GAMES MASTER

The Complete Games Designer
For The SAM Coupé

BETA SOFT

********** CONTENTS **********

INTRODUCTION	1
COPYRIGHT RESTRICTIONS	1
MAKING A WORKING COPY	2
LATE NEWS	2
STARTING TO USE THE PROGRAM	3
PICK SPRITE Option	7
EDIT SPRITE DETAILS Option	7
EDIT GRAPHICS Option	12
SELECT MASKING Option	14
EDIT PALETTES Option	16
GRAB FRAME Option	16
ANIMATE SPRITE Option	17
SPRITE vs. SPRITE COLLISIONS Option	18
SPRITE vs. BLOCK COLLISIONS Option	19
RUN GAME Option	19
OTHER WAYS TO RUN A GAME	19
GAME PAUSE KEY	20
EDIT MODULES Option	20
EDIT GAME DETAILS Option	21
EDIT PATHS Option	23
EDIT SOUNDS Option	24
EDIT KEYS Option	25
EDIT ANIMATION SEQUENCE Option	25
EDIT BLOCKS Option	26
EDITOR EXTRAS Option	28
LOAD SPRITE Option	28
SAVE SPRITE Option	28
LOAD GAME Option	29
SAVE GAME Option	29
Stand-alone CODE Games	29
Game Protection	30
LOAD SCREEN Option	31
SAVE SCREEN Option	31
EXIT TO BASIC Option	31
MEMORY MANAGER Option	31
WHEN YOU ARE SHORT OF MEMORY	32
CLEAR SUBMENU Option	32
THE GAMES MASTER CONTROL LANGUAGE	33
Introduction	33
The Coordinate System	34
Sprite Planes	34
Sprite Numbers	35
GMCL Expressions	36
GMCL COMMANDS	37
GMCL FUNCTIONS	54
EXAMPLE PROGRAMS	56
UTILITY PROGRAMS	56
MODIFYING THE EDITOR	57
SPRITE FILE FORMAT	57

INTRODUCTION

Games Master is a complete games development package for the SAM Coupe. It allows you to create and modify sprite graphics, and tell the computer how to move and animate them. Other parts of the program allow you to edit sound effects, control keys, load and save games and sprites, set up collision actions, etc. Because the computer does most of the work, you need to do very little programming and can produce results quickly. No delving into machine code is required. What programming you do will probably be using the Games Master Control Language (GMCL for short). You can use Basic or machine code subroutines if you like, but you probably won't need to.

I am always glad to hear from users. If you have any comments, suggestions for improvements or problems, please write to:

BETASOFT, 24 WYCHE AVE., KINGS HEATH, BIRMINGHAM, B14 6LQ.

I would like to put together a compilation disk of programs written using the Games Master system, including contributions from users. This would be sent to users for a small fee. Prizes (and fame!) would be given for the best contributions. Please send in anything you feel might be suitable.

COPYRIGHT RESTRICTIONS

Most of the game development and editing is done using a large Basic program called (logically enough) the EDITOR. This program is copyright and you MUST NOT give copies to anyone else. The games you produce with the Editor will be self-contained machine code (unless you have used Basic subroutines) and you are free to copy these games and give them away or sell them, provided that you mention that the program was produced using Games Master in your documentation or packaging. However, the sprite driver and other routines which are part of the machine code saved with each game must only be used as part of a game produced by Games Master - you cannot use them in any other product.

This program took me a lot of time and effort to write, and I hope it reflects that. The price is very reasonable. Please do not give away my work - let your friends buy their own copy, so that I can make a living and continue to develop new products for this excellent machine! I could have protected this program, but I thought it would inconvenience you, and it shouldn't be necessary. Don't let me down! The disks and their contents are individually marked and copies can be traced to their source.

ACKNOWLEDGEMENTS

I'd like to thank Dave Tonks, David Ledbury and Stephen Wilson for providing some of the graphics, and Glen Cook for entertaining phone calls. I'm also grateful to my wife Celia, whose patience lasted for most of this project.

MAKING A WORKING COPY OF GAMES MASTER

You should use a working copy or copies of the supplied disk for day-to-day use, rather than the original disk. To make a working copy, turn on the computer and load your DOS. Have ready a formatted disk. Place the Games Master disk in drive 1. If you are using MasterDOS on a 2-drive system, place the blank formatted disk in drive 2 and then use BACKUP "d1" TO "d2" to copy the disk. On a 1-drive system use BACKUP "d1" TO "d1" and swap the disks when you are prompted to. You might want to erase SAMDOS from the completed copy and replace it with MasterDOS or the MasterDOS/MasterBasic combination. This will improve the speed and reliability of disk operations and, with MasterBasic, the speed of the Editor program will be increased.

If you are using SAMDOS, I strongly recommend that you order a copy of MasterDOS from Betasoft (only £15.99) but in the meantime on a 2-drive system put the blank disk in drive 2 and use FORMAT "d2" TO "d1" or on a 1-drive system use COPY "d1:*" TO "d1:*" and swap the disks in response to the prompts.

Ensure that you keep the original Games Master disk in a safe place - though naturally I will replace it if anything bad happens to it. Just return the disk and enclose a stamped addressed envelope. If you live abroad forget the stamp but enclose an International Postal Reply Coupon instead.

If you bought this product directly from Betasoft, you are already recorded as a customer with possible upgrade privileges. If you bought it elsewhere, it would be a good idea to send me your name and address and tell me where you got the product. This information may be used to tell you about future products.

LATE NEWS

If there IS any Late News, for example, additions to the program or manual, they will be described in the file called "readme" on the Games Master disk - just LOAD "readme". You might be interested in the Justify procedure that the program uses, too!

STARTING TO USE THE PROGRAM

To load the program, place the working disk in drive 1 and press F9. The Editor and supporting machine code will load and run. You will be presented with a superficially somewhat formidable main menu which provides all the major editor features in an easily-accessible way. In general, you move the cursor bar to the option you want using the cursor arrow keys and press RETURN. Most options can be abandoned by pressing F9, or by pressing just RETURN when you are prompted to input a value or file name.

Most possible errors are trapped within the program, but if the program returns to Basic with an error message, or because you have pressed ESC or used the EXIT TO BASIC option, you can restart it by GO TO main or RUN. RUN resets some variables such as cursor positions, masking method, outline option, etc. but leaves your game intact. You will probably have to type r-u-n, rather than just pressing F4, since the Editor re-defines keys.

LOADING A SPRITE

As a simple exercise to get something moving on the screen, we will load a sprite from the ready-defined selection on the disk. A sprite is a computer thingy that combines graphic data with other information that controls how it moves and acts on the screen. Select the LOAD SPRITE option on the upper right-hand side of the menu. You will be presented with a list of numbered entries, each of which can hold a sprite. You are asked to select a free entry, using the cursor keys, and then press RETURN. In this case you can just press return, since the cursor bar will point to entry 1 already, and all the entries are free (there are no sprite names listed in them.)

You will now see a directory of files that end in ".s" - these are all sprite files, and contain the data that defines a single sprite. Now type in "egship" or "egship.s" in response to the request for a file name. The file will load, and you will be returned to the main menu.

LOOKING AT GRAPHICS

From the main menu, you can have a look at the sprite you have just loaded by selecting EDIT GRAPHICS. The left and right cursor keys let you look through the different views of the same sprite; these are called "frames". They can be placed on the screen in various "animation sequences" to make a sprite flash, or appear to walk, flap its wings, rotate, etc. For now, we will avoid altering the graphics in any way - just press F9 and then "N" or F9 again to return to the main menu.

LOOKING AT SPRITE PROPERTIES

You can examine the non-graphical properties of the selected sprite by using the EDIT SPRITE PROPERTIES option. You will see the first of two pages of data about the sprite, much of which we will ignore for now. Notice, however, that ANIMATION SEQUENCE is set to 0. This means that there is no animation sequence assigned to the sprite, so the current frame will be displayed all the time it is on-screen. The appearance of the sprite won't change.

As you know from looking at the graphics, and from the current display, the sprite has seven frames, and we will use them later on. For the moment, we will leave the sprite non-animated. It will, however, move around the screen. MOVEMENT TYPE is set to 1 (SIMPLE) which means that the X SPEED and Y SPEED values shown will be used to move the sprite to successive new positions on the screen. BOUNCES is set to YES, which means that the sprite bounces off obstructions, and EDGE LIMITED is set to YES, meaning "treat the screen edge as an obstruction". Perhaps you can guess that this combination should make the sprite move around the screen, bouncing off the edges when it hits them. Now press F9 to return to the main menu, and select EDIT MODULES.

EDITING A MODULE

Our sprite is ready to go - but we still have to tell the computer to start it off. To do this we need to use a single command in Games Master Control Language. Enter 1 as the module to edit (this is the default so you can just press return) and you will see a blank module, which is an empty page into which you can type GMCL commands. The cursor is the white square at the top left of the screen. You do not need and cannot use line numbers, so just type PLACE 1,10,40,4. This means "place sprite 1 at x coordinate 10, y coordinate 40, sprite plane 4. The sprite plane determines whether sprites can collide with other sprites or blocks, or pass over or under them, but don't worry about it for now. Press F7 to compile the module; the command will be translated into machine code.

You now have various choices such as saving or listing the module, but just press a key such as RETURN and you will be returned to the main menu.

SETTING THINGS GOING

Finally we are ready to run the "game". Select RUN GAME. The program will execute module 1, the sprite will be set going, and you should see the sprite bouncing around the screen. Module 1 is no longer being executed - you have told the computer "Place sprite one at these coordinates and then keep handling it, according to its properties". After that, no further commands are needed to keep the sprite moving.

ANIMATION

Now leave the game by pressing F9, and let's modify things to be a little more interesting. Select EDIT SPRITE PROPERTIES and move the cursor bar down to ANIM. SEQUENCE using the up and down cursors, then press the right cursor until the value increases to 7. Use the left cursor to decrease the value if you overshoot. Now press F7 to use the new values and return to the main menu. (F9 would have thrown away any changes before returning to the main menu.)

We have made the sprite animated, using animation sequence 7, but what does that mean? There is nothing in the sprite data to say what animation sequence 7 is; instead, animation sequences are part of the overall game, and can be edited in isolation from the sprites. In a way this is a pity, because it means that a

sprite's behaviour is not determined solely by the sprite data, but also by more general game data. However, there are advantages in doing things this way. Animation sequences can be complex without expanding the size of the sprite data, since only a single value determines which complex sequence from the games's list of sequences is used. Sequences can be shared by many sprites, some of which can be at a different position in the sequence, and a single change to a sprite's data can select another sequence and change its behaviour completely.

Many of the most common animation sequences are ready-programmed, so you need to do very little. Sequence 7 is one of these. If you select EDIT ANIM SEQ and enter 7, you will see that this sequence is defined to use frames 1 to 6 in sequence, then 5 to 2, with a "time" of 1. The "time" is the number of times to use each frame before going on to the next one. In this case the computer will use frame 1 just once, move the sprite, use frame 2 just once, move the sprite, etc. If the "time" was 10, the animation (i.e. alterations of the sprite's appearance, as distinct from its position) would be much slower, because the same frame would be used during 10 sprite moves. Different frames can have different times, allowing, say, a giant eye to be unchanging for 20 moves, and then blink quickly by using 2 or 3 "blinking" frames for just 1 move each. (You might like to load the "eye" sprite from disk, and set it going.) When the end of the animation sequence is reached, it will repeat automatically.

A list of the ready-programmed animation sequences is given in the section on the EDIT ANIM SEQ option, which also explains how to use the MODULE column to cause actions at particular points in the animation. This is just a quick run-through, though, so back to the main menu now!

ALTERING SPRITE PROPERTIES

Run the game again using the RUN GAME option. The sprite should be animated, as well as moving. Go back to the SPRITE PROPERTIES option and test the effect of altering X SPEED and Y SPEED - if you make the values negative, the initial direction of motion will be reversed. Try setting EDGE LIMITED to NO. Next you can switch to manual control by increasing MOVEMENT to 2 (PLAYER). As supplied, the program will control the sprite using the cursor keys, or a joystick.

ADDING MORE SPRITES

Perhaps by now you would like some company for your single sprite, so use the LOAD SPRITE option to load "egstar". This sprite will become the "selected sprite" which EDIT GRAPHICS, EDIT SPRITE PROPERTIES, etc. refer to. (To return to editing "egship" you would have to use the F5 and F2 keys, or the PICK SPRITE option on the main menu, to make it the "selected sprite" once more.) If you look you will see that "egstar" has only a single frame, and is not animated, although it is moving. It is not edge-limited so it will wrap round if it goes off-screen. Use EDIT MODULES to edit module 1. On the line after the PLACE command you entered before, type: PLACE 2,50,60,4 and on the line below that type: PLACE 2,66,110,4. (These particular coordinates are not important.)

It appears that we are telling the same sprite, number 2, to be in two different places at the same time! Press F7 to compile the module, then run the game, and you will see that this is exactly what happens! Games Master can handle multiple copies of the same basic sprite - you just have to use multiple PLACE (or other) commands. These commands make active copies from the unchanged master copies of the sprites that you load. Now try adding more copies of sprite 2 to module 1, perhaps using a final value (plane number) of 1 or 2, which will let the new copies pass under any sprite with a higher plane number, or a plane number of 8, 16 or 32, which will let the new copies pass over the existing sprites. You will learn more about the significance of these numbers later on.

COLLISIONS

To complete this trivial demonstration, let's make the "egstar" sprites make a noise when they hit the "egship". Select the SPR vs. SPR COLLISIONS option from the main menu. You will see a table of possible collision pairs. Move the cursor down by one row, so that it is on the row dealing with collisions of "egstar" with other sprites. We are already in the right column, since column 1 relates to sprite 1 ("egship"). Any module number you type now will be the module that will be executed when sprite 2 hits sprite 1. I usually use module numbers over 30 for collision handling, but you can use any number between 2 and 128. (Module 1 is already in use, of course.)

After typing the module number, place the cursor over the number, if it has moved, and press F8. The module will be ready for editing. (This is another way into the main menu EDIT MODULES option - you could use that instead if you preferred.) Now type SOUND 1, press F7 to compile, and you will return to the collision table, from which you can exit using F9, unless you want to specify more collision actions.

Running the game now will produce a sound on any egstar/egship collision, using pre-programmed sound 1. You can use a different pre-programmed sound, or edit sound 1 using the EDIT SOUNDS option if you like. If you have your sound output correctly connected to a system able to produce stereo sound, the perceived stereo position of the sound should vary according to where on the screen the collision occurs.

