Large-Scale Physics-Based Terrain Editing

Juraj Vanek and Bedřich Beneš ■ *Purdue University*

Adam Herout ■ *Brno University of Technology*

Ondřej Šťava ■ *Purdue University*

Terrain editing is an important element of a digital-content creator's workflow. Terrain databases of digital elevation models are freely available, various approaches for procedural terrain generation exist, and users can edit terrains with a variety of tools and software. (For more background on terrain editing, see the sidebar.) Easy-to-use physics-based simulations could potentially improve interactive-content creation and authoring for computer animation and the entertainment industry, providing an additional dimension of control for terrain modeling.

But physics-based editing isn't common practice, for two main reasons. First, simulations usually are unintuitive, difficult to control, and unpredictable. Second, and more important, interactive editing is usually possible only at a small scale because computations are time-consuming and require much memory. Attempts to edit large datasets are usually below interactive frame rates because the methods simply don't scale well.

However, many physics-based simulations are spatially separable and can execute in parallel. This localizes editing operations—users can apply them only to areas needing simulation. Additionally, the frequency of changes varies over the terrain, so areas with many changes can be simulated with higher precision and areas with less variance require less precision.

On the basis of these observations, we've developed a physics-based system for large-scale terrain editing. It's user friendly, intuitive, accessible, suitable for digital-content authors (such as game designers, artists, and 3D modelers), and assumes no in-depth knowledge of physics-based simulations. To address scalability issues, we harness the simulations' parallelism and provide an adaptive GPU-amenable solution. With our approach, users can interactively edit terrain sizes that they couldn't with previous approaches.

Most terrain modelers fail on large-scale phenomena and focus only on limited effects. An intuitive physics-based system can process terrain sizes that weren't possible with previous approaches.

System Overview

Our terrain consists of multiple layers of materials and uses a layered data representation.[1] This allows efficient representation of different materials and their erosion and deposition.

In our system, the preprocessing (see Figure 1) 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 mosaicked interactively from various files. In the last mode, the user defines each layer by dragging the input images over the layer and dropping them at a desired location. A user-selected blending mode then merges the input image with the existing data. The image can be added, multiplied, or subtracted from the existing layer. Our input data isn't restricted to a rectangular domain because tiles for some layers might not be present. We call our data structure *virtual layered terrain*.

After the system defines the virtual terrain, it

Further Reading in Terrain Modeling

Procedural terrain-modeling approaches employing fractals,[1] multifractals, and hypertextures[2] can be used to generate arbitrarily sized datasets. However, they provide no reasonable user control, and the results might often appear unrealistic. One procedural approach employs user interaction to sketch ridges and valleys that guide a fractal system,[3] but this approach is ad hoc, isn't physically based, and doesn't allow terrain editing.

Researchers have extended other procedural approaches by using example-based modeling to modify the terrain's shape and structure to a predefined pattern.[4] However, such approaches don't allow small terrain changes or support physics-based editing.

The available software packages (such as Bryce or Terragen) usually provide no physics-based editing tools or fail to edit large-scale phenomena.

F. Kenton Musgrave and his colleagues introduced physics-based approaches to terrain modeling by water diffusion and thermal weathering.[5] Chris Wojtan and his colleagues extended this in many directions, including corrosion and erosion, on the basis of materials' chemical properties.[6]

Norishige Chiba and his colleagues performed 2D simulations of hydraulic erosion, the most important long-lasting terrain-forming phenomenon.[7]

Bedřich Beneš and Rafael Forsbach introduced layered data representation for erosion simulation.[8] Later, Beneš and his colleagues introduced a full 3D simulation of hydraulic erosion.[9]

Xing Mei and his colleagues improved erosion simulation by offloading it to the GPU.[10] Ondřej Šťava and his colleagues modeled erosion of both running and still water on multilayered terrains.[11] These two approaches provide good user control for small to medium terrains, but because of GPU memory and performance constraints, they're unsuitable for large terrains.

Whereas most of the these approaches are based on the Eulerian solution of Navier-Stokes equations for fluid dynamics, Peter Krištof and his colleagues used the Lagrangian solution by means of smoothed-particle hydrodynamics coupled with hydraulic erosion.[12] However, rapid increases in the number of particles required for the simulation can hinder this solution.

References

1. B.B. Mandelbrot, *The Fractal Geometry of Nature*, W.H. Freeman, 1982.
2. K. Perlin and E.M. Hoffert, "Hypertexture," *Proc. Siggraph*, ACM Press, 1989, pp. 253–262.
3. A.D. Kelley, M.C. Malin, and G.M. Nielson, "Terrain Simulation Using a Model of Stream Erosion," *Proc. Siggraph*, ACM Press, 1988, pp. 263–268.
4. H. Zhou et al., "Terrain Synthesis from Digital Elevation Models," *IEEE Trans. Visualization and Computer Graphics*, vol. 13, no. 4, 2007, pp. 834–848.
5. F.K. Musgrave, C.E. Kolb, and R.S. Mace, "The Synthesis and Rendering of Eroded Fractal Terrains," *ACM Siggraph Computer Graphics*, vol. 23, no. 3, 1989, pp. 41–50.
6. C. Wojtan et al., "Animating Corrosion and Erosion," *Proc. Eurographics Workshop Natural Phenomena*, Eurographics Assoc., 2007, pp. 15–22.
7. N. Chiba, K. Muraoka, and K. Fujita, "An Erosion Model Based on Velocity Fields for the Visual Simulation of Mountain Scenery," *J. Visualization and Computer Animation*, vol. 9, no. 4, 1998, pp. 185–194.
8. B. Beneš and R. Forsbach, "Layered Data Representation for Visual Simulation of Terrain Erosion," *Proc. 17th Spring Conf. Computer Graphics* (SCCG 01), IEEE CS Press, 2001, pp. 80–86.
9. B. Beneš et al., "Hydraulic Erosion," *Computer Animation and Virtual Worlds*, vol. 17, no. 2, 2006, pp. 99–108.
10. X. Mei, P. Decaudin, and B.-G. Hu, "Fast Hydraulic Erosion Simulation and Visualization on GPU," *Proc. 15th Pacific Conf. Computer Graphics* (PG 07), IEEE CS Press, 2007, pp. 47–56.
11. O. Šťava et al., "Interactive Terrain Modeling Using Hydraulic Erosion," *Proc. Eurographics/ACM Siggraph Symp. Computer Animation* (SCA 08), Eurographics Assoc., 2008, pp. 201–210.
12. P. Krištof et al., "Hydraulic Erosion Using Smoothed Particle Hydrodynamics," *Computer Graphics Forum*, vol. 28, no. 2, 2009, pp. 219–228.

further analyzes and divides the terrain into tiles. The computer's main memory stores information about tile properties and placement, and the system uploads and processes this data on the GPU as needed.

