# Zelda: Catch the Koroks

Evan Zhang, Ethan Lin, Justin Tien-Smith

CS 4620 Final Project

Cornell University

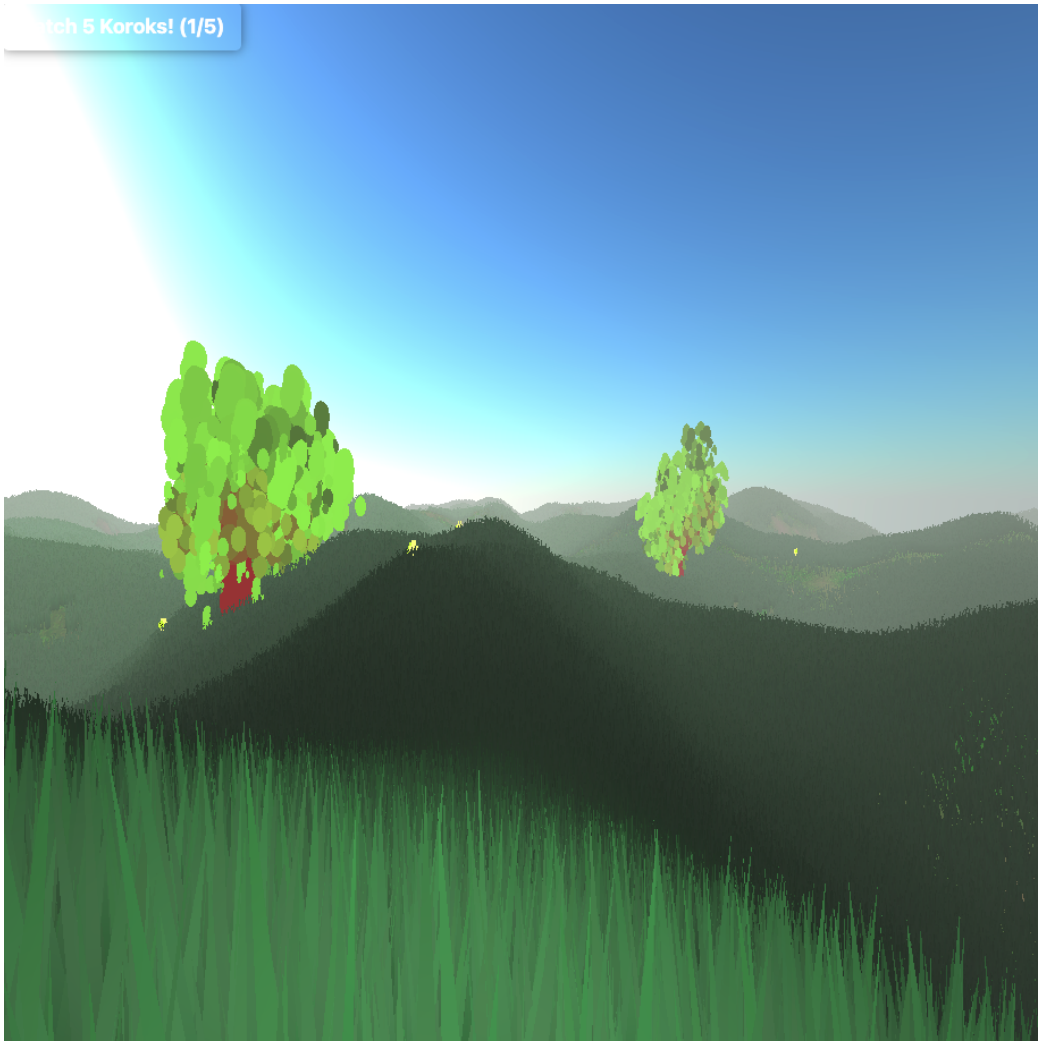


**Figure 1:** *A great big (infinite!) world*

## Group member credits

- Evan Zhang:
  - Terrain, Korok, Interaction Modes Implementation.
  - Gaming Logic and User Interaction Design (music, model, texture, and sound effect).
- Ethan Lin:
  - Grass, Skybox, Lighting, and Fog implementation.
  - Trees shader and optimization (modifying trees to use instanced particles for speed up like the grass).
  - Variable texture mapping for the terrain based on gradients in the shader.
- Justin Tien-Smith:
  - L-system trees and terrain integration.
  - Procedurally generated rocks (scrapped), terrain texture, and helped with grass animation.

