Compo Clues (1)

All being well, this is the first in an 'occasional series' of Archive articles on 'Compo' . I've been prompted to write these because I use Compo quite a lot. Indeed, I have used it to create hundreds of graphics for webpages, and find it powerful, quick, and easy to use. Yet Compo seems to get very little attention in RO magazines and I suspect the many potential users have no real idea of what it can do. Hence these articles. I won't even attempt to present what I write as an 'art class' as my own talents in that area are pretty limited. Instead I want to convey information on the scope and ease-of-use of the application, and show how easy the creation of graphics becomes when using it.

OK, the basic info is that !Composition is nominally a 'layout and graphics' package. It has been written by Rob Davison, and sold by Clares Micro Supplies. It differs from most graphics applications in two important ways. Firstly, it isn't primarily a 'painting package' that works only on bitmaps. Secondly, it has an 'object oriented' approach. Both of these factors may seem off-putting to potential new users, and give a misleading impression. This may explain why people often spend two or three times as much on buying other graphics applications without realising that Compo might well suit them very well indeed!

If you don't know about Compo, let me mention just a couple of things about it that you may find interesting.

1) Compo has had 'layers' and graded object transparency even since it first appeared.

2) It can load a wide variety of types of image file, including most bitmaps and also both DrawFiles and ArtWorks files.

3) The latest version allows the user to write and integrate their own processes using 'CompoScript'. For example, to automate creating sets of 'thumbnail' images, or creating an animation.

4) It has a number of features designed specifically to help the user create web graphics. For example, colours can be specified in the' #RRGGBB' format common on webpages. You can also quickly set the canvas background by dropping a webpage html file onto it.

5) It can now even act as a presentation app, including operating in full-screen mode, respond to mouse clicks on displayed objects to change this display, show animations, etc.

That should be enough specific points to get your attention, so on to looking at it in detail.

I'll start with a simple example to show how easy webpage graphic creation can become. I am not sure how well the following images will come out in 'Archive'. Paul's printer has been doing an excellent job of late, but just in case I will suggest he also provides images of the example on the monthly disc and on the Archive website. I suspect you will need a 32k colour display (or more) to really appreciate the quality of some of the following images. I'd love to show animations as well, but I suspect that Paul's printers would find that even harder to print! :-)

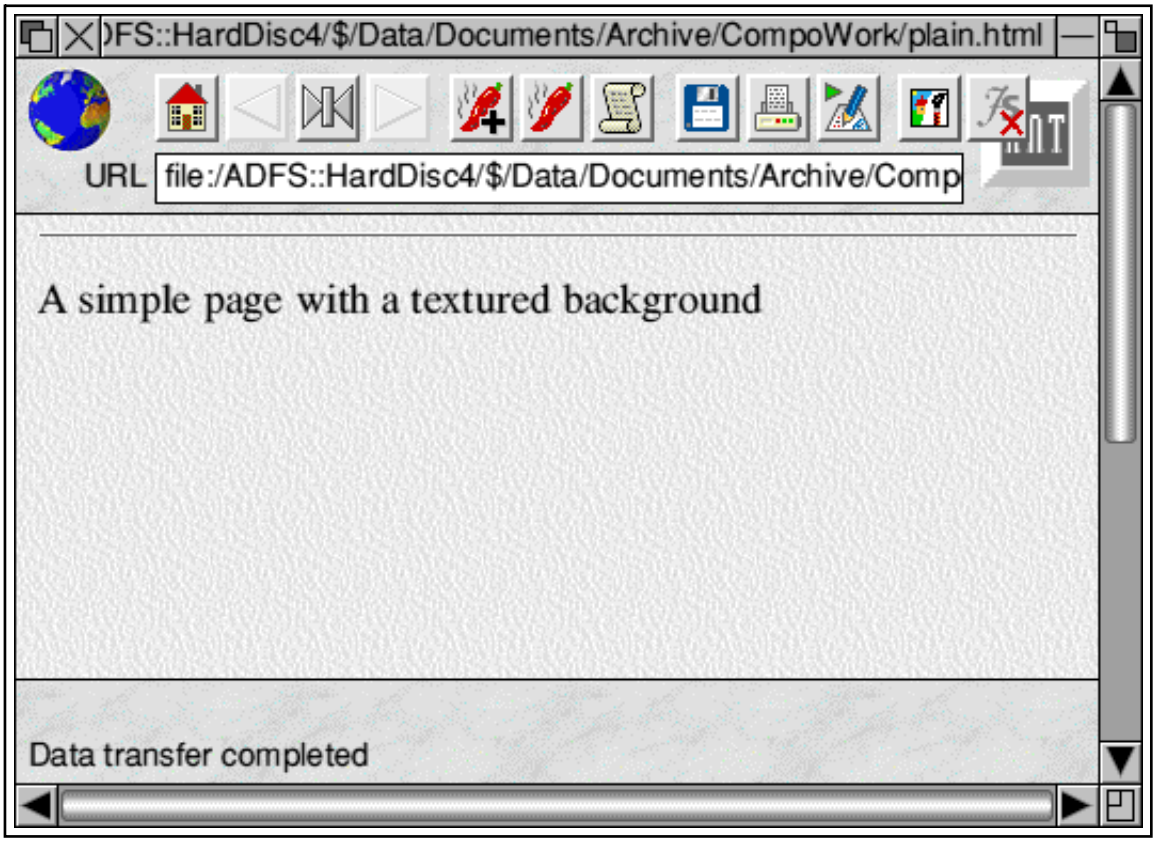

Figure 1

Let’s start with the fairly simple webpage shown below. This just displays a line of text over a ‘textured’ background. Now we can use Compo to quickly create a text display graphic of the kind often found on webpages. These frequently have a transparent ‘drop shadow’ with soft edges. This means that the textured background has to show through the shadow to some extent.

One of Compo’s useful features is that it can load information from a webpage. So if we drag the webpage file and drop it onto the Compo window (or ‘canvas’) Compo then pops up a window giving us some choices. It can then set the canvas background colour to match the webpage, or it will load the background texture. (If we wish it will also load all the images from the page onto the canvas as well!) In this case we can just choose to tell Compo to load the texture as the canvas background. As a result the Compo canvas will now look as shown below.

Figure 2

We can now click <select> on the text creation icon (the pair of ‘T’s), choose a font and colour, and create a ‘text object’. Then click <select> on the drop shadow creation icon (the ‘2001 like’ slab with a shadow) to make the object throw a shadow. This shadow will already by translucent but to finish off the effect we can now click with <adjust> on the shadow creation icon. This pops up a shadow mask editing window from which we can choose to ‘smudge’ the shadow and give it a soft (blurred) edge. We can then export the result as a GIF suitable for display on a webpage. The above process is much quicker to do than to explain and the result is shown in Figure 3.

Paul’s printer willing, if you look at the result you will see that the background texture does indeed appear visible through the text’s shadow and makes it seem more realistic and ‘3D’. The text therefore seems to float above the webpage in the desired manner.

Figure 3

This particular graphic isn't very interesting, and wouldn't win any prizes for artistic design. However it serves to show that webgraphics with translucent drop shadows are quick and easy to create using Compo.

Compo is similar in many ways to conventional 'vector' graphics package like !Draw or !Vector. Each object on the canvas can be edited and altered moved around and placed on top (or underneath) each other. Each object has a set of attributes and properties. The above shows some of these. The object has a 'shadow mask' which controls the shadow it 'throws' on objects below it. The object also has a 'transparency mask' which can be used for various effects.

Figure 4

Figure 4 shows an example. Compo allows you to easily create colour 'blocks' which you can then manipulate. I quickly created one and gave it a transparency mask which fades to complete invisibility at the horizontal edges. Placing this above (i.e. 'closer to the observer' not towards the top of the

screen) the text, you can see how the text object, and its shadow, and the background canvas texture, all show through. I then copied the block and changed the copy's mask to a uniform transparency before moving this to towards the bottom right of the other objects. You can compare the two colour blocks and see the difference between them which is just due to their differing transparency masks.

The resulting image can be exported as a GIF and will appear on a webpage as shown below. However the important point is that all these manipulations of transparency, position, etc, are fully re-editiable so I can regard this graphic just as if I were dealing with objects in !Draw. The result also displays on the webpage with the objects seeming to be translucent. This example isn't artistically very good one, but it should now be clear that, used with some taste, it becomes possible to create some quite impressive webgraphics, very easily and quickly.

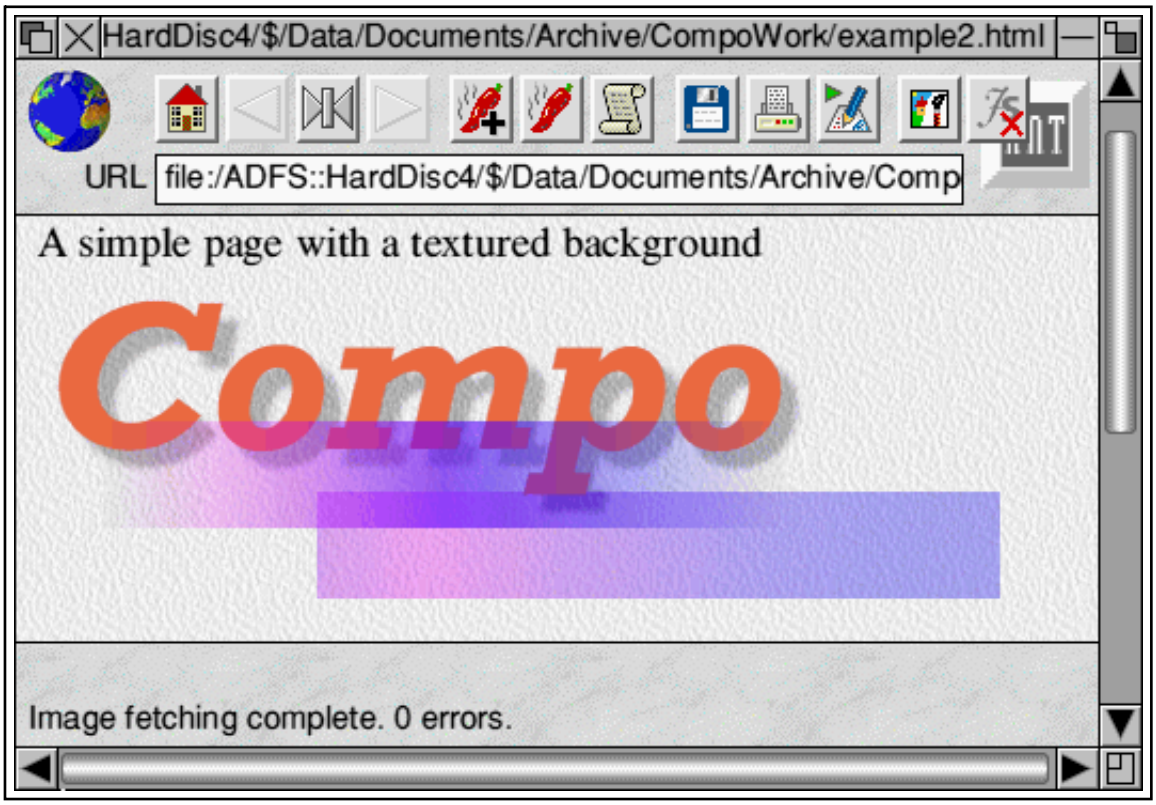

Figure 5

If you are interested in creating graphics and haven't used it, I hope that you found this first article interesting. All, being well, I'll say more and give some other examples in future issues. Until then, if you are on the web, you might like to visit a few of my webpages that show examples of graphics created using Compo. Here are a few sample addresses:

http://www.st-and.ac.uk/~www_pa/Scots_Guide/audio/cableshift/cp.html
http://www.st-and.ac.uk/~www_pa/Scots_Guide/RadCom/part15/page3.html
http://www.st-and.ac.uk/~www_pa/Scots_Guide/MMWave/QO/antennas/page1.html

