Writing Games in PicoMite MMBasic

Introduction

PicoMite MMBasic is a remarkably capable platform for writing games, offering a suite of firmware-accelerated subsystems purpose-built for game development. Whether you want to create a side-scrolling platformer, a Wolfenstein-style 3D shooter, a sprite-based arcade game, or a rotating-cube puzzle, MMBasic provides the building blocks -- all accessible from BASIC with no C coding required.

This guide provides an overview of the game development facilities, explains how they fit together, and offers practical guidance on structuring a game. For full command reference details, see the dedicated manuals:

- SPRITE_User_Manual.md -- Sprite engine (loading, display, collision)
- TILEMAP_User_Manual.md -- Tile map engine (scrolling worlds, tile collision)
- Raycaster_User_Manual.md -- First-person 3D raycaster (RP2350 only)
- 3D_Graphics_User_Manual.md -- Quaternion-based 3D object rendering
- BLIT_User_Manual.md -- Block image transfer and scaling
- FRAME_User_Manual.md -- Text-mode UI panels (HUDs, menus)
- PLAY_SAMPLE_User_Manual.md -- Wavetable synthesis with ADSR envelopes

---

Choosing Your Display Mode

The first decision is your display output and resolution. This determines available colours, performance, and which subsystems you can use.

VGA / HDMI Output

Mode	Resolution	Colours	Best For
MODE 1	640x480	2 (per tile, 16 palette)	Text-heavy games, retro style
MODE 2	320x240	16 (RGB121)	Recommended for most games
MODE 3	640x480	16 (RGB121)	Higher-res sprite games
MODE 4	800x600	2	Large monochrome games
MODE 5	1024x768	2	Large monochrome games

MODE 2 (320x240, 16 colours) is the sweet spot for game development. It provides a full 16-colour RGB121 palette, fast framebuffer operations, and is the recommended mode for the TILEMAP and RAY engines.

On RP2350 hardware, additional VGA222 modes offer 64 colours at various resolutions.

SPI LCD Panels

Supported panels include ILI9341 (320x240), ST7789 (240x240 or 320x240), ST7796 (480x320), ILI9488 (480x320), and many more. On RP2350, buffered SPI variants (e.g., ST7796SPBUFF, ILI9341BUFF) support full framebuffer compositing, making them viable for game development.

The RGB121 Palette

Most game subsystems (TILEMAP, RAY, BLIT operations) work with the 4-bit RGB121 palette:

Index	Colour	Index	Colour
0	BLACK	8	RED
1	BLUE	9	MAGENTA
2	MYRTLE	10	RUST
3	COBALT	11	FUCHSIA
4	MIDGREEN	12	BROWN
5	CERULEAN	13	LILAC
6	GREEN	14	YELLOW
7	CYAN	15	WHITE

Recolouring Without Redrawing - the MAP Command

Those sixteen slots are not fixed colours: they are indexes. What each one looks like is decided by the display at scanout, from a sixteen-entry colour map. Change the map and everything already on the screen in that slot changes colour - with no redraw, and no per-frame cost at all.

That makes it the cheapest animation on the machine. A pulsing crystal, a throbbing force field, a warning lamp, water shifting between shades, a power-up that glints: draw the object once in a reserved slot, then just move the map.

```
CONST GLOW = 10                    ' a slot nothing else in the artwork uses
DIM INTEGER glow(5), phase
glow(0) = RGB(0,0,96)    : glow(1) = RGB(0,64,160)  : glow(2) = RGB(0,128,224)
glow(3) = RGB(96,192,255): glow(4) = RGB(0,128,224) : glow(5) = RGB(0,64,160)

' ... draw the crystal once, using colour GLOW ...

DO
  phase = (phase + 1) MOD 6
  MAP(GLOW) = glow(phase)
  MAP SET                          ' commits every pending change at once
  ' ... the rest of the frame ...
LOOP
```

MAP SET is what makes staged changes live, so several slots can be moved and committed together. MAP RESET puts the standard sixteen back.

Four things to know before relying on it:

Reserve the slot. The map is global. Recolour a slot your hero's sprite also uses and your hero changes colour too. Decide early which slots are "effect" slots and keep the artwork out of them.

MAP(i) does not read back the colour. As a function it returns the RGB value that selects slot i, not the colour that slot is currently showing. Nothing can ask the display what a slot looks like, so keep your own copy if you need to know.

The map survives RUN. It is display state, not program state. A game that leaves a slot recoloured finds it still recoloured next run - and so does the next program the user starts. MAP RESET : MAP SET during start-up, and again on the way out, saves a great deal of confusion.

Firmware before 6.03.02b8 ignores the map in SAVE IMAGE. Screenshots come out in the default palette, so a map effect cannot be checked from a saved BMP on older firmware - look at the screen.

Writing Games in PicoMite MMBasic

The Framebuffer -- Your Rendering Canvas

Almost all game rendering in MMBasic uses double buffering via the FRAMEBUFFER system. The pattern is simple: draw everything to an off-screen buffer, then copy the finished frame to the display in one operation. This eliminates flicker and tearing.

Setting Up

```
MODE 2                    ' 320x240, 16 colours
CLS
FRAMEBUFFER CREATE        ' Allocate off-screen buffer (F)
FRAMEBUFFER WRITE F       ' Direct all drawing to the framebuffer
```

The Core Game Loop Pattern

```
DO
  FRAMEBUFFER WRITE F     ' Draw to framebuffer
  CLS RGB(BLACK)          ' Clear the frame

  ' --- All your rendering goes here ---
  ' Draw background, tilemap, sprites, HUD, etc.

  FRAMEBUFFER COPY F, N   ' Flip: copy framebuffer to display

  ' --- Input and game logic ---
  k$ = INKEY$
  ' Process input, update positions, check collisions...

LOOP UNTIL done%
FRAMEBUFFER CLOSE
```

Do not hand the flip to the other core

FRAMEBUFFER COPY F, N has a third form, FRAMEBUFFER COPY F, N, B, which gives the copy to core 1 and returns immediately. It looks like free performance and in a game loop it is a trap, because the very next thing the loop does is CLS the buffer that is still being copied. On a display fast enough you get away with it; on a slower one the screen is shown a frame being blanked underneath it, which reads as flashing that comes and goes with how much you are drawing - and looks for all the world like a tearing or double-buffering problem rather than what it is.

Use the plain two-argument form in the loop. If you want the background copy, you must not touch that buffer until the copy has finished, which in practice means having something else to draw into. Note also that the form is only honoured by some display types and is silently a no-op on the rest, so it can appear to work for months on the machine you develop on.

Layer Compositing

The framebuffer system supports multiple buffers for compositing:

Buffer	Name	Purpose
N	Display	The physical screen

F	Framebuffer	Primary off-screen buffer
L	Layer	Secondary overlay buffer
T	Top Layer	Additional overlay (RP2350 VGA/HDMI)
2	Second FB	Second framebuffer (VGA/HDMI)

Use FRAMEBUFFER LAYER to create the layer buffer for overlaying HUD elements, particle effects, or other transparent overlays on top of the game world. How the layer is composited onto the display depends on your output type -- this is a key difference between LCD and VGA/HDMI builds.

LCD Displays: Explicit MERGE

On LCD builds (SPI or parallel panels), layer compositing is performed explicitly using FRAMEBUFFER MERGE or BLIT MERGE. These commands take the Layer buffer, overlay it on top of the Framebuffer with a transparency colour, and send the composited result directly to the physical LCD. Neither the Layer nor the Framebuffer is modified -- only the LCD receives the merged output.

```
' LCD layer compositing
FRAMEBUFFER CREATE
FRAMEBUFFER LAYER
FRAMEBUFFER WRITE F
' ... draw game world on framebuffer ...
FRAMEBUFFER WRITE L
CLS 0                        ' Clear layer, colour 0 = transparent
TEXT 10, 10, "SCORE: 1000"    ' Draw HUD on layer
FRAMEBUFFER MERGE 0           ' Composite layer over framebuffer, send to LCD
```

Use BLIT MERGE to composite only a sub-rectangle (e.g., just the HUD area) for better performance. Background and repeating modes let the merge run on core 1, freeing the main core for game logic.

VGA / HDMI Displays: Automatic Layer Compositing

On VGA and HDMI builds, FRAMEBUFFER MERGE and BLIT MERGE do not exist. Instead, layer compositing happens automatically as part of the scanline output -- the hardware renders the Layer buffer on top of the display buffer every frame with no explicit command needed.

Simply draw to the Layer buffer and the compositing is handled for you:

```
' VGA/HDMI layer compositing (automatic)
FRAMEBUFFER CREATE
FRAMEBUFFER LAYER
FRAMEBUFFER WRITE L
CLS 0                        ' Clear layer, colour 0 = transparent
TEXT 10, 10, "SCORE: 1000"    ' Draw HUD on layer
FRAMEBUFFER WRITE F
' ... draw game world on framebuffer ...
FRAMEBUFFER COPY F, N         ' Copy framebuffer to display -- layer is composited automatically
```

On RP2350 VGA/HDMI in certain modes (MODE 2, 3, 5), a Top Layer (T) is also available via FRAMEBUFFER LAYER TOP, creating a three-layer composite with this priority order:

1. Top Layer (T) -- highest priority (checked first)
2. Layer (L) -- middle priority
3. Display (N) -- base layer (shown where the layers above are transparent)

Each layer has its own transparency colour. This allows separating different overlay elements -- for example, the Layer for in-game particle effects and the Top Layer for a fixed HUD.

Writing Portable Code

Two more things differ between machines, and both are cheap to allow for.

MODE may not exist. A VGA or HDMI build has modes; an SPI panel has one fixed resolution and the command is an error there, which stops your program on its second line. Ask for the mode you want and let it fail:

```
ON ERROR SKIP 1
MODE 2
ON ERROR CLEAR
```

Whatever the display already is, is what you then draw on.

The screen may be taller than the area you draw. If you have written for 320x240 and the panel is 320x320, everything still works - but FRAMEBUFFER CREATE takes HRES * VRES / 2 bytes, so you are paying for eighty rows you never use. POKE DISPLAY VRES n changes MMBasic's own idea of the height, with nothing written to flash, and must come before FRAMEBUFFER CREATE:

```
DIM INTEGER realVres
realVres = MM.VRES
IF realVres > 240 THEN POKE DISPLAY VRES 240
FRAMEBUFFER CREATE
```

On the 320x320 panel that is 12 KB of heap back - which, as the next section explains, is the difference between a game that runs and one that does not. Put the height back on the way out, on every exit path, and be aware that OPTION DISPLAY VRES is a different command that is refused outright when the panel is the console.