SAVING A GAME

To end this brief tour of Games Master, save the game using the SAVE GAME option. You can either use the SAVE DATA option, which saves just the game data, or the FULL GAME or AUTO GAME options, which save the machine code needed to handle the sprites as well, and allow the game to run without the Editor being loaded.

The rest of this manual deals with all the main menu options, in greater depth, followed by an explanation of the Games Master Control Language. Example programs and demos are included on the disk, along with sprite files and utility programs.

PICK SPRITE Option

The number and name of the "selected sprite" that EDIT GRAPHICS, EDIT SPR DETAILS, GRAB FRAME, ANIMATE SPRITE and SAVE SPRITE works on are shown on the main menu under PICK SPR. You can alter the selected sprite (no matter where the main menu cursor is) using the F5 and F2 keys. However, to look at the complete list of sprite numbers and names you must use the PICK SPR option. Move the cursor over PICK SPR and press RETURN.

You will be shown a page of 32 numbered entries, each of which can hold a sprite name up to eight characters long. Extra pages can be examined by using the F1 and F3 keys to show the complete set of 96 entries. The cursor bar can be moved left and right and up and down with the cursor keys. RETURN selects an entry as the "selected sprite" and returns you to the main menu. F8 can also be used to enter a new name or alter an existing one. The display also shows the amount of free memory available for holding more sprites. The name and number of the new selected sprite are shown on the main menu under PICK SPR.

EDIT SPRITE DETAILS Option

This option both displays details of the selected sprite, and allows you to edit most of them, using the up and down cursors to move the cursor bar, and left and right cursors to decrease or increase numbers, or toggle YES to NO and vice versa. F7 confirms all the changes you have made, and F9 abandons them. There are two pages of details. You can swap between them by pressing SPACE. It doesn't matter what page you are on when you press F7.

FRAMES cannot be altered. It shows the total number of frames.

WIDTH cannot be changed. It is the sprite's width in pixels.

HEIGHT cannot be changed. It is the sprite's height in pixels.

CURRENT FRAME can be changed, but this will be irrelevant unless the sprite has an animation sequence of zero (i.e. it is not animated) in which case it will determine what frame is seen when the sprite is displayed.

ANIMATION SEQUENCE when zero means that the sprite is not animated. Values between 1 and 32 denote an animation sequence from those editable using the main menu EDIT ANIM SEQ option. The sequence should be defined, if needed, before a game is run, or you will get an error message. If the sequence assigned to the sprite uses higher-numbered frames than the sprite actually has, the sprite will flicker and appear very odd, but no harm will be done.

CONDITIONAL refers to animation being conditional on sprite movement or not. NO means that the sprite will be constantly animated, YES that animation will stop when the sprite stops moving. For example, if the player controls a walking man, CONDITIONAL should be YES or the man will walk on the spot. Normally, when animation stops, frame 1 is selected, but this can be modified using a STOP module - see below.

MOVEMENT TYPE can have values from 0 to 4.

0 (UNMOVING) means that the sprite does not move.

1 (SIMPLE) means that the sprite makes simple moves initially determined by X SPEED and Y SPEED. Colliding with obstructions may stop the sprite, or cause it to bounce, but the speed and direction of movement will not change otherwise.

2 (PLAYER) means that the sprite is controlled by the player via a keyboard or joystick. The keys that respond are editable using the EDIT KEYS option on the main menu. The sprite speed when the player makes a move is determined by the values of X SPEED and Y SPEED. Both should normally be positive. (If X SPEED is positive, the sprite moves left when the player tries to move it left, but if it is negative, the sprite moves right!)

3 (PATH) means that the sprite follows a complex defined path from those editable using the EDIT PATHS option on the main menu. PATH should be set to 1-32, and the path should have been defined before the game is run, or you will get an error message. A path consists of a series of movement instructions; when all have been executed, this option goes through the same set of instructions again. If the sprite has moved back to its starting point by this time, the sprite will move repeatedly in a loop; if not, it will progress across the screen.

4 (ALT PATH) is like the previous option, but the sprite follows the path alternately forwards and backwards. This results in no overall movement, whatever the path. Both kinds of path can be interfered with by collisions with other sprites or with blocks.

X SPEED is the number of horizontal units moved when MOVEMENT TYPE is 1 or 2. If the value is negative, the initial direction will be left for movement type 1, otherwise right. Zero gives no horizontal movement. The units moved are normally TWO pixels, or 1 byte. This may seem odd, but it corresponds to exactly 1 byte on the screen, and means that the sprite can be placed onto the screen without any complex manipulations of the data. Also, in most cases movement by 2 pixels is quite acceptably smooth.

However, if you like you can set PIXEL X SPEED to YES and the units used will be single pixels. This has the disadvantage that a second copy of the graphics and masks for this sprite have to be created, doubling its memory usage. The second copies are rolled rightwards by 1 pixel ready to go on the screen, so this option is just as fast as normal. You must ensure that the graphics used for pixel x movement do not use the rightmost column of pixels, or odd pixels rolled off at the right will appear on the left when the sprite is placed on-screen. Restoring PIXEL X SPEED to NO will recover the space used for the second graphics copies.

Y SPEED is similar to X SPEED, but the units are always pixels. A typical value might be 2, with X SPEED being set to 1 (byte), giving diagonal movement.

Very large values of X and Y speeds (above about 10 pixels per move) are not recommended due to the jerkiness involved. They also make it possible for sprites to pass right through narrow barriers without the collision detection being triggered!

PATH is the path number followed by a sprite when movement type 3 or 4 is selected. See the main menu EDIT PATHS option for details.

COLLISION TYPE is initially the same as the sprite number, for sprites 1-32. This value is used in the SPR vs. SPR and SPR vs. BLOCK collision tables to control collision actions. However, there are only 32 collision types, and you can have up to 96 sprites, so sprites 33-96 will have to be assigned a suitable collision type in the range 1-32 if you want them to have collision-triggered actions. This means that they may have to have the same collision type as another sprite, but this is not usually a problem, since one often wants several different sprites to act similarly when collisions occur.

BOUNCES determines whether the sprite bounces when it hits an obstruction. The extent of bouncing will be very small unless the movement type is 1 (SIMPLE). (More complex effects than bouncing are controlled via the main menu COLLISION options.) Set to NO the sprite will not bounce off other sprites, and it will be blocked by BLOCK obstructions without bouncing, although it will continue to move in unblocked directions if it has any part of its motion towards them. If the block is removed, the motion will normally restart. Set to YES the sprite will bounce off sprites and obstructions but not slow down.

HALTS ON IMPACT set to YES will over-ride BOUNCES and stop the sprite from "trying" to get anywhere after it hits something. Suitable for custard pie sprites and so on! The sprite can still be moved by external forces, such as pushing by another sprite, or gravity.

FEELS GRAVITY set to YES means that the sprite will tend to fall unless it is supported by something, or unless it does not require support. The exact effects are determined by the force of gravity and maximum falling velocity, which can be altered using the main menu EDIT GM DETAILS option.

NEEDS SUPPORT means that a sprite cannot move upwards unless it is supported by something, at least initially. It should be YES for a sprite which is to leap. FEELS GRAVITY should also be YES. Y SPEED should be quite high - perhaps 3 or 4 - since this determines the initial upwards speed. The force of gravity and the maximum falling speed also affect the form of a leap. NEEDS SUPPORT being NO and FEELS GRAVITY being YES is suitable for a spaceship you want to be able to fly up and down, but to fall if you stop pressing the UP key. (See the "ships.d" demo on your disk.) FEELS GRAVITY set to NO would give a spaceship that simply stopped when you stopped pressing the UP key.

EDGE LIMITED should be set to YES if you want a sprite to be unable to leave the "game frame", or other enclosing blocks. The game frame is normally set to include the entire screen by use of an enclosing BLOCK (see the EDIT BLOCKS option) so edge-limited sprites will be stopped by the screen edge. When EDGE LIMITED is set to NO, a sprite can wander off the screen. For more about how this works, see EDIT GM DETAILS (Delayed Wrap) and the section on the Coordinate System in the explanation of the GMC Language.

LEFT/RIGHT MIRROR can be set to YES if you want a sprite to be mirrored left-to-right when its sideways motion reverses. Used by itself, this physically mirrors all the frames for the sprite. This has some advantages: it is simple, and there is no extra memory usage. One disadvantage is that it is relatively slow, which may give a perceptible pause in the game, especially with a large sprite with lots of frames. Another is that ALL copies of the sprite on the screen will be mirrored, even if they haven't reversed direction, because they all use the same graphic data. This will be irrelevant if you only expect to use one copy of the sprite, of course.

There is another way of dealing with motion reversal, in any case. Leave MIRROR set to YES, but set non-zero values for **LEFT MODULE** and **RIGHT MODULE**. The left module will be executed when the sprite reverses and starts to go left, and it can select a new animation sequence for the sprite, in which all the frames are already mirror-reversed. The right module can re-select the original animation sequence. This is much faster than the original method, and can be used for multiple copies of a sprite without problems, although it uses double the amount of memory, and takes a little effort with the graphics editor, animation sequence editor, and module editor. See ANIM in the Games Master Control Language section for details of how to change an animation sequence within a module. See "demo1.d" on your disk for a mouse and a witch that use this method.

UP/DOWN FLIP is very similar to MIRROR. YES alone will invert the actual graphics data, whereas assigning **UP** and **DOWN** modules can do the same thing using new animation sequences.

STOP MODULE, when non-zero, is the module executed when the sprite's normal animation comes to a stop because it has stopped moving and CONDITIONAL is set to YES. A common application is to change the sprite's animation from walking to standing, or to change frames, using the ANIM or SPOKE commands.

KEY 5-8 MODULES These are only relevant to player-controlled sprites. The main menu EDIT KEYS option allows you to define up to 8 keys or key combinations. The first four of these keys are assigned to move a sprite left, right, up and down, but no actions are assigned to the other four unless you set a non-zero value for the relevant key module. For example, suppose you want to fire a missile and make a sound when the space bar is pressed; you could assign SPACE to be the fifth defined key using the EDIT KEYS option (in fact this is already done), set the KEY 5 MODULE to be, say, 20, and then make module 20 contain commands to EMIT a missile and create a SOUND, using the EDIT MODULES option. Different player-controlled sprites can have a different set of key modules, corresponding to perhaps different level ships with new weaponry. See "ships.u" on the disk.

MISSILE If this is set to YES, the sprite "dies" if it goes completely off screen. This is usually what you want with a missile - otherwise it may buzz around your screen forever. It may still do so, if it is edge-limited and bounces, even if MISSILE is set to YES.

You can "fire" any sprite with the EMIT command, whether MISSILE is set to YES or NO.

UNDER FIRER is usually set to NO, meaning that if this sprite is "fired" using the EMIT command, from a position that overlaps that of the "firer", it will be OVER the firer. If it is set to YES, it will be UNDER the firer. Often you won't care about this, because the initial positions will not overlap, but it lets you fire e.g. missiles from behind something, or not. (Collision detection between missile and firer will not occur until AFTER the two have been "not collided" i.e. the missile is clear of the firer - but you could still be hit by a ricochet!)

ABSOLUTE SPEED when set to NO means that if this sprite is fired using the EMIT command it will add the speed of the firing sprite to its own intrinsic speed (set by X SPEED and Y SPEED). This setting is closest to reality for many sprites. A setting of YES means that the sprite will use only its own speed, and might be more appropriate in some cases - e.g. the bubbles emitted by the fish in "fish.d". Try altering the ABSOLUTE SPEED setting for the "shot" sprite in the "meteor.d" game and study the effects.

MASKLESS is normally NO, meaning that the sprite has a mask and can be used normally. If set to YES, the "sprite" isn't really a sprite at all, since it must be used as a rectangular background. This property is specified when the sprite is created, and cannot be altered.

MEMORY USED can only be changed from this option by altering the state of PIXEL X SPEED. It is the number of bytes used by the sprite, and includes the graphics and a 45-byte data area.

EDIT GRAPHICS Option

You can edit the graphics of any sprite by selecting it using the PICK SPRITE option, followed by the EDIT GRAPHICS option. If the sprite already exists, you will be able to display each frame and edit the desired one by pressing F8. If the sprite is a new one (i.e. you used PICK SPRITE to point to an empty entry in the sprite list, and then typed in a name) you will be prompted for the width and length of the new sprite, in pixels. Once you have approved the settings, you will be ready to edit the first (and so far only) frame of the new sprite.

You cannot change the sprite's dimensions after this without using the CLEAR SUBMENU option on the main menu to erase the sprite and start from scratch. (Although you could PUT the sprite in a safe place first and then GRAB the graphics back - see GRAB FRAME option.) An edited frame can be:

i. Abandoned without the original being altered.
ii. Used as a replacement for the original frame.
iii. Used as a replacement for any other frame of the sprite.
iv. Used as a new additional frame for the sprite by adding it to the end of the sequence of frames.

Assuming you are now ready to edit a new or existing frame, you have quite a few options:

F7 - USE the frame as it is now. Before final approval, masking operations (see later) will be carried out and the sprite will be placed against both a colour 0 and a colour 15 background for you to look at. If you press any key except "N" when prompted, the edited frame will replace the original in the sprite's data. If you press "N" you will have the option to use the edited frame to replace ANY frame for that sprite, or if you use the maximum allowable frame number (which is displayed for you) the frame will be added to the end of sprites frames, making the number of frames increase by one. Using a frame number of 0 will switch to a second screen, where a box cursor the same size as the sprite can be moved around using the cursor keys (plus SHIFT for speed). F7 will place the frame at the current position, F9 will abandon.

F9 - ABANDON any changes that have been made. Both F9 and F7 ask if you want to edit another frame. Any key except "N" or F9 will be taken as "YES".

SPACE - PLOT the pixel pointed to by the arrow cursor in the current colour - this is pointed to by a separate "colour cursor" below the palette display. This cursor can be moved left or right using the F1 and F3 keys, which means that you can alter the current colour without "losing your place" as you might if you had to move the main arrow cursor. The pixel will be plotted on the enlarged view of the sprite on the left, and provided there is room, on a normal-sized view on the right. The enlarged view uses magnifications of 2, 3 or 4 times, according to what will fit on the screen. Very long sprites may partially obscure the list of options at the bottom of the screen.

The other options are:

F4 - MIRROR sprite horizontally. This is useful for creating frames of the sprite moving in the opposite horizontal direction.

F5 - FLIP sprite vertically. Use it for e.g. making spaceships look correct while flying down the screen instead of up.

