Computer Graphics Project Report – **Bald Eagle Village**

Yang Xikun, **Lam Hei-wai**, **Chau Ping-yin**

Nov 2025

1 Project Overview

You can find the complete project source code in our **GitHub repository** (`https://github.com/FrankYang0610/bald-eagle-village`). You can find all the **math knowledge** required for this report from `https://github.com/FrankYang0610/bald-eagle-village/blob/main/appendix.pdf`.

You can **experience our animation** directly at: `https://frankyang0610.github.io/bald-eagle-village/`.

You can watch our **project presentation** at: `https://youtu.be/X-dsIBQdwqg`. You can watch our **animation** at: `https://youtu.be/EyjFPEEO4bk`.

1.1 Story

Our story takes place in a remote village in the mountains of Alaska. At dawn, with fog filling the air, two bald eagles fly side by side as they visit the village. As the fog clears, the two eagles notice each other and begin a chase above the village. The chase is thrilling! The eagles weave up and down, sometimes soaring upward, sometimes diving, sometimes flapping steadily, and sometimes making sudden directional changes that catch the other off guard. The sun rises, daylight arrives, and after tiring themselves out, the two bald eagles fly away side by side.

1.2 Computer Graphics Techniques Used

This project makes extensive use of mathematics and computer graphics concepts, including but not limited to various matrix transformations, model construction, hierarchical transformations, shader processing, water reflection, ripple effects, fog, key points and key frames, and most importantly: WebGL programming techniques.

1.3 Workload Declaration Table

Name	**Student ID**	**Workload**
Yang Xikun, Group Leader	/	Programming, presentation
Lam Hei-wai	/	Presentation
Chau Ping-yin	/	Presentation

2 Key Deliverables

2.1 Basic Requirements (20%)

Objective: Create a simplified scene, such as a solar system with the Sun, Earth, and Moon.

Content	Done?	Notes
Object Models: Create at least 3 distinct 3D objects with appropriate geometric detail (minimum 100 vertices per major object). Objects should have clearly defined shapes relevant to your story.	√	**Objects we designed ourselves:** • Terrain (**96,774** vertices), refer to Subsection 5.1 • Grass Lawn (**16,806** vertices per plane block), refer to Subsection 5.2 • Lamptree (**77,796** vertices) (A magical, glowing tree that can be considered a creative type of streetlight), refer to Subsection 5.3 **Objects created using external tools:** • Bald Eagles (**258,219** vertices), refer to Section 6 **External objects:** • Houses (**125,973** vertices, under CC BY 4.0 License), refer to Subsection 5.4. You can also check the **number of vertices** for each model in the console after the webpage is loaded.
Basic Scene Setup: Implement a ground plane or environmental base	√	The ground plane is the terrain, refer to Subsection 5.1
Basic Scene Setup: Add background elements (skybox, gradient background, or environmental backdrop)	√	We set up a sunrise scene with dynamic sky color changes, and also designed fog to create an overall environmental backdrop. Refer to Subsection 3.2 and Subsection 8.2.
Basic Scene Setup: Position objects thoughtfully to compose your scene	√	We placed the objects and created a small Alaskan village nestled in a valley. We also created two bald eagle models. Refer to Section 5 and Section 6.
Basic Animation: Translation (moving objects through space)	√	The movement trajectories of bald eagles chasing each other involve translation. Refer to Subsection 6.4.
Basic Animation: Rotation (spinning, orbiting, or turning objects)	√	The rotation of the bald eagle's wings during flapping flight involves the use of rotation. The movement trajectories of bald eagles chasing each other also involve rotation. Refer to Subsection 6.2.2 and Subsection 6.4.
Basic Animation: Scaling (growing, shrinking, or pulsating objects)	√	We scale the bald eagle's wings during its flapping flight to avoid hollows, refer to Subsection 6.2.2.
Lighting & Shading: Apply proper shading with at least 2 light sources (e.g., directional + point light). Use Phong shading model, etc.	√	**Implemented:** • **Ambient light:** adds base brightness everywhere. Refer to Subsection 8.1.3 • **Directional light:** sunlight for the whole scene (changes during sunrise). Refer to Subsection 8.1.1. • **Point lights [supporting multiple instances]:** small warm lamps on lamptrees, they flicker and fade with distance. Refer to Subsection 8.1.2.

Texturing: Apply textures or materials to objects with appropriate mapping	√	We assigned different texture materials to the terrain, plain, snow-capped mountain peak, lamptree, water (the underwater part of the terrain), and house. Refer to Section 7.
Camera Control: Implement basic camera system (orbital controls or fixed cinematic view)	√	We have two camera modes: a **fixed-position camera** and a **user-controlled free camera**. Refer to Section 4.

2.2 Middle Level Requirements (20%)

Objective: Create meaningful object interactions and enhanced visual quality.

Content	Done?	Notes
Object Interaction Animation: Implement animations where objects interact with each other: • Objects responding to other objects' movements • Synchronized movements showing relationships (e.g., gears turning together, planets orbiting) • Sequential animations creating cause-and-effect narratives	√	The two bald eagles fly in parallel, notice each other once the fog clears, and begin circling around each other in a chase, eventually flying away after they tire out. This storyline fully satisfies the requirement. Refer to Subsection 6.3 and Subsection 6.4.
Hierarchical Transformations: Use transformation hierarchies for complex object relationships (parent-child transformations enabling coordinated movements)	√	We used hierarchical transformations to manage the bald eagle's body and wings. Please refer to Subsection 6.1.4.
Enhanced Lighting: Implement multiple light sources (minimum 3) with different types: • Proper material properties (ambient, diffuse, specular coefficients) • Dynamic lighting that changes during the animation • Appropriate light colors and intensities supporting mood	√	**Implemented:** • **Ambient light:** adds base brightness everywhere. Refer to Subsection 8.1.3 • **Directional light:** sunlight for the whole scene (changes during sunrise). Refer to Subsection 8.1.1. • **Point lights [supporting multiple instances]:** small warm lamps on lamptrees, they flicker and fade with distance. Refer to Subsection 8.1.2.
Visual Coherence: Ensure all objects maintain consistent art style and contribute to the overall narrative.	√	All objects, textures, and styles are in the style of an Alaskan small village.