If your game needs to run on both LCD and VGA/HDMI hardware, structure the rendering so that the layer compositing step can differ:

```
FRAMEBUFFER CREATE
FRAMEBUFFER LAYER

DO
  FRAMEBUFFER WRITE F
  ' ... draw game world ...
  FRAMEBUFFER WRITE L
  CLS 0
  ' ... draw HUD/overlays ...

  ' For LCD: composite and send to display
  ' FRAMEBUFFER MERGE 0

  ' For VGA/HDMI: just copy framebuffer, layer composites automatically
  FRAMEBUFFER COPY F, N
LOOP
```

Raster Effects -- Moving Every Row or Column

Writing Games in PicoMite MMBasic

The arcade machines and home computers of the 1980s scrolled each line of the screen by its own amount, with a raster interrupt or a copper list changing the scroll register as the beam went down the screen. That is how a racing game bends its road into curves and hills, how a landscape scrolls in parallax bands, and how water ripples. Doing it in BASIC, one BLIT per line, is far too slow: 240 lines are hundreds of interpreted statements every frame.

On the hardware that did it this is called row scroll or line scroll, and column scroll for columns (the names of the arcade video chips, and of MAME); demo coders call a logo wobbled sideways line by line a tech-tech, and the racing games' road a pseudo-3D road.

From V8, the VGA and HDMI builds (MODE 2 to 5) have two commands that do the whole job in one statement each:

```
FRAMEBUFFER WARPH from, to, xoff%() [, step] [, first] [, phase] [, edge] [, transparent] [, rows%()]
FRAMEBUFFER WARPV from, to, yoff%() [, step] [, first] [, phase] [, edge] [, transparent]
```

Both copy buffer from to buffer to (N, F, L, and on the RP2350 also 2 and T; the two must differ). WARPH moves each row sideways by its own offset; WARPV moves each column vertically. The offsets come from a one-dimensional integer array, and the trick that makes it fast is that a program never loops over the lines: it updates the whole array with one MATH command, and the warp moves every line in C.

The Arguments

- xoff%() / yoff%() -- one element for each band. A positive offset moves a row right (WARPH) or a column up (WARPV), as SPRITE SCROLL does. Any value is allowed: a camera position that keeps growing simply wraps round.
- step (default 1) -- how many rows or columns each element covers. With step 2 a 240-line screen needs only 120 elements. The last band is cut off at the edge of the screen.
- first (default 0) -- the first row or column moved. The array's length decides how many bands there are, so rows before first and after the last band are left alone: a score line can stay still at the top or the bottom.
- phase (default 0) -- band b uses element (b + phase) MOD n. Count phase up each frame and a precomputed table (a sine wave, say) moves along with no other work. A table that holds a whole number of waves wraps round without a seam.
- edge -- what the pixels that nothing moves onto become. Omitted, the picture wraps round; -1 leaves them as they were in to; a colour fills them.
- transparent -- source pixels of this colour are not copied, so several warps can be stacked into one buffer (given as for FRAMEBUFFER MERGE: 0-15 in modes 2 and 3, 0-255 in mode 5, a colour in mode 4).
- rows%() (WARPH only) -- the same size as xoff%(): the source row for the first row of each band (the rest of the band follows on), or -1 to leave the band alone. This is how a road gets its hills, a lake its reflection, and a track its alternating kerb stripes.

Everything is checked before a pixel is written, with a precise message, for example Array has 241 elements, step 1 from row 0 fits 240.

Building the Tables with MATH

The offsets usually depend on one or two numbers that change each frame (a camera position, a curvature, a time). Work out once, outside the game loop, how much each row responds to that number, and let MATH do the rest:

What	How
Parallax: each row moves at its own speed	MATH SCALE speed!(), camx, xoff%()
Add a constant (sway, a second camera)	MATH ADD xoff%(), sway, xoff%()
Add two tables	MATH C_ADD a%(), b%(), xoff%()

Keep values in a range	MATH CLAMP t!(), -240, 0, yoff%()
Pick one of several precomputed tables	MATH SLICE tables%(), , k, rows%()
A moving wave of constant size	a sine table and the phase argument

MATH SCALE with a float array in and an integer array out rounds as it goes, which is exactly what a table of offsets wants.

Example: Parallax with a Fixed Score Line

The scenery is drawn once into F. Every row below the score line moves at its own speed, the far rows slowly and the near ones fast:

```
MODE 2
FRAMEBUFFER CREATE
FRAMEBUFFER WRITE F
' ... draw the scenery into F once ...
FRAMEBUFFER WRITE N
DIM speed!(111), xoff%(111)                ' 112 bands of 2 rows = rows 16-239
FOR i = 0 TO 111 : speed!(i) = -(0.1 + i / 112) : NEXT
DO
  camx = camx + 2
  MATH SCALE speed!(), camx, xoff%()       ' every row's offset, one statement
  FRAMEBUFFER WAIT
  FRAMEBUFFER WARPH F, N, xoff%(), 2, 16   ' rows 0-15 (the score) are untouched
LOOP
```

Example: A Lake that Reflects the Scene

Buffer 2 holds the landscape (sky, mountains, trees) in rows 0-149, and F holds the same picture drawn in the water's darker colours. The water (rows 150-239) shows F's rows upside down, rippling more the nearer they are to the viewer. The ripple is amp(k) * SIN(ph(k) + w * t), which splits into two precomputed tables, so each frame costs four MATH statements:

```
DIM mir%(89), wsn!(89), wcs!(89), wa!(89), wb!(89), ripple%(89), bg%(0)
FOR k = 0 TO 89
  mir%(k) = 149 - k                        ' water row 150+k shows scene row 149-k
  amp! = 0.3 + k * 0.075                   ' ripples grow towards the viewer...
  ph! = 9 * SQR(k + 1)                     ' ...and spread out
  wsn!(k) = amp! * SIN(ph!) : wcs!(k) = amp! * COS(ph!)
NEXT
DO
  t = t + 1
  bg%(0) = -t \ 4                          ' the scene pans slowly
  MATH SCALE wsn!(), COS(t * 0.12), wa!()
  MATH SCALE wcs!(), SIN(t * 0.12), wb!()
  MATH C_ADD wa!(), wb!(), wa!()
  MATH ADD wa!(), bg%(0), wa!()
  MATH SCALE wa!(), 1, ripple%()
  FRAMEBUFFER WAIT
  FRAMEBUFFER WARPH 2, N, bg%(), 150                    ' the scene: one band of 150 rows
  FRAMEBUFFER WARPH F, N, ripple%(), 1, 150, , , , mir%()  ' its reflection
LOOP
```

Example: A Road with Curves, Hills and Kerbs

This is the classic "pseudo-3D" road of the arcade racers.

- The picture: F holds a straight, flat road in perspective, drawn twice. Rows 120-239 have red kerbs, light grass and a lane line; rows 0-119 have white kerbs and dark grass. Row r of each copy is the road at one distance.
- Curves: the far rows move furthest, so xoff%() is a curvature times a fixed profile.
- Steering: the near rows move with the car.
- Hills: move the horizon, and squeeze or stretch the road rows into the space below it.
- Kerb stripes: each screen row picks the light or the dark copy of its road row, by which segment of the track that distance falls in. The stripes then stream towards the viewer.

```
DIM zz%(119), cp!(69), sp!(69), t1!(69), t2!(69), ro%(69), rm%(69)
FOR r = 0 TO 119 : zz%(r) = 2400 \ (r + 6) : NEXT    ' how far away road row r is
FOR k = 0 TO 69
  cp!(k) = (1 - k / 69) ^ 2 * 110                    ' a curve moves the far rows most
  sp!(k) = (k + 1) / 70                              ' steering moves the near rows most
NEXT
DO
  t = t + 1
  curve! = 1.3 * SIN(t * 0.009)
  camx! = curve! * 12
  hy = 120 + 16 * SIN(t * 0.013)                     ' the horizon rises and falls: hills
  dist = dist + 5                                    ' how far the car has gone
  MATH SCALE cp!(), curve!, t1!()
  MATH SCALE sp!(), -camx!, t2!()
  MATH C_ADD t1!(), t2!(), t1!()
  MATH SCALE t1!(), 1, ro%()
  FOR k = 0 TO 69                                    ' each band of 2 screen rows from row 100
    y = 100 + 2 * k
    IF y < hy THEN
      rm%(k) = -1                                    ' sky: leave it alone
    ELSE
      rr = MIN((y - hy) * 120 \ (240 - hy), 118)     ' the road row squeezed into the space
      rm%(k) = rr + 120 * (((zz%(rr) + dist) >> 4) AND 1)   ' light or dark copy
    ENDIF
  NEXT
  FRAMEBUFFER WAIT
  ' ... the sky and mountains first, e.g. a WARPH from buffer 2 panned by the curve ...
  FRAMEBUFFER WARPH F, N, ro%(), 2, 100, , , , rm%()
LOOP
```

The car itself lives in the layer L, drawn once: the VGA and HDMI output puts it over the road every frame for nothing.

Example: Melting a Picture Away

WARPV with each column falling at its own speed, and edge -1 so that the uncovered pixels keep whatever is underneath (here a blank screen, or the next picture):

```
FRAMEBUFFER COPY N, F                      ' the picture to melt
DIM st!(159), tm!(159), drop%(159)         ' 160 bands of 2 columns
FOR k = 0 TO 159 : st!(k) = RND * 40 : NEXT
FOR fr = 1 TO 80
  MATH ADD st!(), -0.35 * fr * fr, tm!()   ' each column waits a little, then falls faster
  MATH CLAMP tm!(), -MM.VRES, 0, drop%()   ' (negative is down)
```

```
  FRAMEBUFFER WAIT
  CLS
  FRAMEBUFFER WARPV F, N, drop%(), 2, , , -1
  IF MATH(MAX drop%()) <= -MM.VRES THEN EXIT FOR
NEXT
```

Timing

Measured on a PicoComputer 3 at 315 MHz, for the whole 320x240 screen:

MODE 2 (16 colours)	MODE 5 (256 colours)
FRAMEBUFFER COPY, for comparison	0.08 ms
WARPH	0.9 ms
WARPH with a transparent colour	2.9 ms
WARPV, step 1	4.7 ms
WARPV, step 8	1.6 ms

In MODE 5 the framebuffers are in PSRAM, which sets the speed. A frame at 640x480 and 75 Hz is 13.3 ms, so two to four warps a frame run at the full frame rate in MODE 2.

