Texture Virtualization for Terrain Rendering

Daniel Cornel*
Vienna University of Technology

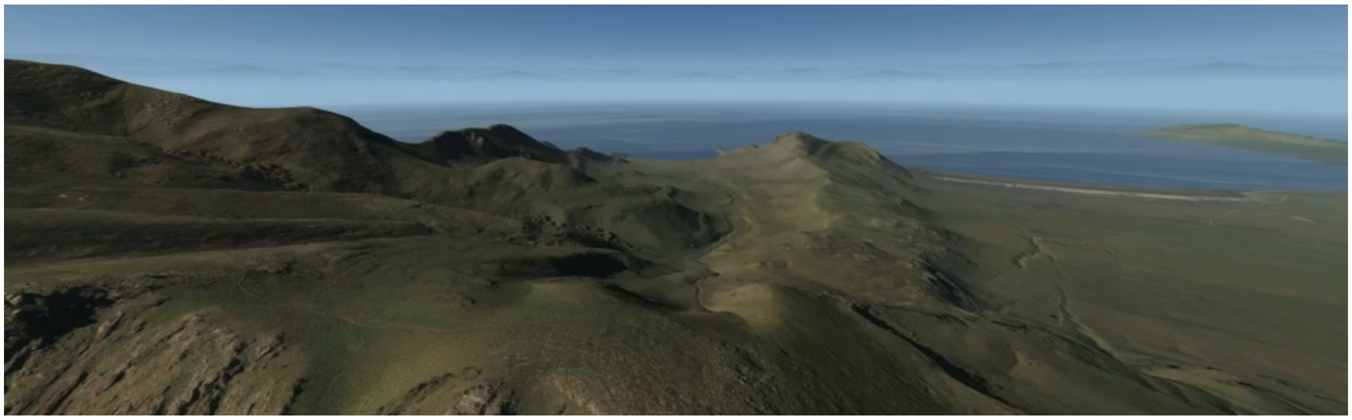
Figure 1: Demonstration of CUDA-accelerated virtual texturing by Hollemeersch et al. [Hollemeersch et al. 2010] using aerial photography of the Antelope Island State Park, Utah. Image retrieved from [Multimedia Lab 2011].

Abstract

Virtual texturing is a technique that allows the use of arbitrarily large textures within the limited physical video memory. Through a paging and streaming system, only the currently visible parts of a mipmap chain are stored in the video memory while the rest of the data may reside in any other memory or storage device. Not only does this enable the use of unique and very detailed textures, but makes high resolution images such as satellite or aerial photography data usable in real-time applications without further modifications or downsampling.

This work sketches the virtual texturing pipeline and discusses the benefits and limitations of it. Due to the nature of terrains in real-time applications, the discussed methods are of particular importance for performant and photorealistic terrain rendering and are thus viewed with regard to these properties and needs. Special emphasis is devoted to recent developments in virtual texturing and possible future fields of application as well as acceleration techniques.

Keywords: terrain rendering, virtual texturing, clipmap

1 Introduction

Realistic real-time rendering of terrain surfaces has been subject to research for the past few years, because the unique characteristics of terrains restrict the application of conventional rendering methods.

*e-mail: daniel.cornel@tuwien.ac.at

In contrast to regular objects placed in the scene and represented by a closed surface with finite extents, a terrain represents a very large surrounding that extends over the whole scene. Due to the large extents, it is most often partially, but never completely visible, with some regions very close to the eye and some regions very far away. Thus, both microstructure details for the nearby and macrostructure details for the more distant surrounding are needed for the surface to appear as a landscape shaped by nature without artifacts or artificial repetitions.

Since the introduction of texture mapping, textures have been a very important way of adding visual details to geometry. However, due to the size of terrains and the limitations of texture size in hardware, mapping a single texture containing the whole terrain surface to the geometry is not feasible. There have been several approaches to overcome this problem by synthesizing the terrain texture through either procedural texture generation or multi-texturing. The latter method has proven very suitable for real-time application and has been used in game engines such as the *CryENGINE 2* [Mittring 2008] and the *Unreal Engine 3* [Epic Games 2012]. Sophisticated implementations such as *procedural shader splatting* [Andersson 2007] used in the *Frostbite Engine* procedurally generate several low- and high-frequency textures which are blended in the rendering step, leading to very few visible repetitive patterns due to the vast amount of possible texture combinations. Texture streaming can be used to reduce the number of texture parts that need to be stored in the video memory, however in naive implementations, this requires pre-generated information of which parts to stream for which part of the scene, based on the view point. A downside of multi-texturing approaches is that, depending on the number of textures used, the resulting image might either look washed out or require a lot of texture lookups. A second limitation is the dependence on a regular tessellation of the terrain geometry such that the texturing can be done for individual tiles of the terrain.

In the end, terrains can be textured quite aesthetically with texture synthesis approaches nowadays, but only with blended synthetic texture tiles. With an increasingly high degree of realism achievable in rendering and the ability to generate very high resolution images, e.g. by aerial photography, methods are needed to use these images as textures in photorealistic rendering applications. Besides the obvious requirement to work within limited video memory, such a method has to perform in real-time on common hardware and has to offer the same visual quality as common texture mapping. Furthermore, texturing should be independent from the mesh tessellation without constraints on geometry or texture data. The first concept to accomplish this was the *clipmap* [Tanner et al. 1998]. Clipmaps make it possible to store only a needed subset of a mipmap chain in the video memory, thus allowing to use arbitrarily large textures in theory. As this concept is a milestone in texture virtualization, the idea of clipmaps is sketched in Section 2.

The further development of clipmaps finally led to extensive adaptations of the content creation and rendering pipelines, summarized under the term *virtual texturing*. Most of these modifications have already been proposed in 2004 [Lefebvre et al. 2004]. However, it was not until the announcement of the video game *Rage* by id Software and its eventual release in 2011 that virtual texturing received wider attention. Thus, literature concerning the topic is sparse, which is why the MA thesis by Mayer [Mayer 2010] currently serves as the most comprehensive publication on the matter. Mayer also proposes standardized terms and a reference implementation with profound evaluations. This serves as reference for the general structure of a virtual texturing system which is discussed in detail in Section 3. Another recent publication by Neu [Neu 2010] providing a similar reference implementation is also taken into account. The aim of this report is to give an insight into the concept of a state-of-the-art virtual texturing system as well as its open problems. As the contributions of Mayer and Neu date back to 2010, an overview over recent developments and applications of virtual texturing is given in Section 3.5.

