Game 3. Rogueima I

- SD-8516 Assembly Language
- Part II Writing Games in Assembly Language
- This is Part III.

Introduction

Let's make a new game. This game will be called Rogueima I: Moongate. Or something. It doesn't really have a name, but I'm calling it that name right now because it's pretty good. Yeah you know what, I'm calling it that! And you can't stop me. You will just have to come up with your own name later. Sorry, it's too late; I called it and I also call no takey-backeys infinity.

Step 1. Program Scope and Structure

A game like this, if you don't know, is a lot like Robots, except that it looks better (it's in a graphics mode) and it has better music and more in-depth gameplay.

a. Scope

It's typically played on a 320×200 screen. If you look closely from left to right you will see that on the left edge there is a blue border, then five “tiles”, the player, five more “tiles”, a center divider; then a status area with a final border on the right. From top to bottom (on the left side) you will see a border, again typically blue. Five tiles, the player, five more tiles, and the bottom border. On the right side on the top you will have a status window for up to 8 characters, a border, then a food/score/gold status, the on the bottom right is a chat window where you can type commands like NORTH, TALK, JOB, etc.

There is another kind of *similar game* with a graphics window on the upper left, a chat/message window in the upper right, and room for six characters on the bottom of the screen (increased to seven in parts 2 and 3).

These layouts are similar to Nethack or Rogue, which was a mostly full-screen character mode game, but which was single player. The three main differecnes between these games besides depth of play are that the first two we mentioned are graphical and have music.

The first kind of game, as I mentioned, has an 11×11 tile window. If you count carefully you will see(from left to right) a border, 11×2 squares used for tiles, another border, and space for sixteen chat characters. Or alternately in the top sections on the right, 15 characters and a border. Lets count:

```
width = 1 + (5*2) + (1*2) + (5*2) + 1 + 16
width = 1 + 10 + 2 + 10 + 17
width = 11 + 12 + 17
width = 40
```

And from top to bottom,

```
height = 1 + 10 + 2 + 10 + 1
height = 24
```

This exactly represents a 40×24 character screen. The reason for this is interesting. In the early days, some computers had a 40×24 screen and some had a 40×25.

The 40×24 camp	**The 40×25 camp**
Apple I	Commodore PET
Apple II	Commodore VIC-20 (extended mode)
Atari 8 bit	Commodore 64
Mattel Aquarius	CGA Graphics
ABC 80 (Sweden)	MSX (ex. MSX1)
Sharp MZ-721	CP/M Videotex terminals
TI-99/4A	Aster CT-80
ColecoVision	
Acorn Electron	
BBC Micro	BBC Micro in Mode 6

Later, 40×25 became more of a standard. After all, when you have a 320×200 screen, 25 8×8 characters will naturally fit on the screen. In 16 color mode this is 32k – a nice, even number with a little bit of room left over for a palette or some system variables. So it became a very common standard.

And, what about the extra line on a C64 or a PC? It was left blank. The reason is because the tilemap is 2×2 characters (16×16 tiles) so there's an odd-tile out between the 22 tiles and two border walls top to bottom. More aggressive ports would expand the the border area by a few pixels and off-set the text, adding an extra line of text in the bottom right window. We could go either way; but for the sake of sanity we will design the game around 80×24 and modify the borders later.

For the curious, the plan there will be to offset the entire game by 4 y-pixels and then re-fill the top and bottom border areas before drawing the bottom-right chat window. A simple fix.

The game itself has a rather large world with sub-maps you can explore, such as dungeons and towns, and sub-games such as “dungeon” or “combat” which differ from the main game. You can ride horses or ships. There is a passive animation to the creatures of the game and some environment objects (ex. magic barriers). The music quality is 'excellent' for the era.

Now with the scope in mind we are almost ready to outline the structure, with one caveat; A game like this requires significant graphics and sound assets. We will design these *after* the program structure, because the program scope and structure will help expand and bound what we will need in terms of game assets.

b. Structure

For “draw_map” we need:

- A main screen drawing function. The main display screen remains the same throughout the

entire game. It will consist of:
- Border drawing function.
- Player status drawing function.
- Party status (food, gold, score).
- Chat window.
- Game view (11×11 tile map of 16×16 tiles).

