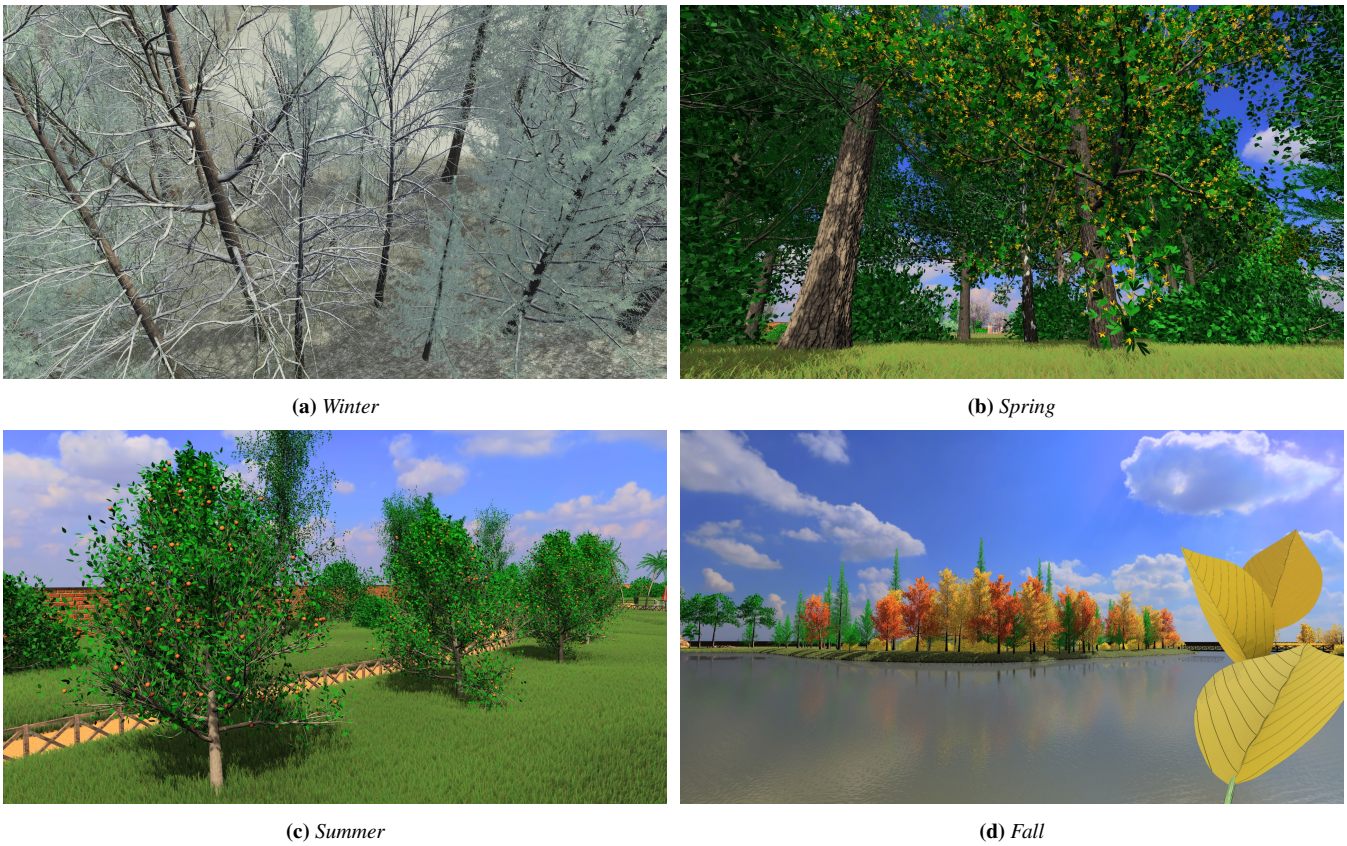
High-Performance Graphics (2025)
A. Knoll and C. Peters (Editors)

# Real-Time GPU Tree Generation

Bastian Kuth[1] Max Oberberger[2] Carsten Faber[1] Pirmin Pfeifer[2] Seyedmasih Tabaei[1] Dominik Baumeister[2] Quirin Meyer[1]

[1]Coburg University of Applied Sciences and Arts, Germany [2]AMD, Germany



**(a)** *Winter* **(b)** *Spring*

**(c)** *Summer* **(d)** *Fall*

**Figure 1:** *From 51 KiB of data in our test scene, we generate and directly render the 1,200 individual trees and bushes of 20 different types using work graphs in real-time. When represented as a conventional static triangle mesh with positions, normals, and texture coordinates, the tree geometry would amount to 34.8 GiB. On our test path, we measure a median frame-to-frame time of 7.74 ms for generating and rendering the trees, leaves, needles, fruits, and blossoms, as well as rendering the entire scene, grass, and visual effects. Our model supports procedural displacement, seasonal changes, complex pruning, animation, culling, continuous LOD, and intuitive artistic control with real-time edits.*

**Abstract**

*Trees for real-time media are typically created using procedural algorithms and then baked to a polygon format, requiring large amounts of memory. We propose a novel procedural system and model for generating and rendering realistic trees and similar vegetation specifically tailored to run in real-time on GPUs. By using GPU work graphs with mesh nodes, we render gigabytes-worth of tree geometry from kilobytes of generation code every frame exclusively on the GPU. Contrary to prior work, our method combines instant in-engine artist authoring, continuous frame-specific level of detail and tessellation, highly detailed animation, and seasonal details like blossoms, fruits, and snow. Generating the unique tree geometries of our teaser test scene and rendering them to the G-buffer takes 3.13 ms on an AMD Radeon RX 7900 XTX.*

**CCS Concepts**

• ***Computing methodologies*** → ***Rendering; Mesh geometry models;*** *Parallel algorithms;*

## 1. Introduction

3D geometry of highly detailed trees for interactive media are typically created with procedural tools [Uni24, Xfr22, Sid23, Ble25]. There, a *procedural model* uses few user parameters to generate 3D geometry of a complex tree. However, computing the 3D geometry may take up to several seconds. As a result, for fast rendering, trees are exported multiple times at different levels of detail (LODs) to a 3D geometry format, requiring large amounts of memory. Graphics processing unit (GPU) performance has grown exponentially over the years, but memory capacity and speed is increasing much slower than compute capability. Thus, the degree of detail is more and more limited by the memory capacity and speed. Geometry compression [MKSS12, KOK*24, MSS24] may increase the number of trees in GPU memory by about an order of magnitude. But, the data that specifies a tree type in a procedural model is surprisingly small and usually takes only hundreds of bytes. This is several orders of magnitudes less than the generated 3D model. Thus, evolving a procedural model directly on the GPU provides an extremely high compression. With the advent of *GPU work graphs* [Mic24], a GPU can dynamically generate work for itself. This greatly simplifies on-chip procedural generation [KOF*24]. We conclude that the next logical step for achieving higher-detailed vegetation without a hefty memory footprint, is to generate more geometry on the fly, in the LOD required for the current frame. Therefore, we make the following contributions:

- We extend Weber's and Penn's procedural tree model [WP95] incorporating smooth stem splines, procedural displacement, leaves, needles, seasonal changes, complex pruning, and animation – all tailored to run in real-time with intuitive artistic control.
- We present a real-time GPU work graphs implementation of this model. It directly forwards generated geometry to the rasterizer keeping memory traffic low. Thus, we generate and render detailed scenes with vegetation, as in Figure 1, in a few milliseconds. It supports culling, continuous LOD, and real-time edits.
- We are the first to utilize *mesh nodes*, a new work graphs extension, and provide a brief introduction.
- We propose several novel techniques applicable for GPU work graphs, like specialized nodes, work coalescing, record compression, and pass fusion for deferred shadow mapping.
- We propose automated continuous LOD selection for maintaining a constant target frame-rate.

