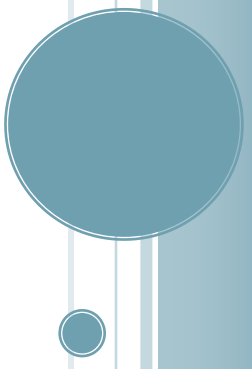
VECTREX32 1.13

User Manual



Contents

1 INTRODUCTION

The Vectrex32 is a cartridge that plugs into the Vectrex Arcade System. Its purpose is to allow people to easily write games for the Vectrex.

The Vectrex32 has a fast, 32 bit microcontroller (a Microchip PIC32) that sends commands to the Vectrex. The PIC32 runs an interactive BASIC interpreter and can be connected to a personal computer via USB.

The latest information about the Vectrex32, including software updates and discussion forums, is at http://Vectrex32.com. There are forums for sharing games, getting tech support, reporting bugs, and requesting new features.

2 ACKNOWLEDGEMENTS

The Vectrex32 would not have been possible without the extraordinary efforts of people who have reverse-engineered the Vectrex. Christopher Salomon and Christopher L. Tumber each wrote tutorials on programming the Vectrex in assembly language. Bruce Tomlin built upon work by Fred Taft to disassemble and comment the Vectrex BIOS. I have made extensive use of their documentation and I thank them heartily.

I also thank the developers of the Vectrex and the writer(s) of the Service Manual and Troubleshooting Manual. Although the Vectrex was a short-lived product, these people should be proud of the work they did.

Finally, many thanks to Dave Lindbergh for reviewing this document and providing valuable feedback, without which several sections would have been incomprehensible.

3 PREREQUISITES

You should have some familiarity with programming a computer. You should read the Galactic Studios BASIC User Manual 1.10 (GSBASIC). You should have a PC or Mac with USB ports. To connect to the Vectrex32, you will need a terminal program.

You *do not* need to read any of the documents mentioned in the Acknowledgements or be familiar with programming the Vectrex in assembly language. In fact, if you know any of that stuff, you'll find that the Vectrex32 alters the programming paradigm significantly.

4 HOW THE VECTREX32 WORKS

The Vectrex32 has a PIC32 microcontroller on it and a dual-port memory chip. The PIC32 writes 6809 machine language instructions into the dual-port memory chip. The 6809 microprocessor in the Vectrex reads those instructions and executes them as if there were a game cartridge there.

The PIC32 does not load an entire game into the dual-port memory. Instead the game logic executes on the PIC32. Thirty times a second or so (it's possible for you to change the rate), the

PIC32 writes 6809 instructions to the dual-port memory. These instructions make the Vectrex draw on the screen, play sounds, or read the game controller status.

Dual-port memory is expensive. The Vectrex32 contains only 2K bytes of it. This would be far too small for a game, but since it's only used for Vectrex commands, it is sufficient for many games. There are no hard-and-fast rules, but 2K should be enough to draw hundreds of vectors. (Whether the Vectrex can draw that many vectors and still refresh the screen fast enough is a separate issue.)

4.1 Technical Specs

The PIC32 model used in the Vectrex32 is the Microchip PIC32MZ2048EFH064. It runs at 200MHz, has a floating point coprocessor, 2MB of Flash memory and 512KB of RAM. The dual port memory chip has a capacity of 2KB. The USB port runs at Full Speed (i.e. the top USB 1.0 speed).

5 Getting Started

5.1 Requirements

You need a text editor, a terminal emulator that can communicate over a USB virtual serial port, a USB mini-B cable, and a Vectrex.

It will be very helpful if your text editor can display line numbers (unlike, for example, Windows Notepad). Whenever GSBASIC detects an error in your program, it will report the line number of the error, so you really want a text editor that shows that.

An excellent tool to use is the Vectrex Integrated Development Environment, or VIDE. It runs on Windows, Macs, and Linux (it's written in Java). It combines a file browser, text editor, and terminal window into one application. You can download it for free at http://vide.malban.de/.

If you would rather use separate applications for the editor and terminal window, read on.

On Windows, I recommend Notepad++. It's free software from https://notepad-plus-plus.org/. For the terminal emulator, I recommend TeraTerm, a free program from https://ttssh2.osdn.jp/index.html.en.

On the Mac, TextWrangler appears to be a good choice for a text editor. It is freely downloadable from http://www.barebones.com/products/textwrangler. The Mac comes with a terminal emulator called 'screen', which can be run from the command line.

If you're a Linux user, I'll assume you already have a capable text editor that you're happy with. As with the Mac, the 'screen' application is a good choice of terminal emulator.

The details of using these programs are beyond the scope of this manual. Refer to their web sites or, in the case of 'screen', its man page.

5.2 Setup

Make sure the Vectrex is turned off. Plug the Vectrex32 into the Vectrex's cartridge slot. The "Vectrex32" logo should be facing up. Plug the USB cable into the Vectrex32 and into your computer. Turn on the Vectrex.

NOTE: Never plug in or remove the Vectrex32 while the Vectrex is turned on.

Be careful not to plug the Vectrex32 in upside down. Vectrex game cartridges are impossible to plug in upside down because of the shape of their plastic cases. The Vectrex32's enclosure, however, does not prevent you from plugging it in upside down. So make sure the Vectrex32 logo is facing up. Turning the Vectrex on while the Vectrex32 is plugged in upside down could damage the Vectrex and the Vectrex32.

The Vectrex32 will appear as two devices on your computer: a mass storage device, like a USB thumb drive, with the label "Vectrex 32", and a virtual serial port (aka a COM port). The drive comes with some sample BASIC programs. See the ReadMe.txt file on the drive for a description of the programs.

Open a terminal emulator (e.g. TeraTerm or screen) and connect to the Vectrex32. On Windows, it may be a challenge to determine which serial port to use. TeraTerm has a pull-down list of available ports (Setup menu -> Serial port...); since you just plugged in the Vectrex32, its serial port is probably the last one on the list. Configure your terminal emulator for 8 data bits, 1 stop bit, no parity. 9600 baud or bps will suffice; you're not transferring large amounts of data through the serial port. Turn off local echo, turn off any line-ending features that add carriage returns or line feeds, and enable VT-100 emulation.

On the Mac the command

```
ls /dev/tty.*
```

will display a list of serial ports. One of them should have "usbmodem" in the name, e.g. /dev/tty.usbmodem1422; that's probably the Vectrex32's serial port. To open a terminal emulator for the Vectrex32, issue the command:

```
screen /dev/tty.usbmodem1422 # use the device from the ls command
```

On Linux, the command

```
dmesg | grep tty
```

will display a list of serial ports to use with the 'screen' command. One of them should have "USB ACM device" in the description, e.g:

```
[ 2437.034588] cdc_acm 2-2.1:1.1: ttyACM0: USB ACM device
```

You can then issue the command:

```
sudo screen /dev/ttyACM0 9600 # use the device from dmesg
```

When you connect to the Vectrex32 with your terminal emulator, you should (usually) see a greeting message and a Ready prompt. If you do not, hit Enter. If you still don't see the Ready prompt, it may be because a program is running (the Vectrex32 comes with a program, Autorun.bas, which runs automatically when the Vectrex32 starts up and displays a menu of BASIC programs). In that case, hit Ctrl+C to break out of the game, then enter the command:

```
stop
```

to stop the program. However, if hitting Ctrl+C doesn't get you to prompt, you probably are using the wrong serial port.

Once you have the Ready prompt, type a GSBASIC command like

```
PRINT 1 + 1
```

Hopefully, you'll get the right answer. Next, you can type

```
load lunar.bas
run
```

and start playing a game.

(Note that you can also type ‘load lunar’; the .bas extension is assumed.)

If you want to end the game early, hit Ctrl+C.

6 VECTREX CONCEPTS

6.1 CRT Displays

The cathode ray tube (CRT) was the most common way to display electronic images from the 1930s until the 1990s. They were used in radars, televisions, oscilloscopes, and computer displays. A CRT is a large vacuum tube that shoots an electron beam at the screen surface, which is covered with phosphor that lights up when hit by electrons. The beam hits just a small spot on the screen, but can be aimed, and turned on and off, electronically.

Because the phosphor lights up only for a brief moment after being hit by the electron beam (and also because the picture may change), this process is repeated 25 to 60 times each second.

Analog television systems (and computer displays based on television technology) paint a picture on the CRT using a system in which the electron beam repeatedly scans the entire screen in rows from top to bottom, with each row being scanned left-to-right. This scan pattern is called a “raster”. The beam is turned on whenever it is over a portion of the picture meant to be bright, and off where it is meant to be dark.

Raster-based computer displays (“raster graphics”) have hundreds or thousands of rows, with each row containing a line of pixels. These rows of pixels correspond to locations in memory – when the memory is changed, the pixels on the display change. The programmer writes the desired image into memory, and the display system hardware automatically draws the raster on

the CRT screen, repeatedly refreshing the picture without further work on the part of the programmer.

6.2 Vector Graphics

Vector graphics systems like the Vectrex also use a cathode ray tube (CRT) for display. But there are no pixels and no raster. Unlike television CRTs, the entire screen isn't scanned row-by-row.

Instead, a vector graphics display aims the electron beam at the place where a graphics object (a character or drawing) is to appear, turns the beam on, and then traces out the shape to be displayed by moving the beam. The beam is then turned off, moved to another location, and another object is drawn the same way.

You can think of it like using a pen on a piece of paper, except that the ink fades very quickly (equivalent to the CRT phosphor lighting only for a moment), so the pen needs to re-draw the picture repeatedly.

In the Vectrex, the screen is typically redrawn between 30 and 60 times per second. The Vectrex's 6809 processor is responsible for doing this.

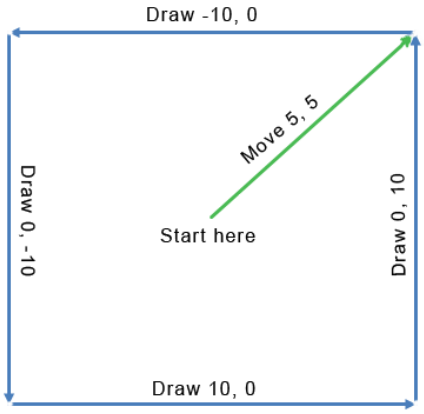
Instead of talking about the electron beam in the Vectrex's CRT, we will simply use the metaphor of the pen, e.g. moving the pen and drawing with the pen.

6.3 Relative Movements

Raster displays use absolute coordinates to identify each pixel on the screen. The Vectrex does not use absolute coordinates to position the pen; everything is relative to the pen's current position. If the pen is at Position A and you tell the Vectrex to move the pen 3 units left and 2 units down, it means 3 units left of Position A and 2 units down from Position A. After the move, the pen is at a new position, Position B. If the next command is to draw a line 5 units right and 1 unit up, that line is drawn from Position B to a position 5 right and 1 up, and again, the pen ends up at a new position, Position C.

On the Vectrex screen, positive X values are to right while negative X values are to the left. Positive Y values are up while negative Y values are down.

To draw a 10x10 square on the screen, a program would do the following:



Assuming we want the square to be centered at the current pen position, the program would first move the pen 5 units up and 5 right. Next, it would draw a line 10 left (written as a negative ten), another line 10 down (-10), a third line 10 right, and the last line 10 up. When done, the pen will be at the top right corner of the square. The commands would look like this:

- Move (5, 5)
- Draw (-10, 0)
- Draw (0, -10)
- Draw (10, 0)
- Draw (0, 10)

Moves and Draws always start at the current pen position and, when the Move or Draw is complete, the pen will be at a new position.

It can be difficult to keep track of where the pen is. So the Vectrex provides a command that moves the pen to the center of the screen. Vectrex programs generally execute this command each time they redraw the display; often, they also execute the command after drawing one shape and before drawing another.

6.4 Units and Scales

When the Vectrex executes a command to move the pen or draw a line, it uses integer X and Y values in the range -128 to +127 (these conveniently fit within the 8 bits the Vectrex's 6809 microprocessor uses). Those values specify a position *relative to* the current pen position ("relative coordinates"). The distance moved by the pen in those units (called the "magnitude" of the line) is given by the Pythagorean Theorem ($\sqrt{X^2 + Y^2}$).

But the magnitude doesn't tell you the length of the line (or the distance moved) in physical inches on the screen. To determine that, there is also a scaling factor.

The Vectrex designers had to decide how long a line with a magnitude of 127 would be. The Vectrex screen is about 10 inches from corner to corner. If a 127 unit line stretched for 10 inches, then a 1 unit line would be about 0.08 inches long. That may sound small, but in practice, it's too big for drawing fine detail or precisely positioning objects. Alternatively, if a 127 unit line stretched for only 1 inch, you could draw fine detail, but you couldn't draw a single line across the screen. You could draw lots of shorter lines, but that requires more programming and won't look as good on the screen (the Vectrex tends to leave a bright dot at the end of lines).

The solution the Vectrex designers came up with was to allow lines and pen movements to use different scales; before moving and drawing, you tell the Vectrex what scale to use. The scale is an integer from 1 to 255. For example, at a scale of 44, a line with a magnitude of 127 is about 1 inch long. That means that each unit is 1/127" or about 0.009 inches. At a scale of 88, a line with a magnitude of 127 is about 2 inches long, so each unit is about 0.018 inches. The scale's effect on line length is roughly linear: double the scale and you approximately, *but not exactly,* double the length of the lines.

> Why is it not exact? You don't need to understand this explanation, but if you're familiar with electronics, here it is: The scale value is used as a timer. The x and y values are used as inputs to a digital-to-analog converter, whose output charges a capacitor. The voltage on the capacitor controls the

position of the electron beam in the CRT. But the capacitors don't charge at a linear rate. So, doubling the time the capacitor charges does not quite double the deflection of the electron beam.

If you want to draw fine detail, or position something with great precision, use a small scale. If you want to draw a long line that crosses the screen, use a large scale. If you need to move the pen a large distance, and then draw an object with fine detail, you can set a large scale, move the pen, then set a small scale and draw the object.

6.5 Pen Drift

Imagine you walk ten steps north, five steps east, ten steps south, and five steps west. In theory, you should end up back where you started. In reality, you'll probably be a little off. And if you do it multiple times, each time you will be off by a different amount.

The same is true of drawing on the Vectrex. With each line drawn and each movement of the pen, a little bit of inaccuracy accumulates in the pen position. Every time the screen is redrawn, the drift is a little different. If nothing is done to prevent this, objects will wave back and forth erratically around their average position. The solution is to periodically execute the Vectrex command that returns the pen to the center of the screen. This eliminates any accumulated drift.

