GPU-Based Rendering and Collision Simulation of Ground Vegetation in Large-Scale Virtual Scenarios

Flavio Paulus Franzin, Cesar Tadeu Pozzer, Bruno Torres do Nascimento
Universidade Federal de Santa Maria
Programa de Pós-graduação em Ciência da Computação
Santa Maria, Brazil
{ffranzin, pozzer, brunotn}@inf.ufsm.br

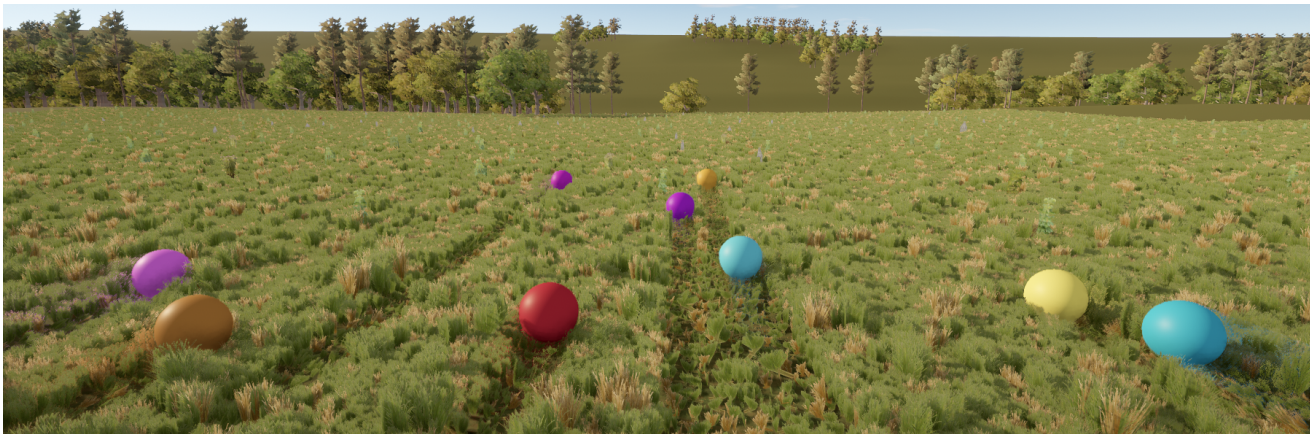

Figure 1. An example of the rendering and collision results. The long-lasting collision generated by the solution is visible at the trail on the ground.

***Abstract*—The user's immersion is highly related to the visual aspects of virtual scenarios. Thus, it is imperative for such applications the rendering of ground vegetation with high fidelity. However, the representation of these elements requires a high storage and graphics processing, which are the bottleneck of the other approaches. We present a GPU-based solution for handling dense ground vegetation on large-scale scenarios. Our proposal includes an architecture to distribute, render, and, to deal collisions with dynamic objects. The plants are placed in the scenario based on positions computed in real-time. Collisions are encoded in vector fields, which store the deformations to be applied to the plants during the rendering. These collisions may result in temporary or permanent deformation on the plants. The rendering performance is optimized through LOD and Instancing techniques, and the memory cost is reduced compressing the vector fields. The results show that the proposed approach is capable of handling ground vegetation for large-scale scenarios, ensuring a pleasant visual with low processing and storage costs.**

***Keywords*-Grass; Ground Vegetation; Collision Simulation; GPU-Based; GPU-Instancing; Real-Time; Large-Scale;**

I. Introduction

The graphics media industry, such as games, simulators, and movies, increasingly demand virtual scenarios with a high level of realism in order to maximize the immersion of its users. Natural environments are examples of virtual scenarios, which are characterized by vegetation and generally have large extensions. Ground vegetation, such as grass or small shrubs, are elements of these environments that can be easily represented by simple geometries. However, these plants must be replicated massively to achieve a considerable level of realism, requiring high demands of graphics processing.

Some proposals, such as [1] [2], address the rendering of individual grass blade using geometry-based approaches, ensuring great visual results. However, these approaches require a high computational cost. [3] [4] use image-based approaches, where plants are represented by textures projected on quads. In this way, these approaches are characterized as being of low computational cost and are mostly employed by real-time applications.

For interactive applications, collisions with dynamic objects become desirable, since they can be triggered by an action of the user or NPCs, when controlling the movement of characters or vehicles through the scenario. Also, a desirable characteristic is the dynamic behavior after the collision, once plants have flexible bodies, requiring distinct periods to recover their natural form. Additionally, the storage of the ground vegetation deformation has a memory cost, being the bottleneck of some approaches [3] [5].

In this paper, we propose an efficient GPU-based solution to render and animate ground vegetation (grass and small shrubs) visually appealing, capable of large-scale handling scenarios in real-time. Our primary focus is to apply adap-

tive deformations to the plants, with different deformation intensity and recovering times. The vector fields store the plant's orientation, which in turn is modified based on the collision with object dynamics. Then, during the rendering, these plants are curved to follow its orientation vector. The deformed vector fields have their initial shape restored through an associated individual cost for each vector, ensuring different behaviors for the plants. The vector fields, in some cases, are compressed to reduce the storage cost. The complex geometries of 3D objects are approximated by spherical and planar colliders. Rendering is done in batches using an image-based approach.

Briefly, this article presents the following contributions:

- Collisions based on vector fields, and management that allows generating adaptive deformations for the plants, capable of representing lasting or permanent collisions with total integrity. To our knowledge, we are the first to approach this feature on large-scale scenarios;
- An efficient GPU-based Discrete Level of Detail, coupled with an improvement on the refinement of billboard's geometry, specifically adapted to maximize the performance of the ground vegetation rendering.

