4

Vegetation Management in Leadwerks Game Engine 4

Josh Klint
Leadwerks Software

4.1 Introduction

Although rendering capabilities of graphics hardware have increased substantially over the last generation, a specialized system for managing vegetation rendering and physics still performs an order of magnitude faster than a general purpose object management system and can use a small fraction of the memory that would otherwise be required. These optimizations are possible due to the unique characteristics of vegetation.

First, trees, bushes, and plants tend to be highly repetitive in nature, at least to the casual observer. One pine tree looks pretty much like any other. Variations in rotation, scale, and color are enough to convince the viewer that an entire forest made up of only a few unique tree models contains infinite variety.

Second, plants tend to be distributed in a roughly uniform manner. Areas with optimal conditions for growth tend to be filled with plant life, while inhospitable conditions (e.g., a steep cliff side) tend to be barren. Individual instances of plants repel one another as they compete for resources, resulting in a roughly uniform distribution. An extreme manmade example of this is an orchard where trees are planted in rows of equal spacing to make maximum use of the earth.

Finally, plants grow along the surface of the earth, which for practical purposes can be considered a 2D plane with a height offset. This means that their position in 3D space can be predicted and described in a compact manner.

These characteristics provide the opportunity for unique optimizations that can manage a larger volume of plant instances than a general-purpose object management system is capable of. This chapter describes our implementation of

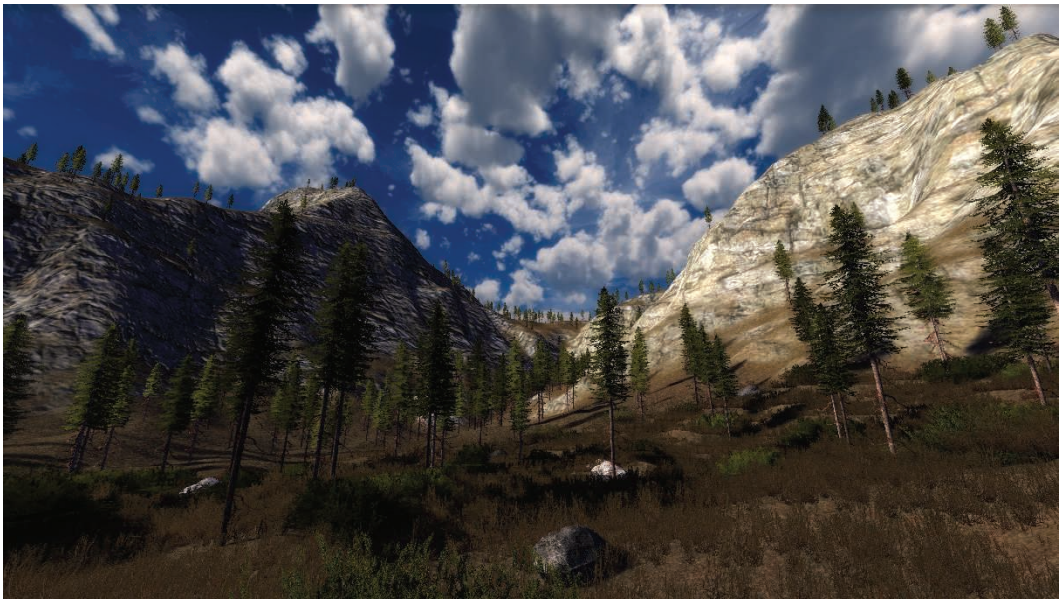

Figure 4.1. A sample scene demonstrating several vegetation layers for different types of objects. This scene is included with the demo on the book's website.

such a system to handle both rendering and physics of large volumes of vegetation. The visual results of our implementation are shown in Figure 4.1, rendered in real time.

4.2 The Problem

In Leadwerks Game Engine 2, vegetation instances were stored in the scene file as a sequence of 4×4 matrices. Instances were read into memory and placed into a quadtree data structure. Chunks of instances were merged into single surfaces made of quads. A series of 2D images of the vegetation object was rendered at run time and displayed on the merged surfaces when the quadtree node reached a certain distance from the camera. Quadtree nodes closer than the billboard distance were iterated through, and all instances were individually culled and displayed as full-resolution 3D models. For physics, a specialized collision type was employed that accepted an array of 4×4 matrices and a scale factor for each axis.

