etRocket Rage – Scripting-Referenz

Die Sprache

RocketRage verwendet die Scriptsprache *lua* in der Version 5.1. Das Handbuch dieser Sprache findet sich hier: http://www.lua.org/manual/5.1/

Globale Variablen

Der aktuelle Level steht in der Variablen *level* zur Verfügung. Außerdem stehen alle im Level vorhandenen Spielobjekte unter ihrem Namen (ID) bereit, z.B. *player* oder *target*.

Freie Funktionen

Threading

```
runThread(threadProc : Function, ...)
```

Startet einen neuen Thread, in dem die angegebene lua-Funktion ausgeführt wird. Alle zusätzlich angegebenen Parameter werden der Funktion übergeben. Die Anzahl der übergebenen Parameter und die Anzahl der von der Funktion erwarteten Parameter sollten dabei übereinstimmen. *runThread* kann nur von innerhalb eines Levels aufgerufen werden.

```
sleep(time : Number)
```

Legt den aktuellen Thread (welcher nicht der Haupt-Thread sein darf) für die angegebene Zeit schlafen. Die Zeitangabe erfolgt in Sekunden – auch Bruchteile von Sekunden sind erlaubt. Dies funktioniert nur, wenn sich das Spiel gerade in einem Level befindet, da dort das Management der Threads durchgeführt wird.

Zufall

```
randInt() : Number
randInt(min : Number, max : Number) : Number
```

Liefert eine zufällige Ganzzahl in den angegebenen Grenzen. Die Variante ohne Parameter nutzt den gesamten möglichen Wertebereich aus.

```
randReal() : Number
randReal(min : Number, max : Number) : Number
```

Liefert eine zufällige reelle Zahl in den angegebenen Grenzen. Die Variante ohne Parameter liefert einen zufälligen Wert zwischen 0 und 1.

```
randDir() : Vec2r
```

Liefert einen zufälligen 2D-Richtungsvektor mit der Länge 1.

Textfarben

```
beginColor(color : Vec4r) : String
beginColor(r : Number, g : Number, b : Number, a : Number) : String
endColor() : String
```

Diese Funktionen liefern spezielle Zeichenketten, die eine Änderung der Textfarbe innerhalb eines Texts erlauben. *endColor* macht die Änderung wieder rückgängig. Es sind beliebig tiefe Verschachtelungen möglich. Mit diesen Funktionen lassen sich beispielsweise Teile einer SMS oder eines Texts farblich hervorheben.
Beispiel: `level:addSMS("Ich", "Hallo! " .. beginColor(1, 0, 0, 1) .. "Dieser Text ist rot." .. endColor());`

Objekte

```
areEqual(object1 : Object, object2 : Object) : Boolean
areEqual(object1 : ScriptObject, object2 : ScriptObject) : Boolean
```

Prüft, ob die beiden angegebenen Objekte dieselben sind.

```
toArmed(object : Object) : Armed
toDamagable(object : Object) : Damagable
```

Castet das angegebene Objekt zu einer der Klassen, von denen manche der Objektklassen erben. Wenn das angegebene Objekt nicht von der Zielklasse erbt, dann ist der Rückgabewert *nil*.

```
toAccelerator(object : Object) : Accelerator
toCheckpoint(object : Object) : Checkpoint
toEnemy(object : Object) : Enemy
toEnergyNode(object : Object) : EnergyNode
toGarage(object : Object) : Garage
toLaser(object : Object) : Laser
toLevelTarget(object : Object) : LevelTarget
toMine(object : Object) : Mine
toPlayer(object : Object) : Player
toPowerUp(object : Object) : PowerUp
toProjectile(object : Object) : Projectile
```

Castet das angegebene Objekt zu einer der von *Object* abgeleiteten Klassen. Wenn das angegebene Objekt keine Instanz der Zielklasse ist, dann ist der Rückgabewert *nil*.

```
toCounter(object : ScriptObject) : Counter
```

Castet das angegebene Objekt zu einer der von *ScriptObject* abgeleiteten Klassen. Wenn das angegebene Objekt keine Instanz der Zielklasse ist, dann ist der Rückgabewert *nil*.

Sound und Musik

```
loadSound(filename : String, soundGroup : String, type : String,
          mode : String) : Sound
```

Lädt einen Sound aus der angegebenen Datei *filename*, deren Name sich auf den Pfad bezieht, in dem die ausführbare Datei liegt. *soundGroup* kann „SFX“ für Soundeffekte, „Music“ für Musik oder „Master“ sein. *type* kann die Werte „Sample“ oder „Stream“ annehmen. Längere Sounds sollten als Stream abgespielt werden. *mode* kann „2D“ oder „3D“ sein. 3D-Sounds können im Raum positioniert werden.
Wenn beim Laden ein Fehler aufgetreten ist, dann ist der Rückgabewert *nil*.

```
playSound(filename : String) : SoundInstance
playSound(filename : String, volume : Number, pitch : Number) : SoundInstance
```

