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 №22

**Виконав:**

студент 5-го курсу

групи ТР-31мп

Позняк В.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.

**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**

Реалізовуємо функцію для завантаження аудіо з мережі інтернет за посиланням.

function createAudio() {  
 *audioContext* = new *window*.AudioContext();  
 *audioSource* = *audioContext*.createBufferSource();  
 createFilter();  
 createAudioPanner();  
 const request = new XMLHttpRequest();  
 request.open("GET", "https://raw.githubusercontent.com/DDoS-73/MoSoVR/CGW/music.mp3", true);  
 request.responseType = "arraybuffer";  
 request.onload = () => {  
 const audioData = request.response;  
 *audioContext*.decodeAudioData(audioData, (buffer) => {  
 *audioSource*.buffer = buffer;  
 if (*music*.filter) {  
 *audioSource*.connect(*audioFilter*);  
 *audioFilter*.connect(*audioPanner*);  
 } else {  
 *audioSource*.connect(*audioPanner*);  
 }  
 *audioPanner*.connect(*audioContext*.destination);  
 *audioSource*.loop = true;  
 }  
 );  
 };  
 request.send();  
}

Створюємо фільтр згідно варіанту

function createFilter() {  
 *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 createAudioPanner() {  
 *audioPanner* = *audioContext*.createPanner();  
 *audioPanner*.panningModel = "HRTF";  
 *audioPanner*.distanceModel = "inverse";  
 *audioPanner*.refDistance = 1;  
 *audioPanner*.maxDistance = 1000;  
 *audioPanner*.rolloffFactor = 1;  
 *audioPanner*.coneInnerAngle = 360;  
 *audioPanner*.coneOuterAngle = 0;  
 *audioPanner*.coneOuterGain = 0;  
  
  
 *audioPanner*.setPosition(*audioPosition*.x, *audioPosition*.y, *audioPosition*.z);  
 *audioPanner*.setOrientation(0,0,0);  
}

Створюємо буфер для зберігання координат сфери

this.BufferDataSphere = function(surfData) {  
 *gl*.bindBuffer(*gl*.*ARRAY\_BUFFER*, this.iVertexBuffer);  
 *gl*.bufferData(*gl*.*ARRAY\_BUFFER*, new *Float32Array*(surfData), *gl*.*STREAM\_DRAW*);  
  
 this.verticesLength = surfData.length / 3;  
}

Реалізуємо графічні елементи такі як перемикач фільтра і кнопка відтворення звуку

*gui*.add( *music*, 'filter' )  
 .name( 'Filter' )  
 .onChange( value => {  
 if (value) {  
 if (*audioContext*) {  
 *audioSource*.disconnect();  
 *audioPanner*.disconnect();  
 *audioSource*.connect(*audioFilter*);  
 *audioFilter*.connect(*audioPanner*);  
 *audioFilter*.connect(*audioContext*.destination);  
 *audioPanner*.setPosition(*audioPosition*.x, *audioPosition*.y, *audioPosition*.z);  
 *audioPanner*.setOrientation(0,0,0);  
 }  
 } else {  
 if (*audioContext*) {  
 *audioSource*.disconnect();  
 *audioPanner*.disconnect();  
 *audioSource*.connect(*audioPanner*);  
 *audioPanner*.connect(*audioContext*.destination);  
 *audioPanner*.setPosition(*audioPosition*.x, *audioPosition*.y, *audioPosition*.z);  
 *audioPanner*.setOrientation(0,0,0);  
 }  
 }  
 } );  
*gui*.add(*music*, 'play');

