Clipmap-based Terrain Data Synthesis

Malte Clasen*
Zuse Institute Berlin

Hans-Christian Hege†
Zuse Institute Berlin

Abstract

Satellite images of the earth are currently commercially available in resolutions from 14.25m (Landsat 7) up to 0.61m (QuickBird). Visualizing large terrains or whole planets at these resolutions requires highly efficient level-of-detail techniques to reduce the amount of information to the capabilities of the display device. The same applies to simulations that drive a visualization pipeline: Although the data could be computed at very high resolutions, it is often not desirable to generate more information than the visualization part can display.

We describe a conceptually simple terrain rendering pipeline that handles on-the-fly data decompression and synthesis in a unified process. Possible data sources range from static satellite imagery over per-texel-processing such as image blending routines to light-weight simulations and synthesizers such as noise and filter based texture generators. These sources are evaluated in multiple threads and the generated data is fed to a geometry clipmap backend that enables high throughput and requires no preprocessing such as tesselation.

1 Introduction

Terrain rendering applications can be found in many different contexts from cartography over landscape planning to virtual outdoor environments. Satellite images at resolutions below 1m and aerial photography data exceed the resolution of the display devices by many orders of magnitude for even moderate sized terrain areas. Efficient level-of-detail algorithms are required to simplify the data sets before rendering without introducing errors. Data should only be processed or synthesized only if it contributes to the final image. Clipmaps were first introduced by Tanner et al. in 1998 [TMJ98] to solve this for stored images. They consist of a stack of images similar to a mipmap. However, whereas each mipmap level covers the whole texture with images of increasing size, the clipmap uses fixed size levels that cover a decreasing area around an arbitrary focus point. Therefore the memory usage is linear to the level of detail, not exponential.
Since clipmap based rendering front-ends have recently become capable of displaying data at display resolution on commodity PCs, the new bottleneck is the preparation of the clipmap. We present an update strategy that feeds very large terrain data to clipmap based rendering front-ends, thereby exploiting the special properties of the rendering algorithm.

*Zuse-Institut Berlin, D-14195 Berlin, Germany
†Zuse-Institut Berlin, D-14195 Berlin, Germany

This data can be static satellite imagery, but since only small regions of interest are processed, almost interactive manipulation of large terrain data (height and color) becomes possible. You can for example adjust brightness and contrast to match color maps of different sources, overlay the DEM with modifications of different planning scenarios or add vector data layers that are rasterized just in time. This enhances clipmap based terrain rendering to a powerful interactive visualization technology.
We focus deliberately on well-known simple and robust techniques. Efficient terrain rendering does actually not require complex algorithms and can be implemented using a few building blocks such as clipmaps, acyclic filter graphs, and manager/worker-multithreading.

2 Related Work

Losasso and Hoppe presented a terrain rendering algorithm based on clipmaps in [LH04]. The clipmap focus follows the position of the viewer. Therefore the area near the viewer can be rendered at high levels of detail while the regions further away are displayed in a lower resolution. This matches the distortion of the perspective projection, so each area is rendered at a resolution that roughly results in a fixed primitive size in screen space. Asirvatham and Hoppe improved the performance of this method in [AH05] by moving nearly all rendering operations to the GPU, leaving only decompression and clipmap updating to the CPU. Clasen and Hege then extended it to spherical domains, showing that this rendering front-end is actually well capable of handling planet-sized datasets at high resolutions [CH06].
Tanner et al. already described a multithreaded clipmap update system in [TMJ98]. We tailored their general strategy to fit the needs of terrain rendering on commodity PCs without specialized hard- or middleware support. In the following we describe the complete system and explicitly emphasize the differences to [TMJ98]. The main difference for visualization applications is the extension from static source images to an interactive framework.
Systems similar to [TMJ98] have been developed in commercial contexts, for example the MultiGen-Paradigm Virtual Texture. The technical documentation [Eph06] describes this approach for the power user, giving a general impression of the design of their implementation. It is also focussed on static imagery and provides no information on the data processing pipeline.
Several terrain rendering front-ends based on an explicit triangulation of the height data have been developed, such as [LP02], [CGG+03], [BPS04], [WMD+04], and [GMC+06]. Although these systems are highly optimized and provide high quality output and interactive framerates, they are harder to implement since height data and color textures cannot share the same algorithms. This results not only in two distinct level-of-detail systems but also imposes restrictions on the the triangulation due to the texture tile boundaries (see [WMD+04]).
Geometry clipmap based systems can use a unified codepath for height maps and color textures. They require no preprocessing such as triangulation, therefore enabling efficient real-time synthesis. By leveraging the high processing power of current GPU, geometry clipmap systems can reach a relatively high performance without obscuring the source

