Tutorial 01: Die erste Szene - Kamera, Licht, Model, Animation

>>> VORWORT:

Dieses Tutorial setzt Grundkenntnisse in Sachen 3D und GML voraus. Es ist von Vorteil, vorher die offiziellen Tutorials auf der **GMOgre** Projektseite durchzuarbeiten, da sich meine Arbeit daran orientiert und darauf aufbaut. Über die Linksammlung weiter unten könnt ihr die offizielle Projektseite besuchen und euch die Tutorials anschauen. Ich werde selbstverständlich versuchen, auf möglichst viele Punkte einzugehen, aber jedes Detail abzuhandeln, würde den Rahmen sprengen.

Die hier verwendeten Ressourcen (3D Modelle und Texturen) stammen aus den Examples von **GMOgre**.

Bei Fragen könnt ihr euch jederzeit melden. Ich - und bestimmt auch einige andere User - werden auf jeden Fall versuchen, auf jedes noch so banale Problem einzugehen.

An dieser Stelle möchte ich frischideu für die Übersetzung der ersten offiziellen Einführung in GMOgre danken. Ich habe es bereits in die nachstehende Linkliste unten aufgenommen. Da ich in meinem Tutorial aufgrund des Umfangs nicht auf sämtliche Grundlagen im Detail eingehen kann, bietet es sich an, direkt auf Deutsche Versionen der „Basic Tutorials“ verweisen zu können. Dankeschön!

>>> LINKS UND DOWNLOADS RUND UM GMOGRE 3D:

Diese Liste enthält Links zu wichtigen Anlaufstellen zum GM Port der OGRE 3D Engine.

01. GMOgre 3D – Project Website [incl. Downloads / Tutorials / Function List]
02. GMOgre 3D – offizielles Forum (Englisch)
03. GMOgre 3D – Thema (www.gm-d.de)
04. GMOgre 3D – "Basic Tutorial 1" Übersetzung von frischideu

Bei Problemen mit GMOgre überprüft bitte, ob ihr die aktuellste Version von DirectX installiert habt.

Übersicht: häufig verwendete Begriffe

Dieser Bereich bietet kurze Erklärungen zu Begriffen rund um das Thema 3D. Wenn ihr im Tutorial auf Begriffe stoßt, die ihr nicht kennt, schaut erst hier nach. Wenn der Begriff noch nicht in dieser Liste steht, meldet euch bitte über www.gm-d.de bei mir – entweder im Thread oder per PM. Ich werde versuchen, die Liste immer aktuell zu halten.

>>> 3D ALLGEMEIN:

Mesh	Objekt aus Polygonen, 3D Model
Ambient Light	Umgebungslicht – Mindesthelligkeit der gesamten Szene
Skeletal / Bones Animation	unsichtbares „Skelett“ gesteuert im Mesh steuert Animation

>>> GMOGRE SPEZIFISCH:

Scene Manager	sämtliche Objekte in der Szene werden hier gespeichert und verwaltet
Scene Nodes	unsichtbare Halter für unbegrenzte Anzahl von Entities
Entities	alle Objekte, die aus Polygonen bestehen, sind Entities werden nur in Szene platziert, wenn sie an einen Scene Node geheftet sind
Viewport	2D Fläche, auf die die Szene gerendert wird (z.B. Fenster)

>>> PART I: GET READY!

Willkommen zu meinem ersten GMOgre 3D Tutorial!

Vorweg: Dieses Tutorial sollte ursprünglich nur die Themen Charaktersteuerung und -animation beinhalten. Da jedoch mehrere Leute Interesse an einem Deutschen Einsteigertutorial geäußert haben, gehen wir den ganzen Weg vom Starten der Engine, über Licht und Kamera, bis hin zur Animation. Nehmt euch also etwas Zeit und versucht bitte, das Tutorial am Stück durchzuarbeiten.

Es wäre ratsam, wenn ihr euch auf der offiziellen Website die Funktionsliste aufruft, falls ihr die DLL verwendet. Ich empfehle, die GEX Erweiterung von GMOgre zu benutzen. Dadurch habt ihr nicht nur sämtliche Funktionen und Argumente im Überblick, sondern auch die vollständige aktuelle Hilfedatei direkt im GM. So könnt ihr schnell die Funktionen nachschlagen, die hier unerwähnt bleiben. Ich muss diese Passagen vor allem dann auslassen, wenn sie zu komplex oder sehr umfangreich sind.

Die Funktionsliste im offiziellen Wiki findet ihr in der linken Spalte als vorletzten Eintrag.

Von mir aus kann's losgehen! 

>>> PART 2: START YOUR ENGINE!

