BusyObjects Tutorial v1.0

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

2003/03/19

This work is freeware and must not be used as part of any commercial products without written permission of the author. Results produced with this work contain code and data that are part of the work; therefore results have the same restrictions as the original work and must not be used in commercial products without permission. Contact the author at brf@brainon.ch for more information.

Table Of Contents

TABLE OF CONTENTS **2**

TABLE OF FIGURES **3**

INTRODUCTION **4**

GENERAL REMARKS ON ANIMATION 4
- *Time code* *4*
- *Continuous animations* *5*
- *Staged animations* *5*

CREDITS **5**

PREREQUISITES **6**

OVERVIEW **8**

STEP BY STEP TUTORIAL **9**

SETTING UP GMAX 9
STEP 1: CREATE THE ANIMATION IN GMAX 10
- *Create a placeholder for the animated object* *10*
- *Create placeholders for obstacles and scenery objects* *10*
- *Create the path for the moving object* *11*
- *Importing a path for the moving object* *12*
- *Editing / Modifying the path* *13*
- *Setting up animation parameters* *13*
- *Connect the animation path to the object* *14*

STEP 2: EXPORT THE RAW ANIMATION DATA 16
STEP 3: COMPILE THE ANIMATION WITH *BOC* 17
- *Creating a parameter file* *17*
- *Running the compiler* *18*

STEP 4: PREPARING AN OBJECT FOR ANIMATION 18
- *Create a gmax model and save it as a library object* *19*

STEP 5: MODIFYING THE API MACRO 21
- *Insert the object rendering code* *22*
- *Using FS variables in the object rendering code* *22*
- *Known limitations and workarounds* *22*
- *Modifying the BusyObjects State Machine* *23*
- *Time code range of animation* *24*
- *The template system* *25*

STEP 6: USING THE API MACRO 27

***BUSYOBJECTS* PROGRAMMING ISSUES** **28**

BUSYOBJECTS VARIABLES 28
TIME CODE METHODS 29
- *StartSequence* *29*
- *GetSequenceCode* *29*
- *GetReverseSequenceCode* *30*

PREPARING THE FRAME FOR ANIMATION 30

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

ADVANCED STATE MACHINE EXAMPLES 31
ENHANCING THE STATE MACHINE 31
CALCULATING THE STAGE TIME CODES 32
- *For a point on the path 32*
- *Based on playback time 32*

IMPLEMENTING THE ENHANCEMENT 33
- *State 1: Idle at point 1 ..... 33*
- *State 2: Moving forwards from point 1 to point 2 ..... 33*
- *State 3: Moving backwards from point 2 to point 1 ..... 34*

SOME FINAL WORDS… 35

Table Of Figures

FIGURE 1, GMAX PREFERENCES 10
FIGURE 2, SET UP OBJECTS IN GMAX 10
FIGURE 3, SPLINE DIALOG 11
FIGURE 4, ANIMATION PATH 11
FIGURE 5, CREATING A ROAD 12
FIGURE 6, SCASM CODE FOR ROAD 12
FIGURE 7, IMPORT SCA FILE 13
FIGURE 8, EDIT PATH 13
FIGURE 9, TIME CONFIGURATION 13
FIGURE 10, TIME CONFIGURATION DIALOG 14
FIGURE 11, ASSIGN POSITION CONTROLLER 14
FIGURE 12, PATH CONSTRAINT 15
FIGURE 13, TRAJECTORIES ROLLOUT 15
FIGURE 14, TIMELINE WITH KEY FRAMES 16
FIGURE 15, *BUSYOBJECTS* ROLLOUT 16
FIGURE 16, EXPORTING THE ANIMATION DATA 17
FIGURE 17, MAKEMDL EXPORT IN GMAX 20
FIGURE 18, FSREGEN MAIN DIALOG 20
FIGURE 19, FSREGEN LIBRARY COMPILER 21
FIGURE 20, API PLACEMENT 27
FIGURE 21, PLACED MACRO 27
FIGURE 22, STAGES ON THE PATH 31

TABLE 1, TEMPLATE VARIABLES 26
TABLE 2, BUSYOBJECTS VARIABLES 29

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

Introduction

Whenever you are close to an airport – flying over it, when taxing on ground or while parking - the detail level and complexity of the scenery around you becomes an important factor in visual quality – or to put it plain: if the scenery is interesting or not.

Putting photo-realistic textures on buildings, creating people always looking at you, having trees, boats and other kind of real-world equipment can enhance a scenery. But most of these enhancements have one thing in common: They leave the scenery "as dead as a mummy" – they are static by default. But that's what you expect from so-called static sceneries, don't you?

Although there is a way to do simple animations in static sceneries, this is often very limited (like rotating a radar or moving a door) and often results in repetitive BGL code, if you want to move and rotate something around a bit smoothly. This way of doing animations is boring and error-prone

The only way out seems to be using Dynamic Sceneries, as they are especially designed to animate objects. But they have problems of their own: dynamic sceneries must not overlap, so you can only have one dynamic scenery in the same spot. This makes combining different sceneries often impossible. Dynamic sceneries are designed for covering a whole airport or area, but are not feasible for "local" action. They may be great for moving and flying aircrafts around your airport, or moving trains and trucks around - but if you want more, you're stuck.

BusyObjects will help you overcome these limitations. It is a technology that exploits a new way of animating objects in static FS2k+ sceneries. It allows you to play animations at varying speeds, begin (and end) animations at any time, reverse animations and have complex path definitions for the animated object. Since the object is rendered at different positions for (nearly) every frame, the object transformation appears very smooth.

To understand the workings and possibilities of the framework, here are some words about animation in general:

General remarks on animation

If you have all the usual equipment placed at an airport, it is quite likely you are already using animated objects in the static scenery. A "wind sail" for example is a kind of environment driven animation (rotates with the wind and moves with wind speed) whereas a rotating radar or an opening door is a kind of time-code animation.

The following will not go further into the details of "wind sail"-like animations; our main focus is time-code animations. These kind of animations define a *relationship* between the *time code* of a frame being rendered and the *position / rotation* of the object being animated.

- ***Time code***

A *time code* is a numerical value that constantly increases with time. It has a step rate (units per second) and a cycle length. The cycle length can be expressed in "steps" or "seconds" (together with the step rate both definitions are equivalent). A

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

real-world example of the *time code* is “second of the day”. It has a step rate of 1 per second and a cycle length of “86400” steps (or seconds in this case).

The *time code* generator in the *BusyObjects* framework always has a cycle length of 65536 steps, but it can have different step rates: from 18 steps per second up to unbelievable 18432 steps per second. These step rates correspond to cycle length in the range of one hour and four seconds.

Whenever the Flight Simulator renders a frame, the animation engine takes the current time code to compute position and orientation of the animated object and to render the object correctly.

The relationship between time code and animation can be classified into two groups (that means time-code animations come in two different flavors):

- **Continuous animations**

are animations that cycle on forever like the rotating radar. The cycle length of the time code (in seconds) determinates the speed of the rotation, the number of steps in one cycle determinate the “smoothness” of the rotation.

- **Staged animations**

are animations like an opening / closing hangar door. I call these kind of animations “staged animations”, because the are made out of different stages that all have a beginning and an end: The animated door has the stage ‘closed’, where it is sitting around waiting for its activation, the stage ‘opening’, a transition stage where it is actually moving open and so on…

During a transition stage (when the object actually moves / rotates), you also make use of one of the time codes to perform the actual transformation of the object – just as for continuous animations. The difference is that the staged animation “starts” and “end” at any given time, so time code generation must be more flexible then in the case of continuous animations.

The *BusyObjects* framework is designed with staged animations in mind. A flexible state machine and time code generator allow complex animation behavior, as this tutorial will show.

Credits

Many thanks to all the people that were involved in beta-testing this first release of BusyObjects. A special thanks to Arno Gerretsen, Nick Witthome, Peter Augustesen and Hugues Caron for their help and support.

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

Prerequisites

If you want to create *BusyObjects*, you need at least…

-

  FS2000 / FS2002

  The API macros created by the framework make use of BGL commands not available in FS versions prior to FS2000. There is no trick to make them work with FS98 or earlier.

  "Flight Simulator" is a registered trademark of Microsoft Corp. (see http://www.microsoft.com/games/fs2002/) for further details.

-

  gmax 1.2

  The sole purpose of the *BusyObjects* framework is to compile an animation path (from a formatted ASCII file) into an API macro source code. Although you can write this animation data file yourself, it makes more sense to use an animation editor and pre-viewer. The best support from the framework in terms of import and export functionality is currently provided for gmax 1.2

  The choice of gmax as a basis for *BusyObjects* is not by chance: Aside from being an excellent freeware animation modeler, gmax is also the declared favorite model editor for all MSFS versions to come. So eventually more and more scenery designers will know how to work with gmax and will have a less steep learning curve for the *BusyObjects* framework.

