CRAZY: A Commodore 64 Raycaster With Texture Mapping

By Wyndex (a.k.a. Steve Judd)

Jul 8, 2024 (last rev Dec 25, 2024)

1. Background

On Robin Harbron's YouTube channel, 8bitShowAndTell, there is a video about the SuperCPU that made the comment that no programs were ever really developed for it (with the partial exception of Metal Dust), i.e. the SCPU wound up mostly just running C64 programs at 20 MHz, and we never got to see what it was really capable of. That seemed a shame, especially since I had written a bunch of SCPU development tools that also weren't much used, and thus the utterly crazy idea was born of using those SCPU development tools to write something that really showed what the SuperCPU could do. Something more than a C64 program running at 20 MHz, Something like a texture mapped ray caster to let you run around a 3D world on a C64.

The first problem was that I had never done a ray caster before, so I used lodev's ray-casting tutorial at https://lodev.org/cgtutor/raycasting.html to figure things out. The second problem was that my C128D no longer worked, but a new power supply and cooling fan for good measure fixed that. The third problem was that I hadn't coded on the C64 for 20 years, but eventually enough of it came back and I was able to figure out my old setup well enough to get up and running. The fourth and biggest problem was that I don't actually have any free time, but getting sick over Christmas break 2023 helped with that, and thus after eking enough time here and there on evenings and weekends the result was Crazy.

The code is set up as a library that anyone can use. It runs in multicolor bitmap mode at 160x160 resolution, leaving rows 160-199 available for displays like health, radar, whatever. Textures are full screen height at 80x160, which gives the proper aspect ratio since since multicolor pixels are 2X wide. Each texture uses a full bank of SuperRAM when processed, and crazy uses 58 of them. Crazy also places tables in all 16M of SuperRAM, runs code out of banks 0 and 2, and relies on multiple 65816-specific features. In the end it pushes the SCPU pretty hard, and is much more than a C64 program running at 20 MHz.

All coding was done on a C128D+SCPU using Slang/Sirius/Jammon (imy old tools). I also upgraded to a uIEC drive from Jim Brain, which was invaluable, and used a RetroTINK adapter with an older HDMI TV, as shown in Figure 1. The graphics were all found online, using imagemagick and some custom matlab/octave scripts to convert them into usable C64 graphics. Octave scripts were also used to generate the maps. These scripts are all described in the appendix. A special thank-you to my family for putting up with this peculiar hobby over the last few months.

Figure 1: There's no school like the old school.

Sections two and three go into the details of ray casting and texture mapping, while section four describes the SCPU implementation of these algorithms. Those wanting to skip the technical details can go straight to section five, which describes Crazy and how to make your own programs using the texture mapping library (and how surprisingly easy it is). Finally, the appendix briefly describes the various programs and scripts used for generating textures and maps, and provides a list of useful links.

Section 1: Background
Section 2: Ray casting
Section 3: Rendering and texture mapping
Section 4: SuperCPU implementation
Section 5: Crazy and the rendering library
Appendix: Scripts and useful links

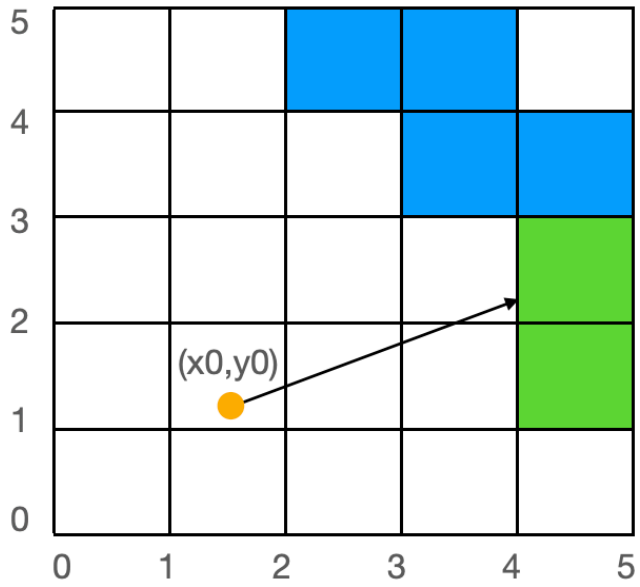

Figure 2: Ray casting concept.

2. Ray Casting

The idea of ray casting is captured in Figure 2. The world consists of an M x N (row x col) map, where each map square is either empty or contains an object, e.g. a wall. The player is located in the map at (x0,y0) looking in some direction (dx,dy). The edges for each map square occur at integer positions, while the player can be located at fractional locations between edges. The job of the ray caster is to cast the ray: to move in the ray direction until an occupied map square is hit.

A scene is built by casting a fan of rays about the look direction, one for each screen column. If a ray hits a wall the projected height of the wall is calculated and a vertical line is drawn of that height. If texture mapping is used the routine further calculates an index into the texture column, and then scales the texture column appropriately. Since Crazy uses a screen that is 160 pixels wide, in every frame it casts 160 rays, and for each ray computes the distance to the wall and the corresponding height, then draws textured vertical lines, one at a time, to build up the scene.

2.1 Iterating along edges

What makes the ray caster algorithm special is that instead of slowly moving along the ray and checking each point, it jumps rapidly from edge to edge, checking the corresponding map square at each iteration. In Figure 2, the routine would jump along the ray to the edge at x=2 and check the map square (mapx,mapy) = (2,1); then jump to x=3 and check (mapx,mapy) = (3,1); then jump to y=2 and check (mapx,mapy) = (3,2); and finally jump to x=4, check (mapx,mapy) = (4,2), and stop at the green square.

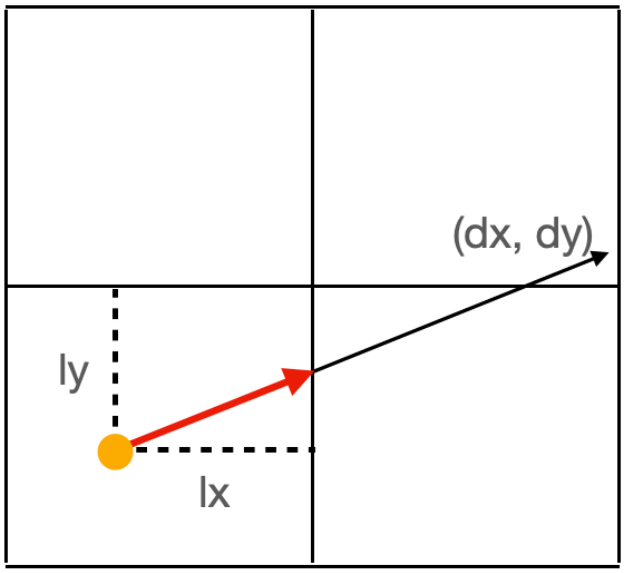

Figure 3: The algorithm iteratively moves along the ray to the closest edge.

Referencing Figure 3, consider a ray in the direction (dx,dy) i.e. with slope dy/dx. The distance from the player to the nearest x-edge is lx, and the distance from the player to the nearest y-edge is ly. The task is to compute the distance *along the ray* to these edges and move to the closest edge, and then continue this iteratively until either a map edge is hit or a maximum number of iterations is reached.

Consider first the ray-distance to the x-edge. If we move along the ray by a distance lx in the x-direction then the corresponding distance in the y-direction is lx * dy/dx, giving the total ray-distance as

$$\text{distX} = \sqrt{lx^2 + \left(lx\frac{dy}{dx}\right)^2} = \frac{lx}{dx}\sqrt{dx^2 + dy^2}. \qquad \text{(1a)}$$

Similarly, the distance along the ray to the y-edge is

$$\text{distY} = \sqrt{ly^2 + \left(ly\frac{dx}{dy}\right)^2} = \frac{ly}{dy}\sqrt{dx^2 + dy^2}. \qquad \text{(1b)}$$

By comparing distX to distY we can determine which edge the ray hits first and jump to that point. In Figure 3 the ray distance to the x-edge is closer, so we move along the ray to that edge and check that map coordinate for an object. At this new point, the x-distance to the next x-edge is exactly one unit and the ray distance is therefore

$$\text{New distX} = \sqrt{1^2 + \left(1\frac{dy}{dx}\right)^2} = \frac{1}{dx}\sqrt{dx^2 + dy^2}.$$

The new ray distance to the y-edge is the old ray distance minus the amount we just moved along the ray,

New distY = Old distY - Old distX.

If instead the y-edge had been closer, the calculation would have been:

$$\text{New distY} = \sqrt{1^2 + \left(1\frac{dx}{dy}\right)^2} = \frac{1}{dy}\sqrt{dx^2 + dy^2}$$

New distX = Old distX - Old distY.

At this point we have a first cut at a ray casting algorithm. If the player is located at (xpos,ypos) in Figure 3, looking in the direction (dx,dy), then the algorithm is

```
r0 = sqrt(dx^2 + dy^2)
distX = r0 * lx/dx
distY = r0 * ly/dy
while map(mapx,mapy)!= 0
     if distX < distY
          mapx++
          newdistX = r0/dx
          newdistY = distY - distX
     else
          mapy++
          newdistY = r0/dy
          newdistX = distX - distY
     end
     distX = newdistX
     distY = newdistY
end
```

We can simplify this considerably by instead using the difference between distX and distY,

```
deltaXY = distX - distY.
```

Because only the sign of deltaXY matters, not its value, the constant term r0 may be factored out of the calculations and the algorithm becomes the compact and fast

```
deltaXY = lx/dx - ly/dy
while map(mapx,mapy)!= 0
     if deltaXY < 0
          mapx++
          deltaXY += 1/dx
     else
          mapy++
          deltaXY -= 1/dy
     end
end
```

This is the basic ray casting routine — very clever, very simple, and very powerful. This simplicity enables a version to be implemented on the Commodore 64 + SuperCPU.

