Skinning XMMS with BuildImage and Skencil

Using the Python package "BuildImage" version 1.2 and its customized code for creating "skins" for the popular automatic music player

Terry Hancock

XMMS is a very nice program for playing music, but the default skin that comes with it is, well, "functional". Fortunately, though, the program uses the same skin files as WinAMP 2.0 (several other programs use these skins as well, which I'll call simply "AMP2 skins"). A "skin" is just a collection of images used to create the appearance of an application such as a music player (Figure 1).

In this case, the skin is simply a ZIP file with a funny extension ".wsz" (which presumably stands for "WinAMP Skin Zipfile"), containing a number of BMP format images with some very odd contents. There's not a lot of rhyme or reason to their layout, but the icons used by the program are extracted from them at runtime. Furthermore, since BMP doesn't support transparency, some very careful copy-and-paste work has to be done to fake transparency, as many AMP2 skins do. The results can be amazing, but unbelievably tedious to create manually.

Figure 1: XMMS unskinned (left) and with the default BuildImage Skin (right)

XMMS is a very nice program for playing music, but the default skin that comes with it is, well, "functional"

You can wade through this technical detail to create a skin using raster tools, or you can accept the limitations of using a commercial application to do the work for you, or — now — you can use BuildImage to generate a skin more sensibly from a drawing in Skencil.

What you need

Naturally, you'll have to make sure you have the software first. Most of the software is available in popular Linux distributions, so you can probably install using mostly binary packages. You will have to install BuildImage from the source, but this is easy, as it's a pure-python package. You'll need:

- Skencil (0.6.15): May still be called "Sketch" in your distribution (e.g. Debian, Red Hat).
- Python (2.2): You needed this to install Skencil, and BuildImage relies on it too.

- Python Imaging Library (1.1.3): You also needed this to install Skencil, so you probably already have it.
- Gimp (1.2): Strictly speaking, you can skip this, but it's useful for composing background images.
- BuildImage (1.2-beta): The AMP2 features are brand-new in this version, so don't try to use an older one. The examples used in this article are included in the distribution.

I'll leave you to the installation instructions for each package, but Skencil and BuildImage both use distutils, so installation is along the lines of:

```
tar zxf Package-version.tgz
cd Package-version
python ./setup.py build
# su to root
python ./setup.py install
```

Once the software is installed, you'll need to create a working directory for your skin project. The easiest way to do this is just to recursively copy from the example/AMP2 directory in the BuildImage distribution, which contains several examples, including the ones in this article, as well as the "default" skin template for BuildImage.
This is what you get:

```
samwise:~/tutor> tree
.
|-- bitmap
| `-- textdef.bmp
|-- scripts
| |-- __init__.py
| |-- animated.py
| |-- default.py
| `-- vanity.py
|-- setup.py
`-- sketch
 |-- amp2.sk
 |-- animated.sk
 |-- myth_bkg.jpg
 |-- myth_ws.jpg
 |-- vanity.sk
 `-- vanity2.sk
```

The file `amp2.sk` is the template for creating a skin drawing in Skencil. You'll rename it when you create a new drawing (such as with the other example files you see here). The `setup.py` file runs the `scripts` package, which contains scripts to build each of the examples. If you open one of these scripts, you can see that the AMP2 support in BuildImage consists of two functions `amp2_img()` which is simply a BuildImage script tailored to the provided `amp2.sk` template, and `amp2_skin()` which compiles the skin archive from those images.
The directory "bitmap" contains an original bitmap of the default XMMS font, which we will be using (you can do this part in Skencil but it's probably a waste of time).
At this point you can make the default skin, and that's probably a good way to make sure you have everything configured. You may have to set an environment variable to make sure that your Python can find Skencil. For example, on my system, I need to type:

```
setenv PYTHONPATH /usr/lib/sketch-0.6.15
```

Now, simply type the following to build the default skins (from the example directory, or your copy of it):

