Pyromaniac II

The Sequel

Gerph, November 2021

Pyromaniac II

The Sequel

Gerph, November 2021

Good evening. This presentation is about the few things I've done on Pyromaniac over the last year. I hope you all find it interesting.

0. Introduction

0. Introduction

Introduction

How I'll do this talk

- Some talk about how RISC OS does things.
- There are 9 sections, with questions spread within them.
- Slides will be available at the end, together with some other resources.
- At the conclusion I'll answer any questions people have for as long as people want.

Introduction

How I'll do this talk

There's going to be a lot of talk about what goes on inside RISC OS. I suspect that most people won't have thought about what happens behind the scenes, so I'll try to explain this as well as I can.
The presentation is split into a few sections. Between some of the sections, I'm going to have a short break to answer questions on that section. This should give us a chance to talk about that section's topics without getting too far ahead and people being completely lost.
Please feel free to use the chat to ask questions as we go. I will try to pay attention to it and answer anything that comes up as we go.
At the end of the presentation, these slides and a bunch of links to resources will be made available.

Introduction

What I'll talk about

1. Some background.
2. VNC and Sprites (and questions).
3. Font Manager and Screen Modes.
4. Documentation (and questions).
5. Testing.
6. Miscellaneous bits.
7. System demo (and questions).
8. Conclusions.

Introduction

What I'll talk about

There are quite a few sections this time.
There will be three demonstrations - one demo of the VNC systems, a some examples of the documentation, and a longer system demonstration at the end.
And after my conslusion, I'll take questions for as long as people want.

3 / 121

1. Background

1. Background

Let's start by giving some background before we get into the meat of the presentation...

Background

Who am I?

- A RISC OS architect and engineer.
- My day job is working with test and build systems.
- In previous times did a lot of things with RISC OS, which you can read about on my site if you're interested - gerph.org/riscos
- I know RISC OS inside and out, and I work on it because it's fun.

Background

Who am I?

I'm a RISC OS architect, and in my day job I'm a software engineer. I've worked with all of RISC OS from chip initialisation through to the applications and development tools. I've written about my previous work in much detail in my RISC OS Rambles on my website.
RISC OS development is something that I enjoy, and I returned to working on RISC OS because I find it to be fun and challenging.

Background

Recap

What did I show last year?

- RISC OS Build system, and how it works - build.riscos.online
  - A cloud system for building and testing RISC OS software.
  - Available to all, for free.
- RISC OS Pyromaniac, the operating system that powers it.
  - A reimplementation of RISC OS from scratch in Python.
  - Intended for debugging and testing.
- This RISC OS presentation system which runs on it!
- An online service which demonstrates RISC OS Pyromaniac - shell.riscos.online
- A lot of open source software and resources - pyromaniac.riscos.online

Background

Recap

If you missed the presentation last year, or have just forgotten, I'll summarise what was shown...
Last year I talked about and demonstrated the RISC OS build service, and explained how it worked behind the scenes. The build service itself provides a cloud based way of building and testing RISC OS software without a RISC OS machine. It's available for free to everyone.
I talked about the system that powered it - an entirely new implementation of RISC OS from scratch in Python. RISC OS Pyromaniac provides a command line shell with some graphical capabilities, and runs on Windows, macOS and Linux.
It functioned well enough that the entire slide presentation was running on it - the presentation system that you're watching right now.
I developed the slide system using Pyromaniac and only tested it occassionally on RISC OS Classic - the term RISC OS Classic is how I refer to the original ARM implementation of RISC OS.
I introduced a web site that provides an interactive demonstration of the operating system. And for the presentation the site pyromaniac.riscos.online includes media generated to show off the system, and links to software that has been released.

Background
What I said I was going to look at

Here's what I said I wanted to have a look at:

- More APIs.
- Better handling of corner cases.
- Sprites (sigh).
- Back Trace Structures.
- Finish the pending branches - Windows, Zipper, EasySockets, Git, DCI4, ...
- Using it for actual testing - that was what it was for!
- So many other opportunities.

Background
What I said I was going to look at

At the end of the presentation I said there were a few things that I would like to look at...
I'd like to say that I've got them all done. But no.
I'll talk about the things that I *have* got done, which isn't too far from this. I mostly attacked this year in the same way as the previous year - by doing what seemed fun, and what grabbed me at the time.

2. VNC and sprites

2. VNC and sprites

Let's talk about some stuff!

VNC server

A more accessible system

It would be cool...

- To have the shell server be more accessible.
- To allow you to have a graphical view on the system.
- Maybe pipe the graphics operations directly to a browser canvas.

But that's a lot of work.
How about using VNC instead? - let's write a library.

VNC server

A more accessible system

One of the things I wanted to do was to make the shell server a little more accessible. At the moment there's one shell server and it shares its filing system with all the users. I wanted something that was a bit more flexible so that it can be used by more people as a general online RISC OS server.
It probably wouldn't be that hard, I reasoned, to make the display from the Cairo graphics system available through the browser. Or even to stream the graphics operations to the browser directly, without Cairo.
That would be really interesting and quite cool. But it's a lot of work, and I'm not *that* interested in it. But, I thought, I've got a Cairo surface, so surely I could present that through a VNC server rather than to the desktop?
I looked. There wasn't a simple library that bridges Cairo to VNC. So I wrote one. You'll find it on GitHub as `cairo_vnc`.

VNC server

Writing a new library (1)

The library ...

- Needed to be reusable
- Needed to work with the common VNC clients.
- Needed to be simple enough to be used without much boilerplate.
- Needed to handle multiple concurrent connections.
- Needed to allow some connections to be read only.
- Needed to be able to take input from the user.
- Needed to be able to change the pointer.
- Needed to be able to handle clipboard.

VNC server

Writing a new library (1)

A VNC server has, traditionally, been something I thought 'gosh that's hard to do'. But actually it's really not that hard. The devil is in the detail as always, but getting something that works isn't actually that hard.
Although I wanted the VNC server inside Pyromaniac, I don't believe in building things for one purpose. That's a key requirement when you're working with an operating system. You should never think 'how do I solve this', but instead think 'what is the class of problem I want to solve'. Because you aren't writing something for a specific purpose, but so that others can use it to do things that you hadn't thought of.
Plus, of course, that makes it easier to test. You can run things in isolation and get rid of the bugs without subjecting it to the mess that is your real application - in this case Pyromaniac.
So I started out by writing a server that could be attached to a Cairo surface, without any real knowledge of what the surface shows or the application that runs it. That way I can build it in to Pyromaniac, but I can make it available to other people if they want to use it.
The library was built up from the specification, just stepping through the different sections and implementing the request/response handling as I went. I had a list of requirements for things I wanted it to be able to do.

VNC server

Writing a new library (2)

What do we support?

- Password controls whether you can control the session, or only view.
- Display format can have most RGB formats.
- Only ever supplies data as raw RGB - never compressed...
- ... but it only delivers changing rows.
- Display size changes can be communicated to the client.
- Mouse and keyboard input is supported.
- Multiple simultaneous connections supported.

But ...

- It doesn't support changing the pointer.
- Or the clipboard.

VNC server

Writing a new library (2)

Once I'd implemented the initial negotiations and security checks, I had to handle actual pixel data. There are a lot of encodings available to the VNC servers, but the bitmap data is only handled as RGB in my server - no JPEG encoding here. It does handle arbitrary bit depths up to 8 bit per channel and has a conversion that handles different shifts.
If you want 2 bits of red, 4 bits of green and 2 bits of blue, in the order blue, green, red it should handle it.
The VNC protocol allows a number of ways of encoding the changed data - you can give rectangles of arbitrary size, or copy regions, and you can compress the data with ZLib if you want. There's a bunch of things you can do. But my server only supports the raw encoding.
It also only delivers whole rows at a time. Because that's a lot easier to handle, and I really don't need to do any more. It does optimise out any rows that are unchanged though. But that's about the only efficiency we apply here.
The animator - that's the name I give to the application that's being served - can change the size of their display. On RISC OS this would happen if you changed the mode and the resolution was different. So the server supports this behaviour - not all clients handle it but that cannot be helped.
Input from the user is buffered until the animator is ready for it - so you can interact with what's being displayed. Each client is able to be configured so that it can be read only (cannot give input) or a standard client (which can give input). The animator retrieves the events and can handle them as it sees fit. There's currently 3 events that are available - mouse click/release, key press/release and mouse movement.
Deciding which types of clients are read only or standard is purely down to which password was given.
There's one for read only and one for standard. I would expect that you would use this as a way to present a single session to multiple users.

VNC server

What does it look like?

```
# Create the animator
screen = Screen()
animate_thread = threading.Thread(target=screen.animate)
animate_thread.daemon = True
animate_thread.start()

# Create the server
server = cairovnc.CairoVNCServer(port=5902, surface=screen.surface)
server.serve_forever()
```

VNC server

What does it look like?

I created small examples to go with it, showing how you run your application on one thread, with the VNC server on another thread handling connections.
Assuming you have some Cairo rendering in a simple class called `Screen`, this is the sort of code you could use for a basic animation. You create your animation object, and you run it on a thread. Then you create a `CairoVNCServer` with a bunch of configuration parameters, and you tell it to serve stuff.
In a real program you need a bit more than this, but the examples show how to do this stuff. The image shows what one of those example programs looks like in a VNC client. The bezier curve moves around and the red square moves up and down.
It's very exciting.

VNC server

Integrating with Pyromaniac

How easy was it to integrate with Pyromaniac?

- Not especially hard.
- Getting the mouse input right was surprisingly frustrating.
- Needed multi-threaded locking adding to the library.
- The entire implementation is 317 lines.
- 130 of them are the key mapping table.

VNC server

Integrating with Pyromaniac

Ok, so having got a library, how was easy was it to integrate with Pyromaniac?
Not especially hard really. It's just another graphics implementation around Cairo. So in that respect it's just a case of taking some of the existing code, stripping out all the desktop UI specific bits, and replacing them with VNC specific bits.
The rendering needed to use a locking system to ensure that we don't try to render whilst the library is grabbing the content for a VNC client. This required an update to the library as I had forgotten that feature.
But the entire VNC graphics implementation in Pyromaniac is about 320 lines - and a third of those lines are just the key mapping from VNC to internal key numbers. You'll find the source to the implementation on the pyromaniac site.
Whilst I'd been working on the VNC server, I'd also been trying to add the ability to plot sprites. Because surely that's not so hard?

Sprites

How do you draw sprites?

Let's look at what happens when you put a sprite on the screen in RISC OS.

- Work out what the sprite is (`OS_SpriteOp 18` or use the sprite name).
- Ask for a colour translation table (`ColourTrans_GenerateTable`).
- Request a plot of the sprite (`OS_SpriteOp` in one of its many forms).

Sprites

How do you draw sprites?

Let's look at what happens when you put a sprite on the screen in RISC OS.

- Work out what the sprite is.
  - Usually calling `OS_SpriteOp` to find the sprite address, although you could just give its name.
- Ask for a colour translation table.
  - This is done through `ColourTrans_GenerateTable` (or `SelectTable`).
  - As the graphics system has been extended, these tables are more complex and if you look in PRM 5a you'll find a table of 4 different combinations of translation table formats that can be used.
  - Essentially the table takes the palette entries and produces values that can be written to the screen for them.
  - There's also the possibility of using a colour transfer function - this allows you to change the colour used in the sprite very easily.
- Finally you request a plot of the sprite.
  - One of the `OS_SpriteOp`s like `PlotScaled` is called, passing in the translation table so that the sprite is rendered with the correct colours.
  - `SpriteOp` takes every pixel that's being rendered, looks up the correct value in the translation table, transforms it into the right position and stores that resulting value on the screen.

Sprites

What do you need to draw sprites?

- Graphics primitives, like a graphics cursor and colour selection.
- Modes that are shallow (paletted), and deep (linear colour components).
- Mode information, like the depth and dimensions.
- Palette information for mapping colours in low modes.
- Colour translations, for finding the best colours.
- Sprite area management, like loading and finding the right sprites.
- Sprite area information, to know what sprites are.
- Sprite rendering, to get the sprite on to the screen.

Sprites

What do you need to draw sprites?

Knowing what happens to draw a sprite to the screen, what parts of RISC OS do we need to work in order to do this?
There are quite a few areas which need to work in order to get a sprite on to the screen. Some are pretty simple, but others cover more of the system. It would be nice to think that sprite rendering was just about putting something on the screen, but that's just the pinacle that sits on quite a few structural foundations.
I'll talk about some of these and how they're handled inside Pyromaniac.

Sprites

Drawing sprites in Pyromaniac (1)

Graphics primitives like being able to select colours to draw with, and then drawing lines and text, may not seem like they're necessary for sprite plotting, but they're needed for a few things...

- Colour selection is used for the rarely needed 'plot mask' operations.
- Graphics cursor positioning is needed for some 'plot at cursor' operations.
- Graphics windowing needs to bound any rendered windows.

Sprites

Drawing sprites in Pyromaniac (1)

Graphics primitives like being able to select colours to draw with, and then drawing lines and text, may not seem like they're necessary for sprite plotting, but they're needed for a few things...

- Colour selection is used for the rarely needed 'plot mask' operations. It's not high on the priority list, but it's needed. Most of the GCOL operations were working fine last year - but only in the paletted modes. And then only reliably on 16 colours and below. 256 colour modes are a special beast. So there was a bit of a revamp of the way in which colour selection worked, especially in the 256 colour modes. Lots of tests added here to check that the behaviour matched that of RISC OS Classic.
- Graphics cursor positioning again might now seem like it's necessary, but some of the sprite operations plot the sprite at the graphics cursor. It's a hang over from the BBC Master. Fortunately, this was pretty much complete last year, so there wasn't anything to do. Yay!
- Graphics windowing is required for all graphics operations, so that what is drawn is kept within the rectangle that the user has specified. This was working last year, but in some cases wasn't quite right. There's a bunch more tests for this in the regular graphics system to ensure that the window is honoured properly.

Sprites

Drawing sprites in Pyromaniac (2a)

Two types of modes:

- Shallow modes are paletted and have 256 colours or fewer.
- Deep modes have linear colour components (15bpp and 24bpp modes).

How are they specified:

- The current mode, usually specified as -1 to interfaces.
- Numbered modes.
- Mode selectors, which give the basic screen parameters for a mode.
- Sprite mode numbers, which just contain the colour type and the density.
- Sprites, which can be treated like modes in some cases.

Sprites

Drawing sprites in Pyromaniac (2a)

Next on the list of requirements was having shallow and deep screen modes.

- Shallow modes are paletted. When you need an RGB colour, you must check all the colours in the palette to find the closest.
- Deep modes haves linear colour components (15bpp and 24bpp modes). Looking up a RGB colour is a calculation. Deep modes weren't really implemented last year. They worked for only some very limited cases.

The mode system had been designed with the intention of making it possible to add deep modes later - but last year, only numbered modes were supported. This mattered because the interfaces that sprites and colour translation uses allow for 'modes' to be specified in a number of different forms:

- You can give -1 to mean the current mode.
- You can use a numbered modes below 256.
- You can use mode selectors, which give the basic screen parameters for a mode.
- There are sprite mode words, which just contain the colour type and the density.
- And you can supply a sprites, which can be treated like modes in some cases.

Of these all the formats need to be handled, but only the first two were supported last year. The details of this are documented in PRM 5a, but I've created new documentation to combine all the mode specification details into one place. You'll find this in the API documentation.

17 / 121

Sprites

Drawing sprites in Pyromaniac (2b)

Getting a `Mode` from a mode specifier:

```
def getmodedef(self, mode_or_address):
    """
    Convert from a mode specifier (number, selector, sprite mode word, etc) to Mode.
    """
    modesel = ModeSelector(self.ro, mode_or_address)
    return modesel.modedef
```


Sprites

Drawing sprites in Pyromaniac (2b)

Internally modes are described with a class called `Mode`. This class describes all the properties that the mode has - it's width, height, depth, density and a bunch of other parameters. Basically everything you would normally find in `OS_ReadModeVariable`. Each `Mode` can also return its palette, and the default palette for that mode - because each mode could have had its palette changed.
That was fine when we only had numbered modes, but to introduce the other ways of handling modes we need to be able to create new `Mode` objects to represent (say) sprites.
So we now have a new class `ModeSelector`, which can handle the different forms of mode specification and turn them into a `Mode`. In some cases, the specification is incomplete so we don't know all the information - the sprite mode word, for example, doesn't give any information about width and height, but does contain depth and density.
All the mode handling passes through a `getmodedef` function which takes one of the mode specifiers, and makes a `Mode` using the `ModeSelector` mechanism.

18 / 121

Sprites

Drawing sprites in Pyromaniac (3)

```
if mode in (-1, 0xFFFFFFFF):
    self.modedef = self.ro.kernel.vdu.getmodedef(-1)

elif mode >= 256 and (mode & 1) == 0:
    sprite_address = self.ro.kernel.api.os_spriteop_get_address(area=mode,
                                                                sprite_name=sprite_name)
    self.modedef = self.ro.kernel.vdu.getmodedef(sprite_address)

else:
    # Numbered mode, or a mode descriptor
    self.modedef = self.ro.kernel.vdu.getmodedef(mode)

self.colours = self.modedef.ncolour + 1
if self.colours == 64:
    self.colours = 256
```

Sprites

Drawing sprites in Pyromaniac (3)

Here you can see an example of the `getmodedef` function in use. This is a section of `ColourTrans` with some complexity cut out.
Because I use a single function to get a `Mode`, it is possible to use the `Service_ModeExtension` interface to provide new numbered modes. This interface was already present last year, but only using the RISC OS 3.1 style of mode extensions. It now also supports the RISC OS 3.5 form.
Fortunately the differences between these are largely related to hardware, and as Pyromaniac doesn't have any hardware, we can completely ignore those parameters. It's only the mode variables we really care about, and we just use them to create the `Mode` object.
Because it's easy to create new modes in this way, there's now a configuration option that allows modes to be defined on the command line.

Sprites

Drawing sprites in Pyromaniac (4a)

- To operate on a sprite we need to find it in a sprite area.
- Pyromaniac has an object for a `SpriteArea`, which can locate sprites by name.
- Within the area, we create objects for sprite itself - a `Sprite` object.
- This `Sprite` object knows how to extract information from it:
  - Width and height
  - Mode - resolution, depth and colour type.
  - Palette
  - Image data
  - Mask data

Sprites

Drawing sprites in Pyromaniac (4a)

Inside Pyromaniac, we need to be able to find out where a sprite is before we can operate on it - we need to find the sprite's address. Even if the user gives the sprite's name, internally the OS still has to do this look up. Is that hard? Not per se, but it requires that all the machinery to look up sprites be handled properly, in approximately the same way as it did in RISC OS Classic.
This means that all the sprite calls need to be able to create python objects that reference the sprites in the sprite area so that we can look them up. But only if the sprite operation being performed need a sprite.
For example, some sprite operations don't need a sprite to be passed to them. Others can be passed a value of 0 for the sprite pointer. And of course, some look up the sprite by name and some look up the sprite by address.

Sprites

Drawing sprites in Pyromaniac (4b)

```
@handlers.osspriteop.register(spriteop.SpriteReason_ReadSpriteSize)
def OS_SpriteOp_28(ro, reason, regs, area, sprite):
    regs[3] = sprite.width
    regs[4] = sprite.height
    regs[5] = 1 if sprite.mask_offset else 0
    regs[6] = sprite.mode

    if ro.kernel.sprites.debug_spriteop:
        print("Read sprite size {!r}, name {!r} => {}x{}, mask {}, mode {} (&{:08x})"
                .format(sprite, sprite.name,
                        sprite.width,
                        sprite.height,
                        bool(sprite.mask_offset),
                        sprite.mode,
                        sprite.mode))
```


Sprites

Drawing sprites in Pyromaniac (4b)

Every time you reference a sprite in a sprite area, the Python code will build a `SpriteArea` object that manages the area, and then builds a list of objects which represent the sprites in that area. The sprite object for a given operation is passed to the function that will operate on it - which in our case will be the plot operation. So all of that machinery for decoding sprite areas and sprites had to be built, and the mechanisms for dispatching the handling of each sprite operation. That had to happen just to be able to operate on the sprite area and find details about the sprites.
The code you can see here handles reading the sprite details. Because it has been passed the object representing the sprite it can update the registers with the correct values and return.
But before that we need to get a translation table.