3 Rendering a scene: texture mapping

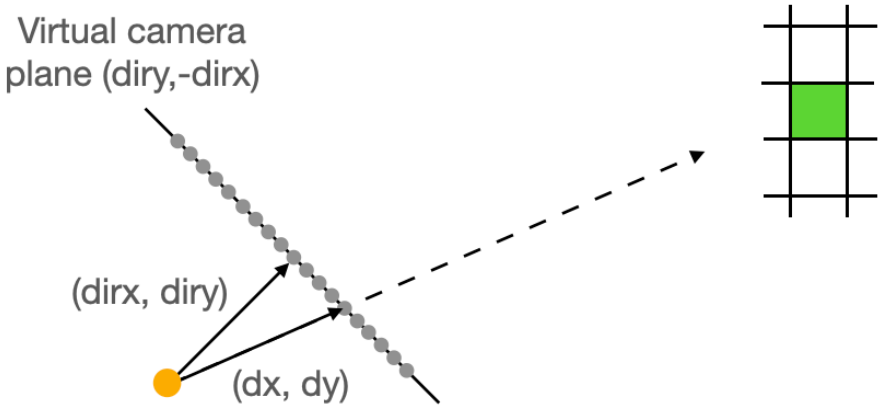

Figure 4: The virtual camera plane (VCP).

To render a scene we use a fan of rays about the look direction (dirx,diry) as shown in Figure 4. Each ray (dx, dy) is cast through a series of evenly spaced points along a line perpendicular to the look direction, where each point corresponds to a pixel column on the screen. The Commodore 64 in multicolor bitmap mode is 160 pixels wide and therefore 160 rays are cast in Crazy. At each point if the ray hits a map edge a vertical line is drawn at that point. The result is a virtual camera plane — an artificial plane that we construct from the look direction vector (dirx,diry). (Note that in a real camera such as your eyeball the image plane is behind the lens, not in front of it.)

The point spacing determines the field of view (FOV). For a fixed number of pixels, a narrow FOV will provide higher resolution over a small area while a wider FOV will provide lower resolution over a wider area. For each ray cast a vertical line is

drawn on the screen of the appropriate height and color / texture. Starting from the center view direction and working outwards, the pseudo-code looks like:

```
xr = 80
RayX = dirX
RayY = dirY
CameraDX = some fraction of diry (sets FOV)
CameraDY = some fraction of -dirx (sets FOV)
while xr < 160
     RayCast(RayX,RayY)
     if a hit then DrawVLine(xr)
     RayX += CameraDX
     RayY += CameraDY
     xr++
end
```

The above casts rays to the right of the look direction; similar code may be used to cast rays to the left. (CameraDX, CameraDY) is a vector perpendicular to the look direction (dirx,diry), as shown in Figure 4, and the magnitude of the vector sets the pixel spacing i.e. the FOV. For each ray, RayCast(RayX,RayY) determines if the ray hits a wall, and if so a line is drawn. To draw the vertical line — DrawVLine(xr) — three pieces of information are needed:

1. What it hit — nothing, or the value of the map square (which texture to use).
2. How far away it is — specifically the height, to render the vertical line.
3. Where it hit — an index into the texture map.

For untextured ray casting only the first two are needed. Crazy uses all three.

3.1 What it hit

The map is stored as an array, so the ray casting routine from section 2 automatically determines the map square as

```
map(mapy,mapx)
```

which determines whether we hit anything at all, and if so, what we hit.

It is also useful to know which of the four sides was hit. In principle a map square could have four different textures assigned to it, depending on whether the x- or y-side was hit in the positive or negative direction. In lodev's writeup this is used to implement things like artificial lighting, however in Crazy only the x- or y-sides are assigned a texture. The side that was hit is determined by whether the last step taken by the ray caster was in the x- or y- direction, and whether it was hit in the plus or minus direction.

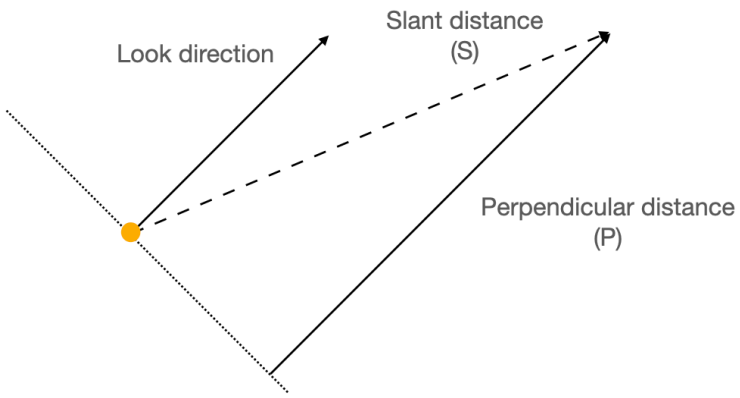

Figure 5: Perpendicular vs. slant distance

3.2 How far away it is (vertical line height)

To determine the "distance" to a map square, consider the slant distance S and the perpendicular distance P as shown in Figure 5. In physical optics there is a concept of the near field and far field. In the near field waves are spherical, corresponding to the slant distance, while in the far field they are planar, corresponding to the perpendicular distance P. When dealing with light we are pretty much always in the far field and so the perpendicular distance P is desired. (Iodev's writeup has examples of what the world looks like using the slant / near field distance S instead.)

The perpendicular distance P is deduced using the two pairs of similar triangles shown in Figure 6. From the left diagram,

$$\frac{P}{S} = \frac{c}{r_0}$$

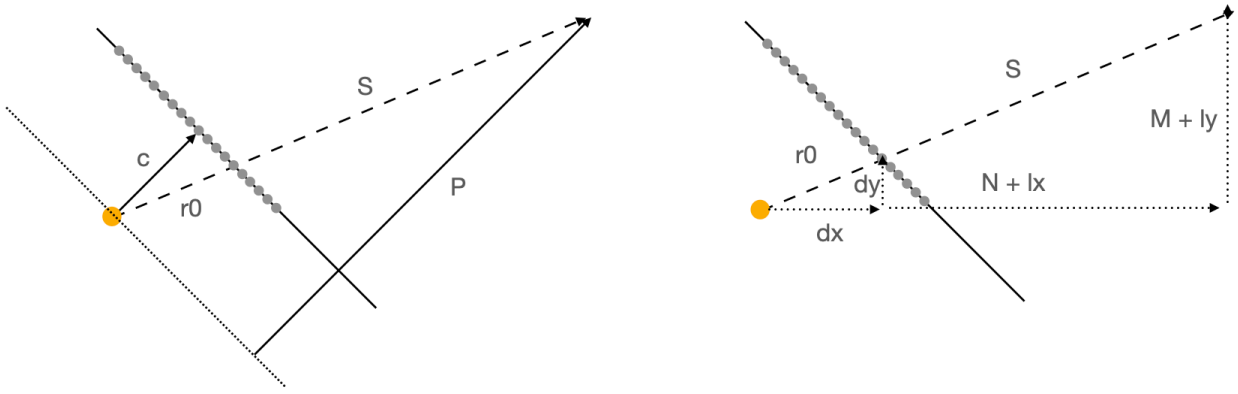

Figure 6: We use similar triangles to determine an expression for P, the perpendicular i.e far-field distance from the player position.

where P and S are the perpendicular and slant distances from the player position while c is the (constant) perpendicular distance to the virtual camera plane (VCP) and

$$r_0 = \sqrt{dx^2 + dy^2}$$

is the length of the ray from the player position to the corresponding point in the VCP.

Referencing Figures 2 and 3, the distance from the player to a map edge consists of an integer number of map squares in the x- and y- directions plus the fractional parts lx and ly. In the right diagram of Figure 6, S is broken down into the x- and y- distances N+lx and M+ly, where N and M are the integer portions and lx and ly are the fractional portions. The similar triangles give

$$\frac{N + lx}{S} = \frac{dx}{r_0}, \quad \frac{M + ly}{S} = \frac{dy}{r_0}$$

which combined with the previous expression gives the perpendicular distance as

$$P_x = c\frac{N + lx}{dx}, \quad P_y = c\frac{M + ly}{dy}$$

depending on whether the x- or y-side was hit by the ray caster.

The integer portions N or M are simply the number of steps taken by the ray caster minus one and are trivial to calculate as

$$N = (\text{mapX} - \text{playerX} - 1), \quad M = (\text{mapY} - \text{playerY} - 1)$$

where the need to subtract one is easy to see in Figure 2 by noting that the ray caster will take three steps but there are only two integer steps to the map edge.

Once the perpendicular distance is known the line height may be calculated. A basic 3D projection result is that the height of the line to be drawn is the inverse of the distance times a magnification factor d, hence for a hit in the x-direction the height h is

$$h_x = \frac{d}{P_x} = \frac{d}{c}\frac{dx}{N + lx} = g\frac{dx}{N + lx}$$

and similarly for a hit in the y-direction,

$$h_y = \frac{d}{P_y} = \frac{d}{c}\frac{dy}{M + ly} = g\frac{dy}{M + ly}$$

where the constant g is a composite magnification factor d/c that may be chosen as desired to make the rendered screen look nice. The line height for each camera ray is

thus remarkably straightforward to compute using quantities already computed either for or by the ray caster, without the need for square roots or trigonometric functions or any other complex mathematical operations.

3.3 Where it hit (index into the texture map)

A texture is an m x n image that is mapped onto the side that is hit. The first task is to compute the x-index into the texture. Note that whether a side is hit in the x- or y-direction, the texture index is always along the x-axis of the texture image.