This system allowed large, expansive landscapes to be rendered in real time with fast performance, but it also presented two problems. First, the rendering

system involved significant CPU overhead. This was particularly taxing when performing frustum culling on quadtree nodes that were closer than the billboard LOD distance because each instance had to be tested individually. Second, each vegetation instance required the storage of a 4×4 matrix in two places (for physics and in the quadtree structure) plus a scaling factor for physics. Although 140 bytes per instance may seem minimal, the memory requirements became quite large in densely packed scenes and could grow to several hundred megabytes of data. This requirement seemed unnecessary, and it limited the ability of our vegetation system to scale.

Since our future plans involve expanded game worlds beyond the bounds of 32-bit floating point precision, it makes sense to build a more scalable vegetation management system now. At the beginning of our implementation of a new vegetation management system for Leadwerks Game Engine 4, we set out with the following design parameters:

- Fast rendering and physics performance.
- Low-or-no marginal memory consumption per instance.
- Ability to dynamically add and remove vegetation instances instantly (to provide better support for farm simulators and other games that alter the landscape).

We targeted hardware that supports OpenGL 4.0, which is equivalent to DirectX 11 and includes both discrete and integrated GPUs from Nvidia, AMD, and Intel.

4.3 The Idea

The Sierra Nevada mountain range in the western United States provides ample opportunities for hiking, mountain biking, and camping, with nine national forests and three national parks to visit. If you've ever hiked through the Sierras, you know that the view is pretty much "trees, everywhere". Trees do not grow directly on top of each other, and no good spot goes bare for long before something grows there.

The simplest approximation of this experience we could design would be a grid of tree instances that surround the camera, stretching out to the maximum view distance. This arrangement of trees lacks the apparent randomness found in nature, but if we use the tree's (x, z) position as a seed value, it should be possible to procedurally generate a rotation and scale for a more natural appearance. An additional offset can be generated to break up the appearance of the grid dis-

tribution and make it appear more random while retaining the roughly constant density of natural vegetation. A texture lookup on the terrain height map can then be used to retrieve the y (vertical) position of the tree on the landscape.

An initial test application was easily implemented and indicated that this gave a satisfactory appearance, but a full vegetation system with culling, LOD, and physics based on this idea was quite another thing to implement. The first challenge was to devise a system that would produce identical procedural noise on both the GPU and CPU.

The first attempt used a pseudorandom noise function in GLSL to generate unique results from each instance's (x, z) position. There were two problems with this idea. First, the performance penalty for using this to generate rotated 4×4 matrices for each vertex would be significant. Second, it was feared that small discrepancies in floating-point computations between different graphics hardware and CPU architectures could result in a misalignment of the procedural data calculated for graphics and physics.

To solve these issues, a small grid of instance data was generated. Each grid space stores a 4×4 matrix with the color contained in the right-hand column as follows.

$$\begin{bmatrix} m_{00} & m_{01} & m_{02} & brightness \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & scale \end{bmatrix}$$

This data is enough to provide a rotation, scale, color, and a small offset value to add to the position. The data is sent to the culling and rendering shaders as a texture. The data was tiled repeatedly across the vegetation field, in the same manner as a tiling texture applied to a surface. It was found that a 16×16 grid was sufficient to eliminate any tiling appearance, especially when variations in the terrain height were present. With this completed, the second requirement for our design was met because the memory usage of additional instances was zero.

Originally, we planned to combine this procedural distribution with the quadtree design of the Leadwerks 2 vegetation system. However, another technique came to our attention that promised to entirely eliminate the CPU overhead of the vegetation rendering system.

4.4 Culling

Instance cloud reduction is an established technique for modern hardware that efficiently culls large numbers of objects entirely on the GPU [Rákos 2010,

Shopf et al. 2008]. The technique works by passing all possible instances to the GPU and discarding instances that are culled by the camera frustum in a geometry shader (see Figure 4.2). Unlike a traditional shader, no fragment output is generated, and the results of the geometry shader are output to a texture buffer. A GPU query is used to retrieve the number of rendered primitives, which corresponds to the number of rendered instances. Visible geometry is rendered in a second pass by using the results of the query as the number of instances drawn and by reading the texture buffer that was populated in the culling pass. This technique seemed to offer the greatest promise for eliminating the overhead of our vegetation rendering system.