2 Clipmaps

The clipmap introduced by Tanner et al. [Tanner et al. 1998] is a clipped mipmap that holds only the subset of the mipmap that is potentially visible in the current frame. The idea behind the clipmap is that with a very large texture, all data of a mipmap is never needed in one frame. As the mipmap level selection for texture sampling aims for a 1:1 ratio between pixels and texels, the window size directly limits the number of texels used from a mipmap level. Thus, the mipmap levels larger than the *stack size* (see Figure 2), i.e. the window size plus a constant margin, can be clipped and do not need to reside fully in the video memory.

In contrast to the constant window size, the view point will likely change during the execution of the application, so a *clipmap stack* has to be updated continuously to contain the needed data. A *clip center* is calculated directly from the view point and specifies a center of interest in texture space and, through the clipmap stack, rings of decreasing texture resolution around it. If the size of a mipmap level is greater than the stack size, a region of the stack size around the clip center is cut out of the mipmap level and stored in the clipmap stack. If the mipmap level is smaller than the stack size, it covers the whole texture at a low resolution and is stored in the *clipmap pyramid*. This way it can be used as a fallback lookup texture for every region of the texture if the desired level of detail of the region around the clip center is not yet available to the video card.

This concept is analogous to the virtual memory management of operating systems where contents of the main memory are outsourced to a cheaper hard disk space. Together with a page table keeping track of all outsourced memory pages, this allows to address a memory space much larger than the physical main memory. Instead of virtual memory, *virtual textures* are handled now and their smallest units are uniformly sized texture *tiles* which are loaded from the hard disk first and then streamed from the main memory. This introduces several problems, like the determination of which tile to be streamed, scheduling of the streaming and addressing of the tiles.

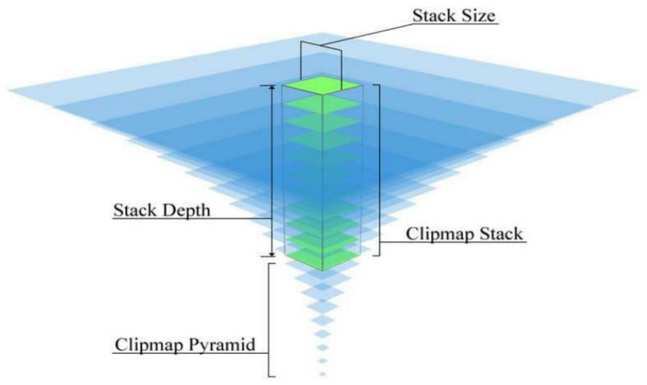

Figure 2: Illustration of a clipmap. Instead of the whole mipmap chain (blue), only the clipmap stack (green) and the clipmap pyramid are stored in the video memory. Image retrieved from [NVIDIA Corporation 2007].

It is obvious that the coherence of the potentially visible tiles between two frames can be used to significantly reduce streaming. As the view point moves, neighboring tiles of the currently visible ones are very likely to be needed soon, so they will be streamed when possible. If the new tiles are needed for rendering before streaming is complete, a lower resolution of the desired tile is used as fallback texture. For this lower resolution is included in the next lower clipmap level, it is important to stream the required textures from bottom up, so fallback tiles of all required tiles are accessible first. To access tiles of the complete mipmap in the clipmap system, Tanner et al. [Tanner et al. 1998] proposed a toroidal addressing scheme for clipmap levels and tiles such that after change of the clip center, the new tiles can be appended to the ones residing in the video memory easily. With the proposed wraparound addressing, however, the limited precision and numeric range supported in hardware limit the maximum number of addressable mipmap levels and therefore the size of the whole texture. This is why the authors did a second virtualization step of the clipmap itself, based on the observation that a single polygon usually only requires samples from very few clipmap levels and thus it is sufficient to store a *virtual clipmap* with this limited range in the video memory. For this additional virtualization, a new addressing hardware was proposed such that the clipmap system could be implemented in hardware.

A core problem of clipmaps is the tile determination by the center of interest. The spatial relation between the clip center and the mapped tile does not take into account any occlusion and is not even necessarily unambiguous. As a result, a major part of the tiles stored in the video memory is not visible because the corresponding geometry is occluded or outside the view frustum. This and the hardware requirements for the complex addressing system dampened the impact of clipmaps on terrain rendering for games, but the guiding idea of virtualization has been refined ever since. However, a noteworthy implementation of modified clipmaps called *MegaTextures* is Splash Damage's *Enemy Territory: Quake Wars* released in 2007 [Kalra and van Waveren 2008]. Clipmaps have been proven very suitable for geometry virtualization as *geometry clipmaps* where vertex buffers rather than textures are used for storage. With geometry clipmaps, terrain deformation [Crause et al. 2011] and adaptive level-of-detail systems [Losasso and Hoppe 2004; Asirvatham and Hoppe 2005] can be designed very efficiently.

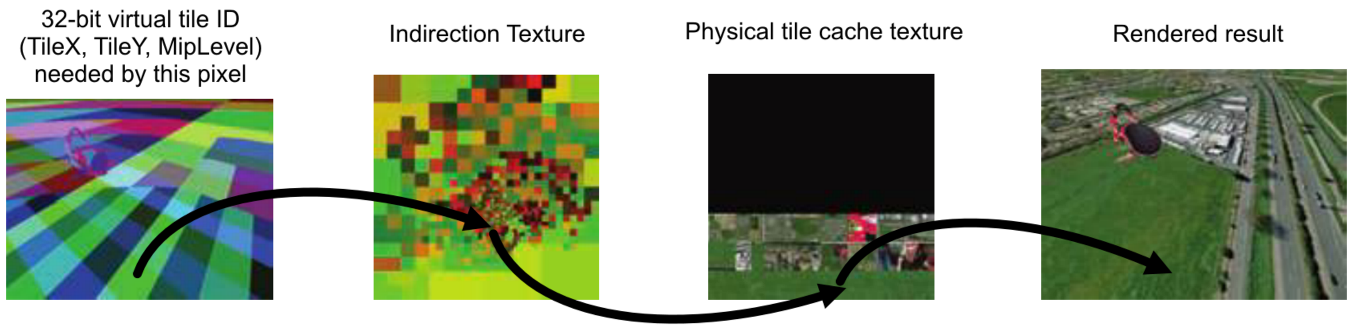

Figure 3: The virtual texturing pipeline. The IDs of all needed tiles are stored in the needbuffer. For rendering, the indirection texture is used to translate virtual to physical texture coordinates. With these, the tiles stored in the video memory can be accessed for texturing. Image retrieved from [Hollemeersch et al. 2010].

