Simulation and Rendering for Millions of Grass Blades

Zengzhi Fan*
Shanghai Jiao Tong University

Hongwei Li†
Karl Hillesland‡
Advanced Micro Devices

Bin Sheng§
Shanghai Jiao Tong University
State Key Lab. of Computer Science,
Inst. of Software, Chinese Academy of Sciences

Figure 1: *A large scene with millions of grass blades and* 128 *objects.*

Abstract

We provide detailed simulated response for individual blades of grass in fields of millions of blades. The field is divided into tiles whose blade data are instanced from a small patch of blades on the GPU to limit memory and bandwidth requirements. We only instantiate simulation state and compute simulation for tiles interacting with objects. The simulation does not stop immediately when objects leave the tile but with a smooth transition to the original GPU-instanced state. Grass motion is solved with collision, length, bending and twisting constraints. Global animation from wind is still handled through conventional, procedural methods in the vertex shader. Our method is also compatible with a rendering level-of-detail (LOD) system. With 128 objects moving in a field with over a million blades of grass, the frame rate is less than 20 ms, with only a few milliseconds of that time for simulation.

CR Categories: I.6.8 [Simulation and Modeling]: Types of Simulation—Animation I.6.8 [Simulation and Modeling]: Types of Simulation—Parallel I.3.8 [Computer Graphics]: Applications;

Keywords: Grass, GPU, instancing, simulation

*e-mail:fanzengzhi@sjtu.edu.cn
†e-mail:Hongwei.Li@amd.com
‡e-mail:Karl.Hillesland@amd.com
§e-mail:shengbin@cs.sjtu.edu.com

1 Introduction

When we walk through an environment as shown in Fig. 1, we expect the grass to respond to our passing. Similarly, we expect other objects moving through the environment to affect the grass as well. This requires individual and independent motion of each blade of grass. Realtime applications typically ignore these effects or use very coarse approximations. To simulate individual movement of many small deformable objects is usually considered computationally impractical.

Our goal was to develop a system that gives us realistic response at high fidelity. Any blade of grass of millions can be individually pushed aside, and will respond appropriately with real-time performance. To avoid wasted simulation, our system carefully selects which blades of grass actually require simulation as a response to collision.

Given that explicit instantiation of millions of blades of grass would require too much memory and bandwidth, our system must work with GPU-based instancing. At the same time, the high quality simulation requires keeping simulation state between frames. Similar to simulation time, we limit memory use to where simulation is required.

Although our system only pays full simulation costs as needed for collision response, we also wanted a system that allows for global effects such as wind. Here we can employ standard procedural animation methods but our system must be able to handle the combination of both, and allow for smooth transition between the two effects.

Overall, our system adopts a tile-based scheme. Each tile in the scene refers to a subset of a pregenerated grass patch from which each grass blade has random length, orientation, width and position (Section 3.1). Each grass blade is described by a curve with 16 knots required for simulation. This curve is expanded to a triangle

strip in a vertex shader before shading (Section 3.2). We store no per-blade data unless simulation is required, at which point we allocate memory for grass blades in the tile and run simulation within this memory (Section 4.1). We adopt blade simulation introduced by Han et al. [2012] and implement procedural wind simulation in a vertex shader (Section 4.2 and 4.3). Blade geometry data are fetched from the pregenereted patch or the tile's own memory in a vertex shader, where blending of wind simulation and collision simulation can be easily achieved.

Our contribution is to show that high quality, constraint-based physical simulation for each individual blade of grass is possible, even when working with millions of individual blades of grass.

2 Previous Work

Modeling, rendering, and animation of grass are challenging because of the quantities involved. All three of these challenges were addressed very early by William Reeves and Ricki Blau [1985]. They modelled grass through a stochastic process, instancing the actual blades for rendering in buckets, so they did not have to store the complete geometry of all grass at once. Animation of grass due to wind was modelled using a procedural animation method. This was not a real-time system, but these basic concepts are still used, including real-time systems such as our own.

