IVANOFF Script Language

en vejledning

version 1.1 - 4. April 2017

Forord

Tillykke med Deres nye *IVANOFF Script Language* Oversætter!
De har med erhvervelsen af dette moderne Stykke Teknologi faaet et teknologisk Forspring, der vil efterlade Deres Konkurrenter i Krampegraad og faa Deres Kunder til at juble! Alt sammen baseret paa den moderne Rumalders nyeste Forskning indenfor Oversætterteknologi.
Hvis de anvender Deres Oversætter efter vore forskrifter, vil den kunne tjene Dem trofast igennem mange Projekter, langt ind i det næste Aartusind.

Med højagtelse

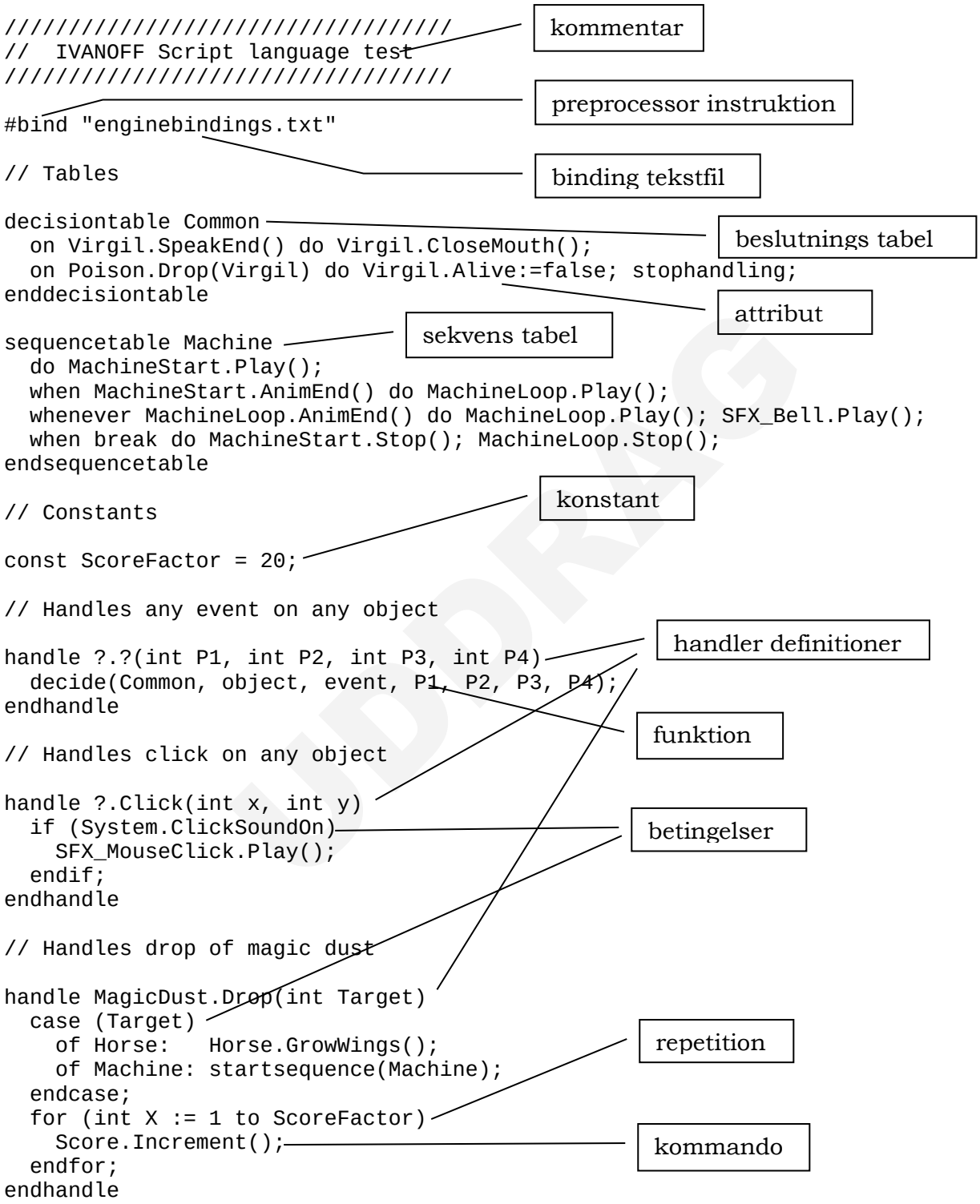
Svane, Halkjær & Nielsen
Oversættere, En Gros & Detail

- Tilegnet Gud Mammon.