During play we need:

- Live animation of some tiles (ex. water, monsters, player)
- Background Music
- Player input
- Passage of time effects (moon phase/day-night, hunger, etc.)

In terms of world design we will need:

- World map with link/entry points to sub maps
- Sub maps like towns, dungeons, other “floors”

In terms of combat and interaction;

- Combat takes place in a “combat mode”.
- There is a talk or interact command which lets us interact with objects and players
- NPCs usually respond to one or two word conversations, and some words may unlock other words. A simple flag based system.

In terms of player inventory, etc.

- Each player can have a simple inventory. Unequipped items can go to a 'party inventory'. There are also variables for gold, food, and for score (or however you are marking progress).

Step 2. Designing the assets.

Musical assets can take quite some time to perfect. For now, we don't need to worry about it, but rather we will keep our eyes and ears open for interesting classical or 'folk' music we can steal and put ito our game. The more musical among us are suggested to come up with something original. What do you think Lord Kenneth Arnold did? He wrote the most amazing game music of all time, that's what he did. But we can settle for some good baroque pieces or a madrigal or two that fit the mood for now and work it out later.

Fine knacks for ladies would make a good stand-in for Rule Britannia. Handel's “Gavotte” could fit in somewhere. We can come up with a riff on “Forest” on our own, and maybe a few more pieces will fall into place by the time we're done? Okay, let's talk about art. The game was in 16 colors. The colors were similar across computers; they all had a decent blue and white for example, same with cyan, but some colors were significantly different, especially reds, orange, brown, and some shades of green. This gave each version a distinctive look. While many people have their own personal preference, general consensus gives the Ultima 4 crown to the Commodore 64, with the Atari 8 bit and Apple II original taking second place, while Ultima 5's top palette goes to the Amiga's rich colors, then the PC's EGA mode, then the Commodore 64 and finally the Apple II.

The EGA upset where it's behind the Commodore and Apple II for Ultima 4, but ahead for Ultima 5 is

likely because the game was designed for the Apple II originally, but Ultima 5 was designed around EGA tiles so that became the look of the game by then. But honestly you can't go wrong with a decent 80s palette. We will be designing our own custom palette based on these ideas.

Designing the Graphics

Do you know how to do graphics? Here's the idea:

	01	**02**	**04**	**08**	**10**	**20**	**40**	**80**	**01**	**02**	**04**	**08**	**10**	**20**	**40**	**80**	
04			1						1								01
08				1				1									00
FC			1	1	1	1	1	1	1								01
76		1	1		1	1	1		1	1							03
FF	1	1	1	1	1	1	1	1	1	1	1						07
FD	1		1	1	1	1	1	1	1		1						05
04	1		1						1		1						05
18				1	1		1	1									00
a																	j
b																	k
c																	l
d																	m
e																	n
f																	o
g																	p
h																	q
	01	**02**	**04**	**08**	**10**	**20**	**40**	**80**	**01**	**02**	**04**	**08**	**10**	**20**	**40**	**80**	

This grid represents four 8×8 cells. At the top and bottom hold hexidecimal cell values, which you sum up into the values on the left and right. These become four 8×8 tiles or one 16×16 tile as follows;

```
C1 = 04, 08, FC, 76, FF, FD, 04, 18
C2 = 01, 00, 01, 03, 07, 05, 05, 00
C3 =  a,  b,  c,  d,  e,  f,  g,  h
C4 =  j,  k,  l,  m,  n,  o,  p,  q
```

So for example in Stellar BASIC you could do

```
DEFCHAR 160, $04, $08, $FC, $76, $FF, $FD, $04, $18
DEFCHAR 161, $01, $00, $01, $03, $07, $05, $05, $00
```

Once you have these numbers, you can use DRAWCHAR to draw the data onto the screen. Of course, we're using Assembly language here, but we can call the same graphics routines BASIC uses. We'll just CALL them when we need to.

For now, here's a great idea! Get some colored markers that represent the palette you wish to use and draw out the art you want to use for your game. If you like, you can use a graph paper notebook to do this. Just tell your mom you need it for your math homework and she'll buy you a new one. A

graph paper notebook is an excellent investment for this stage of game develpment. Carry it with you, and practice drawing some bitmaps at lunchtime. You can draw four 8×8 bitmaps or one 16×16. The values on the left and right become the values you will use as data for your tile-set as demonstrated above.