Rather than passing transformation data to the culling shader, we wanted to generate the transformation data entirely on the GPU. Our repeating grid technique detailed in the previous section was easily integrated into the culling

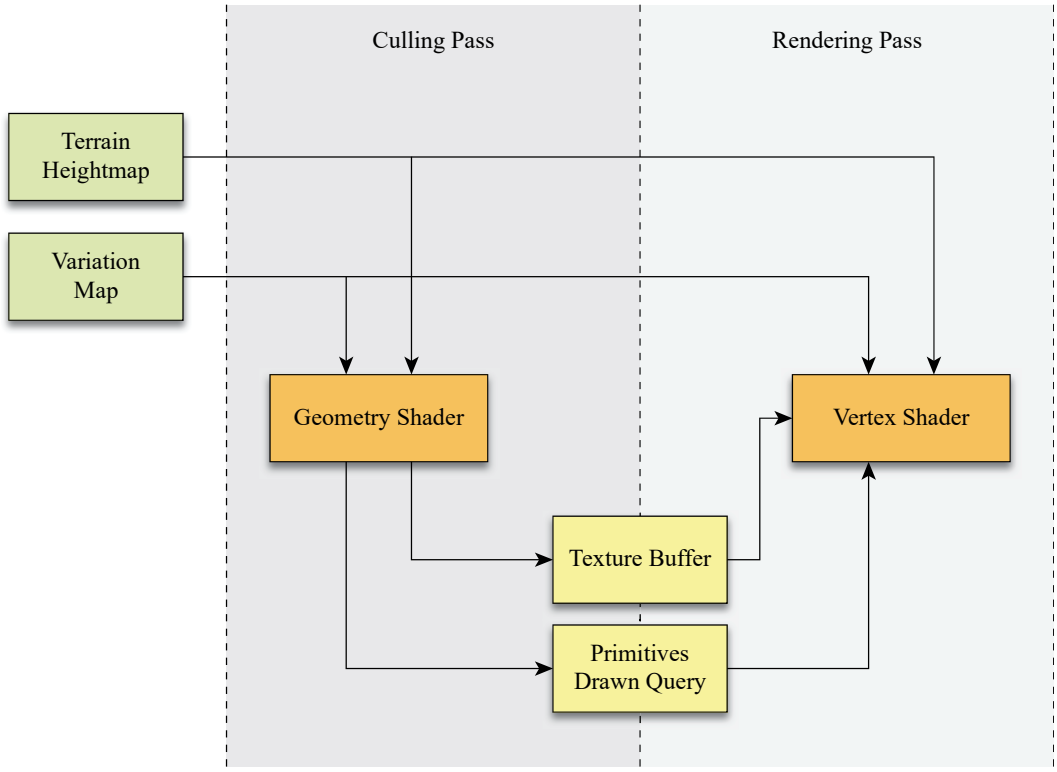

Figure 4.2. Instance cloud reduction is a GPU culling technique that works by writing visible instance IDs into a texture buffer, and then rendering visible objects in an additional pass.

shader. Unlike previous implementations that outputted a 4×4 matrix into the texture buffer, we instead wrote out a 32-bit unsigned integer specifying the instance ID.

Uniform values are sent to the culling shader to specify the grid position, size, and distribution density. The instance ID is used to calculate the *x* and *z* coordinates of each instance according to the following formula.

```
float x = floor(gl_InstandID / gridsize.x);
float z = gl_InstandID - x * gridsize.y;
x += gridoffset.x;
z += gridoffset.y;
```

