Nested widgets in parts

A.T. Hofkamp

March 7, 2009

Contents

1 Introduction

In the current implementation, a GUI window is contructed by means of a widget array containing the data fields for all widgets of the window. These data fields include fixed positions of the edges of each widget relative to the origin of the window.

The following points for possible improvement have been recognized:

- Modifying the size of one widget also means that other widgets have to be re-positioned. This usually involves a lot of recalculating widget edge positions.
- The sizes of the widgets should be more flexible. The program supports many different languages, and not all languages need the same amount of space in each widget.

In this document, *nested widgets* are proposed as a approach to combat these points.

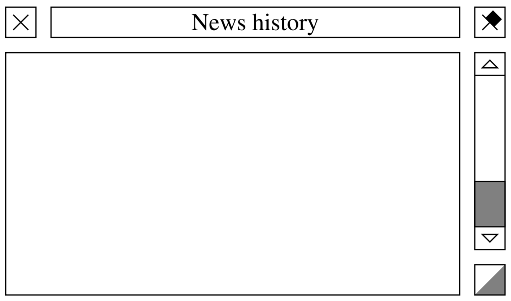

Figure 1: Widgets of the *news history* window.

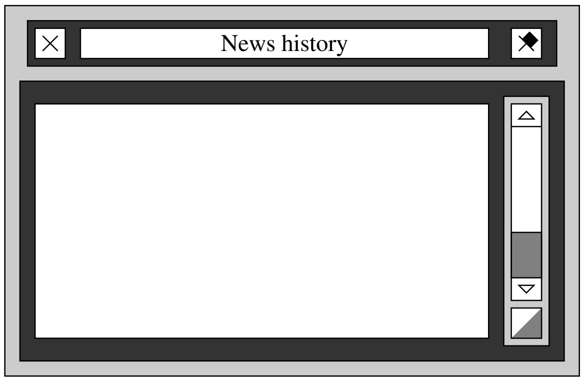

Figure 2: Nested widgets version of the *news history* window.

2 Nested widgets

To overcome the weak points of the current static widget implementation, we may want to switch to using nested widgets.

Nested widgets bring a hierarchy of widgets (a tree of widgets). At the leafs, there are the *user widgets* (buttons, labels, etc). These widgets know their own position and (minimal) size, and know how to react to resizing requests[1]. To keep the user widgets together, *structural widgets* are used. These are at the non-leaf nodes in the tree. They manage the size and position of their child widgets, and ultimately at the root of the hierarchy, the window as a whole.

In the OpenTTD nested widgets, the two main structural widgets are the *horizontal container widget* and the *vertical container widget*.[2] The horizontal container widget puts its child widgets after each other from left to right[3]. The vertical container puts its child widgets underneath each other.

To demonstrate the idea, have a look at Figure 1. It shows the widgets used to create the *news history* window. For clarity, some space is put between the widgets. In Figure 2, structural widgets have been added. The three widgets of the title bar have been put in a horizontal container widget (dark gray area). The vertical scrollbar and the resize button have been put together in a vertical container widget (light gray area). The canvas for the messages and the vertical container holding the scrollbar and resize button are then put in another horizontal container. Finally both horizontal containers are put in the root vertical container widget.

Note that in reality, there is no room between the container edge and its contents, so wrapping container widgets around some other widgets does not make the window bigger.

In terms of C++ objects, the tree of widgets is clearly visible. Figure 3 gives an impression.

Using nested widgets addresses the points raised in Section 1, and has some additional advantages:

- Since each user widget knows its own size, changing its size is a local modification. The structural widgets will shuffle all the other widgets.

[1]Further extensions are possible, for example rendering to the screen.

[2]Much like the HBox and VBox classes in the GTK GUI toolkit, or the QHBoxLayout and QVBoxLayout classes in Qt.

[3]And for countries that use right-to-left languages, from right to left.

