Real-time Rendering and Manipulation of Large Terrains

Shiben Bhattacharjee
shiben@research.iiit.ac.in

Suryakant Patidar
skp@research.iiit.ac.in

P. J. Narayanan
pjn@iiit.ac.in

Center for Visual Information Technology
International Institute of Information Technology, Hyderabad

Abstract

Terrains are challenging geometric objects for real-time rendering and interactive manipulation. State-of-the-art terrain rendering systems use custom, multiresolution, representations like geometry clipmaps for fast rendering on the GPU. In this paper, we present a system that exploits the power and flexibility of the modern GPUs to store, render, and manipulate terrains with minimal CPU involvement. The central idea is to use a regular-grid representation and fixed size blocks/tiles that change in resolution. The potentially visible portion of the terrain is cached at the highest necessary resolution and is rendered from the GPU. The CPU sends a light geometry template which is expanded by the Geometry Shader to the triangles, using the heights stored in the GPU Cache. The CPU performs a coarse culling of the tiles with the GPU performing fine culling. The GPU cache is updated continuously as the viewpoint changes. Our system enables the terrain to be modified procedurally or edited interactively on the GPU with no CPU involvement. The terrain can also interact with a large number of external objects in real-time entirely within the GPU. We achieve a consistent rendering rate of 100 frames per second with terrain modification and interactions as well as a triangle rate of upto 350 million per second on an Nvidia 8800 GTX GPU for large terrains, with a CPU load below 10%.

1. Introduction

Terrains are of great interest in flight simulators, geographic information systems, games, etc. A regular grid of heights is a natural representation for them. Rendering a height-map is straightforward with an $O(n^2)$ rendering load for an $n \times n$ terrain. Natural terrains contain many flat regions which can be converted to a triangulated irregular network representation. The irregular representations contain fewer triangles but more complex to represent and manipulate. Such irregular representations have recently been giving way to a regular grid representation, especially after the availability of fast graphics hardware.

Terrain rendering is a well-studied problem. Triangulated irregular networks are created from regular grids with connectivity typically decided using a triangulation process such as Delaunay's [4, 10, 3, 7, 6]. They provide rendering efficiency at the cost of ease of performing other operations. Terrains have been partitioned into fixed size square patches of different resolutions. The tiled structures provide compact representation and easy rendering. The block boundaries can show artifacts which are taken care of using special zero-area triangles and stitching [13, 12, 5]. The fixed size blocks also limit the range of resolutions supported when a tile is reduced to a single height.

Losasso and Hoppe introduced a multiresolution, fixed memory size scheme for efficient representation and rendering of large terrains, called the geometry clipmaps [12]. They use a square region around the viewer as a geometry clipmap with high resolution at the centre and lower resolutions on the outer rings. The fixed memory structure involves constant rendering load. The geometry clipmap were stored in the GPU and rendered from there [1]. Geometry clipmaps provide good rendering performance, but the representation does not lend itself to editing or modification of the terrain, which is possible especially on today's GPUs. Terrains are traditionally considered static and fixed. Deforming and editing are performed rarely during visualization. Earlier work on terrain modification include multiresolution detail patches by He et al. [9] and modelling soil slippage by Li and Moshell [11]. The height-maps are amenable to quick editing, unlike the irregular representations. Atlan and Garland edit the terrain in real-time using a few editing strokes for applications such as geological simulations [2], using a wavelet-based representation. They use a two-step approach to recover the terrain and to edit it. Our evolution and editing method is similar, but achieves better performance by performing it completely on the GPU, with support for real-time, simultaneous editing and rendering.

In this paper, we present a scheme to render terrains, de-

form them, edit them, and perform physics involving them at real-time rates. We use a representation that combines the fixed-size structure of geometry clipmaps and the regularity of tiled blocks. The terrain is cached on the GPU using fixed-size rectangular blocks. The resolution of the blocks depends on the view and changes with height of the camera. A blocked, tiled, height-map representation resides at the GPU cache at all times for fast rendering and real time modification. The cache is kept updated in extent and resolution by sending data when needed.

The main contributions of this paper are: (a) A terrain rendering system that achieves a rendering speed of 100 frames per second on arbitrarily large terrains without the CPU, the GPU, or the bandwidth between them being the bottleneck.(b) A way to interactively modify and interact with the terrain simultaneously with rendering, performed entirely in the GPU at real-time rates. This enables terrain deformations, interactive editing and the computation of simple physics of external objects interacting with the terrain. The following innovations make the above possible. (i) A terrain representation that uses fixed-size blocks of grids and GPU caching that enables fast rendering and correct editing and manipulation. (ii) A scheme of sending light geometry templates from the CPU to the GPU, which are expanded into the actual geometry. This keeps the CPU free to do other tasks while the GPU performs the bulk of the rendering work. (iii) A two-level culling scheme with the CPU culling in units of large tiles and the GPU culling in units of smaller tilelets for high rendering performance. The rendering rate doubles with this. (iv) Clever interleaving of data transfer from the CPU to the GPU to keep the cache updated correctly without affecting the rendering rates. This guarantees 100 fps rendering of arbitrarily large terrains. (v) Fast, interactive and procedural manipulation and editing of GPU-resident terrains using the fragment shader. The highly parallel GPU resources are employed profitably to do this, improving the system performance.