code with special case handling and elaborated optimizations. While view frustum culling for a single camera with 6 degrees of freedom is already non-trivial on any terrain rendering front-end, doing so for a shadow mapped scene becomes quite involved. Brute force geometry clipmap rendering might not outperform specialized triangulation based systems, but it is much easier to get to an usable performance level even without culling.

3 Rendering Front-End

We implemented a rendering front-end based on [CH06] that works as follows: The static mesh for each level of detail is initialized once on program startup and stored on the GPU. For each frame the current position of the camera (projected straight down to the surface of the planet) is used to set the new clipmap focus. The clipmaps for height map and color map are updated accordingly. The height clipmap is used to displace the vertices of the mesh of the corresponding level of detail. This is done on the GPU in a separated pass by rendering to a texture and copying it to a vertex buffer as texture fetches in the vertex shader are not supported by all GPU vendors. This has the additional advantage that this step can be omitted when the camera position remains unchanged between two rendering passes. The displaced vertices, the static connectivity and the color clipmap are then used to render the terrain.

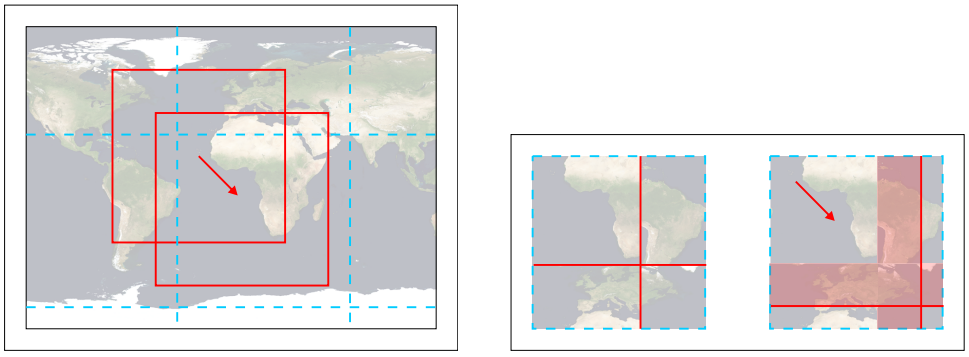
3.1 Clipmap

The original clipmap by Tanner et al. has been implemented as part of IRIS Performer, a visualization middleware. It provides an interface that hides modes of the internal workings to be used as a direct mipmap replacement. Although this greatly simplified the usage by application developers, terrain rendering front-ends like [AH05] can benefit from knowledge about the underlying clipmap since the front-end can skip entire levels instead of rendering them with an artificially limited texture resolution. This motivates an interface that provides more control about the update process.
Current consumer GPU do not support clipmaps directly (e. g. through the OpenGL extension `GL_SGIX_clipmap`), so we have to manage them explicitly using a stack of textures and the usual texture update functions such as `glTexSubImage` and the pixel buffer object extension `GL_ARB_pixel_buffer_object`.
The first initialization of a clipmap is straight forward: Each level is centered around the focus point of the clipmap. The first texture covers the entire terrain and each following level covers a fourth of the previous level (half length in both dimensions). When the focus point moves (see Fig. 1(a), the following cases exist:

1. The focus point moves less than one pixel, so no update is required.

2. The focus point moves less then the edge length of the covered area in both dimensions. In this case some parts of the previous contents can be reused.