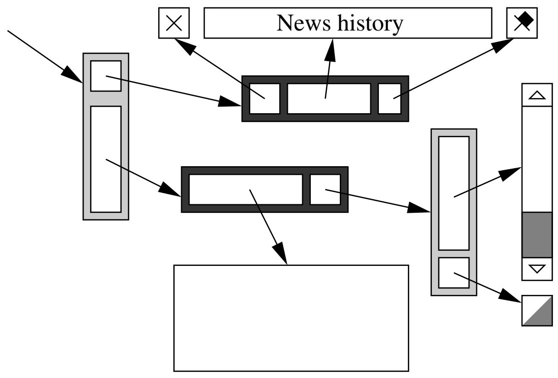

Figure 3: Internal nested widgets data structures.

- The same mechanism can be used to adapt the widget sizes opitmally for each language. Overflow of the text will not happen any more.
- The widget tree computes the minimal size of the window, making the hard-coded values in the WindowDesc class obsolete.
- The widget tree computes the resize steps of the window. (Each user widget knows how to react to resizing. The structural widgets collect that information and propagate it upwards to the root.)

The current aim of the patch is to introduce nested widgets, and to generate the widget array. The nested widgets are however much more powerful, and can be extended to a more powerful replacement of the current widget implementation.

The technical details of constructing a nested widget tree can be read in Appendix A. Basically, what you need to do is to construct the whole tree bottom up by instantiating each widget and putting them in containers, putting those containers other containers, etc until the root of the nested widget tree is reached. At that point, you can initalize the whole tree with just two method calls, and generate the filled widget array with a third call into the nested widget tree.

3 Nested widget parts

Doing that by hand results in a lot of code that is not very nice to read and change. To solve this problem, another data structure has been made, the *nested widget parts* structure. It allows you to define the nested widget tree structure by listing all widgets in the same nested structure. Together with proper indentation, it looks like a program fragment. An example:

```
static const NWidgetPart _nested_select_game_widgets[] = {
    NWidget(WWT_CAPTION, COLOUR_BROWN, 0),
            SetMinimalSize(336, 14), SetDataTip(STR_0307_OPENTTD),
    NWidget(WWT_PANEL, COLOUR_BROWN, 1),

        NWidget(NWID_SPACER), SetMinimalSize(0, 8),
        /* 'generate game' and 'load game' buttons */
        NWidget(NWID_HORIZONTAL),
            NWidget(NWID_SPACER), SetMinimalSize(10, 0),
            NWidget(WWT_PUSHTXTBTN, COLOUR_ORANGE, SGI_GENERATE_GAME),
                    SetMinimalSize(158, 12),
                    SetDataTip(STR_0140_NEW_GAME, STR_02FB_START_A_NEW_GAME),
            NWidget(WWT_PUSHTXTBTN, COLOUR_ORANGE, SGI_LOAD_GAME),
                    SetMinimalSize(158, 12),
                    SetDataTip(STR_0141_LOAD_GAME, STR_02FC_LOAD_A_SAVED_GAME),
            NWidget(NWID_SPACER), SetMinimalSize(10, 0),
        EndContainer(),

        NWidget(NWID_SPACER), SetMinimalSize(0, 6),

        .... // Other lines with buttons deleted to increase clarity.

    EndContainer(),
};
```

This fragment defines the top-part of the window at the ntro screen (the caption and the first row of buttons).

The whole definition is an array of `NWidgetPart` structures. The root is a vertical container widget. A new widget starts at a `NWidget` call. Behind it, additional settings for the widget can be done. For user widgets only settings like `SetMinimalSize()` or `SetDataTip()` are useful. For background (WWT_PANEL, WWT_FRAME, and WWT_INSET widgets) and container widgets (NWID_HORIZONTAL and NWID_VERTICAL) collect all widgets following until they encounter a `EndContainer()`. See Appendix B for the technical details of what is possible.

4 Integration of nested widgets with the current implementation

In the current implementation, the widget array is a central data structure, and cannot be replaced by something else easily.