3 Virtual Texturing

Clipmaps were introduced at the time when hardware was not programmable and streaming of tiles was expensive due to CPU-GPU bandwidth constraints. This is why the virtualization of textures through clipmaps could not be used for general rendering applications back then. Although the video memory size and bandwidth have increased in today's hardware, this is no solution to the problem as the texture size is still by far too large and tendentially growing. Instead, the solution is to reduce the streaming cost through refined, adaptive tile determination and management processes to fit existing bandwidth constraints. The fundamental steps to achieve this have been sketched by Lefebvre et al. [Lefebvre et al. 2004] and Barrett [Barrett 2008] who provide efficient and complete virtual texturing systems to solve the streaming problem. After splitting the physical texture including its whole mipmap chain into tiles of uniform size, the untextured geometry is rendered once to output the visible texture coordinates of the geometry. These coordinates are stored in a buffer that is read back to the CPU to determine the visible texture tiles, which are then uploaded to the video memory. In a second, final rendering step, the *virtual texture coordinates* of the geometry are translated to *physical texture coordinates* pointing to the uploaded tile in the video memory. For the translation, an *indirection texture* is maintained that stores all physical texture coordinates at the positions of their virtual texture coordinates for a fast lookup.

So, the major differences to clipmaps are an improved screen-space determination of visible tiles and the use of an indirection texture analogous to a page table rather than toroidal addressing based on the mipmap level. It is important to see that, despite the name, virtual texturing is a whole system rather than just a texturing step. The whole pipeline in its current form can be separated into different stages (see Figure 3) which are outlined in this section.

3.1 Texture Creation

Virtual texturing requires a complete mipmap chain divided into uniformly sized tiles that are stored in inexpensive memory. For correct filtering at the tile borders, additional data has to be stored for each tile. Although a new tool chain for splitting and merging is needed to create these tiles, it offers several advantages compared to texturing methods mentioned before. Artists or users have the possibility to work with the texture as a whole or just with parts of it. This also allows the use of already existing textures without modification and, in contrast to clipmaps, is not limited to large-scale textures. Current virtual texturing systems are not updated using spatial information in texture space but visibility information, so any textures can be merged together to a huge texture atlas, avoiding frequent texture changes when rendering. With this, *unique texturing* as used in Rage (see Section 3.5) is possible, meaning that every surface of a scene can be textured uniquely with a single texture.

Mayer [Mayer 2010] devotes a lot of attention to further topics of the texture creation which cannot be covered in this work. These include efficient UV unwrapping for unique textures, packing of tiles in a texture atlas, optimal tile sizes, data reduction and texture compression.

3.2 Determination of Visible Tiles

To determine which tiles have to be stored in the video memory, the first step of the system is to analyze which tiles are (potentially) visible in the current frame. Also, a prediction of which tiles will be needed in the succeeding frames is desirable to efficiently precache them. To get the visibility information of the scene, it is rendered to a target that can be read back to the CPU in a first pass. The read back is usually a bottleneck of the system, which is why acceleration techniques are crucial for a high performance.

3.2.1 Generation and Analysis of Visibility Information

For the determination of the visible texture regions, Lefebvre et al. [Lefebvre et al. 2004] propose a two-pass system where the geometry is rendered to a *texture load map* in texture space in the first pass. This is a map of the texture space that contains visibility information of all tiles. Then, tiles are probably being used if a part of the geometry was written to the corresponding pixel of the texture load map. To determine the tiles exactly, numerous subsequent level-of-detail calculations are needed to address the right tile or tile area. This approach allows a conservative tile determination, because occlusions are not taken into account. In practice, it is no longer used for tile determination because of the high amount of necessary calculations and the less-than-ideal results and is thus not discussed in detail.

A more natural and exact two-pass approach proposed by Barrett [Barrett 2008] is to render the visible geometry in screen space as usual to the *needbuffer* [Neu 2010] in the first pass. Instead of applying the real texture to the geometry, however, the texture coordinates as well as the estimated mipmap level are written to the needbuffer (see the leftmost image of Figure 3 for an example). This information, the *tile ID*, unambiguously identifies each tile of the virtual texture. Since a tile is usually required by several fragments - especially for linearly mapped terrains -, the result can be stored in a buffer smaller than the actual display size, thus reducing both rendering and read-back effort. This is because the needbuffer has to be read back to the main memory to analyze the data on the CPU, which induces additional data traffic through the whole system. In this analysis, a list of all required tiles is created and then handed to the management process. An important point stated by Mayer [Mayer 2010] is that the manual mipmap estimation in the shader, done with the *dFdx* and *dFdy* functions, does not necessarily produce the same result as the mipmap selection by OpenGL, since it is not closely specified. For cases like this, the new OpenGL extension *ARB_texture_query_lod* can be used which returns the result of the OpenGL mipmap selection as if a texture lookup had been performed. The new shader function provided at fragment level is not only more accurate, but also faster than the manual mipmap level calculation.

A drawback of both two-pass solutions is that all geometry has to be processed twice which causes considerable additional effort. Although the needbuffer resolution can be smaller than the display size and low detail geometry can be used for the first pass, it remains computationally expensive. With *multiple render targets*, doing both the tile determination and the final rendering in a single pass is possible. Then, instead of doing the tile determination for the current frame before rendering it, the set of tiles calculated in the last frame is used. This might not be exact, but in practice, the deviations will be imperceptible because of the temporal coherence between the needbuffers of two succeeding frames. Even more important, the streaming process cannot provide the tiles just required immediately but with a few frames delay, so even the two-pass approach is lagging behind. A more concerning problem is that with multiple render targets, the needbuffer has to have the same size as the main render target, leading to significantly increased data transfer between the GPU and CPU.

