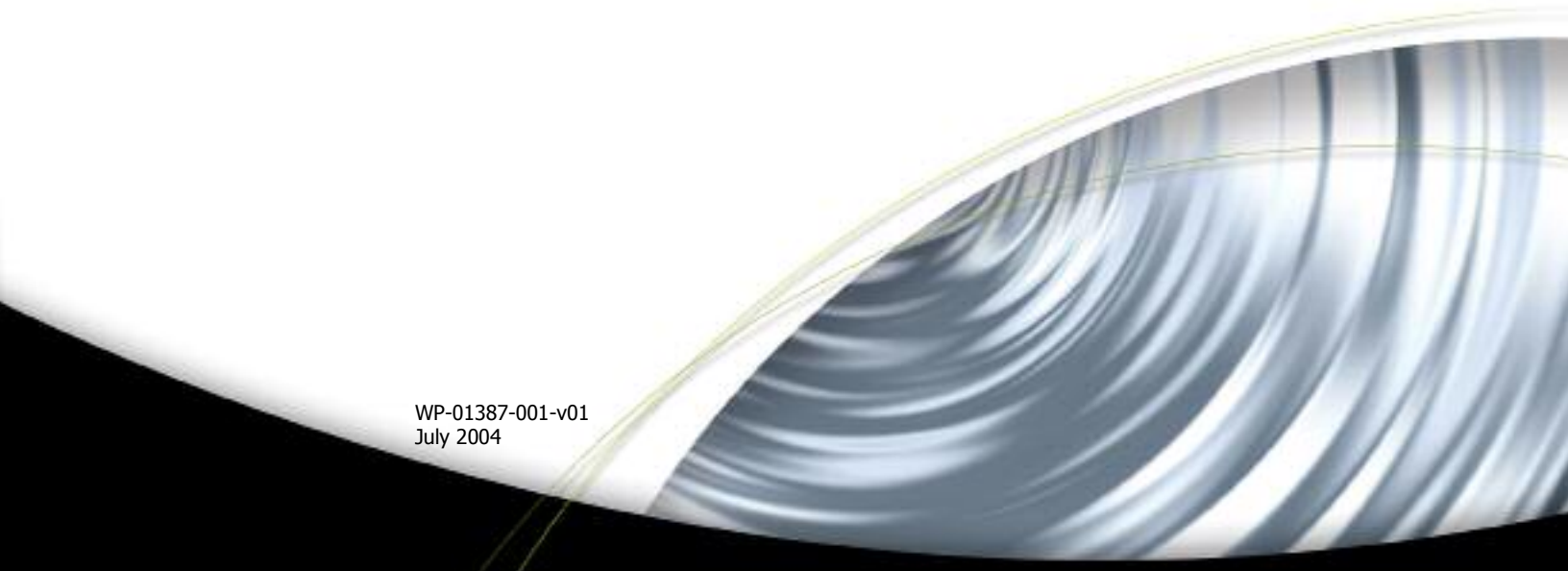
SDK White Paper

Improve Batching Using Texture Atlases

WP-01387-001-v01
July 2004



nVSDK

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, since ultimately game developers want to display many objects that all have different characteristics, and thus typically require render-state changes: game developers therefore require practical techniques that allow them to eliminate state-changes and merge batches.

An internal survey of four DirectX9 titles (some of them still under development) reveals that the following render-state changes occur most frequently:

- **`SetTexture()`**
- **`SetVertexShaderConstantF()`**
- **`SetPixelShader()`**
- **`SetStreamSource()`**
- **`SetVertexDeclaration()`**
- **`SetIndices()`**

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

The technique described here copies multiple textures into one larger texture. This larger texture is referred to as an *atlas*. Models using these packed textures need to remap their texture coordinates to access the relevant sub-rectangles out of the atlas-texture.

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 (i.e., call **`SetTexture()`** again), and finally draw the second quad (i.e., call **`DrawPrimitive()`** again). This rendering technique requires two batches.