6.6 Frames

Vectrex programs draw their display, check for inputs (button presses, joystick movements), and adjust the objects on the screen accordingly, then repeat this process. Each time the screen is drawn, it is a new "frame". Frames should be drawn at a constant rate, typically 30 frames per second.

6.7 Intensity

Intensity determines how bright objects on the screen will appear. A program sets the intensity of the electron beam, and all drawing done after that will be done at the specified intensity.

7 VECTREX32 CONCEPTS

7.1 Overview

The Vectrex32 has a 32 bit PIC32 processor running a BASIC interpreter. It shares 2K of dual-port memory with the 6809 processor in the Vectrex: the PIC32 and the 6809 can each read and write to that memory. When the Vectrex is turned on, the PIC32 writes a small 6809 program into the dual-port memory. The 6809 sees that program, thinks it's a game on a cartridge, and starts running it.

This program does nothing but wait for a signal from the PIC32. When a BASIC program on the PIC32 wants to draw a new frame, the PIC32 copies 6809 code into the dual-port memory and sends a signal telling the 6809 to run the code. In this way, the Vectrex32 sends commands to the Vectrex telling it to draw, make sounds, and check the status of the joystick and buttons.

All the concepts described in the previous section (Vectrex Concepts) are used in BASIC programs written for the Vectrex32. But the Vectrex32 adds features that make programming far easier than writing 6809 code for the Vectrex.

7.2 The Drawing List

As mentioned earlier, the 6809 in the Vectrex must redraw the screen frequently; thirty times per second is a good rate. Vectrex games must have a part of their program that does this work. The Vectrex32 does this work automatically; it has an operating system (OS) that sends commands to the Vectrex to redraw the screen for you. Your program just sets up a list of things to draw, and the Vectrex32 OS will draw them at the beginning of each frame. You alter the list as needed during the game.

Experienced Vectrex programmers will spot a disadvantage to this approach: with the old way of programming the Vectrex, when you wanted to make an object disappear from the screen (e.g. a ship was blown up) you simply didn't redraw it anymore, and it went away. But in the Vectrex32, you need to explicitly remove or disable the object in the drawing list so it is not drawn.

But the drawing list offers a compelling advantage: when you're writing and testing a program, you can stop execution by hitting Ctrl+C, and the display continues to be drawn even though your program has stopped running. You can then debug or experiment by typing in commands that modify the drawing list and see those changes immediately on the screen. The drawing list allows you to tweak lines, shapes, and distances interactively and see the results immediately.

Items in the drawing list are called "sprites". There are regular sprites and pseudo-sprites. A regular spite is drawn on the screen, e.g. lines, dots, or text. A pseudo-sprite is not drawn, but it is an instruction that changes how subsequent sprites in the list are drawn. The available sprites are:

Regular Sprites	Description
Dots sprite	Draws a series of dots on the screen
Lines sprite	Draws a series of lines on the screen
Text sprite	Draws one line of text on the screen
TextList sprite	Draws multiple lines of text on the screen
Lines3D sprite	Draws a series of 3D lines on the screen

Pseudo-sprites	Description
Intensity sprite	Sets the intensity for subsequent Dots, Lines, Text, and TextList sprites
Move sprite	Moves the pen
ReturnToOrigin sprite	Resets the pen to the center of the screen
Scale sprite	Sets the scale for subsequent Dots and Lines sprites and Move pseudo-sprites
TextSize sprite	Sets the size of subsequent Text and TextList sprites

Sprites are objects in GSBASIC, i.e. they are not integers, floats or strings, but they contain multiple pieces of information. To work with a sprite, we use a variable that stores a value that refers to the object; this is called a “handle”. (The Galactic Studios BASIC manual discusses objects and handles in more detail.)

Sprite creation functions create a sprite, add it to the end of the drawing list, and return a handle to the sprite. For example:

```
ts = TextSprite(“HELLO”)
```

creates a Text sprite, adds a its handle to the end of the drawing list, and also places a handle in the variable *ts*. The Text sprite will cause "HELLO" to be drawn at the current pen position i.e. whatever position the pen ended up at due to previous sprite(s) in the drawing list; if the Text sprite is the first item in the drawing list, then "HELLO" will be drawn starting at the center of the screen, since the Vectrex32 always resets to pen to the center of the screen before drawing.

Here's an example of setting up a simple drawing list:

```
call IntensitySprite(127)
call ScaleSprite(50)
call MoveSprite(5, 10)
call TextSprite("HELLO")
```

When this drawing list is processed at the beginning of each frame, it will generate 6809 code that resets the pen to the center of the screen (this is done automatically by the Vectrex32 OS; it is not necessary to put a ReturnToOrigin pseudo-sprite at the beginning of a drawing list), sets the intensity, sets the scale, moves the pen 5 units right and 10 units up, and draws the word HELLO at that position. That code will be copied to the dual-port memory and the 6809 will be signaled to execute it.

7.3 The Structure of a Program

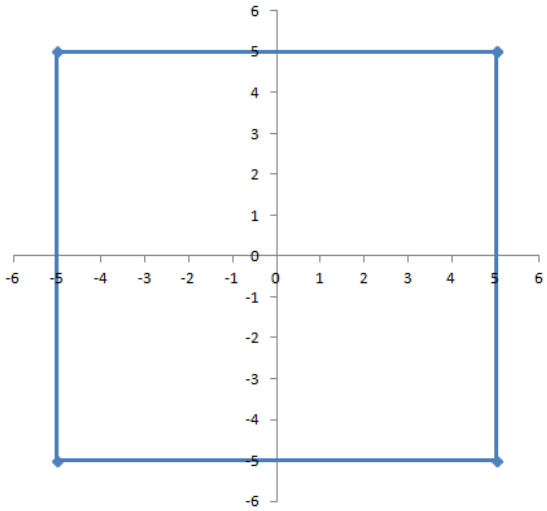
A typical BASIC program on the Vectrex32 will start by constructing a drawing list. Then it will enter an infinite loop.

Inside the loop, the program will call the WaitForFrame function. WaitForFrame will first generate 6809 instructions to read the joystick and buttons, and to reset the pen to the center of the screen. It will then compile the drawing list into 6809 instructions to draw the frame. Finally, WaitForFrame will wait until it is time to start the next frame, at which time it will send a signal to the 6809 to execute the instructions. When WaitForFrame gets a response from the Vectrex with the joystick and button information, it will return to the BASIC program.

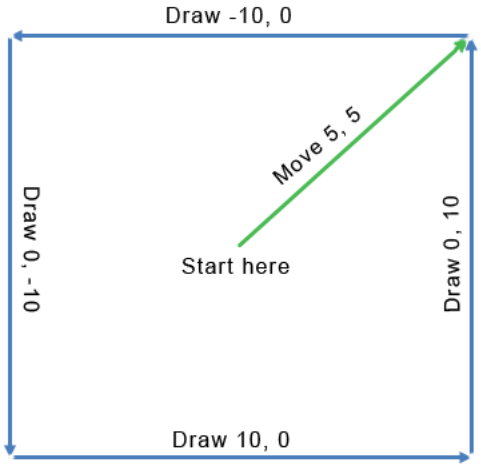
In the rest of the loop, the BASIC program will decide how to respond to the joystick and buttons, modify the drawing list in preparation for the next frame, and set up any sounds that should be played.

7.4 Local Coordinates

As mentioned in section 6.3, Relative Movements, X and Y values in native Vectrex Move and Draw commands are relative to the current pen position. It's confusing for a programmer to use this system. In the example given in Relative Movements, we drew a 10x10 square. If you were to draw a 10x10 square on a graph, it would be:



It's easy to see that the coordinates of the vertices are (5, 5), (-5, 5), (-5, -5), and (5, -5). But when drawing with relative movements, we did:



.

The relative coordinates we passed to the commands were Move (5, 5), Draw (-10, 0), Draw (0, -10), Draw (10, 0), and Draw (0, 10).

Converting the coordinates of vertices into relative movements can be difficult, especially when the shape is more complex than a square. So the Vectrex32 has a feature that does that conversion for you.

Line sprites and Dot sprites in the Vectrex32 use *local coordinates* instead of relative coordinates to make it easier to define the sprite. The local coordinates in the sprite are simply the coordinates of a shape's vertices. To create a Lines sprite that draws that square with edges 10 units long, centered on the middle of the square, we would write:

```
' Create a Lines sprite and put it on the drawing list
square = LinesSprite({ _
   {MoveTo, 5, 5}, _
   {DrawTo, -5, 5}, _
   {DrawTo, -5, -5}, _
   {DrawTo, 5, -5}, _
   {DrawTo, 5, 5}})
```

The local coordinates are (5, 5), (-5, 5), (-5, -5), (5, -5), (5, 5) - which is just the coordinates of the corners of the square. Before the Vectrex32 sends those coordinates to the Vectrex, it will convert them into the relative coordinates (5, 5), (0, -10), (-10, 0), (0, 10), (10, 0). The square's center will be at the current pen location and after the square is drawn, the pen will be at the end of the last line of the square (in this case, 5 units right and 5 units up from where the pen started).

7.5 Positioning a Sprite on the Screen

It's a chore to keep track of where the pen is when drawing many different sprites. Consider an Asteroids game: there are many asteroids on the screen and you need to draw them. Drawing the first is easy: reset the pen to the center of the screen, move the pen to the asteroid's current position, and draw it. The pen ends up located at the end of the last line drawn. How do you move it to the second asteroid's position?

You could do it the hard way: take the first asteroid's position, add the coordinates of the last point in the Lines sprite, and subtract that from the second asteroid's position. The result is the relative coordinates for a Move sprite.

Or you could do it the easy way: reset the pen to the center of the screen by putting a ReturnToOrigin pseudo-sprite on the drawing list, then move to the second asteroid's position and draw it. Since you reset to a known location before drawing each object, the relative coordinates that the Vectrex needs are just the position of the asteroid.

Not only is this the easiest way mathematically, but it also eliminates pen drift. Its one disadvantage is that it's slower: instead of moving the pen directly to the second asteroid's position, you're first moving the pen to the center of the screen, then moving it again to the asteroid's position; moving the pen takes time. If your game is still fast enough, don't worry about it. If it's not, keep in mind that doing it the hard way is an option.

In an Asteroids game, the asteroids move from frame to frame, so whether you choose the easy way or the hard way, for each frame you need to alter the coordinates that the Move sprite uses. For example, you could set up your drawing list like this:

```
' Move to the asteroid's position. We'll decide later where
' the asteroid should be when the game starts, so for now
' just create a Move sprite with the coordinates (0, 0)
asteroid1Position = MoveSprite(0, 0)

' Draw the asteroid
asteroid1 = LinesSprite(asteroidPoints)
```

Then, for each frame of the game, you would calculate the asteroid's new position and alter the coordinates in the Move sprite:

```
call SpriteMove(asteroid1Position, newXCoord, newYCoord)
```

7.6 Advanced Sprite Features

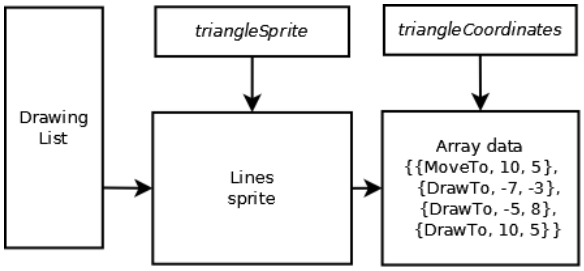
7.6.1 Live Data

Lines sprites, Dots sprites, and TextList sprites use "live data", which means that changing the data will change what appears on the screen. Consider this code that adds a Lines sprite to the drawing list:

```
triangleCoordinates = { _
   {MoveTo, 10, 5}, _
   {DrawTo, -7, -3}, _
   {DrawTo, -5, 8}, _
   {DrawTo, 10, 5}}
triangleSprite = LinesSprite(triangleCoordinates)
```

We're creating and initializing the *triangleCoordinates* array with the dimensions [4, 3]. Each of the 4 rows has three elements: a mode (the predefined constants MoveTo or DrawTo), an X coordinate and a Y coordinate (as mentioned in section 7.4, Local Coordinates, X and Y are local coordinates, so each coordinate is relative to the origin of the triangle). Then, the LinesSprite function creates a Lines sprite with the data from the array. The LinesSprite function returns a handle to the sprite and also adds the sprite to the end of the drawing list. As a result, a triangle will appear on the screen. (We use four coordinates to define a triangle because the first coordinate is moving the pen, not drawing a line).

At this point, data in memory looks like this:



The *triangleCoordinates* variable points to the array of coordinates. The *triangleSprite* variable is a handle to the Lines sprite. The drawing list also has a handle to the Lines sprite. The Lines sprite points to the array of coordinates.

The Vectrex32 will automatically redraw the Lines sprite at the beginning of each frame. The Lines sprite will get the data from the *triangleCoordinates* array, process it (as described in the following section on Transforms and Clipping), convert it to relative coordinates, and compile it into commands for the Vectrex. Notice that the *triangleCoordinates* array is read every time the Lines sprite is redrawn; a handle to the sprite is on the display list, but the data for the sprite is still in the *triangleCoordinates* array! This means your program can change the values in the *triangleCoordinates* array and the changes will show up on the screen. You can write:

```
triangleCoordinates[2, 2] = 30
```

and it will change the Y coordinate one of the vertices of the triangle.

GSBASIC allows you to assign new values to variables that hold arrays. After running the code above, you could have lines of code like:

```
triangleCoordinates = {1, 2, 3}
```

or

```
triangleCoordinates = 5
```

If you do that, the Lines sprite will still be using the array you originally passed in the LinesSprite function, however the *triangleCoordinates* variable will no longer point to that array, so your code will no longer be able to access the elements of that array.

Similarly, if you change the value of *triangleSprite*:

```
triangleSprite = 0
```

the sprite will still be in the drawing list; but the variable *triangleSprite* will no longer have a handle to it.

Lines sprites, Dots sprites, and TextList sprites all use live data. Text sprites do not use live data, i.e. a Text sprite's text cannot be changed after the text sprite is created.

7.6.2 Transforms and Clipping

Lines sprites and Dots sprites can be magnified, rotated, translated, and clipped. Text and TextList sprites do not support these features.

Each Lines sprite and Dots sprite has hidden magnification, rotation, translation, and clipping properties. When sending drawing list data to the Vectrex, the Vectrex32 examines these properties and modifies the data being sent for drawing accordingly.