We demonstrate the performance of our system using an Nvidia 8800GTX GPU. We can achieve a fixed framerate of 100 on a $1M \times 1M$ terrain which uses 2 TB for the heights[1]. Our system renders upto 350 million triangles per second on parts of a flight path and achieves an average rate of 160 million triangles per second. The frame rate of 100 can be maintained even while half the terrain is deforming or is being edited or when 256K balls are bouncing on it. We exploit the advanced SM4 (Shader Model 4.0) features of the GPU to achieve the high performance. The terrain representation and rendering are explained in Sections 2 and 3 respectively. The caching system is explained in Section 4. Terrain manipulation schemes are presented in Section 5. Section 6 presents experimental results. Some concluding remarks are given in Section 7.

[1]The large terrain is a periodic extension of the $16K \times 16K$ Puget Sound terrain. The terrain system is unaware of the replication. The CPU module that loads the terrain is aware of the fact and returns pointers to existing data when going beyond

2. Terrain Representation

We represent the terrain as a regular 2D grid of heights with a fixed post distance in X and Y directions. Fixed-size *blocks* are used as the base units of storage and transfer from the CPU to the GPU. A block consists of *tiles*, which are the basic rendering units. Tiles extend in the ground XY plane and take part in view frustum culling. Tiles are further divided into smaller *tilelets* (Figure 1(b)).

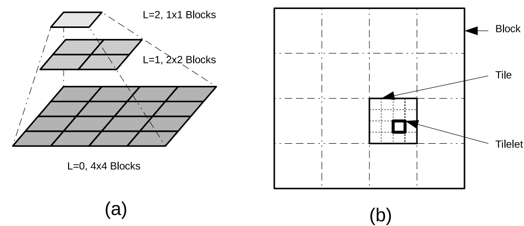

Figure 1. (a) An terrain with highest resolution stored in 4×4 blocks, next in 2×2 blocks and so on, using fixed size blocks. (b) A block with 4×4 tiles each with 4×4 tilelets

CPU Representation: The terrain is stored in main memory and sent to GPU as blocks. We use blocks of size 1024×1024. The CPU also stores all lower resolutions of the terrain as blocks of the same size to facilitate quick transfer of any resolution to the GPU (Figure 1(a)) at the cost of a maximum of $1/3$ more memory. View frustum culling takes place in terms of tiles. Tile size should balance the culling and rendering loads. We use larger tiles in our system, currently 256×256, since geometry is discarded also at the GPU as explained later. Different parts of the

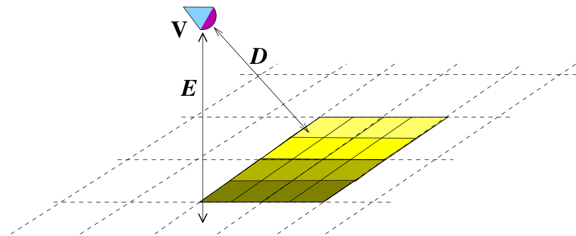

Figure 2. The GPU Cache (shown shaded) is a section of the terrain on the GPU at a resolution determined by the elevation E of the viewpoint V. Rendering resolution depends on the distance D.

terrains need to be rendered at different levels of detail or resolutions. The *level-of-detail (LoD)* at which a tile is rendered depends on two factors: the elevation of the viewpoint and the distance from the viewpoint. The rendering resolution reduces as the elevation E or the distance D increases (Figure 2). Resolution is changed in discrete steps by doubling or halving the post-distance. *Merge level* denotes the LoD of the whole terrain determined by the elevation and *distance level* denotes the LoD of an individual tile due to its distance from the viewpoint. Both take discrete integral values. Level 0 represents the terrain at the highest detail; level $i + 1$ represents the terrain at half the resolution of level i. The LoD is expressed as the number of level shifts from the highest resolution available. The *rendering* LoD of a tile is the sum of merge and distance levels.

Post distance is doubled with each reduction in resolution. Transition distance of LoD of terrain depends on post distance and also doubles with resolution decrease. Farther tiles render with low detail (higher LoD number). Resolution reduction is achieved easily on our representation by dropping alternate rows and columns. Thus, an LoD level l has a post distance that is 2^l times the post distance at level 0. Thus a higher resolution block contains all lower resolution ones, which can be generated by sub-sampling. We choose sub-sampling instead of filtering for creating low resolutions because it preserves height values whereas filtering changes the heights in lower resolutions. Sub-sampling is also fast but produces no artifacts when combined with our blending scheme explained in the next section.

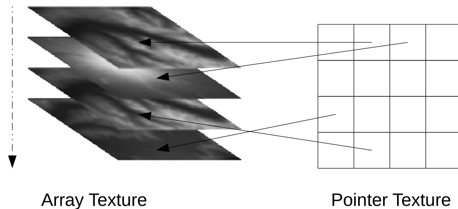

Figure 3. A 16 layer array texture (GPU Cache) with a 4×4 pointer texture storing layer IDs.

