079e21c8-24

COLLABORATORS

	TITLE : 079e21c8-24	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 22, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

079e21c8-24

1.1 EAGUI Tutorial

```
  $RCSfile: Tutorial.guide $
  $Revision: 3.2 $
  $Date: 1994/11/28 22:33:28 $

  Tutorial

 ~Introduction~~~~~~~~~~~~~~~
 ~Concepts~~~~~~~~~~~~~~~~~~
 ~Designing~the~interface~~~
 ~Using~the~interface~~~~~~~
 ~Includes~and~macros~~~~~~~
 ~Standard~methods~~~~~~~~~~
 ~Creating~your~own~methods~
 ~Relations~~~~~~~~~~~~~~~~
 ~Custom~images~~~~~~~~~~~~~
 ~Advanced~topics~~~~~~~~~~~
 ~Bibliography~~~~~~~~~~~~~~
```

1.2 Introduction

```
  Introduction
```

The Environment Adaptive Graphic User Interface (EAGUI) is a system which allows you to build interfaces that, as the name suggests, adapt to the environment they're run in. It uses normal GadTools and BOOPSI gadgets, and does not modify them in any way. This allows programmers to implement EAGUI in existing applications easily.

This tutorial is a practical introduction to EAGUI. After you've read this, you should be able to write your own interfaces, using the AutoDocs as a reference. Apart from reading this document, you might also want to take a look at the examples. The tutorial assumes you are familiar with programming the AmigaOS [2] and some knowledge of C is useful too, because all examples are written in C.

This tutorial is based on a simple example. The full source of this example can be found in example.c .

Not every option is discussed here, so please study the AutoDocs carefully.

1.3 Concepts

Concepts

EAGUI divides the contents of a window in a hierarchical object tree. Objects can be either gadgets, images or groups. Groups can contain any number of objects. There are two types of groups: horizontal and vertical groups. Horizontal groups contain objects that are placed next to each other, vertical groups contain objects that are placed below each other. An object tree contains all information to create the gadgets and images in a window.

For example, let's look at a string requester window that contains a text label, a string gadget, and an "Ok" and "Cancel" button below it. Starting at the top of the tree, we define a vertical group to divide the window in two. The upper object will contain the text label, the middle one the string gadget and the lower one will contain the buttons. The lower object in turn is a horizontal object, which contains three objects: the "Ok" button, an empty spaceholder and the "Cancel" button. The complete tree now looks like this:

```
Vertical group
     Text label
     String gadget
     Horizontal group
          "Ok" button gadget
          Empty spaceholder
          "Cancel" button gadget
```

This tree already holds some information about how objects should be placed in the window, but not enough. Let's assume we want to create a resizable window, since this is clearly the most difficult type of window to maintain.

A window can't be arbitrarily small, or else the objects won't fit inside the window. There is a minimum size. This size is determined by looking at the minimum size for each individual object in the window. We start at the bottom of the tree, where we find three objects. It is obvious that the labels of the buttons have to fit within the frame. The minimum size of these objects therefore is determined by the dimensions of the label. The other object, the empty spaceholder, is merely an object that fills up the empty space between the "Ok" and "Cancel" gadgets. This space can be arbitrarily small, so its minimum sizes are zero. Going up in the tree, we arrive at the Horizontal group object. The minimum sizes of its members are already determined, so the minimum sizes of the group are obtained by adding the horizontal minimum sizes of all children and taking the largest vertical size of the children. The next object, the string gadget, also has a minimum size that is determined by the dimensions of the string that has to fit inside the gadget. Finally, going up again, we arrive at the vertical group. Again, its members are known, so determining the size of the group is easy.

All size calculations can be done at runtime. This is necessary because an application can not know in advance what font or locale the user has selected, and there is no valid reason why the user should be forced to use a

certain font or locale. To allow flexible size calculations, each object uses a method to determine its minimum size. This method is implemented as a Hook. If you want, you can write this Hook function yourself, but for normal gadgets, general purpose methods are already supplied. More information about these Hooks can be found lateron.

Apart from the minimum sizes, objects have borders, whose sizes can also be determined using methods if you want. Borders can be seen as whitespace around objects.

It can sometimes be hard to remember the exact relation between the size, border size and offset of an object. Therefore, we've supplied a simple IFF picture, which shows the relations graphically. Click here to have a look at it.

