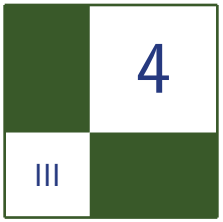

Virtual Texture Mapping 101

Matthäus G. Chajdas, Christian Eisenacher,
Marc Stamminger, and Sylvain Lefebvre

4.1 Introduction

Modern games and applications use large amounts of texture data; the number and the resolution of textures also continues to grow quickly. However, the amount of available graphics memory is not growing at the same pace and, in addition to textures, GPU memory is also used for complex post-processing effects and lighting calculations. *Virtual texture mapping* (VTM) is a technique to reduce the amount of graphics memory required for textures to a point where it is only dependent on the screen resolution: for a given viewpoint we only keep the visible parts of the textures in graphics memory, at the appropriate MIP map level (see Figure 4.1).

In this chapter, we will investigate how to implement a fully functional VTM system. Readers already familiar with VTM might want to skip right to Section 4.3, which covers several non-obvious implementation aspects. Our tutorial implementation follows this article very closely, so we encourage you to look at the relevant code for each section.

4.2 Virtual Texture Mapping

While early texture management schemes were designed for a single large texture [Tanner et al. 98], recent VTM systems are more flexible and mimic the virtual memory management of the OS: textures are divided into small *tiles*, or pages [Kraus and Ertl 02, Lefebvre et al. 04]. Those are automatically cached and loaded onto the GPU as required for rendering the current viewpoint. However, it is necessary to redirect accesses to missing data to a fallback texture. This

Figure 4.1. Uniquely textured terrain rendering using a single virtual texture.

prevents "holes" from appearing in the rendering, or blocking and waiting until the load request finishes.

Our implementation is inspired by the GDC talk of Sean Barrett [Barret 08] and we suggest watching the video of his presentation while reading this section. As illustrated in Figure 4.2, we begin each frame by determining which tiles are visible. We identify the ones not cached and request them from disk. After the tiles have been uploaded into the tile cache on the GPU, we update an *indirection texture*, or page table. Eventually, we render the scene, performing an initial lookup into the indirection texture to determine where to sample in the tile cache.

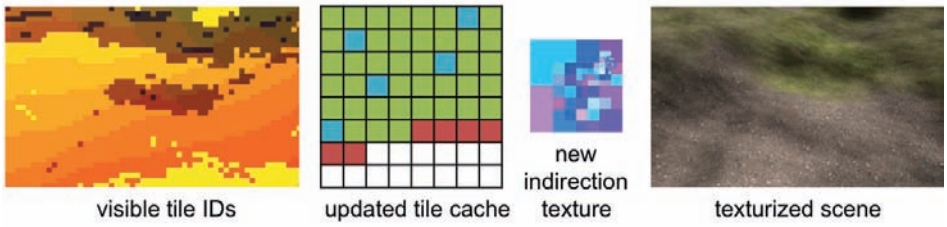

Figure 4.2. We render tile IDs, then identify and upload newly visible tiles into the tile cache (red), possibly overwriting ones that are no longer visible (blue). We update the indirection texture and render the texturized surfaces.

The indirection texture is a scaled down version of the complete virtual texture, where each texel points to a tile in the tile cache. In our case, the tile cache is simply one large texture on the GPU, containing small, square tiles of identical resolution. This means tiles from different MIP map levels cover differently sized areas of the virtual texture, but simplifies the management of the tile cache considerably.

4.2.1 Page Fault Generation

For each frame we determine the visible tiles, identify the ones not yet loaded onto the GPU, and request them from disk. Future hardware might simplify this with native page faults [Seiler et al. 08], but we still need to determine visible tiles, substitute data and redirect memory accesses.

A simple approach is to render the complete scene with a special shader that translates the virtual texture coordinates into a tile ID. By rendering the actual geometry of the scene, we trivially handle occlusion. The framebuffer is then read back and processed on the CPU along with other management tasks. As tiles typically cover several pixels, it is possible to render tile IDs at a lower resolution to reduce bandwidth and processing costs. Also, in order to pre-fetch tiles that will be visible "soon," the field of view can be slightly increased. The corresponding shader code can be found in Section 4.5.2.

