Script.js

```
class Tween {
    constructor(object, targetValues, duration, lerpFunction, callback) {
      this.object = object; // The object whose properties we're interpolating
      this.targetValues = targetValues; // The target values we're interpolating towards
      this.duration = duration; // The duration of the interpolation in milliseconds
      this.initialValues = {}; // Store initial values of properties
      this.elapsedTime = 0; // Track elapsed time
      this.active = false; // Whether the tween is active
      this.lerpFunction = lerpFunction || this.defaultLerp; // Use provided lerp function or default to
linear interpolation
      this.callback = callback || (() => {}); // Use provided callback or default to empty function
      this.initialize();
    }

    initialize() {
      // Store the initial values of the properties we're interpolating
      for (let key in this.targetValues) {
        if (this.object.hasOwnProperty(key)) {
          this.initialValues[key] = this.object[key];
        }
      }

      this.start();
    }

    start() {
      this.active = true;
      this.elapsedTime = 0;
      this.update();
    }

    update() {
      if (!this.active) return;

      // Calculate elapsed time
      this.elapsedTime += 16; // Roughly 60 frames per second
      const t = Math.min(this.elapsedTime / this.duration, 1); // Clamp t between 0 and 1

      // Interpolate each property
      for (let key in this.targetValues) {
        if (this.object.hasOwnProperty(key)) {
          this.object[key] = this.lerpFunction(
            this.initialValues[key],
            this.targetValues[key],
            t
```

```
        );
      }
    }

    // If we have reached the target values, stop the tween
    if (t === 1) {
      this.active = false;
      this.callback();
    } else {
      // Otherwise, request the next frame
      requestAnimationFrame(() => this.update());
    }
  }

  defaultLerp(start, end, t) {
    return (1 - t) * start + t * end;
  }
}

const linear = (start, end, t) => (1 - t) * start + t * end;

const easeInQuad = (start, end, t) => start + (end - start) * t * t;
const easeOutQuad = (start, end, t) => start - (end - start) * t * (t - 2);
const easeInOutQuad = (start, end, t) =>
  t < 0.5
    ? 2 * (end - start) * t * t + start
    : -1 * (end - start) * (--t * (t - 2) - 1) + start;

const easeInCubic = (start, end, t) => (end - start) * t * t * t + start;
const easeOutCubic = (start, end, t) =>
  (end - start) * ((t = t - 1) * t * t + 1) + start;
const easeInOutCubic = (start, end, t) =>
  t < 0.5
    ? 4 * (end - start) * t * t * t + start
    : (end - start) * ((2 * t - 2) * (2 * t - 2) * (2 * t - 2) + 1) + start;

const easeInQuart = (start, end, t) => (end - start) * t * t * t * t + start;
const easeOutQuart = (start, end, t) =>
  -(end - start) * ((t = t - 1) * t * t * t - 1) + start;
const easeInOutQuart = (start, end, t) =>
  t < 0.5
    ? 8 * (end - start) * t * t * t * t + start
    : -1 * (end - start) * ((t = t - 1) * t * t * t - 1) + start;

const easeInQuint = (start, end, t) =>
  (end - start) * t * t * t * t * t + start;
const easeOutQuint = (start, end, t) =>
  (end - start) * ((t = t - 1) * t * t * t * t + 1) + start;
```

```
  const easeInOutQuint = (start, end, t) =>
    t < 0.5
      ? 16 * (end - start) * t * t * t * t * t + start
      : (end - start) * (16 * (t -= 0.5) * t * t * t * t + 1) + start;

  const easeInExpo = (start, end, t) =>
    (end - start) * Math.pow(2, 10 * (t - 1)) + start;
  const easeOutExpo = (start, end, t) =>
    (end - start) * (-Math.pow(2, -10 * t) + 1) + start;
  const easeInOutExpo = (start, end, t) =>
    t < 0.5
      ? ((end - start) * Math.pow(2, 10 * (2 * t - 1))) / 2 + start
      : ((end - start) * (2 - Math.pow(2, -10 * (2 * t - 1)))) / 2 + start;

  const easeInCirc = (start, end, t) =>
    -(end - start) * (Math.sqrt(1 - t * t) - 1) + start;
  const easeOutCirc = (start, end, t) =>
    (end - start) * Math.sqrt(1 - (t = t - 1) * t) + start;
  const easeInOutCirc = (start, end, t) =>
    t < 0.5
      ? (-(end - start) / 2) * (Math.sqrt(1 - 4 * t * t) - 1) + start
      : ((end - start) / 2) * (Math.sqrt(1 - (2 * t - 2) * (2 * t - 2)) + 1) +
        start;

  // Example usage:
  /*
  let obj = { x: 0, y: 0 };
  let customLerpFunction = (start, end, t) => (1 - t) * start + t * end; // A custom lerp function
  let tween = new Tween(obj, { x: 100, y: 200 }, 2000, customLerpFunction); // 2 seconds