INDHOLD

ET SCRIPT

Her følger et eksempel på et script til et ikke eksisterende spil, bare lige for at se giraffen. Noterne udpeger nogle af de begreber der forklares i det efterfølgende.




```
////////////////////////////////
//  IVANOFF Script language test
////////////////////////////////

#bind "enginebindings.txt"

// Tables

decisiontable Common
  on Virgil.SpeakEnd() do Virgil.CloseMouth();
  on Poison.Drop(Virgil) do Virgil.Alive:=false; stophandling;
enddecisiontable

sequencetable Machine
  do MachineStart.Play();
  when MachineStart.AnimEnd() do MachineLoop.Play();
  whenever MachineLoop.AnimEnd() do MachineLoop.Play(); SFX_Bell.Play();
  when break do MachineStart.Stop(); MachineLoop.Stop();
endsequencetable

// Constants

const ScoreFactor = 20;

// Handles any event on any object

handle ?.?(int P1, int P2, int P3, int P4)
  decide(Common, object, event, P1, P2, P3, P4);
endhandle

// Handles click on any object

handle ?.Click(int x, int y)
  if (System.ClickSoundOn)
    SFX_MouseClick.Play();
  endif;
endhandle

// Handles drop of magic dust

handle MagicDust.Drop(int Target)
  case (Target)
    of Horse:   Horse.GrowWings();
    of Machine: startsequence(Machine);
  endcase;
  for (int X := 1 to ScoreFactor)
    Score.Increment();
  endfor;
endhandle
```

HVORDAN, MOR?

— *" The last thing that we find in making a book is to know what we must put first." (Blaise Pascal)*

En måde at håndtere den kompleksitet et stort multimedie program har, er at opdele det i mindre, håndterlige enheder.
Man kan opdele efter forskellige kriterier: efter størrelse (lille enhed = mere overskuelig), tematisk (færre forskelligheder = mere overskuelig) osv. Opdelingen af et spil i trekløveret Editor/Engine/Script hører til i den tematiske kategori.
I en (level-) editor kan man fokusere på det rent visuelle og lade editoren holde styr på koordinater og andre rå data, som programmet i sidste ende skal anvende, men som egentlig ikke er interessante for spil konstruktøren.

Opdelingen mellem game engine og script kan sammenlignes med et gammeldags, selvspillende klaver med hulstrimmel (og sådan et har vi jo alle sammen…), hvor klaveret (enginen) har alle de egenskaber der skal til for at afspille en melodi – tryk på tangent udløser bestemt tone – mens hulstrimlen (scriptet) får det til at ske efter et givet mønster.
Eller mere specifikt (uden flere tåbelige overforenklinger):
Enginen kan alt det som spillet skal kunne, f.eks. at få en bestemt figur til at gå i 8 retninger eller trække bestemte ting rundt med musen og måske putte dem i en pose! Dette kan ske udfra nogle i enginen indbyggede regler, f.eks. kunne ovenstående gående figur respektere vægge eller andre blokerende ting i spillet.
Scriptet er så det manuskript, der aktiverer alle disse muligheder i den rækkefølge som spil designeren har ønsket.

Nu er rækkefølge ofte noget, der kan være vanskeligt at beskrive i interaktive programmer. Derfor er scriptet opdelt på en måde, der forsøger at reducere denne kompleksitet (efter det tematiske princip): scriptet opdeles i sektioner, der hver beskæftiger sig med reaktionen på en bestemt hændelse i game enginen (såkaldte "events" – heraf det berømmelige "event drevne paradigme" som vil være nogle bekendt) f.eks. kunne en sektion beskæftige sig med hvad der sker når enginen registrerer at spilleren har klikket med musen på den blå vase.
Scriptet kan så aktiverer enginens mekanismer ved at sende kommandoer til enginen – i tilfældet "klik på vasen" event, sendes måske en kommando, som får hovedpersonen til at sige en sætning, der omtaler den klikkede vase[1].