In fact, if you browse around my webpages all the graphics exceopt for the equations were created using Compo. There is also some up to date info and more examples on the "Compo Clues" website at

http://www.st-and.demon.co.uk/Compo/clues.html

JCGL
1300 words
2nd Jun 2000

Compo Clues (2)

Compo has undergone a number of developments in recent months. It is therefore worthwhile for me to summarise the current situation so that users can ensure that they are using the most up-to-date version.

At present, the 'official' release version is 1.16. If you have 1.15 you can upgrade to 1.16 by visiting the Clares website. Compo has, however, had many extra features added since then. As I write this, the most up-to-date version is actually 1.18k. Technically this is a 'beta' release – although I have been using it for months without any signs of it being unstable or problematic in any way. The latest beta version can be obtained by visiting Rob Davison's website at http://compo.iconbar.com and following the link given on that page. For safety's sake, keep a backup of 1.15/1.16 when upgrading to the beta, but you should find it works just fine.

Note that you *must* have at least 1.15 to be able to successfully upgrade to 1.16 or later. New versions will probably appear in due course on Rob's website as and when he gets a chance to develop them.

This month I'd like to go through two examples of how to use Compo to create fancy web graphics. One is a simple example that shows off some of Compo's features. The other shows something more complex. For the simple example you don't need any specific version of Compo, but the complex one tend to require the new versions, and is the kind of thing that can be done more easily using a 'Scriptlet'. (Although I won't deal with that here and will leave scripts to a later article.)

The simple example is based on one of the icon buttons I created for some of the 'Compo Clues' webpages. Figure 1 shows the final result as it appears on the webpages.

Figure 1

This is a fairly typical use of icons on a page. There are three images used as a row of buttons for links to other pages. The central image (Compo Clues and the spyglass) will take the browser back to the home page. The images on each side will take the browser to an earlier or later webpage in the relevant series. On the webpage, the images are all GIFs with a transparency mask and antialiased to the page background colour. Here I'll just use the "next page" image as the example. By itself, this is shown in figure 2

Figure 2

The image actually consists of just three basic objects – although one of them does appear three times! At the top of the image there is a sprite of a grey, rectangular button with the text "next page" displayed in blue. The text itself is a "text object" added by Compo on top of a sprite of the button. To demonstrate this, I can move the text object and we get fig 3.

Figure 3

This actually reveals that the original sprite of the button had the letters "OK" on it. The text object was deliberately created with a surrounding grey border so that the text and its border covered the "OK".

The lower part of the image is a set of three arrows, and is in fact just three copies of the same sprite. I originally loaded the arrow sprite onto the canvas and the made two copies of it. I then moved them to be in an overlapping line. Finally, I gave the leftmost arrow an opacity of 40%, and the middle arrow an opacity of 68%. The arrow on the left is placed 'on top' of the middle arrow, which is in turn 'on top' of the one on the right. Hence we achieve the effect of a set of arrows that lead from left to right, visually getting stronger.

Figure 4 shows Compo with the menus and windows used for controlling the opacity of a selected object. You can see that clicking on an object to select it puts a broken red rectangle around the object to highlight it. Clicking on the canvas with the <menu> mousebutton pops up the main menu, headed "compo". You can then follow through "effects -> opacity" to pop up the opacity control window. The opacity can then be altered either by dragging the slider or by typing in a value in the range 0 - 255. As with many operations Compo tries to give you as many ways as it can to alter or choose settings. The opacity is also shown as a percentage, and if you drag the slider with the <adjust> mousebutton the actual object's opacity alters in realtime as you drag.

If you are unfamiliar with Compo, figure 4 is also useful as it shows the various windows and toolbars Compo provides. To the left of the canvas is the main toolbar with the standard tools. Along the

bottom of the canvas window is a "palette bar" which can be used to quickly set the background colour. To the right of the canvas is the new 'Scriptlets' toolbar. This is only present in recent versions of Compo and represents a major new set of features, to which users can add their own facilities. I'll be saying more about the toolbars in later articles. Here I'll just mention a couple of things about the Scriptlets toolbar before moving on to my second example.

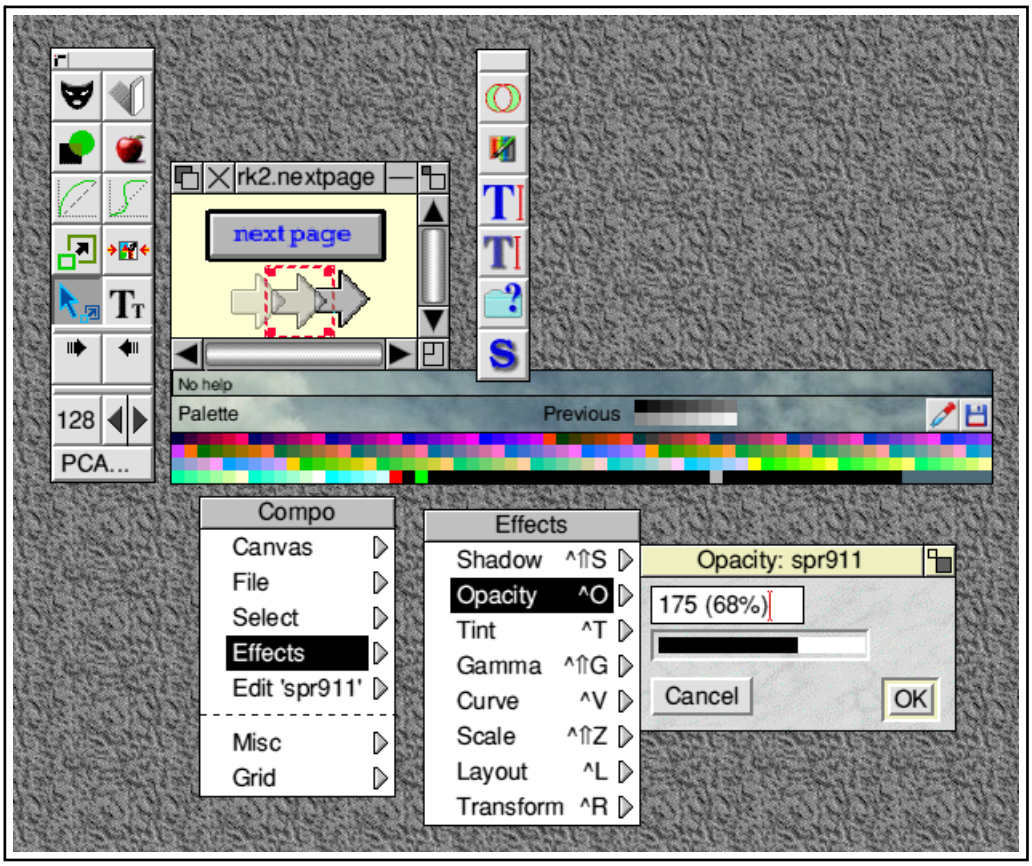

Figure 4

If you have an up-to-date version of Compo you may find that your scriptlet bar doesn't look exactly like the one shown in figure 4. In particular the top two icons (the pair of intersecting rings, and the 'paintbrush and rainbow') aren't there in your copy. That's because these are Scriptlets that I have added. You can obtain these and add them to your copy if you wish, either by visiting my website or – if Paul agrees – by finding these Scriptlets on the 'Archive' disc.

The top icon (the rings) is a Scriptlet called 'framer'. It takes an image (object) on the canvas and allows you to trim it into a complex shape and then add a 'frame' by using drawfiles. The Scriptlet does this automatically, but my second example this month takes you though how to do this with Compo by hand. The result makes a change from always having boring old rectangular images on webpages or documents.

Ok lets start with an example image. Figure 5 shows a bitmap picture. (If you are interested, it is a photo of Glen Devon taken a few years ago.) Now create or choose a suitable drawfile shape for the masking. For the whole process we actually require two similar drawfile shapes which I have included as figures 6a and 6b. Note that 6a is a filled shape with a fill colour of 'white', but 6b has a fill colour of 'none' – i.e. it is an open shape. I actually created 6a by quickly dropping the picture I want to frame into !Vector and scribbling a closed curved path over it, then removing the bitmap and saving

the resulting drawfile shape after filling it with white. I then changed the fill and line colour and saved a second file – shown as 6b – to use as the frame.

Figure 5

Although I used !Vector to create these shapes, !Draw would obviously work just as well. I just prefer !Vector.

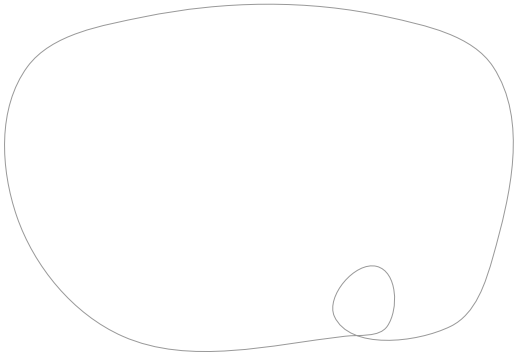

Figure 6a Drawfile shape to be used as the mask.

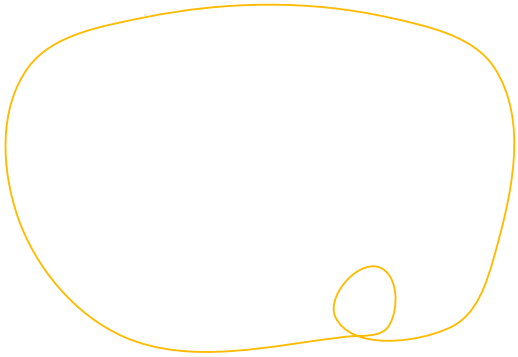

Figure 6b The same basic shape but with no fill and a different line colour and thickness.

