Large-Scale Physics-Based Terrain Editing Using Adaptive Tiles on the GPU

Juraj Vanek, Bedřich Beneš, Adam Herout, Ondřej Šťava

Abstract—Terrain modeling is an important task in digital content creation and physics-based approaches have the potential to simplify it by introducing a higher level of realism. However, most of the existing simulations are hindered by a low level of user control, because they fail on large-scale phenomena, or because they are focused only on the modeling of limited effects. We introduce a new interactive, intuitive, and accessible physics-based framework for digital terrain editing. Our solution is suitable for users involved in digital content authoring (such as game designers, artists, 3D modelers, and digital content providers) and does not assume any in-depth knowledge about physics-based simulations. To address the scalability issues of previous algorithms, we provide an adaptive GPU-amenable solution. Large terrains can be loaded from external sources, generated procedurally, or created manually, and they are edited at interactive framerates on the GPU. We introduce two simplifications that allow us to perform large scale editing. First, the terrain is divided into tiles of different resolutions according to the complexity of the underlying terrain. Second, each tile is stored as a mip-map texture and different levels of detail are used during the physics-based simulation depending on the dynamics of terrain changes. We demonstrate our approach at several examples including the editing of terrains, brushing with hydraulic simulation, and blending terrain patches into an existing terrain. In comparison with nonadaptive computation, by using our approach we can achieve a 50% speedup with a simultaneous 25% savings of memory. Most important, we can process terrain sizes that were not possible to process with previous approaches.

Index Terms—Digital content authoring, large-scale terrain, physics-based simulation, terrain editing, mip-map, hydraulic erosion, GPU.

1 Introduction

Terrain editing is an important element of a digital content creator's workflow. Terrain databases of digital elevation models (DEMs) are freely available, various methods for procedural terrain generation exist, and users can edit the terrains by a variety of tools and software. Intuitive user-accessible physics-based simulations have the potential to improve interactive content creation and authoring for computer animations and the entertainment industry. These simulations can provide an additional dimension of control for terrain modeling. Nowadays, physics-based editing is not a common practice for several reasons. First, simulations usually are not intuitive, are difficult to control, and their final effect is hard to estimate. Second, and more important, only small-scale physics-based interactive editing is usually possible, because computations are time-consuming and require large memory. Attempts to edit large sets of data are usually below interactive frame rates because the methods simply do not scale well.

The first key observation of our approach is that many physics-based simulations are spatially separable and can be done in parallel. In turn, this localizes the editing operations that can be applied only to areas where the simulation is needed. The second key observation is that the frequency of changes over the terrain varies. Areas with many changes can be simulated with higher simulation precision, whereas areas with lower variances require less precision.

Based on these observations, we introduce a new user-friendly, intuitive, and accessible physics-based framework for large-scale digital terrain editing. Our solution is suitable for users involved in digital content authoring (game designers, artists, 3D modelers, and digital content providers, among others) and assume no in-depth knowledge about physics-based simulations. To address the scalability issues, we harness the parallelism of the simulations and provide an adaptive GPU-amenable solution. Terrains of sizes not possible with previous approaches can be interactively edited.

We introduce two simplifications that allow us to perform large-scale editing. First, the input is analyzed and divided into tiles of different resolutions, depending on the complexity of the terrain in each tile. During the editing process the tiles are continually evaluated and, if necessary, resampled to higher or lower resolution. Second, each tile is stored as a mip-map texture. Different levels of detail are used for each tile for physics-based simulation depending on whether the dynamics of the terrain change. The mip-map pyramid allows us to use a different resolution for each pixel while calculating the physics-based simulation. Figure 1 shows an example of large-terrain editing. The upper left image shows the initial terrain in resolution $4k \times 4k$ pixels and

- *J. Vanek and A. Herout, Brno University of Technology, Czech Republic. E-mail: ivanek,herout@fit.vutbr.cz*
- *B.Beneš and O.Šťava Purdue University, USA E-mail: bbenes,ostava@purdue.edu*

three layers of material (approximately 3GB of data), and the lower image shows the corresponding adaptive tile subdivision. The right image shows the same terrain after a surface patch of resolution $1k \times 1k$ is added, running a hydraulic-erosion brush at various areas, and manually creating mountains and valleys. The average time required to compute one simulation step was 23ms, the simulation ran for 10 seconds, and the scene editing took a few seconds.

2 Previous Work

Procedural terrain modeling methods using fractals [1], multi-fractals, and hypertextures [2] can be used to generate arbitrarily sized data sets, but they provide no reasonable user control, and the results may often appear unrealistic. A procedural approach presented by Kelley et al. [3] employs user interaction to sketch ridges and valleys that guide a fractal system, but this approach is ad hoc, is not physically-based, and allow for no terrain editing.

More recently, procedural-based terrain generation approaches have been extended by using example-based modeling techniques that allow for modifying the shape and structure of the terrain to a predefined pattern [4], but example-based methods do not allow small terrain changes and they do not support physics-based editing.

Various software packages for procedural modeling with interactive editing exist, such as Bryce or Terragen, but they usually provide no physics-based editing tools, or they fail to edit large scale phenomena.

Physics-based approaches have been introduced into terrain modeling by water diffusion and thermal weathering in [5] and are later extended in many directions including corrosion and erosion, based on chemical properties of materials in [6]. The most important long-lasting terrain-forming phenomenon is the water and hydraulic erosion was emplyed for 2D cases in [7]. Layered data representation for erosion simulation has been introduced in [8] and a full 3D hydraulic erosion in [9].