**Використання програми**

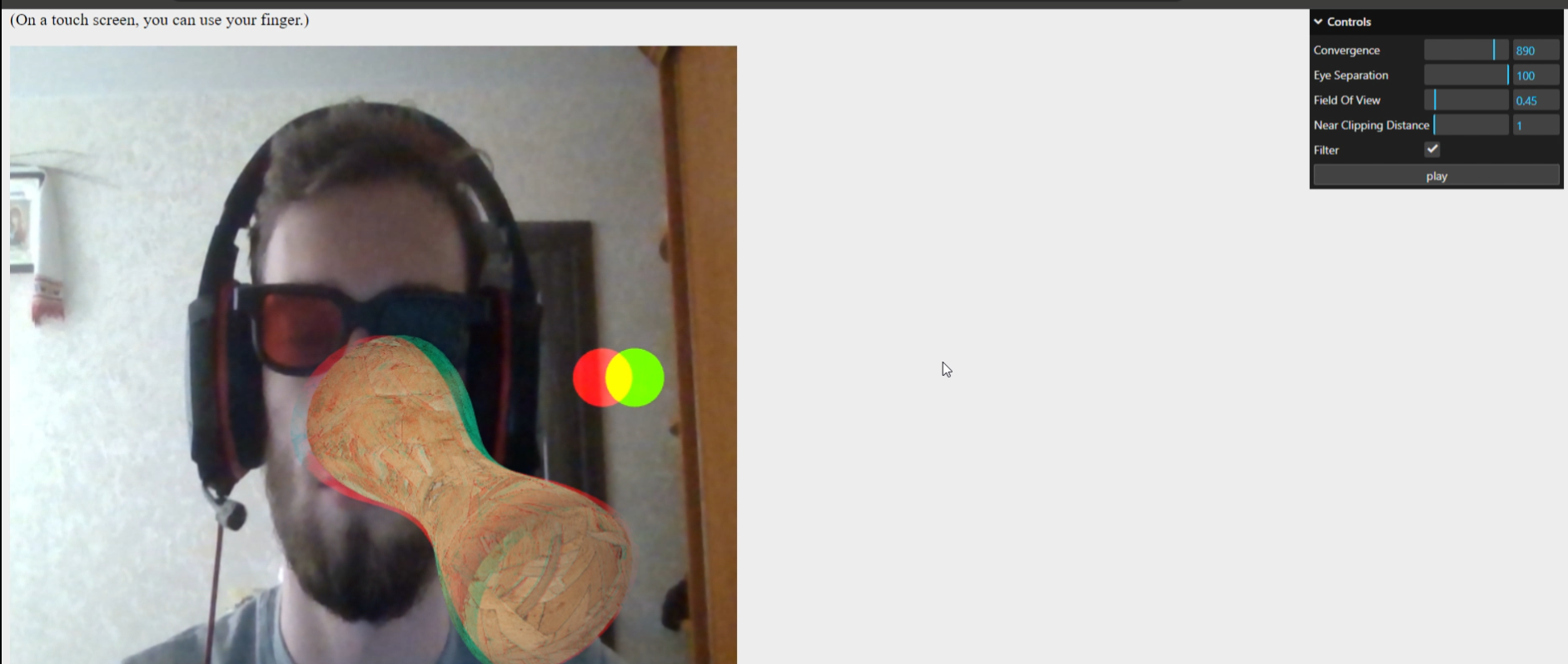
На рисунку 1 зображена сфера в правому положенні, відповідно звук відтворюється в правому навушнику:  