  There are a lot of places on the Internet already where you find tutorials for gmax. One that is specially related to FS scenery design and therefore contains information directly usable with the *BusyObjects* framework, is Arno Gerretsen's gmax page at http://home.wanadoo.nl/arno.gerretsen/gmax.html

  gmax is a registered trademark of Discreet, an Autodesk company. The application can be downloaded for free (registration required) at http://www.discreet.com/products/gmax.

-

  BOC

  The *BusyObjects Compiler* is part of the *BusyObjects* framework and includes MAXScript files to import and export animation path definitions.

-

  ASCII editor

  You can use an editor of your choice to write and modify text files. Make sure your editor is able to save plain (unformatted) ASCII files.

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

- **Scenery design application**

  Although you can compile your airport from plain ASCII files with the SCASM compiler, you normally use a scenery design program to visually assemble the airport. The API macros generated by the *BusyObjects* framework are at least compatible with “Airport for Windows” (http://www.airportforwindows.com/) and “Flight Simulator Scenery Creator” (http://www.fssc.avsim.net/). If you are using another application and have success or failure in placing the API macros, please let me know.

- **fsregen**

  If you use gmax to create the object to be animated (and not only the animation path), you will find fsregen a useful tool to create library objects out of gmax models.

  You can download the tool at http://www.nhreas.com/fsregen.html.

Aside from all the software, you need some basic skills to work with the *BusyObjects* framework. I have to admit that the current version of the framework is only usable by experienced BGL/SCASM coders and is definitely nothing for beginners. You should be able to modify and customize API macros, have a fair knowledge of 3D geometry and math and some enthusiasm to go through the whole process.

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

Overview

To make the most out of the *BusyObjects* framework, it is important to understand the basic concepts on which it is build. The framework strictly separates between *animated object* and *animation path*:

An *animated object* is a visual model of a vehicle or another moving object. It is static by itself, which means it has fixed coordinates within a coordinate system (frame of reference). So all objects modeled for scenery design can be used as *animated object*s.

An *animation path* is a non-visual (abstract) curve. *Animated objects* can travel along this curve (moving/rotating). The *animation path* is independent of the *animated object*; indeed the same *animation path* can be used to animate different objects.

The framework actually deals only with *animation paths*; it is up to you to provide the *animated object,* which can be a library object or SCASM source code.

The BusyObjects framework helps you to define an *animation path* and to compile the path into an API macro. This compiled macro is incomplete; you need to insert the *animated object* by hand before you can use the macro in scenery design.

The link between an *animation path* and an *animated object* is very important: The *animation path* moves and rotates the origin of the frame of reference for the *animated object*. This requires the animated object to be modeled around the origin so that rotations around the origin look “naturally” for the object. Detailed information about this important requirement is given in its own chapter (**Modeling an object for animation**).

The BusyObjects framework purpose is to compile an animation path (from a formatted ASCII file) into an API macro. To create the needed animation data, you can write this file yourself (a tedious job *) or use gmax to define and edit the animation path. The *BusyObjects framework* includes MAXScripts to import a SCASM road definition as an animation path and to export animation data defined in gmax as input data for the framework compiler *BOC* (*BusyObjects Compiler*).

The compiler inserts BGL code and data templates (these make up the *BusyObjects* kernel routines) into every compiled API macros. The kernel routines perform all tasks related to animations (“*time code generation*” and “*animation engine*”). The compiler uses the templates and the animation data to emit API macros that can be compiled with SCASM 2.88.

*) The “proof-of-concept” for the framework actually involved defining the complete path manually. Some sheets of paper with grids on them were all I used for the very first animation. I later switched to AutoCAD to construct the path and used a custom made ARX module to export the animation data.

When I decided to make the framework public I soon realized that using AutoCAD might not be a good choice because its not commonly used among scenery designers. So I looked around for another program where you can define animations that can be converted into *BusyObjects*. I found gmax to be the ideal “solution” for that. It is an animation editor by design and has a lot of interesting features I wouldn’t dare to implement myself. It eased the task of getting the framework usable so much!

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

Step by Step tutorial

This tutorial will guide you through all the steps necessary to create a *BusyObjects* animation. As an example we will animate a vehicle to move along a given path. After completing the tutorial you can easily use the example as a guideline for animating your own model. I highly recommend that you follow the tutorial at least once as close as possible before trying to create your own animation.

You must have the framework installed before you can successfully work through this tutorial. Installation instructions can be found in the *BOC* manual. I also recommend to create a `<BOC>\projects\Tutorial` directory and to use that for the project files.

The following steps are described in detail; they are listed in a sequence you normally use to create an animation:

1. **Creating an animation path in gmax**

   Gmax is used to create the path of movement for a placeholder object. You use gmax features to create and edit the needed key-frame animation and script files to import and export animation data in and out of gmax.

2. **Compiling the animation with BOC**

   Explains the necessary parameters for compiling an animation and how to run the compiler.

3. **Modeling an object for animation**

   Explains where to place the origin “(0,0,0) coordinate” in the model and how to align the model along an axis.

4. **Completing the API macro**

   Insert object to be animated: Modify the macro to call (or draw) the object you want to animate.

   Modify the animation start/end triggers: Change the behavior of the animation by defining start / stop conditions

5. **Using the animation in a scenery**

   Shows how to use the completed macro in scenery design.

Setting up gmax

It is assumed that you have at least some knowledge in working with gmax, so not all settings or even caveats can be mentioned here. Please read the gmax documentation (manuals and tutorials; you can find quite a lot about using gmax on the internet).

One important remark on scaling: You can work in any scale in gmax – that means one gmax unit can be any real world length. But if you are creating a *BusyObjects* animation you have to set the scaling to “1 unit = 1 meter”. The framework expects coordinates in the animation to be in meters.

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

Go the menu option “Customize ► Preferences”; this opens a tabbed dialog where you can (permanently) define gmax settings. Active the “General” tab:

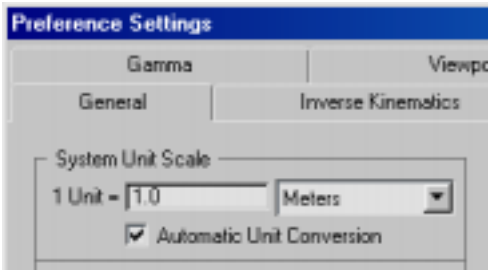


Figure 1, gmax preferences

If you use gmax to create the model as well, you can use different scale for that, but the animation must be defined as described.

Step 1: Create the animation in gmax

- ***Create a placeholder for the animated object***

Start gmax and create a placeholder of the object you want to animate or you use the object directly. I prefer using a placeholder because it’s faster and easier for me. It can help to have a placeholder where you can check not only position, but orientation as well.

- ***Create placeholders for obstacles and scenery objects***

Depending on the complexity of the movement you may want to model placeholders for everything that is placed in that part of the scenery where the animation takes place. You don’t want the animation to “collide” with any other part of the scenery, do you? So a simple set-up may look like this:

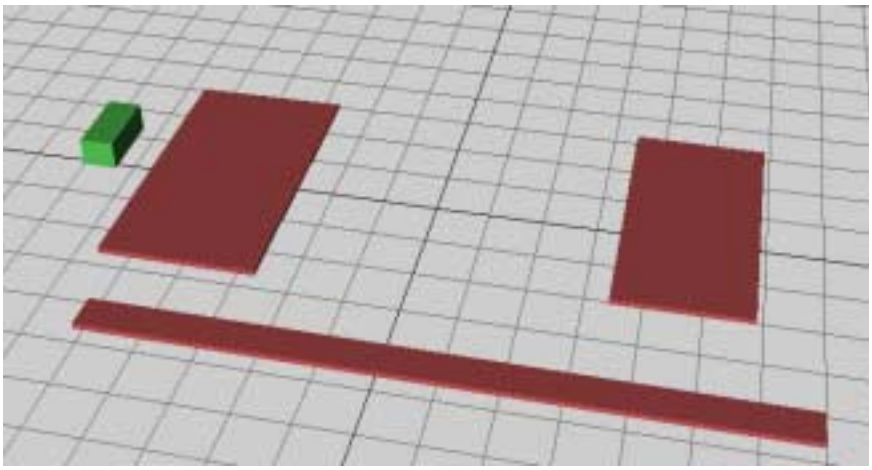


Figure 2, set up objects in gmax

The little green block represent the object we want to animate (a vehicle) and three red buildings (just flat rectangles) we don’t want to collide with.

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

- ***Create the path for the moving object***

There are certainly many ways to set up the path for a moving object like a vehicle or so. Gmax supports many ways of animating objects, so this tutorial will pick one of these many methods and uses that to create the animation:

We want a “looped” path on the ground (one that is closed; start and end point are at the same position and that lays in the x-y-plane (z = 0)) and we also want the object to “look” along the path of movement (this is the most realistic way for vehicles to move).

