

Skript zum Wahlpflichtmodul

# Grundlagen der Spieleentwicklung 2D/3D

© Prof. Dipl.Ing. Jirka Dell'Oro-Friedl  
V2.1, HFU 2017



# Inhaltsverzeichnis

1 Einleitung	3	7 Animation	16	13 Spielereingaben	38
1.1 Lernziele	3	7.1 Timeline-Animation in Flash	17	13.1 Keyboard- / MouseEvents in Flash	38
1.2 Beschränkung auf die IDE	3	7.2 Timeline-Animation in Unity	20	13.2 Input-Klasse von Unity	39
1.3 Umfang	3	8 Skripte	22	14 Kollisionen	40
2 IDE	4	8.1 Konsole	22	14.1 hitTest in Flash	40
2.1 Zeitleiste (Timeline)	4	8.2 Skripte in Flash	23	14.2 Collider in Unity	41
2.2 Szenenbaum, Displaylist	4	8.3 Skripte in Unity	24	14.3 Raycasts	43
2.3 Szene	4	8.4 Datentypen	26	15 Akustik	44
2.4 Projekt	5	8.5 Vektor-Methoden und -Eigenschaften	26	15.1 Flash-Sounds	44
2.5 Grafikfunktionen	5	8.6 Eigenschaften von Transformationen	26	15.2 Audio-Sources und -Listener	45
2.6 Überblick Flash (CC2015-IDE)	6	8.7 Anweisungen zur Transformation	27	16 Virtuelles User Interface (GUI)	46
2.7 Überblick Unity5-IDE	7	9 Logische und geometrische Beziehungen	28	16.1 Flash Components	46
3 Mathematische Grundlagen	8	9.1 Szenenbaum und Displaylist	28	16.2 Unity UI-Elements	47
3.1 Vektor	8	9.2 Aggregation	29	17 Externe Daten	49
3.2 Koordinatensystem	8	9.3 Ereignisse und Nachrichten	29	17.1 Import in Flash	49
3.3 Lokale Koordinatensysteme	9	9.4 Klassenhierarchie	29	17.2 Import in Unity	50
3.4 Transformation	9	9.5 Flash	29	17.3 Text parsen	50
3.5 Konsequenz	10	9.6 Unity	31	18 Lokale Datenspeicherung	51
4 Logischer Aufbau	11	10 Instanziierung zur Laufzeit	32	18.1 Local Shared Object in Flash	51
4.1 Klassenhierarchie in Flash	11	10.1 Symbole und DisplayObjects	32	18.2 PlayerPrefs in Unity	52
4.2 Komponentenmodell in Unity	12	10.2 GameObjects und Prefabs	32	19 Server-Kommunikation	53
5 Erstellung visueller Elemente	13	11 Observer: Events und Messages	34	19.1 Flash	54
5.1 Vektorgrafik mit Flash	13	11.1 Custom-Events in Flash	34	19.2 Unity	54
5.2 Primitive Meshes in Unity	13	11.2 Messages in Unity	35	20 PeerToPeer-Kommunikation	55
6 Symbole und Basis-Koordinatensysteme	14	12 Zeitverhalten	36	20.1 Adobe Cirrus	56
6.1 Flash-Symbole	14	12.1 EnterFrame und Timer in Flash	36	20.2 Unity Master Server	57
6.2 Unity-Prefabs	15	12.2 Update in Unity	37	21 Anhang	58
6.3 Ordner	15			21.1 Unity Class Hierarchy (excerpt)	58
				21.2 Flash Class Diagramm	59
				21.3 Links	60

# **1 Einleitung**

## **1.1 Lernziele**

Folgende Lernziele werden im Modul „Spielentwicklung“ verfolgt:

- die Kenntnis grundlegender Techniken und Entwurfsmuster der Entwicklung hoch interaktiver Anwendungen und Spiele.
- die Befähigung, einfache Spielprototypen in kurzer Zeit in 2D mit Flash oder 3D mit Unity erstellen zu können.
- die Befähigung, schnell Testszenarien zu erstellen, um Anwendungskonzeption voranzutreiben und zu prüfen.
- den sicheren Umgang mit und die geschickte Nutzung der integrierten Entwicklungsumgebungen (IDE) von Flash und Unity.
- die Vertiefung der Programmierung in ECMA-Script (Javascript) – Dialekten.
- die Befähigung hierarchische Strukturen und Kopplungen von Spielementen zu erkennen und geschickt aufzubauen zu können.

## **1.2 Beschränkung auf die IDE**

Flash und Unity werden mit einer mächtigen IDE geliefert, welche diese Werkzeuge gerade für die Entwicklung von Prototypen und kleinen Spielen besonders interessant macht, da

- zur Entwurfs- und Entwicklungszeit visuell in der Szene gearbeitet werden kann.
- keine weiteren Programme erforderlich sind und gelernt werden müssen (bis auf Audioeditoren falls erforderlich).
- schon mit grundlegenden Programmierkenntnissen Anwendungen erstellt werden können, die nicht einem Standardschema folgen müssen.

Daher werden im Modul „Spieleentwicklung“ ausschließlich diese IDEs genutzt, um zunächst deren Potential untersuchen und entfalten zu können.

## **1.3 Umfang**

Die Vorlesung und dieses Skript bieten einen kompakten Einstieg in die Spieleentwicklung im Allgemeinen und im Besonderen mit Flash und Unity. Die Inhalte werden dabei nicht in aller Tiefe oder vollständig behandelt. Im Vordergrund steht die Vermittlung eines umfangreichen Verständnisses und eine Orientierung in den Möglichkeiten der Entwicklungswerkzeuge. Ausgehend davon sollte die weithin verfügbare Literatur und Dokumentation gut verständlich und somit auch herangezogen werden. Der Umgang mit Software für 3D-Modelling und -Animation (z.B. Cinema 4D), Bildbearbeitung (Photoshop), Audiotracking (z.B. Audacity) ist nicht Gegenstand der Veranstaltung, sollte aber darüber hinaus geübt werden.

## 2 IDE

Die IDEs von Flash und Unity ermöglichen es, grafische Elemente der Applikation direkt in der Entwicklungsumgebung zu erstellen, zu animieren und in einer Szene zu platzieren, sowie diese in einen komplexen logischen und geometrischen Kontext zu stellen, ohne dass hierzu weitere Werkzeuge oder Programmierung erforderlich wäre. Der durchdachte und geübte Umgang damit, insbesondere der Aufbau sinnvoller Hierarchien und Strukturen, ist entscheidend für die nachfolgenden Prozesse. Die wichtigsten Metaphern beider Programme werden hier kurz vorgestellt:

### 2.1 Zeitleiste (*Timeline*)

Zentrales Element bei Flash ist die Zeitleiste, da Flash ursprünglich ein Animationswerkzeug ist. Wie bei einem Zeichentrickfilm die Folien, sind hier die Ebenen (Layer), auf denen unabhängig grafische Objekte verteilt werden können, vertikal übereinander angeordnet. Horizontal ist der Ablauf der so entstehenden Bilder (Frames) festgehalten. Dabei sind die Bilder nummeriert von 1 bis zum letzten der Timeline. Der Bezug zwischen Zeit und Bild ergibt sich erst beim Abspielen auf Grund der eingestellten bzw. der erreichbaren Bildwiederholrate (Framerate). Da die Frames aber auch benannt und wahlfrei angesteuert werden können ist es in Flash zudem üblich, verschiedene Zustände oder Interfaces der Anwendung in der Zeitleiste zu visualisieren und zu strukturieren.

Bei Unity spielt die Zeitleiste eine geringe Rolle und ist nur in Bezug auf einzelne Animationen zu finden. Dabei ist es möglich, verschiedene Eigenschaften eines Objektes mit der Zeit auf Basis von grafisch als Kurven dargestellten Funktionen zu verändern. Diese Eigenschaften sind vertikal aufgetragen. Horizontal aufgetragen ist die Zeit in Sekundenbruchteilen, nicht als Frames. So ist die Animation unabhängig von der Framerate.

### 2.2 Szenenbaum, *Displaylist*

Zentrales Element bei Unity ist der Szenenbaum (Scenetree, Hierarchy). Hier werden die hierarchischen Beziehungen zwischen den Objekten grafisch dargestellt und interaktiv gestaltet. Die Funktionalität ist dabei ähnlich der von Ordnerstrukturen in der grafischen Benutzeroberfläche des Betriebssystems. Die geometrischen Transformationen eines übergeordneten Objektes (parent) wirken sich dabei ebenso auf die untergeordneten Objekte (children) aus.

Eine solche Verschachtelung ist auch in Flash üblich und sinnvoll. Symbole (in der Regel MovieClips) enthalten selbst wieder weitere Symbole und diese möglicherweise wieder andere. Da sich aber auf Grund der Zeitleistenmetapher der komplette Szenenbaum sich von Bild zu Bild ändern kann, ergibt die Darstellung des Baumes als weiteres zentrales Element keinen Sinn. Allerdings wird die übergeordnete Hierarchie als Breadcrumb angezeigt, wenn ein untergeordnetes Element bearbeitet wird. Der Szenenbaum der auf der Bühne befindlichen Elemente wird bei Flash „*Displaylist*“ genannt.

### 2.3 Szene

Bei Unity bietet es sich an, zum Beispiel verschiedene Levels eines Spiels als Szenen abzuspeichern. Diese können relativ leicht gewechselt oder hinzugeladen werden. Dabei wird allerdings nicht der aktuelle Gesamtzustand der Szene gespeichert und festgehalten, sondern nur die Szenenbäume

mit ihren Einstellungen und den Verweisen in die Projektbibliothek gespeichert. Nachträgliche Änderungen in der Bibliothek wirken sich beim erneuten Laden der Szene aus. Damit müssen Elemente, die in mehreren Szenen verwendet werden, nicht ständig in allen Szenen aktualisiert werden.

Auch Flash kennt Szenen, allerdings ist deren Nutzung nicht empfehlenswert, da ein Projekt dadurch recht unübersichtlich wird. Diese Szenen werden nicht in eigenen Dateien gespeichert sondern im Dokument „versteckt“. Es hat sich eher durchgesetzt, das Programm mit Hilfe der Timeline zu strukturieren oder in verschiedene Dokumente aufzuteilen. Ein Level kann z.B. als fertig kompilierte Datei zur Laufzeit hinzugeladen werden.

## **2.4 Projekt**

Bei der Arbeit mit Unity ist es unumgänglich, zunächst ein Projekt anzulegen oder zu laden. Letztendlich handelt es sich dabei weitestgehend um die Ordner- und Dateistruktur auf dem Datenträger, in der alle Elemente der Anwendung gespeichert werden. In der Regel ist es auch möglich, externe Daten einfach auf dem Datenträger in ein Projekt zu kopieren, Unity aktualisiert selbstständig das Projekt und importiert die Daten. Ein Projekt kann eine oder mehrere Szenen enthalten.

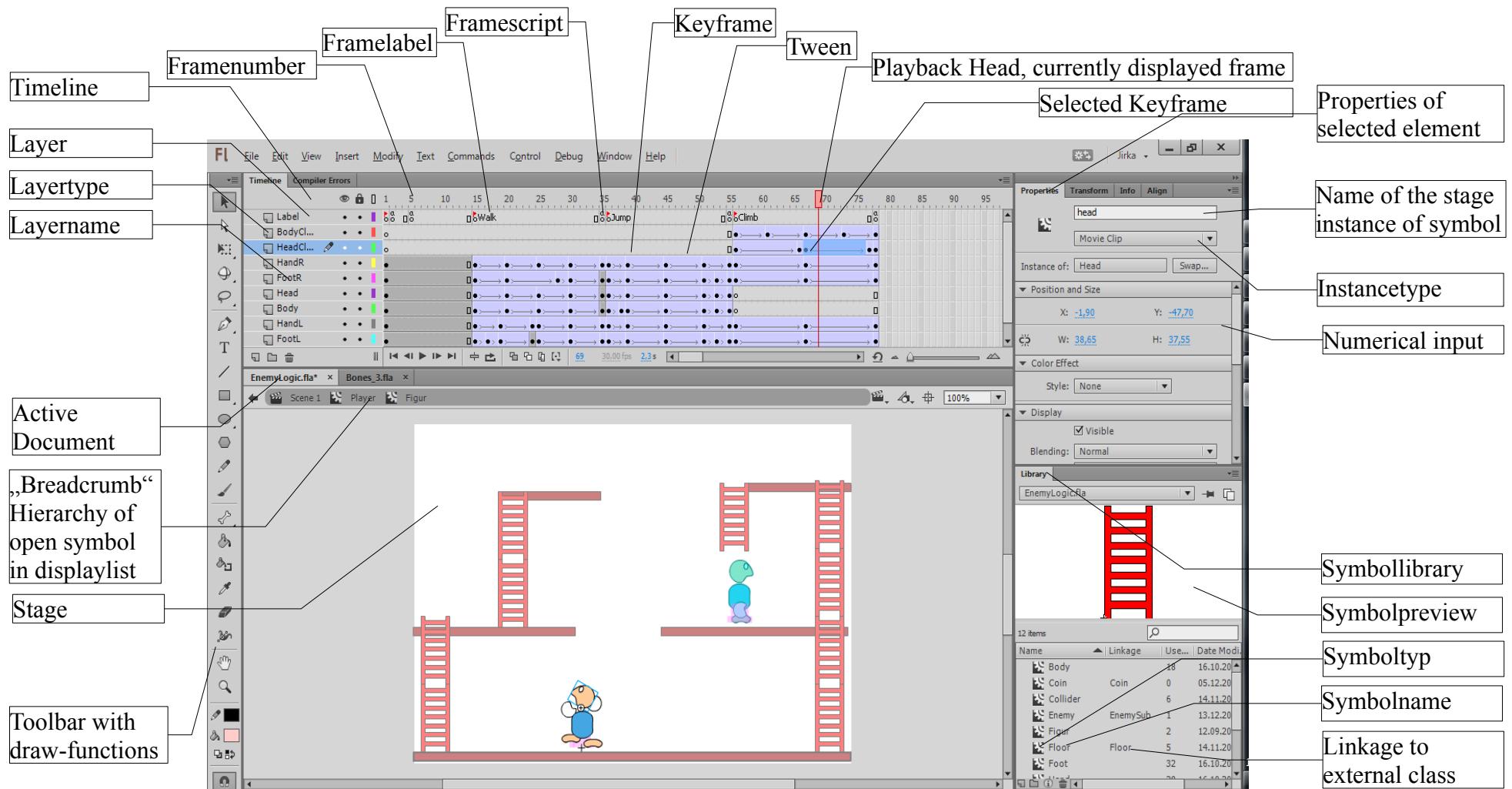
Bei Flash werden die Daten standardmäßig in einem eigenen Dateiformat, dem FLA-File, verwaltet. Importierte Assets werden darin in einer Bibliothek abgelegt, die über die IDE zugänglich ist. Darüber hinaus können auch weitere Dateien referenziert und zur Kompilier- oder Laufzeit geladen werden. Seit Adobe-CC gibt es hierfür keine integrierte Projektverwaltung mehr, es war auch nie erforderlich sie zu nutzen. Stattdessen bietet es sich an, die zusammengehörigen Dateien einfach in einem Ordner ggf. mit entsprechenden Unterordnern im Dateisystem zu verwalten.

## **2.5 Grafikfunktionen**

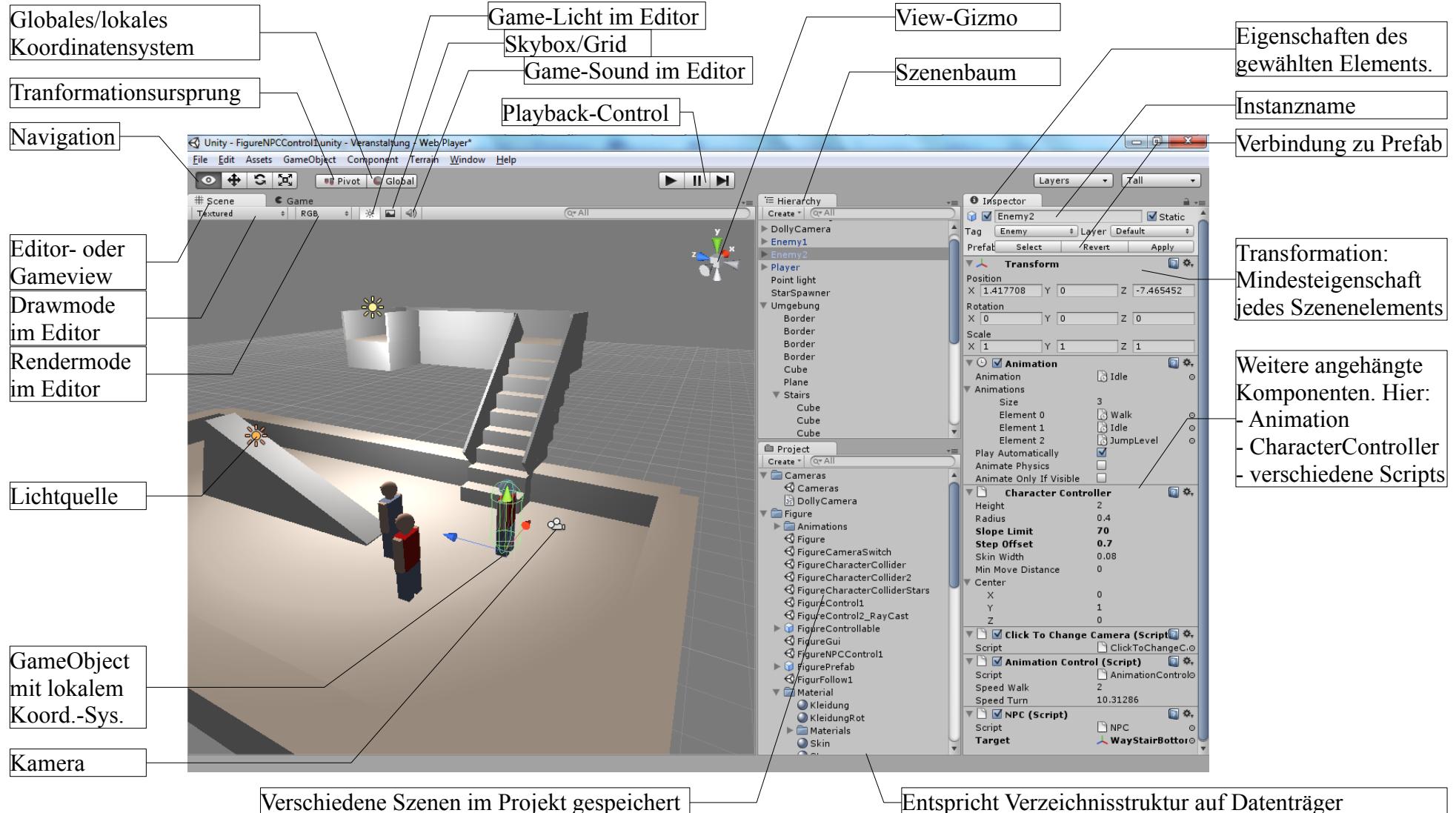
Flash verfügt über ein großes Spektrum an Zeichenfunktionen, von primitiven Grundformen über Freihandlinien bis zu Füllgradienten. Zudem kann es Bitmaps vektorisieren, Vektorgrafiken optimieren und kombinieren und vieles mehr. In gewissen Rahmen kann Flash durchaus als Grafikprogramm genutzt werden.

Unity dagegen ist ausdrücklich kein Modelling-Programm. In der IDE können lediglich die primitiven Grundkörper Cube, Sphere, Capsule, Cylinder, Plane und Quad erzeugt werden. Diese können mit Material-Shadern versehen werden, Texturen müssen an anderer Stelle erzeugt werden. Die Kombination von primitiven Körpern ermöglicht es aber bereits komplexere Gebilde zu schaffen und zu animieren, was für ein „rapid prototyping“ häufig hinreichend ist. Vor diesem Hintergrund verwundern allerdings die Möglichkeiten, welche die IDE zum Erschaffen von detaillierten Landschaften mit Vegetation bietet.

## 2.6 Überblick Flash (CC2015-IDE)



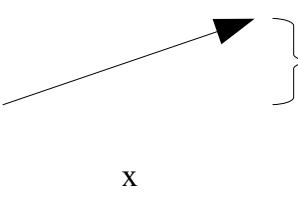
## 2.7 Überblick Unity5-IDE



## 3 Mathematische Grundlagen

Teilgebiete der linearen Algebra spielen bei der Spieleentwicklung eine tragende Rolle. Ein wenig Kenntnis derselben ist häufig entscheidend dafür, ob ein Spiel leicht oder nur mit großen Anstrengungen konzipiert und entwickelt werden kann. Diese Kenntnisse sollen hier wieder ein wenig aufgefrischt werden, und auf die Implementation und Anwendung in Flash und Unity eingegangen werden. Die Darstellung ist bei Weitem nicht umfassend und nur auf die Bereiche beschränkt, welche für die Grundlagen der Spieleentwicklung unverzichtbar erscheinen. Bei der Formulierung steht zudem der intuitive Zugang im Vordergrund, nicht die wissenschaftliche Unanfechtbarkeit.

### 3.1 Vektor

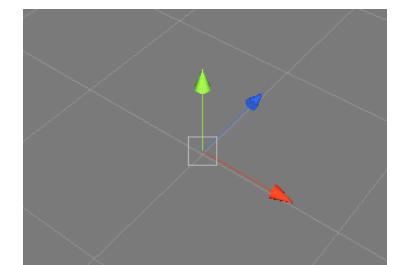


Mit Vektoren ist es möglich, Richtungen und Längen in der Fläche oder im Raum einfach zu beschreiben, zum Beispiel den Betrag und die Richtung einer Kraft oder auch Helligkeit und Einfallsinkel von Licht. In der Fläche wird der Vektor in der Regel mit zwei Komponenten  $x$  und  $y$ , für die zwei unabhängigen Dimensionen Breite und Höhe angegeben. Im Raum kommt noch die Komponente  $z$  hinzu, welche die Tiefe angibt.

Während in den Unity-Klassen konsistent Vektoren verwendet werden, ist dies in Flash leider nur sporadisch der Fall. Meist müssen die Komponenten getrennt voneinander behandelt werden. Häufig ist es aber dennoch sinnvoll, manchmal auch erforderlich, die Point-Klasse zu verwenden, welche einen zweidimensionalen Vektor darstellt.

### 3.2 Koordinatensystem

Die Komponenten der Vektoren beziehen sich dabei auf Koordinatensysteme, welche den Raum bemessen. Die Grundlage bildet in Flash das „globale Koordinatensystem“, welches in der linken oberen Ecke der Stage seinen Ursprung hat, und die Fläche nach rechts und unten im Pixelraster teilt. Bei Unity wird diese Grundlage auch „World“ genannt. Es ist ein „linkshändiges“ Koordinatensystem, da sich die Koordinatenachsen nur an der linken Hand mit Daumen ( $x$ , rechts, rot), Zeigefinger ( $y$ , oben, grün) und Mittelfinger ( $z$ , hinten, blau) darstellen lassen. Die Einheit sind Meter.



Bezogen auf den Ursprung eines Koordinatensystems kann ein Vektor nun einen beliebigen Punkt innerhalb desselben definieren, damit also auch Vertices einer Vektorgrafik oder eines Meshes, oder die Position eines komplexen Objektes. Ein solcher Vektor heißt dann auch Ortsvektor und ist gleichbedeutend mit Punkt-Koordinaten.

### 3.3 Lokale Koordinatensysteme

Jeder MovieClip in Flash und jedes GameObject in Unity bringt aber auch sein eigenes, lokales Koordinatensystem mit. Alle Bestandteile des Objektes sind in Bezug auf dieses lokale Koordinatensystem definiert. Damit kann das Objekt an jeder beliebigen Position innerhalb des globalen

Koordinatensystems rekonstruiert werden. Es muss dazu nur das lokale Koordinatensystem an diese Position verschoben werden.

## **3.4 Transformation**

### **3.4.1 Translation**

Eine solche Verschiebung wird Translation genannt, und lässt sich in Richtung und Betrag intuitiv mit einem Vektor beschreiben. In Unity bewirkt die Translation um den Vektor (2,-1,4) eine Verschiebung um 2m nach rechts, 1m nach unten und 4m nach hinten.

### **3.4.2 Skalierung**

Auch die Skalierung lässt sich mit einem Vektor beschreiben. Die Dimensionen des Vektors geben dann den Betrag an, um die entsprechende Dimension des lokalen Koordinatensystems zu strecken oder zu stauchen. Alles innerhalb dieses Koordinatensystems wird entsprechend gestaucht und gestreckt dargestellt. Aber auch nachfolgende Transformationen berücksichtigen diese Verzerrung. Ein Objekt, welches sich von oben nach unten in einem vertikal stark gestauchten Koordinatensystem bewegt, sieht nicht nur sehr flach aus, sondern bewegt sich auch deutlich verlangsamt.