These shapes are obviously fairly silly for most realistic purposes, but they do show that you can use quite complex shapes for the masking and framing if you wish.

Drop the bitmap image onto the canvas and click <select> on it to make sure it is ‘selected’. Click the <select> mouse button on the masking icon in Compo’s main toolbar. (This is the top-left button and, not surprisingly, looks like a theatrical mask.) This pops up the mask control pane just below the canvas as displaying in figure 7

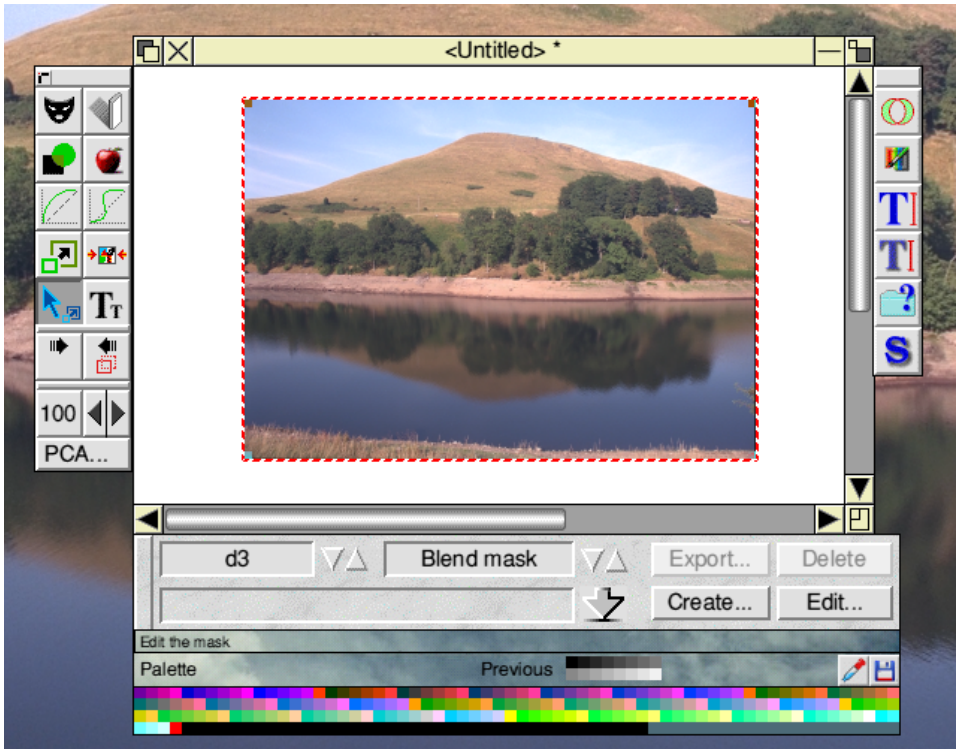

Figure 7

Just to the left of the ‘create’ button in the mask control pane there is a 3D arrow pointing downwards, with its tip touching a horizontal line. This is a ‘drop here’ target onto which you can load a file to be used as a mask. I can now drag and drop my ‘mask’ drawfile onto this. This causes Compo to import the drawfile and apply it as a blend mask to the selected object. The result is shown in figure 8.

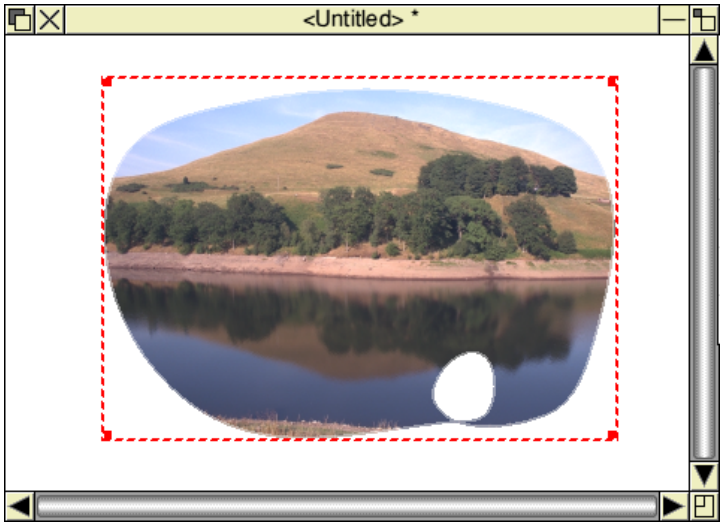

Figure 8

As you should be able to see, the image has now been clipped into the drawfile pattern. In fact, the drawfile colours are converted into a 256-level greyscale and the masking is applied as a pattern of opacity, so the boarder line (which was grey) produced a translucent ‘edge’ which might be useful in some cases. Note that the inner loop in the pattern produced a similarly shaped ‘hole’ in the image. This shows that quite complex effects can be achieved very quickly by applying a drawfile as a mask

for an image.

Next I dropped a frame drawfile shape onto the canvas on top of the object. This had to be moved about a bit to align it correctly, but once this was done the result was as shown in figure 9.

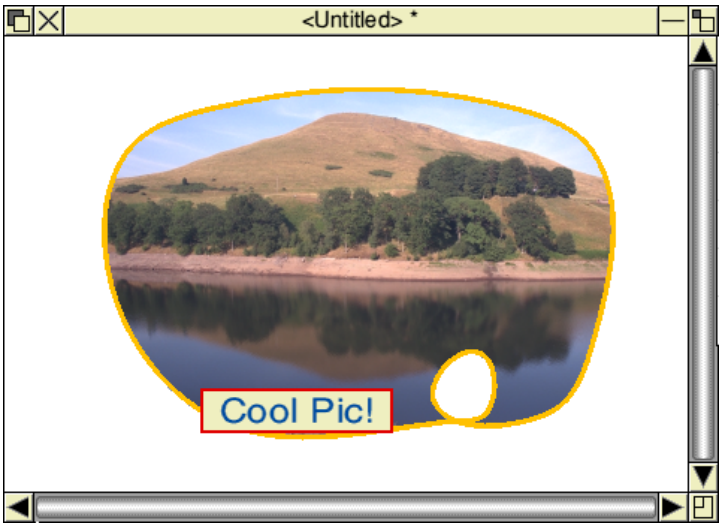

Figure 9

Note that I have cheated a bit here as the frame shape I used had a thicker line than 6b, and some added text in a box. This is just to show that the frame pattern can – if you wish - differ from the mask. You can even have some colourfilled parts and then make the frame translucent to give a sort of 'stained glass window' effect if you wish. If I'd done that figure 9 would have showed the water behind the translucent "Cool Pic!" text and its box.

One of the Scriptlets on my Compo Clues website automates the above process to make it much easier to perform. Using the Scriptlet the process of masking and framing and adding a shadow can be done in a couple of mouseclicks, and the scaling and alignment is done for you. However by going through the process "by hand" here I hope I have shown just how powerful the image manipulation can be. Each object on a Compo canvas actually has a number of assigned masks which allow a number of attributes to be altered so some quite interesting results can be achieved fairly easily.

Ok, that's it for this time. I wonder how Paul's printer will get on with these new images...

1760 words
23rd July 2000
Jim Lesurf

Compo Clues (3)

One of the most common 'complaints' that potential users made in the past about Compo is that "It isn't a painting application!". Strictly speaking, in the past this was true. However with the release of Compo 1.15 and its PCA plug in applications this changed dramatically. PCA allows many extra tools to be created and used as "add ons" to manipulate objects on the Compo canvas. More recently, Compo has acquired the ability to handle CompoScript which gives yet another dimension to its

flexibility. Here I will concentrate on the use of PCA tools and just touch on CompoScript.

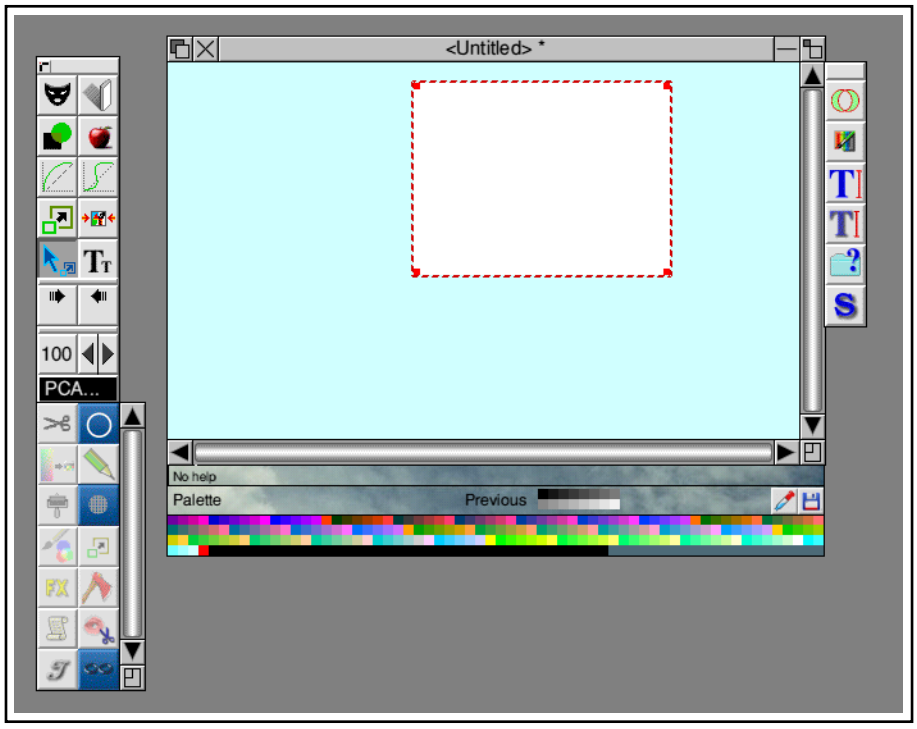

Figure 1

Figure 1 shows the Compo canvas and its associate tool ‘panes’. I have placed a plain white sprite object onto the canvas and selected it. I then clicked on the PCA button in the main toolbar (shown to the left of the main canvas). This drops down a new set of buttons for the PCA tools that have been ‘seen’ by the filer. You can distinguish these from the standard buttons as they are accompanied by a slider along their right-hand side. Note that Compo will only show the PCA toolbar if you have an object selected on the canvas as the PCA tools have to be able to ‘find’ the object they are asked to work upon. The actual set of PCA applications I have is shown in the filer window illustrated in figure 2

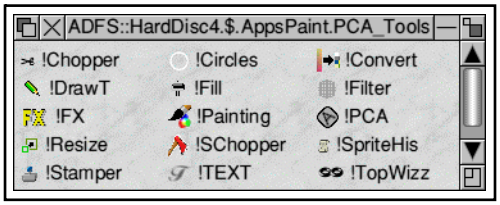

Figure 2

Many of these are provided with Compo 1.15. Others can be obtained from Rob Davison’s website. Most only work as plug-in tools with an application like Compo which supports the PCA protocol, but a a few (for example !SChopper) can be used ‘stand alone’. I can’t cover them all in detail here so I’ll just give a quick idea of what a few of them can do. Lets start with the most obvious ones – !Painting and !DrawT.

