Creating UIs using the Layout API

(written by @waddlesplash)

Introduction

The Layout API is a new addition in Haiku. It allows for the creation of GUIs that are font- and size-aware without requiring the programmer to manually create the GUI based on the current font and language the user has selected.

This guide is mainly aimed at users who are familiar with programming for BeOS/Haiku but haven't used the layout APIs before. If you aren't already familiar with the basics of BeOS/Haiku programming, you should probably get acquainted with it before reading this guide.

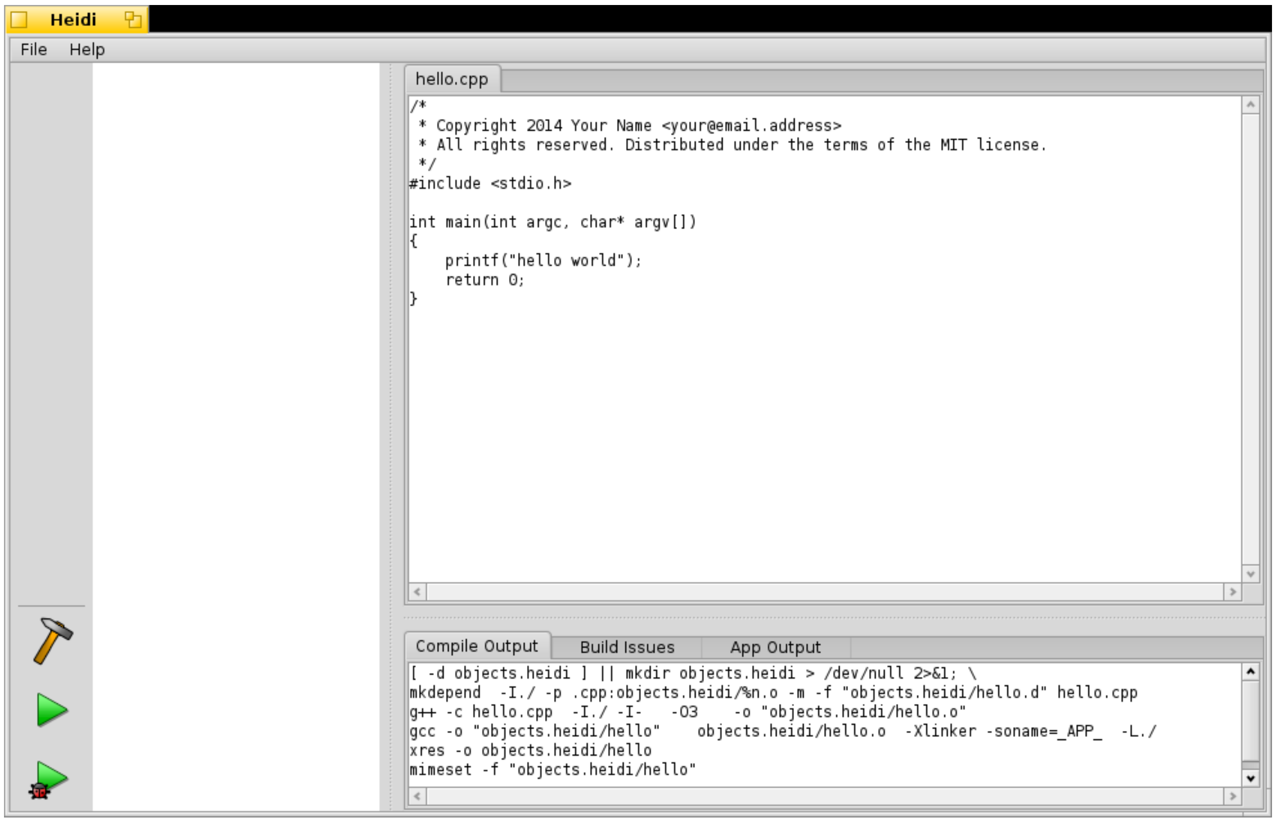
What Layout Is

The layout APIs are a set of classes that dynamically compute the optimal sizes for layouts. This means that they compute font size, string lengths, and window size to determine where a given window's controls should be on the screen.

