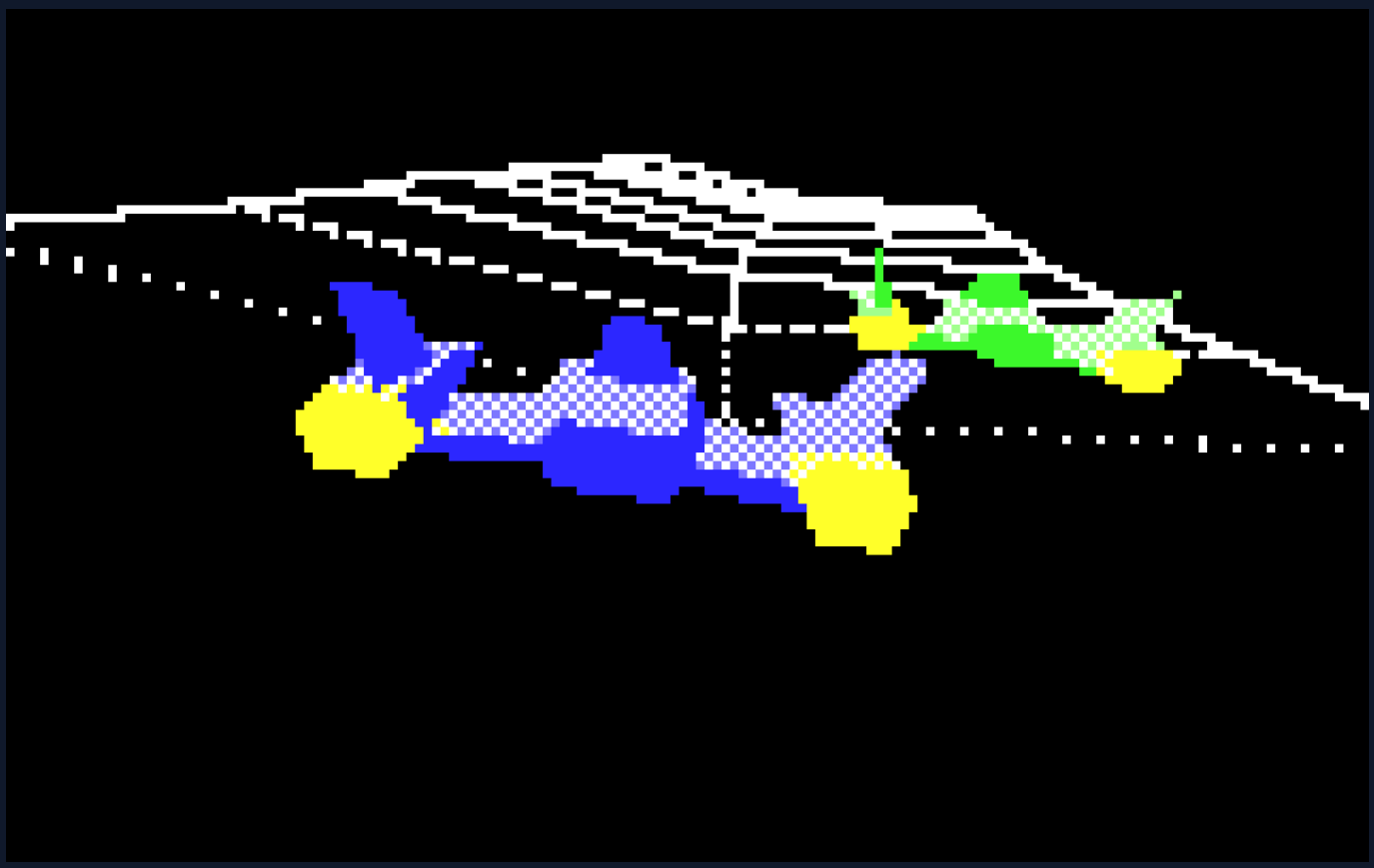
3Dvibe64

Building 3D worlds on the Commodore 64



Two spacecraft on a three-dimensional track. The program calculates the perspective and draws the scene as it runs.

From the stock C64 to the C64 Ultimate at 64 MHz.

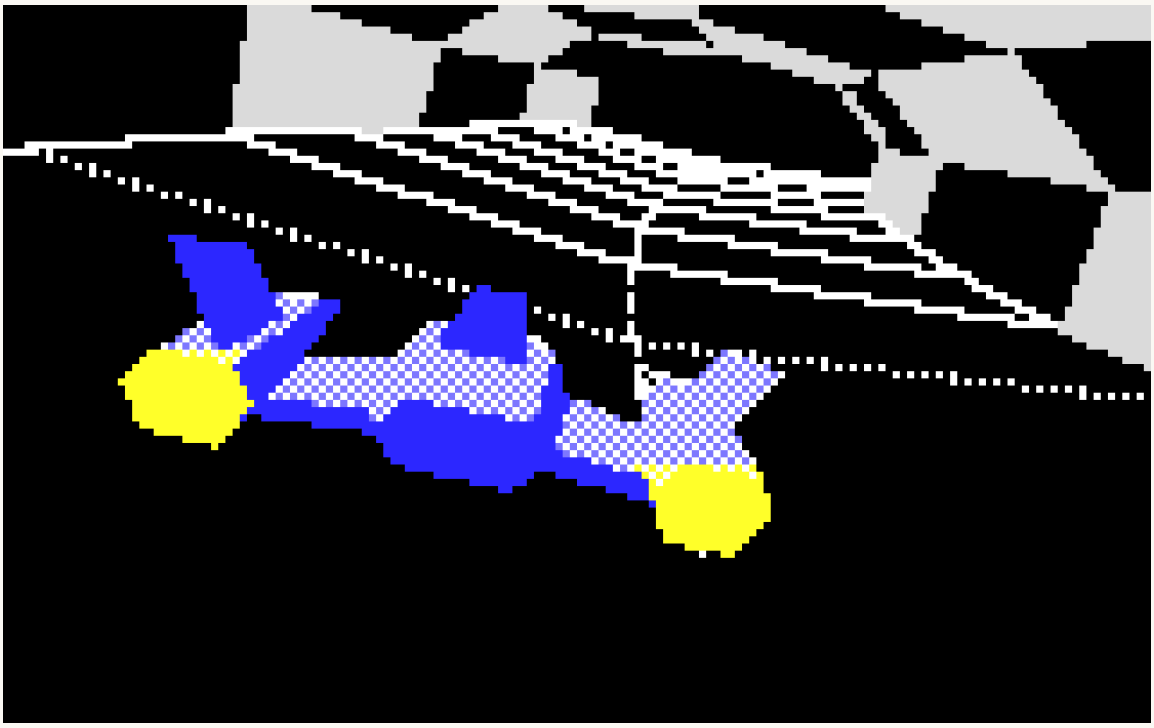
An illustrated guide to the engine’s features, rendering techniques and scene development.

Why build a 3D engine for the C64?

3Dvibe64 is an engine for creating three-dimensional scenes on the Commodore 64. It can display polygonal objects, move the viewpoint, and draw surfaces with lighting or textures. Geometry describes the scene: the program calculates each image from the positions of the objects and the observer, updating it as the scene runs.

A rotating cube provides a useful example of the work involved. The coordinates of its vertices change; some faces become visible while others disappear behind the object. The renderer must project the points onto the screen, determine which surfaces to draw, and fill them with the appropriate colors. The same process applies to a spacecraft, a room or a racetrack.

On the C64, these calculations are performed by the **CPU**, the central processor. The **VIC-II** video chip reads graphics data from memory and displays it on the screen. 3Dvibe64 provides tools for preparing scenes and routines for drawing them within the machine's limits on processing power, memory and color. These reusable functions form the foundation of the engine.



The spacecraft follows a track through bends and tunnels. The camera follows its movement while the renderer calculates each new view.

What a scene contains

A scene describes the objects to display, their positions, the appearance of their surfaces and the viewpoint. Building one means distinguishing an object’s shape from the way it is placed and drawn. These are the main terms used by the engine.

Term	Meaning
Vertex	A point in space, described by three coordinates. A cube has eight vertices, one at each corner.
Face or polygon	A surface bounded by vertices. A cube has six square faces.
Mesh	The collection of vertices and faces that describes an object’s shape.
Instance	An object that uses a particular mesh, with its own position, rotation and size.
Camera	The virtual viewpoint: it defines where we are looking from and in which direction.
Material	The rules that determine a surface’s color, response to light or texture pattern.

Reusing the same shape

Two spacecraft can have the same shape but different positions and colors. In rendering paths that support shared meshes, the original model is stored only once. The renderer keeps the calculated results for each spacecraft separate: the source geometry is shared, while each instance has its own position in the image.

> **FROM A DESCRIPTION TO AN IMAGE**
>
> The **renderer** is the part of the engine that draws the scene. One rendering pass produces a **frame**: a new image of the scene as seen by the camera. The **viewport** is the area of the screen reserved for that image.

From a scene file to a program

3Dvibe64 includes an **SDK**, a software development kit containing source code, tools and documentation. A scene is prepared on a PC as a JSON file: structured text containing names, values and lists. The **builder** reads this data and prepares the C64 program.

Assembly language expresses CPU operations as short, readable instructions. An assembler translates them into executable machine bytes. The resulting **PRG** file contains the program to load on the C64.

01	**Describe**	Geometry, camera, materials and movement define the contents of the scene.
02	**Prepare**	The builder checks the data, generates tables and selects the routines needed by the chosen profile.
03	**Assemble**	Assembly code is translated into machine instructions and packaged in a PRG file.
04	**Run**	The C64 loads the PRG and calculates new images while the program runs.

