

Skinux Technical Overview



**Achieving Dynamic, Realistic Interfaces with
Procedurally Driven Imaging Effects**

December 2002

Table of Contents

Why Skin User Interfaces? _____	1
Media: From Television to Computers _____	1
Computers: From Productivity to Entertainment _____	1
Software: From Graphic User Interfaces to Skin User Interfaces _____	2
Enabling Factors: Technology and Cost _____	2
Skin User Interface as a New Advertising Medium _____	4
Benefits of the Skinux SUI _____	5
Branding Facilitation _____	5
Productivity _____	6
Performance _____	7
Applications of the Skinux SUI _____	8
Entertainment/Media _____	8
Emulators and Simulators _____	9
Control Panels/Launchers _____	11
Games _____	11
Embedded Devices _____	12
Technical Approach: Skinux vs. Traditional API _____	13
Filling the Gap: Technical Advantages of the Skinux System _____	13
Cross Platform _____	14
Interoperability _____	15
Extensibility _____	15
The Skinux Difference (Technical Features) _____	18
Sophisticated Imaging Model _____	18
Scalable Imaging _____	19
Built-in Animation Model _____	24
Layered Windows _____	27
The Skinux Architecture _____	28
Skinux Development Kit (SDK) _____	29
Skinux XML Language _____	31
MMX Optimized Imaging Library _____	33
Industry Experience _____	36
Contact Information _____	37

Why Skin User Interfaces?

Media: From Television to Computers

In 1964, when Marshall McLuhan wrote his revolutionary expose about automation and television, *Understanding Media*¹, television was the dominant, driving force in worldwide entertainment, advertising, and the dissemination of information. Today, the desktop computer connected to the Internet is fast becoming the medium of choice for entertainment, advertising, broadcasting information, and communication. Due to significant strides in computer and Internet technology, people are beginning to spend more time in front of their computers than the TV, sampling music and film clips, downloading media and software, buying products, and communicating with each other in e-mail and IRC chat.

Computers: From Productivity to Entertainment

Over time, advances in technology and the falling price of sophisticated computer hardware have changed the way we use computers. The rich presentation of the Web, advances in high-speed Internet connectivity, development of streaming media, and the decreasing cost of increasingly robust hardware have all transformed static computer workstations into dynamic windows to the world. As a result, average family members now use desktop computers for entertainment and communication, rather than for productivity purposes, alone.

Only a decade ago, the “killer applications” for personal computers included word processors, spreadsheets, and page layout programs, all of which were designed for conducting business at work or at home. Today, media players, file sharing services and instant messengers have become the dominant applications, consuming more and more of people’s spare time. Twenty years ago, parents complained that they couldn’t get their teenagers off the phone when they needed to make an important call. Today, parents complain that they can’t check their e-mail because their teenagers are hogging the family computer, spending endless hours in chat rooms or surfing the Web for music and video files.

¹ Marshall McLuhan, *Understanding Media*, McGraw-Hill, New York, NY, 1964.

Software: From Graphic User Interfaces to Skin User Interfaces

The Graphical User Interface (GUI) was invented back when computers were used to do a job. They preceded the time when computers were used for entertainment and communication and were designed with productivity applications in mind. The main intent of GUI systems was to enforce standard user interface guidelines that led to consistent user behavior across applications and emphasized the branding of the user's operating system.

However, with the advent of the Web and the increased focus on entertainment, consumers and the software companies that sell to them are demanding user interfaces that transcend the traditional GUI. They want a designed appearance that distinguishes the consumer or the vendor, rather than the native computer operating system. In the same way that they choose their clothes, consumers want the "look and feel" of their software to reflect their age group, tastes, and even their cultural outlook on life.

The Skin User Interface (SUI) fills this need in the marketplace. It uses imaging technology, which includes layering and transparency and enables such visual features as drop shadows, glowing and pulsating buttons, and other features that are far more interesting, entertaining, and interactive than the static graphics of the GUI to encode and send its message to the user.

Enabling Factors: Technology and Cost

New imaging technology, increased memory capacity and faster computers are making Skin User Interfaces possible in today's computer marketplace. Furthermore, we have seen incredible reductions in the cost of high-quality computer hardware during the past five years. These factors have enabled more robust interfaces, resulting in a paradigm shift from GUI to SUI that is similar to the transition from character-based interfaces to graphical windowing interfaces in the early 1980s.

The Skin User Interface is being driven today by three converging trends in computer hardware development:

- 24-bit video display cards
- The Intel MMX processor
- The low cost of Random Access Memory (RAM)

Video Display Cards

Even the most inexpensive personal computer sold today is equipped with a fast 32-bit video card that is capable of displaying photographic quality images at high resolutions. At the minimum, these cards come with 16MB of SDRAM with a maximum resolution of up to 1600 x 1200 in either 32-bit true color mode or 16-bit high color mode. As a result, users are no longer restricted to indexed mode displays. Yet, the majority of Microsoft Windows® and Linux® based programs still utilize a 16-color palette to draw the user interface, using only a fraction of what is available to the user. Conversely, Skin User Interface programs leverage the display power that average computers provide for consumers at no extra cost.

The Intel MMX Processor

All Intel processors shipped since the introduction of the Pentium II in 1996 support the MMX instruction set. This technology is based on a single instruction multiple data (SIMD) form of parallel processing designed to improve the performance of image processing and graphics. The MMX is the most significant enhancement to the Intel architecture since the Intel386. It includes new instructions and data types that exploit the parallelism inherent in imaging algorithms.

By using the MMX instruction set, specific image processing operations such as alpha blending and blurring can be made to perform 2-to-4 times faster than they do on conventional Intel processors that run at comparable clock speeds. Although MMX is not a feature generally found in chip sets that compete with Intel, competing processors such as the Power PC have similar SIMD acceleration instructions that can display Skin User Interfaces at no extra cost to the user.

Low Cost of Random Access Memory

The cost of RAM has dropped significantly in the last few years, providing computer users with more memory than ever before to store and manipulate image data. This, too, makes Skin User Interfaces available to the average user.

Summary

Today, the standard PC that sells for under \$1,000 is equipped with a high performance 24 bit graphics card, a 2 GHz Intel Pentium IV with MMX technology, and 256 MB of RAM. The Skin User Interface developed by Skinux makes full use of the power of such a system to deliver a robust, unique software interface that attracts users, makes software more intuitive to use, and encourages interactivity.

Skin User Interface as a New Advertising Medium

Software and Media Content Deflation

One of the major consequences of the Internet revolution has been software and media content price deflation. In the 1980s, software vendors could easily charge several hundred dollars for a single software title. Today, users want to “try out” software programs for free, and they don’t always buy what they try. When they do purchase software, users are no longer willing to pay a lot of money for it.

This deflation has been driven by the following factors:

- The open-source movement of software development and the resulting availability of high-quality “free” software.
- The trend on the part of software development companies to bundle several separate applications into one software product, in order to increase value and decrease cost.
- The emerging popularity of file sharing systems, such as Napster® and Kazaa®, which enable content and software swapping without regard to copyright.

Because of this deflation, software and media developers are beginning to turn to alternate sources of revenue, such as advertising.

SUI: Offsetting Lost Profits with a New Advertising Model

To help offset diminishing revenues caused by software and media content deflation, businesses are beginning to use new advertising models to subsidize the cost of software and media content. Companies such as Google® and Yahoo® generate revenue with banner advertising in much the same way that billboards are used along highways.

Because Skinux produces a visually rich, photo-realistic interface, it can model billboard advertising, thereby reinforcing brand awareness at the interface level. For example, you may very well see a “Coke” or “Toyota” interface on your digital VCR when viewing the World’s Soccer Cup or when playing an MP3 on your media player program. Skinux refers to this technology as “Channel Branding,” and our company plans to file a provisional patent on its use.

