FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Virtual Texturing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Albert Julian Mayer
Matrikelnummer 0126505

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Wien, 14.10.2010

(Unterschrift Verfasser/)

(Unterschrift Betreuer)

Abstract

Virtual texturing (as presented by Mittring in 'Advanced Virtual Texture Topics' and in distinction to clipmap-style systems, to which this term is also applied) is a solution to the problem of real-time rendering of scenes with vast amounts of texture data which does not fit into graphics or main memory. Virtual texturing works by preprocessing the aggregate texture data into equally-sized tiles and determining the necessary tiles for rendering before each frame. These tiles are then streamed to the graphics card and rendering is performed with a special virtual texturing fragment shader that does texture coordinate adjustments to sample from the tile storage texture.
A thorough description of virtual texturing and related topics is given, along with an examination of specific challenges including preprocessing, visible tile determination, texture filtering, tile importance metrics and many more. Tile determination in view space is examined in detail and an implementation for compressing the resulting buffer in OpenCL is presented. Rendering with correct texture filtering from a texture which contains de-correlated texture tiles is attained by using tile borders with specific coordinate adjustment and gradient correction in the fragment shader.
A sample implementation is described and serves to provide results concerning performance and correctness with different settings and architecture choices. Integration into Open Scene Graph for usage within a hybrid point-cloud / polygonal renderer enables rendering of high resolution paintings within catacombs modeled with point clouds. Another application is presented, the real-time display of a highly detailed model of New York with more than 60 GB textures.
Quantitative analysis reveals that frame-rates above 200 FPS are attainable on complex scenes with multi-million polygons even with outdated hardware. At the same time quality remains high, results indicate that "fallbacks", that occur when a needed texture tile is not ready in time, occur only for 0.01% of the pixels on average. These results show that virtual texturing can be a competitive solution for games, scientific and industrial applications, allowing for real-time rendering of scenes that could not be displayed previously, while maintaining acceptable visual quality.

Kurzfassung

Virtual Texturing (so wie von Mittring in 'Advanced Virtual Texture Topics' präsentiert und im Gegensatz zu Clipmap-artigen Systemen, welche ebenfalls genauso bezeichnet werden) ist eine Lösung für das Rendern von Szenen mit enormen Texturdaten die nicht in den Graphik- oder Hauptspeicher passen. Virtual Texturing funktioniert indem die gesamten Texturdaten in gleich große Teile vorverarbeitet werden und die benötigten Teile vor jedem Frame festgestellt werden. Diese Teile werden dann zur Graphikkarte gestreamt und das Rendern findet mit einem speziellen Fragmentshader statt, der eine Koordinatenanpassung vornimmt, um von der Teilspeichertextur zu samplen.
Eine detailierte Beschreibung von Virtual Texturing und verwandten Themen wird gegeben, ebenso eine Untersuchung von spezifischen Herausforderungen, sowie das Herausfinden von den sichtbaren Teilen, Texturfilterung, Metriken für die Teilgewichtung und viele mehr. Das Bestimmen von sichtbaren Teilen im Bildraum wird detailliert untersucht und eine Implementation für das Komprimieren des resultierenden Buffers in OpenCL wird präsentiert. Das Rendern mit korrekter Texturfilterung von einer Textur, die unkorrelierte Texturteile beinhaltet, wird durch Teilränder und spezifische Koordinatenanpassung sowie Gradientenanpassung im Fragmentshader ermöglicht.
Eine Beispielimplementation wird beschrieben und liefert Resultate bezügliche der Performance und der Korrektheit mit verschiedenen Wahlmöglichkeiten und Architekturentscheidungen. Integration in den Open Scene Graph für die Benutzung in einem hybriden Punktwolken / polygonalen Renderer ermöglicht die Darstellung von hochauflösenden Malereien in Katakomben, die mit Punktwolken modelliert sind. Noch eine weitere Anwendung wird präsentiert, die Echtzeitdarstellung eines detaillierten Modells von New York, das mehr als 60 GB Texturdaten beinhaltet.
Quantitative Analysen zeigen, dass Frameraten jenseits von 200 FPS auch bei komplexen Szenen mit Millionen Polygonen selbst mit veralteter Hardware möglich sind. Die Tests zeigen, dass dabei auch die Qualität hoch bleibt, das Zurückfallen auf niedrigauflösende Teile, welches vorkommt wenn die Benötigten nicht rechtzeitig verfügbar sind, kommt durchschnittlich nur bei 0.01% der Pixel vor. Die Resultate belegen, dass Virtual Texturing eine praktikable Lösung für Spiele, sowie wissenschaftliche und Industrieanwendungen ist. Virtual Texturing erlaubt die Echtzeitdarstellung von Szenen, die vorher nicht dargestellt werden konnten, mit akzeptabler visueller Qualität.

Contents

Chapter 1

Introduction

Computer graphics is a growing field that by now most people are exposed to daily, due to the ubiquity of electronic devices and digital media. Texture mapping is an integral part to many computer graphics applications. The problem of ever increasing texture data and limited texture memory is solved by virtual texturing, the topic this thesis is dedicated to.

1.1 Background & Motivation

Real-time 3D surface rendering today is nearly exclusively done by polygon rasterization. Objects are approximated by meshes built out of polygons and the rasterization converts the polygons to a raster image, i.e., the output pixels on screen. Texture mapping is a ubiquitously supported method of adding details by projecting images representing the surface color onto the polygons. However, real-time rendering usually can only be performed if the geometry and the textures representing a scene fit into the finite graphics memory of the graphics card.

Virtual texturing is a solution that allows real-time rendering of scenes that contain texture maps that exceed available graphics and even main memory, and therefore constitutes a so-called out-of-core rendering solution. There has been a lot of research on out-of-core rendering, but most of it has been focusing solely on the geometric side, and nearly all of those resources that address textures have been restricted to terrain (or at least planar) scenes. This problem has remained unsolved up until recently, when programmable GPUs with adequate performance began to enable a solution in the form of so-called *virtual texturing*. The importance of a solution to this problem cannot be overstated because the texture data requirements for real-time applications are rising, not only due to the improved resolution of e.g. aerial imaging and scanning devices but also because of the continuing trend to convert geometric details to texture data ("normal mapping") as well as the desire for visually convincing virtual worlds. While interest in virtual texturing seems to be very high, there are still few public resources on

this subject, and most of them are relatively informal in nature. The most informative resource on this subject to date still remains Sean Barrett's "Sparse Virtual Texturing" webpage [Bar08], which features a video and a sample implementation concerning virtual texturing. There is not a single scholarly paper published in a scientific magazine on this topic yet. High demand combined with little "supply" were part of the motivation for working on virtual texturing, but the elegantly simple nature of this solution for a sophisticated problem and the endless interesting possibilities it enables were just as important. It is our hope that this thesis, eliminates the lack of resources around this topic and serves as a comprehensive reference.

1.2 Aim & Scope

The aim of this thesis is to evaluate the feasibility of virtual texturing for different use cases (with respect to performance and correctness), to be the most comprehensive resource on virtual texturing, to provide a detailed examination of all basic components and issues and to supply empirical data (e.g., benchmarks) for all tradeoffs that have to be made while developing a virtual texturing system. Additionally, we aim to present related ideas as well as our specific implementation (LibVT) and a compelling real-world use-case (New York scene).

This thesis assumes familiarity with computer graphics principles and programming matters related to real-time 3D rendering, including knowledge of 3D graphics APIs and shader development.

The scope of this thesis is virtual texturing and all directly related subjects. The scope includes all components that are common to ordinary virtual texturing implementations as well as solutions to universal problems like filtering. This thesis also includes some indirectly related subjects, connected ideas and matters that are only of importance to some use-cases, however, it does not try to be exhaustive in this area. Put differently, the aim of this thesis is not to include every possible idea that is somehow connected to virtual texturing or may be useful only in a subset of the virtual texturing applications.

This thesis tries to strike a middle ground between a high-level scientific description and a focus on practical applicability. The guideline followed here is that we provide exact details where we deem them to be non-obvious and important but resort to a more elevated explanation in cases where details are considered apparent. For instance we do provide example shader code for virtual texturing because it is considered an integral but not evident part, however, we do not provide any low-level details for developing a page-loading thread because that should be obvious to any programmer.

It is not the aim of this thesis to provide a thorough examination of all virtual texturing matters that are of importance only to a subset of the use-cases, e.g., the gaming

industry. While we do provide some insights for the applicability of virtual texturing in games, these are by no means detailed or exhaustive, and only cover basic issues and not supplementary ideas. Additionally, we try to avoid covering non-essential issues in detail if they have already been handled by other virtual texturing resources. See Section §2.5.5 for complemental virtual texturing resources and particularly [MG08] for an examination from a game-developer point of view.

1.3 Structure

This thesis is structured into the following chapters:

1. Introduction:
   This very chapter.
2. Related Work:
   This chapter evaluates the related work that enables usage of large texture datasets including texture compression, texture synthesis and texture caching and provides a general view of all resources on virtual texturing to date.
3. Overview:
   Provides an outline of the virtual texturing system to ease the understanding of the more detailed main chapter.
4. Virtual Texturing:
   This is the main chapter of this thesis and covers virtual texturing and related issues.
5. Implementation & Results:
   Our implementation, our real-world test scene and results from our tests are presented here.
6. Conclusion:
   This chapter concludes this thesis by providing a summary of our findings and provides suggestions for future research areas.

1.4 Contribution

The contributions made during this thesis can be categorized into the following items:

- Detailed explanation of virtual texturing, categorized into its three distinct parts: tile determination, tile streaming and the virtual texturing shader.
- Examination of all problems and challenges related to virtual texturing including thorough investigation of their solutions.
- Delivery of hard data through tests and benchmarks to quantify most of the trade-offs and choices to be made in a virtual texturing implementation. Data has also been collected for facts which are only indirectly related to virtual texturing, e.g., the relation between file size and file loading time.
- Research of novel ideas related to virtual texturing of varying importance. Some ideas have also been developed independently in parallel with other current virtual texturing efforts (e.g., the GPGPU buffer reduction idea and the first corresponding kernel). Other examples of novel ideas are presented under the terms “tile importance” and “tile request substitution”.
- Implementation of a complete and configurable library for virtual texturing in OpenGL applications called “LibVT”. The universal integrate-ability of the library has been proven by integrating it into two different rendering engines.
- Development of a pipeline for converting (OBJ file based) scenes to virtual texturing, including a texture atlas tool and a tool that generates the virtual texture tile store on disk.
- Processing, conversion and import of a real-world 3D scene (“New York”) with high geometric complexity and exorbitant high texture requirements for a first-time display of the combined scene in real time, serving as a proof of concept.

1.5 Methodology & Setup

The methodology used during this thesis is to generate knowledge by creating and testing a virtual texturing implementation. More specifically:

1. Basics:
   A basic virtual texturing is implemented to learn about the three main components of a virtual texturing system and their interaction.
2. Problems:
   The main problems and difficulties for virtual texturing systems are solved within our sample implementation to completely understand the problems and their solutions.

3. Options:
   Many configurable options are implemented within our test system to be able to quantify the impact and tradeoffs of different choices within a virtual texturing system (tile-size, PBO usage, fallback entries vs. shader looping, read-back performance & correctness, etc).

4. Data:
   For options and choices that are difficult to quantify within a virtual texturing system, independent benchmark applications are developed. Examples include: tile de/compression libraries, coupled & decoupled tile loading, texture sampling methods, etc.

First priority during the tests is given to repeatability, only repeatable results are published. For example, during tests that include hard disk access, repeatability is ensured by disabling the page cache (caching of files by the OS) and clearing the disk buffer (caching of files by the hard disk). Results are generated from 5 test runs (or more) and the charts include the standard deviation as error bars. The tests that continuously track values over time are also performed multiple times to ensure repeatability, but the resulting graph is picked from a single run. Tests involving real-time rendering are performed with view-frustum culling, but without other forms of culling.

All tests are performed with this computer setup:
Mac Pro Quad-core 2006 (2 x Intel Xeon DP 5150 @ 2.66 GHZ, 7 GB DDR2 ECC FB-DIMM, NVIDIA GeForce 8800 GT 512MB graphics card, Seagate 7200.12 1TB hard disk, Mac OS X 10.6)

The only exception is the benchmark testing the time to read files of varying file sizes, which is performed with a “Hitachi Travelstar 7K320 320 GB”.

Chapter 2

Related Work

This chapter aims to provide an overview of the history of texture mapping with special emphasis given to the support for very large texture maps and reveals the progression to the current state of the art, i.e., virtual texturing. The chapter starts with a short introduction to texture mapping and then examines different approaches for handling large texture data sets, namely compression, synthesis and multiple caching variants. The chapter concludes with an examination of the current state of the art with respect to virtual texturing.

2.1 Fundamentals

Texture mapping, a technique to add surface details without adding geometric complexity, was pioneered in 1974 in [Cat74] and could be called the most successful idea in computer graphics. It was already used in the first computer animations in the eighties and is now in ubiquitous use as an integral part of real-time rendering, CGI, computer gaming, visualization and many more. Texture mapping also expanded its scope beyond representing only the surface colors by introducing additional texture maps for properties like normal vector perturbation, specularity, transparency, light, shadows, and diffuse and environmental reflection [Hec86]. Texture mapping in real time was pioneered by computer games in the early nineties [Wik10b] and the inclusion of texture mapping as an integral part of the first consumer 3D graphics accelerators in the late nineties [Wik10a] further validated the technique and led to the fact that texture mapping today is an integral part to nearly all computer graphics applications. While there has been a lot of texture mapping research for example in the areas of parametrization and (pre)filtering [Hec86], this thesis focuses on texture mapping with vast data sets that do not fit completely into memory. Unfortunately most of the research on out-of-core rendering to date has been focused solely on managing geometric complexity, [GM05] provides an overview over common techniques and [CESL+03] is an example of the increasingly involved solutions.

Texture mapping works by projecting an image, called texture (and composited of "texels" instead of pixels), onto the geometric primitives to add colors and simulate more geometric detail. To be able to position, orient and scale the textures onto the geometry in exactly the desired way some form of "texture parametrization" is necessary. The most common way to do texture parametrization is called "UV mapping", here each vertex of the 3D object receives additional parameters "u" and "v" that index into the two-dimensional image-plane of the texture. Setting up this mapping between a three-dimensional object (in object space x/y/z) to a two-dimensional texture (in "texture space " u/v) is also called "UV unwrapping". Two goals during unwrapping are minimizing wasted space in the texture that is not projected onto the model and minimizing the distortion of the texture.

During rendering with texture mapping, "texture filtering" takes place. If the textured object is so close to the virtual camera position that each texel maps to multiple pixels of the screen "texture magnification" happens. "Magnification filtering" can be either off (i.e., use the value of the nearest texel, GL_NEAREST in OpenGL terms) or bilinear, that means the texture contribution for each screen pixel is determined by the weighted average of the four nearest texels (GL_LINEAR in OpenGL terms). Texture filtering usually refers to "minification filtering" though. Here the object is further away and therefore multiple texels should contribute to a single pixel on screen. However, calculating the average from a large number of texels every time this minification happens would not be possible with sensible performance. Because of this a technique called "mipmapping" is performed. The texture is not only stored in its full resolution on the graphics card, but also in half, quarter, 1/8, ... 4x4, 2x2 and 1x1 resolution versions. These multiple differently sized versions of the same texture are called the "mipmap-chain". The base version in full resolution is called "mipmap-level 0", the half-resolution version is mipmap-level 1, and so on. The total memory requirements for every texture are increased only by one third by this technique. During rendering either the closest matching "mipmap-level" is selected (called 'level of detail selection"), or the two closest ones are selected and the result is linearly interpolated, which is called "trilinear filtering".

One problem during hardware-accelerated real-time rendering is that changing the current texture is an expensive process (also because it requires breaking up the draw calls) [Cor04]. The "texture atlas" solution to this problem is to pack multiple textures into a single larger texture, and use this larger texture instead [Cor04]. This permits rendering more geometry with a single call and requires less texture switches. However, adjustment of the texture coordinates is obviously necessary and there can be some problems related to texture filtering [Cor04]. We refer to the individual textures that have been assembled to construct a larger texture as "sub-textures". Packing a texture atlas should follow some constraints to prevent a problem called "mipmap-chain pollution", which occurs when the downsampling to generate the mipmap-chain produces

a texel by downsampling adjacent texels from different sub-textures. Making sure that the sub-textures are power-of-two in size and do not unnecessarily cross power-of-two boundary lines prevents this pollution [Cor04].

2.2 Texture Compression

Texture compression applies the methods of image compression to texture data in order to reduce its memory consumption (and bandwidth requirements). The main difference to image compression is the fact that very fast decompression of arbitrary parts of the texture is necessary during texture sampling (optimized random access). Additionally, some textures like normal maps do not exhibit the same visual properties as common images. Texture compression was pioneered in [BAC96], they achieved a 1:35 compression ratio by using vector quantization (but required additional storage for a large "codebook"). The DXT family of compression algorithms built into all recent graphics accelerators is built on a modification of vector quantization but only achieves a compression rate of 1:4 - 1:8 [Wei04]. The new generation of texture compression formats built into graphics cards, namely BPTC (BC6/BC7) only achieves 1:3 - 1:6. However, programmable GPUs theoretically allow using compression formats that are not directly supported in the hardware.

In [KE02] compression through adaptive resolution and exploitation of free space is proposed, but the technique mainly works in higher dimensions and has problems with filtering and mip-mapping.

[LD07] describe a novel data structure for texture parametrization that works by storing square texture tiles into the leaves of an octree surrounding the surface – a method that natively supports adaptive resolution.

Although texture compression reduces the memory footprint for given textures, it does so by a constant factor and as such is no complete solution for the usage of large texture sets, but should be seen as a complementary aid. Adaptive resolution is a useful but also limited memory-savings technique. For one it only applies to single large objects (since distinct objects can be textured with different resolutions anyway) that exhibit strong varying texture resolutions. Additionally, one can work around the problem by texturing the object with multiple textures with different resolution or with special UV-unwrapping. It should also be noted here that virtual texturing also supports limited adaptive resolution properties, see Section §4.1.6.

2.3 Texture Synthesis

Texture synthesis could be seen as a more extreme form of texture compression that takes advantage of the structural content, therefore allowing the compression ratio to approach infinity [Wei04]. [Wei04] categorizes texture synthesis algorithms into three distinct classes:

- So-called example-based techniques that synthesize arbitrarily large textures from small image samples. [WLKT09] notes several example-based synthesis methods, including pixel-based synthesis and patch-based texture synthesis. Most methods are too slow to execute in real time and therefore do not result in memory savings because the resulting textures have to be precomputed [Wei04].
- Methods that work by generating texture coordinates instead of pixels [Wei04].
- Procedural texture generation algorithms that are suited to real-time generation of textures (e.g., on the GPU) but are limited to specific repetitive texture classes like wood, marble, etc [Wei04].

Texture tiling: Texture tiling is a special form of texture synthesis that uses one or few small texture tiles to generate a larger texture. The most simple form is repeating a special texture with seamless opposing edges. Although the results are often visually insufficient because the repetition is readily apparent, this is the most common form of texture synthesis. Support for this method of texture synthesis is even built into every graphics card and activated is by setting the texture wrap type to “repeat” or “mirror”. This simple scheme is also the most common way to circumvent the graphics memory limitation in games: the designers build their game worlds with multiple blended repeated base texture layers (some can be of lower resolution to break the tiling look) and by adding detail textures [MG08]. The aim of this process is to simulate a uniquely textured world with few textures that fit into memory.

There are more complex texture tiling methods that use multiple base tiles and give better results even with a single layer. [LN03] combines patterns procedurally using an indirection texture. [SD03] introduces stochastic tiling using Wang tiles with interesting results, an approach that is later implemented for programmable GPUs in [Wei04].

Although texture synthesis can approach infinite compression and therefore seems to solve the memory consumption problem even for scenes with arbitrarily detailed texture requirements, it cannot be classified as a complete solution to rendering with very large texture maps. There are several reasons:

- Many texture synthesis techniques cannot be performed on the GPU and therefore do not actually result in memory savings because the textures have to be precomputed.
- Texture synthesis mainly works for generating surface color textures.
- Texture synthesis can only generate some image-classes.
- Most importantly: texture synthesis is no drop-in solution that can convert a given scene to another scene with synthesized textures that looks exactly the same.

So, texture synthesis is a special-case solution, but it can still be useful as a complementary technique within a virtual texturing solution. For example, when creating a large uniquely textured scene, it is practically impossible to create each pixel from scratch. Texture synthesis can be used for the initial texturing of the scene from which to improve with detail textures and more variation. Secondly, texture synthesis could be used in collaboration with virtual texturing, e.g., to lessen the performance requirements by having some parts of the scene covered with synthesized textures. In any case, the programmability and performance of modern GPUs enables many more texture synthesis methods than simple repeating of a single tile, a fact that is largely ignored, e.g., game worlds still consist of simple tiling and blending. However, it remains doubtful if a complex scene with multiple layers of synthesized textures gives higher performance (let alone visual quality) than a full-blown virtual texturing solution where all the layers can be combined, including high-quality precomputed lighting.

2.4 Graphics Memory Virtualization

The Windows Vista Display Driver Model (WDDM) and the Direct3D 10 graphics API provide "virtualized" video memory to applications [Cor06]. This is a more general technique for using resources on the GPU that are too large to fit into the graphics memory, but of course it applies to textures too. The method works similar to the ubiquitous main memory virtualization: if the data exceeds memory, it is paged out to a slower storage medium (the disk in the standard case and the main memory in this new case) and paged back in on demand. The method also allows using a texture data set that is larger than main memory (only) because of the fact that main memory is already virtualized. Although certainly useful, this technique has several severe drawbacks in the use case of rendering scenes with a vast amount of texture data, therefore it cannot be considered a (complete) solution.

- Although details on this technique are very sparse (even within the Direct3D 10 SDK) we expect it to be much slower than (texture) specific solutions that are

realized within the application, since the GPU is stalled on a “page fault” until the required data is paged in. The newer version Direct3D 10.1 allows the GPU to continue performing other work even after a “page fault”, but the render command responsible for the fault is still stalled until “page-in”. In contrast, virtual texturing allows applications to continue rendering at full speed even if some texture tiles are missing, and uses lower resolution fallback textures in the interim time.

- Relying on this technique is not feasible for texture data sets that exceed main memory in a renderable format (i.e., uncompressed or DXT/BPTC compressed), since this is only supported due to the fact that main memory is also virtualized. This is prohibitive from a performance standpoint. Also by its very nature this method requires converting the whole texture data to a renderable format (i.e., uncompressed or DXT/BPTC compressed) before rendering, which will result in the mentioned memory oversubscription. In a dedicated texture streaming solution like virtual texturing, only the necessary texture parts can be decompressed on demand from an highly compressed format like JPEG , resulting in a disproportionally lower memory requirement.
- The restriction to applications using Direct3D 10 on PCs running Windows Vista with Direct3D 10 capable GPUs is not desirable.

2.5 Texture Streaming & Caching

Even when applying texture compression, the texture data may exceed the available memory, but only a very small subset of the aggregate texture data is actually necessary at any point in time. Depending on filtering, 1 - 32 texels are needed for every pixel on screen. Texture streaming takes advantage of this fact by allowing rendering with only a subset of the whole texture data being stored on the graphics card. The difference between texture streaming and texture caching, which of course does streaming too, is the granularity: while texture streaming operates on whole textures (or mipmap-levels), caching works on parts of whole textures.

2.5.1 Texture Streaming

Texture streaming is a facility mostly used in computer games to reduce memory usage. Instead of loading all textures at once, they are loaded and unloaded on a per-need basis. The difficult part is determining when to load or unload the textures. Simple systems subdivide the scene into separate distinct areas and (un-)load textures according to predefined lists when entering areas or buildings. This subdivision of the scene into smaller

sub-scenes could be classified as avoiding instead of solving the problem. According to [MG08], texture streaming is becoming a necessity in games. Texture streaming can be done on a per-mipmap basis for some finer granularity to increase the memory savings, but according to [MG08] this is basically impossible to do on the PC without introducing stalls. One drawback of texture streaming is performance problems with creating and destroying textures at runtime [MG08], this can be avoided by reusing identically sized textures.

2.5.2 Texture (Tile) Caching

Splitting the texture (or better, the whole mipmap-chain) into tiles allows texture streaming on a much finer granularity and permits higher memory savings (i.e., more output-sensitivity). The largest drawback of these systems has been the restriction that either the geometric primitives had to be aligned with tile boundaries, forcing unwanted tessellation, or expensive clipping and masking had to be used [CNF+07]. This restriction has been lifted with the arrival of programmable graphics hardware, giving new importance to the old fundamental work. Both clipmapping and virtual texturing are based on texture tile caching. Besides operating on tiles the major difference to (previous) texture streaming solutions is that texture tile caching methods automatically determine the needed texture tiles.

[Cos94] pioneered real-time terrain rendering using high-resolution textures and first described the (pre-)tiling of the texture (called mosaic by him).

[CE98] mentioned that most texture caching solutions at that time were application-specific solutions while some others were not real time capable. They developed a texture caching solution that calculates (a superset of the) necessary tiles of the "quadtree MIP map" (as they call the pre-tiled mipmap-chain) by means of a geometric computation per polygon [CE98] – and caches just these tiles. Polygons must be split according to the texture tiling. Their system allows them to treat the texture as a bandwidth-limited resource instead of a finite resource [CE98].

2.5.3 Clipmapping

Clipmapping is a solution to allow the usage of very large textures on terrain meshes while only using a small part of the texture (the "clipped" portion) at runtime. [Cor07] defines a clipmap as a "partial representation of a mipmap pyramid which holds all information needed for texturing at every single frame". Since terrain is mostly flat (two-dimensional) and linearly mapped with a single texture, we can define a "center of interest" in texture space which is directly dependent on the world space position of

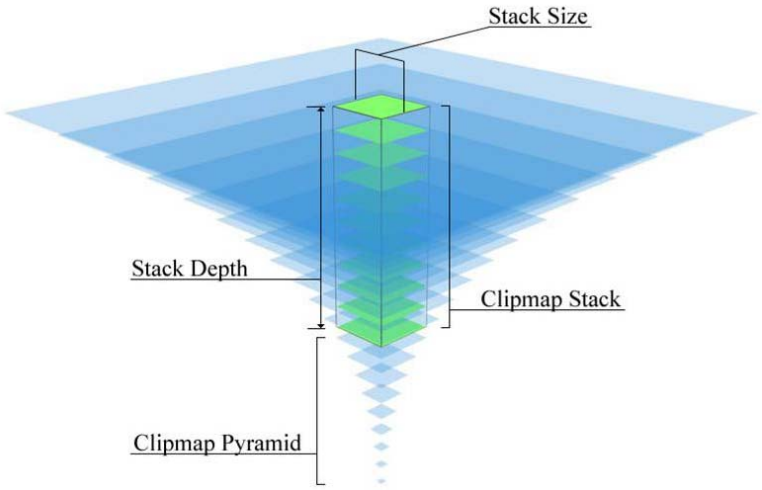

Fig. 2.1: Graphical explanation of the clipmap system. Image from [Cor07].

the virtual camera. The needed resolution of the texture decreases in concentric circles around this center of interest since these areas map to smaller screen space projections by virtue of being further away, as seen in Figure 2.1. So, the highest resolution data (mipmap-level 0) is only loaded for a quadratic portion around the center of interest. The size of this square is called the "stack size" and is chosen to approximate a 1:1 mapping of texels to pixel area [Cor07]. From higher mipmap-levels similarly sized portions are loaded, but each of them covers a four-fold increased world-space region. When the center of interest moves, a clever updating mechanism called "toroidal addressing" decreases the amount of work to be done.

Clipmapping was pioneered at SGI, first mentioned in 1996 and later described in [TMJ98]. The initial implementation required specialized graphics hardware available from SGI, but several years later the increasing programmability of GPUs paved the way for implementations on commodity hardware, and that is where later research headed [Cor07] [SLT+07] [CNF+07]. Other research on clipmap-like systems has been done adapting them for non-color data [EC06] and dynamic texture updates [TSH09].

[Hüt98] invented an alternative to clipmaps called MP-grids, also specialized for terrain rendering. In contrast to clipmaps, they tile the whole texture into *several* mipmap-pyramids, and required texture parts are determined by a geometrical test instead of specifying a center of interest in texture space [Hüt98]. They also proposed modified hardware to prevent the necessity for real-time clipping, but their modifications have not been incorporated.

2.5.4 MegaTexture

In 2005, id Software announced a computer game that would use a new technology called "MegaTexture", allowing the usage of a high-resolution $32k^2$ pixel texture on the terrain [iS05]. Facts on the implementation of this technology were scarce and it was only in 2007 after the game had shipped that it was confirmed to be just an implementation of a clipmap-like system for commodity hardware using the fragment shader [CC05]. However, only shortly after announcing "MegaTexture", John Carmack announced his intent to revise the technology to work on arbitrary geometry [Car05] and in 2006 he confirmed that a newer version (later called "MegaTexture v2") already worked on any geometry [DC06]. Again there were few details on the exact implementation, nevertheless it spurred a lot of interest in virtual / unique texturing and eventually it was confirmed in 2009 [vW09b] that "MegaTexture v2" is essentially virtual texturing as described and implemented by others since that time.

2.5.5 Virtual Texturing

Virtual texturing generalizes previous approaches by adding a level of indirection, i.e., a pagetable, similar to virtual memory subsystems in modern operating systems. The pagetable (texture) allows each fragment to access a suitable tile depending on the required resolution according to standard OpenGL mipmapping calculations, instead of relying on heuristics like distance to viewpoint (as used in clipmapping). Put differently, the major distinction to previous approaches is that virtual texturing allows "level of detail selection" per-pixel, instead of per geometric primitive. Incidentally, this also lifts the restriction of previous approaches to terrain rendering and the required tile-aligned geometry tessellation. The different methods vary in the way they detect and handle "page faults". New features of virtual texturing include rendering all geometry in one call, correct filtering (see Section §4.5.2) and handling of dynamic geometry. The indirection in the fragment shader is facilitated by sampling the "pagetable texture" for coordinate translation, as first mentioned by [KE02] in a slightly different context.