One area that Reeves and Blau did not explore, is grass response to collision with solid objects. Previous work treats grass-object interaction as a process whereby the grass is procedurally moved aside [Guerraz et al. 2003]. In some previous work, there is modeling of motion by a spring-mass system [Hempe et al. 2013; Orthmann et al. 2009] or wave simulaton [Pelzer 2004; Chen and Johan 2010]. In our work, we adopted a simulation method that handles collision as a hard constraint in the solver, as well as edge length constraints, and bending and twisting as soft constraints, for individual blades of grass [Han and Harada 2012].

As in previous work, we divide the grass into a grid for simulation management and rendering levels-of-detail (LODs) [Orthmann et al. 2009; Qiu et al. 2012; Hempe et al. 2013]. By dividing the scene into a grid, we are able to target only those grid tiles that require response to collision events [Orthmann et al. 2009; Hempe et al. 2013].

The closest work to our own is that of Hempe et al. [2013]. We have a few difference with their system, however. In their system, collision between the object and plant clusters are computed on the CPU, and the GPU simulates response as a spring-mass system. By contrast, we perform collision computation for each vertex of each blade against the object on the GPU and apply the results as a constraint during simulation, giving us higher simulation fidelity. Second, our instancing system is a bit more aggressive, and yet still accommodates dynamic simulation.

3 Rendering

3.1 Instancing

Our synthesized scene is composed of millions of individual grass blades. To generate the geometry data for these blades, we first employed a procedural modeling method proposed by Boulanger et al. [2009]. But as each blade has its own geometry, it introduces a gigantic memory footprint (about 4 GB for 4M grass blades with 32 vertices).

Our solution is to pregenerate a list of grass blades. We first manually create a few types of blades resembling the shapes of blades in the wild. Then we instantiate these blade types and randomly plant blades within a square area with random orientation and length. We refer to this pregenerated list as the *patch*.

The grass field is divided into a grid of tiles of the same shape as the pregenerated patch. Each tile refers to a subset of the pregenerated patch whose starting position is determined by a pseudo-random function of the tile index. When rendering, each grass blade within a tile is fetched from our pregenerated grass patch using the tile's starting position in the patch and blade's index within the tile; see Fig. 2 for an example. Since all tiles refer to the same patch of grass, the memory size for all grass geometry is only dependent on the number of grass blades in the patch. In our system, the pregenerated patch has $16,384$ blades which costs about 24 MB, which we found to be sufficient to ensure grass rendering with rich variance as shown in Fig. 1.

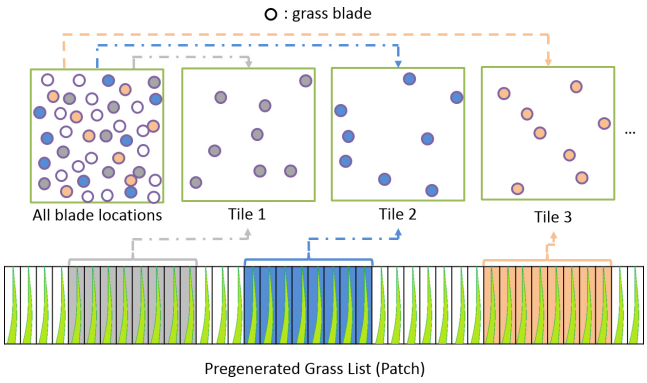

Figure 2: *To render a grass tile in the scene, we choose a fixed-count subset of blades from the pregenerated patch, translating them to the target tile in the vertex shader.*

We can reduce rendering complexity by only displaying tiles in the camera frustum. However, this optimization is not enough in a large scene where the visible tiles are still too many. Therefore we implement an LOD algorithm that draws less grass blades in distant tiles. The number of blades in a tile is determined by an LOD function such as $L(z) = (1 - 7(z - z_{near})/(8z_{far}))C$, where z is the distance of the tile to the camera, z_{far} and z_{near} are the far and near planes of the camera, and C is the largest blade number. The combination of frustum culling and LOD helps to greatly reduce the amount of geometry submitted to the GPU, and makes interactive rendering possible.