***Wichtig:** Sobald **GMOgre** läuft, hat der **GM** keinen Zugriff mehr auf die Zeichenregion des Fensters.*

Das hält uns aber nicht davon ob, die 3D Engine trotzdem zu initialisieren! Also brauchen wir zuerst ein Objekt, welches mit **GMOgre** kommuniziert. Gebt ihm den Namen ***obj_engine***.

Die ersten beiden Zeilen starten die Engine und legen ***DirectX*** zum Rendern fest. Alternativ kann auch ***OpenGL*** verwendet werden, sofern es inzwischen funktioniert.

Schreibt oder kopiert diese 2 Zeilen jetzt ins Create Event von ***obj_engine***:

```
InitializeOgre3D();
StartOgre3DEngine(RENDER_DX9);
```

Gut, jetzt müssen wir der Engine ein paar Infos zukommen lassen. Zuerst soll sie erfahren, wo sich die Ressourcen (z.B. 3D Modelle, Texturen, Scripts, etc.) befinden, die wir im Projekt verwenden wollen. Diese Eigenart von GMOgre ist sehr komfortabel, da wir später nicht für jedes 3D Modell, jedes Material und jegliche sonstige Ressourcen immer wieder das entsprechende Verzeichnis angeben müssen. **GMOgre** durchsucht automatisch alle anfangs festgelegten Verzeichnisse und ZIP-Archive, wenn etwas geladen werden soll.

Fügt hierzu die folgenden Zeilen im Create Event an:

```
AddResourceLocation("./Data/materials/scripts", LOC_FILESYSTEM);
AddResourceLocation("./Data/materials/textures", LOC_FILESYSTEM);
AddResourceLocation("./Data/models", LOC_FILESYSTEM);
InitializeAllResourceGroups();
```

Es gibt 2 Möglichkeiten, Ressourcen zu speichern: in normalen Verzeichnissen oder in ZIP-Archiven:

LOC_FILESYSTEM	*muss für normale Verzeichnisse angegeben werden.*
LOC_ZIP	*muss für ZIP-Archive angegeben werden.*

GMOgre benötigt einen ***Scene Manager,*** der alle vorhandenen Objekte speichert und verwaltet. Ohne ihn geht gar nichts! Also erzeugen wir ihn als nächstes:

```
CreateSceneManager(ST_GENERIC);
```

Es gibt mehrere ***Scene Manager Typen*** für verschiedene Zwecke. Zum Teil erklären sie sich selbst, aber ganz ehrlich – ich wüsste nicht, wann ich welchen einsetzen soll. Deshalb bietet sich für den Anfang ST_GENERIC an. Nicht auf optimierte Performance ausgelegt, aber ideal für den Einstieg. Auf www.ogre3d.org erfahrt ihr mehr...

ST_GENERIC	Generic scene manager
ST_EXTERIOR_CLOSE	Terrain Scene Manager
ST_EXTERIOR_FAR	Nature scene manager
ST_EXTERIOR_REAL_FAR	Paging Scene Manager
ST_INTERIOR	BSP scene manager

Als nächstes wird der ***Viewport*** eingerichtet. Hierbei handelt es sich sozusagen um eine zweidimensionale Fläche, auf die das Bild projiziert wird, welches später unsere Kamera einfängt. In unserem Fall ist es einfach ausgedrückt das Fenster.

Für die Dimensionen des ***Viewports*** übernehmen wir einfach die Maße des Rooms, den wir im Anschluss erstellen. Fügt folgende Zeile ins Create Event ein:

```
view_id = CreateViewport(0, 0, 0, room_width, room_height);
```

Mittels ***Ambient Light*** (Umgebungslicht) können wir der Szene Grundhelligkeit und -farbe verpassen. Da wir später eine Lichtquelle integrieren und auch Schatten sehen wollen, ist hierfür ein dunkler Farbton ideal. Ich hab mich für **c_navy** entschieden.

Ihr habt fast freie Wahl: je heller die Umgebung, desto transparenter Schatten!

Fügt wieder die folgenden Zeilen ins Create Event:

```
SetAmbientLight(c_navy);
SetShadowTechnique(SHADOWTYPE_STENCIL_MODULATIVE);
```

Was in der ersten Zeile passiert, dürfte klar sein. Zeile 2 legt fest, welchen Schattentyp wir für die Szene verwenden. Ich habe mich hier für einen mit mittleren Anforderungen entschieden, der auf jedem Rechner funktionieren sollte.

Auf dieser Seite könnt ihr u.a. die Schattentypen einsehen: Scene Manager Functions

Nun sind alle wichtigen Voreinstellungen getroffen.