The first virtual texturing system (in our sense) was developed by 3Dlabs and built into their graphics cards as a way to support client applications with large textures [Sem99]. The system split textures into 4KB (32^2 pixel) pages and fetched these pages on-demand with a page-fault DMA engine from main memory without CPU intervention [Sem99]. An on-chip virtual memory management unit (MMU) provided for the necessary address translation. In distinction to later virtual texturing systems the system was not able to fall back to lower resolution pages during rendering but introduced unwanted stalls. This proprietary extension is similar to SGIs support of clipmaps, with the difference that clipmaps only work with terrain meshes and required special application support. Unfortunately this support has not been implemented by other vendors

and 3Dlabs eventually ceased their GPU business, but, just as with clipmapping, programmable GPUs allow re-creation of the system on arbitrary hardware.

[LDN04] were the first to describe a modern virtual texturing system that works on arbitrary geometry, but their work received little attention. Their system already looked largely similar to later virtual texturing implementations, with the notable difference being their novel idea of doing tile determination by rendering to UV space.

Interest in virtual texturing grew in and after 2006 when John Carmack reported having a system capable of real-time rendering a uniquely textured world [MG08], however he did not provide technical details (see Section §2.5.4). This eventually spurred Sean Barrett to independently recreate a virtual texturing system and he presented his system at GDC 2008 as "Sparse Virtual Texturing" and provided his sample implementation under public domain as well as a video explaining virtual texturing [Bar08]. The video and the implementation already provided answers to many important questions surrounding virtual texturing, including fast & correct filtering. Many details and developments around virtual texturing were eventually published on the Sparse Virtual Texturing Forum [Spa10] maintained by him.

Martin Mittring from Crytek presented "Advanced Virtual Texture Topics" at SIGGRAPH 2008 and a similarly named chapter appeared in the accompanying class course book [MG08]. Mittring expanded on Barrett's work and provided new insights regarding tile determination (see Section §4.2), virtual texture atlas layout (see Section §4.5.6), mesh parameterization, efficient texture updates (see Section §4.3.2) and mipmap generation. He also gave special insights of importance from a gaming industry standpoint (Direct 3D implementation details and "combo textures").

Also in 2008, "Making Art Studios" released some informal details on their virtual texturing implementation [Stu08].

At SIGGRAPH 2009, J.M.P. van Waveren from id Software provided new details about their virtual texturing system, confirming that it is especially similar to Barrett's re-creation as well as providing some new ideas about LoD bias adaption (see Section §4.3.2) and LoD snap prevention (see Section §4.5.4) [vW09b]. Unfortunately only the slides of the presentation are publicly available, somewhat limiting the use of this resource.

Additional informal resources that appeared during 2009 are the sample virtual texturing implementation with source code from Brad Blanchard [Bla09] and the weblog of Sander van Rossen that features notes of interest concerning his implementation [vR09].

Finally, in 2010 the book "GPU Pro: Advanced Rendering Techniques" was published, containing two chapters about virtual texturing. [HPLdW10] provided insight how to speed up virtual texturing using CUDA (see Section §4.2.1) and [CESL10] presented an introduction to the subject as well as some new insights. Since these chapters

have been developed during the same timeframe and were published around the same time as this thesis, they contain some similar (but independently developed, unless otherwise noted) insights.

The latest resource on virtual texturing is the bachelor thesis of Andreas Neu which includes a comprehensive overview of the subject, novel ideas that aim to improve the visual quality as well as quality assessment methods [Neu10].

Chapter 3

Overview

This chapter addresses the terminology used throughout this thesis and provides a short introduction to "virtual texturing" and its challenges.

3.1 Terminology

A terminology mismatch currently exists in the field of texture caching. Some papers apply the term "virtual texturing" to all systems that use a texture that only partly resides in memory ("virtual texture") and specifically to clipmap-like systems [TSH09] [EC06] [Wei04] [SLT+07]. Other papers and resources apply the term to a newer and considerably different method for large texture support that works on arbitrary geometry [MG08] [CESL10] [HPLdW10]. Barrett has referred to the new virtual texturing as "sparse virtual texturing" for differentiation, a term that has not achieved universal acceptance. This mismatch can be resolved either by inventing a new name for the new method to distinguish it from the old technique (as Barrett tried), or by applying the term "virtual texturing" only to the new method. We chose to use the second option (also because of a lack of a more fitting name for the new technique), i.e., **in this thesis the term "virtual texturing" is restricted to apply only to systems that work on arbitrary geometry**. There is no need to call clipmaps "virtual textures", we can just call them by their original name: "clipmaps". This also makes sense insofar as the new technique exhibits similarities to the fundamental concept of "virtual memory", while the old method does not.

Additionally, it should be noted that the few resources that exist on virtual texturing do not use a common terminology. Table 3.1 is a translation table for some of the most important terms. We chose to use a terminology very similar to [Bar08], which is in contrast to the terminology of [MG08] and [LDN04]. We also refer to the tiles of the virtual texture as "tiles" in contrast to the term "pages". The reason that we call the translation texture "pagetable texture" although we are referring to "tiles" and not "pages" is that the "pagetable" is an established term in computer sciences.

Tab. 3.1: Different terminology of virtual texturing terms.

This Thesis	Mittring [MG08]	Hollemeersch et al. [HPLdW10]	Lefebvre [LDN04]
Physical Texture	Tile Cache	Physical Page Texture	Tile Pool
Pagetable Texture	Indirection Texture	Page Table Texture	Indirection Grids
Tile Determination	Computing Local LOD	Page Resolver	Texture Load Map Comp.

3.2 Outline

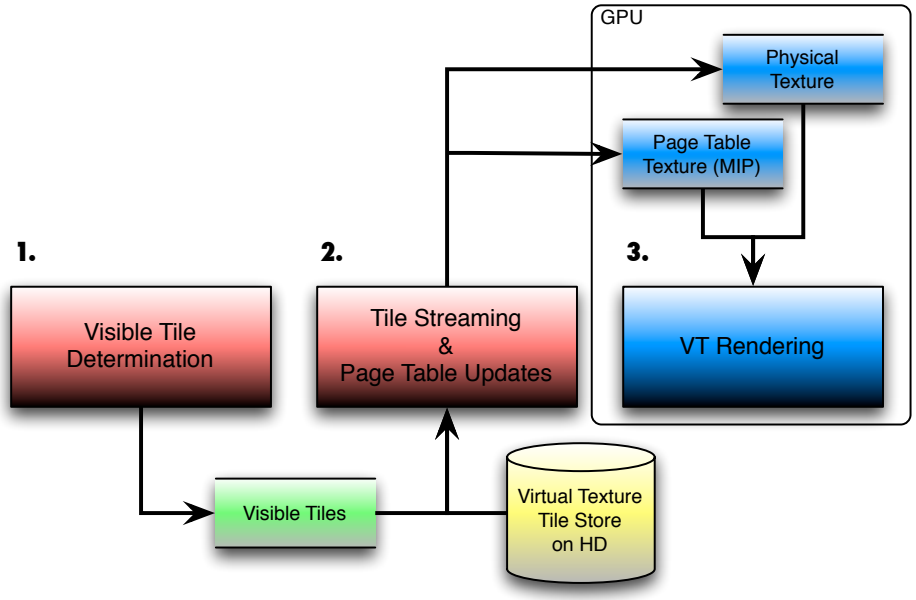

Fig. 3.1: A simplified virtual texturing system.

Virtual texturing is a texture tile caching method, meaning it determines the currently needed tiles of the aggregate texture data, streams these tiles to the GPU and enables rendering from just these needed tiles. Virtual texturing requires all textures of the scene to be combined into a single very huge texture. As a preprocessing step this texture and all its mipmap-levels are tiled into equally sized tiles. At runtime a virtual texture system has three main tasks, as outlined in Figure 3.1.

- Tile determination:
  Determining which tiles of the virtual texture are visible from the current virtual camera position and orientation and therefore necessary for rendering.

- Tile streaming:
  The necessary tiles have to be loaded from their storage location and streamed to a storage texture on the GPU (“physical texture”).

- Virtual texturing shader:
  A special fragment shader uses a texture (“pagetable texture”) that contains the coordinates of all tiles in the physical texture to transform the virtual texture coordinates to the physical coordinates used to sample the right tile at the right position.

The main advantage of virtual texturing compared to older texture caching methods is the indirection in the virtual texturing fragment shader, which enables rendering from a small tile cache texture without forcing unwanted tessellation, or expensive clipping and masking. This is enabled by programmable graphics cards with dependent texture fetches. Previous methods were restricted to terrain rendering and tile-aligned geometric primitives [TMJ98]. The other main benefit of virtual texturing over the previous state of the art is a vastly improved method for efficient determination of the needed tiles – previous methods often relied on geometric computations per polygon [CE98], which is not feasible given the amount of polygons in current scenes. A third important property is that the system does not stall while tiles needed for rendering are streamed in. Virtual texturing continues rendering at full speed and uses lower resolution fallback tiles in the meantime.

3.3 Detailed Overview

Figure 3.2 provides a more detailed overview over the main parts of a virtual texturing system.

The virtual texture and its whole mipmap-chain are stored in pre-computed, equally sized tiles. If the virtual texture is not only used to texture a contiguous mesh, it will consist of multiple textures for different objects, thus essentially being a very large texture atlas [Cor04]. In addition to the generation and pre-tiling of the virtual texture, the geometry has to be textured just as if the virtual texture really existed in memory. Therefore we will name these texture coordinates “virtual texture coordinates”, since they refer to the virtual texture. Offsetting the texture coordinates has to be done to compensate for the combination of multiple textures in a single larger texture as with any texture atlas [Cor04].

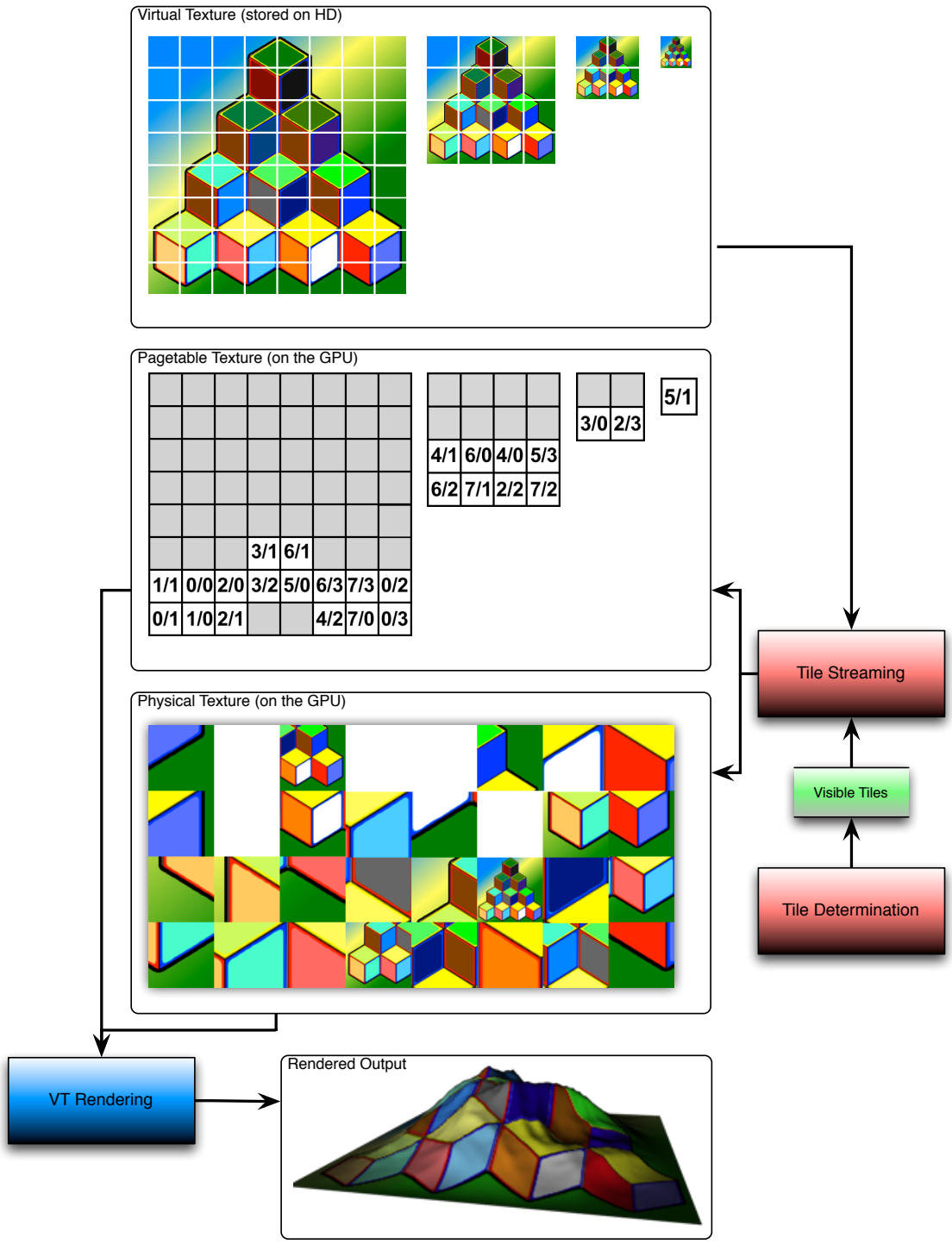

Fig. 3.2: An overview over virtual texturing rendering.

- Tile determination:
  This subsystem determines which tiles of the virtual texture are visible from the current virtual camera position and orientation and therefore necessary for rendering. This is usually done with a pre-pass that is rendered with a special shader that outputs the address of the visible tile at each pixel. The results of this pre-pass are read back to the CPU.

- Tile streaming & texture updates:
  The necessary tiles have to be loaded from their storage location (e.g., the hard disk) and streamed to a storage texture on the GPU, called the "physical texture". The tile streaming system consists of one or more background threads that continuously load and decompress requested tiles. The texture updating operates on the main thread, just before rendering the main pass the tiles that have been loaded so far are streamed to the physical texture. The "pagetable texture" is updated with the translation from "virtual texture coordinates" to "physical texture coordinates". To be more concrete, the pagetable texture has a pixel for every tile in the virtual texture – if the virtual texture is 256^2 tiles at the base level, then the "pagetable texture" is a 256^2 pixels texture. Every pixel in the pagetable texture contains the storage address of the tile it is representing in the physical texture. For tiles that are not currently stored in the physical texture, the address of a "fallback entry" can be stored in the pagetable texture. This just means that the address of the proximate stored lower resolution tile that covers the same texture space is stored instead.

- Virtual texturing shader:
  The virtual texturing fragment shader uses the pagetable texture to transform the virtual texture coordinates (the ones the geometry is textured in) to the physical coordinates used to sample the right tile from the physical texture. Since all virtual textured objects share the same (virtual) texture they can all be rendered with a single draw call. To get the address of the right tile that should be used for rendering, the shader just samples the pagetable texture with the virtual texture coordinates (the ones the geometry is textured in) and an appropriate bias to compensate for the size difference between the pagetable texture and the virtual texture. This texture fetch will yield the address the appropriate tile for rendering in the physical texture, or the address of the next lower resolution tile if the native tile is not currently stored. An alternative to storing these "fallback entries" in the pagetable texture is to adjust the shader to perform the fetch from the pagetable texture with increasing bias in a loop until a valid address is found.

Performing the tile determination, doing the texture updates and rendering the main pass with the virtual texturing shader have to be performed every frame, in this order. The tile streaming runs asynchronously all the time in background threads.

3.4 Challenges

There are two main challenges during the implementation of a virtual texturing system. The first issue is correctness, which means the visual output produced with virtual texturing should be the same as with a theoretical system that supports these large texture sizes . The other issue is performance. Virtual texturing needs to yield performance comparable with normal rendering or other methods for large texture support to be competitive. In practice these two issues are closely intermingled because if the visible texture tiles are not streamed fast enough, lower resolution fallbacks are used, producing visible artifacts. Several other issues constitute a third category of challenges. Note that these challenges have not been known upfront, but only have been progressively identified during the development of the thesis and implementations. Therefore not all of these challenges have been completely solved within our implementation.

3.4.1 Correctness

Precision limitations within GPUs are an important factor affecting the correctness of virtual texturing, especially when increasing the virtual texture size [MG08]. Limitations within the GPU programming model, e.g., the undefinedness of texture fetches inside loops [LK06] are also limiting. Enabling correct texture filtering (or DXT compression) for virtual textures takes even more considerations [vW09b]. Correctness is also affected when necessary tiles are missing during rendering – a virtual texturing system with asynchronous texture streaming is by its very nature prone to these kinds of artifacts. The amount of artifacts depends on the speed of the tile streaming, the ability of the tile determinations system to "look into the future" and the number and divergence of the necessary tiles (see below).

3.4.2 Performance

A virtual texturing system is a complex system and performance problems can arise from individual components being too slow or from the interaction of these components in the pipeline. An example for a performance-critical operation within the virtual texturing system is the tile determination. This is commonly done by rendering the scene in a pre-pass with a special shader that outputs the visible tile coordinates at each pixel [Bar08]. This approach is only feasible if performed at a lower resolution or when compressing the resulting buffer before reading it back to the CPU. The question whether the read-back of the results should be delayed until the next frame demonstrates how the interaction of the components affects performance.

The factor that is affecting runtime performance (and also correctness) the most is actually the virtual texture layout, which is determined in a pre-processing step. The layout of the virtual texture affects how many tiles are necessary during rendering and how divergent this set is from frame to frame.

3.4.3 Miscellaneous

There are a multitude of miscellaneous challenges related to virtual texturing: support for transparent geometry, making artifacts less noticeable, handling non-exact tile determination, coping with low maximum texture sizes of the GPU, tile cache strategies, decal support, the virtual texturing content pipeline, i.e., how to generate virtually textured scenes and the virtual textures themselves, etc.

Chapter 4

Virtual Texturing

This chapter provides a detailed examination of the virtual texturing system as described by [LDN04], [Bar08] and [MG08]. It is possible to build a virtual texturing system that looks quite different from the system described here, and in fact such systems have been proposed and implemented. Garney describes a virtual texturing system that works entirely within the vertex shader, has variably sized tiles and does analytical tile determination [Spa10]. This thesis is restricted to discussion of virtual texturing systems in the style of [Bar08] since this is the only setup where significant details have been published.

At first the basis to virtual texturing is examined, the virtual texture itself. The following three sections each deal with one of the three major distinct parts of a virtual texturing system: tile determination, tile streaming and the virtual texturing shader. The "Problems, Challenges, Advanced Features & Miscellaneous" section deals with different topics related to virtual texturing. Finally, hardware improvements that could help with virtual texturing are investigated.

4.1 The Virtual Texture

A virtual texture is a very high-resolution mipmapped texture that resides only partly in (graphics) memory during rendering [MG08]. The term "virtual" refers not only to the texture not being completely available in memory, but is also chosen in analogy to "virtual memory", which allows transparent access to a virtual address space [MG08]. "Very high-resolution" in this context means that the texture is substantially larger than the largest texture size of a recent GPU. Current virtual texture sizes range from $32k^2$ - $256k^2$ pixels.

The virtual texture and its whole mipmap-chain are tiled into equally sized tiles. Figure 4.1 provides an illustration.

The tiling of the mipmap-chain of the virtual texture stops with a single tile of the

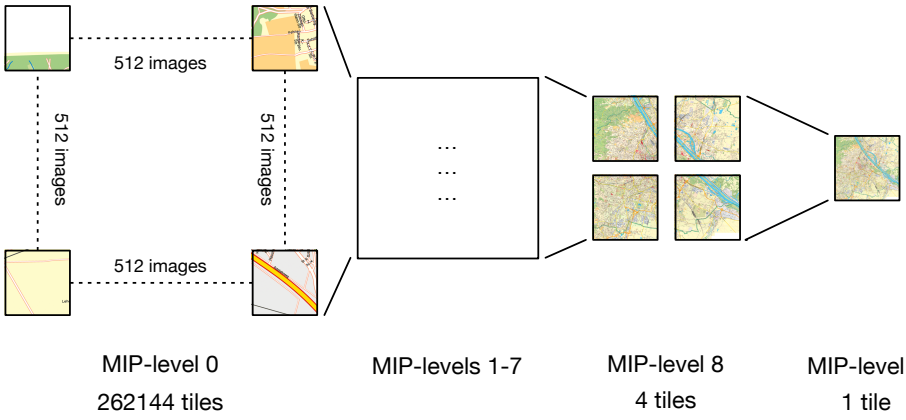

Fig. 4.1: The tiles of a virtual texture with a mipmap-chain length of 10. If the tile-size is 256^2 pixels, then the full virtual texture size is $128k^2$.

given tile-size. There are no tiles stored with a lower than tile-size resolution to accommodate for the highest mipmap-levels. This is in contrast to normal mipmapping and clipmapping, where the highest mipmap-level always consists of a single pixel. [HPLdW10] explains why this does not result in artifacts: it is very unlikely that the virtual texture is ever viewed from so far away that the screen projection falls below the (effective) tile-size. [MG08] also considers this to be a non-issue for most cases and offers workarounds in case they really are required. Since the highest mipmap-level always consists of a singe tile, the length of the mipmap-chain is not only dependent on the (virtual) texture size, but also on the tile-size. Each tile of the virtual texture is uniquely addressable with the tuple (x, y, mip-level).

The tiling of the virtual texture into equally sized tiles is crucial to the virtual texturing system: in principle the virtual texture system has to determine the needed "parts" of the virtual texture and provide them to the virtual texturing shader. This is (only) efficiently possible since the "parts" are precomputed equally sized tiles.

The tiled virtual texture is stored on a hard disk or optical medium and streamed to the GPU at runtime. It is also possible to store the tiles on a server and stream the tiles over the network [MG08], although it has not been demonstrated that the performance of such a setup is acceptable.

Instead of pre-creating the virtual texture, it is also feasible to generate the virtual texture at runtime with procedural generation [MG08] [Bar08]. While still being a virtual texturing system, this would be different from the system described in this thesis, since there is no need to load and decompress virtual texture tiles from the hard disk. Presumably, the scene would still be "textured", but not with the surface color texture map but rather with input for the procedural generation system. Note that the Sparse

Virtual Texture demo [Bar08] does simple procedural generation of the tiles. Unless performance is much better when doing runtime procedural generation it may still be beneficial to pre-calculate the procedural content into tiles and stream them normally, because this allows additional artist refinements, baked light-maps and multiple layers "for free".

4.1.1 Tile-Size

Common tile-sizes range from 64^2 to 256^2 pixels [MG08] [Bar08] [vW09b]. The tile-size is an important factor in a virtual texturing system as it directly determines the amount of pixels needed from the virtual texture to do the rendering. A lower tile-size means that about three times as many tiles but fewer overall pixels are needed. As the tile-size approaches one pixel, the amount of pixels needed in the physical texture approaches the amount of pixels in the viewport. Virtual texturing is therefore an output-sensitive algorithm, since the amount of texture data needed is proportional to the viewport size and not the amount of texture data in the scene. So, decreasing the tile-size decreases the amount of "wasted" pixels in the physical texture, i.e., pixels that have to be there just by virtue of being in a tile that is partly visible. However, decreasing the tile-size also increases the mipmap-chain length and the number of visible tiles. This has the following effects:

- Longer mipmap-chain:
  An increase of the mipmap-chain length by one increases the overall tile count by a factor of roughly 4. To be more exact, the factor is dependent on the current mipmap-chain length n:

$$\frac{\left(4^{(n+1)} - 1\right)}{\left(4^{n} - 1\right)} \tag{4.1}$$

  Since the pagetable texture (covered in Section §4.3.1) has one pixel per tile, its size is also increased by a factor of 4, contributing to (graphics) memory consumption. Considering the common use of tile borders for correct filter (see Section §4.5.2), the "wasted" memory for border-pixels also increases, on hard disk as well as in graphics memory. Updating the pagetable texture also becomes more CPU intensive with a longer mipmap-chain, when fallback entries are used (see Section §4.3.1). A longer mipmap-chain is also undesirable because the maximum coordinates in the identifying tuple (x,y,mip-level) for a single tile increase. When the mipmap-chain length is 9 (e.g., a $64k^2$ pixels virtual texture with 256^2 pixel tiles), there are 256 by 256 tiles at the lowest mipmap-level. This means the (x,y) coordinates of the tiles fit into a single byte each. If the mipmap-chain

length increases, the overflowing bits of the (x,y) coordinates may have to be stored elsewhere, mandating some bit-shifting.

- More visible tiles:
  Having more (but smaller) visible tiles negatively changes the performance characteristics of the tile streaming. Additionally, the increase of operations that have to be done for every new tile has negative implication on the runtime performance. Examples include: finding an empty slot in the physical texture, updating the pagetable (texture), updating data structures and updating the RAM cache.

Figure 4.2 and 4.3 show the relation between the tile-size and the necessary physical texture size. Results can be below zero because some parts of the screen are not covered by virtually textured objects, i.e., the skybox is using a traditional texture. The unfavorable results from the New York scene can be attributed to a problematic virtual texture layout of this particular scene, see Section §5.2.1.

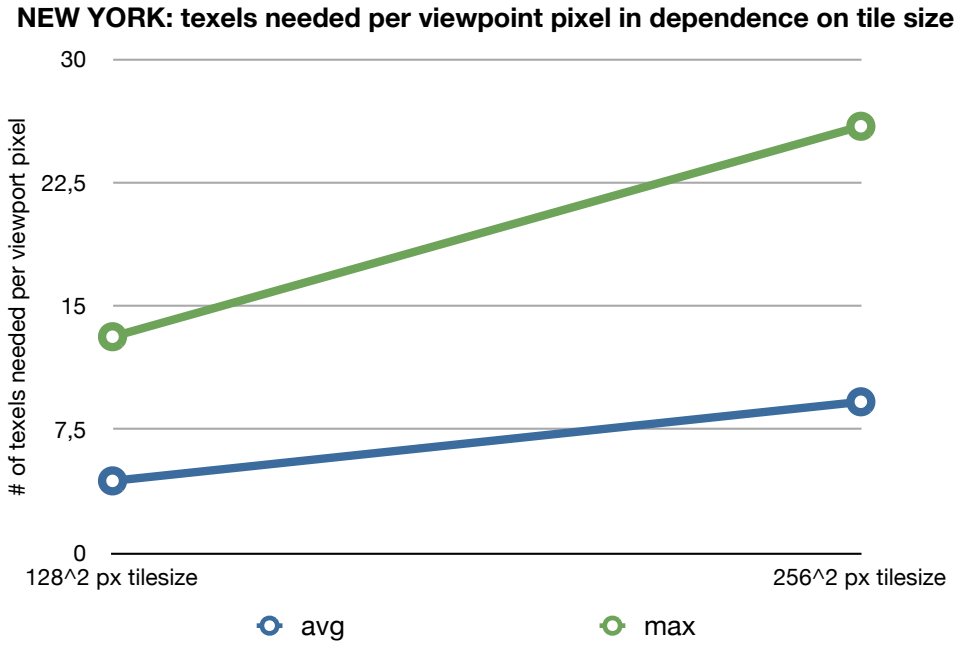

Fig. 4.2: Test showing the number of texels in the physical texture that are needed for every pixel in the viewport (walkthrough New York scene).

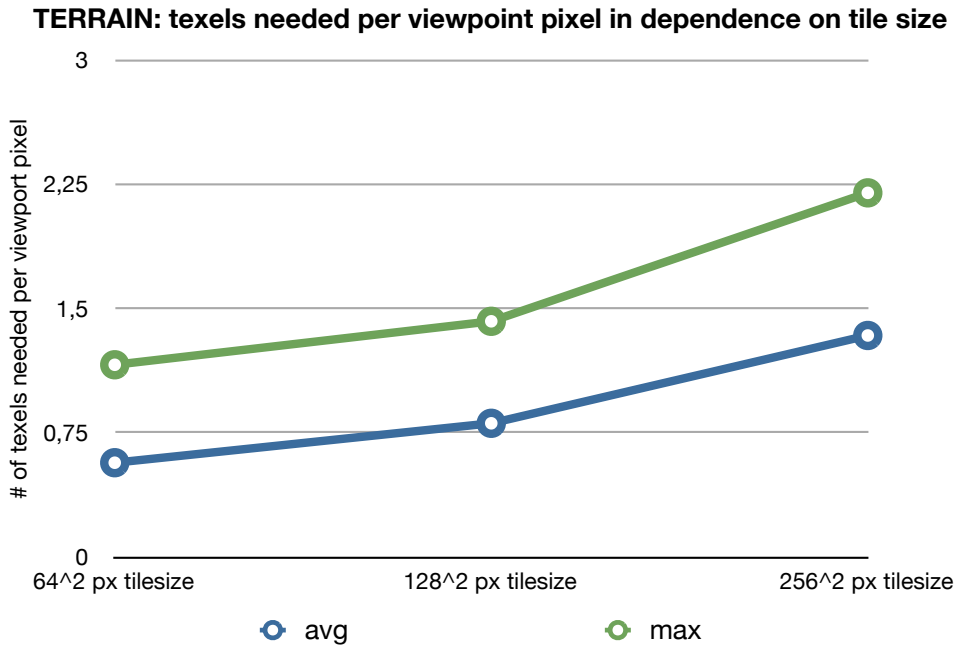

Fig. 4.3: Test showing the number of texels in the physical texture that are needed for every pixel in the viewport (walkthrough terrain scene).

4.1.2 Maximum Virtual Texture Size

As mentioned above, the virtual texture size is dependent on (or reversely: affects) the tile-size and the mipmap-chain length. Although there is no theoretical limit to the maximum virtual texture size, there are restrictions in the current graphics hardware that limit the useful virtual texture size:

- Mipmap-chain length:
  Having a mipmap-chain length of 11 results in a pagetable texture of size 1024^2 pixels and is the current upper limit in our implementation. A longer mipmap-chain results in substantial memory waste and considerable performance hit for pagetable updates. However, Barrett has had success with pagetables up to a size of 4096^2 [Bar08], which already approaches the texture size limitation.

