Interactive Grass Rendering Using Real-Time Tessellation

Klemens Jahrmann
Vienna University of Technology
klemens.jahrmann@net1220.at

Michael Wimmer
Vienna University of Technology
wimmer@cg.tuwien.ac.at

ABSTRACT

Grass rendering is needed for many outdoor scenes, but for real-time applications, rendering each blade of grass as geometry has been too expensive so far. This is why grass is most often drawn as a texture mapped onto the ground or grass patches rendered as transparent billboard quads. Recent approaches use geometry for blades that are near the camera and flat geometry for rendering further away. In this paper, we present a technique which is capable of rendering whole grass fields in real time as geometry by exploiting the capabilities of the tessellation shader. Each single blade of grass is rendered as a two-dimensional tessellated quad facing its own random direction. This enables each blade of grass to be influenced by wind and to interact with its environment. In order to adapt the grass field to the current scene, special textures are developed which encode on the one hand the density and height of the grass and on the other hand its look and composition.

Keywords

Rendering, Grass, Blades, Tessellation, Geometry, Wind, Real-time, LoD

1 INTRODUCTION

Grass rendering plays an important role for rendering outdoor scenes. Especially in virtual reality and computer games, natural scenes should include properly rendered vegetation. For trees and plants, many solutions already exist, but rendering grass has been a problem for a long time due to the high amount of needed geometry. This is why most often grass is rendered as a texture mapped to the ground or as transparent billboard quads. Both approaches have different unwanted artifacts: grass as a texture does not look good if it is shown from a small angle since there are no displacements of the ground, while billboards have problems when they are viewed from above because then they look very flat.

In this paper, we present a technique which is capable of rendering whole grass fields in real time completely as geometry. This can be achieved by using the tessellation shader as a fast geometry enhancement tool and some level-of-detail approaches, see Figure 1. The technique also uses special textures like density and terrain maps to pass important information to the tessellation stage.

Figure 1: Screenshot from our system.

2 STATE OF THE ART

Early grass rendering methods use only flat textures mapped onto the ground. A more sophisticated texture-based method was proposed by Shah et al. [6]. It uses a bidirectional texture function to animate the grass texture and displacement mapping for silhouettes, which produces a better look than a simple texture, but needs more time to render. Until today, almost every method still uses a texture-based technique for rendering grass that is far away from the camera. For grass that is closer to the viewer, different approaches have been developed: Orthmann et al. [5] present a grass-rendering technique which uses two billboards to represent a bunch of grass and which can interact with its environment. Rendering billboards can be done very efficiently on the graphics card but looks flat when viewed from higher angles. Also, billboards lack visual depth. To get deeper looking grass, Habel et al. [3] use half-transparent texture slices, which are placed

on a regular grid, but the visual artifacts for higher angles still exist. In order to solve this problem, Neyret [4] refers to existing fur rendering techniques using 3D textures and extends them to render high-detail geometry for large scenes like grass. This method allows him to render grass with good quality from different angles but still looks artificial. Zhao et al. [8] use the geometry shader to draw bunches of grass as geometry, but due to performance reasons, the generated grass field is very sparse.

Like in this paper, Wang et al. [7] render a whole grass field using geometry by drawing a single blade of grass multiple times. The blade is modeled with several vertices and a skeleton for the animation, but complex level-of-detail and culling aproaches have to be implemented to achieve reasonable framerates for rather sparse grass fields. Boulanger et al. [1] present a hybrid technique which uses the techniques mentioned before as different levels of detail. As geometric representation, they use a single grass patch consisting of blades of grass which are formed by the trajectories of particles during a preprocessing step. To get a large grass field, instanced rendering is used with a random orientation of the patch for each instance to reduce the grid artifacts which come with instancing. Grass that is further away is rendered using a 3D texture-based technique to draw a set of vertical and horizontal texture slices. For rendering high distances, only one horizontal slice is rendered, which leads to a texture mapped onto the ground.

3 OVERVIEW

In contrast to the methods mentioned before, the proposed technique renders *all* blades of a dense grass field completely as smoothly shaped geometry using hardware tessellation. In addition, each blade of grass can be individually deformed by forces like wind or objects.

First, we explain a basic technique suitable for smaller grass fields. The particular point of this technique is that it stores the geometric data of each blade of grass individually. This leads to a perfectly random and uniform grass field and has good performance, because many level-of-detail approaches can be applied to it. However, since each blade is stored by the application it does not scale very good for large scenes.

We then extend the basic technique using instancing: only a single grass patch is stored in memory and drawn multiple times, so that animated grass scenes of practically arbitrary sizes can be rendered in real time. A common handicap of instancing are the repeating patterns that normally appear, but since each blade is generated inside the shader, its look can be easily influenced by the world-space position, which conceals the transitions between the patches. In addition, by using instancing, we can simply add billboard flowers to the scene for visual enhancement.