!Painting does pretty much what you would expect, it allows you to paint onto a selected object. Various painbrush shapes can be chosen (you can add to these and define your own if you wish). The painting can also be ‘sprayed on’ and have a chosen edge/central transparency. You can also paint on colour changes and various effects rather than just adding paint. Any changes can be undone up until

you decide to accept what you have done. For simplicity I chose to paint onto a plain white object and figure 3 shows the effect of just spraying some colours onto this.

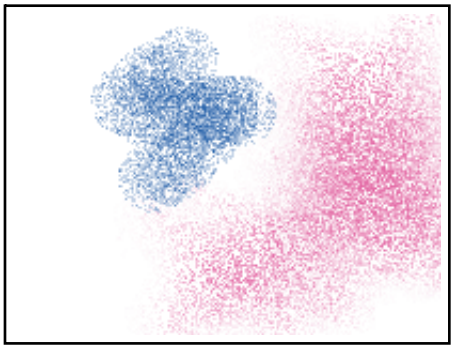

Figure 3

The result isn't meant to be in the slightest 'artistic' as I wasn't trying to paint anything, just to show that you can paint!

The !DrawT tool also allows you to 'paint' DrawFile patterns onto an object on the canvas. This is illusrated in figure 4 which shows a series of things drawn using the methods that !DrawT provides.

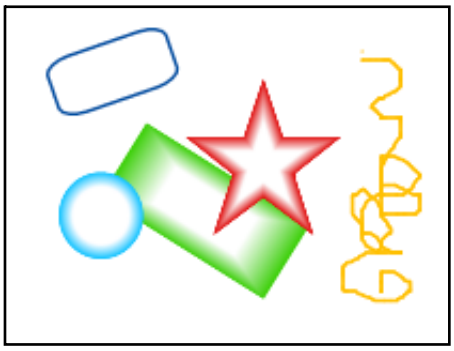

Figure 4

!DrawT provides ways to draw freehand lines and curves as well as a variety of standard types of objects which can be scaled, rotated and coloured as you choose. You can also add your own DrawFiles and get !DrawT to 'paint' them on the object.Once again for figure 4 I have used a plain white object as the basis for drawing for the sake of clarity. Many other tools work in similar ways, for example !TEXT allows you to paint text onto the object as illustrated in figure 5

Figure 5

You can also paint textured fills, and perform many other tasks using the PCA tools. For website authors one of the more interesting tools is !SChopper (older versions of this were called !Chopper). This allows you to easily split up an object and save it as an array of smaller images. The most common reason for doing this is to put the objects into a 'table' on a webpage and – for example – use JavaScript to change the section the mouse is over to provide user interactivity.

Perhaps one of the most common uses for painting applications is to alter photographic images by 'cloning' items or 'painting over' unwanted defects in the original photo. !Painting does this quite neatly, so well in fact that I need to use a rather grossly damaged image as an example to show how it works. Consider figure 6. This shows a photo of some daisies onto which some dimwit has painted some unwanted items! (In this case, of course, the dimwit is me.)

Figure 6

Having recieved such a spoilt image I'd like to try and hide the damage. I can use the !Painting tool to paint over some sections of the object with displaced copies from elsewhere. For the sake of example I can choose to make the center of one of the daisies my starting point and then paint using this. If I just wipe the paintbrush over each of the offending objects I get what is shown in figure 7.

Figure 7

To show how this works I have deliberately worked on a badly damaged original and not attempted to take any real care and tried to completely 'repair' the damage. So you can see the edges of the painted over areas and that some parts of the offending items still show at the edges. In more realistic situations the damage would have been less, I could have chosen a more appropriate softer-edged brush shape, and even used more skill. The result would then have just looked like some daisies without the unwanted items being in view. Painting out small defects in photographic images, and adding copies of things (or moving things around) is relatively easy using this powerful tool.

Another useful PCA tool is !Fill. Once again it does what you'd guess from its name. Figure 8 show the effect of using it to apply a graded fill colour to one of our earlier objects.

Figure 8

This fill was applied to the area surrounding the text so, as usual with flood fills, it isn't applied to isolated areas such as those inside the O's and e's. Figure 9 shows the result of applying a fill patter of some brickwork to the image background.

Figure 9

and figure 10 shows the result of applying a wooden effect to the letters of the top line of text and putting some light marbling into the O's of the second line.

Figure 10

All these effects, and many others, can be applied quickly and easily, and undone if you don't like the results. This makes experimenting with texture and fill effects quick and easy.

The specifications for PCA are available from the Clares website along with examples of how to write your own tools. PCA isn't just meant for working on images, nor need it be limited to Compo. It is a very powerful method for adding extra tools to any application that supports it. Alas, the level of programming skill required to produce new PCA tools for Compo is high enough that most people stick to using the ones that are already available. For most purposes this is fine as the current tools are very powerful. However to provide a rather simpler way to add flexibility Rob Davison developed CompoScript. Unlike PCA, CompoScript will only work with Compo, and you require a relatively up-to-date version to use it. The good news is that CompoScript is much easier to write and use than developing a new PCA tool.

In the last article I went over the use of a 'Scriptlet' to process an image. Scriptlets make use of a specific form of CompoScript that allows the user to add new functions they can call just as if they were a native part of Compo. However you don't need to do this if you don't want to. Instead you can just drop a Script file onto the canvas. Here I will just give a couple of very simple examples and leave considering more complex scripts to a later article.

A CompoScript file is just a plain text file which contains lines that the CompoScript interpreter can understand. To tell Compo that the file is CompoScript, the first line in the file should always be "#CompoScript". A very simple example would contain lines like the following:

```
#CompoScript
select first
scaleto 400 200
rotateto  40
end
```

To see what these do, lets start with the canvas shown in figure 11. This shows just one object (which is an imported DrawFile in this case) in Compo's canvas window. I have selected the object and given it a red dashed border just so you can distinguish the object from the surrounding canvas.

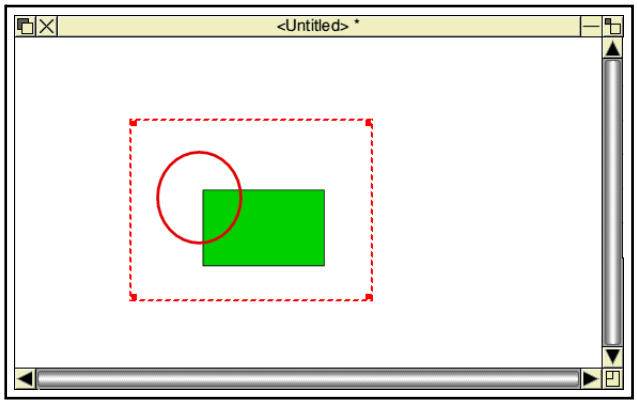

Figure 11

I can now drag the icon for the textfile quoted earlier from its filer window and just drop it on the canvas. This then process the object to produce the result shown in figure 12.

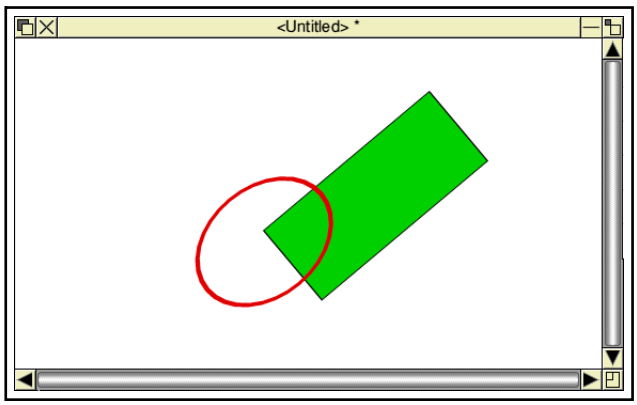

Figure 12

This shows that the size of the object has first been scaled to be 400 pixels wide and 200 high, then the result has been rotated through 40 degrees about its bottom left corner. The commands that do this in the above script should be fairly clear! The only command that needs explaining is the “select first”. This means the script selects the ‘first’ object on the canvas (usually the first loaded) to work on.

We can now process the object a little more using a second example of CompoScript shown below

```
#CompoScript
select first
let .shadow = 1
let .shadow.xoffset = 10
let .shadow.yoffset = 10
makemask SHADOW copy BLEND
processmask SHADOW Antialias Smudge
end
```

If we drop a textfile containing these lines onto the canvas we get the result shown in figure 13.

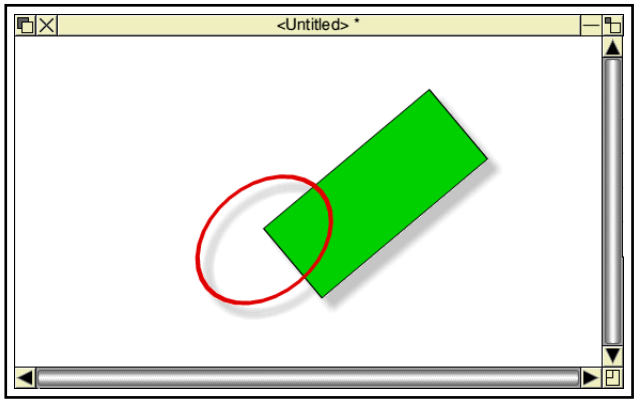

Figure 13

The object has now been given a ‘drop shadow’ with a blurred (smudged) edge. The commands to do this use the ‘let’ instruction to set switch on the display of any shadow (‘let .shadow = 1’) then set the horizontal and vertical shift from the object to the shadow. The line ‘makemask SHADOW copy

BLEND’ uses the object’s transparency mask and copies this as the starting shadow shape. The line ‘processmask SHADOW Antialias Smudge’ then blurs the shadow’s edges by applying the antiliasing smudge command.

The above are relatively trivial examples of what can be acheived using CompoScript. In principle, almost all of the actions Compo can perform may be applied in some combination using a script file. This is very useful when you want to apply the same set of operations to a series of files. It also permits Compo to automate the creation of animations, and even act as a full-screen presentation application – i.e. a bit like PowerPoint on a Windows machine, or !OHP. I’ll say more above this in later articles.