Sprites

Drawing sprites in Pyromaniac (5)

- `ColourTrans_GenerateTable` turns a palette into a translation table for rendering sprites.
- Takes source and destinations, which can be any of the mode specifiers.
- Can change the colours as the table is generated with a transfer function.

A simple call in BASIC for translation from a sprite to the current mode might be:

```
SYS "ColourTrans_GenerateTable", 256, sprite%, -1, -1, 0, %01 TO ,,,,pixtrans_size%
DIM pixtrans% pixtrans_size%
SYS "ColourTrans_GenerateTable", 256, sprite%, -1, -1, pixtrans%, %01
```


Sprites

Drawing sprites in Pyromaniac (5)

Getting a translation table means calling ColourTrans to process the sprite palette and the screen palette to produce the mapping of the pixel values. However the process is more general than that.
The source we're generating a translation for might not be the sprite and its palette, but could be a regular screen mode number, or the current mode. Or it could be a sprite mode word and a palette pointer. Or even a mode descriptor and a marker to use the default palette. Or a mixture of those pairs.
And the destination might be an equivalent set of those combinations.
Additionally, there's a 'transfer function' which can change the colours. This is a pointer to code that `ColourTrans` will call to change the colours as they are being translated. This allows you to do effects like inverting the sprite, or tinting them a particular colour. The WindowManager uses these for selections.
So let's take an example. Let's say we've got a 4 colour sprite and we want to plot it in a 16 colour mode.

Sprites

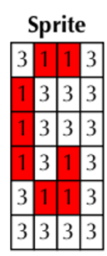
Drawing sprites in Pyromaniac (6a)

Plotting sprites: Source

Source palette

(black)	Colour 0	&00000000
(red)	Colour 1	&0000FF00
(yellow)	Colour 2	&00FFFF00
(white)	Colour 3	&FFFFFF00

Sprite



Sprites

Drawing sprites in Pyromaniac (6a)

Here's what we want to draw. It's a red letter 'G' which is being drawn from a 4 colour mode with the standard palette. It's going to be plotted in a 16 colour mode with the desktop palette.

Sprites

Drawing sprites in Pyromaniac (6b)

Plotting sprites: ColourTrans_GenerateTable

Source palette

	Colour 0	&00000000
	Colour 1	&0000FF00
	Colour 2	&00FFFF00
	Colour 3	&FFFFFF00

Destination palette

	Colour 0	&FFFFFF00
	Colour 1	&DDDDDD00
	Colour 2	&BBBBBB00
	Colour 3	&99999900
	Colour 4	&77777700
	Colour 5	&55555500
	Colour 6	&33333300
	Colour 7	&00000000
	Colour 8	&99440000
	Colour 9	&EEEE0000
	Colour 10	&00CC0000
	Colour 11	&0000DD00
	Colour 12	&BBEEEE00
	Colour 13	&00885500
	Colour 14	&00BBFF00
	Colour 15	&FFBBFF00

Sprites

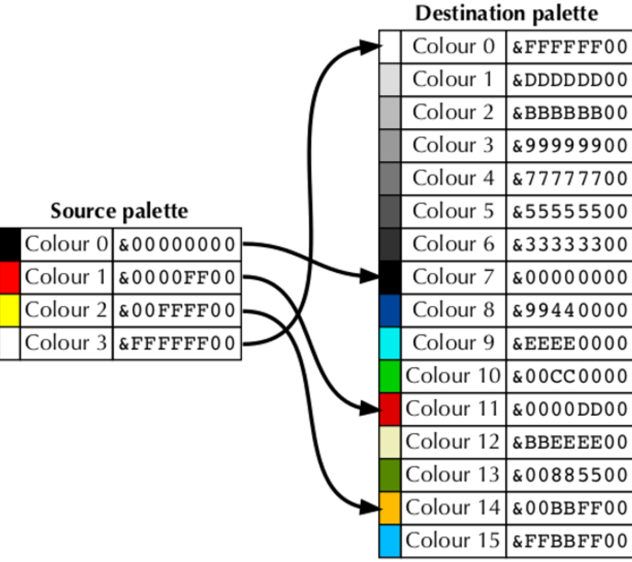
Drawing sprites in Pyromaniac (6b)

So we call ColourTrans_GenerateTable. This call will create a translation table which we can use to translate the colours in the original sprite to the colours on the screen. ColourTrans reads the palette from the sprite and from the current destination.

Sprites

Drawing sprites in Pyromaniac (6c)

Plotting sprites: ColourTrans_GenerateTable



Sprites

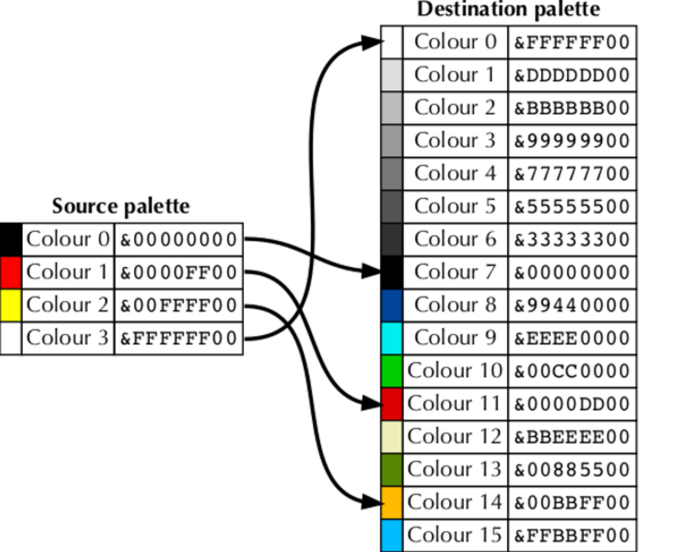
Drawing sprites in Pyromaniac (6c)

It then steps through each of the colours in the source palette and finds the closest match in the destination palette. This is actually done with some weightings to allow for more sensitivity to certain colours.

Sprites

Drawing sprites in Pyromaniac (6d)

Plotting sprites: ColourTrans_GenerateTable



Table

Byte 0	7
Byte 1	11
Byte 2	14
Byte 3	0

Sprites

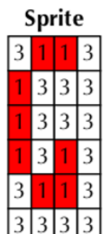
Drawing sprites in Pyromaniac (6d)

These colours are then written into the caller's translation table. In this example we're using a 4 colour source and a 16 colour destination, but in a 256 colour mode, this can result in a lot of comparisons. Classic ColourTrans has optimisations for this, but these aren't present in Pyromaniac at present.
Here you can see that the translation table has the original colours mapped to colour 7, 11, 14, and 0.

Sprites

Drawing sprites in Pyromaniac (6e)

Plotting sprites: OS_SpriteOp



Table

Byte	Value
Byte 0	7
Byte 1	11
Byte 2	14
Byte 3	0

Sprites

Drawing sprites in Pyromaniac (6e)

Having obtained the translation table, we now call `OS_SpriteOp` to actually render the sprite with that table. `OS_SpriteOp` is given the sprite, and the translation table, and it has to put that sprite on the screen. The procedure is very similar to that on RISC OS Classic, but there is only one implementation inside Pyromaniac - whereas in RISC OS Classic there are two. One implementation, for the simpler operations is in the Kernel, and one for the more complex operations is in SpriteExtend.

Sprites

Drawing sprites in Pyromaniac (6f)

Plotting sprites: OS_SpriteOp

Sprite

3	1	1	3
1	3	3	3
1	3	3	3
1	3	1	3
3	1	1	3
3	3	3	3

Byte data

11 01 01 11	= 215
11 11 11 01	= 253
11 11 11 01	= 253
11 01 11 01	= 221
11 01 01 11	= 215
11 11 11 11	= 255

Table

Byte 0	7
Byte 1	11
Byte 2	14
Byte 3	0

Sprites

Drawing sprites in Pyromaniac (6f)

To explain how the sprite is stored, I've included a table showing the bit pattern that's stored in the sprite and the decimal values of those bytes. In reality, all the sprites are aligned to 4-byte words, but I'm trying to keep this simple.

You should be able to see that the 'G' is flipped horizontally. This is because the left most pixel's value is stored in the lowest bits of the byte, and the right most pixel is in the highest bits of the byte.

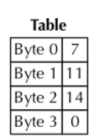
Sprites

Drawing sprites in Pyromaniac (6g)

Plotting sprites: OS_SpriteOp

Byte data

11 01 01 11	= 215
11 11 11 01	= 253
11 11 11 01	= 253
11 01 11 01	= 221
11 01 01 11	= 215
11 11 11 11	= 255



Table

Byte 0	7
Byte 1	11
Byte 2	14
Byte 3	0

Colour lookup

215	= (3, 1, 1, 3)
221	= (1, 3, 1, 3)
253	= (1, 3, 3, 3)
255	= (3, 3, 3, 3)

Sprites

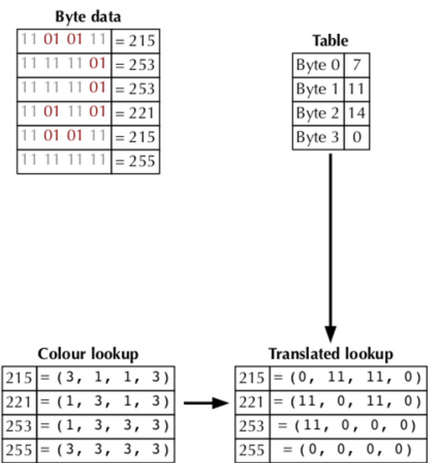
Drawing sprites in Pyromaniac (6g)

Now let's do some actual processing on this data. The first thing that the Pyromaniac code does is to build a table that converts from the byte values to the pixel values - essentially undoing what I did to explain how it was stored in the previous slide. This is actually a static table for each depth of mode.
Because there are only 4 distinct values used in our example sprite, I've only shown these 4 entries in the lookup table.
As you can see, the byte 215 in the sprite actually means colours 3, 1, 1 and 3 in a 4 colour mode. It doesn't matter what those colours are. That's why we use the translation table - ColourTrans has already done that work to map those colours to the new values.

Sprites

Drawing sprites in Pyromaniac (6h)

Plotting sprites: OS_SpriteOp



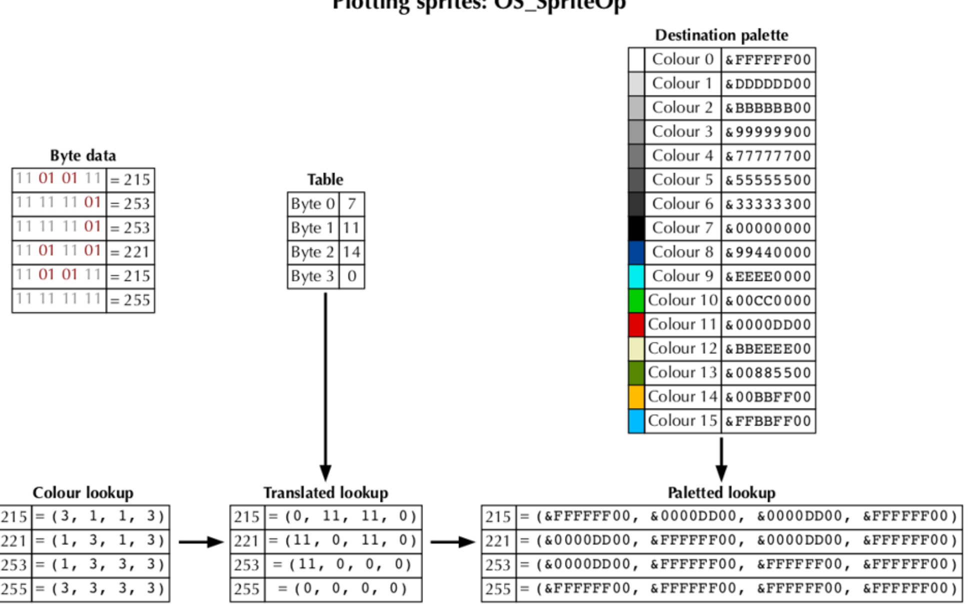
Sprites

Drawing sprites in Pyromaniac (6h)

Next we apply the translation table to the lookup table. This means replacing each of the colour numbers from our lookup with the corresponding number from the translation table.
This new table tells us what colours we should use in the destination mode for a given byte in the source sprite. That would be fine if we were writing to a RISC OS mode, but we're not - Cairo only ever uses a 24 bit per pixel output so we need to convert to its format.

Sprites

Drawing sprites in Pyromaniac (6i)



Sprites

Drawing sprites in Pyromaniac (6i)

We look at the current destination's palette and we apply a mapping to the translated palette, giving the actual colours that will be used on the screen. So now we have a table that maps from the byte that was stored in the sprite to the colour that will be placed on the screen.

At this point you might wonder why we can't skip the translation table entirely and just map the colours directly to those that will be put on the screen from the source palette. The reason is pretty simple - the translation table doesn't have to map colours directly.

They could be made darker, lighter, or inverted. Or the user could be doing some colour cycling - hiding some colours to make a simple animation. This means that you have to honour what the user supplied to you.

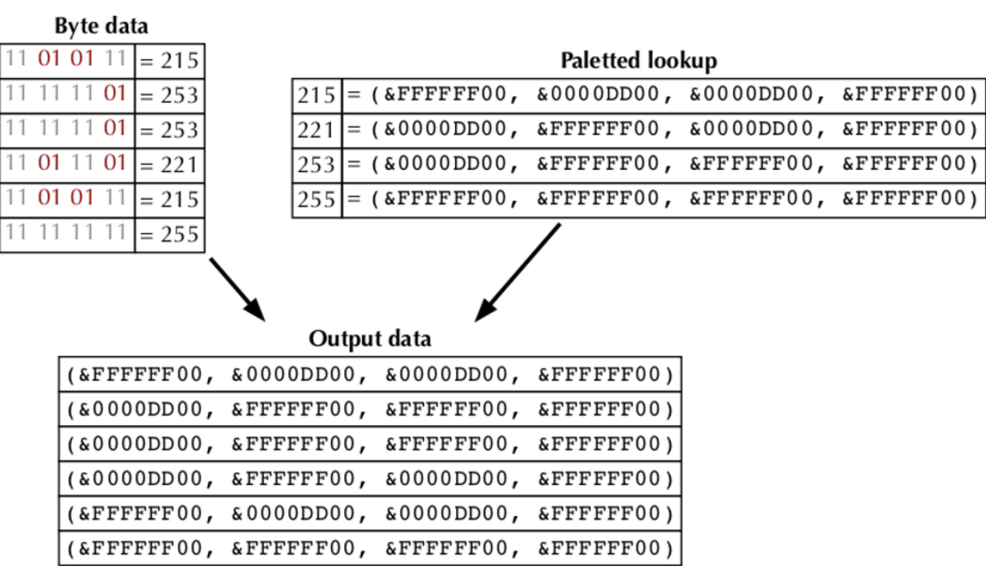
There are special options to some of the sprite plotting calls to plot directly from the palette, and this does offer some opimisation opportunities.

Up to now, we're not actually been concerned with the sprite data. All we've been manipulating is the table of lookups that will be performed on each byte of data.

31 / 121

Sprites

Drawing sprites in Pyromaniac (6j)



Sprites

Drawing sprites in Pyromaniac (6j)

Finally we can put this information together. For every byte in the source sprite data we look up what this will be on the screen in the paletted lookup table. This produces a table of colour values which can then be written to a Cairo surface and rendered on the screen.

Hopefully you can see in the colour values that the image data is of a 'G'.

There are a lot of things I've glossed over here:

- The sprites may not start at bit 0, or end at the end of a word - they might have a few bits at the start of the byte that aren't used. Or if this sprite had only 3 columns instead of 4, the final 2 bits wouldn't be used.
- In both these cases, the Pyromaniac code still converts the data, but cuts the missing parts off before giving the to Cairo. It's slightly less efficient to process that extra data, but a lot clearer in the code to do it that way.
- The colours that are supplied to Cairo aren't actually in the same format as for RISC OS - generally most graphics systems settled on the opposite order for the RGB values. So those are reversed at the palette lookup.
- Masked sprites come in a number of flavours. There is a similar lookup that is performed to create an alpha channel in the Cairo image before it is plotted to the screen.

All of these translation table and processing operations happen for every single sprite plot. In Pyromaniac it's all implemented in Python. You might think that that means that it's going to be slow. And yu'd be absolutely right. But it's not quite as bad as you might expect.

Sprites

Coordinate space (1)

- Coordinate spaces describe where you start drawing from and which direction is positive in each axis.
- RISC OS uses cartesian coordinates, just like the BBC.
- These are mapped to pixels on the screen.
- Pyromaniac has to then map them to the coordinates used by the Cairo graphics system.

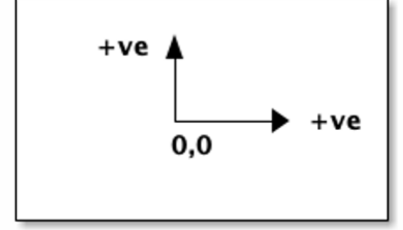
Sprites

Coordinate space (1)

The graphics system in Pyromaniac has a number of coordinate spaces to deal with. Each of the graphics operations needs to convert between these coordinates spaces in order to put something on the screen. The first three spaces are common with RISC OS Classic.
These coordinate conversions aren't really restricted to sprites, as they apply to all the graphics primitives, and Draw and Font rendering.

Sprites

Coordinate space (2)



- Origin is specified by the user.
- X-coordinates increase to the right of the screen.
- Y-coordinates increase up the screen.
- User coordinate space has coordinates which are scaled by the eigenfactors, representing the shape and size of pixels.
- Bottom left coordinates are (-800, -600).
- Top right coordinates are (800, 600).

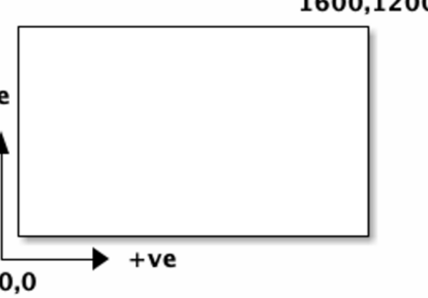
Sprites

Coordinate space (2)

The first space is the user space with an origin. In this example, the origin has been placed at the centre of the screen.
As I step through these spaces, the changing parameters are highlighted in bold.

Sprites

Coordinate space (3)



- Origin is specified **at the bottom left**.
- X-coordinates increase to the right of the screen.
- Y-coordinates increase up the screen.
- User coordinate space has coordinates which are scaled by the eigenfactors, representing the shape and size of pixels.
- Bottom left coordinates are **(0, 0)**.
- Top right coordinates are **(1600, 1200)**.

Sprites

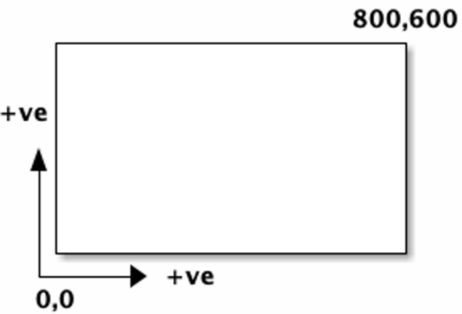
Coordinate space (3)

The space is converted to normalised user space by adding the origin to all the coordinates.
The origin is now in the bottom left, and the coordinates are affected accordingly.