Рисунок 1 Сфера в правому положенні

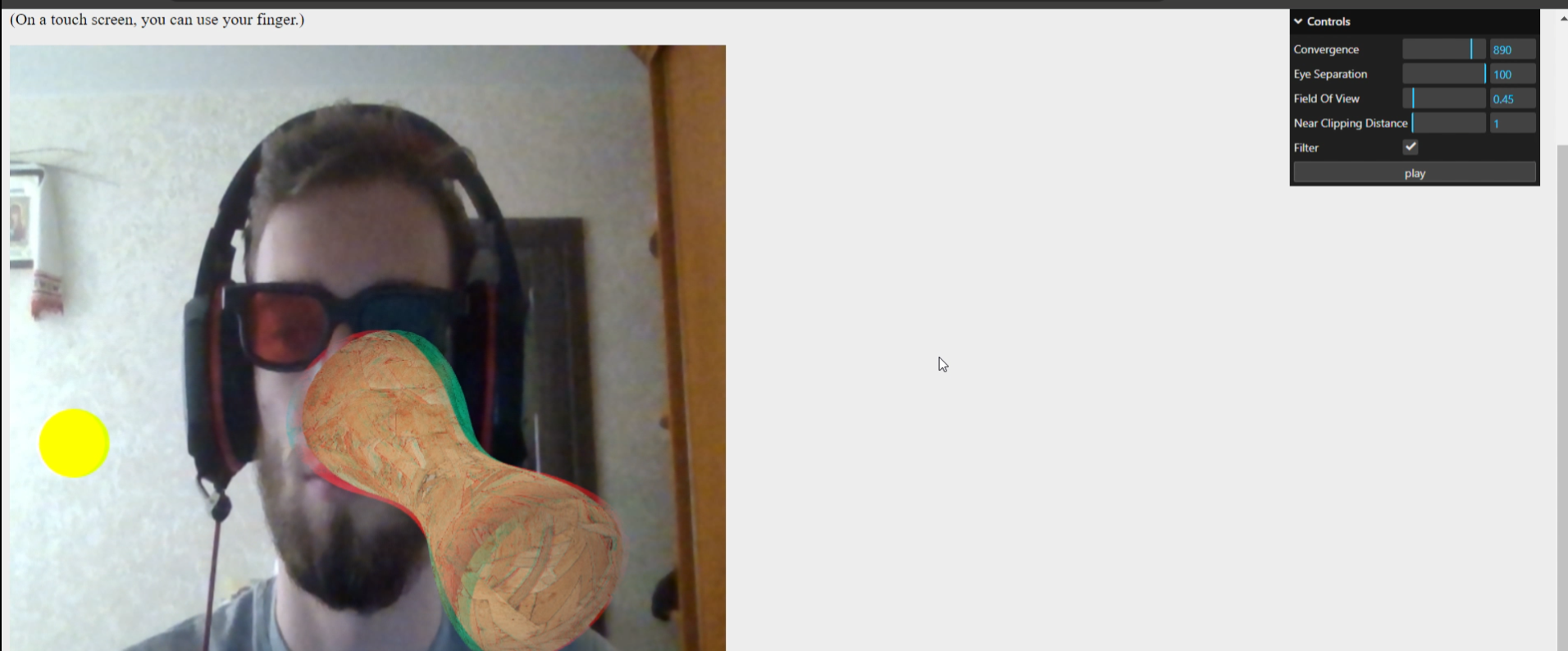
На рисунку 2 зображена сфера в лівому положенні, відповідно звук відтворюється в лівому навушнику: 

Рисунок 2 Сфера в лівому положенні

За допомогою перемикача вимикаємо фільтр, результат зображений на рисунку 3.

