Fluxus Manual 0.16

Table of Contents

Introduction

Fluxus is an environment which allows you to quickly make live animation and audio programs, and change them constantly and flexibly. This idea of constant change (flux) is where it's name comes from.

Fluxus does this with the aid of the Scheme programming language, which is designed for flexibility; and an interface which only needs to provide you with program code floating above the resulting visual output. This interface enables fluxus to be used for livecoding, the practice of programming as a performance art form. Most users of fluxus are naturally livecoders, and some write fluxus scripts in front of audiences, as well as using it to rapid prototype and design new programs for performance and art installation.

This emphasis on live coding, rapid prototyping and quick feedback also make fluxus fun for learning computer animation, graphics and programming – and it is often used in workshops exploring these themes.

This manual is vaguely organised in terms of thing you need to know first being at the beginning and with more complex things later on, but it's not too strict so I'd recommend jumping around to parts that interest you.

Quickstart

When you start fluxus, you will see the welcome text and a prompt – this is called the repl (read evaluate print loop), or console. Generally fluxus scripts are written in text buffers, which can be switched to with ctrl and the numbers keys. Switch to the first one of these by pressing ctrl-1 (ctrl-0 switched you back to the fluxus console).

Now try entering the following command.

```
(build-cube)
```

Now press F5 (or ctrl-e) – the script will be executed, and a white cube should appear in the centre of the screen. Use the mouse and to move around the cube, pressing the buttons to get different movement controls.

To animate a cube using audio, try this:

```
; buffersize and samplerate need to match jack’s
(start-audio “jack-port-to-read-sound-from” 256 44100)

(define (render)
        (colour (vector (gh 1) (gh 2) (gh 3)))
        (draw-cube))

(every-frame (render))
```

To briefly explain, the (every-frame) function takes a function which is called once per frame by fluxus’s internal engine. In this case it calls a function that sets the current colour using harmonics from the incoming sound with the (gh) - get harmonic function; and draws a cube. Note that this time we use (draw-cube) not (build-cube). The difference will be explained below.

If everything goes as planned, and the audio is connected with some input – the cube will flash in a colourful manner along with the sound.

Now go and have a play with the examples. Load them by pressing ctrl-l or on the commandline, by entering the examples directory and typing fluxus followed by the script filename.

User Guide

When using the fluxus scratchpad, the idea is that you only need the one window to build scripts, or play live. F5 (or ctrl-e) is the key that runs the script when you are ready. Selecting some text (using shift) and pressing f5 will execute the selected text only. This is handy for re-evaluating functions without running the whole script each time.

Camera control

The camera is controlled by moving the mouse and pressing mouse buttons.

- Left mouse button: Rotate
- Middle mouse button: Move
- Right mouse button: Zoom

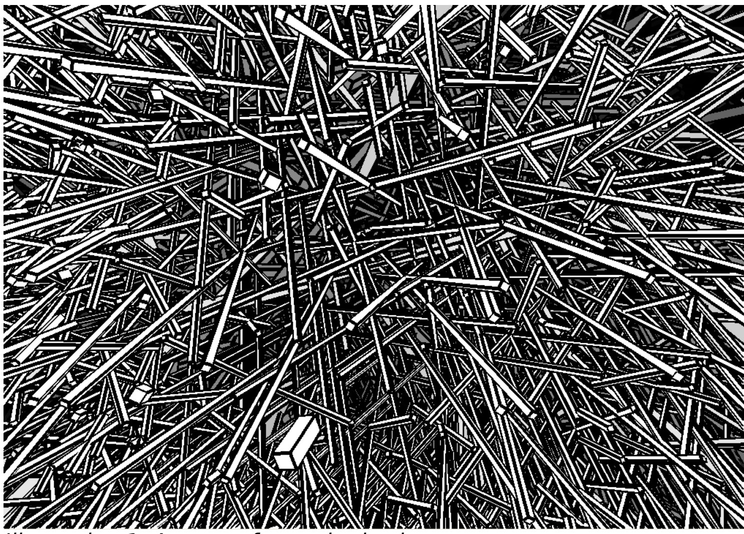


Illustration 1: A mess of stretched cubes

Workspaces

The script editor allows you to edit 9 scripts simultaneously by using workspaces. To switch workspaces, use ctrl+number key. Only one can be run at once though, hitting f5 will execute the currently active workspace script. Scripts in different workspaces can be saved to different files, press ctrl-s to save or ctrl-d to save-as and enter a new filename (the default filename is temp.scm).

The REPL

If you press ctrl and 0, instead of getting another script workspace, you will be presented with a read evaluate print loop interpreter, or repl for short. This is really just an interactive interpreter similar to the commandline, where you can enter scheme code for immediate evaluation. This code is evaluated in the same interpreter as the other scripts, so you can use the repl to debug or inspect global variables and functions they define. This window is also where error reporting is printed, along with the terminal window you started fluxus from.

One of the important uses of the repl is to get help on fluxus commands. For instance, in order to find out about the build-cube command, try typing:

```
(help "build-cube")
```

You can find out about new commands by typing

```
(help "sections")
```

Which will print out a list of subsections, so to find out about maths commands you can try:

```
(help "maths")
```

Will give you a list of all the maths commands availible, which you can ask for further help about. You can copy the example code by moving the cursor left and then up, shift select the code, press ctrl-c to copy it. Then you can switch to a workspace and paste the example in with ctrl-v order to try running them.

Keyboard commands

- ctrl-f : Full screen mode.
- ctrl-w : Windowed mode.
- ctrl-h : Hide/show the editor.
- ctrl-l : Load a new script (navigate with cursors and return).
- ctrl-s : Save current script.
- ctrl-d : Save as – current script (opens a filename dialogue).
- ctrl-1 to 9 : Switch to selected workspace.
- ctrl-0 : Switch to the REPL.
- ctrl-p : Auto format the white space in your scheme script to be more pretty and readable
- F3 : Resets the camera.
- F4 : Execute the current highlighted s-expression
- F5/ctrl-e : Execute the selected text, or all if none is selected.
- F6 : Reset interpreter, and execute text
- F9 : Randomise the text colour (aka the panic button)
- F10 : Make the text more transparent
- F11 : Make the text less transparent

Scheme

Scheme is a programming language invented by Gerald J. Sussman and Guy L. Steel Jr. in 1975. Scheme is based on another language – Lisp, which dates back to the fifties. It is a high level language, which means it is biased towards human, rather than machine understanding. The fluxus scratchpad embeds a Scheme interpreter (it can run Scheme programs) and the fluxus modules extend the Scheme language with commands for 3D computer graphics.

This chapter gives a very basic introduction to Scheme programming, and a fast path to working with fluxus – enough to get you started without prior programming experience, but I don’t explain the details very well. For general scheme learning, I heartily recommend the following books (two of which have the complete text on-line):

The Little Schemer Daniel P. Friedman and Matthias Felleisen

How to Design Programs

An Introduction to Computing and Programming Matthias Felleisen Robert Bruce Findler Matthew Flatt Shriram Krishnamurthi Online: http://www.htdp.org/2003-09-26/Book/

Structure and Interpretation of Computer Programs Harold Abelson and Gerald Jay Sussman with Julie Sussman Online: http://mitpress.mit.edu/sicp/full-text/book/book.html

We'll start by going through some language basics, which are easiest done in the fluxus scratchpad using the console mode – launch fluxus and press ctrl 0 to switch to console mode.

Scheme as calculator

Languages like Scheme are composed of two things – operators (things which do things) and values which operators operate upon. Operators are always specified first in Scheme, so to add 1 and 2, we do the following:

```
fluxus> (+ 1 2)
3
```

This looks pretty odd to begin with, and takes some getting used to, but it means the language has less rules and makes things easier later on. It also has some other benefits, in that to add 3 numbers we can simply do:

```
fluxus> (+ 1 2 3)
6
```

It is common to "nest" the brackets inside one another, for example:

```
fluxus> (+ 1 (* 2 3))
7
```

Naming values

If we want to specify values and give them names we can use the Scheme command "define":

```
fluxus> (define size 2)
fluxus> size
2
fluxus> (* size 2)
4
```

Naming is arguably the most important part of programming, and is the simplest form of what is termed "abstraction" - which means to separate the details (e.g. The value 2) from the meaning – size. This is not important as far as the machine is concerned, but it makes all the difference to you and other people reading code you have written. In this example, we only have to specify the value of size once, after that all uses of it in the code can refer to it by name – making the code much easier to understand and maintain.

Naming procedures

Naming values is very useful, but we can also name operations (or collections of them) to make the code simpler for us:

```
fluxus> (define (square x) (* x x))
fluxus> (square 10)
100
fluxus> (square 2)
4
```

Look at this definition carefully, there are several things to take into account. Firstly we can describe the procedure definition in English as: To (define (square of x) (multiply x by itself)) The “x” is called an argument to the procedure, and like the size define above – it’s name doesn’t matter to the machine, so:

```
fluxus> (define (square apple) (* apple apple))
```

Will perform exactly the same work. Again, it is important to name these arguments so they actually make some sort of sense, otherwise you end up very confused. Now we are abstracting operations (or behaviour), rather than values, and this can be seen as adding to the vocabulary of the Scheme language with our own words, so we now have a square procedure, we can use it to make other procedures:

```
fluxus> (define (sum-of-squares x y)
       (+ (square x) (square y)))
fluxus> (sum-of-squares 10 2)
104
```

The newline and white space tab after the define above is just a text formatting convention, and means that you can visually separate the description and it’s argument from the internals (or body) of the procedure. Scheme doesn’t care about white space in it’s code, again it’s all about making it readable to us.

Making some shapes

Now we know enough to make some shapes with fluxus. To start with, leave the console by pressing ctrl-1 – you can go back at any time by pressing ctrl-0. Fluxus is now in script editing mode. You can write a script, execute it by pressing F5, edit it further, press F5 again... this is the normal way fluxus is used.

Enter this script:

```
(define (render)
       (draw-cube))

(every-frame (render))
```

Then press F5, you should see a cube on the screen, drag the mouse around the fluxus window, and you should be able to move the camera – left mouse for rotate, middle for zoom, right for translate.

This script defines a procedure that draws a cube, and calls it every frame – resulting in a static cube.

You can change the colour of the cube like so:

```
(define (render)
       (colour (vector 0 0.5 1))
       (draw-cube))

(every-frame (render))
```

The colour command sets the current colour, and takes a single input – a vector. Vectors are used a lot in fluxus to represent positions and directions in 3D space, and colours – which are treated as triplets of red green and blue. So in this case, the cube should turn a light blue colour.

Transforms

Add a scale command to your script:

```
(define (render)
       (scale (vector 0.5 0.5 0.5))
       (colour (vector 0 0.5 1))
(draw-cube))

(every-frame (render))
```

Now your cube should get smaller. This might be difficult to tell, as you don't have anything to compare it with, so we can add another cube like so:

```
(define (render)
       (colour (vector 1 0 0))
       (draw-cube)
       (translate (vector 2 0 0))
       (scale (vector 0.5 0.5 0.5))
       (colour (vector 0 0.5 1))
       (draw-cube))

(every-frame (render))
```

Now you should see two cubes, a red one, then the blue one, moved to one side (by the translate procedure) and scaled to half the size of the red one.

```
(define (render)
       (colour (vector 1 0 0))
       (draw-cube)
       (translate (vector 2 0 0))
       (scale (vector 0.5 0.5 0.5))
       (rotate (vector 0 45 0))
       (colour (vector 0 0.5 1))
       (draw-cube))

(every-frame (render))
```

For completeness, I added a rotate procedure, to twist the blue cube 45 degrees.

Recursion

To do more interesting things, we will write a procedure to draw a row of cubes. This is done by recursion, where a procedure can call itself, and keep a record of how many times it’s called itself, and end after so many iterations.

In order to stop calling our self as a procedure, we need to take a decision – we use cond for decisions.

```
(define (draw-row count)
        (cond
                ((not (zero? count))
                        (draw-cube)
                        (translate (vector 1.1 0 0))
                        (draw-row (- count 1)))))

(every-frame (draw-row 10))
```

Be careful with the brackets – the fluxus editor should help you by highlighting the region each bracket corresponds to. Run this script and you should see a row of 10 cubes. You can build a lot out of the concepts in this script, so take some time over this bit.

Cond is used to ask questions, and it can ask as many as you like – it checks them in order and does the first one which is true. In the script above, we are only asking one question, (not (zero? Count)) – if this is true, if count is anything other than zero, we will draw a cube, move a bit and then call our self again. Importantly, the next time we call draw-row, we do so with one taken off count. If count is 0, we don’t do anything at all – the procedure exits without doing anything.

So to put it together, draw-row is called with count as 10 by every-frame. We enter the draw-row function, and ask a question – is count 0? No – so carry on, draw a cube, move a bit, call draw-row again with count as 9. Enter draw-row again, is count 0? No, and so on. After a while we call draw-row with count as 0, nothing happens – and all the other functions exit. We have drawn 10 cubes.

Recursion is a very powerful idea, and it’s very well suited to visuals and concepts like self similarity. It is also nice for quickly making very complex graphics with scripts not very much bigger than this one.

Animation

Well, now you’ve got through that part, we can quite quickly take this script and make it move.

```
(define (draw-row count)
        (cond
                ((not (zero? count))
                        (draw-cube)
```

```
                                (rotate (vector 0 0 (* 45 (sin (time)))))
                                (translate (vector 1.1 0 0))
                                (draw-row (- count 1)))))

        (every-frame (draw-row 10))
```

time is a procedure which returns the time in seconds since fluxus started running. Sin converts this into a sine wave, and the multiplication is used to scale it up to rotate in the range of -45 to +45 degrees (as sin only returns values between -1 and +1). Your row of cubes should be bending up and down. Try changing the number of cubes from 10, and the range of movement by changing the 45.

More recursion

To give you something more visually interesting, this script calls itself twice – which results in an animating tree shape.

```
(define (draw-row count)
        (cond
                ((not (zero? Count))
                        (translate (vector 2 0 0))
                        (draw-cube)
                        (rotate (vector (* 10 (sin (time))) 0 0))
                        (with-state
                                (rotate (vector 0 25 0))
                                (draw-row (- count 1)))
                        (with-state
                                (rotate (vector 0 -25 0))
                                (draw-row (- count 1))))))

        (every-frame (draw-row 10))
```

For an explanation of with-state, see the next section.

Comments

Comments in scheme are denoted by the ; character:

```
; this is a comment
```

Everything after the ; up to the end of the line are ignored by the interpreter.

Using #; you can also comment out expressions in scheme easily, for example:

```
(with-state
        (colour (vector 1 0 0))
        (draw-torus))
(translate (vector 0 1 0))
#;(with-state
        (colour (vector 0 1 0))
        (draw-torus))
```

Stops the interpreter from executing the second (with-state) expression, thus stopping it drawing the green torus.

Let

Let is used to store temporary results. An example is needed:

```
(define (animate)
    (with-state
        (translate (vector (sin (time)) (cos (time)) 0))
        (draw-sphere))
    (with-state
        (translate (vmul (vector (sin (time)) (cos (time)) 0) 3))
        (draw-sphere)))

(every-frame (animate))
```

This script draws two spheres orbiting the origin of the world. You may notice that there is some calculation which is being carried out twice - the (sin (time)) and the (cos (time)). It would be simpler and faster if we could calculate this once and store it to use again. One way of doing this is as follows:

```
(define x 0)
(define y 0)

(define (animate)
    (set! x (sin (time)))
    (set! y (cos (time)))
    (with-state
        (translate (vector x y 0))
        (draw-sphere))
    (with-state
        (translate (vmul (vector x y 0) 3))
        (draw-sphere)))

(every-frame (animate))
```

Which is better - but x and y are globally defined and could be used and changed somewhere else in the code, causing confusion. A better way is by using let:

```
(define (animate)
    (let ((x (sin (time)))
          (y (cos (time))))
        (with-state
            (translate (vector x y 0))
            (draw-sphere))
        (with-state
            (translate (vmul (vector x y 0) 3))
            (draw-sphere))))

(every-frame (animate))
```

This specifically restricts the use of x and y to inside the area defined by the outer let brackets. Lets can also be nested inside of each other for when you need to store a value which is dependant on another value, you can use let* to help you out here:

```
(define (animate)
    (let ((t (* (time) 2)) ; t is set here
          (x (sin t))  ; we can use t here
```

```
            (y (cos t))) ; and here
          (with-state
              (translate (vector x y 0))
              (draw-sphere))
          (with-state
              (translate (vmul (vector x y 0) 3))
              (draw-sphere))))

(every-frame (animate))
```

Lambda

Lambda is a strange name, but it's use is fairly straightforward. Where as normally in order to make a function you need to give it a name, and then use it:

```
(define (square x) (* x x))
(display (square 10))(newline)
```

Lambda allows you to specify a function at the point where it's used:

```
(display ((lambda (x) (* x x)) 10))(newline)
```

This looks a bit confusing, but all we've done is replace square with (lambda (x) (* x x)). The reason this is useful, is that for small specialised functions which won't be needed to be used anywhere else it can become very cumbersome and cluttered to have to define them all, and give them all names.

The State Machine

The state machine is the key to understanding how fluxus works, all it really means is that you can call functions which change the current context which has an effect on subsequent functions. This is a very efficient way of describing things, and is built on top of the OpenGl API, which works in a similar way. For example:

```
(define (draw)
        (colour (vector 1 0 0))
        (draw-cube)
        (translate (vector 2 0 0)) ; move a bit so we can see the next cube
        (colour (vector 0 1 0))
        (draw-cube))

(every-frame (draw))
```

Will draw a red cube, then a green cube (in this case, you can think of the (colour) call as changing a pen colour before drawing something). States can also be stacked, for example:

```
(define (draw)
        (colour (vector 1 0 0))
        (with-state
                (colour (vector 0 1 0))
                (draw-cube))
        (translate (vector 2 0 0))
        (draw-cube))

(every-frame (draw))
```

Will draw a green, then a red cube. The (with-state) isolates a state and gives it a lifetime, indicated by the brackets (so changes to the state inside are applied to the build-cube inside, but do not affect the build-cube afterwards). Its useful to use the indentation to help you see what is happening.

The Scene graph

Both examples so far have used what is known as immediate mode, you have one state stack, the top of which is the current context, and everything is drawn once per frame. Fluxus contains a structure known as a scene graph for storing objects and their render states.

Time for another example:

```
(clear) ; clear the scenegraph
(colour (vector 1 0 0))
(build-cube) ; add a red cube to the scenegraph
(translate (vector 2 0 0))
(colour (vector 0 1 0))
(build-cube) ; add a green cube to the scenegraph
```

This code does not have to be called by (every-frame), and we use (build-cube) instead of (draw-cube). The build functions create a primitive object, copy the current render state and add the information into the scene graph in a container called a scene node.

The (build-*) functions (there are a lot of them) return object id's, which are just numbers, which enable you to reference the scene node after it’s been created. You can now specify objects like this:

```
(define myob (build-cube))
```

The cube will now be persistent in the scene until destroyed with

```
(destroy myob)
```

with-state returns the result of it’s last expression, so to make new primitives with state you set up, you can do this:

```
(define my-object (with-state
               (colour (vector 1 0 0))
               (scale (vector 0.5 0.5 0.5))
               (build-cube)))
```

If you want to modify a object's render state after it’s been loaded into the scene graph, you can use with-primitive to set the current context to that of the object. This allows you to animate objects you have built, for instance:

```
; build some cubes

(colour (vector 1 1 1))
(define obj1 (build-cube))
(define obj2 (with-state
               (translate (vector 2 0 0))
               (build-cube))

(define (animate)
       (with-primitive obj1
               (rotate (vector 0 1 0)))

       (with-primitive obj2
               (rotate (vector 0 0 1))))

(every-frame (animate))
```

A very important fluxus command is (parent) which locks objects to one another, so they follow each other around. You can use (parent) to build up a hierarchy of objects like this:

```
(define a (build-cube))

(define b (with-state
        (parent a)
        (translate (vector 0 2 0))
        (build-cube)))

(define c (with-state
        (parent b)
        (translate (vector 0 2 0))
        (build-cube)))
```

Which creates three cubes, all attached to each other in a chain. Transforms for object a will be passed down to b and c, transforms on b will effect c.

Destroying a object will in turn destroy all child objects parented to it.

You can change the hierarchy by calling (parent) inside a (with-primitive). To remove an object from it's parent you can call:

```
(with-primitive myprim
         (detach))
```

Which will fix the transform so the object remains in the same global position.

Note: By default when you create objects they are parented to the root node of the scene (all primitives exist on the scene graph somewhere). The root node is given the id number of 1. So another way of un-parenting an object is by calling (parent 1).

Note on grabbing and pushing

Fluxus also contains less well mannered commands for achieving the same results as with-primitive and with-state. These were used prior to version 0.14, so you may see mention of them in the documentation, or older scripts.

```
(push)...(pop)
```

is the same as:

```
(with-state ...)
```

and

```
(grab myprim)...(ungrab)
```

is the same as

```
(with-primitive myprim ...)
```

They are less safe than (with-*) as you can push without popping or vice-versa, and shouldn’t be used generally. There are some cases where they are still required though (and the (with-*) commands use them internally, so they won’t be going away)

Input

Quite quickly you are going to want to start using data from outside of fluxus to control your animations. This is really what starts making this stuff interesting, after all.

Sound

The original purpose of fluxus and still perhaps it's best use, was as a sound to light vjing application. In order to do this it needs to take real time values from an incoming sound source. We do this by configuring the sound with:

```
(start-audio “jack-port-to-read-sound-from” 256 44100)
```

Running some application to provide sound, or just taking it from the input on your sound card, and then using:

```
(gh harmonic-number)
```

To give us floating point values we can plug into parameters to animate them. You can test there is something coming through either with:

```
(every-frame (begin (display (gh 0)) (newline)))
```

And see if it prints anything other than 0's or run the bars.scm script in examples, which should display a graphic equaliser.

It's also useful to use:

```
(gain 1)
```

To control the gain on the incoming signal, which will have a big effect on the scale of the values coming from (gh).

Keyboard

Keyboard control is useful when using fluxus for things like non-livecoding vjing, as it's a nice simple way to control parameters without showing the code on screen. It's also obviously useful for when writing games with fluxus.

```
(key-pressed key-string)
```

Returns a #t if the key is currently pressed down, for example:

```
(every-frame
    (when (key-pressed “x”)
       (colour (rndvec))
       (draw-cube)))
```

Draws a cube with randomly changing colour when the x key is pressed.

```
(keys-down)
```

Returns a list of the current keys pressed, which is handy in some situations.

For keys which aren't able to be described by strings, there is also:

```
(key-special-pressed key-code-number)
```

For instance

```
(key-special-pressed 101)
```

Returns #t when the “up” cursor key is pressed. To find out what the mysterious key codes are, you can simply run this script:

```
(every-frame (begin (display (keys-special-down)) (newline)))
```

Which will print out the list of special keys currently pressed. Press the key you want and see what the code is.

Note: These key codes may be platform specific.

Mouse

You can find out the mouse coordinates with:

```
(mouse-x)
(mouse-y)
```

And whether mouse buttons are pressed with:

```
(mouse-button button-number)
```

Which will return #t if the button is down.

Select

While we're on the subject of the mouse, one of the best uses for it is selecting primitives, which you can do with:

```
(select screen-x screen-y size)
```

Which will render the bit of the screen around the x y coordinate and return the id of the highest primitive in the z buffer. To give a better example:

```
; click on the donuts!
(clear)

(define (make-donuts n)
    (when (not (zero? n))
        (with-state
            (translate (vmul (srndvec) 5))
            (scale 0.1)
            (build-torus 1 2 12 12))
        (make-donuts (- n 1))))

(make-donuts 10)

(every-frame
    (when (mouse-button 1)
        (let ((s (select (mouse-x) (mouse-y) 2)))
            (when (not (zero? s))
                (with-primitive s
                    (colour (rndvec)))))))
```

OSC

OSC (Open Sound Control) is a standard protocol for communication between arty programs over a network. Fluxus has (almost) full support for receiving and sending OSC messages.

To get you started, here is an example of a fluxus script which reads OSC messages from a port and uses the first value of a message to set the position of a cube:

```
(osc-source "6543")
(every-frame
    (with-state
        (when (osc-msg "/zzz")
            (translate (vector 0 0 (osc 0))))
        (draw-cube)))
```

And this is a pd patch which can be used to control the cube's position.

```
#N canvas 618 417 286 266 10;
#X obj 58 161 sendOSC;
#X msg 73 135 connect localhost 6543;
#X msg 58 82 send /zzz \$1;
#X floatatom 58 29 5 0 0 0 - - -;
#X obj 58 54 / 100;
#X obj 73 110 loadbang;
#X connect 1 0 0 0;
#X connect 2 0 0 0;
#X connect 3 0 4 0;
#X connect 4 0 2 0;
#X connect 5 0 1 0;
```

Time

I've decided to include time as a source of input, as you could kind of say it comes from the outside world. It's also a useful way of animating things, and making animation more reliable.

```
(time)
```

This returns the time in seconds since the program was started. It returns a float number and is updated once per frame.

```
(delta)
```

This command returns the time since the last frame. (delta) is a very important command, as it allows you to make your animation frame rate independent. Consider the following script:

```
(clear)
(define my-cube (build-cube))
(every-frame
       (with-primitive my-cube
              (rotate (vector 5 0 0))))
```

This rotates my-cube by one degree each frame. The problem is that it will speed up as the framerate rises, and slow down as it falls. If you run the script on a different computer it will move at a different rate. This used to be a problem when running old computer games - they become unplayable as they run too fast on newer hardware!

The solution is to use (delta):

```
(clear)
(define my-cube (build-cube))
(every-frame
        (with-primitive my-cube
                (rotate (vector (* 45 (delta)) 0 0))))
```

The cube will rotate at the same speed everywhere - 45 degrees per second.

The normal (time) command is not too useful on it's own - as it returns an ever increasing number, it doesn't mean much for animation. A common way to tame it is pass it through (sin) to get a sine wave:

```
(clear)
(define my-cube (build-cube))
(every-frame
        (with-primitive my-cube
                (rotate (vector (* 45 (sin (time))) 0 0))))
```

This is also framerate independent and gives you a value from -1 to 1 to play with. You can also record the time when the script starts to sequence events to happen:

```
(clear)
(define my-cube (build-cube))
(define start-time (time)) ; record the time now
(every-frame
        (when (> (- (time) start-time) 5) ; when the script has been running for 5 seconds...
                (with-primitive my-cube
                        (rotate (vector (* 45 (delta)) 0 0)))))
```

Material properties

Now you can create and light some primitives, we can also look more at changing their appearance, other than simply setting their colour.

The surface parameters can be treated exactly like the rest of the local state items like transforms, in that they can either be set using the state stack when building primitives, or set and changed later using (with-primitive).

Line and point drawing modifiers (widths are in pixels):

```
(wire-colour n)
(line-width n)
(point-width n)
```

Lighting modifiers:

```
(specular v)
(ambient v)
(emissive v)
(shinyness n)
```

Opacity:

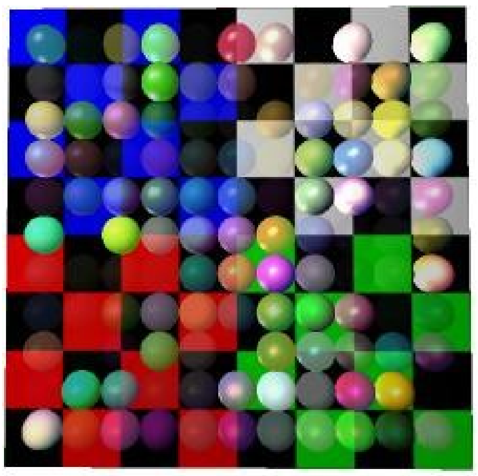


Illustration 2: Some spheres with random materials over a textured plane

```
(opacity n)
(blend-mode src dest)
```

The opacity command causes primitives to be semi transparent (along with textures with an alpha component). Alpha transparency brings with it some problems, as it exposes the fact that primitives are drawn in a certain order, which changes the final result where opacity is concerned. This causes flicker and missing bits of objects behind transparent surfaces, and is a common problem in realtime rendering.

The solution is usually either to apply the render hint (hint-ignore-depth) which causes primitives to render the same way every frame regardless of what's in front of them, or (hint-depth-sort) which causes primitives with this hint to be sorted prior to rendering every frame, so they render from the furthest to the closest order. The depth is taken from the origin of the primitive, you can view this with (hint-origin).

Texturing

Texturing is a part of the local material state, but is a complex topic worthy of a chapter in it's own right.

Loading textures

Getting a texture loaded and applied to a primitive is pretty simple:

```
(with-state
        (texture (load-texture “test.png”))
        (build-cube))
```

Which applies the standard fluxus test texture to a cube. Fluxus reads png files to use as its textures, and they can be with or without an alpha channel.

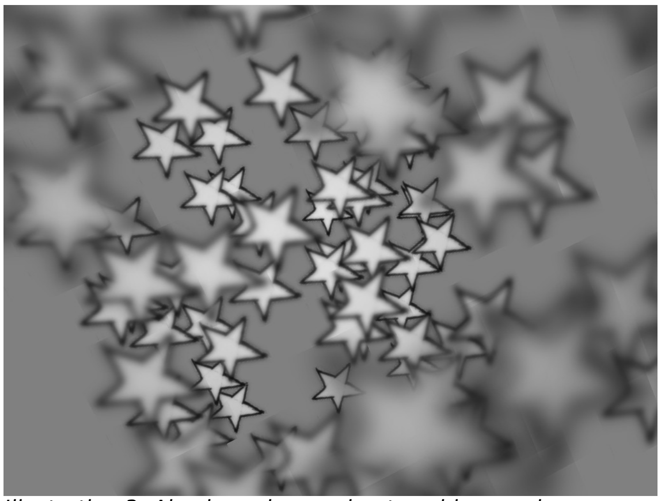


Illustration 3: Abusing mipmapping to achieve a cheap depth of field effect

The texture loading is memory cached. This means that you can call (load-texture) with the same textures as much as you want, Fluxus will only actually load them from disk the first time. This is great until you change the texture yourself, i.e. if you save over the file from the Gimp and re-run the script, the texture won't change in your scene. To get around this, use:

```
(clear-texture-cache)
```

Which will force a reload when (load-texture) is next called on any texture. It's sometimes useful to put this at the top of your script when you are going through the process of changing textures - hitting F5 will cause them to be reloaded.

Texture coordinates

In order to apply a texture to the surface of a primitive, texture coordinates are needed - which tell the renderer what parts of the shape are covered with what parts of the texture. Texture coordinates consist of two values, which range from 0 to 1 as they cross the texture pixels in the x or y direction (by convention these are called "s" and "t" coordinates). What happens when texture coordinates go outside of this range can be changed by you, using (texture-params) but by default the texture is repeated.

The polygon and NURBS shapes fluxus provides you with are all given a default set of coordinates. These coordinates are part of the pdata for the primitive, which are covered in more detail later on. The texture coordinate pdata array is called "t", a simple example which will magnify the texture by a factor of two:

```
(pdata-map!
        (lambda (t)
                (vmul t 0.5)) ; shrinking the texture coordinates zooms into the texture
        "t")
```

Texture parameters

There are plenty of extra parameters you can use to change the way a texture is applied.

```
(texture-params texture-unit-number param-list)
```

We'll cover the texture unit number in multitexturing below, but the param list can look something like this:

```
(texture-params 0 '(min nearest mag nearest)) ; super aliased & blocky texture :)
```

The best approach is to have a play and see what these do for yourself, but here are the parameters in full:

tex-env : one of [modulate decal blend replace]

Changes the way a texture is applied to the colour of the existing surface

min : [nearest linear nearest-mipmap-nearest linear-mipmap-nearest linear-mipmap-linear]

mag : [nearest linear]

These set the filter types which deal with "minification" when the texture's pixels (called texels) are smaller than the screen pixels being rendered to. This defaults to linear-mipmap-linear. Also "magnification", when texels are larger than the screen pixels. This defaults to linear which blends the colour of the texels. Setting this to nearest means you can clearly see the pixels making up the texture when it's large in screen space.