Benefits of the Skinux SUI

Branding Facilitation

The bottom line is that Skin User Interfaces facilitate company and product branding by offering a more intense, wider range of visual communication than their traditional GUI counterparts. This is critical if you want your program's interface to communicate company and/or product brand information in addition to performing a specific function.

Although branding increases profitability for all products, it is more vital in entertainment and consumer appliance environments than it is in the traditional productivity software suites that once dominated desktop computing.

Visually Compelling

Skinux interfaces are visually compelling. For this reason, consumers are naturally attracted to them. Skinux Skin User Interfaces, such as the one we created for X3D Technologies (pictured below), are anti-aliased, photo-realistic, animated, translucent, non-rectangular, and 3-dimensional in appearance. By using Skinux, you can develop a user interface that will truly captivate your customers and communicate your brand—that's an essential ingredient for success if you are developing entertainment-oriented software and competing for the consumer's somewhat short attention span.



X3D's TV Gateway (Skin User Interface by Skinux)

Skinux enables designers to create striking interfaces by giving them pixel-level artistic control over the entire design. This is the same difference in control that can

be found between bitmap image editing programs, such as Adobe PhotoShop® and vector drawing programs, such as Corel Draw®.

Customer Specificity

One of the main uses of Skin User Interface technology is to adapt the user interface to the esthetic preferences of specific customers.

This has actually been the trend in all consumer appliances since the beginning of time, including cars, cameras, radios and toasters. It is only at the beginning of the technological lifecycle that products have a uniform appearance. This is true for software produced today as it was for cars in the 1920s, when Henry Ford joked that you can buy a Ford in any color as long as it was black. Customer-specific user interfaces are becoming the norm, rather than the exception in software design; in some markets, consumers already expect them.

Vendor Specificity

In addition to providing customer-identification, Skin User Interfaces can be used to reinforce a vendor's brand by providing a unique look. The best example of this is Apple's use of skinning in its new OS/X Aqua user interface. Its smooth, semi-transparent appearance quickly identifies the machine on which it resides as "a Mac."

Because of their branding ability, it is very likely that Skin User Interfaces will become a new advertising medium, not unlike a billboard along a highway. In fact, a new business model is being developed to pay for software with advertising dollars. The pioneers in this effort include products like Qualcomm's Eudora® e-mail system and AOL Instant Messenger, which actually display HTML advertisements in a corner window of the programs.

Productivity

The primary benefit of Skinux over other skinning technologies is its remarkable productivity.

Our most striking contribution to companies that use our technology and services is a dramatic decrease in the time it takes to create a Skin User Interface and get it to market. The Skinux tool set enables programmers to develop a Skin User Interface very rapidly, a process that would otherwise take months or years to implement by using conventional tools.

The Skinux tool set also provides plugins for Adobe Photoshop that export artwork created in Photoshop to the Skinux format. Because of this feature, graphic designers

do not need to buy and learn an additional image creation tool to design unique product interfaces.

Finally, all Skinux Skin User Interfaces are specified in standard XML with XSLT support. This feature provides for a separation between the code and the actual front-end design, which enables developers and designers to work with each other independently, in parallel. Skinux XML is extensible, a feature that enables developers to tailor software to the functional and aesthetic demands of specific customers.

Performance

Skinux is based on an extremely fast, highly optimized imaging library that can produce real time alpha blending and image processing procedural effects. More than 50% of our imaging library is written in Intel MMX Assembler code. In this way, Skinux is engineered to support the demands of a high performance environment.

Skin representations in Skinux are extremely compact and very little memory is required to encode them. As a result, they are ideally suited for low bandwidth environments, such as wireless Internet appliances or consumer entertainment devices. This is possible because Skinux is based on a procedural imaging model, rather than a simple bitmap representation. Skins can be downloaded very quickly and rendered on the fly.

Applications of the Skinux SUI

Skin User Interfaces are best suited for entertainment and consumer software, such as media players and control panels. These applications represent an emerging market and are likely to eclipse traditional productivity applications such as word processors and spreadsheets during the next ten years. The following sections describe the primary uses of skinning technology today.

Entertainment/Media

The most successful use of Skin User Interface technology to date has been in the arena of media player technology. Real Networks' Real Jukebox® and NullSoft's WinAmp® are high-profile media players that have "pushed the envelope" on customizable skins and one-of-a-kind interface designs.

Media players have been pushed to the forefront of software development because of the widespread acceptance of the MP3 (Motion Picture Experts) music file format and the popularity of file sharing systems such as Napster. With the exception of browsers, media players are probably the most widely used applications on the Web today.

By combining procedural imaging with layered image compositing, Skinux technology dramatically increases the level of realism and design previously offered by Skin User Interfaces. This is exemplified in the Skinux media player pictured below, which actually runs on top of the Microsoft Media Player®; it is available free of charge on the Skinux Web site, www.skinux.com.



Skinux's Skinamp Media Player

Although media players were the first applications to use SUI technology effectively, they are certainly not the last. Skin User Interface technology can enhance the usability and marketability of a multitude of desktop entertainment programs, from customizable chat clients to music mixing software.

Case Study

In July of 2002, X3D Technologies contacted Skinux to implement a Skin User Interface for its TV Gateway® and PC Gateway® media players. Their previous interfaces were implemented with Microsoft Foundation Classes (MFC)®. The marketing team at X3D felt that a Skin User Interface was necessary to give these products a competitive edge in a crowded entertainment software marketplace.

In 3 months, Skinux was able to conceive and implement the product designs for both products. In October 2002, X3D won Best of Show at Internet World. Within 3 months, X3D sold more than 150,000 copies of the software.



X3D's PC Gateway (Skin User Interface by Skinux)

Emulators and Simulators

Skin User Interfaces can also be used for significant advancements in emulation and simulation software.

To test their designs, hardware manufacturers such as cell phone companies often build mock prototypes that use software to simulate hardware functionality. Third-party software developers use simulation and emulation programs for testing, as well.

By using Skinux, a photo-realistic interface of a hardware device (shown below) can be created and connected to a back end program that emulates the device's behavior.

This can be accomplished within hours with only an image editor such as Adobe PhotoShop, a version of Windows Notepad® for editing the skin's XML, and a copy of the Skinux Development Kit™.



Skinux Hardware Prototype of a Phone Media Player

Control Panels/Launchers

Skinux technology can also be applied very effectively to the creation of control panels and launchers. This includes control panels for scanners and point of sale terminals, as well as software installers/launchers (pictured below). Software and hardware manufacturers are beginning to turn to SUI technology to distinguish their products from their competitors. Skinux is already prepared to provide whatever is necessary to capitalize on this trend.



X3D Game Installer/Launcher

Games

One of the most promising applications for Skinux's Skin User Interface technology is the gaming industry. Many games on the market today are well suited to this new technology, including desktop accessories such as Skinux's Solitaire, shown below (part of the SkinToys™ product line available for free download at the Skinux Web site, www.skinux.com).



Skinux's Solitaire

We are certain that the enhanced photo-realistic effects that are now possible with Skinux's SUI technology can help online casinos and other applications in the online gaming industry increase consumer participation and profits.

Embedded Devices

Skinux technology can also be applied very effectively to embedded Linux consumer device interfaces. X/Windows, the native windowing system for Linux, has little imaging support for Skin User Interfaces. As Linux becomes more popular in the embedded space, Skinux is well poised to be the technology of choice for developing embedded interfaces for those systems.

What makes Skinux so attractive to Linux developers is its access to the source code and the small footprint size of the Skinux engine. The graphic below shows an early version of a Skinux SkinToys application running under Linux on a Compaq Ipaq® system. The hardware configuration includes a 200MHz K6 processor with 32 MB of RAM and a 16 MB Flash memory card.



