Text as Texture

J. Terry Corbet
Campbell, California
July 7, 2013

Background

This document is best understood as an extended comment on Moussa Dembélé's very helpful blog postings under the topic of "Stage3d AGAL experiments".[1] His is one of many similar Internet sources authored by folks with various backgrounds, resources and motivations for making the considerable investment necessary to create and maintain tutorial examples of techniques associated with the Adobe Flash and AIR facilities for delivering desktop, web-enabled or mobile applications that leverage a Graphical Processing Unit [GPU]. Amongst those that he cites, the Molehill postings by Marco Scabia[2] and Jackson Dunstan[3] stand out, and are probably already in your 'bookmarks' list.

I don't have either the background or the resources for joining such a list of luminaries. I do have an interest in the topics and a desire to contribute to the fund of resources available to anyone interested in 3D programming challenges. The reason I am 'tagging along' on Moussa's work is that I appreciate, in what he has already given us, his genuine desire to be helpful. His approach and presentation are in the helpful manner of Scabia and Jackson, but he also has different interests and perspectives that are complementary of the other sources found on the Internet.

Both he and I, as well as most readers, are ever-watchful for the publication of documentation and working examples from the primary toolkit/framework folks at Alternativa3D, Away3D, EasyAGAL, Flare3D, Minko, et. al. We strive to use the best practices and tools commensurate with our limited budgets for applications that mostly are developed for our personal interests, friends and family, or often simply for mind stretching exercise. Thus, without wishing to re-engage Jean-marc in his famous blog postings as to why learning to code in AGAL is the wrong thing to do, we muddle on in the belief that working at this low level will eventually help us to employ any higher level tools we can afford.

Why Text?

In my own areas of 3D interest there are no ogres, no cars, no guns. Indeed, in most of the applications that are of interest to me, there is nothing usually in motion other than the camera. In short, judging by the postings, my 3D world is nothing short of dead and uninteresting – fireworks don't go off, fog does not fill the room and audio is mostly limited to voice over commentary.

Ok, now that the door has swung shut after the rush to the exit, who's left – maybe some historians, geographers, statisticians, economists, architects, engineers, anthropologists, cooks, and photographers. Any game maker who is convinced that the purpose of a game ought to be the elimination of any need for anyone to read anything, is right to have left. On the other hand, even if your game only occasionally has no alternative to end user communication other than the display of a score of some sort, maybe you should stick around because I am going to talk about the use of Textual Components that might be as small and as brief as single Glyphs that could masquerade as icons or other symbols not requiring grammar or much in the way of typography.

Sure, I appreciate the basic fact that a good deal of the value in 3D graphics is that they provide a means of end-user engagement that is decidedly more fun than reading a book, but I have, over the years, found many cases in which the total multi-media end-user experience I am trying to provide absolutely depends upon the integration of textual content. "Yes,", I can hear some of those who are about half-way out the door saying, "I also have some need for text in what I am doing, but most of it does not need to be involved in the 3D pipeline."

"Here, here.", and that is what makes tools like those from the good folks at Starling and Feathers so valuable. But there are, at a minimum, times when the display of information needs must be susceptible of partici-

---

1 See: http://mousman.com/agal/.
2 See: http://www.adobe.com/devnet/flashplayer/articles/how-stage3d-works.html.
3 See: http://jacksondunstan.com/articles/1661.

pation in whatever perspective exists for a scene. There are, moreover, compelling applications in which the display of textual information must behave nicely in the face of camera zooms over a wide range of distances 'into' the scene.

Ok, I'll Stay a Little Longer, but Why Yet Another Tutorial Example?

So, if I have to fight this hard to convince anyone that text is a worthwhile topic, how much applause can I expect if I claim that 'Text as Texture' is probably harder to get right than all that marvelous bling of diffuse, specular or ambient light and shadows? I agree that unless we are talking about an attempt to replicate the game of "Boggle" with actual 3D Glyphs bouncing off one another, most software problems concerning text are rather easier than physics and other animations.

