THE UNIVERSITY
of MANCHESTER

SoftImage SDK Cookbook

How to create custom effects in SoftImage

Marc Evers

Computer Graphics Unit, University of Manchester

Verision 1.0 (Covers SDK version 1.1 Beta 1)

Contents

1.0 Introduction & Context of SDK in animation

Although the computer animation industry uses techniques developed in academia, the interaction between commerce & research has always been awkward. One of the principal reasons for this is the software used in the two camps. Commercial animators normally use large expensive pieces of software (primarily SoftImage and Alias|Wavefront), which are designed to provide an entire development environment which contains everything that an animator will need. These monolithic pieces of software tend to be ‘closed-packages’, which discourage the addition of new pieces of code. As the primary reason for graphics research is the development of new techniques the research community employs smaller, home-grown animation systems. This means that the code developed in research labs cannot be used directly by animators, and we are forced to wait many years for software developers to include the latest ideas.

This situation seems to be changing, as the main animation system developers are redesigning their code to encourage the addition of smaller modules, which can be designed to solve particular animation problems. This is good news for both academia & the industry, as it means research techniques can be evaluated directly by real animators.

This document is a kind of ‘cookbook’, which explains how to integrate pieces of code or algorithms into the commercial software animation package SoftImage 3D. This software now includes a software development kit (SDK), which provides a way of ‘bolting-on’ extra modules, i.e., the researchers code.

This cookbook consists of two parts. In the first part (sections 2.0 to 6.0) I will give a short overview of the SDK and describe the basic steps needed for integrating a a piece of code as a so called custom effect into SoftImage. The next step is a number of ‘How to’s and a description of some bugs and limitations of the used version of the SDK.

In the second part (sections 7.0 to 9.0) I will describe two case studies that I have done. I have integrated the following tools:

- *A Modelling Tool*. This allows an animator to generate blending surfaces between objects defined in SoftImage. These blends can be controlled to generate the correct model, and will then deform during the animation to maintain connections. The original code has been developed by David Hutchinson, this code could not be integrated using the SDK, so I have implemented a simpler algorithm by Martin Preston.
- *A Motion Synthesis Tool*. Here we wish to include code developed for animation NURBS using hyper-surfaces. Such a technique allows the animator to define the appearance of an object at various stages in time, and then the module determines how the object metamorphoses during time to shift between the states. This code has been developed by Martin Preston.

This cookbook has been written for the 1.1 Beta 1 version of the SoftImage SDK. For comments, queries, etc. about this cookbook please contact Martin Preston at the Computer Graphics Unit, e-mail: `preston@afs.mcc.ac.uk`.

Part I Using the SDK

2.0 SoftImage SDK Overview

SoftImage lets the programmer integrate his or her own code and algorithms by creating so called *custom effects*. A dialogue can be created for a custom effect and a number of arguments (i.e. the elements on which the custom effect is based) and results (i.e. the elements that are created or modified by the custom effect) can be specified.

The interface of the SDK through which custom effects communicate with SoftImage is called SAAPHIRE. It offers the programmer an object-oriented view of SoftImage. The SAAPHIRE programmer's interface is however written in plain C.

All SAAPHIRE function names start with `SAA_` and all functions return an error code of type `SI_ERROR`. If the returned value is `SI_SUCCESS`, the function call was successful. See the SAAPHIRE header file *SI_errorTypes.h* for a full list of error codes.

SAAPHIRE defines a hierarchy of object types similar to the one offered by the SoftImage 3D user's interface. There are object types like scenes, databases, all models, etc. Each object type has a number of attributes which can be manipulated using get/set functions. (`SAA_<object>Get<Attribute>()` and `SAA_<object>Set<Attribute>()`).

The basic object types are: database, scene, element, relation, selectlist (the list of elements that are currently selected in SoftImage), updatelist (a list of elements that need updating). The objects like models, camera, etc. are derived from element.

Note that although the SAAPHIRE interface is very similar to the SoftImage 3D interface, it is more restricted: not everything that you can do using the GUI can be done using SAAPHIRE.

3.0 Custom Effects

Each custom effect consists of a dynamically linked library (*.dso* for Silicon Graphics systems and *.dll* for Windows NT systems), a *custom definition file* (with the file extension *.cus*), which defines things like the dialogue, the arguments and results of the custom effect and the name of the custom effect library.

A custom effect can be placed in any of the menus of any of the modules of SoftImage; the name of the menu item is the same as the file name of the custom definition file.

Note: To use custom effects in SoftImage 3D one should first set the environment variables `SI_CUSTOM_*` (for each module in SoftImage there is such a variable) to contain the directories containing the binaries and the custom definition files.

3.1 Arguments, results and operator attributes

The *arguments* of a custom effect are the elements of the scene on which the custom effect is based. When the user executes the custom effect, SoftImage lets the user pick the arguments.

The *results* are the elements which are created or modified by the custom effect, based on the arguments.

It is also possible to enable the user to animate certain attributes of the custom effect by defining so called *operator attributes*.

3.2 Immediate and persistent effects.

A custom effect can be *immediate* or *persistent*: An immediate effect is applied only once and there will be no persistent relation between the arguments and the results.

In the case of a persistent effect SoftImage creates a kind of relation between the arguments and the results. If specific attributes of the arguments are modified (the programmer can specify these attributes in the custom definition file) SoftImage will ask the custom effect to update the result elements.

A persistent custom effect needs to have a so called *icon*, which can be specified in the custom definition file (see Section 4.1, “Custom definition file,” on page 9).

4.0 Basic Integration Steps

Integration of a module as a custom effect roughly consists of the following steps:

1. Determine the arguments and results of the effect
2. Create a custom definition file
3. Create a dialogue for the effect
4. Create the custom effect code:
5. Determine what will be in the init, update and cleanup functions
6. Interface the code with the dialogue (create setup, edit and callback functions)

4.1 Custom definition file

The custom definition file contains the following sections: Menu, Dialog, _SYMBOLS, _EXCLUSIVE, _BROWSER, _DESCRIPTION, Icon, _DYNAMIC_LINK, _OPERATOR.

Lines that begin with a '#' are assumed to be comments. You can also use the C++ like '//' comments to add a comment to the end of a line.

4.1.1 Menu

The *Menu* item specifies the number of the menu to which the custom effect will be added. The numbers can be found in the SAAPHIRE documentation.

4.1.2 Dialog

The *Dialog* section defines the dialogue to be used. Use the Dialog Editor to create and edit the dialogue. If you want to edit the *Dialog* section by hand, see the Dialog Editor manuals for a description of the exact syntax.

Note: trying to save a newly created dialogue to a custom definition file which does not yet contain a dialogue definition does not work. The way to do it is saving the dialogue to a separate file and copying this file into the custom definition file (this problem is described in the Dialog Editor documentation).

4.1.3 _SYMBOLS

The *_SYMBOLS* section contains a number of labels and the corresponding dialogue item identifiers. The labels are used to access these items from the custom effect code. You can use the Dialog Editor to create symbols for dialogue items.

Note: Not all dialogue items can be accessed using symbols (e.g. text views). See Section 5.2.1, "Retrieving and setting values," on page 26 how to access these items.

4.1.4 _EXCLUSIVE

In the *_EXCLUSIVE* section groups of identifiers of mutually exclusive radio buttons can be specified.

4.1.5 _DESCRIPTION

The *_DESCRIPTION* section contains a description of the custom effect. This text will be shown if the dialogue contains an *About* button and the user has clicked on it. The *About* button should have id 10.

4.1.6 _BROWSER

The *_BROWSER* section specifies which dialogue items will activate a browser. Items can activate a browser for files, models in the current scene or lights in the current scene.

4.1.7 Icon

If the custom effect is a persistent effect, it should have an icon. The icon should be a .hrc file (i.e. a model) and the file name of the icon (without the .hrc extension!) has to be specified using the *Icon* directive.

4.1.8 _DYNAMIC_LINK

In the *_DYNAMIC_LINK* section the name of the custom effect binary is specified using the *LINKNAME* directive. The names of the custom effect functions are also declared here, using the *INIT_FUNCTION*, *UPDATE_FUNCTION*, *CLEANUP_FUNCTION*, *SETUP_FUNCTION*, *EDIT_FUNCTION* directives.

4.1.9 _OPERATOR

The *_OPERATOR* section defines the arguments, results and operator attributes for the custom effect. Also a number of options can be specified using the *OPTIONS* directive. One of the important options is the *PICK_TIME* option:

- *OPTIONS: PICK_TIME=PRE_POP* specifies that the user picks the arguments before the dialogue is shown.
- *OPTIONS: PICK_TIME=POST_POP* specifies that the arguments are picked after the dialogue is shown (this is the default).

4.1.9.1 Arguments and results

The syntax of an argument definition is as follows: <category> <name> <chapter> <group> <pre> <scope> <condition> <message> <max>

The syntax of a result definition is as follows: <category> <name> <chapter> <group> <scope> <max>. The fields are described in the table below.

TABLE 1. Argument and result definition fields

Field	Values	Description
category	ARG	definition of an argument
	NEWRES	definition of a result that is created by the custom effect
	RES	definition of a result which is not created by the custom effect but which is one of the arguments modified by the custom effect.
name	any string	name of argument or result

TABLE 1. Argument and result definition fields

Field	Values	Description
chapter	e.g. MODELS	chapter type of the argument or result, e.g. model
group		group of attributes for which a dependency is created between the arguments and the results:
	TRANSL	translation attributes
	SCAL	scaling attributes
	ROTN	rotation attributes
	TRANSFO	composite group, includes translation, scaling and rotation
	GEOM	geometry attributes
	PARAM	any other type of attributes
pre	‘0’ or ‘1’	This option only applies when the definition concerns transformation attributes.
		If this field is ‘1’, SoftImage will first apply the transformation to the argument and will then ask the custom effect to update the results. If this field is ‘0’, SoftImage will apply the transformation after it has ask the custom effect to update its results.
scope	Node	Only the element that the user has picked is passed to the custom effect.
	Branch	The branch of the element that the user has picked is passed to the custom effect.
	Tree	The whole tree containing the element that the user has picked is passed to the custom effect.
condition		Puts a restriction on the elements that the user may pick. COND(attribute = value, value, ...).
		‘-’ means that no condition applies.
message	any string in quotes	Specifies the string that is shown when the user picks the argument. Use ‘-’ to use a default SoftImage message.
		Note that if there is more than one argument definition for a specific argument, only the first definition needs to have a message.
max	‘MAX’ followed by a number	This field is optional. It specifies a variable number of arguments. The number indicates what the maximum number of arguments is the user may pick.

If you want to specify more than group of attributes you will need to create a separate argument definition for each group. If you want e.g. a dependency between the geometry as well as the transformation attributes of the arguments and results, you will need to create a line containing the *TRANSFO* directive and a line containing the *GEOM* directive. See also the example custom definition file in Section 4.1.11, “Example,” on page 12.