Both methods require an initialization step to set up the textures (Section 4). We discuss the basic rendering technique in Section 5, and extensions to handle larger scenes, including instancing, levels of detail and antialiasing in Section 6.

4 INITIALIZATION

Our technique is based on a grass field which forms the boundaries of the space where grass is rendered. The field itself acts as a container for user-specified data like textures and parameters and is represented as a regular grid, where each grid cell contains a grass patch. The size of each grid cell has a significant impact on the performance, but the optimal size cannot be predefined in general, since it depends heavily on the overall scene and application. Each grass patch saves a vertex array object, which contains all necessary data to draw the blades of grass that grow on it. In addition, a bounding box is assigned to the patch to perform view-frustum culling.

The visual representation of the rendered grass field can be adjusted to match a high variety of possible scenes. It is determined by various textures, explained in Section 4.1, and by several parameters, which are listed below. During the initialization step, both the textures and parameters are evaluated to generate the input for the rendering process.

- *Density*: indicates how many blades of grass are initialized on a 1x1 unit-sized square.
- *Width*: the minimum and maximum width of a blade.
- *Height*: the minimum and maximum height of a blade.
- *Bending factor*: determines the maximum bending offset.
- *Smoothness factor*: specifies the maximum tessellation of a blade.

4.1 Special Textures

4.1.1 Density Map

The density map defines the density and the height over a grass field. Its red channel encodes the density value, which is directly used by the application to determine how many blades are generated. The green channel gives a subsampled version of the grass, which is later used when a less detailed model is needed. For example, when rendering water reflections, only the geometry near the coast lines need to be rendered. The blue

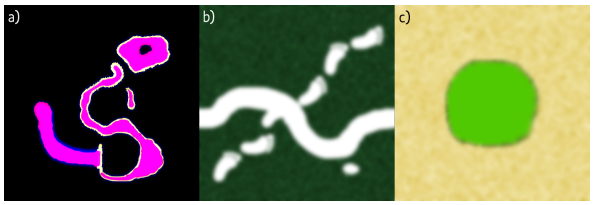

Figure 2: a) Shows an example of a density map, b) illustrates how a terrain map can be used (the white pixels are transparent) and c) shows a vegetation map of an oasis scene.

channel encodes the blades of grass' height at each position. If no special height differences are desired, the blue component can be created by blurring the red channel, which leads to a smoother transition between the regions where grass is rendered and where it is not. An example of a density map can be seen in Figure 2a.

4.1.2 *Terrain Map*

The terrain map itself does not change the appearance of a grass blade, but it determines the texture underneath the grass, which is also very important, since grass is naturally not perfectly dense and it is seen especially at sparse spots or tracks. Basically, the ground beneath the grass is organized in tiles just like the grass. Each tile has its own seamless diffuse texture assigned and the terrain map works as diffuse texture for the whole field. How a ground fragment is shaded is then determined by the alpha value of the terrain map. This can be seen in Equation 1, where c_r is the result color, c_{tt} the sampled tile texture, c_{tm} the sampled terrain map and a its alpha value. An example of a terrain map can be seen in Figure 2b.

$$c_r = c_{tt} \cdot (1-a) + c_{tm} \cdot a \tag{1}$$

4.1.3 *Vegetation Map*

A vegetation map is used if the grass color depends on its position inside a scene. If it is enabled, the color of the grass is determined by the color sampled from the vegetation map rather than from its diffuse texture. This can be used to simulate a desert scene with an oasis, where lush and healthy grass is placed near the water and dry grass further away. An example of a vegetation map can be seen in Figure 2c.

4.2 Creating Geometry

At the end of the initialization step, when all user-specified data has been read, the basic grass geometry that will serve as input to the rendering pipeline is generated for the high- and the low-detailed version of the grass field. For this, the density map is sampled and for each pixel, the world-space area which it refers to is calculated. The area is then multiplied with the red sample and the density value to determine how many blades have to be generated at this specific area. Then for each blade a random position is calculated and added to the high-detail list. If the density map's green sample value is also greater than zero, it is added to the low-detail list too. For each blade, an outline quad is generated with the following vertices:

$$\begin{aligned} w &= w_{\min} + R \cdot (w_{\max} - w_{\min}) \\ h &= (h_{\min} + R \cdot (h_{\max} - h_{\min})) \cdot \text{density}_b \\ \mathbf{p}_1 &= \mathbf{p}_c - [0.5 \cdot w, 0.0, 0.0] \\ \mathbf{p}_2 &= \mathbf{p}_c + [0.5 \cdot w, 0.0, 0.0] \\ \mathbf{p}_3 &= \mathbf{p}_c + [0.5 \cdot w, h, 0.0] \\ \mathbf{p}_4 &= \mathbf{p}_c - [0.5 \cdot w, h, 0.0] \end{aligned} \tag{2}$$

where w is the width and h the height of the blade with their maxima and minima $w_{\max}$, $w_{\min}$, $h_{\max}$ and $h_{\min}$, R is a randomly generated number between zero and one, density_b is the blue component sampled from the density map, $\mathbf{p}_c$ is the blade's center point and $\mathbf{p}_i$ is the i^{th} point of the quad. For higher performance, the height map can also be baked into the position, to reduce the texture lookups during the rendering process.