To make transformations and clipping more accurate, the local coordinates in Lines and Dots sprites need not be integers; they may be floating point values as well. When the drawing list is processed, local coordinates are converted to floating point (if they are not floating point already), transformed (magnified, rotated, translated and clipped), converted to relative coordinates, then rounded to integers before being sent to the Vectrex. If you are not scaling or rotating the sprite, using floats for its coordinates will provide no benefits. But if you designed a sprite by drawing its shape on graph paper and one of the vertices ended up in between two integer coordinates, and if you *are* scaling or rotating, then specifying floating point coordinates might improve the sprite's appearance on the screen.

The following sections describe functions you can call to set and get the values of these transformation and clipping properties.

7.6.2.1 Magnification

If we take *triangleSprite* from the previous example, we can write:

```
call SpriteSetMagnification(triangleSprite, 2.0)
```

This magnifies (or shrinks) the sprite by the factor given in the second parameter. In this example, the sprite gets bigger in both X and Y directions by a factor of 2 (so 4 times bigger in area).

Sprites have a magnification of 1.0 when they are first created.

Remember that a Lines sprite uses live data. This call to magnify the *triangleSprite* does not alter the data in the *triangleCoordinates* array. Instead, each time the Vectrex32 compiles the drawing list into commands for the Vectrex, it gets the values from the *triangleCoordinates* array, multiplies them by the magnification, and send the resulting coordinates to the Vectrex.

Each call to SpriteSetMagnification sets a new magnification value for the sprite:

```
' Create the sprite; it will have a magnification of 1.0
triangleSprite = LinesSprite(triangleCoordinates)

' Change the sprite's magnification to 2.0:
call SpriteSetMagnification(triangleSprite, 2.0)

' Change the sprite's magnification to 0.5:
call SpriteSetMagnification(triangleSprite, 0.5)
```

The difference between magnification and scale might be confusing. Magnification is applied to coordinates before they are sent to the Vectrex. Scale is used by the Vectrex, after receiving the drawing commands, to determine how long each unit of the X and Y coordinates is.

Note that magnification can result in coordinates outside the range of -128 to 127 that the Vectrex allows. For example, if a coordinate in the sprite is 80, and the sprite's magnification is set to 2.0, the coordinate will end up being 160 - too large for the Vectrex to handle. Section 7.6.3, Large Coordinate Values, describes how the Vectrex32 deals with this situation.

7.6.2.2 Rotation

If we take *triangleSprite* from the previous section, we can write:

```
' Rotate the triangle by 30 degrees counterclockwise:
call SpriteRotate(triangleSprite, 30)

' Rotate by another 25 degrees:
call SpriteRotate(triangleSprite, 25)

' Get the current rotation, which will be 55 degrees:
rotation = SpriteGetRotation(triangleSprite)

' Set the rotation to a specific value:
call SpriteSetRotation(triangleSprite, 10)

' Get the current rotation, which will be 10 degrees:
rotation = SpriteGetRotation(triangleSprite)
```

As with magnification, calls to rotate the sprite do not alter the data in the *triangleCoordinates* array. Each time the Vectrex32 refreshes the display, it gets the values from the *triangleCoordinates* array, applies the rotation, then draws the result on the screen.

Sprites have a rotation angle of zero degrees when they are first created.

You can use two rotation functions: SpriteRotate and SpriteSetRotation. SpriteRotate(angle) adds an angle to the sprite's current rotation angle, while SpriteSetRotation(angle) sets the sprite's rotation to specified angle. Angles are in degrees. As with trigonometric functions, positive values rotate counterclockwise and negative values rotate clockwise.

7.6.2.3 Translation

Sprites can also be translated. This simply adds an X and Y offset to each pair of coordinate points, which moves the position of the sprite on the display. As with magnification and rotation, the data in the sprite array isn't affected.

Note that the translation is applied *after* the magnification and the rotation, so the X and Y translation values you use are not themselves magnified or rotated.

```
offset = {3, -4}
call SpriteTranslate(triangleSprite, offset)
```

This example causes the magnified and rotated sprite to be moved 3 units left and 4 units down. The translation array, *offset*, is live data: you can change *offset[1]* and *offset[2]* and the sprite will move accordingly.

Sprites have a translation offset of (0, 0) when they're first created.

Translation has the effect of moving where the sprite is drawn. If you have the following code:

```
' Assume the pen is at Position A

' Add a sprite to the drawing list to move the
' pen 10 right and 10 up
call MoveSprite(10, 10)

' Add a sprite to the drawing list to draw a shape
shape = LinesSprite(linePoints)
```

then the shape is drawn 10 units right and 10 up from Position A. If you then call:

```
call SpriteTranslate(shape, 3, 3)
```

The shape will be drawn 13 units right and 13 up from Position A. So why use translation instead of just changing the coordinates in the Move pseudo-sprite? The reason is that clipping, described in the next section, can make use of translations but not moves.

We're seen three ways to change where an object appears on the screen:

Move pseudo-sprite	This moves the pen a relative distance from its current location. The maximum movement is between -128 and 127 units. Since this is a sprite on the drawing list, it can be preceded and followed by Scale pseudo-sprites; i.e. the scale used for the Move sprite can be different from the scale used for subsequent sprites. The coordinates for a Move pseudo-sprite tell the Vectrex to move the pen relative to its current position.
Lines sprites with MoveTo coordinates	The MoveTo coordinates are not intended to be used for moving the sprite. They are useful if the first vertex of the sprite is not at the center of the sprite. They are also useful if the sprite cannot be drawn as a continuous line, i.e. if you need to "lift the pen up off the paper" while drawing the shape. The coordinates in a MoveTo are interpreted at the same scale as the other coordinates in the Lines sprite. The coordinates for a MoveTo are Local Coordinates, i.e. they are relative to the sprite's origin. Coordinates for a MoveTo are not limited to the -128..127 range (see Section 7.6.3, Large Coordinate Values)
Translation	Translation offsets are added to each coordinate in a Lines or Dots sprite. The coordinates are interpreted at the same scale as the other coordinates in the sprite. Translation is intended to be used with clipping (see section 7.6.2.4). Clipping would not work with Move sprites. Coordinates for a translation are not limited to the -128..127 range (see Section 7.6.3, Large Coordinate Values

7.6.2.4 Clipping

The final operation applied to the drawing data the Vectrex32 sends to the Vectrex is clipping. Imagine you're writing a space combat game, where the player is in a cockpit looking out the window. When an enemy ship flies out of the field of view, you don't want to draw it anymore. That's easy: just check whether it's beyond the boundaries of the window.

But what about when half the ship is still visible through the window and half is not? That's where clipping is useful: define the window as a clipping rectangle, apply it to the enemy ship sprite, and the sprite lines that are outside of the clipping rectangle will not be drawn. If part of a line is outside the rectangle and part of it is inside, the line is "clipped": only the inside part is drawn.

Set a clipping rectangle thusly:

```
' Define the clipping rectangle
clippingRect = {{x1, y1}, {x2, y2}}

' Apply clipping to the spaceship sprites
call SpriteClip(enemyShip1, clippingRect)
call SpriteClip(enemyShip2, clippingRect)
call SpriteClip(enemyShip3, clippingRect)

' Turn off clipping
call SpriteClip(enemyShip1, nil)
call SpriteClip(enemyShip2, nil)
call SpriteClip(enemyShip3, nil)
```

Notice that multiple sprites can use the same clipping rectangle.

The process for clipping a sprite is:

- Fetch the local coordinates of the sprite
- Apply magnification and rotation to the coordinates
- Apply translation to the coordinates.
- Clip the coordinates so that only the dots or lines (or parts of lines) inside the clipping rectangle remain.
- Convert the clipped coordinates to relative coordinates and send them to the Vectrex to be drawn.

Notice that the current pen position and the scale do not affect clipping. Clipping is a purely mathematical process done by the Vectrex32. And so it is your program's responsibility to ensure that the pen position is such that clipping looks right when everything is drawn on the Vectrex.

In the case of the space combat game, this would mean drawing the window and all the enemy spaceships from the same initial pen position and at the same scale. The easiest way to do this is to reset the pen to the center of the screen, draw the cockpit window, reset to the center of the screen again, draw the first enemy ship, reset to the center of the screen a third time, draw the second enemy ship, and so on. The positions of the enemy ships would be determined by the translation offset, not by a Move sprite. Remember that translation uses live data: you can change the position of each enemy ship by changing its translation.

Clipping uses live data too, so you can change the clipping rectangle by changing the array elements. The coordinates can be in any order, e.g. {{left, bottom}, {right, top}} or {{right, bottom}, {left, top}} or {{left, top}, {right, bottom}}, etc.

The sample program lunar.bas, included on the Vectrex32's drive, uses clipping another way. As the lunar module flies over the terrain, the terrain smoothly scrolls to the left or right. This is accomplished by creating a Lines sprite with the terrain and setting a clipping rectangle that is as large as the entire screen (at the specified scale). To scroll the terrain, the translation's X coordinate is changed; the terrain shifts left or right accordingly and the lines are clipped to eliminate the parts that are off the edges of the screen.

7.6.3 Large Coordinate Values

The Vectrex only accepts coordinate values between -128 and 127. When making Lines sprites and Dots sprites, it's easy to exceed those values, especially when applying a magnification to them (e.g. if a sprite has a coordinate of 50 and a magnification of 3, the coordinate becomes 150) or applying large translations to them (which can be necessary if you're doing clipping). The Vectrex will not correctly draw lines whose coordinates are outside of the range -128 to 127. So the Vectrex32 checks for out-of-range coordinates in sprites and breaks them up into multiple lines or multiple moves. So if you applied a translation of 150 units to a sprite, the Vectrex32 would convert that to two move commands for the Vectrex, each 75 units long.

Similarly, if a sprite has a line that's 50 units long and you apply a magnification of 3.0 to it, the result is a line that's 150 units long. The Vectrex32 would draw that as two lines, each 75 units long. Two 75 unit lines appended to each other don't look as good on the Vectrex as a single line (the Vectrex tends to leave a bright dot at the end of each line), so you should not rely on this feature in your finished game; if a line is getting too long, switch to a larger scale.

Note that this feature only exists for Lines and Dots sprites, not for Move pseudo-sprites.

7.7 Built-In Function Names

The names of some of the built-in functions can be confusing, but there is a consistency to them. Consider the functions ScaleSprite and SpriteScale. The first creates a new Scale pseudo-sprite and the second changes the scale of an existing Scale pseudo-sprite.

The rule is that a function that creates a sprite starts with the type of sprite it is creating, e.g. ScaleSprite, IntensitySprite, LinesSprite, etc. A function whose name starts with the word "Sprite" will perform an operation on a sprite that already exists, and the first argument to the function will be a handle to the sprite.

7.8 Sound

The Vectrex uses an AY-3-8912 Programmable Sound Generator (PSG) to produce its sounds. The Vectrex32 provides two ways to generate sounds: one way is limited to playing music while the other gives you full control of the AY-3-8912's functions.

7.8.1 Music

The PSG can play as many as three tones at a time, referred to as Channel A, Channel B and Channel C. You can create your own music or use the music built in to the Vectrex.

The PSG is given the next note to play every time a new frame starts, so the frame rate affects the tempo of the music. Thirty notes per second would be too fast, of course, so when Vectrex music is written, each note is given a duration in frames (e.g. a duration of 12 would cause a note to be played for 12 frames; if the frame rate is 30 per second, then that note would play for 12/30ths of a second or 0.4 seconds).

To play built in music, call the Play function with a constant that specifies the music:

```
call Play(PowerOn)
```

The built in music choices are Berzerk, MelodyMaster, Music1, Music2, Music3, Music4, Music5, Music6, Music7, Music8, PowerOn, Scramble, and SolarQuest.

To write your own music, first create an array with the musical notes. The array is two dimensional: each row has a note and a duration in frames. If you want two or three notes to be played simultaneously, combine them with the ABC function (so named because it combines channels A, B, and C). Here is an excerpt from the program Yankee.bas, which plays Yankee Doodle and which is included on the Vectrex32's drive:

```
yankee = { _
    {NA2, 12}, _
    {NG2, 12}, _
        :
        :
    {ABC(NA2, NA4, NA4 - NOctave), 12}, _
    {ABC(NG2, ND5, ND5 - NOctave), 12}, _
    {ABC(NA2, ND5, ND5 - NOctave), 12}, _
    {ABC(NG2, NE5, NE5 - NOctave), 12}, _
    {ABC(NA2, NFS5, NFS5 - NOctave), 12}, _
        :
        :
    {NA2, 12}, _
    {ABC(NG2, ND5, ND5 - NOctave), 12}, _
    {NA2, 12}}
```

NA2, ND5, NE5, etc. are built-in constants that specify the notes: they start with "N" for note, then specify the note (A through G), optionally specify an S for "sharp", and the octave (2 through 7). So NFS5 is the note F# in the 5th octave.

The predefined constant NOctave is the number of notes in an octave. Yankee Doodle has the third note frequently being the same as the second note, minus an octave; the second and third notes constitute a chord.

The second element in the rows, 12, is the number of frames to play the note.

Once the array with the musical notes is initialized, create a Music object and pass it to Play():

```
ydmusic = Music(yankee)
call Play(ydmusic)
```

To prematurely stop music that is playing, call Play(nil). To check whether the music has finished playing, use the function MusicIsPlaying(): it returns a one if music is playing and a zero if not.

7.8.2 Low-Level Sound Control

The AY-3-8912 PSG is controlled by writing values to its 14 registers (it actually has 16 registers, but two of them are used for something other than sound generation). The Vectrex32 allows you to directly write to those registers so that you can produce any music or sound effect the Vectrex is capable of. It's harder than playing music as described above, but it's more versatile.

The Sound subroutine takes a two dimensional array as an argument. Each row of the array has a register number (an integer from 0 and 13) and a value to write to that register (an integer from 0 to 255). After calling Sound, at the beginning of the next frame (e.g. when your program calls WaitForFrame) the register values are written to the PSG. This means that the frame rate affects the speed at which you can change sounds.

For example:

```
call Sound({{0, $fe}, {1, 0}, {7, $3e}, {8, 15}})
```

If you attempt to play music using the Play() subroutine at the same time you are using the Sound() subroutine to directly control the PSG, they will conflict with each other in unpredictable ways. However, calling Play(nil) will stop any sounds that are being generated by calls to Sound().

See section 12.31 for details about the Sound() function. The PSG's registers are described in Using the Sound Generator Chip.

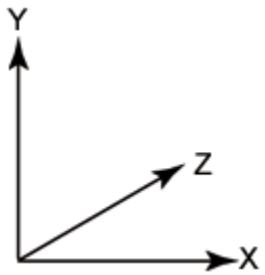
On the Vectrex32's drive, there is a program called Sounds.bas that demonstrates several different sound effects using low-level sound control.

