Tutorials

CraftMan

For CraftMan version 1.0.

Under the copyright laws, neither the documentation nor the software may be copied, photocopied, reproduced, translated or reduced for any electronic medium or machine-readable form, in whole or in part, without the written permission of Xanthus International AB, except in the manner described in the documentation.

All rights reserved.

Stockholm, May 11, 1992

Table of Contents

Table of Contents **iii**

Introduction **1**

Tutorial One: Your First Program **3**

CraftMan Basics 3
The First Program 5
Improving Your Program 26
Writing CraftScript 42

Tutorial Two: Using Timers **57**

What is a Timer? 57
Creating a Timer 57
Resetting a Timer 64

Tutorial Three: A New Class **69**

Classes and Objects 69
The Class ‘memory_button’ 69
Using a User-defined Palette 84
Export a Program to a Text-file 85

Tutorial Four: Using the Database **87**

The Database Object 87
A Phone Book 88
A Find Function 95
Exporting/Importing Data 100

Introduction

This book of CraftMan tutorials is the best way to start using CraftMan. We suggest that you start with at least Tutorial One before reading the Users Guide.

Tutorial One is an introduction to CraftMan. You can complete this tutorial step by step without any prior CraftMan experience. Tutorial One introduces tools like the *ToolBox*, the *Inspector* and the *Color panel*. It also introduces things like *window*, *button*, *slide, slide-show*, *image*, *sound*, *text-field, movie, slider and CraftScript.*

Tutorial Two describes how to use timers. You should have completed Tutorial One first. Tutorial Two introduces things like *timer, super* and *inspecting the program*.

Tutorial Three describes how to use user-defined classes and user-defined palettes. You should have completed Tutorial One first. Tutorial Three introduces tools like the *Class Browser* and the *Console* and things like *sub-classing* and *user-defined palettes*.

Tutorial Four describes how to use the database object and the Database Manager. You should have completed Tutorial One first. Tutorial Four introduces tools like the *Database Manager* and things like *box, control-dragging to the Inspector* and *using the Font panel*.

All the completed tutorial programs can be found in the *Demo* list in the *Info panel*.

Tutorial One: Your First Program

This CraftMan tutorial consists of three parts:

1. In the first section you will create a small CraftMan program.
2. In the second section you will improve the program with visual effects and sound.
3. In the last section you will write some CraftScript code to enhance the functionality of the program.

All the three example programs can be found in the *Demo Programs* list in the *Info panel*.

Note: When there is a description of something that you should do, the sentence will be printed in boldface on its own line and will be preceded by the symbol ↔.

For example if you are supposed to click a button with the title *New*, it will look like this:

↔ **Click the button titled *New*.**

To follow the tutorial make sure you do everything that is marked with a ↔.

CraftMan Basics

Before you start the tutorial, you will learn some basic CraftMan vocabulary.

First you have to start CraftMan.

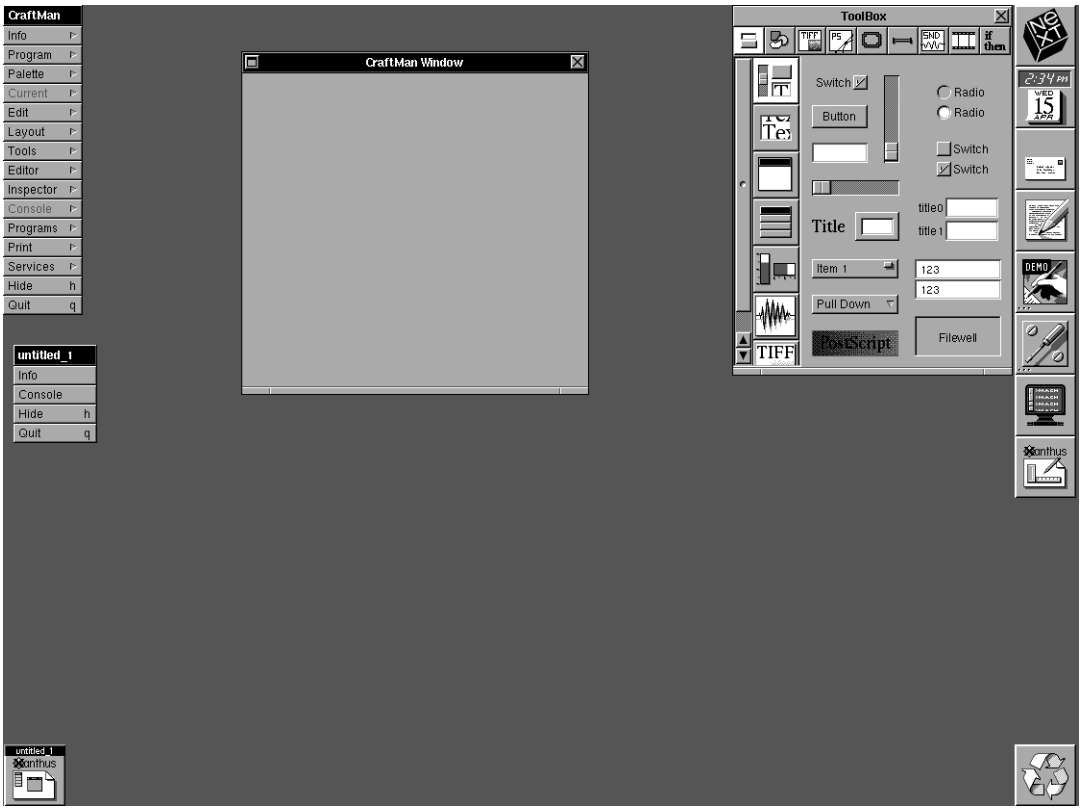
↔ **Double-click the CraftMan program icon in the Workspace.**

Picture 1. The CraftMan program icon.

Note: If you do not have CraftMan installed on your system, refer to the book 'Getting Started' for installation information.

- On the screen will appear four windows.
- One small menu-window titled *untitled_1* at the left side of the screen.
- One larger window titled *CraftMan Window* at the top of the screen.

- One window titled *ToolBox* in the upper right corner of the screen.
- One small mini-window titled *untitled_1* in the lower left corner of the screen.



Picture 2. The screen when CraftMan is started.

- The menu titled *untitled_1* is the main menu for your new program.
- The large window is the first window in your new program. This is where you will put objects in your program.
- The ToolBox window is the window from which you pull objects to your program.
- The small mini-window is the mini-window representing your CraftMan program. Double-clicking on this window is a way to unhide the program if it is hidden in the Workspace.

A program can be in one of two modes: *edit-mode* and *run-mode*. Your program is now in edit-

mode. You can instantly switch to run-mode by choosing *Run* in the menu *Program*, and back to edit-mode by choosing *Edit* in the menu *Program*.

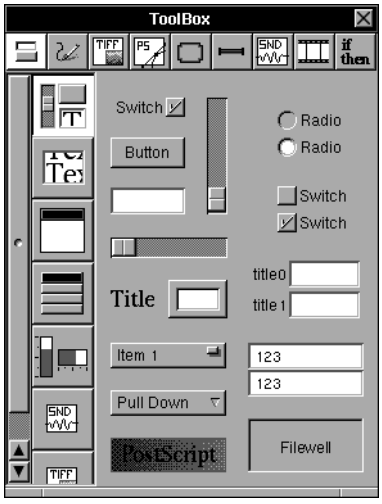
Note: It is useful to learn the command-key equivalents Command-r for Run and Command-e for Edit, since you will be switching a lot.

You can instantly see what mode you are in by looking for the second menu-window at the left. It will only be visible in edit-mode.

The First Program

You will now create a new program in CraftMan step by step from scratch. The program will be a little slide-show with animals.

First take a look at the ToolBox window.



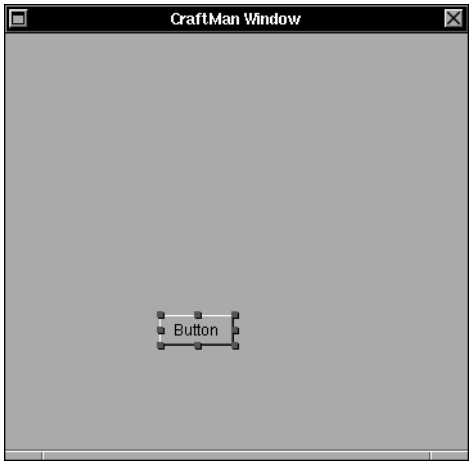
Picture 3. The ToolBox window.

This is the window from which you can pull objects into the windows of your program.

↔ **Pull the button from the palette to the window titled “CraftMan Window”.**

The interior of the window will get a black frame when the button is dragged over it. If you

drop the button over the window, it will stay there.



Picture 4. A dropped button with handles.

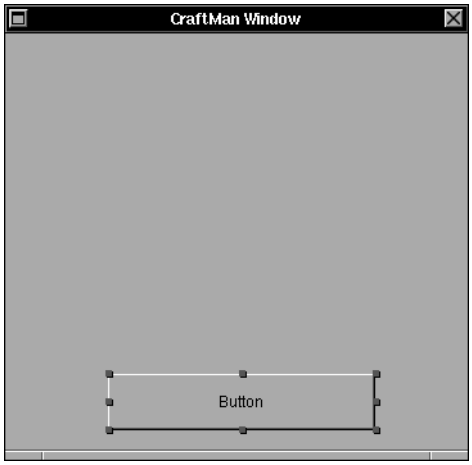
When the button is dropped in the window it gets *handles*. The handles are the eight small gray squares surrounding the button. They indicate that the button is selected. If you click anywhere in the window (but not on the button) the selection is cleared and the handles disappear.

To select something, just click on it.

↔ **Click on the button to select it.**

You can move the button by dragging it with the mouse and resize it by dragging the handles.

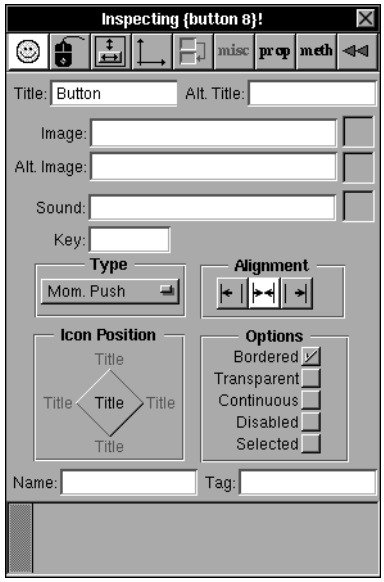
↔ **Move and resize the button to make it look like this:**



Picture 5. The button is moved and resized.

↔ **Make sure the button is selected, then choose *Inspect* form the menu *Inspector.***

On the screen will appear a window in the lower right corner.



Picture 6. The Inspector window.

This is the Inspector you use to inspect objects in CraftMan.

↔ **Replace the text "Button" in the field Title with the text "Next animal" and press return.**

Picture 7. Setting the title of a button.

The next object you need for your program is a *stretch-frame object*. The stretch-frame object is not visible in the ToolBox, so you have to change to another tool in the ToolBox.

Right now you are using the palette tool. You can see the nine different tools in the row of but-

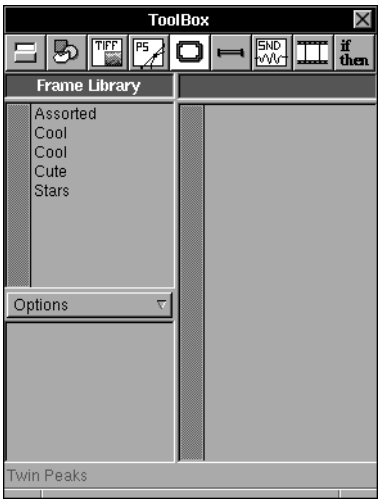
tons at the top of the ToolBox. These button are called the tool buttons.

Picture 8. The tool buttons.

↔ **Click the button for the stretch-frame tool.**

Picture 9. The stretch-frame tool button.

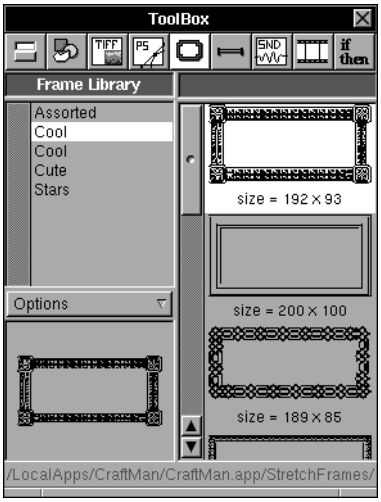
The ToolBox will now change to look like this:



Picture 10. The stretch-frame tool.

To the left is a list of stretch-frame libraries.

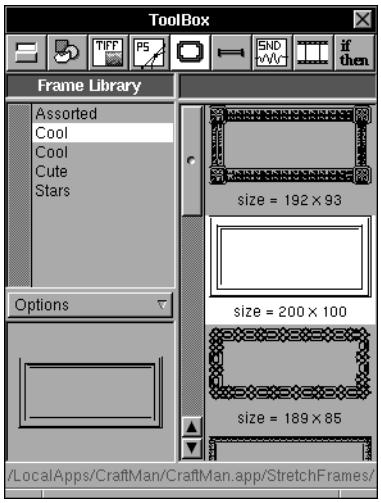
↔ **Select the library named ‘Cool’.**



Picture 11. The library 'Cool' is selected.

To the right will appear a list of all the stretch-frames in this library. The selected stretch-frame will appear at the lower left of the ToolBox. When a stretch-frame is visible there, it can be pulled to your window just like the button from the palette.

