Volume 0 (1981), Number 0 pp. 1–4 COMPUTER GRAPHICS *forum*

Procedural Generation and Realistic Texturing of Wooden Logs Using Blender and Python

Faris Delić

University of Ljubljana
fd4786@student.uni-lj.si

Abstract

Procedural generation is increasingly employed in computer graphics to produce highly detailed and realistic assets with minimal artist intervention. In this paper, I introduce a novel procedural approach to generating realistic wooden logs within Blender, using a combination of parameter-driven geometry construction and physically-based rendering (PBR) materials. My system employs a user-friendly graphical interface for parameter specification, automated JSON parameter parsing, dynamic mesh creation via Blender's Python API, and realistic bark and end-cap texturing. I demonstrate the flexibility and realism of the generated models, discuss the implemented algorithms, present representative results, and outline future avenues for enhancing my pipeline's capabilities.

1. Introduction

Realistic procedural asset generation is crucial in contemporary computer graphics, particularly in areas such as film visual effects, video games, and virtual environments. While substantial research and tools exist for generating entire trees and vegetation, the specific task of modeling individual wooden logs—complete with realistic geometry and textures—remains relatively unexplored. Yet, realistic logs significantly contribute to scene authenticity, especially in forest environments or outdoor game levels.

Manually creating detailed logs can be time-consuming, requiring extensive modeling and texturing efforts. To address this need, I developed an automated system based on Blender's Python scripting environment, which provides an accessible workflow from parameter specification to finished textured models.

This paper describes my procedural generation pipeline, from the initial graphical interface for parameter definition to the generation of detailed geometry and application of physically-based textures. I also explore implementation details and present generated examples to illustrate the system's capabilities.

2. Related Work

Procedural generation of natural phenomena leverages mathematical models and algorithms to automatically produce complex, varied outputs. Perlin noise [**?**] is widely used to introduce controlled randomness, simulating natural irregularities found in textures and shapes. Ebert et al. [**?**] expanded these concepts into comprehensive procedural modeling methods for terrain and natural textures.

Significant research into procedural vegetation modeling typically utilizes L-systems [**?**] or similar branching structures [**?**]. While these approaches effectively handle complex branching patterns, my work specifically addresses the challenge of detailed individual logs. Blender's bmesh module and its Python scripting capabilities offer a robust platform for such specialized procedural tasks [**?**], enabling the sophisticated mesh manipulation techniques I employ.

3. System Overview

My procedural pipeline consists of two primary components: a parameter editing interface and a Blender-based generator script.

3.1. Parameter Specification (`params.py`)

Users specify log characteristics via a graphical user interface built with Python's Tkinter library. Parameters include length, radii (start, center, end), number of vertex rings, taper function (linear, quadratic, exponential), grooves, noise magnitude, bark texture details, and color adjustments. These parameters are serialized into a JSON format for seamless integration with the Blender script.

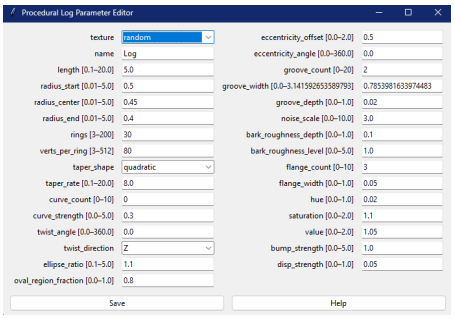


Figure 1: Graphical user interface for parameter specification. The UI allows interactive editing of all geometric and material parameters, which are then saved as JSON files for use by the procedural log generator.

3.2. Procedural Mesh Generation (`logs.py`)

The Blender Python script interprets these JSON parameter files and automatically generates the corresponding 3D models:

1. **Scene Initialization:** Clears existing geometry from Blender's scene to prepare for new log generation.
2. **Cross-sectional Rings Creation:** Constructs vertex rings along the log's length, adjusting radius according to a taper function defined by quadratic Bézier interpolation or alternative user-selected methods.
3. **Perturbations and Details:** Incorporates elliptical cross-sections, flange structures, spiral twists, and surface grooves. Additionally, subtle Perlin noise–based vertex perturbations enhance realism, simulating natural irregularities in bark surfaces.
4. **Mesh Assembly:** Utilizes Blender's bmesh API to connect adjacent vertex rings, forming smooth surfaces suitable for shading and texturing.
5. **End Cap Formation:** Caps log ends with dedicated meshes derived from boundary vertex loops.
6. **Batch Export:** For each JSON file in the parameters folder, the script saves the generated log both as a native Blender file (`.blend`) and as a glTF 2.0 asset (`.glb`) into the designated output directories.