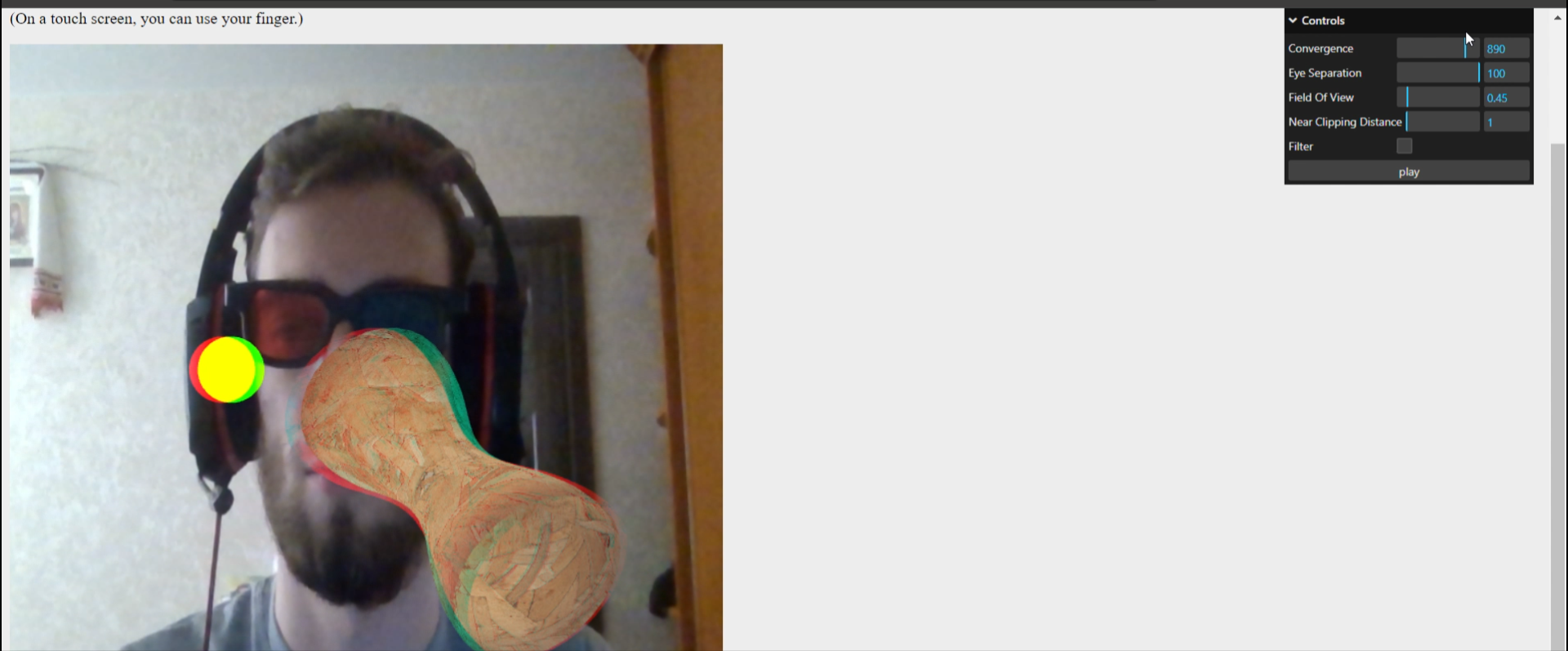


Рисунок 3 Вимкнений фільтр

**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*;  
let *gui*;  
let *imageTexture*, *videoTexture*;  
let *video*;  
let *videoSurface*;  
  
let *audioContext*;  
let *audioSource*;  
let *audioPanner*;  
let *audioFilter*;  
let *audioPosition*;  
  
let *sphere*;  
let *sphereRotation*;  
let *music*;  
  
function deg2rad(angle) {  
 return angle \* *Math*.PI / 180;  
}  
  
function LoadTexture() {  
 *imageTexture* = *gl*.createTexture();  
 *gl*.bindTexture(*gl*.*TEXTURE\_2D*, *imageTexture*);  
 *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*);  
  
 let image = new Image();  
 image.crossOrigin = 'anonymous';  
 image.src = "https://images.pexels.com/photos/168442/pexels-photo-168442.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=1";  
 image.onload = () => {  
 *gl*.bindTexture(*gl*.*TEXTURE\_2D*, *imageTexture*);  
 *gl*.texImage2D(*gl*.*TEXTURE\_2D*, 0, *gl*.*RGBA*, *gl*.*RGBA*, *gl*.*UNSIGNED\_BYTE*, image);  
  
 draw();  
 }  
}  
  
  
// Constructor  
function Model(name) {  
 this.name = name;  
 this.iVertexBuffer = *gl*.createBuffer();  
 this.iTextureBuffer = *gl*.createBuffer();  
 this.count = 0;  
 this.verticesLength = 0;  
  
 this.BufferData = function (vertices, texture) {  
 *gl*.bindBuffer(*gl*.*ARRAY\_BUFFER*, this.iVertexBuffer);  
 *gl*.bufferData(*gl*.*ARRAY\_BUFFER*, new *Float32Array*(vertices), *gl*.*STREAM\_DRAW*);  
  
 this.count = vertices.length / 3;  
  