Skinux SkinToys™ Application on the Compaq Ipaq running Linux

Technical Approach: Skinux vs. Traditional API

Filling the Gap: Technical Advantages of the Skinux System

There are significant advantages to licensing Skinux technology and contracting its services, rather than using a traditional API such as Windows' Win32 or MFC or Linux's GTK+ to develop an interface that merely resembles a skin. For one thing, traditional window systems and their APIs provide little imaging support from which to build such an interface. Therefore, you'd have to write a large volume of code to support such functionality before you could focus on your core competencies.

The support structure needed to build a Skin User interface includes the following:

- A layered image compositing engine
- An MMX optimized imaging library for fast image processing
- Alpha blending at sub-pixel positions for smooth animation
- Anti-aliased rendering of text and shapes
- Procedural imaging model for filter effects
- An animation model for modulating controls over time
- A control library built on a sophisticated imaging model
- A hierarchical pivot coordinate system for rendering imaging controls
- XML architecture for describing Skin User Interface controls as resources
- XSLT architecture for condensing XML skin descriptions
- Alpha blending onto the desktop using layered windows
- An authoring tool for creating 32-bit RGBA images

At this point in time, native operating systems provide very little of this support structure. The balance of APIs for both Microsoft Windows and Linux are too low-level, and those few that are high-level are not appropriate for skinning.

At the low level, the Microsoft Windows GDI offers little imaging support beyond the ability to display a bitmap at an integer coordinate on the screen. Higher-level Microsoft Windows controls, such as those found in MFC® and Visual Basic®, are limited; they are designed only to build software that conforms to standard Microsoft

user interface guidelines for productivity applications. At this point, the Win32 API does not provide the functionality found in a system like Skinux.

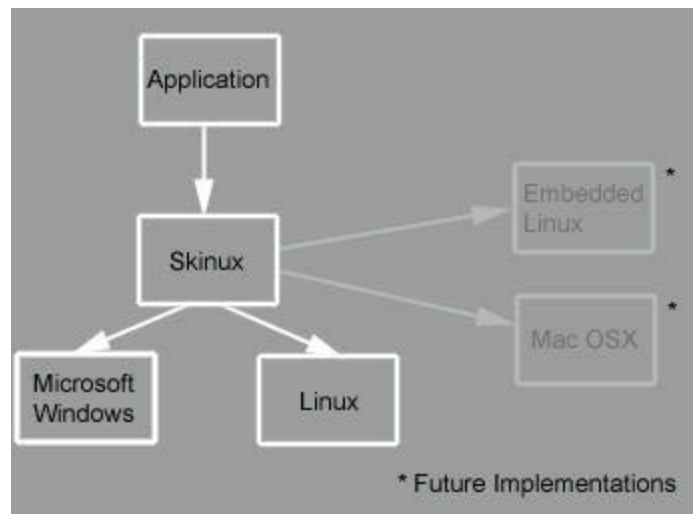
The same limitations are even more prevalent on Linux using X/Windows; the Linux GTK+ API has marginal font support and very little imaging support. There is no built-in alpha blending or any support for layered windows. In fact, X/Windows usually runs significantly slower than Microsoft Windows because of the lack of graphic hardware acceleration. Devices that use embedded Linux often restrict their output to a single window or forego X/Windows altogether in favor of the native FrameBuffer interface.

Specific benefits of Skinux technology vs. a traditional GUI are as follows:

- Cross Platform
- Interoperability
- Extensibility

Cross Platform

Skinux is the first truly cross platform skinning solution for Microsoft Windows and Linux systems. As shown below, a Skinux Skin User Interface is specified in a truly device independent fashion in XML and XSLT and is not dependent on any particular windowing system or operating system manufacturer.



Skinux's Cross Platform Device Independent Architecture

Skinux is written in C++ and its architecture isolates the calls into the native operating system through a well-defined set of service layers. Therefore, it is very portable to a variety of environments. Skinux is particularly well suited to embedded

Linux devices that need a superior Skin User Interface and access to the source code, in order to customize it to precise specifications.

Skinux also provides a binary streaming interface for all the objects defined in the system. This streaming architecture ensures that binary files are readable among different processor types, which may vary between little or big endian notation.

Interoperability

When selecting a user interface technology, it's important to consider whether or not it is interoperable with the display technology of the native operating system environment (specifically the Win32 API on Microsoft Windows or the GTK+ API on Linux). Skinux technology excels at interoperability because it does not try to replace the entire display architecture with a self-contained system with sandbox restrictions. This is particularly beneficial in cases where legacy software is already written in Win32 or MFC and companies want to add a Skin User Interface to a portion of the system without rewriting all the code so that it fits into a new architectural display paradigm.

Skinux manipulates the entire Skin User Interface internally, as a collection of C++ objects. However, it eventually displays the skin to an HWND (in the case of Microsoft Windows) or a GdkDrawable (in the case of GTK+ on Linux). It uses integration code to connect the Skinux event model to the Windows message loop or GTK signals. However, because the developer has access to the source code, the connection can be modified to accommodate any situation.

Extensibility

Extensibility is Skinux's most essential distinguishing feature. We understand that it is impossible for developers to anticipate all the control types that a skinable program might require. Although Skinux's baseline functionality includes common controls, such as buttons, knobs, and sliders, our technology enables developers to build special controls, such as a play list in a media player. This is essential when creating Skin User Interfaces that must have that unique, one-of-a-kind appearance. Lack of extensibility is one of the biggest limitations of Web multimedia systems, which often provide a closed player architecture that prevents developers from extending the system to meet the demands of their customers.

Custom Controls

Skinux is C++ based, with a well-defined class hierarchy for Skin User Interface controls, event types, and compositors. It is very easy for third-party developers to extend the system by sub-classing off this hierarchy. In this way, Skinux has a similar architecture to Microsoft Foundation Class MFC for developing applications.

Automatic Generation of XML Parser from C++ Code

In addition to being able to extend the C++ portion of Skinux with custom controls, third-party developers can also extend the Skinux XML language with new descriptions of their controls. Skinux makes this easy by providing a well-defined architecture for naming new control types, describing their parameters, and adding them to the Skinux Object registry. If developers follow the Skinux guidelines, the code for parsing the new control's XML description is automatically generated by the system.

XSLT Support

The XSLT support that Skinux technology provides is extremely important; without it, the XML skin file description usually becomes unnecessarily large and difficult to maintain. Raw XML is simply the reflection of the internal C++ data type that implements the Skin User Interface control. When multiple filter effects are combined, the actual unique information for a control can get lost in a sea of XML details. XSLT enables the user to factor out the XML code by using macros that can later be expanded into a full XML description. XSLT operates in a similar fashion with the C++ preprocessor. If used properly, it can reduce the size of the XML Skin User Interface description by more than 90%!

The following code example shows how XSLT macros are used. We define a template, 'TopButton' that is used to describe the XML description of a button within a Skinux media player.

```
<xsl:template match="TopButton">
  <Button>
    <Up>
      <StaticChannelControl>
        <Channel>
          <xsl:element name="ChannelFile">
            <xsl:attribute name="mask">
              <xsl:value-of select="@mask"/>
            </xsl:attribute>
          </xsl:element>
        </Channel>
        <Color color="rgb(10,10,10)"/>
      </StaticChannelControl>
    </Up>
    <Down>
      <StaticChannelControl>
        <Channel>
          <xsl:element name="ChannelFile">
```

```

        <xsl:attribute name="mask">
            <xsl:value-of select="@mask" />
        </xsl:attribute>
    </xsl:element>
</Channel>
<Color color="rgb(255,0,0)" />
</StaticChannelControl>
</Down>
<Rollover>
    <StaticChannelControl>
        <Channel>
            <xsl:element name="ChannelFile">
                <xsl:attribute name="mask">
                    <xsl:value-of select="@mask" />
                </xsl:attribute>
            </xsl:element>
        </Channel>
        <Color color="rgb(255,0,0)" />
    </StaticChannelControl>
</Rollover>
<xsl:element name="Id">
    <xsl:attribute name="val">
        <xsl:value-of select="@id" />
    </xsl:attribute>
</xsl:element>
<xsl:element name="Position">
    <xsl:attribute name="x">
        <xsl:value-of select="@x" />
    </xsl:attribute>
    <xsl:attribute name="y">
        <xsl:value-of select="@y" />
    </xsl:attribute>
</xsl:element>
</Button>
</xsl:template>