### **3.4.3 Rotation**

In der Ebene ist nur die Rotation im oder gegen den Uhrzeigersinn relevant. Diese lässt sich also mit einer einzigen Winkelangabe vollständig beschreiben. Im Raum dagegen können wieder drei Drehwinkel angegeben werden, je einer für die Rotation um jede der Koordinatenachsen, die sogenannten Euler-Winkel. Hierbei muss allerdings die Reihenfolge der Rotationen beachtet werden, denn es macht einen Unterschied, ob zuerst um die X-Achse und dann um die Y-Achse gedreht wird, oder umgekehrt. Unity scheint die Rotationen in dieser Folge zu berücksichtigen: Y, X, Z.

Eine optimale Interpolation zwischen verschiedenen Rotationen wird mit Hilfe von Quaternionen erreicht, welche an dieser Stelle nicht behandelt werden sollen.

### **3.4.4 Matrix**

Alle diese Transformationen lassen sich einheitlich mit Matrizen beschreiben. In der Regel ist es nicht erforderlich, diese Matrizen direkt zu manipulieren. Meist genügt die Manipulation über einfache Befehle wie `translate` oder `rotate`. Folgendes ist aber wichtig zu wissen:

- Die Begriffe Matrix und Koordinatensystem entsprechen sich in diesem Zusammenhang weitgehend. Weder Unity noch Flash verfügen über einen Datentyp wie „CoordinateSystem“, sehr wohl aber über Datentypen für Matrizen. Es wird davon ausgegangen, dass allen grafischen Objekten kartesische Koordinatensysteme zu Grunde liegen. Somit genügt es, deren Verschiebungen, Rotationen und Verzerrungen gegeneinander mit Matrizen zu speichern um eine Szene beschreiben und animieren zu können.

- Unabhängig davon, wie viele Koordinatensystem ineinander verschachtelt sind, es lässt sich die Gesamttransformation auch des letzten Objektes in einer solchen Hierarchie sehr leicht mit einer einzigen Matrix beschreiben. Hierbei werden lediglich die einzelnen Transformationsmatrizen der bis dahin verschachtelten Koordinatensysteme in einem simplen Verfahren, der Matrixmultiplikation, zusammengerechnet.
- Für jede Matrix lässt sich auch eine inverse Matrix bestimmen. Damit kann die Position eines Punktes in einem beliebigen lokalen Koordinatensystem leicht in seine Position bezogen auf das globale Koordinatensystem umgerechnet werden und vice versa.
- Die Transform-Komponente in Unity stellt die Matrix (bzw. das lokale Koordinatensystem) des GameObjects dar. Sämtliche Manipulationen werden direkt an dieser Matrix ausgeführt.
- In Flash ist es (leider) nicht möglich, die Matrix eines DisplayObjects direkt zu manipulieren. Hier muss man zunächst die aktuelle Matrix einem Matrixobjekt zuweisen, dieses manipulieren und dann zurückschreiben.

### 3.4.5 Quaternionen

Quaternionen sind mathematische Konstrukte, welche die Behandlung von Rotationen im dreidimensionalen Raum vereinfachen und weiche Interpolationen unterstützen. Die Unity-Methoden `Transform.rotation` und `Transform.localRotation` liefern die Orientierung der Transformation als Quaternion. Die vier Werte des Quaternions sollten nicht direkt manipuliert werden. Stattdessen wird ein Quaternion mit Methoden wie `Quaternion.Lerp`, `.Slerp`, `.RotateTowards` und `.LookRotation` neu kreiert und der Transformation zugewiesen. Dies entspricht der Vorgehensweise mit Matrizen in Flash.

## 3.5 Konsequenz

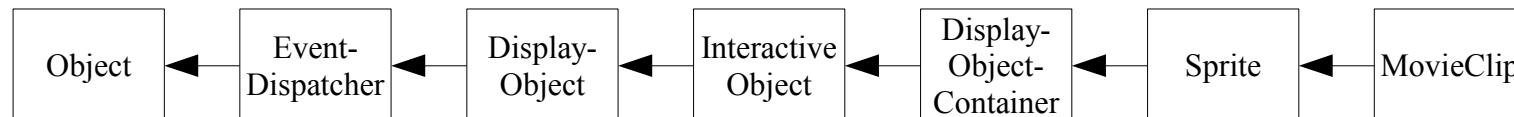
Transformationen sind immer auf Koordinatensysteme bezogen. Bei der Entwicklung von Spielen mit nicht trivialer Grafik ist von Anfang an, sowohl bei der Erstellung von Assets, wie auch bei der Programmierung, streng darauf zu achten, dass die Bezugssysteme korrekt gewählt sind. Neben der Unterscheidung zwischen lokalem und globalem Koordinatensystem, sowie Bezugssystemen in der geometrischen Hierarchie des Szenenbaums und willkürlich gewählten, ist zudem auf die zusätzlichen Transformationssysteme bei der Erstellung oder Animation von visuellen Elementen in Flash bzw. Unity zu achten. Neben dem Pivotpunkt (Unity) und dem Registrationpoint (Flash), welche den Ursprung des lokalen Systems des betreffenden Elementes darstellen, gibt es noch den Centerpoint(Unity), der dem geometrischen Mittelpunkt des Elementes entspricht, und den Transformationpoint (Flash), welcher frei gewählt werden kann. Beide sind für die manuelle Bearbeitung wichtig und hilfreich, spielen aber bei der Programmierung keine Rolle. Hierauf ist dringend zu achten.

## 4 Logischer Aufbau

Flash und Unity verfolgen beim Aufbau der Elemente im Szenenbaum ganz unterschiedliche Ansätze. Gemeinsam ist, dass zwischen übergeordneten und untergeordneten Objekten eine Parent-Child-Beziehung hergestellt wird. Ein Parent-Objekt kann dabei auf mehrere Children verweisen, ein Child-Objekt verweist aber immer nur auf genau einen Parent. Beide Programme verfügen außerdem über Mechanismen um Nachrichten durch den Szenenbaum zu schicken, so dass auch Informationen zwischen Objekten ausgetauscht werden können, ohne die direkte Beziehung nutzen zu müssen.

### 4.1 Klassenhierarchie in Flash

Flash nutzt eine tiefe Klassenhierarchie, welche sich bei der Arbeit in der IDE implizit erweitert. Dies wird an dem unteren Schaubild für die Klasse MovieClip deutlich. Von der Basisklasse Object leitet sich der EventDispatcher ab, der schon Nachrichten senden und empfangen kann. Dieser wird erweitert zur Klasse DisplayObject welche bereits einen Rahmen für eine Visualisierung bereithält und z.B. die Transformation zur Darstellung des Inhaltes auf der Bühne zur Eigenschaft hat. InteractiveObject ist wiederum eine Erweiterung dieser Klasse und enthält einfache Standardverhalten für Maus und Tastatureingaben. Davon wiederum abgeleitet ist der DisplayObjectContainer, welcher in der Lage ist, unter sich eine Liste von DisplayObjects zu verwalten, und damit die Grundlage für den Aufbau der Displaylist (Szenenbaum) darstellt. Sprite schließlich erweitert nun endlich diese Klasse um eine tatsächliche grafische Darstellung, speichert also Vektordaten für Punkte, Linien und Füllungen etc. und hält Drag&Drop-Funktionen bereit. Am Ende dieser Kette steht schließlich der MovieClip der sich vor allem dadurch auszeichnet, dass er selbst eine eigene Zeitleiste und sämtliche Steuerfunktionen besitzt, also im Wesentlichen eine eigene Flash-Bühne oder sogar Applikation darstellt.



Legt man allerdings, wie man es häufig und standardmäßig tut, einen MovieClip in der Library an, wird implizit eine neue Klasse erschaffen, abgeleitet von MovieClip, welche über die besonderen Fähigkeiten des Abspielens und Darstellens der angelegten Animation, die in den Framescripts niedergelegten Funktionen usw. verfügt. Sprich: selbst wenn man noch keinen Code schreibt, arbeitet man in Flash stark objektorientiert.

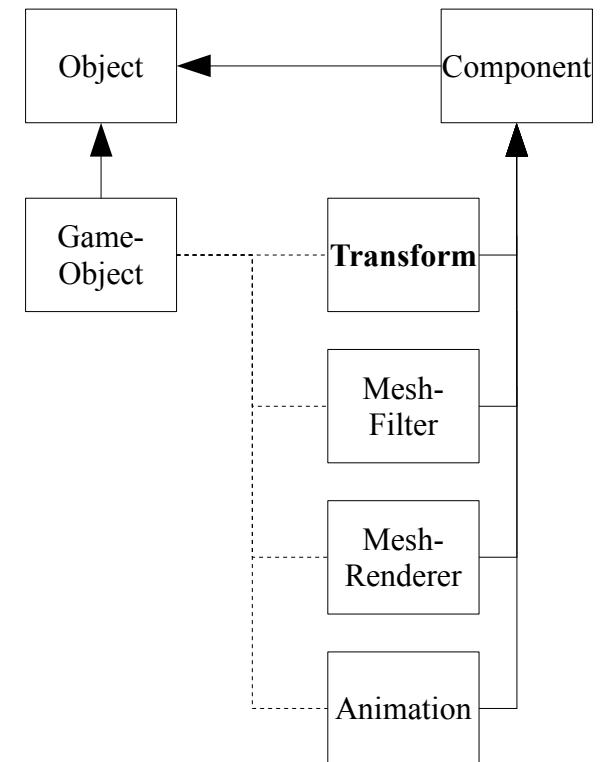
## 4.2 Komponentenmodell in Unity

Unity arbeitet dagegen mit einer sehr flachen Klassenhierarchie. Dagegen wird mit Kompositionen und Aggregationen gearbeitet, wobei Objekte verschiedener Klassen zu größeren Einheiten verknüpft werden. Einer Instanz der Klasse GameObject können verschiedenste Komponenten angefügt und schließlich parametriert werden. GameObject und Component sind beide Subklassen der Basisklasse Object. Ein GameObject kann bereits in der IDE per Maus in den Szenenbaum eingesetzt werden. Die Komponente **Transform** ist dabei standardmäßig bereits an das GameObject gekoppelt und kann nicht entfernt werden. Transform übernimmt dabei zwei wichtige Aufgaben:

- Bereithalten der Transformation für das GameObject (analog DisplayObject in Flash)
- Aufbau der Parent-Child-Beziehung zwischen GameObjects (analog zu DisplayObjectContainer). Da man die Transformation auch als (ggf. verschobenes und verzerrtes) Koordinatensystem betrachten kann, hat man es hier also recht offensichtlich mit einer Verschachtelung von Koordinatensystemen zu tun. Die darzustellenden Objekte werden entsprechend der Hierarchieebene innerhalb dieser Verschachtelung dargestellt.

Im Beispiel rechts sind dem GameObject zudem noch folgende Komponenten angefügt:

- MeshFilter, liefert die Geometriedaten für das GameObject.
- MeshRenderer, appliziert Materialien und Shader etc. und zeichnet das Objekt.
- Animation, bewegt das Objekt zeitgesteuert, sofern Animationsdaten zugewiesen sind.



Zu beachten ist, dass das GameObject und alle Komponenten der Aggregation automatisch eine Referenz auf die Transform-Komponente pflegen. Sie ist also von einer Skript-Komponente mit `this.transform` leicht zu erreichen. Um auf alle anderen Komponenten des GameObjects zuzugreifen ist es dagegen erforderlich, die Referenz zunächst mit der Methode `GetComponent(...)` oder `GetComponents(...)` explizit zu ermitteln.

## 5 Erstellung visueller Elemente

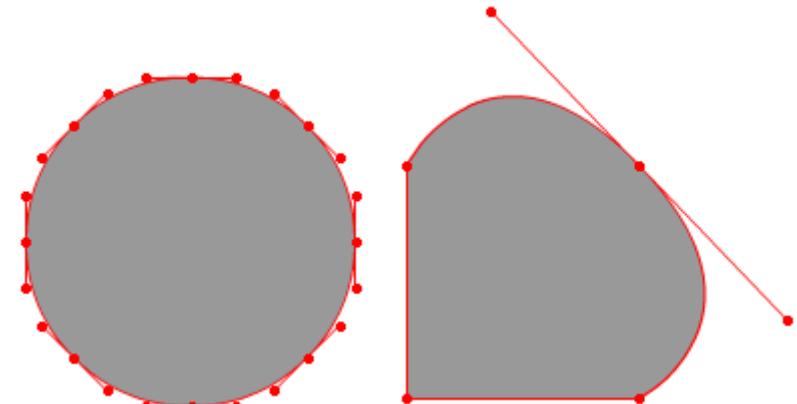
### 5.1 Vektorgrafik mit Flash

Sofern keine Bitmaps importiert werden, arbeitet Flash bei der grafischen Darstellung durchgängig mit Splines. Körper werden mit Hilfe von Punkten (Vertices) beschrieben, zwischen denen Kurven mit unterschiedlichen Gradienten an den Enden gezogen, und zwischen denen wiederum Flächen gefüllt werden. Das Beispiel zeigt einen Kreis und daneben ein Quadrat, dessen rechter oberer Eckpunkt manuell modifizierte Gradienten erhalten hat. Die Außenlinie wurde zur besseren Erkennbarkeit gelöscht.

Überlappen sich mehrere Körper auf einer Ebene (Layer), entstehen an den Schnittpunkten der Kanten neue Vertices. Zudem wird die Körperkombination als ein einzelner Körper zusammengefasst, der aber an den Überlappungskanten wieder geteilt werden kann. So können leicht Boolesche Operationen durchgeführt werden.



Um solche Verschmelzungen oder Schnitte zu vermeiden ist es eine gute Angewohnheit mit Layern nicht zu sparen und die Objekte so zu separieren. Außerdem ist es auf diese Weise leicht möglich, die Anordnung der Objekte in der Tiefe (Z-Achse) festzulegen und bei Bedarf zu ändern. Die Tiefe und Wechsel in der Objektanordnung darin ist bei 2D-Engines ein grundsätzliches Problem, welches bereits zu Beginn eines Projektes bedacht werden sollte. Bei 3D-Engines löst sich dies ad hoc auf, weshalb heute meist auch bei Iso-Spielen mit solchen Engines gearbeitet wird.



### 5.2 Primitive Meshes in Unity

So beschränkt die Möglichkeiten der Objektgenerierung in Unity sind, so einfach sind sie zu handhaben. Über Hierarchy→Create oder GameObject→CreateOther kann man die Primitiven von Cube bis Plane auswählen, die dann im aktuell sichtbaren Bereich der Szene erscheinen. Position, Rotation und Skalierung können dann per Maus und Navigationsbuttons, oder Numerisch im Inspektorview der Transform-Komponente eingestellt werden. Schon hierbei sollten einige Grundregeln beachtet werden:

- Objekte sollten gezielt an eine definierte Position gebracht werden, da sie bei der Generierung auch weit über dem Boden schweben oder darin versunken sein können.
- Da später möglicherweise kinematische Simulationen zum Einsatz kommen, sollten die Objekte realistische Ausmaße haben und in der Einheit Meter definiert sein.

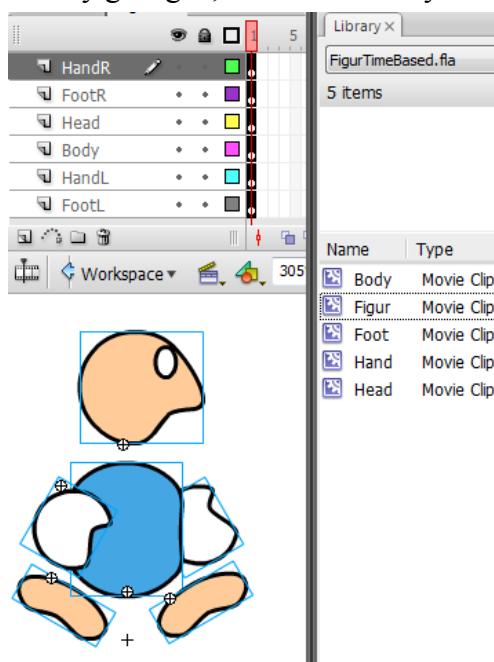
Farbe und Glanz erhalten die Objekte über Materialien, die als eigenständige Elemente zunächst über Project→Create→Material erzeugt werden.

# 6 Symbole und Basis-Koordinatensysteme

Nicht nur in Spielen ist es häufig sehr sinnvoll, selbst erschaffene komplexere Strukturen und grafische Bestandteile wiederverwendbar zu machen. Jedes Spielement, das mehr als einmal im Spiel vorkommt, sollte als eigenes Symbol in einer Bibliothek abgespeichert sein. So können z.B. leicht Hunderte gegnerische Einheiten mit gleichem Aussehen und Verhalten, Projekteile, Fallen oder einfach Zierelemente in der Szene als Instanzen solcher Symbole eingefügt werden. In Unity und Flash sogar einfach per Drag&Drop aus der entsprechenden Bibliothek. Eine Änderung des Symbols bewirkt die Änderung aller Instanzen des selben. Bei der Erstellung solcher Symbole spielt die geschickte Wahl des richtigen Koordinatensystems und Ursprungs eine zentrale Rolle. Bei einer humanoiden Figur ist beispielsweise besonders sinnvoll, den Ursprung auf Höhe der Fußsohlen und horizontal mittig zu setzen. Damit entspricht später die Position einer Instanz der Figur in der Spielwelt auch tatsächlich ihrem „Standpunkt“!

## 6.1 Flash-Symbole

Bei Flash ist eine Library immer fest in eine Flash-Datei (\*.fla) integriert. Ohne Daten zu importieren gibt es in Flash drei Möglichkeiten, Symbole in der Library anzulegen. Erstellte Grafiken können ausgewählt und entweder per Drag&Drop zur Library gezogen, oder mit Modify→Convert to Symbol (F8) konvertiert werden.

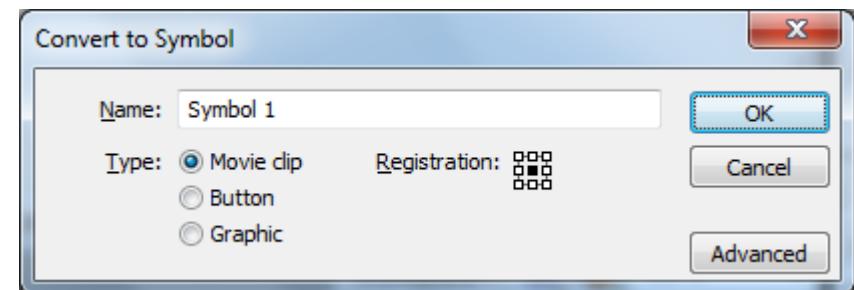


Mit Insert→New Symbol (Ctrl+F8) kann zudem ein neues, leeres Symbol geschaffen werden. In allen drei Fällen wird darum gebeten, Name und Typ des Symbols anzugeben. Der Name sollte möglichst aussagekräftig sein (nicht „Symbol 1“) und als Typ soll an dieser Stelle nur MovieClip genutzt werden, da dieser Typ am mächtigsten ist. Wichtig ist außerdem die richtige Wahl des Registrierungspunktes. Hiermit wird der Koordinatenursprung, also das lokale Koordinatensystem des Symbols festgelegt. Alle Informationen lassen sich später auch noch ändern.

Das Beispiel links zeigt den MovieClip „Figur“, der wiederum aus sechs Instanzen anderer Symbole besteht. Dabei wurden die Symbole für Hand und Fuß zweifach genutzt. Zu beachten ist, dass die Anordnung in der Tiefe durch die Ebenen der Zeitleiste des MovieClips erfolgt, in jeder liegt nur eine Instanz.

Das kleine Kreuz unten ist der Ursprung der Figur, die Kreuze in den Kreisen markieren die Positionen der ausgewählten Instanzen der Körperteile.

Bei der Konvertierung mehrerer und ggf. animierter Objekte in einen MovieClip wird die Zeitleiste nicht mitgenommen. In diesem Fall ist es günstiger, ein neues Symbol zu kreieren und die Zeitleiste mit Ctrl+Alt+C und Ctrl+Alt+V dort hinein zu kopieren.



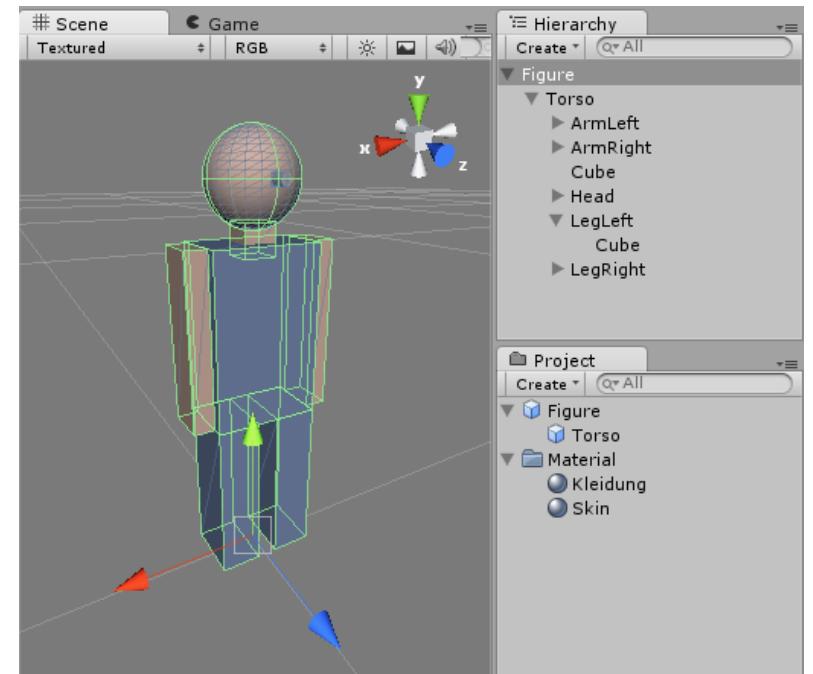
## 6.2 Unity-Prefabs

Die Funktion der Bibliothek übernimmt bei Unity die Projektstruktur. Neben Materialien, Skripten, Szenen und weiterem, können hier auch die eingangs erwähnten Symbole als so genannte Prefabs (engl. Prefab = vorgefertigt) gespeichert werden.

Ein Prefab wird am einfachsten erzeugt, in dem ein Ast aus der Hierarchie in den Projekt-View gezogen wird. Dort wird die Struktur in einem eigenen Datensatz abgelegt und die ursprüngliche Quelle in der Hierarchie farbig als Instanz derselben markiert. Es bietet sich an, zunächst ein leeres GameObject, welches nur eine Transform-Komponente hat, mit `GameObject→CreateEmpty` zu erstellen, und dieses als Parent und damit als Basis-Koordinatensystem für das komplexe Gebilde zu nutzen, welches ein Prefab werden soll. Dieses Koordinatensystem sollte die Einheitsmatrix darstellen, um nicht ungewollte Transformationen in den Symbolen später berücksichtigt werden müssen.

Zu beachten ist auch, dass im Project-View nur die ersten beiden Hierarchieebenen des Prefabs dargestellt wird. So können über die Selektion im Project-View auch nur die Eigenschaften des Wurzelements und dessen Kinder manipuliert werden. Für die Bearbeitung einer tieferen Hierarchie muss also an einer Instanz des Prefabs in der Szene gearbeitet werden. Im Inspector erscheinen bei den Prefab-Instanzen oben die Buttons „Select“, „Revert“ und „Apply“, mit deren Hilfe Änderungen der Instanzeigenschaften auf das Prefab übertragen oder verworfen werden, oder einfach das referenzierte Prefab im Project-View markiert wird.

Im Beispiel ist das GameObject „Figure“ lediglich eine Transformation mit einem Kind „Torso“. Dieses hat weitere Kinder: Arme, Beine, Kopf und einen Cube, welcher den Torso selbst darstellt. Wie im Flash-Beispiel wurde die Figur auf den Ursprung des Koordinatensystems gesetzt, so dass ihre Position auch ihr Standpunkt ist. In drei Dimensionen muss aber auch die Ausrichtung beachtet werden! Die positive Y-Achse zeigt nach oben, die X-Achse nach rechts und die Z-Achse nach vorne. Auch wenn es mathematisch nicht relevant ist, diese Konvention macht die nachfolgende Entwicklung, insbesondere die Programmierung auf Grund verschiedener Standardkonstanten und -methoden deutlich leichter!



## 6.3 Ordner

Um die Bibliothek bzw. das Projekt zu strukturieren sollten Ordner genutzt werden, um Zusammengehöriges auch zusammen zu fassen. Im Unity-Beispiel ist der Ordner „Material“ zu sehen. Hier wird er mit `Project→Create→Folder` erzeugt, bei Flash mit „New Folder“ im Kontextmenü der Library. Diese Symbole und Projektbestandteile können dann per Drag&Drop in die -sinnvoll benannten- Ordner verteilt werden.