Erlaubt es, ohne große Umwege einen Sound abzuspielen, optional mit Lautstärke und Abspielgeschwindigkeit (Standard: beides 1). Der Sound gehört der Soundgruppe „SFX“ an. Die Funktion liefert die abgespielte Soundinstanz zurück, damit sie bei Bedarf noch weiter angepasst werden kann. Wenn ein Fehler aufgetreten ist, ist der Rückgabewert *nil*.

```
playMusic(filename : String) : SoundInstance
playMusic(filename : String, restartIfAlreadyPlaying : Boolean,
          fadeInSpeed : Number, fadeOutSpeed : Number) : SoundInstance
```

Spielt das angegebene Musikstück ab. Die Variante ohne Parameter verwendet als Standardwerte *false* für *restartIfAlreadyPlaying* und jeweils 1 für die Fading-Parameter. Wenn das Abspielen funktioniert hat, ist der Rückgabewert die Instanz der nun spielenden Musik, anderenfalls *nil*.

Pfade

```
getPathFilename(path : String) : String
getPathDirectory(path : String) : String
```

Liefert den Dateinamen bzw. das Verzeichnis eines Pfads. `getPathFilename("data\\test.png")` liefert „test.png“, und `getPathDirectory("data\\test.png")` liefert „data“.

Vektorklassen

Es existieren die Vektorklassen *Vec2r*, *Vec3r* und *Vec4r*, wobei das *r* für *Real* steht.

Konstruktoren

```
Vec2r(v : Number) : Vec2r
Vec3r(v : Number) : Vec3r
Vec4r(v : Number) : Vec4r
Vec2r(x : Number, y : Number) : Vec2r
Vec3r(x : Number, y : Number, z : Number) : Vec3r
Vec4r(x : Number, y : Number, z : Number, w : Number) : Vec4r
```

Die Varianten mit nur einem Parameter kopieren den angegebenen Wert in alle Koordinaten des Vektors. Der Aufruf `Vec3r(1, 1, 1)` ist also bedeutungsgleich mit `Vec3r(1)`.

Attribute

```
readwrite Vec2r.x : Number
readwrite Vec2r.y : Number
readwrite Vec3r.x = Vec3r.r : Number
readwrite Vec3r.y = Vec3r.g : Number
readwrite Vec3r.z = Vec3r.b : Number
readwrite Vec4r.x = Vec4r.r : Number
readwrite Vec4r.y = Vec4r.g : Number
readwrite Vec4r.z = Vec4r.b : Number
readwrite Vec4r.w = Vec4r.a : Number
```

Erlaubt Zugriff auf die Koordinaten des Vektors. *r*, *g*, *b* und *a* sind Synonyme für *x*, *y*, *z* und *w*, außer bei *Vec2r*.

```
readonly Vec2r.perp : Vec2r
```

Liefert den zum Vektor senkrecht stehenden Vektor (*-y*, *x*).

```
readonly Vec*r.isZero : Boolean
```

Ist *true*, wenn der Vektor die Länge null hat.

```
readonly Vec*r.length : Number
readonly Vec*r.lengthSq : Number
```

Erlaubt lesenden Zugriff auf die Länge des Vektors bzw. das Quadrat der Länge.

```
readonly Vec*r.normalized : Vec*r
```

Liefert den normierten Vektor (d.h. die Länge ist 1).

Methoden

```
Vec*r:normalize()
```

Normalisiert den Vektor.

```
Vec*r:dot(rhs : Vec*r) : Number
```

Liefert das Skalarprodukt (Punktprodukt) dieses Vektors mit einem anderen Vektor.

```
Vec3r:cross(rhs : Vec3r) : Vec3r
```

Liefert das Vektorprodukt (Kreuzprodukt) dieses 3D-Vektors mit einem anderen 3D-Vektor.

Operatoren

```
Vec*r + Vec*r : Vec*r
Vec*r - Vec*r : Vec*r
-Vec*r : Vec*r
Vec*r * Number : Vec*r
Number * Vec*r : Vec*r
Vec*r / Number : Vec*r
Vec*r == Vec*r : Boolean
```

Dies sind die üblichen Operatoren, wie man sie von Vektoren kennt.

Sound-Klasse

Die Klasse *Sound* repräsentiert einen geladenen Sound, im Unterschied zur Klasse *SoundInstance*, die eine spezielle Instanz eines Sounds darstellt, die abgespielt werden kann.

