Implementation notes: GPU text rendering with vector textures

Sat, Jan 2, 2016

- The demo data was generated by a C++ program that reads a PDF and spits out three files: a vector atlas, a vertex buffer and a JSON file with data for each page (dimensions, offset into vertex buffer). It could be extended to render all elements of a PDF (images, vector art).
- The vertex buffer contains only world position, color and the texture coordinates of the curve data. The grid texture coordinates and size are stored in the first two slots of the curve data and looked up in the vertex shader. This is so we don't need to store this identical data for every instance of a glyph.
- The vertex buffer file is processed to improve gzip compression: vertex attributes are de-interleaved and the positions are delta coded. This lets an 18 MB file be compressed to 1.5 MB.
- Bezier curve coordinates have 16 bit precision. Each two component vector is encoded in a four byte RGBA texel by splitting the high bits and low bits

into different channels and reconstructing them in the shader. In WebGL2 we could use uniform buffer objects instead.

- Each grid cell stores not only the bezier curves that intersect it but also a boundary region around it. This is because a pixel near the cell's boundary may have a window overlapping neighboring cells.
- The vertices of the quad that is rendered for each glyph are extended outwards by a fixed amount to allow for filter windows reaching past the border of the glyph. This is particularly needed when zoomed out. The latest GPUs support conservative rasterization which is a better way to solve this.
- Indices are one byte each, meaning grids can store 4 indices per RGBA texel. To support 8 indices per cell, a second grid may placed to the right of the first. The preprocessor will only use a second grid if the glyph can't be diced well using only 4 curves per cell.
- We should revert to using a standard texture atlas when text is small enough that one pixel spans more than one grid cell. It could generated at load time by rendering all the glyphs into a texture using the vector atlas. The demo doesn't have this implemented, which is why there is aliasing when very zoomed out.
- It's possible to have failure cases where there are too many curves in one spot, overflowing the maximum number of curves per cell. This is unlikely to occur with most fonts but could be handled by rendering the glyph as a triangle mesh or traditional atlas font. It's also possible to allocate a variable amount of space per grid cell using off an offset/size tuple, but this would be easier to implement on future versions of WebGL.
- LCD subpixel antialiasing, aka ClearType, is tricky to implement in WebGL due to the lack of information about the display's subpixel order. We could assume it is RGB and implement it anyway. Implementing it in the font shader would only work for text that doesn't need to be blended onto a background, since otherwise we would need 3 alpha channels. One idea is to do it in the style of temporal reprojection: have a random subpixel jitter per frame and accumulate results from previous frames with subpixel offsets.

- The fragment shader is probably bottlenecked by memory bandwidth: each bezier curve is 3 x 16 bit vec2 = 12 bytes, and we may read up to 8 of them. We could halve the amount of bandwidth required by reducing the bezier curve precision to 8 bits per component and storing clipped beziers in each grid cell instead of indices. The reduced precision with a 10x10 grid wouldn't be noticeable until the glyph was zoomed to 2560x2560 pixels. I've not implemented this yet.
- Because outlines are composed of many beziers chained together, the starting point of bezier n is usually the end point of bezier n - 1. To take advantage of this the redundant point isn't actually stored and the cell index for bezier n points to the end point of bezier n - 1, saving texture space.
- The shader needs to know whether the the fragment sample position is inside or outside the glyph. This is necessary for dealing with the case where we find no intersections within the pixel window — is the pixel fully covered or uncovered? To find this out we store a flag in each grid cell telling us whether the cell midpoint is inside or outside. We can trace a line from the cell center to the fragment's sample position and flip the flag each time we find an intersection with the cell's beziers.
- The cell center's inside/outside flag is encoded using the order of the first two indices. The order of the second two indices encodes whether there are more than 4 indices stored in the cell — if there are 4 or fewer we save a texture lookup.
- We could use the bezier curve tangent at our ray intersection points to give a better approximation of our wedge integral, reducing the number of samples needed.
- Some browsers (IE, Edge, Safari) premultiply PNG images when you load them, meaning they multiply the RGB channels by the alpha channel. This corrupts the vector data, and there appears to be no way to disable it. Frustrating to say the least. To get around it, the demo downloads the atlas texture as a raw data file instead of using the browser's image loading capability.

Comments