Anfänger

Allgemeine CommandScript-Struktur

Inhaltsverzeichnis

- 1 Allgemeine CommandScript-Struktur in Emergency 4:
- 2 object <NameDesCommands> : CommandScript
- 3 Der Konstruktor "A09_Blaulicht()"
- 4 Die Command-Möglichkeitsabfrage bool CheckPossible(GameObject *Caller)
- 5 Die Zielabfrage bool CheckTarget(GameObject *Caller, Actor *Target, int childID)
- 6 void PushActions(GameObject *Caller, Actor *Target, int childID)

1 Allgemeine CommandScript-Struktur in Emergency 4:

Da ich vor einer Weile angefangen habe selbst Code zu schubsen für dieses doch schon 20 Jahre alte Spiel, habe ich mir ein paar Sachen angeschaut, um ein Grundverständnis zu entwickeln. Das SDK und auch die originalen Scripte von Emergency 4 geben hier bereits Aufschluss über die jeweiligen Mechaniken dahinter. Da ich nur randweise mit C++ bisher zu tun hatte (komme eher aus C#, embedded C und Python, alles im Automotive-Bereich), musste ich mir ein paar Quirks selber beibringen, die aber jetzt sehr nützlich für euch sein könnten.

Da ich euch direkt an der Basis abholen möchte, möchte ich es so simpel wie möglich darstellen. D.h. ihr solltet bereits ein bisschen Coding-Erfahrung haben, jedoch müsst ihr jetzt nicht die größten Cracks dafür sein, um mit dem Scripten anzufangen. (Keine Angst, auch ich hab noch lange nicht alles gelernt, aber einen Teil kann ich bereits mit euch teilen...) An einem simplen Beispiel möchte ich euch nun aufzeigen, wie man ein solches Script baut. Ganz easy wählen wir das Beispiel ein Blaulicht an- oder auszuschalten.

Wie alle Programme folgt auch Emergency 4 einem gewissen Schema, wie es die Commands aufbaut:

2 object <NameDesCommands> : CommandScript

Um der Spieleengine das Command überhaupt lesbar zu machen, kommt das Schlagwort "object" zum Einsatz. Danach müsst ihr euch einen Namen für das Command überlegen. Für unseren Anwendungsfall möchten wir das Blaulicht an-/ausschalten. Da Emergency die Befehle alphabetisch sortiert, habe ich mir eine Art eigene Nomenklatur überlegt. Meine Befehle fangen immer mit A%d%d an. D.h. A gefolgt von zwei Zahlen. Ihr könnt das jedoch machen wie die Dachdecker - ihr müsst euren Befehl nur wiederfinden können :grinning_squinting_face: (Der Vorteil meiner Schreibweise ist, dass ich ihn direkt am Anfang der Liste der Commands wiederfinde)

```
1   object A09_Blaulicht : CommandScript { };
```

Zur Info: mit dem Doppelpunkt und einem Namen ("... : CommandScript") wird hier bereits ein wichtiges Element in der Programmierung aufgezeigt. Es handelt sich um die sogenannte Vererbung. D.h. ihr könnt den Befehlssatz, der in dem anderen Objekt (in diesem Fall der Klasse CommandScript) gespeichert ist, auch für dieses Objekt nutzen. In C++ und in Emergency 4 ist es möglich mehrere Klassen gleichzeitig zu vererben.

Nachdem nun das Objekt errichtet wurde, bewegt ihr euren Mauszeiger in die geschwungenen Klammern, um das Script zu beschreiben. Die geschweiften Klammern {} und das abschließende Semikolon ; sind wichtig, um die Klasse oder das Objekt "abzugrenzen". Sonst kann Emergency nicht entscheiden, was noch alles dazugehört und was nicht.

```
object A09_Blaulicht : CommandScript
{
    //
};
```

Der Kommentar (//) ist nur ein Platzhalter.

3 Der Konstruktor "A09_Blaulicht()"

Im sogenannten Konstruktor werden grundlegende Parameter bei der Erstellung des jeweiligen Objekts festgelegt. Hier könnt ihr zum Beispiel mit den verschiedenen Einschränkungen (Restrictions), Prioritäten und weiteren Methoden arbeiten, die innerhalb der Klasse CommandScript zu finden sind oder einem allgemeinen C++ Befehlssatz entsprechen.

Für unseren Fall bedeutet das, dass wir ein Icon mit namen "Blaulicht" brauchen, und es auf eine niedrige Priorität setzen wollen. Der Code schaut nun wie folgt aus:

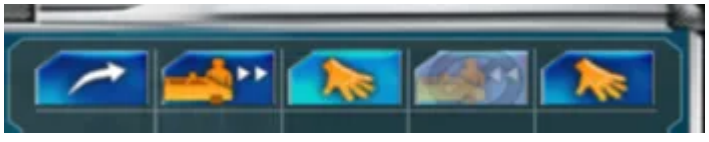
```
object A09_Blaulicht : CommandScript
{
    A09_Blaulicht()
    {
        SetIcon("Blaulicht");
        SetPriority(20);
        SetRestrictions(RESTRICT_SELFEXECUTE);
    }
};
```

4 Die Command-Möglichkeitsabfrage bool CheckPossible(GameObject *Caller)

Der nächste Codeblock, der nun folgt, checkt einen Wahrheitswert (Boolean, kurz bool) ab, ob dieses Command überhaupt ausgeführt werden kann. Da diese Methode einen Übergabeparameter braucht, wird der mit "GameObject *Caller" gesetzt. Ich werde später auf diese Übergabeparameter eingehen, aber erstmal bleiben wir beim Checken.