To achieve this, we define the path the object will move by drawing a spline. In gmax enable the “Create ► Shapes ► Splines” dialog:

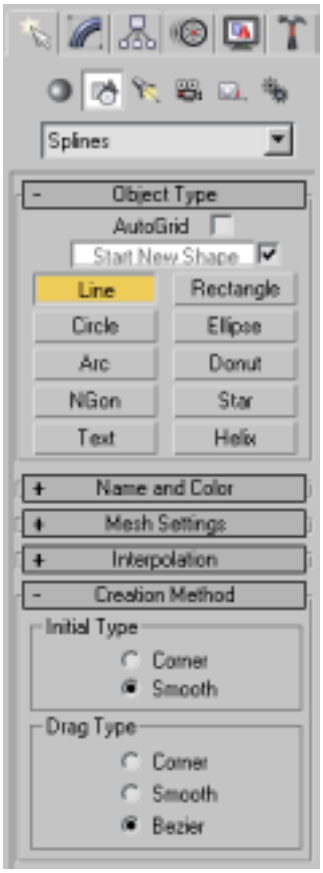


Figure 3, Spline dialog

In this tutorial we use the “Line” mode to draw the path; make sure you use “Smooth” as “Initial Type” and “Bezier” for “Drag Type”.

Draw the path; don’t worry if it is not exactly going the way you wanted it – you can modify it later to look “better”. It makes no sense to drop a line, if one or the other segment is “wrong”. It is easier to modify an existing path then to create a “correct” one in one draw.

Close the path by placing the last point of the curve onto the first point. gmax then asks if it should close the curve. Accept that with “OK”. The scene may look like this now; the blue curve is the defined spline:

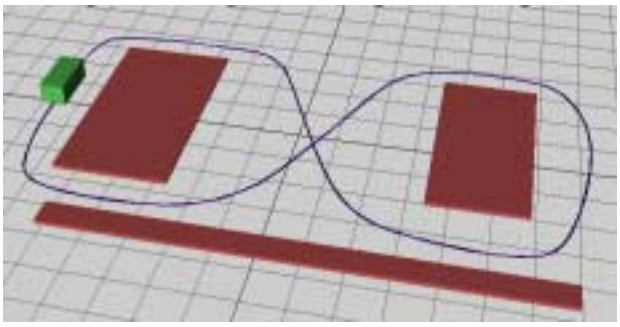


Figure 4, Animation path

The curve is made up of vertices; these are the points you have placed while creating the curve. If you don’t want the object to move on the curve with constant speed (e.g. if you want to slow down, even stop and then accelerate), you should take care of the vertex spacing. The default rule is that the object will always move between two vertices in constant time. This means that the speed of the object depends on the distance of two adjacent vertices: Is that distance small, the object travels slower that between far apart vertices.

You can later change the speed of the object, but it makes editing easier if you have that rule in mind when you create the curve.

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

- ***Importing a path for the moving object***

If your scenery is quite complex and there are many obstacles in the area you want the object to move, it can be hard to reconstruct all objects in gmax. Therefore there is a method to define the curve outside of gmax and then to import that curve as a starting point and process it further in gmax:

Load your scenery into a scenery editor (in this example I use “Airport for Windows” for that purpose). You can now draw the desired path of movement as a “road” in the scenery. In the canvas press the right mouse button and select the “Create Road” option. A dialog box pops up where you specify options:

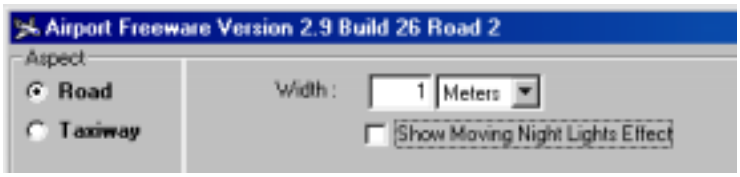


Figure 5, Creating a road

I use a narrow road (1 meter or smaller) so it is not taking up so much space.

You now draw the road (just as usual). Keep in mind that the same rules as for gmax created curves apply: The relative speed of movement on the path is directly linked to the length of a segment. Smaller segments are “slower” than long ones.

If you are done with the curve, select a vertex of the road (or the entire object) and press the right mouse button again. Select the “Edit selected object” option and press the “SCASM code” button at the bottom of the dialog box:

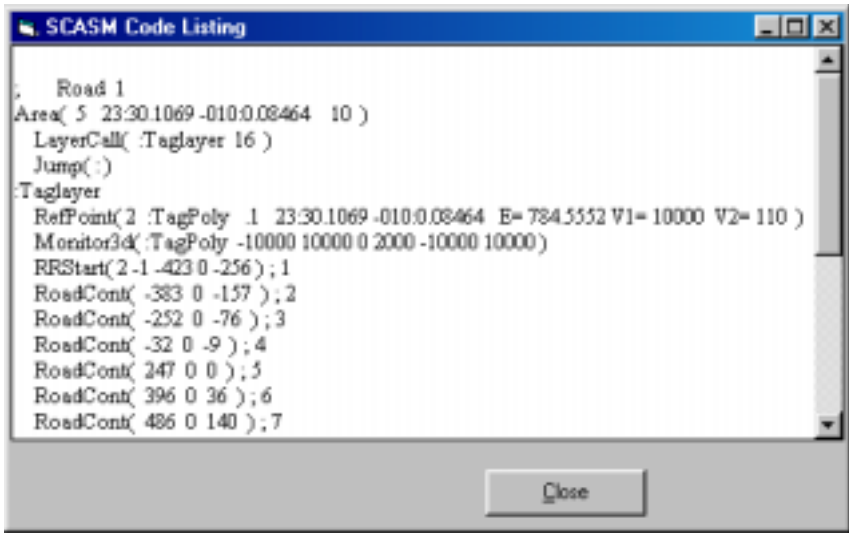


Figure 6, SCASM code for road

Open an ASCII text editor and create a new file named `example.sca` in the project directory. Then select and copy all text in the “SCASM Code Listing” window that pops up and paste it to the file in the editor. Save the file.

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

Figure 7, Import SCA file

To import the path into gmax, select "MAXScript ► Run Script" from the menu. Browse for the script file `BO_ImportSCA.ms` that is located in the directory `<BOC>\gmax` by default. The script pops up a dialog (see image on the left).

Use the "Browse" button to select the file `example.sca` in the project directory and press "Import" to start the import process.

If everything went well, you will see the imported path in the gmax model window. If not, something went wrong and you can have a look on the import protocol by pressing "F11" (this pops up the gmax message listener).

The imported path is already spline'd with a smooth curve as needed for a smooth animation.

Note to developers: You can use the script file `BO_ImportSCA` as a starting point for your own import scripts that import path data defined somewhere else. This allows you to define a path by different applications and to "fine tune" the movement on the path in gmax. That's much easier than to provide a complete set of animation data for the compiler (although you can do that too).

- ***Editing / Modifying the path***

If the created path needs modification, you can modify the vertices of the path curve. Select the curve and switch to the "Vertex" entry.

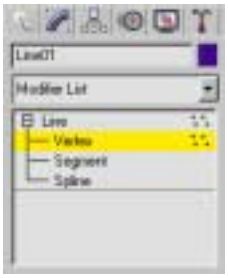


Figure 8, Edit path

All vertices now show up as a little white crosses. You can move these vertices around to give the curve the desired shape.

Keep in mind that the speed of the object on a segment depends on the length of a segment.

You can also delete selected vertices or insert new vertices (Method "Insert" in the "Geometry" rollout) if you need to.

If you have imported the curve from an external application, you might also want to close the curve (in case you have a closed loop). Simply move one endpoint of the curve over the over; gmax will then ask you if it should close the curve.

To end the editing of vertices, select the "Line" entry in the tree (or close the "Line" branch). The editing mode for vertices is terminated.

- ***Setting up animation parameters***

Figure 9, Time configuration

We now have to set up the animation parameters. Click on the "Time Configuration" button in the right bottom icon list:

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

The dialog box that comes up allows you to define the animation length. We will use a value of **200** in the length field in this tutorial.

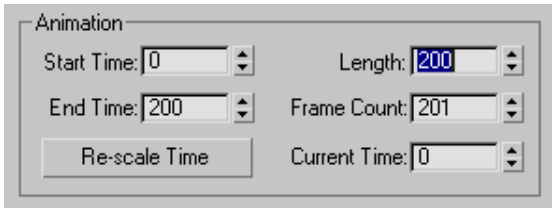


Figure 10, Time configuration dialog

In gmax the number of frames in an animation – together with the playback frame rate – defines the duration of the animation.

So an animation with 100 frames played back with a frame rate of 25 frames/sec takes 4 seconds.