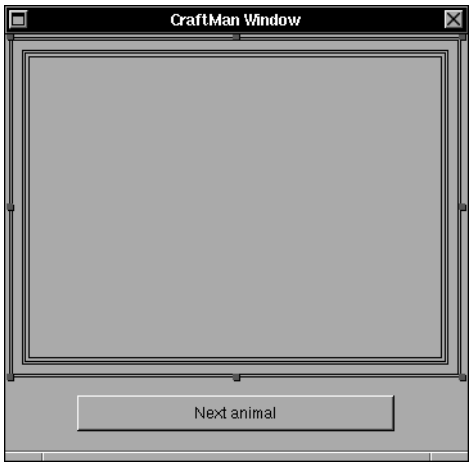
↔ **Select the second stretch-frame in the list.**



Picture 12. The second stretch-frame is selected.

↔ **Pull the stretch-frame from the ToolBox to the slide-show.**

↔ **Resize it by dragging in the handles to make it look like in the picture below.**



Picture 13. A window with a button and a stretch-frame.

The next object you need for this program is a *slide-show object*. The slide-show object is not visible in the ToolBox, so you have to change back to the palette tool again. (The tool button to the left.).

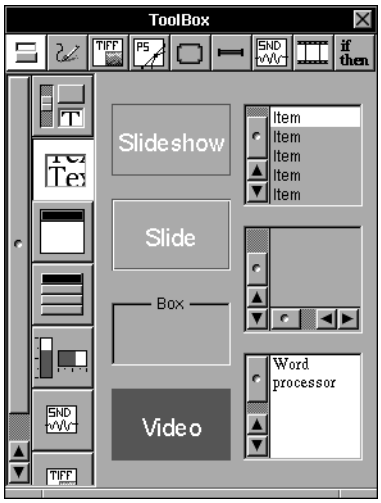
↔ **Select the palette tool in the ToolBox.**

The slide-show is found in another palette in the ToolBox.

↔ **Click the button for the slide palette.**

Picture 14. The button for the slide palette in the ToolBox.

The palette will now change to look like this:



Picture 15. The slide palette.

↔ **Pull the slide-show from the palette (the top left object) and drop it in the same window as the button.**

↔ **Place it and resize it like in the picture below.**

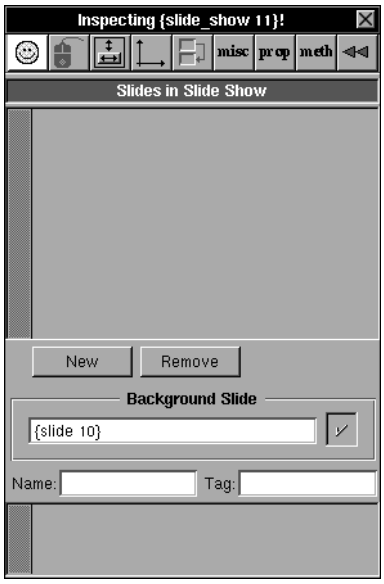
Note that the slide-show should fill the interior of the stretch-frame.

Picture 16. A window with a button, a stretch-frame and a slide-show.

↔ **Make sure the slide-show is selected in the window and choose *Inspect* from the menu *Inspector*.**

Note: If the menu command 'Inspect' is gray, check that the slide-show is selected (has eight

gray handles) and that the window is the key window (has a black title-bar).



Picture 17. Inspecting the slide-show.

The Inspector will now inspect the slide-show. You can see that the slide-show contains a list of slides. Right now the list is empty.

↔ **Name the slide-show *my_show* and press return**

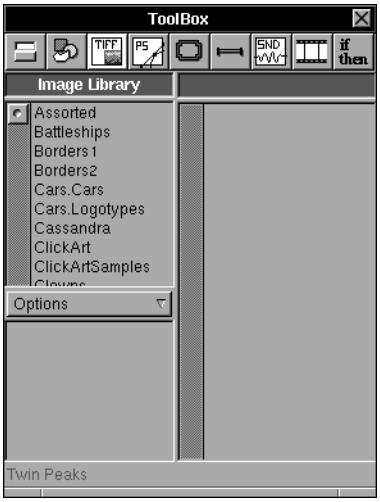
Picture 18. Naming the slide-show.

Now you want an image to put on the background of the slide-show. To do this you have to change tool in the ToolBox window.

↔ **Click the tool button for the image tool.**

Picture 19. The tool button for the image tool.

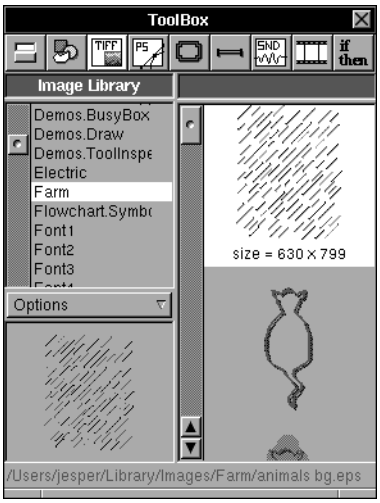
The ToolBox window will now change to look like this:



Picture 20. The image tool in the ToolBox.

To the left is a list of image libraries. Scroll the list of libraries down to a library called *Farm*.

↔ **Select the image library named 'Farm'.**



Picture 21. The library 'Farm' is selected.

In the list to the right will appear all the images in the selected library. You will use the image at the top as a background in your slide-show.

↔ **Select the image at the top by clicking on it.**

The selected image will appear at the lower left of the ToolBox (just like in the stretch-frame tool).

↔ **Pull the image from the ToolBox to the slide-show.**

Note that the slide-show will get a black border when the image is dragged over it to tell you that it will accept the image if it is dropped. When an image is dropped on a slide it is called a picture.

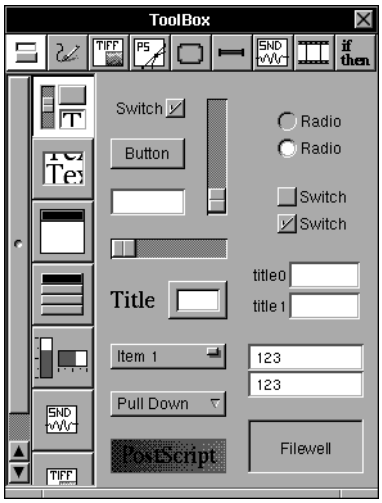
If you reshape the picture with its handles it will stretch to fill the slide-show.

Picture 22. An image dropped on the slide-show and resized.

Now you want a title on the background of the slide-show.

↔ **Select the palette tool in the ToolBox.**

↔ **Select the gadget palette (the palette at the top).**



Picture 23. The gadget palette in the palette tool.

↔ **Pull the text-field with the text “Title” from the ToolBox to the slide-show.**

Picture 24. A text-field dropped on the slide-show.

↔ **Double-click in the text-field to edit the text.**

Picture 25. Editing a text-field.

↔ **Change the text to “Animals” and press return.**

Picture 26. The text of the text-field is changed.

Now you want some slides in your slide-show.

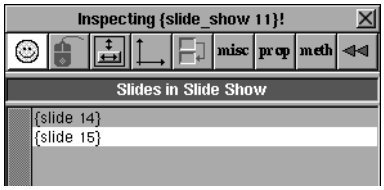
↔ **Make sure the slide-show is selected in the window and choose *Inspect* from the menu *Inspector.***

Note: If the menu command 'Inspect' is gray, check that the slide-show is selected (has eight gray handles) and that the window is the key window (has a black title-bar).

↔ **Click the button *New* in the Inspector.**

A slide appears in the list. Click the button again to create another slide. You should now have

two slides in the list.



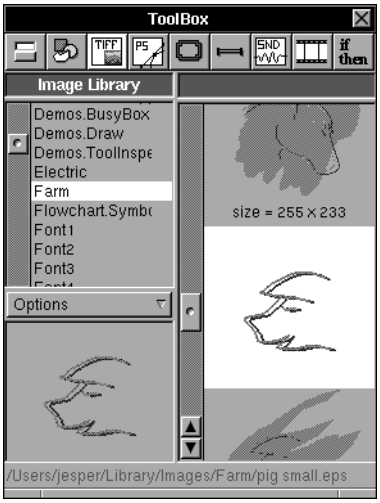
Picture 27. Two slides in the slide-show.

The current slide is the slide with its name highlighted. The current slide is displayed in the slide-show over the background. The reason why the background (with the title) is visible is that a slide is *transparent* by default. The current slide is on top of the background but you cannot see it since it is totally transparent.

Now you want to put a picture on the current slide.

↔ **Select the image tool in the ToolBox.**

↔ **Find the pig image in the list of images.**

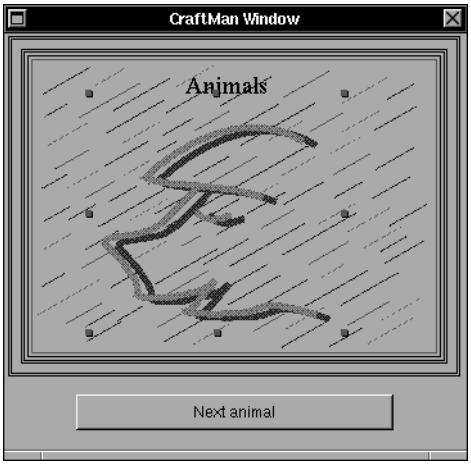


Picture 28. The pig is selected.

↔ **Select the pig image and pull it to the slide-show.**

↔ **Stretch the image in the handles so it becomes a bit larger.**

Now the current slide contains a picture of the pig. Your slide-show should now look something like this:

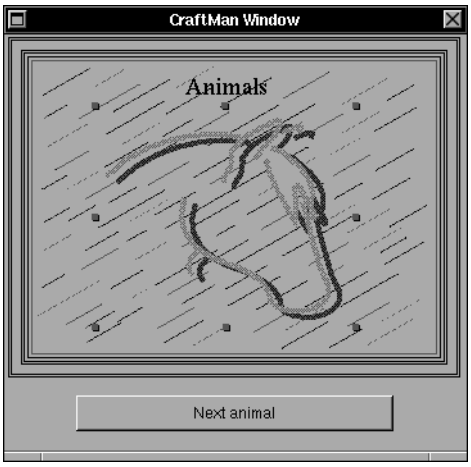


Picture 29. A pig picture on the current slide.

↔ **Change the current slide by clicking on the other slide in the list of slides.**

Note that the pig disappeared from the slide-show. The slide with the pig is no longer a visible slide in the slide-show. Only the slides that are highlighted in the Inspector are visible.

↔ **Find the horse in the ToolBox window and pull it to the empty current slide.**



Picture 30. A horse picture on the other slide.

Now you have a slide-show object with a background that contains a picture and a text-field as well as two slides with animal pictures. You also have a button titled “Next animal”.

What you want is to be able to control the slide-show with the button. A click on the button and the slide-show should change slide.

To do this, you need to tell the button to send the message *next_slide* to the slide-show every time the button is clicked.

↔ **Select the button and choose *Inspect* from the menu *Inspector.***

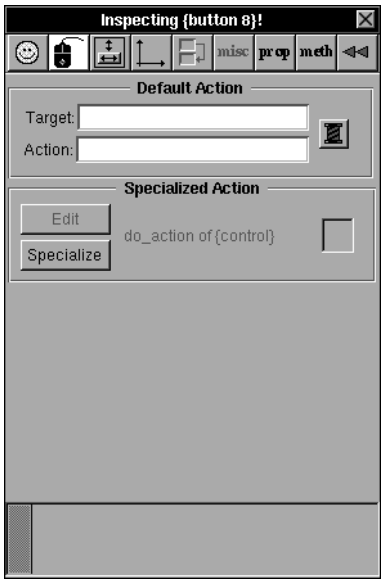
To edit the behavior of the button when it is clicked you have to inspect another *aspect* of the button. Right now you inspect the *appearance aspect* of the button. Now you want to inspect the *control aspect* of the button. The eight buttons at the top of the Inspector are called the *aspect buttons*. You click the button of the aspect you want to inspect.

Picture 31. The aspect buttons.

↔ **Click on the button for the control aspect.**

Picture 32. The button for the control aspect.

You now inspect the control aspect of the button.



Picture 33. Inspecting the control aspect of a button.

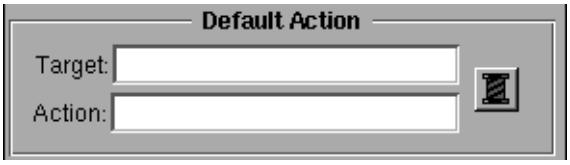
To tell a button what to do when it is clicked you can do one of two things.

1. Tell the button to send a message to another object when the button is clicked. To do this you do not need to write any CraftScript code.
2. Write CraftScript-code that will execute when the button is clicked.

You will use the first method here. The object that gets the message is called the *target* and the method that is sent is called the *action*.

To specify the target and the action you drag an *action-target connection* from the cable-drum

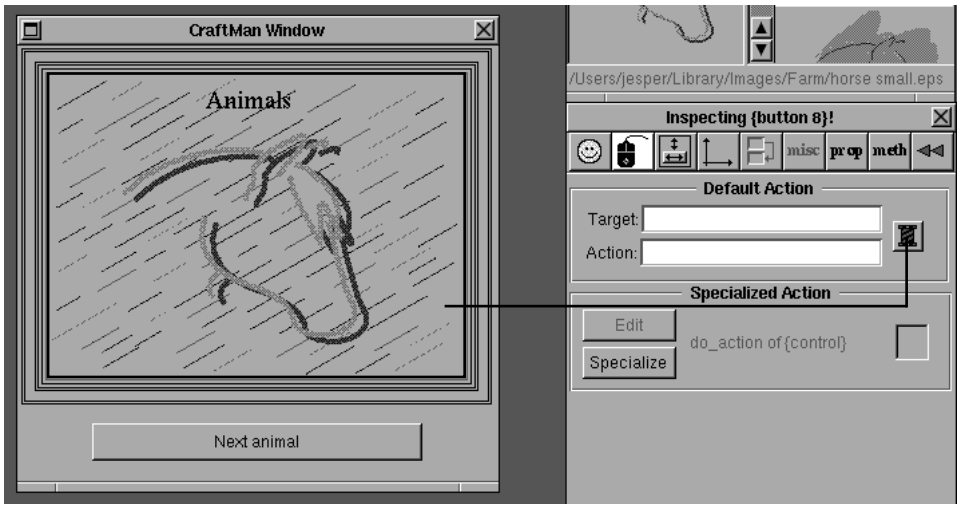
in the Inspector to the target object.



