

Inhaltsverzeichnis

<u>Vorwort.....</u>	<u>2</u>
<u>Dieses seltsamen 3 Buchstaben.....</u>	<u>2</u>
<u>Installation.....</u>	<u>3</u>
<u>Das Archiv.....</u>	<u>3</u>
<u>Varianten der Bibliothek.....</u>	<u>3</u>
<u>Die Mapskripte.....</u>	<u>3</u>
<u>Schnelles Testen.....</u>	<u>4</u>
<u>DEBUG aktivieren.....</u>	<u>4</u>
<u>Aufträge erzeugen.....</u>	<u>5</u>
<u>Aufträge im Questassistenten.....</u>	<u>5</u>
<u>Aufträge im Skript.....</u>	<u>5</u>
<u>Lokalisierung von Aufträgen.....</u>	<u>6</u>
<u>Angaben für einen Auftrag.....</u>	<u>6</u>
<u>Interaktive Objekte.....</u>	<u>7</u>
<u>Nichtspieler-Charaktere.....</u>	<u>8</u>
<u>Funktionen.....</u>	<u>8</u>
<u>Behavior.....</u>	<u>8</u>
<u>Mission-Briefings und Cutscenes.....</u>	<u>9</u>
<u>Grundlagen für Dialoge und Missionsbeschreibungen.....</u>	<u>9</u>
<u>Weggablungen in Dialogen.....</u>	<u>10</u>
<u>Grundlagen für Kameraflüge.....</u>	<u>11</u>
<u>Relative Positionsangaben.....</u>	<u>13</u>

Vorwort

Symfonia, oder auch QSB++, hat es sich zum Ziel gesetzt, das Leben des Mappers zu erleichtern. Viele bekannte und geschätzte Standardfunktionen befinden sich bereits in der QSB und müssen nicht mehr in die eigenen Skripte kopiert werden. Um in den Genuss der Vorzüge dieses Rahmenwerkes (englisch: Framework¹) zu kommen, muss man sich an einige *wenige* Regeln halten, denn ohne Regeln ist das Leben ein Chaos. Das ist überhaupt nicht kompliziert und macht sogar Spaß! Sobald die Grundregeln beachtet werden, geht Dir das Mappen so leicht von der Hand, wie eine Scheibe Brot zu schmieren.

Es ist vollkommen unnötig seine Zeit mit Gedanken wie „jetzt ist alles anders“ oder „ich bin kein Genie“ zu verschwenden. Bei der Programmierung der QSB wurde Wert darauf gelegt, altbewährte Vorgehensweisen beizubehalten und zu erweitern. Zusätzlich sind einige Dinge hinzugekommen, die unkompliziert eingesetzt werden können um aus den eigenen Maps etwas besonderes zu machen. Dabei sind Kreativität und Individualität keine Grenzen gesetzt! Bald wirst Du in den Genuss der einzigartigen Möglichkeiten kommen, die Dir die QSB anbieten kann.

Doch zuvor stehen einige Erklärungen aus. Niemand erwartet von Dir sofort alles zu verstehen. Und genau das ist das großartige an Symfonia: es gibt nichts zu verstehen! Die gesamte Handhabung ist intuitiv designt. Und wer einmal den Dreh raus hat, wird nie wieder etwas anderes wollen. Aber genug der langen Vorrede: Jetzt werde ich die wichtigsten Dinge erklären, die Du benötigst um Symfonia zu nutzen. Ich werde auf die speziellen Funktionalitäten eingehen, angefangen bei der Installation und den ersten Schritten zur Verwendung bis hin zu einigen speziellen Features.

Dieses seltsamen 3 Buchstaben

Symfonia führt ein neues Konzept ein. Es gibt Funktionen, die für den Gebrauch durch den Benutzer gedacht sind. Sie bieten komfortable Fehlererkennung und weisen explizit auf falsche Benutzung hin, indem sie eine Fehlermeldung ausgeben und sich danach selbst beenden. Auf diese Weise werden Probleme, wie z.B. Abstürze, verhindert.