3.2.2 GPU-Accelerated Needbuffer Processing

To reduce the read-back delay, Hollemeersch et al. [Hollemeersch et al. 2010] propose a GPGPU method for data reduction with CUDA before downloading the buffer. Although their implementation relies on multiple render targets, this method can be used likewise in the two-pass approach. Since the needbuffer tends to contain a lot of redundant information because of the spatial coherence between neighboring pixels, the idea is to generate a buffer that only contains unique tile IDs. Usually, this buffer is much smaller than the original needbuffer, so the read back is faster. One result of this implementation is depicted in Figure 1, using large-scale aerial photography to reconstruct an island in real-time.

It has to be mentioned that evaluations done by Hollemeersch et al. [Hollemeersch et al. 2010] do not show any difference in performance when using the full or a reduced needbuffer resolution. In contrast, the OpenCL implementation of this method done by Mayer [Mayer 2010] performs significantly better with a lower resolution. This can have several causes, including internal API differences between CUDA and OpenCL as well as hardware generation and vendor limitations. Hollemeersch et al. point out that depending on the hardware, CUDA might either just lock the data of the needbuffer for use or copy it, which of course would increase the effort. Finally, it is noteworthy that the implementation of virtual texturing by Neu [Neu 2010] also uses multiple render targets and therefore the full-sized needbuffer, but does not process the data before the read back. Yet, the presented results suggest an overall high visual quality of the method, which has been evaluated for various existing indoor and outdoor game levels to proof suitability for video games.

3.3 Tile Management and Streaming

The tile management stage is the main component of the texture virtualization pipeline because this is where access to the virtual texture is provided. It receives a list containing the IDs of all tiles needed for the current frame from the previous tile determination stage and ensures their availability. For this, three tasks are performed, namely maintenance of an *indirection texture*, loading and streaming of tiles.

3.3.1 Maintenance of the Indirection Texture

The indirection texture, comparable to a page table, is a structure that translates virtual texture coordinates into physical ones, which is necessary because the tiles in the physical video memory are stored in a tile cache. This tile cache is a large texture subdivided into addressable frames of the same size as tiles. Since the tile cache is not large enough to store all tiles of the virtual texture, the contents will change constantly depending on the currently visible surfaces. It is the task of the indirection texture to keep track of all these changes. Therefore, each tile of the virtual texture including its mipmap chain is represented in the indirection texture by one entry. For the rendering step, *entry* means texel because a physical texture or a texture array [Mittring 2008] is generated. However, as the structure is needed in both the management stage and the rendering, it might also be represented by a quad-tree on the CPU side. The quad-tree also has to store state information for each tile such as if it is already loaded. The position of a tile in the virtual texture corresponds to its position in the indirection texture, which is why the tile ID stored in the needbuffer can be used to directly look up the value of the indirection texture for this tile.

For each tile requested by the tile determination stage it is checked if it already resides in the tile cache. If this is not the case and it is not currently being loaded or streamed either, streaming of it has to be initiated. All tiles to be streamed are stored in a *priority queue* which is constantly updated according to specified criteria such as the distance to the view point or the frequency of requests for this tile. The tile priority can be seen as a measure for the visual impact of this tile or its absence on the scene. Generally, tiles of lower resolution have a higher priority because they can be used as fallback textures, which leads to a progressive increase of details all over the scene at runtime. If a conservative tile determination or a tile prediction is implemented, these additional tiles can be pre-cached as well if there are spare capacities. Neu [Neu 2010] devotes much attention to various page priority heuristics and offers detailed evaluations of the visual quality using Structural Similarity as a measure. Additionally, he introduces the concept of a *Look-Ahead Camera* used to track the current camera motion and predict the tiles needed in the next frames.

3.3.2 Loading of a Requested Tile

Before the tile of highest priority can be streamed, it has to be loaded from the secondary storage. Doing this in near real-time is a non-trivial task due to the variety of hardware configurations a developer has to consider as target system. In a simple scenario, all tiles residing on the hard disk drive are loaded into the main memory and are then ready to be streamed. Even in this case, the diversity of memory interface, size and speed as well as storage device type (HDD, solid-state drive) and interface is huge. A minor improvement would be to create a second caching level in the main memory to pre-load neighboring tiles of recently loaded once in advance to lessen the impact of a slower hard disk on the system.

Tiles might also be stored on portable or optical storage devices. In the case of video games for consoles, tiles might have to be loaded directly from a DVD which performs quite poorly. Not to speak of the limited storage capacity of DVDs that made it a necessity to store the game Rage and the heavily compressed texture data on three dual-layer DVDs that have to be swapped at runtime. To optimize reading from optical devices, larger chunks have to be read sequentially, thus influencing the choice of the tile size [Mittring 2008]. A deeper insight into loading terrain textures from slow storage devices is given by van Waveren [van Waveren 2008].

Yet another and particularly promising way of data acquisition is over Internet. Mittring already presented the idea of streaming whole game levels for multiplayer games as an alternative to optical devices. An actual realization of streaming over Internet has been provided recently by Andersson and Göransson [Andersson and Göransson 2012] for a WebGL implementation of virtual texturing. It is imaginable that for portable devices, virtual texturing with network streaming will be used for tasks such as displaying satellite photos in GPS navigation services. In such cases, a near real-time streaming to the video memory is usually not a priority, so delays due to the network transfer of tiles and the limited computational power to process them are bearable.