Performance of the erosion methods can be improved by off-loading the simulation to a GPU as done by Mei *et al.* [10], [11] and Šťava *et al.* [12] who modeled erosion of both running and still water on multi-layered terrains. These methods provide very good user control for small to medium sized terrains but, because of memory and performance constraints of the GPUs, they are unsuitable for the modeling of large terrains.

Whereas most of the above-described approaches are based on the Eulerian solution of Navier-Stokes equations for fluid dynamics, the Lagrangian solution by means of smoothed particle hydrodynamics was coupled with hydraulic erosion in [13]. This solution can be hindered by rapidly increasing the number of particles, required for the simulation.

Our paper continues with a high-level system overview. Section 4 provides an in-depth description with all details about the tiling generation, resolution, size, and resampling. The same section also describes the simplified physics-based simulation algorithm that runs on an adaptive mip-map pyramid. Section 5 discusses implementation details, and Section 6 describes results of our experiments, performance measurements, optimizations, errors, and GPU memory considerations. The paper concludes with a section that discusses limitations and possible future work.

3 System Overview

Our terrain consists of multiple layers of materials and uses layered data representation introduced in [8]. This approach allows for efficient representation of different materials and transitions (erosion and deposition) from one type to another. The overview of our system is depicted in Figures 2 and 3. The preprocessing step includes data definition and subdivision into tiles of different resolution. The input data can be defined in different ways. Each layer can be generated procedurally, loaded as a single large texture, or it can be mosaicked from various files interactively. In the last mode, the user defines each layer by dragging the input images over the layer and dropping them at a desired location. The input image is then merged with the existing data, using a user-selected blending mode. The image can be added, multiplied, or subtracted from the existing layer. Our input data is not restricted to a rectangular domain, because tiles for some layers may not be present. We call our data structure a virtual (layered) terrain.

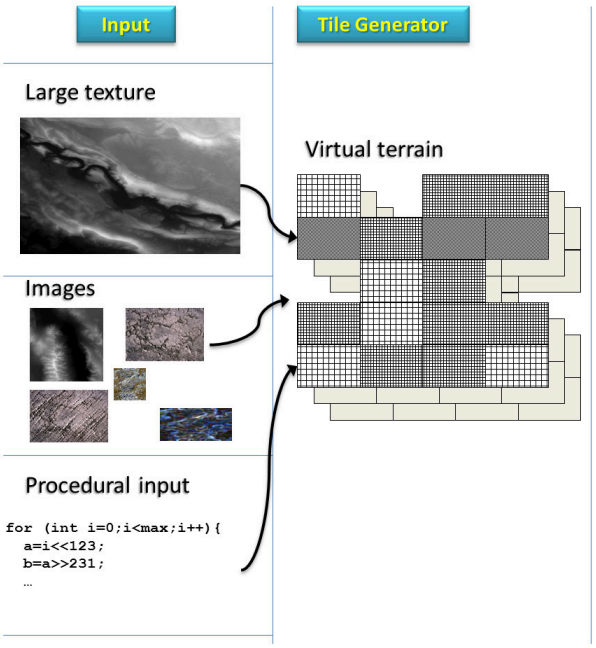

Fig. 2: Data preprocessing. Virtual layered terrain is composed of tiles of different resolutions. Each layer can be loaded as a whole, generated procedurally, or composed from multiple images.

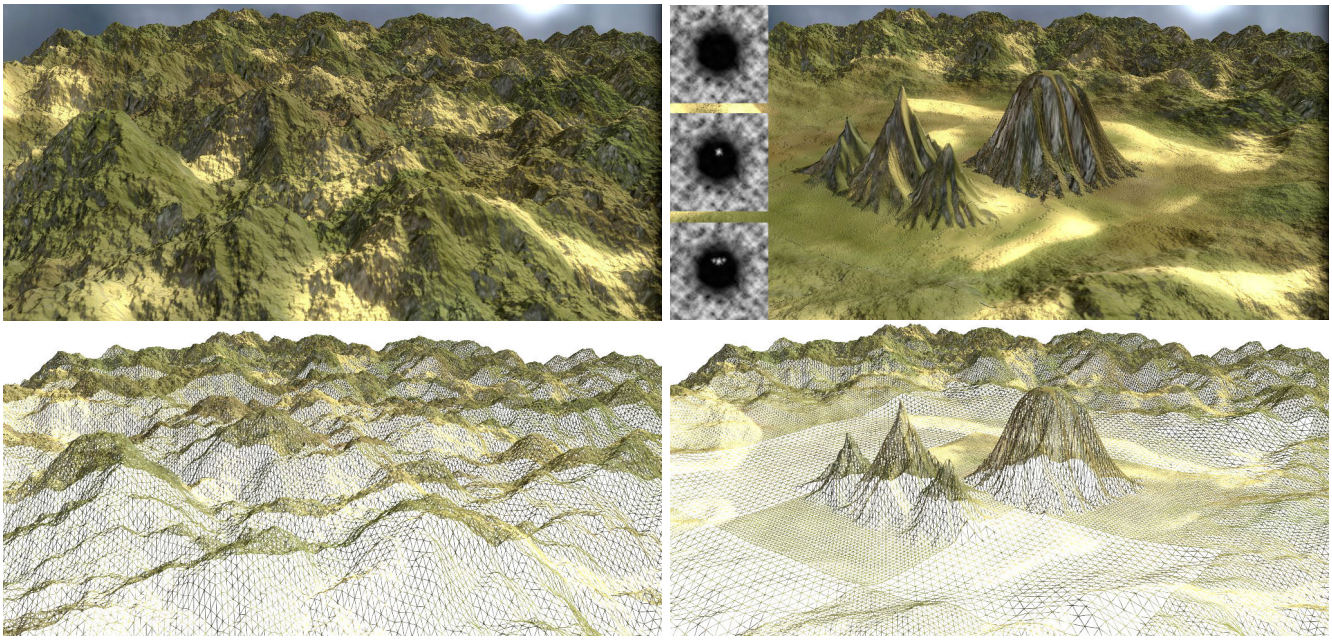

Fig. 1: The upper row shows the input scene in resolution $4k \times 4k$ and nearly 1 GB of data before (left) and after (right) interactive editing. The user added several patches of terrain (shown as successive insets in the upper right image) in resolution $1k \times 1k$, edited some valleys and mountains, and ran hydraulic erosion on certain areas. The overall editing of the scene took 10 seconds and the average runtime of the simulation per frame was 23 ms. The lower row shows the corresponding tiling of the entire scene, where areas with high-altitude variation are represented in higher resolution than areas with small variation are.

