Міністерство освіти і науки України

Національний технічний університет України «КПІ ім. Ігоря Сікорського»

Кафедра цифрових технологій в енергетиці

Розрахункова графічна робота

з дисципліни «Методи синтезу віртуальної реальності»

ВАРІАНТ №1

**Виконав:**

студент 5-го курсу

групи ТР-31мп

Бородавченко О. О.

**Перевірив:**

Демчишин А.А.

Київ – 2023

**Завдання**

1. Повторно використати код із практичного завдання №2;
2. Ті, хто має сертифікат курсу навчання дронів FPV: реалізувати обертання джерела звуку навколо геометричного центру ділянки поверхні протягом певного часу (цього разу поверхня залишається нерухомою, а джерело звуку рухається). Відтворити улюблену пісню у форматі mp3/ogg, маючи просторове розташування джерела звуку, кероване користувачем;
3. Візуалізувати положення джерела звуку за допомогою сфери;
4. Додати звуковий фільтр (використати інтерфейс BiquadFilterNode) для кожного варіанту. Додати елемент прапорця, який вмикає або вимикає фільтр. Встановити параметри фільтра на свій смак.

**Варіант**

1 - Фільтр низьких частот

**Теорія**

WebGL - це відкритий стандарт, який дозволяє розробникам веб-додатків створювати та відтворювати 3D-графіку в веб-браузері без використання плагінів. WebGL базується на OpenGL ES, стандарті для мобільної та вбудованої графіки.

WebGL працює за допомогою графічного процесора (GPU) комп'ютера. GPU - це спеціалізований мікропроцесор, який призначений для обробки графічних даних. WebGL дозволяє розробникам передавати дані GPU за допомогою JavaScript.

Процес рендерингу 3D-графіки в WebGL можна розділити на кілька етапів:

1. Введення геометрії. На цьому етапі розробник передає GPU дані про 3D-об'єкти, які потрібно відобразити. Ці дані можуть включати координати вершин, нормалі, текстурні координати та інші параметри.
2. Рендеринг вершин. На цьому етапі GPU перетворює дані про вершини в 2D-координати пікселів. Цей процес називається вершинним шейдингом.
3. Рендеринг пікселів. На цьому етапі GPU визначає, як забарвити кожен піксель. Цей процес називається піксельним шейдингом.

Web Audio API — це потужний і гнучкий інтерфейс для роботи зі звуком у веб-додатках. Він дозволяє розробникам створювати, обробляти і контролювати аудіо за допомогою JavaScript. API надає засоби для відтворення, мікшування, обробки та аналізу звуку, що робить його ідеальним для широкого спектру застосувань: від простого відтворення аудіофайлів до складних звукових ефектів у іграх і музичних додатках.

Основні концепції Web Audio API:

* Аудіо Контекст (AudioContext): Основний об'єкт, який представляє аудіо-середовище. Він використовується для створення і керування аудіо вузлами
* Аудіо Вузли (AudioNodes): Основні будівельні блоки для обробки звуку. Вузли можуть бути джерелами звуку (джерела), обробниками (фільтри), і кінцевими точками (пунктами виводу). Вузли можуть бути з'єднані між собою для створення звукових ланцюгів. Основні типи аудіо вузлів: OscillatorNode: генерує простий звуковий сигнал. GainNode: контролює гучність звуку. BiquadFilterNode: застосовує різні фільтри до звукового сигналу (наприклад, low-pass, high-pass).
* AudioBufferSourceNode: відтворює звук з буфера. З'єднання аудіо вузлів: Вузли з'єднуються між собою за допомогою методу connect(). Це дозволяє створювати ланцюги обробки звуку.

**Деталі виконання**

Функція для завантаження аудіо з мережі інтернет за посиланням.

async function createAudio() {

  audioContext = new window.AudioContext();

  audioSource = audioContext.createBufferSource();

  getFilter();

  getAudioPanner();

  try {

    const response = await fetch('https://raw.githubusercontent.com/Mag1sterY0da/WebGL/CGW/sound.mp3');

    const audioData = await response.arrayBuffer();

    const buffer = await audioContext.decodeAudioData(audioData);

    audioSource.buffer = buffer;

    if (music.filter) {

      audioSource.connect(audioFilter);

      audioFilter.connect(audioPanner);

    } else {

      audioSource.connect(audioPanner);

    }

    audioPanner.connect(audioContext.destination);

    audioSource.loop = true;

  } catch (error) {

    console.error('Error loading audio:', error);

  }

}

Створюємо фільтр згідно варіанту

function getFilter() {

  audioFilter = audioContext.createBiquadFilter();

  audioFilter.type = 'lowpass';

  audioFilter.frequency.value = 1000;

  audioFilter.Q.value = 1;