35 / 121

Sprites

Coordinate space (4)



- Origin is specified at the bottom left.
- X-coordinates increase to the right of the screen.
- Y-coordinates increase up the screen.
- **Coordinates map directly to pixels.**
- Bottom left coordinates are (0, 0).
- Top right coordinates are **(800, 600)**.

Sprites

Coordinate space (4)

Then we convert to internal coordinates by scaling by the eigenfactors. The eigenfactors basically describe the shape of a pixel - how rectangular it is.
This means that every addressable coordinate now maps to a pixel, and the top-right is now the size of the screen in pixels.

Sprites

Coordinate space (5)

- Origin is specified at the **top** left.
- X-coordinates increase to the right of the screen.
- Y-coordinates increase **down** the screen.
- Coordinates map directly to pixels.
- Bottom left coordinates are **(0, 600)**.
- Top right coordinates are **(800, 0)**.

Sprites

Coordinate space (5)

Finally the Cairo implementation has to turn everything upside down.
Like most other graphics systems, Cairo uses an origin at the top left, with Y-coordinates increasing down the screen.
This means subtracting the coordinates from the height of the screen.
Every graphics operation has to go through these transformations to get to the position that they will appear on the Cairo surface.

Sprites

Transformations (1)

To resize the sprites with `OS_SpriteOp`...

- Some calls always render 1:1 on the screen.
- Some calls take a transformation matrix.
- Some calls take a scale block.

Pyromaniac has `Scale` and `Matrix` objects to handle these operations.

Sprites

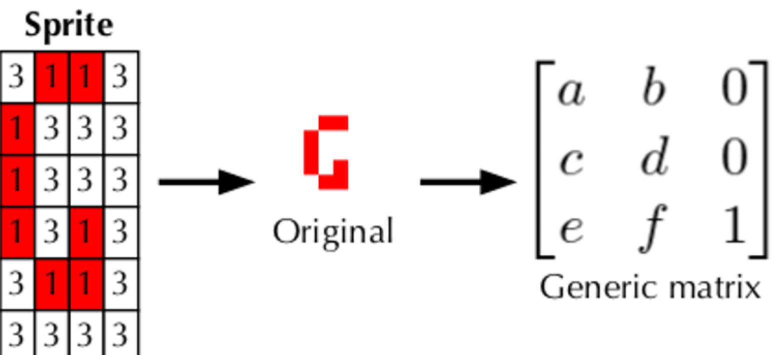
Transformations (1)

The user can say that they want to plot sprites with a transformation matrix. They might do this to scale things up or down, or skew the image. Working with matrices can be pretty unfun, especially when you have to convert between different coordinate spaces.
Alternatively, the user can specify a scaling ratio. In this case they only give the ratio as a multiplier and a divider for both the X and Y dimensions. Inside Pyromaniac the `Scale` and `Matrix` objects descend from a common `Transform` object, which means that applying them to coordinates can be done through common functions.

Sprites

Transformations (2)

Transformation



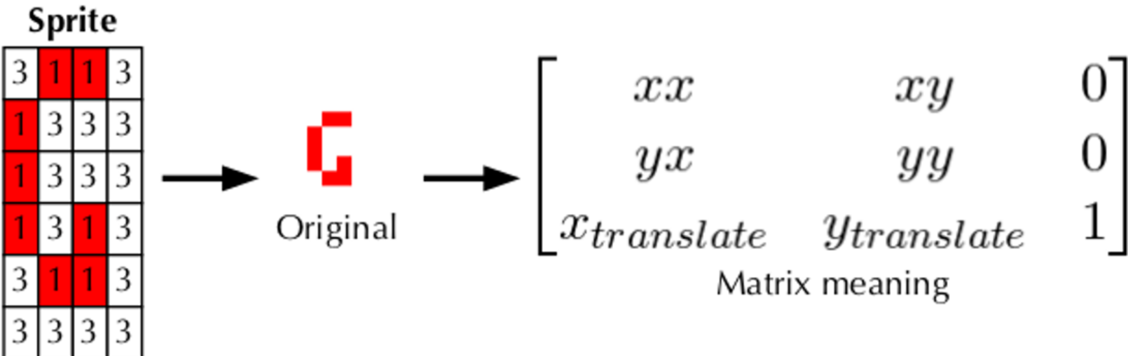
Sprites

Transformations (2)

This is the generic form of a transformation, using a matrix to calculate the new coordinates. Only the 6 values `a` to `f` are supplied to RISC OS.

Sprites

Transformations (3)



Sprites

Transformations (3)

This is what those variables mean.

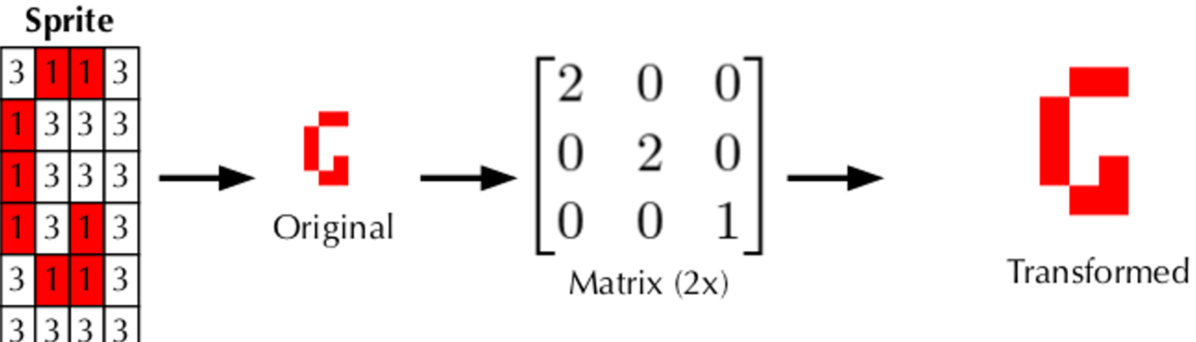
- The top 2 are used to assign values to values to x, from the original x and y coordinates.
- The middle 2 are used to assign values to y, again from the original x and y coordinates.
- The lower 2 are used to translate the x and y coordinates.

If you need to do matrix calculationsthere's a lot of information online to help. Now that I've got the `Matrix` object working I use it to do all the calculations for transformations because the maths is easy to get wrong.

40 / 121

Sprites

Transformations (4)



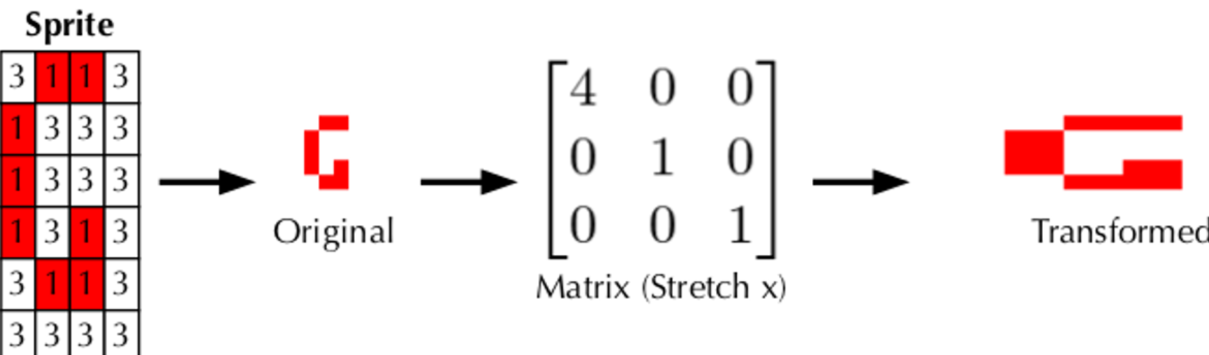
Sprites

Transformations (4)

This is an example of scaling a sprite up to twice its original size. Two of the values in the matrix are set to 2, to scale both dimensions.

Sprites

Transformations (5)



Sprites

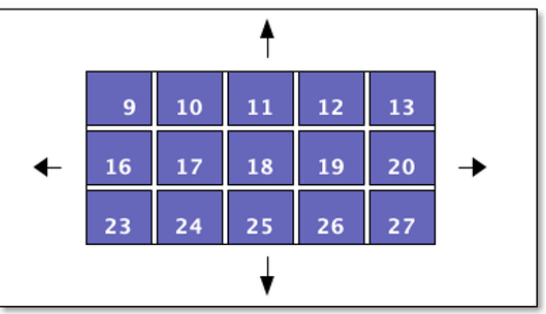
Transformations (5)

And if you wanted to stretch the graphic you'd give these different values.
There's actually a lot more you can do with this, from skewing the graphic at an angle, to rotating the graphic around an origin.
The matrix operations in Pyromaniac are processed methodically, and then when we get to Cairo they have to be modified to account for the coordinate space being upside down in relation to RISC OS. It's kinda frustrating, but now that it's working it's pretty nice.

Sprites

Tiling (1)

- Tiling is used for the desktop background tile.
- Traditionally this was done by repeatedly calling `OS_SpriteOp`, like this:



Sprites

Tiling (1)

Tiling of sprites isn't used that often in RISC OS, but the one place it is used is quite prominent - the mottled sprite on window backgrounds. Traditionally this was drawn by finding the region that the sprite needed to fill, and repeatedly rendering the same sprite at offsets so that it covers the entire graphics window.

You can see that *many* sprite operations might be needed to fill a region here. This isn't such a big deal when you're plotting to the screen with direct screen access, but if you've got an accelerated graphics system this can matter.

In Select I introduced a new call to request that a sprite be plotted tiled into the graphics window.

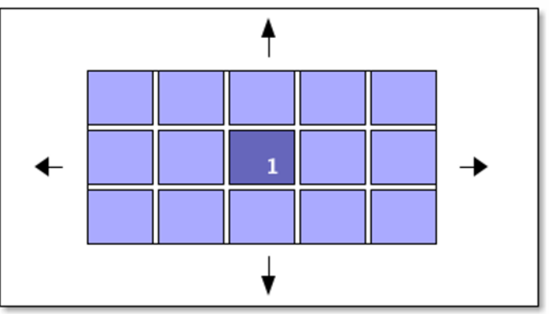
This made it easier for applications wishing to tile sprites, but more importantly it made it possible for accelerated interface to perform this tiling operation efficiently. The software implementation still uses the same algorithm when there is no acceleration, but when there is acceleration available, this can run much faster.

This acceleration was provided in the ViewFinder driver to make its tiling faster than it would have been when drawn normally.

Sprites

Tiling (2)

Tiling a sprite with the interface is simple - plot the sprite at a single location and it fills the graphics window:



Sprites

Tiling (2)

The interface is very simple - plot the sprite at a single location, and it will be automatically repeated into the surrounding space. Only 1 `SpriteOp` call is made here, and the whole area is filled.

Sprites

Tiling (3)

```
if surface:
    spattern = self.cairo.SurfacePattern(surface)
    spattern.set_filter(self.cairo.FILTER_NEAREST)
    spattern.set_matrix(invcmatrix)
    if tile:
        spattern.set_extend(self.cairo.Extend.REPEAT)
    context.set_source(spattern)
    graphics._set_action(plot_action)
else:
    graphics._set_colour(plot_colour, plot_action)

if tile:
    context.rectangle(graphics.windowx0, graphics.scrheight - graphics.windowy1 - 1,
                      graphics.windowx1 - graphics.windowx0 + 1,
                      graphics.windowy1 - graphics.windowy0 + 1)

context.set_matrix(cmatrix)
if not tile:
    context.rectangle(0, 0, sprite.width, sprite.height)

if translucency:
    alpha = (255 - translucency) / 255.0
    context.clip()
    context.paint_with_alpha(alpha)
else:
    context.fill()
```


Sprites

Tiling (3)

How does that related to Pyromaniac? In modern graphics system - and I'm using modern to mean 'the last 30 years or so' - it has been common for this operation to be provided by the hardware system. Instead of bounding a given sprite to its own extent, it is repeated within the region it is drawn. This is commonly just a single configuration change when plotting a sprite for whether it should repeat or not.

Cairo provides such an option, and so being able to call the Cairo rendering once and have it fill the whole region was very simple.

This example is the tail end of the sprite plotting code, where the tiling was inserted. This happens after the RISC OS sprite data has been converted to a Cairo surface, and all the matrix calculations have been performed.

The changes for the tiling are very simple:

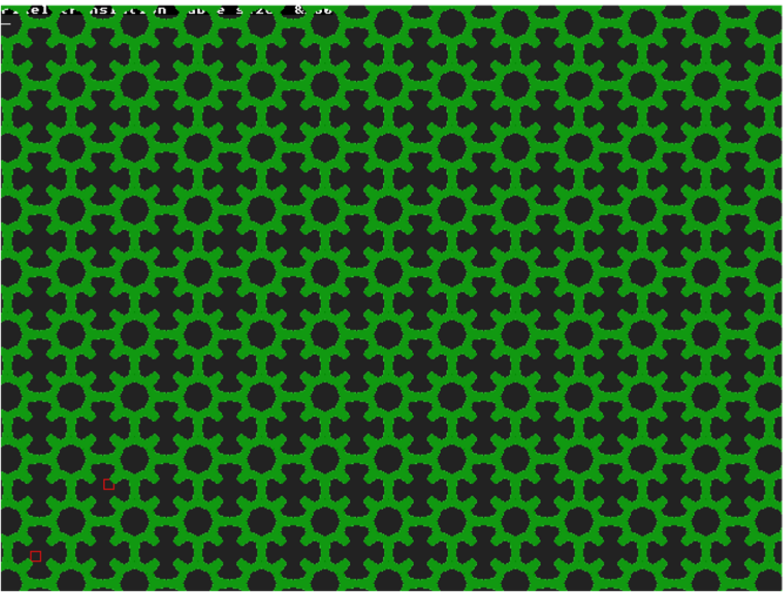
- We change the sprite rendering to `REPEAT` across the page.
- We give it the whole graphics window as the region it will be repeated over.
- And if we're not tiling, we only draw the little rectangle to cover the sprite.

As you can see the extra code needed to tile the sprite is pretty simple and easy to understand.

Sprites

Tiling (4)

Tiling a cog sprite:



Sprites

Tiling (4)

This is what it looks like when you tile the cog sprite. You can see some red marks towards the bottom left of the image, which show where the actual plot of the sprite was performed.

The Pyromaniac implementation was kinda fun when I first ran it because it just worked. Adding the single option to repeat the image was all it took and tiled sprites were then accelerated.

VNC game demo

To play your own game, instructions are at: https://railpro.riscos.online/
Connect to VNC at: vnc.railpro.riscos.online display 8 (port 5908)
Password: `password`

VNC game demo

It's time for our first demonstration. This demonstration builds upon what I've discussed up to now. If you have a VNC client, you can connect to the server at address vnc.railpro.riscos.online on display 8. You'll be playing a different game to the one I'm displaying - each connection is a separate game.
I'm going to connect to the server here, and I'll show you the VNC session running.
<launch the VNC server and then share its screen in Zoom>
This is the game RailPro, by Jos Keijzers. The idea is to guide the trains to platforms and out again. That's it. No ray guns and explosions. Well, there might be explosions if you crash the trains. We try not to do that.
It takes a little while to start because it reads the map using BGET operations, which aren't very fast.
It's a mouse based game, which uses the cursor keys to move around the map. As you can see here we've got a load of tracks in grey, and a number of signals in red and green. Those are all masked sprites being rendered. The background is drawn in different colours - there was some fun getting the correct set of colours to appear.
I can scroll left and right to see the two parts of the map. <cursor right>
Let's start the game. <press S>
There's a train coming in which is meant to go to **which platform it's being sent to**.
As you can see, the train is moving slowly towards our region and we need to get it there. It's necessary to change the signals and the points to make the train reach its destination. Although the train is moving slowly, another one will be along soon.
Although Pyromaniac is not fast itself, this sort of game is practical because it does not require rapid redraw. The gameplay like this is almost the same as the original game when I played it on my A5000.
<adlib as necessary - there's not much to say, but a minute or two of gameplay won't hurt>
The game continues until you cause catastrophic crashes and they fire you, or... well, I'm not sure. I always get fired, although I have managed to last for over 20 minutes in one game.
The actual server is providing a separate invocation of the RISC OS Pyromaniac system inside its own docker

3. Fonts

3. Fonts

Ok, let's talk about the revamp that the font system has had over the past year...

Fonts

Where were things last year?

- FontManager kinda worked.
- But it was on a branch - I wasn't confident with it yet.
- Didn't handle control codes properly, or consistently.
- Required a lot of work for me to be happy with it.

Fonts

Where were things last year?

I'd mentioned in the prior presentation about how the font system was painful.
The FontManager allows strings to be passed to it which are to be plotted on to the screen. Usually those strings are just plain text, and that's commonly what you'd expect. But some parts of the system - the WindowManager particularly - use control codes that are embedded in the font string to change its behaviour. For example, you can change the font mid-string, move the baseline, change the colours or even change the transformation matrix.
Little bits of this were implemented last year, but only inside the `Font_Paint` call - and wasn't implemented at all in any of the other font SWIs.
These control codes needed to be handled properly, mostly for the WindowManager. Without them, there would be no 'shift' characters on menus and the positioning of the right aligned text would not work.

Fonts

Simple text (1)

- Simple rendering was simple - stop on a 0 byte.
- Fine for the presentation, because it doesn't do anything fancy.

To render fonts, RISC OS Pyromaniac has two major parts:

- The graphics system's font interface.
  - For example selecting fonts, sizing simple text, drawing text with transformations.
- The RISC OS-facing SWI interface.
  - For example `Font_FindFont`, `Font_ScanString`, or `Font_Paint`.

Largely, Pyromaniac just turned the SWI calls into graphics system calls in this simple text system... but the WindowManager needs more.

Fonts

Simple text (1)

The first attempt to get the FontManager working - which was demonstrated last year as part of the presentation - was to just take the string and stop as soon as a 0 byte was encountered. For the presentation system this was fine because there wouldn't be any anything control codes in there.

To render fonts, RISC OS Pyromaniac has two major parts:

- The graphics system's font interface.
- The RISC OS-facing SWIs.

The Pyromaniac graphics system has primitives to select font faces, size text, and to draw text in a colour with a transformation. When the FontManager wants to perform operations on the fonts, it calls these primitives. This means that what the operations do can change - and it does depending on the graphics implementation. Under Cairo, this is converted to the 'Toy' text interface. If a different graphics back end was used then only the font primitives need to change - FontManager is exactly the same.

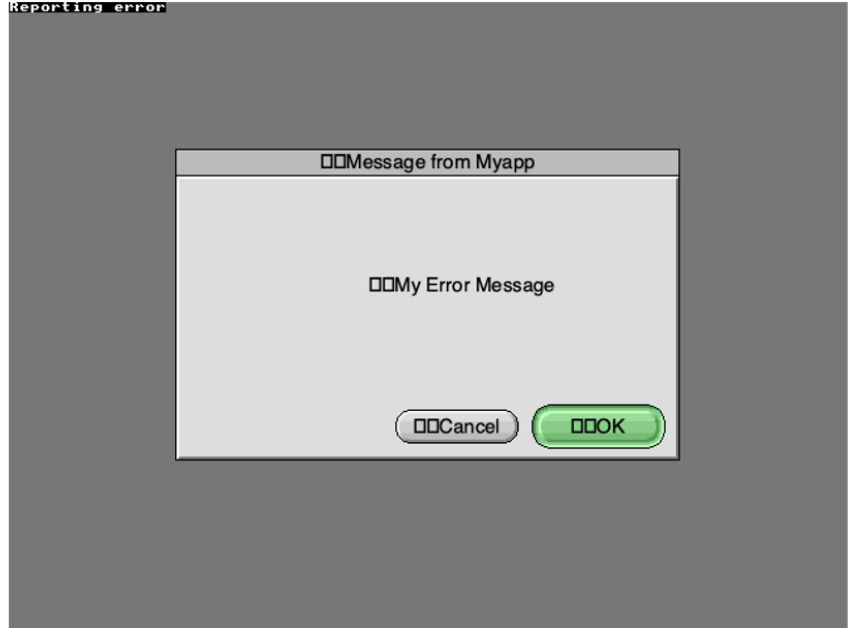
In RISC OS terms this would be equivalent to keeping the SWI interface but switching out the Font Outlines parsing and rendering for a different system like FreeType. RISC OS Pyromaniac is intended for debugging and trying things out, so my intention was to give the system a structure that might then be applied to a RISC OS module in the future.