F0 - TURN sprite clockwise by 90 degrees. Only possible if the sprite is square. The operation can be repeated to give frames rotated by 180 or 270 degrees compared with the original.

SHIFT and CURSOR KEYS rolls the sprite left, right, up or down by one pixel (or more, if you keep pressing the key). No information is lost, so you can reverse the process. Useful for moving frames that you GRABed a bit wrong, amongst other things.

F6 - SWAP colour pointed to by the arrow cursor with the colour pointed to by the colour cursor, for the whole frame. The colour cursor will automatically shift so that if you press F6 again, the colours will swap back to the original state. This function is very useful when editing sprites from different sources so that they all look good with the same palette. You will often find that you need to change the way that colours are used. For example, a pinkish colour may be part of the palette you have decided to use, but the sprite you are editing has a green face and pink boots when you use that palette. If you cannot redesign the palette without causing problems for other sprites, you need to edit the selected sprite, so you point the arrow cursor at the green face, and the colour cursor at the pink colour in the palette, press F6 and the face (and anything else using the same colour) will become pink and the boots (and anything else using the same colour) will become green. However, working out the swaps needed to get all the colours right can be tedious, and the use of F6 is best avoided unless all 16 colours are in use in the sprite you are editing - F2 below will be simpler.

F2 - SET colour pointed to by the arrow pointer to be the colour pointed to by the colour cursor, for the whole frame. Note that this option may reduce the colours used by a sprite by making areas that were originally different colours indistinguishable. Once you have told the program to USE the frame, you may be unable to reverse any changes.

DECIMAL POINT - FLASH colour pointed to by the colour cursor. Useful for checking which, if any, pixels in the sprite have a particular colour. (This may not be obvious with some palettes!)

F8 - FILL area pointed to by the arrow cursor with the colour pointed to by the colour cursor. Like the normal Basic FILL - only connected areas will be FILLed.

ADDING A FRAME

There are two main methods of adding a frame to an existing sprite. If you want the new frame to bear some resemblance to the existing frames, the best method is to press F7 to Use an existing frame, then reply "N" to "Use as frame x?", "Y" to "PUT, or use as another frame?" and then enter the frame number given as Max. This will add the frame to the end of the existing frame

list. If you want to clear the frame before drawing anything, one way is to edit it, move the colour cursor to the extreme left, and use F2 to set each colour in use to zero.

Alternatively, you could use the GRAB FRAME option to grab the next frame from the second screen. This might have been loaded from disk, or had frames placed on it, or been left as it was at the end of a game.

SELECT MASKING Option

Any sprite handling program that uses other than rectangular sprites needs two kinds of graphics information: 1. What do the graphic frames look like? 2. Which bits of them are to be actually put on the screen? If the entire frame is put on the screen, the sprite will have a border that will over-write nearby backgrounds and other sprites. The information about which parts of the frame are to be used, and which are to be "masked off", is called a "mask". This information can be prepared in different ways according to the application. The various options are shown on the Mask Menu, and can be chosen by moving the cursor to them and pressing RETURN. None of them have any immediate effect - you must use EDIT GRAPHICS to select a frame, and then select USE, for the new mask to be created using the method currently selected. The options are:

SURROUND

This is the initial setting. The program assumes that the colour in the top left-hand corner of the frame is the background colour, and any part of the sprite's border that is this colour is masked off. This is usually satisfactory. Any internal details on the sprite will be left alone, even if they are the same colour as the background. The actual background colour is irrelevant.

BLACK

This option assumes anything in colour 0 (usually black) is background and should be masked off. This works when SURROUND will not - for example, if the top left hand corner of the frame is used by the sprite. It can also be used to create a mask (and sprite) with holes in it; for example the "trellis" sprite on the disk has internal holes which we expect to see the background through, and BLACK masking had to be used to create it.

SURROUND+1

This option is like SURROUND, but a 1-pixel border of the background colour is added to the edge of the sprite data used (if the frame size allows). This can help a sprite to show up against a similar-coloured background by giving it a contrasting outline.

BLACK+1

Like BLACK, but adding a 1-pixel border as above.

MANUAL 1

There may be some occasions when the options above are not exactly what you want. This option overlays graphics edited with the EDIT GRAPHICS option with a chequered pattern that shows the current mask. You can edit this manually by selecting the chequered pattern at the right-hand end of the colour palette, and plotting any extra mask pixels where you want them. Mask pixels can also be over-written by using one of the normal colours. You will find it best to create masks with one of the other options, and then use MANUAL 1 just to touch things up a little.

MANUAL 2

This option was added late-on, when I realised that masks that were partially normal and partially translucent could be used to create interesting effects, such as shadows. It allows you to edit, not the sprite's graphics, but the actual mask data, just as though it were graphic data. A normal mask will appear as colour 15 (usually white) on colour 0 (usually black). Masking works at the binary level - 15 is 1111 in binary, and means that when the sprite is put on the screen, all four bits that make up a pixel in the sprite frame will replace the bits on the screen. On the other hand, 0 is 0000 in binary, and this means that none of the bits in the sprite frame are used, and the screen data is left alone.

Now, what happens if the mask is partly colour 8? This is 1000 in binary, and means that a single bit ("worth" 8) from the sprite frame will be used at that pixel, mixed with 3 bits from the screen. Let's assume the background screen uses mostly colours 8, 9 and 10 which look like grey (light black!) light blue and light red. Assume also that part of the mask has been edited to colour 8, and the rest is colours 0 and 15. The part of the sprite's graphics equivalent to the colour 8 bit of the mask should be some colour between 1 and 7, so that its binary number has a leading zero - e.g. 0001. The colour 8 mask area will use this zero, mixed with the rightmost 3 bits of the background colours (which are 1000, 1001 and 1010 in binary) to give 0000, 0001 and 0010 at those pixels (colours 0, 1 and 2). These colours could be black, dark blue and dark red, meaning that that area of background is darker than before, but still visible. In other words, shadowed! By appropriate use of mask and sprite colours it is possible to get effects like stained-glass windows or coloured, translucent bubbles. The demo programs on the disk show the use of shadows.

The mask data is used in collision detection, and only areas in colour 15 count - so a sprite that is entirely shadow will not collide with another sprite, and sprites will not bounce off each others shadows.

When you use EDIT GRAPHICS in this mode, the pixels you plot on the mask will also be plotted on the smaller view of the sprite at the right-hand side. This helps show you where you are in relation to graphic data, but has no other effects.

FROZEN

After you have laboriously hand-prepared masks with the MANUAL 2 option above, you may want to edit the graphics data again - and to do this you have to leave MANUAL 2 mode. Beware! If you go back to SURROUND, as soon as you edit a graphic and USE it, the SURROUND mask will replace your manual mask. You may want to set masking to FROZEN - this will prevent any alterations to the current masks while you edit graphics.

EDIT PALETTE Option

Each game has sixteen pre-defined palettes that can be selected within a module using the PAL command. The initial settings of all but palette 0 are the standard ones for the Coupe. Palette 0 is entirely black, and is useful for blanking the screen while graphics are set up. The EDIT PALETTE option allows you to edit any of these palettes. The display shows the 128 possible colours in the upper part of the screen, grouped roughly according to colour. The lower part of the screen shows the 16 colours of the selected palette - 0 to 7 in the first row, 8 to 15 in the row below. The palette number follows a hash sign; next to this is a rectangle which will display the current colour.

A box cursor can be moved with the cursor keys to point to one of the 128 colours, and RETURN will make that colour the current colour. The colour number of this colour may be of interest - it is shown next to the current colour rectangle. If you move the box cursor over one of the colours at the bottom of the screen and press RETURN, the current colour will be assigned to that part of the current palette.

You will often want to see what effect altering the palette has on the appearance of your sprites. Holding down the decimal point key will display the second screen using the current palette. This screen will show the game screen in the state you left the game in, or the last loaded screen, or the screen as you left it after PUT with EDIT GRAPHICS, or after GRAB FRAME. If your box cursor is over one of the 16 colours of the palette, that colour will flash on the second screen, showing you where that colour is used.

Pressing F7 confirms that you want to use the palette as it is now, and EDIT GRAPHICS and other options will use this palette to display graphics. The F9 key abandons any changes. Both keys return you to the main menu.

GRAB FRAME Option

This option is used for capturing sprites or backgrounds from (usually) a loaded screen. If there is currently an existing selected sprite, GRAB FRAME assumes you want to add frames to it. If you don't, select a free entry in the sprite list with PICK SPR and enter a name using F8. If you forget to do this, the GRAB FRAME option will give you the opportunity to select another sprite, or provide a name for the selected sprite using F8, before the main GRAB option is offered.

If you want to add frames to an existing sprite, you may get the frame dimensions wrong at your first attempt, but this doesn't matter, because the box cursor will be automatically reset to the right size to match existing frames, and you can try again.

The initial display tells you which keys to use to move and change the size of a box cursor. SHIFT plus the cursor keys moves the box by its own width or height, which is often useful when picking up the next frame from the screen. You can input the box size directly, after pressing F8, as well as by eye using function keys. Using F8 is a good idea if you want to grab a strip of screen the full width of the screen. You can do this in one go - but the frame will be too large to edit later, so it is better to deal with this in smaller chunks. You can input a width of 64 pixels, and any old height, then grab 4 frames from the screen by moving left as far as possible, setting the box height by eye, grabbing the frame, using shift+right to move across 64 pixels, grabbing the frame, etc. Four times 64 comes to 256 so the 4 frames make up a complete screen width that can be used within a game as a background.

Input of the box size by pressing F8 also re-selects the "crib sheet" of what key does what, which you might want to re-read!

Pressing F7 will grab the frame. If this is the first frame of the selected sprite, you will be asked CREATE MASKS? If you press any key except "N", masks will be created and the graphic will be a normal sprite. If you press "N", the graphic cannot be used as a real moving sprite, only as a background, although the program and Editor will handle it as a sprite in most other ways. However, the "sprite" will take up half the memory it would otherwise. All subsequent frames will use the CREATE MASKS setting of the first frame.

Next you will have the chance to accept or reject the frame for use as any valid frame of the sprite you are building up. The default frame will add the new frame after any existing ones.

ANIMATE SPRITE Option

This gives a quick, fairly crude way of seeing how your frames are going to look when animated. You can cycle through all the frames from first to last, repeatedly, or alternately reversing direction. (As you might want for e.g. a wing rising and falling - the "rising" frames are just shown in reverse to depict "falling".) The animation speed can also be increased or decreased. However, for more flexible control use EDIT SPR DETAILS to assign an animation sequence to the sprite, if needed, EDIT ANIM SEQ to set up a suitable sequence, and include the sprite in the current game so you can see it animating for real.

SPRITE vs. SPRITE COLLISIONS Option

This option displays part of a 32 row by 32 column table which determines what action is taken for each possible sprite vs. sprite type collision. The rows are named as well as numbered; the names come from the sprite list which can be examined using the PICK SPRITE option on the main menu. The columns could have been named in identical fashion, but there wasn't room. The naming assumes that the collision type number is the same as the sprite number, which is true initially for sprites 1-32. (See the EDIT SPR DETAILS option.) However, the names are simply intended as an aid to memory, and should be ignored for sprites 33-96. The type number is what matters.

If a table entry is zero, no special collision actions take place beyond those (such as bouncing) implicit in the sprite details. If an entry is non-zero, it is the module number that will be executed when a collision occurs between the row-numbered sprite and the column-numbered sprite. The row-numbered sprite is considered to be the "colliding" or "current" sprite as far as the module is concerned, and the column-numbered sprite is considered to be the "hit" or "other" sprite. This is a bit arbitrary, since all collisions between sprites are in fact detected twice, once as e.g. shot vs. ship and once as ship vs. shot, so both take a turn as "colliding" and "hit". Many effects can be handled in two ways, by assigning actions to either one of the two collision entries in the table - e.g. 2 vs. 5 or 5 vs. 2.

The cursor bar can be moved to any position in the table; the screen will scroll as needed. A module number can then be typed in - there is no need to press RETURN when the number is complete, just press any non-numeric key. If you press F8 with the cursor over a module number, that module will appear for editing or inspection. Exiting the module editor with F7 or F9 returns you to the collision editor.

An example: If sprite 2 is a missile, and sprites 4 and 5 are different types of ship which should explode when hit, and sprite 10 is an explosion, you could enter the number of a free module - say, 30 - at row 2, column 4 and column 5. Module 30 might contain:

```
SOUND 5
TRANSFORM 255,10,0,0
```

This would make a noise and transform the "other" sprite (the ships) to sprite number 10 at the same coordinates. Sprite 10 can have an animation sequence that ends with the sprite vanishing.

When you have finished defining collisions, press F9 to return to the main menu.

SPRITE vs. BLOCK COLLISIONS Option

This option is similar to the previous one, but the 32 row by 32 column table deals with collisions between the 32 possible sprite types and block types. The columns correspond to the block TYPES, not block numbers, and they are not related to sprite type numbers. You can have many blocks of the same type while requiring only one table entry to handle collisions of a particular sprite type with such blocks. For example, if you want sprite type 10 to make a sound when it hits block type 2, enter a module number on row 10, column 2. The module with that number would contain a SOUND command. You could have many blocks of type 2. If you also wanted sprite types 11 and 15 to make the same sound on hitting blocks of type 2, you would enter the same module number in row 11, column 2 and row 15, column 2.

The sprite that collides with the block is the "current sprite" (see Introduction to GMCL) and there is no "other" sprite.

RUN GAME Option

The current game is run. First, in Basic, a CLS is done and the current blocks are outlined if outlining is selected (see EDITOR EXTRAS option). Then the game proper is run by LET e=USR start. (The variable "start" holds the start address of the game code, and E carries out an error code if the game is terminated by an error.) Any sprites or sounds that are in use are terminated, the variables A to Z are zeroed, and module 1 is executed. Module 1 may call or jump to other modules. When these modules finish, the computer carries on handling any sprites that have been brought into use. F9 can be used to return to the main menu, or ESC can stop the game and simultaneously break into the Editor. Exit using F9 is recommended. The game screen is preserved on SCREEN 2 and can be inspected using the GRAB FRAME, EDIT PALETTES or SAVE SCREEN options.

When an error occurs in the game, the error message includes the module number where the error occurred.

OTHER WAYS TO RUN A GAME

It is possible to run a game direct from the Basic command line by typing:

```
CALL start
```

The variable START will hold the required address. The normal CLS done by the RUN GAME option will be omitted, and you can have sprites moving about over a program listing by e.g. LIST TO 100: CALL start.

You could also type:

```
RUNG
```

This acts like the Editor's RUN GAME option.