4.2.2 Page Handler

The page handler loads requested tiles from disk, uploads them onto the GPU, and updates the indirection texture. Depending on disk latency and camera movement, loading the tiles might become a bottleneck. To illustrate this we fly over a large terrain covered by a single virtual texture and graph the time per frame in Figure 4.3. Given a reasonably large tile cache, very few tiles are requested and on average we need less than ten ms per frame for I/O and rendering. However, in frame 512 we turn the camera 180 degrees and continue backwards. This u-turn requests over 100 tiles, taking 350 ms to load.

To ensure smooth rendering we simply limit the number of updated tiles per frame. For requests not served in the same frame we adjust the indirection texture and redirect texture access to the finest parent tile available in the tile cache. The coarsest level is always present, and this distributes load spikes over several frames. If the update limit is larger than the average number of requested tiles, we are guaranteed to catch up with the requests eventually. For our example we request fewer than five tiles in 95% of the frames, and set the upload limit to a very conservative 20 tiles.

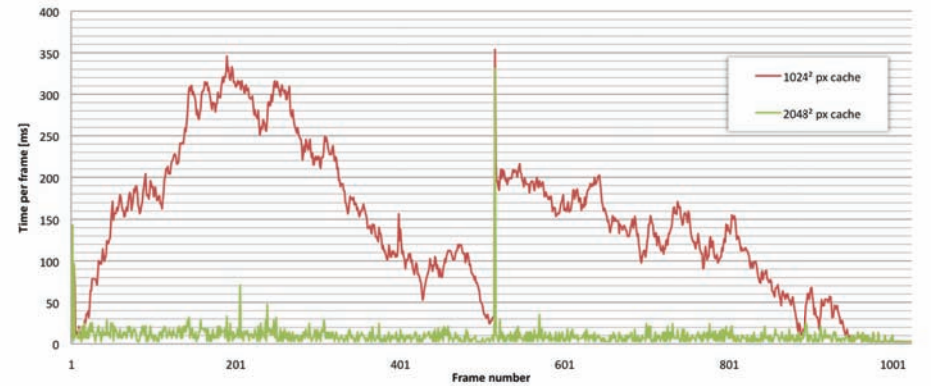

Figure 4.3. We fly over a terrain with one large virtual texture and record the time per frame. In frame 512 we turn the camera 180 ° and continue backwards. This turn is a worst case scenario for VTM: many new tiles—which are no longer in the cache—become visible and have to be loaded. While sufficiently large caches prevent thrashing, they help little in this challenging event.

Of course the missing tiles reduce visual quality. Therefore we upload the requested tiles with all their ancestors, prioritized from coarse to fine. This increases the total number of cache updates, but as Figure 4.4 shows, image quality is restored in a more balanced fashion—very similar to progressive JPEG. As more ancestors are present in the cache, this also improves quality in less challenging situations and reduces artifacts when rendering tile IDs with a very low resolution.

Figure 4.4. Half a second after the u-turn. Left: waiting for all tiles provides superior image quality but stalls for 330 ms. Middle: limiting the uploads per frame and using coarser MIP map levels as fallback provides smooth frame rates, but MIP levels vary strongly. Right: using prioritized loading of ancestors improves fallback, and image quality is much more balanced after the same time.

4.2.3 Rendering

When texturing the surface we perform a fast unfiltered lookup into the indirection texture, using the uv-coordinate of the fragment in virtual texture space. This provides the position of the target tile in the cache and the actual resolution of its MIP map level in the pyramid of the indirection texture. The latter might be different from the resolution computed from the fragment's MIP map level due to our tile upload limit. We add the offset inside the tile to the tile position and sample from the tile cache. The offset is simply the fractional part of the uv-coordinate scaled by the actual resolution:

$$\text{offset} := \text{frac}(uv \times \text{actualResolution}) = \text{frac}(uv \times 2^{\text{indTexEntry.z}}).$$

Note that storing the actual resolution as $\log_2(\text{actualResolution})$ allows us to use 8-bit textures. The complete shader code including the computation of correct texture gradients (see Section 4.3.3) can be found in Section 4.5.3.