3.3.3 Compression and Encoding of a Tile

After the tile has been loaded into the main memory, depending on its compression (e.g. JPEG, PNG), it has to be decompressed. Benchmarks by Mayer [Mayer 2010] show significant performance differences between different compressions and decompressing libraries which have to be taken into account. Generally, JPEG offers both higher (lossy) compression in the secondary storage and faster decompression compared to PNG. After decompressing the tile, it is usable by the GPU, but relatively large in size. The streaming delay is usually the bottleneck of the virtual texturing pipeline, so a texture compression that is supported by the GPU such as DXT is suggested [Hollemeersch et al. 2010; Mayer 2010]. With DXT, a significant reduction of the upload bandwidth and video memory consumption can be achieved. A promising addition to the GPGPU utilization for virtual texturing suggested in the contributions of both Hollemeersch et al. and Mayer is texture transcoding to DXT on the GPU. Instead of decompressing the tiles on the CPU, then recompress them to DXT and streaming them, the compressed (JPEG) tiles can be transcoded from their current encoding to DXT on the GPU. Since JPEG offers a much better compression ratio than DXT, the upload bandwidth can be further reduced. The *GPU transcode*, however, recently fell out of favor when used in Rage to accelerate streaming, at which it occasionally failed (see Section 3.5).

3.3.4 Streaming of a Tile

Streaming itself is done asynchronously in an own thread that continuously checks for loaded and possibly CPU transcoded tiles. If such a tile exists, a frame in the physical tile has to be found. In the simple case, the tile cache is not yet full and the tile can be uploaded to one of the cache's empty frames. Once the cache is full, it has to be determined whether to replace one of the existing tiles in the cache or to discard the already loaded tile. The latter should happen when all cached tiles are currently needed and each of the tiles has a higher priority than the recently loaded one. To keep track of visibility information and e.g. the last query of a tile for *least recently used scheduling*, the additional tile attributes can be stored in the CPU-side representation of the indirection texture.

If the tile passes this last test, it is finally uploaded to the physical memory. Then, the indirection texture has to be updated accordingly. The entry representing the new tile has to be set such that it stores the position of the tile cache frame it resides in as well as its original mipmap level which is already stored in the information retrieved from the needbuffer. If another tile has been replaced for the new one, *all* entries of the indirection texture pointing at this tile cache frame have to be changed to point to the frame of the most appropriate fallback tile for the replaced one. This is the tile of the next lower virtual mipmap level that covers the area of the previous tile and resides in the cache. Thus, uploading the lowest mipmap level tile covering the whole virtual texture in a single tile guarantees a fallback tile for each tile at all time. The last step of the management stage is to upload the changes of the CPU-side indirection texture to the GPU. Figure 3 shows an example of an indirection texture. It can be seen that the outer, currently not visible regions of the virtual texture are mapped to a few low-resolution tiles, apparent through the large homogeneous areas.

3.4 Rendering

In the last stage of the pipeline, the generated data, namely the tile cache texture and the indirection texture, has to be used to texture the visible geometry correctly. As the per-vertex texture coordinates of the geometry correspond to the location in the virtual texture, a translation step comparable to the address translation of clipmaps is needed. Instead of using toroidal addressing, a lookup into the indirection texture is used, which is depicted in Figure 4.

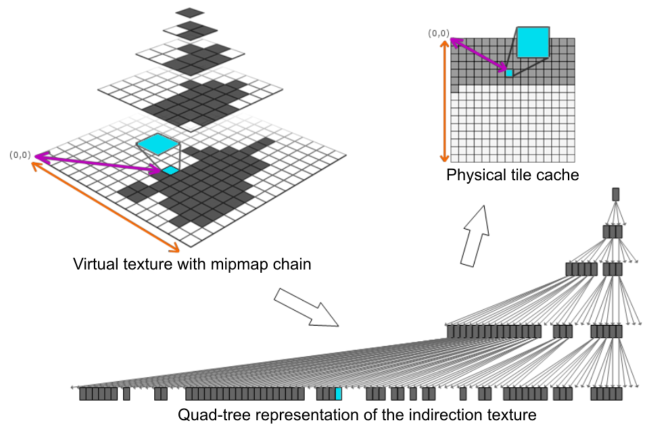
(0,0)
Physical tile cache
(0,0)
Virtual texture with mipmap chain
Quad-tree representation of the indirection texture

Figure 4: Translation from virtual to physical texture coordinates. Image retrieved from [van Waveren and Hart 2010].

3.4.1 Address Translation with the Indirection Texture

The correct texture coordinates for the physical texture depend on the position of the tile in the tile cache that covers the relevant region as well as the internal offset within this tile. Since each texel of the indirection texture corresponds to a tile of the virtual texture, an unfiltered lookup can be performed with the interpolated per-fragment texture coordinates at fragment stage. This fetch already returns the tile position in the cache because this is exactly what the indirection texture stores. Now it should also be clear why the fallback tiles are propagated down to all children in the quad-tree representation at indirection texture updates. It guarantees that for each virtual texture coordinate, a tile in the cache can be found, even if it is of low resolution.

Calculating the internal offset is a bit more difficult as it is relative to the virtual tile position and thus dependent on the original mipmap level of the tile. This is why the mipmap level has already been written to the indirection texture when streaming the tile. The mipmap level can be stored as the third component of the indirection texture and is fetched together with the physical tile position. The virtual texture coordinates then have to be biased and scaled according to the mipmap level. An exemplary derivation of the solution is presented by van Waveren and Hart [van Waveren and Hart 2010] (see Figure 4). However, shader implementations with constraints such as quadratic power-of-two textures can use arithmetic tricks to perform the translation with very few instructions [Barrett 2008; Mittring 2008; Hollemeersch et al. 2010].

With the physical texture coordinate retrieved, a second lookup into the tile cache texture can be performed to get the texture information. From then, rest of the rendering is done as usual. At this point it should be stated that the format or content of the tiles can be arbitrary. Thus, all types of data such as normal, specular, displacement, light and shadow maps and - although of questionable practicability - even indirection textures [Mayer 2010] can be virtualized and used in the shader. If the tile cache is represented by a texture array rather than a 2D texture, it is easy to store tiles of different attributes for the same surface at the same x and y coordinates in the texture array but on different layers. Then, the texture coordinate translation has to be done only once for a lookup in all desired layers.