En sektion i scriptet som håndterer et bestemt event, kaldes en "handler" (udtales på engelsk – det er ikke den frie købmand vi taler om).

Selve script teksten er udformet som et almindeligt programmerings sprog – dvs. et stærkt formaliseret sprog med udgangspunkt i engelsk. Det består af en række indbyggede ord som kan sammenstilles efter et sæt regler (sprogets "grammatik") og som fungerer som en række kommandoer, der udføres i den angivne rækkefølge.

Sproget adskiller sig dog på to områder fra "almindelige" sprog af denne type: for det første er der udover de omtalte kommandoer, der udføres i deres opstillede rækkefølge, mulighed for opstilling af specielle tabeller, der har en større grad af "mekanik" indbygget. Dette betyder at man i højere grad noterer hvad målet er, end hvordan det nås

[1] her kan enginen så evt. have den indbyggede regel at når hoved personen siger noget afspilles en tale animation – skal hovedpersonen derimod også kunne tale uden animation går det ikke med en sådan automatik, og man må i stedet sende to kommandoer: en "sig en sætning" kommando og "vis taleanimation" kommando – man kan med andre ord konkludere, at opdelingen mellem hvad enginen tager sig af og hvad scriptet tager sig af, ikke er knivskarp.

(såkaldt deklarativ programmering). Dette skal ses som tilpasning til sprogets specielle anvendelses område. I modsætning til mere generelle programmeringssprog, der netop tilstræber at være generelle, kan vi her tillade os at automatisere de almindeligst forekommende/mest omstændelige problemer og indføre specielle notations former for disse.
For det andet kan sproget let tilpasses en game engine. Der skal blot inkluderes en tekstfil med oversigt over navnene på de kommandoer som enginen forstår og de events som den kan registrere og videresende. Scriptet kan med andre ord let udvides i takt med enginen blot ved at tilføje nye kommandoer og events til denne tekst fil.
Alle ting i spillet, der skal kunne refereres til i scriptet (det kan være objekter, locations osv), forudsættes at have et unikt navn som er med i tekst filen.
Denne tekstfil kaldes ”binding” filen (lyder mest cool udtalt på engelsk) fordi den binder et symbolsk navn sammen med enginens interne repræsentation (som regel en talværdi).

Disse navne skal bestå af tegnene: ”a”-”z” (små eller store), ”_” (underscore) eller ”0”-”9” (cifre). Dog må de ikke begynde med et ciffer.
Dette gælder alle navne i scriptet som ikke er en indbygget del af script sproget.

De fleste eksempler i denne tekst følger nogenlunde det samme system med hensyn til opdeling af scriptet på tekst linier, men faktisk ignoreres linieskift af compileren, så scriptet kunne faktisk skrives på én tekst linie (men det kan næppe anbefales af hensyn til læsbarheden).
Kun ét sted har linieskiftet en betydning: ved kommentarer der begynder med ”//” (se ”kommentarer”).

Og hvad er det så compileren gør?
En compiler oversætter simpelthen et sprog til et andet efter et sæt regler.
I dette tilfælde oversættes det script sprog, der (forhåbentlig) er velegnet for mennesker til at skrive scripts i, til et binært maskinsprog, som er velegnet for maskinen at fortolke og udføre.

Script filerne er almindelige tekst filer som f.eks. Notepad laver dem. et skal være ren ANSI tekst filer, så mere eksotiske formater med formatterings koder som f.eks. Word går altså ikke (man kan altid prøve at åbne en fil i Notepad og se om der er mærkelige tegn og koder imellem den egentlige tekst.
Der findes forskellige editors, beregnet til at lave program kode, som laver rene tekst filer, men som har flere faciliteter end Notepad.