For the *BusyObjects* framework the value of "number of frames" has a completely different meaning (the duration is set as a compiler parameter): it specifies the number of points used to re-sample the animation.

The *BusyObjects* framework can't handle gmax animation definitions directly, but relies on a number of equally spaced (in time) points in the animation, where the objects position and orientation is known. Between theses points the framework performs a linear interpolation. So if your curve is bending a lot, you might need to increase the number of re-sampling points; if your curve bends less, you can decrease the value.

There is no good rule of thumb to estimate the number of steps needed. But you can check a setting by single stepping through an existing animation (at least through the parts where the curve bends most): If you think the change from one frame to another is too large, you should increase the value and check again. With time you will learn to correlate between curve bending and number of re-sampled points. This is helpful, because changing the number of frames later is a bid tedious in some cases especially if you are using key-frame animations. Because we will use a constraint animation in our example, changing the value later on makes no problem at all.

All other parameters that can be set in the dialog have no influence on the animation created by the *BusyObjects* framework. So twiddling around with these settings doesn't change anything in MSFS playback.

- ***Connect the animation path to the object***

We can now link the path and the object, so that the object will be animated along the path. The procedure is to *link the path to the object*, not the other way round (you can't link the object to the path). This makes sense, since we want the object to move along the path and not vice versa.

So select the object to be animated (or its placeholder) and activate the "Motion" tab on the right dialog bar.

Figure 11, Assign position controller

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

Open the “Assign Controller” rollout and select the “Position” entry. Click the “Assign Controller” button above the tree and select “Path Constraint” in the dialog box that appears. Press “OK” to use the selection.

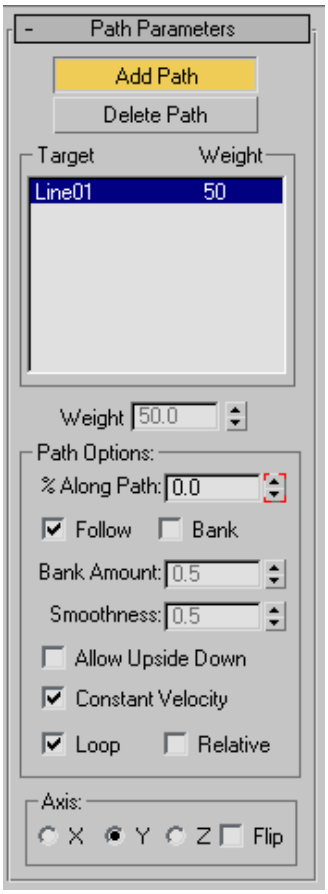


Figure 12, Path constraint

The “Path constraint” means we will not define positions by hand but want the object to move along a given curve (the one you have just drawn). Open the rollout “Path Parameters” and press the “Add Path” button; then select the curve in the active viewport.

You can see the result so far by paying back the animation:

You have certainly noticed that the object moved along the path but doesn’t change orientation while moving. We need to set a constraint on the orientation to follow the path as well.

We can do this by activating the “Follow” checkbox in the “Path Parameters” options. This will not only define the position of the object but will orient the object along the path as well.

The “Axis” selection specifies, which axis of the object should follow the path. Our object is oriented along the Y-axis.

You can specify different other options as well. The most important for us (and therefore checked) is the “Constant Velocity” option. This makes the speed of the object independent of curve segment length and things a lot easier.

You are now ready for a final check on the animation; play back the animation in a loop and possibly single-step through the “critical” frames. If everything is fine so far, save the model and the animation just created.

The last step is to convert the constraint animation into a key-frame animation (only key-frame animations are exportable; if you export now, you will loose the orientation information). To start the conversion, do the following:

Select the object to be animated and activate the “Motion” tab on the right dialog bar. Select the “Trajectories” mode and open the “Trajectories” rollout:

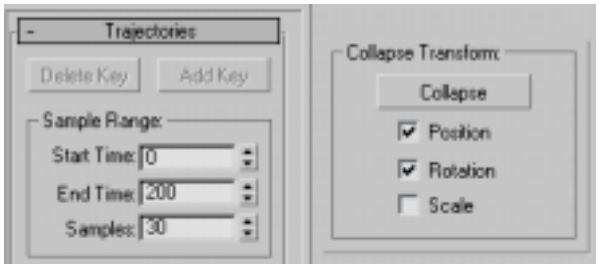


Figure 13, Trajectories rollout

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

Make sure you select the correct values in the “Sample Range” dialog for our example, so “Start Time” is 0, “End Time” is **`200`** and “Samples” is **`50`**. This “Sample” value has nothing to do with the re-sampling of the path done by the *BusyObjects* framework, but is a separate setting: It defines the number of key-frames that will be inserted into the specified animation between “Start Time” and “End Time”. So its value tells how well the key-frame representation will “match” the animation path. The more bended your path definition is, the higher the “Samples” value should be. This value should be equal to or greater than the number of vertices to define the path.

You complete the conversion by selecting “Position” and “Rotation” in the “Collapse Transform” section and then clicking the “Collapse” button. Now the specified number of key-frames are generated; you can see the key-frame positions in the timeline; key-frames show up as little red “needles”:

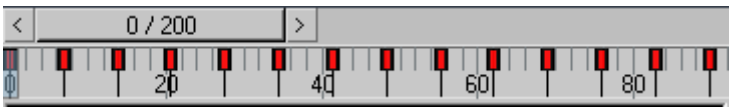


Figure 14, Timeline with key frames

Step 2: Export the raw animation data

Now it is time to export the model. Select “MAXScript ► Run Script” from the gmax menu and browse for the script file `BusyObjects.ms` that is located in the directory `<BOC>\gmax` by default. The script pops up a dialog:

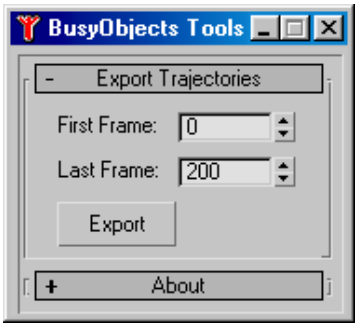


Figure 15, *BusyObjects* rollout

Now select the object to be animated and the “Export Trajectories” section let you define the first and last frame to be exported. We want it all, so we specify “First Frame = **0**” and “Last Frame = **200**”. Click on the “Export” button to start the export.

Gmax has been stripped of this file writing capabilities, so exporting the data is actually handwork. When the export has finished, you see nothing: no file generated, no dialog boxes or anything else. The reason is that gmax has logged our output in a listener window. Press “F11” to display that window:

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

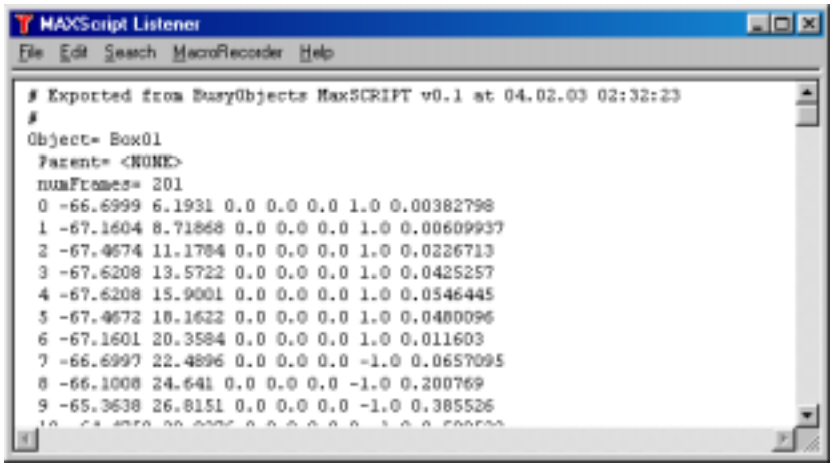


Figure 16, Exporting the animation data

Open an ASCII editor and create a new file `Object1.gani` in the project directory. This file will receive all the output from the MAXScript Listener. Copy and paste the data from the listener window into your text file. You can only copy 10kB at a time (another gmax limitation), so you have to copy data in chunks of 220 frames or so. When all data is copied, save the file and close the editor. Congratulations! You have just exported your first *BusyObjects* animation.

Step 3: Compile the animation with *BOC*

- ***Creating a parameter file***

A BOC parameter file controls the compiler, so we have to create such a file first. Open an ASCII editor and create a new text file `Object1.bod` in the project directory. The file should contain the following lines:

```
# format of animation import
ImportFormat=gmax

# file name of import data
ImportFile=Object1.gani

# list of invariant degrees-of-freedom
Invariants=ZPB

# type of animation
Type=1

# desired length of animation (in seconds)
Duration=324

# animation scale
Scale=0.01

# model scale
ModelScale=1.0

# format for animation export
ExportFormat=API
```


Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

```
# file name of export data
ExportFile=Object1.API

# file name of export template
ExportTemplate=Object1.tpl
```

The entry for an ExportTemplate is optional; if you do not specify a template file here, the compiler will use a default template. More information on the template mechanism is presented in the following chapter.