3.4.2 Texture Mapping with Filtering

When sampling a texture, filtering is of course desirable. Relying on the hardware filtering, however, leads to artifacts at the tile borders. Residing in the tile cache, the tile is surrounded by arbitrary tiles, so all information needed for bilinear or anisotropic filtering has to be included in the tile itself. Doing the filtering manually in the shader is possible, but requires to translate the texture coordinates for each sample, leading to an unfeasible overhead [Neu 2010]. This is why in Section 3.1 it has already been stated that the tile creation tools need to append borders for correct filtering, as the neighborhood of a tile border has to be taken into account for the filtering. However, this either reduces the usable area of the tile or increases the tile size, both leading to potentially higher upload bandwidth for the sake of visual quality. When using DXT encoded tiles, it is necessary to append full 4 x 4 texel blocks to the border for correct filtering [Mittring 2008].

Trilinear filtering generally requires a second mipmap level to sample from. Barrett [Barrett 2008] proposes the use of a physical tile cache with one additional mipmap level to store the cache contents in half size, which is sufficient because virtual texturing by design already samples from the best matching level. This enables the usage of the graphic API's trilinear filtering, but is, according to Mittring [Mittring 2008], an unnecessary increase of video memory consumption (by a quarter) that does not pay. The reason for this is that the half-resolution data created for the mipmap level already exist somewhere in the system, maybe even in the tile cache. Thus, Neu [Neu 2010] suggests to rely on the API's bilinear filtering and to do the trilinear filtering manually in the fragment shader. For this, twice the amount of texture lookups into indirection texture and tile cache are needed. Additionally, the tile management has to ensure that the required tile of next-lower resolution is cached, too, so a slightly modified tile determination algorithm requests an additional direct parent tile of each visible tile. This increases the streaming effort and the number of tiles needed in the tile cache, but a portion of the additional tiles will already be cached as fallback textures.

3.4.3 Reduction of Pop-In Artifacts

Although at this point the virtual texturing system should behave as if the large-scale texture was used with common texture mapping, one deviation from the virtual ground truth is hard to eliminate. The whole pipeline is designed for asynchronous updates such that it never stalls on uploads or read-backs. Thus, whenever a task is not yet finished and the rendering starts, it has to handle those unfinished jobs, which is what the fallback tiles are used for. The faster the pipeline performs, the faster all textures are available to the shader for the desired output. A last question is how to behave when the streaming of a new tile has finished. If the new tile is used for texturing straight away in the next frame, the higher detail of the tile will pop in suddenly in a disturbing manner. It is similar to the pop in of level-of-detail algorithms, which is dealt with by forcing a smooth transition of the levels at runtime, e.g. by blending the results of both levels.

In virtual texturing, blending can be used easily if trilinear filtering is used, since this already implements gradual blending of textures. Depending on the time since the arrival of the new tile, the lower-resolution tile is weighted manually to force the smooth transition to the higher detail [Mayer 2010]. For blending with bilinear filtering only, van Waveren and Hart [van Waveren and Hart 2010] propose dithered updating of the tile cache. This means that each frame, only a part of the new tile is uploaded and combined with the lower-resolution tile. For this, the region of the lower-level tile that is covered in the new tile is upsampled and stored in the tile cache. Then, this tile is gradually updated with portions of the higher-resolution tile each frame until the new contents completely replaced the old ones. This, of course, again leads to more streaming and delayed streaming of other tiles, so a good compromise has to be found.

3.5 Recent applications

In the preceding sections, the complete design of a state-of-the-art virtual texturing system including acceleration techniques has been presented. The structure is mainly based on the contributions of Mayer [Mayer 2010] and Neu [Neu 2010], which both date back to 2010. Since then, virtual texturing has become a valuable tool for the terrain rendering in large-scale applications, including but not limited to video games [van Waveren and Hart 2010; Widmark 2012] such as the popular video game Rage. This section is devoted to these recent developments. The aim is to give an overview over specific applications of the presented system in the already wide field of texture virtualization. A particularly interesting topic is the virtualization of general data such as geometry or volume data.

3.5.1 id Software's Rage

Since its announcement in 2007, the video game Rage by id Software has been eagerly anticipated because of the use of texture virtualization. In contrast to the clipmap-like *MegaTextures* implementation used for terrain texture virtualization in the games Enemy Territory: Quake Wars and BRINK by Splash Damage, Mega-

Textures have been extended to a complete virtual texturing system for Rage [van Waveren 2009]. Right after the release in 2011, however, a lot of PC customers had severe troubles with the rendering and especially the virtual texturing in the game. Besides tearing and glitches that showed wrong tiles scattered all over the screen, the main causes of artifacts were tile popping and the generally slow streaming of tiles [Burnes 2011]. In many cases, only fallback tiles were shown and obviously never replaced.

For this report, the game has been tested at 1920 x 1200 pixels screen resolution on a system including a GeForce GTX 480 video card, Intel i7-950 CPU, 12 GB of main memory and a solid-state drive. With this, a 180 degree turn in game took approximately *five seconds* to replace all low-resolution fallback textures. Additionally, even the tiles of higher resolution looked blurry. To fix these problems, a game patch as well as a reference to console commands that change virtual texturing behavior such as the tile cache size were published. After applying both patch and configuration changes, the visual quality of the rendering increased drastically to the point where no more popping artifacts or fallback textures were visible on the test system.

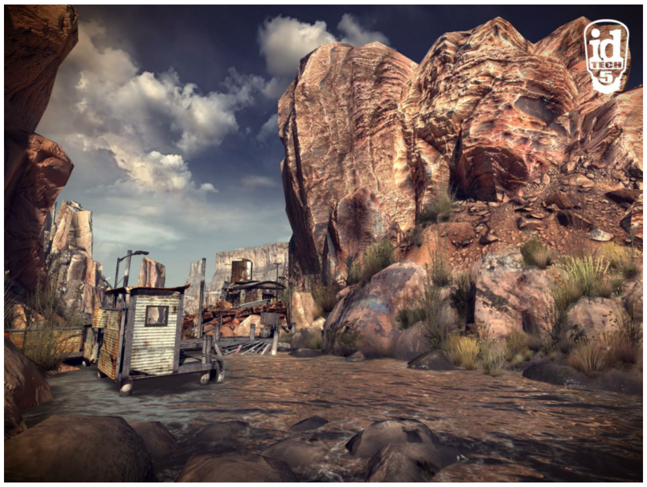

Figure 5: Screenshot of the virtually textured terrain in Rage as presented at the SIGGRAPH '09. Image retrieved from [van Waveren 2009].