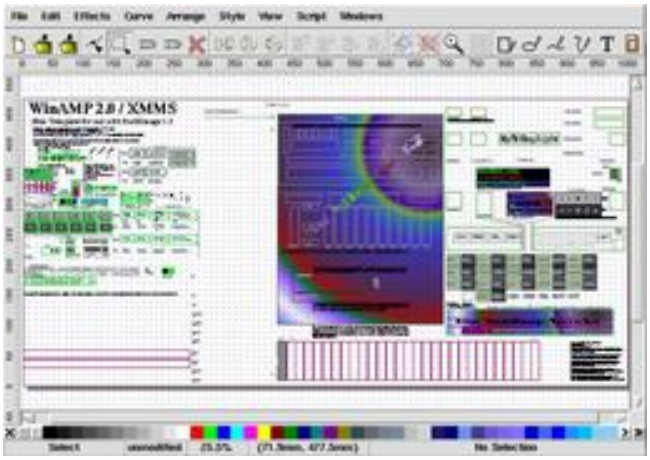
```
python ./setup.py
```

This will run the `scripts/__init__.py` file that calls BuildImage to create the skins. Like "make", BuildImage is very noisy, printing lots of messages to the screen. This is meant to make it easier to trace a problem when one occurs. You won't really have to worry about missing-font warnings — the important text in the drawings has been converted to curves to avoid portability problems. Only the text labeling in the template (which doesn't print) should be affected by font problems, and you'll still be able to read them, even if the font is not quite right. Also like compiling a package with "make", you can expect the build to put a fairly heavy load on your system.

> Altering the "skin" is one of the simplest ways of personalizing an application — either to make it fit better with an overall look-and-feel of the desktop (sometimes called a "theme"), or simply for fun

At the end, you'll find a series of named build directories — one for each drawing file. These contain all of the image components that were needed to make the skins as well as the actual skin files (`*.wsz`). You can now install the skin files into your `~/.xmms/Skins/` directory to make them available for you only, or switch to root and install them into the XMMS installation on your system (probably `/usr/share/xmms/Skins`), to make them available for all users. Yes, there's a lot of template in there. But

Figure 2: The BuildImage AMP2 template for skinning automatic music players



don't worry, you won't have to use all of it to get results. In fact, we can get a fair amount of mileage by just changing the background pictures. This is what I call a "vanity skin", and it goes a long way towards what most people are probably hoping to do with hardly any effort.

Now launch XMMS, and press the "O" options button and select "Skin Browser". You should find the new skins there (the names are defined in the script files and are usually variants on the name of the build directory). Select them, and see what the default skin looks like. The result should looks something like the right side of Figure 1.

So far, so good.

Simple "vanity" skin

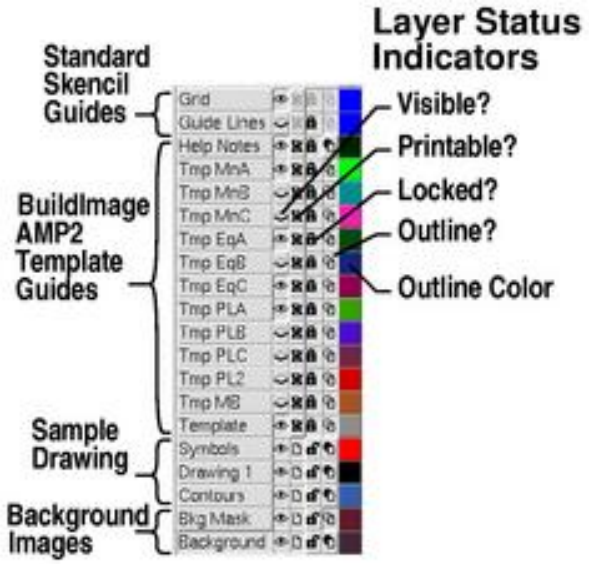
Naturally if you just wanted an off-the-shelf skin, you wouldn't be reading this article, so let's start changing things!

Take a deep breath, copy `amp2.sk` to something else (say `tutor.sk`), and open it in Skencil (Figure 2). *Don't Panic!*

First, use the "`Windows -> Layers`" option to bring up the Layers dialog box in Skencil (Figure 3). There are, as you can see, a lot of layers in this drawing. That's because the templates are prioritized to make things easier for you. Start with the "Background" layer, so make sure it's unlocked and select it. You probably want to make sure all of the other layers are locked so you don't move anything unintentionally. You will find you need to pay special attention to the layers menu throughout, as drawing on the wrong layer can give you confusing results — and remember that what "prints" (or appears in the BuildImage output) is different from what is "visible" in the Sketch editor window. So you also need to pay attention to what the settings are for each layer: will they print? are they visible in the editor? are they editable? and perhaps most confusingly, are they displayed as "outlines"? Experiment a little with these settings to make sure you understand what they do. Note that the drawing contains a key to the layers, right underneath the title block on the upper left side. The labels for each layer will show up whenever that layer is made visible.

One thing about the AMP2 template layers that you should understand is that they are displayed on multiple "outline" layers, both to color-code them, and to make it possible to remove the ones you don't need for a particular project. Obviously there's a lot you can do with skins for AMP2 players, and you probably don't really want to do all of those things at once. You can think of these layers as additional guidelines — by default, they are locked, so they won't interfere with your drawing (but you can use "`Arrange -> Snap to Objects`" to snap your drawings to them, which I highly recommend).