Der Inhalt kann nicht angezeigt werden, da du keine Berechtigung hast, diesen Inhalt zu sehen.

Hier könnt ihr jede Menge an Parameter abprüfen, um zu entscheiden, ob das Command überhaupt ausgeführt werden kann. Um es ein wenig simpler zu halten, habe ich nur geprüft, ob der Caller, also das ausführende Fahrzeug, überhaupt berechtigt/gültig ist. Weitere mögliche Abfrageparameter wären zum Beispiel das Abfragen, ob der Motor an ist oder ob das Auto noch an ein Elektrokabel angeschlossen ist oder oder oder. Der Rückgabewert der Methode gibt einen Wahrheitswert aus und ist entscheidend, ob das Command anklickbar ist oder "ausgegraut".



Hier sieht man zum Beispiel, dass mehrere Befehle möglich sind, jedoch das Einsteigen in das Fahrzeug noch nicht geht, weil entweder eine Voraussetzung zum Ausführen des Befehls nicht erfüllt ist oder das Ziel noch nicht passt. Die Zielabfrage folgt als nächster Schritt.

5 Die Zielabfrage bool CheckTarget(GameObject *Caller, Actor *Target, int childID)

Der nachfolgende Codeblock ermöglicht es dir zu prüfen, ob das "Ziel" deines Commands auch wirklich passt. Da wir das nur auf das Fahrzeug selbst anwenden wollen und nicht auf ein anderes Objekt, prüfen wir nun, ob Ziel und Herkunft die gleiche ID haben.

```
bool CheckTarget(GameObject *Caller, Actor *Target, int childID)
{
    return Caller -> GetID() == Target -> GetID();
}
```

Auch hier sehen wir, dass der Rückgabewert der Methode ein Wahrheitswert (bool) ist. Dieser kann true (richtig / ja) oder false (falsch / nein) sein. Anhand des Rückgabewertes wird nun angezeigt, ob dieser Befehl an seinem Ziel ausgeführt werden kann. Das Ziel kann hier ein Punkt auf der Map, ein Gebäude, Fahrzeug, Person, oder oder oder sein. Dies muss dann aber in CheckTarget oder als Restriction im Konstruktor (z.B. *SetValidTargets(ACTOR_STREET|ACTOR_FLOOR);*) hinterlegt sein. Hier kann man ganz schnell aber auch zu einem Problem kommen, deshalb ist es wichtig, dass ihr gut Debuggen könnt und euren Code versteht.

6 void PushActions(GameObject *Caller, Actor *Target, int childID)

```
void PushActions(GameObject *Caller, Actor *Target, int childID)
{
    Vehicle fahrzeug(Caller);

    // Pruefung, ob Licht an ist
    if (!fahrzeug.IsBlueLightEnabled())
    {
        fahrzeug.EnableBlueLights(true);
    }
    else
    {
        fahrzeug.EnableBlueLights(false);
    }
}
```

Im obigen Codeblock werdet ihr nun ein paar verschiedene Dinge sehen, die ich mache, um mir das Arbeiten etwas zu erleichtern. *Aber zuerst, was macht das void PushActions(...)?*

- void ist nur der Rückgabewert eurer Methode - wie bereits oben bei **bool** gesehen, kann eine Methode einen Rückgabewert haben. Da PushActions "nur ausgeführt" werden muss, gibt es nichts zurück (deshalb das Schlagwort "**void**"). Wäre es notwendig etwas zurückzugeben, würde auch der Typ des Rückgabewertes ein anderer sein.
- GameObject *Caller : der Caller ist vom Typ GameObject und kann z.B. eine Person oder ein Fahrzeug sein. (Ihr könnt euch hierzu die Vererbungsstruktur im SDK anschauen, welche Klasse welche Eigenschaften erbt)
- Actor *Target : das Target ist euer Zielobjekt (also z.B. der Mapboden, eine Person, oder was auch immer eure Maus gerade als Ziel hat)
- int childID : dieser sogenannte Integer (Ganzzahl) kann genutzt werden, um innerhalb eines Scripts bestimmte Logiken abzugreifen. Dieser Integer hat einen Normalzustand (default) von 0. D.h. wird dieser Methode nichts in den Parametern übergeben, das das childID setzen würde, übernimmt er immer 0.
- Pointer (*) : Das ist eine Sache für sich selbst, da man in C++ sehr viel mit Pointern arbeitet, sich aber auch ganz schnell das Genick brechen kann. Ich habs zwar verstanden, kann es aber noch nicht soooo gut erklären, dass es Hand und Fuß hat. Mehr oder weniger ist es eine Referenz zu einem

