Unity-Tutorial: Erste Schritte mit der Game Engine

Spieltrieb

Christian Geiger,
Patrick Pogscheba

Ein Computerspiel sollte vor allem Spaß machen, aber auch einfach zu bedienen und grafisch anspruchsvoll sein. Wer ein solches Spiel entwickeln will, stellt bald fest, dass er Hilfe benötigt. Die bietet unter anderem die Game Engine Unity. Allerdings ist sie selbst so komplex, dass sorgfältige Einarbeitung erforderlich ist. Ein dreiteiliges Tutorial zeigt, was es bei der Entwicklung eines einfachen Spiels alles zu beachten gilt.

Unity ist eine Multiplattform-Spieleumgebung, entwickelt seit 2005 von der dänischen Firma Unity Technologies. Mittlerweile ist das Framework in der Version 4.3.2 verfügbar und gilt als eine der populärsten Spiele-Engines weltweit. Wie kaum eine andere Engine unterstützt Unity den Ansatz der „Demokratisierung der Spieleentwicklung“. Ziel des Herstellers ist es, möglichst vielen Nutzern ein Werkzeug zur Entwicklung interaktiver 3D-Inhalte zur Verfügung zu stellen. Dazu bietet Unity auf hoher Abstraktionsebene die notwendige Funktion an, unterstützt viele Hardwareplattformen (Desktop, Konsolen, mobile Geräte) und garantiert eine für freie Entwickler bezahlbare Preis- und Lizenzpolitik.

Dadurch ist die Game Engine sowohl bei professionellen Entwicklungsstudios als auch bei freien Entwicklern und Hobbyprogrammierern sehr beliebt. Hochschulen, an denen mittlerweile spezielle Studiengänge im Bereich Digitale Medien und Spieleentwicklung stark nachgefragt werden, nutzen Unity ebenfalls. Inzwischen gibt es mehr als zwei Millionen registrierte Nutzer (Quelle: http://unity3d.com/company/public-relations/, siehe „Alle Links“), mindestens 400 000 Entwickler sind laut Hersteller monatlich aktiv und der Unity Webplayer ist 225 Millionen Mal installiert worden.

Einfacher Einstieg und komplexe Funktionen

Ein Grund dafür ist sicherlich der einfache Einstieg in die 3D-Welt durch eine integrierte Entwicklungsumgebung mit gut benutzbaren Werkzeugen, die der Anwender über spezielle Plug-ins wie Playmaker sogar vollständig visuell ohne Kenntnis einer Programmiersprache einsetzen kann. Gleichzeitig lassen sich in Unity komplexe Funktionen wie Augmented Reality, fortgeschrittene Render-Verfahren oder aufwendige Simulationsberechnungen integrieren und als einfach wiederzuverwendende Komponenten zur Verfügung stellen.

Wie sich die Arbeit mit Unity konkret gestaltet, soll ein dreiteiliges Tutorial zeigen. Es entwickelt Schritt für Schritt eine kleine spielerische Applikation. Dieser erste Teil stellt zunächst die grundlegenden Konzepte der Game Engine vor. Danach folgt anhand detaillierter Beschreibungen die Entwicklung des Jump-and-Run-Spiels, das die zwei folgenden Teile kontinuierlich ausbauen. Auf grundlegende Aspekte der Computergrafik kann dieser

Dreiteiler dabei nicht eingehen. Dazu sei auf weiterführende Literatur verwiesen. Grundlegende Programmierkenntnisse in C# werden ebenfalls vorausgesetzt.

Neben den genannten technischen Vorteilen erklärt die Lizenz- und Preispolitik der Dänen den hohen Marktanteil von Unity. In der „Indie-Version" für Desktop-Plattformen ist die Game Engine kostenlos, lässt sich aber trotzdem für kommerzielle Projekte nutzen. Die Pro-Version, die viele zusätzliche Funktionen wie 3D-Texturen, Effekte für Full Screen Postprocessing, Level of Detail, weiche Schatten, 3D-Texturen und Videostreaming ermöglicht, kostet für die gängigen PC-Plattformen (Windows, Mac OS, Linux) 1500 US-$. Die für größere Projekte wichtige Unterstützung lokaler und räumlich getrennt arbeitender Teams durch Asset Cache Server und Versionierungskontrolle bietet der Hersteller für 500 US-$ pro Arbeitsplatz an.

Hoher Marktanteil auch durch Preispolitik

Unity-Projekte lassen sich typischerweise ohne größere Hürden für mobile Endgeräte bereitstellen, eine Pro-Lizenz für Android, iOS oder BlackBerry kostet ebenfalls jeweils 1500 US-$. Verzichtet man auf die Pro-Funktionen und das Streaming von Assets auf die Endgeräte sind die Basic-Varianten für die genannten mobilen Plattformen wie die „Indie"-Version kostenlos. Windows Phone Pro ist sogar in der Pro-Version enthalten.

Erheblich teurer hingegen gestaltet sich zum Teil die Entwicklung für Spielkonsolen, da deren Hersteller bisher hohe Anforderungen an Entwickler stellen. Mit dem Erscheinen neuer Konsolengenerationen wie Wii U, Xbox One und PS4 ändert sich diese Situation teilweise, und Nintendo, Microsoft und Sony werben verstärkt um einzelne Entwickler und kleine Studios.

Kochrezept für die Entwicklung mit Unity

Die folgende Auflistung der wichtigsten Schritte bei der Erstellung eines Unity-Spiels sollen einen kurzen Überblick darüber geben, wie Entwickler vorgehen. Dabei sind einzelne Schritte im Verlauf der meist iterativen Entwicklung auch mehrmals durchführbar.

Vorab ist es aus Sicht der Autoren sinnvoll, auf der Website des Herstellers die zu verschiedenen Themen umfangreich zur Verfügung gestellten Informationen zu lesen. Empfehlenswert ist zunächst die direkte Übersicht zu Unity, die einige Beispiele enthält (siehe „Alle Links").

Anschließend sollte man sich in den Workflow einlesen und sich bei Interesse weiterführende Informationen zu den verschiedenen Werkzeugen verschaffen. Eine umfangreiche Referenz ist das Unity-Benutzerhandbuch.

Wer nun selbst in die 3D-Entwicklung unter Unity einsteigen will, muss Unity zunächst laden und installieren. Danach hilft bei jedem neuen Projekt ein „Kochrezept":

- Erzeuge ein neues Unity-Projekt mit dem Project Wizard.
- Importiere die notwendigen Assets wie 3D-Modelle, Texturen, Klänge et cetera.
- Erzeuge für jedes Spielelevel eine eigene 3D-Szene.
- Platziere für jede Szene die passenden Assets als Szenenobjekte.
- Stelle die Parameter der *Main Camera* passend ein und positioniere sie.
- Füge Lichtobjekte hinzu beziehungsweise konfiguriere das Umgebungslicht.
- Weise den Objekten passende Materialien und weitere Eigenschaften zu, beispielsweise Physik, Animationen oder Kollisionserkennung.
- Schreibe notwendige Scripts und weise sie den Objekten zu.
- Teste den Ablauf der Applikation im Editor.
- Publiziere die fertige Applikation auf der gewünschten Plattform.

Registrierte Wii-U-Entwickler erhalten eine spezielle Version von Unity Pro für die Konsole kostenlos. Microsoft bewirbt sein Independent Developer Programm für die Xbox One ebenfalls mit einer freien Unity-Pro-Lizenz.