3. The focus point moves further away, so the entire level has to be updated.



(a) (b)

Figure 1: (a) Since the clipmap contains only a subset of the corresponding mipmap, it has to be updated when the focus (e. g. camera position) changes. (b) Toroidal indexing results in relatively small update stripes instead of a full clipmap rebuild.

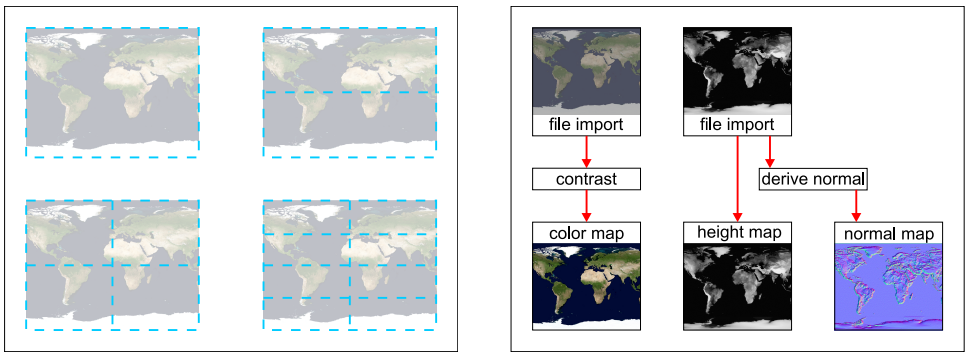
The second case is the most interesting one since clever reuse of existing data can reduce the bandwidth requirements significantly. Tanner et al. solved this by toroidal indexing: The focus point is not set to the center of the texture but to its world position modulo the size of the current level. Therefore the content has a fixed position in the texture and does not have to be moved (see Fig. 1(b)).

4 Texture Generation

When the camera moves smoothly through the scene, most clipmap updates change only a few texels. Given an efficient rendering front-end, this can go down to $1 \times N$ (for texture size $N \times N$) texel small update stripes that have to be calculated on every frame. This results in highly inefficient texture generation for almost anything more complicated than a simple copy operation. If, for example, a jpeg is used as image source, only full DCT blocks of 8×8 pixels could be decoded. Many image synthesis algorithms also perform better on larger blocks because higher code and data locality improves branch prediction and cache usage.
Tanner et al. divide the map into tiles with a size in the same order of magnitude as the clipmap texture. Smaller tiles improve the paging latency, but larger tiles are processed more effiently. These tiles are generated by image sources and stored in tile caches.

4.1 Image Sources

An image source returns a section of a map when given the section size and offset. Since we operate on a fixed tile grid, we pass the grid size and the grid coordinates. We use a grid hierarchy that corresponds to the mipmap levels: A grid subdivides the map into $2^m \cdot 2^n$



(a) (b)

Figure 2: (a) The map is subdivided into different grid resolutions that can be anisotropic when using spherical clipmaps. (b) Image sources form a dependency tree: In this example, the color clipmap receives the tiles from a contrast filter which depends on a file loader. The height clipmap and the normal clipmap share the height map file loader source as the normal map is generated on-the-fly.

tiles (see Fig. 2(a)), with $m = n$ for planar clipmaps and $m > n$ for spherical clipmaps (see [CH06]).
Tanner et al. have a fixed source design: Since their approach tries to mimic a conventional mipmap as close as possible, they have one large mipmap stored on the disc that is paged into the clipmap transparently. However, this simple cache logic can be easily replaced by a texture synthesis framework: We use a generalized image source system that generates the tiles on-the-fly (Fig. 2(b)). Similar to the usual clipmap usage, our main image source is a file loader based on the ECW file format (see [Uef01]). This wavelet format is optimized for streaming and region-of-interest decoding. We used it for a Landsat texture with about $1.4 \cdot 10^6 \times 0.6 \cdot 10^6$ RGB pixels compressed to 26 GB in our tests ($97 : 1$). Losasso et al. [LH04] used likewise a hierarchical file format with spatially localized bases to enable efficient region-of-interest decoding.
One of the main advantages of this flexible clipmap back-end is the ability to reuse sources in different clipmaps. For example normal maps are usually derived from height maps. Many common operations can be implemented based on this image source framework:

1. Map Resampling
   Most georeferenced maps do not cover the whole planet. A map loader source can therefore realign and possibly resample the map on the fly to hide this fact from the following rendering steps. Missing texels are filled with an arbitrary value, e. g. transparent or black.