3.2 Expansion and shading

The grass blades in the pregenerated patch are curves and need to be expanded to triangles before rendering. For blade width variation, we expand at runtime. The most straightforward way is by geometry shader; however, a vertex instancing technique gives us higher performance. More specifically, we draw one line segment as two degenerate triangles (i.e., a quad). At runtime, every degenerate quad is expanded according to blade width and the direction of the vertex binormal. The expansion width of the grass blade curve gradually decreases from root to tip in order to form the blade shape. Fig. 3 illustrates this expansion process.

We render the grass blades using a modified Phong shading model. We adopt a simple shading model for the sake of performance, as there are millions of grass blades to render. We also incorporate a moderate degree of the subsurface scattering effect [Sousa 2007] by blending the back face shading value with a fraction of the front face shading value. Moreover, we find the choice of textures is the key to the high quality grass rendering, especially when the shading model is simple. We collected dozens of grass images, calcu-

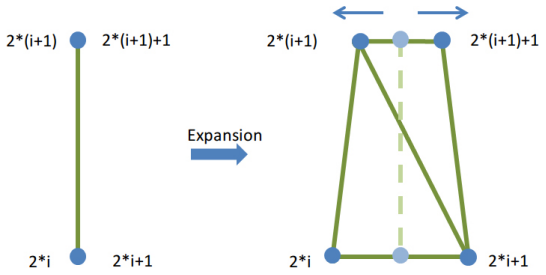

Figure 3: *Each segment of the blade curve is in fact a degenerate quad and is expanded to a normal quad with given width in the vertex shader. The translation directions of vertices follow the binormals of the vertices.*

lated a color histogram from those images, and use the histogram to guide the creation of grass textures [Jiao et al. 2012].

4 Simulation

As discussed in the introduction, our requirements for simulation are as follows.

1. It must give realistic response to contact, including deformation, at the level of individual blades.
2. It must handle millions of grass blades at interactive rates.
3. It must be compatible with procedural animation methods.
4. It must be compatible with the GPU instancing method we use for rendering, which requires no per-blade data.

In our system, we use *Bullet*, a standard rigid body system, running on the CPU to simulate the motion of objects other than grass. We make the approximation that these objects are not affected by the grass itself, trusting in Bullet's standard friction model.

For realistic response, we adopted an existing technique capable of handling contact, bending, and twisting of individual blades of grass, while maintaining edge length. The method we chose also runs completely on the GPU in order to give us the performance we need. This method is described in Section 4.2.

In order to handle millions of blades of grass at interactive rates, we cannot carry out simulation computation over all the grass in the scene but only for those tiles where simulation is needed. This is similar to previous work [Orthmann et al. 2009; Hempe et al. 2013]. Additionally, since our rendering system does not require any per-blade memory, we only want per-blade memory costs where simulation is required. This is different from previous work. We accomplish these goals through a tile management system described in Section 4.1.

As for most vegetation systems, we still employ procedural methods. Wind affects all blades of grass, and in our implementation, is a procedural method within the vertex shader. However, we make the approximation that this motion is additive with simulation due to collision response with objects.

4.1 Tile Management

In our instanced rendering scheme with pregenerated patch, there is no per-blade data for each instance. All tiles refer to the single pregenerated patch. However, in the simulation, each tile should acquire its own grass data; otherwise, the collision response will appear in a tile which does not interact with any objects, just because it shares the data with another tile that does. For memory efficiency, this per-blade memory is only allocated when needed and deallocated when the simulation is finished. To be more specific, when an object enters a grass tile, we activate the simulation in this tile by allocating memory for each blade and instantiating tile blades into this memory. After the object leaves the tile, we keep the simulation on for a while until the collision energy has been dissipated. Then we deactivate the tile, release its memory, and restore the tile back to match the original pregenerated patch state.

