004 – Display a sprite

Prerequisite: 001 – Display a background

This tutorial describes:

- How to create a sprite and his logical code
- How to associate images with an object
- How to set the object position on screen and off screen
- Types of image rendering
- The display priority between sprites

Introduction

The game engine provides multiple ways to display a sprite on TO8.

On many 8/16 bits computers without VDP (Video Display Processor), the fastest way to display an image is to use a “compilated sprite”. It’s an assembly code that will directly writes an image to a memory location, instead of copying data from one memory location to another.

It will be 6x time faster than RLE rendering but does not allow to display an image partially. If the image goes outside the memory location, it will be fully hided to avoid memory corruption.

While you can write compilated sprite by hand, the game engine Builder is able to automatically convert your images into compilated sprites.

The engine only operates graphics in 160x200px 16 colors mode, and provides:

- **display priority** (what sprite should be on top of the others, with 8 layers)
- **screen coordinates** (ability to place a sprite at a screen position)
- **playground coordinates** (ability to place a sprite in a playground, the sprite position on screen will be calculated in conjunction with the camera location).
- **rendering types**:
    - (compilated sprite) write an image
    - (compilated sprite) save background and write an image
    - (RLE) write an image
- **X and/or Y mirroring** (will need duplicate images produced by the Builder)
- **2px or 1px draw precision on x axis** (1px precision will need two images)
- **manual hide** (temporary hide of a sprite)
- **automatic hide** (when a sprite is outside or cross the screen top or bottom)
- **parametric hide** (choose if a sprite should be hidden or not when he crosses the screen left or right limit)
- **automatic erase/draw** (the engine will not redraw a sprite if not required, it’s based on sprite position, collision, image, priority)
- **double buffering** (handle all those functions in a double buffering mode)
- **sprite animations** (see related tutorial)
- **tile map** (see related tutorial)
- **tile map scroll** (see related tutorial)

Display an overlay sprite

In this first part we are going to display a background and an overlay sprite.

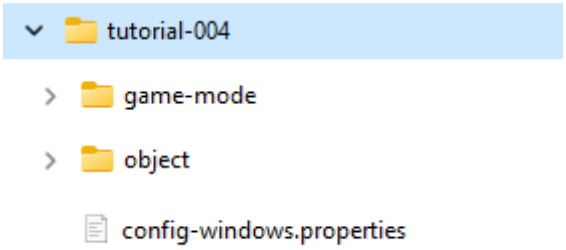
File structure

For this tutorial, we need to create a game mode and an object.

The game-mode will hold the main loop of the program.

The object will hold the sprite logic.

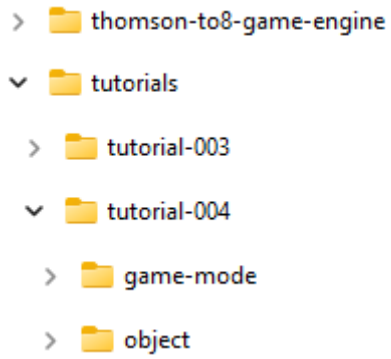
Create a new folder structure like this one :



The configuration of the “config-windows.properties” file is covered by tutorial 000.

You can place your directory inside the game-project or outside.

As an example, for this directory structure :



You should configure you properties file like this one :

You will notice that relative path should point to the game engine directory.

If the project is inside the game-project directory the relative path will be prefixed by :

```
../..
```

If you decided to go outside with a “tutorial” parent directory, it will be :

```
../../thomson-to8-game-engine
```

Using an external directory may be useful if you want to use separate repositories (one for the engine, and one or several for your games).

In this tutorial, the game-mode will be named infiniteloop, so you should have this configuration in your .properties file:

```
gameModeBoot=infiniteloop
gameMode.infiniteloop=./game-mode/infinite-loop/infinite-loop.properties
```

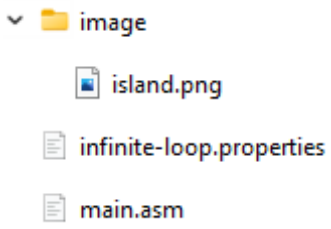
Game Mode definition

Add a background

Use the tutorial 001 to create a game mode with this background image :


island.png

Now you should have this structure :



Add an object

To tell what object will be used by this game mode, add this to your infinite-loop.properties :

```
# Objects
object.foreground=./object/foreground/foreground.properties
```

When the game-mode will be loaded in RAM, it will also load this object.

