Chapter 2

Advanced Virtual Texture Topics

Martin Mittring[1]
Crytek GmbH

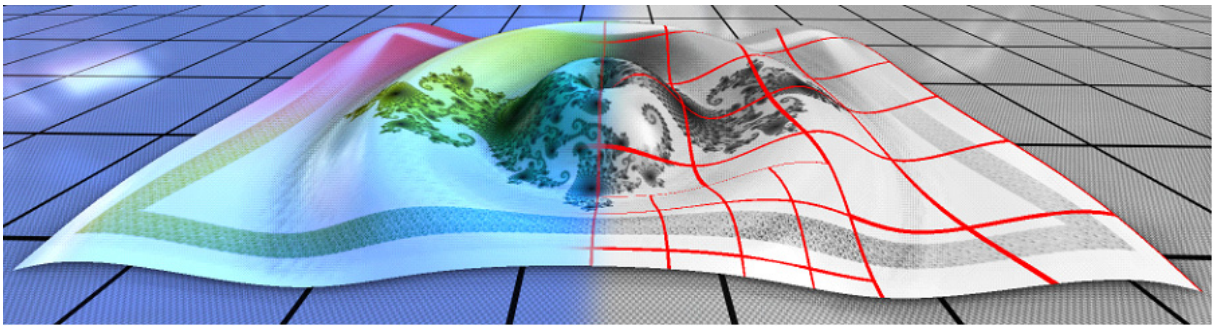

Figure 1. *A visual representation of a virtual texture*

2.1 Abstract

A *virtual texture*[2] is a mip-mapped texture used as cache to allow a much higher resolution texture to be emulated for real-time rendering, while only partly residing in texture memory. This functionality is already accessible with the efficient pixel shader capabilities available on the recent generations of commodity GPUs. In this chapter we will be discussing technical implications on engine design due to virtual textures use, content creation issues, results, performance and image quality. We will also cover several practical examples to highlight the challenges and to offer solutions. These include texture filtering, block compression, float precision, disk streaming, UV borders, mip-map generation, LOD selection and more.

[1] Martin@Crytek.de

[2] The term is derived from the OS/CPU feature "virtual memory", which allows transparent memory access to a larger address space than the physical memory.

2.2 Motivation

The diagram in *Figure* shows how different hardware devices for texture storage can be classified with a different speed/amount ratio. Caching is a common technique to allow fast access to larger data set to live in slower memory. The virtual texture described here uses using traditional texture mapping to cache data coming from the respective slower content device.

Figure 2. *Hardware can be classified depending on a speed/amount ratio.*

Note: On the GPU a texture lookup operation is limited to one texture only and random access to the whole video memory is not possible, limited in size[3] or high latency. Because of that the diagram in *Figure* lists "Texture" and "Video memory" as separate units.

2.2.1 Texture Streaming is Becoming a Necessity

Texture mapping is common-place and highly efficient on consumer GPUs for over a decade. Many challenges have been solved by hardware support for mip-mapping, advanced texture filtering, border clamp/mirror rules and compressed texture formats. Modern real-time rendering engines are faced with another challenge: Screen resolution and higher quality standards now require high resolution textures and for draw call efficiency it's even advised to share one texture for multiple objects [NVIDIA04]. Some graphics hardware already supports texture resolutions up to 8K (8192), but that might not be enough for some applications, and, more of a problem, the memory requirements grow rapidly with texture size. Because the simulated world size is also expected to be much larger it's no longer possible to keep all textures in graphic card memory (a typical limit is 512MB) and not even in main memory. Having more main memory doesn't help when limited being by the 32 bit address space (2GB on typical 32bit OS). A 64 bit OS allows using more main memory but most installed OS and

[3] The maximum size of a texture can be a limiting factor (usually from 1024 to 8192 depending on a specific hardware generation).

hardware is still 32 bit. Limited amount of physical memory can be compensated by using virtual memory from hard drive. Unfortunately this option is not viable for real-time rendering as traditional virtual memory (as an OS and hardware feature) stalls until the request is resolved. The situation is exacerbated without an available hard-drive as it might be the case on consoles. To overcome this and to get a fast level loading time modern engines are required to do texture streaming.

2.2.2 Problems with Per Mip-Map Texture Streaming

We can avoid stalling while requesting to load a specific texture mip level from the hard-drive by using a lower mip level as a fallback until the desired level is uploaded to the graphics card. This is acceptable for real-time games and the lower resolution texture can go unnoticed with sufficient care. Unfortunately it's basically impossible to add or remove a mip-map dynamically. The Direct3D9 function `SetLOD()` was made for that but that only affects the video memory alone and doesn't change the issue of the physical and virtual memory. Most hardware keeps all mip-maps in one block of continuous memory and updating a single mip-level becomes a full mip chain update. In Crysis™ (**Error! Reference source not found.**) we wanted to save virtual memory so to adjust the mip-level we had to create and release textures at runtime. That is very bad for stable performance and MultiGPU (SLI/Crossfire) scaling but it was a manageable solution at the time. Streaming allowed us to stay within the 32 bit limits with run-time data requirement sometimes exceeding the limits. On 64 bit and enough main memory or when using half resolution textures the texture streaming is not necessary and performance is more stable.

***Figure 3**. The screenshot from the game* Crysis™ *shows the need for texture streaming: large rich environments with many details.*

Avoiding `Create()` and `Release()` calls at runtime is possible when textures can be reused, but only if texture formats and sizes match. This is very restricting and even wasteful on the memory usage, and therefore not a practical option. As a result, this problem is a serious issue for game developers and deserves better API and hardware support.

The idea of virtual textures is to manage the texture memory at a different granularity than the mip-map level. Often only small areas are required in each mip-map and thus uploading a full mip-map would be wasteful. It's much more efficient if we only upload the actually required portions. The virtual texture method itself is simple, but it has a lot of interesting related topics attached to it which we will discuss here after explaining the basic method itself.

2.3 Implementing Virtual Textures with Pixel Shaders

2.3.1 Virtual Texture – A Definition

For the virtual texture we only keep relevant parts of the texture in fast memory and asynchronously request missing parts from a slower memory (while using the content of a lower mip-map as fall-back). This implies that we need to keep parts of lower mip-maps in memory and these parts need to be loaded first. To allow efficient texture lookups coherent memory access is achieved by slicing up the mip-mapped texture into reasonably sized pieces. Using a fixed size for all pieces (now called texture tiles) the cache can be managed more easily and all tile operations, e.g. reading or copying, have constant time and memory characteristics. Mip-maps smaller than the tile size are not handled by our implementation. This is often acceptable for applications like terrain rendering where the texture is never that far away and little aliasing is acceptable. If required, this can be solved as well by simply packing multiple mip-maps into one tile. For distant objects it's even possible to fall back to normal mip-mapped texture mapping, but that requires a separate system for managing that.

As shown in Figure 4, a typical virtual texture is created from some source image format at preprocessing time without imposing the API and graphics hardware limits on texture size. The data is stored on any lower access speed device, such as hard drive. Unused areas of the virtual texture can be dropped (saving memory as a nice side bonus). The texture in the video memory (now called *tile cache*) consists of tiles required to render the 3D view. We also need the indirection information in order to efficiently reconstruct the virtual texture layout. Both the tile texture cache and the indirection texture are dynamic and adapt to one or multiple views.

To manage our virtual texture we use a quad-tree because all required operations can be implemented in constant time. Here the state of the tree represents the currently used texture tiles for a virtual texture. All nodes and leaves are associated with a texture

tile and in a basic implementation only the highest available resolution in the quad-tree is used. The lower resolution data is stored as fall-back when we drop some leafs and can also be used to fade in higher resolution texture tiles gradually. We refine or coarsen the virtual texture only at the leaf level. In addition to the quad-tree we need additional code implementing the cache strategy (e.g. Least Recently Used).