 *gl*.bindBuffer(*gl*.*ARRAY\_BUFFER*, this.iTextureBuffer);  
 *gl*.bufferData(*gl*.*ARRAY\_BUFFER*, new *Float32Array*(texture), *gl*.*STREAM\_DRAW*);  
 }  
  
 this.BufferDataSphere = function(surfData) {  
 *gl*.bindBuffer(*gl*.*ARRAY\_BUFFER*, this.iVertexBuffer);  
 *gl*.bufferData(*gl*.*ARRAY\_BUFFER*, new *Float32Array*(surfData), *gl*.*STREAM\_DRAW*);  
  
 this.verticesLength = surfData.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.iTextureBuffer);  
 *gl*.vertexAttribPointer(*shProgram*.iAttribTexture, 2, *gl*.*FLOAT*, false, 0, 0);  
 *gl*.enableVertexAttribArray(*shProgram*.iAttribTexture);  
  
 *gl*.drawArrays(*gl*.*TRIANGLE\_STRIP*, 0, this.count);  
 }  
  
 this.DrawSphere = 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.verticesLength);  
 };  
}  
  
  
// 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.iDrawSphere = false;  
  
 this.Use = function () {  
 *gl*.useProgram(this.prog);  
 }  
}  
  
// Constructor  
function StereoCamera(  
 Convergence,  
 EyeSeparation,  
 AspectRatio,  
 FOV,  
 NearClippingDistance,  
 FarClippingDistance  
) {  
 this.mConvergence = Convergence;  
 this.mEyeSeparation = EyeSeparation;  
 this.mAspectRatio = AspectRatio;  
 this.mFOV = FOV  
 this.mNearClippingDistance = NearClippingDistance;  
 this.mFarClippingDistance = FarClippingDistance;  
 this.mProjectionMatrix;  
 this.mModelViewMatrix;  
  
 this.ApplyLeftFrustum = function () {  
 let top, bottom, left, right;  
  
 top = this.mNearClippingDistance \* *Math*.tan(this.mFOV / 2);  
 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;  
  
 left = -b \* this.mNearClippingDistance / this.mConvergence;  
 right = c \* this.mNearClippingDistance / this.mConvergence;  
  
 this.mProjectionMatrix = m4.frustum(left, right, bottom, top,  
 this.mNearClippingDistance, this.mFarClippingDistance)  
 this.mModelViewMatrix = m4.identity()  
 this.mModelViewMatrix = m4.multiply(m4.translation(0.01 \* this.mEyeSeparation / 2, 0.0, 0.0), this.mModelViewMatrix, this.mModelViewMatrix);  
 }  
  
 this.ApplyRightFrustum = function () {  
 let top, bottom, left, right;  
  
 top = this.mNearClippingDistance \* *Math*.tan(this.mFOV / 2);  
 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;  
  
 left = -c \* this.mNearClippingDistance / this.mConvergence;  
 right = b \* this.mNearClippingDistance / this.mConvergence;  
  
 this.mProjectionMatrix = m4.frustum(left, right, bottom, top,  
 this.mNearClippingDistance, this.mFarClippingDistance)  
 this.mModelViewMatrix = m4.identity()  
 this.mModelViewMatrix = m4.multiply(m4.translation(-0.01 \* this.mEyeSeparation / 2, 0.0, 0.0), this.mModelViewMatrix, this.mModelViewMatrix);  
 }  
}  
  
  
/\* 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(animate = false) {  
 *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 / 8, 1, 8, 12);  
 // let projection = m4.orthographic(-20, 20, -20, 20, -20, 20);  
 /\* 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);  
  
 /\* 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);  
 *gl*.uniformMatrix4fv(*shProgram*.iModelViewProjectionMatrix, false, m4.identity());  
 *gl*.bindTexture(*gl*.*TEXTURE\_2D*, *videoTexture*);  
 *gl*.texImage2D(*gl*.*TEXTURE\_2D*, 0, *gl*.*RGBA*, *gl*.*RGBA*, *gl*.*UNSIGNED\_BYTE*, *video*);  
 *videoSurface*.Draw();  
 *gl*.clear(*gl*.*DEPTH\_BUFFER\_BIT*);  
 *gl*.bindTexture(*gl*.*TEXTURE\_2D*, *imageTexture*);  
  