duration
  tween.start();
  */

  const poster = document.getElementById("poster");

  // Get the canvas element by ID
  const canvas = document.getElementById("canvas");

  const videos = [
    "https://alieninterfaces.com/static/pages/14-monster/assets/videos/header1.mp4",
    "https://alieninterfaces.com/static/pages/14-monster/assets/videos/header2.mp4"
  ];
  let currentIndex = 0; // Index of currently playing video
  let videoElements = []; // Array of video elements

  const initialValues = {
    exposure: 0.0,
    contrast: 1.0,
```

```
  brightness: 0.0,
  distortion: 1.0
};

const targetValues = {
  exposure: 1.0,
  contrast: 3.0,
  brightness: 1.9,
  distortion: 3
};

const values = Object.assign({}, initialValues);

const switchVideo = () => {
  currentIndex = (currentIndex + 1) % videos.length; // Cycle through the list of videos
};

function createVideoElement(src) {
  return new Promise((resolve, reject) => {
    // Create a video element
    /*
    const video = document.createElement("video");

    // Set video attributes
    video.src = src; // source URL of the video
    video.autoplay = true; // make video autoplay when it's loaded
    video.loop = true; // make video loop when it ends
    video.muted = true; // mute the video
    video.playsInline = true; // to allow the video to play inline on iOS devices.
    video.crossorigin="anonymous"
    */
    const video = document.querySelector(`.hide-vid[src="${src}"]`);
    console.log("vid", video);
    video.setAttribute("crossorigin", "anonymous");

    // Event listener for successful loading of video
    video.addEventListener("canplaythrough", () => {
      video.play();

      setTimeout(() => {
        poster.style.opacity = "0";
      }, 200);
      resolve(video);
    });

    // Event listener for errors while loading video
    video.addEventListener("error", () => {
      reject(new Error(`Failed to load video from source: ${src}`));
```

```
    });

    // Load the video
    video.load();
  });
}

// Example of using the createVideoElement function to asynchronously load multiple video
files
async function loadVideos(videoSources) {
  try {
    const videoElements = await Promise.all(
      videoSources.map((src) => createVideoElement(src))
    );
    // Do something with the loaded video elements, like appending them to the DOM
    //videoElements.forEach((video) => document.body.appendChild(video));
    return videoElements;
  } catch (error) {
    console.error("Error loading videos:", error);
  }
}

function initializeWebGLCanvas() {
  // Attempt to get the WebGL rendering context
  let gl =
    canvas.getContext("webgl") || canvas.getContext("experimental-webgl");

  if (!gl) {
    console.error(
      "Unable to initialize WebGL. Your browser may not support it."
    );
    return null;
  }

  // Set clear color to black, fully opaque
  gl.clearColor(0.0, 0.0, 0.0, 1.0);

  // Clear the color buffer with specified clear color
  gl.clear(gl.COLOR_BUFFER_BIT);

  return gl;
}

function drawVideoOnCanvas(gl, video) {
  // Initialize shaders
  const vertexShaderSource = `
    attribute vec2 position;
    varying vec2 vTexCoord;
```

```
    void main() {
        vTexCoord = vec2(position.x * 0.5 + 0.5, 1.0 - (position.y * 0.5 + 0.5));
        gl_Position = vec4(position, 0.0, 1.0);
    }
    `;

  const fragmentShaderSource = `
precision mediump float;
varying vec2 vTexCoord;
uniform sampler2D uSampler;
uniform float uDistortion; // uniform variable for spherical distortion
uniform float uExposure;   // uniform variable for exposure
uniform float uContrast;   // uniform variable for contrast
uniform float uBrightness; // uniform variable for brightness

void main() {
    vec2 center = vec2(0.5, 0.5);
    vec2 coord = vTexCoord - center; // translate to center
    float dist = length(coord);

    // apply spherical warp and zoom distortion
    //vec2 newCoord = coord / (1.0 + uDistortion * dist * dist) + vec2(0.5);
    vec2 newCoord = center + normalize(coord) * pow(dist, uDistortion);

    vec4 color = texture2D(uSampler, newCoord);

        // adjust exposure, contrast, and brightness
    color.rgb = (color.rgb - 0.5) * uContrast + 0.5; // contrast
    color.rgb += uBrightness; // brightness
    color.rgb = color.rgb * pow(2.0, uExposure); // exposure

    gl_FragColor = color;

}

    `;
  const vertexShader = gl.createShader(gl.VERTEX_SHADER);