The instance 4×4 matrix is then calculated according to the following formula, where `texture5` is the 16×16 pixel variation map that we generated.

```
mat4   mat;

float sy = 1.0 / variationmapresolution;
float sx = sy * 0.0625;

mat[0][0] = texture(texture5, vec2((x * 16.0 + texcoord.x + 0.0) * sx,
       texcoord.y + z * sy)).r;
mat[0][1] = texture(texture5, vec2((x * 16.0 + texcoord.x + 1.0) * sx,
       texcoord.y + z * sy)).r;
mat[0][2] = texture(texture5, vec2((x * 16.0 + texcoord.x + 2.0) * sx,
       texcoord.y + z * sy)).r;
mat[0][3] = texture(texture5, vec2((x * 16.0 + texcoord.x + 3.0) * sx,
       texcoord.y + z * sy)).r;

mat[1][0] = texture(texture5, vec2((x * 16.0 + texcoord.x + 4.0) * sx,
       texcoord.y + z * sy)).r;
mat[1][1] = texture(texture5, vec2((x * 16.0 + texcoord.x + 5.0) * sx,
       texcoord.y + z * sy)).r;
mat[1][2] = texture(texture5, vec2((x * 16.0 + texcoord.x + 6.0) * sx,
       texcoord.y + z * sy)).r;
mat[1][3] = texture(texture5, vec2((x * 16.0 + texcoord.x + 7.0) * sx,
       texcoord.y + z * sy)).r;

mat[2][0] = texture(texture5, vec2((x * 16.0 + texcoord.x + 8.0) * sx,
```

```
        texcoord.y + z * sy)).r;
mat[2][1] = texture(texture5, vec2((x * 16.0 + texcoord.x + 9.0) * sx,
        texcoord.y + z * sy)).r;
mat[2][2] = texture(texture5, vec2((x * 16.0 + texcoord.x + 10.0) * sx,
        texcoord.y + z * sy)).r;
mat[2][3] = texture(texture5, vec2((x * 16.0 + texcoord.x + 11.0) * sx,
        texcoord.y + z * sy)).r;

mat[3][0] = texture(texture5, vec2((x * 16.0 + texcoord.x + 12.0) * sx,
        texcoord.y + z * sy)).r;
mat[3][1] = texture(texture5, vec2((x * 16.0 + texcoord.x + 13.0) * sx,
        texcoord.y + z * sy)).r;
mat[3][2] = texture(texture5, vec2((x * 16.0 + texcoord.x + 14.0) * sx,
        texcoord.y + z * sy)).r;
mat[3][3] = texture(texture5, vec2((x * 16.0 + texcoord.x + 15.0) * sx,
        texcoord.y + z * sy)).r;

mat[3][0] += x * density;
mat[3][2] += z * density;

vec2 texcoords = vec2((mat[3][0] + TerrainSize * 0.5) / TerrainSize +
        1.0 / TerrainResolution * 0.5, (mat[3][2] + TerrainSize * 0.5) /
        TerrainSize + 1.0 / TerrainResolution * 0.5);

mat[3][1] = texture(texture6, texcoords).r * TerrainHeight;
```

This shader could be optimized further by using an RGBA floating-point texture that combines matrix rows into a single texture lookup.

The first three elements of the fourth matrix row holds the instance's position in global space. We first check to see if the instance position is beyond the bounds of the terrain as follows.

```
if (mat[3][0] < -TerrainSize * 0.5) return;
if (mat[3][0] > TerrainSize * 0.5) return;
if (mat[3][2] < -TerrainSize * 0.5) return;
if (mat[3][2] > TerrainSize * 0.5) return;
```

We then calculate the distance to the camera and discard instances that are too far away from the camera to be seen:

```
if (dist >= viewrange.y) return;
```

We added slope and height constraints. This can be used to ensure that trees only appear above sea level, or to prevent plants from growing on steep cliff faces.

```
if (slope < sloperange.x) return;
if (slope > sloperange.y) return;
if (mat[3][1] < heightrange.x) return;
if (mat[3][1] > heightrange.y) return;
```

Finally, we calculate the actual center of the object's axis-aligned bounding box (AABB) and radius as follows.

```
vec3 scale = vec3(length(mat[0].xyz), length(mat[1].xyz),
       length(mat[2].xyz));
scale = scalerange.x + (scale - 1.0) * (scalerange.y - scalerange.x);

vec3 aabbcenter = aabbmin + (aabbmax - aabbmin) * 0.5;
size = aabbmax - aabbmin;
float radius = length(size * scale);

vec3 center = mat[3].xyz;
center.x += aabboffset.x * scale.x;
center.y += aabboffset.y * scale.y;
center.z += aabboffset.z * scale.z;
```

Then, we perform an intersection test between the camera frustum and a sphere that completely encloses the instance. The `PADDING` macro can be set to 1.0 to shift the sides of the frustum volume inward and visually confirm that the algorithm is working correctly. The frustum plane uniforms are the six planes that define the camera frustum volume. If a point lies outside of any of these planes, then by definition it does not intersect the camera frustum volume and is not visible.

```
#define PADDING 0.0

if (PlaneDistanceToPoint(frustumplane0, center) > radius - PADDING) return;
```

```
if (PlaneDistanceToPoint(frustumplane1, center) > radius - PADDING) return;
if (PlaneDistanceToPoint(frustumplane2, center) > radius - PADDING) return;
if (PlaneDistanceToPoint(frustumplane3, center) > radius - PADDING) return;
if (PlaneDistanceToPoint(frustumplane4, center) > radius - PADDING) return;
if (PlaneDistanceToPoint(frustumplane5, center) > radius - PADDING) return;
```

Finally, once all our tests have passed, we write the instance ID into the texture buffer and emit a vertex:

```
transformfeedback0 = ex_instanceID[0];
EmitVertex();
```

The outputted instance IDs are read by the vertex shader during the next step.