- Tile-size:
  Increasing the tile-size above 256^2 pixel is not advisable since the tiles necessary for rendering any particular view do not fit into the largest possible physical texture on current-generation GPUs when using complex scenes (see Section §4.1.1). The basis for this claim is provided by tests with the New York scene, see Section §5.2. (Note: options to overcome the texture size limit with the physical texture are explored in Section §4.3.2.) The amount of necessary tiles is influenced not only by the tile-size but also by the viewport size, the virtual texture atlas layout and the geometry and how it is textured. The "best case" in this regard is a contiguous (terrain) mesh with a linearly mapped virtual texture. A 512^2 pixel tile-size is possible in this case, but in this scenario clip-mapping could also be used instead of virtual texturing anyway.

The combination of the maximum mipmap-chain length of 12 with the maximum tile-size of 256^2 pixels leads to the current limitation of the practical virtual texture size to $256k^2$ pixels. This is a practical and not a theoretical limit and subject to change with future CPU/GPU hardware. Mittring comes to a similar conclusion [MG08]. When combining this tile-size with a 4096^2 pagetable, the maximum virtual texture size is $1024k^2$. The currently common maximum texture size 8192^2 effectively limits the maximum virtual texture size (with 256^2 tiles) to $2048k^2$, but this can be worked around by virtualizing the pagetable texture, see Section §4.5.7.

An issue that prevents effortless scaling to higher resolution virtual textures is that (as mentioned) the maximum (x,y) coordinates for the tiles increase. This can be a problem when using the tile determination method of rendering in view space and reading back the tile coordinates (see Section §4.2.1), because the coordinates of the visible tiles have to be packed into the frame buffer. Up to the maximum of 256 by 256 tiles on the lowest mipmap-level (mipmap-chain length: 9) it is possible to store these coordinates

in a byte/channel each. Up to a mipmap-chain length of 11 one can fit the remaining bits into the mipmap-level channel. Above 11, a fourth byte/channel becomes necessary to store the coordinates. When leaving one bit free for distinguishing discarded pixels, one can fit information for a mipmap-chain length of 14 into a 4 byte render target (x 15 bit, y 15 bit, mip 4 bit, 1 bit distinction, 1 bit free).

According to Mittring, another issue that bounds the maximum virtual texture size is (floating point) precision within the GPU:

> "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. [...] integer maths [sic] avoid this all together."[MG08]

On an NVIDIA GeForce 8800 GT we have not seen obvious precision artifacts even with a $128k^2$ virtual texture. On an ATI Radeon HD 2600 PRO precision problems are visible even with much smaller virtual texture sizes.

Precision issues for the texture coordinates may also play a role for the texture coordinates when increasing the virtual texture size if some polygons occupy only a very small space of the texture space. At some point switching to double precision texture coordinates may become necessary, alternatively using multiple virtual textures is possible.

4.1.3 Texturing for a Virtual Texture aka 'Unique Texturing'

The geometry has to be textured just like the virtual texture would really exist in memory. If multiple objects are used, their textures are combined into the virtual texture exactly like with a texture atlas [Cor04]. The only difference here is that texture atlases usually do not exceed the size of the texture size limit on the GPU. Offsetting the texture coordinates is done to compensate for the packing of the textures.

Using a virtual texturing system provides a large enough texture space so that it is no longer required to simulate detail in a scene with a repeating base texture blended with a detail texture [MG08]. Instead, a virtual world can be "uniquely" textured with a single non-repeating texture. Figure 4.4 serves to illustrate the visual benefits resulting from a uniquely textured world. While unique texturing has been possible for the terrain for some time [Mal00], with virtual texturing it can be applied to arbitrary geometry. Benefits of unique texturing include that tiling patterns can be completely eliminated [Mal00] and if static lighting is used, the light-map can be baked right into the (surface color) virtual texture [Mal00], allowing for much higher effective light-map resolution

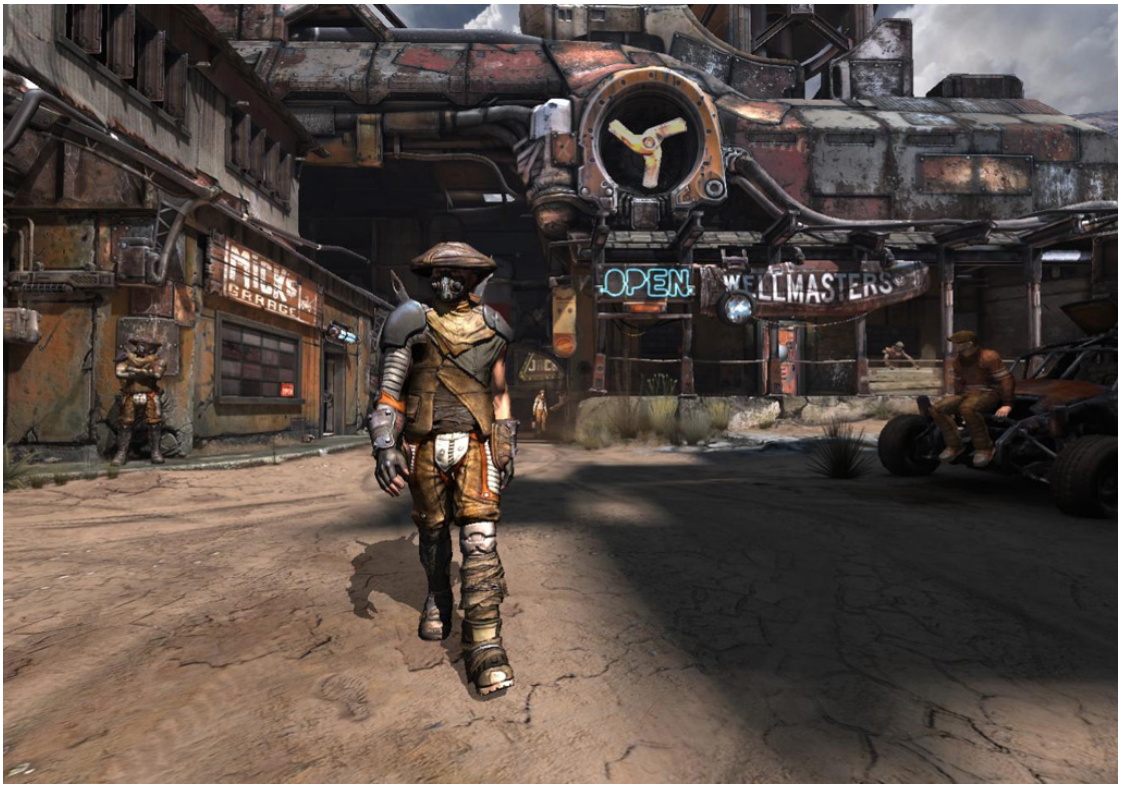

Fig. 4.4: The visual quality resulting from a uniquely textured virtual world is the biggest selling point of virtual texturing. Image copyright by id software.

than traditionally possible. This prevents stepping and banding, common artifacts with light map implementations.

Using virtual texturing requires geometry that is UV-unwrapped [MG08] (but alternatively combining virtual texturing with a texture parametrization that does not require unwrapping is also possible). According to Mittring, using an unwrapping algorithm that is aware of the tile borders at different mipmap-levels (quad-tree) can save memory on the hard disk and more importantly in the physical texture [MG08]. The empty space that occurs in all unwrapping methods should cover few tiles fully instead of many tiles partly. This is an important optimization to reduce the amount of needed tiles at runtime. Unique unwrapping is not generally needed by virtual texturing but made necessary when non-shareable additional data like light maps or bump maps are used [MG08].

The ability to create uniquely textured virtual scenes lifts a limitation of previous real-time rendering systems, but it also means that creating a uniquely textured scene

(a $128k^2$ texture contains more than 16 billion pixels) is more work than just using a repeating base texture blended with different details. It is suggested to move from a "manipulate individual pixels" approach to an approach where pre-created detail textures are stamped onto the scene [Car07].

When creating a scene and textures for a virtual texturing system using standard 3D-suites and image manipulation software, there are issues to be aware of. These tools currently do not use a virtual texturing system and therefore are not able to use or manipulate very large images. Intermediate images should not exceed $16k^2$, since $32k^2$ images are prohibitively slow and $64k^2$ images cannot even be imported or exported by common software. When viewing the scene in 3D-suites, only as many textures as fit into graphics memory should be enabled.

Adding up these problems results in a difficulty to create a large uniquely textured scene using currently common software. A possible solution is to write a custom tool that combines 3D editor, image manipulation functionality and virtual texturing. id Software has built such an editor for virtual textured scenes, allowing artists to texture a large scene in real time with a stamp system [Car07].

Unique textured scenes do not necessarily have to be created by artists from scratch. 3D scenes and the accompanying textures are also commonly created by 3D scanning (3D vision), aerial photography and satellite imagery. These methods already often result in so large amounts of data that it is difficult to display them in real time using established methods. Virtual texturing can be used very well to display these scenes, as demonstrated in Section §5.2.

Virtual texturing enables rendering with a very large virtual texture, but it makes still sense to consider how large a scene (or virtual world) textured with a common virtual texture size actually can be. For a very high quality output it is desirable to have a 1:1 mapping of texels to pixels [Cor07]. We assume a virtual world that consists only of relatively flat terrain. The camera is placed at two meters above the ground, which is reasonably similar to the height of standing humans (when the virtual character crouches the texturing resolution does not have to be perfect). We also assume the camera has a field of view of 90 degrees and is looking straight down to the ground. This means that the terrain portion rendered to the screens is about four "virtual" meters wide. Considering a medium-resolution display with a resolution of 1024 by 768 we want these four virtual meters to be textured with 1024 pixels to maintain the perfect 1:1 texel-to-pixel mapping. With this same texture density even a $128k^2$ virtual texture would only provide data for 512 by 512 meters of terrain. For a more realistic virtual world size of 8 kilometers by 8 kilometers, this virtual texture would only have 64 pixels per four meters. This results in the fact that even a virtual texturing system which allows using very large textures is not enough to texture the terrain of a large virtual world at high detail. Adding detail textures to the terrain may still be necessary, and in fact may

be easy to implement with virtual texturing when storing additional data for the detail texture id and blend factor.

4.1.4 Assembling the Virtual Texture

There are three steps to assembling the virtual texture:

- Merge to a texture atlas:
  As already mentioned, if the virtual texture covers multiple objects with distinct textures, they need to be merged into a single texture atlas. The usual rules governing texture atlas creation have to be followed, e.g., it is preferable to align sub-textures at power-of-two boundaries to prevent mipmap-chain pollution [Cor04] (see also Section §4.5.6 for a discussion of points to be aware of, and problems that can result from using a texture atlas). Since the texture atlas in this case is as large as the virtual texture, it does not fit completely into memory. Only as many sub-textures as fit into memory should be loaded at a single time. Also, as mentioned above, images above $16k^2$ should not be generated, so this stage either has to output multiple images or should be merged with the next step. Additionally, the texture atlas stage has several virtual texturing specific constraints:
  - Retain empty tiles that have been produced by the tiling-aware unwrapping algorithm (see above) by not placing sub-textures at arbitrary positions. This point is moot if this step is combined with the texture unwrapping, which could be preferred but is probably only possible in a very integrated pipeline.
  - Aim for similarity between texture space and world space to reduce the amount of needed tiles at runtime [MG08].
  - Do not position sub-textures with highly divergent texel-to-world density ratios within single tiles, see Section §5.2.1.
- Tiling and mipmapping:
  Mipmaps have to be generated and every mipmap-level has to be stored in equally sized tiles. It is again necessary to ensure that the memory is not oversubscribed, since the uncompressed virtual texture may exceed 192 GB in size [Duf06]. [MG08] discusses out-of-core mipmap generation as well as different downscaling kernels and their properties. If the virtual texture consists of multiple textures and therefore constitutes a texture atlas, care must be taken to avoid mipmap-chain pollution during downsampling [Cor04]. Additionally, borders for artifact free filtering are generated at this stage.
- Format and layout:
  The tiles have to be transcoded into a specific format and stored according to the

chosen layout, this is explained in the next subsection. Since this stage operates on the individual tiles, memory constraints are not a problem and the stage is easily parallelizable.

4.1.5 Storing the Virtual Texture

As mentioned above it is possible to stream the virtual texture tiles over the network or procedurally generate them at runtime, but the common setup is to store them on a local storage device. Two choices to make are the format and the layout of the tiles.

- Format:
  Since the virtual texture tiles are just normal pictures, using any image format is possible. There are basically three choices: uncompressed, compressed and GPU-compressed (compression formats that can be used directly on the GPU like DXTC).

  A fourth option would be to use DXT compressed textures but compress them further for on-disk storage, e.g., by compressing the DXT color blocks using JPEG. This option is explored and dismissed as unsatisfactory in [vW06b]. Implications of the format choice and benchmarks are discussed in Section §4.3.1.

  Computer games or visually advanced simulations typically do not use only a surface color texture map, but also additional data like the bump map and specular map. Using this data is possible with virtual texturing, in the simplest case the same UV-mapping is used for the additional data. That means for every surface color RGB pixel in the virtual texture, we also have a bump/specular pixel specifying the additional properties at the very same object-space position [MG08]. Retaining the same UV-mapping but using differently sized tiles for different data types is also possible [MG08]. A more complex and flexible system for multitexturing is proposed in [Bar08] and described in Section §4.5.15. When choosing a compressed format for the tile storage, it has to be determined if the compression format is also acceptable for the additional data, but this is not different from a system without virtual texturing. Differently compressed "layers" are obviously possible. Three conflicting requirements for compressed tile formats are:

  - Decompression speed: since the tiles have to be decompressed during streaming, they should be very fast to decompress. Uncompressed or DXT compressed tiles do not have to be decompressed but are too large to be streamed efficiently.
  - Tile-size: the compression rate of the tiles should be high to lessen the bandwidth requirements during streaming. This is even an issue with storage on the hard disk and even more so with optical media or network streaming

  - Visual quality: the compression should result in low visual degradation. If the tiles are stored DXT compressed in the physical texture, the best option here is to use DXT pre-compressed tiles, because an offline DXT compressor achieves higher quality. Additionally, this prevents "adding" the compression artifacts of two different lossy formats.

- Layout:
  The simplest layout for the virtual texture tiles is to store each tile into a single file. Storing the tiles in a lossy compression format means that the size of a single (128^2 pixel) tile is relatively small, typically ranging from 5 to 25 kilobytes. Tiles with good compression attributes will even fall below the common cluster or block size of the hard disk, resulting in wasted space. Due to the way rotational/mechanical storage devices work, reading such small files is not significantly faster than reading files that are a few times larger, latency is the defining factor here. Figure 4.5 shows the relation between file size and the average time needed to read the file. Reading a 256KB file takes only about twice as long as reading a 4KB file, although it is 64 times larger.

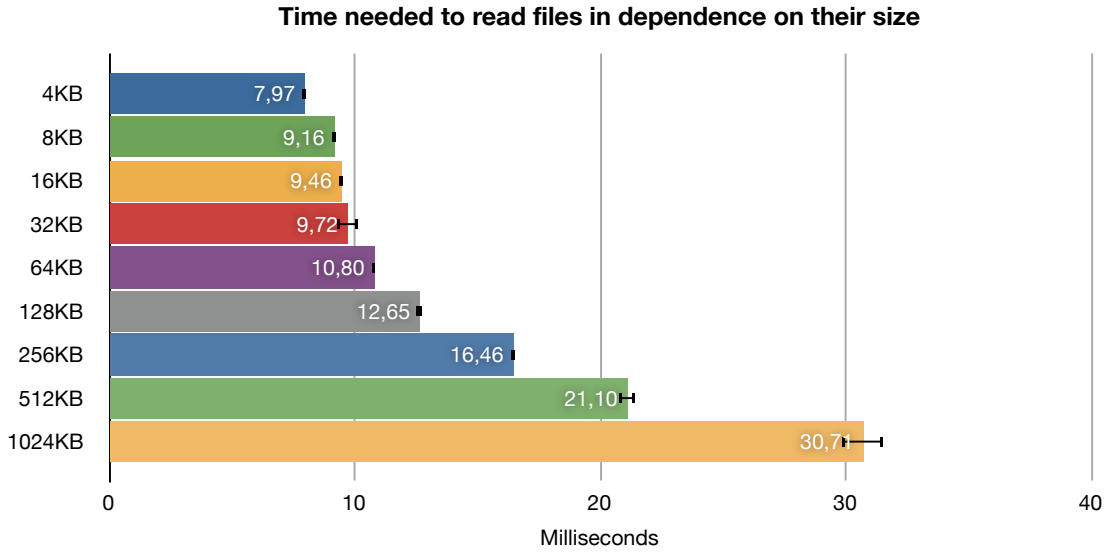

Fig. 4.5: The performance of file reading in dependence on file size.

Considering these issues it is preferable to pack the virtual texture tiles into fewer files or even a single file. [CESL10] come to a similar conclusion. The tiles can be grouped inside the file(s) so that multiple tiles that are likely to be visible at the same time can be loaded at the same time. Heuristics for this could include packing tiles that are adjacent in word space, alternatively data about simultane-

ous loads could be gathered from runtime experiments. Considering Figure 4.5, the optimal size of data to read in at once probably is between 64 and 256 KB. Also, fragmentation of the tiles is expected to be less severe when storing them as a single file. [MG08] offers some insights on efficient streaming from a single file on disk. The test in Figure 4.5 disables the effects of the page cache with `/usr/bin/purge` and the effects of the disk buffer by previously accessing unrelated files that amount to a few times the size of the disk buffer. Actual file reading happens with `fopen() / fread()`.

These issues (and the general issue of latency when streaming tiles) will be much less serious on the upcoming solid-state drives.

However, with optical media the issue is more severe (higher latency, lower throughput [MG08]), but the layout can also be controlled better on read-only media. [vW09a] provides a comprehensive discussion of streaming from slow storage devices for a clipmap-like texture streaming system.

4.1.6 Data Reduction

In storage-constrained situations, the virtual texture may require too much space even after applying lossy compression to the tiles. Moreover the virtual texture can contain high-resolution textures that are never visible anyway in the simulation due to movement constraints. For many types of simulations it is possible to determine which parts of the scene are textured with higher density than will be used at runtime. Analytical or brute-force solutions are thinkable, and in video game settings, test player runs could be analyzed. After determining that parts of the scene are textured with unnecessarily high detail, there are two options: Reducing the size of the sub-textures for these parts, leading to different texture coordinates after coordinate adjustment. The second option is to leave the texture coordinates as-is and delete the relevant tiles from the lowest mipmap-level. This option is preferable (only) if the scene is (to be) textured with equal texture density, e.g., to ease tile determination. The second option also allows reducing the resolution only for parts (tiles) of sub-textures, instead of whole sub-textures, providing a crude form of adaptive resolution.

Additionally, there likely are empty tiles because any UV-unwrapping always leaves some areas unused, these can also be deleted [MG08] since they will not be requested anyway.

4.2 Tile Determination

The tile determination system has to find out which of the tiles of the virtual texture are currently visible, so they can be transferred to the GPU before rendering. Tile determination is not a new technique that is unique to virtual texturing, but an established component of all (tile-based) texture caching methods. However, previous attempts cannot be used for virtual texturing: approaches like [CE98] rely on doing geometric computations per polygon, which is not feasible with current polygon counts ranging in the millions. Other approaches like clipmaps define a center of interest in texture space and are therefore limited to terrains [TMJ98].

While cached tiles can be transferred to the GPU in the same frame, tiles that are not cached take multiple frames to be loaded from the hard disk, decompressed and re-compressed. It is desirable to know in advance which tiles will be visible in a few frames, so these tiles can be streamed just in time, preventing artifacts. This is only possible in controlled walkthroughs where the virtual camera path is predetermined, but not in an interactive simulation where the user controls the camera position and orientation.

Tile determination systems can be categorized into exact, conservative, aggressive and approximative systems, in analogy to visibility. Exact systems deliver the exact set of tiles currently visible, while conservative systems also include tiles that are currently not visible (but might become visible soon). Aggressive systems, which do not find out all visible tiles, are not particularly useful because they result in artifacts. Approximative systems may be useful with changes to the virtual texturing shader. All exact tile determination systems and other systems which take occlusion into account can be seen as a superset of the functionality of an occlusion culling algorithm and therefore could be classified as a "hard problem".

Our tests have shown that complex scenes (with suboptimal virtual texture layout) need so many tiles that they barely fit into the limit for the physical texture (see Section §5.2). Therefore an exact system is necessary and a conservative system is not acceptable for some scenes. However, since conservative systems commonly include tiles that might become visible soon, they can be useful in addition to an exact system. While only requests from the exact system are streamed to the physical texture, during idle times requests from the conservative system are served, to have these tiles already stored decompressed in the cache. Also, other setups with simpler scenes, better virtual texture layout, smaller viewport or larger physical texture may very well get along with just a conservative tile determination system. Mittring also prefers a conservative system [MG08]. Besides correctness, performance and the ability to "look into the future", a good tile determination system should also be able to sort tile requests by priority – tiles that cover more screen pixels should be loaded first.

An important point between the tile determination and the tile streaming system is the handling of tiles that were requested in previous frames but are not visible anymore. These requests should be discarded to prevent a worst case scenario: if tiles are requested at a faster rate than they can be streamed (for example when the camera moves very fast), this would result in the tile streaming system falling behind the tile determination more and more, only loading tiles that have been visible some seconds ago.

4.2.1 Exact Tile Determination in View Space

Barrett devised and implemented a method where the scene is rendered with a special fragment shader that outputs the tile coordinates at each pixel into the frame-buffer [Bar08]. The buffer is read back to the CPU. The benefit of the method is that it is completely exact, takes occlusion into account and is simple to implement. Additionally it uses the GPU to solve the relevant issues of visibility and occlusion. Drawbacks include necessitating another render pass, the cost and latency of reading back the buffer to the CPU, the fact that the buffer is a large list of the necessary tile at each pixel and not a small list of unique tiles that are necessary.

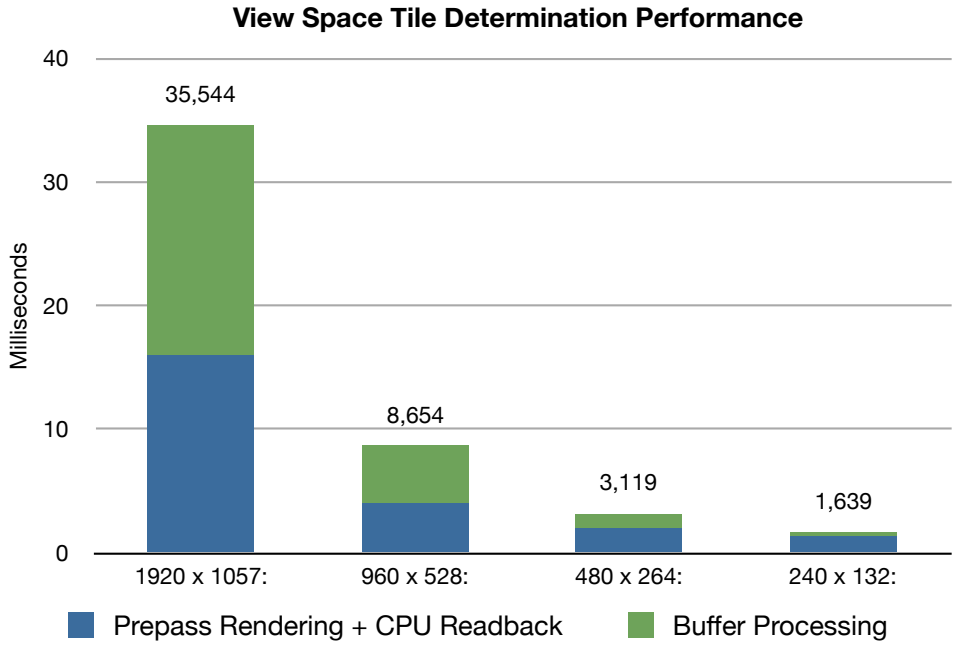

Fig. 4.6: The performance of the view-space tile determination in full, half, quarter and 1 / 8 resolution.

Performance can be increased by doing this render pass at a lower resolution than the

viewport at the cost of exactness. Reducing the width/height by powers of two is particularly easy: if the width/height is shifted right by "x" then a subtraction of "x" from the determined mipmap-level in the shader compensates for the smaller resolution. Reducing resolution not only increases the speed of rendering this pass, more importantly it also reduces the time for transferring the buffer to the CPU and post-processing it there. Less detailed geometry can also be used for rendering this pass. When halving the resolution of the tile determination pass, visible tiles that only occupy single pixels may be lost. Conversely, the resulting artifacts are limited to single pixels too. It is common that multiple adjacent pixels necessitate the same tile, therefore artifacts caused by reducing the resolutions are infrequent. Note that a smaller tile-size increases the artifacts because more tiles will be missed. We experienced good performance and few artifacts when rendering at quarter resolution (480 x 300) with 256^2 pixel tiles. Figure 4.6 provides a performance overview of the method at different resolutions. Note that the read-back can occur asynchronously using a PBO, which frees the CPU to do other work during the transfer but does not usually reduce the total transfer time. Supposedly the cost of the read-back is much lower on gaming consoles [Spa10]. Additionally, more efficient implementations may lower the costs of iterating even over large buffers.

Figure 4.7 illustrates the percentage of the visible tiles that are determined at lower resolutions. The terrain scene is the best case scenario for this test because the virtual texture is mapped linearly over the terrain-mesh, so each tile occupies a large contiguous area in screen space and is therefore "hard to miss" even at lower resolutions. The New York scene represents a mediocre case since each of the thousands of buildings occupies only a few pixels on the screen but pulls in different tiles that are missed at lower resolutions – a fact that is mainly caused by problems with the texture atlas layout of this particular scene (see Section §5.2.1) and not by the geometry. The problem mainly occurs in flyover scenarios when viewing most of the scene, but not from a "normal" viewpoint near the ground. Scenes that do have a better virtual texture layout should see significantly better results than the New York scene.

Program 1 shows an example GLSL fragment shader for performing the tile determination pass. This example calculates the mipmap-level of the visible tiles using `dFdx()` / `dFdy()`. Unfortunately the OpenGL standard leaves the actual mipmap selection function up to each implementation (see the OpenGL specification, Section 3.8.11 [Gro09]), so it is impossible to match all implementations with a single calculation function. The shader can be modified to determine the mipmap-level (and even the coordinates) of the visible tiles by sampling from a texture instead of calculation, to get exactly the result the host OpenGL implementation provides. To do this one writes distinct values to the different mipmap-levels of a texture, the result of the fetch then indicates from which mipmap-level the fetch has been performed. This wastes memory, but although a texture fetch now incurs, the performance is the same as with the calculation. When using the texture fetch just to determine the mipmap-level (and not

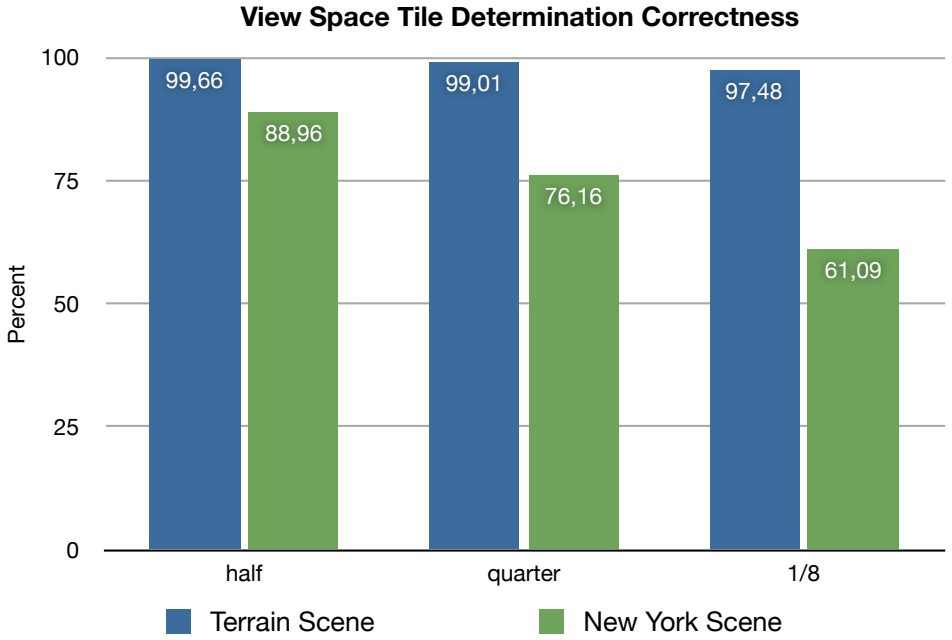

Fig. 4.7: The correctness of the view-space tile determination in half, quarter and 1 / 8 resolution.

the coordinates), the required data can be stuffed into the pagetable texture to prevent an increase of texture unit and texture memory requirements. More concretely, depending on the implementation, the pagetable texture may still have a free channel left, e.g., the alpha-channel. This channel can be filled with different values, depending on the mipmap-level, so that a fetch from this texture yields the selected mipmap-level of the tile. However, if it is desired that the x/y coordinates of the tile are also determined from the fetch to save shader calculation instructions, a dedicated texture for this purpose is necessary. A third option is the new OpenGL extension ARB_texture_query_lod, which provides exactly the functionality needed – determine the mipmap-level that would be used if a texture were sampled – but is not commonly available just yet. The example shader stores the (x,y) coordinates of visible tiles in one channel each (red and green) of the frame-buffer/FBO and therefore restricts the maximum coordinates to 255 (mipmap-chain length 9).

