Inhaltsverzeichnis

Vorwort

Die Symfonia-QSB, oder auch QSB Plus, hat das Ziel das Leben des Mappers zu erleichtern. Aber wie es stets der Fall ist mit neuen Technologien, muss auch hier ein klein wenig beachtet werden um in den Genuss der Vorzüge dieses Frameworks[1] zu kommen.

Nachfolgend werde ich die wichtigsten Dinge, angefangen mit der Installation, erklären.

Installation

QSB-Datei

Sobald du eine neue Map erstellt hast, muss die QSB in die Map *importiert* werden. Siehe dazu das SEED Handbuch, falls dir der Prozess nicht klar ist. Ein Release kommt mit *zwei* Versionen der QSB, die sich nur in einem Punkt unterscheiden: dem Speicherplatz.

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 unveränderte QSB. Sie enthält noch alle Leerzeichen und Code-Kommentare und eignet sich zum stöbern, sollte Dir die Dokumentation nicht ausreichen um Deine Fragen zu klären.
>
> Die Datei `questsystembehavior_min.lua` wurde auf ein Minimum komprimiert, sodass sie etwa um die Hälfte weniger Speicherplatz benötigt. Allerdings ist der Code durch das Entfernen von Leerzeichen und Kommentaren sehr unleserlich.

Letztlich ist es egal, welche der beiden Dateien du importierst. Die minimierte QSB bietet dir nur den Vorteil weniger Speicher zu benötigen.

Die Mapskripte

Die QSB wird nicht mit den Standardskripten des Mapeditors funktionieren! Als erstes musst Du im Mapeditor den Expertenmodus aktivieren. Sobald das getan ist, kannst du eigene Skripte importieren. Siehe dazu das SEED Handbuch.

Im Release-Ordner befinden sich die Skripte `mapscript.lua` und `localmapscript.lua`. Diese Dateien müssen in deine Map importiert werden.

DEBUG aktivieren

Du möchtest vorher vielleicht den DEBUG aktivieren, damit du Fehler schneller entdecken kannst. Füge dazu in der `Mission_FirstMapAction` im globalen Skript die Funktion `API.ActivateDebugMode` nach dem Aufruf von Init `InitKnightTitleTables` ein.

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

Eine ausreichende Konfiguration ist folgende:

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

Dies aktiviert alle Checks und die Konsole, aber nicht Quest Trace. Für mehr Informationen zum Debug findest du in der Dokumentation (doc/index.hml).

Schnelles Testen

Nach *jeder* Änderung im Skript das Spiel verlassen, den Editor starten und das Skript neu importieren zu müssen, 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 Skripte, 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.

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 Quests[2], 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 weglassen zu können. Dies geht einher mit einer neuen Schreibweise.

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

Ein Beispiel für die neue Schreibweise:

```
local QuestName = API.AddQuest {
    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.

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

Angaben für einen Auftrag

Aufträge haben viele mögliche Angaben. Die Eigenschaften eines Quests 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 Quests angegeben. Dabei ist zu beachten, dass jeder sichtbare Quest nur ein Goal haben kann. Auslöser, oder auch Trigger, können mehrere vorhanden sein. Sie müssen alle wenigstens einmal ausgelöst haben, damit der Quest startet.

Mission-Briefings und Cutscenes

Aufstieg eines Königreichs bietet nicht von Haus aus an, Missions-Briefings oder Kameraflüge zu erstellen. Allerdings bietet die Engine[3] des Spiels einen entsprechenden Kameramodus an. Das so genannte BriefingSystem unterteilt dabei in zwei verschiedene Arten von Briefings. Das Missions- oder Dialog-Briefing und der Kameraflug oder Cutscene.

Neu ist jetzt, das Briefings (bzw. Cutscenes) an einen Quest mittels der Behavior Reward_Briefing und Reprisal_Briefing „angehangen“ werden können. Über den Auslöser Trigger_Briefing kann anschließend ein *anderer* Quest gestartet werden, *sobald* das Briefing *vollständig* durchgelaufen ist.

Das Briefing-Gerüst

Damit das BriefingSystem ordnungsgemäß arbeiten kann, muss ein Grundschema eingehalten werden. Dieses Schema werde ich nun grundlegend beleuchten.

```
local briefing = {
    barStyle = "big",
    disableGlobalInvulnerability = false,
    restoreCamera = true,
    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 Quests. Dies ist die empfohlene Grundkonfiguration für ein Missionsbriefing, 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 weggeklickt werden. Zum Schluss werden noch ein paar Schönheitseinstellungen gemacht und die Kameraposition vor dem Briefing gespeichert.

3 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.

Anschließend folgen die Deklarationen der Seiten. Für ein Dialog-Briefing empfehle ich diese Kurzschreibweise: Entity, Name des Sprechers, gesprochener Text, Dialogsicht an/aus. Natürlich steht es Dir frei auch die alte Schreibweise mit AP zu verwenden und 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 deklariert, 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.

Das Cutscene-Gerüst

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,
    skipAll = true,
    hideFoW = true,
    showSky = true,
    hideBorderPins = true
};
local AP = AddPages(cutscene)
```

Der Kopf der Cutscene sieht etwas anders aus. Eine Cutscene kann nur komplett abgebrochen werden. Zudem benötigen wir die schmalen Balken um mehr von der Spielwelt zu zeigen, während sich die Kamera bewegt.

Eine Cutscene verwendet ausschließlich die Funktion AP, welche hier etwas anders funktioniert.

```
AP {
    view            = {
        LookAt      = {X= 47146.44, Y= 56857.15, Z= 3802.56},
        Position    = {X= 47066.24, Y= 56801.98, Z= 3825.47},
        Duration    = 0.0,
    },
}

AP {
    text            = "Hier wird Text angezeigt.",
    view            = {
        LookAt      = {X= 47469.25, Y= 57228.40, Z= 3699.98},
        Position    = {X= 47402.09, Y= 57157.33, Z= 3720.89},
        Duration    = 15.0,
        FlyTime     = 15.0,
    },
}
```

Relative Positionsangaben

Neben der Angabe von XYZ-Koordinaten können auch Entities verwendet werden. Hier ist es außerdem möglich sich in einer gewissen Entfernung relativ im Winkel zur Entity zu bewegen.

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

Erneut fehlt nur noch der Fuß.

```
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.