4.5 Rendering

Visible rendering of the instances takes place in a second pass. The results of the primitive count query are retrieved, and this tells us the number of instances to render. The texture buffers written in the culling pass are used to retrieve the ID of each rendered instance.

By default, our system uses two levels of detail, one that draws the full polygonal models (the *near* instances) and another that displays billboard representations of the models (the *far* instances). It quickly became apparent to us that two culling passes are necessary, one to collect the IDs of all near instances, and another for all far instances. The results are written into two separate texture buffers. A separate query object is used during each pass to retrieve the number of rendered primitives.

The necessity to synchronize with the GPU is a potential problem with this design. To reduce this problem, we perform vegetation culling at the beginning of the camera render function, and we perform the visible render pass toward the end, leaving as much activity between the two steps as possible. We found it was possible to reverse this order in the shadow rendering pass, so that rendering results were always one frame behind. This could also be done in the visible rendering pass, but quick movements of the camera resulted in very noticeable glitches, so this was reverted.

The demo accompanying this chapter on the book's website shows our system using three vegetation layers for trees, bushes, and grass. Source code for the vegetation class is included. Performance on discrete Nvidia and AMD GPUs

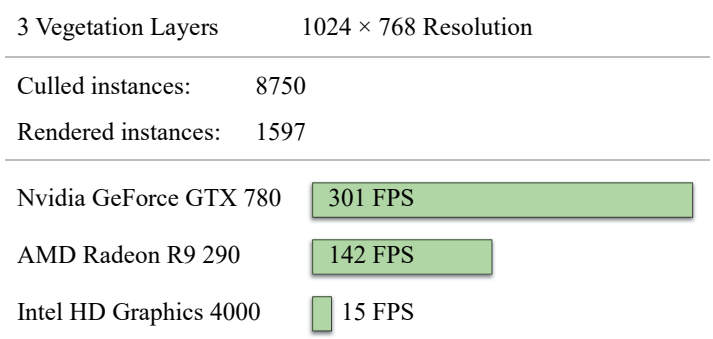

Figure 4.3. Performance is shown for the example scene running on Nvidia, AMD, and Intel graphics hardware.

was surprisingly fast with heavy loads in spite of the synchronization issue (Figure 4.3). Intel hardware performed at a comparatively slower speed, but this is to be expected for an integrated graphics chip. The use of atomic counters (available in GLSL 4.3) promises to provide a mechanism to avoid CPU/GPU synchronization, but this was not implemented in our first pass since our minimum system requirements are OpenGL 4.0 hardware.

4.6 Level of Detail

An efficient culling system capable of managing hundreds of thousands of vegetation instances requires an equally efficient detail reduction system, since so many objects can be drawn at once, and the vast majority of them are far away from the viewer. In Leadwerks 4, we opted to use a simple billboard system to draw vegetation instances that were positioned an adjustable distance from the camera. Instances closer than the billboard distance are rendered as full polygonal meshes with no intermediate LOD stages. This simplifies the engine workflow and limits the number of required culling passes to two, or to a single pass for layers that do not use billboarding (e.g., small plants and rocks).

We started by performing a series of 16 orthographic renders of the vegetation model. The diffuse, normal, and emission channels of the camera buffer provided a convenient mechanism to store the billboard data. Render results were copied into a single texture as shown in Figure 4.4. To provide some depth to the billboards when viewed from above, we performed an additional top-down render of the object and added a second upwards-facing quad to each billboard.

Figure 4.4. A series of 16 orthographic views are rendered to the billboard texture. The closest two views are interpolated between based on camera position.

This remains stationary and provides a better appearance than camera-facing billboards.

The Leadwerks 2 vegetation system suffered from severe "popping" when models transitioned to billboard and when billboard views were switched based on the camera viewing angle. Creating a smooth transition between detail levels and billboard faces was a priority.