Diese Funktionen sind alle mit dem Präfix API² gruppiert. Sie können aber mehr als nur Fehler erkennen. Sie erkennen zudem auch, ob sie aus der falschen Skriptumgebung aufgerufen werden. Und falls möglich, rufen sie sich selbstständig in der richtigen Umgebung auf! Sollte dies nicht möglich sein, werfen sie einen Fehler.

1 Framework: ist ein Programmiergerüst, welches vereinfacht und dem Anwender Vorteile gegenüber der nativen Möglichkeiten einer Umgebung bietet, in der er sich bewegt.

2 API: Application Programmers Interface (Schnittstelle des Programmierers) bietet dem Anwender Dienste eines geschlossenen Systems nach außen an.

Installation

Das Archiv

Die QSB wird als ZIP-Archiv ausgeliefert. Wenn du es bis hier her geschafft hast, weißt du auch, wie man ein solches Archiv entpackt und die Anleitungen findet. Und viel mehr gibt es dazu auch nicht zu wissen. Du kannst den Ordner an einem beliebigen Ort auf dem Rechner auspacken. Darin enthalten sind alle Dateien, die du benötigst, um die QSB zu verwenden: Eine große Bibliotheksdatei und zwei kleine leere Skripte. Alle drei sind LUA-Dateien.

Varianten der Bibliothek

Sobald Du eine neue Map erstellt hast, muss die QSB in die Map *importiert* werden. Dazu öffnest Du den Mappeditor und wählst *Datei > Questbibliothek importieren*. Nun ist Konzentration gefragt! Navigiere in den Ordner, den Du zuvor entpackt hast, und wähle die Bibliothek aus. Die Bibliothek ist die riesengroße Textdatei mit dem Namen „questsystembehavior“.

Ein Release kommt mit *zwei* Versionen der QSB, die sich nur in einem Punkt unterscheiden: der Dateigröße. Alles andere ist von der ersten bis zur letzten Funktion identisch!

Sobald Du in den Release-Ordner gegangen bist, wirst Du zwei verschiedene Dateien vorfinden: `questsystembehavior.lua` und `questsystembehavior_min.lua`.

Die Datei `questsystembehavior.lua` ist die unkomprimierte QSB. Das heißt, dass sie noch alle Leerzeichen und Kommentare enthält. Sie eignet sich zum stöbern, sollte Dir die Dokumentation nicht ausreichen um Deine Fragen zu klären. Sie ist allerdings sehr groß, etwa 1 Megabyte. Aus diesem Grund liegt sie auch komprimiert vor.

Die Datei `questsystembehavior_min.lua` wurde auf ein Minimum reduziert, sodass sie etwa um die Hälfte weniger Speicherplatz benötigt. Es wurden unnötige Leerzeichen und Kommentare entfernt, genauso wie Namen von Variablen gekürzt. Alles das macht den Code für den Menschen sehr unleserlich. Aber der benötigte Speicherplatz hat sich um etwa die Hälfte reduziert.

Welche der Dateien Du importierst, ist vollkommen Deiner Einschätzung überlassen. Beide Dateien bieten Dir den exakt gleichen Funktionsumfang. Dir entstehen also keine Nachteile, wenn du die minimierte Bibliothek anstelle der traditionellen benutzt!

Die Mapskripte

Ein kleiner Nachteil der QSB ist, dass sie nicht mit den Mapskripten kompatibel ist, die der Mappeditor ausspuckt. Darum müssen die mitgelieferten Dateien verwendet werden. Im Release-Ordner befinden sich die Textdateien `mapscript.lua` und `localmapscript.lua`. In diesen Dateien erstellst Du den Ablauf deiner Map. Anschließend müssen diese Dateien in deine Map importiert werden, damit sie gespielt werden kann!

