МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ
КАФЕДРА ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Звіт
з виконання розрахунково-графічної роботи
з дисципліни «Методи синтезу віртуальної реальності»
Варіант 11

Виконала:
Студентка 5-го курсу ІАТЕ
групи ТР-22мп
Краєвська Марія Дмитрівна
Перевірив:
Демчишин Анатолій Анатолійович

Київ-2023

Завдання розрахунково-графічної роботи полягає в створення інтерактивної аудіо програми, в якій користувач міг би керувати просторовим положенням джерела звуку за допомогою осяжного інтерфейсу на основі коду з попередньої практичної роботи.

Мета роботи: налагодження взаємодії між користувачем і аудіо програмою, що дозволяє відтворювати музичний твір та контролювати просторове положення звукового джерела.

Для досягнення цієї мети були сформульовані наступні завдання:

- Реалізація обертання джерела звуку: Завдяки інтерфейсу, користувач повинен мати змогу контролювати обертання джерела звуку навколо геометричного центру поверхневого патчу. Це дозволить створити враження просторового звучання і змінити напрямок звуку.

- Візуалізація положення джерела звуку: Для наглядності та легкості сприйняття, було вирішено візуалізувати положення джерела звуку за допомогою сфери. Це допоможе користувачу чітко бачити, як змінюється положення звукового джерела під час його обертання.

- Використання шелфового фільтра низьких частот: Додатковою вимогою було впровадження звукового фільтра. Варіантом обрано шелфовий фільтр низьких частот. Також до реалізованої програми було додано елемент прапорця, який дозволяє увімкнути або вимкнути фільтр. Користувач може налаштовувати параметри фільтра відповідно до свого смаку.

Теоретичні відомості

WebAudio HTML5 API є потужним інструментом для створення інтерактивних аудіо програм у веб-середовищі. Він надає можливості для створення, обробки та відтворення звуку в реальному часі. Основна концепція API полягає у побудові аудіо графу, який складається з вузлів, що представляють різні аудіо джерела, ефекти та виходи.

Ключові компоненти WebAudio API включають аудіо контекст (AudioContext), який відповідає за керування аудіо простором та часовим розкладом, аудіо вузли (AudioNode), які представляють оброблювальні одиниці аудіо графу, аудіо джерела (AudioSourceNode), що представляють початкові джерела звуку, аудіо ефекти (AudioEffectNode), які використовуються для обробки звуку, та виходи (AudioDestinationNode), які відтворюють оброблений звук.

Шелфовий фільтр низьких частот є одним із типів фільтрів, що використовуються для обробки аудіо сигналів. Він дозволяє пропускати частоти нижче заданої точки затухання, а при цьому знижувати рівень амплітуди сигналу на вищих частотах.

Основними параметрами шелфового фільтра низьких частот є частота затухання (Cutoff Frequency), яка визначає початок зниження амплітуди сигналу, коефіцієнт посилення (Gain), який визначає ступінь зниження амплітуди на вищих частотах, та схилення (Slope), що визначає нахил затухання поза частотою затухання.

Застосування шелфового фільтра низьких частот може бути корисним для керування еквалайзерами, підсилення або приглушення окремих частотних діапазонів звуку, дозволяючи створювати бажаний звуковий ефект або вирівнювати звучання аудіо матеріалу.

У тривимірному аудіо просторі звук може бути розміщений у просторі з використанням координат x, y, z. Це дозволяє створювати ефекти звуку, що рухається, або розташовувати звук у віртуальному просторі для створення реалістичного просторового враження.

Для керування просторовим розташуванням звуку використовуються аудіо позиціонування та панорамування. Аудіо позиціонування визначає положення звукового джерела у тривимірному просторі, в той час як панорамування контролює панорамування звуку між лівим та правим стереоканалами. Це дозволяє точно контролювати розташування звуку у віртуальному просторі та створювати ефекти просторового звучання.

Реалізація

Відповідно до поставлених задач, розпочнемо з візуалізації джерела звуку. В нашому випадку це сфера. Аналогічно до нашої основної фігури обчислюємо координати сфери.

```
function CreateSphereSurface(radius = 2) {
  const vertices = [];
  const increment = 0.5;
  const halfPi = Math.PI * 0.5;
  const fullPi = Math.PI;

  for (let longitude = -fullPi; longitude < fullPi; longitude += increment) {
  for (let latitude = -halfPi; latitude < halfPi; latitude += increment) {
  const lon1 = longitude;
  const lon2 = longitude + increment;
  const lat1 = latitude;
  const lat2 = latitude + increment;
  vertices.push(
    ...sphereSurfaceData(radius, lon1, lat1),
    ...sphereSurfaceData(radius, lon2, lat1),
    ...sphereSurfaceData(radius, lon1, lat2),
    ...sphereSurfaceData(radius, lon2, lat2),
    ...sphereSurfaceData(radius, lon2, lat1),
    ...sphereSurfaceData(radius, lon1, lat2)
  );
}
}
return vertices;
}
```

Створюємо об'єкт моделі сфери Sphere за допомогою конструктора Model("Sphere").
Далі викликається метод bufferData(), який відповідає за завантаження даних вершин сфери у буфер.