The index is determined by the coordinate where the ray intersects the map edge. Referencing Figure 6, if the side is hit in the x-direction the y-distance moved is

$$\frac{dy}{dx}(N + lx) = dy\frac{N + lx}{dx}$$

and similarly, if the side is hit in the y-direction the distance moved is

$$\frac{dx}{dy}(M + ly) = dx\frac{M + ly}{dy}.$$

These expressions use the same values as used in the calculation of the height, hence the distance traveled is simply

$$dx\frac{g}{h_y} \quad \text{or} \quad dy\frac{g}{h_x}$$

and for a player located at (x0,y0) the x/y coordinate of the hit is therefore

$$\left(x0 + dx\frac{g}{h_y}, \quad y0 + dy\frac{g}{h_x}\right)$$

where the x-coordinate is used when the y-side is hit and the y-coordinate is used when the x-side is hit (staring at Figure 2 is helpful to visualize this). Since the map edges are all at integer spacing the texture index is simply the fractional portion of the above times the texture width n, hence

$$n \times \text{frac}\left(y0 + dy\frac{g}{h_x}\right) \quad \text{or} \quad n \times \text{frac}\left(x0 + dx\frac{g}{h_y}\right)$$

depending on whether the x- or y- side was hit respectively, where frac() denotes the fractional part of the result. Note that n is the multiplier in both cases, because the texture index is always along the x-axis of the texture image.

In Figure 2 the ray hits the left x-edge of the map square. In the texture coordinate system the first texture column (Tx=0) is at map coordinate y=3 while the final column (Tx=N) corresponds to map coordinate y=2, thus the texture and map coordinate systems are inverted. That is, the above expressions are only correct for the y-edge from below and the x-edge from the right but will produce an inverted image when hitting the leftmost x-edge (as in figure 2) or the topmost y-edge. For the latter cases the modified index expressions are

$$n \times \left(1 - \text{frac}\left(y0 + dy\frac{g}{h_x}\right)\right) \quad \text{or} \quad n \times \left(1 - \text{frac}\left(x0 + dx\frac{g}{h_y}\right)\right).$$

3.4 Texture height scaling

The last step is to scale the texture by the render height h. Each ray results in a vertical line that either compresses or expands a single column in the texture. Figure 7 illustrates three cases of interest. As the player moves towards a textured wall the line height starts smaller than the screen height, growing to the size of the screen when close, to beyond the edges of the screen when very close.

Mapping a texture of height m to a vertical line of height h requires a) a starting point within the texture column and b) the step size to step through the texture column. For example, if the rendered height on the screen is h=3, the texture map will consist of three pixels: the first, middle, and last row in the texture column. In this case the starting point will be zero and the step will be (m-1)/2 giving the points 0, (m-1)/2, and (m-1).

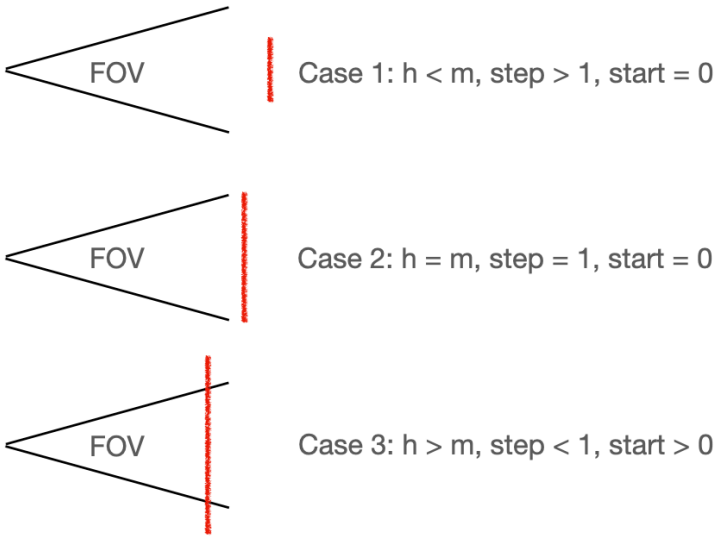

Figure 7: Texture scaling at different heights.

In the topmost image of Figure 7 the wall is far away and the vertical line height h is smaller than the texture height m, so only a subset of the texture will be rendered. The renderer will start at the first pixel in the texture column (start index = 0) and step through the texture with a relatively large step, greater than one.

In the middle image the wall is closer and the vertical line height h is exactly the texture height m — in Crazy, the texture height is exactly the screen height of 160 pixels. The texture is rendered one for one, with a step size equal to one and a starting index of zero.

In the bottom image the wall is very close, larger than the screen. In this case parts of the texture are off-screen and only a subset of the texture is rendered: the starting index is now greater than zero and the step size is less than one, which will have the effect of stretching the texture pixels.

For Crazy, where the screen height and texture height are both 160 pixels, with indices 0..159, the texture scaling algorithm is

```
step = 159 / (height-1)
if step >= 1 [if height <= 160]
     start = 0
else
     start = 80 - 80*step
end
```

When step < 1 the scaled texture is larger than the screen. The center of the texture should be in the center of the screen, and the texture is rendered by taking 80 steps (half the screen height) to both sides of the texture center. Hence the starting point is the center of the texture minus 80 steps. Note that height=1 is treated as a special case.

3.5 Summary

Gaze then upon the beauty that is a 3D ray caster: a group of simple routines that all share few basic quantities with no complex mathematical calculations and run extremely fast, able to generate very complex 3D scenes with an almost trivial amount of effort. A marvelous example of an old-school algorithm, using extra thinking to replace brute force calculations. An algorithm that really wants to work, and is within the capabilities of a SuperCPU.

4 SuperCPU Implementation

This section describes the SuperCPU implementation of the raycasting and rendering routines. Crazy uses multicolor bitmap mode with a 160x160 screen. Rows 160-199 are left available for future games to use, e.g. for health displays, radar, and so forth. The routines are implemented more or less as a library of routines, thus anyone can use them. The library is described in chapter 5, with this chapter focusing on the source code implementation and related issues.

I made extensive use of several of the old SCPU references, online versions of which are helpfully available. The most important are:

- SuperCPU user's guide: https://commodore.software/downloads?task=download.send&id=6613:supercpu-128-v2-manual&catid=213
- Commodore World issue 16 (Introduction to the 65816): https://commodore.software/downloads?task=download.send&id=12760:commodore-world-issue-16&catid=636
- Commodore World issue 19 (SuperCPU RAM expansion and timing): https://commodore.software/downloads?task=download.send&id=12763:commodore-world-issue-19&catid=636

4.1 Background

All calculations are done using 16-bit fixed-point math, and in one important case 17-bit. 8-bit calculations were tried initially for speed but 8 bits simply does not provide enough resolution for the ray caster, height, or texture calculations, while 16/17 bits works well.

All values are stored as 8.8 fixed point numbers: 8 bits for the integer portion and 8 bits for the fractional or remainder portion. This simplifies many calculations, especially since they tend to depend on either the integer or fractional portion of the number. Sometimes this is denoted as N.R, where N is the integer and R the remainder, or as x = xh + xl/256, where xh (high byte) is the integer portion and xl (low byte) the remainder.

To multiply two numbers together the fast multiply routine is used extensively. A fast multiply uses a table of f(x) = x^2/4 to compute a*b as f(a+b) - f(a-b). The fast multiply is described in some detail, including for signed numbers, in C=Hacking issue #16, available at http://www.ffd2.com/fridge/chacking/c=hacking16.txt

The C=Hacking article also describes how to do 16-bit projections of the form x*d/z, which is exactly the form of the raycaster height and texture index routines. The idea is to shift both x and z right until z' (the shifted z) is eight bits, use a table lookup for d/z', and then do an 8-bit fast multiply of x' and d/z' with a 16-bit result.

Although 16-bit arithmetic is used the code is all written in (optimized) 8-bit mode on the 65816. The reason is a bit lame: originally everything was written using 8-bit arithmetic on the hopes that it would work well enough and be much faster. Only late in the project, once the rendering routines were up and running, did it become apparent that 8-bits was not sufficient and 16-bits would be required. By this time the

focus was on getting the rendering working, and rather than rewrite and debug all the code into 16-bits the already working eight-bit routines were simply extended to perform the 16-bit arithmetic. I have tried out some 16-bit versions of key routines, and think they would make a noticeable improvement, but simply have not have the time to redo the code. Sounds like a great opportunity for some motivated individual to improve the code.

Rendering the screen with texture mapping is by far the most time consuming operation and so got the most optimization, followed by the iterative ray casting routine. There's surely plenty of room for improvement in all the routines, and I highly encourage you to figure out better and faster ways to do all this stuff.

Finally, the code below is all written in Slang. Slang includes the full Sirius assembler, allowing a combination of higher level commands (like if/then, repeat- and for-loops, subroutines, etc.) and assembly language. I am very happy with how this worked out in practice: it allowed routines to be prototyped quickly to get them working, then optimized in assembly where needed while using Slang routines for things that are tedious to do in assembly. Slang is what made this all possible, and I highly recommend it (and not just because I wrote it, like a bazillion years ago). For more information on Slang the home page is (amazingly) still up and running at http://www.ffd2.com/fridge/slang/

4.1.1 Quadrants and initial values

Crazy is optimized using quadrants. The ray caster sends out rays in directions (dx,dy), tracking the map location (mapx,mapy) as it goes. Using the .Y and .X registers to track mapx and mapy during the raycasting iterations results in an efficient algorithm, and this is enabled by considering direction quadrants.