Jetzt kümmern wir uns darum, dass unsere Szene regelmäßig aktualisiert und dargestellt wird.

Setzt folgende Zeile ins End Step Event von ***obj_engine:***

```
RenderFrame();
```

Dieser Befehl veranlasst **GMOgre** in jedem Step (bzw. am Ende jedes Steps) das von der Kamera eingefangene Bild auf den eben eingerichteten Viewport zu zeichnen.

Hinweis: `RenderFrame()` ***kann alternativ auch ins Step Event gesetzt werden.***

Da **GMOgre** nun für euer Projekt eingerichtet ist, könnt ihr ***obj_engine*** schließen und wir wenden uns den nächsten Dingen zu.

Ich schlage vor, wir erstellen an dieser Stelle erstmal den Room, in dem sich alles abspielen wird.
Wie das geht, dürfte jeder wissen, der schonmal mit ***GM*** gearbeitet hat.

Einen besonderen Namen braucht ihr dem Room nicht zu geben. Das spielt weder für **GMOgre** noch für dieses Tutorial eine Rolle.

Wichtiger ist die Größe des Rooms. Ihr könnt sie einstellen, wie ihr wollt. Vergesst nur nicht, dass die Größe des Fensters (siehe ***Viewport***) den Maßen des Rooms entsprechen wird.
Mein Room hat folgende Dimensionen: 800 x 600 px.

Wenn ihr euch entschieden habt, platziert ***obj_engine*** im Room, vorzugsweise in der linken oberen Ecke, da noch ein paar Objekte folgen und wir nicht die Übersicht verlieren wollen.

Großartig, dieses Kapitel können wir als abgeschlossen betrachten!

>>> PART 3: DIE WELT LIEGT DIR ZU FÜßEN...

Wir sollten einen Boden erzeugen, auf dem unser Roboter später umher stolzieren kann. Also, erzeugt ein neues Objekt und nennt es ***obj_floor***. Platziert es anschließend im Room, rechts neben ***obj_engine***.

Kopiert nun den folgenden Codeblock ins Create Event. Danach schauen wir uns die Zeilen näher an.

```
ent_id = CreateFloorEntity(1500, 1500, 20, 20, 16, 16);
node_id = CreateRootChildSceneNode();
AttachEntityToSceneNode(ent_id, node_id);
SetEntityMaterial(ent_id, "Examples/BumpyMetal");
```

In der ersten Zeile wird die Bodenfläche erzeugt. Ihre **ID** wird in **ent_id** gespeichert.
Hier ist eine nähere Erklärung des Scripts:

CreateFloorEntity(*Weite, Höhe , X-Segmente , Y-Segmente , Textur hrepeat , Textur vrepeat* ***);***

Vielleicht fällt euch auf, dass hier gar keine Koordinaten vorhanden sind, um die Fläche in der Szene zu positionieren. Hier werden neben Länge und Breite nur die Anzahl der Segmente angegeben, und wie oft die Textur wiederholt werden soll. Wir treffen hier auf eine weitere Eigenart der OGRE 3D Engine, die bei näherem Betrachten aber recht praktisch sein kann:

Sämtliche ***Meshes*** (Objekte, die aus Polygonen bestehen) müssen einem sogenannten ***Scene Node*** untergeordnet werden, um in der 3D Szene aufzutauchen. ***Scene Nodes*** sind unsichtbare Punkte („Knoten“) mit eigenen änderbaren Koordinaten. Ihnen können unendlich viele ***Meshes*** zugewiesen werden, sogar andere ***Scene Nodes*** lassen sich unterordnen.

Schauen wir uns Zeile 2 an: Hier wird ein ***Scene Node*** als ***Child*** des ***Root Node*** erzeugt. Das bedeutet nur, dass der ***Scene Node*** direkt dem ***Scene Manager*** untergeordnet ist.

Zeile 3 ordnet die Bodenfläche dem eben erzeugten ***Scene Node*** unter und Zeile 4 legt die Textur fest.

Lest zu diesem Thema auch frischideu's Übersetzung des Basic Tutorial 1: Basiswissen 1

Bitte versucht vorerst nicht, die Textur des Bodens in der letzten Zeile zu ändern. Die Materialzuweisung erfolgt durch ein externes Script.

Dazu gibt es ein weiteres Tutorial von mir:

[GMOgre 3D] Tutorial 02: Materials (auf gm-d.de)

>>> PART 4: AUF SENDUNG!

Nun, da wir einen Boden haben, können wir auch ruhig schonmal einen Blick auf die Szene werfen. Genau wie beim Filmdreh benötigen wir eine Kamera, die die Szene von ihrer aktuellen Position aufnimmt, um sie dann an den ***Viewport*** weiterzugeben.