When the virtual terrain is defined, it is further analyzed and divided into tiles. All tiles cover the same area of the virtual terrain, but the actual resolution of each tile depends on the data complexity in all underlying layers. Information about tile properties and placement of the tiles is stored in the main memory of the computer, and their data are uploaded and processed on the GPU when needed.

After the preprocessing step, the virtual terrain is ready to be edited, as shown in Figure 3. The user can apply various editing operations, such as smoothing, pulling, and pushing of vertices and parts of the terrain, area selection, copy and paste, etc. Editing mode uses physics-based simulations. We have implemented two physics-based operations: thermal weathering [5] and hydraulic erosion [12]. Each modified tile is periodically evaluated to determine if its resolution should be recalculated by the tile generator. Concurrently, a mip-map pyramid is calculated for each tile. The virtual terrain is rendered at the end.

Our algorithm is implemented with strong GPU support. All simulations and editing operations are implemented as GPU shaders. The tiling scheme allows for an efficient out-of-core simulation. (Whereas this term usually refers to a procedure that does not fit into the main memory and is offloaded from a hard drive, we use it loosely as the simulation that does not fit into the GPU memory and is loaded from the main memory). All affected tiles are processed on the GPU, and the results are loaded back into the main memory only when necessary. The support of frame buffer objects allow for seamless full GPU-supported rendering of the generated terrain with advanced real-time effects, such as HDR rendering, screen space ambient occlusion, parallax mapping, shadows, and refractions for water.

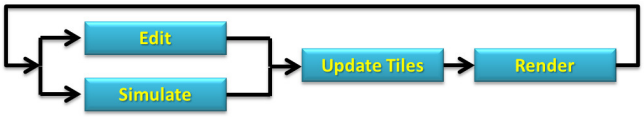

Fig. 3: The interactive terrain is edited using our approach. Each tile can be edited either manually or by using the physics-based simulation. After the terrain changes, the affected tiles are evaluated, and, if necessary, the tile resolution is changed. The entire terrain is visualized in each step.

4 Terrain Tiling

Let's recall that the result of data preprocessing is a set of 2D images (layers) in the main memory of the computer called the virtual terrain, which is subdivided into tiles with a constant size, but varying resolutions.

4.1 Tile Resolution

Each tile resolution must be the power of two. Moreover, it must fall within the user defined range $2^{min}, 2^{min+1}, \ldots, 2^{max}$. In our implementation, the lowest efficient tile resolution was $min = 5$ and the upper

limit was constrained by the size of available main memory. Values greater than $max = 10$ resulted in frequent swapping between the GPU and the main memory, thus slowing down the simulation. Each tile covers the same area, but the actual resolution depends on the complexity of the tile content, which can be seen in Figure 4.

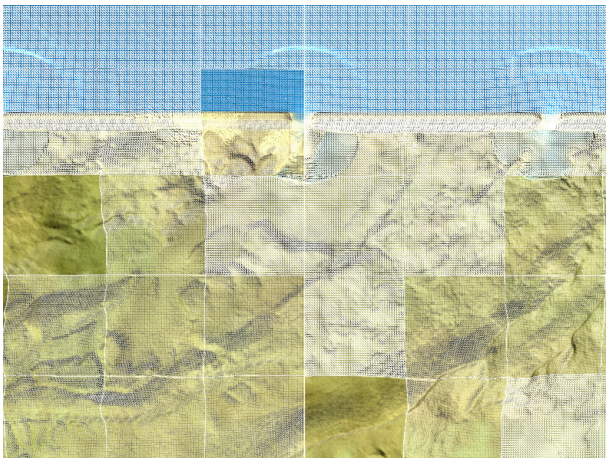

Fig. 4: Each tile covers the same area of the virtual terrain, but its actual resolution depends on the complexity of the underlying layers.

We determine the tile complexity by a simple metric that measures the overall differences of terrain altitudes. First, we find the minimum and the maximum of altitudes of the entire terrain. For each tile, we then use the parallel reduction algorithm on the GPU. The tile is repeatedly scaled-down to the resolution 32×32, and the difference between the minimum and the maximum is found. The difference is then normalized into an interval defined by the minimum and maximum values from the entire terrain. The tile resolution is found by linearly mapping the normalized tile difference onto the selected interval of texture resolutions. Note that each tile consists of various layers of materials; thus the actual calculation is performed several times on each layer. The tile resolution depends on the layer with the highest complexity. This method is efficient, runs on the GPU, and is used not only to preprocess the input data, but also to evaluate the tile changes on the fly.

After a tile has been updated either by user interaction or by physics-based simulation (see Figure 3), its content is evaluated to determine if the texture needs to be resampled to either a higher or a lower resolution. We apply the above-described algorithm for resolution selection on the modified tile and compare the selected resolution with the actual one. If the new resolution is different, the tile is resampled. The global minimum and maximum of the virtual terrain are also updated, so a change of a single tile can cause a resampling of tiles covering different areas.