All these definitions should have the same argument name. For each group of attributes there should be a separate result definition. In other words, each relation between specific attributes between the arguments and the results, requires its own argument and result definition lines in the custom definition file.

Example: ***Note:*** All argument definitions for the same argument should be placed together and the results should be defined before the arguments.

4.1.10 Operator attributes

The syntax of an operator attribute definition is as follows: <category> <name> <mod> <type> <range>. The fields are described in the table below.

TABLE 2. Operator attribute definition fields

Field	Values	Description
category	'OAT'	'OAT' indicates an operator attribute definition
name		Name of the operator attribute. This name should match a name defined in the _SYMBOLS section. In other words there should be a dialogue item that holds the value of this attribute. Note that SoftImage will use only the first six characters of the name.
mod	'A'	Specifying an 'A' here enables the user to animate the attribute using a function curve. It is possible to specify other options but these have not been documented anywhere.
type	'FLT'	The type of the operator attribute. Only floats (FLT) are supported.
range	RANGE(min, max)	Optional range specification. '-' means no range restriction.

Note: specifying a range for an operator attribute does not seem to have any effect in the 1.1 Beta 1 version of the SDK.

4.1.11 Example

Here is an example custom definition file (it is a modified version of the custom definition file of the motion tool, see also Section 8.0, "Motion tool," on page 43).

```
# This is an example .cus file for a persistent effect

# This example puts this effect in the Shape menu of the
# Motion module (this is menu number 59):
Menu 59

# Dialogue definition:
Dialog
# The first line defines things like the size, position, and
```

```
# the title of the dialogue. It also specifies the number
# of items on the dialogue.
20,"Example Dialog", 26,1,12,356, 310, 922, 713
   1,2,"Ok",816, 336, 906, 373, 0
   2,2,"Cancel",716, 336, 806, 373, 0
# This dialog uses items with reserved ids
#(see Section 4.6.1, "Reserved dialogue item ids," on page 23)
# SoftImage will automatically handle buttons 3 to 6 and
# will always show the current frame in text item 7
   3,2,"Set Key",821, 412, 894, 450, 0
   4,2,"Next Key",713, 458, 799, 495, 0
   5,2,"Prev Key",811, 457, 893, 495, 0
   6,2,"Delete Key",712, 412, 809, 450, 0
   7,5,"1",848, 561, 903, 591, 2, -200, 2000, 0
# The reserved items 8, 9, and 11 are not used, so I've
# created (empty) text labels for them
# Item 10 is the 'about' button and when the user clicks on it,
# the text in the _DESCRIPTION section is shown.
   8,0,"Start frame:",376, 603, 484, 618, 0
   9,0,"",806, 642, 806, 657, 0
   10,2,"About",618, 336, 708, 373, 0
   11,0,"",806, 642, 806, 657, 0

   12,5,"0.0",850, 540, 905, 570, 1, 0, 1, 0
   13,0,"Time parameter:",713, 550, 848, 565, 0
   14,5,"1",485, 589, 544, 619, 2, 0, 400, 0
   15,3,"Use current start and end frames",377, 624, 685, 639, 1
   16,0,"(c) 1996 Computer Graphics Unit",373, 688, 652, 703, 0
   17,0,"Frame:",714, 596, 768, 611, 0
   18,6,"graf",706, 425, 913, 646, 1
   19,7,"tvdir",373, 363, 499, 462, 0
   20,5,"1.0",527, 431, 606, 461, 1, -20, 50, 0
   21,0,"Sections:",374, 470, 455, 485, 0
   22,0,"Current weight:",529, 467, 664, 482, 0
   23,2,"Set Weight Factor",528, 383, 691, 420, 0
   24,5,"0",398, 316, 418, 346, 1, MIN, MAX, 0
   25,5,"1",850, 584, 905, 614, 2, -200, 2000, 0
   26,5,"2",367, 316, 386, 346, 4, 0, 100, 0

# The _SYMBOLS section defines symbols for some text items
# in order to access them using the custom value list.
_SYMBOLS
  NBARGS    11
```

```
  Time    12
  STRTFRM    14
  AUTOFRM    15
  WEIGHT    20
  W1    24
_END

# The following text will be shown if the user selects
# the 'About' button on the dialogue
_DESCRIPTION
This custom effect does the following things...
Version 1.x

_END

# Use the following icon to represent this effect.
# The file 'TheIcon1.hrc' is used.
Icon icon TheIcon1

# The _DYNAMIC_LINK section defines the binary name and
# the names of the custom effect functions.
_DYNAMIC_LINK

# The binary file for this custom effect is called
# 'effect.dso'. Do not put the .dso extension here.
LINKNAME effect

# This effect has all five custom effect functions
# If you would want to define the init, update or cleanup
# functions as SDK v1.0 style, use the CUSTOM_INIT_FUNCTION
# CUSTOM_UPDATE_FUNCTION or CUSTOM_CLEANUP_FUNCTION
# directives instead
INIT_FUNCTION           init
UPDATE_FUNCTION         update
CLEANUP_FUNCTION        cleanup
DIALOG_SETUP_FUNCTION   setup
DIALOG_EDIT_FUNCTION    edit

_END

# Sample operator definition.
_OPERATOR
OPTIONS: PICK_TIME = PRE_POP
```

```
# PICK_TIME=PRE_POP / POST_POP ,
# PRE_POP means the user picks the arguments before the
# dialogue is shown.

# For this effect there are two dependencies between the
# arguments and results, namely a geometry dependency
# and transformation dependency
# Note that each pair of result lines and argument lines
# has the same name, so they designate the same objects.
--------------------------------------------------------
cat    name  chap   group  pre scope  cond           msg
--------------------------------------------------------
NEWRES  blnd MODELS TRANSFO     NODE   -
NEWRES  blnd MODELS GEOM        NODE   -

# The user will only be able to pick NURBS surfaces (NSRF)
# or NURBS curves (NCRV)
# The 'transformation' line has the PRE field set to 1,
# because we want SoftImage to update the arguments first
# before the custom effect updates the result
ARG  srf1 MODELS TRANSFO  1  NODE COND(MdlType=NSRF, NCRV)   "Pick
a surface or curve" MAX 10
ARG  srf1 MODELS GEOM     0  NODE COND(MdlType=NSRF, NCRV) -

# This effect has one operator attribute, named Time.
# 'Time' also has been defined in the _SYMBOLS section.
OAT Time   A FLT -

_END
```

4.2 General Custom Effect program structure

4.2.1 Custom effect functions

A custom effect generally consists of one or more of the following functions: `init()`, `update()`, `cleanup()`, `setup()`, `edit()`. You can specify your own function names in the custom definition file.

The custom effect functions should have the following signature:

```
_CUS_EXTERN SI_Error entry_point( SAA_CustomContext context );
```

The first thing that any custom effect should do is to call the `SAA_sceneInit()` function to initialise SAAPHIRE (`result = SAA_sceneInit();`).

init(). The `init()` function is meant for initialisation of the custom effect and is called by SoftImage only once, namely when the effect is applied (that is after the arguments have been picked and the dialogue has been shown).

In the case of a persistent effect, the `init()` function is also called when a scene containing the effect is loaded. To determine whether the scene is being loaded, extract the results array from the custom context (see below) and check the *elemid* field of any of the result elements. If this field is -1 it means that the results do not yet exist. If it is not equal to -1, then the result already exists and the scene is being loaded. Note that this only applies to results that have been declared as *NEWRES* results.

Example

An excerpt from the `init()` function which sets 'NEWRES' results if necessary. It sets a result that has two definitions in the custom definition file, like in the example in Section 4.1.11 on page 12.

```
_CUS_EXTERN SI_Error init( SAA_CustomContext context ) {
  SAA_Elem * aModel;
  SAA_Elem res[2];

  result = SAA_sceneInit();
  ...
  SAA_customContextGetResults( context, 2, res );
  if ( res[0].elemid == -1) {
    res[0] = aModel;
    res[1] = aModel;
    SAA_customContextSetResult( context, 2, res );
  }
}
```

update(). The `update()` function is called each time the scene is updated or when arguments are modified.

The `update()` function is called for each result definition line. The function `SAA_customContextGetCurrentResultIndex()` returns the index in the result array of the element to be updated. Note that if one result has multiple lines in the custom definition file, `update()` is called for each separate line, so check this to prevent the `update()` function from updating the results more often than is necessary.

cleanup(). The `cleanup()` function is called when the effect is finished (in case of immediate effects) or when the effect is deleted (in case of persistent effects) and can be used to clean up any memory used by the custom effect.

A persistent effect is deleted by deleting a result, the icon or one of the arguments.

setup() & edit(). The `setup()` function is called just before the dialogue is first shown. The `edit()` function is called when the user selects *Edit Parameters* from the menu.

Both functions can be used to set up the dialogue and to attach so called callback functions to dialogue items.

A custom effect is not required to have all five functions. A immediate effect e.g. can do with just the `init()` function, while a persistent effect can do with only the `update()` function.

Notes:

- The `SAA_customContextGetCurrentResultIndex()` does not work properly in the 1.1 Beta 1 version. See Section 6.2, “Problems with results in custom context,” on page 31 for how to work around it.
- If `init()` or `setup()` return the error code `SI_ERR_CUSTOM_FATAL`, the effect is cancelled.
- The `update()` function is always called directly after the `init()` function. Init just needs to produce a result, update can perform the actual effect.

4.2.2 Callback functions

A callback function is a function attached to a dialogue item. The function is called each time the item is clicked or modified (depends on the type of the item) and can be used e.g. to apply parameters to the scene without having to close the dialogue.

Callback functions have to be installed in the `setup()` and `edit()` functions using the `SAA_dialogitemAddCallback()` function. You can install at most one callback function per item, but one callback function may be attached to more than one item.

The parameters that are passed to the callback function are: the custom context of this custom effect, the id of the item that has triggered the callback and a pointer to some user data. The latter can be used to pass some data from the `setup()` or `edit()` functions to the callback functions.

Callback functions are automatically removed when the dialogue is closed. If you need to remove a callback explicitly, use the `SAA_dialogitemRemoveCallback()` function.

Example

In the following piece of code a callback called ‘Callback’ is attached to the *Ok* button by the `setup()` function.

```
#define DLG_OK  1

/* Callback functions should always have the following
   signature */
SI_Error Callback (
  const SAA_CustomContext context,
  int dialogItemId,
  void *userData
 );
```

```
_CUS_EXTERN SI_Error setup ( SAA_CustomContext context ) {
  SAA_dialogitemAddCallback( context, DLG_OK, &okCallback,
    NULL );
}
```

4.2.3 Calling order

Figure 1 on page 18 shows the order in which the custom effect functions are called during the initialisation phase of the effect.