Note :

There is a way to define common objects for different game modes, it will be covered in a dedicated tutorial.

If you don't use common objects, when two game modes are using the same object, the object binary is duplicated in the FD/ROM, because :

- the object code is assembled for a specific RAM location
- the data is loaded into RAM by pages, not by objects

The common objects also remains in RAM between a game mode change.

Here is the full properties for this game-mode:

infinite-loop.properties

Sprite display main loop

To display sprites, the main loop of the game mode should have this structure :

```
MainLoop
        jsr   WaitVBL
        jsr   UpdatePalette
        jsr   RunObjects
        jsr   CheckSpritesRefresh
        jsr   EraseSprites
        jsr   UnsetDisplayPriority
        jsr   DrawSprites
        bra   MainLoop
```

WaitVBL: swap video pages for double buffering

UpdatePalette : change palette colors when screen spot is not in the visible area

RunObjects : call objects routine

CheckSpritesRefresh : check all sprites and flag the ones that needs to be refreshed on screen

EraseSprites : clear the sprites on screen when a sprite is deleted or need to be refreshed

UnsetDisplayPriority : update the display priority data in case of priority change or sprite removal

DrawSprites : write sprites on screen

Here is the full main.asm for this game-mode:

main.asm

In this file you will see new includes :

```
        INCLUDE "./game-mode/infinite-loop/main.equ"
        INCLUDE "./Engine/Constants.asm"
        INCLUDE "./game-mode/infinite-loop/ram-object.asm"
```

Engine data structures

main.equ

This is the file that hold equates definitions.

It is mandatory to size the engine data structures for your game mode, here as we only have one sprite, we will set the nb_graphical_objects to 1.

```
nb_graphical_objects                equ 1 ; only count objects that will be
rendered on screen (max 64 total)
```

main.equ

ram_object.asm

This is the file that defines RAM data space for object variables.

```
Object_RAM
                            fcb   ObjID_foreground

                            fill  0,object_size-1
Object_RAM_End
```

There is two way of instantiate an object :

- hardcoded
- at runtime

To instantiate an object, simply set the first byte of the object data to the object id.

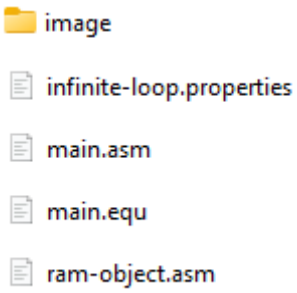
The object id “equate” is simply a string concatenation of : “ObjID_” and the object name (as defined in the game mode properties, here: `object.sonic`)

In this example we are making an hardcoded instantiation.

Dynamic instantiation will be covered in another tutorial, but keep in mind that the engine provides routines to **find** free object data structure, and also routines to **clear** an object data structure.

ram-object.asm

You should now have this file structure in the infinite-loop directory:



Engine/Constants.asm

To help you understand what are the data associated to an object, I just want you to review some content of the engine Constants.asm file.

This is the file that hold equates definitions for all the engine data structures.