Neuartige Ein- und Ausgabegeräte wie Oculus Rift, Leap Motion, Sixense Razer/Stem oder Virtuix Omni bieten eine Anbindung an Unity, damit 3D-Entwickler ihre Produkte schnell akzeptieren. Ein weiterer Grund für die Beliebtheit dieser Entwicklungsumgebung ist der Asset Store, ein digitaler Marktplatz für unterschiedliche Unity-Inhalte. Viele Entwickler stellen hier ihre Lösungen für spezielle Anforderungen vor oder bieten einen einfachen Zugriff auf erstellte 3D-Modelle, Animationen oder Texturen. Die Inhalte dieser Assets reichen von komplexen 3D-Szenen (beispielsweise Big Environment Pack) über KI-Algorithmen zum Finden von Wegen (etwa A* Pathfinding Project) bis zu komplexen Werkzeugen/Plug-ins für MotionCapturing mit Kinect, visuelle Spieleentwicklung (beispielsweise Playmaker) oder HDR image-based lighting (etwa Skyshop).

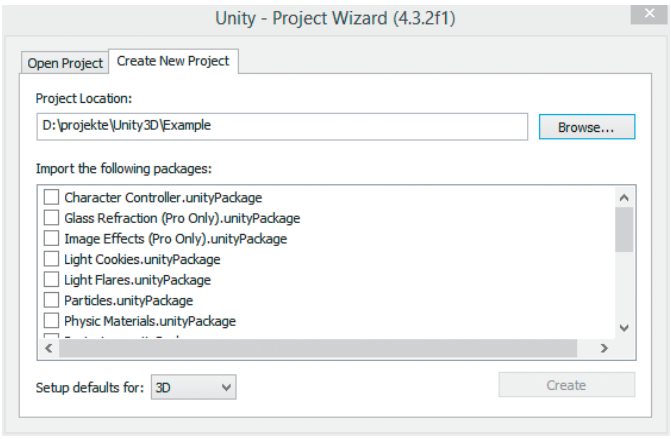
Durch einen schnellen Einkauf der meist zwischen 10 und 100 US-$ teuren Assets kann man eine Menge Zeit sparen, indem man die existierenden Inhalte nutzt. Dabei erweist sich die integrierte Nutzerbewertung als bewährtes Werkzeug, um die Spreu vom Weizen zu trennen. Wer trotz guten Produkts keine ausreichende Dokumentation oder gar eine Demoszene zur Verfügung stellt, nur selten neue Versionen seiner Software anbietet oder nicht auf Anfragen der Käufer antwortet, erhält schnell eine kritische Bemerkung der anspruchsvollen Community. Ein späterer Teil dieses Tutorials wird eine Übersicht ausgewählter, aus Sicht der Autoren lohnender Produkte im Asset Store geben.

Immer auf der Höhe der Zeit bleiben

Unity Technologies entwickelt die Game Engine ständig weiter und integriert regelmäßig neue oder verbesserte Techniken. In der im August 2013 erschienenen Version 4 waren dies vor allem das neuartige Animationssystem Mecanim und fortgeschrittene Render- und Shading-Techniken sowie die Unterstützung für

iX-TRACT

- Unity ist eine weitverbreitete Game Engine, die sich unter anderem für anspruchsvolle 3D-Spiele eignet und in der „Indie"-Version frei verfügbar ist.
- Der Hersteller entwickelt die Engine kontinuierlich weiter und integriert fortgeschrittene Render- und Shading-Techniken ebenso wie aktuelle Animationssysteme.
- Für das Editieren von Quellcode – unterstützt werden C# sowie zwei Scriptsprachen – wird die Entwicklungsumgebung MonoDevelop mitgeliefert, andere Editoren lassen sich aber bei Bedarf einbinden.



Beim ersten Start von Unity erscheint der Project Wizard (Abb. 1).

Linux. Die Version 4.3, das Ende 2013 erschienene kleinere Update, weist neben vielen Detailverbesserungen zwei zentrale Neuerungen auf.

Verbesserter Umgang mit Animationen: Seit der Version 4 bietet Unity mit Mecanim ein mächtiges Animationssystem, das alle notwendigen Arbeiten direkt in der Game Engine ermöglicht. Dabei lassen sich importierte Animationen auf andere 3D-Figuren anwenden (retargeting). Komplexe Animationen erstellt der Anwender in der Regel einzeln in externen Werkzeugen als Keyframe-Animation oder nimmt sie mit MotionCapturing-Systemen direkt auf und importiert sie anschließend in eine Game Engine. Unity 4.3 bietet mit Blend Trees und Zustandsautomaten die Möglichkeit, einzelne Clips miteinander zu kombinieren und so eine große Anzahl unterschiedlicher Animationen zu erstellen. Über spezielle Plug-ins im Asset Store wie Mixamo hat der Anwender zudem Zugriff auf eine riesige Bibliothek vordefinierter Bewegungen und kann ein 3D-Face-Tracking in Echtzeit durchführen.

2D Engine included: Bisher war Unity als reine 3D-Engine konzipiert. Es gab aber seit Jahren Projekte, die reine 2D-Applikationen waren, und Produkte im Asset Store (etwa 2DToolkit), die eine einfache Erstellung von 2D-Inhalten ermöglichen, sind Verkaufsschlager. Unity hat diese Ideen nun in der Version 4.3 selbst als neuen 2D-Workflow umgesetzt und ermöglicht sprite-basierte 2D-Animationen sowie die effiziente Kombination von 3D- und 2D-Inhalten.

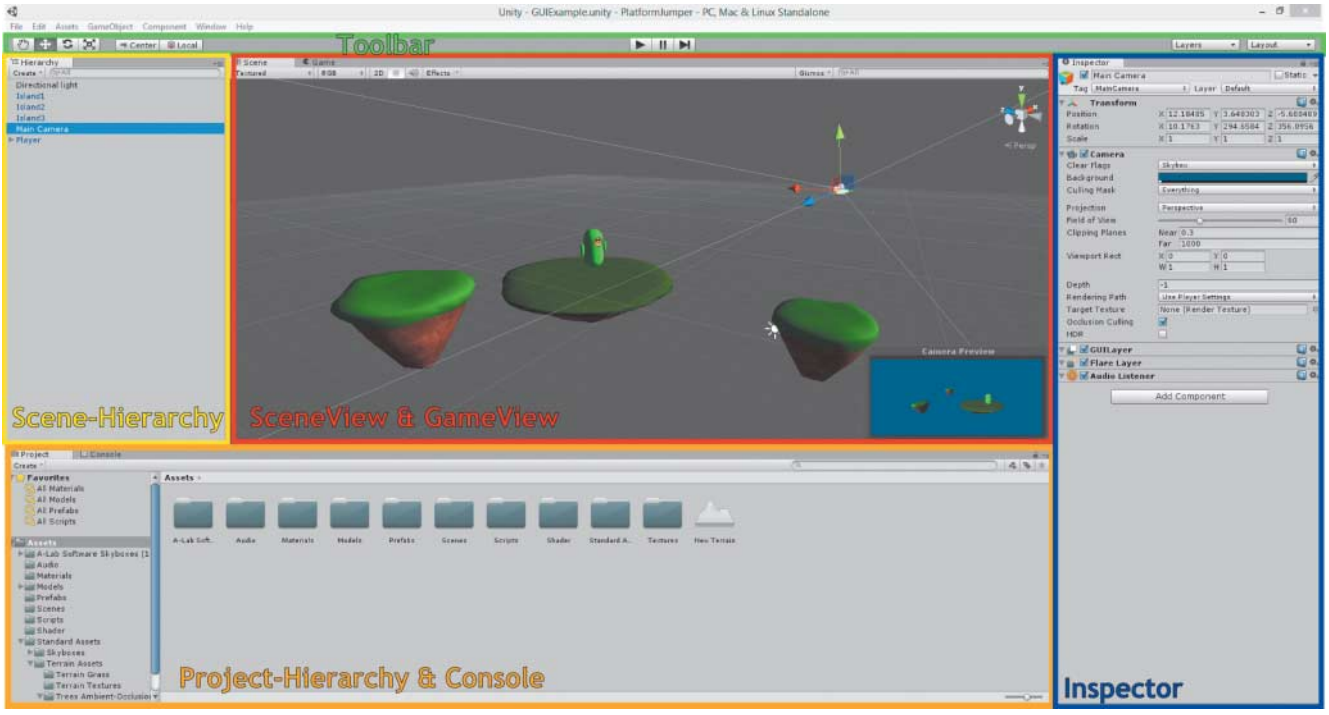
Im Prinzip stellt Unity einen 3D-Level-Editor zur Verfügung, in dem der Benutzer existierende Objekte importiert, in einer Szene arrangiert und einzelne Elemente mit weiteren Eigenschaften versieht. Dieses Vorgehen ist vergleichbar mit der Arbeitsweise gängiger 3D-Werkzeuge wie Maya oder 3ds Max, auch wenn Unity nur wenige Modellierungs- oder Texturierungsfunktionen bereitstellt. Der Schwerpunkt des Werkzeugs liegt auf der Entwicklung komplexer Scripts sowie der Konfiguration einzelner Elemente in der Szene.