HANDLERS

— " Is this a dagger which I see before me, The handle toward my hand?" (Shakespeare)

En handler er som nævnt grundinddelingen i scriptet. En handler består af en specifikation af hvilket event den håndterer og en række script kommandoer, der udføres når eventet opstår.
Alle events der opstår i enginen og som skal sendes til scriptet bliver sendt. Hvis ikke der findes en handler der håndterer eventet, sker der blot ikke yderligere. Eks: hvis enginen sender "museklik på ting i spillet" events til scriptet modtager scriptet alle disse events. Klikkes spilleren på et bord modtager scriptet et "museklik" event med oplysning om at det blev udført på "bord". Hvis der ikke er en handler, der tager sig af dette specifikke event på denne ting, sker der simpelthen ikke yderligere. Er der en "museklik på bord" handler udføres de kommandoer, der er i denne handler.
Hvis museklik eventet har fået navnet "MouseClick" og bordet har fået navnet "KitchenTable" (her menes de navne i den tidligere omtalte binding tekstfil, som er fastlagt på forhånd som en "aftale" mellem script og engine) ser den enkleste handler for dette, således ud:

```
handle KitchenTable.MouseClick()
endhandle
```

Denne handler vil blive udført ved museklik på bord. Problemet med netop denne handler er at den ikke gør noget som helst – den er tom!
Kommandoer der skal udføres ved klik på bord skrives mellem disse to linier, f.eks:

```
handle KitchenTable.MouseClick()
   SFXKnockWood.Play();
   KitchenTable.HasBeenClicked := true;
endhandle
```

Nu sker der noget! Hvis der klikkes på bordet udføres kommandoerne.
Første kommando *SFXKnockWood.Play();* () udfører kommandoen Play på objektet SFXKnockWood. Følgende er forudsat for at denne kommando vil føre til noget:
Der findes et SFXKnockWood objekt i spillet (kunne passende være en lydeffekt) og enginen forstår at udføre kommandoen Play på dette objekt (som passende kunne starte afspilningen af bemeldte lydeffekt).

Kommandoen slutter med et semikolon der altid anvendes til at afslutte kommandoer i scriptet. Parenteserne "()" efter kommando/event navnet har to funktioner: dels angiver det at der er tale om en handling i modsætning til en attribut (se beskrivelse af næste kommando), dels kan der mellem parenteserne angives yderligere oplysninger (parametre) vedrørende handlingen – mere herom senere.

Anden kommando *KitchenTable.HasBeenClicked := true;* sætter KitchenTable's attribut HasBeenClicked. Her forudsættes altså at KitchenTable har en sådan attribut defineret.

Objektnavn.Aktionsnavn
notations formen anvendes så vidt muligt konsekvent i script sproget – jf. KitchenTable.MouseClick() og SFXKnockWood.Play() – man kunne i den forbindelse snildt hævde at en kommando fra scriptet bliver events til enginen og at events til scriptet er kommandoer sendt fra enginen – eller: en kommando bliver til event hos modtageren og et event sendes som kommando fra afsenderen – derfor har de samme notation.

En attribut er en egenskab som er tildelt objektet – data vedrørende objektet – som kan aflæses og evt. sættes fra scriptet[2].
I dette tilfælde sættes KitchenTable's attribut HasBeenClicked til "true"[3]. Der er altså tale om at vi registrerer at der er klikket på KitchenTable. Vi kan så senere aflæse KitchenTable.HasBeenClicked attributten, for at se om der har været klikket på bordet.

Bundlinien i dette er at man som hovedregel skriver en handler for hvert af de events, som kan opstå i enginen, og som skal kunne føre til en reaktion der ikke allerede automatisk sker i enginen (nogle situationer håndteres nemmere med tabeller, men mere om dette senere).

For nogle events kan der være yderligere oplysninger tilstede som kan nås gennem parametre, der defineres mellem parenteserne.
Enginen kunne f.eks. tænkes at medsende koordinaterne for et museklik, i fald nogen skulle have brug for denne oplysning! Hvis enginen sender x koordinaten som første parameter og y koordinaten som anden kan parametrene defineres således:

```
handle KitchenTable.MouseClick(int X, int Y)
endhandle
```

Inde i handleren kan der nu refereres til "X" og "Y" der så vil repræsentere hhv. x og y koordinaten i museklikket. det foranstillede "int" fortæller script compileren at der er tale om heltal – en teknisk nødvendighed for at få skidtet til at fungere optimalt…

Ligeledes kan der medsendes yderligere oplysninger med kommandoer:

```
SFXKnockWood.Play(3);
```

Her medsendes værdien "3" som oplysning til kommandoen Play på SFXKnockWood. Effekten af dette afhænger af hvordan enginen anvender denne værdi. Dette kunne f.eks. være antallet af gange lydeffekten skal loop'es. Sådanne forhold skal selvsagt være defineret af enginen før de kan bruges i scriptet. Det ville ret sikkert være grimt hvis det var antallet af loops, men scripteren der skrev det troede at det var lydstyrken!