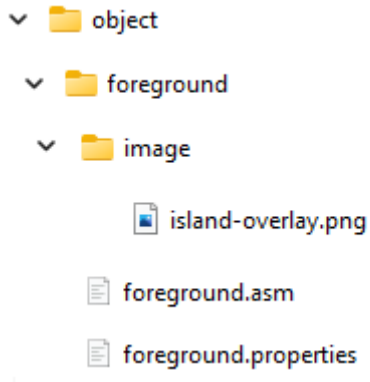
Here is the list of equates (available to the developer) for the object data structure :

```
* =============================================================================
* Object Constants
* =============================================================================

object_size                 equ 114 ; the size of an object - DEPENDENCY ClearObj routine
next_object                 equ object_size

id                          equ 0           ; reference to object model id (ObjID_) (0: free slot)
subtype                     equ 1           ; reference to object subtype (Sub_)
render_flags                equ 2

* --- render_flags bitfield variables ---
render_xmirror_mask         equ $01 ; (bit 0) DEPENDENCY should be bit 0 - tell display engine to mirror sprite on horizontal axis
render_ymirror_mask         equ $02 ; (bit 1) DEPENDENCY should be bit 1 - tell display engine to mirror sprite on vertical axis
render_overlay_mask         equ $04 ; (bit 2) DEPENDENCY should be bit 2 - compilated sprite with no background save
render_motionless_mask      equ $08 ; (bit 3) tell display engine to compute sub image and position check only once until the flag is removed
render_playfieldcoord_mask  equ $10 ; (bit 4) tell display engine to use playfield (1) or screen (0) coordinates
render_hide_mask            equ $20 ; (bit 5) tell display engine to hide sprite (keep priority and mapping_frame)
render_todelete_mask        equ $40 ; (bit 6) tell display engine to delete sprite and clear OST for this object
render_xloop_mask           equ $80 ; (bit 7) (screen coordinate) tell display engine to hide sprite when x is out of screen (0) or to display (1)

priority                    equ 3           ; display priority (0: nothing to display, 1:front, ..., 8:back)
anim                        equ 4  ; and 5  ; reference to current animation (Ani_)
prev_anim                   equ 6  ; and 7  ; reference to previous animation (Ani_)
anim_frame                  equ 8           ; index of current frame in animation
anim_frame_duration         equ 9           ; number of frames for each image in animation, range: 00-7F (0-127), 0 means display only during one frame
anim_link                   equ 10          ; allow animation swap without reseting anim_frame and duration
image_set                   equ 11 ; and 12 ; reference to current image (Img_) (0000 if no image)
x_pos                       equ 13 ; and 14 ; x playfield coordinate
x_sub                       equ 15          ; x subpixel (1/256 of a pixel), must follow x_pos in data structure
y_pos                       equ 16 ; and 17 ; y playfield coordinate
y_sub                       equ 18          ; y subpixel (1/256 of a pixel), must follow y_pos in data structure
xy_pixel                    equ 19          ; x and y screen coordinate
x_pixel                     equ 19          ; x screen coordinate
y_pixel                     equ 20          ; y screen coordinate, must follow x_pixel
x_vel                       equ 21 ; and 22 ; horizontal velocity
y_vel                       equ 23 ; and 24 ; vertical velocity
routine                     equ 25          ; index of current object routine
routine_secondary           equ 26          ; index of current secondary routine
routine_tertiary            equ 27          ; index of current tertiary routine
routine_quaternary          equ 28          ; index of current quaternary routine

ext_variables               equ 88 ; to 113 ; reserved space for additionnal variables (25 bytes)
```

There are also “rsv_” equates (not mentioned here) that are reserved to internal engine routines.

Implement an object

Create this file and directory structure :



Files for reference :

(for png constraints refers to tutorial 001)

Declare sprite images

An object ".properties" file defines :

- assembly code for the object
- images
- animations
- sounds
- …

Here we want to associate a simple image to the object.

```
code=./object/foreground/foreground.asm

# Sprites
sprite.Img_foreground=./object/foreground/image/island-overlay.png;ND0
```

After the image file name, you have to specify an option. It will determine the image types and variations that will be available at runtime (multiple values possible, but only one is mandatory).

Here is a complete list of values (must be comma separated):

- NB0 : no flip, background backup / draw / erase compilated sprite, no x offset
- ND0 : no flip, draw compilated sprite, no x offset
- NB1 : no flip, background backup / draw / erase compilated sprite, 1px x offset
- ND1 : no flip, draw compilated sprite, 1px x offset
- XB0 : x flip, background backup / draw / erase compilated sprite, no x offset
- XD0 : x flip, draw compilated sprite, no x offset
- XB1 : x flip, background backup / draw / erase compilated sprite, 1px x offset
- XD1 : x flip, draw compilated sprite, 1px x offset
- YB0 : y flip, background backup / draw / erase compilated sprite, no x offset
- YD0 : y flip, draw compilated sprite, no x offset
- YB1 : y flip, background backup / draw / erase compilated sprite, 1px x offset
- YD1 : y flip, draw compilated sprite, 1px x offset

- XYB0 : xy flip, background backup / draw / erase compilated sprite, no x offset
- XYD0 : xy flip, draw compilated sprite, no x offset
- XYB1 : xy flip, background backup / draw / erase compilated sprite, 1px x offset
- XYD1 : xy flip, draw compilated sprite, 1px x offset Declare a graphical object (list where it is used)

In this tutorial, we will be using the ND0, that means a simple compilated sprite with no background backup and a positioning precision of 2px in the x coordinates.