After preprocessing, the virtual terrain is ready for editing (see Figure 2). Users can apply operations such as smoothing, pulling, and pushing vertices and parts of the terrain; selecting areas; and copying and pasting. Editing mode uses physics-based simulation. We've implemented two physics-based operations: thermal weathering and hydraulic erosion (for more information, see the sidebar). The system periodically evaluates each modified tile to determine whether the tile generator should recalculate its resolution. Concurrently, it calculates a mip-map pyramid for each tile. Finally, the system renders the virtual terrain.

Our system includes strong GPU support. It implements all simulations and editing operations as GPU shaders. The tiling scheme allows for efficient

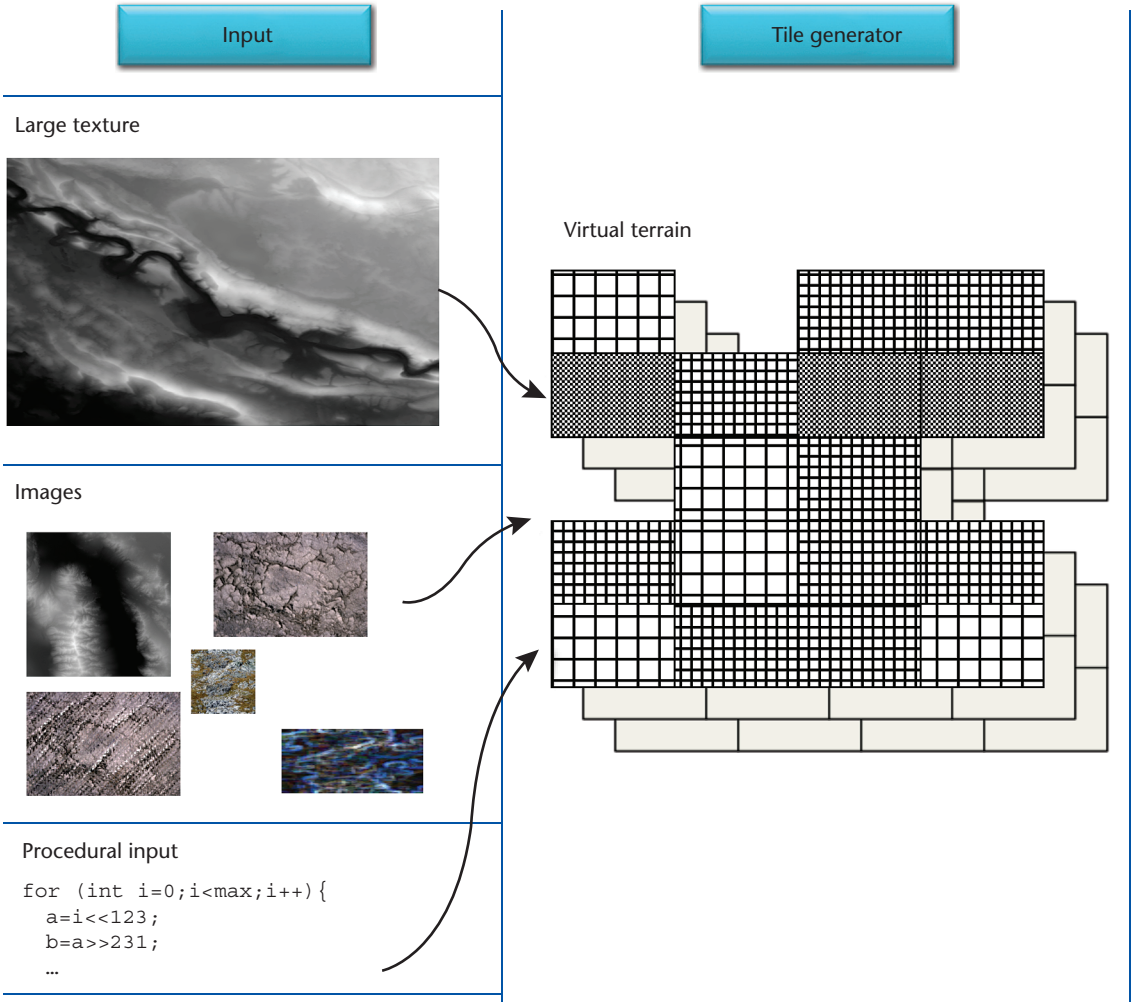


Figure 1. Data preprocessing. Virtual layered terrain consists of tiles of different resolutions. Each layer can be loaded as a whole, generated procedurally, or composed from multiple images.

out-of-core simulation. (This term usually refers to a procedure that doesn't fit in the main memory and is offloaded from a hard drive. We use it loosely to describe a simulation that doesn't fit into GPU memory and is loaded from the main memory.) The system processes all affected tiles on the GPU; it loads the results back into the main memory only when necessary. Support for frame buffer objects allows for seamless full GPU-supported rendering of the generated terrain with advanced real-time effects, such as high-dynamic-range rendering, screen space ambient occlusion, parallax mapping, shadows, and refractions for water.

Terrain Tiling

Here, we examine more closely how our system divides terrain into tiles.

Tile Resolution

Each tile resolution must be to the power of two. Moreover, it must fall within the user-defined range 2^{min}, 2^{min+1}, ..., 2^{max}. In our implementation, the lowest efficient tile resolution was min = 5; the upper limit was constrained by the available main memory. Values greater than max = 10 resulted in frequent swapping between the GPU and the main memory, thus slowing the simulation. Although each tile covers the same area, the actual resolution depends on the tile's complexity (see Figure 3).