This paper is structured as follows: Section II explores related works. In Section III, we present an overview of our approach, which is discussed more in-depth in Sections IV through VIII. Finally, in Sections IX and X, we present and discuss the achieved results.

II. Previous Work

The main techniques used to render ground vegetation can be classified as image-based, geometry-based, and hybrid-based. The geometry-based approaches [2] [1] guarantee pleasant visual results by rendering individual grass leaves but are computationally expensive. Image-based approaches [3] [4] use billboards to represent sets of grass, being highly targeted by real-time applications because they require low computational costs. Hybrid-based approaches [6] integrate geometry and image-based approaches with the Level of Detail, where grams near the observer are rendered using geometry-based approaches, and distant grams are rendered using image-based approaches.

Level of Detail (LOD) is widely used in real-time applications in order to minimize costs during rendering. [2] [7] employ Continuous LOD (CLOD) using tessellation. However, given the quantity of geometry for the representation of plants, that approach may require substantial processing. As an alternative to CLOD, Discrete LOD approaches (DLOD) [1] [5] discretize the geometries (LOD0, LOD1, ...), and define which LOD will be used to render an object based on its distance from the camera. However, it is crucial — in real-time applications — that the rendering of the ground vegetation is done in batches. Accordingly, using DLOD approaches, LOD is associated with the entire batch, which is limited to rendering a single geometry.

Few studies have been explored to deal with collisions between ground vegetation and dynamic objects. [1] employs collisions between individual grass blades and simple geometries while rendering, at the vertex shader stage. [2] extends that approach by integrating collisions with complex geometries. However, the proposed solutions are limited to quick collisions, so that when the object moves, the grass immediately recovers its shape. [5], [3], and [8] present GPU-based solutions for dealing with collisions, storing the state of these in auxiliary structures, ensuring long-lasting deformations. However, these proposed solutions are limited to small scenes (e.g., [8] uses 4.8x4.8km scenes) because they demand much storage.

To increase the realism of plants during rendering, [2] folds the plants that have collided with dynamic objects using curves. [9] applies a global directional wind, ensuring great visual results.

III. Proposed architecture

In the proposed architecture, the virtual scenario is segmented into a set of MxN *Cells* indexed by a hash table (Fig. 2). The information — positions, scales and collisions — needed to represent the ground vegetation (that we refer to as plants) are stored in buffers in VRAM. The *Cells* maintain the reference to access these buffers.

The *Manager* coordinates submodules for plants' distribution, collision, and recovery, as well as memory management and rendering submodules. It is assigned to perform the selection of the *Cells* that need to undergo a specific procedure, invoking the submodule that performs that task.

The *Distribution* submodule invoke a compute shader that generates the plants' position once (Sec. IV), for *Cells* visible or close to the viewer. The generated positions are stored in the *Plants_Buffer* associated with each *Cell*.

The *Collision* submodule (Sec. V) performs the collisions between plants and moving colliders. The collisions are encoded in vector fields, stored in *Collisions_Buffer*, which are associated with each *Cell*. Each vector in the vector field represents the orientation of the plants that are placed close to it. *Collision Recovery* (Sec. V-D) gradually recovers deformed vectors — that collided with some collider — from vector fields.

Rendering is performed in batches using GPU-Instancing. For *Cells* that do not contain vector fields, batches are determined by the *Plants_Buffer*. The batches of the *Cells* that have associated vector fields are generated by the *Individual DLOD* submodule and are stored in the *DLOD_Buffers*.

Throughout *Rendering* (Sec. VII), only the deformations of vector field vectors and the wind animation are applied to the plants' billboard.

The *Memory Management* submodule (Sec. VIII) selects the *Cells* that are far from the viewer, and are not in contact with colliders, to have the *Plants_Buffer* discarded. In some cases, *Collisions_Buffer* are discarded or compressed.

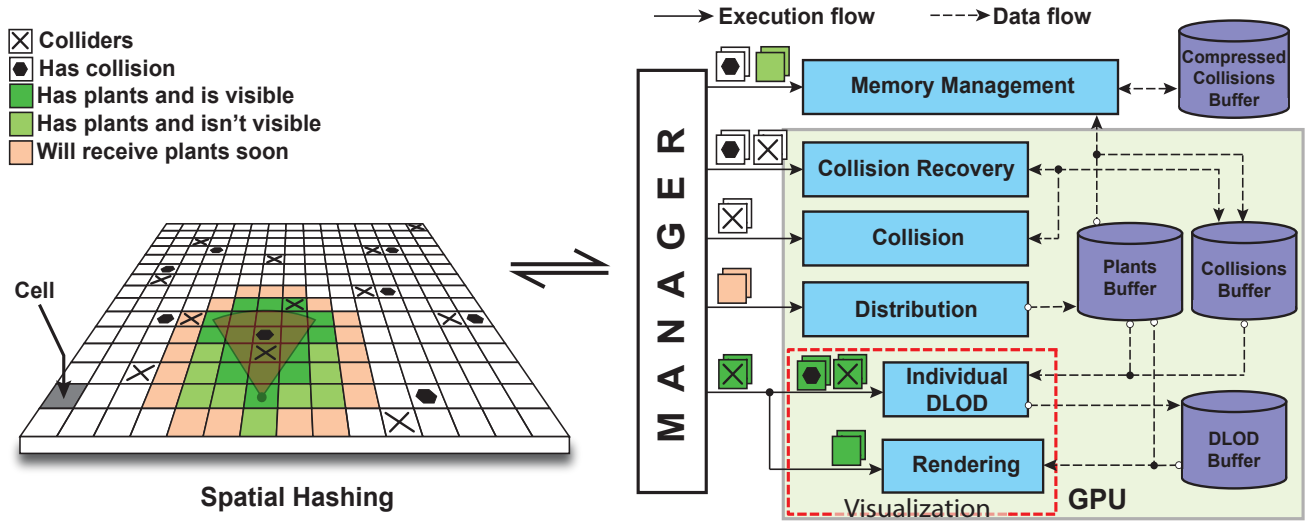

Figure 2. Overview of the architecture. Spatial Hashing is used to segment the virtual scenario. The Blue elements represent the submodules that processes ground vegetation and the demands of the system. The Purple elements represent the buffers needed to store information to represent the ground vegetation. The *Cells* with the assigned vector field is figured by *X* and the *hexagons*.