Apart from the vertex position, each vertex sends additional information to the graphics card: two-dimensional texture coordinates and the blade's center position, with a y-value of zero if it is a lower vertex or one if it is an upper vertex. Up to now, each blade of grass differs only in its height and its position. To make each blade unique, random values have to be generated and added to the graphics pipeline. For this, each of the transfered vectors is filled with random values until they are all four-dimensional, and an additional vector containing only random values is passed too. In total, there is one random value for the position and center position vector, two values for the texture coordinates and four additional ones, resulting in eight random values, which will all be needed in the rendering process.

5 REAL-TIME GENERATION

The rendering of the grass is done completely by the graphics card, which has to be able to execute tessellation shaders. If a grass patch is visible inside the view frustum, all included blades of the desired detail are drawn. Since each blade of grass is just a two-dimensional quad, backface-culling has to be disabled for full visual appearance.

5.1 Grass-Blade Generation

At first, each blade of grass is represented just by a random sized quad aligned on the x-y-plane, which is then tessellated into subquads using the tessellation

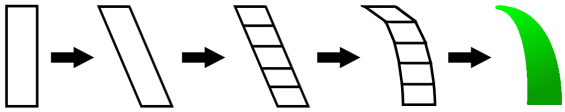

Figure 3: Illustration of the creation of a single blade of grass. The first step shows the input quad, the following steps show the results after the vertex shader, the tessellation control shader, the tessellation evaluation shader and finally the shaded blade of grass after the fragment shader.

shader. The resulting vertices are aligned on two parallel splines, which are formed by the upper and lower vertices of the input quad and two additional control points that determine the curvature of a blade. The final shape of the blade is determined by an alpha texture. The following will describe the path of the quad through each shader stage to become a nicely formed and shaded blade of grass. The procedure is also illustrated in Figure 3.

During the vertex shader stage, the blade's orientation is specified by applying a rotation around the blade's center position with one of the random values as angle. For the upper vertices, which are identified by their center position's y-value, an offset in x-z-direction is applied by using two other random numbers R_1, R_2 multiplied by the maximal bending factor b specified for the current grass patch. The calculation of the offset can be seen in Equation 3.

$$\text{offset} = \begin{bmatrix} b \cdot (2R_1 - 1) \\ 0 \\ b \cdot (2R_2 - 1) \end{bmatrix} \tag{3}$$

In the tessellation control shader, the distance between the blade and the camera position is computed. Using this distance together with a specified maximum distance, the tessellation factor is calculated by linearly interpolating the integer values between 0 and the given smoothness parameter s, which indicates the maximum tessellation of the blade. The interpolation can be seen in Equation 4, where d describes the distance to the camera and $d_{\max}$ the maximum distance.

$$\text{level} = \left\lceil s \cdot \left(1 - \frac{d}{d_{\max}}\right) \right\rceil \tag{4}$$

This value is applied as inner tessellation level and also as outer tessellation level for the right and left side of the quad. At and above the maximum distance, the tessellation level of zero culls a blade completely, and near the camera, the tessellation level of s subdivides the quad to s quads along the y-axis. An alternative approach for determining the tessellation level is to calculate the screen-space size of a blade and setting the level so that every subquad occupies the same amount of pixels. This is shown in Equation 5, where $\mathbf{u}_y$ and $\mathbf{l}_y$ indicate the y-coordinate of the upper and lower vertices of the quad, h_s is the vertical screen resolution and s is the smoothness parameter.

$$\text{level} = \min\left(\frac{((\mathbf{u}_y \cdot 0.5 + 0.5) - (\mathbf{l}_y \cdot 0.5 + 0.5)) \cdot h_s}{\#\text{ pixels per quad}}, s\right) \tag{5}$$

Two additional control points are generated in the tessellation control shader to determine the shape of the blade in the next step. For this, the lower points' x- and z-coordinate together with the upper y-coordinate can simply be taken, but in order to generate slightly differently shaped blades of grass, the coordinates are also varied by two random numbers. The variation can be seen in Equation 6, where $\mathbf{l} = (\mathbf{l}_x, \mathbf{l}_y, \mathbf{l}_z)$ and $\mathbf{u} = (\mathbf{u}_x, \mathbf{u}_y, \mathbf{u}_z)$ describe the lower and upper vertex and R_i are random values. In our application, R_1 lies between $-\frac{1}{4}$ and $\frac{1}{4}$ and R_2 between $\frac{3}{4}$ and $\frac{5}{4}$.

