# FilledVector

| COLLABORATORS | | | |
|---|---|---|---|
| | *TITLE* :<br><br>FilledVector | | |
| *ACTION* | *NAME* | *DATE* | *SIGNATURE* |
| WRITTEN BY | | December 8, 2024 | |

| REVISION HISTORY | | | |
|---|---|---|---|
| NUMBER | DATE | DESCRIPTION | NAME |
| | | | |

# Contents

**1 FilledVector** **1**

1.1 FilledVector.guide . . . 1
1.2 overview . . . 1
1.3 author . . . 2
1.4 polygon context . . . 3
1.5 polygon context overview . . . 3
1.6 newpolycontext . . . 4
1.7 freepolycontext . . . 4
1.8 setpolybitmap . . . 5
1.9 setpolycliprect . . . 5
1.10 setpolyflags . . . 6
1.11 vectorobject . . . 6
1.12 newvectorobject . . . 7
1.13 freevectorobject . . . 8
1.14 copyvectorobject . . . 8
1.15 clonevectorobject . . . 9
1.16 getvobjectpoints . . . 9
1.17 newvlist . . . 10
1.18 freevlist . . . 10
1.19 addvlist . . . 11
1.20 remvlist . . . 11
1.21 sortvlist . . . 12
1.22 drawvobject . . . 12
1.23 drawvlist . . . 13
1.24 movedrawvlist . . . 13
1.25 rendering . . . 14
1.26 matrix . . . 14
1.27 matrixinfo . . . 15
1.28 setmatident . . . 15
1.29 setmatrotate . . . 16

1.30 setmatscale . . . . . 16
1.31 matsize . . . . . 17
1.32 matmult . . . . . 18
1.33 matapply3 . . . . . 18
1.34 Example code . . . . . 19

# Chapter 1

# FilledVector

## 1.1 FilledVector.guide

```
Filled Vector Module for AmigaE 2.5+
```



```
This document describes the usage of a suite of 3d filled vector routines
compiled for the AmigaE language.

The following sections are available:

   OverView     some of the ideas behind the module

   Polygon Context  describing destination memory

   Vector Object    vector object creation/manipulation

   Rendering        rendering functions

   Matrix           coordinate manupulation functions


    a few playthings i came up with ...

NOTE: You need to have:

MODULE 'tools/filledvector', 'tools/filledvdefs'

Somewhere in your MODULE include section


This module should work with all Amiga's
```

## 1.2 overview

```
Filled Vector Module Overview

Whats this module all about?

This module contains some reasonably 'optimised' rendering code
to render 3d polygon filled vectors.  This includes some a
reasonably complex and versatile object format that can be
used to create complex iconvex vector objects.

Because it uses the blitter to do rendering, it means you
can have very complex objects (ones with holes in them
and so on) without slowing it down too much.  Unfortunately,
performance on accelerated machines isn't so hot ...

BTW, i say "optimised", but its not that fast really ... one
reason is because i'm keeping the system intact :)
Another reason, is this code is somewhat old by now, and i
dont really have the time to update it to the latest
kill-os version i have (the kill-os version uses a very
interrupt-intensive customblitter queue, and it operates
on parameters fixed at compile-time.  It is also a LOT
faster though ...).  It also uses some more efficient
data structures and algorithms that i havent had time
to put into this one.

One of the "cool" features it DOES have though, is the ability
to perform z clipping as its rendering objects.  This allows
you to smoothly run through objects as you run into them,
rather than having them dissapear just as they start to get big.
It doesn't use a particularaly reliable or fast z clipping
algorithm but its still not too bad. (again, my kill-os
vector code uses a much more efficient version ...).


Hopefully someone out there will be able to do more with it
than just another version of ZedWB :)  (although, for
full-on vector world creation, you need some more
tools ...)

 MZ
```