Note: If you specify only one image (either with no shift or with 1px shift), you have to place this image on odd or even x coordinate. One will display the sprite the other no. It depends on the sprite width and the shift.

Here are the combinations :

- no shift and even witdh will be displayed on even x coordinate.

- no shift and odd witdh will be displayed on odd x coordinate.

- 1px shift and even witdh will be displayed on odd x coordinate.

- 1px shift and odd witdh will be displayed on even x coordinate.

This may be the number one reason why your sprite won't show on screen.

Object logical code

The object code defines what assembly code will be executed for the object each time the main game engine loop will be executed.

This code is called by `RunObjects` with register U loaded with the address of the object structure data.

```
; ------------------------------------------------------------------------------
; Object - fgrnd
;
; input REG : [u] pointer to Object Status Table (OST)
; ---------
;
; ------------------------------------------------------------------------------

        INCLUDE "./Engine/Macros.asm"

fgrnd
        lda   routine,u
        asla
        ldx   #fgrnd_Routines
        jmp   [a,x]

fgrnd_Routines
        fdb   fgrnd_Init
        fdb   fgrnd_Main
```

```
fgrnd_Init
        ldb   #1
        stb   priority,u

        ldd   #$807F
        std   xy_pixel,u

        lda   render_flags,u
        ora   #render_overlay_mask
        sta   render_flags,u

        ldd   #Img_foreground
        std   image_set,u

        inc   routine,u

fgrnd_Main
        jmp   DisplaySprite
```

This code is based on a routine lookup table. The routine value can index a maximum of 128 routines.

As the object data is initialized to 0, the first executed routine is always the first in the list, Here fgrnd_Init.

```
fgrnd
        lda   routine,u
        asla
        ldx   #fgrnd_Routines
        jmp   [a,x]

fgrnd_Routines
        fdb   fgrnd_Init
        fdb   fgrnd_Main
```

The engine handle 4 levels of routines (routine, routine_secondary, routine_tertiary, routine_quaternary), you can add levels if needed thru custom data.

The first execution is used to initialize the object data, it also change the routine to execute the main routine one next step :

```
        inc   routine,u
```

There is only one image rendered for each object, you have to create another object to have one more image.

You have to call DisplaySprite each time you want to show the sprite, otherwise it will be hidden.

Here we are using a “jmp DisplaySprite” to avoid making a jsr and next a rts :

```
        jmp   DisplaySprite
```

You should have noticed that DisplaySprite is also called in the init routine since there is no rts call between sonic_init and sonic_main.

Now that we have the code structure, let see how to assign an active image to the object.

```
        ldd   #Img_foreground

        std   image_set,u
```

Note : The label refers to the one used in the .properties file

This image should have a position, we will be using screen coordinates :

```
        ldd   #$807F

        std   xy_pixel,u
```

Note : We can also set the x and y position independently with (x_pixel or y_pixel)

Ok, I'm sure you want a better way of doing that … because $807F is not quite understandable.

Simply use one of the engine macro and some equates :

```
        _ldd  screen_left+80,screen_top+99
```

The sprite will be centered at the 80,99 px screen position, that's better now …

For an easy positioning the png for this sprite is the size of the screen, so it will align perfectly with the background.

You will have to include the macros at the beginning of your asm file :

```
        INCLUDE "./Engine/Macros.asm"
```

Note : All macro names begin with a "_"

This expression will also use the following engine constants (Engine/Constants.asm), you don't need to include this file in the object code :

```
screen_width                equ 160    ; in pixel
screen_top                  equ 28     ; in pixel
screen_bottom               equ 28+199 ; in pixel
screen_left                 equ 48     ; in pixel
screen_right                equ 48+159 ; in pixel
```

The engine provide 8 level of sprite priority (or layer). You must set this parameter, so let's set it to 1 for this tutorial, values are: 0 (nothing to display), 1 (front), ..., 8(back)

```
        ldb   #1
        stb   priority,u
```

The last parameter is only required when you want to display a sprite with no background backup.

```
        lda   render_flags,u
```

```
        ora    #render_overlay_mask
        sta    render_flags,u
```

It sets the overlay bit to 1, and tells the engine that we want to use the “D” image (refers to images types ND0, ND1, XD0, …). If the value bit were 0 (default) it would have mean the “B” image (with background backup).