For a more detailed explanation of the options see the BOC manual.

- ***Running the compiler***

Open a shell and change into the project directory. Start the compiler with the following command:

```
<BOC>/BOC Object1.bod
```

The compiler now compiles the animation from raw data and creates the API macro in the specified file. You have to modify and complete this macro first before you can use it in “Airport for Windows”.

Step 4: Preparing an object for animation

The *BusyObjects* frameworks sole purpose is to define an animation path and to transform this path into compilable SCASM source code. It does not support you in creating an object to be animated.

But nevertheless the framework defines some requirements that must be met by the model - or the animation can look somewhat “funny”. Flying or diving cars, trucks moving backwards or even upside down – all this can happen, if the model is misplaced in the coordinate system.

It is important to understand that **the framework is rotating a coordinate system and moving its point of origin** along a defined curve. So it is not moving the object directly but it transforms the origin of the coordinate system in which the object is rendered. Since the position of the object within that coordinate system is fixed, it appears to move when the origin of that coordinate system is moved. This also means that the framework can move only rigid objects (that are objects that don’t move in themselves but keep their form all the time).

Sounds as complicated as Einstein’s “Special Theory of Relativity”? Well, there are some striking similarities, but in the end this is a lot easier – as long you are not moving your object with a velocity close to that of light, of course ;-)

For every object you want to animate you have to find a point in the model that will actually be moved along the animation path. Since this point is also the point used to rotate the model, you must take some care to find a good one or the animation will look somewhat “strange”. If you have identified that point, move the complete model so that that point is located at the origin (0,0,0). Align the model along the positive x- or y-axis (your choice).

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

I will give you some hints on what to consider; as an example I will move a truck around. The theoretical approach would be to use the center of gravity (COG) of the object, but the rendering engine in any way does not match that degree of realism. So it is much easier to find that point (let's call the point ***P***) by reasoning:

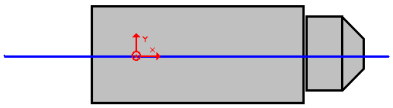
(The following images show a top-down view of the truck model. The point of origin is depicted by a symbol showing the direction of the x-, y- and z-axis)

The first image shows the truck that is aligned along the x-axis and placed somewhere in the coordinate system.

We will find the location of point ***P*** (represented by x,y,z coordinates) by the following reasoning:

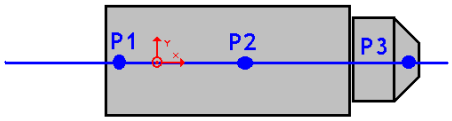
The height z can be set to 0, since the object is a ground vehicle and will move on the ground (it is not supposed to fly).

The y ordinate should be placed on the symmetry axis of the vehicle. This rule should be followed by all objects that have a left/right symmetry – that means it makes no difference if the object turns left or right; one movement is the mirror of the other.



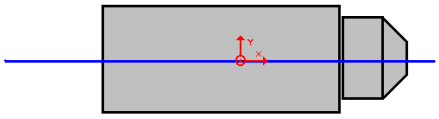
So by reasoning we have found two ordinate values (y and z), so we only have to find the last one: the x value of point ***P***.

To find a good x value, we have to rotate the shape around different x values and decide which one looks best. I will demonstrate this with three setting:



If we rotate the model around P1, the front part is moving a lot even for small angles ("wagging the front"). We have the same problem with P3; small angle let the back part of the vehicle move a lot ("wagging the tail"). Rotating around P2 gives the best results, as the movement related to rotation is equally split between front and back.

So we now know where to place the origin of the rendering coordinate system for our track:



Although this reasoning can be applied to most vehicles there may be individual variations; a tractor with front mounted equipment may require the origin to be in the back part of the overall shape to give a good animation.

- ***Create a gmax model and save it as a library object***

We now have an animation but nothing to animate – we need a model for that. I recommend creating the model with gmax also.

As said before, you can use any scale to create your model. Later on you will have to specify the scale of the model relative to 1 meter. So if you are using

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

the setting “1 unit = 1 feet” to construct the model, your scale factor is 0.2982 because 1 feet is 0.2982 meter.)

If you create the model, take care of where the origin is located and how the object is aligned with axis as explained in the previous chapter.

You export the model using the “MakeMDL export” plug-in (File ► Export ► Flightsim BGL file). Name the export file “`Object1.bgl`”.

Make sure you are enabling “*Keep files*” in the “Options” tab (The *.asm files are needed by *fsregen* to create the library objects).

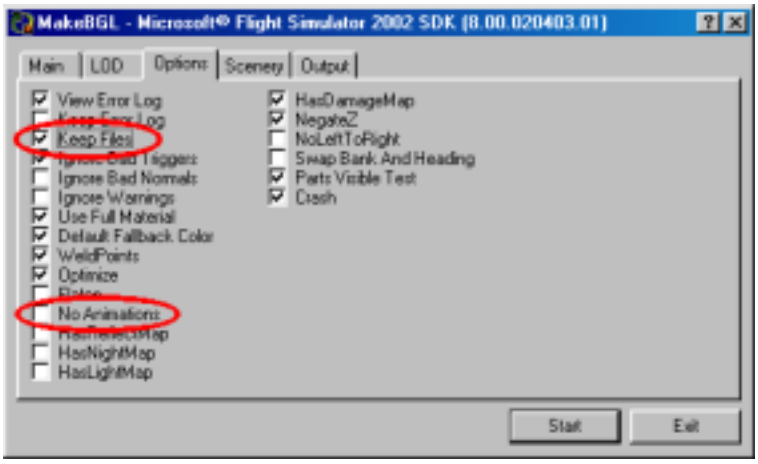


Figure 17, MakeMDL export in gmax

If the object has an animation defined with it, make sure you are NOT exporting this animation data in this step! You have to enable “*No Animations*” in the “Options” tab, if you have an animation defined with the object. Exporting the animation data is a separate step.

After the export the project directory contains the files “`Object1.asm`”, “`Object1_0.asm`” and “`Object1.bgl`”. You can delete “`Object1.bgl`”, that file is not needed any more.

Use *fsregen* to convert the output file(s) into library objects and to add them to a BGL library file:

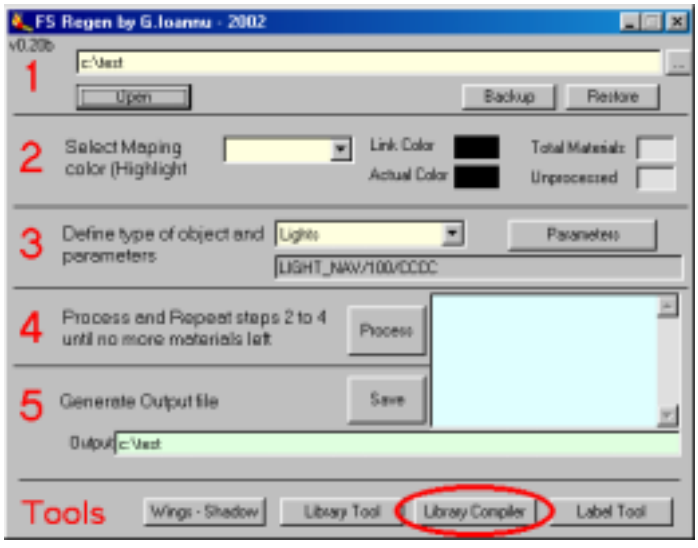


Figure 18, FSRegen main dialog

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

Use “Library Compiler” to create the library: As “Output file” specify “<PRJ>\TutLib.asm”. Browse for the “Object1.asm” as “New Object”. Assign a GUID (“U” means “unique”!!) to the object (and write the number down, we need that again later). Then “Add to List” and “Create Library”:

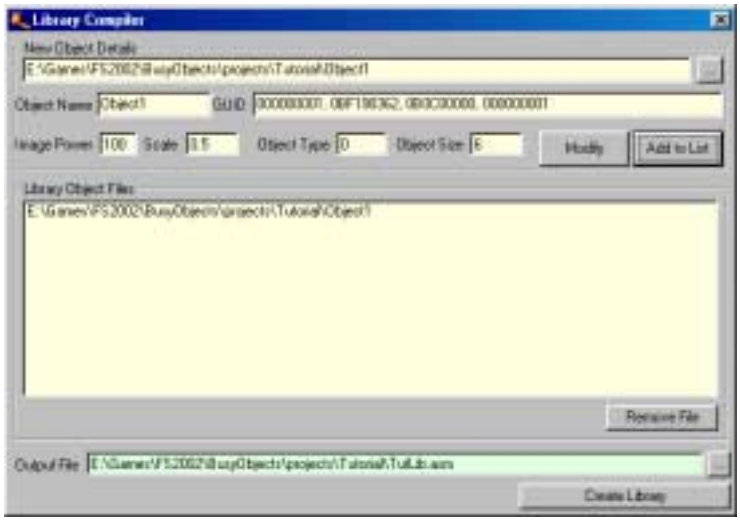


Figure 19, FSRegen library compiler

The name of the library in this tutorial is “<PRJ>\TutLib”. *Fsregen* is only producing an assembler file that needs to be compiled into a ready-to-use BGL file. So start a shell, change into the project directory and enter the following command:

```
BGLC /BGL TutLib.asm
```

If you see an error message saying “BGLC: warning: Relocations found…” - well, simply ignore it ;-)

There is a library “TutLib.bgl” in the project directory now. It contains the object we want to animate. If you want to play back the animation later in the FlightSimulator, you have to copy the library BGL file to a place where it is found by the scenery loader!