## 1.3 author

```
I'm another one of these poor students, but i live in Australia, not Europe!
I've been stuffing about with this vector shit since i had a '64, but never
really got to do anything with it ...

Presently, i study 'from time to time' :-) in order to obtain a Computer
Systems Engineering degree from the Univerity Of South Australia.
I'm the 'Zed' of FRONTIER in my anti-os hours.

I can be contacted in the following ways:

Internet email:
```

```
9107047w@lux.levels.unisa.edu.au
  till the end of '94 at least - reliable

zucchi@hal9000.apana.org.au
zucchi@bkroom.apana.org.au
  until i keep acounts there (?)

'Real Mode' (tm) mail:

Michael Zucchi
PO BOX 824
Waikerie
South Australia 5330
  slow, but very reliable - till mum sells the house :)

Michael Zucchi
110 Dunrobin Rd
Warradale
South Australia 5046
  till end of '94
```

## 1.4 polygon context

```
This section describes the functions used for creating and manipulating
'polygon contexts'.

   Polygon Context OverView

  newPolyContext()  create a new one
  freePolyContext() free one
  setPolyBitMap()   change where it renders
  setPolyClipRect() change clipping window

All of these functions operate with system libraries starting from V33
(workbench 1.2).  However, please note that interleaved bitmaps are
only possible when using V39 libraries (WB3.0+).
```

## 1.5 polygon context overview

```
A polygon context is similar to a rastport, but instead of containing
all and sundry general purpose drawing variables as a rastport must,
it only contains data necessary for polygon rendering.

The following is an example of some of the information that is stored in
the polygon context:

  clipping rectangle
  minimum dimensions of current polygon (for blitting)
  size of destination bitmap
  bytesperrow of bitmap and screen
```

```
  pointer to a bitplane to render polygons into
  pointer to bitmap to blit thus renderd polygons
  some temporary storage

Because of the context sensitive nature of this data object, when one
is allocated it must be attached to a specific screen or bitmap - and MUST NOT
be used for another screen or bitmap, unless it is absolutely clear that
they are of the same dimensions (the buffered screen routines guaruntee this).

All fields of the polygon context are !PRIVATE! and must only be
accessed via the provided functions!
```

## 1.6 newpolycontext

```
filledvector.m/newPolyContext                    filledvector.m/newPolyContext

    SYNTAX

  polycontext:=newPolyContext( bitmap, workspace )

    PURPOSE

  Allocates memmory required for the polygon context and associated
  memory.  A single bitplane the same size as the bitmap is allocated
  for blitting into along with some work memory.

    INPUTS
  bitmap  a standard 'Amiga bitmap' that is to be the destination
    for rendering operations.  This may be changed during runtime
    with the setPolyBitMap() function.
  workspace a number used to allocate workspace for the polygon
    rendering functions.  Currrently make this the maximum number
    of points in any single object, plus 16.

    OUTPUTS
  polycontext
    pointer to a private polgon context handle that can
    be passed to the other functions

    SEE ALSO
  freePolyContext(), Rendering functions
```

## 1.7 freepolycontext

```
filledvector.m/freePolyContext                   filledvector.m/freePolyContext

    SYNTAX

  VOID freePolyContext( poly context )

    PURPOSE
```

```
  Frees the memory associated with a given polygon context.  This
  function should always be called before a program exits.

    INPUTS
  polycontext
    Pointer to a polygon context handle that was previously
    allocated with newPolyContext().  It MUST have
    been allocated with this function.  This value MUST NOT
    be NIL.

    SEE ALSO
  newPolyContext()
```

## 1.8 setpolybitmap

```
filledvector.m/setPolyBitMap                    filledvector.m/setPolyBitMap

    SYNTAX

  VOID setPolyBitMap( poly context, bitmap )

    PURPOSE

  Sets the bitmap that the rendering functions will render into.  This
  bitmap must be either the one used to allocate the polygon context,
  or one with identical structure.

    INPUTS
  polycontext
    Previously allocated (using newPolyContext()), valid polygon
    context.  Must not be NIL.
  bitmap
    Pointer to a standard Amiga bimap that is to become the new
    destination for rendering.

    SEE ALSO
  newPolyContext()
```