Für Kameras gibt es unzählige Einstellungsmöglichkeiten. Wenn wir später den Roboter in der Szene haben, werden wir eine sehr nützliche Funktion kennenlernen.

Ihr könnt gern schon einen Blick auf die Funktionen werfen: Camera Functions

Erstellt nun ein neues Objekt, nennt es ***obj_camera*** und platziert es im Room neben ***obj_floor***.

Das Create Event eröffnen wir mit folgendem Zweizeiler:

```
cam_id = CreateCamera(room_width / room_height, 5, 0, 45);
SetViewportCamera(obj_engine.view_id, cam_id);
```

Mit der ersten Zeile wird die Kamera erzeugt und ihre **ID** zurückgegeben. Grundsätzlich spricht nichts dagegen, die hier eingesetzten Parameter auch in euren kommenden Projekten so zu verwenden. Hier könnt ihr euch mal anschauen, wofür die unterschiedlichen Argumente stehen:

CreateCamera(*Seitenverhältnis, znear , zfar , field of view****);***

Für das Seitenverhältnis haben wir einfach Raumbreite / Raumhöhe eingesetzt. **znear** und **zfar** bestimmen in dieser Reihenfolge, ab welcher Nähe und ab welcher Entfernung Objekte nicht mehr gerendert werden. Unter **field of view** stellt man das Sichtfeld der Kamera ein (standardmäßig zwischen 45° und 60°). Mit diesem Wert lassen sich erfahrungsgemäß schon beim **GM** nette Effekte erzielen.

In der zweiten Zeile wird die Kamera dem ***Viewport*** zugewiesen, den wir anfangs in ***obj_engine*** definiert haben.

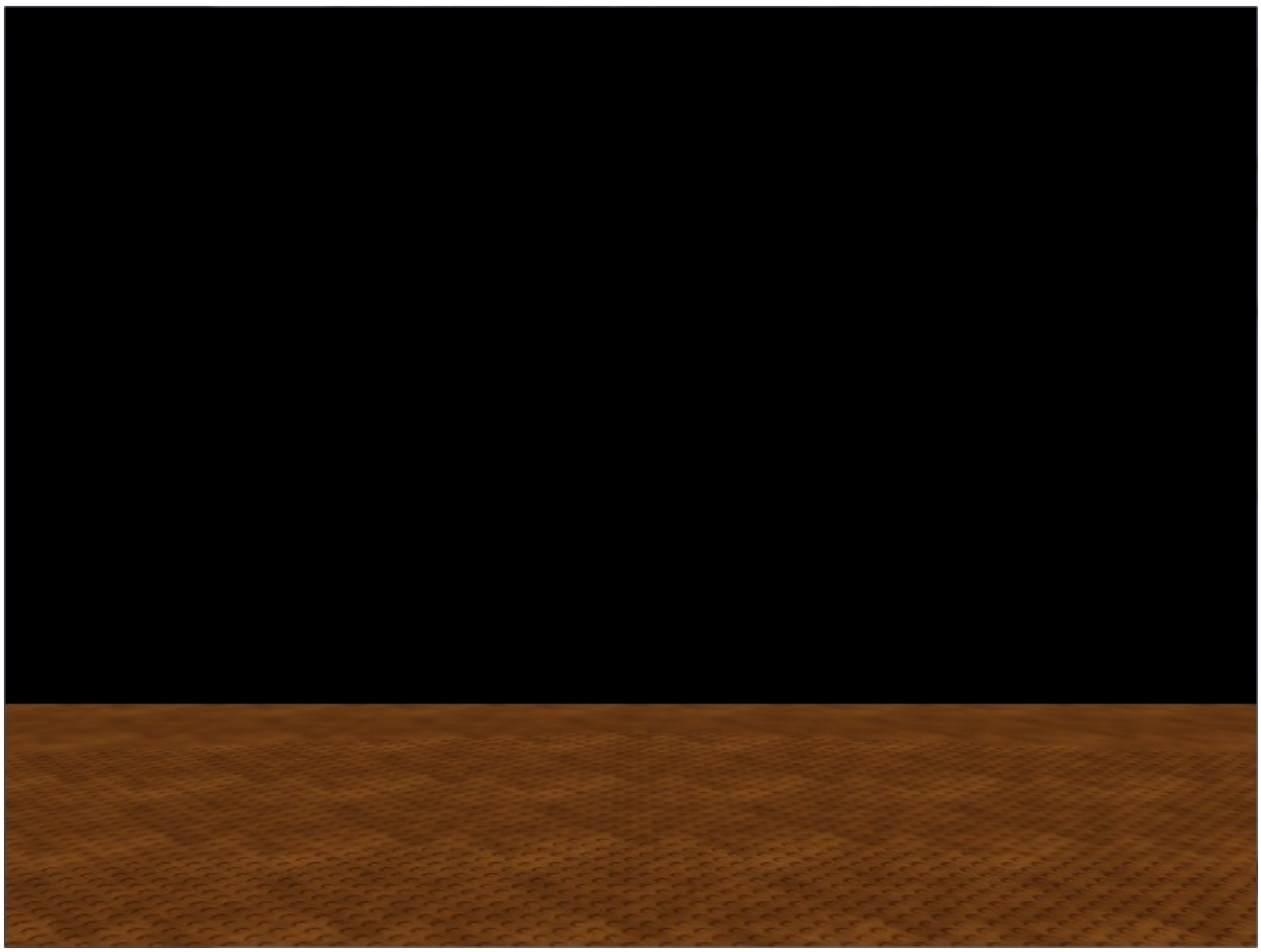
Die Kameraposition wird durch Einfügen dieser Zeile auf festgelegt:

```
SetCameraPosition(cam_id, 0, 0, 150);
```