Barrett later proposed refining the method to allow reducing the resolution completely without artifacts by means of temporal subsampling:

> "You can reduce your read-back bandwidth by 75% by only drawing one quarter of the screen each frame, and cycling through those quarters. Naively, you would divide the screen into physical quarters: top left, top right, etc. More sensibly, you'd chunk the naive pixels into 2x2 clusters and only render a particular subpixel of all clusters each frame. This amounts

Program 1 A GLSL fragment shader for exact tile determination in view space with a render pass. Based upon Barrett's SVT demo shader [Bar08].

```
const float readback_reduction_shift = 2.0;
const float vt_dimension_pages = 128.0;
const float vt_dimension = 32768.0;

uniform float mip_bias;

// analytically calculates the mipmap level similar to what OpenGL does
float mipmapLevel(vec2 uv, float textureSize)
{
    vec2 dx = dFdx(uv * textureSize);
    vec2 dy = dFdy(uv * textureSize);
    float d = max(dot(dx, dx), dot(dy, dy));

    return 0.5 * log2(d)   // explanation: 0.5*log(x) = log(sqrt(x))
          + mip_bias - readback_reduction_shift;
}

void main()
{
    // the tile x/y coordinates depend directly on the virtual texture coords
    gl_FragColor.rg = floor(gl_TexCoord[0].xy * vt_dimension_pages) / 255.0;
    gl_FragColor.b = mipmapLevel(gl_TexCoord[0].xy, vt_dimension) / 255.0;
    gl_FragColor.a = 1.0; // BGRA: mip, x, y, 255
}
```

> to rendering a 1/4 sized screen but displacing it by a subpixel amount each frame to try to hit any thin stuff. Which means if you're already doing subresolution, you can view it as a way to reduce resolution further, or to keep the current resolution but improve quality. You'd want any tile IDs you hit to stay valid for at least 4 frames, which might slightly decrease overall quality." [Spa10].

A simpler method that tries to do the same thing in a less sophisticated way is randomly jittering the viewport in each frame by a small amount.

In OpenGL, reading back rendered content can be done either by rendering to the back-buffer or to a FBO. The actual read-back can be done using `glGetTexImage()` or with `glReadPixels()`. We have implemented and tested all four possible combinations. Micro-benchmarks of the read-back operation proved to be indecisive, therefore we tested whole-application performance, the results can be seen in Figure 4.8.

The tile determination render pass can be combined with any other render pass (e.g.: z-only, deferred pre-pass, main render pass) by writing out the tile determination information into another render target (MRT). The drawback here is that this forces the tile determination pass to be done at full resolution, and the full-resolution buffer is prohibitively slow to transfer back and post-process (see Figure 4.6). Regardless of the combination with another render pass, the buffer can be converted using GPGPU into a smaller buffer, for faster transfer to and usage on the CPU.

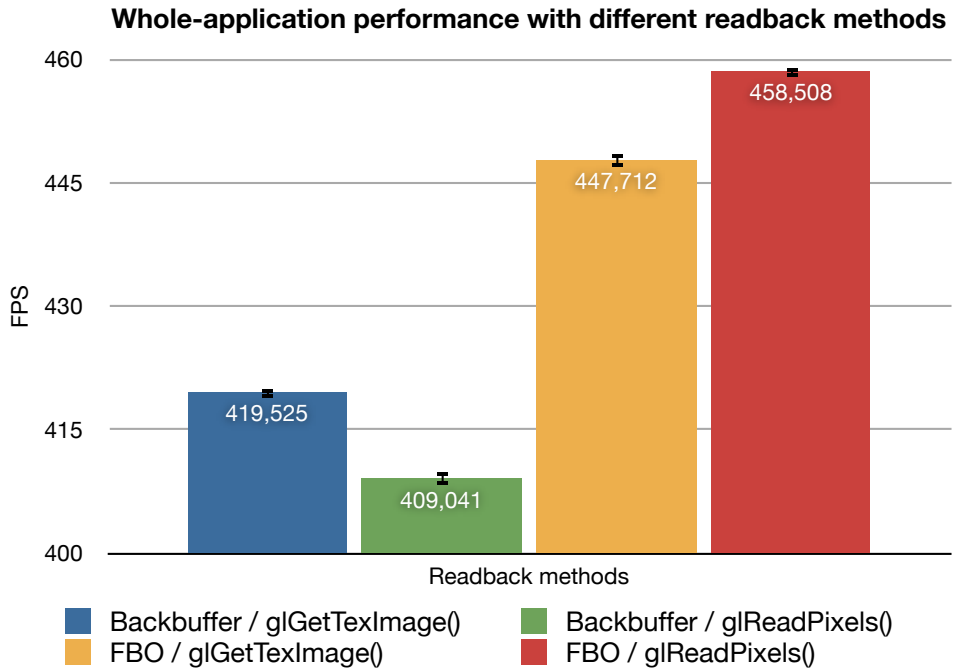

Fig. 4.8: Whole-application performance with different read-back methods during a terrain fly-over.

Expansion of Barrett's method to include tiles that are not currently visible is possible, effectively transforming it into a conservative method. The Field of View (FoV) can be increased during rendering so that tiles outside the real view frustum are also marked as needed, which gives improvements when the virtual camera is rotated. The calculated mipmap-level of the tiles has to be corrected to adjust for the FoV-change, and the method can no longer be combined with other passes. This idea is also noted by [CESL10], they get away without adjusting the mipmap-level because they only increase the FoV slightly. Other ideas include performing additional tile determination passes in other viewing directions (to help with rotation) or from an extrapolated camera position/orientation. [Neu10] had the same idea and performed tests with a "LookAhead Camera", with mixed results.

Regardless of the used tile determination method it is also possible to try to derive tiles that might become visible soon from the tiles that are currently visible. Lower or higher mipmap-levels of currently visible tiles can become visible in the future. For the parts of the virtual texture that have a high affinity of texture space to world space (e.g., parts that cover linearly mapped terrain) one can also load adjacent tiles.

4.2.2 GPGPU Buffer Compression for View-Space Tile Determination

The resulting buffer of a view-space tile determination pre-pass contains the address of the requested tile for each pixel. Since neighboring pixels are very likely to request the same tile, most adjacent positions in the buffer contain the same value. What we are actually interested in, is not the repetition of these same values over and over again, but a list of unique values, i.e, the list of visible tiles. The number of visible tiles usually only ranges in the hundreds. Therefore we want to compress the buffer to this list of unique visible tiles.

Converting the buffer to a more compact representation serves two purposes. Reading back a compact representation of the necessary information significantly reduces the time to transfer the data to the CPU and processing it there. Furthermore this allows the tile determination pass to be done at full resolution, which means it can be combined with other render passes, eliminating the requirement of an additional pass.

Having to compress a very large buffer that only sparsely contains useful information is not a new problem unique to virtual texturing, but is commonly known under the terms stream compaction or stream reduction [BOA09]. Prefix sum-based algorithms [SLO06] are an established and common solution that also operate on GPUs. [BOA09] proposes a new solution developed with GPU/SIMD architectures in mind. Recent advanced shadow-mapping solutions also face a very similar problem to ours, they also generate tile requests on the GPU and have to compress this request buffer for efficient read-back performance [LSO07]. [LSO07] compress their request buffer with the following stages:

- Removing some redundant requests between neighboring pixels by marking requests which differ from their neighbors.
- Elimination of invalid requests.
- Removing all remaining redundant requests by sorting the requests, marking unique elements and finally removing the non-unique requests.

This method could be used for virtual texturing as is, but the necessity of sorting a very large amount of data suggests there might be performance problems – additionally the process is comparably complicated.

[HPLdW10] have presented a much simpler method to convert the buffer to a compact list of needed tiles in CUDA. We have implemented the same method using OpenCL. While the first stage (kernel) was developed independently from [HPLdW10], the second stage is directly based on their sample code.

Program 2 First OpenCL kernel for buffer reduction - converts to a quadtree.

```
kernel void buffer_to_quadtree(image2d_t img, global uchar *pagetable,
                               constant int *offsets, constant int *widths,
                               constant int frame)
{
    int image_width = get_image_width(img);
    int image_height = get_image_height(img);
    int x = get_global_id(0);
    int y = get_global_id(1);


    // only process if if are inside the real buffer dimensions
    if ((x < image_width) && (y < image_height))
    {
       // read the values from the tile determination pass
        float4 clr = read_imagef(img, CLK_NORMALIZED_COORDS_FALSE |
                                 CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST,
                                 (float2)(x, y));
        uchar indx_x, indx_y, indx_mip;
        indx_x = convert_uchar_sat(clr.x * 255.0f);    // extract x coordinate
        indx_y = convert_uchar_sat(clr.y * 255.0f);    // extract y coordinate
        indx_mip = convert_uchar_sat(clr.z * 255.0f);  // extract mipmap-level

        // the offsets into the pagetable buffer are passed to the kernel
        int index = offsets[indx_mip] + indx_y * widths[indx_mip] + indx_x;
        pagetable[index] = frame; // mark tile as used in the pagetable buffer
    }
}
```

Program 2 shows the first kernel, it processes the buffer as produced by program 1 and marks the visible tiles in a pagetable structure that contains an element per existing tile (this quadtree/pagetable is conceptually identical to the pagetable texture). Program 3, the second kernel, processes this quadtree and inserts the marked visible tiles into a fixed-length list. This list is then transferred to the CPU. Basically the first kernel "uniquifies" the tile requests and the second kernel compacts the requests by inserting only the visible tiles into a list. This two-stage method circumvents the need for any sorting of the requests for redundancy elimination. Each kernel has a bottleneck. The first kernel is throttled by the common access from different work-items to the same global memory locations in the pagetable. Although the pagetable needs only one bit per tile, we use one byte per tile because the bit-packing would intensify this bottleneck and severely slow down the kernel. A side effect of this memory-waste is that it allows us to tag tiles by frame-id, reducing the need for clearing the pagetable from every frame to every $1/255$th frame. Note that OpenCL, in distinction to CUDA, has no `memset()` built in, therefore clearing memory must be done either with a kernel or by overwriting the memory with a buffer transferred from the CPU. We use a kernel to clear the quadtree every $1/255$th frame, but use a transfer to clear the list index every frame.

The bottleneck of the second kernel is also the common access to shared memory, in this case the result list. However, since there usually are only a few hundred visible tiles,

Program 3 Second OpenCL kernel for buffer reduction – converts to a list. Based upon the CUDA kernel with the same purpose in [HPLdW10].

```
kernel void quadtree_to_list(global uchar *pagetable, constant int numItems,
                             global uint *tileList, constant int frame)
{
    int x = get_global_id(0);

    if (x < numItems)
    {
        // if the tile is used this frame, i.e. marked with the frame number
        if (pagetable[x] == frame)
        {
            int i = atom_inc(tileList);  // increase the list size (element 0)
            tileList[i+1] = x;           // and append the item to the list
        }
    }
}
```

this is manageable. The second kernel only works correctly if there are less tiles visible than fit into the fixed-length list. With dynamic LoD adjustment (see Section §4.3.2) and an appropriately sized list this can be easily ensured, alternatively an additional check is necessary. The original kernel presented in [HPLdW10] does not have this issue because the CUDA atomic increment function allows to set an upper bound. Compared to the original kernel we have also dropped the calculation of tile coordinates to write to the list and just write the indices into the quadtree. This allows us to simplify the kernel (reducing register usage), and these tile coordinates can be easily calculated on the CPU from the quadtree indices. A specific problem that was encountered during the implementation is that it is not possible to start using OpenGL before OpenCL has completely finished (which is ensured by `clFinish()`). This includes the read-back of the final list. Although the list is small and the read-back takes minimal time, it would be preferable to have this transfer run asynchronously while other (OpenGL) commands are issued. A possible workaround would be to copy this list to a buffer with OpenCL (a fast GPU to GPU copy) and then start the transfer through OpenGL via PBOs.

Figure 4.9 provides a performance overview of view-space tile determination with the presented OpenCL buffer reduction. When comparing the results with the results without OpenCL from Figure 4.6 we see a performance increase by a factor of 3 to 4. The CPU is also free to do other work during kernel execution, unless `clFinish()` is called "too early". The performance difference to [HPLdW10], which claim a buffer reduction time of 1.2 milliseconds, can be explained by the different GPU hardware, the GeForce 8800 GT we used is the earliest generation hardware supporting GPGPU. Additionally, OpenCL is still relatively new and not as mature as CUDA, but since it is a vendor independent technology we see it as more future-proof. It should be noted that it is trivial to perform the buffer reduction on a lower resolution than is actually rendered. E.g., it is possible to combine the tile determination render pass with the main

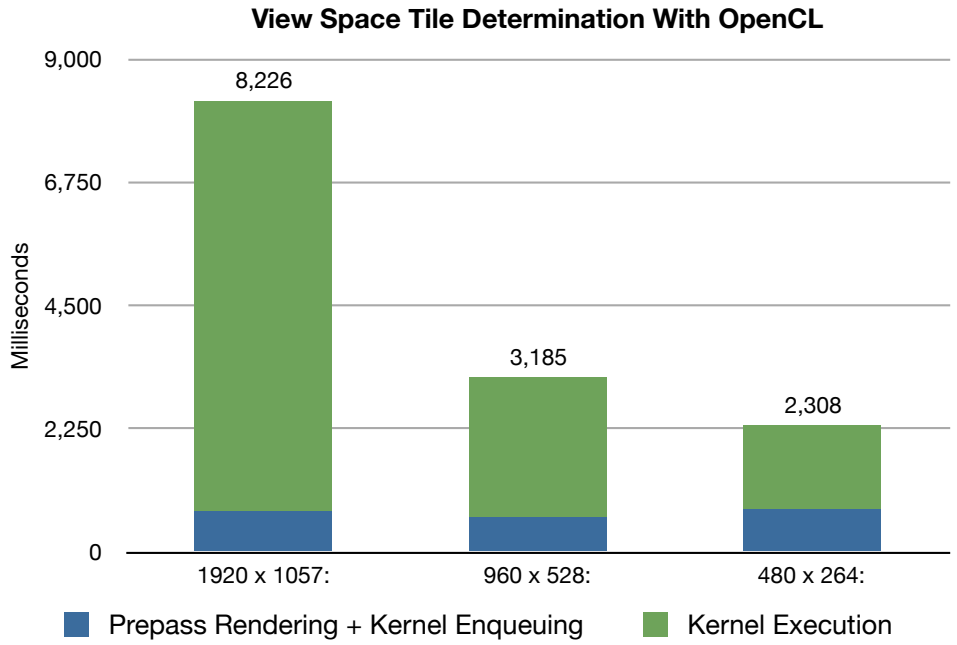

Fig. 4.9: The performance of the view-space tile determination in full, half, quarter and 1 / 8 resolution when doing OpenCL buffer reduction.

render pass, write out the tile requests using MRT and still do the buffer analyzing using OpenCL only on every 4th element. In contrast to [HPLdW10] we did see a significant speed difference between processing a full-resolution and a half-resolution buffer.

A downside of using this OpenCL buffer reduction is the loss of some information: it is no longer known how many pixels request a given tile. This information can be used to sort tile requests by importance for improved visual quality (seen Section §4.5.9). This information can be retained during the buffer reduction, one option is to store the quadtree in `ushort` instead of `uchar` format, doubling the memory requirement (since tiles can be requested by more than 256 pixels). The size of the final list also doubles since it alternately contains tile numbers and their pixel usage.

The new OpenGL extension EXT_shader_image_load_store finally introduces scattered memory writes to GLSL shaders and can be of considerable help for tile determination. One possibility to explore is writing out the tile requests from the shader directly in the quadtree format. This would save time and memory because the first kernel becomes redundant.

4.2.3 Other Tile Determination Methods

Various other tile determination methods are possible:

- Pre-calculated conservative tile determination – Potentially Visible Set for virtual texture tiles:
  For static environments it is possible to pre-calculate the tiles that can potentially be visible from designated areas (cells) in the virtual world, similar to the Potentially Visible Set (PVS) system for occlusion culling [Air90]. The tradeoffs are similar to the existing PVS for occlusion culling: no runtime overhead, limitation to static geometry and overestimation of the exact set.

- Conservative tile determination in texture space:
  Tile determination by rendering in texture space was first proposed in [LDN04]. This method does not take occlusion into account, a deficiency which could also be used to advantage by interpreting it as a tile prediction method. This provides a definite benefit when large parts of the scene suddenly become disoccluded. The method can be performed either on the GPU or the CPU. Refer to [LDN04] and [MG08] for an elaboration of the method.

- Approximative tile determination in world space:
  Mittring mentions distance-based approximative tile determination systems – the required tiles can be calculated per triangle or draw call [MG08]. Content with a varying texture density requires more complicated treatment [MG08]. Approximative tile determination is easiest to do with height-map-based terrain: the mipmap-level borders are at concentric circles or squares around the center of interest. This is tile determination as done in clipmaps systems [TMJ98].

- Special cases:
  There are some special cases where the tile determination becomes easier. For example building an image viewer based on virtual texturing technology results in an orthographic projection with a constant perpendicular viewing angle and therefore makes tile determination trivial.

4.2.4 Adapting Rendering to Tile Determination

As mentioned above, calculation of mipmap-levels in the shader is not able to exactly match the mipmap-level-selection that OpenGL does. This results in the fact that even “exact” tile determination in view space is not really “exact” but only requests approximately the right tiles. (Unless mipmap-level calculation by texture fetching is used, as explained above.) Approximative algorithms do not even try to fetch exactly the “right”

tiles. Since the exact mipmap-level calculation is not clearly defined in the OpenGL specification (see above), algorithms that do not use the OpenGL mipmap-level calculation (via texture sampling or ARB_texture_query_lod) will never exactly match the tiles OpenGL would use.

Because sampling from the pagetable texture uses the OpenGL mipmap-level calculation, there is a discrepancy to the tile determination system, which in the worst case can result in a fallback to a low-resolution tile for rendering. Consider the following example: the tile determination system requests a tile from mipmap-level 0 for a group of pixels. However, OpenGL selects mipmap-level 1 for these pixels, which is not available, therefore the fallback entry from mipmap-level 4 is used, resulting in a completely blurred result at these pixels.

The solution is to either adapt the fallback mechanism or the sampling from the pagetable to the tile determination system. For example when using view-space tile determination with mipmap calculation, one should adapt the virtual texturing shader to sample into the pagetable with `texture2DLod()`, with the mipmap-level calculated in the same way as during the tile determination. This prevents the mismatch with the OpenGL mipmap-level calculation. When using an approximative tile determination, one could adapt the fallback mechanism so that the approximated tile is used.

A related question is whether the fallback should fall back to higher resolution tiles too, i.e., should we consider also higher resolution tiles or only lower-resolution tiles for rendering when the native tile is not currently available.

4.3 Tile Streaming System & Texture Updates

The tile streaming system has to load visible tiles from the storage medium. If the loaded tiles are stored compressed, they have to be decompressed. They also have to be re-compressed, if the tiles should be stored compressed on the GPU. It is also possible to store the tiles already compressed in a GPU compression format, so only loading and no transcoding has to be done. The tile streaming happens asynchronously in additional threads, while the texture updates are usually performed on the render thread, just before the main render pass with the virtual texturing shader. For the texture updates, “new tiles” are considered, i.e., tiles that have been loaded by the tile streaming since the last frame. These will be needed during rendering but are not currently available in the physical texture. These new tiles are streamed to the physical texture on the GPU. Additionally, the pagetable texture has to be updated. Figure 4.10 provides an overview of the tile streaming system.

The following subsections describe the tile streaming and the texture updates.

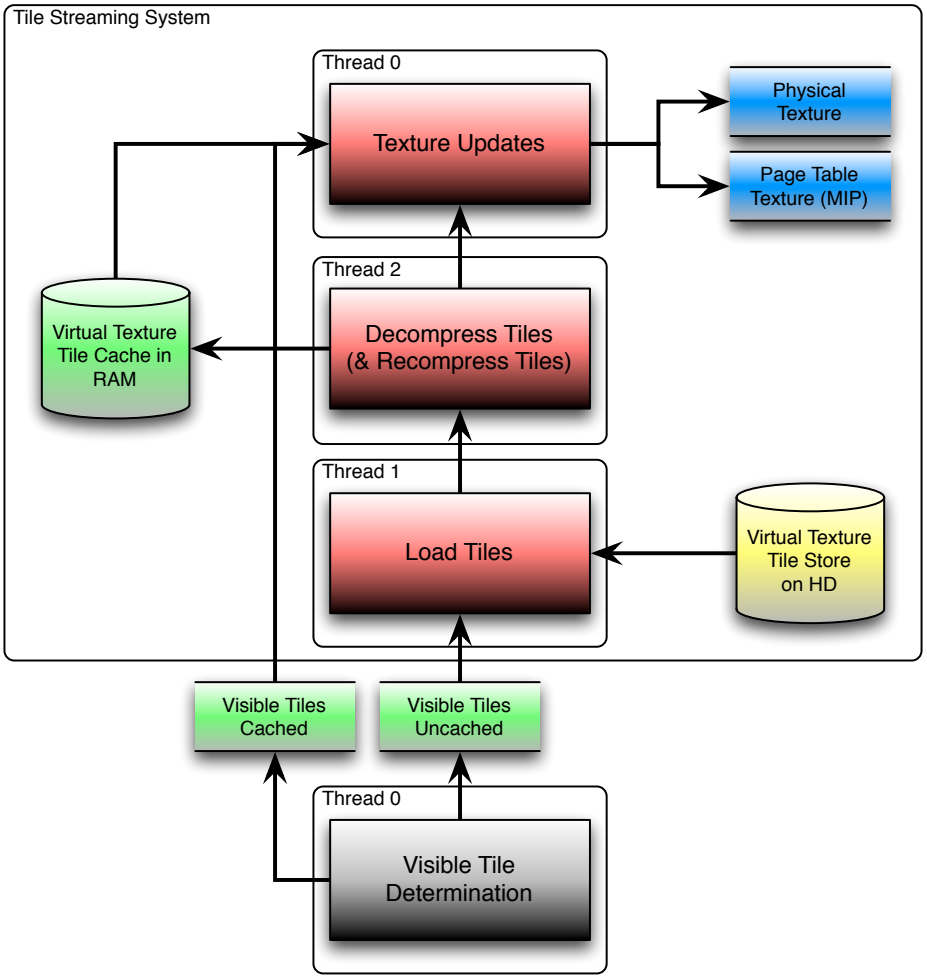

Fig. 4.10: An example tile streaming system.

4.3.1 Tile Streaming

The tile streaming system loads virtual texture tiles from the storage medium, decompresses them if they are compressed, and re-compresses them if they should be stored compressed on the GPU. Since the tile determination system cannot look into the future (and often requests too many tiles if it tries to) the time needed to stream tiles from the storage medium to the GPU is a determining factor for the visual quality of virtual texturing. The performance of the tile streaming system can be measured by the latency (time from request to arrival) and the related factor of throughput. The latency is a factor that affects visual quality continuously, the throughput is more of a "binary" factor, either it is high enough to cope with the demand, or it is too low. Figure

4.11 provides an overview of throughput of common JPEG and PNG decompression libraries. Note that this and other benchmarks concerning tile streaming are performed on 1024 tiles of size 256^2 pixels. The JPEG tile store is compressed 1:10.8, the PNG tile store is compressed 1:2.1 (the PNG tiles are over 5 times larger than JPEG). The results suggest that the PNG format is not particularly suited to high performance decompression – the fastest library (libpng) reaches 17,6 megapixels per second. JPEG decompression throughput is much higher, with the fastest contender (libjpeg-turbo [liba]) decompressing at nearly 100 megapixels per second.

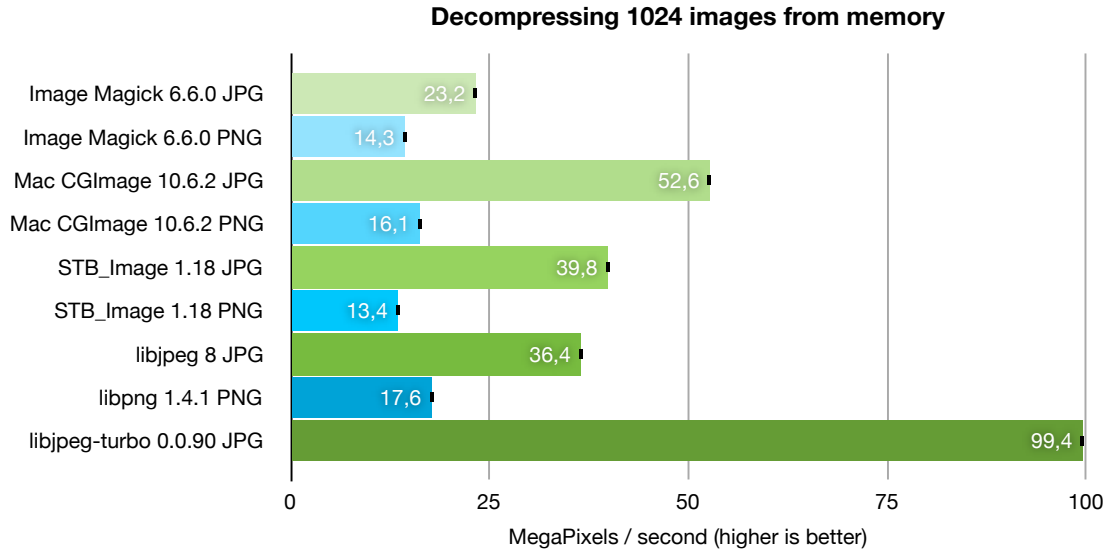

Fig. 4.11: Throughput of common JPEG and PNG decompression libraries.

However, the tiles have to be streamed from the storage device and not from memory. Figure 4.11 shows the performance when loading and decompressing the tiles. Some results are labeled as "decoupled", which means the decompression is done in a separate thread. The de-coupling helps immensely with the throughput. Throughput for libjpeg [Gro] and libpng [libb] is much lower for decompressing from hard disk than for decompression from memory, unless the loading and decompression are decoupled. In the decoupled case libjpeg and libpng nearly achieve the same throughput as when decompressing from memory. De-coupling also helps much for libjpeg-turbo, but only 2/3 of the speed of decompression from memory is achievable, because libjpeg-turbo is so fast that the disk loading becomes the bottleneck here. In this case a higher compression rate could increase the throughput because disk loading times are decreased. We expect the throughput to be the same as when decompressing from memory with a compression ratio above 1:20. While decoupling increases the throughput, the latency is not decreased. One result here is this: Decompressing tiles from memory is not significantly

faster than doing decoupled loading from hard disk and decompression. A conclusion to draw from this result is that it does not make much sense to cache JPEG or PNG compressed tiles. Since it is just as fast to load and decompress them, one should rather cache the tiles uncompressed (or DXT compressed), even if less tiles fit into the cache. The situation is completely reversed on gaming consoles, where latency and bandwidth are far worse (when streaming from an optical medium) and RAM is limited. It is also possible to use both caches in combination, see Section §4.5.5.

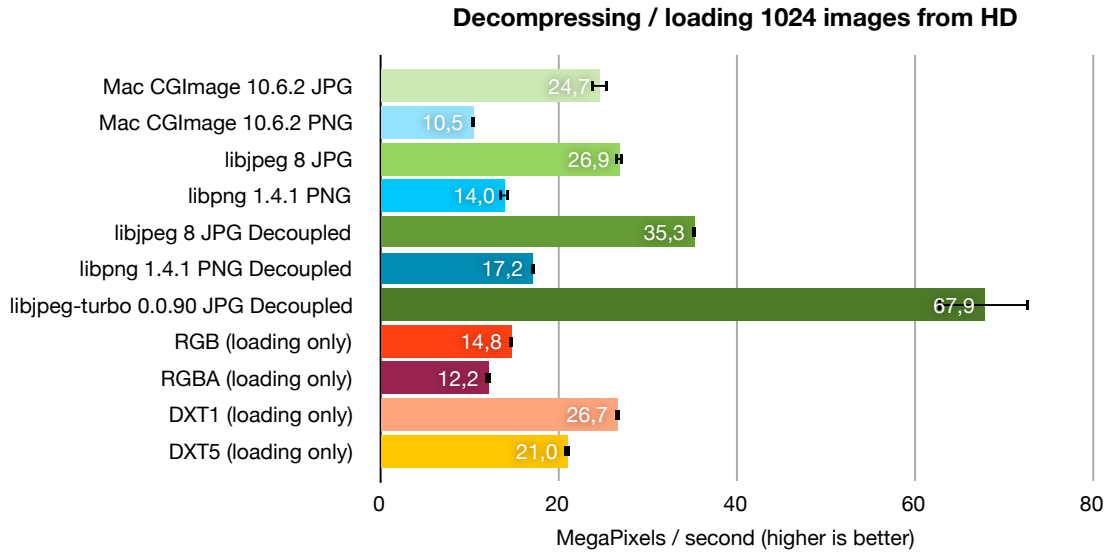

Fig. 4.12: Throughput when loading and decompressing from hard disk. The uncompressed and DXT compressed tiles are only loaded.

As discussed in Section §4.1.5, it may be beneficial to store multiple or all tiles in a single file, and read multiple tiles in one batch. A request scheduler could combine multiple requests, sort them according to disk layout, etc. No tests have been done measuring performance of this technique.

There are also caches outside of the virtual texturing application: there is the disk cache built into the hard disk ("disk buffer"), additionally the operating system caches recently used files in RAM ("page cache"). Since we concluded above that we do not want to cache compressed tiles, it makes sense to prevent the operating system from caching the files. This allows the whole available RAM to be used to cache decompressed tiles within the application. On Mac OS X one can turn off caching for a file descriptor using:

```
fcntl(file_descriptor->_file, F_GLOBAL_NOCACHE, 1);
```

Mittring states that under Windows, I/O Completion Ports can be used for efficient file access with disabled page cache [MG08].

Figure 4.12 also features loading times for uncompressed and DXT compressed formats in addition to the decompression times. It can be seen that throughput of uncompressed tiles is much lower even though no decompression is necessary, because so much more data has to be loaded. Even PNG provides higher performance. Tiles that are pre-compressed in DXT provide better performance than uncompressed or PNG, they are on par (DXT1) or below (DXT5) slow JPEG libraries. The upshot of using DXT pre-compressed tiles is an extremely simplified texture streaming system. Additionally, offline DXT compression algorithms generally provide much higher quality than real-time algorithms. In the JPEG case the artifacts of the lossy JPEG compression are added to the artifacts of the real-time DXT generation. When using virtual texturing in a controlled environment where storage space is not an issue and the highest possible throughput is not necessary, DXT tiles might be an acceptable choice. Fast disks or SSDs can also greatly help with the throughput – DXT1 loading on a fast SSD could even outpace libjpeg-turbo.

