SDK White Paper

Improve Batching Using Texture Atlases

WP-01387-001-v01
July 2004

Motivation

Batching, or rather the lack of batching is a common problem for game developers. A batch consists of a number of render-state changes followed by a draw-call. Submitting hundreds or worse, thousands of batches per frame inevitably makes an application CPU-limited due to inherent driver overhead. See [Wloka2003] for a detailed characterization of this problem.

While game developers are aware of and understand this problem, it is nonetheless difficult to avoid: most games require many objects of different characteristics to be displayed, thus they typically require a significant number of render-state changes. Therefore, game developers require practical techniques that allow them to eliminate state-changes and merge batches.

An internal survey of a few recent DirectX9 titles reveals that the following render-state changes occur most frequently: **`SetTexture()`**, **`SetVertexShaderConstantF()`**, **`SetPixelShader()`**, **`SetStreamSource()`**, **`SetVertexDeclaration()`**, and **`SetIndices()`**.

`SetTexture()` is one of the most common batch-breakers. This paper describes a technique for reducing batches caused by having to repeatedly bind different textures, i.e., repeated calls to DirectX9's **`SetTexture()`**.

The technique copies multiple textures into one larger texture: we call this larger texture an atlas (sometimes also called a texture page). Models using these packed atlases need to remap their texture coordinates to access the relevant sub-rectangles out of the texture atlas.

We describe this technique in practical detail in 'How Batching via Texture Atlases Works'. 'Using Atlases in Everyday Life' explains why atlas techniques work, despite common misconceptions. Finally, the section 'Conclusions' sums up our findings.

How Batching via Texture Atlases Works

The most straightforward way to render, say, two textured quads is to bind the texture of the first quad (i.e., call **`SetTexture()`**), draw the first quad (i.e., call **`DrawPrimitive()`**), then bind the texture for the second quad, and finally draw the second quad:

```
SetTexture();
DrawPrimitive();
SetTexture();
DrawPrimitive();
```

This rendering technique requires two batches.

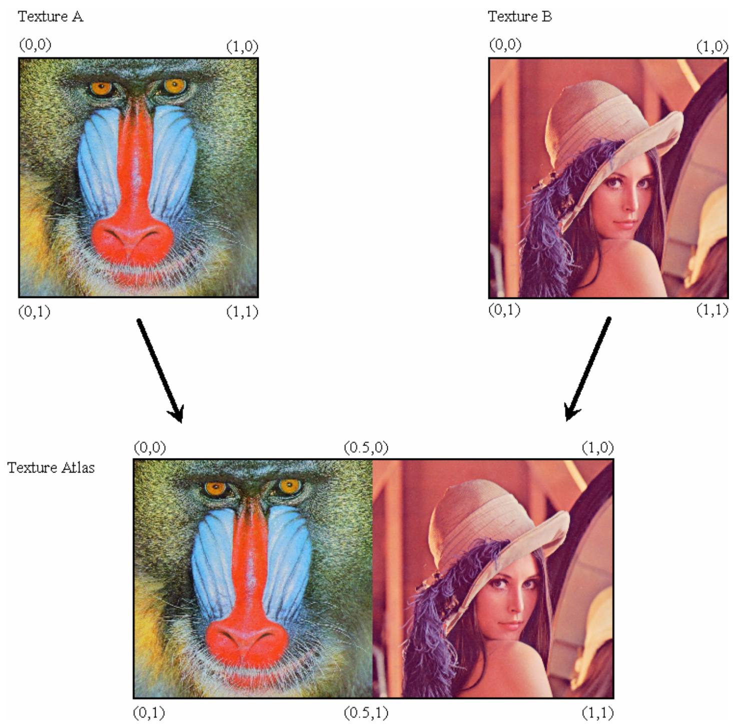

Figure 1: Combining two textures into an atlas. The texture coordinates for accessing data out of an atlas are adjusted according to where the original texture is in the atlas.