Damit sitzt die Kamera praktisch in der linken oberen Ecke unseres Rooms in einer Höhe von 150.

So, dann lasst uns mal einen ersten Blick auf unsere Szene werfen! Wenn ihr alles richtig gemacht habt, solltet ihr im unteren Bereich des Fensters nun den Boden sehen. Je nachdem, welche Farbe ihr anfangs in ***obj_engine*** für das ***Ambient Light*** festgelegt habt, ist die Szene mehr oder weniger gut sichtbar.

So sollte es in etwa aussehen:



Ich habe für diesen Screenshot das Umgebungslicht in obj_engine geändert, damit man deutlich sieht, was wir bisher geschafft haben. Ihr könnt auch gern selbst ein wenig experimentieren:

```
SetAmbientLight(c_orange);
```

Im nächsten Kapitel werden wir für bessere Sichtverhältnisse sorgen.

>>> PART 5: INS RECHTE LICHT GERÜCKT

Lichtquellen sind für Spiele unglaublich wichtig. Es lassen sich wunderschöne Akzente setzen und zusammen mit der richtigen akustischen Untermalung lässt sich die gesamte Atmosphäre bestimmen.

In unserem Fall brauchen wir aber nur ein Licht, welches die Szene aufhellt, damit wir sowohl den Boden, als auch den Roboter gut sehen können. Von dieser Lichtquelle hängt später auch die Berechnung des Schattens ab.

Also erstellt ein neues Objekt und nennt es ***obj_light***. Platziert es gleich wieder im Room neben dem zuvor erstellten Objekt ***obj_camera***.

Da wir uns ausnahmsweise mit nur einem Licht zufrieden geben, reicht folgender Code im Create Event:

```
light_id = CreateLight();
SetLightType(light_id, LT_POINT);
SetLightPosition(light_id, 150, 0, 200);
```

Ihr habt sicher inzwischen kapiert, wie **GMOgre** funktioniert und dass das Erzeugen von Entities, Kameras und Lichtquellen im Grunde immer dem gleichen Muster folgt.

Richtig, in Zeile 1 wird die Lichtquelle erzeugt und erhält wie gewohnt eine **ID**.

In Zeile 2 wird festgelegt, welche Art von Licht benutzt werden soll. Zur Verfügung stehen uns diese 3 Arten: ***Point Light***, ***Spotlight*** und ***Directional Light***. Je nach Typ lassen sich mit verschiedenen weiteren Funktionen diverse Einstellungen vornehmen. Da ich mich zugunsten der Einfachheit für ein ***Point Light*** entschieden habe, genügt es vorerst, die Position festzulegen und das geschieht - wie ihr euch sicher denken könnt - in der dritten Zeile.

Dann lasst euren **GM** die Szene nochmal starten. Jetzt sollte man deutlich mehr erkennen.

Da wir die Kamera später ohnehin nochmal umschreiben müssen, können wir jetzt auch dafür sorgen, dass ihr euch mithilfe der Maus in der Szene umschauen könnt. Wenn ihr das nicht wollt, könnt ihr auch gern die nächste Seite auslassen und direkt zu Kapitel 6 springen.

>>> PART 5 B: MOUSELOOK

Den Mouselook kennt ihr sicher aus vielen Spielen, vorrangig First-Person-Shooter. Die Blickrichtung der Kamera wird mithilfe der Maus geändert. Die Umsetzung ist in GMOgre kinderleicht.

Schließt ***obj_light*** und öffnet stattdessen wieder ***obj_camera***.