IV. Plants Distribution

The plants are distributed, using a compute shader, to the *Cells* that have not yet generated positions and are close to or in the view range.

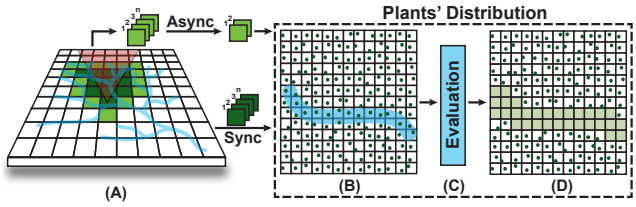

Figure 3. (A) Selection of *Cells* to perform Synchronously or Asynchronously plants' distribution. (B) The complete distribution of the plant's positions for a *Cell*, (C) followed by the evaluation of each position, (D) remaining only positions placed in appropriate areas (outside of roads, rivers, and lakes).

The *Distribution* submodule can be performed Synchronously (Sync) or Asynchronously (Async) (Fig. 3-A). The Sync distribution occurs for *Cells* that don't have the plant's positions computed, and must be rendered in the current frame. As a result, it blocks the graphic pipeline until the distribution is complete, and no restrictions on the number of requests can be established. In contrast, the Async distribution is applied to *Cells* that are close to the frustum area, but not visible. This distribution method is limited to a maximum number of simultaneous requests and can be executed across several frames, thus avoiding possible overloads on the GPU. For those reasons, it is attempted to distribute the plants before the rendering request, through the Async distribution.

During the plants' distribution, the *Cells* are divided into a regular grid of MxN *grid_cells*, where the center of these *grid_cells* corresponds to potential positions for plants. Also, a displacement, within these *grid_cell* boundaries, is applied to each position (Fig. 3-B). Through this approach ensures dense distributions using billboards as few as possible.

Another property of the plants' distribution consists of evaluate and eliminate positions inside inappropriate regions (Fig. 3-C). The resulting positions (Fig. 3-D) are associated with a scale that determines the plant's size during rendering. The plant's positions and scales are stored in the *Plants_Buffer*, assigned for the *Cell* that holds the distribution.

The scale of the plants can be defined by any noise function (e.g., Perlin Noise). However, once the generated plant's positions can be discarded for memory reasons (Sec. VIII), a deterministic seed is necessary to retain a consistent position's layout, since it may be necessary to redistribute those positions.

V. Collisions

The collisions are encoded in vector fields, which represent the plants' orientation. Each vector field is associated with a *Cell* and its vectors may be deformed individually when some collider interacts with them. Moreover, collision processing depends only on colliders, so the collisions are also applied in non-visible *Cells*.

A. Definition of the Vector Fields

The vector field vectors are evenly distributed within the *Cell* boundaries and stored in a 2D texture. The vector fields have a fixed density, which may be different from

that established to the plants. Each vector covers an area of the *Cell*, so, dense vector fields ensure greater accuracy to represent the collisions, and, consequently, provide more realism. However, that requires more processing to compute the collision.

A vector field, when assigned for a *Cell*, is initialized with all vectors pointing up. Also, to improve the collision calculations accuracy, the vectors are scaled proportionally to the plants' scale. Thus, the scale of the vectors is defined by the same noise used during the plants' distribution (Sec. IV), and discarded positions (e.g., inside the road) generate null vectors. It is important to emphasize that the plants' positions are not necessary for the initialization of the vector fields.

B. Definition of the Colliders

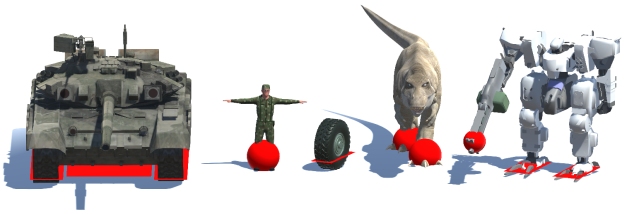

Figure 4. Examples of planar and spherical colliders used to approximate complex 3D objects.

The collision with 3D geometries, depending on their complexity, may be computationally expensive. Because of this, we use planar and spherical colliders to approximate complex 3D objects (Fig. 4).

It is important to highlight that colliders are not associated with the object's hierarchy, thus, it is possible to assign constraints to their transformations. For example, in Fig. 4, the collider cannot rotate in the direction of rotation of the wheel. In specific cases where a plane acts as a collider, it can have its normal vector changed, reflecting directly in the collisions.

Each collider has a radius of influence r, that covers all the collider's geometry, defined to reduce computational cost during collision calculations.

C. Collision Detection

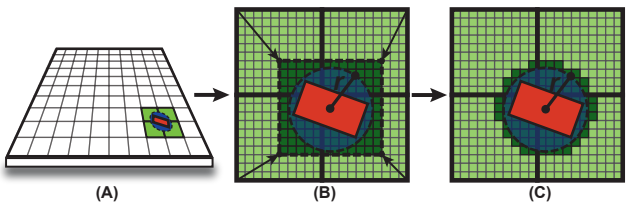

Figure 5. (A) Selection of *Cells*' vector fields in contact with colliders. The resulting vector fields are (B) cut off and (C) the resulting vectors that are outside of the collider's r are discarded.