Attribute

```
readwrite Sound.defaultMinDistance : Number
readwrite Sound.defaultMaxDistance : Number
```

Minimale und maximale Standarddistanz für neu erstellte Instanzen dieses Sounds. Diese Distanzen bestimmen, ab welcher minimalen Distanz der Sound nicht mehr lauter wird, bzw. ab welcher maximalen Distanz der Sound nicht mehr leiser wird. Sie können für die einzelnen Instanzen des Sounds überschrieben werden. Gilt nur für 3D-Sounds!

```
readwrite Sound.instanceCreationInterval : Number
readwrite Sound.maxPlayingInstances : Number
```

instanceCreationInterval legt die minimale Wartezeit (in Sekunden) zwischen dem Erzeugen zweier Instanzen dieses Sounds fest, *maxPlayingInstances* legt die maximale Anzahl gleichzeitig vorhandener Instanzen fest.

Methoden

```
Sound:createInstance() : SoundInstance
Sound:createInstance(forceCreation : Boolean) : SoundInstance
```

Erzeugt eine Instanz des Sounds und liefert sie zurück, oder *nil* im Falle eines Fehlers (oder wenn auf Grund der Attribute *instanceCreationInterval* bzw. *maxPlayingInstances* gerade keine neue Instanz erzeugt werden kann). Wenn jedoch *true* für *forceCreation* angegeben wird, dann werden diese Beschränkungen ignoriert.

SoundInstance-Klasse

Die Klasse *SoundInstance* steht für eine Instanz eines Sounds. Nur Instanzen können tatsächlich abgespielt werden.

Attribute

```
readwrite SoundInstance.position : Vec2r
readwrite SoundInstance.absPosition : Vec2r
```

Bietet Zugriff auf die relative Position und die absolute Position der Soundinstanz. Die relative Position bezieht sich auf das Objekt, an das die Soundinstanz gebunden ist (falls sie das ist). Gilt nur für 3D-Sounds!

```
readwrite SoundInstance.volume : Number
readwrite SoundInstance.pitch : Number
```

volume ist die Lautstärke des Sounds (0: stumm, 1: volle Lautstärke). Mit *pitch* wird die Abspielrate bestimmt. Bei 1 wird der Sound mit seiner normalen Geschwindigkeit abgespielt, bei 0.5 mit seiner halben, bei 2 mit seiner doppelten, usw.

```
readwrite SoundInstance.looping : Boolean
readwrite SoundInstance.muted : Boolean
readwrite SoundInstance.paused : Boolean
```

Kontrolliert, ob die Soundinstanz in einer Schleife abgespielt werden soll (*looping*), ob sie stummgeschaltet sein soll (*muted*), oder ob sie pausieren soll (*paused*).

```
readwrite SoundInstance.playCursor : Number
```

Setzt oder liest den Abspiel-Cursor der Soundinstanz, angegeben in Sekunden.

```
readwrite SoundInstance.minDistance : Number
readwrite SoundInstance.maxDistance : Number
```
Erlaubt Zugriff auf die Distanzen für diese Soundinstanz. Diese Distanzen bestimmen, ab welcher minimalen Distanz der Sound nicht mehr lauter wird, bzw. ab welcher maximalen Distanz der Sound nicht mehr leiser wird. Sie können für die einzelnen Instanzen des Sounds überschrieben werden. Gilt nur für 3D-Sounds!

```
readwrite SoundInstance.calculateVelocity : Boolean
```
Legt fest, ob die Geschwindigkeit der Soundinstanz anhand der Positionen automatisch berechnet werden soll. Dies ist für den Dopplereffekt wichtig. Gilt nur für 3D-Sounds!

Methoden

```
SoundInstance:play()
SoundInstance:stop()
```
Spielt die Soundinstanz von Beginn an ab, bzw. stoppt sie. Eine gestoppte Soundinstanz darf nicht wieder verwendet werden, da sie anschließend gelöscht wird! Wenn eine Soundinstanz gestoppt und weitergespielt werden soll, ist dafür das Attribut *paused* zu verwenden.

```
SoundInstance:attachTo(Object object)
SoundInstance:attachTo(Object object, maintainAbsPosition : Boolean)
SoundInstance:detachFromParent()
SoundInstance:detachFromParent(maintainAbsPosition : Boolean)
```
Mit *attachTo* wird die Soundinstanz an ein Spielobjekt gekoppelt. Die relative Position (Attribut *position*) bezieht sich dann auf dieses Objekt. Mit *detachFromParent* wird die Soundinstanz von dem Objekt wieder abgekoppelt. Wenn für *maintainAbsPosition* der Wert *true* übergeben wird, dann behält die Soundinstanz während des An-/Abkoppelns ihre absolute Position bei.

```
SoundInstance:setVolumeTarget(target : Number, speed : Number)
SoundInstance:setPitchTarget(target : Number, speed : Number)
```
Legt Zielwerte (*target*) für Lautstärke bzw. Abspielgeschwindigkeit fest, die mit einer bestimmten Geschwindigkeit (*speed*) erreicht werden sollen.

```
SoundInstance:setStopCountDown(time : Number)
SoundInstance:setPauseCountDown(time : Number)
```
Startet einen Stop- bzw. Pause-Count-Down. Nach Ablauf der Zeit wird die Soundinstanz gestoppt (dieselbe Wirkung wie *stop*()) bzw. pausiert. Wird für *time* der Wert -1 angegeben, dann wird die Aktion (Stoppen bzw. Pausieren) dann ausgeführt, wenn der zuvor mit *setVolumeTarget* gesetzte Lautstärkezielwert erreicht wurde. Analoges gilt für einen *time*-Wert von -2 und dem Abspielgeschwindigkeitszielwert (*setPitchTarget*).