We determine tile complexity by a metric that measures the overall differences of terrain altitudes. First, we find the entire terrain's minimum and maximum altitudes. For each tile, we then perform parallel reduction on the GPU. We successively scale the tile down to a resolution of 32 × 32 pixels and find the difference between the tile's minimum and maximum altitudes. We then

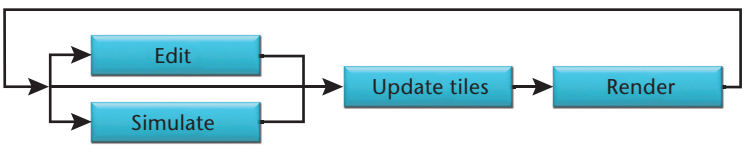


Figure 2. Terrain editing. Users can edit each tile manually or through the physics-based simulation. After the terrain changes, the system evaluates the affected tiles and, if necessary, changes the tile resolution. The entire terrain is visualized in each step.

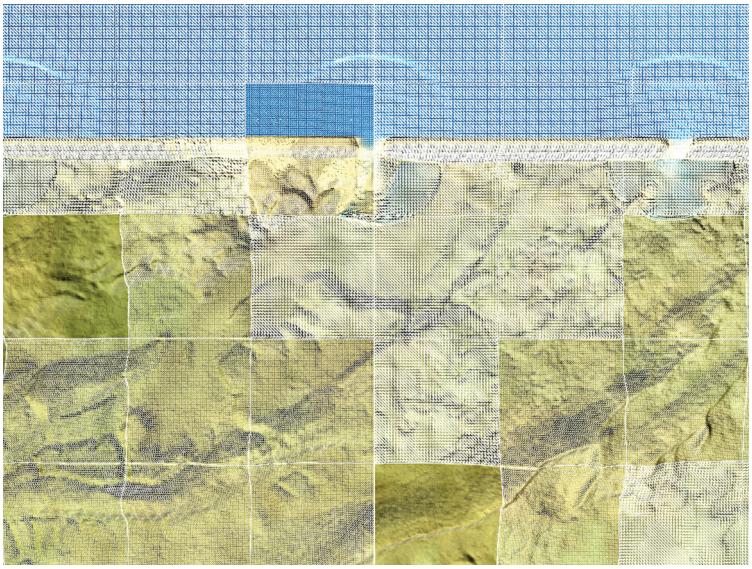


Figure 3. Terrain tiles. Each tile covers the same area of the virtual terrain, but its resolution depends on the underlying layers' complexity.

normalize the difference into an interval defined by the minimum and maximum values from the entire terrain.

To find the tile resolution, we linearly map the normalized difference onto the selected interval of texture resolutions. Because each tile consists of various layers of materials, the calculation executes several times on each layer. The tile resolution depends on the most complex layer. This method is efficient, runs on the GPU, preprocesses the input data, and evaluates tile changes on the fly.

After a tile has been updated, either by user interaction or physics-based simulation (see Figure 2), we evaluate its content to determine whether to resample the texture to a higher or lower resolution. We apply our algorithm for resolution selection on the modified tile and compare the selected resolution with the actual one. If the actual resolution is different, we resample the tile. We also update the virtual terrain's global minimum and maximum, so a change to a single tile can cause a resampling of tiles covering different areas.

Physics-Based Simulation on a Mip-Map

We use two erosion algorithms; both are fully implemented on the GPU. The first is for thermal weathering, which causes small particles of a material to fall from elevated locations and pile up. The material's inner friction slows the falling, which stops when the piled material reaches the *talus angle*. This angle is approximately 30 degrees for sand, which is the value we use in our implementation.

The second algorithm is for hydraulic erosion, which is caused by running water and the forces it exerts on the terrain. Various hydraulic-erosion algorithms exist (for more information, see the sidebar); our system uses force-based hydraulic erosion. The force applied to the terrain separates a certain amount of material that's 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-and-deposition model with the water transportation. We use the pipe model, which is an approximation of the solution of the Navier-Stokes equation for fluid simulation applied to a special case of shallow-water transportation.[2]

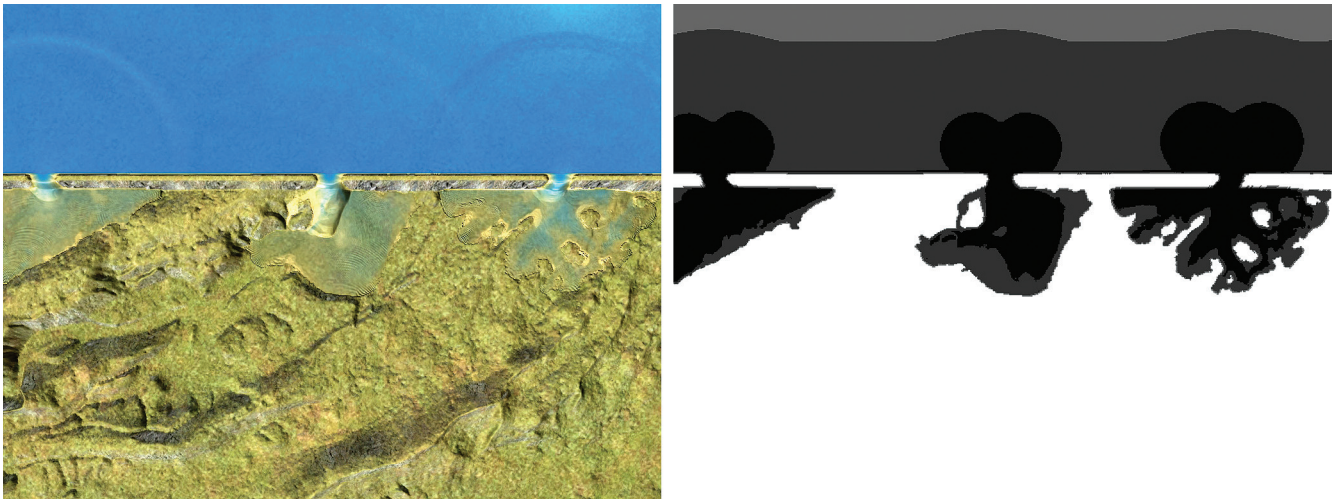
In our simulation, the material on the topmost layer and the water can change locations. The topmost layer is the only one that's eroded, and the deposited material is also deposited to the topmost layer. The topmost layer needn't always be the same. For example, an eroded layer of rock might be deposited on sand. Both 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; the new values depend on the values from the previous steps:

- Water flow is a vector computed from the water-height differences between the selected cell and the neighboring cells.
- Water height is the actual water level; it depends on the inflow and outflow from and to neighboring cells.
- Layer composition must change 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 angle between neighboring cells determines the amount of removed material due to thermal weathering.