```

Once defined, this XSLT template can then be referenced repeatedly within the XML skin description of the media player panel as follows:

```

<TopButton mask="close_button.jpg"
    x="&EXIT_POS_X;"
    y="&EXIT_POS_Y;"
    id="&ID_EXIT;" />

<TopButton mask="min_button.jpg"
    x="&MIN_POS_X;"
    y="&MIN_POS_Y;"
    id="&ID_MIN;" />

<TopButton mask="skinamp.jpg"
    x="&SKINAMP_POS_X;"
    y="&SKINAMP_POS_Y;"
    id="&ID_SKINAMP;" />

```

When loading a skin, the C++ programmer simply specifies the XML file and the XSLT that operates on it:

```
PCFObject skinObject = FReadXML2Object(CFString("skin.xml"), CFString("skin.xsl"));
```

The Skinux Difference (Technical Features)

Sophisticated Imaging Model

Skinux was designed from the ground to be a state-of-the-art Skin User Interface development system. Our primary intent was to employ a sophisticated imaging model that supports all of the following:

- Layered Image Compositing
- Procedural Imaging
- Sub-pixel Positioning

Layered Image Compositing

Layered Image Compositing is a technology for constructing 24 bit images from a collection of 32-bit RGB(alpha) images, using alpha blending. The alpha layer of an image is used to represent its transparency value during the compositing process. This technology was initially developed to support image editing and video editing software. Layers make it possible for an image to be organized into distinct components, which can be moved and edited independently.

Skin User Interfaces use layered image compositing in the same way because each element of a Skin User Interface is represented by a 32-bit RGB(alpha) image. Instead of compositing images to produce a static photograph or video frame, a Skin User Interface system produces a dynamic control panel. The end result is an interface that is as photo-realistic as an image designed in Adobe PhotoShop. This is significantly different from the current graphical user interfaces found in Microsoft Windows or Linux, which use a much less sophisticated rendering process that is based on rectangular windows.

Procedural Imaging

Procedural imaging is a technology for describing images mathematically or synthetically modifying them. This includes:

- Antialiased shape rasterization
- Antialiased text
- Gradients

- Patterns and textures
- Images processing filters (drop shadow, emboss edge, etc.)

Skinux provides an architecture that enables developers to integrate procedural image-based controls into a Skin User interface framework very easily. Currently, Skinux provides a number of built-in image processing filters, such as drop shadows, emboss edge, fringe edge, and blur. More importantly, developers can create and apply custom filters and add them to the framework easily without needing to recompile the system.

There are significant advantages to using procedural imaging, instead of static image data:

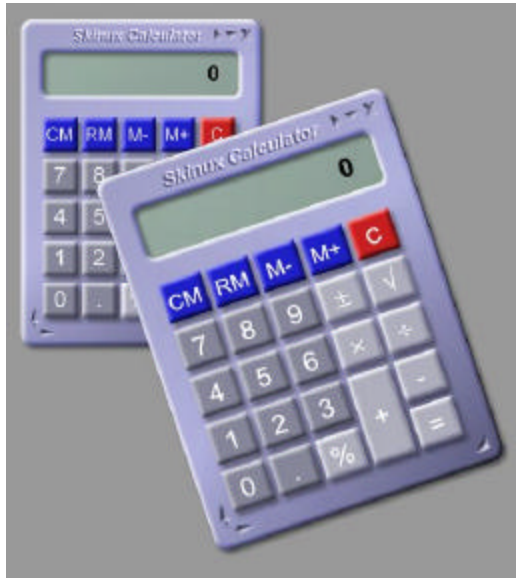
- Images are described mathematically, so that their parameters can be modulated over time.
- The procedural description is usually more compact than the resulting image data, which makes possible greater image compression ratios.
- Procedural images are very easy to edit.

Sub-pixel Positioning

The Skinux imaging model is based entirely on floating point image coordinates. This means the system can composite a target 32-bit RGB(alpha) image on a target image at a non-integer position. This is extremely important when supporting smooth animation imaging effects. For example, Skinux allows a user to move an image control along a Bezier path without the jumps that would be visible if one used simple integer based alpha blending. This subtle yet powerful feature is usually not supported in more primitive user interface systems.

Scalable Imaging

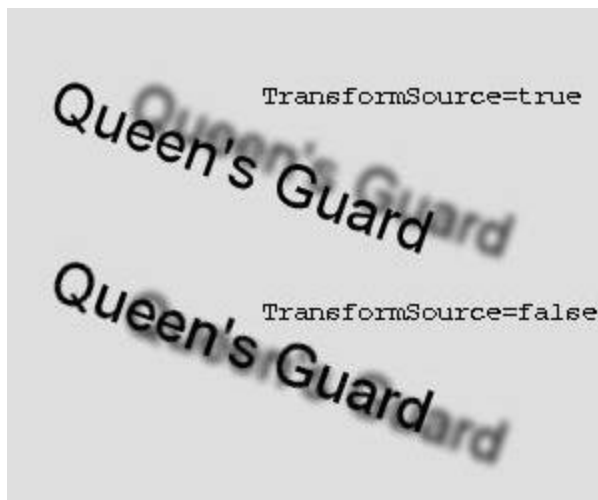
What distinguishes Skinux from traditional layered image compositing systems found in image and video editing tools is that its imaging model is scalable! You can actually resize Skinux imaging controls and have them rendered to any resolution without loss of detail. The reason for this is that Skinux supports a procedural and vector based model, in addition to a pixel image data model. The procedural imaging, filters, and compositing functions can be used to generate an imaged look while having all the scaling capabilities of a vector language such as PostScript. For example, the graphic below shows a Skinux Calculator skin that was resized to a larger resolution; note that there is no loss of detail caused by pixel scaling effects.



Scalable Skinux™ Calculator

Skinux also allows designers to specify whether or not a filter effect control can delegate a transformation onto its source bitmap control. Like our sub-pixel positioning, this is a subtle feature that sets Skinux apart from other user interface systems. For example, if you consider a drop shadow effect applied to a masked image, you might not want the transformation to apply to the drop shadow; instead, you might only want to apply it to its source.

The flexibility of this feature is exemplified in the following graphic. In the top example (`TransformSource = true`), the text “Queen’s Guard” is rotated by 20 degrees, after which a drop shadow filter with an offset of (40, 2) is applied. In the bottom example (`TransformSource = false`), a drop shadow filter with an offset of (40,2) is applied to the text, after which the result is rotated by 20 degrees.

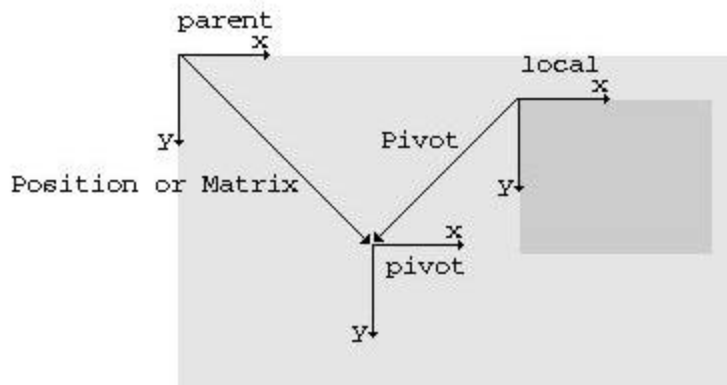


Source level transformation by a Skinux filter effect control

Pivot Coordinate System

What makes the Skinux imaging model scalable is its notion of a pivot coordinate system assigned to each control. In Skinux, all Skin User Interface controls belong to a hierarchy that defines a containment relationship between them. If a control is contained by another control, the container is said to be the “parent” control and the contained control is said to be its “child.” For example, a Group control can contain other controls, which are its children. Similarly, a button control contains child bitmap controls that describe its up, down and rollover states, respectively. This containment relationship between parent and child controls also defines how coordinate systems work within Skinux.