$$\mathbf{h} = \begin{bmatrix} \mathbf{l}_x \cdot R_1 + \mathbf{u}_x \cdot (1 - R_1) \\ \mathbf{l}_y \cdot R_2 + \mathbf{u}_y \cdot (1 - R_2) \\ \mathbf{l}_z \cdot R_1 + \mathbf{u}_z \cdot (1 - R_1) \end{bmatrix} \tag{6}$$

In the tessellation evaluation shader, the tessellated quad is shaped by aligning the vertices along two parallel quadratic splines. Each spline is defined by three control points. We use De Casteljau's algortithm [2] to compute each desired curve points $c(v)$, with v being the domain coordinate (equals $i/level$ for the i^{th} subquad) as parameter. The evaluation of one vertex on one spline can be seen in Equation 7 and is also illustrated in Figure 4a, where $\mathbf{p}_b$ and $\mathbf{p}_t$ indicate the bottom and top vertices and $\mathbf{h}$ is the additional control point. By using this recursive approach, the tangent $\vec{t}$ of a spline can also be found easily.

$$\begin{aligned} \mathbf{a} &= \mathbf{p}_b + v \cdot (\mathbf{h} - \mathbf{p}_b) \\ \mathbf{b} &= \mathbf{h} + v \cdot (\mathbf{p}_t - \mathbf{h}) \\ \mathbf{c}(v) &= \mathbf{a} + v \cdot (\mathbf{b} - \mathbf{a}) \\ \vec{t} &= \frac{\mathbf{b} - \mathbf{a}}{\|\mathbf{b} - \mathbf{a}\|} \end{aligned} \tag{7}$$

When both splines are computed, the final position and tangent can be interpolated using the domain coordinate v as parameter. The bitangent results directly from the two spline points. After tangent and bitangent are calculated, the normal can be computed by the cross product. This can be seen in Equation 8 and is also illustrated in Figure 4b.

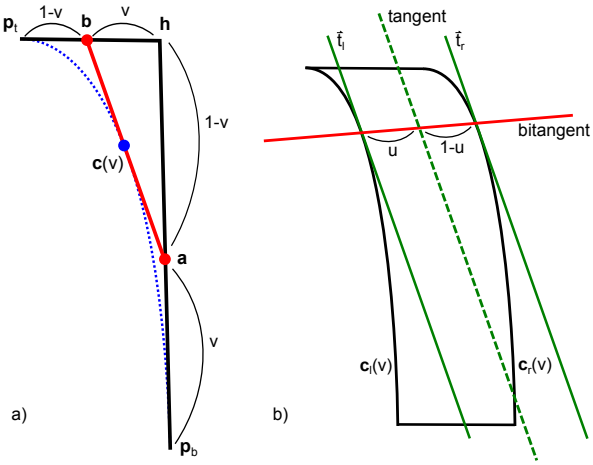

Figure 4: a) Illustration of De Casteljau's algorithm for calculating a point on a curve with the parameter v. b) shows how the tangent and bitangent of the blade of grass can be calculated.

Figure 5: Shows an example of an alpha texture (top) and a diffuse texture (bottom).

$$\begin{aligned} \text{position} &= \mathbf{c}_l(v)\cdot(1-u)+\mathbf{c}_r(v)\cdot u \\ \text{bitangent} &= \frac{\mathbf{c}_r(v)-\mathbf{c}_l(v)}{\|\mathbf{c}_r(v)-\mathbf{c}_l(v)\mathbf{a}\|} \\ \text{tangent} &= \frac{\vec{t}_l\cdot(1-u)+\vec{t}_r\cdot u}{\left\|\vec{t}_l\cdot(1-u)+\vec{t}_r\cdot u\right\|} \\ \text{normal} &= \frac{\text{tangent}\times\text{bitangent}}{\|\text{tangent}\times\text{bitangent}\|} \end{aligned} \tag{8}$$

In the fragment shader, the final shape of the blade is formed by the masking using the alpha texture. The diffuse color is then sampled either by the blade's diffuse texture or by the vegetation texture as has been explained in Section 4.1.3. In order to achieve a better variance, the components of the sampled color are modified slightly by three random values. Due to shadow effects, a blade of grass is normally darker near the ground and lighter at its tip, which can be simulated by multiplying the color with the vertical texture coordinate, leading to a better looking parallax effect. An example of an alpha and diffuse texture is shown in Figure 5.