Tips

- Call FRAMEBUFFER WAIT first when writing to N. Both commands write the destination from the top down, so they stay ahead of the beam.
- MAP SET waits for the frame too. If you change the palette in a frame, use MAP SET instead of FRAMEBUFFER WAIT, not as well, or every such frame costs two.
- One buffer can hold many strips. With rows%() the source rows can be anywhere, so a single framebuffer can hold the sky, the mountains, a second copy of the road and a text strip at once.
- A warp is also a compositor. FRAMEBUFFER WARPH L, F, z%(), MM.VRES, , , , 0, with z%(0) = 0, puts the layer over F, transparent pixels and all.
- Two warps make a 2D wave. A WARPH into buffer 2, then a WARPV from 2 into N, waves text both ways.
- The demo warp_demo.bas (in the firmware repository's Bas folder) shows all of this at the full frame rate: a starfield with a waving logo and a sine scroller, a lake from day to dusk, the road, and a melt between each.

Approach 1: Sprite-Based Games

The SPRITE engine is ideal for arcade-style games with individual moving objects -- shooters, platformers, Pong, Breakout, top-down RPGs, and similar. It provides hardware-accelerated sprite management with automatic collision detection.

Key Capabilities

- Up to 64 sprites (numbered 1-64)
- Plus 64 static objects, numbered 1-64 in their own right - invisible collision rectangles that cost no sprite slot and no image memory, taking the number of things a sprite can collide with to 128
- 5 layers (0-4) for z-ordering
- Automatic collision detection: sprite-to-sprite, sprite-to-edge, sprite-to-static-object

- Background preservation: sprites save and restore the background beneath them
- Rotation and mirroring (8 orientations)
- Batch movement via SPRITE NEXT / SPRITE MOVE
- Background scrolling with wrap-around (SPRITE SCROLL)

Loading Sprites

Sprites can be loaded from several sources:

```
' From a .spr text file (compact, easy to edit)
SPRITE LOAD "player.spr", 1

' From a BMP image
SPRITE LOADBMP #1, "hero.bmp"

' From a PNG with alpha (RP2350 only)
SPRITE LOADPNG #1, "hero.png"

' From an array (procedurally generated)
DIM INTEGER pixels%(255)
' ... fill array ...
SPRITE LOADARRAY #1, 16, 16, pixels%()

' Capture from screen
SPRITE READ #5, 100, 100, 32, 32
```

Sprite file format (.spr): A simple text format where the first line defines width, count [, height] and each sprite is defined row-by-row using hex colour characters (0-9, A-F). Spaces are transparent. Lines starting with ' are comments.

Creating Copies

If you need multiple instances of the same sprite (e.g., a swarm of enemies), use SPRITE COPY to share image data efficiently:

```
SPRITE LOAD "enemy.spr", 1        ' Load the template
SPRITE COPY #1, #2, 9             ' Create 9 copies: sprites #2 through #10
```

Displaying and Moving

```
' Show a sprite at position (100, 50) on layer 1
SPRITE SHOW #1, 100, 50, 1

' Move it (simple approach)
SPRITE SHOW #1, 110, 50, 1        ' Redraws at new position

' Move with overlap safety (slower but correct when sprites overlap)
SPRITE SHOW SAFE #1, 110, 50, 1

' Batch movement (most efficient for multiple sprites)
SPRITE NEXT #1, 110, 50
SPRITE NEXT #2, 200, 100
SPRITE NEXT #3, 50, 150
SPRITE MOVE                        ' All three move simultaneously
```

Animation via Sprite Swapping

The most efficient animation technique is SPRITE SWAP -- load multiple animation frames into separate buffers, then swap between them:

```
SPRITE LOAD "walk.spr", 1         ' Loads multiple frames into buffers 1, 2, 3, 4
SPRITE SHOW #1, 100, 100, 1       ' Display frame 1

' In game loop:
frame% = (frame% MOD 4) + 1
SPRITE SWAP #displayed%, #frame%   ' Swap to next animation frame
```

Collision Detection

The sprite engine provides three types of collision detection:

1. Sprite-to-Sprite Collisions

```
SPRITE INTERRUPT collision_handler

' In the interrupt handler:
collision_handler:
  who% = SPRITE(S)                    ' Which sprite triggered it?
  count% = SPRITE(C, #who%)           ' How many collisions?
  FOR i% = 1 TO count%
    other% = SPRITE(C, #who%, i%)     ' What did it hit?
    IF other% < &h80 THEN
      PRINT "Hit sprite"; other%
    ENDIF
  NEXT i%
  IRETURN
```

2. Screen Edge Collisions

```
edges% = SPRITE(E, #1)
IF edges% AND 1 THEN PRINT "Hit left edge"
IF edges% AND 4 THEN PRINT "Hit right edge"
```

3. Static Object Collisions -- invisible rectangular trigger zones:

```
' Define walls and platforms
SPRITE STATIC #1, 0, 0, 320, 10        ' Top wall
SPRITE STATIC #2, 0, 230, 320, 10      ' Bottom wall
SPRITE STINTERRUPT wall_hit

wall_hit:
  spr% = SPRITE(ST, COLLISION)          ' Which sprite hit?
  obj% = SPRITE(ST, OBJECT)             ' Which static object?
  IRETURN
```

Static objects are a separate pool of 64, numbered 1 to 64 independently of the sprites, so a game that needs more collidable things than the sprite engine offers can have 128 of them. They are much the cheaper half: a static object is four numbers, with no image, no buffer and no background to save and restore, so it costs neither a sprite slot nor any of the memory a sprite costs.

Reach for them whenever the thing being collided with is already being drawn by something else. Level geometry that a tilemap or a piece of background art is already putting on the screen does not need a sprite - it needs a rectangle in the

same place. The same goes for anything deliberately invisible: doorways, ladders, damage zones, the boundary that turns an enemy round at the end of its platform.

Two details worth knowing. They scroll with the background, as layer 0 sprites do, so the geometry of a scrolling level stays where it belongs without your moving each one. And they come back through the same collision list as sprites - which is what the &h80 test in the sprite-to-sprite handler above is really distinguishing. A collision code of &h80 to &hBF is a static object, and the object number is the code AND &h3F:

```
other% = SPRITE(C, #who%, i%)
IF other% < &h80 THEN
  PRINT "hit sprite"; other%
ELSE
  PRINT "hit static object"; other% AND &h3F
ENDIF
```

SPRITE STATIC #n, OFF removes one and SPRITE STATIC CLEAR removes all of them, so changing level is a clear and a rebuild rather than bookkeeping.

4. Background Pixel Collision -- tests sprite pixels against the background:

```
result% = SPRITE(B, #1)
IF result% = 2 THEN
  ' Pixel-level collision with background
  push_left% = SPRITE(B, #1, 4)        ' Penetration from left
  push_right% = SPRITE(B, #1, 5)       ' Penetration from right
ENDIF
```

Utility Functions

```
SPRITE(X, #n)       ' X position (-10000 if not displayed)
SPRITE(Y, #n)       ' Y position
SPRITE(W, #n)       ' Width in pixels
SPRITE(H, #n)       ' Height in pixels
SPRITE(L, #n)       ' Layer (-1 if not displayed)
SPRITE(D, #n1, #n2) ' Distance between two sprites
SPRITE(V, #n1, #n2) ' Angle from sprite #n1 to #n2 (radians)
SPRITE(N)            ' Number of displayed sprites
```

Background Scrolling

For side-scrolling games, SPRITE SCROLL moves the background and all layer-0 sprites together:

```
SPRITE SCROLL 2, 0        ' Scroll right by 2 pixels, wrap-around (no fill colour)
' Layer 0 sprites scroll with background
' Layers 1-4 remain fixed (good for HUD elements)
```

For rows that each scroll at their own speed (parallax bands, a rippling reflection, a curving road) use FRAMEBUFFER WARPH - see Raster Effects above.

Approach 2: Tile Map Games

The TILEMAP engine (RP2350 only) is designed for 2D scrolling worlds -- platformers, top-down adventure games,

puzzle games, and anything with a tile-based world. It provides hardware-accelerated rendering of large maps with sub-tile smooth scrolling.

Why Use TILEMAP?

Drawing a 320x240 viewport with 16x16 tiles requires rendering ~336 tiles per frame. Doing this from BASIC with individual BLIT calls would be far too slow. TILEMAP DRAW does it all in a single C-level call with automatic viewport clipping and sub-tile scrolling.

Setting Up a Tile Map

Step 1: Prepare a tileset image -- a BMP with all tiles arranged in a grid (e.g., 256x64 pixels = 64 tiles of 16x16):

```
FLASH LOAD IMAGE 1, "tileset.bmp"
```

Image slots 1-3 are flash. On a board with PSRAM, slots 4-8 are the RAM slots 1-5: the same command loads them, BLIT FLASH, TILEMAP and MM.INFO(FLASH ADDRESS) take the same slot number, and nothing else changes. A RAM image loads faster and wears no flash, but it is cleared at every reset, so load it at every start (with O, since the slot may still hold it after a RUN). Use a constant for the slot number and the choice is one edit.