2.3 Advanced Level Requirements (20%)

Objective: Apply advanced techniques to create an immersive, polished experience.

Content	Done?	Notes
Advanced Visual Effects (any 2): • Normal mapping or bump mapping for surface detail • Fog or atmospheric effects • Reflection/refraction effects (mirrors, water, glass) • Custom shader effects (animated textures, dissolve effects, energy fields) • OR other related visual effects (students need to specify)	√	**Implemented:** • Fog, refer to Subsection 8.2 • Water reflections, refer to Subsection 8.3.1
Storytelling Excellence: Clear narrative arc with beginning, middle, and end	√	The narration and story explanation will be displayed at the bottom of the screen.
Storytelling Excellence: Visual storytelling through object behavior and interaction	√	The progression of the story can be clearly seen from the visuals.
Storytelling Excellence: Effective use of pacing, timing, and dramatic moments	√	The story of the bald eagle begins in the fog, gradually rises to a climax, and ends with the two flying off side by side. We think the pacing of our story is pretty good, and it has some dramatic moments.
Storytelling Excellence: Emotional impact or viewer engagement through visual composition	√	The bald eagles chasing each other and then flying away has quite a bit of emotional impact and viewer engagement. We hope people will like it.
Interactivity (any 2): • User-triggered animations or scene changes • Interactive camera controls (fly-through, object focus) • Dynamic scene modifications based on user input • Time control (pause, speed adjustment, rewind)	√	**Implemented:** • Interactive camera controls (Using `W`|`A`|`S`|`D` / `Up`|`Down`|`Left`|`Right`), refer to Section 4 • Time control (The user can pause, resume, fast-forward at 2× speed, return to the beginning of the story, or manually adjust the fog state), refer to Section 10.

2.4 Bonus Marks (10% additional)

3D Character Animation: • Implement an animated 3D character (humanoid, robot, animal, or creature) with articulated body parts • Use hierarchical skeletal structure for realistic movement • Implement at least 2 distinct character animations (walking, waving, dancing, etc.) • Character must contribute meaningfully to the story • Properly cite if importing character models from external tools	√	• We created a 3D bald eagle model with articulated body parts, including the body (body, eyes, etc.), the left wing, and the right wing. Refer to Section 6. • We used hierarchical transformations to manage the bald eagle's body and wings. Refer to Subsection 6.1.4. • The bald eagle has two flight postures: gliding and flapping flight, refer to Subsection 6.2. • The bald eagles are the subject of the story we want to tell, refer to Subsection 1.1. • We used AI to generate the entire bald eagle (since the art production is overly complex and not the focus of this project), then manually separated and refined it in Blender, exported it as an OBJ model, and used it in WebGL. Refer to Subsection 6.1.1.
Advanced Techniques Through Your Own Research: • Post-processing (bloom, motion blur, depth of field, color grading) • Particle systems (fire, smoke, rain, snow, sparkles, magic effects) • Shadow techniques (shadow mapping, shadow volumes, or approximations) • Procedural effects (waves, ripples, deformations)	√	We designed the wave ripples on the water surface. Refer to Subsection 8.3.

3 Scene

3.1 Coordinate Origin

We take $(0, 0, 0)$ as the origin and build the world outward in all directions. We consider the sea level to be the plane $y = 0$. Any area below $y = 0$ is regarded as being covered by water.

3.2 Sky and Sky Color Change

Please refer to `sunrise.js`. Specifically, we created a 30-second sunrise.

- The sky color gradually changes from the **the night sky** $(0.12, 0.14, 0.22)$ to the **dawn** $(0.22, 0.18, 0.28)$, and finally to the **light blue of the day** $(0.62, 0.78, 0.92)$.
- The sun rises from below the horizon, moving from $(-0.30, 0.50, -0.10)$ to a high position $(-0.30, -0.80, -0.10)$. The sunlight color changes as the sun rises: from a **dim gray-brown** $(0.10, 0.09, 0.08)$, to **gold at sunrise** $(1.00, 0.68, 0.28)$, and finally to a **warm daylight tone** $(1.00, 0.95, 0.85)$.
- The ambient light gradually brightens from a dark blue at night $(0.05, 0.06, 0.09)$ to a blue-gray daytime color $(0.35, 0.42, 0.50)$, simulating the lighting changes from late night to daytime.

4 Camera

Our program supports two camera modes: the **Fixed-Position Camera** and the **User-Controlled Camera**. After entering the scene, you can press `1` or `2` to switch between the modes: press `1` for activating the Fixed-Position Camera and press `2` for activating the User-Controlled Camera. Our program uses the User-Controlled Camera mode by default.

4.1 Fixed-Position Camera

In the Fixed-Position Camera mode, the camera remains fixed in one spot, facing the village. Users can see the bald eagles flying in, chasing each other, and then flying away.

4.2 User-Controlled Camera

In the User-Controlled Camera mode, you can freely control the camera's position using the `A`, `W`, `S`, `D` keys or the `Up`, `Down`, `Left`, and `Right` arrow keys, which allows you to move around and explore the village freely.

Note that pressing `W` moves you forward while descending, whereas pressing `S` moves you backward while ascending. The `Up` arrow key moves you upward in place, and the `Down` arrow key moves you downward in place.

For the camera control equations, please refer to Section 4.3.

4.3 Camera Control Equations

In the User-Controlled Camera mode, the camera moves according to the keyboard input provided by the user. The motion of the camera corresponds directly to the motion of the viewing direction. The camera state is defined by its position and orientation, and the following equations describe the complete camera and view movement model.

4.3.1 Camera Overview

Let the camera position be $\mathbf{p} = (x, y, z)$, the center (look-at) point be $\mathbf{c} = (c_x, c_y, c_z)$, and the world up-direction be $\mathbf{u} = (0, 1, 0)$. The forward viewing direction is defined as the normalized vector from the camera to the center:

$$\mathbf{f} = \text{normalize}(\mathbf{c} - \mathbf{p})$$

The right direction is obtained from the cross product of the forward vector and the world up vector:

$$\mathbf{r} = \text{normalize}(\mathbf{f} \times \mathbf{u})$$

These basis vectors build an orthonormal frame, and we use this frame to update how the camera moves.

