Real-Time Grass (and Other Procedural Objects) on Terrain

Dimitris Papavasiliou

Figure 1. Two kinds of objects scattered over a large terrain area. One is regular grass, which covers areas of the terrain that appear green, while the other consists of simple gravel-like particles which cover brown areas.

Abstract

Dense vegetation is ubiquitous in nature, but challenging to render due to its great geometric complexity. This paper presents a simple approach to vegetation rendering, based on modeling each plant separately using tessellation hardware in modern GPUs. To reduce scene complexity with as little loss in image quality as possible, we use continuous level-of-detail techniques. We do this both at the level of individual plants and the whole distribution of plants on the terrain. Additionally, we describe how to control the distribution of plants over the terrain, using a technique that does not require any additional maps, other than the terrain texture map, and supports multiple species of vegetation with little overhead. In order to leverage the flexibility afforded, we present a method of using separate tessellation and geometry programs for each vegetation species, which enables the efficient rendering of diverse vegetation, or any other objects, provided that their geometry can be generated through tessellation and geometry shaders on the GPU. The basic technique is applicable to any static triangulated mesh, but it can be adapted to the dynamic meshes typical in terrain applications. Details are given for the implementation on ROAM-based terrain.

1. Introduction

Grass is characterized by a relatively simple geometric structure compared to other kinds of vegetation, such as shrubs or trees, but this advantage is offset by the sheer number of separate plants found on even moderately sized areas of terrain. The brute-force approach of rendering grass by covering the whole terrain with individual models of grass blades is therefore not practical if we want interactive performance on all but the smallest terrain sizes.

One way of coping with the geometric complexity of grass is by approximating it using image-based techniques. The simplest approach is to rely on a simple texture covering the terrain, but the use of billboards [Pelzer 2004] is also very common. Other methods have been inspired by the field of volume rendering, for instance by using methods based on shell rendering [Bakay and Heidrich 2002].

Although these techniques are effective in reducing the complexity of the problem, the resulting image is of lower quality, missing or incompletely capturing important cues, which are indispensable for realistic results. Examples of such cues include variation of the height of the canopy due to variations in the height and shape of each plant, partial occlusion of objects placed inside the canopy, the parallax effect and surface shading details, which cannot be adequately captured in a texture.

Consequently, researchers [Perbet and Cani 2001; Boulanger et al. 2009; Qiu et al. 2012] were led to attempt to combine different methods based on the distance from the observer, exploiting the fact that the bulk of vegetation visible in a typical scene lies far away from the viewer, where rendering quality tends to diminish in importance. Indeed, due to the foreshortening effect of perspective projection, low and dense vegetation, such as grass and flowers, tend to appear like a flat canopy of relatively uniform texture at even moderate distances away from the viewer. A simple texture is therefore enough to render the perceivable detail at such distances. Closer to the viewer on the other hand, separate plants begin to become distinguishable, so that more complex representations are required to render vegetation convincingly. A typical approach involves the use of a geometry-based representation of each individual plant close to the viewer, followed by some sort of volumetric or billboard representation for intermediate distances and a simple texture for large distances.

The technique discussed below is the result of an attempt to dispense with the additional complexity of using multiple methods of representation. Instead we rely on the existing terrain texture for distant terrain and use a geometry-based approach for close and intermediate distances. To reduce the volume of geometry that would be required to densely cover near terrain, continuous level-of-detail techniques are employed to vary the density of the population of plants as well as the geometric complexity of each individual plant with respect to distance from viewer. To ensure the seamless blending between the rendered plants and the terrain as plant density falls off at larger distances, the terrain texture is used to determine both the plant

distribution and the dominant color of each plant.

Schematically the process of rendering a terrain with vegetation includes the following steps:

1. Generate terrain geometry, cull it to the view frustum and render it.

2. Distribute individual plants over the surface of the visible terrain in a uniform and natural manner and with variable local density of vegetation, depending on the underlying terrain type, as well as viewer location and orientation.

3. Generate and render a geometric model for each separate plant with adequate variability in the shape, size and color to ensure a natural appearance of the overall canopy.

We present the algorithm in its most basic form, which can operate over any static triangulated surface, in section 2. Since this basic algorithm covers the whole mesh with vegetation, it is only practical for very small areas. In section 3, we address this problem by proposing modifications and optimizations on the basic algorithm to adapt it to large terrains.

The basic idea consists of using the ROAM algorithm in order to utilize its view frustum culling mechanism to cull grass as well. We provide the details of one way to adapt the algorithm to the dynamic mesh generated by the ROAM algorithm, pointing out several optimizations which are necessary to make the process more efficient. Similar approaches should be able to adapt the basic algorithm to other complex terrain tessellators. To illustrate the point, we provide a sketch for the case of geometry clipmaps in the last section.

Finally we also cover other improvements, such as shadows cast onto the terrain, as well as a set of techniques that facilitate support for multiple and geometrically diverse kinds of vegetation or other objects that can be tessellated on the GPU. Examples of such objects include gravel, rocks or woodland debris, such as twig fragments. More complex objects, can be approximated through billboards.

2. Outline

We will assume, during this discussion of the basic outline of the algorithm, that our terrain geometry is already available in the form of a triangle mesh which must be covered with vegetation.

2.1. Seeding the Terrain

We begin with the distribution of plants over the terrain mesh by sampling points, or "seeds", on the terrain triangles. The sampling density, which does not need to be

constant over the whole surface, is critical as it presents a trade-off between image quality and computational load.

In general, the number of plants required to achieve adequate coverage depends on the triangle's size and its orientation and position with respect to the viewer. We determine the number of plants to seed on a terrain triangle by calculating its screen-space area and multiplying it by a user-adjustable density. We also clamp the result to a maximum value to avoid excessive plant densities close to the viewer and because it allows greater flexibility in controlling the plant distribution. The plant count for each terrain triangle therefore becomes

$$n = \min\left(\rho A_{\mathrm{s}}, n_{\max}\right), \tag{1}$$

where $n_{\max}$ is the user-chosen plant count ceiling, ρ is the plant density, and A_{s} is the projected screen-space area of the terrain triangle.

A problem arises in the case of triangles that cross the near plane, which would require clipping for proper area calculation. To avoid this complication, we can observe that these triangles are, by definition, very close to the viewer and hence would probably result in plant counts close or equal to the maximum value. We therefore simply set the plant count equal to this maximum value when we detect that any of the vertices lie behind the near plane.

The screen-space area of the triangle is a simple and robust metric and we can calculate it efficiently, as we will show in section 3.1, but if it proves computationally expensive, it can easily be replaced. We could, for example, calculate the plant count based on the world-space area of the terrain triangle as

$$n' = \min\left(\frac{\rho A_{\mathrm{w}}\left(\hat{n}\cdot\frac{\vec{v}}{\|v\|}\right)}{\|v\|^2}, n_{\max}\right), \tag{2}$$

where ρ and $n_{\max}$ are defined as before, A_{w} is the world-space area of the triangle, $\vec{v}$ is the vector from the observer to the triangle center, and $\hat{n}$ is the triangle normal.

2.2. Sampling Seed Locations

The geometry for each plant will be generated entirely on the GPU based on a few parameters typical for the plant species. We therefore only need to send the three vertices of the terrain triangle it grows on, from which we calculate the plant root position by uniformly sampling the surface of the triangle using the equation

$$p = (1-\sqrt{r_1})\vec{v_0} + \sqrt{r_1}(1-r_2)\vec{v_1} + \sqrt{r_1}r_2\vec{v_2}. \tag{3}$$

Here, p is the plant root position, $\vec{v_i}$ is the ith triangle vertex, and r_1, r_2 are random variables uniformly sampled within $[0, 1]$.