Algorithm 1 Tile Management

```
ToActivateTileList = ∅

// Enumerate all objects and find tiles that should be activated
// this frame.
for Object in ObjectList do
    for Tile that collides with Object do
        ToActivateTileList += Tile
    end for
end for

for Tile in ToActivateTileList - ActivatedTileList do
    // Activate the new tiles.
    Tile.Geometry = Instantiate(PregeneratedPatch)
    StartSimulation(Tile)
    ActivatedTileList += Tile
end for

// See if any current activated tile should be deactivated.
for Tile in ActivatedTileList do
    if Tile.SimulationEnergy == 0 then
        ActivatedTileList -= Tile
        Destroy(Tile.Geometry)
        Tile.Geometry = PregeneratedPatch
    end if
end for
```

Algorithm 1 describes how we maintain these activated tiles efficiently. We create a so-called Activated-Tile list to record these tiles and update them every frame. At the start of each frame, we enumerate each object in the scene and add all tiles that are interacting with any objects to a temporary To-Activate-Tile list. Then we find tiles in this list but not in the Activated-Tile list. We activate these new tiles by allocating memory and instantiating blades from the pregenerated patch and add them to the Activated-Tile list. Then the simulation process will calculate blade motion and update geometry data for each activated tile separately, avoiding affecting all unrelated tiles. For current active tiles in the Activated-Tile list, we check if any of them has their simulation energy dissipated. If so, we deactivate them by removing them from Activated-Tile list and deallocating all associated memory. Fig.4 illustrates tile management progress. The activated tiles are marked with red color.

For each tile, we also set up a per-tile object list that keeps track of the objects that interact with the tile. This list is essential, especially when there are a large amount of objects in the scene. When calculating intersection of objects and tiles, we enumerate all objects in the scene and add an object to a tile's list when its bounding box intersects with the tile's. With this per-tile object list, we don't have to test each tile against all objects in the scene, but only against those objects in the list.

4.2 Blade simulation

For realistic response, we use an existing method based on Verlet integration and iterative constraint solving [Han and Harada 2012].

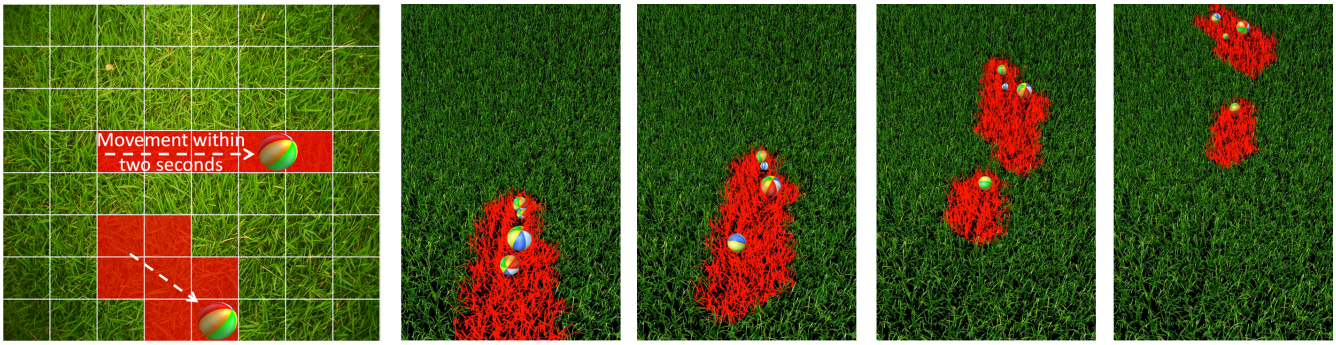

Figure 4: *(a) Activated tiles are updated according to the positions of objects in the scene. (b)-(e) are sequential screenshots that show activated tiles in the scene during the motion of four objects. The activated tiles are marked with red color.*

It allows for deformation in response to collision. It was originally designed to simulate tens of thousands of individual hair strands at interactive rates by efficient implementation on graphics hardware. Although this method was introduced for hair simulation, it has many similarities to blade simulation. As with a hair strand, a blade of grass can be simulated as a connected set of line segments. In this system, each blade of grass moves and deforms in response to contact, but converges back to its original state.