4.3.2 Forward and Backward Motion.

If the user asks the camera to move forward by distance d, we change the camera's position and center as follows:

$$\mathbf{p}' = \mathbf{p} + d\mathbf{f}, \qquad \mathbf{c}' = \mathbf{c} + d\mathbf{f}$$

4.3.3 Vertical Motion.

Vertical motion (i.e., moving the camera up or down) changes its height but keeps the viewing geometry unchanged:

$$\mathbf{p}' = \mathbf{p} + (0, d, 0) \qquad \mathbf{c}' = \mathbf{c} + (0, d, 0)$$

To keep the camera from going underground or too high, we clamp its height value:

$$y' = \min\left[\max(y,\, y_{\min}),\, y_{\max}\right]$$

4.3.4 Rotation Around the Center.

To rotate the camera around the center point, we represent the offset vector $\mathbf{d} = \mathbf{p} - \mathbf{c}$ in spherical coordinates:

$$r = \|\mathbf{d}\|, \qquad \theta = \arctan\left(\frac{d_x}{d_z}\right), \qquad \phi = \arccos\left(\frac{d_y}{r}\right)$$

Applying a horizontal rotation $\Delta\theta$ and a vertical rotation $\Delta\phi$, the angles become

$$\theta' = \theta + \Delta\theta, \qquad \phi' = \min\left[\max(\phi + \Delta\phi,\, \phi_{\min}),\, \phi_{\max}\right]$$

where the clamping keeps the poles from causing gimbal lock.

The updated camera position is obtained by converting back to Cartesian coordinates:

$$\mathbf{p}' = \mathbf{c} + \begin{bmatrix} r\sin\phi'\sin\theta' \\ r\cos\phi' \\ r\sin\phi'\cos\theta' \end{bmatrix}$$

4.3.5 View Matrix Update.

Finally, every motion or rotation recomputes the view matrix using the standard look-at formulation:

$$\text{View} = \text{LookAt}(\mathbf{p}',\, \mathbf{c}',\, \mathbf{u})$$

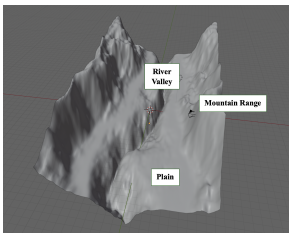
5 Static Objects

5.1 Terrain (96,774 vertices): Mountain, Plain and River Valley

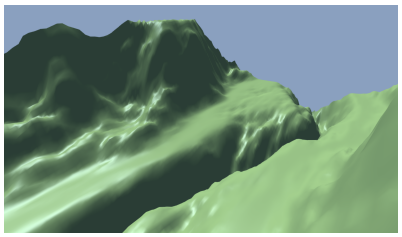
To create a natural-looking small village, we need to construct three types of terrain: **mountains**, **river valleys**, and **plains**. Houses and grass lawns can be placed on the plains.

Our terrain was created using **Blender**, its **sculpture tools**, and the **A.N.T. Landscape** add-on.

Specifically, for Operator Preset we choose **river**; for Subdivision we set $X \times Y = 128{\times}128$, and Mesh Size is $X \times Y = 2 \times 2$. We enabled **Smooth**. We set Noise Type to **Marble**, and Noise Basis to **Blender**. Then, we scaled X, Y and Z of the terrain model to 10, 20 and 50, respectively. We also used the **sculpture tool** to make the terrain more detailed and refined. In this way, we have successfully generated a terrain with a river valley in the middle, high mountains on both sides, and a small amount of plains between the valley and the mountains.



(a) Terrain Model



(b) Look toward the terrain with the directional light behind the camera

Figure 1: The terrain.

5.2 Grass Lawn (16,806 vertices)

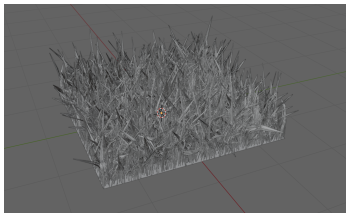
The structure of the lawn needs to include randomness, and the number of vertices must not be too high; otherwise the computer will struggle to render it and the performance will become very laggy.

We use Blender to create the grass lawn planes. To increase randomness, we use the **Particle System**. The **Hair** option in the Particle System can randomly generate a large amount of hair on a plane.

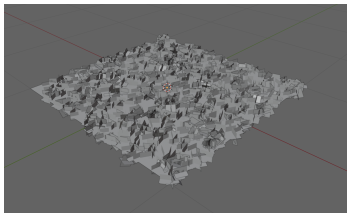
Based on our tests (please refer to `assets/grass-high-resolution/grass.blend` and `assets/grass-high-resolution/grass.obj`), if we generate a large amount of highly realistic grass by using **"rendered by path"** and assigning them with widths, the number of vertices will increase dramatically, making it impossible for WebGL to render.

Finally, we choose to use **two intersecting small planes** (like the "`X`" shape) as the grass object to be rendered (**"render by objects"**). In this case, the grass lawn may not look very realistic when looking up close. However, when viewed from a distance, with the help of textures and lighting, it still appears quite realistic.

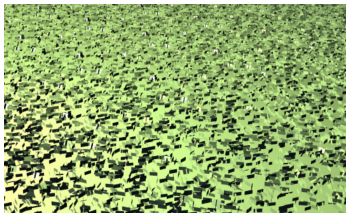
Finally, these lawn planes are assembled on the terrain, completing construction of the large grass lawn.



(a) High-resolution grass lawn using "render by path"



(b) Grass lawn using two intersecting small planes and "render by object"



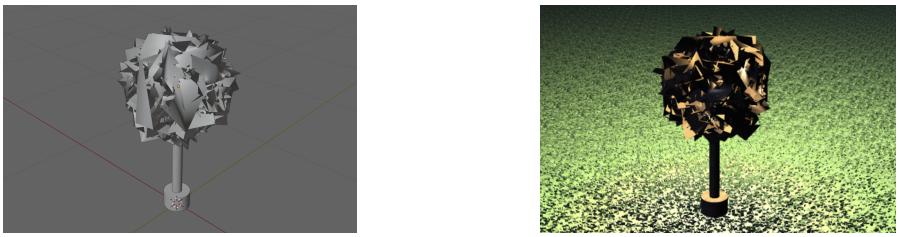
(c) Large grass lawn with green texture and under lighting

Figure 2: Comparison of different grass lawn rendering techniques.