## 7 Animation

Kaum ein Spiel kommt ohne Animation aus. Die Beachtung einiger wichtiger Grundregeln kann es deutlich vereinfachen, Animationen auf Basis der mathematischen Prozeduren von Flash und Unity zu erstellen:

- Lokale Koordinatensysteme: die lokalen Koordinatensysteme für die einzelnen zu animierenden Bestandteile müssen richtig gewählt werden. Ein Rad, dessen Ursprung nicht im Zentrum, sondern eher am Rand liegt, lässt sich schwer rollen. Ebenso sollte der Ursprung eines Arms die Schulter sein.
- Koordinatensystem der Komposition: Es ist streng darauf zu achten, dass ein komplexes animiertes Gebilde einen sinnvollen Ursprung hat, mit dem in der Folge bei der Programmierung voraussehbar gearbeitet werden kann. Eine humanoide Figur sollte beispielsweise den Ursprung am Boden zwischen den Füßen haben, und nicht „an der Nase“ herumgeführt werden müssen. Alle unnötigen Rotationen und Skalierungen sind ebenso zu vermeiden.
- Zeitleiste vs. Skript: es muss entschieden werden, welche Bewegungen über die Zeitleiste und welche von der Skriptprogrammierung gesteuert werden. Beispielsweise findet die Laufanimation auf der Stelle statt, die Bewegung einer Figur durch die Szene steuert das Skript. Daraus folgt, dass man diese Überlegung beim Aufbau der Hierarchien in der Szene mit einbeziehen muss. Um Konflikte zu vermeiden sollte ein animiertes Objekt, welches per Skript durch die Szene bewegt werden soll, Kind eines Eltern-Objektes sein, wobei das Skript die Transformation des Eltern-Objektes manipuliert. Eine Komposition, die so aufgebaut am Nullpunkt der Szene ohne weitere Transformation eingesetzt wird, muss auch dort erscheinen, korrekt ausgerichtet sein und die richtige Größe haben!
- Sequenzen: die Animation muss in sinnvolle kleine Einheiten aufgeteilt werden, die aneinander gereiht und wieder verwendet werden können. Ein Sprung mit zwei Schritten Anlauf ist aus Sicht des Spielers schwer zu steuern, eine mehrere Minuten lange Idle-Animation sinnlos.
- Kombination: gegebenenfalls muss das zu animierende Objekt in einzeln zu animierende Gruppen zerlegt werden, um Animationen später flexibel zu kombinieren. Ein Reiter muss sich möglicherweise unabhängig von seinem Pferd bewegen können, die beiden sollten dann nicht in einer Animationssequenz stecken.

Die Animation wird festgelegt, in dem in der Zeitleiste Keyframes definiert werden. An diesen Keyframes werden Eigenschaften, meist die Transformationen, des zu animierenden Objektes manipuliert und abgespeichert. Beim Abspielen der Animation werden diese Eigenschaften von einem Keyframe zum nächsten über die dazwischenliegenden Frames interpoliert. Im einfachsten Fall handelt es sich dabei um eine lineare Interpolation, es ist aber auch möglich zu Beschleunigen, Abzubremsen oder über komplexe Kurvenverläufe zu interpolieren.

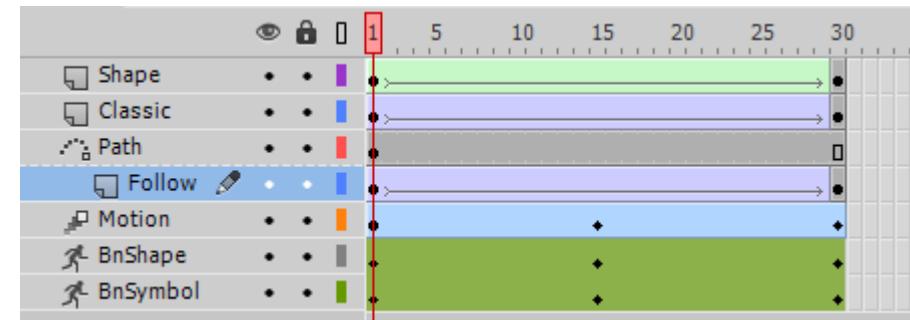
## 7.1 Timeline-Animation in Flash

Flash wartet gleich mit sechs verschiedenen Möglichkeiten auf, Animationen auf der Timeline zu erstellen. Die einfachen Shape- und Classic-Tweens sind an Pfeilen zwischen den Keyframes zu erkennen. Für Pfadanimationen, Motion-Tweens und Bone-Animationen werden spezielle Layer erzeugt, die eigene Icons mitbringen.

Keyframes können mit F6 erzeugt, mit Shift+F6 gelöscht, per Drag&Drop verschoben bzw. dabei mit gedrückter Alt-Taste kopiert werden. Alle weiteren Timeline-Funktionen lassen sich gut mit den kontextsensitiven Menüs steuern.

Auch um die Bestandteile eines Objektes unabhängig voneinander animieren zu können ist es erforderlich, sie auf Ebenen zu vereinzeln. Sind in einem

Keyframe mehrere Objekte definiert, ist die Interpolation von einem Keyframe zum nächsten nicht möglich, und Flash fasst stattdessen die Symbole zu einem neuen Tween-Symbol zusammen und legt sie in der Library ab. Es ist darauf zu achten, dass nicht versehentlich solche Tween-Symbole dort entstehen!



Verschiedene Tweens in der Timeline

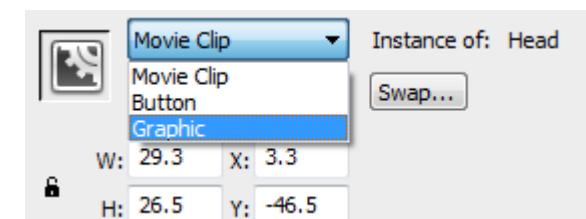
### 7.1.1 Shape Tween

kommt nur zum Einsatz, wenn man Vertices einer Vektorgrafik animieren möchte. So „morph“ eine Gestalt in eine andere. Liegen die Vertices in den unterschiedlichen Keyframes allerdings weit auseinander, oder stimmt die Anzahl nicht überein, was durch Linienkreuzungen leicht passieren kann, können sie für die Interpolation nicht korrekt zugeordnet werden. Mit Modify→Shape→Add Shape Hint können von Hand kritische Vertices im Startkeyframe und dann im Endkeyframe verknüpft werden. Nach Klick auf den Tween in der Timeline kann mit dem Ease-Parameter noch festgelegt werden, ob die Animation linear gleichmäßig ablaufen, beschleunigen oder bremsen soll.

### 7.1.2 Classic Tween

wird für die schnelle und einfache Animation von Symboleigenschaften verwendet. Ein Keyframe speichert die Eigenschaften eines Symbols auf der Bühne wie Transformation, Farbeffekt, Filter etc. Bei der Arbeit mit Classic-Tween sollte die Library im Auge behalten werden, damit keine Tween-Symbole entstehen.

Zu Beachten ist im Besonderen, dass die Animation in Flash immer nur für eine Hierarchieebene übersichtlich angelegt werden kann, da die Children eines MovieClips eine eigene Zeitleiste haben. In einem gewissen Maße lassen diese sich allerdings synchronisieren, indem man den Typ der Instanz auf



Anzeige eines MovieClips als Grafik zur Synchronisation

„Graphic“ wechselt. Dann kann man einen Frame angeben ab dem der Child-Clip spielen oder stoppen soll. Alles weitere muss dann per Skript gelöst werden.

Bei Drehungen ist nun die Eigenschaft „Rotate“ des Tweens zu beachten. Hier kann eingestellt werden, ob die Rotation in oder gegen den Uhrzeigersinn interpoliert werden soll. Außerdem kann auch eine höhere Anzahl an Rotationen eingestellt werden. Bei „Auto“ wird über den kleineren Winkel interpoliert.

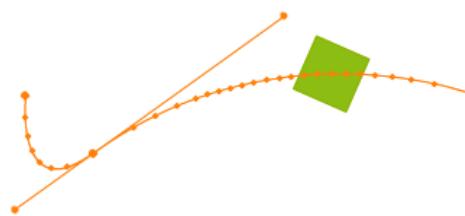
### 7.1.3 Pfadanimation

Soll sich ein Objekt entlang eines bereits existierenden Pfades bewegen, kann es unnötig mühsam sein, diesen mit Hilfe von Keyframes zu rekonstruieren. Stattdessen kann eine Linie auf einem Layer gezeichnet werden, dessen Typ auf „Guide“ eingestellt wird. Diesem kann nun ein weiterer Layer mit einem Classic-Tween per Drag&Drop untergeordnet werden. An den Keyframes muss nun das zu animierende Objekt mit seinem Transformationpunkt an der Linie eingerastet sein, dann folgt die Animation dem Pfad. Auch kann die Rotation dem Pfad folgen, wenn dies als Eigenschaft des Tweens eingestellt wurde. Sie wird ggf. mit den in den Keyframes vorgegebenen Drehungen und der Einstellung von „Rotate“ verrechnet, wenigstens die anfängliche Ausrichtung muss im ersten Keyframe von Hand justiert werden. Außerdem kann auch die Linienstärke und die Linienfarbe zur Manipulation der Animation herangezogen werden.

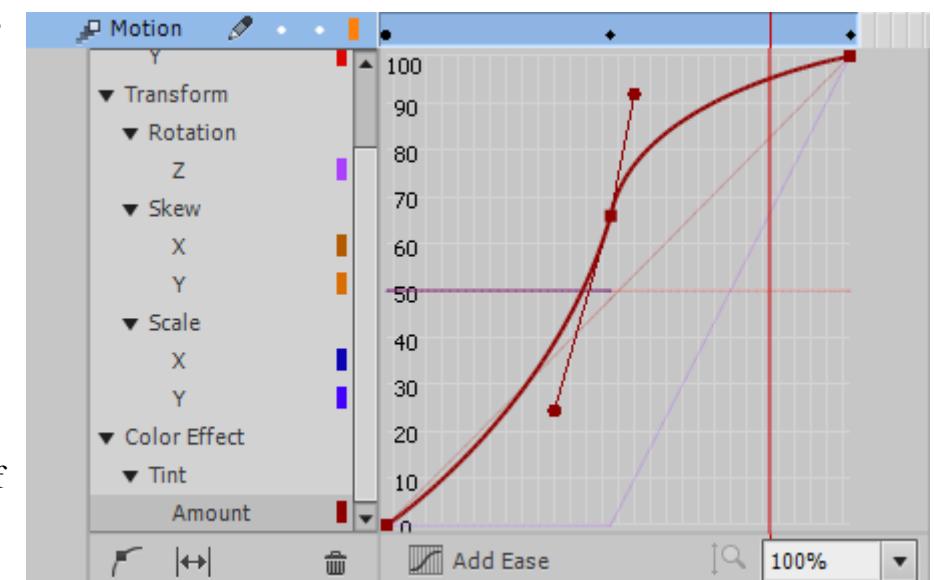
### 7.1.4 Motion Tween

erlaubt es, unabhängig voneinander verschiedene Eigenschaften eines Symbols zu animieren. Ein Keyframe hält also nur ausgewählte Eigenschaften fest und es werden bei einer Manipulation des Objektes auf der Stage nur die veränderten Eigenschaften automatisch aufgezeichnet. Diese Keyframes

erscheinen als kleine Punkte im Tween. Animationen der Position erscheinen zudem bei der Bearbeitung als Pfade auf der Stage und können dort direkt auch mit Hilfe von Splinepunkten manipuliert werden.

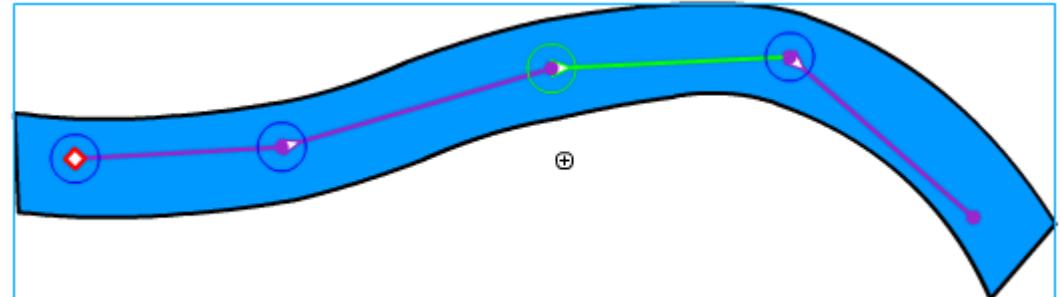


Hinzu kommt ein Motion-Editor, der Analogien zu dem Animation-Editor von Unity aufweist. Er wird über Doppelklick oder Rechtsklick→Refine Tween auf dem bereits angelegten Motion-Tween aufgerufen und erscheint in der Timeline. Hier sind die Veränderungen als Graphen aufgetragen, deren Haltepunkte und Tangenten direkt mit der Maus manipuliert werden können.



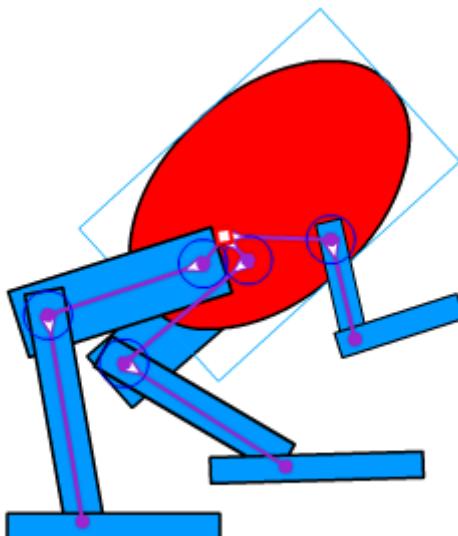
### 7.1.5 Boneanimation mit Shapes

Um natürlich wirkende Bewegungen von Abbildern von Gliedmaßen, Tentakeln oder anderen länglichen Formen herzustellen, kann man in diese eine Vektorkette wie eine Wirbelsäule einziehen. Diese Vektoren werden „Bones“ genannt und haben neben Länge und Richtung noch weitere Eigenschaften wie Transformationsrestriktionen. Ausgehend von einem Rootbone schließen sich die anderen Vektoren am Ende des jeweiligen Vorgängers an und stellen letztlich eine Vektoraddition dar. Bei einer Bewegung der Bones folgen die Vertices des Shapes den jeweils naheliegendsten Gelenken der Kette. Besonders komfortabel wird die Animationserstellung durch die inverse Kinematik, die es erlaubt einen beliebigen Bone mit der Maus zu bewegen, worauf die ganze Kette sich entsprechend ausrichtet. Sobald Bones genutzt werden, erstellt Flash hierfür eine spezielle „Armature“-Ebene, welche die Informationen verwaltet.



*Bones in Shape, gerade ist der dritte Vektor selektiert.*

### 7.1.6 Boneanimation mit Symbolen



*Skelett für ein laufendes Ei. Vom Wurzelement gehen drei Bones aus.*

In Verbindung mit Symbolen lassen sich mit Bones komplexe Skelette aufbauen. Dabei ist jeweils ein Symbol an ein Gelenk gebunden. Von einem Gelenk können mehrere Bones ausgehen, somit steht für die Arbeit an der Animation eine hierarchische Struktur auf derselben Timeline-Ebene mit inverser Kinematik zur Verfügung.

Alle so verbundenen Symbole werden auf der „Armature“-Ebene zusammen gefasst. Sie lassen sich aber per „Swap“ weiterhin austauschen. Sollen weitere hinzugefügt werden, müssen sie zunächst auf einer anderen Ebene auf der Bühne platziert und dann per Bone mit dem Skelett verbunden werden. Zu beachten ist, dass die Bones nicht an den Registrierungspunkten, sondern an den Transformationspunkten ansetzen und deren Position beim Erstellen des Skeletts gegebenenfalls neu definieren.

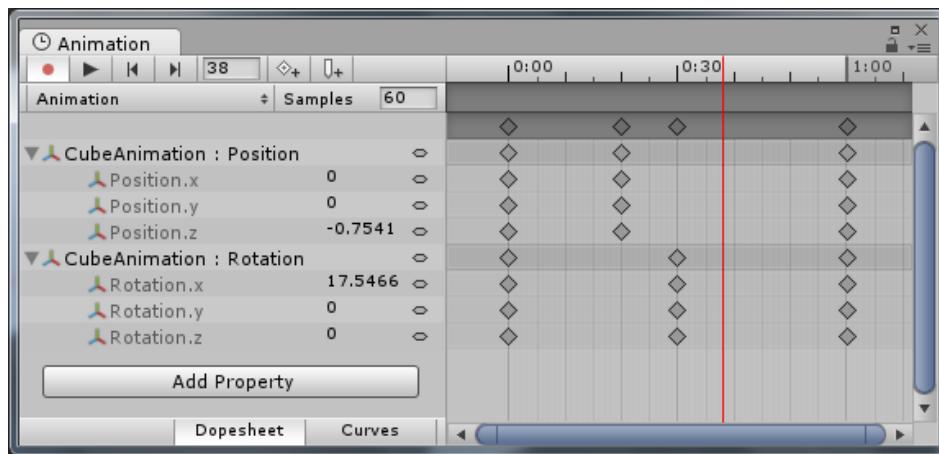
Sind die Symbole MovieClips, so laufen deren Animationen weiterhin ab bzw. lassen sich wie beim Classic-Tween als Grafik synchronisieren. Ebenso sind sie weiterhin per Skript ansprechbar. Allerdings ist zu beachten, dass die Skelethierarchie zur Laufzeit nicht existiert, also keine Parent-Child-Beziehung zwischen den Symbolen hergestellt ist.

## 7.2 Timeline-Animation in Unity

In Unity sind zwei Animationssysteme implementiert: Die Komponente „Animation“ nutzt das klassische System (Legacy), „Animator“ dagegen das Mecanim-System. Letzteres verwendet neben den eigentlichen Animationsdaten einen zusätzlichen Datensatz, den (Animation-) Controller. Damit werden einzelne Animationsteile zueinander in Beziehung gesetzt und ein Zustandsautomat zur Steuerung und Überblendung derselben aufgebaut. Damit kann die Animationslogik zu einem gewissen Grad auf einer visuellen Ebene für die Programmierung vorbereitet werden. Hierauf wird in der Folge nicht weiter eingegangen. Zu beachten ist aber, dass beide Systeme mehrere Animationen an einem GameObject verwalten können, bei der Animation-Komponente beschränkt sich dies auf die Auswahl der zugeordneten Animationen und der Startanimation und erfolgt direkt dort.

Eine der beiden Komponenten Animation oder Animator muss an das zu animierende GameObject angeheftet werden. Die Animationsdaten beider Systeme lassen sich dann auf die gleiche Weise erzeugen und bearbeiten. Hierzu muss das zu animierende Objekt angewählt und der Animation-Editor geöffnet werden (Window→Animation oder Ctrl+6). Damit ist die zeitgesteuerte Animation des Objektes und aller diesem untergeordneter Hierarchieebenen möglich. Verfügen Child-Objekte selbst über Animationen, werden diese eingerechnet, sofern kein Widerspruch entsteht.

### 7.2.1 Dopesheet



Der Animation-Editor arbeitet in zwei unterschiedlichen Modi. Mit dem Dopesheet kann die Animation durch Aufnahme der Werte von Objekteigenschaften in Keyframes angelegt werden. Ist der Aufnahmebutton (roter Punkt) aktiv, entsteht bei Manipulation des Objektes an der Position des Playback-Heads (rote Linie) automatisch ein Keyframe und die zugehörigen Einträge für die veränderte Eigenschaft. Mit „Add Property“ kann auch explizit eine Eigenschaft ausgewählt werden. Sofern noch keine Animation definiert wurde, öffnet sich zuerst ein Dateidialog. Auch die Animationen werden nämlich als einzelne Dateien im Projekt abgelegt. Auf der linken Seite ist dann die Eigenschaften-Hierarchie dargestellt die es erlaubt, die Werte der Eigenschaften an den Keyframes per numerischer Eingabe zu verändern. Im Dopesheet können zudem leicht Keyframes oder Teile davon verschoben und kopiert sowie gelöscht werden.

### 7.2.2 Curve

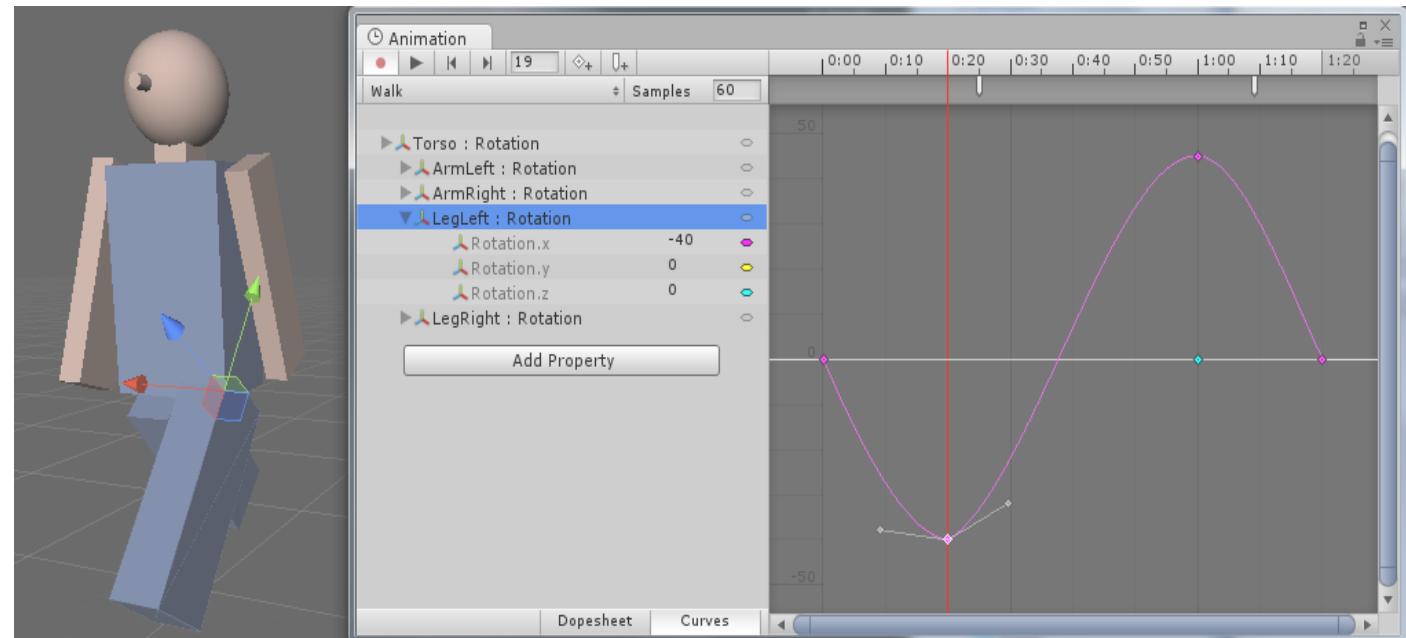
Bei der Arbeit mit dem Dopesheet legt Unity die Interpolation zwischen den Keyframes als Splines an. Dadurch entstehen sanfte Übergänge, manchmal aber auch Überschwingungen. Im Curve-Modus kann dies korrigiert und andere Interpolationen sowie abrupte Wertänderungen eingestellt werden. Es werden die Kurvenverläufe für die links selektierten Eigenschaften angezeigt. Die einzelnen Keys können hier mit Hilfe der Maus nicht nur

in der zeitlichen Abfolge, sondern auch in ihrem Wertebereich verschoben werden. Per Rechtsklick auf einen Key ist die Manipulation der Kurventangenten möglich.

Das Beispiel zeigt das Heben und Senken des linken Beins mit Hilfe der Animation der Rotation des Objektes „LegLeft“. An der Kurve kann man erkennen, dass die Bewegung schnell beginnt, zum Umkehrpunkt hin verlangsamt, dann in die Gegenrichtung wieder beschleunigt und schließlich abrupt endet. Die Tangenten am zweiten Key werden gerade bearbeitet.

Zu beachten ist, dass die Figur nur aus primitiven Körpern zusammengesetzt ist, deren lokales Koordinatensystem immer im Mittelpunkt liegt. Um das Bein an der oberen Kante drehen zu können, ist der Cube, der das Bein repräsentiert, als Child in das leere GameObject „LegLeft“ verschachtelt.

Dieses wiederum liegt auf der Unterkante des Torsos, lediglich um 15cm nach rechts verschoben. Der Cube jedoch ist in Bezug darauf um die Hälfte seiner Länge nach unten verschoben, und reicht somit vom Boden bis zum Hüftgelenk.



### 7.2.3 Tipps

Zur Navigation in der Zeitleiste ist das Mausrad äußerst hilfreich. Es zoomt um die aktuelle Mauscursorposition ein und aus, wobei die Dimension mit Ctrl und Shift eingeschränkt werden kann, und bei gedrücktem Mausrad oder mit gedrückter Alt-Taste scrollt die Zeitleiste mit der Mausbewegung.