Step 5: Modifying the API macro

The compiler only processes animation data but no model information. So the generated API macro has no “object drawing code” included if you use the default template; you have to fill in this code yourself. If you don’t, you can actually run the animation – but nothing visible is moved!

This may be frustrating to very inexperienced users that can’t code in BGL, but is a relief to all others: It gives you the full flexibility of rendering your object or controlling the animation behaviour. You can modify the macro to fit your need as long as you follow some basic rules that are explained in the next sections.

To edit the API macro, load the API file into an ASCII editor of your choice.

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

- ***Insert the object rendering code***

Find the line starting with "**`:Object`**"; this is around line number 50 and looks like (bold line **(A)** not generated by the default template):

```
;================================================
;  Object to be animated.
;================================================
:Object
  Transform_Mat(
         0.000   0.000   0.000
       100.000   0.000   0.000
         0.000 100.000   0.000
         0.000   0.000 100.000
  )
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;;   insert object code HERE              ;;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(A)   CallLibObj( 0 1 BF190362 B0C00000 1 )

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  TransformEnd
  Return
```

The easiest way is to call a library object here **(A)**, so you only need to insert a single line of code. If you insert BGL code to draw the object here directly, make sure you use the scale you have defined in the parameter file.

- ***Using FS variables in the object rendering code***

If you call a library object that is using FS variables (reacts to season, weather settings and the like), you need to wrap the call to the library object like this:

```
VarBase( :[-1] )
CallLibObj( 0 … )
VarBase( :AnimData )
```

This wrapper is necessary; otherwise the object and/or animation will not work. You can use the same wrapper if you code the object rendering directly.

You can find more information on programming in *BusyObjects* and using variables in the chapter "*BusyObjects* programming issues".

- ***Known limitations and workarounds***

Some functions of the MSFS engine may behave strange when called as a library object or inside the object rendering code. The reason for this behavior is not known; it's probably to the "unconventional" use of the Transform_Mat command to perform scaling.

If the problematic command is inside a library object (and you have no access to the model or the source code); you are stuck in most cases. The only solution in this case is to use the same scale for model and animation and to delete the Transform_Mat() … TransoformEnd nesting completely.

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

If you are hand-coding the object, there might be a work-around for the problem; if you found a problematic code (and a work-around), please let me know and I will include it here:

RotateToAircraft(): This call is problematic if called inside a Transform_Mat … TransformEnd sequence. Depending on the ratio between model and animation scale this leads to more or less abrupt changes in the orientation. A work-around for this is to use the command outside the nesting:

```
:Object
 RotateToAircraft( :Obj2  0 0 0  0 0 1  0 0 0 )
 Return
:Obj2
 Transform_Mat( ... )
 ; object code
 TransformEnd
 Return
```

- ***Modifying the BusyObjects State Machine***

The *BusyObjects* framework allows you define triggers to start (and end) an animation. Since these triggers are something user-defined, the framework only generates default triggers.

The default triggers are set to:

- start the animation when COM1 frequency is set to 135.60
- stop the animation after one loop

If these defaults are OK with you, you don't have to change anything; your API macro is ready to use.

If you want to change these triggers, you have to change the code of the so-called "state machine" that is something like the animation control center. Have a look at the default implementation (as generated by BOC); it can be found directly following the "Object" section:

```
;================================================
;  Animation control (state machine).
;================================================

:StateMachine

  ; save COM1 freq in user var #1
  Low64k2Var( 1344 7BE )

:S1
  ; state = "idle" (waiting for activation) ?
  IfVarRange( :S2 18 1 1 )

       ; yes: prepare rendering at start position.
       SetVar( 1E -32768 )
       Call( :PrepareFrame )
```


Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

```
        ; waiting for activation (COM1 freq = 135.60)
(A)     IfVarRange( :SM_DONE 1344 0x3560 0x3560 )
             ; activated: switch to state "active"
             SetVar( 18 2 )
             ; start a new sequence timer
             Call( :StartSequence )
             Jump( :SM_DONE )
:S2
   ; state = "active" ?
   IfVarRange( :SM_DONE 18 2 2 )

        ; yes: render object at current time code.
        Call( :GetSequenceCode )
        Call( :PrepareFrame )

        ; this would run forever, so here is the
        ; off-switch
(B)     IfVarRange( :SM_DONE 1E 13898 32767 )
             ; de-activated: switch to state "idle"
             SetVar( 18 1 )

:SM_DONE:
   Return
```

The StateMachine section is the place where you control the animation. Sophisticated animations have quite a few states and may be even nested to allow hierarchical animations (but this is beyond the scope of this tutorial). A more sophisticated example is explained in the last section.

Just a few hints now: variable 0x18 holds the “state” of the animation. Initially set to “1”, the state machine controls the transition of states. The default implementation of the *BusyObjects* API macro has two states: “idle” (state = 1) and active (state = 2).

The **start trigger (A)** is tested while being in state “idle”. If the trigger is pulled, the state is set to “active” and the animation sequence is started. You usually change at least the start trigger for an animation.

The **end trigger (B)** is tested while the animation is running (state “active”). If the time code (variable 0x1E) has reached the last frame (here **13898**), the animations stops playing (switches back to “idle”). The value for the last frame is calculated by the compiler and should not be changed without need.

Playing around with the state machine might be a bit confusing if you have never encountered one before, but you will realize how useful the concept is when it comes to more sophisticated animations (see chapter on advanced state machines).

- ***Time code range of animation***

You might have noticed that the so-called “last frame” of our animation is not **32767** (that would correspond to a full time code range), but a smaller value – in our case this is **13898**. The reason for this is simple:

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

We have specified a desired length for our animation ("**`duration=324`**") and the BusyObjects framework tries to play the animation at the desired speed. That helps moving a vehicle at the correct speed.

Time code generation in the framework is not flexible enough to handle any desired cycle length. Instead only discreet cycle lengthes are available, so the compiler finds the smallest cycle length greater than (or equal to) the desired animation length. This means, the animation is not using all of the value range of the time code, but just a section: so it ends before reaching 32767.

If you want the animation to play at a different speed, the only way to manage this is changing the duration value in the parameter file and to recompile the macro again. (if you have already made changes to the object rendering or state machine code, make sure you save that stuff first!! The compiler is overwriting its API output file without warning!! You might consider using the template mechanism described in the next chapter.)

- ***The template system***

The template system allows you to write the object rendering and animation control code in a way that "survives" compiler runs. If you switch between modifying the macro and adjusting the animation path, it is a disaster if you didn't save your work in another file first.

To create a new template for your project, you should run the compiler on the project once. This produces an API macro from the default template.

Now copy the API macro file to a file named `Object1.tpl`. Go to the end of the state machine section and *delete all lines from that point* – this will delete the *BusyObjects* kernel code and data, the animation data, the user variable definitions and the End-of-Area marker.

Next replace all single '%' characters in the remaining file with double ones ('%%'). You can than save the file.

Edit the parameter file and add an entry for the `ExportTemplate` setting (see Step 3) referring to the file you have just created.

Re-run the compiler (that now uses your template) and check the resulting API macro for any errors. Correct errors directly in the template and repeat until everything is fine.

You can now edit the template file to change the object rendering code or you can change the state machine that controls the animation. Any changes made in the template are preserved between compiler runs.

If you want the template mechanism be able to handle varying animation definitions, you will have to use *variables* in the template to be able to adjust to different animation curves and settings. A *variable* in a template is identified by a string starting with '$' and uppercase characters following.

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

The framework defines a set of variables:

Variable	Description
$SCALE	The scale as set in the parameter file or as computed by the AutoScale function
$SRATIO	Model scale / animation scale ratio (used for “scale” correction via transformation matrix)
$HWIDTH, $HLENGTH	Half-width and Half-length of the area covered by the animation (always centred at origin). These settings are used to draw a symbol in “Airport for Windows”
$SYMPATH	This variable is substituted with the SCA code for the path representation in the symbol.
$LASTFRAME	The last frame of an animation
$VISRANGE	Compiler-calculated V2 visibility range
$USER01 … $USER10	Access to user variables (their index depends on the animation data being stored in the API and can change between compiler runs with different curves or settings).