Fügt im Create Event diese Zeile an:

```
EnableMouseLook(cam_id, true);
```

Und diese Zeile muss ins Step Event:

```
UpdateMouseLook();
```

Mehr ist nicht nötig. Ihr könnt jetzt gern wieder einen Testlauf starten.

Dann können wir nun zum Hauptteil kommen: dem Animieren und Steuern des Roboters! Keine Angst, es klingt zwar umfangreich, aber so schwer ist es gar nicht. **GMOgre** bietet aktuell über 550 Funktionen und die meisten davon sind so ausgelegt, dass sie uns sehr viel Arbeit abnehmen.

Überzeugt euch selbst. Auf der nächsten Seite geht's weiter.

>>> PART 6: EIN ROBOTER LERNT LAUFEN

GMOgre unterstützt ***Skeletal Animations***. Hierfür erhält ein 3D Model ein unsichtbares Skelett, bestehend aus sogenannten ***Bones*** (Knochen). Dieser Prozess wird direkt in der Modellierungssoftware eurer Wahl durchgeführt, nicht in **GMOgre**!

Um ein 3D Modell zu animieren, können diese ***Bones*** transformiert werden. Fertige Animationen werden direkt im 3D Modell gespeichert und können von **GMOgre** abgerufen und ausgeführt werden.

Wenn ich die bis dato undokumentierten Funktionen richtig verstanden habe, lassen sich sogar einzelne ***Bones*** ansprechen und transformieren.

Erstellt nun wieder ein neues Objekt, nennt es ***obj_robot*** und platziert es wie gewohnt im Room. Diesmal solltet ihr aber einen gewissen Abstand zum Rand einhalten, da die Kamera von der linken oberen Ecke aus den Roboter fixieren soll. Eine Position von etwa X 200 und Y 200 ist ganz gut.

Los geht's mit folgendem Codeblock im Create Event:

```
ent_id = CreateEntity("Robot.mesh");
EnableEntityCastShadows(ent_id, true);
node_id = CreateRootChildSceneNode(x,y,0);
AttachEntityToSceneNode(ent_id, node_id);
```

Das meiste kennt ihr ja: ***Mesh*** laden – ***Scene Node*** erzeugen – ***Mesh*** an ***Scene Node*** heften.

Zeile 2 aktiviert die Schattenberechnung für dieses Modell. Mehr ist dazu nicht nötig!

Ich nehm's mal vorweg: Die Blickrichtung des eben geladenen 3D Modells entspricht leider nicht der **direction** von obj_robot. Es ist sozusagen um 90° im Uhrzeigersinn gedreht. Das ist keine Seltenheit bei 3D Modellen, stellt aber auch absolut kein Problem dar.

Da wir die „direction" von obj_robot später brauchen, um die Laufrichtung festzulegen, drehen wir jetzt ganz einfach den zuständigen ***Scene Node*** um 90° gegen den Uhrzeigersinn:

```
YawSceneNode(node_id, 90);
```

Mit dieser Funktion wird der Wert '90' zur aktuellen Ausrichtung addiert.

Die Vorbereitungen für den Roboter sind abgeschlossen. Jetzt nehmen wir seine Animation in Angriff.

Da 3D Modelle meist über mehrere Einzelanimationen verfügen (z.B. schleichen, rennen, zielen, etc.), muss festgelegt werden, welche Animation wir abspielen wollen. Dazu muss man die Bezeichnungen der einzelnen Animationen kennen.
In unserem Fall kenne ich diese Namen aus dem offiziellen **GMOgre** Beispiel Advanced Tutorial 1.

Fügt bitte folgenden Codeblock im Create Event an:

```
anim_state_id = GetEntityAnimationState(ent_id, "Idle");
EnableAnimationStateLoop(anim_state_id, true);
EnableAnimationState(anim_state_id, true);
```

In der ersten Zeile wird die Animation „Idle“ aktiviert. Hierbei handelt es sich um die Stehanimation. In Zeile 2 wird die Animation als Endlosschleife eingestellt und Zeile 3 startet die Animation letztendlich.

Unser Roboter verfügt glücklicherweise auch schon über eine Gehanimation namens „Walk“.

Erwartet aber nichts annähernd professionelles! 

Das Create Event können wir nun abschließen und uns voll und ganz dem Step Event widmen.