Die Einheit der Zeitleiste besteht aus zwei Teilen. Vor dem Doppelpunkt stehen Sekunden, danach die Zahl des Frame in dieser Sekunde. Wie viele Frames in der Sekunde für die Definition von Keyframes zur Verfügung stehen sollen, wird in „Samples“ festgelegt.

Neben „Samples“ befindet sich die Auswahl der zu bearbeitenden Animation. Anders als in Flash, sollten bei Unity alle möglichen Bewegungen als einzelne Animation-Clips abgelegt werden wie „idle“, „walk“, „jump“, „punch“, „die“, „sit“, „standup“ etc. So lassen sie sich leicht verwalten und kombinieren. Zum Anlegen eines neuen Clips gibt es den Eintrag „Create New Clip“.

## 8 Skripte

Unity ermöglicht es, Skripte in C#- und Javascript-Syntax zu schreiben. Flash spricht ActionScript. ActionScript und Unity-Javascript sind sich sehr ähnlich und syntaktisch prinzipiell Dialekte der standardisierten Scriptsprache ECMA-Script. Somit können beide Systeme mit der gleichen syntaktischen Struktur behandelt werden. C# wird daher in dieser Veranstaltung nicht weiter behandelt.

Anders als Javascript im Webbrower werden ActionScript und UnityScript nicht zur Laufzeit als Klartext interpretiert, sondern zunächst in einen Zwischencode kompiliert. Dieser wird dann auf dem Zielsystem von einem just-in-time-compiler ausgeführt, und ist damit sehr schnell. Beide Sprachen gehen zudem deutlich über den von Javascript gewohnten Umfang hinaus und stellen ein Superset dar, ähnlich zu TypeScript:

- die explizite und statische Typisierung ist möglich und sollte unbedingt genutzt werden. Somit werden Fehler durch unpassende Typenkonversion schnell erkannt. Der Typ wird bei der Deklaration durch Doppelpunkt getrennt hinter den Namen geschrieben.
  - Beispiel      `var xyz:int = 1;`
  - Primitive Datentypen (Auswahl)
    - ActionScript:      `void, String, int, uint, Boolean, Number`
    - UnityScript:      `void, String, int, uint, boolean, float, double`
- Objektorientierte Programmierung mit Klassen, Interfaces, Vererbung und polymorphen Objekten ist mit den üblichen Schlüsselwörtern wie `class`, `extends`, `implements`, `super`, `this` etc. möglich. Auch wenn der Anfänger noch nicht explizit objektorientiert programmiert, sind im Hintergrund bereits diese Mechanismen am Werk und es ist empfehlenswert, sie mit fortschreitender Kenntnis auch gezielt und explizit zu nutzen.
- Sichtbarkeits- und Zugriffsmodifikation kann und sollte genutzt werden. In Unity werden öffentliche Variablen in der IDE angezeigt und ihre Werte können dort zur Lauf- und Entwicklungszeit per Tastatur oder Maus manipuliert werden. Mit Hilfe der direktive [`Inspectable`] kann man Ähnliches auch in Flash erreichen.

### 8.1 Konsole

Unity und Flash bieten beide eine Ausgabekonsole an, in der zur Laufzeit Fehler angezeigt, aber auch eigene Ausgaben gemacht werden können. Zum Öffnen und für die Ausgabe stehen folgende Tastenkürzel und Anweisungen zur Verfügung:

Flash:      F2                  `trace(...);`

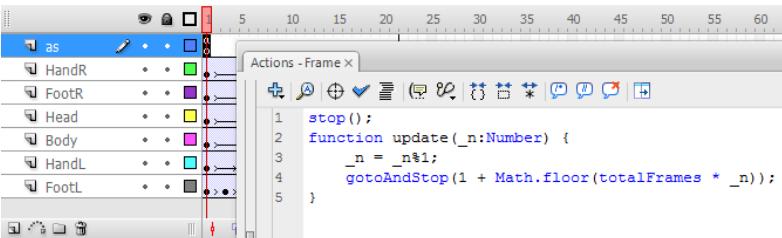
Unity        Ctrl+Shift+C      `print(...); Debug.Log(...);` (Debug bietet weitere Möglichkeiten zu Prioritäten etc.)

## 8.2 Skripte in Flash

Code kann in Flash auf verschiedenen Ebenen eingebunden werden: In der Timeline oder als externe Dateien. Letztere können automatisch Symbole erweitern, explizit importiert werden oder auf Basis der darin definierten Klassen zur Erzeugung von Objekten genutzt werden.

### 8.2.1 Zeitleisten-Skripte

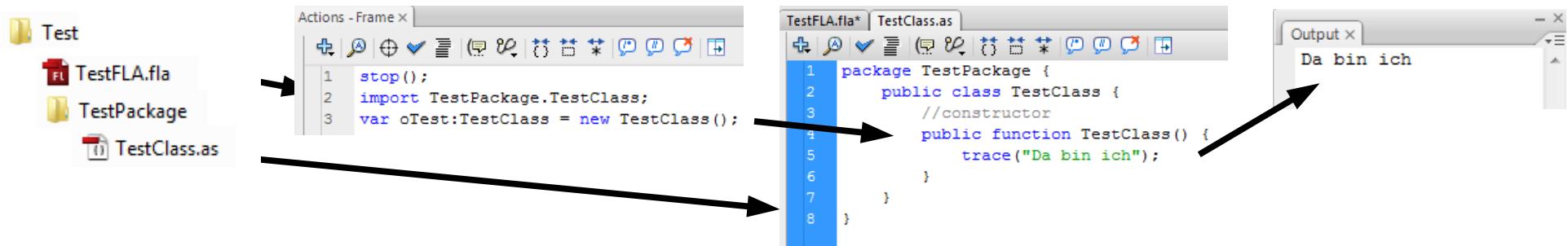
Für diese Skripte sollte ein eigener Layer eingefügt werden, um sie leichter warten zu können. Sie sind ab dem Keyframe, dem sie zugeordnet sind aktiv (Editor: F9). Timeline-Skripte eignen sich besonders für die Steuerung der Timeline selbst mit Hilfe von `stop()`, `play()` und `goto...()`-Anweisungen. `this` verweist zudem auf den MovieClip, dem die Timeline zugeordnet ist, so dass dessen Eigenschaften direkt manipuliert werden können (z.B. die Position mit `this.x`). Wenn beim Kompilieren die Subklassen für die MovieClips generiert werden, werden die Timelineskripte als Methoden dort „eingebacken“.



### 8.2.2 Externe Skripte

Include: Der einfachste Fall Code in eine externe Datei auszulagern um ihn an verschiedenen Stellen wieder zu verwenden ist, die Datei mit `include ...` in das Skript einzubinden. Der so eingebettete Code wird dann allerdings tatsächlich an jeder Stelle erneut kompiliert.

Import: Klassen und ganze Packages können importiert werden. Dabei muss auf die korrekte Benamung der Dateien, Packages und Klassen geachtet werden. Die Dateinamen müssen identisch zu den Klassennamen plus „.as“ sein. Sofern diese sich in einem anderen Ordner als die FLA Datei befindet, muss zudem der Packagename dem Ordnernamen entsprechen. Hinter die Anweisung `import ....` wird kein String geschrieben, sondern der Verweis auf die Klasse in der Form „`Package.Class`“, also mit Punkt-Syntax.



Implizit: AS-Dateien, die sich zusammen mit der FLA-Datei in einem Ordner befinden, werden automatisch importiert. Allerdings darf dann auch kein Package-Name in den Dateien vergeben sein, sie beginnen also mit package { ... }. Die Klassen können dann sofort genutzt und Objekte davon erstellt werden.

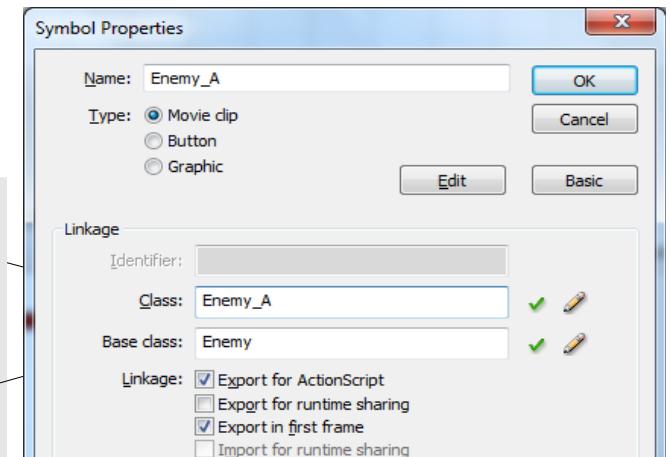
Bindung an Symbole: Klassen, welche direkt oder indirekt von MovieClip abgeleitet sind, können als Symbolklasse dienen. Sofern mehrere Symbole von der gleichen Klasse sein sollen, wird die Klasse als Basisklasse angegeben, da die Symbole auf Grund der Unterschiede in der Timeline selbst wieder einzelne Subklassen darstellen. Die Zuordnung geschieht einfach über den Eigenschaften-Dialog des Symbols. Dazu muss „Export für ActionScript“ aktiv sein.

Sofern aus einer solchen Basisklasse heraus untergeordnete MovieClips der Timeline angesprochen werden, müssen diese in der Basisklasse deklariert werden. Die automatische Deklaration muss dann deaktiviert werden:

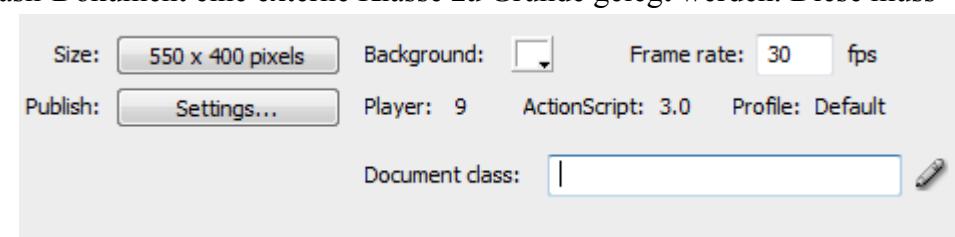
File→Publish Settings→Flash→ActionScript Settings→Stage

Klasse Enemy\_A wird erzeugt unter Berücksichtigung der Timeline.

Klasse Enemy ist extern und direkt oder indirekt von MovieClip abgeleitet.



Dokumentklasse: Analog zur Bindung an Symbole kann auch dem ganzen Flash-Dokument eine externe Klasse zu Grunde gelegt werden. Diese muss ebenso von MovieClip abgeleitet sein. In dieser Klasse bringt man idealerweise Methoden unter, welche man von verschiedenen Stellen der Applikation aus nutzen möchte, oder die man als Standardmethoden in weiteren Projekten auf dieser Ebene wieder verwenden möchte. Die Dokumentklasse wird angegeben im Eigenschaften-Dialog des Dokuments.



### 8.3 Skripte in Unity

Alle Skripte sind bei Unity in eigenen Dateien untergebracht. Bei Javascript erhalten diese automatisch die Endung „.js“. Erzeugt werden neue Skripte einfach mit Project→Create→Javascript oder direkt auf einem GameObject mit AddComponent→New Script. Ein Doppelklick auf das Skript öffnet in der Standardinstallation von Unity den dort mitgelieferten externen Editor „MonoDevelop“. Unter Edit→Preferences→External Tools kann auch ein anderer Editor eingetragen werden, die Screenshots in dieser Dokumentation stammen aus Notepad++ bzw. UniSciTe.

Die Skripte sind eigenständige Komponenten, die GameObjects angehängt werden können. Sie sind Subklassen von „MonoBehaviour“, welche über „Behaviour“ von „Component“ abgeleitet ist. Diese Klasse stellt eine Vielzahl von Methoden zur Verfügung, die überschrieben werden können und automatisch von Unity aufgerufen werden, wie:

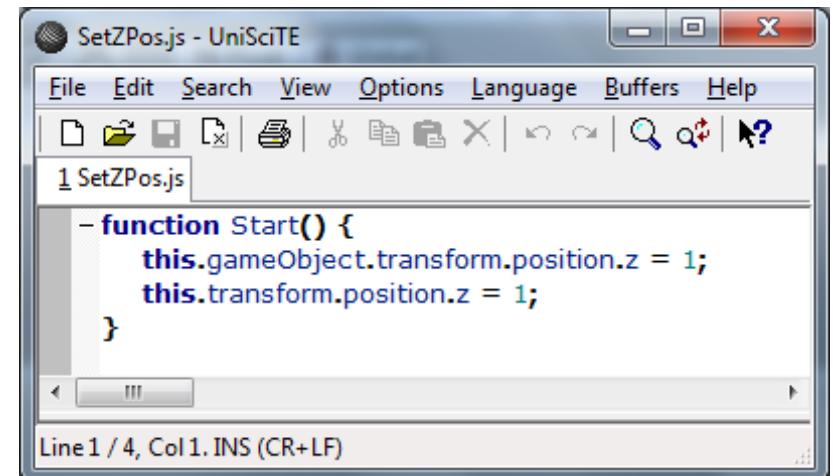
- `Update()`, wird bei jeder Bildaktualisierung aufgerufen,
- `OnMouseDown()`, wird aufgerufen, wenn die Maustaste gedrückt wurde, während der Zeiger über einem Collider oder einem GUI-Element des gleichen GameObjects steht,
- `OnCollisionEnter(Collision collisionInfo)`, wird aufgerufen, wenn der Collider oder Rigidbody des GameObjects mit einem anderen Collider/Rigidbody kollidiert.

Zu beachten ist, dass `this` in einem Skript nicht auf das GameObject verweist, an dem das Skript hängt, sondern auf die Instanz des Skriptes selbst. Um das GameObject mit Hilfe des Skripts zu manipulieren, kann mit Hilfe der Komponenteneigenschaft `gameObject` darauf zugegriffen werden, respektive also mit `this.gameObject`. Das GameObject selbst verfügt aber über nur wenige Eigenschaften, welche direkt manipuliert werden können, nämlich `name`, `hideFlags`, `isStatic`, `layer`, `tag` und `active`. Alles andere, wie Transformation und Erscheinung, ist in den angehängten Komponenten festgelegt. Diese sind mit Hilfe der Methode `GetComponent(...)` oder `GetComponents(...)` erreichbar.

Lediglich die Transformkomponente ist über die Eigenschaft `transform` sowohl über das GameObject als auch über alle anderen diesem angefügten Komponenten erreichbar. Die nebenstehenden Zeilen zur Manipulation der Position beispielsweise sind also gleichbedeutend, da auch das Skript als Komponente die Eigenschaft `transform` hat, die beim Anheften auf die Transform-Komponente des GameObjects verweist.

An einem GameObject können mehrere unterschiedliche Skript-Komponenten hängen. Sollen Methodenaufrufe zwischen den Skripten erfolgen muss mit angegeben werden, auf welche Komponente sich der Aufruf bezieht. Da jedes Skript als eigener Objekttyp unter dem Namen des Skripts bekannt ist, kann die Komponente mit der Anweisung `GetComponent(type)` gefunden werden. Soll also aus Skript1 die Methode `Test()` in Skript2 am gleichen GameObject aufgerufen werden, so kann dies mit folgender Zeile bewerkstelligt werden:

```
this.GetComponent(Skript2).Test();
```



```
SetZPos.js - UniSciTE
File Edit Search View Options Language Buffers Help
1 SetZPos.js
function Start() {
    this.gameObject.transform.position.z = 1;
    this.transform.position.z = 1;
}

Line 1 / 4, Col 1. INS (CR+LF)
```

### 8.3.1 Explizite Klassen

Zunächst nicht sichtbar legt Unity bei jedem Script eine eigene Klasse an, welche von `MonoBehaviour` abgeleitet es. Als Klassename wird hierbei der Dateiname heran gezogen. Bei der Benennung des Skriptes ist dies zu beachten, da Konflikte mit Standardklassen entstehen können. Oft ist es sinnvoll, die Klasse explizit zu definieren (`class MyClass extends MonoBehaviour`). Wird statt von `MonoBehaviour` von `ScriptableObject` abgeleitet, kann eine statische Klasse implementiert werden, die nicht an ein GameObject gebunden werden muss!

## 8.4 Datentypen

	Flash	Unity
Vektor	Point	Vector2, Vector3, Vector4
Matrix	Matrix	Transform, Matrix4x4

## 8.5 Vektor-Methoden und -Eigenschaften

	Flash (Point)	Unity (Vektor2, Vektor3, Vektor4)
Komponenten	x, y	x, y (, z (, w))
Länge	length	magnitude
Normalisieren	normalize(newlength)	Normalize(), normalized
Addieren	add(Point)	+
Subtrahieren	subtract(Point)	-
Interpolieren	interpolate(Point, Point, Number)	Lerp(Vector, Vector, float)
Distanz	distance(Point, Point)	Distance(Vector, Vector)
Gleichheit	equals(Point)	==
Weiteres	polar, clone	*, /, MoveTowards, Scale, Dot, Cross, Reflect, Angle.....

## 8.6 Eigenschaften von Transformationen

	Flash	Unity
Position	DisplayObject.x/.y , Matrix.tx/.ty	Transform.position.x/.y/.z
Rotation	DisplayObject.rotation	Transform.eulerAngles/.localEulerAngles/.localRotation/.rotation (Quaternionen)
Skalierung	DisplayObject.scaleX/.scaleY	Transform.localScale

## 8.7 Anweisungen zur Transformation

	Flash	Unity
Verschieben	DisplayObject.x/.y +/- Matrix.translate(dx,dy)	Transform.Translate Transform.position +/-, .x/.y/.z +/-
Rotieren	DisplayObject.rotation +/- Matrix.rotate(angle)	Transform.eulerAngles +/-, .x/.y/.z +/- Transform.Rotate/.RotateAround/.LookAt
Skalieren	DisplayObject.scaleX/.scaleY +/- Matrix.scale(sx,sy)	Transform.localScale +/-, .x/.y/.z +/-
Umrechnung lokal->global	DisplayObject.localToGlobal(Point)	Transform.TransformPoint Transform.localToWorldMatrix
Umrechnung global->lokal	DisplayObject.globalToLocal(Point)	Transform.InverseTransformPoint Transform.worldToLocalMatrix

## 9 Logische und geometrische Beziehungen

Beziehungen zwischen Klassen und deren Objekten zu definieren und herzustellen ist eine Kernaufgabe bei der Entwicklung objektorientierter Programme. Bei Spielen sind diese Objekte häufig sichtbare Elemente auf dem Bildschirm, sie können aber ebenso völlig abstrakt sein. Bei großen und komplexen Spielen wird sogar häufig zwischen der Logik und der Visualisierung getrennt, so dass das Spiel prinzipiell auch völlig ohne grafische Ausgabe lauffähig ist und die Visualisierung lediglich die Zustände des abstrakten Modells darstellt.

Die Qualität der Modellierung der Spielwelt als Netz von Objektbeziehungen hat entscheidenden Einfluss auf die gesamte nachfolgende Entwicklung. Ein wenig durchdachtes Modell zieht einen großen Aufwand nach sich und führt zu undurchsichtigem und fehleranfälligem Programmcode. Auch schon bei kleinen Spielen lohnt es sich daher, sich im Vorfeld Gedanken zum Aufbau von Objektbeziehungen zu machen.

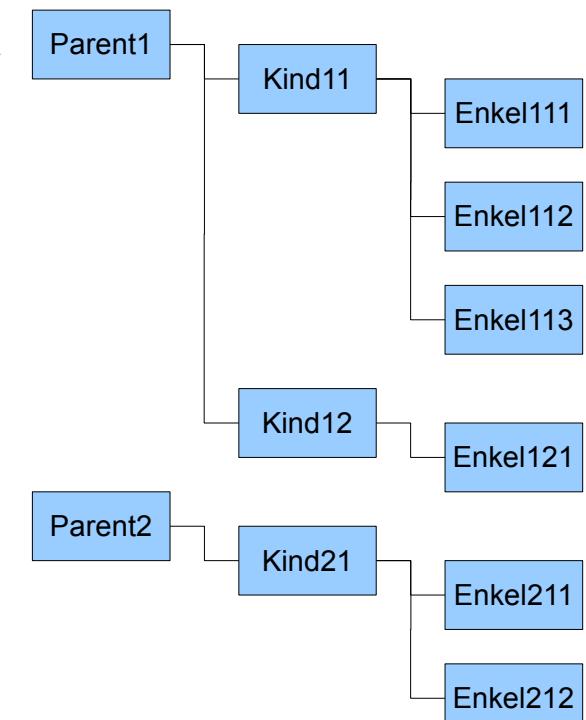
Die folgenden Betrachtungen beziehen sich vordringlich auf Beziehungen zwischen Objekten mit grafischer Repräsentation, also MovieClips in Flash oder GameObjects in Unity. Das Besondere daran ist nämlich, dass diese Objekte für die Visualisierung grundsätzlich bereits von Flash bzw. Unity in einer vordefinierten Struktur verwaltet werden, der Displaylist bzw. dem Scenetree.

### 9.1 Szenenbaum und Displaylist

Die einfachste Möglichkeit Beziehungen zwischen Objekten herzustellen, ist diese im Szenenbaum von Unity bzw. in der Displaylist in Flash in eine Parent-Child-Beziehung zu bringen. Dies drängt sich natürlich intuitiv auf, wenn die Kindobjekte Teil einer größeren Struktur sind und sich mit dieser bewegen, also die Transformationen übernehmen sollen. Beispielsweise für die Räder als Kinder eines Parent-Fahrzeugs genügt es dann, um die eigene Achse zu rotieren und etwas auf und ab zu federn. Bewegt sich das Fahrzeug voran, kommen die Räder automatisch mit.

Der Szenenbaum kann aber auch zur logischen Gruppierung von Objekten genutzt werden, um diese nicht für jede Manipulation in der Szene suchen zu müssen oder weitere Parallelstrukturen wie Arrays hierfür aufzubauen zu müssen. Die im Spiel eingesammelte Objekte eines Inventars können beispielsweise als Children eines nur hierfür vorgesehenen Parent-Objektes verwaltet werden, wobei die geometrische Beziehung nicht von besonderer Relevanz ist.

Als Daumenregel sollte aber der direkte Zugriff per Skript durch eine Verkettung von Beziehungen nicht über eine Ebene hinaus gehen. Konstrukte wie `this.parent.parent.childX.doSomething()` sind zu vermeiden. Änderungen im Szenenbaum ziehen ansonsten Änderungen im Code nach sich, die nur noch schwer nachzuvollziehen sind.



*Parent-Child-Beziehungen im Szenenbaum bzw. der Displaylist. Die Namen sind willkürlich zur Orientierung gewählt*

## 9.2 Aggregation

Die Skripte können auch direkte Verweise auf andere Objekte mit Hilfe von entsprechenden Eigenschaftsvariablen verwalten. Somit können Objekte unabhängig von der geometrischen Hierarchie in Beziehung gesetzt werden und kommunizieren. Wenn Objekte vom betreffenden Skript erzeugt werden, ist diese Beziehung leicht herzustellen (siehe Instanziierung zur Laufzeit). Sofern die zu referenzierenden Objekte aber bereits existieren, müssen sie zunächst gefunden werden. Die hierfür zu verwendenden Mechanismen werden in der Folge für Flash und Unity getrennt beschrieben.

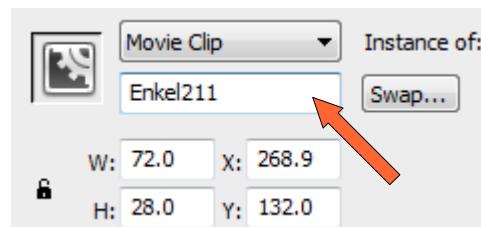
## 9.3 Ereignisse und Nachrichten

Neben dem oben erwähnten Suchen nach Objekten, können Objekte selbst Nachrichten durch die Szenenhierarchie versenden und sich auf diese Weise finden lassen. Ein anderes Objekt, welches die Nachricht erhält, kann darauf bei Bedarf reagieren oder sich den Absender merken, um später mit ihm zu kommunizieren. Auch bestimmte Ereignisse wie Mausklicks oder Kollisionen können automatisch Nachrichten erzeugen, welche Verweise auf betroffene Objekte mitliefern (siehe Observer: Events und Messages).

## 9.4 Klassenhierarchie

Eine weitere, rein logische Beziehung nicht von Objekten aber von Klassen, ist die Erweiterung mit Hilfe der Vererbung. Hiermit bezieht sich eine Klasse auf die Fähigkeiten und Eigenschaften einer anderen Klasse und erweitert diese lediglich um Spezifika. Richtig verwendet ist dies ein mächtiges Werkzeug um ein robustes, modulares und gut zu pflegendes System zu kreieren.