## 1.9 setpolycliprect

```
filledvector.m/setPolyClipRect                  filledvector.m/setPolyClipRect

    SYNTAX

  VOID setPolyClipRect( poly context, [minx, miny, maxx, maxy]:INT )

    PURPOSE

  This function sets the viewport through which all rendering will
  take place.  The values supplied, minx/miny/maxx and maxy must
  all be within the size of the bitmap being rendered into, with a
  further restriction that maxx>minx and so on.
```

```
    INPUTS
  polycontext
    Previously allocated (using newPolyContext()), valid polygon
    context.  Must not be NIL.
  [minx, miny, maxx, maxy]:INT
    An array of 4 16 bit numbers that describe the clipping
    rectangle to be used.

    SEE ALSO
  newPolyContext(), Rendering functions
```

## 1.10 setpolyflags

```
filledvector.m/setPolyFlags                          filledvector.m/setPolyFlags

    SYNTAX

  VOID setPolyFlags( poly context, newflags, mask )

    PURPOSE

  This function sets the polygon context flags (see below) for
  the given polygon context.  Not much can be changed just yet.
  Like several Amiga system functions, the mask value sets
  which of the bits in the newflags will become set to those
  values (0 or 1, or whatever).

  PCF_ZCLIP
    This flag will set zclipping on or off.  If this bit
    is on, zclipping will be performed for all polygons
    rendered.  Turning off zclipping can reduce
    calulcations somewhat, but they can't get too close
    to the virtual eye-level of the viewer, without
    stuffing up.

    INPUTS
  polycontext
    Previously allocated (using newPolyContext()), valid polygon
    context.  Must not be NIL.
  newflags
    State of new flags
  mask  A 1 in a bit position of the mask indicates that the
    corresponding bit in the newflags parameter is to be
    copied to the polygon context flags field.  A 0
    indicates that that bit is to remain unchanged.

    SEE ALSO
  newPolyContext()
```

## 1.11 vectorobject

```
This section describes the functions available for creating, destroying,
and working with 'vector objects'.  These are high level object
definitions that can be used to describe quite complex objects, which
can then be rendered very efficiently.

Creating new objects can be difficult, designing objects explains
more information about how to go about this.


  newVectorObject()   create a new vector object
  freeVectorObject()  free a vector object
  copyVectorObject()  make an efficient copy of an object
  cloneVectorObject() make a minimal copy of an object

  getVObjectPoints()  access the array of vertices


  newVList()    create a list header for linking objects
  freeVList()   free the list, and optionally all objects in it
  addVObject()  add an object to an object list
  remVObject()  remove an object from an object list

  sortVList()   do a fast sort in descending Z order of the list


  drawVObject()   draw a single vector object
  drawVList()     draw a list of objects
  moveDrawVList() render a scene of objects

All of these functions operate with system libraries starting from V33
(workbench 1.2).
```

## 1.12 newvectorobject

```
filledvector.m/newVectorObject                    filledvector.m/newVectorObject

    SYNTAX

  vobject := newVectorObject( type, numpoints, numfaces, points, faces )

    PURPOSE

  Creates a new vector object, and initialises its fields to those
  supplied.

    INPUTS
  type  type of object, currently only '0' is valid
  numpoints
    the number of vertices/points in the object
  numfaces
    the number of faces in the object
  points  an array of INT's which are the vertices used by the object.
    Each entry in the list consists of 3 words the X, Y and Z
    coordinates of that point.  There must be at least
```

```
    numpoints*3 INT's in this list.
  faces an array of "vface" data structures which describe the faces
    of the object.  There must be at least as many of these
    as the numfaces argument.  See designing objects for more
    information.

    OUTPUTS
  vobject Pointer to a vobject type OBJECT that can be used to manipulate
    the object in some ways.

    SEE ALSO
  freeVectorObject(), copyVectorObject(), cloneVectorObject(),
  Designing objects
```

## 1.13 freevectorobject

```
filledvector.m/freeVectorObject                filledvector.m/freeVectorObject

    SYNTAX

  VOID freeVectorObject( vobject )

    PURPOSE

  Free's a vector object, and any associated memory.  This function
  should be used to free all objects previously created or copied,
  before the program exits.

    INPUTS
  vobject previously allocated vector object (either using
    newVectorObject(), copyVectorObject() or cloneVectorObject()),
    it may also be NIL, in which case it does nothing.

    SEE ALSO
  newVectorObject(), copyVectorObject(), cloneVectorObject(),
```