Um dies zu tun, musst Du den Expertenmodus aktivieren. Im Expertenmodus können die Kartenskripte importiert und exportiert werden. Bitte lasse Dich nicht von der Bezeichnung „Expertenmodus“ oder dem Warnhinweis abschrecken! Klar besteht die Möglichkeit viel falsch zu machen, aber dafür gibt es ja die QSB. Mit ihr reduziert sich das Risiko auf ein Minimum! Sobald die Kartenskripte importiert sind, kann es losgehen!

Schnelles Testen

Nach *jeder* Änderung im Skript müsste man eigentlich das Spiel verlassen, den Editor starten und das Skript neu importieren. Das ist auf Dauer anstrengend und kostet viel Zeit! Daher bietet es sich für die Entwicklung an, das globale und lokale Skript auszulagern. Dazu wird ein Hilfsskript benötigt, das anstelle der eigentlichen Skripte in die Map importiert wird.

```
-----  
local PathToProject = "C:/MyProjects/MyMap";  
if not GUI then  
    Script.Load(PathToProject .. "/mapscript.lua");  
else  
    Script.Load(PathToProject .. "/localmapscript.lua");  
end  
-----
```

Dieses Skript lädt automatisch die entsprechenden Dateien, da es zuerst prüft, ob es sich in der globalen oder lokalen Skriptumgebung befindet. Dadurch kannst Du während der Entwicklung viel Zeit sparen. Passe einfach den Pfad **PathToProject** an Deine Bedürfnisse an. Beachte: vor der Veröffentlichung musst Du aber die eigentlichen "richtigen" Skripte in die Map importieren.

Ein Beispiel für einen angepassten Pfad könnte wie folgt aussehen:

```
-----  
local PathToProject = "E:/Maps/Scripts/mf02_gutkirschenessenmitmanfred";  
-----
```

Solange der Pfad stimmt und alle Dateien vorhanden sind, wird es funktionieren.

DEBUG aktivieren

Bevor Du in den kreativen Schöpfungsprozess einsteigst, möchtest Du vielleicht den DEBUG³ aktivieren, um Fehler schneller entdecken zu können. Der Debug lässt sich als Aufruf in der **Mission_FirstMapAction**, einer Funktion im *globalen* Skript, aktivieren. Die Funktion **API.ActivateDebugMode** ist dafür zuständig. Sie muss nach dem Aufruf von **InitKnightTitleTables** eingefügt werden.

Du kannst diese Einstellung verwenden:

```
-----  
API.ActivateDebugMode(true, false, true, true);  
-----
```

Dies aktiviert alle Checks und die Konsole, aber nicht die Live-Verfolgung des Queststatus. Mehr Informationen zum Debug, findest du in der Dokumentation. Alternativ kann der Debug auch mit einem Behavior aktiviert werden, **Reward_DEBUG**.

³ Debug: Kommt aus der Sprache der Software-Entwickler. Fehler werden aus Bugs (Käfer) bezeichnet. Fehler beseitigen bedeutet also Ungeziefer auszumerzen, das einem das Leben schwer macht. Begebe dich auf Kakerlackenjagd und lösche die Fehler alle aus!

Diese Funktion kann natürlich auch über ihren alten Namen, `ActivateDebugMode`, genutzt werden, wie jede andere API-Funktion auch.

Aufträge erzeugen

Aufträge im Questassistenten

Die QSB unterstützt natürlich wie gewohnt das Erzeugen von Aufträgen im Auftragsassistenten des Mapeditors. Allerdings gibt es keine Vorzüge, wie z.B. das weglassen von Parametern oder das einfache kopieren von Aufträgen.

Für eine Anleitung zum Auftragsassistenten des Editors konsultiere bitte das SEED Handbuch.

Aufträge im Skript

Aufträge, oder auch Quests⁴, im Skript zu erzeugen, ist keine wirklich neue Idee. Eine Möglichkeit Aufträge zu generieren bot z.B. schon `MachQuest` an, dessen Handhabung jedoch nicht wirklich zufriedenstellend war. Neu ist allerdings, Parameter von Behavior und Eigenschaften des Quests weglassen zu können. Dies geht einher mit einer neuen Schreibweise.