Der kan defineres flere handlere til samme event, hvilket letter arbejdet når flere scriptere arbejder på samme projekt. Hvis tre scriptere alle har defineret en "handle KitchenTable.MouseClick()" handler vil alle tre handlere blive udført ved klik på bordet. Rækkefølgen de udføres i er bestem af den rækkefølge de optræder i scriptet (den der optræder først, udføres først).
Dette kan overstyres ved at prioriterer ens handlere. Dette gøres ved at sætte "#" + et tal efter handler navnet. Handlerne udføres da med laveste tal først. I tilfældet:

```
handle KitchenTable.MouseClick() #2
endhandle;

handle KitchenTable.MouseClick() #1
endhandle;
```

[2] Det er ikke givet at alle attributter kan sættes fra scriptet. Har enginen f.eks. en attribut der registrerer antallet af gange spilleren har besøgt en location, kan den sandsynligvis kun aflæses. Hvis værdien kunne sættes fra scriptet kan man ikke længere regne med at den indeholder det faktiske antal gange spilleren har været på locationen.
[3] "true" er en indbygget værdi i scriptet der repræsenterer værdien "sand" i logiske udtryk. Der findes også en "false" som repræsenterer værdien "falsk".

udføres den sidste først, selvom den optræder sidst i script teksten, fordi den har det laveste prioritets nummer. Handlere uden nummer har altid laveste prioritet.

Der kan desuden defineres et hierarki af handlere som behandler events på mere eller mindre specifikke niveauer. Handleren:

```
handle KitchenTable.MouseClick()
endhandle
```

udføres som sagt når der klikkes på bordet.

Ved at udskifte ”KitchenTable” med ”?”:

```
handle ?.MouseClick()
endhandle
```

udføres handleren ved museklik på ethvert objekt. Hvis både denne mere generelle handler og den specifikke handler er defineret, udføres først den generelle og siden den specifikke.
Der kan defineres to andre typer af generelle handlere:

```
handle KitchenTable.?()
endhandle

handle ?.?()
endhandle
```

Den første udføres ved ethvert event på KitchenTable objektet. Den anden udføres ved ethvert event på ethvert objekt (med andre ord: ret tit!).

Rækkefølgen for udførelse af de ovenstående handlere ved museklik på bordet er:

```
1) handle ?.?()
2) handle ?.MouseClick()
3) handle KitchenTable.?()
4) handle KitchenTable.MouseClick()
```

Hvis ordet ”stophandling;” indsættes i en handler, stopper udførelsen af handleren i den pågældende linie og fortsætter ikke i mere specifikke handlere. Heller ikke andre handlere på samme niveau (hvis der f.eks. findes to ” KitchenTable.?()” handlere). Udførelsen af dette event stopper simpelthen!

Selvom man er i en generel handler (med ”?” i), kan det være rart at vide hvilket event eller objekt der er tale om.
Til det formål findes symbolerne ”object” og ”event” som altid vil indeholde værdien for det aktuelle objekt hhv. event[4].

[4] Da al kode i et script bliver aktiveret ved at enginen kalder en handler, vil ”object” og ”event” altid indeholde gyldige værdier. I en ”KitchenTable.MouseClick()” handler vil ”object” således være synonymt med ”KitchenTable” og ”event” være synonymt med ”MouseClick”.

KOMMANDOER

— *”For the Athenians command the rest of Greece, I command the Athenians, your mother commands me, and you command your mother” (Plutarch)*

Kommandoer er scriptets måde at aktivere de funktioner som ligger i game enginen. Hver kommando har et unikt navn, som ligger i den tidligere omtalte binding tekstfil. Internt i enginen optræder kommandoerne som regel som talværdier, så navnet er derfor en symbolsk repræsentation, aftalt mellem engine-konstruktør og scripter, som gør det lettere for scripteren at huske de enkelte kommandoer (og scriptet bliver også en del nemmere at læse efterfølgende).