Figure 3: The layers dialog in Skencil, showing the layers in the AMP2 template



Under the Skin

A “skin” is a collection of image resources used to create the on-screen appearance of an application, and altering the “skin” is one of the simplest ways of personalizing an application — either to make it fit better with an overall look-and-feel of the desktop (sometimes called a “theme”), or simply for fun.

The skins for “automatic music players” described here were originally for WinAMP 2.0, and adopted by XMMS and other players, hence the Windows-centric design. The file is simply a ZIP file, containing BMP formatted image files. Components for on-screen widgets are derived by cropping out sections of these image files. If you are curious about the layout of the skin itself, I recommend unzipping one of the skins in the tutorial and examining the files you find.

There are also a couple of text files in INI format which specify color-scheme information for parts of the display that are not controlled by the image files (mainly the text part of the playlist window, but also the colors used for the visualization effects).

Manipulating these images to create the desired effect is tricky when working directly with the bitmap images, and that is what the BuildImage AMP2 support is meant to address. These skins were meant to be used with XMMS, since that’s what I test against, but they will probably work with WinAMP 2.0 as well. One difference worth mentioning, is the `mb.bmp` file which is for the “mini-browser” which some versions of WinAMP have and XMMS apparently does not. This file is supported by copying the similar data from the playlist skin, except for the mini-browser controls which are included in the template for completeness (look near the playlist controls on the right side).

Naturally, a vector drawing doesn’t really have a pixel size. So in preparing a template for conversion to raster images, a scale has to be assigned. In the AMP2 template, this scale is 1 mm = 1 pixel. So you should probably go to the “`File -> Options`” menu and make your default units “mm”, which will allow you to measure pixel sizes with the rulers and set your grid accordingly (in addition to snapping to the template objects, you can simply snap to this pixel grid, which can be convenient).

For a “vanity” skin, all you have to do is to change the backgrounds

For a “vanity” skin, all you have to do is to change the backgrounds. Click on the “image tool” in Skencil (far right of the toolbar) to browse for a good background image. Once you’ve got it, you can place it in the right place to show up on the combined XMMS main and equalizer windows (and a little bit of the Playlist, if you want). You may want to either duplicate that image or find another related one to place on the “Winshade” background (Figure 4).

The background images for the vanity.sk example were clipped in ImageMagick (I also could’ve used Gimp), to make them snap nicely into the backgrounds. However, you don’t have to use exact-fit images with the template. There is a “Background Mask” layer above the Background layer. This layer is simply an opaque white field with holes in it to show through to the background. As delivered, this layer is in outline mode, so you won’t see the clipping effect in the editor until you take it off of outline. This can be a real “gotcha” if you accidentally draw stuff on the background layer instead of in one of the upper layers and it doesn’t show through!

Figure 4: The background areas used in the “vanity” skin

Like an HTML file, a Skencil file stores relative links to the

images by default, so be aware that you need to keep the images in the right place. If you don't like this behavior, you can embed the images in the drawing file by selecting the image, then right-clicking, and selecting "embed image" from the resulting menu.

To get a good look at where the masked background regions are, you can just make the "Bkg Mask" layer visible and change the fill color (select and right click to get a menu including the fill option) — be sure to change it back. The regions include the combination Main+Equalizer+Playlist background, the Winshade background, and a special "Signature" region that appears on the bottom of the playlist when it is expanded.

The background for the playlist will be generated so that it matches with the equalizer and playlist when double-sized

Once you've changed the images, save the result. You'll then want to edit `scripts/__init__.py` to reflect the new name for the .sk file, and the filename of the target skin (minus the .wsz extension). Then do the same procedure as before to build and install the skins. This should give you a template something like the one in Figure 5.

Designing the controls

"Okay", you say, "that looks great. But the buttons don't look right anymore — I need something cooler still". No problem, all of the button designs are laid out in the template (Figure 6). In fact, you have three drawing layers here, for convenience: the actual "Drawing 1" layer which is reserved for your use; a "Symbols" layer with common symbols for the various buttons; and a "Contours" layer which contains drawn button backgrounds.

In the default template, the "Contours" layers is turned off, so you get a very background-heavy result with minimalist buttons. This is good for showing off the background in a vanity skin like the one just created, which is why it's the default. You can get a rather different looking skin simply by turning the contours on, in which case, you'll get contoured and shaded button drawings underneath the symbols for them (Figure 7).