Ein Beispiel für die neue Schreibweise:

```
local QuestName = API.CreateQuest {  
    Name          = "VisitCloister",  
    Suggestion    = "Ich sollte die Mönchen des Klosters besuchen!",  
    Success       = "Guten Tag, werter Herr!",  
  
    Goal_DiscoverPlayer(4),  
    Reward_Diplomacy(1, 4, "EstablishedContact"),  
    Trigger_OnQuestSuccess("SomeOtherQuest", 8),  
}
```

Jeder Auftrag benötigt wenigstens einen Namen. Wird kein Goal angegeben, wird automatisch `Goal_InstantSuccess` hinzugefügt. Fehlt der Trigger, wird `Trigger_AlwaysActive` den Behavior hinzugefügt. Auf diese Weise können Aufträge bei Bedarf kurz gehalten werden. Auf diese Weise wird der Platzverbrauch reduziert. Du gewinnst dadurch eine größere Übersicht über Dein Skript – was immer einen Vorteil bedeutet.

Andererseits, falls Behavior und/oder Felder wie z.B. Success angegeben sind, kann der Name weggelassen werden. In diesem Fall wird ein automatischer Bezeichner als Name gewählt. Auf jeden Fall wird der Name des Quests von `API.CreateQuest` zurückgegeben und kann auch so auf andere Behavior übertragen werden.

Ein Beispiel:

⁴ Quest: Ein Auftrag, der durch den Spieler erfüllt werden muss, meistens um die Handlung voranzutreiben oder Vorteile für den Spielverlauf zu erhalten.

```

local QuestName = API.CreateQuest {
    Suggestion    = "Ich sollte die Mönchen des Klosters besuchen!",
    Success       = "Guten Tag, werter Herr!",

    Goal_DiscoverPlayer(4),
    Reward_Diplomacy(1, 4, "EstablishedContact"),
    Trigger_OnQuestSuccess("SomeOtherQuest", 8),
}

local NextQuest = API.CreateQuest {
    Suggestion    = "Ich bin der Abt des Klosters!",
    Sender        = 4,
    Trigger_OnQuestSuccess(QuestName, 10)
}

```

Lokalisierung von Aufträgen

Die QSB sieht es vor, dass alle angezeigten Texte in Deutsch und Englisch angegeben werden können. Es wird dann automatisch der deutsche Text angezeigt, wenn es sich um die deutsche Version des Spiels handelt. Andernfalls wird der englische Text genutzt. Diese Funktionalität steht natürlich auch für Aufträge zur Verfügung.

Ein Beispiel für einen lokalisierten Auftragstext:

```

Suggestion = {
    de = "Ich sollte die Mönchen des Klosters besuchen!",
    en = "I should visit the monks of the monastery!",
},

```

Angaben für einen Auftrag

Aufträge haben viele mögliche Angaben. Die Eigenschaften eines Auftrags werden als Felder bezeichnet. Eigenschaften sind z.B. der Name des Auftrags. Du kannst die wichtigsten Eigenschaften eines Auftrags der nachfolgenden Tabelle entnehmen.

Eigenschaft	Beschreibung
Name	Über den Namen werden die Quests verwaltet
Suggestion	Der Text des Sprechers zu Beginn des Auftrages
Success	Der Text des Sprechers bei erfolgreichem Abschluss
Failure	Der Text des Sprechers im Falle eines Fehlschlages
Description	Die Beschreibung für benutzerdefinierte Auftragstypen
Visible	Erzwingt die Sichtbarkeit, wenn Suggestion nicht angegeben ist.
EndMessage	Erzwingt die Sichtbarkeit der Endnachricht.
Sender	ID des Auftraggebers
Receiver	ID des Auftragnehmers
Time	Zeit bis zum automatischen Fehlschlag/Erfolg

Die Behavior eines Quests werden als Funktionsaufrufe nach den Eigenschaften des Auftrages angegeben. Dabei ist zu beachten, dass jeder sichtbare Auftrag nur ein Goal haben kann. Auslöser, oder auch Trigger, können mehrere vorhanden sein. Sie müssen alle wenigstens einmal ausgelöst haben, damit der Auftrag startet.