Due to the sequential nature of traditional random number generators their use is not practical on a GPU. Conceptually, what we need is a hash function $h(i, c) \rightarrow [0, 1]$ which we can use to calculate a random number, by plugging in the plant index and a number that is unique for the current triangle.

The process is reminiscent of cryptography where a message and a key is mapped to a random result and indeed cryptographic hashes present a method of producing repeatable random numbers, that is very well suited for implementation on a GPU. Salmon et al. [2011] describe several such generators with different statistical qualities and requirements in terms of time and space. Our demands are modest, so the simplest among these is adequate to ensure seemingly random plant distribution.

We use the Philox-2×32-3 generator [Salmon et al. 2011] to calculate two random numbers for equation 3 based on the x and y coordinates of the triangle center and the plant index. We also use the generated values to seed a simple LCG random number generator in order to produce additional random values to be used when calculating other plant parameters.

2.3. Growing Plants

The geometry of each plant is generated using the tessellation hardware present in modern GPUs. We will confine this discussion to grass or grass-like vegetation, although the techniques proposed in sections 3.3 and 3.5 support multiple plant species with little overhead. Each plant will therefore be represented by a quadrilateral strip, shaped and textured according to plant species. More complex, but still relatively planar, shapes can be approximated by using an appropriate alpha mask if necessary.

The tessellation hardware found in GPUs consists of three serial stages. In the first stage we create a patch consisting of a single vertex, the root position calculated previously, and direct the next stage to subdivide it into a single line strip. The number of segments in the strip is chosen as

$$n = \frac{n_{\max}}{\max\left(\|\vec{v}\|, 1\right)^2}, \tag{4}$$

where n is the number of segments for the current strip; $n_{\max}$ is the maximum number of segments, which is a user-selectable parameter that determines the level of detail of the plant geometry; and $\vec{v}$ is the vector from the viewer to the plant's root. We clamp the distance in order to avoid divisions by zero and to ensure a band of maximum resolution plants near the viewer.

In the last stage the generated line strip is shaped. The plants are modeled as a series of rigid segments joined via spring-loaded hinge joints (see figure 2) which allows us to capture the shape of many different species of small plants, such as flowers or grass blades, by selecting proper values for the stiffness of the springs. We will derive the equations of static equilibrium for the second segment of the three segment model shown in figure 2, as that is the most general case.

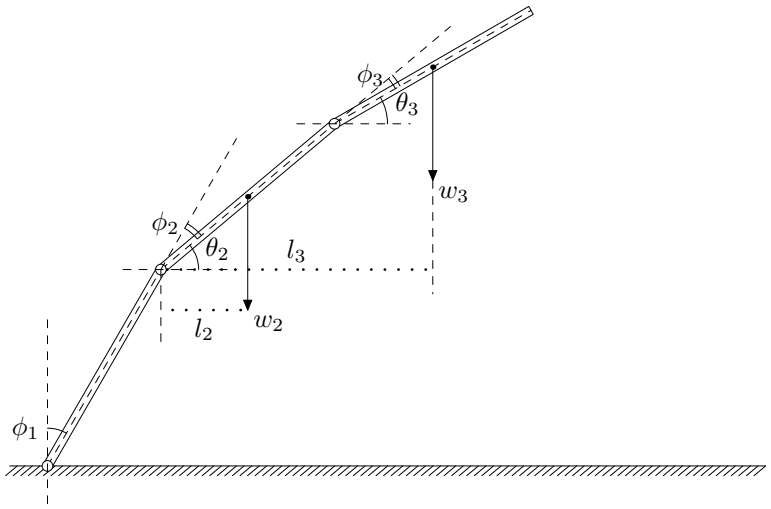


Figure 2. A single blade of grass bending under its weight. The blade is made up of a number of rigid segments of mass m and height h, connected via spring-loaded hinges with constant k (shown with hollow circles).

The torque acting on the second joint due to gravity is

$$\tau_{\mathrm{g}} = w_2\, l_2 + w_3\, l_3. \tag{5}$$

Here, w_2, w_3 are the weights of the second and third segments. These exert a torque on the spring at the second joint, deflecting the spring. Variables l_2, l_3 are the respective lever arm lengths. We assume that the material of the segments is homogeneous, so

$$w_2 = \rho h g. \tag{6}$$

Variable ρ is the linear density of the material of the segments, h is the height of each segment, and g is the acceleration of gravity. Since the material is homogeneous the center of mass lies at the center of each segment, so l_2 is equal to $\frac{h}{2}\cos\theta_2$, while l_3 is equal to $h\left(\cos\theta_2 + \frac{\cos\theta_3}{2}\right)$.

The torque exerted by gravity therefore becomes

$$\tau_{\mathrm{g}} = \rho h^2 g \left(\frac{3}{2}\cos\theta_2 + \frac{1}{2}\cos\theta_3\right). \tag{7}$$

It is balanced by the torque from the spring at the second joint, which is

$$\tau_{\mathrm{s}} = -k\phi_2. \tag{8}$$

Since the segment is in a state of equilibrium, the sum of all torques must vanish. This gives

$$\rho h^2 g \left(\frac{3}{2}\cos\theta_2 + \frac{1}{2}\cos\theta_3\right) - k\phi_2 = 0. \tag{9}$$

We rewrite the above equation solely in terms of ϕ_i angles by noting that $\theta_2 = \frac{\pi}{2} - (\phi_1 + \phi_2)$, $\theta_3 = \frac{\pi}{2} - (\phi_1 + \phi_2 + \phi_3)$ and using the trigonometric identity $\cos(\frac{\pi}{2} - \phi) = sin(\phi)$ to arrive at

$$\rho h^2 g \left(\frac{3}{2} \sin(\phi_1 + \phi_2) + \frac{1}{2} \sin(\phi_1 + \phi_2 + \phi_3) \right) - k\phi_2 = 0. \tag{10}$$

Finally, we divide both sides by $\rho h^2 g$ and set $k' = {}^{k}/_{\rho h^2 g}$, so that we only have one parameter to adjust. This yields the equation

$$\frac{3}{2} \sin(\phi_1 + \phi_2) + \frac{1}{2} \sin(\phi_1 + \phi_2 + \phi_3) - k'\phi_2 = 0. \tag{11}$$

The analysis for the other segments is similar. The general case of the nth segment in a blade made up of N segments is therefore described by

$$\sum_{i=n}^{N} \left[\sum_{j=1}^{i-1} \sin\theta'_j + \frac{\sin\theta'_i}{2} \right] - \sum_{i=1}^{n-1} \sin\theta'_i - k'\phi_k = 0, \tag{12}$$

where θ'_n is the complementary angle of θ_n and is equal to $\sum_{i=1}^{n} \phi_i$. By numerically solving the non-linear system of equations describing each segment of the plant for various values of the parameter k', we obtain the plant shapes shown in figure 3.