4.2 Physics-Based Simulation on a Mip-Map

We use two erosion algorithms. The first is thermal weathering, introduced in [5] and later used for layered data [8]. Thermal weathering causes small particles of a material to fall from elevated locations and pile up. The falling is slowed by inner friction of the material and it stops when the so-called talus angle is reached. This angle is about 30^o for sand and is the value we use in our implementation.

The second physics-based simulation used in our system is hydraulic erosion. This is caused by running water and the forces it exerts on the underlying terrain. Various hydraulic erosion algorithms have been introduced [6], [7], [9], [11]. Without loss of generality, we use force-based hydraulic erosion in our system [12]. The force applied to the terrain separates a certain amount of material that is transported in the running water and eventually deposited at a different location when the water slows. The key element of an efficient hydraulic erosion algorithm is the coupling of the erosion/deposition model with the water transportation. We use the pipe model [14], which is an approximation of the solution of the Navier-Stokes equation for fluid simulation applied to a special case of shallow-water transportation. Both thermal weathering and force-based hydraulic erosion are fully implemented on the GPU.

In our simulation, the material on the topmost layer and the water can change locations. The topmost layer is the only layer that is eroded, and the deposited material is also deposited to the topmost layer. It is important to note that the topmost layer need not always be the same. For example, the eroded layer can be a rock and it will be deposited on another location such as sand. All erosion algorithms are material preserving.

Each data point of the tile stores the water level, water flow, and height of each layer of material. These data are efficiently packed into texture on the GPU and accessed via shaders. Several values must be recalculated in each simulation step, and the new values depend on the values from the previous steps:

- Water flow is a vector computed from the water-height differences between the actual and neighboring cells.
- Water height is the actual water level, and it depends on the inflow/outflow from/to neighboring cells.
- Layer composition needs to be changed according to the force-based erosion. Fast flowing water captures sediments from the topmost layer and deposits them elsewhere when the water flow slows.
- The amount of removed material caused by thermal weathering is determined by the angle between neighboring cells.

The tile resolution is determined by the complexity of the terrain, but it does not necessarily reflect the complexity of the flowing water. Because the physics simulation is the most complicated procedure, we use

another spatial subdivision for each tile. We generate a mip-map pyramid of textures, and the hydraulic erosion is calculated at the level that corresponds to the speed of the moving water. Intuitively, slowly moving water exerts smaller forces on the terrain, and the effect can be applied to smaller resolutions. In this way, we trade numerical precision of the physics-based simulation for speed of the application.

Each tile that contains water stores the mip-map pyramid. Let's denote the maximum resolution of tile $D(w, h)$ and its n mip-map levels $D(n)$ where each pixel in a higher level contains averaged data from the four corresponding pixels of the previous level. Theoretically, we could use an arbitrary cascading scheme, but mip-mapping has a great support in GPU hardware.

The actual computing precision of the erosion algorithm is evaluated on a per-pixel basis. To determine the actual level used for each pixel, an *importance map* denoted by $A(w, h)$ is calculated for each tile. The map has the same resolution as the highest level of the pyramid and stores an index to each mip-map level as shown in Figure 5. The value is determined from the maximum value of the water flow normalized over terrain that is mapped to the number of mip-map levels. The value of the importance map is used to select the actual mip-map level from which the hydraulic erosion is calculated.

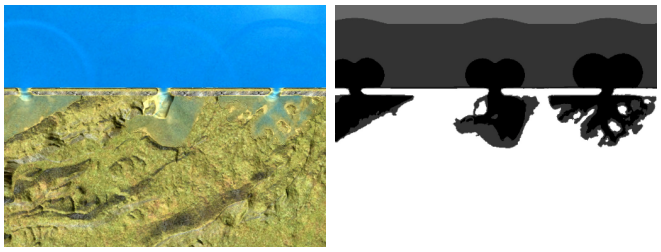

Fig. 5: An example of a terrain (lower part) on the edge of a lake (upper part) with water flowing over the bank. The right figure shows the corresponding importance map.

Accessing the mip-map cascade could present a significant overhead during the calculation; thus the mip-map is used only to determine the resolution at which the erosion is calculated. The actual mip-map is merged into a 2D image denoted by $\hat{D}(\hat{w}, \hat{h}) : \hat{w} = w, \hat{h} = h$ that has the same resolution as the highest level of the pyramid. The merging of different mip-map levels into a single texture is achieved by a successive lookup into different mip-map levels, comparing with the importance texture and broadcasting the values into the four corresponding pixels as schematically shown in Figure 6. Here, only pixels denoted by A-D and E-L are calculated in the highest precision. The other values are calculated at lower resolution and their values are broadcast to multiple pixels. Algorithm 1 describes the hydraulic erosion on the mip-map pyramid.

Algorithm 1 Hydraulic erosion on a mip-map.

Input: Merged data structure $\hat{D}$, importance map A
Output: New mip-map $D^{(N)}$

1: **for** each mip-map level $i = 0$ **to** N **do**
2: **for all** pixels in $\hat{D}$ **do**
3: **if** the pixel has the same importance level $A(x, y) == i$ **then**
4: calculate physics on this level of detail $D^{(i)}(x, y) := \mathrm{DF}(\hat{D}(x, y))$
5: **end if**
6: **end for**
7: **end for**

DF computes hydraulic erosion

The algorithm tests each pixel of the highest level of the pyramid N times, where N is the number of mip-map levels. If the importance value for the given pixel is equal to its value from the merged map (step 3) the physics-based algorithm is executed. When the algorithm ends, the mip-map and the merged map are recalculated.

The speedup of the adaptive calculation is demonstrated by the merged map in Figure 6. The original tile has a resolution of 8×8 pixels; however, only 20 unique values (A-T) are calculated.