Picture 34. The cable-drum in the Inspector.

You put the cursor over the cable-drum and press the mouse-button to start the action-target connection.

↔ **Drag an action-target connection to the slide-show.**



Picture 35. Dragging an action-target connection.

Just like when you were pulling things from the ToolBox, the object that will become the target if you release the mouse will notify you by drawing a black frame around itself.

When you release the mouse over the slide-show, a window will appear with a list of messages that the slide-show understands.

Check that the list you get looks like this:

Picture 36. A list of action messages for the slide-show.

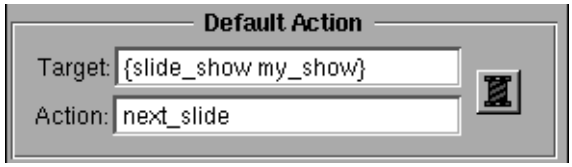
Note: If you have got another list, click 'Cancel' and try again. You have probably dragged the action-target connection to the wrong object.

↔ **Select the message *next_slide* and click the *OK* button.**

Picture 37. The message 'next_slide' selected.

When you have successfully dragged an action-target connection the Inspector should look like

this:



Picture 38. Action-target connection successfully made.

Now you have a working program and it is time to test it.

↔ **Choose *Run* in the menu *Program*.**

Note: When you switch to run-mode the slide-show automatically switches to the first slide.

↔ **Click the *Next animal* button to see the slides change.**

Now that you have crafted your first CraftMan program you want to save it to some other time.

↔ **Choose *Edit* from the menu *Program*.**

↔ **Choose *Save* from the menu *Program* and a Save panel will appear.**

↔ **Give your program a name and click the *OK* button.**

The program is now stored on disk and can be opened by choosing *Open* in the menu *Program*.

Note: If you are running a demo version of CraftMan or if your copy of CraftMan is not registered the 'Save' command will be disabled. Then continue with the next section without saving your program.

Improving Your Program

In the second exercise, you will do three things. First you will add visual effects to the slide-show, and then you will add graphics that you draw with CraftMan's ornament tools. Finally you will add a sound to the button.

↔ **Open the program you created in the previous section by choosing *Open* in the menu *Program*.**

When the program is opened, try it by clicking the button titled *Next animal*. The slide-show switches between the two animals at each click.

Now you will improve your program by adding visual effects to the slide show. These visual effects determine how the slides in the slide-show should switch from one to another.

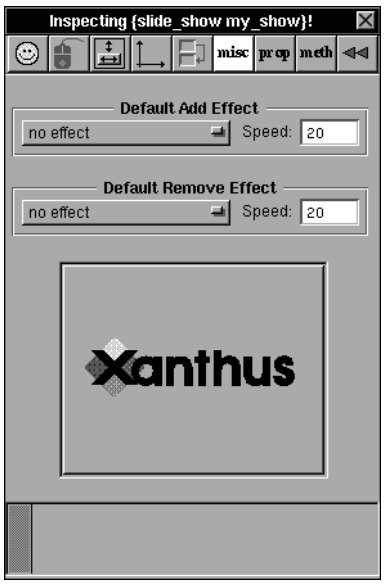
↔ **Select the slide-show and choose *Inspect* from the menu *Inspector.***

To set the visual effects you have to inspect the miscellaneous aspect of the slide-show.

↔ **Click the miscellaneous aspect button.**

Picture 39. The miscellaneous aspect button.

The Inspector should now look like this:



Picture 40. The miscellaneous aspect of a slide-show.

Here you can set the *add effect*, the *remove effect* and the *speed of the effects*. At the bottom of

the Inspector is a small slide-show that shows how the effects will look.

Picture 41. A small slide-show to show the visual effects you have chosen.

↔ **Set an add effect and a remove effect using the pop-up menus in the Inspector.**

Picture 42. A pop-up menu to set an effect.

Note: Do not use visual effects that includes a 'dissolve'. They do not work properly when the slides are transparent like they are in this example. (A known bug in the current version of the Appkit.) For non-transparent slides (as in the mini slide-show in the Inspector) 'dissolve' effects works fine.

↔ **Try the effects by clicking on the small slide-show in the Inspector.**

↔ **Switch to run-mode.**

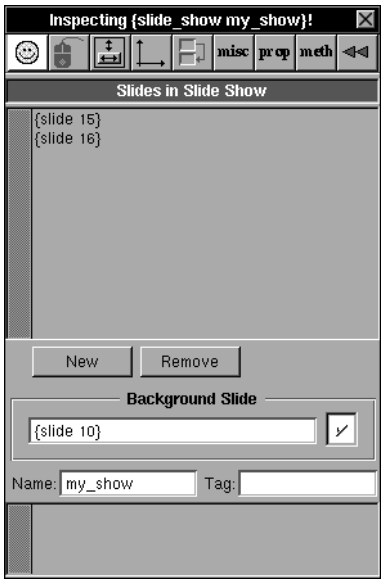
When you click the button *Next animal* the effects you chose will be used by the slide-show.

Now you will edit the background slide of the slide-show.

↔ **Switch back to edit-mode again.**

↔ **Choose the appearance aspect of the slide-show again by clicking on the aspect button to the left (with the little smiling face).**

↔ **Click on the small square with a check-mark at the bottom right of the Inspector.**



Picture 43. The background slide of the slide-show is selected.

This square is called an *object-handle* and it represents the background slide of the slide-show.

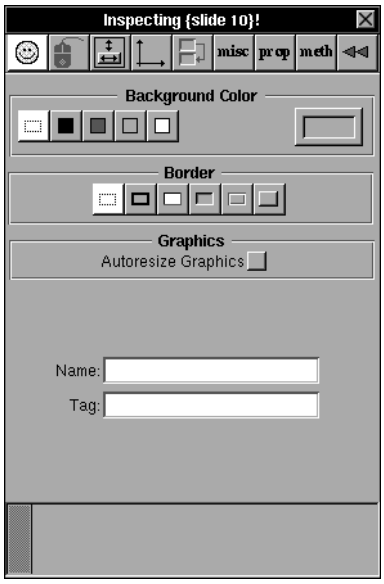
When the object handle is highlighted you will only see the background slide in the slide-show.

None of the slides with animals are visible.

Picture 44. Only the background slide visible.

↔ **Double-click on the object handle for the background slide in the Inspector.**

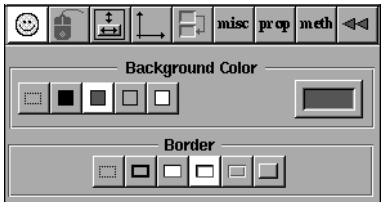
You will then inspect the background slide.



Picture 45. Inspecting the background slide.

As you can see, this Inspector lets you change the color and the border-type of the slide. The default appearance of the background slide in a slide-show is to be light gray and to have no border.

↔ **Change the slide to be dark gray and bezel.**



Picture 46. Changing the color and border type in the Inspector.

The background slide in the slide-show should now look like this:

Picture 47. A dark gray and bezel background slide.

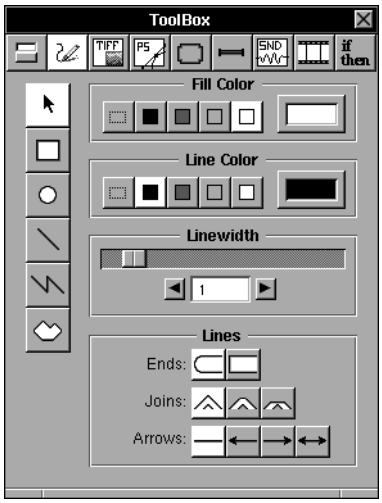
Now you will add some graphics to the background slide of the slide-show.

To draw graphics you need to switch to the ornaments tool.

Picture 48. The tool button for the ornaments tool.

↔ **Click the button for the ornaments tool at the top of the ToolBox window.**

The ToolBox should now look like this:



Picture 49. The ornaments tool.

To the left is a list of drawing tools: *rectangle*, *oval*, *line*, *polyline* and *polygon*. The arrow tool is not a real drawing tool. It represents the normal mode when you can select, drag and resize objects in a window.

↔ **Select the oval tool by clicking the oval button.**

Picture 50. The oval button.

↔ **Put the cursor over the background slide in the slide-show and press the mouse-key.**

While you drag the mouse with the mouse-key down you will see the outline of the oval you are drawing. The cursor will be a cross-hair during the drawing.

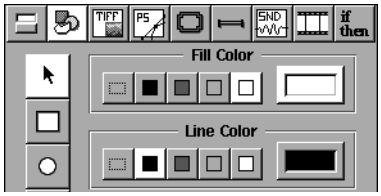
When you release the mouse-key, the oval is drawn. Note that the oval is also the selected

object in the window.

Picture 51. An oval on the current slide.

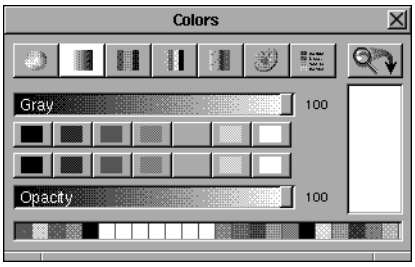
The *fill-color*, the *line-color* and the *line-width* can be set in the ToolBox.

↔ **Click on the border of the color-well for the fill color in the ToolBox.**



Picture 52. The border of a color-well highlights when you click on it.

The Color panel will open.



Picture 53. The Color panel.

↔ **Set the color of the oval:**

Picture 54. Change the color of the oval with the Color panel.

↔ **Make sure the oval is selected and its window is the key window.**

↔ **Choose *Copy* from the menu *Edit*.**

↔ **Choose *Paste* from the same menu.**

You will now have a new oval in the same size as the first one but positioned a little down and

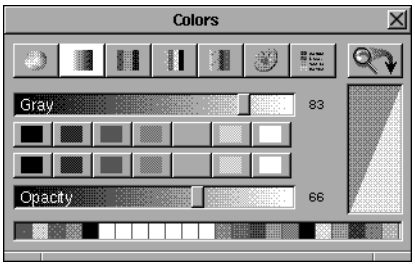
to the right.

Picture 55. Two ovals.

↔ **Make the new oval sink behind the first oval by choosing *Sink* in the menu *Layout*.**

↔ **Change the color of the new oval with the Color panel.**

↔ **Set a partly transparent color by using the Opacity slider in the color Panel.**



Picture 56. Setting a partly transparent color.

Picture 57. Two ovals on the background slide of the slide-show.

↔ **Close the color Panel.**

Finally, you will add a sound to the button. The sound will be played every time the button is clicked.

↔ **Choose the sound tool in the ToolBox.**

Picture 58. The button for the sound tool.

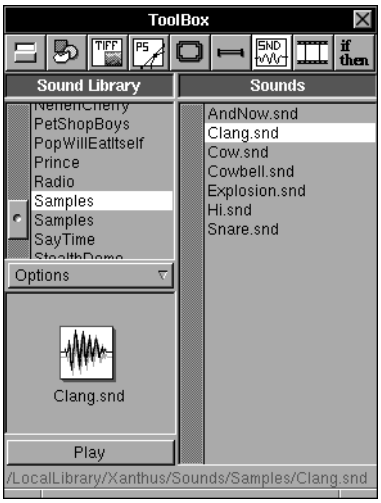
The ToolBox should now look like this:

Picture 59. The sound tool.

↔ **Select the ‘Samples’ sound library.**

↔ **Select the ‘Clang.snd’ sound.**

The ToolBox should now look like this:



Picture 60. The sound 'Clang' selected.

To hear the sound, click the *Play* button below the sound icon or double-click the name of the sound in the list.

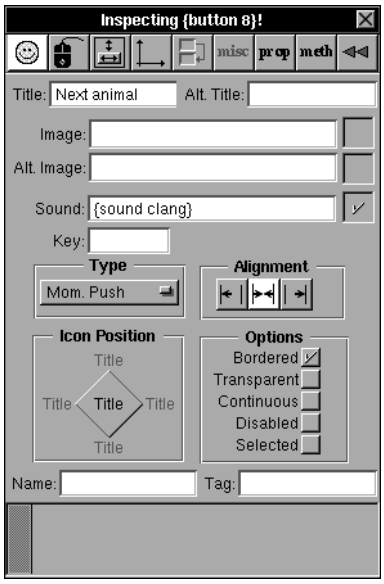
↔ **Click the *Play* button in the ToolBox and you will hear the sound.**

↔ **Pull the sound icon from the lower left of the ToolBox to the button titled *Next animal* in your window.**

The button will get a black border when it accepts the sound.

↔ **Drop the sound on the button.**

↔ **Select the button and choose *Inspect* from the menu *Inspector*.**



Picture 61. Inspecting the button.

In the Inspector you can see that the property *Sound* has the value *{sound clang}*.

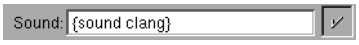
If you switch to run-mode and click the button, the sound will play every time the button is clicked.

↔ **Switch to run-mode.**

↔ **Try clicking the button.**

↔ **Switch back to edit-mode.**

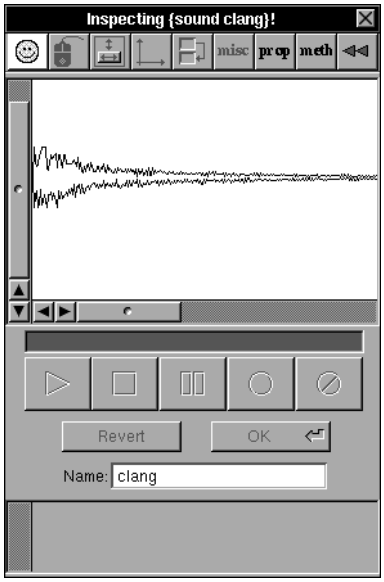
Note the little square with the check-mark to the right of the Sound property in the Inspector. The square is called an *object handle* and the check-mark tells you that the property is set.