5.3 Lamptrees (Street Lamps, 77,796 vertices)

A tree can be easily created using Blender's **Modifier** features. Specifically, the **Skin Modifier** can connect vertices into a skeleton-like surface that can then be sculpted, while the **Displace Modifier** can deform the surface to create an uneven texture. With the addition of a tall cylinder, the result resembles a tree.

In the end, **the tree was designed to function as a lamp**, hence the name "lamptree". The lighting process of the lamptrees is independent of the `.obj` models; it is handled within the WebGL program.



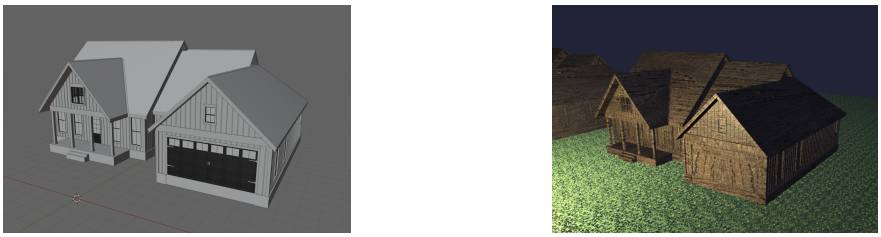
(a) Lamptree model

(b) Glowing lamptree on the grass lawn

Figure 3: The lamptree.

5.4 Houses [External Resource] (125,973 vertices)

Since the project is set in a village, we need some small houses. We chose an American-style house under a CC BY 4.0 license, which can be downloaded from here. This house also comes with many textures, which can therefore be used in other places as well. For detailed information about the textures, please refer to Section 7.



(a) House model

(b) House model with a wooden texture and front-facing lighting placed on a grass lawn

Figure 4: The house model.

6 Bald Eagle Model (258,219 vertices), Interactions and Hierarchical Transformations

6.1 Animal Object

6.1.1 Generate a 3D Bald Eagle Model

This part and only this part uses AI tools. Specifically, we used Meshy AI, a text-to-3D model to generate a 3D bald eagle model. **The refinement, compression, decimation, splitting, and animation of the AI-generated model do not involve any AI!**

We used a text prompt to request the Meshy AI and obtained the bald eagle model. The prompt is:

A bald eagle gliding with its wings spread, its talons slightly tucked in.

Finally, please refer to Figure 5 for the output result from Meshy AI.

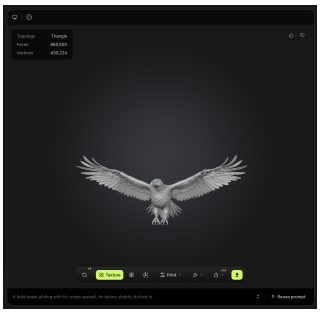
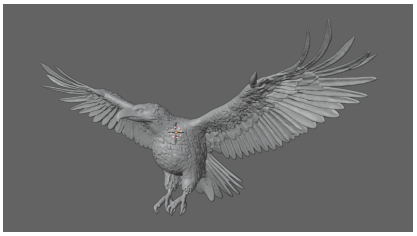


Figure 5: Output result of Meshy AI.

6.1.2 Model Refinement, Compression and Decimation

Because the bald eagle model generated by Meshy AI has 860,500 faces and 430,224 vertices, using it directly by importing it into WebGL and rendering animations would consume a large amount of resources and could cause stuttering or even crashes.

We imported the generated bald eagle model (`.obj` file) into Blender, added the **Decimate** modifier, and **set the ratio to 0.01**. In this way, we reduced the model to 1% of its original size without sacrificing its overall structure. The final bald eagle model and the untextured bald eagle model under lighting are shown in Fig 6.



(a) Bald eagle model

(b) Bald eagle model with a front-facing directional light

Figure 6: The bald eagle model.

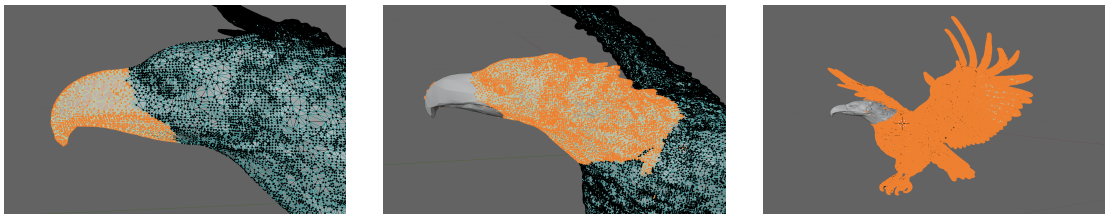
6.1.3 Bald Eagle Coloring

The bald eagle has a yellow beak, a white head, and a black body and wings, so we also need to paint our bald eagle model accordingly.

Our approach is to using **Circle Select** in Blender's **Edit Mode**, and then select the beak, the head, and the body (including the wings) in sequence. Note that the sequence order is important. For example, once the beak has been selected, brushing to select the head will automatically skip the beak area. Please refer to Figure 7.

To make the model more detailed, after selecting the head we also selected the eyes, pupils, and eye sockets. We will paint the eyes white, and the pupils and eye sockets black. This makes the bald eagle model much more refined. Please refer to Figure 8.

Finally, the finished bald eagle model appears in the WebGL rendering as shown in Figure 9.



(a) Bald eagle's selected beak (b) Bald eagle's selected head (c) Bald eagle's selected body

Figure 7: The bald eagle's beak, head, and body selected using the Circle Select tool.

(a) Bald eagle's selected eye pupil (b) Bald eagle's selected eye socket

Figure 8: Further select the eyes from the bald eagle's head and determine the positions of the eye sockets and pupils.

6.1.4 Animal Armature and Hierarchical Transformations

We split the bald eagle model into three articulated body parts: **the body, the left wing, and the right wing**.