7.9 3D Graphics

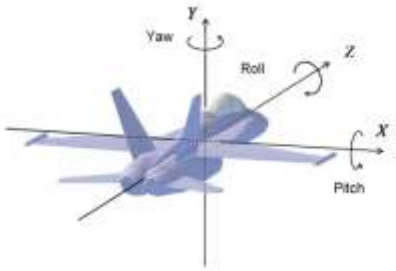
Vectrex32 version 1.10 introduced support for 3D graphics, as seen in games like Battlezone and Tail Gunner. You can create 3D sprites, place them at 3D locations in the "world", and move a virtual camera around the "world" to determine what is shown on the screen. Sprites can be rotated and the camera can be pointed in different directions to determine what the player sees. All the trigonometry and transforms required for this are done by the Vectrex32; you just specify the angles and positions.

7.9.1 The Coordinate System

The Vectrex32 defines 3D space with the origin (0, 0, 0) - i.e. X = 0, Y = 0, and Z = 0 - at the center of the screen. Positive X coordinates go to the right, positive Y coordinates go up, and positive Z coordinates go into the screen, away from the player:



Rotations are defined as Pitch, Roll, and Yaw. Pitch is rotation around the X axis, Roll is rotation around the Z axis, and Yaw is rotation around the Y axis:



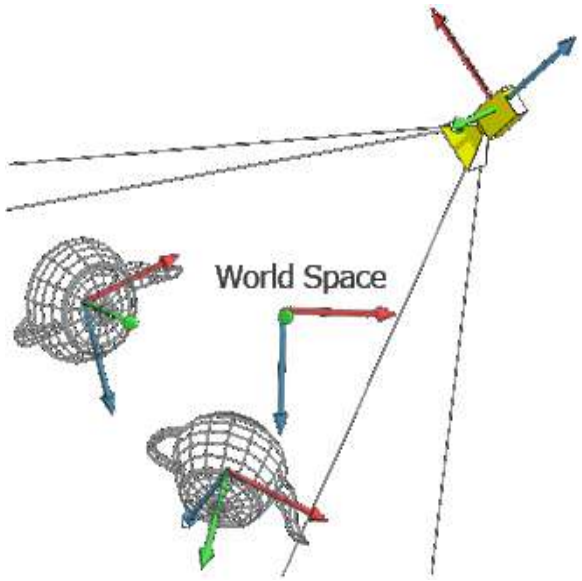
Positive angles always rotate counter-clockwise *when viewed from the positive end of the axis*. So if you're facing the Vectrex, you move to your right (positive X direction), and look back to the origin, a positive pitch will rotate the object counter-clockwise. If you soar above the Vectrex (positive Y direction) and look down, positive yaw will rotate counter-clockwise. It gets non-intuitive for roll because you need to stand behind the Vectrex (positive Z) and look towards the player to see that a positive roll is counter-clockwise.

7.9.2 Units

When writing a 2D game, our approach to units is haphazard: you pick the coordinates to use when defining the sprite, then juggle the magnification and Vectrex scale to make things look right on the screen. For 3D graphics, it's a good idea to take a more organized approach. The Vectrex32 draws 3D objects in perspective, so that objects will look smaller when they're farther away. So the units you use to define the object should be the same as the units you use to specify their position in the world. I suggest you define everything in meters, even if you're in the US, mainly because the camera's view is defined in meters. (However, you can change the camera's units and thus use different units for your world.)

7.9.3 From World to Screen

You define 3D objects, place them in your world, rotate them along their three axes, specify where the camera is and where it's pointing, and the Vectrex32 figures out how to draw that on the screen. Here's how:



- The Vectrex32 takes the 3D sprite and rotates it according to its pitch, roll, and yaw.
- It translates the 3D sprite to place it in its position in the world.
- If the camera is not at (0, 0, 0), it translates the sprite to make it relative to the camera's position (yes, the Vectrex32 moves the world instead of moving the camera).
- If the camera is not pointing straight down the Z axis (which is the direction a player in front of the Vectrex is facing), the Vectrex32 rotates the sprite's position in the world so that the camera's view is straight down the Z axis.
- The Vectrex32 projects the camera's view into a 2D image by applying perspective: objects look smaller when they're farther away.

- Up till now, the Vectrex32 has been working in "world" units, e.g. meters or feet or whatever you've chosen. The final step is for the Vectrex32 to scale that to Vectrex units (i.e. -128 through +127). Is does that by multiplying by a ratio you supply. That ratio must take the Vectrex Scale setting into account. (By default, the Vectrex32 assumes you've set a Vectrex scale of 64 and that 324 Vectrex units at scale 64 would draw a line on the screen that is 0.097 meters long.)

Note that this is all happening on a sprite-by-sprite basis. You're still building a drawing list, you're still putting the 3D sprites in the drawing list (there can be regular 2D sprites in the list as well), you're still setting the scale, and you're still resetting the pen to the origin periodically (in fact, you need to do so before each 3D sprite if you want the Vectrex32 to position it properly). But when the Vectrex32 sees a 3D sprite in the drawing list, it goes through all the steps above.

7.9.4 The Camera

In photography, a 50mm lens on an SLR is considered to produce a "normal" view, i.e. a field of view similar to what the human eye sees. A longer lens, like 300mm, gives a telephoto view while a shorter lens, like 28mm, gives a wide angle view. By default, the Vectrex32 sets its camera's focal length to 50mm. But you can change that with the CameraSetFocalLength subroutine and thereby show a zoomed in or zoomed out view of the 3D world.

The camera can be moved around the 3D world and rotated to point in any direction. It captures the view that will be drawn on the screen. There are important concepts related to moving the camera around and determining how wide a field of view the camera captures.

When a BASIC program starts, the camera is positioned at (0, 0, 0) and it is facing the same direction the player is facing: into the screen, straight down the Z axis. If you want the player to move forward (e.g. in a game of Battlezone, the player is moving his tank forward) you would change the camera's position by adding to its Z coordinate.

But what if the player has turned his tank 90˚ to the right, so that it's facing down the X axis, and then wants to go forward? If you add to the Z coordinate, the player will see the scene slide sideways; you need to add to the X coordinate instead. The direction gets even harder to determine if the player has rotated at an arbitrary angle, like 37˚; the forward direction is no longer along any of the axes.

The solution is to rotate a vector. A vector in geometry is drawn like an arrow: it has a length and a direction. If the player's tank is facing down the Z axis and you want him to move forward by one meter, a vector describing that would have a length of 1 meter and its direction would be along the Z axis; it would start at (0, 0, 0) and it would end at (0, 0, 1).

But if the player has rotated by 37˚, you would rotate the vector by 37˚. Then, it would start at (0, 0, 0) and end at (0.6018, 0, 0.7986). You can add those coordinates to the tank's location to advance it by 1 meter in the camera's direction.

The Vectrex32 has a function, VectorRotate, that does the necessary calculations for you.

7.9.5 Clipping

Two dimensional sprites offer the powerful feature of clipping, as described in section 7.6.2.4. Lines3D sprites support clipping too. The clipping is applied to the image the camera sees, so it

happens after the 3D world has been flattened to a 2D image with perspective. The clipping rectangle is still measured in Vectrex units, just as it is with Lines and Dots sprites.

8 WRITING PROGRAMS

Let's put all this information together and write a Vectrex program. First, we will write ScaleTest.bas, which is included on the Vectrex32 drive. It's a small program, but it demonstrates the important principles. After that, we will write a program that uses 3D graphics.

8.1 ScaleTest

ScaleTest is a tool for measuring lines and the screen at different Scale values. When you run it, it displays the instructions. After the user has read them, he can hit button 1 to start the "game". ScaleTest then displays a crosshair in the center of the screen and displays text showing the current scale and the crosshair's X and Y offset from the center (which, initially, is (0, 0)).

Buttons 1 and 2 increase and decrease the scale. When the scale changes, the crosshair is re-positioned at the origin. The joystick moves the crosshair. As the crosshair moves, a line is drawn between it and the center of the screen, while the text on the screen is updated with the crosshair's offset in the current scale.

Using this program, you can see how long a line is at a given scale. By dragging the crosshair to the edge of the screen, you can see how many units across the screen is, at a given scale.

The first thing we do in the program is check the version of the Vectrex32 firmware. When you write a program, you might use Vectrex32 features that didn't exist in previous versions. In the case of ScaleTest, it was written for version 1.00 (or, as the Version() function would return, 100), so if it is running on an older version, it advises the user to upgrade:

```
if Version() < 100 then
    print "Upgrade the Vectrex32 firmware to version 1.00 or
newer."
    stop
endif
```

Next we display the instructions:

```
call IntensitySprite(72)

textSize = {40, 5}
call TextSizeSprite(textSize)
instructions = {{-50, 90, "INSTRUCTIONS"}, _
                {-80, 70, "BTN 1&2 CHANGES SCALE"}, _
                {-80, 50, "JOYSTK MOVES CROSSHAIR"}, _
                {-80, 30, "READ COORDINATES"}, _
                {-80, 0, "PRESS BTN 1 TO START"}}

call TextListSprite(instructions)

' Wait for button 1
controls = WaitForFrame(JoystickNone, Controller1, JoystickNone)
while controls[1, 3] = 0
    controls = WaitForFrame(JoystickNone, Controller1, _
                            JoystickNone)
endwhile
```

The call to IntensitySprite puts a pseudo-sprite on the drawing list. The pseudo-sprite sets the brightness of whatever is drawn afterwards.

We create a *textSize* array and pass it to the TextSizeSprite() function. This puts a pseudo-sprite on the drawing list which specifies how big text should be drawn on the screen. The first number, 40, specifies how wide each letter should be while the second number, 5, specifies the height. Surprisingly, this does not mean that the letters are 8 times wider than they are tall; the Vectrex interprets these numbers in its own inscrutable way.

We create an array, *instructions*, with five lines of text. Each row of the array has an X coordinate, a Y coordinate, and the text. We pass the array to TextListSprite, which adds a Text List sprite to the drawing list.

Next, we'll enter a loop waiting for the user to press button 1. We call WaitForFrame(JoystickNone, Controller1, JoystickNone). The first argument means that we do not want to check the status of the joystick. The second argument means we are only interested in the status of controller 1 (not controller 2). The third argument specifies whether we want the joystick's X position, Y position, both, or neither; we want neither.

WaitForFrame waits until it is time for the next frame to be drawn, sends code to the Vectrex to read the controller(s) and draw the contents of the drawing list. When WaitForFrame gets the requested controller status, it returns that status in an array which we assign to the controls variable.

The returned array has the dimensions [2, 6]. Row 1 has the status of controller 1, row 2 has the status of controller 2. In each row, the six elements are: the joystick's X position, the joystick's Y position, button 1's status (1 if it's pressed, 0 if it isn't), button 2's status, button 3's status, and button 4's status. Only the elements you requested will be filled in; e.g. if you did not check the status of the joystick, the first two elements of the row will not be initialized, and if you did not check the status of controller 2, the second row will not be initialized.

We loop while button 1 is unpressed. In each iteration of the loop, we call WaitForFrame which draws the next frame and gets the latest controller status.

If you run ScaleTest, this first loop would be a good time to hit Ctrl+C to enter Debug mode and do some experimenting. Modify the elements of the *textSize* array and the *instructions* array. The Text Size pseudo-sprite and the TextList sprite use live data, so your changes will appear on the screen. When you're done experimenting, you can give the run command to resume running ScaleTest.

The loop exits when the user presses button 1 on controller 1. We don't need the instructions anymore, so we discard everything in the drawing list:

```
call ClearScreen()
```

Next, we start building a new drawing list for the rest of the program.

```
call IntensitySprite(72)

info = {{-35,90,"SCALE: 50"},{-35,70,"X = 0"},{-35,50,"Y = 0"}}
infoDisplay = TextListSprite(info)

call ReturnToOriginSprite()

' This is the scale that the user will change
scale = 50
scaleDisplay = ScaleSprite(scale)
```

As we did earlier, we put an Intensity pseudo-sprite on the list. We then add a TextList sprite that will show the current scale setting and the X and Y coordinates of the crosshair.

We have no idea where the pen will be positioned when it's done drawing the TextList sprite, so we add a ReturnToOriginSprite to the drawing list. This will position the pen in the center of the screen.

We'll start with a scale of 50, so we set the *scale* variable to 50 and call ScaleSprite to add a Scale pseudo-sprite to the drawing list. The Scale sprite sets the Vectrex to draw at a scale of 50.

```
' Put a dummy move in. For some reason, drawing at (0, 0) after
' a ReturnToOrigin doesn't work
call MoveSprite(1, 1)
call MoveSprite(-1, -1)

' Set the position of the crosshair the user will move around
xy = {{0, 0}}
crossPos = LinesSprite(xy)
```

Something weird happens on the Vectrex if you draw at (0, 0) right after doing a ReturnToOrigin; things don't get drawn correctly. Christopher Salomon says in his tutorial that it has something to do with timing, and that it can be fixed by doing a Move. So we add two Move pseudo-sprites to move the pen by (1, 1), then (-1, -1) (remember, these are relative movements, so we're moving away from the origin and then back to it).

We then create a Lines sprite that draws a line to (0, 0). We're already at (0, 0) so it just draws a dot, but when the user moves the crosshair, we will change the coordinates in *xy* so that a line will be drawn to where the crosshair is positioned.

```
crossScale = ScaleSprite(20)

crosshair = {{MoveTo, 0, 50}, {DrawTo, 0, -50}, _
             {MoveTo, -50, 0}, {DrawTo, 50, 0}}
crossDisplay = LinesSprite(crosshair)
```

Next, we draw the crosshair. The point of ScaleTest is to allow the user to change the scale and see the effects, but we don't want the crosshair's size to change when the user changes the scale. So we put a new Scale pseudo-sprite on the drawing list to set the scale to 20. This Scale pseudo-sprite will never change.

Next we initialize the variable *crosshair* as an array with four rows and use it to create a Lines sprite. Notice that each row in the *xy* array had two elements, an X and Y coordinate, while each row in the *crosshair* array has three elements. The Lines sprite can accept either format. If there are three elements in the row, they are Mode (MoveTo or DrawTo), X and Y. It's common in a Lines sprite for the first row to have a mode of MoveTo. That's because you usually want the sprite to be centered at the current pen position, but if there's no line at the center of the sprite, you need to do a MoveTo to move the pen to where it should start drawing.