But, just to convince you that the proper handling of text is not 'slam dunk' simple, I pose this challenge. Use whatever tools in whatever work flow you like, but at the end of the day I would like to see s simple ruler appear on my monitor. I would like the etch marks on the ruler to be perfectly matched to the real world, so if the image on my screen of a one-foot ruler is properly displayed, it will consume one-foot of glass across the screen. The numbers for each inch should be displayable in a user-selectable range of font faces. Once your demo is working, write a description of the mathematics required to correctly place the ruler in World Space as a function of whatever controls are provided over the parameters of the Perspective Lens in use.

The answer to the question here is that most of the available tutorials dealing with applying textures to objects in a 3D scene do not do a good job of handling the 'pixel-perfect' requirements of text. If you want something in your terrain to be blurred, you will have no trouble finding experts in blurring, with or without GPU assistance. Or, to go in the other direction, if you want something in your scene to be well etched, you will have no trouble finding experts in edge detection, just don't expect the example to concern itself with the edges in complex, concave polygons like those exhibited by the little letters of your everyday alphabet.

The answer to the question here is that the proper rendering of text in a 3D context is hard, and for whatever brilliance is shown by those folks at NVIDIA and AMD, it seems like architecting optimized solutions to the needs of handling text is not on the table. You know what I would like to do 128 or 256 times in parallel, execute the core functions of PostScript, that's what I'd like to do! But, since the hardware of the GPU is much better optimized to solving problems in interpolating the particles of smoke coming out of the back of some NASCAR beast or the reflection of the fire blast coming out of the front of an AK-47, getting that hardware to work correctly in the enlargement of the Capital Letter M is 'left as an exercise to the student'.

The Problems

Let me list the problems that I run into when trying to use text as a texture. These problems are manifest even in the constrained environment of applying such texture to rudimentary objects that should be a 'slam dunk'. I'm talking about your basic, primitive, single-sided plane. But, using most frameworks or following most AGAL examples, when text dresses up a plane, you get:

- ✔ a distorted aspect ratio
- ✔ the jaggies

The way the GPU applies a texture to a triangle – even when the triangle is perpendicular to the camera – will almost always produce both of these problems.

So, "What", you may ask, "is the big deal?" Those problems exist for every application of any texture. Yes, and if your application mostly zooms by leaving a trail of fire and smoke, you are, no doubt, so happy with the Molehill-enabled frame rate that the problems simply are not worth any more optimization. But, with text, with textual messages that we expect the user to read, optimization is required.

If the leaf on some tree that appears a hundred times in the background has some minor distortion or weak anti-aliasing, it probably won't be noticed, probably won't ruin your game. The same result, however, if you are displaying the words to your country's national anthem, will drive people mad.

Enough Talk, Show Me Some Code

Let me describe the attached, commented demo code. I have tried to keep it compact so that you can mostly concentrate on the specific matters at hand. I have mostly simply borrowed the code from Moussa, Marco and Jackson since most of you will have already devoted considerable time and energy to studying those examples, so common aspects of a general AGAL rendering solution will be familiar. The major components and objects of the example are simply some triangles, a bitmap texture and a camera.