Although the tile resolution reflects the terrain's complexity, it doesn't necessarily reflect the flowing water's complexity. 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 calculate the hydraulic erosion at the level corresponding to the moving water's speed. Intuitively, slow water exerts smaller forces on the terrain; we can apply this effect to smaller resolutions. In this way, we trade the physics-based simulation's numerical precision for application speed.

Each tile that contains water stores the mip-map



(a) (b)

Figure 4. (a) Terrain on the edge of a lake, with water flowing over the bank. (b) The corresponding importance map. This is used to determine the precision used for the calculations.

pyramid. We denote the maximum resolution of tile D as $D(w, h)$, where w represents width and h represents height, and its n mip-map levels as $D(n)$, where each pixel in a higher level contains averaged data from the previous level's four corresponding pixels. Theoretically, we could use an arbitrary cascading scheme, but mip-mapping works particularly well on GPUs.

We evaluate the hydraulic-erosion algorithm's precision on a per-pixel basis. To determine each pixel's level, we calculate an importance map, $A(w, h)$, for each tile. The map has the same resolution as the pyramid's highest level and stores an index to each mip-map level (see Figure 4). We determine the map's value from the maximum value of the water flow normalized over terrain. The maximum is then mapped to the number of mip-map levels. We use the importance map's value to select the actual mip-map level, from which we calculate the hydraulic erosion.

Accessing the mip-map cascade could produce significant calculation overhead. So, we use the mip-map only to determine the resolution at which to calculate the erosion. We merge the mip-map 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 pyramid's highest level. We merge the mip-map levels into a single texture by a successive lookup into different mip-map levels, comparing the actual value with the importance texture and broadcasting the values $\hat{D}(\hat{w}, \hat{h})$ into the four corresponding pixels (see Figure 5). In Figure 5, we calculate only pixels A through D and E through L at the highest precision. We calculate the other values at lower resolution and broadcast their values to multiple pixels. Figure 6 shows the pseudocode for hydraulic erosion on the mip-map pyramid.

The algorithm tests each pixel of the pyramid's highest level N times, where N is the number of mip-map levels. If a pixel's importance value is equal to its value from the merged map (step 3 in Figure 6), the physics-based erosion algorithm executes. When the algorithm ends, the system recalculates the mip-map and merged map.

We can infer the adaptive calculation's speedup from Figure 5c. The original tile has a resolution of 8 × 8 pixels; however, we calculate only 20 unique values (A–T).

Interactive Editing

Interactive editing exploits the spatial locality of changes invoked by the terrain modifications. When the user employs a brush tool to edit terrain, the system detects the affected tiles, transfers them to the GPU, and applies the editing operation. The active-tile lookup occurs quickly because each tile's address is derived from the terrain origin and information about the cursor position.

If an editing operation changes the height differences in terrain, the automated mechanism for determining tile resolution will immediately determine the resolution and resample it in real time.

Implementation

We performed all tests on a desktop PC with Windows 7 (with 64 bits, a 2.67-GHz Intel i7 920, and an Nvidia GTX 480), using 1.5 Gbytes of memory. We implemented our system in C++; it uses OpenGL 4.0 and the OpenGL Shading Language (GLSL). GLSL strongly supports hardware mip-mapping by allowing data fetching from different levels. GLSL version 400 also includes functions allowing single-instruction fetching of surrounding texels (called *texture gathering*). Moreover, all major GPU manufacturers support GLSL.

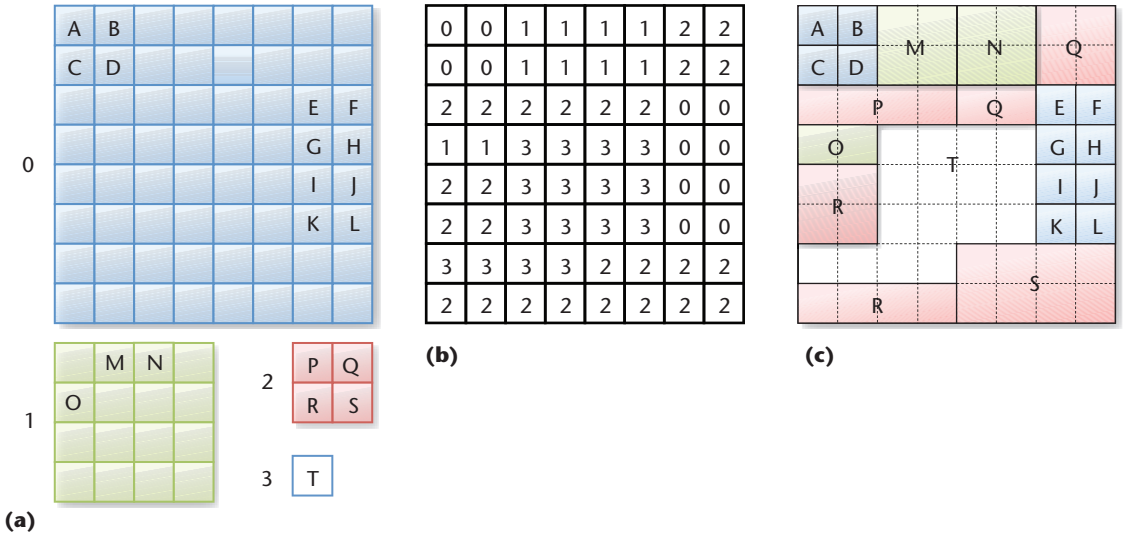


Figure 5. Merging (a) mip-map levels *D*, using (b) importance map *A*, into (c) the merged map $\hat{D}$. We broadcast values from higher levels of the mip-map into lower levels (see the upper-left quadrant).

```
Input: Merged data structure D̂, 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 D̂ 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):= DF(D̂(x, y))
5    end if
6   end for
7  end for
DF computes hydraulic erosion
```

Figure 6. An algorithm that implements hydraulic erosion on a mip-map. Instead of calculating the erosion for each pixel, it is evaluated with varying precision according to the underlying water and terrain complexity.