}

Створюємо Panner. Він використовується для керування розміщенням звуку в просторі

function getAudioPanner() {

  audioPanner = audioContext.createPanner();

  audioPanner.refDistance = 1;

  audioPanner.maxDistance = 1000;

  audioPanner.rolloffFactor = 1;

  audioPanner.coneInnerAngle = 360;

  audioPanner.coneOuterAngle = 0;

  audioPanner.coneOuterGain = 0;

  audioPanner.panningModel = 'HRTF';

  audioPanner.distanceModel = 'inverse';

  audioPanner.setPosition(audioPosition.x, audioPosition.y, audioPosition.z);

  audioPanner.setOrientation(0, 0, 0);

}

Створюємо буфер для зберігання координат сфери

  this.SBufferData = function (surfData) {

    gl.bindBuffer(gl.ARRAY\_BUFFER, this.iVertexBuffer);

    gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(surfData), gl.STREAM\_DRAW);

    this.sphereVerticesLength = surfData.length / 3;

  };

Додавання відтворення музики та фільтру по натисканні на кнопку Play та Filter.

  document.getElementById('filter').addEventListener('change', function (event) {

    if (!audioContext) return;

    const value = event.target.checked;

    audioSource.disconnect();

    audioPanner.disconnect();

    if (value) {

      audioSource.connect(audioFilter);

      audioFilter.connect(audioPanner);

      audioFilter.connect(audioContext.destination);

    } else {

      audioSource.connect(audioPanner);

      audioPanner.connect(audioContext.destination);

    }

    audioPanner.setPosition(audioPosition.x, audioPosition.y, audioPosition.z);

    audioPanner.setOrientation(0, 0, 0);

  });

  document.getElementById('playBtn').addEventListener('click', playMusic);

Зміна положення сфери та звуку:

function animateSphere() {

  let step = 0.05;

  let xAnimation = true;

  setInterval(() => {

    if (xAnimation) {

      sphereRotation.x += step;

      if (sphereRotation.x > 4 || sphereRotation.x < -4) {

        step = -step;

      }

      audioPosition.x = sphereRotation.x;

    } else {

      sphereRotation.y += step;

      if (sphereRotation.y > 2 || sphereRotation.y < -2) {

        step = -step;

      }

      audioPosition.y = sphereRotation.y;

    }

    audioPanner.setPosition(audioPosition.x, audioPosition.y, audioPosition.z);

    sphere.SBufferData(CreateSphereData());

    draw();

    if (xAnimation && Date.now() - startTime >= 30000) {

      xAnimation = false;

    }

  }, 100);

  let startTime = Date.now();

}

**Використання програми**

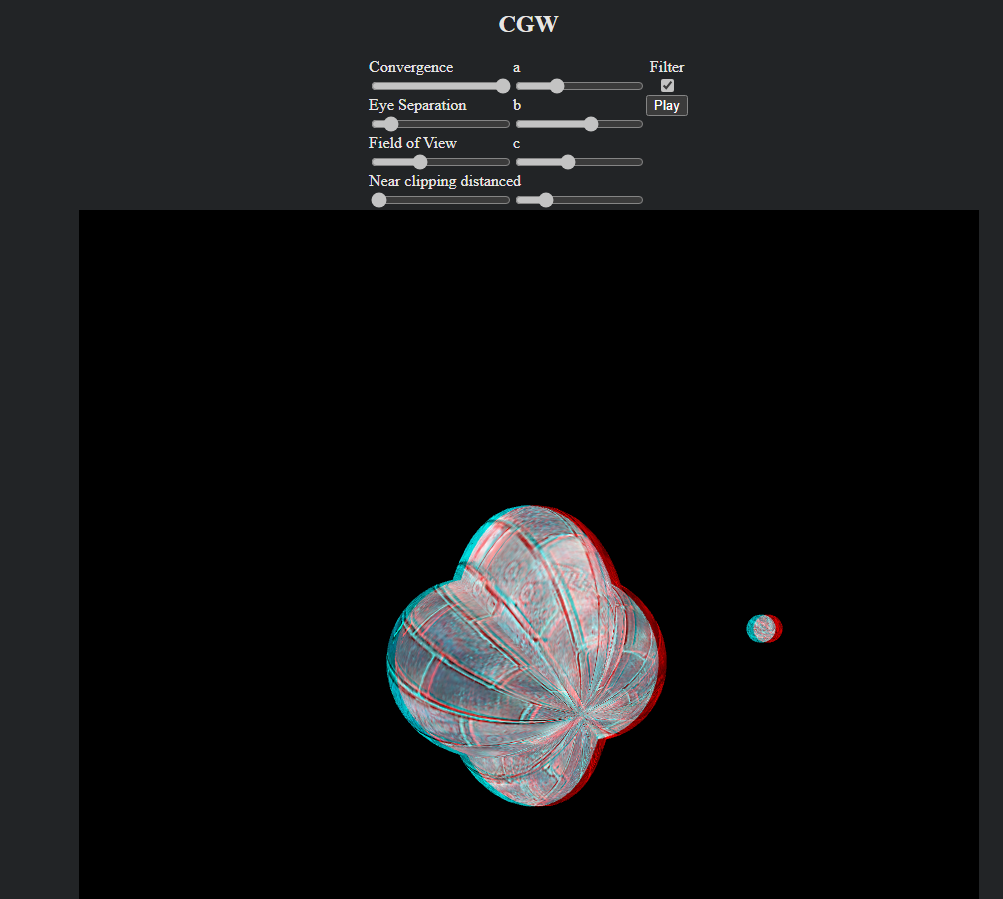
На рисунку 1 зображена фігура в правому положенні, відповідно звук відтворюється в правому навушнику:  