1800 words

J. C. G. Lesurf 10th Oct 2000

Compo Clues 4

In this article I want to examine the use of CompoScript. The example I have chosen is one that allows Compo to be used as a simple ‘presentation’ package to make full-screen, interactive, presentations. Although primarily a graphics editor and layout application, this example is designed to indicate the sheer range of uses to which Compo can be put. You will need an up-to-date version of Compo for this to work.

The presentation consists of a series of ‘slides’ (or ‘frames’) to be displayed. For the sake of example, I will use some simple slides which just show a few objects. I also want to show how Compo can provide interactive control using the mouse by clicking over an object. Figure 1 shows the main objects I will use for control, along with their names. Note that when Compo loads a graphic file onto its canvas it gives the resulting object the name of the file from which it was loaded.

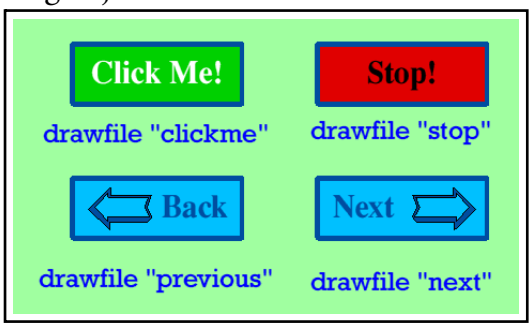

Figure 1

Let’s now look at the first part of the CompoScript used to run the presentation. This consists of the lines shown below. Note it will be followed by more lines in the file, but we will ignore them for now.

```
#CompoScript
# demo presentations
makecanvas 640 480 0
scriptoptions fullscreen "X640 Y480 C16M"
scriptoptions hidepanes
let thisframe = 1
proc load_frame
mouseover "stop" on click proc stop
mouseover "next" on click proc nextframe
mouseover "previous" on click proc previousframe
mouseover "clickme" on click proc loadtadaa
```

```
let done = 0
repeat
until done
scriptoptions showpanes
end
```

This represents the 'main' part of the CompoScript. It uses a number of commands I have not mentioned in previous articles, so let's look at them one by one. Note that in what follows I'll adopt the convention of using angled brackets "<>" to indicate a value you can choose for yourself.

"makecanvas <X> <Y> <C>" causes Compo to create a new canvas. This means that any existing canvas will be discarded, so only use this if you have no work on the canvas that is unsaved, otherwise you will lose what you were doing previously. The <X> and <Y> values set the width and height of the canvas, in pixels. The <C> value sets the background colour, in RGB format. So here the command creates a new canvas 640 pixels wide, 480 pixels high, with a black background.

"scriptoptions fullscreen <string>" makes Compo go into fullscreen mode. This means you now leave the desktop and all windows except Compo's will vanish. The string sets the screenmode chosen when this happens. Note that you should normally set the same size and depth here as for the canvas otherwise the display may not 'fit' correctly. Here I have chosen 640 by 480 as this is the largest display most RiscPCs can manage in 16million colour depths. In principle, you can choose any size and colour depth your machine can provide for both the canvas and the fullscreen settings. I have a ViewFinder card which allows me to use much larger, deeper, displays, but more of that later...

"scriptoptions hidepanes" makes the toolbars disappear. If you don't do this they may show up and cover part of the display. Since we are using all the screen we want to avoid this, so use this command to make the toolbars hide. "let thisframe = 1" then creates a variable "thisframe" and gives it the value, 1. This sets which frame of the series available will be manipulated by the next line which is, "proc load_frame".

CompoScript provides the ability to write and use procedures, just like BASIC and other languages do. This makes writing complicated scripts easier. In this case there will be a procedure called "load_frame" which does what its name implies. It loads the chosen frame and displays it. Initially this will be the first frame, selected by setting "thisframe" to 1. We'll look at how the procedure does this later on. Figure 2 shows the first frame of my demonstration presentation as it should now appear on screen.

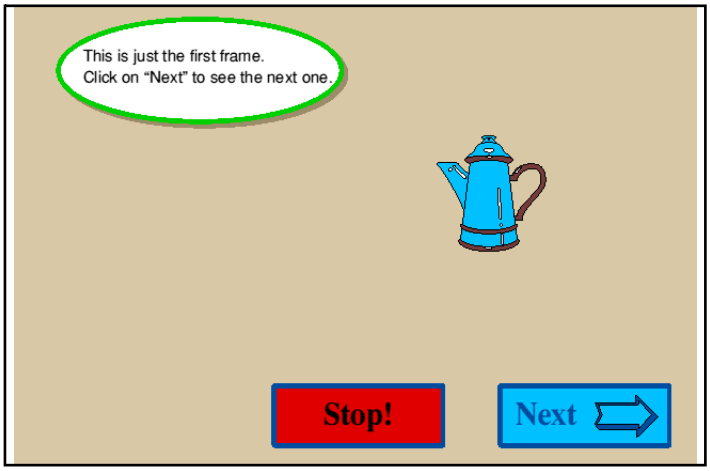

Figure 2

The next few lines of the CompoScript use the "mouseover" command. This asks Compo to respond when the mouse pointer is moved over the relevant object. Here we ask Compo only to do something when the pointer is over an object and the mouse is clicked. The command is "mouseover < objectname> on click proc <dothis>". So in this case, if we move the pointer over the object named "stop" and click, Compo knows to find the procedure called "stop" and perform the actions it contains. Note that there is no need for the object name and that of the procedure to be the same. Nor does this have to be the same as an text that may display in the object on the canvas. So, when we put the pointer over the object that displays the word "next" and click, the procedure that is run as a result is one I have chosen to call "nextframe". In most cases it makes sense to use similar names for the procedures and the objects which call them as it makes it easier to work out what is going on, but you can actually call them whatever you wish!

The next few lines are a bit of a trick. We want the script to go on running until we are finished with it. We therefore set a new variable "let done = 0" and then run an infinite "repeat... until" loop. This keeps testing to see if "done" is now 1. (CompoScript treats "0" and "1" as being equivalent to "false" and "true". Again, this is a fairly common trick amongst programming languages.) The script therefore continues to control things until we manage to set "done" to 1 in some way. Once this happens the loop ends and we move on to the next line "scriptoptions showpanes". This makes the toolbars visible again. The line "end" then causes the script to stop running and Compo then reverts to desktop operation.

Let's now look at the procedures which follow the "end" line in the CompoScript file. The first two are as follows:

```
defproc load_frame
loadcompo <.Composcript$dir>.frames.frame<thisframe>
endproc

defproc stop
let done = 1
endproc
```

Note that as with BASIC, the defn of the procedure starts with a "defproc <procname>" and ends with an "endproc". The "load_frame" procedure only has one active line. It uses the "loadcompo < filename>" command to load a new compo canvas. Note the value ".Composcript$dir" shown in angle brackets. This allows Compo to use the name of the directory from which the CompoScript file comes. Figure 3 shows the filer windows where I can view the script file (which I called "presVIDC") and also the contents of the subdirectory, "frames" where I put the frames to be displayed. When I drag and drop this file onto the canvas, Compo will set ".Composcript$dir" to the directory from which the script came. (By the way, note the "." at the start of this variable. This is a convention for CompoScript variables which Compo itself tends to set.)

Since I started off by setting "thisframe" to hold the value, 1, the procedure will initially load "frame1" from this subdirectory as it will be able to find it using the values I have given.

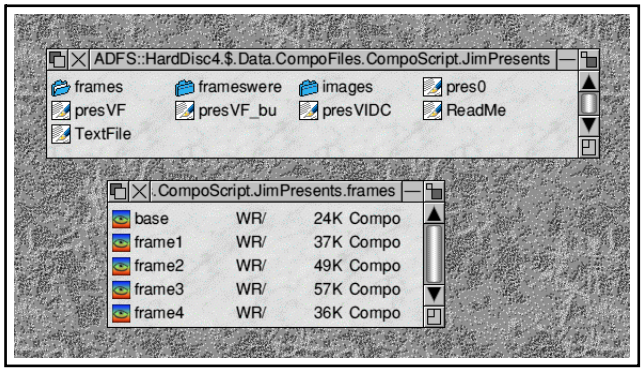

Figure 3

The next procedure "stop" will be carried out when you click on the object of the same name. This sets "done" to 1. As a result the loop ends, the script ends, and Compo reverts back to desktop mode. This means that clicking on the "stop" object finishes the presentation.

```
defproc nextframe
let thisframe = thisframe + 1
fileinfo <.Composcript$dir>.frames.frame<thisframe>
if .fileinfo.there = 1 then
proc load_frame
else
let thisframe = thisframe - 1
endif
endproc
```

The next procedure is "nextframe". By now you should be beginning to get the idea of what is going on here. This increases the "thisframe" number by one, thus moving attention to the next frame. It then uses "proc load_frame" to actually load the new canvas, replacing the old one. However, for safety, before doing this it uses the "fileinfo" command to get information on the proposed new frame. This sets a series of CompoScript values which can, when needed, be used by Compo to decide what to do with the file. In this case the reason for this is to check that the file actually exists! If it does, then the "fileinfo" line will have set the variable ".fileinfo.there" to equal 1. If the file does not exist, this will be set to 0. We can then use the "if... then... else... endif" construction to decide what to do. In this case, if the file exists it is loaded, if it is not, the value of "thisframe" is set back to its old value. The result of all this is that when we click the mouse over the "next" object we will get the next frame displayed – if it exists! – and nothing will change if it doesn't.

Having displayed the first frame, when we click on the displayed "next" object the display promptly changes to the next frame (frame2) which is shown in figure 4. This frame has some different objects These include the "previous" object which shows the word "back" on the canvas. When clicked over, this causes the "previousframe" procedure to be carried out. This is almost identical to the "next" procedure as it performs much the same task except that it steps back one frame instead of forwards one.

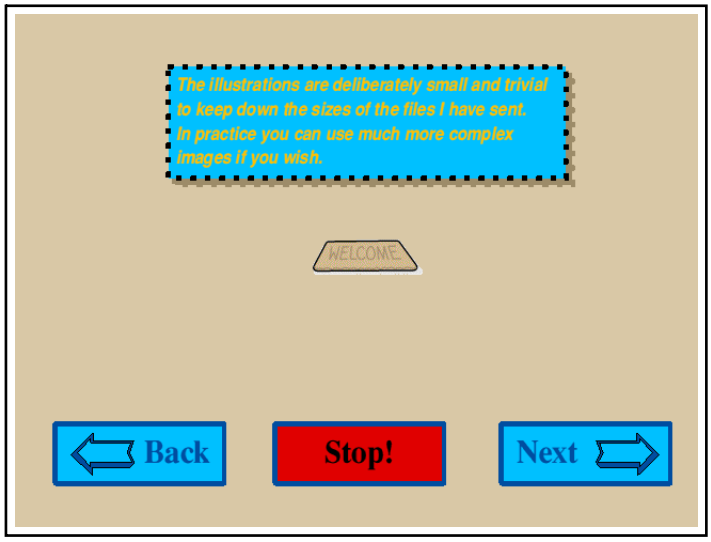

Figure 4

The "previousframe" procedure therefore looks like the following:

```
defproc previousframe
let thisframe = thisframe - 1
fileinfo <.Composcript$dir>.frames.frame<thisframe>
if .fileinfo.there = 1 then
proc load_frame
else
let thisframe = thisframe + 1
endif
endproc
```

By including the above procedures and the main script into a file you can therefore get the script to make Compo display, in turn, the "frame" files held in the directory "frames". This can hold as many frames (slides) as you wish provided that they are numbered consecutively, starting from "frame1". Then using the "back/stop!/next" objects as if they were control buttons on the displayed frames you can step through the presentation and quit when you like.