En kommando kan imidlertid ikke stå alene – den skal have et mål. Målet er et af de objekter som (i lighed med kommandoerne) er navngivet og listet i binding tekstfilen. For at udføre ”Jump” kommandoen på ”Virgil” objektet, skrives: ”Virgil.Jump();”. Husk parenteserne som angiver at det er en kommando og ikke en attribut. Semikolonet er ikke en del af kommandoen men angiver blot at kommandoen slutter her (en slags skilletegn).

Hvis kommandoen kræver yderligere oplysninger skrives de som parametre mellem parenteserne. Kommandoen ”GoTo” kunne f.eks. kræve to tal som angiver koordinaten som objektet skal gå til. Man skriver ”Virgil.GoTo(2, 10);” for at få Virgil til at gå til position 2, 10 – dette forudsætter selvsagt at enginen forstår denne kommando og forventer disse to parametre i den angivne rækkefølge. Antallet og rækkefølgen af parametre til en kommando er fastlagt på forhånd af enginen.

SEKVENSTABELLER

— *"Order gave each thing view" (Shakespeare)*

Blandt andet for at komme uden om det berømte ”færøske kædedanser problem” (s.d.) introduceres sekvenstabellen.
Sekvenstabellen gør det muligt at opstille sekventielle forløb på en overskuelig måde, men baseret på de samme events, som resten af scriptet.

En sekvenstabel har formen:

```
sequencetable Tabelnavn
  do Sætninger1;
  when Eventspecifikationer1 do Sætninger2;
  whenever Eventspecifikationer2 do Sætninger3;
  when break do Sætninger4;
endsequencetable
```

Sekvenstabeller aktiveres ved funktionen ”startsequence(tabelnavn);”. De deaktiveres igen med funktionen ”endsequence(tabelnavn);” eller funktionen ”breaksequence(tabelnavn);”.

Når tabellen aktiveres udføres sætningerne i alle linier, som starter med ”do”.
En sætninger der starter med ”when” eller ”whenever” udføres når der kommer et event fra enginen som matcher eventspecifikationen.
Forskellen på ”when” og ”whenever” er, at linier med ”when” udføres første gang et matchende event ankommer (indtil tabellen deaktiveres og aktiveres igen), mens linier med ”whenever” udføres hver gang et matchende event ankommer.
I linier der starter med ”when break”, udføres sætningerne når tabellen deaktiveres med ”breaksequence” funktionen. ”breaksequence” og ”endsequence” fungerer ens, bortset fra at ”endsequence” ikke aktiverer ”when break” linien/linierrne.

En styrke ved sekvenstabeller er at eventspecifikationerne kan være mere komplicerede end i beslutningstabeller. Man kan på samme måde specificerer events med parametre, men man kan desuden specificere flere events og kombinere dem med de logiske operatorer ”and” og ”or”.

”Det færøske kædedanser problem” blev første gang registreret i efteråret 1995, under udarbejdelsen af ”Magnus & Myggen 1”.
Problemet opstår når flere hændelser skal udføres i sekvens, hvor den ene starter lige efter at den forgående er afsluttet.
Skal der f.eks. først afspilles en lyd som derefter følges af en animation der derefter følges af en replik, håndteres det normalt på følgende måde:
Lyden sættes i gang et sted i koden! Der laves en handler, som kaldes ved afslutning af lyden (afslutning af medie afspilning udløser nemlig ofte et event). I denne handler startes animationen. Der laves en handler, som kaldes ved afslutning af animationen. I denne handler startes replikken.
Dette fører til et mylder af handlere, der tager hinanden ”i hænderne” én efter én, ganske som deltagerne i de populære* færøske kædedanse.
Det er ikke alene vanskeligt at få overblik over – det er også ~~fandens~~ overordentligt irriterende at skulle skrive.
Problemet opstår fordi der midt i den hændelses styrede, ikke lineære arkitektur som scriptet baserer sig på, opstår behov for en lineær sekvens.
Dette forsøges imødekommet i sekvenstabellerne, uden at bryde med arkitekturen.