Anyhow, the graphics system's font primitives are pretty much sufficient to do everything that FontManager requires. But FontManager has to use them properly if the SWIs are to work as expected, and in this initial implementation the FontManager just took a string and threw it at the graphics system to be rendered.

That's fine for the presentation system, but when it comes to the WindowManager, there's a lot more going on. At that time, the WindowManager didn't work very well - but well enough that the Wimp_ReportError call could function.

Fonts

Simple text (2)



Fonts

Simple text (2)

That's what happens if you don't process any of the control codes before you try rendering it. The leading control characters appear as squares in the output.

You can also see that at that time the graphics system was working well enough to draw the rest of the window - the outline and 3D border are working, and the IconBorderRound module is able to draw its buttons just fine. Aside from colour selection, those are all operations that could have run on the BBC so they're really not that special.

Fonts

Control codes

Font control codes the SWIs need to support:

- `0, 10, 13` - terminates the string
- `9, 11` - moves the cursor horizontally and vertically by a specified amount
- `17` - changes the foreground colour
- `18` - changes the foreground and background like `Font_SetFontColours`, using GCOLs
- `19` - changes the foreground and background like `ColourTrans_SetFontColours`, using RGB values.
- `21` - hides text until the next control character
- `25` - sets the underline parameters
- `26` - selects the font to use
- `27, 28` - changes the transformation matrix (with and without translation)

Fonts

Control codes

This wasn't working out right, so I needed to handle the control strings better. The control strings are styled to be like the VDU codes with the same numbers, which is kinda cute and makes understanding them a lot easier. What you can't tell from this list is that few of these calls are used, even in the WindowManager.

The codes are clearly important to getting the interface correct, but just to get things working, it wasn't necessary to process them all. Those squares preceding the text were actually a 26, <font handle> sequence that the WindowManager inserts before every icon render.

I replaced the bare string read with a parser that would read the bytes from the string and just throw away all the control codes entirely - except the first one. The first one was always processed because that's how the WindowManager selected a font. Not ideal, but it gets past the problem of those squares turning up.

52 / 121

Fonts

Spacing parameters

Spacing in menus would usually be put in the menu text like this:

```
Save     ^F3
```

And then the WindowManager handles the alignment for you:

- It gets the size of the string without any extra spaces.
- Subtracts from the menu width.
- And then uses the remaining space as the inter-word spacing parameter.

Pyromaniac only supported this in a simple way by splitting on spaces. And it didn't support the inter-character spacing.

Fonts

Spacing parameters

There are other wrinkles, too. Menu items in the desktop are usually described with simple strings. In your application you include a string that has the shortcut on the end, like the example.
And you expect the `^F3` shortcut to be right aligned. This worked when the desktop used the monospaced system font, but with proportional fonts those spaces won't give the right alignment. What the WindowManager does is to reduce the trailing spaces to just one space and any other spaces to hardspaces.
Why ? Because then it can use a trick to move the text. It could insert a `9` sequence to move the text, but it doesn't - I'm not sure why, but instead it changes the word spacing when it plots the text.
Changing the word spacing is commonly used to provide a cheap form of justification, and that's essentially what's being done here.
The WindowManager calls the `Font_ScanString` to get the size of its modified string. It then subtracts that from the space it's has available in the menu, and uses that as the word spacing it supplied to `Font_Paint`.
Because it has ensured that only a single space is present and that this is just before the shortcut, the `^F3` becomes right aligned.
In the Pyromaniac FontManager at that time, if a word spacing was supplied, the string would be split on spaces and it would just repeat the rendering on each fragment, adding the word spacing to the baseline.
This was good enough, and menus would show their shortcuts right aligned.
The FontManager still didn't handle the inter-character spacing, or any of the control codes, but it was good enough.

Fonts

Font encodings (1)

- `WIMPSymbol` provides the shift and other arrows which were in the VDU 4 only Wimp.
- The WindowManager switches the font string to the `WIMPSymbol` font when it sees these characters.
- When you use the character 139 (&8b) - the scroll up arrow - it gets converted to:
  - `26`, *WIMPSymbol font handle*, `139`, `26`, *desktop font handle*
- The FontManager uses font-specific encodings to handle this.
- So Pyromaniac has a version of this internally.
- The `python-codecs-riscos` module was updated to add the encodings for the symbol fonts `Sidney`, `Selwyn` and `WIMPSymbol`.

Fonts

Font encodings (1)

There is a font used by the WindowManager which is used to render special characters. The `WIMPSymbol` font provides a few characters which were previously used in the system font for window furniture - the close icon, back icon and the arrows. The Window Manager replaces these if you use them in strings with a reference to the font.
So when you use character 139 for scroll up, it gets converted to the sequence you see.
With my code that stripped out the control characters, you'd be left with a regular character 139 in the desktop font, so that didn't look great. But even if I did process the control characters, code 139 isn't an up arrow in any font encoding - it's only present in the system font because the WindowManager reprograms the characters when it starts.
Fortunately, this is what font encodings are for. Normally when you select a font you tell it what encoding you're going to use, or use the default usually Latin 1. These regular encodings were already implemented. However, fonts can have their own encodings that they use in preference to the encoding you supply. This is used for symbol fonts. The fonts Selwyn and Sidney use this to ensure that they always look the same regardless of your current alphabet.
So I had to add the ability of fonts to override their supplied encoding and always use their own - and to define what that encoding is for each of those fonts. Fortunately, defining the encoding was pretty easy because I can just add them to the encodings provided by the `python-codecs-riscos` module.
If you look at the GitHub repository, you'll see that there specific encodings for the Selwyn, Sidney, and WIMPSymbol fonts. Those are used as font encodings so that the fonts are rendered as expected.

Fonts

Font encodings (2)

Fonts

Font encodings (2)

This is Selwyn, being rendered with its own font encoding. Internally this turns the byte codes into unicode characters with a suitable font under macOS. Most of the characters are exactly as you'd expect.

Fonts

Improved control codes parsing (1)

How the WindowManager handles `Replace shift-F5` as a menu item:

- `[26, 1]` - *change font to desktop font*
- `"Replace "` - *regular string*
- `[26, 2]` - *change font to WIMPSymbol*
- `"\x8b"` - *regular string for the shift character*
- `[26, 1]` - *change font to desktop font*
- `"F5"` - *regular string*

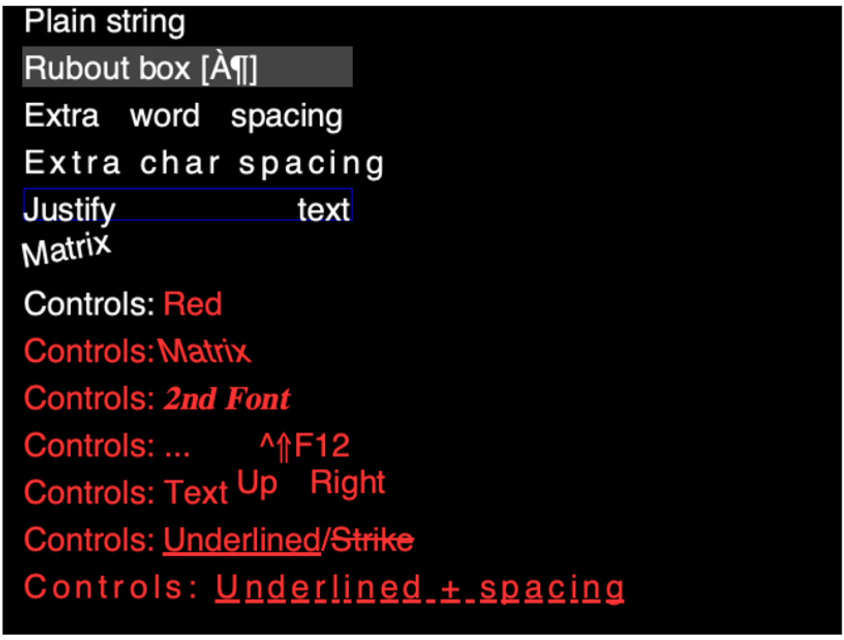
Fonts

Improved control codes parsing (1)

With the encoding problem solved, we still need to be able to change fonts in the middle of the line, or do any of the other control codes.
So I wrote a parser which would step through the bytes that you gave it and return a list of either control codes to process or the string to render. If you gave it a string with 'Replace shift-F5', it would give you a list like this.
As you can see it's broken down into segments that use the different fonts.
The `Font_Paint` code would then step through the list and process each operation. When it got a string, it would call the rendering code, and when it got a font change it would update the font handle. And the other control codes were also processed to move the baseline, change colours or whatever.
And that worked pretty well.
With that in place it was also possible to handle inter-character spacing as well by processing each character in the string separately.

Fonts

Improved control codes parsing (2)



Fonts

Improved control codes parsing (2)

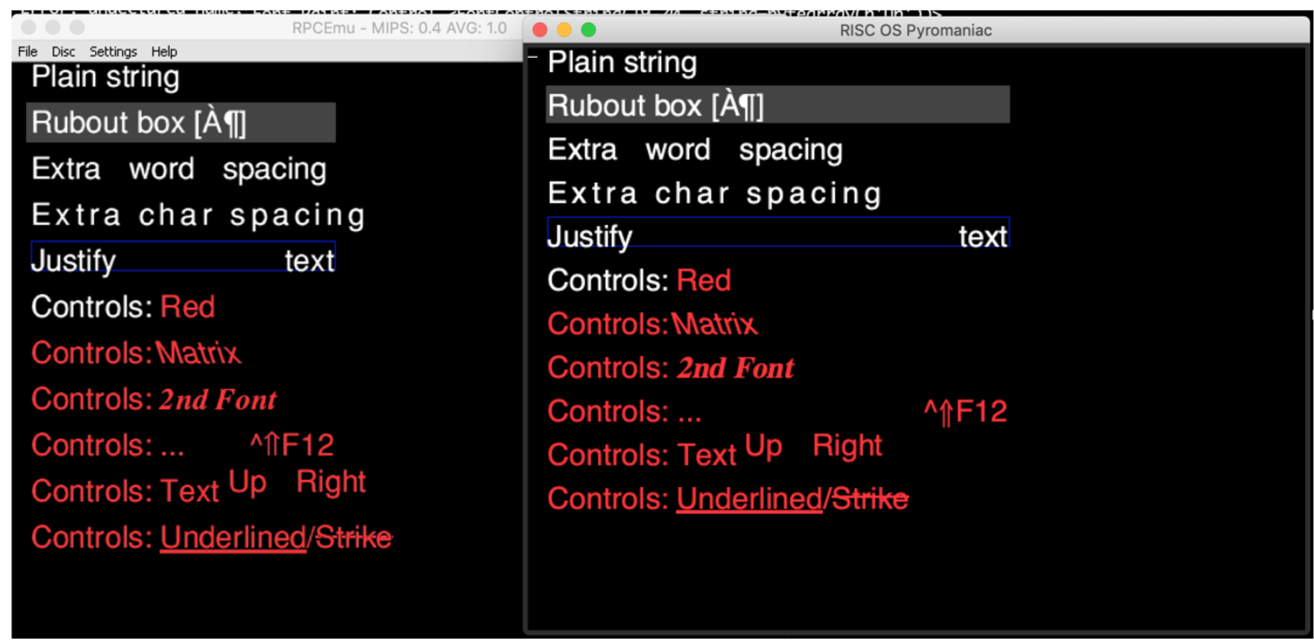
This is what it looks like when you run my manual tests. Each of the lines tests one or more of the features of `Font_Paint`.

- The first is just a string with no parameters.
- The second shows the rubout box in being used behind the string - there's a UTF-8 character rendered in Latin 1 for the paragraph marker showing that we really are using the configured alphabet.
- Then we show off the word spacing as working.
- And the character spacing as well.
- The justify option spreads the space left over in a region between the space characters.
- Passing in a matrix when the font is painted is shown next by angling the text upwards.
- Then the control code tests start - changing colour to red.
- Applying a matrix in the middle of the line makes the text lean backwards.
- Changing to a different font comes next, with Trinity Italic used here.
- Now there's something similar to what the Wimp does - using justification to right align the text after the dots. And the shift character is using the switch to WIMPSymbol to render that one character.
- Here we exercise the vertical movement by lifting the baseline up, and then moving it horizontally right.
- The underline is limited, but it can be used to make a strike-through as well.
- Finally we apply the character spacing and the underline operations. Here you see a problem - splitting the characters up and moving between them means that the underline doesn't work as well as you'd like.

The program that creates this can be found in the `riscos-examples` repository.

Fonts

Improved control codes parsing (3)



Fonts

Improved control codes parsing (3)

And here you can see the output of a slightly earlier version alongside RPCEmu rendering the same content. Pyromaniac is on the right.
There's some differences in the font metrics which mean that things are spaced slightly differently, and the font is a little bolder in Pyromaniac. But basically, it's looking like they do the same thing.

Fonts

Proper control code parsing (1)

- This is still a bit of a bodge.
- We only process things properly in `Font_Paint`.
- `Font_ScanString` and friends still use the control code stripping method.
- So I started again...
- ... But learnt from what I'd done before.
- The new parser was written using the principles of what I'd learnt about processing control code segments.
- It wasn't written inside Pyromaniac at all.

Fonts

Proper control code parsing (1)

It's still not right though. And the reason it's not right is that it's still a bit of a bodge. Not only is it not processing the codes properly, the processing is performed by the `Font_Paint` code. Because `Font_ScanString` is still using the old method of stripping all the control codes out, it's not actually compatible. Adding the same code to `Font_ScanString` would be difficult, so... it's time to do it right.

The code that was parsing the strings was separated but a bit ugly, and the code that acted on it was built into the `Font_Paint` code. The code needed to be reworked to be able to be reused in the `Font_ScanString` implementation and able to be tested well.

So, I started again, from scratch - well, using the principles of what I'd done in the previous version. The new implementation has a lot more structure, and is designed to be used without needing Pyromaniac. I did this so that I could develop and test it without having to worry about the rest of the system.

The non-Pyromaniac version takes a set of bytes containing the font string to be processed. In Pyromaniac the functions that access the bytes of the string instead read from the emulated memory. This ensures that we only access bytes as we need to.

59 / 121

Fonts

Proper control code parsing (2)

- `Matrix` and `Bounds` - define structures to manage transformation matrices and bounding boxes.
- `FontContext` - holds all the state for sizing or rendering, and controls processing.
- `FontControlParser` - performs reading the string and creating a list of operations.
- `FontControlSequence` - manages the list of operations described by the control sequence.
- `FontControl*` - classes which perform the operations for each control, eg `FontControlString`, `FontControlRGB`.

Fonts

Proper control code parsing (2)

So what makes up the parser?

Despite being separated from the Pyromaniac, I'm still using the graphics structure definitions for `Matrix` and `Bounds`. These aren't really related to the memory access so they're reusable.

There's a `FontContext` object which has the state stored in it, and is how we can control the sizing and rendering operations. It has functions which allow it to convert between GCOL and RGB values and to perform operations on font handles. Outside of Pyromaniac these are just simple operations that return dummy values. Inside Pyromaniac it is subclassed so that those functions are overridden by the ones that talk to the real graphics system.

The `FontContext` also performs the processing on a list of operations to either render or size the provided strings.

The `FontControlParser` takes the font string and parses it into a list of operations. It can give more information on what it has processed, but is largely used to create a `FontControlSequence` which holds all the information about the string that was processed. In Pyromaniac, the `FontControlParser` is subclassed to replace the memory reading, allowing it to access emulated memory.

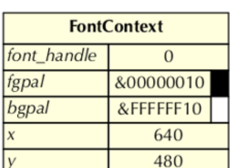
The `FontControlSequence` is a list of objects which describe each of the operations in the string. The `FontControlSequence` can also create a new list of operations which has injected movements for the word spacing or character spacing. This allows us to process the basics of the string and then split up it up with the spacing as necessary.

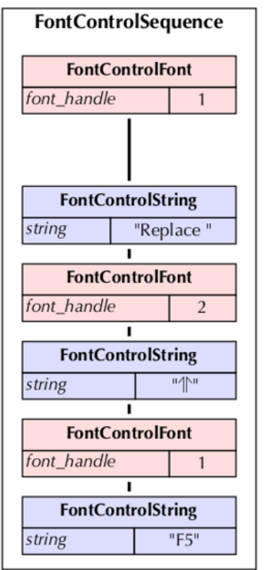
The sequence contains objects which are based on the `FontControlBase` class. Each object is an instance of a class which can perform the changes to the `FontContext`, or the necessary rendering for that operation when methods are called.

Fonts

Proper control code parsing (3a)

Font control sequence processing





Fonts

Proper control code parsing (3a)

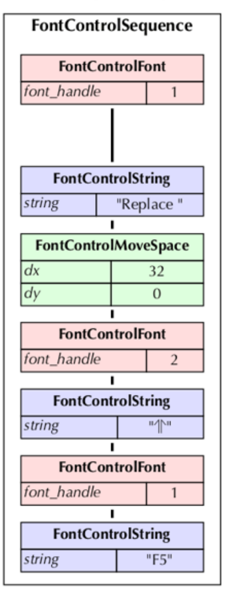
That's a lot of description so here's what actually happens when you try to render the `Replace <shift>F5` string for a menu. When the FontManager initialises, it creates a `FontContext` which holds the current colours, current font, and other information. This will be updated when a `Font_Paint` is performed, colours are selected or a font is selected.

Here we can see that the FontContext contains the requested parameters - we're going to plot with the current font (which is unset), we have black on white, and we're plotting at 640, 480. The `FontControlParser` is called to generate a list of objects in a `FontControlSequence` from the input string.

This is what you can see here - there's font selection operations and strings to be rendered. To render this, the `Font_Paint` SWI asks the `FontContext` to perform a `paint` operation.

Fonts

Proper control code parsing (3b)



Fonts

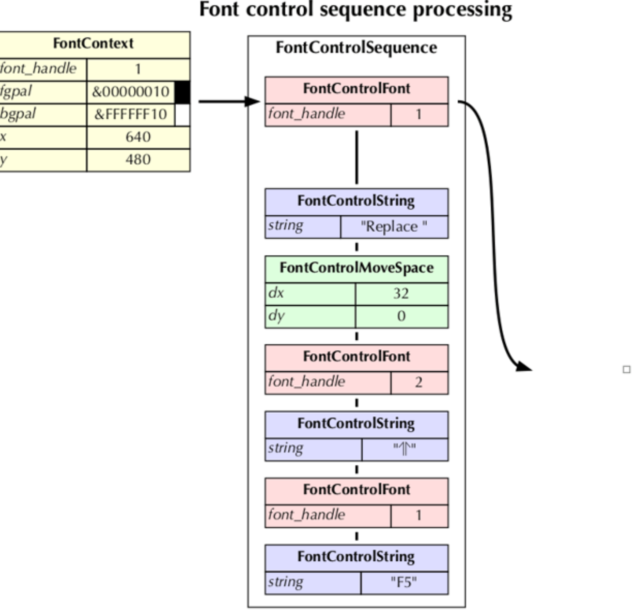
Proper control code parsing (3b)

The first thing it does is to expand the sequence to include an extra movement after the space for the word spacing.

