Best Practices for Working with Texture Data

Texture data is often the largest portion of the data your app uses to render a frame; textures provide the detail required to present great images to the user. To get the best possible performance out of your app, manage your app's textures carefully. To summarize the guidelines:

- Create your textures when your app is initialized, and never change them in the rendering loop.
- Reduce the amount of memory your textures use.
- Combine smaller textures into a larger texture atlas.
- Use mipmaps to reduce the bandwidth required to fetch texture data.
- Use multitexturing to perform texturing operations in a single pass.

Load Textures During Initialization

Creating and loading textures is an expensive operation. For best results, avoid creating new textures while your app is running. Instead, create and load your texture data during initialization.

After you create a texture, avoid changing it except at the beginning or end of a frame. Currently, all iOS devices use a tile-based deferred renderer, making calls to the `glTexSubImage` and `glCopyTexSubImage` functions particularly expensive. See Tile-Based Deferred Rendering for more information.

Use the GLKit Framework to Load Texture Data

Loading texture data is a fundamental operation that is important to get right. Using the GLKit framework, the `GLKTextureLoader` class makes creating and loading new textures easy. The `GLKTextureLoader` class can load texture data from a variety of sources, including files, URLs, in-memory representations, and CGImages. Regardless of the input source, the `GLKTextureLoader` class creates and loads a new texture from data and returns the texture information as a `GLKTextureInfo` object. Properties of `GLKTextureInfo` objects can be accessed to perform various tasks, including binding the texture to a context and enabling it for drawing.

Note: A `GLKTextureInfo` object does not own the OpenGL ES texture object it describes. You must call the `glDeleteTextures` function to dispose of texture objects when you are done using them.

Listing 9-1 presents a typical strategy to load a new texture from a file and to bind and enable the texture for later use.

Listing 9-1 Loading a two-dimensional texture from a file

```
GLKTextureInfo *spriteTexture;

NSError *theError;


NSString *filePath = [[NSBundle mainBundle] pathForResource:@"Sprite" ofType:@"png"]; // 1


spriteTexture = [GLKTextureLoader textureWithContentsOfFile:filePath options:nil
error:&theError]; // 2

glBindTexture(spriteTexture.target, spriteTexture.name); // 3
```

Here is what the code does, corresponding to the numbered steps in the listing:

1. Create a path to the image that contains the texture data. This path is passed as a parameter to the `GLKTextureLoader` class method `textureWithContentsOfFile:options:error:`.
2. Load a new texture from the image file and store the texture information in a `GLKTextureInfo` object. There are a variety of texture loading options available. For more information, see *GLKTextureLoader Class Reference*.
3. Bind the texture to a context, using the appropriate properties of the `GLKTextureInfo` object as parameters.

The `GLKTextureLoader` class can also load cubemap textures in most common image formats. And, if your app needs to load and create new textures while running, the `GLKTextureLoader` class also provides methods for asynchronous texture loading. See *GLKTextureLoader Class Reference* for more information.

Reduce Texture Memory Usage

Reducing the amount of memory your iOS app uses is always an important part of tuning your app. That said, an OpenGL ES app is also constrained in the total amount of memory it can use to load textures. Where possible, your app should always try to reduce the amount of memory it uses to hold texture data. Reducing the memory used by a texture is almost always at the cost of image quality, so you must balance any changes your app makes to its textures with the quality level of the final rendered frame. For best results, try the techniques described below, and choose the one that provides the best memory savings at an acceptable quality level.

Compress Textures

Texture compression usually provides the best balance of memory savings and quality. OpenGL ES for iOS supports multiple compressed texture formats.

All iOS devices support the the PowerVR Texture Compression (PVRTC) format by implementing the GL_IMG_texture_compression_pvrtc extension. There are two levels of PVRTC compression, 4 bits per pixel and 2 bits per pixel, which offer a 8:1 and 16:1 compression ratio over the uncompressed 32-bit texture format respectively. A compressed PVRTC texture still provides a decent level of quality, particularly at the 4-bit level. For more information on compressing textures into PVRTC format, see Using texturetool to Compress Textures.

OpenGL ES 3.0 also supports the ETC2 and EAC compressed texture formats; however, PVRTC textures are recommended on iOS devices.

Use Lower-Precision Color Formats

If your app cannot use compressed textures, consider using a lower precision pixel format. A texture in RGB565, RGBA5551, or RGBA4444 format uses half the memory of a texture in RGBA8888 format. Use RGBA8888 only when your app needs that level of quality.