Step 2: Define the map in DATA statements:

```
TILEMAP CREATE mapdata, 1, 1, 16, 16, 16, 20, 15
' Parameters: label, slot, flash_slot, tileW, tileH, tiles_per_row, cols, rows
END

mapdata:
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
' ... more rows ...
DATA 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

Tile index 0 is always empty (transparent). Indices 1+ reference tiles from the tileset image.

Step 3: Define tile attributes for collision:

```
CONST SOLID = &b0001
CONST LADDER = &b0010
CONST DAMAGE = &b0100
CONST COLLECT = &b1000

TILEMAP ATTR attrdata, 1, 6
END

attrdata:
DATA 0            ' Tile 1: passable
DATA SOLID        ' Tile 2: solid brick
DATA SOLID        ' Tile 3: solid stone
DATA LADDER       ' Tile 4: ladder
DATA COLLECT      ' Tile 5: coin
DATA DAMAGE       ' Tile 6: spikes
```

Rendering the Map

```
' Draw the visible viewport
```

```
TILEMAP DRAW 1, F, camX, camY, 0, 0, 320, 240

' With transparency (for overlay layers)
TILEMAP DRAW 1, F, camX, camY, 0, 0, 320, 240, 0
```

The viewport coordinates (camX, camY) provide pixel-precise smooth scrolling. Partially visible tiles at edges are automatically clipped.

Camera Control

```
' Follow the player
camX = playerX - 160
camY = playerY - 120

' Or use built-in scroll with clamping
TILEMAP SCROLL 1, 2, 0       ' Scroll right 2 pixels (auto-clamps to bounds)
camX = TILEMAP(VIEWX 1)
camY = TILEMAP(VIEWY 1)

' Or set absolute position
TILEMAP VIEW 1, playerX - 160, playerY - 120
```

Tile-Based Collision Detection

The TILEMAP engine includes attribute-based collision -- define what each tile type means (solid, ladder, collectible, etc.) and test rectangular regions against the map:

```
' Movement with solid-tile blocking
CONST SOLID = &b0001
newX% = px% + speed%
IF TILEMAP(COLLISION 1, newX%, py%, 14, 22, SOLID) = 0 THEN
  px% = newX%        ' No solid obstacle -- allow movement
ENDIF

' Gravity with landing detection
newY% = py% + velY%
IF TILEMAP(COLLISION 1, px%, newY%, 14, 22, SOLID) = 0 THEN
  py% = newY%        ' Falling
ELSE
  velY% = 0          ' Landed
  py% = (newY% \ 16) * 16 - 22   ' Snap to tile boundary
ENDIF

' Collectible pickup
t% = TILEMAP(TILE 1, px% + 8, py% + 12)
IF t% > 0 AND (TILEMAP(ATTR 1, t%) AND COLLECT) THEN
  TILEMAP SET 1, (px% + 8) \ 16, (py% + 12) \ 16, 0   ' Remove coin
  score% = score% + 100
ENDIF
```

TILEMAP Sprites

The TILEMAP system includes its own lightweight sprite layer for game entities. These sprites reference tiles from the

tilemap's tileset and are rendered in batch for performance:

```
' Create player sprite using tile 7 from tilemap 1's tileset
TILEMAP SPRITE CREATE 1, 1, 7, 160, 120

' Move it
TILEMAP SPRITE MOVE 1, playerX%, playerY%

' Animate by changing tiles
IF frame% MOD 10 < 5 THEN
  TILEMAP SPRITE SET 1, 7     ' Walk frame A
ELSE
  TILEMAP SPRITE SET 1, 8     ' Walk frame B
ENDIF

' Draw all sprites in one call
TILEMAP SPRITE DRAW F, 0      ' F = framebuffer, 0 = transparent colour

' Check sprite-sprite collision
IF TILEMAP(SPRITE HIT 1, 2) THEN PRINT "Player hit enemy!"
```

Parallax Scrolling

Use multiple tilemaps with different scroll rates for a parallax effect:

```
FLASH LOAD IMAGE 1, "bg_tiles.bmp"
FLASH LOAD IMAGE 2, "fg_tiles.bmp"

TILEMAP CREATE bgdata, 1, 1, 16, 16, 8, 50, 15    ' Background
TILEMAP CREATE fgdata, 2, 2, 16, 16, 16, 100, 15   ' Foreground

' In game loop:
TILEMAP DRAW 1, F, camX\2, camY\2, 0, 0, 320, 240          ' BG at half speed
TILEMAP DRAW 2, F, camX, camY, 0, 0, 320, 240, 0            ' FG with transparency
TILEMAP SPRITE DRAW F, 0                                      ' Sprites on top
```

A background that repeats across the width of the screen (sky, hills, a skyline) need not be drawn every frame at all: draw it once into a framebuffer and let FRAMEBUFFER WARPH move every band of rows at its own speed, as many parallax layers as there are rows - see Raster Effects.

Dynamic Maps

Modify the map at runtime for interactive elements:

```
TILEMAP SET 1, col%, row%, 0       ' Remove a tile (e.g., collected coin)
TILEMAP SET 1, col%, row%, 3       ' Place a tile (e.g., close a door)
```

Memory Efficiency

Map data is stored internally as uint16_t (2 bytes per cell), far more efficient than using BASIC integer arrays (8 bytes each). A 200x30 map uses only 12 KB.

Approach 3: First-Person 3D (Raycaster)

The RAY engine (RP2350 only) provides a Wolfenstein 3D-style first-person renderer with textured walls, patterned floors/ceilings, billboard sprites, sliding doors, and built-in collision detection.

Quick Start

```
MODE 2
CLS

DIM m$(3) LENGTH 4
m$(0) = "1111"
m$(1) = "1001"
m$(2) = "1001"
m$(3) = "1111"

FRAMEBUFFER CREATE
FRAMEBUFFER WRITE F

RAY MAP 4, 4, m$()
RAY CAMERA 2.5, 2.5, 0, 66
RAY COLOUR 12, 3, 8, 1, 1, 3    ' Floor/ceiling colours and patterns

DO
  RAY RENDER                     ' Render the 3D scene
  FRAMEBUFFER COPY F, N          ' Display
  k$ = INKEY$
  IF k$ = "w" THEN RAY MOVE 0.1
  IF k$ = "s" THEN RAY MOVE -0.1
  IF k$ = "a" THEN RAY TURN -5
  IF k$ = "d" THEN RAY TURN 5
LOOP UNTIL k$ = CHR$(27)

RAY CLOSE
FRAMEBUFFER CLOSE
```

Map Definition

Maps use a grid where each cell is either empty (0) or a wall type (1-31). The string-array format uses just 1 byte per cell:

```
DIM m$(50) LENGTH 57
m$(0) = "11111111111111111111111111111111111111111111111111111111"
m$(1) = "10000000000000000000000000001000000000000000000000000001"
' ... etc ...
```

Characters 0=empty, 1-9=wall types 1-9, A-Z=wall types 10-35.

Wall Customisation

Each wall type has configurable appearance using fill patterns:

```
RAY DEFINE 1, 8, 10, 3        ' Type 1: red/rust, pattern 3
```

```
RAY DEFINE 2, 6, 4, 5         ' Type 2: green/midgreen, pattern 5
RAY DEFINE 7, 1, 3, 6, 1      ' Type 7: blue door (door flag=1)
```

Movement with Collision

RAY MOVE handles collision detection automatically -- the player slides along walls:

```
RAY MOVE 0.15                  ' Move forward
RAY MOVE -0.15                 ' Move backward
RAY MOVE 0.15, 0.1             ' Forward + strafe right
RAY TURN 5                     ' Rotate 5° clockwise
```

Sliding Doors

Up to 8 doors can animate simultaneously:

```
' Animate opening over multiple frames
door_offset! = 0.0
DO WHILE door_offset! < 1.0
  door_offset! = door_offset! + 0.1
  RAY DOOR door_x%, door_y%, door_offset!
  RAY RENDER
  FRAMEBUFFER COPY F, N
  PAUSE 50
LOOP
```

Billboard Sprites

Place 2D sprites in the 3D world -- they always face the camera and are depth-sorted automatically:

```
SPRITE LOADARRAY 1, 16, 16, enemy_img%()   ' Load sprite image
RAY SPRITE 0, 1, 5.5, 3.5                   ' Place at world position

' Move a sprite
RAY SPRITE 0, 1, newX!, newY!

' Remove
RAY SPRITE REMOVE 0
```

Ray Casting for Interaction

Cast a single ray to detect what the player is looking at -- useful for "use" buttons, shooting, or proximity checks:

```
RAY CAST RAY(CAMA)             ' Cast in camera direction
IF RAY(CASTDIST) < 2.0 THEN
  wall_type% = RAY(CASTWALL)
  cell_x% = RAY(CASTX)
  cell_y% = RAY(CASTY)
  ' Handle interaction...
ENDIF
```

Minimap Overlay

```
RAY RENDER
RAY MINIMAP 2, 2, 48           ' Draw minimap at (2,2), 48px wide
FRAMEBUFFER COPY F, N
```

Approach 4: 3D Object Graphics

The DRAW3D system provides quaternion-based 3D rendering for displaying and rotating solid objects -- useful for 3D puzzle games, object viewers, rotating logos, or in-game 3D elements.

Key Features

- Up to 8 objects, 3 cameras
- Quaternion rotation (no gimbal lock)
- Depth-sorted face rendering
- Surface normal hidden-face removal
- Configurable lighting with ambient levels
- Per-face colour, edge colour, and fill colour

Creating a 3D Object

Objects are defined by vertices, faces, and colours:

```
DIM vertex(7, 2) AS FLOAT       ' 8 vertices x 3 coordinates
DIM facecount(5) AS INTEGER     ' 6 faces (cube)
DIM faces(23) AS INTEGER        ' 6 faces x 4 vertices each
DIM colours(6) AS INTEGER       ' Colour palette
DIM linecolour(5) AS INTEGER    ' Edge colour per face
DIM fillcolour(5) AS INTEGER    ' Fill colour per face

' ... fill arrays with vertex/face data ...

3D CAMERA 1, 400                ' Set up camera (viewplane distance = 400)
3D CREATE 1, 8, 6, 1, vertex(), facecount(), faces(), colours(), linecolour(), fillcolour()
```

Rotation with Quaternions

```
DIM quat(4) AS FLOAT
angle! = RAD(10)
quat(0) = COS(angle!/2)         ' w
quat(1) = 0                     ' x
quat(2) = SIN(angle!/2)         ' y (rotate around Y axis)
quat(3) = 0                     ' z
quat(4) = 1                     ' magnitude

3D RESET 1                      ' Reset to original orientation
3D ROTATE quat(), 1             ' Apply rotation
3D SHOW 1, 0, 0, 200            ' Display at (0, 0, z=200)
```

Lighting

```
3D LIGHT 1, -100, 100, -50, 30  ' Light position + 30% ambient
3D SET FLAGS 1, 8, 0, 6         ' Enable lighting on all 6 faces
```

Drawing Primitives

MMBasic provides a complete set of drawing primitives that are essential for backgrounds, HUD elements, particle effects, and any custom rendering.

Available Primitives

Command	Description
PIXEL x, y, colour	Set/read individual pixels
LINE (x1,y1)-(x2,y2), colour, width	Lines with variable width
BOX x, y, w, h, linewidth, colour, fill	Rectangles
RBOX x, y, w, h, radius, colour, fill	Rounded rectangles
CIRCLE x, y, r, linewidth, colour, fill	Circles and ellipses
ARC x, y, r, start, end, colour	Arcs and wedges
TRIANGLE x0,y0,x1,y1,x2,y2, colour, fill	Triangles
POLYGON n, x(), y(), colour, fill	N-sided polygons
BEZIER x0,y0,x1,y1,x2,y2,x3,y3, colour	Bezier curves
FILL x, y, colour	Flood fill
TEXT x, y, str$, just, font, scale, fc, bc	Render text
CLS colour	Clear screen/buffer

Using Primitives for Game Graphics

Health bar:

```
BOX 10, 5, 100, 10, 1, RGB(WHITE), RGB(BLACK)      ' Border
BOX 11, 6, health%, 8, 0, RGB(RED), RGB(RED)        ' Fill
```

Particle effects:

```
FOR i% = 0 TO num_particles%
  PIXEL px%(i%), py%(i%), RGB(YELLOW)
  py%(i%) = py%(i%) + vy%(i%)
  vy%(i%) = vy%(i%) + 1                              ' Gravity
NEXT i%
```

Radar/minimap:

```
CIRCLE radar_x%, radar_y%, 30, 1, RGB(GREEN)
FOR i% = 0 TO num_enemies%
  dx% = enemy_x%(i%) - player_x%
  dy% = enemy_y%(i%) - player_y%
```

```
  PIXEL radar_x% + dx%/scale%, radar_y% + dy%/scale%, RGB(RED)
NEXT i%
```

BLIT Operations -- Fast Image Transfer

The BLIT commands provide fast block transfer of rectangular pixel regions between buffers. These are the workhorses for custom rendering pipelines.

Key BLIT Commands

Command	Purpose
BLIT FLASH slot, dst, sx, sy, dx, dy, w, h [, trans]	Draw from an image slot (flash 1-3, RAM 4-8) to buffer
BLIT FRAMEBUFFER src, dst, x1, y1, x2, y2, w, h [, trans]	Copy between F/L/N/T buffers
BLIT RESIZE src, dst, sx, sy, sw, sh, dx, dy, dw, dh [, trans]	Scale/resize between buffers
BLIT MERGE trans, x, y, w, h	Partial-area layer merge
BLIT LOAD #n, "file.bmp"	Load BMP into blit buffer

FLASH-Based Image System

For games that need to draw background images, tile-like elements, or UI graphics without an SD card at runtime, load BMPs into flash once and blit from flash during gameplay:

```
' One-time setup (load assets into flash)
FLASH LOAD IMAGE 1, "background.bmp"
FLASH LOAD IMAGE 2, "ui_elements.bmp"

' During gameplay -- blit from flash to framebuffer
BLIT FLASH 1, F, 0, 0, 0, 0, 320, 240              ' Full background
BLIT FLASH 2, F, 0, 0, 250, 5, 64, 16, 0           ' UI element with transparency
```

Installing Assets Once - and Knowing They Are Yours

FLASH LOAD IMAGE erases and rewrites a whole flash slot. That takes a noticeable moment and wears the flash, so a game that does it on every run pays for it every run. Write the slot once, then check it. (A RAM slot, 4-8 on a board with PSRAM, is the opposite case: it is cleared at every reset, so load it every run with O and skip the checks below.)

The check is built in: without OVERWRITE, FLASH LOAD IMAGE refuses with "Already programmed" when the slot is in use. A skipped error is therefore the program asking "has this already been done?":

```
SUB InstallImage(slot, file$)
  LOCAL e$
  ON ERROR CLEAR                  ' so a stale error cannot be mistaken for ours
  ON ERROR SKIP 1
  FLASH LOAD IMAGE slot, file$
  e$ = MM.ERRMSG$
  ON ERROR CLEAR
  IF e$ <> "" AND INSTR(e$, "Already programmed") = 0 THEN ERROR e$
END SUB
```

That answers "is something in the slot?" but not "is it mine?", and the difference matters more than it sounds. The user may have another game's images in those slots, and your own artwork changes between versions. Loading the wrong sheet does not crash - it draws a plausible-looking picture made of the wrong squares, which is a far more baffling bug report than a crash would be.

A flash image begins with its own dimensions, so start there:

```
LOCAL a
a = MM.INFO(FLASH ADDRESS slot)
IF PEEK(WORD a) <> wantW OR PEEK(WORD a + 4) <> wantH THEN
  ERROR "FLASH ERASE " + STR$(slot) + ", then run again"
ENDIF
```

But the same size does not mean the same picture. If a new version of your artwork renumbers the tiles without changing the sheet's dimensions, the size check passes and the map cheerfully indexes the old tiles. To catch that, compare a few words of the picture itself. What FLASH LOAD IMAGE stores is exactly determined by the BMP, so a build script can work out what those words must be and emit them as DATA:

Offset	Contents
0	image width, 32-bit
4	image height, 32-bit
8 onward	the picture, top row first, (width + 1) \ 2 bytes per row

Each byte holds two pixels with the left one in the low nibble. The nibble is the RGB121 code - red bit 7, green bits 7 and 6, blue bit 7 of the colour - which is not the BMP's own palette index, so a build script must put the BMP palette through the same conversion. Sample words from several places in the image, and skip any that come out blank: a run of zeroes matches any other image's blank run and proves nothing.

Note the ordering. BMPs store their rows bottom-up, but the flash image is written top row first, so the last row in the file is the first in flash.

Scaling with BLIT RESIZE

```
' Scale a 64x64 region up to 128x96
BLIT RESIZE F, L, 32, 16, 64, 64, 100, 40, 128, 96

' Scale with transparency
BLIT RESIZE F, L, 32, 16, 64, 64, 100, 40, 128, 96, 0
```

Input Handling

Games need responsive input. MMBasic offers several options depending on your hardware.

Keyboard Input

INKEY$ -- simple single-key polling:

```
k$ = INKEY$
IF k$ = "w" THEN move_forward
IF k$ = CHR$(27) THEN quit
```

KEYDOWN -- which keys are physically held right now, up to six at once.

The argument is a slot number, not a key code. KEYDOWN(n) takes n from 0 to 8 and anything outside that range is an error, so IF KEYDOWN(32) THEN Fire does not ask whether space is held - it fails. What comes back is the key code occupying that slot:

```
n% = KEYDOWN(0)                    ' how many keys are held, 0 to 6
FOR i% = 1 TO n%
  PRINT "held: "; KEYDOWN(i%)      ' the key code in slot i%
NEXT i%
```

argument	returns
0	how many keys are held, 0 to 6
1 to 6	the key code in that slot, 0 if the slot is empty
7	modifier bits: 1 left Alt, 2 left Ctrl, 4 left GUI, 8 left Shift, 16 right Alt, 32 right Ctrl, 64 right GUI, 128 right Shift
8	lock state: 1 caps, 2 num, 4 scroll

So asking "is the up arrow down?" means looking through the slots for its code. A game wants that answered for several keys every frame, so scan them once into an array instead - which is what the next section does.

Press versus hold. KEYDOWN reports what is held now, so acting on a key directly fires on every frame it is down. For actions that should happen once per press - jump, a single shot, opening a door, changing weapon - remember what was held last frame and require a release first:

```
DIM INTEGER kNow(255), kWas(255)

SUB ReadKeys
  LOCAL INTEGER i, n
  FOR i = 0 TO 255 : kWas(i) = kNow(i) : kNow(i) = 0 : NEXT i
  n = KEYDOWN(0)
  FOR i = 1 TO n : kNow(KEYDOWN(i)) = 1 : NEXT i
END SUB

FUNCTION Held(k)    : Held = kNow(k) : END FUNCTION
FUNCTION Pressed(k) : Pressed = kNow(k) AND kWas(k) = 0 : END FUNCTION
```

Call ReadKeys once at the top of the frame - never twice, or the second call overwrites the previous state and Pressed() never fires. Then use Held() for movement and thrust, Pressed() for everything else. Keep the whole key set in one array and the two behaviours stay one line apart rather than scattered through the game.

Do not mix INKEY$ and KEYDOWN for the same key. INKEY$ takes a keypress out of the console buffer; KEYDOWN reports the physical state of the keyboard. Consuming a key with INKEY$ does not stop KEYDOWN reporting it as still down until it is physically released, so a game that reads both will act on the same press twice. Pick one and use it throughout.

Testing note: writing key codes to the console over a serial link does not drive a KEYDOWN loop - a KEYDOWN call clears pending console input as a side effect. A game built on KEYDOWN can only be driven from a real keyboard, so plan another way to test it automatically.

USB Gamepad (USB Keyboard Builds)

Full USB gamepad support with up to 4 controllers, including PS4 special features:

```
' Set up gamepad interrupt
```

```
GAMEPAD INTERRUPT ENABLE my_handler

my_handler:
  ' Read gamepad state
  IRETURN

' PS4 rumble
GAMEPAD HAPTIC 1, 128, 255         ' Left motor half, right motor full

' PS4 light bar
GAMEPAD COLOUR 1, RGB(RED)
```

Wii Controllers via I2C

Nunchuk -- joystick + accelerometer + 2 buttons:

```
WII NUNCHUCK OPEN d_pin, c_pin
' Read joystick: variables set by OPEN
```

Classic Controller -- full gamepad with dual sticks and 12+ buttons:

```
WII CLASSIC OPEN d_pin, c_pin
```

IR Remote Control

```
IR dev%, cmd%, ir_handler

ir_handler:
  ' dev% and cmd% contain the received Sony SIRC code
  IRETURN
```

Touch Screen

For SPI LCD panels with touch controllers:

```
x% = TOUCH(X)
y% = TOUCH(Y)
IF TOUCH(DOWN) THEN handle_touch x%, y%
```

Mouse (USB Keyboard Builds)

```
MOUSE OPEN
MOUSE INTERRUPT ENABLE mouse_handler
```

Sound and Music

MMBasic provides comprehensive audio capabilities from simple beeps to multi-channel synthesis.

Simple Tones

```
PLAY TONE 440, 440, 200          ' 440Hz both channels, 200ms
```

Multi-Channel Synthesizer

Up to multiple simultaneous sound channels with selectable waveforms:

```
' Channel 1: sine wave, 440Hz, left speaker, volume 50
PLAY SOUND 1, L, S, 440, 50

' Channel 2: square wave, 220Hz, right speaker, volume 30
PLAY SOUND 2, R, Q, 220, 30

' Available waveforms: Sine, sQuare, Triangle, saWtooth, Noise, Pink, User, Off
```

The BBC Micro's SOUND and ENVELOPE

PLAY BBC SOUND and PLAY BBC ENVELOPE are the BBC Micro's own statements, with the same parameters and the same units. They are worth knowing even if the BBC means nothing to you, because what they give you is a four-channel sequencer: a note is queued with a duration and plays in its turn, so a tune is a run of statements rather than a timing problem your game loop has to solve.