The collision calculations are preceded by three steps (Fig. 5) defined exclusively to select *Cells* and vectors that have the possibility of interacting with the colliders.

- **Step 1:** The *Cells* in contact with colliders are selected, in the CPU, using the colliders' position and radius of influence r (Fig. 5-A);
- **Step 2:** The vector fields of the selected *Cells* are clipped using the colliders' position and r (Fig. 5-B);
- **Step 3:** In parallel on the GPU, the vectors of Step 2 that are outside the range of influence of the collider are discarded (Fig. 5-C).

It is important to emphasize that the application of Step 2 is possible because the vector fields have an uniform distribution, so it is possible to estimate which vectors can be affected by a certain collider. Also, through Step 2, it is possible to notice that the cost to compute the collisions is defined mainly by the collider's influence radius, and the *Cell* size has no impact on the collision.

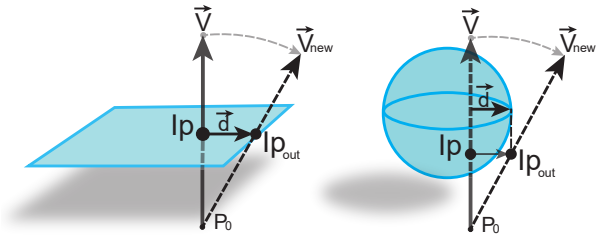

Figure 6. Vector field vector placed outside the geometry of the collider when they intersect.

The vectors resulting from Step 3, collisions are fully employed in parallel in the GPU from the intersection between the colliders and the vector field vectors. Consequently, when there is an intersection between them, it is needed to redirect the vector out of the collider's geometry (Fig. 6).

For vectors' redirection, it is first defined as the intersection point (Ip) between the vectors and the colliders. Through this point and from $\vec{d}$ — that describes the direction and the displacement of the collider relative to the last collision test — it is possible to define the point outside the geometry (Ip_{out}). Having Ip_{out} and position P_0 of $\vec{V}$, it is calculated the $\vec{V}_{new}$ (1), which represents $\vec{V}$ bent out of collider's geometry.

$$\vec{V}_{new} = \overrightarrow{Ip_{out}P_0} \cdot \|\vec{V}\| \tag{1}$$

D. Collision Recovery

An individual recover cost β is assigned to each vector that intersects with a collider, allowing to set distinct behaviors to vectors that interact with different colliders. The β associated with a given vector may be high enough to never establish its original shape (parallel to $\vec{Up}$), resulting in permanent deformations.

Vector recovery begins, in the CPU, with the selection of *Cells* that contains deformed vector fields. In the GPU, using a compute shader, each vector is gradually reestablished, based on β until it is completely recovered.

The β is defined proportionally to an empiric parameter associated on the collider based on some factors (e.g., collider's mass or height relative to the ground). For example, the colliders associated with the body and tracks of the Tank (Fig. 4) may receive different β so the vectors in contact with these colliders may exhibiting different behaviors.

Besides the parameter associated with the collider, it is possible to expand the approach so that other conditions and rules are considered, such as the soil type, where β can be a higher value for muddy soils than for dry soils.

VI. Level Of Details

It is common to render batches of plants using a single draw call through the GPU-Instancing. The Discrete LOD (DLOD) approaches define the billboard's LOD for the entire batch, which is used to render all the positions contained on it. Because of this, DLOD approaches are highly efficient but are limited to render a single geometry by batch. Thus, to deform the plants (e.g., using splines), it is necessary to assign a geometry with high-level of detail to the entire batch. However, it is not suitable to render all positions of a *Cell* that have collisions using a fixed geometry, because plants that should not be deformed can be rendered through simple geometries. For those reasons, the visible *Cells* are selected and divided into two possibles pathways to define the billboard's LOD.

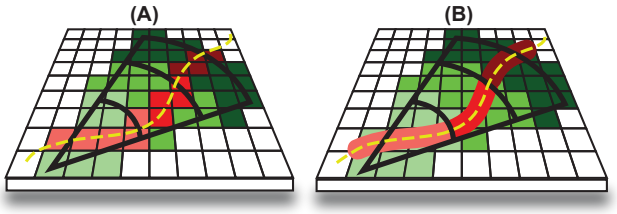

Figure 7. Comparative of the billboard's LOD distribution using only a traditional DLOD and (B) using our DLOD approach. Yellow represents a vector field deformation. Green *Cells* describe the LODs' distribution (that in our case we consider 3, shown by the color gradient) for plants that do not will be bend, and Red the LODs' distribution for plants that are bend. Plants placed in the Red areas are rendered through the dense billboard's geometry to achieve a smooth curvature for them, and in (B) it is possible to observe that the area with these geometries is reduced.

The first path is defined for the *Cells* that do not contain vector fields (7-A - Green *Cells*), and a unique billboard's LOD is assigned to render all plants (stored on *Plants_Buffer*) contained inside it. The billboards' LOD are determined by the distance between the *Cell* and the camera [5]. After the LOD definition, the plants of these *Cells* go straight to Render.

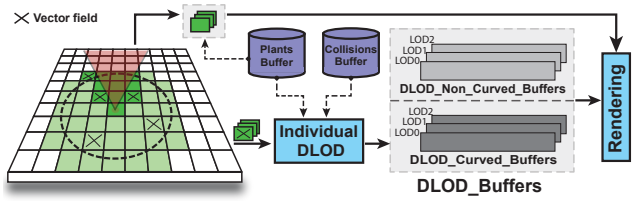

Figure 8. Selection of *Cells* to prepare for rendering. The *Cells* without vector field are rendered straight. Plants of the *Cells* with the vector field are rendered through the *DLOD_Buffers*, which define the plants' LOD.

