**Technical Overview**

**Summary**

<GAME NAME> is written 100% in 6502 assembler. No construction sets were used. Code for a few routines was customized for use in the game, such as 32-bit division and random number generation (see Acknowledgements before for a list)

**Development Tool Chain**

Development is being done using the SBASM (version 3?) cross assembler by Stan Bergman <URL). The binary files created by the cross assembler are added to a disk image using AppleCommander. Testing is primarily done using the AppleWIN emulator and a physical Apple IIe. The disk images are copied to floppy disk using ADT PRO (by David Schmidt)

**Graphics**

The graphics for <GAME NAME> are managed using a tile-based graphics engine. Each shape is stored as a bit-map 2 screen bytes wide X 16 lines deep (14pixels x 16 pixels). These shapes can be displayed in any location on the screen within a 17x11 tile grid.

The primary input to the graphics engine is tile data. Each shape has an 8-bit Tile\_ID (/aka Tile\_Type), which is used by the graphics engine to determine which shape to draw. The game map is a long string of data comprised of a Tile\_ID associated with each tile on the map. This data is compressed on disk and in memory until it is needed.

The graphics engine is a series of subroutines which can be called to draw a single tile or row/column of tiles in location specified by the calling routine.

Using these subroutines and the tile data for the world map, the game can draw the tiles that make of the 1st screen the player sees at game launch and subsequently update the tiles on screen when the player moves. The later also incorporates a concept called graphics scrolling and page flipping which improve the speed and quality of the graphics rendering (these topics are discussed further in a later section).

**Animation**

<GAME NAME> contains a lot of animation including water, mobs and the player icon. Animation is implemented using 4 frames which are all in sync. Each frame is a bit mapped shaped the size of a tile which shows the shape in a slightly different position. The animation manager in the game starts at the top of the screen and flips each animation tile on the screen to the next frame, and then starts over at the top of the screen and does same thing over and over. Once the 4th frame is reached, it displays the 1st frame.

There are several ways to approach animation screen draws. Graphics books discussing animation will talk about techniques like draw-draw and draw-erase. <GAME NAME> uses draw-draw, and additionally page flipping to ensure the player can't see the animation being drawn.

For example, when the animation manager draws the next frame for each tile on the screen it does so to the background hi-res graphic screen. When complete it flips the background page to the foreground, so the first time the player sees the new screen it is fully drawn. In this model if animation is drawn to slow it will be detected as a lag between frame cycles rather than seeing frames out of sync on the screen, which can look very aesthetically displeasing.

I actually started using draw-draw to the foreground and it was fast enough for mobs, even when there were a lot on the screen. However, when the screen was 3/4 full or more with animated water, the draw rate was just too slow, even in assembly language, at least in the way I implemented it.

I think it may be possible to get animated to work as a foreground draw, but using page flipping fixed the problem so I moved on to other things.

<later section, explain how deep water works>

**Deep Under the Hood**

**Look Mom, No OS!**

<GAME NAME> does not use an operating system (DOS 3.3 or otherwise). Most commercial games in the 1980s created custom bootloaders which allowed the programmers to run the machine instructions created by their assemblers directly on the "bare metal". This means the code talks directly to the hardware (video screen, keyboard, disk drive, etc) since the operating system isn't there to handle that.

The reason for this approach was because of memory constraints. DOS3.3 takes up <X%> of the 48k of available main memory. To achieve high quality graphical games, the programmers simply could not afford to let the operating system (which wasn't needed, except usually RWTS) take up that much space.

I created a custom boot loader for <GAME NAME>. All disk/read writes are managed by a subroutine I wrote to interface directly with RWTS. RWTS (Read/Write Track/Sector) is the floppy disk drive controller.

**Text on Hi-Res Graphics Screen**

Hi-Res Character Generation (HRCG) in <GAME NAME> is facilitated through a hack of the COUT ROM routine commonly used to output characters to the text page.

The hack causes the COUT routine to instead call a custom routine which draws a shape table on the Hi-Res page. The shape table is selected based on the ascii value of the character COUT tells the custom routine to draw (i.e. the ascii value of whatever character the calling routine to COUT told it to output).

The shape tables for each ascii character were obtained from <SOURCE> *not sure yet what the final character set is that we'll be using.*

**NPC Schedules**

See Chart 1.3 and the file and subroutine documentation in NPC\_BUILDING.MANAGER.ASM

**Acknowledgements**

Code available under General Public License which was customized for use in the game includes:

<list author name and URL for each, verify GPL is correct license terminology>

32-bit division

random number generation

**GLOSSARY**

MO = MAP OBJECT

MOB = MONSTER OR EVENTUALLY PROBABLY NPCs.

NPC = NON-PLAYER CHARACTER