Object-Klasse

Alle Spielobjekte sind von *Object* abgeleitet. Sie können jedoch zusätzlich noch von anderen Klassen erben, wie *Armed* für bewaffnete Objekte oder *Damagable* für Objekte, die beschädigt werden können.

Attribute

```
readonly Object.id : String
readonly Object.type : String
```
Die ID und der Typ des Objekts. Die ID ist einzigartig. Sie entspricht der ID, die in der Level-Datei angegeben wurde. Wenn sie dort fehlt, dann wird für das Objekt eine automatische ID nach dem Muster „Obj_*Nr*“ erzeugt. Der Typ eines Objekts entspricht dem Namen der Objektklasse, z.B. „Player“ oder „LevelTarget“.

```
readonly Object.zombie : Boolean
readwrite Object.turnToZombie : Boolean
```
zombie gibt an, ob das Objekt ein Zombie ist. Zombie-Objekte sind nicht wirklich tot und können wiederbelebt werden (so wie der Spieler in einem Checkpoint wiederbelebt wird). *turnToZombie* kontrolliert, ob das Objekt nach seinem Tod (z.B. durch Aufruf der Methode *kill*) zu einem Zombie werden soll oder nicht.

```
readwrite Object.position : Vec2r
readwrite Object.angle : Number
readwrite Object.direction : Vec2r
```
Bietet Zugriff auf die Position und die Ausrichtung des Objekts. *angle* und *direction* sind aneinander gekoppelt. Der Winkel wird im Bogenmaß angegeben. Eine Erhöhung bedeutet eine Drehung im Uhrzeigersinn. *direction* ist die Richtung der y-Achse des Objekts, die normalerweise nach unten zeigt. Ein Winkel von 0 bedeutet eine Richtung von (0, 1).

```
readwrite Object.velocity : Vec2r
readwrite Object.angularVelocity : Number
```
Der Geschwindigkeitsvektor des Objekts und die Winkelgeschwindigkeit. Die Geschwindigkeit wird in Kästchen pro Sekunde angegeben, die Winkelgeschwindigkeit im Bogenmaß (pro Sekunde). Diese Attribute haben auf statische Objekte keinerlei Auswirkung.

```
readwrite Object.mass : Number
```
Bezeichnet die Gesamtmasse des Objekts. Die Masse des Spielers beträgt 1.

```
readwrite Object.gravityInfluence : Boolean
```
Dieses Attribut gibt an, ob das Objekt durch die Gravitation beeinflusst wird (*true*) oder nicht (*false*). Auf statische Objekte hat es keine Wirkung.

```
readwrite Object.movementFriction : Number
readwrite Object.rotationFriction : Number
```
Diese Attribute bestimmen, welcher Anteil der Bewegung bzw. der Rotation des Objekts pro Sekunde erhalten bleibt. Ein Wert von 1 bedeutet, dass 100% erhalten bleiben, also kein Reibungsverlust. Die Standardwerte sind 0.75 und 0.1. Auf statische Objekte haben diese Attribute keine Wirkung.

```
readwrite Object.material : String
```
Das Material des Objekts. Es bestimmt, wie Kollisionen verarbeitet und dargestellt werden. Möglich sind u.a. „Metal“, „Rubber“, „Trigger“ oder „Player“.

```
readwrite Object.visible : Boolean
```
Bestimmt, ob das Objekt sichtbar sein soll. Unsichtbare Objekte werden nicht gerendert, aber trotzdem aktualisiert.

```
readwrite Object.frozen : Boolean
```
Erlaubt es, Objekte einzufrieren. Eingefrorene Objekte werden nicht aktualisiert (aber gerendert). Das Einfrieren von Objekten könnte jedoch unvorhergesehene Folgen haben.

```
readwrite Object.passThrough : Boolean
```
Kollisionen, an denen ein Objekt beteiligt ist, dessen *passThrough*-Attribut auf *true* gesetzt ist, haben keine physikalische Auswirkung, die Objekte können sich also gegenseitig durchqueren.

```
readwrite Object.target : Object
```
Dieses Attribut bestimmt das Zielobjekt eines Objekts. Es wird nur von Gegnern und zielsuchenden Geschossen beachtet. Gegner haben als Standardziel den Spieler.

```
readwrite Object.playerCollisionDamageScale : Number
```
Ein Faktor, der die Schadenswirkung dieses Objekts bei einer Kollision mit dem Spieler skaliert. Ein Wert von 0.5 bedeutet beispielsweise, dass der Spieler bei einer Kollision nur die Hälfte des „normalen“ Schadens erleidet.

```
readwrite Object.categoryBits : Number
readwrite Object.collideBits: Number
```
Bei diesen Attributen handelt es sich um eine bitweise Oder-Verknüpfung von bestimmten vorgegebenen Flags. *categoryBits* bestimmt, welchen Kollisionskategorien das Objekt angehört, und *collideBits* bestimmt, mit welchen Kategorien dieses Objekt kollidieren kann. Da es in lua keine bitweisen Verknüpfungsoperatoren gibt, muss man die Flags mit dem +-Operator addieren, was in diesem Fall zum selben Ergebnis führt. Folgende Kategorien stehen zur Verfügung:

- *CollisionBits.None*: Keine Kategorie
- *CollisionBits.Static*: statische Objekte
- *CollisionBits.Player*: der Spieler
- *CollisionBits.Enemy*: Gegner
- *CollisionBits.Projectile*: Geschosse
- *CollisionBits.Laser*: Laserstrahl
- *CollisionBits.Object*: dynamische Objekte
- *CollisionBits.BlockView*: Objekt kann die Sicht blockieren
- *CollisionBits.All*: alle Bits

```
readwrite Object.onCollision : Function
```
Legt eine Callback-Funktion für die Kollisionen dieses Objekts fest. Die Funktion wird mit zwei Parametern aufgerufen, die die beiden kollidierenden Objekte enthält. Um den Callback wieder aufzuheben, muss man diesem Attribut den Wert *nil* zuweisen.

```
readwrite Object.onKilled : Function
readwrite Object.onReanimated : Function
```
Legt Callback-Funktionen für den Tod und die Wiederbelebung eines Objekts fest. Die Funktionen bekommen als Parameter das Objekt übergeben, das gestorben ist bzw. wiederbelebt wurde.

Methoden

```
Object:setCollisionEnabled(object : Object, enable : Boolean)
Object:isCollisionEnabled(object : Object) : Boolean
```
Mit *setCollisionEnabled* kann die Kollision dieses Objekts mit einem anderen Objekt ein- oder ausgeschaltet werden. Ausgeschaltete Kollisionen werden ignoriert. Es wird auch kein Kollisions-Callback aufgerufen. Mit *isCollisionEnabled* kann überprüft werden, ob die Kollision mit einem anderen Objekt ein- oder ausgeschaltet wurde.

```
Object:relToAbsPos(position : Vec2r) : Vec2r
Object:absToRelPos(position : Vec2r) : Vec2r
Object:relToAbsAngle(angle : Number) : Number
Object:absToRelAngle(angle : Number) : Number
Object:relToAbsDir(direction : Vec2r) : Vec2r
Object:absToRelDir(direction : Vec2r) : Vec2r
```
Rechnet objektrelative Positionen, Winkel und Richtungen in absolute Werte um, oder umgekehrt. `player.relToAbsPos(Vec2r(0,-5))` liefert beispielsweise die absolute Position des Punkts, der sich 5 Einheiten vor dem Spieler befindet.

```
Object:kill()
Object:reanimate()
```
kill sorgt dafür, dass das Objekt beim nächsten Update „unspektakulär“ gelöscht wird. Wenn jedoch *turnToZombie* auf *true* gesetzt ist, wird das Objekt nicht wirklich gelöscht, sondern verwandelt sich in einen Zombie. *reanimate* belebt das Objekt wieder, was aber nur funktioniert, wenn es ein Zombie ist. Achtung: Attribute wie *health* werden durch die Wiederbelebung nicht verändert! Wenn das Objekt also gestorben ist, weil seine Gesundheitspunkte null geworden sind, müssen sie sofort nach der Wiederbelebung wieder auf einen positiven Wert gesetzt werden, denn sonst stirbt das Objekt gleich wieder.

```
Object:explode(debrisCount : Number, debrisType : Number, size : Number,
               makeHole : Boolean, volume : Number)
```
Erzeugt eine Explosion an der Position des Objekts (es wird dadurch aber nicht gelöscht). *debrisCount* gibt die Anzahl der Trümmerstücke an, *debrisType* ihren Typ. 0 erzeugt eine Mischung aus allen Trümmertypen. *size* gibt die Größe der Explosion an. *makeHole* bestimmt, ob die Explosion ein Loch im Papier hinterlassen soll, und *volume* gibt die Lautstärke des Explosionssounds an (von 0 bis 1).

```
Object:canSee(object : Object) : Boolean
Object:canSee(object : Object, maxDistance : Number, relEyePos : Vec2r) : Boolean
```
Testet, ob dieses Objekt das angegebene Objekt sehen kann. *maxDistance* ist die maximale Sichtweite (Standard: 25), und *relEyePos* gibt die zum Objekt relative Position des „Auges“ an. Von dieser Stelle aus wird ein Sichtstrahl in Richtung des Zielobjekts geschossen und geprüft, ob dieser irgendein sichtblockierendes Objekt trifft.