If the two textures are combined into a texture atlas, as shown in Figure 1, you 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, reduce batch-count from two to one.

To access the same texels out of an atlas instead of out of the original texture, however, one has to modify the texture coordinates of the models referring to that texture. For example, a quad that displays an entire texture uses the texture coordinates of the four corner texels, i.e., **`uv-coordinates (0, 0), (0, 1), (1, 0),`** and **`(1, 1)`**. A quad wanting to show the same texels, but accessed out of an atlas, refers to the atlas 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 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 **`uv-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-change calls issue as single **`DrawPrimitive()`** calls.

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 the following free tools:

- **Atlas Creation Tool [AtlasCreation2004]**
  This tool 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 current options.
- **Atlas Comparison Viewer [AtlasViewer2004]**
  This tool 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.

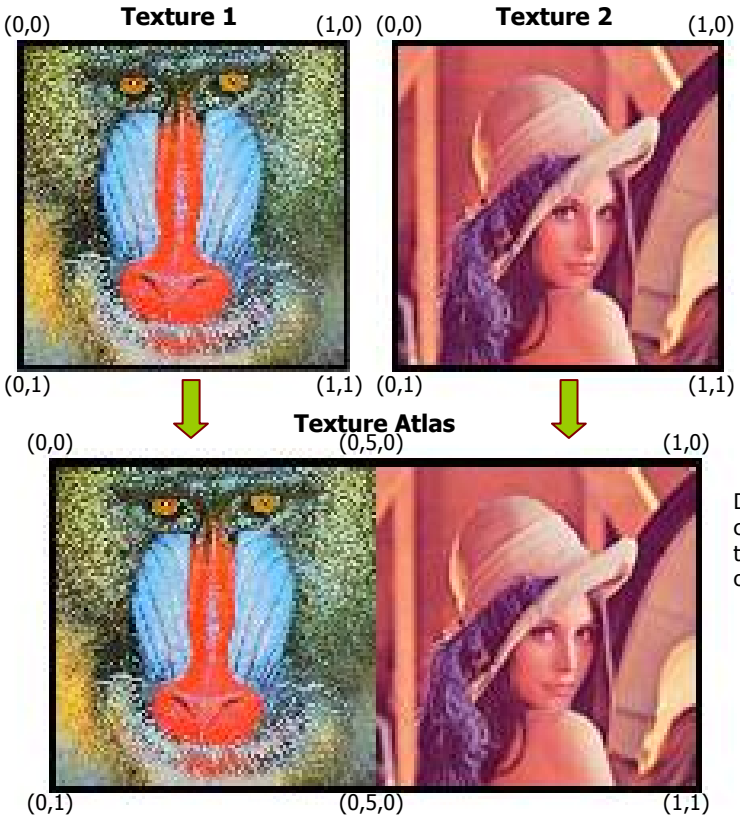


Figure 1. Combining Two Textures into a Single Atlas

NVIDIA currently does not provide a tool to re-map texture coordinates of general models since game-developers use a variety of model-formats and their tool-chains differ. Therefore, to create such a general tool is ambitious. Since the Atlas Comparison Viewer, however, performs a similar task (re-mapping of texture coordinates for quads), and since its source code is freely available, game-developers could use the provided source as blueprints for their internal tools.

Using Mip-Maps with Atlases

Mip-mapped textures are essential for achieving any kind of rendering performance. Packing mip-mapped textures into an atlas, however, seems to imply that the mip-maps 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 therefore seems that using an atlas with mip-maps creates horrible artifacts as soon as the graphics processing unit (GPU) accesses them.

Truth is that the tool-chain generates the mip-maps for individual textures before these are packed into an atlas. To obtain the highest fidelity results, a special-purpose mip-map filter should be used (such as NVIDIA's texture tools [TextureTools2000]).

When packing textures and their mip-chains into an atlas, the textures, as well as their mip-chains, copy directly into their respective mip-map levels. Because texels are never combined—just copied—no smearing or cross-pollution occurs.