This system computes intersection of the object with each vertex of the underlying grass blade (a degenerated quad strip), solving for non-overlap as a constraint at a per-vertex level. A length constraint preserves grass blade length. Bending and twisting are treated as soft constraints. This gives us high quality collision response, allowing each blade to bend around the object, or even get mashed between two objects, with little to no stretching. We found two or three constraint iterations to be sufficient. These are the capabilities we inherited from the original simulation model. Our contribution is making this model feasible for such a large quantity of grass.

4.3 Wind simulation

We simulate wind effects with a procedural method in the vertex shader during rendering. This is common practice for vegetation systems, as its cost is low enough to enable movement in an entire field of grass at interactive rates [Pelzer 2004]. As it is only a function of time, no memory is required to store previous frame state. This also makes the method compatible with instancing techniques.

In our implementation, each vertex is moved according to its parametric distance from the root node using a sum of sin functions. We added small, random perturbations to prevent visible directional patterns.

5 Implementation

5.1 Rendering

We create a parameter buffer for every tile that is going to be rendered in that frame. Rendering parameters, including the tile starting position at the pregenerated patch and the tile world coordinate, are all stored in this buffer. In the vertex shader for grass rendering, we fetch parameters through the instance id. Blade geometry data are obtained with the vertex index within the patch added to the tile starting position in the pregenerated patch. With tile world coordinate, we can place grass blades in their world positions by translating vertices from the local space of the patch.

Without LODs, all tiles are rendered with a single instanced draw call. However, our LOD algorithm requires a different index count per instance, which is not supported in Direct3D 11. Therefore, we adopted a hybrid approach. From the CPU, we dispatch a number of instanced draw calls, to which we pass different index counts; see Fig. 5. In the vertex shader, we calculate the blade number using the LOD function and trim off redundant blade vertices by moving them outside the frustum. Refer to Section 6 for the performance improvement with our hybrid LOD approach.

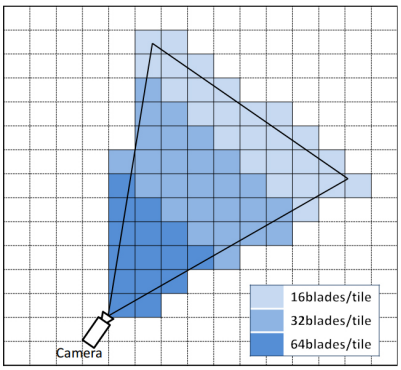

Figure 5: *At each frame, we dispatch 3 instanced draw calls whose blade numbers are 64, 32 and 16 per tile, respectively. In the vertex shader, the actual number of blades to be rendered is calculated through an LOD function and redundant blades are discarded.*

5.2 Simulation

In implementation, collision energy dissipation is hard to detect. Thus we adopt a more intuitive method to approximate this procedurally, similar to previous work [Hempe et al. 2013; Orthmann et al. 2009]. We attach a timestamp to each activated tile that records its activation time. This timestamp will be refreshed whenever an object enters the tile. At each frame, we compare the timestamp with the current time. If the difference exceeds a threshold, meaning the tile has not been touched with any objects in the past few seconds, we assume this tile's energy is gone and the tile becomes inactivate. After sufficient experimentation, we found a two-second time delay is appropriate for energy dissipation. Therefore we deactivate a tile two seconds after all objects have left it.

As the list of activated tiles keeps changing, the GPU memory allocation and deallocation is frequent enough to slow down the entire system. Thus we preallocate a GPU memory pool for fast memory

recycling. We allocated enough memory for 4096 tiles, which is sufficient for 128 objects to interact with 32 tiles each at the same time. Suppose each tile has 64 blades and each blade contains 32 vertices, the memory size of the pool is roughly 384 MB.

