



## The Woopsi User Interface Library

# Contents

## Chapter 1 [Introduction](#)

---

DS Application Development  
The Woopsi User Interface Library

## Chapter 2 [Setting up a Development Environment](#)

---

Installing DevKitARM and LibNDS  
Installing a DS Emulator  
Installing Woopsi

## Chapter 3 [Naming Conventions and Style](#)

---

General Text Formatting  
Classes  
Namespaces  
Methods  
Constants  
Members  
Variables  
Enums  
Types

## Chapter 4 [Coding for the DS](#)

---

Developing for Embedded Systems – The C++ Subset  
Stack Limits  
RAM Size  
RAM Emulation Difficulties  
Floating-Point Maths  
Datatypes and Speed  
Memory Copying

## Chapter 5 [Woopsi's Hardware Abstraction Layer](#)

---

Hardware Initialisation  
Framebuffer  
DMA Hardware  
Vertical Blanks  
SDL Capabilities

## Chapter 6 [Woopsi Utility Classes](#)

---

String Handling  
  
The WoopsiString Class  
The Stringlterator Class

The Text Class

Collections

The WoopsiArray Class

Dates

The Date Class

## Chapter 7 Graphics Classes

---

Bitmaps

The BitmapBase Class  
The BitmapWrapper Class  
The MutableBitmapBase Class  
The FrameBuffer Class  
The Bitmap Class

Bitmap Manipulation

The Graphics Class  
The GraphicsPort Class

Animation

The Animation Class

Fonts

The FontBase Class  
The FixedWidthFontBase Class  
The Font Class  
The MonoFont Class  
The PackedFontBase Class  
The PackedFont1 Class  
The PackedFont16 Class

## Chapter X Gadgets

---

Buttons

The Button  
The BitmapButton  
The AnimButton  
The CycleButton  
The DecorationGlyphButton

Text Containers

The Label  
The TextBox  
The MultiLineTextBox  
The ScrollingTextBox

Screens and Windows

The Screen

The AmigaScreen  
The Window  
The AmigaWindow

#### Requesters

The Alert  
The Requester  
The FileRequester

**Chapter X**    **Basic Program Structure**

**Chapter X**    **Co-ordinate Systems**

# Introduction

## DS Application Development

The Nintendo DS is the ideal handheld platform for developing homebrew applications. The barrier to entry is lower than that of the Xbox360 and iPhone/iPod Touch – there are no developer fees, no requirements to use a particular operating system or IDE, and no opaque approval process before your applications are available. Applications can be released quickly and frequently, and the sourcecode can be made public for crowdsourced bughunting and development. The potential audience for DS homebrew is considerably larger than that of the GP32/GP2X/Wiz as there are far more DS consoles in circulation. The DS provides the potential for more intuitive and expressive applications than the PSP due to its touchscreen and dual displays.

However, the standard DS development kit includes no high-level support for writing applications. If you need to present a user with a button containing some arbitrary text that can respond to the stylus, you will need to:

- Write your own rectangle drawing routine that hits the framebuffer directly;
- Write your own string drawing routine that can position text within the button;
- Scan the touchscreen every vertical blank for collisions with your button.

The button is one of the simplest components of an interactive graphical user interface, yet even creating this most basic functionality represents an unwanted burden upon the developer. Writing user interfaces is therefore a tedious, lengthy chore, and applications that need them typically fall into one of two types:

- They are great programs with hideous, unintuitive interfaces;
- They have great interfaces but are hideous, unintuitive programs.

More often than not, homebrew applications are abandoned long before development has finished. The difficulty and tedium in creating a user interface is arguably one of the primary reasons for this.

## The Woopsi User Interface Library

Woopsi is intended to solve this problem. It is the first object-orientated graphical user interface library for the DS. It may even have been the first OO GUI library for a handheld console.

Woopsi has been specifically designed to enable developers to rapidly put together interfaces for their programs. It abstracts away the complexities of working with the DS hardware and provides a comprehensive set of UI components, bitmap manipulation and drawing tools and collection classes. It makes use of many features of the DS hardware, including its dual screens, touchscreen, gaming controls and DMA, to provide a fast, lightweight and cohesive user interface.

# Setting up a Development Environment

## Installing DevKitARM and LibNDS

Woopsi uses the devkitARM SDK. This can be obtained from the following website:

<http://www.devkitpro.org>

The devkitPro website includes instructions for installing the SDK on OSX and other UNIX-like operating systems. Windows users can download an automated installer that will do everything for you.

## Installing a DS Emulator

To test the applications you create, you will need to download a DS emulator. This is a program that simulates the behaviour of a DS and can run DS programs. The recommended emulator for Woopsi is DeSmuME, which can be downloaded for multiple operating systems from here:

<http://www.desmume.org>

Once downloaded, simply unzip the archive to your hard disk.

## Installing Woopsi

Once you have installed devkitARM, download the latest Woopsi source package from the following website:

<http://www.sourceforge.net/projects/woopsi>

To install Woopsi:

- Unzip the archive to your hard disk.
- Open the folder that was created during the unzip process.
- Copy the “Woopsi/libwoopsi” folder to your devkitPro folder. Assuming you are using Windows and the default settings, this will be “C:\devkitPro”. If you followed the recommended procedure for other operating systems, it will be “/opt/devkitpro”.
- Copy the “Woopsi/libs/libfreetype” folder to your devkitPro folder.

To test the installation:

- Open a command line.
- CD to the directory containing the unzipped Woopsi archive.
- CD to the “Woopsi/template” folder.
- Type “make”.
- Open the “Release/template.nds” ROM in your DS emulator.

If everything has gone to plan, the emulator should show a typical “Hello World” application.

# Naming Conventions and Style

The Woopsi library follows strict naming conventions. It is recommended that code using the library follows the same conventions, but this is not mandatory. Code intended for submission back to Woopsi itself must follow these guidelines.

TODO: This should be an appendix.

## General Text Formatting

- Text files should be indent with tabs.
- Tab width should be 4 spaces.
- Comments should aim not to exceed 80 characters wide (this does not apply to the code itself).
- Comments within functions should precede the block of code that they are relevant to. Comments should never appear after the block of code that they discuss.