Kurz erklärt: Grundkonzepte der Unity Engine

Eine Unity-Applikation besteht aus folgenden Elementen:

Project: Eine zu entwickelnde Unity-Anwendung (etwa ein Spiel) wird durch den Project Wizard in einem eigenen Projekt organisiert und in einem speziellen Verzeichnis gespeichert.

Scene Graph: Eine Unity-Anwendung besteht aus einer Anzahl verschiedener Szenen (vgl. Spiele-Level), die der Entwickler einzeln in der Engine erstellen kann. Eine Szene besteht aus einer Menge von Elementen (*GameObjects*), die hierarchisch in einem Szenengraphen angeordnet sind. Bei Objekten, die in einer Eltern-Kind-Beziehung stehen (Unity nennt das Parenting), vererben Objekte in höheren Ebenen („Eltern“) ihre Position und Orientierungsinformationen an da-



Die Benutzeroberfläche der Game Engine präsentiert dem Entwickler die wichtigsten Informationen auf einen Blick. Individuelle Anpassungen sind möglich (Abb. 2).

runterliegende „Kind“-Objekte. Der Szenengraph ist in der 3D-Computergrafik ein mächtiges Konzept zur Strukturierung von Szenen und erscheint in der Unity-Umgebung in einem eigenen Fenster.

GameObjects: Sie sind das zentrale Konzept in Unity, da jedes benötigte Element in einer Anwendung als *GameObject* definiert ist. Zunächst handelt es sich dabei nur um einen Container ohne eigentliche Funktion. Aus programmiertechnischer Sicht stellt er die Basisklasse aller Entitäten in Unity dar.

Components: Die benötigte Funktion der *GameObjects* stellen Komponenten bereit. Je nach Objekt können verschiedene Komponenten – etwa Audioquellen, Animationen, Lichter, Texturen oder Partikelsysteme – in einem *GameObject* existieren. Eine der wichtigsten Komponenten ist *Transform*, da jedes *GameObject* erst durch sie seine Position und Orientierung in der 3D-Szene definiert und diese an andere *GameObjects* im Szenengraphen weitergibt. Jedes *GameObject* besitzt eine solche *Transform*-Komponente.

Scripting: Das Verhalten der *GameObjects* beschreiben die in Unity vordefinierten Komponenten, die bereits recht mächtig sind. Für darüber hinausgehende Funktionen nutzt man den speziellen Komponententyp *Script*. Er unterstützt die Programmierung in C#, UnityScript (eine an JavaScript angelehnten Sprache) sowie Boo (eine .NET-Sprache mit Python-ähnlicher Syntax). Da ein *Script* eine Komponente ist, muss man es nach seiner Erstellung mit einem *GameObject* verbinden, damit es aktiv wird.

Prefabs: *Prefabs* sind ein weiteres wichtiges Konzept in Unity, um existierende Lösungen wiederzuverwenden. Dabei handelt es sich um einfach zu erstellende spezielle *GameObjects*, die mehrfach wiederverwendbar sind und im Prinzip das Original für eine Reihe geklonter Objekte darstellen. Instanziiert man ein solches *Prefab*, erzeugt man eine Kopie, die man wie ein normales *GameObject* in seiner Szene verwendet. Alle Instanzen (oder besser Kopien) sind dabei mit dem ursprünglichen *Prefab*-Objekt verknüpft. Änderungen am Original leitet Unity automatisch an alle Instanzen weiter.

Assets: Unter Assets verstehen Entwickler normalerweise alle digitalen Medieninhalte wie Grafiken, 3D-Modelle, Musikdateien et cetera. Wie bei anderen Game Engines ist in Unity ein effizienter und flexibler Asset Workflow besonders wichtig, da Entwickler viele Medieninhalte mit externen Werkzeugen erstellen und importieren. Unity bietet hier den direkten Zugriff auf viele Industriestandards für 3D-Modellierung, Texturen und Audio- und Videodateien. Ein Vorteil ist dabei, dass die Engine auf die meisten Assets nativ zugreift. Verändert man mit dem externen Werkzeug den Inhalt, sieht man dies direkt in der Applikation. Neben reinen Medieninhalten betrachtet Unity auch externe *Prefabs* als Assets. Unitys Asset Store ist ein umfangreicher Marktplatz für solche *Prefabs* sowie klassische *Assets*.

Auf geht's: Erste Schritte in der Engine

Beim ersten Start von Unity erscheint der Project Wizard (Abbildung 1). Hier kann der Anwender entweder bereits erstellte Projekte aus einer Liste auswählen (im Tab „Open Project“) oder neue erstellen („Create New Project“). Im zweiten Fall wählt man zunächst ein Projektverzeichnis aus (unter Windows über den Menüpunkt „Browse …“, auf anderen Plattformen kann es in der Menüführung leichte Unterschiede geben). Weiterhin besteht die Möglichkeit, Assets aus Unitys Standardrepertoire zu importieren. Dazu gehören vor allem verschiedene Effekte (sowohl als Shader als auch als Postprocessing) und nützliche Scripts.

Das Tutorial-Beispiel startet zunächst mit einem leeren Projekt (passend benannt als *iXUnityTutorial*). Auf die Auswahl zusätzlicher Assets verzichtet es. Nach dem Erstellen eines Projekts landet der Benutzer im Default-Layout der Benutzeroberfläche (Abbildung 2). Die wichtigsten Bereiche sind

– die Szenen-Hierarchie, die alle *GameObject*s in einer Art Szenengraph darstellt (linke Spalte),

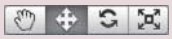
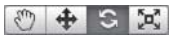
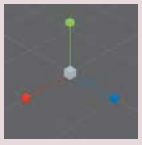
– die Rendering-Bereiche (mittlere Spalte) für das Editieren und Manipulieren der Szene (SceneView, Tab Scene), wie es aus den üblichen 3D-Modellierungswerkzeugen bekannt ist, sowie dem GameView (Tab Game, mittlere Spalte),

– der Inspector, in dem ein gewähltes *GameObject* mit all seinen Komponenten dargestellt und konfiguriert wird (rechte Spalte),

– der Projekt-Explorer (ProjectView) mit allen Scripts, Materialien, Prefabs und sonstigen Dateien, die der Anwender durch Import oder Kopieren im Unterordner *Assets* des Projektordners abgelegt hat (Tab Project), und die Konsole, auf der die Engine Debug-Informationen, Warnungen und Fehler ausgibt.