## 1.14 copyvectorobject

```
filledvector.m/copyVectorObject                filledvector.m/copyVectorObject

    SYNTAX

  vobject := copyVectorObject( vobject )

    PURPOSE

  Makes an efficient copy of a vector object.  The entire object is
  copied, including the points, the face definitions (colours etc),
  but the polygon definitions are not copied to save space.

    INPUTS
  vobject previously allocated vector object (either using
```

```
    newVectorObject(), copyVectorObject() or cloneVectorObject()),
    it may also be NIL, in which case it returns NIL.

    OUTPUTS
  vobject a copied vector object.

    SEE ALSO
  newVectorObject(), freeVectorObject(), cloneVectorObject()
```

## 1.15 clonevectorobject

```
filledvector.m/cloneVectorObject                filledvector.m/cloneVectorObject

    SYNTAX

  vobject := copyVectorObject( vobject )

    PURPOSE

  Makes a minimal copy of a vector object.  The points array, and the
  face array are NOT copied.  This allows an identical object to be
  created and positioned independently of the original, but any
  manipulation of either object (colours or points) will affect the
  other.

    INPUTS
  vobject previously allocated vector object (either using
    newVectorObject(), copyVectorObject() or cloneVectorObject()),
    it may also be NIL, in which case it returns NIL.

    OUTPUTS
  vobject a cloned vector object.

    SEE ALSO
  newVectorObject(), freeVectorObject(), copyVectorObject()
```

## 1.16 getvobjectpoints

```
filledvector.m/getVObjectPoints                 filledvector.m/getVObjectPoints

    SYNTAX

  points := getVObjectPoints( vobject )

    PURPOSE

  Allows a pointer to the internal point/vertice array to be obtained.
  This can be used with the  and other manipulation
  functions (you are free to do what you like) for post-processing the
  object.

  This array will contain as many points as the number of points used
```

```
  in the argument to the initial newVectorObject() call.

    INPUTS
  vobject previously allocated vector object (either using
    newVectorObject(), copyVectorObject() or cloneVectorObject()),
    it may also be NIL, in which case it returns NIL.

    OUTPUTS
  points  A pointer to an array of INT's that are the 3d points of the
    object.  If the object has been cloned, this will point to
    the same physical array as the parent's point list.

    SEE ALSO
  newVectorObject()
```

## 1.17 newvlist

```
filledvector.m/newVList                                filledvector.m/newVList

    SYNTAX

  vlist := newVList()

    PURPOSE

  Creates a new list header,and initialises it.  This is used for
  linking several objects into a bigger object, or as a way of
  efficiently manipulating several objects at a time.

    OUTPUTS
  vlist pointer to a newly allocated vlist header, or NIL incase of
    failure.

    SEE ALSO
  freeVList(), , ,
  , ,
```

## 1.18 freevlist

```
filledvector.m/freeVList                               filledvector.m/freeVList

    SYNTAX

  VOID freeVList( vlist, freenodes )

    PURPOSE

  Free's a vector object list header, and optionally, all of the
  vector objects connected to the list.

    INPUTS
  vlist pointer to a previously allocated vlist header, allocated with
```

```
    , this must not be NIL!
  freenodes
    Boolean value which indicates whether all of the objects in
    the vlist are also to be free'd.

    SEE ALSO
  newVList()
```

## 1.19 addvlist

```
filledvector.m/addVList                                filledvector.m/addVList

    SYNTAX

  VOID addVList( vlist, vobject )

    PURPOSE

  Adds a vector object to the vlist.

    INPUTS
  vlist pointer to a previously allocated vlist header, allocated with
    , this must not be NIL!
  vobject previously allocated vector object (either using
    newVectorObject(), copyVectorObject() or cloneVectorObject()),
    it may also be NIL, in which case it returns NIL.

    SEE ALSO
  newVList(),
```

## 1.20 remvlist

```
filledvector.m/remVList                                filledvector.m/remVList

    SYNTAX

  VOID addVList( vlist, vobject )

    PURPOSE

  Removes the vector object from the vlist.

    INPUTS
  vlist pointer to a previously allocated vlist header, allocated with
    , this must not be NIL!
    (currently, this field is unused, since the list uses a doubly
    linked list.  If int he future, this changes, then this
    will become important)
  vobject a vobject that has previosly been added to the vlist using
    .

    SEE ALSO
```

```
  newVList(),
```