You are ready to build and run your program.

Details in 000 tutorial

The expected result :

Conclusion:

This first sprite demo show a simple way to “print” a sprite as an overlay. Usually it is used for non-moving objects, or foreground. You can also use it to simulate a continuous pencil draw for example.

If you run a debugger, you will notice that the sprite is only rendered twice, one for each video buffer and so merged with the background. If another sprite in a lower priority layer cross this overlay sprite, the overlay will be automatically refreshed.

Display a sprite without affecting the background

In this second part we are going to display a moving sprite .

Add a new object

In the game mode **infinite-loop.properties**, add a new object :

```
object.sonic=./object/sonic/sonic.properties
```

In the **ram-object.asm**, add a new object instance for sonic:

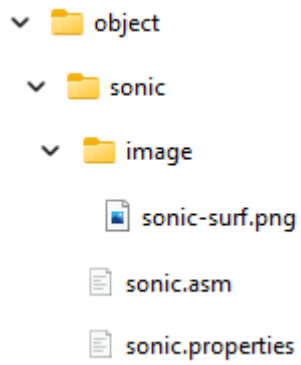
```
Object_RAM
                                fcb   ObjID_foreground
                                fill  0,object_size-1

                                fcb   ObjID_sonic
                                fill  0,object_size-1
Object_RAM_End
```

In the **main.equ**, add one more object

```
nb_graphical_objects                equ 2 ; only count objects that will be
rendered on screen (max 64 total)
```

Declare sprite images

Create this file and directory structure :



Files for reference :

sonic-surf.png

Declare sprite images

Here we want to associate an image to the object.

```
code=./object/sonic/sonic.asm
```

```
# Sprites
sprite.Img_sonic_surf=./object/sonic/image/sonic-surf.png;NB0
```

After the image file name, you will specify NB0. (no flip, background backup / draw / erase compilated sprite, no x offset)

Object logical code

```
; ---------------------------------------------------------------------------
; Object - sonic
;
; input REG : [u] pointer to Object Status Table (OST)
; ---------
;
; ---------------------------------------------------------------------------

        INCLUDE "./Engine/Macros.asm"

sonic
        lda   routine,u
        asla
        ldx   #sonic_Routines
        jmp   [a,x]

sonic_Routines
        fdb   sonic_Init
        fdb   sonic_Main

sonic_Init
        ldb   #2
        stb   priority,u

        _ldd  screen_left+49,screen_top+158
        std   xy_pixel,u

        ldd   #Img_sonic_surf
        std   image_set,u

        inc   routine,u

sonic_Main
        lda   x_pixel,u
        adda  #2
        sta   x_pixel,u
        jmp   DisplaySprite
```

You will have to include the macros at the beginning of your asm file :

We need to set sonic on a lower layer to be under the overlay sprite (here priority 2).

If you wanted to use layer 1 for sonic, you need to declare sonic before the foreground in Object_RAM data structure. Sprites on the same layer are stacking from back to front.

```
        ldb   #2
        stb   priority,u
```

No need to set the render_flags, as this type of sprite is the default one.

You are ready to build and run your program.

Details in 000 tutorial

The expected result :

Conclusion:

You now have a smooth 50fps animation with two sprites and an overlay.

The overlay is only refreshed when sonic surfs behind.

TODO :

Use the code of the overlay sprite and convert to a sprite

Move the object

Hide/show the object

Delete the object (show that the object is removed from screen)

Explain double buffering and background save

Change a sprite image from overlay to normal

Declare an image as overlay and normal, and x y and 1px

Switch between overlay and normal mode (like a stamp mode)

Switch to mirror

Switch to 2px or 1px mode

Display an overlay sprite (RLE)

Declare an image as RLE overlay and show size gain (bytes)

Show cycle count between compilated sprite and RLE with dcmoto debug mode

(Conclude that it is best to use when no speed is required)

Display sprites on different layers (priority)

Create many sprites on different priority layers

Show dynamic layer change (press of a button)

Explain automatic redraw of moving / priority changing sprites

Use playground coordinates

Presents the example of the sword in Zelda

Show auto hide for top and bottom

Show wrap and hide for x axis

Use joypad control