The second path is defined for the *Cells* that contain vector fields (7-A - Red *Cells*). For these *Cells* are dispatched compute shaders to analyze and distribute each position in temporary buffers, called *DLOD_Buffers* (Fig. 8). In this case, the plants are rendered through these buffers, that are defined to render a billboard with a specific LOD; therefore, by directing the plant's data to a given *DLOD_Buffers* its LOD is defined.

The analyzed plants are distributed in 6 possible *DLOD_Buffers* (Fig. 8), three related to the plants' LOD that must be curved (*DLOD_Curved_Buffer*) and three to those that should not be curved (*DLOD_Non_Curved_Buffer*). The output buffer of each plant's is determined by two factors. 1) The plant's position is mapped to the vector field, resulting the group that each position will be inserted (*DLOD_Curved_Buffer* or *DLOD_Non_Curved_Buffer*). Once the group is established, 2) the LOD is defined through the distance between the plant's position and the camera. For each plant analyzed, the position and scale are stored in *DLOD_Buffers*. For deformed plants, the direction of folding is also stored in *DLOD_Buffers*.

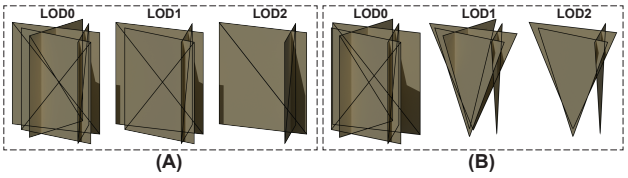

Figure 9. (A) Billboard's LOD generated by the removal of quads and (B) through the proposed improvement. In this example, the amount of triangles used to generate the three LODs is [8-6-4] for (A) and [8-4-3] for (B).

Image-based approaches use groups of quads randomly oriented to design the plants' textures, and LOD is generated through the discard of these quads (Fig. 9-A). Based on this, we present an improvement to generate the geometry's LOD used to render the plants. The improvement consists of generating the geometry of LOD1 by replacing the quads (of the geometry used by LOD0) by triangles (Fig. 9-B).

Through the proposed approach, it is possible to ensure further gains in the reduction of the geometries. Also, the popping between LOD0-LOD1 transition decreases, because the areas susceptible to the highest occurrence of popping are close to the ground, thus being partially or totally

occluded by billboards of the LOD0. The proposed optimization is only suitable for dense ground vegetation because, for lower densities or larger objects, there may be an increase in popping.

VII. Render

The rendering is done using GPU-Instancing, which is intended to instantiate a batch (several instances) of the same geometry using a single draw call. Each instance generated by the GPU-Instancing receives an ID, which is used to access the information stored in *Plants_Buffer* — for *Cells* without vector fields — or *DLOD_Buffers* — for *Cells* with vector fields assigned. Through the information obtained from the buffers, the *ObjectToWorld* matrix of each instance is set.

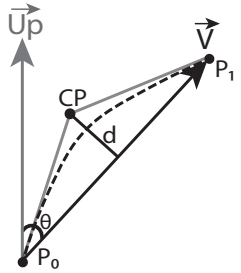

Figure 10. Description of Bézier's control points used to fold the billboard of the plants.

During the rendering, in the Vertex Shader stage, the folding of the plants — applied only to the *DLOD_Curved_Buffer* — is done using a Quadratic Bézier curve (Fig. 10). The Bezier's Control Points P_0 and P_1 are defined, respectively, by P_0 and P_1 of $\vec{V}$. The Control Point CP is defined by a perpendicular point to $\vec{V}$ and towards the $\vec{Up}$. The distance d (2), between CP and $\vec{V}$, is defined so that the maximum curvature is obtained when $\theta = 45°$ and minimum when $\theta = 0°$ or $\theta = 90°$. The curvature in $\theta = 90°$ is defined to describe the kneads on the plants and to prevent them from bending. D_{max} represents the maximum distance allowed between CP and $\vec{V}$ and is static for all vectors.

$$d = sin(2\theta) \cdot D_{max} \tag{2}$$

The displacement of the vertices is done by sampling the Bézier curve, where the interpolation parameter t is defined by the V coordinate in the billboard vertex's UV.

When turning a flat surface into a curve, their volume is distorted, requiring the application of volume conservation techniques. However, for the folding of plants, we consider the dealing unnecessary, since the ground vegetation are very dense structures and the distortions are not perceptible to the user, in addition to providing an efficient and straightforward rendering.

The wind animations applied to the plants were based on [9]. The proposed approach applies a directional global wind $\vec{W}_G$, which is varied in relation to its intensity and direction, together with a local directional wind $\vec{W}_L$, individual to each plant, used to cause small attenuations in $\vec{W}_G$. Using the attenuated $\vec{W}_G$, a displacement is applied to the vertices of the billboards, varying according to the plant scale and the V coordinate in the billboard vertex's UV, so that vertices close to the ground $(V = 0)$ are not displaced.

The plants' texturing is done by employing a Texture Array, where each plant instance receives an index to access the Texture Array. The index of each plant is defined during the rendering using a deterministic seed generated from the plant's position. The approach allows rendering multiple species of plants, using the same geometry, with a single draw call.

VIII. Memory Management

Given the amount of data needed to represent the plants, considering all buffers used, it is expected to have a significant storage demand. Due to this, memory management is made to discard the *Plants_Buffer* and *Collisions_Buffer* of *Cells* that 1) is not in contact with the colliders, 2) are not visualized for a time interval and 3) are outside the view range and view frustum.

The *Cells* that satisfy the conditions have their *Plants_Buffer* discarded. In contrast, *Collisions_Buffer* is only discarded if the entire vector field has already been restored; otherwise, these vector fields are compressed.