You can see the extra movement has been added, as the green `FontControlMoveSpace`. This is a variant of the regular `FontControlMove` control code, which will also add an underline in between, if an underline was required. A regular move control code would not produce an extra underline. It's this different type of object that will avoid the problem with the underline being split up as we saw in the earlier example.

Fonts

Proper control code parsing (3c)



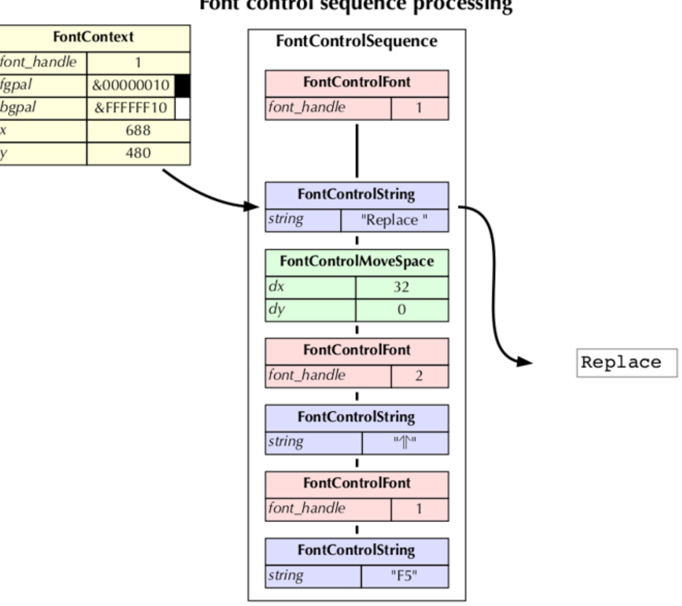
Fonts

Proper control code parsing (3c)

Now the `FontContext` painter steps through each of the objects and asks it to perform a `paint` operation. On the right hand side, I've drawn what is present as after the operation completes. At this point, there's still nothing on the screen. However, the context has been updated to reflect the new font that we need to use.

Fonts

Proper control code parsing (3d)



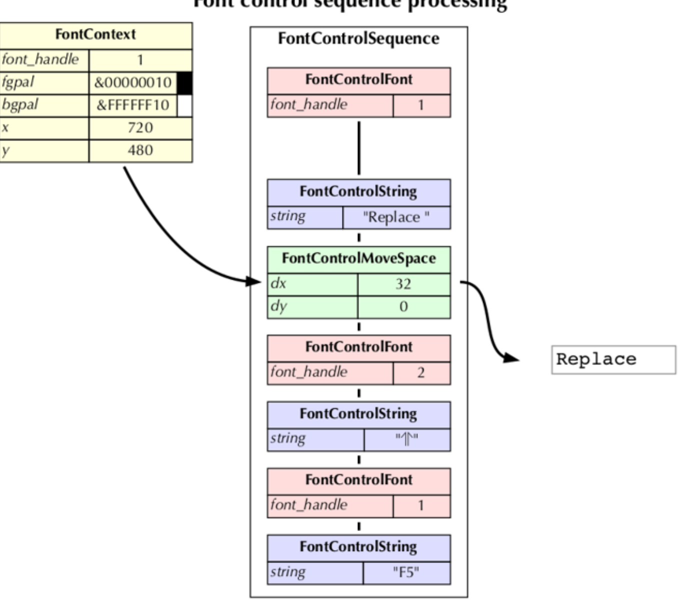
Fonts

Proper control code parsing (3d)

We draw the word `Replace`, and a space following it, and the baseline position in the context is incremented by the size of that string.

Fonts

Proper control code parsing (3e)



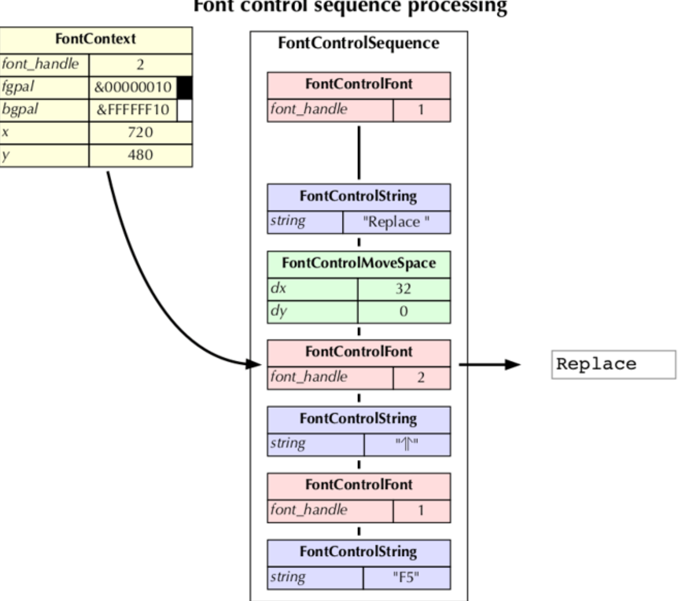
Fonts

Proper control code parsing (3e)

Then we move a little further - just the baseline x position is updated.

Fonts

Proper control code parsing (3f)



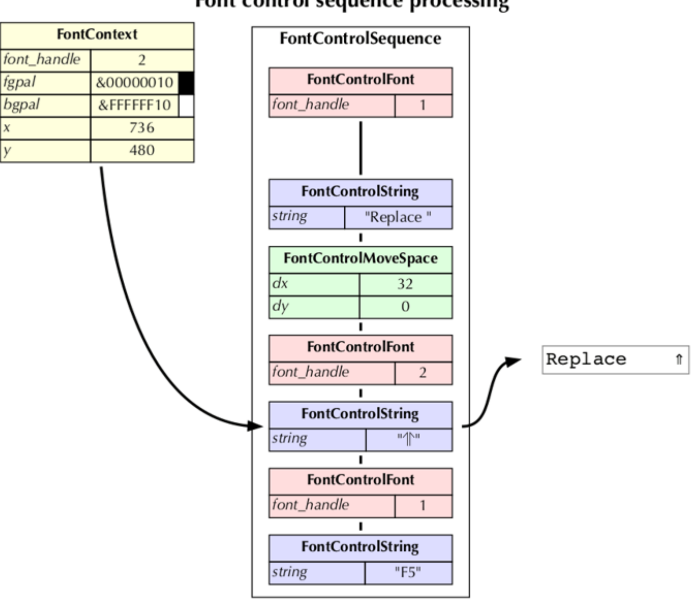
Fonts

Proper control code parsing (3f)

Switch to the WIMPSymbol font by updating the context.

Fonts

Proper control code parsing (3g)



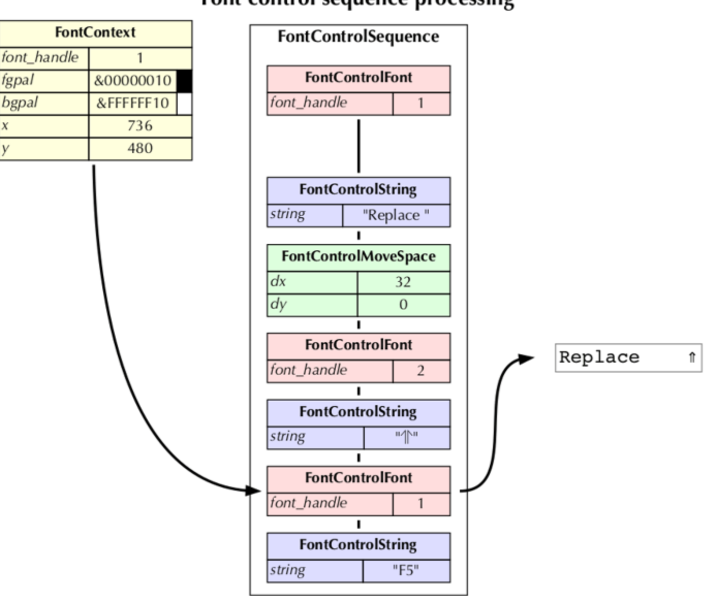
Fonts

Proper control code parsing (3g)

Draw the shift character, and advance the baseline position.

Fonts

Proper control code parsing (3h)



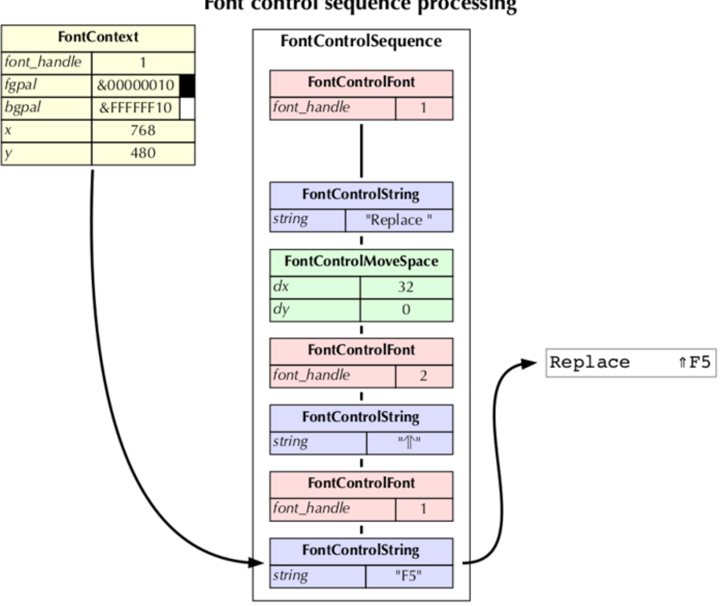
Fonts

Proper control code parsing (3h)

Switch back to the desktop font.

Fonts

Proper control code parsing (3i)



Fonts

Proper control code parsing (3i)

Finally draw the `F5`.
At this point the `FontContext` returns, and `Font_Paint` can return to the user. For shear simplicity, I've omitted the bounding box from these diagrams, but after each operation the bounding box of the covered area is updated in the context. The `OS_ChangedBox` bounds will be updated if they have been configured to do track updated regions.

Fonts

Proper control code parsing (4)

What about `Font_ScanString` and friends?

- They do the same thing, but call `size` instead of `paint`.
- At the end they can return other parameters such as the size and indexes.
- They update a `future_context` for `Font_FutureFont` or `Font_FutureRGB`.

Fonts

Proper control code parsing (4)

What does this mean for `Font_ScanString` and its friends?
Well, it does exactly the same process, but instead of calling `paint`, it calls `size`.
This does almost exactly the same as `paint`, but doesn't render, and if a terminal condition like stepping past the limits is reached, the scan ends. The diagram above omits it, but each of the objects also contains a property which gives the index of the end of the operation in the original string. This is used to return the correct offset when the `Font_ScanString` needs to say where the scan ended.
`Font_ScanString` also does something slightly different with the `FontContext`. Instead of operating on the current context, it copies the current context to a `future_context` and operates on that instead. It is this `future_context` that will be queried when the user calls `Font_FutureFont` or `Font_FutureRGB`.
This was all implemented as a standalone test first, and then updated in the FontManager code. This meant that the complexities of different types of operations could be experimented with, and avoided problems caused by the rest of the system.

Fonts

How's it compare?

Plain string
Rubout box [À¶]
Extra word spacing
Extra char spacing
Justify text
Matrix
Controls: Red
Controls: Matrix
Controls: *2nd Font*
Controls: ... ^⇑F12
Controls: Text Up Right
Controls: Underlined/~~Strike~~
Controls: Underlined + spacing

Fonts

How's it compare?

This is what the new parser's rendering looks like. It's looks the same as the old parser, except that the underline is no longer broken up. Plus, it can now support the `Font_ScanString` operations as well.

It is slightly different though. Because `Font_ScanString` honours these control codes and returns data in a slightly different way, some of the bounding boxes are very slightly different. However, it's more consistent now, so that's alright.

I think that the structure I've implemented is probably similar to the Classic FontManager, but my recollection is that the assembler in the FontManager was a lot more adhoc with its use of data structures.

If you're interested, there is a repository on GitHub which contains the entire parser and tests for the font control code system.

Fonts

What does it look like?

`Font_ScanString` has code that looks like this:

```
self.context.copy(to=self.future_context)
self.future_context.select_font(rohandle)

memstring = self.ro.memory[regs[1]]
fc = FontControlParserPyromaniac(self.ro)
fc.debug_enable = self.debug_fontparser
fc.parse(memstring, string_length)

self.future_context.transform = transform

split_char = chr(split_character) if split_character is not None else None
(split_offset, splits) = self.future_context.size(fc.sequence, spacing=spacing,
                                                   limits=(xmilli, ymilli),
                                                   split_char=split_char)
```

Fonts

What does it look like?

`Font_ScanString` has a lot of code in it, mostly related to working out what you want it to do. But the core of the code looks like this.

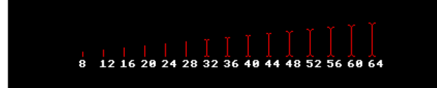
We set up the `future_context` as a copy of the current context - so that we can work on it without affecting the current context.

We apply the parser to the string that was supplied. This gives us a sequence of operations as above. And then we do the `size` operation on it - that updates all the properties in the `future_context`.

The properties from the `future_context` are then returned to the user in the registers or blocks as required.

Font caret

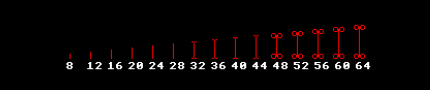
You can see here what the caret looks like at different heights:



Pyromaniac also offers you the option of putting loops on the ends; here's what the PRM says:

> *The height of the symbol, which is a vertical bar with 'loops' on the end, can be varied to suit the height of the text, or the line spacing.*

When enabled, the loops look like this:



Fonts

Font caret

The font caret shows where your text will be entered. It's a pretty simple rendering, but one that the Classic FontManager drew by directly writing to the screen. In RISC OS Pyromaniac we draw the caret using the graphics primitives.

It changes its shape based on the height requested. You can see here what the caret looks like at different heights.

The documentation actually says that the caret will be drawn with loops at the top and bottom. To the best of my knowledge this has never actually been the case. But it is an option that you can enable in Pyromaniac. And this is what it looks like.

I really don't like it. Perhaps spending some more time would help, but the point of Pyromaniac is to make it easy to try things - this one isn't quite right, but it wasn't too hard to try out.

Screen Modes

Introducing deep modes (1)

- Deep modes need to be handled like paletted modes.
- Palettes within the modes aren't indexed.

New `Palette` objects created with common interface:

- `palette.copy()`: Creates a copy of the palette.
- `palette.key()`: Return a hash value for this palette.
- `palette.lookup(rgb)`: Returns an exact colour number from an RGB.
- `palette.find_closest(rgb)`: Find the closest colour to the RGB value.
- `palette.find_furthest(rgb)`: Find the furthest colour from the RGB value.
- `palette.generate_32k_table()`: Return a 32K array of colour numbers.

Screen Modes

Introducing deep modes (1)

Whilst discussing the changes for the sprites, I talked a little bit about the use of mode selectors, and how the system needs to handle them wherever we have a mode supplied. This went hand in hand with adding support for deep modes - 15 bit per pixel and 24 bit per pixel modes.
Adding those modes was surprisingly easy. The palette system needed to be updated so that they didn't *have* to be indexed. Places where the system iterated over the whole palette were updated to be aware of the new depths. Each of the palette objects has a number of common methods which can be used to operate on the palette.
The modes that had a palette would be able to access the individual paletted entries. To find the closest colour they need to do a search. The deep modes, however, could just use simple calculations to return the closest colours.
The 32K tables take a long time to generate - they are essentially 32 * 32 * 32 calls to the `find_closest()` function. Because of this, the `ColourTrans` module keeps a cache of translation tables for common palettes. This means that we do not need to waste time building them at run time. This is very similar to how the Classic `ColourTrans` module handled translation tables.
These palette operations are obviously at the heart of drawing anything onto the display and handling sprites. They could be optimised in the future if they are found to be a bottleneck, but right now the performance isn't terrible.

Screen Modes

Introducing deep modes (2)

Buggy code:

```
def lookup(self, rgb):
    """
    Look up a colour from the palette by RGB value.

    @param rgb: The RGB value (&BBGGRRxx) to lookup

    @return: index of the colour, or -1 if not found
    """
    r = (rgb>>8) & 255
    if (r & 7) != (r >> 5):
        return -1   # Inexact colour
    g = (rgb>>8) & 255
    if (g & 7) != (g >> 5):
        return -1   # Inexact colour
    b = (rgb>>24) & 255
    if (b & 7) != (b >> 5):
        return -1   # Inexact colour

    return (r>>3) | ((g>>3)<<5) | ((b>>3)<<10)
```

Screen Modes

Introducing deep modes (2)

There was a fun bug introduced when the palettes were created. In the 15 bit per pixel modes, the selection of the colours in the `lookup` function was a little broken...

The problem was the the red component was mapped into green as well - the shift in the green assignment should be moving the value by 16 bits, not 8. It meant that the colours in that mode had no green component at all.

Because the deep modes were harder to test automatically, this wasn't caught for a month. Another reminder of why it's important to have automated tests, even when it's hard.

Screen Modes

Mode strings (1)

Processing mode strings is needed so that BASIC for: `MODE "X800 Y600 C16M"`.
3 new SWI reasons were needed:

- `OS_ScreenMode 13`: Decode mode string to a mode specifier.
- `OS_ScreenMode 14`: Encode mode string from a mode specifier.
- `OS_ScreenMode 15`: Select mode by mode string.

Screen Modes

Mode strings (1)

When modes were just numbered, it was easy to select them. Using `VDU 22` or `OS_ScreenMode 0` was all that was necessary. However, the mode selection can also be performed by a mode string. There are 3 reason codes in `OS_ScreenMode` which process the mode strings into mode specifiers.
These calls allow the specification of mode as a string and are required for BASIC's to mode selection. This was surprisingly easy to add. The `ModeSelector` class had already been created which was able to hold all the details about a mode.

Screen Modes

Mode strings (2)

Simple!

```
@handlers.osscreenmode.register(osscreenmode.ScreenModeReason_SelectModeString)
def OS_ScreenMode_15(ro, reason, regs):
    """
    OS_ScreenMode 15 (Select mode by mode string)
    => R0 = 15
       R1 = pointer to mode string

    This SWI is used to select a mode, given a mode string. Internally this is
    implemented as a conversion to a mode specifier (OS_ScreenMode 13) and
    mode selection (OS_ScreenMode 0), and is provided for convenience.
    """
    mode_string = ro.memory[regs[1]].string_ctrl

    with ro.kernel.da_sysheap.allocate(20 + 2 * 4 * 13 + 4) as modesel:
        spec = ModeSelector(ro, string=mode_string)
        buf = BufferData(ro, modesel)
        spec.write_selector(buf)

        ro.kernel.vdu.select_mode(modesel.address)

    return True
```


Screen Modes

Mode strings (2)

The string decoding and encoding was implemented within this class as methods, and a simple interface used to write the selector to memory. It's great when the code needed to implement interfaces is really obvious. The code needed to implement the *Select mode by mode string* interface was very readable.

- First we allocate some memory to put the mode selector in.
- Then we create a mode selector using the string.
- We write it into the buffer.
- And then use it to select a mode.

The memory size could probably do with a bit more commentary, but it's still pretty readable.

Screen Modes

Mode strings (3)

- The mode strings had originally been processed by the WindowManager.
- This presented a problem for greyscale modes - the `G` specifier.
- `Decode mode string` followed by `Select mode` would not know that it should be greyscale.
- New flag in the mode flags (mode variable 0) for greyscale modes (bit 9).
- The Mode objects spot this flag and report the default palette as greyscale.