## Classes

- Each class should be split into a .h and a .cpp file.
- Classes to be submitted to the Woopsi library must be included in the “WoopsiUI” namespace.
- Classes should be named using UpperCamelCase.
- Class .h files should use `#ifndef _MY_CLASS_H_` and `#endif` guards rather than `#pragma once`:

```
#ifndef _MY_CLASS_H_
#define _MY_CLASS_H_

class MyClass {
    ...
};

#endif
```

## Namespaces

- Namespaces should be named using UpperCamelCase:

```
namespace MyNamespace {
};
```

## Methods

- Methods should be named using lowerCamelCase:

```
class MyClass {
public:
    void doSomeStuff();
};
```

- Methods must have an associated Javadoc-style comment:

```
/**
 * Method that does something.
 * @param value An argument of some sort.
 * @return Something related to the function.
 */
u32 doSomething(u32 value);
```

## Constants

- Constants should be named using UPPER\_CASE\_WITH\_UNDERSCORES:

```
#define SOME_CONSTANT 12345
const int ANOTHER_CONSTANT = 1;
```

## Members

- Member variables should be prefixed with underscores.
- Member variables should be named using `_lowerCamelCase`:

```
class MyClass {
private:
    char* _somePrivateMember;
    u8 _anotherPrivateMember;
};
```

## Variables

- Variables should be named using lowerCamelCase:

```
void myFunction() {
    u8 someValue;
    char anotherValue;
};
```

## Enums

- Enums should be named using UpperCamelCase.
- Individual enum values should be named using UPPER\_CASE\_WITH\_UNDERSCORES.
- Individual enum values should be prefixed with a name reflecting the name of the enum to avoid polluting the global namespace:

```
enum ComputerType {  
    COMPUTER_TYPE_MAC,  
    COMPUTER_TYPE_PC  
};
```

## Types

Woopsi exclusively uses a set of 6 defines mapped to the standard types listed below.

- u8 - unsigned char
- u16 - unsigned short
- u32 - unsigned int
- s8 - signed char
- s16 - signed short
- s32 - signed int

The “char” type is used when dealing with characters or character arrays. Literal strings must be “const char”.

## Coding for the DS

The DS' hardware imposes a number of limitations upon the programmer. If they are known in advance they can effectively be worked around.

## Developing for Embedded Systems – The C++ Subset

The DS is essentially an embedded system. Compared with modern desktop systems, or even home consoles, it has miniscule CPU power and very limited RAM. Meanwhile, C++ is an extraordinarily large and powerful programming language. Some concessions have to be made in order to effectively use C++ as a language with which to program an embedded system.

Woopsi uses a very limited subset of C++. For example, the following features are not used:

- STL (strings, streams, containers, etc)
- RTTI
- Exceptions

Various techniques that are common in desktop programming are ignored. The result of `malloc()` is not checked, for example. If `malloc()` fails there is no way to inform the user of the error, so there is little point in wasting CPU cycles in checking it. Though it makes sense for standard programs to use STL classes for strings or vectors, they haul in so many other dependencies that they have a significant impact on the size of the resultant ROM. RTTI and exceptions both have negative impacts on the performance of C++ code.

## Stack Limits

The DS' stack is limited to **16 kilobytes**. This severely limits the amount of information that can be stored on the stack. Since both C and C++ use the stack as a way to store function arguments, a complex chain of function calls (such as a deeply recursive function) can cause a stack overflow crash. It is highly recommended that large data structures are stored on the heap, by the appropriate use of either “new” or “malloc()”, instead of the stack.

## RAM Size

It is not just the stack that is limited on the DS. The DS has **4MB of RAM** in total. Whereas traditional consoles map cartridge ROM into RAM as an additional region, the DS loads the cartridge into the available RAM. Traditional consoles therefore have a guaranteed amount of available RAM regardless of the size of the ROM, whilst the DS' available RAM is equal to the initial 4MB *minus* the size of the loaded ROM. The larger the ROM, the less working RAM is available at runtime. It is therefore very important that ROM sizes are kept to a minimum. This is the primary motivation for excluding the C++ STL from Woopsi.

## RAM Emulation Difficulties

It is important to note that most emulators do not accurately model the DS' memory. They frequently initialise memory to zero whereas the real hardware leaves it in an indeterminate state. They do not limit the stack to 16K, and they do not allocate memory in the same way. Referencing a NULL pointer, referencing deleted memory or overflowing the stack will typically crash a DS, whilst an emulator may continue running normally.

## Floating-Point Maths

The DS does not support floating-point maths. Fixed-point maths can, of course, be used.

## Datatypes and Speed

The DS uses two 32-bit ARM CPUs. Their native data size is, therefore, 32-bits wide. They perform at optimal speed when fed 32-bit data. Woopsi uses the full range of data sizes afforded by the subset of C++ used (char, short and int) to improve the clarity of the Woopsi API, but this comes at a small cost to performance.

## Memory Copying

The DS features a multitude of ways to copy data around. At its most basic, programmers can use a simple for/next loop or the standard memcpy() function. It also provides two BIOS functions – swiCopy() and swiFastCopy() – at least one of which suffers from a bug that cripples its speed. Another possibility is the DMA hardware, which is exceptionally fast when copying to VRAM, but is exceptionally slow when copying to RAM.

To complicate matters further, the DS' main CPU, an ARM9, has an on-chip cache. The DMA cannot use the cache, so it must be flushed before the DMA is used to ensure that the copy occurs correctly.

The best choice for any given situation is dependent on a variety of factors. An in-depth discussion of the pros and cons of each method falls outside the scope of this document, but it is important to be aware of the issues when trying to write optimal data copying routines.

**TODO: CHANGE THIS SO THAT IT INCLUDES DETAILS OF COPYING METHODS – SPEEDS, BUGS, ETC.**

## Woopsi's Hardware Abstraction Layer

Woopsi abstracts away the details of the underlying DS hardware. When writing an application using Woopsi, there is rarely any reason for a programmer to need to interact with the DS hardware at all. In fact, the information given in this chapter will most likely be irrelevant to the majority of Woopsi users.