Because dx and dy may be positive or negative there are four different quadrants for the algorithm to work in: (dx,dy) = (+,+), (dx,dy) = (+,-), (dx,dy) = (-,+), and (dx,dy) = (-,-). In Crazy, dx and dy are made always positive and the quadrant is stored explicitly. The quadrant determines whether the ray caster algorithm increments or decrements .X and .Y, i.e. whether it uses INX/INY or DEX/DEY.

The quadrant also determines the values of the fractional terms lx and ly. Referencing Figure 3, the initial map coordinates mapx and mapy are simply the integer portion of xpos and ypos, while the terms lx and ly are the x and y distance to the nearest edge. If dx < 0 then lx is the distance to the left edge and lx = R, the remainder term. If dx > 0 then lx is the distance to the right edge and lx = 1-R. The same logic holds for dy < 0 and dy > 0, and the setup code thus looks like

```
if dx<0
    quadrant = 2
    dx = -dx
    lda xpos    ;edge is to the left, dist=xpos
    sta lx
else
    quadrant = 0
```

```
    lda #00     ;edge is to the right, dist=256-xpos
    sec
    sbc xpos ;just the lower 8-bits
    sta lx
endif

if dy<0
    quadrant++
    dy = -dy
    lda ypos
    sta ly
else
    lda #00
    sec
    sbc ypos
    sta ly
endif
```

An important special case exists when we are exactly on a map boundary, with R=0. The values of lx and ly represent the distance to the next map edge, so one might wonder whether this is the current edge (lx=0) or an adjacent one (lx=1). The rule is that if dx is negative then lx=0 and if dx is positive then lx=1, with similar consideration for dy and ly — exactly as the code above gives. (An intuitive way to verify this is to consider very small R, e.g. R=0.001, in Figure 3.)

4.2 Rendering the screen

The DrawCamera routine casts a fan of rays about the look direction (dirx,diry) and draws a vertical line when a hit occurs. It is easiest to move left and right from the look direction, so two virtual camera planes are used: (rxplus,ryplus) for rays to the right of the look direction and (rxminus,ryminus) for rays to the left. At each iteration a ray is cast, and if a hit occurs the raycaster returns the map square value, the side that was hit, the height, and the texture index, all of which are used to draw a vertical line.

```
  repeat
    RayCast(rxplus,ryplus,xpos,ypos)
    if RayCast_hit > 0
         ldx RayCast_hit      ;map value
         lda RayCast_side
         bpl :xp
         lda ymaptex,x
         bra :psta
:xp      lda xmaptex,x
:psta    sta texture
```

```
      DrawVLine(xp, texture, RayCast_tx, &
             RayCast_height, buffer)
    endif

    RayCast(rxminus,ryminus,xpos,ypos)
    if RayCast_hit > 0
         ldx RayCast_hit ;map value
         lda RayCast_side
         bpl :xm
         lda ymaptex,x
         bra :msta
:xm      lda xmaptex,x
:msta    sta texture

      DrawVLine(xm, texture, RayCast_tx, &
         RayCast_height, buffer)
    endif
;
; Take a step in the + and - dirs
;
    rxplus = rxplus+DY128
    ryplus = ryplus-DX128
    xp++

    rxminus = rxminus-DY128
    ryminus = ryminus+DX128
    xm--

    count--
  until count=0
```

The map is stored as a 64x64 array of values, where a zero value indicates an empty square. In Crazy each map square may have a different texture for the x- and y-sides. Rather than store a specific texture number in each map square the map stores an index into a texture table: in the above code, the map value (RayCast_hit) is used as an index into the tables xmaptex and ymaptex depending on whether the x-side or y-side was hit, and the resulting texture value is then passed to DrawVLine.

Using this technique different textures may be used on all four sides of the map square but Crazy implements this only for x- and y-sides. This approach also makes it possible to change textures without changing the map, which is very useful during development and also enables things like animated textures.

4.3 Ray Caster Implementation

Recall that the raycaster routine is very simple to describe:

```
deltaXY = lx/dx - ly/dy
while map(mapx,mapy)!= 0
     if deltaXY < 0
          mapx++
          deltaXY += 1/dx
     else
          mapy++
          deltaXY -= 1/dy
     end
end
```

The implementation turns out to be fairly involved because of all the setup work needed to allow this routine to run fast, in particular the quantities 1/dx, 1/dy, lx/dx, ly/dy, and deltaxy = lx/dx - ly/dy. The first step is to calculate 1/dx, 1/dy, and deltaxy = lx/dx - ly/dy:

```
jsr SetDxDy          ;Set up vars
jsr CalcDeltaXY      ;Calculate deltas
```

SetDxDy sets the quadrant, sets lx and ly based on the quadrant, and makes dx and dy positive, using the code in section 4.1.1.

The inverse quantities 1/dx and 1/dy are calculated by shifting dx or dy until they are 8-bits, then doing a table lookup, then shifting them back. This is a part of the code that could be made faster by using a 16-bit table lookup, at the cost of a bank of SuperRAM. The results are then multiplied by lx and ly, using multiple fast multiply routines.

With the preliminaries in place four raycaster routines are used, one for each quadrant; the “plus/plus” routine is shown below, corresponding to dx>0, dy>0. The map locations are first stored in .Y and .X, and a maximum iteration count is set:

```
;
; dx,dy > 0
;
raypp
          ldy xpos+1    ;mapx (integer portion of xpos)
          ldx ypos+1    ;mapy

          lda #00
          sta hit
          lda #16       ;max iterations
          sta iter
```

The x-coordinate is stored in .Y and the y-coordinate is stored in .X, which seems backwards but is done this way for the map lookup later in the routine. The maximum number of iterations is set at 16. The number of iterations controls how far one can

“see” in the map. A higher number of iterations will render far-off map squares on the screen but slow things down. Sixteen was chosen as a compromise between having a screen that looks pretty good while remaining pretty fast.

The main loop then runs, adding or subtracting to deltaxy as it iterates along the ray:

```
:loop
          ror side     ;C contains result of sbc
          bmi :pos

          iny          ;inc mapx
          lda deltaxy
          clc
          adc invdx
          sta deltaxy
          lda deltaxy+1
          adc invdx+1
          sta deltaxy+1
          bra :cont
:pos
          inx          ;inc mapy
          lda deltaxy
          sec
          sbc invdy
          sta deltaxy
          lda deltaxy+1
          sbc invdy+1
          sta deltaxy+1
:cont
          lda mapylo,x ;we could optimize
          sta zp       ;but this is cleaner
          lda mapyhi,x
          sta zp+1
          lda (zp),y ;map x/y
          bne :hit

          dec iter
          bne :loop    ;.A=0
```

The .X and .Y registers store the map location and are incremented appropriately. The variable deltaxy is calculated as a 17-bit number, using the carry bit to indicate the sign. The sign determines whether the next step is in the x- or y- direction, thus the sign is stored in the “side” variable *before* each iteration: if a map square is hit after the iteration, it stores whether it the last step was in the x- or y-direction. This is the value checked in the main loop in section 4.2 to determine which side was hit, and hence which texture to use.

The map is stored as a 64x64 array at $c000-$cfff. A simple lookup table is used to compute the map row, and .Y is used as the column index.

About this time you may be saying to yourself: surely this could be optimized using 16-bit instructions, and the map lookup could be sped up using an alternative memory arrangement up in SuperRAM, and this seems like a very 8-bit approach to a 16-bit problem. And you would be right. But almost any way you work it there has to be a switch between 8 and 16 bits, and a PHA/PLA in there, and any time you use SuperRAM you take a timing hit, and overall you run into the chronic 65816 problem of too few registers combined with 8-bit RAM. Using 16-bit registers you save maybe 5 cycles overall per iteration; using SuperRAM maybe you could get another 10 or so. Is it worth it, at 20 MHz? Could be. Sounds like a great opportunity for a young or not-so-young coder to try it out and see what happens.

4.4 Calculating the height

When an occupied map square is hit the next step is to calculate the height of the line to be drawn. From section 3.2 the height is given by

$$h_x = g\frac{dx}{N + lx}, \quad h_y = g\frac{dy}{M + ly}$$

depending on whether the hit was in the x- or y-direction. The quantities dx and dy are already known and lx and ly are set based on the quadrant before calling the ray caster. Thus the next step is determining M or N, the integer distance in the x- or y-direction.

In Figure 2, the raycaster starts at the square (mapx,mapy) = (1,1) and will take three steps before arriving at the green map square (mapx,mapy) = (4,2). Although the raycaster takes three steps, from the figure the integer distance is just two steps and therefore the integer distance is N = final_mapx - start_mapx - 1, i.e. we have to subtract one from the number of steps the algorithm takes.

There is however one very important special case, that of being exactly on a map edge such that the fractional part lx=0 or ly=0. If the player position in Figure 2 were exactly on the left edge, the next edge is a full integer distance away and we do not subtract one. The code to compute the integer distance in the x-direction is

```
          lda side
          bmi :yside
:xside
          tya           ;mapx
          clc
          ldy lx        ;check for lx=0
          bne :sbc      ;set or clear c
          sec
:sbc      sbc xpos+1    ;n-1, n if lx=0
          clc ;GetHeight flag
          jmp GetHeight
```

with similar code for a hit in the y-direction.

On a modern computer, using a high level language, the distance lx to the nearest edge would be calculated in floating point as lx=1. The issue here is that we are tracking the integer and fractional parts separately, and since the fractional part is always less than one this gets treated as a special case.

Once the integer distance is known, the appropriate height expressions are rewritten as

$$h_x = dx\frac{g}{N+lx}, \quad h_y = dy\frac{g}{M+ly}$$

and calculated using the C=Hacking projection method described in the introduction to section 4: both dx and N+lx are shifted right until the denominator is 8-bits, the fraction is determined using a lookup table of g/x, and the value is multiplied by the shifted dx using a series of fast multiply routines.