GPU Representation: The *GPU Cache* holds a contiguous 2D grid of blocks at the merge-level resolution around the *point of reference* determined by the camera location. The resolution depends on the view elevation E (Figure 2) and is the highest resolution needed for rendering from that elevation. Tiles are further divided into 2×2 *tilelets* by the GPU and used for the finer *two-level of culling* explained later. The GPU Cache is updated when the merge level changes due to elevation or the region changes due to change in camera location. It is to be noted that the extent on the ground of the blocks and tiles change with the merge level as they have fixed memory sizes.

Implementation Details: The GPU Cache is stored as an *array texture*, introduced in SM4.0. The cache can be attached to a single texture unit and a height can be accessed on the fly by the GPU.Heights are accessed using three coordinates: l to select the block (also called a layer) and x, y to fetch the post from that layer. Each layer of the array texture can be updated randomly. We use a separate *pointer-texture* to store the layer IDs (Figure 3). The pointer-texture is a 2D array of layer IDs and presents the GPU Cache as a contiguous 2D array of blocks. The l coordinate is fetched using the 2D indices of the block. The pointer-texture is updated with new layer numbers when the GPU cache is updated. The unified architecture of SM4.0 provides fast access to the texture for all shader units.

The merge and distance levels provide a unified LoD scheme with nearly constant triangle count on the screen for all elevations of the camera. The amount of data to be rendered also is nearly constant at all elevations due to the shift in resolution. Our system also supports mapping of real texture images to the terrain. These textures are kept in a parallel cache on the video memory with a matching block of texture for each block in the GPU cache.

3. Terrain Rendering

The GPU performs most of the rendering under CPU's coordination. The CPU culls every tile in the GPU cache to the view frustum. It then sends the geometry template, consisting of a VBO (vertex buffer object) of points, for each tile to the GPU. This keeps the CPU load and communications to the GPU very low. The GPU discards tilelets of the geometry that lie outside frustum and expands the rest into the triangles.

3.1. Stage 1: CPU

The 2D grid scheme makes it easy to compute the extents of tiles, blocks, and tilelets using simple calculations. Each tile has an index in the grid of tiles. The CPU eliminates tiles outside the view frustum and computes the LoD level for the rest of the tiles. It then sends the corresponding geometry templates to the GPU.

View Frustum Culling: The orthogonal footprint of the view frustum and its bounding box are estimated in the grid of tiles first, as shown in Figure 4. The bounding sphere of each tile is tested against the six planes of the view frustum and those lying outside are discarded. Tiles that intersect a frustum wall are tagged specially boundary tags as their tilelets will undergo a second level of culling in the GPU.

Level of Detail (distance-level): The GPU cache holds the terrain at the current merge-level. The distance-level of each tile denotes the drop in resolution from the data stored

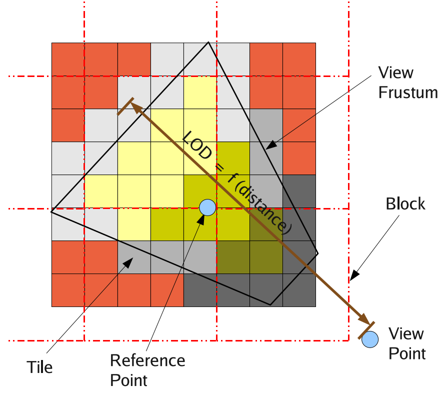

Figure 4. Tiles outside view frustum (marked red) are discarded by the CPU. Intersecting tiles (gray) will go through a second level of culling by the GPU. Interior tiles (yellow) are rendered directly. Rendering LoDs of tiles is a function of distance from the viewpoint.

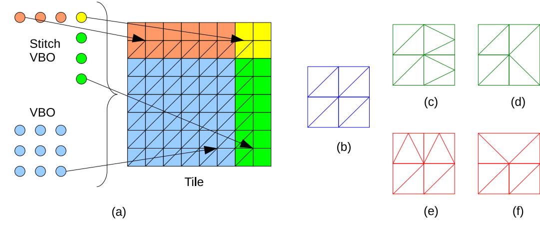

Figure 5. (a) The CPU renders each tile (E.g. size 9×9) as points using two geometry templates, one for the interior (blue) and the other for the boundary (red/yellow/green) of the tile. (b) Tilelet used in the interior of the tiles. In the eastern border (green), tilelet (c) is used when the neighbor has a higher LoD and (d) is used if lower. In the northern border (red), tilelet (e) is used when neighbor has a higher LoD and (f) is used if lower. Yellow region gets handled automatically.

in the GPU cache. Farther tiles are rendered with lower detail and proximate tiles with high detail (Figure 4). The detail factor d_l is computed as $\log_{1.5}(1 + \frac{d}{t})$, where d is the distance E of the mid point of the tile from camera (Figure 2) and t is the current diagonal length of the tile. The term d/t will give linear LoD bands and the logarithm will ensure exponential thickness for equi-detail bands. This results in near-uniform distribution of tile-detail on the screen. The base of the logarithm affects the width of the LoD bands. A value of 1.5 gives acceptable triangle count and quality in our experience and ensures a minimum LoD band thickness of one tile. Thus, adjacent tiles will not differ by more than one level which is necessary for seamless stitching as explained later. The integer part $\lfloor d_l \rfloor$ of the detail factor is used as the distance-level l_d and the fractional part is used as the *morphing factor* α for the entire tile. Morphing of different LoDs is necessary to avoid popping artifacts as explained later.