FIGURE 1. Custom effect initialisation phase

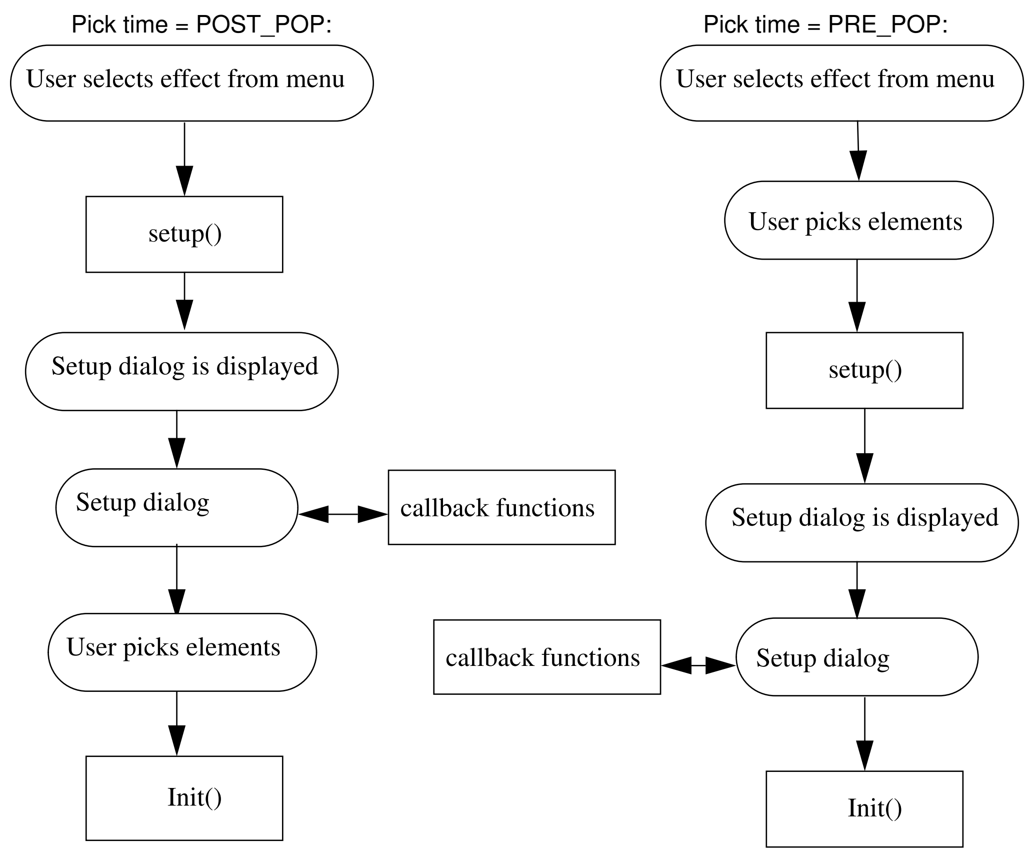
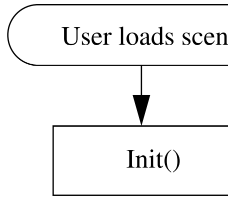


Figure 2 on page 19 shows the initialisation phase of a persistent custom effect when it is loaded.

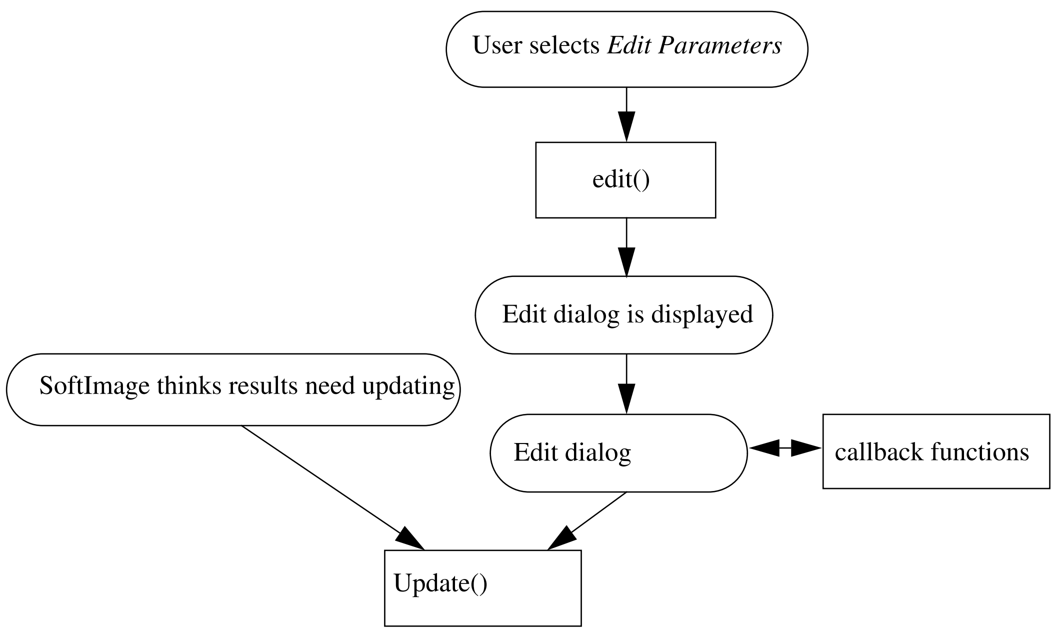
FIGURE 2. Persistent effect, initialisation phase when loading a scene



If the custom effect is an immediate effect, then after the initialisation phase, the `update()` function is called just once and then `cleanup()` is called.

If the custom effect is a persistent effect, then the `update()` function is called each time the scene needs updating. Furthermore, if the user selects *Edit Parameters* from the menu, the dialogue will be shown. Figure 3 on page 19 shows the calling order of the different functions in these situations.

FIGURE 3. Persistent effect, update & edit parameters



4.3 Custom context

The *custom context* is a data structure that contains all the relevant data for an instance of a custom effect. The custom context is passed to all the custom effect functions and it contains pointers to the instance data, the arguments, the results, the custom value list (described in Section 4.6, "Dialogues," on page 22). It also contains the number of results and the number of arguments.

Note: version 1.0 of the SDK does not have custom contexts; instead all the items are passed separately to the custom effect functions. A version 1.0 style `init()` function has the following prototype:

```
_CUS_EXTERN SI_Error init ( SAA_CustomValueList valueList, int
nbargs, SAA_Elem *args, int  nbres, SAA_Elem *res, void
**instdata );
```

4.4 Instance data

Each instance of a custom effect can have its own piece of data, called the *instance data*. The instance data is generally allocated in the `init()` function and deallocated in the `cleanup()` function. SAAPHIRE stores the pointer to the instance data in the custom context of the custom effect.

The instance data can be set using the `SAA_customContextSetInstanceData()` function and it can be retrieved using the `SAA_customContextGetInstanceData()` function.

Note: It is possible to share data between different instances of one custom effect: any data that is global in the custom effect will be shared by all instances. This data will be retained as long as there is at least one instance active: as soon as the last instance is deleted, the shared data will be lost.

Example

The following piece of code allocates the instance data, stores the current scene in it and passes the instance data to SoftImage.

```
  InstanceData * idata;

  idata = ( InstanceData * ) malloc( sizeof( InstanceData ) );
  if( idata != NULL ) {
    result = SI_SUCCESS;
  } else {
    printf( "Error allocating instance data memory." );
    result = SI_ERR_CUSTOM_FATAL;
  }
/* Store the current scene in the instance data. This
   is more efficient than getting it from SoftImage each time
*/
  if( result == SI_SUCCESS ) {
    result = SAA_sceneGetCurrent( & idata->scene );
/* Pass the pointer to the instance data to SoftImage */
    result = SAA_customContextSetInstanceData(context,
      (void *) idata);
  }
```

The following piece of code retrieves the instance data:

```
InstanceData * idata;
result = SAA_customContextGetInstanceData(context,
  (void **) &idata);
```

4.5 Arguments and results

The arguments and results of a custom effect are stored in arrays of SAAPHIRE elements. For each result or argument line in the custom definition file there will be an element in the array.

When using a variable number of arguments or results all arguments or results corresponding to a single line will be consecutively in the array. An example:

If the custom definition file contains the following argument definitions:

```
ARG a1 MODELS TRANSFO 1 NODE - -
ARG a1 MODELS GEOM 0 NODE - -
ARG a2 MODELS TRANSFO 1 NODE - - MAX 4
ARG a2 MODELS GEOM 0 NODE - - MAX 4
ARG a3 MODELS TRANSFO 1 NODE - -
ARG a3 MODELS GEOM 0 NODE - -
```

and the user has picked 2 models for argument *a2*, then the argument array will have 8 elements. The table below shows what the argument array looks like. The number between parentheses following *a2* indicates whether the element corresponds to the first or the second element that the user picked for *a2*.

Index	Element
0	a1 (TRANSFO)
1	a1 (GEOM)
2	a2 (1) (TRANSFO)
3	a2 (2) (TRANSFO)
4	a2 (1) (GEOM)
5	a2 (2) (GEOM)
6	a3 (TRANSFO)
7	a3 (GEOM)

The number of arguments and the arguments array can be retrieved from the custom context with the `SAA_customContextGetNbArguments()` and `SAA_customContextGetArguments()` functions.

The function `SAA_customContextGetArgumentOccurrences()` returns an array with the number of occurrences picked by the user for each argument (that is, for each argument definition line in the custom definition file).

Note: There are two functions to retrieve the number of arguments, one to get the number of arguments picked by the user, the other to get the size of the argument array. The functions are called `SAA_customContextGetNbArguments()` and

SAA_customContextGetNbArgumentLines(). The former returns the size of the arguments array, so I assume the latter will return the number of arguments picked by the user. According to the documentation of the SDK version 1.1 Beta 1, this should be the other way around.

The number of results and the results array can be retrieved from the custom context with the SAA_customContextGetNbResults() and SAA_customContextGetResults() functions. To store new results (i.e. results defined as 'NEWRES' in the custom definition file) use the SAA_customContextSetResult() function.

Example

For an example of setting the results, see Section 4.2.1, "Custom effect functions," on page 15. An example piece of code for getting the results:

```
SAA_Elem * res;
int nbRes;
SAA_customContextGetNbResults( context, &nbRes );
res = (SAA_Elem *) calloc( nbRes, sizeof( SAA_Elem ) );
SAA_customContextGetResults( context, &nbRes, res );
```

Example piece of code for getting the arguments:

```
SAA_Elem * args;
int nbArgs;
SAA_customContextGetNbArguments( context, &nbArgs );
args = (SAA_Elem *) calloc( nbArgs, sizeof( SAA_Elem ) );
SAA_customContextGetArguments( context, &nbArgs, args );
```

Notes:

- If a persistent effect creates new results, these results can only be set in the init() function and not in e.g. the update() function. Furthermore if a result has multiple lines in the custom definition file, the results array will contain elements for all lines and you will need to set all of them to the same result.
- The SAA_customContextSetResult() function does not work properly in the 1.1 Beta 1 version of the SDK. See Section 6.2, "Problems with results in custom context," on page 31 how to work around it.