If we combine the two textures into a texture atlas, as shown in Figure 1, we no longer need to call **`SetTexture()`** between drawing the two quads, and thus are able to combine the two **`DrawPrimitive()`** calls into one. In other words, we reduce batch-count from two to one:

```
SetTexture();
DrawPrimitive();
```

To access the same texels out of an atlas instead of the original texture, however, one has to modify the texture coordinates of the models referring to that texture. For example, a quad displaying an entire texture uses texture coordinates (0, 0), (0, 1), (1, 0), and (1, 1). In contrast, a quad wanting to show the same texels, but accessed out of an atlas, refers to the atlas's sub-rectangle containing that texture. Figure 1 shows the texture coordinates for the corner texels of textures A and B, as well as the texture coordinates required to access the same information out of their common atlas.

The following sequence of steps thus enables improved batching via texture atlases:

1. Select a collection of textures that are responsible for breaking batches.
2. Pack this texture collection into one or more texture atlases.
3. Update the texture coordinates of all models using any of the textures in that collection to access the appropriate sub-rectangles of an atlas instead.
4. Ensure that sequential **`DrawPrimitive()`** calls that are uninterrupted by state-changes issue as a single **`DrawPrimitive()`** call.

Ideally, steps 1-3 integrate into an existing tool chain, and step 4 is part of the rendering engine.

Selecting a suitable collection of textures for step 1 could be as easy as grouping all textures of the same format into an atlas. To help with steps 2 and 3 NVIDIA provides free tools:

- The Atlas Creation Tool [AtlasCreation2004] is a command-line tool that accepts a collection of textures and packs them into atlases. It also generates a file describing how textures and texture coordinates map from the original texture to a texture atlas. The accompanying user's guide describes its options.
- The Atlas Comparison Viewer [AtlasViewer2004] reads and interprets these mapping-files so as to correctly display textures out of atlases. The Atlas Comparison Viewer also demonstrates the feasibility of texture atlases: it provides a pixel-by-pixel comparison of the results of texturing out of textures versus atlases.

A tool to re-map texture coordinates of general models, i.e., a solution to step 3 above, is not provided: game developers use a variety of model formats and their tool-chains differ, so creating such a general tool is ambitious. But since the Atlas Comparison Viewer performs a similar task (re-mapping of texture coordinates for quads), and since its source code is included, we hope that game developers are able to use the provided source as blueprints for their internal tools.

Using Atlases in Everyday Life

Using Mipmaps with Atlases

Mipmapped textures are essential for achieving rendering performance. Packing mipmapped textures into an atlas, however, seems to imply that the mipmaps of these packed textures combine, until eventually the lowest mip-level of 1x1 resolution smears all textures of an atlas into a single texel. It thus seems that using an atlas with mipmaps creates undesirable image-artifacts.

The truth is that the tool-chain generates (or should generate) the mipmaps for individual textures before these are packed into an atlas. To obtain the highest fidelity results a special-purpose mipmap filter should be used (see for example, [TextureTools2000]).

When packing textures and their mip-chains into an atlas, the textures as well as their mip-chains copy directly into their respective mipmap levels. Because we never combine texels – we just copy them – no smearing or cross-pollution occurs.