All Skinux coordinate and position information is described using a two-dimensional affine floating point Cartesian coordinate system. Furthermore, the positive direction of the ‘y’ vertical axis always points downwards so that the control’s origin is always in the upper left-hand corner. A control’s coordinate system is always relative to its parent control’s coordinate system. However, the transformation between coordinate systems involves an intermediate coordinate system known as the ‘pivot’ coordinate system. The pivot coordinate system is necessary in order to preserve a control’s center of gravity and alignment.



- The control’s coordinate system is referred to as the ‘local’ coordinate system.
- The control’s pivot coordinate system is referred to as the ‘pivot’ coordinate system.
- The parent’s coordinate system is referred to as the ‘parent’ coordinate system

The pivot point is the origin of the control’s pivot coordinate system and is expressed in local coordinates. The pivot point is also the connection between the parent and the local coordinate system. The transformation between a local coordinate system and its parent is given by the following formula:

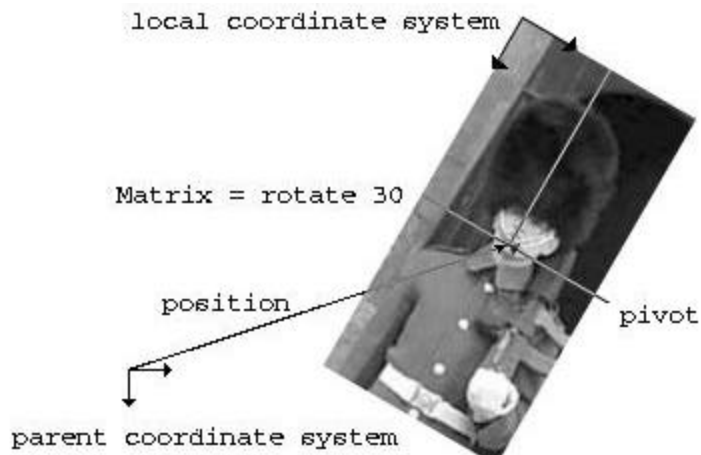
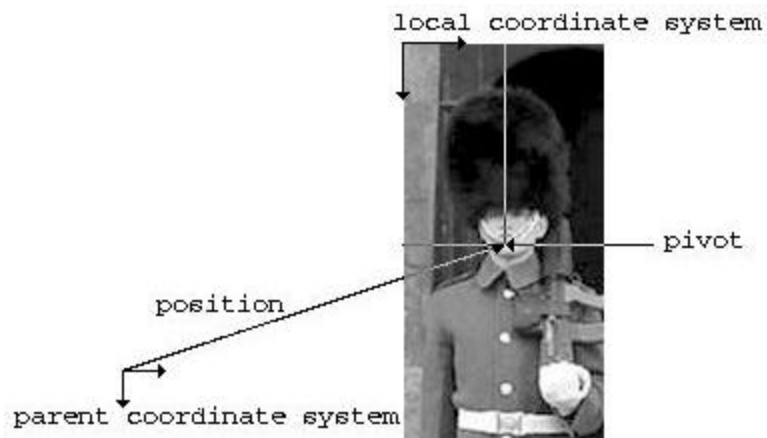
$$[u \ v \ w] = [x \ y \ 1] * \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -Pivot_x & -Pivot_y & 0 \end{vmatrix} * Matrix_{control}$$

and:

$$X_{parent} = u / w$$

$$Y_{parent} = v / w$$

Developers might ask, “Why define a third coordinate system, using a pivot point?” The reason the pivot is important is that it enables the user to control how a child control is placed, relative to its parent control. Child local coordinate systems are always ‘fixed’ with respect to the definition of the control type. For example, the local origin of a bitmap control is always the upper left-hand corner of the bitmap. However, the user may want to position the center of the bitmap, rather than its upper left-hand corner, relative to the parent coordinate system, as shown in the graphics below. That is where the pivot point comes in handy. In this case, the user would define a pivot point of $(s_x / 2, s_y / 2)$, where (s_x, s_y) is the width and height of the bitmap.



Composite Level Transformation

Another advantageous feature of Skinux is its composite level transformation architecture. This architecture enables Skinux controls to be transformed only into their parent coordinate space at the exact time that they need to be composited onto their parent panel control. Even though Skinux controls consist primarily of image data, Skinux treats them mathematically as vector data, transforming them only when they are rendered.

When a Skinux control is transformed, no image data is actually scaled. Instead, the transformation is recorded by concatenating it to the local matrix. It is only when the control's parent panel is rendered that an intermediate transformed image is computed for the purpose of projecting it into the parent coordinate space. The control always preserves the source of its original image data (if it has any) from which the intermediate forms are computed. As a result, there is no jagged appearance or other degradation in resolution caused by repeatedly applying linear transformations to pixel data.

This feature makes it possible for a slider knob, for example, to be rotated smoothly as it is dragged across a Bezier path, as shown in the heart-shaped Skinux Media Player™ below.



Rotating knob along a Bezier Path in a customized Skinux Media Player™

Layout Managers

As part of its image scaling architecture, Skinux supports the concept of control layout managers. Often, a resize operation should not necessarily result in a simple linear scaling transformation. Instead, the user might expect some elements to be scaled and others not. For example, in a panel that contains two sliders for horizontal and vertical scrolling, one would expect that when the panel is resized, the length of the sliders would be resized accordingly but their thickness would remain constant.

A simple linear transformation cannot accomplish this task because it would scale everything indiscriminately. To move beyond this limitation, the architecture needs a way to specify control position and sizes in a relative way when responding to a resize event. This is what Skinux layout managers accomplish; they enable a Skinux control's position and boundary to be specified relative to their parent control or sibling control in absolute, relative, or percentage-based coordinates. Using a layout manager, a Skinux developer can specify a slider by an absolute width of 10 pixels but a variable height to 100% of the height of its parent control. Furthermore, the position can be specified as 10 pixels from the right edge and 50% from the top, as an example.

Built-in Animation Model

Since its inception, Skinux was designed to support 32-bit image based animation. Controls can be positioned and resized anywhere within their parent panels over time. Skinux prevents 'jerky' animation behavior by supporting sub-pixel positioning in its layered image compositing model. As elements are moved across the screen, they can be composited at non-integer coordinates. Moreover, the Skinux architecture allows any control parameter (in addition to position information) to be modulated over time. For example, it is very easy to achieve a pulsating drop shadow or glow effect around an image control in Skinux. Similarly, gradients can cycle their colors over time. Skinux achieves this by providing a name-binding interface for every control parameter and supporting built-in parameter type interpolation over time. In this way, if a control has a color c_0 at time t_0 and a color c_1 at time t_1 , Skinux will automatically compute intermediate colors c for any t between t_0 and t_1 .

What is striking is that Skinux achieves this animation model within a full 32-bit continuous-tone color space with transparency information, unlike animation schemes that are constrained to 256 color spaces, such as GIF89A. Skinux can do this in real time because its imaging library is highly optimized for the Intel MMX™ processor.