Damagable-Klasse

Von der *Damagable*-Klasse erben alle Objekte, die beschädigt werden können.

Attribute

```
readwrite Damagable.health : Number
readwrite Damagable.maxHealth : Number
```
Die Gesundheit und die maximale Gesundheit des Objekts.

```
readwrite Damagable.invincible : Boolean
```
Legt fest, ob das Objekt unbesiegbar sein soll. In diesem Fall kann der *health*-Wert nicht verringert werden.

Armed-Klasse

Von dieser Klasse erben alle bewaffneten Objekte.

Attribute

```
readwrite Armed.weapon : Number
```
Die aktive Waffe des Objekts. Folgende Waffen stehen zur Verfügung:

- *Weapon.BlasterBlue*: die Standard-Tintenkanone (blau)
- *Weapon.Missile*: Streichholzraketen
- *Weapon.Cannon*: Erbsenkanone
- *Weapon.BlasterGreen*: Tintenkanone (grün)
- *Weapon.AA*: grünes Flugabwehrgeschoss

Methoden

```
Armed:setupWeapon(weapon : Number, relPos : Vec2r)
```
Richtet die angegebene Waffe an der angegebenen relativen Position des Objekts ein.

```
Armed:setAmmo(weapon : Number, ammo : Number)
Armed:getAmmo(weapon : Number) : Number
```
Setzt bzw. liefert die Munition für die angegebene Waffe. Wird für *ammo* der Wert -1 angegeben, dann erhält das Objekt unendlich viel Munition für diese Waffe.

```
Armed:fire(direction : Vec2r, absDirection : Boolean) : Projectile
Armed:fireAt(target : Object) : Projectile
```

Feuert die aktuelle Waffe in eine bestimmte Richtung (*direction*) oder auf ein Zielobjekt (*target*) ab. Bei der Richtungsangabe wird mit *absDirection* kontrolliert, ob diese absolut oder relativ zum Schützen ist. Zurückgeliefert wird das abgefeuerte Projektil oder *nil*, falls nicht gefeuert wurde (z.B. weil die Waffe noch nicht nachgeladen oder keine Munition mehr vorhanden war).

Player-Klasse

Diese Klasse repräsentiert den Spieler.

Erbt von: *Object*, *Damagable*, *Armed*

Attribute

```
readwrite Player.score : Number
readwrite Player.fuel : Number
readwrite Player.shield : Number
readwrite Player.mouseAiming : Boolean
```

Diese Attribute erlauben den Zugriff auf die Punktzahl des Spielers, seinen Treibstoff (0 bis 100), die verbleibende Schutzschildzeit (in Sekunden) und den Status der Mauszielhilfe.

Laser-Klasse

Die Klasse *Laser* steht für einen Laserstrahler, der seine Energie aus Energieknoten (*EnergyNode*) bezieht.

Erbt von: *Object*, *Damagable*

Attribute

```
readwrite Laser.active : Boolean
readonly Laser.powered : Boolean
```

Mit *active* kann der Laser ausgeschaltet werden, auch wenn er noch mit Energie versorgt wird. Das geschützte Attribut *powered* liefert *true*, wenn der Laser noch Energie erhält.

PowerUp-Klasse

Diese Klasse repräsentiert ein Power-Up.

Erbt von: *Object*

Attribute

```
readonly PowerUp.subType : Number
```

Liefert den Typ des Power-Ups.

```
readwrite PowerUp.reward : Number
readwrite PowerUp.shieldDuration : Number
```

reward bestimmt die Anzahl der Punkte, die der Spieler für das Einsammeln erhält. *shieldDuration* gibt beim Schutzschild-Power-Up die Schutzschilddauer (in Sekunden) an.

Checkpoint-Klasse

Diese Klasse steht für einen Checkpoint, bei dem der Spieler wieder neu einsteigen kann, nachdem seine Rakete zerstört wurde.

Erbt von: *Object*

Attribute

```
readonly Checkpoint.activated : Boolean
readonly Checkpoint.activationTime : Number
```

Diese geschützten Attribute geben an, ob der Checkpoint schon aktiviert wurde, und falls ja, wann dies geschehen ist (in Sekunden seit Level-Start).

```
readwrite Checkpoint.number : Number
```

Bestimmt die Nummer des Checkpoints. Der Spieler steigt immer beim Checkpoint mit der höchsten Nummer wieder ein. Haben mehrere Checkpoints dieselbe Nummer, dann gewinnt der, der zuletzt aktiviert wurde.

```
readwrite Checkpoint.healthAmount : Number
readwrite Checkpoint.fuelAmount : Number
readwrite Checkpoint.shieldDuration : Number
```