The splitting process is similar to the selecting process described in Subsection 6.1.3. To ensure that both the left and right wings are fully selected, we used the **X-ray** functionality, which allows us to select vertices **through the surface**. Please refer to Figure 10.

To allow the eagle's body and its two wings to move in a coordinated manner, we apply **hierarchical transformations** to the bald eagle model. We use `parentM` as the parent (root/body) model matrix, applying translation, rotation around the model center (controlled by `useModelCenterPivot` and `model.center`), and scaling to the entire model. All child components inherit these parent transformations so that they move in a coordinated way.

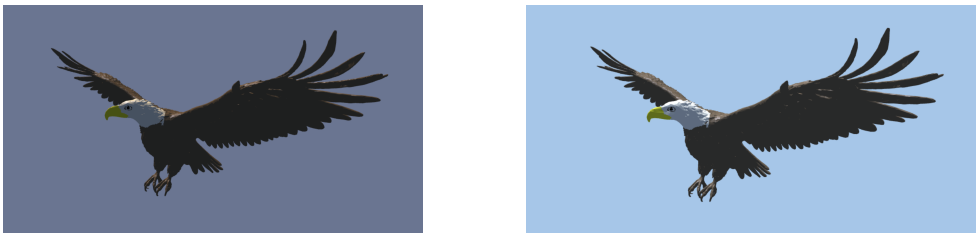
The fixed parts (body, beak, head, eyes, etc.) are drawn directly under `parentM` as `childFixedParts`, without any local transformation. The left and right wings each copy the parent matrix into `leftWingChildM` and `rightWingChildM`, respectively. The wings not only inherit the parent transform but also apply additional local flapping animation (rotation around the Z-axis) and an optional compression along the X-axis (to prevent gaps from appearing between the wings and the body, please refer to Subsection 6.2.2 for details).

You can clearly find the code for the hierarchical transformations in `baldEagle.js`.

6.2 Animal Animation

We implemented two flight postures for the bald eagle: **gliding** (with both wings spread, held still and level) and **flapping flight** (with both wings moving up and down symmetrically).

For the appearance of the two flight postures, see Figure 11. For the graphics and mathematical treatment of gliding and flapping flight, refer to Subsection 6.2.1 and Subsection 6.2.2, respectively.



(a) Dim light shining from behind the bald eagle (b) Bright light shining from behind the bald eagle

Figure 9: The WebGL rendering results of the bald eagle model under different lighting conditions.

(a) Bald eagle's selected right wing (b) Bald eagle's selected left wing

Figure 10: Select the right wing and left wing of the bald eagle model.

6.2.1 Gliding

Handling gliding is easy. We just need to use the model's default form, which already has its wings spread.

6.2.2 Flapping flight

To ensure that WebGL can correctly rotate the wings, we need to adjust the position of the bald eagle model. After the adjustment, the center of the wings (which is also the center of the entire model) is placed at the origin of its local coordinate system; the wings extend horizontally along both directions of the x-axis; and the head is aligned parallel to the y-axis. Consequently, the wings only need to rotate up and down around the z-axis.

In addition, because the model itself is **hollow**, a large **gap** appears between the wings and the body when the wings flap. To address this issue, we applied two strategies:

- (1) filling the hollow areas between the body and the wings and using **knife tools** and **sculpture tools** to pull the geometry outward, so that the wings remain closer to the body during flapping;
- (2) scaling the wings along the X-axis. In the code, we use the variable `xCompressRatio` $\in [0, 1]$ to represent the amount of X-axis compression applied to both wings. When the wings flap, their positions are scaled according to `xCompressRatio`, which naturally moves the wings closer to the center and thus closer to the body, making gaps less likely to appear.

Please refer to Figure 12 to see how we filled the hollows.

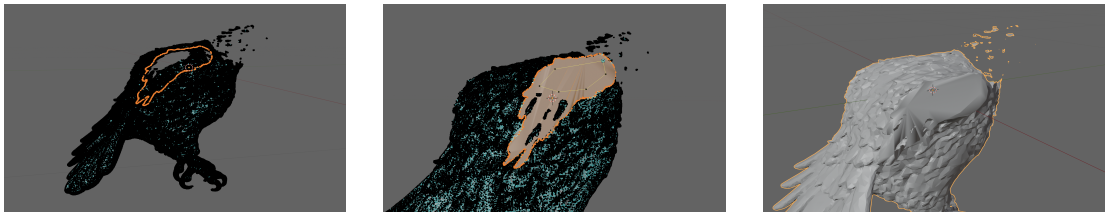
Finally, we introduce the equations of wing rotation. Both wings rotate about the local z-axis. The pivot of rotation is the local origin $(0, 0, 0)$.

Let the wing angle be

$$\theta(t) = \begin{cases} A\sin(2\pi f t), & \text{flapping flight} \\ 0, & \text{gliding} \end{cases}$$

(a) Gliding (b) Flapping flight

Figure 11: Two flight postures



(a) Use `Option` + Click to select the hollow boundary (b) Use `F` (New Edge from Vertices) to fill the hollow (c) Use the **knife tools** and sculpture tools to pull the geometry outward

Figure 12: Using Blender to fill in the hollows on the bald eagle's body

The left and right wings should rotate in opposite directions around the z-axis:

$$\theta_{\text{left}}(t) = +\theta(t) \qquad \theta_{\text{right}}(t) = -\theta(t)$$

So, the left-wing child matrix and the right-wing child matrix are updated by:

$$M_{\text{left}}(t) = M_{\text{parent}}\ \theta_{\text{left}}(t)\ S_x\big(c(t)\big), \qquad M_{\text{right}}(t) = M_{\text{parent}}\ \theta_{\text{right}}(t)\ S_x\big(c(t)\big)$$

where $c(t)$ is the x-axis compression factor applied to both wings:

$$c(t) = \begin{cases} \texttt{flap.xCompressRatio}, & \text{when flapping} \\ 1, & \text{otherwise} \end{cases}$$

and $S_x(c) = \text{diag}(c, 1, 1, 1)$ denotes a scale transform on the x-axis only.

6.3 Animals Interactions

There are two bald eagles in our scene. They encounter each other one after the other, circle and chase each other above the village, and then fly away together.