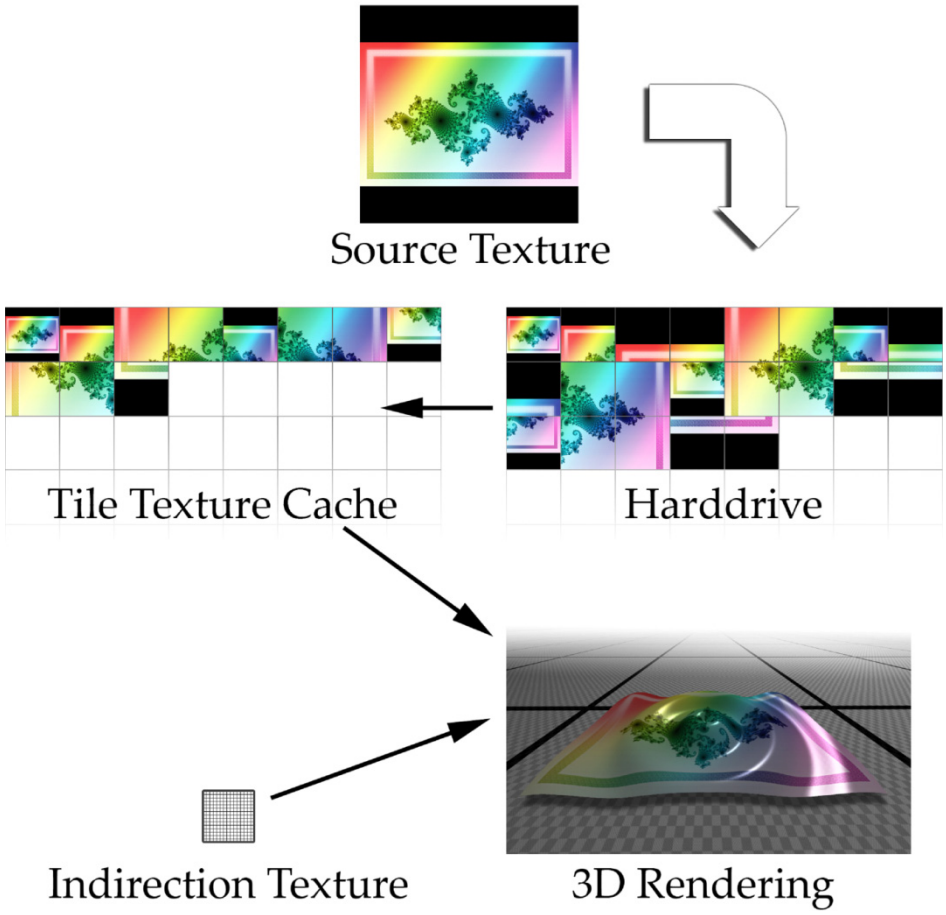

Figure 4. *Typical usage scenario of the virtual texture method*

2.3.2 Reconstruction in the Pixel Shader

The reconstruction needs to be very efficient. While some applications allow efficient indirections per draw call[4] or pre-split geometry[5], this is too limiting in general and hard to implement efficiently for general geometry. Simply using a pixel shader to implement this functionality is straight-forward and intuitive. This code returns the texture

[4] Games like Far Cry™ or Crysis™ render one terrain sector with texture tile per draw call. That allows the tile cache to be split in individual texture and that simplifies tile updates.
[5] It's possible to setup the vertex texture coordinates to render multiple tiles in one draw call.

coordinates in the tile cache texture for a given virtual texture coordinate. This even allows emulating texture coordinate addressing modes like “warp” or “mirror”.

Here we assume at least 32 bit float precision in the computations (which is not supported by older pixel shader versions, but is common by latest generations of DirectX® 9.0c-capable graphics hardware). Note that on DirectX® 9 you have to offset your texture lookups by half a texel. In OpenGL you have to do similar computations.

Efficiency is very important as this code is executed for every pixel. This is why the quad-tree traversal in the pixel shader is replaced by a single unfiltered texture lookup. This texture (now called indirection texture) can be quite small in memory and because of the coherent texture lookups it is also very bandwidth friendly. A single texture lookup allows computing the texture coordinates in the tile cache with simple math in constant time.

The idea of implementing a virtual texture using a pixel shader received a lot of attention after John Carmack mentioned the “Mega texture” technique he has been working on (as described in [IDTECH507]). The technique is used in the commercial product “Quake Wars” and is currently developed to a more generalized solution at `id` Software. The basic idea is clear, but implementation details are not described. Sean Barrett investigated further and shared his knowledge and his version of the method at GDC 2008. It’s an advised read for anyone that wants to implement it [BARRETT08].

The following HLSL code can be used in the pixel shader to compute for the given virtual texture coordinate the associated tile cache texture coordinate:

```
float4 g_vIndir;             // w,h,1/w,1/h indirection texture extend
float4 g_Cache;              // w,h,1/w,1/h tile cache texture extend
float4 g_CacheMulTilesize;   // w,h,1/w,1/h tile cache texture extend
                             //             * tilesize

sampler IndirMap = sampler_state
{
  Texture   = <IndirTexture>;
  MipFilter = POINT;
  MinFilter = POINT;
  MagFilter = POINT;
//  MIPMAPLODBIAS = 7;       // using mip-mapped indirection texture,
                             // 7 for 128x128
};

float2 AdjustTexCoordforAT( float2 vTexIn )
{
  float fHalf = 0.5f;        // half texel for DX9, 0 for DX10

  float2 TileIntFrac = vTexIn*g_vIndir.xy;
  float2 TileFrac = frac(TileIntFrac)*g_vIndir.zw;
  float2 TileInt = vTexIn - TileFrac;
  float4 vTiledTextureData = tex2D(IndirMap,TileInt+fHalf*g_vIndir.zw);

  float2 vScale = vTiledTextureData.bb;
  float2 vOffset = vTiledTextureData.rg;

  float2 vWithinTile = frac( TileIntFrac * vScale );

  return vOffset + vWithinTile*g_CacheMulTilesize.zw + fHalf*g_Cache.zw;
}
```

Listing 1. *HLSL Shader Code to compute the texture coordinates in the tile cache texture*

The code can be optimized further but care must be taken to keep the floating point error minimal. The texture cache should not have mip-maps and the lookup should be bilinear only.

In [BARRETT08] Sean Barrett mentions the simplest shader fragment he came up with (with a hint from John Carmack):

```
tex page   , vtc , tex0     , 2D
mad phys.xy, vtc , page.xyxy, page.zwzw
tex color  , phys, tex1     , 2D
```

Listing 2. *Pixel shader assembler code to compute the texture coordinates with the tile cache lookup*

This would be 2 instructions only for the texture coordinate computation. We haven't tried that but float precision might be an issue, especially when using 64 bit or even 32 bit textures - using a 128 bit texture may be slower on some hardware.

2.3.2.1 *The Indirection Texture*

The indirection texture can be easily generated from the data in the quad-tree. With a single unfiltered lookup into the indirection texture and simple math with constants we can compute the texture coordinates in the tile cache. The values stored in a texel contain the scale of the tile and the 2D offset in tile cache. In our implementation we use a 64 bit texture with FP16 channels. Using a 32 bit texture format with 8 bit channels is possible but you have to adjust the values in the pixel shader with additional instructions. Beware that scaling might not return the values you would expect. By storing a value from 0 to 255 in the texture you get values from 0.0 to 1.0 in the pixel shader. This result is a guaranteed. Scaling these values by 255.0 you would think would result in integer values. However, this may not be the case. Floating point math can be an issue, but even worse is that on some hardware the precision seems to be lower, rather comparable to 16 bit floats. In [BARRETT08] the problem was solved by rounding[6], but the author admits that this might be not the most efficient approach.

Here we use a 64 bit (4 channels FP16) texture format as it is compact and doesn't suffer from the issues mentioned before. The memory bandwidth requirements for our use are minimal as the method has a very high texture lookup coherency, i.e. texture cache misses are rare. However depending on the hardware the lookup itself might require multiple cycles on 64 bit or 128 bit texture formats. Integer textures, an integer texture lookup[7] and integer math can be a good choice on some hardware as we want constant precision over whole domain and float is likely to cause problems here. This would also allow using higher resolution texture caches (> 8K) easily.