It is, however, possible for programmers to use any low-level functions they desire. This could include functionality from libnds, custom hardware-hitting code or even inline ARM assembly. Using Woopsi does not preclude this level of interaction with the DS. However, accessing the hardware directly will introduce difficulties into porting any Woopsi application to another platform.

## Hardware Initialisation

All hardware initialisation is handled by Woopsi. Woopsi initialises the 2D hardware and backgrounds for the LCD screens. It creates a vblank callback function and creates structs to track the state of the console's various input methods. This should suffice for the majority of Woopsi projects.

If other hardware is needed, such as 3D capabilities, microphone or WiFi, this must be handled in user code.

## Framebuffer

The framebuffer is a region of memory that is directly mapped to a physical display. Changes made to the framebuffer are immediately drawn to the display. Framebuffers are one of the simplest ways to present information on a display.

Woopsi initialises two framebuffer objects in its setup routine, which can be accessed via the global array "frameBuffer". The framebuffer objects provide abstracted read and write access to the DS' displays. Each framebuffer object is an instance of the FrameBuffer class, which forms a controlled wrapper around the underlying DS video memory.

Developers are strongly advised not to manually work with the framebuffer objects. Interaction with the framebuffer should be performed via an instance of the "GraphicsPort" class, which is discussed later.

## DMA Hardware

The DS includes "direct memory access" (DMA) hardware. This is designed to copy memory around at speed without the need for the CPU to become involved. Woopsi makes heavy use of the DMA hardware as a way of optimising its graphical output.

Working with the DMA hardware is rather complex, not least because of the problems discussed in chapter 4. To alleviate these problems, Woopsi provides two global functions:

woopsiDmaCopy() and woopsiDmaFill(). The former will copy memory around using the DMA, if possible, whilst the second will fill memory with a given value.

These two functions are solely designed to work with graphical data.

## Vertical Blanks

The woopsiWaitVBL() function syncs with a vertical blank of the DS' displays. However, Woopsi calls this function automatically. Attempting to manually work with VBLs may have strange effects.

## SDL Capabilities

Blah

# Woopsi Utility Classes

Woopsi provides a number of classes for dealing with low-level concepts such as strings, collections and dates.

## String Handling

Woopsi does not use the STL string class. The primary reason for this is the amount of dependencies that the string class has on other heavyweight C++ features, most notably exceptions. Instead, Woopsi uses a homegrown `WoopsiString` class. The entire Woopsi API uses `WoopsiStrings` for working with textual data. Additionally, Woopsi includes a `Text` class that includes features such as wrapping.

### The `WoopsiString` Class

---

The `WoopsiString` can be thought of as a hybrid between a traditional immutable string and the `StringBuilder` from Java and C#. It is intended to represent string data, which makes it like a typical string, but it can also be used to manipulate that data in ways more usually available to `StringBuilder`-like classes.

The `WoopsiString` class provides a mutable string to which text can be appended, inserted or removed. It is mutable to reduce the amount of data copying that must be done when working with immutable strings. The class performs some pointer and memory gymnastics to reduce the amount of memory management that is performed internally when the strings are altered.

Here is an example of the `WoopsiString` class in use:

```
// Create a string
WoopsiString* myString = new WoopsiString("This is some text");

// Append some text to the end of the string
myString->append("\nSome more text");

// Free the string object
delete myString;
```

The `WoopsiString` class uses the UTF-8 encoding to store Unicode data.

### The `StringIterator` Class

---

The `WoopsiString` class supports unicode, and represents its data using the UTF-8 encoding. This is a variable-length encoding system, in which each character can be either 1, 2, 3 or 4 bytes wide. There is no way of knowing where a given character index appears within the data other than by starting at one end of the string and consuming characters until the desired index is reached.

This poses some problems. Printing the 10 middle characters in a 1,000 character long string, for example, will involve iterating over roughly 500 characters 10 times in order to extract each one.

The `StringIterator` class is the solution to this performance-sapping dilemma. It can iterate over a `WoopsiString` much more efficiently than attempting to do so manually. It remembers

its position within the string, so retrieving sequences of characters are very efficient. It will even determine the shortest route to reach a given character index when jumping to a new location in the string, either by iterating from the start, end or the current location.

Using the StringIterator is very simple. Here's a short example:

```
// Create a string to iterate over
WoopsiString hello = "hello";

// Get a new iterator object from the string
StringIterator* iterator = hello.newStringIterator();

// Read out the 3rd, 4th and 5th characters from the string
iterator->moveTo(2);
u32 thirdChar = iterator->getCodePoint();
iterator->moveToNext();
u32 fourthChar = iterator->getCodePoint();
iterator->moveToNext();
u32 fifthChar = iterator->getCodePoint();

// Clean up
delete iterator;
```

When iterating over a string, the StringIterator class should always be used in preference over the WoopsiString::getCharAt() method unless only one character is being extracted.

## The Text Class

---

The Text class builds upon, and indeed inherits from, the WoopsiString class. It maintains a view of a string that has been formatted to fit within a column of a particular width. Like the WoopsiString class, its text can be appended to, inserted into or removed from. However, the Text class is designed to aid with textual output rather than be used solely for storage and manipulation. It is a presentation class more than a data container.

The Text class formats its text to fit within a particular width based on the size of the font that will be used to render it. The spacing between lines can be specified. The class enables users to get the length of individual lines, with or without trailing white space, in either characters or pixels. It can produce pointers to individual lines within the text, the width of the longest line of text (in pixels) and the height of each line (in pixels). It can even locate the line of text that contains a particular character index.

## Collections

Woopsi does not use the STL collections. Since working with C-style arrays is cumbersome, and as re-implementing basic container classes is tedious, Woopsi provides a vector replacement "WoopsiArray" class.

## The WoopsiArray Class

---

The WoopsiArray class is essentially a slimmed-down version of the STL vector class. It does not include support for STL iterators, as the STL is off-limits, so it does not include the vector::begin() and vector::end() methods. Instead, items within the array are retrieved by using WoopsiArray::at() or the overloaded [] operator.