Rendering: The CPU sends each tile to the GPU along with l_d, the morphing factor, and the boundary flag. To reduce CPU load, a *geometry template* as a VBO of point primitives is sent for each tile. Each point of the template represents a tilelet to be rendered. We currently use 2×2 tilelets. A 128×128 VBO is used for the full 256×256 tile. Smaller VBOs are used if the distance-level is greater than 0 (low detail). The same template can be used for all tiles at a particular distance-level. This process is explained in detail in Section 3.2.

Adjacent tiles can have different resolutions which causes visual inconsistencies at the joints. We use a *stitching* process for border rows and columns to avoid this. Each tile stitches with its northern and eastern neighbors as explained later. Stitching requires an extra row and column of indices of the neighbor to be available to each tile. Thus, the actual tile size used is 257×257 with its last row and column being the first row and columns of the adjacent tiles. CPU sends separate *stitch templates* to effect correct stitching (Figure 5(a)).

3.2. Stage 2: GPU

Tilelet Generation and Culling: The GPU receives the index, the LoD number, the morphing factor, and geometry template for each tile (Figure 5(a)). The index is mapped to the block index of the pointer texture and the layer and tile ID of the GPU Cache. The coordinates of the incoming point primitive of the template represents the top-left corner of the corresponding tilelet and the LoD can be used to compute the other corners. The heights of the corners are fetched from the GPU cache. These four points are tested against the view frustum by the vertex shader. The tilelet is tagged as outside if all four points are outside before sending down the pipeline. In practice, conservative testing is performed to counter possible error introduced by the quadrilateral approximation of the tilelet. This process of second level culling is performed only on the tiles that intersect the view frustum walls as tagged by the CPU. Other points are passed down the pipeline without testing.

The geometry shader of the GPUs can discard primitives from the pipeline or add primitives to it. The tilelets that are tagged by the vertex shader are discarded. This second level culling accomplishes accurate culling with no load to

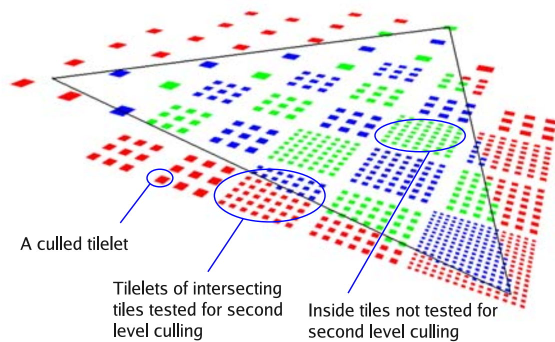

Figure 6. Tilelets after VFC. Farther tiles need fewer tilelets. The red tilelets are discarded by the second level culling on the GPU.

the CPU with performance doubled (Figure 7) We experimented with different tilelet sizes. The smallest tilelet with stitching capability has a size of 3×3. This performs the best due to the deterioration in performance of the geometry shader as the amount of data it outputs increases [8]. The geometry shader generates triangles for the remaining tilelets. The coordinates of the top left point and the LoD number are used to access the 3×3 grid points. The post distance at level l is 2^l times the post distance at level 0. Triangles of the tilelet, as shown in Figure 5(b-f), are sent down the pipeline.

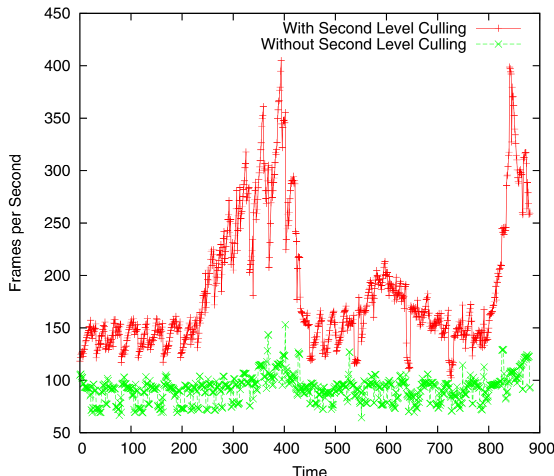

Figure 7. Framerates for a typical flight over the terrain with (red) and without (green) the second level of culling.

Figure 7 shows the performance of the system over a typical flight over the terrain with and without the second level culling. The second level culling improves the system performance by a factor of 2 overall. The camera is looking approximately down between frames 250 to 450. The tiles nearer to the camera contain more tilelets as seen in Figure 6. The second level culling is most effective on them as a result.