The Player (Example)

	01	**02**	**04**	**08**	**10**	**20**	**40**	**80**	**01**	**02**	**04**	**08**	**10**	**20**	**40**	**80**	
a1																	a2
b1							6	6	6								b2
c1						6	6	6	6	6							c2
d1							6	6	6					7			d2
e1								6						7			e2
f1				F	F	8	8	F	b	b	6			7			f2
g1				F	8	F	F	F	b	b		6		7			g2
h1				F	F	8	F	F	b				7	7	7		h2
a3				F	F	F	8	F	b					7			a4
b3					F	8	F	4									b4
c3						F	4	4	4								c4
d3						4	4		4	4							d4
e3					4	4				4	4						e4
f3					6	6				6	6						f4
g3				6	6	6				6	6	6					g4
h3																	h4
	01	**02**	**04**	**08**	**10**	**20**	**40**	**80**	**01**	**02**	**04**	**08**	**10**	**20**	**40**	**80**	

I've taken the liberty of using colors here. For 1-color characters or sprites, you calculate pixels as shown above. But for multi-color images you need to understand the screen format.

First let's talk colors. On the EGA 16 color palette:

- 2 = green
- 4 = red
- 6 = yellow/brown
- 7 = light gray
- 8 = dark gray
- 15 (0xF) white

However, the 320×200 screen is a 4bpp format; each byte represents two pixels:

ONE BYTE	
Low Byte	High Byte
4 bits	4 bits
Pixel 1	Pixel 2
Color 0-15	Color 0-15

Therefore, calculating what bytes to use for this image is a bit tricker.

The Player (Example)

	a1	b1	a2	b2	a3	b3	a4	b4	a5	b5	a6	b6	a7	b7	a8	b8
a1																
b1							6	6	6							
c1						6	6	6	6	6						
d1							6	6	6					7		
e1								6						7		
f1				F	F	8	8	F	b	b	6			7		
g1				F	8	F	F	F	b	b		6		7		
h1				F	F	8	F	F	b				7	7	7	
a2				F	F	F	8	F	b					7		
b2					F	8	F	4								
c2						F	4	4	4							
d2						4	4		4	4						
e2					4	4				4	4					
f2					6	6				6	6					
g2				6	6	6				6	6	6				
h2																
	a1	b1	a2	b2	a3	b3	a4	b4	a5	b5	a6	b6	a7	b7	a8	b8

here, you need to combine 2 pixels into a low/high pair. So on row b1, the 5th and 6th columns will combine as 0 in the low byte and 6 in the high byte. This becomes:

```
  0110 0000 = #96 or 0x60
```

you know it's 0x60 because the two high bits are 0x40 and 0x20; $40 + $20 = $60.

However way you intend to encode the graphic information is up to you. In fact, there's one more way; “sprite sheets”, or, tile-sheets. To make a sprite-sheet you can just use a program like GIMP to make an image and then draw the individual tiles that way. You access them by loading the image and copying the data from the image to the screen when you want to use it.

But, there's one more way.

3. Super Easy Tiles

Actually I'm very lazy and I don't like to carry a notebook around with me or use GIMP/images directly. I like to keep everything in the source code, you see – makes things more portable. So what if you could to it this way:

```
.tile_player
    .bytes #16, #16
    .bytes "                "
    .bytes "      666       "
    .bytes "     66666      "
    .bytes "      666    7  "
```

```
    .bytes "       6     7  "
    .bytes "   FF88Fbb6  7  "
    .bytes "   F8FFFbb 6 7  "
    .bytes "   FF8FFb   777 "
    .bytes "   FFF8Fb    7  "
    .bytes "    F8F4         "
    .bytes "     F444        "
    .bytes "     44 44       "
    .bytes "    44   44      "
    .bytes "    66   66      "
    .bytes "   666   666     "
    .bytes "                 "
```