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

The importance map adds a channel, and the original data textures are automatically converted into mip-maps on the GPU. Mip-map merging occurs in separate data structures, which again represent water and sediment heights, flow, and terrain. The importance map has additional memory requirements and requires additional work to calculate cascade levels and merge them into a single data texture. Overall, our approach would need 50 percent more memory if it didn't use tiling. However, in our practical examples, we saved approximately 30 percent memory.

The Eulerian approach to fluid simulation is suitable for GPU implementation because it can fully utilize the GPU's parallel computing. All cells can be calculated independently of each other because they use values only from neighboring cells created in previous simulation steps. During the rendering loop, writing and reading from the same data texture at the same time isn't allowed. We solve this by first rendering the terrain into a mip-map chain and then merging the result into a final image. We then use the merged data in the next step as a read-only texture for mip-map generation.

The implementation uses frame buffer objects and fragment shaders to perform all screen-space calculations. Writing into data texture is performed by attaching the texture to frame buffer objects and drawing it as a full-screen quad with the shader activated. Because the system uses a lot of dynamic branching with indexed arrays, it requires a fast GPU, with support for an OpenGL version higher than 3. On GPUs with OpenGL 4 support, we can use fast instructions to gather all surrounding fragments around active fragments.

Rendering employs OpenGL 4 with a programmable graphics 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 a vertex shader. We then visualize the result in a fragment shader. Each terrain layer has different color textures; we blend the colors to create smooth transitions between layers according to each layer of thickness. Water is displayed on top of the terrain and is partially blended with the terrain layers.

Results

Figure 7 shows an example of interactive editing. The user initially created procedural terrain

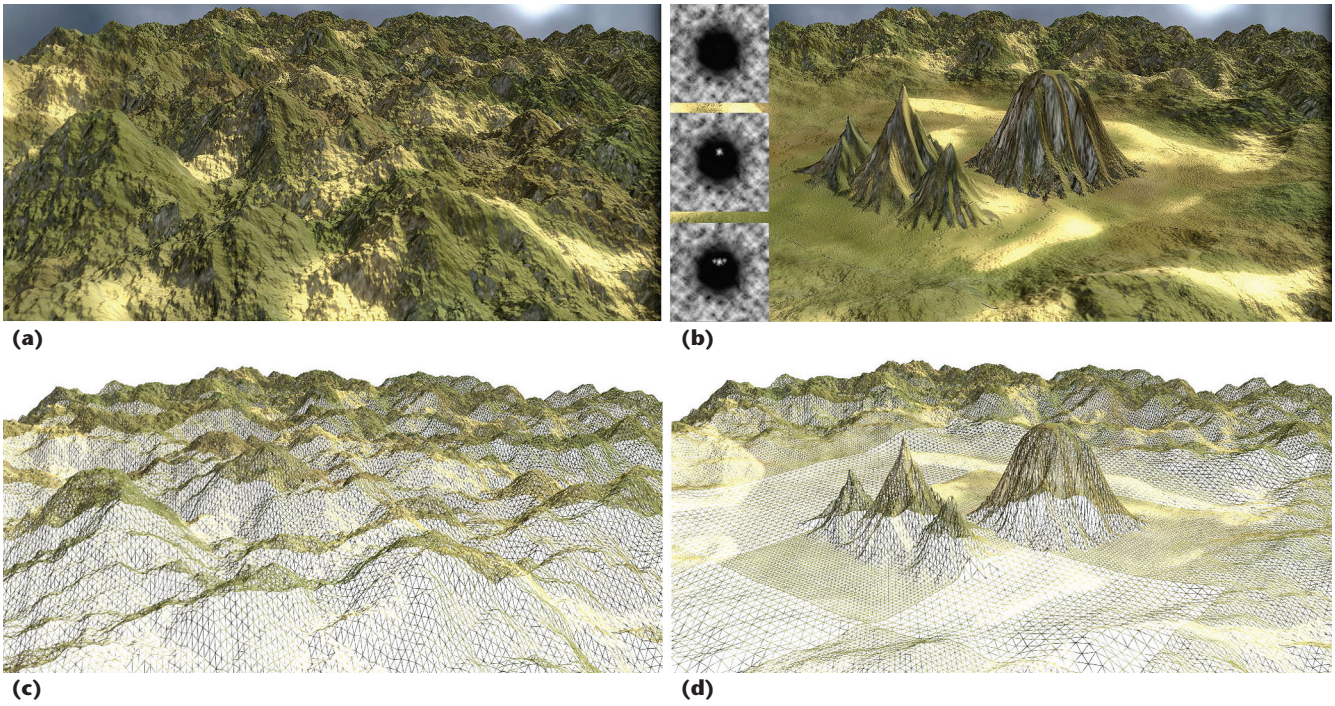


Figure 7. Large-terrain editing. (a) The input scene at 4,096 × 4,096 resolution before interactive editing. (b) The scene after interactive editing. The user added several patches of terrain (shown as successive insets) at 1,024 × 1,024 resolution, edited some valleys and mountains, and ran hydraulic erosion on certain areas. The overall editing took 10 seconds, and the simulation's average runtime per frame was 23 ms. (c) The corresponding tiling of the input scene. (d) The corresponding tiling of the scene after interactive editing. Areas with a large variation in altitude are in higher resolution than areas with a small variation.