For this reason the aim of this patch is only introduction of the nested widget data structure for describing GUI windows. The WindowDesc data structure has been extended to allow nested widget parts as window specification. If it contains both the current widget array and the nested widget parts, the latter is converted to another widget array, and both arrays are compared with each other. If the original widget array is not given and nested widget parts are specified, the latter is used as window specification.

NLeafWidget type	**Mininal size**	**Filling**	**Resize step**
WWT_PUSHBTN	(0, 0)	(false, false)	(0, 0)
WWT_IMGBTN	(0, 0)	(false, false)	(0, 0)
WWT_PUSHIMGBTN	(0, 0)	(false, false)	(0, 0)
WWT_IMGBTN_2	(0, 0)	(false, false)	(0, 0)
WWT_TEXTBTN	(0, 0)	(false, false)	(0, 0)
WWT_PUSHTXTBTN	(0, 0)	(false, false)	(0, 0)
WWT_TEXTBTN_2	(0, 0)	(false, false)	(0, 0)
WWT_LABEL	(0, 0)	(false, false)	(0, 0)
WWT_TEXT	(0, 0)	(false, false)	(0, 0)
WWT_MATRIX	(0, 0)	(false, false)	(0, 0)
WWT_EDITBOX	(0, 0)	(false, false)	(0, 0)
WWT_SCROLLBAR	(12, 0)	(false, true)	(0, 1)
WWT_SCROLL2BAR	(12, 0)	(false, true)	(0, 1)
WWT_HSCROLLBAR	(0, 12)	(true, false)	(1, 0)
WWT_CAPTION	(0, 14)	(true, false)	(1, 0)
WWT_STICKYBOX	(12, 14)	(false, false)	(0, 0)
WWT_RESIZEBOX	(12, 12)	(false, false)	(0, 0)
WWT_CLOSEBOX	(11, 14)	(false, false)	(0, 0)
WWT_DROPDOWN	(0, 12)	(false, false)	(0, 0)
WWT_DROPDOWNIN	(0, 12)	(false, false)	(0, 0)

Figure 4: Default sizing settings for each type of `NLeafWidget` object.

A Technical details of making nested widget trees

A.1 User widgets

User widgets are created with the `NLeafWidget` class, as in

```
NWidgetLeaf *nwid = new NWidgetLeaf(type, colour, index, data, tip);
nwid->SetMinimalSize(minx, miny); // optional
nwid->SetFill(fillx, filly);      // optional
nwid->SetResize(stepx, stepy);    // optional
```

The widget is created by instantiating the `NLeafWidget` class. The `type` parameter defines what widget to create. Depending on the type, there are default values for the sizing settings listed in Figure 4. The `data` and `tooltip` defaults are `0x0` and `STR_NULL` respectively. The `colour` parameter defines the colour of the widget, and `index` sets at what position the widget should be placed in the widget array. The `data` and `tip` parameters define the values for the Widget::data and the Widget::tooltips fields.

If the default sizing settings are not correct, they can be modified by calling `SetMinimalSize()` for chaning the minimal size of the widget, `SetFill()` for changing how the widget is extended during filling, and `SetResize()` for modifying the resize step of the widget.

A.2 Container widgets

The container widgets are the structural nested widgets. They do not have size and resize properties of their own, but instead derive them from their child widgets. The simplest container widget is the vertical container widget. It puts its child widgets underneath each other. Instantiating a vertical container widget is done like

```
    nwid = new NWidgetVertical();

    // Add child widgets by calls like
    nwid->Add(child_widget);
```

The container widget is never stored in the widget array, and thus has no parameters or methods for them. After creation, adding a child widget to the container is done with the `Add()` method. The first added child widget is put at the top, each next added child widget is put below the previously added one.