## 1.21 sortvlist

```
filledvector.m/sortVList                                filledvector.m/sortVList

    SYNTAX

  VOID sortVList( vlist )

    PURPOSE

  Takes the vlist argument, and scans all of the vector objects on
  the list.  It looks at the vobject.pz value, and uses this to
  sort the list indescending Z order.

  The algorithm used by this sort is a modified mergesort, which
  uses a prescan stage to break the list into already-sorted
  sublists, which it then takes pairs of, and merges to create
  larger sublists, until sorted.  This will mean that a nearly
  sorted list can be sorted VERY quickly.  Even with a completely
  reversed list, the properties of the mergesort algorithm
  guarentee a very fast worst case performance.

    INPUTS
  vlist pointer to a previously allocated vlist header, allocated with
    , this must not be NIL!
  vobject a vobject that has previosly been added to the vlist using
    .

    SEE ALSO
  newVList(),
```

## 1.22 drawvobject

```
filledvector.m/drawVObject                              filledvector.m/drawVObject

    SYNTAX

  VOID drawVObject( polygon context, vobject )

    PURPOSE

  Draws a single vector object into the polygon context described.
  The object's position and angles are taken into account, resulting
  in a positioned and rotated object being rendered into the
  destination bitmap.

    INPUTS
  poly context
    Pointer to a polygon context handle that was previously
    allocated with newPolyContext().  It MUST have
```

```
    been allocated with this function.  This value MUST NOT
    be NIL.
  vobject previously allocated vector object (either using
    newVectorObject(), copyVectorObject() or cloneVectorObject()),
    it must NOT be NIL.

    SEE ALSO
  Polygon Context, Vector Objects
```

## 1.23 drawvlist

```
filledvector.m/drawVList                                filledvector.m/drawVList

    SYNTAX

  VOID drawVList( polygon context, vlist )

    PURPOSE

  Scans the vector object list, and draws all of the items contained
  within it.  No sorting of the objects is done whatsoever, so if
  you wish to have correctly depth sorted (i.e. painters algorithm)
  objects,  must be called first.

  Each object is rendered using its position and angles as specified
  in the vobject OBJECT.

    INPUTS
  poly context
    Pointer to a polygon context handle that was previously
    allocated with newPolyContext().  It MUST have
    been allocated with this function.  This value MUST NOT
    be NIL.
  vlist pointer to a previously allocated vlist header, allocated with
    , this must not be NIL!

    SEE ALSO
  newVList(), Polygon Context, Vector Objects
```

## 1.24 movedrawvlist

```
filledvector.m/moveDrawVList                    filledvector.m/movewDrawVList

    SYNTAX

  VOID movewDrawVList( polygon context, vlist, position )

    PURPOSE

  This is a high level function which performs a lot of processing
  in one step.  Initially, it scans the list of objects, and
  uses the supplied position OBJECT to rotate and position all
```

```
  of the objects into an internal list.  Once this has taken place,
  this list is depth sorted, and all objects in the list are
  rendered, starting from the back.

  The positions within each vobject now become relative to the
  position supplied as an argument above.  The angles should
  be relative too, but currently the angles stored in each vobject
  are ignored when this function is called.

    INPUTS
  poly context
    Pointer to a polygon context handle that was previously
    allocated with newPolyContext().  It MUST have
    been allocated with this function.  This value MUST NOT
    be NIL.
  vlist pointer to a previously allocated vlist header, allocated with
    , this must not be NIL!
  position
    an OBJECT position object, which describes the position
    and angle at which the object list is to be drawn.

    SEE ALSO
  newVList(), Polygon Context, Vector Objects
```

## 1.25 rendering

```
  Currently, the only available rendering functions are
  those in the vector object section.

  Sometime in the future, some more low-level, but simple
  to use polygon rendering functions will be provided.
```

## 1.26 matrix

```
This section describes the range of general purpose matrix-based functions
that are available for manipulating sets of points in 2d and 3d.

Have a look at matrix information on just how to use these
functions.

  setMatIdent()   setup a matrix to do nothing (identity matrix)
  setMatRotate()  setup a matrix to perform a rotation
  setMatScale()   setup a matrix to perform a scaling operation

  matSize()       scale the rows of a matrix
  matMult()       multiply two matrices

  matApply3()     apply a matrix to a set of 3d points

All functions here work with all machines.
```