There, now doesn't that look a whole lot better? It actually looks better, and is more practically useful. Let's get the computer to do all the calculation work for us, too:

```
;;;;;;;;;;;;;;;;;;;;;;
;; draw_tile
;; IN:    ELM -- Address of tile data
;;          X -- X address to draw tile
;;          Y -- Y address to draw tile
;;;;;;;;;;;;;;;;;;;;;;
read_tile:
    PUSH I     ; Save these as we will use them for loop counters.
    PUSH J
    PUSH ELM   ; Save ELM, X and Y since we will modify them.
    PUSH Y
    PUSH X
    LDI #16     ; initialize width and height loop counters.
    LDJ #16
    ; So now: I and J are our loop counters.

dt_y_loop:
    MOV I, #16      ; Reset the width counter
    POP X           ; Restore X from stack
    PUSH X          ; Save for next Y-loop

dt_x_loop:
    LDCL [ELM, +]   ; Read a charater from the tile data.
    CMP CL, #' '    ; is it a space?
    JNZ @dt_draw    ; no, continue
    LDCL #0         ; it's a space, convert CL to 0.

dt_draw:
    SUB CL, #'0'    ; convert '0' to #0 (number), '1' to #1, etc.
    CMP CL, #16     ; if CL >= 16 then set carry.
    JC @dt_x_end    ; Don't draw anything, jump to end (we can use this for
transparency later).
```

```
    LDAH #01        ; Draw Mode3 pixel (at 4bpp) (draws C=color at X, Y)
    INT 0x18
dt_x_end:
    INC X           ; Increase X count (for next pixel)
    DEC I           ; Decrease width count
    JNZ @dt_x_loop  ; repeat inner x-drawing loop
    DEC J           ; decrease height counter after width loop
    JNZ @dt_y_loop  ; Repeat outer y loop


    POP X
    POP Y
    POP ELM
    POP J
    POP I
    RET             ; Return to caller
```

Now, I haven't tested this code yet, but I'm pretty sure it works. Just wing it. I mean, all it does is read in the data you typed in as characters and draws it to the screen. But, things are never as they seem in Assembly Language...

If it takes 10 instructions to process the character and the plot function is another 40 instructions – as the crow flies – that's 50 instructions to plot a pixel. If you had to plot every pixel on the screen then, that becomes 320×200 pixels! That means at 1.28 MIPS it might take 3 seconds to draw the screen. This is a problem! What can we do?

The standard tricks are to try blitting (which is a memory copy) and to use a dirty map (i.e. only redraw tiles that have changed). The loop for blitting means we have at least 8 times fewer plots to make, and the dirty map ensures that most of the time we only need to redraw a small number of tiles on the screen – even if the user is mashing move keys, it's usual to see less than half the map area needing a redraw. So even if we had to redraw 20 or 30 tiles every frame, this amounts to just 480 MEMCOPY instructions. Even at 50 supporting instructions per MEMCOPY (a worst case) this won't take up more than 3% of the CPU. A big win.

Note: Using MEMCOPY we must pixel-align all tiles to pixels divisible by 2. This is because when we copy one byte, we are copying two pixels. Therefore we can only blit to an X divisible by 2. That's usually no problem for a tile-based game. If it draws at 0,0 it will be blittale at 16,16, 32,32, etc.

Okay, lets set this up. To initialize, we will need to pre-draw all the tiles into a tilesheet image. Each tile will be referred to by number, which means it will be easy to calculate it's location from the base of the data (as each tile is 16×16, or 128 bytes (at 2 pixels per byte).

here's the function to save the data after the framebuffer:

```
;;;;;;;;;;;;;;;;;;;;;
;; copy_to_index
;; Copies the tile data from 0,0 to 15x15 into the index location (I)
;; on the tilesheet. We do not need to clamp X and Y, if the tile is
;; 8x8 it will be copied inside the 16x16 sqaure.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;

.equ M3_FRAMEBUFFER $020000   ; Mode 3 framebuffer
.equ SHAPES_DATA    $027D00   ; Tile Data; space for 256 tiles x 128 bytes
each

copy16_to_index:
    PUSH ELM
    PUSH FLD
    PUSH C
    PUSH I
    LDELM @SHAPES_DATA
    LDFLD @SHAPES_DATA
    MUL I, #128           ; get offset
    ADD FLD, I            ; get memory location to store tile data
    LDC #16               ; repeat 16 times (one for each row of the tile)
cti_loop:
    MEMCOPY ELM, FLD, #8  ; copy first 16 pixels (8 bytes; 4bpp = 16 pixels)
    ADD ELM, #160         ; span to next screen line
    ADD FLD, #8           ; store tile data row after row.
    DEC C
    JNZ @cti_loop         ; if C iz not yet 0, repeat the loop.

    POP I
    POP C
    POP FLD
    POP ELM
    RET
```