The resulting height can be very large when one is close to the wall, i.e. when N or M is zero, and thus the height is calculated as a 16-bit value. In Crazy it can be as large as 1024, which covers most cases of interest. This is another example of 8 bits not being sufficient, even though the screen is only 160 pixels high.

4.4 Calculating the texture x-index

From section 3.3, the ray hits the wall at either the x- or y-coordinate given by

$$\left(x0 \pm dx\frac{g}{h_y}, \quad y0 \pm dy\frac{g}{h_x} \right)$$

where h is the height just computed. The texture index is therefore calculated using the same table of g/x, using the height h as the index, once again using the trick of shifting h and dx (or h and dy) if h is greater than 8-bits. The result is then multiplied by dx or dy to get the relative distance moved from the starting position.

To get the final coordinate we add or subtract the relative distance from (x0,y0), depending on the quadrant, and multiply the fractional part by the texture width (80 in Crazy) using another lookup table of 80/x. At this point yet another subtlety kicks in regarding texture orientation. Texture columns are numbered 0 to 79 left to right as rendered on the screen. Referencing Figure 2 yet again, when hitting a y-edge from below (dy > 0) the texture columns map from 0 to 79 left-to-right. However, vieweing the upper y-edge from above (dy < 0) will result in a flipped image when using the same mapping, with a similar result along the x-edges. Therefore textures must be flipped when hitting the left x-edge (dx > 0) or the upper y-edge (dy < 0).

The code for this is not very interesting and omitted here — lots of fast multiplies and such. Those interested should check the source code.

4.5 Drawing a vertical line

The last step is to draw the texture as a vertical line on the screen. This is probably the most challenging part of Crazy and thus will be discussed in a series of steps, gradually working up to the full routine to understand the logic of it all.

Crazy uses two screen buffers, with the first screen at $6000-$7FFF and the second at $A000-$BFFF. SuperCPU optimization is used so that memory is written to the C64 only for the buffer in use. Writes to these buffers are always at 1 MHz.

4.5.1 Vertical line blaster, no texture

To start, consider a vertical line blaster: a big unrolled loop that stores a bit pattern into a bitmap buffer. This is not the actual code used by Crazy, but we start here to illustrate the concept. First consider a fixed bit pattern. If the bit pattern is stored in .Y and the column offset in .X, then the code would look something like

```
tya
ora row0,x
sta row0,x
tya
ora row1,x
sta row1,x
…
tya
ora row159,x
sta row159,x
rts
```

The memory arrangement of the C64 bitmap means two routines are needed, one for columns 0-31 (multicolor pixels 0-127) and another for columns 32-39 (multicolor pixels 128-159). The idea is to calculate the start and end rows, putting an rts at the ending point (self-modifying code, gasp!) and performing a jsr into the entry point, to either one of the two routines depending on the column being written to. Note that a blaster routine like this does not run at the full 20 MHz since the buffer writes are done at 1 MHz, however it is still the most efficient method.

4.5.1 Texture column offsets

Texturing complicates things considerably. We need to map a texture column to a screen column, where the texture is scaled. The first problem is that there is not a 1:1 correspondence between the texture map and the screen map. As a reminder, the C64 bitmap is arranged in 8x8 chunks as

byte 0	byte 8	byte 16	…
byte 1	byte 9	byte 17	…
byte 2	byte 10	byte 18	…
…	…	…	…
byte 7	byte 15	byte 23	…

where each multicolor pixel consists of two bits, hence there are four pixels per 8-bit screen column. We can't just use a bit mask on the texture map, since any four texture pixels in a texture column could map to any four screen pixels in a screen column. The first challenge is thus to map an arbitrary texture column, one of four bit pairs within a byte, to an arbitrary bitmap column, also one of four bit pairs.

The second challenge is that the texture column is scaled. Section 3.3 derived an expression for the texture column to use, while section 3.4 derived expressions for the starting point in the texture and the step size within the texture column. Therefore the task of the texture mapper is to take an arbitrary texture column, step through it from an arbitrary starting point using an arbitrary step size, and map that into an arbitrary C64 bitmap column, all within the C64 multicolor memory map.

To first order, what is needed is a modified line blaster along the lines of

```
lda [texp],y        ;texture pointer
ora row_n,x
sta row_n,x
(increment y by texture step)
lda [texp],y
ora row_n+1,x
sta row_n+1,x
```

The idea is to use [texp] as a pointer to the texture column and .Y as an index into the column, then step through the texture and ORA it into the bitmap. The first challenge with mapping texture columns is that a given texture pixel pair can map into any of four different bitmap offsets. For example, if the texture pixel is '11', then this could map to a bitmask of '11000000', '00110000', '00001100', or '00000011', depending on the bitmap pixel column. To solve this problem Crazy stores four copies of the texture: the first copy has each 2-bit pixel column in the first bit pair position, the second copy has each column in the second bit pair position, and so on.

Each texture is 80x160 = 12,800 pixels = $3200 bytes. Therefore the four offset textures may be stored in a single bank of SuperRAM: $0000-$3200, $4000-$7200, $8000-$B200, $C000-$F200, where each texture is stored as follows:

0-159	column 1
160-319	column 2
320-479	column 3
…	

(This is partly why textures are chosen as 80x160 — big enough to look good on the screen but small enough to fit four copies in a single 64k bank of RAM.) The texture

pointer [texp] is then set up to point to the appropriate texture map column in the appropriate bank of SuperRAM, and code along the above lines may be used.

4.5.2 Scaled texture columns

The next challenge is scaling the texture columns. It is far too cumbersome to increment .Y each time by some scale factor. Since most scale factors are less than one this becomes a 16-bit operation, consuming far too many cycles and too much memory, so a better method is needed. The idea is to try to use something like

```
ldy tex_index
lda [texp],y        ;texture pointer
ora row_0,x
sta row_0,x
ldy tex_index+1
lda [texp],y        ;texture pointer
ora row_1,x
sta row_1,x
```

That is, if we can load the texture index from a pre-computed table things will be much more efficient. What Crazy does is compute a mapping of texture columns to screen columns, one for each height, and stores them in SuperRAM. For example, a vertical line of height=3 will occupy screen rows 79, 80, and 81. The corresponding scaled texture uses texture rows 0, 80, and 159, i.e. the first point, midpoint, and last point in the texture column. In this example, the mapping from texture pixels to screen pixels is

screen pixel row	texture pixel row
rows 0-78	n/a
row 79	0
row 80	80
row 81	159
rows 82-159	n/a

which can be stored as a 159-byte table with bytes 0-78 and 82-159 random, byte 79=0, byte 80=80, and byte 81=159. A similar table can be generated for every possible height as determined by the raycaster.

Since the textures don't use a full bank of SuperRAM there is RAM available in each bank to store tables. If tables were stored in every bank of SuperRAM at $0xfc00, one bank for each height, the data bank register DBR may be used to select which table to use based on the height. The blaster routine then becomes

```
ldy $fc00           ;row 0
lda [texp],y        ;texture pointer
ora $006000,x
sta $006000,x
```

```
ldy $fc01           ;row 1
lda [texp],y
ora $006001,x
sta $006001,x
…
```

For example, the height=3 table is stored in bank 4, so DBR would be set to 4 before calling the blaster routine. The ldy $fc00 then becomes effectively ldy $04fc00, while the ora/sta are unchanged as they use the long address form to write to bank 0. Because the line is three pixels high, it only reads the three entries at $fc00+79, $fc00+80, and $fc00+81, and plots the corresponding texture pixels. And the same routine can be used for any texture simply by changing texp. Slick.

4.5.3 Large scaled texture columns

Since there are only 256 banks of SuperRAM available, the next challenge is for large heights, e.g. texture map heights greater than 256. There are two issues. The first is that the upper banks of SuperRAM are used by the system and not available for tables, thus 8-bit heights > 238 or so do not have tables. The second is the case of heights > 256, e.g. height=640, since the tables only go up to 238 or so.

The solution to the second problem is to create additional tables, one for each high byte value of the height. Thus

height=$00-$ff	tables at $fc00
height=$100-$1ff	tables at $fd00
height=$200-$2ff	tables at $fe00
height=$300-$3ff	tables at $ff00

To solve the problem of heights with low byte > 238, we simply check the low byte and set it to 238 if it is greater than 238. This sacrifices a tiny bit of resolution when textures are really large, but if I hadn't told you about it you never would have noticed.

Four sets of tables requires four sets of line blaster routines: one with ldy $fc00, one with ldy $fd00, one with ldy $fe00, and one with $ff00. And there needs to be two routines in each case, one for screen columns < 32 and one for screen columns >= 32, for a total of eight line blaster routines. And then there needs to be two sets total, one for each bitmap buffer, for a total of sixteen routines.

In Crazy, these routines all fit in bank 2 since the blaster code is so compact. (Bank 1 is a better choice, but bank 1 is unfortunately used by Slang so I left it alone for now. Someone ought to try moving it to bank 1 and seeing how much speedup results.). The core code element is

```
ldy textab
lda [texp],y
ora $123456,x
sta $123456,x
```

which is 13 bytes, times 160 rows is 2080 bytes per blaster, stored as follows

$020000-$023fff buffer 0, screen columns 0-31
    $020000 height=0..$ff ($fc00)
    $021000 height=$100-$1ff
    $022000 height=$200-$2ff
    $023000 height=$300-$3ff

$024000-$027fff buffer 0, screen columns 32-39
    $024000 height=0..$ff ($fc00)
    $025000 height=$100-$1ff
    $026000 height=$200-$2ff
    $027000 height=$300-$3ff

$028000-$02bfff buffer 1, screen columns 0-31
    $028000 height=0..$ff ($fc00)
    $029000 height=$100-$1ff
    $02a000 height=$200-$2ff
    $02b000 height=$300-$3ff

$02c000-$02cfff buffer 1, screen columns 32-39
    $02c000 height=0..$ff ($fc00)
    $02d000 height=$100-$1ff
    $02e000 height=$200-$2ff
    $02f000 height=$300-$3ff

And since only 2080 bytes or so are used for each blaster there is still room at the top of bank 2 for the texture height tables at $02fc00-$02ffff.