3.3. Tile Stitching and Blending

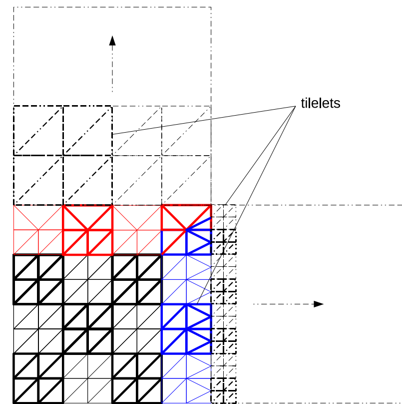

Figure 8. A tile of size 9×9 with a northern neighbor of lower resolution and an eastern one of higher resolution. The tilelets shown in Figure 5 are used for correct stitching.

Stitching is necessary when the northern or eastern neighbor tile has a different LoD. Gaps in the terrain may be visible otherwise. Zero area triangles have been used to alleviate this problem in the literature [12], but is an inelegant and incorrect solution. We use a special L-shaped geometry template with indices of only the northern and eastern borders of a generic tile for stitching (Figure 5(a)).

The stitching templates use the same vertex shader. Their geometry shader selects one of the tilelets in the border areas (Figures 5(b-f)) based on the LoD numbers. Tilelets of the eastern border can be rendered with equal LoD to the parent tile (Figure 5(b)), lower LoD than the parent tile (Figure 5(d)), or higher LoD than the parent tile (Figure 5(c)). The same goes for the tilelets along northern border (Figure 5(b, f, e)). The geometry shader can recognize and render these tessellation styles using the available information (E.g. Figure 8). Our stitching scheme maintains coherence at the borders of tiles with no abrupt changes or gaps. It also requires less number of triangles compared to schemes like zero area triangles and introduce no extraneous geometry.

The morphing factor is used to smoothen the change in LoD to avoiding popping of geometry. The morphing factor α has a range $[0, 1)$ and is used to interpolate

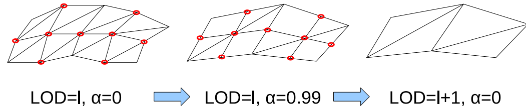

Figure 9. A tile at LoD = l (left) blends its alternate heights (shown in red) with its lower LoD (middle) using α. When the tile shifts its LoD, the change is not noticeable. This process is valid in reverse as well.

between the heights of the current LoD level and of one lower level. Thus, the final height used for rendering is $h = \alpha h_l + (1 - \alpha)h_{l+1}$ where l is the LoD of the tile (Figure 9). When a tile moves from far to near, α changes from 0 to 1 smoothly and improve the shape of the tile. For the corner heights, the morphing factor for the adjacent tile is used, as the neighbouring tile will change its LOD independently.

4. Caching

The GPU Cache contains $N \times N$ blocks at the merge-level resolution, which is the maximum resolution of the terrain needed at the elevation of the camera. The size N depends on the maximum visibility required at the highest resolution. The visibility doubles as the merge-level increases. We use $N = 8$ for most of our experiments, needing storage for 64 blocks on the GPU.We try to keep the GPU cache symmetric with respect to a **reference point**, which is the centre of the orthographic projection of the view frustum onto the ground (Figure 10).

4.1. Lateral Motion of Viewpoint

Lateral motion, pan, and tilt at a constant elevation bring in new data to the GPU cache at the same resolution. We use the position of the reference point in the cached terrain to trigger the data transfer. If the reference point goes outside the central 2×2 block of the GPU cache the cache is re-centered by bringing another row or column of blocks at the current merge level, (Figure 10) discarding blocks on the other side of the cache. We load the new blocks by overwriting the discardable blocks. The data from the CPU is loaded to selected layers of the array texture and the pointer texture is updated to rearrange the layer IDs on the GPU. For an 8×8 GPU Cache with each block taking 2MB of memory (1024×1024, 16-bit height values), a lateral motion cache update needs 16MB of data to be uploaded to the GPU. The data transfer time is controlled using a *job-queuing* scheme explained later.

4.2. Vertical Motion

When the camera goes up, the extent of the visible terrain increases and the resolution decreases. Similarly when the camera comes down, the terrain extent decreases and the detail increases. For this, we change the merge level or the base resolution of the GPU Cache. This process keeps the memory footprint constant without compromising the requirements for rendering.

Ascending Motion: The merge level increases and the resolution halves when the viewpoint moves up. A quarter of the GPU cache can be filled by sub-sampling and merging the current contents of the cache (Figure 10). New data has to be brought to the remaining space. The merging is performed on the GPU using a separate fragment shader pass that sub-samples and copies heights from 2×2 blocks into a single unused block. This is achieved by the binding the target and source blocks as frame buffer objects or textures and drawing a block-sized quad. At the end of the merging process 75% of the blocks will be free (Figure 10). New data is brought to them from the CPU in the proximity order from the reference point and stored in unused layers. The data transfer is triggered before the new area is needed and can complete over a few frames. For an 8×8 GPU Cache, we merge 64 blocks into 16 blocks in the GPU using 16 merge operations. After that, 48 blocks or 96MB are uploaded from the CPU to the GPU.