Screen Modes

Mode strings (3)

One of the reasons that the mode strings parsing was moved into the Kernel, was that it had previously been inside the WindowManager. BASIC had been modified to call into the WindowManager when a mode string had been used. This is clearly insanity - the WindowManager should not need to be present for a language interpreter to work.
However, it introduced an interesting problem. The selection of greyscale modes was a WindowManager thing. There was no way to express in a mode selector that the mode should be greyscale. The WindowManager didn't care - it knew that that was the type of mode requested and reset the palette after it changed mode.
But how was that to translate to the 'decode mode string, select mode' sequence so that when you specified a greyscale mode, the mode selected would actually have a greyscale palette? The solution was to introduce a new mode flag for greyscale which allows this information to be passed through.
That's what Select introduced, and of course the same problem exists here in RISC OS Pyromaniac, so the mode flag for greyscale modes was passed through. When the flag is seen by the mode information code, it changes the default palette for the mode to a greyscale palette.

Screen Modes

Mode enumeration

- `OS_ScreenMode 2` performs enumeration through `Service_EnumerateScreenModes`.
- Usually this would be handled by ScreenModes, using a loaded MDF.
- RISC OS Pyromaniac instead enumerates through the numbered modes.
- You can still select other modes manually.
- ScreenModes can be run and be used to load an MDF, if you really wanted.

Screen Modes

Mode enumeration

With RISC OS 3.5, the system brought in a system of monitor specific modes which could be enumerated by a call to `OS_ScreenMode`. The list of modes that were available in this way were generally serviced by the ScreenModes module.

This module manages the list of modes available by using a Monitor Definition File to describe what the monitor can handle. There isn't any real reason why that has to be the case, and indeed on modern systems it's unlikely that you'd have any monitor definition except to override when the monitor provided incorrect information.

RISC OS Pyromaniac doesn't have any hardware, and so very few of the restrictions that it would bring. The enumeration call is now issued by the kernel when requested, and it can be configured to respond with the details from the numbered modes. This means that any numbered modes that are defined through the Mode Extension interface will appear as options to be selected. You can still select any other resolution you like manually. But the modes that are known to the system through the numbers will appear in the enumerated list.

79 / 121

Screen Modes

Multiple displays

- RISC OS Select allows multiple displays to be connected and switched between.
- RISC OS Pyromaniac doesn't yet allow that.
- But it can describe the display that is connected.
- It will be fun introducing multiple displays!

Screen Modes

Multiple displays

RISC OS Select introduced the multiple display interfaces, allowing you to switch between display drivers at will. Eventually, RISC OS Pyromaniac will support this, but right now, there's only one display.
However, the calls that read information about the display drivers are now supported. `OS_ScreenMode 12` will return details about the current graphics implementation as its details. This means that anything which tries to read the information about the displays will get back useful information.
Providing multiple display drivers in RISC OS Pyromaniac is probably pretty easy. There's a bunch of entangled information about the graphics system and its presentation, but that's the same as was the case on Select. The harder part will be convincing the interface implementation like WxWidgets to redraw the different displays independently. They just weren't written with that in mind.
It'll be fun, but it will be later.
Before I move on to talk about documentation, are there any questions about the font system of screen modes in Pyromaniac?

4. Documentation

4. Documentation

Let's move on to talk about documentation.

Documentation

Pyromaniac's APIs

- The PRM-in-XML project was created in 2001 to make it possible to migrate documentation to a more maintainable format.
- Pyromaniac had documentation for some of the changed APIs in this format.
- In particular, `OS_AMBControl` is fully documented.
- These have been expanded from 3 to 8 documents.

Documentation

Pyromaniac's APIs

One of the very last things that was done on Pyromaniac before the presentation last year was to begin creating API documentation. This would go alongside the main product documentation to explain some of the differences in its the Operating System. I had, over the previous year, begun to resurrect the PRM-in-XML documentation project. This was a project started in around 2001 and which was made public at that time to very little interest.
Essentially the project intended to replace all the documentation in its disparate forms with structured documentation specially designed for describing interfaces used by RISC OS. I described this in much more detail in an article on Iconbar earlier this year, so if you're interested in the rationale and some more detail about it, you can find out more there.
For Pyromaniac, I needed to describe the details of how the interfaces were different where it mattered. Primarily this was for new APIs, or places where existing documentation didn't exist. There are a few new APIs, but the one that caused me to need some documentation was the `OS_AMBControl` SWI.
This SWI manages 'Application Memory Blocks' - the memory that lives in application space.
It had no real documentation, and it was a vital call necessary for the WindowManager to function. Not only was there no documentation, it would be necessary for parts of the behaviour to differ from the Classic version. The reasons for the deviation were largely related to simplifying the behaviour. In some cases, the interface does not translate well to a modern system.
PRM-in-XML was an ideal way to do this, and the documentation was added for a few pages. Last year there were 3 API documents in Pyromaniac. Now there are 8 API documents. Not a huge number for an Operating System, but it's a step in the right direction.

Documentation

PRM-in-XML documentation project

- An article was written for Iconbar to explain why PRM-in-XML exists, and what it can do.
- Alan Robertson got involved, and tried out process of creating documents.
- He found many issues, which we addressed some of.
- I created a staging area to collect converted documents.
- Alan converted some functional specification documents from HTML to PRM-in-XML.

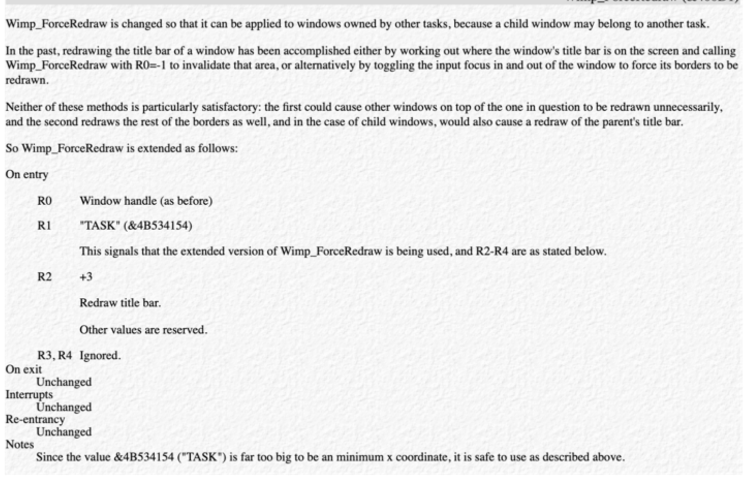
Documentation

PRM-in-XML documentation project

As I mentioned, I'd rebuilt the original PRM documentation so that it could be used as the basis for full documents. I intended to talk to some people about getting this released because it was some good work and there had been very little useful effort towards documenting RISC OS properly. In my opinion.
After writing the article about documentation, there was some good interest from Alan Robertson to try it out. I gave him access to the tools and a skeleton guide that I'd written for 'how to convert documents to PRM-in-XML'.
He very quickly found a number of problems, and asked a lot of useful questions. The problems ranged from the tools not working on RISC OS, to bugs in my documentation and the transformation. After a little while, I set up a little staging area for documents that were worth capturing but weren't ready for the PRMs yet. This was also a little playground to try things out.
Alan has converted a number of functional specifications to the PRM-in-XML format. This showed a number of problems with the way in which it was structured. The tool was designed for PRM-like documentation. The manner in which Acorn wrote its functional specifications didn't translate directly - but you could definitely make a good showing with it.

Documentation

Alan's conversions



Wimp_ForceRedraw (&400D1)

Wimp_ForceRedraw is changed so that it can be applied to windows owned by other tasks, because a child window may belong to another task.

In the past, redrawing the title bar of a window has been accomplished either by working out where the window's title bar is on the screen and calling Wimp_ForceRedraw with R0=-1 to invalidate that area, or alternatively by toggling the input focus in and out of the window to force its borders to be redrawn.

Neither of these methods is particularly satisfactory: the first could cause other windows on top of the one in question to be redrawn unnecessarily, and the second redraws the rest of the borders as well, and in the case of child windows, would also cause a redraw of the parent's title bar.

So Wimp_ForceRedraw is extended as follows:

On entry

R0 Window handle (as before)

R1 "TASK" (&4B534154)

This signals that the extended version of Wimp_ForceRedraw is being used, and R2-R4 are as stated below.

R2 +3

Redraw title bar.

Other values are reserved.

R3, R4 Ignored.

On exit
Unchanged

Interrupts
Unchanged

Re-entrancy
Unchanged

Notes
Since the value &4B534154 ("TASK") is far too big to be an minimum x coordinate, it is safe to use as described above.

Wimp_ForceRedraw
(SWI &400D1)

On entry

R0 = Window handle (as before)
R1 = "TASK" (&4B534154)

This signals that the extended version of Wimp_ForceRedraw is being used, and R2-R4 are as stated below.

R2 =

Value	Meaning
+3	Redraw title bar
	Other values are reserved

R3 - R4 = Ignored

On exit

unchanged

Interrupts

Interrupts are undefined
Fast interrupts are enabled

Processor mode

Processor is in svc mode

Re-entrancy

SWI is not re-entrant

Use

Wimp_ForceRedraw is changed so that it can be applied to windows owned by other tasks, because a child window may belong to another task.

In the past, redrawing the title bar of a window has been accomplished either by working out where the window's title bar is on the screen and calling Wimp_ForceRedraw with R0=-1 to invalidate that area, or alternatively by toggling the input focus in and out of the window to force its borders to be redrawn.

Neither of these methods is particularly satisfactory: the first could cause other windows on top of the one in question to be redrawn unnecessarily, and the second redraws the rest of the borders as well, and in the case of child windows, would also cause a redraw of the parent's title bar.

So Wimp_ForceRedraw is extended as shown above.

Note: Since the value &4B534154 ("TASK") is far too big to be an minimum x coordinate, it is safe to use as described above.

Related APIs

None

Documentation

Alan's conversions

This is an example of one of the conversions that Alan made. It updates the Acorn-era functional specification for the Nested Wimp to be documented in the PRM-in-XML format.

The content is largely unchanged and that's the point. The conversion hasn't changed the meaning; only the presentation. Updating the content to be written more like the PRMs would be relatively trivial, and then the documentation could be dropped into the regular manual.

In this case that's not quite true because this is one of the special extended reason codes for `Wimp_ForceRedraw` . But it's a lot easier to work with in this form - and in a more familiar form.

Alan has converted a number of documents, and they're going quite well. The biggest problem is my having time to review them.

Documentation

Updating the PRMs

- Tools now work on RISC OS and linux.
- They're more flexible about what they can do and how they report errors.
- The PRMs themselves have been updated to include more documents which had not been present previously.
- The results are looking good so far.

Documentation

Updating the PRMs

Meanwhile, I updated the tools to add some new features and make them more reliable for users - myself and Alan at this stage. There was a lot of progress over a few months, and then I gave him access to the partially converted PRMs.
He's been updating those and adding new bits. It's going well, but there's still a lot to do. I've done a lot of the ground work and I don't see myself converting these documents, so it makes sense for someone to come at it with fresh eyes, and I can do some of the tooling work.
Back when the original PRM-in-XML documents were created, Andrew Hill did a lot of work in converting them based off the original HTML versions that David Thomas produced. Working with Alan has been really good fun so far, and I'm hopeful that this work will become available to other people sometime in the new year.
It's been a broad collaboration amongst a number of people over a long period of time - it feels like it's coming together now.

Documentation

Collaboration (1)

- Wanted a way to represent key and mouse input.
- Original method: `&shift;` and `&ctrl;` and the like are pretty inflexible.
- Try some elements to describe modifiers...

```
<key ctrl='yes' shift='yes' alt='yes' meta='yes'>X</key>
```

- Some other ideas...

```
<mouse shift='yes' button='select'/>
```

```
<input key='ctrl'/><input key='x'/>
```

```
<input shift='yes' mouse='select'/>
```

Documentation

Collaboration (1)

There were a lot of little things that Alan has suggested, and some things that I've brought in. I'm going to just give one example that was very effective.

In the original PRMs back in the early 2000s I wanted to allow people to include key input that looked like the RISC OS documentation - using the upward arrow for shift, and caret for ctrl, and labelled use of the cursor keys. I wanted this to be flexible so I added some entities for these keys.

The problem with that is that it's not actually flexible enough. In some cases you want the keys to be displayed differently, and you cannot easily do this with entities. Alan had asked the best way to describe key presses and I wanted to make things more flexible.

At first I suggested a simple representation that allowed modifiers.

Which would be simple to write, and would allow the representation to be decided by the transformation.

However, this implies that you can only combine some modifiers with keys - pre-defined ones.

I set out a number of things that we need to be able to represent, like presses, modifiers, mouse button actions, and sequences. From this I suggested a number of different ways of describing them, and Alan pulled in some examples from how DocBook and Wikipedia handle this.

A number of different ideas were tried, and some are shown here. We finally settled on a grouping `input` element, with some more support for mouse input type (scroll wheels and the like), which looks like this:

Documentation

Collaboration (2)

Eventual layout and example output:

```
<input><key name='W'></input>
<input><key name='ctrl'/><key name='X'/></input>
<input><mouse name='select' repeat='2'/></input>
<input><mouse name='select' action='drag'/></input>
<input><key name='shift'/><mouse name='select'/></input>
```

- W - the W key
- CTRL X - control and X (eg a cut)
- 2×SELECT - double click select (eg run a file)
- SELECT - drag select (eg making a selection)
- Press SELECT - press select (eg starting a selection)
- SHIFT SELECT - shift + select click (eg adding to a selection)
- ESC - press escape twice (eg some special operation to get out of a system?)
- SHIFT CTRL F12 - shift + control + F12 (eg shutdown)

Documentation

Collaboration (2)

Once we'd settled on a way of representing the key sequences in the XML, the transformation could be updated to present it in a nice way. The nice thing is that it can change the style relatively easily when it's been structured correctly. One of the reasons for using the well known names like `select` and `ctrl` was that those could be translated into their common forms of SELECT and ^, but if we decide in the future that we want to change that presentation, we can do so in the transformation without changing every single document. The transformation that creates HTML 5 output is quite nice for these input sequences, and even has a special option to make function keys red. The example picture shows what these keys look like in the current development version. It's just CSS, so if you don't like it you can restyle it however you like.

Documentation

Making it look good

- The most recent version for the transformation is what you have seen here.
- This transformation supports HTML5 and CSS, with new elements planned to make it easier to structure certain elements.
- New styles can be applied to the CSS to change how it looks for a given purpose.
- A number of 'variants' of the styles are supplied and can generally be overlaid.
- This allows people to change what they don't like in the styling, and obviously the structured content itself is easier to manipulate.
- Acorn applied different styles over the years, with each manual looking subtly (or strikingly) different to the one that went before.

Documentation

Making it look good

As always it's a matter of time and energy that decides what you can do. I spent a lot of time working on fixes and updates to make the PRM-in-XML tranformations HTML 5 and CSS compliant... and then making them look good.

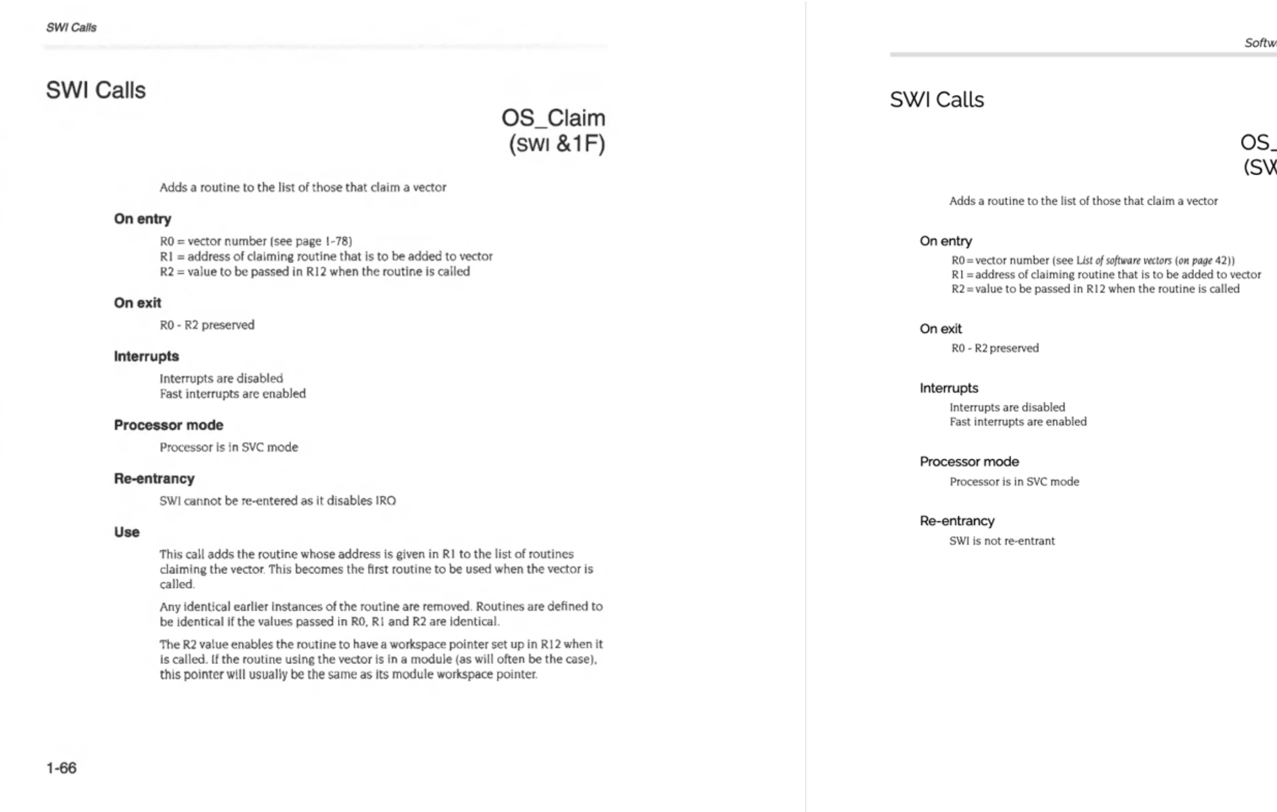
'Looking good' is a pretty subjective thing, but the goal with the PRM-in-XML tool is that you should be able to re-style or restructure things if you want, so what is really needed is a good baseline that people can work from. The style used by the PRM is the baseline I'd chosen, and I think what I've produced matches it pretty well.

However, I wasn't happy with just one variation. There was an excessive amount of focus on getting the fonts right in some of the discussions online. This is, to me, a typical example of bike-shedding - discussing the trivialities to the exclusion of what's actually important. What's important about manuals is conveying information, and whilst it is important to get a good font, it's by no means the most important thing.

That said, people have strong opinions about it, so obviously it's useful to be able to accomodate them. The PRM-in-XML variations allows different styles to be chosen, either from a pre-configured set that I've put together, or which you choose yourself.

The font is one thing, but actually there have been quite different styles used for the manuals over the years. The RISC OS 3 PRMs, and the supplement for 3.5, are the best known, although the supplement itself has a slightly different style. However, this differs considerably from the style used by RISC OS 2's PRMs.

88 / 121



SWI Calls

SWI Calls

OS_Claim
(SWI &1F)

Adds a routine to the list of those that claim a vector

On entry

R0 = vector number (see page 1-78)
R1 = address of claiming routine that is to be added to vector
R2 = value to be passed in R12 when the routine is called

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered as it disables IRQ

Use

This call adds the routine whose address is given in R1 to the list of routines claiming the vector. This becomes the first routine to be used when the vector is called.

Any identical earlier instances of the routine are removed. Routines are defined to be identical if the values passed in R0, R1 and R2 are identical.

The R2 value enables the routine to have a workspace pointer set up in R12 when it is called. If the routine using the vector is in a module (as will often be the case), this pointer will usually be the same as its module workspace pointer.