## Abstract

**Zelda** is the group's favorite game, and **Koroks** are the group's favorite characters in Zelda. These tiny little cuties are hiding in the bushes, under the rocks, or in the water, waiting for Link to spot them. Our project, *Zelda: Catch the Koroks*, aims to recreate the enchanting world of Zelda by developing an immersive 3D environment where players can explore vast, procedurally generated terrains to locate and collect Koroks. When exploring in our scene, you will hear the most classic Zelda **BGM** and the "**Yah-Hah-Hah**" voice when you collect the Koroks.

## 1 Interaction Mode

We have two first player interaction modes in our application: **inspector** and **player**. Both modes of interaction are implemented with the **angle-axis quaternion** multiplications.

### 1.1 Inspector Interaction Mode

The `Inspector Interaction Mode` allows users to **ignore collision** with the scene and navigate the 3D environment with first-person controls. Under inspector interaction mode, users are able to **fly** around and stay **floating** at whichever 3D position they want, as shown in Fig 2. This mode is primarily used for debugging during development. It has the following key features:

- **First-Person Navigation:** Move using `W`, `A`, `S`, and `D` keys with a movement speed of `40.0` units/second.
- **Mouse Rotation:** Adjust camera yaw and pitch via mouse movement with a sensitivity of `0.002`.
- **Zooming:** Use the mouse wheel to move the camera forward or backward along its forward vector.

Overall, the `FirstPersonInspectorInteraction` class offers a framework for first-person navigation, enhancing both development and user interaction within the application.

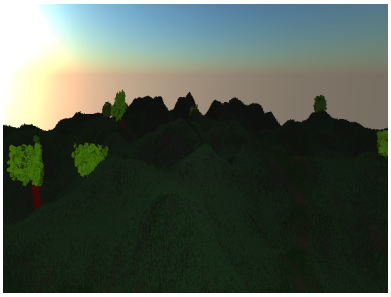

**Figure 2:** *Floating in Inspector Mode*

### 1.2 Player Interaction Mode

The `Player Interaction Mode` enables users to navigate the 3D environment with realistic movement mechanics, including **collision handling** and **jumping**. This mode is intended for end-users, providing an immersive experience within the application, as shown in Fig 3. This interaction mode has the following key features:

- **First-Person Navigation:** Move using `W`, `A`, `S`, and `D` keys with a movement speed of `10.0` units/second.
- **Mouse Rotation:** Adjust camera yaw and pitch via mouse movement with a sensitivity of `0.002`.
- **Zooming:** Use the mouse wheel to move the camera forward or backward along its forward vector, maintaining terrain height when not jumping.
- **Jumping Mechanics:** Press the `Space` key to initiate a jump, incorporating gravity for realistic vertical movement.
- **Collision Handling:** Prevents the camera from passing through terrain by updating the camera height to match the current terrain level.
- **Movement Detection:** Tracks active movement inputs and updates the application state accordingly.

Overall, the `FirstPersonPlayerInteraction` class provides a comprehensive framework for player navigation, balancing responsiveness with realistic movement mechanics to deliver an engaging user experience within the application.