Before the compression, the vector field vector that requires the longest time to have its form reestablished is saved, and its recovery is maintained. When this vector is fully recovered, it means that the entire vector field is restored, and, consequently, it can be discarded. Afterward, the *Collisions_Buffer* is transferred from VRAM to the RAM so that it is compressed and stored in the *Compressed_Collisions_Buffer*.

When the *Render* or the *Collision* submodule demands a *Cell* that contains a compressed *Collisions_Buffer* this buffer is decompressed and returns to the VRAM. Next, *Collision Recovery* is applied regarding the time interval between the moment of compression and the current one. After that, the *Collisions_Buffer* is ready for use.

The compression of the vector fields is done through the Deflate algorithm, since it has high levels of compression and, mainly, excellent decompression times (Sec. IX-A). The cost to decompress becomes essential because data compression is only aimed at reducing storage costs, and can be done with Threads whenever the CPU is not fully loaded and over several frames. On the other hand, decompression is intended to apply new collisions in the vector fields or to be used by the rendering. Consequently, when requested, the vector fields must be decompressed quickly; otherwise, application performance may be compromised.

IX. RESULTS

The experiments were performed on an AMD Ryzen 5 3.6 GHz processor, with 16 GB of RAM and an NVIDIA GeForce GTX 1070 graphics card, with 8 GB VRAM. The proposed architecture was implemented in C# and HLSL languages using the Unity engine, however, it is a generic solution and can be implemented on any graphical API or integrated into any modern game engine.

The experiments were done in three scenarios generated from Digital Elevation Models (DEMs) with different dimensions (30kmx40km, 70kmx30km, and 110kmx60km). However, terrain size is not a limitation for our solution, as the cost of collision is attributed only to the number of colliders in the scenario and the density of the vector field. Also, the rendering cost is defined by the view range, plants' distribution density, and billboard's complexity to render them.

During the time measurements, costs with memory allocation as well as elements and characteristics of the scenario (e.g., trees, anti-aliasing, post-processing, and other) were disregarded.

A. Performance

For a better explanation of the performance results obtained, it is necessary to list common situations in games and simulators. A scene that renders plants at a distance of 150m from the observer accesses 190-210 *Cells* with 16x16m/frame. A vehicle moving in the same direction at 100km/h requires the system to create 25-30 new *Cells*/s, as well as a character that moves at 10km/h and causes the generation of 1-6 *Cells*/s.

Table I shows the processing and storage costs to distribute plants for different quantity of *Cells*, as well as with the different amount of instances. For the explored cases, the last line is highlighted as the worst-case — considering the view range of 150m — being executed during initialization or teleports.

Table I. Plant Distribution Times and Storage

Cells	Instances / m²							
	16		36		64		100	
50	0.32ms	3.1MB	0.38ms	7.0MB	0.48ms	12.5MB	0.54ms	19.5MB
100	0.55ms	6.2MB	0.65ms	14.1MB	0.71ms	25MB	0.77ms	39.1MB
150	0.88ms	9.4MB	0.98ms	21.1MB	1.07ms	37.5MB	1.18ms	58.6MB
200	1.24ms	12.4MB	1.31ms	28.2MB	1.52ms	50MB	1.81ms	78.1MB

Processing times for computing the collisions with vector fields of distinct densities are shown in Table II. In the experiments, the planar and spherical collider were dimensioned with an area of $10m^2$ ($360\vec{V}$) approximately.

Table II. Collisions Times (*ms*)

Vector Field Density ($\vec{V}$/m²)	Planar Colliders			Spherical Colliders		
	50	150	300	50	150	300
16	0.15	0.25	0.33	0.11	0.15	0.25
36	0.21	0.43	0.57	0.19	0.39	0.52
64	0.32	0.66	0.93	0.24	0.58	0.71
100	0.45	0.98	1.26	0.36	0.67	1.13

The Table III presents the average of times and the storage cost for vector fields, with different densities, before and after their compression. The experiment was performed with some colliders moving sparsely through the scenario, being concluded when an amount of 100 *Cells* was reached. (De)compression times and compression rates were measured using the LZ4 [10] and Deflate [11] algorithms.

Table III. Vector Field (De)Compression Analysis

Algorithm	Vector Field Density (V/m²)	Compression (ms)	Decompression (ms)	Required Memory (MB)	
				Decompressed	Compressed
LZ4	16	0.21	0.11	6,25	0,70
	64	0.65	0.35	25,00	2,61
	100	0.90	0.59	39,06	4,14
Deflate	16	0.92	0.18	6,25	0,38
	64	3.41	0.36	25,00	1,40
	100	4.45	0.61	39,06	2,07

As discussed in Section VIII, the decompression cost of vector fields is a crucial factor, so the LZ4 algorithm shows to be the most suitable for the problem since it presents lower decompression costs. However, if the compression of the vector fields can be done through Threads, the Deflate algorithm becomes more advantageous, since its decompression cost resembles the LZ4, and the compression rates are significantly higher. For these reasons, we chose the Deflate algorithm, with the compression being done through Threads.

At this point, teleport is also the worst-case, being necessary to decompress all vector fields of the *Cells* in the view frustum. In applications that this extreme case may occur, the need to use threads to decompress vector fields should be considered, and the decompression time can be masked by special effects (e.g., fade-in effect).

The Table IV shows the time required to define batches, through the *Individual DLOD*, for *Cells* that have vector fields. In these experiments, it was considered sets of visible *Cells* that have vector field assigned. Also, the different vector field densities were considered.