In the following billboard vertex shader, the relative angle of the camera position to the billboard's local space is calculated with the code below. The varying output `stage` represents the index of the billboard image to display, and the `blend` value is used to interpolate between the current and next stage.

```
float a = (atan(hdiff.y, hdiff.x) + pi) / (2.0 * pi) - 0.25;
a -= GetYaw(id);
a = mod(a, 1.0);
float stage = floor(a * billboardviews);
blend = a * billboardviews - stage;
```

Texture coordinates for the two nearest stages are calculated and outputted in varying `vec2` values as follows.

```
ex_texcoords0 = vertex_texcoords0;
ex_texcoords0.x /= billboardviews;
ex_texcoords0.x += stage / billboardviews;

stage = ceil(a * billboardviews);
if (stage >= billboardviews) stage = 0.0;
ex_texcoords1 = vertex_texcoords0;
ex_texcoords1.x /= billboardviews;
ex_texcoords1.x += stage / billboardviews;
```

In the following fragment shader, a dissolve effect is used to smoothly transition between billboard stages based on the `blend` varying value. This works by generating a pseudorandom floating-point value using the screen coordinate as a seed value, and comparing it to the `blend` value. The same technique is also used to smoothly fade billboards in and out with distance. When combined with multisampling, this produces results similar to alpha blending but retains accurate lighting information for the deferred renderer.

```
// Generate psuedorandom value
float f = rand(gl_FragCoord.xy / buffersize * 1.0 +
       gl_SampleID * 37.45128 + ex_normal.xy);

// Diffuse
vec4 outcolor = ex_color;
vec4 color0 = texture(texture0, ex_texcoords0);
vec4 color1 = texture(texture0, ex_texcoords1);

vec4 normalcolor;
vec4 normalcolor0 = texture(texture1, ex_texcoords0);
vec4 normalcolor1 = texture(texture1, ex_texcoords1);

vec4 emission;
vec4 emission0 = texture(texture2, ex_texcoords0);
vec4 emission1 = texture(texture2, ex_texcoords1);

// Dissolve blending
if (f > blend)
{
    outcolor = color0;
    normalcolor = normalcolor0;
    emission = emission0;
}
else
{
    outcolor = color1;
    normalcolor = normalcolor1;
    emission = emission1;
}
```

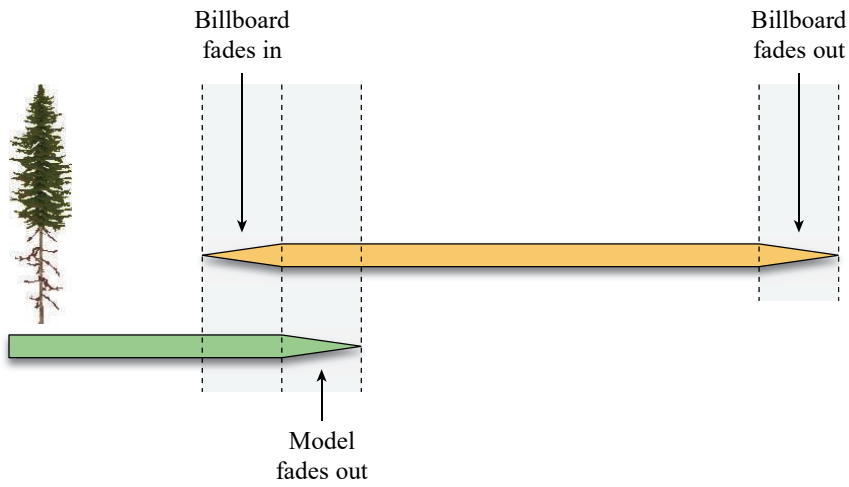

Figure 4.5. When an instance reaches a certain distance from the camera (positioned on the left), the billboard object begins to fade in using a dissolve effect. Once the billboard is completely visible, the model begins to fade out. This provides a smooth transition between LOD stages. The billboard also fades out as the maximum view range is reached.

As the distance from an instance reaches the billboard rendering distance, the billboard begins dissolving in. At the distance where the billboard is completely solid, the 3D model begins dissolving out. Finally, when the maximum view distance is reached, the billboard is dissolved out over a short distance. See Figure 4.5 for an illustration of these transitions.

The billboard orientation we used, combined with the dissolve effect for distance fading and billboard view blending, results in smooth transitions and accurate lighting that eliminate the "popping" artifacts observed in our previous vegetation system.