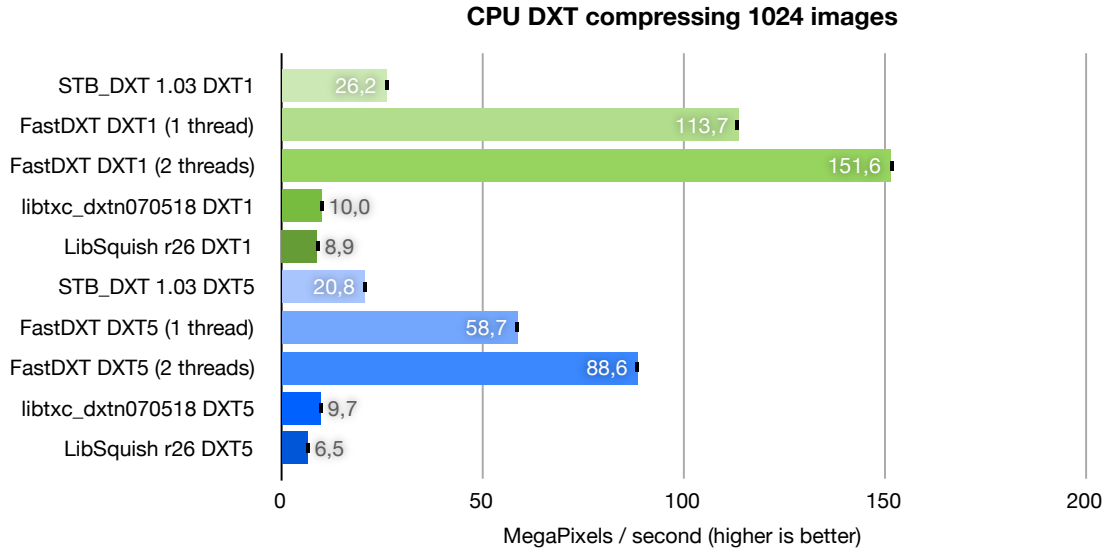

Fig. 4.13: The performance of publicly available CPU DXT compression libraries.

So far we have only covered the loading and decompression performance. Converting the tiles to DXT before storing them in the physical texture gives several benefits: significantly reduced memory requirement for the physical texture, increased performance when streaming the tiles to the physical texture and the fact that more tiles can be cached in RAM. Downsides are quality reductions (especially because of the real-time

compression) and the further reduced tile streaming performance. Figure 4.13 measures the performance of commonly available high-performance DXT compression libraries that perform compression on the CPU. The NVIDIA Texture Tools were not measured because its CPU compressor only wraps the already benchmarked libraries. ImageLib [Dor] had too low performance to be included. According to [Cor10] the fastest available CPU DXT compression code is "Extreme DXT Compression" [Uli], ranging from 206 - 910 MP/s, but we could not run the provided code. The performance of the measured libraries varies widely, depending on optimization and quality/performance trade-offs. All libraries were benchmarked with their standard settings, only squish [squ] was used with the "kColourRangeFit" setting to improve performance. [Cor10] compares performance of CPU and GPU compressors, showing that GPU compressors are an order of magnitude faster (up to 13000 MP/s). The disadvantage of GPU-based compressors in a virtual texturing pipeline is that no bandwidth is saved when transferring tiles to the GPU, and, unless the compressed tiles are read back to the CPU, no space is saved in the RAM tile-cache.

Real-time DXT compression is a well researched subject [vW06a] [vWC07], extensions for compressing normal maps have also been researched [vWC08]. [Blo08] provides a detailed technical overview of the performance and quality of different DXT compression implementations.

After evaluating the performance of commonly available libraries for JPEG / PNG decompression and DXT compression on the CPU, we conclude that the fastest measured libraries are suitable for the texture streaming pipeline in a virtual texturing system. A satisfactory pipeline can be built using libjpeg-turbo and FastDXT [Ren] (or, presumably, Extreme DXT). If tile storage size is an issue, one should look at formats providing higher compression ratios. In 2006, Waveren concluded that CPUs were not yet fast enough for real-time tile streaming of tiles in more advanced formats but might become fast enough later:

> "As faster CPUs and systems with more CPUs or cores become available it will become advantageous to use compression formats that achieve better quality and higher compression ratios at the cost of more expensive decompression. As more CPU time becomes available compression formats like JPEG 2000 and HD Photo typically achieve acceptable quality at higher compression ratios and as such improve the streaming throughput [...]" [vW06b].

JPEG 2000 decompression results are not included in the results because the available open source decompression libraries are not tuned for performance yet. In a virtual texturing pipeline there is no restriction to existing formats for tile storage. In [vW06b] decompression performance of a format which is very similar to JPEG is discussed. The

DXT5 compression can be modified to use the YCoCg color space for improved quality [vWC07].

An interesting option for a texture streaming pipeline would be to do the decompression and re-compression completely on the GPU. This would minimize the necessary upload bandwidth to the GPU, and many more tiles would fit into the RAM tile cache. GPUs are already proven to be more than 10 times faster at DXT compression [Cor10]. However, decoding a complex format like JPEG on the GPU is not trivial and there are non-parallelizable sub-tasks that do not map easily to the GPU programming model. There is one project which does JPEG decoding on the GPU using CUDA [Din09]. An additional possibility is to explore the usage of the new texture compression formats BC6/BC7 (ARB_texture_compression_bptc).

4.3.2 Physical Texture Updates

The physical texture is used to store the tiles of the virtual texture that are currently necessary for rendering. Updates depend on how exactly the tiles are stored. We only consider methods of storage that can apply (hardware) texture filtering during the texture fetch:

- Single 2D texture: Using a single 2D texture is the method Mittring and Barrett use. The tiles are stored in chequer pattern in the texture, tightly packed. 2D textures are ubiquitously supported and feature all filtering modes. The limiting factor here is the maximum size for a single 2D texture, which ranges from 2048^2 (Intel GMA / integrated graphics chips), 4096^2 (older hardware), 8192^2 (new hardware) to 16384^2 (new ATI hardware with very new drivers). Depending on the scene, viewport size and texture atlas layout the actual requirements range from 2048^2 up to over 8192^2. The method works fine if the maximum texture size the hardware/driver provides matches the requirements.

- Multiple 2D textures: Using multiple textures is possible (either by fetching both textures and masking the result or with a conditional) but slow [MG08].

- 3D texture: 3D textures can be used to overcome the size limitation of 2D textures, but it is not possible to enable filtering only in two directions. When sampling directly from the slices the results should still be correct, but the performance may be even worse than multiple 2D textures.

- Texture arrays: Texture arrays would be the ideal way to store the necessary tiles (although Mittring finds them to be a “bit” slow [MG08]), but unfortunately many hardware/driver combinations do not provide enough array layers to store one tile

per layer. (For comparison: a 8192^2 2D texture can hold 1024 256^2 tiles or 4096 128^2 tiles.) NVIDIA only offers 512 layers, Intel does not support texture arrays at all and only recent ATI hardware can support 8192 layers.

Figure 4.14 shows the results of a simple benchmark measuring the sampling performance from different texture sources.

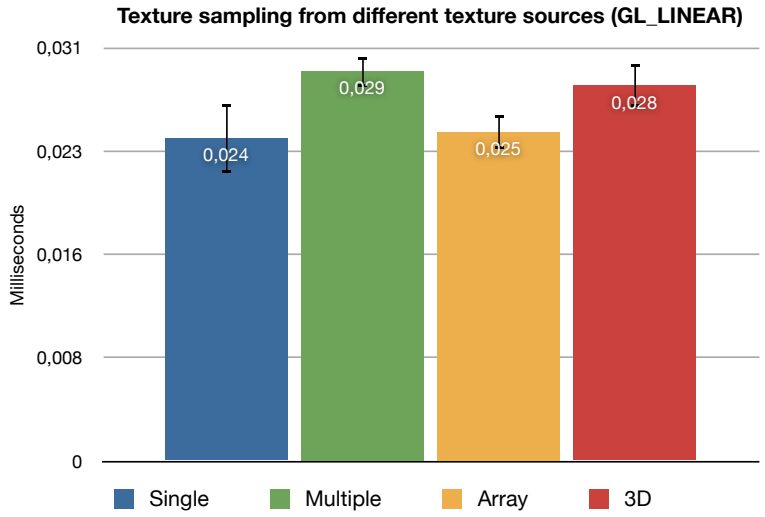

Fig. 4.14: Simple benchmark measuring the average time for a render pass (1920 x 1057) consisting only of texture fetches from a specific texture source.

If the size restriction of the chosen physical texture storage method poses a problem, there are at least four solutions:

- Dynamic LoD adjustment: One can dynamically adjust the bias used for the mipmap-level calculation in the tile determination system so that the number of necessary tiles is reduced until they fit into the physical texture [vW09b] [MG08]. This results in uniformly blurred output.
- Dropping “unimportant tiles”: If the physical texture is full, tiles that are deemed to contribute only marginally to the visual quality can be dropped (as discussed in Section §4.5.9). This results in some parts of the scene being blurrier.
- VRAM cache: An additional buffer in graphics memory, from which tiles can be transferred to the physical texture very fast, is used [MG08]. Tiles that are not currently visible but might become visible soon are transferred to the physical texture on demand [MG08]. This is a solution to the predicted tiles not fitting into the physical texture as opposed to the necessary tiles.

- Multiple virtual textures: It is straightforward to use multiple virtual textures in an application (e.g., one for terrain and one for objects), just like multiple standard textures are used nowadays. Although this necessitates multiple draw calls, this should still be a reduction of draw calls. Each virtual texture has one associated pagetable texture, the physical texture can either be shared or be unique to a virtual texture. Tile determination in view space can be extended to determine tiles for multiple virtual textures in one pass [Bar08], see Section §4.5.15 for more information.

Regardless of the texture storage method (2D texture, texture array, etc.), a slot for new texture tiles has to be found. If no slot is free, tiles that are not visible anymore can be replaced, for example with the "least recently used" principle. Mittring discusses several problems and solutions for updating parts (tiles) of a 2D texture, but they are specific to Direct3D [MG08]. We experienced no problems updating the physical texture using `glTexSubImage2D()`. Pixel buffer objects can be used to hide the latency of updating the texture, but since there is a variable number of new tiles per frame, a threshold for the PBO buffer size must be set. This is a compromise between memory waste and the maximum amount of new tiles per frame. One can also transfer the remaining tiles in frames with many tiles synchronously.

Texture memory on the PC/OpenGL architecture cannot be directly written to, and the texture updates are performed through OpenGL with `glTexSubImage2D()`. Since there is one context per thread in OpenGL this means that the physical texture updates either have to happen on the main OpenGL thread (which is different from the thread that loads the tiles) or the physical texture has to be shared between the main context and another context on another thread. We have not explored updating the physical texture from a shared context.

Finding a slot in the physical texture (amongst other tasks) necessitates a data structure that holds information about all tiles stored in the physical texture. A simple two dimensional array of a structure containing the identifying tuple (x, y, mip-level) and the `lastVisibleTime` sufficed in our case.

4.3.3 Pagetable Texture Updates

Each tile of the virtual texture is represented by a pixel in the pagetable texture that stores the position of the corresponding tile in the physical texture. Because of this 1:1 correspondence between the tiles of the virtual texture and the pixels of the pagetable texture, the pagetable texture has as many pixels on its lowest mipmap-level as there are tiles on the lowest mipmap-level of the virtual texture. If the virtual texture has 512^2 tiles at the lowest mipmap-level, the pagetable texture is a 512^2 pixel texture Additionally,

the pagetable texture is mipmapped (because the virtual texture is mipmapped too), and the pixels of the mipmap-levels of the pagetable texture correspond to the virtual texture tiles of the same mipmap-levels. For more details see the Section §4.4. The pagetable texture has to be updated every time a new tile is added to the physical texture (there are no removals, only replacements). The pagetable texture and the physical texture do not have to be synchronized during the virtual texturing rendering. The pagetable texture updates can happen on the same thread as updates of the physical texture, which should be the main rendering thread unless shared contexts are used.

Updates of the pagetable depend on storage of "fallback entries" in the pagetable. If fallback entries are used, the pixels corresponding to the tiles that are not currently stored in the physical texture are filled with the coordinates of "fallback tiles". Usually the highest resolution tile above the missing tile in the mipmap-chain that is currently available is used as a fallback tile. Keeping the top tile of the mipmap-chain in the physical texture all the time ensures the existence of a valid fallback tile at all times. During rendering, a graceful fallback to lower resolution tiles happens for those tiles that are not available because they have not yet been loaded, or have been missed by the tile determination system. If no fallback entries are used, the fallback happens with looping in the fragment shader, and the addition of a single tile to the physical texture only necessitates the change of a single pixel in the pagetable texture. If fallback entries are used, all pixels in the mipmap pyramid below the pixel have to be updated in the worst case. More specifically, when mapping a tile, pagetable entries "below" the mapped tile are evaluated recursively and if they are not "mapped" but store a fallback-entry, the fallback-entry is updated. The recursion stops when encountering a mapped tile. When un-mapping a tile, first the replacement fallback tile must be determined, it is stored in the tile above the tile to be unmapped. Then the recursion replaces the fallback entries from the to-be-unmapped tile with the other tile. Recursion again stops when encountering a mapped tile. Usage of fallback entries complicates and slows down the pagetable texture updates.

The information stored in the pagetable texture is necessary not only on the GPU during rendering but is also needed in the CPU-based code. For example the tile determination system needs to know if a tile is already mapped. It is possible to use the same format and structure for the pagetable on both the CPU and GPU. This saves the cost of conversion and synchronization.

Updating the pagetable texture is a compromise between the number of OpenGL calls and the wasted bandwidth & fill-rate. Minimizing draw calls can be done by replacement of the whole pagetable texture. Bandwidth & fill-rate can be minimized by updating each changed pixel in the texture individually. Performance-wise the optimum is somewhere in between, depending on the hardware and software setup.

Actual updating can happen either with `glTexSubImage2D()` or by rendering

points / quads. When using `glTexSubImage2D()`, PBOs can be used to let the transfer happen asynchronously. We implemented a simple compromise between draw calls and bandwidth & fill-rate: For each mipmap-level of the pagetable texture, the minimum and maximum coordinates of modified pixels are remembered. Only modified mipmap-levels are uploaded using `glTexSubImage2D()`, and only between the minimum and maximum pixels. This results in a worst case of updating a whole mipmap-level for just two pixels in opposing corners. Barrett developed a more sophisticated method for pagetable updates because his pagetable is up to 4092^2 in size and he is using fallback entries [Bar08]. Another interesting option is to create the pagetable texture entirely on the GPU using geometry shaders, as described in [HPLdW10].

4.4 Virtual Texturing Shader

The virtual texturing shader has to transform the virtual texture coordinates (the ones the geometry is textured in) to the physical coordinates, so that the right tile (in the physical texture) is sampled at the right position. If the required tile is not in the physical texture, a fallback to a lower resolution version is used. The "trick" to be able to do that efficiently is the pagetable texture. The pagetable texture stores the position of the tile in the physical texture for every tile in the virtual texture. The pagetable is a mipmapped texture that has a pixel for every tile in the virtual texture. If the virtual texture has 512^2 tiles at the lowest mipmap-level, the pagetable texture is a 512^2 pixel texture. Every pixel of the pagetable texture corresponds to a tile in the virtual texture and stores the coordinates of the corresponding tile in the physical texture. If the corresponding tile is not stored in the physical texture, it contains the coordinates of the fallback entry (if those are used). The virtual texturing shader samples the pagetable texture (always without filtering, i.e., GL_NEAREST) with a bias balancing the size differences between the pagetable texture and the virtual texturing shader. All lookups for virtual texture coordinates that correspond to a specific virtual texture tile get the pixel from the pagetable texture that holds the coordinates of this tile in the physical texture. Figure 4.15 tries to illustrate this relation. For pixels that should be textured with the tile from the rightmost bottom (and mipmap-level 0) of the virtual texture, the lookup into the pagetable texture also goes to the rightmost bottom pixel, where the coordinates of the corresponding tile in the physical texture are stored.

This explains how the fragment shader determines the coordinates of the visible tile in the physical texture, but the coordinates within the tile are also necessary. The within-tile coordinates (relative to the corner of the tile) depend on the mipmap-level of the sampled tile because at higher mipmap-levels the same coordinate space is covered with fewer pixels [Bar08]. The straightforward but inefficient way to calculate the within-tile coordinates is to convert the texture coordinates to be measured in pixels (instead of

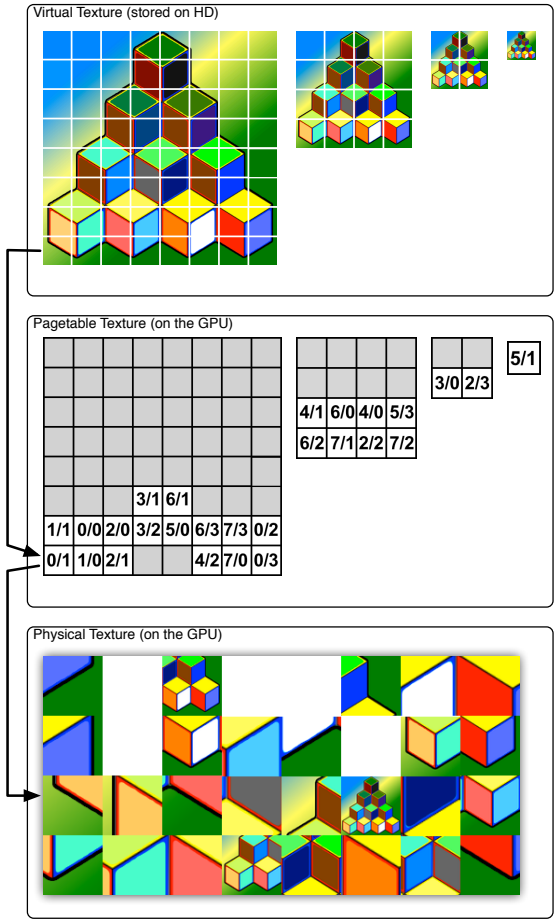

Fig. 4.15: Illustration for the relation between virtual texture, pagetable texture and physical texture.

going from zero to one) and then mod them by the tile-size [Bar08]. The faster option is to calculate `exp2()` of the reverse-mipmap-level, multiply this by the virtual texture coordinates and take the fraction of the result [Bar08]. The reverse-mipmap-level is 0 for the highest mipmap-level, 1 for the second highest, etc.

Program 4 shows a simple unoptimized GLSL fragment shader for virtual texturing. The division can be easily replaced with a multiplication and the `exp2()` can also be avoided with constant or uniform arrays, or by passing the pre-calculated value if the mipmap-chain is shorter than 10. By using a pagetable texture in floating point format, most of the computation can be avoided altogether (since it is page-constant) and the shading overhead of virtual texturing can be reduced to a dependent texture-sample and a multiply-add instruction – at the cost of graphics memory [Bar08]. Adjustments are necessary for bilinear/trilinear/anisotropic filtering, or if no fallback en-

Program 4 A GLSL fragment shader for virtual texturing rendering. Based upon Barrett's SVT demo shader [Bar08].

```
const float phys_tex_dimension_pages = 32.0;
const float page_dimension = 256.0;
const float page_dimension_log2 = 8.0;

uniform float mip_bias;
uniform sampler2D pageTableTexture;
uniform sampler2D physicalTexture;

// converts the virtual texture coordinates to the physical texcoords
vec2 calulateVirtualTextureCoordinates(vec2 coord)
{
    float bias = page_dimension_log2 - 0.5 + mip_bias;
    vec4 pageTableEntry = texture2D(pageTableTexture, coord, bias) * 255.0;
    float mipExp = exp2(pageTableEntry.a); // alpha channel has mipmap-level
    vec2 pageCoord = pageTableEntry.bg; // blue-green has x-y coordinates
    vec2 withinPageCoord = fract(coord * mipExp);

    return ((pageCoord + withinPageCoord) / phys_tex_dimension_pages);
}

void main(void)
{
    vec2 coord = calulateVirtualTextureCoordinates(gl_TexCoord[0].xy);

    vec4 vtex = texture2D(physicalTexture, coord);

    gl_FragColor = vtex;
}
```

tries are stored into the pagetable texture. When using looping in the fragment shader, it is required to avoid usage of `texture2D()` in the loop to prevent incorrect results because texture sampling is undefined in branches and loops [LK06]. The workaround is to calculate the derivatives before the loop and use `texture2DGrad()` in the loop. Since `texture2DGrad()` does not accept a bias argument, these must be enforced by dividing the derivatives by 2^{-bias}. No adjustments are necessary when using `texture2DLod()` to access the pagetable, this version works fine in loops.

Figure 4.16 compares the cost of a render pass with different shaders. Although virtual texturing adds a dependent texture fetch and some math instructions, the performance of a base virtual texturing shader is virtually identical to "normal" rendering. The numbers are the average milliseconds for the main render pass of the terrain scene, the virtual texture size is $32k^2$ and the texture size for the normal rendering is $8k^2$. The shading cost of virtual texturing rises significantly when modes are used that necessitate gradient calculation (trilinear or anisotropic), whereas these features are indistinguishable speed-wise with normal rendering (trilinear filtering is used in the tests for normal rendering). The overhead of looping in the fragment shader is also noticeable (and rises significantly when more tiles are missing), but it saves costs elsewhere.

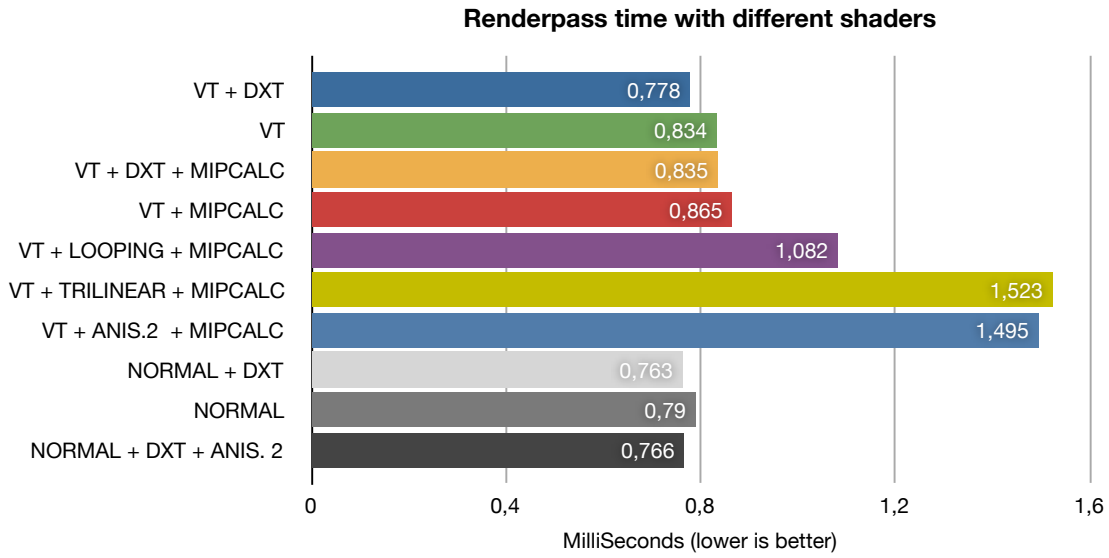

Fig. 4.16: Comparison of the performance of virtual texturing shaders.

4.5 Problems, Challenges, Advanced Features & Miscellaneous

This chapter aims to cover problems and challenges during the implementation of a virtual texturing system that are not directly related to the already covered main parts of a virtual texturing system. Advanced features and miscellaneous issues are also covered here.

4.5.1 The Virtual Texturing Runtime Pipeline

Arguably, the most difficult challenge when implementing virtual texturing is not developing any of the parts like the tile determination or streaming system, but rather combining them in a high-performance pipeline that makes optimal use of the available CPU and GPU resources. Simple examples for speedups in the pipeline are usage of PBOs for asynchronous texture transfers or decoupled tile loading and decompression. More speedups are possible by two frame-alternating PBOs for the same texture transfer or by delaying the tile determination read-back until the next frame. As mentioned, combining the view-space tile determination pass with another render pass is possible and useful if one has a CUDA/OpenCL solution to compress the result efficiently. An optimized pipeline needs to make sure the CPU and GPU are fully utilized and not

stalled waiting for data. The best pipeline setup probably depends on the specific application and settings, virtual texture layout, geometric and shading complexity, hardware, etc. One problem with asynchronous transfers using PBOs is that there is no (built in) way to know when the transfer is finished. Possible solutions are fence objects or doing the transfer synchronously but in another thread with the help of context sharing.

4.5.2 Texture Filtering

The virtual texturing system as described in this chapter only works correctly (as far as visual output is concerned) without hardware texture filtering (GL_NEAREST in OpenGL terms). Problems of hardware-based filtering when using "indirect" texturing are well known and discussed e.g. in [LN03]. Adaptations are required for bilinear, trilinear and anisotropic filtering. Regarding texture filtering, it should be noted that virtual texturing using an unfiltered (GL_NEAREST) physical texture already corresponds to non-virtual textured rendering with GL_NEAREST_MIPMAP_NEAREST because sampling occurs from the closed matching mipmap-levels of the tiles. A virtual texture is always mipmapped and rendering with a virtual texture always uses these mipmap-levels. Virtual textured rendering with bilinear filtering (GL_LINEAR on the physical texture) corresponds to normal rendering with GL_LINEAR_MIPMAP_NEAREST. (The terms "bilinear" and "trilinear" are not adequate for describing all texture filtering possibilities.)

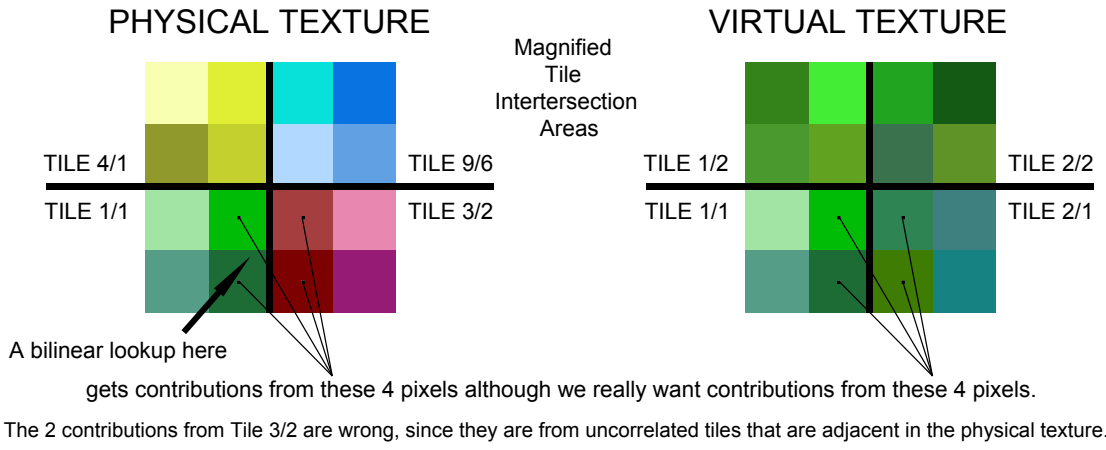

Fig. 4.17: An example explaining why enabling bilinear filtering for virtual texturing results in artifacts at tile boundaries.

Bilinear filtering (aka GL_LINEAR): Let us examine a part of the problem: the physical texture contains de-correlated tiles from different parts of the virtual texture. At tile borders the texture filtering will sample adjacent pixels from tiles that are adjacent in the physical texture but not in the virtual texture. Figure 4.17 tries to explain this with an example. This can result in artifacts since pixels from completely different parts of the virtual texture are used for filtering at tile borders. There are various fixes to this issue: add an empty border around tiles, use texture arrays instead of a single 2D-texture or contract the texture coordinates within the tiles by half a pixel (sampling the dead center of a pixel does not incur a fetch of neighboring pixels for the bilinear filtering). Although these "solutions" fix artifacts from incorrectly sampling into neighboring tiles in the physical texture, they are still not visually correct: the pixel at the tile borders are still not filtered with the correct pixels, those that are adjacent to them in the virtual texture. There is no correct filtering across tile boundaries.

One solution to provide correct bilinear filtering is to add a (1 pixel) border around every tile that contains the pixels from the adjacent tiles. This ensures completely correct bilinear filtering across tile boundaries although it wastes memory [Bar08]. When adding such a border (of any size) there are basically two options:

- Expand the tile-size:
  The border is added to the tile-size, resulting in tiles that no longer have power-of-two sizes, i.e., tiles are 130^2 instead of 128^2 pixels or 258^2 instead of 256^2 pixels. Unfortunately this breaks the memory alignment of the tiles and also destroys the power-of-two size property of the physical texture, resulting in performance problems[MG08]. Padding can be used to circumvent the performance problems, [MG08] but this results in even more memory waste. The benefit of this method is that the (source) virtual texture dimensions are still a power-of-two. It should be noted that there is no public virtual texturing implementation using this method so the exact modifications necessary are not known – some math in the virtual texturing shader depends on the power-of-two properties.

- Retain the tile-size:
  With this method the physical tile-size stays the same but due to the border the actually usable size is decreased, i.e., only 126^2 pixels out of a 128^2 are actually used, except for the filtering. This method does not suffer from the problems of the other approach but requires source virtual texture dimensions that are no longer a power-of-two. Some additional math in the fragment shader is necessary if no floating point pagetable is used – since we want to use only a subset of the tiles now and the border pixels are there just for filtering we need a coordinate contraction for the within-tile coordinates:

$$withinPageCoord = withinPageCoord * (page\_dimension - border\_width * 2.0)/page\_dimension + border\_width/page\_dimension;$$

Mittring mentions that the visual quality suffers a lot when using this method "due to aliasing in the mipmaps caused by the down sampling of the source texture to slightly less than its half size"[MG08]. We cannot confirm this and are not sure what he is referring to – there is only downsampling to exact half sizes going on in our implementation. Perhaps he refers to using this method with a source virtual texture which is still a power-of-two – this should obviously be avoided. For example, when working with $128k^2$ virtual texture using a border of one pixel means the artists can only fill a texture atlas of size 130048^2 pixels instead of the full 131072^2 pixels. A drawback of this method is that since changes of the tile-size or border size change the size of the required source texture atlas, not only the atlas has to be re-generated but also the texture coordinate offsets change, necessitating regeneration of the (texture coordinate adjusted) geometry.