Therefore, the interactions between the two bald eagles satisfy multiple criteria at the same time: **objects responding to other objects' movements** (eagles chase each other), **synchronized movements** (eagles fly away side by side), and **sequential animations** (eagles begin to chase each other because they encounter one another).

For details on how the two bald eagles actually fly, please refer to Subsection 6.4.

6.4 Flying Paths

6.4.1 Program Implementations

For the bald eagle model, please refer to `baldEagle.js`. For the processing of the bald eagle's flight trajectory (including interpolation and related operations), please refer to `baldEagleAnimations.js`. For the bald eagle's flight path, please refer to `main.js` and Subsection 6.4.2.

Firstly, `baldEagle.js` is the factory for creating animatable bald eagles. It generates a bald eagle object with various properties including **position**, **rotation**, **scale**, **color**, and **body components** (beak, eyes, eye-socket, pupils, head, body, left wing, right wing). Its `render()` method computes **wing flapping**, **flight posture**, and **possible motion paths** (for example, movement along the Z-axis or a custom controller). It uses **matrix and hierarchical transformations** to assemble the local and global poses of the body and wings. Finally, its `renderOnly()` method draws the eagle's body and flapping wings, thus producing a flying animation (flapping flight or gliding) during rendering.

After creating the bald eagle object, `baldEagleAnimations.js` provides a **timeline system** that allows the entire animation to be broken into **sequential motion segments**. All we need to do is define what the bald eagle should do during each time segment (for example, move along a straight line, perform interpolation, fly in circles, or hover) and how long each segment is. The controller then automatically determines which segment the eagle should currently be in, where it should be positioned, and which direction it should face, based on the current animation time. This script also decides the eagle's pitch, wing-flap amplitude, or flight mode according to the configured posture rules. **In this way, the bald eagle can move continuously along a complex and smooth flight path, achieving a configurable animation system similar to an animation script.** A key frame describing the bald eagle's motion can be written in the following form: `{ position: [0, 30, 400], durationSeconds: 7, pose: { flightMode: 'gliding', rotationX: (2 * Math.PI / 180) } }` .

For the key points that the two bald eagle models should pass through, please refer to the `initSceneContent` function in `main.js` as well as Subsection 6.4.2.

6.4.2 Key Points of Two Eagles

In the Table 6 and Table 7, we list the key points that the two bald eagles pass through, the flight posture specified in the key frame that follows each key point, and the corresponding narrative description for that segment of the story.

Table 6: Key Points (Intervals) and Corresponding Flight Postures of Bald Eagle A

Step	**Time (s)**	**Pose**	**Story (from code)**
Fog Flying	0-7	Gliding, head up	Flies in fog. Keeps head up.
Outer Move	7-9	Flapping	Goes to the outside place.
Go Up	9-11	Flapping	Goes up.
Down and Up	11-13	Flapping	Goes down through the center. Goes up at the back.
Low Fly	13-15	Gliding	Flies low.
Middle Up	15-17	Flapping	Goes up to the middle.
Small Circle	17-19	Gliding	Makes a small circle. Flies slowly.
Second Pass	19-21	Flapping	Goes through again. Goes up to leave.
Keep Following	21-28	Mixed	Keeps following.
Fly Away	28-30	Gliding	Flies away.

Table 7: Key Points (Intervals) and Corresponding Flight Postures of Bald Eagle B

Step	Time (s)	Pose	Story (from code)
Fog Flying	0-7	Gliding, head up	Flies in fog.
Inner Move	7-9	Flapping	Goes to the inside place.
Go Up	9-11	Flapping	Goes up.
Fast Down	11-13	Flapping	Goes down fast.
Low Inside Fly	13-15	Gliding	Flies low inside.
Middle Up	15-17	Flapping	Goes up to the middle.
Inside Circle	17-19	Gliding	Makes a small inside circle. Flies slowly.
Second Pass	19-21	Flapping	Goes through again. Goes up.
Keep Following	21-28	Mixed	Keeps following.
Fly Away	28-30	Gliding	Flies away.

7 Textures

We introduced several types of materials for different objects. All texture files are stored under `assets/textures/`.

- **plain area and grass lawn** (modified from a photo taken by Yang Xikun in Yosemite Village, Yosemite National Park, CA, refer to `grass-or-plain.jpeg`),
- **snow-capped mountain peak** (modified from a photo taken by Yang Xikun in Tioga Pass, Yosemite National Park, CA, refer to `snow-mountain.jpeg`),
- **water** (modified from a photo taken by Yang Xikun in Lake Tahoe, CA/NV, refer to `water-or-cloud.jpeg`),
- **lamptree** (from a photo taken by Yang Xikun in Sequoia National Park, CA, refer to `tree.jpg`),
- **wooden house** (external resource from the House object, refer to `wood.jpg` and Subsection 5.4).

8 Lighting, Shading and Dynamic Effects

8.1 Blinn-Phong Lighting

The Blinn-Phong Reflection Model is a useful reflection model to approximate both diffuse and specular reflection on 3D surfaces. The following mathematical derivations can be found in any computer graphics textbook. However, for the completeness in this report, we include the full derivations here.

Let $\mathbf{x}$ be the surface point (world coordinate), $\mathbf{N} = \text{normalize}(\mathbf{n})$ be the normal vector, $\mathbf{V} = \text{normalize}(\mathbf{v}_{\text{cam}} - \mathbf{x})$ be the viewer's direction, and one or more light sources, the shading can be decomposed into three parts: **ambient, diffuse, and specular**.

8.1.1 Directional Lights

The light direction (pointing from the light toward the surface) is constant in directional lights. Let the light direction be $\mathbf{L}_d$. The diffuse term using Lambert's cosine law can be defined as:

$$D_d = \max(\mathbf{N} \cdot \mathbf{L}_d,\ 0)$$

Blinn introduced the **half-vector**, which provides a more stable **specular highlight** than the original Phong reflection model:

$$\mathbf{H}_d = \text{normalize}(\mathbf{L}_d + \mathbf{V})$$

In Blinn's model, the specular term is:

$$S_d = [\max(\mathbf{N} \cdot \mathbf{H}_d,\ 0)]^s$$

where s is the shininess coefficient that controls how bright the highlight is.