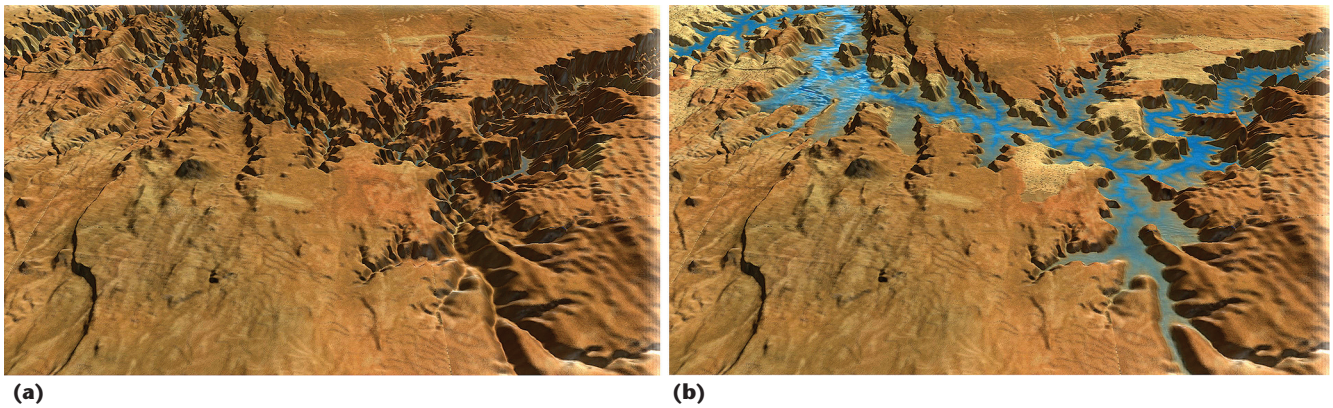


Figure 8. Employing a digital elevation map of the Grand Canyon. (a) The original map, which had a resolution of 8,192 × 4,096 pixels. (b) An erosion simulation using the map as input. This shows the underlying terrain flooded from various sources of water.

from Perlin noise in three layers of material, each at 4,096 × 4,096 resolution (see Figure 7a). This model was loaded into the system and tiled. The entire model has a theoretical size of nearly 3 Gbytes, but the tiled size was only 1 Gbyte. The user then employed three images at 1,024 × 1,024 resolution (see the insets in Figure 7b) and added them to the terrain using blending mode. The system automatically recalculated tile resolution (see Figures 7c and 7d). This editing took less than 15 seconds; tile recalculation and thermal weathering took approximately 25 milliseconds.

Figure 8 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 8,192 × 4,096.

Figure 9 shows an example of interactive physics-based erosion. A scene with a mountain of different materials that has been eroded by a water source manually positioned over them (see the blue circle). The simulation time was 25 ms per frame; the resolution was 4,096 × 4,096. The example shows how the different materials erode at different speeds and how the sand is being deposited at the foot of the mountain.

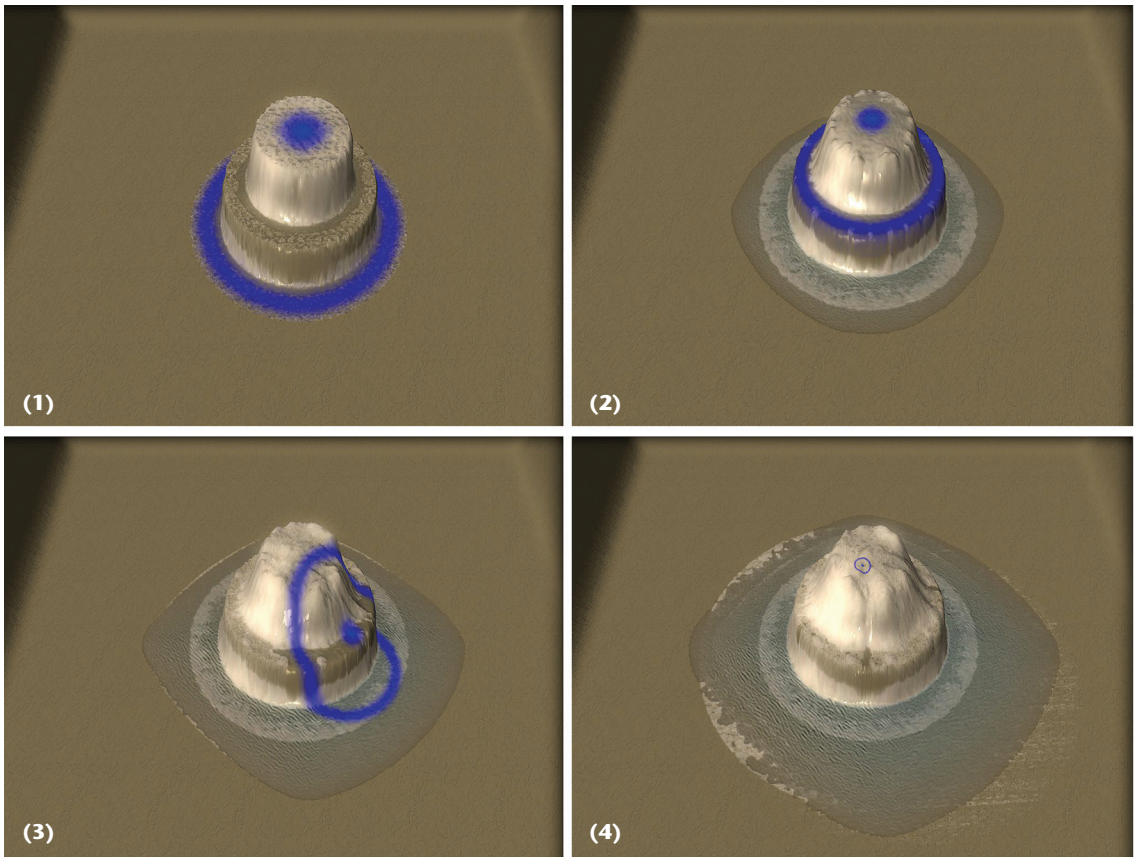


Figure 9. Interactive editing using a physics-based brush (the blue circle) with localized rain. The object is made of hard material with soil on its top. This example shows how different materials erode at different speeds and how sand is being deposited at the foot of the mountain.