4.3 Implementation Details

In this section we will investigate various implementation issues with a strong emphasis on texture filtering. Again we will follow the processing of one frame, from page fault generation over page handling to rendering.

4.3.1 Page Fault Generation

MIP map level. To compute the tile ID in the tile shader we need the virtual texture coordinates and the current MIP map level. The former are directly the interpolated uvs used for texturing, but on DX 9 and 10 hardware, we have to compute the latter manually using gradient instructions [Ewins et al. 98, Wu 98]: let $ddx = (\frac{\delta u}{\delta x}, \frac{\delta v}{\delta x})$ and $ddy = (\frac{\delta u}{\delta y}, \frac{\delta v}{\delta y})$ be the uv gradients in x- and y-direction. Using their maximal length we compute the MIP map level as

$$\text{MIP} = \log_2(\max(|ddx|, |ddy|).$$

The corresponding shader code can be found in Section 4.5.1.

DX 10.1 provides the HLSL function `CalculateLevelOfDetail()`. Further DX 11 gives access to coarse gradients (`dd{x|y}_coarse()`) which might provide an even faster alternative to the level of detail function.

4.3.2 Page Handler

Compressed tiles. For efficient rendering it is desirable to have a DXTC compressed tile cache. It requires less memory on the GPU and reduces the upload and

rendering bandwidth. However, as the compression ratio of DXTC is fixed and quite low, we store the tiles using JPEG and transcode them to DXTC before we upload them. This also allows us to reduce quality selectively and e.g., compress tiles of inaccessible areas stronger.

Disk I/O. For our tutorial implementation we store tiles as individual JPEG files for the sake of simplicity. However, reading many small files requires slow seeks and wastes bandwidth. Packing the tiles into a single file is thus very important, especially for slow devices with large sectors like DVDs.

It is possible to cut down the storage requirements by storing only every second MIP map level and computing two additional MIP maps for each tile: if an intermediate level is requested, we load the corresponding four pages from the finer level instead. More ideas about storage and loading can be found in [van Waveren 08].

Cache saturation. Unused tiles are overwritten with newly requested tiles using an LRU policy. However, the current working set might still not fit into the cache. In this case we remove tiles that promise low impact on visual quality. We replace the tiles with the finest resolution with their lower-resolution ancestors. This plays nicely with our progressive update strategy and quickly frees the tile cache. Other good candidates for removal are tiles with low variance or small screen space area.

Tile upload. Uploading the tiles to the GPU should be fast, with minimum stalling. Using DX 9, we create a managed texture and let the driver handle the upload to the GPU. Other approaches for DX 9 are described in detail by [Mittring 08]. For DX 10 and 11, we create a set of intermediate textures and update these in turn. The textures are copied individually into the tile cache [Thibieroz 08]. DX 11 adds the possibility to update the tiles concurrently, which further increases performance.

Indirection texture update. After the tiles have been uploaded, we update the indirection texture by recreating it from scratch. We start by initializing the top

Figure 4.5. Creating the indirection texture for a camera looking over a large terrain: initializing the top level with the lowest resolution tile, we copy parent entries into the next finer level and add entries for tiles present in the cache.

of its MIP map pyramid with an entry for the tile with the lowest resolution, so each fragment has a valid fallback. For each finer level we copy the entries of the parent texels, but replace the copy with entries for tiles from that level, should they reside in the cache. We continue this process until the complete indirection texture pyramid is filled (see Figure 4.5).

If tiles are usually seen at a single resolution, we can upload only the finest level to the GPU. This reduces the required upload bandwidth, simplifies the lookup, and improves performance. This is sufficient when every object uses an unique texture, in particular for terrain rendering.

4.3.3 Rendering

While rendering with a virtual texture is straight forward, correct filtering, especially at tile edges, is less obvious. Neighboring tiles in texture space are very likely not adjacent to each other in the tile cache. Filtering is especially challenging if the hardware filtering units should be used, as those rely on having MIP maps and correct gradients available. The following paragraphs describe how to use HW filtering with an anisotropy of up to 4:1 as shown in Figure 4.6. The corresponding shader code can be found in Section 4.5.3.