Table IV. Individual DLOD Times (*ms*)

Cells	Instances / m²			
	16	36	64	100
50	0.08	0.09	0.14	0.22
100	0.12	0.16	0.27	0.41
150	0.17	0.23	0.39	0.63
200	0.22	0.31	0.53	0.81

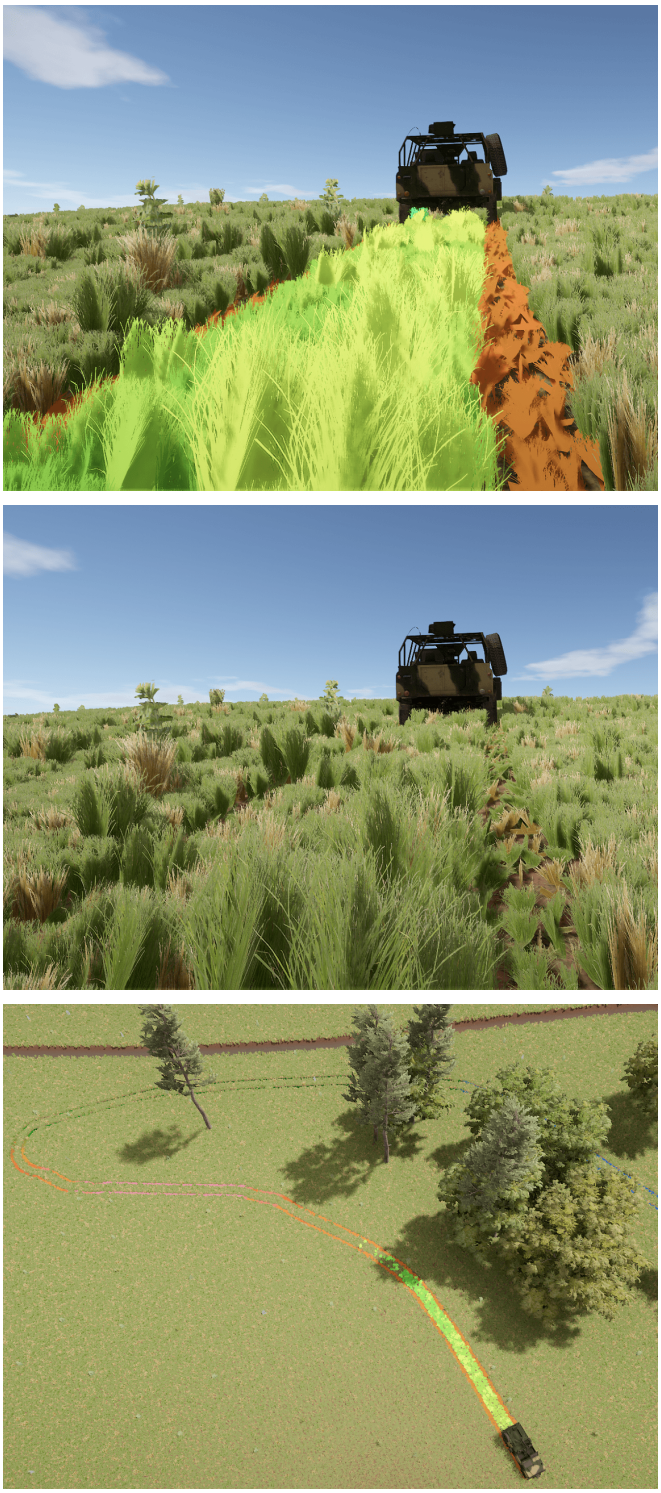

Figure 11. Example of an object that sets different costs for the plants' recovering based on the height to the ground of the colliders. The figures at the top and middle show that colliders associated with the vehicle's base define a lower cost to the vectors, and the wheels generate long-lasting deformations. The bottom figure shows that some plants that have collided with the vehicle's base have already recovered.