Even if one were to generate a complete mip-map chain for an atlas on the fly, polluting mip-maps with texels from neighboring textures is avoidable: the filter generating the mip-maps should properly clamp at a texture's borders, a requirement also common when generating mip-maps for non-atlas textures.

Finally, even generating mip-map chains of atlases on the fly with a two-by-two box-filter does not pollute mip-maps 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 32x32 atlas containing one 16x16 texture and twelve 8x8 textures must ensure that the 16x16 texture is in one of the four corners of the atlas—it cannot be in the dead-center of the atlas (see Figure 2).

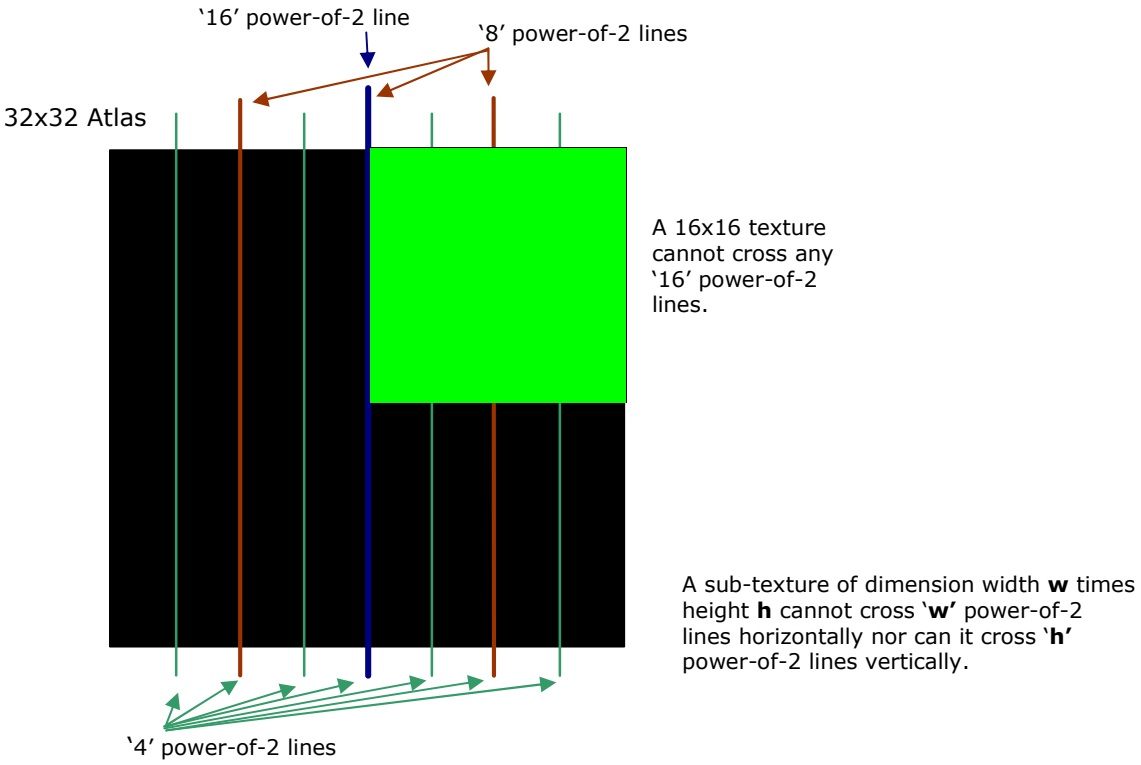


Figure 2. '16', '8', and '4' Power-of-2 Lines for a 32x32 Atlas

As the various mip-levels are generated, texels of separate textures do not combine. That is until the 2x2 level.

In the 4x4 level, the 8x8 textures reduce to single texels each and a 2x2 texel block represents the 16x16 texture. To be able to represent the 16x16 texture with a single texel, you need to also generate the 2x2 level. And thus the 2x2 level contains one texel representing the 16x16 texture and three texels representing the combination of four 8x8 textures each: Pollution occurs (see Figure 3).