To restart the Editor, you can GO TO main. Alternatively, just RUN; this will reset some variables such as cursor bar positions, masking method, outline y/n etc. to their start-up values, but the game data will be intact. Unless you have Master Basic, there will be a perceptible delay as the program re-initialises itself after RUN.

GAME PAUSE KEY

Pressing TAB during a game halts the program and sets the border flickering. Useful when someone rings the doorbell or the phone rings, or you can't think what to do next. Press the key again to restart the game.

EDIT MODULES Option

First you are asked which module (a section of Games Master Control Language) you want to edit. The default is the module you edited last.

Modules can fill no more than one editing screen, but you can have up to 128 of them. The EDIT MODULES option shows the current module number after a hash sign. No line numbers are needed, or allowed - simply type commands, each on a separate line. Spaces are not significant, and upper and lower case letters are equivalent. New text can be inserted within a line by moving the cursor to the required location and typing the text. Text can be deleted using the DELETE key. The line that the cursor is on can be deleted by pressing F5. A new line is inserted at the line the cursor is on by pressing F2. (Any text scrolled off the bottom of the screen will be lost.)

See the section on the Games Master Control Language for details of the commands and functions you can use. For now, you can experiment with the editor by typing text preceded by REM.

You can Abandon any changes you have made by pressing F9, or COMPILE the module by pressing F7. During a Compile operation, the module text is converted into an internal format that can be handled very quickly. If a syntax error is detected during this process - for example, if a command is not recognised, or it has the wrong number of parameters after it - then an inverse question mark will appear after the error or errors, and a beep will sound. Press any key to resumed editing. Your cursor will be located after the last error automatically.

If you have used NEXT without a matching FOR, or a GOTO without a matching LABEL, you will get a "Missing FOR or LABEL" error message, and the module will not compile.

After Compile or Abandon, you can edit the previous module, edit the next module or LLIST the current module. You can also save the module to disk, or overwrite the existing module by loading a module from disk. Module file names have ".m" automatically added. Pressing anything other than the keys specified will return you to the main menu.

The EDITOR EXTRAS Option allows you to select 64 columns for editing, instead of the usual 32. This may be needed for entering commands with parameters that are complex expressions, since each command must fit on a single line.

EDIT GAME DETAILS Option

This option allows you to alter several important properties of the overall game. You can alter the same things from within a module using the VPOKE command.

MINIMUM GAME DELAY

This is a method of setting a maximum speed for a game. If the game involves your ship starting off with 10 largish enemies and no other sprites on the screen, by the time you have disposed of all the enemies, the game may have speeded up to an unnerving extent, because the computer will have less to do. By increasing the Minimum Game Delay the speed-up is limited. If the game is slow enough as it is, the Game Delay has no effect. The normal value is 1, which means that the computer takes at least one 50th of a second to handle all the sprites, even if it can do it quicker. Values of 0-5 might be reasonable for a game. Higher values - e.g. 15 slow things down a lot, and can be useful in debugging a game.

FORCE OF GRAVITY

This variable controls how quickly falling sprites (with the AFFECTED BY GRAVITY property - see EDIT SPR DETAILS option) accelerate. Higher values make sprites attain the maximum falling velocity sooner. If the value is negative, objects fall upwards!

MAXIMUM FALLING SPEED

This variable sets the maximum Y speed that a falling sprite can reach, in pixels per move.

BORDER COLOUR

The initial border colour of the game. Should be 0-15.

IMMEDIATE WRAP

Normally NO, which means that sprites can exist in an invisible off-screen "phantom zone" to the left and right of the screen. If set to YES, sprites wandering off-screen to right or left immediately wrap round to the other side, rather than entering the "phantom zone". See THE COORDINATE SYSTEM for more details.

EVERY CYCLE MODULE

Used to select a particular module which will be executed every time the entire set of sprites has been moved or checked (a "cycle"). Any non-zero value is the module number. In order to show different demos for a particular length of time, such a module might look like this:

```
LET T=T+1
IF T=100: JPMOD 10
IF T=200: JPMOD 11
IF T<>300: RETURN
LET T=0
SCLEAR
JPMOD 1
```

You would have to avoid using the variable T in any other module.

The Every Cycle Module is very useful for generating new sprites every now and then - for example, the meteors in the "meteor.d" game on your disk, or the bubbles in "fish.d".

The current Every Cycle Module can be modified within a program using the VPOKE command.

See also the Introduction to GMCL.

ROM INTERRUPTS

Normally NO, meaning that Games Master uses its own system of interrupts, 100 times per second, rather than the 50 times per second ROM interrupts. This has implications for sound and PALETTE LINE colour changes. See the EDIT SOUNDS option for more about the implications of this.

ALLOW EXIT FROM GAME

Normally YES, meaning that pressing ESC or F9, or errors such as "Missing sprite" will terminate a game. You may wish to set it to NO once you have finished developing a game. In this state program errors may cause odd behaviour or a crash. ESC and F9 are ignored, so you cannot return to the Editor to change back to YES, or to change anything else about the game! So always keep at least one copy of the game which DOES allow EXIT.

TOP EDGE

This specifies the coordinate above which sprites will trimmed. It is normally 191, so that sprites normally vanish as they move off the top of the screen. If TOP EDGE is reduced, for example to 182, then sprites moving upwards will vanish before they hit the screen top, leaving the top 9 pixel rows of the screen untouched. You could place text or backgrounds here, since TEXT, BFILL, BACK, etc. are not limited by the top edge value. This is useful for a game "frame" or score and lives display area.

BOTTOM EDGE

This specifies the coordinate below which sprites will be trimmed. Normally it is 0, but it can be increased to protect the bottom part of the screen from being crossed by moving sprites.

When you have edited the game variables, F7 accepts any changes and F9 abandons them.

EDIT PATHS Option

This option lets you define up to 32 complex paths for sprites to follow. To make a particular sprite follow one of these paths, you must use the EDIT SPR DETAILS option on the main menu to set its movement type to 3 (PATH) or 4 (ALT PATH) and set its PATH value to the desired path number.

To use the EDIT PATHS option, enter a path number. You will then be asked whether to use pixel x moves or not; the default is NO. Sprites can move horizontally by bytes (2 pixels) or by pixels - see the EDIT SPR DETAILS option. The amount of horizontal movement seen on the screen during a game when a sprite follows a particular path depends on this property. You can design paths with the X moves setting in either state, so that what you see matches the action of sprites with the same setting. Sprites with a different pixel x move setting will still be able to follow the path, but their movement patterns will be squeezed or stretched horizontally.

The current shape of the specified path will now be shown. If the path is undefined, you will see just a dot in the middle of the screen. If you press F8 you can re-design the path. At this point use the F1 and F3 keys to select a step size for the path. The smaller the step, the smoother and slower the motion of the sprite. Step size can be changed at any time within the path, to give varying speeds at different points - see the paths in the "fish.d" demo on your disk. You can also press F8 to enter a module number at any point; this module will be executed at this point in the path, and can cause e.g. a missile to be fired.

Move the arrow cursor to where you want the path to start from. This isn't critical, since paths are relative to a sprite's starting position in the game, but you need to allow enough room to draw the path on-screen. Now press RETURN and a dot will be plotted. Move the cursor to a new position and press RETURN again to draw a line of dots from the previous position. These show potential sprite positions. Keep doing this until the path is almost complete. You can finish in two ways. Pressing F7 will close the path into a loop by drawing a line to the initial position, and there will be no cumulative movement as a sprite repeats the path. Alternatively, press F9 and the path will terminate, but not close. Sprites following it will make some overall movement.

When you have ended the path definition, the path will be redrawn using colour cycling to show direction of movement. If it is not a closed path you will have the chance to repeat the path to see what the overall movement effects will be. The screen scrolls if needed. You can then approve the path, enter a new one, or abandon any changes.

EDIT SOUNDS Option

This option allows you to edit or define up to 32 sounds that can be produced from within a module using a simple SOUND (number) command. Sounds can be loaded from disk if they have been previously saved. The display for each sound shows bar graphs for volume and pitch, and status "buttons" for tone and noise status. In the volume graph at the top of the screen, any bar can have a height of 0 to 15 units, which determines the loudness of the sound during that time interval. In the pitch graph in the middle, the sound frequency can be set much more precisely. To change the height of a bar, place the cursor at the desired new height and press SPACE. There can be up to 60 bars in each graph. The sound you hear after pressing F7 will follow the pattern of the two graphs; it will finish when the data in both graphs has finished. You can have intervals of zero volume if you like.

The lower part of the display shows "buttons" for each bar. The rows the buttons can occur in are marked T for Tone, N for Noise, H for High, M for Medium, L for Low and V for Variable. A button can be turned ON by placing the cursor in the right position and pressing SPACE. When a cyan Tone button is ON (visible) then tone is selected for that bar, and the sound will be at the pitch shown in the pitch graph above. If a green Noise button is ON then noise (hissy sound) is emitted. Its pitch depends on the position of the magenta button that will appear below. This can be in the High, Medium, Low or Variable position; the first three settings are independent of the pitch graph, but the Variable setting uses the pitch value in the bar above. Both Tone and High, Medium or Low pitched Noise can be ON at the same time - a mixture of pure and noisy sounds will be heard. Tone will not be effective while Variable pitched Noise is in use.

Limitations within the sound chip mean that no more than two sounds can use different-pitched noise simultaneously, but there is no such limitation with tone. The Games Master software does its best to allocate sounds to whatever sound chip channel is appropriate, and available, so that up to six sounds can be produced at the same time. If all six channels are in use, any further SOUND commands will overwrite earlier ones before they have finished.

The time interval denoted by a single bar is either a 50th of a second or a 100th of a second, according to whether ROM interrupts are set to YES or NO using the EDIT GAME DETAILS main menu option. The slower 50th of a second rate with YES allows less exact shaping of sounds, particularly short ones, but the maximum sound duration is doubled. This mode also works with PALETTE LINE to give more than 16 colours on the screen, and with Sound Machine or Master Basic to give background music. (Load the Basic program "musfish" for an example.) Both of the latter interfere with the production of sound effects, however. It is up to you which interrupt setting you want to use.

You can Abandon an edited sound, which will leave the original version (if any) of the sound intact. Alternatively, you can compile it - this will turn the graphs into an internal format for storage. You can then hear the sound repeatedly by pressing a non-function key, such as RETURN. If you do not like the sound, pressing F8 will let you edit it again. Pressing F7 will accept

the sound for use, and it will replace the old version permanently. You can also SAVE a complex sound as a disk file by pressing F4. This allows you to re-use the sound in another game. Even at this stage, you can also Abandon a sound, leaving the original version intact.

Disk sound files have ".n" added to the end of the name automatically. You do not need to include ".n" when saving or loading these files.

EDIT KEYS Option

This option shows you a list of eight keys or key combinations that will be recognised by the game. The first four in the list make player-controlled sprites move left, right, up and down, other factors permitting. They are initially set up to respond to the cursor keys OR the numerals produced by most joysticks. The fifth entry is set up to respond to the space bar OR a joystick fire button. The others are not set up. All keys can be re-defined - enter the key number, or just press RETURN or F9 to exit this option.

The keys are redefined by pressing the key or making the joystick movement you want to use, when prompted. You will have the chance to allow another key to be used as well (like the cursors OR joystick) or require another key to be pressed at the same time - so that it is possible to program e.g. fire+left on a joystick to do something special.

See the EDIT SPRITE PROPERTIES option for details of how to associate particular actions with key entries 5 to 8.

EDIT ANIMATION SEQUENCE Option

This option allows you to view and alter animation sequences which can be used by any sprite by entering the sequence number into the sprite details using the main menu EDIT SPR DETAILS option. Frame numbers should be between 1 and the number of frames in the sprite, TIME should be between 1 and 255, and module will usually be null (just press RETURN). TIME is the number of moves (or possible moves) the frame will be used for. Input "q" to finish. The program, as usual, tries to anticipate what you might want to input and makes that the default, so you can just press RETURN much of the time. Some commonly used animation sequences are already programmed in. Here is a list:

1 Frames 2-8, repeating
2 Frame 1 for time 20, then frames 2-5, alternating
3 Three frames, alternating
4 Four frames, repeating
5 Four frames, alternating
6 Six frames, repeating
7 Six frames, alternating
8 Eight frames, repeating
9 Five frames, repeating

If you look you will see that "Three frames, alternating" isn't frames 1,2,3,3,2,1 as you might guess, but 1,2,3,2/ 1,2,3,2/ 1,2,3,... Each sequence repeats automatically. Sequences that

start at frame 2 are often useful, because the default action when animation stops is to select frame 1, which can be a view of the sprite standing still, rather than e.g. poised with one foot raised. Frame 2 onwards would provide the animation for normal movement.

The TIME value can be different for different frames if you like. The list of frames, times and actions can be very long.

By supplying a module number, you can make a sprite do something, such as make a sound (footsteps, bird chirps, etc) or drop a bomb, at a particular point in its animation sequence. See the EDIT MODULES Option, and the example and demo programs on the disk.

You might even want an animation sequence for a sprite with a single frame. For example, to have a sprite that looks the same all the time but fires a missile every 20 moves:

```
FRAME TIME MODULE
  1    19    -
  1     1    50
```

Module 50 would have to be set up to fire the missile.

When you have finished, you can abandon the new values (the old ones will be used instead) or use them if they are what you want.

EDIT BLOCKS Option

BLOCKs are rectangular areas of the screen that can be used to restrict sprite movement, support sprites, or trigger collision actions. They can also be filled with complex patterns. Blocks come in sets, and the first thing this option asks you is which set you want to edit. There can be up to 255 blocks in a set, but a set normally starts with just a single block defined. This surrounds the screen and defines an outer limit for edge-limited sprites. When you have chosen a set, the EDIT BLOCKS option shows all existing blocks outlined in yellow, with the number of the block in the top left hand corner. You can then select any existing block to modify, or enter the next block number to create a new block.

When you choose a block, you are given a box cursor that can be moved with the cursor keys. The box height can be changed using F2/F5 and its width using F1/F3. Using SHIFT plus the cursors moves the box faster. (To be specific, by one box width or one box height per move.) When you are happy with the block size and position, press F7. You will then see a display of which planes the block exists on, and other information. The plane settings determine which sprites can collide with the block; if ON PLANE 4 is YES, and the rest are NO, then only sprites on plane 4 can detect collisions with the block and bounce off it or respond to it in other ways.

A block can exist on multiple planes. If it exists on none, the block no longer exists - you can use this to delete a block. A typical application of the plane settings might be an aerial view

of a city, with the buildings defined as blocks on plane 1 only. Car sprites could wander the streets, bouncing off the buildings, while aircraft passed overhead unobstructed - unless perhaps you made taller buildings exist on higher planes as well.