4.6 Dialogues

Dialogues contain the parameters of a custom effect and enable the user to modify these parameters. The items of a dialogue retain their values for an instance of a custom effect and the values will be saved when the scene containing the effect is saved.

The values of the items that have an associated symbol are stored in the so called *custom value list*. This list is only updated when the user has selected the Ok button in the dialogue. To retrieve the value of an item from the custom value list, use the SAA_cusvalGetStateValue(), SAA_cusvalGetFloatValue(), `SAA_cusvalGetIntValue()`, and `SAA_cusvalGetStringValue()` functions. To get the length of a value of type string, use the `SAA_cusvalGetStringLength()` function.

The item is identified by the name declared in the symbols section of the custom definition file.

Note: The contents of text view items are not stored in the custom value list and will not be saved with the custom effect. The text items will however retain their contents throughout the SoftImage session.

Example

Getting the floating point value from the dialogue item with symbol 'Time' from the example custom definition file as given in Section 4.1.11 on page 12:

```
_CUS_EXTERN SI_Error update( SAA_CustomContext context )
{
  float timeFactor;
  SAA_CustomValueList valueList;

  SAA_customContextGetCustomValueList( context, &valueList );
  SAA_cusvalGetFloatValue( valueList, "Time", &timeFactor );
}
```

4.6.1 Reserved dialogue item ids

The dialogue item ids 1 to 11 are reserved for special purpose dialogue items. If a dialogue item has one of these ids, SoftImage will automatically attach some default action to it, so do not use these ids for other items. If you do not use these special items, create empty text items as 'dummies'. See the Dialog Editor User's Guide for more information.

Ids 1 and 2 are reserved for the *Ok* and *Cancel* buttons; ids 3 to 6 are reserved for key framing buttons; id 7 is reserved for a *current frame* text edit; id 10 is reserved for the *About* button; ids 9 and 11 are reserved for items that work on shaders.

Note: If you attach callback functions to items with a reserved id, first your callback will be called, then SoftImage will perform the default behaviour for that item. You cannot override the default behaviour.

4.6.2 Operator attributes

For each operator attribute, SoftImage will create a function curve with the same name.

Operator attributes should have a corresponding dialogue item. This item is the 'interface' of the operator attribute to the user and to the programmer. The user can edit this function curve using the dialogue or by using the fCurve window in SoftImage. The programmer can use the value of the dialogue item to retrieve values of the function curve.

4.7 Notes and tips for programming custom effects

- There is only one include file for SAAPHIRE, namely *SAA.h*.
- In the *_OPERATOR* section of the custom definition file, do not put any comments between the *_OPERATOR* keyword and the *OPTIONS* keyword. If you do so, the options will be ignored by SoftImage.
- If you use a C++ compiler take care that the init, update, cleanup, edit, setup and callback functions are compiled as C functions.
- If one needs to save additional data about the custom effect, one can use the user data field of an element. Attach the user data to one of the result elements (note that in the 1.1 Beta 1 version, the user data feature does not yet work, see also Section 6.6, “User data functions,” on page 33.
- The values in the items of a dialogue will be kept throughout a SoftImage session, so don’t forget to clear/reset the items when necessary.
- SAAPHIRE does not do any memory management. The programmer is responsible for allocating and deallocating memory for all results. There is one exception, namely the function `SAA_modelAllocUserData()`.
- When you retrieve the coordinates of the vertices of a model using `SAA_modelGetVertices()`, you will get the local coordinates. To convert them to global coordinates, retrieve the global transformations of the model and apply these to the vertices.
- A callback attached to a text view item is triggered if the user selects an item in the text view. The user can also select more than one item or zero items so beware of this.
- Once the user has picked the arguments for an instance of a custom effect, it is not possible to specify different or additional arguments for that instance.
- If you have modified e.g. the results from one of the callback functions, you will need to call the `SAA_sceneRefresh()` function to refresh the SoftImage windows and to make the changes visible to the user. Example: `result = SAA_sceneRefresh( context );`

4.7.1 Debugging

- It is possible to use dbx to debug custom effects. You can also use `printf()` and functions like `scanf()` and `gets()`: the output is directed to the terminal window from which SoftImage has been started.
- SoftImage has a number of verifying flags. If these are set to TRUE, SoftImage will e.g. check the types of the arguments of SAAPHIRE functions. Switching the flags off will make the custom effect execute slightly faster (I assume).
- If the custom effect library file is changed, SoftImage does not need to be restarted. If the effect is persistent, the new version will be used only after all current instances of the effect have been deleted.
- If the custom definition file is changed, you will need to restart SoftImage for the changes to have effect.

5.0 How to...

5.1 How to use operator attributes and custom function curves

It is not possible to retrieve the function curve of an operator attribute directly via SAAPHIRE. If you need e.g. the value of the operator attribute at the current frame number, use the value of the dialogue items corresponding to the operator attributes. These items will always contain the operator attribute value of the current frame.

The operator attribute function curves are initialised with the value that the user enters in the operator attribute dialogue items during the setup dialogue. This will result in a constant function curve.

Example

The dialogue item with symbol ‘Time’ from the example custom definition file as given in Section 4.1.11 on page 12 holds the value of the ‘Time’ operator attribute. If you need the value of that operator attribute at the current frame, just read the value of the item. See the example in Section 4.6, “Dialogues,” on page 22.

5.1.1 Enabling key frame editing by the user

The editing of an operator attribute function curve by the user is almost completely handled by SoftImage. To enable the user to modify key frames for the operator attributes, the only thing you need to do is to create buttons with the reserved ids 3 to 6; these buttons will respectively act as set key button, next key button, previous key button, and delete key button. To let the user set and view the current frame number, create a text field with reserved id 7. See also the example custom definition file in Section 4.1.11 on page 12.

SoftImage will automatically handle these buttons and the current frame field: the user can use the key buttons and the frame number field to step through the key frames that have been set for the operator attribute function curves. The user can set a key frame by entering the value in the operator attribute dialogue items and clicking the set key button.

When the dialogue is active the user can also use the SoftImage timeline controls to change the current frame number.

Notes:

- SoftImage automatically updates the scene when the current frame number is changed.
- The user can also edit the operator attribute function curves directly using the fCurve window in SoftImage. The operator attribute function curves are selected using the *FcrvSelect->Custom* menu in the Motion module.
- The Dialog Editor will assign ids in order of creation of the items, so you may need to edit the custom definition file to change the ids.

5.2 How to communicate with dialogues

5.2.1 Retrieving and setting values

Apart from the custom value list, there are a number of functions to get and set the values of dialogue items directly, using the `SAA_dialogitemGet*Value()` and `SAA_dialogitemSet*Value()` functions. These functions can only be used in the `setup()`, `edit()` and callback functions and enable you to make 'interactive' dialogues.

The difference with the custom value list is that the latter will only be updated after the dialogue has been closed. The `SAA_dialogitemGet*Value()` functions always return the current value of the dialogue item. Furthermore, these functions also give access to dialogue items of which the values are not stored in the custom value list. These functions access the dialogue items using their ids (and not by symbols).

Example

The following piece of code first gets the value of the 'Weight' text item from the example custom definition file as given in Section 4.1.11 on page 12 and then sets it.

```
#define DLG_WEIGHT  20

float w;
SAA_dialogitemGetFloatValue( context, DLG_WEIGHT, &w );
... /* do something with w */
SAA_dialogitemSetFloatValue( context, DLG_WEIGHT, w );
```

5.2.2 Using Text View items

Text view items (an item that contains a number of strings which the user can select) are accessed in a different way. To clear a text view, use `SAA_textlistClear()`; lines can be added or removed using the `SAA_textlistCreateItem()` and `SAA_textlistDestroyItem()` functions; the selection mode of the text view (single or multi) is set and retrieved using the `SAA_textlistSetSelectionMode()` and `SAA_textlistGetSelectionMode()`.

The `SAA_textlistGetNbSelected()` function returns the number of lines which are selected. The function `SAA_textlistGetSelected()` returns an array containing the indexes of the selected lines. These indexes can be used to get the actual selected strings using the `SAA_dialogitemGetStringValue()` function.

Example

In the following code, the `edit()` function fills a text view and attaches a callback to it. The callback retrieves the selected item from the text view.

```
#define DLG_TXTVW  19

_CUS_EXTERN SI_Error edit( SAA_CustomContext context )
{
  SAA_textlistClear( context, DLG_TXTVW );
  SAA_textlistCreateItem( context, DLG_TXTVW, 0, "Item 1");
```

```
  SAA_textlistCreateItem( context, DLG_TXTVW, 0, "Item 2");
  SAA_textlistCreateItem( context, DLG_TXTVW, 0, "Item 3");
  SAA_textlistSetSelectionMode( context, DLG_TXTVW,
    SAA_SELECT_SINGLE );
  SAA_dialogitemAddCallback( context, DLG_TXTVW,
    &weightCallback, NULL );
}

SI_Error weightCallback ( const SAA_CustomContext context,
  int dialogItemId, void * userData )
{
  int nbSel, idx, size;
  char s [80];

  SAA_textlistGetNbSelected( context, DLG_TXTVW, &nbSel );
  if ( nbSel == 1 ) {
    SAA_textlistGetSelected(context, DLG_TXTVW, nbSel, &i);
    SAA_dialogitemGetStringLength( context, DLG_TXTVW, i,
      &size );
    SAA_dialogitemGetStringValue( context, DLG_TXTVW, i,
      size, s );
    ...
  }
}
```

5.2.3 Enabling and disabling items

You can enable or disable dialogue items with the `SAA_dialogitemSetSensitivity()` function. Disabled items appear greyed on the dialogue and they will not respond to the user.

Note: If the *sensitivity* parameter of the `SAA_dialogitemSetSensitivity()` function is TRUE, the item will be disabled; if it is FALSE, the item will be enabled.

Example

The following piece of code disables the 'Weight' text item from the example custom definition file as given in Section 4.1.11 on page 12:

```
  SAA_dialogitemSetSensitivity( context, DLG_WEIGHT, TRUE );
```

5.3 How to handle NURBS from SoftImage

The way SoftImage stores the control vertices of a NURBS differs slightly from the normal way. Whereas normally the vertices of a NURBS are in the form $(w \cdot x, w \cdot y, w \cdot z, w)$, SoftImage stores them as (x, y, z, w).

In the case of an open NURBS, SoftImage stores two knots less than is normally the case[1]. It assumes that the first and the last knot of the knot vector are equal to the second and second-last ones.

5.3.1 Retrieving NURBS parameters

In the table below several functions for retrieving the parameters of NURBS curves and surfaces are described.

