity Engine - Rendering*. http://unity3d.com/unity/engine/rendering. Version: August 2012, Abruf: 31.08.2012

[55] UNITY: *Unity Manual - Behind the Scenes*. http://docs.unity3d.com/Documentation/Manual/BehindtheScenes.html. Version: August 2012, Abruf: 31.08.2012

[56] UNITY: *Unity Manual - Input*. http://docs.unity3d.com/Documentation/Manual/Input.html. Version: August 2012, Abruf: 31.08.2012

[57] UNITY: *Unity Manual - Plugins*. http://docs.unity3d.com/Documentation/Manual/Plugins.html. Version: August 2012, Abruf: 31.08.2012

[58] UNITY: *Unity Manual - Shader*. http://docs.unity3d.com/Documentation/Manual/Shaders.html. Version: August 2012, Abruf: 31.08.2012

[59] UNITY: *Unity Manual - ShaderLab1*. http://docs.unity3d.com/Documentation/Manual/ShaderTut1.html. Version: August 2012, Abruf: 31.08.2012

[60] UNITY: *Unity Manual - ShaderLab2*. http://docs.unity3d.com/Documentation/Manual/ShaderTut2.html. Version: August 2012, Abruf: 31.08.2012

[61] UNITY: *Unity Manual - Using DLL*. http://docs.unity3d.com/Documentation/Manual/UsingDLL.html. Version: August 2012, Abruf: 31.08.2012

[62] UNITY: *Unity Script Reference - Mono Compatibility*. http://docs.unity3d.com/Documentation/ScriptReference/MonoCompatibility.html. Version: August 2012, Abruf: 31.08.2012

[63] UNITY: *Unity Script Reference - Script compilation*. http://docs.unity3d.com/Documentation/ScriptReference/index.Script_compilation_28Advanced29.html. Version: August 2012, Abruf: 31.08.2012

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Franziska Kolbe

Berlin, 4. Februar 2013