Interaktive Objekte

Mit den erweiterten interaktiven Objekten kannst Du neue Elemente in Deine Maps bringen. Helden können z.B. versteckte Schalter finden um geheime Gänge zu öffnen. Der große Vorteil ist, dass sich so ziemlich jedes Entity in ein interaktives Objekt umwandeln lässt. Zum Beispiel könnte einem Schloss der richtige Schlüssel fehlen.

```
CreateObject {
    Name          = "trigger1",
    Title         = "Versteckter Schalter",
    ConditionUnfulfilled = "Dir fehlt der benötigte Schlüssel!",
    Condition      = function(_Data)
        return HasFoundKey == true;
    end,
    Callback      = function(_Data)
    end,
};
```

Diese Konfiguration ist ein Beispiel für ein versteckten Schalter, der aber erst aktiviert werden kann, wenn die Variable **HasFoundKey** gesetzt ist. Vorher bekommt der Spieler eine Meldung, dass er einen Schlüssel benötigt.

Natürlich können interaktive Objekte auch mit Aktivierungskosten versehen werden.

```
CreateObject {
    Name  = "fire1",
    Title = "Signalfeuer",
    Costs = {Goods.G_Wood, 30, Goods.G_Honeycomb, 5},
};
```

Ein interaktives Objekt kann auch eine Belohnung beinhalten.

```
CreateObject {
    Name    = "chest1",
    Title   = "Schatztruhe",
    Reward  = {Goods.G_Gold, 3000},
};
```

Die verschiedenen Eigenschaften des Objektes können beliebig gesetzt oder weggelassen werden. Schreibt man sie nicht, werden Standardwerte benutzt. Für eine Liste aller möglichen Eigenschaften, sieh Dir bitte die Dokumentation an.

Nichtspieler-Charaktere

Funktionen

Nichtspieler-Charaktere (NPC) sind Siedler, die von einem Helden explizit angesprochen werden müssen, damit sie etwas auslösen. Du kannst sie Dir als lebende interaktive Objekte vorstellen. Ein NPC ist dadurch zu erkennen, dass er auf der Spielwelt glitzert.

```
API.NpcCompose {  
  Name      = "hakim",  
  Callback = function(_Npc, _Hero)  
    -- Hier kann was passieren  
  end,  
}
```

Dies ist ein Beispiel für einen einfachen NPC.

NPCs haben noch weitere Funktionalitäten. Du kannst einen Helden vorgeben, der den NPC ansprechen muss. Diese Option ist allerdings nur sinnvoll, wenn der Spieler in Deiner Map mehr als einen Helden gleichzeitig steuert. Für die meisten Maps ist diese Angabe daher unnötig.

```
API.NpcCompose {  
  Name      = "hakim",  
  Hero      = "marcus",  
  WrongHeroMessage = "Ich spreche nicht mit dir!",  
  Callback  = function(_Npc, _Hero)  
    -- Hier kann was passieren  
  end,  
}
```

Wird der NPC nun mit einem Helden angesprochen, dessen Skriptname nicht **marcus** ist, wird das Callback nicht ausgelöst und der Text von **WrongHeroMessage** als Nachricht am linken Rand auf dem Bildschirm ausgegeben.

Behavior

Du kannst Deine NPCs auch an Quests anbinden, anstelle sie entkoppelt zu verwenden, indem Du die entsprechenden Behavior verwendest.

```
Goal_NPC("hakim", "hero")
```

Als erstes wird der Name des NPC angegeben. Als zweites kann optional der Name des Helden folgen, der den NPC ansprechen muss. Je nach dem, ob ein Held vorgegeben ist oder nicht, wird sich das Auftragsfenster anpassen.