Figure 5: The finished "vanity" skin

Alternatively, you may be dissatisfied with the default symbols provided, in which case you may want to alter the contents of the "Symbols" layer. You don't have to use this division of the drawing of course, I've simply provided these three layers for convenience. You can just as easily do all your work on the "Drawing 1" layer and turn the other two off. You may also want to note that if you have objects selected, you can "Move Selection to Here" by right clicking on the target layer in the Layers dialog.

Finally, you should note that the "Drawing 1" layer also contains the color palettes and gradients for the skin. These are simply boxes from which BuildImage derives the necessary color information which goes into the .txt control files in the skin, which determine the color scheme for the text in the playlist, the colors used by the visualization widget, and the font colors.

You will also find the drawings of the numbers used in the counter. One nice thing is that Skencil provides a ready-made "LCD text" object in the "`Edit -> Create`" menu. This can be a nice choice for the AMP2 counter

Vectors and Rasters: Source and Binary

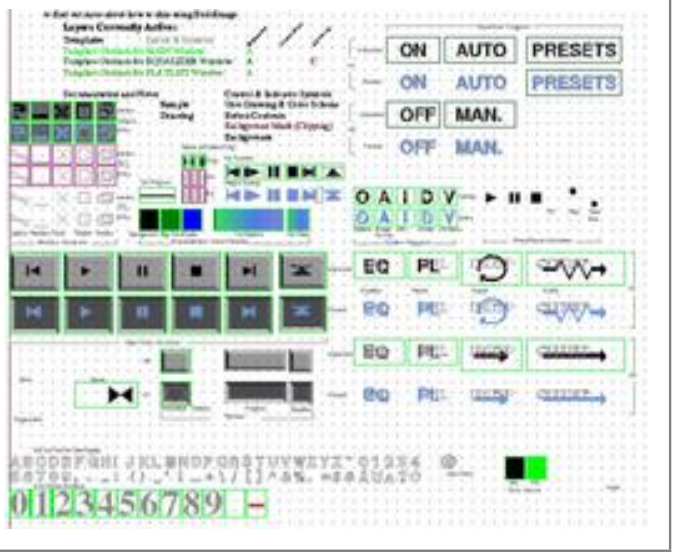
There are two broadly available ways of representing graphical information in digital form: so-called “raster” format (more often called, albeit incorrectly, “bitmap”) and “vector” format. Rasters are simply collections of picture elements (or “pixels”), usually aligned in a row-major grid format in the file. Rasters have the advantage of extreme simplicity for image processing and display, since they contain data in the format in which it will usually be presented on the screen or printed page.

By contrast, vector graphic files contain a conceptually simple representation of the graphic, as a description of the drawing processes used to create it. This adds additional overhead to displaying the image, since a complex procedure must be followed to translate the description into pixels for display. But it gains a terrific advantage of being much easier to manipulate and edit if the graphic should need to be altered.

This trade-off should sound familiar to programmers — it is the same relationship as exists between the source code for a program (analogous to the vector graphic) and the binary executable program that results from compiling that source (the raster image).

BuildImage attempts to follow this analogy a step further by automating the conversion process in a way that allows it to be mingled with build scripts such as those provided by the Python distutils or the much older “make” utility. Like “make”, BuildImage uses file dates and dependency information to decide what steps need to be repeated, to avoid having to repeat all of the steps in the build each time the script is run.

Figure 6: Most of the button and widget designs are collected on the upper left of the template



numbers. You can also simply use numbers from an appropriate text font. I recommend, however, that for any fonted text that you create for actual display in the resulting skin, you use the “`Curve -> Convert to Curve`” option to eliminate any dependency on the font file. This can save you a lot of headaches, as the environment that BuildImage runs in is not exactly the same as the one you use with the interactive editor, so it’s not guaranteed to find your installed fonts.

You can do the same to define the alphanumeric characters used by the data displays (used for the title of the song you are currently playing, for example). This is really tricky though, as those characters are only 5 x 6 pixels! The default bitmapped font is not anti-aliased, by the way, but the image is full-color, so there’s no reason why you can’t use anti-aliased characters to improve its appearance (whether or not the result is really an improvement is a matter of taste, though). In the examples, however, I have always used the `use_font` option in the scripts (see the next section for more on altering the BuildImage script files).

You will notice that there isn’t a drawing provided for every single possible button in the template. By default, in fact, many of the button template layers (specifically the “B” and “C” layers) are turned off. This is because in most cases you don’t need to define them, and the buttons they define will follow suit with the ones you have already drawn. For example, the window controls for the Equalizer and Playlist windows will, by default, simply copy the ones provided for the main window.