And now, we need the tile drawing function:

```
; draw_tile16 - Draw a 16x16 tile to Mode 3 framebuffer
; Input: I = tile index, X, Y = location to draw (X must be even)
draw_tile16:
    PUSH ELM
    PUSH FLD
    PUSH C
    MOV A, X  ; save copy
    MOV B, Y

    ; Calculate source: SHAPES_DATA + I * 128
    LDFLD @SHAPES_DATA
    MUL I, #128
    ADD FLD, I

    ; Calculate dest: M3_FRAMEBUFFER + (Y * 160) + (X / 2)
    LDELM @M3_FRAMEBUFFER
    MUL B, #160
    ADD ELM, B
```

```
    SHR A, #1             ; X / 2 for byte offset
    ADD ELM, A

    LDC #16  ; copy 16 rows
dt_loop:
    MEMCOPY FLD, ELM, #8  ; 16 pixels = 8 bytes
    ADD FLD, #8            ; next tile row
    ADD ELM, #160          ; next screen row
    DEC C
    JNZ @dt_loop

    POP C
    POP FLD
    POP ELM
    RET
```

The draw for a 16×16 sprite is therefore 18 instructions plus 16×5 instructions; that's 98, or 100 to pick a nice round number. For an 8×8 it would only be 58 (or, 60). If half the screen was 16×16 and the other half 8×8 then you could expect 500×100 plus 500×60 operations total to draw the screen. That's 80,000 instructions – already more than enough to draw 10 times a second. Add in a dirty map and we're golden.

Map Projection and Dirty Tiles

We don't want to draw a tile that we have already drawn. So, how then do we draw a tile? Well we need to know what tiles are supposed to be there in the first place. We will do this by keeping track of a tile draw area:

```
tile_window:
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,1,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0

tile_window_drawn:
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,1,0,0,0,0,0
```

```
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0
```

You may wonder what the 1 is in the middle. That's where the player will be. I just made it a 1 so you would see where the player will be drawn. Technically the player will always be drawn in the middle of the map so we never need to draw it. Anyways, what happens is:

- 1. We want to draw a tile. So we set the byte in the tile_area to the tile index of the tile we want to draw (0 to 128).
- 2. Every render cycle (at least 60 times per second) the rendering function checks each byte in `tile_window` against `tile_window_drawn`. If the bytes are different, it draws the tile and sets the relevant byte in `tile_window_drawn` to the ID of the tile it drew.

Something like;

```
; render_dirty_tiles - Draw only changed tiles
render_dirty_tiles:
    PUSH ELM
    PUSH FLD
    PUSH A
    PUSH B
    PUSH C
    PUSH I

    LDELM @tile_window
    LDFLD @tile_window_drawn
    LDI #0                  ; linear index 0..128

rdt_loop:
    LDAL [ELM]              ; tile_window[i]
    LDBL [FLD]              ; tile_window_drawn[i]
    CMP AL, BL
    JZ @rdt_skip            ; same — skip

    STAL [FLD]              ; Store new value into drawn map

    LDX #8     ; tile area to draw in starts at 8,8
    LDY #8
    LDC #0B0B  ; draw an 11x11 set of 16x16 tiles.

    ; This area shoud now loop to draw the 11x11 (121 tiles) in the tile
area. Starting at X=0 and Y=0 we run two loops.
FIXME

rdt_skip:
    ; we need to keep track of the X and Y position even if we don't draw
the tile
```

```
    ; /add code here//

    INC ELM
    INC FLD
    INC I
    CMP I, #127
    JNC @rdt_loop   ; If I >= 127 then set carry. No carry = loop.

    POP I
    POP C
    POP B
    POP A
    POP FLD
    POP ELM
    RET
```

That's a very simple routine!

more coming soon