Now that we know the minimum sizes, we also know the minimum inner size of the window. Please note that to determine the minimum size of the window, we have to add the sizes of the window borders. This presents us with a small problem, since we do not know the thickness of the borders in advance. There are several ways of dealing with this problem. These will be discussed lateron.

The next step is to actually open the window. After we've done that, we actually know the inner sizes of the window, and therefore we know the size of the root (top) object, which, in our example, is the vertical group. The object is set to this size. From hereon, the sizes of all other object are calculated.

We've already mentioned the minimum sizes, but there is another important object attribute we've left out up til now. Suppose that the window is bigger than the minimum size: how do we divide the available space? Some objects will want to use as much space as possible, while others will want to remain the same size. To solve this problem, each object can have a weight. This is a factor, which indicates how an object will grow in relation to other objects in the same group. Let's return to our example.

The vertical group contains three objects. Each object has a vertical size, that really doesn't need to grow bigger. In practice, this means that the whole window doesn't need to be resizable in the vertical direction, only in the horizontal direction. The horizontal size of the objects is automatically the same as the window's inner width. Now we take a look at the button gadgets. These are grouped in the lower horizontal group. If this group becomes wider, the "Cancel" button should move to the right too. Looking at the three objects in this group, we should allow the middle one (i.e. the empty spaceholder) to grow bigger. We do this by setting its weight to one. Both buttons can have a fixed size, so their weights are zero. An alternative is to give the buttons a weight of one too. That means that the available horizontal space will be evenly divided between the three objects, because each object has the same weight. As you can see, this is a very simple, yet powerful concept.

If you want to have a look at the example window, click here to run the example.

One thing to keep in mind is that if a group contains no weighed objects, it is not filled completely. This is perfectly legal, starting with release~2.2 and you don't need an empty object at the end of such a group.

1.4 Designing the interface

Designing the interface

The first thing you have to do, is make a sketch of the interface. You can either do this on a piece of paper, or by using some structured drawing tool. At this stage, you'll have to figure out how to divide the interface in groups of objects.

When you start designing an interface, make sure you've read The Amiga User Interface Style Guide [1] . It contains a lot of guidelines, and following these will make your interface easier to use.

Also, it is good practice to look at available software, and see how particular problems are solved. Especially the preference editors on your Workbench are good examples of how interfaces should look like (although they don't font adapt).

1.5 Using the interface

Using the interface

Basically, there are three main services you must provide to work with a window. First, you must initialize everything. Then you must handle incoming messages, until at some point the window is forced to close again, which is the third and last service. This is pretty straightforward.

Let's look at the initialization first. Creating a new object is quite simple, and anybody who has used BOOPSI will be familiar with the type of function call. All you do is call ea_NewObjectA(). This function takes the object type as the first argument, and a pointer to a taglist as the second one. Alternatively, you can use ea_NewObject(), which allows you to pass the individual tags as arguments. If all went well, the function will return a pointer to the object, which can be used as a handle. It is not encouraged nor allowed to make assumptions about the data that the pointer points to. The ea_NewObjectA() function can be used to create a whole tree easily. For more information, look at the AutoDocs. Let's take another look at our example:

```
LONG init(VOID)
{
     if (!(winobj_ptr = ea_NewObject(EA_TYPE_VGROUP,
                EA_Child, ea_NewObject(EA_TYPE_CUSTOMIMAGE,
               [...]
                         ),
          EA_Child,      ea_NewObject(EA_TYPE_GTGADGET,
               EA_GTType,     STRING_KIND,
               [...]
               ),
          EA_Child,      ea_NewObject(EA_TYPE_HGROUP,
               EA_Child,      ea_NewObject(EA_TYPE_GTGADGET,
                    EA_GTType,     BUTTON_KIND,
                    EA_GTText,     "Ok",
                    [...]
                    ),
```

```
                    EA_Child,      ea_NewObject(EA_TYPE_CUSTOMIMAGE,
                         [...]
                         ),
                    EA_Child,      ea_NewObject(EA_TYPE_GTGADGET,
                         EA_GTType,     BUTTON_KIND,
                         EA_GTText,     "Cancel",
                         [...]
                         ),
                    [...]
                    ),
               [...]
               )))
          {
               Printf("Couldn't create object tree.\n");
               return(20);
          }
          return(0);
     }
```

Some tags have been left out here, as marked by the ellipsis [...], because they are not essential to the example. Some other tags must be specified for GadTools or BOOPSI gadgets, and information on this can be found in the AutoDocs of these libraries. The most important thing is the way you can hierarchically define a tree. If something along the way goes wrong (which is usually because there wasn't enough memory) then nothing is created (all objects in the tree which were already created will be freed automatically), and NULL is returned.