Da wir den Roboter mit den Cursortasten steuern wollen und seine 2 Animationen davon abhängig sind, schauen wir uns erstmal an, was überhaupt passieren soll:

1. ***Wenn <Cursortaste links oder rechts> gehalten wird, dreht sich der Roboter.***
2. ***Wenn <Cursortaste rauf> gedrückt wird (pressed), schaltet die Animation auf "Walk" und setzt speed = 1.***
3. ***Wenn <Cursortaste rauf> gehalten wird, läuft die Animation "Walk" (und speed bleibt auf '1').***
4. ***Wenn <Cursortaste rauf> losgelassen wird (released), schaltet die Animation auf "Idle" und setzt speed = 0.***
5. ***Wenn <Cursortaste rauf> nicht betätigt wird, läuft die Animation "Idle".***

Bevor's losgeht: Die Animation "Idle" läuft im Verhältnis zu "Walk" viel zu schnell. Somit müssen beide Animationen in unterschiedlichen Geschwindigkeiten abgespielt werden. Wenn wir beide mit der gleichen Geschwindigkeit abspielen könnten, bräuchten wir weniger Codezeilen.

Beginnen wir also mit der Umsetzung des ersten Punktes: dem Drehen des Roboters mittels der <*Cursortasten links und rechts*>.

Schreibt oder kopiert folgenden Code ins Step Event:

```
if(keyboard_check(vk_left)) {direction += 2; YawSceneNode(node_id, 2);}
if(keyboard_check(vk_right)) {direction -= 2; YawSceneNode(node_id, -2);}
```

Eine Erklärung brauchen diese 2 Zeilen nicht. Über grundlegende **GML** Kenntnisse solltet ihr verfügen und **YawSceneNode()** haben wir bereits im Create Event benutzt.

Kommen wir also zu Punkt 2 und leiten die Gehanimation ein. Fügt folgende Zeilen dem Step Event hinzu:

```
if(keyboard_check_pressed(vk_up)) {
    anim_state_id = GetEntityAnimationState(ent_id, "Walk");
    EnableAnimationStateLoop(anim_state_id, true);
    EnableAnimationState(anim_state_id, true);
    speed = 1;
    }
```

Den Großteil kennt ihr ja bereits aus dem Create Event. Die Animation „Walk“ löst „Idle“ ab und speed wird auf '1' gesetzt, sobald der Tastendruck der <*Cursortaste rauf*> beginnt. Somit setzt sich der Roboter in Bewegung. Viel mehr gibt's hier nicht zu sagen.

In Punkt 3 geht es darum, dass der Roboter die Animation „Walk“ endlos durchläuft und speed unverändert auf '1' bleibt, solange die <*Cursortaste rauf*> gehalten wird. Dafür ist folgender Codeblock zuständig:

```
if(keyboard_check(vk_up)) {
    AddAnimationStateTime(anim_state_id, .03);
    SetSceneNodePosition(node_id, x, y, 0);
    }
```

Der Wert '0.03' in der Funktion **AddAnimationStateTime()** bestimmt die Animationsgeschwindigkeit.
Die zweite Zeile überträgt die 2D-Koordinaten (x & y) von ***obj_robot*** auf den ***Scene Node***, dem das ***Mesh*** zugeordnet ist. Das ist nötig, wenn man die internen GM Variablen (speed, direction, etc.) benutzen will, um den Roboter zu steuern.

Schauen wir uns nun den Code zu Punkt 4 an. Er beinhaltet annähernd das Gleiche. Diesmal wird „Walk“ allerdings durch „Idle“ abgelöst und speed auf '0' gesetzt, um den Roboter wieder anzuhalten, wenn die <*Cursortaste rauf*> losgelassen wird.

Fügt folgende Zeilen einfach wieder im Step Event an:

```
if(keyboard_check_released(vk_up)) {
    anim_state_id = GetEntityAnimationState(ent_id, "Idle");
    EnableAnimationStateLoop(anim_state_id, true);
    EnableAnimationState(anim_state_id, true);
    speed = 0;
    }
```

Punkt 5 ist letztendlich für das Abspielen der Stehanimation „Idle" zuständig und soll nur dann ausgeführt werden, wenn die <*Cursortaste rauf*> nicht betätigt wird:

```
if(!keyboard_check(vk_up)) {AddAnimationStateTime(anim_state_id, .005);}
```

Wie ihr seht, beträgt die Animationsgeschwindigkeit hier '0.005' und könnte meiner Meinung nach sogar noch einen Tick langsamer sein. Aber das überlasse ich euch...

Erwähnenswert finde ich die Tatsache, dass die beiden Animationen "Idle" und "Walk" fließend ineinander übergehen. Das kann man hier sehr gut beobachten, da in unserem Fall eine Animation nicht "resettet" wird, bevor die andere aufgerufen wird.

Bevor wir nun aber den finalen Testlauf starten, werden wir die Kamera noch so einrichten, dass sie immer den Roboter anvisiert. Ihr könnt ***obj_robot*** jetzt schließen, damit sind wir fertig.