## 9.5 Flash



MovieClips in der Szene können mit Hilfe des Eigenschaften-Fensters mit einem Namen versehen werden. Im Programmcode eines Objektes können dessen Kinder dann direkt über diesen Namen referenziert werden, z.B. bei Parent2 `this.kind21`. Hat `kind21` selbst wieder benannte Kinder (z.B. `enkel211`) in der Playlist, können diese ebenso von dieser Ebene mit Hilfe der Punkt-Syntax erreicht werden: `this.kind21.enkel211`. Dieses Verfahren ist sehr intuitiv und empfehlenswert für unveränderliche hierarchische Strukturen, welche in der IDE angelegt werden. Werden keine Namen explizit vergeben, legt Flash sie automatisch zur Laufzeit nach dem Muster „instance“ + Index an, wobei der Index eine fortlaufende Zahl ist. Die Eigenschaft kann auch per Skript als Attribut „name“ gelesen und geschrieben werden.

```
for (var i:int = 0; i< this.numChildren; i++) {  
    trace(this.getChildAt(i).name);  
}
```

Dieser Code in `Kind21` erzeugt die Ausgabe "Enkel111", "Enkel112" und "Enkel123"

Für die andere Richtung, also nach oben in der Hierarchie, genügt das Schlüsselwort `parent`, da jedes Objekt nur ein Elternobjekt haben kann.

Enkel113 und seine „Geschwister“ haben alle den gleichen Parent: Kind11.

Um weiter nach oben in der Hierarchie zu kommen, sind die Eigenschaften `stage` und `root` verfügbar, die bei jedem Objekt in der Displaylist automatisch gesetzt werden. Dabei ist darauf zu achten, dass die beiden nicht identisch sind. Die Stage hat Eigenschaften wie `stageWidth` oder `quality` und bezieht sich auf das Anzeigefenster. Root dagegen ist ein DisplayObject und bezeichnet das Wurzelobjekt in einer SWF-Datei. Werden in einer Flash-Anwendung mehrere SWF-Dateien hinzugeladen, gibt es entsprechend viele Root-Objekte welche von den darin liegenden Objekten referenziert werden. Stage dagegen gibt es nur einmal und jedes angezeigte Element hat eine Referenz darauf. Anmerkung: die DocumentClass erweitert die Root-Klasse für die jeweilige SWF.

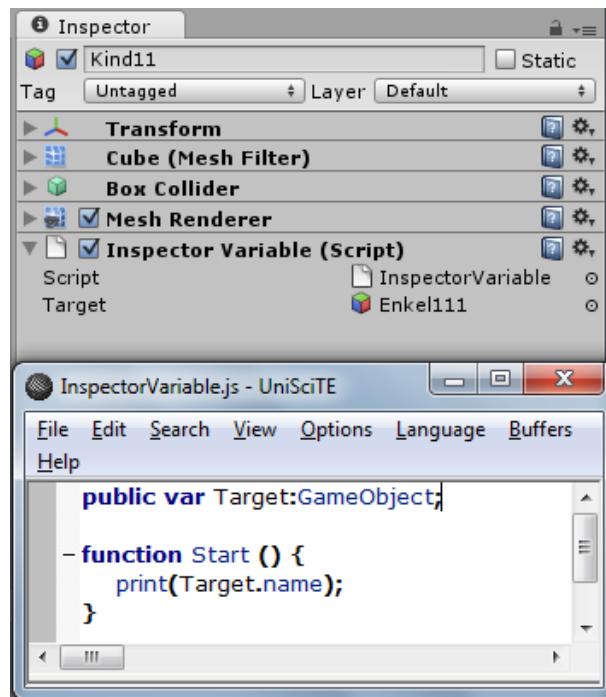
Referenzen auf Objekte können Methoden übergeben werden. So ist es einem Elternobjekt (z.B. der Stage bzw. Root) leicht möglich, Kinder in verschiedenen Ästen der nachfolgenden Hierarchie miteinander bekannt zu machen. Ein beispielhafter Aufruf könnte so aussehen:

```
this.root.parent2.follow(this);
```

Sollen die Pfade durch die Hierarchie veränderlich sein oder prinzipiell nicht hart kodiert werden, wird es kniffliger. Flash bietet keine effizienten Methoden an, um Objekte in der Hierarchie zu suchen. In diesem Falle ist es sinnvoll, wenn die Objekte sich zunächst bei ihrem Wurzelobjekt oder der Stage „anmelden“ und diese dann die Referenzierung dieser Objekte untereinander per Methodenaufruf vornehmen. Dann kann zudem auf die Benamung der Instanzen in der IDE verzichtet werden, da nur Referenzen (wie z.B. `this`) übergeben werden.

Für die Herstellung von Objektbeziehungen in einer variablen Hierarchie kann zudem das Event-System genutzt werden. Damit können auch andere Objekte als Root oder Stage die Verknüpfung übernehmen. Mehr dazu im entsprechenden Kapitel Observer: Events und Messages.

## 9.6 Unity



Eine direkte Referenzierung von Objekten über den Namen und die Punkt-Syntax ist in Unity nicht vorgesehen. Dafür gibt es aber eine andere, sehr elegante Möglichkeit mit Hilfe der IDE Objektbeziehungen jenseits der geometrischen Hierarchie anzulegen. Als `public` deklarierte Variablen im Kopf eines Skripts erscheinen im Inspector-View und lassen sich dort mit einem Initialwert befüllen. Variablen vom Typ `Transform` oder `GameObject` können so sehr bequem per Drag&Drop aus dem Hierarchy-View mit Referenzen auf Spielobjekte bzw. deren transform-Komponente besetzt werden. Im weiteren Verlauf des Skripts kann das Objekt direkt mit dem entsprechenden Variablenamen adressiert werden.

Um Objekte im Szenenbaum zur Laufzeit zu finden, sind in Unity mehrere Möglichkeiten implementiert. Da die Szenenhierarchie durch die Parent-Child-Beziehungen der Transform-Komponenten definiert ist, bietet diese bereits Grundlegendes an. Sie besitzt die Eigenschaften `parent` und `childCount` sowie `root`, welche eine Referenz auf das oberste Objekt in der Hierarchie enthält. Die Transform-Komponente implementiert zudem das Iterator-Interface um bequem die Kinder durchzugehen.

```
for (var Child:Transform in transform) {
    print(Child.name);
}
```

Dieser Code in `Kind11` erzeugt die Ausgabe "Enkel111", "Enkel112" und "Enkel123"

Anders als bei Flash, wo `root` auf den SWF-Container verweist, enthält dieses Attribut bei Unity eine Referenz auf das oberste GameObjekt im jeweiligen Ast. Für `Enkel121` beispielsweise ist das `Parent2`.

Weiterhin bietet die Transform-Komponente die Methode `Find (Name)`, welche die Transform-Komponente des Kindes mit dem gesuchten Namen liefert. Auf Enkel und Urenkel etc. kann zugegriffen werden, indem als Parameter der Pfad durch den Ast der Hierarchie angegeben wird, z.B. "`kind12/enkel121/urenkelXY`".

Objekte in beliebigen Ästen der Hierarchie können mit der Methode `Find (Name)` der Klasse `GameObject` gefunden werden. Da dies relativ Zeitaufwändig ist, sollten einmal gefundene Referenzen gespeichert werden.

Objekte können in Unity auch mit einer Markierung, einem sogenannten „Tag“ versehen werden. Auf diese Weise können sie schnell gefunden werden und mehrere Objekte mit dem gleichen Tag können als Gruppe über Hierarchiegrenzen hinweg behandelt werden. Hierfür stellt die `GameObject`-Klasse die Methoden `FindWithTag (....)` und `FindGameObjectsWithTag (....)` zur Verfügung.

Hinweis: OnCollision- und OnTrigger-Handlern wird eine Referenz auf den Kollisionspartner automatisch übergeben!

## 10 Instanziierung zur Laufzeit

Szenen und Animationen in der IDE zu erstellen ist enorm praktisch und spart Entwicklungszeit. In der Regel ist es aber auch erforderlich, zur Laufzeit Objekte in der Szene neu zu platzieren oder auch wieder verschwinden zu lassen. Hierzu müssen Objekte instanziert und auch wieder vernichtet werden, sowie gezielt im Szenenbaum oder der Displaylist eingefügt und wieder entfernt werden.

### 10.1 Symbole und DisplayObjects

Eine Instanz eines Symbols wird in Flash per Skript, wie andere Klassen auch, schlicht mit `new` kreiert. Speichert man die dabei erzeugte Referenz auf das neue Objekt in einer Variablen, kann man dieses Objekt mit `addChild(...)` einem DisplayObjectContainer als Kind anfügen. Der Name der Variablen ist dabei unabhängig von der Eigenschaft `name` des Objektes. Die Referenz kann zur weiteren Manipulation des Objektes mit Hilfe der Punkt-Syntax genutzt werden, so dass häufig das `Name`-Attribut des Objektes nicht relevant ist. Selbstverständlich kann der Konstruktor auch überschrieben werden, so dass z.B. auch Parameter bei der Objekterzeugung übergeben werden können.

Zu beachten ist, dass die Reihenfolge der Kinder deren Darstellung beeinflusst. Die zuletzt angefügten Objekte erscheinen oberhalb der anderen im entsprechenden Container. Um diese Reihenfolge gezielt zu kontrollieren stehen die Methoden `addChildAt(...)` sowie `setChildIndex(...)`, `swapChildren(...)`, `swapChildrenAt(...)` zur Verfügung.

Ein Objekt kann aus der Displaylist mit `removeChild(...)` oder `removeChildAt(...)` wieder entfernt werden.

Eine gezielte Zerstörung eines Objektes ist nicht erforderlich. Sobald keine Referenzen auf das Objekt mehr gespeichert sind, wird es für die Garbage Collection freigegeben. Diese räumt den Speicher auf, sobald das System genug Zeit dafür hat oder der Vorgang forciert wird. Wenn das Objekt also nicht mehr in der Displaylist erscheint, und keine Variablen mehr darauf zeigen, indem sie z.B. den Wert `null` erhalten haben, ist das Objekt verloren.

### 10.2 GameObjects und Prefabs

Um ein Objekt in Unity zur Laufzeit zu erzeugen wird die spezielle Anweisung `Instantiate(...)` verwendet. Der Methode wird ein Objekt übergeben, im einfachsten Fall mit Hilfe einer Inspector-Variablen (siehe Objektbeziehungen). So kann auch leicht ein Prefab referenziert werden. Bei der Erzeugung kann auch gleich ein Positionsvektor und ein Quaternion für die Orientierung mitgegeben werden.

```
var Item:GameObject;  
  
-function Start () {  
    var oItem = Instantiate(Item, Vector3(10,0,0), Quaternion.identity);  
    oItem.transform.parent = this.transform;  
}
```

Dieser Code erzeugt ein Objekt vom Typ `Item` an der Weltkoordinate (10,0,0) und ordnet es dem Objekt des Skripts unter.

Das neue Objekt erscheint auf der obersten Hierarchieebene. Um es an einer bestimmten Stelle im Szenenbaum einzufügen, muss das Attribut `parent` der Transform-Komponente mit einer Referenz auf die Transform-Komponente des gewünschten Elternobjekts überschrieben werden. Die Gesamttransformationsmatrix des Objektes bleibt dabei erhalten, und somit seine Lage und Erscheinung in der Welt.

Der Anweisung `Instantiate(...)` steht `Destroy(...)` gegenüber, mit der ein Objekt entfernt und der Speicher vom Garbage-Collector freigegeben werden kann.

### 10.2.1 Resource-Folder

Um ein Prefab zu instanziieren, welches nur über den Namen und nicht per Inspector-Variable referenziert werden soll, muss dieses in einem speziellen mit „Resources“ benannten Ordner im Projekt liegen. Dann kann es mit Hilfe der Resources-Klasse geladen und zur Instanziierung genutzt werden.

```
res = Resources.Load("PrefabName") as GameObject;  
var inst:Transform = Instantiate(res.transform, Vector3(i,0,0), Quaternion.identity);
```

# 11      Observer: Events und Messages

Mit Hilfe des Event-Systems in Flash und des Message-Systems in Unity ist es möglich, Nachrichten und Ereignisinformationen zwischen Spielobjekten zu verschicken, ohne die Objekthierarchien im Programmcode wieder abilden zu müssen. Die Informationen wandern selbstständig durch den Szenenbaum bzw. die Displaylist, und können von den Objekten darin abgefangen werden. Dabei stellt es auch kein Problem dar, wenn eine Nachricht zwar verschickt wird, aber kein Empfänger sie entgegen nimmt. Die Objekte selbst müssen sich also gegenseitig nicht kennen und es müssen keine expliziten Objektbeziehungen hergestellt werden. Allerdings ist es erforderlich, dass die Objekte in der Szenenhierarchie in einer direkten Linie liegen. Eine Nachricht kann also nicht ohne Weiteres in einen benachbarten Ast geschickt werden.

## 11.1     Custom-Events in Flash

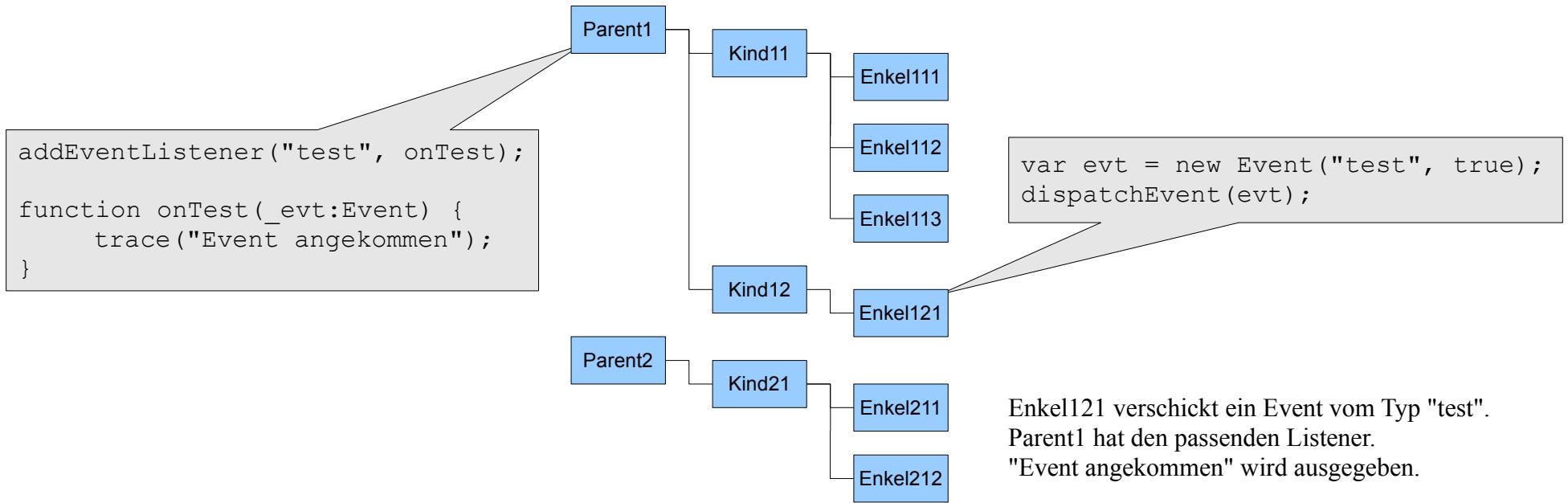
Das Event-System stellt die Grundlage der Programmierung von Flash mit ActionScript 3 dar. Es ist eine Implementation des ECMA-Standards, wie auch Javascript im Webbrowser. Ständig werden Nachrichten durch das System geschickt, welche an verschiedenen Stellen abgehört und daraufhin Methoden aufgerufen werden. Custom-Events sind dabei solche, die nicht wie System- (z.B. bei Mausklick) automatisch verschickt, sondern explizit von einem Skript definiert, generiert und verschickt werden.

Ein Event ist in der einfachsten Form ein Datenobjekt, Event ist ebenso der Name der Basisklasse aller Ableitungen davon. Die Event-Klasse ist direkt abgeleitet von Object, der obersten Superklasse in ActionScript. Das Attribut type ist schlicht ein String und wird genutzt um die Art des Events zu identifizieren. Im einfachsten Fall kann mit `var evt = new Event("...", true, ...)` ein Custom-Event erzeugt werden. Der erste Parameter ist dabei eine beliebige Zeichenkette, der zweite sollte den Wert true haben, damit das Event durch die Displaylist bis zur root wandern kann. Dieser Wert wird nämlich der Eigenschaft bubbles des Events zugewiesen. Das Event steigt dann wie eine Blase durch die Displaylist nach oben.

Dafür muss es zunächst losgeschickt werden, dies geschieht mit dem Befehl `dispatchEvent(evt)`, den jede Subklasse von EventDispatcher unterstützt, also auch jedes DisplayObject. Beginnend mit dem Objekt, welches das Event so verschickt hat, wird geprüft, ob dieses Objekt auf das Event reagieren soll und ggf. die Reaktion ausgelöst. Dann wird das Event in der Szenenhierarchie weiter nach oben gereicht und der Vorgang wiederholt.

Soll ein Objekt auf ein Event reagieren, dann muss es auch ein „Ohr“ dafür haben, einen sogenannten Listener. Dieser wird dem Objekt einfach mit der EventDispatcher-Methode `addEventListener(...)` angefügt. Als Parameter werden übergeben der Event-Typ, also die identifizierende Zeichenkette, und eine Referenz auf die Methode, welche bei Eintreten des Ereignisses aufgerufen werden soll. Diese Methode schreibt man in der Regel selbst, wobei darauf geachtet werden muss, dass sie das Event als Parameter entgegen nimmt. Dadurch ergibt sich ein Standardmuster für einen solchen „Handler“: `function onMyEvent(_evt:Event) {....}`

Das Event-System erscheint zunächst recht kompliziert, ist aber sehr mächtig und äußerst flexibel. Die eigentliche Stärke entfaltet es, wenn die Events wichtige Informationen durch das System tragen. Bei Custom-Events erweitert man hierzu die Event-Klasse und fügt beliebige Attribute oder



Methoden an System-Events, die automatisch verschickt werden, tragen in der Regel bereits eine Fülle von verwertbaren Daten. Diese Events können, anders als Custom-Events, die Hierarchie auch nach unten wandern! Mehr dazu in folgenden Kapiteln.

## 11.2 Messages in Unity

Das Message-System in Unity spielt eine weniger zentrale Rolle als das Event-System in Flash. Für bestimmte Aufgaben ist die Verwendung aber äußerst praktikabel. Es stehen drei Methoden zur Verfügung. `SendMessage` (Methodename, Parameter, Options) sucht in den Skript-Komponenten des aufrufenden GameObjects nach Methoden mit dem passenden Namen und übergibt diesen den Parameter von beliebigem Typ. `SendMessageUpwards(...)` macht das gleiche und wiederholt dies bei allen Objekten, welche in der Hierarchie oberhalb des Objektes stehen bis zum Root-Objekt. `BroadcastMessage(...)` arbeitet analog aber in die andere Richtung, ruft also die Methoden außer im aufrufenden Objekt auch in seinen Kindern, Enkeln, usw. auf. Das Argument Options gibt an, ob es wenigstens einen Empfänger für die Nachricht geben muss, ansonsten wird ein Fehler ausgegeben, oder ob die Nachricht auch „ungehört“ verstreichen darf.

Daneben existiert ein „Event-System“ welches aber vordringlich für Nutzerinteraktion insbesondere im virtuellen Userinterface verwendet wird.

## 12 Zeitverhalten

In der „klassischen“ Spieleprogrammierung gibt es meist eine sogenannte Game-Loop. Man kann sich dies als Schleife vorstellen, die so lange läuft, bis das Spiel beendet wird. Innerhalb der Schleife werden bestimmte Methoden der verschiedenen Elemente des Spiels immer wieder aufgerufen. Erst damit ist es möglich, dass sich der Zustand des Spiels auch ohne Nutzereinfluss ändert, Objekte sich bewegen oder rechnergesteuerte Spielfiguren zustandsabhängige Entscheidungen treffen. Im einfachsten Fall läuft diese Game-Loop so schnell wie möglich, was dann gegebenenfalls die erreichbare Bildwiederholrate widerspiegelt. In anderen Fällen kann auch eine Zeit-Taktung implementiert sein oder bestimmte Prozesse werden häufiger angestoßen als andere. So ist in der Regel eine flüssige Animation wünschenswert, also eine möglichst häufige Aktualisierung der grafischen Darstellung, während es gegebenenfalls für Prozesse der Spiellogik genügt diese nur in größeren Zeitintervallen aufzurufen.

In Unity und in Flash ist es nicht erforderlich eine zentrale Game-Loop zu implementieren. Vielmehr laufen die Prozesse automatisch im Hintergrund und man kann eigene Skripte von dort aufrufen lassen. In Flash erreicht man dies mit Hilfe des Event-Systems, bei Unity werden dazu vordefinierte Methoden überschrieben. Jedes einzelne Spielobjekt kann damit selbstständig auf den Zeitverlauf reagieren. Zu unterscheiden ist dabei zwischen einer zeitbasierten Taktung und der Abhängigkeit von der Bildwiederholrate.

### 12.1 EnterFrame und Timer in Flash

Zu Beginn jeder Bildwiederholung wird bei Flash ein bestimmtes Event durch die Displaylist geschickt. Der Typ ist eine Konstante der Event-Klasse und als Event.ENTER\_FRAME erreichbar. Somit genügt nebenstehender Code, um ein Objekt bei jeder Bildwiederholung um ein Pixel zu verschieben.

```
var speed:Number = 3.0;
var time:int = getTimer();
this.addEventListener(Event.ENTER_FRAME, update);

function update(_evt:Event) {
    var elapsed:int = getTimer() - time;
    this.x += speed*elapsed;
    time += elapsed;
}
```

Für Aufrufe in größeren oder von der Bildwiederholrate möglichst unabhängigen Intervallen bietet sich die Timer-Klasse an. Nebenstehender Code bewirkt, dass der update-Handler zwanzig mal im Abstand von 100 Millisekunden aufgerufen wird.

```
this.addEventListener(Event.ENTER_FRAME, update);

function update(_evt:Event) {
    this.x += 1;
}
```

Hierbei wird ersichtlich, dass das Objekt langsamer wird, wenn die Bildwiederholrate sinkt. Sollte aber die Geschwindigkeit beibehalten werden, so muss die Zeit mit eingerechnet werden. Die Methode getTimer() liefert die Zeit in Millisekunden seit Programmstart. Damit kann die Zeitspanne zwischen den Aufrufen der Methode gemessen und mit einer Geschwindigkeit multipliziert werden. Sinkt nun die Bildwiederholrate, so macht das Objekt größere Sprünge und behält damit seine Geschwindigkeit bei. Der Nebenstehende Code bewirkt genau dies.

```
var tick:Timer = new Timer(100,20);
tick.addEventListener(TimerEvent.TIMER, update);
tick.start();
```

## 12.2 Update in Unity

Die Methoden `Update()`, `LateUpdate()` und `FixedUpdate()` eines Unity-Skriptes werden automatisch aufgerufen, sofern dessen Eigenschaft `enabled` den Wert `true` besitzt. Dabei wird die `Update`-Methode mit der Bildwiederholrate aufgerufen. Die `LateUpdate`-Methoden werden aufgerufen, wenn alle Updates abgearbeitet sind. Damit kann z.B. sicher gestellt werden, dass eine Kamera ein schnelles Objekt nicht aus dem Sichtfeld verliert.

Um gleichmäßige Bewegungsgeschwindigkeiten unabhängig von der Bildwiederholrate zu ermöglichen, muss ebenso wie bei Flash die Zeit zwischen den Bildwiederholungen berücksichtigt werden. Hierfür steht in Unity bequem eine statische Instanz der `Time`-Klasse zur Verfügung. Deren Eigenschaft `deltaTime` gibt die Zeit in Sekunden an, welche für die letzte Bildwiederholung benötigt wurde.

```
- function Update () {
    this.transform.position.x += fSpeed * Time.deltaTime;
}
```

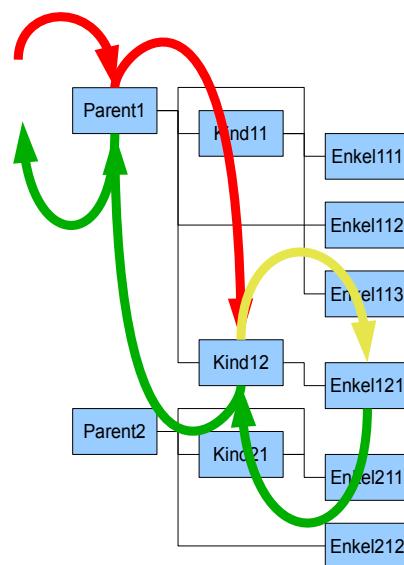
Die Methode  `FixedUpdate` wird vordringlich im Zusammenhang mit der internen Physikengine genutzt. Sollen beispielsweise skriptgesteuert Kräfte auf Körper wirken, so ist es wichtig, dass diese möglichst in konstanten Zeitintervallen berücksichtigt werden. Standardmäßig liegt dieses Zeitintervall bei 0.02 Sekunden.