The size and resize properties of the container are derived from its child widgets in the following way:

- Its horizontal size is the biggest horizontal size of one of its child widgets. Smaller child widgets that allow horizontal fill are expanded in horizontal direction. Smaller child widgets that cannot be horizontally filled are centered.
- Its vertical size is the sum of the vertical sizes of all its child widgets.
- It can fill in horizontal direction if all child widgets can do horizontal fill.
- It can fill in vertical direction if at least one child widget can do vertical fill.
- The horizontal resize step is non-zero only if the horizontal resize step of all its child widgets are non-zero. Its step size if the smallest common multiple of all its child widget horizontal resize steps.
- The vertical resize step is 0 if none of its child widgets can resize verrtically. Otherwise, it is the resize step of the first vertically resizable child widget. All other child widgets are assumed not to be vertically resizable.[4]

The horizontal container (`NWidgetHorizontal` class) acts just like the vertical container, except that vertical and horizontal are swapped. Which way 'top' goes depends on the language used. With languages that write from left to right, the horizontal container [5] starts at the left and extends to the right. With languages that write from right to left, the 'top' goes to the right and the widget put each child at the left of the previous child.

A.3 Background widgets

The WWT_PANEL, WWT_INSET, and WWT_FRAME widgets can be used like a normal user widget, or as a widget that functions as a background of another (container) child widget.

In the first case, instantiation is something like

[4]What you would want here is that it takes the smallest non-zero child vertical resize step, and also distribute resizing evenly over all vertically resizable child widgets. The nested widget tree can easily be extended to handle that. However it is not supported by the resize flags in the widget array, thus there is currently no point in doing this.

[5]The direction of stacking is not a fixed property of the horizontal container widget, but is instead a parameter of the method that assigns sizes and positions. This allows switching between both types of languages without difficulties.

```
NWidgetBackground *nwid = new NWidgetBackground(type, colour, index, NULL);
nwid->SetMinimalSize(minx, miny);
nwid->SetFill(fillx, filly);      // optional
nwid->SetResize(stepx, stepy);    // optional
```

The `type` parameter must be one of WWT_PANEL, WWT_INSET, or WWT_FRAME. The `colour` defines the colour of the background widget, and the `index` sets the index of the widget in the widget array. The default minimal size is (0, 0), the widget allows filling in both directions, but no resizing. Usually, you should set its minimal size to a non-zero size by using the `SetMinimalSize()` method. If the default filling and resizing are not wanted, these can be changed with the `SetFill()` and `SetResize()` methods.

For the second case (use as a background widget), the instantiation sequence is something like

```
NWidgetContainer *ncont = new NWidgetVertical(); // Construct a container widget
// Fill the container
NWidgetBackground *nwid = new new NWidgetBackground(type, colour, index, ncont);
```

or more like a container widget:

```
NWidgetBackground *nwid = new new NWidgetBackground(type, colour, index, NULL);
// Make a child widget
nwid->Add(child_widget);
```

The fourth parameter of the background widget can be used to specify the child container widget to use. That container widget may be filled before hand, or afterwards. The `Add()` method of the background widget is simply forwarding the new widgets to its child container widget. If you don't specify a container, a vertical container is created automatically with the first call to `Add()`.

A.4 Space widget

The space widget is a user widget that only takes room, it is never rendered onto the window. Its purpose is to allow gaps between user widgets (thus creating groups of widgets), so the user has a better overview of the window interface structure. Instantiation is done like

```
NWidgetSpacer *nwid = new NWidgetSpacer(sizex, sizey);
nwid->SetMinimalSize(minx, miny); // optional
nwid->SetFill(fillx, filly);      // optional
nwid->SetResize(stepx, stepy);    // optional
```

In the instantiation statement, the minimal size of the space is defined. Normally, either `sizex` or `sizey` is 0. In that way, the space widget only influences one direction which makes understanding the widget structure easier. By default, the space widget does not change size. This can be modified with the same methods as used with the user widgets.