 *camera*.ApplyLeftFrustum();  
 modelViewProjection = m4.multiply(*camera*.mProjectionMatrix, m4.multiply(*camera*.mModelViewMatrix, 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*.iDrawSphere, true);  
 *sphere*.DrawSphere();  
 *gl*.uniform1i(*shProgram*.iDrawSphere, false);  
 *gl*.clear(*gl*.*DEPTH\_BUFFER\_BIT*);  
  
 *camera*.ApplyRightFrustum();  
 modelViewProjection = m4.multiply(*camera*.mProjectionMatrix, m4.multiply(*camera*.mModelViewMatrix, matAccum1));  
 *gl*.uniformMatrix4fv(*shProgram*.iModelViewProjectionMatrix, false, modelViewProjection);  
 *gl*.colorMask(false, true, true, false);  
 *surface*.Draw();  
 *gl*.uniform1i(*shProgram*.iDrawSphere, true);  
 *sphere*.DrawSphere();  
 *gl*.uniform1i(*shProgram*.iDrawSphere, false);  
 *gl*.colorMask(true, true, true, true);  
 /\* Draw the six faces of a cube, with different colors. \*/  
 *gl*.uniform4fv(*shProgram*.iColor, [1, 1, 0, 1]);  
 if(animate){  
 *window*.requestAnimationFrame(draw);  
 }  
}  
  
function CreateSurfaceData() {  
 let vertexList = [];  
 let textureList = [];  
 let scale = 0.1;  
 let R2 = 4;  
 let R1 = 1.3 \* R2;  
 const a = R2 - R1;  
 const c = 4 \* R1;  
 const b = c;  
  
 // Surface of Conjugation of Coaxial Cylinder and Cone  
  
 for (let z = 0; z < b; z += 0.5) {  
 for (let beta = 0; beta < 2 \* *Math*.PI; beta += 0.2) {  
 const p1 = createPoint(beta, z);  
 const p2 = createPoint(beta, z + 0.5);  
 const p3 = createPoint(beta + 0.2, z);  
 const p4 = createPoint(beta + 0.2, z + 0.5);  
  
 vertexList.push(scale \* p1.x, scale \* p1.y, scale \* p1.z);  
  
 vertexList.push(scale \* p2.x, scale \* p2.y, scale \* p2.z);  
  
 vertexList.push(scale \* p3.x, scale \* p3.y, scale \* p3.z);  
  
 vertexList.push(scale \* p4.x, scale \* p4.y, scale \* p4.z);  
  
 const texturePoint1 = [z / b, beta / (2 \* *Math*.PI)]  
 const texturePoint2 = [(z + 0.5) / b, beta / (2 \* *Math*.PI)]  
 const texturePoint3 = [z / b, (beta + 0.2) / (2 \* *Math*.PI)]  
 const texturePoint4 = [(z + 0.5) / b, (beta + 0.2) / (2 \* *Math*.PI)]  
  
 textureList.push(...texturePoint1, ...texturePoint2, ...texturePoint3, ...texturePoint4);  
 }  
 }  
  
 return { vertices: vertexList, texture: textureList };  
}  
  
function createPoint(beta, z) {  
 let R2 = 4;  
 let R1 = 1.3 \* R2;  
 const a = R2 - R1;  
 const c = 4 \* R1;  
 const b = c;  
  
 const r = a \* (1 - *Math*.cos(2 \* *Math*.PI \* z / c)) + R1;  
 const x = r \* *Math*.cos(beta);  
 const y = r \* *Math*.sin(beta);  
 return { x, y, z };  
}  
  
  
/\* Initialize the WebGL context. Called from init() \*/  
function initGL() {  
 let prog = createProgram(*gl*, vertexShaderSource, fragmentShaderSource);  
  
 *shProgram* = new ShaderProgram('Basic', prog);  
 *shProgram*.Use();  
  