TABLE 3. NURBS parameter retrieval functions

Function	Purpose
SAA_modelGetNbVertices()	Get the number of control vertices of a NURBS curve or surface
SAA_modelGetVertices()	Get the control vertex array of a NURBS curve or surface
SAA_nurbsCurveGetDegree()	Get the degree of a NURBS curve
SAA_nurbsSurfaceGetDegree()	Get the degree of a NURBS surface
SAA_nurbsCurveGetClosed()	Get the open/closed parameter of a NURBS curve
SAA_nurbsSurfaceGetClosed()	Get the open/closed parameter of a NURBS surface
SAA_nurbsCurveGetNbKnots()	Get the number of knots of a NURBS curve
SAA_nurbsSurfaceGetNbKnots()	Get the number of knots of a NURBS surface
SAA_nurbsCurveGetKnots()	Get the knot vector of a NURBS curve
SAA_nurbsSurfaceGetKnots()	Get the knot vector of a NURBS surface

Example

The following piece of code retrieves all information from a NURBS curve.

```
void getNURBSinfo( SAA_scene * scene, SAA_Elem * curve )
{
  int degree, nbVertices, nbKnots;
  SAA_Boolean closed;
  SAA_Elem * ctrlVertices;
  double * knots;

  SAA_nurbsCurveGetDegree( scene, curve, &degree );
  SAA_modelGetNbVertices( scene, curve, &nbVertices );
  SAA_nurbsCurveGetClosed( scene, curve, &closed );
  ctrlVertices = (SAA_Elem *) calloc( nbVertices,
    sizeof( SAA_Elem ) );

  SAA_modelGetVertices( scene, curve, SAA_GEOM_ORIGINAL, -1,
```

1. The number of knots required is equal to the number of control points plus the order of the NURBS.

```
    nbVertices, ctrlVertices );
  SAA_nurbsCurveGetNbKnots( scene, curve, &nbKnots );
  knots = (double *) calloc( nbKnots, sizeof( double ) );

  SAA_nurbsCurveGetKnots( scene, curve, SAA_GEOM_ORIGINAL,
    -1, nbKnots, knots );
  ...
}
```

5.3.2 Modifying existing SoftImage NURBS

If you want to modify the attributes of a SoftImage NURBS curve or surface, you will need to use the functions `SAA_nurbsCurveEdit()` and `SAA_nurbsSurfaceEdit()`. These functions set the control vertices, degree, parametrisation type and closing of the NURBS all in once.

The knot vectors can be set independently using the `SAA_nurbsCurveSetKnots()` and `SAA_nurbsSurfaceSetKnots()` functions.

Example

The following piece of code shows how to set the knot vector of a NURBS curve.

```
void setKnots( SAA_scene * scene, SAA_Elem * curve )
{
  int nbKnots;
  double * knots;

  SAA_nurbsCurveGetNbKnots( scene, curve, &nbKnots );
  knots = (double *) calloc( nbKnots, sizeof( double ) );
/* set all the knot values */
  ...
  SAA_nurbsCurveSetKnots( scene, curve, SAA_GEOM_ORIGINAL,
    -1, nbKnots, knots );
}
```

5.3.3 Closed NURBS

SoftImage appears to close NURBS by replicating a number of control vertices at display time. These extra vertices are however not displayed to the user and the programmer.

A closed NURBS curve in SoftImage has as many vertices as the open version (say n), but it only has n+1 knots. The closing method is the following (also described in [Piegl95]): if the degree of the NURBS is d, replicate the d first control vertices and add them to the end of the control vertex list.

The extra knots are created by ‘wrapping around’ the knot vector and adding 2*d knots, that is not by duplicating the knot values itself, but by adding knots so that the differences between consecutive new knots are equal to the differences between consecutive knots at the beginning of the knot vector.

The actual values of the knots are not really important, the differences between the knots which are relevant.

The algorithm to convert a closed SoftImage NURBS to an open NURBS that looks exactly the same, is (where *n* is the number of control vertices and *d* the degree):

- Duplicate the first $\frac{d}{2}$ (rounded up) control vertices at the end of the vertex array and the last $\frac{d}{2}$ (rounded down) vertices at the beginning of the array.
- Create *d* knots at the end of the knot vector by using the differences between consecutive knots at the beginning of the knot vector and create *d* knots at the beginning of the knot vector by using the differences between consecutive knots at the end of the knot vector:

$$
\begin{gathered}
\text{new knot vector} = k_{-d}k_{-d+1}\ldots k_{-1}\,[k_0k_1\ldots k_n]\,k_{n+1}k_{n+2}\ldots k_{n+d} \\
\text{where:} \\
k_0\ldots k_n = \text{the original SI knot values} \\
k_{n+1} = k_n + (k_1 - k_0)\text{, etc.} \\
k_{-1} = k_0 - (k_n - k_{n-1})\text{, etc.}
\end{gathered}
\qquad \textbf{(EQ 1)}
$$

This will produce a NURBS which looks closed. The problem with this NURBS is that it is not clamped: the first and last knots do not have multiplicity *d*+1. Some algorithms will need clamped NURBS. For ‘clamping’ them, there is a simple algorithm, also described in [Piegl95], page 576.

5.4 How to store extra information about the custom effect

If you need to store extra information about the custom effect, which is not contained in the dialogue items and which cannot be reconstructed from the arguments and the dialogue data alone, you can use the user data of the result models.

Note: The user data functions do not work in the 1.1 Beta 1 version. See Section 6.6, “User data functions,” on page 33 how to work around this.

6.0 Bugs and limitations of SDK 1.1 Beta 1

6.1 Limitations

- It is not possible to retrieve the coordinates in the u/v space of projection and trim curves.

6.2 Problems with results in custom context

In the 1.1 Beta 1 version of the SDK something seems to go wrong with the number of results in the custom context. As a result the functions `SAA_customContextSetResult()` and `SAA_customContextGetCurrentResultIndex()` do not work properly. The former will always give an error and the latter will not return a correct index.

To set results, you should declare the `init()` function version 1.0 style[1]. The init function should be declared in the custom definition file using CUSTOM_INIT_FUNCTION and it should have the following signature: `SI_Error init ( SAA_CustomValueList cusvals, int nbargs, SAA_Elem *args, int nbres, SAA_Elem *res, void **instdata );`

The old-style `init()` function does not have a custom context parameter, but it has the argument and result arrays as parameters. To set the results, simply assign the results to the elements in the result array.

To work around the problem with `SAA_customContextGetCurrentResultIndex()`, you will need to determine in some other way which result is currently being updated. If the custom effect has a fixed number of results, you could e.g. put a variable in the instance data which indicates the result to be updated next. The `update()` function increments the variable each time it is called and resets it after all results have been updated.

Example

The following piece of code shows how a version 1.0 `init()` file would look like.

```
SI_Error init ( SAA_CustomValueList cusvals, int nbargs, SAA_Elem
*args, int nbres, SAA_Elem *res, void **instdata )
{
  InstanceData * idata;
  SAA_Elem * aModel;

  idata = ( InstanceData * ) malloc( sizeof( InstanceData ) );
  ...
/* Pass the pointer to the instance data to SoftImage */
  (*instdata) = idata;
```

1. SAAPHIRE allows mixing of old and new style init, update and cleanup functions. The edit(), setup() and callback functions however always need to be new style.

```
  ...
/* store the results */
  res[0] = aModel;
  res[1] = aModel;
}
```

6.3 Argument and result count bug

When a variable number of arguments is used and the custom effect is loaded, the arguments and results arrays will contain too many elements.

I have worked around this problem by saving the number of arguments in a dialogue (which is disabled to prevent the user from interfering) item when the effect is created and using the value of this item instead of the value provided by SoftImage. This works because the variable argument definition is the last one in my custom definition files, and the elements of the arguments array are correct up till the elements corresponding to the first variable argument definition line.

This method however requires that the number of picked arguments is known during the setup dialogue phase, which means that the arguments have to be picked before the dialogue is shown.

Note: The same problem may arise when using a variable number of results.

6.4 SAA_trimCurveExtract() problems

The `SAA_trimCurveExtract()` function has two problems:

1. It does not have a prototype in the SAAPHIRE header files, although it does exist in the library. If you need it, use the prototype from the manuals:
   ```
   SI_Error SAA_trimCurveExtract
     (
      const SAA_Scene *pscn,
      const SAA_Elem *pelmSrf,
      int nCrv,
      SAA_SubElem *psubelmCrv,
      SAA_Elem *pelmCrv
     );
   ```
2. It only works with projection curves. When used on trimming curves, it always gives an error message.

6.5 Model verify flag and SAA_nurbsCurveGetKnots()

If the model verify flag is on, calling the `SAA_nurbsCurveGetKnots()` function will always return an error. Switch off the flag before calling the function and switch it on again after.

Example

```
SAA_verifyFlagSet( SAA_VERIFY_MODEL, FALSE );
SAA_nurbsCurveGetKnots( ... );
SAA_verifyFlagSet( SAA_VERIFY_MODEL, TRUE );
```

6.6 User data functions

The user data functions do not work properly, so it is not possible to use user data in the 1.1 Beta 1 version of the SDK. If you want to save any data, create a special text edit item for it on the dialogue. Its contents will be saved when the custom effect is saved.

Set the values in the `setup()`, `edit()` or callback functions using the `SAA_dialogitemSet*Value()` functions. When the data is needed, it can be retrieved using the custom value list.

Notes:

- I have tried setting the values in the `setup()` function, but it did not seem to work. Install a callback function for the *Ok* button instead to set the values.
- It is not possible to hide these items from the user; the best you can do is disabling them in the `setup()` and `edit()` functions. If you try to place items outside the dialogue borders, SoftImage will move them inside the borders.

6.7 Bugs in Dialog Editor

- In the ‘text edit parameters’ dialogue: if *alphanumeric* has been selected as the type, then when the dialogue is shown again, the type *numeric integer positive not null* will also be selected. Each time one opens this dialogue, one has to select *alphanumeric* again.
- In the ‘text edit parameters’ dialogue it is not possible to type a decimal point in the minimum and maximum value fields. You should edit the custom definition file instead.

6.8 SAA_selectlistSetMode bug

If you try to set the selectList to single selection mode, SoftImage will give an error and the selection mode will not be changed.

Part II Case Studies

7.0 Modelling tool

7.1 Surface blending tool (introduction)

The surface blending custom effect allows the user to blend two surfaces together, according to curves projected onto these surfaces. It lets the user pick two surfaces and curves on these surfaces, and it generates a third surface which forms the blend between the selected curves on the picked surfaces. Changing the original surfaces will automatically update the blend surface.

7.2 Idea behind blending algorithm

Originally, we wanted to use a blending algorithm written by David Hutchinson, but we couldn't use it because it requires the coordinates of projected curves in the u/v space of the surfaces onto which they have been projected. The current version of the SoftImage SDK cannot give these coordinates.