2. Vector Rasterizer
   Apart from raster data, vector shapes are quite important in terrain rendering applications. These should be rasterized in map space and rendered together with the under-

lying height map to avoid visible accuracy errors as described by [KD02]. They use an *on-demand texture pyramid* which fits conceptionally well to our clipmap source system.

3. Overlay
   Several maps can be overlaid on-the-fly. This is useful for example when only parts of a map are available in high resolution or to implement the thematic lens effect introduced by Döllner in [DBH00].

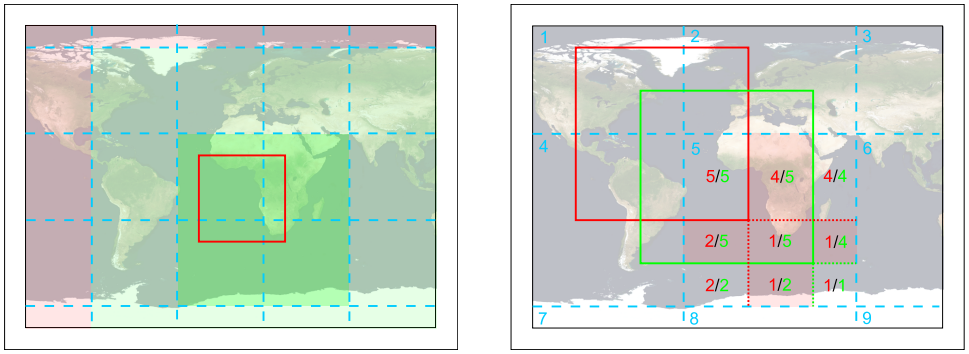
4. Detail Synthesizer
   When a few representative high resolution sample textures are given for a low resolution map, they can be used to give a visual impression of the missing details using the constrained texture synthesized by Wang et al. [WM04]. Although this method adds guessed information to an otherwise reliable data set, it is a powerful tool when applied judiciously. If only the general appearance of a terrain should be visualized, pure synthesis algorithms can be used. For example Berger sketched in [Ber03] a real-time texture synthesis technique suitable for terrain rendering that used spectral noise to generate the different levels of detail.

5. Normal Map
   Normal maps are usually derived from height maps and not sampled directly. This requires only one fast additional filter in the source tree instead of a whole precalculated map which would quadruple the required disk space.

6. Filters
   Adjusting texels without context is one of the fastest ways to modify the image. Changing hue, brightness, contrast, saturation, gamma curves etc. adds virtually no overhead (texture decompression or synthesis is noticable slower). These operations are nonetheless invaluable because they work without expensive precalculation or overhead in the rendering front-end. Since only the visible tiles are processed, changing the parameters of these sources is almost interactive.

4.2 Tile Cache

The tile cache decouples the clipmap update algorithm from the image sources. Image sources store their output in a source-specific tile cache and the clipmap update routine gets it from there. Tanner et al. used a fixed cache size of n^2 tiles that are accessed in the same toroidal way as the clipmap textures. We implemented a more flexible solution based on an associative array: The tile locator that identifies a tile unambigously is used to index the tiles. There is no fixed limit on the size of the cache, instead it is regularly pruned according to an arbitraty criterion. We currently use the distance to the focus point (see Fig. 3(a)), although more intelligent strategies would model the basic idea better: Those tiles that are least likely to be used again should be removed. The distance correlates with this, but camera movement and view direction could provide valuable hints.