Objekt, das dann wiederum ausgelesen und/oder weiterverwendet werden kann.
- PushActions: ist der Methodenname

Jetzt zum Teil innerhalb der geschweiften Klammern:

- Um das Arbeiten leichter zu machen, habe ich ein Objekt vom Typ "Vehicle" mit dem Namen "fahrzeug" erschaffen, das dann mit den Parametern des Callers gesetzt wird. (Das ist jetzt vielleicht schon ein bisschen Advanced, deshalb gehe ich nicht genauer drauf ein - aber wollt ihr ein Objekt haben, mit dem ihr innerhalb der Methode arbeiten wollt, dann müsst ihr es so machen wie dort oben. Ist euer GameObject aber zum Beispiel eine Person, dann muss das natürlich auch richtig gesetzt werden.) Da wir mit einer lokalen Variable arbeiten, ist der Name in dem Fall kleingeschrieben. Das ist ein nützlicher Codingstyle-Tipp, falls ihr später mit globalen, lokalen und Übergabeparametern arbeitet.
- Nach den zwei Forwardslashes (//) folgt ein Kommentar, der keinen Einfluss auf das Script hat. Hier könnt ihr z.B. ToDos reinschreiben oder den Sinn eures Codeblocks dokumentieren. Bestenfalls schreibt ihr aber auch nicht kryptisch, da es ein anderer Entwickler auch lesen soll und sich nicht alles doppelt anschauen muss. (Btw - selbstsprechende Namen für Variablen sind auch sinnvoll, weil ich in Zeile 2000 nicht mehr weiß was "v.HasChild" ist während ich schon weiß, was "fahrzeug.HasChild" ist.)
- Als nächstes Element folgt eine Logikabfrage, die ich wie folgt im Kopf formuliert h ***"Falls das Blaulicht NICHT an ist, schalte das Blaulicht an. Andernfalls das Blaulicht aber bereits an ist, schalte es aus."***
- Mit einem if-Statement wird eine Kondition abgefragt, die in den runden Klammern hinterlegt ist. In unserem Fall wird das mit dem Ausrufezeichen (!) negiert, da ich den obigen Satz als Ausgang hatte.
- Ist die Logikabfrage nun erfüllt, also das Blaulicht ist AUS (IsBlueLightEnabled gibt FALSE zurück, es wird aber negiert zu TRUE), schaltet das Script das Blaulicht an dem Fahrzeug an (fahrzeug.EnableBlueLights(true);)
- Ist die Logikabfrage nicht erfüllt (springt das Programm in den ELSE Block), also das Blaulicht ist AN (IsBlueLightEnabled gibt TRUE zurück, es wird aber negiert zu FALSE), schaltet das Script das Blaulicht an dem Fahrzeug aus (fahrzeug.EnableBlueLights(false);)
- Da wir keine weitere Logik brauchen und keine weiteren Anweisungen vorgesehen sind, schließt mit diesem Statement auch das komplette Objekt ab. Der vollständige Code lautet nun:



FYI: Ihr hättet die Blaulichtabfrage auch andersherum aufzäumen können, ich selbst habe aber entschieden immer den ausgeschalteten Zustand als den Default zu nehmen, deshalb muss ich die Abfragen oft negieren. Das kann, besonders bei großen Teams mit mehreren Programmierern, zu Verwirrung führen, deshalb KOMMENTIERT euer Zeug ordentlich! :winking_face:

Ich hoffe, es hat euch geholfen diesen grundlegenden Überblick über die Scriptstruktur für ein CommandScript zu bekommen und könnt nun selbst loslegen.

LG Korbi :grinning_squinting_face:

Vollständige Nomenklar, falls ihr das wissen wollt.

Der Vollständigkeithalber hier meine vollständige Nomenklatur für Befehle:

- **A00 - Befehle für Events und Alarmierungen**
- **A01 - Befehle der Feuerwehr**
- **A02 - Befehle des Rettungsdienstes**
- **A03 - Befehle der Polizei**
- **A04 - Befehle der Techniker**
- *A05 - aktuell nicht belegt*
- *A06 - aktuell nicht belegt*
- *A07 - aktuell nicht belegt*
- *A08 - aktuell nicht belegt*
- **A09 - Befehle für alle Fahrzeuge**
- **A99 - Befehle zur Steuerung von WorldEvents, Debugging, uvm.**

Verwendete Programme, Einstellungen und Darstellung im Arbeitsbereich

- Visual Studio Code (free)
- C++ Darstellung (free, vorinstalliert in VSC)
- Dark - High Contrast Theme (free, vorinstalliert in VSC)
- Git zur Source Control (prinzipiell erstmal free, braucht aber einen Account auf Github oder einen Adressserver und ein bisschen tieferes Wissen zur Versionskontrolle;)
- es wäre SVN auch möglich, aber ich benutze ein privates Github-Repo für meine persönlichen Arbeiten

Das vollständige Window sieht aktuell so aus (nur ein Auszug aus den Arbeiten, die ich gerade mache):