There are 3 common types of layouts: grid, group, and split. Group layouts are simple horizontal or vertical containers for controls (or other layouts), split layouts are like group layouts, except the user can dynamically change the sizes of the different regions, and grid layouts should be self-explanatory.

Using the LayoutBuilder to Create a Window

Using the layout-builder to create a window is pretty straightforward. Let's consider an early version of Heidi's main window:

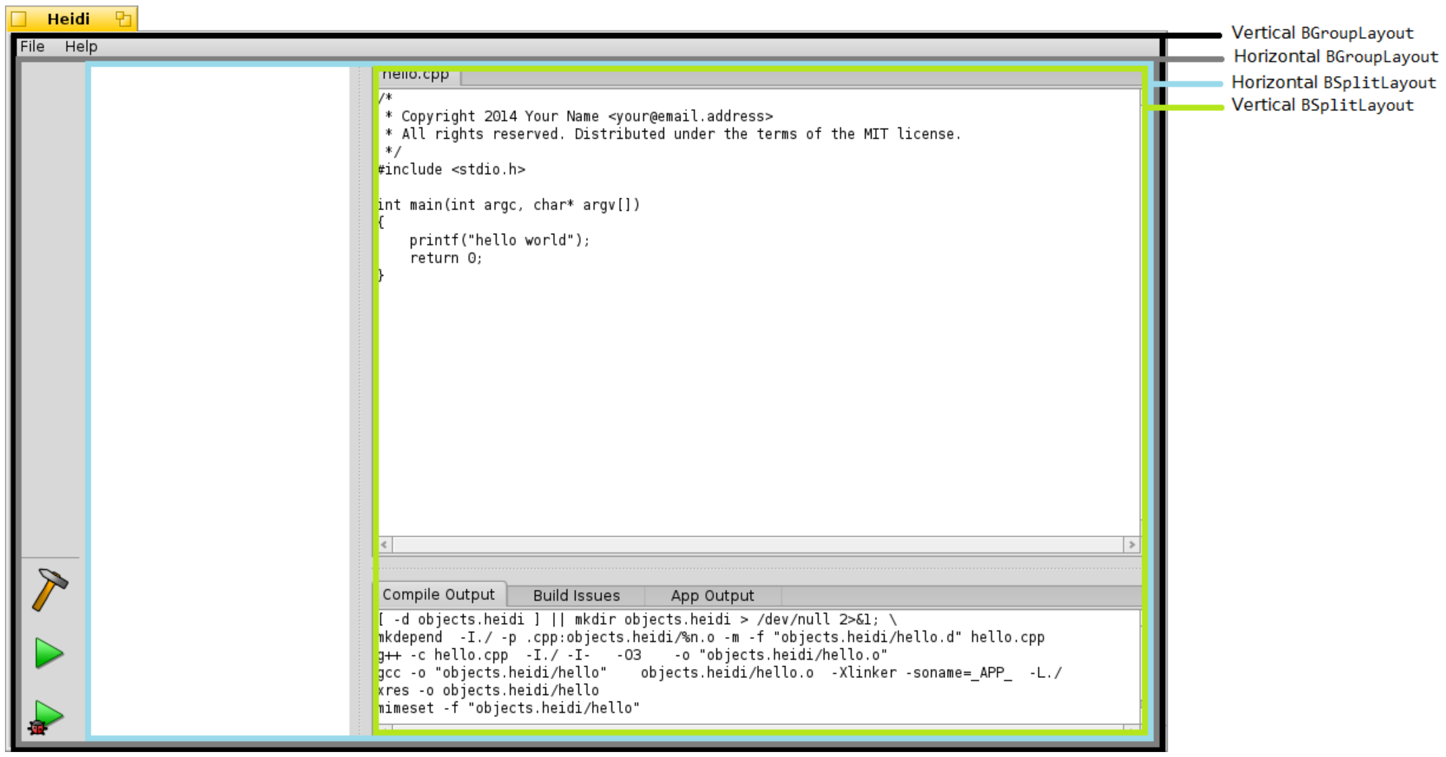


```
/*
 * Copyright 2014 Your Name <your@email.address>
 * All rights reserved. Distributed under the terms of the MIT license.
 */
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("hello world");
    return 0;
}
```

```
[ -d objects.heidi ] || mkdir objects.heidi > /dev/null 2>&1; \
mkdepend  -I./ -p .cpp:objects.heidi/%n.o -m -f "objects.heidi/hello.d" hello.cpp
g++ -c hello.cpp  -I./ -I-   -O3    -o "objects.heidi/hello.o"
gcc -o "objects.heidi/hello"    objects.heidi/hello.o  -Xlinker -soname=_APP_  -L./
xres -o objects.heidi/hello
mimeset -f "objects.heidi/hello"
```