5.2 Applying Forces

Until now, the grass is rendered nicely, but it is not animated and does not interact with its environment. In nature, grass is always in motion, either influenced by wind or by objects which also leave tracks. Wind can

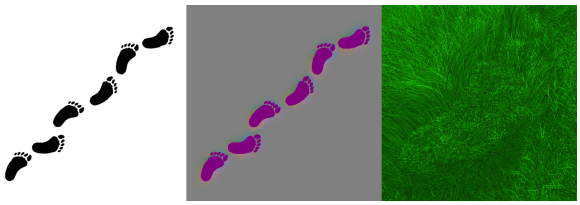

Figure 6: The left image shows an image of foot prints, the middle illustrates its corresponding force map and the right shows the rendered grass scene with the force map applied.

either be applied through a texture which is computed in some application-specific way, e.g. by a physical simulation, or it can be calculated using a wind function inside the shader. We implemented a wind function $w(\mathbf{p})$ that takes the world space position $\mathbf{p} = (\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z)$ and the current time t as parameter and calculates wind as two overlapping sine and cosine waves along the x-axis and a cosine wave along the z-axis. The function can be seen in Equation 9, where the constants c_i determine the shape of the wind waves together with a small number ε to avoid a division by zero. The result of the wind function can then be applied directly to the offset of the quad's upper vertices in the vertex shader.

$$\begin{aligned} w(\mathbf{p}) &= \sin(c_1\cdot a(\mathbf{p}))\cdot\cos(c_3\cdot a(\mathbf{p})) \\ a(\mathbf{p}) &= \pi\cdot\mathbf{p}_x+t+\frac{\frac{\pi}{4}}{|\cos(c_2\cdot\pi\cdot\mathbf{p}_z)|+\varepsilon} \end{aligned} \tag{9}$$

In addition to wind, each blade of grass can be influenced individually by external forces. For this, we use a force map which indicates the direction in which a blade is pushed at a certain position. In case of simulating tracks, the vectors of areas which have been fully under pressure by an object point towards the negative y-direction. At the boundary of an object, the force vectors point away from the object's center point, which can be approximated by the normal vector at the boundary. The vectors of the regions around an object also point away from its center, but with decreasing length. The size of the surrounding region depends on the impact of the applied force. An example can be seen in Figure 6. The sampled force-map vector can then be added to the offset of the quad's upper vertices.

6 HANDLING LARGE SCENES

6.1 Visibility

When dealing with large scenes, not all grass patches are visible in each frame. So the application has to determine which patches can be seen by a camera frustum, so that only visible blades of grass are transmitted to the graphics card. For this we implemented a simple hierarchical rasterization approach, where every two patches

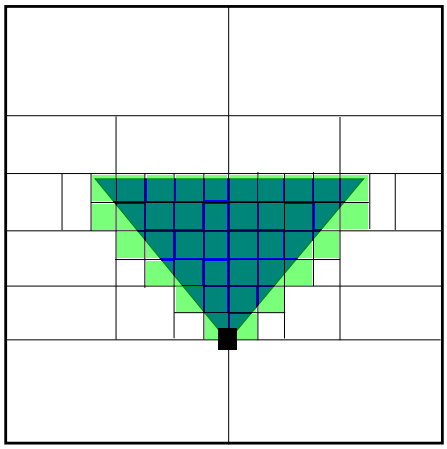

Figure 7: Illustration of the visibility test for the hierarchical rasterization approach.

are combined to one bigger patch until the grass field is represented by only one macro-patch. Then each frame the macro-patch is tested for visibility with the view frustum and if it is visible, its subpatches are tested iteratively. Figure 7 shows the visibility test for the hierarchical rasterization approach. The boxes representing the big patches are tested against the blue frustum and the green quads indicate the visible grass patches.

6.2 Instancing

As mentioned in Section 3, storing each single blade of grass in memory has hard limits regarding the maximum scale of a scene. Therefore, we implemented an instanced rendering technique, which only needs to store a single patch of grass as geometry data, in order to minimize the memory overhead of the scale of a scene.

When drawing grass patches by instancing, the basic grass-rendering method stays the same, but some adaptations have to be done. The grass field still needs to have a maximum size in order to generate lookup coordinates for the various global textures, but a tiling factor can be applied so that the textures do not need to scale with the scene if they are seamless. Instead of a list of patches, the grass field has to create just a single patch. The appropriate transformation matrices for the visible patch positions are calculated during the rasterization of visible patches. Then before each draw call, the transformation matrices have to be transmitted to the graphics card.

In the rendering process, only the vertex shader stage has to be changed. The vertex position has to be transformed by the given transformation matrix, and the density value has to be tested if the blade is visible. In addition, the height map has to be sampled, since it cannot be baked into the position anymore. These changes lead to an additional transformation and two additional texture lookups for each vertex of each blade, which is a certain overhead over the basic technique for smaller scenes.