# 13 Spielereingaben

Interaktion ist ein Kernelement des Spiels, der Mensch muss in der Spielwelt wirksam werden können. Hierzu stehen Eingabemöglichkeiten zur Verfügung, für PC- und Browserspiele derzeit vordringlich Tastatur und Maus. Dabei gibt es zwei prinzipielle Herangehensweisen um die Eingaben des Spielers zu registrieren. Beim Polling werden in kurzen Zeitabständen, z.B. bei Bildwiederholung, die Zustände der Eingabeschnittstellen abgefragt und die Daten abgeholt. Geht man dagegen ereignisgesteuert vor, werden automatisch bei Spielerinteraktion bestimmte Methoden aufgerufen, welche die Verarbeitung der Eingabedaten vornehmen oder weiter delegieren. Die beiden Herangehensweisen schließen sich nicht gegenseitig aus und werden häufig kombiniert.

## 13.1 Keyboard- / MouseEvents in Flash

Die Koordinaten des Mauszeigers werden für jedes DisplayObject auf der Bühne als die Eigenschaften `mouseX` und `mouseY` automatisch mitgeführt und liegen somit in direktem Zugriff. Die Koordinaten beziehen sich dabei auf das lokale Koordinatensystem des Objekts. Sollen sie auf Bühnenkoordinaten umgerechnet werden, oder von dort wieder auf das Koordinatensystem eines anderen Objektes, erweisen sich die Methoden `localToGlobal(...)` und `globalToLocal(...)` als praktikabel.



Eventphasen: rot=Capture,  
gelb=Target, grün=Bubble

Werden die Mausbuttons oder die Tastatur bedient, so wandern wieder Events durch das System. Ebenso werden Events „gefeuert“ wenn der Mauszeiger ein DisplayObject berührt oder diese Berührung gerade beginnt oder endet. Die Event-Objekte sind Instanzen der Klassen `MouseEvent` oder `KeyboardEvent` und tragen eine Fülle an Informationen in ihren Attributen.

Die `MouseEvent`-Klasse definiert eine ganze Reihe von Event-Typen wie `.CLICK`, `.MOUSE_OVER`, `.RIGHT_MOUSE_DOWN` etc. Neben den lokalen und globalen Mauszeigerkoordinaten trägt ein verschicktes `MouseEvent`-Objekt zusätzlich Informationen, z.B. zum Status der Alt, Strg und Shift-Tasten der Tastatur. Diese letztgenannte Information tragen auch die Objekte der `KeyboardEvent`-Klasse, sie definiert aber nur die zwei Event-Typen `.KEY_UP` und `.KEY_DOWN`. Zentral wichtige Information ist der `KeyCode`, der die Nummer einer gedrückten oder losgelassenen Taste wiedergibt. Die Klasse `Keyboard` hält viele Konstanten wie `.LEFT`, `.UP`, `.F1` oder `.ENTER` zum Vergleich des `KeyCodes` bereit.

Bei diesen Events kommt nun das von Javascript bekannte Konzept der Event-Phasen zum Einsatz. Ein `MouseEvent` auf einem `DisplayObject` (in der Grafik `Enkel121`), oder ein `KeyboardEvent` auf einem Objekt, welches den Fokus hat, steigt wie gewohnt in der Displaylist-Hierarchie auf. Es „bubbled“. Zuvor allerdings wird es durch die Display-Hierarchie von der Stage aus nach unten bis zu dem Objekt durchgereicht. Das ist die sogenannte Capture-Phase. Auf dem Objekt selbst befindet sich das Event in der Target-Phase. Erst dann kommt es in die Bubble-Phase und steigt wieder auf. Somit können Objekte, die in der Hierarchie weiter oben stehen,

ein Event schon in der Capture-Phase abfangen, bevor das eigentliche Target das Event bearbeiten kann. Dies ist zum Beispiel sehr nützlich, wenn ein Ereignis für eine Gruppe von Objekten relevant ist und überprüft werden muss, bevor es weitergegeben wird. Das Weiterreichen des Events kann mit `stopPropagation()` unterbunden werden. Soll ein Listener ein Event bereits in der Capture-Phase abfangen, so muss als dritter Parameter der `addEventLister`-Methode der Wert `true` übergeben werden. Die Eigenschaft `target` eines Events verweist auf das Objekt, an dem das Ereignis eingetreten ist (im Beispiel `Enkel121`), `currentTarget` dagegen verweist auf das Objekt, dessen Listener das Event aufgefangen hat.

## 13.2 Input-Klasse von Unity

Spielereingaben werden in Unity durch die Klasse `Input` für das Polling zur Verfügung gestellt.

Im einfachsten Fall kann man beispielsweise die Eigenschaft `.mousePosition` auslesen, sie ist vom Typ `Vector3` und enthält die Mauszeigerkoordinaten in Pixel. Mit den Klassenmethoden `GetMouseButton(...)`, `GetMouseButtonDown(...)` und `GetMouseButtonUp(...)` werden die Tasten der Maus abgefragt, wobei als Parameter die Nummer der Taste übergeben wird.

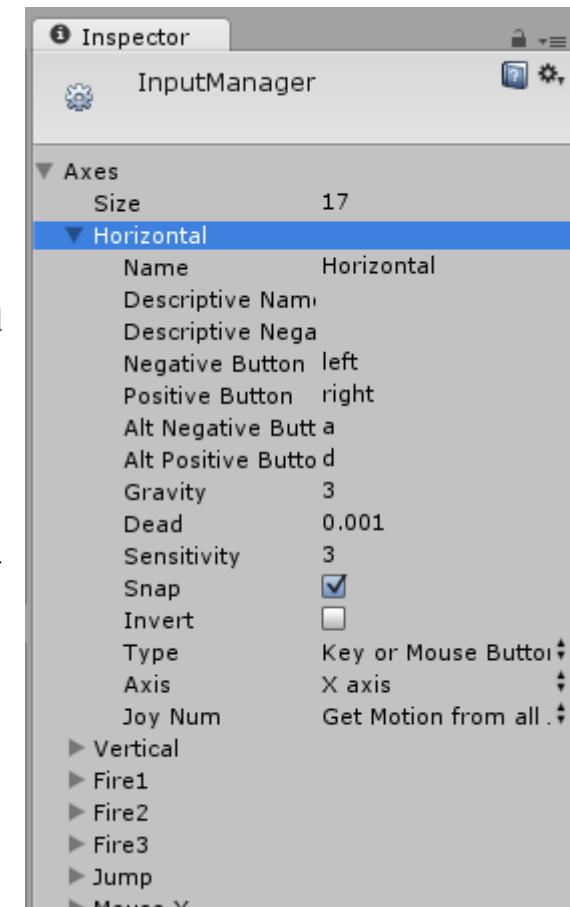
Analog stehen für die Tastaturabfrage die Klassenmethoden `GetKey(...)`, `GetKeyDown(...)` und `GetKeyUp(...)` zur Verfügung. Parameter kann das Tastaturzeichen als String sein, zum Beispiel "a". Besonders praktikabel ist hier -wie in Flash- die `KeyCode`-Klasse, welche Konstanten wie `.UpArrow` für die Tasten bereit hält.

Unity bildet zudem die Spielereingaben auf sogenannte „Achsen“ ab, welche in der IDE aber auch bei Bedarf zur Laufzeit definiert und parametriert werden können. Die horizontale Input-Achse gibt beispielsweise an, ob der Spieler die Pfeiltasten links oder rechts, die Tasten a oder d oder den Joystick zur Seite gedrückt hält. Statt all diese Optionen einzeln abzufragen genügt es, den Rückgabewert der Methode `Input.GetAxis("Horizontal")` zu verwenden. Dieser wird zudem gedämpft, so dass er nicht zwischen -1, 0 und 1 hin und her springt, sondern zu diesen Werten sanft hingeführt wird. Will man dies nicht, kann man auch `GetAxisRaw(...)` verwenden. In der IDE können die Achsen mit Hilfe des Input-Managers konfiguriert werden, der unter `Edit → Project Settings → Input` zu finden ist.

```
- function OnMouseDown() {
    Destroy(this.gameObject);
}
```

Dieser Code zerstört ein  
GameObject beim Anklicken.

Die Ereignissesteuerung ist bei Unity dagegen wieder durch automatische Aufrufe bestimmter Methoden realisiert. Diese beziehen sich allerdings ausschließlich auf Aktivitäten der Maus. Wenn der Cursor ein GameObject z.B. berührt oder verlässt oder dabei die Maustaste gedrückt wird, werden in den angehängten Skripten des Objektes Methoden wie `OnMouseEnter()`, `OnMouseDrag()` und `OnMouseDown()` aufgerufen.



## 14 Kollisionen

Neben der Interaktion mit einem oder mehreren Spielern ist es in Computerspielen häufig auch erforderlich, Interaktionen der Spielemente untereinander abzubilden. Dabei spielt die räumliche Nähe oft die tragende Rolle. Bezogen auf die Ursprünge der lokalen Koordinatensysteme lassen sich die Distanzen leicht bestimmen (siehe Kapitel "Mathematische Grundlagen"). Durch periodische Prüfung der Distanzen bewegter Objekte lässt sich so feststellen, wenn eine festgelegte Reaktionsdistanz unterschritten wird und die erforderliche Interaktion starten.

Unity und auch Flash bieten zusätzliche Möglichkeiten um festzustellen, ob ein Objekt einen bestimmten Punkt überdeckt oder eine Linie im Raum kreuzt, oder sich mit einem anderen Objekt überlappt. Im letzten Fall wird zwischen einer groben und schnellen Überprüfung einer möglichen Überlappung, und einer tatsächlichen Überlappung der Umrisse der Objekte unterschieden, welche deutlich anspruchsvoller ist.

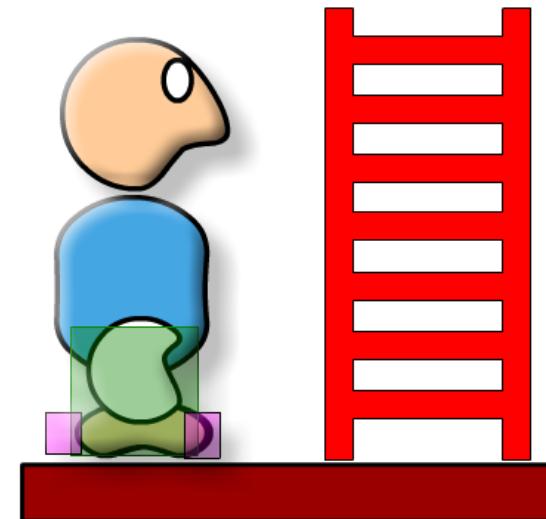
### 14.1 hitTest in Flash

Die kantenscharfe Überlappung ist in Flash nicht ohne weiteres zu überprüfen. Um zu prüfen, ob sich die Bounding Boxes, also die umhüllenden Rechtecke zweier DisplayObjects überschneiden, steht die einfache Methode `Kind11.hitTestObject(Kind12)` zur Verfügung. Die Methode liefert den bool'schen Wert `true`, wenn sich die Begrenzungsrahmen (welche auch die Enkel usw. umfassen) überlappen.

`Kind11.hitTestPoint(x, y, shapeflag)` prüft dagegen, ob der Punkt mit den Bühnenkoordinaten x,y innerhalb des Begrenzungsrahmens von Kind11 liegt. Wird als dritter Parameter `true` mitgegeben, so wird getestet, ob an dieser Stelle auch tatsächlich ein Pixel von Kind11 gezeichnet wird, oder das DisplayObject dort transparent ist.

Es gibt noch eine weitere `hitTest`-Methode mit deren Hilfe die Überlappung der tatsächlichen Konturen zweier Objekte möglich ist. Hierzu muss allerdings auf den Zeichenbereich, also das Bitmap- bzw. das `BitmapData`-Objekt, in welchen das Objekt zunächst gerendert wird, zugegriffen werden. Außerdem müssen die Transformationen der Objekte zusätzlich explizit berücksichtigt werden.

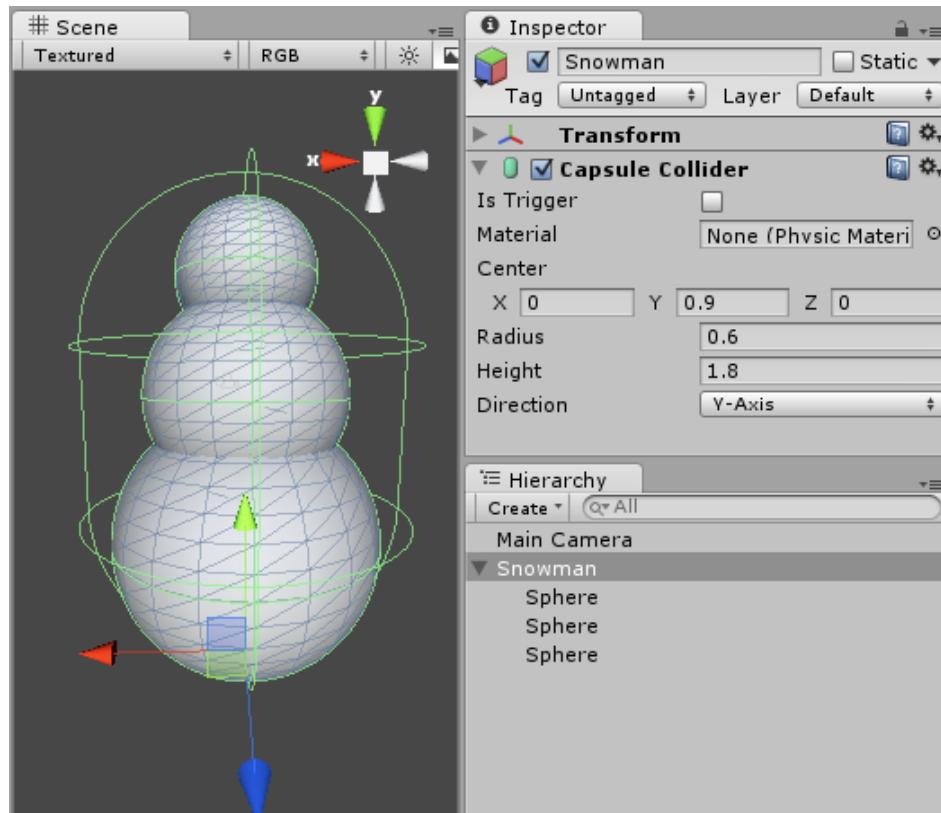
In vielen Fällen genügt es aber, mit der einfachen `hitTestObject`-Methode zu arbeiten. Dabei kann es hilfreich sein, nicht zwingend die Umrisse eines Objektes für die Kollisionsprüfung heranzuziehen, sondern zu diesem Zweck dem Objekt spezielle Kollisionsflächen heranzuziehen. Die Figur im Bild ist mit einem großen Collider (grün) ausgestattet, mit dessen Hilfe verhindert wird, dass er in den Boden einsinkt. Eine Leiter kann die Figur erst besteigen, wenn beide pinken Collider gleichzeitig mit der Leiter kollidieren. Alle Collider werden im Konstruktor der Figur ausgeblendet und sind somit natürlich im Spiel unsichtbar.



## 14.2 Collider in Unity

In drei Dimensionen genügt es nicht, die Begrenzungsrahmen zweier Objekte zu vergleichen, man benötigt Begrenzungsräume. In Unity stehen dafür zunächst drei Formen zur Verfügung. Die Box, die Kugel und die Kapsel (ein Zylinder mit einer Halbkugel an jeder Stirnfläche). Hierzu sind von der Collider-Klasse die entsprechenden Subklassen `BoxCollider`, `SphereCollider` und `CapsuleCollider` abgeleitet.

Zur Laufzeit prüft Unity selbstständig Überlappungen der Collider in der Szene. Wird eine Kollision erkannt, so werden bei beiden Kollisionspartnern je nach Ereignis die Methoden `OnCollisionEnter(...)`, `OnCollisionExit(...)` und `OnCollisionStay(...)` aufgerufen. Als Parameter wird dabei eine Instanz der Klasse `Collision` übergeben, deren Attribute Informationen über die Kollision beinhalten, darunter einen Verweis auf das `GameObject`, mit dessen Collider kollidiert wurde.



Diese Collider lassen sich bezogen auf die Transform-Komponente parametrieren und positionieren, und sind unabhängig von den dazugehörigen Meshes oder der untergeordneten Hierarchie. Der Schneemann im Bild besteht aus einem leeren `GameObject`, dem drei Kugeln untergeordnet sind. Der `CapsuleCollider` ist als Komponente dem leeren Objekt zugeordnet. Er ist auf einen Radius von 0.6 und eine Höhe von 1.8 parametriert und kann damit den Schneemann geräumig umschließen. In Bezug auf das lokale Koordinatensystem des Schneemanns, welches seinen Ursprung am Boden der unteren Kugel hat, musste er dazu um 0.9 nach oben verschoben werden.

Die Eigenschaft "IsTrigger" eines Colliders bewirkt, dass der Kollider nicht in die Berechnung physikalischer Wirkung von Kollisionen einbezogen wird. Er dient dann ausschließlich der Prüfung der Kollision, worauf die Methoden `OnTriggerEnter(...)`, `OnTriggerExit(...)` und `OnTriggerStay(...)` aufgerufen werden. Für die Steuerung der Spiellogik ohne relevante Physiksimulation sind diese Trigger das Mittel der Wahl. Ein großer `SphereCollider` als Trigger kann somit sehr bequem herangezogen werden um festzustellen, ob sich ein Objekt einer Stelle im Spiel auf den Triggerradius nähert. So ergeben auch Collider Sinn, die an ein leeres `GameObject` ganz ohne sichtbare Teil gekoppelt sind.

Die Berechnungsanforderung bei der Kollisionserkennung steigt mit steigender Anzahl von zu überprüfenden Objekten in der Szene exponentiell. Daher ist es erforderlich, näher zu bestimmen, welche Objekte mit welchen anderen kollidieren können. Da die Kollisionserkennung von der

eingebauten Physikengine übernommen wird, kommen dabei weitere Aspekte der Physiksimulation zum Tragen, auch wenn eine solche Simulation gar nicht gewünscht ist. So wird eine Kollision nur erkannt, wenn wenigstens einer der beiden Kollisionspartner eine Rigidbody-Komponente hat. Diese Komponente gibt dem Objekt eine Masse und unterwirft es physikalischen Kräften. Ist letzteres nicht erforderlich, so kann und sollte die Eigenschaft "IsKinematic" aktiviert werden. So lässt sich das Objekt wie gewohnt geometrisch transformieren. Die folgende Tabelle zeigt, bei welchen Kombinationen von Kollisionspartnern die Kollisionen erkannt werden.

Collision detection occurs and messages are sent upon collision						
	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider			Y			
Rigidbody Collider	Y		Y			
Kinematic Rigidbody Collider			Y			
Static Trigger Collider						
Rigidbody Trigger Collider						
Kinematic Rigidbody Trigger Collider						

Trigger messages are sent upon collision						
	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider					Y	Y
Rigidbody Collider				Y	Y	Y
Kinematic Rigidbody Collider				Y	Y	Y
Static Trigger Collider		Y	Y		Y	Y
Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y
Kinematic Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y

Layer Collision Matrix						
	Default	TransparentFX	Ignore Raycast	Water	Layer1	Layer2
Default	<input checked="" type="checkbox"/>					
TransparentFX	<input checked="" type="checkbox"/>					
Ignore Raycast	<input checked="" type="checkbox"/>					
Water	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Layer1	<input type="checkbox"/>	<input checked="" type="checkbox"/>				
Layer2	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Layer3	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Um die Performance bezüglich der Kollisionsbehandlung weiter zu optimieren kann das Layer-System von Unity herangezogen werden (Abbildung rechts). Per Dropdown-Box oben im Inspector kann jedes GameObject einem Layer zugeordnet werden. Unter Edit → ProjectSettings → Physics kann dann für jeden Layer gegenüber jedem anderen Layer eingestellt werden, ob deren Inhalte gegeneinander auf Kollision geprüft werden sollen.

Für eine kantenscharfe Objektkollision steht die Komponente MeshCollider zur Verfügung, welche natürlich gegenüber den einfachen Collidern einen höheren Rechenaufwand mit sich bringt.

Der CharacterController ist eine Collider-Komponente mit erweiterter Funktionalität. Mit den Befehlen SimpleMove(...) und Move(...) kann eine Spielfigur leicht durch eine Szene bewegt werden, wobei sie in gewissen Grenzen der physikalischen Gesetzen gehorcht, über Kanten bestimmter Höhe laufen und Steigungen bis zu definierbarer Größen überwinden kann. Die Figur kann oder sollte dann allerdings nicht mehr mit den üblichen Transformationen bewegt werden.

WheelCollider werden den Rädern eines Fahrzeugs angefügt und ermöglichen auf einfache Weise dessen Fahrverhalten abzubilden.

## 14.3 Raycasts

Häufig ist es nicht erforderlich oder gewünscht, eine komplexe Kollisionserkennung durchzuführen. Oft genügt es zu ermitteln, ob ein Objekt von einer Linie durch den Raum geschnitten wird. So kann beispielsweise geprüft werden, ob eine Spielfigur sich im Sichtfeld einer anderen befindet, oder von einer Wand verdeckt wird. Man kann sich vorstellen, dass man hierzu einen geraden Lichtstrahl (ray) in die Szene wirft (cast) und daran erkennt, welches Objekt er trifft.

Unity stellt hierfür die polymorphe Methode `Physics.Raycast(...)` zur Verfügung. Allen Ausprägungen der Methode gemein ist, dass Daten, welche den Strahl mit seinem Startpunkt in der Szene und seiner Richtung definieren, übergeben werden. Weiterhin wird eine Distanz angegeben, bis zu der geprüft werden soll, da es in der Praxis häufig nicht erforderlich ist, die ganze Szene zu testen. Außerdem können bestimmte Layer von der Prüfung ausgenommen werden.

Trifft der Strahl ein Objekt, werden eine ganze Menge an Daten berechnet. Neben einer Referenz auf den getroffenen Collider, und der Länge des Strahls bis zum Trefferpunkt, werden auch die Koordinaten des Trefferpunktes in der Welt, der Normalenvektor auf dem Objekt, die Texturkoordinaten des Trefferpunktes und weiteres geliefert. Um all diese Daten von der Methode zu erhalten, übergibt man eine Referenz auf ein leeres Datenobjekt (Struct) vom Typ `RaycastHit`, dessen Attribute beim Treffer gefüllt werden. Der eigentliche Rückgabewert der Methode ist schlicht vom Typ `boolean` und besagt, ob der Strahl etwas getroffen hat oder nicht.

```
var hit : RaycastHit;
if (Physics.Raycast(transform.position, -Vector3.up, hit, 100.0)) {
    var distanceToGround = hit.distance;
}
```

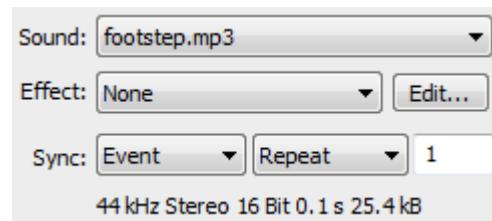
## 15 Akustik

Leider verfügen weder Flash noch Unity über Möglichkeiten, Audiodateien zu erstellen oder zu bearbeiten. Hier ist man gezwungen auf vorgefertigtes Material und/oder entsprechende externe Audioeditoren zurück zu greifen. Bei der Aufbereitung sollten die Sounds in der Regel keine Pausen am Anfang oder am Ende aufweisen, so dass die Synchronisation mit Spielereignissen oder das Loopen möglich ist. In der akustischen Wahrnehmung fallen auch kleine Fehler und Verzögerungen sehr stark auf und gefährden das Spielerlebnis.

### 15.1 Flash-Sounds

Es gibt mehrere Möglichkeiten Sound in Flash einzubinden und zu steuern. Sie können in die Library importiert oder zur Laufzeit von externen Quellen geladen werden, und direkt in die Timeline eingebunden oder per Skript angesteuert werden. Flash ist in der Lage, Sounds in der Library bei der Veröffentlichung mit vielfältig einstellbaren Parametern zu komprimieren, um die Größe der erzeugten SWF-Datei gering zu halten. Nebenstehend ein Ausschnitt aus dem Eigenschaften-Dialog einer importierten WAV-Datei.