Picture 62. An object handle.

↔ **Double-click the object handle.**

This will inspect the object that the object handle represents. In this case you will inspect the sound.



Picture 63. Inspecting a sound.

In this Inspector you can cut and paste in the sound as you like to modify it. Use the sound control buttons to play and record.

Play *Stop* *Pause* *Record* *Erase*

↔ **Click *OK* when you are through editing the sound.**

To get back to the button Inspector quickly you can click the 'inspect previous' button at the

right of all the aspect buttons.

Picture 64. The 'inspect previous' button is at the top.

Clicking this button will always take you back to the object you last inspected.

↔ **Switch to run-mode.**

↔ **Test your edited sound.**

↔ **Choose *Edit* from the menu *Program*.**

↔ **Choose *Save* from the menu *Program* and a Save panel will appear.**

↔ **Give your program a name and click the *OK* button.**

The program is now stored on disk and can be opened by choosing *Open* in the menu *Program*.

Note: If you are running a demo version of CraftMan or if your copy of CraftMan is not registered the 'Save' command will be disabled. Then continue with the next section without saving your program.

Writing CraftScript

As the last exercise in this section, you will write some scripts in the CraftScript language to control the behavior of a movie object. CraftScript is the scripting language in CraftMan that allows you to control your program in great detail. CraftScript is the most fundamental part of CraftMan and a good understanding of CraftScript is essential to fully utilize CraftMan.

↔ **Open the program you created in the previous section by choosing *Open* in the menu *Program*.**

First you will create a new slide in the slide-show that should contain the movie.

↔ **Inspect the slide-show.**

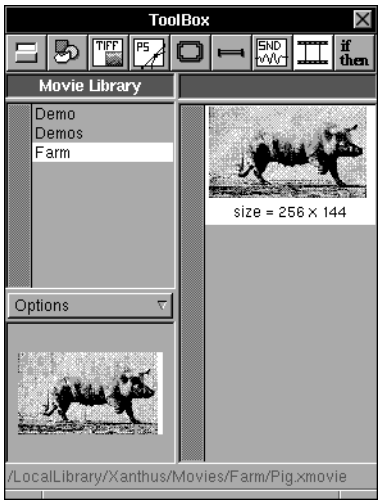
↔ **Choose the appearance aspect.**

↔ **Click the button *New* to get a new slide in the slide-show.**

↔ **Choose the movie tool in the ToolBox.**

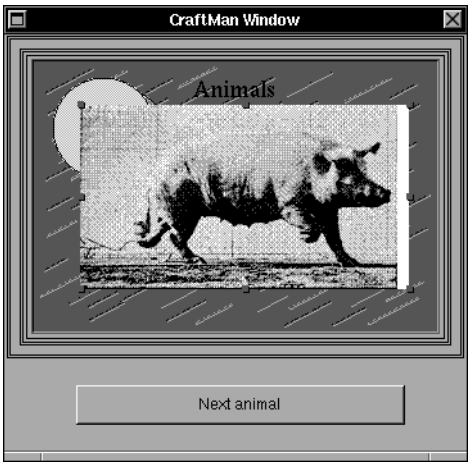
Picture 65. The button for the movie tool.

↔ **Select the movie library named ‘Farm’.**



Picture 66. The movie tool.

↔ **Pull the movie with the pig to the new slide in the slide-show.**



Picture 67. A movie dropped on the slide-show.

CraftMan and CraftScript are object-oriented. This means that a CraftMan program consists of a collection of objects. The objects communicate by sending messages to each other. Some messages are sent from an object to another object, some messages are sent from CraftMan to an object. A message of the last type is called an *event message*.

For instance, when the user presses the mouse-key, CraftMan sends a *mouse_down* message to the object that is below the cursor. When a slide is made the current slide, CraftMan sends a *slide_was_added* message to the slide.

If you want a slide to do something special when it is made the current slide, you simply create a special *slide_was_added* method for the slide. A method is a script written in CraftScript that will execute when a message with the same name as the method is sent to the object. All objects have a number of methods by defaults. You can add as many methods as you like.

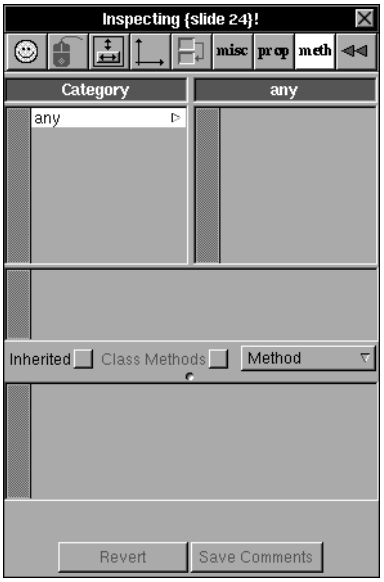
↔ **Inspect the new slide by double-clicking on the slide in the slide-list in the Inspector.**

To create a method for a slide you have to choose the methods aspect of the slide.

Picture 68. The button for the methods aspect.

↔ **Choose the methods aspect.**

The Inspector should now look like this:

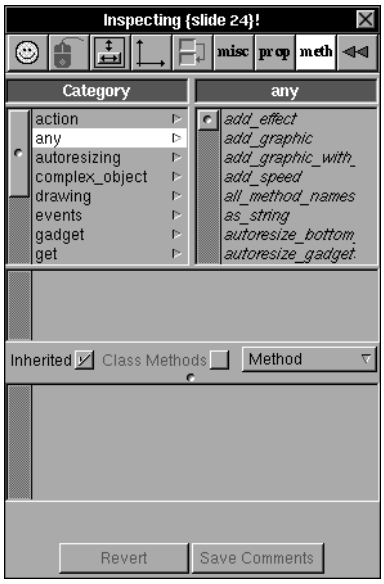


Picture 69. The methods aspect of a slide.

As you can see the Inspector is quite empty. This slide has no methods of its own. If you click the button *Inherited*, you will see all the methods that the slide inherits from its super-classes.

↔ **Click the button *Inherited*.**

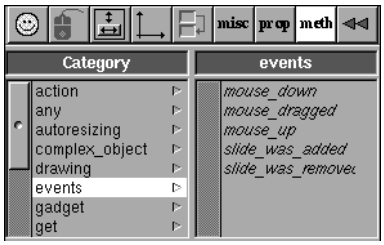
The inherited methods will be printed in italic.



Picture 70. The Inspector shows all the inherited methods.

To the left is a column called *Category*. This is a list of all the different method categories of a slide. Methods are grouped into categories to make them easier to find. Right now the category '*any*' is selected.

↔ **Select the category *'events'*.**



Picture 71. The 'events' category is selected.

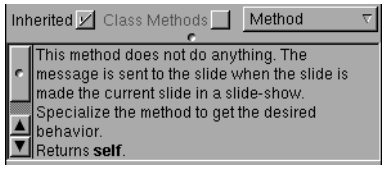
In the events category are listed the methods that responds to event messages from CraftMan. As you can see, the *mouse_down* and *slide_was_added* methods are listed here. By default,

these methods does not do anything. You can now write your own version of a method to make something happen.

You will now write two methods. One *slide_was_added* method and one *slide_was_removed* method. The *slide_was_added* method should start the movie that is on the slide and the *slide_-was_removed* method should stop the movie.

↔ **Select the *slide_was_added* method in the list to the right.**

At the bottom of the Inspector will appear a short description of the method.



Picture 72. A method description.

Note: You can always get a description of a method by selecting it in the Inspector.

↔ **Choose *Specialize* in the pull-down menu *Method* at the right in the Inspector.**

An Editor window will appear on the screen.

Picture 73. An Editor window.

The Editor window contains a script for the method *slide_was_added*. We will now go through each line of the script and explain what they mean.

- The *first* line states that this is a script for a method called *slide_was_added*.
- The *second* line states that the method belongs to the object *{slide 24}*.

Note: *The number 24 might be different on your screen but that is not important.* ***Never change that number in the Editor.***

- The *third* line states that the method should be grouped in the category *'events'*. This line is optional and does not need to be there.
- The *fourth* line states that now starts the actual code of the script.
- The *fifth* line is the only line of code in this script.
- The *sixth* line states that now ends the actual code of the script.

You want the method to start the movie. To do this you have to send the message *start* to the movie.

↔ **Delete the fifth line of the script and replace it with the following line:**

```
{movie pig}:start
```

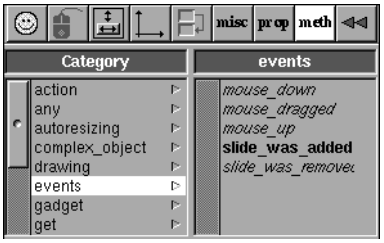
The script should now look like this:

```
define method slide_was_added
   of {slide 24}
   in events
   as
   {movie pig}:start
end define
```

↔ **Save the script by choosing *Save* from the menu *Editor*.**

↔ **Close the Editor window by clicking in the close box in the upper right corner of the window.**

In the Inspector the name of the method *slide_was_added* has changed from italic black to *boldface* and *blue*. This means that the method is defined in the slide and that it is created by you.



Picture 74. The method name in boldface and blue.

If you want to edit the method script, just double-click on the method-name in the list. The Editor window will then appear with the script.

Now do the same thing with the method *slide_was_removed*:

↔ **Select the method *slide_was_removed* in the list.**

↔ **Choose *Specialize* in the pull-down menu *Method* at the right in the Inspector.**

↔ **Enter the following script in the Editor. (Only the fifth line needs to be changed.)**

```
define method slide_was_removed
   of {slide 24}
   in events
   as
   {movie pig}:stop
end define
```

↔ **Save the script by choosing *Save* in the menu *Editor*.**

↔ **Close the Editor window.**

If you now switch to run-mode you will see that the movie is started every time the slide is shown.

↔ **Switch to run-mode.**

↔ **Try the slide-show.**

Note: The first time the movie is started it will be a bit slow. The movie is then loaded into memory from the disk. As soon as it is loaded it will run as usual.

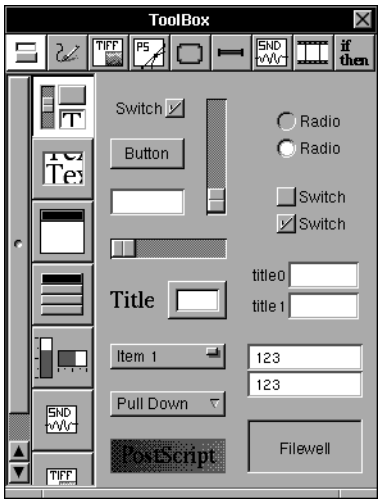
If you want the user of your program to be able to control the speed of the movie (i.e. number of frames per second) you need some sort of control to set the *frames_per_second* property of the movie.

↔ **Switch to edit-mode.**

↔ **Make sure the slide with the movie is the visible slide in the slide-show If not then inspect the slide-show, select the appearance aspect and select that slide in the list of slides.**

↔ **Choose the palette tool in the ToolBox.**

↔ **Choose the gadgets palette (the top palette in the palette-list).**



Picture 75. The gadget palette in the palette tool.

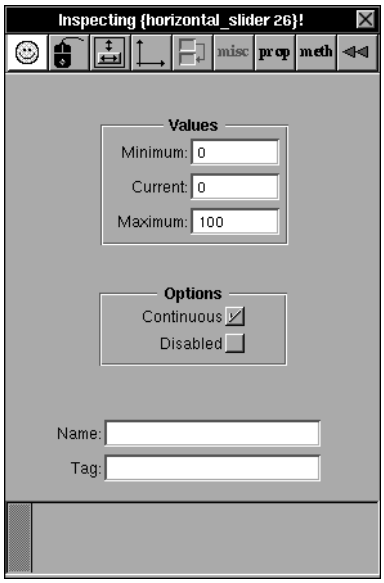
↔ **Pull a slider to your window.**

Picture 76. A slider in the window.

A slider is a control object that can be used to set properties in other objects.

↔ **Make sure the slider is selected and choose *Inspect* in the menu *Inspector*.**

↔ **Select the appearance aspect.**

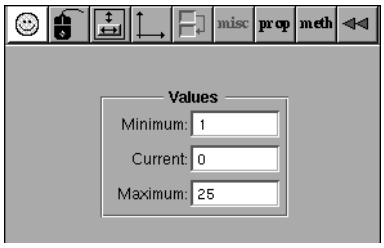


Picture 77. Inspecting a slider.

↔ **Change the Minimum value to 1.**

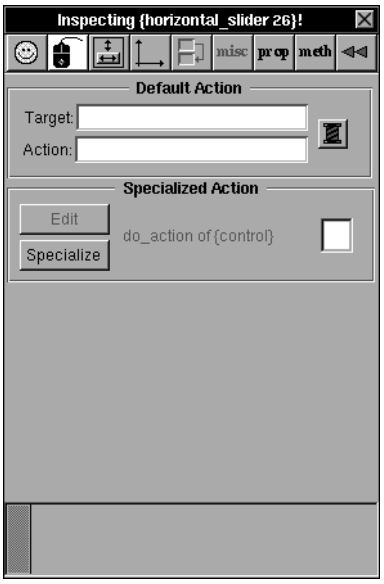
↔ **Change the Maximum value to 25.**

This means that the slider can be set to values from 1 to 25.



Picture 78. Changing the values for the slider.

↔ **Select the control aspect.**



Picture 79. The control aspect of the slider.

This is the fastest way to create your own *do_action* method for the slider.

↔ **Click the *Specialize* button to open an Editor window.**

Note: You could have done this in the methods aspect in the same way as you did with the 'slide_was_added' and 'slide_was_removed' methods. However, when you want to write a new 'do_action' method, this is the fastest way to do it.