4.5.4 Optimal column ordering

There is one last optimization in the Crazy line routine, that takes advantage of the relationship of line height to screen position. Vertical lines are always drawn centered on the screen: a line of height 1 will be a single pixel in row 80 (middle of the screen). A line of height 2 will be a pixel in rows 79 and 80. A line of height 3 will be pixels in rows 79, 80, and 81.

That is, vertical lines are not plotted to arbitrary locations on the screen. Rather, a line of height n always begins and ends at the exact same row on the screen — a line of height 3 is always at rows 79-81, never e.g. at the top of the screen — and lines grow by adding one pixel to the line, alternating between the top and bottom of the line. Thus the line blaster can be re-ordered as

    plot row 0
    plot row 159

plot row 1
plot row 158
…
plot row 79
plot row 81
plot row 80
rtl

A line of height 1 is plotted using the last line of routine. A line of height 2 is plotted using the last two lines of the routine. And a line of height h is plotted using the last h lines of the routine. Therefore we set an entry point based on the height, using a lookup table, jsl to the appropriate bank 2 routine, and let it rip. This approach eliminates any self-modifying code and becomes significantly more efficient.

4.6 Vertical line texture mapping: DrawVLine

We are now in a position to describe the vertical line drawing routine, DrawVLine.

Textures are unpacked and stored in SuperRAM, one texture per 64k bank. Four copies of the texture are stored, each shifted by one multicolor pixel (two bits) corresponding to the bitmasks 11000000, 00110000, 00001100, and 00000011. That is, the first copy unpacks all the texture pixels into column 1 (bitmask 11000000), the second copy unpacks all the texture pixels into column 2, etc. Each texture copy takes up $3200 bytes, and the copies are stored at offsets $0000, $4000, $8000, and $c000. A total of 238 textures may be used; Crazy uses 58.

To scale the textures a series of tables, or more specifically column mappings, are generated, one for every possible texture height. The column mapping for height=1 is stored in bank 2 at $02fc00, the mapping for height=2 is stored in bank 3 at $03fc00, and so forth. Texture heights of value $01xx are stored at $02fd00, $03fd00, etc.; heights of values $02xx are stored at $02fe00, $03fe00, etc.; and heights of value $03xx are stored at $02ff00, $03ff00, etc. Thus texture heights up to 1023 are permitted.

Vertical lines are drawn using a line blaster routine of the form

```
ldy $fc00           ;or $fd00, $fe00, $ff00
lda [texp],y        ;texture pointer
ora $006000+screen_row_offset,x
sta $006000+screen_row_offset,x
```

with one code fragment per row in one large unrolled loop. Before calling the routine [texp] is set to point to the texture column to be mapped, where the upper 8 bits determines the bank and hence texture to use, while the lower 16 bits point to the texture column, as determined by the ray caster, in the correct bitmap offset, as

determined by the screen coordinate. The data bank register (DBR) is also set to determine which scaling table to use, based on the height.

Four separate routines are used for heights $00xx, $01xx, $02xx, and $03xx, using the tables at $fc00, $fd00, $fe00, and $ff00. Two sets of routines are used for screen columns 0-31 and 32-39, and two versions are used for buffers at $6000 and $a000, for a total of sixteen different routines. These sixteen routines are stored in bank 2 at $020000, $024000, $028000, and $02c000.

The routine DrawVLine first sets the texture RAM bank to texture number + 3:

```
          lda texnum
          jsr SetTexture

sub SetTexture(@a)    ;split out to be
          clc
          adc #3
          sta texp+2   ;callable elsewhere
          rts
```

The routine then checks the height to ensure it is in range and sets the height of the line to be drawn in screen pixels (max 160). The texture pointer is then set. The bank was set above by SetTexture. The bitmap column determines whether to use the texture map at $0000, $4000, $8000, or $c000, and since each column is 160 bytes the column begins at texture_x*160, where texture_x is the texture location computed in section 4.4:

```
;
; Set up texture pointer
;
;          texp = tx*160  ;column offset

          ldx tx
          cpx #80
          bcc :cx1
          ldx #79
:cx1      lda lo160,x
          sta texp
          lda hi160,x
          sta texp+1

          lda x           ;add in base
          and #03
          tax
          lda :textab,x
          clc
          adc texp+1
          sta texp+1
```

The blaster entry point is then computed using a table lookup

```
:code1
         lda Code1RowLo,y
         sta :jmpaddr+1
         lda Code1RowHi,y
         clc
         adc hoffset     ;different heights
         sta :jmpaddr+2
         bra :next
```

and finally DBR is set and the appropriate routine is called. Before changing DBR, SEI is used to disable interrupts. Without the SEI the system gets horribly confused. The code also switches to RAM at $a000. If this were just a demo then this could be done once at the beginning of the code, but this is a library that is designed to be compatible with games and programs that use kernal routines in $a000, including Slang (math routines, etc.).

```
;
; Draw the line
;
         sei           ;nonzero dbr
         lda $01
         pha
         and #$fe      ;RAM at $a000
         sta $01       ;so ORA works
         phb
         lda height
         bne :inc      ;things like height=512 have low
byte=0
         inc           ;so we use texptr for 513 instead
:inc     inc
         pha
         plb
:jmpaddr jsl $020000
         plb
         pla
         sta $01
         cli
:rts     rts
```

5 Crazy and the Crazy code library

The routines just described are part of a callable library of routines that can be used by anyone to create 3D texture mapped programs, without having to understand all the details of those routines. Crazy is a separate program that calls these routines, and this section will describe the routine library and the program Crazy — and just how easy it is to create 3D textured worlds using the routines.

5.1 The Crazy library

Believe it or not, despite all the previous discussions of algorithms and code the entire ray casting library is just 4k in size, with another 8k of tables (not counting all the code and tables generated in SuperRAM). The memory map is as follows:

```
; Memory map:
;   3000-3fff: library code
;   4000-4bff: Open (sprites etc.)
;   4c00-4fff: color map, buffer 1
;   5000-5fff: tables
;   6000-7fff: buffer 1
;   8000-8bff: Open (sprites etc.)
;   8c00-8fff: color map, buffer 2
;   9000-9fff: tables, misc
;   a000-bfff: buffer 2
;   c000-cfff: map
```

The ray casting library sits at $3000 and consists of seven callable routines accessible via a jump table, described in the following subsections.

```
jmp InitGfx
jmp AddTexture
jmp InitBitmaps
jmp ClearBitmap
jmp SetVICBuffer
jmp DrawCamera
jmp RayCast
```

5.1.1 InitGfx

InitGfx initializes the SuperRAM blaster routines and texture scaling tables. It creates all of the routines in bank 2, and sets up the texture mappings at $fc00-$ffff in SuperRAM banks 3-240. Example usage:

```
InitGfx()
```

5.1.2 AddTexture

AddTexture adds a texture into SuperRAM. A texture is created by loading a regular 4-color 80x160 multicolor bitmap into $8000 and calling AddTexture with the texture number in .A (1-236). AddTexture then unpacks the bitmap into columns and stores the textures in SuperRAM. Note that no color map information is stored — it really needs to be just four colors. Example usage:

```
load "blues80x.bitmap",dev,$8000
AddTexture(2)
```

5.1.3 InitBitmaps

InitBitmaps takes four colors as arguments and sets up VIC and the bitmap color buffers at $4c00, $8c00, and $d800. Example usage:

```
InitBitmaps(black, red, blue, gray2)
```

5.1.4 ClearBitmap

ClearBitmap clears the bitmap buffer contained in .A (0 or 1). Example usage:

```
ClearBitmap(buffer)
```

5.1.5 SetVICBuffer

SetVICBuffer sets $dd00 to the buffer in .A (0 or 1). Example usage:

```
SetVICBuffer(buffer)
```

5.1.6 DrawCamera

DrawCamera is the main workhorse routine to draw a screen, i.e. the virtual camera plane. You give it a position and look direction and it does the rest, drawing the screen in the specified buffer. Example usage:

```
DrawCamera(theta,xpos,ypos,buffer)
```

5.1.7 RayCast

RayCast is the routine to cast a single ray. The thinking was that in a game or whatever one might use this to, say, fire a weapon and see what it hits. Eample usage:

```
RayCast(rxplus,ryplus,xpos,ypos)
```

5.2 The map

The map is a 64x64 array, stored at $c000. Two tables are used at $5000 (xmaptex) and $5080 (ymaptex) as a key to determine which textures correspond to which map values, as follows. Each map square has either zero or nonzero value. When a nonzero value is hit it is used as an index into these tables, where one table is for hits in the x-direction and the other is for hits in the y-direction. The table value determines the texture.

Therefore to create your own maps two things are needed: the map at $c000, and the key at $5000-$50ff. For Crazy I used a simple Matlab/Octave script to generate the map and key, given in Appendix 4.

It is important to have a nonzero border on the map. The raycaster does not check for map edges, but instead just keeps going until a nonzero value is hit.

There is nothing particularly special about 64x64. The ray caster uses the tables at $5e00 and $5e80 to look up the low and high bytes respectively of the map row addresses, using the code in section 4.3. If you're feeling adventurous you can modify these tables to allow much larger maps, up to 128x128 if you switch out ROM at $d000-$ffff.