So the *crosshair* array will move the pen up to (0, 50), draw down to (0, -50), move to (-50, 0), and finally draw to (50, 0).

```
' If a button is pressed, we wait until it's released before
accepting
' another button press
refractoryPeriod = 1
```

We initialize *refractoryPeriod*, which is a flag indicating that we should ignore button presses. The Vectrex32 checks the buttons every time we call WaitForFrame, e.g. 30 times per second. When the user presses a button, it will remain pressed for at least $^{1}/_{10}$ of a second – probably longer. We don't want to register multiple button presses, so when a button press happens, we will ignore more presses until we see that the button has been released. The user already pressed button 1 to get rid of the instructions and the button is probably still being pressed, so we turn the refractory period on.

```
' Do forever
while 1
   :
   :
endwhile
```

The rest of the program happens inside an infinite loop.

```
    ' Wait until it's time to prepare the next frame
    controls = WaitForFrame(JoystickDigital, Controller1, _
                            JoystickX + JoystickY)
```

The first thing we do in the infinite loop is call WaitForFrame. The first argument means we want to read the joystick in Digital mode (as opposed to Analog); we don't care how far it is pushed in any direction, we only want to know that it *has* been pushed.

The second argument means we only want to read controller 1.

The third argument specifies that we want to read joystick movements in both the X and Y directions.

WaitForFrame returns an array with the status of the controller. We put it in the *controls* variable.

```
    if controls[1, 1] < 0 then
        xy[1, 1] = xy[1, 1] - 1
    elseif controls[1, 1] > 0
        xy[1, 1] = xy[1, 1] + 1
    endif
    if controls[1, 2] < 0 then
        xy[1, 2] = xy[1, 2] - 1
    elseif controls[1, 2] > 0
        xy[1, 2] = xy[1, 2] + 1
    endif
```

controls[1, 1] tells us the X movement of controller 1's joystick and *controls[1, 2]* tells us the Y movement. Recall that the array *xy* is live data for the Lines sprite, so by changing the values in xy, we'll change the endpoint of that line: if the joystick is to the left, we subtract 1 from *xy[1, 1]*; if the joystick is to the right, we add 1 to *xy[1, 1]*. Similarly, we change *xy[1, 2]* based on the joystick's up-down position.

The crosshair's sprite is on the drawing list after the *xy* Lines sprite. So we're changing the line drawn by *xy*, the pen ends up at the end of the *xy* line, and that's where the crosshair gets drawn.

```
    if controls[1, 3] or controls[1, 4] then
        if refractoryPeriod = 0 then
            xy[1, 1] = 0
            xy[1, 2] = 0
            if controls[1, 3] then
                scale = max(scale - 1, 0)
            else
                scale = min(scale + 1, 255)
            endif
            call SpriteScale(scaleDisplay, scale)
            info[1, 3] = "SCALE: " + scale
            refractoryPeriod = 1
        endif
    else
        refractoryPeriod = 0
    endif
```

Next we check *controls[1, 3]* and *controls[1, 4]*. These tell us the state of buttons 1 and 2. If they are pressed, and if we are not in the refractory period (*refractoryPeriod* = 0), we reset the *xy* line to (0, 0) and increment or decrement *scale*. Scale pseudo-sprites do not use live data, so we need to call SpriteScale to change the scale that we draw the *xy* line at. We then update *info[1, 3]* (the live data for the TextList sprite) with the new scale and turn on the refractory period.

However, if neither button 1 nor 2 are pressed, we turn the refractory period off (*refractoryPeriod* = 0).

```
    info[2, 3] = "X = " + xy[1, 1]
    info[3, 3] = "Y = " + xy[1, 2]
```

Finally, we update the live data for the TextList with the new X and Y coordinates. That concludes the code in the infinite loop and completes the ScaleTest program.

8.2 Demo3D

Demo3D.bas, which is included on the Vectrex32's drive, demonstrates 3D graphics on the Vectrex32. It creates a 3D sprite - a pyramid - allows you to rotate it, move the camera around, and create more pyramids.

The first part of the program displays instructions and waits for the user to hit a button. It's similar to the beginning of ScaleTest, so there's no need to review it. (In fact, this description of Demo3D will focus on the 3D-specific code and skip over much of the program's logic.)

By default, the camera in the 3D world is positioned at (0, 0, 0). We want to be able to move it around, so we create an array and use it as the translation coordinates for the camera:

```
cameraTrans = {0, 0, 0}
call CameraTranslate(cameraTrans)
```

The camera, like sprites, uses live data for its translation. We'll be able to move the camera just by changing the values in the cameraTrans array.

With 2D graphics, you set the Vectrex scale to whatever you want and that's all the Vectrex32 needs to know. In 3D, the Vectrex32 will be converting world units (meters, feet, or whatever you choose) to Vectrex units. In order to do that, it needs to know the ratio of world units to Vectrex screen units. So when we create a Scale sprite to set the scale, we also set the ratio:

```
call ScaleSprite(64, 324 / 0.097)
```

In Demo3D, we're using meters as our world coordinates. At a Vectrex scale of 64, a line that is 324 units long will draw a line on the screen that is 0.097 meters long (about 3.8 inches). We determined this by running ScaleTest, setting a scale of 64, drawing a line straight up, and measuring its length with a ruler.

Next, we define an array with the lines that draw a pyramid. Notice that, unlike the *crosshair* array in ScaleTest, the *pyramidLines* array has X, Y, and Z coordinates:

```
pyramidWidth = 2
pyramidHeight = 2
pyramidLines = { _
    {MoveTo, 0, pyramidHeight, 0}, _
    {DrawTo, pyramidWidth / 2, 0, 0}, _
    {DrawTo, 0, 0, pyramidWidth / 2}, _
    {DrawTo, -pyramidWidth / 2, 0, 0}, _
    {DrawTo, 0, 0, -pyramidWidth / 2}, _
    {DrawTo, pyramidWidth / 2, 0, 0}, _
    {MoveTo, 0, pyramidHeight, 0}, _
    {DrawTo, -pyramidWidth / 2, 0, 0}, _
    {MoveTo, 0, pyramidHeight, 0}, _
    {DrawTo, 0, 0, -pyramidWidth / 2}, _
    {MoveTo, 0, pyramidHeight, 0}, _
    {DrawTo, 0, 0, pyramidWidth / 2}}
```

Demo3D calls a function, NewPyramid, to create a pyramid sprite. The program allows the user to create multiple pyramids, which we'll keep track of in an array, so the NewPyramid function will return an array of one element so that later, we can easily append arrays.

For now, let's look at NewPyramid:

```
function NewPyramid()
    dim pyramid[1]
    pyramid[1] = Lines3DSprite(pyramidLines)
```

It starts off creating a Lines3D sprite and saving its handle in the array. Lines3DSprite also adds the sprite to the drawing list.

We want to randomly position the new pyramid in front of the camera. We start by randomly generating X, Y, and Z coordinates. The Z coordinate is between 50 and 150 meters from the camera. The X and Y coordinates are in a small 150mmx180mm area - the size of the Vectrex screen. We'll change that in a moment:

```
z = rand() mod 100 + 50
x = (rand() mod (150 - pyramidWidth) - 75) / 1000.0
y = (rand() mod (180 - pyramidHeight) - 90) / 1000.0
```

We have X and Y coordinates on the screen for the pyramid to appear, but we need to place that in the 3D world at a distance of *z* from the camera. Assuming the user is sitting 0.5 meters from the screen, and we want the screen to appear as a window that he's looking through, we figure out the world coordinates based on the following formula:

$$world\ coordinate = screen\ coordinate \times \frac{(distance\ to\ object\ +\ user's\ distance\ from\ screen)}{user's\ distance\ from\ screen}$$

or in code:

```
x = x / 0.5 * (z + 0.5)
y = y / 0.5 * (z + 0.5)
```

So we've conjured up X, Y, and Z coordinates, but if camera is not facing directly down the Z axis, the pyramid may not be in the camera's field of view. We need to rotate the coordinates the

same way the camera is rotated. So we get the camera's pitch, roll, and yaw, make a vector with the X, Y, and Z coordinates, and rotate the vector:

```
cameraAngles = CameraGetRotation()
vector = VectorRotate({x, y, z}, cameraAngles[1], _
   cameraAngles[2], cameraAngles[3])
```

The camera also might not be at (0, 0, 0). So again, to ensure that the pyramid appears in front of the camera, we have to offset its position the same way the camera is offset from (0, 0, 0). Fortunately, we earlier set the array cameraTrans to be the camera's location:

```
vector[1] = vector[1] + cameraTrans[1]
vector[2] = vector[2] + cameraTrans[2]
vector[3] = vector[3] + cameraTrans[3]
```

The NewPyramid function finishes up by setting the pyramid's position and random angles of rotation:

```
    call SpriteTranslate(pyramid[1], vector)
    call SpriteRotate(pyramid[1], rand() mod 360, _
        rand() mod 360, rand() mod 360)
    return pyramid
endfunction
```

After NewPyramid returns, Demo3D enters an infinite loop, waits for the frame and gets the controller's status, and has a series of IF statements deciding what to do based on button presses and joystick movements. If button 1 is pressed, it creates a new pyramid. Notably, it first puts a ReturnToOrigin sprite into the drawing list. It's important for every 3D sprite to be preceded by a ReturnToOrigin since the Vectrex32 is figuring out where to draw it on the screen:

```
call ReturnToOriginSprite()
pyramids = AppendArrays(pyramids, NewPyramid())
```

Combinations of button presses and joystick movements cause all the pyramids to rotate counter-clockwise or clockwise in pitch, roll, or yaw. In those cases, Demo3D loops through its list of pyramids and calls subroutines like:

```
' Pitch counter-clockwise
call SpriteRotate(pyramids[i], 1, 0, 0)
' Roll clockwise
call SpriteRotate(pyramids[i], 0, -1, 0)
' Yaw counter-clockwise
call SpriteRotate(pyramids[i], 0, 0, 1)
```

The user can pitch, roll, and yaw the camera too. Some examples:

```
call CameraRotate(-1, 0, 0)
call CameraRotate(0, 1, 0)
call CameraRotate(0, 0, -1)
```

Moving the camera requires the same trick with vector rotation that we did when positioning a new pyramid. We have the cameraTrans array that specifies the X, Y, and Z coordinates of the

camera. And if the camera is pointing straight down the Z axis, we could easily move it forward by incrementing its Z coordinate. But if the camera is rotated in an arbitrary direction and we want it to move forward (e.g. the player's character has turned and then wants to step forward) we need to figure out what X, Y, and Z increments cause the camera to move in the same direction it is pointing. So we get the camera's rotation, create a vector that's pointing down the Z axis, rotate the vector, and then translate the camera with the result:

```
camRotation = CameraGetRotation()
v = {0, 0, 0}
if controls[1, 2] < 0 then
    v = {0, 0, -1}
elseif controls[1, 2] > 0
    v = {0, 0, 1}
endif
v = VectorRotate(v, camRotation[1], camRotation[2], _
    camRotation[3])
cameraTrans[1] = cameraTrans[1] + v[1]
cameraTrans[2] = cameraTrans[2] + v[2]
cameraTrans[3] = cameraTrans[3] + v[3]
```

9 TIPS

These tips should help you when writing programs on the Vectrex32.

9.1 Workflow

When writing a game, you need to edit the code, test it, edit it again, test it again, etc. My strategy is to open the BASIC program in a text editor that displays line numbers - I use Notepad++ on Windows - make the changes I want, save it (but stay in the editor), then switch to the terminal window, reload the program, and run.

Be sure to read the sections about Debugging, Breakpoints and Tracing in the Galactic Studios BASIC manual. There are powerful tools available for you to troubleshoot your program.

9.2 The Vectrex Reset Button

The reset button on the Vectrex resets the 6809 but does not reset the Vectrex32. The only way to reset the Vectrex32 is to turn the Vectrex off and on.

9.3 Intensity vs. Disabled Sprites

You will occasionally want to remove objects from the display (e.g. when a ship blows up). There are three ways of doing this. First, you can call RemoveSpriite(<sprite>) to take the sprite off the drawing list. Second, you can call SpriteEnable(<sprite>, false) to tell the Vectrex32 to stop drawing it. Third, before you add the sprite to the list, you can add an Intensity pseudo-sprite. Then, when you want the sprite to disappear, you set the Intensity spite to have an intensity of zero.

There are important differences between these three methods. If you remove or disable the sprite, then the sprite is not drawn. That saves time and it means that the pen does not move. But

if you merely set the intensity to zero, the sprite is still drawn, the pen is moved, but you just can't see the lines or dots.

If there is anything in the drawing list that is after the sprite (but before a ReturnToOrigin pseudo-sprite), its position will be affected by whether you have disabled the sprite or just set the intensity to zero.

9.4 Time

You can use the frame rate to measure time. Get the current frame rate by calling the function GetFrameRate(). Then, increment a counter every time you call WaitForFrame(...). When the counter is equal to the frame rate, one second has passed; reset the counter to zero and start counting for the next second.

If you choose a "bad" frame rate, your timing will not be accurate. The Vectrex32 operating system works on a time base of 960 cycles per second. If your frame rate is 30 times per second, then the Vectrex32 refreshes the screen when it counts 32 cycles (960 / 30). If your frame rate is 60 times per second, the Vectrex32 refreshes every 16 cycles. Thirty two and 16 are whole numbers, so your clock will be accurate.

But if you choose a frame rate of 75, the Vectrex32 will refresh the screen every 12 cycles (960 / 75 = 12.8, which gets truncated to 12 cycles), or 80 times per second. Your counting code will assume that every 75 frames is one second when in fact every 80 frames is one second, so your timing will run fast.

If timing is critical, choose a frame rate that evenly divides 960, such as 30, 40, 60, 80, or 120.

9.5 Making Complex Shapes

The Vectrex32 provides some built-in functions that are useful for creating complex Lines sprites. For a good example, look in lunar.bas, which is included on the Vectrex32's drive, and see how the lunar module sprite is created:

- The top part of the LEM is created in a function called AscentModule(). It uses the built-in function RegularPolygon() to generate an array of local coordinates for an octagon.
- The bottom part of the LM is created in a function called DescentModule(). It returns an array of local coordinates for a rectangle.
- The landing gear is created in a function called Legs().
- The function LEM() calls AscentModule(). The octagon returned by AscentModule() has its center at (0, 0). LEM() then calls the built-in function Offset() to shift the octagon up so that its base line has a Y coordinate of 0. Then it calls DescentModule(), calls Offset() again to shift the descent module's rectangle downso that it is beneath the Y = 0 line. After getting the remaining pieces of the LEM (the legs, the exhaust bell, and different options for exhaust flames) it combines all these arrays into one with the built-in function AppendArrays(). The result is passed to LinesSprite to create a lunar module sprite.