Table 1, Template variables

Here are the changes applied to the default template; take it as an example how to use variables in templates:

```
RefPoint( 7 : $SCALE %%1 %%2 )
$SYMPATH
Points( 0
  -$HWIDTH 0 -$HLENGTH
  -$HWIDTH 0  $HLENGTH
   $HWIDTH 0  $HLENGTH
   $HWIDTH 0 -$HLENGTH
)
RefPoint( 2 :Quit $SCALE %%1 %%2 v1= %%10 v2= $VISRANGE
Transform_Mat(
      0.000  0.000  0.000
     $SRATIO 0.000  0.000
      0.000 $SRATIO 0.000
      0.000  0.000 $SRATIO
)
Low64k2Var( $USER01 7BE )
     IfVarRange( :SM_DONE $USER01 0x3560 …
     IfVarRange( :SM_DONE 1E $LASTFRAME 32767 )
```

You can also use the template system to create one (or more) generic template(s) that you use as a starting point for new projects. A generic template should include all the stuff you would normally change in the compiler generated API anyway…

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

Step 6: Using the API macro

The API macro is now ready-to-use. You can use a scenery design application (like “Airport for Windows” as used in this example) to insert the animation into your scenery. After loading your scenery into the program, select the option “Create 3D object”. This opens a dialog box:

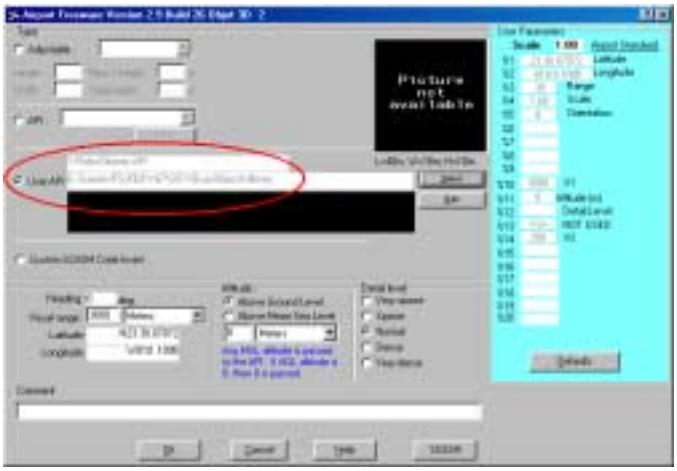


Figure 20, API placement

Select the “User API” type and press the “Select” button to locate the API macro file. Set the scale (%4) according to the parameter definition, although this value does not influence the scale of the object in FS, but for the symbol inserted into the scenery in “Airport for Windows”. If you don’t enter the correct scale the symbol representing the animation will have the wrong size. Any other parameters can be set to fit your need.

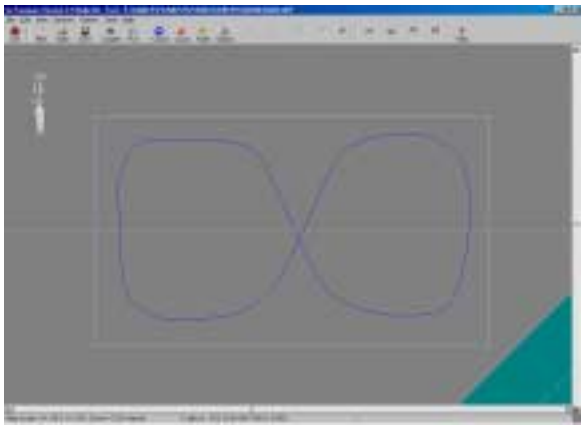


Figure 21, Placed macro

If you use the AutoScale option for the compiler, the generated API macro may have difficulties to render correctly in the design program.

***Please note**: If you want detail level to be correctly processed by the macro, you have to add that in the macro file yourself or insert that into your default template (so you have to do it only once)! This option is ignored in the default implementation.*

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

BusyObjects programming issues

The concept of state machines in the *BusyObjects* framework allows much more than just to define triggers to start (and end) an animation. The next chapter will describe some state machines used for more sophisticated animations.

Before the flexibility of the framework can be used to full extent, you will need a good understanding of the programming issues like variable mapping and time code generation in *BusyObjects* (more general remarks on time codes can be found in the introduction to this tutorial), so here are some words on programming issues in *BusyObjects*:

BusyObjects Variables

Whenever the state machine is called, it is in “local variable base” mode. This means, variables do not address the “normal” variables, but variables defined by the framework. If you want to work with FS variables in your state machine, I highly recommend that you copy these variables over to local variables at the beginning of the state machine.

The framework reserves 10 local variables for the user. These variables are listed at the end of the API macro. In the comment you find the variable index for each variable; this value depends on the actual animation data and may change for one API macro to another. **Make sure you always use only these ten variables at the specified indices; if you don’t, disaster is close…**

Copying the variables look like the line already inserted by the compiler; add more lines as needed:

```
Low64k2Var( 1344 7BE )
```

The local variable can now be used inside the state machine like the original FS variable and you don’t have to take care of variable bases.

If you want to use a FS variable directly (without copying it to a local variable), you have to make sure the correct variable base is used like:

```
VarBaseOverride( 0 )
IfVarRange( :LABEL 7BE 0x3560 0x3560 )
```

This form of access only makes sense if there is a single access to the variable throughout the whole state machine. Otherwise this approach is not recommended.

The local variable table (that is accessible by the state machine) looks like:

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

<table>
<tr><th>Variable</th><th>Description</th><th>Index</th></tr>
<tr><td>State</td><td>Holds the current state of the animation</td><td>0x18</td></tr>
<tr><td>TimeBase</td><td>Divider that specifies cycle length (in time) of the time code. The following table shows the relation between cycle length and divider:
<table>
<tr><td>1h 0' 40"</td><td>0</td></tr>
<tr><td>0h 30' 20"</td><td>1</td></tr>
<tr><td>0h 15' 10"</td><td>2</td></tr>
<tr><td>0h 7' 35"</td><td>3</td></tr>
<tr><td>0h 3' 48"</td><td>4</td></tr>
<tr><td>0h 1' 54"</td><td>5</td></tr>
</table>
<table>
<tr><td>0h 0' 57"</td><td>6</td></tr>
<tr><td>0h 0' 28"</td><td>7</td></tr>
<tr><td>0h 0' 14"</td><td>8</td></tr>
<tr><td>0h 0' 7"</td><td>9</td></tr>
<tr><td>0h 0' 4"</td><td>10</td></tr>
</table>
The divider is computed by the compiler and preset in the API macro. You normally don’t (have to) change this setting.</td><td>0x1A</td></tr>
<tr><td>TimeCode</td><td>Current time code value used to render the frame</td><td>0x1E</td></tr>
</table>

Table 2, BusyObjects variables

A good advice: Don’t try to mess around with other variables in the variable table. All that can happen is that nothing happens anymore ;-)

Time code methods

There are three methods provided by the time code generator:

- ***StartSequence***

The StartSequence method is used to start a new animation sequence. The initial time code of the sequence is specified as a parameter. This parameter must be set in time code variable 0x1E before the method is called:

```
SetVar( 1E 16422 )
Call( :StartSequence )
```

- ***GetSequenceCode***

The GetSequenceCode method is used to retrieve the current time code of the sequence. You must have started a sequence with the “StartSequence()” call before or the result of this call is undefined.

The time code is returned in the time code variable 0x1E:

```
Call( :GetSequenceCode )
IfVarRange( :LABEL 1E 14000 14010 )
```


Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

You normally call this method at the beginning of a „running the animation sequence“ state.

- ***GetReverseSequenceCode***

The GetReverseSequenceCode method is just like the “GetSequenceCode()” call used to retrieve the current time code of the sequence – but this time counting down instead of up. Again you must have started a sequence with the “StartSequence()” call before or the result of this call is undefined.

The time code is returned in the time code variable 0x1E:

```
Call( :GetReverseSequenceCode )
IfVarRange( :LABEL 1E 14000 14010 )
```

You use this method at the beginning of a „reversing animation sequence“ state. This allows you to play animations “backwards”. The result is an exact mirroring in time, so this is not really realistic for vehicles (these normally turn and drive back instead of using a “reverse” gear) but allows nice animations if objects like doors or so are involved.