EXAMPLE BUILD USING A COMPATIBLE EXTERNAL SCENE

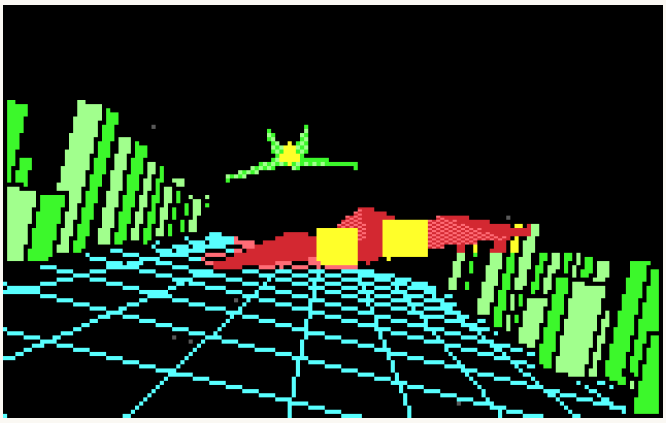
```
pwsh -NoProfile -File work/build-3Dvibe64.ps1 `
  -Precision q8 -GraphicsMode 7 `
  -TexturePrecision perspective `
  -SceneFile ../scene/example.json `
  -Q8Camera stationary -VideoStandard pal `
  -OutputDirectory ../build/example
```

Here, **q8** selects numerical precision, **GraphicsMode 7** selects the textured rendering path, and **perspective** selects perspective-correct sampling. The scene file must meet that profile’s requirements. Specialization lets the program devote its memory and code to the features it actually uses.

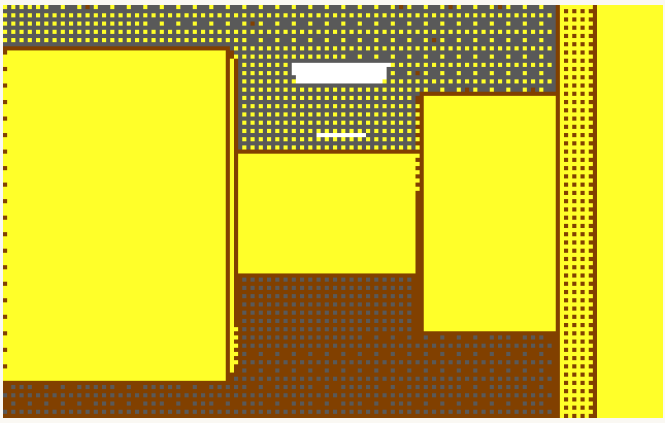
Different scenes, different demands



The labyrinth and character are drawn in the central window. Inventory slots and status indicators occupy the rest of the screen.



A spacecraft chase with a moving camera.



Exploring rooms connected by openings.

The flight scenes show how the renderer handles objects that rotate and overlap. In an interior, walls and floors occupy much of the view and require more surface filling. The demos also include features built around the renderer for each scene, such as the labyrinth interface or spacecraft controls.

The starting point: a stock C64

Stock means a C64 in its original configuration, without a CPU accelerator. Its 6510 processor runs at approximately 0.985 MHz in PAL systems and 1.02 MHz in NTSC systems. One megahertz means one million clock cycles per second; an instruction may take several cycles.

PAL and NTSC are families of television standards with different video timings. A **bit** holds either 0 or 1; eight bits form a **byte**. These are the basic units the program uses to represent numbers, colors and instructions.

Three resources to manage carefully

Processing time. The 6510 mainly operates on 8-bit values, one byte at a time. It has no native integer multiplication or division instruction. Common 3D calculations therefore require sequences of instructions or lookup tables containing results calculated in advance.

Memory. The 64 KB must hold code, geometry, tables, scene state and images. In this context, one kilobyte means 1,024 bytes. Data must also be placed in areas accessible to the CPU and VIC-II: the two chips do not necessarily see memory in the same way.

Memory access. The VIC-II must read graphics data at the rate needed to generate the video signal. At certain times, these accesses take cycles away from the CPU. The time available for rendering therefore also depends on the video chip’s activity.

> **WHAT THIS MEANS FOR A SCENE**
>
> Both geometric complexity and the area to be filled matter. A small, distant mesh may cover only a few pixels; the same mesh close to the camera may fill almost the entire screen. Drawing cost changes even when the vertex count stays the same.

On a stock C64, start with simple shapes and measure the cost of additions. An extra object, a closer wall or a more demanding texture can slow the scene in different ways. Recognizable silhouettes and distinct colors help produce clear images with relatively little detail.

What acceleration changes

An accelerator lets the CPU execute more instructions in the same amount of time. A program can use that capacity to draw more geometry, calculate with greater precision or complete more frames. The actual gain also depends on the available memory and the speed at which the processor can access it.

Configuration	Processing capability	What to consider
Stock C64	CPU at its original speed	Keep object counts and surface drawing costs within a small budget.
C64 with an accelerator	More instructions per second	CPU, fast memory and peripheral access depend on the device.
Ultimate 64 and Ultimate 64 Elite	FPGA motherboards, up to 48 MHz	Performance depends on configuration and bus access.
Ultimate 64 Elite-II	Newer FPGA motherboard, up to 64 MHz	More processing capacity, with video and I/O constraints.
C64 Ultimate	Complete FPGA-based computer, up to 64 MHz	Includes a case and keyboard; video and I/O still affect performance.

FPGA: recreating digital circuits

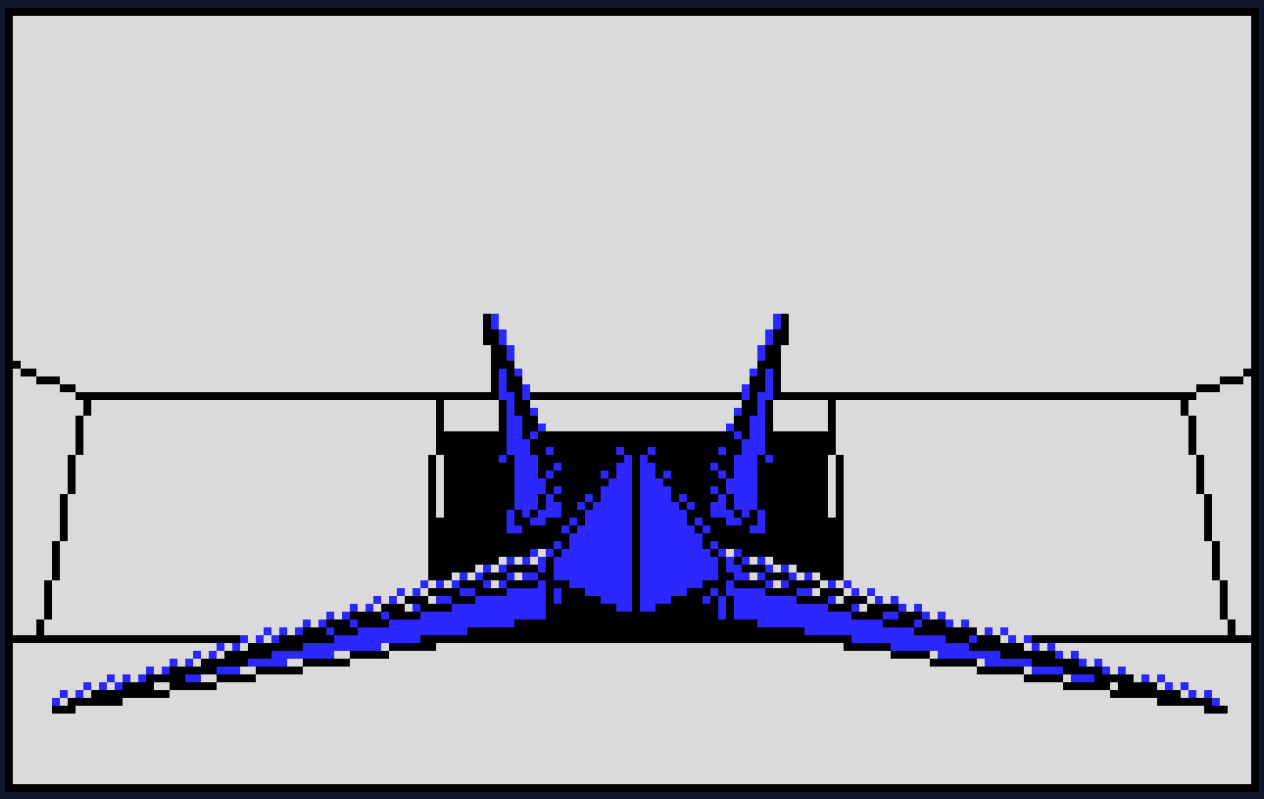
An **FPGA** is a chip that can be configured to implement digital circuits. In the C64 Ultimate, it recreates the computer’s architecture, including its CPU and devices. Accelerated operation increases the processor’s speed while retaining the C64 computing environment.

Ultimate 64 and Ultimate 64 Elite are replacement motherboards for a C64 case and keyboard. The Elite revision adds automatic voltage and filter selection for installed SID sound chips. Elite-II is a later board generation. **C64 Ultimate** names the complete computer sold by Commodore.

A 3Dvibe64 PRG contains 6502/6510 code. To understand its performance on accelerated hardware, consider the actual configuration: clock speed alone does not describe the speed of every operation, especially when video or peripherals are involved.

C64 Ultimate at 64 MHz

More instructions for geometry, lighting and textures.



The spacecraft inside the hangar. The walls define the space, while outlines separate the hull from the background.

The C64 Ultimate at 64 MHz is particularly relevant to 3Dvibe64 because rendering runs on the CPU. The higher speed makes more instructions available for vertex transformations, polygon clipping, lighting and texture sampling. Scenes and settings that would heavily tax the original processor become more practical.

The C64’s graphics characteristics remain: a 16-color palette, the rules of multicolor cells and the organization of video memory. Acceleration allows the program to calculate an image more quickly using those same building blocks.

The improvement varies between scenes. A surface with a perspective-correct texture needs many calculations for its pixels; a wireframe object mainly needs transformations and line drawing. Video waits and peripheral accesses also influence the result.

Clock speed, buses and frames

The **bus** is the set of connections through which components exchange data. Even with an accelerated CPU, the VIC-II must obtain the memory accesses needed for video. Peripherals and the external bus retain timing constraints imposed by the machine's design.