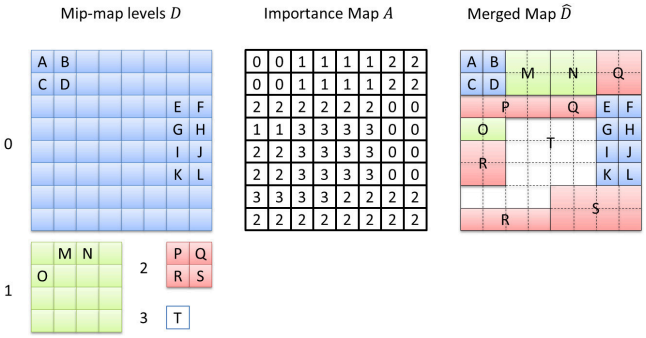

Fig. 6: Merging mip-map levels into the merged map $\hat{D}$ using importance map A. Values from higher levels of the mip-map are broadcast into lower levels, as shown in the upper left quadrant.

4.3 Interactive Editing

Interactive editing exploits the spatial locality of changes invoked by the terrain modifications. When using a brush to edit terrain, we first detect the affected tiles; then we transfer them on the GPU, and edit them on the GPU. The active tile lookup is done quickly, because each tile has its address derived from the terrain origin and information about the cursor position. The accompanying video demonstrates some of the editing operations, such as pulling and pushing some areas of the terrain, applying terrain patches, and smoothing. One interesting operation is editing the layer that is under another one that allows for creation of a rising objects under another

layer. The thermal weathering automatically creates the effect of falling sand.

An editing operation can change the height differences in terrain, but the automated mechanism for determining the tile resolution introduced in Section 4 will immediately determine the resolution and resample it in real-time.

5 Implementation

The algorithm is implemented in C++ and uses OpenGL 4.0 and GLSL. All tests were performed on a desktop with Windows7 64 bits, Intel i7 920 clocked at 2.67GHz, and NVIDIA GTX 480 with 1.5GB of memory. We used GLSL because it has a great support for hardware mip-mapping as it allows fetching data from different levels. GLSL, in its latest version 400, also includes functions allowing a single instruction fetching of surrounding texels (called texture gathering). Moreover, GLSL is supported by all major graphic chip manufacturers.

Our algorithm uses several 2D data structures: water level, sediment level, water flow, and terrain layers. Water flow and terrain layers are implemented as four-channel 32-bit floating point textures; water and sediment levels are packed into a single two-channel texture.

The importance map adds one single-channel, and the original data textures are automatically converted into mip-maps on the GPU. Mip-map merging is performed in separate data structures which again represent water and sediment heights, flow, and terrain. The importance map has additional memory requirements, and there is also additional work required to calculate cascade levels and to merge them into a single data texture. Overall, our method would need 50% more memory if tiling were not used. However, in practical examples shown in this paper (Table 1), the memory saving was about 30%.

The Eulerian approach to fluid simulation is suitable for GPU implementation because it can fully utilize the parallel computing power of GPU. All cells can be calculated independently on each other because they use values only from neighboring cells created in previous simulation steps. Care must be paid during the rendering loop because writing and reading from the same data texture at the same time is not allowed. This is solved by first rendering into a mip-map chain and then merging into a final image. Merged data are then used in the next step as read-only texture for mip-map generation. Implementation uses frame buffer objects (FBO) and fragment shader to perform all screen-space calculations. Writing into data texture is performed by attaching the texture to FBO and drawing it as a full screen quad with the shader activated. Because the algorithm uses a lot of dynamic branching with indexed arrays, a fast GPU with support for an OpenGL version higher than 3 is needed. On GPUs with an OpenGL 4 support, it is possible to use fast instructions to gather all surrounding fragments around active fragments

Rendering is also done using OpenGL 4 with a programmable graphic pipeline. Each tile contains a rectangular mesh with the number of polygons based on the tile resolution. The mesh is displaced by a compound height of terrain and water layers in vertex shader. The result is then visualized in fragment shader. Each terrain layer has different color textures, and the colors are blended together to create smooth transitions between layers according to each layer of thickness. Water is displayed on top of the terrain and it is also partially blended with the terrain layers.

To further improve the visual quality, we use advanced real-time rendering methods, such as high dynamic range lighting, screen-based ambient occlusion, water reflection and refraction, dynamic shadows, surface bump, and parallax mapping.

6 Results and Discussion

The first result in Figure 1 shows an example of interactive editing. The user initially created procedural terrain from a Perlin noise in three layers of material each in resolution $4k \times 4k$. This model was loaded into the system and tiled. The entire model has a theoretical size of nearly 3 GB, but the tiled size was only 1 GB. The user then used three images in $1k \times 1k$ resolution (showed as insets of the second image) and added them into the terrain using the blending mode. The system automatically recalculated the resolution of the tiles, as seen in the second row. Overall, this editing took less than 15 seconds, and the tile recalculation as well as the thermal weathering took about 25 ms.

The second result in Figure 7 shows a real data digital elevation model of the Grand Canyon that was artificially flooded by a strong water source. The simulation time was 100 ms per frame, and the terrain resolution was $8k \times 4k$.

The next scene in Figure 8, shows an example of interactive physics-based erosion. A scene with mountains from different materials has been eroded by a water source manually located over them, as indicated by the blue circle. The simulation time was 25ms per frame, and the terrain resolution was $4k \times 4k$. The example shows how the different materials erode with different speed and how the sand is being deposited under the mountains.