Parameter Late Binding Interface

Skinux supports the concept of an object registry, which records information about every object class and control used within an application. Specifically, Skinux stores name and type information about the parameters to every Skinux control class. It is from this registry that Skinux is able to process an XML description of a control in a file that describes a Skin User Interface. Each parameter name is unique within the control class in which it is defined. In C++, object registry information for a control is specified by using a predefined set of macros, as shown below. For example, the C++ object registry entry for the drop shadow filter control and its parameters would be expressed by the following code snippet:

```
const CFString CFDropShadowControl::OffsetParam(L"Offset");
const CFString CFDropShadowControl::RadiusParam(L"Radius");
const CFString CFDropShadowControl::ShadowColorParam(L"Color");
const CFString CFDropShadowControl::ScaleParam(L"Scale");

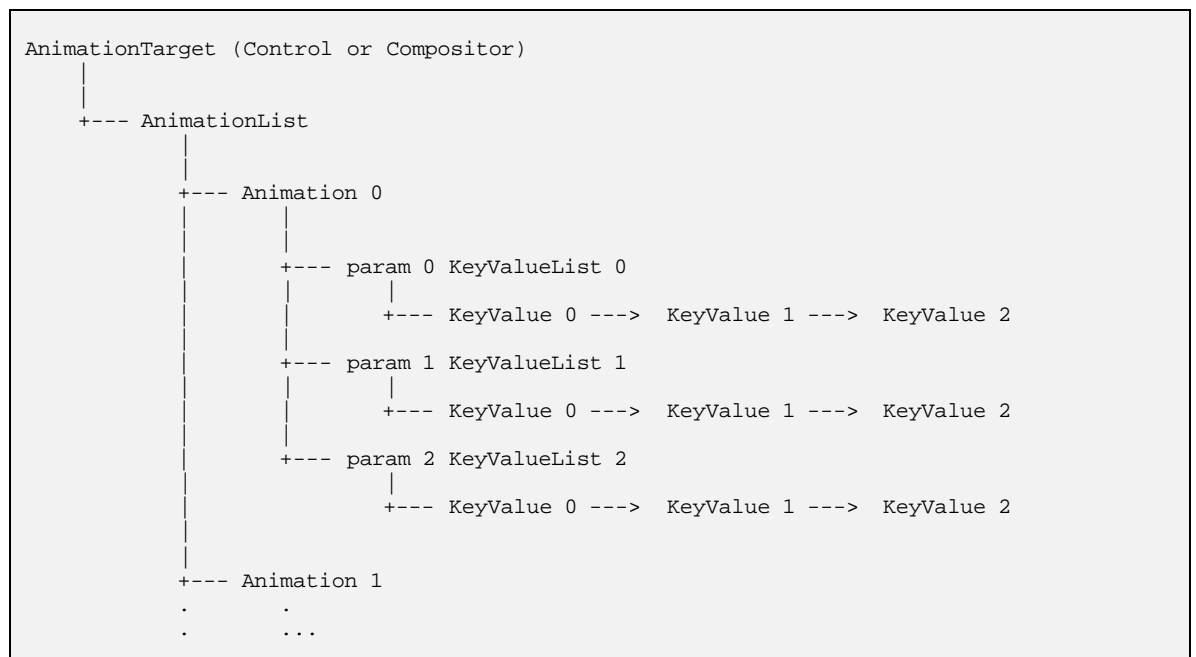
static const CFParamDef ControlParams[] =
{
    CFParamDef(CFDropShadowControl::OffsetParam, CFDoublePoint::QClassType(),
true,
        XML_PARAM_AUTOGEN),
    CFParamDef(CFDropShadowControl::RadiusParam, CFInt32::QClassType(), true,
        XML_PARAM_AUTOGEN),
    CFParamDef(CFDropShadowControl::ShadowColorParam, CFRgb32::QClassType(),
true,
        XML_PARAM_AUTOGEN),
    CFParamDef(CFDropShadowControl::ScaleParam, CFDouble::QClassType(), true,
        XML_PARAM_AUTOGEN)
};

DEFINE_PARAM_SET(CFDropShadowControl, CFFilterControl, ControlParams);
DEFINE_CMP_SERIAL_OBJECT(CFDropShadowControl, "DropShadowControl", CFFilterControl)
```

In this example, there are four parameters unique to a drop shadow control called “Offset”, “Radius”, “Color” and “Scale.” These specific names are used by an animation object to modulate the parameter values over time.

Parameter Lists and Key Values

In Skinux, an animation object is a dictionary of key/value pairs that modulate specific parameter names of the animation target to which it is associated. An animation target consists of a Skin User Interface control or a compositor object. Furthermore, a control or compositor can have multiple animation objects at the same time, which means that two animations that modulate different parameters can be running in parallel. A key value defines a key (or time), which is a floating point number, and a value that is of the same type as the animation target parameter to which it refers. It essentially states what the value of the parameter is (specified by its value) at a specific time (specified by its key). Distinct animations within an animation target are differentiated by unique id(s).



An animation also is a key (or time) denoted by its Key element. Changing the value of this animation Key animates a control or compositor. The system modulates a parameter by finding (and, if necessary, interpolating) the closest key value in the KeyValueList associated with the parameter.

A control's parameters are set by specifying a single animation time value and having Skinux determine those values by using the Animation dictionary.

Animation Triggers

In Skinux, animations associated with controls can be triggered by events such as mouse rollovers or button presses. The object that determines when and how an animation is played out is known as an Animation Trigger object. Triggers specify a time value, a frequency interval, and whether or not the animation cycles.

Layered Windows

Microsoft Windows 2000® and XP® introduced a new layered window style. When the layered window flag is on, rendering to the window is cached in a bitmap of the size of the window, instead of being written directly to the screen. It can then be blended together with the whole screen display, using alpha blending. A layered window provides a new visual effect and improves system performance at the cost of extra cache bitmaps.

Prior to the introduction of layered windows, it was very difficult to display a non-rectangular window on the screen. Windows 95/98 allowed developers to set a window region, but this was limited to an on/off Boolean mask, which produced very 'jagged' edges. By using layered windows, it is possible to alpha-blend smooth anti-aliased windows onto the desktop in a translucent fashion.

Skinux supports layered windows in the Microsoft Windows 2000 and XP environments.



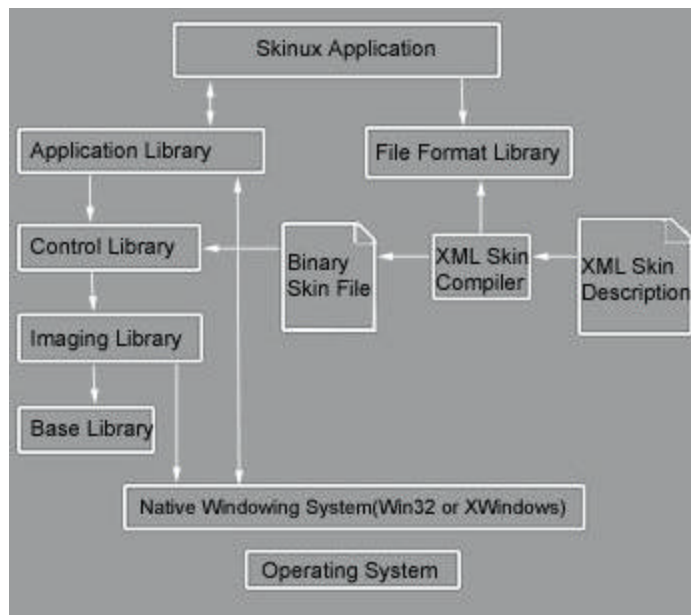
Skinux Layered Windows

The Skinux Architecture

The Skinux architecture consists of the following components:

- Skinux SDK
- Skinux XML Language
- MMX Optimized Imaging Library

The architecture of a Skinux program and its relationship to the Skinux libraries is shown in the diagram below:



Our company developed Skinux with the realization that Adobe PhotoShop is the primary tool that artists and designers use to create and edit image data. Rather than reinvent the wheel with another image editor, we decided to leverage Adobe's success. In fact, the Skinux authoring process uses Adobe PhotoShop at the design level intensively. Usually, an artist creates the initial design in PhotoShop first, before a single line of code is written. Because the Adobe image layering architecture closely resembles the Skinux one, it is ideally suited to map out designs. In fact, PhotoShop layers are implemented as 32-bit RGB (alpha) images. To aid in the design process, Skinux has provided two plugins that extract the 24-bit color and 8 bit alpha information from a PhotoShop layer.