We precalculate the shape for a normalized plant of unit density and segment height and store it in a 2D texture map where each texel column stores the height and

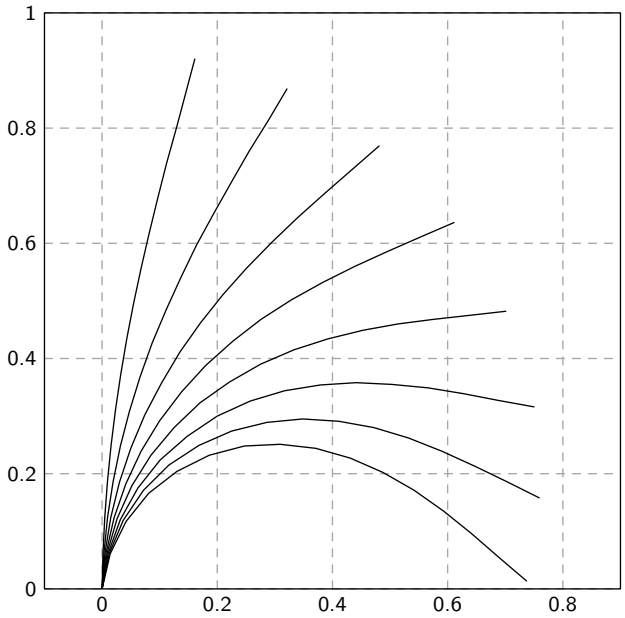


Figure 3. A plot of the blade shapes calculated by solving the system of equations 12 for various values of k'.

lateral deflection of the endpoints of each segment, as well as the segment inclination, for some given value of the parameter k'. At runtime, we then sample this texture using the plant's stiffness and line strip segment index and use the height and deflection to position the segment's endpoints.

We furthermore rotate the strip around the vertical axis to change its orientation and scale it to determine its height. The orientation angle can either be sampled uniformly from the range $[0, 2\pi)$ or we can use a unimodal distribution centered around the direction opposite to the direction of sunlight if the plant species exhibits strong phototropic behavior.

Plant height and stiffness are also chosen randomly from some uniform distribution, with minimum and maximum values depending on the plant species. Values resulting in very small plants should be avoided as these will typically be occluded by larger neighbors without contributing to the perceived density of the canopy. Similar considerations apply to very low stiffness values, which result in low, drooping plants that are occluded by taller neighbors.

We pass the calculated strip vertices to the geometry shader along with a tangent and normal vector, which can be calculated from the sampled deflection angle and the orientation angle. The geometry shader then simply generates a triangle strip by extruding the line strip vertices along the tangent vector and passes it on to the fragment shader along with the calculated normal vector. The width at the base of the strip can be chosen randomly within some bounds and by shortening it towards the top we can shape the rectangular strip into a grass blade.

2.4. Shading

A typical vegetation canopy can take up half of the screen and many pixels are filled more than once due to the high degree of occlusion between neighboring plants. Furthermore, the canopy usually consists of small objects which, being scattered on the ground, lie typically relatively far away from the viewer. We therefore find it neither practical nor essential to employ an accurate shading model. A simple approximation should suffice, while saving GPU cycles, as long as we get the basics right.

Our model is based on the Phong model, where we add terms for the occlusion of light due to neighboring plants as proposed by Reeves and Blau [1985], as well as for the translucency of a plant's surface. A simple Lambertian model is used for the latter, so that the illumination at each point of a plant's surface due to the transmission of incident light is given by

$$I_\mathrm{t} = \begin{cases} I_\mathrm{i} k_\mathrm{t} \max\left(-\hat{n}\cdot\hat{\omega}_\mathrm{i}, 0\right) \exp\left(\frac{\alpha}{\hat{n}\cdot\hat{\omega}_\mathrm{i}}\right) & \text{if } \hat{n}\cdot\hat{\omega}_\mathrm{i} < 0 \\ 0 & \text{if } \hat{n}\cdot\hat{\omega}_\mathrm{i} \geq 0, \end{cases} \tag{13}$$

where $\hat{n}$ is the local surface normal vector, $\hat{\omega}_\mathrm{i}$ is the vector from the point to the the light source, and I_i is the incident light intensity. The coefficient k_t determines the

color of the transmitted light, while the exponential factor accounts for the attenuation of the light beam due to absorption or scattering inside the plant. This is determined by the attenuation coefficient α and the length of the path traveled by the beam inside the plant. We model the plant as a slab of unit width, so that the latter is equal to the reciprocal of $-(\hat{n} \cdot \hat{\omega}_\mathrm{i})$.

Combining equation 13 with the equation describing the Phong model allows us to calculate the total illumination coming from the shaded surface point as

$$I_\mathrm{o} = I_\mathrm{a} k_\mathrm{a} + I_\mathrm{t} + I_\mathrm{i}\Big(k_\mathrm{d} \max(\hat{n} \cdot \hat{\omega}_\mathrm{i}, 0) + k_\mathrm{s} \max(\hat{v} \cdot \hat{r}, 0)^n\Big). \tag{14}$$

Here, I_a denotes the ambient light intensity, $\hat{v}$ and $\hat{r}$ are vectors pointing in the directions of the viewer and pure specular reflection respectively, while k_a, k_d and k_s are the ambient, diffuse and specular reflection coefficients. The exponent n determines the specularity of the surface.

The latter, along with k_t in equation 13 above, determine the color of the plant surface and we calculate them as a function of the color of the terrain under the plant. This ensures a seamless transition into the unseeded regions of the terrain that lie further away from the viewer. A simple way of determining these coefficients is by scaling the terrain color by a coefficient ranging from 1 to some lower value, as the height of the shaded point above the ground approaches zero [Reeves and Blau 1985]. We also modulate by a detail texture to render plant surface detail if viewpoints close to the ground are expected.

Finally, we note that we can render plants with more complex shapes by using a texture containing an alpha mask coupled with alpha testing to reject all pixels of the quadrilateral strip that lie outside the mask. The edges of the plant can be anti-aliased by using multisample anti-aliasing and converting the value of the alpha mask to a coverage mask for each pixel. This approach avoids the cost of sorting each plant by depth, which would be prohibitive, given the large number of plants in each scene.

3. Implementation Details and Optimizations

The previous section described the algorithm in its most basic form, by making the assumption that a set of terrain triangles is available, which must be covered with vegetation. As we always cover all triangles with grass, this essentially means that we can only hope to be able to handle very small terrains at interactive frame rates. In this section we propose modifications to the basic algorithm and provide implementation details that allow the practical application of the basic technique to large terrains.

3.1. Determining Where to Seed

We can shrink the effective size of the terrain to a fraction of the total size with little loss in image quality, by culling terrain triangles that lie outside the view frustum or are too far away for individually rendered plants to be discernible.

One approach involves a hierarchical segmentation of the terrain into nested rectangular tiles using a quad-tree [Boulanger et al. 2009], followed by testing of each node against the view frustum. The process is terminated once a predetermined level of the quad-tree has been reached. This incurs a space and time overhead for the creation and traversal of the quad-tree during each frame, which grows with the size of the terrain. One can try to remedy this by limiting the depth of the search, but this leads to larger nodes, coarser culling and more plants being rendered outside the view frustum.

The approach described here takes advantage of the LOD algorithm used to render the terrain itself. To this end we use the ROAM algorithm [Duchaineau et al. 1997], which approaches the problem of rendering terrain from an optimization perspective: The terrain is modeled using a pair of binary trees with a right isosceles triangle at each node (see figure 4). The two triangles at the root span the whole heightfield and each further level of the tree can be created by splitting the parent triangle along its base thus forming another pair of right isosceles triangles. By sampling the heightmap at the new point introduced by the splits, an arbitrarily fine representation of the terrain can be generated. The target of the optimization algorithm, is to determine an optimal triangulation for each frame, that minimizes on-screen error given a fixed budget of triangles.