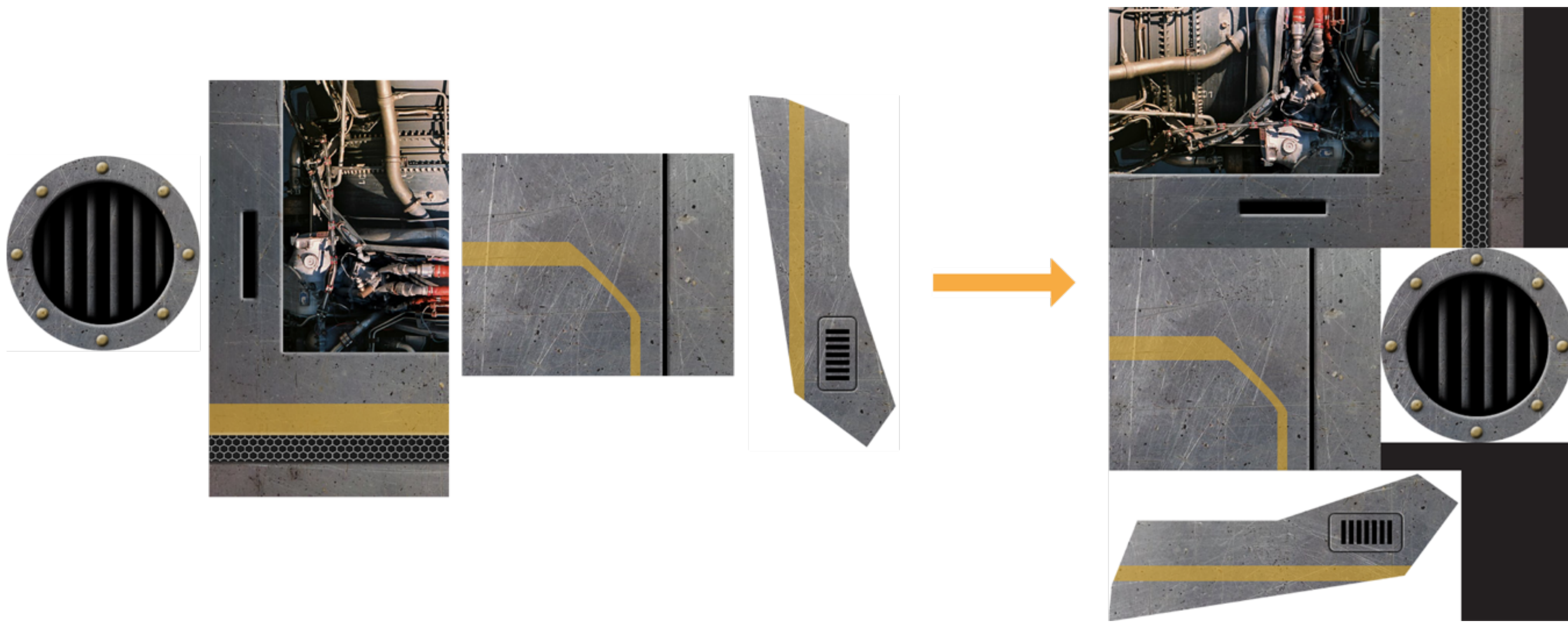
Use Properly Sized Textures

The images that an iOS-based device displays are very small. Your app does not need to provide large textures to present acceptable images to the screen. Halving both dimensions of a texture reduces the amount of memory needed for that texture to one-quarter that of the original texture.

Before shrinking your textures, attempt to compress the texture or use a lower-precision color format first. A texture compressed with the PVRTC format usually provides higher image quality than shrinking the texture—and it uses less memory too!

Combine Textures into Texture Atlases

Binding to a texture takes time for OpenGL ES to process. Apps that reduce the number of changes they make to OpenGL ES state perform better. For textures, one way to avoid binding to new textures is to combine multiple smaller textures into a single large texture, known as a *texture atlas*. When you use a texture atlas, you can bind a single texture and then make multiple drawing calls that use that texture, or even coalesce multiple drawing calls into a single draw call. The texture coordinates provided in your vertex data are modified to select the smaller portion of the texture from within the atlas.



Texture atlases have a few restrictions:

- You cannot use a texture atlas if you are using the `GL_REPEAT` texture wrap parameter.
- Filtering may sometimes fetch texels outside the expected range. To use those textures in a texture atlas, you must place padding between the textures that make up the texture atlas.
- Because the texture atlas is still a texture, it is subject to the OpenGL ES implementation's maximum texture size as well as other texture attributes.

Xcode can automatically build texture atlases for you from a collection of images. For details on creating a texture atlas, see Xcode Help. This feature is provided primarily for developers using the Sprite Kit framework, but any app can make use of the texture atlas files it produces. For each `.atlas` folder in your project, Xcode creates a `.atlasc` folder in your app bundle, containing one or more compiled atlas images and a property list (.plist) file. The property list file describes the individual images that make up the atlas and their locations within the atlas image—you can use this information to calculate appropriate texture coordinates for use in OpenGL ES drawing.

Use Mipmapping to Reduce Memory Bandwidth Usage

Your app should provide mipmaps for all textures except those being used to draw 2D unscaled images. Although mipmaps use additional memory, they prevent texturing artifacts and improve image quality. More importantly, when the smaller mipmaps are sampled, fewer texels are fetched from texture memory which reduces the memory bandwidth needed by the graphics hardware, improving performance.

The `GL_LINEAR_MIPMAP_LINEAR` filter mode provides the best quality when texturing but requires additional texels to be fetched from memory. Your app can trade some image quality for better performance by specifying the `GL_LINEAR_MIPMAP_NEAREST` filter mode instead.

When combining mip maps with texture atlases, use the `TEXTURE_MAX_LEVEL` parameter in OpenGL ES 3.0 to control how your textures are filtered. (This functionality is also available in OpenGL ES 1.1 and 2.0 through the `APPLE_texture_max_level` extension.)

Use Multitexturing Instead of Multiple Passes

Many apps perform multiple passes to draw a model, altering the configuration of the graphics pipeline for each pass. This not only requires additional time to reconfigure the graphics pipeline, but it also requires vertex information to be reprocessed for every pass, and pixel data to be read back from the framebuffer on later passes.

All OpenGL ES implementations on iOS support at least two texture units, and most devices support at least eight. Your app should use these texture units to perform as many steps as possible in your algorithm in each pass. You can retrieve the number of texture units available to your app by calling the `glGetIntegerv` function, passing in `GL_MAX_TEXTURE_UNITS` as the parameter.

If your app requires multiple passes to render a single object:

- Ensure that the position data remains unchanged for every pass.
- On the second and later stage, test for pixels that are on the surface of your model by calling the `glDepthFunc` function with `GL_EQUAL` as the parameter.