Skinux Development Kit (SDK)

The Skinux Developer Kit (SDK) is a collection of C++ source code class libraries used to develop Skin User Interfaces. Our product is cross platform and provides a unified application-programming interface API across Microsoft Windows and Linux. Currently, the SDK assumes a Win32 API on Microsoft Windows and a GTK+ API on Linux. The SDK also works with the Microsoft Foundation classes (MFC); however, it does not require them.

At the heart of the SDK is a very fast 32-bit RGB-alpha image compositing engine. The code is highly optimized for the Intel MMX™ processor and provides assembler as well as C++ versions of the most critical loops. Every Skin User Interface element has an associated 32-bit bitmap, including backgrounds, static bitmaps, or interface controls such buttons or sliders. The 8-bit alpha channel is used to control transparency or other compositing effects.

The libraries also include procedural methods for specifying image data, including paths, gradients, patterns, and filters. The SDK supports a 2D transformational model for placing, scaling, and rotating image data anywhere within the compositing space. It also supports sub-pixel positioning and an animation model for creating smooth transitions over time. In this way, the SDK enables developers to build photo-realistic and well-designed skins very easily and much more quickly than they could if they used conventional GUI libraries on Windows or Linux.

The Skinux SDK consists of the following C++ libraries:

- Skinux Base Library
- Skinux Imaging Library
- Skinux Control Library
- Skinux Application Library
- Skinux File Format Library
- Skinux XML Library

Base Library

The Skinux Base library is a collection of base C++ classes used to implement the elements of a Skin User Interface. The classes are portable across all operating system environments and provide a framework from which to build portable Skinux applications.

Imaging Library

The Skinux Imaging library contains most of the code that performs the low level imaging operations. Large sections of the Imaging library are written in MMX assembler, for speed optimization on the Intel chipset. However, to facilitate portability, all assembler sections also have an equivalent C++ section. The imaging library is portable across all operating system platforms.

The imaging library provides a number of raster operations, presented in the form of C functions, which operate on both 32-bit RGBA bitmaps and 8-bit grayscale channels. The raster operations allow the programmer to blur, copy, fill, offset, and scroll bitmaps or channels very quickly. They also provide functions to average bitmaps (for anti-aliasing), resize, or distort them to an arbitrary quadrangle. In addition, there are raster operations for a number of image compositing functions, including alpha blending, merging, embossing, minimum, maximum, addition, subtraction, multiplication, and difference. Furthermore, all image compositing functions operate on sub-pixel positions to produce clean anti-aliased results that ultimately enable smooth animations.

The imaging library also provides imaging filter functions. These include shape rasterization to an arbitrary Bezier path and procedural gradients (line, elliptical, rectangular, and arc). There are also bitmap edge filters to create a fringed, feathered, or embossed look to a button or other control. Also, Skinux provides a fast drop shadow filter to create the appearance of 3D across a program's user interface.

Hierarchical Control Library

The Skinux control library contains the higher level imaging controls from which to build a Skin User Interface. In addition to providing a widget library, it also includes a portable event model, an animation model, and a hierarchical compositing engine. All Skinux widgets are essentially 32-bit RGBA semi-transparent images that can be alpha blended, using a variety of compositing functions.

All Skinux controls provide a late binding interface that supports a general animation model. Skinux animations enable any control parameter to be changed and interpolated over time.

XML Library

The Skinux XML Library contains the parser for the Skinux XML language. This library is cross platform and runs on both Microsoft Windows and Linux. The XML library ensures that a skin description is a well-formed XML document and that it conforms to the Skinux XML syntax defined by the Skinux Document Type Definition. To facilitate macro programming, Skinux also supports the standard XSLT programming language in XML. The Skinux XML Library is based on the

code of the libxml2 library provided by XMLSoft. More information about XMLSoft can be found at www.xmlsoft.org.

Format Library

The File Format library provides file format import capabilities to Skinux applications. The purpose of the File Format library is enable developers to import artwork, either in the form of image data or path data, that is created with third-party authoring tools such as Adobe PhotoShop, Gimp®, or Corel Draw. The File Format library is based on the Independent JPEG Group's software. More information about the Independent JPEG Group can be found at www.ijg.org.

Standardizing on JPEG rather than the Microsoft BMP® format makes skins significantly more compact in memory size because JPEG has much better compression ratios.

Application Library

The Skinux Application Library provides an operating system independent and portable framework from which to build skinned applications. The Application library provides the “glue” code between a native windowing system's event loop and the Skinux portable event-handling model. By using the Skinux Application library, programmers are free to focus on their core application code, rather than on the intricacies of porting between different operating system environments. The Application library defines the message loop and window procedures in the Win32 environment, as well as the necessary event handlers in the GTK X/Windows Linux environment. It also takes care of marshaling events and messages between the native operating system window systems and the Skinux application.

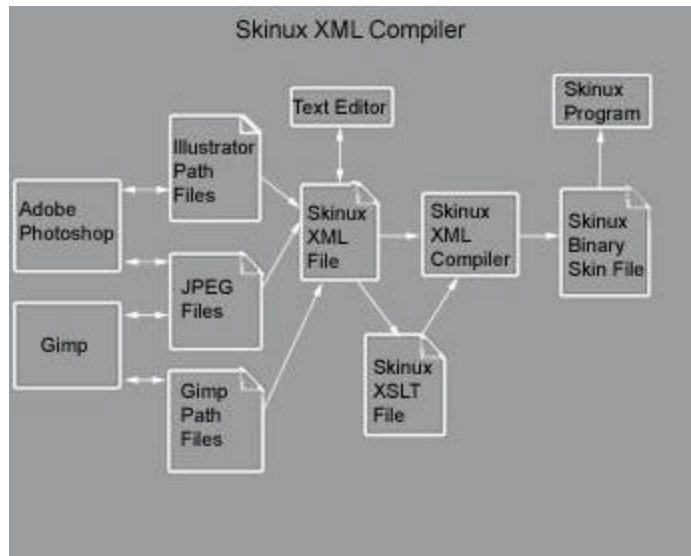
Skinux XML Language

One of the main advantages of Skinux is that it uses XML (Extensible Markup Language) to define the skins of an application's user interface. XML is widely understood and accepted among developers, and many authoring tools are available for it. Skinux allows designers to specify the skin of an application's user interface independently of the code by using a combination of XML, JPEG image files, Gimp path and gradient files, and Adobe Illustrator® path files. De-coupling the skin specification from the application code provides the following advantages:

- Programmers and designers can work independently of one another
- Designers do not need to know C++
- Applications do not need to be recompiled in order to change skins

- Process facilitates internationalization
- Program executables are smaller

As shown in the graphic below, the Skinux design process is very simple. The designer authors a skin using the Skinux XML language. We have also added XSLT processing, which enables a more compact XML skin description, along with a faster way to change selected attributes all at once.



Extensible XML

One of the most beneficial advantages to XML is that it's extensible! The same is also true of the Skinux XML language. As stated earlier in this document, Skinux uses information about control types and their parameters from the object registry to determine how to process an XML description. This means that, as long as a developer adds a new control to the object registry, Skinux will be able to parse its XML description.

The XML specification for the Skinux base classes and standard controls is open source and can be found on the Web at www.skinux.com/SkinuxXML/skinux-xml/book1.html. However, third-party developers can extend it in any way they see fit.

DTD Generation and Python Support

The Skinux XML compiler supports a switch `-dtd` that, when set, will output the XML Data Type Definition file for the entire registered Skinux object to standard out. This DTD can be used for XML syntax validation of Skin User Interface definition files.