[6] To get the integer value from of a 8 bit channel this shader code was used: floor(channel*255+0.5)
[7] Direct3D 10 offers the HLSL Load() but only unfiltered

The following C/C++ code can be used to compute the indirection texture content:

```
// float to fp16(s1e5m10) conversion (does not handle all cases)
WORD float2fp16( float x )
{
  uint32 dwFloat = *((uint32 *)&x);
  uint32 dwMantissa = dwFloat & 0x7fffff;
  int iExp = (int)((dwFloat>>23) & 0xff) - (int)0x7f;
  uint32 dwSign = dwFloat>>31;

  return (WORD)( (dwSign<<15)
      | (((uint32)(iExp+0xf))<<10)
      | (dwMantissa>>13) );
}

WORD texel[4];              // texel output
RECT recTile;               // in texels in the tilecache texture
int iLod;                   // 0=full domain, 1=2x2, 2=4x4, ...
int iSquareExtend;          // indirection texture size in texels
float fInvTileCache;        // tile.Width / texCacheTexture.Width

texel[0] = float2fp16(recTile.left*fInvTileCache);
texel[1] = float2fp16(recTile.top*fInvTileCache);
texel[2] = float2fp16((1<<iLod)/ iSquareExtend);

texel[3] = 0;               // unused
```

Listing 3. *C/C++ code to compute the content of the FP16 indirection texture*

Using a mip-mapped indirection texture requires a few more texture update operations but it also allows per-pixel LOD which looks much smoother. The per-pixel LOD code computes a lower (or the same) LOD that is available in the texture tile cache. Standard texture mapping with LOD adjustment[8] can be sufficient but anisotropic texture mapping would provide better quality at steep angles. In *Figure* different texture filtering modes are shown.

The virtual texture technique can be extended to more than two dimensions. By using volume textures or multiple slices in a 2D texture the lookup is still quite efficient. However multidimensional content scales quickly regarding memory demand and then it's better to adapt at a finer granularity, i.e. a smaller tile size is needed. In GPGPU applications (such as [LEFOHN03]) data is often processed in volume textures and often barely fits into video memory. Caching can be done as usual but processing might require the full data set and locally varying LOD is not common and thus dynamic methods are often not used there.

[8] See `MIPMAPLODBIAS` in the pixel shader code

2.3.2.2 *Efficient Filtering Through Borders*

A naive implementation of bilinear filtering requires 4 lookups into the indirection texture, 4 lookups in the tile cache, followed by bilinear interpolation in the shader. While this may be somewhat reasonable for a hardware implementation, in the pixel shader implementation this is wasteful with respect to performance. Adding a small border is much more efficient as the much more efficient built-in hardware bilinear filtering can be used. A one-pixel border is enough to get bilinear filtering on uncompressed textures but in order to add support for DXT compressed textures a 4 pixel border is necessary. This is because the DXT block compression is based on 4x4 blocks and to avoid seams you need to add a full block to the border. Furthermore it's better to center the tile to get more stable results for imprecise texture coordinate computations. That also simplifies the implementation of more advanced filtering like bi-cubic filtering or anisotropic filtering. The later one would be an interesting topic for this chapter but because we haven't done any implementation we skip it here.

Unfortunately the additional borders waste memory, destroy the power-of-two texture extents, and break the memory alignment of the tiles. If you have non-power-of-two tiles it's probably better to add some padding to the tile cache to create the texture with power-of–two dimensions[9]. Otherwise you might be faced with undefined memory and performance characteristics from the APIs and graphic card drivers.

Alternatively, instead of adding the border to the tile we can also reduce the tile size by the border to allow the sum of both to be power of two[10]. This is best for the hardware implementation but resulting visual quality can suffer a lot. That loss is due to aliasing in the mip-maps caused by the down sampling of the source texture to slightly less than its half size. A good down-sampling algorithm can limit the aliasing in the lower mip-maps but even the top mip-map is affected by this design decision and that can be visible especially when using regular patterns in the texture.

2.3.2.3 *Maximum Size of the Virtual Texture*

As mentioned, our implementation is based on a single indirection only and assumes all tiles in the tile cache have the same size you can compute the virtual texture resolution:

$$\text{Resolution}_{\text{virtual texture}} = \text{Resolution}_{\text{Indirection texture}} * \text{Resolution}_{\text{texture tile without border}}$$

Examples: 16k = 128 * 128
65k = 256 * 256
256k = 256 * 1024

[9] e.g. for 7 tiles with 128+4 pixel extend (128+4)*7=924, the next power of two extend would be 1024
[10] e.g. for 8 tiles with 124+4 pixel extend (124+4)*8=1024, but the usable tile size is no longer power of two

Using a larger tile size limits the adaptive property of the method and using a larger indirection texture becomes inefficient when updating the texture, especially with CPU updates. Unfortunately there is another limit for virtual texture resolution. With a typical implementation using floating point math to run on older hardware the precision of the float computations becomes a problem when the virtual texture resolution becomes close to 65K. We may see reasonable performance for colour look-ups; however, bilinear filtering will no longer be efficient. We can alleviate the problem by careful ordering of our floating point operations, however, integer maths avoid this all together.

2.3.2.4 *Storing Different Attributes in the Tile Caches*

You can comprise one tile cache of multiple textures to store attributes like diffuse, specular or normal maps as long they share the same tile positions. After computing the position in the tile cache it can be efficiently used for multiple lookups. With a bigger border you can even use differently sized textures for the attributes you want to store.

2.3.2.5 *Splitting the Tile Cache Over Multiple Textures*

Instead of storing different attributes you can use multiple tile caches to get more cache units, but here a new problem appears. Normal hardware rendering only allows to texture from the same textures in one draw call. Performance can be much worse when trying to overcome this limitation: Texture lookups through texture arrays are a bit slower (Direct3D® 10 only) and the alternative of fetching data from several textures and masking the result is even slower. That's why it's good to keep all tiles required for one draw call in the in the same tile cache.

When using multiple tile caches you might end up with one tile cache overused while another one is underused. Moving objects between different caches might be an option but performance will no longer be constant, no matter what strategy you pick. Grouping specific types of objects (e.g. one tile cache for terrain and one for objects) is the simpler solution.

The maximum texture resolution supported on some hardware limits your tile cache size. Here we have two simple solutions: You can tweak the LOD computation and accept blurrier results or you add another cache in the graphic card memory, between the texture and the main memory. Copying between VRAM and the texture is expected to be fast. By computing the local LOD required as small as possible the tile cache can be kept small (see the LOD computation methods described later). Rendering a view is possible from the main tile cache, changing view angle additionally requires the secondary cache and moving the view position requires secondary cache updates. The

later ones can come from a slow media like the DVD and to minimize latency more cache stages can be done on the hard drive and even in main memory.

2.3.2.6 *Tile Cache Texture Updates*

As already mentioned, to avoid bilinear filtering artefacts with DXT block compressed textures we require an additional border of 4 pixels. Due to the lossy DXT compression we need exactly the same block content when compressing blocks of neighbour tiles; otherwise we might reconstruct wrong colour values and thus resulting in visible seams.

There are three basic methods to update a part of a mip-map from CPU. Depending on the specific graphics API[11] and on the graphics card use and the driver version, performance characteristics may be not clearly defined. This is actually is the major problem of the implementation and on some configurations it might even make the method unusable. Further testing is needed to quantify this claim. Experience shows that such driver issues often get addressed after a major game shipped using the technology.

For the update of the tile cache texture we have some requirements:

- Fast in latency and throughput
- Bandwidth efficient (copy only the required part)
- Small memory overhead
- Updates should happen without stalls but correctly synchronized to get the right texture state
- When updating the content by CPU there should be no copy from GPU memory to CPU memory (discard should be used)
- For fast texturing from the tile cache texture it should be in the appropriate memory layout (swizzled[12]) and memory type (video memory). Note that on some hardware compressed textures are stored in linear form (not swizzled).
- Multiple tile updates should have linear or better performance

All methods require some locking of either a full texture or a part of the texture. The driver might do a full update and you probably would only notice on less powerful hardware or with heavy bus usage. We're still investigating further into this area. To describe the three basic methods we use the Direct3D 9 API.