I have implemented a much simpler algorithm instead. The idea behind this simple blending algorithm is that on each of the two surface two projected curves are selected. If these two curves are close enough, the surface between them will be more or less on the original surface (it's a kind of linear approximation of the derivative of the original surface on the curve).

To give the user some control over the blend surface, he/she can change two parameters, one for each argument surface. These parameters determine the distance between the two curves on the argument surfaces.

The other way for the user to control the blend surface is to specify a number of intermediate NURBS curves, which are inserted in the blend surface.

Instead of using projected curves for the second argument surface, the user can specify two curves with a constant *u* or *v* parameter on the surface.

7.3 Remarks about design of custom effect

7.3.1 Source files

The blending surface custom effect consists of the following source files:

blend.c. Contains the custom effect 'body': processing the dialogue, processing the surfaces picked by the user, cleaning up, etc.

blending.c. Contains a number of function that operate on NURBS in NURBS Procedure Library format, like the blending algorithm, a NURBS normalisation algorithm and NURBS clamping functions.

oslo.c. Contains the oslo algorithm for increasing the number of knots/vertices (functions adapted from Graphics Gems IV)

convert.c. Contains a number of conversion functions and functions which implement some aspects of conversion between SoftImage NURBS format and the NURBS Procedure Library format. Also some 'general purpose' function for SoftImage NURBS are in this module.

`blending.c`, `oslo.c` and `convert.c` are described in Section 9.0, "'Library' description," on page 47.

7.3.2 Initialisation phase

Normally, one would allocate the instance data in the `init()` function. I needed the instance data already in the `setup()` function, which stores some data in it. This means that `init()` only needs to allocate the instance data if the effect is loaded.

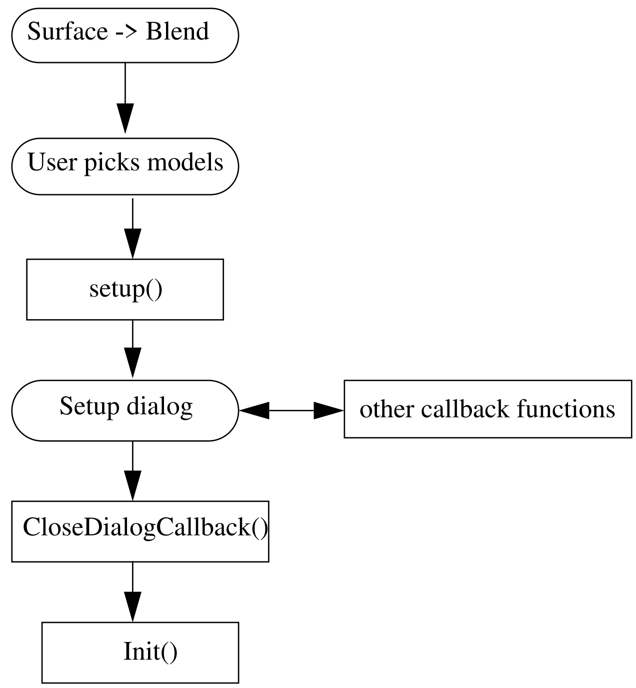
So what happens is (see also the figure below):

- `setup()` always allocates the instance data. It then extracts the projection curves from the argument surfaces and stores them in the *surfaces* field of the instance data.
- When the setup dialogue is closed, the `CloseDialogCallback()` function stores the curves selected by the user in the *surface1Curves* and *surface2Curves* fields of the instance data. This function also stores the number of arguments on the dialogue. It finally frees the memory occupied by the *surfaces* field of the instance data.
- `init()` checks if the effect is being created or loaded from disk. If it is being created, `init()` knows `setup()` has initialised the instance data and it only has to retrieve the instance data from the custom context and retrieve the projection curves from the *surface1/2Curves* fields.

  If it is being loaded, `init()` has to allocate the instance data and determine which projection curves are used, store these curves in the *surface1/2Curves* fields of the instance data.

  Finally, if the surface is created from scratch, `init()` creates the blending surface.

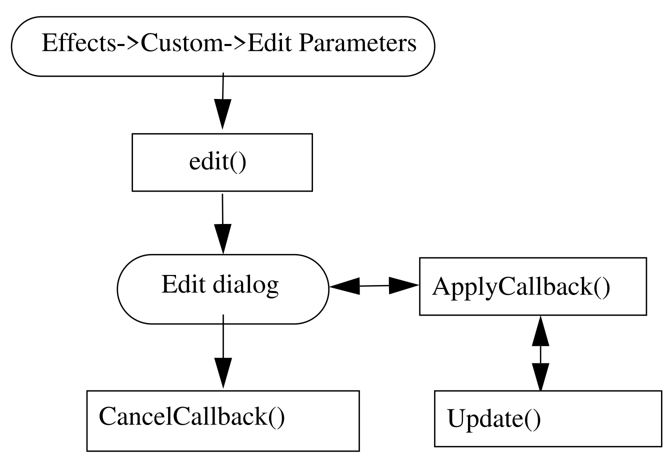
FIGURE 4. Initialisation program structure



7.3.3 Editing parameters of a blend surface

The figure below describes the structure of the *Edit Parameters* part of the custom effect.

FIGURE 5. Edit parameters program structure



7.4 blend.c

In this section the implementation of the surface blending custom effect (source file `blend.c`) is described.

7.4.1 Instance data

The fields of the instance data are described in the table below.

TABLE 4. Instance data fields

Field name	Description
result	last (error) code that was returned
scene	current scene
nbArgs	number of arguments picked by the user
nbIntmCurves	number of intermediate curves
surfaces	structure that is used during the setup phase of the creation of the blend surface. It is used to store all projection curves and extracted projection curves of the surfaces picked by the user.
preview	the preview blend surface (only used during the setup dialogue phase)
surface1Curves, surface2Curves	inner and outer curves of both surfaces. If constant u/v curves are used for surface 2, the field surface2Curves is not used.
r1, r2	curvature parameters
constantParam	flag indicating whether to use constant u/v curves instead of projected curves for the second surface.
constU	flag indicating whether the u or the v direction is constant
innerValue, outerValue	parameter values for constant u/v curves
updateFlag	The updateFlag is used to prevent this effect from updating the blend surface too often: SoftImage calls update for each result definition line in the custom definition file. This custom effect has 2 result definitions. We toggle the flag in update() so that of each two calls of update() only one real update takes place.We do not use the SAA_customContextGetCurrentResult() function because it does not return the right indexes.

7.4.2 init()

The `init()` function is version 1.0 style because of the bug in the `SAA_customContextSetResult()` function (see Section 6.2 on page 31).

If the blend surface is created from scratch, it performs the actual creation of the blend surface using the curves specified by the user. If the blend surface is loaded, `init()` takes care of the initialisation of instance data. In ‘pseudo-code’ the `init()` function looks like this:

```
init()
  if (the result already exists) then
```

```
    allocate instance data
    retrieve nr of arguments from the dialog box
    get the ids of the projection curves from the dialog box
    use these ids to get projection curve data from SI
     and store data in instance data
     (fields surface1Curves and surface2Curves)
  else
    get pointer to instance data
  end if

  get parameter values from the dialog box and store them
   in the instance data

  if (the result does not yet exist) then
    extract the projected and other curves
     (using getCurves())
    get intermediate curves (if any) from argument array
    create blend surface (using do_blending2() )
    convert surface to SI
    destroy the curves that have been extracted
    tell SI this surface is the result of the custom effect
  end if
```

7.4.3 update()

The `update()` function updates the blend surface. It extracts the inner and outer curves using `getCurves()`, gets the intermediate curves from the argument array, calls `do_blending2()` to create the surface. Then the control vertices and knots are copied from the result of `do_blending2()` to the actual blend surface object in SoftImage.

`Update()` uses the surface blending parameters from the instance data.

Before I copy control vertices and knots to the SoftImage blend surface (using the `copyPoints()` function from `convert.c`), I save the *step* parameter. The reason for this is that `copyPoints()` uses `SAA_nurbsSurfaceEdit()`, which resets the *step* parameter:

```
SAA_nurbsSurfaceGetStep( &idata->scene, &blendedSurface[0],
  &stepU, &stepV );
copyPoints( &prBlend, &blendedSurface[0], &idata->scene,
  FALSE, FALSE );
SAA_nurbsSurfaceSetStep( &idata->scene, &blendedSurface[0],
  stepU, stepV )
```

`Update()` toggles the *updateFlag* in the instance data and performs the actual update only when this flag is true to prevent unnecessary updates.

7.4.4 cleanup()

`Cleanup()` frees the memory occupied by the instance data.

7.4.5 setup()

The `setup()` function sets up the setup dialogue. This means:

1. allocating the instance data
2. getting the projected curves on the surfaces the user has picked
3. enabling and disabling specific dialogue items
4. installing callback functions for the dialogue

The first items are done by the function `initInstanceData()`. Item 3 includes putting the names of the extracted projection curves of the surfaces into the text view items on the dialogue.

The `setup()` function puts the names of the projection curves of the surfaces into the four text view items.

The following callback functions are installed: `closeDialogCallback()` to handle the pressing of the *Ok* and *Cancel* buttons; `selectCurveCallback()` to highlight the curve the user has selected in the text views; `toggleCallback()` which is attached to the check box to choose between projected curves and constant u/v curves for surface 2; `previewCallback()` which is attached to the *Apply* button, to create a preview blend surface.

7.4.6 setup callback functions

7.4.6.1 previewCallback()

The function `previewCallback()` creates a blending surface of the curves that the user has selected on the dialogue, so that the user can preview the blending surface. The preview surface is stored in the *preview* field of the instance data.

7.4.6.2 toggleCallback()

When the check box to choose between projected curves and constant u/v curves for surface 2 is clicked, `toggleCallback()` will enable and disable (according to the state of the check box) the text view items for surface 2 and the u/v parameter items.

7.4.6.3 selectCurveCallback()

The function `selectCurveCallback()` checks which curve in which text view items has been selected, uses the *surfaces* field of the instance data to retrieve the corresponding SoftImage curve and selects this curve by putting it into the selectList.

7.4.6.4 closeDialogCallback()

The function `closeDialogCallback()` checks if the *Ok* button was clicked. If so, the curves selected in the text view items are stored in the *surface1Curves* and *surface2Curves* fields of the instance data. The ids of these curves are stored in a number of fields on the dialogue (it would be better to use user data of models for this, but the user data feature of SAAPHIRE contains some bugs).

This function also stores the number of arguments on the dialogue (see also Section 6.3, “Argument and result count bug,” on page 32).

If the user has not selected a curve in one of the text views, a default curve is selected by `closeDialogCallback()`.

`CloseDialogCallback()` always destroys the extracted projection curves of the surfaces and frees the memory occupied by the *surfaces* field of the instance data. It also destroys the preview surface if it has been created.

