**Graphics Details**

*Note: assumes reader knows, or doesn't care, exactly how pixels are draw on the Apple Graphics screen. This document focuses on the systems used by the graphics engine to facilitate the drawing.*

**Overview**

At its core, the graphics engine's input numbers representing which shape to draw, and its output is a drawn shape on the screen.

This section discussed the numbers which represent the shapes (Tile Data), and how the graphics engine uses those numbers to locate that data for the associated shape.

**Tile Data**

**Summary**

As mentioned, the graphics engine draws shapes at various tile locations on the screen based on the Tile\_ID (/aka Tile\_Type) associated with that position on the map.

In this section we'll walk through how tile data gets from the floppy disk (or disk image) into the final state from which the graphics engine uses it to draw shapes. We'll do this in reverse chronology, starting with the final state.

**Screen Tile Locations & Screen Arrays**

The game logically divides the graphics screen into section we call tiles. We refer to this division as the screen tile grid. Each tile on the screen tile grid is associated with a specific location referred to as screen tile location.

There are several screen arrays which contain tile data in the screen tile grid which is used to draw the graphics screen. Their assembler labels all start with SCREEN. For example, SCREEN.TILE.DATA, SCREEN.MO\_TRANSPORT.DATA, SCREEN.MO\_MOB.DATA.

SCREEN.TILE.DATA contains the terrain Tile\_ID for each map location on the current view screen. Which map locations are on the current view screen is calculated based on the players map position (RMAP) and applying various constants as offsets to calculate the RMAP associated with each tile on the screen.

The Tile\_IDs are 8-bit hexadecimal numbers ($0-$FF). Each terrain type, or any shape appearing on the screen is assigned a Tile\_ID. For example, $00 = Mountains. $34 = Grass. $97 = Orc. $7B = Caravel (boat).

SCREEN.TILE.DATA contains a series of the 8-bit tile IDs, 187 to be exact, the number of tiles on the view screen. For example, the first four values stored in SCREEN.TILE.DATA could be 00.00.34.34. This would represent two mountain tiles followed by two grass tiles.

SCREEN.MO\_TRANSPORT.DATA and SCREEN.MO\_MOB.DATA have a similar format but they only contain the Tile ID for transport map objects (example $7B: Caravel) and mob map objects (example $97:ORC) respectively their format is a little more complicated since each object is stored as a four byte record, containing the X,Y coordinates of the object (relative to the player) and a flag byte, in addition to the Tile\_ID.

In general, for each location on the screen, the graphics engine draws the terrain tiles unless there is a map object present in that location, in which case it draws the Tile\_ID associated with

the map object. If there are multiple map objects in the same location, then the object which is stored last in the array will be the object drawn on screen.

See chart <INSERT> for the mapping of screen tile locations to the screen arrays, this is effectively an illustration of the screen tile grid thought the byte/line dimensions of each tile are now shown. The screen tile location is the index for each of the screen arrays.

**Key Variables**

The player's position on the regional map (RZONE.ARRAY) is tracked in the variable RMAP (16-bit). See chart <insert> for a map of RMAP values to RZONE.ARRAY elements.

The player's position on the world map is tracked in GMAP.X and GMAP.Y, but currently there variables are not used for anything. Eventually they will probably be used to validate the players position for (E)nterable locations.

SMAP (16-bit) contains the RMAP value of the tile in the upper left corner of the view screen. SMAP is calculated by subtracting an offset (stored in a constant) from RMAP.

SMAP is used by the graphics engine as the starting point when drawing tiles on the screen. SMAP.CURRENT is used to contain a working copy of SMAP which is incremented to advance draw routines to the next tile.

**Ok, so how do the SCREEN ARRAYS get their data?**

SCREEN.MO\_TRANSPORT.DATA and SCREEN.MO\_MOB.DATA are discussed in another section <INSERT>

As mentioned, SCREEN.TILE.DATA is associated with locations on the map. Both a world map and a regional map is stored.

In summary, SCREEN.TILE.DATA gets its tile data from the regional map (RZONE.ARRAY). RZONE.ARRAY is stored in main memory. It get's its tile data from the world map zones which are stored in auxiliary memory after they are loaded from disk.

A more detailed explanation is below, but first we need to talk about Tile Data Compression.