A.5 Initializing a nested widget tree

After constructing the nested widget tree, it needs to be initialized. This is done by two method calls:

```
bool rtl = false; // We are not using a right-to-left language.
// Construct the entire nested widget tree.
// Assuming 'nwid_root' is the root widget of the window.
int biggest_index = nwid_root->ComputeMinimalSize();
nwid_root->AssignPosition(0, 0,
                          nwid_root->min_x, nwid_root->min_y,
                          (nwid_root->resize_x > 0), (nwid_root->resize_y > 0),
                          rtl);
```

The first call (`ComputeMinimalSize()`) performs a bottom-up traversal of the widget tree, and pulls minimal size, fill, and resize information from the user widgets up to the root of the tree. After the call, the (`min_x`, `min_y`) pair contains the minimal size of the window, and the (`resize_x`, `resize_y`) pair contains the horizontal and vertical resize step sizes. The `AssignPosition()` call pushes the collected information down again, and makes all data consistent with each other. The fifth `rtl` boolean parameter defines whether a right-to-left language is used. Unfortunately, OpenTTD cannot handle the value `true` for the `rtl` parameter at this time.

The details to generate a widget array from an initialized nested widget tree is a single `StoreWidget()` method call. Details on how to do it can be found in the `InitializeNWidgets()` function.

B Technical details of using nested widget parts

The nested widget parts build on the nested widgets discussed in Appendix A.

The nested widget parts are slightly less expressive than making a nested widget tree manually. The reason for this is to make using them easier (for example, the background widgets always use a vertical container for their child widgets). In general, this should not give any problems.

B.1 Leaf widget parts

A leaf widget is written in neted widget parts in the following way:

```
NWidget(type, colour, index),
SetMinimalSize(minx, miny),
SetFill(fillx, filly),
SetResize(stepx, stepy),
SetDataTip(data, tip),
```

The meaning of the parameters is the same as with the nested widgets. The order of the settings of the second line and further is not relevant, but for consistency, the above order is recommended.

If the default values of a `NLeafWidget` are good, you can omit explicitly setting the value.

B.2 Container widget parts

Creating a container with all its child widgets is done with a declaration like

```
NWidget(NWID_HORIZONTAL),  // Use NWID_VERTICAL for vertical containers
    ... // Put child widgets here
EndContainer(),
```

B.3 Background widget parts

As in the nested widgets, the nested widget parts also allow two forms of background widgets, on the one hand as a user widget for rendering content, and on the other hand as a background widget for its child.

A background widget for rendering own content is created with

```
NWidget(type, colour, index),
SetMinimalSize(minx, miny),
SetFill(fillx, filly),
SetResize(stepx, stepy),
SetDataTip(data, tip),
EndContainer(),
```

You can define the same fields as for a leaf widget. The `type` parameter must be one WWT_PANEL, WWT_INSET, or WWT_FRAME. The `EndContainer()` is always needed. As with the leaf widgets, if the default field values of the background widget are good, you can omit setting it again.

A background widget used as a background for other widgets acts as a vertical container. Use it as follows:

```
NWidget(type, colour, index),
SetDataTip(data, tip),
    ... // Put child widgets here
EndContainer(),
```

where `type`, `colour`, and `index` is the same as above. In this case, the background widget gets its sizing information from the child widgets, so you don't need to set them here. The `SetDataTip()` part is only useful if its value must be different from the default values.

B.4 Space widget part

The space widget is never stored in the widget array, and thus only has a few data fields. Instantiation is done like

```
NWidget(NWID_SPACER),
SetMinimalSize(minx, miny),
SetFill(fillx, filly),
SetResize(stepx, stepy),
```

The `minx` parameter is normally 0 in vertical container widgets, and the `miny` parameter is normally 0 in horizontal container widgets. If the default fill and resize settings are correct, they can be omitted here.

B.5 Creating a nested widget tree from its parts

Constructing a nested widget tree from an array of nested widget parts is done by a single function call:

```
NWidgetContainer *nwid_root;
nwid_root = MakeNWidgets(_nested_select_game_widgets,
                         lengthof(_nested_select_game_widgets));
```