Method 1: Direct CPU update:

The destination texture needs to be in `D3DPOOL_MANGED` and via the `LockRect()` function a section of the texture is updated. This method wastes main memory, and most likely the transfer is deferred till a draw call is using the texture. This method is simple but likely to be less optimal when compared with the next two methods.

[11] OpenGL, Direct3D® 9, Direct3D® 10 or the APIs used on modern consoles

[12] A swizzled memory layout is a cache friendly layout for position coherent texture lookups.

Method 2: With (small) intermediate tile texture:
Here we need one lockable intermediate texture hat can hold one tile (including border). With the `LockRect()` function this texture is updated as a whole. With the `StretchRect()` function the texture is then copied into the destination texture to replace one tile only. This does not work with compressed textures (DXT) as they cannot be used as render target format. For the `StretchRect()` function the source and the destination requires to be in `D3DPOOL_DEFAULT` and that requires the source to be `D3DUSAGE_DYNAMIC` to be lockable. To find out if the driver supports dynamic textures, you are ought to check the caps bits for `D3DCAPS2_DYNAMICTEXTURES` but according to the list in the DirectX SDK even the lowest SM20 cards support this feature.

Method 3: With (large) intermediate tile cache texture:
This method requires a lockable intermediate texture in `D3DPOOL_SYSTEM` with the full texture cache extend. With `LockRect()` the intermediate texture is updated only where required and a following `UpdateTexture()` function call is transferring the data to the destination texture. `UpdateTexture()` requires the destination to be in `D3DPOOL_DEFAULT`.

2.3.2.7 *Indirection Texture Update*

Once the new tile is in the texture cache the indirection texture can be updated. We wish to make this an efficient operation. The indirection texture requires little memory therefore bandwidth is not a issue. However, using several indirection textures and updating them often can become a performance bottleneck. The texture can be updated from the CPU by locking or uploading a new texture. This can cause irregular performance characteristics on current APIs but at the same time has proven to be an acceptable solution. Locking a resource that is in use by the GPU can definitely produce some hitches unless clever renaming is done on the driver side.
If you choose to have a render target texture format for your indirection texture you can also consider GPU updates triggered by CPU. Updates can be done by draw calls and simple quads can be rendered to the texture to update even large regions efficiently.
There shouldn't be too many updates as tile cache updates should be rare and both rely on each other.

In our implementation the indirection texture still has a channel left and storing a tile blend value is possible. With extra shader cost this allows hiding texture tile replacement by slowly blending tiles in or out. A filtered blend value would be even nicer as it allows hiding the seams between the tiles. However this can hide details and experience showed satisfying results without this feature.

2.3.3 Mip-Mapping and Virtual Textures

It's best to generate the mip-maps and do the texture compression in an offline process. This way a high quality implementation can be used and the data is optimally prepared for fast access. This is similar to a normal production pipeline but some details differ.

For efficient streaming the mip-maps are organized on disk as tiles of continues blocks and in that form the data needs to be generated. Most mip-map computation implementations assume that the textures can be fully loaded into memory. In that case, developers may waste memory on a large texel format and buffer duplication. When using huge virtual textures you have to assume that the texture cannot be processed in memory, especially if you want your tools to run on 32 bit OS. Fortunately it's not too difficult to make the mip-map generation code running without keeping the full data in memory.

2.3.3.1 *Out of Core Mip-Map Generation*

Typically the input data is stored on the disk in a standard image format. For efficient processing we convert it into a form that allows fast access for algorithms with read and write operations with strong locality. Here again we can make use of some tile based data layout. The tile size here is not dependent on the virtual texture tile size and the border pixels are undesired because that would add redundancy. Redundant data like this can speed up the processing under certain circumstances, but it might cause other problems in the system.

We need a class that can read a section of the image but completely hiding the tile based data layout. The input area can be defined by any rectangle, even outside of the source domain. Pixels outside of the domain can return the wrapped content or a border colour. The class caches tiles that have been requested recently and it also should supports writing tiles temporary on the hard drive. This is used to store the mip-map levels during processing. Using the same functions to access the source data and the intermediate data simplifies the code a lot.

Generating mip-maps is simple: Generate the lowest mip-map of the image recursively by requesting images one mip-map higher and run your favourite mip-mapping filter (add border depending on kernel). In a second pass the generated tiles are extracted, compressed and stored into the streaming friendly format (add border for bilinear filtering and compression).

2.3.3.2 *Kernel Size for the Mip-Map Generation*

For the mip-map generation you can pick either an even (e.g. 2×2 or 4×4) or an odd sized (e.g. 3×3 or 5×5) kernel to down sample the texture to a quarter (half in both

dimensions). This affects the result a lot and that it even has implications on the virtual texture pixel shader implementation.

The **even-sized kernel** is used for normal hardware mip-mapped texture mapping and is therefore good to be used if a consistent look is important. This might be the case if you want to render some object, maybe depending on distance, without the virtual textures.

Example: Box 2×2:

1/4	1/4
1/4	1/4

The **odd-sized kernel** simplifies the virtual texture pixel shader a bit and has the nice property that texel positions with a unfiltered colour in a lower mip always exists in all higher mips. This allows specifying UV coordinates that get always get a defined colour in higher mip-maps. This is useful if you want to get seamless texture mapping or unique UV unwrapped geometry. The alternative is to add border pixels but that's only limiting the problem, not solving it.

Example: Gauss 3×3:

1/16	2/16	1/16
2/16	4/16	2/16
1/16	2/16	1/16

Properties of the different mip-mapping kernel sizes:

	Even mip-map kernel size	Odd mip-map kernel size
Example kernels	2x2 box, 4x4 gauss, 4x4 sharpen	**3x3 gauss, 5x5 gauss sharpen**
GPU creation support	Yes (fast for a full mip chain)	**No but easy to implement in PS**
GPU rendering support	Yes (allow tri-linear and anisotropic)	**Bilinear yes, Mip mapped rendering has offset issues**
Creation Speed	fastest	**fast**
Rectangular charts	**filtering artefacts**	**Precise (texel center in higher mips remain at position)**

Slightly better quality can be achieved by making use of the higher mip-maps (not only the next higher one). This is even more important if your mip-maps are not exactly half the size. As mentioned earlier it can be useful to keep the tile size including the border a power of two, but that requires a more complex mip-map generation algorithm.

Hardware tri-linear filtering

Just a note on trilinear filtering – trilinear filtering is simple to implement in the shader with additional texture fetches. If we wish to use hardware filtering for this, we need a mip-mapped tile cache, i.e. redundant storage and updates of the texture tiles. Additionally this requires a wider texture border and storing the tiles in two resolutions wastes memory and complicates the updating a fair amount.

2.3.4 Possible Tile Sources

2.3.4.1 *Streaming from Disk*

A very good source for virtual texture tiles can be the hard drive [WAVEREN06]. Requests are fulfilled in more or less constant time and memory requirements are constant. Normal IO functions on modern OS should be avoided because they try to cache and combine requests which not only results in variable latency but the memory for this cache is now competing with our application. Code or data which is not frequently accessed can be paged out and we get frame-rate hitching. Windows offers IO Completion ports which can be used efficiently and caching from the OS can be disabled. Using this you have a more direct hardware access but the reads now needs to align with the disk cluster size. As this can be different depending on which formatting the user chose, it's best to align your tile size and placement. The cluster size is a power of two and often in the range from 4K to 32k[13]. Knowing this it's clear that for best performance the header size of the streaming file should be 0 or padded to align with the cluster size. Each tile should be split in as few clusters as possible and preferably nearby to tiles that have a similar locality. With non static compression where each tile can have a different memory size this can be a bit tricky.

Streaming from DVD/CD/BD/HD DVD is more challenging as latency and bandwidth is much worse [NOGUCHI08]. On first sight the latency might not sound like a problem if you think of one tile being requested and coming in after that time. With very few tiles in flight you actually get good performance but with more tiles the seek time is dominating more and more. The alignment here is more important because the data is read and decoded in the cluster size and ignoring this is wasting the precious performance. It

[13] It's good to have the code hiding the cluster alignment from the user code so the code works for clusters of any size.

might make sense to consider a larger tile size to get better performance from hardware with high latency.