Badlines are periods when the VIC-II fetches screen-cell data. Core settings determine whether the CPU can continue working with internal addresses during these accesses. Comparing two performance runs requires keeping these settings the same.

SELECTING THE CPU CLOCK THROUGH A REGISTER

```
; C64 Ultimate has already been identified
; Register control enabled in the menu
lda $d031
and #$f0       ; preserve the other bits
ora #$0f       ; select speed index 15
sta $d031
```

With "U64 Turbo Registers" enabled, index 15 in register **$D031** selects 64 MHz on the C64 Ultimate. On the earlier Ultimate 64 boards, it selects 48 MHz. The $ symbol introduces a hexadecimal number; this example preserves the other configuration bits.

A refresh can repeat the same image

FPS measures how many new, complete views are produced in one second. The video hardware may display an image several times while the renderer prepares the next one. This is why 50 screen refreshes per second do not automatically mean 50 new 3D frames.

> **A COUNTING EXAMPLE**
>
> If a program presents 20 new images over 2 seconds on the target machine, its average is 10 FPS. Repeated refreshes do not increase that count. The captures in this guide illustrate the demos' appearance; they are not performance measurements from a physical C64 Ultimate.

From a point in space to a pixel

To draw an object, the renderer must convert mesh coordinates into screen coordinates. It first applies the object's position, rotation and scale, then calculates where its vertices lie relative to the camera. These steps, together with the drawing operations that follow, form the rendering **pipeline**.

01	**Transform**	Calculate vertex coordinates using the object's placement and the viewpoint.
02	**Project**	The ratio between a coordinate and depth determines its screen position. Distance reduces apparent size.
03	**Select**	Visibility tests and clipping discard or trim geometry that cannot contribute to the image.
04	**Draw**	The rasterizer traverses visible surfaces and writes their colors into the bitmap.

THE PERSPECTIVE RELATIONSHIP, IN CONCEPTUAL FORM

```
screen_x = center_x + focal_length * camera_x / depth
```

Imagine two points at the same sideways distance from the camera. If one is twice as far away in depth, its distance from the center of the screen is halved. This is what makes the edges of a track converge and objects appear smaller as they recede.

The **focal length** controls the field of view and the apparent size of objects. The engine performs these calculations with integers. In scene coordinates, X points right, Y points forward and Z represents height. Depending on the profile, some visibility checks and clipping operations happen before projection and others after it.

Fractions without floating point

Moving a camera by a quarter of a unit requires a way to represent fractions. 3Dvibe64 can do this with **fixed-point arithmetic**: it stores an integer and assigns some of its bits to the fractional part. The processor still works with integers, but those values can describe movements smaller than one unit.

Format	How it stores 3.5	Step between values
Q4	3.5 × 16 = 56	1/16 = 0.0625
Q8	3.5 × 256 = 896	1/256 = 0.00390625

The program stores 56 or 896 as an integer. The format convention determines where the fractional point belongs. After some operations, the result must be brought back to the correct scale, for example with a **shift**, which moves the bits left or right.

What the extra precision provides

More fractional bits let transformations and intersections represent smaller changes. This helps when a surface approaches the camera or a trajectory needs smooth increments. The screen still has its own resolution: calculation precision and pixel count are separate properties.

What the extra precision costs

The 6510 needs more instructions to process wider values, handle carries and stay within valid ranges. Q8 therefore makes some geometry operations more demanding. The **legacy**, **legacy-plus** and **q8** profiles use particular numerical formats and support specific combinations of features; they are selected at build time.

> **A CHOICE THAT AFFECTS THE WHOLE SCENE**
>
> Geometric precision does not automatically enable a particular graphics mode. Coordinate precision, lighting and texture sampling can be considered separately, then combined in a configuration supported by the engine.

Clipping and hiding surfaces



A nearby object occupies much of the view.



Walls and the character must overlap in the correct order.

A face may extend beyond the edge of the screen or pass too close to the camera. **Clipping** trims the polygon and keeps the portion that can be drawn. Newly created boundary points must also receive the correct lighting and texture data; otherwise the surface would change appearance as it is clipped.

Culling removes entire faces that should not be drawn according to the profile's rules. A typical example is the back of a single-sided surface. Trimming a face and removing it are different operations, with different consequences as the camera approaches.

The painter's algorithm

The **painter's algorithm** draws from far to near: a closer face covers the pixels of a more distant face, like a brushstroke painted over another. Depth **buckets** group faces to help organize this order. In supported shared rendering paths, faces from different objects enter the same ordering process.

The painter's algorithm does not store the depth of every pixel as a **Z-buffer** would. When two faces intersect, neither is entirely in front of the other, so a single drawing order may be insufficient. Meshes must be designed with this in mind, for example by splitting surfaces where they intersect.

The bitmap: pixels inside bytes

The **bitmap** is the memory area that describes the image. The rasterizer updates it by breaking polygons into horizontal segments called **spans**. Once it has located the edges on a row, it fills the interval between them.

In the C64 multicolor bitmap, each pixel uses a two-bit code, so one byte contains four codes. A code selects a color source: the actual color depends on the screen cell containing that pixel.

01 10 11 01 = four pixels in one byte

Code	Where the color comes from
00	The global background color in register $D021.
01	The upper four bits of the corresponding Screen RAM byte.
10	The lower four bits of the same Screen RAM byte.
11	The value in the cell’s Color RAM location.

The bitmap specifies which of the four codes to use for each pixel. **Screen RAM** and **Color RAM** determine which colors correspond to the cell’s three local codes. Each color index occupies four bits, a **nibble**, enough to represent all 16 entries in the C64 palette.

> **LOGICAL PIXELS AND PHYSICAL PIXELS**
>
> In the normal polygon viewport, the renderer works at 160 × 100 logical pixels. Each covers a 2 × 2 block of physical pixels, producing a 320 × 200 image. Rows are doubled vertically, reducing drawing work while preserving recognizable shapes and proportions.

Writing whole bytes is efficient. At the edges of a span, masks are needed: bit operations that change only the intended pixels and preserve their neighbors.

Colors, materials and dithering

The C64 has 16 colors in total, but a single multicolor bitmap cell can use four: the shared background and three local choices. A cell covers an 8 × 8 block of physical pixels. Objects that overlap within it therefore share part of their palette.

A **material** provides a set of pigments and a response to light. Names such as satin and gloss describe behaviors within the engine's discrete lighting model. A material called mirror does not imply that it reflects the surrounding environment.

PACKING THREE PIGMENTS INTO COLOR MEMORY

```
screenByte = (Dark << 4) | High
colorRam   = Highlight

; Example: Dark=9, High=8, Highlight=10
; screenByte=$98, colorRam=$0A
```

The **<< 4** shift puts the Dark pigment in the upper four bits; the **|** operation combines those bits with High. The third pigment goes into Color RAM. This fragment illustrates packing: storing several values within the same byte.

More apparent shades from the same colors

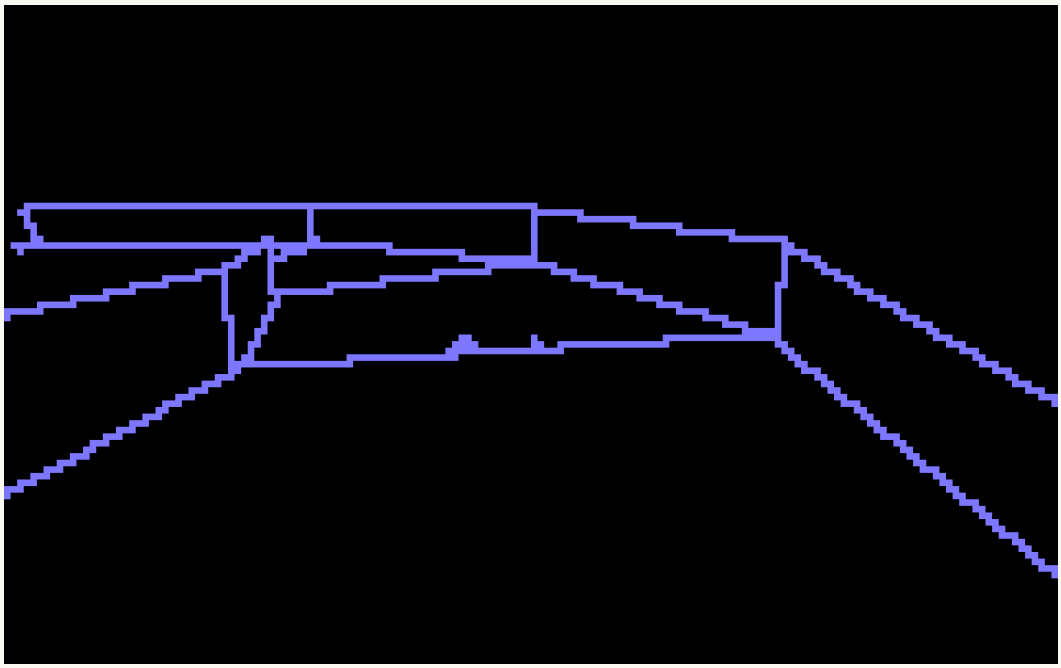
Dithering alternates pigments in small patterns. Viewed together, the dots create the impression of an intermediate shade. A pattern with few bright dots looks darker; increasing their density makes the surface appear brighter. The hardware palette stays the same.

> **WHEN TWO SURFACES SHARE A CELL**
>
> If two polygons require different colors within the same 8 × 8 block, they must still fit within the three available local colors. The renderer handles this conflict according to the profile's rules. Materials with compatible palettes can reduce its visible effects.