This layout looks simple: a menu on top, a toolbar on the left, an empty outline list, a splitter, and then two tabviews. But in reality, it's much more complex than that:



```
/*
 * Copyright 2014 Your Name <your@email.address>
 * All rights reserved. Distributed under the terms of the MIT license.
 */
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("hello world");
    return 0;
}
```

```
[ -d objects.heidi ] || mkdir objects.heidi > /dev/null 2>&1; \
mkdepend  -I./ -p .cpp:objects.heidi/%n.o -m -f "objects.heidi/hello.d" hello.cpp
g++ -c hello.cpp  -I./ -I-   -O3    -o "objects.heidi/hello.o"
gcc -o "objects.heidi/hello"    objects.heidi/hello.o  -Xlinker -soname=_APP_  -L./
xres -o objects.heidi/hello
mimeset -f "objects.heidi/hello"
```

Additionally, the horizontal group layout and both splitter layouts are weighted -- this means that the layout code will give some views more screen space than others by default. To see how all of this was achieved, here's the code from the `BWindow`'s constructor:

```
// This is the root layout as seen in the diagram. The "0" passed as
// an argument here sets the spacing between the child views that this
// layout will hold to 0 pixels.
BLayoutBuilder::Group<>(this, B_VERTICAL, 0)
      // Insets are the "margins" of the layout. The root layout has
      // insets of "0" because the menu bar and other root controls need
      // to be right up against the window border.
      .SetInsets(0)

      // This is the main BMenuBar. It was created earlier in the constructor.
      .Add(menuBar)

      // This is the first child layout as seen in the diagram. Once again,
      // the "0" passed as an argument here sets the spacing between the child
      // views that this layout will hold to 0 pixels.
      .AddGroup(B_HORIZONTAL, 0)
            // This is the toolbar that appears on the left-hand side of the
            // window. It's a BToolbar created with orientation "B_VERTICAL".
            .Add(fToolbar)

            // This is the horizontal splitter as seen in the diagram. The
            // B_USE_HALF_ITEM_SPACING is one of the layout spacing constants
            // (it's currently defined as 4 pixels).
            .AddSplit(B_HORIZONTAL, B_USE_HALF_ITEM_SPACING)
            // The "1" passed here is the "weight" that this item has. You
            // can specify it for any item added to the layouts, but it's
            // not required. If you don't set it (like I did above) then
            // it defaults to "1".
            .AddSplit(B_VERTICAL, B_USE_HALF_ITEM_SPACING, 1)
                // This is the BOutlineListView as seen in the diagram.
                .Add(fProjectTree)
            .End()

            // Giving this splitter a weight of "3" and the one above a
            // weight of "1" means that this splitter will occupy 3/4 of
            // the available space, and the above splitter will occupy 1/4.
            // Though they are both splitters, the user can change that ratio
            // at any time.
            .AddSplit(B_VERTICAL, B_USE_HALF_ITEM_SPACING, 3)
                // Same thing as the above: the editors tab will get 3/4 of
                // available space, the outputs tab will get 1/4.
                .Add(fEditorsTabView, 3)
                .Add(fOutputsTabView, 1)
            .End()
        .End()
    .End();
```