(a) (b)

Figure 3: (a) While tiles at the current clipmap position should be kept to avoid regeneration on the next clipmap update, the tiles that are not directly adjacent to these can usually be discarded to free memory. Keeping the adjacent ring avoids cache thrashing when the camera moves back and forth over a tile border. (b) The texture update depends on exactly four tiles. The two wrap-around positions split the texture (displayed over tile 5) into nine update regions (red number: previous tile index; green number: new tile index)

5 Clipmap Update

We divide the clipmap updates into two steps: Gathering the necessary map tiles and copying the data to the clipmap texture.

5.1 Quads

The number of tiles required for a clipmap update depends on the ratio between tile size and clipmap texture size. Instead of the arbirary ratio of Tanner et. al, we prefer using tiles of the same size as the clipmap texture. This reduces the updating process to a small and efficient algorithm:
Using a size ratio of $1 : 1$, a clipmap update requires exactly four map tiles (ignoring the degenerate case when the clipmap contains exactly one map tile). These map tiles form a so-called quad. By the time of the clipmap update, the tile cache should already contain the necessary tiles. Therefore we can just retrieve them and pass the quad to the following step. Fig. 3(b) illustrates this with a simple example: The previous clipmap location is represented by the red square, the new one is drawn in green. The new clipmap intersects the map tiles 1, 2, 4 and 5 which are passed to the update step.

5.2 Update Regions

If the focus point has been moved by more than one tile, the complete clipmap texture has to be updated and no further optimizations can be applied. However, the common case in

terrain rendering is a relatively slow moving camera, so we can reuse parts of the clipmap texture.
We describe this process by means of the example in Fig. 3(b): Since the clipmap texture content is placed modulo the tile size, we draw the clipmap texture over map tile 5. As Fig. 1(b) illustrated, the toroidal indexing splits the clipmap texture into four regions, each corresponding to a different map tile. Splitting the clipmap texture by both the old and the new wrap-around position, we end up with nine regions. Now we can determine the corresponding map tile for each part for both the old and the new focus point. These two tile indices are compared for each region, and only those regions with a differing tile index have to be updated.
The map tile indices are then used to get the correct tile from the quad. Now the region can be copied directly from the tile to the clipmap texture without translation.

6 Multithreading

Terrain rendering as presented in [AH05] is almost completely done on the GPU which runs asynchronously to the CPU. The bottleneck becomes visible at the clipmap update routine: At high framerates only small stripes of the clipmap texture have to be replaced on every frame which can be interleaved nicely with the rendering routines. But, depending on the camera movement, every now and then a new map tile has to be generated. This relatively large task stalls the update process and in turn the rendering, so passing over the tile grid lines results in noticable stutter.
We move the tile generation task to separate threads to avoid this. Although the total workload remains the same, the tile generation can be spread in time over multiple frames on single CPU systems, avoiding the occasional stutter. Multicore CPUs and systems with multiple CPUs can run the tile generation in parallel with the rendering thread.
Tiles are enqueued according to their priority. It depends on the visibility (pre-fetched tiles are less important than visible ones) and level (lower (coarser) levels are more important as they are rendered first). Tiles within one priority class have no special order.
We employ the well-known manager-worker pattern (see [FK97]) to handle the parallel generation. The list of required tiles is determined by the render thread and sent to the manager thread. The render thread can now continue its main task while the manager distributes the workload among the worker threads and deals with tile dependencies. Tile dependencies can be resolved by a depth-first traversal of the source tree starting at the final soures. This yields the correct order in a single pass.
Thread synchronisation is limited to three interfaces: The two tile queues between renderer/manager and manager/worker are common message queues. The third synchronization is required inside the tile cache because multiple workers can write to the same cache simultaneously while the render thread tries to read. We use one mutex per cache which appears to be sufficiently fine grained. Since the tiles themselves are immutable, no further synchronization is required there.
The render thread runs with the highest relative priority, the manager thread below that and the worker threads at the lowest level. This way the render thread is not blocked by the tile

generation process and can continue updating the display at a coarser resolution to improve interactivity.

7 Algorithm

When the application is initialized, a fixed number of worker threads is spawned. They simply block inside the message queue when no tasks are available and take no CPU time. No further setup is required.
Clipmap updates and rendering are handled in a single loop over the visible levels for each frame (Alg. 1). These are determined by the front-end (see [CH06]). The other two input parameters are the current clipmap focus and the time limit. The latter is used to target a fixed frame-rate.