6 Results

We implemented the system with C++ and Direct3D∼11 and benchmarked it on a Windows 7 PC with an AMD A10-6800K CPU running at 4.1GHz and a Radeon HD 7970 graphics card. We used 4X MSAA in all experiments. Fig. 6(g) shows a bird's eye view of simulation and rendering for a scene with more than 800K blades and 128 objects. In Fig. 6(a)-(f) are 6 snapshots of grass blade motion. It shows how grass blades interact with objects during the objects' movement. Although we skip the collision detection between grass blades, interactive response gives the viewer a feeling that grass blades move naturally.

Rendering performance For the great amount of blades to be rendered in every frame, the rendering performance is unsurprisingly limited by vertex processing, yet our system is able to achieve interactive performance (28 ms per frame) even with more than 30 M triangles. Table 1 illustrates rendering performance with and without LOD. From the data, we observe that LOD is effective in improving performance without a distinct change on the scene's appearance, especially for scenes with a large quantity of tiles.

Table 1: *Rendering performance with and without LOD in FPS.*

Tile No.	**2K**		**4K**		**8K**		**16K**	
Blades/Tile	LOD		LOD		LOD		LOD	
	no	yes	no	yes	no	yes	no	yes
64	71	73	66	67	54	56	28	44
128	72	76	56	62	32	60	15	41
256	57	72	30	49	16	36	7	23
512	31	50	16	35	8	20	4	12

Table 2: *Frame profiling data for simulation and rendering.*

	GPU (ms)	**VS**	**PS**	**CS**	**Tex**
Grass simulation	5.6	0	0	99.9%	99.7%
Grass rendering	5.4	99.9%	75.0%	0	32.1%
Skybox	0.3	0.11%	31.3%	0	92.2%
Terrain	0.3	99.5%	72.7%	0	10.9%

Simulation performance With the same grass blade density, tile size is crucial for simulation efficiency. We have compared simulation efficiency between a smaller tile size, which has 64 blades per tile, and a bigger tile size, which has 256 blades per tile, with the same grass blade density. In this test, the total grass blade count of the scene is set to 1.28M. There are in total 20K smaller tiles or 5K bigger tiles. We throw 128 objects in the scene and count how many blades are activated and record the simulation time after 5 seconds. We obtain an average simulation time with five tests. The result is 5.52 ms (0.01 ms per tile) with the smaller tile size and 10.52 ms (0.02 ms per tile) for the bigger tile size. From these numbers we see that for the same grass density, a smaller tile (i.e., 64 blades/tile) gives us better performance because it results in less unnecessary simulation for those grass blades that are not interacting with other objects. The reason we use 64 as the minimum blade amount per tile is that it is the wavefront size of our target hardware. Therefore, we use 64 grass blades per tile in our system and all other tests.

The objective of our system is to add physical simulation for the whole scene but only pay cost for tiles where physical simulation is required. In Fig. 7, the blue line represents simulation time as a function of activated blade amount, and the orange line represents simulation time as a function of total blade amount. In the test for simulation time and activated tile amount, the total blade amount is set to 1.28M. And in test for simulation time and total blade amount, we hardcode 64K blades to be activated throughout the test. As shown in Fig. 7, the result verifies our design that simulation overhead only relates to the amount of activated tiles.

We use a per-tile object list to reduce the number of testing objects in the simulation computation. Compared to the case without the list, we found the simulation time decreases by 40% to 60%, depending on the maximum object amount in the scene. This performance increase shows that a per-tile object list is especially beneficial when we have a large amount of objects in the scene.

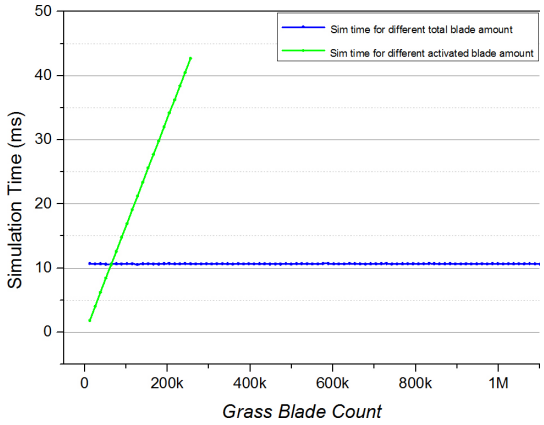

Figure 7: *The simulation time does not change with the total number of blades in the field because the simulation is only active for blades that interact with objects. This simulation time is linearly proportional to the amount of activated blades. The orange line has 64 K activated tiles for all measurements, which is why the two lines intersect at that point.*