Rendering modes: lines to maps

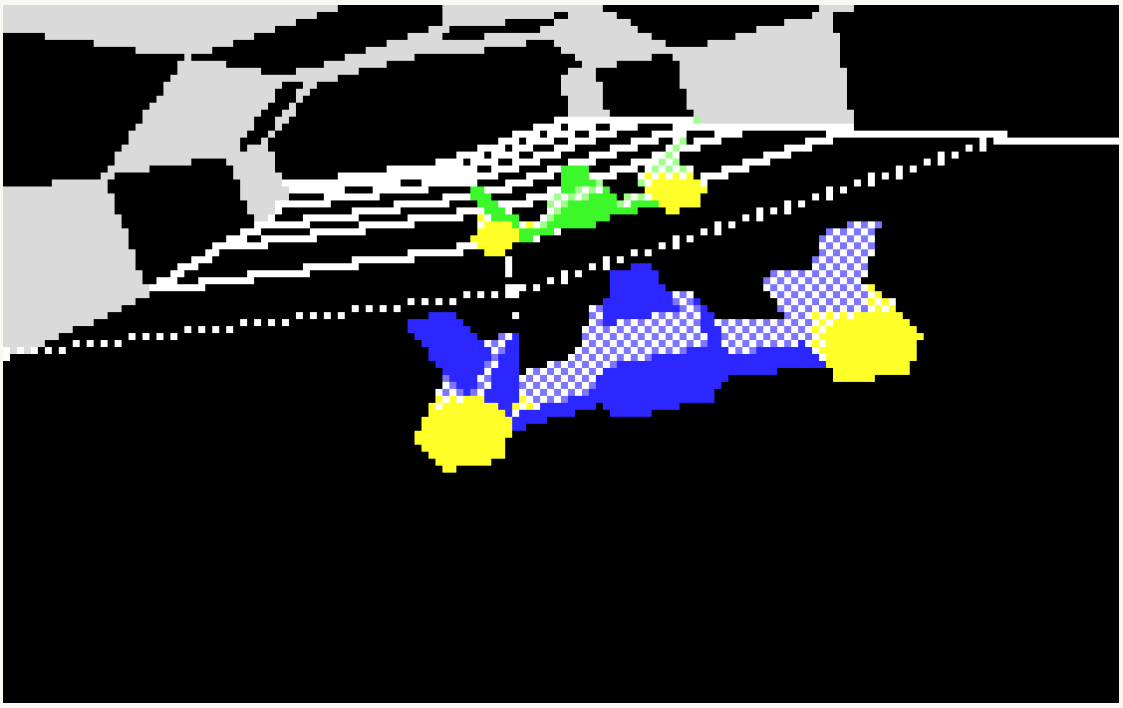
A **rendering mode** determines how geometry becomes an image. The first seven operate on polygon meshes; mode 8 uses data and algorithms designed for 2.5D maps. Each number identifies a family of features whose compatibility must be checked against the chosen profile.

Mode	What it draws
1	Wireframe: draws the edges of the geometry.
2	Hidden-wire: masks rear edges with the surfaces covering them.
3	Filled surfaces with static shading.
4	Dynamic flat shading: calculates one lighting value per face.
5	Dynamic flat shading with surface outlines.
6	Gouraud: interpolates lighting between vertices and displays it with dithering.
7	Texture mapping: applies small patterns to surfaces.
8	2.5D maps: builds the view from cells and vertical profiles.



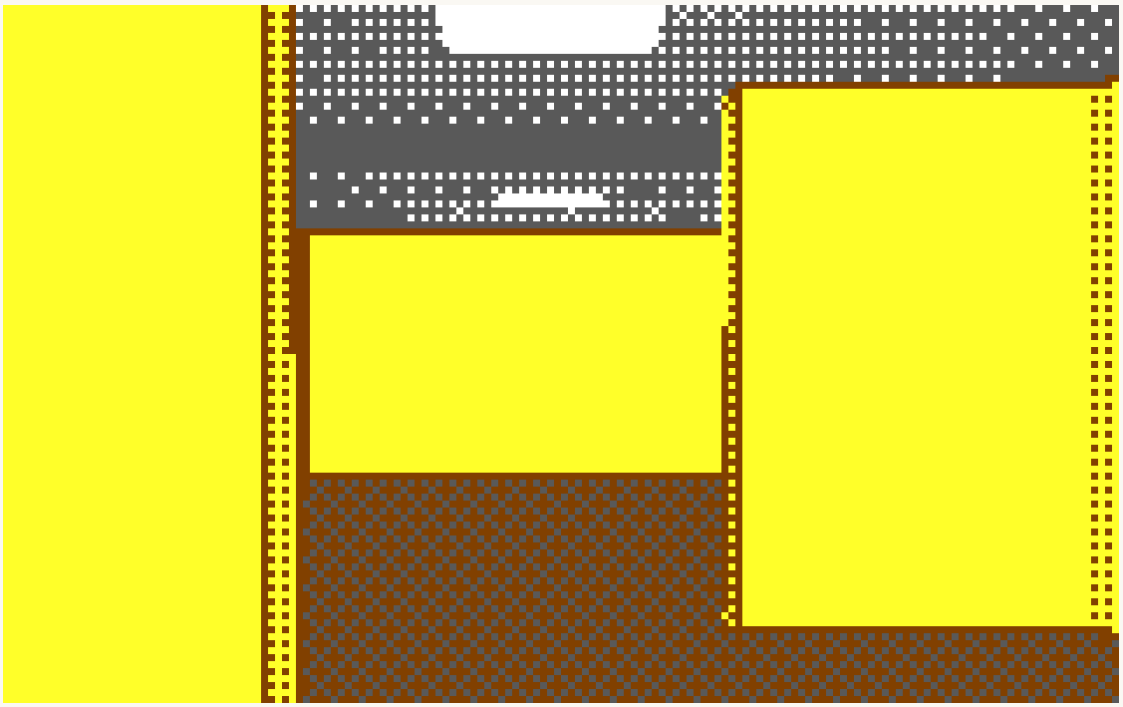
Hidden-wire mode displays visible edges and hides those covered by surfaces in front of them.

Lighting makes shapes readable



Flat shading. Each face has uniform brightness. Differences between faces reveal the orientation of their surfaces.

The lighting calculation uses the face **normal**, a direction perpendicular to the surface, to assess its orientation relative to the light source. Flat shading produces a single value for the whole face. Sharp changes between light and dark areas emphasize the model's edges.



Gouraud shading on the ceiling. Lighting is calculated at vertices and interpolated across the surface. A 4 × 4 Bayer pattern represents 33 discrete levels. The walls in this scene use flat shading.

Applying textures to surfaces



The wall and floor patterns become smaller with distance, making the corridor's perspective easy to see.

A **texture** is a small image assigned to a surface. In this rendering path, texture pages contain 16 × 16 samples: 256 bytes at runtime, plus alignment and metadata. Each sample, called a **texel**, selects one of the material's pigments.

U and V coordinates specify a location to read in the texture, like a column and row in an image. Assigning them to each face corner determines how the pattern covers the polygon. Adjacent faces can use different coordinates along the same shared edge, creating a texture seam.

Affine and perspective-correct sampling

Affine interpolation varies U and V linearly across the screen. On a wall that recedes from the camera, this approximation can distort the pattern. **Perspective-correct** sampling also accounts for depth, so the texture compresses with distance according to the same projection as the surface.

THE IDEA BEHIND PERSPECTIVE-CORRECT SAMPLING

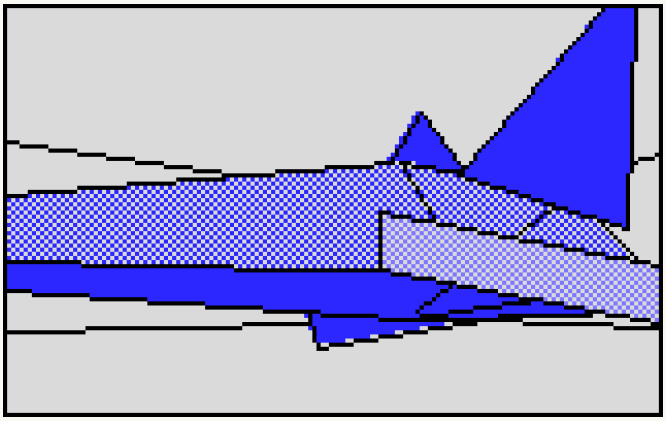
```
W ~ 1 / depth
UW ~ U * W        VW ~ V * W
U = UW / W        V = VW / W
```

W represents inverse depth. Interpolating U × W, V × W and W, then dividing, reconstructs the texture location to read. The actual values remain quantized to the renderer's numerical formats.

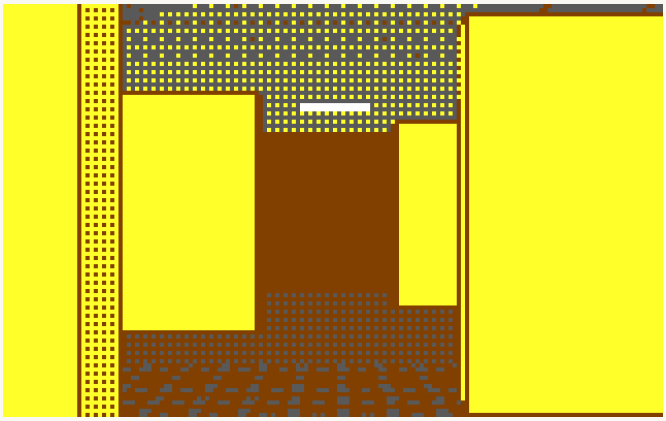
Image quality and speed

To draw a texture, the renderer calculates which samples to use at the points covered by a surface. Repeating this for every point can be expensive. Some options reduce calculation by approximating the result; others simplify distant details that occupy few pixels and are difficult to distinguish.

Choice	Practical effect
Nearest-neighbor	Selects the texel closest to the calculated coordinate. Edges remain sharp and blocky.
Texture fast	Approximates some sampling and reconstructs omitted rows, reducing work while changing the result.
Gradual LOD	Progressively switches to a simpler pattern in the distance. LOD means level of detail.
Exact optimization	Reuses results and specializes routines in supported paths without the visual approximations described above.



At close range, the hull covers more of the image.



Sampling the floor and ceiling textures requires memory reads and calculations.

A **cache** avoids recalculating results that are already available; a **specialized kernel** runs a sequence of operations tailored to a specific case. These exact optimizations apply only to the profiles that support them. Q8 precision, perspective-correct textures and LOD are selected separately, in combinations accepted by the builder.