If the original terrain is one million square, we can get $\log_2 1024 = 10$ global LoDs. We reduce the resolution in factors of 2 until the entire terrain fits into the GPU Cache. Thus the number of merge-levels depend on the cache size and the total size of the terrain. We use a GPU Cache size of 8×8 of $1K \times 1K$ sized blocks for the $16K \times 16K$ Puget Sound data. It contains 14 LoD levels and only one $(= \log_2 16384 - \log_2 8192)$ merge-level before the GPU cache is filled. A $1M \times 1M$ data with the same cache can use 7 $(= \log_2 1M - \log_2 8192)$ merge levels. The data transfer time is controlled using a *job-queuing* scheme explained later.

Descending Motion: When the resolution increases due to a reduction in elevation, the blocks of the cache are replaced by higher resolution blocks, and the total extent of the terrain reduces. This is data intensive as the entire GPU Cache needs to be replaced. A quarter of the GPU Cache that will remain in the view are first identified. The physical area of each block is to be replaced by four high resolution blocks (Figure 10). The increase in resolution is also anticipated ahead of time to avoid visible update changes. For a 8×8 blocks, we have to bring the 128 MB into the GPU memory to increase the merge level resolution. The data transfer time is controlled using a *job-queuing* scheme explained next.

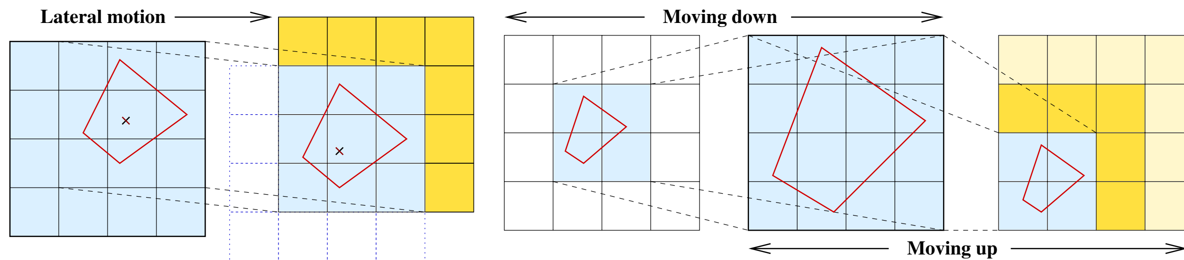

Figure 10. Lateral motion and pan/tilt (left) involve discarding an L-shaped region and bringing in new blocks (yellow) from the CPU. When the viewpoint comes down, the merge level decreases (middle). The extents of GPU cache are halved and data at a higher resolution is brought in from the CPU. When the viewpoint goes up, the extents of the cache are doubled and the existing data is compressed into one quadrant. New data is brought in from the CPU.

4.3. Job Queuing Scheme

Our primary goal is to maintain a steady frame-rate. We treat the data transfer and merge operations as “jobs” and queue them to be executed when the rendering process has time. Running all the operations at the same frame can freeze the rendering at times and affect the quality of visualization. We execute as many jobs as possible to keep the total frame time T_t, safely within the fps constraints. The total frame time is, $T_t = T_r + T_u$ where T_r is time for rendering and T_u is the total time taken by the jobs in the cache updating process. For 100 fps rendering, T_t is 10 ms and we are left with $T_u \leq 10 - T_r$ ms for updating. We steal cycles for the update process when $T_r < 10$ms without affecting the fps.

The transfer of a block of 2MB from the main memory to the GPU takes 2 ms on the current GPUs. A merge operation takes less than 0.5 ms. The number of jobs to be performed is calculated as $n = (10 - T_r)/K$ where K is a constant denoting the worst case time for the job. For example if $T_r = 5$ ms, and $K = 2$ ms for a layer update, then $n = 2$ jobs can be performed per frame. If $n < 1$, we do half jobs, by uploading half of a block. Over some number of frames, all operations are completed. This adaptive job-queuing ensures a frame-rate of 100 in practice without any hiccups.

When the merge-level changes, the GPU cache gets updated completely over a finite time. Until then, artifacts can appear since the cache is mixed with old and new layers. We use *dirty texture* flags to handle this. As soon as the merge-level changes, all the blocks are marked *dirty*. When marked dirty, the renderer uses the lower resolution, as with the old merge-level. As soon as the new layers get updated and older layers are made unused, new blocks are marked clean. This way the rendering remains free of artifacts.

5. Terrain Deformation and Manipulation

Terrains are traditionally used only as static geometric entities. The rectangular grid representation makes it easy to manipulate them interactively or procedurally as well as to simulate interactions of other objects with it.