The script should look like this:

```
define method do_action
   of {horizontal_slider 26}
   in events
   as
   return super:do_action
end define
```

↔ **Change the script to look like this:**

```
define method do_action
   of {horizontal_slider 26}
   in events
   as
   x := self:integer_value
   {movie pig}:set_frames_per_second(x)
end define
```

First the variable *x* is set to the integer value of the slider. You simply send the *integer_value* message to the slider itself.

Then you send the message *set_frames_per_second* with the value *x* to the movie object.

The script could also be written:

```
define method do_action
   of {horizontal_slider 26}
   in events
   as
   {movie pig}:set_frames_per_second(self:integer_value)
end define
```

↔ **Save the script by choosing *Save* in the menu *Editor*.**

↔ **Close the Editor window.**

↔ **Switch to run-mode.**

↔ **Try the slide-show.**

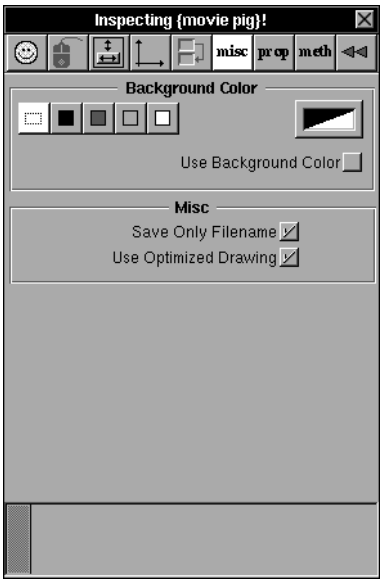
↔ **Change the speed of the movie with the slider.**

As you can see the speed of the movie does not change very much when you drag the slider. This is because the movie is not using *optimized drawing*. A movie that is not using optimized drawing can run *behind other objects* and *through partly transparent objects*. This is a much slower way of drawing the movie. In this case you do not need that functionality so you will now change the movie so that is uses optimized drawing.

↔ **Switch to edit-mode.**

↔ **Make sure the movie is selected and choose *Inspect* in the menu *Inspector*.**

↔ **Click on the button for the miscellaneous aspect in the Inspector.**



Picture 80. The miscellaneous aspect of a movie.

↔ **Set the switch *Use Optimized Drawing*.**

↔ **Switch to run-mode and change the speed of the movie again.**

As you can see the movie runs much faster when it is not using optimized drawing.

Now you have completed the three exercises in this section. Please refer to the book *Users Guide* to read more about how to use slide-shows, images, sounds and movies.

Tutorial Two: Using Timers

This tutorial consists of two parts:

1. In the first section you will create a timer that will switch the slides in a slide-show at regular intervals.
2. In the second section you will improve the timer so that it is reset every time the user switches the slides in the slide-show.

This example program can be found in the *Demo Programs* list in the *Info panel*.

Note: When there is a description of something that you should do, the sentence will be printed in boldface, on its own line and will be preceded by the symbol ↔.

For example if you are supposed to click a button with the title *New*, it will look like this:

↔ **Click the button titled *New*.**

To follow the tutorial make sure you do everything that is marked with a ↔.

What is a Timer?

A timer is an object that receives a *do_action* message at a specific interval. Timers can be used to make things happen automatically after a certain time, for example switching a slide in a slide-show or quitting a program.

You may have any number of timers in your program.

Creating a Timer

↔ **Open the first program you created in the first tutorial by choosing *Open* in the menu *Program*.**

Note: If you cannot find that program or if you never saved it, open the Info Panel and double-click on the Tutorial_1.1 program in the list of demo programs. You can then choose Save As in the menu Program to save a copy of that program.

This program contains one slide-show with two slides and a button that switches the slides in

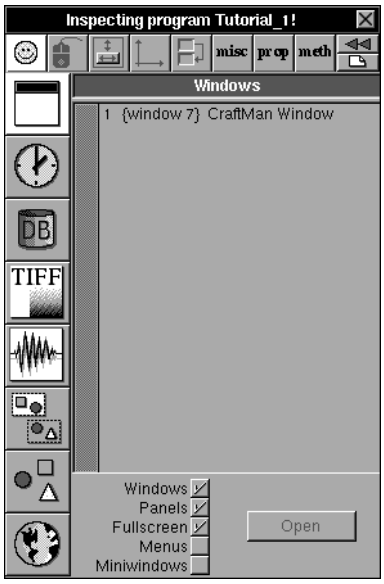
the slide-show. You will now create a timer that switches the slides in the slide-show.

Picture 81. The first program of the first tutorial.

Timers are created in the appearance aspect of the program inspector.

↔ **Switch to edit-mode by choosing *Edit* in the menu *Program*.**

↔ **Inspect the program by choosing *Inspect the_program* in the menu *Inspector*.**

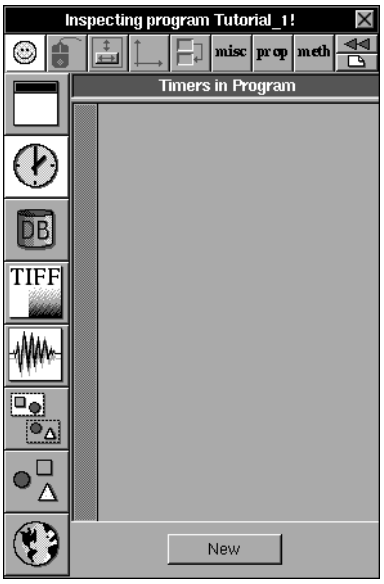


Picture 82. The program appearance inspector.

To the left in the Inspector you will see a number of buttons. To see the list of timers in the program click the timer button.

Picture 83. The timer button.

↔ **Click the timer button.**

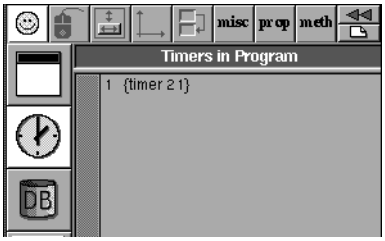


Picture 84. The list of timers in the program.

Right now there are no timers in the program.

↔ **Click the *New* button in the Inspector.**

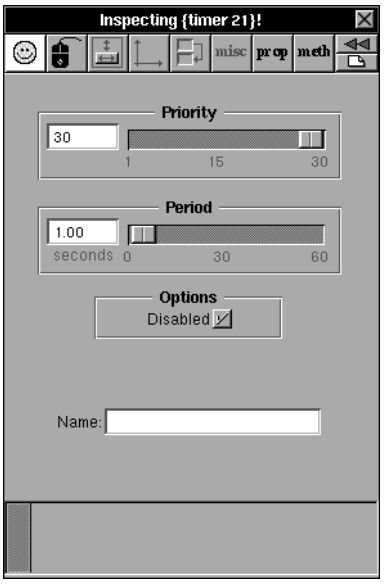
A new timer will be created.



Picture 85. A new timer.

↔ **Double-click on the timer to inspect it.**

You will now inspect the new timer.



Picture 86. Inspecting the appearance aspect of a timer.

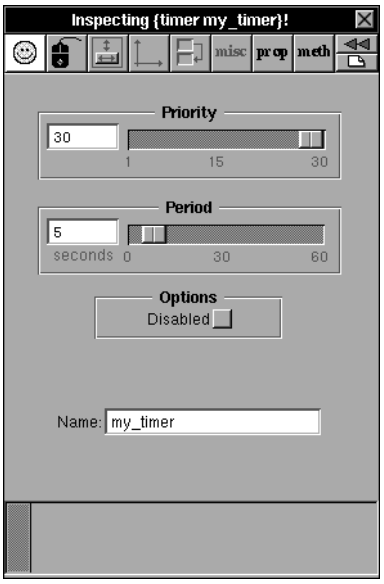
↔ **Name the timer *my_timer*.**

↔ **Click on the *Disabled* switch to enable the timer.**

↔ **Set the *Period* to 5 seconds.**

This means that the timer is enabled and that it will receive a *do_action* message every five sec-

onds.

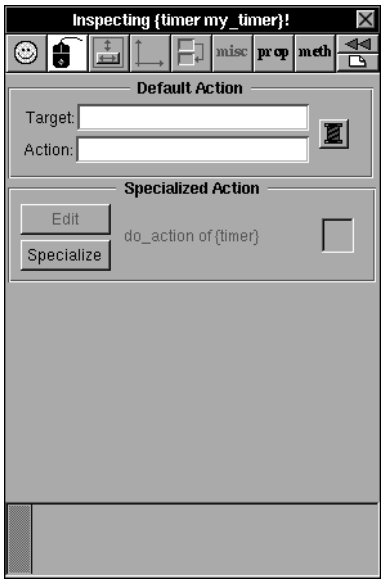


Picture 87. The edited timer inspector.

You will now specify what the timer should do when it receives a *do_action* message. This is the same message as control objects like buttons receive when the user clicks on them.

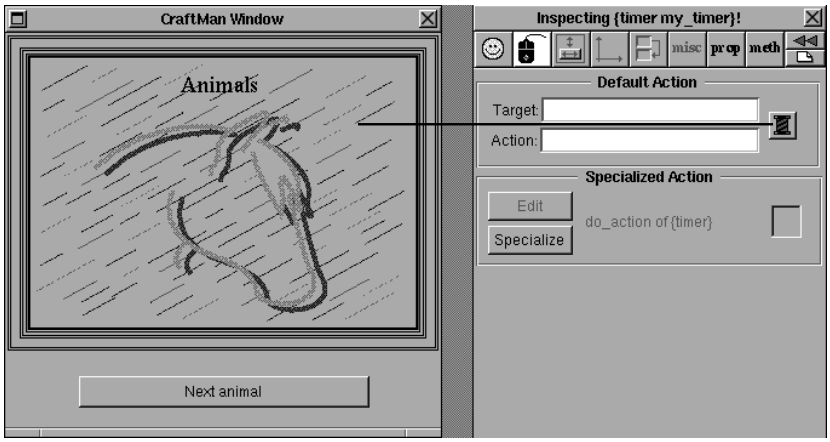
To specify what the timer should do when it receives a *do_action* message you do like you have done with buttons, setting the *action* and *target* properties in the control aspect of the Inspector.

↔ **Select the control aspect of the Inspector.**



Picture 88. The control aspect of the timer inspector.

Drag an action-target connection from the cable-drum in the Inspector to the slide-show in the CraftMan window.



Picture 89. Dragging an action-target connection to the slide-show.

↔ **Select the *next_slide* message in the Action message panel.**

Now you have a timer that will send the *next_slide* message to the slide-show every five seconds.

↔ **Switch to run-mode.**

Note how the slides switches every five seconds.

Resetting a Timer

Often you want to have a timer that does something if the user waits to long. In the slide-show example above you would like the slide to switch if it was more than five seconds since the user switched slide with the button

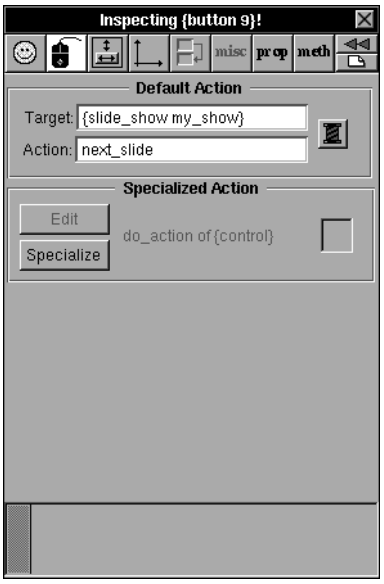
The problem is that the timer does not know anything about how long time that has passed since the button was clicked. To solve this we have to *reset the timer* each time the button is clicked.

↔ **Switch to edit-mode.**

↔ **Select the button.**

↔ **Choose *Inspect* in the menu *Inspector.***

You should now inspect the control aspect of the button.



Picture 90. Inspecting the control aspect of the button.

Instead of just sending the *next_slide* message to the slide-show the button should have its own *do_action* method that first resets the timer and then send *next_slide* to the slide-show.

↔ **Click the *Specialize* button in the *Inspector*.**

You will get an Editor window where you can create a special *do_action* method for the button.

Picture 91. An Editor window for the do_action method.

↔ **Enter the following script in the Editor:**

```
define method do_action
   of {button 7}
   in events
   as
   {timer my_timer}:set_is_enabled(false)
   {timer my_timer}:set_is_enabled(true)
   return super:do_action
end define
```

First you disable the timer, then you enable the timer again. This will make the timer start from zero again.

Finally you pass the *do_action* message to something called *super*. *super* is the object above the current object in the class hierarchy. In this case *super* is equal to the *button* class.

This means that the button will do what it usually does when it receives a *do_action* message, i.e. send the *action* message to the *target* object. In this case it means sending *next_slide* to the slide-show.

↔ **Save the method by choosing *Save* in the menu *Editor*.**

↔ **Close the Editor window.**

↔ **Switch to run-mode.**

Now the slide-show will automatically switch slide five seconds after the button is clicked.

Tutorial Three: A New Class

This tutorial consists of three parts:

1. In the first section you will create your own button-class called *memory button*.
2. In the second section you will paste the new class into a user-defined palette.
3. In the third section you will export your program to a text-file.

This example program can be found in the *Demo Programs* list in the *Info panel*.

Note: When there is a description of something that you should do, the sentence will be printed in boldface, on its own line and will be preceded by the symbol ↔.

For example if you are supposed to click a button with the title *New*, it will look like this:

↔ **Click the button titled *New*.**

To follow the tutorial make sure you do everything that is marked with a ↔.

Classes and Objects

CraftMan is object-oriented and there are many system classes from which you can create objects, for example *button*, *window*, *image* and *collection*.

In addition to the system classes, you can create your own new classes in CraftMan. If you for example want several buttons in your program to understand the same methods, it might be a good idea to create a new button-class that implements those methods. Then there will only be one copy of the CraftScript code which is easier to change and support.

Note: A new class only exists locally in a program, but you may copy and paste classes between programs.