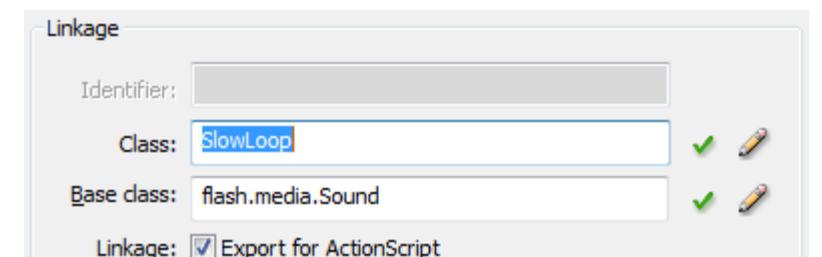
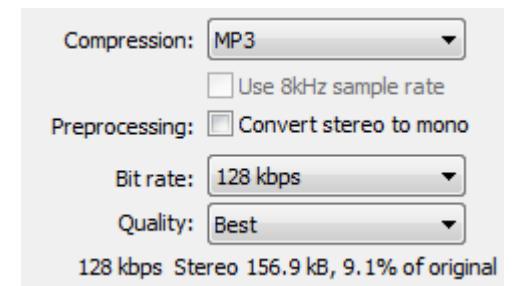
Importierte Sounds können an einen Keyframe der Zeitleiste gekoppelt werden (Frame anwählen und Sound auf die Stage ziehen), in den nachfolgenden Frames wird dann der Signalverlauf angezeigt. Im Eigenschaften-Dialog des Frames sind wieder diverse Einstellungen möglich.



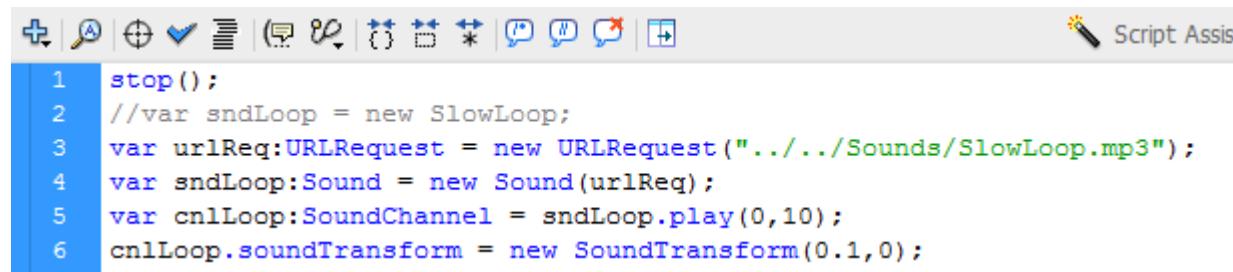
Mit Effect lässt sich ein Panning und ein Ein- und Ausblenden einstellen, auch eine einfache, selbstgestaltete Hüllkurve lässt sich einrichten. Das Einstellen der Lautstärke ist damit recht umständlich. Bei Sync wird gewählt, was bei Erreichen des Keyframes passieren soll, wobei Stream eine besondere Bedeutung hat. Hier übernimmt der Sound die Zeitleistensteuerung, womit z.B. lippensynchrone Animationen erstellt werden können. Bei Repeat kann die Anzahl der gewünschten Wiederholungen oder einfach Loop eingestellt werden.

Sofern ein importierter Sound für ActionScript erreichbar gemacht ist, kann er auch, wie andere Klassen, einfach mit „new“ per Skript instanziert werden. Er ist eine Subklasse der Klasse „Sound“. Mit play (...) lässt er sich bereits abspielen.

Für die Steuerung der individuellen Lautstärke des Sounds und weiterer Effekte sind dann aber noch andere Klassen erforderlich. Play(...) liefert ein SoundChannel-Objekt zurück, dessen soundTransform-Eigenschaft ein SoundTransform-Objekt zugewiesen werden muss, welches die Information über die Lautstärke enthält.



Genauso wird auch bei Sounds verfahren, die zur Laufzeit von einer externen Ressource geladen werden. Geladen wird mit der Sound-Klasse und einem URLRequest, welcher den Pfad zu der Sound-Datei enthält. Der nebenstehende Code lädt einen externen Sound, und spielt ihn zehnmal mit einer Lautstärke von 10% ab. Die oben beschriebene Instanzierung aus der Library ist auskommentiert.



```

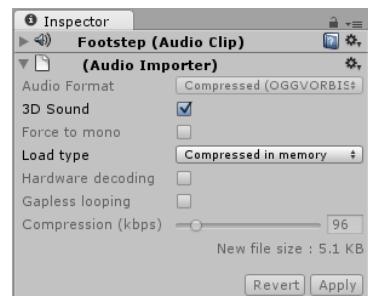
1 stop();
2 //var sndLoop = new SlowLoop;
3 var urlReq:URLRequest = new URLRequest("../Sounds/SlowLoop.mp3");
4 var sndLoop:Sound = new Sound(urlReq);
5 var cnlLoop:SoundChannel = sndLoop.play(0,10);
6 cnlLoop.soundTransform = new SoundTransform(0.1,0);

```

Gegebenenfalls muss beim Laden im Netz sichergestellt werden, dass der Ladevorgang abgeschlossen ist, bevor abgespielt werden soll. Hierzu muss das COMPLETE-Event abgefangen werden.

## 15.2 Audio-Sources und -Listener

So wie die Kamera im dreidimensionalen Raum das sichtbare Bild bestimmt, wird in Unity ein AudioListener-Objekt wie ein Mikrophon genutzt. Dessen relative Position zu in der Szene verteilten AudioSource-Objekten bestimmt, was schließlich für den Spieler hörbar wird. Standardmäßig ist ein solcher Listener an jede Kamera geheftet, es sollte also darauf geachtet werden, dass immer nur ein Listener aktiv ist.

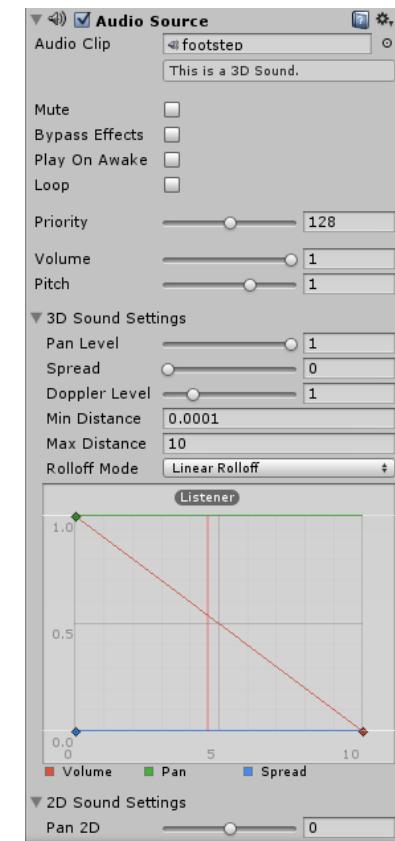


Auch Sounds sind in Unity wieder ausschließlich externe Ressourcen, welche in das Projekt eingebunden werden, z.B. per Drag&Drop aus dem Windows-Explorer. Einem solchen Sound wird ein AudioImporter zugeordnet, mit dessen Hilfe sich wiederum diverse Kompressionsparameter einstellen lassen. Wird „3D-Sound“ deaktiviert, kann der Sound positionsunabhängig abgespielt werden, er ist dann ein 2D-Sound.

Mit der Komponente „Audio Source“ wird ein Audio-Clip an ein GameObject gekoppelt. Diese Komponente kann wieder vielfältig parametriert werden (siehe rechts).

Erklärungsbedürftig ist hierbei vordringlich die Attenuation bei 3D-Sounds. Auf grafische Weise kann hier die Lautstärkedämpfung über die Entfernung des Listeners zur Quelle definiert werden. Min- und Max Distance geben hierfür den Wirkungsbereich vor. Ebenso können auch die Eigenschaften „Pan“ (Mischung von lokalisierbarem 3D- und nicht lokalisierbarem 2D-Anteil) und „Spread“ (die Trennschärfe für Surround-Systeme) eingerichtet werden. Die Bedienung des grafischen Editors ist von der Zeitleiste bekannt, insbesondere auch bezüglich der Nutzung des Mausrades.

-Komponenten lassen sich auch per Skript steuern. Dazu kommt die Möglichkeit, über die Klassen-Methode `PlayClipAtPoint(...)` einen Soundeffekt kurzfristig in die Szene zu setzen.

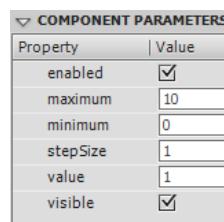


# 16 Virtuelles User Interface (GUI)

In vielen Spielen steht dem Spieler neben oder auf dem Blick auf das Spielszenario ein virtuelles Userinterface, also eine virtuelle Schnittstelle zur Spielwelt zur Verfügung. Hiermit erhält er Informationen über den Zustand des Spiels und ausgewählter Elemente. Ebenso werden häufig zusätzliche Interaktionsmöglichkeiten in diesem Interface angeboten. Die zugehörigen Elemente können dabei fest auf dem Bildschirm verortet sein oder an Positionen von Spielobjekten auftauchen. Dabei bildet die Schnittstelle in der Regel die oberste Ebene der Sichtbarkeit. Obwohl die Darstellung der Spielwelt bereits ein grafisches Userinterface ist, wird dieser Begriff und seine Abkürzung GUI meist auf die virtuelle Schnittstelle bezogen. Häufig wird auch der Begriff HUD (Head-Up-Display) verwendet, der ursprünglich aber lediglich ein System beschreibt, welches Informationen halbtransparent über die Szenerie projiziert und keine Interaktion beinhaltet. In der Folge wird nur der Begriff Userinterface (UI) genutzt.

## 16.1 Flash Components

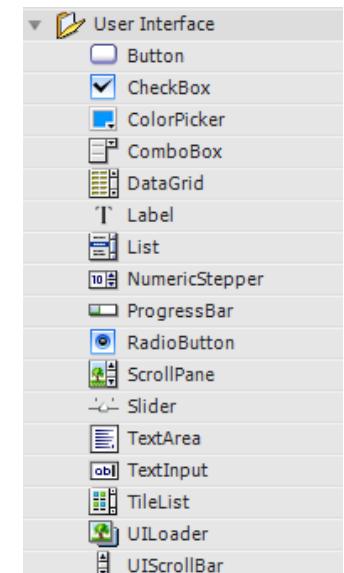
Ein solches Interface in Flash zu erstellen ist trivial, da Flash ohnehin eine 2D-Entwicklungsumgebung ist. Lediglich die Ebenenreihenfolge ist zu beachten, damit die Schnittstellen-Elemente vor den Spielelementen liegen. Für viele Anwendungen und Prototypen ist es sinnvoll, die integrierten Flash-Components zu nutzen. Sie bieten eine einfache und schnelle Möglichkeit, ein Userinterface nach gängigen Standards zu erstellen. Diese Komponenten umfassen z.B. Buttons, Checkboxen, Radiobuttons, Sliders, Scrollpanes, Comboboxes etc. Dabei ist die komplette Maus- und Tastaturbedienung der Komponenten bereits fertig implementiert.



Die Komponenten können direkt aus dem Auswahlfenster mit der Maus auf die Bühne gezogen werden. In Lage und Größe sind sie dann veränderbar, wobei sie sinnvoll skaliert werden. Außerdem erscheint am Eigenschaftenfenster der Komponente ein Dialogfeld für die Parametrierung der Komponente. Hier können die wichtigsten Eigenschaften zur Steuerung direkt eingegeben werden. Links ist dieser Dialog für einen NumericStepper (Spin-Control) zu sehen. Die Parameter lassen sich auch per Skript in der Regel mit dem gleichen Namen ansprechen und verändern.

Bei Bedienung der Komponente zur Laufzeit werden Events verschickt, auf die das Programm reagieren kann. Die wichtigsten sind Event.CHANGE oder ComponentEvent.LABEL\_CHANGE. In vielen Fällen genügt es aber bereits die Werte der Eigenschaften value, selected, selectedIndex oder selectedItem abzufragen. Informationen zu speziellen Events der jeweiligen Komponente sind in der Referenz nachzulesen.

Die Komponenten können in Ihrem Erscheinungsbild in gewissem Umfang verändert werden. Prinzipiell kann man nach dem Einfügen der Komponente durch Doppelklick darauf die Einzelteile bearbeiten. Farbänderungen sind dabei recht leicht machbar. Aber schon die Änderung der Schriftfarbe ist so nicht mehr zu bewerkstelligen.

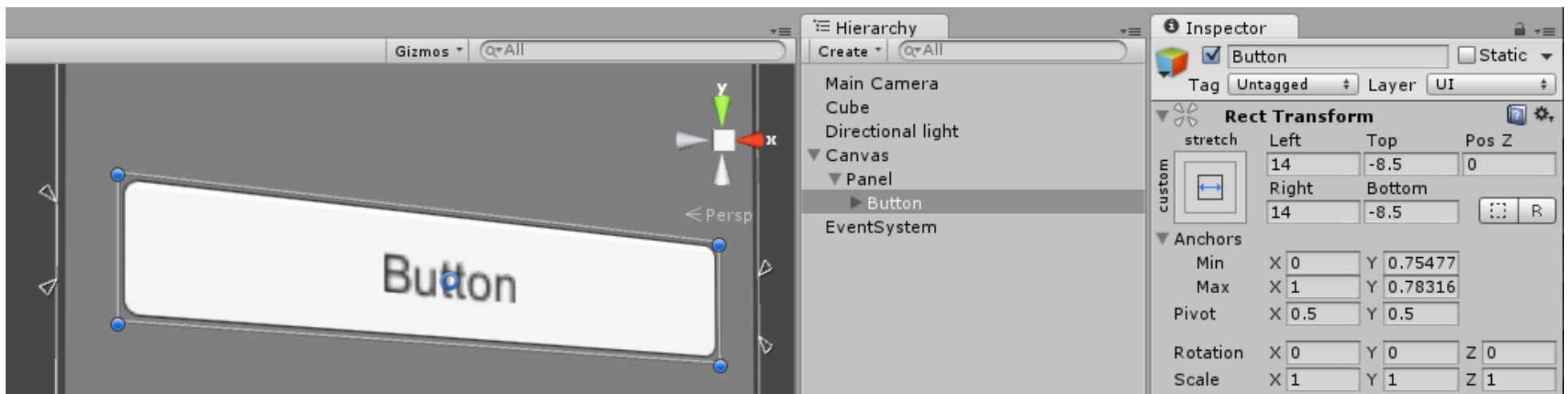


## 16.2 Unity UI-Elements

Zur Darstellung eines Userinterfaces wird im Szenenbaum ein spezieller Ast angelegt, dessen Wurzelement vom Typ Canvas ist. Alle UI-Elemente müssen diesem untergeordnet werden um sichtbar zu sein und zu funktionieren. Ein Canvas wird ggf. automatisch erzeugt, wenn noch nicht vorhanden. Bei der Erzeugung des Canvas wird auch eine Instanz der Klasse EventSystem auf der obersten Ebene des Szenenbaums abgelegt, welche die Nutzereingaben verwaltet.

### 16.2.1 RectTransform

Wie GameObjects immer über eine Transform-Komponente verfügen, so ist jedem UI-Element und dem Canvas eine RectTransform-Komponente zugeordnet. RectTransform ist eine Subklasse von Transform und erweitert diese um Informationen, wie sich das betreffende Element in Bezug auf das Parent-Element verhalten soll, wenn letzteres in seiner Größe verändert wird. Im Kern geschieht das durch die Positionierung der Ankerpunkte in Bezug auf Länge und Breite des Elterelements. Jeder Eckpunkt des Kindelementes hält einen konstanten Abstand zu seinem korrespondierenden



Der Button wird immer den horizontalen Abstand zum Panel halten und vertikal nur im mittleren Drittel skalieren

Ankerpunkt. Ebenso gibt es einen Pivotpunkt der dem Transformationspunkt in Flash entspricht. Zu beachten ist, dass Screen-Koordinaten nicht von der linken oberen Ecke des Fensters ausgehen, sondern von unten.

### 16.2.2 Canvas

Die Integration in den Szenenbaum erlaubt es, auch das UI in die 3D-Szene zu integrieren und bei Bedarf innerhalb der Spielwelt erscheinen zu lassen.

Hierzu stellt die Canvas-Komponente des Canvas-Elementes die Optionen „Screen Space - Camera“, bei welcher das UI immer in einer bestimmten Distanz vor der Kamera erscheint, und „World Space“ zur Verfügung. In der Folge wird nur die Einstellung „Screen Space - Overlay“ berücksichtigt, bei welcher das UI nachträglich auf die fertig gerenderte Szene gezeichnet wird.

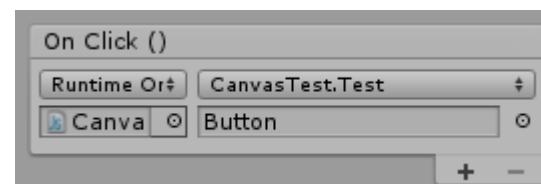
Der Canvas erscheint auch im Editor in der Szene. Insbesondere bei einem ScreenSpace-Canvas ist das überraschend, aber für die Gestaltung praktisch. Allerdings ist er so skaliert, dass ein Pixel des UserInterfaces einer Unity-Einheit entspricht. Bei der für die Physik-Engine sinnvolle Zuordnung von einem Meter pro Unity-Einheit, kann das UI durchaus eine Fläche von einem Quadratkilometer oder mehr in der Welt einnehmen. Wenn es stört, kann die Editor-Ansicht mit Layers→UI aus- oder eingeschaltet werden.

### 16.2.3 CanvasScaler

Der RectTransform eines ScreenSpace-Canvas kann nicht manuell eingestellt werden. Seine Größe ist immer abhängig von den tatsächlichen Dimensionen des Ausgabefensters und der grafischen Auflösung des Ausgabegerätes. Den Zusammenhang definiert die Komponente CanvasScaler des Canvas. Entsprechend des dort eingestellten Skalierungsmodus sind noch weitere Parametrierungen möglich.

### 16.2.4 Interaktion

UI-Elements reagieren bereits mit grafischer Rückmeldung auf Nutzerinteraktion. Welche Wirkung sie darüber hinaus entfalten sollen, muss aber programmiert werden. Hierfür gibt es drei grundlegende Ansätze.



Die Komponente des UI-Elements zeigt im Inspector ein Feld, welches mit dem Interaktionsoption beschriftet ist. Beim Button ist das „On Click“. Dort muss zunächst ein Zielobjekt eingetragen werden, dann kann eine Funktion des Objektes als Handler für das Ereignis eingetragen werden. Ist die Funktion mit einem Parameter definiert, kann auch dieser eingetragen werden. Im Beispiel wird bei Klick die Funktion Test des Skriptes CanvasTest auf dem Objekt Canvas aufgerufen und der Button selbst übergeben,

```
private var b:UnityEngine.UI.Button;

function Start() {
    b = this.GetComponent(UnityEngine.UI.Button);
    b.onClick.AddListener(ButtonTest);
}

function ButtonTest() {
    Debug.Log("ButtonHallo");
}
```

Diese Methode erscheint zunächst relativ einfach, wird aber schnell umständlich und unübersichtlich. Den im Inspektor dargestellten Interaktionsmöglichkeiten können Handler aber auch per Skript zugeordnet werden. Die UI-Komponente sammelt hierzu Listener in Variablen z.B. mit dem Namen onClick.

Um alle Möglichkeiten nutzen zu können, welche die UI-Elemente bieten empfiehlt es sich aber, ein Skript zu schreiben, welches explizit die Klasse UIBevaviour erweitert und Interfaces wie IPointerClickHandler oder IPointerEnterHandler implementiert. Damit werden dann die zugehörigen Funktionen wie OnPointerClick oder OnPointerHandler automatisch aufgerufen und zudem ein EventData-Objekt übergeben, welches Details zur Interaktion enthält.

## 17 Externe Daten

Die Bibliotheken von Flash und Unity werden bei der Veröffentlichung gepackt und in speziellen Formaten gespeichert. Häufig ist es aber wünschenswert oder erforderlich, dass zur Laufzeit der Applikation weitere Daten importiert und verarbeitet werden können, die nicht zuvor mit kompiliert wurden. Sofern dies vorgesehen ist, lässt sich z.B. das Verhalten oder das Aussehen der Applikation steuern, ohne diese in der IDE öffnen, verändern und neu kompilieren zu müssen. Dabei ist es allerdings wichtig zu beachten, dass die Ladeprozesse asynchron laufen. Das Programm wird also nicht automatisch gestoppt, bis die Daten verfügbar sind. Um festzustellen, ob die externen Daten geladen wurden, bieten Unity und Flash unterschiedliche Möglichkeiten an.

### 17.1 Import in Flash

Relativ simpel ist das Laden externer Daten mit den Klassen „Loader“ (für Bilder und kompilierte SWF-Dateien) und „URLLoader“ (für Text und Binärdateien). Beide verfügen über die Methode `load(...)`, welcher ein Objekt der Klasse „URLRequest“ übergeben wird. Ein Objekt dieser Klasse kann neben der eigentlichen Adresse der Daten z.B. auch HTTP-Variablen verwalten. Der nebenstehende Code genügt bereits, um die Textdatei „External.txt“ zu laden, die im gleichen Pfad wie die Applikation liegt.

```
var oLoader = new URLLoader();
var oUrl:URLRequest = new URLRequest("External.txt");
oLoader.load(oUrl);
```

Die geladenen Daten liegen bei einem URLLoader-Objekt in der Eigenschaft `data` direkt im Zugriff. Da ein Loader-Objekt komplexere Datenstrukturen lädt, verweist es zu deren Beschreibung mit `contentLoaderInfo` auf ein „LoaderInfo“-Objekt, dessen Eigenschaft `content` schließlich das geladene DisplayObject enthält. Es ist aber auch möglich, das Loader-Objekt selbst zur Displaylist zu addieren.

```
oLoader.contentLoaderInfo.addEventListener(Event.COMPLETE, LoadSwfComplete);

function LoadSwfComplete(_oEvent:Event) {
    var oSwf:DisplayObject = addChild(_oEvent.target.content);
    oSwf.x = 180;
    oSwf.y = 350;
}
```

Um festzustellen, wann die Daten fertig geladen sind, hängt man dem URLLoader-Objekt bzw. dem LoaderInfo-Objekt einen Eventlistener an, der auf `Event.COMPLETE` hört. In dem dann aufgerufenen Handler kann dann entsprechend über `target.data` bzw. `target.content` auf den geladenen Inhalt zugegriffen werden.

## 17.2 Import in Unity

Für den Datenimport bietet sich die Klasse „WWW“ an. Damit ist es möglich, Texte, Binärdaten, Texturen, Videotexturen und ganze Bibliotheken (AssetBundles) zur Laufzeit aus dem Netz oder von lokalen Datenträgern zu laden. Die zu ladende Datei muss mit einem vollständigen Pfad adressiert werden, bei lokalen Speichermedien ist es zudem erforderlich den Prefix „file://“ voran zu stellen. Beim Laden aus dem Netz können zusätzliche HTTP-Parameter mitgegeben werden.

Der Ladeprozess beginnt automatisch mit der Instanzierung des Objektes, das Ende kann mit Hilfe der `yield`-Anweisung abgewartet werden. Damit wird die aktuelle Methode wie mit `return` zunächst beendet, wird aber mit der folgenden Anweisung automatisch fortgesetzt, wenn die im `yield` angegebene Methode abgeschlossen ist. Der nebenstehende Code lädt den Text der Datei „External.txt“ aus dem Ordner „ExternalData“, welcher auf der gleichen Ebene wie der Asset-Ordner erzeugt wurde, und speichert diesen in der Variablen `iniData`.

Die Klasse „Application“ erlaubt den Zugriff auf einige Informationen und Methoden der Laufzeitumgebung.

```
function Start() {
    var filename:String = "ExternalData/External.txt";
    var iniData:String = "";
    var prefix:String = "";

    if(!Application.isWebPlayer)
        ....
        prefix = "file://";
    var pathname:String = prefix + Application.dataPath + "/.." + filename;
    var www:WWW = new WWW(pathname);
    yield(www);
    iniData = www.text;
}
```

## 17.3 Text parsen

```
<data>
<level Points="123">
    Der Anfang
</level>
</data>
```

Flash verfügt über einen integrierten XML-Parser, so dass es sich anbietet, zur Laufzeit zu ladende Texte und Zahlenwerte im XML-Format abzulegen. Die geladene Textdatei kann dann einfach mit `var xmlDoc:XML = new XML(_oEvent.target.data)` in ein XML-Document umgewandelt werden, auf dessen Knoten und Attribute man mit der Punkt-Syntax und dem @-Zeichen zugreifen kann, und über welches man leicht iterieren kann. Ebenso ist aber auch eine JSON-Klasse implementiert, welche einen String in der Javascript-Object-Notation interpretieren kann.

```
{
    "level": [
        {"name": "Der Anfang",
        "points": 123
    }]
}
```

Unity verfügt mit `JsonUtility` über einen einfachen JSON-Parser, der allerdings an strikte Strukturen gebunden werden muss. Um den String `iniData` mit dem nebenstehenden Daten im JSON-Format zu laden, muss eine kongruente Klasse definiert sein. Damit kann dann die generische Funktion `FromJson` aufgerufen werden

```
var external:Data = JsonUtility.FromJson.<Data>(iniData);
```

```
class Data {
    class Level {
        ....
        var name: String;
        var points: int;
    }
    var level: Level[];
}
```



## 18 Lokale Datenspeicherung

Spielstände und Einstellungen werden häufig lokal auf dem Rechner gespeichert, an dem gespielt wird. Eine Netzwerkverbindung ist dann nicht erforderlich, allerdings sind die Informationen dann auch nur auf der entsprechenden Maschine verfügbar. Bei internetbasierten Anwendungen ist allerdings häufig ein beliebiger Zugriff auf die Datenspeicher des Rechners nicht erwünscht und unterliegt Restriktionen. Daher werden spezifische Informationen meist in Cookies gespeichert, der Nutzer muss dann selbst auch nicht den Speicherort wählen. Flash und Unity ermöglichen Ähnliches, wobei beachtet werden muss, dass der Speicherplatz limitiert ist.