The structure of the binary triangle tree implies a hierarchical parent-child relationship, as each node spans the area of its descendants and hence contains the whole subtree rooted at it. The ROAM algorithm exploits this property to cull triangles in a hierarchical manner, by rejecting entire subtrees if it can be determined that the triangle at their root is not visible. This process efficiently calculates a set of triangles which are potentially visible and from which we can derive the input to the seeding stage of our algorithm.

We therefore begin by traversing the triangle bintree created during terrain tessellation by the ROAM algorithm. We skip the nodes that have been determined to be outside the view frustum and furthermore perform an additional test to discard triangles for which the closest vertex to the viewer lies further away than some user-adjustable threshold. This should be chosen based on the typical height of the vegetation as the distance at which separate plants become too small to be distinguishable from the terrain surface. Finally we also discard any terrain triangles that face away from the viewer by performing simple back-face culling. If a triangle is discarded during any of these tests, we terminate our recursion, clipping the entire subtree rooted at the discarded triangle.

The only modification we need to make to the ROAM algorithm itself, is to adjust the error metric (the error bound used as the the thickness of the "pie-wedge" bound shapes called "wedgies" in ROAM parlance) to account for the additional height of the canopy during view frustum culling. This is as simple as adding the maximum

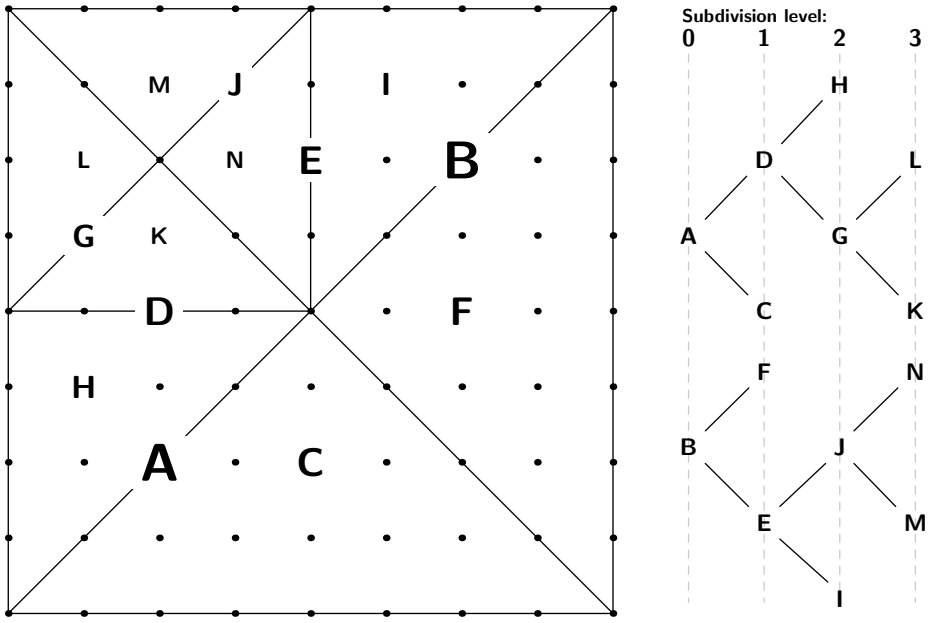


Figure 4. A triangulation of a 9×9 heightmap as carried out by the ROAM tessellation algorithm. The triangulation begins with two triangles, marked A and B above, which span the entire heightmap and proceeds with splitting operations, which yield new pairs of triangles. The process therefore creates a binary tree of triangles, with the important property that all triangles within a subtree are nested within the triangle at which the subtree is rooted.

possible plant height, which is a design parameter, to the calculated error bound.

One problem presented by the ROAM algorithm is that it dynamically changes the triangulation as the viewport moves. Since terrain triangles are sampled randomly to determine plant locations, based on seeds calculated by the triangle vertices (see section 2.2 above), this implies that the plant distribution will also change. We overcome this problem by always picking triangles to seed from the same level in the binary tree. If a leaf triangle in the binary triangle tree is at a lower level, we further refine it on-the-fly by recursively splitting along the midpoint of the base edge, without taking the heightmap into account. The only computational overhead with respect to ROAM bintree traversal therefore consists in calculating the midpoint.

The choice of the target level determines the culling granularity and constitutes a trade-off between CPU and GPU load. It can be chosen either empirically or based on the world-space area of the terrain triangles, which can be trivially calculated as a function of recursion level and terrain resolution and size.

We chose to perform the culling operations in eye-space because it allows the efficient calculation of the screen-space triangle area, and also because culling operations simplify to simple scalar comparisons in this space. We can transform the vertices of each triangle efficiently during tree traversal or additional refinement by noting the

following:

A. When traversing or refining a triangle we only need to transform a single vertex, the one generated by the split. The rest have already been transformed during our recursion and can be passed as parameters on the stack.

B. We can avoid any transformation calculations whatsoever during the refinement stage, which can make up the dominant part of culling work, by splitting the world to eye space transformation into a rotational and a translational part. We then only need to apply the rotational part of the transform to the initial ROAM triangles as we can calculate the rotated coordinates of refined triangles by simple interpolation due to the linearity of rotations. These intermediate "rotated coordinates" can then be converted to eye coordinates by adding the translational part of the transform, that is, the world-space location of the viewer.

3.2. Submitting Geometry to the GPU

The basic algorithm submits three vertices for each plant. We can avoid this by using geometry instancing, which allows us to submit each terrain triangle only once, with an instance count equal to the number of plants to be grown on it. Furthermore to avoid having to submit each triangle separately to the GPU we can bin terrain triangles together into a small set of bins based on the plant count and submit these to the GPU in batches. The bin count is not particularly crucial, as long as it is not overly large or too small to capture the variation of plant count in a typical terrain scene. Our implementation uses a set of 32 bins and experimentation has shown that performance is not affected significantly by tuning this parameter.

The choice of bin center is more critical, but the obvious approach of using some form of clustering technique, such as K-means clustering, to determine the optimal center is not practical. While it can be implemented efficiently in an incremental fashion, by performing one iteration of the K-means algorithm per frame, the approach suffers from slow convergence and spurious results due to convergence to local maxima. Furthermore the process of convergence is animated on-screen, with plants rearranging themselves en masse as the algorithm converges.

By studying the results of the K-means algorithm however, we observe that the resulting cluster centers typically follow the shape of a geometric progression. We therefore simply choose the bin centers according to

$$c_i = n_{\min} \left(\frac{n_{\max}}{n_{\min}} \right)^{\frac{i}{n_b - 1}}, \tag{15}$$

where n_b is the number of bins, $n_{\max}$, $n_{\min}$ are the maximum and minimum plant count per triangle over the whole terrain, and c_i is the center of the ith bin. The

minimum and maximum plant count can be chosen empirically or we can keep track of it for each frame and use it to recenter the bins during the next.

3.3. Controlling Plant Distribution

Vegetation in nature does not always uniformly cover the ground. The density usually varies due to variations in the underlying soil and parts of the terrain may be altogether bare. It is therefore desirable to be able to control plant density.