Diese Attribute bestimmen, was geschieht, wenn der Spieler an diesem Checkpoint neu einsteigt. *healthAmount* gibt den Gesundheitszustand an, den der Spieler dann haben soll (0 bis 100), *fuelAmount* seine Treibstoffmenge und *shieldDuration* die Schutzschilddauer. Wenn für *fuelAmount* 0 oder eine negative Zahl angegeben wird, dann wird der Betrag dieser Zahl auf den noch vorhandenen Treibstoff des Spielers addiert. Ist *fuelAmount* beispielsweise -25, dann erhält der Spieler 25 Treibstoffeinheiten zusätzlich zu seinem zuvor vorhandenen Treibstoff beim Wiedereinstieg an diesem Checkpoint. Ist *fuelAmount* jedoch +25, dann hat der Spieler nach dem Wiedereinstieg genau 25 Treibstoffeinheiten.

Garage-Klasse

Diese Klasse repräsentiert eine Werkstatt/Tankstelle.

Erbt von: *Object*

Attribute

```
readwrite Garage.repairPrice : Number
readwrite Garage.refuelPrice : Number
```

Legt die Preise für das Reparieren und Auftanken (pro Einheit) fest. Standardwerte sind 50 und 25.

```
readwrite Garage.repairCapacity : Number
readwrite Garage.refuelCapacity : Number
```

Legt die Kapazität dieser Werkstatt/Tankstelle fest.

Mine-Klasse

Die *Mine*-Klasse repräsentiert eine Mine, die alle mit ihr kollidierenden Objekte zerstört (wenn sie von *Damagable* erben).

Erbt von: *Object*

Attribute

```
readwrite Mine.activated : Boolean
```

Erlaubt es, die Mine ein- und auszuschalten. Ausgeschaltete Minen explodieren nicht bei Kontakt mit Objekten.

Methoden

```
Mine:detonate()
```

Lässt die Mine detonieren.

Accelerator-Klasse

Diese Klasse steht für einen Beschleuniger. Er beschleunigt alle dynamischen Objekte, die sich in seinem Wirkungsbereich befinden, mit einer bestimmten Stärke auf eine Zielgeschwindigkeit.

Attribute

```
readonly Accelerator.radius : Number
readwrite Accelerator.activated : Boolean
readwrite Accelerator.targetSpeed : Number
readwrite Accelerator.strength : Number
```

Der Beschleuniger beschleunigt die Objekte, die sich innerhalb seines Radius (*radius*) mit der Stärke *strength* auf die Zielgeschwindigkeit *targetSpeed*. Ist das Objekt bereits schneller und fliegt in die richtige Richtung, dann passiert nichts. Mit *activated* kann der Beschleuniger ein- und ausgeschaltet werden.

Enemy-Klasse

Diese Klasse besitzt noch keine speziellen Attribute oder Methoden.

Erbt von: *Object*, *Damagable*, *Armed*

EnergyNode-Klasse

Diese Klasse besitzt noch keine speziellen Attribute oder Methoden.

Erbt von: *Object*, *Damagable*

LevelTarget-Klasse

Diese Klasse besitzt noch keine speziellen Attribute oder Methoden.

Erbt von: *Object*, *Damagable*

ScriptObject-Klasse

Die Klasse *ScriptObject* ist die Basisklasse von Script-Hilfsobjekten wie Zeitzählern.

Attribute

```
readonly ScriptObject.type : String
```

Der Typ eines Script-Objekts entspricht dem Namen der Objektklasse, z.B. „Counter“.

Methoden

```
ScriptObject:kill()
```

Sorgt dafür, dass das Script-Objekt beim nächsten Update gelöscht wird.

Counter-Klasse

Diese Klasse repräsentiert Zeitzähler, die vorwärts oder rückwärts laufen und wahlweise am unteren Bildschirmrand dargestellt werden können. Beim Ablauf der vorgegebenen Zeit wird eine frei

wählbare Rückruffunktion aufgerufen.

Erbt von: *ScriptObject*

Attribute

```
readwrite Counter.displayMode : Number
readwrite Counter.color : Vec4f
readwrite Counter.numBeeps : Number
```

Diese Attribute bestimmen das „äußere Erscheinungsbild“ des Zählers. *displayMode* kann 0, 1 oder 2 sein. Bei 0 wird der Zähler nicht angezeigt, bei 1 werden nur die Minuten und Sekunden angezeigt, und bei 2 werden zusätzlich auch Hundertstelsekunden angezeigt. *color* gibt die Farbe des Texts an, und *numBeeps* bestimmt, ab wie vielen Sekunden vor dem Ablauf der Zeit Piepstöne abgespielt werden soll. Bei 0 wird kein Ton abgespielt.

```
readwrite Counter.direction : Number
readwrite Counter.duration : Number
readwrite Counter.time : Number
readwrite Counter.paused : Boolean
readwrite Counter.onElapsed : Function
```

direction gibt die Zählrichtung an: ein positiver Wert steht für vorwärts, ein negativer Wert steht für rückwärts. *duration* ist die Zeit, nach der der Zähler abläuft. Mit *time* kann die bereits abgelaufene Zeit (vorwärts) bzw. die noch übrige Zeit (rückwärts) abgefragt oder gesetzt werden. Mit *paused* kann der Zähler angehalten werden. Wenn der Zähler abgelaufen ist, wird die Funktion *onElapsed* aufgerufen. Sie erhält als Argument das *Counter*-Objekt. Wenn *nil* angegeben wird, wird keine Funktion aufgerufen.