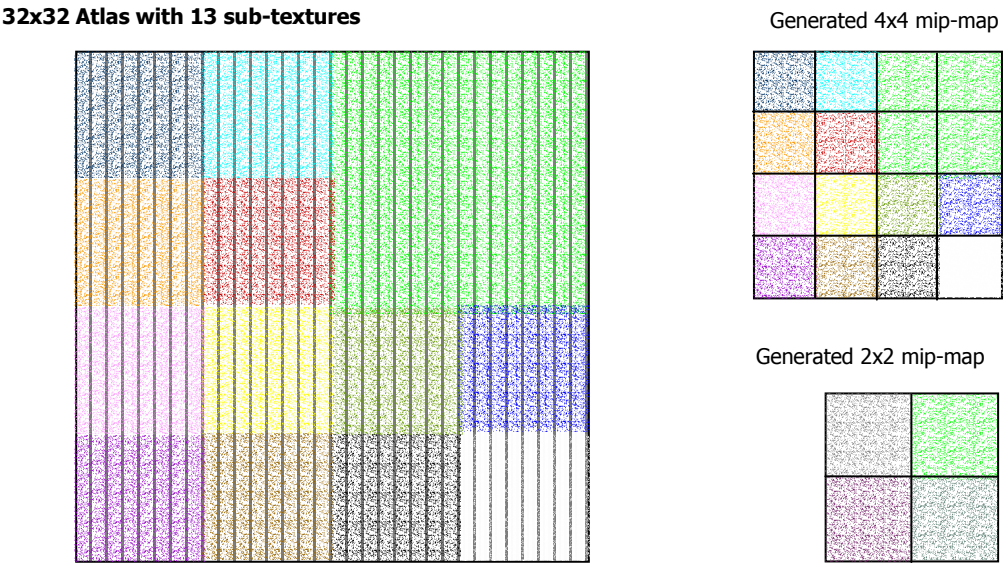


Figure 3. Texture pollution at the 2x2 Mip-map level for an Atlas Containing 16x16 and 8x8 Sub-textures

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 number of mip-map 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. However, that would typically have severe performance and image quality implications. Another approach would be to limit only same or similar size textures to pack into the same atlas. Luckily, these measures are uncalled for.