2.3.4.2 *Streaming over Network*

Wouldn't it be nice to join a multiplayer match without the need to download a map? You can join an ongoing game and even see all the detailed changes like tire tracks and decals that happened to the map. The normal game server or some special extra server can bake decals and distribute them to the clients on demand. Compression might be more important here, but it seems that downstream bandwidth will be much less of a problem in the future.

2.3.4.3 *Procedural Content Generation*

Nowadays especially massive multiplayer games require a huge amount of work to fill the world with details. Different areas should have a unique look to provide a richer gaming experience. Having a technique that allows rendering more detail doesn't help either.

Many techniques [BRUNETONNEYRET08] [LHN04] [ANDERSSON07] [WOODARD05] [QMK06] [GLANVILLE04] [DACHSBACHER06] are known to generate procedural content and if the generation step is fast enough this can be even done on the fly similar to a texture tile request from hard drive. Data can be generated on the CPU or the GPU. It's important to give the artists enough control over this; otherwise you only remove the tiled texture look without getting the content you want.

One of the simplest ways to generate huge texture resolutions is already used in many games rendering terrain; Terrain detail often consists of some tiled textures that are blended with interpolation information at a much lower resolution [BLOOM00]. This gives good results, especially when combined with a low resolution texture to add variation and break the tiling look.

Crysis uses terrain material blending (see *Figure*) to get a detailed ground texture for a huge terrain but that method requires multiple materials to be blended at the pixel level. The overdraw can be up to three as each terrain vertex is assigned to one material and so blending within a triangle requires up to three passes. Additionally the detail materials are modulated by a low frequency texture.

Using such techniques in real-time gets slower the more features you pack into the system. Baking this in an offline process allows much more sophisticated blending and keeps the rendering performance constant. This also allows baking details like roads or tire tracks. You might have to use a lower resolution but this can be hidden with detail texturing.

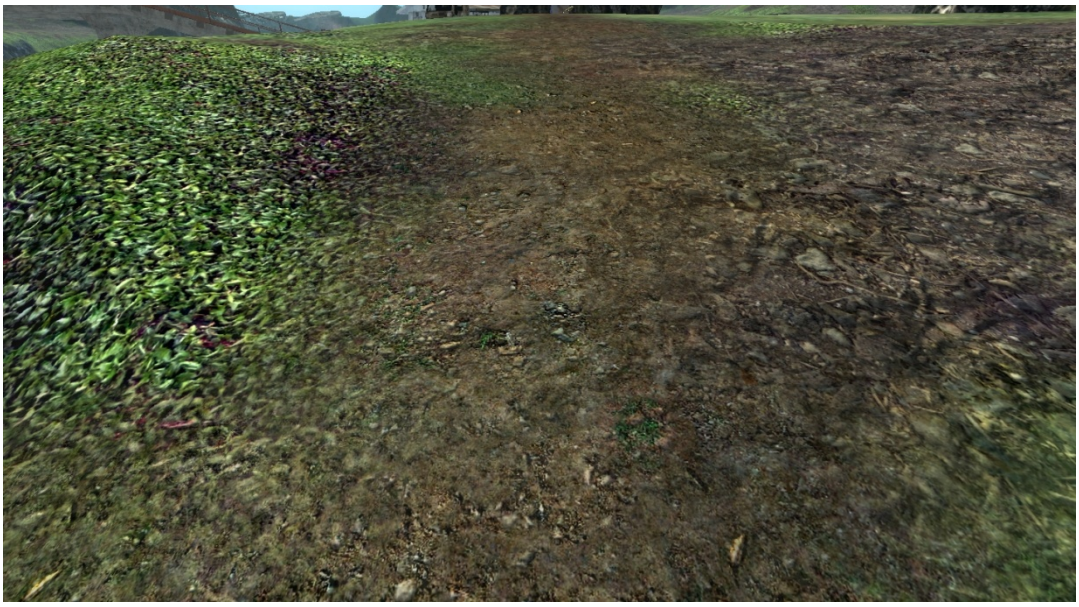

***Figure 5**. Terrain material blending as it can be seen in* Crysis *(terrain detail objects have been removed). Note that this is simply a reference – this particular screenshot was not implemented using virtual texture method*

Placing decals is one way to give the artists control over local areas. Decals can be used in multiply mode to bring brightness and colour variation, or they can be used in alpha-blend mode to locally replace existing content. As in nature, self similarity is common and it is smart to reuse the decal. This can be taken to the limit where a large number of placed decals form the surface itself. This allows breaking the tiled texture look but doing it without control results in a repetitive decal look.

***Figure 6**. Example of a scenario that would benefit from using virtual textures - decals in the game* Crysis *(roads, tire tracks, dirt) used on top of terrain material blending.*

Decals or *Roads* (as quantities in our game) can be seen as some special form of vector graphics which supports high resolutions naturally. Classic 2D vector graphics can be used for content like signs, advertisements or large scale paintings and even longer text sections are possible. Even large content like terrain with roads and other man made structure are suited for vector graphics content [BRUNETONNEYRET08]. Again the data can be generated on the fly or in an offline process.

The offline process has the advantage of constant rendering performance and even more important, constant memory requirements. Rendering 3D content is often no good idea as it requires additional memory for textures and meshes but there is one interesting application that doesn't suffer from this as it shares data with the normal scene rendering. [FFBG01] describes a method named „Adaptive shadow maps"(ASM) and for semi static scenes this can be an interesting option. The technique does not require any mesh unwrapping as the shadow map is projected from the light source perspective.

Other methods often require a unique UV unwrapping to store some intermediate data of the shading computation. Instead of storing the shadow map depth value, the light occlusion factor for a given surface position can be baked. Ambient occlusion, the effect of multiple lights or static global illumination can be baked as well. Taking this to the limit allows baking the final resulting colour of shading into a virtual texture. The data can still be dynamic as the tiles can be updated but now shading is decoupled from rendering and this brings along nice properties. It might be not the right time to ship games fully based on that concept as graphics hardware is not made for that. In real-time rendering this is known [BAKER05] but rarely used, in offline rendering decoupling shading from rasterization is used to great success in the REYES rendering system. In contrast to per-pixel-shading the REYES system does heavy tessellation of the geometric primitives, computes shading at that level and introduces other interesting things worth checking out [GUAN07].

2.3.4.5 *Sparse Textures*

Encoding material properties might be limiting and blending materials with alpha masks allows for much more flexible material blending. In [ANDERSSON07] sparse textures are used to compress this kind of data efficiently. Texture tiles with the same content can index to the same cache element and by using masks with larger areas of the same value this allows good compression. If only few tiles need to be stored you might not even require a dynamic cache. Then you just index with a static indirection texture into a static tile cache. This can be seen as a special case of the normal adaptive texture method and this way it integrates nicely with the dynamic virtual texture asset pipeline. Even multiple indirections for multiple masks can be stored in one indirection texture[14].

[14] As a hint: The UV position of the tiles in the texture cache can be encoded in one component and the scale might not be required.

2.3.4.6 *Combo Textures*

When using multiple masks the mentioned technique becomes less efficient. Compression of the sparse textures is only efficient when using one indirection per mask. An alternative is a novel technique we call the "Combo texture". The method allows to store up to 2^n masks in n texture channels in a compressed form. The compression is lossless under some controllable conditions and becomes lossy with complex material blends. Texture filtering can be used as usual but filtering introduces blending which again can suffer from the same compression issues.

