Fresh Texture Atlases

Let me say first that it is not clear that texture atlases are a definite performance win on iOS hardware. This whole feature may be a red herring. That said, I will design it with a view to implementing it only after its efficacy has been experimentally confirmed.

# Current Support

Fresh currently supports texture atlases in a passive sense. In FreshGraphics, texture coordinates and transforms are fully supported. Mipmapping is generally not used and so does not present problems (though this would change in a truly 3D or highly scaled 2D game).

In FreshScene2D, DisplayObjectWithMesh inherits this generalized, agnostic support for atlases.

Sprite supports atlases slightly more directly through its textureWindow property. I don’t believe I’ve tested this feature thoroughly, but in principle Sprites can share a texture and use a “textureWindow” to select a rectangular area within that shared texture. That is indeed texture atlasing, at least from the run-time perspective. But since there is no current support for authoring atlased textures or textureWindows, this feature is not yet really useful.

# Needed Support

For texture atlasing to be useful, we need the ability to:

1. Given a set of textures, produce a unified atlas texture along with meta-data about texture slices (aka “windows”) within the atlas.
2. Simplify the designation and use of texture slices. That is, treat a texture slice as an asset, so that it may be named and therefore easily specified by a Sprite or other graphical object. Ideally, it would be as easy to designate a texture slice as it currently is to designate a texture.

The key challenge in providing this latter feature is the difficult associated with fiddling with specific texture coordinates and scales. Fundamentally, a texture slice is a pointer to a texture plus a texture coordinate rectangle in the form of four floating point numbers (*left, bottom, right, top* or perhaps *left, bottom, width, height*). Having Sprites or other DisplayObjects specify their window numerically is tedious and error-prone. Rather than this:

<object class=”Sprite”>

<pTexture>Texture’shared’</pTexture>

<textureWindow>(0.125, 0.45, 0.2, 0.2)</textureWindow>

</object>

…we want this:

<object class=”Sprite”>

<pTexture>TextureSlice’Monster’</pTexture>

</object>

Simple enough. The question, though is how class TextureSlice relates to class Texture, and how they both relate to DisplayObjectWithMesh’s Texture::Ptr pTexture member.

It might make sense for TextureSlice to extend Texture. Now pTexture can point directly to a texture slice and use it like a texture (e.g. calling Renderer::applyTexture() or Texture::apply()). But there are a couple of problems with this arrangement.

* A TextureSlice can’t really be applied independent of knowledge about the current VBO’s texture coordinates. Do we assume that the VBO’s coordinates have min and max (0,0) and (1,1) and that the slice should therefore both scale and translate them? Or do we assume that the VBO’s coordinates have a max that is pre-set to the number of slices within the atlas (as with fonts and most grid-based atlases) and that we should therefore only translate the coordinates?
* A TextureSlice fundamentally shares a texture with other slices. If a TextureSlice *is* a texture, then it has its own OpenGL texture Id, its own information about alpha and dimensions, potentially its own copy of the texture pixels (where retention of this information is desired). In other words, TextureSlice “wants to be” a small, lightweight class. If it extends Texture, it won’t be.

Actually, this last protest is a bit trumped-up. In practice we don’t retain a copy of the texture pixels. It is easy enough (though not trivial) to expand Texture to include the notion of sharing an id with another texture. And the “redundant” alpha and dimensions information wouldn’t actually be redundant for a slice, because a slice can indeed have its own alpha setting and it certainly has its own dimensions. Maybe that protest was entirely invalid, and a slice truly can be a texture.

The former protest can also be addressed, though perhaps not as elegantly as we might like. Regardless of the design, we do have to take care of various texture coordinate configurations as we apply a texture transform based on a slice. Inheriting from Texture only encourages us to do this in as invisible a way as possible. Yet it cannot always be invisible. Indeed, it may be that a texture slice can *never* apply the transform, but only store information about the window, which the user of the slice must interpret and apply itself. This seems also an unpleasant extreme.

Perhaps we could do this.

class Texture { public: virtual void apply(); }

class TextureSlice {

public:

virtual void apply() override

{

apply( vec2( 1.0f, 1.0f )); // Assume default sprite UVs

}

virtual void apply( const vec2& texCoordDimensions );

{

// r is the Renderer

r.setToIdentity( MAT\_Texture );

r.scale( window.dimensions/texCoordDimensions, MAT\_Texture );

r.translate( window.position, MAT\_Texture );

Super::apply();

}

}

|  |  |  |  |
| --- | --- | --- | --- |
| window dimensions | quad tex coords | desired scale | implied equation |
| 0.25 | 1.0 | 0.25 | *d=w\*q* or *d=w/q* |
| 0.125 | 1.0 | 0.125 | *d=w\*q* or *d=w/q* |
| 0.25 | 0.5 | 0.5 | *d=q-w* or *d=w/q* |
| 0.125 | 0.5 | 0.25 | *d=w/q* |
| 0.5 | 0.5 | 1.0 | *d=w/q* |
| 0.5 | 0.25 | 2.0 | *d=w/q* |

Well, *d=w/q* it is, then.

So, Sprites render easily without knowledge whether they have a texture or a slice:

void Sprite::draw()

{

// …

getTexture()->apply();

// …

}

An object needing to give more detail about texture coordinates would need to know about the slice specifically:

void DisplayObjectWithMesh::draw()

{

getTextureSlice()->apply( uvSize() );

}

But these examples are somewhat disingenuous, because a Sprite *is-a* DisplayObjectWithMesh, and so it would have the getTextureSlice() and uvSize() methods anyway.

Seems valid though. I feel confident in this design.

But I won’t implement it until I’m confident that the performance gains are genuine.