*) på Færøerne

Et simpelt eksempel på en sekvenstabel hvor 3 animationer skal afspilles i sekvens kunne se således ud:

```
sequencetable ASimpleSequenceTable
  do Animation1.Play();
  when Animation1.End() do Animation2.Play();
  when Animation2.End() do Animation3.Play();
  when break do Animation1.Stop(); Animation2.Stop(); Animation3.Stop();
endsequencetable
```

Når tabellen aktiveres udføres den første linie da den ikke er afhængigt af et event.
Når der kommer et event, som signalerer at Animation1 er slut[10], udføres linie 2, som igen starter Animation2 osv.
Bemærk linien med "when break". Den stopper alle tre animationer. Dvs. at en "breaksequence(ASimpleSequenceTable);" vil bevirke at den igangværende animation (uanset hvilken vi er nået til) vil blive stoppet. Uden denne linie (eller hvis "endsequence" anvendes i stedet) ville den igangværende animation kører til ende[11].
Her forudsættes at enginen ikke har noget imod at man beder om at få stoppet en animation der ikke er i gang (kun én af de tre vil være kørende på et givet tidspunkt) – som regel vil sådanne tiltag blot blive ignoreret, men check med din engine-udbyder!

Ud over de simple eventspecifiktioner i eksemplet ovenfor, kan man som sagt lave nogle mere komplicerede, f.eks:

```
sequencetable ACleverSequenceTable
  do Anim1.Play(); Anim2.Play();
  when (Anim1.End() and Anim2.End()) do Twist.AndShout();
endsequencetable
```

Når tabellen aktiveres startes animationerne Anim1 og Anim2. Når de begge to er slut (dvs. at der er kommer både et "Anim1.End()" event og et "Anim2.End()" event) udføres "Twist.AndShout()" kommandoen. Hvilket et der kommer først, eller hvor lang tid der er imellem dem, betyder ikke noget – når den sidste af dem ankommer, udføres kommandoen.

Hvis man nu f.eks. finder ud af at det er fint nok at kommandoen "Twist.AndShout()" udføres når de to animationer begge er slut, MEN at man også skal kunne klikke med musen som en slags genvej, skriver man blot:

```
sequencetable ACleverSequenceTable
  do Anim1.Play(); Anim2.Play();
  when ((Anim1.End() and Anim2.End()) or ?.MouseClick()) do Twist.AndShout();
endsequencetable
```

Derved udføres "Twist.AndShout()" kommandoen når enten beggen animationer er slut eller der klikkes på et eller andet. Klikker man med musen vil der altså ikke længere blive ventet på de to slut events.
Ønsker man at "Twist.AndShout()" kommandoen skal udføres både når musen klikkes og når begge animationer er slut skrives:

[10] Alle events og kommandoer forudsættes implementeret i den aktuelle engine, men det kan selvfølgelig skifte fra engine til engine hvorledes et animation-slut event ser ud (og om der overhovedet eksisterer et).
[11] Hvilket jo også kan være fint nok, hvis det er det man ønskede.

```
sequencetable ACleverSequenceTable
  do Anim1.Play(); Anim2.Play();
  when (Anim1.End() and Anim2.End()) do Twist.AndShout();
  when ?.MouseClick() do Twist.AndShout();
endsequencetable
```

Skal kommandoen ydermere udføres enten når begge animationer er slut eller *hvergang* der klikkes med musen skrives:

```
sequencetable ACleverSequenceTable
  do Anim1.Play(); Anim2.Play();
  when (Anim1.End() and Anim2.End()) do Twist.AndShout();
  whenever ?.MouseClick() do Twist.AndShout();
endsequencetable
```

Hvis tabellen skal deaktiveres når begge animationer er slut, kan man skrive:

```
sequencetable ACST
  do A1.Play(); A2.Play();
  when (Anim1.End() and Anim2.End()) do Twist.AndShout(); endsequence(ACST);
  whenever ?.MouseClick() do Twist.AndShout();
endsequencetable
```

hvorved tabellen så at sige deaktiverer sig selv!