In fact you can do more than this since Compo can alter the displayed details of the canvas in most of the usual ways. If you wished you could arrange that some objects would move, or become translucent, when you clicked on them. This allows you to add some interactive variety to a series of slides. As an example, I added another "mouseover" procedure which is called when you click on an object called "clickme". This is displayed on the third frame of my demonstration. When it first appears this looks like figure 5.

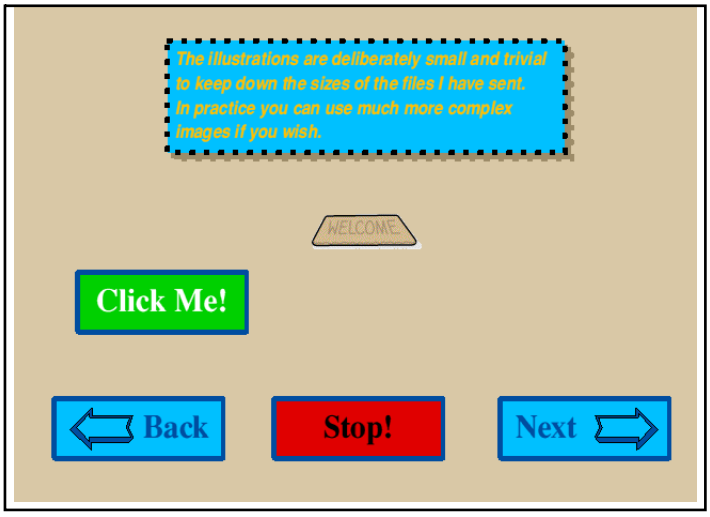

Figure 5

This particular frame includes an object that has the name "clickme". If you look back at the main part of the composcript, there is a "mouseover" action defined for this object. It calls the procedure "tadaa" which is as follows

```
defproc loadtadaa
nib up
loadimage <.Composcript$dir>.images.tadaa
select "tadaa"
moveto 200 100
select NONE
nib down
endproc
```

This finds the file called "tadaa" in the "Images" directory you can see in the filer window shown in figure 3, loads it onto the canvas and locates it with its bottom left corner at a position 200 pixels to the right and 100 up from the bottom left corner of the canvas. The result is that a new object appears as shown in figure 6.

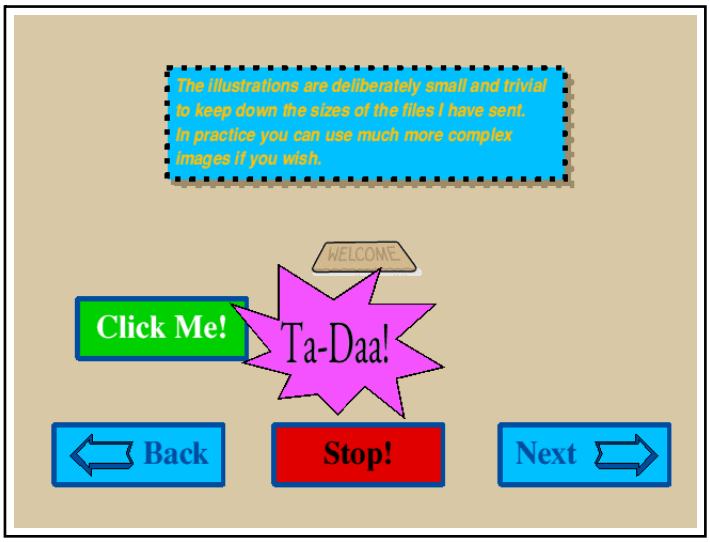

Figure 6

This serves as a simple example of a way in which you can temporarily alter the currently displayed

frame when you wish to add something to surprise your audience. It shows that you aren't limited to just a slide-by-slide show.

Before finishing, a few words about screen displays. For the above I assumed you would use the 640 by 480 screenmode in 16 million colours. At then end of the CompoScript I have given as an example the computer goes back onto the desktop but stays in this mode. If you wish, you can revert to the previous mode by adding an extra line or two. The simplest way is to make use of CompoScript's flexible “star” command which just acts like a command typed into the computer's command line. So to return to, say, a larger 256 colour mode you would use a line like “star "WimpMode X1024 Y768 C256"”. (Note the quotes around the values that follow the “WimpMode”.)

If like me you are now lucky enough to be using a ViewFinder card you should also add some extra lines to ensure that you get the modes you intend as so many extra modes are possible with a ViewFinder! In my case I alter the first few lines of the main section of the CompoScript to

```
makecanvas 640 480 0
star "WimpMode V X640 Y480 C16M F91"
scriptoptions fullscreen "X640 Y480 C16M"
```

This gives me a 91Hz framerate for the fullscreen display, avoiding the flicker of the standard VIDC output. At the end of the main section of the CompoScript I also add a line

```
star "WimpMode V X1280 Y1024 C16M F80"
```

to return to a large desktop mode that I prefer to use with the ViewFinder.

Ok, that's it for this month. This was a deliberately complex and challenging example of CompoScript to give you some idea of its flexibility. I'd like to take the opportunity to thank Rob Davison, both for enabling Compo to support presentations, and for providing me with a demo upon which mine is based. I'll probably return to simpler example next time. I'll also ask Paul (if there is room) to put the demonstration presentation onto the magazine disc for you to try out if your copy of Compo is up to date.

2310 words
6th Nov 2000
Jim Lesurf

Compo Clues #5

The subject for this month's article was suggested to me by some discussions on the Archive emailing list. One of the tasks that many people wish to perform is to combine two (or more) images to make up a ‘panorama’ image that shows a much wider field of view than possible with just one image. With the increasing use of digital cameras and slide/print scanners this task becomes useful more and more often. Hence this article is designed to show how easily this task can be done using Compo. I'll tackle the task in three ways so you can see the choices available. For the sake of example I'll use the two images shown on the Compo canvas in figure 1.

Figure 1

For demonstration purposes I have chosen two sections of a larger initial photograph of a lady feeding some swans. They represent two views, each of which has something missing. (In one case the tails of the swans, in the other case one of the heads.)

Now the simplest way to join these images is just to select one and drag it 'by hand' – i.e. with the mouse or the arrow-keys. In recent versions of Compo image dragging is interactive. This means that the image itself moves, not just an outline rectangle box. This makes it easier to see how you are doing when using the mouse. However screen redrawing of large images can be slow on a RiscPC, and alignment can be quite fiddly to perform precisely. You may also need a ViewFinder card to avoid the feeling that what you are doing is a computer version of 'keyhole surgery'! Alas, these are mainly limitations of the RiscPC, not of Compo. However despite them I was able to move the images to get the result shown in Figure 2 in about a minute.

Figure 2

Paul’s printers have done very well with recent issues, so I hope you can see in the printed magazine that the result looks fairly good, and that the join where one image overlaps the other is effectively invisible. Please note, though, that the images I have sent to Paul are scaled down from much larger originals so as to avoid his having to cope with huge files. Hence some details may have been lost.

To make the whole process easier we can now use a second method, based upon Composcript. This helps get around the irritating limitations of the RiscPC, and makes the whole process quicker and easier. Lets see how the script can be used in practice first, then examine how it works. The script itself is called “alignmarks” and it uses a pair of simple drawfiles as markers.

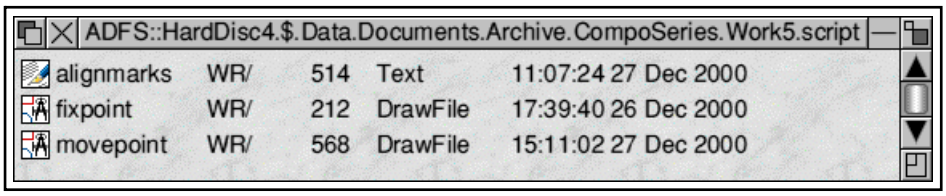

Figure 3

Lets start again with the Compo canvas as shown in Figure 1 and simply drag and drop the “alignmarks” file onto the canvas. This loads onto the canvas a pair of drawfiles which appear as coloured cross-hairs. Figure 4 shows these markers as they first appear.

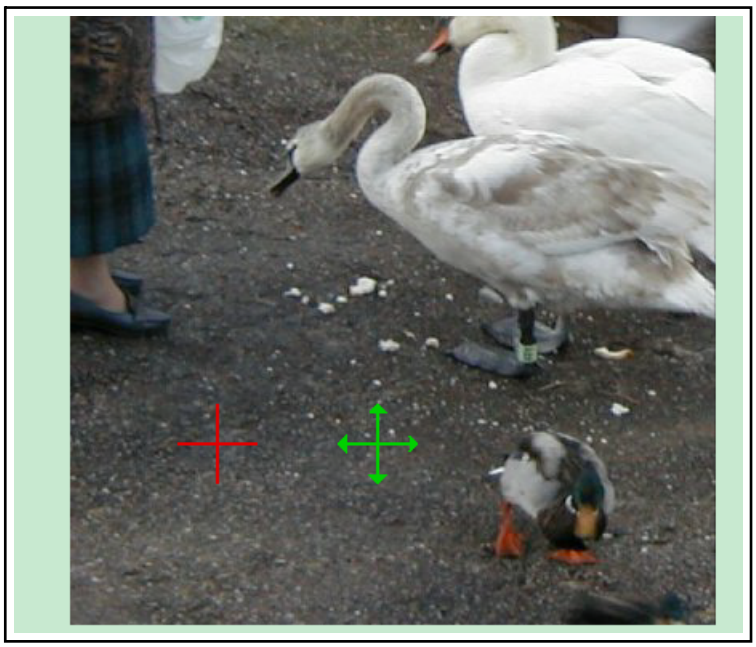

Figure 4

To make the difference between them clear, one marker is red and has plain lines, whilst the other is green and has arrowheads. The plain red marker is “fixpoint”, the green arrowheaded one is “movepoint”. Once these are on the canvas, select each in turn and move them to matching points in the parts of the pair of images that show the same item. For the sake of this demonstration I have used a feature on the bill of one swan. Hence having moved the markers I get the canvas as shown in Figure 5. (In fact I adjusted the Compo window to only show a section of the canvas and zoomed in whilst doing this to make the process easier.)