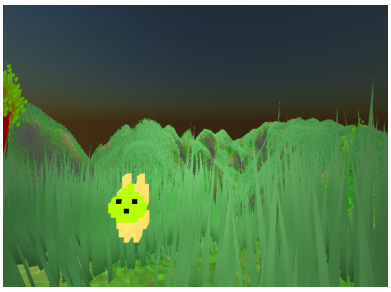

**Figure 3:** *Walking in Player Mode*

## 2 Scene Composition

### 2.1 Infinite Terrain

The terrain within the application is **procedurally** generated to create expansive and realistic 3D environments. This generation process leverages a combination of noise functions, tiled terrain management, and dynamic feature placement to ensure both visual fidelity and performance efficiency. Key components during terrain generation are listed below:

- **Tiled Terrain System:** The terrain is divided into smaller, manageable tiles using the `TerrainModel` class. This segmentation allows for efficient loading and unloading of terrain segments based on the player's position, optimizing memory usage and rendering performance.
- **Height Map Generation:** Each terrain tile utilizes a height map to define elevation variations. Height maps are generated using **multi-octave simplex noise**, providing natural and detailed terrain features. Parameters such as `maxHeight`, `frequency`, `octaves`, and `persistence` control the roughness and complexity of the terrain. One big advantage to use multi-octave noise is that it combines large-scale undulations with fine details, mimicking real-world terrain features. The texture of the terrain is also varied based on the approximated gradient of the terrain to produce rock-wall like features. This is done by mixing the texture map with a gray color based on the gradient value. These features are shown in Fig 4.
- **Noise Functions:** The `fast-simplex-noise` library's `makeNoise2D` function generates 2D simplex noise. We use the same seed for consistency across terrain tiles.
- **Dynamic Tile Management:** As the player navigates the environment, the system dynamically **loads** new terrain tiles within a specified range around the camera and **unloads** those

that move out of range. This approach maintains performance by only rendering necessary terrain segments.

- **Vegetation and Placement of Features:** Trees and grass are procedurally placed based on terrain height and slope. The system ensures natural distribution by avoiding overly steep areas and maintaining minimum distances between objects, enhancing the realism of the environment.
- **Collision Handling and Terrain Sampling:** Accurate collision detection is achieved by sampling terrain heights at specific coordinates using **bilinear interpolation** from the height maps. This ensures the player remains grounded and interacts naturally with the terrain surface.

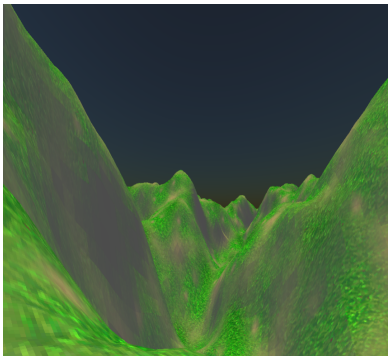

**Figure 4:** *Terrain generated with multi-octave noise*

## 2.2 Trees

The `TreeModel` class is designed to procedurally generate a tree-like structure as shown in Fig 5 based off of a **Lindenmayer system (L-system)**. An L-system is a string rewriting system we found that is used to model the growth patterns of plants and fractal structures. For our `TreeModel`, the L-system string defines rules for creating a branching, tree-like form. The model interprets the string to create instanced particles that represent trunk and leaf particles. The key features include the following:

- **L-system String and Rules:** The tree is generated from a symbolic string that defines the structure:
  - The initial string, `this.str`, starts as `"S"`, a seed symbol.
  - Multiple rewriting iterations expand the seed into a more complex string. This value is defined in `this.iterations`.
  - The different symbols represent either movement, branching, or leaf placement. These symbols are as follows:
    - `'F'` indicates moving forward and creating a branch particle.
    - `'B'` indicates the start of branching.
    - `'l'`, `'r'`, `'u'`, `'d'` (and their combinations like `'lu'`, `'ru'`, etc.) indicate rotations around different axes.
    - `'['` and `']'` indicate pushing or popping transformations onto or off of a stack in order to remember the position and orientation before branching.
    - `'L'` represents the placement of leaves.

  Our `expand` method applies the rewriting rules to transform the symbols into more complex patterns involving branching and leaves. This iterative process leads to a final expanded string that encodes a tree's structure.