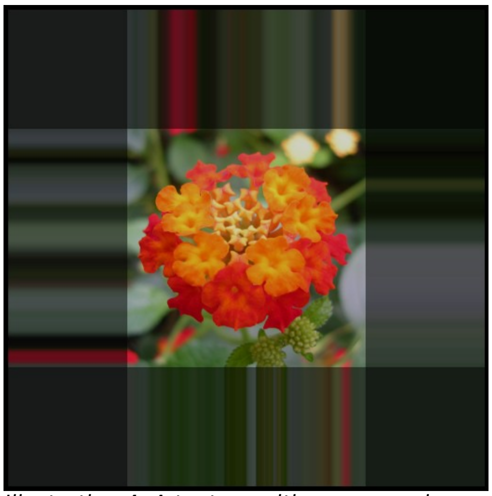


Illustration 4: A texture with wrap-s and wrap-t set to clamp

wrap-s : [clamp repeat]

wrap-t : [clamp repeat]

wrap-r : [clamp repeat] (for cube maps)

What to do when texture coordinates are out of the range 0-1, set to repeat by default, clamp "smears" the edge pixels of the texture.

border-colour : (vector of length 4)

If the texture is set to have a border (see load-texture) this is the colour it should have.

priority : 0 -> 1

I think this controls how likely a texture is to be removed from the graphics card when memory is low. I've never used this myself though.

env-colour : (vector of length 4)

The base colour to blend the texture with.

min-lod : real number (for mipmap blending - default -1000)

max-lod : real number (for mipmap blending - default 1000)

These set how the mipmapping happens. I haven't had much luck setting this on most graphics cards however.

This is the code which generated the clamp picture above:

```
(clear)
(texture-params 0 '(wrap-s clamp wrap-t clamp))
(texture (load-texture "refmap.png"))
(with-primitive (build-cube)
    (pdata-map!
        (lambda (t)
            (vadd (vector -0.5 -0.5 0) (vmul t 2)))
    "t"))
```

Multitexturing

Fluxus allows you to apply more than one texture to a primitive at once. You can combine textures together and set them to multiply, add or alpha blend together in different ways. This is used in games mainly as a way of using texture memory more efficiently, by separating light maps from diffuse colour maps, or applying detail maps which repeat over the top

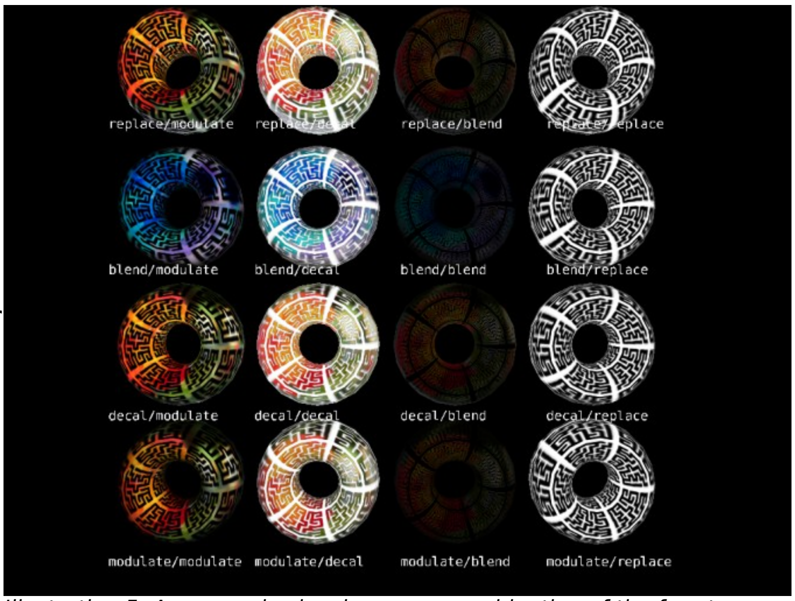


Illustration 5: An example showing every combination of the four types of texture environment blends with two textures

of lower resolution colour maps.

You have 8 slots to put textures in, and set them with:

```
(multitexture texture-unit-number texture)
```

The default texture unit is 0, so:

```
(multitexture 0 (load-texture "test.png"))
```

Is exactly the same as

```
(texture (load-texture "test.png"))
```

This is a simple multitexturing example:

```
(clear)
(with-primitive (build-torus 1 2 20 20)
    (multitexture 0 (load-texture "test.png"))
    (multitexture 1 (load-texture "refmap.png")))
```

By default, all the textures share the "t" pdata as their texture coordinates, but each texture also looks for it's own coordinates to use in preference. These are named "t1", "t2", "t3" and so on up to "t7".

```
(clear)
(with-primitive (build-torus 1 2 20 20)
    (multitexture 0 (load-texture "test.png"))
    (multitexture 1 (load-texture "refmap.png")))
    (pdata-copy "t" "t1") ; make a copy of the existing texture coords
    (pdata-map!
        (lambda (t1)
            (vmul t1 2)) ; make the refmap.png texture smaller than test.png
        "t1"))
```

This is where multitexturing really comes into it's own, as you can move and deform textures individually over the surface of a primitive. One use for this is applying textures like stickers over the top of a background texture, another is to use a separate alpha cut out texture from a colour texture, and have the colour texture "swimming" while the cut out stays still.

Mipmapping

There is more to (load-texture) than we have explained so far. It can also take a list of parameters to change how a texture is generated. By default when a texture is loaded, a set of mipmaps are generated for it - mipmaps are smaller versions of the texture to use when the object is further away from the camera, and are precalculated in order to speed up rendering.

One of the common things you may want to do is turn off mipmapping, as the blurriness can be a problem sometimes.

```
(texture (load-texture "refmap.png"
       '(generate-mipmaps 0 mip-level 0))) ; don't make mipmaps, and send to the top mip level
(texture-params 0 '(min linear)) ; turn off mipmap blending
```

Another interesting texture trick is to supply your own mipmap levels:

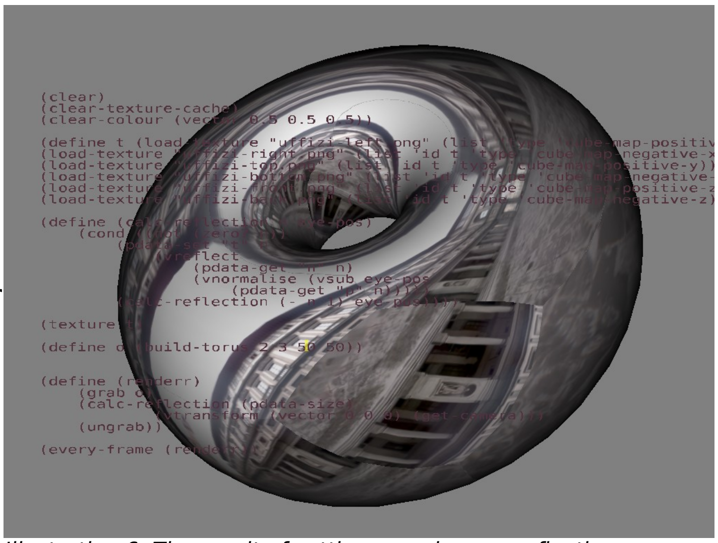
```
 ; setup a mipmapped texture with our own images
 ; you need as many levels as it takes you to get to 1X1 pixels from your
 ; level 0 texture size
 (define t2 (load-texture "m0.png" (list 'generate-mipmaps 0 'mip-level 0)))
```

```
(load-texture "m1.png" (list 'id t2 'generate-mipmaps 0 'mip-level 1))
(load-texture "m2.png" (list 'id t2 'generate-mipmaps 0 'mip-level 2))
(load-texture "m3.png" (list 'id t2 'generate-mipmaps 0 'mip-level 3))
```

This shows how you can load multiple images into one texture, and is how the depth of field effect was achieved in the image above. You can also use a GLSL shader to select mip levels according to some other source.

Cubemapping

Cubemapping is set up in a similar way to mipmapping, but is used for somewhat different reasons. Cubemapping is an approximation of an effect where a surface reflects the environment around it, which in this case is described by six textures representing the 6 sides of a cube around the object.



```
(clear)
(clear-texture-cache)
(clear-colour (vector 0.5 0.5 0.5))

(define t (load-texture "uffizi-left.png" (list 'type 'cube-map-positiv
(load-texture "uffizi-right.png" (list 'id t 'type 'cube-map-negative-x
(load-texture "uffizi-top.png" (list 'id t 'type 'cube-map-positive-y))
(load-texture "uffizi-bottom.png" (list 'id t 'type 'cube-map-negative-
(load-texture "uffizi-front.png" (list 'id t 'type 'cube-map-positive-z
(load-texture "uffizi-back.png" (list 'id t 'type 'cube-map-negative-z)

(define (calc-reflection n eye-pos)
    (cond ((not (zero? n))
        (pdata-set "t" n
            (vreflect
                (pdata-get "n" n)
                (vnormalise (vsub eye-pos
                    (pdata-get "p" n)))))
        (calc-reflection (- n 1) eye-pos))))

(texture t)

(define o (build-torus 2 3 50 50))

(define (renderr)
    (grab o)
    (calc-reflection (pdata-size)
        (vtransform (vector 0 0 0) (get-camera)))
    (ungrab))

(every-frame (renderr))
```

Illustration 6: The result of setting an cubemap reflection on a torus

```
(define t (load-texture "cube-left.png" (list 'type 'cube-map-positive-x)))
(load-texture "cube-right.png" (list 'id t 'type 'cube-map-negative-x))
(load-texture "cube-top.png" (list 'id t 'type 'cube-map-positive-y))
(load-texture "cube-bottom.png" (list 'id t 'type 'cube-map-negative-y))
(load-texture "cube-front.png" (list 'id t 'type 'cube-map-positive-z))
(load-texture "cube-back.png" (list 'id t 'type 'cube-map-negative-z))
(texture t)
```

Lighting

We have got away so far with ignoring the important topic of lighting. This is because there is a default light provided in fluxus, which is pure white and is attached to the camera, so it always allows you to see primitives clearly. However, this is a very boring setup, and if you configure your lighting yourself, you can use lights in very interesting and creative ways, so it's good to experiment with what is possible.

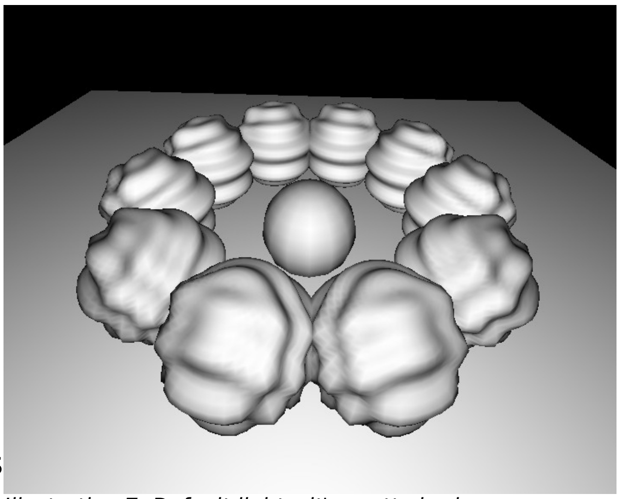


Illustration 7: Default light – it's pretty boring

The standard approach in computer graphics comes from photography, and is referred to as 3 point lighting. The three lights you need are:

- Key – the key light is the main light for illuminating the subject. Put this on the same side of the subject as the camera, but off to one side.

- Fill – the fill light removes hard shadows from the fill light and provides some diffuse lighting. Put this on the same side as the camera, but to the opposite side to the key light.
- Rim or Back light, to separate the subject from the background. Put this behind the subject, to one side to illuminate the edges of the subject.

Here is an example fluxus script to set up such a light arrangement:

```
; light zero is the default camera light - set to a low level
(light-diffuse 0 (vector 0 0 0))
(light-specular 0 (vector 0 0 0))

; make a big fat key light
(define key (make-light 'spot 'free))
(light-position key (vector 5 5 0))
(light-diffuse key (vector 1 0.95 0.8))
(light-specular key (vector 0.6 0.3 0.1))
(light-spot-angle key 22)
(light-spot-exponent key 100)
(light-direction key (vector -1 -1 0))

; make a fill light
(define fill (make-light 'spot 'free))
(light-position fill (vector -7 7 12))
(light-diffuse fill (vector 0.5 0.3 0.1))
(light-specular fill (vector 0.5 0.3 0.05))
(light-spot-angle fill 12)
(light-spot-exponent fill 100)
(light-direction fill (vector 0.6 -0.6 -1))

; make a rim light
(define rim (make-light 'spot 'free))
(light-position rim (vector 0.5 7 -12))
(light-diffuse rim (vector 0 0.3 0.5))
(light-specular rim (vector 0.4 0.6 1))
(light-spot-angle rim 12)
(light-spot-exponent rim 100)
(light-direction rim (vector 0 -0.6 1))
```

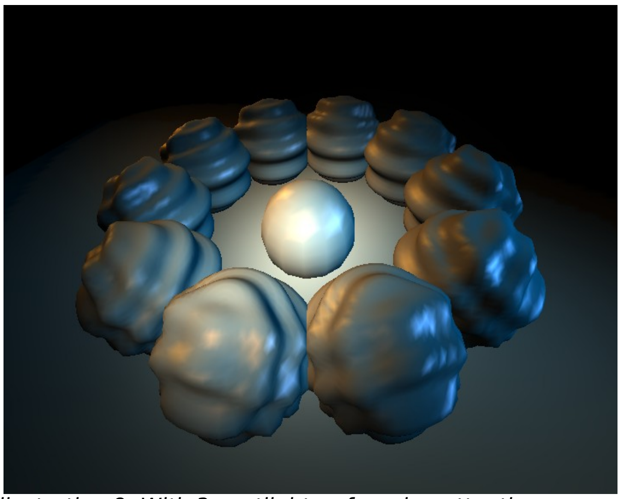


Illustration 8: With 3 spotlights – focusing attention on the sphere in the centre of the scene.

Shadows

The lack of shadows is a big problem with computer graphics. They are complex and time consuming to generate, but add so much to a feeling of depth and form.

However, they are quite easy to add to a fluxus scene:

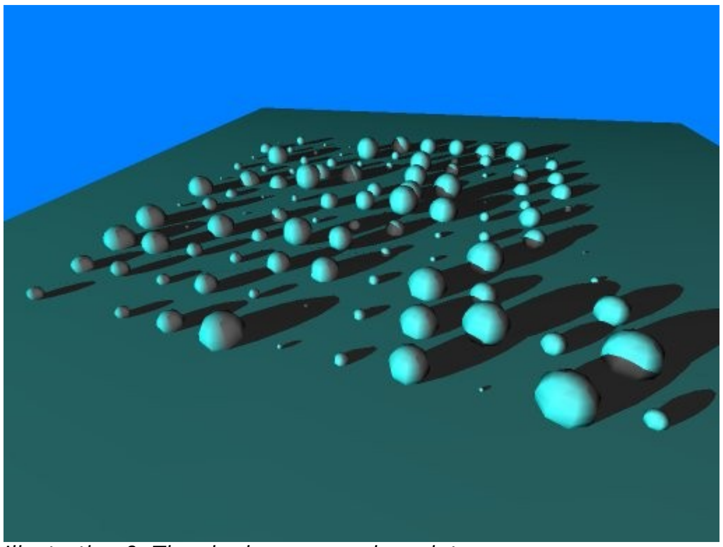


Illustration 9: The shadows example script

```
(clear)
(light-diffuse 0 (vector 0 0 0)) ; turn the main light off
(define l (make-light 'point 'free)) ; make a new light
(light-position l (vector 10 50 50)) ; move it away
(light-diffuse l (vector 1 1 1))
(shadow-light l) ; register it as the shadow light

(with-state
    (translate (vector 0 2 0))
    (hint-cast-shadow) ; tell this primitive to cast shadows
    (build-torus 0.1 1 10 10))

(with-state ; make something for the shadow to fall onto.
    (rotate (vector 90 0 0))
    (scale 10)
    (build-plane))
```

Problems with shadows

There are some catches with the use of shadows in fluxus. Firstly they require some extra data to be generated for the primitives that cast shadows. This can be time consuming to calculate for complex meshes - although it will only be calculated the first time a primitive is rendered (usually the first frame).

A more problematic issue is a side effect of the shadowing method fluxus uses, which is fast and accurate, but won't work if the camera itself is inside a shadow volume. You'll see shadows disappearing or inverting. It can take some careful scene set up to avoid this happening.

A common alternatives to calculated shadows (or lighting in general) is to paint them into textures, this is a much better approach if the light and objects are to remain fixed.

Render hints

We've already come across some render hints, but to explain them properly, they are miscellaneous options that can change the way a primitive is rendered. These options are called hints, as for some types of primitives they may not apply, or may do different things. They are useful for debugging, and sometimes just for fun.

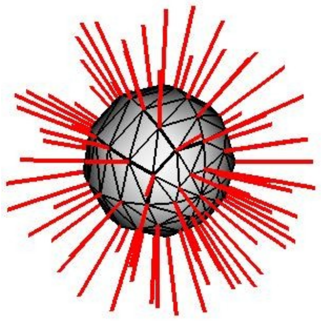


Illustration 10: A poly sphere with normals, wire frame and solid hints, and line width set to4 pixels

```
(hint-none)
```

This is a very important render hint – it clears all other hints. By default only one is set – hint-solid. If you want to turn this off, and just render a cube in wire frame (for instance) you'd need to do this:

```
(hint-none)
(hint-wire)
(build-cube)
```

For a full list of the render hints, see the function reference section. You can do some pretty crazy things, for instance:

```
(clear)
(light-diffuse 0 (vector 0 0 0))
(define l (make-light 'point 'free))
(light-position l (vector 10 50 50))
(light-diffuse l (vector 1 1 1))
(shadow-light l)

(with-state
    (hint-none)
    (hint-wire)
    (hint-normal)
    (translate (vector 0 2 0))
    (hint-cast-shadow)
    (build-torus 0.1 1 10 10))

(with-state
    (rotate (vector 90 0 0))
    (scale 10)
    (build-plane))
```

Draws the torus from the shadowing example but is rendered in wireframe, displaying it's normals as well as casting shadows. It's become something of a fashion to find ways of using normals rendering creatively in fluxus!

About Primitives

Primitives are objects that you can render. There isn’t really much else in a fluxus scene, except lights, a camera and lots of primitives.

Primitive state

The normal way to create a primitive is to set up some state which the primitive will use, then call it’s build function and keep it’s returned ID (using with-primitive) to modify it’s state later on.

```
(define myobj (with-state
               (colour (vector 0 1 0))
               (build-cube))) ; makes a green cube

(with-primitive myobj
       (colour (vector 1 0 0))) ; changes it’s colour to red
```

So primitives contain a state which describes things like colour, texture and transform information. This state operates on the primitive as a whole – one colour for the whole thing, one texture, shader pair and one transform. To get a little deeper and do more we need to introduce primitive data.

Primitive Data Arrays [aka. Pdata]

A pdata array is a fixed size array of information contained within a primitive. Each pdata array has a name, so you can refer to it, and a primitive may contain lots of different pdata arrays (which are all the same size). Pdata arrays are typed – and can contain floats, vectors, colours or matrices. You can make your own pdata arrays, with names that you choose, or copy them in one command.

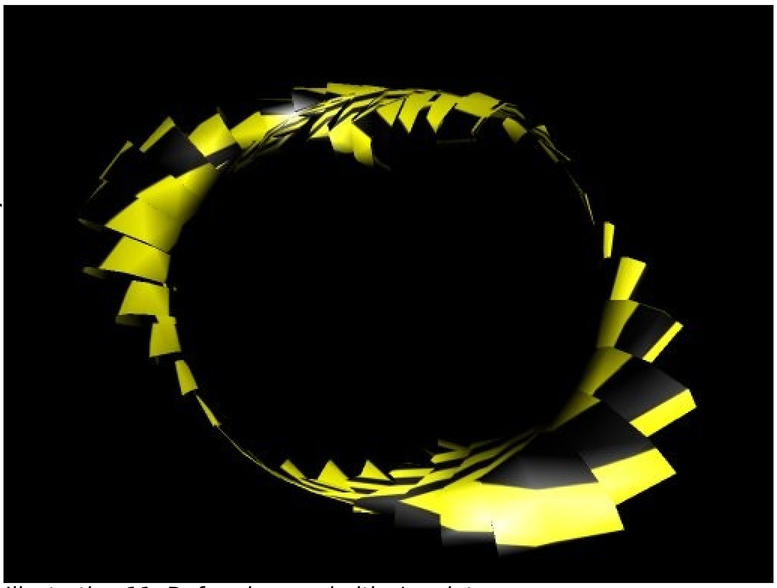


Illustration 11: Deforming a primitive's pdata

Some pdata is created when you call the build function. This automatically generated pdata is given single character names. Sometimes this automatically created pdata results in a primitive you can use straight away (in commands such as build-cube) but some primitives are only useful if pdata is setup and controlled by you.

In polygons, there is one pdata element per vertex – and a separate array for vertex positions, normals, colours and texture coordinates.

So, for example (build-sphere) creates a polygonal object with a spherical distribution of vertex point data, surface normals at every vertex and texture coordinates, so you can wrap a texture around the primitive. This data (primitive data, or pdata for short) can be read and written to inside a with-primitive corresponding to the current object.

```
(pdata-set! Name vertnumber vector)
```

Sets the data on the current object to the input vector

```
(pdata-ref name vertnumber)
```

Returns the vector from the pdata on the current object

```
(pdata-size)
```

Returns the size of the pdata on the current object (the number of verts)

The name describes the data we want to access, for instance “p” contains the vertex positions:

```
(pdata-set! “p” 0 (vector 0 0 0))
```

Sets the first point in the primitive to the origin (not all that useful)

```
(pdata-set! “p” 0 (vadd (pdata-ref “p” 0) (vector 1 0 0)))
```

The same, but sets it to the original position + 1 in the x offsetting the position is more useful as it constitutes a deformation of the original point. (See Deforming, for more info on deformations)

Mapping, Folding

The pdata-set! And pdata-ref procedures are useful, but there is a more powerful way of deforming primitives. Map and fold relate to the scheme functions for list processing, it’s probably a good idea to play with them to get a good understanding of what these are doing.

```
(pdata-map! Procedure read/write-pdata-name read-pdata-name ...)
```

Maps over pdata arrays – think of it as a for-every pdata element, and writes the result of procedure into the first pdata name array.

An example, using pdata-map to invert normals on a primitive:

```
(define p (build-sphere 10 10))


(with-primitive p
(pdata-map!
        (lambda (n)
                (vmul n -1))
        "n"))
```

This is more concise and less error prone than using the previous functions and setting up the loop yourself.

```
(pdata-index-map! Procedure read/write-pdata-name read-pdata-name ...)
```

Same as pdata-map! But also supplies the current pdata index number to the procedure as the first argument.

```
(pdata-fold procedure start-value read-pdata-name read-pdata-name ...)
```

This example calculates the centre of the primitive, by averaging all it’s vertex positions together:

```
(define my-torus (build-torus 1 2 10 10))
(define torus-centre
        (with-primitive my-torus
            (vdiv (pdata-fold vadd (vector 0 0 0) “p”) (pdata-size)))))
```

```
(pdata-index-fold procedure start-value read-pdata-name read-pdata-name ...)
```

Same as pdata-fold but also supplies the current pdata index number to the procedure as the first argument.

Instancing

Sometimes retained mode primitives can be unwieldy to deal with. For instance, if you are rendering thousands of identical objects, or doing things with recursive graphics, where you are calling the same primitive in lots of different states – keeping track of all the Ids would be annoying to say the least.

This is where instancing is helpful, all you call is:

```
(draw-instance myobj)
```

Will redraw any given object in the current state (immediate mode). An example:

```
(define myobj (build-nurbs-sphere 8 10)) ; make a sphere

(define (render-spheres n)
       (cond ((not (zero? n))
              (with-state
              (translate (vector n 0 0)) ; move in x
              (draw-instance myobj))     ; stamp down a copy
              (render-spheres (- n 1)))))    ; recurse!

(every-frame (render-spheres 10)) ; draw 10 copies
```

Built In Immediate Mode Primitives

To make life even easier than having to instance primitives, there are some built in primitives that can be rendered at any time, without being built:

```
(draw-cube)
(draw-sphere)
(draw-plane)
(draw-cylinder)
```

For example:

```
(define (render-spheres n)
       (cond ((not (zero? n))
              (with-state
                     (translate (vector n 0 0)) ; move in x
                     (draw-sphere))           ; render a new sphere
              (render-spheres (- n 1)))))    ; recurse!

(every-frame (render-spheres 10)) ; draw 10 copies
```

These built in primitives are very restricted in that you can't edit them or change their resolution settings etc, but they are handy to use for quick scripts with simple shapes.

Primitive types

Primitives are the most interesting things in fluxus, as they are the things which actually get rendered and make up the scene.

Polygon Primitive

Polygon primitives are the most versatile primitives, as fluxus is really designed around them. The other primitives are added in order to achieve special effects and have specific roles. The other primitive types often have commands to convert them into polygon primitives, for further processing.

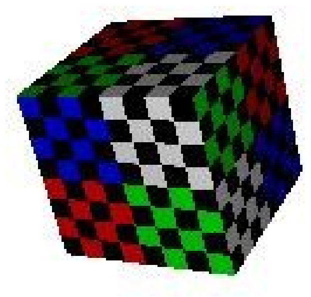


Illustration 12: A poly cube, textured with test.png

There are many commands which create polygon primitives:

```
(build-cube)
```

```
(build-sphere 10 10)
(build-torus 1 2 10 10)
(build-plane)
(build-seg-plane 10 10)
(build-cylinder 10 10)
(build-polygons 100 'triangles)
```

The last one is useful if you are building your own shapes from scratch, the others are useful for giving you some preset shapes. All build-* functions return an id number which you can store and use to modify the primitive later on.

Pdata Types

The pdata arrays available for polygons are as follows:

Use	**Name**	**Data Type**
Vertex position	p	3 vector
Vertex normal	n	3 vector
Vertex texture coords	t	3 vector for u and v, the 3rd number is ignored
Vertex colours	c	4 vector rgba

Polygon topology and pdata

With polygonal objects, we need to connect the vertices defined by the pdata into faces that describe a surface. The topology of the polygon primitive sets how this happens:

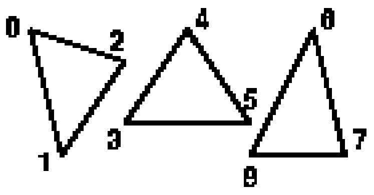


Illustration 14: triangle-list topology

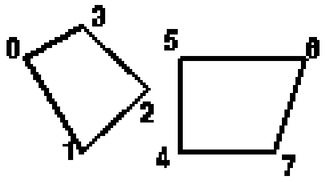


Illustration 13: quad-list topology

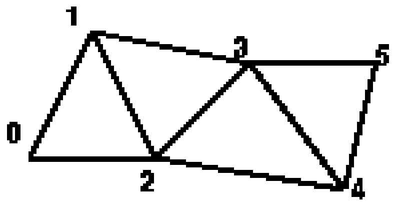


Illustration 15: triangle-strip topology

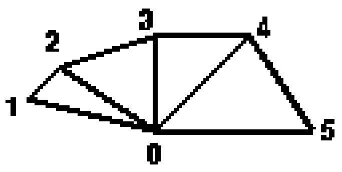


Illustration 16: triangle-fan topology

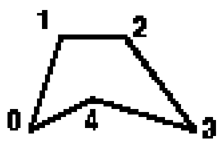


Illustration 17: polygon topology

This lookup is the same for all the pdata on a particular polygon primitive – vertex positions, normals, colours and texture coordinates.

Although using these topologies means the primitive is optimised to be very quick to render, it costs some memory as points are duplicated – this can also cause a big speed impact if you are doing lots of per-vertex calculation. The most optimal poly topology are triangle strips as they maximise the sharing of vertices, but also see indexed polygons for a potentially even faster method.

You can find out the topology of a polygon primitive like this:

```
(define p (build-cube))

(with-primitive p
    (display (poly-type))(newline)) ; prints out quad-list
```

Indexed polygons

The other way of improving efficiency is converting polygons to indexed mode. Indexing means that vertices in different faces can share the same vertex data. You can set the index of a polygon manually with:

```
(with-primitive myobj
        (poly-set-index list-of-indices))
```

Or automatically – (which is recommended) with:

```
(with-primitive myobj
        (poly-convert-to-indexed))
```

This procedure compresses the polygonal object by finding duplicated or very close vertices and “gluing” them together to form one vertex and multiple index references. Indexing has a number of advantages, one that large models will take less memory – but the big deal is that deformation or other per-vertex calculations will be much quicker.

Problems with automatic indexing

As all vertex information becomes shared for coincident vertices, automatically indexed polygons can’t have different normals, colours or texture coordinates for vertices at the same position. This means that they will have to be smooth and continuous with respect to lighting and texturing. This should be fixed in time, with a more complicated conversion algorithm.

NURBS Primitives

NURBS are parametric curved patch surfaces. They are handled in a similar way to polygon primitives, except that instead of vertices, pdata elements represent control vertices of the patch. Changing a control vertex causes the mesh to smoothly blend the change across it’s surface.

```
(build-nurbs-sphere 10 10)
(build-nurbs-plane 10 10)
```

Illustration 18: A NURBS sphere, with a vertex tweaked

Pdata Types

Use	Name	Data Type

Control vertex position	p	3 vector
Control vertex normal	n	3 vector
Control vertex texture coords	t	3 vector for u and v, the 3rd number is ignored

TODO: NURBS topology

Particle primitives

Particle primitives use the pdata elements to represent a point, or a camera facing sprite which can be textured – depending on the render hints. This primitive is useful for lots of different effects including, water, smoke, clouds and explosions.

```
(build-particles num-particles)
```

Illustration 19: Some particles used as sprites by assigning them a texture

Pdata Types

Use	**Name**	**Data Type**
Particle position	p	3 vector
Partilcle colour	c	3 vector
Particle size	s	3 vector for width and height, the 3rd number is ignored

Geometry hints

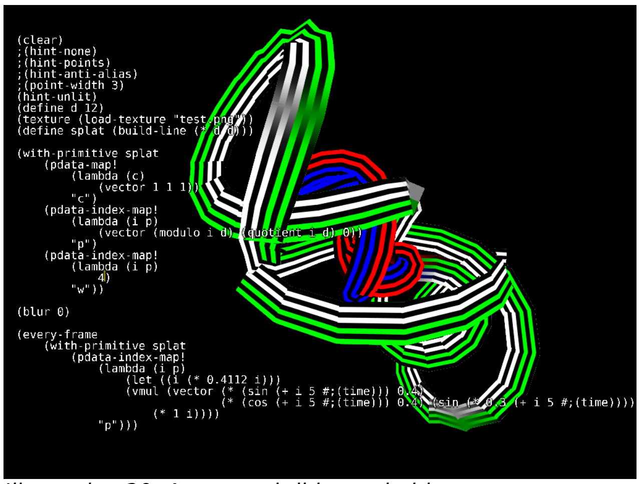
Particle primitives default to camera facing sprites, but can be made to render as points in screen space, which look quite different. To switch to points:

```
(with-primitive myparticles
(hint-none)    ; turns off solid, which is default sprite mode
(hint-points)) ; turn on points rendering
```

If you also enable (hint-anti-alias) you may get circular points, depending on your GPU – these can be scaled in pixel space using (point-width).

Ribbon primitives

Ribbon primitives are similar to particles in that they are either hardware rendered lines or camera facing textured quads. This primitive draws a single line connecting each



```
(clear)
;(hint-none)
;(hint-points)
;(hint-anti-alias)
;(point-width 3)
(hint-unlit)
(define d 12)
(texture (load-texture "test.png"))
(define splat (build-line (* d d)))

(with-primitive splat
    (pdata-map!
        (lambda (c)
            (vector 1 1 1))
        "c")
    (pdata-index-map!
        (lambda (i p)
            (vector (modulo i d) (quotient i d) 0))
        "p")
    (pdata-index-map!
        (lambda (i p)
            4)
        "w"))

(blur 0)

(every-frame
    (with-primitive splat
        (pdata-index-map!
            (lambda (i p)
                (let ((i (* 0.4112 i)))
                (vmul (vector (* (sin (+ i 5 #;(time))) 0.4)
                              (* (cos (+ i 5 #;(time))) 0.4) (sin (* 0.3 (+ i 5 #;(time))))
                    (* 1 i))))
            "p")))
```

Illustration 20: A textured ribbon primitive pdata vertex element together. The texture is stretched along the ribbon from the start to the end, and width wise across the line. The width can be set per vertex to change the shape of the line.

```
(build-line num-points)
```

Pdata Types

Use	Name	Data Type
Ribbon vertex position	p	3 vector
Ribbon vertex colour	c	3 vector
Ribbon vertex width	w	number

Geometry hints

Ribbon primitives default to render camera facing quads. You can also switch them to draw wire frame lines in screen space:

```
(with-primitive myline
        (hint-none) ; turns off solid, which is default mode
        (hint-wire)) ; turn on lines rendering
```

The wire frame lines can be scaled in pixel space using (line-width). The lines also pick up the colour pdata information.