## 2. Background

Plant generation uses rule-based algorithms [Hon71], grammars [AK84], particle systems [RB85], fractals [Opp86], L-systems [Pru86], and biologic principles [dREF*88, PHL*09]. Other methods provide explicit artistic modelling control [WP95, LD96, LD98, LRBP12] or create tree geometry from images [NFD07, LWG*21]. Recently, Li et al. [LSP*24] propose to use volumetric strands. Inverse modelling reconstitutes model parameters from a 3D tree model [SPK*14]. Neural networks directly learn branching rules [ZLB*24] or L-system grammars [LLB23].

To compute leaves, biologically motivated algorithms [RFL*05] and methods that generate leaves inside a predefined outline [HSVGB05, KK17] exist. Garg [Gar11] models leaves with simple parameters to generate single- and multi-lobed leaves.

L-systems can be evaluated quickly on GPUs [LH04, LWW10]. Kohek and Strnad [KS14] propose a two-level geometry-shader-based GPU implementation that significantly speeds tree generation over its serial implementation [PHL*09]. However, the method suffers from high memory consumption, and generation runs in the order of a second. Large-scale forest require a more complex LOD, and generation takes multiple seconds [KSvK19]. Kohek and Strnad [KS18] further provide a GPU implementation for a particle-flow tree generation method [RCSL03]. Their incremental generation reverts to volumetric LOD for not-yet generated or distant trees. They limit generation-time budget per frame to 10 ms and achieve 12 – 50 frames per second (fps) for 500,000 trees. Fully generating 2,000 detailed trees takes about 0.8 s. Marvie et al. [MBG*12] evaluate shape grammars with geometry and tessellation shaders to create one tree in ca. 40 ms. Steinberger et al. [SKK*14a] evaluate the same tree with their system in 0.84 ms. Kuth et al. [KOF*24] generate ivy in real-time with GPU work graphs. For wind interaction, tree models simulate wind during generation [WP95, PNH*14] or use a skeleton-like hierarchy on an already generated tree [Zio07, HKW09, QYH*18]. Steinberger et al. [SKK*14b] combine GPU shape-grammars with culling and LOD to generate and render cities in real-time.

### 2.1. The Weber-Penn Model

We choose the *Weber-Penn model* [WP95] as our basis: It efficiently maps to GPU work graphs and produces convincing 3D models of arbitrary complexity. Further, we find it more accessible to artists than grammars for which GPU implementations exist [LWW10, MBG*12, SKK*14a]. It does not suffer from high memory requirements [KS14, KSvK19] or from additionally generating a volume [KS18]. Finally, it natively supports animations.

Weber and Penn model their trees with up to four *levels* of *stems*: One level 0 stem is the trunk of the tree. A level $i$ stem has *child* branches of level $i+1$, which protrude from their parent. The last level of stems has leaves as children. Each level is associated with user configurable *parameters* regarding its curvature, length, radius and taper, or number and placement of children. In addition to this level hierarchy, the model also supports stems *splitting* into multiple *clones* with the same set of parameters. Clones continue their growth where their clone template left off, but with a different trajectory each. The trajectory of a stem level is defined by generated transformations, consisting of a position and an orientation each. Each transformation forms a ring. In the original implementation, for the stem geometry, two succeeding rings are then connected by a truncated cone, called a *segment*. One of the user-configurable parameters altering a stems trajectory is the *upwards attraction*. A positive upward attraction causes branches to grow towards the sun. A negative one models gravity pulling stems down. Leaf shapes are selected from a predefined list and are not procedural. See Fig. 2 for an example of a two-level Weber-Penn tree.

### 2.2. Mesh Nodes

We call a GPU program a *shader*, which is executed in *thread groups*. A thread group consists of one or multiple single instruction, multiple data (SIMD) *waves* of usually 32 threads. A *GPU*

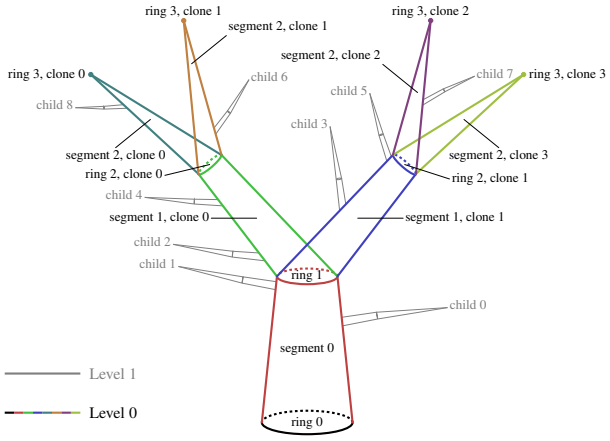



**Figure 2:** *Weber-Penn Model. We mark the different elements of an example tree with two levels. Level 0 splits into four clones.*

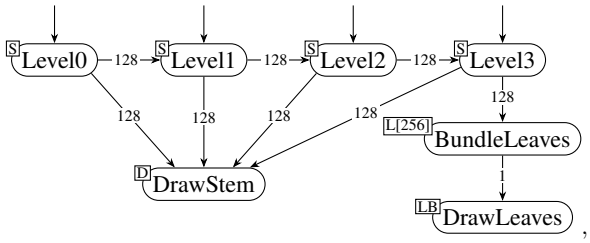
| | thread | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
|---|---|---|---|---|---|---|---|---|---|
| Iteration 0 | compute ring 1, clone... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | write D for segment 0, clone... | 0 | – | – | – | – | – | – | – |
| | write S for child... | 0 | 1 | – | – | – | – | – | – |
| Iteration 1 | compute ring 2, clone... | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| | write D for segment 1, clone... | 0 | – | – | – | 1 | – | – | – |
| | write S for child... | 2 | 4 | – | – | 3 | 5 | – | – |
| Iteration 2 | compute ring 3, clone... | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| | write D for segment 2, clone... | 0 | – | 1 | – | 2 | – | 3 | – |
| | write S for child... | 8 | – | 6 | – | 7 | – | – | – |

**Table 1:** *Stem Level Thread Allocation. We show our thread allocation for computing the trunk in Fig. 2 for an eight-wide wave. Inactive threads are marked with "–". At each split, the threads get divided up into colored blocks, one per clone.*

*work graph* [Mic24] is a directed graph of shader *nodes*. Each node can create *work records* for other nodes. Record execution scheduling is handled by the hard-/firmware itself. While being previously limited to compute shaders, new *mesh nodes* allow for direct output of geometry to the rasterizer. Mesh nodes may not create work records and are thus leaf nodes. A mesh node contains a mesh shader, an optional pixel shader, and all other state associated with a rendering pipeline, like viewport and culling settings. The *mesh launch mode* spawns a grid of thread groups, similar to a mesh shader draw call.

## 3. The Tree Generation Work Graph

The only input to our method is a scene with tree positions, and parameter sets of tree *types*. In its simplest form, our work graph looks like the following:




, (1)

where we mark each (node) with the record [type] struct it receives, and each edge with the maximum possible number of output records. We deduced output counts from the modeled tree types and the work graphs total limit of 256. The *stem struct* [S] contains the first transformation of a stem level, consisting of a position and rotation quaternion, together with other information for generating a Weber-Penn stem level. The nodes (Level0-3) perform cloning when required, write stem segments to be drawn, and create records for their children. For more detail see Sec. 3.1. To assure that there is a leaf output in the end of the level hierarchy, trees with four stem levels enter the graph at (Level0), trees with three levels at (Level1), etc. The *draw stem struct* [D] contains all information to draw one stem segment using the (DrawStem) node described in Sec. 3.2.