This has traditionally been achieved by defining a map over the terrain, mapping each point to an average plant density. The space requirements for this approach rise quadratically with respect to terrain size and linearly with the number of separate plant species, making it impractical for large terrains unless a very low resolution map is used. We use a different approach that is based on the method described by Papavasiliou [2011] in the context of detail map application.

For each terrain type we choose a reference color and at each point of the terrain we define a score for that type, based on the inverse distance of the terrain type's reference color to the terrain's base map color at that point. Conceptually, our aim is to be able to apply a grass texture map and also draw blades of grass wherever the terrain's base map appears green. In this context the HSV color space is more suitable, both for the definition of the reference colors and for the calculation of color distances. Scores for each terrain type are therefore calculated using the equation

$$s_i = \left\| (\vec{r_i} - \vec{b}) \odot \vec{w} \right\|^{-p}, \tag{16}$$

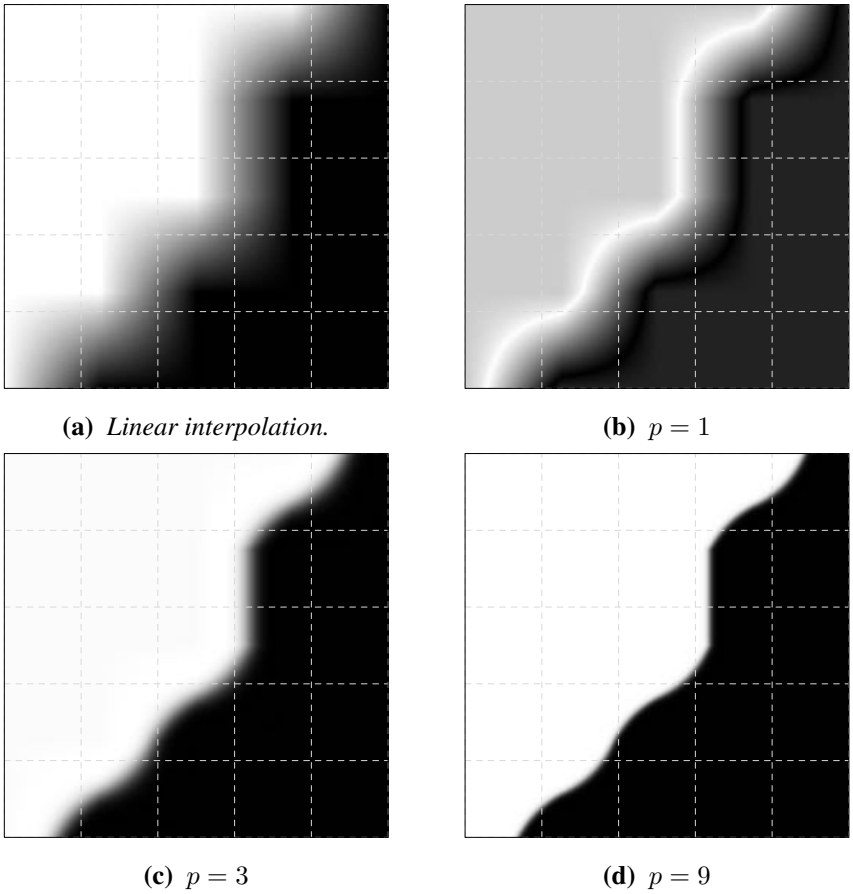
where s_i is the score for the ith terrain type, $\vec{r_i}$ is the ith type's reference color, $\vec{b}$ is the terrain's base map color at the currently sampled point, $\vec{w}$ is a weight vector, and p is a user-specified exponent. The operator $\odot$ denotes element-wise vector multiplication.

Using the calculated scores to blend detail textures for each terrain type can be done by simple interpolation, using the scores as weights. The final pixel color is computed from a palette of N detail textures as

$$C = \frac{\sum_{i=1}^{N} s_i T_i}{\sum_{i=1}^{N} s_i}, \tag{17}$$

where s_i and T_i are the score and detail texture color for the ith terrain type respectively and C is the blended output color.

The weights vector allows us to ignore certain color components during the calculation of the distance. This can be useful when assigning a color to rocky terrain for example, where a reasonable choice is gray for which hue is meaningless. Selecting a hue at random would introduce errors into the score calculations, which can be avoided by setting the weight for hue to zero. It is also useful in many cases to discard the lightness component, as its variation over the terrain is often the result of shadowing effects, which are unrelated to terrain type.



(a) *Linear interpolation.* **(b)** $p = 1$

(c) $p = 3$ **(d)** $p = 9$

Figure 5. A 5×5 monochrome map resampled to a size of 500×500. The upper-left plot shows the result of simple linear interpolation. The other plots were produced with a simplified grayscale version of the algorithm, by mixing two constant colors, black and white, instead of detail maps with various values for the exponent p. The reference gray levels assigned to black and white were 0.12 and 0.78 respectively.

The exponent p is useful in controlling the sharpness of the transition between different terrain types. By using lower values for p, broader transitional regions between terrain types can be achieved while larger values create sharper, more defined transitions. This is demonstrated in figure 5. It is worth noting that we avoid the blocky artifacts that would result from linear interpolation if we were to sample a map directly, without computing distances.

Other benefits of this technique include the reuse of the terrain's texture map instead of defining additional terrain type masks and the fact that a single map is necessary to control an arbitrary number of different terrain types. It would also be possible to change or even animate the change of reference colors in real time, thus shifting types of terrain on-screen, but we haven't investigated the usefulness of this

technique.

On the other hand calculating the scores requires some additional computation with respect to simple mask textures and some control over the distribution of terrain types is lost, as we're now only able to determine it indirectly, by changing the reference colours. This control can be regained by using a separate distribution map instead of the terrain's texture map, but this is less efficient in terms of space, than using simple terrain type masks, unless more than three terrain types are needed.

To vary the local vegetation based on terrain type, we sample the terrain's texture map at the root position of each plant and calculate the score for each terrain type. We select the two largest resulting values s_0 and s_1, calculate a random number z and compare it to $s_1/s_0 + s_1$. If z is smaller than this quantity, we select the reference color that corresponds to the second largest score and vice versa. We then proceed to grow a plant of the appropriate species, unless the selected reference color corresponds to bare soil, in which case the seed is rejected. Selecting the plant species using a random number ensures random mixing of plants in areas that lie on the borders between different terrain types. If one of the bordering areas involved is barren the result is a natural-looking falloff in plant density.

In certain cases, it is also useful to define a threshold on the score of a certain species and discard it, effectively setting its score to zero, if it is below that threshold. This allows us to fine-tune the extents of the regions covered by this kind of vegetation so that it doesn't grow on areas of very low score, only because all other defined terrain types score even lower.

If the selected reference color does not correspond to bare soil we can also calculate a modified version of the score as

$$s' = \begin{cases} 1 - \dfrac{s_1}{s_0 + s_1} & \text{if } z \geq \dfrac{s_1}{s_0 + s_1} \\ \dfrac{s_1}{s_0 + s_1} & \text{otherwise.} \end{cases} \tag{18}$$

When s_1 is equal to s_0, the value of s' is equal to 0.5, which indicates that the two dominant terrain types are perfectly balanced. As s_1 tends towards zero, s' tends to either 1 or 0, depending on whether the largest or second-largest score has been selected. We can use this value to modulate certain plant parameters, such as plant size or stiffness, to give a more natural appearance to our plant distributions.