4.7 Physics

Leadwerks Game Engine uses the Newton Game Dynamics library to simulate physics. Newton was chosen due to its accuracy, stability, and ease of implementation. One of the useful features Newton provides is a mechanism for specifying a user-defined collision mesh. This works by invoking a user-defined callback function any time an object intersects a specified bounding box. The callback builds a mesh dynamically for all mesh faces within the volume that the colliding

object occupies. This can be used for efficient collisions with height map terrain or voxel geometry. In our case, the system allowed us to provide vegetation collision data to the physics simulator without having to actually store all the 4×4 matrices for each vegetation instance in memory.

A function to get all vegetation instances in an arbitrary AABB was implemented as follows.

```
int VegetationLayer::GetInstancesInAABB(const AABB& aabb,
       std::vector<Mat4>& instances, const float padding)
{
    int count = 0;
    iVec2 gridmin;
    iVec2 gridmax;
    float r = 0.0;
    Mat4 mat, identity;
    Vec3 scale;
    Vec3 center;
    AABB instanceaabb;

    if (model)
    {
        r = Vec2(model->recursiveaabb.size.x,
            model->recursiveaabb.size.z).Length() +
            Vec2(model->recursiveaabb.center.x,
            model->recursiveaabb.center.z).Length();
    }

    gridmin.x = floor((aabb.min.x - scalerange[1] * r) / density - 0.5);
    gridmin.y = floor((aabb.min.z - scalerange[1] * r) / density - 0.5);
    gridmax.x = ceil((aabb.max.x + scalerange[1] * r) / density + 0.5);
    gridmax.y = ceil((aabb.max.z + scalerange[1] * r) / density + 0.5);

    for (int x = gridmin.x; x <= gridmax.x; x++)
    {
        for (int y = gridmin.y; y <= gridmax.y; y++)
        {
            mat = GetInstanceMatrix(x, y);
            instanceaabb = Transform::AABB(model->recursiveaabb, mat,
                   identity, false);
```

```
            if (instanceaabb.IntersectsAABB(aabb, padding))
            {
                instanceaabb = Transform::AABB(model->recursiveaabb, mat,
                       identity, true);

                if (instanceaabb.IntersectsAABB(aabb, padding))
                {
                    count++;
                    instances.push_back(mat);
                }
            }
        }
    }

    return count;
}
```

The same function that retrieves an instance matrix in GLSL was implemented in C++ as follows.

```
Mat4 VegetationLayer::GetInstanceMatrix(const int x, const int z)
{
    Mat4   mat;

    int ix = Math::Mod(x, variationmapresolution);
    int iz = Math::Mod(z + variationmapresolution / 2,
           variationmapresolution);
    int offset = (iz * variationmapresolution + ix) * 16;

    if (variationmatrices.size() == 0)
    {
        BuildVariationMatrices();
    }

    mat[0][0] = variationmatrices[offset + 0];
    mat[0][1] = variationmatrices[offset + 1];
    mat[0][2] = variationmatrices[offset + 2];
    mat[0][3] = 0.0;
```

```
    mat[1][0] = variationmatrices[offset + 4];
    mat[1][1] = variationmatrices[offset + 5];
    mat[1][2] = variationmatrices[offset + 6];
    mat[1][3] = 0.0;

    mat[2][0] = variationmatrices[offset + 8];
    mat[2][1] = variationmatrices[offset + 9];
    mat[2][2] = variationmatrices[offset + 10];
    mat[2][3] = 0.0;

    mat[3][0] = variationmatrices[offset + 12] + x * density;
    mat[3][2] = variationmatrices[offset + 14] + z * density;
    mat[3][1] = terrain->GetElevation(mat[3][0], mat[3][2]);
    mat[3][3] = 1.0;

    float scale = scalerange.x + variationmatrices[offset + 15] *
           (scalerange.y - scalerange.x);
    mat[0] *= scale;
    mat[1] *= scale;
    mat[2] *= scale;
    return mat;
}
```