**Tile Data Compression**

An important aspect that comes into play at this level of the code is tile data compression. Tile data is compressed by representing repeating adjacent Tile\_IDs using two bytes. For example, consider map position that results in the view screen containing 9 mountains in a row. Uncompressed, this tile data would be represented as:

00.00.00.00.00.00.00.00.00

Using compression, these tiles would be represented by the following, with the 1st byte referring to the quantity of tiles and the second byte referring to the Tile\_ID:

09.00

Compressing the tile data saves a lot of disk space and memory, especially considering that there many repeating tiles on the map such as long stretches of deep water.

**How Do the Screen Arrays Get Their Data? (take 2)**

SCREEN.TILE.DATA and RZONE.ARRAY stores tile data in an uncompressed format. RZONE.ARRAY. The world zones in auxiliary memory and disk contain compressed tile data.

Thus, the sequence goes as follows:

\*During game boot (GAME.LOADER1) world zone data (compressed) is read from disk into a temporary main memory buffer and copied into auxiliary memory.

\*During game launch (DRIVER, REGION.UNCOMPRESS.ALL) tile data is uncompressed from the world zones in auxiliary memory and copied into RZONE.ARRAY

\*During game launch (DRAW.SCREEN, TILE.LOOKUP.SCREEN), uncompressed tile data is copied into SCREEN.TILE.DATA

There are 9 world zones stored on the regional map (RZONE.ARRAY) at all times. Each zone is 16 tiles x 16 tiles, which results in 256 total tiles and 256 total bytes (uncompressed), fitting neatly into a page of memory.

More details on this process are found in the loader zone documentation <INSERT>.

**TILE SHAPE TABLES**

As mentioned, each shape is stored as a bit-map 2 screen bytes wide X 16 lines deep (14pixels x 16 pixels). Thus, each shape is 32 bytes in size (2 bytes \* 16 lines). The shape's are only 14 pixels wide because each screen byte contains a color bit which itself is not a pixel. Each shape can be displayed in any location on the screen within the 17x11 tile grid.

The shape tables (32 bytes in size) are loaded into auxiliary memory during game boot (GAME.LOADER1) starting at aux memory address $7000. Thus, the second shape table starts at $7020, the third shape table starts at $7040, and so on.

The graphics engine calculates the start address of each shape table using lookup tables (TILE.SHAPES.LO/HO), which use the screen location of the tile to draw as it's index. Tile screen locations are mapped to the SCREEN.TILE.DATA array via chart <INSERT>. For example, if the graphics engine needed to draw a tile in the second screen location (the upper left corner of the screen, 1 tile to the right), it would use $01 as the index to TILE.SHAPES.LO/HO.

Once the start address of the shape table is known the graphics engine copies it into main memory for using with the drawing subroutines.

**Animated Tiles**

There is an additional step if the Tile\_ID is in the animation range. Animation in general is handled by the Animation\_Manager.ASM module, but the general subroutines of the graphics engine still make sure that when they draw an animated tile on the screen that the current animation frame is used.

Each animation frame is 32 bytes in size. During game boot (GAME.LOADER1) the animation tiles are loaded into auxiliary memory starting at memory address $8000. Thus the first animation tile (4 frames), occupies $8000-$807F.

When processing an animated Tile\_ID the graphics engine uses the lookup tables mentioned before to retrieves the start address of the shape table, which in this case is the start address for the entire 4 frame shape table. The graphics engine then needs to calculated the start address of the shape table for the specific animation frame that is required.

The current animation frame is tracked in a variable (0-3), which is multiplied by 32 (using the ASL opcode). The product is added to the start address, which equals the start address of the current animation frame for the animated Tile\_ID.

**Closing Thought**

For non-animated Tiles\_IDs, why use lookup tables to calculate the start address of the shape tables instead of multiplication? Multiplication is very slow when the multiplier is anything other than a power of 2 (2,4,8,16,32,64,128 etc), which is less than or equal to 128. A multiplier in that range can be done using the ASL opcode. Any power of 2 larger than 128 requires a 16-bit multiplication routine.

This is important because these shape table address calculations are being done in the inner most loops of the graphics engine, and the graphics engine's routines have to be extremely fast to draw the screen in a time frame that is acceptable from a gameplay perspective.

\*\*note reference the screen array chart and rzone chart...or maybe the rzone chart only in the loader zone docs