Whether the block is ENCLOSING or not is very important - if the setting is NO, the block detects collisions with its inside, and it can keep sprites OUT, act as a moving belt, and be a supporting or a non-supporting block. If ENCLOSING is set to YES, the block detects collisions when sprites try to leave it; it keeps sprites IN, like the screen-edge block. It cannot act as a belt, and its bottom surface is always supporting.

The TYPE number is used in Sprite vs. Block collision detection in order to trigger particular modules when blocks of a particular type are hit. See the SPR v BLK COLLIS Option for details. The block type can be altered in the range 1-32. Block types can have certain properties, and these too can be altered, but this will affect the properties of ALL blocks with that type number. Block types can be SUPPORTING, which means that sprites can stand on them, and that sprites cannot penetrate inside them (always provided that they exist on the same plane as the block). A block which is not supporting might be used just for collision detection - touching it might trigger a door to open, a score to change, etc. via a module. See the example programs on the disk. Blocks can also be BELTS, which move right (positive belt speeds) or left (negative belt speeds); anything touching the upper surface will be moved sideways - see e.g. "platform.d" file.

Blocks can also be filled by repeats of any sprite or background using the BFILL command within a module, and you might want to define one just for that purpose.

The GMCL command BLOCKSET can be used to select a block set. BLOCK can be used to alter a block in the current set, and can even set up a block which is partly off-screen. (See The Coordinate System for an explanation of the "phantom zone".) BTYPE can alter the properties of a given block type. See these commands for details.

The EDITOR EXTRAS option (see below) normally has OUTLINE set to YES so that the blocks in the current set (the last edited set) are drawn on the screen before a game starts, provided it is run from the Editor. This is useful for debugging purposes - if OUTLINE is set to NO, blocks are invisible until you perform extra actions, such as filling them using the BFILL command, or by using BACK to place a section of background in the same place, from within a module. The first module should avoid using the CLS you might otherwise use.

The OUTER edge of the drawn outlines show the edge of each block, whether it is an enclosing block or not. Outlines cannot be drawn if the game is run as stand-alone code without the Editor resident. (You could chop out and adapt the Editor procedure Outline if you wanted to.)

EDITOR EXTRAS Option

This option controls some Editor facilities. Move the cursor bar to the desired position and press RETURN to reverse a setting or execute the desired operation. Alternatively, press F9 to abandon the option.

You can select YES or NO for block outlining. (See the EDIT BLOCKS Option above.)

The EDIT MODULES option can use 32 or 64 columns.

LIST VARS will show the current values of the GMCL variables A to Z, in two pages. The associated Limit and Step values are only relevant for FOR-NEXT variables. Press any key for the next page, or to return to the EXTRAS menu.

LIST MODS will list all modules with text in them. Press any key when the listing finishes, to return to the EXTRAS menu.

LLIST MODS is similar to LIST MODS, but sends output to a printer via stream 3.

Press F9 to return to the main menu.

LOAD SPRITE Option

This option allows you to load sprite files containing graphic and other information from disk. You are first given the opportunity to select a free sprite entry in the sprite list by pointing to it with the bar cursor and pressing RETURN. A directory of files ending in ".s" for Sprite is then shown and you are prompted for a file name. Just press RETURN to abandon the LOAD. Otherwise, the file name you supply automatically has ".s" added to it and the sprite file is loaded. (If the file does not exist, you will be returned to the main menu after an error message.) The file name is placed in the sprite list and can be examined with the PICK SPRITE option. The loaded sprite is now the "selected sprite" to which options such as EDIT GRAPHICS and EDIT SPR DETAILS refer by default. The name and number of this sprite is shown on the main menu on the PICK SPR option.

SAVE SPRITE Option

This option allows you to save a sprite to disk. You are prompted for a file name to save the selected sprite under. If you simply press RETURN, the default name (from the sprite list, with ".s" added) will be used, but you can enter another name. The ".s" suffix is added automatically. The sprite details and graphics are saved in a CODE type file.

LOAD GAME Option

This option allows you to load either the data to let the Editor run a game, or a full game including the sprite handling code. Which one you choose makes little difference, since the sprite handling code will already be in memory and will simply be over-written by the saved version when it loads. However, if later versions of the sprite handling code are ever released, loading a GAME DATA version of the file might be preferable so that the latest handling code was used. The GAME DATA option gives a DIR of all files ending in ".d" and automatically adds a ".d" to the file name you supply, if you do not do it yourself. The FULL GAME option acts similarly but uses ".g" instead.

SAVE GAME Option

This option allows you to save either just the data that make up a game, or a full game including the sprite handling code, with the option to auto-run on loading. Which you choose doesn't matter much if you plan to run games under the control of the editor, but the DATA option will allow you to use the same game data with any later versions of the sprite handling code. A FULL GAME file can in any case be converted to a DATA file by loading it and resaving it without the first 9984 bytes - SAVE "name.d" CODE 116480,(length)-9984. The GAME DATA option gives a DIR of all files ending in ".d" and automatically adds a ".d" to the file name you supply, if you do not do it yourself. The FULL GAME option acts similarly but uses ".g". AUTO GAME is identical to FULL GAME apart from saving the game so that it auto-runs on loading.

If you want a game to load and run on a machine with less memory, you should have a look at its Total Game Size using the MEMORY MANAGER option, and reduce the size of the Workspace area if possible, before saving the game.

Stand-alone CODE Games

If you use the SAVE GAME option to save a Full Game, and the game does not use Basic or machine code subroutines, then no other program is needed to run the game; you can simply BOOT the computer and then LOAD "game name" CODE: CALL 106496. This address is the one the code was saved from and the address it will normally load back to. It is &1A000 in hexadecimal, which may be easier to remember. The code will also work correctly at multiples of 16K above or below this address, provided it does not go too low and hit Basic, or too high and hit DOS or the screen memory. For example, you could use: LOAD "game name" CODE 40960: CALL 40960. Here are some possible addresses:

24576 (&6000), 40960 (&A000), 57344 (&E000), 73728 (&12000), 90112 (&16000), 106496 (&1A000), 122880 (&1E000).

The address 24576 will only allow a very small Basic program. Why would you want to run the code at a different address? One reason is to minimize memory use so that a game developed on a 512K SAM works on a 256K machine. Running the code at a lower address without the Editor loaded may allow the code to run. Even on a

512K machine, it might be used to free higher memory pages for use as extra screens or for RAM disks, COPY or BACKUP to use.

If memory is really tight on a 256K machine, you can persuade Games Master to overwrite the DOS by including in a Basic loader:

```
IF PEEK SVAR 450=13 THEN POKE &5100+13,0
```

This means: "If DOS is in the usual page for a 256K machine (13) then mark that page in the page allocation map as free." Basic will probably crash when you exit the game.

If you save a game using the AUTO GAME option, the code runs automatically and can be loaded and run with a simple: LOAD "name" CODE or LOAD (file number). A game which is not auto-running could be converted to an auto-running one, perhaps at a different address, by loading and then re-saving it using e.g.:

```
LOAD "game.g" CODE &a000
SAVE "new name" CODE &a000,leng,&A000
```

"Leng" could be read from a DIR listing or obtained using MasterDOS's FSTAT function.

GAME PROTECTION

All game and data files are automatically scrambled before saving and unscrambled on loading. This has been done to give some measure of protection to games you write with the system. Normally, anyone else using the Games Master Editor will be able to load your games and see how you have written them. You may be pleased to allow this. However, if you want to sell your games, or just hide your secrets, use the EDIT GAME DETAILS option to make it impossible to leave the game (provided it does not use BASIC subroutines). Then use the SAVE GAME option to save an auto-running game. Another user cannot use his or her Editor to examine the game. Even if the game code can be read directly from the disk, it will not have been unscrambled and cannot be used by the Editor.

This system does NOT copy-protect your game in any way, it just makes it harder for another user to see how it works. Since Games Master produces standard CODE files, any techniques you know for protecting such files may be applicable.

LOAD SCREEN Option

This option gives a directory of all SCREEN$ files and of CODE files of roughly the right length to be MODE 3 or 4 screen data. Just press RETURN when prompted to input a file name if you want to abandon the LOAD. After loading a file, any flashing colours are turned off and you are returned to the main menu when you press any key. You can then use GRAB FRAME to read sprite frames from the screen, or you can use EDIT PALETTES to copy the loaded palette into one of the internal game palettes.

SAVE SCREEN Option

This option allows you to save the second screen, which will show the last state of a game, or the last loaded screen, or show sprites that you have PUT there with the EDIT GRAPHICS option, whichever was most recent. The screen which will be saved is shown until you press any key, then you are prompted for a file name. Just press RETURN if you want to abandon the save. A normal SCREEN$ file is saved to disk.

EXIT TO BASIC Option

This simply restores some key definitions to normal and goes to a STOP statement. You can use GO TO main or RUN to restart the Editor. RUN clears the second screen and resets some variables, but the game data is left alone.

MEMORY MANAGER Option

This option allows you to change the size of the two main memory areas in a game. The initial display shows you the current status. The TOTAL GAME SIZE is the amount of computer memory used by the game, and it is always a certain number of 16K pages, plus half a page. This includes a WORKSPACE, which is used for storing copies of sprites and areas of background. Initially this is made as large as possible. As sprites are loaded or defined, the free workspace decreases. If it is too small, you may get an error message when you try to load a sprite or run the game. If it is needlessly large, you may make it impossible to run a game developed on a 512K machine on a 256K machine, and you will tie up RAM that you could have used for e.g. RAM disks. However, this may not be a worry for your application. Workspace size can be increased or decreased by whole 16K pages using the Memory Manager menu.

The other memory area that Games Master uses is called a HEAP. This area holds movement paths, animation sequences, sound definitions, and modules. Its initial size is 4096 bytes. You may run out of space, and the Memory Manager menu allows you to increase or decrease that size without loss of data.

The workspace is not saved with the program, so FILE SIZE will be less than GAME SIZE. File size is the size the game will be when saved on a disk.

The LOAD GRAPHICS setting that is shown tells you if sprites are loaded complete with graphics, or not. The setting can be altered with the Memory Manager menu so that games can be developed without using too much memory. (See below.)

Press F9 in response to the menu if you do not want to alter anything. Press RETURN to choose an option, and either input a number, or press RETURN or F9 to exit the option and return to the main menu.

WHEN YOU ARE SHORT OF MEMORY

This is most likely to occur if you are using a 256K SAM, but could also happen with a large project on a 512K machine. The Editor needs about 80K of RAM while a program is being developed, but it is not needed while the game runs. In contrast, all the graphics are needed for a finished game, but most testing and development can be carried out without all the graphics present, by using the LOAD GRAPHICS option on the Memory Manager menu. This reverses the LOAD GRAPHICs status; if it is NO, then when you load a sprite, only the sprite details are used, and no memory is used for graphics. The sprite will appear as a strange blob on the screen. Many aspects of the game can be tested using these blobs. In practice, only some of your sprites (probably memory-hungry backgrounds) need to be loaded with the LOAD GRAPHICS option set to NO; the others can be normal. Graphic shapes are used in collision detection, so "blobs" may act slightly differently to their normal equivalents in collisions.

When you want to try out a version with full graphics, save a Full Game using the SAVE GAME option, then EXIT the Editor and LOAD "LMG.u" from the Games Master disk. This stands for Load Missing Graphics. LMG is much shorter than the Editor, and this allows a larger version of a game to be created. The program will prompt you for the name of the game file, load it, and then load the graphics for any sprites that do not have any graphics, using the sprite names in the game file's sprite list, plus ".s". You can then save the new version. If LMG reports it is out of memory, you will have to shorten the game somehow.

The CLEAR SUBMENU option on the main menu may allow memory usage to be reduced by erasing unwanted data items.

CLEAR SUBMENU Option

This allows you to CLEAR or erase selected data types. You can clear all the sprites, or just some of the sprites, reset the sprite vs. sprite collision table or the sprite vs. block table, and clear all the paths, animation sequences, sounds, or modules.

THE GAMES MASTER CONTROL LANGUAGE

Introduction

This is the special language used to control some of the Games Master features, and it is known as GMCL for short. MODULES are sections of program in GMCL, which are converted into fast machine code when you press F7 to compile them. (See the EDIT MODULES option.) WHAT? A free Compiler with every copy of Games Master? Can this be true? Sort of. The GMC language is fairly simple, so it isn't nearly as hard to compile as, say, SAM Basic. It uses integer (whole number) arithmetic and only variables A to Z. There are no string variables, you can have only one command per line, and control structures are very limited. However, it does work much faster than Basic and it allows you to write a game that is a stand-alone code file. If you need to, you can call Basic or machine code subroutines from GMCL to provide missing commands or functions.

A GMCL module can be executed when the program is run, or as part of an animation, or in response to a collision, or a keypress, or at other times. In fact, GMCL is what is called an "event-driven" language, one of the latest things in computing, apparently. (I discovered this after I wrote it!) The way it is controlled is fundamentally different from normal languages; instead of being executed in (more or less) sequential fashion, as with Basic, a GMCL program is a collection of modules which may have no connections with each other and which are often executed by outside events.

Well, the outside events are not VERY outside, since they often come from the computer noticing something like a sprite collision while running a game, but they are external events in the sense that they are not caused by the GMCL program itself. This can take a bit of getting used to.

Module 1 is always run first when you run a game, and this is fairly conventional. Other modules may be branched to from module 1. However, fairly soon the start-up modules will have finished, and the computer will handle the sprites by itself, without any GMCL programming. From time to time, collisions, key presses, animation sequences or paths may make a specific module or modules execute. Your player-controlled sprite might trigger a "next level" module by reaching a particular point and a sprite vs. block collision being recognised. The module would clear the screen and sprites, set up new backgrounds, replace your sprite and place some fresh hazards on-screen before finishing. No further GMCL actions would be needed for a while. For example, see the game "platform.d" on the disk - if you can grab the crown and reach the door on the right, you will trigger the setup of the next (rather boring) room. (Hint: turn off "feels gravity" and "needs support" if you want to cheat!)

There is one rather different way a module can be executed: a specific module can be executed after each game "cycle" in which all the sprites are moved or checked. This module can keep a count of how many cycles have gone by, which is useful for demos, because it can let you display a screenful of sprites for a time, and then switch to a new one, and finally back to the start. Also it can introduce a hazzard into the game every now

and then, or keep the game running for a particular number of cycles after the player is "killed". See the EDIT GAME DETAILS option for details, and the demos on your disk for examples.