– Die Toolbar enthält fünf Kontrollelemente, mit denen der Entwickler bestimmte Elemente steuern kann: Trans-

Manipulatoren

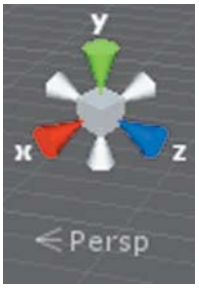
 Translation	 oder Taste „W“	Änderung der Position eines Objekts entlang der Achsen (Pfeile) beziehungsweise entlang der zwischen den Hauptachsen aufgespannten Ebenen (Flächen im Ursprung)
 Rotation	 oder Taste „E“	Änderung der Objektausrichtung (ein Ring pro Achse, äußerer Ring zur Rotation um die Kameraachse, 2. Ring von außen zur freien Rotation um den Up- beziehungsweise Side-Vektor der Editorkamera)
 Skalierung	 oder Taste „R“	Größenänderung entlang der einzelnen Achsen (Würfel am Ende der Achsen) oder uniform in alle Richtungen (Würfel im Ursprung)

form Tools (SceneView), Transform Gizmo Toggles (SceneView Gizmos), Ablaufkontrollen (GameView), Layer-Auswahl (SceneView) sowie Layout-Auswahl (Editor).

Layout der Oberfläche individuell anpassen

Das Default-Layout der Unity-Oberfläche kann der Benutzer individuell anpassen, indem er die Bereiche in ihrer Größe ändert oder über den Tab verschiebt. Auch ein Abdocken der Fenster ist möglich, sodass er sie beliebig auf dem Bildschirm platzieren kann. Über den Menüpunkt „Window -> Layouts“ oder über die Toolbar-Schaltfläche „Layouts“ kann er andere Layout-Presets laden oder eigene speichern. Hier muss jeder seine eigenen bevorzugten Ansichten herausfinden, manchmal ist es sinnvoll, sowohl den Szenen-Editor als auch das Game-Window gleichzeitig zu sehen (zum Beispiel „2 by 3“) oder aber alle Seitenansichten (Top, Side, Front in orthografischer Perspektive) im Blick zu haben, um Objekte besser ausrichten zu können.

Ist das richtige Layout eingestellt (die Autoren preferieren ein abgewandeltes „Wide“-Layout, da dort ein großer Szenen-Editor zur Verfügung steht), kann man mit der Bearbeitung einer kleinen Szene beginnen, um die einzelnen Oberflächen ein wenig besser kennenzulernen.



Das Scene Giszmo hilft beim Einstellen der Perspektive (Abb. 3).

In jeder neuen Szene erzeugt Unity automatisch ein Kameraobjekt, das als *Main Camera* betitelt ist und mit dem sich das Tutorial zunächst noch nicht befasst. Über den Menüpunkt „GameObject -> Create Other -> Cube“ erzeugt das Beispiel ein *GameObject* mit einer Würfelgeometrie. Es erscheint als ausgewähltes Objekt direkt im Inspector, der Szenenhierarchie und im Szenen-Editor. Im Inspector wird zunächst die Komponente *Transform* genauer betrachtet, die weiteren Komponenten werden im späteren Verlauf des Tutorials vorgestellt.

Jedes *GameObject* besitzt eine *Transform*-Komponente, die im Wesentlichen Funktionen und Variablen zur Verfügung stellt, um ein Objekt im Raum zu positionieren und in seiner Größe zu ändern. Der Inspector stellt den Positionsvektor, die Eulerwinkel für die Rotation sowie den Skalierungsvektor im lokalen Koordinatensystem des Objekts zur Bearbeitung zur Verfügung (die globale Transformation ergibt sich durch die Verkettung einzelner Objekt-Transformationen in der Szenenhierarchie, Stichwort: Szenengraph). Intern verwendet Unity jedoch Quaternionen zur Darstellung von Rotation, da Eulerwinkel nicht immer eindeutig sind.

Die Werte im Inspector lassen sich entweder per Tastatur oder durch ein Ziehen der über dem Feld-Label gedrückten linken Maustaste nach links/unten oder rechts/oben ändern.

Zusätzlich dazu zeigt der Szenen-Editor an der Objektposition einen Manipulator, mit dem man die Transformation des Objekts direkt in der Szene ändern kann. Diese visuellen Hilfssysteme im Szenen-Editor und im GameView heißen in der Unity-Terminologie Gizmos.

Es gibt drei Formen des Manipulators (siehe „Manipulatoren“), die entweder im lokalen oder im globalen Koordinatensystem eines Objekts agieren. In der Toolbar (Gizmo Display Toggles) befindet sich die Schaltfläche „Local“ zum Umschalten des Manipulators zwischen lokalem und globalem Koordinatensystem. Die einzelnen Manipulatoren kann der Anwender entweder über die Manipulatorschaltflächen (Transform Toggles) oder über die Tastatur anwählen. Bei gedrückter Strg-Taste kann er ein Snapping aktivieren, das er über den Menüpunkt „Edit -> Snap Settings …“ einstellen kann.

Die genaue Ausrichtung von Objekten erfordert, dass man sie aus verschiedenen Positionen betrachten kann. Dazu bietet Unity mehrere Navigationstechniken. Zum einen gibt es in der oberen rechten Ecke des Szenen-Editors das Scene Gizmo (Abbildung 3), das sowohl die aktuelle Rotation der Szenenkamera visualisiert, als auch eine Änderung der Kamera bewirken kann, indem diese an den einzelnen Achsen im globalen Koordinatensystem ausgerichtet wird. Außerdem ist eine Änderung der Projektionsart zwischen perspektivisch und orthogonal (ohne perspektivische Verzerrung) mit diesem Gizmo möglich.

Perspektiven und Navigation im Szenen-Editor

Zum anderen ist eine Navigation mit Maus und Tastatur die wohl am häufigsten genutzte Technik (siehe „Navigation …“). Wer dieses Tutorial aktiv weiter verfolgen will, sollte die verschiedenen Navigationstechniken nun ausprobieren und verinnerlichen, um im Weiteren sicher damit umgehen zu können.

Im nächsten Schritt wird der erstellte Würfel gewählt und ein Reset der Transformation durchgeführt. Das geschieht, indem man rechts auf die Komponente *Transform* im Inspector drückt und den Punkt „Reset“ auswählt. Der Würfel liegt nun im Nullpunkt der Szene. Jetzt wird er (über Manipulator oder Inspector) ein Stück auf der x-Achse bewegt (zwei bis drei Einheiten).

Anschließend wird eine Kugel erstellt („GameObject -> Create Other -> Sphere“) und über die Szenenhierarchie auf den Würfel gezogen. Die Position der Kugel im Szenen-Editor bleibt gleich, jedoch hat sich die lokale Position im Inspector aufgrund der erstellten Hierarchie verän-

dert. Lässt man den Würfel nun um die y-Achse rotieren, rotiert auch die Kugel, allerdings erfolgt die Rotation nicht um den Manipulator, der sich im Mittelpunkt der Geometrien befindet, sondern um den Pivot-Punkt des Würfels. (In Modellierungsprogrammen kann man den Pivot-Punkt eines Objektes definieren. Er entspricht dem Dreh- und Angelpunkt und ist der Ausgangspunkt für alle folgenden Transformationen.) Über die Toolbox kann ein Klick auf die Schaltfläche „Center“ den Manipulator in den Pivot-Punkt verschieben, was die Rotation verständlicher macht. Durch die Erstellung einer Hierarchie bleibt die relative Position der Kugel zum Würfel immer gleich.