Figure 10 shows a very large scene that did not fit into the memory of the GPU. The original scene of $12k \times 12k$ was tiled into 12×12 tiles with maximum resolution $1k \times 1k$. The scene used about 2.5 GB of memory. The average simulation time of the scene was 43 ms per frame. The scene represented on the GPU used maximum of 1.5 GB and ran at 12 fps with both rendering and simulation. The inset on the right shows a detail of the image from the left.

6.1 Tile Size and GPU Memory Considerations

One of the aspects significantly affecting performance is the actual size and, as a consequence, the total number

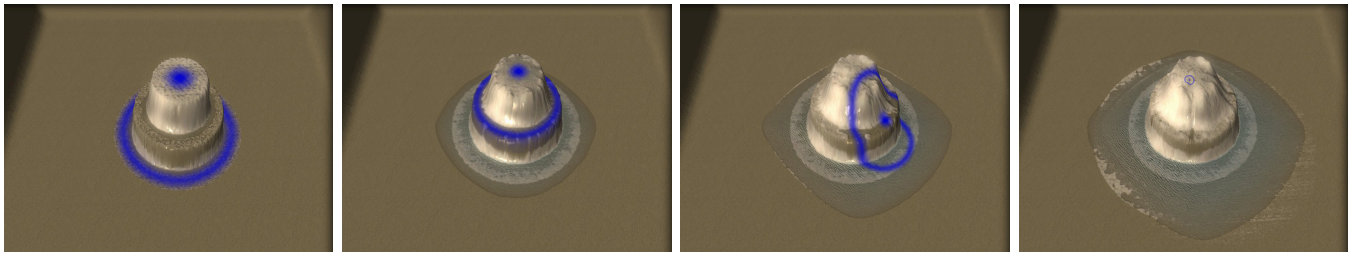

Fig. 8: Example of interactive editing using a physics-based brush (displayed as a blue circle) with a localized rain. The object is made of rock with soil on its top.

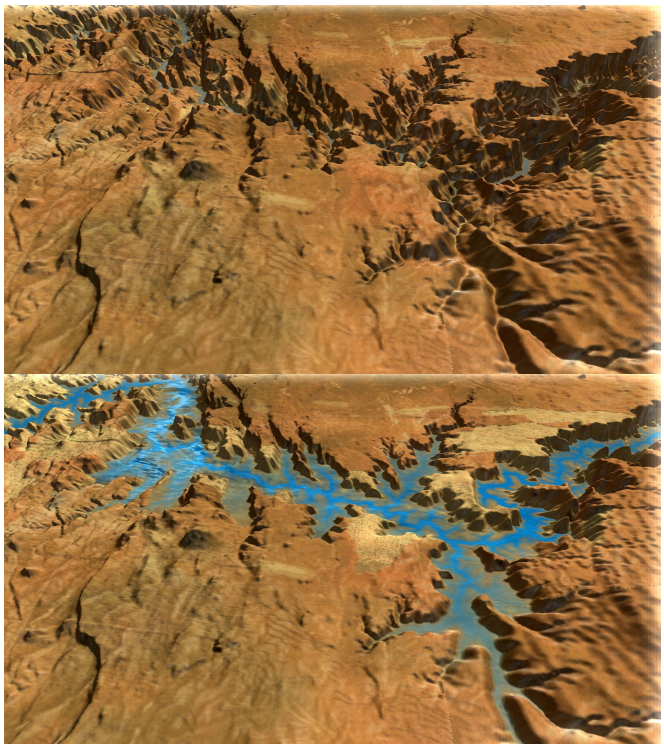

Fig. 7: Digital elevation map of the Grand Canyon in resolution $8k \times 4k$ pixels (up) was used as an input of the erosion simulation (down).

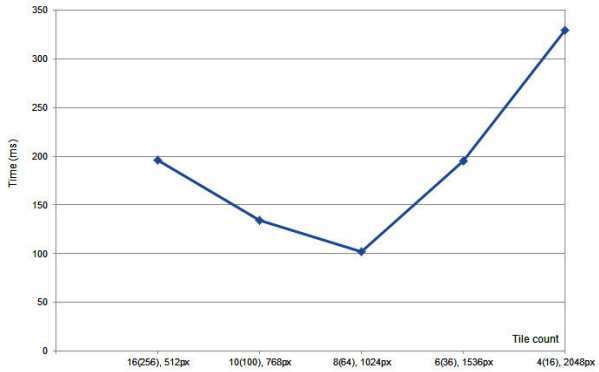

Fig. 9: Large scene simulation as the function of tile size. Large numbers of small tiles have performance overhead and small numbers of large tiles do not utilize the adaptability efficiently. A reasonable size of the tile is about 1-5% of the terrain area's total size.

of tiles. Intuitively, a large number of tiles will cause a performance hit because of the tile synchronization overhead. A small number will obfuscate the performance gain of the importance map and varying tile resolution. Because the actual effect of the number of tiles can be affected by various aspects, we have created a benchmark that determines the optimal number of tiles. As can be seen on the shape of the simulation time in Figure 9, the measurement confirms the intuition. The best performance is achieved for tiles covering approximately 1-5% of the area of the virtual terrain.

There is a hardware limit of the GPU memory. Because most calculations are done on the GPU, storing as many tiles as possible on the GPU is beneficial. We could fit up to 256 tiles with small resolution (128px) or a few large tiles (1024px) into the GPU memory. However, the application is fully functioning even when processing much larger scenes that do not fit into the GPU memory, because the operating system driver will swap the least-recently used memory pages into the main computer memory.

Tiles are synchronized directly on the GPU. If a pixel lies on the border of a tile, the neighboring cells are selected from the appropriate tiles because all tiles can access textures from their surrounding tiles. This process guarantees seamless transition of water flow and accompanied quantities among tiles of the same resolution. When neighboring tiles have different resolution, hardware-level linear interpolation is used when fetching values that introduce a minor interpolation error. However, we have not observed any significant impact of this error in our experiments.