The Coordinate System

The GMC Language uses a coordinate system in which the horizontal and vertical scales (x and y axes) both range from 0 to 255. The x coordinate is in units of TWO pixels (1 byte). Only x values of 0 to 127 are on-screen. This system has the advantage of allowing fast, single byte operations on the coordinate while giving the ability to position sprites off-screen in an invisible "phantom zone" that you never see. With the default game property of IMMEDIATE WRAP: NO, sprites can gradually emerge from, or vanish into this zone, provided they do not collide with anything. See the EDIT GM PROPERTIES option. You will have to mentally multiply an x coordinate in this system by two to estimate where it will be on the screen in the usual system. If the x coordinate of the right-hand side of a sprite increases beyond 255 it will "wrap round" to 0 and the side of the sprite will appear on the left of the screen, emerging from the "phantom zone". X coordinates going below 0 become 255 and the sprite moves gradually into the "phantom zone" on the left, and eventually, if it keeps going, emerges again from the right-hand screen edge. Collisions can occur within the "phantom zone", but missiles cannot be fired. If IMMEDIATE WRAP is set to YES, x coordinates below 0 or above 127 immediately wrap to the other side of the screen, so there is no "phantom zone" to left or right. Attempts to PLACE sprites at x coordinates of 128-255 will result in use of 0-127 instead.

Y coordinates of 191 (at the screen top) to 0 (at the screen bottom) allow for an invisible "phantom zone" when y is 191 to 255. This can be thought of as both above and below the screen, since sprites "wrap round" if they move too high or too low. You can set up an "attack wave" of downwards-moving sprites at y coordinates of 255 or so - they will gradually move onto the visible screen. The meteors in the "meteor.d" game are introduced in this way.

Sprite Planes

Games Master uses a system of a background and six sprite planes. The background contains static graphics that the sprites move over. Sprites move in "collision planes" which exist as layers on top of the background. They are numbered 1, 2, 4, 8, 16, and 32, with plane 1 being the lowest, just above the background, and 32 being the highest. (This numbering system may seem odd, but it allows sprites and blocks to exist on several planes at once by adding the plane numbers together.) Sprites can only collide with something that is on the same plane, and they move OVER anything on a lower plane and UNDER anything on a higher plane. If sprites on the same plane are allowed to cross each other, the one placed last will be on the top.

Sprites can also be placed on a "non-collision plane", 64, which is on top of the collision planes. Nothing on this plane can collide with anything else, even if it is also on plane 64. You might consider using this plane if you want to have a large

number of sprites that do not need collision detection - the program will run faster if they are on plane 64 rather than one of the other planes. (The time taken by collision detection increases according to the number of sprites SQUARED, so it can be significant with, say, 50 sprites.)

Sprite Numbers

GMCL commands generally refer to sprites using a sprite number - this is the number you see next to the sprite's name in the sprite list presented by the PICK SPR option. The command definitions below use the abbreviation "spr". Some commands work on the master copy of a sprite, which is used to generate the copy or copies that are actually used in a game. These commands can use any sprite number that refers to a defined sprite in the list. Other values will give a "Number out of range" or a "Missing sprite" report. Other commands work on the active copies of a sprite, and here the sprite numbers refer to the active copies in the game. If the sprite is not in use you will get an error message, even if the sprite exists in the game's set - you must have put it into use with PLACE, EMIT or TRANSFORM.

Commands that work on active sprite copies can use two special sprite number values, 0 and 255. A sprite number of zero means "the current sprite". If a sprite has just collided with something, it is the "current sprite" and its animation sequence, for example, can be altered in a collision-triggered module using e.g. ANIM 0,7,1 (see ANIM command below). This means that the same module can handle sprites with different numbers, and probably more importantly, many copies of the same sprite (which all have the same number) can be handled by the system without ambiguity, since zero refers to the sprite that has just collided. If a sprite has just been PLACEd or EMITed, it will also be the current sprite, and can be referred to using zero as the sprite number. Using the actual sprite number will be ambiguous unless there is only one copy of the sprite in use. If there are many copies, only the first one created will be affected.

A sprite number of 255 also has a special action - it means "the other sprite" in a collision-triggered module. You can use this to make the sprite that was collided with do something - e.g. switch to a new animation sequence, or die. If you use the sprite number 255 when there has been no collision, the command will have no effect.

GMCL EXPRESSIONS

GMCL deals only with whole numbers between 0 and 65535. This has substantial speed advantages, because numbers can fit in only two bytes and can be handled by very fast and simple machine code operations. Values that you might expect to be negative, like 1-2, "wrap round" to 65535. This is a bit like a milometer being wound back past zero to 99999, because 65535 is FFFF in hexadecimal or 1111111111111111 in the binary that computers use. 1-3 would be 65534 (FFFEH). When a single byte value is expected by a command, negative values wrap round to 255, 254, etc. (FFH, FEH etc.)

Only literal numbers can be entered with "unary" (leading) minus signs, so that e.g.:

```
TEXT -1  is O.K. but
TEXT -A  is evaluated as just A. The minus is ignored.
```

However, you could use:

```
TEXT 0-A, which has the same effect.
```

A line containing something like SPEED 1,2,-1 is converted into an internal format in which the numbers are stored as byte values of 1, 2 and 255. When the line is re-listed it is recreated from the internal format (saving a lot of memory) and it will appear as SPEED 1,2,255. This has exactly the same action as the original SPEED 1,2,-1.

GMCL does not allow all of Basic's operators to be used. You can use +,-,*,/,=,<>,<,> AND, OR and XOR. The priority of the operators is the same as usual. Division, like every other operation, always gives a whole number result, so 10/3 is 3, not 3.33333. Comparators (=,<>,<,>) give 1 for True, 0 for False, like Basic. AND, OR and XOR can combine conditions, as in BASIC - e.g. IF X>2 AND X<9. IF (cond A) XOR (cond B) means "IF either cond A or cond B but not both" do something. AND, OR and XOR also work at the binary level, which may be useful if you understand binary. For example, TEXT 16 OR 8 gives 24, because anything which is a binary 1 in the first number OR the second number is a 1 in the result. (The operation is 10000 OR 01000=11000 in binary.) The more "techie" of you might use AND to read individual bits controlling sprite properties - see SPOKE, SET and RES.

You cannot use brackets to force parts of an expression to be evaluated first - so e.g. TEXT (X+Y)*3 is not allowed. You would have to assign X+Y to a temporary variable, and then multiply that by 3, or use: TEXT X*3+Y*3 (which is equivalent).

THE GMCL COMMANDS - in Alphabetical Order

ANIM spr, sequence, position

e.g. ANIM 1,4,5 or ANIM 4,1,1

Sets the animation sequence for the specified sprite copy to a particular sequence, and a particular position in that sequence. For example, to make sprite 5 use sequence 7, and begin at the start of that sequence, you would use:

```
ANIM 5,7,1
```

An undefined sequence will give an error message. The position should be 1 or more, and refers to the row of the sequence definition which is to be used as a start point. This allows you to have many walking or flapping sprites which use the same animation sequence but are not in step with each other, because they started at different points in the sequence. For example:

```
PLACE 1,20,30,4
ANIM 0,7,1
PLACE 1,40,30,4
ANIM 0,7,5
```

If you use a position past the sequence end, the beginning will be used instead.

The MANIM command is a related command that alters the master copy of a sprite so that ALL copies of that sprite PLACEd or EMITed after the command have the new animation sequence. For example: MANIM 4,7,2. Sprite numbers of 0 and 255 cannot be used.

BACK spr, x, y, frame

Places an image of a sprite on the background at the specified x and y coordinates, using the specified frame of the sprite's graphics. The graphics do not have to have a mask, which saves memory if you do not need a clipped outline. This command is mainly used to place sections of background scenery which will be moved over by the game sprites.

BFILL block, spr, frame

FILLs a specified block (in the current set) with a specified frame of a specified sprite, repeated or truncated as necessary to cover the area. If the sprite has a mask, the pattern will be masked and may contain holes or be translucent. Only the background is modified - no sprites are made active. For example, to fill the entire game area (block 1) with copies of sprite 2's third frame, use:

```
BFILL 1,2,3
```

This is useful both for creating overall backgrounds using suitable patterns (see e.g. "trellis.s" and "grass.s" sprite files) and for making obstacles and static platforms visible.

BLOCK block, X, Y, width, length, plane, type

Defines a block which sprites can collide with or bounce off, and which can be filled. The same block can be redefined multiple times. The main menu EDIT BLOCKS option is another way of defining blocks which is more interactive; however, it can't be used in the middle of a game!

The block number should be between 1 and the number of blocks in the current set. (You can force a block set to contain a particular number of blocks - e.g. 50 - by doing a dummy edit of block 50 using the EDIT BLOCKS option.) X and Y are the coordinates of the top left corner of the block. The x coordinate system and width are in units of 2 pixels, so on-screen x coordinates run from 0 to 127, but you can use larger values to define a block which is partly or entirely in the phantom zone. Sprites can still collide with such blocks. The y coordinate system and height are in pixels, with 191 at the screen top and 0 at the bottom.

The plane should be 1,2,4,8,16 or 32, or the sum of some of these numbers, if you wish the block to exist on several collision planes. Add 64 to this value if the block is to be "enclosing".

The type should be 1-32; it determines the collision type. See the SPR vs. BLK COLLIS option for an explanation.

After a complex bit of block creation (e.g. for a Breakout game - see disk) it can be instructive to use the EDIT BLOCKS option to look at the block outlines.

BLOCKSET n

Makes the game use block set N. N should be between 1 and 32, and the set must have at least one block in it or you will get an error message. Sprites can collide with or bounce off these blocks, and they can also be filled. Block sets are created using the main menu EDIT BLOCKS option.

BORDER n

e.g. BORDER 5 or BORDER b+1

Like Basic's BORDER command. N must be between 0 and 15.

BTYPE block type, value

This can be used to set the properties of one of the 32 block types during a game. To calculate the value, start with 128 if the block should be supporting/excluding, otherwise use zero; add the belt speed (1 to 15) if the block should be a right-moving belt, and add (32 minus belt speed) if the belt should move left.

CALLBAS line number

Calls a specified Basic subroutine, which should end in a RETURN. For example:

```
CALLBAS 100
```

and in the Basic program:

```
20000 FOR n=1 TO 20: BEEP .1,n
20010 NEXT n: RETURN
```

With graphics or print commands, you will need to repeat the command for each of the two screens used by the game, e.g.:

```
20100 SCREEN 1: CIRCLE 80,88,20
20110 SCREEN 2: CIRCLE 80,88,20: RETURN
```

Certain commands should not be used in a Basic subroutine, because they corrupt the game code. These are FILL, GRAB, PUT, ROLL and SAVE.

The GMCL variables A-Z can be examined or modified from Basic using simple procedures and functions such as those in the Utility program "BASint.u" on your disk.

CALLCD page, offset

For advanced users who happen to need it, calls a machine code subroutine in a specified page at a specified offset. The offset should be between 0 and 32767 (7FFFH). The page will be switched in at address 0, and the page above at 16384 (4000H). The stack is located near the end of memory, and interrupts are running in mode 2, so 0038H does not handle interrupts as it does normally. One of the two screens will be paged in at 32768 (8000H). Important system variables can be read from the list detailed under VPOKE. On entry to your code, the IX register will hold the address of the table of GMCL variables A-Z in section C of the memory map (32768-49151). The table will need paging in using the value in the D register sent to port 251. For each variable, the value is in LSB/MSB form, followed by 4 bytes that are reserved for use during FOR-NEXTs, followed by the next variable.

CALLMOD module number

Calls the specified module. When this has finished, the program continues at the next line. The module number should be 1-128.

CLS

Like Basic's CLS command. Clears the entire screen to the current PAPER colour.

EMIT spr1, spr2, x offset, y offset

This command emits (or fires, or launches) spr2 from spr1, starting spr2 at a position offset by specified amounts from the position of spr1's top left-hand corner. The offsets let you launch a missile or fire a shot from a particular bit of the firing sprite. For example, if a tank is sprite number 6 and a shot is sprite number 2, then:

```
EMIT 6,2,0,-4
```

might be sensible if we had a side view of the tank facing left. The shot would start with its left-hand edge level with the front of the tank (x offset 0) and its top edge below the top of the tank (y offset -4).

FLIP spr

This command flips a sprite's graphics top to bottom. The sprite number can be any sprite in the sprite list, or zero for the current sprite. (The distinction between master copies and working copies of a sprite is blurred here, because both are connected to a single set of graphics. Running a game again re-FLIPs graphics if needed to regain the original orientation.) The offsets and direction of any missiles launched after a FLIP are automatically adjusted as explained for MIRROR.

FOR variable=start, finish, step

This resembles Basic's FOR command but is simplified. Commas replace the TO and STEP keywords, and a step must always be specified. The variable must be A to Z. The numbers must all be positive, but if the start value is higher than the finish value the step value will automatically be treated as negative by the program. The lines after FOR are always executed at least once. The NEXT at the end of the loop checks to see if the program should exit the loop or branch back to just after the FOR. Here are some examples:

```
FOR X=10,110,10
PLACE 4,X,100,4
NEXT X

CLS
FOR A=5,0,1
TEXT A
NEXT A
```

NOTE: The same FOR-NEXT variable should not be used for more than one FOR-NEXT loop in any given module. Avoid e.g.:

```
FOR N=1,90,1
PLOT RND(255),RND(191)
NEXT N
REM delay 1 second
FOR N=1,9000,1
NEXT N
```


Reusing the variable will confuse the compiler. Alter the second use of N to some other letter.

See also: NEXT

GOTO letter

Continue the program at the specified LABEL (see below). The letter used can be A-Z; it has no relationship to the variable of the same name. For example:

```
LABEL T
LET T=T+RND(3)
TEXT T," ";
IF T<30: GOTO T
```

The LABEL must be within the current module, and it must exist, or you will get an error message when you try to compile the module.

HOME spr, x offset, y offset

This command is used to move a specified sprite towards or away from the player's sprite. The offsets are the number of units closer the specified sprite will move, if possible. If they are negative, the sprite will move away. A HOME command used in a module that is part of a sprite's animation sequence or movement path can make it attack or flee with varying speeds, perhaps including some element of randomness.