7.4.7 edit()

The `edit()` function sets up the edit parameters dialogue. This means:

- enabling and disabling specific dialogue items
- installing callback functions for the dialogue

The following callback functions are installed: `applyCallback()` is attached to items like the *Ok* button, *Apply* button and the items that modify the curvature parameters; `cancelCallback()` is attached to the *Cancel* button.

7.4.8 edit callback functions

7.4.8.1 applyCallback()

The `applyCallback()` function reads the blending surface parameter values from the dialogue items and stores these values in the instance data. Then it calls `update()` to apply any changes to the blending surface.

7.4.8.2 cancelCallback()

The `cancelCallback()` function restores the blending surface parameters to their old values by reading them from the custom value list and storing these values in the instance data. Then it calls `update()` to restore the blending surface.

7.4.9 initInstance()

The function initialises SAAPHIRE by calling the `SAA_sceneInit()` function. It then allocates memory for the instance data and stores the current scene in it.

7.4.10 initInstanceData()

The function `initInstanceData()` first calls `initInstance()` to allocate instance data memory. Then it checks the arguments picked by the user. The first and the last arguments should be NURBS surfaces, the rest should be NURBS curves.

If all the arguments are ok, `initInstanceData()` calls `getProjectionCurves()` to retrieve projecting curve data into the *surfaces* field of the instance data. The function also checks if surface 1 has at least one projection curve. If not, the custom effect will be cancelled.

7.4.11 getProjectionCurves(SAA_Scene * scene, SAA_Elem * srf, ProjectionCurvesInfo * info)

The function `getProjectionCurves()` gets the projection curves of a given NURBS surface and stores them in the given *ProjectionCurvesInfo* structure. If the surface does not have any projection curves, the *nbCrv* field will become 0 and the other fields will be set to NULL.

7.4.12 getCurves(InstanceData * idata, SAA_Elem * curves, SAA_Elem * srf1, SAA_Elem * srf2)

The function `getCurves()` extracts the curves from the given surfaces and returns them in *curves*. *curves* should be an array of 4 elements. Element 0 will become the inner curve of surface 1, element 1 the outer curve of surface 1, element 2 the inner curve of surface 2, element 3 the outer curve of surface 2.

For surface 2, `getCurves()` checks the *constantParam* field of the given instance data. If it is true, it extracts two surface curves with the u or v parameters as specified by the *innerValue* and *outerValue* fields of the instance data.

7.4.12.1 do_blending2(SAA_Elem * curves, int nbCurves, PR_nurb * out, SAA_Scene * scene, double r1, double r2)

This function converts the given NURBS curves (in SoftImage format) to NURBS Procedure Library format and normalises them, so that they all have the same degree, knot vector and number of control vertices. All the curves are clamped using the `clampNURBScurve()` function.

Then the `blender()` function is called, which performs the actual creation of the blend surface. The curvature parameters r1 and r2 are passed to `blender()`.

nbCurves should be the number of curves in the *curves* array. The first two and last two elements of *curves* will be the inner and outer curves. The other curves, if any, will be the intermediate curves.

7.5 Suggestions for improvement

- A way of improving the speed is to reorganise the memory management. At the moment, each time the blend surface is updated, there takes place a lot of allocating and freeing of memory. If we could reduce this, the effect will be faster.

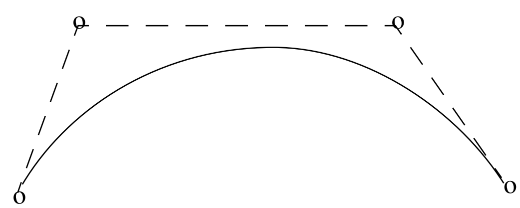
8.0 Motion tool

8.1 Hypersurface animation tool (introduction)

The hyper-surface animation tool allows the user to create a hyper-surface animation based on two or more NURBS surfaces. It lets the user pick two or more surfaces and creates a hyper-surface from it.

Animation using hyper-surfaces is a concept which is quite different from function curves, which are the regular way of creating animation in SoftImage. The idea of hyper-surface animation is that the animation is represented by a 4D surface (a so called hyper-surface).

A hyper-surface is a generalisation of curves and surfaces. As with curves and surfaces, which have control points that generally don't lie on the curve or surface itself (see picture below), the hyper-surface is based on a number of surfaces (called sections), which (apart from the first and last one) generally don't lie on the hyper-surface itself.



The hyper-surface represents the behaviour in time of an animated surface. To get the animated surface at a specific point in time, the hyper-surface will be evaluated at that point on its time axis, which results in a NURBS surface.

In SoftImage, you will not see the hyper-surface itself, but only the animated surface corresponding to the current frame.

For a more detailed description of using hyper-surfaces for animation, see [Preston94].

This motion tool differs from the model tool in that it uses operator attributes to enable the user to animate an attribute of the custom effect. Furthermore, the hyper-surface itself is not a SoftImage object; it is stored in the instance data. The result surface in SoftImage is only the result of evaluating the hyper-surface at the current time.

8.2 Source files

The hyper-surface animation custom effect consists of the following source files:

hyper.c. Contains the custom effect 'body': processing the dialogue, processing the surfaces picked by the user, cleaning up, etc.

blending.c. Contains a number of function that operate on NURBS in NURBS Procedure Library format, like the blending algorithm, a NURBS normalisation algorithm and NURBS clamping functions.

oslo.c. Contains the oslo algorithm for increasing the number of knots/vertices (functions adapted from Graphics Gems IV)

convert.c. Contains a number of conversion functions and functions which implement some aspects of conversion between SoftImage NURBS format and the NURBS Procedure Library format. Also some ‘general purpose’ functions for SoftImage NURBS are in this module.

misc.c. Contains a number of generally applicable functions.

`blending.c`, `oslo.c`, `convert.c`, and `misc.c` are described in Section 9.0, “‘Library’ description,” on page 47.

8.3 hyper.c

In this section the implementation of the hyper-surface animation custom effect (source file `hyper.c`) is described.

8.3.1 Instance data

The fields of the instance data are described in the table below.

TABLE 5. Instance data fields

Field name	Description
result	last (error) code that was returned
scene	current scene
nbArgs	number of arguments picked by the user
hyperSrf	the hyper-surface in NURBS Procedure Library format
fStart, fEnd	the start and end frames of the hyper-surface animation
autoFrame	flag indicating whether to use fStart and fEnd or to use the current start and end frames
weights	array of weight factors. For each section there is a weight factor
updateFlag	The updateFlag is used to prevent this effect from updating the hyper-surface animation too often: SoftImage calls update for each result definition line in the custom definition file. This custom effect has 2 result definitions. We toggle the flag in update() so that of each two calls of update() only one real update takes place.We do not use the SAA_customContextGetCurrentResult() function because it does not return the right indexes.

8.3.2 init()

The `init()` function is version 1.0 style because of the bug in the `SAA_customContextSetResult()` function (see Section 6.2 on page 31).

`Init()` initialises the instance data, retrieves the number of arguments and the weight factors from the dialogue and stores them in the instance data (the reason why the weight factors are stored on the dialogue is described in Section 6.6, “User data functions," on page 33). It then creates the hyper-surface from the surfaces picked by the user.

If the hyper-surface animation is created from scratch, `init()` evaluates the hyper-surface at some point in time[1], converts the result to SoftImage and passes it as the result to SoftImage.

8.3.3 update()

The `update()` function updates the current frame of the hyper-surface animation by evaluating the hyper-surface at the current time. It uses the *Time* operator attribute function curve to map the SoftImage time axis onto the hyper-surface time axis.

The result of the evaluation is copied into the SoftImage result element using the `copyPoints()` function from `convert.c`.

If the user has specified start and end frame numbers, update() will use these. If the current frame is smaller than the start frame or larger than the end frame, the evaluation point will be set to 0.0 or 1.0 respectively.

The value of the *Time* operator attribute is clipped to the range [0, 1], because the behaviour for the hyper-surface outside this range is not defined and the user can specify values outside this range.

`Update()` toggles the *updateFlag* in the instance data and performs the actual update only when this flag is true to prevent unnecessary updates.

8.3.4 cleanup()

`Cleanup()` frees the memory occupied by the instance data.

8.3.5 setup()

The `setup()` function sets up the setup dialogue. This means:

1. enabling and disabling specific dialogue items
2. installing callback functions for the dialogue

The following callback functions are installed: `okCallback()` to handle the pressing of the *Ok* button.

8.3.6 okCallback()

The `okCallback()` function takes care of storing a number of parameters on the dialogue box. This is needed because the model user data feature of SAAPHIRE is not yet working. The function stores the number of arguments and initialises the weight factors (on the dialogue) with the value 1.0.

8.3.7 edit()

The `edit()` function sets up the edit parameters dialogue. This means:

1. An arbitrary time can be chosen, because update() will update the result directly after init() has created it. Update() will use the correct time.

1. enabling and disabling specific dialogue items
2. putting a string into the text view item for each section
3. installing callback functions for the dialogue

For each section of the hyper-surface, the `edit()` function adds a string ‘Section#’ to the text view item.

The following callback functions are installed: `updateCallback()` is attached to the *Update* button; `weightCallback()` is attached to the *Set Weight* button and to the sections text view.

8.3.8 edit callback functions

8.3.8.1 weightCallback()

This function first checks the *dialogItemId* parameter to find out which item has triggered this callback. If it was triggered by the *Set Weight* button, the value of the weight factor text edit item is written to the instance data as the weight factor for the section that is currently selected in the text view item.

If this callback was triggered by the text view item, the weight factor of the currently selected section in the text view is shown in the weight factor text edit item.

8.3.8.2 updateCallback()

This callback function re-creates the hyper-surface and stores the weight factors in the instance data in the dialogue.

8.3.9 initInstance()

The function initialises SAAPHIRE by calling the `SAA_sceneInit()` function. It then allocates memory for the instance data and stores the current scene in it.

8.3.10 createHyperSurface(SAA_Scene * scene, SAA_Elem * args, int nbSections, float * weights, PR_nurb4 * out)

This function creates a hyper-surface in NURBS Procedure Library format, based on a number of sections in SoftImage format. *args* is an array of SoftImage NURBS surface with *nbSections* elements; *weights* is an array of weight factors, one weight factor for each section.

The result will be placed in *out*. The calling function should allocate memory for *out*.

This function first converts all the NURBS surfaces to NURBS Procedure Library format. If the surface is closed in the u or v direction, the converted surface is clamped in that direction (using the `clampNURBSsurface()` function from `blending.c`).

It also uses the `adjustWeights()` function to apply the weight factors to the converted surfaces.

9.0 ‘Library’ description

9.1 Introduction

In this chapter I will describe the functions of the following source files: `convert.c`, `blending.c`, `oslo.c`, and `misc.c`. All these files contain functions that can be reused when creating other custom effects.