 *shProgram*.iAttribVertex = *gl*.getAttribLocation(prog, "vertex");  
 *shProgram*.iModelViewProjectionMatrix = *gl*.getUniformLocation(prog, "ModelViewProjectionMatrix");  
 *shProgram*.iColor = *gl*.getUniformLocation(prog, "color");  
 *shProgram*.iAttribTexture = *gl*.getAttribLocation(prog, "textureCoord");  
 *shProgram*.iTMU = *gl*.getUniformLocation(prog, "tmu");  
 *shProgram*.iDrawSphere = *gl*.getUniformLocation(prog, "drawSphere");  
  
 *surface* = new Model('Surface');  
 *videoSurface* = new Model('Surface');  
 LoadTexture();  
 const data = CreateSurfaceData();  
 *surface*.BufferData(data.vertices, data.texture);  
 *videoSurface*.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]  
 );  
  
 *videoTexture* = CreateTexture();  
 CreateWebCamera();  
  
 *camera* = new StereoCamera(1000, 100, 1, 0.45, 1, 15,)  
 *music* = {  
 filter: true,  
 play: playMusic  
 }  
 *gui* = new GUI()  
 *gui*.add(*camera*, 'mConvergence', 350, 1000, 10).name('Convergence').onChange(draw)  
 *gui*.add(*camera*, 'mEyeSeparation', 0, 100).name('Eye Separation').onChange(draw)  
 *gui*.add(*camera*, 'mFOV', 0.1, 3.1).name('Field Of View').onChange(draw)  
 *gui*.add(*camera*, 'mNearClippingDistance', 6, 14).name('Near Clipping Distance').onChange(draw)  
  
 *gui*.add( *music*, 'filter' )  
 .name( 'Filter' )  
 .onChange( value => {  
 if (value) {  
 if (*audioContext*) {  
 *audioSource*.disconnect();  
 *audioPanner*.disconnect();  
 *audioSource*.connect(*audioFilter*);  
 *audioFilter*.connect(*audioPanner*);  
 *audioFilter*.connect(*audioContext*.destination);  
 *audioPanner*.setPosition(*audioPosition*.x, *audioPosition*.y, *audioPosition*.z);  
 *audioPanner*.setOrientation(0,0,0);  
 }  
 } else {  
 if (*audioContext*) {  
 *audioSource*.disconnect();  
 *audioPanner*.disconnect();  
 *audioSource*.connect(*audioPanner*);  
 *audioPanner*.connect(*audioContext*.destination);  
 *audioPanner*.setPosition(*audioPosition*.x, *audioPosition*.y, *audioPosition*.z);  
 *audioPanner*.setOrientation(0,0,0);  
 }  
 }  
 } );  
 *gui*.add(*music*, 'play');  
  
 *sphere* = new Model('Sphere');  
 *sphere*.BufferDataSphere(CreateSphereData());  
  
 *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;  
  
 initPositions();  
 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);  
 draw(true);  
}  
  
function CreateTexture() {  
 const 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*);  
 *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*);  
 return texture;  
}  
  
function CreateWebCamera() {  
 *video* = *document*.createElement('video');  
 *video*.setAttribute('autoplay', true);  
 *window*.vid = *video*;  
 *navigator*.getUserMedia({ video: true, audio: false }, function (stream) {  
 *video*.srcObject = stream;  
 }, function (e) {  
 *console*.error('Rejected!', e);  
 });  
 return *video*;  
}  
  
let *parameters* = {};  
function createAudio() {  
 *audioContext* = new *window*.AudioContext();  
 *audioSource* = *audioContext*.createBufferSource();  
 createFilter();  
 createAudioPanner();  
 const request = new XMLHttpRequest();  
 request.open("GET", "https://raw.githubusercontent.com/DDoS-73/MoSoVR/CGW/music.mp3", true);  
 request.responseType = "arraybuffer";  
 request.onload = () => {  
 const audioData = request.response;  
 *audioContext*.decodeAudioData(audioData, (buffer) => {  
 *audioSource*.buffer = buffer;  
 if (*music*.filter) {  
 *audioSource*.connect(*audioFilter*);  
 *audioFilter*.connect(*audioPanner*);  
 } else {  
 *audioSource*.connect(*audioPanner*);  
 }  
 *audioPanner*.connect(*audioContext*.destination);  
 *audioSource*.loop = true;  
 }  
 );  
 };  
 request.send();  
}  
  