Note that we have split the string gadget and its label in two seperate objects. We did this to demonstrate the custom image facilities implemented in EAGUI. Of course it is possible to let GadTools place the label above the string gadget.

Now that we've initialized a tree, we must also have a way to clean it up. This is even simpler. All we need is a simple call to ea_DisposeObject(). This removes an object and all its children. In our example, it would look something like this:

```
VOID cleanup(VOID)
{
     ea_DisposeObject(winobj_ptr);
}
```

The next step is to calculate the minimum sizes. If you want the window to be resizable, then you must set the window sizing limits with the WindowLimits() call. Before that, you must first obtain the minimum sizes. This can be done easily by using ea_GetAttrs() on the 'winobj_ptr'. The following example explains all this:

```
/* obtain the minimum sizes of every object in the tree */
ea_GetMinSizes(winobj_ptr);

/* get some attributes */
ea_GetAttrs(winobj_ptr,
     EA_MinWidth,        &w,
     EA_MinHeight,       &h,
     EA_BorderLeft,      &bl,
```

```
          EA_BorderRight,     &br,
          EA_BorderTop,       &bt,
          EA_BorderBottom,    &bb,
          TAG_DONE);

     /* open the window */
     win_ptr = OpenWindowTags([...]);

     /* set the window limits */
     WindowLimits(
          win_ptr,
          w + win_ptr->BorderLeft + win_ptr->BorderRight + bl + br,
          h + win_ptr->BorderTop + win_ptr->BorderBottom + bt + bb,
          ~0,
          h + win_ptr->BorderTop + win_ptr->BorderBottom + bt + bb);

     /* fill in the inner sizes of the window in the root object */
     ea_SetAttrs(winobj_ptr,
          EA_Width,      win_ptr->Width -
                         win_ptr->BorderLeft -
                         win_ptr->BorderRight -
                         bl -
                         br,
          EA_Height,     win_ptr->Height -
                         win_ptr->BorderTop -
                         win_ptr->BorderBottom -
                         bt -
                         bb,
          EA_Left,       win_ptr->BorderLeft,
          EA_Top,        win_ptr->BorderTop,
          TAG_DONE);

     /* now determine the object sizes and positions */
     ea_LayoutObjects(winobj_ptr);

     /* create the list of gadgets for this window */
     rc = ea_CreateGadgetList(winobj_ptr, &gadlist_ptr,
          visualinfo_ptr, drawinfo_ptr);

     /* add the gadgetlist to the window */
        AddGList(win_ptr, gadlist_ptr, -1, -1, NULL);
        RefreshGList(gadlist_ptr, win_ptr, NULL, -1);
        GT_RefreshWindow(win_ptr, NULL);

        /* finally, we render the custom imagery, if there is any */
        ea_RenderObjects(winobj_ptr, win_ptr->RPort);

     if (rc != 0)
     {
          /* bail out */
          exit(20);
     }

     /* now you're ready for business */
```

You can now handle events in exactly the same way you normally do. The only exception is the IDCMP_NEWSIZE event. This is where you can use EAGUI to adapt

to the new window size. If you receive one of these, you must do the following things:

a) Store any unsaved changes the user has made to the gadgets. This means storing strings that were entered in string gadgets, items that were selected in listviews, cycle gadgets or radio buttons, and things like that. Depending on your program structure, this may not be necessary at all, because the changes were already stored when the IDCMP message that notified that change was received.