How much memory a scene uses

Code, geometry, images and temporary data must all fit in C64 memory. Different additions have different costs: a more complex mesh needs extra vertices and faces, a texture needs a page of samples, and double buffering needs a second image area.

Resource	Cost or characteristic
Full bitmap	8,000 bytes per image. Two buffers require two areas.
Screen RAM	1,000 bytes for the cells of a complete image.
Color RAM	1,000 nibbles used by the cells; this is a single shared area.
Texture page	256 bytes at runtime for 16 × 16 samples, plus alignment and associated data.
Geometry and workspace	Vertices, faces, tables and temporary storage depend on the profile and scene.

Intermediate results need memory too

Scratch space is temporary storage used by routines. It may hold transformed coordinates, points created by clipping or intermediate lighting results. These values are often needed only while a frame is being drawn, but their memory areas must still be reserved and coordinated with the rest of the program.

The builder checks addresses and may reject a configuration before it reaches a nominal vertex or object limit. The memory map of the actual build determines how much space remains. The size of the PRG file alone does not describe every area reserved during execution.

> **WHAT SHARING A MESH SAVES**
>
> Storing one shape for several objects reduces the amount of source geometry. Each object still needs its own position and transformation results. Assessing the saving therefore requires considering both the shared mesh and the data belonging to individual instances.

Multiplying with a lookup table

The 6510 has no multiplication instruction. One solution used by the engine obtains a product from precomputed values: it reads two table entries and subtracts the second from the first. Each entry Q(n) contains the square of n divided by four, rounded down.

a × b = Q(a + b) - Q(|a - b|)

Q(n) = floor(n² / 4)

To calculate 37 × 19, use the sum 56 and the difference 18. Reading Q(56) and Q(18) gives **784 - 81 = 703**. The table takes memory, but avoids running a full multiplication sequence each time.

SOURCE EXCERPT: THE FINAL 16-BIT SUBTRACTION

```
er_diff_ready:
  tax
  sec
  lda prodlo
  sbc sqlo,x
  sta prodlo
  lda prodhi
  sbc sqhi,x
  sta prodhi
  rts
```

TAX copies the table index into the X register. **LDA** loads a byte into the accumulator, **SBC** subtracts the table byte from it, and **STA** stores the result. The routine processes the low byte first, then the high byte: together, they form the 16-bit value.

SEC sets the carry flag, a CPU status bit also used in subtraction; any borrow passes from the low byte to the high byte. **RTS** returns to the caller. The complete routine also handles **overflow**, when the sum exceeds the capacity of the value used to hold it, and observes the permitted numerical ranges.

Presenting a complete image

The renderer builds an image piece by piece. If the VIC-II reads the same area while the CPU modifies it, the screen may show parts of the old frame mixed with parts of the new one. **Double buffering** uses two memory areas, called buffers: one holds the visible image while the other holds the image being prepared.

01	**Buffer A**	The video hardware displays a complete image while the program works on buffer B.
02	**Buffer B**	The renderer finishes the new view. The switch is coordinated with video timing.
03	**Swap**	B becomes visible; A becomes available for drawing the next view.

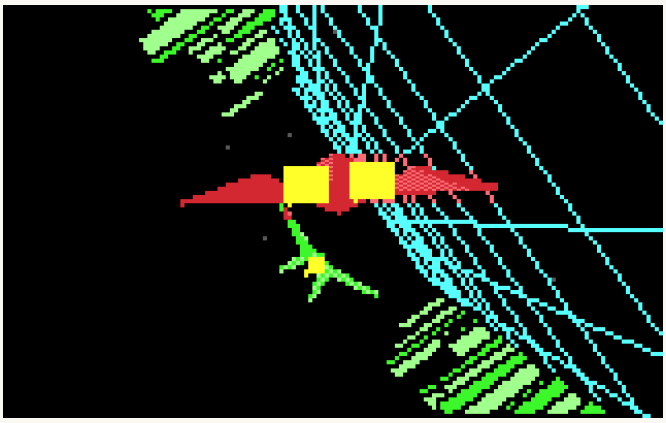
SOURCE EXCERPT: COMPLETING AND PRESENTING A FRAME

```
render_frame_end:
  jsr wait_raster
  jsr show_buffer
  jsr fps_frame_done
  jmp main_loop
```

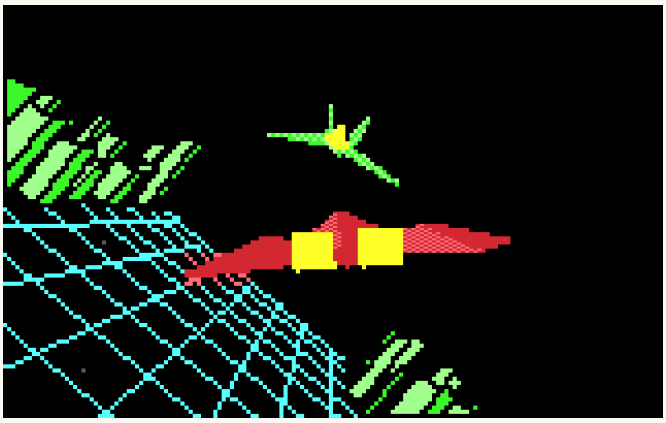
JSR calls a subroutine. **wait_raster** waits for the intended point in the video scan; **show_buffer** presents the complete image; **fps_frame_done** records a newly completed view. **JMP** returns control to the main loop. Conditional branches from the source are omitted here.

The **raster** is the scan through the video lines. Synchronizing the switch reduces display artifacts, but does not make drawing instantaneous. Color RAM is also a single area: switching the bank, the region of memory read by the VIC-II, does not create a second copy of the cell colors.

Objects, cameras and scene timing



Camera roll tilts the view and its horizon.



More distant spacecraft appear smaller.

A **pose** describes position and orientation. Changing an object's pose moves or rotates it within the scene. Changing the camera's pose changes the viewpoint on every object. **Roll** is camera rotation around its viewing direction: it tilts the horizon in the image.

Scene time and rendering time

The standard timeline uses **50 logical ticks per second**. A tick is one step of simulation time: 100 ticks represent two seconds. The timeline can organize states, durations, poses and visibility. Some demos, particularly those using Q8 profiles, have dedicated control routines.

If a spacecraft always moved one unit per rendered frame, it would slow down whenever rendering took longer. By measuring elapsed time, the program can calculate displacement from the actual duration of the interval. Average movement speed then remains independent of FPS, although fewer frames still make motion look less smooth.

Music and interrupts

The **SID** is the C64's sound chip. A music routine updates it periodically and consumes some CPU time. An **IRQ** is an interrupt that lets a routine run at the required moment. Music and rendering code must use registers and temporary memory without overwriting data that the other still needs.

From graphics to an interactive world

Drawing an environment is only part of making it explorable. It also needs movement controls, collision rules, objects with state, and a way to manage level data. Alongside its renderer, 3Dvibe64 provides modules for these tasks, each with documented features and limits.

Module family	What it helps organize
Entities	Objects with state and behavior, such as a door that can open or close.
Materials	Per-surface settings, texture animation, UV scrolling and lighting prepared on the PC.
Movement	Controls, trajectories and object updates in supported paths.
Levels and worlds	Groups of data, connections between areas and memory budgeting.
Characters	Meshes or billboards: flat images placed in space, with masking rules.

Integrating a module

A module’s documentation specifies the data it accepts, the resources it uses and the renderers it works with. These rules form its **contract**. Some experimental modules require separate build tools. Check their compatibility with the chosen profile before using them.

The demos also contain code written for particular scenes. In the labyrinth, for example, the renderer draws the polygonal character, while its animation and the inventory interface may be handled by the demo program. Recreating that experience requires those parts as well.

What 2.5D means

In the 2.5D path, an environment is described by a floor plan divided into cells, with floor and ceiling heights assigned to them. The program derives a perspective view from this data. The structure allows fewer shapes than a freely modeled polygon scene, but makes it possible to calculate and draw the environment with specialized algorithms.

From a floor plan to screen columns

Raycasting traces virtual rays from the observer's position through the floor plan. The **DDA** algorithm follows each ray from one cell to the next to identify the surfaces it encounters. Their distances and heights determine how much vertical space they occupy on screen. Extra samples may be needed near edges and openings.

FROM A CELL TO COORDINATES: A CONCEPTUAL EXAMPLE

```
index = 32 * y + x

; Cell center in Q8.8 coordinates
xQ8 = 256 * x + 128
yQ8 = 256 * y + 128
```

A Mode 8 map uses a **32 × 32 cell** grid. Multiplying y by 32 locates the start of a row; adding x selects the cell. In Q8.8 coordinates, adding 128 moves half a cell, placing the point at its center.

Which environments it can describe

The path supports one floor/ceiling pair per XY position, along with ramps and openings defined by the available templates, and a solid map boundary. A room directly above another room would require multiple volumes at the same floor-plan position and falls outside this model.

Mode 8 is separate from the mesh renderer: it does not automatically turn a polygon model into a map or freely combine the two rendering paths. Its bitmap viewport contains 128 × 144 logical pixels. The project supports execution on a stock C64.

The 2.5D family on other machines

The 2.5D pipeline underpins four projects for other Commodore computers. Each adaptation, or **port**, uses its machine's own video hardware and memory. The repositories contain source code and documentation for the respective systems.

2.5vibe128 - Commodore 128

Uses the VIC-IIe in native C128 mode. It enables FAST, the higher-speed CPU mode, during the video borders. The multicolor bitmap view contains 128 × 144 logical pixels. It offers automatic and interactive exploration; the continuous-world path uses a specialized build.

github.com/Librologica/2.5vibe128

2.5vibePlus4 - Commodore Plus/4

Adapts the pipeline to the Plus/4's TED chip. It provides a 128 × 144 logical-pixel multicolor bitmap, double buffering, and automatic or interactive demos. Memory and video organization follow the Plus/4's hardware requirements.

github.com/Librologica/2.5vibePlus4

2.5vibeVic - VIC-20 with 24 KB expansion

Requires a 24 KB RAM expansion. It offers a choice between monochrome output at 128 × 96 logical pixels and multicolor output at 64 × 96. Using color therefore trades away some horizontal resolution.