3.4. Clustered Seeding

Soil composition often exhibits a certain degree of locality, which leads to clustering in the distribution of plants. We can exploit this by performing the calculations needed to determine the plant species to grow, which include base map lookup, score calculation and sorting, once for each cluster of plant. This leads to a substantial speed-up with little loss of quality.

Instead of drawing a single plant for each patch instance submitted to the GPU, we tessellate and draw a cluster of plants of the same species, which is determined by sampling the terrain at the location of the cluster center. The number of separate plants in each cluster can be determined arbitrarily in advance, but the size and shape of the cluster is critical. It must preserve the uniform distribution of plants over the terrain and ensure that clusters are compact and fall within the seeded triangle.

An approach that satisfies these requirements is to use stratified sampling. Instead of mapping uniform random variables in the unit square $[0, 1) \times [0, 1)$ onto the triangle via the transform described in equation 3, we divide the unit square into n^2 sub-domains or strata $[i/n, i+1/n) \times [j/n, j+1/n)$, where $0 <= i, j < n$ and sample N/Cn^2 seed clusters from each separate stratum, where N is the total seed count for the triangle and C is the number of seeds per cluster. For each sampled cluster, we sample C seeds within the stratum in which the cluster was sampled, which are then tessellated into plants as before.

As shown in figure 6, strata form distinct trapezoidal regions which span the entire terrain triangle and hence seeds sampled within each stratum will lie relatively close together within the triangle. The trapezoidal shape is certainly not ideal, but the random sampling of seeds within each stratum tends to make its exact shape indistinguishable and hence irrelevant.

We would like the number of strata to be as large as possible, as this would result in smaller clusters, but this poses a problem for terrain triangles where the number of sampled clusters is not large enough to fill all the strata. In these cases, which can easily make up the majority of terrain triangles, clusters and hence plant seeds will not be sampled from the entire surface of the triangle, resulting in visible artifacts.

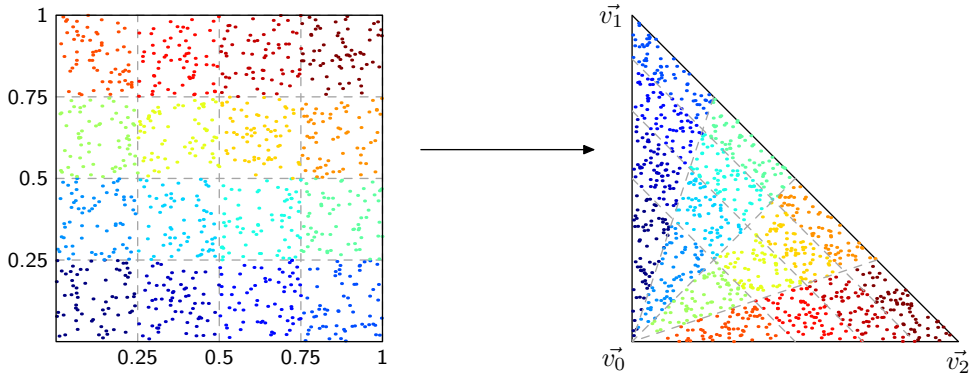


Figure 6. The left plot shows the domain $[0, 1) \times [0, 1)$ subdivided into 4×4 strata with 64 points sampled in each stratum. The transform described in equation 3 maps this onto the triangle with vertices $\vec{v_0}$, $\vec{v_1}$ and $\vec{v_2}$, where each stratum now has a trapezoidal shape except for the strata in the first column, which become triangular, as equation 3 maps the whole y axis to the first vertex $\vec{v_0}$.

Ideally we would like to choose the number of strata based on the number of seeds to be sampled on a terrain triangle, but the naive solution of simply dividing the unit square into $M \times M$ strata where $M = \left\lfloor \sqrt{N/C} \right\rfloor$, N and C being defined as before, isn't practical. As a triangle approaches the viewer, N rises, resulting in changes in strata configurations and hence abrupt redistribution of plants over the triangle.

To avoid this, we sample in a hierarchical fashion. The first cluster is sampled from the whole triangle, the next four clusters from a partitioning of the triangle into 2×2 strata and so on. To find out what partitioning to use when seeding the ith cluster, we make the following observation: The total number of sampled clusters for each successive repartitioning follows the sequence of square pyramidal numbers

$$P_n = \sum_{k=1}^{n} k^2 = \frac{2n^3 + 3n^2 + n}{6}. \tag{19}$$

Therefore, if we're currently seeding the ith cluster, we should partition our domain into $\lceil r \rceil \times \lceil r \rceil$ strata, where r is the only real root of the equation

$$P_n - i = 0. \tag{20}$$

The root, which can be calculated analytically, as $P_n - i$ is a cubic polynomial, is

$$r = \sqrt[3]{\frac{3}{2}i + \sqrt{\left(\frac{3}{2}i\right)^2 - \frac{1}{12^3}}} + \sqrt[3]{\frac{3}{2}i - \sqrt{\left(\frac{3}{2}i\right)^2 - \frac{1}{12^3}}} - \frac{1}{2}. \tag{21}$$

However, because in our case an exact solution is not crucial, we can use the approximate root

$$r = \sqrt[3]{3i} - \frac{1}{2}. \tag{22}$$

We thus implement clustered seeding efficiently with few changes. If a triangle is to be seeded with fewer plants than the chosen cluster size C, we proceed exactly as before without any clustering, otherwise we submit N/C patch instances to the GPU and make the following two modifications: We calculate the position of the cluster by partitioning the triangle into $\lceil r \rceil \times \lceil r \rceil$ strata, sampling the cluster center from within a randomly chosen stratum and, additionally, request a set of C line strips, instead of only one. We then create a separate plant out of each strip as before and choose its location randomly within the stratum of the cluster.

Stratified sampling, as we implemented it, can compromise the sampling uniformity to some degree. This seems to stem mostly from the fact that there aren't, in general, enough seeds to fully populate the last level of stratification, which can expose the underlying structure of trapezoidal strata. Although these sampling artifacts are virtually unnoticeable, as they are concealed by the random shape and size of the plants, the distribution of seeds can be improved to a large extent and at a very slight

cost in terms of execution speed by changing the way in which we sample points on the triangle. Instead of using equation 3 we can use the following equations:

$$r_1' = r_1\left(1 - \lfloor r_1 + r_2 \rfloor\right) + (1 - r_2)\lfloor r_1 + r_2 \rfloor, \quad (23)$$

$$r_2' = r_2\left(1 - \lfloor r_1 + r_2 \rfloor\right) + (1 - r_1)\lfloor r_1 + r_2 \rfloor, \quad (24)$$

$$p = \vec{v_0} + r_1'\left(\vec{v_1} - \vec{v_0}\right) + r_2'\left(\vec{v_2} - \vec{v_0}\right), \quad (25)$$

Equations 23 and 24 essentially reflect across the diagonal of the unit square all sampled points that lie above it. This turns the unit square into the triangle $r_1', r_2' \in [0,1) \times [0,1)\,,\ r_1' + r_2' < 1$, which is then mapped onto the terrain triangle by equation 25.

3.5. Multiple Kinds of Vegetation

Although we have limited this discussion to the case of grass, we can efficiently scatter various kinds of objects since all geometry generation is carried out by tessellation and geometry shaders inside the GPU. However, since all shaders have to be the same for every seed, this inevitably leads to multiple branches in each shader loaded for the tessellation, geometry and fragment stages which can slow down execution considerably. Another, perhaps more serious problem, is that we're limited to a single mode of tessellation, as it cannot be changed dynamically during shader execution.