According to Burnes [Burnes 2011], the cause of the initial performance issues is a biased adaptive quality regulation for PC systems. On the console platforms the game ran fine from the start - despite the higher loading latency from DVDs as stated in Section 3.3, because the hardware is uniformly specified and the system could be tweaked to fit the hardware requirements. In contrast, the variety of PC hardware combinations made it a necessity to dynamically adapt parts of the pipeline to the hardware at runtime to a predefined upper level to prevent side effects. This also includes GPU transcoding. As mentioned before, the purpose of GPU transcoding is to take work load off the CPU by decompressing loaded textures on the GPU. Obviously, this has a huge impact on the overall performance in Rage, so with the patch, a possibility is given to limit the number of GPU transcodings per frame or to disable it completely. Rage seems to use the GPU to capacity already so that the hardware acceleration proposed by Hollemeersch et al. [Hollemeersch et al. 2010] in fact slows down the system. This might be due to the fact that JPEG-like decompression is hard to parallelize and thus cannot benefit substantially from a GPU implementation.

Rage can be considered a milestone in texture virtualization not only because of its general extent and performance, but because it also virtualizes the small-scale textures for unique texturing. The ambitious aim to use a unique texture for each piece of geometry in the scene can partially be blamed for the game's bumpy start since the amount of tiles that have to be streamed each frame is massive. It is all the more surprising that id Software finally managed to implement it at an interactive frame rate and overall impressive quality (see Figure 5). Yet, while this allows all textures to be treated in the same way, the visual quality does not benefit very much from it in Rage compared to recent games with conservative texture mapping for smaller objects. The benefit of texture virtualization for the obviously large terrain textures with regards to the high level of detail compared to multi-texturing approaches, on the other hand, is beyond doubt.

3.5.2 Virtual Texturing for WebGL

Besides the gaming segment, virtual texturing will certainly be of increasing importance for geographical data visualization. Especially online and mobile services for mapping, positioning and navigation could greatly benefit from the accelerated display of aerial images. That the implementation of virtual texturing is generally possible on mobile platforms has been proven by Andersson and Göransson [Andersson and Göransson 2012] with their WebGL implementation. An interactive frame rate in this implementation, however, could not always be achieved. The predominant issues of WebGL and OpenGL ES 2.0 implementations tend to be the restrictive feature set and unsatisfactory browser support. As mentioned by the authors, the loading of DXT compressed images is still not possible and with the lack of pixel buffer objects, a readback of the needbuffer or upload of a tile stalls the whole thread until completion. However, it should only be a matter of time before these features are supported. Currently, GPGPU acceleration is also impossible through WebGL. The API, WebCL, is already under development though, so some of the tweaks presented might be applicable in WebGL soon.

3.5.3 General Data Virtualization

The rise of virtual texturing linked to Rage also triggered the research and development of acceleration methods. Thus, increasing attention is devoted to data virtualization in general. As with clipmaps, the concept of virtual texturing can be extended for geometry virtualization. One recent contribution by Sugden et al. [Sugden and Iwanicki 2011] is the *Mega Meshes* system - in analogy to MegaTextures - which combines sculpturing tools and virtual texturing in game development. In the content creation, enormous geometry data is stored in a hierarchical structure like a quad-tree with several subdivision levels that can be streamed. At runtime, texture data according to the subdivision level such as normals and ambient occlusion data is used in a virtual texturing system.

Virtualizing volume data, which is commonly represented by 3D textures, has been introduced by Crassin et al. [Crassin et al. 2008] under the term *GigaVoxels*. In 2011, Crassin proposed a complete system for volume virtualization usable for e.g. biomedical applications and voxel-based game engines [Crassin 2011]. This is particularly interesting because voxels allow to overcome the (artificial) separation of geometry and surface texture. Thus, geometry and texture streaming could be unified, although content creation would get much more expensive for such an engine.

A contribution by Mayer et al. [Mayer et al. 2011] presents the combination of virtual texturing and point cloud rendering for cultural heritage preservation. The point cloud, however, is not virtualized in this application but relies on another proven method of out-of-core rendering. Still, this shows the direction in which real-time rendering is heading.

3.5.4 Hardware-Implemented Virtual Texturing

As a result of this uprising, partial hardware support for the virtualization pipeline seems to be usable in the near future. Video cards using AMD's Graphics Core Next architecture such as the Radeon HD 7970 offer a redesigned fragment shader stage as well as support for *partially resident textures* [Bilodeau et al. 2012] through the *AMD_sparse_texture* extension. With these it is possible to define a large-scale texture with only partial allocation. For the use in virtual textures, they could make indirection texture and tile cache obsolete, as all visible tiles can be stored right in the sparse texture. If the virtual texture coordinates can be used to access the partially resident texture through an internal texture coordinate translation, the whole virtualization step can be done by the hardware. For tile determination, shaders have been modified such that lookups to a partially resident texture fetch the stored data - if present - and additionally return a fetch state. If no data is stored at the lookup location, a *Fail* is returned that can be read back to the CPU, indicating that the corresponding tile is not yet available and needs to be streamed. A second state is the *LOD warning* which according to a predefined level-of-detail limit for a texture indicates that a higher-resolution representation of the current tile might be needed soon, thus offering hardware supported tile prediction. Even tasks like the filtering are re-simplified with this extension. Future implementations of virtual texturing using partially resident textures will show if the extension holds up to its promises.

4 Conclusion and Outlook

Due to their properties, terrains can benefit greatly from the current aerial and satellite image acquisition techniques. They extend over the whole scene, so a huge amount of data is required to texture them in a satisfactory way. This exceeds the capabilities of current hardware in both texture size and total memory consumption. Based on the observation that only a fraction of the texture data and its mipmaps is needed to texture one frame at runtime, different approaches have been suggested to detect and upload only this information. Resorting to the virtual memory management of operating systems, the use of clipmaps for texture data virtualization has been introduced, which evolved into the more flexible and efficient virtual texturing. The virtual texturing pipeline as presented in this report relies on a multi-threaded system that is divided into three independent stages. A main benefit of this system compared to previous methods is the tile determination in screen space, providing an exact solution for the tile visibility problem.