Рисунок 1 - Сфера в правому положенні

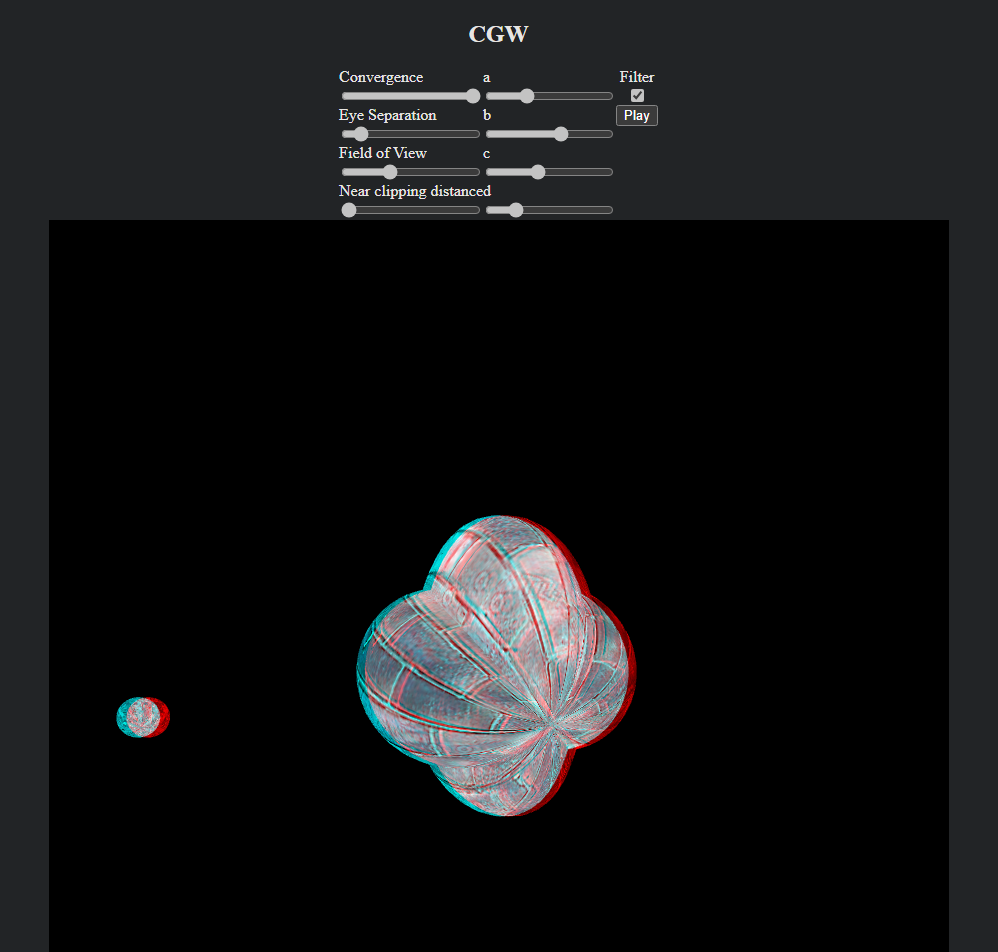
На рисунку 2 зображена сфера в лівому положенні, відповідно звук відтворюється в лівому навушнику:

Рисунок 2 Сфера в лівому положенні

За допомогою перемикача вимикаємо фільтр

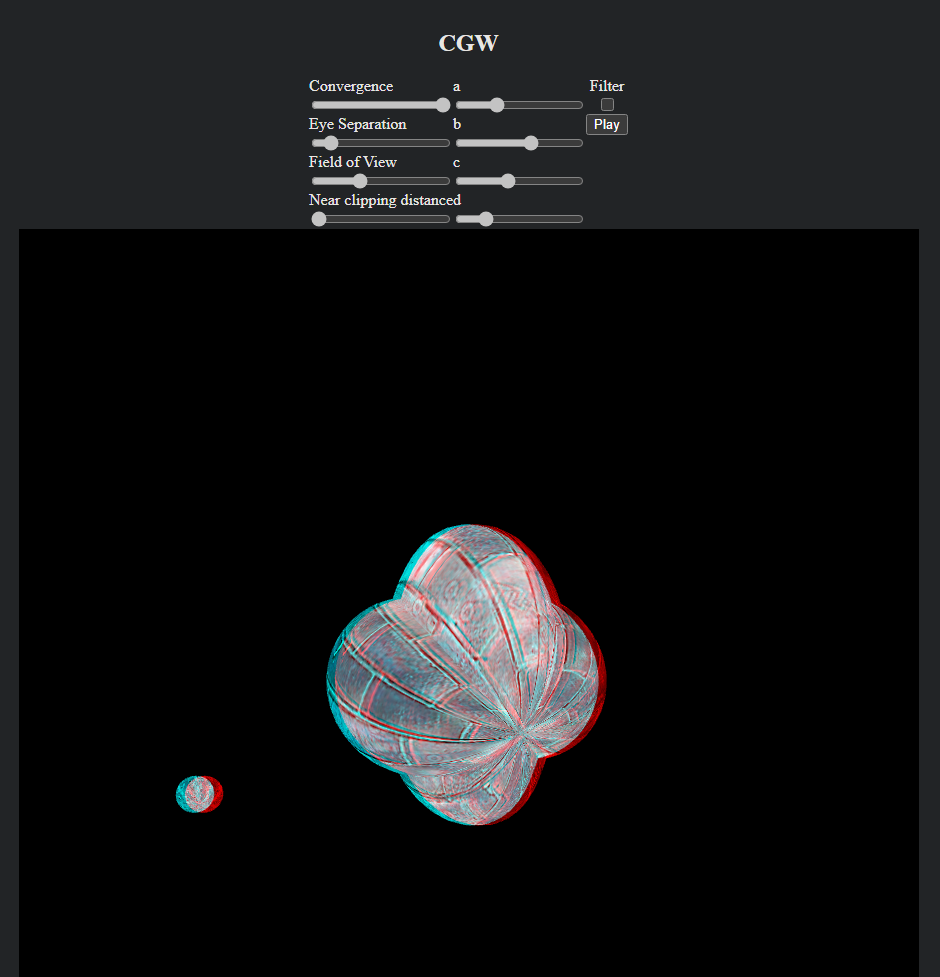


Рисунок 3 - Вимкнений фільтр

**Код програми**

'use strict';

let gl; // The webgl context.

let surface; // A surface model

let shProgram; // A shader program

let spaceball; // A SimpleRotator object that lets the user rotate the view by mouse.

let camera; // A StereoCamera object that manages the stereo camera parameters.

let webcam; // A Webcam object that obtains the camera feed.

let textureWebcam; // A texture object that holds the webcam feed.

let texture; // A texture object that holds the texture image.

let surfaceWebcam; // A surface model that displays the webcam feed in the background.

let track; // A MediaStreamTrack object that holds the camera feed.

let sphere;

let sphereRotation;

let audioContext;

let audioSource;

let audioPanner;

let audioPosition;

let audioFilter;

let music = { filter: true };

function deg2rad(angle) {

  return (angle \* Math.PI) / 180;

}

function CreateWebcamTexture() {

  textureWebcam = gl.createTexture();

  gl.bindTexture(gl.TEXTURE\_2D, textureWebcam);

  gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MIN\_FILTER, gl.LINEAR);

  gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MAG\_FILTER, gl.LINEAR);

  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);

}

function CreateCamera() {

  webcam = document.createElement('video');

  webcam.setAttribute('autoplay', true);

  navigator.getUserMedia(

    { video: true, audio: false },

    function (stream) {

      webcam.srcObject = stream;

      track = stream.getTracks()[0];

    },

    function (e) {

      console.error('Rejected!', e);

    }

  );

}

// Constructor

class StereoCamera {

  constructor(Convergence, EyeSeparation, AspectRatio, FOV, NearClippingDistance, FarClippingDistance) {

    this.mConvergence = Convergence;

    this.mEyeSeparation = EyeSeparation;

    this.mAspectRatio = AspectRatio;

    this.mFOV = (FOV \* Math.PI) / 180.0;

    this.mNearClippingDistance = NearClippingDistance;

    this.mFarClippingDistance = FarClippingDistance;

    this.projection = m4.identity();

    this.modelView = m4.identity();

  }

  ApplyLeftFrustum() {

    const top = this.mNearClippingDistance \* Math.tan(this.mFOV / 2);

    const bottom = -top;

    const a = this.mAspectRatio \* Math.tan(this.mFOV / 2) \* this.mConvergence;

    const b = a - this.mEyeSeparation / 2;

    const c = a + this.mEyeSeparation / 2;

    const left = (-b \* this.mNearClippingDistance) / this.mConvergence;

    const right = (c \* this.mNearClippingDistance) / this.mConvergence;

    this.projection = m4.frustum(left, right, bottom, top, this.mNearClippingDistance, this.mFarClippingDistance);

    this.modelView = m4.translation(this.mEyeSeparation / 2, 0.0, 0.0);