Die Entstehung eines interaktiven Spiels

Nachdem die wichtigsten Grundlagen zum Umgang mit der Unity-Benutzeroberfläche nun erläutert sind, geht es weiter zum nächsten Teil. Die Erstellung eines kleinen interaktiven Spiels soll weitere Einblicke in die Unity-Welt gewähren. Das Ziel ist ein kleines Plattform-Spiel, in dem eine Figur, die der Benutzer aus einer Third-Person-Perspektive sieht, Gegenstände sammeln kann. Dazu muss sie Hindernisse überwinden und darf nicht zu tief fallen oder zu viel Zeit verstreichen lassen. GUI-Elemente sollen zusätzlich Start- und End-Screens einblenden.

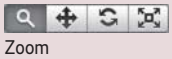
Die beiden nächsten Teile des Tutorials bauen das Spiel weiter aus. Wer die einzelnen Schritte nachvollzieht, sollte am Ende eigenständig mit Unity umgehen können. Natürlich können drei Zeitschriftenartikel nicht alle Gebiete dieser Game Engine behandeln, aber immerhin in die wichtigsten Grundlagen einführen. Dazu zählen Scripting, Character Control, Camera Control, Kollisionserkennung und -verarbeitung, Audio, GUI, einfache Animation und Physik.

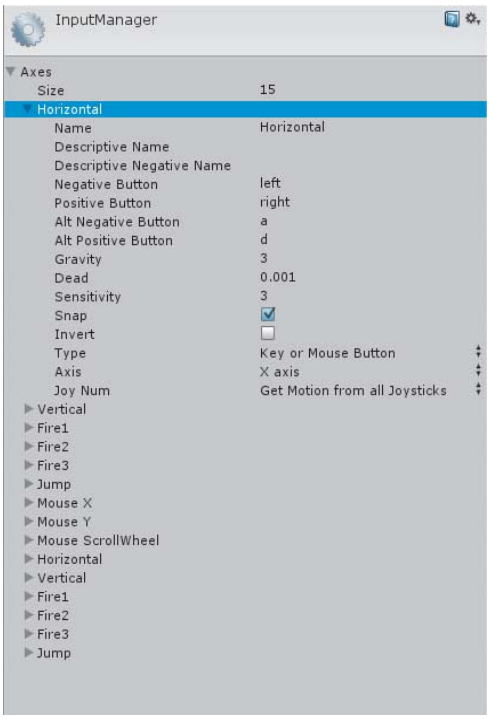
Für Texturen und Sounds greift das Tutorial vorwiegend auf freie Assets zu. Bei Bedarf und je nach eigenen Fähigkeiten kann man sie natürlich auch selbst erstellen. Zu Anfang behandelt das Tutorial möglichst alle Einzelheiten im Detail, im späteren Verlauf werden die Instruktionen reduziert.

Der erste Schritt ist die Erstellung einer neuen Szene, der der Menüpunkt „GameObject -> Create Other -> Plane“ einen Boden hinzufügt. Ein Klick mit der rechten Maustaste auf „Transform -> Reset“, bewirkt, dass diese *Plane* im Ursprung liegt. Standardmäßig hat sie eine Seitenlänge von zehn Einheiten. Für ausreichend Platz wird sie in alle Richtungen mit dem Faktor 1000 skaliert, was ein wenig zu groß ist, aber im weiteren Verlauf Vorteile bringen wird. Da sich die *Plane* im Spielverlauf nicht bewegen soll, kann man rechts oben im Inspector die Option „Static“ anwählen, damit man Optimierungen der Szene durchführen kann (Stichworte: Batching, Occlusion Culling). Die *Plane* – oder besser die Rendering-Komponente (*MeshRenderer*) – erhält das Unity-Standardmaterial *Default-Diffuse*, dessen Farbe weiß ist.

Als Nächstes geht es an die Erstellung eines neuen Materials. Dazu wird im ProjectView ein neuer Ordner für Materialien angelegt (man sollte ein Projekt von Anfang an gut strukturieren, indem verschiedene Arten von Assets in ihren eigenen Ordnern abgelegt werden). Dies kann durch einen Rechtsklick auf den Ordner *Assets* in der Baumstruktur des ProjectView erfolgen. Dadurch öffnet sich ein Kontextmenü, in dem der obers-

Navigation mit Maus und Tastatur

 Move	1. linke Maustaste (bei gewähltem Hand-Tool, kein Manipulator aktiv); 2. Alt-Taste und mittlere Maustaste (Manipulator kann aktiv sein)
 Orbit	Alt-Taste und linke Maustaste (Hand-Tool oder Manipulator aktiv)
 Zoom	1. Strg-Taste und linke Maustaste (bei aktivem Hand-Tool); 2. Mausrad; 3. Alt-Taste und rechte Maustaste (Manipulator kann aktiv sein)
Focus	Mit der Taste „F“ kann der Anwender das gewählte Objekt in der Bildmitte zentrieren und zur vollen Größe aufziehen. Dabei muss die Maus sich über dem Szenen-Editor befinden.
Walking	Mit den Pfeiltasten lässt sich die Szenenkamera vorwärts oder seitwärts ohne Änderung der Ausrichtung über das Gitternetz bewegen. Shift beschleunigt die Bewegung.
Flythrough	Bei gedrückter rechter Maustaste kann der Anwender ähnlich wie in einem Ego-Shooter mit den Tasten „W“, „A“, „S“, „D“, „Q“ und „E“ (vorwärts, seitwärts links, hoch, rückwärts, seitwärts rechts und runter) entlang der Kameraausrichtung navigieren. Shift beschleunigt die Bewegung.



Über den *InputManager* kann der Entwickler die vordefinierten Achsen und Buttons individuell einstellen (Abb. 4).

te Punkt *Create* ausgewählt wird. Hier bietet sich die Möglichkeit, Scripts, Materialien, Shader und andere Assets anzulegen. Die Beispielanwendung wählt zunächst den Punkt *Folder*. Wie in einem normalen Dateimanager entsteht im rechten Teil des ProjectView ein Ordner, der direkt in *Materials* umbenannt werden kann. Ein Doppelklick wechselt in diesen Ordner, und auf die gleiche Art (oder mit einem Rechtsklick in den rechten Teil des ProjectView) wird ein neues Material erstellt, das die Bezeichnung *Floor* erhält.

Dieses Material erscheint direkt im Inspector. Hier kann man zunächst einen Shader-Typ auswählen, das Beispiel bleibt jedoch beim voreingestellten Shader namens *Diffuse*, der eine diffuse Reflexion des Lichts bewirkt (nach Lambert). Über den Punkt *Main Color* kann der Benutzer die Farbe über einen Klick auf das Farbfeld (es öffnet sich ein Farbdialog) oder über den Klick auf die Pipette (Colorpicker; der nächste Klick wählt die Farbe unter dem Maus-Cursor aus) einstellen. Das Beispiel wählt zunächst ein sattes Grau. Per Drag & Drop kann der Anwender das erstellte Material auf die *Plane* im SceneView oder in der Szenenhierachie ziehen. Im Prinzip lassen sich alle Assets per Drag & Drop aus dem ProjectView in der Szene platzieren.