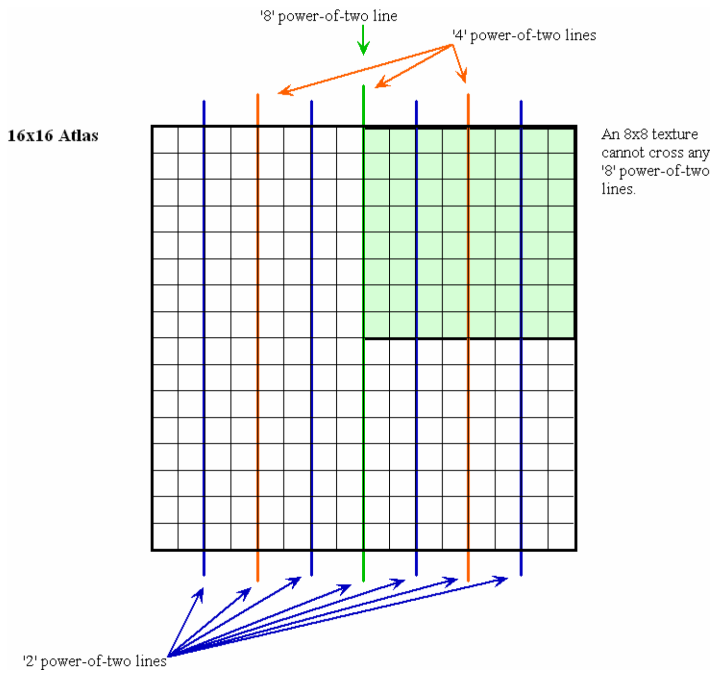

Figure 2: The '8', '4', and '2' power-of-two lines for a 16x16 atlas (note that only vertical power-of-two lines are shown). A sub-texture of dimensions WxH cannot cross 'W' power-of-two lines horizontally nor can it cross 'H' power-of-two lines vertically.

Even when generating a complete mipmap chain for an atlas on the fly, polluting mipmaps with texels from neighboring textures is avoidable: the filter generating the mipmaps should properly clamp at a texture's borders, a requirement also common when generating mipmaps for non-atlas textures.

Finally, even generating mipmap chains of atlases on the fly with a two-by-two box-filter does not pollute mipmaps with neighboring texels if the atlas is a power-of-two texture and contains only power-of-two textures that do not unnecessarily cross power-of-two lines. For example, a 16x16 atlas containing one 8x8 texture and twelve 4x4 textures must ensure that the 8x8 texture is in one of the four corners of the atlas – for example, it cannot be in the center of the atlas (see Figure 2).

As we generate the various mip-levels for such an atlas, texels of separate textures do not combine until the 2x2 level. In the 4x4 atlas-level, the 8x8 texture corresponds to a 2x2 texel block and the 4x4 textures reduce to single texels each. To be able to represent the 8x8 texture with a single texel, we also need to generate the 2x2 level of the atlas. And thus the 2x2 level contains 1 texel representing the 8x8 texture and 3 texels representing the combination of 4 4x4 textures each: pollution occurs (see Figure 3).

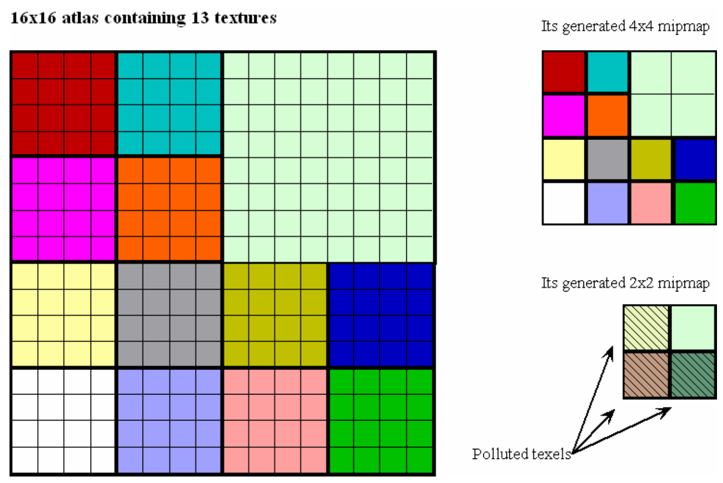

Figure 3: Texture pollution at the 2x2 mip-level for an atlas containing 8x8 and 4x4 textures.

Even when copying mip-chains into an atlas a similar problem occurs: because textures can differ in size and large textures have longer mip-chains than smaller textures, the largest texture packed into an atlas determines the minimum number of mipmap levels in that atlas. The smaller textures thus have effectively longer than necessary mip-chains whose bottom-most levels are uninitialized (see Figure 4).

Solution approaches might be to abridge an atlas's mip-chain to the length of the mip-chain of the smallest texture contained in the atlas – at the expense of performance and image quality. Another approach is to limit only same or similar size textures to pack into the same atlas. Luckily, these measures are uncalled for.

Because models using texture atlases use modified texture coordinates (see above Section 'How Batching via Texture Atlases Works'), a triangle's texture coordinates never span across multiple atlas sub-rectangles containing separate textures. Thus, even when a single triangle spans an entire sub-texture in an atlas, and even if that triangle maps to a single pixel on screen, then only the one-texel representation of that texture is accessed. This one-texel representation is filled with valid non-polluted data (see Figure 5). In other words, in order to be able to access the bottom-most, uninitialized or polluted mip-levels, a sub-texture spanning triangle would have to be smaller than half a pixel. DirectX's rasterization rules make it unlikely that such a triangle generates any pixels. Thus, corruption due to accessing these bottom-most mip-levels does not occur. Using a positive mipmap LOD-bias to artificially blur textures, however, forces access to these bottom-most mip-levels and is thus to be avoided.