  }

  ApplyRightFrustum() {

    const top = this.mNearClippingDistance \* Math.tan(this.mFOV / 2);

    const bottom = -top;

    const a = this.mAspectRatio \* Math.tan(this.mFOV / 2) \* this.mConvergence;

    const b = a - this.mEyeSeparation / 2;

    const c = a + this.mEyeSeparation / 2;

    const left = (-c \* this.mNearClippingDistance) / this.mConvergence;

    const right = (b \* this.mNearClippingDistance) / this.mConvergence;

    this.projection = m4.frustum(left, right, bottom, top, this.mNearClippingDistance, this.mFarClippingDistance);

    this.modelView = m4.translation(-this.mEyeSeparation / 2, 0.0, 0.0);

  }

}

// Constructor

function Model(name) {

  this.name = name;

  this.iVertexBuffer = gl.createBuffer();

  this.iTexCoordsBuffer = gl.createBuffer();

  this.count = 0;

  this.sphereVerticesLength = 0;

  this.BufferData = function (vertices, normals, texCoords) {

    gl.bindBuffer(gl.ARRAY\_BUFFER, this.iVertexBuffer);

    gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(vertices), gl.STREAM\_DRAW);

    gl.bindBuffer(gl.ARRAY\_BUFFER, this.iTexCoordsBuffer);

    gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(texCoords), gl.STREAM\_DRAW);

    this.count = vertices.length / 3;

  };

  this.Draw = function () {

    gl.bindBuffer(gl.ARRAY\_BUFFER, this.iVertexBuffer);

    gl.vertexAttribPointer(shProgram.iAttribVertex, 3, gl.FLOAT, false, 0, 0);

    gl.enableVertexAttribArray(shProgram.iAttribVertex);

    gl.bindBuffer(gl.ARRAY\_BUFFER, this.iTexCoordsBuffer);

    gl.vertexAttribPointer(shProgram.iAttribTexCoord, 2, gl.FLOAT, false, 0, 0);

    gl.enableVertexAttribArray(shProgram.iAttribTexCoord);

    gl.drawArrays(gl.TRIANGLES, 0, this.count);

  };

  this.SBufferData = function (surfData) {

    gl.bindBuffer(gl.ARRAY\_BUFFER, this.iVertexBuffer);

    gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(surfData), gl.STREAM\_DRAW);

    this.sphereVerticesLength = surfData.length / 3;

  };

  this.SDraw = function () {

    gl.bindBuffer(gl.ARRAY\_BUFFER, this.iVertexBuffer);

    gl.vertexAttribPointer(shProgram.iAttribVertex, 3, gl.FLOAT, false, 0, 0);

    gl.enableVertexAttribArray(shProgram.iAttribVertex);

    gl.drawArrays(gl.TRIANGLE\_STRIP, 0, this.sphereVerticesLength);

  };

}

// Constructor

function ShaderProgram(name, program) {

  this.name = name;

  this.prog = program;

  // Location of the attribute variable in the shader program.

  this.iAttribVertex = -1;

  // Location of the uniform specifying a color for the primitive.

  this.iColor = -1;

  // Location of the uniform matrix representing the combined transformation.

  this.iModelViewProjectionMatrix = -1;

  this.iSphere = false;

  this.Use = function () {

    gl.useProgram(this.prog);

  };

}

/\* Draws a colored cube, along with a set of coordinate axes.

 \* (Note that the use of the above drawPrimitive function is not an efficient

 \* way to draw with WebGL.  Here, the geometry is so simple that it doesn't matter.)

 \*/

function draw() {

  const a = document.getElementById('a').value;

  const b = document.getElementById('b').value;

  const c = document.getElementById('c').value;

  const d = document.getElementById('d').value;

  const getF = (a, b, v) => {

    return (a \* b) / Math.sqrt(Math.pow(a, 2) + Math.pow(Math.sin(v), 2) + Math.pow(b, 2) \* Math.pow(Math.cos(v), 2));

  };

  const getVertex = (u, v) => {

    const uRad = u;

    const vRad = v;

    const x =

      (1 / 2) \*

      (getF(a, b, vRad) \* (1 + Math.cos(uRad)) +

        ((Math.pow(d, 2) - Math.pow(c, 2)) \* (1 - Math.cos(uRad))) / getF(a, b, vRad)) \*

      Math.cos(vRad);

    const y =

      (1 / 2) \*

      (getF(a, b, vRad) \* (1 + Math.cos(uRad)) +

        ((Math.pow(d, 2) - Math.pow(c, 2)) \* (1 - Math.cos(uRad))) / getF(a, b, vRad)) \*

      Math.sin(vRad);

    const z = (1 / 2) \* (getF(a, b, vRad) - (Math.pow(d, 2) - Math.pow(c, 2)) / getF(a, b, vRad)) \* Math.sin(uRad);

    return [x, y, z];