Here, the triangles are kept as simple as possible, we will be mapping our text texture to two rectangular planes. Of course in all the framework toolkits, there is some sort of a mesh object that manages triangle geometry, and indeed, most of the interesting API controls for a scene are based on the mesh object. I have added the most minimal mesh, just so the use of the techniques I am discussing will take on the basic semblance of a real application.

```
 98      var vertices:Vector.<Number> = Vector.<Number> ([
 99          -1.5,  0.5, 0, 0, 0,
100           1.5,  0.5, 0, 1, 0,
101           1.5, -0.5, 0, 1, 1,
102          -1.5, -0.5, 0, 0, 1,
103
104          -.25,  1.5, 0, 0, 0,
105           .25,  1.5, 0, 1, 0,
106           .25, -1.5, 0, 1, 1,
107          -.25, -1.5, 0, 0, 1
108          ]);
109      // Create VertexBuffer3D. 8 vertices, of 5 Numbers each.
110      vertexBuffer = context3D.createVertexBuffer (8, 5);
111      // Upload VertexBuffer3D to GPU. Offset 0, 8 vertices.
112      vertexBuffer.uploadFromVector (vertices, 0, 8);
113
114      var indices:Vector.<uint> = Vector.<uint> ([
115          0, 1, 2, 2, 3, 0,
116          4, 5, 6, 6, 7, 4
117          ]);
118      // Create IndexBuffer3D. Total of 12 indices. 2 triangles of 3 vertices for 2 planes.
119      indexBuffer = context3D.createIndexBuffer (12);
120      // Upload IndexBuffer3D to GPU. Offset 0, count 12.
121      indexBuffer.uploadFromVector (indices, 0, 12);

130      spm1 = new SillyPlaneMesh (0, new Vector3D (0.4, -0.6, 0.0), 3.0, 1.0,
131          Vector.<Number> ([.8, .8, 0.0, 1]));
132      spm2 = new SillyPlaneMesh (6, new Vector3D (0.7, 0.4, -0.01), 0.5, 3.0,
133          Vector.<Number> ([0.0, .8, .8, 1]));
```

Lines **99-102** define a plane that is wider than high, and lines **104-107** define one that is tall and thin. The UV coordinates for each of them is the same because it is the intention of the modeler to apply exactly the same texture to each of them. Lines **115** and **116** provide the same, clockwise triangular sequence of indexes into the Vertex Buffer assuring that the 'back' of these planes will be culled, as it is not the modeler's intention that a camera view 'from the rear' will ever be needed in the application.

The global variables **spm1** and **spm2** point to very rudimentary mesh class objects with properties that will provide three important facts when it comes times to render these planes:

- their position in World Space

- their dimensions
- a fill color

We will look in more detail at other aspects of the setup/initialization code later, but this should be enough to get us started in looking at the first problem usually encountered when using text as a texture.

"Hey, just a minute, where's the bitmap?" This demonstration extends to the frequent requirement for dynamically-generated text to appear as a texture, so there is no embedded bitmap, no bitmap being loaded at startup. What is embedded is a font, and that font is used to generate different bitmaps containing different text. Since this tutorial is not about fonts, I won't go into any more detail about this other than to make two points which I think are important.

First, the solutions provided here for a first-class handling of text in a 3D context extend naturally from considerations of the first-class handling of text in a 2D context. That's actually a big part of my disappointment that Adobe's 3D tool has not been given the attention that has defined Adobe's years of leadership in the business of font creation and font rendering software. One would ideally like to see a seamless work flow from Illustrator to Text Texture. So, while that seamless work flow does not exist 'out of the box' it has been my intention to address some of it, and we will look at that in more detail a little later.

Second, wherever there are font considerations there are layout considerations, and that is the case when moving up to 3D rendering from 2D. The solution here, provides some help in that area. Please look at lines **143-145** and as you play with the demo code, change which kind of text layout the software will be dealing with. In the base case, I am mimicking a multi-line layout of some text with the usual horizontal flow. The second test case looks at a layout that is rather more frequent in our 3D environment than in the 'flat' world of 2D – a columnar display. The final test case is also common in the 3D world – the use of a single font glyph. Indeed it is this use case that often causes the modeler to become concerned about the problems of texture mapping because the use of the glyph may take on special significance as a very visible symbol for which the most precise preservation of appearance is critical.