Algorithm 1: Per frame update and rendering.

```
Input: minLevel, maxLevel, timeLimit, focus
1  begin
2      if focus changed then
3          foreach clipmap do                          // height, color, etc.
4              set focus;
5              gather new tile tasks → taskList;
6          end
7          cancel previous tile tasks;
8          enqueue taskList;
9      end
10     for i ← minLevel to maxLevel do
           // prepare level
11         foreach clipmap do
12             get quad from tile cache;                       // can block
13             update clipmap texture;
14         end
           // render level
15         nextLevel ← i +1;
16         if nextLevel ≤ maxLevel
17         ∧ IsReady(nextLevel)
18         ∧ GetCurrentTime() < timeLimit then
19             render ring level;
20         else
21             render full level;
22         end
23     end
24 end
```

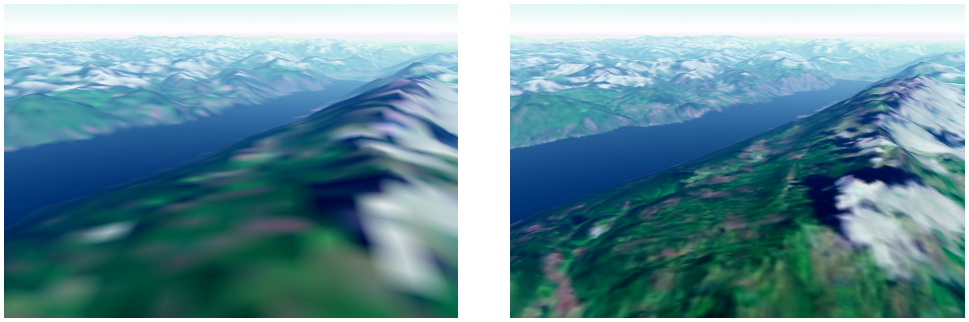
If the clipmap focus has changed since the previous frame, the required tiles might have changed. Therefore we iterate over all clipmaps and gather the new tile generation tasks. The worker threads are still running during this step since the probability of generating usable tiles is quite high in the common case of slow camera movements. When all tasks are known, the current tasks are cancelled and the new ones are enqueued. Cancelling the tasks does not interrupt the worker threads, it affects only the message queue of the manager. This is also because of the high probability of generating usable tiles.
The following loop over the rendering levels starts with the level that has the lowest resolution and proceeds with increasing resolution. This order ensures that we can interrupt the rendering process and still cover the whole terrain, just with less detail than expected. Note that this order is different from [LH04]: Losasso renders from fine to coarse to exploit hardware occlusion culling. This results in higher framerates for a given level of detail, but does not allow a fixed framerate.
At first the current level of each clipmap is prepared. This requires the tiles that are generated asynchrously, so getting the quad can block if they are not ready. The following update just copies the appropriate regions of the quad to the texture. When all clipmaps are ready, we can actually render the terrain. There are two alternatives: If further levels of detail follow, we render a ring, otherwise we render a disk (see [AH05]). It depends on three conditions whether a finer level follows:

1. The level visibility calculation of the rendering front-end limits the maximum resolution. Since image quality is also limited by the output device, increasing the terrain resolution to more than one primitive per screen pixel is rarely useful.

2. If preparing one of the clipmaps of the next level would block, we stop the rendering process at the current level. This keeps the framerate predictable, and there's a good chance that this level can be rendered on one of the next few frames due to the asynchronous worker threads.

3. If the time limit is reached, no further levels should follow. This is a rare case since the operations in this loop usually do not take much CPU time. It can actually happen due to a blocking update of the lowest level when the camera altitude is increased rapidly. However, most of the time it's just a safety check against unfortunate thread scheduling and other external influences.