We can overcome these limitations, at practically no cost in terms of execution speed, by breaking the shading of seeds into two stages. During the first, the vertex shader is executed, creating seed clusters, which are then sorted in the geometry shader according to vegetation kind and fed back into separate buffers, residing in GPU memory. For each buffer we then load a shading program, consisting of tessellation, geometry and fragment shaders for that particular vegetation species and resubmit it to the GPU.

This two-stage approach theoretically incurs the cost of an extra round-trip of the cluster-related data generated in the first stage to the GPU memory and back, but we haven't been able to measure any adverse impact on execution speed in our implementation. On the other hand we can now efficiently scatter objects of various shapes and sizes over the terrain. The only requirements are that they be small, as larger objects would force us to use a very large seeding horizon, and have a shape that can be generated with tessellation and geometry shaders or approximated with billboards. Examples of objects that fall into this category are rocks and gravel, fallen leaves, or woodland debris such as fragments of twigs.

3.6. Casting Shadows

Images without shadows tend to look flat and unrealistic, lacking cues which allow us to perceive the relative positioning of objects. Plants which don't cast shadows on the

underlying ground therefore appear to be floating vaguely over it, instead of growing out of it.

We can easily implement simple planar shadows cast from the blades onto the ground by using geometry shader instancing and requesting two invocations per geometry shader, which generates two instances of the plant geometry. The first instance is passed to the fragment shader as before and the second is projected onto the terrain triangle's plane along the light direction and shaded with a flat black color. To keep shadows from appearing unnaturally opaque, we can set a semi-transparent alpha value in conjunction with alpha to coverage, in order to get a screen-door transparency effect for the shadow polygons. No depth sorting is required using this approach.

4. Results

In order to measure the performance of the algorithm under various conditions, we set up a large terrain covered with grass and recorded various statistics during a thirty second animation. We used a machine equipped with an Intel Core i7-3770 CPU, 16GB of system RAM and an NVIDIA GeForce GTX650 Ti GPU with 1GB of video RAM. The operating system was a recently upgraded distribution of Linux, with the latest OpenGL drivers.

During the animation, the camera starts at a high elevation, where individual plants are mostly indistinguishable and therefore not rendered, approaches near-zero altitude after ten seconds and finally starts to ascend again after the twentieth second. The terrain covers an area of 20 square kilometers, sampled at a resolution of 5 meters and the measured implementation includes all optimizations described in section 3. Initial seeding was done in clusters of 8 seeds, which were then expanded and tessellated to strips with a maximum of 8 segments. Rasterization and shading was done at a resolution of 1920×1080 pixels and anti-aliasing was performed with 8 samples per fragment. A recorded video of the animation is available for download to aid in the interpretation of the measurements.

The load placed on the system depends greatly on the vantage point, mainly due to the level-of-detail nature of the algorithm, but also because of the uneven distribution of plants over the terrain. Therefore, instead of averaged values over the whole test run, the measurements are presented as a plot of averages over one-second intervals.

The average frame latency is plotted in figure 7. This frame time reflects, not only the vegetation rendering task, but also other tasks such as drawing the terrain and atmosphere and other miscellanea. For completeness, and in order to be able to evaluate the performance of the vegetation subsystem itself, we have also measured the frame latency during an identical test run where we've skipped all grass-related tasks. This is drawn in darker gray in figure 7.

Figure 8 shows the load placed on the GPU by measuring the number of clusters

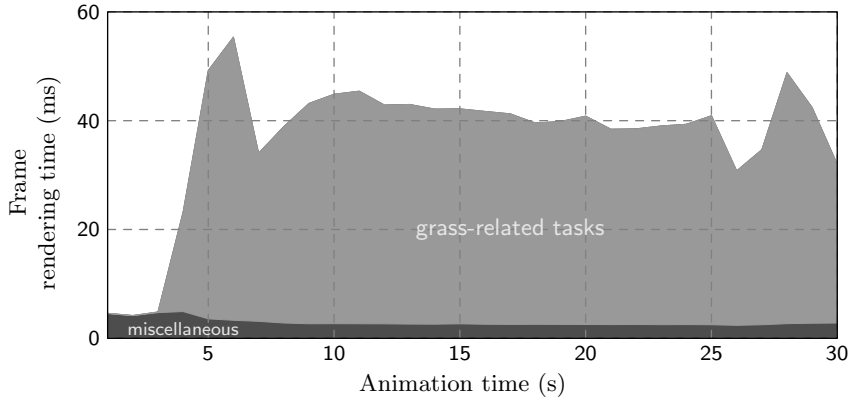


Figure 7. Frame rendering time averaged over one-second intervals. The darker area represents the time spent on tasks that are not vegetation-related, such as tessellating and rendering the terrain and sky and calculating the camera path.

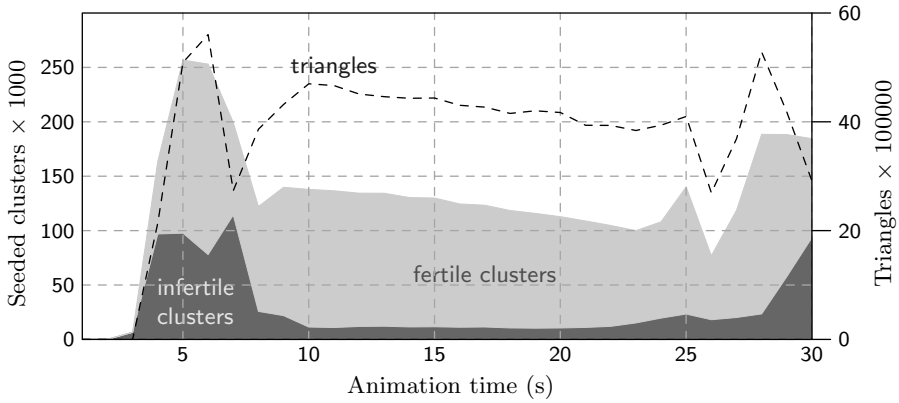


Figure 8. Plot of various statistics describing the geometrical load placed on the GPU. The filled plot represents seed clusters submitted to the first stage shader, with clusters picked over terrain that is designated as bare soil colored in darker gray. The clusters that make it to the second stage and are subsequently expanded into seeds and tessellated into individual plants are shown in light gray. The number of tessellated triangles, including those needed to render planar shadows, are plotted with a dashed line against the vertical scale on the right side of the graph.

submitted to the first stage vertex shader, the number of clusters fed back to the second stage shader for tessellation and rendering, as well as the resulting number of triangles used to render the individual plants.