Text primitive

Text primitives allow you to create text based on texture fonts. The font assumed to be non proportional – there is an example font shipped with fluxus.

```
(texture (load-texture “font.png”))
(build-text text-string)
```

The primitive makes a set of quads, one for each character, with texture coordinates set to show the correct character from the texture. This provides a fast and cheap way of displaying text, and is useful for debugging, or if the text is quite small. You can probably also find quite creative ways of using this just as a way of chopping up a texture into little squares.

Type Primitive

The type primitive provides a much higher quality typeface rendering than the text primitive. It creates polygonal geometry from a ttf font and some text. Optionally you can also extrude the text which results in 3D

Illustration 21: Boring old helvetica, extruded

shapes.

```
(build-type ttf-font-filename text)
(build-extruded-type ttf-font-filename text extrude depth)
```

You can also convert the type primitive into a polygon primitive for further deformation or applying textures, with:

```
(type->poly type-prim-id)
```

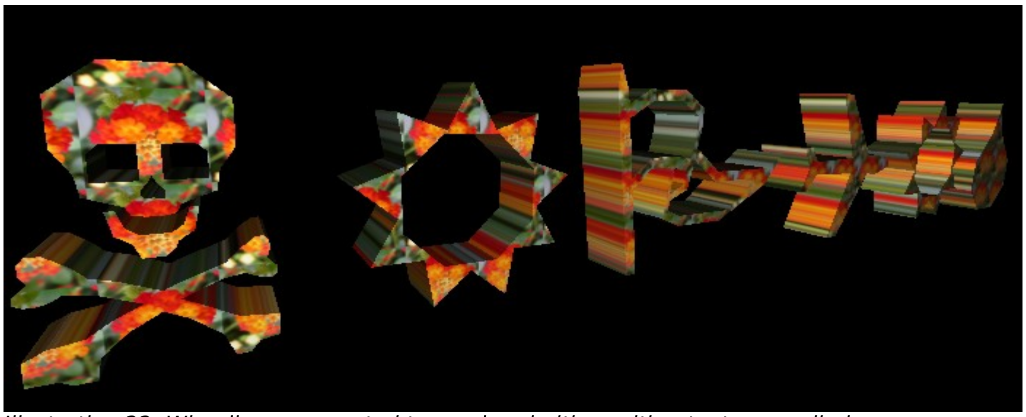


Illustration 22: Wingdings, converted to a poly primitive, with a texture applied

Locator primitive

The locator is a null primitive, as it doesn’t render anything. Locators are useful for various tasks, you can use them as a invisible scene node to group primitive under. They are also used to build skeletons for skinning. To view them, you can turn on hint-origin, which draws an axis representing their transform.

```
(hint-origin)
(build-locator)
```

Pixel primitive

A pixel primitive is used for making procedural textures, which can then be applied to other primitives. For this reason, pixel primitives probably wont be rendered much directly, but you can render them to preview the texture on a flat plane.

```
(pixel-primitive width height)
```

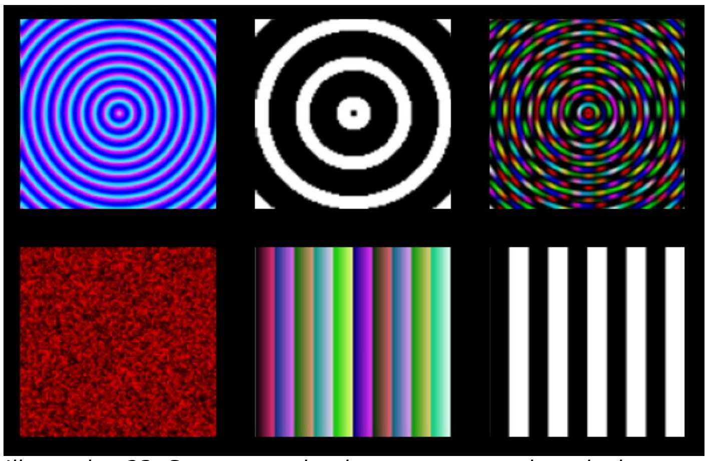


Illustration 23: Some procedural textures created as pixel primitives

Pdata Types

Use	Name	Data Type

Pixel colour	c	3 vector
Pixel alpha	a	number

Extra pixel primitive commands

A pixel primitive’s pdata corresponds to pixel values in a texture, you write to them to make procedural texture data. The pixel primitive comes with some extra commands:

```
(pixels-upload pixelprimitiveid-number)
```

Uploads the texture data, you need to call this when you’ve finished writing to the pixelprim, and while it’s grabbed.

```
(pixels->texture pixelprimitiveid-number)
```

Returns a texture you can use exactly like a normal loaded one.

See the examples for some procedural texturing. It’s important to note that creating textures involves a large amount of processing time, so you don’t want to plan on doing something per-pixel/per-frame for large textures. The pdata-func extensions could be used to help here in the future.

This is a simple example of creating a noise texture on a pixel primitive:

```
(with-primitive (build-pixels 100 100)
        (pdata-map!
            (lambda (colour)
                (rndvec))
            "c")
        (pixels-upload))
```

Blobby Primitives

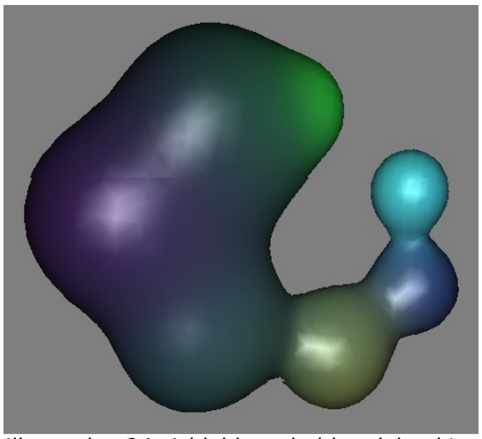


Illustration 24: A blobby primitive doing it's thing

Blobby primitives are a higher level implicit surface representation in fluxus which is defined using influences in space. These influences are summed together, and a particular value is “meshed” (using the marching cubes algorithm) to form a smooth surface. The influences can be animated, and the smooth surface moves and deforms to adapt, giving the primitive it’s blobby name. (build-blobby) returns a new blobby primitive id. Numinfluences is the number of “blobs”. Subdivisions allows you to control the resolution of the surface in each dimension, while boundingvec sets the bounding area of the primitive in local object space. The mesh will not be calculated outside of this area. Influence positions and colours need to be set using pdata-set.

```
(build-blobby numinfluences subdivisionsvec boundingvec)
```

Pdata Types

Use	Name	Data Type
Position	p	3 vector
Strength	s	number
Colour	c	3 vector

Converting to polygons

Blobbies can be slow to calculate, if you only need static meshes without animation, you can convert them into polygon primitives with:

```
(blobby->poly blobby-id-num)
```

Deforming

Deformation in this chapter signifies various operations. It can involve changing the shape of a primitive in a way not possible via a transform (i.e. bending, warping etc) or modifying texture coordinates or colours to achieve a per-vertex effect. Deformation in this way is also the only way to get particle primitives to do anything interesting.

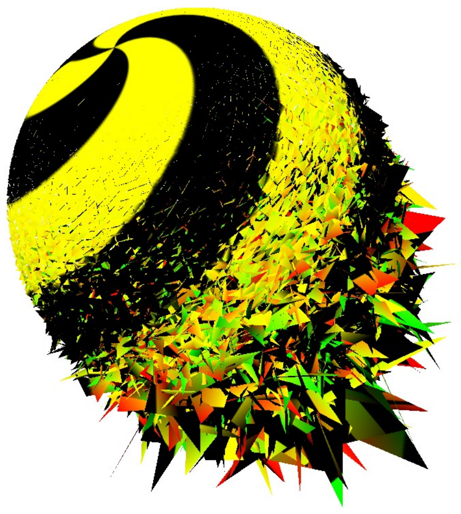


Illustration 25: A sphere being deformed

Deforming is all about pdata, so, to deform an entire object, you do something like the following:

```
(hint-unlit) (hint-wire) (line-width 4)

(define myobj (build-sphere 10 10))

(with-primitive myobj
        (pdata-map!
                (lambda (p)
                        ; add a small random vector to the original point
                        (vadd (vmul (rndvec) 0.1)) p)
             "p")))
```

When deforming geometry, moving the positions of the vertices is not usually enough, the normals will need to be updated for the lighting to work correctly.

```
(recalc-normals smooth)
```

Will regenerate the normals for polygon and nurbs primitives based on the vertex positions. Not particularly fast (it is better to deform the normals in your script if you can). If smooth is 1, the face normals are averaged with the coincident face normals to give a smooth appearance.

When working on polygon primitives fluxus will cache certain results, so it will be a lot slower on the first calculation than subsequent calls on the same primitive.

User Pdata

As well as the standard information that exists in primitives, fluxus also allows you to add your own per vertex data to any primitive. User pdata can be written or read in the same way as the built in pdata types.

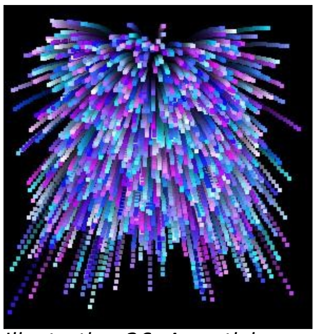


Illustration 26: A particle explosion

```
(pdata-add name type)
```

Where name is a string with the name you wish to call it, and type is a one character string consisting of:

f : float data

v : vector data

c : colour data

m : matrix data

```
(pdata-copy source destination)
```

This will copy a array of pdata, or overwrite an existing one with if it already exists. Adding your own storage for data on primitives means you can use it as a fast way of reading and writing data, even if the data doesn’t directly affect the primitive.

An example of a particle explosion:

```
; setup the scene
(clear)
(show-fps 1)
(point-width 4)
(hint-anti-alias)

; build our particle primitive
(define particles (build-particles 1000))

; set up the particles
(with-primitive particles
        (pdata-add “vel” “v”) ; add the velocity user pdata of type vector
        (pdata-map! ; init the velocities
                (lambda (vel)
                        (vmul (vsub (vector (flxrnd) (flxrnd) (flxrnd))
                                        (vector 0.5 0.5 0.5)) 0.1))
                “vel”)
        (pdata-map! ; init the colours
                (lambda (c)
                        (vector (flxrnd) (flxrnd) 1 0))
              "c"))

(blur 0.1)

; a procedure to animate the particles
```

```
(define (animate)
        (with-primitive particles
                (pdata-map!
                        (lambda (vel)
                                (vadd vel (vector 0 -0.001 0)))
                        “vel”)
                (pdata-map! vadd “p” “vel”)))

(every-frame (animate))
```

Pdata Operations

Pdata Operations are a optimisation which takes advantage of the nature of these storage arrays to allow you to process them with a single call to the scheme interpreter. This makes deforming primitive much faster as looping in the scheme interpreter is slow, and it also simplifies your scheme code.

```
(pdata-op operation pdata operand)
```

Where operation is a string identifier for the intended operation (listed below) and pdata is the name of the target pdata to operate on, and operand is either a single data (a scheme number or vector (length 3,4 or 16)) or a name of another pdata array.

If the (update) and (render) functions in the script above are changed to the following:

```
(define (update)
        ; add this vector to all the velocities
        (pdata-op “+” “vel” (vector 0 -0.002 0))
        ; add all the velocities to all the positions
        (pdata-op “+” “p” “vel”))

(define (render)
        (with-primitive ob
                (update)))
```

On my machine, this script runs over 6 times faster than the first version.

(pdata-op) can also return information to your script from certain functions called on entire pdata arrays.

Pdata operations

“+”   : addition

“*”   : multiplication

“sin”   : writes the sine of one float pdata array into another “cos”   : writes the cosine of one float pdata array into another “closest” : treats the vector pdata as positions, and if given a single vector, returns the closest position to it – or if given a float, uses it as a index into the pdata array, and returns the nearest position.

For most pdata operations, the vast majority of the combinations of input types (scheme number, the vectors or pdata types) will not be supported, you will receive a rather cryptic runtime warning message if this is the case.

Pdata functions

Pdata ops are useful, but I needed to expand the idea into something more complicated to support more interesting deformations like skinning. This area is messy, and somewhat experimental – so bear with me, it should solidify in future.

Pdata functions (pfuncs) range from general purpose to complex and specialised operations which you can run on primitives. All pfuncs share the same interface for controlling and setting them up. The idea is that you make a set of them at startup, then run them on one or many primitives later on per-frame.

```
(make-pfunc pfunc-name-symbol))
```

Makes a new pfunc. Takes the symbol of the names below, e.g. (make-pfunc ‘arithmetic)

```
(pfunc-set! pfuncid-number argument-list)
```

Sets arguments on a primitive function. The argument list consists of symbols and corresponding values.

```
(pfunc-run id-number)
```

Runs a primitive function on the current primitive. Look at the skinning example to see how this works.

Pfunc types

All pfunc types and arguments are as follows:

arithmetic

For applying general arithmetic to any pdata array

operator string : one of: add sub mul div

src string : pdata array name

other string : pdata array name (optional)

constant float : constant value (optional)

dst string : pdata array name

genskinweights

Generates skinweights – adds float pdata called “s1” -> “sn” where n is the number of nodes in the skeleton – 1

skeleton-root primid-number : the root of the bindpose skeleton for skinning
sharpness float : a control of how sharp the creasing will be when skinned

skinweights->vertcols

A utility for visualising skinweights for debugging.

No arguments

skinning

Skins a primitive – deforms it to follow a skeleton’s movements. Primitives we want to run this on have to contain extra pdata – copies of the starting vert positions called “pref” and the same for normals, if normals are being skinned, called “nref”.

Skeleton-root primid-number : the root primitive of the animating skeleton bindpose-root primid-number : the root primitive of the bindpose skeleton skin-normals number : whether to skin the normals as well as the positions

Using Pdata to build your own primitives

The function (build_polygons) allows you to build empty primitives which you can use to either build more types of procedural shapes than fluxus supports naively, or for loading model data from disk. Once these primitives have been constructed they can be treated in exactly the same way as any other primitive, ie pdata can be added or modified, and you can use (recalc-normals) etc.

Cameras

Without a camera you wouldn't be able to see anything! They are obviously very important in 3D computer graphics, and can be used very effectively, just like a real camera to tell a story.

Moving the camera

You can already control the camera with the the mouse, but sometimes you'll want to control the camera procedurally from a script. Animating the camera this way is also easy, you just lock it to a primitive and move that:

```
(clear)
(define obj (build-cube)) ; make a cube for the camera to lock to

(with-state ; make a background cube so we can tell what's happening
    (hint-wire)
    (hint-unlit)
    (texture (load-texture "test.png"))
    (colour (vector 0.5 0.5 0.5))
    (scale (vector -20 -10 -10))
    (build-cube))

(lock-camera obj) ; lock the camera to our first cube
(camera-lag 0.1)  ; set the lag amount, this will smooth out the cube jittery movement
```

```
(define (animate)
    (with-primitive obj
        (identity)
        (translate (vector (fmod (time) 5) 0 0)))) ; make a jittery movement

(every-frame (animate))
```

Stopping the mouse moving the camera

The mouse camera control still works when the camera is locked to a primitive, it just moves it relative to the locked primitive. You can stop this happening by setting the camera transform yourself:

```
(set-camera-transform (mtranslate (vector 0 0 -10)))
```

This command takes a transform matrix (a vector of 16 numbers) which can be generated by the maths commands (mtranslate), (mrotate) and (mscale) and multiplied together with (mmul).

You just need to call (set-camera-transform) once, it gives your script complete control over the camera, and frees the mouse up for doing other things.. To switch mouse movement back on, use:

```
(reset-camera)
```

More camera properties

By default the camera is set to perspective projection. To use orthographic projection just use:

```
(ortho)
```

Moving the camera back and forward has no effect with orthographic projection, so to zoom the display, use:

```
(set-ortho-zoom 10)
```

And use:

```
(persp)
```

To flip the camera back to perspective mode.

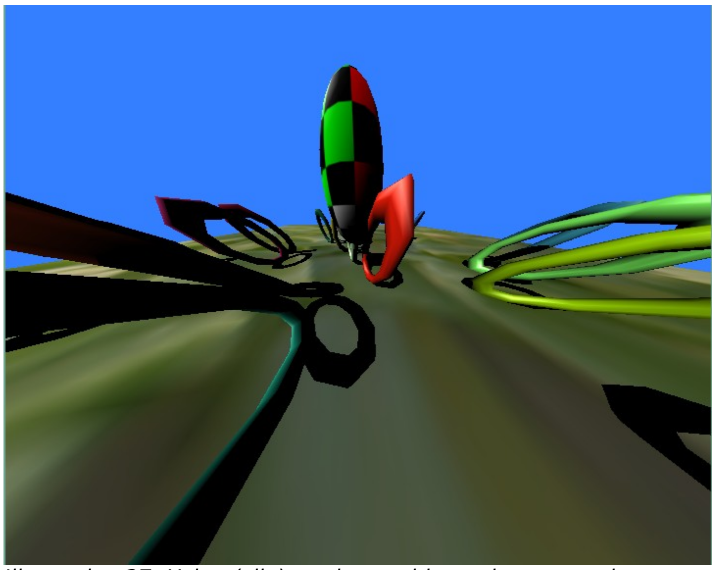


Illustration 27: Using (clip) to give a wide angle perspective on the camera

The camera angle can be changed with the rather confusingly named command:

```
(clip 1 10000)
```

Which sets the near and far clipping plane distance. The far clipping plane (the second number) sets where geometry will start getting dropped out. The first number is interesting to us here as it sets the distance from the camera centre point to the near clipping plane. The smaller this number, the wider the camera angle.

Fogging

Not strictly a camera setting, but fog fades out objects as they move away from the camera - giving the impression of aerial perspective.

```
(fog (vector 0 0 1) 0.01 1 1000)
```

The first number is the colour of the fog, followed by the strength (keep it low) then the start and end of the fogging (which don't appear to work on my graphics card, at least).

Fog used to be used as a way of hiding the back clipping plane, or making it slightly less jarring, so things fade away rather than disappear - however it also adds a lot of realism to outdoor scenes, and I'm a big fan of finding more creative uses for it.

Using multiple cameras

Todo: hiding things in different camera views

Noise and randomness

Noise and randomness is used in computer animation in order to add “dirt” to your work. This is very important, as the computer is clean and boring by default.

Randomness

Scheme has it's own built in set of random operations, but Fluxus comes with a set of it's own to make life a bit easier for you.

Random number operations

These basic random commands return numbers:

```
(rndf) ; returns a number between 0 and 1
(crndf) ; (centred) returns a number between -1 and 1
(grndf) ; returns a Gaussian random number centred on 0 with a variance of 1
```

Random vector operations

More often than not, when writing fluxus scripts you are interested in getting random vectors, in order to perturb or generate random positions in space, directions to move in, or colours.

```
(rndvec) ; returns a vector where the elements are between 0 and 1
(crndvec) ; returns a vector where the elements are between -1 and 1
(srndvec) ; returns a vector represented by a point inside a sphere of radius 1
(hsrndvec) ; a vector represented by a point on the surface of a sphere radius 1 (hollow sphere)
(grndvec) ; a Gaussian position centred on 0,0,0 with variance of 1
```

These are much better described by a picture:

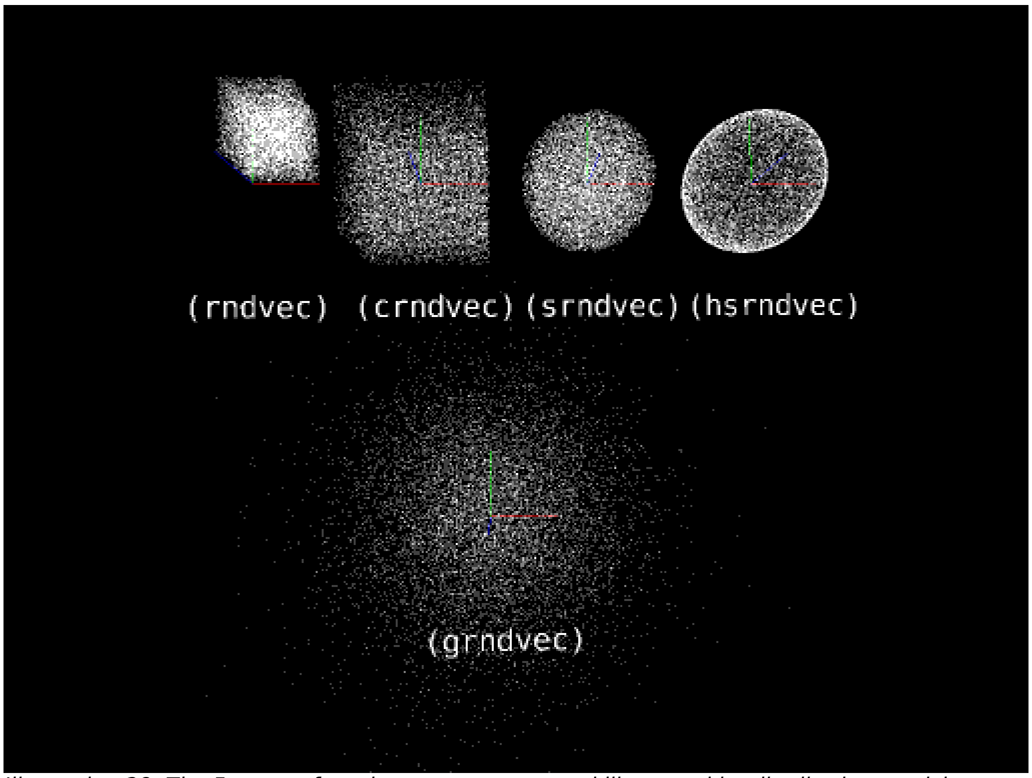


Illustration 28: The 5 types of random vector command illustrated by distributing particles using them

Noise

Todo

Scene Inspection

So far we have mostly looked at describing objects and deformations to fluxus so it can build scenes for you. Another powerful technique is to get fluxus to inspect your scene and give you information on what is there.

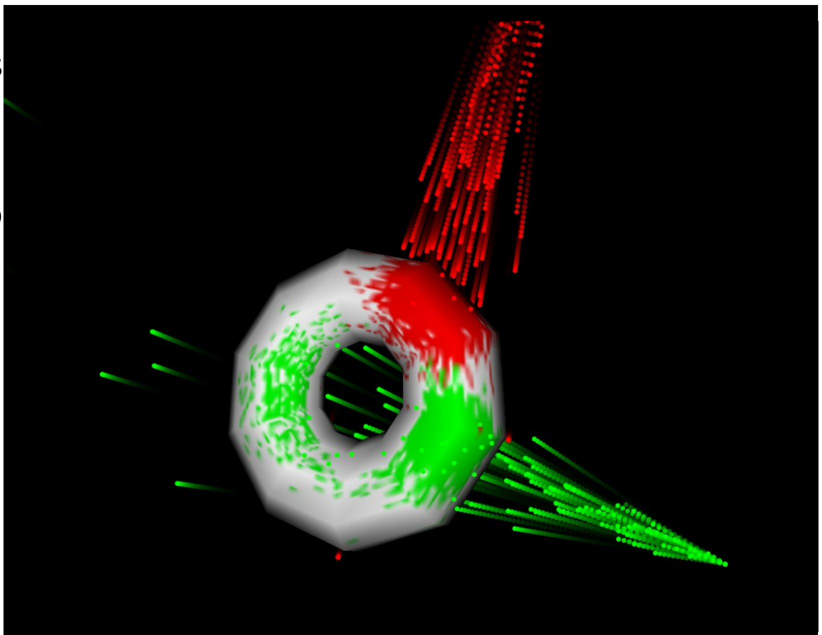


Illustration 29: Procedurally painting a texture using a particle system and ray casting to find collision points and texture coordinates – and then writing to a pixel primitive

Scene graph inspection

The simplest, and perhaps most powerful commands for inspection are these commands:

```
(get-children)
(get-parent)
```

(get-children) returns a list of the children of the current primitive, it can also give you the list of children of the scene root node if you call it outside of (with-primitive). (get-parent) returns the parent of the current primitive.

These commands can be used to navigate the scenegraph and find primitives without you needing to record their Ids manually. For instance, a primitive can change the colour of it's parent like this:

```
(with-primitive myprim
       (with-primitive (get-parent)
              (colour (vector 1 0 0))))
```

You can also visit every primitive in the scene with the following script:

```
; navigate the scene graph and print it out
(define (print-heir children)
    (for-each
        (lambda (child)
            (with-primitive child
                (printf "id: ~a parent: ~a children: ~a~n" child (get-parent) (get-children))
                (print-heir (get-children))))
        children))
```

Collision detection

A very common thing you want to do is find out if two objects collide with each other (particularly when writing games). You can find out if the current primitive roughly intersects another one with this command:

```
(bb-intersect other-primitive box-expand)
```

This uses the automatically generated bounding box for the primitives, and so is quite fast and good enough for most collision detection. The box expand value is a number with which to add to the bounding box to expand or shrink the volume it uses for collision detection.

Note: The bounding box used is not the same one as you see with (hint-box), which is affected by the primitive's transform. bb-intersect generates new bounding boxes which are all axis aligned for speed of comparison.

Ray Casting

Another extremely useful technique is to create rays, or lines in the scene and get information about where on primitives they intersect with. This can be used for detailed collision detection or in more complex techniques like raytracing.

```
(line-intersect line-start-position line-end-position)
```

This command returns a list of pdata values at the points where the line intersects with the current

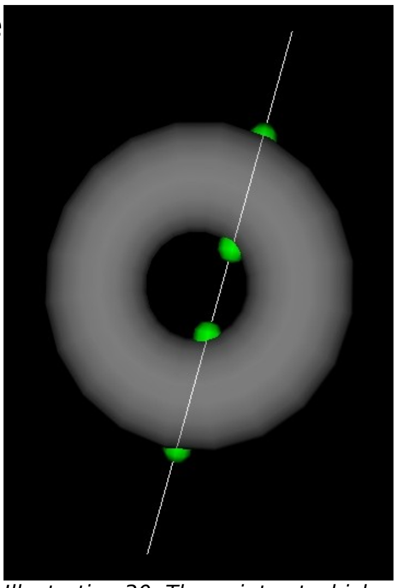


Illustration 30: The points at which a line intersects with a torus

primitive. The clever thing is that it values for the precise intersection point – not just the closest vertex.

The list it returns is designed to be accessed using Scheme (assoc) command. An intersection list looks like this:

```
(collision-point-list collision-point-list ...)
```

Where a collision point list looks like:

```
((p . position-vector)
 (t . texture-vector)
 (n . normal-vector)
 (c . colour-vector))
```

The green sphere in the illustration are positioned on the p pdata positions returned by the following snippet of code:

```
(with-primitive s
     (for-each
          (lambda (intersection)
              (with-state ; draw a sphere at the intersection point
                  (translate (cdr (assoc "p" intersection)))
                  (colour (vector 0 1 0))
                  (scale (vector 0.3 0.3 0.3))
                  (draw-sphere)))
      (line-intersect a b))))
```

Primitive evaluation

Pdata-for-each-face

pdata-for-each-triangle

pdata-for-each-tri-sample

Physics

Primitive loading And Saving

It's useful to be able to load and save primitives, this is for several reasons. You may wish to use other programs to make meshes and import them into fluxus, or to export primitives you've made in fluxus to render them in other programs. It's also useful to save the primitive as a file if the process to create it in fluxus is very slow. There are only two commands you need to know to do this:

```
; load a primitive in:
(define newprim (load-primitive filename))
; save it out again:
(with-primitive newprim
       (save-primitive filename))
```

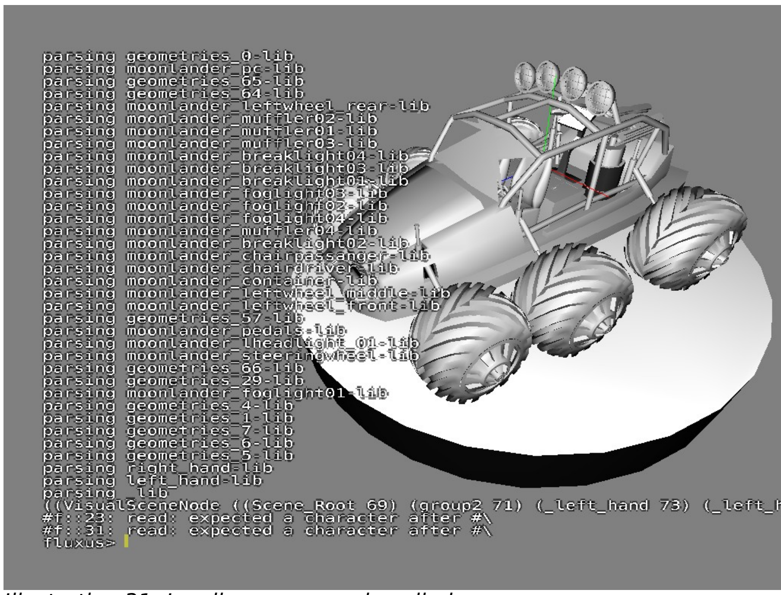
At present these commands work on poly and pixel primitives, and load/save obj and png files respectively.

COLLADA format support

Collada is a standard file format for complex 3D scenes. Collada files can be loaded, currently supported geometry is triangular data, vertex positions, normals and texture coordinates. The plan is to use collada for complex scenes containing different geometry types, including animation and physics data. Collada export is planned.

See the documentation for the command:

```
(collada-import filename)
```



```
parsing geometries_0-lib
parsing moonlander_pc-lib
parsing geometries_65-lib
parsing geometries_64-lib
parsing moonlander_leftwheel_rear-lib
parsing moonlander_muffler02-lib
parsing moonlander_muffler01-lib
parsing moonlander_muffler03-lib
parsing moonlander_breaklight04-lib
parsing moonlander_breaklight03-lib
parsing moonlander_breaklight01-lib
parsing moonlander_foglight03-lib
parsing moonlander_foglight02-lib
parsing moonlander_foglight04-lib
parsing moonlander_muffler04-lib
parsing moonlander_breaklight02-lib
parsing moonlander_chairpassanger-lib
parsing moonlander_chairdriver-lib
parsing moonlander_container-lib
parsing moonlander_leftwheel_middle-lib
parsing moonlander_leftwheel_front-lib
parsing geometries_57-lib
parsing moonlander_pedals-lib
parsing moonlander_lheadlight_01-lib
parsing moonlander_steeringwheel-lib
parsing geometries_66-lib
parsing geometries_29-lib
parsing moonlander_foglight01-lib
parsing geometries_4-lib
parsing geometries_1-lib
parsing geometries_7-lib
parsing geometries_6-lib
parsing geometries_5-lib
parsing right_hand-lib
parsing left_hand-lib
parsing _lib
((VisualSceneNode ((Scene_Root 69) (group2 71) (_left_hand 73) (_left_h
#f::23: read: expected a character after #\
#f::31: read: expected a character after #\
fluxus>
```

Illustration 31: Loading an example collada scene

Shaders

Hardware shaders allow you to have much finer control over the graphics pipeline used to render your objects. Fluxus has commands to set and control GLSL shaders from your scheme scripts, and even edit your shaders in the fluxus editor. GLSL is the OpenGL standard for shaders across various graphics card types, if your card and driver support OpenGL2, this should work for you.

```
(shader vertshader fragshader)
```

Loads, compiles and binds the vertex and fragment shaders on to current state or grabbed primitive.

```
(shader-set! paramlist)
```

Sets uniform parameters for the shader in a token, value list, e.g.:

```
(list “specular” 0.5 “mycolour” (vector 1 0 0))
```

This is very simple to set up – in your GLSL shader you just need to declare a uniform value eg:

```
uniform float deformamount;
```

This is then set by calling from scheme:

```
(shader-set! (list “deformamount” 1.4))
```

The deformamount is set once per object/shader – hence it’s a uniform value across the whole object.

Shaders also get given all pdata as attribute (per vertex) parameters, so you can share all this information between shaders and scripts in a similar way:

In GLSL:

```
attribute vec3 testcol;
```

To pass this from scheme, first create some new pdata with a matching name:

```
(pdata-add “testcol” “v”)
```

Then you can set it in the same way as any other pdata, controlling shader parameters on a per-vertex basis.

Samplers

Samplers are the hardware shading word for textures, the word sampler is used to be a little more general in that they are used as a way of passing lots of information (which may not be visual in nature) around between shaders. Passing textures into GLSL shaders from fluxus is again fairly simple:

In your GLSL shader:

```
uniform sampler2D mytexture;
```

In scheme:

```
(texture (load-texture “mytexturefile.png”))
(shader-set! (list “mytexture” 0))
```

This just tells GLSL to use the first texture unit (0) as the sampler for mytexture. This is the texture unit that the standard (texture) command loads textures to.