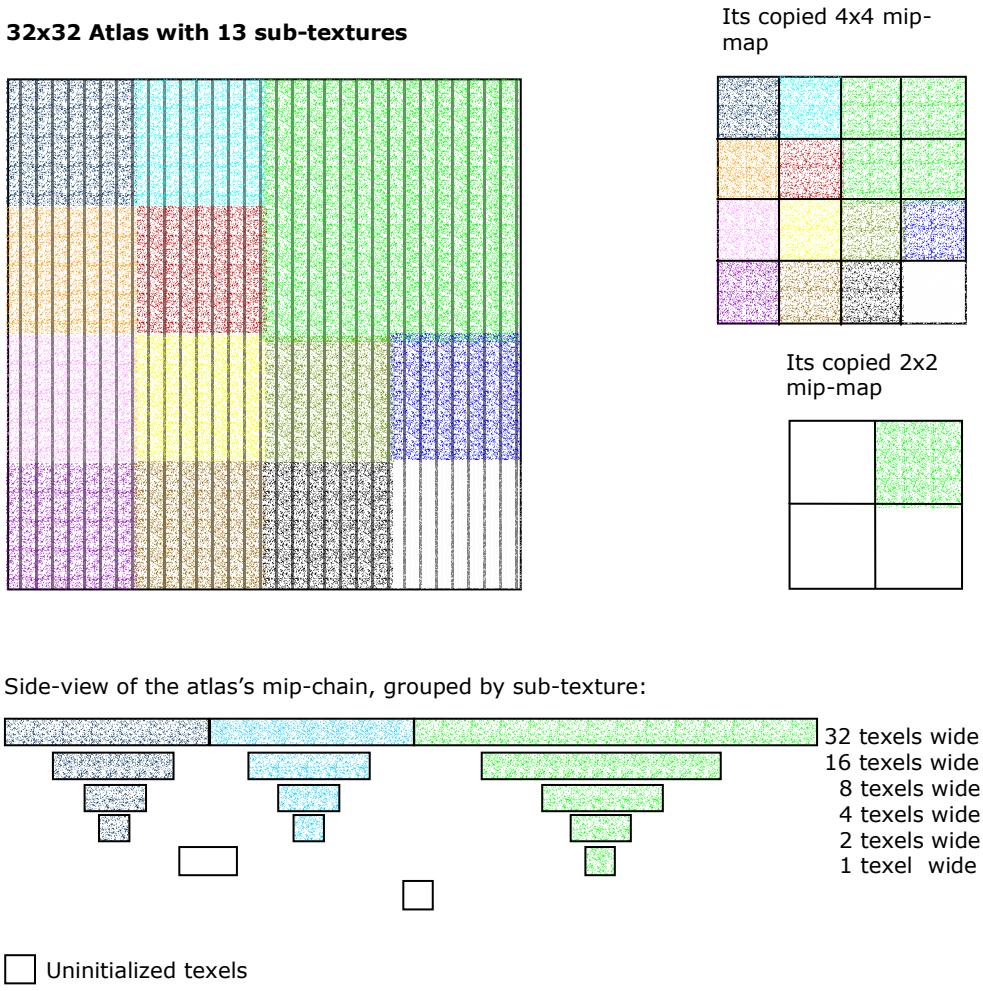


Figure 4. Uninitialized Texels at the 2x2 and 1x1 Mip-maps for an Atlas Containing 16x16 and 8x8 Sub-textures

Since models using texture atlases use modified texture coordinates (see *Batching Using Texture Atlases* on page 2), a triangle's texture-coordinates never span across multiple atlas sub-rectangles containing separate textures. Thus, even when a 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 small triangle generates any pixels. Thus, corruption due to accessing these bottom-most mip-levels does not occur.

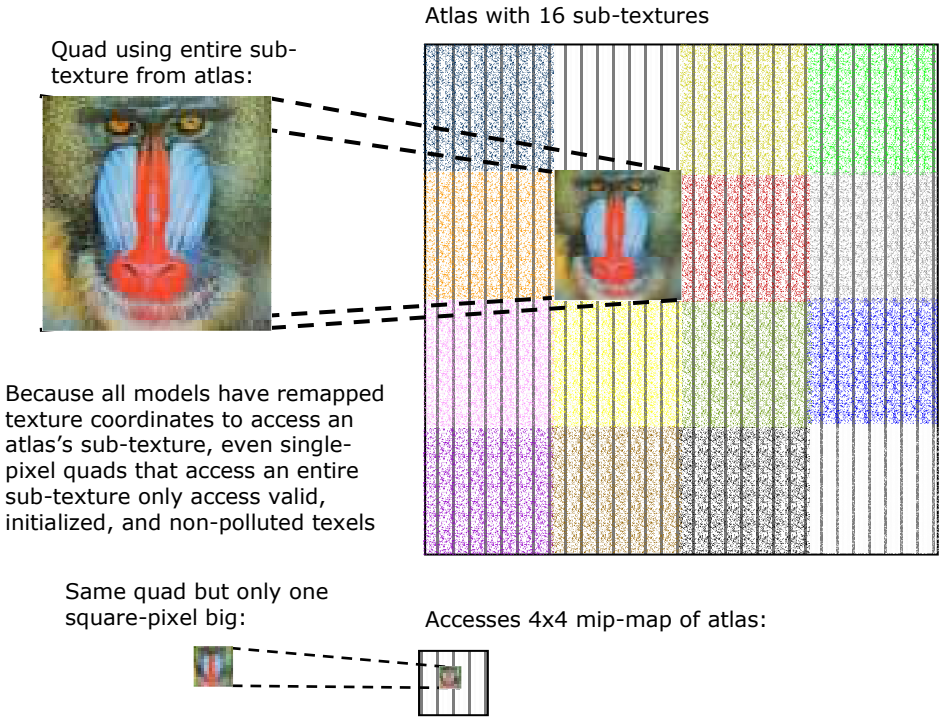


Figure 5. Mechanics of Accessing Atlas Textures.