### 18.1 Local Shared Object in Flash

Flash legt sogenannte Flash-Cookies tief im System unter den Benutzerdaten in einer tiefen Ordnerstruktur, ab die den Pfad zur Applikation widerspiegelt. Mit der Klasse SharedObject ist es sehr einfach diese Cookies anzulegen. Dabei unterstützt sie auch verschiedene Datentypen bis hin zu komplexen Objekten.

Im nebenstehenden Code wird mit `getLocal` ein Cookie mit Namen „TestCookie“ geladen oder, sofern er noch nicht existiert, automatisch angelegt. Das Objekt `so` der Klasse SharedObject dient in der Folge als Schnittstelle, die einzelnen Informationen lassen sich mit der `data`-Eigenschaft als Schlüssel-Werte-Paare verwalten. So wird unter „`testString`“ die Zeichenkette „Hallo“ abgelegt und der Wert 123.4 unter „`testNumber`“. Existiert bereits ein Eintrag unter „`testObject`“, so wird dessen `x`-Wert um eins erhöht, ansonsten wird ein Objekt der Klasse `Point` erzeugt und gespeichert. Ganz am Ende wird mit `flush` der Cookie abgespeichert. Mit jedem Start des Programms sieht man nun in der Konsole, wie sich der `x`-Wert des Punktes erhöht.

Hinweis: wird der Flash Media Server genutzt, können mit SharedObjects verschiedene Clients automatisch synchronisiert werden.

```
var so:SharedObject = SharedObject.getLocal("TestCookie");
so.data["testString"] = "Hallo";
so.data["testNumber"] = 123.4;

if (so.data["testObject"]) {
    so.data["testObject"].x++;
}
else {
    so.data["testObject"] = new Point(10,20);
}

trace(so.data["testObject"].x);
so.flush();
```

## 18.2 PlayerPrefs in Unity

Bei Unity können Informationen mit Hilfe der statischen Klasse PlayerPrefs gespeichert werden. Hierbei werden allerdings nur die Datentypen String, int und float unterstützt. Die Datei wird mit einem Namen versehen, welche den User und den Pfad widerspiegelt, und im Benutzerverzeichnis abgelegt.

Der nebenstehende Code versucht aus der Datei einen Integerwert unter dem Namen „testNumber“ zu finden und diesen in der gleichnamigen Variablen zu speichern. Ist die Datei noch nicht angelegt, so kommtt der im Aufruf angegebene Standardwert zurück, in diesem Fall 0. Analog wird mit der Zeichenkette testString verfahren, hier ist im Beispiel kein expliziter Standardwert angegeben. testNumber wird inkrementiert und zusammen mit testString ausgegeben. Schließlich werden die beiden Werte unter dem entsprechenden Schlüssel wieder in PlayerPrefs gesetzt und mit Save die Datei gespeichert. Beim ersten Aufruf des Programms erscheint nur die Zahl „1“ in der Konsole, beim nächsten dann „Hallo2“ und bei den folgenden Starts „Hallo3“ usw.

Achtung: Startet man in Windows die Applikation im Editor oder als Standalone, werden die Informationen in der Registry abgelegt. Hierbei werden die unter Project Settings → Player abgelegten Namen von Company und Product zur Identifikation genutzt.

```
function Start () {  
    var testNumber: int = PlayerPrefs.GetInt("testNumber", 0);  
    var testString: String = PlayerPrefs.GetString("testString");  
  
    testNumber++;  
    Debug.Log(testString + testNumber);  
  
    PlayerPrefs.SetInt("testNumber", testNumber);  
    PlayerPrefs.SetString("testString", "Hallo");  
    PlayerPrefs.Save();  
}
```

## 19 Server-Kommunikation

In Onlinespielen ist es häufig wünschenswert und meist erwartet, dass das Spiel auch Daten zum Server senden kann, um dort beispielsweise Highscore-Listen oder Spielerprofile zu pflegen. Im einfachsten Fall können solche Informationen entsprechend dem http-Protokoll mit Hilfe der GET- oder POST-Methode im Formular-Format, also als Schlüssel-Werte-Paare, verschickt werden.

Serverseitig können die Daten dann leicht entgegen genommen werden. Wird ein PHP-Skript verwendet, landet die Information in einem assoziativen Array namens `$_REQUEST`. Das folgende Skript wandelt dieses Array mit dem Befehl `json_encode` in einen JSON-String um, schreibt diesen in eine Datei mit dem Namen `processdata.txt` und liefert den String zurück als Ergebnis des http-Aufrufs.

```
<?php
    $json = json_encode($_REQUEST);
    $fileSave = fopen("processdata.txt", "w+") or die("Can't create processdata.txt");
    fwrite($fileSave, $json);
    fclose($fileSave);
    print($json);
?>
```

Natürlich muss nun auch ein Server vorhanden sein, der das Skript ausführt. Das kann ein externen Webserver sein oder beispielsweise eine XAMPP-Installation auf dem Entwicklungsrechner. Bei einem externen Webserver ist gegebenenfalls noch darauf zu achten, dass die Schreibberechtigungen für das Verzeichnis, in welches die Datei `processdata.txt` geschrieben werden soll, vergeben sind. Dies lässt sich in der Regel mit dem FTP-Programm prüfen und verändern.

Die folgenden Beispiele für die Kommunikation mit Flash und Unity referenzieren das PHP-Skript relativ und es wird davon ausgegangen, dass es im gleichen Verzeichnis auf einem Server liegt. Beim Testen in der Entwicklungsumgebung empfiehlt es sich, den absoluten Pfad anzugeben bzw. zu prüfen, ob die Anwendung im Browser läuft und entsprechend den Pfad anzupassen. Wird mit einer XAMPP-Installation getestet, sollte das Skript am einfachsten in einem Verzeichnis unterhalb von `xampp/htdocs/` liegen und mit der Url „`http://localhost/.../processdata.php`“ aufgerufen werden.

Die hier dargestellte Vorgehensweise ist natürlich nicht sicher gegen Manipulation und es wird in den Beispielen auch keine nennenswerte Fehlerbehandlung durchgeführt. Der Code hierfür würde deutlich größer und komplexer sein. Ein Einstieg in das Thema und die Möglichkeit, funktionsfähige Prototypen mit Einbindung von Serverkommunikation zu entwickeln, sollte aber hiermit gegeben sein

## 19.1 Flash

Die Klasse URLRequest hält Methoden und Eigenschaften bereit, um die Server-Kommunikation zu ermöglichen. Zur Nutzung sind noch weitere Klassen erforderlich. Die zu sendenden Daten werden in einem Objekt der Klasse URLVariables gesammelt. Beim Verschicken wird diese Information automatisch in das passende Format umgewandelt. Das Datenobjekt wird einfach der Eigenschaft data des Request-Objektes zugewiesen und die Methode mit Hilfe der Konstanten von URLRequestMethod festgelegt.

Das Request-Objekt kann dann wie gewohnt mit einem Loader genutzt werden, um den Transfer zu bewerkstelligen. Im Beispiel wird die Serverantwort in ein Textfeld mit Namen „out\_txt“ geschrieben.

## 19.2 Unity

Die zu sendenden Daten werden in einem Objekt der Klasse WWWForm gesammelt. Das Objekt wird einfach als zweiter Parameter bei der Erzeugung des WWW-Objektes übergeben, wodurch die Daten mit der POST-Methode verschickt werden. Im Beispiel wird die Antwort des Servers einer Variablen vom Typ String zugewiesen, deren Inhalt in OnGUI angezeigt wird.

Beim Testen in der Entwicklungsumgebung mit XAMPP muss zusätzlich unter Project Settings → Editor → WWW Security Emulation die Adresse „http://localhost“ eingetragen werden, um Fehlermeldung bezüglich einer Sicherheitsverletzung zu vermeiden.

```
var variables:URLVariables = new URLVariables();
variables["name"] = "Player1";

var request:URLRequest = new URLRequest("processdata.php");
request.method = URLRequestMethod.POST;
request.data = variables;

var loader:URLLoader = new URLLoader();
loader.addEventListener( Event.COMPLETE, httpRequestComplete );
loader.load( request );

function httpRequestComplete(_event:Event):void {
    out_txt.text = _event.target.data;
}
```

```
private var tAnswer:String = "...";

function Start() {
    var url:String = Application.dataPath + "processdata.php";
    var form:WWWForm = new WWWForm();
    form.AddField("name", "Player1");

    var www:WWW = new WWW(url, form);
    yield (www);
    tAnswer = www.text;
}

function OnGUI() {
    GUI.Label(Rect(10,10,200,20), tAnswer);
}
```

## 20 PeerToPeer-Kommunikation

Ein Szenario für netzwerkbasierte Spiele ist, dass die Client-Rechner direkt miteinander kommunizieren. In solchen Szenarien läuft das Spiel also nicht auf einem Server in einem Rechenzentrum der den Spielzustand an die Clients verschickt. Stattdessen wird ein allgemein erreichbarer Server nur dafür genutzt, die Verbindung zwischen den Client-Rechnern herzustellen. Ein Client kann dabei eine besondere Rolle einnehmen und als sogenannter Host auftreten. Dort laufen dann Informationen zusammen und werden wieder an die Clients verschickt. Somit kann zu einem gewissen Grade auch eine Client/Server-Architektur simuliert werden, ohne dass ein dedizierter Server hierfür erforderlich ist. Damit ist das Verfahren gerade zum Entwickeln von Multiplayer-Prototypen hilfreich.

Auf den nächsten Seiten sind zwei Skripte dargestellt, je eines für Flash und Unity, welche als Einstieg in die P2P-Kommunikation dienen. Dabei können Zeichenketten zwischen den Peers übertragen werden. Die Skripte sind vor allem darauf ausgelegt, noch auf eine Seite in diesem Vorlesungsskript zu passen und dass alle Clients, auch der „Host-Client“, mit diesem Skript sofort funktionieren können. Die verwendeten Objekte verfügen noch über weitaus mehr Eigenschaften und Methoden, es gibt noch deutlich mehr Events die man auswerten kann und unbedingt sollte, und es können neben Texten auch komplexe Datenstrukturen versendet werden. Es wird keinerlei Fehlerbehandlung durchgeführt. Firewall-Einstellungen, Verbindungsprobleme, Portbelegungen, Abbrüche durch Nutzer sind alles mögliche und häufige Fehlerquellen, die entsprechend studiert und berücksichtigt werden müssen, um ein stabiles System in inhomogenen und unbekannten Umgebungen zu entwickeln.

## 20.1 Adobe Cirrus

„Cirrus“, bzw. seit Flash Player V10.1 „Cirrus2“, heißt der Service von Adobe, mit dem ein ganzes P2P-Netzwerk aufgebaut werden kann. Hierzu ist ein Developer-Key erforderlich, der bei [labs.adobe.com/technologies/cirrus/](http://labs.adobe.com/technologies/cirrus/) zu beziehen ist. Mit Hilfe eines NetConnection-Objektes und des Keys wird eine Verbindung zum Cirrus-Server unter rtmfp://p2p.rtmfp.net hergestellt. War die Verbindung erfolgreich, so wird darauf im Beispiel eine Gruppe mit Namen „hfutestGroup“ angelegt. Dieser Name sollte möglichst eindeutig sein, da man einer Gruppe beitritt, wenn der Name schon existiert. Dies geschieht also bei weiteren Rechnern, auf denen dieser Beispielcode gestartet wird. Die Kommunikation erfolgt nun mit Hilfe des NetGroup-Objektes. Die Funktion postMessage schickt darüber Zeichenketten an die Gruppe. Er wird bei den Empfängern ausgegeben, wenn im switch NetGroup.Posting.Notify zutrifft.

```
const CirrusAddress:String = "rtmfp://p2p.rtmfp.net";
const DeveloperKey:String = "Get your key from Cirrus and enter it here";
stop();
// Objects needed for communication
var netConnection:NetConnection;
var netGroup:NetGroup;

// Open connection and listen for status changes
netConnection = new NetConnection();
netConnection.addEventListener( NetStatusEvent.NET_STATUS, handleNetStatus );
netConnection.connect(CirrusAddress, DeveloperKey);

function postMessage(_m:String) {
    netGroup.post(_m);
}

// create or join group and listen for status changes
function joinGroup() {
    var groupSpecifier:GroupSpecifier;
    groupSpecifier = new GroupSpecifier("hfutestGroup");
    groupSpecifier.serverChannelEnabled = true;
    groupSpecifier.postingEnabled = true;
    netGroup = new NetGroup(netConnection,groupSpecifier.groupspecWithoutAuthorizations());
    netGroup.addEventListener(NetStatusEvent.NET_STATUS, handleNetStatus);
}

// display status changes and react to them
function handleNetStatus(_evt:NetStatusEvent):void {
    trace("handleNetStatus: " + _evt.info.code);
    switch (_evt.info.code) {
        case "NetConnection.Connect.Success" :
            joinGroup();
            break;
        case "NetGroup.Posting.Notify" :
            trace(_evt.info.message);
            break;
    }
}
```

## 20.2 Unity Master Server

Für die Unterstützung von P2P-Netzwerken gibt es die Komponente NetworkView. Im Beispiel ist sie der Kamera zugefügt, welche auch das nebenstehende Skript erhält. NetworkView ist in der Lage automatisch Eigenschaften von GameObjects über das Netzwerk auf den Clients zu synchronisieren. Weiterhin können sogenannte „Remote Procedure Calls“ ausgeführt werden, wodurch Methoden in den Skripten der Clients aufgerufen werden. Dies wird für die Realisierung des Zeichenkettenversands im Beispiel heran gezogen.

Die statische Klasse MasterServer stellt die Verbindung zum Unity-Server her. Die Basisklasse MonoBehaviour stellt zudem eine ganze Reihe von Methoden zur Verfügung, die auf Grund von Netzwerkereignissen automatisch aufgerufen werden, und zur Verwaltung der Verbindung überschrieben werden können.

Im Beispiel wird in der Methode Start über MasterServer eine Liste von angemeldeten Spielinstanzen zu einem selbstgewählten Hostnamen bezogen. Die Methode OnMasterServerEvent wird aufgerufen, wenn die Liste vorliegt. Enthält die Liste Einträge, wird mit Hilfe der Klasse Network eine Verbindung zur ersten Instanz aufgebaut. Andernfalls wird versucht einen neuen Server auf der Maschine zu initialisieren. Ist dies geschehen, wird der Server auf dem MasterServer als Host mit der Spielinstanz „TestGame“ registriert.

Ist die Verbindung etabliert, können Remote Procedure Calls ausgeführt werden wie in der Methode postMessage gezeigt. Dazu müssen aber die aufzurufenden Methoden durch die Compilerdirektive @RPC markiert worden sein.

```
private var tHostName:String = "NetworkRemoteTest";

function postMessage(_m:String) {
    networkView.RPC("ReceiveText", RPCMode.AllBuffered, _m);
}

// try to connect to host
function Start() {
    MasterServer.ClearHostList();
    MasterServer.RequestHostList(tHostName);
}

// continue when host list received
function OnMasterServerEvent(msEvent: MasterServerEvent) {
    Debug.Log("! MasterServerEvent: " + msEvent.ToString());

    if (msEvent == MasterServerEvent.HostListReceived) {
        var hostData: HostData[] = MasterServer.PollHostList();
        if (hostData.Length > 0) {
            // hosts found, connect to first in list
            MasterServer.ClearHostList();
            var e: NetworkConnectionError = Network.Connect(hostData[0]);
            Debug.Log("? Connect: " + e);
            return;
        }

        // no host with the sought-after name found: initialize one!
        Network.InitializeServer(8, 25001, true);
    }
}

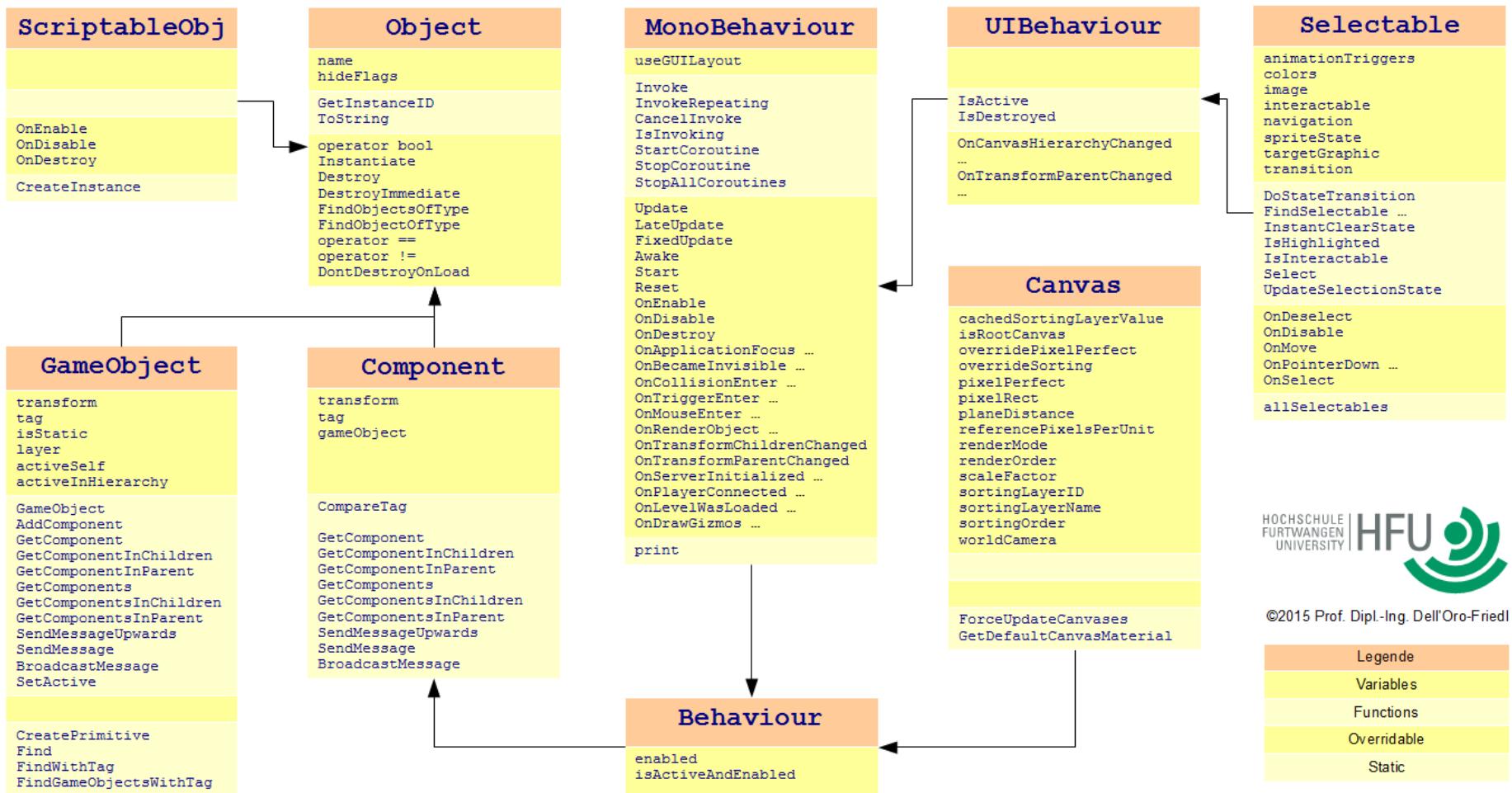
// register host and instance of game
function OnServerInitialized() {
    MasterServer.RegisterHost(tHostName, "TestGame");
}

// post a message when connection established
function OnConnectedToServer() {
    postMessage("Here I am!");
}

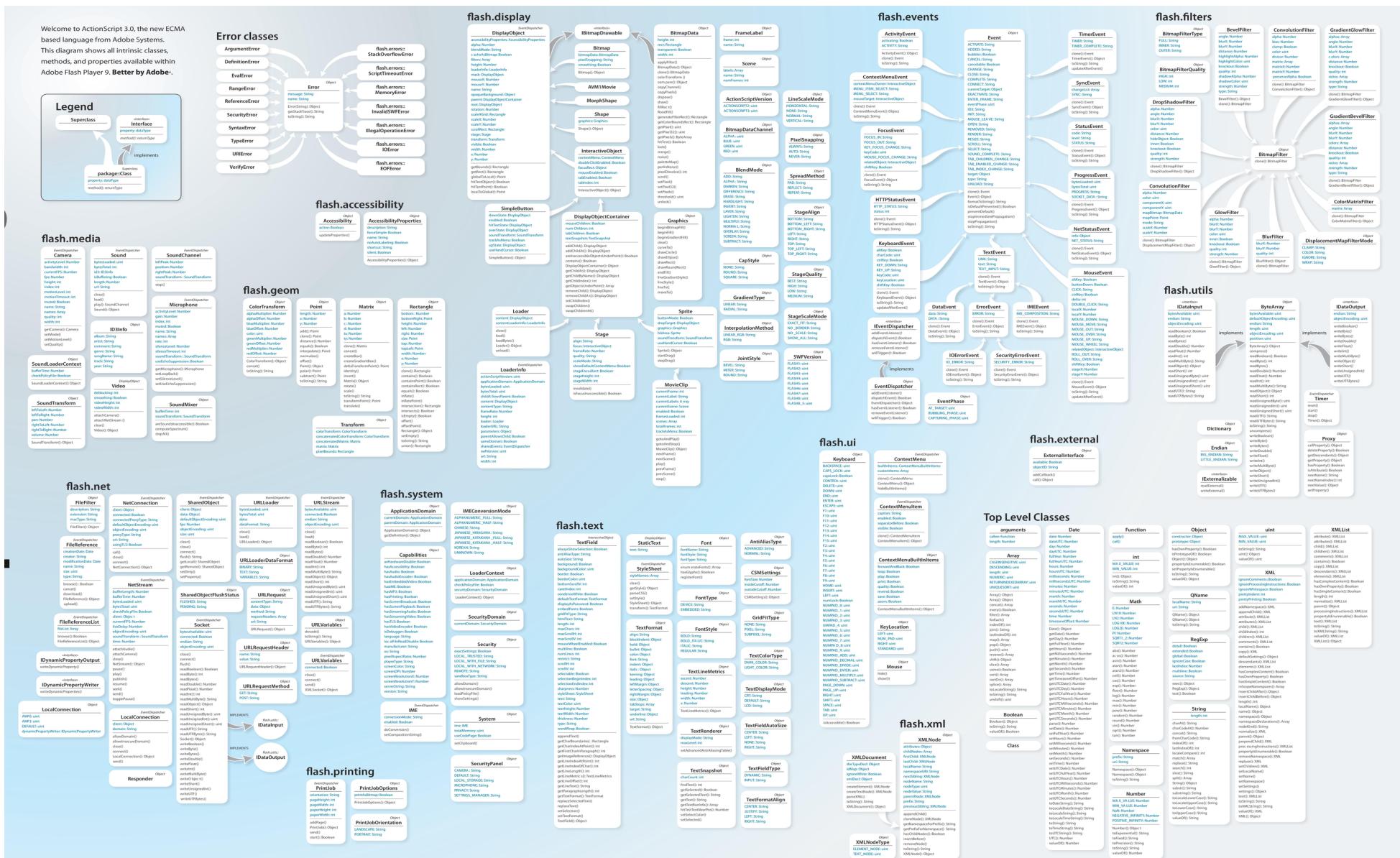
// function defined for remote procedure call
@RPC
function ReceiveText(_text:String) {
    Debug.Log(_text);
}
```

## **21 Anhang**

## 21.1 Unity Class Hierarchy (excerpt)



## **21.2 Flash Class Diagramm**



## **21.3      *Links***

<http://www.adobe.com/devnet/flash.html>

[http://help.adobe.com/en\\_US/FlashPlatform/reference/actionscript3/index.html](http://help.adobe.com/en_US/FlashPlatform/reference/actionscript3/index.html)

<http://www.senocular.com/>

<http://www.kirupa.com/forum/>

<http://unity3d.com/>

<http://docs.unity3d.com/Manual/index.html>

<http://docs.unity3d.com/ScriptReference/index.html>

<http://www.unity3dstudent.com/>

<https://www.assetstore.unity3d.com/en/>