b) Remove the gadget list from the window, and clean it up, like this:

```
RemoveGList(win_ptr, gadlist_ptr, -1);
ea_FreeGadgetList(winobj_ptr, gadlist_ptr);
gadlist_ptr = NULL;
```

c) Examine the new dimensions of the window, and set the root object accordingly, like this:

```
ea_GetAttrs(winobj_ptr,
     EA_BorderLeft,      &bl,
     EA_BorderRight,     &br,
     EA_BorderTop,       &bt,
     EA_BorderBottom,    &bb,
     TAG_DONE);

ea_SetAttrs(winobj_ptr,
     EA_Width,           win_ptr->Width -
                         win_ptr->BorderLeft -
                         win_ptr->BorderRight -
                         bl -
                         br,
     EA_Height,          win_ptr->Height -
                         win_ptr->BorderTop -
                         win_ptr->BorderBottom -
                         bt -
                         bb,
     EA_Left,            win_ptr->BorderLeft,
     EA_Top,             win_ptr->BorderTop,
     TAG_DONE);

ea_LayoutObjects(winobj_ptr);
```

d) Build the gadget list again, and connect it to the window. Currently, the method I use to refresh the window looks somewhat strange. I haven’t quite figured out what to do with it. It seems that GadTools refreshes the gadgets directly after resizing the window, but before you’re notified of that fact. Therefore, it renders over your window borders. After that, you get a IDCMP_NEWSIZE message, and you have to redraw everything yourself (including the window borders). Finally redraw the custom images. This is example of working code:

```
rc = ea_CreateGadgetList(winobj_ptr, &gadlist_ptr,
     visualinfo_ptr, drawinfo_ptr);
if (rc != 0)
{
     /* bail out */
```

```
        exit(20);
    }
    EraseRect(win_ptr->RPort,
        win_ptr->BorderLeft,
        win_ptr->BorderTop,
        win_ptr->Width - win_ptr->BorderRight - 1,
        win_ptr->Height - win_ptr->BorderBottom - 1);

    RefreshWindowFrame(win_ptr);

    AddGList(win_ptr, gadlist_ptr, -1, -1, NULL);
    RefreshGList(gadlist_ptr, win_ptr, NULL, -1);
    GT_RefreshWindow(win_ptr, NULL);

    /* finally, we render the imagery, if there is any */
       ea_RenderObjects(winobj_ptr, win_ptr->RPort);
```

1.6 Includes and macros

Includes and macros

When using the library in your own program, there are certain headers that you need to include.

First, you must include EAGUI.h , which contains all necessary information for using the library. This header then automatically includes EAGUI_protos.h , which contains all needed function prototypes. There is one switch that may be of use. If you #define NOEAGUIMACROS before you include the header file, you won't get the macros in EAGUI_macros.h , otherwise you will. For normal circumstances, the macros seem to work well, but you may have your own personalized set of macros that you're more comfortable with, so we won't force you to use them.

The second file you should include (at least when you're using SAS/C) is a pragmas file (called EAGUI_pragmas.h). Note that this file is not part of the distribution, because it is compiler specific, and can be generated from the EAGUI.fd file. Users of other languages may find the EAGUI.fd file useful too. If you've written header files for any other language, and you want us to include them in the EAGUI package, please contact us.

1.7 Standard methods

Standard methods

For all GadTools gadgets, minimum size and border methods are supplied in the library.

The methods that are supplied will try to adapt to all different tags that have an influence on the size of the object. The minimum size methods basically make the gadgets big enough to just fit. String gadgets for example will be high enough for the selected font, and wide enough for the border to be rendered succesfully. Border size methods will adapt to labels which are placed above, left of, right of or below a gadget. Please note that if, for example, you place a text above a gadget, the top border will only be high

enough for the text to fit. If the text is wider than the gadget, the gadget will not be adjusted: the label will simply be rendered, and you'll have to make sure that this doesn't conflict with other objects.

To use the standard methods, you must specify the EA_StandardMethod tag, like this:

```
ea_NewObject([...]
     EA_StandardMethod,  value,
     [...]
```

The 'value' parameter can be EASM_MINSIZE, EASM_BORDER or a combination of both (EASM_MINSIZE | EASM_BORDER), depending on what standard methods you want to use for this object. The EAGUI_macros.h file already uses this tag, so if you use these macros, the standard methods will be used automatically.

1.8 Creating your own methods

Creating your own methods

Although the standard methods will be sufficient in many cases, there might be times when you want to create your own methods. A good example would be when you use BOOPSI gadgets. Other reasons for creating your own methods could be that your interface uses gadgets of a much simpler form, or that it only uses certain types of gadgets. Custom methods might improve execution speed slightly, since they don't need to be 'universal'.

When creating you own methods, many things are possible, but there are a few things you have to keep in mind.

Every method is called using a callback hook. For more information about initializing and using hooks, please refer to the RKRM's [2] or the <utility/hooks.h> header file.