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 8 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 mip-maps.

Using Clamp, Wrap, or Mirror Modes with Atlases

GPUs provide different address modes for when texture coordinates are outside the zero to one range. If 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 (see Figure 6). Wrap mode discards the integer part of a texture coordinate and just relies on the fractional part to address a texture (see Figure 5). Mirror mode repeatedly mirrors the texture image for texture coordinates outside the [0, 1] interval (see Figure 6).

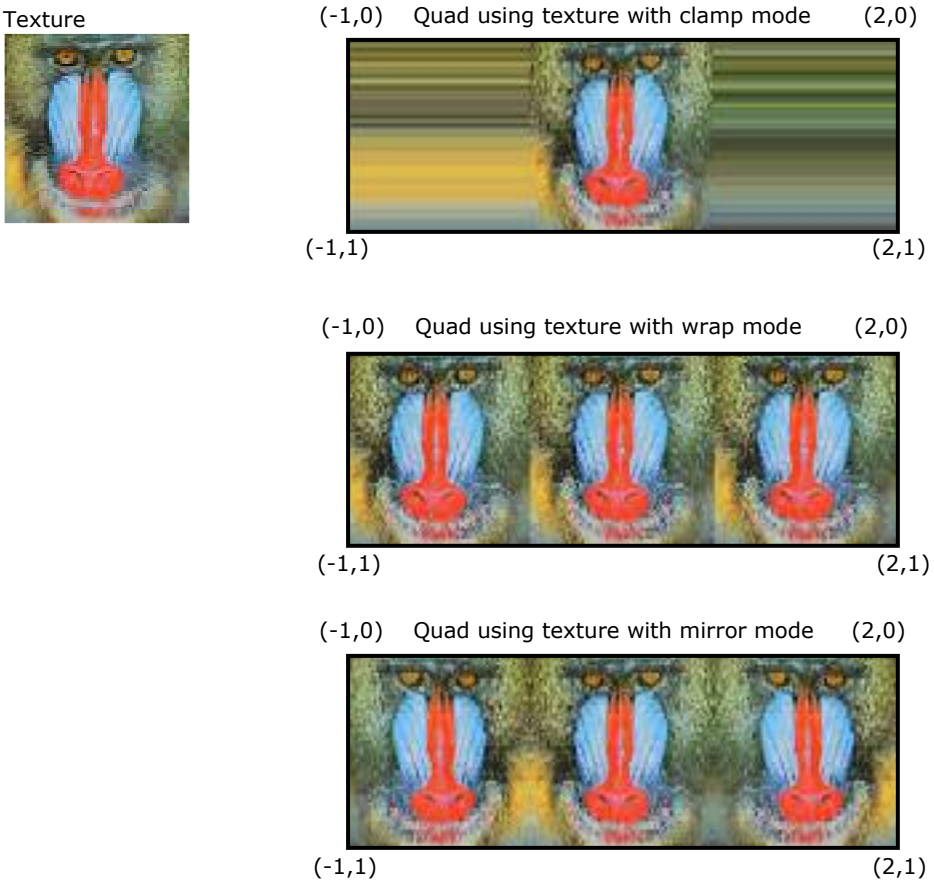


Figure 6. Clamp, Wrap, and Mirror Address-modes.

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).