3.3. Material and Texture Application

To achieve realistic appearances, I define physically-based materials using Blender's node system:

- **Bark Material:** Employs image textures for color, roughness, normal mapping, and optional displacement. A hue and saturation node allows color variations to match desired artistic effects.
- **End-cap Material:** Applies dedicated textures to the log ends, reflecting natural wood cross-section patterns.

I dynamically assign materials to corresponding mesh regions within the script, ensuring automated and consistent texturing.

4. Implementation Details

4.1. Mesh Construction Algorithm

I define each vertex position using parametric functions along the log's length parameter $t \in [0,1]$. The radius interpolation is generally represented as:

$$r(t) = (1-t)^2 r_{\text{start}} + 2(1-t)t\, r_{\text{center}} + t^2 r_{\text{end}},$$

though alternative interpolation methods such as linear and exponential functions are supported.

Surface details, including flanges and grooves, are algorithmically embedded using angular and radial perturbation techniques applied at vertex creation time.

4.2. Texturing Procedure

Textures are mapped using Blender's UV projection methods, and the node-based setup ensures consistent PBR shading properties. Displacement maps provide additional depth realism, enhancing bark texture fidelity.

4.3. Influence of Parameters on Geometry Construction

In this section, I describe how each JSON parameter drives the underlying mesh calculations in `generate_proc_log()`.

With this parameterization, I can cover everything from a perfectly straight cylinder (`groove_count`=0, no noise, no twist) to a heavily twisted, grooved, and rough-barked log by simply adjusting JSON values.

4.4. Parametric Model Overview

Rather than listing every equation in detail, this section gives an intuitive description of how each parameter shapes the log geometry:

- **Taper and Radius Interpolation.** The three radii—start, center and end—are blended along the length of the log. You can choose a straight-line blend (linear), a power-curve blend (exponential, controlled by the "taper rate"), or the familiar quadratic Bézier blend (which bows out toward the center radius).
- **Centerline Curve.** To introduce a bend, the script shifts each ring's center in the *xy* plane according to a sinusoidal wave. You specify how many full cycles to fit along the log ("curve count") and how far the wave moves the log off-axis ("curve strength").
- **Twist.** The entire cross section is rotated progressively from base to tip. At the very start there is zero twist, and at the very end the ring has been turned through your chosen total angle, around the chosen axis (X, Y or Z).
- **Elliptical Region.** For the portion of the log from 0
- **Eccentricity Offset.** You can shift the entire ring off-center in the *xy* plane by a fixed distance and angle, simulating a log that grew slightly off its central axis.
- **Groove Carving.** The script carves out a number of longitudinal notches. You choose how many grooves to place around the circumference, how wide each notch is (in angular terms) and how deep to cut into the radius. Wherever a vertex falls inside one of those angular slices, its radius is reduced accordingly.
- **Perlin-Noise Roughness.** To simulate natural bark irregularities, each vertex's radius is jittered by a Perlin noise sample. You control the frequency of the noise (how fine the wrinkles are) and the amplitude (how deep or shallow those wrinkles become).

Parameter	Effect on Geometry / Material
`name`	Identifier only (object name & filename). No mesh impact.
`length`	Scales the z–axis: ring i is at $z = t_i \cdot \text{length}$.
`radius_start`, `radius_center`, `radius_end`	Control points r_0, r_1, r_2 in the quadratic Bézier $r(t) = (1-t)^2 r_0 + 2(1-t)t\, r_1 + t^2 r_2$ defining cross-section radius.
`rings`	Number of longitudinal slices (t steps). More rings $\rightarrow$ smoother taper, higher polycount.
`verts_per_ring`	Vertices per slice. Higher $\rightarrow$ smoother circumference, more faces.
`taper_shape`	`linear`: $r = r_0 + (r_2 - r_0)t$. `exponential`: $r = r_0 + (r_2 - r_0)t^k$ ($k =$ `taper_rate`). `quadratic`: Bézier form above.
`taper_rate`	Exponent k for exponential taper; larger k concentrates taper near $t = 1$.
`curve_count`, `curve_strength`	Sinusoidal axis offset $(x, y) = \big(\sin(2\pi n t), \cos(2\pi n t)\big) \times s$, with $n =$ `curve_count`, $s =$ `curve_strength`.
`twist_angle`, `twist_direction`	Rotate each ring by $\theta = t \cdot$ `twist_angle` about the chosen axis.
`ellipse_ratio`, `oval_region_fraction`	For $t < f$ use $(r_x, r_y) = (r \cdot e, r/e)$, $e =$ `ellipse_ratio`, then revert to circle.
`eccentricity_offset`, `eccentricity_angle`	Offset ring center by $(o\cos a, o\sin a)$ in the xy–plane.
`groove_count`, `groove_width`, `groove_depth`	Carve m notches: if $\lvert\theta - \frac{2\pi k}{m}\rvert < \frac{w}{2}$ then $r \mapsto r - d$, else no offset.
`noise_scale`, `bark_roughness_depth`, `bark_roughness_level`	Perlin noise $N(t \cdot s, \theta \cdot s)$ adds $N \cdot d \cdot \ell$ to radius, controlling frequency and amplitude.
`flange_count`, `flange_width`	Every $\lfloor \text{rings}/c \rfloor$th ring gets an extra radial offset of w, forming ridges.
`hue`, `saturation`, `value`	HSV node shifts on the bark base-color texture.
`bump_strength`	Scales the normal map's effect in the shader.
`disp_strength`	Scales the Displacement Node's height input on the bark.