The *leaf struct* [L] contains all information to draw one leaf. As one leaf does not have enough geometry to fill a mesh shader group, we employ an extra node (BundleLeaves) in coalescing launch mode, that collects up to 256 [L] into one *leafBundle struct* [LB] before dispatching it to (DrawLeaves) in Sec. 3.3. All the record structs also contain the respective index referencing the tree type and a random seed that is derived from the seed of the parent node.

### 3.1. Stem Level

The (Level0–3) nodes receive an initial stem transformation, compute the stem trajectory, write segment records for drawing that stem, and create records for children branching away from them.

**Thread Allocation** Implementing the stems splitting into clones could be solved by outputting new work records at each split, but early experiments showed that the work graph launch overhead is too high. Instead, we propose to perform the splitting between the threads of a wave, thus we launch one group of 32 threads per [S] using the broadcasting launch mode. Consider the wave in Tab. 1 simplified to eight threads instead of 32 for brevity. Initially, all threads of our example wave belong to the only clone 0. On each split into $s$ clones, we separate the $t$ threads into *blocks*, where a block receives $\lfloor \frac{t}{s} \rfloor$ threads each. In a block, the thread with the lowest index is responsible for writing the record for drawing the segments of that clone. Child record writing is distributed among the threads of the respective block. This approach can be extended to multiple waves to support more than 32 clones, but we did not model a tree type where this is required.

**Hierarchical Generation Culling** Writing of a child record is omitted, if a conservative bounding capsule lies outside the view and shadow frustum. This way, generation only runs for children that have the chance to generate visible geometry. Similar culling applies to writing the [D] or [L] records.

**(a)** *surface* **(b)** *Worley* **(c)** *fractal Perlin* **(d)** *bark*

**Figure 3:** *Procedural Bark Displacement. For our bark (d), we offset the (a) original surface by a weighted sum of (b) clamped Worley [Wor96] and (c) multiple frequencies of Perlin [Per02] noise.*

### 3.2. Stem Drawing

The (DrawStem) node is a mesh node that receives one [D] as input, containing two stem transformations $\vec{P}_{\text{from}} \in \mathbb{R}^3, \vec{R}_{\text{from}} \in \text{SO}(3)$ and $\vec{P}_{\text{to}} \in \mathbb{R}^3, \vec{R}_{\text{to}} \in \text{SO}(3)$. To smooth out the coarse cone geometry from the original Weber-Penn model, we interpret a stem as a cubic Hermite spline tube, where a stem segment is also a spline segment. The spline control points of the segment are the positions $P_{\text{from}}$, $P_{\text{to}}$. For the Hermite spline tangents, we extract the forward axis of the quaternions $R_{\text{from}}, R_{\text{to}}$ and scale them by the distance $\| P_{\text{to}} - P_{\text{from}} \|$. When writing a [D] record, the nodes (Level0–3) compute the following quantities:

**Pixel Space Dimensions** Different tessellation factors between segments would lead to cracks. To have the same vertices between segments, we define the factor for each ring in-between. For this, we project the radius of a ring into the pixel space. If both radii $r_{\text{from/to}}$ of a segment are smaller than a user defined threshold, e.g., one pixel, we cull the segment. In addition, we project the two ring centers of one segment into pixel space and compute their distance in pixels $l$ as a sufficient estimation of the spline arc length.

**Opening Angle** To mitigate the generation of back-facing triangles, we approximate the required tessellation opening angle $\theta$ for each ring. See Fig. 4 for reference. Let $r$ be the ring radius and $z$ the rate in growth direction in world units. Then, $\frac{dr}{dz}$ is the change of radius at the ring. Given the cosine $x$ of the angle between the camera direction and the growth direction, the required opening angle

$$\theta = \cos^{-1}\left(\max\left(\frac{dr}{dz}\cdot\frac{x}{\sqrt{1-x^2}}, -1\right)\right).$$

As we assume orthographic projection, $\theta$ is slightly more conservative than needed for perspective projection. We also neglect the radius variations caused by procedural displacements, but did not find a case where this is a problem.

**Tessellation Factors** For stem tessellation, we employ *fractional odd spacing* as shown in Figs. 4c and d. In the simplest case, a stem is represented by a quad of four vertices and two triangles, where we force $\theta = \frac{\pi}{2}$. Going finer, two more vertices get smoothly interpolated into the edge. A segment requires four tessellation factors: For the axis around the stem radius, the two outer tessellation factors $f$(rom) and $t$(o) at the rings, and the inner tessellation factor for this axis $u$. For the other axis along the stem only the factor $v$ is required, as no neighboring patch exists. With a user-defined constant *pixels per triangle* $\Delta$, the opening angles $\theta_{\text{from/to}}$ in radians, the radii at the rings $r_{\text{from/to}}$, the tessellation factors are:

$$f = \frac{2\cdot\theta_{\text{from}}\cdot r_{\text{from}}}{\Delta}, \quad t = \frac{2\cdot\theta_{\text{to}}\cdot r_{\text{to}}}{\Delta}, \quad u = \frac{f+t}{2}, \quad v = \frac{l}{\Delta}.$$

**Displacement** At close viewing distance, we blend in a procedural bark displacement map, see Fig. 3 for details. In the pixel shader we also evaluate the displacement as a normal map, together with other procedural physically based rendering (PBR) maps.

### 3.3. Leaves

Leaves grow from the last level of stems offset by a parameterized leaf stem length. Our leaf model is inspired by Garg [Gar11], who describes the shape of leaves with quadratic B-spline curves. We prefer it over other models [RFL*05, HSVGB05, KK17], because of its speed and artistic expressiveness. For fast and pixel-perfect evaluation of smooth leaf edges, we use the pixel shader inside/outside test for quadratic Bézier curves proposed by Loop and Blinn [LB05]. If a pixel falls outside the quadratic Bézier curve defined by the triangle, we discard it, similar to opacity mapping.

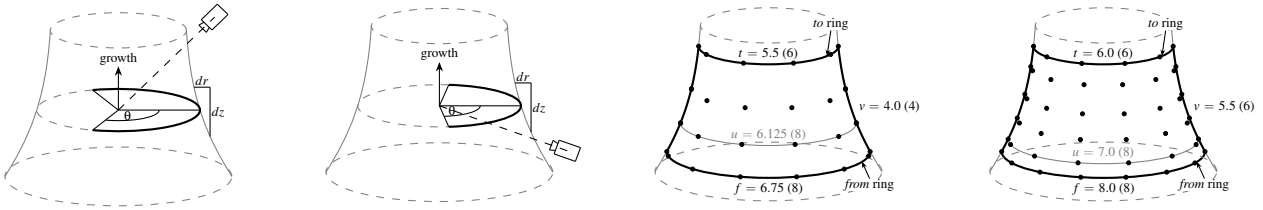
**Geometry** A leaf consists of one or multiple *lobes*. The user defines one half of the shape of a lobe per tree type with the parameters shown in Fig. 5a. From that we compute vertices $\vec{v}_i \in \mathbb{R}^2$. $\vec{v}_0, \vec{v}_4, \vec{v}_8$ are set as in Fig. 5a. We compute in-between vertices $\vec{v}_1$ to $\vec{v}_3$ of Fig. 5b from a weighted sum:

$$\begin{bmatrix}\vec{v}_1\\ \vec{v}_2\\ \vec{v}_3\end{bmatrix} = \frac{1}{4}\begin{bmatrix}4 & 2 & 0 & 0\\ 2 & 1 & 2 & 1\\ 0 & 0 & 4 & 2\end{bmatrix}\cdot\begin{bmatrix}\vec{v}_0^{(x)} & \vec{v}_0^{(y)}\\ \sin(\alpha)\,l & \cos(\alpha)\,l\\ \vec{v}_4^{(x)} & \vec{v}_4^{(y)}\\ \sin(\beta)\,l & \cos(\beta)\,l\end{bmatrix},$$

where the tangent scale $l = \frac{\|\vec{v}_4 - \vec{v}_0\|_2}{2}$. $\vec{v}_5$ to $\vec{v}_7$ are computed analogously. Note that the number of inner triangles increases when a Bézier segment is not convex. Multiple lobes are determined with the parameters of Fig. 5c. As the vertices and triangles of one leaf do not adequately fill the output of a mesh shader group, we need to combine multiple. Similar to the work graph instancing [KOF*24], we use (BundleLeaves) in coalescing launch mode to combine up to 256 small draw records of leaves [L] into one big leaf bundle [LB].