To pass more than one texture, you need multitexturing turned on:

In GLSL:

```
uniform sampler2D mytexture;
uniform sampler2D mysecondtexture;
```

In scheme:

```
; load to texture unit 0
(multitexture 0 (load-texture “mytexturefile.png”))
; load to texture unit 1
(multitexture 1 (load-texture “mytexturefile2.png”))
(shader-set! (list “mytexture” 0 “mysecondtexture” 1))
```

Turtle Builder

The turtle polybuilder is an experimental way of building polygonal objects using a logo style turtle in 3D space. As you drive the turtle around you can place vertices and build shapes procedurally. The turtle can also be used to deform existing polygonal primitives, by attaching it to objects you have already created.

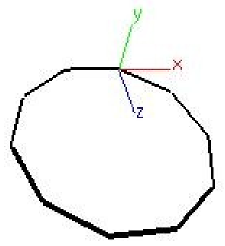


Illustration 32: A circle

This script simply builds a single polygon circle, by playing the age old turtle trick of looping a function that moves a bit, turns a bit...

```
(define (build n)
(turtle-reset)
(turtle-prim 4)
(build-loop n n)
(turtle-build))
```

```
(define (build-loop n t)
       (turtle-turn (vector 0 (/ 360 t) 0))
       (turtle-move 1)
       (turtle-vert)
       (if (< n 1)
              0
              (build-loop (- n 1) t)))


(backfacecull 0)
(clear)
(hint-unlit)
(hint-wire)
(line-width 4)
(build 10)
```

For a more complex example, just modify the (build-loop) function as so:

```
(define (build-loop n t)
       (turtle-turn (vector 0 (/ 360 t) 0))

       (turtle-move 1)
       (turtle-vert)
       (if (< n 1)
              0
              (begin
                     ; add another call to the recursion
                     (build-loop (- n 1) t)
                     (turtle-turn (vector 0 0 45))   ; twist a bit
                     (build-loop (- n 1) t))))
```

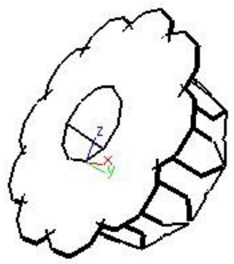


Illustration 33: A circle of circles

Notes on writing large programs in fluxus

When writing large programs in fluxus, I've found that there are some aspects of the language (the PLT scheme dialect to be more precise) which are essential in managing code and keeping things tidy. See the PLT documentation on structs and classes for more detail, but I'll give some fluxus related examples to work with.

For example, at first we can get along quite nicely with using lists alone to store data. Let's use as an example a program with a robot we wish to control:

```
(define myrobot (list (vector 0 0 0) (vector 1 0 0) (build-cube)))
```

We use a list to store the robot's position, velocity and a primitive associated with it.

We could then use:

```
(list-ref myrobot 0) ; returns the position of the robot
(list-ref myrobot 1) ; returns the velocity of the robot
(list-ref myrobot 2) ; returns the root primitive of the robot
```

To get at the values of the robot and use them later. Seems pretty handy, but this has problems when we scale up the problem, say we want to make a world to keep our robots in:

```
; build a world with three robots
(define world (list (list (vector 0 0 0) (vector 1 0 0) (build-cube))
       (list (vector 1 0 0) (vector 1 0 0) (build-cube))
       (list (vector 2 0 0) (vector 1 0 0) (build-cube)))
```

And then say we want to access the 2nd robot's root primitive:

```
(list-ref (list-ref world 1) 2)
```

It all starts to get a little confusing, as we are indexing by number.

Structs

Structs are simple containers that allow you to name data. This is a much saner way of dealing with containers of data than using lists alone.

Let's start again with the robots example:

```
(define-struct robot (pos vel root))
```

Where “pos” is the current position of the robot, “vel” it's velocity and “root” is the primitive for the robot. The (define-struct) automatically generates the following functions we can immediately use:

```
(define myrobot (make-robot (vector 0 0 0) (vector 0 0 0) (build-cube)) ; returns a new robot
(robot-pos myrobot) ; returns the position of the robot
(robot-vel myrobot) ; returns the velocity of the robot
(robot-root myrobot) ; returns the root primitive for the robot
```

This makes for very readable code, as all the data has meaning, no numbers to decipher. It also means we can insert new data and not have to rewrite a lot of code, which is very important.

The world can now become:

```
(define-struct world (robots))
```

And be used like this:

```
; build a world with three robots
(define myworld (make-world (list (make-robot (vector 0 0 0) (vector 1 0 0) (build-cube))
       (make-robot (vector 1 0 0) (vector 1 0 0) (build-cube))
       (make-robot (vector 2 0 0) (vector 1 0 0) (build-cube)))))

; get the 2nd robot's root primitive:
(robot-root (list-ref (world-robots myworld) 1))
```

Mutable state

So far we have only used get's not set's. This is partly because setting state is seen as slightly distasteful in Scheme, so you have to use the following syntax to enable it in a struct:

```
(define-struct robot ((pos #:mutable) (vel #:mutable) root))
```

This (define-struct) also generates the following functions:

```
(set-robot-pos! robot (vector 1 0 0))
(set-robot-vel! robot (vector 0 0.1 0))
```

So you can change the values in the program.

For a full example, see the file dancing-robots.scm in the examples directory.

Classes

Making Movies

Fluxus is designed for real-time use, this means interactive performance or games mainly, but you can also use the frame dump commands to save out frames which can be converted to movies. This process can be fairly complex, if you want to sync visuals to audio, osc or keyboard input.

Used alone, frame dumping will simply save out frames as fast as your machine can render and save them to disk. This is useful in some cases, but not if we want to create a movie at a fixed frame rate, but with the same timing as they are generated at – ie synced with an audio track at 25fps.

Syncing to audio

The (process) command does several things, it switches the audio from the jack input source to a file, but it also makes sure that every buffer of audio is used to produce exactly one frame. Usually in real-time operation, audio buffers will be skipped or duplicated, depending on the variable frame rate and fixed audio rate.

So, what this actually means is that if we want to produce video at 25fps, with audio at 44100 samplerate, 44100/25 = 1764 audio samples per frame. Set your (start-audio) buffer setting to this size. Then all you need to do is make sure the calls to (process) and (start-framedump) happen on the same frame, so that the first frame is at the start of the audio. As this process is not real-time, you can set your resolution as large as you want, or make the script as complex as you like.

Syncing with keyboard input for livecoding recordings

You can use the keypress recorder to save livecoding performances and rerender them later.

To use the key press recorder, start fluxus with -r or -p (see in-fluxus docs for more info). It records the timing of each keypress to a file, it can then replay them at different frame rates correctly.

The keypress recorder works with the process command in the same way as the audio does (you always need an audio track, even if it’s silence). So the recorder will advance the number of seconds per frame as it renders, rather than using the real-time clock – so again, you can make the rendering as slow as you like, it will appear correct when you view the movie.

Recording OSC messages is also possible (for storing things like gamepad activity). Let me know if you want to do this.

Syncing Problems Troubleshooting

Getting the syncing right when combining audio input can be a bit tricky. Some common problems I've seen with the resulting movies fall into two categories.

Syncing lags, and gets worse with time

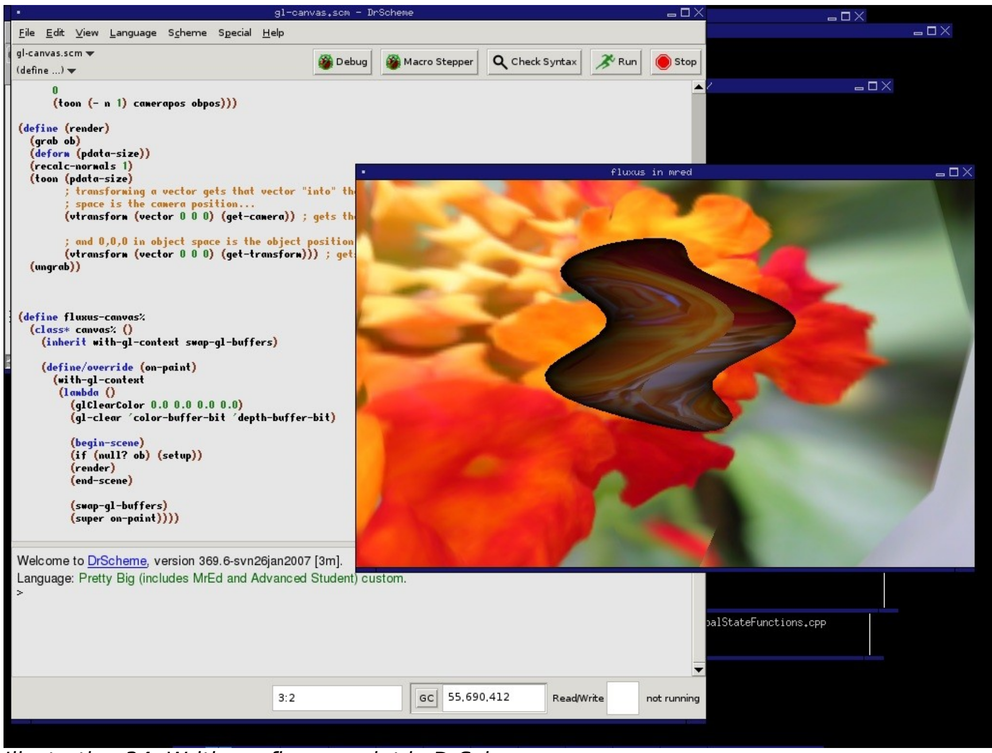
The call to (start-audio) has the wrong buffer size. As I set this in my .fluxus.scm I often forget this. Set it correctly and re-render. Some lagging may happen unavoidably with really long (over 20 minutes or so) animations.

Syncing is offset in a constant manner

This happens when the start of the audio does not quite match the first frame. You can try adding or removing some silence at the beginning of the audio track to sort this out. I often just encode the first couple of seconds until I get it right.

Fluxus In DrScheme

DrScheme is a "integrated development environment" for Scheme, and it comes as part of PLT scheme, so you will have it installed if you are using fluxus.



```
      0
      (toon (- n 1) camerapos obpos)))

(define (render)
  (grab ob)
  (deform (pdata-size))
  (recalc-normals 1)
  (toon (pdata-size)
        ; transforming a vector gets that vector "into" th
        ; space is the camera position...
        (vtransform (vector 0 0 0) (get-camera)) ; gets th

        ; and 0,0,0 in object space is the object position
        (vtransform (vector 0 0 0) (get-transform))) ; get
  (ungrab))


(define fluxus-canvas%
  (class* canvas% ()
    (inherit with-gl-context swap-gl-buffers)

    (define/override (on-paint)
      (with-gl-context
       (lambda ()
         (glClearColor 0.0 0.0 0.0 0.0)
         (gl-clear 'color-buffer-bit 'depth-buffer-bit)

         (begin-scene)
         (if (null? ob) (setup))
         (render)
         (end-scene)

         (swap-gl-buffers)
         (super on-paint))))
```

```
Welcome to DrScheme, version 369.6-svn26jan2007 [3m].
Language: Pretty Big (includes MrEd and Advanced Student) custom.
>
```

3:2 GC 55,690,412 Read/Write not running

Illustration 34: Writing a fluxus script in DrScheme

You can use it instead of the built in fluxus scratchpad editor for writing scheme scripts. Reasons to want to do this include:

- The ability to profile and debug scheme code
- Better editing environment than the fluxus scratchpad editor will ever provide
- Makes fluxus useful in more ways (with the addition of widgets, multiple views etc)

I use it a lot for writing larger scheme scripts. All you need to do is add the following line to the top of your fluxus script:

```
(require fluxus-[version]/drflux)
```

Where [version] is the current version number without the dot, eg "016".

Load the script into DrScheme and press F5 to run it – a new window should pop up, displaying your graphics as normal. Rerunning the script in DrScheme should update the graphics window automatically.

Known issues

Some commands are known to crash DrScheme, (show-fps) should not be used. Hardware shading probably won’t work. Also DrScheme requires a lot of memory, which can cause problems.

Miscellaneous important nuggets of information

This chapter is for things I really think are important to know, but can't find the place to put them.

Getting huge framerate speeds

By default fluxus has it's framerate throttled to stop it melting your computer. To remove this, use:

```
(desiredfps 1000000)
```

It won't guarantee you such a framerate, but it will stop fluxus capping it's speed (which defaults to something around 50 fps). Use:

```
(show-fps 1)
```

To check the fps before and after. Higher framerates are great for VJing, as it essentially reduces the latency for the results of the audio calculation getting to the visual output - it feels much more responsive.

Unit tests

If you want to check fluxus is working ok on a new install - or if you suspect something is going wrong, try:

```
(self-test #f)
```

Which will run through every single example scriptlet in the function reference documentation. If it crashes, or errors - please run:

```
(self-test #t)
```

Which will save a log file - please post this to the mailing list and we'll have a go at fixing it.

Its also highly recommended for developers to run this command before committing code to the source repository, so you can see if your changes have affected anything unexpected.

Fluxus Scratchpad And Modules

This chapter documents fluxus in slightly lower level, only required if you want to hack a bit more.

Fluxus consists of two halves. One half is the window containing a script editor rendered on top of the scene display render. This is called the fluxus scratchpad, and it’s the way to use fluxus for livecoding and general playing.

The other half is the modules which provide functions for doing graphics, these can be loaded into any mzscheme interpreter, and run in any OpenGL context.

Modules

Fluxus’s functionality is further split between different Scheme modules. You don’t need to know any of this to simply use fluxus as is, as they are all loaded and setup for you.

fluxus-engine

This binary extension contains the core rendering functions, and the majority of the commands.

fluxus-audio

A binary extension containing a jack client and fft processor commands.

fluxus-osc

A binary extension with the osc server and client, and message commands.

fluxus-midi

A binary extension with midi event input support

Scheme modules

There are also many scheme modules which come with fluxus. Some of these form the scratchpad interface and give you mouse/key input and camera setup, others are layers on top of the fluxus-engine to make it more convenient. This is where things like the with-* and pdata-map! Macros are specified and the import/export for example.

Fluxa

Fluxa is an optional addition to fluxus which add audio synthesis and sample playback. It’s also an experimental non-deterministic synth where each ‘note’ is it’s own synth graph.

Fluxa is a framework for constructing and sequencing sound. It uses a minimal and purely functional style which is designed for livecoding. It can be broken down into two parts, the descriptions of synthesis graphs and a set of language forms for describing procedural sequences.

(Fluxa is also a kind of primitive homage to supercollider – see also rsc, which is a scheme binding to sc)

Example:

```
   (require fluxus-016/fluxa)
```

```
(seq
       (lambda (time clock)
              (play time (mul (sine 440) (adsr 0 0.1 0 0)))
              0.2))
```

Plays a sine tone with a decay of 0.1 seconds every 0.2 seconds

Non-determinism

Fluxa has decidedly non-deterministic properties – synth lifetime is bound to some global constraints:

- A fixed number of operators, which are recycled (allocation time/space constraint)
- A maximum number of synths playing simultaneously (cpu time constraint)

What this means is that synths are stopped after a variable length of time, depending on the need for new operators. Nodes making up the synth graph may also be recycled while they are in use – resulting in interesting artefacts (which is considered a feature!)

Synthesis commands

```
(play time node-id)
```

Schedules the node id to play at the supplied time. This is for general musical use.

```
(play-now node-id)
```

Plays the node id as soon as possible – mainly for testing

Operator nodes

All these commands create and return a nodes which can be played. Parameters in the synthesis graph can be other nodes or normal number values.

Generators

```
(sine frequency)
```

A sine wave at the specified frequency

```
(saw frequency)
```

A saw wave at the specified frequency

```
(squ frequency)
```

A square wave at the specified frequency

```
(white frequency)
```

White noise

```
(pink frequency)
```

Pink noise

```
(sample sample-filename frequency)
```

Loads and plays a sample – files can be relative to specified searchpaths. Samples will be loaded asynchronously, and won't interfere with real-time audio.

```
(adsr attack decay sustain release)
```

Generates an envelope signal

Maths

```
(add a b)
(sub a b)
(mul a b)
(div a b)
```

Remember that parameters can be nodes or number values, so you can do things like:

```
(play time (mul (sine 440) 0.5))
```

or

```
(play time (mul (sine 440) (adsr 0 0.1 0 0)))
```

Filters

```
(mooghp input-node cutoff resonance)
(moogbp input-node cutoff resonance)
(mooglp input-node cutoff resonance)
(formant input-node cutoff resonance)
```

Global audio

```
(volume 1)
```

Does what is says on the tin

```
(eq 1 1 1)
```

Tweak bass, mid, treble

```
(max-synths 20)
```

Change the maximum concurrent synths playing – default is a measly 10

```
(searchpath path)
```

Add a path for sample loading

```
(reset)
```

The panic command – deletes all synth graphs and reinitialises all the operators – not rt safe

Sequencing commands

Fluxa provides a set of forms for sequencing.

```
(seq (lambda (time clock) 0.1))
```

The top level sequence – there can only be one of these, and all code within the supplied procedure will be called when required. The time between calls is set by the returned value of the procedure – so you can change the global timing dynamically.

The parameters time and clock are passed to the procedure – time is the float real time value in seconds, to be passed to play commands. It’s actually a little bit ahead of real time, in order to give the network messages time to get to the server.

You can also mess with the time like so:

```
(play (+ time 0.5) ...)
```

Which will offset the time half a second into the future. You can also make them happen earlier – but only a little bit.

Clock is an ever increasing value, which increments by one each time the procedure given to seq is called. The value of this is not important, but you can use zmod, which is simply this predefined procedure:

```
(define (zmod clock v) (zero? (modulo clock v)))
```

Which is common enough to make this shortening helpful, so:

```
(if (zmod clock 4) (play (mul (sine 440) (adsr 0 0.1 0 0))))
```

Will play a note every 4 beats.

```
(note 10)
```

A utility for mapping note numbers to frequencies (I think the current scale is equal temperament) [todo: sort scala loading out]

```
(seq
        (lambda (time clock)
                (clock-map
                        (lambda (n)
                                (play time (mul (sine (note n)) (adsr 0 0.1 0 0)))) clock
                (list 10 12 14 15))
        0.1))
```

clock-map maps the list to the play command each tick of the clock – the list can be used as a primitive sequence, and can obviously be used to drive much more than just the pitch.

```
(seq
(lambda (time clock)
(clock-switch clock 128
(lambda ()
(play time (mul (sine (note n)) (adsr 0 0.1 0 0)))) (lambda ()
(play time (mul (saw (note n)) (adsr 0 0.1 0 0))))) 0.1))
```

This clock-switch switches between the procedures every 128 ticks of the clock – for higher level structure.

Syncing

A osc message can be sent to the client for syncing for collaborative performances the format of the sync message is as follows:

```
/sync [iiii] timestamp-seconds timestamp-fraction beats-per-bar tempo
```

When syncing, fluxa provides you with two extra global definitions:

sync-clock : a clock which is reset when a /sync is received sync-tempo : the current requested tempo (you are free to modify or ignore it)

[note: there is a program which adds timestamps to /sync messages coming from a network, which makes collaborative sync work properly (as it doesn’t require clocks to be in sync too) email me if you want more info]

Known problems/todos

- Record execution – cyclic graphs wont work
- Permanent execution of some nodes – will fix delay/reverb

Frisbee

Frisbee is a simplified games engine. It is written in a different language to the rest of fluxus, and requires no knowledge or use of any of the other fluxus commands.

The language it uses is called 'Father Time' (FrTime), which is a functional reactive programming language available as part of PLT Scheme. Functional reactive programming (frp) is a way of programming which emphasises behaviours and events, and makes them a central part of the language.

Programming a graphics environment like a game is all about describing a scene and behaviours which modify it over time. Using normal programming languages (like the base fluxus one) you generally need to do these things separately, build the scene, then animate it. Using FRP, we can describe the scene with the behaviours built in. The idea is that this makes programs smaller and simpler to modify, thus making the process of programming more creative.

A simple frisbee scene

This is the simplest frisbee scene:

```
(require fluxus-015/frisbee)

(scene
  (object))
```

(scene) is the main frisbee command - it is used to define a list of objects and their behaviours.

(object) creates a solid object, by default a cube (of course! :)

We can modify our object by using optional 'keyword parameters', they work like this:

```
(scene
  (object #:shape 'sphere))
```

This sets the shape of the object - there are some built in shapes:

```
(object #:shape 'cube)
(object #:shape 'sphere)
(object #:shape 'torus)
(object #:shape 'cylinder)
```

Or, you can also load in .obj files to make your own shapes:

```
(object #:shape "mushroom.obj")
```

These object files are relative to where you launch fluxus, or they can also live somewhere in the searchpaths for fluxus (which you can set up in your .fluxus.scm script using (searchpath)).

If we want to change the colour of our cube we can add a new parameter:

```
(object
    #:colour (vec3 1 0 0))
```

The vec3 specifies the rgb colour, so this makes a red cube. Note that frisbee uses (vec3) to make it's vectors, rather than (vector).

Here are the other parameters you can set on an object:

```
(object
    #:translate (vec3 0 1 0)
    #:scale (vec3 0.1 1 0.1)
    #:rotate (vec3 0 45 0)
    #:texture "test.png"
    #:hints '(unlit wire))
```

It doesn't matter what order you specify parameters in, the results will be the same. The transform order is always translate first, then rotate, then scale.

Animation

FrTime makes specifying movement very simple:

```
(object
    #:rotate (vec3 0 (integral 10) 0))
```

Makes a cube which rotates 10 degrees every second. Rather than setting the angles explicitly, integral specifies the amount the rotation changes every second. We can also do this:

```
(object
    #:rotate (vec3-integral 0 10 0))
```

Which is easier in some situations.

Making things reactive

What we have made with the integral command is what is called a behaviour - it's value depends on time. This is a core feature of FrTime, and there are many ways to create and
manipulate behaviours. Frisbee also gives you some default behaviours which represent
changing information coming from the outside world:

```
(object
    #:rotate (vec3 mouse-x mouse-y 0))
```

This rotates the cube according to the mouse position.

```
(object
    #:colour (key-press-b #\c (vec3 1 0 0) (vec3 0 1 0)))
```

This changes the colour of the cube when you press the 'c' key.

```
(object
    #:translate (vec3 0 (key-control-b #\q #\a 0.01) 0))
```

This moves the cube up and down as you press the 'q' and 'a' keys, by 0.01 units.

Spawning objects

So far all the objects we have created have stayed active for the duration of the program running. Sometimes we want to control the lifetime of an object, or create new ones. This is obviously important for many games! To do this, we need to introduce events. Events are another basic part of FrTime and therefore Frisbee, and behaviours can be turned into events and vice versa. Events are things which happen at a specific time, rather than behaviours which can always be asked for their current value. For this reason events can't be directly used for driving objects in Frisbee in the same way as behaviours can - but they are used for triggering new objects into, or out of existence.

Here is a script which creates a continuous stream of cubes:

```
(scene
    (factory
        (lambda (e)
            (object #:translate (vec3-integral 0.1 0 0)))
        (metro 1) 5))
```

There are several new things happening here. Firstly the metro command is short for metronome, which creates a stream of events happening at the rate specified (1 per second in this case). (factory) is a command that listens to a stream of events - taken as it's second argument, and runs a procedure passed as it's first argument on each one (passing the event in as an argument to the supplied function).

So in this case, each time an event occurs, the anonymous function is run, which creates an object moving away from the origin. Left like this frisbee would eventually slow down to a crawl, as more and more cubes are created. So the factory also takes a third parameter, which is the maximum number of things it can have alive at any time. Once 5 objects have been created it will recycle them, and remove the oldest objects.

Frisbee come with some built in events which we can visualise with this script:

```
(scene
    (factory
        (lambda (e)
            (object #:translate (vec3-integral 0.1 0 0)))
        keyboard 5))
```

Which spawns a cube each time a key is pressed on the keyboard.
So far we have been ignoring the event which gets passed into our little cube making function, but as the events the keyboard spits out are the keys which have been pressed, we can make use of them thusly:

```
(scene
    (factory
        (lambda (e)
            (if (char=? e #\a) ; make a bigger cube if 'a' is pressed
                (object
                    #:translate (vec3-integral 0.1 0 0)
                    #:scale (vec3 2 2 2))
                (object #:translate (vec3-integral 0.1 0 0))))
        keyboard 5))
```

Converting behaviours to events

You can create events when a behaviour changes:

```
(scene
    (factory
        (lambda (e)
            (object #:translate (vec3-integral 0.1 0 0)))
        (when-e (> mouse-x 200)) 5))
```

Particles

Frisbee comes with it's own particle system primitive - which makes it easy to make different particle effects.

It is created in a similar way to the solid objects:

```
(scene
    (particles))
```

And comes with a set of parameters you can control explicitly or via behaviours:

```
(scene
    (particles
        #:colour (vec3 1 1 1)
        #:translate (vector 0 0 0)
        #:scale (vector 0.1 0.1 0.1)
        #:rotate (vector 0 0 0)
        #:texture "test.png"
        #:rate 1
        #:speed 0.1
        #:spread 360
        #:reverse #f))
```

Function reference

This is an exhaustive description of each function in fluxus, and is adapted from the in-fluxus help system.

fluxa

Description

Fluxa is the fluxus audio synth for livecoding, it contains quite basic atomic components which can be used together to create more complicated sounds. It uses an experimental and fairly brutal method of graph node garbage collection which gives it certain non-deterministic qualities. It's also been battle tested in many a live performance. The fluxa server needs to be run and connected to jack in order for you to hear anything. Also, fluxa is not in the default namespace, so use eg (require fluxus-016/fluxa).

(reload)

Returns void

Causes samples to be reloaded if you need to restart the fluxa server

Example

```
(reload)
```

(sine frequency-number-or-node)

Returns node-id-number

Creates a sine wave generator node

Example

```
 (play-now (mul (sine 440) (asdr 0.1 0.1 0 0)))
```

(saw frequency-number-or-node)

Returns node-id-number

Creates a saw wave generator node

Example

```
 (play-now (mul (saw 440) (asdr 0.1 0.1 0 0)))
```

(tri frequency-number-or-node)

Returns node-id-number

Creates a triangle wave generator node

Example

```
 (play-now (mul (tri 440) (asdr 0.1 0.1 0 0)))
```

(squ frequency-number-or-node)

Returns node-id-number

Creates a square wave generator node

Example

```
 (play-now (mul (squ 440) (asdr 0.1 0.1 0 0)))
```

(white frequency-number-or-node)

Returns node-id-number

Creates a white noise generator node

Example

```
 (play-now (mul (white 5) (asdr 0.1 0.1 0 0)))
```

(pink frequency-number-or-node)

Returns node-id-number

Creates a pink noise generator node

Example

```
(play-now (mul (pink 5) (asdr 0.1 0.1 0 0)))
```

(add number-or-node number-or-node)

Returns node-id-number

Maths node - adds two signals together

Example

```
(play-now (mul (add (sine 440) (sine 220)) (asdr 0.1 0.1 0 0)))
```

(sub number-or-node number-or-node)

Returns node-id-number

Maths node - subtracts two signals

Example

```
(play-now (mul (sub (sine 440) (sine 220)) (asdr 0.1 0.1 0 0)))
```

(mul number-or-node number-or-node)

Returns node-id-number

Maths node - multiplies two signals

Example

```
(play-now (mul (mul (sine 440) (sine 220)) (asdr 0.1 0.1 0 0)))
```

(mul number-or-node number-or-node)

Returns node-id-number

Maths node - multiplies two signals

Example

```
(play-now (mul (div (sine 440) 2) (asdr 0.1 0.1 0 0)))
```

(pow number-or-node number-or-node)

Returns node-id-number

Maths node - produces a signal raised to the power of another

Example

```
(play-now (mul (pow (adsr 0 0.1 0 0) 10) (sine 440)))
```

(adsr attack-number-or-node decay-number-or-node sustain-number-or-node release-number-or-node)

Returns node-id-number

Creates an envelope generator node

Example

```
(play-now (mul (sine 440) (asdr 0.1 0.1 0 0)))
```

(mooglp signal-node cutoff-number-or-node resonance-number-or-node)

Returns node-id-number

Creates an low pass moog filter node

Example

```
(play-now (mul (mooglp (squ 440) 0.1 0.4) (asdr 0.1 0.1 0 0)))
```

(moogbp signal-node cutoff-number-or-node resonance-number-or-node)

Returns node-id-number

Creates an band pass moog filter node

Example

```
(play-now (mul (moogbp (squ 440) 0.1 0.4) (asdr 0.1 0.1 0 0)))
```

(mooghp signal-node cutoff-number-or-node resonance-number-or-node)

Returns node-id-number

Creates an high pass moog filter node

Example

```
(play-now (mul (mooghp (squ 440) 0.1 0.4) (asdr 0.1 0.1 0 0)))
```

(formant signal-node cutoff-number-or-node resonance-number-or-node)

Returns node-id-number

Creates a formant filter node

Example

```
(play-now (mul (formant (squ 440) 0.1 0.4) (asdr 0.1 0.1 0 0)))
```

(sample sample-filename-string frequency-number-or-node)

Returns node-id-number

Creates a sample playback node

Example

```
(play-now (sample "helicopter.wav" 440))
```

(crush signal-node frequency-number-or-node bit-depth-number-or-node)

Returns node-id-number

Creates a crush effect node

Example

```
(play-now (crush (sine 440) 0.4 8))
```

(distort signal-node amount-number-or-node)

Returns node-id-number

Creates a distortion effect node

Example

```
(play-now (distort (sine 440) 0.9))
```

(klip signal-node amount-number-or-node)

Returns node-id-number

Creates a hard clipping distortion effect node

Example

```
(play-now (klip (sine 440) 0.9))
```

(echo signal-node delay-time-number-or-node feedback-number-or-node)

Returns node-id-number

Creates a hard clipping distortion effect node

Example

```
(play-now (klip (sine 440) 0.9))
```

(play time node)

Returns void

Plays a supplied node at the specified time.

Example

```
(play (+ (time-now) 10) (mul (adsr 0 0.1 0 0) (sine 440)))
```

(play-now node)

Returns void

Plays a supplied node as soon as possible

Example

```
(play-now (mul (adsr 0 0.1 0 0) (sine 440)))
```

(fluxa-debug true-or-false)

Returns void

Turns on or off fluxa debugging, the server will print information out to stdout

Example

```
(fluxa-debug #t)
```

(volume amount-number)

Returns void

Sets the global volume

Example

```
(volume 2.5)
```

(pan pan-number)

Returns void

Sets the global pan where -1 is left and 1 is right (probably)

Example

```
(pan 0)
```

(max-synths number)

Returns void

Sets the maximum amount of synth graphs fluxa will run at the same time. This is a processor usage safeguard, when the count is exceeded the oldest synth graph will be stopped so it's nodes can be recycled. The default count is 10.

Example

```
(max-synths 10)
```

(searchpath path-string)

Returns void

Add a searchpath to use when looking for samples

Example

```
(searchpath "/path/to/my/samples/)
```

(eq bass-number middle-number high-number)

Returns void

Sets a simple global equaliser. This is more as a last resort when performing without a mixer.

Example

```
(eq 2 1 0.5) ; bass boost
```

(comp attack-number release-number threshold-number slope-number)

Returns void

A global compressor. Not sure if this works yet.

Example

```
(comp 0.1 0.1 0.5 3)
```

(note note-number)

Returns frequency-number

Returns the frequency for the supplied note. Fluxa uses just intonation by default.

Example

```
(comp 0.1 0.1 0.5 3)
```

(reset)

Returns void

Resets the server.

Example

```
(reset)
```

(clock-map)

Returns void

A way of using lists as sequences. The lists can be of differing length, leading to polyrhythms.

Example

```
(seq (lambda (time clock)
               (clock-map
                       (lambda (nt cutoff)
                               (play time (mul (adsr 0 0.1 0 0)
                                       (mooglp (saw (note nt)) cutoff 0.4))))
                       clock
                       (list 39 28 3)
                       (list 0.1 0.1 0.4 0.9))))
```

(zmod clock-number count-number)

Returns true-or-false

Just shorthand for (zero? (modulo clock-number count-number)), as it can be used a lot.

Example

```
(seq (lambda (time clock)
    (when (zmod clock 4) ; play the note every 4th beat
        (play time (mul (adsr 0 0.1 0 0) (sine (note nt)))))))
```

(seq proc)

Returns void

Sets the global fluxa sequence procedure, which will be called automatically in order to create new events. seq can be repeatedly called to update the procedure as in livecoding.

Example

```
(seq (lambda (time clock)
    (when (zmod clock 4) ; play the note every 4th beat
        (play time (mul (adsr 0 0.1 0 0) (sine (note nt)))))))
```

scheme-utils

Description

High level fluxus commands written in Scheme.