## 1.27 matrixinfo

So, there's all of these matrix functions - just what the hell do you do with
them? Well, matrices are an efficient way to manipulate things like points
in 3d. You can do things like scaling, rotation, shearing, and so on all using
a single matrix. You can also combine operations, like doing several rotations
at once, or a rotation and a scale, by combining the individual transformations
into one matrix, and then applying this matrix in one go to the points.

Ok, how about an example. Say we wish to rotate several points, and then scale
them up by 100%, in the X direction.

```
DEF matrotate:matrix, matscale:matrix

setMatRotate(matrotate, anglex, angley, anglez);  -> rotate angles
setMatScale(matscale, 2048, 1024, 1024);    -> scale 2x in x
matMult(matscale, matrotate)      -> create new transform

matApply3(matrotate, 10, points, buffer); -> apply the matrix
```

Because the scaling matrix is left-multiplied with the rotation matrix
(the matMult call above), then the new matrix will act as if two seperate
transformations had occured, the first being a rotate, and the second
a scaling.

If the order of multiplication of the matrices was reversed, then the
matrix would represent first a scaling operation, and THEN a rotation.
These two different matrices result in quite different outputs.

This operation (multiplying the two matrices) is known as concatenation,
and is mentioned throughout the function descriptions.

Another way to perform scaling is to use the matSize() function. This
modifies the rows of the matrix directly, and will produce similar
results (and is actually more efficient), but it always acts as if
it is the last operation.

See the various functions for more information about what is available.
A textbook on 3d graphics, or simply one on linear algebra which talks
about affine transformations may also be handy.

## 1.28 setmatident

```
filledvector.m/setMatIdent                          filledvector.m/setMatIdent

    SYNTAX

  VOID setMatIdent( matrix )

    PURPOSE

  Sets up the contents of the matrix pointed to by matrix to
```

```
  the internal identity matrix value.

    OUTPUTS
  matrix  Matrix setup to do nothing.

    SEE ALSO
  ,
```

## 1.29 setmatrotate

```
filledvector.m/setMatRotate                          filledvector.m/setMatRotate

    SYNTAX

  VOID setMatRotate( matrix, anglex, angley, anglez )

    PURPOSE

  Sets up the contents of the matrix pointed to by matrix to
  a rotation matrix which represents the rotations provided
  in the arguments.  This matrix is itself a concatentated
  matrix (but calculated much more efficiently than taking
  3 2d rotations and multiplying them together), and as such
  has certain properties.  For example, the order of rotations
  becomes important - i think the order is anglez, angly then
  anglex.

  If you wish to rotate with other angle ordering, then
  you can make 2d rotation matrices using this function,
  and setting two of the angles to 0, then concatenating
  the resultant matrices yourself (using ).

    INPUTS
  anglex, angley, anglez
    The angles to use as the basis of the rotations.
    0 = 0 degrees, and 512 = 360 degrees.

OUTPUS
  matrix  Matrix setup with a rotation matrix

    SEE ALSO
  ,
```

## 1.30 setmatscale

```
filledvector.m/setMatScale                            filledvector.m/setMatScale

    SYNTAX

  VOID setMatScale( matrix, scalex, scaley, scalez )

    PURPOSE
```

```
  Sets up the contents of the matrix pointed to by matrix to
  a scaling matrix which represents the scaling provided
  in the arguments.

  This provides a true mathematicall accurate scaling operation,
  that can be concatenated and ordered correctly.  For a simpler
  version, see .

    INPUTS
  scalex, scaley, scalez
    The scaling values (fixed point) to be used to setup
    the matrix.  These fixed point values are normalised
    to 1024.  This means a value of 512 will mean a
    halving along that axis, and a value of 2048
    a doubling etc.

    OUTPUTS
  matrix  Matrix setup with a scaling operation

    NOTES
  Dont try to scale above 32 times larger! (32767), negative
  values are also acceptable, and will tend to flip the
  object inside out along that axis (odd results).

    SEE ALSO
  , ,
```