- **Using a stack to keep track of branching**: The L-system's bracket symbols are essential as they handle the branching. When encountering a `'['` symbol, the current transformation for that particle (position, rotation) is pushed onto the stack and when encountering a `']'` symbol, the last transformation is popped off of the stack, going back to that previous position and orientation. This design allows the tree to branch out and then return to the previous point to start and other branches without losing track of where it came from.

- **Drawing the Branches and Leaves as Particles:** As the system interprets each `'F'` symbol, it moves forward and creates a new TrunkParticle at the new position. When it encounters a leaf symbol `'L'`, it creates smaller LeafParticles. The leaf generation step appears to add multiple individual leaf particles around the branch endpoint, but one LeafParticle is an instance of ten randomly generated spheres with varying green colors. Using instanced particles for the leaves like this allows for a more dense canopy, without compromising the frame rate of our scene, which was an issue we initially faced before we changed the particle system into an instanced particle system.

- **Transformations with Quaternions:** The code uses quaternions to handle rotations. Symbols like `'l'`, `'r'`, `'u'`, `'d'` indicate rotations about the previous particle's axes. `'l'` and `'r'` rotate around the Z-axis (left/right). `'u'` and `'d'` rotate around the X-axis (up/down). The combined symbols like `'lu'` or `'rd'` apply a combination of rotations in both axes to produce more complex directional change. An additional **"gnarliness"** parameter is added to introduce more natural variation, twisting the trunk and branches slightly so the tree isn't perfectly straight.

- **Particle Scaling and Color Interpolation:** The depth in the tree (how many branches deep) affects the particle size and color. The base branches are thicker and have darker bark-like color. As you move further up, increasing depth, the segments become thinner and the color transitions to a greener hue. This is achieved by interpolating between two colors and two sizes as depth increases.

- **Swaying Animation:** To animate the swaying of the trees, we used the vertex shader for the leaf particles and trunk particles. We updated the position in the shaders using a sinusoidal function concerning time.

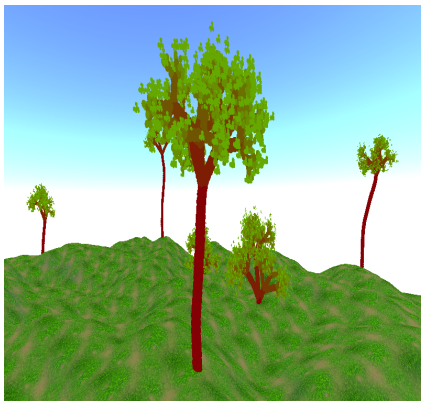

**Figure 5:** *Trees procedurally generated with an L-system using six iterations instead of just five like in our final scene.*

## 2.3 Grass

The grass rendering in this project is inspired by *The Legend of Zelda: Breath of the Wild*. A key challenge in rendering grass is achieving dense coverage over large spaces, which requires millions of grass blades. Our initial naive implementation using the example particle system allowed rendering only **10,000** blades at **25 FPS**, which was insufficient for large-scale coverage. Even after employing an instanced particle system, where each grass blade shared the same geometry, rendering beyond 20,000 instances caused significant performance drops due to GPU limitations. This is especially problematic in the context of infinite terrain generation where dense coverage over a large area is extremely important. After spending a lot of time on this problem we came up with the solution described below which allows us to render millions of blades of grass under the limitations of ThreeJS and Anigraph.