Preparing the frame for animation

Every stage should call the “PrepareFrame()” method to ensure the object is rendered at the correct position and orientation for the frame. The method expects the time code to be used in local variable 0x1E; it is normally called after a “GetSequenceCode()” or “GetReversedSequenceCode()” invocation (that set variable 0x1E correctly) but you can call it with any time code as well:

```
SetVar( 1E -32768 )
Call( :PrepareFrame )
```


Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

Advanced State Machine examples

We will use the same animation data as before, but we will make the behaviour of the animated object a bit more complex (and interesting ;-)

Enhancing the state machine

We define three points in the animation (see image below): Point 1 is the start/end point of the animation as we have defined it. At points 2 and 3 we want the animation to stop.

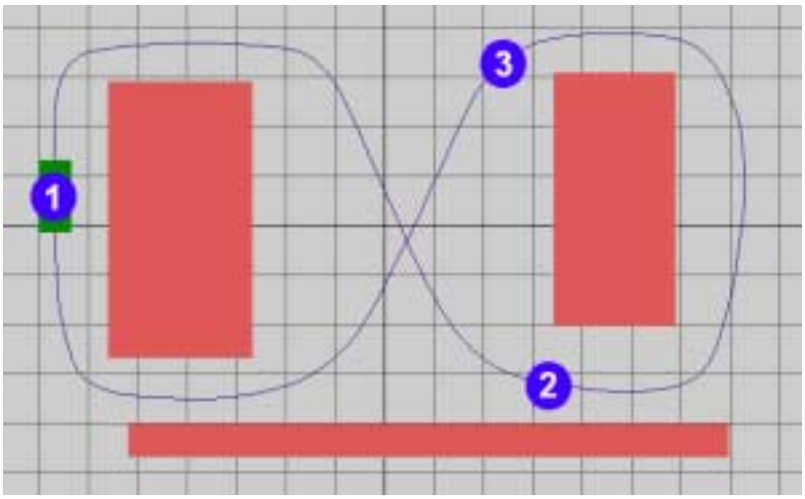


Figure 22, Stages on the path

To make things even more interesting we define the following behaviour for the object:

Object is at point 1:
    COM1 = 135.60?
        Object moves forward to point 2

Object is moving forward between 1 and 2:
    COM1 <> 135.60?
        Objects moves backwards to point 1
    Point 2 reached:
        Stop

Object is moving backward between 1 and 2:
    COM1 = 135.60?
        Objects moves forwards to point 2
    Point 1 reached:
        Stop

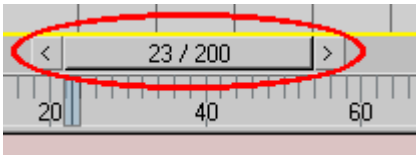
Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

Sounds a bit confusing? It's as simple as that: The COM1 freq defines how the object moves between point 1 and 2: If it is set to 135.60, the object moves forward; if it is set to another frequency, the object moves backwards. By switching frequencies you can switch the direction the object moves!

Calculating the stage time codes

We sure have to know the time code values for the points where the animation stops. How to get these values depends on its definition:

- ***For a point on the path***



Restart gmax and load the model used to create the animation. Use the frame slider (see left) to move the object to the points where you want some "special" action to happen. Write down the frame numbers $\mathbf{f_1}$, $\mathbf{f_2}$, $\mathbf{f_3}$,….

You now have to calculate the time code using the following formula; let **f** be the frame number of the point, **tl** the time code for the last frame (computed by the compiler; in our example equal to **13898**), **fl** the number of the last frame in the gmax animation (in our example this is equal to **200**) and **tc** the resulting time code:

tc = ((f-1) * (tl+32768) / (fl-1)) - 32768

If we want the time code for frame 23, we get:

tc = ((23-1) * (13898+32768) / (200-1)) - 32768
= (22 * 46666 / 199) - 32768
= 5159 – 32768
= -27609

Make sure your result is between –32768 and the frame code for the last frame; if it is not, you should recalculate the time code.

- ***Based on playback time***

You can also calculate the time code for a point a defined time into the animation. For example you can suspend the animation for a while when the object has moved for 20 seconds.

To calculate the time code for a "real time" (playback time of the animation), you can use the following formula; let **t** be the "real time" of the point in seconds, time code for the last frame (computed by the compiler; in our example equal to **13898**), **d** the duration of the animation in seconds (as defined in the compiler parameter file; in our example the value is 324) and **tc** the resulting time code:

tc = (t * (tl+32768) / d) - 32768

If we want the time code for t = 20 sec, we get:

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

tc = (20 * (13898+32768) / 324) - 32768
= (20 * 46666 / 324) - 32768
= 2880 – 32768
= -29888

Make sure your result is between –32768 and the frame code for the last frame; if it is not, you should recalculate the time code.

Implementing the enhancement

Let's have a look at the implementation of the state machine that realizes such behavior (this is just a fragment of the complete state machine; I have included the ready-to-use macro Y-COMPLEX.API in the library directory of the installation. There are different frequencies that control the behavior on the other two segments: 133.20 between 2 and 3, 134.80 between 3 and 1):

- ***State 1: Idle at point 1***

```
    IfVarRange( :S2 18 1 1 )
        ; yes: render object point 1.
(A)     SetVar( 1E -32768 )
        Call( :PrepareFrame )
        ; waiting for activation (COM1 freq = 135.60)
        IfVarRange( :SM_DONE 134A 0x3560 0x3560 )
            ; activated: switch to "forward 1 to 2"
            SetVar( 18 2 )
            ; start sequence
(B)         Call( :StartSequence )
            Jump( :SM_DONE )
```

While being in state 1, the object sits around and is not animated at all. So we don't need a generated time code to render the object; instead we can set the time code directly that corresponds with point 1 **(A)**. Since point 1 is the starting point, the associated time code is the initial time code value (-32768). After setting the time code we prepare the frame for rendering.

When the object is activated (COM1 frequency = 135.60), we switch state and start a new sequence **(B)**. Remember that the start method requires the start time code to be set in variable 0x1E. In our case that value is set correctly with code fragment **(A)**.

- ***State 2: Moving forwards from point 1 to point 2***

```
    IfVarRange( :S3 18 2 2 )
        ; yes: render object at current time code.
(A)     Call( :GetSequenceCode )
        Call( :PrepareFrame )
        ; sequence completed?
(B)     IfVarRange( :S2A 1E -10536 32767 )
            ; yes: switch to state "idle at point 2"
```


Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

```
            SetVar( 18 4 )
            Jump( :SM_DONE )
:S2A
        ; still activated (COM1 freq = 135.60)?
        IfVarRange( :S2B 134A 0x3560 0x3560 )
        Jump( :SM_DONE )
:S2B
            ; no: switch to state "reverse 1 to 2"
(C)         Call( :StartSequence )
            SetVar( 18 3 )
            Jump( :SM_DONE )
```

While being in state 2, the object moves forwards to point 2. We prepare the rendering with the sequence code for this frame **(A)**.

Because this is a transition state (we are actually moving), we have to check if the end of the sequence is reached. This is done by checking the current time code (variable 0x1E) **(B)**.

When the COM1 frequency Is tuned away from 135.60 while moving towards point 2, we switch state and start a new sequence **(C)** for the reverse movement. The parameter for the start method (variable 0x1E) is already set to the correct value with code fragment **(A)**.

- ***State 3: Moving backwards from point 2 to point 1***

```
    IfVarRange( :S4 18 3 3 )
        ; yes: render object at reversed time code.
(A)     Call( :GetReverseSequenceCode )
        Call( :PrepareFrame )
        ; sequence completed?
(B)     IfVarRange( :S3A 1E -32768 -32750 )
            ; yes: switch to state "idle at point 1"
            SetVar( 18 1 )
            Jump( :SM_DONE )
:S3A
        ; activated (COM1 freq = 135.60)?
        IfVarRange( :SM_DONE 134A 0x3560 0x3560 )
           ; yes: switch to state "forward 1 to 2"
           SetVar( 18 2 )
(C)        Call( :StartSequence )
           Jump( :SM_DONE )
```

While being in state 3, the object moves backwards from point 2 to point 1. This movement “mirrors” the “forward” movement in time and is therefore the ideal “reverse” for doors, barriers and things like that.

We prepare the rendering with the sequence code for this frame **(A)**.

Because this is a transition state (we are actually moving), we have to check if the end of the sequence is reached. This is done by checking the current time code (variable 0x1E) **(B)**.

When the COM1 frequency Is tuned to 135.60 while moving backwards, we switch state and start a new sequence **(C)** for the forward movement. The

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.

parameter for the start method (variable 0x1E) is already set to the correct value with code fragment **(A)**.

As you can see the code fragments of the stages for forward and reverse movement are very similar.

Some final words…

Remember this is a first release of the software, so not everything runs as smoothly as it should. If you encounter a problem, let me know at send an email to brf@brainon.ch

If you have a problem, make sure you include all files (gmax model, parameter file) in the email so I can reproduce the problem and work on a solution.

Copyright © 2002,2003 by Bernd R. Fix. All Rights Reserved.