```
PLAY BBC ENVELOPE 1, 1, 0,0,0, 0,0,0, 126, -50, 0, 0, 100, 0
PLAY BBC SOUND &H13, 1, 100, 1      ' channel 3, flushed, envelope 1, 1/20 s
```

- channel is a word, not a number: the low nibble picks channel 0-3, &10 flushes whatever that channel already has queued, &100/&200/&300 hold notes back so several channels start together.
- amplitude is -15 to 0 for a plain volume, or 1 to 16 to select an envelope, which then shapes the volume and can bend the pitch as it plays.
- pitch is in quarter semitones, 89 being the A above middle C.
- duration is in twentieths of a second, or 255 for "until something flushes it".

Three things to plan around:

Each channel queues eight notes, and a ninth statement waits. That is what paces a tune for free - a loop of PLAY BBC SOUND statements runs at the speed of the music with no timing code at all. It also means a sixteen-note jingle fired from inside your game loop stops the game dead halfway through. Keep a burst under eight notes, or split it around whatever else has to happen.

Channel 0 is the noise generator, and its pitch 3 and 7 are clocked by channel 1. A noise effect that wants a sweep gets it by playing a tone on channel 1 at the same time, which means channel 1 is not free for music while that effect is sounding.

Flushing is how effects retrigger, and how they cut each other off. An engine noise that reasserts itself every frame on a flushed channel will also silence an explosion started on that channel a moment earlier - the explosion never gets past its first frame. Decide which channel each class of effect owns, and if two must share, gate the quieter one.

Playing Audio Files

```
PLAY WAV "explosion.wav"
PLAY FLAC "music.flac"
PLAY MP3 "bgmusic.mp3"
PLAY MODFILE "chiptune.mod"        ' Amiga MOD tracker files!
```

MOD Files and Sound Effects with PLAY MODSAMPLE

Amiga MOD tracker files are an excellent choice for game background music -- they are compact, loop naturally, and have an authentic retro sound. MMBasic plays them via PLAY MODFILE, which uses 4 tracker channels to mix the music in real time.

The key advantage for game developers is PLAY MODSAMPLE, which lets you trigger individual samples from the loaded MOD file as one-shot sound effects while the music continues playing. MOD files contain up to 32 embedded instrument samples (explosions, bleeps, thuds, etc.), and you can play any of them on demand over the top of the background music using up to 4 dedicated sound-effect channels.

Syntax:

```
PLAY MODSAMPLE sample_number, channel [, volume]
```

Parameter	Description
sample_number	Sample number within the MOD file (1-32)
channel	Sound effect channel (1-4)
volume	Optional playback volume (0-64, default 63)

This means you can design your MOD file to serve double duty: the tracker patterns provide background music, while unused or dedicated samples in the same file provide your entire sound-effect library. No additional WAV files, no extra memory, no audio channel conflicts.

Practical example -- background music with sound effects:

```
' Start the background music
PLAY MODFILE "game.mod"

' Later, during gameplay:
PLAY MODSAMPLE 5, 1            ' Trigger sample 5 (e.g., laser) on effect channel 1
PLAY MODSAMPLE 8, 2, 40        ' Trigger sample 8 (e.g., coin) on channel 2 at volume 40
PLAY MODSAMPLE 12, 3           ' Trigger sample 12 (e.g., explosion) on channel 3
```

Tips for using MODSAMPLE in games:

- Compose your MOD file in a tracker (e.g., OpenMPT, MilkyTracker) with samples designated for sound effects -- put them in higher sample slots (e.g., 17-32) so they are easy to identify
- Use different effect channels for different sound categories (e.g., channel 1 for weapons, channel 2 for pickups, channel 3 for UI) so that rapid-fire sounds replace only their own category
- The sound effects mix with the MOD music automatically -- no additional code for mixing is needed
- If you trigger a new sample on a channel that is already playing an effect, the new sample replaces the previous one on that channel

Wavetable Synthesis with ADSR (RP2350)

For sophisticated sound effects, PLAY SAMPLE provides a wavetable synthesizer with per-channel ADSR envelopes:

```
' Define a sine waveform
CONST N = 256
DIM INTEGER wave%(N - 1)
FOR i% = 0 TO N - 1
  wave%(i%) = INT(SIN(2 * PI * i% / N) * 32000)
NEXT i%
```

```
' Play with ADSR envelope (attack=50ms, decay=100ms, sustain=60%, release=200ms)
PLAY SAMPLE wave%(), wave%(), 440, 50, 100, 60, 200

' Trigger release (note fades out)
PLAY RELEASE
```

Volume Control

```
PLAY VOLUME 80, 80                ' Left, right (0-100)
PLAY PAUSE                        ' Pause audio
PLAY RESUME                       ' Resume audio
```

Text UI with FRAME

The FRAME system provides a character-cell based panel system for HUDs, menus, inventories, and dialogue boxes. It works on both VGA displays and serial terminals.

```
FRAME CREATE
FRAME BOX 0, 0, 40, 3             ' Create a panel
FRAME TITLE 1, "STATUS"
FRAME PRINT 1, "HP: 100  MP: 50  Gold: 1234"
FRAME WRITE                        ' Render to screen
```

FRAME is especially useful for text-adventure style games, RPG interfaces, or overlaying text HUDs on graphical games.

Structuring a Game

Game Architecture Pattern

Most MMBasic games follow a common structure:

```
OPTION EXPLICIT

' ===== Constants =====
CONST SCREEN_W = 320
CONST SCREEN_H = 240

' ===== Initialisation =====
MODE 2
CLS
FRAMEBUFFER CREATE
FRAMEBUFFER WRITE F

' Load assets
SPRITE LOAD "player.spr", 1
SPRITE LOAD "enemies.spr", 10
```

```
' FLASH LOAD IMAGE 1, "tiles.bmp"
' TILEMAP CREATE ...

' Set up game state
DIM INTEGER score%, lives%, level%, gameOver%
DIM FLOAT playerX!, playerY!, playerVelX!, playerVelY!
playerX! = 160 : playerY! = 120 : lives% = 3

' Set up interrupts
' SPRITE INTERRUPT collision_handler
' SETTICK 16, game_tick              ' 60 FPS timer

' ===== Main Game Loop =====
DO
  ' --- Input ---
  ReadKeys                           ' see Input Handling: fills kNow()/kWas()
  IF Held(130) THEN playerVelX! = playerVelX! - 0.5
  IF Held(131) THEN playerVelX! = playerVelX! + 0.5
  IF Held(128) THEN playerVelY! = playerVelY! - 0.5

  ' --- Update ---
  playerX! = playerX! + playerVelX!
  playerY! = playerY! + playerVelY!
  playerVelX! = playerVelX! * 0.9    ' Friction
  playerVelY! = playerVelY! * 0.9
  ' ... update enemies, projectiles, physics ...

  ' --- Render ---
  CLS RGB(BLACK)
  ' Draw background / tilemap
  ' Draw sprites / entities
  ' Draw HUD
  TEXT 5, 5, "SCORE: " + STR$(score%), , 1, 1, RGB(WHITE), -1

  FRAMEBUFFER COPY F, N              ' Flip

  ' --- Frame timing ---
  PAUSE 16                           ' ~60 FPS cap

LOOP UNTIL gameOver%

' ===== Cleanup =====
SPRITE CLOSE ALL
FRAMEBUFFER CLOSE
END

' ===== Interrupt Handlers =====
' collision_handler:
'   ...
'   IRETURN
```

Performance Tips

1. Use MODE 2 (320x240) -- half the pixels of MODE 3, twice the speed

2. Use FRAMEBUFFER -- always double-buffer to avoid flicker
3. Use SPRITE NEXT/MOVE -- batch sprite movements are faster than individual SHOW calls
4. Pre-load assets into flash -- FLASH LOAD IMAGE eliminates SD card access during gameplay
5. Minimise BASIC loops -- use firmware-accelerated commands (TILEMAP DRAW, RAY RENDER, SPRITE MOVE) instead of per-pixel BASIC code
6. Use integer variables where possible -- suffix with % for faster maths
7. Use OPTION EXPLICIT -- catches typos and forces variable declaration
8. Use string maps for RAY -- 1 byte per cell vs 8 bytes for integer arrays
9. Use SPRITE COPY -- shared image data saves memory when you need many identical sprites
10. Use tile attributes -- one TILEMAP(COLLISION ..., mask) call replaces dozens of individual tile checks
11. Separate the simulation tick from the draw - run the game's logic at a fixed rate and draw whatever the clock allows, rather than letting the physics run as fast as the frame does. A game whose speed depends on how much is on screen is very hard to tune, and impossible to reproduce a bug in. Time the two halves separately: knowing that a slow frame is 0.3 ms of logic and 6.8 ms of drawing tells you immediately which one to attack.
12. Move the hot loop into a CSUB when interpretation dominates. Above a certain amount of per-object work, the cost is the interpreter reading statements, not the arithmetic in them - and no amount of tightening the BASIC helps. Rewriting one inner loop in C can be a hundredfold: a per-object physics update measured at 9.8 ms in BASIC ran in 0.28 ms as a CSUB doing exactly the same arithmetic. Keep the CSUB's state in an integer array shared with the BASIC so nothing has to be copied across the boundary each call. The mmb2csub tool (user-tools/mmb2csub.py, described in mmb2csub.pdf) does the conversion for you, from the BASIC routine itself.
13. Let one command move many lines. An effect that changes from line to line (parallax, a road, ripples, a melt) belongs in FRAMEBUFFER WARPH or WARPV, with its table made by MATH, not in a BASIC loop over the lines - see Raster Effects.

Finding the hot spots

Measure before you change anything. OPTION PROFILING ON at the top of the program prints a report at END: the commands executed most often, and the SUBs and FUNCTIONs called most and taking the most time. Profile one representative run, for example a fixed number of frames, and work on the top few entries only. option-profiling-cache.pdf has the full reference.

Earlier versions offered a statement cache, OPTION TRACECACHE with OPTION CACHE SUB and OPTION CACHE DEBUG, aimed at games. It was removed in V6.04.00 because it rarely paid for itself and could make programs slower. Delete those lines from existing programs: they now stop with "Invalid Option". The one part of it that helped everywhere, the DO loop fast path, is now automatic in every build.

Its replacement, and a far more powerful one, is the mmb2csub tool (user-tools/mmb2csub.py). Once the profile has named the one or two routines that take the time, mmb2csub compiles each of them to machine code and puts it back into your program as a CSUB, with the calls unchanged - typically 10 to 35 times faster for loops and arithmetic, where the trace cache gained a few percent at best. mmb2csub.pdf explains how to use it.

Memory Considerations

PicoMite has limited RAM. Budget carefully:

Resource	Typical Size
Framebuffer (320x240x4bpp)	~38 KB
Layer buffer	~38 KB
Tilemap (200x30)	12 KB
Sprite buffer (32x32)	~512 bytes

64 sprites (32x32 each)	~32 KB
Raycaster state	~6.5 KB
Flash images	Stored in flash, not RAM

Use FLASH LOAD IMAGE to keep large images in flash rather than RAM. Use SPRITE COPY for multiple instances of the same sprite. Use string-array maps for the raycaster.

Living within the heap

Everything above competes for one pool, and several of the biggest consumers are invisible in your source: the framebuffer, each DRAW3D object, the audio buffers. MM.INFO(HEAP) reports what is free and MEMORY breaks it down. Three habits make the difference between a game that fits and one that dies unpredictably.

Claim fixed-size buffers at start up, not on first use. A STATIC array inside a SUB is allocated the first time that SUB runs, which for a dashboard or a scratch buffer is somewhere in the middle of the first level - by which time the title screen's image has been and gone and the world is full of objects. The same array declared globally is claimed when the heap is empty and can never fail. It is the same memory either way; only the moment differs, and the late moment is the one that fails.

Never find a limit by running into it. The obvious way to discover how many objects you can afford is to create them until one fails and count. Do not: taking the heap to nothing takes the interpreter with it, and the error then surfaces from whatever statement runs next rather than from the allocation, so the ON ERROR SKIP you carefully put on the allocation never sees it. What you get is Not enough System Heap memory reported against an innocent line.

Price one unit instead and do the arithmetic, keeping a reserve back for everything still to come:

```
h0 = MM.INFO(HEAP)
Draw3D CREATE 1, nv, nf, 1, v(), fc(), f(), col(), ec()
cost = h0 - MM.INFO(HEAP)
Draw3D CLOSE 1
maxObj = (h0 - HEAPRESERVE) \ cost
```

The reserve is doing real work, not being timid: in the game this comes from, three objects at 4 KB each still left an in-flight floor of only 7.6 KB, so a fourth would have left 3.5 KB and a fifth none.

Let the allocation fail softly anyway. However good the sum, the moment an object is actually wanted is not the moment you measured, so guard the real call: if it fails, do without that object this frame - draw the ship as a dash on the scanner rather than a mesh - and bring the cap down to what the machine really has. The player sees slightly less detail; the alternative is the program stopping.

A heap error names the allocation that asked, not the one at fault. This is worth internalising because it will mislead you for hours. The pool is shared, so whichever allocation happens to ask when it is empty is the one reported - and it is usually an innocent bystander. In one case a 10 KB dashboard buffer was blamed three times running; the real fault was an object pool over-committed ten to one, and the buffer was simply the next thing to ask. If you suspect an allocation, move it to start up and see where the failure goes. If it moves, the thing you suspected was innocent.

Choosing the Right Approach

Game Type	Recommended Engine
Arcade shooter	SPRITE engine

Pong / Breakout	SPRITE engine + drawing primitives
Side-scrolling platformer	TILEMAP + TILEMAP SPRITE
Top-down RPG / adventure	TILEMAP + TILEMAP SPRITE
Wolfenstein / Doom-like FPS	RAY engine
3D puzzle / object viewer	3D graphics system
Text adventure / RPG menus	FRAME system
Board games / card games	Drawing primitives + BLIT
Rhythm game	PLAY SAMPLE + drawing primitives
Racing game with a curving, hilly road	FRAMEBUFFER WARPH + a layer for the car

Combining Systems

These systems can be combined. For example:

- TILEMAP + FRAME: A platformer with a text-based inventory overlay
- RAY + SPRITE: A first-person game with billboard sprites for enemies and items
- 3D + drawing primitives: A rotating 3D object with 2D HUD overlays
- TILEMAP + PLAY SOUND: A platformer with multi-channel sound effects
- Any game + PLAY MODFILE: Background chiptune music from MOD files

---

Example: Minimal Platformer Skeleton

```
OPTION EXPLICIT
MODE 2 : CLS

' --- Assets ---
FLASH LOAD IMAGE 1, "tiles.bmp"

CONST SOLID = &b0001
CONST COLLECT = &b1000
CONST GRAVITY! = 0.5
CONST JUMP_VEL! = -6.0

DIM INTEGER score%
DIM FLOAT px!, py!, vx!, vy!
DIM INTEGER onGround%
px! = 32 : py! = 100 : score% = 0

' --- Create tilemap ---
TILEMAP CREATE mapdata, 1, 1, 16, 16, 16, 20, 15
TILEMAP ATTR attrdata, 1, 4
TILEMAP SPRITE CREATE 1, 1, 3, px!, py!    ' Player sprite

FRAMEBUFFER CREATE
FRAMEBUFFER WRITE F

' --- Game Loop ---
DO
```

```
  ' Input
  ReadKeys                          ' see Input Handling: fills kNow()/kWas()
  IF Held(130) THEN vx! = -2
  IF Held(131) THEN vx! = 2
  IF Pressed(128) AND onGround% THEN vy! = JUMP_VEL!
  IF NOT Held(130) AND NOT Held(131) THEN vx! = vx! * 0.8

  ' Gravity
  vy! = vy! + GRAVITY!

  ' Horizontal movement with collision
  newX! = px! + vx!
  IF TILEMAP(COLLISION 1, INT(newX!), INT(py!), 14, 16, SOLID) = 0 THEN
    px! = newX!
  ELSE
    vx! = 0
  ENDIF

  ' Vertical movement with collision
  newY! = py! + vy!
  IF TILEMAP(COLLISION 1, INT(px!), INT(newY!), 14, 16, SOLID) = 0 THEN
    py! = newY!
    onGround% = 0
  ELSE
    IF vy! > 0 THEN onGround% = 1
    vy! = 0
  ENDIF

  ' Collectibles
  DIM INTEGER t%
  t% = TILEMAP(TILE 1, INT(px!) + 7, INT(py!) + 8)
  IF t% > 0 AND (TILEMAP(ATTR 1, t%) AND COLLECT) THEN
    TILEMAP SET 1, (INT(px!) + 7) \ 16, (INT(py!) + 8) \ 16, 0
    score% = score% + 100
  ENDIF

  ' Camera
  camX% = INT(px!) - 152
  camY% = INT(py!) - 112

  ' Render
  CLS 0
  TILEMAP DRAW 1, F, camX%, camY%, 0, 0, 320, 240
  TILEMAP SPRITE MOVE 1, INT(px!) - camX%, INT(py!) - camY%
  TILEMAP SPRITE DRAW F, 0
  TEXT 5, 5, "SCORE:" + STR$(score%), , 1, 1, RGB(WHITE), -1
  FRAMEBUFFER COPY F, N

  PAUSE 16

LOOP UNTIL INKEY$ = CHR$(27)

TILEMAP CLOSE
FRAMEBUFFER CLOSE
END
```

```
mapdata:
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,4,4,4,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0
DATA 0,0,0,0,2,2,2,0,0,0,0,0,2,2,2,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,2,2,0,0,0,0,0,0,2,2,0,0,0,0,0,2,2,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DATA 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

attrdata:
DATA SOLID        ' Tile 1: ground
DATA SOLID        ' Tile 2: platform
DATA 0            ' Tile 3: player graphic
DATA COLLECT      ' Tile 4: collectible
```

Porting an Existing Game

If you are bringing a game across from another machine rather than inventing one, the hardest part is not the graphics - it is knowing whether your version behaves like the original. A few habits make that tractable.

Get a reference you can run. A disassembly you read is not the same as an implementation you can execute. If you can run the original - in an emulator, or in a small interpreter for its CPU - you can put it side by side with your port and compare, which turns "does this feel right?" into a question with an answer.

Drive both sides with the same randomness. Record every random number the original draws, keyed by where it drew it, and feed those numbers to your port at the matching places. Otherwise the two diverge immediately for reasons that have nothing to do with your code being wrong.

Compare all the state, not just the obvious part. The temptation is to compare the things you already think about - the objects, the player. Everything else drifts in silence, and the bug surfaces hours later somewhere unrelated. In this project the object table was compared for weeks while a single timer elsewhere was never decremented, which meant no door in the game could open; the scenes that covered doors all passed. Widening the comparison to the whole of the shared state found that in one run, and three more bugs with it.

Prove that each check can fail. A comparison that has never failed may be watching nothing at all. Deliberately reintroduce a bug you have fixed and confirm the harness catches it, and where. This is the cheapest test you will ever write and it repeatedly finds checks that were quietly disabled.

Reaching the code is not exercising it. Coverage tells you a routine ran, not that it did anything. Three scenes reached the door-handling routine every time and all passed, while no door was capable of opening.

Where exactness is impossible, measure the gap instead of hiding it. Some original behaviour depends on things your port does not have - reading pixels back from the screen, for instance, or exact timing. Do not quietly let those cases pass. Count how often they agree, report the number, and write down why it cannot be 100%, so that a change making

it worse is visible rather than invisible.

The notes are not the code. A good disassembly comes with prose explaining what its author worked out, and that prose is the most useful thing in the repository - and it is not authoritative. In one of these ports the notes said the game ran at 50 frames a second, in four separate places. The code did not: the routine named for waiting on the vertical sync actually spins on the centisecond clock until it reads 3, so the frame is 30 ms and the game runs at 33.3 Hz. Every rate in the game - rotation, gravity, the periodic force update, the countdown - was derived from that one tick, so believing the notes made the port 1.5x too fast in every axis at once.

That class of error is invisible, which is what makes it expensive. Nothing looks wrong. No single behaviour is identifiably off, because they are all off by the same factor and stay in proportion to one another. What it feels like from outside is that the game is unreasonably hard, which is exactly how the first tester described it. If a port feels unfairly difficult and you cannot point at anything specific, suspect the frame rate before you suspect your physics.

Find the frame before you write anything else. Locate the routine the original loops on, work out from the instructions how long it really waits, and derive every rate in your port from that number. In the same port the notes gave the rotation as "about 8.4 steps per second"; the code is a three-frames-in-four test, which at the real rate is 25 steps a second. You cannot get that figure at all until you know what a frame is.

Behaviour that reads as a bug is usually faithful. You will find things that look obviously broken: a game that starts you with an empty tank, a refuelling test that only succeeds from one side, a pickup that requires you to move away from the thing you are collecting, a shield that burns fuel and protects you from nothing. Every one of those was in the original, and every one is part of how the game plays. Check the code before you "fix" anything. And when you keep a rule that surprises people, say so in the game - leaving without the cargo silently costs a life in the original, and a two-word warning in the status panel turned that from a stream of bug reports into a mechanic.

Transcribing a table proves nothing about the code that reads it. The sound in one port was audited by decoding every sound and envelope block out of the original binary and diffing it against what the port emitted. All of them matched byte for byte - and eight call sites were still wrong, because the blocks were only primitives. The original played almost none of them directly: it wrapped them in named routines, and every piece of behaviour lived in the wrappers. One of them muted the engine for a fixed number of frames after an explosion, and without it a player who was thrusting when they died cut their own explosion off after a single frame. Audit the callers, not just the constants.

A transcribed test still needs its inputs checked. That same audit missed a bug a player found in the first minute: the walking sound played continuously whether or not the character was moving. The test itself was a faithful transcription - "did he move?" - and what was wrong was the flag feeding it, which eight places in the update set for their own reasons, one of them firing every frame while he stood still on solid ground. Comparing our code against the original could never have found it, because the line that was wrong had no counterpart in the original at all. When you transcribe a condition, check that the values it tests are derived the way the original derives them - and prefer to measure state rather than flag it. "His position changed since last frame" cannot be set by accident in a branch that was thinking about something else.

Generate the source. These ports are a few thousand lines of MMBasic each, and most of that is DATA: level maps, sprite bitmaps, terrain runs, sound blocks, physics tables. None of it was typed. A short Python script per subsystem reads the original binary and prints the DATA statements, and a build.py splices them into the hand-written code and writes the finished .bas. When you discover the level format was wrong you re-run the generator instead of re-keying three hundred lines, and the generator is where you write down what you learned about the format.

One rule that costs an afternoon to learn: have the build write straight to the file you actually load. A build that lands in the tools directory beside the generator is a build nobody runs, and when the game on the board stubbornly fails to change you will look for the fault in your code long before you look for it in the path.

Shipping a Finished Game

Writing Games in PicoMite MMBasic

A game that only runs from the root of A: is awkward to hand to anyone else. Two firmware features let it live in a folder on any drive and start with a single RUN.

Finding your own assets

MM.INFO(PATH) is the directory the running program was loaded from. Use it for every asset path and the game works wherever it is installed:

```
DIM home$
home$ = MM.INFO(PATH)
IF home$ = "NONE" THEN home$ = "A:/"
OPEN home$ + "level1.dat" FOR INPUT AS #1
```

The path comes from a '#filename header the loader writes as the program's first line, so it is only available to a program loaded from a file. One that arrived over AUTOSAVE has no header and MM.INFO(PATH) returns "NONE", which is why the fallback is worth keeping while developing.

Three things that mechanism does not tell you, each of which is a bug that only shows up on somebody else's machine.

The current drive and directory have nothing to do with where your program was loaded from. A program loaded from B:/mygame/ starts with the current drive still set to whatever it was, usually A:, so a bare "level1.dat" does not resolve beside the program - it resolves somewhere you have never thought about. That is why every path needs the prefix rather than most of them, and it is why developing from the root of A: hides the fault completely: there, the two happen to be the same place.

Drive letters are not the same devices on every machine. On a PicoCalc A: is the internal flash and B: is the SD card; on other builds the roles and the sizes differ. A hardwired A:/ is not merely inflexible - on some machines it points at a different physical device from the one the user put your game on. Let MM.INFO(PATH) say which drive you are on instead of deciding for the user.

Everything the game writes needs the same treatment as everything it reads.

Saved games, high scores, screenshots and settings are easy to miss, because they are not "assets" and they are usually written from a different part of the code. A game installed on B: whose save file is hardwired to A:/save.dat works perfectly on the desk it was written on and fails for the first person who puts it anywhere else.

Two details. MM.INFO(PATH) includes its trailing slash, so home$ + "level1.dat" is right and home$ + "/level1.dat" is not. And anything optional should be asked about rather than assumed, so that a missing file degrades instead of stopping the game:

```
IF DIR$(home$ + "title.jpg", FILE) <> "" THEN LOAD JPG home$ + "title.jpg"
```

Code too big for program memory

Program memory is split into two halves of the same size - the program and the library - and a large game can outgrow the program half on its own, with no CSUB involved. The library is not only for code: every CONST, DIM and DATA block you move into it costs the program nothing, because the library's top level runs before your first line and its declarations are simply there when you start. A game whose declarations and generated tables ran to 33 KB moved all of them and bought back the whole of that from the program half.

That splits the build as well as the program, so let a script do it: keep the declarations in their own source file, have build.py emit both halves, and have it check that what it puts in the library is only declarations. The one rule to enforce is that nothing game-specific can go in the library that another program would choke on, because a library runs in front of every program on the machine, not just yours.

A large CSUB costs its hex text as well as its compiled code and can easily exceed program memory. Put it in the library instead - and have the program install its own, rather than relying on the user having run LIBRARY SAVE:

```
LIBRARY LOAD MM.INFO(PATH) + "mygame_lib.bas", RAM
OPTION EXPLICIT
' ... the rest of the program ...
```

RAM puts the library into RAM slot 5 in PSRAM on a board that has it (firmware 6.03.02b11 on), where it takes the place of the flash library until END or the next RUN. The user's own flash library is never touched, no question is asked, and a repeat load of the same file is immediate because the slot keeps the file's hash. Without PSRAM the keyword is ignored and the load goes to flash as described below.

Three rules come with it:

- It must be the program's first statement. The library's own top level runs before your program's first line, so one loaded later would arrive too late to be initialised. Comment lines ahead of it are fine, including the '#filename header the loader added.
- It is idempotent. The hash of the source is kept in the options, so a program doing this on every run reads the file, finds the library already matches, and touches no flash at all. Only a different library costs anything.
- It restarts the program. A library cannot be initialised mid-run, so when one is written execution begins again at line 1 - where the command is now a no-op. Your program sees a clean start, not a resumed one, so do not put anything before it that should happen once.

Do not add OVERWRITE in a released game. Without it the board asks before replacing a library that may belong to something the user cares about, and the question only appears when the library actually differs - so an ordinary run is silent anyway.

One RUN from nothing

Put that together with the flash-slot install above and the whole thing is automatic: the library writes itself and restarts, the flash slots fill if they are empty and are verified if they are not, and everything else loads from home$. The user copies one folder and types one command. The first run takes a minute; every run after that starts in a second.

Ship a plain-text README in the same folder. It is the only documentation most players will ever see, and it is the right place for the keys, the install command, and what to include in a bug report.

MMBasic Traps Worth Knowing

A few that are easy to hit and hard to diagnose:

The board remembers its last error. When a game dies while you are not watching the console - which is most of the time, because the console is usually the screen the game is using - the information is still there at the prompt afterwards:

```
PRINT MM.ERRNO          ' the number
PRINT MM.ERRMSG$        ' the text
PRINT MM.ERRLINE        ' the line, 1-based
PRINT MM.INFO(CURRENT)  ' which file it was
```

Those are cleared by RUN and by ON ERROR CLEAR, and by nothing else - not by PRINT, not by Ctrl-C. So a crash stays diagnosable long after the fact, and reading them is almost always quicker than reproducing the fault. Read them first, before you try anything that might run the program again and wipe them. The line number maps onto the built .bas if your build strips comments, which is another reason to have the build write the file you actually load.

RESTORE to a label, always. An unqualified READ takes the program's first DATA statement, wherever that happens to be. Add a generated DATA block ahead of an existing one and the old READ silently starts consuming the new numbers - which surfaces as a nonsense error a long way from the cause.

```
RESTORE LevelData
FOR i = 1 TO n : READ tile(i) : NEXT i
LevelData:
DATA 1, 2, 3, 4
```

TEXT with a background colour only erases its own width. Printing a shorter string over a longer one leaves the tail of the old one behind. Pad to a fixed width, or clear the area first.

Work out how many characters actually fit. Font 7 is 6 pixels wide, font 1 is 8. A 64-pixel status panel holds ten characters of font 7 and no more, so "SCORE 1000" fits and "SCORE 10000" does not - and the overflow is silent. Count before designing a panel, and leave headroom for the largest value a field can reach, not the value it shows at the start.

Names are unique irrespective of type suffix, and case does not separate them either. k and k$ are the same name, and so are FIREGAP and fireGap - which collides with the natural habit of naming a constant after the variable it initialises. CONST FIREGAP = 12 followed by a variable fireGap is a redeclaration, not a pair. Give the constant a different word, not a different case. A no-argument built-in function name cannot be used as a variable at all.

A SUB cannot be named after a command. SUB Play looks perfectly reasonable and then fails at every call site with "Unknown command", because PLAY is parsed as the statement it is. The same goes for any other keyword.

Poking at a running game from the console has two sharp edges. A program's CONST, DIM and OPTION EXPLICIT are statements: none of those names exist until the program has executed them. Call one of its SUBs from the prompt before any RUN and it runs with every constant reading as an undeclared zero, and without OPTION EXPLICIT there to object - so the call succeeds and quietly tests something other than what you meant. Start the program, let it reach its title screen and break out of it first. After that: variables cannot be invented at the prompt, the command line truncates around 250 characters, a FOR and its NEXT cannot span two entries, and MM.ERRNO is cleared only by a RUN, so a stale value from a typo three commands ago will sit there through every later test. Breaking out with Ctrl-C while a SUB is executing leaves that SUB's locals declared, which makes the next call behave differently from the first.

Further Reading

Each subsystem has a detailed reference manual available as a PDF:

Manual	Contents
SPRITE_User_Manual.pdf	Full sprite command/function reference, collision system, static objects, animation techniques
TILEMAP_User_Manual.pdf	Tile map creation, attributes, collision, sprites, parallax scrolling, performance notes
Raycaster_User_Manual.pdf	Raycaster setup, wall types, doors, sprites, minimap, coordinate system, technical internals
3D_Graphics_User_Manual.pdf	3D object creation, quaternion rotation, lighting, camera setup, complete cube example
BLIT_User_Manual.pdf	All BLIT commands: bare copy, READ/WRITE/CLOSE buffers, LOAD BMP, MERGE, RESIZE scaling, FLASH, FRAMEBUFFER copy, COMPRESSED, MEMORY
FRAME_User_Manual.pdf	Character-cell frame buffer, panels, box-drawing, text HUDs

PLAY_SAMPLE_User_Manual.pdf	Wavetable synthesis, ADSR envelopes, waveform generation
PLAY_BBC_User_Manual.md (in docs/)	The BBC Micro's SOUND and ENVELOPE: channel words, envelope parameters, queues, flushing, synchronised chords
Game_Input_Devices_Manual.pdf	Keyboard (INKEY$, KEYDOWN), USB gamepads (PS4/PS3/Xbox/Generic), Wii Nunchuck & Classic, GPIO buttons

Happy game making!