(detach-parent)

Returns void

Removes the parent for the current primitive, and fixes up the transform so the primitive doesn't move. Use (parent 1) to avoid this fix up.

Example

```
; builds and animates a random heirarchical structure,
; click on the objects to detach them from their parents
(define (build-heir depth)
    (with-state
        (let ((p (with-state
                        (translate (vector 2 0 0))
                        (scale 0.9)
                        (build-cube))))
            (when (> depth 0)
                (parent p)
                (for ((i (in-range 0 5)))
                    (when (zero? (random 3))
                        (rotate (vector 0 0 (* 45 (crndf))))
                        (build-heir (- depth 1))))))))

(define (animate-heir children depth)
    (for-each
        (lambda (child)
            (with-primitive child
                (rotate (vector 0 0 (sin (+ depth (time)))))
                (animate-heir (get-children) (+ depth 1))))
        children))

(define (animate)
    (animate-heir (get-children) 0)
    (when (mouse-button 1)
        (let ((s (select (mouse-x) (mouse-y) 2)))
            (when (not (zero? s))
                (with-primitive s
                    (detach-parent))))))

(clear)
(build-heir 5)
(every-frame (animate))
```

(with-state expression ...)

Returns result of last expression

Encapsulates local state changes, and removes the need for push and pop.

Example

```
; state hierachy, by nesting with-state:
(with-state
   (hint-vertcols)
   (colour (vector 0 0 1))
   (with-state
       (translate (vector 1 0 0))
       (build-sphere 10 10))
    (build-torus 1 2 30 30))

; making primitives:
(define my-torus (with-state
   (hint-vertcols)
   (colour (vector 0 0 1))
   (build-torus 1 2 30 30)))
```

(with-primitive primitive expression ...)

Returns result of last expression

Encapsulates primitive state changes, and removes the need for grab and ungrab.

Example

```
(define my-torus (with-state
   (colour (vector 0 0 1))
   (build-torus 1 2 30 30)))

; change the torus colour:
(with-primitive my-torus
   (colour (vector 0 1 0)))
```

(pdata-map! procedure read/write-pdata-name read-pdata-name ...)

Returns void

A high level control structure for simplifying passing over pdata arrays for primitive deformation. Should be easier and less error prone than looping manually. Writes to the first pdata array.

Example

```
(clear)
(define my-torus (build-torus 1 2 30 30))

(with-primitive my-torus
  (pdata-map!
     (lambda (position)
         (vadd position (vector (flxrnd) 0 0))) ; jitter the vertex in x
     "p")) ; read/write the position pdata array

(with-primitive my-torus
  (pdata-map!
      (lambda (position normal)
          (vadd position normal)) ; add the normal to the position (expand the
object)
      "p" "n")) ; read/write the position pdata array, read the normals array
```

(pdata-index-map! procedure read/write-pdata-name read-pdata-name ...)

Returns void

A high level control structure for simplifying passing over pdata arrays for primitive deformation. Same as pdata-map! except pdata-index-map! supplies the index of the current pdata element as the first argument to 'procedure'.

Example

```
 (clear)
 (define my-torus (build-torus 1 2 30 30))

 (with-primitive my-torus
   (pdata-index-map!
      (lambda (index position)
          (vadd position (vector (gh index) 0 0))) ; jitter the vertex in x
      "p")) ; read/write the position pdata array
```

(pdata-fold procedure start-value read-pdata-name ...)

Returns result of folding procedure over pdata array

A high level control structure for doing calculations on pdata arrays. Runs the procedure over each pdata element accumulating the result. Should be easier and less error prone than looping manually.

Example

```

 (define my-torus (build-torus 1 2 30 30))

 ; find the centre of the primitive by averaging
 ; the points position's together
 (let ((centre
        (with-primitive my-torus
                        (vdiv (pdata-fold
                               vadd
                               (vector 0 0 0)
                               "p") (pdata-size)))))

   (display centre)(newline))
```

(pdata-index-fold procedure start-value read-pdata-name ...)

Returns result of folding procedure over pdata array

Same as pdata-fold except it passes the index of the current pdata element as the first parameter of 'procedure'.

Example

```
(define my-torus (build-torus 1 2 30 30))

; can't think of a good example for this yet...
(let ((something
       (with-primitive my-torus
                        (vdiv (pdata-index-fold
                               (lambda (index position ret)
                                   (vadd ret (vmul position index)))
                               (vector 0 0 0)
                               "p") (pdata-size)))))

  (display something)(newline))
```

(collada-import filename-string)

Returns void

Loads a collada scene file and returns a scene description list. Files need to contain triangulated model data - this is usually an option on the export. Note: this is slow for heavy models

Example

```
;(collada-import "test.dae")
```

(vmix a b t)

Returns void

Linearly interpolates the two vectors together by t

Example

```
; mix red and blue together
(colour (vmix (vector 1 0 0) (vector 0 0 1) 0.5))
```

(vclamp a)

Returns void

Clamp the vector so the elements are all between 0 and 1

Example

```
; make a valid colour from any old vector
(colour (vclamp (vector 2 400 -123)))
```

(vsquash a)

Returns void

Clamp the vector so the elements are all between 0 and 1

Example

```
; make a valid colour from any old vector
(colour (vclamp (vector 2 400 -123)))
```

(pixels-circle pos radius colour)

Returns void

Draws a circle into a pixels primitive

Example

```
(with-primitive (build-pixels 100 100)
    (pixels-circle (vector 50 50 0) 30 (vector 1 0 0 1))
    (pixels-upload))
```

(pixels-blend-circle pos radius colour)

Returns void

Draws a blended circle into a pixels primitive

Example

```
(with-primitive (build-pixels 100 100)
    (pixels-blend-circle (vector 50 50 0) 30 (vector 1 0 0 1))
    (pixels-upload))
```

(pixels-dodge pos radius strength)

Returns void

Lightens a circular area of a pixels primitive

Example

```
(with-primitive (build-pixels 100 100)
    (pixels-dodge (vector 50 50 0) 30 (vector 1 0 0 1))
    (pixels-upload))
```

(pixels-burn pos radius strength)

Returns void

Darkens a circular area of a pixels primitive

Example

```
(with-primitive (build-pixels 100 100)
    (pixels-burn (vector 50 50 0) 30 (vector 1 0 0 1))
    (pixels-upload))
```

(pixels-clear col)

Returns void

Sets all of the pixels to the supplied colour

Example

```
(with-primitive (build-pixels 100 100)
    (pixels-clear (vector 1 0 0))
    (pixels-upload))
```

(poly-type)

Returns void

Returns a symbol representing the type of the current polygon primitive. primitive.

Example

```
(define p (build-polygons 3 'triangle-strip))
(with-primitive p
    (display (poly-type))(newline))
```

(pdata-for-each-face proc pdatanames)

Returns list of pdata values

Calls proc with the indices for each face in a polygon primitive

Example

(pdata-for-each-triangle proc)

Returns list of pdata values

Calls proc with the pdata for each triangle in a face - assumes all faces are convex.

Example

(pdata-for-each-tri-sample proc samples-per-triangle)

Returns void

Calls proc with the triangle indices and a random barycentric coord.

Example

(rndf)

Returns number

Returns a random number in the range 0->1

Example

```
(display (rndf))(newline)
```

(crndf)

Returns number

Returns a random number in the range -1->1 (centred on zero)

Example

```
(display (crndf))(newline)
```

(rndvec)

Returns vector

Returns a random 3 element vector with each element in the range 0->1. If you visualise a lot of these as points, they will fill the unit cube (see the example).

Example

```
(clear)
(hint-none)
(hint-points)
(point-width 4)
(define p (build-particles 1000))

(show-axis 1)

(with-primitive p
    (pdata-map!
        (lambda (p)
            (vector 1 1 1))
        "c")
    (pdata-map!
        (lambda (p)
            (rndvec))
        "p"))
```

(crndvec)

Returns vector

Returns a random 3 element vector with each element in the range -1->1. If you visualise a lot of these as points, they will fill a cube centred on the origin (see the example).

Example

```
(clear)
(hint-none)
(hint-points)
(point-width 4)
(define p (build-particles 1000))

(show-axis 1)

(with-primitive p
    (pdata-map!
        (lambda (p)
            (vector 1 1 1))
        "c")
    (pdata-map!
        (lambda (p)
            (crndvec))
        "p"))
```

(srndvec)

Returns vector

Returns a random 3 element vector. If you visualise a lot of these as points, they will fill a sphere centred on the origin (see the example).

Example

```
(clear)
(hint-none)
(hint-points)
(point-width 4)
(define p (build-particles 1000))

(show-axis 1)

(with-primitive p
    (pdata-map!
        (lambda (p)
            (vector 1 1 1))
        "c")
    (pdata-map!
        (lambda (p)
            (srndvec))
        "p"))
```

(hsrndvec)

Returns vector

Returns a random 3 element vector. If you visualise a lot of these as points, they will cover the surface of a sphere centred on the origin (see the example). The name stands for "hollow sphere".

Example

```
(clear)
(hint-none)
(hint-points)
(point-width 4)
(define p (build-particles 1000))

(show-axis 1)

(with-primitive p
    (pdata-map!
        (lambda (p)
            (vector 1 1 1))
        "c")
    (pdata-map!
        (lambda (p)
            (hsrndvec))
        "p"))
```

(grndf)

Returns number

Returns a gaussian random number in the range centred on zero, with a variance of 1

Example

```
(display (grndf))(newline)
```

(grndvec)

Returns vector

Returns a gaussian random 3 element vector. If you visualise a lot of these as points, you will see a normal distribution centred on the origin. (see the example).

Example

```
(clear)
(hint-none)
(hint-points)
(point-width 4)
(define p (build-particles 1000))

(show-axis 1)
```

```
(with-primitive p
    (pdata-map!
        (lambda (p)
            (vector 1 1 1))
        "c")
    (pdata-map!
        (lambda (p)
            (grndvec))
        "p"))
```

(rndbary)

Returns vector

Returns a vector representing a uniformly distributed triangular barycentric coordinate (wip - doesn't seem to be very uniform to me...)

Example

```
(rndbary)
```

(rndbary normal)

Returns vector

Returns a vector representing a random point on a hemisphere, defined by normal.

Example

```
(clear)
(hint-none)
(hint-points)
(point-width 4)
(define p (build-particles 1000))

(show-axis 1)

(with-primitive p
    (pdata-map!
        (lambda (p)
            (vector 1 1 1))
        "c")
    (pdata-map!
        (lambda (p)
            (rndhemi (vector 0 1 0)))
        "p"))
```

(hrndbary normal)

Returns vector

Returns a vector representing a random point on a hollow hemisphere, defined by normal.

Example

```
(clear)
(hint-none)
(hint-points)
(point-width 4)
(define p (build-particles 1000))

(show-axis 1)

(with-primitive p
    (pdata-map!
        (lambda (p)
            (vector 1 1 1))
        "c")
    (pdata-map!
        (lambda (p)
            (hrndhemi (vector 0 1 0)))
        "p"))
```

(pdata-for-each-tri-sample proc samples-per-triangle)

Returns void

Calls proc with the triangle indices and a random barycentric coord.

Example

(pdata-for-each-tri-sample proc samples-per-triangle)

Returns void

Calls proc with the triangle indices and a random barycentric coord.

Example

(occlusion-texture-bake tex prim samples-per-face rays-per-sample ray-length debug)

Returns void

Bakes ambient occlusion textures. See ambient-occlusion.scm for more info.

Example

midi

Description

MIDI stands for Musical Instrument Digital Interface, and it enables electronic musical instruments, computers, and other equipment to communicate, control, and synchronize with each other. Fluxus can receive MIDI control change and note messages.

Example

```
(display (midi-info))(newline)

(midi-init 1)

(define (midi-test)
    (with-state
        (scale (vector (+ 1 (midi-ccn 0 1))
                       (+ 1 (midi-ccn 0 2))
                       (+ 1 (midi-ccn 0 3))))
        (draw-cube)))

(every-frame (midi-test))
```

(midi-info)

Returns a list of (midi-port-number . midi-port-name-string) pairs

Returns information about the available MIDI input ports.

Example

```
(midi-info)
```

(midi-init port-number)

Returns void

Opens the specified MIDI input port.

Example

```
(midi-init 1)
```

(midi-cc channel-number controller-number)

Returns controller-value-number

Returns the controller value.

Example

```
(midi-cc 0 1)
```

(midi-ccn channel-number controller-number)

Returns controller-value-number

Returns the controller value normalised to the (0, 1) interval.

Example

```
(midi-ccn 0 1)
```

(midi-note)

Returns #(on-off-symbol channel note velocity) or #f

Returns the next event from the MIDI note event queue or #f if the queue is empty.

Example

```
(midi-note)
```

(midi-peek)

Returns msg-string

Returns the name, and event type, and parameter bytes of the last MIDI event as a string for debugging purposes.

Example

```
(display (midi-peek))(newline)
```

turtle

Description

The turtle polybuilder is an experimental way of building polygonal objects using a logo style turtle in 3D space. As you drive the turtle around you can place vertices and build shapes procedurally. The turtle can also be used to deform existing polygonal primitives, by attaching it to objects you have already created.

Example

```
(define (build n)
    (turtle-reset)
    (turtle-prim 4)
    (build-loop n n)
    (turtle-build))

(define (build-loop n t)
    (turtle-turn (vector 0 (/ 360 t) 0))
    (turtle-move 1)
```

```
     (turtle-vert)
     (if (< n 1)
         0
         (build-loop (- n 1) t)))
```

(turtle-prim type-number)

Returns void

Starts building a new polygon primitive with the turtle. The type specifies the polygon face type and is one of the following: 0: triangle strip, 1: quad list, 2: triangle list, 3: triangle fan, 4: general polygon

Example

```
 (turtle-prim 0)
```

(turtle-vert)

Returns void

Creates a new vertex in the current position, or sets the current vertex if the turtle builder is attached.

Example

```
 (turtle-vert)
```

(turtle-build)

Returns primitiveid-number

Builds the object with the vertex list defined and gives it to the renderer. Has no effect if the turtle builder is attached to a primitive.

Example

```
 (define mynewshape (turtle-build))
```

(turtle-move distance-number)

Returns void

Moves the turtle forward in it's current orientation.

Example

```
 (turtle-move 1)
```

(turtle-push)

Returns void

The turtle build has it's own transform stack. Push remembers the current position and orientation.

Example

```
(turtle-push)
```

(turtle-pop)

Returns void

The turtle build has it's own transform stack. Pop forgets the current position and orientation, and goes back to the state at the last push.

Example

```
(turtle-pop)
```

(turtle-turn rotation-vector)

Returns void

Rotates the turtle's orientation with the supplied euler angles (rotations in x, y and z).

Example

```
(turtle-turn (vector 45 0 0))
```

(turtle-reset)

Returns void

Resets the current position and rotation of the turtle to the origin.

Example

```
(turtle-reset)
```

(turtle-attach primitiveid-number)

Returns void

Attaches the turtle to an existing poly primitive. This means you are able to deform an existing objects points using the turtle builder.

Example

```
(define myshape (build-sphere 10 10))
(turtle-attach myshape)
```

(turtle-skip count-number)

Returns void

When attached, causes the turtle to skip vertices. This value may be negative, which will set the turtle to write to previous vertices.

Example

```
(turtle-skip -1)
```

(turtle-position)

Returns count-number

When attached, returns the current pdata index the turtle is writing to.

Example

```
(display (turtle-position))(newline)
```

(turtle-seek position-number)

Returns void

When attached, sets the absolute pdata index the turtle is writing to.

Example

```
(turtle-seek 0)
```

physics

Description

The physics system used in fluxus is based on the ode library, which allows you to add physical properties to objects and set them in motion. Since ODE is designed for rigid-body simulations, structures are described in terms of objects, joints and forces. A much more comprehensive explanation of these concepts can be found in the ODE documentation, which you have probably downloaded if you have compiled fluxus, or can be found at @url{http://ode.org/ode-docs.html} To help with debugging joints, try calling (render-physics) every frame, which will render locators showing you positions and axes of joints that have positional information.

(collisions on/off-number)

Returns void

Enables or disables collision detection. Defaults to off.

Example

```
(collisions 1)
```

(ground-plane plane-vector offset-number)

Returns void

Create an infinite passive plane for use as the 'ground'

Example

```
(ground-plane (vector 0 1 0) 0)
```

(active-box primitiveid-number)

Returns void

Enable the object to be acted upon by the physics system, using a box as the bounding volume. As an active object, it will be transformed by ode. Note: rotations only work correctly if you specify your transforms scale first, then rotate (translate doesn't matter) basically, ode can't deal with shearing transforms.

Example

```
(define mycube (build-cube))
(active-box mycube)
```

(active-cylinder primitiveid-number)

Returns void

Enable the object to be acted upon by the physics system, using a cylinder as the bounding volume. As an active object, it will be transformed by ode. Note: rotations only work correctly if you specify your transforms scale first, then rotate (translate doesn't matter) basically, ode can't deal with shearing transforms.

Example

```
(define mycube (build-cube))
(active-cylinder mycube)
```

(active-sphere primitiveid-number)

Returns void

Enable the object to be acted upon by the physics system, using a sphere as the bounding volume. As an active object, it will be transformed by ode. Note: rotations only work correctly if you specify your transforms scale first, then rotate (translate doesn't matter) basically, ode can't deal with shearing transforms.

Example

```
(define mycube (build-cube))
(active-sphere mycube)
```

(passive-box primitiveid-number)

Returns void

Enable the object to be acted upon by the physics system, using a box as the bounding volume. As a passive object, active objects will collide with it, but it will not be transformed. Note: rotations only work correctly if you specify your transforms scale first, then rotate (translate doesn't matter) basically, ode can't deal with shearing transforms.

Example

```
(define mycube (build-cube))
(passive-box mycube)
```

(passive-cylinder primitiveid-number)

Returns void

Enable the object to be acted upon by the physics system, using a cylinder as the bounding volume. As a passive object, active objects will collide with it, but it will not be transformed. Note: rotations only work correctly if you specify your transforms scale first, then rotate (translate doesn't matter) basically, ode can't deal with shearing transforms.

Example

```
(define mycube (build-cube))
(passive-cylinder mycube)
```

(passive-sphere primitiveid-number)

Returns void

Enable the object to be acted upon by the physics system, using a sphere as the bounding volume. As a passive object, active objects will collide with it, but it will not be transformed. Note: rotations only work correctly if you specify your transforms scale first, then rotate (translate doesn't matter) basically, ode can't deal with shearing transforms.

Example

```
(define mycube (build-cube))
(passive-sphere mycube)
```

(surface-params slip1-number slip2-number softerp-number softcfm-number)

Returns void

Sets some global surface attributes that affect friction and bouncyness. see section 7.3.7 of the ODE docs for an explanation of these parameters

Example

```
(surface-params 0.1 0.1 0.1 0.1)
```

(build-balljoint primitiveid-number primitiveid-number axis-vector)

Returns void

Creates a balljoint to connect two objects (see the ode docs for a detailed description of the differences between the joint types). ODE considers joints to be a constraint that is enforced between two objects. When creating a joint, it is important to have the two primitives being joined in the desired positions before creating the joint. Joints can be created, modified and indexed in a similar way to other primitives.

Example

```
(clear)
(ground-plane (vector 0 1 0) -1)
(collisions 1)

(define shape1 (with-state
        (translate (vector -1 0 0))
        (build-cube)))
(active-box shape1)

(define shape2 (with-state
        (translate (vector 1 0 0))
        (build-cube)))
(active-box shape2)

(build-balljoint shape1 shape2 (vector 0 0 0))
(kick shape1 (vector 0 2 0))

(set-physics-debug #t)
```

(build-fixedjoint primitiveid-number)

Returns void

Creates a joint to connect an object to the global environment. This locks the object in place.

Example

```
(clear)
(define shape1 (with-state
        (translate (vector 0 1 0))
        (build-cube)))
(active-box shape1)

(build-fixedjoint shape1) ; not very exciting...
```

(build-hingejoint primitiveid1-number primitiveid2-number anchor-vector axis-vector)

Returns hingeid-number

Creates a ball joint to connect two objects (see the ode docs for a detailed description of the differences between the joint types). ODE considers joints to be a constraint that is enforced between two objects. When creating a joint, it is important to have the two primitives being joined in the desired positions before creating the joint. Joints can be created, modified and indexed in a similar way to other primitives.

Example

```
(clear)
(ground-plane (vector 0 1 0) -1)
(collisions 1)

(define shape1 (with-state
        (translate (vector -1 0 0))
        (build-cube)))
(active-box shape1)

(define shape2 (with-state
        (translate (vector 1 0 0))
        (build-cube)))
(active-box shape2)

(build-hingejoint shape1 shape2 (vector 0 0 0) (vector 0 0 1))
(kick shape1 (vector 0 2 0))

(set-physics-debug #t)
```

(build-sliderjoint primitiveid1-number primitiveid2-number axis-vector)

Returns hingeid-number

Creates a slider joint to connect two objects (see the ode docs for a detailed description of the differences between the joint types). ODE considers joints to be a constraint that is enforced between two objects. When creating a joint, it is important to have the two primitives being joined in the desired positions before creating the joint. Joints can be created, modified and indexed in a similar way to other primitives.

Example

```
(clear)
(ground-plane (vector 0 1 0) -1)
(collisions 1)

(define shape1 (with-state
        (translate (vector -1 0 0))
        (build-cube)))
(active-box shape1)
```

```
(define shape2 (with-state
        (translate (vector 1 0 0))
        (build-cube)))
(active-box shape2)

(build-sliderjoint shape1 shape2 (vector 1 0 0))
(kick shape1 (vector 0 2 0))

(set-physics-debug #t)
```

(build-hinge2joint primitiveid1-number primitiveid2-number anchor-vector axis1-vector axis2-vector)

Returns hingeid-number

Creates a hinge2 joint to connect two objects (see the ode docs for a detailed description of the differences between the joint types). ODE considers joints to be a constraint that is enforced between two objects. When creating a joint, it is important to have the two primitives being joined in the desired positions before creating the joint. Joints can be created, modified and indexed in a similar way to other primitives.

Example

```
(clear)
(ground-plane (vector 0 1 0) -1)
(collisions 1)

(define shape1 (with-state
        (translate (vector -1 0 0))
        (build-cube)))
(active-box shape1)

(define shape2 (with-state
        (translate (vector 1 0 0))
        (build-cube)))
(active-box shape2)

(build-hinge2joint shape1 shape2 (vector 0 0 0) (vector 1 0 0) (vector 0 1 0))
(kick shape1 (vector 0 2 0))

(set-physics-debug #t)
```

(build-amotorjoint primitiveid1-number primitiveid2-number axis-vector)

Returns hingeid-number

Creates a angular motor joint to connect two objects (see the ode docs for a detailed description of the differences between the joint types). ODE considers joints to be a constraint that is enforced between two objects. When creating a joint, it is important to have the two primitives being joined in the desired positions before creating the joint. Joints can be created, modified and indexed in a similar way to other primitives.

Example

```
 (clear)
 (ground-plane (vector 0 1 0) -1)
 (collisions 1)

 (define shape1 (with-state
         (translate (vector -1 0 0))
         (build-cube)))
 (active-box shape1)

 (define shape2 (with-state
         (translate (vector 1 0 0))
         (build-cube)))
 (active-box shape2)

 (build-amotorjoint shape1 shape2 (vector 1 0 0))
 (kick shape1 (vector 0 2 0))

 (set-physics-debug #t)
```

(joint-param jointid-number param-string value-number)

Returns hingeid-number

Sets the joint parameter for a joint where param is one of the following: "HiStop", "Vel", "FMax", "FudgeFactor", "Bounce", "CFM", "StopERP", "StopCFM","SuspensionERP", "SuspensionCFM", "Vel2", "FMax2". see section 7.5.1 of the ODE docs for an explanation of each of these parameters, and which joint types they apply to.

Example

```
 (clear)
 (ground-plane (vector 0 1 0) -1)
 (collisions 1)

 (define shape1 (with-state
         (translate (vector -1 0 0))
         (build-cube)))
 (active-box shape1)

 (define shape2 (with-state
         (translate (vector 1 0 0))
         (build-cube)))
 (active-box shape2)

 (define j (build-hinge2joint shape1 shape2 (vector 0 0 0) (vector 1 0 0)
(vector 0 1 0)))
 (joint-param j "Vel2" 0.1)
 (joint-param j "FMax2" 0.2)
 (joint-param j "LoStop" -0.75)
 (joint-param j "HiStop" 0.75)

 (set-physics-debug #t)
```

(joint-angle jointid-number angle-number vel-number)

Returns void

Set a new angle for this joint, with a given velocity taken to get there

Example

```
(clear)
(ground-plane (vector 0 1 0) -1)
(collisions 1)

(define shape1 (with-state
        (translate (vector -1 0 0))
        (build-cube)))
(active-box shape1)

(define shape2 (with-state
        (translate (vector 1 0 0))
        (build-cube)))
(active-box shape2)

(define j (build-hingejoint shape1 shape2 (vector 0 0 0) (vector 0 1 0)))
(joint-param j "FMax" 20)
(joint-param j "LoStop" -1)
(joint-param j "HiStop" 1)

(set-physics-debug #t)

(define (animate)
    (joint-angle j 0.1 (* 5 (sin (time)))))
```

(joint-slide jointid-number force)

Returns void

Applies the given force in the slider's direction. That is, it applies a force with magnitude force, in the direction slider's axis, to body1, and with the same magnitude but opposite direction to body2.

Example

```
(clear)
(ground-plane (vector 0 1 0) -1)
(collisions 1)

(define shape1 (with-state
        (translate (vector -1 0 0))
        (build-cube)))
(active-box shape1)

(define shape2 (with-state
        (translate (vector 1 0 0))
        (build-cube)))
(active-box shape2)

(define j (build-sliderjoint shape1 shape2 (vector 1 0 0)))
(joint-param j "FMax" 20)
```

```
(joint-param j "LoStop" -1)
(joint-param j "HiStop" 1)

(set-physics-debug #t)

(define (animate)
    (joint-slide j (* 5 (sin (time)))))
```

(set-max-physical max-number)

Returns void

Sets the maximum number of objects the physics system can deal with. When the max level has been reached the oldest objects are automatically destroyed.

Example

```
(clear)
(set-max-physical 200)

(every-frame
    (with-state
    (translate (vector 0 5 0))
        (scale (srndvec))
        (colour (rndvec))
        (let ((ob (build-cube)))
            (active-box ob)
            (kick ob (vmul (srndvec) 3))
            (twist ob (vmul (srndvec) 2)))))
```

(set-mass primitiveid-number mass-number)

Returns void

Sets the mass of an active object

Example

```
(clear)
(ground-plane (vector 0 1 0) 0)
(collisions 1)
(set-max-physical 20)

; not a great example, but these boxes will have
; different mass, so behave a bit differently.

(every-frame
    (when (> (rndf) 0.92)
        (with-state
            (translate (vector 0 5 0))
            (scale (vmul (rndvec) 5))
            (colour (rndvec))
            (let ((ob (build-cube)))
                (active-box ob)
                (set-mass ob (* (rndf) 10))
                (kick ob (vmul (srndvec) 3))
                (twist ob (vmul (srndvec) 2))))))
```

(gravity gravity-vector)

Returns void

Sets the strength and direction of gravity.

Example

```
(clear)
(ground-plane (vector 0 1 0) 0)
(collisions 1)
(set-max-physical 20)

(every-frame
    (begin
        (gravity (vector 0 (sin (time)) 0)) ; change gravity! :)
        (when (> (rndf) 0.92)
            (with-state
                (translate (vector 0 5 0))
                (scale (rndvec))
                (colour (rndvec))
                (let ((ob (build-cube)))
                    (active-box ob)
                    (kick ob (vmul (srndvec) 3))
                    (twist ob (vmul (srndvec) 2)))))))
```

(kick primitiveid-number kick-vector)

Returns void

Applies translation force to the object

Example

```
(clear)
(collisions 1)
(set-max-physical 20)
(gravity (vector 0 0 0))

(every-frame
    (when (> (rndf) 0.92)
        (with-state
            (scale (rndvec))
            (colour (rndvec))
            (let ((ob (build-cube)))
                (active-box ob)
                (kick ob (vmul (srndvec) 3))
                (twist ob (vmul (srndvec) 2))))))
```

(twist primitiveid-number spin-vector)

Returns void

Applies rotational force to the object

Example

```
(clear)
(collisions 1)
(set-max-physical 20)
(gravity (vector 0 0 0))

(every-frame
    (when (> (rndf) 0.92)
        (with-state
            (scale (rndvec))
            (colour (rndvec))
            (let ((ob (build-cube)))
                (active-box ob)
                (kick ob (vmul (srndvec) 3))
                (twist ob (vmul (srndvec) 2))))))
```

(has-collided primitiveid-number)

Returns void

Returns true if the grabbed object collided in the last frame

Example

```
(clear)
(ground-plane (vector 0 1 0) 0)
(collisions 1)
(set-max-physical 20)

(define ob (with-state
    (translate (vector 0 5 0))
    (build-cube)))

(active-box ob)

(every-frame
    (when (has-collided ob)
        (with-primitive ob
            (colour (rndvec)))))
```

maths

Description

These functions are optimised for 3D graphics, and the collision of computer science and maths is apparent here, so scheme vectors representing maths vectors are in this context taken to be 3 elements long, quaternions are vectors of length 4, and matrices are vectors of 16 elements long.

(vmul vector number)

Returns result-vector

Multiplies a vector by a number

Example

```
(vmul (vector 1 2 3) 2)
```

(vadd vector vector)

Returns result-vector

Adds two vectors together

Example

```
(vadd (vector 1 2 3) (vector 1 2 3))
```

(vsub vector vector)

Returns result-vector

Subtracts a vector from another

Example

```
(vsub (vector 1 2 3) (vector 1 2 3))
```

(vdiv vector number)

Returns result-vector

Divides a vector by a number

Example

```
(vdiv (vector 1 2 3) 2)
```

(vtransform vector matrix)

Returns result-vector

Multiplies (transforms) a vector by a matrix

Example

```
(vtransform (vector 0 1 0) (mrotate (vector 90 0 0)))
```

(vtransform-rot vector matrix)

Returns result-vector

Multiplies (transforms) a vector by a matrix, but leaves out the translation part.

For operations involving normals.

Example

```
(vtransform-rot (vector 0 1 0) (mrotate (vector 90 0 0)))
```

(vnormalise vector)

Returns result-vector

Returns the normalised form of the vector (length=1)

Example

```
(vtransform-rot (vector 0 1 0) (mrotate (vector 90 0 0)))
```

(vdot vector vector)

Returns result-number

Returns the dot product of two vectors

Example

```
(vdot (vector 0 1 0) (vector 1 0 0))
```

(vmag vector)

Returns result-number

Returns the magnitude, or length of the vector

Example

```
(vmag (vector 0 1 1))
```

(vreflect vector vector)

Returns result-vector

Returns the reflection of one vector against another.

Example

```
(vreflect (vector 0 1 1) (vector 1 0 1))
```

(vdist vector vector)

Returns result-number

Treating the vectors as points, returns the distance between them.

Example

```
(vdist (vector 100 100 0) (vector 0 0 100))
```

(vdist-sq vector vector)

Returns result-number

Treating the vectors as points, returns the squared distance between them. Faster than vdist.

Example

```
(vdist-sq (vector 100 100 0) (vector 0 0 100))
```

(vcross vector vector)

Returns result-vector

Returns the cross product of two vectors, resulting in a vector that is perpendicular to the crossed ones.

Example

```
(vcross (vector 100 100 0) (vector 0 0 100))
```

(mmul matrix-vector matrix-vector)

Returns matrix-vector

Multiplies two matrices together

Example

```
(mmul (mtranslate (vector 1 0 0)) (mrotate (vector 0 90 0)))
```

(madd matrix-vector matrix-vector)

Returns matrix-vector

Adds two matrices together

Example

```
(madd (mtranslate (vector 1 0 0)) (mrotate (vector 0 90 0)))
```

(msub matrix-vector matrix-vector)

Returns matrix-vector

Subtracts a matrix from another.

Example

```
(msub (mtranslate (vector 1 0 0)) (mrotate (vector 0 90 0)))
```

(mdiv matrix-vector matrix-vector)

Returns matrix-vector

Divides a matrix by another

Example

```
(mdiv (mtranslate (vector 1 0 0)) (mrotate (vector 0 90 0)))
```

(mident)

Returns matrix-vector

Returns the identity matrix

Example

```
(mident)
```

(mtranslate vector)

Returns matrix-vector

Returns a matrix representing the specified transform

Example

```
(mtranslate (vector 100 0 0))
```

(mrotate vector)

Returns matrix-vector

Returns a matrix representing the specified rotation. Accepts a vector of euler angles, or a quaternion.