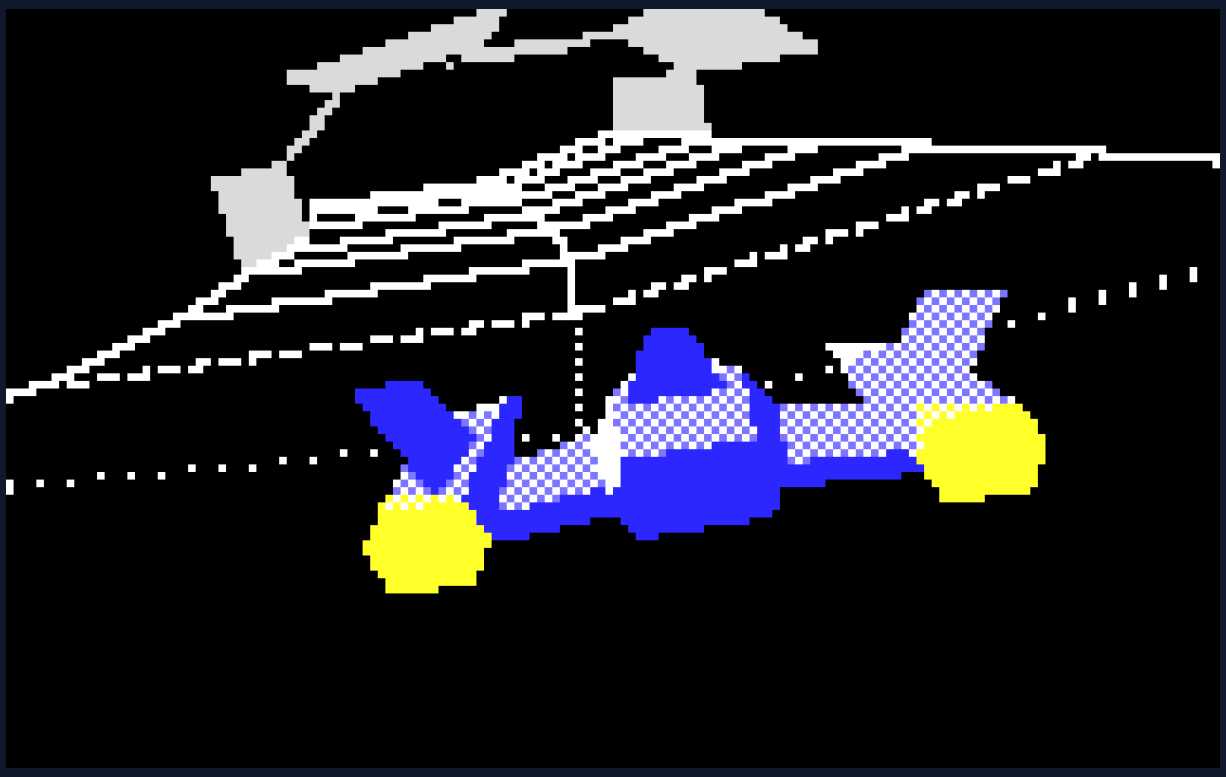
github.com/Librologica/2.5vibeVic

2.5vibePet - PET 4032 with 32 KB

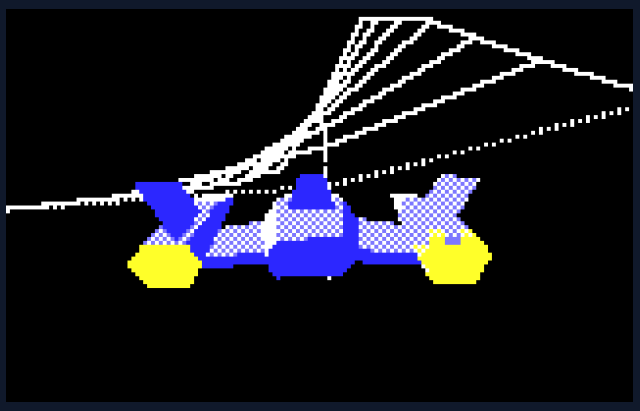
A monochrome port with procedural worlds: environments generated by program rules. It includes automatic and interactive exploration, 6502 source code and pipeline documentation, with output adapted to the PET.

github.com/Librologica/2.5vibePet

Flying through bends and tunnels



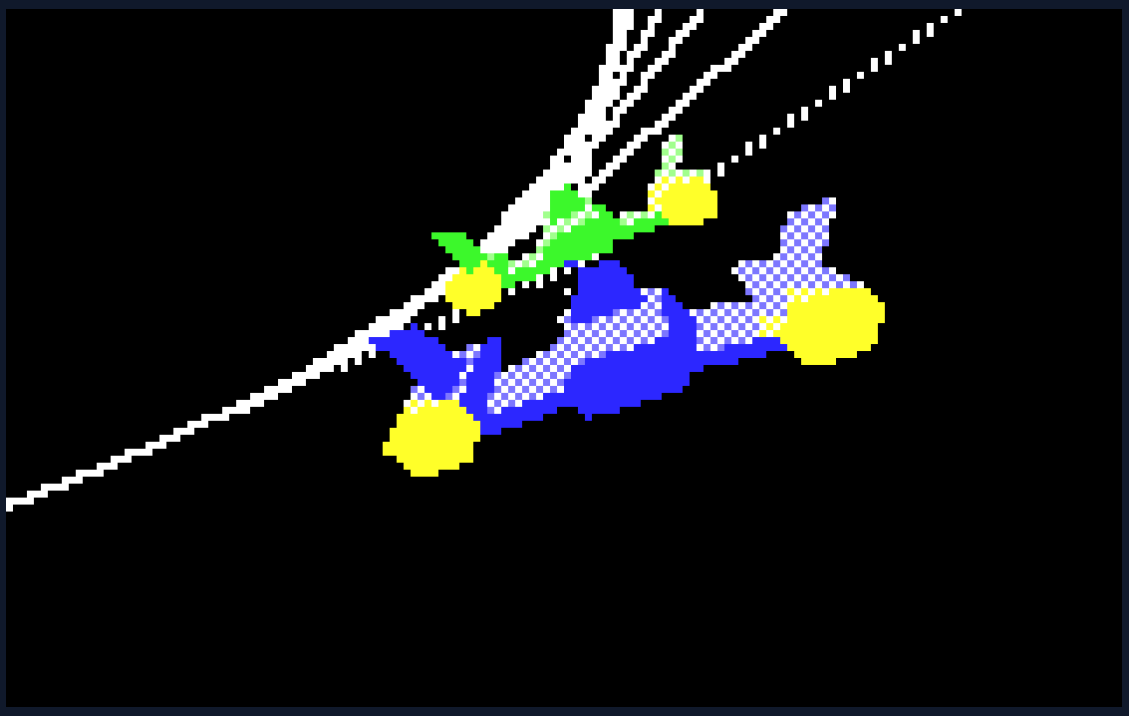
The camera follows the spacecraft as the track changes direction and passes through a tunnel.



What to look for. Perspective makes the track edges converge in the distance. The camera follows the bends and changes its tilt, while the spacecraft stays in the foreground.

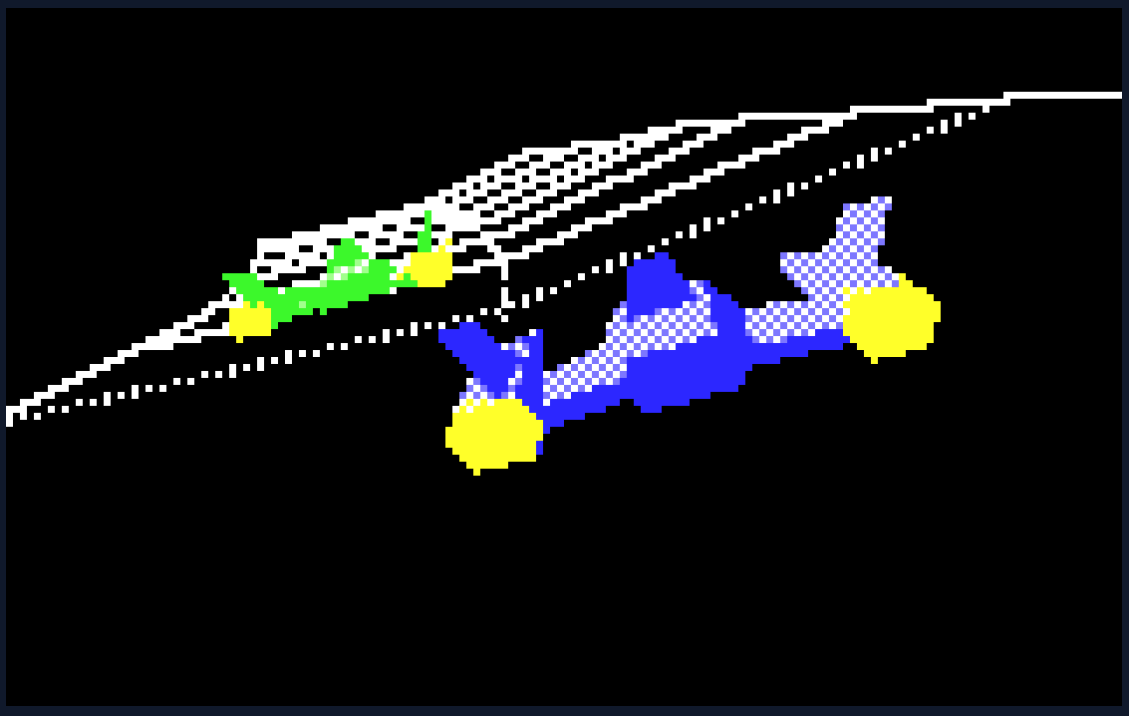
At a tunnel entrance, walls begin to occupy much of the view. In open sections, the track and spacecraft dominate the image. This alternation changes the number and screen coverage of the surfaces being drawn, and therefore the work needed to complete a frame.