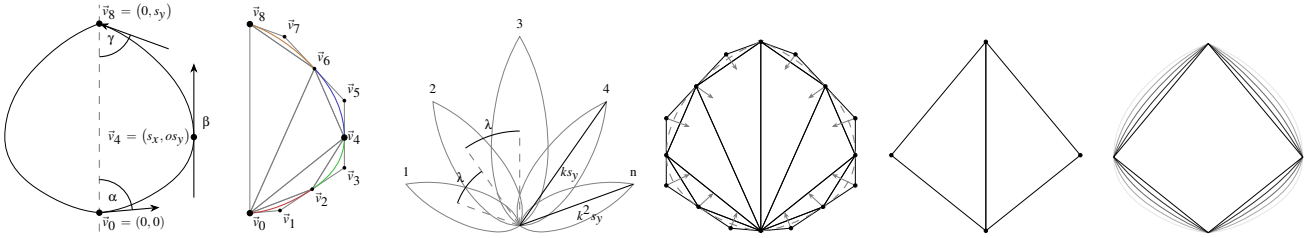
**Level of Detail** For a continuous leaf LOD, we reduce the amount of geometry per leaf at increasing distance as shown in Fig. 5d to f. As soon as the lower LOD is reached, we use a different mesh node (DrawLeavesLow). This is implemented with a work graph node array for (BundleLeaves). For multi-lobe leaves, we shrink the rotation angle $\sigma$ to merge all lobes into one at higher distances. In addition, we employ the common trick to remove leaves at large distances and compensate by increasing the scale of the remaining ones. To make this continuous, disappearing leaves slowly shrink.

**Material and Needles** For more leaf detail, our pixel shader adds a line strand in the center and parabolic strands to either side of the leaf, affecting the base color, normal, and roughness of the surface. The strand count, spacing, curvature, and thickness are further leaf parameters shown in Fig. 6a. At very close distances as in Fig. 6b,



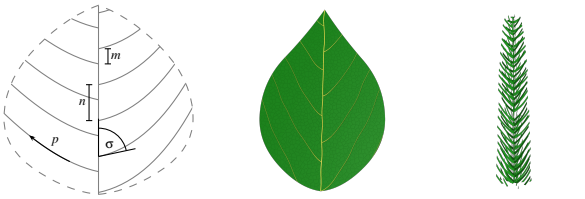

**(a)** *high angle, large* θ **(b)** *low angle, small* θ **(c)** *lower tessellation* **(d)** *higher tesselation*

**Figure 4:** *Segment Tessellation. We highlight the forward-facing part of a stem ring for a high (a) and low (b) camera angle. In (c, d), we provide examples with* $\theta = \frac{\pi}{2}$ *of different fractional tessellation factors and indicate the actual, ceiled tessellation factors in parentheses.*




**(a)** *parameters* **(b)** *triangulation* **(c)** *lobes* **(d)** *high LOD* **(e)** *low LOD* **(f)** *LOD interpolation*

**Figure 5:** *Leaf Geometry. (a) The user defines the lobe shape by the angles* $(\alpha, \beta, \gamma)$*, an offset o, or a scale* $(s_x, s_y)$*. (b) This results in Bézier curves, shown in different colors, with their corresponding tessellation for later filling the leaf interior. (c) For multilobe leafs, the user defines the lobe count n, the angle between lobes* λ*, and a lobe scale-falloff k. (f) For continuous LOD, we interpolate the (d) high-LOD positions to the edges of the (e) low-LOD.*




**(a)** *strand parameters* **(b)** *final leaf* **(c)** *final needles*

**Figure 6:** *Leaf Strand Parameters. Parameters for the curvature exponent p, the rotation angle* σ*, the strand spacing n, and the side offset m define the strands in (a). We show the strands in our final renderer for (b) a leaf, and (c) a needle leaf.*

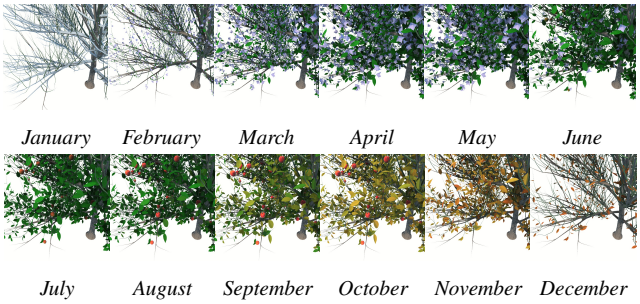



**Figure 7:** *Seasons. We show how the generation of an apple tree changes over the course of a year.*

we add cellular detail with Worley noise [Wor96]. In case of a needle leaf in Fig. 6c, we use the same strands for opacity mapping.

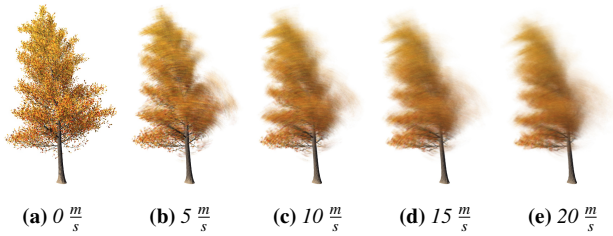
### 3.4. Seasons

We specify a season parameter $S \in [0,4)$, where 0 corresponds to full winter, 1 to spring, etc. We also support in-between values, resulting in the continuous changes of Fig. 7. Tree species do not necessarily respond to seasonal changes simultaneously. To model this, we introduce a seasonal offset parameter per tree type. We consider summer as the default season. Next, we describe what we change to obtain other seasons:

**Snow** For winter of Fig. 1a, we omit rendering of leaves. Needles receive a white frosty outline. To model snow on stems, we increase the displacement from Fig. 3 for stem portions pointing upwards. These extruded regions receive snow material properties. We also decrease the vertical stem attraction to make stems bend down with increasing snow load.

**Blossoms** In spring, the leaves in Fig. 1b grow back. We add blossoms rendering them like leaves with lobes arranged in a circle.

**Fruits** In summer, a random fraction of blossoms grows into fruits of Fig. 1c. We use a cubic Bézier curve to model one side of a fruit profile. This profile is then rotated around the fruit's vertical



**(a)** $0\ \frac{m}{s}$ **(b)** $5\ \frac{m}{s}$ **(c)** $10\ \frac{m}{s}$ **(d)** $15\ \frac{m}{s}$ **(e)** $20\ \frac{m}{s}$

**Figure 8:** *Wind Speed Comparison. We accumulate the motion of a black tupelo tree in fall under varying wind speeds over a 10 second period. Wind blows from the right and pulls the branches and leaves to the left. Also see supplemental video.*

axis and tessellated in a mesh shader. Fruits decrease the vertical attraction of their father stem.

**Shedding** In fall, leaf colors of Fig. 1d change from yellow to red to brown. Later, leaves shed and disappear. Each leaf has a seasonal offset, making leaves vanish gradually.

### 3.5. Animation

The Weber-Penn model describes basic wind animation but ignores wind direction. We incorporate wind direction during stem generation, by rotating each segment away from the wind source. The amount of rotation chaotically oscillates with the current timestamp using Perlin noise: We scale the amplitude by the segment radius and the frequency by the inverse square-root of the stem length. The succeeding segments and children are generated with the animated transformation of the current segment. This implicitly gives us a highly detailed skeleton with one bone per segment without the need for explicitly storing it. For the Sassafras tree of Tab. 2 this would amount for 16.5 k bones. Thus, we obtain the highly detailed animations shown in Fig. 8 and the supplemental video. If more control over the animation is required, we suggest adding an additional density parameter to adjust the elasticity of a stem.