The header file of `convert.c` is called `convert.h`. The blending.c and oslo.c share a header file called `blending.h`. The header file of `misc.c` file is called `misc.h`.

blending.c. This module contains a number of function that operate on NURBS in NURBS Procedure Library format (see [Hewitt93]), like the blending algorithm, a NURBS normalisation algorithm and NURBS clamping functions.

oslo.c. This module contains the oslo algorithm for increasing the number of knots/vertices (these functions have been adapted from [Peterson94])

convert.c. This module contains a number of conversion functions and functions which implement some aspects of conversion between SoftImage NURBS format and the NURBS Procedure Library format. Also some ‘general purpose’ function for SoftImage NURBS are in this module.

misc.c. This module contains a number of miscellaneous functions.

9.2 convert.c

9.2.1 convertNURBSToSI (PR_nurb * PR_src, SAA_Elem * SI_dest, SAA_Scene * scene)

The function `convertNURBSToSI()` converts the given NURBS curve (in NURBS Procedure Library format) or surface *src* to SoftImage format. The result will be created in the given scene *scene* and will be stored in *SI_dest.* The calling function should allocate memory for *SI_dest*. Note that the SoftImage NURBS will be open.

9.2.1.1 Implementation notes

- This function checks if the NURBS is a surface or a curve. SoftImage has different object types for NURBS curves and surface.
- Normally, the vertices are in the form $(w \cdot x, w \cdot y, w \cdot z, w)$. SoftImage however stores them in the form (*x, y, z, w*). In `convertNURBSToSI()` we have to divide the *x, y, z* coordinates by the weight before passing them to SoftImage.
- We ignore the first and the last knot of the knot vector: SoftImage always assumes these are equal to the second and second-last knots.

9.2.2 convertNURBSFromSI (SAA_Elem * SI_src, PR_nurb * PR_dest, SAA_Scene * scene)

The function `convertNURBSFromSI()` converts the given SoftImage NURBS curve or surface *SI_src* (from the given scene) to NURBS Procedure Library format. The result will be stored in *PR_dest*. The calling function should allocate memory for *PR_dest*.

If the NURBS is closed, the NURBS that will be created will look exactly like the closed SoftImage NURBS. The algorithm that is used is described in Section 5.3.3, "Closed NURBS," on page 29. Note that the result will be unclamped; use `clampNURBScurve()` or `clampNURBSsurface()` to clamp it.

9.2.2.1 Implementation notes

- This function uses two functions, `convertCurveFromSI()` and `convertSurfaceFromSI()`. These functions are actually split in two parts: the first part extracts all the necessary information from the SoftImage NURBS. Then they call the function `createPRNURBS()` to create the actual NURBS.
- When copying the control vertices we also take into account the way SoftImage stores the vertex coordinates: we multiply *x, y, z* by the weight.

9.2.3 scaleKnotVector(PR_dir * knotVector, double newMax)

This function scales a given knot vector *knotVector*. One can only specify the new upper limit of the knot vector, *newMax*.

I have written this function because the knot vector scaling function of the NURBS Procedure Library contains a bug.

9.2.4 copyPoints(PR_nurb * src, SAA_Elem * dest, SAA_Scene * scene, SAA_Boolean closeU, SAA_Boolean closeV)

The function `copyPoints()` copies the vertices and knots from the *src* (in NURBS Procedure Library format) to the SoftImage NURBS *dest* (from the given scene).

The parameters *closeU* and *closeV* specify whether the SoftImage NURBS should be closed in the U and V directions. I have included these parameters because closing SoftImage NURBS is a costly operation and doing it in the `copyPoints()` function is less costly.

Both NURBS should have the same number of vertices and the same number of knots, otherwise undefined behaviour may occur.

Note: The copying of vertices only affects the local coordinates of the vertices of an object. The coordinates of its centre remain unchanged.

9.2.5 nurbsSurfaceSetClosed(SAA_Scene * scene, SAA_Elem * srf, SAA_Boolean closedU, SAA_Boolean closedV)

The function `nurbsSurfaceSetClosed()` closes or opens the given NURBS surface *srf* from the given scene. The parameters *closedU* and *closedV* specify whether the surface will be opened or closed in the u and v directions.

9.2.6 dirtyCopy(SAA_Scene * scene, SAA_Elem * srf, SAA_Elem * dest)

This function makes a copy of the NURBS surface *srf* and returns it in *dest*. The calling function should allocate memory for *dest*. The copying is done by retrieving all the data from the source and then creating a new NURBS with this data.

9.2.7 globalize(SAA_Elem * src, PR_nurb * dest, SAA_Scene * scene)

This function, which takes a SoftImage NURBS and a NURBS in NURBS Procedure Library format, makes the coordinates of the control vertices of the latter ‘global’, using the SoftImage NURBS. This function is needed because the function to retrieve the coordinates of the control vertices of a SoftImage NURBS (the function `SAA_modelGetVertices()`) returns local coordinates (coordinates relative to the centre of the NURBS). See also Section 4.7, “Notes and tips for programming custom effects,” on page 24.

9.2.8 dumpNURBS(FILE * f, SAA_Scene * scene, SAA_Elem * srf)

The function `dumpNURBS()` writes information about the given NURBS *srf* to the file stream *f*. The following properties are written: degree, number of vertices and knots, closed/open, knot vector.

9.2.9 readNURBS(const char * inputFile, SAA_Scene * scene)

This function reads a NURBS from the file named *inputFile* using the NURBS Procedure Library and converts it into a SoftImage NURBS using the conversion functions.

9.2.10 writeNURBS(const char * outputFile, SAA_Scene * scene, SAA_Elem * SI_NURBS)

This function converts an SoftImage NURBS into NURBS Procedure Library format and saves it to a file named *outputFile* using the NURBS Procedure Library.

9.3 blending.c

9.3.1 blender(PR_nurb * curves, int nbCurves, PR_nurb * result, double r1, double r2)

This function creates a NURBS surface from an array of NURBS curves and returns it in *result*. The calling function should allocate memory for *result*.

The array *curves* should contain at least 4 curves:

- curves[0] = outer curve of surface 1
- curves[1] = inner curve of surface 1
- curves[nbCurves-2] = inner curve of surface 2
- curves[nbCurves-1] = outer curve of surface 2
- All other curves are treated as intermediate curves.

The ‘inner’ curves are treated in a special way: the parameters r_1 and r_2 are used to shift the inner curves to or from the outer curves. If v_1 is a point on the inner curve and v_2 the corresponding point on the outer curve, than the actual point v_3 will be:

$$v_3 = v_2 + r \cdot (v_1 - v_2) \qquad \text{(EQ 2)}$$

This function first normalises all the curves so that they all have the same knot vector and the same number of control vertices.

9.3.2 normalise(PR_nurb * list, int direction, int num_sections)

This function normalises an array of NURBS curves, i.e. it ensures all curves are elevated to the correct order, and all curves have the correct number of control points. This function is an adapted version of a function originally written by Martin Preston.

9.3.3 clampNURBScurve(PR_nurb * src, PR_nurb * dest)

The function `clampNURBScurve()` clamps an unclamped NURBS curve according to the algorithm described in [Piegl95], p. 576. The resulting curve is put into *dest*. The calling function should allocate memory for *dest*.

9.3.4 clampNURBSsurface(PR_nurb * src, PR_nurb * dest)

The function `clampNURBSsurface()` clamps an NURBS surface in the u-direction. The surface *src* should be unclamped in the u-direction. This function uses the algorithm described in [Piegl95], p. 576. The resulting surface is put into *dest*. The calling function should allocate memory for *dest*.

9.3.5 void InsertKnot(Ppoint4 * ptsSrc, Pfloat * knotsSrc, int np, int order, Ppoint4 * ptsDest, Pfloat * knotsDest, double u, int s, int r, int k)

The function `InsertKnot()` inserts a given knot value one or more times in the given knot vector of a NURBS curve. The parameters are described in the table below.

Parameter	Description
ptsSrc	control vertex array of source NURBS
knotsSrc	knot array of source NURBS
np	number of control vertices of source NURBS
order	order of NURBS
ptsDest	control vertex array of destination NURBS.
knotsDest	knot array of destination NURBS
u	value of the knot to be inserted
s	multiplicity of the knot in the original knot vector
r	number of times to insert the knot
k	index in the knot vector of src after which u will be inserted

Note that the calling function should allocate enough memory for *ptsDest* and *knotsDest*.

This algorithm has been adapted from [Piegl95], p. 151.

9.3.6 void RefineCurve(PR_nurb * src, PR_nurb * dest, PR_knots * newKnots)

This function refines the curve *src* using the knot vector *newKnots*. The result is placed into *dest*. The caller should allocate memory for *dest*.

The refinement is carried out by calling `InsertKnot()` for each knot to be inserted.

Note: this function is not used at the moment. It does not take into account any ‘special cases’ and has not really been tested.

9.4 oslo.c

This module contains the functions `FindBreakPoint()`, `CalcAlpha()`, and `RefineSurface()`. These functions have been taken from [Peterson94] and have been adapted to use the NURBS Procedure Library format.

`RefineSurface()` is an implementation of the Oslo algorithm and refines a given NURBS using a given knot vector. Note that the Oslo algorithm as it is implemented here implicitly assumes the NURBS to be clamped.

9.5 misc.c

9.5.1 void DeleteSectionList(SectionRecord ** the_list)

This function frees memory occupied by the given list of SectionRecords, and makes the pointer to it NULL.

9.5.2 void deallocate_nrb(PR_nurb * nrb)

This function deallocates the vertex vector and the knot vectors of the given NURBS, setting the pointers to NULL.

9.5.3 void adjustWeights(PR_nurb * current, double f)

This function adjusts weight of given NURBS by multiplying them by the given factor *f*.

10.0 References

[Hewitt93]. Hewitt, W.T.; Yip, D., *The NURBS Procedure Library*, CGU 76, 1993.

[Peterson94]. J.Peterson, W., 'Tesselation of NURBS Surfaces'. In: Heckbert, P.S. (ed.), *Graphics Gems IV*, Academic Press, 1994, p. 286-320. The Graphics Gems Series.

[Piegl95]. Piegl, L., Tiller, W., *The NURBS book*, Berlin, etc., Springer-Verlag, 1995. Monographs in visual communications.

[PrestonWWW]. Preston, M., *Animation using NURBS*, http://info.mcc.ac.uk/CGU/research/animation/nurbs_animation.html.

[Preston94]. Preston, M., Hewitt, W.T., Animation using NURBS, *Computer Graphics Forum* **13(4)**, 1994, p. 229-241

[SI]. SoftImage documentation (printed and HTML), including SoftImage 3D Quick Start and SoftImage 3D Workbook.

[Watt92]. Watt, A., Watt, M., *Advanced animation and rendering techniques, Theory and practice*, New York, NY, ACM Press, 1992.