Duels and overtaking



The two spacecraft occupy different positions and appear at different sizes during a turn.

During an overtaking maneuver, the depth order of the spacecraft changes. The renderer must update which surfaces cover others. Their colors make the two craft easy to follow; differences in brightness between faces help reveal the orientation of wings and hulls.

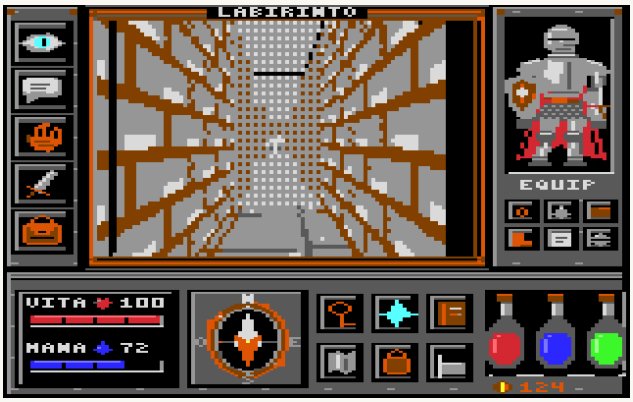


The two spacecraft follow the same track at different distances from the camera.

Exploring a labyrinth



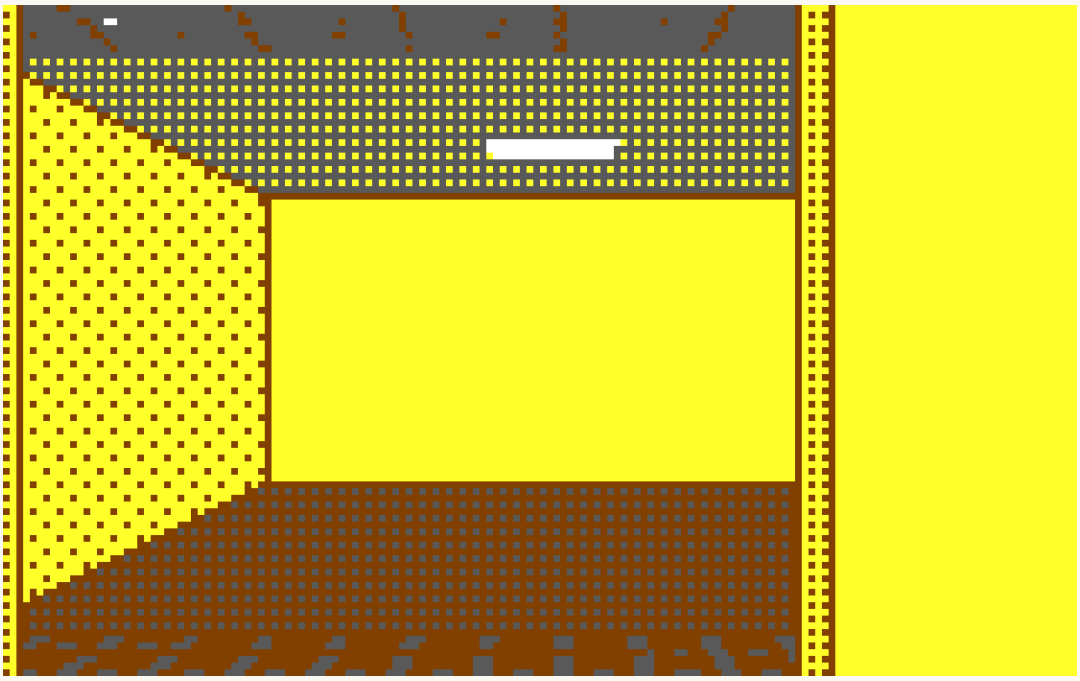
The character is made from polygonal body segments and is drawn together with the labyrinth walls.



What to look for. The interface reserves space for inventory and player status. The 3D viewport contains 96 × 60 logical pixels: a smaller window reduces the area that must be drawn.

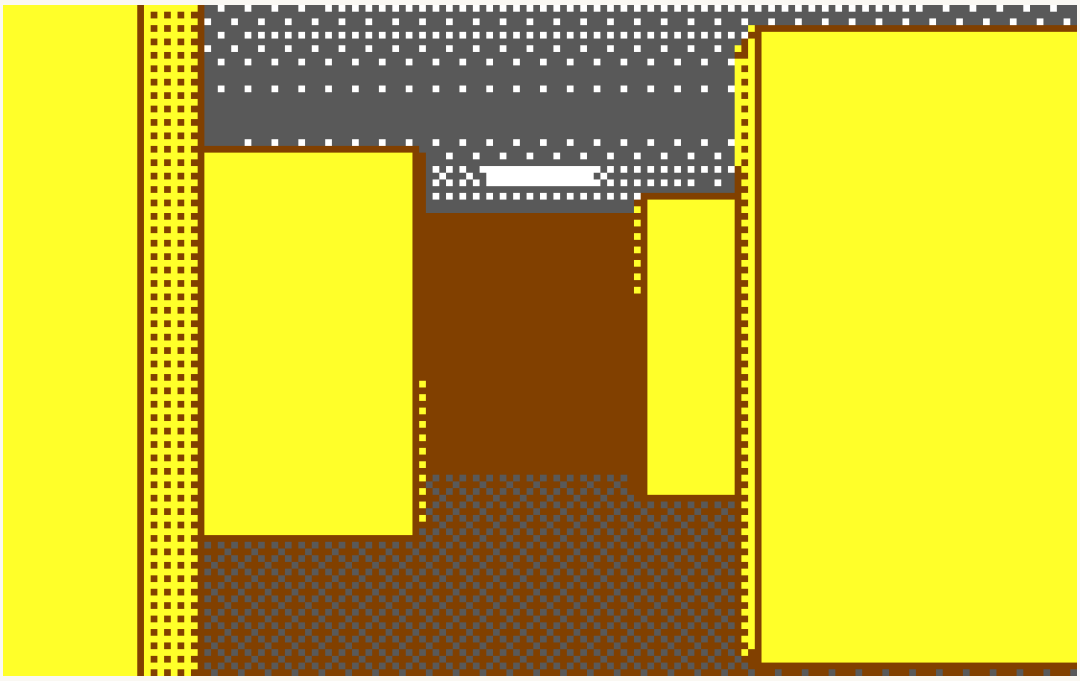
Repeating patterns distinguish the walls from the floor and ceiling. The character uses simple components to describe the torso and limbs. Keeping inventory slots and indicators outside the 3D view leaves the renderer with a smaller area to update.

Continuous rooms and ceiling light



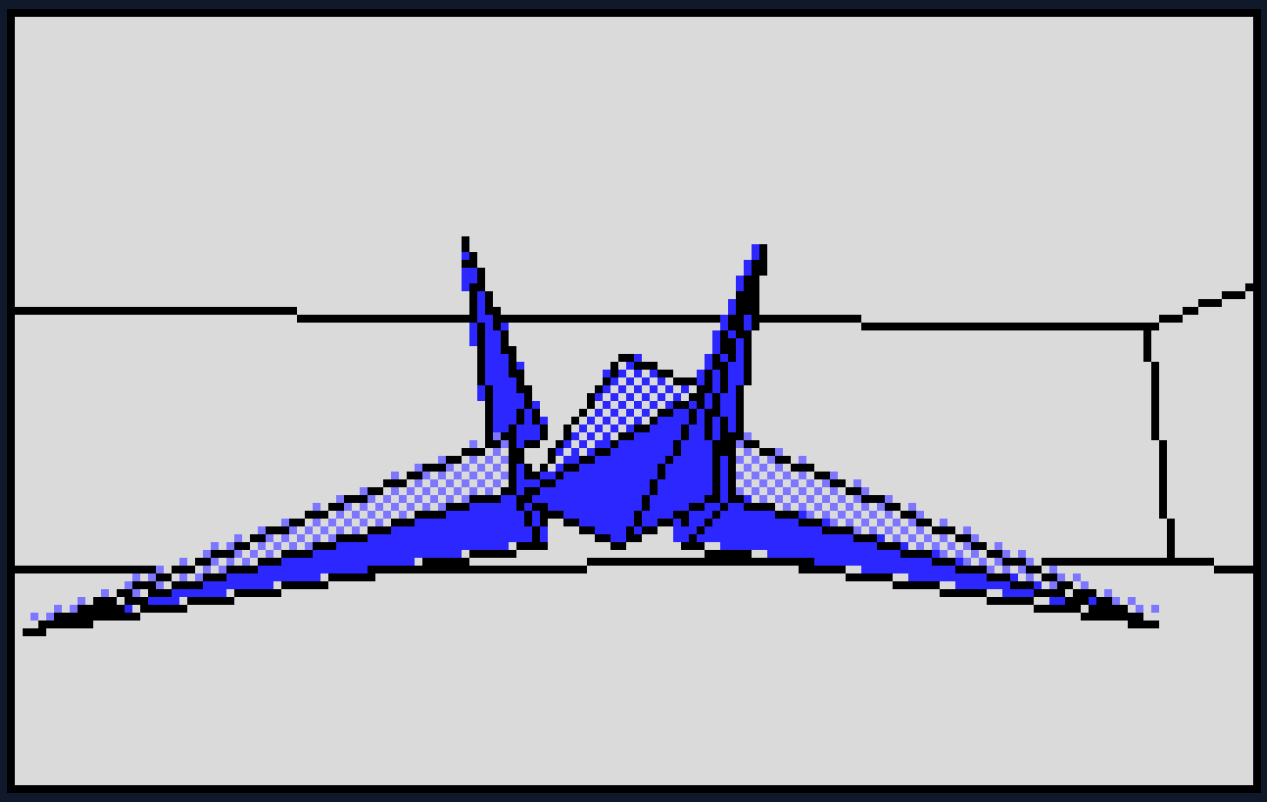
The demo reuses rooms held in memory along the route, creating the impression of a larger environment.

The program keeps part of the environment in memory and reuses its elements during exploration. The route can continue without storing every room at once. The second image shows a variant with Gouraud lighting on the ceiling, represented through dithering.