The Class 'memory_button'

We will now create a new button class called *memory_button*. Buttons of this class will remember all mouse-clicks. If you send the message *print_log* to a memory button it will create a text-file with one line per mouse-click.

The class *memory_button* will have three new methods:

1. The method *do_action* will remember the mouse-click and then let the button handle

the mouse-click as usual.

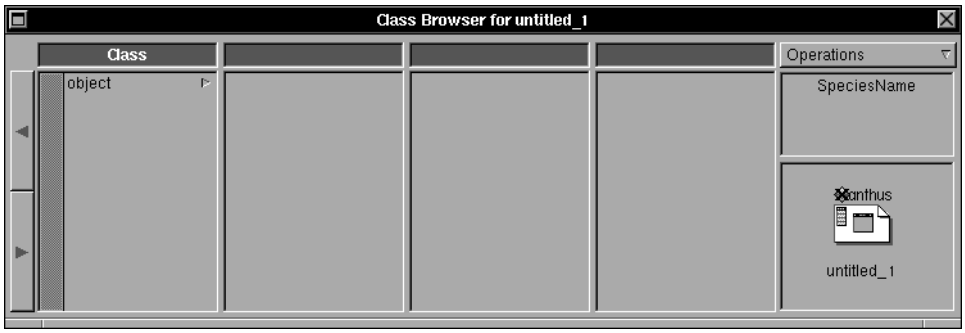
Note: *CraftMan sends the do_action message to a button when you click on it.*

2. The method *print_log* will print all the mouse-clicks to a text file.

3. The method *init* will make the button forget all mouse-clicks.

↔ **Create a new CraftMan program by choosing *New* in the menu *Program*.**

↔ **Open the Class Browser by choosing *Class Browser* in the menu *Tools*.**



Picture 92. The Class Browser.

↔ **Select the class *object* in the column named *Class*.**

The two sub-classes of *object* are *basic_object* and *complex_object*.

Basic objects are classes like *number*, *string* and *collection*. They cannot be sub-classed.

Complex objects are all other classes and they can be sub-classed.

↔ **Select the class *complex_object* in the column named *object*.**

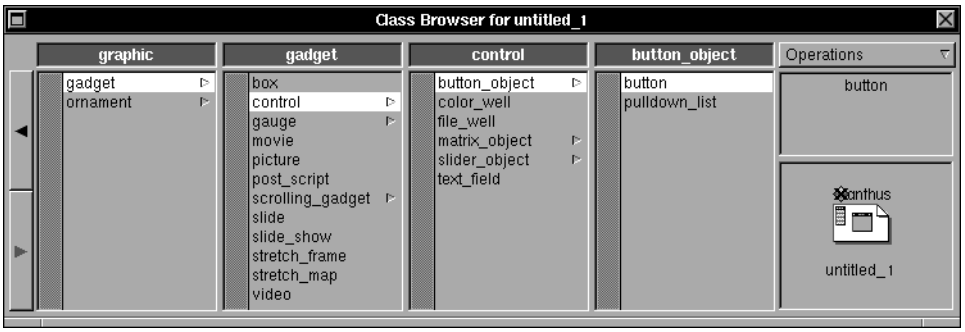
↔ **Select the class *graphic* in the column named *complex_object*.**

↔ **Select the class *gadget* in the column named *graphic*.**

↔ **Select the class *control* in the column named *gadget*.**

↔ **Select the class *button_object* in the column named *control*.**

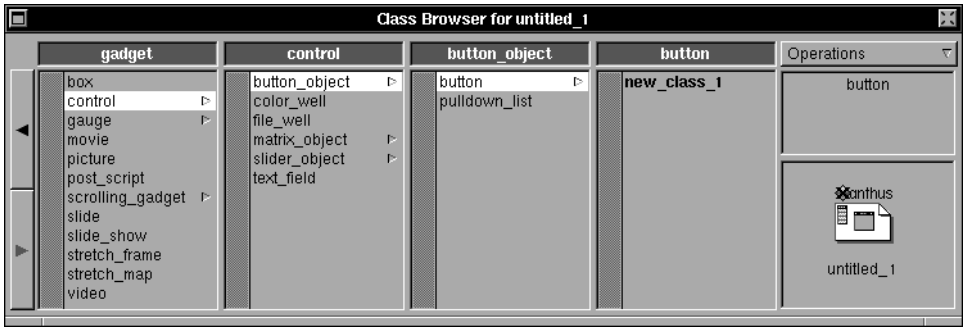
↔ **Select the class *button* in the column named *button_object*.**



Picture 93. The class button is selected.

In this way you can browse through the class hierarchy.

To create a sub-class to the class *button* choose *Define subclass* in the pull-down menu *Operations* that is located to the right in the Class Browser.



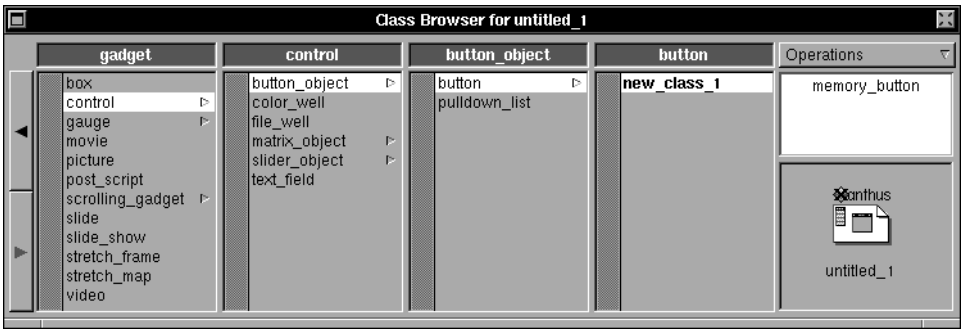
Picture 94. A new class.

New classes will be printed in blue (or in boldface on a monochrome system).

↔ **Select the new class in the column named *button*.**

You can now change the name of the new class in the field to the right.

↔ **Name the new class *memory_button*.**

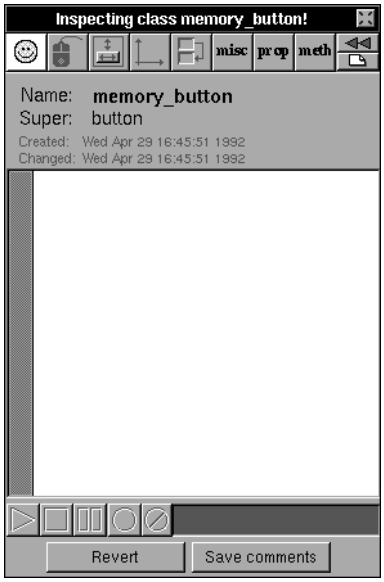


Picture 95. The new class is renamed.

↔ **Double-click on the new class in the column *button*.**

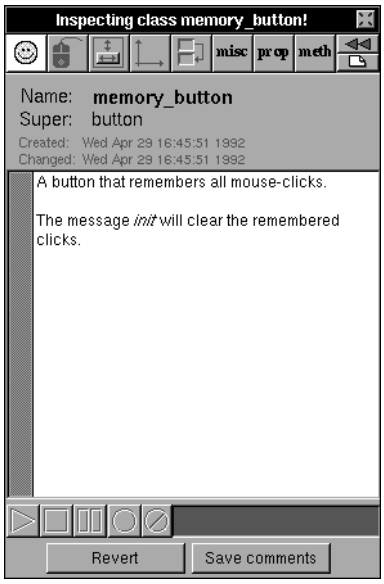
The inspector will then inspect the new class.

↔ **Choose the appearance aspect of the inspector.**



Picture 96. The appearance aspect of a class.

↔ **Type comment that describes the features of the class.**



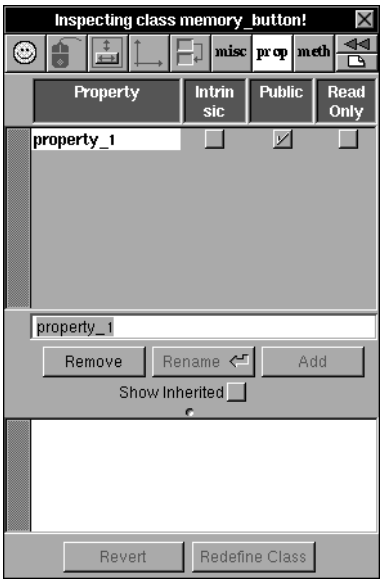
Picture 97. A comment to a class.

↔ **Click the *Save comment* button in the Inspector.**

This comment will be included when you export the program to a text-file.

Now you will define a property for the class *memory_button*. The property will be called *click_log* and it will be a collection of mouse-clicks. Each time the button is clicked information about the mouse-click will be added to the collection.

↔ **Select the properties aspect of the class.**



Picture 98. The properties aspect of a class.

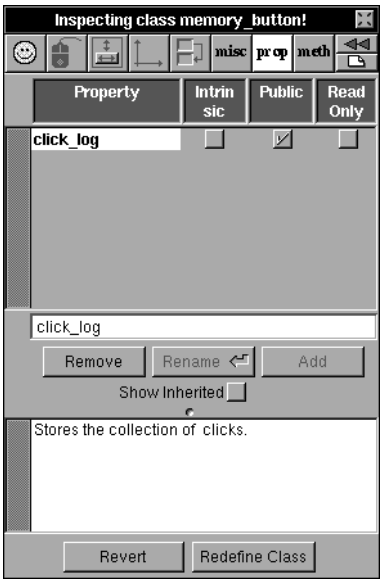
There is one default property called *property_1*.

↔ **Select the property *property_1*.**

↔ **Rename the property *property_1* to *click_log*.**

↔ **Click the *Rename* button.**

↔ **Type a comment that describes the property.**



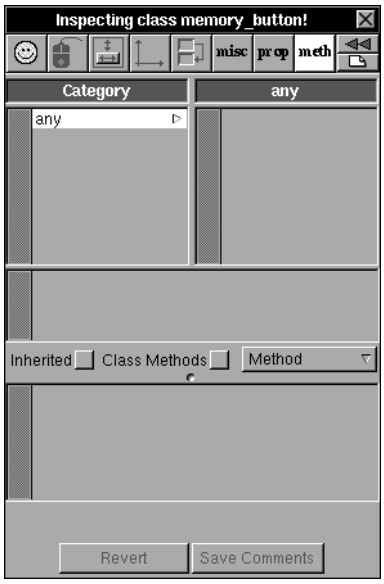
Picture 99. A property with a comment

↔ **Click the *Redefine Class* button.**

Now you will override the default behavior of a button. Before the button does whatever it is programmed to do a memory button should put the information about the mouse-click into its collection of mouse-clicks called *click_log*.

When a button is clicked CraftMan sends a *do_action* message to the button. You will now create a special *do_action* for *memory_button* that first logs the mouse-click and then passes the *do_action* message to the superclass *button*.

↔ **Select the methods aspect of the class.**

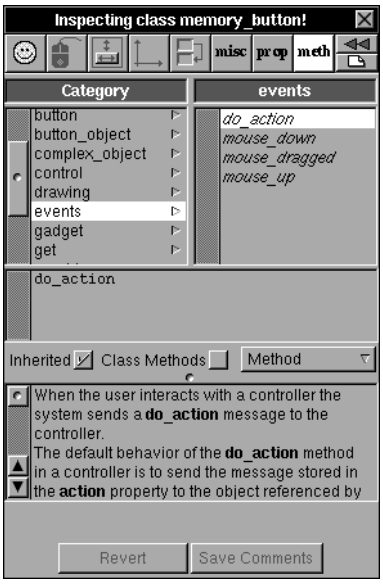


Picture 100. The methods aspect of a class.

↔ **Click the *Inherited* button.**

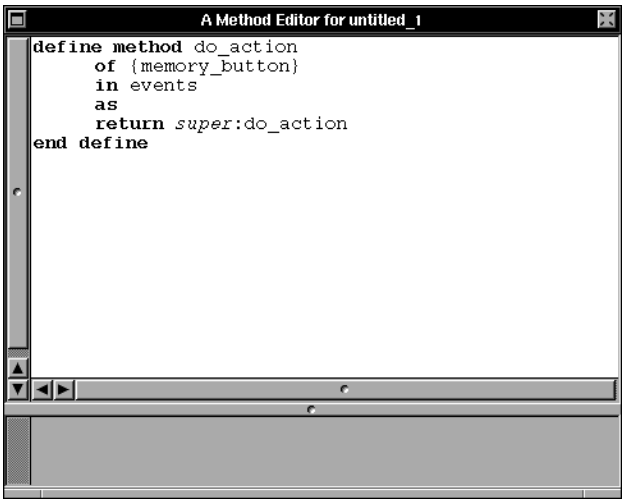
↔ **Select the *events* category.**

↔ **Select the *do_action* method.**



Picture 101. The do_action method selected.

↔ **Choose *Specialize* in the menu *Method* in the Inspector.**



Picture 102. An Editor window.

↔ **Enter the following CraftScript into the Editor window:**

```
define method do_action
   of {memory_button}
   in events
   as
   s := the_program:time
   s:add_last(\nl)
   click_log:add_last(s)
   return super:do_action
end define
```

The method *do_action* does four things:

1. Gets a string with the current time from the program.
2. Adds the *newline* character at the end of the string.
3. Adds the string at the end of the *click_log* collection.
4. Passes the *do_action* message to the super-class (*button* in this case).

Note: The last thing is fundamental. If you omit that line then the button will log mouse-clicks but it will not work as a button. The last line makes the button send its action

message to the target.

↔ **Close the Editor window.**

↔ **Choose *Add in Action* in the menu *Method* in the Inspector.**

↔ **Enter the following CraftScript into the Editor window:**

```
define method print_log
   of {memory_button}
   in action
   as
   name := self:as_string
   name:remove_first
   name:remove_last
   name:remove_all(\space)
   s := click_log:as_string
   s:write_to_file(name)
   f := "open " + name
   the_program:send_to_unix(f)
   return self
end define
```

The method *print_log* does four things:

1. Creates a file-name without curly brackets and spaces from the name of the button.
2. Creates a string from the collection *click_log*.
3. Writes the string to a file with the file-name.
4. Opens the file in Edit.