# Dates

## The Date Class

---

The Date class is primarily used by the Calendar gadget. It includes all of the necessary functionality for working with dates. Date objects are created by providing a day, month and year to the constructor. Once created, the objects can be used to identify the name of the date's month and day, the day of the week it represents, the number of days in the year and whether or not it is a leap year.

The date can be altered either by adding days, months or years, or by resetting the entire date to a new value. Date objects can be compared to each other using the overloaded equality and inequality operators.

# Graphics Classes

As a user interface library, Woopsi deals extensively with graphics. To make this easier, the library includes a wealth of classes designed to simplify working with bitmaps, drawing to bitmaps, working with fonts and working with animations. These classes are available for Woopsi developers to use in their own applications.

## Bitmaps

### The BitmapBase Class

---

The BitmapBase class is the most fundamental class for working with bitmaps in Woopsi. It contains the most basic information necessary to describe an immutable bitmap – width, height, and a way to retrieve a pointer to its internal array of u16 data. It also provides a way to retrieve the colour of a pixel at a given set of co-ordinates.

The class itself does not implement these last two methods – they are pure virtual functions - so it must be subclassed in order for an instance to be created. This means that the class can store its bitmap data in any way that it needs to. Although any bitmap must expose its data as a 16-bit array, internally it can store the data in 8-bit or 32-bit formats, or even as an SDL surface. This possibility is used by the FrameBuffer class.

The Woopsi API can treat any class that inherits from BitmapBase as a bitmap.

### The BitmapWrapper Class

---

The BitmapWrapper class inherits from BitmapBase and provides an immutable bitmap class that can be instantiated. It does not allocate any internal memory on which to store a bitmap. Instead, it expects to be given a pointer to a pre-existing raw u16 array in its constructor. Thus, any bitmaps converted to C code and included in a project can be wrapped inside a bitmap object and treated like any other bitmap.

The destructor for the BitmapWrapper does not automatically delete the data it is told to wrap around. This ensures that no attempt will be made to data marked const, such as that included by converting a bitmap to C code.

Here's an example of the BitmapWrapper in use:

```
// Allocate an array of shorts to use as the bitmap data
u16* myData = new u16[100];

// Wrap the short array inside a BitmapWrapper object
BitmapWrapper* wrapper = new BitmapWrapper(10, 10, myData);

// Delete the wrapper
delete wrapper;

// Delete the data array
delete myData;
```

### The MutableBitmapBase Class

---

Neither the `BitmapBase` nor the `BitmapWrapper` classes provide a way to store mutable bitmap data. If a bitmap needs to be altered, by drawing to it, these classes are of limited use.

Woopsi provides the `MutableBitmapBase` class as a way of defining bitmaps that can be drawn to. It extends the `BitmapBase` with a set of methods designed to change the internal bitmap data:

- `setPixel()`, for setting an individual pixel;
- `blit()`, for copying from one bitmap to another;
- `blitFill()`, for setting a range of pixels in one command.

`MutableBitmapBase` is an abstract class and must be subclassed. Any class that inherits from this class can be used throughout Woopsi as a mutable bitmap.

---

### The FrameBuffer Class

The `FrameBuffer` class is a subclass of `MutableBitmapBase` intended for a single purpose. It forms a wrapper around the DS' framebuffer and allows them to be used in the same way as any other mutable bitmap.

All Woopsi interaction with the DS' framebuffer occurs through instances of the `FrameBuffer` class. Since all framebuffer interaction is therefore entirely centralised, the `FrameBuffer` class can be modified to allow Woopsi to be ported to other platforms. The standard class includes conditional compilation directives that allow it to work with an SDL surface instead of the DS' framebuffer.

---

### The Bitmap Class

The `Bitmap` class is the most generally useful of Woopsi's bitmap classes. It provides a mutable bitmap that creates its own internal data array.

Here's an example:

```
// Create bitmap
Bitmap* bitmap = new Bitmap(100, 100);

// Draw a blue line to the from the left to the right at row 10
bitmap->blitFill(0, 10, woopsiRGB(0, 0, 31), 100);

// Delete the bitmap
delete bitmap;
```

## Bitmap Manipulation

The mutable bitmap classes include only the most basic functions for drawing – pixel plotting and blitting of rows of pixels. These are clearly insufficient for creating complex interfaces or drawing interesting designs to bitmaps. As a way of keeping the bitmaps lightweight, all of the advanced drawing functionality is stored in a separate “Graphics” class.

---

### The Graphics Class

The Graphics class provides the ability to perform complex drawing operations to an instance of any class that inherits from MutableBitmapBase. Its drawing functions include:

- `getPixel()`, to get the colour of a pixel at a given set of co-ordinates;
- `drawPixel()`, to set the colour of a pixel at a given set of co-ordinates;
- `drawRect()`, to draw the outline of a rectangle;
- `drawFilledRect()`, to draw a filled rectangle;
- `drawHorizLine()`, to draw a horizontal line;
- `drawVertLine()`, to draw a vertical line;
- `drawLine()`, to draw a line in any direction;
- `drawCircle()`, to draw the outline of a circle;
- `drawFilledCircle()`, to draw a filled circle;
- `drawEllipse()`, to draw the outline of an ellipse;
- `drawFilledEllipse()`, to draw a filled ellipse;
- `drawText()`, to print a string;
- `drawBitmap()`, to draw a second bitmap (or portion of it) to the current bitmap;
- `drawBitmapGreyScale()`, to draw a second bitmap in greyscale to the current bitmap;
- `drawXORPixel()`, to XOR a pixel at a given set of co-ordinates;
- `copy()`, to copy one rectangular area of the bitmap to another;
- `dim()`, to halve the brightness of a rectangular area of the bitmap;
- `greyScale()`, to convert a rectangular area of the bitmap to greyscale.

All of these functions are clipped to a rectangular region which can be specified by using the `Graphics::setClipRect()` method. If this region is not set, it will default to the dimensions of the bitmap. This ensures that the drawing methods do not attempt to draw outside the confines of the bitmap, which could result in graphical corruption at best, or crashes at worst.