1-66

Software vectors

SWI Calls

OS_Claim
(SWI &1F)

Adds a routine to the list of those that claim a vector

On entry

R0 = vector number (see *List of software vectors (on page 42)*)
R1 = address of claiming routine that is to be added to vector
R2 = value to be passed in R12 when the routine is called

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

51

This is an example of a PRM-in-XML formatted page compared to same page from the the RISC OS 3 manuals.

- On the left we have a scan of the `OS_Claim` SWI from the paper manuals.
- On the right, you can see the same page but this time generated from the PRM-in-XML content using the RISC OS 3 PRM variant.

The spacing is a little different and the fonts are slightly different for the headings, but it's a pretty fair version of the documentation. More importantly, it contains the information you need.

89 / 121

SWI Calls

OS_Claim
(SWI &1F)

Adds a routine to the list of those that claim a vector

On entry	R0 = vector number R1 = address of claiming routine R2 = value to be passed in R12 when the routine is called
On exit	R0 - R2 preserved
Interrupts	Interrupts are disabled Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI cannot be re-entered as it disables IRQ
Use	This call adds the routine whose address is given in R1 to the list of routines claiming the vector. This becomes the first routine to be used when the vector is called. Any earlier instances of the same routine are removed. Routines are defined to be the same if the values passed in R0, R1 and R2 are identical. The R2 value enables the routine to have a workspace pointer set up in R12 when it is called. If the routine using the vector is in a module (as will often be the case), this pointer will usually be the same as its module workspace pointer. See below for a list of the vector numbers. Example:

```
MOV R0, #ByteV
ADR R1, MyByteHandler
MOV R2, #0
SWI "OS_Claim"
```

Related SWIs	OS_Release (SWI &20), OS_CallAVector (SWI &34), OS_AddToVector (SWI &47)
Related vectors	All

58 Software vectors: SWI Calls

SWI Calls

OS_Claim
(SWI &1F)

Adds a routine to the list of those that claim a vector

On entry	R0 = vector number (see *List of software vectors (on page 39)*) R1 = address of claiming routine that is to be added to vector R2 = value to be passed in R12 when the routine is called
On exit	R0 - R2 preserved
Interrupts	Interrupts are disabled Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call adds the routine whose address is given in R1 to the list of routines claiming the vector. This becomes the first routine to be used when the vector is called. Any identical earlier instances of the routine are removed. Routines are defined to be identical if the values passed in R0, R1 and R2 are identical. The R2 value enables the routine to have a workspace pointer set up in R12 when it is called. If the routine using the vector is in a module (as will often be the case), this pointer will usually be the same as its module workspace pointer. Note that this SWI cannot be re-entered as it disables IRQs.

46 Software vectors: SWI Calls

This is the same page, but using the RISC OS 2 PRM style.
Again, the original printed version is on the left, and on the right is the PRM-in-XML transformed manual, but using the RISC OS 2 variant.
As you can see the style differs quite a bit from the RISC OS 3 manual. Obviously there's a big difference in how the headings are laid out, and the dividing line which separates the content. This style was a little harder to generate because it doesn't fit quite so easily with the CSS, but the reproduction is pretty good.

Documentation demo

Documentation demo

As you can see there, the tooling can produce these different styles from the same source. Not only that, but using the CSS paged media styles, it's possible to produce good quality documents.

Those of you who are in the Pub will be able to see a couple of example documents I've produced. These just have a few chapters in them, but they show the type of output that can be produced when printed. There's two styles there - the RISC OS 2 style and the RISC OS 3 style.

I'll quickly flick through the pages of the documents so that those of you not in the pub can see what they're like.

<Load up the prm-networking-example.pdf and share its window> <move to the contents page>

The RISC OS 3 variation is similar to what you might expect, and there's not a lot to say about the content. However, you should be able to see that the contents page covers all the chapter details. The page numbers are all linked, just like you'd expect and although there are only 3 chapters in this example, you can see that they cover a few different areas.

<Click on the Resolver chapter>

If we have a look at the Resolver chapter, we can see that it's structured in the usual way - the Resolver document was written based on earlier documentation the described the interface, but formatted into the correct style.

<scroll to the technical details section>

You can see that SWI references have page numbers listed beside them, just like the contents.

<Scroll to the structure definition for Host entries>

The host entries here are described with a structured offset table. These structured tables mean that they can be restyled if we need to. If we go to a SWI we can see how the interfaces are structured.

<click on the SWI Resolver_GetHost entry>

Again, the SWIs are presented as you'd expect, with all the usual information available. All the fields are

5. Testing

5. Testing

You might remember that one of the things I care about is testing, so let's talk about that...

Testing

How do you test these features

- Testing in most of RISC OS Pyromaniac is through expectation tests.
    - That means we write some output, and we compare it to what we expect.
    - Differences mean the tests fail.
- But that's not quite as easy for graphics.
    - Unless you make the graphics pixels into text too.
    - `PBM` files (plus `PGM`, `PPM` strictly) can be text forms of the graphics.
- When rendered into a small mode, the comparison becomes more manageable.

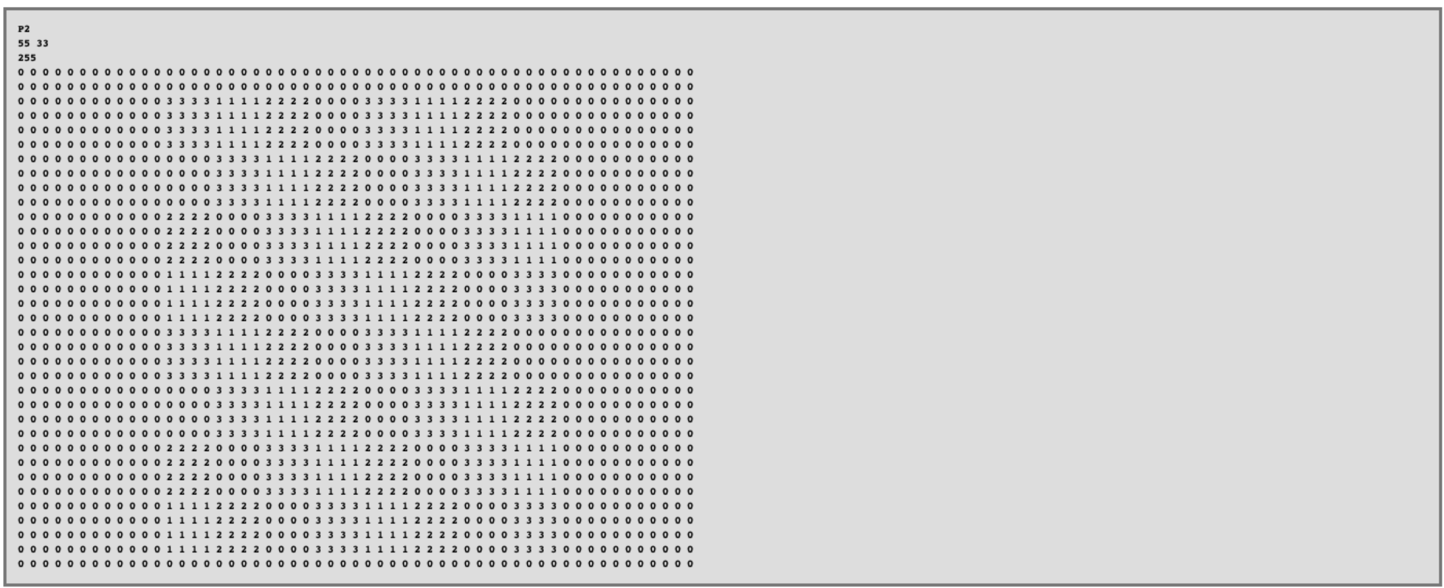
Testing

How do you test these features

In my recent presentation about software testing I talked a little about how you can test operating system features. Essentially I have a lot of expectation testing of the commands and system calls, using the text output to track whether things work as needed. That's fine when you're exercising the *Commands, or simple SWI interfaces, but for graphics it gets a bit harder.
But really it's only a *bit* harder.
In other places I've done some work with producing diffs of images, and presenting the results nicely, but here I'm dealing with simple images, in paletted modes. The simplest way to handle it is to save the screen to a textual image format and compare the results in the normal way.
The textual image format I use is PBM, because it's simple. Obviously, such output files are viewable by the RISC OS thumbnail system through ImageFileConvert, but I'm using macOS... and macOS supports them through its preview system too.
Of course, saving the whole screen to display simple graphics would be overkill in most modes, so the graphics tests use a custom mode which is just 80 pixels square. That's not a lot, but it is enough to exercise the tests.

Testing

Testing sprites with text



```
P2
55 33
255
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 1 1 1 1 2 2 2 2 0 0 0 0 3 3 3 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```


Testing

Testing sprites with text

This is what the expectation files look like on disc. Actually this is slightly cut down to fit on the slide. The PBM format describes the width and height of the image at the header of the file, and the maximum colour number that is used.

Then each of the colours is listed. Depending on how well you spot patterns you may be able to see a chequer-board like pattern of different colour numbers in the middle.

There 3s, then 1s, then 2s, then 0s.

As that's the colours that are in the original sprite, that's correct.

Because the system compares the different text files, any time that the graphics system puts the wrong colours down, we get a difference in the expectations and the tests fail.

Testing
ColourTrans testing

- ColourTrans testing can be hard when there are so many combinations.
- It just deals with numbers, so we need to check they're the right numbers.
- Some interfaces like `ColourTrans_ReturnGCOL` can have some sampled tests.

```
Test ReturnGCOL conversion
Colour &0000ff00 => 1           Opposite  => 6
Colour &00ff0000 => 2           Opposite  => 5
Colour &ff000000 => 4           Opposite  => 3
Colour &ffffff00 => 7           Opposite  => 0
Colour &80808000 => 7           Opposite  => 0
Colour &81397900 => 4           Opposite  => 3
```

Testing
ColourTrans testing

ColourTrans is quite frustrating to test. We can ignore the fact that it's dealing with colour, generally - it's just manipulating arrays of numbers. The simple calls like `ColourTrans_ReturnGCOL` will return the colours being used, and we can compare them to some simple expectation.
This is an example for that SWI in a standard 16 colour mode. It's testing both the conversion to a GCOL number and to the opposite colour number.
The same sort of test is also run in other modes with different depths to check that their behaviour too. And then you get to the 256 colour modes which have crazy colour numbers and tints.
I've compared my results with those from RISC OS Classic and I think I'm getting it to do the right thing.
`ColourTrans_GenerateTable` is another matter though. There are many combinations of parameters that can be supplied to the call, and I only have 24 combinations that I am testing at the moment.
It was interesting working with the 256 colour modes and finding that the colour numbers returned differed from Classic ColourTrans. The reason was largely that I hadn't been accounting for the ColourTrans weightings. With the weightings added, it was still slightly different, but good enough.

95 / 121

Testing

Testing user interfaces and platforms

- There are 3 user interfaces that you can use in the desktop:
  - WxWidgets
  - GTK
  - VNC
- None of them have any explicit tests.
- There are 2 applications that are produced:
  - Windows application
  - macOS application
- Neither have any explicit tests.
- Both work for me, and I've had some success with friends testing them. Eventually.

Testing

Testing user interfaces and platforms

Testing is great, but it would be really good to be able to test the user interfaces. I've got 3 interfaces - WxWidgets, GTK and VNC.
And I have no testing for them at all. I'd like to test them but user interface testing is more tricky. I've got a simple environment set up under Linux which allows me to test applications in a desktop environment, but getting that into the testing in a structured way will need more thought.
Similarly, I'm producing a couple of applications for different platforms, but they aren't tested either. There's a Windows application, and a macOS application - both of which contain a bundled Python, all the packages and the sources. But they're never tested.
And that's a problem, because it turned out that both those applications had been broken for about a year. Fortunately they're now fixed, but it would be good to add some testing for them soon!

Testing

How much testing is there?

- There are now 1592 tests (up from 1022 last year).
- Code coverage is 67.19% (up from 65.8% last year).
- There are 82450 lines of python (up from 57997).
- There's a bunch more statistics up on the pyromaniac.riscos.online site.

Testing

How much testing is there?

So having just said how there's some big holes in the testing, how much do I actually have?

- There are nearly 1600 tests now, up from just over 1000 last year.
- Code coverage - the measure of how much code is actually run by those tests - has increased to 67% from 66%.
- Given that there are about 25,000 lines more code this year - an increase of 40%, I'm quite pleased that the coverage has been held and increased.

Some of that code coverage difference is in code that's not exercised in the user interface and platform code.
And some is due to the large amounts of debug code that's conditional and never hit in normal tests.
If you like statistics, you'll find more information on the resource site.

Testing

Trace features (1)

- Better disassembly of some instructions.
- Can disassemble FPA instructions.
- More information about register and constant values in live debug:

```
3852384: LDR     r1, [r10, #&dc]            ; R10 = &05405738
3852388: TST     r1, #&1000000              ; R1 = &00000852, #16777216 = bit 24
```

- Decoding of dispatch tables and region names:

```
3841f78: CMP     r11, #&3e                  ; R11 = &00000032, #62  = '>'
3841f7c: ADDLO   lr, r11, #&2f              ; R11 = &00000032
3841f80: MOVLO   r11, r1                    ; R1 = &00000000
3841f84: ADDLO   pc, pc, lr, LSL #2         ; Table dispatch index #97
3842110: B       &0384331C                  ; -> Function: SWIWimp_ReadSysInfo
384331c: {DA 'ROM', module 'WindowManager': Function SWIWimp_ReadSysInfo}
```

Testing

Trace features (1)

That's enough about how I test the OS... let's talk about its test features have improved.
The disassembly produced by the tracing system has been improved in a lot of little ways. The code generated is more like RISC OS disassembly in some places, with more special cases to turn the more modern assembly format to things that RISC OS users are familiar with.
This includes the FPA floating point instructions - there's a bunch of dedicated code that handles decoding those instructions into the mnemonics we expect. This was added largely because some floating point code turned up in a module I was debugging and I wanted to understand what it did.
There's also some better logic in place for decoding the register values when the disassembly is using live registers. This means that some values from the registers will be shown as numbers where they weren't before, and some places will include bit numbers. This can make it easier to follow the code's execution.
Together with the function name decoding as we enter new regions of code, and the special decoding of a dispatch table entry, it's a lot easier to see what code is doing.

Testing

Trace features (2)

- Improved MSR constant decoding:

```
3841e3c: MSRVS   apsr_nzcvq, #&20000000    ; #------ --- -- -- qvCzn
```

- `OS_WriteS` now reports the string that follows the instruction.
- A few more SWI interfaces report 'misuse' warnings.
- LegacyBBC can now report when its old interfaces are triggered.

Testing

Trace features (2)

The status register can be decoded in a number of places, which means that it's easier to see what flags will be selected. The example shows the regular condition flags being updated explicitly, and only the flags in the PSR which are changed are shown.
There are a few more SWIs that now use the trace system's 'warning' to report cases where the interface has been used within invalid values.
This includes the LegacyBBC module which can now report when you use calls that it provides. This could help replace code that uses those ancient interfaces.

Testing

Trace features (3)

- Locations in the trace now report region names.

```
Locations:
    r5  -> [&07065abc, &00000000, &00000000, &00000010] in DA 'Module area', module
'ColourPicker%Base' private word pointer
    r6  -> [&00000000, &0680141d, &06801426, &00000010] in DA 'Module area', module
'ColourPicker%Base' workspace
    r8  -> [&06801378, &068013c4, &00000000, &00000000] in DA 'Module area', module
'ColourPicker%Base' workspace
    r9  -> Function: resource_templates_free in DA 'Module area', module 'ColourPicker'
    r10 -> [&00000000, &00000000, &00000000, &00000000] in DA 'SVC Stack'
    r11 -> [&070528e4, &00000001, &07065aac, &00000000] in DA 'SVC Stack'
    pc is DA 'Module area', module 'ColourPicker': Function model_register+&a0
    lr is DA 'Module area', module 'ColourPicker': Function rgb_initialise+&104
```

Testing

Trace features (3)

When a watchpoint or tracepoint or SWI trap is triggered, the state of the registers are listed as locations if the memory exists. This includes referencing the function name if one is known. If the register refers to the private word of a module, that's also described.

Testing

Trace features (4)

- SWI traps allow a trace dump to be generated when a SWI is called.
- They now allow automatic transitions of the trace system:
  - `report`: Just reports the state as the SWI is entered and exited.
  - `trace`: Turns on code tracing whilst the SWI is executing.
  - `traceon`: Turns on code tracing when the SWI is entered.
  - `traceoff`: Turns off code tracing when the SWI is entered.
- This aids debugging if you know a SWI is called near where you're interested in.

Testing

Trace features (4)

There are new options to the SWI traps which allow the trace state to be changed dynamically when the SWIs are executed. This means that if you know that a particular SWI is operating badly you can turn on the tracing just whilst it's executing.
Or if you know that a failure happens after a particular SWI call you can turn on the tracing from that point.
In the future, this might be extended to have other types of triggers, but I haven't needed them yet.

Testing

UI Debug

- Command line debug options can be used to set the debug from the start.
- Inside RISC OS you can use `*PyromaniacDebug <options>` to change that.
- But sometimes it's easier to select things from the user interface... so...
- UI has a debug menu - which I'll show in a moment - so that you can change the debug information live.

Testing

UI Debug

Sometimes when running the system you want to interact with it out of band of the actual system. By that I mean the system's running and you just want to see what it's doing. But you can't get to the command line to turn on the debug.
You can sometimes restart the system, enable debug on the command line, and get back to where you were, but that's not a nice option even when you can do that.
So in the user interface there's an extra menu item to control which debug options are enabled whilst the system is running. This makes it a lot easier to check what's going on when you get to the right place.
When we come to the system demo, we'll see that in use.

6. Miscellaneous bits

6. Miscellaneous bits

There are a few bits and bobs that are important, so I'll talk about them and then we'll move on to the system demo.

Miscellaneous bits

Sound system (1)

- Original testing was using BBC program from *BBC Micro Music Masterclass*.
- This was a simple rendition of *Hall Of The Mountain King*.