  };

  gl.clearColor(0, 0, 0, 1);

  gl.clear(gl.COLOR\_BUFFER\_BIT | gl.DEPTH\_BUFFER\_BIT);

  /\* Set the values of the projection transformation \*/

  let projection = m4.perspective(Math.PI / 4, 1, 6, 14);

  /\* Get the view matrix from the SimpleRotator object.\*/

  let modelView = spaceball.getViewMatrix();

  let rotateToPointZero = m4.axisRotation([0.707, 0.707, 0], 0.7);

  let translateToPointZero = m4.translation(0, 0, -10);

  let matAccum0 = m4.multiply(rotateToPointZero, modelView);

  let matAccum1 = m4.multiply(translateToPointZero, matAccum0);

  camera.mConvergence = parseFloat(document.getElementById('conv').value);

  camera.mEyeSeparation = parseFloat(document.getElementById('eyes').value);

  camera.mFOV = parseFloat(document.getElementById('fov').value);

  camera.mNearClippingDistance = parseFloat(document.getElementById('near').value);

  gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false, m4.identity());

  gl.bindTexture(gl.TEXTURE\_2D, textureWebcam);

  gl.texImage2D(gl.TEXTURE\_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED\_BYTE, webcam);

  surfaceWebcam.Draw();

  gl.bindTexture(gl.TEXTURE\_2D, texture);

  gl.clear(gl.DEPTH\_BUFFER\_BIT);

  /\* Multiply the projection matrix times the modelview matrix to give the

       combined transformation matrix, and send that to the shader program. \*/

  let modelViewProjection = m4.multiply(projection, matAccum1);

  camera.ApplyLeftFrustum();

  modelViewProjection = m4.multiply(camera.projection, m4.multiply(camera.modelView, matAccum1));

  gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false, modelViewProjection);

  gl.colorMask(true, false, false, false);

  surface.Draw();

  gl.uniform4fv(shProgram.iColor, [1.0, 1.0, 0.0, 1]);

  gl.uniform1i(shProgram.iSphere, true);

  sphere.SDraw();

  gl.uniform1i(shProgram.iSphere, false);

  gl.clear(gl.DEPTH\_BUFFER\_BIT);

  camera.ApplyRightFrustum();

  modelViewProjection = m4.multiply(camera.projection, m4.multiply(camera.modelView, matAccum1));

  gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false, modelViewProjection);

  gl.colorMask(false, true, true, false);

  surface.Draw();

  gl.uniform1i(shProgram.iSphere, true);

  sphere.SDraw();

  gl.uniform1i(shProgram.iSphere, false);

  gl.colorMask(true, true, true, true);

  gl.uniform4fv(shProgram.iColor, [1, 1, 0, 1]);

  gl.colorMask(true, true, true, true);

}

function animate() {

  draw();

  window.requestAnimationFrame(animate);

}

function updSrf() {

  surface.BufferData(...CreateSurfaceData());

  draw();

}

function CreateSurfaceData() {

  let vertexList = [],

    normalList = [],

    textureList = [];

  const a = document.getElementById('a').value;

  const b = document.getElementById('b').value;

  const c = document.getElementById('c').value;

  const d = document.getElementById('d').value;

  const getF = (a, b, v) => {

    return (a \* b) / Math.sqrt(Math.pow(a, 2) + Math.pow(Math.sin(v), 2) + Math.pow(b, 2) \* Math.pow(Math.cos(v), 2));

  };

  const getVertex = (u, v) => {

    const uRad = deg2rad(u);

    const vRad = deg2rad(v);

    const x =

      (1 / 2) \*

      (getF(a, b, vRad) \* (1 + Math.cos(uRad)) +

        ((Math.pow(d, 2) - Math.pow(c, 2)) \* (1 - Math.cos(uRad))) / getF(a, b, vRad)) \*

      Math.cos(vRad);

    const y =

      (1 / 2) \*

      (getF(a, b, vRad) \* (1 + Math.cos(uRad)) +

        ((Math.pow(d, 2) - Math.pow(c, 2)) \* (1 - Math.cos(uRad))) / getF(a, b, vRad)) \*

      Math.sin(vRad);

    const z = (1 / 2) \* (getF(a, b, vRad) - (Math.pow(d, 2) - Math.pow(c, 2)) / getF(a, b, vRad)) \* Math.sin(uRad);

    return [0.75 \* x, 0.75 \* y, 0.75 \* z];

  };

  for (let u = 0; u <= 360; u += 5) {

    for (let v = 0; v <= 360; v += 5) {

      let vertex1 = getVertex(u, v);

      let vertex2 = getVertex(u + 5, v);

      let vertex3 = getVertex(u, v + 5);

      let vertex4 = getVertex(u + 5, v + 5);

      vertexList.push(...vertex1);

      vertexList.push(...vertex2);

      vertexList.push(...vertex3);

      vertexList.push(...vertex3);

      vertexList.push(...vertex2);

      vertexList.push(...vertex4);

      textureList.push(u / 360, v / 360);

      textureList.push((u + 5) / 360, v / 360);

      textureList.push(u / 360, (v + 5) / 360);

      textureList.push(u / 360, (v + 5) / 360);

      textureList.push((u + 5) / 360, v / 360);

      textureList.push((u + 5) / 360, (v + 5) / 360);