The standard Bitmap class includes a method for creating a new Graphics object that can draw to the bitmap. Here is an example:

```
// Create bitmap
Bitmap* bitmap = new Bitmap(100, 100);

// Get graphics object
Graphics* gfx = bitmap->newGraphics();

// Draw a circle to the bitmap
gfx->drawCircle(50, 50, 10, woopsiRGB(0, 10, 0));

// Delete the graphics object
delete gfx;

// Delete the bitmap
delete bitmap;
```

The circle could be made any size, within the constraints of the integer datatypes used to represent its radius, and it would not be drawn past the boundaries of the bitmap.

Note that it is essential that the Graphics object is deleted once it is no longer needed in order to avoid a memory leak.

## The GraphicsPort Class

---

The GraphicsPort class uses the Graphics class to draw to any number of clipping rectangles. It is primarily used when drawing within a Woopsi user interface gadget.

## Animation

### **The Animation Class**

---

Blah

## Fonts

### **The FontBase Class**

---

Blah

### **The FixedWidthFontBase Class**

---

Blah

### **The Font Class**

---

Blah

### **The MonoFont Class**

---

Blah

### **The PackedFontBase Class**

---

Blah

### **The PackedFont1 Class**

---

Blah

### **The PackedFont16 Class**

---

Blah

# Gadgets

The most basic element of a user interface, in Woopsi parlance, is the “gadget”. Other UI systems call these “widgets” or “components”. Windows, buttons and textboxes are all examples of a Woopsi gadget.

All of the essential functionality of a gadget is contained within the Gadget class. All Woopsi UI components inherit from this class. Subclasses of the Gadget class can extend the base class’ functionality by providing drawing routines to render the gadget, customised behaviours when a user interacts with the gadget, and so on.

Woopsi provides a suite of gadgets that are ready for use. These can be customised by subclassing and overriding their behaviours. Entirely new gadgets can also be created.

Gadgets fall into one of two categories: **basic** and **compound**.

A compound gadget is a gadget made of other gadgets. For example, the ScrollingTextBox, which is a textbox with a vertical scrollbar, comprises of a MultiLineTextBox gadget and a ScrollbarVertical gadget. The Alert gadget, which shows a message and an “OK” button, is made out of an AmigaWindow, a TextBox and a Button.

A basic gadget, on the other hand, is a gadget consisting of just one gadget. The Button is an example of a basic gadget.

## Buttons

Buttons are one of the most basic features of a user interface. Woopsi provides several variations.

### The Button

---

The most basic Woopsi button class is called, obviously enough, “Button”. It shows a user-defined string inside a rectangular box. Its edge is bevelled outwards until it is clicked and held down, at which point the box is bevelled inwards. Clicking the button fires an ACTION event.

**TODO: Picture of a button here**

**TODO: Example usage**

### The BitmapButton

---

The second variety of button that Woopsi provides is the “BitmapButton”. Instead of containing a string, BitmapButtons display images instead. They show one image when clicked and another image when not. Like the Button, they fire an ACTION event they are clicked.

**TODO: Picture here**

**TODO: Example usage**

## The AnimButton

---

AnimButtons are similar to BitmapButtons, but instead of showing a static image they show an animation. They show one animation when clicked and another when not. They fire an ACTION event when clicked.

**TODO: Picture here**

**TODO: Example usage**

## The CycleButton

---

The CycleButton is, perhaps, more of a list gadget that has been miscategorised than a button. A button's purpose is to raise an event when it is clicked, and that is true of the CycleButton, but it has a second purpose. The CycleButton contains a list of options that can be added to and removed from. The text it displays is pulled from the currently-selected option.

Clicking the button selects the next option in the list or, if no more options are available, the first option. The value of the currently selected option can be retrieved using the gadget's `getValue()` method.

Example usage:

```
// Create a cycle button
CycleButton* button = new CycleButton(10, 10, 100, 30);

// Add options
button->addOption("Option 1", 1);
button->addOption("Option 2", 2);
```

**TODO: Picture here**

## The DecorationGlyphButton

---

The DecorationGlyphButton is used by the AmigaScreen and AmigaWindow gadgets as a way of adding buttons to their borders. It is not typically useful outside of this context.

# Text Containers

Blah

## The Label

---

The Label class is the most basic way of getting text onto the screen. A Label gadget can display a single line of text at a given set of co-ordinates. The text within the label can be aligned horizontally to the left, right or centre of the gadget, and aligned vertically to the top, bottom or centre.

A limitation of the Label is that it does not scroll. Though the string it contains can be any

length, limited only by available RAM, only the portion of the string that fits within the label will be displayed.

Example usage:

```
// Create the label
Label* label = new Label(10, 10, 100, 30, "Some text");
```

## The TextBox

---

The TextBox extends the Label with facilities to allow users to edit the string within the gadget. It can display a cursor that the user can move either by using the DS' d-pad or by clicking the TextBox with the stylus. Double-clicking the TextBox will automatically pop up a keyboard allowing the user to type into the gadget, though this behaviour can be disabled.

Unlike the Label, the contents of the TextBox scrolls to follow the cursor.

Example usage:

```
// Create the textbox
TextBox* textbox = new TextBox(10, 10, 100, 30, "Some
text");
```

## The MultiLineTextBox

---

Though the TextBox can scroll horizontally to follow the cursor, allowing more text to be visible than the Label, it is still limited to a single line of text. The MultiLineTextBox provides a textbox that can contain multiple lines of text. Scrolling through the text is achieved by dragging the contents of the textbox with the stylus up or down. The text is automatically wrapped to fit the width of the textbox.

Like the TextBox, the MultiLineTextBox provides a cursor that can be moved through the text via stylus taps or the d-pad. It can be hidden if the textbox is to be used for display only. When double-clicked, it opens up the keyboard. Again, this functionality can be disabled.

**TODO: Discuss the way the textbox forgets the topmost rows. Should it be possible to make the textbox remember *\*all\** rows?**

**TODO: Example usage**

**TODO: Picture**

## The ScrollingTextBox

---

The ScrollingTextBox is a compound gadget consisting of a MultiLineTextBox and a ScrollbarVertical. It functions in the same way as a MultiLineTextBox, but includes a scrollbar on the right of the textbox.