Barrett mentions the possibility of using borders that are only on two sides of a tile, effectively halving the memory waste [Bar08] [Spa10]. The memory waste is not a significant problem when using just a one pixel border for bilinear filtering, but larger borders are necessary for other filtering modes (see below). The additional fragment shader line is changed to:

$$withinPageCoord = withinPageCoord * (page\_dimension - border\_width)/page\_dimension + 0.5/page\_dimension;$$

Trilinear filtering (GL_*_MIPMAP_LINEAR): As noted above, virtual texturing by its very nature already samples from the closest matching tile mipmap-levels, corresponding to GL_*_MIPMAP_NEAREST filtering. "True" trilinear filtering, which interpolates between the two closest matching mipmap-levels (GL_*_MIPMAP_LINEAR), is possible by adding a second, half-resolution mipmap layer to the physical texture. The physical texture already contains tiles from different mipmap-levels, the closest matching tile for each pixel. By adding a half-resolution layer we have two tiles to interpolate from for each pixel – there is no need for the physical texture to have a full mipmap-chain.

There are three options for obtaining the down-sampled texture tiles. One option is to down-sample all the textures after loading and decompression. We use this option and it works fine performance-wise, however this is obviously not feasible for DXT pre-compressed tiles. The second option is to store all the tiles in both full-resolution and half-resolution versions, so that both versions are loaded at once. This only increases the size of the tiles by one quarter. However, a down-sampled version of each tile, except

the tile at the highest mipmap-level, already exists, namely the respective tile at the next higher mipmap-level. Therefore the third option is to always load the tile below too for every tile request and extract the down-sampled version from there (it covers just one quarter of the tile). Compared to option two the size of the virtual texture tile store is not increased, but a single tile request can necessitate loading two tiles. On the upside, always having this lower resolution tile available can be beneficial in case it becomes necessary for rendering.

When updating the physical texture, both mipmap-levels must be updated. The half-resolution tile can either be generated at runtime or stored in the virtual texture (probably only a good idea for DXT pre-compressed virtual textures, since DXT data cannot be down-sampled directly).

Mittring and Barrett mention that the border around the tiles should be increased (to two pixels) for trilinear filtering [MG08] [Bar08]. This makes sense since this still leaves a one pixel border at the half-resolution tile. However, we have not observed any noticeable artifacts when using trilinear filtering with a one pixel border.

Adaptation of the virtual texturing shader is also necessary. Trilinear sampling is dependent on the gradients to select the mipmap-level factors, but these are not correct when sampling from the physical texture at the tile boundaries [Bar08]. The correct gradients have to be calculated and the physical texture must be sampled with `texture2DGrad()` to ensure the correct result. `texture2DGrad()` allows passing a gradient to be passed to the texture fetch to be used instead of the built-in gradient calculation. Also a bias of -0.5 is necessary in the tile determination and when accessing the pagetable texture. This is because we are not interested in "exactly matching" tiles anymore, but tiles that are of a higher resolution, because we are interpolation this higher-resolution tile with its down-sampled version when sampling from the physical texture now. Program 5 shows the calculation of the corrected gradients.

Usage of `texture2DGrad()` requires either GL_EXT_gpu_shader4 or OpenGL 3.0 and therefore is not ubiquitously supported. Barrett developed a workaround that minimizes trilinear filtering artifacts on hardware without `texture2DGrad()` support.

Program 5 GLSL code for calculating the corrected gradients as first published by Barrett [Bar08].

```
float page_unit_to_phys = ((page_dimension - border_width * 2.0) / page_dimension)
                          / phys_tex_dimension_pages;
float gradient_scale = exp2(mip_bias + mip_trilerp_bias) *
                       page_unit_to_phys * mipExp;

vec2 gradx = dFdx(texcoord.xy) * gradient_scale;
vec2 grady = dFdy(texcoord.xy) * gradient_scale;
```

Anisotropic filtering: Virtual texturing with anisotropic filtering is a bit more involved because enabling anisotropy changes the mipmap-level selection [Spa10]. Anisotropy cannot be enabled on the pagetable texture (to get proper mipmap-level selection) because the filtering would result in completely wrong values being returned from fetch – pagetable entries contain the coordinates of the corresponding tiles in the physical texture and cannot be "filtered". Therefore the modified mipmap-level must be calculated in the shader and selected explicitly from the pagetable texture with `texture2DLod()`. Tile determination must also be changed accordingly. Note that the view-space tile determination method with mipmap-level calculation by sampling from a texture (with anisotropy enabled) did not work out for us. Calculation of the mipmap-level using a formula derived from the anisotropy extension specification performs well. Program 5 shows the modified mipmap calculation function. The virtual texturing shader also needs to use the same path as with trilinear filtering, meaning calculation of proper derivatives and sampling of the physical texture with `texture2DGrad()`. What is left is enabling anisotropy on the physical texture and making sure the tile border is at least `anisotropy / 2` pixels wide.

Program 6 Modified GLSL mipmap-level calculation method for the view-space tile determination and for sampling from the pagetable texture.

```
float mipmapLevel(vec2 uv, float textureSize)
{
    vec2 dx = dFdx(uv * textureSize);
    vec2 dy = dFdy(uv * textureSize);
    float Pmax = max(dot(dx, dx), dot(dy, dy));
    float Pmin = min(dot(dx, dx), dot(dy, dy));

    return 0.5 * log2(Pmax / min(ceil(Pmax/Pmin), max_anis * max_anis))
           + mip_bias - readback_reduction_shift;
}
```

"Software" filtering: Of course it is possible to ignore the hardware texture filtering and do the texture filtering in the shader, but this requires 8 texture samples (4 into the pagetable texture, 4 into the physical texture) followed by an interpolation for bilinear filtering [MG08]. [Neu10] determined this approach to be unfeasible from a performance standpoint. Similar, anisotropic filtering is possible. Although this technique does not need additional borders around the tiles the performance likely is not acceptable.

If one uses hardware bilinear filtering, it is possible to extend that to trilinear filtering in the shader [MG08] with just an additional lookup into the pagetable and physical texture. More concretely this would mean sampling the pagetable texture two times with biases 0.5 and -0.5, then sampling the physical texture at both resulting coordinates and interpolating the result in the shader. This way no second mipmap layer is needed for the

physical texture [MG08]. [Neu10] successfully used this method. The tile determination system must be modified to fetch the two closest matching tiles instead of only the single closest tile.

4.5.3 Texture Compression

We have already discussed how (DXT) compression fits into the tile streaming pipeline, but not which additional steps must be performed to render from a DXT compressed physical texture. Using DXT compressed textures is transparent to the shader, and necessitates no adjustments to the virtual texturing shader. However, DXT is a block compression (4x4) algorithm and the color compression is context dependent (the same pixel can compress to different colors in different blocks) [Spa10]. Therefore the border around the tiles must be extended to 4 pixels to prevent artifacts [MG08] [Bar08]. This border width enables an anisotropy of 8 at the same time. Barrett notes that in the trilinear filtering case, the DXT compression may require even (much) larger borders [Bar08] [Spa10]. Using DXT with trilinear filtering is not fully explored and it is not known how visible the theoretical artifacts are in practice.

4.5.4 LoD Pop-in

Even with an optimized tile streaming system there will be cases where a low-resolution fallback of a tile has to be used, especially with a slow storage medium. When the requested tile is finally available, the switch to the high-resolution tile texture will result in a visible pop-in [vW09b] [Bar08] [MG08]. This is similar to the geometric LoD pop-in and the solution is also similar: blending of the old low-resolution data with the new high-resolution data. Waveren states: “If we used trilinear filtering, blending in detail would be easy” [vW09b]. This could be done in the following way: When a new tile arrives, map the half-resolution version (trilinear filtering means the physical texture has mipmap-level one layer filled with half-resolution tiles) with scaled data from the fallback tile. Store a blend factor in the pagetable entry that forces the lookup to the half-resolution version. Gradually change the factor to blend in the high detail tile just fetched. When the blending is done, replace the half-resolution version with the real data.

If only bilinear filtering is used, the blending is more involved. It is still possible to use a blend factor in the shader and gradually blend with a lower resolution version, but the shader cost would likely be prohibitive, not to mention that the next lower resolution tile is not necessarily available. Waveren proposes another solution: up-sample the coarse tile immediately and gradually update and blend the tile with fine data every frame until the blend factor for the high-resolution data reaches one [vW09b]. To save

bandwidth, one can still transfer the new tile to the graphics memory only once (to another buffer) and do the blending only on the GPU.

A completely different method that circumvents the LoD pop-in without dealing with blending is discussed in [CESL10] and [Neu10]: by always first loading all the ancestor tiles from low to higher resolution for a new tile request, the quality is improved more gradually. Since this also makes more ancestors available in the physical texture, this is also supposed to improve quality after rotations, at least in terrain scenes [CESL10].

4.5.5 Tile Caches

Between the necessary parts of a virtual texturing system, the physical texture and the tile storage on disk, various caches at different parts of the pipeline can be used. Figure 4.18 depicts these possible caches. As concluded in Section §4.3.1, it makes more sense to cache tiles as they are sent to the GPU (DXT or uncompressed) and not as they are stored on the disk (e.g., JPEG), at least when using a fast hard drive, and from a "throughput perspective". But this is not a mutually exclusive decision. It is possible to use both caches together to unburden the hard disk bandwidth. Additionally, it is possible to cache tiles in VRAM as Mittring mentions [MG08].

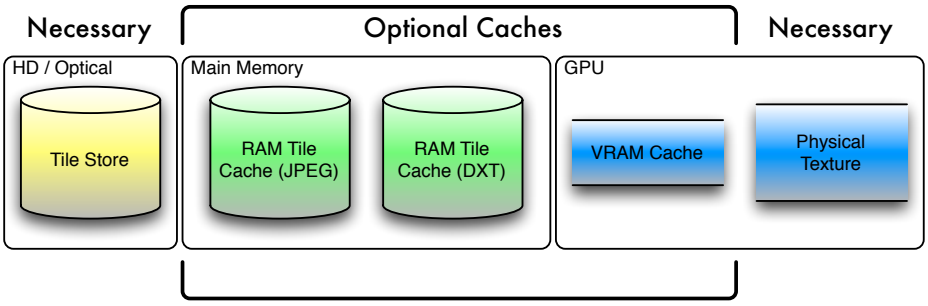

Fig. 4.18: Illustration of the various possible caches sitting between the tile store and the physical texture.

To manage RAM caching, additional data structures are needed for their replacement strategy. The RAM cache could be managed with an LRU algorithm, possibly with a bias towards tiles from higher mipmap-levels. These data structures are used from the tile determination system (to update the time of visibility), the tile streaming system (to insert pages into the cache) and the rendering system (to transfer cached pages to the GPU) and therefore could be considered a fourth integral part of the virtual texturing system. The access from different threads mandates locking of the data structure. When

using tile determination in view space without buffer reduction, it is necessary to iterate over a quite large list of mostly similar pages to either touch their modification date or request them for streaming. The cache management should be developed with special emphasis on efficiency, since touching the cache management data structure can become one of the bottlenecks of the virtual texturing system.

4.5.6 Texture Atlas Problems

Unless virtual texturing is used to render a single mesh with a single texture, there is always a texture atlas involved. There are some well-known problems when using a texture atlas for rendering [Cor04], as well as some new virtual texturing-specific problems.

Virtual Texture Atlas Layout

The unwrapping and the layout of the sub-textures in a virtual texture (atlas) is one of the most important factors contributing to the performance (and by extension, the visual quality) of a virtual texturing system. See Section §4.1.4 and §5.2.1 for a discussion of this vital point.

Filtering Problems

Problems that apply to texture atlas rendering apply to virtual texturing (with an atlas) too and have to be handled. One problem is mip-chain pollution, which can be avoided by a variety of measures [Cor04]. A second problem is that most objects are incorrectly textured with the full (0-1) range for convenience (prior to coordinate offsetting), leading to sampling into neighboring textures even without texture filtering. This can be avoided by additional coordinate adjustment in the atlas tool [Cor04]. Another problem is that the texture filtering samples into different physical textures, resulting in artifacts [Cor04]. This problem is related but not exactly the same as the problem mentioned in the Section §4.5.2. To prevent artifacts here, we only need to prevent sampling across the boundaries of individual textures (and not provide continuity across borders). Virtual texturing provides us with a mechanism to combat texture atlas artifacts that normal rendering does not have – the tiles and their borders. If the individual sub-textures are aligned with tile boundaries, we can mark these edges and construct the border not from the adjacent tile (because it contains a different sub-texture that we do not want to filter into) but duplicate the outer pixels of the tile in the border. Alternatively one could contract the texture coordinates for these tiles in the virtual texturing shader, so the border is not sampled for these tiles, but this would result in a runtime performance hit. In

any case, the texture atlas layout algorithm should prefer placing individual sub-textures at tile border, so that this artifact prevention can be used. Padding can be used where sub-texture borders do not align with the tile borders. Any artifact reduction technique should only be applied if artifacts resulting from the texture atlas mechanism have been observed and found to be a problem.

4.5.7 Virtualized Pagetable Texture

As discussed in Section §4.1.2, one limiting factor for the maximum virtual texture size is the pagetable texture. Small tiles (less waste) and large virtual texture sizes are desired, but this results in increasing memory waste for a large pagetable texture and the performance problems with updating it. Supposedly there is an upcoming solution for large textures: virtual texturing! The pagetable texture can be virtualized to save memory. There is no need for an additional "tile determination" system because it is already known which pixels of the pagetable texture will be accessed (when using fallback entries). The necessary pixels of the pagetable texture are exactly those that correspond to currently visible tiles. The modified system would look like this: the shader looks into a "pagetable for the pagetable" texture to get the address of the right pixel in the "pagetable physical" texture. The values of this pixel are then used to determine the coordinates for sampling from the (real) physical texture, just as with "normal virtual texturing". The additional shader cost compared to normal rendering would effectively double. One thing to consider is that the higher mipmap-levels of the pagetable texture do not contain coarser versions of the data of the lower levels, therefore a fallback mechanism makes no sense and is not possible. The "tile-streaming" has to occur synchronously, so the textures are fully updated and current each frame. Since no filtering is desired for the "pagetable physical" texture, the results should be completely correct.

4.5.8 Texture Virtualization for Arbitrary Textures

Section §4.5.7 demonstrates that even the pagetable texture can be virtualized. But can any texture (e.g., textures used for global illumination data) be virtualized? In fact, any texture can be virtualized as long as a few (practical) restrictions are met:

- It can be determined in advance which parts of the texture will be required in this frame, i.e., a tile determination system is possible.
- We can effectively calculate or load tiles of the texture, i.e., a tile streaming system is possible.

- The various artifacts that can occur for various reasons with virtual texturing do not pose a problem. (Though, most of these can be worked around or depend on the required filtering. A virtual texturing system with synchronous tile streaming for an unfiltered texture should be completely correct. However, synchronous streaming is not fast enough in most cases.)
- Only a small part of the texture is visible/necessary each frame and the necessary parts have a high temporal coherency.
- The overhead when accessing the texture through indirection is acceptable.

One texture commonly used in real-time computer graphics where a really high resolution is desired, although only a small part of it is actually required each frame, is the shadow-map. There are parts of a shadow-map that are accessed by a relatively large portion of the screen and thus needed in high resolution, other parts are only needed in low resolution or are not visible at all. Classic solutions try to combat this problem with different parametrization or by splitting the shadow map into smaller parts [SWP10]. Newer and more complex adaptive algorithms like "Adaptive Shadow Maps" or "Resolution Matched Shadowmaps" already work similarly in spirit to a virtual texturing system: they analyze the scene to determine which tiles of the shadow-map are required [SWP10].

Using virtual texturing for shadow-mapping would entail a tile streaming system that renders from the viewpoint of the light source to generate a tile. One could render directly to the physical texture, so the actual streaming to the GPU would be omitted. The tile determination part could be combined with the view-space tile determination pass. One difference to normal virtual texturing is that all tiles would have to be invalidated when the light source moves and some, when objects in the scene move. The performance of such a system would likely depend on the number of required new tiles per frame, since each tile request necessitates a render pass. Since virtual texturing already provides a way to store high-resolution light-maps as a means to shadow static objects, it is questionable if virtualized shadow-maps are useful in addition.

Other possible candidate textures for virtualization are high-resolution height-maps. When combining height-map-based terrain rendering with geometry-clipmap style LoD, only parts of the height-map are required in full resolution, those around the center of interest. The required resolution would decrease in concentric circles or squares around the center of interest. Tile determination for virtualized height-maps would therefore be a simple, fast analytical calculation on the CPU. Although virtual texturing could be applied in this case, it does not seem necessary: a 4096^2 height-map already provides enough data for a height-field with 33 million triangles (without LoD).

4.5.9 Tile Importance

When replacing tiles (in the RAM cache(s) or the physical texture) or when streaming tiles, it would be beneficial to define a metric for the "importance" of a tile. Although streaming tiles in FIFO order and replacing tiles with the LRU strategy works, a scheme that considers the importance of a tile is expected to perform better. A possible definition of importance is dependent on the use case:

Tile streaming and replacing a used tile in the physical texture: (Note: used tiles are only replaced in the physical texture if no other method like automatic LoD adjustment is used, see Section §4.3.2.) The importance of a tile in this context is the visual impact that results from the existence of this tile in the physical texture. The visual impact of a tile is dependent on the number of pixels the tile occupies on the screen and the difference of the mipmap-level of the tile to the mipmap-level of the fallback tile that would be used instead. Although these two factors are obvious, the exact visual impact cannot be calculated since it is a function of the human visual and perceptual system. We could define the visual impact for a tile like this:

$$vi(tile) = p(tile) * d(tile) \tag{4.2}$$

Where `p(tile)` is the number of pixels on the screen the tile would occupy and `d(tile)` is the difference in mipmap-levels to the fallback tile that would be used. To improve the result, we could modify `d(tile)` to take the perceived image difference between the mipmap-levels into account and not only the difference in mipmap-levels. While building the virtual texture, the perceived image difference between the mipmap-levels (scaled to the same size) should be pre-calculated (with an algorithm like "Structural SIMilarity" (SSIM) [WBS+04]) and stored with the virtual texture. [Neu10] also had the same idea, they calculate and store the "root mean squared errors", which they call NoiseValues, during the virtual texture creation for improved tile prioritization during streaming. They also include a detailed analysis of the visual quality when different tile importance metrics are used.

When streaming tiles, we can give preference to those with the highest visual impact, similarly replacements are done on tiles with the lowest visual impact. Simpler approximations for the visual impact use just the number of pixels on screen [CESL10] or the absolute mipmap-level [Bar08]. Although dropping high-resolution tiles first from the physical texture has been mentioned in both [CESL10] and [Bar08] we do not believe it can be asserted that they contribute less to the visual quality than lower resolution tiles in every case.

A very important factor that we left out here is time. A requested tile likely will not be available within the same frame, and can even take multiple frames to load. Therefore we are actually interested in the visual impact that a tile will make when it arrives

from streaming and in all subsequent frames where it is visible. Unfortunately it is impossible to calculate these values exactly in interactive simulations. However, certain methods and heuristics can try to predict the future importance. [Neu10] experimented with a heuristic called "HotSpot" designed to help with tile streaming during rotations, it gives preference to tiles located on the side of the screen where the rotation is headed to, and de-emphasizes tiles on the side of the screen where the tiles are becoming invisible. They also experiment with performing the tile determination pass from an extrapolated camera position. Their results show that "HotSpot" helps with rotations, but not movements, and the "LookAhead Camera" helps overall, but has some problem cases where it performs worse then no prediction at all, e.g., when a fast rotations suddenly comes to stop.

Screen space tile prediction: We propose a new method for predicting the future tile importance: "screen space tile prediction". Our idea is to calculate the screen-space center of each tile, which should be easy and fast to perform, unless a buffer reduction method is used. The comparison of the tile center positions with their last-frame values results in the movement vectors for the tiles. We now look at tiles that are located near the screen borders: if their movement vectors point towards the screen edges, they likely will become invisible and their importance can be decreased. Conversely if their vectors show that they are moving to the screen center, their importance should be increased. This method is designed to modify the already calculated visual impact (e.g., from the number of pixels on screen), and not serve as the only metric. In contrast to the "HotSpot" method, this should give benefits also during movements and exhibit no worst cases like the "LookAhead Camera". The "HotSpot" method also favors tiles in the screen center if no rotations are occurring, leading to the fact that important pages that can be missed if they are moving from the screen edges towards the center, e.g., during backwards movement of the camera. Another benefit due to the screen-space nature of the method is the handling of dynamic objects. A problem with this method is that it is based on the assertion that most tiles contain texture-data from a single sub-texture, or that if multiple sub-textures occupy the tile, either only one of them is visible on screen or that they are correlated in their world-space and therefore also screen-space positions. If multiple sub-textures that are placed on the tile (texture atlas!) are simultaneously visible at different regions of the screen, the concept of the screen-space center looses its meaning. However, given that the size of the sub-textures used to construct the virtual texture likely is larger than the tile size, and the texture atlas tool should strife for tile-aligned sub-textures and texture to world space similarities anyway (see Section §4.1.4), this seems like a reasonable assumption.

Replacing an unused tile in the physical texture or RAM cache: In this case the importance of the tiles is different, because they are not currently visible, but might

become visible again. Ideally we want to replace those tiles that will not become visible again while retaining the tiles that will become visible soon. Additionally, we prefer to replace those tiles that will have a low visual impact when becoming visible and retain the tiles that will have a high visual impact. Most of these values are impossible to calculate since there is no way to know when the tiles will become visible and what fallback tiles are available when they become visible. Heuristics can be used, e.g., managing the tiles in the cache with an LRU strategy and preferring to drop high-resolution tiles. When using a conservative tile determination system (or with the special case of terrain) information from the tile determination system can be used to obtain the likelihood of the tiles becoming visible again.

4.5.10 Tile Request Substitution

We devised an improvement for a virtual texturing system named tile request substitution. Imagine this scenario: a rapid camera movement or some other event leaves a significant screen portion left with only a coarse fallback tile. Since the necessary tile is not available in the cache, it is requested for streaming and will take several frames to arrive, resulting in degraded image quality for this time. It is possible some higher resolution fallback tile (perhaps even the mipmap-level just below) is available in the RAM cache and could be placed in the physical texture quickly to drastically reduce the image quality reduction until the necessary tile is available. The downside is the added complexity (also with respect to removal of those pages from the physical texture) and also a choice has to be made up to which difference in mipmap-levels this approach is useful. The coarse-to-fine tile loading discussed in [CESL10] could lead to a similar positive effect if implemented in a way that allows higher resolution tiles to outpace coarser tiles when already available in the RAM cache.

Additionally, this idea can be combined with the replacement of visible tiles in the physical texture. Instead of dropping single "unimportant" tiles without replacement (see Section §4.5.9), multiple adjacent cached pages are dropped, but the tile directly beneath them in the mipmap-chain is uploaded instead. This guarantees only a slight quality decrease and saves up to three tile-slots in the physical texture. This is also mentioned in [CESL10], but they only do this with tiles of the highest resolution.

A related idea is generating requested tiles on the fly by downsampling higher resolution tiles, if all 4 corresponding higher resolution tiles are cached.

4.5.11 Recursive Virtual Textures

As noted in Section §4.1.2, the pagetable texture size (combined with the tile-size) limits the effective virtual texture size. But if the fixed pagetable texture is replaced with a

more generic quadtree structure, it could be allowed that quadtree nodes specify their ancestors as their children, creating a recursion in the pagetable quadtree structure. When the corresponding tiles are specially crafted self-similar tiles (e.g., ones for " rock" or "bark") this could simulate infinite texture detail. This is analogous to the "Infinite Surface Detail" idea from Olick for voxel ray-casting [Oli08]. Baked shadows would pose a problem with this idea, and accessing such a quadtree structure is likely much more expensive than a simple texture lookup.

4.5.12 Modifying the Virtual Texture

Runtime modification of a virtual texture is an expensive operation that should be avoided except when building an editor for a virtual textured scene. As discussed in Section §4.1.3, an efficient pipeline for creating scenes for a virtual texturing rendering system could require such an editor. Other cases that seem to necessitate virtual texture modifications possibly can be handled with decals or in some other way. Changing a tile at the lowest mipmap-level (and only those should ever be modified, since the other levels are derived) at least requires updates of the corresponding tiles in all higher mipmap-levels. Only affected parts of higher mipmap-levels need to be touched, e.g., updating a tile only necessitates refreshing 1/4 of the next tile and 1/8 of the next but one tile, etc. Once the changes contribute to less than a pixel, the filtering must also consider pixels from unchanged parts of the virtual texture. Mittring explores different kernels used for mipmap generation for virtual texture and their properties and speed in detail [MG08]. The affected tiles need to be changed on disk and in the physical texture. Doing runtime changes requires the virtual texture to be stored in an uncompressed or lossless format so the compression artifacts do not add up. Modifying tiles likely can still be done at interactive frame rates, but modifying the layout of the texture atlas, that composes the virtual texture, can be much more expensive. Removing objects from the scene or adding objects if there is still enough free space in the virtual texture can be done with minimal updates when accepting sub-optimal virtual texture layout until a full-rebuild. A full rebuild requires reading in all (uncompressed) tiles from the base-level which can easily surpass 50 GB for complex scenes and therefore takes a few minutes at the very least. Increasing the size of the texture for an object (or changing the UV unwrapping) is not possible without relocation unless the sub-texture is surrounded by free space or free space of the necessary size is available elsewhere. Adaptation of texture coordinates is necessary too and can be done at runtime with texture matrix manipulation when breaking up the draw calls. Building such an editor is not trivial.

4.5.13 Decals

Static decals can be baked into the virtual texture just like the light-maps, because the scene is uniquely textured anyway. However, in games/simulations there is also the need for dynamic decals (skid marks, dirt, blood, etc.) to be added to the scene. Rendering decals in another render pass is unwanted and can be avoided by adding the decals to the virtual texture tiles (with different scales and coordinates depending on the mipmap-level) when they are loaded. Adding decals directly to the virtual texture (tiles) is only possible if a unique unwrapping is used [MG08]. Already loaded and affected tiles must either be invalidated and re-loaded or the decals are rendered to the physical texture with different scale factors to the affected tiles. It is also possible to cut the CPU-based decal application and only do the GPU-based rendering of decals to the physical texture, but in this case all decals must stay in graphics memory (instead of main memory) all the time. Similar to runtime modification of the virtual texture (Section §4.1.2 above), applying a decal affects at least as many tiles as there are mipmap-levels, but there are several fundamental differences between both techniques.

Aim: Decals do not want to change the virtual texture as stored on the hard disk but just change the rendered result, since decals are a dynamic addition to the scene.

Scope: Decal application only wants to change the texture data and never the texture atlas layout.

Complexity: Since decal application is the simple addition of a limited number of small alpha blended additional textures it can be done easily during tile loading. In contrast, an editor for a virtual textured scene may desire to make a larger number of more complex changes to the texture data, making it "cheaper" to actually change the tiles on the disk. However, a decal application system could be beneficial in addition – if there is only a small number of limited changes they are added via the decal system, but if the number of changes crosses some threshold the changes are rendered to virtual texture tiles on the hard disk.

4.5.14 Transparency

Virtual texturing works fine with transparency (alpha blending) in principle, but the tile determination method in view space does not. There are several options here:

- Use another tile determination method or an additional tile determination method for transparent geometry.

- Adapt the view-space tile determination to handle transparency, e.g., by using multiple passes.
- If the amount of transparent textures is low and there is commonly free space in the physical texture, then just keep the transparent tiles in the physical texture all the time
- Just do not use virtual texturing for the transparent parts of the virtual world.
- Restrict to usage of alpha testing instead of alpha blending.

The alpha channel necessary for alpha blending also needs consideration. It is not desired to waste a channel for the alpha values for opaque parts of the virtual texture. It is possible to use the same channel for other values for opaque parts of the virtual texture, e.g., as specular highlight. This necessitates the use of two slightly different virtual texturing shaders, one for transparent geometry and one for opaque. Additionally, it could be beneficial to restrict the parts of the virtual texture used for transparent geometry to a subset with borders at power-of-two boundaries to prevent polluting this channel at higher mipmap-levels.

4.5.15 Multitexturing & Multiple Virtual Textures

Section §4.1.5 mentions a simple way to support multiple texture layers in combination with virtual texturing: reuse the same virtual texture layout and just provide additional layers for additional data. Note that multiple surface color layers are not needed in a virtual texturing system because they can be combined. Thus, additional data refers to emissive/bump/specular/etc. layers.

Barrett's implementation supports multiple virtual textures in the same scene without needing more render or tile determination passes [Bar08]. One virtual texture can be used for the terrain and another virtual texture can be used for the remaining objects. However, it is not possible to use hundreds of virtual textures or even one virtual texture per sub-texture (to circumvent the texture atlas requirement for virtual texturing rendering). A multitude of problems occurs when the number of virtual textures grows above a few, so only some examples are mentioned: since the mipmap-chain of virtual textures only goes down to a single tile, there would be artifacts when viewing these textures minified below the tile-size. Additionally, the performance is expected to be severely impacted, since when viewing several of these small virtual textures from a distance it would be necessary to load the smallest tiles from each of these textures, whereas in a normal virtual texturing situation the needed pixels would be combined into far fewer tiles.

The main motivation for using multiple virtual textures is because some parts of the scene might need different layer combinations, but there is still the desire to share common data/layers. Reusing the same tiles from different virtual textures is possible [Bar08].

4.5.16 Texture Reuse

Current scenes used in real-time rendering often reuse the same texture data throughout the scene by using the same texture in multiple places (often blended with different other textures to make them look unique) or using "repeating" as the texture wrap mode. Since these textures only exist as parts of the virtual texture, these techniques need consideration. This section refers to the textures as sub-textures, because these source textures are used in conjunction with other sub-textures to produce the virtual texture with the texture atlas mechanism.

- Non-unique unwrapping:
  Just as with any texture atlas, it is possible to reuse the same sub-textures at multiple places in world space – Mittring calls this "non-unique unwrapping" or "overlapping primitives in the texture space" [MG08]. Drawbacks include that light-map baking can no longer be used and that the reused texture is completely identical unless some additional method to blend details is used.

