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

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
```

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, then FRAMEBUFFER MERGE to combine the layer with the framebuffer using a transparency colour. This is useful for overlaying HUD elements or particle effects.

```
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 ...
FRAMEBUFFER MERGE 0            ' Merge layer onto framebuffer (0 = transparent)
FRAMEBUFFER COPY F, N          ' Display result
```

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)
- 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
```

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, -2    ' Scroll right by 2 pixels, wrap-around
```

```
' Layer 0 sprites scroll with background
' Layers 1-4 remain fixed (good for HUD elements)
```

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"
```

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
```

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 flash image 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
```

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 -- simultaneous key detection (up to 7 keys):

```
IF KEYDOWN(128) THEN move_up       ' Up arrow
IF KEYDOWN(129) THEN move_down     ' Down arrow
IF KEYDOWN(130) THEN move_left     ' Left arrow
IF KEYDOWN(131) THEN move_right    ' Right arrow
IF KEYDOWN(32) THEN fire           ' Space
n% = KEYDOWN(0)                    ' Number of keys currently pressed
```

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
```

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 ---
  k$ = INKEY$
```

```
  IF KEYDOWN(130) THEN playerVelX! = playerVelX! - 0.5
  IF KEYDOWN(131) THEN playerVelX! = playerVelX! + 0.5
  IF KEYDOWN(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

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.

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

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
  IF KEYDOWN(130) THEN vx! = -2
  IF KEYDOWN(131) THEN vx! = 2
  IF KEYDOWN(128) AND onGround% THEN vy! = JUMP_VEL!
  IF NOT KEYDOWN(130) AND NOT KEYDOWN(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 0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0
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
```

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, an
TILEMAP_User_Manual.pdf	Tile map creation, attributes, collision, sprites, parallax scrolling, performan

Raycaster_User_Manual.pdf	Raycaster setup, wall types, doors, sprites, minimap, coordinate system, te
3D_Graphics_User_Manual.pdf	3D object creation, quaternion rotation, lighting, camera setup, complete cu
BLIT_User_Manual.pdf	All BLIT commands: bare copy, READ/WRITE/CLOSE buffers, LOAD BMP
FRAME_User_Manual.pdf	Character-cell frame buffer, panels, box-drawing, text HUDs
PLAY_SAMPLE_User_Manual.pdf	Wavetable synthesis, ADSR envelopes, waveform generation
BITSTREAM_User_Manual.pdf	High-speed bit-bang output (WS2812 LED strips for physical feedback)

Happy game making!