**TODO: Picture**

**TODO: Example usage**

# Basic Program Structure

A basic Woopsi application consists of the following:

- A directory, which has the same name as the application itself and contains:
  - A makefile;
  - A subclass of the Woopsi class that implements its startup() and shutdown() methods;
  - A main.cpp file that instantiates the Woopsi application subclass.

## Creating a Simple Program

### Folders and Makefiles

---

To understand how to put a basic Woopsi application together, let's consider one of the simplest examples – a “hello world” program. This application will:

- Open a screen;
- Add a window to the screen;
- Add a textbox to the window containing the text “Hello world!”

Firstly, create a folder called “woopsitest” somewhere on your hard disk. The path that contains this folder cannot contain any spaces. devkitARM does not work correctly when any directories above the sourcecode contain spaces.

Once this folder has been created, copy the “makefile” file from the folder “/woopsi/Woopsi/template” into your new “woopsitest” folder.

Laslty, create a new folder in “woopsitest” called “source”.

At this point, we have set up the directory structure necessary for a Woopsi application. The structure should look like this:

- woopsitest/
  - source/
  - makefile

### Boilerplate Code

---

Next, we will create textfiles containing the functionality that every Woopsi application needs in order to start up and shut down.

Open a text editor and paste in the following code:

```
#include "woopsitest.h"

int main(int argc, char* argv[]) {
    WoopsiTest app;
    return app.main(argc.argv);
}
```

This code creates an instance of the “WoopsiTest” class, which we will define in a moment. It then starts that instance running by calling its “main()” method. This is boilerplate code that we will use in every Woopsi application we write.

Save this file to the “source” folder with the name “main.cpp”.

Next, create a blank text document and paste in this code:

```
// Prevent this file from being included multiple times.
// This can happen frequently in large projects.
#ifndef _WOOPSI_TEST_H_
#define _WOOPSI_TEST_H_

// Make the code from the “woopsi.h” file available within this
// file. That file includes the definition of the Woopsi
// class.
#include “woopsi.h”

// Tell the compiler that we’re working within the WoopsiUI
// namespace. That namespace includes all of the Woopsi
// classes.
using namespace WoopsiUI;

// Create a new class called “WoopsiTest” that inherits from
// the Woopsi class.
class WoopsiTest : public Woopsi {
private:

    // The class includes two methods - startup and shutdown.
    void startup();
    void shutdown();

};

// End the #ifndef we started at the top of the file
#endif
```

This code defines the structure of a “WoopsiTest” class. It is the class we instantiated in the previous file. Save it to the “source” folder with the name “woopsitest.h”.

Our directory structure should now look like this:

- woopsitest/
  - source/
    - main.cpp
    - woopsitest.h
  - makefile

## A Basic Woopsi Program

---

We have one file left to create. This file contains all of the program’s real functionality. Create another blank text document and paste in this code:

```
// Include all code from the “woopsitest.h” file.
#include “woopsitest.h”

// The startup() method is called when Woopsi starts running.
// It should contain all user interface setup code.
void WoopsiTest::startup() {
```

```

        // Enable drawing. Drawing is initially disabled to
        // enable the GUI to be built without it being drawn
        // to the screen in sections.
        enableDrawing();

        // Redraw the display.
        redraw();
    }

    // The shutdown() method is called when Woopsi closes down.
    // All user interface teardown code should be in here.
    void WoopsiTest::shutdown() {

        // Call the base method.
        Woopsi::shutdown();

    }

```

Save it in the “source” folder with the name “woopsitest.cpp”.

The final directory structure looks like this:

- woopsitest/
  - source/
    - main.cpp
    - woopsitest.h
    - woopsitest.cpp
  - makefile

Now we’re ready to compile this application and see what it does. Open a command line and CD to the “woopsitest” directory. Type “make” and you should see something like this:

```

Compiling woopsitest.cpp
Linking...
Nintendo DS rom tool 1.46 - Nov 30 2009
by Rafael Vuijk, Dave Murphy, Alexei Karpenko
Built: woopsitest.nds
dsbuild 1.21 - Nov 21 2009
using default loader
Built: woopsitest.ds.gba
Built: woopsitest.sc.nds

```

Open the “woopsitest” directory and you should find a new directory called “Release”. Open it and load the “woopsitest.nds” file into your emulator. You will see something that looks like this:



**TODO: This image is no longer correct – top should be grey**

This is what Woopsi looks like before any gadgets, or user interface components, are added to it.

## Adding a Screen

---

Quit the emulator and open the “woopsitest.cpp” file again so that we can make it more interesting. Delete its current contents and replace it with this code:

```
#include "woopsitest.h"

// Include the amigascreen.h file.
#include "amigascreen.h"

void WoopsiTest::startup() {

    // Create a screen that will contain all other gadgets.
    AmigaScreen* screen = new AmigaScreen(
        "Hello World Screen",
        Gadget::GADGET_DRAGGABLE,
        AmigaScreen::AMIGA_SCREEN_SHOW_DEPTH |
        AmigaScreen::AMIGA_SCREEN_SHOW_FLIP);

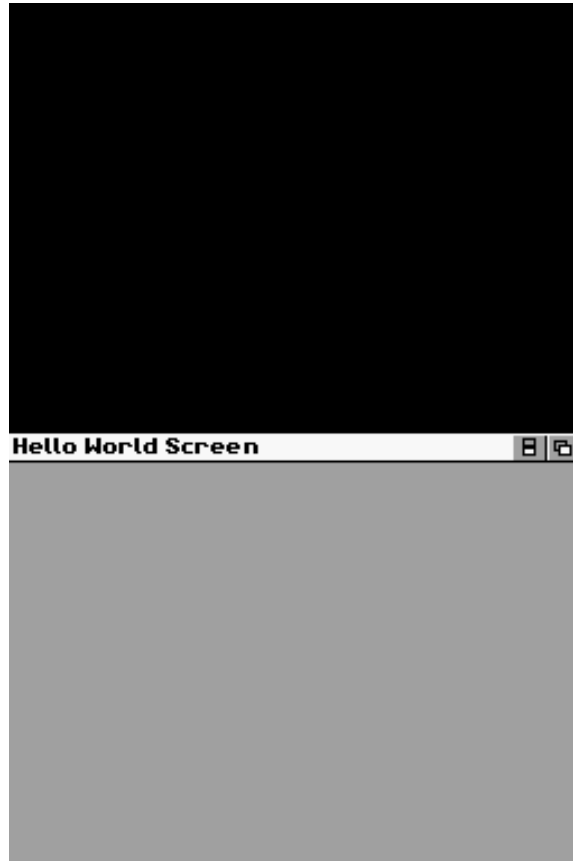
    // Add the screen to the user interface.
    addGadget(screen);

    enableDrawing();
    redraw();
}
```

```
void WoopsiTest::shutdown() {
    Woopsi::shutdown();
}
```

Compile this new version in the same way as the last, by using the command line and the “make” command. Open the resultant woopsitest.nds file, the “ROM”, in your emulator.