↔ **Close the Editor window.**

↔ **Choose *Add in Action* in the menu *Method* in the Inspector.**

↔ **Enter the following CraftScript into the Editor window:**

```
define method init
   of {memory_button}
   in action
   as
   click_log := []
```

```
   return self
end define
```

The method *init* does only one thing. It assigns the property *click_log* to the empty collection.

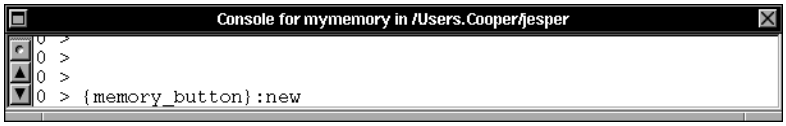
Now you have a new class called *memory_button* that remembers every time it is clicked upon. The class *memory_button* has one property called *click_log* that stores a collection of mouse-clicks.

The class *memory_button* has three methods to manage the *click_log* collection and to write it to a text-file.

Now you want to create an object of the *memory_button* class. Since you cannot find a memory button in the ToolBox you will have to do it by typing CraftScript in the Console window.

↔ **Open the Console window by choosing *Console* in the menu *Tools*.**

↔ **Type *{memory_button}:new* and press return.**



Picture 103. Typing an expression in the Console window

When you send the message *new* to the class *memory_button* it will return a new object of the class. You will see the return value in the Console.

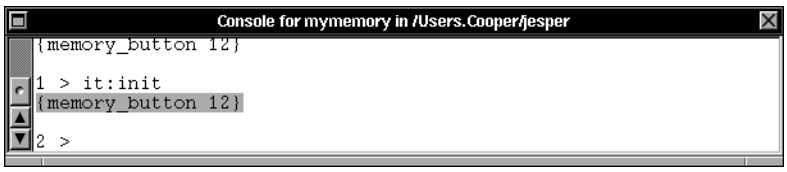
Note: When you execute CraftScript in the Console the program enters run-mode.

The return value of the latest evaluated expression is always stored in a global variable called *it*. So if you want to initialize the new *memory_button* you send the message *init* to *it*.

↔ **Type *it:init* in the Console.**

Now you want to put the new button in the CraftMan window.

↔ **Select the return value in the Console window.**



Picture 104. Selecting the return value in the Console.

↔ **Choose *Copy As Object* in the menu *Copy As*.**

↔ **Switch to edit-mode.**

↔ **Click in the CraftMan window and choose *Paste* in the menu *Edit*.**

↔ **Set the title to *Memory*.**

Picture 105. A memory button in the CraftMan window.

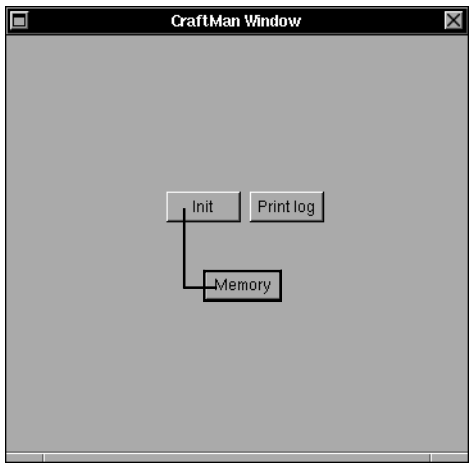
↔ **Drag a button from the ToolBox.**

↔ **Set the title to *Init*.**

↔ **Select the button and choose *Duplicate* in the menu *Edit*.**

↔ **Set the title to *Print log*.**

↔ **Control drag from the *Init* button to the *Memory* button.**



Picture 106. Control dragging from the 'Init' button.

↔ **Select the *init* message in the Action message panel.**

Picture 107. The Action message panel.

↔ **Control drag from the *Print log* button to the *Memory* button.**

↔ **Select the *print_log* message in the Action message panel.**

Now you should have three buttons in your program, one memory button, one button that sends the *init* message to the memory button and one button that sends the *print_log* message to the memory button.

It is now time to test the program.

↔ **Switch to run-mode.**

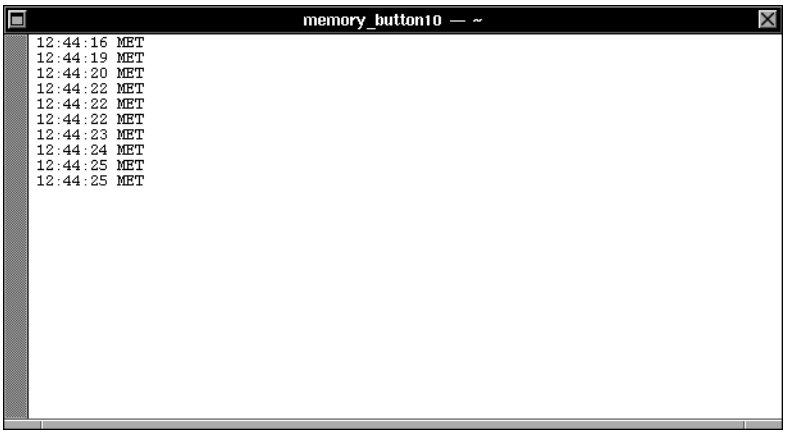
↔ **Click the *Init* button.**

This will reset the memory button so that it forgets all previous mouse-clicks.

↔ **Click the *Memory* button a couple of times.**

The button will remember each click.

↔ **Click the *Print log* button.**



Picture 108. A text-file with the log of mouse-clicks.

The *click_log* will be printed to a text-file and it will be opened in the Edit application.

In the text-file there will be one line for each mouse-click. Each line contains the time when the mouse-click occurred.

Using a User-defined Palette

If you want to put several memory buttons in your program it is convenient to put a memory button in a palette in the ToolBox. Then it is as simple as dragging an ordinary button into your program.

↔ **Switch to edit-mode.**

↔ **Create a new user-defined palette by choosing *New* in the menu *Palette*.**

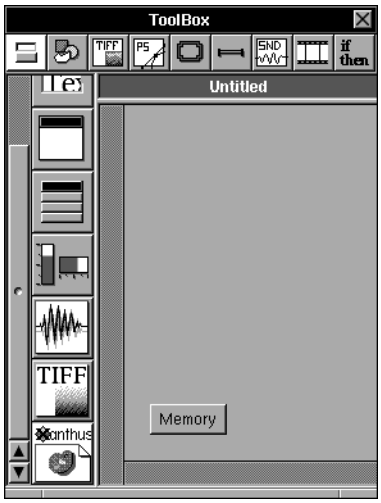
The new palette will appear after all the system palettes.

↔ **Select the memory button by clicking on it.**

↔ **Choose *Copy* in the menu *Edit*.**

↔ **Make sure the ToolBox is the key window by clicking in it.**

↔ **Choose *Paste* in the menu *Edit*.**



Picture 109. A memory button in a user-defined palette.

You can now drag memory buttons from the user-defined palette into your program.

↔ **Save the new palette by choosing *Save* in the menu *Palette*.**

If you would like to put memory buttons in another program in the future you open the palette by choosing *Open* in the menu *Palette*.

Export a Program to a Text-file

To get an overview over a program it is sometimes useful to get a report on paper that describes the contents of the program.

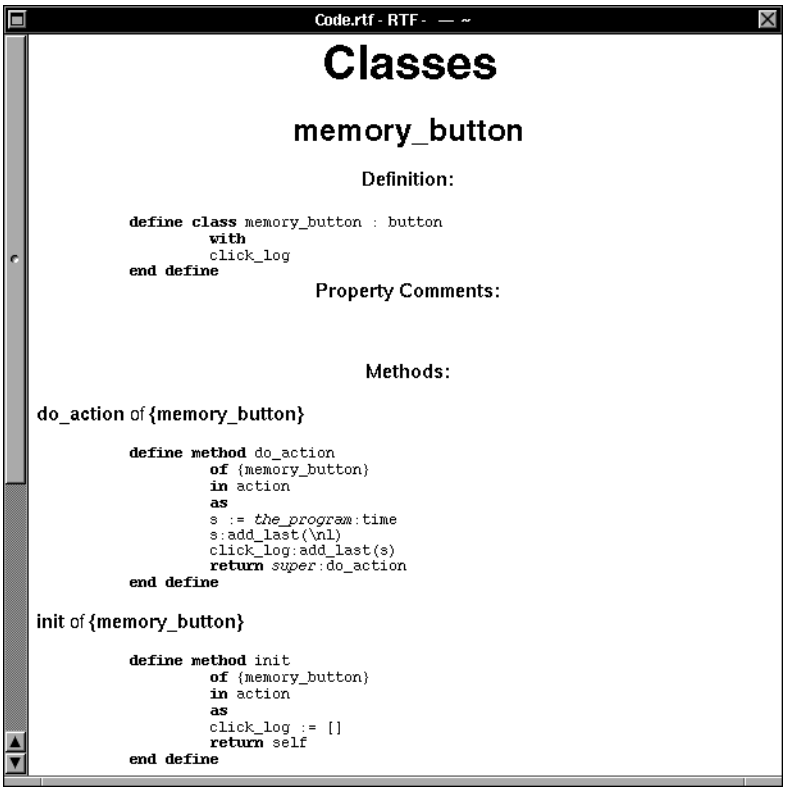
In CraftMan there is an Export function that allows you to export selected parts of a program to a text-file that can be printed.

↔ **Choose *Export* in the menu *Print*.**

Picture 110. The Export panel.

↔ **Click the *Export* button in the panel.**

A text-file named *Code.rtf* will be created and opened in Edit.



Picture 111. The Code.rtf file.

In the Export panel you can select which parts of the program you want to export and to which file-name.

Tutorial Four: Using the Database

This tutorial consists of three parts:

1. In the first section you will build a small phone book.
2. In the second section you will implement a find function for the phone book.
3. In the third section you will export and import data to the phone book.

This example program can be found in the *Demo Programs* list in the *Info panel*.

Note: When there is a description of something that you should do, the sentence will be printed in boldface, on its own line and will be preceded by the symbol ↔.

For example if you are supposed to click a button with the title *New*, it will look like this:

↔ **Click the button titled *New*.**

To follow the tutorial make sure you do everything that is marked with a ↔.

The Database Object

Database objects are designed to store large amounts of data. A database may be sorted and searched and it can import/export data from/to text files.

A database is an ordered collection of *records*. A record is a number of values stored in *fields*:

	Field 1	**Field 2**
Record 1:	**Value**	**Value**
Record 2:	**Value**	**Value**
Record 3:	**Value**	**Value**
. . .		
. . .		
Record n:	**Value**	**Value**

Every field has a type that determines the type of data that can be stored in the field. Valid types are *string*, *integer*, *float*, *boolean*, *color* and *text*.

There is always a *current record* in the database. Usually the values of the fields in the current

record are shown in text-fields in your program. From the current record you can reach the next record and the previous record. You can also go to the first record and the last record in the database.

You cannot refer to a record directly by index in a database.

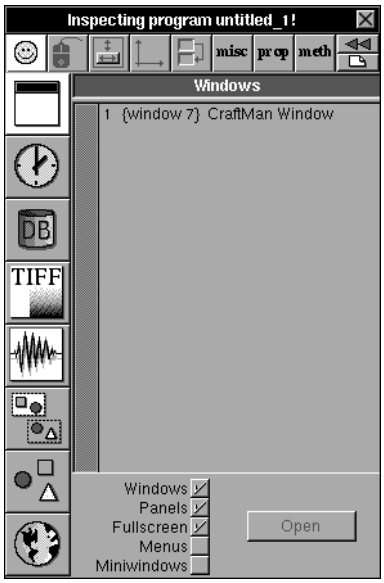
The fields in the database may be connected to controller objects in your program. Each field may have one output controller and one input controller connected to itself. The output controller continuously shows the value in the field of the current record. The input controller may change the value in the field of the current record.

A Phone Book

In this tutorial you will create a small telephone book where you can store name and telephone numbers.

↔ **Create a new CraftMan program by choosing *New* in the menu *Program*.**

↔ **Inspect the program by choosing *Inspect the_program* in the menu *Inspector*.**



Picture 112. The program appearance inspector.

To the left in the Inspector you will see a number of buttons. To see the list of databases in the

program click the database button.

Picture 113. The database button.

↔ **Click the database button.**

Right now there are no databases in the program.

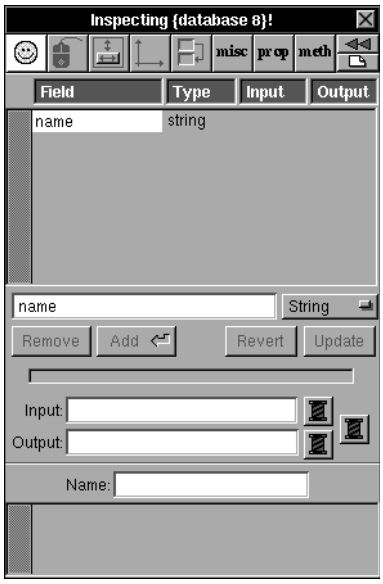
↔ **Click the *New* button in the Inspector.**

A new database will be created.

Picture 114. A new database.

↔ **Double-click on the database to inspect it.**

You will now inspect the new database.



Picture 115. Inspecting the appearance aspect of a database.

↔ **Name the database *phone* and press return.**

Picture 116. Renaming the database.

You can see that there is one field *name* in the database by default.

↔ **Select the field *name* and type *phone* in the text-field.**

↔ **Click the *Add* button in the Inspector.**

You now have two fields in the database, *name* and *phone*.

↔ **To commit the new field to the database click the *Update* button in the Inspector.**

↔ **Now drag two text-fields and four buttons from the ToolBox and place them in the CraftMan window like this:**

Picture 117. Two text-fields and four buttons in the CraftMan window.