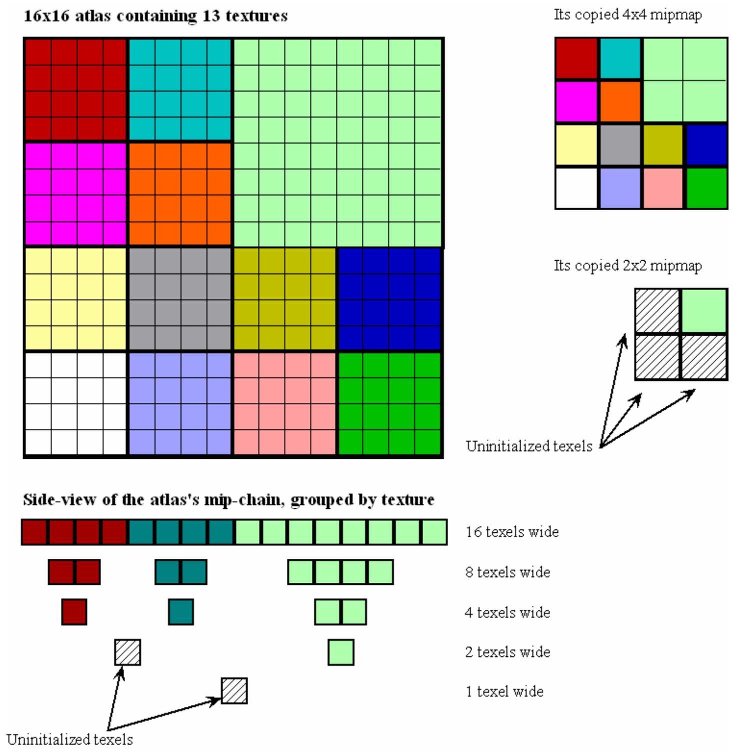

Figure 4: Uninitialized texels at the 2x2 and 1x1 mipmaps for an atlas containing 8x8 and 4x4 textures.

To save video-memory, it is nonetheless good practice to avoid storing completely uninitialized mip-levels. For example, a 1kx1k atlas containing 16 256x256 textures should only store eight mip-levels: the 2x2 and 1x1 levels do not contain relevant data and are superfluous.

The Atlas Creation Tool [AtlasCreation2004] follows these principles and copies a texture's mip-chain into the generated atlas. For textures without complete mip-chains it first generates the complete mip-chain and then copies the data. The uninitialized texels of an atlas contain black. The Atlas Comparison Viewer [AtlasViewer2004] shows no visible artifacts even at extreme viewing angles that force access to the lowest mipmaps.