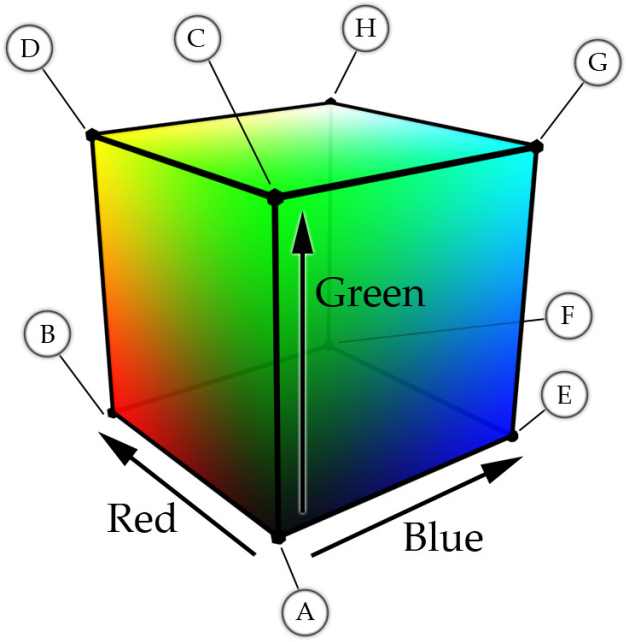

***Figure** 7.The RGB Cube shows how up to 8 materials (A..H) can be blended with one colour only.*

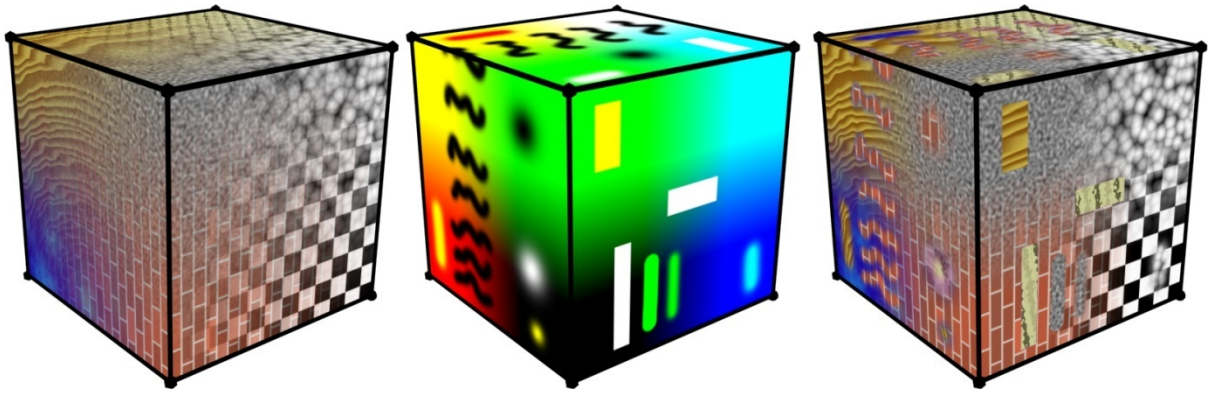

***Figure 8**. Multiple materials can be blended using the combo texture method.* Left*: Seven materials are blended on the RGB cube colours.* Middle*: A combo texture is projected on a cube, setup to blend materials in many different ways.* Right*: Seven materials are blended based on the combo texture from the image before.*

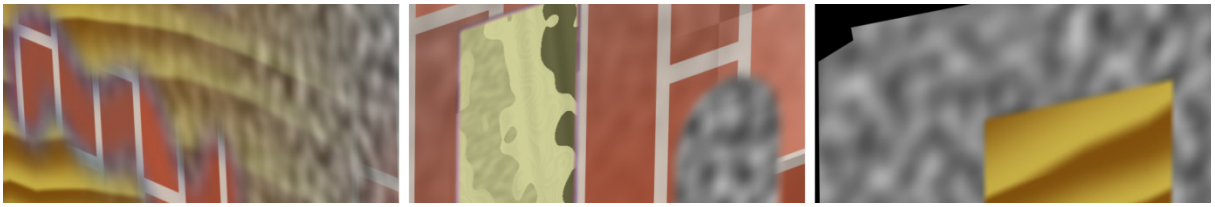

Figure 9*.Combo texture results zoomed in*
Left: Blending between two incompatible colours (yellow, black) causes leaking.
Middle: Bilinear filtering between two incompatible colours (black, white) causes leaking.
Right: Bilinear filtering between two compatible colours (green, yellow) blends nicely.

If we use a 3 channel combo texture the properties of the method can be described on the RGB cube (*Figure*). Each corner of the cube represents one material and eight corners allow therefore eight materials to be blended. All corners and the edges allow lossless compression as the linear blend between two materials can be expressed as a linear blend between two positions in the cube. Expressing arbitrary material mixes or blending between materials in general can introduce material-leaking[15] or the wrong amounts of certain materials. By smarty placing the materials in the right corners artefacts often can be avoided. Artists can paint these combo textures easily but if they paint material masks the material placement in the combo texture can be rearranged much easier at a later stage. Material masks can be converted to combo textures or any other representation before normal texture compression is applied in the art pipeline. If the material leaking becomes too apparent the bilinear filtering can be replaced by shader code that doesn't exhibit this problem. These artefacts are usually only visible in magnification where virtual textures can help to increase the resolution.

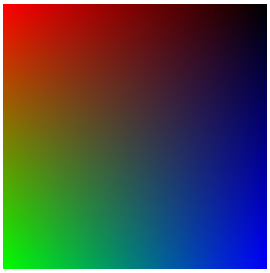

Figure 10*. Four simple materials (colour: red, black, green, blue) are blended in a single pass which results in the best quality.*

The combo texture method can be implemented in a simple single pass shader if the shader of the different materials are simple and share common properties (e.g. detail materials, Phong materials that share textures and differ only in colours and specular power). *Figure* 10 shows four most simple materials (solid colour) blended in a single pass. With complex materials the possible shader permutation count explodes and a lot of bandwidth would be wasted by data that gets masked away.

[15] Materials that haven't been part of the mix are blending into the result.

A multi-pass solution based on frame-buffer blending would be more flexible.

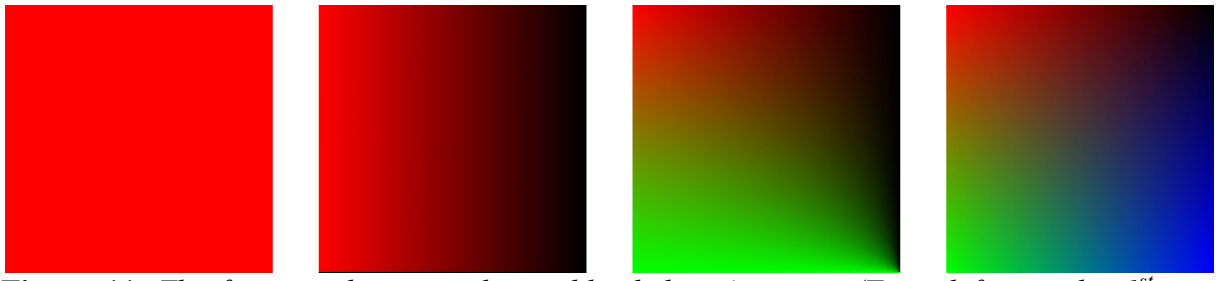

Figure 11. *The four simple materials are blended in 4 passes. (From left to right: 1st pass, 2nd pass, 3rd pass, final pass)*

Figure 11 demonstrates how such a technique works by using alpha blending. A simple implementation could use additive blending but this requires more than 8 bit precision in the frame buffer for an acceptable look. Better is to lerp[16] with the frame buffer content. To compute the blend factor we need to know the material blend factor and the sum of the blend factors we have blended already. Because frame buffer blending isn't flexible and precise enough we compute the sum in the shader. The following shader code illustrates the computation for eight materials:

```
float3 g_ComboMask;             // RGB material combo colour
                                // (3 channels for 8 materials)
                                //   000,100,010,110,001,101,011,111
float4 g_ComboSum0,g_ComboSum1; // RGBA sum of the masks blended so far
                                // including the current
                                // (8 channels for 8 materials)
                                //   10000000, 11000000, 11100000, 11110000
                                //   11111000, 11111100, 11111110, 11111111

float ComputeComboAlpha( BETWEENVERTEXANDPIXEL_Unified InOut )
{
  float3 cCombo = tex2D(Sampler_Combo,InOut.vBaseTexPos).rgb;

  float3 fSrcAlpha3 = g_ComboMask*cCombo + (1-g_ComboMask)*(1-cCombo);
  float fSrcAlpha3 = fSrcAlpha3.r*fSrcAlpha3.g*fSrcAlpha3.b;

  float4 vComboRG = float4(1-cCombo.r,cCombo.r,1-cCombo.r,cCombo.r)
                  * float4(1-cCombo.g,1-cCombo.g,cCombo.g,cCombo.g);

  float fSum = dot(vComboRG,g_ComboSum0)*(1-cCombo.b)
             + dot(vComboRG,g_ComboSum1)*(cCombo.b);

  // + small numbers to avoid DivByZero
  return (fSrcAlpha+0.00000001f)/(0.00000001f+fSum);
}
```

Listing 4. *Shader to compute the alpha value when rendering a material with the combo texture method*

[16] lerp = linear interpolation A*(1-factor)+B*factor

Additional Notes:

- Using up to 8 materials and using an optional alpha channel for custom properties is nice and useful; when using more than three channels the technique becomes less intuitive and more artefacts can appear.
- A smart usage of combo textures might allow dropping other textures like diffuse or specular textures.
- You can modify the combo texture to dynamically blend between different materials (dirt, rust, damaged, wet).
- To avoid overdraw it is good to use alpha test and for static combo textures you can create a static index buffer per material.