A method prototype should look like this:

```
ULONG method(
     struct Hook *       hook_ptr,
     struct ea_Object *  object_ptr,
     APTR                message_ptr);
```

The hook_ptr contains the pointer to the hook structure of the method.

The object_ptr points to the object that the method should be applied to.

The message_ptr is not used at the moment.

The method can change the attributes of the object. We strongly discourage you to let the method change any other attributes than the ones it should set. So a border method should only set the border attributes of the object, and a minsize method should only set the minsize attributes. You are allowed to read other attributes, if you need them for your calculations. Always use ea_GetAttrs() and ea_SetAttrs() to read and write the attributes respectively.

Note that when the object is created, it is possible to pass border, size or minsize attributes. EAGUI protects these values, so your method can't change them, even if it tries to. All this is done automatically, so you don't need to worry about this.

1.9 Relations

Relations

In some interfaces, you want to create special relations between objects. One example that comes to mind is a simple window with an "Ok" and a "Cancel" button. If these both have a fixed size, then one of them will be bigger than the other, simply because the label strings don't have the same width. This does not look very good. To correct this, you can add a relation between the two objects. All objects in a relation must have the same direct parent. This relation is added to the parent of the objects. For example:

```
ea_NewRelation(obj_hgroup_ptr, &relhook,
     EA_Object,          obj_okgad_ptr,
     EA_Object,          obj_cancelgad_ptr,
     TAG_DONE);
```

Any number of objects can be connected using one and the same relation. You can simply specify a list of objects by using the tag shown above. Relations also use the Hook mechanism. They're called after the minimum sizes were calculated, so you can read the minimum sizes of the objects in this method.

To explain how to create relations, a simple example is included below:

```
/* same size relation */
ULONG rel_samesize(struct Hook *hook_ptr, struct List *list_ptr,
     APTR msg_ptr)
{
     struct ea_RelationObject *ro_ptr;
     ULONG minx, miny;
     ULONG x, y;

     minx = 0;
     miny = 0;

     /* examine the list of objects that are affected by the relation */
     ro_ptr = (struct ea_RelationObject *)list_ptr->lh_Head;
     while (ro_ptr->node.ln_Succ)
     {
          ea_GetAttrs(ro_ptr->object_ptr,
               EA_MinWidth,        &x,
               EA_MinHeight,       &y,
               TAG_DONE);

          /* find the maximum values of the minimum sizes */
          minx = MAX(x, minx);
          miny = MAX(y, miny);

          ro_ptr = (struct ea_RelationObject *)ro_ptr->node.ln_Succ;
     }
```

```
        /* set all objects to the newly found minimum sizes */
        ro_ptr = (struct ea_RelationObject *)list_ptr->lh_Head;
        while (ro_ptr->node.ln_Succ)
        {
             ea_SetAttrs(ro_ptr->object_ptr,
                  EA_MinWidth,        minx,
                  EA_MinHeight,       miny,
                  TAG_DONE);

             ro_ptr = (struct ea_RelationObject *)ro_ptr->node.ln_Succ;
        }
        return(0);
    }
```

The example is pretty self-explaining.

1.10 Custom images

```
  Custom images
```

Support for images is something new in Release 2 of the library. Because of the diversity of custom imagery, it still requires a fair amount of programming, but it is integrated with the rest of the interface, so it will probably save you work in the long run.

Basically, when you want to use a custom image, you have to do the following things:

- Combine all extra attributes your image needs to render itself into one custom structure, and use the EA_UserData tag to add this structure to the custom image object.
- Write a method that determines the minimum size the image occupies.
- Write a method that renders the image.

Let’s make a simple custom image that you’ll probably need anyway: a text label image. The full code to this example can be found in TextField.c . First, we’ll define the attributes that this image needs to render itself. We need a pointer to the string that must be displayed, a pointer to a TextAttr structure, which determines the font we want to use, a field that contains some layout flags (which determine how the text is aligned within the object space) and a pen number, which determines which color will be used to render the text.

These are all combined in the ci_TextField structure, which is declared in the TextField.c source.

The next thing is a method that determines the minimum size of the object. This is pretty straightforward. All we have to do is determine the dimensions of the text. EAGUI provides two functions for that: ea_TextLength() and ea_TextHeight(). We simply use these to fill in the appropriate fields. Look at the source to see how this is implemented.