5.3 Crazy

Crazy is written in Slang, which combines high level language commands with a built-in assembler, allowing one to mix and match as desired. Using Slang makes it very easy to get up and running quickly. The references section at the end of this paper contains links to the Slang documentation. A set of subroutine interface to the library routines are defined at the top:

```
sub asm InitGfx@$3000()
sub asm AddTexture@$3003(@a)
sub asm InitBitmaps@$3006( &
      byte color0@$f9, byte color1@$fa, &
      byte color2@$fb, byte color3@$fc &
      )
sub asm ClearBitmap@$3009(@a)
sub asm SetVICBuffer@$300c(@a)
sub asm DrawCamera@$300f(ubyte theta@$f7, int x0@$f8, int
y0@$fa, ubyte buf@$fc)
```

These are just little stubs that allow Slang to call these assembly routines as if they were Slang subroutines. The routines can be called easily from assembly language as well.

5.3.1 Initialization

The first step is to load the library tables at $5100,

```
sub LoadTables()
  byte dev@$ba

  load"tables5100-5fff",dev,$5100
endsub
```

followed by the map and map key:

```
sub InitMap()
  ubyte dev@$ba

  load "arena1.map",dev,$c000 ;map
  load "arena1.key",dev,$5000 ;symbol-texture map
endsub
```

Next, the raycaster library and textures are loaded in

```
sub LoadGfx()
  byte dev@$ba

  load"raycast3.o",dev,$3000

  CountTex(1)
  load "redbrick.bitmap",dev,$8000
  AddTexture(1)
  CountTex(2)
  load "blueston.bitmap",dev,$8000
  AddTexture(2)
  …
endsub
```

As seen in the code above, textures are loaded to $8000 and AddTexture is called to process the textures into SuperRAM. The subroutine CountTex() simply prints out the texture being loaded, to give the user a little feedback.

Finally, the library routines InitGfx(), InitBitmaps(), ClearBitmap(), and SetVicBuffer() are called to get the graphics set up, and we are off and running.

```
  InitGfx()
  buffer = 1
  InitBitmaps(black, red, blue, gray3)
  ClearBitmap(buffer)
  SetVICBuffer(buffer)
```

5.3.2 Position and direction

The Crazy code stores the user position (xpos,ypos) as a 16-bit fixed-point number, where the high byte is the integer portion and the low byte is the fractional, plus a look direction. The look direction theta ranges from 0 to 63, corresponding to 0-360 degrees, or about three degree resolution. The initial values are

```
xpos=512
ypos=1011
theta=16
```

5.3.3 Main loop

The main loop is almost absurdly simple. Here it is, in its entirety:

```
repeat
  Animate()
  buffer = buffer eor 1
  OptimizeSCPU(buffer)
  DrawCamera(theta,xpos,ypos,buffer)
  SetVICBuffer(buffer)

  ldx theta
  lda costablo,x    ;64*cos(theta)
  sta dirx
  lda costabhi,x
  sta dirx+1
  lda sintablo,x
  sta diry
  lda sintabhi,x
  sta diry+1
  jsr ReadJoy
forever
```

That's all it takes. The routine Animate() is an almost stupidly simple routine that updates the map key at $5000 to cycle through different textures. SuperCPU optimization is used to only copy RAM to the active bitmap buffer, the screen is rendered to the draw buffer (DrawCamera), and bring it forward (SetVICBuffer).

The routine ReadJoy() reads the joystick, turns left/right by increasing/decreasing theta, and moves forward/backwards by taking a trial step in the direction theta, checking if the map square is occupied, and moving forward if not. The code above to set dirx and diry should really be inside ReadJoy(). The tables costablo/hi and sintablo/hi are library routine tables that are used here as a convenience.

And that's it. So now it's your turn — to make the code better, to create some cool new programs, and to get out there and show what a C64+SCPU can really do!

Appendices: Scripts and useful links

The appendices give some useful references and describe the scripts used to generate the texture graphics and the map used in Crazy. The scripts are all written in Matlab/Octave, but are pretty easy to translate to any other language.

A.1 Useful links and references

lodev's ray casting tutorial: https://lodev.org/cgtutor/raycasting.html

SuperCPU user's guide: https://commodore.software/downloads?task=download.send&id=6613:supercpu-128-v2-manual&catid=213

Commodore World issue 16 (Introduction to the 65816): https://commodore.software/downloads?task=download.send&id=12760:commodore-world-issue-16&catid=636

Commodore World issue 19 (SuperCPU RAM expansion and timing): https://commodore.software/downloads?task=download.send&id=12763:commodore-world-issue-19&catid=636

C=Hacking Issue 16 (fast multiplication and division): http://www.ffd2.com/fridge/chacking/c=hacking16.txt

Slang home page (binaries and documentation): http://www.ffd2.com/fridge/slang/

A.2 Texture creation

To create a texture, I downloaded images from the web and used imagemagick downscale the images to 80x160 png format using a command like

```
convert face1.jpeg -resize 80x160\! face80x160.png
```

I then used the script "dither_test.m" to convert the image to C64 bitmap format, either using dithering or closest color match. In general 4-color dithering doesn't work well because the texture mapped image appears to "shimmer" as you move. I left a few in Crazy to illustrate the effect, and the Joker seen in Figure 1 worked pretty well (in fact the illusion of purple hair is so strong that you really have to zoom in close to see it as just red and blue pixels).

```
%
% Dither an image to create a 4-color texture file for C64 applications
%
% SLJ 1/25/2024
%

clear all;
```

```
%dtype = 1  % Floyd-Steinberg
%dtype = 2  % Sierra lite
%dtype = 3 % Stucki
dtype = 4 % Closest match

%fname = "eagle80x160.png";
%fname = "colorstone80x160.png";
%fname = "redbrick80x160.png";
%fname = "purplestone80x160.png";
%fname = "bluestone80x160.png";
%fname = "greystone80x160.png";
%fname = "redsmiley80x160.png";
%fname = "angrysmiley80x160.png"; % Use        192 49 29   % red
%fname = "mossy80x160.png";

%fname = "blues80x160.png";
%fname = "marilyn80x160.png";
%fname = "joker80x160.png";
%fname = "joker280x160.png";
%fname = "face80x160.png";
%fname = "stooges380x160.png";

%fname = "brick180.png";
%fname = "brick280.png";
%bad fname = "cstone180.png";
%fname = "whiteb180x160.png";
%bad fname = "whiteb280x160.png";
%fname = "blackb80x160.png";

%fname = "commodore80x160.png";
%fname = "red80x160.png";
%fname = "A80.png"

%fname = "girl1.png";
%fname = "girl280x160.png";
%fname = "girl380x160.png";
%fname = "girl480x160.png";
%fname = "left80x160.png";
%fname = "right80x160.png";
%fname = "painter80x160.png";
%fname = "viola180x160.png";
%fname = "ballet180x160.png";
%fname = "hockey180x160.png";
fname = "dino180x160.png";

data = imread(fname);

if length(size(data))==2  % Grayscale image
  dither(:,:,1) = data;
  dither(:,:,2) = data;
  dither(:,:,3) = data;
else
  dither = data;
end
texture = zeros(size(data(:,:,1))); % This will be color number, rather than
rgb
```

```
% r g and b values above are 0..255

if false
colors = [
        0 0 0     % black
        200 0 0   % red
        200 0 200 % violet
        200 200 200]; % lt gray
endif

if true % general colormap
colors = [
        0 0 0 %50 50 50     % dk gray
        192 49 29   % red
        0 0 150 % blue
        187 187 187 % lt gray
        ];
endif

if false % Use for red brick
colors = [
        0 0 0 % black
        192 49 29   % red
        96 25 15 % dk red
        255 128 128 % lt red
        ];
endif

if false % use for fonts
colors = [
        0 0 0 % black
        137 219 162 % cyan blue
        234 166 58   % orange (red)
        187 187 187 % lt gray
        ];
endif

if dtype==1
    % Floyd-Steiberg (1/16)
    %       X   7
    %  3    5   1
    dmat = [  % +row +col factor
      0 1 7/16
      1 -1 3/16
      0 -1 5/16
      1 1 1/16
      ];
elseif dtype==2
    % Sierra lite (1/4)
    %       X   2
    %  1    1
    dmat = [  % +row +col factor
      0 1 1/2
      1 -1 1/4
      0 -1 1/4
      ];
elseif dtype==3
      % Stucki
```

```
%              X   8   4
%     2   4    8   4   2
%     1   2    4   2   1
%
%           (1/42)
    dmat = [
      0 1 8/42
      0 2 4/42
      1 -2 2/42
      1 -1 4/42
      1 0 8/42
      1 1 4/42
      1 2 2/42
      2 -2 1/42
      2 -1 2/42
      2 0 4/42
      2 1 2/42
      2 2 1/42
    ];
elseif dtype==4 % closest match
    dmat = [0 0 0];
endif

% Now dither the image
rowmax = 160;
colmax = 80;
drow=0;
dcol=0;

for row=1:rowmax
  for col=1:colmax
    r = double(dither(row,col,1));
    g = double(dither(row,col,2));
    b = double(dither(row,col,3));

    % Find closest color

    mindist = 1e6;
    mincol = 0;
    for k=1:4
      d0 = (r-colors(k,1))^2 + (g-colors(k,2))^2 + (b-colors(k,3))^2;
      if d0 < mindist
       mincol=k;
       mindist = d0;
      endif
    endfor

    % Calculate error

    err_rgb = [
      r - colors(mincol,1)
      g - colors(mincol,2)
      b - colors(mincol,3)
      ];

    % And diffuse

    dither(row,col,:) = colors(mincol,:)';
```

```
    texture(row,col) = mincol-1; % 0..3
    for k=1:length(dmat(:,1))
      drow = row+dmat(k,1);
      dcol = col+dmat(k,2);
      c = dmat(k,3);
      if (drow <= rowmax) && (dcol <= colmax) && (dcol > 0)
        dither(drow,dcol,:) = squeeze(dither(drow,dcol,:)) + c*err_rgb;
      endif
    endfor

  endfor
endfor

% Display the results

figure(1); clf; imshow(data);
figure(1+dtype); clf; imshow(dither);


% Output a C64 bitmap, to check in VICE

k=0
bitmap = zeros(8*40*20);  # 320x200
for row=1:rowmax
  for col=1:colmax
    x = col-1;
    y = row-1;
    idx = 1 + bitand(y,248)*40 + bitand(y,7) + bitand(2*x,504);

    bitp = texture(row,col)*(4^(3-bitand(x,3)));
    bitmap(idx) = bitor(bitmap(idx),bitp);

    k=k+1;
    if (row==999)
      debug = [k x y idx bitmap(idx)]
    endif
  endfor
endfor

% Check to see if it worked

cmap = zeros(160,80);
idx = 1
y = 0
offset = 0
for row=1:160
  x=0;
  y=row-1;
  for col=1:40
    idx = 1 + bitand(y,248)*40 + bitand(y,7) + 8*(col-1);
    byte = bitmap(idx);
    x=x+1;
    cmap(row,x) = bitand(byte,0xc0)/64;
    x=x+1;
    cmap(row,x) = bitand(byte,0x30)/16;
    x=x+1;
    cmap(row,x) = bitand(byte,0x0c)/4;
    x=x+1;
```

```
    cmap(row,x) = bitand(byte,0x03);
  endfor
endfor
figure(16); clf; surf(cmap);
xlabel("x"); ylabel("y");

bitname = [fname(1:8) ".bitmap"];
fid = fopen(bitname,"w");
fwrite(fid,0); % Put in a fake load address of $8000
fwrite(fid,128);
for k=1:length(bitmap)
  fwrite(fid,bitmap(k));
endfor
fclose(fid);
```