Figure 5

Now select the image over which the “movepoint” marker is placed. (In this case the image to the right) and once again drag and drop the “alignmarks” scriptfile onto the canvas. This now moves the selected image so as to shift the marked position on it to align with the marked position on the canvas. i.e. it moves the image so that the place that was under the green arrow-headed marker is now under the plain red one. The result is shown in Figure 6.

Figure 6

The whole process takes far longer to describe than to perform. In practice I found that by using the “alignmarks” script it took me less than 20 seconds to align the images as shown in Figure 6. You can zoom in when moving the markers so as to place them accurately. However even if you only place them roughly, the result can easily be adjusted using the arrow-keys to ‘nudge’ one of the images a pixel at a time until you are satisfied with the result. If you wish, you can then also trim away some of the duplicated area where one image sits over the other.

The markers are left on the canvas once the process is done. This is so you can reuse them to align another pair of images if you wish. Just move them to new locations, select the new image you wish to move, and drag-and-drop the scriptfile onto the canvas again. Alternatively, you can delete the markers once finished with them.

The script which performs the process consists of three sections. The first part is as follows:

```
# Composcript

ifexists "movepoint" then
 proc moveit
 else
 proc loadpoints
 endif
end
```

This looks to see if an object called “movepoint” is already present on the canvas. If it is not, the script calls a procedure “loadpoints”. If it finds the object is already present it calls a procedure called “moveit”. The file therefore provides both the loading of the marker files and the movement process depending upon whether you are ready for the move or not. The second section of the script file is the procedure

```
defproc loadpoints
let done = 0
loadimage <.composcript$dir>.fixpoint
select "fixpoint"
moveto 100 100
loadimage <.composcript$dir>.movepoint
select "movepoint"
moveto 200 100
select none
endproc
```

This finds the two drawfiles “fixpoint” and “movepoint” which are in the same directory as the script file and loads them onto the canvas at two convenient locations near the bottom left corner of the canvas. The final section of the script file is the other procedure

```
defproc moveit
let chosen = .leafname
select "fixpoint"
let fx = .xpos
let fy = .ypos
select "movepoint"
let mx = .xpos
let my = .ypos
select "<chosen>"
moveby fx-mx fy-my
endproc
```

This makes a note of the name of the object you have selected for movement and records its name in the script variable “chosen”. It then determines the locations of the two markers. (To do this it has to select each marker point object in turn.) It then reselects the object you want moved and shifts it by the difference in positions between the markers. Provided you have placed the markers carefully the result is that the images now align.

Having shown how a script can make the process easier, we can go on and make things even simpler by using a Compo Scriptlet. To do this we have to make a few minor changes to the text in the script file and put the result into the directory “!Compo.Resources.Scripts”. The essential changes are quite simple; we just add two lines near the start of the file. These are

```
#cs_align,pcs_align
#Stitch images together using markers
```

The first line gives the names of two 32 by 32 pixel sprites which Compo can use as icon buttons to run the scriptlet. You need to add these two sprites to the sprite file “!!CSIcon” which also resides in Compo’s scripts directory. These icons can look however you wish. I have created a suitable pair and will give them to Paul to distribute on the magazine disc if he has room. The above two lines and the extra sprites are all that are required to turn a script into a scriptlet that Compo will now run at the click of an icon. Having added these, when I run Compo my scriplets bar now looks like as shown in figure 7.

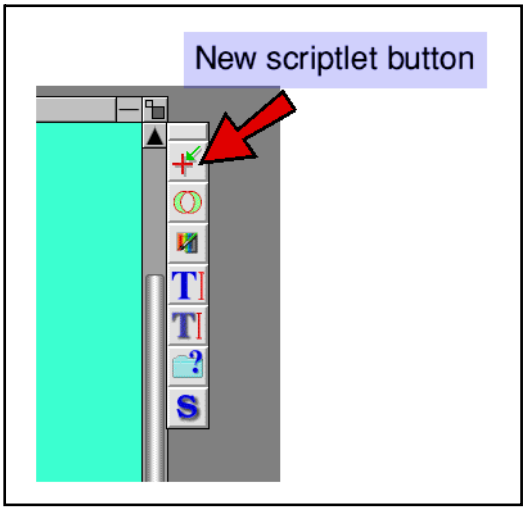

Figure 7

The second line added to the script is a help text that will show along the top of Compo’s palette bar when you move the mouse pointer over the scriptlet’s icon. This is useful as a reminder of which icon does what!

In principle that’s all we have to do, however in practice it is sensible to make one more change. The original script assumed that the drawfiles it uses as markers sit beside it in the same directory as the script file. We could put these drawfiles into Compo’s scripts directory, but if we keep adding odd files like this the directory may fill up with odds and ends and we may get muddled. It therefore makes sense to move the drawfiles somewhere else, and get the scriptlet to find them there.

For this kind of purpose I have produced a simple resource called “!CompoBits”. This just acts as a place Compo can look for resource files required by scripts and scriptlets. Figure 8 shows the contents of the directory were I keep !Compo and !CompoBits beside each other.

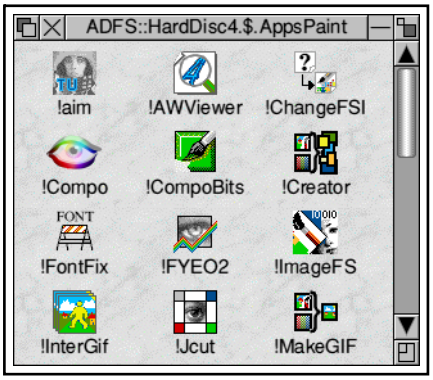

Figure 8

I have placed copies of the marker drawfiles into a sub-directory 'shapes' inside CompoBits and altered the lines in the scriptlet that load these file to read as follows

```
loadimage <CompoBits$Dir>.shapes.fixpoint
```

and

```
loadimage <CompoBits$Dir>.shapes.movepoint
```

Provided that CompoBits has been 'seen' by the filer these can now be found and loaded when required. I will send Paul a copy of !CompoBits along with the other items described in this article so if he wishes he can distribute them on the magazine disc. I may also put copies onto the 'Compo Clues' website for readers who have web access but aren't subscribers to the Archive monthly disc.

When I tested aligning some pairs of images using the scriptlet it typically took me around 20 seconds to complete the process. Most of this time was taken up with shifting the marker icons to where I wanted them. This isn't a task I perform often myself, but I hope that the script and scriptlet will be useful to those who frequently want to create larger images by stitching together images.

1730 words
J. C. G. Lesurf
29th Dec 2000

Compo Clues #6

Firstly, apologies for the lack of articles during the last few months. This has been due to "circumstances beyond my control" as RailTrack would say. In this article I want to have a look at some of the new features that have been added to Compo recently with the release of version 1.21. There are changes in a number of areas, some subtle, some bugfixes. However a couple are worth looking at in detail as they are so useful. The first of these is some extensions to the text creation capabilities.

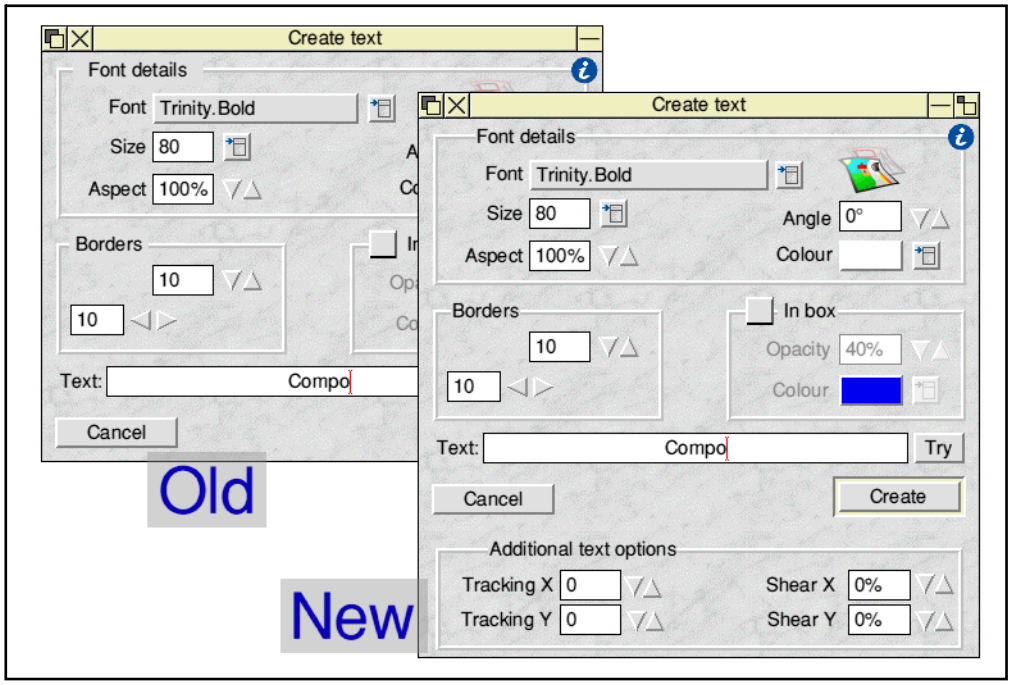

Figure 1

Figure one shows both the “old” and “new”text creation dialogue windows. You can see that the new version has some extra features which can be used to modify the way the text is displayed.

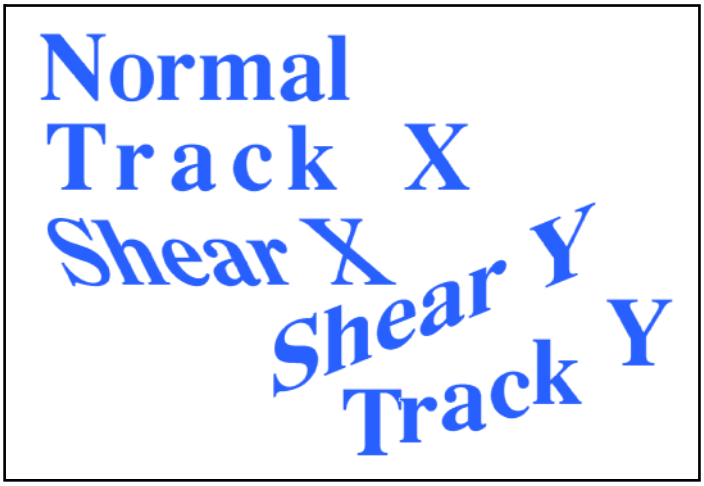

Figure 2

Figure 2 shows some examples of the new effects. Note that you can combine them if you wish. For anyone unfamiliar with the term “tracking” as applied to text, the examples show that X-tracking essentially means slightly displacing each character horizontally from its normal position beside its neighbours. Y-tracking means a vertical displacement from each character to the next. (Strictly speaking we should not say horizontal or vertical here as the directions are defined with respect to the nominal baseline of the characters.)