Example

```
(mrotate (vector 0 45 0))
```

(mscale vector)

Returns matrix-vector

Returns a matrix representing the specified scaling.

Example

```
(mscale (vector 0.5 2 0.5))
```

(mtranspose matrix-vector)

Returns matrix-vector

Returns the transpose of the input vector

Example

```
(mtranspose (mident))
```

(minverse matrix-vector)

Returns matrix-vector

Returns the inverse of the input vector.

Example

```
(minverse (mscale (vector 0.5 2 0.5)))
```

(maim aim-vector up-vector)

Returns matrix-vector

Returns a matrix representing an aiming rotation so that the x axis points down the aim direction, and the y axis points up the up vector. Probably suffers from gimbal lock.

Example

```
(maim (vector 0 0 1) (vector 0 1 0))
```

(qaxisangle axis-vector angle-number)

Returns quaternion-vector

Returns the quaternion representing rotation of angle degrees about the specified axis.

Example

```
(qaxisangle (vector 0 1 0) 45)
```

(qmul quaternion-vector quaternion-vector)

Returns quaternion-vector

Multiplies two quaternions together.

Example

```
(qmul (qaxisangle (vector 0 1 0) 45) (qaxisangle (vector 0 0 1) 180))
```

(qnormalise quaternion-vector)

Returns quaternion-vector

Normalises a quaternion.

Example

```
(qnormalise (qaxisangle (vector 0 19 0) 45))
```

(qtomatrix quaternion-vector)

Returns matrix-vector

Converts a quaternion into a rotation matrix.

Example

```
(qtomatrix (qaxisangle (vector 0 1 0) 45))
```

(qconjugate quaternion-vector)

Returns quaternion-vector

Conjugatea a quaternion.

Example

```
(qconjugate (qaxisangle (vector 0 1 0) 45))
```

(fmod numerator-number denominator-number)

Returns real-number

Returns the floating-point remainder of numerator/denominator.

Example

```
(fmod 14.4 10)
```

(snoise real-number ...)

Returns real-number

Returns 1D/2D/3D/4D Simplex Noise in the range -1->1 depending on the number of parameters.

Example

```
(snoise 1.0 2.0) ; 2D noise
(snoise 6.1 2.4 .5 1.3) ; 4D noise

; example on a pixel prim
(clear)
(with-primitive (build-pixels 100 100)
    (pdata-index-map!
        (lambda (i c)
            (snoise (* 0.1 (modulo i (pixels-width)))
                    (* 0.1 (quotient i (pixels-height)))))
        "c")
    (pixels-upload))
```

(noise real-number ...)

Returns real-number

Returns the Perlin Noise value at specified coordinates.

Example

```
(noise 1.0 2.0) ; 2D noise
(noise 6.1 2.4 .5) ; 3D noise

; example on a pixel prim
(clear)
(with-primitive (build-pixels 100 100)
    (pdata-index-map!
        (lambda (i c)
            (noise (* 0.1 (modulo i (pixels-width)))
                   (* 0.1 (quotient i (pixels-height)))))
        "c")
    (pixels-upload))
```

(noise-seed unsigned-number)

Returns void

Sets the seed value for noise.

Example

```
(noise-seed 1)
```

(noise-detail octaves-number falloff-number)

Returns void

Adjusts the character and level of detail produced by the Perlin noise function.

Example

```
(noise-detail 4) ; noise with 4 octaves
(noise-detail 4 .5) ; noise with 4 octaves and .5 falloff
```

lights

Description

Without lights you wouldn't be able to see anything. Luckily fluxus gives you one for free by default, a white diffuse point light attached to the camera. For more interesting lighting, you'll need these functions. Using the standard fixed function graphics pipeline, simplistically speaking, OpenGL multiplies these values with the surface material (set with local state commands like ambient and diffuse) and the texture colour value to give the final colour.

Example

```
; turn off the main light
(light-diffuse 0 (vector 0 0 0))
```

```
(light-specular 0 (vector 0 0 0))
(light-ambient 0 (vector 0 0 0))

(define mylight (make-light 'point 'free))
(light-position mylight (vector 5 2 0))
(light-diffuse mylight (rndvec))
(light-ambient mylight (vmul (rndvec) 0.1))
(light-specular mylight (vmul (rndvec) 10))

(with-state
    (ambient (vector 1 1 1))
    (colour (vector 1 1 1))
    (specular (vector 0.5 0.5 0.5))
    (shinyness 20)
    (build-torus 1 2 20 20))
```

(make-light type-symbol cameralocked-symbol)

Returns lightid-number

Makes a new light. The type can be one of: point, directional or spot. If the cameralocked string is not free then it will be attached to the camera, and move around when you move the camera.

Example

```
; turn off the main light
(light-diffuse 0 (vector 0 0 0))
(light-specular 0 (vector 0 0 0))
(light-ambient 0 (vector 0 0 0))

(define mylight (make-light 'point 'free))
(light-position mylight (vector 5 2 0))
(light-diffuse mylight (rndvec))
(light-ambient mylight (vmul (rndvec) 0.1))
(light-specular mylight (vmul (rndvec) 10))

(with-state
    (ambient (vector 1 1 1))
    (colour (vector 1 1 1))
    (specular (vector 0.5 0.5 0.5))
    (shinyness 20)
    (build-torus 1 2 20 20))
```

(light-ambient lightid-number colour)

Returns void

Sets the ambient contribution for the specified light.

Example

```
; turn off the main light
(light-diffuse 0 (vector 0 0 0))
(light-specular 0 (vector 0 0 0))
(light-ambient 0 (vector 0 0 0))
```

```
(define mylight (make-light 'point 'free))
(light-position mylight (vector 5 2 0))
(light-diffuse mylight (rndvec))
(light-ambient mylight (vmul (rndvec) 0.1))
(light-specular mylight (vmul (rndvec) 10))

(with-state
    (ambient (vector 1 1 1))
    (colour (vector 1 1 1))
    (specular (vector 0.5 0.5 0.5))
    (shinyness 20)
    (build-torus 1 2 20 20))
```

(light-diffuse lightid-number colour)

Returns void

Sets the diffuse contribution for the specified light.

Example

```
; turn off the main light
(light-diffuse 0 (vector 0 0 0))
(light-specular 0 (vector 0 0 0))
(light-ambient 0 (vector 0 0 0))

(define mylight (make-light 'point 'free))
(light-position mylight (vector 5 2 0))
(light-diffuse mylight (rndvec))
(light-ambient mylight (vmul (rndvec) 0.1))
(light-specular mylight (vmul (rndvec) 10))

(with-state
    (ambient (vector 1 1 1))
    (colour (vector 1 1 1))
    (specular (vector 0.5 0.5 0.5))
    (shinyness 20)
    (build-torus 1 2 20 20))
```

(light-specular lightid-number colour)

Returns void

Sets the specular contribution for the specified light.

Example

```
; turn off the main light
(light-diffuse 0 (vector 0 0 0))
(light-specular 0 (vector 0 0 0))
(light-ambient 0 (vector 0 0 0))

(define mylight (make-light 'point 'free))
(light-position mylight (vector 5 2 0))
(light-diffuse mylight (rndvec))
(light-ambient mylight (vmul (rndvec) 0.1))
```

```
(light-specular mylight (vmul (rndvec) 10))

(with-state
    (ambient (vector 1 1 1))
    (colour (vector 1 1 1))
    (specular (vector 0.5 0.5 0.5))
    (shinyness 20)
    (build-torus 1 2 20 20))
```

(light-position lightid-number position-vector)

Returns void

Sets the position of the specified light. In worldspace if free, in camera space is attached.

Example

```
; turn off the main light
(light-diffuse 0 (vector 0 0 0))
(light-specular 0 (vector 0 0 0))
(light-ambient 0 (vector 0 0 0))

(define mylight (make-light 'point 'free))
(light-position mylight (vector 5 2 0))
(light-diffuse mylight (rndvec))
(light-ambient mylight (vmul (rndvec) 0.1))
(light-specular mylight (vmul (rndvec) 10))

(with-state
    (ambient (vector 1 1 1))
    (colour (vector 1 1 1))
    (specular (vector 0.5 0.5 0.5))
    (shinyness 20)
    (build-torus 1 2 20 20))
```

(light-spot-angle lightid-number angle-number)

Returns void

Sets the spotlight cone angle of the specified light. If it's not a spot light, this command has no effect.

Example

```
; turn down the main light
(light-diffuse 0 (vector 0.1 0.1 0.1))
(light-specular 0 (vector 0 0 0))
(light-ambient 0 (vector 0 0 0))

(define mylight (make-light 'spot 'free))
(light-position mylight (vector (+ 4 (crndf)) (crndf) 2))
(light-diffuse mylight (rndvec))
(light-ambient mylight (vmul (rndvec) 0.1))
(light-specular mylight (vmul (rndvec) 10))
(light-spot-angle mylight (+ 5 (random 40)))
```

```
(light-spot-exponent mylight 500)
(light-attenuation mylight 'constant 1)
(light-direction mylight (vector -1 0 -1))

(with-state
    (ambient (vector 1 1 1))
    (colour (vector 1 1 1))
    (specular (vector 0.5 0.5 0.5))
    (shinyness 20)
    (build-torus 1 2 20 20)
    (scale (vector 10 10 10))
    (translate (vector -0.5 -0.5 0))
    (build-seg-plane 20 20))
```

(light-spot-exponent lightid-number exponent-number)

Returns void

Sets the spotlight exponent (fuzzyness of the cone) of the specified light. If it's not a spot light, this command has no effect.

Example

```
; turn down the main light
(light-diffuse 0 (vector 0.1 0.1 0.1))
(light-specular 0 (vector 0 0 0))
(light-ambient 0 (vector 0 0 0))

(define mylight (make-light 'spot 'free))
(light-position mylight (vector (+ 4 (crndf)) (crndf) 2))
(light-diffuse mylight (rndvec))
(light-ambient mylight (vmul (rndvec) 0.1))
(light-specular mylight (vmul (rndvec) 10))
(light-spot-angle mylight (+ 5 (random 40)))
(light-spot-exponent mylight 500)
(light-attenuation mylight 'constant 1)
(light-direction mylight (vector -1 0 -1))

(with-state
    (ambient (vector 1 1 1))
    (colour (vector 1 1 1))
    (specular (vector 0.5 0.5 0.5))
    (shinyness 20)
    (build-torus 1 2 20 20)
    (scale (vector 10 10 10))
    (translate (vector -0.5 -0.5 0))
    (build-seg-plane 20 20))
```

(light-attenuation lightid-number type-symbol attenuation-number)

Returns void

Sets the light attenuation (fade off with distance) of the specified light. The type symbol can be one of: constant, linear or quadratic.

Example

```
; turn down the main light
(light-diffuse 0 (vector 0.1 0.1 0.1))
(light-specular 0 (vector 0 0 0))
(light-ambient 0 (vector 0 0 0))

(define mylight (make-light 'spot 'free))
(light-position mylight (vector (+ 4 (crndf)) (crndf) 2))
(light-diffuse mylight (rndvec))
(light-ambient mylight (vmul (rndvec) 0.1))
(light-specular mylight (vmul (rndvec) 10))
(light-spot-angle mylight (+ 5 (random 40)))
(light-spot-exponent mylight 500)
(light-attenuation mylight 'constant 1)
(light-direction mylight (vector -1 0 -1))

(with-state
    (ambient (vector 1 1 1))
    (colour (vector 1 1 1))
    (specular (vector 0.5 0.5 0.5))
    (shinyness 20)
    (build-torus 1 2 20 20)
    (scale (vector 10 10 10))
    (translate (vector -0.5 -0.5 0))
    (build-seg-plane 20 20))
```

(light-direction lightid-number direction-vector)

Returns void

Sets the direction of a directional light. If it's not a directional light, this command has no effect.

Example

```
; turn down the main light
(light-diffuse 0 (vector 0.1 0.1 0.1))
(light-specular 0 (vector 0 0 0))
(light-ambient 0 (vector 0 0 0))

(define mylight (make-light 'spot 'free))
(light-position mylight (vector (+ 4 (crndf)) (crndf) 2))
(light-diffuse mylight (rndvec))
(light-ambient mylight (vmul (rndvec) 0.1))
(light-specular mylight (vmul (rndvec) 10))
(light-spot-angle mylight (+ 5 (random 40)))
(light-spot-exponent mylight 500)
(light-attenuation mylight 'constant 1)
(light-direction mylight (vector -1 0 -1))

(with-state
    (ambient (vector 1 1 1))
    (colour (vector 1 1 1))
    (specular (vector 0.5 0.5 0.5))
    (shinyness 20)
    (build-torus 1 2 20 20)
    (scale (vector 10 10 10))
    (translate (vector -0.5 -0.5 0))
    (build-seg-plane 20 20))
```

renderer

Description

These commands are the low level renderer controls. You shouldn't need to deal with these unless you are being wily, or implementing a fluxus renderer outside of the scratchpad interface.

(make-renderer)

Returns rendererid-number

Makes a new scenegraph renderer.

Example

```
(make-renderer)
```

(renderer-grab rendererid-number)

Returns void

Make this renderer the current context for commands.

Example

```
(renderer-grab renderer)
```

(renderer-ungrab)

Returns void

Pop the renderer context stack.

Example

```
(renderer-grab renderer)
```

(fluxus-render)

Returns void

Clears the backbuffer, and renders everything

Example

```
(fluxus-render)
```

(tick-physics)

Returns void

Update the physics system.

Example

```
(tick-physics)
```

(render-physics)

Returns void

Render the physics system (for helper graphics). This is the low level command - use set-physics-debug instead.

Example

```
(render-physics)
```

(reset-renderers)

Returns void

Deletes all the renderers and makes a new default one.

Example

```
(reset-renderers)
```

(reshape width-number height-number)

Returns void

Calls reshape on the current renderer

Example

```
(reshape 100 100)
```

(fluxus-init)

Returns void

Inits the whole rendering system, only needs calling once.

Example

```
(fluxus-init)
```

(fluxus-error-log)

Returns void

Returns a string containing error information for the last frame.

Example

```
 (display (fluxus-error-log))
```

audio

Description

This part of fluxus is responsible for capturing the incoming sound, and processing it into harmonic data, using fft (Fast Fourier Transform). The harmonics are bands of frequency which the sound is split into, giving some indication of the quality of the sound. It's the same as you see on a graphic equaliser - in fact, one of the example scripts (bars.scm) acts as a graphic equaliser display, and should be used to test the audio is working.

Example

```
 (start-audio "alsa_pcm:capture_1" 1024 44100)
 (define (animate)
                (colour (vector (gh 1) (gh 2) (gh 3))) ; make a colour from the
harmonics, and set it to be the current colour
                (draw-cube)) ; draw a cube with this colour
 (every-frame (animate))
```

(start-audio jackport-string buffersize-number samplerate-number)

Returns void

Starts up the audio with the specified settings, you'll need to call this first, or put it into $HOME/.fluxus.scm to call it automatically at startup. Make the jack port name an empty string and it won't try to connect to anything for you. You can use qjackctrl or equivelent to do the connection manually. Fluxus reads a single mono source.

Example

```
 (start-audio "alsa_pcm:capture_1" 1024 44100)
```

(gh harmonic-number)

Returns harmonic-real

Fluxus converts incoming audio into harmonic frequencies, which can then be plugged into your animations using this command. There are 16 harmonic bands availible, the harmonic-value argument will be wrapped around if greater or less than 16, so you can use this command without worrying about out of range errors.

Example

```
 (define (animate)
               (colour (vector (gh 1) (gh 2) (gh 3))) ; make a colour from the
harmonics, and set it to be the current colour
               (draw-cube)) ; draw a cube with this colour
 (every-frame (animate))
```

(gain gain-number)

Returns void

Sets the gain level for the fft sound, it's 1 by default.

Example

```
 (gain 100) ; too quiet?!
```

(process wavfile-string)

Returns void

This command temporarally disables the realtime reading of the input audio stream and reads a wav file instead. For use with the framedump command to process audio offline to make music videos. The advantage of this is that it locks the framerate so the right amount of audio gets read for each frame - making syncing of the frames and audio files possible.

Example

```
 (process "somemusic.wav") ; read a precorded audio file
```

(smoothing-bias value-number)

Returns void

A kind of weighted average for the harmonic bands which smooth them out over time. This setting defaults to 1.5. The best value really depends on the quality of the music, and the buffer sizes, and ranges from 0 -> 2. It's more obvious if you give it a try with the bars.scm script

Example

```
 (smoothing-bias 0) ; no smoothing
```

(update-audio)

Returns void

Updates the audio subsytem. This function is called for you (per frame) in fluxus-canvas.ss.

Example

```
 (update-audio)
```

global-state

Description

Global state is really anything that controls the renderer globally, so it affects all primitives or controls the renderer directly - ie camera control or full screen effects like blurring.

(clear-engine)

Returns void

Clears the renderer, and physics system. This command should not be called directly, use clear instead, as this clears a few other things, and calls clear-engine itself.

Example

```
(clear-engine) ; woo hoo!
```

(blur amount-number)

Returns void

Sets the full screen blur setting. Less is more, but if you set it too low it will make the on screen editing impossible to read, so save your script first :)

Example

```
(blur 0.1) ; for nice trails
```

(fog fogcolour-vector amount-number begin-number end-number)

Returns void

Sets the fogging parameters to give a visual depth cue (aerial perspective in painter's jargon). This can obscure the on screen editing, so keep the amount small.

Example

```
(clear-colour (vector 0 0 1))   ; looks nice if the background matches
(fog (vector 0 0 1) 0.01 1 100) ; blue fog
```

(show-axis show-number)

Returns void

Shows the worldspace origin axis used.

Example

```
(show-axis 1)
```

(show-fps show-number)

Returns void

Shows an fps count in the lower left of the screen. used.

Example

```
 (show-fps 1)
```

(lock-camera primitiveid-number)

Returns void

Locks the camera transform onto the specified primitive's transform. It's like parenting the camera to the object. This is the easiest way to procedurally drive the camera. Use an id number of 0 to unlock the camera.

Example

```
 (clear)
 (define obj (build-cube)) ; make a cube for the camera to lock to

 (with-state ; make a background cube so we can tell what's happening
     (hint-wire)
     (hint-unlit)
     (texture (load-texture "test.png"))
     (colour (vector 0.5 0.5 0.5))
     (scale (vector -20 -10 -10))
     (build-cube))

 (lock-camera obj) ; lock the camera to our first cube
 (camera-lag 0.1)  ; set the lag amount, this will smooth out the cube jittery
movement

 (define (animate)
     (with-primitive obj
         (identity)
         (translate (vector (fmod (time) 5) 0 0)))) ; make a jittery movement

 (every-frame (animate))
```

(camera-lag amount-number)

Returns void

The camera locking has an inbuilt lagging which means it will smoothly blend the movement relative to the primitive it's locked to.

Example

```
 (clear)
 (define obj (build-cube)) ; make a cube for the camera to lock to

 (with-state ; make a background cube so we can tell what's happening
     (hint-wire)
     (hint-unlit)
     (texture (load-texture "test.png"))
     (colour (vector 0.5 0.5 0.5))
     (scale (vector -20 -10 -10))
     (build-cube))

 (lock-camera obj) ; lock the camera to our first cube
 (camera-lag 0.1)  ; set the lag amount, this will smooth out the cube jittery
movement

 (define (animate)
     (with-primitive obj
         (identity)
         (translate (vector (fmod (time) 5) 0 0)))) ; make a jittery movement

 (every-frame (animate))
```

(load-texture pngfilename-string optional-create-params-list)

Returns textureid-number

Loads a texture from disk, converts it to a texture, and returns the id number. The texture loading is memory cached, so repeatedly calling this will not cause it to load again. The cache can be cleared with clear-texture-cache. The png may be RGB or RGBA to use alpha transparency. To get more control how your texture is created you can use a list of parameters. See the example for more explanation. Use id for adding more texture data to existing textures as mipmap levels, or cube map faces. Note: if turning mipmapping off and only specifing one texture it should be set to mip level 0, and you'll need to turn the min and mag filter settings to linear or nearest (see texture-params).

Example

```
 ; simple usage:
 (texture (load-texture "mytexture.png"))
 (build-cube) ; the cube will be texture mapped with the image

 ; complex usages:

 ; the options list can contain the following keys and values:
 ; id: texture-id-number (for adding images to existing textures - for
mipmapping and cubemapping)
 ; type: [texture-2d cube-map-positive-x cube-map-negative-x cube-map-positive-y
 ;        cube-map-negative-y cube-map-positive-z cube-map-negative-z]
 ; generate-mipmaps : exact integer, 0 or 1
 ; mip-level : exact integer
 ; border : exact integer

 ; setup an environment cube map
 (define t (load-texture "cube-left.png" (list 'type 'cube-map-positive-x)))
 (load-texture "cube-right.png" (list 'id t 'type 'cube-map-negative-x))
 (load-texture "cube-top.png" (list 'id t 'type 'cube-map-positive-y))
 (load-texture "cube-bottom.png" (list 'id t 'type 'cube-map-negative-y))
 (load-texture "cube-front.png" (list 'id t 'type 'cube-map-positive-z))
 (load-texture "cube-back.png" (list 'id t 'type 'cube-map-negative-z))
 (texture t)
 ; setup a mipmapped texture with our own images
 ; you need as many levels as it takes you to get to 1X1 pixels from your
 ; level 0 texture size
 (define t2 (load-texture "m0.png" (list 'generate-mipmaps 0 'mip-level 0)))
 (load-texture "m1.png" (list 'id t2 'generate-mipmaps 0 'mip-level 1))
 (load-texture "m2.png" (list 'id t2 'generate-mipmaps 0 'mip-level 2))
 (load-texture "m3.png" (list 'id t2 'generate-mipmaps 0 'mip-level 3))


 (texture (load-texture "mytexture.png"
                               (list
                                       'generate-mipmaps 0  ; turn mipmapping
off
                    'border 2)))          ; add a border to the texture

 (build-cube) ; the cube will be texture mapped with the image
```

(clear-texture-cache)

Returns void

Clears the texture cache, meaning changed textures on disk are reloaded.

Example

```
 (clear-texture-cache)
```

(frustum top-number bottom-number left-number right-number)

Returns void

Sets the camera frustum, and thus the aspect ratio of the frame.

Example

```
 (frustum -1 1 -0.75 0.75) ; default settings
```

(clip front-number back-number)

Returns void

Sets the front & back clipping planes for the camera frustum, and thus the viewing angle. Change the front clipping distance to alter the perspective from telephoto to fisheye.

Example

```
 (clip 1 10000) ; default settings
```

(ortho)

Returns void

Sets orthographic projection - i.e. no perspective.

Example

```
(ortho)
```

(persp)

Returns void

Sets perspective projection (the default) after ortho has been set.

Example

```
(persp)
```

(set-ortho-zoom amount-number)

Returns void

Sets the zoom level for the orthographic projection.

Example

```
(set-ortho-zoom 2)
```

(clear-colour colour-vector)

Returns void

Sets the colour we clear the renderer with, this forms the background colour for the scene.

Example

```
(clear-colour (vector 1 0 0)) ; RED!!!
```

(clear-frame setting-number)

Returns void

Sets the frame clearing on or off.

Example

```
(clear-frame 0)
(clear-frame 1)
```

(clear-zbuffer setting-number)

Returns void

Sets the zbuffer clearing on or off.

Example

```
(clear-zbuffer 0)
```

```
(clear-zbuffer 1)
```

(clear-accum setting-number)

Returns void

Sets the accumulation buffer clearing on or off.

Example

```
(clear-accum 1)
```

(build-camera)

Returns cameraid-number

Adds a new camera/view and returns it's id

Example

```
(clear)
(viewport 0 0.5 0.5 0.5)

(define cam2 (build-camera))
(current-camera cam2)
(viewport 0.5 0 0.5 1)

(define cam3 (build-camera))
(current-camera cam3)
(set-camera (mmul (mtranslate (vector 0 0 -5))
        (mrotate (vector 0 45 0))))
(viewport 0 0 0.5 0.5)

; render a primitive in one view only
(define t (with-state
    (translate (vector 3 0 0))
    (scale 0.3)
    (colour (vector 1 0 0))
    (build-torus 1 2 10 10)))

(with-primitive t
    (hide 1) ; hide in all
    (camera-hide 0)) ; unhide in current camera


(current-camera 0)

(define c (with-state
        (hint-cull-ccw)
        (hint-unlit)
        (hint-wire)
        (line-width 2)
        (colour (vector 0.4 0.3 0.2))
        (wire-colour (vector 0 0 0))
        (scale 10)
        (build-cube)))
```

```
(define p (with-state
        (scale 3)
        (load-primitive "widget.obj")))

(every-frame
    (with-primitive p
        (rotate (vector 0 1 0))))
```

(current-camera cameraid-number)

Returns void

Sets the current camera to use

Example

```
(clear)
(viewport 0 0.5 0.5 0.5)

(define cam2 (build-camera))
(current-camera cam2)
(viewport 0.5 0 0.5 1)

(define cam3 (build-camera))
(current-camera cam3)
(set-camera (mmul (mtranslate (vector 0 0 -5))
        (mrotate (vector 0 45 0))))
(viewport 0 0 0.5 0.5)

; render a primitive in one view only
(define t (with-state
    (translate (vector 3 0 0))
    (scale 0.3)
    (colour (vector 1 0 0))
    (build-torus 1 2 10 10)))

(with-primitive t
    (hide 1) ; hide in all
    (camera-hide 0)) ; unhide in current camera


(current-camera 0)

(define c (with-state
        (hint-cull-ccw)
        (hint-unlit)
        (hint-wire)
        (line-width 2)
        (colour (vector 0.4 0.3 0.2))
        (wire-colour (vector 0 0 0))
        (scale 10)
        (build-cube)))

(define p (with-state
        (scale 3)
        (load-primitive "widget.obj")))
```

```
(every-frame
    (with-primitive p
        (rotate (vector 0 1 0))))
```

(viewport x-number y-number width-number height-number)

Returns void

Sets the viewport on the current camera. This is the area of the window the camera renders to, where 0,0 is the bottom left and 1,1 is the top right.

Example

```
(clear)
(viewport 0 0.5 0.5 0.5)

(define cam2 (build-camera))
(current-camera cam2)
(viewport 0.5 0 0.5 1)

(define cam3 (build-camera))
(current-camera cam3)
(set-camera (mmul (mtranslate (vector 0 0 -5))
        (mrotate (vector 0 45 0))))
(viewport 0 0 0.5 0.5)

; render a primitive in one view only
(define t (with-state
    (translate (vector 3 0 0))
    (scale 0.3)
    (colour (vector 1 0 0))
    (build-torus 1 2 10 10)))

(with-primitive t
    (hide 1) ; hide in all
    (camera-hide 0)) ; unhide in current camera


(current-camera 0)

(define c (with-state
        (hint-cull-ccw)
        (hint-unlit)
        (hint-wire)
        (line-width 2)
        (colour (vector 0.4 0.3 0.2))
        (wire-colour (vector 0 0 0))
        (scale 10)
        (build-cube)))

(define p (with-state
        (scale 3)
        (load-primitive "widget.obj")))

(every-frame
    (with-primitive p
        (rotate (vector 0 1 0))))
```

(get-camera)

Returns matrix-vector

Gets the current camera transform matrix. This is the low level function, use get-camera-transform instead.

Example

```
(get-camera)
```

(get-locked-matrix)

Returns matrix-vector

Gets the current camera lock transform matrix. Takes the lag into account

Example

```
(get-locked-matrix)
```

(set-camera)

Returns void

Sets the camera transform matrix. This is the low level interface used by set-camera-transform, which you should generally use instead.

Example

```
(set-camera (mtranslate (vector 0 0 -10)))
```

(get-projection-transform)

Returns projection-matrix

Gets the current projection matrix.

Example

```
(get-projection-transform)
```

(get-screen-size)

Returns size-vector

Returns a vector containing the current width and height of the window.

Example

```
(get-screen-size)
```

(set-screen-size size-vector)

Returns void

Sets the window width and height.

Example

```
(set-screen-size (vector 10 10)) ; small window time :)
(set-screen-size (vector 720 576)) ; and back again!
```

(select screenxpos-number screenypos-number pixelssize-number)

Returns primitiveid-number

Looks in the region specified and returns the id of the closest primitive to the camera rendered there, or 0 if none exist.

Example

```
(display (select 10 10 2))(newline)
```

(desiredfps fps-number)

Returns void

Throttles the renderer so as to not take 100% cpu. This gives an upper limit on the fps rate, which doesn't quite match the given number, but I'm working on it...

Example

```
(desiredfps 100000) ; makes fluxus render as fast as it can, and take 100% cpu.
```

(draw-buffer buffer_name)

Returns void

Select which buffer to draw in for stereo mode you'd do 'back-right and 'back-left

Example

```
(draw-buffer 'back)
```

(read-buffer buffer_name)

Returns void

Select which buffer to read from

Example

```
(read-buffer 'back)
```

(set-stereo-mode mode)

Returns bool

select which stereo mode to use currently only 'crystal-eyes and 'no-stereo are supported the return indicates if the operation was successful or not 'crystal-eyes will return false if you don't have a stereo window

Example

```
(set-stereo-mode 'crystal-eyes)
```

(set-colour-mask vector)

Returns void

sets the colour mask give it a quat of booleans which correspond to the red, green, blue and alpha channels respectively after this operation you'll only see those colour which you set to true (this is useful for stereo with red-blue glasses)

Example

```
(set-colour-mask #(#t #f #f #t))
```

(shadow-light number-setting)

Returns void

Sets the light to use for generating shadows, set to 0 to disable shadow rendering.

Example

```
(shadow-light 1)
```

(shadow-length number-setting)

Returns void

Sets the length of the shadow volume rendering.

Example

```
(shadow-length 10)
```

(shadow-debug number-setting)

Returns void

Turns on debug rendering of the shadow volume rendering.

Example

```
(shadow-debug 1)
```

(accum mode-symbol value-number)

Returns void

Controls the accumulation buffer (just calls glAccum under the hood). Possible symbols are: accum load return add mult

Example

```
(accum 'add 1)
```

(print-info)

Returns void

Prints out a load of renderer information

Example

```
(print-info)
```

(set-cursor image-name-symbol)

Returns void

Changes the mouse cursor. Cursor image name symbols can consist of: 'right-arrow, 'left-arrow, 'info, 'destroy, 'help, 'cycle, 'spray, 'wait, 'text, 'crosshair, 'up-down, 'left-right, 'top-side, 'bottom-side, 'left-side, 'right-side, 'top-left-corner, 'top-right-corner, 'bottom-right-corner, 'bottom-left-corner, 'full-crosshair, 'none, 'inherit The default cursor image symbol when the window is created is 'inherit.

Example

```
(set-cursor 'crosshair)
```

local-state

Description

The local state functions control rendering either for the current state - or the state of the current primitive. In fluxus state means the way that things are displayed, either turning on and off rendering features, changing the style of different features, or altering the current transform.

(push)

Returns void

Pushes a copy of the current drawing state to the top of the stack. The drawing state contains information about things like the current colour, transformation and hints. This function has been superseded by (with-state).

Example

```
(colour (vector 1 0 0)) ; set current colour to red
(push)                  ; copy and push drawing state
(colour (vector 0 1 0)) ; set current colour to green
(draw-cube)             ; draws a green cube
(pop)                             ; forget old drawing state
; current colour is now red again
```

(pop)

Returns void

Destroys the current drawing state, and sets the current one to be the previously pushed one in the stack. The drawing state contains information about things like the current colour, transformation and hints. This function has been superseded by (with-state).

Example

```
(colour (vector 1 0 0)) ; set current colour to red
(push)                  ; copy and push drawing state
(colour (vector 0 1 0)) ; set current colour to green
(draw-cube)             ; draws a green cube
(pop)                   ; forget old drawing state
; current colour is now red again
```

(grab object-id)

Returns void

Grabs the specified object. Once an object has grabbed it's state can be modified using the same commands used to set the current drawing state. (ungrab) needs to be used to return to the normal drawing state. Grabbing can also be stacked, in which case ungrab pops to the last grabbed primitive. This function has been superseded by (with-primitive).

Example

```
(colour (vector 1 0 0))      ; set the current colour to red
(define mycube (build-cube)) ; makes a red cube
(grab mycube)
(colour (vector 0 1 0)) ; sets the cubes colour to green
(ungrab)                                  ; return to normal state
```

(ungrab)

Returns void

Ungrabs the current object, and either returns to the normal drawing state, or pops to the last grabbed primitive. This function has been superseded by (with-primitive).