IF condition : command

IF the condition is true, the command after the colon will be executed; if not, the program continues with the next line. The fact that only single commands on a line are allowed (apart from IF plus another command) means that IF condition : GOTO has to be used quite often - this reminds me of my old ZX80! For example:

```
IF RND(5)>3: GOTO X
PLACE 1,40,50,4
SOUND 1
RETURN
LABEL X
PLACE 2,40,50,4
SOUND 2
```

JPMOD module number

This makes the program continue at a specified module. The number should be 1-128. For example:

```
IF Z=100: JPMOD 5
```

KILL sprite

The command works on active sprite copies. The specified sprite will be "killed" and will vanish from the screen. This command can be used as part of a path or animation sequence; for example, an explosion sprite with several frames could "kill itself" with a module containing KILL 0 activated when the last frame is shown. An example of this is shown in the "meteor.d" game on your disk. The "explode" sprite, number 5, uses animation sequence 9, which goes through 5 frames before activating module 32 which contains KILL 0.

LABEL letter

Acts as a label that GOTO can use as a destination. The letter used can be A-Z, but has no relation to the variable of the same name. LABELs can only be GOTOed from within the same module. Different modules may contain the same LABEL values without confusion. See also: GOTO

LET variable=value

Assigns a value to numeric variable A-Z. The variables are all set to zero when the game is run, but they always exist. They can hold values of 0-65535. (Some commands treat 128 to 255 as -128 to -1 like machine code does, and you can enter values in the form of negative numbers.) For example:

```
CLS
LET X=2*4
TEXT X
LET X=-6
TEXT X
```

LOCATE x,y

Sets the text output position used by TEXT, LTEXT and STEXT to a given x and y position. The next text output will start with the top left-hand corner of the first character at that position. The x-axis runs from 0 to 127 and the y-axis from 191 at the top to 0 at the bottom. For example:

```
LOCATE 0,191
TEXT "Top left"
LOCATE 64,95
TEXT "middle"
```

LTEXT - see STEXT

MANIM - see ANIM

MIRROR spr

This command mirrors a specified sprite's graphics left to right. The sprite number can be that of any sprite in the sprite list, or you can use zero for the current sprite. This operation can also be performed automatically when a sprite reverses direction by using the EDIT SPR DETAILS option.

Repeated MIRROR commands will make a sprite's orientation alternate. SCLEAR, or re-running the game will restore the initial orientation.

If the sprite EMITs any missiles after mirroring, the offsets applied to them will be adjusted automatically so that they appear at the same initial position relative to the launcher. In other words, if the missile came from the left side of the sprite, after a MIRROR it will come from the right side. The x and y speeds of the missile will also be adjusted, so that it moves in the desired direction.

The missile graphics themselves will not be mirrored, which is fine for many missile types. However, for some missiles this might look silly, and a more complex approach is required, keeping track of the sprite's orientation using a variable or SPEEK of the sprite's data. This allows a different missile sprite, with reversed graphics, to be emitted when the orientation of the launcher is reversed.

MOVE spr, x offset, y offset

This command moves the specified sprite copy by a specified distance on the x and y axes. Positive x offsets are to the right, positive y offsets are upwards. The x offsets are in whatever units the sprite moves by - pixels or double pixels.

You could use this command for several purposes. For example, you could implement a "teleport" - when a sprite touches a particular block, have the collision activate a module that generates a sound effect and moves the sprite 50 units away to a "receiver". Or you could use a penetrable block triggering a MOVE 0,0,1 command to move the sprite upwards so long as it stayed in contact with the block. If the sprite has the Feels Gravity property, this MOVE command may only partly overcome gravity, but a figure would be able to jump higher, as though on a trampoline. If the sprite is not affected by gravity, the block will act like a vertical belt. MOVE can also produce special user-controlled movements via a sprite's Key Modules (see the EDIT SPR DETAILS Option).

MPATH - see PATH

MSPOKE - see SPOKE

NEXT variable

Terminates the loop of the matching FOR command. The variable should be A to Z. See FOR.

PAL n

Changes the entire palette to one of the pre-defined game palettes. For example, PAL 3 selects palette 3. Palettes can be edited using the EDIT PALETTES option on the main menu. N should be between 0 and 15. All the palettes except zero start off defined to be the standard SAM colours. Palette zero starts as entirely black, and can be used to blank the screen while graphics are being set up. Selection of another palette then instantly reveals the scene.

PALETTE palette entry, colour number

This is a simplified form of Basic's PALETTE command. The palette entry should be 0-15 and the colour number 0-127.

PAPER n

Like Basic's PAPER command, but must be used on its own, not as part of another command. N must be between 0 and 15. Sets the colour used by CLS and the background colour for text produced by the TEXT command.

PATH spr, path, position

Sets a path for the specified sprite copy, and makes its movement type 3 (PATH). A particular position in the path can be selected - 1 is the start, and later positions are the number of moves from the start. For example, to make sprite 2 use path 7, and begin at the start of that path, you would use:

```
PATH 2,7,1
```

An undefined path will give an error message when the command is executed. Many sprites can use the same path, and they can each start at different points if desired. For example:

```
PLACE 1,20,30,4
PATH 0,7,1
PLACE 1,40,30,4
PATH 0,7,50
```

If you use a position past the path end, the beginning will be used instead.

The MPATH command is a related command that alters the master copy of a sprite so that ALL copies of that sprite PLACEd or EMITed after the command have the new path setting. For example: MPATH 4,7,2. Sprite numbers of 0 and 255 cannot be used.

PAUSE n

Makes the program wait for N 50ths of a second. For example, PAUSE 50 would wait for 1 second. The pause will be cut short if you press a key. Values between 1 and 255 give delays up to about 5 seconds. PAUSE 0 waits forever, unless a key is pressed.

For a delay that cannot be shortened by a key press, use an empty FOR-NEXT loop. For a 1-second delay, use something like:

```
FOR T=1,9000,1
NEXT T
```

PEN n

Like Basic's PEN command, but must be used on its own, not as part of another command. N must be between 0 and 15. Sets the pen colour for text produced by the TEXT command.

PLACE spr, x, y, plane

This command brings a specified sprite into use at a given position on a particular collision plane, by making an active copy from the master copy. The sprite number must be 1 to 96, and the sprite must exist or you will get an error message when the module is executed. The plane should be 1, 2, 4, 8, 16, 32 or 64, or the sum of some of these numbers if the sprite is to exist on several planes. The same sprite number can be PLACEd multiple times, generating multiple copies of the sprite from the master copy. Changes to the properties of these copies can be made using ANIM, SPEED and SPOKE after a PLACE command, if desired.

PLOT x,y

This is similar to PLOT in BASIC, except that PEN, OVER etc. cannot be included in the command. A pixel is plotted in the current PEN colour. Unlike the case with all the other commands, the x scale used by PLOT runs from 0 on the left of the screen to 255 on the right. This allows the precision to plot any desired pixel. Plotting is best done before any sprites are placed on the screen, because pixels cannot be successfully plotted "under" an existing sprite - you may get flickering pixels.

RANDOM n

Similar to Basic's RANDOMIZE. Sets the random number generator to a specified value, if n is between 1 and 65535. If n is 0, a random value is used. RANDOM 0 is useful in ensuring any random actions in a game are different with each go, whereas other values would be used to make a particular pseudo-random sequence happen every time. See also: RND(n) function.

REM comment

REM is like REM in Basic - it precedes comments. For example:

```
REM man has hit platform 2
```

RES spr, byte, bit

Resets (makes equal to zero) a particular bit in a sprite's data. See the list after the SPOKE command. For example, to make a sprite non-edge-limited, you could use RES (sprite number),38,2. The commannd works on active sprite copies. See also: SET command, SPEEK function.

RETURN

Makes a module end at once, rather than when the commands run out. Often it is used in the form:

```
IF (condition): RETURN
(more commands)
```

SCLEAR

Stands for Sprite CLEAR. Clears the active sprite copies so that none are in use. This is also done automatically when a game is first run. SCLEAR also turns off any sounds, and resets all the sprites to their normal orientations, cancelling any MIRROR, FLIP, TURNL or TURNR operations.

SET spr, byte, bit

Sets (makes equal to 1) a particular bit in a sprite's data. See the list after the SPOKE command. For example, to make a sprite have the BOUNCES property, you could use SET (sprite number),32,6. The command works on active sprite copies. See also: RES command.

SOUND sound number

Causes a pre-defined sound to be made. The EDIT SOUNDS option is used to define a sound. The sound may continue for long after the SOUND command has finished, since an interrupt-driven system is used. The number should be between 1 and 32, and the sound should be predefined, or you will get an error message. The apparent location of the sound in the stereo field is determined by the x coordinate of the current sprite.

SOUNDX sound number, x coordinate

For use when you want to produce sounds without a sprite "source". You specify the apparent source of the sound as part of the command. For example, SOUNDX 1,63 would make pre-defined sound 1 seem to come from near the middle of the screen. An x coordinate of 0 would be near the left-hand edge, and 127 would be at the right.

SPEED spr, x speed, y speed

This command alters the X and Y speeds of a given sprite copy. X speed and Y speed can be positive or negative. For example, to place 6 copies of a sprite at random positions with random speeds, you could use this:

```
FOR N=1,6,1
PLACE 1,RND(100),RND(100)+30,4
SPEED 0,RND(2)-1,RND(2)-1
NEXT N
```

The SPEED command here makes the current sprite (denoted by 0) have an x speed and a y speed of -1, 0 or 1 unit. You could use RND(4)-2 for values of -2, -1, 0, 1 or 2. Positive X speeds are to the right, negative to the left. Positive Y speeds are upwards, negative downwards. Speeds of zero produce no movement.

The units used for the x speed will give a speed of 2 pixels per move unless the sprite has the Pixel X Speed property set to YES. Y speeds are always pixels per move. The sprite's Movement Type should also be 1 (Simple) or possibly 2 (Player) in order for SPEED to have any effect. (See the EDIT SPR DETAILS option.)

A similar command, MSPEED, alters the speeds in the master copy of the sprite from which all other copies are derived. For example, to make all later copies of sprite 1 have left and upwards speed:

```
MSPEED 1,-1,2
```

SPOKE spr, offset, value

This command is nothing to do with bicycles! It stands for Sprite POKE, and allows you to alter most of a sprite's properties from within a program. It can do everything that ANIM and SPEED can do, amongst other things, and is more flexible. It is less easy to use, however, and you may never need to use it.

The command alters one byte in the data that defines a particular sprite, to the specified value. The command works on active sprite copies. The offset is the byte to alter within the sprite data and should be 0 to 44, and the value used will depend on the byte you are altering, but it must be between 0 and 255.

A related command, MSPOKE, alters the master copy of the sprite data from which all copies are made.

Below is a list of the bytes in the sprite data and what they do. It looks rather intimidating, but most of the bytes never need to be altered, and are given just for the sake of completeness.

0 LEVL Sprite plane, or 255 if sprite is out of use.
1 SSLO
2 SSHI Low and high bytes of length to next sprite, minus 2.
3 FSLO
4 FSHI Low and high bytes of frame size.
5 ANTY Animation type. Bit 0=1 if sprite is animated, bit 2=1 if sprite is temporarily stopped, bit 4=1 if anim. is conditional.
6 FRAM Current frame number.
7 FRCT Frame counter for moves per frame counting.
8 CTTY Movement type. Bits 1 and 0=00 for unmoving, 01 for simple movement, 10 for player, 11 for path.
Bit 5 is 1 if alternating path direction.
Bit 7 is 1 if moving backwards on a path, 0 if forwards.
Bit 2 is 1 if movement is temporarily stopped.
9 XSPD Current X speed. Values of 1 to 127 are right, 128 to 255 are left (-128 to -1).
10 YSPD Current Y speed. Values of 1 to 127 are down, 128 to 255 are upwards (-128 to -1).
11 XCRD Current X coordinate.
12 YCRD Current Y coordinate in inverted form (0 at the top).
13 OXCD Previous X coordinate.
14 OYCD Previous Y coordinate in inverted form.
15 WDTH Sprite width in bytes.
16 LNGT Sprite height in pixels.
17 GRPG Page of sprite's graphics as an offset.
18 GROL
19 GROH Low and high bytes of sprite's graphics offset address.
20 FRMS Number of frames.
21 SPNO Sprite number.
22 SAC1 Key 5 module.
23 SAC2 Key 6 module.
24 SAC3 Key 7 module.
25 SAC4 Key 8 module.
26 SAC5 Mirror Left module.
27 SAC6 Mirror Right module.
28 SAC7 Flip Up module.
29 SAC8 Flip Down module.
30 SOXS Sprite's own X speed.
31 SOYS Sprite's own Y speed.
32 COLF Collision flags.
Bit 7 is 1 if supported.
Bit 6 is 1 if bounces.
Bit 4 is used internally.
Bit 3 is 1 if standing.
Bit 2 is 1 if hit block.
Bit 1 is 1 if hit sprite.
Bit 0 is used internally.
33 ORIE Orientation flags.
Bit 7 is 1 if left/right mirror ON.
Bit 6 is 1 if up/down flip ON.
Bit 4 is 1 if mirrored.
Bit 3 is 1 if flipped.
Bits 2-0 give TURN direction.
34 STPM Stop module.
35 COLT Collision type.

36 FLGS Flags.
Bit 7 is 1 if feels gravity.
Bit 5 is 1 if needs support.
Bit 4 is 1 if sprite to be killed.
Bit 3 is 1 if halts on impact.
Bit 2 is 1 if edge-limited.
Bit 1 is 1 if absolute speed.
Bit 0 is 1 if under firer.
37 FLG2 More flags.
Bit 6 is 1 if 1 pixel should be added to x coordinate.
Bit 5 is 1 if pixel x moves allowed.
Bit 4 is 1 if maskless.
Bit 3 is used internally.
Bit 2 is reserved.
Bit 1 is reserved.
Bit 0 is 1 if missile.
38 FVEL Current falling speed due to gravity.
39 PDCB Path.
40 PDPL
41 PDPH Low and high bytes of path position (0,3,6,9 etc.).
42 ADCB Animation sequence.
43 ADPL
44 ADPH Low and high bytes of sequence position (0,3,6,9 etc.).

STAND

Usually used after a collision between two sprites, when you want the colliding sprite to be able to stand on the sprite it has hit. As you would expect, standing is possible only if the collision occurred on the bottom surface of the colliding sprite, i.e. if the sprite is a man, he has to hit with his feet (unless he is upside down, in which case his head will do...).

In other collision directions, STAND simply excludes the sprite from the sprite it has hit. This uses the rectangular box shape the sprite was designed in, and it may look odd unless the graphic fills most of this rectangle.

When the command is effective, the standing sprite will be prevented from falling through the sprite it is standing on, and will be supported with a 1-pixel overlap of the bottom of the colliding sprite and the top of the enclosing rectangle of the sprite below it. For the best appearance, design any sprites that another sprite will stand on to have a top row of unused pixels.