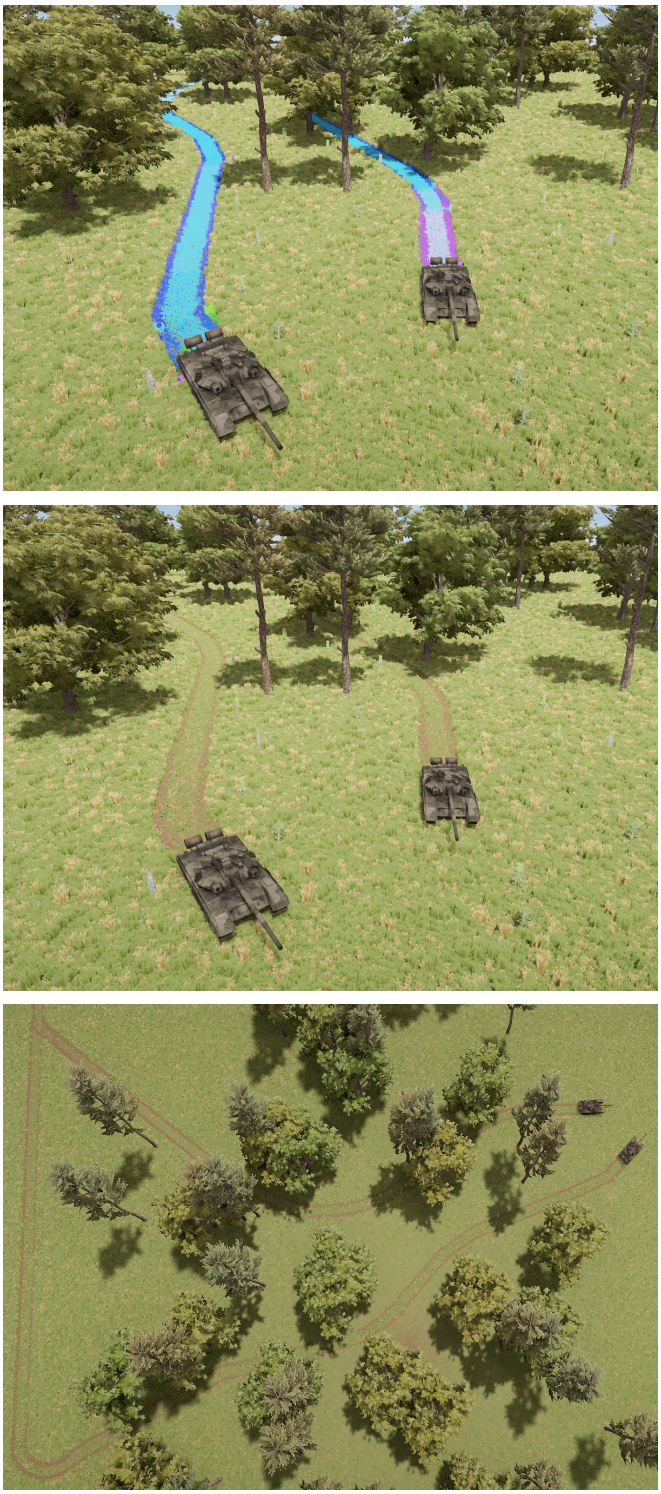

Figure 12. Example of a 3D object with colliders that define different costs to the plants' recovery, being defined by the elevation of the collider relative to the ground. In top and middle figures it can be seen that the collider of the Tank's base causes less deformation in the plants than the tracks. The bottom figure shows the same scene from a top view camera.

B. Visual Results

The visual results of the plants and collisions can be analyzed in Figures 1, 11, and 12. In these figures, the plants were defined with a density of 36 instances/m^2 and the vector fields with 36 vectors/m^2. The colors defined in some images represent the vector field vectors' direction. For better viewing, some images were captured with *Collision Recovery* turned off. Other visual results can be analyzed in the *video*.

X. Conclusion

In this article, we present a GPU-based approach for distribution, collision, and rendering of ground vegetation. The proposed approach ensures that the plants are distributed in real-time. The collisions are encoded in vector fields, and during the rendering, the deformations are applied to the plants. Moreover, the deformations generated in the vector fields may require different times to reestablish the original shape, thus generating different behaviors to the plants. In order to reduce the computational overhead during rendering, an *Individual DLOD* was proposed, as well as an improvement in the refinement of the geometry of the plants, maximizing the gains of the LOD technique. The performance of our solution is guaranteed by the high-level of parallelization in GPU used to perform the ground vegetation demands.

The main limitation found by the other solutions is the storage costs, which we solve using an architecture that supports managing individual parts of the scenario, as well as compressing the vector fields. This significant gain in storage allows us to generate long-lasting, or even permanent, deformations to the plants. As a result, the approach proves capable of handling ground vegetation on large-scale scenarios.

One drawback of our solutions is that when calculating collisions in parallel, it is not possible to simulate the resistance that the plants' bodies impose on the colliders.

As future work, it is possible to add deformations that are not generated only by the contact, for example, the deformation of the plants with the air displacement generated by the helicopters [2] [12]. Also, the rules used to distribute the plants were only defined to describe the distribution model, however, more promising works may be considered, for example, [13] proposes a highly parameterizable GPU-based approach using curve systems to distribute trees. Vector fields, after compressed, are stored in RAM, but in applications where this demand is big, it is possible to extend the approach to store the compressed vector fields on the Hard Drive.

Acknowledgment

We thank the Brazilian Army for the financial support through the SIS-ASTROS project.

References

[1] Z. Fan, H. Li, K. Hillesland, and B. Sheng, "Simulation and rendering for millions of grass blades," in *Proceedings of the 19th symposium on interactive 3D graphics and games*. ACM, 2015, pp. 55–60.

[2] K. Jahrmann and M. Wimmer, "Responsive real-time grass rendering for general 3d scenes," in *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 2017, p. 6.

[3] O. Jens, C. R. Salama, and A. Kolb, "Gpu-based responsive grass," 2009.

[4] K. Pelzer, "Rendering countless blades of waving grass," *GPU Gems*, vol. 1, pp. 107–121, 2004.

[5] H. Qiu, L. Chen, and G. Qiu, "A novel approach to simulate the interaction between grass and dynamic objects," *WSEAS Trans. Comput*, vol. 12, no. 7, pp. 277–287, 2013.

[6] K. Boulanger, S. N. Pattanaik, and K. Bouatouch, "Rendering grass in real time with dynamic lighting," *IEEE Computer Graphics and Applications*, vol. 29, no. 1, pp. 32–41, 2008.

[7] K. Jahrmann, Michael and M. Wimmer, "Interactive grass rendering using real-time tessellation," 2013.

[8] K. Chen and H. Johan, "Real-time continuum grass," in *2010 IEEE Virtual Reality Conference (VR)*. IEEE, 2010, pp. 227–234.

[9] B. Knowles and O. Fryazinov, "Increasing realism of animated grass in real-time game environments," in *ACM SIGGRAPH 2015 Posters*. ACM, 2015, p. 48.

[10] MiloszKrajewski, "LZ4." [Online]. Available: https://github.com/MiloszKrajewski/K4os.Compressio.LZ4

[11] L. P. Deutsch, "Deflate." [Online]. Available: https://tools.ietf.org/html/rfc1951

[12] C. Wang, Z. Wang, Q. Zhou, C. Song, Y. Guan, and Q. Peng, "Dynamic modeling and rendering of grass wagging in wind," *Computer Animation and Virtual Worlds*, vol. 16, no. 3-4, pp. 377–389, 2005.

[13] B. T. do Nascimento, F. P. Franzin, and C. T. Pozzer, "Gpu-based real-time procedural distribution of vegetation on large-scale virtual terrains," in *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE, 2018, pp. 157–15 709.