RAGE iOS format notes

This is my documentation of the RAGE iOS map format, as determined by a lot of poking, prodding, and browsing through Hex editors (and eventually a few hints from John Carmack himself. Thanks!) By no means is it definitive or guaranteed accurate, and while many elements have been verified via parsing and rendering the results, some of it may simply work through sheer dumb luck. Tread carefully.

It should be noted that RAGE and anything related to it is the property of id Software, who does not authorize or endorse this document in any way.

Header

Lumps work similar to those found in Quake 3 or other id formats, but rather than giving the lump size in bytes we are given the number of elements in a lump. Lump offsets are still given in bytes.

The only other element that might merit some explanation is the Max Texture Count. That appears to be the maximum number of textures that are used in a single scene by this map. You should pre-allocate at least this many textures, and probably another 16 besides for pre-caching.

`string[4]` - Magic Number? Hex: 29 55 44 77 or ASCII: “)UDw”
`string[128]` - File path that looks like it references the map’s source files
`long` - Texture size, typically 1024. All textures used by the map will be this size (square).
`long` - *unknown*, always 2
`long` - *unknown*, always 2

`ulong` - Lump 0 Offset (Vertices)
`ulong` - Lump 1 Offset (Indices)
`ulong` - Lump 2 Offset (Textures)
`ulong` - Lump 3 Offset (Meshes)
`ulong` - Lump 4 Offset (Path)

`ulong` - Lump 0 Element Count
`ulong` - Lump 1 Element Count
`ulong` - Lump 2 Element Count
`ulong` - *unknown* - always 0
`ulong` - Lump 3 Element Count
`ulong` - Lump 4 Element Count

`ulong` - Max Texture count

Lump 0 - Vertices

Vertex positions are, interestingly enough, stored as signed short integer values. Makes sense in terms of space, especially when considering the precision issues that may be run into using a half precision float as an alternative. Does put some practical limits on how small details can be in the world, as well as how large the world can actually be. Doesn't seem to be a problem for the Rage maps, however, which are pretty spacious.

Texcoord's are read in as signed shorts as well, and represent a "packed" texture coordinate. (This is similar to how normal maps store packed vectors in their color components). To get the final texcoord you divide this number by 65,535 and then add 0.5 to yield a fixed precision range from 0.0 to 1.0. This can easily be done in a shader or pre-calculated, depending on your needs.

Element size: 10 bytes

`short` - X
`short` - Y
`short` - Z

`short` - S
`short` - T

Lump 1 - Indices

Indicies are stored as straightforward unsigned shorts, which plays very well with OpenGL ES. Since the range of the shorts is not large enough to index all of the vertices (at least not on any of the existing maps) these values must be used in conjunction with the offsets given in lump 2 to render properly.

Element size: 2 bytes

`ushort` - Index

Lump 2 - Textures

This one is a little deceptive, because while there's only 4 (sensible) values to see here, the index of the object also serves as the index into the texture array. It took a bit of nudging from Carmack for me to realize that the number of elements in this lump is the same as the number of textures in the maps associated texture file! Therefore, if a mesh points to the first element in

this array that mesh also uses texture 0 from the texture file.

Each map has a single associated texture file (ie: for HD_RageLevel1.iosMap textures are stored in HD_RageLevel1.iosTex). The offset for each texture is calculated as: (index * 327680), and the following 327,680 bytes represent a 1024 x 1024 texture stored as 2BPP PVRTC (image data only, no headers). Interestingly enough, this is consistent for both the SD and HD versions of the game. The HD version uses more textures, not larger ones.

The values given here are offsets into the vertex and index buffers that are referenced by the elements by the Meshes (lump 3). The offsets given are in terms of elements, not bytes. The vertex and index count elements aren't needed to render anything, but can be useful for debugging.

Element size: 32 bytes

`ulong` - Vertex Offset
`ulong` - Vertex Count
`ulong` - Index Offset
`ulong` - Index Count
`string[16]` - *unknown*, always 0's

Lump 3 - Meshes

A "mesh" in this case consists of an index into the Mesh Offests (lump 2), a start index relative to the index offset, and an index count. It is worth noting that the same piece of world geometry will typically have multiple different meshes that represent it, each with a different set of texture coordinates to correspond to the various pre-built texture atlases. It would seem sensible to use this same mechanism to manage LOD meshes, but I'm not sure if they do.

Element size: 20 bytes

`ulong` - Texture (lump 2 and texture file) Index
`ulong` - First Index
`ulong` - Index Count
`string[8]` - *unknown*

To render a single mesh, use the following pseudocode (assuming the appropriate index and vertex buffers are already bound):

```
function renderMesh(mesh):
      var texture = this.texture[mesh.texture];
      var indexOffset = texture.indexOffset + mesh.startIndex;
```

```
     glBindTexture(GL_TEXTURE_2D, this.texturemaps[mesh.texture]);

     glVertexAttribPointer(pos_attrib, 3, GL_SHORT, false, 10,
(texture.vertOffset * 10));
     glVertexAttribPointer(tex_atrib, 2, GL_SHORT, false, 10, 6 +
(texture.vertOffset * 10));
     glDrawElements(GL_TRIANGLES, mesh.indexCount,
GL_UNSIGNED_SHORT, indexOffset * 2);
```

Lump 4 - Path

The elements in this lump each represent a point in the path traveled by the player through the level. Each point contains a position and a quaternion that represents the camera orientation, and movement through the level is accomplished by interpolating between these two values on your current point and the upcoming one. The path data does not appear to contain any information about the various stops and turns the player does when facing mutants in game, and it's likely that this behavior is automatic based on enemy position.

(Note: In order to get these values to line up with level geometry in my rendered I had to swap the Y and Z values and invert Y, but that could probably be handled just as easily through matrix manipulation.)

There are 16 textures listed for each point, and contrary to what you might expect they are not the textures needed to render the scene from that point, but instead a list of textures needed for upcoming points (assuming you are moving forward along the path, that is). This is great for streaming, since you can start loading these textures before they are needed and have them resident in memory by the time you reach a point that requires them.

The Visible Mesh List is an array of unsigned shorts that begins at the indicated byte offset. It's a list of indices into the Meshes array (lump 3), and shows you which meshes should be rendered when you are standing at or near the given point.

Element size: 84 bytes

`float` - Position X
`float` - Position Y
`float` - Position Z
`float` - Orientation X
`float` - Orientation Y
`float` - Orientation Z
`float` - Orientation W
`ulong` - Visible Mesh List Offset
`ulong` - Visible Mesh List Element Count

`ulong` - *unknown*, always 0
`ulong` - *unknown*, always 0
`string[8]` - *unknown*
`short[16]` - Upcoming Texture Indices. May contain -1, which means nothing more to load.