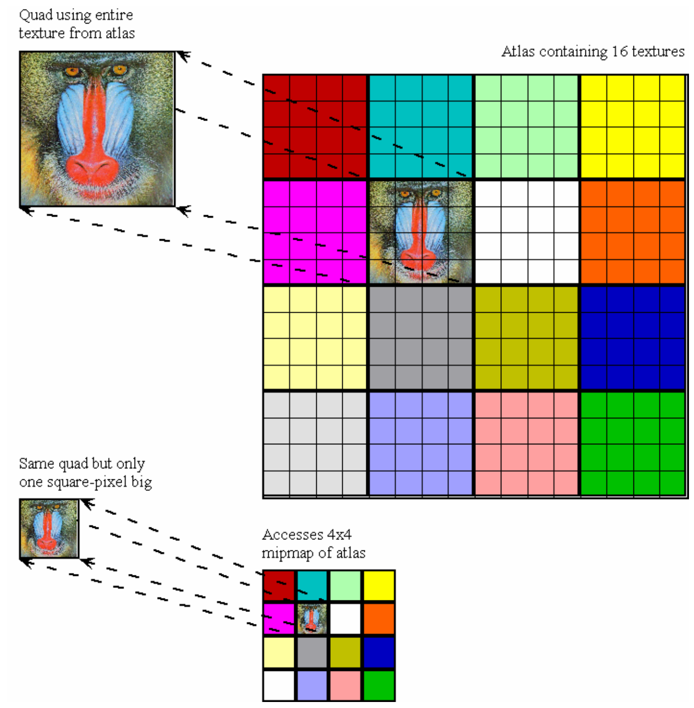

Figure 5: Because all models have remapped texture coordinates to only access an atlas's sub-texture, even single-pixel quads that access an entire sub-texture only access valid, initialized, and non-polluted texels.

Using Clamp, Wrap, or Mirror Modes with Atlases

GPUs provide different address modes for when texture coordinates are outside the zero to one range. In clamp mode, coordinates outside the [0, 1] interval clamp to either zero or one. The visual effect of this mode is that a texture's border texels repeat indefinitely. Wrap mode discards the integer part of a texture coordinate and just relies on the fractional part to address a texture. Mirror mode repeatedly mirrors the texture image for texture coordinates outside the [0, 1] interval. Figure 6 illustrates these different texture addressing modes.

To access a texture packed into the center of an atlas one uses texture coordinates that are a strict subset of [0,1], thus a GPU's address modes never apply. Worse, remapping texture coordinates outside of the [0, 1] range, i.e., texture coordinates making use of address modes, results in atlas coordinates that access neighboring textures in the atlas (see Figure 7).

A possible workaround is to replicate the same texture multiple times into an atlas. For example, if a texture wraps up to five times, then this texture copies five times into an atlas. This technique wastes large amounts of texture memory, especially when address modes in the u- and v-dimensions are used simultaneously. It also complicates the atlas-packing algorithm, as it now requires usage information about the textures: what address modes are particular textures using and what are their minimum and maximum texture coordinates in use?

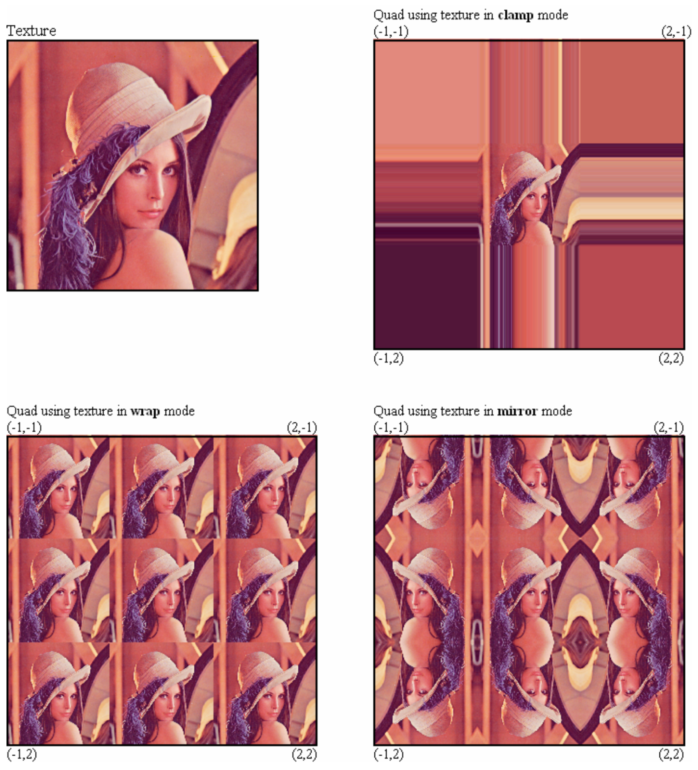
Texture
Quad using texture in clamp mode
(-1,-1)
(2,-1)
(-1,2)
(2,2)
Quad using texture in wrap mode
(-1,-1)
(2,-1)
(-1,2)
(2,2)
Quad using texture in mirror mode
(-1,-1)
(2,-1)
(-1,2)
(2,2)

Figure 6: Clamp, wrap, and mirror address modes.