A Shader for Text – Reducing the Distortion Problem

The intended, natural aspect ratio of whatever text, in whatever font you want to render, becomes distorted because the aspect ratio of the plane to which the texture is being applied is different from the aspect ratio of the font. The way that interpolation is done by the GPU, the plane wins, so your font loses. The fix demonstrated here is to limit the text applied during interpolation to a sub-rectangle whose aspect ratio is based on that of the source text. Any remaining surface on the targeted plane will be background color filled.

As soon as I type that last paragraph, I can almost hear you singing the lines from the great old tune recorded so memorably by Peggy Lee, "Is that all there is?" Well, yes, that's most of what it is, but its probably a little better than you might think if you think that means that we will just render the text in some fixed space and fill everything else up with splashy colors. The solution at least meets you half-way – half-way between the desire to completely cover some plane with your text and the desire for the text to not be stretched beyond recognizability.

It does that by taking into consideration the aspect ratio of the font, and making sure that it wins, not the aspect ratio of the surface being painted. If the dimensions of the receiving surface in the horizontal are closer to the source text, the horizontal paint job will extend end-to-end and fill color will only be applied to the non-conforming height of the plane. Likewise, if the dimensions of the receiving surface in the vertical are closer to the source text, the paint job will extend top-to-bottom and fill color will only be applied to the non-conforming width of the plane.

The AGAL code to implement this solution is divided between the Vertex and the Fragment shaders.

```
148      // Vertex Attribute Register 0 holds the target object's local position.
149      // Vertex Attribute Register 1 holds the UV mapping from target to texture.
152      "m44 op, va0, vc0 \n" +
153      "mov vt0, va1 \n" +
154      "mul vt0.xy, vt0.xy, vc4.xy \n" +
155      "add vt0.xy, vt0.xy, vc4.zw \n" +
156      "mov v0, vt0"
```

In the former we pre-scale the dimensions of the UV coordinates from their native settings to the appropriate aspect ratio. In the direction of the non-conforming dimension, the positions are offset so as to cause the centering of the result.

```
159      // Fragment Constant Register 0 holds 0-1 values to drive true-false logic.
160      // Fragment Constant Register 1 holds the Fill Color used to pad the texture.
163      "tex ft1, v0, fs0 <2d, linear, miplinear, clamp> \n" +
164      "sge ft2.xy, v0.xy, fc0.xx \n" +
165      "mul ft2.x, ft2.x, ft2.y \n" +
166      "slt ft3.xy, v0.xy, fc0.yy \n" +
167      "mul ft3.x, ft3.x, ft3.y \n" +
168      "mul ft4.x, ft2.x, ft3.x \n" +
169      "mul ft1.xyz, ft1.xyz, ft4.xxx \n" +
170      "sub ft5.x, fc0.y, ft4.x \n" +
171      "mul ft6.xyz, fc1.xyz, ft5.xxx \n" +
172      "add ft6.xyz, ft6.xyz, ft1.xyz \n" +
173      "mov ft6.w, fc1.w \n" +
174      "mov oc, ft6"
```

When the GPU's interpolator uses those adjusted UV values, depending upon the projection trigonometry, the calculated results can go out of range. The code above seems long and complex, but it is just insuring that out of range conditions are handled successfully.

In summary, the steps necessary to insure that text when rendered as a texture has proper proportions is neither complex nor consumptive of measurable GPU resources. That's the good news. The bad news is that the results still are often not so pleasing.

A Shader for Text – Reducing the Jaggies Problem

The problem is that the difference between the area of the source bitmap and the on-screen result, depending upon the projection trigonometry may result in the GPU's interpolator introducing jagged edges for the text outlines. As with everything else we have discussed here, this is a common problem, not at all unique to the use of text as a texture. Consequently, many of the techniques for overcoming the problem should be applied to our use case.

The primary technique is mipmapping. It is the same technique as used for games, but the sense is generally opposite. In games, the modeler will use mipmaps to gracefully reduce the clarity of the background scene objects in order to conserve other important GPU resources. In our case we want to gracefully enhance the clarity of foreground scene objects even at the expense of GPU resources.