Table 1 shows performance and memory requirements of the scenes from this paper. All scenes were rendered in the non-adaptive mode in the maximum possible resolution of 928MB of memory and the timing was a nearly constant 34ms. Our adaptive method shows an average speedup of 1.46 and average memory savings of 75%

6.2 Simulation Error

To bring the physics-based terrain editing to interactive frame rates, we introduce numerical simplifications at

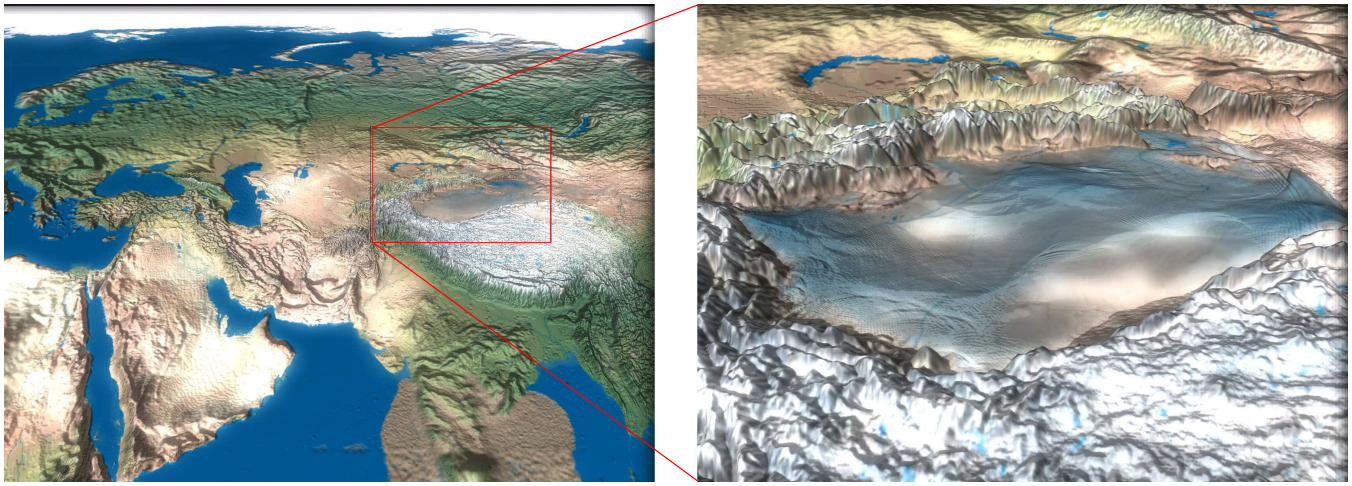

Fig. 10: Very large scene (2.5 GB) of resolution $12k \times 12k$ occupies 1.5 GB of the GPU memory and is eroded and rendered on the GPU at 12 fps.

Scene	Time [ms]	Speedup	Memory [MB]	Savings [%]
Figure 1 L	23.6	1.44	784	78
Figure 1 R	21.1	1.61	698	75
Figure 4	25.4	1.34	700	75
Figure 7	22.7	1.45	770	82

TABLE 1: Time and memory requirements of scenes from the paper. The timing for all scenes in non-adaptive mode was 34 ms and the memory requirements were 928MB of video RAM. The table shows a runtime of the adaptive method and its speedup, as well as the memory requirements and memory savings. The scene from Figure 10 is not included, because it can be edited only in the adaptive mode

many different levels. The sources of the possible errors are: different tile resolution, evaluation of the hydraulic erosion at varying levels of detail, and conversion of data from different levels of resolution (merging between mip-maps and between neighboring tiles in different resolution). It would be quite difficult to track the effect of all these simplifications and to provide their in-depth comparison. However, it is important to note that this approach is intended for interactive editing and not for physically precise simulations. The pipe model used for the shallow-water simulation itself is not an exact physical simulation. Moreover, fluid simulation is a dynamic system and is extremely sensitive to initial conditions. A small change in the initial conditions will cause the same system to significantly diverge in the solution after a few steps even for physically exact simulations that make the comparison even more difficult.

We try to make all possible attempts to ensure that the simplifications and inducted inaccuracies will create no visually distracting errors. Merging between different levels, for example, can cause oscillations in the water simulation. Linear interpolation between different levels smoothes visual artifacts and is more visually plausible than faster nearest-neighbor interpolation. Figure 11 (and the accompanying video) shows sequence of a large scene with water erosion simulation. Both simulations run for 550 frames and there is no significant visual difference.

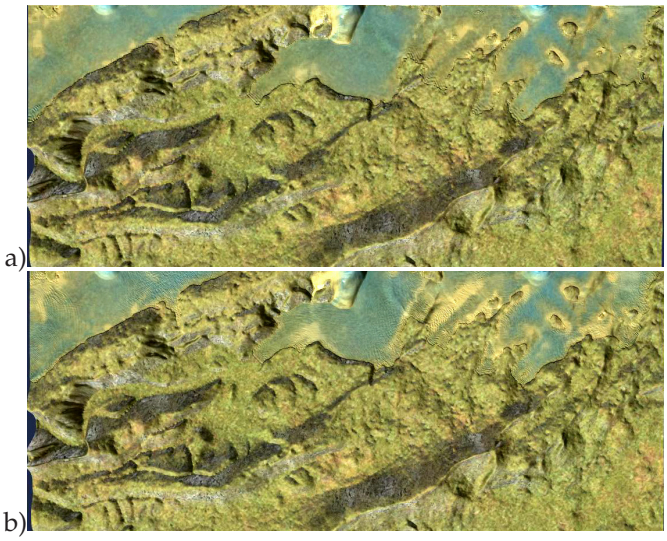

Fig. 11: Comparison of a) the adaptive version with b) full erosion simulation. Both simulations were run for 550 frames to make sure the water had enough time to propagate through the scene. There were no significant visual differences.