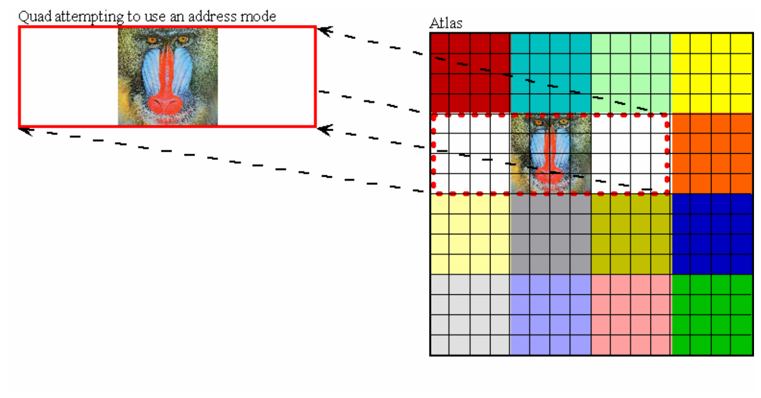

Figure 7: Original texture coordinates outside the [0, 1] range, i.e., coordinates indicating the use of an address mode, map to atlas coordinates that access texels outside the intended sub-texture.

Fortunately, replicating textures multiple times into an atlas is unnecessary. The preferred solution is to tessellate models so that their texture coordinates are always in the [0, 1] range. These resulting additional vertices typically do not incur a performance penalty as modern GPUs are rarely vertex-processing bound.

Another solution is to emulate these address modes with a pixel shader; the next section provides the details.

Emulating Clamp, Wrap, and Mirror Addressing with Pixel Shaders

The Atlas Comparison Viewer [AtlasViewer2004] implements clamp, wrap, and mirror addressing for textures that are part of an atlas. It uses pixel shaders to emulate these addressing modes.

For example, to implement clamping, a shader modifies the texture coordinates used to access an atlas. Instead of using the incoming texture coordinates directly to access the atlas, the shader first clamps these coordinates to the maximum and minimum values allowed for the given sub-rectangle that contains the relevant texture in the atlas.

For wrapping and mirroring, shaders similarly transform the incoming texture coordinates to emulate the respective address modes. In particular for wrapping, care has to be taken to avoid skewing which mip-level is used for pixels at the wrap borders. Using the newly wrapped texture coordinates directly via HLSL's **`tex2D(s, t)`** (assembly's **`texld`**) call is inadvisable: **`tex2D(s, t)`** computes which mip-level to use from the supplied texture coordinates. Pixels displaying texels at the wrapping borders have texture coordinates that are discontinuous, i.e., jump from say 0.99 to 0.0 over the span of 1 pixel. The texture-coordinate derivatives are thus very large, forcing use of lower mip-levels, and thus producing wrong results (see Section 'Using Mipmaps with Atlases' above).

Instead, we avoid altering which mip-level is accessed. The HLSL call **`tex2D(s, t, ddx, ddy)`** (assembly **`texldd`**) instructs GPUs to use particular derivatives to decide which mip-level to use. We thus compute the arguments **`ddx`** and **`ddy`** in the shader from the original texture coordinates using HLSL's **`ddx()`** and **`ddy()`** (assembly **`dsx`**, **`dsy`**) instructions and then use the results to access the atlas. See also the source code for the Atlas Comparison Viewer [AtlasViewer2004], more specifically the Wrap.ps pixel shader for how this technique works.

Emulating address modes in the pixel shader as described has drawbacks compared to tessellating models to enforce texture coordinates in the [0, 1] range (see Section 'Using Clamp, Wrap, or Mirror Modes with Atlases's above). First, emulation requires use of the pixel-shader assembly instructions **`dsx`**, **`dsy`**, which are only supported by pixel shader profiles ps_2_a and later. Thus, it excludes GPUs that only support ps_2_0 or ps_2_b profiles. Second, selecting which sub-rectangle of an atlas to clamp, wrap, or mirror to has to be encoded in the vertex stream and passed to the pixel-shader. Simply changing a pixel-shader constant as demonstrated in the Atlas Comparison Viewer [AtlasViewer2004] is unacceptable as it would break the batch. Thus, additional software work is required to integrate this feature into an existing engine. Third and finally, modifying texture coordinates in the pixel shader costs pixel shader performance: a concern since today's games are more likely pixel shader bound than vertex shader bound.