2.3.5 Mesh Parameterization

The virtual texture method relies on unwrapped 3D models and that can be done with the usual techniques either by hand or in some automated way [FARIN02] [CARRHART02]. Knowing that the unwrapping is used with a virtual texture the layout can be optimized for better performance. Unused areas are common in all unwrapping methods but packing the data specifically to reduce the amount of wasted texture tiles allows faster loading with less cache and storage memory required. The packing algorithm needs to avoid the quad-tree borders at multiple levels (see Figure 12). Another important criterion is to aim for similar locality between the texture space and world space, that again at the granularity of the texture tiles.

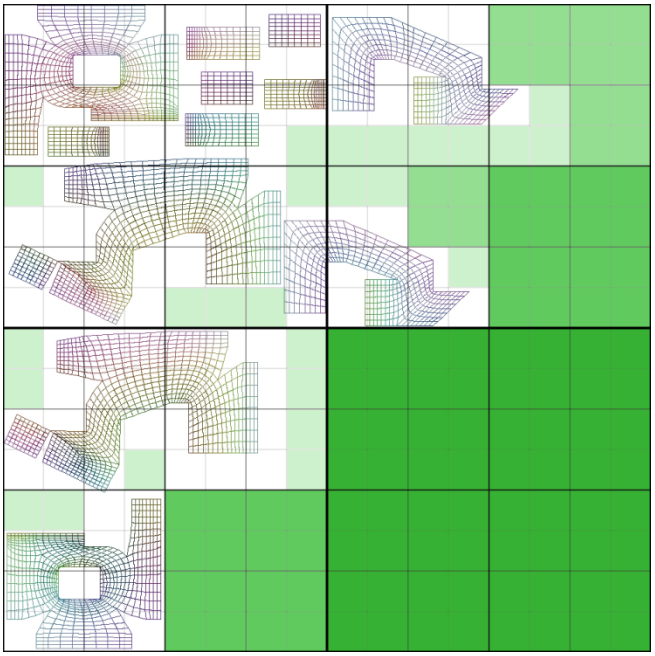

Figure 1. *By carefully placing the unwrapped mesh in the UV space less tiles are requires at multiple levels of the quad tree (green tiles don't need to be stored).*

2.3.5.1 *Unique Unwrapping*[17]

The unwrapping does not have to be unique when using virtual textures but having a unique unwrapping can have additional benefits. For the virtual texturing a unique mapping has a higher but more consistent memory demand in the texture cache (see "Computing the local LOD required"). Dynamic tile content virtually requires unique unwrapping as texture updates should not affect other areas. Even static content might require unique unwrapping if the content cannot be shared. Examples for that are light maps or object-space normal maps. Normal maps in tangent-space are easier to compress but object/world-space allows better quality and simpler shaders.

A simple form of unique unwrapping almost unwraps every geometric primitive (triangle, quad, grid, …) individually. Once tessellation is commonly supported in hardware this together with displacement maps or geometry images might be a good solution [THC*04] [RECMULTI4E][LMH00]. It should be mentioned that this would allow moving the indirection to the primitive to become more efficient.

2.3.6 Computing the Local LOD

2.3.6.1 *How Much Tile Cache Memory is Required?*

As mentioned in "mesh parameterization" when using a unique mapping the virtual texturing method has a higher but more consistent memory demand in the texture cache.

This is because non unique mappings allow sharing some tiles if the viewer is close to a 3D surface that shares the same area on the virtual texture. Assuming unique unwrapping the memory required in the texture cache can be approximated by multiplying the screen width and height with the average over-draw and a user defined quality factor. The computation is only approximating the real value because it ignores mip-mapping, anisotropy, bilinear filtering and waste because of tile borders. The wasted memory becomes more when used with a bigger tile size as full tiles are required even if only partly used. The pixel overdraw is 0 if that pixel is not using the adaptive texture method and it's greater than 1 if alpha blending is used.

Graphic cards normally compute the mip-map level depending of the screen texture derivation in *x* and *y* for a two pixel block[18]. For high anisotropy levels like for walls or roads seen in the distance the mip-level for *x* and *y* can be quite different. Non anisotropic filtered lookup on the graphic card here uses the maximum of the values

[17] Unique texture unwrapping: A position on the texture is only mapped to one model position; there is no reuse of the texture.

[18] ddx() is computed from an odd/even horizontal pixels pair, ddy() from a vertical pair

and as a result the texture looks blurry but it benefits from good texture cache coherence and has no shimmering in motion.

Figure 2. *Different texture filtering techniques shown on rendering of a road section*

In *Figure* a road is shown with different texture filtering modes. Starting from the left you can see standard bilinear which only picks the lowest mip-map. The second image shows the bilinear filtering without the use of mip-maps. That looks good on the screenshot but in motion the noisy aliasing is very apparent. The following images show 2X, 4X and 8X anisotropic texture filtering. Note how the white stripes keep the details in the distance depending on the anisotropic filtering level.

2.3.6.2 *Computing Exact Local Tile LOD*

With a high frame-rate on current hardware it's usually not possible to get the local LOD and the resulting update accomplished within the same frame. Then the perfect local LOD would have to include future frames and that's almost unsolvable for dynamic scenes. We have to assume a latency of multiple frames and therefore it's better to find a more conservative LOD computation.

Occluded tiles can save tile cache space but those might become quickly visible. To find the texture tiles that are hidden behind objects for many frames a high level streaming prediction system based on some occlusion system (e.g. a static PVS) can help.

Rapid view direction changes are common in real-time games and it is important to integrate in the local LOD computation.

2.3.6.3 *Approximating Local Tile LOD*

A reasonably conservative approximation can be acceptable and can sometimes even be better than the exact LOD for the current frame. It's better to have tiles already available to be prepared for quick camera angle changes or fast object movements. Computing the LOD based on the distance per triangle or draw call can be such an approximation (easy to implement for height-maps[19]). For content that has a varying world to Texel density the LOD computation can be extended further.

[19] Height-map rendering can make use of the virtual texture cache without any pixel shader indirection if the mesh is organized in a similar quad-tree structure.

2.3.6.4 Exact Local Tile LOD in View Space with Occlusion

In [BARRETT08] the author describes a method to compute the local LOD by rendering the view as a pre-pass with some special shader that outputs the tile-id with the required mip-map level. The resulting image is used to get the LOD required for some local area of the virtual texture. It can be tricky to get the data back to the CPU efficiently and some latency cannot be avoided. With a high frame-rate this can be multiple frames and therefore the cause of artefacts. Hidden object parts that become visible (from occlusion, animation or by entering the view) suffer from temporal local blurriness.

2.3.6.5 Exact Local Tile LOD in Texture Space

Most of the mentioned problems can be avoided when computations are performed in texture space [LDN04B]. Such methods even work outside of the view or when being occluded by other objects or the object itself. The object needs to be rendered in texture space using a medium resolution with a pixel shader computing the colour of the Texel as LOD for the point in world space. Overlapping primitives in the texture space require extra treatment which is of course not required when using a unique UV unwrapping. The overlapping areas should get the colour value of the highest LOD required there and luckily the z buffer can easily be abused for that.