The Skinux XML compiler can also be used to generate a file of Python functions that can be used to output the XML text descriptions of various Skinux elements. Python support was initially added to Skinux because it helps to cut down the size of skin XML files. However, Skinux ultimately included XSLT support, as well, because it is written in XML and is therefore better suited for that task. For more information about Python see www.python.org.

MMX Optimized Imaging Library

It is the Intel MMX processor that makes Skinux's outstanding performance possible. Our system was built to support and utilize the SIMD (Single Instruction Multiple Data) form of parallel processing offered by the MMX architecture. Our company has observed that performance increases by 2-3 times when specific image processing operations such as alpha-blending are optimized for the MMX instruction set. This has placed a special burden on our company's developers because MMX code must be written in 80x86 Intel Assembly code. Furthermore, this coding style can be very difficult because of load balancing between the U and V pipes of the processor through instruction pairing and because of hardware latency issues. Over 30% of the Skinux system is written in MMX assembler code, with the equivalent C code for algorithm portability.

The following code snippet of the Skinux bitmap alpha blending provides an idea of what the MMX assembler code looks like. Note that the MMX code also has a C equivalent that is usually shown, below.

```

/*-----*/
|           BlendBitmapFast
/*-----*/
static void BlendBitmapFast(PCFBitmap dest,
                           PCFBitmap source,
                           const CFPoint &pos,
                           const CRect &clipRect)
{
    FLock2(dest, source);

    CRect sourceRect = source->QSizeRect() + pos;
    CRect destRect = dest->QSizeRect();
    CRect r = sourceRect & destRect & source->QClipRect(pos, clipRect);

    CFPoint size = r.QSize();
    CFPoint dstTopLeft = r.QTopLeft();
    CFPoint srcTopLeft = dstTopLeft - pos;

    if(!r.QIsEmpty())
    {
#ifdef WIN32
        if(CFSystemConfiguration::QIsMMXEnabled())
        {
            for(int32 y = 0; y < size.Y; y++)
            {
                rgb32 *pDest = dest->QPixelPtr(dstTopLeft.X, dstTopLeft.Y + y),
                    *pSrc = source->QPixelPtr(srcTopLeft.X, srcTopLeft.Y + y);

```

```

uns32 lastX = size.X - 1;

int64 AlphaAndMask = CONST64(0x00000000FFFFFFFF);
int64 alphaMask = CONST64(0xFF000000);

_asm
{
    mov esi, pSrc
    mov edi, pDest
    mov ecx, lastX

    pxor mm7, mm7

mmx_blend_loop:
    movd mm0, [esi + 4 * ecx]    ; mm0 = src pixel in lower dword
    movd mm4, [edi + 4 * ecx]    ; mm1 = destination pixel in lower dword
    punpcklbw mm0, mm7

    punpcklbw mm4, mm7
    movq mm1, mm0

    punpckhbw mm1, mm1

    punpckhbw mm1, mm1

    punpcklbw mm1, mm7          ; mm1 = 0A0A0A0A0A
    pcmpeqd mm6, mm6

    pmullw mm0, mm1
    psrlw mm6, 8

    psubusb mm6, mm1            ; mm6 = 255 - A

    pmullw mm4, mm6
    movq mm1, mm0

    punpckhwd mm0, mm7          ; mm0 = A', R' unpacked to 32 bits
    movq mm5, mm4

    punpckhwd mm4, mm7          ; mm4 = DA', DR'
    pcmpeqd mm6, mm6

    punpcklwd mm1, mm7          ; mm1 = G', B' unpacked to 32 bits
    padd mm0, mm4               ; mm0 = FA * 255, FR * 255

    punpcklwd mm5, mm7          ; mm5 = DG', DB'
    movq mm2, mm0

    padd mm1, mm5               ; mm1 = FG * 255, FR * 255
    pslld mm2, 8

    padd mm0, mm2               ; mm0 = FA, FR (8.16)
    movq mm3, mm1

    pslld mm3, 8
    movq mm2, mm0

    padd mm1, mm3               ; mm1 = FG, FB (8.16)
    psrld mm2, 15

    psrld mm6, 31
    movq mm3, mm1

    psrld mm0, 16
    pand mm2, mm6

    movd mm4, [edi + 4 * ecx]
    padd mm0, mm2               ; mm0 = FA, FR (8. rounded)

    movq mm5, alphaMask
    psrld mm3, 15

```

```

        psrld mm1, 16
        pand mm3, mm6

        padd mm1, mm3                ; mm1 = FA, FR (8. rounded)
        pand mm4, mm5

        packuswb mm1, mm0            ; mm1 = 00AA00RR00GG00BB
        packuswb mm1, mm7            ; mm1 = 00000000AARRGGBB

        pandn mm5, mm1

        por mm5, mm4

        movd [edi + 4 * ecx], mm5

        sub ecx, 1
        jnc mmx_blend_loop
    }
}

_asm emms;
}
else
#endif
{
    for(int32 y = 0; y < size.Y; y++)
    {
        rgb32 *pDest = dest->QPixelPtr(dstTopLeft.X, dstTopLeft.Y + y),
            *pSrc = source->QPixelPtr(srcTopLeft.X, srcTopLeft.Y + y);

        for(int32 x = 0; x < size.X; x++)
        {
            uns32 srcAlpha = UNPACK_A(*pSrc),
                srcRed = UNPACK_R(*pSrc),
                srcGreen = UNPACK_G(*pSrc),
                srcBlue = UNPACK_B(*pSrc),
                dstRed = UNPACK_R(*pDest),
                dstGreen = UNPACK_G(*pDest),
                dstBlue = UNPACK_B(*pDest),
                dstAlpha = 255 - srcAlpha;

            dstRed = BYTEMULT(dstAlpha, dstRed) + BYTEMULT(srcAlpha, srcRed);
            dstGreen = BYTEMULT(dstAlpha, dstGreen) + BYTEMULT(srcAlpha, srcGreen);
            dstBlue = BYTEMULT(dstAlpha, dstBlue) + BYTEMULT(srcAlpha, srcBlue);

            dstAlpha = UNPACK_A(*pDest);

            *pDest = PACK_RGBA(dstRed, dstGreen, dstBlue, dstAlpha);

            pDest++;
            pSrc++;
        }
    }
}

FUnlock2(dest, source);
}

```

In summary, the primary benefits of the Skinux architecture are its modular and portable SDK; its easy to use resource description language, which is based on standard XML; and its very fast MMX optimized imaging library, which enables animation effects and procedural imaging in real time.

Industry Experience

Richard Krueger is the founder, CEO and CTO of Skinux Inc., founded in February 2000. During the past five years, Richard has been developing the foundations for the Skinux product, resulting in a highly sophisticated imaging architecture.

Prior to founding Skinux, Richard was the Director of Imaging at Macromedia® Inc. from August 1995 to October 1997. He was the author and lead programmer for Macromedia xRes®, which later formed the basis of the Macromedia Fireworks® product.

In 1992, Richard founded Fauve Software™ with his brother, Fred Krueger, to develop a natural media paint product for the Windows and Macintosh® systems. Their flagship product – Fauve Matisse™ – was voted Product of the Year by *Imaging Magazine* and received a four-star review from *Publish* magazine. They sold the company to Macromedia in August of 1995. Prior to founding Fauve Software, Richard worked for the Institute of System Science in Singapore, developing a 3-D medical imaging system. In the 1980s, he worked for SAS Institute® in their 3-D graphics division and for IBM® in their PC graphics division.

Contact Information

When responding to this document, please use the following contact information:

Skinux, Inc.
Attn: Richard Krueger
106A Fountain Brook Circle
Cary, NC 27511
Telephone: 919.461.8989
E-mail: krueger@skinux.com