Finally, to get an idea of the benefit of clustered seeding in terms of execution time, we turned off rasterization and measured the GPU time spent during the two phases of seeding, with various cluster sizes. The measured work consists of initial seeding and classification based on terrain base color and subsequent expansion of

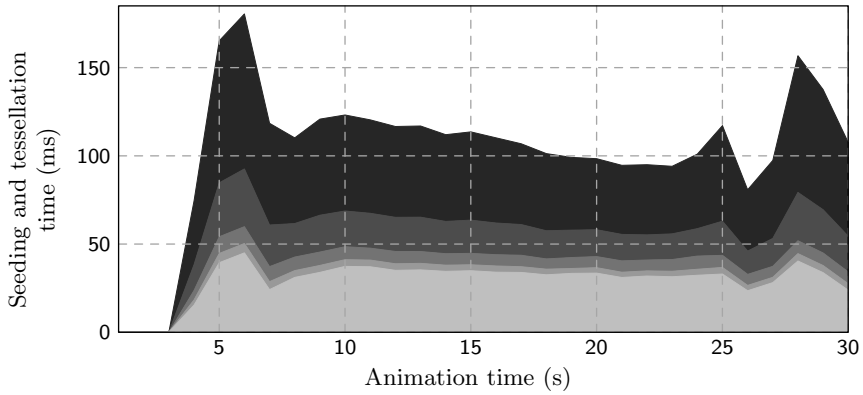


Figure 9. Plot of the execution times of the vertex-processing related tasks with various cluster sizes, as well as with no clustering at all. From dark to light gray we have cluster sizes of 1, 2, 4, 8 and 16 seeds per cluster.

first-stage seeds into clusters, followed by tessellation of individual plants and shadows. The results are presented in figure 9.

Unsurprisingly, the load due to vegetation rendering dominates all other tasks, even though these include ROAM tessellation and a relatively expensive multiresolution terrain splatting algorithm. On the other hand, the system achieves interactive frame rates on inexpensive consumer hardware and for scenes consisting of many hundreds of thousands of individually tessellated and drawn grass blades made up of millions of triangles. In a typical application such as a video game there would of course be many other tasks, demanding additional processing and memory bandwidth, so that the applicability of the algorithm to production systems depends heavily on the application at hand.

In any case the most attractive properties of the algorithm should not be sought in its performance characteristics but rather in its simplicity. At its core it simply picks out points on triangles and tessellates geometry on them, trying to maintain a target density regardless of the view point. This makes it applicable, at least in principle, not only to many different kinds of terrain systems but to any triangulated surface, such as, for instance, hedges or topiaries.

We used the ROAM algorithm in order to take advantage of its culling mechanism but most other terrain tessellators should be equally applicable. To illustrate the point, we now sketch an implementation using the popular geometry clipmaps algorithm.

1. Ensure that the highest resolution clipmap is large enough to fit the seeding horizon.

2. Submit the highest resolution geometry to the GPU after performing coarse culling on the CPU. Test each triangle against the view frustum and feed back visible triangles to a vertex buffer. Also calculate the projected area of the triangle and use it to choose the number of seeds to grow on the triangle. Feed back the number of seeds along with the vertex data.

3. Sort the triangles into a set of bins according to seed count and proceed as before.

Such an implementation would certainly result in much less complexity, and it could perhaps be optimized further by performing the sorting on the GPU using multiple output streams. It's more difficult to predict how it would fare in terms of performance though, as it involves the brute-force culling and sorting of a large number of triangles.

Few assumptions are made regarding the tessellated geometry as well, so that it's easy to grow objects entirely unrelated to grass, as can be seen in figure 1. Here seeds picked over bare soil are tessellated to simple diamond-shaped gravel particles. Furthermore, as each "plant" is tessellated and shaded individually, much flexibility is allowed in controlling its shape and appearance, based primarily on its location, as has been done in the implementation described here, but perhaps taking into consideration other parameters, such as time, as well.

A practical implementation, such as the one described in the previous sections is, of course, much more complicated than this basic form of the algorithm, but most of the flexibility afforded by its basic simplicity has nevertheless survived intact.

On the other hand our approach also suffers from most of the problems commonly plaguing level-of-detail algorithms. Although the level-of-detail technique helps in reducing the time and space needed to render a typical scene, the load, particularly on the graphics subsystem, is still considerable. Additionally popping also manifests itself in the form of grass blades suddenly appearing and vanishing on the ground. This is further aggravated with clustered seeding as it then happens on the level of clusters, but it is generally not too unpleasant, given sufficient plant density. Some sort of geomorphing or level-of-detail blending technique should be applicable to remedy the situation, but we have not made any investigations in this area.

References

BAKAY, B., AND HEIDRICH, W. 2002. Real-time animated grass. In *Proceedings of Eurographics (short paper)*. 27

BOULANGER, K., PATTANAIK, S. N., AND BOUATOUCH, K. 2009. Rendering grass in real time with dynamic lighting. *IEEE Computer Graphics and Applications 29*, 1, 32–41. 27, 35

DUCHAINEAU, M., WOLINSKY, M., SIGETI, D. E., MILLER, M. C., ALDRICH, C., AND MINEEV-WEINSTEIN, M. B. 1997. Roaming terrain: Real-time optimally adapting meshes. In *Proceedings of the 8th Conference on Visualization '97*, IEEE Computer Society Press, Los Alamitos, CA, USA, VIS '97, 81–88. 35

PAPAVASILIOU, D., 2011. Color-based per-pixel blending of detail textures. http://www.nongnu.org/techne/research/blending/, Nov. 38

PELZER, K. 2004. Rendering countless blades of waving grass. In *GPU Gems*, R. Fernando, Ed. Addison Wesley, Mar., 107–121. 27

PERBET, F., AND CANI, M.-P. 2001. Animating prairies in real-time. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, ACM, New York, NY, USA, I3D '01, 103–110. 27

QIU, H., CHEN, L., CHEN, J. X., AND LIU, Y. 2012. Dynamic simulation of grass field swaying in wind. *JSW 7*, 2, 431–439. 27

REEVES, W. T., AND BLAU, R. 1985. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *SIGGRAPH Comput. Graph. 19*, 3 (July), 313–322. 33, 34

SALMON, J. K., MORAES, M. A., DROR, R. O., AND SHAW, D. E. 2011. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, New York, NY, USA, SC '11, 16:1–16:12. 30

Index of Supplemental Materials

Apart from the video of the benchmarking run, we also provide a GNU Octave script which solves the system of equations 12, a C header file with the output of this script as a C array ready for use as texel data for a texture, as well as GLSL code implementing both stages of the algorithm. The GLSL code has been adapted, without loss in functionality, from our implementation and includes all optimizations mentioned in the text. It is provided for reference and to clarify any points which may not have been sufficiently explained and no guarantees are made that it will function as-is.

The material is organized as follows:

`bladeshape/` contains the script used to calculate the blade shape texture, as well as the provided C header file with texel data.

`glsl/` contains the reference GLSL code laid out as follows

- **`classification_*.glsl.c`** contains the source code for the first stage of the algorithm, which picks clusters, classifies them according to vegetation species and sorts them into separate output buffers to be fed back into the second stage.
- **`grass_*.glsl.c`** contains the source code implementing the second stage of the algorithm for grass. This includes cluster expansion and tessellation of individual grass blades as well as shading.
- **`common.glsl.c`** contains auxiliary routines which are needed by both stages, being mainly related to generating random cluster and seed locations.
- **`rand.glsl.c`** contains the random number generators used in the algorithm, including the cryptographic hash function, as well as a few LCG generators.
- **`color.glsl.c`** contains auxiliary functions related to color space manipulation.

Author Contact Information

Dimitris Papavasiliou
51 Hpeirou
Athens, 15231
dpapavas@gmail.com

Papavasiliou Dimitris, Real-Time Grass (and Other Procedural Objects) on Terrain, *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, no. 1, 26–49, 2015
http://jcgt.org/published/0004/01/02/

Received:	2014-06-16		
Recommended:	2014-07-10	Corresponding Editor:	Morgan McGuire
Published:	2015-02-05	Editor-in-Chief:	Morgan McGuire