Öffnet also ein letztes Mal ***obj_camera***.

Löscht als erstes das Step Event, das wird nicht mehr benötigt. Ihr erinnert euch: das wurde nur für den Mouselook gebraucht.

Wer Kapitel 5b übersprungen hat, wird natürlich kein Step Event vorfinden.

Im Create Event löscht ihr bitte folgende Zeile:

```
EnableMouseLook(cam_id, true);
```

Nun ist die Maussteuerung wieder entfernt und wir können stattdessen den Roboter als Ziel einstellen.

Hierzu muss folgende Zeile ins Create Event:

```
EnableCameraAutoTrackSceneNode(cam_id, true, obj_robot.node_id);
```

Da ***obj_camera*** nun bereits im Create Event versucht, den Roboter, bzw. ***obj_robot*** zu fokussieren, wird uns der **GM** mit einer Fehlermeldung begrüßen, wenn wir die Szene starten.

Das liegt daran, dass wir ***obj_robot*** erst im Room platziert haben, als ***obj_camera*** bereits positioniert war. Damit die Kamera auf ein existierendes Objekt gerichtet werden kann, müsste ***obj_robot*** platziert werden, bevor ***obj_camera*** darin vorkommt.

Die Lösung ist also ganz einfach. Löscht obj_camera aus dem Room und platziert es gleich wieder an derselben Stelle. Dann sollte es klappen.

Alternativ könnte man natürlich auch obj_camera dahingehend anpassen, dass nicht gleich im Create Event versucht wird, sich auf den Roboter auszurichten.
Aber ich habe mich für die einfachere Lösung entschieden.

Im Großen und Ganzen sind wir jetzt fertig! Wer noch Lust hat, kann im folgenden Kapitel ein paar Übungsaufgaben lösen...

>>> PART 7: HOMEWORK!

Hier gibt es noch ein paar Aufgaben, die ihr eigenständig lösen könnt, um einfach noch wenig mit **GMOgre** zu experimentieren. Natürlich nur, wenn ihr wollt...

Hier findet ihr Hilfe: Tutorials und Funktionsliste

1. **Verändert die Position der Kamera und richtet sie auf einen festen Punkt eurer Wahl aus.**
2. **Vergrößert, verkleinert, streckt oder staucht den Roboter (bzw. seinen Scene Node):**
   SetSceneNodeScale(id, x_scale, y_scale, z_scale);
3. **Die Sterbeanimation des Roboters nennt sich "Die". Probiert sie mal aus.**
4. **Probiert verschiedene Farben für das Umgebungslicht (Ambient Light) und die Lichtquelle. Probiert auch die anderen beiden Lichtarten ("Point Light", "Spotlight" und "Directional Light") und experimentiert mit ihren unterschiedlichen Einstellungsmöglichkeiten.**
   Hier findet ihr die benötigten Funktionen...
5. **Alles weitere bleibt eurer Kreativität überlassen. =)**

>>> DAS WÄR'S DANN...

Ich hoffe, die ganze Arbeit war nicht völlig umsonst und ihr hattet trotz des Umfangs ein wenig Spaß beim Rumprobieren. Und wenn auch nur ein einziger User was daraus gelernt hat und sich nun für **GMOgre** bereit fühlt, hat sich die Arbeit schon gelohnt. =)

Falls ihr Probleme bei den zusätzlichen Aufgaben haben solltet, könnt ihr mich über www.gm-d.de kontaktieren. Schreibt einfach im Thread oder schickt mir eine PM.

Zum Abschluss möchte ich mich noch bei den Leuten bedanken, die mich beim Entwickeln dieses Tutorials unterstützt haben. Vor allem muss hier copyboy erwähnt werden, der die gmk-Dateien in gm6-Dateien umgewandelt hat. Außerdem ist ihm dabei aufgefallen, dass **GMOgre** eigentlich noch gar nicht GM6-kompatibel ist, da in den Scripts mehrfach ein Befehl vorkommt, der erst seit GM7 existiert. Ohne jetzt zu weit ins Detail zu gehen - nach ein paar kleineren Anpassungen konnte er das Problem lösen und die hier bereitgestellten GM6 Dateien sollten somit einwandfrei funktionieren. Falls nicht, meldet euch.

Außerdem danke ich Dur'Rean für die investierte Zeit zum Testen der GM6 Files mitten in der Nacht. Und ein abschließendes Danke an F4LL0UT für die Unterstützung bei der Wortfindung, wenn ich wieder mal zum Opfer meines Hirnmatsches wurde.

Danke für's Lesen und viel Spaß weiterhin mit **GMOgre**,

mauge