### 3.6. Advanced Editing

To add better real-time editing capabilities, we extend the stem trajectory computation of (Level0–3).

**Custom Spline Editing** We can bias the stem transformations of a tree trunk towards $b$ user-defined transformations. Let $c$ be the number of stem segments. The $i$-th segment transformation with $0 \leq i < c$ is biased towards the relative next user transformation $j = \left\lfloor \frac{i \cdot b}{c} \right\rfloor, 0 \leq j < b$. We use this feature in Sec. 4 with two custom transformations to model hedge arches of our park test scene.

**Object Pruning** Vegetation growing through solid walls is a common immersion-breaker. We can decide to shorten stems if they collide with certain objects. We use hardware ray-tracing to shoot rays into the growth direction [KOF*24]. If there is a hit, we shorten the length of the stem accordingly.

### 3.7. Advanced Optimizations

**Mesh Group Coalescing** In early experiments, we noticed a drastic performance improvement by combining multiple dispatches to mesh nodes into a single one. Thus, similar to the (BundleLeaves) node, we also employ a bundler node in coalescing launch mode before launching any mesh node. Note that this optimization might not apply to other architectures or driver versions.

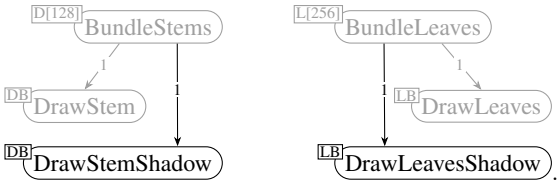
**Quad Stem** Dispatching a full mesh shader group for small stems only consisting of two triangles and four vertices is wasteful. To optimize this, we employ a special bundler and mesh node that renders a bundle of several of these quads per group, similar to the leaves bundling. This is again implemented as a node array. In the future, it could be feasible to employ even more specialized mesh shaders for different tessellation scenarios.

**Stem Leaves Mesh Node** As described in Sec. 3.2, we omit thin stems for rendering. We extend this idea by creating a special mesh node (DrawStemLeaves) that creates and samples a stem spline to only output the geometry for the leaves. The actual stem geometry is not drawn. With this node, we can skip the generation of the last stem level at far camera distances.

**Record Compression** To save on memory bandwidth and capacity during work graph execution, we propose to compress work records when submitting them and to decompress again on node launch. Besides using trivial bitfield compression of limited range integers like the tree level or type, we extend the octahedron mapping for unit vectors [MSS*10] to compress the rotational component of the stem transformations, thus a unit quaternion. For compression, we quantize the imaginary parts of the quaternion in L1 norm into 10 bits each and store them, along with the sign bit of the original real component, in 32 bits. This enables storing an entire stem transformation in 4 hardware dwords. While it is often feasible to omit the sign bit of the real component by negating the whole quaternion when $r < 0$, we include it for correct interpolation of the stem tessellation frames.

### 3.8. Pass Fusion for Deferred Shadow Mapping

We use deferred rendering [DWS*88] with shadow mapping [WSP04]. That would typically require a *geometry pass* to fill the G-buffers, a *shadow pass* to create a shadow map, and a *composition pass* to compute lighting. This, however, would force us to store the full generation result, or to generate geometry twice, i.e., once for the geometry and once for the shadow pass. Instead, we propose to *fuse* geometry and shadow pass and run the inner work graph nodes of our generation only once. For this, we submit to an extra mesh node that creates our shadow map in our graph of (1):




(DrawStem) and (DrawLeaves) are the nodes that fill the G-buffers. We duplicate them to (DrawStemShadow) and

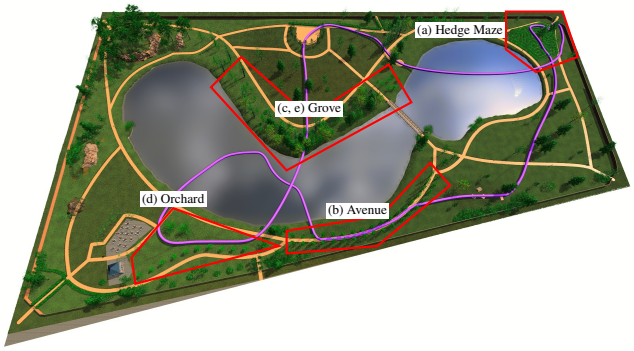



**Figure 9:** *Park Test Scene. Our scene is based on the Ørstedsparken public park in Copenhagen, Denmark. In pink, we visualize our test camera path going through different (a–e) regions encircled in red.*

| | One Sassafras Tree | | Trees of Entire Scene | |
|---|---|---|---|---|
| **Stem Vertices** | 1,479,896 | 45.2 MiB | 493,756,958 | 14.7 GiB |
| **Stem Triangles** | 2,038,580 | 23.3 MiB | 684,357,207 | 7.6 GiB |
| **Leaf Vertices** | 1,974,720 | 60.3 MiB | 302,936,912 | 9.0 GiB |
| **Leaf Triangles** | 1,974,720 | 22.6 MiB | 302,936,912 | 3.4 GiB |
| **Total** | | **151.4 MiB** | | **34.8 GiB** |

**Table 2:** *Theoretical Memory. We list* vertex *and* triangle *count and theoretical memory requirements for the* stem *and the* leaves *of one* Sassafras tree *and tree geometry of the* entire scene. *As we create the required geometry on the fly, the actual permanent memory requirements for the geometry is only ca. 51 KiB.*

DrawLeavesShadow to create our shadow map. For those nodes, we can apply several shadow-map generation optimizations: we create geometry at a lower level-of-detail, evict the pixel shader, and tune pipeline-state and mesh shader. Note that Level0-3 must execute on the union of the light and the camera frustum.

Shadow nodes and non-shadow nodes must operate on distinct depth buffers. However, we cannot switch depth buffers during a work graph launch. Instead, we create a single depth buffer large enough for both tasks. Then, we partition it into two *texture array slices*, and set the corresponding slice index in the mesh shaders.

### 3.9. Automatic Level of Detail

Dynamic resolution scaling (DRS) [Bin11] is a common optimization for video games. Here, the rendering resolution is dynamically reduced for demanding scenes to meet a target frame time. However, sudden resolution changes are conspicuous and can break effects based on temporal accumulation. As our generation runs every frame and changing quality is continuous, we can dynamically adjust the geometric detail of our trees. For each quality parameter, like the pixels per triangle from Sec. 3.2 or the leaf density from Sec. 3.3, the user configures an acceptable range. The parameter ranges are then ordered by priority: the least noticeable parameter gets degraded first if the performance goal was not met in the last frame. If the performance is better than the goal, the parameters are reverted again. Constraining the maximum rate of change of the parameters assures that no sudden quality change occurs.

## 4. Results and Discussion

We evaluate the Direct3D12 implementation of our method. All measurements were taken on an AMD RX 7900 XTX GPU, driver version 24.30.31.03, at a 1920×1080 resolution. We use the scene of Fig. 9. It spans 6.6 hectares and is filled with 1,200 trees and bushes of 20 different types, including 17 with custom splines.