For a scene with 23K activated grass blades, 4M total grass blades, 64 blades per tile, and 128 objects, we have profiled simulation and rendering of a frame using AMD's GPU PerfStudio. Each frame takes around 12.5 ms. CPU time is 10.55 ms while GPU time is 12.17 ms (CPU and GPU are executing on parallel, so the total frame time can be smaller than the sum of the two). Since CPU time is not the bottleneck, we now turn to more detailed analysis of the GPU time, though we propose methods to reduce CPU time in the discussion of future work (Sec. 7).

Table 2 shows the profiling results for items taking more than 0.1 ms of GPU time. In the table, percentage numbers indicate how busy the hardware stage is. We list some of the important stages in the graphics pipeline. These values are likely to sum to more than 100 percent because of the high parallelism of the GPU. VS, PS, and CS represent vertex shader, pixel shader and compute shader respectively. Generally speaking, the stage with the highest percentage is indicative of the bottleneck of the draw call. From the table, there are a few observations: (1) the bottleneck of the rendering is the vertex processing of grass, for which the occupancy ratio is much larger than for the PS; (2) there is some space for optimization in the simulation seeing the frequent texture fetch. These observations inspire some future work.

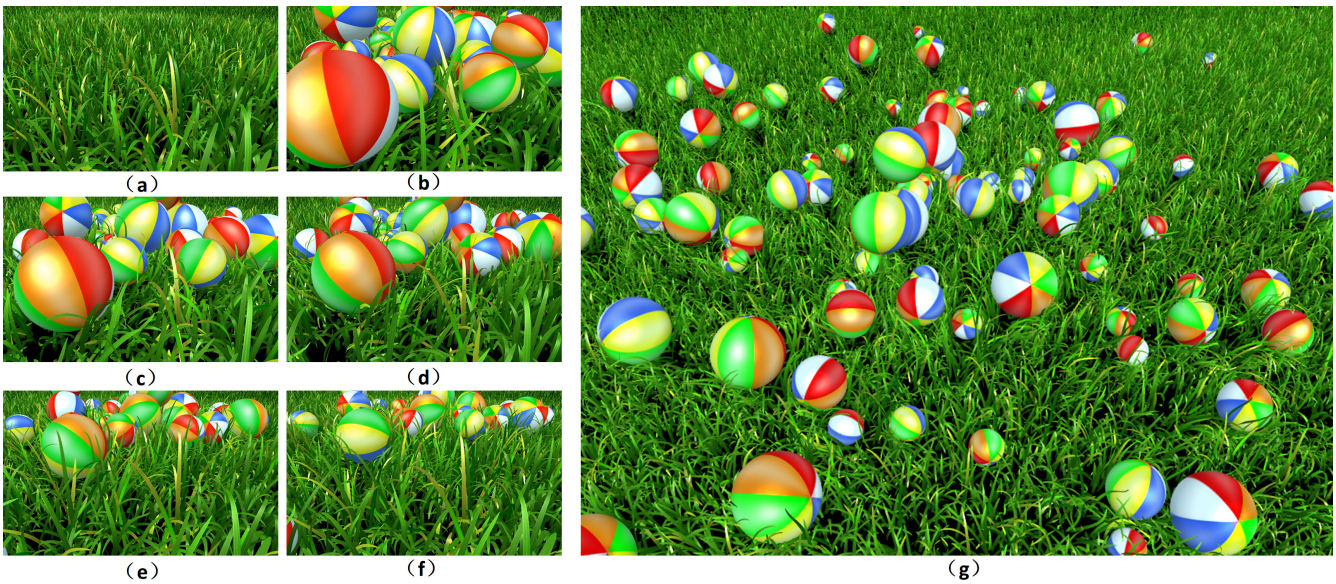

Figure 6: *(a)-(f) a sequence of snapshots of grass blade motion after throwing a bunch of objects; (g) A bird's eye view of the scene with more than* $800K$ *blades after throwing* 128 *objects.*