Let $\mathbf{C}_d$ be the directional light color and $\mathbf{C}_{\text{surf}}$ be the surface diffuse reflectance, then the Blinn–Phong directional lighting is

$$\mathbf{L}_{\text{dir}} = D_d\,(\mathbf{C}_d \odot \mathbf{C}_{\text{surf}}) \ +\ S_d\,\mathbf{C}_{\text{spec}}$$

where $\mathbf{C}_{\text{spec}}$ is the specular light color and $\odot$ is the Hadamard product (element-wise product).

8.1.2 Point Lights

Let a point light be located at $\mathbf{p}_i$ with color $\mathbf{C}_i$ and intensity I_i, the direction of the light is:

$$\mathbf{L}_i = \text{normalize}(\mathbf{p}_i - \mathbf{x}), \qquad d_i = \|\mathbf{p}_i - \mathbf{x}\|$$

We adopt a standard **attenuation model**, which assumes inverse attenuation with constant, linear, and quadratic terms:

$$A_i = \frac{I_i}{k_{c,i} + k_{l,i} d_i + k_{q,i} d_i^2}$$

We can use the same Blinn-Phong formula to get the diffuse and specular terms:

$$D_i = \max(\mathbf{N} \cdot \mathbf{L}_i,\ 0), \qquad \mathbf{H}_i = \text{normalize}(\mathbf{L}_i + \mathbf{V}), \qquad S_i = [\max(\mathbf{N} \cdot \mathbf{H}_i,\ 0)]^s$$

Finally, the total light from the point light is

$$\mathbf{L}_{\text{pt}} = \sum_{i=1}^{N} \{A_i\,[D_i\,(\mathbf{C}_i \odot \mathbf{C}_{\text{surf}}) \ +\ S_i\,(\mathbf{C}_{\text{spec}} \odot \mathbf{C}_i)]\}$$

8.1.3 Ambient Term

The ambient lighting approximates "ubiquitous" light in the scene:

$$\mathbf{L}_{\text{amb}} = \mathbf{C}_{\text{amb}} \odot \mathbf{C}_{\text{surf}}$$

8.1.4 Final Shaded Color

The final color is made by adding all the lights:

$$\mathbf{C}_{\text{final}} = \mathbf{L}_{\text{amb}} + \mathbf{L}_{\text{dir}} + \mathbf{L}_{\text{pt}}.$$

8.1.5 Lighting Result

In the previous sections, we have presented a large number of figures demonstrating the rendering results of different objects under various lighting conditions. Please refer to the Figure 1b (terrain with directional lighting), Figure 2c (grass lawn with lighting), Figure 3b (lighting lamptree), Figure 4b (house with facing lighting), Figure 9b (bald eagle lit from behind).

8.2 Fog

The fragment shader computes fog as an **exponential attenuation** that depends on **distance**, **height (y coordinate)**, and **animated noise**. Generally, fog thickens near the ground, varies in patches due to FBM noise, and drifts over time with wind. Please refer to Subsection 8.2.8 for the final rendering results of the fog.

Let $\mathbf{x}$ be the world coordinate and $\mathbf{v}$ be the view/camera position, we can compute:

8.2.1 Distance Term

$$d = \|\mathbf{v} - \mathbf{x}\|$$

8.2.2 Height Falloff

Let y be the fragment height, h_0 be the fog base height, then,

$$h_{\text{above}} = \max(y - h_0, 0), \qquad H = e^{-k_h h_{\text{above}}}$$

Note that k_h is defined in `uFogHeightFalloff` and H decreases as height increases.

8.2.3 Noise Term

Let horizontal position be $\mathbf{p} = (x, z)$. Wind-shifted coordinate can be defined as,

$$\mathbf{p}' = (\mathbf{p} + \mathbf{w}t)s_n$$

where $\mathbf{w}$ is defined in `uFogWind`, t is defined in `uTime`, and s_n is defined in `uFogNoiseScale`.
Then, apply a 2-layer fractal noise N and clamp it (please refer to `noise-generation.pdf` on our GitHub repo for more details):

$$n = 0.6 \cdot N(\mathbf{p}') + 0.3 \cdot N(2\mathbf{p}') \qquad n_{01} = \text{clamp}(n, 0, 1)$$

Define noise strength $s \in [0, 1]$ and get the noise factor:

$$\text{noiseFactor} = \max\left[1 + (2n_{01} - 1)s, 0\right]$$

where $\mathbf{i} = \lfloor \mathbf{p} \rfloor$, $\mathbf{u} = \mathbf{f}^2(3 - 2\mathbf{f})$, and $h(\cdot)$ is the hash pseudo-random function.

8.2.4 Effective Fog Density

Define base fog density as ρ_0, and the fog density ρ can be computed as:

$$\rho = \rho_0 \cdot H \cdot \text{noiseFactor}$$

8.2.5 Exponential Fog Factor

Define the exponentialfog factor as $e^{-\rho d}$ and clamp it to $[0, 1]$:

$$F = \text{clamp}(e^{-\rho d}, 0, 1)$$

Apparently $F = 1$ indicates no fog and $F = 0$ indicates a full fog.

8.2.6 Fog Color Blending

Let $\mathbf{C}_{\text{surf}}$ be the lit surface color, $\mathbf{C}_{\text{fog}}$ be the fog color, and α be the fog fade coefficient. Note that $\alpha = 0$ indicates a disabled fog system and $\alpha = 1$ indicates the full fog system. We can then define the first blend (fog based on distance/height/noise) and the global fog:

$$\mathbf{C}_{\text{fogged}} = (1 - F)\mathbf{C}_{\text{fog}} + F\mathbf{C}_{\text{surf}}, \qquad \mathbf{C}_{\text{final}} = (1 - \alpha)\mathbf{C}_{\text{surf}} + \alpha\mathbf{C}_{\text{fogged}}$$

8.2.7 Summary

Finally, the full fog formula is:

$$\mathbf{C}_{\text{final}} = (1 - \alpha)\mathbf{C}_{\text{surf}} + \alpha[(1 - F)\mathbf{C}_{\text{fog}} + F\mathbf{C}_{\text{surf}}], \qquad F = \exp\left\{-d\rho_0 e^{-k_h \cdot \max(y - h_0, 0)}[1 + 2n_{01} - 1]s\right\}$$

8.2.8 Final Result

The early-morning fog effect is shown in Figure 13. About the water surface effect, we will discuss it in Subsection 8.3.