Example

```
 (colour (vector 1 0 0))      ; set the current colour to red
 (define mycube (build-cube)) ; makes a red cube
 (grab mycube)
 (colour (vector 0 1 0)) ; sets the cubes colour to green
 (ungrab)                                  ; return to normal state
```

(apply-transform optional-object-id)

Returns void

Applies the current object transform to the vertex positions of the current object and sets it's transform to identity. Will also use the optional id passed in for the aniquated version of this command

Example

```
 (rotate (vector 45 0 0))
 (define mycube (build-cube)) ; makes a cube with a rotation
 (with-primitive mycube (apply-transform)) ; applies the rotation to the points
of the cube
```

(opacity value)

Returns void

Sets the opacity of the current drawing state, or the current primitive.

Example

```
 (opacity 0.5)
 (define mycube (build-cube)) ; makes a half transparent cube
```

(wire-opacity value)

Returns void

Sets the wireframe opacity of the current drawing state, or the current primitive.

Example

```
 (hint-none)
 (hint-wire)
 (backfacecull 0)
 (line-width 5)
 (wire-colour (vector 1 1 1))
 (wire-opacity 0.5)
 (build-cube) ; makes a half transparent wireframe cube
```

(shinyness value)

Returns void

Sets the shinyness of the current drawing state, or the current primitive. This value sets the tightness of the specular highlight.

Example

```
(shinyness 100)
(specular (vector 1 1 1)) ; sets the specular colour
(define mysphere (build-sphere 10 10)) ; makes a shiny cube
```

(colour colour-vector)

Returns void

Sets the colour of the current drawing state, or the current primitive.

Example

```
(colour (vector 1 0.5 0.1)) ; mmm orange...
(define mycube (build-cube)) ; makes an orange cube
```

(colour-mode mode)

Returns void

Changes the way Fluxus interprets colour data for the current drawing state, or the current primitive. Colourmode symbols can consist of: rgb hsv

Example

```
(clear)
(colour-mode 'hsv)

(for ((x (in-range 0 10)))
  (translate (vector 1 0 0))
  (colour (vector (/ x 10) 1 1))
  (build-cube))
```

(rgb->hsv colour-vector)

Returns vector

Converts the RGB colour to HSV.

Example

```
(rgb->hsv (vector 1 0.5 0.1))
```

(hsv->rgb colour-vector)

Returns vector

Converts the HSV colour to RGB.

Example

```
(clear)
(for* ((x (in-range 0 10))  ; builds a 10x10 HSV colour pattern
       (y (in-range 0 10)))
    (identity)
    (translate (vector x y 0))
    (colour (hsv->rgb (vector (/ x 10) (/ y 10) 1)))
    (build-cube))
```

(wire-colour colour-vector)

Returns void

Sets the wire frame colour of the current drawing state, or the current primitive. Visible with (hint-wire) on most primitives.

Example

```
(wire-colour (vector 1 1 0)) ; set yellow as current wire colour
(hint-wire)
(define mycube (build-cube)) ; makes a cube with yellow wireframe
```

(specular colour-vector)

Returns void

Sets the specular colour of the current drawing state, or the current primitive.

Example

```
(specular (vector 0 0 1)) ; set blue as specular colour
(define mysphere (build-sphere 10 10)) ; makes a shiny blue sphere
```

(ambient colour-vector)

Returns void

Sets the ambient colour of the current drawing state, or the current primitive.

Example

```
(ambient (vector 0 0 1)) ; set blue as ambient colour
(define mysphere (build-sphere 10 10)) ; makes a boringly blue sphere
```

(opacity value)

Returns void

Sets the emissive colour of the current drawing state, or the current primitive.

Example

```
(emissive (vector 0 0 1)) ; set blue as emissive colour
(define mysphere (build-sphere 10 10)) ; makes an bright blue sphere
```

(identity)

Returns void

Sets the drawing state transform to identity, on the state stack, or the current primitive.

Example

```
 (define mycube (with-state
     (scale (vector 2 2 2)) ; set the current scale to double in each dimension
     (build-cube))) ; make a scaled cube

 (with-primitive mycube
     (identity)) ; erases the transform and puts the cube back to its original
state
```

(concat matrix)

Returns void

Concatenates (multiplies) a matrix on to the current drawing state or current primitive.

Example

```
 (define mymatrix (mrotate (vector 0 45 0))) ; make a matrix
 (concat mymatrix) ; concat it into the current state
 (build-cube) ; make a cube with this rotation
```

(translate vector)

Returns void

Applies a translation to the current drawing state transform or current primitive.

Example

```
 (translate (vector 0 1.4 0)) ; translates the current transform up a bit
 (build-cube) ; build a cube with this transform
```

(rotate vector-or-quaternion)

Returns void

Applies a rotation to the current drawing state transform or current primitive.

Example

```
 (rotate (vector 0 45 0)) ; turns 45 degrees in the Y axis
 (build-cube) ; build a cube with this transform
```

(scale vector)

Returns void

Applies a scale to the current drawing state transform or current primitive.

Example

```
(scale (vector 0.5 0.5 0.5)) ; scales the current transform to half the size
(build-cube) ; build a cube with this transform
```

(get-transform)

Returns matrix-vector

Returns a matrix representing the current state transform or for the current primitive.

Example

```
(clear)
; build a hierarchy
(define a
    (with-state
        (colour (vector 1 0.5 0.5))
        (build-cube)))
(define b (with-state
        (colour (vector 0.5 1 0.5))
        (parent a)
        (translate (vector 2 0 0))
        (build-cube)))
(define c (with-state
        (colour (vector 0.5 0.5 1))
        (parent b)
        (translate (vector 2 0 0))
        (build-cube)))

(define (animate)
    ; animate the heirarchy
    (with-primitive a (rotate (vector 0 0 (sin (time)))))
    (with-primitive b (rotate (vector 0 0 (sin (time)))))
    (with-primitive c (rotate (vector 0 0 (sin (time)))))

    ; position a yellow sphere with c's local transform
    (with-state
        (concat (with-primitive c (get-transform)))
        (opacity 0.5)
        (colour (vector 1 1 0))
        (draw-sphere))

    ; position a purple sphere with c's global transform
    (with-state
        (concat (with-primitive c (get-global-transform)))
        (opacity 0.5)
        (colour (vector 1 0 1))
        (draw-sphere)))

(every-frame (animate))
```

(get-global-transform)

Returns matrix-vector

Returns a matrix representing the current global transform for the current primitive.

Example

```
(clear)
; build a hierarchy
(define a
    (with-state
        (colour (vector 1 0.5 0.5))
        (build-cube)))
(define b (with-state
        (colour (vector 0.5 1 0.5))
        (parent a)
        (translate (vector 2 0 0))
        (build-cube)))
(define c (with-state
        (colour (vector 0.5 0.5 1))
        (parent b)
        (translate (vector 2 0 0))
        (build-cube)))

(define (animate)
    ; animate the heirarchy
    (with-primitive a (rotate (vector 0 0 (sin (time)))))
    (with-primitive b (rotate (vector 0 0 (sin (time)))))
    (with-primitive c (rotate (vector 0 0 (sin (time)))))

    ; position a yellow sphere with c's local transform
    (with-state
        (concat (with-primitive c (get-transform)))
        (opacity 0.5)
        (colour (vector 1 1 0))
        (draw-sphere))

    ; position a purple sphere with c's global transform
    (with-state
        (concat (with-primitive c (get-global-transform)))
        (opacity 0.5)
        (colour (vector 1 0 1))
        (draw-sphere)))

(every-frame (animate))
```

(parent primitive-id)

Returns void

Parents the current primitive to the supplied parent primitive. The current primitive will now be moved around with the parent by aquiring all the parent's transforms.

Example

```
 (define parent-prim (build-cube)) ; make a parent cube
 (translate (vector 2 0 0)) ; move a bit in x
 (parent parent-prim) ; set parent-prim as the current parent
 (define child-prim (build-cube)) ; make a child cube
 (grab parent-prim)
 (rotate (vector 0 45 0)) ; the child will now be moved by this transform in
addition to its own
 (ungrab)
```

(line-width value)

Returns void

Sets the line width (in screen space) of the current drawing state, or the current primitive. Affects wireframe and things like that.

Example

```
 (line-width 5)
 (hint-wire)
 (build-sphere 10 10) ; make a sphere with thick wireframe
```

(point-width value)

Returns void

Sets the point width (in screen space) of the current drawing state, or the current primitive. Affects point rendering and particles in hardware point mode.

Example

```
 (point-width 5)
 (hint-points)
 (build-sphere 10 10) ; make a sphere with thick points
```

(blend-mode src dst)

Returns void

Sets the blend mode of the current drawing state, or the current primitive. This is the way that alpha is composited to the rendering surface. Blendmode symbols can consist of: zero one dst-color one-minus-dst-color src-alpha one-minus-src-alpha dst-alpha one-minus-dst-alpha Also src-alpha-saturate as an option for the source blendmode only.

Example

```
 ; list out all the possible blendmodes

 (define src-blend (vector 'zero 'one 'dst-color 'one-minus-dst-color 'src-alpha
                    'one-minus-src-alpha 'dst-alpha 'one-minus-dst-alpha
                    'src-alpha-saturate))

 (define dst-blend (vector 'zero 'one 'dst-color 'one-minus-dst-color 'src-alpha
                    'one-minus-src-alpha 'dst-alpha 'one-minus-dst-alpha))
```

```
; picks a random element
(define (pick-rnd-item l)
    (vector-ref l (random (vector-length l))))

; make lots of random spheres
(define (rnd-sphere n)
    (push)
    (hint-depth-sort)
    (opacity 0.5)
    (colour (vector (flxrnd) (flxrnd) (flxrnd)))

    ; set a random blendmode
    (blend-mode (pick-rnd-item src-blend) (pick-rnd-item dst-blend))

    (translate (vector (flxrnd) (flxrnd) (flxrnd)))
    (scale (vector 0.1 0.1 0.1))
    (build-sphere 10 10)
    (pop)
    (if (zero? n)
        0
        (rnd-sphere (- n 1))))

(clear)
(clear-colour (vector 0.5 0.5 0.5))
(rnd-sphere 100)
```

(hint-solid)

Returns void

Sets the render hints to solid of the current drawing state, or the current primitive. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint.

Example

```
(hint-solid) ; this is the default render style so this isn't too exciting
(build-cube) ; make a solid rendered cube
```

(hint-wire)

Returns void

Sets the render hints to wireframe of the current drawing state, or the current primitive. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint.

Example

```
(hint-wire)
(build-cube) ; make a wirefame rendered cube
```

(hint-wire-stippled)

Returns void

Sets the render hints to stippled wireframe of the current drawing state, or the current primitive.

Example

```
(hint-none)
(hint-wire-stippled)
(build-cube) ; make a stippled wirefame cube
```

(line-pattern factor pattern)

Returns void

Factor specifies a multiplier for each bit in the line stipple pattern. Pattern specifies a 16-bit integer whose bit pattern determines which fragments of a line will be drawn when the line is rasterized.

Example

```
(hint-none)
(hint-wire-stippled)
(line-pattern 4 #x0aaaa)
(build-cube) ; make a stippled wirefame cube
```

(hint-normal)

Returns void

Sets the render hints to display normals in the current drawing state, or the current primitive. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint.

Example

```
(hint-normal)
(build-cube) ; display the normals on this cube
```

(hint-points)

Returns void

Sets the render hints to display points in the current drawing state, or the current primitive. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint.

Example

```
(hint-points)
(build-cube) ; display the vertex points on this cube
```

(hint-anti-alias)

Returns void

Sets the render hints to anti-alias in the current drawing state, or the current primitive. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint.

Example

```
(hint-anti-alias)
(build-cube) ; display a smoothed cube
```

(hint-unlit)

Returns void

Sets the render hints to unlit in the current drawing state, or the current primitive. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint.

Example

```
(hint-unlit)
(build-cube) ; display an unlit cube
```

(hint-vertcols)

Returns void

Sets the render hints to use vertex colours in the current drawing state, or the current primitive. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint. Vertex colours override the current (colour) state.

Example

```
(clear)
(hint-vertcols)
(define mycube (build-cube)) ; make a cube with vertcols enabled

(with-primitive mycube
    (pdata-map!
        (lambda (c)
            (rndvec)) ; randomise the vertcols
        "c"))
```

(hint-box)

Returns void

Sets the render hints to bounding box display in the current drawing state, or the current primitive. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint.

Example

```
(hint-box)
(build-sphere 10 10) ; make a sphere with bounding box displayed
```

(hint-none)

Returns void

Clears the render hints in the current drawing state, or the current primitive. This allows you mainly to get rid of the default solid style, but also means that you can turn on and off hints without using push or pop.

Example

```
(hint-none)
(hint-wire)
(build-cube) ; make a cube only visible with wireframe
```

(hint-origin)

Returns void

Sets the render hints to display the object space origin of the primitive in the current drawing state, or the current primitive. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint.

Example

```
(hint-origin)
(build-sphere 10 10) ; make a sphere with the origin displayed
```

(hint-cast-shadow)

Returns void

(note: Not yet implemented) Sets the render hints to cast shadows for the current drawing state, or the current primitive. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint.

Example

```
(hint-origin)
(build-sphere 10 10) ; make a sphere with the origin displayed
```

(hint-depth-sort)

Returns void

Sets the render hints to depth sort for the current drawing state, or the current primitive. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint.

Example

```
(hint-depth-sort)
(build-sphere 10 10)
```

(hint-ignore-depth)

Returns void

Sets the render hints to ignore depth tests for the current drawing state, or the current primitive. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint. This feature is useful for rendering transparent objects, as it means objects will be shown behind previously rendered ones.

Example

```
(clear)
(with-state
    (hint-ignore-depth)
    (opacity 0.6)
    (with-state
        (colour (vector 1 0 0))
        (build-cube))
    (with-state
        (colour (vector 0 1 0))
        (translate (vector 1 0 0))
        (build-cube)))
```

(hint-lazy-parent)

Returns void

Sets the render hints to prevent this primitive passing it's transform to it's children. Render hints change the way that primitives are rendered, but may have different effects - or no effect on certain primitive types, hence the name hint.

Example

```
(hint-lazy-parent)
(build-sphere 10 10) ; make a sphere with the origin displayed
```

(hint-cull-ccw)

Returns void

Flips the faces which get backface culled

Example

```
(hint-cull-ccw)
(build-sphere 10 10) ; make an inside out
```

(texture textureid-number)

Returns void

Sets the texture of the current drawing state, or the current primitive. Texture ids can be generated by the load-texture function.

Example

```
(texture (load-texture "mytexture.png"))
(build-sphere 10 10) ; make a sphere textured with mytexture.png
```

(multitexture textureunit-number textureid-number)

Returns void

Sets the texture of the current drawing state, or the current primitive in the same way as the texture function, but allows you to specify the texture unit (0-7) to apply the texture to. Multitexturing allows you to apply different textures and texture coordinates to the same object at once. Texture unit 0 is the default one (which uses the pdata "t" for it's texture coords) texture unit n looks for pdata "tn" - ie multitexture 1 looks for "t1". You need to add these yourself using (pdata-add) or (pdata-copy). Multitexturing is useful when the textures contain alpha, as they can be overlayed, i.e. decals placed on background textures. Note: fluxus needs to be built using scons MULTITEXTURE=1 to enable this feature.

Example

```
(clear)
(define p (build-torus 1 2 20 20))

(with-primitive p
    (multitexture 0 (load-texture "refmap.png"))
    (multitexture 1 (load-texture "transp.png")))
```

(print-scene-graph)

Returns void

Prints out the current scene graph, useful for debugging.

Example

```
(print-scene-graph) ; exciting...
```

(hide hidden-number)

Returns void

Sets the hidden state for the current primitive (also affects all child primitives). Hidden primitives can be treated as normal in every way - they just won't be rendered.

Example

```
(define obj (build-cube))
(grab obj)
(hide 1) ; hide this cube
(ungrab)
```

(camera-hide hidden-number)

Returns void

Sets the hidden state for the current primitive, with the current camera (also affects all child primitives). Allows you to turn off rendering for primitives under different cameras

Example

```
(define obj (build-cube))
(with-primitive obj
    (camera-hide 1)) ; hide this cube
```

(selectable selectable-number)

Returns void

Sets whether the current primitive can be selected or not using the select command.

Example

```
(define obj (build-cube))
(grab obj)
(selectable 0) ; now it won't be "seen" by calling select
(ungrab)
```

(backfacecull setting-number)

Returns void

Turns backface culling on or off. Backface culling speeds up rendering by removing faces not orientated towards the camera. Defaults to on, but this is not always desired, eg for double sided polygons.

Example

```
(backfacecull 0)
```

(shader vertexprogram-string fragmentprogram-string)

Returns void

Loads, compiles and sets the GLSL harware shader pair for the current drawing state, or the current primitive. Requires OpenGL 2 support. The shader's uniform data can be controlled via shader-set! and all the pdata is sent through as per-vertex attribute data to the shader.

Example

```
; you need to have built fluxus with GLSL=1
(clear)
(fluxus-init) ; this is important to add when using shaders
              ; at the moment, it will be moved somewhere
              ; to run automatically...

(define s (with-state
    ; assign the shaders to the surface
    (shader "simple.vert.glsl" "simple.frag.glsl")
    (build-sphere 20 20)))

(with-primitive s
    ; add and set the pdata - this is then picked up in the vertex shader
    ; as an input attribute called "testcol"
    (pdata-add "testcol" "v")
    ; set the testcol pdata with a random colour for every vertex
    (pdata-map!
        (lambda (c)
            (rndvec))
        "testcol"))

(define (animate)
    (with-primitive s
        ; animate the deformamount uniform input parameter
        (shader-set! (list "deformamount" (cos (time))))))

(every-frame (animate))
```

(shader-source vertexprogram-source-string fragmentprogram-source-string)

Returns void

Same as shader, but uses the supplied strings as shader sourcecode. This allows you to embed GLSL shader source inside your scheme scripts.

Example

```
; you need to have built fluxus with GLSL=1
```

(clear-shader-cache)

Returns void

Clears the shader cache

Example

```
(clear-shader-cache)
```

(shader-set! argument-list)

Returns void

Sets the uniform shader parameters for the GLSL shader. The list consists of token-string value pairs, which relate to the corresponding shader parameters names and values.

Example

```
; you need to have built fluxus with GLSL=1
(clear)
(fluxus-init) ; this is important to add when using shaders
              ; at the moment, it will be moved somewhere
              ; to run automatically...

(define s (with-state
    ; assign the shaders to the surface
    (shader "simple.vert.glsl" "simple.frag.glsl")
    (build-sphere 20 20)))

(with-primitive s
    ; add and set the pdata - this is then picked up in the vertex shader
    ; as an input attribute called "testcol"
    (pdata-add "testcol" "v")
    ; set the testcol pdata with a random colour for every vertex
    (pdata-map!
        (lambda (c)
            (rndvec))
        "testcol"))

(define (animate)
    (with-primitive s
        ; animate the deformamount uniform input parameter
        (shader-set! (list "deformamount" (cos (time))))))

(every-frame (animate))
```

(texture-params texture-unit-number parameter-list)

Returns void

Sets the current texture state for the specified texture unit. This state controls how the texture is applied to the surface of the object, and how it's filtered in texels, or miplevels. The texture unit is used if you are using multitexturing - the default texture unit is 0. You may need to read up a bit on OpenGL or experiment to find out more about these options.

Example

```
 ; parameters are the following:
 ; tex-env : [modulate decal blend replace]
 ; min : [nearest linear nearest-mipmap-nearest linear-mipmap-nearest linear-
mipmap-linear]
 ; mag : [nearest linear]
 ; wrap-s : [clamp repeat]
 ; wrap-t : [clamp repeat]
 ; wrap-r : [clamp repeat] (for cube maps)
 ; border-colour : (vector of length 4)
 ; priority : real number 0 -> 1
 ; env-colour : (vector of length 4)
 ; min-lod : real number (for mipmap blending - default -1000)
 ; max-lod : real number (for mipmap blending - default 1000)
 (texture-params 0 '(min nearest mag nearest))
```

primitive-data

Description

Primitive data (pdata for short) is fluxus' name for data which comprises primitives. In polygon primitives this means the vertex information, in particle primitives it corresponds to the particle information, in NURBS primitives it's the control vertices. Access to pdata gives you the ability to use primitives which are otherwise not very interesting, and deform and shape other primitives to give much more detailed models and animations. You can also add your own pdata, which is treated exactly like the built in types. Primitive data is named by type strings, the names of which depend on the sort of primitive. All pdata commands operate on the currently grabbed primitive.

Example

```
 ; a function to deform the points of an object
 (define (deform n)
     (pdata-set! "p" n (vadd  (pdata-ref "p" n)                ; the original
point, plus
         (vmul (vector (flxrnd) (flxrnd) (flxrnd)) 0.1)))     ; a small random
vector
     (if (zero? n)
         0
         (deform (- n 1))))

 (hint-unlit) ; set some render settings to
 (hint-wire)  ; make things easier to see
 (line-width 4)
 (define myobj (build-sphere 10 10)) ; make a sphere
 (grab myobj)
 (deform (pdata-size)) ; deform it
 (ungrab)
```

(pdata-ref type-string index-number)

Returns value-vector/colour/matrix/number

Returns the corresponding pdata element.

Example

```
(pdata-ref "p" 1)
```

(pdata-set! type-string index-number value-vector/colour/matrix/number)

Returns void

Writes to the corresponding pdata element.

Example

```
(pdata-set! "p" 1 (vector 0 100 0))
```

(pdata-add name-string type-string)

Returns void

Adds a new user pdata array. Type is one of "v":vector, "c":colour, "f":float or "m":matrix.

Example

```
(pdata-add "mydata" "v")
(pdata-set "mydata" 0 (vector 1 2 3))
```

(pdata-op funcname-string pdataname-string operator)

Returns void

This is an experimental feature allowing you to do operations on pdata very quickly, for instance adding element for element one array of pdata to another. You can implement this in Scheme as a loop over each element, but this is slow as the interpreter is doing all the work. It's much faster if you can use a pdata-op as the same operation will only be one Scheme call.

Example

```
(clear)
(define t (build-torus 1 4 10 10))

(with-primitive t
    (pdata-op "+" "p" (vector 1 0 0))  ; add a vector to all the pdata vectors
    (pdata-op "+" "p" "n")  ; add two pdata vectors element for element
    (pdata-op "*" "n" (vector -1 -1 -1)) ;  multiply a vector to all the pdata
vectors
    (pdata-op "*" "n" "p")  ; multiply two pdata vectors element for element
    (let ((pos (pdata-op "closest" "p" (vector 100 0 0)))) ;  returns position
of the closest vertex to this point
         (with-state ; draw a sphere there
             (translate pos)
             (scale (vector 0.1 0.1 0.1))
             (build-sphere 5 5)))
     ; can't think of a good example for these...
     ;(pdata-op "sin" "mydata" "myotherdata")  ; sine of one float pdata to 
another
     ;(pdata-op "cos" "mydata" "myotherdata")  ; cosine of one float pdata to 
another
     )

 ; most common example of pdata op is for particles
 (define p (with-state
     (hint-points)
     (point-width 10)
     (build-particles 100)))

 (with-primitive p
     (pdata-add "vel" "v") ; add a velocity vector
     (pdata-map!
         (lambda (vel)
             (srndvec)) ; set random velocities
         "vel")
     (pdata-map!
         (lambda (c)
             (rndvec)) ; set random colours
         "c"))

 (every-frame (with-primitive p
     (pdata-op "+" "p" "vel")))
```

(pdata-copy pdatafrom-string pdatato-string)

Returns void

Copies the contents of one pdata array to another. Arrays must match types.

Example

```
 (pdata-copy "p" "mydata") ; copy the vertex positions to a user array
```

(pdata-size)

Returns count-number

Returns the size of the pdata arrays (they must all be the same). This is mainly used for iterating over the arrays.

Example

```
 (define (mashup n)
     (pdata-set "p" n (vector (flxrnd) (flxrnd) (flxrnd))) ; randomise the 
vertex position
     (if (zero? n)
         0
```

```
        (mashup (- n 1)))) ; loops till n is 0

 (define shape (build-sphere 10 10))
 (grab shape)
 (mashup (pdata-size)) ; randomise verts on currently grabbed primitive
 (ungrab)
```

(recalc-normals smoothornot-number)

Returns void

For polygon primitives only. Looks at the vertex positions and calculates the lighting normals for you automatically. Call with "1" for smooth normals, "0" for faceted normals.

Example

```
 (define shape (build-sphere 10 10)) ; build a sphere (which is smooth by
default)
 (grab shape)
 (recalc-normals 0) ; make the sphere faceted
 (ungrab)
```

primitives

Description

Primitives are objects that you can render. There isn't really much else in a fluxus scene, except lights, a camera and lots of primitives.

Example

(build-cube)

Returns primitiveid-number

A simple cube, texture mapped placement per face.

Example

```
 (define mynewcube (build-cube))
```

(build-polygons verts-number type-symbol)

Returns primitiveid-number

Builds a raw polygon primitive with size vertices (everything is initially set to zero). Type is a symbol that refers to the way the vertices are interpreted to build polygons, and can be one of the following: triangle-strip quad-list triangle-list triangle-fan polygon

Example

```
(define mynewshape (build-polygons 100 'triangle-strip))
```

(build-sphere slices-number stacks-number)

Returns primitiveid-number

A sphere with the resolution specified, texture mapped in normal "world map" style.

Example

```
(define mynewshape (build-sphere 10 10))
```

(build-torus inner-radius-number outer-radius-number slices-number stacks-number)

Returns primitiveid-number

A torus with the resolution specified, control the size and thickness of the donut with the inner and outer radius.

Example

```
(define mynewshape (build-torus 0.5 1 12 12))
```

(build-plane)

Returns primitiveid-number

A single quad plane, texture mapped from 0->1 in both dimensions.

Example

```
(define mynewshape (build-plane))
```

(build-seg-plane vertsx-number vertsy-number)

Returns primitiveid-number

A tesselated poly plane, texture mapped from 0->1 in both dimensions.

Example

```
(define mynewshape (build-plane))
```

(build-cylinder hsegments rsegments)

Returns primitiveid-number

A capped cylinder, texture map wrapped around, and badly wrapped around the ends.

Example

```
(define mynewshape (build-cylinder 10 10))
```

(build-ribbon numpoints-number)

Returns primitiveid-number

Builds a ribbon consisting of numpoints points. The geometry is constantly camera facing and is texture mapped so the texture is stretched along the ribbon from start to finish. You use the pdata functions to edit the postions and widths of the segments. If used lit, the normals are faked to approximate a circular cross section. Additionally, if solid rendering is cleared with (hint-none) and (hint-wire) is activated, a faster constant width line will be drawn - width specified by the (line-width) command.

Example

```
(define mynewshape (build-ribbon 10))
```

(build-text text-string)

Returns primitiveid-number

Builds a sequence of planes, texture mapped so that a font texture can be used to display text. Might also be useful for more abstract things. The font assumed to be non proportional - there is an example font shipped with fluxus Ok, so this isn't a very good font texture :)

Example

```
(texture (load-texture "font.png"))
(define mynewshape (build-text "hello"))
```

(build-type ttf-filename text-string)

Returns primitiveid-number

Builds a geometric type primitive from a ttf font and some text.

Example

```
(clear)
(wire-colour (vector 0 0 1))
; this font should be on the default fluxus path (as it's the editor font)
(define t (build-type "Bitstream-Vera-Sans-Mono.ttf" "fluxus rocks!!"))

; make a poly primitive from the type
(define p (with-state
    (translate (vector 0 4 0))
    (type->poly t)))

; set some texture coords on the poly prim and load a texture onto it
(with-primitive p
    (pdata-map!
        (lambda (t p)
            (vmul p 0.5))
        "t" "p")
    (texture (load-texture "refmap.png")))
```

(build-extruded-type ttf-filename text-string extrude-depth)

Returns primitiveid-number

Builds an extruded geometric type primitive from a ttf font and some text.

Example

```
 (clear)
 (wire-colour (vector 0 0 1))
 ; this font should be on the default fluxus path (as it's the editor font)
 (define t (build-extruded-type "Bitstream-Vera-Sans-Mono.ttf" "fluxus rocks!!"
1))

 ; make a poly primitive from the type
 (define p (with-state
     (translate (vector 0 4 0))
     (type->poly t)))

 ; set some texture coords on the poly prim and load a texture onto it
 (with-primitive p
     (pdata-map!
         (lambda (t p)
             (vmul p 0.5))
         "t" "p")
     (texture (load-texture "refmap.png")))
```

(type->poly typeprimitiveid-number)

Returns polyprimid-number

Converts the mesh of a type primitive into a triangle list polygon primitive. The poly primitive will be a bit slower to render, but you'll be able to do everything you can do with a poly primitive with it, such as add textures and deform it.

Example

```
 (clear)
 (wire-colour (vector 0 0 1))
 ; this font should be on the default fluxus path (as it's the editor font)
 (define t (build-extruded-type "Bitstream-Vera-Sans-Mono.ttf" "fluxus rocks!!"
1))

 ; make a poly primitive from the type
 (define p (with-state
     (translate (vector 0 4 0))
     (type->poly t)))

 ; set some texture coords on the poly prim and load a texture onto it
 (with-primitive p
     (pdata-map!
         (lambda (t p)
             (vmul p 0.5))
         "t" "p")
     (texture (load-texture "refmap.png")))
```

(text-params width-number height-number stride-number wrap-number)

Returns primitiveid-number

Sets parameters for making fonts from texture maps. Defaults: 16/256 16/256 16 0

Example

```
; don't use me!
```

(build-nurbs-sphere hsegments rsegments)

Returns primitiveid-number

Builds a tesselated nurbs sphere, texture mapped in the same fashion as the poly sphere.

Example

```
(define mynewshape (build-nurbs-sphere 10 10))
```

(build-nurbs-plane hsegments rsegments)

Returns primitiveid-number

Builds a tesselated nurbs plane, texture mapped in uv direction.

Example

```
(define mynewshape (build-nurbs-plane 10 10))
```

(build-particles count-number)

Returns primitiveid-number

Builds a particles primitive containing num points, all initially set to the origin. You use the pdata functions to edit the postions, colours and sizes. Particles come in two flavors, camera facing sprites, which are the default, can be textured and individually scaled; and points (when hint-points is set), which cannot be textured but are much faster to render, as they are hardware supported gl points. By default these point particles are square, turn on hint-anti-alias to make them circular.

Example

```
(define mynewshape (build-particles 100))
```

(build-locator)

Returns primitiveid-number

A locator is an empty primitive, useful for parenting to (when you don't want to have the parent object visible). This primitive can only be visualised with (hint-origin) to display it's local transform origin.

Example

```
(define mynewshape (build-locator))
```

(locator-bounding-radius size-number)

Returns void

Sets the bounding box radius for the locator

Example

```
(define mylocator (build-locator))
(with-primitive mylocator
    (locator-bounding-radius 23.4))
```

(load-primitive)

Returns primitiveid-number

Loads a primitive from disk

Example

```
(define mynewshape (load-primitive "octopus.obj"))
```

(clear-geometry-cache)

Returns void

Clears cached geometry, so subsequent loads with come from the disk.

Example

```
(clear-geometry-cache)
```

(save-primitive)

Returns void

Saves the current primitive to disk

Example

```
(with-primitive (build-sphere 10 10)
    (save-primitive "mymesh.obj"))
```

(build-pixels width-number height-number)

Returns primitiveid-number

Makes a new pixel primitive. A pixel primitive is used for making procedural textures, which can then be applied to other primitives. For this reason, pixel primitives probably wont be rendered much, but you can render them to preview the texture on a flat plane.

Example

```
(define mynewshape (build-pixels 100 100))
(with-primitive mynewshape
    (pdata-map!
        (lambda (c)
            (rndvec))
        "c")
    (pixels-upload)) ; call pixels upload to see the results
```

(pixels-upload)

Returns void

Uploads the texture data, you need to call this when you've finished writing to the pixelprim, and while it's grabbed.

Example

```
(define mynewshape (build-pixels 100 100))
(with-primitive mynewshape
    (pdata-map!
        (lambda (c)
            (rndvec))
        "c")
    (pixels-upload)) ; call pixels upload to see the results
```

(pixels->texture pixelprimitiveid-number)

Returns textureid-number

Returns a texture you can use exactly like a normal loaded one.

Example

```
(define mypixels (build-pixels 100 100))
(with-primitive mypixels
    (pdata-map!
        (lambda (c)
            (rndvec))
        "c")
    (pixels-upload))

(with-state
    (texture (pixels->texture mypixels))
    (build-torus 1 2 10 10))
```

(pixels-width)

Returns width-number

Returns the width of the current pixel primitive.

Example

```
(define mynewshape (build-pixels 100 100))
(with-primitive mynewshape
    (display (vector (pixels-width) (pixels-height)))(newline))
```

(pixels-height)

Returns width-number

Returns the height of the current pixel primitive.