Then the data can be extracted for different texture areas with Occlusion queries or other methods like CPU read-back. This 2 pass algorithm allows multiple tests in different areas of the quad-tree. As the returned information is changing slowly the renderings can be distributed over multiple frames without introducing severe problems.

We haven't tried the GPU method; instead we implemented a CPU based method. It was easier to implement (with some approximations) and doesn't suffer from the latency issues the GPU solution has. The idea here is to build up a quad tree in texture space to compute distances to quad-tree nodes in world space. That can be CPU heavy and memory intensive but it's a good reference solution.

The described methods have quite some different properties and it depends on the application and hardware platform which works best. The view space method with occlusion only returns the minimal set of tiles required but it offers no good prediction. Concerning prediction, the other two methods show their strengths and a combination with one of them can be a good idea. Which one is the better choice depends on the platform and on the data. Computing the local tile LOD per triangle is very bad with huge triangles spanning multiple tiles. This can be solved by subdividing the mesh but then the method becomes more complex and memory intensive.

2.4 Future Directions

Many interesting graphic algorithms rely on some form of LOD query and local texture updates but the currently available hardware and API (DirectX®/OpenGL) is not supporting this very well.

We [game developers] need asynchronous partial texture update and mip-level adjustment with predictable performance. It seems we get virtualized memory for graphic card memory which is nice from an OS perspective but for games we don't want stalling virtual memory. The application or the engine can deal with missing data on a more high level and can provide fallbacks until the request is resolved. Whatever methods are used and the described virtual texture method is just one, we need to be able to deliver frame rate quality and see this as a valuable feature (Quality of Service).

2.5 Acknowledgements

This chapter wouldn't be the same without the passionate work of the many programmers, artist and designers at Crytek. Working there is both inspiration and demanding as we take every good idea to the limit and challenge it against other methods. Thanks to Efgeni Malachewitsch for the 3D model, Nick Kasyan, Anton Kaplanyan, Michael Kopietz and all others that helped me with this text. Special thanks to Natasha Tatarchuk, Kev Gee, Miguel Sainz, Yury Uralsky, Henry Morton, Carsten Dachsbacher and the many others from the industry for the interesting discussions on my favorite topic: Graphics

2.6 References

[ANDERSSON07] ANDERSSON, J. 2007. Terrain Rendering in Frostbite using Procedural Shader Splatting, Advanced Real-Time Rendering in 3D Graphics and Game, Course 28, Siggraph 2007, San Diego, CA. http://ati.amd.com/developer/gdc/2007/Andersson-TerrainRendering(Siggraph07).pdf

[BAKER05] BAKER, D. 2005. Advanced Lighting Techniques, Meltdown, Seattle 2005. http://www.slideshare.net/mobius.cn/advanced-lighting-techniques-dan-baker-meltdown-2005

[BARRETT08] BARRETT, S. 2008. Sparse Virtual Texture Memory, Game Developer Conference, San Francisco, CA. http://silverspaceship.com/src/svt

[BLOOM00] BLOOM, C. 2000. Terrain Texture Compositing by Blending in the Frame-Buffer, http://cbloom.com/3d/techdocs/splatting.txt

[BRUNETONNEYRET08] BRUNETON, E.AND NEYRET, F. 2008. Real-time rendering and editing of vector-based terrains, In Proceedings of Eurographics 2008, Vol. 27, Num. 2, http://www-evasion.imag.fr/Publications/2008/BN08/article.pdf

[CARRHART02] CARR, N. A. AND HART, J. C. 2002. Meshed Atlases for Real-Time Procedural Solid Texturing, ACM Transactions on Graphics (TOG), , Vol. 21, No. 2, pp. 106 – 131, http://graphics.cs.uiuc.edu/~nacarr/papers/rtpst.pdf

[DACHSBACHER06] DACHSBACHER, C. 2006.Cached Procedural Textures for Terrain Rendering, ShaderX^4: Advanced Rendering Techniques, Engel, W. (Editor), Charles River Media, Cambridge, MA.

[FFBG01] FERNANDO, R., FERNANDEZ, S., BALA, K., AND GREENBERG, D. P., 2001. Adaptive Shadow Maps. In Proceedings of ACM SIGGRAPH 2001, Computer Graphics Proceedings, Annual Conference Series, pages 387–390

[GUAN07] GUAN, S.-H. 2007. Reyes and Shader Pipeline, http://www.scribd.com/doc/7346/Reyes-and-Shader-Pipeline

[GLANVILLE04] GLANVILLE, R. S.2004. Texture Bombing, in *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, Fernando, R. (Editor), Addison-Wesley, April 2004. http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch20.html

[QMK06] Qin, Z., McCool, M. D. AND Kaplan, C. S. 2006. Real-Time Texture-Mapped Vector Glyphs, I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games, pp. 125 – 132, Redwood City, CA. http://www.cgl.uwaterloo.ca/~zqin/i3d2006/ft_gateway.cfm

[LLOYD05] LLOYD, B., YOON, S., TUFT, D. AND MANOCHA, D. 2005. Subdivided Shadow Maps, Technical Report TR05-024, University of North Carolina at Chapel Hill, http://gamma.cs.unc.edu/ssm/ssm_TR05-024.pdf

[LEFOHN05] LEFOHN, A.2005. A Dynamic Adaptive Multi-resolution GPU Data Structure, GPGPU: general-purpose computation on graphics hardware, Siggraph 2005 Courses, Los Angeles, CA, August 2005, http://www.gpgpu.org/s2005/slides/lefohn.AdaptiveCaseStudy.ppt

[FARIN02] FARIN, G., FEMIANI, J., AND RAZDAN, A. 2002. Parametrizing Triangulated Meshes, Seminar, PRISM RA, 2002. http://prism.asu.edu/publications/pr_list_details.php?ref_num=117

[IDTECH507] id Software Debuts id Tech 5 Press Release, 2007 http://www.idsoftware.com/business/press/index.php?date=20070611000000

[NVIDIA04] NVIDIA WHITEPAPER, 2004. Improve Batching Using Texture Atlases http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/BatchingViaTextureAtlases/AtlasCreationTool/Docs/Batching_Via_Texture_Atlases.pdf

[EPHANOV06] EPHANOV, A. AND COLEMAN, C. 2006. Virtual Texture: A Large Area Raster Resource for the GPU, http://www.multigen-paradigm.com/pdf_content/2006IITSEC_VTPaper_2.pdf

[LMH00] LEE, A., MORETON, H. AND HOPPE, H. 2000. Displaced subdivision surfaces, In Proceedings of SIGGRAPH 2000, pp. 85-94, August 2000. http://research.microsoft.com/~hoppe

[LHN04] LEFEBVRE, S., HORNUS, S. AND NEYRET, F. 2004. All-Purpose Texture Sprites, ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5209.pdf

[LDN04B] LEFEBVRE, S., DARBON, J. AND NEYRET, F., 2004. Unified Texture Management for Arbitrary Meshes, Technical Report RR5210-, INRIA, Number RR5210, May 2004, http://www-evasion.imag.fr/Publications/2004/LDN04/RR-5210.pdf

[LEFOHN03] LEFOHN, A. 2003. Dynamic Volume Computation and Visualization on the GPU, http://www.vis.uni-stuttgart.de/vis03_tutorial/lefohn.pdf

[NOGUCHI08] NOGUCHI, M. 2008. Running Halo 3 Without a Hard Drive, Presentation http://www.bungie.net/images/Inside/publications/presentations/Loading_done_gdc_2008.pptx

[THC*04] TARINI, M. HORMANN, K., CIGNONI, P. AND MONTANI, C. 2004. PolyCube-Maps, In Proceedings of Siggraph 2004, pp. 853-860, Los Angeles, CA, 2004. http://vcg.isti.cnr.it/polycubemaps/resources/sigg04.pdf

[WAVEREN06B] J.M.P. VAN WAVEREN, 2006. Real-Time Texture Streaming & Decompression, http://softwarecommunity.intel.com/articles/eng/1221.htm

[WOODARD05] WOODARD, T. 2005. Real-time GPU-based texture Synthesis, In proceedings of IMAGE 2005, http://www.diamondvisionics.com/docs/Real-time%20GPU-based%20Texture%20Synthesis.pdf