Normally, when using instancing on patches, a repeating grid pattern appears over the scene, which is very disturbing. This effect can be partly hidden by rotating or mirroring the instances, like it is done by Boulanger et al. [1]. Since in our method, the blades are generated inside the shaders, the world-space position can be taken directly as variation factor of each random value R so that no grid pattern is visible without any rotation or mirroring. The influence of the world-space position on a random value can be described by one of the following Equations 10-12, where 11 as well as 12 can also be calculated using the z-coordinate or in 12, the cosine function can be used too. In these equations, $\mathbf{p} = (\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z)$ is the world-space position, $\vec{\text{dimension}} = (\text{dimension}_x, \text{dimension}_y)$ refers to the size of the grass field, c stands for any constant value and the function fract() gives the fractional part of a number.

$$R_{\text{new}} = \text{fract}\left(\frac{R \cdot \mathbf{p}_x}{\mathbf{p}_z}\right) \tag{10}$$

$$R_{\text{new}} = R \cdot \frac{\mathbf{p}_x}{\text{dimension}_x} \tag{11}$$

$$R_{\text{new}} = R \cdot \sin(c \cdot \mathbf{p}_x) \tag{12}$$

6.3 Level-of-Detail

Drawing each single blade of grass each frame has some disadvantages: First of all, it leads to a low framerate, and secondly, aliasing artifacts become visible the more blades of grass are drawn onto the same pixel. So several level-of-detail approaches have to be implemented to overcome these problems. As mentioned in Section 4, a highly downsampled version of a grass patch is generated for additional effects like reflections. Following this approach, more versions can be calculated at startup, each with fewer blades of grass. Then, for rendering, the distance to the grass patch determines the detail which is drawn.

This will then lead to step artifacts when the distance falls beyond a detail threshold. To achieve smooth transitions, more random blades of grass have to be culled the higher the distance gets. This takes place in the tessellation control shader, where the tessellation level is set to zero for some blades depending on one of its random values and the ratio between distance and maximum distance. Since the blades are normally distributed, hardly any artifacts are visible. In addition, the tessellation level of a blade of grass gets smaller the further away it is, because at far distances, there is no difference if the geometry of a blade is just a single quad.

In order to reduce the maximum amount of blades that are drawn, each blade that lies beyond a given maximal distance is culled by the tessellation control shader. This will result in a hard-edged circle of grass around the camera position. A smoother transition can be achieved by lowering the height of a blade of grass the further it is away. For these distance-driven approaches different distance functions were tested during the research process, and surprisingly a linear function yields the best results regarding performance and appearance.

6.4 Antialiasing

Although fewer blades of grass are drawn at higher distances, there are still aliasing and z-fighting artifacts visible. The reason is that it is impossible to render the perfect amount of blades for all distances and viewing directions. Nevertheless, the artifacts can be hidden by blurring the result image in consideration of the pixel's depth value. To maintain real-time performance, we use a separable 7x7 blur kernel with half-pixel sampling for a screen resolution of 1200x800. Since OpenGL's depth values go from one at the near clipping plane to zero at the far clipping plane, the depth value can be taken directly as center weight for the kernel function. The other kernel weights can either be the same small value, which leads to a simple average filter for higher distances, or small gradient values, leading to a Gaussian-like filter. The blur will then be processed in two separate passes, the first in horizontal and the second in vertical direction. For each pass, five samples are taken, the first at the pixel's center and the other at the border between the neighboring pixel and its neighbor, so that for a 7x7 kernel, only ten texture lookups are needed.

Blurring the image can reduce the aliasing artifacts only to a certain amount, but at farther distances, high-frequency noise can still be seen. So we developed another approach, which applies a penalty to the material of a blade of grass depending on its distance to the camera. This leads to a smooth darkening towards the horizon, which not only reduces the aliasing a lot, but also delivers a good sense of depth of a scene. Together with the distance blur, the artifacts can be reduced to a pleasing amount. To illustracte the impact of these effects, Figure 8 shows different result images.

7 RESULTS

For the performance evaluation, three scenes have been tested: First, a small oasis scene with reflections and very dense grass around the lake as well as dry and sparse grass further away. The second scene shows a meadow with complex objects and with a force map of huge foot steps applied. The last tested scene shows a large undulating grass field with flowers and the force map of the second scene using instanced rendering.

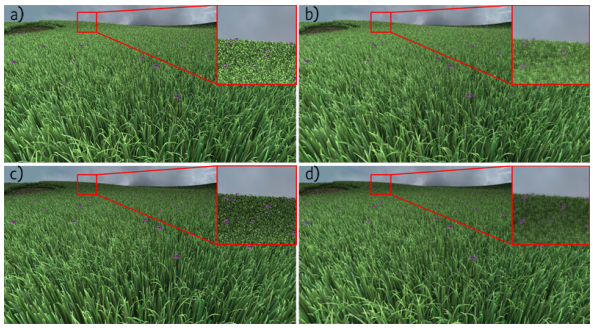

Figure 8: These result images show the effects of the distance blur and the depth darkening. Image a) is rendered with both effects disabled, b) is blurred without depth darkening, c) has depth darkening enabled but is not blurred and d) is rendered with both effects applied.