To set the font of the text-fields and buttons just select them and open the Font panel (Choose *Font panel* in the sub-menu *Font* in the menu *Layout*).

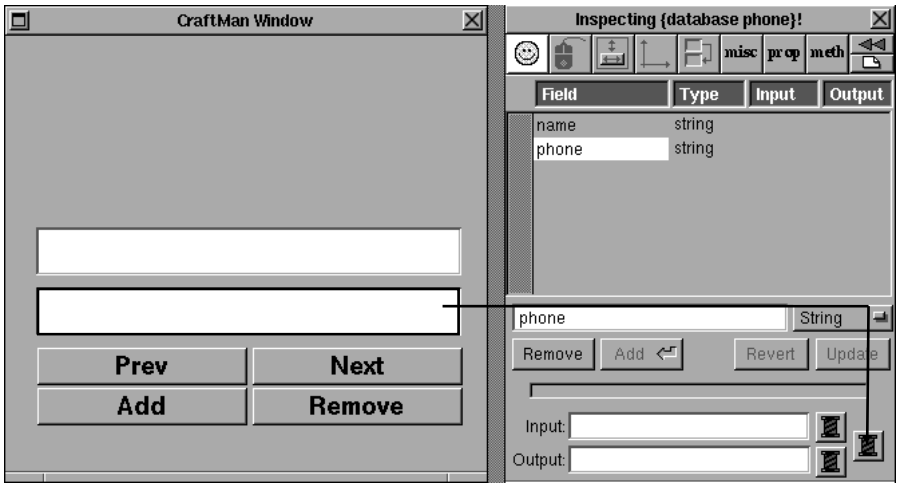
Now you will connect the fields in the database to the text-fields in the CraftMan window.

↔ **Make sure you are inspecting the appearance aspect of the database.**

↔ **Select the field *phone*.**

↔ **Drag a *field-controller connection* from the cable-drum to the right in the Inspec-**

tor to the text-field at the bottom in the CraftMan window.



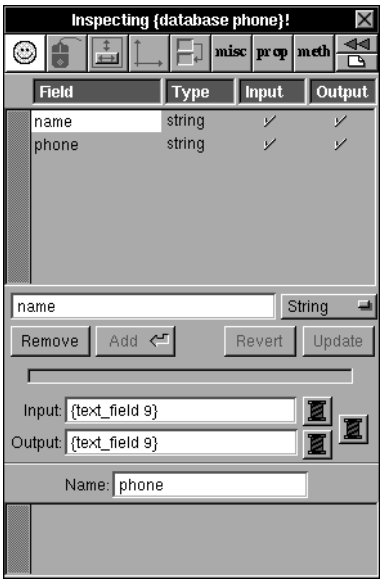
Picture 118. Dragging a field-controller connection from a field to a controller.

↔ **Select the field *name*.**

↔ **Drag a *field-controller connection* from the cable-drum to the right in the Inspector to the text-field at the top in the CraftMan window.**

↔ **To commit the new field-controller connections to the database click the *Update* button in the Inspector.**

The Inspector should now look like this:

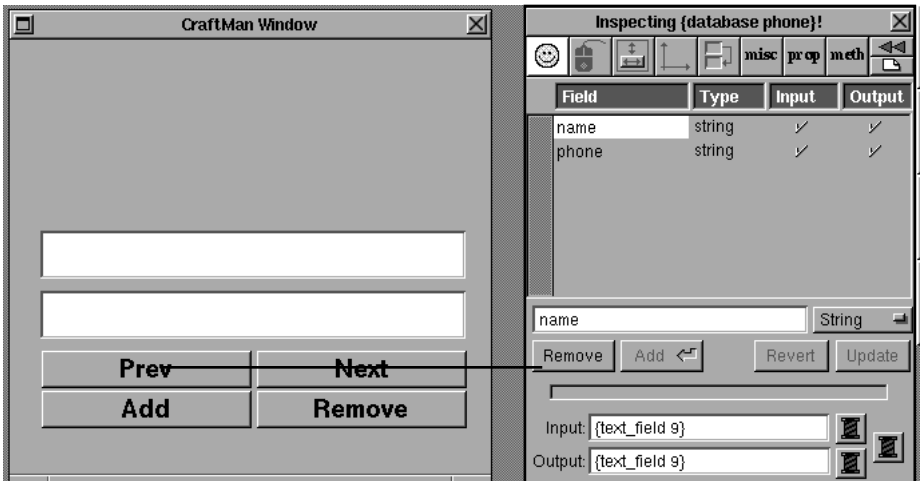


Picture 119. The database inspector.

Now you will make the buttons control the database.

↔ **Press the *Control-key* and drag an action-target connection from the Button *Prev* in the CraftMan window to the Inspector.**

↔ **When the Inspector gets a black border you release the mouse-key.**



Picture 120. Control-dragging to the Inspector.

Note: You can always drag connections to the Inspector window. It is equivalent to dragging a connection to the object that is inspected. In the case of the database this is the only way to drag a connection since the database does not have an appearance of itself.

↔ **Select the *goto_previous_record_and_display* message in the Action message panel.**

↔ **Drag action-target connections from the other three buttons but let the button *Next* send the *goto_next_record_and_display* message, the button *Add* send the *load_current_record_and_display* message and the button *Remove* send the *remove_current_record* message to the database.**

Now you have a database that will show its current record in the text-fields in the CraftMan window and four buttons that let you add a new record, remove the current record and move forward and backward in the database.

↔ **Switch to run-mode.**

↔ **Enter a name and a phone-number.**

↔ **Click the *Add* button.**

↔ **Enter a new name and phone-number.**

↔ **Click the *Add* button.**

The database now contains two records and you can browse through them with the *Next* and *Prev* buttons.

The *Remove* button will remove the current record but it will not display the new current record. If you want that to happen you have to write your own *do_action* method for the *Remove* button that send two messages to the database:

```
define method do_action
   of {button 14}
   in events
   as
   {database phone}:remove_current_record
   {database phone}:display_current_record
end define
```

A Find Function

Usually you want to be able to search your phone book for names or fragments of phone-numbers to find what you are looking for.

You will now implement a *find function* for your phone book.

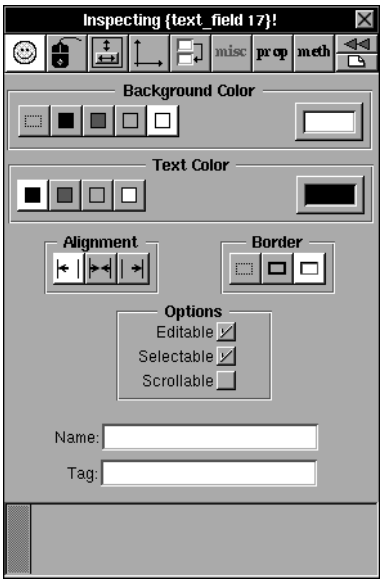
↔ **Drag one text-field and two buttons from the ToolBox and place them in the**

CraftMan window like this:

Picture 121. A text-field and two buttons.

↔ **Make sure the new text-field is selected.**

↔ **Choose *Inspect* from the menu *Inspector*.**



Picture 122. Inspecting the text-field.

↔ **Rename the text-field to *find_string* and press return.**

Picture 123. Renaming the text-field.

↔ **Select the new text-field and the two buttons.**

↔ **Choose *Group In Box* in the menu *Layout*.**

↔ **Double-click on the title of the box and change it.**

Picture 124. The find box.

↔ **Inspect the *Find first* button.**

↔ **Select the control aspect.**

↔ **Click the *Specialize* button to create a new *do_action* method for the button.**

↔ **Enter the following CraftScript into the Editor window:**

```
define method do_action
   of {button 16}
   in events
   as
   s := {text_field find_string}:string_value
   d := {database phone}
   d:goto_first_record
   if d:search(d:fields, s, #within, true, false, false, false)
<> false then
      d:display_current_record
   else
      {sound}:beep
   end if
   return self
```

```
end define
```

This method will start to search for the string from the first record in the database. If it finds a record with the string then that record will be displayed. Otherwise it will beep.

↔ **Save the method by choosing *Save* in the menu *Editor.***

↔ **Close the Editor window.**

↔ **Inspect the *Find next* button.**

↔ **Select the control aspect.**

↔ **Click the *Specialize* button to create a new *do_action* method for the button.**

↔ **Enter the following CraftScript into the Editor window:**

```
define method do_action
   of {button 17}
   in events
   as
   s := {text_field find_string}:string_value
   d := {database phone}
   d:goto_next_record
   if d:search(d:fields, s, #within, true, false, false, false)
<> false then
      d:display_current_record
   else
      d:goto_previous_record_and_display
      {sound}:beep
   end if
   return self
end define
```

This method will start to search for the string from the record after the current record in the database. If it finds a record with the string then that record will be displayed. Otherwise it will go back to the original record and beep.

Note: Search is always done from the current record so if you do not start the search from the record after the current record you would always find the current record.

↔ **Save the method by choosing *Save* in the menu *Editor.***

↔ **Close the Editor window.**

↔ **Switch to run-mode and try the find function.**

Exporting/Importing Data

The database object can export and import data from text-files. To do that you just send the *export* or *import* message to the database.

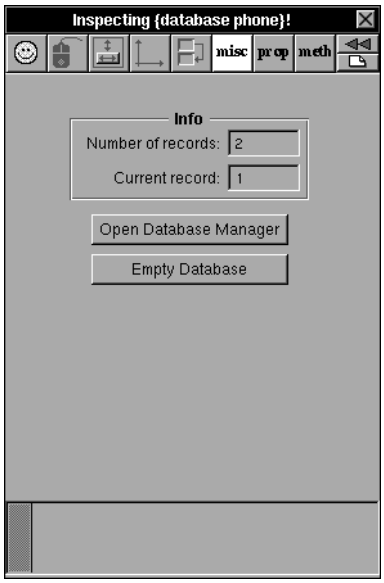
To make it easier to work with the database there is a Database Manager window that can control the database in all the ways you can by sending CraftScript messages.

You can open the Database Manager in two ways:

1. Send the *open_manager* message to the database.
2. Click the *Open Manager* button in the miscellaneous aspect of the database inspector.

↔ **Switch to edit-mode.**

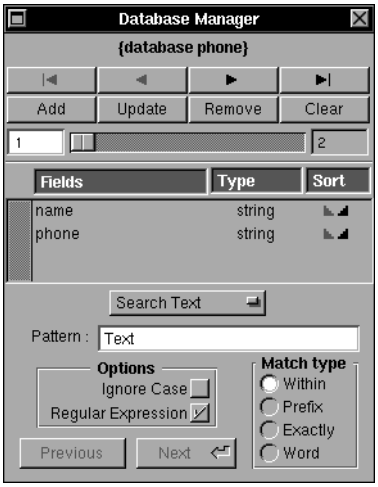
↔ **Inspect the miscellaneous aspect of the database.**



Picture 125. The miscellaneous aspect of the database.

↔ **Click the *Open Manager* button.**

The program will switch to run-mode and the Database Manager window will appear.



Picture 126. The Database Manager window.

As you can see the Database Manager has all the buttons necessary for browsing the database.

In the pop-up menu you can choose between *Search Text*, *Find Numbers*, *Compute Info*, *Sort*, *Export* and *Import*.

↔ **Choose *Export* in the pop-up menu.**

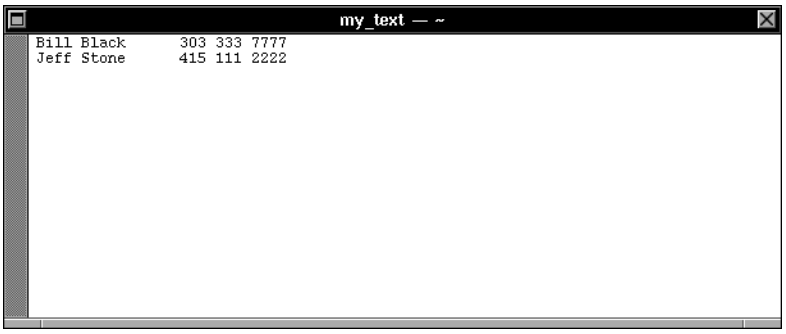
You can choose your own delimiters between fields and records but we will use the default now, *tab* and *newline*.

↔ **Click the *Export all* button in the Database Manager.**

↔ **Type a file-name in the panel that appears.**

The data in the database will be exported to the text-file and it will be opened in Edit. This text-

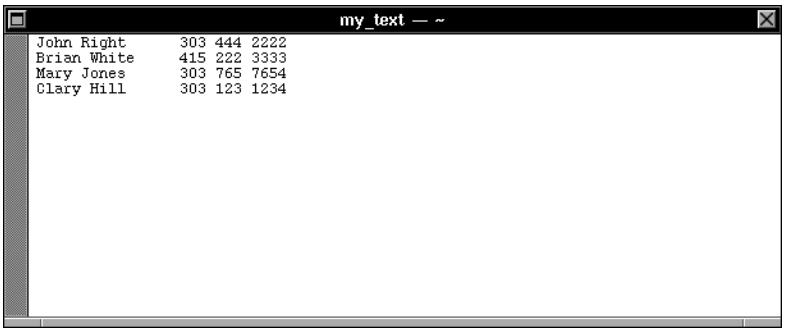
file can be opened in most word processors and spreadsheet programs on the market.



Picture 127. The exported text-file.

In a similar way you can import data from a text-file.

↔ **Create a text-file with records like this.**



Picture 128. A text-file with records (tab between the fields).

↔ **Choose *Import* in the pop-up menu.**

You can choose your own delimiters between fields and records but we will use the default now, *tab* and *newline*.

↔ **Click the *Import* button in the Database Manager.**

↔ **Choose the file you just created in the Panel.**

You will now have added all the records in the text-file to the database.

The export and import functions of the database makes it easy to exchange data between the database and other third-party applications.