    }

  }

  return [vertexList, normalList, textureList];

}

/\* Initialize the WebGL context. Called from init() \*/

function initGL() {

  CreateCamera();

  let prog = createProgram(gl, vertexShaderSource, fragmentShaderSource);

  shProgram = new ShaderProgram('Basic', prog);

  shProgram.Use();

  CreateWebcamTexture();

  shProgram.iAttribVertex = gl.getAttribLocation(prog, 'vertex');

  shProgram.iColor = gl.getUniformLocation(prog, 'color');

  shProgram.iAttribTexCoord = gl.getAttribLocation(prog, 'texCoord');

  shProgram.iModelViewProjectionMatrix = gl.getUniformLocation(prog, 'ModelViewProjectionMatrix');

  shProgram.iSphere = gl.getUniformLocation(prog, 'iSphere');

  surfaceWebcam = new Model();

  surfaceWebcam.BufferData(

    [-1, -1, 0, 1, 1, 0, 1, -1, 0, 1, 1, 0, -1, -1, 0, -1, 1, 0],

    [],

    [1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0]

  );

  surface = new Model('Surface');

  surface.BufferData(...CreateSurfaceData());

  sphere = new Model('Sphere');

  sphere.SBufferData(CreateSphereData());

  document.getElementById('filter').addEventListener('change', function (event) {

    if (!audioContext) return;

    const value = event.target.checked;

    audioSource.disconnect();

    audioPanner.disconnect();

    if (value) {

      audioSource.connect(audioFilter);

      audioFilter.connect(audioPanner);

      audioFilter.connect(audioContext.destination);

    } else {

      audioSource.connect(audioPanner);

      audioPanner.connect(audioContext.destination);

    }

    audioPanner.setPosition(audioPosition.x, audioPosition.y, audioPosition.z);

    audioPanner.setOrientation(0, 0, 0);

  });

  document.getElementById('playBtn').addEventListener('click', playMusic);

  gl.enable(gl.DEPTH\_TEST);

}

/\* Creates a program for use in the WebGL context gl, and returns the

 \* identifier for that program.  If an error occurs while compiling or

 \* linking the program, an exception of type Error is thrown.  The error

 \* string contains the compilation or linking error.  If no error occurs,

 \* the program identifier is the return value of the function.

 \* The second and third parameters are strings that contain the

 \* source code for the vertex shader and for the fragment shader.

 \*/

function createProgram(gl, vShader, fShader) {

  let vsh = gl.createShader(gl.VERTEX\_SHADER);

  gl.shaderSource(vsh, vShader);

  gl.compileShader(vsh);

  if (!gl.getShaderParameter(vsh, gl.COMPILE\_STATUS)) {

    throw new Error('Error in vertex shader:  ' + gl.getShaderInfoLog(vsh));

  }

  let fsh = gl.createShader(gl.FRAGMENT\_SHADER);

  gl.shaderSource(fsh, fShader);

  gl.compileShader(fsh);

  if (!gl.getShaderParameter(fsh, gl.COMPILE\_STATUS)) {

    throw new Error('Error in fragment shader:  ' + gl.getShaderInfoLog(fsh));

  }

  let prog = gl.createProgram();

  gl.attachShader(prog, vsh);

  gl.attachShader(prog, fsh);

  gl.linkProgram(prog);

  if (!gl.getProgramParameter(prog, gl.LINK\_STATUS)) {

    throw new Error('Link error in program:  ' + gl.getProgramInfoLog(prog));

  }

  return prog;

}

/\*\*

 \* initialization function that will be called when the page has loaded

 \*/

function init() {

  let canvas;

  sphereRotation = { x: 0, y: 0, z: 0 };

  audioPosition = { x: 0, y: 0, z: 0 };

  try {

    canvas = document.getElementById('webglcanvas');

    gl = canvas.getContext('webgl');

    if (!gl) {

      throw 'Browser does not support WebGL';

    }

  } catch (e) {

    document.getElementById('canvas-holder').innerHTML = '<p>Sorry, could not get a WebGL graphics context.</p>';

    return;

  }

  try {

    initGL(); // initialize the WebGL graphics context

  } catch (e) {

    document.getElementById('canvas-holder').innerHTML =

      '<p>Sorry, could not initialize the WebGL graphics context: ' + e + '</p>';

    return;

  }

  spaceball = new TrackballRotator(canvas, draw, 0);

  camera = new StereoCamera(1000, 0.1, 1, 45, 1, 15);