```
 10 REM
 20 REM *** HALL ***
 30 REM
 40 REPEAT
 50 READ P
 60 IF P=0 THEN END
 70 SOUND 1,-10,P,5
 80 UNTIL FALSE
 90
100 DATA 61,69,73,81,89,73,89,89,85,69,85,85,81,65,81,81,61,69,
73,81,89,73,89,109,101,89,73,89,101,101,101,101,0
```

Miscellaneous bits

Sound system (1)

When I implemented the sound system, which was present in the version last year, I'd wired the SoundChannel system to the host's MIDI. And I tested it mostly with the simplest of things - sample code from old books on the BBC sound system.
Hall Of The Mountain King is probably relatively familiar to people, although maybe not in its BASIC form here.
That was the way I tested much of the original sound system. However, this meant that only some of the `SOUND` calls were actually being tested.

Miscellaneous bits

Sound system (2)

Converting RISC OS pitches to BBC pitches:

```
if pitch >= 0x100 and pitch <= 0x7FFF:
    # In BBC sound, 53 is middle C, with 4 steps per semitone;
    #    48 steps per octave
    # In RISC OS sound, &4000 is middle C, with &1000/12 steps per semitone;
    #    &1000 steps per octave
    riscos_pitch = pitch
    pitch = riscos_pitch / 4096.0 * 48
    pitch = int(pitch - 139 + 0.5)
    if self.debug_soundchannels:
        print("Converted RISC OS pitch &{:x} to BBC pitch {}".format(riscos_pitch,
pitch))
```


Miscellaneous bits

Sound system (2)

In particular, Arthur had introduced a different form of pitch specification which allowed much more granular setting of the frequency. This wasn't supported at all - and some of the newer sound tests that I'd been using needed to use this format.
So I had to do a bit of maths to convert the pitch down from the new pitch format to the BBC format - because really that's all we support internally at the moment.
There were also a bunch of bug fixes for different interfaces - it hadn't been possible to select channel 8 at one point, and some interfaces had off-by-one errors so would select the wrong instruments. It produced some very ugly sounds at times.

Miscellaneous bits

Sound system (3)

- For completeness I wanted to get the SoundScheduler working.
- This lets you play notes at later times using a beat schedule.
- Or you can call any SWI.
- Speeds were all wrong... but that fixed itself by not debugging it so much.
- SoundDMA is still on a branch and hasn't been updated this year.
- The sound system is complicated, but the implementation here works well enough.

Miscellaneous bits

Sound system (3)

The SoundScheduler is the next layer up the stack from SoundChannels. Its not actually used that often, but it provides an arbitrary scheduler for SWI calls. This actually matters because some applications do use it to play music.
The SoundScheduler, under RISC OS Pyromaniac, is semi-faked. It doesn't use a clock from the sound system, but instead uses the regular monotonic timer. Because of how the system is configured this is approximately the same thing.
When it was first being tested, the scheduler was playing really slowly and I spent some time trying to work out where the problems were. In the end it turned out that I'd left an excessive level of debug enabled - turn that off and it worked pretty well.
There's also an implementation of SoundDMA on a branch, but I've not touched it since last year. It has a number of lazy assumptions that really need fixing.
The sound system is a complex and badly documented beast, but I think the implementation is pretty close to the original, at least in the main functions.

Miscellaneous bits

Flashing cursor (1)

- Text cursor had only been implemented as a stub that didn't actually flash.
- Wanted to test that `OS_RemoveCursors` and `OS_RestoreCursors` were used properly.

All the code that needed to handle cursors has a context handler around them. The Font_Paint code looks like this:

```
with self.ro.kernel.graphics.vducursor.disable():
    self.context.paint(fc.sequence, spacing)
```


Miscellaneous bits

Flashing cursor (1)

In the middle of the year I decided that I'd make the text cursor - the flashing line where you're going to input text - actually appear. I'd had stub code that tracked the state of whether it should be on the screen or not, but hadn't implemented the actual rendering of the cursor. I'd been somewhat reluctant to add in the flashing cursor because it wastes processing time and serves no diagnostic and testing function.
Almost.
With all the other graphics changes that had been added, I'd been trying to include the necessary calls around them to ensure that if the cursor were there it would be removed. As you can see, the way it's invoked makes it very simple to be sure that the cursors are handled properly.
But there was no way easy way to know whether that *was* right or not except by actually having a real cursor.
Adding the cursor was a little fiddly because the way it's usually implemented is that you find the scanline in the screen memory, read a character's worth of bytes, invert them, and write them back.
However, in the implementation of the `*ScreenSave` command, I'd added a means by which a horizontal line can be read from the screen buffer. It can return each pixel from the screen as a colour number, just like you'd have on RISC OS Classic. Creating a similar function to write a horizontal line of pixels to the screen was actually quite simple. It's not quite the same routine backwards but it's pretty close.

Miscellaneous bits

Flashing cursor (2)

The actual body of the cursor code looks like this:

```
(x0, y0, x1, y1) = self.ro.kernel.graphics.vdu4_coords(tx, ty)
y0 = y1 - self.ro.kernel.vdu.cursor_endline
y1 = y1 - self.ro.kernel.vdu.cursor_startline

fg = self.ro.kernel.vdu.fg
fg = 255 if self.ro.kernel.vdu.ncolour == 63 else self.ro.kernel.vdu.ncolour
for y in range(y0, y1 + 1):
    hline = self.ro.kernel.graphics.read_hline_internal(x0, x1, y)
    hline = [col ^ fg if col is not None else col for col in hline]
    hline = self.ro.kernel.graphics.write_hline_internal(x0, x1, y, hline)

# Notification that the bank was updated, so that the frame is rendered
self.ro.kernel.graphics.display_bank_updated()
```


Miscellaneous bits

Flashing cursor (2)

The actual body of the cursor code looks like this.
I hope you can tell that - in comparison to how this would have been implemented in assembler - it's relatively simple.

- We get the bounding box of the cursor's character.
- We update this to reflect the cursor shape - how many lines it covers.
- We work out the colour.
- Then we read a line from each, invert, and store back.
- Finally, we ensure that the display updates.

My reluctance to add the cursor as a default was still there, and I settled on some simple logic. If you're using one of UI implementations, the cursor is on by default, and in other cases it be off. But you can explicitly override that, if you really want it on, even if you can't see it.
Once the flashing cursor was added the system really began to feel like a real OS. That was genuinely surprising to me. Obviously I'd had a cursor when I was using it in the terminal, but seeing the cursor flashing at a Star prompt made it somehow more like a real system. I know that not much has changed much by adding that, but it felt much more like a real RISC OS after that point.

Miscellaneous bits

Vectoring

- `WrchV`, which vectors all the VDU output (since BBC days) wasn't implemented.
  - Required for some `Wimp_CommandWindow` to work properly.
  - But it slows things down to do this for every character.
  - Now implemented, but bypassed if there are no claimants.
- `DrawV` is used by the `Draw` module to augment its interface.
- `FontV` had been intended to be vectored, but was disabled by Acorn.
  - Pyromaniac allows it to be enabled through configuration.

Miscellaneous bits

Vectoring

One of the earliest things that was implemented in Pyromaniac - probably *the* earliest - was the output stream which was used by the `OS_Write*` SWIs. However, this just went straight into the VDU system. In RISC OS Classic, characters pass through `WrchV` - the write character vector.
This allowed for some efficiencies for Pyromaniac - strings could be written out in one go, rather than being broken up. However, this meant that some things didn't work as you might expect. You've probably not encountered it, but the `Wimp_CommandWindow` traps output so that it can insert the drawing of the command window when your task starts writing output.
And on RISC OS Select, the BootLog module would capture all the output before the desktop was entered so that errors could be diagnosed more easily.
The problem is that breaking up the output so that it goes to the vector, one character at a time, would make everything a lot slower. So as with the flashing cursor, I made this a configurable option. Two options that you have are to always use the vector, or to never use it - but the default is to use the vector 'as required'. If someone has claimed the vector, it will split everything up and pass all the characters to the vector one by one. But nobody has claimed the vector, the fast internal path using whole strings will be used.
This made the command window work properly, and the BootLog recorded things just like you'd expect.
There are two other interfaces that weren't using the vectors which could.
The Draw module was expected to be passed through `DrawV`. This was pretty easy to do, but it looked like there was no way for the vector to report an error, so I've changed the interface slightly to allow unclaimed SWIs to report errors.
Similarly, the FontManager module was meant to use `FontV`, but (for some reason) this was disabled in '92, and had never been added back again as far as I can see. Pyromaniac can be configured to support using `FontV` if you want, with an identical interface to `DrawV`.

Miscellaneous bits

Hourglass

- The `riscos-hourglass-maker` repository has been updated.
- Support (on a branch) added for percentage digits, or a progress bar.
- The 'cog' hourglass within Pyromaniac uses these.

Miscellaneous bits

Hourglass

The hourglass isn't very exciting - last year, I created the scripts that build different hourglass modules and published them up on GitHub. They look ok, but nothing special, but I do like the little spinning cog. You should see it pop up now and then whilst I'm demonstrating.
I created an experimental branch for the hourglass maker that allows percentages to be displayed, either as digits below the hourglass, or as a little bar indicator. Or with both. It's up on GitHub if anyone wants it.
The point of creating the percentages was that the data generated by the maker can be dropped into Pyromaniac, and with some small tweaks to the Python code, we have percentages in Pyromaniac as well.
We'll see the hourglass percentages in the demo shortly.

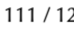
Miscellaneous bits
OS_Plot changes

Actions:

- `OS_Plot` only supported the simple 'set' operation.
- `EOR` is used quite often to animate shapes.
- Cairo provides a similar `DIFFERENCE` operator.
- Not quite right, but sufficient to work most of the time.

Dotted lines:

- `OS_Plot` can be used to draw lines using a dot pattern.
- The pattern is configured by `VDU 23, 6`, the length by `OS_Byte 163`.
- Cairo's dot pattern is defined by on-off lengths, so these needed converting.



Miscellaneous bits
OS_Plot changes

Originally when I created the Cairo graphics implementation I didn't think it would be possible to do all the `OS_Plot` actions. Fortunately, most of the actions are pretty useless - so they would rarely be needed.
But the 'Exclusive OR' operation is commonly used to provide simple animation effects by plotting and unplotting shapes.
Whilst it won't work quite the same way in paletted modes, the effect can be simulated using the `DIFFERENCE` operator to draw shapes. In paletted modes it won't do the right thing - the colours drawn will not match the exclusive OR'd colour number. But it works well enough some of the time.
`OS_Plot` can also be used to draw lines with a pattern of dots. It's configured by `VDU 23, 6` and `OS_Byte 163`, and it also allows a continuation of the pattern, so that you can draw a pattern which continues around corners.
Cairo can be configured with a dot pattern to use, but it is a bit fiddly - converting from the bit pattern used by RISC OS to a similar pattern used by Cairo was surprisingly difficult.
In the end it was just 66 lines, but it took quite a few hours to get it right. And it still doesn't support those continuations.
There will be more work on the `OS_Plot` interfaces but there are fewer interfaces that *need* to work now.

Miscellaneous bits

Mouse input

- Mouse clicks worked, but...
- ... mouse double clicks never happened.
- Maybe the time wasn't reported properly?
- The mouse timestamp was wrong.
- ... but that wasn't the problem.
- The mouse buffer wasn't implemented, so maybe that was the reason.
- So I implemented the mouse buffer...

Miscellaneous bits

Mouse input

At one point I had some problems trying to get mouse input with the UI. The mouse clicks worked fine usually. But if I double clicked I wouldn't get any events being delivered. I decided that it was probably due to the system running a lot slower than it was expecting. If the system took longer to process the mouse click than the double click delay, then that would mean that we missed out on detecting the second click.
So went to add the code to print out the time that each event was received and when it was delivered... and found that the code wasn't quite right. Instead of the mouse event time being a value from the monotonic timer, it had been given as the unix epoch time times 100.
Whilst that meant that it still counted up in centiseconds, it wasn't comparable to the value from `OS_ReadMonotonicTime`, so could mean that the click events weren't detected properly. I changed that code so that reading the time returned correct values for the timestamp.
Still double clicks didn't work.
The mouse has a few ways of being read. Some of them go through the mouse buffer. Some of them don't. I'd not used the mouse buffer at all in my implementation, and everything was being read directly. Maybe because the clicks weren't buffered, the problem with the implementation still being slow was still reasonable.
So I added some code to debug what was being delivered and started implementing the mouse buffer - it's pretty simple as it's just a matter of pushing some data into a queue on one side and pulling it out at the other.

Miscellaneous bits

Mouse input

- Mouse clicks worked, but...
- ... mouse double clicks never happened.
- Maybe the time wasn't reported properly?
- The mouse timestamp was wrong.
- ... but that wasn't the problem.
- The mouse buffer wasn't implemented, so maybe that was the reason.
- So I implemented the mouse buffer...
- ... but then noticed that the events were never delivered to RISC OS.
- The WxWidgets interface weren't being delivered, because they were never requested!

Miscellaneous bits

Mouse input

However... I noticed in testing it that when I double clicked there *weren't* any events delivered from the interface for the second click.
If there are no events being delivered, there's no way RISC OS could report them. A little more debugging and it turned out that the WxWidgets implementation handles double clicks separately, and I hadn't requested they be delivered.
A few changes to the event registration inside the window handler and double clicks now worked! I finished the mouse buffer code, and so we have a nice mouse buffer and the events are delivered properly now.

Miscellaneous bits

New modules

- Zipper module
- TimerMod
- CryptRandom
- CDFSSoftPyromaniac driver
- Squash

Miscellaneous bits

New modules

A few modules were implemented and finished...

- Zipper module
  - Allows zipping and unzipping of regular Zip archives and those with RISC OS filetypes extensions. Together with the MiniZip and MiniUnZip command line tools, archives can be handled easily.
  - As an aside the `python-zipinfo-riscos` library has been updated to make it easier to create RISC OS style archives from non-RISC OS systems.
- TimerMod
  - Provides a high resolution timer for applications.
- CryptRandom
  - Provides cryptographically random numbers. But the implementation here can provide less random numbers, because that's useful for testing. Why create it at all? Because the new Internet stack uses it, so thought it useful to have.
- CDFSSoftPyromaniac driver
  - Allows you to mount ISO images from the host system. Of course you still need to load CDFSDriver, CDFS and FileSwitch to use it, but it's nice to have working.
- Squash
  - One of the modules needs to decompress a sprite with Squash.
  - Using the Classic Squash module took around 40 seconds to decompress it - and that sucks.
  - So I made a new native module which does it in under a second instead.

System demo

System demo

I suspect this will very quickly become a very boring demo as what you're seeing will be very familiar. However, I'll be explaining what things are and why they matter as we go, so hopefully this will make it more interesting.

For this demo, please feel free to jump in and ask questions about the things that I'm demonstrating. I don't mind if it's a question about what I'm demonstrating or how it works under the hood - there is a lot going on in the system so if I'm glossing over something, speak up!

<Start running the demo with `pyrodev --config-file wimp.pyro 2> trace.txt`. Ensure the terminal window is on the left of the screen and the Pyromaniac window is on the right. Ensure that desktop icons are hidden.>

<Share the new window>

This is the boot menu. It was kinda working last year, but now it's not using any special changes to work correctly. There's a sprite for the cog, and the graphics operations all work properly. If I cursor down the page it scrolls properly. We can exit to the command line just like you might expect.

<select the 'exit to command line'>

The only reason I've done that is to show that the flashing cursor works. There's not a lot else to say about that.

<`*Desktop`>

The desktop banner here is one of the examples of a squashed sprite that's decompressed. Using the native decompression code it's reasonably fast.

And we have a desktop. There are only a few tasks running at the moment - we can see Task Manager, Display Manager and the Resource filer on the iconbar. And there's Pinboard running in the background with a nice gradient.

<Open TaskManager menu>

The TaskManager menu is built using `MessageTrans_MakeMenus`, so you can see this is working properly. You

8. Conclusion

8. Conclusion

Let's wrap this up with a conclusion...

Conclusion

What did I get done? (1)

- Graphics system improvements. Sprites, ColourTrans, deep modes, VNC server, hourglass.
- Filesystem improvements. More commands supported, FSControl, GBPB improvements, encoding improvements.
- Input improvements. Keyboard scans and mouse buffer handling.
- Debug improvements. Better trace reports, more info in disassembly. FPA instructions.
- Sound system improvements. SoundChannels. SoundScheduler.
- New modules. Zipper, TimerMod, Squash, CryptRandom, CDFSSoftPyromaniac.
- Many fixes across the system.
- PRM-in-XML documentation improvements.

Conclusion

What did I get done? (1)

So what did I get done over the last year?
Lots of graphics system improvements. There were a lot of little fixes to the graphics and VDU system. The new ColourTrans and Sprite handling is now pretty reliable. Deep modes are usable, and 256 colour modes are now reliable. The new VNC implementation means that more doors are opened for using the system in the cloud. The hourglass finally has a percentage.
Lots of filesystem improvements for how many FSControl and GBPB operations work, and commands like `*Wipe` are now implemented. The encoding used by filenames is now handled much better.
Input now works much better, with keyboard scans being properly supported and the mouse buffer being used properly.
A bunch of debug features were added to make it easier to trace when certain calls were made, and to improve the disassembly and reports. FPA instructions are now disassembled.
Improved SoundChannels, and new SoundScheduler module for timed playback. A bunch of new modules added to round out the use of the system a little more.
Many fixes to parts of the system, including problems with creating module instances, filing system commands, and application space.
And improvements to the PRM-in-XML documentation system to make it much more useful.
There were a bunch of other things that I have worked on which weren't discussed here - a server for the documentation project, the FanController system, URL fetcher, a project called PRIDE, some experiments with different architectures, and a few build tool changes. None of those are at a point they're ready to be shown yet, but maybe next year they will be.

Conclusion

What did I get done? (2)

- Shell server updated semi-regularly.
- RISC OS build server back end updated at the same time.
- The information site has been updated: https://pyromaniac.riscos.online/
- Lots of information about what's supported in the `Docs- Features` documentation.
- The full changelog in `Docs- Change Log` summarises many other things I couldn't cover here.

Conclusion

What did I get done? (2)

The shell server was updated at the beginning (and had been updated about every other month). Although there's been no website changes for the RISC OS build server over the last year, the back end has been updated equally regularly, taking that month's release.
I've updated the information site at https://pyromaniac.riscos.online/ with more resources from the collection of images that I created over the last year, and updated documentation.
I would recommend people take a look at the `FEATURES` documentation on the site to see what is and isn't supported. And take a look at the `CHANGELOG` for the last year, too.
Whilst you might expect that I'm pleased with the work that I've done on the operating system, I'm also rather proud of the fact that it's pretty well documented and the changes are well recorded.

Conclusion

Is it still fun?

- Generally still fun!
- Collaborative working with Alan on the documentation system has been great.
- Taking the opportunity to talk about testing on RISC OS was a nice break in the middle of the year, and maybe I should do that more often.
- Sometimes investigating problems goes nowhere, but makes for interesting experiments.
- This presentation has been stressful to prepare - packing a year's things into a couple of hours is tricky.

Conclusion

Is it still fun?

I guess for most people writing notes about what they've done and keeping a decent change log isn't that fun. But I think it's a general part of what I enjoy. Not only does it show me what I've achieved, but it's the right thing to do. So what if I've got no users and nobody's shown much interest...
So yeah, RISC OS development is still pretty fun, and I'm still enjoying the work that I'm doing.
Working with Alan Robertson to develop parts of the documentation system has been a lot of fun. It is one of those areas that collaboration really works well. I suggest something, Alan gives his opinion, we decide on the right way forward - or vice-versa. Similarly Alan reports bugs and we work together until we find what's wrong. That's been great.
My small diversion in the middle of the year to discuss software testing was largely due to comments by a RISC OS developer about poor testing being a reason for a bug they'd found - something I'd previously commented on and written about. That work was born of frustration, but realising that I could address it with some education was good - and I may not be able to meet all my desires, but I can do that. That's fun.
David Thomas reported a bizarre problem with PrivateEye crashing which I attempted to reproduce within RISC OS Pyromaniac. I had to implement a lot of things, and fix a lot more before I could even run PrivateEye. And then I still couldn't reproduce the problem. But it was a lot of fun to get to that point, and I did find a few issues in PrivateEye that wouldn't have been seen otherwise.
I expect to find other things that have similar problems and will address deficiencies in Pyromaniac as time goes on.
I've really looked forward to doing this presentation and I've spent a bit of time trying to get it right. It's made me really stressed at times, but I've had some help from my girlfriend which has made the process easier. I'll do another one next year, but I'm not sure that leaving a whole year between them is useful - there's just been so many things I've had to drop and not talk about.

Conclusion

What do I want to do next?

- Fixes for some of the known problems.
- Filesystem registration - allowing more than just the native filesystem to work.
- I have some ideas about sprite redirection.
- Multiple displays.
- Back Trace Structures.
- Port to Python 3.
- Experiment with changes to some of the internal interfaces.
- Use it to develop and test some things!

Conclusion

What do I want to do next?

I've listed a few things that I'd like to look at here. Going by previous experience, I might get about a third of those looked at, and do other things for the rest of the time.
I know there's a bunch of problems with the system that are only exposed when they come to be used in anger. Fixing those and getting some tests to check their behaviour would really help - the crash at the start of Fireworkz seemed simple enough, but behind it there were some bad assumptions that need a little restructure to fix properly, for example.
But essentially, I'll do what seems fun as I come to it.

Questions

Info site: https://pyromaniac.riscos.online/
Shell: http://shell.riscos.online/

Info site: https://pyromaniac.riscos.online/
Shell: http://shell.riscos.online/

121 / 121