- Texture repeat wrapping:
  Support of repeated textures is possible in some cases when the texture coordinates prior to adjustment fall within the [0-1] range, i.e., the primitives are aligned with the texture borders. This is actually a case of non-unique unwrapping but the result looks like texture repeating. Real texture repeating (or mirroring and clamping) for texture atlas applications can be simulated by additional fragment shader operations [Cor04] [MG08].

- Replication/Repeating in the virtual texture:
  The same sub-texture can be used at multiple places in the virtual texture (space) [Cor04], which leads to storage waste, but also allows baked light-maps and unique details. This can be used to get a repeated or a replicated sub-texture.

- Replication/Repeating with tile sharing:
  When replicating sub-textures in the virtual texture, it is possible to reduce the storage overhead by sharing identical tiles [Bar08] [Spa10] (similar to the sharing detailed in Section §4.5.15). This works with sub-textures that are aligned with the tile borders in the virtual texture and are a multiple of the tile-size. Identical tiles are marked and handled specially during tile-streaming, so it is not necessary

to store them twice on disk and in the physical texture. This allows for replicated sub-textures to have limited variation, while only the parts with difference are stored. The tile sharing only works at the lowest mipmap-level(s), at higher mipmap-levels the tiles are uniquified by virtue of being a combination of different context tiles and therefore cannot be shared [Spa10].

4.6 Hardware Support

Virtual texturing as described in this chapter already works on hardware supporting GL_EXT_gpu_shader4, but could still benefit substantially from changes made to the GPU hardware or drivers. The following is a non-exhaustive list of possible improvements:

- Fix per-mipmap texture streaming: This is not a suggestion to help with virtual texturing but would rather make full-blown virtual texturing unnecessary in some cases. Mittring claims that per mipmap streaming (see Section §2.5.1) is currently basically impossible (on the PC) [MG08]. While the texture memory savings using per mipmap streaming are not nearly as high as with virtual texturing [MG08], there are cases where it would be a sufficient solution if it worked.

- Larger 2D texture sizes and more texture array layers: As discussed in Section §4.3.2, the size limitations for a single 2D texture pose a serious problem because the required tiles for complex scenes with large viewports might not fit into the largest possible texture. Some consumer graphics cards today ship with 2048MB of graphics memory, which requires at least 8 uncompressed or 32 DXT5 compressed or 64 DXT1 compressed textures of size 8192^2 to be filled. While a 8192^2 texture size should be enough for many or even most (virtual texturing) purposes, we propose an increase to 16384^2 or preferable even higher (it needs 4 DXT1 compressed 32768^2 textures to fill up 2048 MB!) to prevent lagging behind the hardware capabilities. ATI already moved to 16384^2 recently on new hardware, but other vendors need to catch up. The 2048^2 limit on Intel GMA and some other integrated "graphics cards" is so low that it might very well prevent a usable virtual texturing implementation on this hardware (although this hardware often drives low-resolution viewports). The same goes for texture array layers. The limit on NVIDIA hardware of 512 layers prevents using this facility for virtual texturing, at least when using one tile per layer.

- Help with tile determination: The read-back from view-space tile determination is an expensive operation and large read-backs generally do not fit well within a real-time rendering pipeline. Providing mechanisms to help with tile determination

would be a great benefit. One example would be an extension that provides a list of (unique) pixels that have been accessed from a specific texture during a render pass. Enabling this feature on the pagetable texture would provide us with a list of currently visible tiles. As a lesser aid, random access writes could also be of use with tile determination, and they already arrived during the writing of this thesis in the form of EXT_shader_image_load_store. Note that proposing hardware modifications for tile determination is not new, but was first done by [GY98].

- Full virtual texture support: Full virtual texturing support had actually been built into graphics cards from 3Dlabs starting in 1999 [Sem99] and eventually ending with their GPU production halt. Current vendors could re-create that support and build virtual texturing directly into the hardware/driver combination. The application could create a virtual texture with a given size and tile-size. After each frame, the GPU would report a list of missing tiles and use fallback tiles instead of stalling during rendering. The application would stream in these tiles as fast as possible (an additional conservative tile determination system could help here). Pagetable and physical texture handling would be up to the GPU, as would be filtering issues. It is questionable whether the improved performance of such a system would outweigh the loss in flexibility.

Chapter 5

Implementation & Results

As part of this thesis, a virtual texturing library called "LibVT" was designed and implemented. Additionally, a very large scene that represents New York was converted to use virtual texturing and imported into the demo application to evaluate virtual texturing for a real-world use case. Finally, some benchmarks were performed to assess the importance and optimal settings of different parameters and tradeoffs in a virtual texturing system from a performance and visual quality standpoint.

5.1 LibVT

LibVT is a library implementing virtual texturing in the style of [vW09b], similar to Barrett's demo [Bar08] (even using some of Barrett's shader code), although it does not do procedural generation, but tile streaming from disk. LibVT is available as an open-source project at the following URL: `http://www.sf.net/projects/libvt/`

Features include:

- Written in C with C++ being used for data structures and threading.
- Using OpenGL with shaders in GLSL or Cg.
- Designed as a library to allow easy integration into existing rendering engines (e.g., OpenSceneGraph integration has been done).
- Compatible with OpenGL ES 2.0 and support for iOS (iPhone / iPad).
- Tile determination in view space using a read-back:
  - With either analytically calculated mipmap-level or determination by texture sampling.
  - With configurable lower resolution.

  - Rendered either into the back-buffer or into a FBO.
  - Read-back either with `glGetTexImage()` or `glReadPixels()`.
  - Optional buffer compression using OpenCL for faster read-back and combination with other render passes.

- Configurability of physical texture dimension, RAM-cache size, tile border, tile-size, virtual texture size, resident mipmap-levels, cache warming, etc.
- Support for bilinear, trilinear and anisotropic filtering.
- Mipmap-chain length of up to 11, allowing a virtual texture resolution of $256k^2$ with 256^2 pixel tiles.
- Multiple tile decompression libraries: LibPNG, LibJPEG, LibJPEG-Turbo, STBI [Bar], ImageMagick [ima] and CoreGraphics [Com].
- Usage of compressed (JPEG, PNG, etc), uncompressed or DXT1/5 pre-compressed tiles.
- Multithreaded and decoupled tile streaming using boost::thread and with optional real-time DXT compression using FastDXT.
- All texture transfers (read-back, pagetable texture and physical texture) optionally asynchronous using PBOs.
- Either stores fallback entries in the pagetable texture or uses looping in the fragment shader.
- Optional dynamic adjustment of the LoD bias to fit visible tiles into the physical texture.

LibVT also includes a pipeline for generating virtual textures out of individual texture files:

- **generateTextureAtlas**: can generate a large ($128k^2$ at least) texture atlas (and a coordinate offset file) out of individual texture files and can place a subset of the files in a grid e.g. if part of the texture atlas should cover linearly mapped terrain.
- **generateVirtualTextureTiles**: build the virtual texture tiles of all mipmap-levels out of a large picture (single texture or atlas).
- **mergeVirtualTextureTiles**: merge multiple virtual texture tile stores into a single one.

- **convertVirtualTextureTiles**: transcode a virtual texture tile store into another image compression format.
- **offsetObjTexcoords**: applies the coordinate offset file produced by **generateTextureAtlas** to an Alias Wavefront OBJ file so it uses a single (atlas) texture instead of multiple textures.

5.2 New York Scene

Fig. 5.1: A screenshot from the New York scene.

The New York scene [BRO08] is a complex scene with very high texture requirements that has been used as a testbed for our virtual texturing implementation and the content pipeline. Figure 5.1 depicts a screenshot from this scene. The dataset is composed of 16 3DS Max scenes (.3ds). Each sub-scene has between 190k and 1094k triangles. Each sub-scene has between 485 and 2062 textures, most of them consisting of 1024^2 pixels. The fully combined scene has 8 million triangles, 9.4 million vertices and about 19,500 textures. The textures are compressed to about 3.2 GB and would take more than 60 GB uncompressed. The scene is composed of buildings and terrain.

Here is a rough outline of the steps performed to import the dataset into the virtual texturing demo application:

- Convert each of the 16 sub-scenes to OBJ file format and separate the buildings and terrain into individual files (MeshLab [CCR08]).
- Connect the meshes (which were provided chopped in a checkerboard pattern) and remove duplicate vertices.
- Combine the sub-scenes until there are only two files: one for the terrain and one for the buildings.
- Reduce the terrain mesh with an edge collapse algorithm (MeshLab).
- Rotate and move the meshes to the coordinate origin and align them with the word space axis.
- Combine the $\sim$5k terrain textures in their natural order to16 files (ImageMagick).
- Build a $128k^2$ texture atlas out of the $\sim$14k building textures (some have been scaled down) and the 16 pre-combined terrain textures (generateTextureAtlas).
- Build a tiled virtual texture out of the texture atlas (generateVirtualTextureTiles).
- Apply the texture coordinate offsets produced alongside with the texture atlas for the building mesh (offsetObjTexcoords).
- Import the final OBJ files into the demo application by converting to a custom octree format.

5.2.1 Problems

There are problems with the New York dataset that result in very low performance (quality) with virtual texturing. These problems are a direct result of the layout of the individual sub-textures within the texture atlas. Nevertheless these results are interesting from a scientific point of view since they lead to new knowledge about things to avoid when doing virtual texturing. Specifically the texture atlas ignores both critical points that Mittring raises, grouping free space to cover whole tiles and striving for world to texture space similarity [MG08]. Even more importantly, the dataset violates a third point that we discovered to be very significant: varying world-to-texel density ratios. Within most tiles, sub-textures of different world-to-texel density ratios are combined, leading to the fact that multiple mipmap-level versions of the same tiles are necessary and additionally many tiles from high mipmap-levels are requested because they contain at least one sub-texture with a low texel-to-world ratio. Figure 5.2 shows a minified detail of the texture atlas. The source of this problem likely is the origin of the dataset, since aerial photographs provide a different resolution depending on the angle. So,

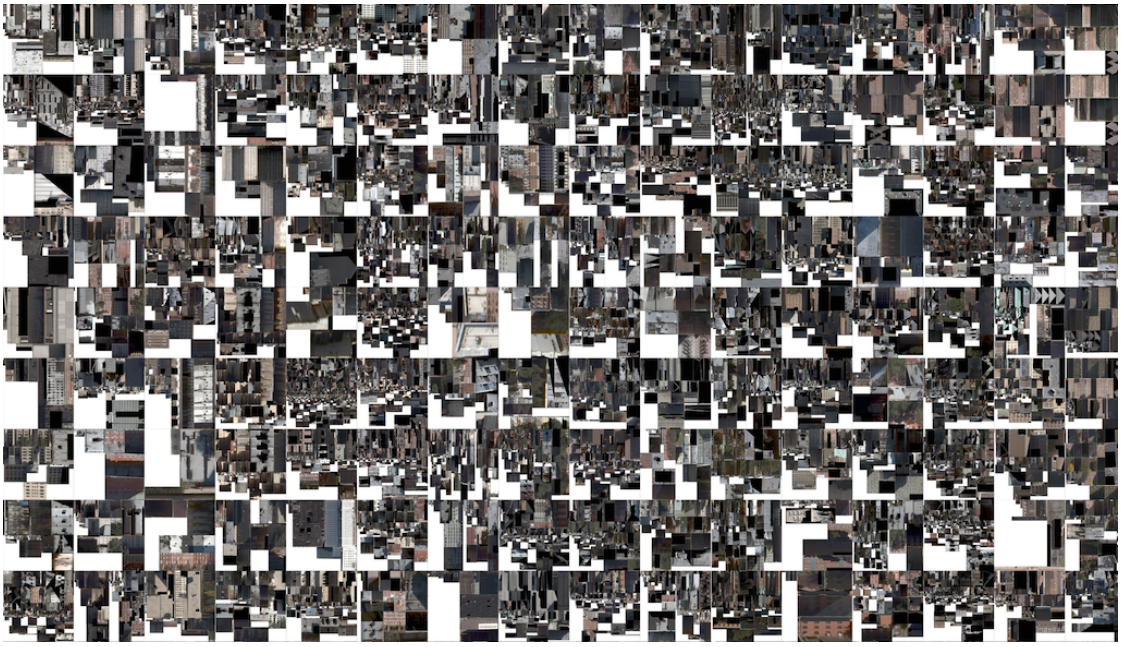

Fig. 5.2: Detail of the New York scene texture atlas that discloses some of the problems.

while virtual texturing does not require a fixed world-to-texel density and different ratios for different larger objects pose absolutely no problem, the frequent changing of this property within most individual tiles aggravates the problem that this dataset requests "too many" tiles for rendering. Concretely, an example flyover with the terrain scene requires 41 tiles on average per frame and a maximum of 68 tiles (256^2 pixel tiles). An example flyover in the New York scene takes 284 tiles on average and a maximum of 802 tiles in the worst frame. The average is nearly 7 times worse and the maximum case is nearly 12 times worse compared to the terrain scene. Fixing all mentioned problems is not (easily) possible for this dataset since the texture data is provided pre-combined into small texture atlas files with a resolution of 1024^2 pixels and it would be necessary to split these textures into their multiple sub-textures. The runtime performance of the dataset would be significantly improved if the individual textures had been provided (even though numbering in the hundred thousands) instead of pre-combined files with large empty space areas.

It may be possible to use non-standard mesh parametrization, e.g., [THC04] to reduce the wasted texture space for virtually textured objects.

5.3 Open Scene Graph Integration / Scanopy & Terapoints

The flexibility of both Open Scene Graph (OSG) and LibVT allows the usage of LibVT and therefore virtual texturing within the OSG rendering framework, without tightly integrating LibVT into OSG and therefore tying it to this specific framework. Using LibVT from an OSG application can be done by attaching a "pre-pass camera" to the scene-graph which renders the geometry with the tile determination shader. The pre-pass camera is attached to an image object which is passed to LibVT in a so called `PostDrawCallback`. The main pass is rendered with the virtual texturing shader. Using a single-pass solution with OpenCL buffer reduction should be even simpler to use, but has not been attempted yet.

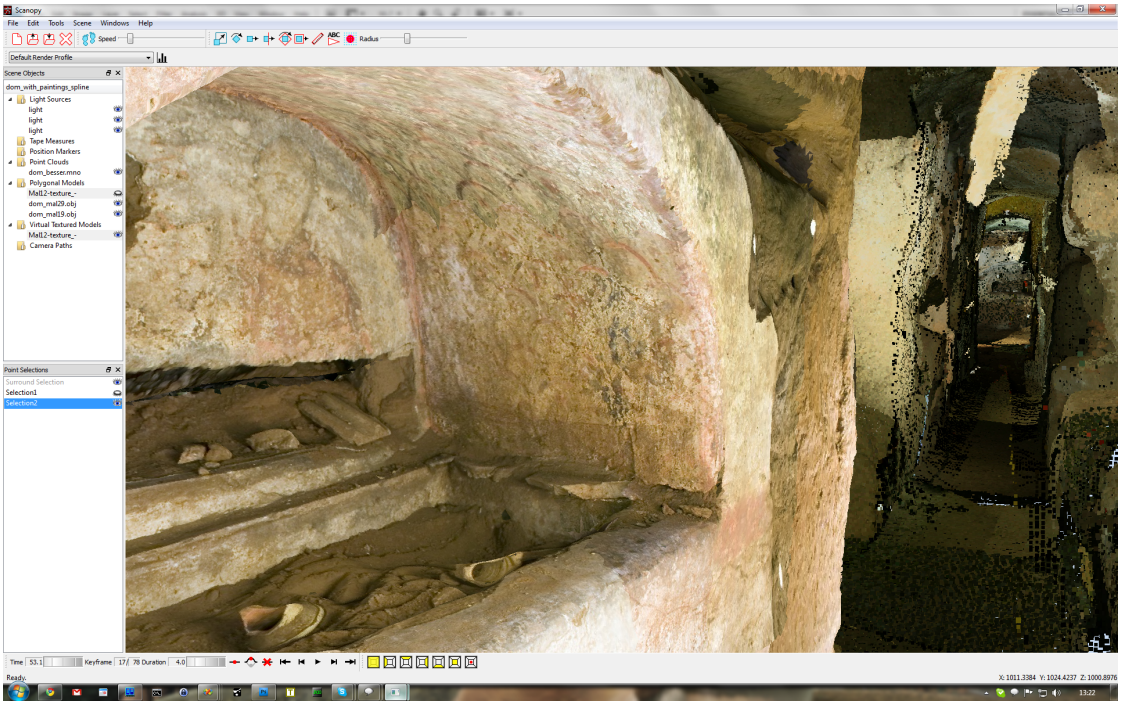

Fig. 5.3: Screenshot of the Scanopy application featuring integration of virtually textured polygonal models with point-rendering.

The Scanopy project [SP08], which has been succeeded by the Terapoints project, aims to provide efficient algorithms for working with 3D laser scan data. They have developed a point-rendering application within the OSG framework which is able to do real-time out-of-core rendering of a dataset of the Domitilla Catacomb in Rome which consists of (currently) 1.9 billion points. The dataset was provided by the FWF START-Project "The Domitilla-Catacomb in Rome. Archaeology, Architecture and Art History

of a Late Roman Cemetery" [oT09]. In addition to the point-data, the dataset features polygonal meshes of the graves with high-resolution textures of the paintings. A single grave can feature more than 20 textures of size 4064 x 2704, and there are 80 graves in total, though not all are fully textured yet. Because of the high texture requirements the graves could not be rendered previously, which is now possible with virtual texturing. The aggregate texture data of the paintings is expected to fill a $128k^2$ virtual texture.

5.4 Results

Section §3.4 identifies performance and correctness as the two major challenges for a virtual texturing implementation, consequently this section is focused on assessing these two attributes. Since most settings only affect either the performance or the quality, the benchmarks include only settings that make a difference for the tested attribute.

5.4.1 Performance

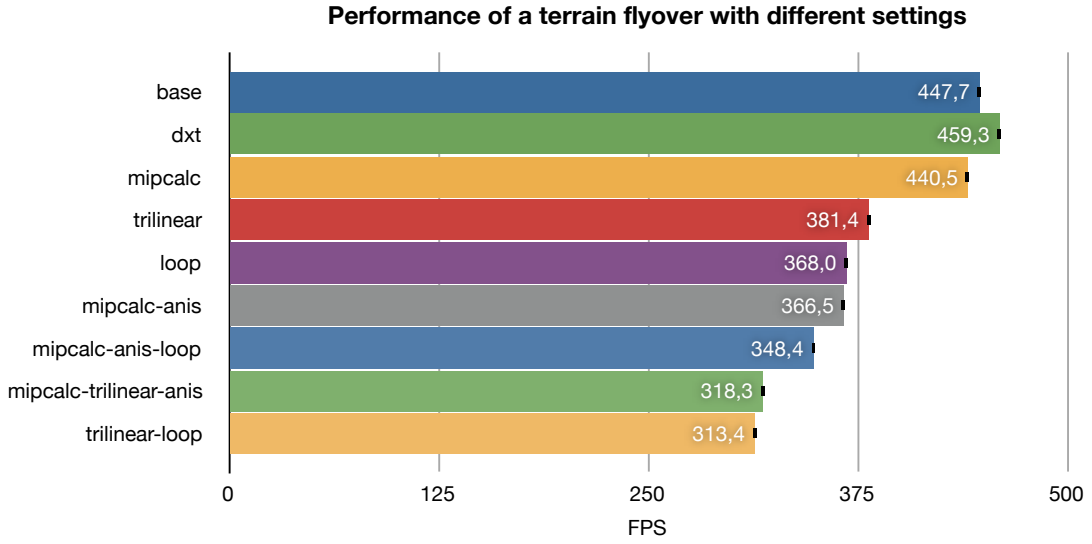

Fig. 5.4: A benchmark comparing the performance of virtual texturing rendering with different settings in the terrain scene with a $32k^2$ virtual texture and a tile-size of 256^2.

Figure 5.4 compares the performance with different settings during a flyover in the terrain scene. "base" denotes a baseline without special settings and uses mipmap-level calculation by fetching from a texture. "mipcalc" does mipmap-level calculation analytically. "loop" performs a loop in the fragment shader and does not use fallback

entries. The results show that features which improve the visual quality like trilinear or anisotropic filtering have a considerable impact on the performance. Similarly, looping in the fragment shader (instead of storing fallback entries in the pagetable) also has a large negative consequence, despite saving CPU time and bandwidth during pagetable texture updates. Turning on real-time DXT compression increases the frame-rate because the implied slower tile streaming system mainly effects quality (and not performance), but the rendering and bandwidth to the GPU is improved. Turning on PBO transfers of the read-back, pagetable updates and physical texture updates did not make a difference in this demo application because there are no other CPU-intensive calculations to be done, but is bound to provide speedups in most real-world applications. For comparison, rendering this terrain scene without virtual texturing (with a $8k^2$ texture, trilinear filtering, 2x anisotropy) achieves an average frame-rate of about 1150 FPS.

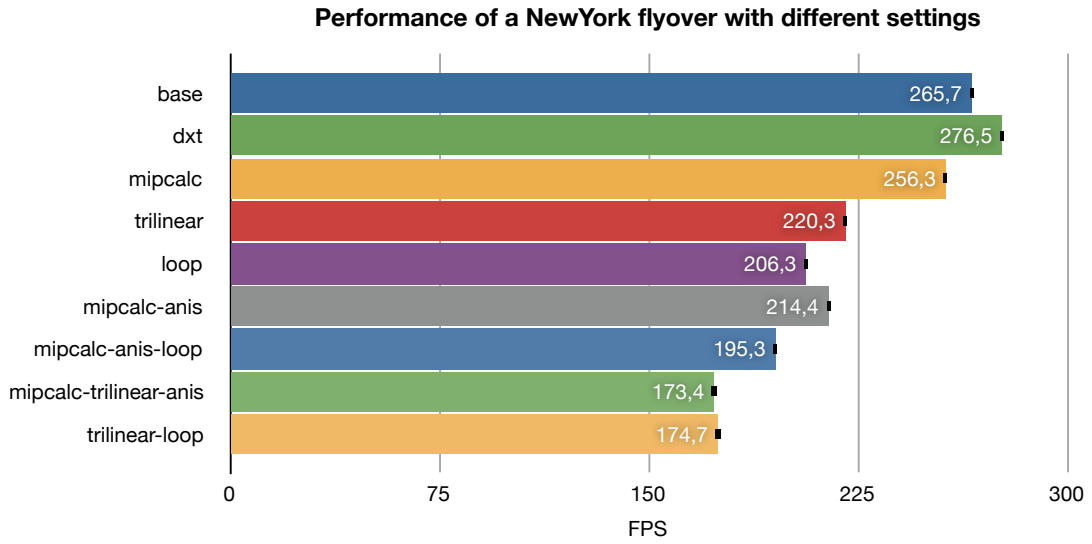

Fig. 5.5: A benchmark comparing the performance of virtual texturing rendering with different settings in the New York scene with a $32k^2$ virtual texture and a tile-size of 256^2.

Figure 5.5 performs the same tests on the New York scene, with similar results. The performance hit from looping in the fragment shader is slightly increased because in the New York scene more needed tiles are missing and therefore more looping occurs. Rendering the New York scene without virtual texturing (with a $8k^2$ texture, trilinear filtering, 2x anisotropy) achieves an average frame-rate of about 760 FPS.

Figure 5.6 compares the render performance in the terrain scene with different tile determination modes: reading back the visible tile buffer in the same frame, reading back the buffer in the next frame and usage of the OpenCL buffer reduction kernel with two render passes and with a single render pass. All tests are performed with a full, half and quarter-resolution visible tile buffer. In the OpenCL single-pass solution, the visible

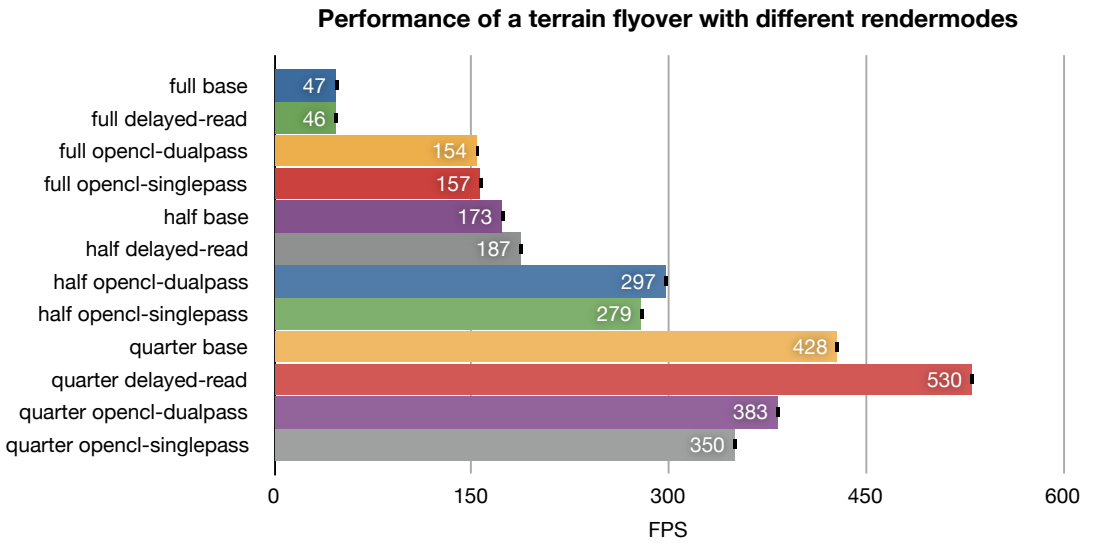

Fig. 5.6: A benchmark comparing the performance of virtual texturing rendering with different methods in the terrain scene with a $32k^2$ virtual texture and a tile-size of 256^2.

tile buffer is rendered in full resolution and just processed in a lower resolution, all other modes render the visible tile information in a lower resolution. The results show that the modes using the OpenCL buffer reduction are three times as fast in a full-resolution setting, nearly two times faster in half resolution, but slower in quarter resolution. Reading back the buffer in the next frame is noticeably faster in half resolution and is especially fast in quarter resolution, being the fastest solution with nearly a 25 % gap. Surprisingly the OpenCL solution is faster in this case when integrated into a two-pass solution than in a single pass. The reason for this is that this scene has so few polygons (between 5k and 50k triangles per frame) that the overhead of the additional render pass is lower than the (fill-rate) hit from performing the visible tile calculation at full resolution.

Figure 5.7 performs the same tests as Figure 5.6, but on the New York scene, which has much higher polygonal requirements (between 20k and 800k triangles per frame). Note that this test does not use the $128k^2$ virtual texture, as other tests with the New York scene do, but rather a $32k^2$ virtual texture. This is because our OpenCL solution is not yet compatible with long mipmap chains. As expected, the single-pass OpenCL solution is much faster in this scene, because it avoids having to render a large number of triangles twice per frame. It outperforms the normal render-mode by nearly a factor of two at half resolution and is still slightly faster at quarter resolution. However, the fastest result overall is the non-OpenCL solution with quarter resolution and delayed read-back. When discarding the quarter-resolution results because of its lowered quality, the OpenCL solution is the best option from a performance standpoint. There are also other factors that are in support of the OpenCL solution:

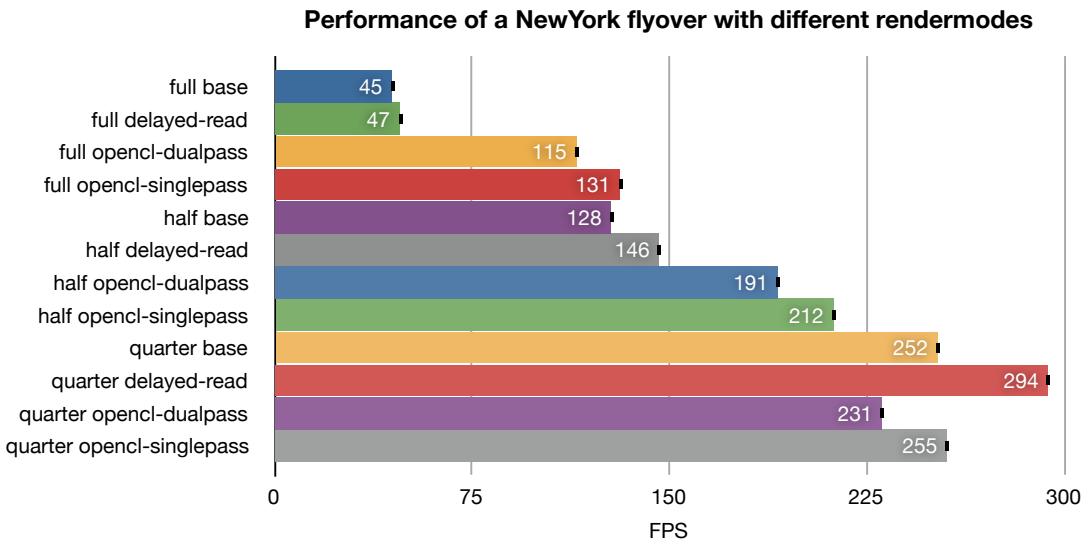

Fig. 5.7: A benchmark comparing the performance of virtual texturing rendering with different methods in the New York scene with a $32k^2$ virtual texture and a tile-size of 256^2.

- The current OpenCL drivers are still immature and are expected to get faster.
- The tests were performed with a graphics card that is of the first generation of GPGPU capable cards. Newer graphics cards likely provide disproportionally faster OpenCL support.
- Our OpenCL solution has not received extensive tuning, and is especially immature in comparison to our two-pass setup, which has been tested for months. Areas for speedups within the shader itself are sharing of results that are currently calculated twice for the two render target outputs, as well as usage of the new OpenGL extension EXT_shader_image_load_store to eliminate the first kernel.

As mentioned above the New York scene flyover yields about 760 FPS and the terrain scene about 1150 FPS without virtual texturing (with a significantly lowered texture resolution of $8k^2$). This shows that virtual texturing has a severe performance hit ("normal" texturing is about 2.1 - 2.5x faster than the fastest virtual texturing result), but there currently are no competing methods that would allow rendering these scene with a $128k^2$ texture. However, since the "normal" texturing is a very cheap render-mode that does not even use shaders in our test, the high FPS difference is a bit misleading. Converting these numbers to milliseconds per frame shows that the overhead of virtual texturing compared to this simplest rendermode is only between 1 millisecond (terrain scene) and 2 milliseconds (New York scene) per frame. Additionally, virtual texturing can replace current setups that use blending to simulate unique textures, which results in

costly overdraw [MG08] and likely are not much faster, if at all. In any case, rendering the semi-complex New York scene with nearly 300 frames per second shows that virtual texturing is a valid technique from a performance point of view, even for outdated (graphics) hardware.