**Memory Requirements** Tab. 2 shows the theoretical memory needed to render our test scene without our method. We assume the highest static discrete tree LOD has all leaves at maximum quality, and stems tessellated with one triangle per centimeter. For reference, the highest quality tree of the PBRT landscape scene [PJH23] amounts to about 4 million triangles, similar to our Sassafras tree. We assume one vertex to consist of a position, a normal, and a texture coordinate, amounting to 32 bytes. The memory required for just the trees of the scene would exceed the 24 GiB memory capacity of our high-end GPU. Note that recreating features of our method such as LODs, animation, seasonal changes, or normal mapping would require significantly more memory. In comparison, our method only requires 704 bytes of parameters per tree type. With 20 tree types, 1,200 initial transformations with a type index, and 17 custom spline positions, this amounts to 51 KiB for the trees in the scene. A work graph requires temporary backing memory to store records, with the size range determined by the driver. Users must select a size within this range. Our work graph, including nodes for grass and asset rendering, as well as debugging, requires 1.5 GiB of backing memory. However, testing across different GPU architectures and driver versions shows significant variation in this requirement. Note that this memory can be reused, freed or re-allocated outside the work graph execution.

**Performance** Fig. 10 plots the time required to generate and render different fractions of the frames from the camera path of Fig. 9. The median tree generation and rendering time to the G-buffer is 3.13 ms. As generation does not have to run twice for the shadow map, measuring it together with the G-buffer amounts to 4.72 ms. Frame-to-frame timings of 7.74 ms additionally contains rendering all other scene geometry including grass, the compositing pass, screen space ambient occlusion and reflections, temporal anti-aliasing, tone mapping, and present. Furthermore, the performance plots follow the total number of rasterized triangles of a frame, except for the orchard region, where we generate many extra triangles for blossoms. In addition, our automatic LOD succeeds in keeping the times under the set target of $8.\overline{3}$ ms.

**Continuous LOD** In Fig. 11, we demonstrate how the tree generation changes at different distances to the camera, and measure the number of triangles. Our method succeeds in preserving the overall appearance of the full detailed 3.6 M triangle tree, while drastically reducing the amount of geometry to 8.7 k when far away. To evaluate how seamless this LOD change is, we take the interval of Fig. 11 from a to b, and demonstrate how our smooth LOD distributes the resulting ꟻLIP [ANA*20] error to several frames compared to a conventional LOD switch. As can be seen, in-between frames fur-

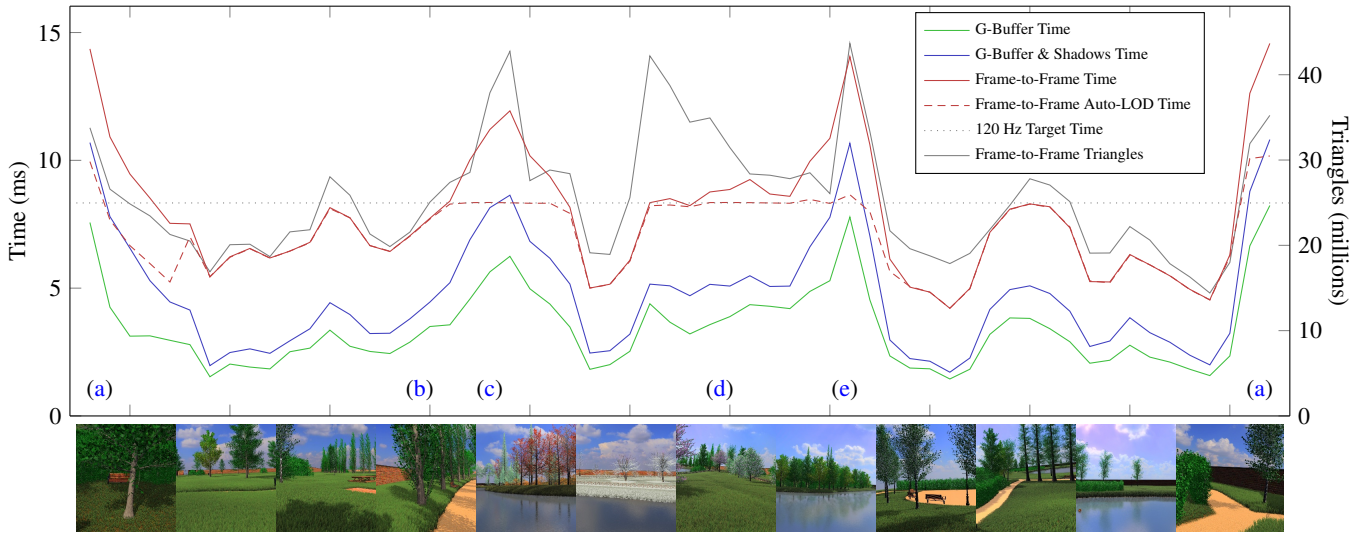



**Figure 10:** *Performance along a Camera Path. We fly through our scene from Fig. 9, measure frame times, and (a–e) mark the corresponding regions from the camera path. In green, we plot the time to generate and render the tree geometry of the current frame to the G-buffer. In blue, we additionally enable creation of the shadow map. Red shows the total frame-to-frame time, including all other graphical effects, with the secondary axis dashed showing the total number of triages for this. The violet plot shows this number with our auto-LOD enabled.*

ther away from the camera have a higher error to their predecessor. This is desirable, because changes at greater distance are less visible. The greatest visual error happens between image (10) and (14), where leaf lobes start to merge as described in Sec. 3.3. This process finishes between image (13) to (14), which explains the sudden decline in number of triangles at 50 m. Note that, for a reasonable camera speed, many more intermediate frames would be generated, as shown in the supplemental video.

**Editing** A user interface allows changing tree parameters. As the parameters fit in ca. 1 KiB, GPU upload times after edits are negligible. Thus, we get real-time feedback making modelling intuitive and efficient. Fig. 13 and the supplemental video show tree edits. Same as the original model, we do not automatically handle tree-self or tree-tree intersection, but can manually prevent them by placing pruning meshes.

## 5. Conclusion and Future Work

We presented a system and model for real-time tree generation on GPUs. By generating tree geometries for the current frame on the fly, our method dramatically reduces memory requirements for detailed vegetation. In future work, we want to explore how real-time ray-tracing can profit from fast vegetation generation.

## Acknowledgments

We thank Niels Fröhling and Gustaf Waldemarson.

## References

[AK84] AONO M., KUNII T. L.: Botanical tree image generation. *IEEE Computer Graphics and Applications 4*, 5 (1984), 10–34. 2

[ANA*20] ANDERSSON P., NILSSON J., AKENINE-MÖLLER T., OSKARSSON M., ÅSTRÖM K., FAIRCHILD M. D.: ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 2 (2020), 15:1–15:23. 7

[Bin11] BINKS D.: Dynamic resolution rendering. In *Game Developers Conference (GDC)* (2011). 7

[Ble25] BLENDER ONLINE COMMUNITY: *Blender 4.3*, 2025. URL: www.blender.org. 2

[dREF*88] DE REFFYE P., EDELIN C., FRANÇON J., JAEGER M., PUECH C.: Plant models faithful to botanical structure and development. *SIGGRAPH Comput. Graph. 22*, 4 (June 1988), 151–158. 2

[DWS*88] DEERING M., WINNER S., SCHEDIWY B., DUFFY C., HUNT N.: The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1988), SIGGRAPH '88, Association for Computing Machinery, p. 21–30. 6

[Gar11] GARG S.: *Procedural Modeling and Constrained Morphing of Leaves*. PhD thesis, National University of Singapore, 2011. 2, 4