Tastaturkürzel unter Windows

Bearbeiten (bei selektiertem *GameObject*)	
Strg+D	Duplizieren
Strg+C/Strg+V/ Strg+X	Kopieren/Einfügen/ Ausschneiden
Entfernen	Löschen
Bearbeiten (allgemein)	
F	Objekt ins Bild einpassen (zentrieren)
Strg+A	alles auswählen
Ablaufkontrolle	
Strg+P	Play (Szene starten)
Strg+Shift+P	Pause
Strg+Alt+P	Einzelschritt
Dateiverwaltung	
Strg+N	neue Szene
Strg+O	Szene öffnen
Strg+S	Szene speichern
Strg+Shift+S	Szene speichern als ...
Manipulation	
Strg+linke Maustaste	Snapping
V	Vertex Snapping
Q	Pan View (Handtool)
W	Translation
E	Rotation
R	Skalierung
Z	Wechsel zwischen Pivot und Center
X	Wechsel zwischen Lokal und Global

Eine weitere Komponente auf dem Boden ist der *MeshCollider*, der es ermöglicht, Kollisionen mit der *Plane* zu erkennen. Ein *MeshCollider* bildet die Kollisionsgeometrie anhand des zugrunde liegende Körpers, hier eine Ebene, und ist die präziseste, jedoch rechenintensivste Version eines Collider. Nur durch die Aktivierung des Flags *Convex* kann ein *MeshCollider* auch mit einem anderen *MeshCollider* kollidieren. Neben jeder Komponente im Inspector ist ein kleines Buch mit einem Fragezeichen abgebildet, über das der Benutzer direkt die Dokumentationsseiten von Unity aufrufen kann.

Nach den ersten Änderungen in der Szene sollte man speichern. Dazu wird unter *Assets* ein Ordner *Scenes* erstellt, in dem die Szene gespeichert wird. Regelmäßiges Speichern erspart viel Frust, wenn Unity einmal abstürzen sollte.

Der statischen Szene Leben einhauchen

Nachdem das Spielfeld hergerichtet ist, kann es an die Erstellung der Figur gehen. Der Spieler soll sie per Maus- und Tastatureingaben in einer Third-Person-Perspektive durch die Szene navigieren. Hier kommt ein essenzieller Teil der Game Engine zum Tragen: das Scripting. Nur damit kann der Entwickler einer statischen Szene Leben und Logik einhauchen.

Doch zunächst erzeugt man wiederum die nötigen Geometrien. Als Erstes wird ein neues *GameObject* – in diesem Fall eine *Capsule* – samt Material erstellt, etwa einem schönen Grün, und passend benannt. Diesen *Player* positioniert man nun so, dass er knapp über der *Plane* steht (y>1), und fügt über die Schaltfläche „Add Component -> Physics“ die Komponente *Character Controller* hinzu.

Der *Character Controller* ist eine spezielle Form des Collider (oder auch des *Capsule Collider*, da dieselbe Geometrie zugrunde liegt) speziell für Spielfiguren. Hier kann der Entwickler definieren, wie tief der Collider in einen anderen eindringen kann (*Skin Width*, etwa um ein Feststecken zu vermeiden) und welche Steigungen (*Slope Limit*) oder Stufen (*Step Offset*) die Figur überwinden kann.

Nun ist es an der Zeit, ein erstes Script zu erstellen. Dazu wird unter *Assets* zunächst ein Ordner *Scripts* angelegt und dort über „Rechtsklick -> Create“ ein C#-Script hinzugefügt, das den Namen *ThirdPersonController* erhält und das direkt auf den Player gezogen wird. Durch einen Doppelklick auf das Script im ProjectView oder durch einen Rechtsklick auf das Script im Inspector und anschließende Auswahl von „Edit Script“ öffnet sich die Entwicklungsumgebung MonoDevelop.

Steuern über die *Input*-Klasse

In dieser IDE können Entwickler alle Codes in Unity bearbeiten, seien es Scripts oder Shader. Natürlich lassen sich auch andere Editoren verwenden, von einfachen Texteditoren bis hin zu anderen Entwicklungsumgebungen wie Visual Studio. Die gewünschte Einstellung kann unter Windows über „Edit -> Preferences -> External Tools -> External Script Editor“ erfolgen.

Ein Unity-Script ist abgeleitet von der Klasse *MonoBehavior* und nach dem Erstellen bereits mit den Funktionen *Start* (zum Initialisieren) und *Update* (zum Aktualisieren des Zustands) ausgestattet. Als Erstes implementiert das Tutorialbeispiel eine einfache Steuerung, indem es die Eingabeklasse *Input* verwendet. Mit ihr lassen sich Maus, Tastatur und definierbare Achsen (Axis, etwa Vertical, Horizontal) und Knöpfe (Buttons, etwa Fire1, Jump) abfragen.

Über den InputManager („Edit -> Project Settings -> Input“, Abbildung 4) kann

Listing 1: Update-Funktion mit Vergleich zwischen gefilterter und ungefilterter Eingabe

```
1  void Update () {
2    float forward = Input.GetAxis("Vertical");
3    float forwardRaw = Input.GetAxisRaw("Vertical");
4    Debug.Log("gefiltert: " + forward + "\tgefiltert: " + forwardRaw);
5  }
```

Listing 2: Erste Version des *ThirdPersonController*-Skripts

```
6  using UnityEngine;
7  using System.Collections;
8
9  [RequireComponent(typeof(CharacterController))]
10 public class ThirdPersonController : MonoBehaviour {
11
12    CharacterController characterController;
13    public float rotationSpeed = 60.0f;
14    public float walkingSpeed = 8.0f;
15    // Use this for initialization
16    void Start () {
17      characterController = this.GetComponent<CharacterController>();
18    }
19
20    // Update is called once per frame
21    void Update () {
22      float forward = Input.GetAxis("Vertical");
23      float rotate = Input.GetAxis("Horizontal");
24      float strafe = Input.GetAxis("Strafe");
25
26      transform.Rotate(Vector3.up, rotate * rotationSpeed * Time.deltaTime);
27      Vector3 direction = this.transform.right * strafe + this.transform.forward * forward;
28      characterController.SimpleMove(direction * walkingSpeed);
29    }
30  }
```

der Entwickler alle vordefinierten Achsen und Buttons individuell einstellen sowie eigene definieren. Dies kann man direkt tun, indem man die Achse *Horizontal* markiert und per Strg-D dupliziert (alternativ über die Maus „Duplicate Array Element“). Die neue Achse erhält den Namen *Strafe*, und die Tasten *Q* und *E* werden als *Negative* beziehungsweise *Positive Button* definiert. Die alternativen Buttons (*Alt* …) werden gelöscht.

Die *Update*-Funktion des Script fragt zunächst die horizontale, vertikale und die seitwärtige (*strafe*) Achse ab, die sich auch über die Pfeiltasten oder die Tasten WASD steuern lässt. Dazu kann man die Member-Funktionen *GetAxis* (geglättet mit einem Tiefpassfilter, keine harten Sprünge) und *GetAxisRaw* (ungefiltert) nutzen. (Listing 1 zeigt die *Update*-Funktion des Script mit einem Vergleich der beiden Werte.) Die Funktion *Debug.Log* gibt sie auf die Unity-Konsole aus.

Tutorialinhalt

Teil 1: Aufbau einer einfachen Szene und Interaktion mit ihr

Teil 2: Spiellogik, erweiterte Interaktion, GUI

Teil 3: Aufbau eines kompletten Levels (Plattformen, Platzieren von Gegenständen, Münzen), Integration aller Teile, Spielablauf mit mehreren Szenen (Startbildschirm), Effekte

Nach dem Start der in Listing 2 zu sehenden Szene über den Play-Button wechselt die Ansicht zum GameView, wobei das Drücken der Tasten W oder S (der Fokus muss im GameView sein) eine entsprechende Ausgabe bewirkt. In der Statusleiste sieht der Entwickler die letzte Ausgabe der Konsole, alternativ kann er die ganze Konsole über einen Tab im Bereich des ProjectView öffnen. Diese beiden Werte dienen dazu, den *CharacterController* vorwärts laufen beziehungsweise um seine y-Achse rotieren zu lassen. Dazu muss das Script die Komponente *CharacterController* zur Verfügung stellen. Dies geschieht über die Funktion *GetComponent* der Klasse *MonoBehavior* (entweder als generische Funktion oder mit dem Komponentennamen als Parameter). Um sicherzugehen, dass es eine solche Komponente auf dem *GameObject* gibt, sollte das Script zu Beginn das Attribut *RequireComponent* nutzen. Damit kann ein *GameObject* eine Komponente nicht mehr einfach löschen. Sie wird mit dem Script automatisch hinzugefügt.