The application was tested on a Windows 7 64-bit system, with an 8-core Intel i7 processor at 3.8GHz with 16GB of RAM and an NVIDIA GeForce GTX 680 with 4GB of graphics memory.

In Table 1, the average frames-per-second together with the corresponding amount of blades drawn are measured for the three different scenes. It is tested with and without post-processing effects (the distance blur and the depth darkening) and with differently scaled grass fields. Some tests also contain additional geometry besides the grass field, which is stated in the column *Add. drawn*.

Scene	FPS	Vis. Blades	Scale	PP effects	Add. drawn
Oasis	75	136,650	160x160	enabled	-
Oasis	78	136,650	160x160	disabled	-
Oasis	65	330,816	320x320	enabled	-
Oasis	68	330,816	320x320	disabled	-
Meadow	84	348,370	300x300	enabled	-
Meadow	88	348,370	300x300	disabled	-
Meadow	83	348,370	300x300	enabled	Objects
Meadow	87	348,370	300x300	disabled	Objects
Meadow	81	340,716	600x600	enabled	Objects
Meadow	85	340,716	600x600	disabled	Objects
Instance	60	660,000	300x300	enabled	-
Instance	62	660,000	300x300	disabled	-
Instance	56	824,000	600x600	enabled	-
Instance	58	824,000	600x600	disabled	-
Instance	55	1,008,000	2,000x2,000	enabled	Flowers
Instance	57	1,008,000	2,000x2,000	disabled	Flowers
Instance	55	1,008,000	4,000x4,000	enabled	Flowers
Instance	57	1,008,000	4,000x4,000	enabled	Flowers
Instance	55	1,008,000	8,000x8,000	enabled	Flowers
Instance	57	1,008,000	8,000x8,000	enabled	Flowers
Instance	55	1,008,000	90,000x90,000	enabled	Flowers
Instance	57	1,008,000	90,000x90,000	enabled	Flowers

Table 1: Test results of three different scenes. Scene Oasis containing water reflections and a vegetation map and scene Meadow containing highly detailed objects are rendered both with the basic technique. Scene Instance is rendered using the instanced rendering approach together with billboard flowers.

For the results we tried to position the camera for worst-case scenarios where most blades of grass are drawn. The corresponding images can be seen in the

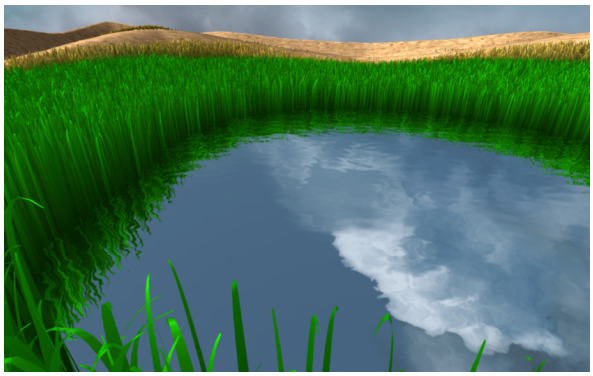

Figure 9: The rendered image of the Oasis scene.

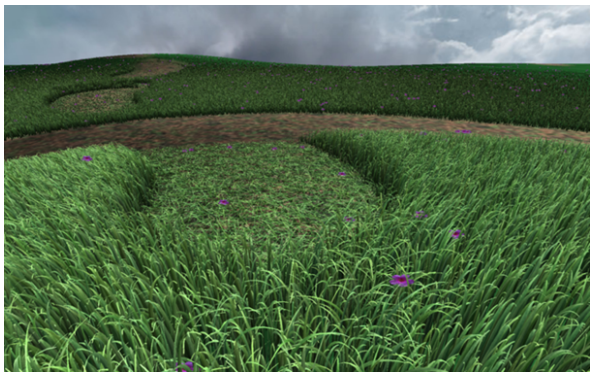

Figure 11: The rendered image of the Instance scene.

Figure 10: The rendered image of the Meadow scene.

Figure 12: The grass-rendering technique applied to a wheat field.

Figures 9, 10 and 11. As stated in earlier sections, for smaller scenes the highly optimized version is superior to the instanced version, but the big advantage of the instanced version is that its performance stays constant with the amount of blades drawn regardless of the size of the overall scene. The only limitation which the size of the scene has lies in the memory used for the grass field's hierarchical rasterization structure. The difference in the performance between the basic and the instanced method can be seen at the 300x300 and 600x600 sized instances. Even when drawing complex objects, the optimized basic version is clearly faster than the instanced version. However, the basic version is not able to render fields greater than 1000x1000, because then the application's memory is full.