## 1.31 matsize

```
filledvector.m/matSize                                  filledvector.m/matSize

    SYNTAX

  VOID matSize( matrix, scalex, scaley, scalez )

    PURPOSE

  Modifies the matrix, by scaling each of its rows by the
  3 values provided.  This provides a more efficient
  way to add a scaling operation to a matrix, but it always
  acts as if it was the last operation performed on the
  matrix.

    INPUTS
  scalex, scaley, scalez
    The scaling values (fixed point) to be used to setup
    the matrix.  These fixed point values are normalised
    to 1024.  This means a value of 512 will mean a
    halving along that axis, and a value of 2048
    a doubling etc.

    OUTPUTS
  matrix  Matrix modified with the scaling factors above
```

```
    SEE ALSO
```

## 1.32 matmult

```
filledvector.m/matMult                                  filledvector.m/matMult

    SYNTAX

  VOID matMult( source matrix, dest matrix )

    PURPOSE

  Concatenates the two matrix operations, by left multiplying
  (go look in a maths book!) the destination matrix by the
  source matrix, and storing the result in the dest matrix.
  The net result is that the two transformation matrices are
  combined, with the effect being that the resultant matrix
  will represent the operation originally performed by the
  dest matrix, followed by the operation performed by the
  source matrix.

  Maybe an  would help!

    INPUTS
  source matrix
    operation to be performed by the concatenated result
    last.  This matrix is left-multiplied with the
    dest matrix.
  dest matrix
    operation to be performed by the concatenated result
    first.  The result of the entire operation is also
    stored in here

    OUTPUTS
  dest matrix
    the result of the operation is stored in the dest matrix

    NOTES
  The result of any concatenation must fit within the size
  of the numbers used to prevent any errors.  This means
  that the net result of any operation must not result
  in a scaling up of more than 32x.

    SEE ALSO
  , ,
```

## 1.33 matapply3

```
filledvector.m/matApply3                              filledvector.m/matApply3

    SYNTAX
```

```
VOID matApply3( matrix, number, source points, dest points )

  PURPOSE

Multiplies each 3d point in the points list by the matrix,
and stores the result somewhere else.

The matrix can be setup to perform any affine transformation
that it has been setup to (currently functions exist for
setting up scaling and rotation matrices only).

  INPUTS
matrix  matrix containing transformation to apply.
number  the number of points to apply the operation to.
  Currently, must be >0.
source points
  An array of INT's which contains the 3d coordinates
  to process

  OUTPUTS
dest points
  An array, at least as big as number*3 INT's in which
  to store the result.  This value may be the same
  as the source points parameter, if you just wish to
  process an existing list of points.

  SEE ALSO
, , ,
```

## 1.34 Example code

```
I've had this module floating around on my hdd for 6-12 months
actually ... i just needed to gt my finger out and write
all this damn documentation :)

(i was going to recode it using some ideas i've had since then,
but i never got around to it ..!)

Anyway, it means i've had time to come up with some decent
(if still trivial) examples.

The Vxx+ below shows which versions of the OS the examples work
with.  Since some use the ScrBuffer module, they require
Workbench 3.0+ (V39+).

    V33+

A very simple example which demonstrates the basics required to
get a vector object spinning on the screen.  It uses the
cube designed in the , and also some
simple Workbench 1.2 functions to do the page flipping.

  V39+

Another simple demo of the module.  This one also uses the
```

Workbench 3.0+ double buffering routines to make it run smoother,
and allow you to drag the screen.

    V39+

One of the first examples i coded :)  Its a spinning Zed logo,
and uses WB3.0 double buffering, and demonstrates the use
of that, the vector module, multiple part objects, and some
of the matrix routines.  Use the left mousebutton to make it
go away, and right to make it come closer.  (you can still
drag its screen BTW).  Both to exit.

  V37+

An animated workbench backdrop!  Uses an offscreen render bitmap to
draw a picture, which is then blitted to the workbench screen.
This has been tested and works on machines with custom graphics
cards! (if slowly ...)

  V37+

Identical to wb1200, but uses a different object.  The object
in this one was created in imagine, and converted using a
longwinded and labourious process ...