### Configuration

- **Instanced Geometry as Patches**: The key idea was to reduce the number of instanced particles (which we discovered was the main bottleneck) while increasing the number of blades. This was accomplished by grouping blades into patches, with each patch containing multiple blades. Random offsets within each patch created natural-looking grass distributions, allowing us to increase visual density with fewer than **1,000** instanced patches per tile.
- **Terrain Adaptation**: For non-flat terrain with variable heights, initializing every blade in each patch at the correct height was challenging. The shader dynamically queries the terrain height map to position each blade accurately during rendering, rather than relying on initialization.
- **Dynamic Grass Placement**: Grass was restricted from appearing on steep surfaces like rock walls during initialization by approximating the terrain gradient or normal. A discard condition in the shader removed grass blades that leaked onto unsuitable surfaces, ensuring a clean terrain.
- **Final Configuration**: The final setup for our scene uses **600** blades per patch, **500** patches per tile, and **9** tiles. This is a total of **600×500×9=2.7 million** blades rendered per frame at maximum fps without trees and unnoticeable performance loss with trees. We also tested the limits of the approach and were able to render 50 million blades at 25 fps with 1000 blades per patch, 2000 patches per tile, and 25 tiles. Fig 6 depicts terrain with 18 million blades of grass (1000 blades per patch, 2000 patches per tile, 9 tiles) that was rendered at 70 fps. The density is so great that the limiting factor in the visual is the height map resolution which leads to patching artifacts due to the same approximated gradient across blades.

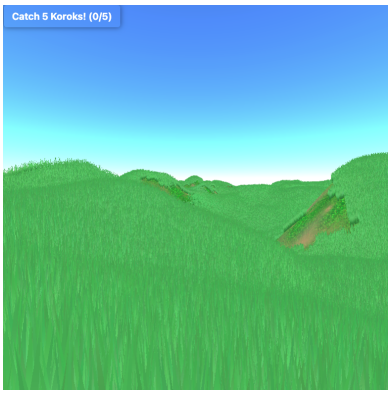


**Figure 6:** *18 million blades of grass rendered at 70 fps.*

### Shader Integration

- **Wind Simulation**: Grass sway was achieved using a sinusoidal function in the **vertex shader**, modulated by time. The sway offset scaled with the blade's local height (y-value), creating a realistic wind effect.
- **Height Mapping**: As mentioned above, we used a height map in the vertex shader to determine the height of each vertex. We also approximated the gradient to determine if vertex should be discarded if the hill was too steep.
- **Lighting**: Grass was rendered with **Blinn-Phong shading** to simulate dynamic lighting conditions. In order to account for the simple geometry of each blade, instead of clipping the specular and diffuse strength to 0.0 if the normal and corresponding vector were unaligned (dot product less than 0.0) we multiplied by **-1.0** to make it positive. This ensured that both night and day modes displayed appropriate lighting, accounting for the user's view light and sunlight. The Blinn-Phong shading results in some unnatural effects as all of the grass normals face a single direction in order to maintain a cartoonish effect. This causes shadow regions depending on the view light and sun light which we think adds to the scene even if it is a little unnatural. Randomly assigning the normals makes the lighting much more accurate, but leads to grass shadows and graininess. Fig 7 is an example of this phenomenon.

By combining these optimizations and shader techniques, the grass system achieved high visual fidelity and performance while covering large terrains effectively.

## 2.4 Fog

To implement the fog effect as shown in Fig 8, we went into the fragment shader for the trees, leaves, grass, and terrain and simply blended a white base color with the original color based on distance away from the camera.