7 Conclusions and Future Work

We have presented a novel and practical framework for simulation of individual grass blades in very large scenes. With tile management, we can achieve an interactive simulation and rendering for millions of grass blades while keeping the memory footprint at a moderate size. With respect to simulation, compute and memory cost are only proportional to the number of tiles that are activated rather than the whole scene. Moreover, our simulation is compatible with procedural animation methods for the grass blades.

In the future, we would like to do the tile management on the GPU. This would not only free up CPU compute time, and reduce dynamic buffer update costs, but would enable more options for tile deactivation. For example, we could deactivate a tile by actually using blade movement data that is already in GPU memory rather than simply waiting for a fixed time interval. Our instancing and tiling system also creates new challenges with respect to plastic deformation and fracture of the grass, as well as authoring that we would like to explore. We would also like to look at integration with more traditional billboard-based systems, and other vegetation types.

8 Acknowledgements

We would like to thank Jason Yang, Dongsoo Han and Jay McKee for their helpful discussion and tool support, and anonymous reviewers for their valuable suggestions.

The work is supported by the National Natural Science Foundation of China (No.61202154, 61133009), the National Basic Research Project of China (No. 2011CB302203), Shanghai Pujiang Program (No.13PJ1404500), the Science and Technology Commission of Shanghai Municipality Program (No. 13511505000), and the Open Project Program of the State Key Lab of CAD&CG (Grant No. A1401), Zhejiang University.

References

BOULANGER, K., PATTANAIK, S., AND BOUATOUCH, K. 2009. Rendering grass in real time with dynamic lighting. *Computer Graphics and Applications, IEEE 29*, 1 (Jan), 32–41.

CHEN, K., AND JOHAN, H. 2010. Real-time continuum grass. In *Proceedings of the 2010 IEEE Virtual Reality Conference*, IEEE Computer Society, Washington, DC, USA, VR '10, 227–234.

GUERRAZ, S., PERBET, F., RAULO, D., FAURE, F., AND CANI, M.-P. 2003. A procedural approach to animate interactive natural sceneries. In *Proceedings of the 16th International Conference on Computer Animation and Social Agents (CASA 2003)*, IEEE Computer Society, Washington, DC, USA, CASA '03, 73–78.

HAN, D., AND HARADA, T. 2012. Real-time hair simulation with efficient hair style preservation. Eurographics Association, Darmstadt, Germany, J. Bender, A. Kuijper, D. W. Fellner, and E. Guerin, Eds., 45–51.

HEMPE, N., ROSSMANN, J., AND SONDERMANN, B. 2013. Generation and rendering of interactive ground vegetation for real-time testing and validation of computer vision algorithms. *Electronic Letters on Computer Vision and Image Analysis 12*, 2.

JIAO, S., WU, W., HENG, P.-A., AND WU, E. 2012. Using time-varying texels to simulate withering grassland. *Computer Graphics and Applications, IEEE 32*, 78–86.

ORTHMANN, J., REZK-SALAMA, C., AND KOLB, A. 2009. GPU-based responsive grass. *Journal of WSCG 17*, 65–72.

PELZER, K. 2004. Countless Blades of Waving Grass. In *GPU Gems*, Oxford University Press, Oxford, R. Fernando, Ed.

QIU, H., CHEN, L., CHEN, J. X., AND LIU, Y. 2012. Dynamic simulation of grass field swaying in wind. *Journal of Software*, 431–439.

REEVES, W. T., AND BLAU, R. 1985. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '85, 313–322.

SOUSA, T. 2007. Vegetation procedural animation and shading in Crysis. In *GPU Gems 3*, Pearson Education, Inc., Boston, H. Nguyen, Ed.