Mission-Briefings und Cutscenes

DIE SIEDLER - Aufstieg eines Königreichs bietet nicht von Haus aus an, Missions-Briefings oder Kameraflüge zu erstellen. Das hat uns der Vorgänger unseres Spiel voraus! Allerdings bietet die Engine⁵ des Spiels einen entsprechenden Kameramodus an, der für die Thronsaalszenen in der Kampagne genutzt wird. Das so genannte Briefing System kann diesen Kameramodus zweckentfremden und es ermöglichen, Missions-Briefings zu erzeugen. Dabei unterteilt es in zwei verschiedene Arten von Briefings. Das Missions- oder Dialog-Briefing für Beschreibungen und der Kameraflug oder Cutscene für szenerische Darstellung.

Neu ist jetzt, das Briefings (bzw. Cutscenes) an einen Auftrag „angehängen“ werden können. Dies passiert mit den Behavior **Reward_Briefing** und **Reprisal_Briefing**. Über den Auslöser **Trigger_Briefing** kann anschließend ein *anderer* Auftrag gestartet werden, *sobald* das Briefing oder der Kameraflug *vollständig* durchgelaufen ist. Damit dies funktioniert, muss der Quest angegeben werden, an den das Briefing gehangen wurde.

Auf diese Weise musst Du nicht mehr selbst überwachen, wenn ein Briefing beendet ist. Das System nimmt Dir diese Arbeit ab. Ein Grund weniger für Kopfschmerzen! Des weiteren können verschiedene Effekte für Briefings und Kameraflüge ein- und ausgeschaltet werden. Dazu zählen Anzeigen des Himmels, verstecken der Grenzen von Territorien, Typ der verwendeten Cinematic Decoration und vieles mehr.

Grundlagen für Dialoge und Missionsbeschreibungen

Damit das BriefingSystem ordnungsgemäß arbeiten kann, muss ein Grundschema eingehalten werden. Es ähnelt dem altbekannten Schema, wurde aber um einige neue Inhalte ergänzt. Das werde ich nun grundlegend beleuchten.

```
-----  
local briefing = {  
    barStyle = "big",  
    disableGlobalInvulnerability = false,  
    restoreCamera = true,  
    restoreGameSpeed = false,  
    skipPerPage = true,  
    hideFoW = true,  
    showSky = true,  
    hideBorderPins = true  
};  
local AP, ASP, ASMC = AddPages(briefing)  
-----
```

Dies ist der Kopf des Briefings. Hier werden die Eigenschaften des Briefings notiert, ähnlich wie bei den Aufträgen. Dies ist die empfohlene Grundkonfiguration für ein Missions-Briefing, die Du getrost ohne zu Fragen übernehmen kannst. Wir wollen nicht viel von der Spielwelt zeigen, daher werden die breiten Balken verwendet. Der Nutzer soll lesen, daher muss jede Seite einzeln

⁵ Engine: ist ein spezielles Framework für Computerspiele, das den Spielverlauf steuert und für die visuelle Darstellung des Spielablaufes verantwortlich ist. In der Regel werden derartige Plattformen auch als Entwicklungsumgebung genutzt und bringen dafür auch die nötigen Werkzeuge mit.

weggeklickt werden. Zum Schluss werden noch ein paar Schönheitseinstellungen gemacht und die Kameraposition vor dem Briefing gespeichert. Wer will, kann auch die Spielgeschwindigkeit am Ende des Briefings wiederherstellen lassen. Dies wird notwendig, wenn Du nicht willst, dass Briefings höhere Spielgeschwindigkeiten herabsetzen.

Anschließend folgen die Deklarationen der Seiten. Für ein Dialog-Briefing empfehle ich diese Kurzschreibweise: Entity, Name des Sprechers, gesprochener Text, Dialogsicht an/aus, Action. Dies sind die Paramater für einen Aufruf von **ASP**. Natürlich steht es Dir frei auch die alte Schreibweise mit **AP** zu verwenden und dies nach belieben frei zu mischen.

```
ASP("alandra", "Alandra", "Hallo Marcus, schön Dich zu sehen.", true);
ASP("marcus", "Marcus", "Danke! Dann frisch ans Werk!", true);
ASP("alandra", "Alandra", "Lasst uns keine Zeit verlieren.", true);
```