Level-Klasse

Diese Klasse repräsentiert den gesamten Level. Sie übernimmt Aufgaben wie die Verwaltung, das Aktualisieren und das Rendern der Spielobjekte.

Attribute

```
readonly Level.filename : String
readonly Level.name : String
readonly Level.type : String
readonly Level.description : String
```

Diese geschützten Attribute erlauben den Zugriff auf den Dateinamen, den Namen, den Typ („Standard“, „TimeAttack“) und den Beschreibungstext des Levels.

```
readwrite Level.gravity : Vec2r
```

Der Gravitationsvektor des Levels. Er bestimmt Richtung und Stärke der Gravitation. Standard ist (0, 8).

```
readonly Level.time : Number
readonly Level.raceTime : Number
```

Die Uhren des Levels – die Anzahl der seit Start des Levels (*time*) bzw. seit Start des Rennens (*raceTime* – nur bei Time-Attack-Levels) vergangenen Sekunden.

```
readwrite Level.cameraTracking : Boolean
readwrite Level.cameraTarget : Object
readwrite Level.cameraPosition : Vec2r
readwrite Level.cameraZoom : Number
readwrite Level.cameraAngle : Number
```

Diese Attribute bestimmen das Verhalten der Kamera. *cameraTracking* schaltet die automatische Kameraverfolgung ein oder aus. Wenn sie eingeschaltet ist, verfolgt die Kamera das unter *cameraTarget* angegebene Objekt. Anderenfalls können mit den anderen Attributen die Position, der Zoom (Skalierungsfaktor für die Ansicht) und der Winkel der Kamera direkt gesetzt werden.

```
readonly Level.objects : Collection
readonly Level.visibleObjects : Collection
```
Diese Attribute erlauben es, mit einem *for in*-Schleifenkonstrukt alle Objekte (*objects*) oder alle momentan gerenderten Objekte (*visibleObjects*) zu durchlaufen.

Methoden

```
Level:getObject(id : String) : Object
```
Liefert das Objekt mit der angegebenen ID, oder *nil*, falls kein solches Objekt existiert.

```
Level:addPlayerScore(amount : Number)
Level:addPlayerScore(amount : Number, textPosition : Vec2r)
```
Fügt dem Spieler Punkte hinzu (oder zieht sie ihm ab, wenn *amount* negativ ist). Die zweite Variante erzeugt gleichzeitig einen Text, der die hinzugekommene/abgezogene Punktzahl anzeigt. Hierzu muss eine Textposition angegeben werden. Achtung: Der Text wird in Time-Attack-Levels nicht angezeigt!

```
Level:addText(text : String, position : Vec2r, color : Vec4r, size : Number,
              timeToLive : Number)
```
Erzeugt einen neuen Text an der angegebenen Stelle, der langsam nach oben schwebt und unsichtbar wird. *color* ist die Textfarbe im RGBA-Format (Farbkomponenten jeweils zwischen 0 und 1), *size* ist die anfängliche Texthöhe in Kästchen, und *timeToLive* ist die Lebensdauer des Texts in Sekunden. Der Text kann Farbcodes (*beginColor*, *endColor*) enthalten.

```
Level:addSMS(sender : String, text : String)
```
Schickt eine SMS an den Spieler. *sender* ist der Absender, *text* ist die eigentliche Nachricht. Wenn die Nachricht zu lang ist, wird sie auf mehrere Seiten aufgeteilt. Jede Seite besteht aus 6 Zeilen. Sowohl der Absender als auch die Nachricht können Farbcodes (*beginColor*, *endColor*) enthalten.

```
Level:addTimerCallback(callback : Function, interval : Number)
Level:removeTimerCallback(callback : Function)
```
Fügt einen neuen Timer-Callback hinzu bzw. entfernt ihn wieder. Die angegebene Callback-Funktion wird im durch *interval* angegebenen Intervall (in Sekunden) kontinuierlich aufgerufen, bis der Callback wieder entfernt wird. Die Callback-Funktion bekommt als Parameter diese Intervalldauer übergeben.

```
Level:createCounter(displayMode : Number, direction : Number, duration : Number,
                    onElapsed : Function) : Counter
```
Erzeugt ein Objekt vom Typ *Counter* mit den angegebenen Parametern (siehe Abschnitt „*Counter*-Klasse“).