Figure 10 shows a very large scene that didn't fit in GPU memory. The original 12,288 × 12,288 scene was tiled into 12,288 × 12,288 tiles with 1,024 × 1,024 maximum resolution. The scene used approximately 2.5 Gbytes of memory. The average simulation time was 43 ms per frame. On the GPU, the scene used a maximum of 1.5 Gbytes and ran at 12 fps for both rendering and simulation. For more on the simulations, see http://doi.ieeecomputersociety.org/10.1109/MCG.2011.66.

Tile Size and GPU Memory

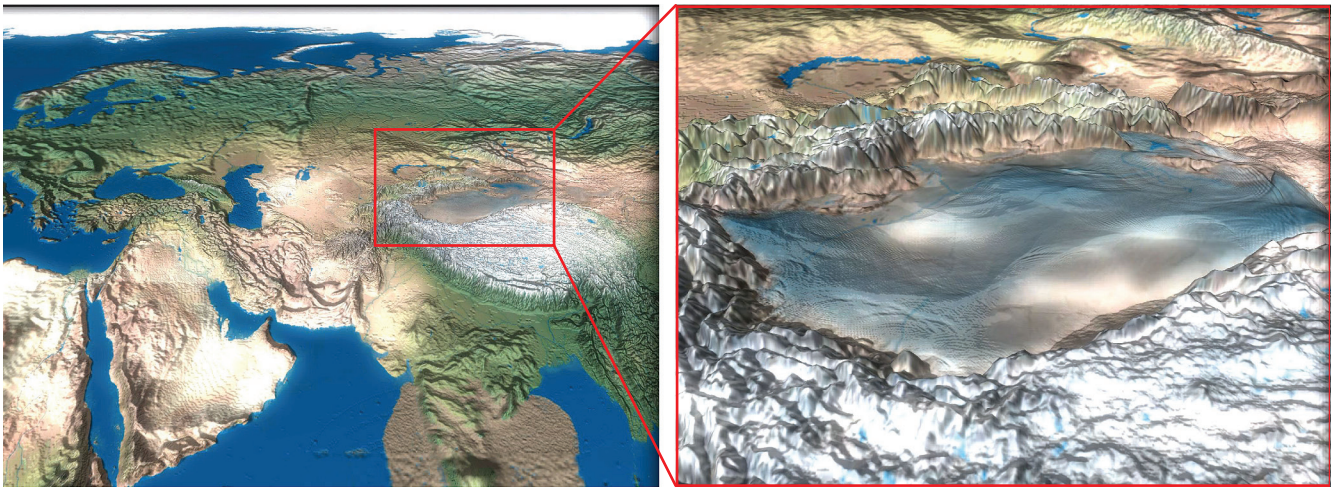
The actual size and, consequently, total number of tiles significantly affect performance. Intuitively, a large number of tiles will decrease performance because of tile synchronization overhead. A small number will obfuscate the performance gain of the importance map and varying tile resolution. Because different aspects can influence the effect of the number of tiles, we've created a benchmark that determines the optimal number of tiles.

As the shape of the simulation time curve in Figure 11 shows, the measurements confirm our intuition. The best performance occurs for tiles covering approximately 1 to 5 percent of the virtual terrain's area.

The GPU memory has a hardware limit. Because most calculations occur on the GPU, storing as many tiles as possible on it is beneficial. We could fit up to 256 tiles with small resolution (128 pixels) or a few large tiles (1,024 pixels) into GPU memory. However, the application is fully functioning even when processing much larger scenes that don't fit into GPU memory. This is because the OS 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 a tile's border, the system selects the neighboring cells from the appropriate tiles because all tiles can access textures from their surrounding tiles. This process guarantees seamless transition of water flow and the accompanied material among tiles of the same resolution. When neighboring tiles have different resolutions, the system uses hardware-level linear interpolation when fetching values, which introduces a minor interpolation error. However, this error wasn't significant in our experiments.

Table 1 shows the performance and memory requirements of the scenes from this article. We rendered all the scenes in nonadaptive mode at the maximum-possible resolution with 928 Mbytes of



(a) (b)

Figure 10. Handling a very large scene. (a) The entire scene, which took up 2.5 Gbytes at 12,288 × 12,288 resolution. (b) A detail. The scene occupied 1.5 Gbytes of GPU memory and was eroded and rendered on the GPU at 12 fps.

memory; the timing was a nearly constant 34 ms. Our adaptive method shows an average speedup of 1.46 and an average memory savings of 75 percent.

Simulation Error

To bring the physics-based terrain editing to interactive frame rates, we introduce numerical simplifications at different levels. Some sources of possible errors are different tile resolutions, evaluation of the hydraulic erosion at varying levels of detail, and conversion of data from different resolutions (merging between mip-maps and between neighboring tiles in different resolutions). Tracking the simplifications' effect and comparing them in depth would be difficult.

However, we intend this approach for interactive editing, not for physically precise simulations. The pipe model we use for the shallow-water simulation isn't an exact physical simulation. Moreover, fluid simulation is a dynamic system and is extremely sensitive to initial conditions. A small change in those conditions will cause a system to significantly diverge in the solution after a few steps, even for physically exact simulations.

We tried to ensure that the simplifications and inducted inaccuracies wouldn't create 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 12 shows a sequence of a large scene with water erosion simulation.