9.6 Mixing 2D and 3D

A game can include both 2D sprites and 3D sprites. For example, in the game Battlezone, there are mountains in the background. No matter how long the player drives his tank towards them, he will never approach them. If you were writing a Battlezone game, you would draw the mountains as 2D sprites while the enemy tanks would be 3D sprites. Just as you should use a

ReturnToOrigin sprite before drawing each Lines3D sprite, you should have a ReturnToOrigin sprite when you switch from 3D sprites to 2D sprites.

10 KNOWN PROBLEMS

- Sometimes, when you run your terminal program and connect to the Vectrex32, the Vectrex32 will not immediately display the Ready prompt. If this happens, just hit Enter.
- Do not try to open a terminal to the Vectrex32 while you are copying a file to the Vectrex32's drive; the terminal is likely to crash and you will need to turn the Vectrex off and on. (It is OK to open a terminal window first, then copy a file, just don't open a new window while a copy is being done.)
- The Vectrex32's drive is slow. Be patient when copying files to it.
- Single character Text sprites are not displayed correctly. If you want to display a single character, add a space to it.

11 Built-In Constants

These are the constants built into the Vectrex32. They add to the constants built in to GSBASIC itself, which are documented in the Galactic Studios BASIC manual.

Built-in constant names are case-insensitive.

11.1 Controller Constants

Description

These constants define the possible arguments to the WaitForFrame function.

More Information

The defined names are:

Controller1	JoystickAnalog	JoystickX
Controller2	JoystickDigital	JoystickY
ControllerNone	JoystickNone	

See the description of the WaitForFrame function for details about how to use these contants.

11.2 DrawTo

Description

Defined as the integer 1. This constant is useful when creating Lines sprites.

Example

```
triangleCoordinates = {
   {MoveTo, 10, 5}, _
   {DrawTo, -7, -3}, _
   {DrawTo, -5, 8}, _
   {DrawTo, 10, 5}}
triangleSprite = LinesSprite(triangleCoordinates)
```

See Also

MoveTo

11.3 MoveTo

Description

Defined as the integer 0. This constant is useful when creating Lines sprites.

Example

```
triangleCoordinates = { _
   {MoveTo, 10, 5}, _
   {DrawTo, -7, -3}, _
   {DrawTo, -5, 8}, _
   {DrawTo, 10, 5}}
triangleSprite = LinesSprite(triangleCoordinates)
```

See Also

DrawTo

11.4 Music Constants

Description

These define the pieces of music built into the Vectrex.

More Information

The defined names are:

Berzerk	Music4	PowerOn
MelodyMaster	Music5	Scramble
Music1	Music6	SolarQuest
Music2	Music7	
Music3	Music8	

See the section Sound for details about how to use these constants.

11.5 Musical Notes

Description

These define the musical notes from G2 (note G, 2nd octave) up to AS7 (A sharp, 7th octave).

More Information

The defined names are:

NG2	NG4	NG6
NGS2	NGS4	NGS6
NA2	NA4	NA6
NAS2	NAS4	NAS6
NB2	NB4	NB6
NC3	NC5	NC7
NCS3	NCS5	NCS7
ND3	ND5	ND7
NDS3	NDS5	NDS7
NE3	NE5	NE7
NF3	NF5	NF7
NFS3	NFS5	NFS7
NG3	NG5	NG7
NGS3	NGS5	NGS7
NA3	NA5	NA7
NAS3	NAS5	NAS7
NB3	NB5	
NC4	NC6	
NCS4	NCS6	
ND4	ND6	
NDS4	NDS6	
NE4	NE6	
NF4	NF6	
NFS4	NFS6	

See the section Sound for details about how to use these constants.

11.6 NOctave

Description

This constant defines the difference between the same notes in different octaves (e.g G2 and G3)

More Information

See the section Sound for details about how to use NOctave.

12 Built-In Functions and Subroutines

These are the functions and subroutines built into the Vectrex32. They add to the functions built in to GSBASIC itself, which are documented in the Galactic Studios BASIC manual.

Built-in function and subroutine names are case-insensitive.

12.1 ABC

Description

Combines one, two, or three musical notes into a single entry for a music array.

Example

```
yankee = { _
         :
         :
    {ABC(NA2, NA4, NA4 - NOctave), 12}, _
         :
         :
    {NA2, 12}}
```

More Information

The Vectrex can play notes on up to three channels (channels A, B and C) at a time. To indicate that multiple notes should be played simultaneously, combine them into a single value with the ABC function. ABC can be called with 1, 2, or 3 arguments.

See Also

Sound

12.2 CameraGetFocalLength

Retrieves the focal length in millimeters that the virtual camera is using.

Example

```
focalLength = CameraGetFocalLength()
```

More Information

In photography, a 50mm lens on an SLR is considered to produce a "normal" view, i.e. a field of view similar to what the human eye sees. A longer lens, like 300mm, gives a telephoto view while

a shorter lens, like 28mm, gives a wide angle view. By default, the Vectrex32 sets its camera's focal length to 50mm, so that CameraGetFocalLength would return 50.

See Also
CameraSetFocalLength, The Camera

12.3 CameraGetRotation
Retrieves the current pitch, roll, and yaw of the camera.

Example
```
' Prints an array with three elements, the pitch, roll , and yaw
print CameraGetRotation()
```

More Information
Returns the camera's current pitch, roll, and yaw, in degrees, in a 1x3 array.

See Also
CameraRotate, CameraSetRotation, The Camera

12.4 CameraRotate
Rotates the camera by the specified pitch, roll, and yaw.

```
call CameraRotate(pitch, roll, yaw)
```

More Information
Rotates the camera by the specified angles, which are in degrees. If the camera is already rotated, the pitch, roll, and yaw angles are added to its current rotation.

See Also
CameraGetRotation, CameraSetRotation, The Camera

12.5 CameraSetFocalLength
Sets the focal length of the camera in millimeters.

Example
```
call CameraSetFocalLength(150)
```

More Information
In photography, a 50mm lens on an SLR is considered to produce a "normal" view, i.e. a field of view similar to what the human eye sees. A longer lens, like 300mm, gives a telephoto view while a shorter lens, like 28mm, gives a wide angle view. By default, the Vectrex32 sets its camera's focal length to 50mm, but you can change that and thus zoom in or zoom out from a scene using the CameraSetFocalLength subroutine.

See Also
CameraGetRotation, The Camera

12.6 CameraSetRotation
Rotates the camera to the specified pitch, roll, and yaw.

```
call CameraSetRotation(pitch, roll, yaw)
```

More Information

Rotates the camera to the specified angles, which are in degrees. If the camera is already rotated, the pitch, roll, and yaw angles replace its current rotation.

See Also

CameraGetRotation, CameraRotate, The Camera

12.7 CameraTranslate

Description

Sets the translation of the camera.

Example

```
translation = {10, 15, 0}
CALL CameraTranslation(translation)
```

More Information

Sets the camera's offset in the world from the origin (0, 0, 0). The array is live data, so the translation can be changed by modifying the values in the array.

See Also

The Camera

12.8 ClearScreen

Description

Removes all sprites and pseudo-sprites from the drawing list.

Example

```
call ClearScreen
```

See Also

The Drawing List

12.9 Distance

Description

Returns the distance between two points or between a point and a line segment.

Example

```
point1 = {10, 10}
point2 = {20, 20}
line = {{1, 1}, {100, 3}
PRINT Distance(point1, point2) ' prints 14.14213
PRINT Distance(point1, line) ' prints 8.81638
PRINT Distance(line, point1) ' prints 8.81638

' Also works with 3D points and lines
point3 = {10, 10, 10}
line3 = {{1, 1, 1}, {100, 3, 5}}
PRINT Distance(point3, line3)
```

More Information

A point argument is a 1 dimensional array with two or three elements, x, y, and optionally z. A line segment argument is a 2 dimensional array where each row has two or three elements, x, y, and optionally z, which are the endpoints of the line. The distance between two points is calculated using the Pythagorean Theorem; the distance between a point and a line segment is the distance between the point and the closest location on the line segment.

See Also

PtInRect

12.10 DotsSprite

Description

Creates a sprite that will draw dots on the screen.

Example

```
dots = {{10, 10}, {20, 20}}
sprite = DotsSprite(dots)
```

More Information

The argument is an array of (x, y) coordinates. The Dots sprite is created, added to the end of the drawing list, and its handle is returned.

The coordinates are relative to the position of the pen at the time the Dots sprite begins drawing, e.g. if the pen starts at (5, 5), then the dots in the example will be drawn at (15, 15) and (25, 25). This is different from traditional Vectrex programming where the first dot would be drawn relative to the pen position, and then the second dot would be drawn relative to the first dot.

After drawing a Dots sprite, the pen is automatically reset to the center of the screen.

See Also

The Drawing List, Local Coordinates, Advanced Sprite Features

12.11 DumpSprite

Description

Prints the details of a single sprite.

Example

```
sprite = DotsSprite(dots)
call DumpSprite(sprite)
```

More Information

This subroutine may be useful when debugging.

See Also

DumpSprites

12.12 DumpSprites

Description

Prints a list of all the sprites and pseudo-sprites on the drawing list.

Example

```
call DumpSprites
```

More Information

This subroutine may be useful when debugging.

See Also

DumpSprite

12.13 GetFrameRate

Description

Returns the current frame rate.

Example

```
rate = GetFrameRate()
```

More Information

The value returned is the frames per second, e.g. 30.

See Also

Frames, SetFrameRate

12.14 IntensitySprite

Description

Creates a pseudo-sprite that sets the intensity at which subsequent sprites will be drawn.

Example

```
intensity = IntensitySprite(60)
```

More Information

The intensity is the brightness that sprites are drawn at; larger numbers are brighter. The intensity value can range from 0 to 127.

See Also

SpriteIntensity

12.15 Lines3DSprite

Creates a sprite that will draw a 3D object on the screen.

Example

```
pyramidLines = { _
    {MoveTo, 0, 2, 0}, _
    {DrawTo, 1, 0, 0}, _
    {DrawTo, 0, 0, 1}, _
    {DrawTo, -1, 0, 0}, _
    {DrawTo, 0, 0, -1}, _
    {DrawTo, 1, 0, 0}, _
    {MoveTo, 0, 2, 0}, _
    {DrawTo, -1, 0, 0}, _
    {MoveTo, 0, 2, 0}, _
    {DrawTo, 0, 0, -1}, _
    {MoveTo, 0, 2, 0}, _
    {DrawTo, 0, 0, 1}}
pyramidSprite = Lines3DSprite(pyramidLines)
```

More Information

The argument is an array of (mode, x, y, z) rows. The mode values can be MoveTo (which equals 0) or DrawTo (1). The Lines3D sprite is created, added to the end of the drawing list, and its handle is returned.

The coordinates are relative to the position of the sprite in 3D space (as determined by the sprite's translation).

See Also

3D Graphics

12.16 LinesSprite

Creates a sprite that will draw lines on the screen.

Example

```
lines = {{10, 10}, {20, 20}}
sprite1 = LinesSprite(lines)
modeLines = {{MoveTo, 10, 10}, {DrawTo, 20, 20}, _
             {MoveTo, 30, 30}, {DrawTo, 40, 40}}
sprite2 = LinesSprite(modeLines)
```

More Information

The argument can be an array of (x, y) coordinates or an array of (mode, x, y) rows. The Lines sprite is created, added to the end of the drawing list, and its handle is returned.

If the array includes mode values, they can be MoveTo (which equals 0) or DrawTo (1).

The coordinates are relative to the position of the pen at the time the Lines sprite begins drawing, e.g. if the pen starts at (5, 5), then the lines in the first part of the example will be drawn to (15, 15) and (25, 25). This is different from traditional Vectrex programming where the first line would be drawn relative to the pen position, and then the second line would be drawn relative to the first line.

See Also

The Drawing List, Local Coordinates, Advanced Sprite Features

12.17 MoveSprite

Description

Creates a pseudo-spite that moves the pen.

Example

```
move = MoveSprite(10, 20)
```

More Information

The Move pseudo-sprite is created, added to the end of the drawing list, and its handle is returned.

See Also

SpriteMove

12.18 Music

Description

Creates a Music object that can be played.

Example

```
yankee = { _
    {NA2, 12}, _
    {NG2, 12}, _
        :
        :
    {NA2, 12}}
ydmusic = Music(yankee)
CALL Play(ydmusic)
```

More Information

Takes an array of notes as an argument and returns the handle to a Music object. That handle can be passed to the Play subroutine.

See Also

Play, Sound, Sound subroutine

12.19 MusicIsPlaying

Description

Returns a 1 if the Vectrex is currently playing music; otherwise returns 0.

Example

```
IF not IsMusicPlaying() THEN
      Play(nil) ' Stop the music
ENDIF
```

More Information

When you call the Play subroutine, it returns immediately and the music starts playing. This function can tell you when the music has finished.

See Also

Play, Sound

12.20 Offset

Description

Adds the specified X, Y, and optionally Z values to each coordinate in an array.

Example

```
lines = {{10, 10}, {20, 20}}
CALL Offset(lines, 5, 6) ' lines will be {{15, 16}, {25, 26}}
modeLines = {{MoveTo, 10, 10}, {DrawTo, 20, 20}, _
             {MoveTo, 30, 30}, {DrawTo, 40, 40}}
CALL Offset(modeLines, 11, 12) ' X and Y coords will be changed

lines3d = {{MoveTo, 0, 2, 0},{DrawTo, 1, 0, 0},{DrawTo, 0, 0, 1}}
CALL Offset(lines3d, 5,6, 7) ' X, Y, and Z coords will be changed
```

More Information

The Offset subroutine can handle arrays with (x, y) coordinates, (mode, x, y), or (mode, x, y, z) so it can be used on the arrays that DotsSprite, LinesSprite, and Lines3DSprite use.

12.21 Peek

Description

Retrieves values from the Vectrex's memory.

Example

```
DIM peekdata[2]
' Send a message to the Vectrex to peek at memory
CALL Peek($f000, 16, peekdata)
CALL WaitForFrame(JoystickNone, ControllerNone, JoystickNone)
' Wait for the array to be filled with the dumped memory
WHILE peekdata[1] = 0
ENDWHILE

data = peekdata[2]
FOR i = 1 TO UBound(data)
   PRINT data[i]
NEXT i
```

More Information

The Vectrex32 does not have direct access to the Vectrex's memory, so a multi-step procedure is necessary for the Vectrex32 to read the Vectrex's memory: tell the Vectrex to copy some memory into dual-port memory, wait until the Vectrex tells us that the data has been copied, and then copy it into a BASIC array.

To use Peek, create an array with two elements (e.g. *peekdata*). Call Peek, passing in the first Vectrex memory address you want to fetch, the number of bytes you want (must be 16 or fewer), and the array. Peek() will initialize the first element of the array to 0, indicating that the data is not ready.