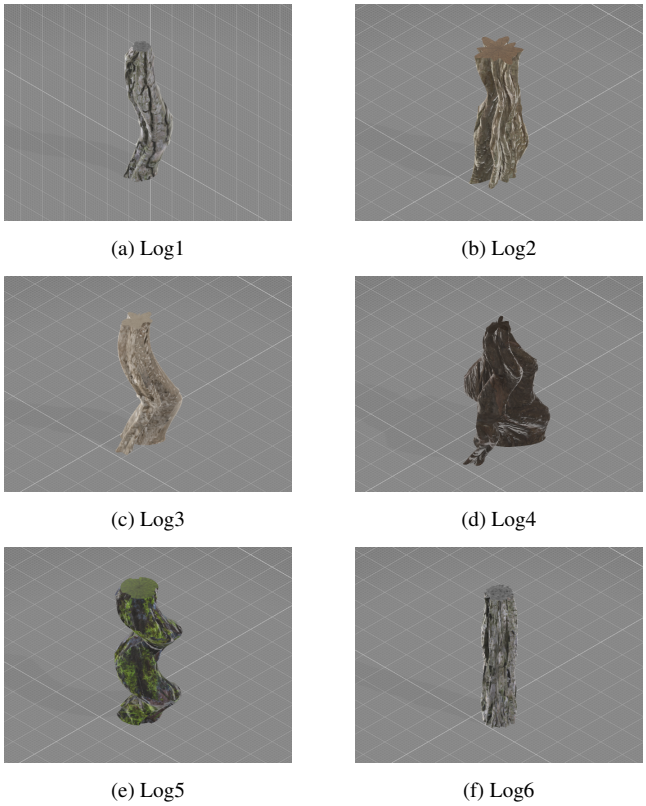
Table 1: How each JSON parameter influences the procedural log generation and texturing.

- **Flanges (Ridges).** Periodically, every so many rings you can add a uniform bump around the entire cross section. This creates raised ridges or "flanges" along the log, as if the bark peeled or thickened in bands.

- **Putting It All Together.** For each ring, the code first computes a base radius from the taper curve, then applies any elliptical scaling and eccentric offset to the ring center. It then reduces the radius for grooves, adds the noise displacement, and bumps it for flanges. Finally, each vertex is placed around that ring at the computed radius and rotated by the twist amount, and the whole ring is translated by the curved centerline offset.

This parametric procedure lets you generate everything from a simple straight cylinder (zero grooves, zero noise, zero twist) to a highly irregular, twisted, grooved log, all by adjusting intuitive JSON parameters.

5. Sample Outputs



(a) Log1 (b) Log2

(c) Log3 (d) Log4

(e) Log5 (f) Log6

Figure 2: Six representative logs generated by my system.

6. Conclusion and Future Work

In this paper I have presented a robust, fully-automated pipeline for procedural generation and PBR texturing of wooden logs in Blender. By driving mesh construction entirely from a human-readable JSON parameter set—and providing a Tkinter GUI to

edit those parameters—users gain fine-grained control over geometry features such as taper profiles, curvature, twist, grooves, noise perturbations and flanges. The accompanying Python scripts handle scene setup, bmesh-based mesh assembly, node-based material creation, and batch export, enabling even complex variants to be generated in under one second on commodity hardware. Through a combination of quadratic Bézier interpolation, sinusoidal centerline offsets, Perlin noise, and shader-node displacements, the system achieves a high degree of realism while retaining simplicity and repeatability.

My results demonstrate that a single, unified parameter interface can span a wide design space—from perfect cylinders to heavily twisted, deeply grooved, and naturally roughened logs—without manual modeling or texture authoring. This approach not only accelerates asset creation for games, VFX, and virtual environments, but also reduces the iteration loop for artists exploring variations. The procedural model's efficiency and flexibility underscore the value of parameter-driven workflows in modern computer graphics.

Future improvements include:

- **Advanced UV packing:** Optimize bark alignment to reduce seams and stretching.
- **Interactive preview:** Integrate real-time parameter changes directly in the 3D viewport.
- **Branch and knot modeling:** Extend my system to procedurally add branches, knots, and bark chipping.
- **User study:** Conduct formal evaluations comparing perceived realism against hand-modeled logs.