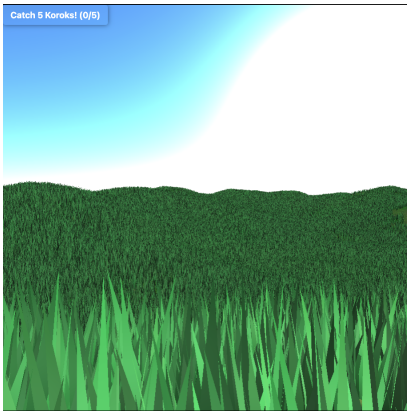


**Figure 7:** *Grainy Grass with Random Normals*

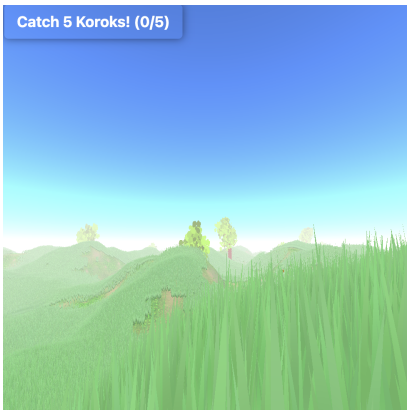


**Figure 8:** *Scene with Fog*

## 2.5 Koroks

Koroks shown in Fig 9 are collectible entities inspired by the Zelda series, enhancing gameplay by providing objectives and interactive elements. Korok models are imported from Blender. Key features include:

- **Generation:** Koroks are procedurally generated during terrain tile creation. Their positions are randomized within the tile, and their heights are adjusted based on the terrain to ensure they appear naturally placed.
- **Collision and Collection:** Players collect Koroks by entering their **bounding boxes**. Upon collection, the Korok is removed, a Korok sound effect from Zelda is played, and the player's score increases. Distant Koroks are removed to **optimize** performance, ensuring responsive and efficient gameplay. When the player reaches a score of 5, a **victory page** will be displayed.
- **Movement:** Koroks move autonomously with **velocity** vectors, updating their positions every frame. **Random direction** changes add unpredictability, keeping interactions dynamic and engaging.
- **Deforming Animation:** Koroks feature **periodic** scaling animations driven by sine wave functions, simulating lifelike behaviors like breathing or idle motions. To preserve their volume, the base and height scales are adjusted **inversely**, ensuring a natural and consistent appearance.

**Figure 9:** *A wandering Korok*

## 2.6 Sky / Lighting

The sky in our project is not a custom feature, as we utilized the shaders provided by the three.js Sky add-on. This implementation is based on the paper *"A Practical Analytic Model for Daylight"* and offers a realistic atmospheric effect. We viewed this approach as analogous to using pre-made skybox textures, as it provides high-quality results with minimal effort. However, integrating these shaders into our system did require some work, including creating a custom `SkyModel` class in AniGraph to support the shaders and manage uniform values.

Using the Sky Model allowed us to create dynamic lighting effects by positioning a "sun" point light that matched the sun's location in the Sky Model. This setup enabled variable lighting on the grass based on the sun's position, including pretty cool sunset effects.

To further enhance realism, we adjusted the Sky Model's parameters—such as "**turbidity**," "**Rayleigh scattering**," and "**mieCoefficient**"—as functions of the sun's height (**polar angle**). These adjustments ensured consistent and visually appealing results across different times of day.

In addition to tuning the Sky Model parameters for varying solar elevations, we also adjusted the strength of the view light based on the sun's polar angle. When the sun's intensity is low, such as during night mode, the view light becomes stronger, creating a **flashlight** effect. This addition enhances the immersion and provides a better visual experience in low-light conditions, making the night mode both functional and visually appealing. In night mode, we chose to make the trees and Koroks glow in the dark to make the scene more interesting, as shown in Fig 10.

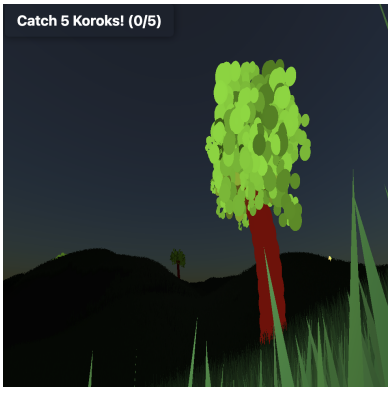


**Figure 10:** *Night Mode.*

## 3 References


- Website for Korok 3D model: Korok 3D Model on Sketchfab
- YouTube tutorial, which inspired our L-system tree generator: Drawing plants with L-Systems
- ThreeJS Sky Reference: ThreeJS Sky Model.
- Atmospheric Sky Model Paper: Paper on research gate.