This time, you will see this:



**TODO: This image is no longer correct – top should be grey**

Whereas we previously created a user interface without any gadgets on it, this time we have created a user interface containing an “AmigaScreen” gadget. Screens are very basic gadgets that are simply a space for other gadgets to exist. The AmigaScreen is a specialised Screen gadget designed to look like the user interface from the Commodore Amiga computer. It consists of three “child” gadgets:

- The title gadget displays the name of the screen;
- The button at the top-right moves the visible screen to the bottom of the screen “stack”, exposing the screen beneath it;
- The remaining button causes the screen to flip to the top physical display, assuming there is more than one screen in the stack.

The AmigaScreen’s constructor takes three arguments. The first is obviously the name of the screen. If you change this and recompile the project, you will find that the text displayed on the screen changes to match.

The second argument defines some very basic properties of the screen. If you click on the title bar with your mouse and drag it downwards, you’ll find that the screen can be dragged up and down. Changing this second argument to zero will prevent the screen from being dragged. This argument is known as the “gadget flags” argument, and is common amongst most gadgets. More on this later.

The last argument is another flags-style argument. In this argument, we tell the screen that we want it to show both the display-flipping button (AMIGA\_SCREEN\_SHOW\_FLIP) and the depth-swapping button (AMIGA\_SCREEN\_SHOW\_DEPTH). If you remove one of these from the argument and recompile, you will find that one of the buttons disappears. Replace the argument with a zero and both buttons will disappear.

## Adding a Window

---

Unfortunately, this still isn't a very exciting demo. Let's add a window so that we have something to play with. Open the "woopsitest.cpp" file and replace its code with the following:

```
#include "woopsitest.h"
#include "amigascreen.h"

// Include the "amigawindow.h" file.
#include "amigawindow.h"

void WoopsiTest::startup() {

    AmigaScreen* screen = new AmigaScreen(
        "Hello World Screen",
        Gadget::GADGET_DRAGGABLE,
        AmigaScreen::AMIGA_SCREEN_SHOW_DEPTH |
        AmigaScreen::AMIGA_SCREEN_SHOW_FLIP);

    addGadget(screen);

    // Create the window and add it to the screen.
    AmigaWindow* window = new AmigaWindow(10, 30, 100, 50,
        "Hello World Window",
        Gadget::GADGET_DRAGGABLE,
        AmigaWindow::AMIGA_WINDOW_SHOW_CLOSE |
        AmigaWindow::AMIGA_WINDOW_SHOW_DEPTH);

    screen->addGadget(window);

    enableDrawing();
    redraw();
}

void WoopsiTest::shutdown() {
    Woopsi::shutdown();
}
```

When you compile this example, you will get this result in your emulator:



**TODO: This image is no longer correct – top should be grey**

Try clicking on the window, then on the screen. When you click on the window, its border will turn blue; this shows that it is the currently active gadget and has “focus”. When you click on the screen, the window loses focus and its border becomes grey again.

You can drag the window around by clicking on its title bar and moving it. The button on the top-left of the window will close it, whilst the button on the top-right will move it to the bottom of the window stack if there is more than one window on the screen.

If you try dragging the screen down again, by clicking on its title, you’ll find that the window is dragged down along with it. The screen “owns” the window, and repositioning the screen will reposition any gadgets that it owns. A gadget that is owned by another is called its **child**, whilst the owning gadget is called the **parent**. In this situation, the window is a child of the screen, which is in turn a child of the WoopsiTest object. Ownership of gadgets is established via the “addGadget( )” method used to add the screen to the WoopsiTest object, and the window to the screen.

The arguments passed to the window’s constructor should look familiar. The same GADGET\_DRAGGABLE flag is passed as the gadget flags argument, and very similar values are passed for the last argument. In this case, they show the close and depth buttons, instead of the flip and depth buttons used by the screen. The only novel arguments are the four numeric parameters. These correspond to the x and y co-ordinates of the window and its width and height respectively. Try playing around with these values to see how they affect the window. Note that the co-ordinates are relative to the screen that contains the window, not the physical display itself.