5.4.2 Quality

To evaluate the visual quality of our virtual texturing implementation, we use a simple metric, the average number of pixels that have to fallback to a lower resolution tile because their "native" tile is not available.

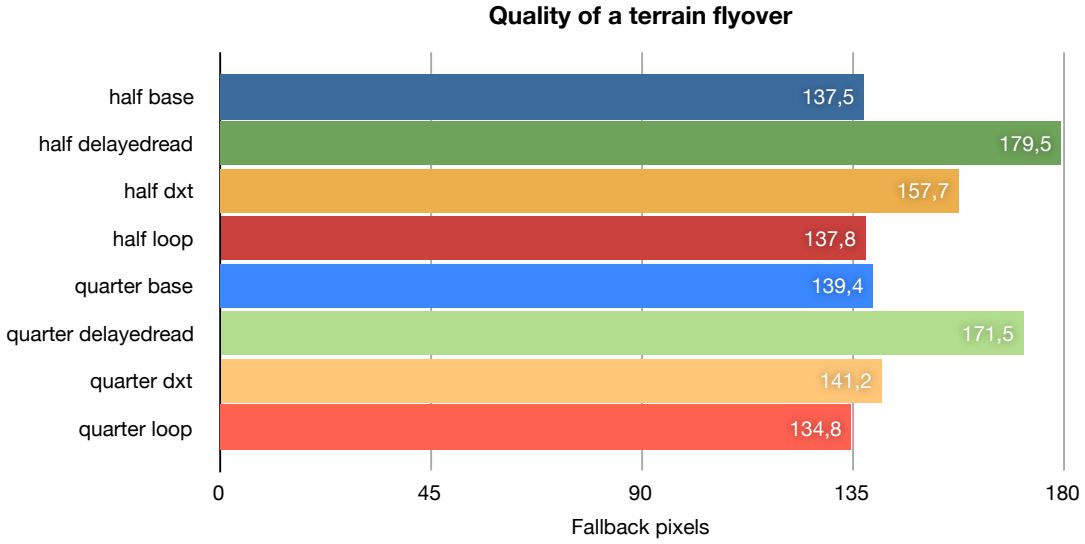

Fig. 5.8: Quality of the terrain scene with different settings.

Figure 5.8 features the quality for 4 render-modes: normal virtual texturing, with a delayed read-back, with DXT compression and with looping in the fragment shader. Because of the randomness of hard disk accesses the results had a very high variation, therefore we decided to include only the best result from each mode, instead of the average of 5 runs. The results show that looping in the fragment shader has basically no influence on the quality and the influence of DXT compression (which slows the tile streaming) is marginal. The impact of delaying the read-back by one frame is noticeable, but small. Because of the rendering speedup and the small influence on visual quality this option is beneficial. The results also show that (with a tile-size of 256^2 pixels) there is practically no difference in visual quality when rendering the visible tile information at quarter resolution. As mentioned in Section §4.2.1 this is especially true for terrain scenes, but not for other scenes.

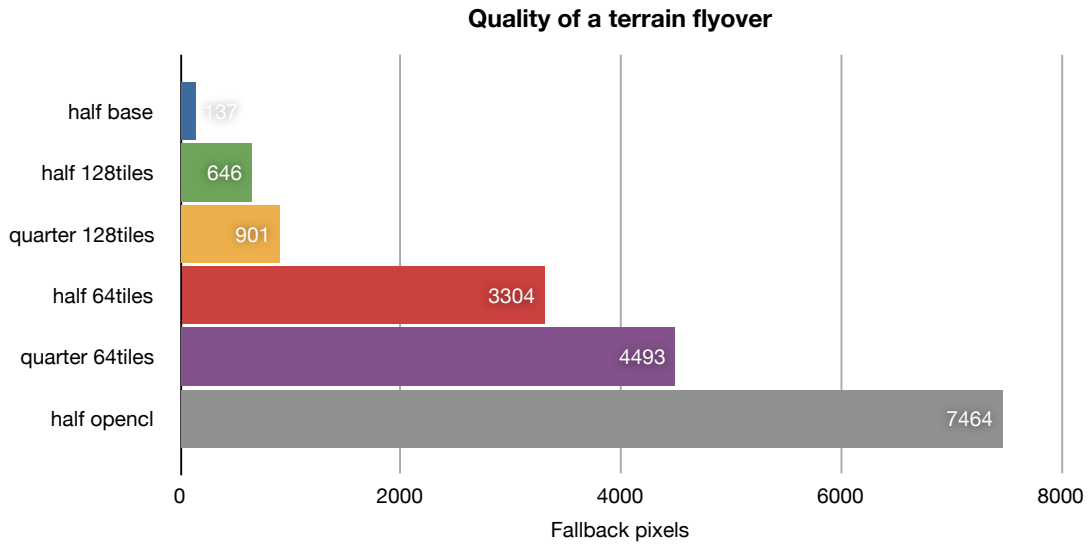

Fig. 5.9: Quality of the terrain scene with different settings.

Figure 5.9 features options that have a much larger impact on the visual quality. The figure shows that smaller tile-sizes lead to significantly worse image quality. Additionally, the difference of doing the visible tile determination in quarter resolution becomes much more pronounced. The negative influence of small tile-sizes could be lessened by tuning the tile streaming system for (many) small tiles, i.e., combining multiple tiles into fewer files and loading multiple tiles at the same time. The option which has the largest negative effect on quality is the OpenCL buffer reduction. Because the buffer reduction looses the "priority-information", i.e., how many pixels reference a requested tile, tiles are loaded in random order instead of sorted by priority. This shows that page-loading sorted by priority has a vast effect on visual quality, We expect that an OpenCL buffer reduction algorithm that retains this priority information is comparable in quality to a read-back solution.

To put these results into perspective, we have to consider that this test was run at 1280 x 960, i.e., with 1,228,800 pixels. Even with OpenCL / without tile priority, only every 164th pixel falls back to a lower resolution on average. This is about 0.6 percent. With any of the 4 options in Figure 5.8, only 0.01 percent of the pixels fall back to a lower resolution on average. Since the fallback to lower resolution tiles is the main cause of artifacts in a virtual texturing application, we can conclude that virtual texturing is an acceptable solution with respect to visual quality.

The New York scene has a problematic texture atlas layout (see Section §5.2.1), which also results in degraded quality. The average number of pixels with fallbacks in the "quarter base" setting is 7391, which is comparable to the terrain scene without tile

priority. This confirms that the texture atlas layout affects quality more than most other implementation details. Nevertheless, just as with the OpenCL solution, this still is only a fallback for 0.6 percent of the pixels. Giving the application two seconds time at the start to pre-cache some textures lowers the average number of fallbacks to about 5000.

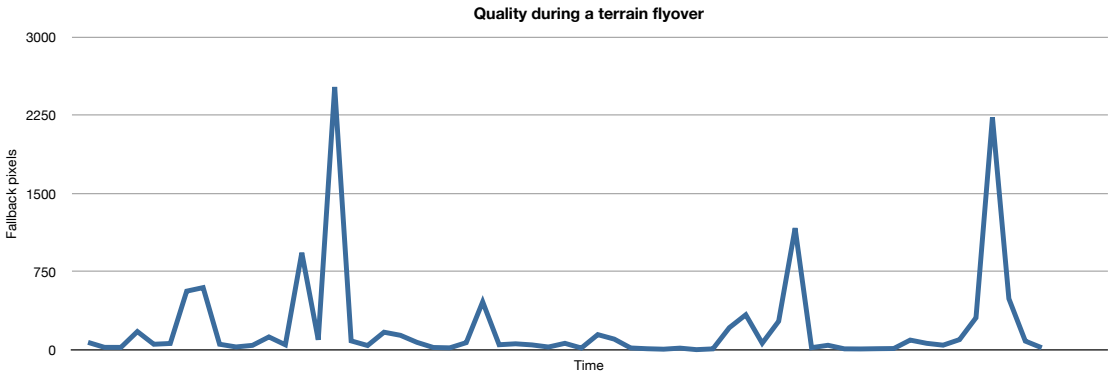

Fig. 5.10: Quality over time (terrain).

Figure 5.10 shows the number of fallback pixels plotted over time, calculated from the average of 5 test-runs with the base settings. We can see that there are very few fallback pixels most of the time, but there are some spikes, corresponding to sudden rotations or movements that the tile determination system could not anticipate.

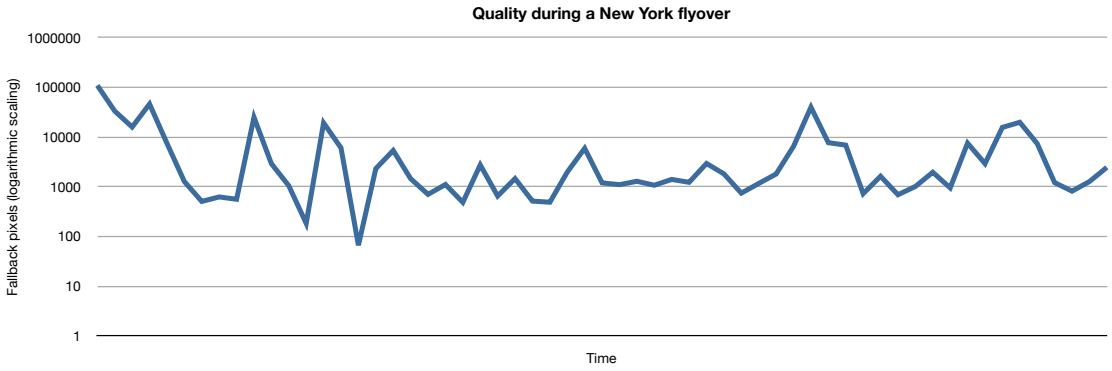

Fig. 5.11: Quality over time (NewYork).

Figure 5.11 also shows the number of fallback pixels plotted over time, but for the NewYork scene and with logarithmic scaling.

Chapter 6

Conclusion

Chapter §4 provided a comprehensive overview on the field of virtual texturing. Section §4.1 started by examining the virtual texture itself, its assembly and storage, its maximum size ($256k^2$ is easily attainable) and the issue of tile-size. It was shown that data sizes below 64 kilobyte are not read efficiently by current hard disk drives, leading to the fact that the method of storing each tile into a single file, while already suboptimal for 256^2 pixel tiles, becomes increasingly inefficient for smaller tile-sizes.

Section §4.2 concerned itself with the determination of the visible tiles. We examined the "exact tile determination in view space" in detail and determined that this method is only feasible performance wise (on the PC) if either done at a lower resolution or when compressing the resulting data using a GPGPU program before reading it back to the CPU. We presented such a buffer reduction solution using OpenCL, which is based on a similar CUDA solution presented in [HPLdW10], and assessed its performance as sufficient even on first-generation OpenCL hardware. The "exact tile determination in view space" was found to be especially valuable in scenarios where requesting many tiles is not acceptable because the scene already borders on having tile requirements that exhaust the physical texture size and tile streaming performance. In any scenario it would be beneficial to have a system that determines tiles that will become visible in the future, but it is not known if this requirement is best served by having an additional conservative tile determination system, or by some metrics that supplement (e.g., loading adjacent tiles) or methods that expand on exact view space tile determination (e.g., increasing the FoV or extrapolating the camera position). Streaming such predicted tiles is an area for future research.

Section §4.3.1 discussed the tile streaming system and evaluated common libraries that can be used for the decompression step within this system. It was determined that PNG is an unacceptable format for a virtual texturing implementation, but JPEG is acceptable, especially when using a tuned decompression library like libjpeg-turbo, which proved sufficient in our implementation. DXT pre-compressed tiles were shown to yield performance of only about a third of libjpeg-turbo and therefore only prefer-

able under very specific circumstances. Several CPU-based real-time DXT compression libraries were evaluated, and only FastDXT was found to have a good enough performance. An area for later experiments within tile streaming is the usage of newer compression formats with higher compression rates, as the hard disk throughput was the limiting factor with a fast JPEG decompression. Section §4.3.2 presented several methods to work around the size limitation for the physical texture. We feel that while this is an important point for hardware with a very low texture size limit (Intel), the limit on ATI or NVIDIA cards should be high enough even for high definition viewports, unless the texture atlas layout is sub-optimal.

Section §4.4 dealt with the virtual texturing shader that makes virtual texturing possible. We do not see much area for improvement in the virtual texturing shader – methods like using a floating point pagetable texture might provide constant time speedups, but are likely of marginal importance compared with other factors in a virtual texturing system.

Section §4.5 handled a plethora of other topics surrounding virtual texturing. Texture filtering was discussed in detail and could be called a solved issue, although trilinear and anisotropic filtering require a costly gradient correction which severely limits the performance of the virtual texturing shader. One discovery here that has not been noted in other resources is that it is sufficient to have the border on two instead of all four sides of the tiles, effectively halving the waste.

Combined with the existence of fallbacks to low-resolution tiles, the issue of the "LoD Pop-in" is the most severe visual artifact in a virtual texturing system. Solving this issue can be done by gradually blending-in details or by loading unrequested ancestor pages prior to the actually needed pages. Another topic that was referenced in this section is the issue of the virtual texture atlas layout which was determined to be of utmost importance. The New York scene as an example of a bad virtual texture layout requires more than an order of magnitude more tiles for display than a terrain scene. To combat artifacts that result from the texture atlas technique we proposed the idea to prefer aligning sub-textures with tile borders and construct the borders of these tiles by clamping the sub-texture instead of from the adjacent sub-texture.

The topics of "tile importance" and "tile request substitution" are (semi-)advanced virtual texture topics that have only theoretically and briefly been touched and should be focused in future work. We implemented just the simplest tile importance metric, namely sorting tile requests by their pixel count, and this has proven to be of very high importance in the quality benchmarks. We presented some novel ideas like "screen space tile prediction" which should be evaluated in quantitative tests. The remainder of this section dealt with other issues like decals, transparency and texture reuse.

Chapter §5 presented our implementation (LibVT), its applications (New York scene, OSG Terapoints) and the results derived from it. The results section (§5.4) deals with

the performance and the quality of virtual texturing. The performance results show that when doing tile determination at full or half resolution, the fastest option is a single-pass solution with the presented OpenCL buffer reduction method. In lower tile determination resolutions, the fastest option is a dual-pass solution with delaying the read-back until the next frame. Of course the actual performance is subject to the specific application and their other CPU and GPU requirements, but the results show that at full or half tile determination resolution, the OpenCL solution is very performance competitive and at lower resolutions a dual-pass solution is also viable. Doing tests on newer hardware (which should provide much better OpenCL performance) and tests with an OpenCL solution that retains tile importance information should prove worthwhile in the future. An interesting possibility to explore is the exploitation of the functionality offered by the new OpenGL extension EXT_shader_image_load_store, which should eliminate the need for the first OpenCL kernel, saving time and memory. The quality results show that the base virtual texturing implementation with 256^2 pixel tiles provides very good quality with only 0.01 percent of the pixels being subject to a fallback on average. Settings like delaying the read-back and DXT compression made only a marginal difference. We determined that lower tile-sizes made a much larger difference, increasing the fallbacks by a factor of about five (128^2) respectively about 25 (64^2). We feel that only a small part of this deterioration is caused by smaller pages being easier to "miss" at low-resolution tile determination, and most of it is caused by the inefficiency of streaming small tiles when they are stored in a file per tile. Combining the tiles into a single file and having an intelligent tile streaming scheduler that streams multiple tiles per request is an area for future work. The worst result quality wise (increase of fallbacks by a factor of 50) was attained when using our OpenCL buffer reduction, but only because it does not retain tile importance information yet. Nevertheless, this still only corresponds to an average fallback of 0.6 percent, which we still deem acceptable. We expect the quality results to be worse by an order of magnitude for the New York scene because of its aforementioned problems. Quality benchmarks on complex scenes remain an interesting topic for future research.

Because there are no competing solutions to virtual texturing for real-time display of arbitrary scenes with vast texture requirements, we could not compare virtual texturing to other methods, but had to evaluate it on its own. We feel that the performance and quality results confirm that virtual texturing is a good method for real-time rendering of scenes with so high texture requirements that they could not previously be displayed. Whether virtual texturing should be used depends on the particular application. In computer games, these large scenes do not yet exist, because during the scene creation methods like texture repeating are used to minimize the texture requirements. Virtual texturing would provide the possibility to switch to scenes that have real unique details and therefore provide higher visual fidelity, at the expense of a runtime performance hit as well as the cost of developing new content tools and changing the

art pipeline. For scientific and industrial applications these scenes often already exist and can now be displayed entirely instead of only partly or only with scaled down textures – as depicted in Figure 6.1. However, as we have seen with the New York scene, these scenes, which are often automatically generated (e.g., from aerial imaging), can have properties that can make them perform suboptimal in a virtual texturing application. Therefore we see the automatic generation of optimized texture atlases for virtual texturing as the most important area for future research. The results also showed that the New York scene exhibited very distinct properties from the terrain scene. The conclusion to draw here is that any future research on virtual texturing should always be done on scenes with a virtual texture that consists of a texture atlas, and not on simple terrain scenes with a single linearly mapped texture. Ideas, like loading ancestor tiles as proposed in [CESL10], that provide benefits on terrain scenes might not help with more complex scenes at all, and terrain scenes could be handled with clipmapping alone.

Fig. 6.1: Virtual texturing provides independence from graphics memory constraints.

Going back to the aim and question of this thesis, based on the results and data provided in this thesis we assert that virtual texturing definitely is a feasible technique for real-time rendering scenes with out-of-core texture data sets. We conclude this thesis by referring to Figure 6.1, which suggests that virtual texturing finally brings independence from graphics memory constraints.

Acknowledgements

We thank Sean Barrett for providing the basis to this thesis with his work and for his kind help and detailed explanations. We also thank Matthäus G. Chajdas and Charles-Frederik Hollemeersch for providing pre-release chapters of their related book chapters so they could be referenced in this thesis. Finally, thanks go to Michael Wimmer and Anna Wirnsberger for correcting this thesis.

List of Figures

List of Programs

Bibliography

[Air90] John Milligan Airey. *Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations*. PhD thesis, The University of North Carolina at Chapel Hill, 1990. Director-Brooks, Jr., Frederick P.

[BAC96] A Beers, M Agrawala, and N Chaddha. Rendering from compressed textures. *Proceedings of the 23rd . . .* , Jan 1996.

[Bar] Sean Barrett. STBI. `http://nothings.org/stb_image.c`.

[Bar08] Sean Barrett. Sparse virtual textures. `http://silverspaceship.com/src/svt/`, 2008.

[Bla09] Brad Blanchard. Brad blanchard - online portfolio: Virtual texturing. `http://linedef.com/personal/demos/?p=virtual-texturing`, 2009.

[Blo08] Charles Bloom. DXTC part 1, DXTC part 2, DXTC part 3, DXTC part 4, DXTC summary. `http://cbloomrants.blogspot.com/2008/12/12-08-08-dxtc-summary.html`, 2008.

[BOA09] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 159–166, New York, NY, USA, 2009. ACM.

[BRO08] 3D CAD BROWSER. 3D CAD Browser - New York City (USA). `http://www.3dcadbrowser.com/preview.aspx?ModelCode=16404`, 2008. [Online; accessed 20-May-2010; This is part 1 of 16, other parts can be accessed by increasing the ModelCode in the URL].

[Car05] John Carmack. John carmack keynote at quakecon 2005. `http://us.generation-nt.com/john-carmack-quakecon-2005-`

keynote-complete-transcript-help-29744482.html, 2005.

[Car07] John Carmack. John carmack id tech 5 demo at quakecon 2007. http://www.youtube.com/watch?v=bnkaxuIzGO4, 2007.

[Cat74] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, Department of Compuer Science, University of Utah, December 1974.

[CC05] Richard Connery and John Carmack. MegaTexture in Quake Wars. http://www.beyond3d.com/content/articles/95/3, 2005.

[CCR08] Paolo Cignoni, Massimiliano Corsini, and Guido Ranzuglia. MeshLab: an open-source 3D mesh processing system, April 2008.

[CE98] D Cline and PK Egbert. Interactive display of very large textures. *Proceedings of the conference on Visualization'98*, page 350, 1998.

[CESL+03] YJ Chiang, J El-Sana, P Lindstrom, R Pajarola, and CT Silva. Out-of-core algorithms for scientific visualization and computer graphics. *IEEE Visualization*, 2003.

[CESL10] Matthäus G. Chajdas, Christian Eisenacher, Marc Stamminger, and Sylvain Lefebvre. Virtual texture mapping 101. In W. Engel, editor, *GPU Pro: Advanced Rendering Techniques*. A K Peters, 2010.

[CNF+07] Roger Crawfis, Eric Noble, Michael Ford, Frederic Kuck, and Eric Wagner. Clipmapping on the GPU. Technical report, Ohio State University, 2007.

[Com] Apple Computer. Quartz 2D / Core Graphics. http://developer.apple.com/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/.

[Cor04] NVIDIA Corporation. Texture atlas whitepaper - improve batching using texture atlases. http://download.nvidia.com/developer/NVTextureSuite/Atlas_Tools/Texture_Atlas_Whitepaper.pdf, 2004.

[Cor06] Microsoft Corporation. The windows vista developer story: Directx. http://www.microsoft.com/indonesia/msdn/wvddirectx.aspx, 2006.

[Cor07] NVIDIA Corporation. Clipmaps whitepaper. http://developer.download.nvidia.com/SDK/10/direct3d/Source/Clipmaps/doc/Clipmaps.pdf, 2007.

[Cor10] NVIDIA Corporation. Real-time DXT compression - nvidia-texture-tools. http://code.google.com/p/nvidia-texture-tools/wiki/RealTimeDXTCompression, 2010.

[Cos94] Michael Cosman. Global terrain texture: Lowering the cost. In Eric G. Monroe, editor, *Proceedings of 1994 IMAGE VII Conference*, pages 53–64. The IMAGE Society., 1994.

[DC06] Cain Dornan and John Carmack. MegaTexture Q&A. http://www.team5150.com/~andrew/carmack/johnc_interview_2006_MegaTexture_QandA.html, 2006.

[Din09] Ramazan Dinçer. CUDA JPEG decoder. http://sourceforge.net/projects/cudajpegdecoder/, 2009.

[Dor] Jason Dorie. Imagelib. http://www.jasondorie.com/ImageLib.zip.

[Duf06] Robert Duffy. id Software's programming director and lead designer explain why rage will kick ass. http://www.maximumpc.com/article/features/id_softwares_programming_director_and_lead_designer_explain_why_rage_will_kick_ass, 2006.

[EC06] Anton Ephanov and Chris Coleman. Virtual texture: A large area raster resource for the GPU. In *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC) 2006*, 2006.

[GM05] E Gobbetti and F Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM Transactions on Graphics (TOG)*, 24(3):878–885, 2005.

[Gro] Independent JPEG Group. Libjpeg. http://www.ijg.org/.

[Gro09] Khronos Group. The OpenGL graphics system: A specification. http://www.opengl.org/registry/doc/glspec31.20090528.pdf, 2009.

[GY98] ME Goss and K Yuasa. Texture tile visibility determination for dynamic texture loading. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 1998.

[Hec86] PS Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, 1986.

[HPLdW10] Charles-Frederik Hollemeersch, Bart Pieters, Peter Lambert, and Rik Van de Walle. Accelerating virtual texturing using CUDA. In W. Engel, editor, *GPU Pro: Advanced Rendering Techniques*. A K Peters, 2010.

[Hüt98] T Hüttner. High resolution textures. *Late Breaking Hot Topics, IEEE Visualization 98 CD-ROM Proc*, 1998.

[ima] ImageMagick. `http://www.imagemagick.org/`.

[iS05] id Software. id Software storms the front lines with enemy territory: Quake wars. `http://www.splashdamage.com/content/id-software-storms-front-lines-enemy-territory-quake-wars`, 2005.

[KE02] M. Kraus and T. Ertl. Adaptive Texture Maps. In *Proc. SIGGRAPH/EG Graphics Hardware Workshop '02*, pages 7–15, 2002.

[LD07] S Lefebvre and C Dachsbacher. Tiletrees. *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, page 31, 2007.

[LDN04] Sylvain Lefebvre, Jerome Darbon, and Fabrice Neyret. Unified texture management for arbitrary meshes. Technical Report RR5210-, INRIA, may 2004.

[liba] Libjpeg-turbo. `http://libjpeg-turbo.virtualgl.org/`.

[libb] Libpng. `http://www.libpng.org/`.

[LK06] Bill Licea-Kane. The OpenGL pipeline newsletter - volume 001 - new texture functions with awkward names to avoid ugly artifacts. `http://www.opengl.org/pipeline/article/vol001_5/`, 2006.

[LN03] S Lefebvre and F Neyret. Pattern based procedural textures. *Proceedings of the 2003 symposium on Interactive ...*, Jan 2003.

[LSO07] Aaron E. Lefohn, Shubhabrata Sengupta, and John D. Owens. Resolution matched shadow maps. *ACM Transactions on Graphics*, 26(4):20:1–20:17, October 2007.

[Mal00] Shahzad Malik. Dynamic level of detail representation of interactive 3d worlds. `http://www.cs.toronto.edu/~smalik/downloads/paper_495.pdf`, 2000.

[MG08] Martin Mittring and Crytek GmbH. Advanced virtual texture topics. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 23–51, New York, NY, USA, 2008. ACM.

[Neu10] Andreas Neu. Virtual texturing. *CoRR*, abs/1005.3163, 2010.

[Oli08] Jon Olick. Next generation parallelism in games. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–89, New York, NY, USA, 2008. ACM.

[oT09] Vienna University of Technology. Scanopy project. `http://www.cg.tuwien.ac.at/research/projects/Scanopy/`, 2009.

[Ren] Luc Renambot. Fastdxt. `http://www.evl.uic.edu/cavern/fastdxt/`.

[SD03] M Stefan and H Deussen. Wang tiles for image and texture generation. *ACM Transactions on Graphics*, Jan 2003.

[Sem99] 3Dlabs Semiconductor. Virtual textures - texture management in silicon. `http://www.graphicshardware.org/previous/www_1999/presentations/v-textures.pdf`, 1999.

[SLO06] Shubhabrata Sengupta, Aaron E. Lefohn, and John D. Owens. A work-efficient step-efficient prefix sum algorithm. In *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, pages D–26–27, May 2006.

[SLT+07] Antonio Seoane, Rubén López, Javier Taibo, Alberto Jaspe, and Luis Hernández. Hardware-independent clipmapping. In *WSCG 2007, The 15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2007*, pages 177–183, 2007.

[SP08] FWF START-Project. FWF START-Project 'the domitilla-catacomb in rome. archaeology, architecture and art history of a late roman cemetery'. `http://www.oeaw.ac.at/antike/institut/arbeitsgruppen/christen/domitilla.html`, 2008.

[Spa10] Sparse virtual texturing discussion forum. `http://mollyrocket.com/forums/viewforum.php?f=21`, 2008 - 2010.

[squ] squish. `http://code.google.com/p/libsquish/`.

[Stu08] Making Art Studios. Making art studios - lightning engine - virtual texturing. http://blog.makingartstudios.com/?tag=virtual_textures, 2008.

[SWP10] Daniel Scherzer, Michael Wimmer, and Werner Purgathofer. A survey of real-time hard shadow mapping methods. In *State of the Art Reports Eurographics*, May 2010.

[THC04] M Tarini, K Hormann, and P Cignoni. Polycube-maps. *ACM SIGGRAPH 2004 . . .*, Jan 2004.

[TMJ98] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158, New York, NY, USA, 1998. ACM.

[TSH09] Javier Taibo, Alberto Seoane, and Luis A. Hernández. Dynamic virtual textures. In *Journal of WSCG, Volume 17, Number 1, 2009*, 2009.

[Uli] Peter Uličiansky. Extreme DXT compression. http://www.cauldron.sk/files/extreme_dxt_compression.pdf.

[vR09] Sander van Rossen. Sander's blog: virtual texture. http://sandervanrossen.blogspot.com/search/label/virtual%20texture, 2009.

[vW06a] J.M.P. van Waveren. Real-time DXT compression. http://www.intel.com/cd/ids/developer/asmo-na/eng/324337.htm, 2006.

[vW06b] J.M.P. van Waveren. Real-time texture streaming & decompression. http://software.intel.com/file/17248/, 2006.

[vW09a] J.M.P. van Waveren. Geospatial texture streaming from slow storage devices. http://software.intel.com/en-us/articles/geospatial-texture-streaming-from-slow-storage-devices/, 2009.

[vW09b] J.M.P. van Waveren. id tech 5 challenges - from texture virtualization to massive parallelization. http://s09.idav.ucdavis.edu/talks/05-JP_id_Tech_5_Challenges.pdf, 2009.

[vWC07] J.M.P. van Waveren and Ignacio Castaño. Real-time YCoCg-DXT compression. http://developer.nvidia.com/object/real-time-ycocg-dxt-compression.html, 2007.

[vWC08] J.M.P. van Waveren and Ignacio Castaño. Real-time normal map DXT compression. http://developer.nvidia.com/object/real-time-normal-map-dxt-compression.html, 2008.

[WBS+04] Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, Student Member, Eero P. Simoncelli, and Senior Member. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13:600–612, 2004.

[Wei04] Li-Yi Wei. Tile-based texture mapping on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches*, pages 55–63, New York, NY, USA, 2004. ACM.

[Wik10a] Wikipedia. S3 virge - wikipedia. http://en.wikipedia.org/w/index.php?title=S3_ViRGE&oldid=352517099, 2010. [Online; accessed 5-May-2010].

[Wik10b] Wikipedia. Ultima underworld: The stygian abyss - wikipedia. http://en.wikipedia.org/w/index.php?title=Ultima_Underworld:_The_Stygian_Abyss&oldid=350829492#Development, 2010. [Online; accessed 5-May-2010].

[WLKT09] Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report, EG-STAR*. Eurographics Association, 2009.