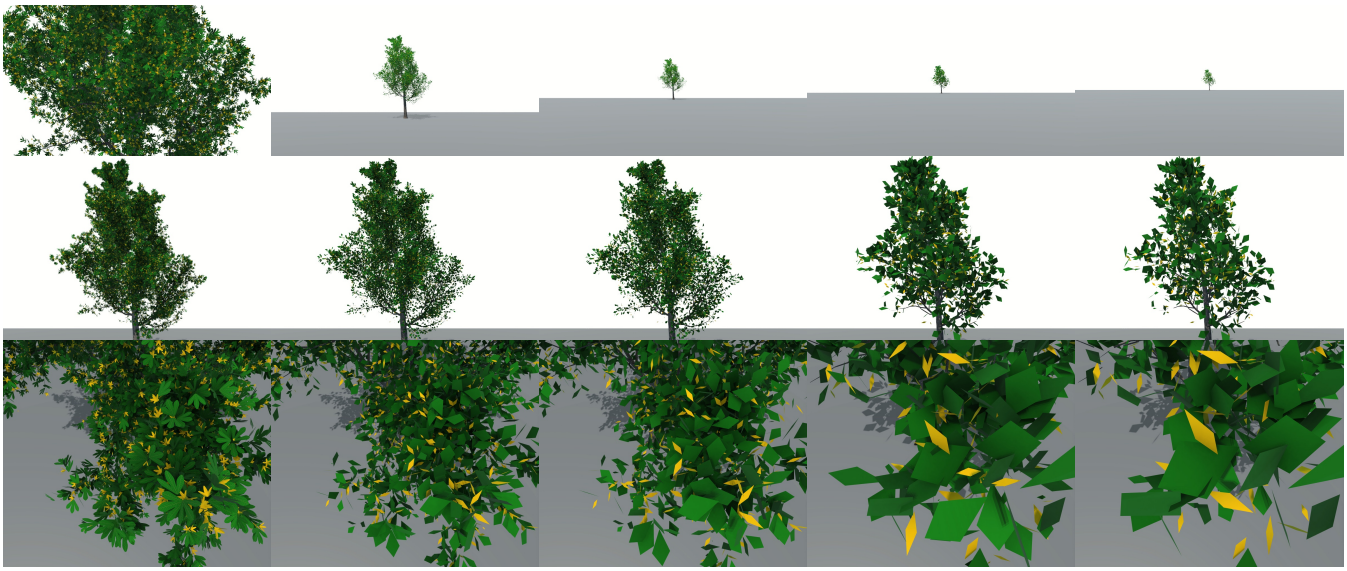
[HKW09] HABEL R., KUSTERNIG A., WIMMER M.: Physically guided animation of trees. *Computer Graphics Forum (Proceedings EUROGRAPHICS 2009) 28*, 2 (Mar. 2009), 523–532. 2

[Hon71] HONDA H.: Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology 31*, 2 (1971), 331–338. 2

[HSVGB05] HONG S. M., SIMPSON B., V. G. BARANOSKI G.: Interactive venation-based leaf shape modeling. *Computer Animation and Virtual Worlds 16*, 3-4 (2005), 415–427. 2, 4

[KK17] KIM D., KIM J.: Procedural modeling and visualization of multiple leaves. *Multimedia Syst. 23*, 4 (July 2017), 435–449. 2, 4

[KOF*24] KUTH B., OBERBERGER M., FABER C., BAUMEISTER D., CHAJDAS M., MEYER Q.: Real-time procedural generation with gpu work graphs. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 7*, 3 (2024), 1–16. 2, 4, 6



**(a)** *10 m, 3.5 M triangles* **(b)** *50 m, 78.9 k triangles* **(c)** *100 m, 44.7 k triangles* **(d)** *150 m, 9.8 k triangles* **(e)** *200 m, 5.8 k triangles*

**Figure 11:** *Continuous LOD. We demonstrate continuous LODs at different camera distances. Top row shows the geometry from the camera the LOD is meant for. Mid and bottom rows show the resulting geometry from fixed perspectives to visualize the geometry simplification.*

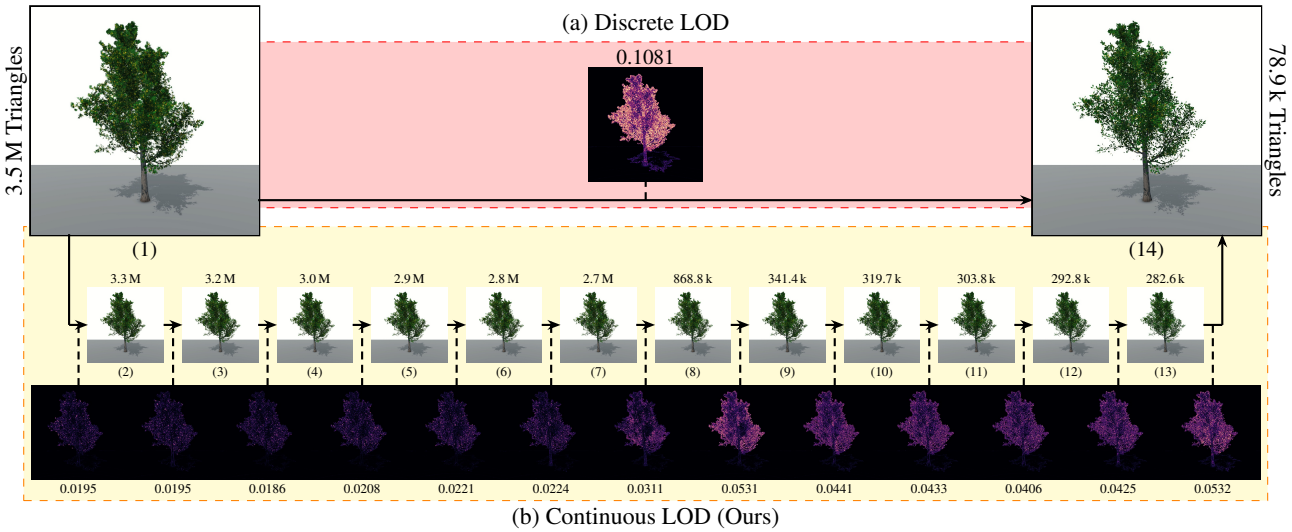
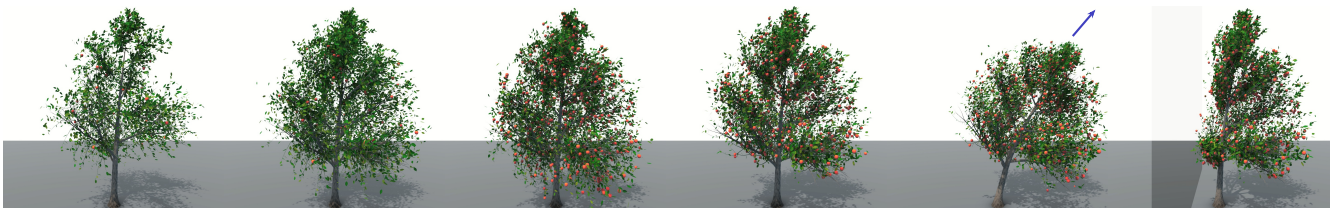



**Figure 12:** *Discrete versus Continuous LOD. (a) shows a conventional, sudden LOD switch from frame (1) to (14) with a high ∇LIP error. In (b), our method can distribute the change over additional frames (2 – 13), also distributing the visible ∇LIP error.*



**(a)** *Baseline* **(b)** *More Branches* **(c)** *More Apples* **(d)** *Vertical Attraction* **(e)** *Custom Spline Bias* **(f)** *Pruning*

**Figure 13:** *Apple tree editing. All edits are visible for the user instantly in the next rendered frame. From a given baseline, we can (b) alter number of branches and (c) apples, (d) push the stems upwards, (e) bend the trunk spline, or (f) prune with an object.*

[KOK*24] KUTH B., OBERBERGER M., KAWALA F., REITTER S., MICHEL S., CHAJDAS M., MEYER Q.: Towards Practical Meshlet Compression. In *Vision, Modeling, and Visualization* (2024), Linsen L., Thies J., (Eds.), The Eurographics Association. 2

[KS14] KOHEK V., STRNAD D.: Interactive synthesis of self-organizing tree models on the GPU. *Computing 97* (02 2014), 145–169. 2