Schlussendlich fehlt noch der Fuß.

```
briefing.finished = function()
end
return StartBriefing(briefing);
```

Im Fuß wird eine Funktion vereinbart, die am Ende des Briefings ausgeführt wird. In dieser Funktion können beliebige Dinge geschehen, sie kann aber auch weggelassen werden. Wichtiger ist die letzte Zeile der Funktion. **StartBriefing** führt das Briefing aus und gibt die ID zurück. Diese ID wird für die Anbindung an das Questsystem benötigt. Du darfst auf keinen Fall das **return** vergessen, sonst funktioniert es nicht!

Weggablungen in Dialogen

Weggabelungen in Dialogen sind unter der Bezeichnung Multiple Choice bekannt. Man versteht darunter einen Auswahldialog mit mehreren Optionen, aus denen der Spieler eine wählen muss, damit er fortfahren kann. Um die einzelnen Zweige eines solchen Briefings zu trennen, werden Leerseiten und Sprünge verwendet.

Eine Leerseite ist ein Aufruf von **AP** ohne Argumente. Dies signalisiert, dass das Briefing zu Ende ist. Andernfalls würde einfach die nächste Seite angezeigt.

```
AP();
```

Ein Sprung bietet die Möglichkeit zu einer anderen Seite zu wechseln, ohne das ein weiterer Auswahldialog benötigt wird. Als Argument wird die Nummer der Zielseite angegeben.

```
AP(11);
```

Der Auswahldialog ist der wichtigste Teil. Hier wählt der Spieler die Optionen aus. **ASMC** ähnelt dabei **APS**, nur das an die Stelle des Callbacks die Optionen getreten sind.

```
local CP = ASMC("hero", "", "Wie entscheidest Du Dich?", true,
    "Gut, ich werde es tun!", 5
    "Niemals, vergesst es!", 10
);
```

In diesem Beispiel wird zur Seite 5 gesprungen, wenn der Spieler die erste Antwort auswählt, und zur Seite 10, wenn der Spieler die zweite Antwort auswählt. Die Anzahl an Optionen ist unbegrenzt, übertreibe es aber nicht! Versuche nicht mehr als 8 Möglichkeiten pro Seite anzuzeigen.

Anschließend an das Briefing möchtest Du vielleicht wissen, welche Antwort gewählt wurde. Dies geschieht mit der Funktion **MCGetSelectedAnswer**. Ihr wird als Argument die erzeugte Seite übergeben, die von **ASMC** zurückgegeben wurde. Du erhältst die Nummer der gewählten Antwort. Die Nummer entspricht von oben betrachtet der Position der Antwort.

```
if MCGetSelectedAnswer(CP) == 1 then
    -- Mach was!
else
    -- Mach was anderes!
end
```

Auf diese Weise kannst Du, basierend auf der gewählten Antwort, auf die Entscheidung des Spielers reagieren. Wozu Du dies nutzt, bleibt deiner Fantasie überlassen.

Grundlagen für Kameraflüge

Unter einer Cutscene versteht man einen effektvollen Kameraflug. Cutscenes sind *nicht* dafür gedacht, Dialoge abzubilden. Zudem stehen einige Funktionen nicht zur Verfügung, wie z.B. das Setzen von Markierungen auf der Map oder Multiple Choice. Diese sind Briefings vorbehalten. Dafür können aufwendige Bewegungsabläufe erzeugt werden.

```
local cutscene = {
    barStyle = "small",
    disableGlobalInvulnerability = false,
    restoreCamera = false,
    restoreGameSpeed = false,
    skipAll = true,
    hideFoW = true,
    showSky = true,
    hideBorderPins = true
};

local AF = AddFlights(cutscene)
```

Dies ist der Kopf einer Cutscene mit den empfohlenen Einstellungen. Für den Anfang wäre es weise diese zu übernehmen. Später spricht nichts dagegen mit den möglichen Einstellungen herum zu experimentieren. Da wir etwas von der Welt sehen wollen, verwenden wir die schmale Dekoration. Der Nutzer soll die Landschaft genießen, daher werden die Seiten automatisch übergehen. Zum

Schluss werden noch ein paar Schönheitseinstellungen gemacht. Wer will, kann auch die Spielgeschwindigkeit am Ende des Briefings wiederherstellen lassen. Dies wird notwendig, wenn Du nicht willst, dass Briefings höhere Spielgeschwindigkeiten herabsetzen.