Visual artifacts can appear on tile borders when adjacent tiles have different resolution. Though data synchronization errors can be small, visual artifacts are caused because each tile uses its own polygonal mesh, and there can be visible cracks between two meshes with different mesh density. This problem could be removed by putting all tiles on single large mesh and using the appropriate terrain LOD algorithm. Although potentially problematic, especially for neighboring tiles with significantly different resolution, the terrain tiling error was minimal in our experiments.

7 Conclusion

We have introduced an intuitive physics-based framework for digital content authoring (game designers, artists, 3D modelers, and digital content providers, among others). To address the scalability issues of previous algorithms, we provided an adaptive GPU-oriented solution. Very large terrain (tens of GB of data) can be loaded from external sources, generated procedurally, and created manually, and they are edited at interactive frame rates. We introduced two simplifications that allow for large-scale editing. First, the terrain is divided into tiles of different resolutions depending on the amount of detail. Second, each tile is stored as a mip-map texture, and different levels of detail are used during the physics-based simulation, depending on the dynamics of the water flowing over the terrain. We demonstrate our approach using several examples including large terrain editing, brushing terrains with hydraulic simulation, and blending terrain patches into an existing terrain. Using our approach, we achieve a 50% speedup comparing to non-adaptive computation, and we can process terrains of sizes that were not possible with previous approaches.

However, this approach has several notable issues and potential pitfalls. The mip-map subdivision will be rendered ineffective for large water dynamics scenes such as rain. The tile resolution subdivision will be ineffective for white noise scenes or other scenes with high-frequency information equally spatially distributed. Another issue is the limit of the GPU memory. When the simulation exceeds the amount of available memory on the GPU the driver starts swapping memory pages from the GPU with the main memory which has a performance penalty. In-house memory management, such as per-tile LRU cache, could address this problem and could allow the editing of larger datasets. Last but not least, a good error evaluation and analysis could improve algorithm robustness.

There are many possible avenues for future work. As mentioned above, a better memory management would be useful. Similarly, our current implementation resamples all tiles that need resampling immediately, and that is costly. It would be possible to implement a priority queue that would process only the tiles that are being edited, or that have significant visual importance. We are convinced that our method is a framework that would allow not only the incorporation of different physics-based methods, but also new editing techniques.

Acknowledgments

We would like to thank NVIDIA for providing graphics hardware. This work has been supported by NSF IIS-0964302, NSF OCI-0753116 Integrating Behavioral, Geometrical and Graphical Modeling to Simulate and Visualize Urban Areas, by the research program LC-06008 (Center for Computer Graphics), and by the research plan MSM0021630528.

References

[1] B. B. Mandelbrot, *The Fractal Geometry of Nature*. W.H. Freeman and Company, 1983.

[2] K. Perlin and E. M. Hoffert, "Hypertexture," in *Proc. of SIGGRAPH '89*, 1989, pp. 253–262.

[3] A. D. Kelley, M. C. Malin, and G. M. Nielson, "Terrain simulation using a model of stream erosion," in *Proc. of SIGGRAPH '88*, 1988, pp. 263–268.

[4] H. Zhou, J. Sun, G. Turk, and J. M. Rehg, "Terrain synthesis from digital elevation models," *IEEE Trans. Visual. Comp. Graph.*, vol. 13, no. 4, pp. 834–848, 2007.

[5] F. K. Musgrave, C. E. Kolb, and R. S. Mace, "The synthesis and rendering of eroded fractal terrains," in *Proc. of SIGGRAPH '89*, 1989, pp. 41–50.

[6] C. Wojtan, M. Carlson, P. J. Mucha, and G. Turk, "Animating corrosion and erosion," in *Eurographics Workshop on Natural Phenomena*, 2007.

[7] N. Chiba, K. Muraoka, and K. Fujita, "An erosion model based on velocity fields for the visual simulation of mountain scenery," *The Journal of Visualization and Computer Animation*, vol. 9, no. 4, pp. 185–194, 1998.

[8] B. Beneš and R. Forsbach, "Layered data representation for visual simulation of terrain erosion," in *SCCG '01: Proc. of the 17th Spring conference on Computer graphics*, 2001, p. 80.

[9] B. Beneš, V. Těšínský, J. Hornyš, and S. K. Bhatia, "Hydraulic erosion," *Computer Animation and Virtual Worlds*, vol. 17, no. 2, pp. 99–108, 2006.

[10] N. H. Anh, A. Sourin, and P. Aswani, "Physically based hydraulic erosion simulation on graphics processing unit," in *Proc. of GRAPHITE '07*, 2007.

[11] X. Mei, P. Decaudin, and B.-G. Hu, "Fast hydraulic erosion simulation and visualization on GPU," in *Proc. of Pacific Graphics*, 2007, pp. 47–56.

[12] O. Šťava, B. Beneš, M. Brisbin, and J. Křivánek, "Interactive terrain modeling using hydraulic erosion," in *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '08. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 201–210.

[13] P. Krištof, B. Beneš, J. Křivánek, and O. Šťava, "Hydraulic erosion using smoothed particle hydrodynamics," *Computer Graphics Forum (Proceedings of Eurographics 2009)*, vol. 28, no. 2, mar 2009.

[14] N. Holmberg and B. C. Wünsche, "Efficient modeling and rendering of turbulent water over natural terrain," in *Proc. of GRAPHITE '04*, 2004, pp. 15–22.