Visual artifacts can appear on tile borders when adjacent tiles have different resolutions. Although data synchronization errors can be small, visual artifacts happen because each tile uses its own polygonal mesh, and there can be visible cracks between two meshes with different mesh densities. We could remove this problem by putting all tiles on a single large mesh and using the appropriate terrain LOD (level-of-detail) algorithm. Although the terrain-tiling error is potentially problematic, especially for

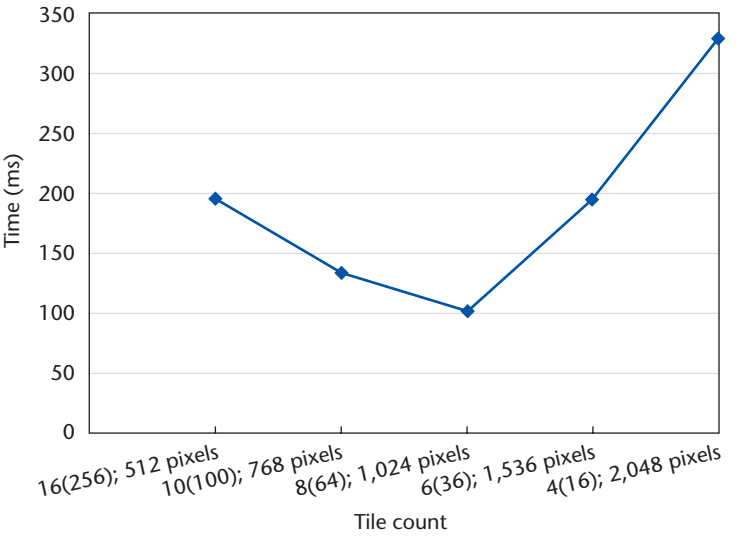
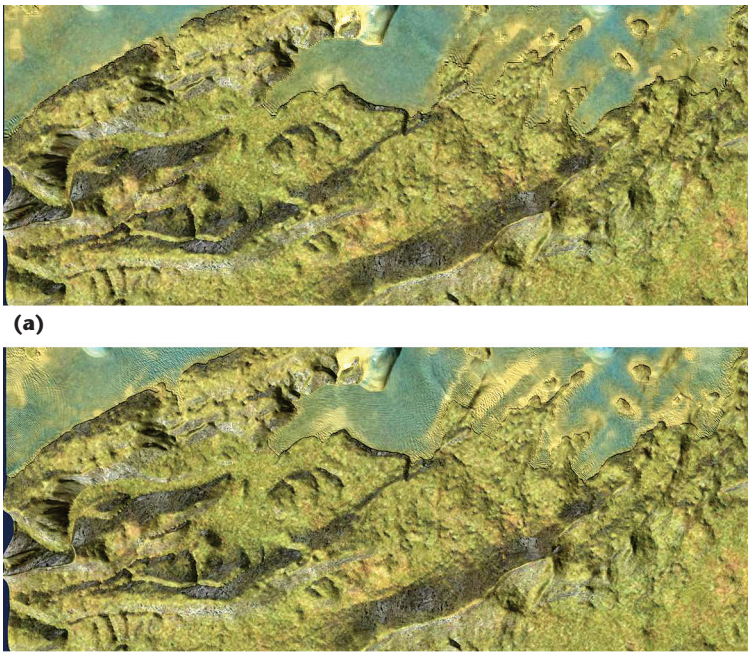


Figure 11. Large scene simulation as the function of tile size. Large numbers of small tiles create considerable performance overhead; small numbers of large tiles don't utilize the adaptability efficiently. A reasonable tile size is approximately 1 to 5 percent of the terrain area's total size.

Table 1. Time and memory requirements.*

Scene	Time (ms)	Speedup	Memory (Mbytes)	Savings (%)
Figure 3	25.4	1.34	700	75
Figures 7a, 7c	23.6	1.44	784	78
Figures 7b, 7d	21.1	1.61	698	75
Figure 8	22.7	1.45	770	82

**The table doesn't include the scene from Figure 10 because it can be edited only in adaptive mode.*



(a)

(b)

Figure 12. Comparing our simplified simulation with a detailed one. (a) The adaptive version of a scene. (b) The scene with full erosion simulation. We ran both simulations for 550 frames to ensure the water had enough time to propagate through the scene. There were no significant visual differences.

neighboring tiles with significantly different resolutions, it was minimal in our experiments.

Our approach still has several notable issues and potential pitfalls. The mip-map subdivision will be ineffective for large water dynamics scenes (such as rain). The tile resolution subdivision will be ineffective for white-noise scenes or scenes with high-frequency information equally spatially distributed.

Also, as we mentioned before, when the simulation exceeds the GPU's available memory, the driver swaps memory pages from the GPU with the main memory. This incurs a performance penalty. In-house memory management, such as per-tile least-recently-used cache, could address this problem and let users edit larger datasets.

Finally, good error evaluation and analysis could improve the algorithm's robustness.

Many possible avenues for future work exist. Our current implementation resamples all affected tiles immediately, which is costly. We could implement a priority queue that would process only the tiles that are being edited or that have significant visual importance. Also, we're convinced our approach will allow the incorporation of not only different physics-based methods but also new editing techniques.

Acknowledgments

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

References

1. B. Beneš and R. Forsbach, "Layered Data Representation for Visual Simulation of Terrain Erosion," *Proc. 17th Spring Conf. Computer Graphics* (SCCG 01), IEEE CS Press, 2001, pp. 80–86.
2. N. Holmberg and B.C. Wünsche, "Efficient Modeling and Rendering of Turbulent Water over Natural Terrain," *Proc. 2nd Int'l Conf. Computer Graphics and Interactive Techniques in Australasia and South East Asia* (GRAPHITE 04), ACM Press, 2004, pp. 15–22.

Juraj Vanek *is a PhD student in Purdue University's Department of Computer Graphics and Multimedia. His research interests include real-time rendering, procedural modeling, and physics-based computer graphics. Vanek received an MS in computer science from VUT Brno and is a PhD student of computer graphics at Purdue University. Contact him at vanek@purdue.edu.*

Bedřich Beneš *is an associate professor and Purdue University Faculty Scholar in Purdue University's Department of Computer Graphics Technology. His research interests include procedural modeling, real-time rendering, and general-purpose computing on GPUs. Beneš has a PhD in computer science from Czech Technical University in Prague. Contact him at bbenes@purdue.edu.*

Adam Herout *is an associate professor in Brno University of Technology's Department of Computer Graphics and Multimedia. His research interests include real-time rendering, video compression, and GPU computing. Herout received a PhD in Computer Science from VUT Brno. Contact him at herout@fit.vutbr.cz.*

Ondřej Šťava *is a PhD student in Purdue University's department of Computer Graphics. His research interests include procedural modeling, real-time rendering, and GPU-based computer graphics. Šťava received an MS in computer science from Czech Technical University in Prague. Contact him at ostava@purdue.edu.*