[KS18] KOHEK V., STRNAD D.: Interactive large-scale procedural forest construction and visualization based on particle flow simulation. *Computer Graphics Forum 37*, 1 (2018), 389–402. 2

[KSvK19] KOHEK V., STRNAD D., ŽALIK B., KOLMANIČ S.: Interactive synthesis and visualization of self-organizing trees for large-scale forest succession simulation. *Multimedia Systems 25*, 3 (2019). 2

[LB05] LOOP C., BLINN J.: Resolution independent curve rendering using programmable graphics hardware. In *ACM SIGGRAPH 2005 Papers*. 2005, pp. 1000–1009. 4

[LD96] LINTERMANN B., DEUSSEN O.: Interactive modelling and animation of branching botanical structures. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation '96* (Berlin, Heidelberg, 1996), Springer-Verlag, p. 139–151. 2

[LD98] LINTERMANN B., DEUSSEN O.: A modelling method and user interface for creating plants. *Computer Graphics Forum 17*, 1 (1998), 73–82. 2

[LH04] LACZ P., HART J.: Procedural geometry synthesis on the GPU. In *Workshop on General Purpose Computing on Graphics Processors* (2004). 2

[LLB23] LEE J. J., LI B., BENES B.: Latent L-systems: Transformer-based Tree Generator. *ACM Trans. Graph. 43*, 1 (Nov. 2023). 2

[LRBP12] LONGAY S., RUNIONS A., BOUDON F., PRUSINKIEWICZ P.: TreeSketch: Interactive Procedural Modeling of Trees on a Tablet. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling* (2012), Singh K., Kara L. B., (Eds.), The Eurographics Association. 2

[LSP*24] LI B., SCHWARZ N. A., PAŁUBICKI W., PIRK S., BENES B.: Interactive invigoration: Volumetric modeling of trees with strands. *ACM Trans. Graph. 43*, 4 (July 2024). 2

[LWG*21] LIU Z., WU K., GUO J., WANG Y., DEUSSEN O., CHENG Z.: Single image tree reconstruction via adversarial network. *Graph. Models 117*, C (Sept. 2021). 2

[LWW10] LIPP M., WONKA P., WIMMER M.: Parallel generation of multiple L-systems. *Computers & Graphics 34*, 5 (2010). CAD/GRAPHICS 2009 Extended papers from the 2009 Sketch-Based Interfaces and Modeling Conference Vision, Modeling & Visualization. 2

[MBG*12] MARVIE J.-E., BURON C., GAUTRON P., HIRTZLIN P., SOURIMANT G.: GPU Shape Grammars. *Computer Graphics Forum* (2012). 2

[Mic24] MICROSOFT COOPERATION: *DirectX-Specs*, 2024. URL: https://github.com/microsoft/DirectX-Specs. 2

[MKSS12] MEYER Q., KEINERT B., SUSSNER G., STAMMINGER M.: Data-parallel decompression of triangle mesh topology. *Computer Graphics Forum 31*, 8 (2012), 2541–2553. 2

[MSS*10] MEYER Q., SÜSSMUTH J., SUSSNER G., STAMMINGER M., GREINER G.: On floating-point normal vectors. In *Computer Graphics Forum* (2010), vol. 29, Wiley Online Library, pp. 1405–1409. 6

[MSS24] MLAKAR D., STEINBERGER M., SCHMALSTIEG D.: End-to-end compressed meshlet rendering. *Computer Graphics Forum 43*, 1 (2024). 2

[NFD07] NEUBERT B., FRANKEN T., DEUSSEN O.: Approximate image-based tree-modeling using particle flows. *ACM Trans. Graph. 26*, 3 (July 2007), 88–es. 2

[Opp86] OPPENHEIMER P. E.: Real time design and animation of fractal plants and trees. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1986), SIGGRAPH '86, Association for Computing Machinery, p. 55–64. 2

[Per02] PERLIN K.: Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), pp. 681–682. 4

[PHL*09] PAŁUBICKI W., HOREL K., LONGAY S., RUNIONS A., LANE B., MĚCH R., PRUSINKIEWICZ P.: Self-organizing tree models for image synthesis. *ACM Trans. Graph. 28*, 3 (July 2009). 2

[PJH23] PHARR M., JAKOB W., HUMPHREYS G.: *Physically based rendering: From theory to implementation*. MIT Press, 2023. 7

[PNH*14] PIRK S., NIESE T., HÄDRICH T., BENES B., DEUSSEN O.: Windy trees: computing stress response for developmental tree models. *ACM Trans. Graph. 33*, 6 (Nov. 2014). 2

[Pru86] PRUSINKIEWICZ P.: Graphical applications of L-systems. In *Proceedings on Graphics Interface '86/Vision Interface '86* (CAN, 1986), Canadian Information Processing Society, p. 247–253. 2

[QYH*18] QUIGLEY E., YU Y., HUANG J., LIN W., FEDKIW R.: Real-time interactive tree animation. *IEEE Transactions on Visualization and Computer Graphics 24*, 5 (2018), 1717–1727. 2

[RB85] REEVES W. T., BLAU R.: Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1985), SIGGRAPH '85, Association for Computing Machinery, p. 313–322. 2

[RCSL03] RODKAEW Y., CHONGSTITVATANA P., SIRIPANT S., LURSINSAP C.: Particle systems for plant modeling. *Plant growth modeling and applications. Proceedings of PMA03, Hu B.-G., Jaeger M.,(Eds.). Tsinghua University Press and Springer, Beijing* (2003). 2

[RFL*05] RUNIONS A., FUHRER M., LANE B., FEDERL P., ROLLAND-LAGAN A.-G., PRUSINKIEWICZ P.: Modeling and visualization of leaf venation patterns. *ACM Trans. Graph. 24*, 3 (July 2005), 702–711. 2, 4

[Sid23] SIDE EFFECTS SOFTWARE INC.: *Houdini 20*, 2023. URL: www.sidefx.com/products/houdini. 2

[SKK*14a] STEINBERGER M., KENZEL M., KAINZ B., MÜLLER J., WONKA P., SCHMALSTIEG D.: Parallel generation of architecture on the GPU. *Computer Graphics Forum* (2014). 2

[SKK*14b] STEINBERGER M., KENZEL M., KAINZ B., WONKA P., SCHMALSTIEG D.: On-the-fly generation and rendering of infinite cities on the gpu. In *Computer graphics forum* (2014), vol. 33, Wiley Online Library, pp. 105–114. 2

[SPK*14] STAVA O., PIRK S., KRATT J., CHEN B., MŹCH R., DEUSSEN O., BENES B.: Inverse procedural modelling of trees. *Comput. Graph. Forum 33*, 6 (Sept. 2014), 118–131. 2

[Uni24] UNITY TECHNOLOGIES: *SpeedTree - 3D Vegetation Modeling and Middleware*, 2024. URL: https://speedtree.com. 2

[Wor96] WORLEY S.: A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, Association for Computing Machinery, p. 291–294. 4

[WP95] WEBER J., PENN J.: Creation and rendering of realistic trees. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (1995), pp. 119–128. 2

[WSP04] WIMMER M., SCHERZER D., PURGATHOFER W.: Light space perspective shadow maps. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques* (2004), EGSR'04, Eurographics Association, p. 143–151. 6

[Xfr22] XFROG INC.: *Xfrog*, 2022. URL: www.xfrog.com. 2

[Zio07] ZIOMA R.: GPU-generated procedural wind animations for trees. In *GPU Gems 3*, Nguyen H., (Ed.). Addison-Wesley Professional, 2007, ch. 6. 2

[ZLB*24] ZHOU X., LI B., BENES B., FEI S., PIRK S.: Deeptree: Modeling trees with situated latents. *IEEE Transactions on Visualization and Computer Graphics 30*, 8 (Aug. 2024), 5795–5809. 2