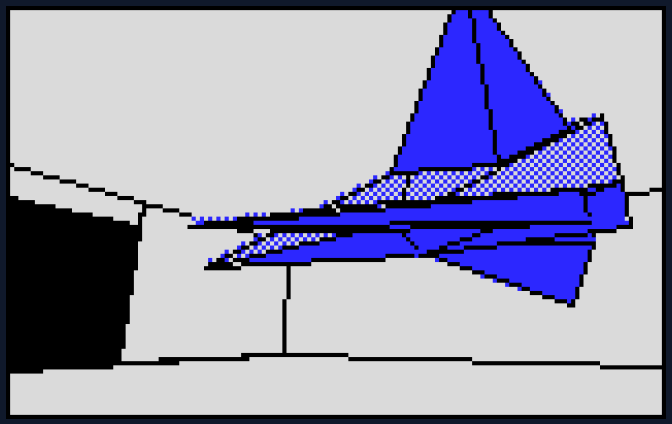


In this variant, Gouraud shading varies brightness across the ceiling. Changes in the density of bright dots reveal the gradations.

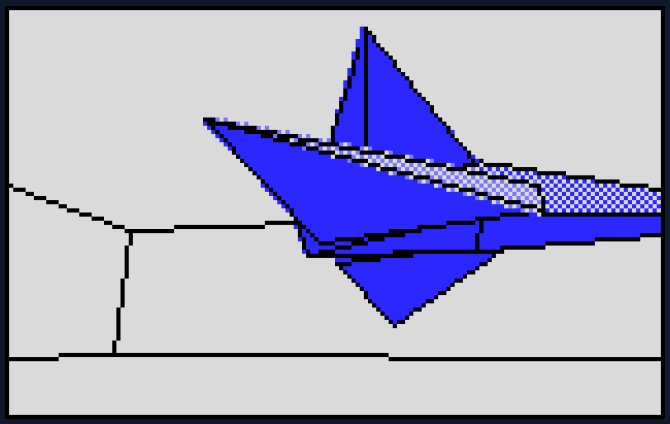
The spacecraft inside the hangar



The entire spacecraft is visible against the hangar walls and floor.



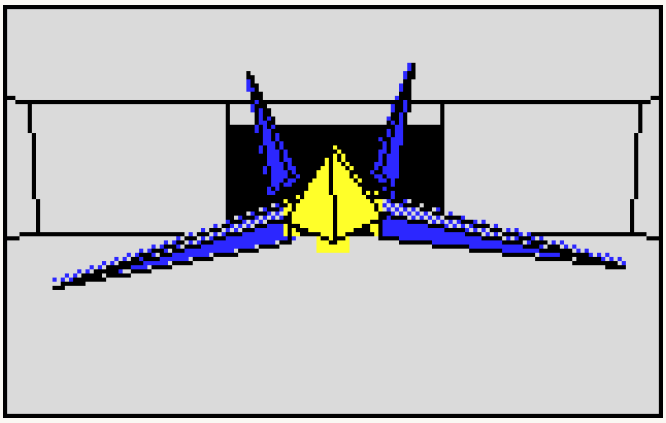
Inspecting the side profile.



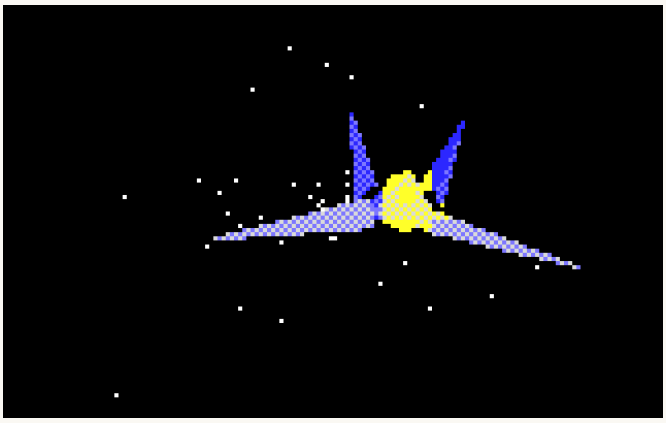
Viewing the lower surfaces.

The camera moves around the spacecraft, first showing its complete silhouette and then details of its wings and hull. Lines on the walls help reveal changes in viewing direction. Blue surfaces and dark outlines keep the model distinct against the hangar's light background.

From inspection to flight



The departure phase inside the hangar.



The spacecraft outside, against the starfield.

The demo has several phases: inspecting the spacecraft, departing and flying outside. The program coordinates the movements of the camera and craft, as well as the transition between the two environments. After leaving the hangar, it uses dynamic flat shading for the spacecraft and continues with dedicated flight code.

FIELDS EXTRACTED FROM THE HANGAR SCENE

```
"graphicsMode": 5,
"camera": {
  "id": "observer",
  "mode": "walkLite"
}
```

The **graphicsMode** field selects the rendering path; **id** assigns an identifier to the camera; **mode** selects a supported camera mode. These fields belong to a larger description containing geometry, materials and configuration.

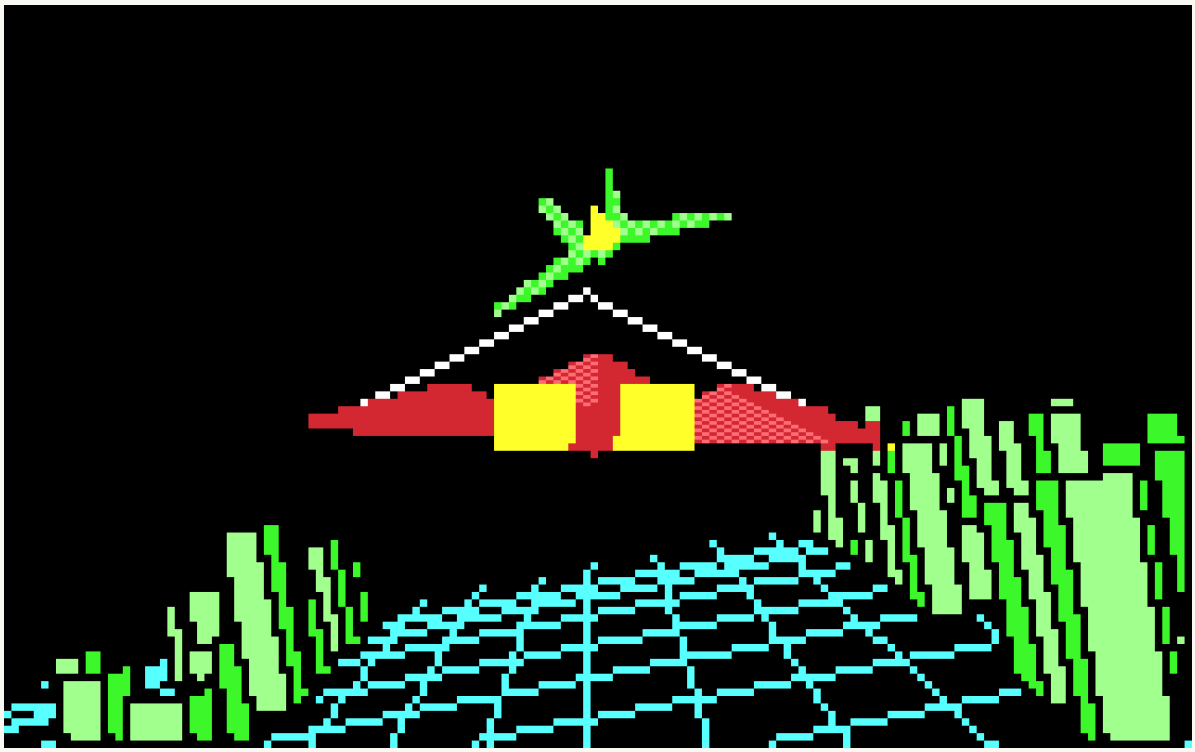
The scene file describes the initial elements, but the sequence also requires code to update positions, orientations and phase transitions. The demo combines that control with Q8 geometry, SID music and two surface treatments: the one used inside the hangar and dynamic lighting during flight.

Designing a scene with 3Dvibe64

Define the scene. Before adding detail, decide which objects should appear, how they move and where they are viewed from. A flight sequence needs trajectories and camera framing; an explorable labyrinth needs routes, openings and movement rules.

Choose the required features. The environment guides the choice between polygon meshes and a 2.5D map. Materials, lighting, textures and numerical precision must be selected in supported combinations. The builder checks the data and verifies that it fits in memory.

Run the program on the intended machine. A stock C64 calls for more restrained scenes; a C64 Ultimate at 64 MHz can handle more demanding calculations. Testing should cover the whole route: a close-up view can cost much more to draw than a distant view of the same object.



A chase scene combines object movement with camera movement. Both change the view that must be drawn.

The project and its source code

The 3Dvibe64 repository contains the engine’s source code, build tools and documentation. The guides describe scene formats, available options and the combinations supported by the different renderers.

3Dvibe64

A polygon engine for the Commodore 64, with a separate Mode 8 path for 2.5D maps. Prepare your project’s scenes and assets according to the documentation, then provide them to the build tools.

github.com/Librologica/3Dvibe64

Putting the ideas into practice

A useful starting point is a simple scene with a few objects and an easily controlled camera. Change one element at a time: rotate a mesh, replace a material or move the viewpoint closer. Comparing the results helps explain what each option does and how much work it adds.

For a closer look, follow the same steps in the assembly source: transforming vertices, calculating lighting, filling the bitmap and presenting the frame. Comments and documentation help connect the routines to the data they process.

> **THE IMAGES IN THIS GUIDE**
>
> The screenshots are native captures of demos running in VICE. They were cropped to remove the video border and enlarged with sharp pixel edges. They show the programs’ actual output, including their original interface labels. Diagrams and formulas in the text explain the underlying principles.

The 2.5D variants for the Commodore 128, Plus/4, VIC-20 and PET have their own links in “Maps and variants”. Each project includes documentation for its target machine.