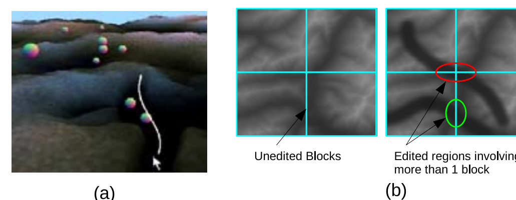

Figure 11. (a) The mouse motion over the screen triggers interactive editing of the terrain. (b) A terrain of 2×2 block (left) and the results of editing it (right). Editing can involve multiple blocks at boundaries (marked)

Interactive and Procedural Manipulation: The GPU representation of the terrain that we use lends itself to interactive and procedural manipulation easily, exploit the computing power and architecture of the modern GPUs. A fragment shader can operate on each height value independently or in relation to its neighborhood. The parameters for the deformation process should be given to the shader. This includes the user inputs like the mouse path for interactive editing and relevant parameters for procedural deformation. A deformation pass is triggered on each block by drawing a block-sized quad after setting up the parameters. Deformation passes are sandwiched between rendering passes. This simulates terrain dynamics in regular frame intervals. Interactive editing of the terrains can also be performed with the user guiding the change in heights (Figure

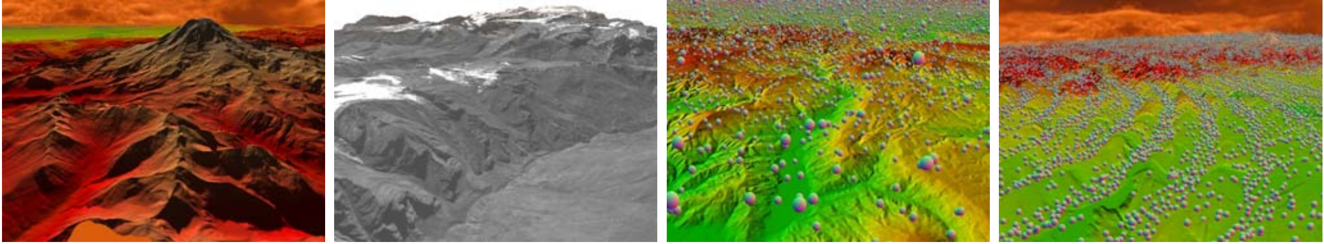

Figure 12. A view of Mt Rainier, a terrain with real texture, realtime physics with balls, realtime physics with a deforming terrain.

11(a)). The screen point is back projected to world point and transformed to the the terrain coordinates to get a point of impact. Heights are modified based on the distance from the point using user selected radius and intensity of impact. Figure 11(a) shows how a channel can be cut on the terrain by dragging the mouse. Multiple blocks may need to be edited, based on the point of impact (Figure 11(b)). For procedural dynamism based on time and distance, each deformation pass makes incremental changes to the terrain, between rendering passes.A deformation pass takes about 250 microseconds per block. We see no drop in framerates when only a few blocks are modified in each frame. With rendering time mostly around 5ms, around 20 blocks can be deformed per frame for 100fps performance. We start a simultaneous process on the CPU to effect the same changes on the base terrain. This is similar to write-through of memory caches. The CPU can keep up with the GPU for user-guided editing since the CPU load is low. For procedural deformations, only the last state needs to be created on the CPU. This can be performed as the CPU is lightly loaded. The GPU cache is a single array texture. It can be bound as a single FBO and modified in place using a fragment shader. Layered rendering of the current GPUs enables independent editing of multiple blocks in single deformation step. The modified terrain can be rendered immediately as the GPU Cache itself is updated in place.

Real-Time Object-Terrain Interaction: Terrains can be used as the base to simulate several interactions with external objects like a bouncing ball. Though the exact physics involved could be quite complex, effective simulation and visualization can be achieved with moderate computation power. We take the example of multiple balls bouncing over the terrain. The positions and velocities of balls are stored as two textures in the GPU memory with one pixel representing one ball. The positions will be updated by a fragment shader using the velocity and time difference in an update pass that takes place between rendering passes. The fragment shader has access to the textures through an FBO for the update pass. The update pass will also implement the physics such as collision with the terrain. The velocity may change as a result of the physics. The terrain height has to be looked up for a given 3D position of the ball to check for collision. This is done by converting the xy location to the GPU cache block and grid coordinates, looking up the layer ID from the block number using the pointer texture, and accessing the height. If balls are present, a rendering pass renders them at their current locations. The vertex shader fetches the positions of a ball, checks for visibility in the frustum and renders it procedurally as a front facing circle. A quarter of a million balls can interact and be rendered with the terrain at 100 fps. The system can achieve 60 fps with 1 million balls[2].

6. Results

We performed all our experiments on an Nvidia 8800 GTX using OpenGL and GLSL shaders on Linux with a Pentium Core 2 Duo CPU running at 2.4 GHz. We use the Puget Sound data, consisting of a 16384×16384 grid of 16-bit heights covering a square region of length about 160 km, for most experiments. We also use the 8192×4096 Blue-Marble grid with earth texture and an 8192×4096 height data with monochrome satellite image as texture. We simulated a very large terrain by first tiling 4 sets of Puget-Sound, flipping it along the vertical and horizontal edges. This $32K \times 32K$ terrain occupies 2 GB of space and can be tiled along X and Y directions infinitely. We simulated a $1M \times 1M$ terrain by replicating it 32 times each in X and Y directions. Replication was effected using modulo computation without additional memory. The data-access module on the CPU was the only unit aware of the replication. The terrain system was unaware of it.