Tanner et al. handle this differently because of they target full transparency for the user: Instead of explicitly communicating the availability of clipmap levels, they simply limit the *MaxTextureLOD* value, enforcing clipmap lookups in the lower levels only. Since our terrain rendering front-end also generates the geometry based on the clipmap, there's no benefit in rendering highly detailed meshes of coarse raster data. Therefore we prefer using an application specific method to deal with level availability.



(a) (b)

Figure 4: (a) The full area is rendered at a low level of detail if the more detailed clipmap levels are missing. (b) The same scene rendered in full quality

8 Results

8.1 Visuals

The image quality depends primarily on the rendering front-end. The only effect introduced by our method is the omission of higher levels of detail: Fig. 4(a) shows a view of the Alps where the rendering has been stopped a few levels early because of delayed clipmap updates.
Fig. 4(b) shows the same scene a few frames later, when the required data is available.

8.2 Resources

The following results were produced on a dual CPU Xeon (3.2GHz) with 2 GB RAM and a NVidia Geforce 7900GTX (512 MB RAM), running Windows XP x64. The color map is a Landsat satellite image with $1.4 \cdot 10^6 \times 0.6 \cdot 10^6$ 8 Bit RGB pixels (2 TB compressed to 26 GB). The height map is a SRTM data set with $432 \cdot 10^3 \times 216 \cdot 10^3$ 16 Bit greyscale pixels (180 GB compressed to 5 GB). The color clipmap uses a texture size of 256^2, the height clipmap uses 64^2 texels. The normal map is derived on the fly from the height map at full color texture resolution. About 800 MB RAM were used.
Due to framework issues, the maximum framerate is limited to 64 frames per second (independent of the vertical refresh rate). We chose front-end settings that allow rendering at this speed for a static camera when all clipmap data is ready. The framerate is therefore only limited by date processing and clipmap updates. Table 1 shows the maximum velocity at which the back-end could keep up with the front-end and the corresponding framerates. At velocities higher than about $altitude \cdot \frac{4}{5s}$, the highest levels of detail are occasionally ommitted while the framerate stays quite constant. Moving slower also does not affect the framerate significantly due to the overhead in the clipmap update routines (more levels are considered visible at lower altitudes).

altitude	levels	velocity	framerate
800 km	1-5	640 km/s	64 fps
80 km	3-9	64 km/s	60 fps
50 km	3-10	40 km/s	50 fps
20 km	4-11	16 km/s	38 fps
6000 m	5-13	4800 m/s	36 fps
2000 m	5-14	1600 m/s	36 fps
800 m	6-16	640 m/s	32 fps
200 m	7-17	160 m/s	32 fps
50 m	8-20	40 m/s	28 fps

Table 1: The maximum velocity at which the back-end can keep up and the corresponding framerates

A second test using a height generator based on 3D simplex noise and a derived color map showed basically the same behaviour, although it was about 5 times slower.

9 Conclusion

While terrain rendering using clipmaps depends on low latencies but requires little CPU time, preparing the clipmaps takes high CPU bandwidth but is of limited urgency. We presented a way to decouple these two tasks to enable terrain rendering with a constant high framerate and an adaptive level of detail. Most details are ommited when the camera moves fast, which is exactly the situation in which the user would not perceive it anyway. As soon as the camera stops, the resolution converges to the maximum for the given terrain rendering front-end. This allows terrain visualization applications with state of the art rendering front-ends that do not have to rely on explicit coarse replacement geometry or other placeholders to remain highly interactive in a professional visualization setting, including on-the-fly data manipulation.

10 Future Work

Although the current solution already works fine, we found a few points that could need further methological improvement:

1. Prefetch
   The cache prefetch strategy currently only takes the clipmap focus into account. However, prefetching the tiles lying on an extrapolated camera path would be preferable as those behind the camera are less likely needed for the following frames. It could also be desirable to allow explicit prefetching for non-interactive visualizations where the application knows in advance which regions are visited next.