  LoadTexture();

  spaceball = new TrackballRotator(canvas, draw, 0);

  draw();

  animate();

}

function LoadTexture() {

  texture = gl.createTexture();

  gl.bindTexture(gl.TEXTURE\_2D, texture);

  gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MIN\_FILTER, gl.LINEAR);

  gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MAG\_FILTER, gl.LINEAR);

  const image = new Image();

  image.crossOrigin = 'anonymus';

  image.src =

    'https://static.turbosquid.com/Preview/2014/08/01\_\_12\_04\_02/Urban\_\_Brickwall1.jpg766465EF-01F6-40FD-A9055898D5FDCEA3.jpgLarger.jpg';

  image.onload = () => {

    gl.bindTexture(gl.TEXTURE\_2D, texture);

    gl.texImage2D(gl.TEXTURE\_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED\_BYTE, image);

    console.log('imageLoaded');

    draw();

  };

}

function animateSphere() {

  let step = 0.05;

  let xAnimation = true;

  setInterval(() => {

    if (xAnimation) {

      sphereRotation.x += step;

      if (sphereRotation.x > 4 || sphereRotation.x < -4) {

        step = -step;

      }

      audioPosition.x = sphereRotation.x;

    } else {

      sphereRotation.y += step;

      if (sphereRotation.y > 2 || sphereRotation.y < -2) {

        step = -step;

      }

      audioPosition.y = sphereRotation.y;

    }

    audioPanner.setPosition(audioPosition.x, audioPosition.y, audioPosition.z);

    sphere.SBufferData(CreateSphereData());

    draw();

    if (xAnimation && Date.now() - startTime >= 30000) {

      xAnimation = false;

    }

  }, 100);

  let startTime = Date.now();

}

const cosTable = new Array(360);

const sinTable = new Array(360);

for (let i = 0; i <= 360; i++) {

  cosTable[i] = Math.cos(deg2rad(i));

  sinTable[i] = Math.sin(deg2rad(i));

}

function CreateSphereData() {

  const radius = 0.2;

  const res = [];

  const sphereX = sphereRotation.x;

  const sphereY = sphereRotation.y;

  const sphereZ = sphereRotation.z;

  for (let u = 0; u <= 360; u += 10) {

    for (let v = 0; v <= 360; v += 10) {

      const cosU = cosTable[u];

      const sinU = sinTable[u];

      const cosV = cosTable[v];

      const sinV = sinTable[v];

      const cosU1 = cosTable[(u + 10) % 360];

      const sinU1 = sinTable[(u + 10) % 360];

      const cosV2 = cosTable[(v + 10) % 360];

      const sinV2 = sinTable[(v + 10) % 360];

      res.push(

        sphereX + radius \* cosU \* sinV,

        sphereY + radius \* sinU \* sinV,

        sphereZ + radius \* cosV,

        sphereX + radius \* cosU1 \* sinV2,

        sphereY + radius \* sinU1 \* sinV2,

        sphereZ + radius \* cosV2

      );

    }

  }

  return res;

}

const params = {};

async function createAudio() {

  audioContext = new window.AudioContext();

  audioSource = audioContext.createBufferSource();

  getFilter();

  getAudioPanner();

  try {

    const response = await fetch('https://raw.githubusercontent.com/Mag1sterY0da/WebGL/CGW/sound.mp3');

    const audioData = await response.arrayBuffer();

    const buffer = await audioContext.decodeAudioData(audioData);

    audioSource.buffer = buffer;

    if (music.filter) {

      audioSource.connect(audioFilter);

      audioFilter.connect(audioPanner);

    } else {

      audioSource.connect(audioPanner);

    }

    audioPanner.connect(audioContext.destination);

    audioSource.loop = true;

  } catch (error) {

    console.error('Error loading audio:', error);

  }

}

function getFilter() {

  audioFilter = audioContext.createBiquadFilter();

  audioFilter.type = 'lowpass';

  audioFilter.frequency.value = 1000;

  audioFilter.Q.value = 1;

}

function getAudioPanner() {

  audioPanner = audioContext.createPanner();

  audioPanner.refDistance = 1;

  audioPanner.maxDistance = 1000;

  audioPanner.rolloffFactor = 1;

  audioPanner.coneInnerAngle = 360;

  audioPanner.coneOuterAngle = 0;

  audioPanner.coneOuterGain = 0;

  audioPanner.panningModel = 'HRTF';

  audioPanner.distanceModel = 'inverse';

  audioPanner.setPosition(audioPosition.x, audioPosition.y, audioPosition.z);

  audioPanner.setOrientation(0, 0, 0);

}

function playMusic() {

  params.audioPlay = !params.audioPlay;

  if (params.audioPlay) {

    if (audioContext) {

      audioContext.resume();

    } else {

      createAudio();

      audioSource.start(0);

    }

    setTimeout(animateSphere, 2000);

  } else audioContext.suspend();

}