A numerical comparison of the performance between the presented technique and the related papers mentioned in Chapter 2 cannot be done correctly, because most of the related techniques use static level-of-detail and image based methods, whereas our method only uses dynamic level-of-detail and geometry. The main advantage of the presented algorithm is its flexibility, so that it is extensible and can easily fit to any possible scene. This is because in each frame, the blades are procedurally generated inside the shader. With our method, also grass-like scenes like for example a wheat field can be rendered, which is shown in Figure 12. In addition, grass rendered with the proposed technique can be used in physically correct simulations, since each blade of grass can be influenced by its environment seperately.

Compared to Boulanger et al. [1], our technique enables each single blade to be animated and influenced by forces at all distances, since the grass field is completely rendered as geometry. Wang et al. [7] can also animate each single blade of grass seperately, but due to the high amount of vertices which have to be stored for a scene, the grass field can only be sparse in order to achieve interactive framerates. The difference between our method and the related works can be seen in Figure 13 and Figure 14.

The mentioned scenes and effects can also be seen in motion on the accompanying video and a more detailed view of the grass and flowers can be seen in Figure 15.

8 CONCLUSION AND FUTURE WORK

Although rendering dense grass fields completely as geometry seemed to be impossible for a long time, this paper introduced a quite simple method which is able to draw large grass scenes in real time using the tessellation shaders as geometry enhancement tool. One of the main advantages is its flexibility regarding different

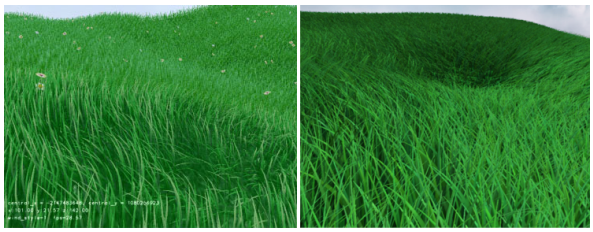

Figure 13: The left image shows a picture of a grass field with a tornado-like wind taken from [7] and the right one shows a similar scene rendered with our method, where the tornado is simulated with a forcemap.

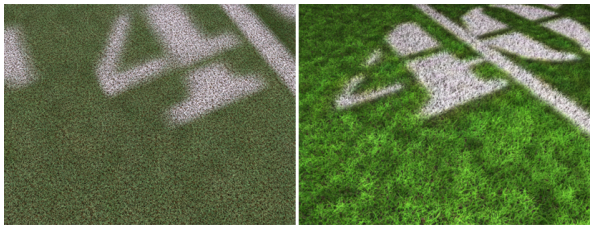

Figure 14: The left image shows a picture of a football field taken from [1] and the right one shows a similar scene rendered with our method.

Figure 15: A more detailed view of the grass and the flowers.

scenes, since the blades of grass are procedurally generated inside the shader. Following this aspect, many extensions can be made to the proposed technique. For example, a life-time model can be implemented, making grass grow over time, or the grass can adapt itself to its environment, like growing along the shape of objects. Until now the grass can only grow in y-direction, but it might be simple to give each blade of grass a direction, so that grass can grow on arbitrary locations. For example, grassy objects or roots of underground scenes can be rendered using this approach.

In the technique as presented, each blade of grass can be influenced by forces, but the animation and interaction is just a simple simulation. A physically based wind and bending model with collision detection, like in [7], has to be implemented for more realistic looking scenes.

9 REFERENCES

[1] K. Boulanger, S.N. Pattanaik, and K. Bouatouch. Rendering Grass in Real Time with Dynamic Lighting. *Computer Graphics and Applications, IEEE*, 29(1):32–41, 2009.

[2] Gerald E. Farin and Dianne Hansford. *The essentials of CAGD*. A K Peters, 2000.

[3] Ralf Habel, Michael Wimmer, and Stefan Jeschke. Instant Animated Grass. *Journal of WSCG*, 15(1-3):123–128, 2007.

[4] Fabrice Neyret. A General and Multiscale Model for Volumetric Textures. In *Graphics Interface*, pages 83–91. Canadian Human-Computer Communications Society, 1995.

[5] J. Orthmann, C. Rezk-Salama, and A. Kolb. GPU-based Responsive Grass. *Journal of WSCG*, 17:65–72, 2009.

[6] Musawir A. Shah, Jaakko Kontinnen, and Sumanta Pattanaik. Real-time rendering of realistic-looking grass. In *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '05, pages 77–82, New York, NY, USA, 2005. ACM.

[7] Changbo Wang, Zhangye Wang, Qi Zhou, Chengfang Song, Yu Guan, and Qunsheng Peng. Dynamic modeling and rendering of grass wagging in wind: Natural phenomena and special effects. *Comput. Animat. Virtual Worlds*, 16(3-4):377–389, July 2005.

[8] Xiangkun Zhao, Fengxia Li, and Shouyi Zhan. Real-time animating and rendering of large scale grass scenery on gpu. In *Proceedings of the 2009 International Conference on Information Technology and Computer Science - Volume 01*, pages 601–604. IEEE Computer Society, 2009.