If you have done this in Immediate mode, the Vectrex32 will automatically be refreshing frames, but if this is in a BASIC program, you must call WaitForFrame to send the command to the Vectrex. When the data is ready for you to read, the first element in your array (e.g. *peekdata[1]*) will be set to 1. If you're doing this in Immediate mode, it will happen so fast that you can just assume the data is ready, but if it's in a program, you should wait (e.g. in a loop) until the element has been set to 1. The data will be put in the second element of the array, and it will be an array

of integers (e.g. *peekdata[2]* will be an array, which is different from *peekdata* being a two-dimensional array).

In Immediate mode, you can just print the array; in a program you can examine the returned data.

Vectrex32 comes with a sample program, DumpMemory.bas, on its drive. It demonstrates using Peek to dump large blocks of the Vectrex's memory.

12.22 Play

Description

Plays sounds.

Example

```
CALL Play(MelodyMaster)
CALL Play(nil)
yankee = { _
    {NA2, 12}, _
    {NG2, 12}, _
        :
        :
    {NA2, 12}}
ydmusic = Music(yankee)
CALL Play(ydmusic)
```

More Information

The Play subroutine starts a sound playing and returns immediately (i.e. it does not wait for the sound to finish). Its argument is a predefined constant specifying music built-in to the Vectrex, or a handle to Music object created by the Music function, or nil which stops any sound that is currently playing (including sounds generated with the Sound() subroutine).

See Also

Music, MusicIsPlaying, Sound

12.23 PtInRect

Description

Returns 1 if the specified point is within the specified rectangle; otherwise returns 0.

Example

```
point1 = {10, 10}
point2 = {20, 1.5}
rect = {{1, 1}, {100, 3}}
x1 = 1
y1 = 1
x2 = 100
y2 = 3
PRINT PtInRect(rect, point1) ' prints 0
PRINT PtInRect(rect, point2) ' prints 1
PRINT PtInRect(x1, y1, x2, y2, 1, 100) ' prints 1
```

More Information

The PtInRect function accepts the coordinates of a rectangle and a point, and returns 1 if the point lies within the rectangle or on any of the edges. The rectangle can be specified in a 2x2 array or as four arguments. The rectangle does not need to be normalized, i.e. the first X coordinate can be either larger or smaller than the second X coordinate, and the first Y can be larger or smaller than the second Y.

See Also

Distance

12.24 PutSpriteAfter

Description

Change the position of a sprite in the drawing list.

Example

```
CALL PutSpriteAfter(spritePosition, sprite)
```

More Information

If *sprite* is in the drawing list, it is removed. Then, it is inserted into the drawing list after *spritePosition*. If *spritePosition* is not in the drawing list, *sprite* is added to the end of the drawing list.

It is unusual that you would use this in a program; it is more likely that you would use it in Immediate mode or Debug mode when you are experimenting with how to draw what you want.

See Also

PutSpriteBefore, RemoveSprite

12.25 PutSpriteBefore

Description

Change the position of a sprite in the drawing list.

Example

```
CALL PutSpriteBefore(spritePosition, sprite)
```

More Information

If *sprite* is in the drawing list, it is removed. Then, it is inserted into the drawing list before *spritePosition*. If *spritePosition* is not in the drawing list, *sprite* is added to the start of the drawing list.

It is unusual that you would use this in a program; it is more likely that you would use it in Immediate mode or Debug mode when you are experimenting with how to draw what you want.

See Also

PutSpriteBefore, RemoveSprite

12.26 RegularPolygon

Description

Returns an array of (mode, x, y) rows that form a regular polygon.

Example

```
hexagon = RegularPolygon(6, 10)
octagon = RegularPolygon(8, 15, 180.0 / 8)
octasprite = LinesSprite(octagon)
```

More Information

This function returns an array that can be used as an argument to LinesSprite to draw a regular polygon (a polygon where all the sides are the same length) on the screen. The first argument is the number of sides. The second argument is the radius of the polygon (the distance from the center to one of the vertices). The optional third argument is the number of degrees to rotate the polygon counter-clockwise; if the argument is omitted, the rotation is zero degrees and one of the vertices will rest on the X axis.

In the returned array, each row will have (mode, x, y). The first row will have mode set to MoveTo (0), so the pen will move to the first vertex without drawing a line. The rest of the rows will have mode set to DrawTo (1) so the edges of the polygon are drawn. Because of the MoveTo at the beginning, a polygon of N sides will need an array of N+1 rows.

When the polygon is drawn, the current pen position will be the center of the polygon.

See Also

Offset

12.27 RemoveSprite

Description

Removes a sprite from the drawing list.

Example

```
CALL RemoveSprite(octagon)
```

See Also

PutSpriteBefore

12.28 ReturnToOriginSprite

Description

Creates a pseudo-sprite that moves the pen back to the center of the screen.

Example

```
rtn = ReturnToOriginSprite()
```

More Information

The ReturnToOrigin pseudo-sprite is created, added to the end of the drawing list, and its handle is returned. This pseudo-sprite is useful for putting the pen at a known location (the center of the screen) and for eliminating pen drift.

12.29 ScaleSprite

Description

Creates a pseudo-sprite that changes the scale at which the Vectrex moves the pen and draws lines.

Example

```
scale = ScaleSprite(60)
scale = ScaleSprite(64, 324 / 0.097)
```

More Information

The Scale pseudo-sprite is created, added to the end of the drawing list, and its handle is returned. The scale value can vary from 1 to 255.

When the Vectrex32 is drawing Lines3D sprites, it needs to know how to convert Vectrex units to actual distances on the screen. So when you're using 3D graphics, you should include the optional second parameter to ScaleSprite which specifies the conversion ratio. In the example above, the scale is being set to 64. Using the ScaleTest.bas program included with the Vectrex32, we can determine that at scale 64, a line that is 324 units long is 0.97 meters long on the screen (about 3.8 inches). The denominator should use the same units that your Lines3D sprites are using. For example, if you were specifying the coordinates of your 3D objects in inches, the second argument would be 324 / 3.8.

See Also

SpriteScale

12.30 SetFrameRate

Description

Sets the rate, in frames per second, at which the screen is refreshed.

Example

```
CALL SetFrameRate(50)
```

More Information

See section 6.6, Frames.

See Also

GetFrameRate

12.31 Sound

Description

Allows direct control of the Vectrex's Programmable Sound Generator (PSG).

Example

```
' Make a sound like a phaser firing. First, set register 7
' to enable Channel A, register 8 to maximum volume for
' channel A, and register 0 to 48, the initial tone to play
' through channel A
regs = {{7, 0x3e}, {8, 15}, {0, 0x30}}
soundTone = 0x30

' This could be the infinite loop of your game
while true do
    ' Start a new frame
    controllers = _
       WaitForFrame(JoystickDigital, Controller1, JoystickX)

    ' If we're playing the sound
    if soundTone >= 0x30 and soundTone <= 0x70 then

        ' Set channel A's tone
        regs[3, 2] = soundTone

        ' Calling Sound() gives the Vectrex32 the new
        ' register values
        call Sound(regs)

        ' Go to the next tone to play
        soundTone = soundTone + 1
        ' When the next frame starts, the register values will
        ' be written to the PSG

    ' Else if we've swept the tone from 0x30 to 0x70
    elseif soundTone = 0x71 then

        ' Disable channel A
        regs[1, 2] = 0x3f

        ' We're done playing the sound
        soundTone = 0
    endif

    ' Do your other game stuff here
endwhile
```

More Information

The Sound subroutine takes a two dimensional array as an argument. Each row of the array has a register number (an integer between 0 and 13) and a value to write to that register (a value from 0 to 255). At the beginning of the next frame (e.g. when your program calls WaitForFrame) the register values are written to the PSG. This means that the frame rate affects the speed at which you can change sounds.

You can find details of the PSG's registers' functions in Using the Sound Generator Chip.

See Also

Low-Level Sound Control, Using the Sound Generator Chip, Play

12.32 SpriteClip

Description

Defines a clipping rectangle for a sprite.

Example

```
left = -50
right = 50
top = 60
bottom = -60
clippingRect = {{left, top}, {right, bottom}}
sprite = LinesSprite(lineArray)
CALL SpriteClip(sprite, clippingRect)
CALL SpriteClip(sprite, nil) ' Stop clipping the sprite
```

More Information

Applies a clipping rectangle to a sprite. The array holding the rectangle is live data; you can modify its contents and the clipping rectangle applied to the sprite will change. Multiple sprites can use the same array for the clipping rectangle (and since it's live data, changing the array's values will change the clipping for all the sprites).

Calling SpriteClip with nil instead of a rectangle removes the clipping rectangle.

SpriteClip works only with Dots sprites, Lines sprites, and Lines3D sprites (in which case the clipping is applied after the 3D object has been flattened into a 2D image)..

See Also

Transforms and Clipping

12.33 SpriteEnable

Description

Turns a sprite or pseudo-sprite on or off.

Example

```
CALL SpriteEnable(sprite, false) ' Disable the sprite
CALL SpriteEnable(sprite, true)  ' Re-enable the sprite
```

More Information

When a sprite is created and added to the drawing list, it is initially enabled, which means it will draw (in the case of a sprite) or effect how subsequent sprites are drawn (in the case of a pseudo-sprite). SpiteEnable can disable the sprite so that it is not drawn or does not affect subsequent sprites, and it can re-enable the sprite.

This function is useful if you want to have objects disappear from the screen or reappear on the screen. An alternative is to remove and add sprites to the list with SpriteRemove, PutSpriteBefore and PutSpriteAfter.

12.34 SpriteGetRotation

Description

Gets the current rotation, in degrees, applied to a sprite.

Example

```
angle = SpriteGetRotation(sprite)
angles = SpriteGetRotation(sprite3D)
```

More Information

The returned angle is normalized to be between 0 and 360 degrees. It indicates counter-clockwise rotation.

If the sprite is a Lines or Dots sprite, the returned angle is a single number. If it's a Lines3D sprite, the returned value is a 1x3 array with pitch, roll, and yaw.

See Also

SpriteRotate, SpriteSetRotation

12.35 SpriteIntensity

Description

Changes the intensity setting of a previously created Intensity pseudo-sprite.

Example

```
intensity = IntensitySprite(60)
CALL SpriteIntensity(intensity, 70)
```

More Information

This subroutine works only on Intensity pseudo-sprites. The intensity can range from 0 to 127.

See Also

IntensitySprite

12.36 SpriteMove

Description

Changes the coordinates of a previously created Move pseudo-sprite.

Example

```
move = MoveSprite(10, 10) ' Move the pen by (10, 10)
CALL SpriteMove(move, 20, 20) ' Changes the 'move' sprite so that
it moves to (20, 20)
```

More Information

This subroutine works only on a Move pseudo-sprite.

See Also

MoveSprite

12.37 SpriteRotate

Description

Adds to the rotation of a sprite.

Example

```
CALL SpriteRotate(sprite, 45) ' Sprite is rotated 45 degrees
CALL SpriteRotate(sprite, 20) ' Sprite is now rotated 65 degrees

' Rotates a Lines3D sprite by 10 degrees pitch, 20 degrees roll,
' and 30 degrees yaw
CALL SpriteRotate(sprite3D, 10, 20, 30)
```

More Information

Rotates the sprite counter-clockwise by the specified angle in degrees. If the sprite had been rotated previously, this adds to the rotation.

See Also

SpriteSetRotation

12.38 SpriteScale

Description

Changes the scale of a previously created Scale pseudo-sprite.

Example

```
scale = ScaleSprite(60) ' Sets the Vectrex scale to 60
' Changes the Vectrex scale to 70
CALL SpriteScale(scale, 70)

' Optionally set scale-to-length ratio for 3D graphics
' At scale 80, a 262 unit line is 0.098 meters long
CALL SpriteScale(scale, 80, 262 / 0.098)
```

More Information

This subroutine works only on a Scale pseudo-sprite.

See Also

ScaleSprite

12.39 SpriteSetMagnification

Description

Alters the magnification of a sprite.

Example

```
CALL SpriteSetMagnification(sprite, 2.5) ' Enlarges the sprite
CALL SpriteSetMagnification(sprite, 1) ' Returns the sprite to
its original size
```

More Information

This subroutine works only on Dots sprites, Lines sprites, and Lines3D sprites.

See Also

Magnification

12.40 SpriteSetRotation

Description

Sets the rotation of a sprite.

Example

```
CALL SpriteSetRotation(sprite, 45) ' Sprite is rotated 45 degrees
CALL SpriteSetRotation(sprite, 20) ' Sprite is now rotated 20
degrees

' 3D sprite is now rotated 4 degrees pitch, 5 degrees roll,
' and 6 degrees yaw
CALL SpriteSetRotation(sprite3d, 4, 5, 6)
```

More Information

Sets the rotation of the sprite to the specified angle in degrees. If the sprite had been rotated previously, this replaces the old rotation value.

See Also

SpriteRotate

12.41 SpriteTranslate

Description

Sets the translation of a sprite.

Example

```
translation = {10, 15}
CALL SpriteTranslation(sprite, translation)

translation3D = {10, 15, 20}
CALL SpriteTranslation(sprite3D, translation3D)
```

More Information

Adds the specified X, Y, and (in the case of a 3D sprite) Z offsets to the sprite before drawing it. The array is live data, so the translation can be changed by modifying the values in the array.

See Also

Translation

12.42 TextListSprite

Description

Creates a sprite that will draw multiple strings of text on the screen.

Example

```
instructions = {{-50, 90, "INSTRUCTIONS"}, _
                {-80, 70, "BTN 1&2 CHANGES SCALE"}, _
                {-80, 50, "JOYSTK MOVES CROSSHAIR"}, _
                {-80, 30, "READ COORDINATES"}, _
                {-80, 0, "PRESS BTN 1 TO START"}}

tl = TextListSprite(instructions)
```

More Information

The TextList sprite is created, added to the end of the drawing list, and its handle is returned. The argument is an array of (x, y, text) rows. The TextList sprite is created, added to the end of the drawing list, and its handle is returned. The array is live data, so changing its contents changes what is drawn on the screen.

The Vectrex replaces some ASCII characters with special symbols when it draws them on the screen. See On-Screen Characters for a character map.

After drawing a TextList sprite, the pen is automatically reset to the center of the screen.

See Also

TextSizeSprite, TextSprite,

On-Screen Characters

12.43 TextSizeSprite

Description

Creates a pseudo-sprite that sets the size at which subsequent Text and TextList sprites will be drawn.

Example

```
textSize = {40, 5}
call TextSizeSprite(textSize)
```

More Information

The TextSize pseudo-sprite is created, added to the end of the drawing list, and its handle is returned. The array specifies the width and height of each letter in the text. Strangely, they don't seem to be using the same units, e.g. in the example where the *textSize* is {40, 5}, letters are not 8 times wider than they are tall; in fact, they look quite nicely proportioned.