The standing sprite will share the motion, if any, of the sprite it stands on. A common use is in platform games where a player-controlled sprite stands on moving platforms - see the example program "platform.d" on the disk. (If the platform does not have to move, use a BLOCK, not a sprite - it will give a faster program, and you will not need the STAND command at all.)

Note that just as in real life, jumping from a moving object can give you added speed.

STEXT spr,text list

This stands for Special TEXT. It prints text in a similar way to TEXT, but uses the frames of a specified sprite as a character set. This means that letters can be any size, have 16 colours and be masked or transparent, like any sprite, but need more memory. Here is an example:

```
LOCATE 0,80
STEXT 5,"X=",X
```

The frames of the sprite must correspond to the ASCII character set - the CODE of a character, minus 31, gives the required frame number. The first frame should be blank, for a space, the next "!", the 34th. frame should be "A" and the 59th. "Z", etc. A complete set isn't required, but even so the number of frames may impose a formidable memory requirement, since "space" to "@" need to be included before we even reach the range of letters. A sprite file in this format is on your disc as "cset1.s".

A closely related command, LTEXT, for Letter TEXT, assumes that the letter "A" will be the first frame of the sprite, and thus reduces the memory requirements for a character set which is just used to print text. The CODE of a character, minus 64, gives the required frame number. A suitable sprite file is included on the disk as "letset.s".

An even more memory-sparing method is to have just the characters you need as frames of the character-set sprite; if you want to print "SCORE" for example, make frame 1 be "S", 2 be "C", etc. and then LTEXT (sprite number),"ABCDE" which will print frames 1 to 5.

STEXT and LTEXT both space their sprite characters one character-width apart. When a line is full, the extra characters just wrap round to the start of the line again. No carriage returns occur after printing, so normally each STEXT or LTEXT command is preceded by a LOCATE command to set the start of the text.

SUPPORT

Usually used after a collision between two sprites, or a sprite and a block, when you want the colliding sprite to be supported by the sprite it has hit. Support is provided whatever the collision direction, and however the two objects overlap. Unlike the case with STAND, the SUPPORTed sprite is not excluded from the object it collides with. A SUPPORTed sprite such as a player-controlled man can move freely up and down as long as it keeps in touch with the supporting object.

A common application is to allow a sprite to ascend a ladder. An example is included in "platform.d" on your disk. SUPPORT is used in a module triggered by collision of the man with a non-supporting, penetrable block, made long and narrow and superimposed on the background ladder graphic. As long as sprite/block contact is maintained, the man can move upwards, despite having "needs support" and "affected by gravity" properties set by the EDIT SPR PROPERTIES option.

Any motion of the support is not imparted to the supported sprite when you use this command.

TEXT (print list)

TEXT is a very simple print command. It can handle ordinary text enclosed in quotes, or numeric expressions, in a list with comma separators. These act like ";" in a Basic PRINT command. You cannot use AT, TAB, OVER, PAPER, etc. as part of the command. The text is printed in the current PEN and PAPER colours. An automatic carriage return occurs at the end of each TEXT command, unless it it suppressed with a trailing semicolon (;). LOCATE can be used to set the print position. Here are some examples:

```
PEN 15
PAPER 0
TEXT "hello ";
TEXT "SCORE:",s*10
VIEW
PEN 4
PAPER 15
LOCATE 50,100
TEXT A," ",B," ends"
```

The VIEW command must be used to force immediate printing before PEN or PAPER are altered for the next TEXT command. Otherwise, the text will flash.

When the bottom of the screen is reached, you do not get scrolling. Instead, the text will wrap round to the top of the screen.

The horizontal and vertical separation of characters can be altered by using the VPOKE command - normally these are 8 and 9 pixels, respectively. The characters are derived from a copy of the normal SAM character set, and must fit in an 8 by 8 pixel grid. This character set is held as part of a game. You can replace it by any other character set by leaving the Editor, then:

```
LOAD "name" CODE DPEEK(va+4)+bs
```

The code file should be 96*8 bytes long, for 96 8-byte characters, and it could have been saved from the normal character set by e.g.:

```
SAVE "name" CODE UDG " ",96*8
```

See also: LOCATE, PEN, PAPER.

TRANSFORM spr1, spr2, x offset, y offset

Spr1 vanishes and is replaced by spr2, offset from spr1's position as specified. For example, for a spaceship to become a cloud of gas, if the ship is sprite 1 and the gas cloud sprite 12 and you want the gas at the same position, use:

```
TRANSFORM 1,12,0,0
```

If the gas cloud was smaller than the ship, you might want to offset its position to keep it centred on the ship's former position, using e.g. TRANSFORM 1,12,2,-4. On the other hand, if the gas cloud is bigger than the ship, you will need different offsets, e.g. TRANSFORM 1,12,-2,4. The exact offsets will depend on the relative sizes of the sprites, and the effect you want to achieve.

TURNL spr

Turns the graphics of the specified sprite left by 90 degrees, and swaps the width and height values for the sprite. This will only be completely successful if the height of the sprite is an even number of pixels - odd heights will result in some screen corruption, but are not harmful otherwise. The new top left-hand corner of the sprite will be in the same place as the old one was. This command is most often used to turn square sprites.

Since the graphics themselves are rotated, the command may not be suitable when multiple copies of a sprite are in use, because the changes will affect the appearance of all the copies. Multiple TURNL commands will turn a sprite clockwise indefinitely. The sprite number can be any number in the sprite list, or zero for the current sprite.

If the sprite EMITs any missile after rotation, the offsets applied to the missile will be adjusted automatically to make it appear at the same initial position relative to the launcher. In other words, if the missile came from the top of the sprite, after a TURNL it will come from the left of the sprite (the old top). The movement direction of the missile will also be adjusted.

TURNR spr

Like TURNL, but turns the graphics right by 90 degrees.

VPOKE var number,value

This command provides a method of changing some important special game variables from within a program. The variables include the ones that can be set from the EDIT GM DETAILS option on the main menu. The variable number should be between 0 and 20. The VPOKED values that are sensible vary according to the variable involved. A list of the variables and their functions follows. Not all should be altered, and some are unlikely to be useful. Values can be examined with the VPEEK function. The variables are accessible from machine code at address FE01H and above, and from Basic using PEEK(start+7681+var number).

0 OTHERP Page of other screen. This variable has a different value according to which screen is in use; it holds the page value to write to port 251 (HMPR) in order to select the OTHER screen page, so that the screen can be addressed at 32768-57343 (8000H-DFFFH). Do not alter.
1 DBASEP Page of start of program data. Do not alter.
2 GRAXP Page of start of graphics, as an offset from DBASEP. Do not alter.
3/4 GRAPHIX Offset of start of graphics from page start. Do not alter.
5/6 Reserved. Do not alter.
7 WIDTH Character horizontal separation with TEXT.
8 HEIGHT Character vertical separation with TEXT.
9 BORDER colour.
10 MINIMUM GAME DELAY.
11 FORCE OF GRAVITY.
12 MAXIMUM FALLING VELOCITY.
13 IMMEDIATE WRAP. Zero if YES, non-zero if NO.
14 SCREEN Screen number in use - 1 or 2.
15 EVERYAC Module number to execute every "cycle", or zero.
16 INTMODE Zero for Games Master 100 ints./sec. or non-zero for the ROM's 50 ints./sec. version.
17 ESCOPT Zero if it is impossible to exit a game, 1 if it is possible. Normally 1.
18 Game pause key port, low byte.
19 Game pause key port, high byte.
20 Game pause key bit mask with bit for active key high. Normally the port is &f7f9 and the mask is &40, denoting the TAB key.
21 Collision direction in last collision; 0 means the sprite hit with its top, 1 with its left, 2 with its bottom and 3 with its right.
22 Top edge. Normally 191.
23 Bottom edge. Normally 0.

VIEW

Normally, most graphics operations are done on a hidden screen, which is only made visible when the GMCL module or modules have finished. For example, if the module contains:

```
FOR N=1,20,1
TEXT N," ";
PAUSE 20
NEXT N
```

You will see nothing until all 20 numbers have been printed. However, if you add the line VIEW just before PAUSE 20, the screen will be displayed after every number is printed. Use of VIEW will slow a program down, especially if there are lots of sprites on the screen, but it can be useful.

Note: For technical reasons, if you use enough TEXT or BACK commands in one module, VIEW will happen automatically. Try the example above without VIEW but using: FOR N=1,400,1 instead of: FOR N=1,20,1.

THE GMCL FUNCTIONS

Functions are keywords that return a value, unlike commands. They must always be used after a command, not on their own.

BPEEK(block, offset)

This function allows you to read stored information about a given block. If the block number is zero, the last block collided with will be assumed. The format is as follows:

Offset	Value
0	Plane
1	Left margin (x coordinate)
2	Bottom margin
3	Right margin (x coordinate+width-1)
4	Top margin (191-y coordinate)
5	Type

The top and bottom margins are stored in an inverted y-scale with 0 at the top and 191 at the bottom. (This works faster, for the computer.)

For a working example, see the BREAKOUT game on your disk. This uses BPEEK to read the x and y coordinates of the brick collided with, enabling it to be removed from the screen.

FALLS

Gives the falling speed of the current sprite after a collision. The value will always be positive or zero (if the sprite is moving upwards). The "platform.d" game on the disk uses FALLS to kill the man if he falls too far. The setting of the force of gravity and maximum falling speed in the EDIT GAME DETAILS option are relevant. The higher the force of gravity, the faster a particular falling speed is reached, and the shorter the drop that will be "lethal" if you trigger "death" on a particular falling speed using e.g. IF FALLS >6: KILL 0.

Actually, FALLS gives an impact speed, rather than an absolute speed. This makes no difference if the collision is with something stationary, such as a block. However, just as in real life, you will be more likely to live if you jump down onto a lift that is moving downwards than one that is moving upwards.

INKEY

Returns the key number being pressed, or zero if no keys are pressed. The key numbers are those shown on the keyboard map on page 180 of The Sam Coupe User's Guide, except that SYMBOL is 55, CNTRL is 63 and SHIFT is 71. You could try this:

```
LABEL L
LOCATE 0,191
TEXT INKEY,"  "
IF INKEY<>64: GOTO L
```

Press the space bar (key number 64) to exit the loop.

RND(n)

Returns a random number between 0 and N, like BASIC's RND(n). N must be between 0 and 255. Here is an example that produces 12 numbers between 0 and 9:

```
FOR T=1,12,1
TEXT RND(9)
NEXT T
```

Try it with RND(99) or RND(255) or RND(2)+2. RND is very useful for making games interestingly unpredictable. See also RANDOM.

SPEEK(spr,offset)

Returns the value of a particular byte in a sprite copy's data. For example: TEXT SPEEK(5,0) would print the first (offset 0) byte of sprite 5 (if it was in use) which is the plane number. See SPOKE for more details of sprite data format. You can use a sprite number of zero to mean "the current sprite" and 255 to mean "the other sprite" in a collision.

MSPEEK(spr,offset)

Like SPEEK but reads a byte from the master copy of a sprite. Any sprite examined with MSPEEK must exist.

VPEEK(variable number)

Returns the value of one of the internal game variables. For example, VPOKE 11,VPEEK(11)+1. See VPOKE and EDIT GM DETAILS for details of these.

EXAMPLE PROGRAMS

On your disk are some simple example game data files and demos that show how to implement common game elements, such as changing sprite orientation, scrolling landscapes, moving platforms and lifts, and missile launching. The files all end in ".d" and can be loaded with the LOAD GAME (DATA option) and examined. The modules are REMed. None of these programs is supposed to be a complete game. Rather, they are examples that cry out for improvement, modification and extension - so get going!

UTILITY PROGRAMS

On your disk are a number of Utility programs that range from a program to reduce the size of sprite graphics in existing sprite files, to specialised sprite and scenery drawing programs that may be a useful source of ideas to those who, like me, have no artistic ability. They all finish in ".u" and are REMed. You can use DIR 1"*.u" (or just DIR "*.u" if you have MasterDOS) to list the file names.

LMG.u	Load Missing Graphics utility (see: When You Are Short of Memory).
reducer.u	Takes a sprite file and produces another file containing a smaller (by various percentages) version of the sprite.
BASint.u	Allows passing values back and forth to BASIC.
alpine.u	Produces random landscapes.
eye.u	Draws the "eye.s" sprite.
scrnrot.u	Rotates a 192-pixel square section of screen - this might be useful before loading a screen and grabbing frames from it.

MODIFYING THE EDITOR

The Editor is a BASIC program with a few machine code subroutines. It isn't a particularly elegant program, because it was written to live as happily as possible with older ROMs. It could easily be improved, and you are free to do this. However, you must NOT sell or give away even modified copies of the Editor.

The Editor is a BIG Basic program, containing many procedures. This means that there is a delay after the program has been edited, before execution starts. This delay is MUCH shorter with Master Basic loaded, and it will make the edit - RUN - re-edit cycle much faster. Also, you will find the program search command REF invaluable. (And oodles of other commands, all for just £15.99 from Betasoft.... hint, hint. Send an SAE for info.)

You may want to modify the Editor by adding Basic subroutines for use by your game. I suggest adding these at, say, line 20000 and above. When you have finished a version of the game, save the game file - and don't forget to save the subroutines too! You can EXIT to Basic, and either save the entire Editor, plus subroutines, or CLEAR: DELETE TO 19999 and then save the subroutines alone for later loading or merging.

Some of the first lines in the Editor program create variables that set Outline status (1 for yes, 2 for no), masking method and edit mode (4 for mode 4, 32 columns, 3 for mode 3, 64 columns). The next line defines file extensions for Sprite, Game, Data, Sound and Module files. These can be altered.

SPRITE FILE FORMAT

These details may be of use to anyone converting from or to other formats. It may be easier to use LOAD SCREEN and GRAB FRAME, or EDIT GRAPHICS (PUT option) and SAVE SCREEN instead, though.

Sprite files normally have names ending in ".s". They are CODE files. The first byte is 123 and is checked for on loading. The bytes that follow are as described under SPOKE, from SSLO to ADPH. After these come the graphics data for frame 1, then the mask for frame 1, then the graphics for frame 2, etc., until the last frame. If the sprite allows pixel x movement, a second set of graphics and masks, shifted right by 1 pixel, follows the first set. The frame and mask data is like a GRAB string stripped of its first 3 bytes; i.e the data for the top row of pixels comes first, then the data for the second row, etc. Each byte of the data codes for two pixels.