A.3 Map creation

The script “mapmaker.m”, also in the source distribution, was used to create the map. The map is edited as a text array which is then converted to a C64 binary file at $c000 along with the key at $5000.

```
%
% Map maker -- construct a map for the arena.  Define the map
% as an array, then process the array into a C64 file that can be
% read in.
%
% SLJ 2/22/2024
%
clear all;

map = {
'1111111111111111111111111111111111111111111111111111111111111111'
'1   44RED.ROOM4l444444l4A444MAZE444ING444444445555             1'
'1   4                                       45  e 5555555555551'
'1   4                                       45 55 5          f1'
'1   4                  4444c44444d4444l4444 45 5  5 55555555 51'
'1   4                  c                  4 r5 55i5 55555555 51'
'1   4          w       4                  4               e5 51'
'1   4                  d                  5555555 5i55555555 51'
'1   4                  4                        5 5        5 51'
'1   3   v44c444d444c4444     5e5               555 5        5 51'
'1   3   3                  5 5555i55555555   e   5        5 51'
'1   3   3                  5             5   55555        5 51'
'1   3   3                  555555 555555 5                5 51'
'1   3   3                       5 5    5 5555i5555i5555i555 51'
'1   3   3                       i i    5                    51'
'1   3   3                       5 5    5 5555i5555 5555i555f51'
'1   3   3                       5 5    5 5       5 5         1'
'1   3   3                       5 5    5 5       5 5         1'
'1   3   3                       5 5    5 5       5 5         1'
'1   3   3                       5 5    5 5       5 5         1'
'1b333   33333333WOO.WOO.WOO333    5 5    5 5 5e5   5 5555555   1'
'1b                         3    i i    5 555 5   5       5   1'
'1b                         3    5 5    5     5   5555555 5   1'
'1b    b                    3    5 5    5555555         5e5   1'
```

```
'1b                              3    5 5            55555555555   1'
'1b333333333333333333333x   2    5 55555        5          5   1'
'1                          2   2    5     f        5 5555555 5   1'
'1                          2   2    55555 55555i5555 5     5 5   1'
'1                          G   2        5            5     5 5   1'
'1                          N   2        5555555 555555     5 555 1'
'1                          I   2              5 5          i   e 1'
'1                          O   2              5 5          5 555 1'
'1                          G   2              5 5          5 5   1'
'1                          .   2              5 5          5 5   1'
'1                          P   2              5 5     555555 5   1'
'1                          E   2              5 5     5      i   1'
'1                          E   2     55555    5 5     5  555 555 1'
'1                          K   2     e   5    5 5     5   f5   5 1'
'1                          2   2     555 5    5 5     555 5555 5 1'
'1                          2   2       5 555555 5       5  5e  5 1'
'122222222a22222222         2   2       5        5       55 55555 1'
'1a                2222a2222    2       5555555555        5    f  1'
'1a                             2         111JASON111     5555 5  1'
'1a       a                     2         1         R       55 5  1'
'1a                             2         k         O       5  5  1'
'1a                2222a22222222          1         B       5 55  1'
'1222222s    s222222111CORALp1NICHOLASq111          I       5  5  1'
'1      1    1      1                                 N       55 5  1'
'1      E    R      j                          1NEKKOD1       5  5  1'
'1      U    O      A                          16666LAST.ROOM65 55  1'
'1      L    O      R                          1g               5   1'
'1      B    M      U                          16   66666666665f5   1'
'1      1    1      A                          1h   hhhhhhhhhhh     1'
'1      i    i      L                          1h             h     1'
'1      1    1      1oNMUTUAnELLEONmYBBA11     1h     hhh     h     1'
'1      1    1                              1  1h     hhh     h     1'
'1eENTERy    zHEREe1111111111111111GREETZ!     1h     hhh     h     1'
'Y                                             Wh     hhh     u222221'
'Z                                             Yh                   21'
'A                                             Nhhhhhh2222222222    21'
'R                                             11?22222222222222    21'
'C                                                                  21'
'1                                                                  21'
'111111111i1111111111111111111111111111111111ZELUR.UPCREPUS2k2221'
};

%
% Table of maze values and textures
%
% Texture list:
%
% 1 white brick
% 2 blue stone
% 3 gray stone
% 4 red brick 1
% 5 red brick 2
% 6 black brick
%
% 16 blues80x
% 17 stooges
% 18 marilyn
% 19 face80x
```

```
% 20 angry smile
% 21 red smile
% 22 joker1
% 23 joker2
% 24 eagle80x
% 25 redhead
% 26 C= logo
% 27 girl1
% 28 left arrow
% 29 right arrow
% 30 painter
% 31 viola
% 32 ballet
% 33 hockey
% 34 dino
%
% 48-53 crazy1-crazy6
%
% Alphabet: 64-89
%  A - 64
%  B - 65
%  etc.
%
% 90 solid red (space)
%
% 127 animated texture

% symbol value X-texture Y-texture
mazeval = {
 '1', 1, 1, 1
 '2', 2, 2, 2
 '3', 3, 3, 3
 '4', 4, 4, 4
 '5', 5, 5, 5
 '6', 6, 6, 6
 '7', 7, 1,24
 'a',16,16,16
 'b',17,17,17
 'c',18,18,18
 'd',19,19,19
 'e',20,20,20
 'f',21,21,21
 'g',22,22,22
 'h',23,23,23
 'i',24,24,24
 'j',25,25,25
 'k',26,26,26
 'l',27,27,27
 'm',30,30,30
 'n',31,31,31
 'o',32,32,32
 'p',33,33,33
 'q',34,34,34
 'r',48, 4, 5
 's',49, 1, 2
 't',50, 1, 2
 'u',51,23, 2
 'v',52, 3, 4
```

```
'w',53,19,18
'x',54, 2, 3
'y',55,21,29
'z',56,21,28
'!',57, 1,29
'A',64,64,64
'B',65,65,65
'C',66,66,66
'D',67,67,67
'E',68,68,68
'F',69,69,69
'G',70,70,70
'H',71,71,71
'I',72,72,72
'J',73,73,73
'K',74,74,74
'L',75,75,75
'M',76,76,76
'N',77,77,77
'O',78,78,78
'P',79,79,79
'Q',80,80,80
'R',81,81,81
'S',82,82,82
'T',83,83,83
'U',84,84,84
'V',85,85,85
'W',86,86,86
'X',87,87,87
'Y',88,88,88
'Z',89,89,89
 '.',90,90,90
'?',127,48,48 % animated texture
};

%
% Now go through the map, starting from the bottom left, and create the
% data arrays (map and table).
%
mapc000 = zeros(64,64);
xmaptex = zeros(128);
ymaptex = zeros(128);

for y=0:63
  mapy = 64-y;
  for x=0:63
    mapx = x+1;
    sym = map{mapy}(mapx);
    sval = 0;
    for k=1:length(mazeval)
      if sym == mazeval{k,1}
        sval = mazeval{k,2};
        mapc000(y+1,x+1) = sval;
        xmaptex(sval+1) = mazeval{k,3};
        ymaptex(sval+1) = mazeval{k,4};
      endif
    endfor
  endfor
```

```
 endfor

%
% Output the files in C64 format
%

bitname = ["arena1.map"];
fid = fopen(bitname,"w");
fwrite(fid,0); % Put in a load address of $c000
fwrite(fid,192);
for y=1:64
  for x=1:64
    fwrite(fid,mapc000(y,x));
  endfor
endfor
fclose(fid);

fid = fopen("arena1.key","w");
fwrite(fid,0); % Put in a load address of $5000
fwrite(fid,80);
for x=1:128
  fwrite(fid,xmaptex(x));
endfor
for x=1:128
  fwrite(fid,ymaptex(x));
endfor
fclose(fid);
```