## **Adding a TextBox**

---

The final gadget we need to add to our application is a textbox. Open the “woopsitest.cpp” file again and replace its code with this:

```
#include "woopsitest.h"
#include "amigascreen.h"
#include "amigawindow.h"

// Include the "textbox.h" file.
#include "textbox.h"

void WoopsiTest::startup() {

    AmigaScreen* screen = new AmigaScreen(
        "Hello World Screen",
        Gadget::GADGET_DRAGGABLE,
        AmigaScreen::AMIGA_SCREEN_SHOW_DEPTH |
        AmigaScreen::AMIGA_SCREEN_SHOW_FLIP);

    addGadget(screen);

    AmigaWindow* window = new AmigaWindow(10, 30, 100, 50,
        "Hello World Window",
        Gadget::GADGET_DRAGGABLE,
        AmigaWindow::AMIGA_WINDOW_SHOW_CLOSE |
        AmigaWindow::AMIGA_WINDOW_SHOW_DEPTH);

    screen->addGadget(window);

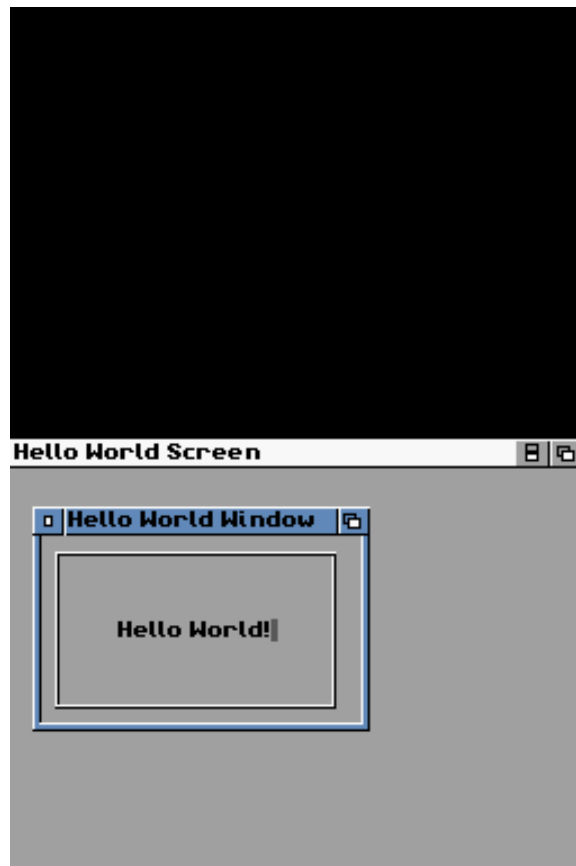
    // Add a textbox to the window
    TextBox* textbox = new TextBox(10, 20, 125, 70,
        "Hello World!");

    window->addGadget(textbox);

    enableDrawing();
    redraw();
}

void WoopsiTest::shutdown() {
    Woopsi::shutdown();
}
```

This last step will leave you with a ROM that looks like this when run:



**TODO: This image is no longer correct – top should be grey**

The textbox's constructor receives its x and y co-ordinates, its width and height, and the text to display as its parameters. As with the window, and indeed all gadgets, the co-ordinates are relative to the textbox's parent gadget, which in this case is the window.

The small grey box at the end of the text is a cursor. If you click on the text within the textbox you will find that the cursor jumps to the clicked character within the text. You can also move it by using the d-pad left and right buttons. If you double-click the textbox the screen will jump up to the top DS display and a keyboard will appear at the bottom. You can type into the textbox using the keyboard.

# Co-ordinate Systems

## Woopsi Space

In Woopsi-space, the top-left pixel of the bottom physical screen has the co-ordinate (0,0). Its bottom-right pixel has the co-ordinate (255,191). The top display has the co-ordinates (0,512) to (255,703).

Graphically:

-----	0	The bottom display is the main screen,
Bottom		which is why co-ords start at (0,0)
Display		
-----	192	Deadzone exists to enable screens to be dragged
Deadzone		downwards without overlapping gadgets in bottom
		screen.
-----	512	The top display starts at 512 so that it is
Top		aligned to a y co-ordinate that is a power of 2.
Display		
-----	704	There is no second deadzone as there are no more
		lower screens to overlap.

## Framebuffer/Bitmap Space

In this co-ordinate system, (0,0) is the top-left pixel of the framebuffer and (255,191) is the bottom-right pixel. This co-ordinate system is used for both the top and bottom framebuffers.

Woopsi treats the framebuffers as just another kind of mutable bitmap, which means that this co-ordinate system applies to any bitmap, not just the framebuffers. The only difference is that other bitmaps can be any size, memory allowing. Only the framebuffers are limited to 256x192 pixels.

## Gadget Space

In gadget space, the top-left corner of the gadget has the co-ordinate (0,0). The bottom-right corner has the co-ordinate (width - 1, height - 1). Width and height change depending on the size of the gadget.

Each gadget defines its own space. Imagine we have a GUI consisting of a screen, which contains a window, which contains a button. The screen's gadget space begins at (0,0), which is the top-left corner of the screen. The window's gadget space begins at (0,0), which is the top-left corner of the window. Lastly, the button's gadget space begins at (0,0), which is - obviously enough - the top-left corner of the button.

When adding a gadget to another gadget, the child is added to the parent's gadget space. Thus, its co-ordinate should be specified relative to the gadget space of the parent. If the

window in the previous scenario should exist at co-ordinates (10,15) relative to the screen, it should be created using the code:

```
Window* window = new Window(10, 15, ... );
```

## GraphicsPort Space

A GraphicsPort internally works in Woopsi space. When drawing to a framebuffer, it converts its co-ordinates from Woopsi space to framebuffer space. Externally it works in GraphicsPort space. When a GraphicsPort is created, the area in which it can draw is specified as part of its constructor arguments. These are supplied in Woopsi co-ordinates; the `Gadget::newGraphicsPort()` and similar methods handle this automatically. Once it has been created, all drawing arguments passed to its drawing functions must be passed in GraphicsPort space, which is defined relative to the drawing area defined in the parameters passed to its constructor.

If a GraphicsPort is created like this, it can draw across the entire visible surface of a gadget:

```
GraphicsPort* port = new GraphicsPort(getX(), getY(), _width,
    _height, isDrawingEnabled(), ... );
```

This code would result in a GraphicsPort that cannot draw to a 10 pixel wide column on the left-hand side of the gadget' surface:

```
GraphicsPort* port = new GraphicsPort(getX() + 10, getY(),
    _width - 10, _height, isDrawingEnabled(), ... );
```

In both circumstances, GraphicsPort space starts at (0,0). The difference is that the first example results in a GraphicsPort in which GraphicsPort space is identical to Gadget space. In the second example, GraphicsPort space is offset from Gadget space by 10 pixels along the x axis, and is 10 pixels thinner.

## Converting Between Co-ordinate Systems

To obtain the co-ordinates of a gadget relative to its parent (ie. expressed as the co-ordinates of the gadget within its parent's gadget space), use the gadget's `_x` and `_y` values.

To obtain the co-ordinates of the gadget within the overall Woopsi space, use the `getX()` and `getY()` methods. These recurse up through the gadget tree until the top is reached, summing the co-ordinates of each gadget.