  gl.shaderSource(vertexShader, vertexShaderSource);
  gl.compileShader(vertexShader);

  const fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
  gl.shaderSource(fragmentShader, fragmentShaderSource);
  gl.compileShader(fragmentShader);

  const shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);
```

```
gl.useProgram(shaderProgram);

// Initialize buffer
const vertices = new Float32Array([
  -1.0,
  -1.0,
  1.0,
  -1.0,
  -1.0,
  1.0,
  1.0,
  1.0
]);
const vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

const positionAttribLocation = gl.getAttribLocation(
  shaderProgram,
  "position"
);
gl.vertexAttribPointer(positionAttribLocation, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(positionAttribLocation);

const uDistortionLocation = gl.getUniformLocation(
  shaderProgram,
  "uDistortion"
);
const uExposureLocation = gl.getUniformLocation(shaderProgram, "uExposure");
const uContrastLocation = gl.getUniformLocation(shaderProgram, "uContrast");
const uBrightnessLocation = gl.getUniformLocation(
  shaderProgram,
  "uBrightness"
);

// Set initial value for distortion
gl.uniform1f(uDistortionLocation, values.distortion);
gl.uniform1f(uExposureLocation, values.exposure);
gl.uniform1f(uContrastLocation, values.contrast);
gl.uniform1f(uBrightnessLocation, values.brightness);

// Initialize texture
const texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
```

```
  const adjustSize = () => {
    const canvasAspectRatio = gl.canvas.width / gl.canvas.height;
    const videoAspectRatio = video.videoWidth / video.videoHeight;
    let renderWidth, renderHeight;

    if (canvasAspectRatio > videoAspectRatio) {
      renderWidth = gl.canvas.height * videoAspectRatio;
      renderHeight = gl.canvas.height;
    } else {
      renderWidth = gl.canvas.width;
      renderHeight = gl.canvas.width / videoAspectRatio;
    }

    const xOffset = (gl.canvas.width - renderWidth) / 2;
    const yOffset = (gl.canvas.height - renderHeight) / 2;

    gl.viewport(xOffset, yOffset, renderWidth, renderHeight);
  };

  // Animation loop
  const animate = () => {
    adjustSize();

    const currentVideo = videoElements[currentIndex];

    if (currentVideo.readyState >= video.HAVE_CURRENT_DATA) {
      gl.bindTexture(gl.TEXTURE_2D, texture);
      gl.texImage2D(
        gl.TEXTURE_2D,
        0,
        gl.RGBA,
        gl.RGBA,
        gl.UNSIGNED_BYTE,
        currentVideo
      );
    }
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.uniform1f(uDistortionLocation, values.distortion);
    gl.uniform1f(uExposureLocation, values.exposure);
    gl.uniform1f(uContrastLocation, values.contrast);
    gl.uniform1f(uBrightnessLocation, values.brightness);

    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
    requestAnimationFrame(animate);
  };
  animate();
}
```

```
const gl = initializeWebGLCanvas();

(async () => {
  videoElements = await loadVideos(videos);
  canvas.width = videoElements[0].videoWidth;
  canvas.height = videoElements[0].videoHeight;
  drawVideoOnCanvas(gl, videoElements[0]);

  setTimeout(() => {
    transitionOut();
  }, 1000);
})();

window.addEventListener("click", () => transitionOut());

const body = document.querySelector("body");
const sectionA = document.querySelector(".sectionA");
const sectionB = document.querySelector(".sectionB");

sectionB.style.display = "none";

const transitionOut = () => {
  sectionA.style.opacity = 0;
  sectionB.style.opacity = 0;

  new Tween(values, targetValues, 1000, easeInQuad, () => transitionIn());
};

const transitionIn = () => {
  switchVideo();

  body.classList.toggle("dark");

  let tween = new Tween(values, initialValues, 1000, easeOutQuad, () => {
    if (currentIndex === 1) {
      sectionA.style.display = "none";
      sectionB.style.display = "block";
      sectionB.style.opacity = 0;
      setTimeout(() => {
        sectionB.style.opacity = 1;
      }, 100);
    } else if (currentIndex === 0) {
      sectionA.style.display = "block";
      sectionB.style.display = "none";
      sectionA.style.opacity = 0;

      setTimeout(() => {
```

```
        sectionA.style.opacity = 1;
      }, 100);
    }
  });
};

const videoButtons = document.querySelectorAll(".video-btn");
videoButtons.forEach((videoButton) => {
  videoButton.addEventListener("mouseover", () => {
    videoButton.play();
  });
  videoButton.addEventListener("mouseout", () => {
    videoButton.pause();
  });
});
```