Texture gradients. When two adjacent tiles in texture space are not adjacent in the tile cache, as shown in Figure 4.7, the *uv*-coordinates for the final texture lookup will vary a great deal between neighboring fragments. This results in large texture gradients and the graphics hardware will use a very wide filter for sampling, producing blurry seams. To address this, we manually compute the

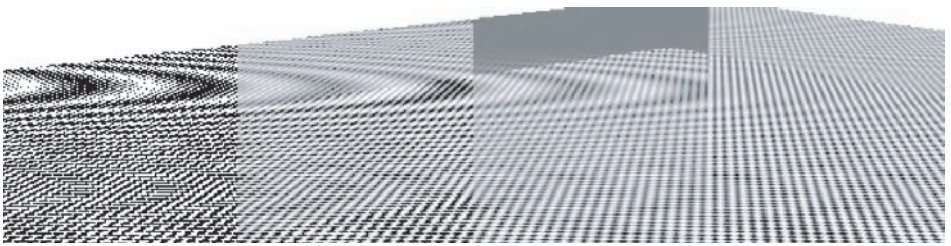

Figure 4.6. From left to right: Hardware point (175 fps), bilinear (170 fps), trilinear (170 fps), and 4:1 anisotropic filtering (165 fps) on an NVIDIA 9700M GT. The transition between tiles is especially pronounced with trilinear filtering. A single tile has to serve several MIP map levels in this example, but only one additional MIP map level is available for filtering per tile. All other methods seem to not use MIP maps at all with the tested driver version.

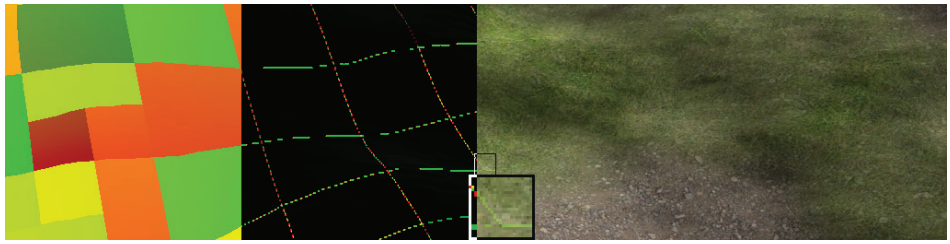

Figure 4.7. Even though the surface has a continuous mapping, the arrangement of the tiles in the cache causes discontinuities. From left to right: direct visualization of the lookup position into the tile cache; *uv* gradients, wrong across tile edges; blurry seams resulting from too wide filters when sampling. Manually scaled gradients fix this artifact.

gradients from the original virtual texture coordinates, scale them depending on the MIP map level of the tile and pass them on to the texture sampler.

Tile borders. Even with correct texture gradients, we still have to filter into neighboring pages, which very likely contain a completely different part of the virtual texture. To avoid the resulting color bleeding we need to add borders. Depending on what constraints we want to place on the size of the tile, we can use *inner* or *outer* borders.

We use the latter and surround our 128^2 tiles with a four-pixel border, making them 136^2. This keeps the resolution a multiple of four, allowing us to compress them using DXTC and perform 4:1 anisotropic filtering in hardware.

DXTC border blocks. As Figure 4.8 illustrates, adding a border to tiles might lead to different DXTC blocks at the edges of tiles. As the different blocks will be compressed differently, texels that represent the same points in virtual texture space will not have the same values in both tiles. This leads to color bleeding across tile edges, despite the border. By using a four-pixel outer border, these compression related artifacts vanish.

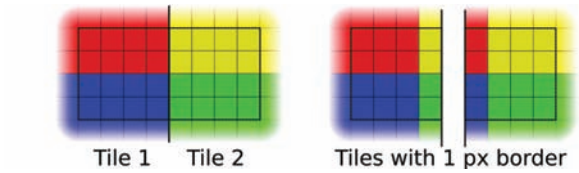

Figure 4.8. Adding a one-pixel border around a tile creates different DXTC blocks for neighboring tiles. Depending on the texture they might be compressed differently, leading to visible seams in the virtual texture despite having borders.