Using Coordinates in the Zero to One Range

DirectX defines texture coordinates zero and one to coincide, i.e., a vertex with texture coordinates (0, 0) and a vertex with texture coordinates (1, 1) both access the identical texel (irrespective of filtering mode) when using the wrap texture addressing mode. To access all texels of a texture of dimensions width by height once and only once, models need to reduce the addressable range of the texture by using u-coordinates in the range

$$\left[\frac{1}{2\,width}, 1-\frac{1}{2\,width}\right] \tag{1}$$

and v-coordinates in the range

$$\left[\frac{1}{2\,height}, 1-\frac{1}{2\,height}\right]. \tag{2}$$

Most applications, however, use texture coordinates ranging from zero to one inclusive, nonetheless. While such coordinates actually invoke wrap, clamp, or mirror address modes, the benefit of being texture-dimension independent outweighs the slight image-quality reduction.

Because texture coordinates in the inclusive [0, 1] interval thus address an area larger than the actual texture, directly re-mapping these coordinates to atlas coordinates also accesses an area larger than the texture's assigned sub-rectangle. The Atlas Creation Tool [CreationTool2004] offers several solutions to this problem.

The first option is to use the Atlas Creation Tool's default setting. In that case, the Atlas Creation Tool maps the coordinates directly. If the original texture coordinates are in the range specified by equations **(1)** and **(2)**, then the atlas coordinates correctly access only texels of the original texture. The Atlas Comparison Viewer's [ComparisonViewer2004] display mode '**Original Adjusted, Atlas Adjusted**' demonstrates the resulting image quality.

If the original texture coordinates, however, range from zero to one inclusive, then, yes, the atlas coordinates do access texels of neighboring textures. The resulting image artifacts are, however, minimal as the Atlas Comparison Viewer demonstrates via its '**Original NOT adjusted, atlas NOT adjusted**' display mode.

A better solution is to specify the Atlas Creation Tool's '**`–halftexel`**' option. It instructs the tool to rescale all texture coordinates to fit into the range specified by equations **(1)** and **(2)**. The corresponding Atlas Comparison Viewer's display mode '**Original NOT adjusted, atlas adjusted**' thus shows this scaling in the difference view. If the intent of specifying zero to one inclusive coordinates is to refer to an entire texture and no more, while maintaining texture-dimension independence, then integrating the Atlas Creation Tool into the tool-chain realizes both intents.

Applying the previous section's pixel-shader to fix the one-texel wrapping, clamping, or mirroring is another possible solution. This solution, however, is heavy-weight and unnecessary as the Atlas Comparison Viewer's display mode '**Original NOT adjusted, atlas NOT adjusted**' demonstrates.

Applying Texture Filtering To Atlases

Specifying coordinates in the range of equations **(1)** and **(2)** (see previous section) samples texels at their exact center. Sampling a texel at its center means that only that texel contributes to the filtered output, even when bilinear filtering is enabled. Conversely, sampling a texel off-center and bilinearly filtering it, results in adjacent texels to contribute. This behavior, however, is to be avoided for texels lining the border of sub-textures in an atlas, as they are in danger of pulling in texels from unrelated textures.

While bilinear filtering of the highest resolution mip-level is thus safe, anisotropic filtering of the same mip-level does potentially access unrelated neighboring texels. Worse, bilinear or anisotropic filtering of all lower mip-maps also access unrelated neighboring texels, as Figure 8 demonstrates.

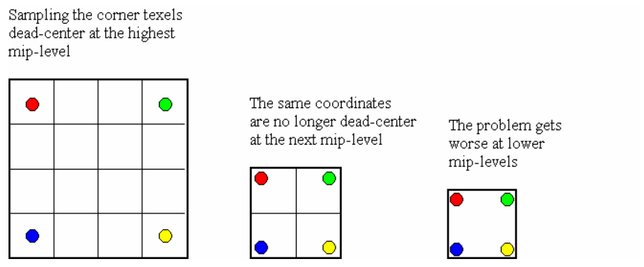
Sampling the corner texels dead-center at the highest mip-level
The same coordinates are no longer dead-center at the next mip-level
The problem gets worse at lower mip-levels

Figure 8: Bilinear filtering of lower mip-levels accesses texels from unrelated neighboring textures.