In the Newton collision callback function, all the intersecting instances are retrieved. Their vertex positions and normals are transformed to global space and added to arrays that are returned to the physics simulator. The `FastMath` class functions are optimized inline functions designed for demanding real-time operations like the following.

```
int count = layer->GetInstancesInAABB(aabb,
       layer->instances[threadNumber]);

// Merge all intersecting instances into vertex / indice arrays
for (int n = 0; n < count; n++)
{
    for (int p = 0; p < vert_count; p++)
    {
        offset = p * 3;
```

```
        pos.x = layer->vertexpositions[offset + 0];
        pos.y = layer->vertexpositions[offset + 1];
        pos.z = layer->vertexpositions[offset + 2];

        FastMath::Mat4MultiplyVec3(layer->instances[threadNumber][n],
            pos, result);

        offset = ((n * (vert_count + tris_count)) + p) * 3;
        layer->collisionsurfacevertices[threadNumber][offset + 0]
               = result.x;
        layer->collisionsurfacevertices[threadNumber][offset + 1]
               = result.y;
        layer->collisionsurfacevertices[threadNumber][offset + 2]
               = result.z;
    }

    for (int p = 0; p < tris_count; p++)
    {
        offset = p * 3;
        pos.x = layer->facenormals[offset + 0];
        pos.y = layer->facenormals[offset + 1];
        pos.z = layer->facenormals[offset + 2];

        FastMath::Mat3MultiplyVec3(Mat3(layer->instances[threadNumber][n]),
               pos, result);

        float m = 1.0F / result.Length();
        offset = ((n * (vert_count + tris_count)) + vert_count + p) * 3;
        layer->collisionsurfacevertices[threadNumber][offset + 0]
               = result.x * m;
        layer->collisionsurfacevertices[threadNumber][offset + 1]
               = result.y * m;
        layer->collisionsurfacevertices[threadNumber][offset + 2]
               = result.z * m;
    }

    // Add indices
    memcpy(&layer->collisionsurfaceindices[threadNumber]
        [n * layer->indices.size()], &layer->indices[0],
        layer->indices.size() * sizeof(int));
```

```
    // Offset indices
    index_offset = n * (collisionsurface->CountVertices() +
           collisionsurface->CountTriangles());

    for (int p = 0; p < tris_count; p++)
    {
        offset = n * layer->indices.size() + p * 9;

        layer->collisionsurfaceindices[threadNumber][offset + 0]
               += index_offset;
        layer->collisionsurfaceindices[threadNumber][offset + 1]
               += index_offset;
        layer->collisionsurfaceindices[threadNumber][offset + 2]
               += index_offset;
        layer->collisionsurfaceindices[threadNumber][offset + 4]
               += index_offset;
        layer->collisionsurfaceindices[threadNumber][offset + 5]
               += index_offset;
        layer->collisionsurfaceindices[threadNumber][offset + 6]
               += index_offset;
        layer->collisionsurfaceindices[threadNumber][offset + 7]
               += index_offset;
    }
}
```

Although iterating through and transforming vertices is not an optimal design, in practice it has proven to be fast enough for production use. In the future, a new collision type could be implemented in Newton that retrieves an array of 4×4 matrices of all intersecting instances.

With this step complete, we finally have a system with parity between the instance orientations generated on both the CPU and GPU without actually having to store the data for each individual instance. Although it is not present in this implementation, a filter map can easily be added to discard instances at specific grid positions and allow dynamic insertion and deletion of instances.

4.8 Future Development

Because Leadwerks Game Engine 4 is a commercial product already on the market, system requirements are locked on OpenGL 4.0 hardware. In the future, an

implementation using atomic counters could eliminate CPU/GPU synchronization when OpenGL 4.3-compliant drivers are detected.

Implementation of an "instance cloud" collision type in Newton Game Dynamics would provide better performance than dynamically constructing a mesh from transformed vertices.

Although this system does an excellent job of producing natural spacing among vegetation instances, different vegetation layers have no communication of the spaces they occupy. Multiple dense layers can easily appear on top of one another in unnatural arrangements, like a tree in the middle of a rock. As our worlds get bigger, the need for intelligent placement of instances will become more important.

Our culling algorithm only performs simple frustum culling on vegetation instances, which loses the advantages of the hierarchical occlusion system our standard object management system uses. In the future, nearby occluding volumes could be sent to the culling shader and used to discard occluded vegetation instances.

Our system is suited for geometry that is generally arranged on a 2D surface, but it could be adapted to 3D to handle things like a dense procedurally generated asteroid field.

References

[Rákos 2010] Daniel Rákos. "Instance Culling Using Geometry Shaders". RasterGrid Blog, 2010. Available at http://rastergrid.com/blog/2010/02/instance-culling-using-geometry-shaders/

[Shopf et al. 2008] Jemery Shopf, Joshua Barczak, Christopher Oat, and Natalya Tatarchuk. "March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatures on GPU". *ACM SIGGRAPH 2008: Proceedings of the conference course notes, Advances in Real-Time Rendering in 3D Graphics and Games*, pp. 52–101.

This page intentionally left blank