Notes on FontData

Joseph21
October 29, 2023

Introduction

Goal of this code is to find out how Javid encodes the sprite font into the PGE, and create a tool to apply alternatives font if so desired.

Note: I did some digging into this subject before, while developing my SDL based Game Engine (SGE) – see: <devpath>_12. Joseph21 - SDL_GameEngine_old code\Sprite fonts in code and the notes therein.

Approach

If you look at where and how the font sprite data is loaded in the PGE, you find this function olc_ConstructFontSheet() around line 3300:


```
3306  void PixelGameEngine::olc_ConstructFontSheet()
3307  {
3308      std::string data;
3309      data += "?Q`0001oOch0o01o@F40o0<AGD4090LAGD<090@A7ch0?00O7Q`0600>00000000";
3310      data += "O000000nOT0063Qo4d8>?7a14Gno94AA4gno94AaOT0>o3`oQ400o7QN00000400";
3311      data += "Of80001oOg<7O7moBGT7O7lABET024@aBEd714AiOdl717a_=TH013Q>00000000";
3312      data += "720D000V?V5oB3Q_HdUoE7a9@DdDE4A9@DmoE4A;Hg]oM4Aj8S4D84@`00000000";
3313      data += "OaPT1000Oa`^13P1@AI[?g`1@A=[OdAoHgljA4Ao?WlBA7l1710007l100000000";
3314      data += "ObM6000oOfMV?3QoBDD`O7a0BDDH@5A0BDD<@5A0BGeVO5ao@CQR?5Po00000000";
3315      data += "Oc``000?Ogij70PO2D]??0Ph2DUM@7i`2DTg@7lh2GUj?0TO0C1870T?00000000";
3316      data += "70<4001o?P<7?1QoHg43O;`h@GT0@:@LB@d0>:@hN@L0@?aoN@<0O7ao0000?000";
3317      data += "OcH0001SOglLA7mg24TnK7ln24US>0PL24U140PnOgl0>7QgOcH0K71S0000A000";
3318      data += "00H00000@Dm1S007@DUSg00?OdTnH7YhOfTL<7Yh@Cl0700?@Ah0300700000000";
3319      data += "<008001QL00ZA41a@6HnI<1i@FHLM81M@@0LG81?O`0nC?Y7?`0ZA7Y300080000";
3320      data += "O`082000Oh0827mo6>Hn?Wmo?6HnMb11MP08@C11H`08@FP0@@0004@000000000";
3321      data += "00P00001Oab00003OcKP0006@6=PMgl<@440MglH@000000`@000001P00000000";
3322      data += "Ob@8@@00Ob@8@Ga13R@8Mga172@8?PAo3R@827QoOb@820@0O`0007`0000007P0";
3323      data += "O`000P08Od400g`<3V=P0G`673IP0`@3>1`00P@6O`P00g`<O`000GP800000000";
3324      data += "?P9PL0200`<`N3R0@E4HC7b0@ET<ATB0@@l6C4B0O`H3N7b0?P01L3R000000020";
3325
3326      fontSprite = new olc::Sprite(128, 48);
3327      int px = 0, py = 0;
3328      for (size_t b = 0; b < 1024; b += 4)
3329      {
3330          uint32_t sym1 = (uint32_t)data[b + 0] - 48;
3331          uint32_t sym2 = (uint32_t)data[b + 1] - 48;
3332          uint32_t sym3 = (uint32_t)data[b + 2] - 48;
3333          uint32_t sym4 = (uint32_t)data[b + 3] - 48;
3334          uint32_t r = sym1 << 18 | sym2 << 12 | sym3 << 6 | sym4;
3335
3336          for (int i = 0; i < 24; i++)
3337          {
3338              int k = r & (1 << i) ? 255 : 0;
3339              fontSprite->SetPixel(px, py, olc::Pixel(k, k, k, k));
3340              if (++py == 48) { px++; py = 0; }
3341          }
3342      }
3343
3344      fontDecal = new olc::Decal(fontSprite);
3345
3346      constexpr std::array<uint8_t, 96> vSpacing = { {
3347          0x03,0x25,0x16,0x08,0x07,0x08,0x08,0x04,0x15,0x15,0x08,0x07,0x15,0x07,0x24,0x08,
3348          0x08,0x17,0x08,0x08,0x08,0x08,0x08,0x08,0x08,0x08,0x24,0x15,0x06,0x07,0x16,0x17,
3349          0x08,0x08,0x08,0x08,0x08,0x08,0x08,0x08,0x08,0x17,0x08,0x08,0x17,0x08,0x08,0x08,
3350          0x08,0x08,0x08,0x08,0x17,0x08,0x08,0x08,0x08,0x17,0x08,0x15,0x08,0x15,0x08,0x08,
3351          0x24,0x18,0x17,0x17,0x17,0x17,0x17,0x17,0x17,0x33,0x17,0x17,0x33,0x18,0x17,0x17,
3352          0x17,0x17,0x17,0x17,0x07,0x17,0x17,0x18,0x18,0x17,0x17,0x07,0x33,0x07,0x08,0x00, } };
3353
3354      for (auto c : vSpacing) vFontSpacing.push_back({ c >> 4, c & 15 });
3355
3356  }
3357
```

With the yellow notes marked you can see

- D – data – hardcoded in the header file
- P – processing stuff
- 1 – concerns the initialization of the font sprite (variable fontSprite in the PGE) using data at D1;

- 2 – concerns the initialization of the spacing array (variable vSpacing in the PGE) using data at D2;

So we will first find out how to create an alternative data string for part D1 from a font sprite file, and then how to create the (alternative) spacing info that goes with it for part D2.