To understand the mipmap technique shown here, let's step back a bit to the more familiar world of 2D text. Outline fonts elegantly apply scalable vectors to allow you to print an amazing range of character sizes meeting the needs of book printing as well as wall-sized poster production. From the smallest readable font size to the largest, there will only be perfectly straight lines and graceful curves – no jaggies.

The genesis of this article was Moussa's provocative posting concerning <<vectorized>> shaders. Would that we could actually get AGAL to instruct a GPU on the proper way to draw such lines and curves, but we cannot. The GPU can only attempt to interpolate an expanded text character by bitmap methods, and that introduces jaggies. The best we an do to reduce the jaggies is to try to exactly match the font size in a texture with the final drawing size on the screen. Unless our application be entirely static, that goal cannot be met.

The best we can really do is to try to provide a range of text sizes as close as possible to the dynamic range of sizes that will need to be projected in the application. That is what is shown in this demo. Look at the function **`uploadTextureWithMipMaps()`** [**`188-209`**]. It is an adaptation of the one that Scabia uses when he introduces us the the basics of mipmapping. The adaptation is crucial.

Whereas his example shows the usual case of starting with some bitmap of some highest level of precision, and scaling down, thus avoiding jaggies at the expense of visual fidelity, we start at some largest anticipated

font size and scale down gracefully by using successively smaller font sizes that preserve visual fidelity through the actual use of vector graphics primitives in 2D. It is exactly the kind of <<vectorization>> that Moussa has suggested, but it is implemented in a rather different manner.

Rather than my trying to write about it, the purpose of the demo is for you to just be able to observe the behavior. Being able to do that is provided in two ways. First, so that you can see the 2D result, the code at lines **199** thru **203** have been added. As each successively smaller font is rendered into a texture, a snapshot of the bitmap is added to the 2D stage. Viewed in that manner, you can quickly see the different font sizes that where chosen based on the anticipated viewing requirements of the applications. Once you have gotten your head around that part of the implementation, it might be more helpful to no-op those lines if you wish.

The manner in which I have implemented the texture maps makes it possible to view the effects of GPU interpolation at work by application of the camera controls.

- Up and Down Arrows move camera Up and Down
- Left and Right Arrows move camera Left and Right
- Home Key moves camera Forward
- End Key moves camera Backward.

At each new Zoom Level the camera will be adjusting the projection trigonometry resulting in a different decision by the interpolator as to how to solve the problem. At some critical shift points, the GPU will select a different mipmap which means a different native font size. Between those shift points the GPU will be attempting to solve the problem by interpolation. In the real application, of course, the contents of each mipmap would be identical as to content, only varying by native size, so your eye would not discern the transitions, they would be seamless.

Like a TV Pitch Man – Wait, There's More!

There are two additional techniques for achieving better results when using text for a texture. They do not have to do with 3D, so they do not use AGAL, but they add to the overall results and should be understood as being a part of a panoply of techniques at your disposal in the Flash/AIR application space. Both additions are found in the **`paintText()`** function.

First, in the vicinity of the code between lines **241** and **245**, you will notice that we are able to take advantage of Filters to increase the visual impact of the text that will eventually become a texture. If some combination of filters make a 2D bitmap more rich, that richness ought not to be lost in 3D.

Second, before we had AGAL, we had PixelBender. As with built-in filters, if there are specialized filtering effects best implemented in PixelBender, we an use them in combination with our GPU-based solution. In this case, I am solving a common problem in the use of text in a 3D scene. In the real world the planes upon which text might appear will not always be simple Bill Boards conveniently facing the camera. The most common 2D technique for emphasizing text when it is commingled with other visual content is draw a box around it. Between lines **259** and **264** we generate the proper border to be painted around each of the mipmaps when it is finally painted onto the plane. Nicely-behaved lines dynamically generated in that manner will minimize jaggie problems and are another way in which Moussa's concept of <<vectorized>> shading might be implemented.