Example

```
(define mynewshape (build-pixels 100 100))
(with-primitive mynewshape
    (display (vector (pixels-width) (pixels-height)))(newline))
```

(build-blobby numinfluences subdivisionsvec boundingvec)

Returns primitiveid-number

Blobby primitives are a higher level implicit surface representation in fluxus which is defined using influences in 3 dimesional space. These infuences are then summed together, and a particular value is "meshed" (using the marching cubes algorithm) to form a smooth surface. The influences can be animated, and the smooth surface moves and deforms to adapt, giving the primitive it's blobby name. build-blobby returns a new blobby primitive. Numinfluences is the number of "blobs". Subdivisions allows you to control the resolution of the surface in each dimension, while boundingvec sets the bounding area of the primitive in local object space. The mesh will not be calculated outside of this area. Influence positions and colours need to be set using pdata-set.

Example

```
(clear)
(define b (build-blobby 5 (vector 30 30 30) (vector 1 1 1)))

(with-primitive b
    (shinyness 100)
    (specular (vector 1 1 1))
    (hint-vertcols)
    (pdata-set "p" 0 (vector 0.75 0.25 0.5))
    (pdata-set "c" 0 (vector 0.01 0 0))
    (pdata-set "s" 0 0.01)
    (pdata-set "p" 1 (vector 0.25 0.75 0.5))
    (pdata-set "c" 1 (vector 0 0.01 0))
    (pdata-set "s" 1 0.01)
    (pdata-set "p" 2 (vector 0.75 0.75 0.5))
    (pdata-set "c" 2 (vector 0 0 0.01))
    (pdata-set "s" 2 0.01)
    (pdata-set "p" 3 (vector 0.25 0.25 0.5))
    (pdata-set "c" 3 (vector 0.01 0.01 0))
    (pdata-set "s" 3 0.01)
    (pdata-set "p" 4 (vector 0.5 0.5 0.5))
    (pdata-set "c" 4 (vector 0.01 0.01 0.01))
    (pdata-set "s" 4 0.025))
```

(blobby->poly blobbyprimitiveid-number)

Returns polyprimid-number

Converts the mesh of a blobby primitive into a triangle list polygon primitive. This is useful as the polygon primitive will be much much faster to render, but can't deform in the blobby way. Doesn't convert vertex colours over yet unfortunately.

Example

```
(clear)
(define b (build-blobby 5 (vector 30 30 30) (vector 1 1 1)))

(with-primitive b
    (shinyness 100)
    (specular (vector 1 1 1))
    (hint-vertcols)
    (pdata-set "p" 0 (vector 0.75 0.25 0.5))
    (pdata-set "c" 0 (vector 0.01 0 0))
    (pdata-set "s" 0 0.01)
    (pdata-set "p" 1 (vector 0.25 0.75 0.5))
    (pdata-set "c" 1 (vector 0 0.01 0))
    (pdata-set "s" 1 0.01)
    (pdata-set "p" 2 (vector 0.75 0.75 0.5))
    (pdata-set "c" 2 (vector 0 0 0.01))
    (pdata-set "s" 2 0.01)
    (pdata-set "p" 3 (vector 0.25 0.25 0.5))
    (pdata-set "c" 3 (vector 0.01 0.01 0))
    (pdata-set "s" 3 0.01)
    (pdata-set "p" 4 (vector 0.5 0.5 0.5))
    (pdata-set "c" 4 (vector 0.01 0.01 0.01))
    (pdata-set "s" 4 0.025))

(define p (with-state
    (translate (vector 1 0 0))
    (blobby->poly b)))
```

(draw-instance primitiveid-number)

Returns void

Copies a retained mode primitive and draws it in the current state as an immediate mode primitive.

Example

```
(define mynewshape (build-cube))
```

```
(colour (vector 1 0 0))
(draw-instance mynewshape) ; draws a copy of mynewshape
```

(draw-cube)

Returns void

Draws a cube in the current state in immediate mode primitive.

Example

```
(define (render)
    (draw-cube))
(every-frame (render))
```

(draw-plane)

Returns void

Draws a plane in the current state in immediate mode primitive.

Example

```
(define (render)
    (draw-plane))
(every-frame (render))
```

(draw-sphere)

Returns void

Draws a sphere in the current state in immediate mode primitive.

Example

```
(define (render)
    (draw-sphere))
(every-frame (render))
```

(draw-cylinder)

Returns void

Draws a cylinder in the current state in immediate mode primitive.

Example

```
(define (render)
    (draw-cylinder))
(every-frame (render))
```

(draw-torus)

Returns void

Draws a torus in the current state in immediate mode primitive.

Example

```
(define (render)
    (draw-torus))
(every-frame (render))
```

(destroy primitiveid-number)

Returns void

Deletes a built primitive from the renderer. primitive.

Example

```
(define mynewshape (build-sphere 10 10))
(destroy mynewshape)
```

(poly-indices)

Returns void

Gets the vertex indices from this primitive primitive.

Example

```
(define p (build-cube))

(with-primitive p
    (poly-convert-to-indexed)
    (display (poly-indices))(newline))
```

(poly-type-enum)

Returns void

Returns the enum value representing the type of the current polygon primitive. This is needed as I can't get my scheme scripts to recognise symbols returned from here. Use (poly-type) instead of this directly. primitive.

Example

```
(define (poly-type)
  (let ((t (poly-type-enum)))
    (cond
      ((eq? t 0) 'triangle-strip)
      ((eq? t 1) 'quad-list)
      ((eq? t 2) 'triangle-list)
      ((eq? t 3) 'triangle-fan)
      ((eq? t 4) 'polygon))))
```

(poly-indexed?)

Returns void

Returns true if the current polygon primitive is in indexed mode. primitive.

Example

```
(define p (build-polygons 3 'triangle-strip))
(with-primitive p
    (poly-convert-to-indexed)
    (display (poly-indexed?))(newline))
```

(poly-set-index index-list)

Returns void

Switches the primitive to indexed mode, and uses the list as the index values for this primitive. primitive.

Example

```
(clear)
; lets build our own cube primitive...
(define p (build-polygons 8 'quad-list))

(with-primitive p
    ; setup the vertex data
    (pdata-set "p" 0 (vector -1 -1 -1))
    (pdata-set "p" 1 (vector  1 -1 -1))
    (pdata-set "p" 2 (vector  1 -1  1))
    (pdata-set "p" 3 (vector -1 -1  1))
    (pdata-set "p" 4 (vector -1  1 -1))
    (pdata-set "p" 5 (vector  1  1 -1))
    (pdata-set "p" 6 (vector  1  1  1))
    (pdata-set "p" 7 (vector -1  1  1))
    (pdata-set "c" 0 (vector  0  0  0))
    (pdata-set "c" 1 (vector  0  0  1))
    (pdata-set "c" 2 (vector  0  1  0))
    (pdata-set "c" 3 (vector  0  1  1))
    (pdata-set "c" 4 (vector  1  0  0))
    (pdata-set "c" 5 (vector  1  0  1))
    (pdata-set "c" 6 (vector  1  1  0))
    (pdata-set "c" 7 (vector  1  1  1))

    (hint-wire)
    (hint-unlit)
    (hint-vertcols)

    ; connect the verts together into faces
    (poly-set-index (list 7 6 5 4  5 6 2 1
            4 5 1 0  1 2 3 0
            3 7 4 0  6 7 3 2)))
```

(poly-convert-to-indexed)

Returns void

Converts the currently grabbed polygon primitive from raw vertex arrays to indexed arrays. This removes duplicate vertices from the polygon, making the pdata arrays shorter, which speeds up processing time.

Example

```
(define mynewshape (build-sphere 10 10))
(grab mynewshape)
(poly-convert-to-indexed)
(ungrab)
```

(build-copy src-primitive-number)

Returns primitiveid-number

Returns a copy of a primitive

Example

```
(define mynewshape (build-sphere 10 10))
(define myothernewshape (build-copy mynewshape))
```

(make-pfunc name-string)

Returns pfuncid-number

Makes a new primitive function. pfuncs range from general purpose to complex and specialised operations which you can run on primitives. All pfuncs share the same interface for controlling and setting them up - pfunc-set! All pfunc types and arguments are as follows: arithmetic For applying general arithmetic to any pdata array operator string : one of add sub mul div src string : pdata array name other string : pdata array name (optional) constant float : constant value (optional) dst string : pdata array name genskinweights Generates skinweights - adds float pdata called "s1" -> "sn" where n is the number of nodes in the skeleton - 1 skeleton-root primid-number : the root of the bindpose skeleton for skinning sharpness float : a control of how sharp the creasing will be when skinned skinweights->vertcols A utility for visualising skinweights for debugging. no arguments skinning Skins a primitive - deforms it to follow a skeleton's movements. Primitives we want to run this on have to contain extra pdata - copies of the starting vert positions called "pref" and the same for normals, if normals are being skinned, called "nref". skeleton-root primid-number : the root primitive of the animating skeleton bindpose-root primid-number : the root primitive of the bindpose skeleton skin-normals number : whether to skin the normals as well as the positions

Example

```
(define mypfunc (make-pfunc 'arithmetic))
```

(pfunc-set! pfuncid-number argument-list)

Returns void

Sets arguments on a primitive function. See docs for make-pfunc for all the arguments.

Example

```
(define mypfunc (make-pfunc 'arithmetic))
(pfunc-set! mypfunc (list 'operator "add"
                          'src "p"
                          'const 0.4
                          'dst "p"))
```

(pfunc-run id-number)

Returns void

Runs a primitive function on the currently grabbed primitive.

Example

```
(define mypfunc (make-pfunc 'arithmetic))
```

(line-intersect start-vec end-vec)

Returns void

Returns a list of pdata values at each intersection point of the specified line.

Example

```
(clear)
(define s (with-state
        (build-torus 1 2 10 10)))

(define l (with-state
        (hint-none)
        (hint-unlit)
        (hint-wire)
        (build-line 2)))

(define (check a b)
    (with-primitive s
        (for-each
            (lambda (intersection)
                (with-state ; draw a sphere at the intersection point
                    (translate (cdr (assoc "p" intersection)))
                    (colour (vector 0 1 0))
                    (scale (vector 0.3 0.3 0.3))
                    (draw-sphere)))
        (line-intersect a b))))

(every-frame
    (with-primitive l
        (pdata-set "p" 0 (vector 0 -5 0))
        (pdata-set "p" 1 (vector (* 5 (sin (time))) 5 0))
        (check (pdata-ref "p" 0) (pdata-ref "p" 1))))
```

(bb-intersect prim thresh)

Returns void

Returns #t if the current primitive bounding box intersects with the supplied one, with an additional expanding threshold.

Example

```
 (clear)

 (define a (with-state
      (build-sphere 10 10)))

 (define b (with-state
      (translate (vector 2 0 0))
      (build-sphere 10 10)))

 (every-frame
     (begin
         (with-primitive b
             (translate (vector (* -0.1 (sin (time))) 0 0)))
         (with-primitive a
             (when (bb-intersect b 0)
                 (colour (rndvec))))))
```

(get-children)

Returns void

Gets a list of primitives parented to this one.

Example

```
 ; build a random heirachical structure
 (define (build-heir depth)
     (with-state
         (let ((p (with-state
                         (translate (vector 2 0 0))
                         (scale 0.9)
                         (build-cube))))
             (when (> depth 0)
                 (parent p)
                 (for ((i (in-range 0 5)))
                     (when (zero? (random 3))
                         (rotate (vector 0 0 (* 45 (crndf))))
                         (build-heir (- depth 1))))))))

 ; navigate the scene graph and print it out
 (define (print-heir children)
     (for-each
         (lambda (child)
             (with-primitive child
                 (printf "id: ~a parent: ~a children: ~a~n" child (get-parent)
(get-children))
                 (print-heir (get-children))))
         children))
 (clear)
 (build-heir 5)
 (print-heir (get-children))
```

(get-parent)

Returns void

Gets the parent of this node. 1 is the root node.

Example

```
 ; build a random heirachical structure
 (define (build-heir depth)
     (with-state
         (let ((p (with-state
                         (translate (vector 2 0 0))
                         (scale 0.9)
                         (build-cube))))
             (when (> depth 0)
                 (parent p)
                 (for ((i (in-range 0 5)))
                     (when (zero? (random 3))
                         (rotate (vector 0 0 (* 45 (crndf))))
                         (build-heir (- depth 1))))))))

 ; navigate the scene graph and print it out
 (define (print-heir children)
     (for-each
         (lambda (child)
             (with-primitive child
                 (printf "id: ~a parent: ~a children: ~a~n" child (get-parent)
(get-children))
                 (print-heir (get-children))))
         children))

 (clear)
 (build-heir 5)
 (print-heir (get-children))
```

util-functions

Description

Handy functions to make your life easier...

(time)

Returns time-number

Returns the number of seconds (+ fraction) since midnight January 1st 1970. This is the simpest animation source for your scripts.

Example

```
(define (animate)
    (rotate (vector (sin (time)) 0 0))
    (draw-cube))
(every-frame (animate))
```

(delta)

Returns time-number

Time in seconds since the last frame. Used to make animation frame rate independant.

Example

```
(define (animate)
    (rotate (vector (* (delta) 10) 0 0))
    (draw-cube))
(every-frame (animate))
```

(flxrnd)

Returns random-number

Returns a random number between 0 and 1.

Example

```
(define (animate)
    (colour (vector (flxrnd) (flxrnd) (flxrnd)))
    (draw-cube))
(every-frame (animate))
```

(flxseed seed-number)

Returns void

Seeds the random number generator so we can get the same sequence.

Example

```
(define (animate)
    (colour (vector (flxrnd) (flxrnd) (flxrnd)))
    (draw-cube))
(flxseed 10)
(every-frame (animate)) ; the same sequence of colours will be generated
```

(set-searchpathss paths-list)

Returns void

Sets a list of search path strings to use for looking for fluxus related files, such as textures, shaders etc. Paths will be searched in order each time, and need trailing slashes.

Example

```
 (set-searchpaths (append (get-searchpaths) (list "/path/to/my/textures/" 
"/path/to/my/other/textures/")))
```

(get-searchpaths paths-list)

Returns void

Gets the list of search path strings to use for looking for fluxus related files, such as textures, shaders etc. Paths will be searched in order each time.

Example

```
 (display (get-searchpaths))(newline)
```

(fullpath filename-string)

Returns fullpath-string

Searches the search paths for the specified file and returns the first location it finds.

Example

```
 (fullpath "myfile")
```

(framedump filename)

Returns void

Saves out the current OpenGL front buffer to disk. Reads the filename extension to decide on the format used for saving, "tif", "jpg" or "ppm" are supported. This is the low level form of the frame dumping, use start-framedump and end-framedump instead.

Example

```
 (framedump "picture.jpg")
```

(framedump filename)

Returns void

For rendering images that are bigger than the screen, for printing or other similar stuff. This command uses a tiled rendering method to render bits of the image and splice them together into the image to save. Reads the filename extension to decide on the format used for saving, "tif", "jpg" or "ppm" are supported.

Example

```
 (tiled-framedump "picture.jpg" 3000 2000)
```

osc

Description

OSC stands for Open Sound Control, and is a widely used protocol for passing data between multimedia applications. Fluxus can send or receive messages.

Example

```
An example of using osc to communicate between pd and fluxus.
A fluxus script to move a cube based on incoming osc messages.
-- osc.scm

(define value 0)

(define (test)
    (push)
    (if (osc-msg "/zzz")
        (set! value (osc 0)))
    (translate (vector 1 0 value))
    (draw-cube)
    (pop))

(osc-source "6543")
(every-frame (test))

--- EOF
A PD patch to send control messages to fluxus:
--- zzz.pd
#N canvas 618 417 286 266 10;
#X obj 58 161 sendOSC;
#X msg 73 135 connect localhost 6543;
#X msg 58 82 send /zzz \$1;
#X floatatom 58 29 5 0 0 0 - - -;
#X obj 58 54 / 100;
#X obj 73 110 loadbang;
#X connect 1 0 0 0;
#X connect 2 0 0 0;
#X connect 3 0 4 0;
#X connect 4 0 2 0;
#X connect 5 0 1 0;
```

(osc-source port-string)

Returns void

Starts up the osc server, or changes port. Known bug: seems to fail if you set it back to a port used previously.

Example

```
(osc-source "4444")     ; listen to port 4444 for osc messages
```

(osc-msg name-string)

Returns msgreceived-boolean

Returns true if the message has been received since the last frame, and sets it as the current message for subsequent calls to (osc) for reading the arguments.

Example

```
(cond
    ((osc-msg "/hello")              ; if a the /hello message is recieved
        (display (osc 1))(newline)))   ; print out the first argument
```

(osc argument-number)

Returns oscargument

Returns the argument from the current osc message.

Example

```
(cond
    ((osc-msg "/hello")              ; if a the /hello message is recieved
        (display (osc 1))(newline)))   ; print out the first argument
```

(osc-destination port-string)

Returns void

Specifies the destination for outgoing osc messages. The port name needs to specify the whole url and should look something like this "osc.udp://localhost:4444"

Example

```
(osc-destination "osc.udp:localhost:4444")
(osc-send "/hello" "s" (list "boo!"))  ; send a message to this destination
```

(osc-peek)

Returns msg-string

This util function returns the name, and format string and number/string arguments of the last sent message as a string - for debugging your osc network.

Example

```
(display (osc-peek))(newline)
```

(osc-send name-string format-string argument-list)

Returns void

Sends an osc message with the argument list as the osc data. Only supports floats, ints and strings as data. The format-string should be composed of "i", "f" and "s", and must match the types given in the list. This could probably be removed by using the types directly, but doing it this way allows you to explicitly set the typing for the osc message.

Example

```
 (osc-destination "osc.udp:localhost:4444")
 (osc-send "/hello" "sif" (list "boo!" 3 42.3))  ; send a message to this
destination
```

scratchpad

Description

Functions available as part of the fluxus scratchpad.

(reset-camera)

Returns void

Resets the camera transform, useful if it becomes trashed, or you get lost somewhere in space. Also turns off camera locking to objects with (lock-camera)

Example

```
 ; ruin the camera transform
 (set-camera-transform (vector 123 41832 28 0.2 128 0.001 123 41832 28 0.2 128
0.001 0.2 100 13 1931))
 ; set it back to the starting position/orientation
 (reset-camera)
```

(set-camera-transform transform-matrix)

Returns void

Overrides and locks the camera transform with your own. To unlock again call reset-camera

Example

```
 (set-camera-transform (mtranslate (vector 0 0 -10)))
```

(get-camera-transform)

Returns transform-matrix

Returns the current camera transform. To unlock again call reset-camera

Example

```
 (define tx (get-camera-transform))
```

(set-help-locale! locale-string)

Returns void

Sets the language for the documentation

Example

```
(set-help-locale! "pt") ; switch to portuguese
(set-help-locale! "en") ; and back to english
```

(help function-string)

Returns void

Displays help information on a fluxus function. For running in the repl mainly.

Example

```
(help "pop")
```

(key-pressed key-string)

Returns boolean

Returns true if the specified key is currently pressed down.

Example

```
(when (key-pressed "q") (display "q pressed!"))
```

(keys-down)

Returns keys-list

Returns a list of keys pressed down

Example

```
(display (keys-down))(newline)
```

(key-special-pressed key-number)

Returns boolean

Returns true if the specified special key is currently pressed down. Special keys are ones which do not map to ascii values. The easiest way of finding what they are is to print out the result of key-special-pressed while holding down the key you are after.

Example

```
(when (key-special-pressed 100) (display "left cursor pressed"))
(when (key-special-pressed 102) (display "right cursor pressed"))
(when (key-special-pressed 101) (display "up cursor pressed"))
(when (key-special-pressed 103) (display "down cursor pressed"))
```

(keys-special-down)

Returns keys-list

Returns a list of special keys pressed down

Example

```
 (display (keys-special-down))
```

(mouse-x)

Returns coord-number

Returns the x position of the mouse

Example

```
 (display (mouse-x))
```

(mouse-y)

Returns coord-number

Returns the y position of the mouse

Example

```
 (display (mouse-y))
```

(mouse-button)

Returns boolean

Returns true if the specifed mouse button is pressed

Example

```
 (display (mouse-button 1))
```

(mouse-wheel)

Returns boolean

Returns 1 if the mouse wheel was moved in one direction in the last frame or -1 if it was turned the other way, otherwise returns 0.

Example

```
 (display (mouse-wheel))
```

(mouse-over)

Returns primitiveid-number

Returns the object the mouse is currently over.

Example

```
(grab (mouse-over))
(colour (vector 1 0 0)) ; paints objects the mouse is over red
(ungrab)
```

(every-frame callback-function)

Returns void

Sets a function to be called every time the render is about to draw a new frame.

Example

```
(define (myfunc)
    (colour (rndvec))
    (draw-torus))

(every-frame (myfunc))
```

(clear)

Returns void

Clears out the renderer of all objects and lights. Clears the physics system and resets the every-frame callback. Generally a Good Thing to put this at the beginning of scripts to make sure everything is cleared out each time you execute.

Example

```
(clear) ; without this we would accumulate a new cube every time F5 was pressed
(build-cube)
```

(start-framedump name-string type-string)

Returns void

Starts saving frames to disk. Type can be one of "tif", "jpg" or "ppm". Filenames are built with the frame number added, padded to 5 zeros.

Example

```
(start-framedump "frame" "jpg")
```

(end-framedump)

Returns void

Stops saving frames to disk.

Example

```
(end-framedump)
```

(set-physics-debug boolean)

Returns void

Call with #t to turn on debug rendering for the physics.

Example

```
(set-physics-debug #t)
```

(override-frame-callback callback-function)

Returns void

Allows you to override the frame callback, to control the rendering loop of fluxus in a more detailed way.

Example

```
(override-frame-callback myfunc)
(override-frame-callback default-fluxus-frame-callback) ; set it back again...
```

(set-auto-indent-tab size-number)

Returns void

Sets the tabs size for the prettification auto indent on ctrl-p. Defaults to 2.

Example

```
(set-auto-indent-tab 2)
```

(set-camera-update #t/#f)

Returns void

Turns off camera update - allowing you to use (set-camera) - otherwise it gets written over by the mouse camera update. The reason for needing this is that (set-camera-transform) doesn't work with multiple cameras - need to fix.

Example

```
(set-camera-update #f)
(set-camera-update #t)
```

(spawn-task)

Returns void

Launches a new per-frame task, a tasks: * execute once per graphic frame * are called in lexigraphical order by name * have unique names and if the same name is used the old task is removed prior to the new task being added * a task that returns #f will remove itself after executing (every-frame (build-cube)) is equivalent to (spawn-task (lambda () (build-cube)) 'every-frame-task)

Example

```
(spawn-task (lambda () (draw-torus)) 'torus-task)
(rm-task 'torus-task)
```

(rm-task)

Returns void

Removes a task from the tasklist

Example

```
(spawn-task (lambda () (draw-torus)) 'torus-task) ; add a task
(rm-task 'torus-task) ; remove it again
```

(rm-all-tasks)

Returns void

Removes all task from the tasklist, including the every-frame task.

Example

```
(rm-all-tasks)
```

(ls-tasks)

Returns void

Prints a list of current a tasks

Example

```
(spawn-task (lambda () (draw-torus)) 'torus-task) ; add a task
(ls-tasks)
(rm-task 'torus-task)
```

(spawn-timed-task time thunk)

Returns void

Launches a new timed task, which will happen in the future, on the frame that the time specifies. Use (time-now) rather than (time) to obtain the time. I need to sort that out.

Example

```
(spawn-timed-task (+ (time-now) 10) ; schedule a task 10 seconds from now
    (lambda () (display "hello future!") (newline)))
```

high-level-scratchpad-docs

Description

Some useful high level documentation lives here, this won't make much sense if you are reading this in a different place from the help system in the fluxus scratchpad app, but it might be useful anyway...

(tophelp)

Returns

Example

```
Fluxus documentation
--------------------
"act of a flowing; a continuous moving on or passing by, as of a
flowing stream; a continuous succession of changes"

Fluxus is a realtime rendering engine for livecoding in Scheme.
For more detailed docs, see: fluxus/docs/fluxus-documentation.txt

The fluxus scratchpad has two modes of operation, the console
(you are using this now) which allows you to enter commands and
see the results immediately. The other mode is the editor which
is more like a normal text editor - there are 9 workspaces,
(which allow you to edit more than one script at once) switch to
them using ctrl-1 to ctrl-9 and switch back to the console with
ctrl-0.

To copy/paste examples, hit the right cursor until you move behind
the prompt, navigate to the example, use shift to select it,
press ctrl-c to copy, then ctrl-0 and ctrl-v to paste into a
text buffer.

More help topics:
(help "keys") for keyboard commands for controlling fluxus
(help "console") for more help on the console
(help "editor") for more help on the livecoding editor
(help "camera") for help on the camera controls
(help "language") for more info on the fluxus commands
(help "misc") for miscellaneous fluxus info
(help "toplap") for the toplap manefesto
(help "authors") who made this?
```

(keys)

Returns

Example

```
Fluxus keys
-----------

ctrl-f : Fullscreen mode.
```

```
ctrl-w : Windowed mode.
ctrl-h : Hide/show the text.
ctrl-l : Load a new script (navigate with cursors and return).
ctrl-s : Save current script.
ctrl-d : Save as - current script (opens a filename dialog).
ctrl-p : Auto format the whitespace in your scheme script to be more pretty
ctrl-1 to 9 : Switch to selected workspace.
ctrl-0 : Switch to the REPL.
F3 : Resets the camera if you get lost.
F5 : (or ctrl-e) Executethe selected text, or all if none is selected.
F6 : Completely resets the interpreter, then executes the selected text,
     or all if none is selected.
F9 : Randomise the text colour (aka the panic button)
F10 : Decreases the text opacity
F11 : Increases the text opacity
```

(console)

Returns

Example

```
Fluxus console (or REPL)
------------------------

If you press ctrl and 0, instead of getting another script
workspace, you will be presented with a Read EvaluatePrint
Loop interpreter, or repl for short. This is really just an
interactive interpreter similar to the commandline, where
you can enter scheme code for immediate evaluation. This code
is evaluated in the same interpreter as the other scripts, so
you can use the repl to debug or inspect global variables and
functions they define. This window is also where error
reporting is printed, along with the terminal window you
started fluxus from.
```

(editor)

Returns

Example

```
Fluxus editor
-------------

When using the fluxus scratchpad, the idea is that you only
need the one window to build scripts, or play live. f5 is the
key that runs the script when you are ready.  Selecting some
text (using shift) and pressing f5 will execute the selected
text only. This is handy for reevaluating functions without
running the whole script each time.

Workspaces
----------

The script editor allows you to edit 9 scripts simultaneously
```

```
by using workspaces. To switch workspaces, use ctrl+number
key. Only one can be run at once though, hitting f5 will
execute the currently active workspace script.

Auto focus
----------

The editor includes an auto scaling/centering feature which is
enabled by default. To disable it - add the line:
(set! fluxus-scratchpad-do-autofocus 0)
to your .fluxus.scm file - or create a new file called that in
your home directory, containing that line.
```

(camera)

Returns

Example

```
Fluxus camera control
---------------------

The camera is controlled by moving the mouse and pressing
mouse buttons.

Left mouse button: Rotate
Middle mouse button: Move
Right mouse button: Zoom
```

(misc)

Returns

Example

```
Fluxus init script
------------------
Fluxus looks for a script in your home directory called
.fluxus.scm which it will run if it is found. This is useful
for putting init commands (like connecting to jack or setting
the help text language etc)

Frame rate throttling
---------------------
By default fluxus throttles the framerate to around 40fps.
to disable this (and run at 100% cpu), use desiredfps with
some arbitrary large number:
(desiredfps 100000000)
To display the fps use (show-fps 1)

Command line options
---------------------
The easiest way to load a script into fluxus is to specify it on
the command line, eg:
$ fluxus myscript.scm
Will launch fluxus and load the script into the editor.
```

```
$ fluxus -x myscript.scm
Will launch fluxus, load, hide and execute the script.
Use -h to print all commandline options.

Fluxus also contains a keypress and mouse event recorder for
recording livecoding sessions:
$ fluxus -r filename : record to keypresses file
$ fluxus -p filename : playback from file
$ fluxus -p filename -d time : seconds per frame time override for
                               playback (for use with frame-dump)
```

(authors)

Returns

Example

```
Authors
-------
Glauber Alex Dias Prado
Artem Baguinski
Dan Bethell
Nik Gaffney
Dave Griffiths
Claude Heiland-Allen
Alex Norman
Gabor Papp
Fabien Pelisson
Jeff Rose
James Tittle

"Computers are useless. They can only give you answers".
    Pablo Picasso (1881 - 1973).
```

(language)

Returns

Example

```
Language Docs
-------------

Fluxus is comprised of a set of functions which
extend Scheme for use in realtime computer graphics.

Scheme itself is out of the scope of this documentation,
but fluxus is a good way of learning it. I reccommend
"The Little Schemer" by by Daniel P. Friedman and Matthias
Felleisen.

The functions are grouped into sections to make things
a bit easier to find.
```

```
(help "sections") for a list of all sections
(help "sectionname") to find out more about a section
(help "functionname") to find out more about a function

The idea is that you can find a function you are interested
in by doing something like this:

(help "sections")
   ... list of sections ...
(help "maths")
   ... description and list of maths functions ...
(help "vmul")
   ... details about the function with example ...
```

(toplap)

Returns

Example

```
TOPLAP MANEFESTO
We demand:
 * Give us access to the performer's mind, to the whole human
   instrument.
 * Obscurantism is dangerous. Show us your screens.
 * Programs are instruments that can change themselves
 * The program is to be transcended - Artificial language is the way.
 * Code should be seen as well as heard, underlying algorithms viewed
   as well as their visual outcome.
 * Live coding is not about tools. Algorithms are thoughts.
   Chainsaws are tools. That's why algorithms are sometimes
   harder to notice than chainsaws.

We recognise continuums of interaction and profundity, but prefer:
 * Insight into algorithms
 * The skillful extemporisation of algorithm as an
   expressive/impressive display of mental dexterity
 * No backup (minidisc, DVD, safety net computer)

We acknowledge that:
 * It is not necessary for a lay audience to understand the
   code to appreciate it, much as it is not necessary to know
   how to play guitar in order to appreciate watching a guitar
   performance.
 * Live coding may be accompanied by an impressive display of
   manual dexterity and the glorification of the typing interface.
 * Performance involves continuums of interaction, covering
   perhaps the scope of controls with respect to the parameter
   space of the artwork, or gestural content, particularly
   directness of expressive detail. Whilst the traditional
   haptic rate timing deviations of expressivity in
   instrumental music are not approximated in code, why repeat
   the past? No doubt the writing of code and expression of
   thought will develop its own nuances and customs.
Performances and events closely meeting these manifesto
conditions may apply for TOPLAP approval and seal.
```

frisbee

Description

Frisbee is an experimental high level game engine written for FrTime, a functional reactive programming language available as part of PLT Scheme. It's completely separate to the main fluxus commands, and represents a different way of creating games or other behavoural systems.

(vec3 x y z)

Returns result-vector

Creates a new vector usable inside frisbee - use this rather than (vector)

Example

```
(vec3 1 2 3)
```

(vec3-x v)

Returns result-number

Returns the x component of the frisbee vector

Example

```
(vec3-x (vec3 1 2 3))
```

(vec3-y v)

Returns result-number

Returns the y component of the frisbee vector

Example

```
(vec3-y (vec3 1 2 3))
```

(vec3-z v)

Returns result-number

Returns the z component of the frisbee vector

Example

```
(vec3-z (vec3 1 2 3))
```

(vec3-integral v)

Returns result-vector

Returns the integral of the frisbee vector in respect to time

Example

```
(vec3-integral (vec3 0 0.01 0))
```

(scene collide-b)

Returns void

Sets the frisbee scene up. The list can contain primitive structures, or more lists.

Example

```
(scene (list (cube)))
```

(scene scene-list)

Returns void

Sets the frisbee scene up. The list can contain primitive structures, or more lists.

Example

```
(scene (list (cube)))
```

testing-functions

Description

A set of higher level control structures for manipulating objects and state in fluxus in a cleaner and safer manner.

(self-test do-logging)

Returns void

Runs all of the function reference scripts in a separate thread so you can watch it in action. Just checks for syntactic errors in the scheme, changes in bound C++ function signatures and crashes. Graphical code is difficult to test for correctness further (that's my excuse). If do-logging is true it outputs a log text file to the current directory for debugging.

Example

```
(self-test #t)
```

(run-scripts path-to-examples)

Returns void

Runs all of the example scripts in a separate thread so you can watch it in action.

Example

```
 (run-scripts path-to-scripts seconds-per-script)
```