function createFilter() {  
 *audioFilter* = *audioContext*.createBiquadFilter();  
 *audioFilter*.type = "lowpass";  
 *audioFilter*.frequency.value = 1000;  
 *audioFilter*.Q.value = 1;  
}  
function createAudioPanner() {  
 *audioPanner* = *audioContext*.createPanner();  
 *audioPanner*.panningModel = "HRTF";  
 *audioPanner*.distanceModel = "inverse";  
 *audioPanner*.refDistance = 1;  
 *audioPanner*.maxDistance = 1000;  
 *audioPanner*.rolloffFactor = 1;  
 *audioPanner*.coneInnerAngle = 360;  
 *audioPanner*.coneOuterAngle = 0;  
 *audioPanner*.coneOuterGain = 0;  
  
  
 *audioPanner*.setPosition(*audioPosition*.x, *audioPosition*.y, *audioPosition*.z);  
 *audioPanner*.setOrientation(0,0,0);  
}  
  
function playMusic() {  
 if (*parameters*.audioPlay) {  
 *audioContext*.suspend();  
 } else {  
 if (*audioContext*) {  
 *audioContext*.resume();  
 } else {  
 createAudio();  
 *audioSource*.start(0);  
 }  
 }  
 *parameters*.audioPlay = !*parameters*.audioPlay;  
  
 setTimeout(() => {  
 let step = 0.05;  
 let xInterval = setInterval(() => {  
 *sphereRotation*.x += step;  
  
 if (*sphereRotation*.x > 2 || *sphereRotation*.x < -2) {  
 step = -step;  
 }  
 *audioPosition*.x = *sphereRotation*.x;  
  
 *audioPanner*.setPosition(*audioPosition*.x, *audioPosition*.y, *audioPosition*.z);  
 *audioPanner*.setOrientation(0,0,0);  
  
 *sphere*.BufferDataSphere(CreateSphereData());  
 draw();  
 }, 100);  
  
 setTimeout(() => {  
 clearInterval(xInterval);  
 setInterval(() => {  
 *sphereRotation*.y += step;  
  
 if (*sphereRotation*.y > 2 || *sphereRotation*.y < -2) {  
 step = -step;  
 }  
 *audioPosition*.y = *sphereRotation*.y;  
  
 *audioPanner*.setPosition(*audioPosition*.x, *audioPosition*.y, *audioPosition*.z);  
 *audioPanner*.setOrientation(0,0,0);  
  
 *sphere*.BufferDataSphere(CreateSphereData());  
 draw();  
 }, 100);  
 }, 30000)  
 }, 2000);  
}  
  
function initPositions() {  
 *sphereRotation* = new Position(0, 0, 0);  
 *audioPosition* = new Position(0, 0, 0);  
}  
  
function CreateSphereData()  
{  
 let radius = 0.2;  
 let res = [];  
 for (let u = 0; u <= 360; u += 10) {  
 for(let v = 0; v <= 360; v += 10) {  
 let uRad = deg2rad(u);  
 let vRad = deg2rad(v);  
 let uRad1 = deg2rad(u + 10);  
 let vRad2 = deg2rad(v + 10);  
 res.push(*sphereRotation*.x + (radius \* *Math*.cos(uRad) \* *Math*.sin(vRad)), *sphereRotation*.y + (radius \* *Math*.sin(uRad) \* *Math*.sin(vRad)), *sphereRotation*.z + (radius \* *Math*.cos(vRad)));  
 res.push(*sphereRotation*.x + (radius \* *Math*.cos(uRad1) \* *Math*.sin(vRad2)), *sphereRotation*.y + (radius \* *Math*.sin(uRad1) \* *Math*.sin(vRad2)), *sphereRotation*.z + (radius \* *Math*.cos(vRad2)));  
 }  
 }  
 return res;  
}  
  
class Position {  
 constructor(x, y, z) {  
 this.x = x;  
 this.y = y;  
 this.z = z;  
 }  
}