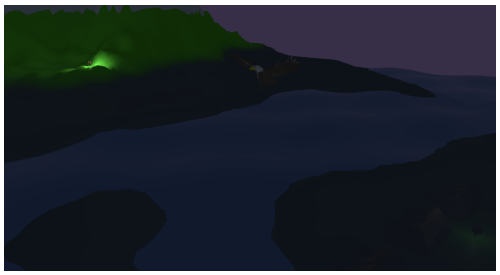


Figure 13: Early morning fog

8.3 Water

Our water includes:

- **Lighting:** Ambient light + a main directional light + Blinn–Phong highlights, with color and direction dynamically changing during the sunrise animation.
- **Reflections:** Mixing sky color into the water color using a Fresnel weight.
- **Ripples:** In the vertex shader we apply a time-varying $\sin(x) \cdot \cos(z)$ displacement and compute normals, with customized frequency/amplitude/speed.
- **Transparent blending:** We use alpha blending, set transparency to around 0.9, and temporarily disable back-face culling during rendering to ensure visibility.

Next, we will explain in detail how we achieve this mathematically. Please refer to Subsection 8.3.3 for the final rendering results of the water surface.

8.3.1 Water Reflection

The reflection effect on the water surface is achieved using the same Blinn–Phong lighting model, as described in Subsection 8.1. Additionally, to ensure that the water surface appears **sparkly and reflective from different viewing angles**, we also introduced an extra Fresnel factor.

- **Fresnel factor (angle effect):** We also introduce a Fresnel term to increase the sky color at grazing angles (i.e., angles where the light comes in almost sideways):

$$F = \left[1 - \max\left(\mathbf{N} \cdot \mathbf{V}, 0\right)\right]^3$$

- **Color blend:** To achieve more detailed water surface effects, we blend the water color with the sky color on the water surface. This step creates a more fancy reflection-like effect. The final color can be calculated as:

$$\mathbf{C}_{\text{final}} = (1 - 0.5F)\,\mathbf{C}_{\text{water}} + (0.5F)\,\mathbf{C}_{\text{sky}}$$

8.3.2 Water Ripple

- **Wave field:** A simple ripple can be written as

$$h(x, z, t) = A \sin(k_x x + \omega t + \phi_x) \cos(k_z z + \omega t + \phi_z)$$

  where A is amplitude, $k_x, k_z = 2\pi/\lambda$ are frequencies, and $\omega = 2\pi f$ is the angular frequency.

- **Normals of height field:** Surface normal can be calculated from:

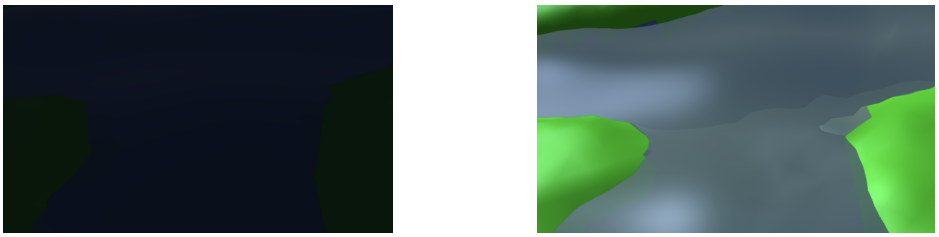
$$\mathbf{n} \propto \left(-\frac{\partial h}{\partial x},\ 1,\ -\frac{\partial h}{\partial z} \right)$$

  where:

$$\frac{\partial h}{\partial x} = A k_x \cos(k_x x + \omega t) \cos(k_z z + \omega t), \qquad \frac{\partial h}{\partial z} = -A k_z \sin(k_x x + \omega t) \sin(k_z z + \omega t)$$

8.3.3 Final Result

For the complete implementation of the mathematical derivations described above, please refer to `water.frag` and `water.vert`. The final rendering result is shown in Fig. 14.



(a) Water surface at night with almost no light (b) Water surface in morning under a directional light

Figure 14: Rendering effects of the water surface with and without lighting

9 Timers

To achieve animation, pause/play, and camera movement, we implemented multiple timers.

- The **animation timer** (using function `requestAnimationFrame`) is called once per frame and thus responsible for the entire website rendering loop.
- The **global timer** is controlled by pause and `timeScale`. The global timer is used for scene animations, sunrise, fog, and light flickering.
- The **camera timer** is updated using `unscaledDeltaSeconds`. This camera time is not affected by pause or `timeScale`, so the camera can still move when the animation is paused.

10 User Control

The user can select options in the lower-left corner of the webpage to play the animation, return to the beginning, resume playback, fast-forward, restore normal speed, and reload the fog. Users can still freely move the viewpoint when the animation is paused.

11 Conclusion

In this project, we used a variety of computer graphics techniques to achieve detailed visuals. From creating individual objects, to implementing the bald eagles' flight postures, to producing the final animation. Our animation includes the sunrise, water reflections and ripples, fog, the eagles' chase, and camera control. We developed each component step by step in separate modules, ultimately completing the final product.

12 References

12.1 Model Resources

- American House

12.2 App / Website Resources

- 2D Photo to 3D Model: Meshy AI

12.3 Literature References

- Vector (Wikipedia)
- Matrix (Wikipedia)
- Transformation matrix (Wikipedia)
- 3D Projection (Wikipedia)
- Gimbal lock (Wikipedia)
- Blender 4.0 Reference Manual
- Blender 5.0 (Latest) Reference Manual
- Brownian motion (Wikipedia)
- Wavefront `.obj` file (Wikipedia)
- Wavefront OBJ (Blender Reference Manual)
- Phong shading (Wikipedia)
- Phong reflection model (Wikipedia)
- Blinn-Phong reflection model (Wikipedia)
- Fractional Brownian motion (Wikipedia)
- Smoothstep (Wikipedia)
- Fresnel equations (Wikipedia)
- Schlick's approximation (Wikipedia)
- Angle of incidence (optics) (Wikipedia)