Font data creation

Need to know – in the encoding approach that Javidx9 implemented:

- Sprite pixels are encoded in 0 for empty (either black or blanc) and 1 for non-empty (every other value) – so it's best to work with B/W font sprites;
- The sprite is scanned in column – row order – Not sure why he chose to do that, probably because it made a better match between the algorithm and the sprite size;
- 6 pixels are grouped in 1 byte, and an offset of 48 is applied to that byte. The choice of 6 bits in combination with the offset of 48 ensure that the result is in the range [48, 114], which are all nicely printable characters in the character set. So this helps to produce readable characters to represent the data.
- 4 bytes are grouped into 1 int32_t. Note that the endianness is reversed here (see also the alternative notes from the SGE development);

I first wrote a piece of code that performs this trick, and applied it to the original font: the nes font Javidx9 introduced for the RPG video series). Tweeked it until the output was identical to the original datastring in the PGE.

Then I created my own font sprite: nesfont_slim.txt

Used some excel trickery to get this done.

Then I created a convenience function to read in the sprite font in .txt format, and create a black and white sprite from it.

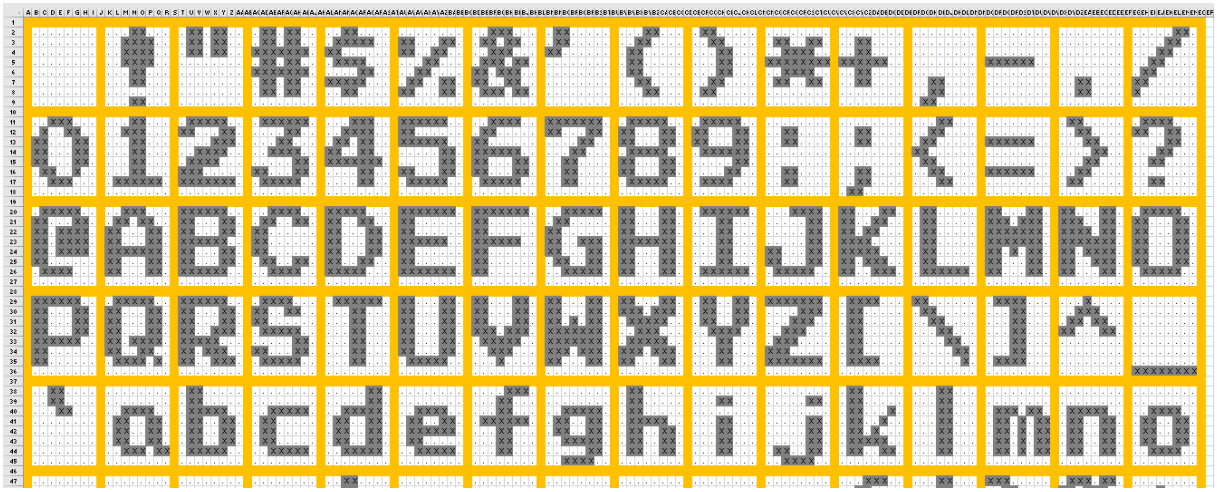
Then I applied my previous algo to it, and copy-pasted the resulting data string into my copy of the PGE.

Spacing

The spacing puts x and y in one 8 bit integer (uint8_t). You can see this looking at the auto for loop in this snippet:

```
3345
3346    constexpr std::array<uint8_t, 96> vSpacing = { {
3347        0x03,0x25,0x16,0x08,0x07,0x08,0x08,0x04,0x15,0x15,0x08,0x07,0x15,0x07,0x24,0x08,
3348        0x08,0x17,0x08,0x08,0x08,0x08,0x08,0x08,0x08,0x08,0x24,0x15,0x06,0x07,0x16,0x17,
3349        0x08,0x08,0x08,0x08,0x08,0x08,0x08,0x08,0x08,0x17,0x08,0x08,0x17,0x08,0x08,0x08,
3350        0x08,0x08,0x08,0x08,0x17,0x08,0x08,0x08,0x08,0x17,0x08,0x15,0x08,0x15,0x08,0x08,
3351        0x24,0x18,0x17,0x17,0x17,0x17,0x17,0x17,0x17,0x33,0x17,0x17,0x33,0x18,0x17,0x17,
3352        0x17,0x17,0x17,0x17,0x07,0x17,0x17,0x18,0x18,0x17,0x17,0x07,0x33,0x07,0x08,0x00, } };
3353
3354    for (auto c : vSpacing) vFontSpacing.push_back({ c >> 4, c & 15 });
3355
```

If you compare the x spacing values with the sprite sheet…



… it’s pretty easy to see that the x value of the spacing represents the nr of empty columns on the left of the character.

```
1143        std::vector<olc::vi2d> vFontSpacing;
```

After some examination how the spacing y value is used, it turns out it represents the width of the character + 1. *However never more than 8 – so for instance the underscore is 8 pixels wide, but its spacing is 8. The same holds for the asterisk*