Finally, we have to write a method that actually renders the object on screen. This callback hook is new, and it is a bit different from the ones mentioned

previously, because it uses the third hook parameter to pass some extra parameters. The ea_RenderMessage structure contains a pointer to the root object and a pointer to the RastPort that should be used for rendering. Why we need the root pointer might not be immediately obvious. Because the object itself only contains relative offsets, we need it to determine the position of the object in the window (ie relative to the root pointer). EAGUI provides two functions for that, which both need that pointer: ea_GetObjectLeft() and ea_GetObjectTop(). The actual rendering can then be performed. Take a look at the source. It is a pretty basic example, that can be expanded (for example to support an underlined hotkey, or even more fancy stuff).

1.11 Advanced topics

Advanced topics

This section contains a collection of specific advanced topics. Some of them are workarounds for known problems, others are about creating new types of objects.

~A~string~gadget~connected~to~a~listview~
~GadTools~gadgets~and~refreshing~~~~~~~~~
~Creating~pages~of~gadgets~~~~~~~~~~~~~~~

1.12 A string gadget connected to a listview

A string gadget connected to a listview

There is a known problem with GadTools. Connecting a string gadget to a listview gadget will not work. This is because the taglist of the listview gadget must contain a pointer to an already created string gadget. It is not possible to have these kind of dependencies. The only remedy is not to use this option, and glue the two gadgets together in your application. Fixing this problem is not easy, especially since it’s a create-time only tag, so there’s no way to make the connection afterwards.

1.13 GadTools gadgets and refreshing

GadTools gadgets and refreshing

A strange thing, that you might have noticed, is that when you resize a window to a smaller dimension, GadTools will refresh the gadgets in their old state, and draw them over the window borders. In fact, this refresh is done if you make the window larger too, but it’s harder to notice it in that case. This refresh is something that happens before you’re even notified of a size change, so there’s not much you can do about it unless you use the method described here, which has a drawback too.

As stated, when you receive the IDCMP_NEWSIZE message, it’s already too late. However, if your program code allows you to use IDCMP_SIZEVERIFY messages, you can do this:

```
     case IDCMP_SIZEVERIFY:
          RemoveGList(win_ptr, gadlist_ptr, -1);
          ea_FreeGadgetList(winobj_ptr, gadlist_ptr);
          gadlist_ptr = NULL;
          break;
```

Right after you've replied to this message, you'll receive a IDCMP_NEWSIZE message, where you do everything you normally do (except remove the gadget list, which you've already done of course). Before you use this method, make absolutely sure that you read the warnings in the AutoDocs about the verify messages. They can be found in the intuition.library/ModifyIDCMP() AutoDoc. Don't say we didn't warn you!

1.14 Creating pages of gadgets

Creating pages of gadgets

You might have wondered how to create pages of gadgets with EAGUI. Although GadTools normally doesn't support this, it is possible to create pages yourself. It will require some work, but you can use EAGUI to do all the hard work for you. If you want to see the pages example, click here to run it now.

Let's examine the concept of pages a bit closer. In essence, pages are nothing more than a special kind of group: the page group. Each child of this group is a page, and only one page is shown at a time.

Let's start with using one of the group types supplied by EAGUI: the horizontal group. This is chosen arbitrarily. Nothing would change if we'd chosen the vertical group. Let's make three pages. Because only one of them should be shown at any time, we must disable the two others. To do this, we simply set the EA_Disabled tag to TRUE for two of the children.

To determine which group is currently selected, we can use an ordinary cycle gadget. This gadget is then placed directly above the group by making a vertical group containing the cycle gadget and the page group.

Every time the user changes the page, we must select the right page, disable the others and recreate the complete gadget list. Actually, we should only change the gadgets in the group, but this is more tricky and is therefore not done in this example.

There's only one additional thing that needs to be taken care of: calculating the correct minimum size of a page group. If we use the MinSize method of a horizontal group, we get the size in which all pages would fit next to each other. That is not correct. Therefore, we should write our own method for this group. It should look at the minimum sizes of each child and choose the largest minimum size of them. This will ensure that the window size will never be too small if the user merely changes the page. Writing such a method is rather trivial.

The complete source of an example that uses three pages can be found in the pages.c file. You should be able to understand it after having read this section. It is basically a modified version of the example.c source file.

1.15 Bibliography

Bibliography

[1] The Amiga User Interface Style Guide, Addison Wesley

[2] The Amiga ROM Kernel Reference Manuals, Addison Wesley