The main change that I want to discuss is that Compo now permits direct access to image pixel values. Subscribers to the “Archive on Line” mailing list may recall discussions some months ago about ways to automate aligning sets of images into a ‘panorama’. Ideally, this requires the ability to read the pixel information so as to be able to compare the patterns in individual images, find

similarities, and use this to decide where to place the images relative to one another.

At the time Compo didn't provide any way for a simple script or external program to do this without being a full multitasking PCA-compliant desktop application. However, from version 1.21 onwards, Compo allows direct access to image data by two relatively easy methods. The first of these is via a new set of composcript commands and variables.

From version 1.21 onwards the composcript command, "readpixel x y", will set four values - ".r .g .b .p". (In each case the dot before the letter indicates that Compo uses these as its own variable.) The .r, .g, and .b, values are the levels of red, blue, and green at the pixel that is x pixels across the image from the left-hand edge, and y pixels up from its lower edge. These three values are integers in the range 0 - 255. The same information is also stored in the .p variable, but in this case it is a 32 bit RGB word of the form XXBBGGRR. This command can therefore be used to read information about the selected object, and you can use whichever of the data formats provided that you prefer.

In an equivalent manner you can now use "writepixel x y red blue green" to set the colour of the pixel at the location x y to have the levels of red blue and green specified. So the command "writepixel 20 50 100 150 200" would change the colour of the pixel at the location 20 pixels from the left and 50 pixels up from the bottom to have a red level of 100, a blue level of 150, and a green level of 200. Alternatively you can write values using "writepixel x y z" where z is now a 32bit value in the same format as .p. In fact the writepixel command will also use the most recently read values, so if you just use the command "writepixel x y" the colour of that pixel will be set to the last value that was returned by a "readpixel" command.

Figure 3

Figure 3 shows a simple illustration of the kinds of things you can do using pixel reading and writing. The image on the left shows an object (picture of Tom Baker as Dr Who) before the example script was run. The one on the right shows the result of running the script listed below.

```
# Composcript
nib up
select first
let ystart = .height/4
let ystop = 3*ystart
let xstart = .width/4
```

```
let xstop = 3*xstart
for ystep = ystart to ystop
for xstep = xstart to xstop
readpixel xstep ystep
writepixel xstep ystep .r+100 .g+100 .b+100
next xstep
redraw 1 <ystep> <.width> 1
next ystep
nib down
end
```

The script reads through the values of pixels in a central section of the image and then writes back new, lighter, values to the same places. The result is to brighten a central part of the image. In principle you can manipulate each value as you choose. This new feature is potentially very powerful. However there is one snag – it is a very slow way to process a large image! The example image used here was just 150 pixels wide and 150 pixels high. Yet altering just the central quarter of the image took over three minutes! The process would have been faster if the script had not issued a “redraw” after each line had been processed, and closed the window that reports script progress, but the process would still be slow.

The “redraw” command is also new. It allows you to request that a portion of the image be redrawn even when the ‘nib’ is up (and hence automatic redrawing is disabled). If you just include “redraw” in the script it prompts a complete redrawing of the selected object. If you just want redraw a portion of the object you can use “redraw x y w h” where x and y would indicate bottom left corner of the rectangle to be redrawn, w and h would indicate the width and height of the rectangle to redraw. (All these values, as before, being integers in pixels.) The command used in the script therefore redraws each line of pixels in the image after it has been altered so you can (slowly!) watch the change creeping up Dr Who’s face.

The reason the composcript readpixel/writepixel method is slow is that composcript is an interpreted language using system variables. When the progress window is open it also multitasks, which is useful to see how things are progressing and pick up any errors. However, to speed things up we really need to be able to use a single tasking program to manipulate the image data – preferably written in assembler, or compiled from a language like ‘C’. To permit this, version 1.21 of Compo now provides direct access to the image data it holds for manipulation by other programs. An example of its use is shown below.

Figure 4

Figure 4 shows a ‘before and after’ pair of images similar to those in figure 3. In this case, however, the basic image is 420 by 420 pixels. Yet this time, lightening the central section of the image only took about a quarter of a second – i.e. around 750 times faster than before! Since the image is bigger than the Dr Who one, the reduction in the time taken to process each pixel is even greater. As a result, the process is actually around 1000 times faster than the previous example. The reason for this is that process was performed using a small program written in ‘C’.

The program itself is too long to include here, and won’t be of much use to you if you don’t understand ‘C’ and don’t have a compiler. However if Paul has space it should be on the magazine disc. In fact, I could have written a program in BASIC, or any other language (e.g. FORTRAN) and used it in the same way. Since BASIC is interpreted, it would probably be slower than compiled ‘C’, but it would still be a lot faster than doing it all using composcript. In addition, by using languages like these we can access all the methods, and features they have to offer. Hence this represents a dramatic increase in the flexibility of the ways we can read and process images on the Compo canvas. In effect, we can now use Compo as a ‘host’ for many forms of image manipulation.

In each case the program can now be used via a simple script of the form shown below

```
#CompoScript
star "WimpTask <.composcript$dir>.lightsquare <.selected.imagebase>"
redraw
end
```

This script uses a command “WimpTask” to launch the program that will process the image. In this example it calls a program I have named “lightsquare” which is placed in the same directory as the script file. The WimpTask call also sends the variable “<.selected.imagebase>” to the program. This variable provides the information the program needs to find the actual image data which is held by Compo as a deep sprite. The called program can therefore now find the image data and is able to read and write directly to the image on the Compo canvas. Since the script itself does not select the object to be worked upon you need to select one before running the above script. In practice, you could arrange to select what object you wish in the script itself if you wish. You could also add other

variables and values to the "WimpTask" line to communicate them to the program.

In practice, of course, simply brightening the center of an image isn't likely to be very useful. I chose this just to ensure the effect shows up clearly when Paul prints the result in greyscale. A more subtle effect (which alas may not show up so well in the magazine) is illustrated in figure 5

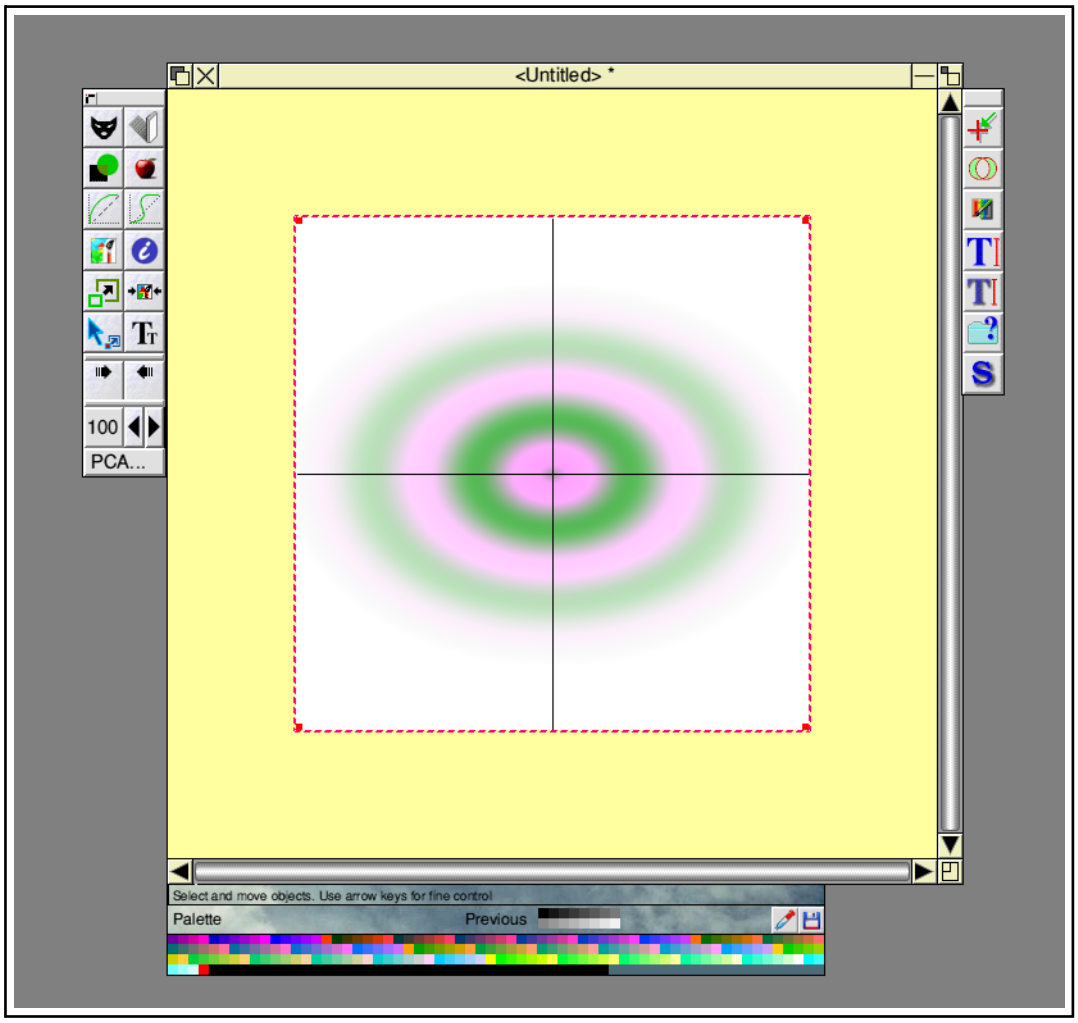

Figure 5

Figure 5 is an example of an image that was entirely created by using a program to write values directly onto an object on the Compo canvas. For the sake of example I have plotted a coloured sinusoidal wave pattern that radiates elliptically from the center of the object and fades gradually away from the center. I have also overplotted a pair of axies. If I had wanted, the plot could have come from a data file of some kind. Also, if I'd wanted, I could have plotted a series of similar images and then used the composcript to save them as an animated GIF to show ripples moving outward.

The point here isn't the specific pattern I have drawn, but that you can draw whatever you wish, based on any data or expressions you choose. The results can then be saved either as a single image, or in animated form. It should also be possible to make this process using scriptlet buttons, etc, if desired. In a similar way, it is now possible to load images onto the Compo canvas and analyse the patterns they contain. You can then either save data files recording the results, or use the result to enhance or alter the image in whatever way you choose. This makes Compo now very attractive as a platform for

image analysis, and enhancement, as well as for image creation.

1850 words
Jim Lesurf
22nd Jun 2001