Die Funktionen *Move* oder *SimpleMove*, die sich in ihrer Komplexität unterscheiden, steuern die Bewegung des *CharacterController*. *SimpleMove* wendet automatisch eine Gravitationskraft an und unterdrückt jegliche Anteile in Richtung der y-Achse. Für die Beispielanwendung genügt das zunächst, im späteren Verlauf kommen ein paar Änderungen hinzu und es wird auf die Methode *Move* gewechselt.

$$\Delta s = v \cdot \Delta t\,,\left[m = \left(\frac{m}{s}\right) \cdot s\right],$$
$$\Delta \alpha = \omega \cdot t$$

Beziehung zwischen Geschwindigkeit und Weg: Wegänderung ist gleich Geschwindigkeit multipliziert mit der Zeitveränderung, analog bei der Winkeländerung (Abb. 5).

Über die Methode *Rotate* der *Transform*-Komponente erfolgt die Drehung der Spielefigur. Zur Steuerung der Geschwindigkeit werden zwei Variablen (als *public*) angelegt, eine für die Bewegungsgeschwindigkeit (*walkingSpeed*) und eine für die Drehgeschwindigkeit (*rotationSpeed*). Sie erhalten einen vernünftigen *default*-Wert (soll m/s beziehungsweise grad/s entsprechen). Öffentliche Variablen zeigt der Inspector an, und der Entwickler kann sie dort direkt ändern.

Als Parameter erhält *Rotate* die Rotationsachse (*Vector3.up* für die globale y-Achse) sowie den Rotationswinkel, der sich aus der Eingabeachse, der Drehgeschwindigkeit sowie der Zeit seit dem letzten Aufruf der *Update*-Funktion (*Time.deltaTime*) zusammensetzt, damit eine von der Frame-Rate unabhängige Bewegung entsteht (eine geschwindigkeitsabhängige Weg- beziehungsweise Winkeländerung ist immer bezogen auf die betrachtete Zeit, Abbildung 5).

Wo Licht ist, ist auch Schatten

Die Richtung, in der sich die Figur bewegen soll, berechnet sich aus der Addition der mit der jeweiligen Eingabe (Wert zwischen 0 und 1) gewichteten, aktuellen Vorwärts- beziehungsweise Seitwärtsausrichtung (*this.transform.forward* beziehungsweise *this.transform.right*) der *Transform*-Komponente. Ihre Speicherung erfolgt in der *Vector3*-Variablen. Diese Richtung wird wie bei der Rotation mit der Bewegungsgeschwindigkeit multipliziert und an die Funktion *SimpleMove* übergeben. Da diese bereits die zeitlichen Abhängigkeiten berücksichtigt, muss sie nicht mehr explizit berechnet werden.

Beim Start der Szene sieht der Betrachter sie aus Sicht der *Main Camera*. Alles scheint ein wenig zu dunkel zu sein, offensichtlich fehlt noch ein Licht. Im Szenen-Editor verwendet Unity standardmäßig eine automatische Beleuchtung, solange eine Anwendung nicht explizit ein Licht erzeugt hat.

Der Hintergrund ist nun blau, was in den Kameraeinstellungen (Inspector) über Background geändert werden kann. Steuert der Spieler die Figur nun über WASD, bewegt sie selbst sich zwar, aber die Kamera bleibt an Ort und Stelle. Auch eine Rückmeldung über eine Drehung der Figur fehlt, die Kapsel sieht von allen Seiten gleich aus. Da der Boden ebenfalls keine Änderungen aufweist, wird eine Navigation schwierig.

Doch da kann Abhilfe geschaffen werden. Zunächst zum Licht: Es wird ein Directional Light erzeugt (wie gehabt, „GameObject -> Create Other -> Directional Light“), da es am besten für die Gesamtbeleuchtung der Szene geeignet ist. Es hat keine Bindung an einen Ort, nur der Einfallswinkel ist ausschlaggebend. Diesen kann man über die Rotation so wählen, dass eine gute Ausleuchtung der Figur gewährleistet ist. Später kann man weitere Lichter hinzufügen, um die Szene besser auszuleuchten.

Licht ist auch ein dramaturgisches Gestaltungsmittel, das Spieleentwickler an bestimmten Stellen bewusst stark oder nur schwach einsetzen. Zu viele Lichtquellen sind rechenintensiv, deshalb kann man sie teilweise in eine statische Umgebung hineinrendern (Light Baking), um die Performance zu steigern. Zwischendurch kann man immer wieder das Spiel starten, um die Änderungen zu begutachten.

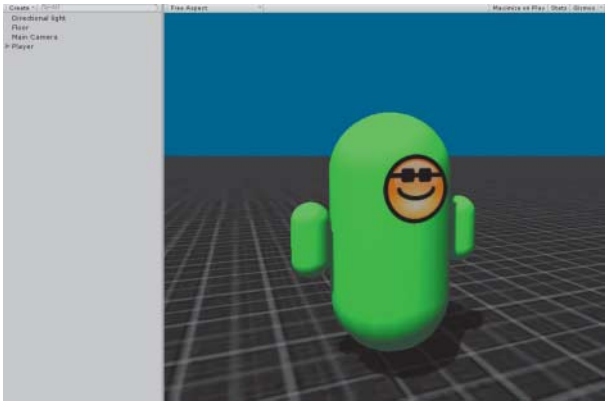
Für eine bessere räumliche Wahrnehmung ist die Verwendung von Schatten wichtig. Da Unity ab Version 4.2 Echtzeit-Schatten für ein direktionales Licht berechnen kann, nutzt das Tutorial diese Möglichkeit, um dem Spiel einen besseren räumlichen Eindruck zu verpassen. Die Game Engine unterscheidet zwischen harten Schatten (Hard Shadows) und solchen mit einem weichen Verlauf im Anfangsbereich (Soft Shadows). Die freie Version erlaubt nur harte Schatten.

Bei selektiertem Licht kann der Entwickler unter *Shadow Type* den Schattentyp auswählen. Die Stärke und Auflösung kann er hier ebenfalls verändern. Im Szenen-Editor ist der Schatten sofort zu sehen.

Das grüne Männchen bekommt Profil

Als nächstes soll die Figur ein wenig Struktur bekommen, damit auch Drehungen besser zu sehen sind. Dazu wird zunächst eine weitere *Capsule* erzeugt, die dieselbe Farbe erhält wie der *Player* und in der Szenenhierarchie auf den *Player* gezogen wird. Das neue Objekt wird so positioniert und skaliert, dass es einem Arm entsprechen könnte (vor allem am Anfang ist es für Einsteiger mit wenig Erfahrung wichtig, einiges an Positionen und Skalierungen auszuprobieren, damit sie ein Gefühl für die Modellierung bekommen). Sinnvollerweise benennt man das neue Objekt gleich passend. Durch die Tastenkombination Strg+D (oder über den Punkt „Duplicate“ des Kontextmenüs) kann man es duplizieren und anschließend an die Position des anderen Arms setzen (einfach ein Minus vor den x-Wert setzen). Bitte auch hier die Namensänderung nicht vergessen.