2. Clipmap Mipmaps
   There's a general problem with manual clipmaps as introduced by Asirvatham in

[AH05]: When only small portions of a texture are updated, the cost of regenerating a mipmap for this texture are relatively high. However, mipmaps with anisotropic filtering would improve both rendering performance and quality significantly when the camera aims at the horizon and not the geocenter. However, avoiding the explicit clipmap texture and passing the quads to the GPU would circumvent this issue, but at a cost of four times more texture samplers and no interpolation between tiles which is equivalently undesirable.

3. Scheduling
   Currently we do not employ any clever scheduling strategy to feed the worker threads. This might result in a bottleneck when using complex source trees on systems with many CPUs, but we didn't do any further research in this direction, yet.

11 Acknowledgements

We would like to thank the Bundesministerium für Bildung und Forschung (BMBF, http://www.bmbf.de/) for supporting the SILVISIO project (FKZ 0330560B). The free ER Mapper SDK for ECW and JPEG 2000 images (http://www.ermapper.com) has been used as image loader back-end. We also appreciate the publication of the Landsat and Blue Marble texture sets and the Shuttle Radar Topography Mission data by NASA (http://earthobservatory.nasa.gov/) and their distribution by http://www.geotorrent.org. We also used the SRTM V3 data by CGIAR-CSI.

References

[AH05] Arul Asirvatham and Hugues Hoppe. *GPU Gems 2*, chapter Terrain Rendering Using GPU-Based Geometry Clipmaps, pages 27–46. Addison-Wesley, 2005.

[Ber03] Daniel R. Berger. Spectral texturing for real-time applications. Siggraph 2003, Sketches and Applications, July 2003.

[BPS04] Xiaohong Bao, Renato Pajarola, and Michael Shafae. SMART: An efficient technique for massive terrain visualization from out-of-core. In *Proceedings Vision, Modeling and Visualization (VMV)*, pages 413–420, 2004.

[CGG+03] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet-sized batched dynamic adaptive meshes (p-bdam). In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 20, Washington, DC, USA, 2003. IEEE Computer Society.

[CH06] Malte Clasen and Hans-Christian Hege. Terrain rendering using spherical clipmaps. In *EuroVis Proceedings*, 2006.

[DBH00] Jürgen Döllner, Konstantin Baumman, and Klaus Hinrichs. Texturing techniques for terrain visualization. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 227–234, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[Eph06] Anton Ephanov. Understanding virtual texture. MultiGen-Paradigm Support, March 2006.

[FK97] Bernd Freisleben and Thilo Kielmann. *Coordination Languages and Models, Second International Conference COORDINATION '97 Berlin, Germany, September 1-3, 1997 Proceedings*, volume 1282 of *Lecture Notes in Computer Science*, chapter Coordination Patterns for Parallel Computing, pages 414 – 417. Springer Berlin / Heidelberg, 1997.

[GMC+06] Enrico Gobbetti, Fabio Marton, Paolo Cignoni, Marco Di Benedetto, and Fabio Ganovelli. C-bdam - compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum*, 25(3), sep 2006. To appear in Eurographics 2006 conference proceedings.

[KD02] Oliver Kersting and Jürgen Döllner. Interactive 3d visualization of vector data in gis. In *Proceedings of the 10th ACM International Symposium on Advances in Geographic Information Systems (ACMGIS 2002)*, pages 107–112, Washington D.C., November 2002.

[LH04] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *Siggraph 2004*, volume 23 (3), pages 769–776, New York, NY, USA, 2004. ACM Press.

[LP02] Peter Lindstrom and Valerio Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002. http:www.cc.gatech.edu/ lindstro/papers/tvcg2002/paper.pdf.

[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 Press.

[Uef01] Christoph Ueffing. Wavelet based ecw image compression. *Photogrammetric Week 01, Wichmann Verlag, Heidelberg*, pages 299–306, 2001.

[WM04] Lujin Wang and Klaus Mueller. Generating sub-resolution detail in images and volumes using constrained texture synthesis. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 75–82, Washington, DC, USA, 2004. IEEE Computer Society.

[WMD+04] Roland Wahl, Manuel Massing, Patrick Degener, Michael Guthe, and Reinhard Klein. Scalable compression and rendering of textured terrain data. In *Journal of WSCG*, volume 12, 2004.