For some elements, such as the volume and balance slides, there is no drawing so that the background can simply show through. You actually have three ways to define these slides

Figure 7: A skin with (left) and without (right) the default "Contours" layer

if you want: you can draw them directly over the background where they will appear, or you can draw them in the provided "base" widgets provided in the template, or you can animate them, as will be shown later.

This is achieved by simply layering the images transparently so that the dependent images are layered on top of the base images. If they have no content (they are simply pure-white which BuildImage interprets as "clear" by default), then the base image will show through. If you want more control over these additional widgets, just make the "B" layers for each component visible, and you can draw them as well. Ignore the "C" layer for now (I'll come back to it).

Once again, when you're satisfied with the template, build and install the skin to test it. Note that you can go through this cycle as many times as you need. Normally, BuildImage will correctly figure out which steps it needs to repeat, but if you want to force it to do a complete rebuild, you can do so by recursively deleting the build directory for the skin you want to rebuild (`rm -rf builddir`).

At this point, you can already make a very nice basic skin.

Special effects

You may have noticed that some skins seem to alter the behavior of the controls on the player. This isn't really true – the logical behavior of the player is unchanged. However, for most of the buttons, there are multiple button states that can be displayed. For example: for when the button is being pushed, or when it is toggled to "on" or "off" states.

With skinning, you can control what each of these states looks like, even down to making them look invisible – and that's what creates the impression of differing behavior.

For example, the AMP2 players display either "mono" or "stereo" indicators, according to the audio file they are playing. To me, this seemed a bit redundant, and took space away from my background. So, in the default skin, the mono indicators are both transparent, showing the background regardless of whether "mono" is set, and the "stereo" indicator only displays an iconic "stereo" symbol when the track is stereo. That works for me, as I'm happy to assume that a track is "mono" if it's not stereo.

You can also create various appearances based on the four states provided for many of the toggle buttons: there is an "off" and "on" state, but also "off-pressed" and "on-pressed" states to define what happens when you click on them. Likewise, the sliders have pressed and unpressed states to alter their appearance when you are actually using them. Whether you define these alternate states will affect the visual behavior of the skin, despite the fact that the same things are going on logically inside the program.

Now AMP2 skins are composed of bmp format files, which don't support transparency, so all of the "transparency" we have is faked by copying the background to the image that will be pasted over it. This is how BuildImage manages the elements, which you define with transparency (or pure-white color – like the basic page color).

> There are two broadly available ways of representing graphical information in digital form: so-called "raster" format (more often called, albeit incorrectly, "bitmap") and "vector" format

This works most of the time, but there are places where it doesn't – specifically any place where the widget in question has to move in front of the background. In some places, that's just tough cookies, you'll have to use a bit of artistic sense to solve those problems.

However, AMP2 skinners long ago discovered that WinAMP, when confronted with a bmp file of the wrong size, will fail silently in the event that the needed section of the image is not actually defined (i.e. it just won't blit that data). This may once have been a bug, but it turns out

Making raster backgrounds with Gimp

Although vector graphics are very useful in designing skins, you will probably find it desirable to compose raster background images with a program such as Gimp. Ideally, you would want to design these elements to snap exactly into their places in the template, and this is not hard to do.

All of the background elements are 275 pixels wide, and the main, equalizer and (minimal) playlist windows are each 116 pixels high. You can compose a separate image for each of these of course, but it is more often the case that the main and equalizer backgrounds are composed as a single image, which will appear complete when the two are "docked" on the desktop. This gives an image size of 275 by 232. If you add the minimal playlist (the integrated playlist background will only show up in a few places due to all of the tiled areas of the playlist border that allow it to be resized), you will make this 275 by 348.

Similarly, when the three elements are minimized to their "winshade" form, they can have a different background, this time 275 by 14 for each element, or 275 by 42 total.

You probably should actually design your Gimp image at an integral multiple of these sizes (say 550 by 696) instead of exactly matching pixel-for-pixel. The final result then, would be down-sampled and anti-aliased, possibly resulting in a slightly higher quality image (depending on the original).

Once you create these elements, use the "image" tool in Skencil to place the images in the right place on the template drawing, and proceed, just as I have described, for pure-vector originals.

to be useful and XMMS emulates it. Skins that take advantage of this, cut off parts of various image files in the skin to do things like make the volume and balance sliders disappear, or make the position bar narrower. BuildImage supports this through options to the amp2_skin() function in the build script.