Figure 13 shows the system performance on a flight over the 1 trillion sample terrain. The camera moves laterally till about frame 2000 with significant tilt. The thin peaks in update time correspond to lateral cache updates. The camera

[2]Transform feedback is the recommended mode on SM4.0 GPUs to generate positions and geometry on the fly. Updating the positions using transform feedback is slow and achieves about 17 fps with 4K balls. Fragment shaders are much faster

goes up and merge shifts occur near frame 2200. The triangle rate falls when many distance-levels are used as the terrain access doesn't benefit from its caching scheme. The camera starts to come down around frame 3200.

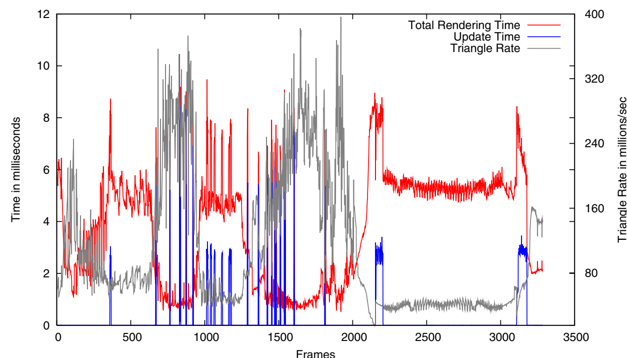

Figure 13. Cache update time, total rendering time, and the triangle rate for a typical flight over the terrain.

The rendering time below 8 milliseconds per frame at all times, with the average around 2.5 ms on the trillion sample terrain under different viewer motions. The system can provide a guaranteed 100 fps rate without the CPU, the GPU, or the bandwidth between them being a bottleneck. The system achieves a rendering rate of upto 350 million triangles per second and an average rate of over 160 MT/s. This remarkable rates are made possible by exploiting the power of the GPU. The CPU load stays between 5-10% even when the viewpoint moves up/down. We ran experiments on Puget Sound data using the geometry clipmap demo provided by Hoppe [1] on the same GPU. Their system renders an average of 300K triangles per frame and obtain an average triangle rate of 100 million triangles per second. Our system renders an average of 450K triangles per frame with a peak triangle rate of 350 MT/s.

7. Conclusions

In this paper, we presented a system for real-time rendering, deformation, editing, and physics computation of large terrains. The representation enables quick rendering and the ability to manipulate the terrain on-line. The GPU plays the key role in representation, rendering, and manipulation of the terrain. The CPU load is kept very low using the geometry template based rendering, second level culling, and terrain manipulation using fragment shaders. We demonstrate fairly sustained frame rates of over 100 fps and triangle rates of upto 350 million.

The primary limitation of our system is the need for the whole terrain to be present on the CPU memory. This limits the size of the largest terrain that can be handled since data cannot be accessed from disks at that rate. However, the terrain on the CPU can be thought of as a cache at an appropriate resolution of the terrain that resides on the disk or over the network. A scheme very similar to what is used for the GPU cache can then be used to manage the data on the CPU at an appropriate resolution. Since the CPU cache will need occasional updates, we can update it with a parallel low priority thread using today's dual core processors. The other limitation concerns the speed limit on the viewer imposed by the GPU cache updating. This will improve as the CPU to GPU bandwidth improves on future GPUs.

Acknowledgments: We gratefully acknowledge the partial financial support of Microsoft Research's Virtual Earth Programme.

References

[1] A. Asirvatham and H. Hoppe. Terrain rendering using gpu-based geometry clipmaps. *GPU Gems 2*, pages 46–53, 2005.

[2] S. Atlan and M. Garland. Interactive multiresolution editing and display of large terrains. *Computer Graphics Forum*, 25(2):211–223, 2006.

[3] P. Cignoni, E. Puppo, and R. Scopigno. Representation and visualization of terrain surfaces at variable resolution. *The Visual Computer*, 13, 1997.

[4] D. Cohen-Or and Y. Levanoni. Temporal continuity of levels of detail in delaunay triangulated terrain. In *IEEE Visualization*, pages 37–42, 1996.

[5] S. Deb, S. Bhattacharjee, S. Patidar, and P. J. Narayanan. Real-time streaming and rendering of terrains. In *ICVGIP '06*, pages 276–288. LNCS 4338, 2006.

[6] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Comput. Graph. Forum*, 18(3):83–94, 1999.

[7] L. D. Floriani, P. Magillo, and E. Puppo. Building and traversing a surface at variable resolution. In *IEEE Visualization*, pages 103–110, 1997.

[8] R. Geiss. *Generating Complex Procedural Terrains Using the GPU*. Addison Wesley, 2007.

[9] Y. He, J. Cremer, and Y. E. Papelis. Real-time extendible-resolution display of on-line dynamic terrain. In *Graphics Interface*, 2002.

[10] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization*, pages 35–42, 1998.

[11] X. Li and J. M. Moshell. Modeling soil: realtime dynamic models for soil slippage and manipulation. In *SIGGRAPH*, pages 361–368, 1993.

[12] F. Losasso and H. Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3):769–776, 2004.

[13] D. Wagner. Terrain geomorphing in the vertex shader. *ShaderX2, Shader Programming Tips and Tricks with DirectX 9, Wordware Publishing*, 2004.