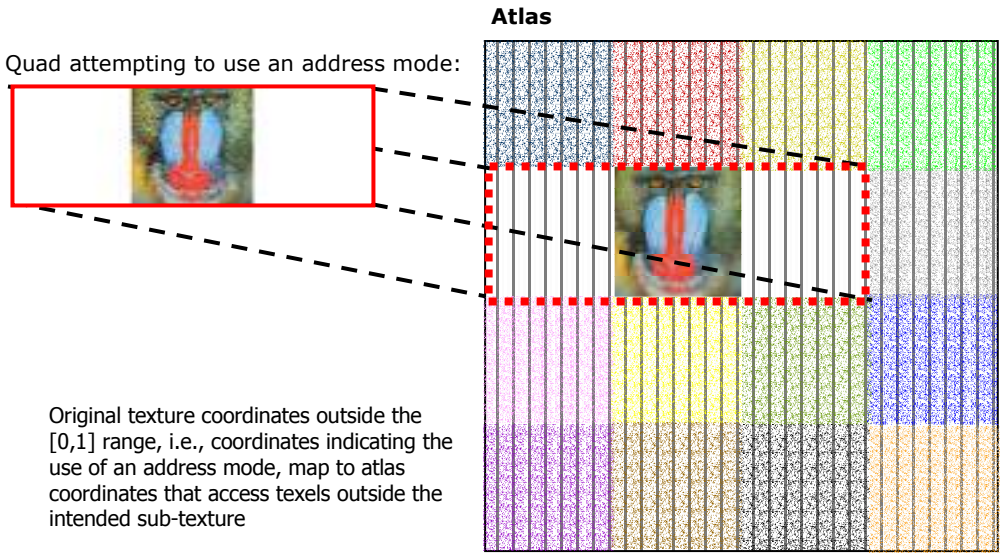


Figure 7. Atlas Coordinates that Access Neighboring Textures in the Atlas

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 **u-** and **v-address** modes are used simultaneously. It also complicates the atlas packing algorithms, as they now require usage information about the textures: which address modes is a particular texture using and what are the maximum and minimum texture coordinates used for that texture?

Fortunately, replicating textures multiple times into an atlas is unnecessary. Pixel-shaders are able to emulate these address modes. The Atlas Comparison Viewer [AtlasViewer2004] implements clamping, wrapping, and mirroring of textures that are part of an atlas.

To implement these GPU address modes in a pixel-shader requires pixel-shader version 2.a; more specifically the **ddx** and **ddy** pixel-shader instructions.

Using Coordinates in the Zero-to-One Range

DirectX defines texture coordinates 0 and 1 to coincide, i.e., a vertex with texture coordinates (0, 0) and a vertex with texture coordinates (1, 1) both access the exact same texel. To access all texels of a texture of dimensions width by height once and only once, models need to use u-coordinates in the range [.5/width, 1-.5/width] and v-coordinates in the range [.5/height, 1-.5/height].

Most applications, however, use texture coordinates ranging from zero to one, inclusive, nonetheless. While such coordinates actually wrap, clamp, or mirror a texture by exactly one texel, 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 then the texture's assigned sub-rectangle. The Atlas Creation Tool [CreationTool2004] offers several solutions to this problem.

The first option is to use the default setting in the Atlas Creation Tool. In this case, the Atlas Creation Tool maps the coordinates directly. If the original texture coordinates range from .5/texture-dimension to 1-.5/texture-dimension, 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 0 to 1 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, however, is to specify the Atlas Creation Tool's –**halftexel** option if the original texture coordinates range from 0 to 1 inclusive. It instructs the tool to rescale the texture coordinates to only range from .5/texture-dimension to 1-.5/texture-dimension. 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 0 to 1 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 largely 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 [`.5/texture-dimension, 1-.5/texture-dimension`] range (see previous section) samples a texture's texel's dead-center. Sampling a texel at its center means that even when bilinear filtering is enabled that only that texel contributes to the filtered output. Conversely, sampling a texel off-center and bilinearly filtering it, results in other texels contributing to the filtered output; behavior to be avoided, however, for texels lining the border of sub-textures in an atlas, as they are in danger of pulling in texels from foreign 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 and anisotropic filtering of all lower mip-maps also access unrelated neighboring texels, as Figure 8 demonstrates.