4.4 Conclusion

In this chapter, we described a basic virtual texture mapping system. It is simple and fast, even on older hardware. We gave a few pointers on how to further improve performance, should your application or game require it. Even with newer hardware that might natively support page faults, strategies for loading, compressing and filtering textures will still be required. We hope this article and our tutorial implementation will help you to get started with VTM. You can integrate it into your own application or just play with different parameters and strategies.

4.5 Shader Code

4.5.1 MIP Map Calculation

```
float ComputeMipMapLevel(float2 UV_pixels, float scale)
{
    float2 x_deriv = ddx(UV_pixels);
    float2 y_deriv = ddy(UV_pixels);

    float d = max(length(x_deriv), length(y_deriv));

    return max(log2(d) - log2(scale), 0);
}
```

4.5.2 Tile ID Shader

```
float2 UV_pixels = In.UV * VTMResolution,

float mipLevel = ComputeMipMapLevel(UV_pixels, subSampleFactor);
mipLevel = floor(min (mipLevel, MaxMipMapLevel));

float4 tileID;
tileID.rg = floor(UV_pixels / (TileRes * exp2(mipLevel)));
tileID.b  = mipLevel;
tileID.a  = TextureID;

return tileID;
```

4.5.3 Virtual Texture Lookup

```
float3 tileEntry = IndTex.Sample(PointSampler, In.UV);
float actualResolution = exp2(tileEntry.z);

float2 offset = frac(In.UV * actualResolution) * TileRes;

float scale = actualResolution * TileRes;
float2 ddx_correct = ddx(In.UV) * scale;
float2 ddy_correct = ddy(In.UV) * scale;

return TileCache.SampleGrad(TextureSampler,
                            tileEntry.xy + offset,
                            ddx_correct,
                            ddy_correct);
```

4.6 Acknowledgments

We'd like to thank J.M.P van Waveren for generously sharing his insights on virtual texture mapping.

Bibliography

[Barret 08] Sean Barret. "Sparse Virtual Textures." 2008. http://silverspaceship.com/src/svt/.

[Ewins et al. 98] JP Ewins, MD Waller, M. White, and PF Lister. "MIP-Map Level Selection For Texture Mapping." *Visualization and Computer Graphics, IEEE Transactions on* 4:4 (1998), 317–329.

[Kraus and Ertl 02] Martin Kraus and Thomas Ertl. "Adaptive Texture Maps." In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 7–15. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002.

[Lefebvre et al. 04] Sylvain Lefebvre, Jerome Darbon, and Fabrice Neyret. "Unified Texture Management for Arbitrary Meshes." Technical Report RR5210, INRIA, 2004. Available online (http://www-evasion.imag.fr/Publications/2004/LDN04).

[Mittring 08] Martin Mittring. "Advanced Virtual Texture Topics." 2008. http://ati.amd.com/developer/SIGGRAPH08/Chapter02-Mittring-Advanced_Virtual_Texture_Topics.pdf.

[Seiler et al. 08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. "Larrabee: A Many-Core x86 Architecture for Visual Computing." *ACM Trans. Graph.* 27:3 (2008), 1–15.

[Tanner et al. 98] C.C. Tanner, C.J. Migdal, and M.T. Jones. "The Clipmap: A Virtual Mipmap." In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 151–158. ACM New York, 1998.

[Thibieroz 08] Nicolas Thibieroz. "Ultimate Graphics Performance for DirectX 10 Hardware." GDC Presentation, 2008.

[van Waveren 08] J. M. P. van Waveren. "Geospatial Texture Streaming from Slow Storage Devices." 2008. http://software.intel.com/en-us/articles/geospatial-texture-streaming-from-slow-storage-devices/.

[Wu 98] Kevin Wu. "Direct Calculation of MIP-Map Level for Faster Texture Mapping." Hewlett-Packard Laboratories, 1998. http://www.hpl.hp.com/techreports/98/HPL-98-112.html.