See Also

TextListSprite, TextSprite

12.44 TextSprite

Description

Creates a sprite that will draw a single string of text on the screen.

Example

```
text = TextSprite("HELLO")
```

More Information

The text sprite is created, added to the end of the drawing list, and its handle is returned. The sprite will draw the specified text at the current pen location.

The Vectrex replaces some ASCII characters with special symbols when it draws them on the screen. See On-Screen Characters for a character map.

After drawing a Text sprite, the pen is automatically reset to the center of the screen.

See Also

TextListSprite, TextSizeSprite,

On-Screen Characters

12.45 VectorRotate

Rotates a 2D or 3D vector by a given angle.

Example

```
' Make a 2D vector 1 unit long at a 45 degree angle
vector2D = {0.7071, 0.7071}
' Rotate it counter-clockwise 45 degrees
vector2D = VectorRotate(vector2D, 45)
' The result will be about {1, 0} (with some rounding errors)
print vector2D

' Make a 3D vector 5 units long pointing down the Z axis
vector3D = {0, 0, 5}
' Rotate it 45 degrees pitch (counter-clockwise around
' the X axis)
vector3D = VectorRotate(vector3D, 45, 0, 0)
' The result will be {0, 3.53553, 3.53554}
print vector3D
```

More Information

Rotating a vector is useful for determining how to move an object forward, when its forward direction is not along the X, Y, or Z axis. This is discussed in section 7.9.4, The Camera and in section 8.2, Demo.

12.46 Version

Description

Returns the version number of the Vectrex32 firmware.

Example

```
IF Version() < 110 THEN
    PRINT "Upgrade the Vectrex32 firmware to version 1.00 or
newer."
    STOP
ENDIF
```

More Information

The returned version number is the major and minor version number times 100, e.g. if the firmware is version 3.14, Version() will return 314.

12.47 WaitForFrame

Description

Blocks execution until the next frame is drawn on the screen; returns the current state of the controllers.

Example

```
controllers = WaitForFrame(JoystickDigital, _
   Controller1 + Controller2, JoystickX + JoystickY)
controllers = WaitForFrame(JoystickAnalog, _
   Controller1 + Controller2, JoystickX + JoystickY, 4)
```

More Information

This function is used within a BASIC program to draw the screen and get the status of the controllers. If your program does not call this function, the screen will not get drawn. (When in Immediate or Debug mode, the WaitForFrame is called by the Vectrex32's operating system.)

The first argument specifies how to read the joystick(s): it can be JoystickAnalog, JoystickDigital, or JoystickNone. The second argument specifies which controller(s) to read: it can be Controller1, Controller2, Controller1 + Controller2, or ControllerNone. The third argument specifies which axes of the joystick(s) to read: it can be JoystickX, JoystickY, JoystickX + JoystickY, or JoystickNone.

When using JoystickAnalog, you can specify an optional fourth argument, the joystick resolution. This value should be zero or a power of two, less than or equal to 128 (so values are 0, 1, 2, 4, 8, 16, 32, 64, or 128). The larger the number, the lower the joystick resolution.

The return value is an array with two rows. The first row has the status of controller 1 and the second row has the status of controller2. Not all the values in the array might be set; if you didn't ask for a controller or a joystick to be read, the corresponding entries in the array will be zero.

The first two elements of the row are the X and Y positions of the joystick. Their values range from -128 to +127. If the first argument is JoystickDigital, then the value returned will be either zero, 64, or -64; if the argument is JoystickAnalog, the value can be anywhere between -128 and +127 (unfortunately, joysticks can get out of calibration - my joystick reads -16 at its center position - so try to account for that in your program).

The third through sixth elements are the statuses for buttons 1, 2, 3 and 4; a 1 means the button is pressed while a 0 means it is not.

Appendix A. On-Screen Characters

The Vectrex can display upper case letters, digits, some punctuation, and some special characters. This table shows how ASCII characters in text sprites map to symbols on the Vectrex screen.

ASCII	Displayed	ASCII	Displayed	ASCII	Displayed	ASCII	Displayed	ASCII	Displayed	ASCII	Displayed
(space)	(space)	0	0	@	@	P	P	`	car figure?	p	
!	!	1	1	A	A	Q	Q	a	↑	q	
"	"	2	2	B	B	R	R	b	♪	r	
#	#	3	3	C	C	S	S	c	↓	s	
$	$	4	4	D	D	T	T	d	○	t	
%	%	5	5	E	E	U	U	e	●	u	
&	&	6	6	F	F	V	V	f	•	v	
'	'	7	7	G	G	W	W	g	©	w	
(	(	8	8	H	H	X	X	h	spaceship?	x	
)	)	9	9	I	I	Y	Y	i	man?	y	
*	*	:	:	J	J	Z	Z	j	☺	z	
+	+	;	;	K	K	[	[	k	☹	{	
,	,	<	<	L	L	\	\	l	∞	|	
-	-	=	=	M	M	]	]	m	□	}	
.	.	>	>	N	N	^	^	n	■	~	
/	/	?	?	O	O	_	_	o	■	(del)	

Appendix B. USING THE SOUND GENERATOR CHIP

The Vectrex produces sounds using the General Instruments AY-3-8912 Programmable Sound Generator (PSG) chip. The Vectrex32 allows direct, low-level control of this chip via the Sound function. This section describes the low-level control of the PSG and was adapted from the data sheet for the AY-3-8912.

> If you read the original data sheet, you might notice that registers are numbered differently there. The data sheet uses octal for the registers whereas we use decimal here. So, for example, registers R10, R11, and R12 in the data sheet are referred to as registers R8, R9, and R10 here.

Description

The AY-3-8912 Programmable Sound Generator (PSG) is an LSI Circuit which can produce a wide variety of complex sounds under software control.

In order to perform sound effects while allowing the processor to continue its other tasks, the PSG can continue to produce sound after the initial commands have been given by the control processor. The fact that realistic sound production often involves more than one effect is satisfied by the three independently controllable channels available in the PSG.

Architecture

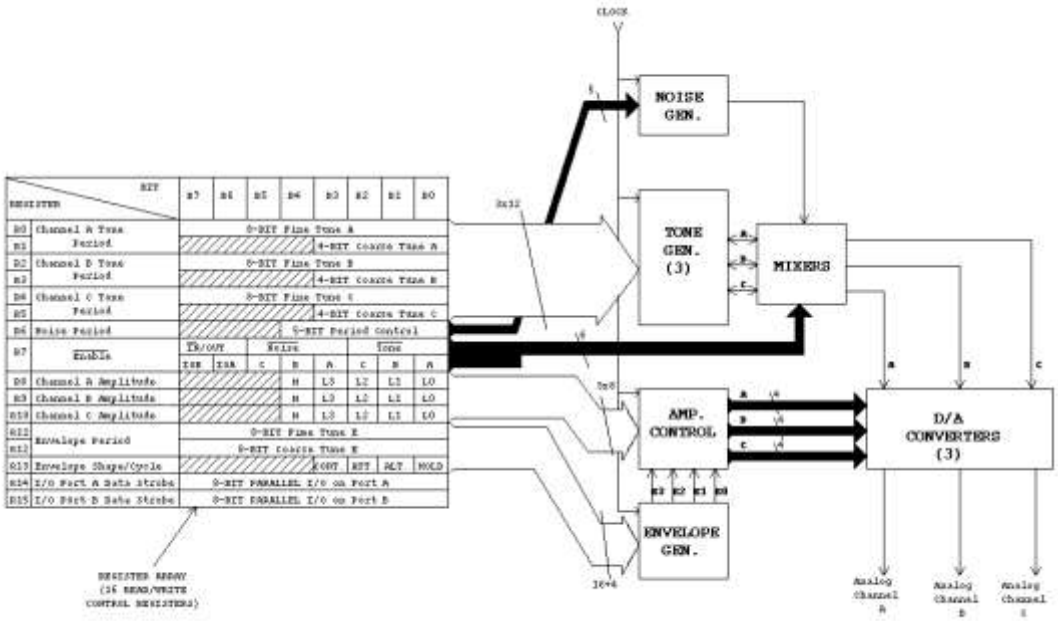
The AY-3-8912 is a register oriented Programmable Sound Generator (PSG). Control commands are issued to the PSG by writing to 16 registers.

All functions of the PSG are controlled through the 16 registers which once programmed, generate and sustain the sounds, thus freeing the system processor for other tasks.

Sound Generating Blocks

The basic blocks in the PSG which produce the programmed sounds include:

- Tone Generators produce the basic square wave tone frequencies for each channel (A,B,C)
- Noise Generator produces a frequency modulated pseudo random pulse width square wave output.
- Mixers combine the outputs of the Tone Generators and the Noise Generator. One for each channel (A,B,C)
- Amplitude Control provides the D/A Converters with either a fixed or variable amplitude pattern. The fixed amplitude is under direct CPU control; the variable amplitude is accomplished by using, the output of the Envelope Generator.
- Envelope Generator produces an evelope pattern which can be used to amplitude modulate the output of each Mixer.
- D/A Converters the three D/A Converters each produce up to a 16 level output signal as determined by the Amplitude Control.



PSG Block Diagram

Operation

Since all functions of the PSG are controlled by the processor via a series of register loads, a detailed description of the PSG operation can best be accomplished by relating each PSG function to the control of its corresponding register. The function of creating or programming a specific sound or sound effect logically follows the control sequence listed:

Operation	Registers	Function
Tone Generator Control	R0-R5	Program tone periods
Noise Generator Control	R6	Program noise period
Mixer Control	R7	Enable tone and/or noise on selected channels
Amplitude Control	R8-R10	Select "fixed" or "envelope variable" amplitudes
Envelope Generator Control	R11-R13	Program envelope period and select envelope pattern
Not used for sound	R14-R15	

Tone Generator Control (Registers R0, R1, R2, R3, R4, R5)

The frequency of each square wave generated by the three Tone Generators (one each for Channels A,B and C) is obtained in the PSG by first counting down the input clock by 16, then by further counting down the result by the programmed 12-bit Tone Period value. Each 12-bit value is obtained in the PSG by combining the contents of the relative Course and Fine Tune registers, as illustrated in the following:

Coarse Tune Register	Channel	Fine Tune Register
R1	A	R0
R2	B	R1
R3	C	R2

Course Tone Register								Fine Tune Register							
B7	B6	B5	B4	B3	B2	B1	B0	B7	B6	B5	B4	B3	B2	B1	B0
NOT USED				TP11	TP10	TP9	TP8	TP7	TP6	TP5	TP4	TP3	TP2	TP1	TP0
				12-bit Tone Period (TP) to Tone Generator											

Noise Generator Control (Register R6)

The frequency of the noise source is obtained in the PSG by first counting down the input clock by 16, then by further counting down the result by the programmed 5-bit Noise Period value. This 5-bit value consists of the lower 5-bits (B4--B0) of register R6, as illustrated in the following:

Noise Period Register R6

B7	B6	B5	B4	B3	B2	B1	B0
NOT USED			5-bit Noise Period (NP) to Noise Generator				

Mixer Control Enable

Register R7 is multi-function $\overline{\text{Enable}}$ register which controls the three Noise/Tone. The line above "Enable" indicates inverse logic, i.e. when a bit is 0, the corresponding function is enabled and when the bit is 1, the corresponding function is disabled.

The Mixers, as previously described, combine the noise and tone frequencies for each of the three channels. The determination of combining neither/either/both noise and tone frequencies on each channel is made by the state of bits B5--B0 of R7.

These functions are illustrated in the following:

Mixer Control Enable Register R7 1

B7	B6	B5	B4	B3	B2	B1	B0
NOT USED		$\overline{\text{Noise Enable}}$			$\overline{\text{Tone Enable}}$		
		C	B	A	C	B	A

Amplitude Control (Registers R8, R9, R10)

The amplitudes of the signals generated by each of the three D/A Converters (one each for Channels A, B and C) is determined by the contents of the lower 5-bits (B4-B0) of registers R8, R9 and R10 as illustrated in the following:

Amplitude Control Register	Channel
R10	A
R11	B
R12	C

B7	B6	B5	B4	B3	B2	B1	B0
NOT USED			M	L3	L2	L1	L0
			amplitude "mode"	4-bit "fixed" amplitude level			

Envelope Generator Control (Registers R11, R12, R13)

To accomplish the generation of fairly complex envelope patterns, two independent methods of control are provided in the PSG: first, it is possible to vary the frequency of the envelope using registers R11 and R12; and second, the relative shape and cycle pattern of the envelope can be varied using register R13. The following paragraphs explain the details of the envelope control functions, describing first the envelope period control and then the envelope shape/cycle control.

Envelope Period Control (Registers R11, R12)

The frequency of the envelope is obtained in the PSG by first counting down the input clock by 256, then by further counting down the result of the programmed 16-bit Envelope Period value. This 16-bit value is obtained in the PSG by combining the contents of the Envelope Coarse and Fine Tune registers, as illustrated in the following:

Envelope Coarse Tune Register R12								Envelope Fine Tune Register R11							
B7	B6	B5	B4	B3	B2	B1	B0	B7	B6	B5	B4	B3	B2	B1	B0
EP15	EP14	EP13	EP12	EP11	EP10	EP9	EP8	EP7	EP6	EP5	EP4	EP3	EP2	EP1	EP0
16-bit Envelope Period (EP) to Envelope Generator															

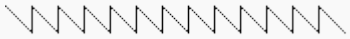
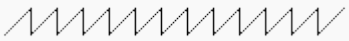
Envelope Shape/Cycle Control (Register R13)

The Envelope Generator further counts down the envelope frequency by 16, producing a 16-state per cycle envelope pattern as defined by its 4-bit counter output, E3 E2 E1 E0. The particular shape and cycle pattern of any desired envelope is accomplished by controlling the count pattern (count up/count down) of the 4-bit counter and by defining a single cycle or repeat-cycle pattern.

This envelope shape/cycle control is contained in the 4 bits (B3--B0) of register R13. Each of these 4 bits controls a function in the envelope generator, as illustrated in the following:

B7	B6	B5	B4	B3	B2	B1	B0
NOT USED				Hold	Alternate	Attack	Continue
				Function (To Envelope Generator)			

Envelope Shape/Cycle Operation

R15 Bits				GRAPHICAL REPRESENTATION OF ENVELOPE GENERATOR OUTPUT (E3 E2 E1 E0)
B3	B2	B1	B0	
Continue	Attack	Alternate	Hold	
0	0	x	x	
0	1	x	x	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1