Specifically, here's what you can do with the options to amp2_skin:

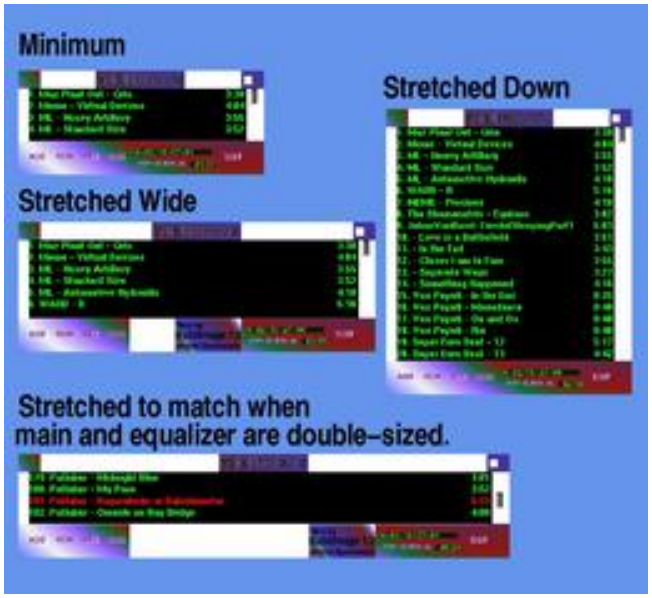
- Turn off `show_volume_sliders` or `show_balance_sliders` (set them to 0 or False) to trim off the volume/balance sliders.
- Turn off `show_equalizer_sliders` to eliminate the equalizer slides entirely (there is no way to remove only the slider widget, it's all-or-nothing – this is due to the way the image file is laid out).
- Alter `position_bar_height`. The default size is 10, but you can make it smaller by setting the desired size here.
- The `use_font` option tells amp2_skin to use the bitmap font specified for `text.bmp`, regardless of whether you draw text in your template or not. The colors will be altered by looking at the "text fg" and "text bg" colors in your drawing (black pixels will be rendered in the foreground color, white in the background, and grey pixels will be mapped to a simple gradient between them).
- We'll use `playlist4double` in the next section, but if true, it defines the playlist background based on the assumption that XMMS will normally be displayed "double-sized".

There's also an option `author_readme` to put some text into your skin (possibly license terms, your name, etc). This will appear in a `README` file within the archive. You can also specify a build target name with `tgt_dir` (you'd need to pass the same name to `amp2_img` as well), which can allow you to use more than one source drawing to create a single skin. We won't be using these options.

Normally, BuildImage will correctly figure out which steps it needs to repeat, but if you want to force it to do a complete rebuild, you can do so by recursively deleting the build directory for the skin you want to rebuild

These options have already been put to use in the example scripts, and this is where the variant skins based on each

Figure 8: Typical sizes for the playlist window, showing the tiling behavior



template come from. Obviously, you'll want to tweak these options to suit your own skins as you design them.

Getting the playlist right

Because the main and equalizer windows have a constant size, they are relatively easy to skin. Even when double-sized, the skin is simply double-sized with them. However, the playlist is a more conventional, resizable window, and therefore the skinning concept gets stretched a little bit. Or rather, "tiled", which is how the program handles the stretchy bits of the playlist window (Figure 8).

Getting this to look right in your skin can be a challenge. There are three basic possibilities you can try.

Setting up the playlist to look right when docked at minimum size

Although the playlist is stretchy, it will minimize to the same size as the main and equalizer menus. One approach to designing it, then, is to make it look right when it's at this size, and just ignore the tiling effects. This works okay for the sections of the playlist that are defined as static elements: the center and both ends of the top bar and the left and right sides of the bottom bar (which meet when the playlist is minimized). This is the first behavior that BuildImage supports, by using portions of the integrated background to create these parts of the playlist skin.

Even in this configuration, however, the top and sides of the window are tiled, so there is no way to have the background wrap completely around the frame. BuildImage copes with this situation by generating a gradient, either horizontally (for the vertical tiling elements) or vertically (for the horizontal tiling elements), which is derived by averaging the background over the tiled region in the minimized state. This generally produces a tolerable, if not exactly ideal default. You probably will want to override this behavior by creating your own tiling elements.

All of these elements are laid out in a (hopefully) logical arrangement in the upper right hand corner of the template – around the color scheme drawing for the text that appears within the playlist, as you will notice. You will definitely want to use the labels (on the "Template" layer) and help (on the "Help Notes" layer), when working with the playlist for the first time.