Unfortunately, these artifacts are not easily overcome. While their overall effect on image quality is small, they are nonetheless noticeable (see Atlas Comparison Viewer [ComparisonViewer2004]). Experimentation with the Atlas Comparison Viewer shows that enabling anisotropic filtering actually minimizes these errors. That behavior seems counter-intuitive since an anisotropic filter penetrates deeper into a bordering texture than a bilinear filter. Anisotropic filters, however, have by definition narrower footprints than bilinear filters, and thus fewer unrelated texels enter the equation.

Adding border texels to a texture is a possible solution to reducing artifacts due to texture filtering. For example, to add an n-pixel border to a width x height texture, one would rescale the original texture to dimensions width-2n x height-2n, place this rescaled texture at the center of the new texture, and extend the scaled texture's border texels to the borders of the new texture. Rescaling textures is necessary to maintain, for example, power-of-two restrictions on a texture's dimensions.

Placing textures with similar hues, similar border texels, or similar mipmaps into a common atlas is another way to minimize texture-filtering artifacts.

Using Volume Textures as Atlases

Volume textures are seemingly perfect for storing multiple textures: each slice of a volume texture stores exactly one original texture. To access different textures one only varies the third, i.e., w, texture-coordinate.

Clamp, wrap, or mirror address modes work correctly for the u- and v-dimensions of each slice of the volume texture even without pixel-shader emulation, as long as all textures stored in slices are of the same dimensions. If a texture is smaller than the dimension of a slice, then texture memory is wasted and clamp, wrap, or mirror only work correctly if the slice's empty space duplicates texel data according to the desired address mode.

Unfortunately, mipmaps of volume textures reduce in size in all dimensions, e.g., a 4x4x4 volume texture has mipmaps of dimension 2x2x2 and 1x1x1. Thus, storing mipmapped textures in a volume texture proves impossible, as there is not enough space available in a volume texture's mip-chain.

Volume textures are nonetheless useful as texture atlases for textures guaranteed to not need mipmaps, such as 2D user-interface textures. 2D user-interface textures are always screen-aligned and maintain the same distance from the camera, i.e., they do not minify. Thus, if they have a mip-chain, a GPU never accesses it; storing a mip-chain for these textures is superfluous. These textures are therefore ideal for storing into a non-mipmapped volume texture for batching purposes. To avoid accessing data from neighboring volume slices, care has to be taken to only sample at the center of w-slices when bilinear filtering is on. The Atlas Creation Tool [AtlasCreation2004] supports volume textures via the '**`-volume`**' option.

Conclusions

Texture atlases are not a new technique; many games use them for specialized situations, e.g., rendering text or sprite animations. Some games even use them as described here.

As GPUs continue to follow Moore's Law squared [Wloka2003] and GPUs thus become comparatively faster than CPUs, it is important for game developers to aggressively reduce batches. For a CPU-limited game fewer batches means higher frame-rates or more eye-candy, physics, and AI CPU-computations.

The texture atlases technique is one tool that can reduce batch-counts. While texture atlases have the stigma of producing lower image quality, the Atlas Comparison Viewer [AtlasViewer2004] demonstrates this to be largely a misconception. This paper explains how to use texture atlases and how to avoid common pitfalls. Where appropriate we point out potential performance, visual quality, and programming costs associated with atlases. Since the Atlas Comparison Viewer [AtlasViewer2004] and the Atlas Creation Tool [AtlasCreation2004] are freely available (source code inclusive), we hope game developers take a second look at texture atlases as a technique to be integrated into their tool chains.

Bibliography

[AtlasCreation2004] "*Atlas Creation Tool User Guide*," NVSDK 7.0, March'04.

[AtlasViewer2004] "*Atlas Comparison Viewer User Guide*," NVSDK 7.0, March'04.

[TextureTools2000] "*Texture Tools User Guide*," NVSDK 7.0, March'04.

[Wloka2003] "*Batch, Batch, Batch: What Does It Really Mean*," Matthias Wloka, GDC 2003, San Jose, CA.
http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.ppt

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com