Mittlerweile könnte man meinen, hier entstehe ein kleines Android-Männchen. Damit es nicht zu ungewollten Kollisionen und Störungen des *CharakterController* kommt, werden die automatisch hinzugefügten *Collider* entfernt. Damit der *Player* ein wenig mehr Persönlichkeit bekommt, wird als Nächstes ein Gesicht konstruiert, allerdings ohne eine komplizierte Texturierung des ganzen Körpers. Dazu erzeugt die Beispielanwendung eine *Plane* mit dem Namen *Face* (oder ähnlich), die ebenfalls in der Hierarchie unter den *Player* gezogen wird. Durch geschickte Transformation sollte man sie so verändern, dass sie die Größe eines Gesichts hat und an der passenden Stelle sitzt (in Richtung des blauen Pfeils der Figur, forward, am Manipulator). Sollte die Ebene nach einer Rotation nicht mehr zu sehen sein, liegt das daran, dass die Rückseite in Richtung



Nicht länger ein Android-Männchen: Die fertige Spielefigur hat ein Gesicht bekommen (Abb. 6).

Listing 3: Script *ThirdPersonCameraController*

```
31 using UnityEngine;
32 using System.Collections;
33
34 public class ThirdPersonCameraController : MonoBehaviour {
35
36   public Transform target;
37   public float distance;
38   public float height;
39
40   // Use this for initialization
41   void Start () {
42     if(target == null)
43     {
44       this.enabled = false;
45     }
46   }
47
48   // Update is called once per frame
49   void LateUpdate () {
50
51     this.transform.position = target.position +
       target.transform.TransformDirection(new Vector3(0,height,-distance));
52     this.transform.LookAt(target);
53   }
54 }
```

Listing 4: Erweiterung von *ThirdPersonCameraController*

```
55 public float rotationSpeed  = 1.0f;
56 Quaternion rotationOffset = Quaternion.identity;
57 Vector3 lastMousePosition;
58
59 // Update is called once per frame
60 void LateUpdate () {
61   if (Input.GetMouseButtonDown(1))
62   {
63     lastMousePosition = Input.mousePosition;
64   }
65   else if (Input.GetMouseButton(1))
66   {
67     Vector3 delta = Input.mousePosition - lastMousePosition;
68     rotationOffset *= Quaternion.AngleAxis(delta.x *
                          rotationSpeed * Time.deltaTime, Vector3.up);
69     lastMousePosition = Input.mousePosition;
70   }
71   this.transform.position = target.position +
72     target.transform.TransformDirection(rotationOffset *
                                  new Vector3(0,height,-distance));
73   this.transform.LookAt(target);
74 }
```

Kamera zeigt und aufgrund des Backface Culling nicht gerendert wird.

Die Textur für das Gesicht findet man im Internet. eine Bildersuche nach „free smiley“ liefert viele nette Gesichter, die für das Projekt infrage kommen, beispielsweise unter http://www.freestockphotos.biz/stockphoto/16633 (am besten ist eine Public Domain oder Creative-Commons-Lizenz). Es bietet sich eine ganze Smiley-Palette an, da eventuell eine Änderung des Gesichtsausdrucks während des Spiels möglich sein soll. Die Autoren haben sich für die SVG-Version entschieden, aus der sie mit Inkscape einen Smiley als PNG-Datei (128 × 128 Pixel) exportiert haben. Das unterhalb von *Assets* gespeicherte Bild wird wie gehabt per Drag & Drop auf die *Plane* für das Gesicht gezogen (Abbildung 6).

Die Kamera richtig positionieren

Nachdem die Figur ein wenig mehr Profil erhalten hat, kann es darangehen, dass die Kamera dem grünen Männchen bei einer Rotation folgt. Doch zunächst sei noch angemerkt, dass sich die Bewegung der Figur über den Inspector feinjustieren lässt, indem man die einzelnen Geschwindigkeiten für Rotation beziehungsweise Translation ändern; so lässt sich über die Erhöhung der Rotationsgeschwindigkeit zum Beispiel ein kleinerer Wendekreis einstellen.

Der erste einfache Schritt ist ein Script mit dem Namen *ThirdPersonCameraController*, das die öffentlichen Variablen *target* vom Typ *Transform* sowie *distance* und *height* vom Typ *float* definiert. Die *Start*-Funktion fragt ab, ob das *target* gesetzt ist und deaktiviert das Script im Fehlerfall. Da zunächst alle Objekte die Zeit haben sollen, ihren Zustand zu ändern, nutzt das Script für die Änderung der Kameraposition die Funktion *LateUpdate*. Hier werden die Position der Kamera auf die Position des Ziels geändert und ein Offset addiert, das aus der Höhe und dem Abstand im lokalen Koordinatensystem resultiert (Listing 3, Zeile 51). Dazu dient die Funktion *Transform.TransformDirection*, die einen Vektor in das Koordinatensystem des *GameObject* transformiert. Anschließend bewirkt die Funktion *Transform.LookAt*, dass die Kamera genau auf die Figur gerichtet ist.

Nach dem erneuten Starten des Spiels kann der Anwender die Figur navigieren und sieht sie immer von hinten. Das Drücken der rechten Maustaste soll nun die Kamera um die Spielfigur rotieren lassen. Dazu muss die Mausposition beim ersten Drücken gespeichert und im Anschluss eine Differenz aus aktueller und letzter Mausposition berechnet werden. Eine Variable *rotationOffset* vom Typ *Quaternion* (zur Darstellung von Rotationen) wird anhand der Differenz in der x-Achse und einer Rotationsgeschwindigkeit wie schon bei der Bewegung des *CharacterController* geändert (durch Multiplikation mit der Änderung in Form einer *AngleAxis*-Rotation). Zusätzlich beeinflusst der Rotations-Offset die Position der Kamera (Multiplikation lokaler Position mit dem Rotations-Offset, Listing 4).

Nun lässt sich die Figur bewegen und von allen Seiten betrachten. Über die mittlere Maustaste sollte noch ein Reset des Rotations-Offset erfolgen (mit *Quaternion.idendity* gleichsetzen). Alle verwendeten Dateien sind einzeln sowie als Unity-Package mit komplettem Projekt über den *iX*-Listingservice verfügbar (siehe „Alle Links“).

Ausblick

Dieser erste Tutorialteil hat die Grundlagen von Unity und dessen Benutzeroberfläche vorgestellt. Er sollte klar gemacht haben, wie man eine einfache Szene aufbaut und mit ihr interagieren kann. Manipulationen der Transformationsmatrizen sollten keine großen Mühe mehr bereiten.

Der nächste Teil bearbeitet die bisher erstellten Assets weiter und erklärt, wie die Spielfigur Gegenstände einsammeln kann, wie sich Audiosequenzen einfügen sowie fortgeschrittene Scripts und Spiellogiken integrieren lassen. Außerdem werden die ersten Elemente eines Spielmenüs und einer Statusanzeige eingebaut. (ka)

Prof. Dr. Christian Geiger

lehrt und forscht an der FH Düsseldorf im Bereich Mixed Reality und Visualisierung.

Patrick Pogscheba

arbeitet als wissenschaftlicher Mitarbeiter unter anderem im Bereich Mixed Reality im Fachbereich Medien an der FH Düsseldorf.

Literatur

[1] Sue Blackmann; Beginning 3D Game Development with Unity 4; All-in-One, multi-platform game development; Second Edition; Apress, 2013

[2] Penny de Byl; Holistic Game Development with Unity; An All-In-One Guide to Implementing Game Mechanics, Art, Design, and Programming; Focal Press, 2011

Alle Links: www.ix.de/ix1402036