You may notice that there are some elements of the playlist layout that do not show up in the minimized configuration – these regions, including the "signature" area are only displayed when the playlist is enlarged.

Setting up the playlist to look right when at minimum docked size with the main and equalizer windows double-sized

On large screens, an AMP window can start to look pretty tiny, which is why it conveniently provides a double-size mode. The playlist is unaffected by double-size mode, however, and will appear normal-sized. You can expand it horizontally to match, so there is also a "minimized double-size playlist" configuration. However, this screws up the alignment with the playlist if it was designed as in the past section. BuildImage's amp2_skin function provides a means to make the skin look best in this configuration.

In the amp2.sk template, turn on the "PL4 Dbl Sz" ("Playlist for double-size") layer. This now shows the correct layout for this alternate configuration. You will, unfortunately, not be able to use the "Bkg Mask" layer to handle this part of the background (there's a cut-out in the middle). It also currently interferes with defining a graphical equalizer graph, a template design bug, which will probably be fixed in a later version of BuildImage. You will also need to pass the "playlist4double" option to both `amp2_img` and `amp2_skin` functions in your build script.

Now, the background for the playlist will be generated so that it matches with the equalizer and playlist when double-sized. You may note that some of the variations of the examples include this double-size option. The results are not quite seamless, since the playlist will still be at twice the resolution of the equalizer, so there is still a visual line created at the interface. But the results are pretty satisfactory, and of course, in this configuration, your "signature" will be visible.

Setting up the playlist to look right at any size

The best practice, though, and the one the designers probably intended, is to make the playlist look good at any size, by changing your design sense from simply showing the background, to artistically complementing the other two elements. Needless to say, this is a greater aesthetic challenge. I'll have to mostly leave this to your imagination, but there are a few technical points to observe about the way the playlist window resizes.

Tiling zones

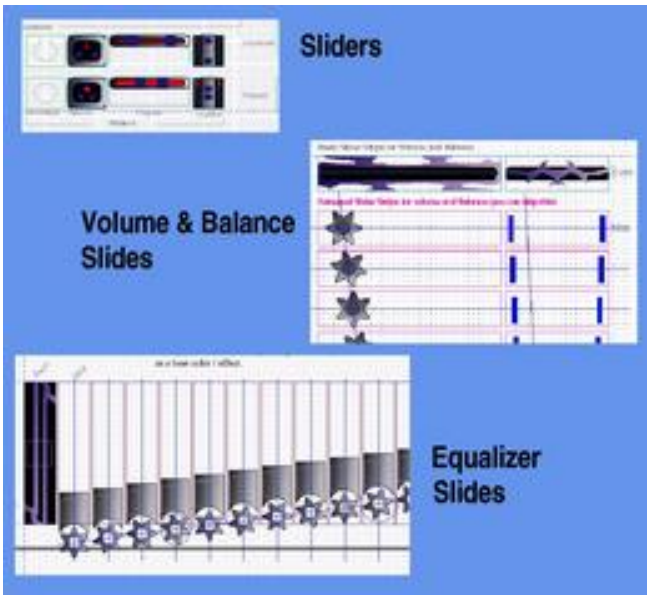
The sides of the playlist window are designed to tile as the window is resized. Careful observation of the behavior of the window, in fact, shows that it "snaps" to these tiled points. So, it's very unlikely that you will have to deal with split tiles, except in two particular cases – both when the playlist is docked. At the minimum sizes, the window has an extra snap position, this time with the tile patterns on the top at about 2-1/2 tiles on either side. That means that the horizontal tiles should be designed to look okay if snapped at the halfway point.

Now the easiest way to deal with this is to punt and just use a vertical gradient pattern (effectively, a one-pixel-wide tile) on the horizontal tiles, and a horizontal one on the vertical tiles. That does work, and many skins use it – BuildImage does this by default. It produces a nice, smooth window appearance. This may not be appropriate for your design, however (you might want more of an ornate picture frame look, for example). You'll have to consider what these tiles look like together, and how they interface with the center and corner elements of the playlist (and in the center, how they interface in full- and half-tiled cases).

Animation

Each of the volume, balance, and equalizer slides has a 28-step animation sequence defined in the skin file. BuildImage removes this complexity by copying the "base" for each slide over each frame of animation. But clearly, you may want to define them separately. You'll find these are defined on the "Main C" and "Equalizer C" template layers in the AMP2 template (Figure 9).