Eine Cutscene verwendet ausschließlich die Funktion **AF**. Natürlich kann auch **AP** Cutscenes nutzen, allerdings bietet **AF** eine übersichtlichere Notation. Du musst Dir keine Gedanken darüber machen, wie viele Seiten Dein Kameraflug hat. Außerdem willst Du keine permanenten Änderungen vornehmen, wie z.B. Gebiete aufdecken und Markierungen anbringen.

Aus diesem Grund müssen die einzelnen Seiten nicht zugreifbar sein. **AF** kann aus einer Liste von Positionen eine Bewegung erzeugen, die alle Positionen nacheinander abfährt.

Ein Flight besteht immer aus mindestens zwei Punkten: Einem Startpunkt und einer Position, zu der sich bewegt werden soll. Jeder einzelne Punkt kann einen eigenen Text anzeigen. Die Dauer der Anzeige wird mit **Duration** eingestellt. Alle Stationen eines Fluges, außer dem Startpunkt, teilen sich die Gesamtdauer der Bewegung gerecht untereinander auf.

Hat ein Flight z.B. 5 Punkte (Startpunkt und 4 Stationen) und eine Anzeigedauer von 12 Sekunden, wird erst zum Startpunkt gesprungen und danach jede der 4 Stationen angeflogen. Die Bewegung von einem Punkt zum anderen wird durch 4 geteilt. Also dauert die Anfahrt zum nächsten Punkt immer 3 Sekunden (12 Sekunden Anzeigedauer / 4 Positionen = 3 Sekunden Flugzeit).

```
AF {  
  {  
    Position = {X= 12300, Y= 23000, Z= 3400},  
    LookAt   = {X= 22000, Y= 34050, Z= 200},  
    Text      = "Das ist ein Text....",  
  },  
  {  
    Position = {X= 12300, Y= 23000, Z= 3400},  
    LookAt   = {X= 22500, Y= 31050, Z= 350},  
    Text      = "Das ist ein Text....",  
  },  
  FadeOut   = 0.5,  
  FadeIn    = 0.5,  
  Duration  = 24,  
};
```

Abgeschlossen wird die Cutscene abermals mit ihrem Fuß. Er besteht, wie gewohnt, aus einer Finished-Funktion und dem Startbefehl.

```
cutscene.finished = function()  
end  
return StartCutscene(cutscene);
```

Im Fuß wird eine optionale Funktion deklariert, die am Ende der Cutscene ausgeführt wird. Wichtiger ist die letzte Zeile der Funktion. **StartCutscene** führt die Cutscene aus und gibt wieder die ID zurück. Diese ID wird für die Anbindung an das Questsystem benötigt.

Relative Positionsangaben

Mit Koordinaten im dreidimensionalen Raum zu arbeiten kann auf Anfänger abschreckend wirken. Daher gibt es neben der Angabe von XYZ-Koordinaten auch die Möglichkeit, Entities zu verwenden. Man kann entweder die direkte Position eines Entity verwenden, oder die relative Position in einem bestimmten Abstand und Winkel.

```
-- {Entity, Kamerahöhe, Distanz, Winkel}  
LookAt      = {"Blickpunkt1", 200},  
Position    = {"Kameraposition1", 450, 3000, 25},
```

In diesem Beispiel ist die Blickrichtung absolut und die Position der Kamera relativ. Natürlich geht das auch andersrum. Mit diesen speziellen Einstellungen sind interessante Kameraeffekte möglich. Scheue Dich nicht, es zu probieren!