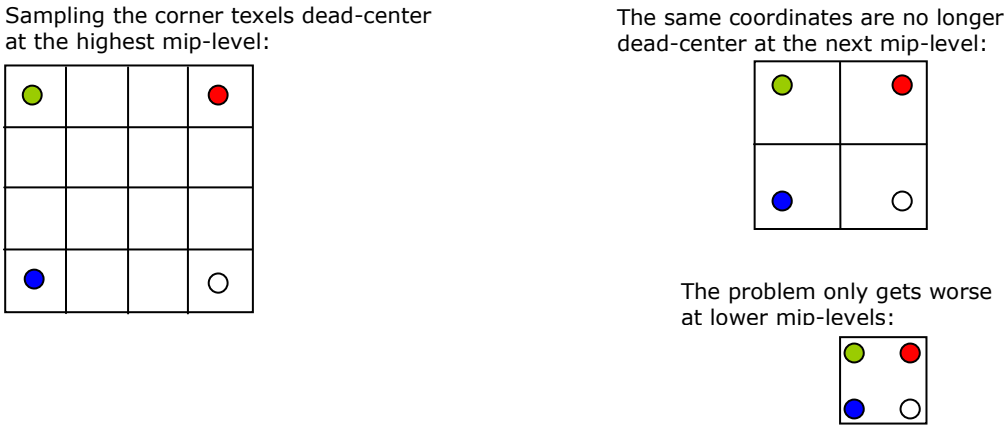


Figure 8. Bilinear Filtering of Lower Mip-levels Accesses Texels from Unrelated Neighboring Textures

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

There are two possible solutions that potentially eliminate access of foreign texels due to texture filtering. These solutions have not yet been implemented, so the following discussion is only theory.

The first solution is to write a pixel-shader to clamp atlas coordinates to corresponding sub-textures taking into account which mip-level the texture operation is about to access. For the highest mip-level the atlas coordinates remain unchanged, yet for lower mip-levels the atlas coordinates are remapped closer to the center, until at the 1x1 mip-map all coordinate map to the center of the texture. This technique requires ps.2.a pixel-shader support and a comparatively complex and thus expensive shader.

The second solution is to pad textures and their mip-chains with border texels. For example, surround the 1x1 mip-map with identical texels so that it consumes a total of 3x3 texels space. These border texels of lower mip-maps, however, need to map up to the higher mip-levels. In the above example, the padded 3x3 mip-map is the lowest mip-level. The one above it would be of dimension 6x6 for a 2x2 block of data. This scheme quickly wastes texture-memory.

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 w-coordinate. As long as filtering in the w-dimension is set to point-filtering, textures stored in different slices do not influence one another.

Address modes such as clamp, wrap, or mirror, 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 size. 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, mip-maps of volume textures reduce in size in all dimensions, e.g., a 4x4x4 volume texture has mip-maps of dimension 2x2x2 and 1x1x1. Thus, storing mip-mapped textures in a volume-texture proves impossible, as there is not enough memory available in a volume texture's mip-chain.

Volume textures are nonetheless useful as texture atlases for textures guaranteed to not need mip-maps, such as 2D user-interface textures. 2D user-interface textures are always screen-aligned and stay at the same distance from the camera, i.e., they do not minify. Thus, if they have a mip-chain, a GPU never accesses it regardless. Storing a mip-chain for these textures is thus superfluous, and these textures are therefore ideal for storing into a non-mip-mapped volume texture for batching purposes.

Conclusions

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

As GPUs continue to follow Moore's Law squared [Wloka2003] and get comparatively faster than CPUs, it is important for game developers to aggressively reduce batches. The fewer batches a game uses the higher its frame-rate, or rather the more eye-candy, physics, or AI computations are available to the game. Texture atlases promise to be one tool that allows reduced batch-counts.

While texture atlases have a perceived stigma of producing lower image quality, the Atlas Comparison Viewer demonstrates this to be a misconception. This paper explains how to use texture atlases and how to avoid common pitfalls. Since the Atlas Comparison Viewer and the Atlas Creation Tool come with source code, we hope that 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

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2004 NVIDIA Corporation. All rights reserved

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