Figure 9: In considering the animated sliders, you will need to work with the animation sequences for the volume, balance, and equalizer slides, as well as the slider widgets for all of them

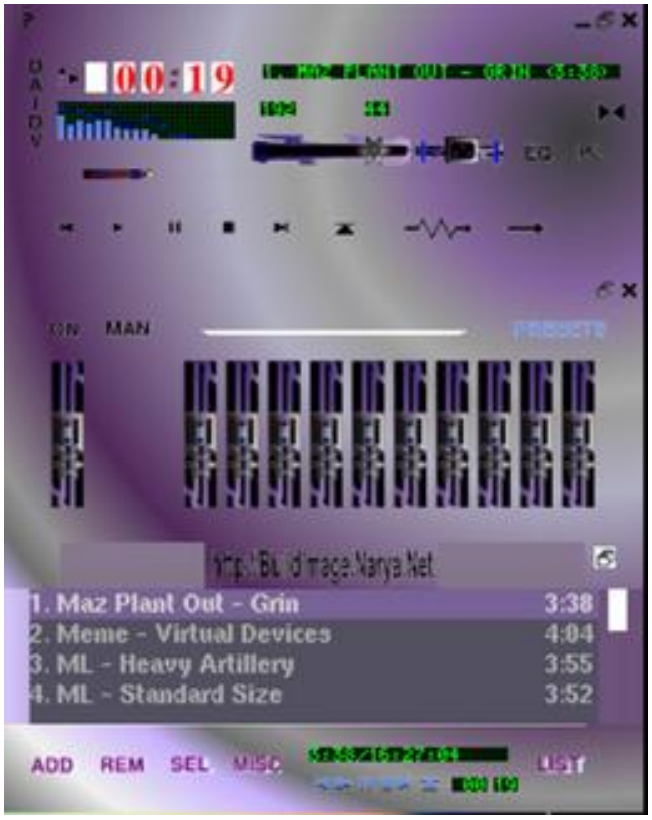


For this, I'll use a new example skin (animated.sk), which shows some of the things you can do with animation effects. Quite frequently, you'll want to use the options to amp2_skin for eliminating the volume and/or balance sliders, so as to provide a clear spot for your animation effects (you may want to put an animated volume slider in that is no longer constrained to be rectangular or of the given size, for example).

There is no way to do this for the equalizer, however, as you will recall that if you have the animated slides at all, you also have the slider widget. The only way to deal with this is to make the slide background so that the widget disappears against it, or to incorporate the design of the widget into your animation effect (which is what I've done with the example).

One thing I find a bit illogical is that the balance slider doesn't go from "full left" to "full right", but rather from "centered" to "fully deflected" (left or right). That seems like the wrong decision to me, but it's determined by the

Figure 10: The finished animated skin



program, so you can't really change it with your skin. You'll have to keep this in mind in your design. In the example, I've left the balance slider in, partly because of this issue.

Drawing smooth animation frames in Skencil requires a bit of trickery. If you look at the guidelines layer in the example template, you can see that I've added a line of motion with circles marked on it to snap the animated "gears" into. Combined with judicious use of the "`Arrange->Align`" dialog and/or the various "Snap" options, this can make drawing the animated elements much easier.

After you build the animated example, you should see something like Figure 10.

That's it!

You're now ready to exercise your creative energy on new skins for XMMS. I've pretty much covered everything the BuildImage AMP2 scripts support, and that allows you to do just about everything the skin format can do. Remember, if you do find extra tricks, you can always insert additional processing between the `amp2_img` and `amp2_skin` calls, or after the `amp2_skin` call (you will likely use the Python Imaging Library for this. You may want to examine the source code for the `amp2_skin` function for clues on how to get it to do what you want). I would really appreciate finding out about any such tricks, of course, since they're likely candidates for inclusion into a later version of BuildImage.

The `amp2_img` and `amp2_skin` scripts can also be viewed as examples in themselves, for specialized uses of BuildImage for skinning application programs. They aren't really traditionally designed functions, but rather BuildImage build scripts encapsulated into functions, so they have something of a bricolage design sense. I'd certainly be interested in seeing the same concepts applied to other programs.

Sources

WinAMP (http://www.winamp.com)
XMMS (http://www.xmms.org)
BuildImage (http://buildimage.narya.net)
Skencil (http://www.skencil.org)
Python (http://www.python.org)
Python Imaging Library (http://www.pythonware.com/products/pil/)
Gimp (http://www.gimp.org)

Copyright information

© 2005 by Terry Hancock
(The following license is effective immediately)
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is available at http://www.gnu.org/copyleft/fdl.html

About the author

Terry Hancock is co-owner and technical officer of Anansi Spaceworks (http://www.anansispaceworks.com), dedicated to the application of free software methods to the development of space.