Several possibilities for quality improvement and acceleration at different stages of the virtual texturing in both software and hardware have been discussed, although their benefit for general applications is yet to be evaluated. Especially the hardware support for partially resident textures in AMD video cards gives hope that virtualization in general and virtual texturing in particular is made applicable for the masses. Until now, implementing virtual texturing is a complex task that only pays off for large-scale applications. What is even harder is to get the single components of the pipeline to work together in an efficient manner. The difficult start of virtual texturing flagship Rage has shown that the interaction of these components requires a lot of fine-tuning and leaves room for more robust solutions in the future. A point to consider game-dependently is if unique texturing of small object is useful and pays off with respect to the streaming overhead compared to static textures. The insight that the GPGPU acceleration of texture decompression can have a negative impact on the performance should lead to a more careful work load distribution over all components in such dynamic systems. One additional deficiency that has to be pointed out is that WebGL and OpenGL ES do not seem to be fit for real-time virtual texturing yet. With the lack of features already common in OpenGL, virtual texturing on mobile devices is one or even two steps behind. For the aimed-at application on these devices, however, an interactive frame rate might not be needed.

Considering the quality achievable with unique large-scale textures that can be composed by artists as a whole, virtual texturing outperforms current terrain texturing methods. With its exact tile determination and the possibility to minimize artifacts through suitable tile priority calculation, it is also superior to other current texture streaming methods. This is why it is safe to assume that the virtualization of all kinds of data will be continued and adopted in future rendering systems.

References

ANDERSSON, S., AND GÖRANSSON, J. 2012. *Virtual Texturing with WebGL*. Master's thesis, Department of Computer Science & Engineering, Chalmers University of Technology, Gothenburg.

ANDERSSON, J. 2007. Terrain Rendering in Frostbite Using Procedural Shader Splatting. In *Proceedings of ACM SIGGRAPH '07 course notes, course 28, Advanced Real-Time Rendering in 3D Graphics and Games*, ACM, 38–58.

ASIRVATHAM, A., AND HOPPE, H. 2005. Terrain Rendering Using GPU-based Geometry Clipmaps. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, 27–45.

BARRETT, S., 2008. Sparse Virtual Textures. http://silverspaceship.com/src/svt/, retrieved Apr 14, 2012.

BILODEAU, B., SELLERS, G., AND HILLESLAND, K. 2012. Partially Resident Textures on Next-Generation GPUs. Game Developers Conference.

BURNES, A., 2011. How To Unlock Rage's High Resolution Textures With A Few Simple Tweaks. http://www.geforce.com/whats-new/articles/how-to-unlock-rages-high-resolution-textures-with-a-few-simple-tweaks/, retrieved Apr 15, 2012.

CRASSIN, C., NEYRET, F., AND LEFEBVRE, S. 2008. Interactive GigaVoxels. Tech. Rep. RR-6567, INRIA.

CRASSIN, C. 2011. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, University of Grenoble.

CRAUSE, J., FLOWER, A., AND MARAIS, P. 2011. A System for Real-Time Deformable Terrain. In *Proceedings of the SAICSIT '11*, ACM, 77–86.

EPIC GAMES, 2012. Terrain Advanced Textures. http://udn.epicgames.com/Three/TerrainAdvancedTextures.html, retrieved May 13, 2012.

HOLLEMEERSCH, C.-F., PIETERS, B., LAMBERT, P., AND VAN DE WALLE, R. 2010. Accelerating Virtual Texturing Using CUDA. In *GPU Pro - Advanced Rendering Techniques*, W. Engel, Ed. AK Peters, 623–642.

KALRA, A., AND VAN WAVEREN, J. 2008. Threading Game Engines: QUAKE 4 & Enemy Territory QUAKE Wars. Game Developers Conference.

LEFEBVRE, S., DARBON, J., AND NEYRET, F. 2004. Unified Texture Management for Arbitrary Meshes. Tech. Rep. RR-5210, INRIA.

LOSASSO, F., AND HOPPE, H. 2004. Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. In *Proceedings of ACM SIGGRAPH '04*, ACM, 769–776.

MAYER, I., SCHEIBLAUER, C., AND MAYER, A. J. 2011. Virtual Texturing in the Documentation of Cultural Heritage. *Geoinformatics FCE CTU*.

MAYER, A. J. 2010. *Virtual Texturing*. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology.

MITTRING, M. 2008. Advanced Virtual Texture Topics. In *ACM SIGGRAPH '08 classes*, ACM, 23–51.

MULTIMEDIA LAB, 2011. Demos - Multimedia Lab. Ghent University. http://multimedialab.elis.ugent.be/demonstrations, retrieved Apr 1, 2012.

NEU, A. 2010. Virtual Texturing. *CoRR abs/1005.3163*.

NVIDIA CORPORATION, 2007. Clipmaps whitepaper. http://developer.download.nvidia.com/SDK/10/direct3d/Source/Clipmaps/doc/Clipmaps.pdf, retrieved Apr 10, 2012.

SUGDEN, B., AND IWANICKI, M. 2011. Mega Meshes - Modelling, rendering and lighting a world made of 100 billion polygons. Game Developers Conference.

TANNER, C. C., MIGDAL, C. J., AND JONES, M. T. 1998. The Clipmap: A Virtual Mipmap. In *Proceedings of ACM SIGGRAPH '98*, ACM, 151–158.

VAN WAVEREN, J., AND HART, E. 2010. Using Virtual Texturing to Handle Massive Texture Data. NVIDIA GPU Technology Conference.

VAN WAVEREN, J., 2008. Geospatial Texture Streaming From Slow Storage Devices. http://software.intel.com/en-us/articles/geospatial-texture-streaming-from-slow-storage-devices/, retrieved Apr 15, 2012.

VAN WAVEREN, J. 2009. id tech 5 Challenges - From Texture Virtualization to Massive Parallelization. ACM SIGGRAPH '09 – Beyond Programmable Shading Course.

WIDMARK, M. 2012. Terrain in Battlefield 3: A modern, complete and scalable system. Game Developers Conference.