В цьому методі виконуються наступні кроки:

- Виконується зв'язування буфера iVertexBuffer з цільовим буфером gl.ARRAY_BUFFER, що означає, що ми прив'язуємо буфер для роботи з вершинними атрибутами.
- Викликається gl.bufferData() з параметрами gl.ARRAY_BUFFER, new Float32Array(vertices) та gl.STREAM_DRAW. Це означає, що

дані вершин у форматі Float32Array(vertices) будуть завантажені до буфера gl.ARRAY_BUFFER з режимом gl.STREAM_DRAW, що вказує, що дані можуть змінюватися рідко.

- Встановлюється змінна count, яка визначає кількість вершин у сфері. Це обчислюється за допомогою формули vertices.length / 3, оскільки кожна вершина задана трьома координатами (x, y, z).

```
bufferData(vertices) {
  gl.bindBuffer(gl.ARRAY_BUFFER, this.iVertexBuffer);
      gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STREAM_DRAW);

      this.count = vertices.length / 3;
}
```

Наступним кроком відредагуємо функцію draw(), яка відповідає за відтворення картинки, звуку та налаштування звукового панорамування.

Перевіримо наявність об'єкта audioPanner. Якщо він існує, то виконуються додаткові дії, пов'язані з налаштуванням звуку.
Змінюються параметри фільтрації звуку (audioFilter) - частота (frequency) і Q-фактор (Q), встановлюється гучність звуку (volume) і швидкість відтворення (playbackRate).
В залежності від працездатності гіроскопу встановлюються координати позиції звукового панорамування (audioPanner.setPosition()). Якщо він доступний, то використовуються координати, отримані з гіроскопа (x, y, z). Якщо ні, то використовуються координати задані вручну.
Отже, функція draw() відтворює звук та налаштовує звукове панорамування в залежності від заданих параметрів.

```
if (audioPanner) {
  audioFilter.frequency.value = centerFrequencyInput;
  audioFilter.Q.value = Q_value;
  audio.volume = Volume;
  audio.playbackRate = PlaybackRate;

  if (GyroscopeRotate) {
      audioPanner.setPosition(
          (x * 1000).toFixed(2),
          (y * 1000).toFixed(2),
          (z * 1000).toFixed(2)
      );
  } else {
      audioPanner.setPosition(
          World_X * 1000,
          World_Y * 1000,
          World_Z * 1000
      );
  }
```

Перейдемо до обробка події "play" для аудіооб'єкта. Коли аудіо починає відтворюватись, виконується наступна послідовність дій:

- Перевіряється наявність об'єкта audioContext. Якщо він не існує, створюється новий аудіо контекст (AudioContext) з використанням доступного API (window.AudioContext або window.webkitAudioContext).
- Створюється аудіо джерело (audioSource) на основі вхідного аудіо (audio) за допомогою audioContext.createMediaElementSource().
- Створюються об'єкти для звукового панорамування (audioPanner), фільтрації звуку (audioFilter) за допомогою відповідних методів audioContext.createPanner(), audioContext.createBiquadFilter(), audioContext.createConvolver().
- Встановлюються параметри для звукового панорамування (panningModel, distanceModel), фільтрації звуку (type, frequency, Q) а також гучність звуку (volume) та швидкість відтворення (playbackRate).

- Викликається метод audioContext.resume(), який відновлює аудіо контекст, якщо він був призупинений.

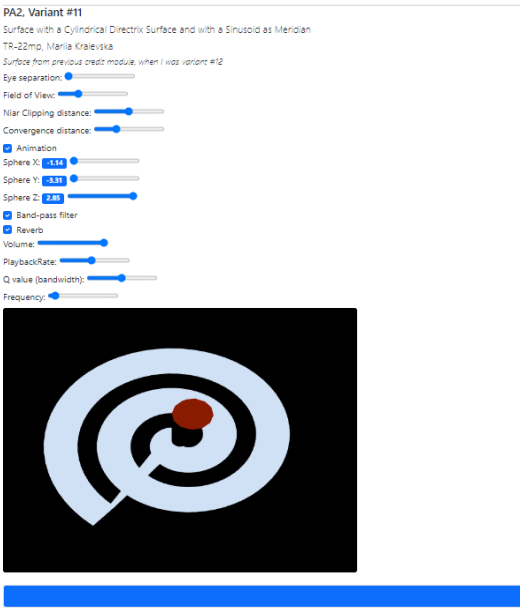
```
audio.addEventListener("play", () => {
  if (!audioContext) {
      audioContext = new(window.AudioContext || window.webkitAudioContext)();
      audioSource = audioContext.createMediaElementSource(audio);

      audioPanner = audioContext.createPanner();
      audioFilter = audioContext.createBiquadFilter();
      reverbNode = audioContext.createConvolver();

      audioPanner.panningModel = "HRTF";
      audioPanner.distanceModel = "linear";
      audioFilter.type = "lowpass";
      audioFilter.frequency.value = centerFrequencyInput;
      audioFilter.Q.value = Q_value;

      audio.volume = Volume;
      audio.playbackRate = PlaybackRate;

      audioContext.resume();
  }
```

Цей фрагмент коду виконує необхідні підготовчі дії для налаштування звукових ефектів та параметрів перед відтворенням аудіо.

Інструкція користувача

Розглянемо інтерфейс розробленого програмного продукту. Для управління сферою встановлено такі інструменти управління як Sphere x, Sphere y, Sphere z .

Оскільки не всі мобільні пристрої можуть запускати аудіо ресурси в браузері, а в нашому випадку координати центра сфери задаються з сенсора гіроскопу, що блокує просторове відтворення звуку. Також маємо управління шелфовим фільтром низьких частот, частотою звуку, швидкістю.



Лістинг коду

```
function readGyroscope() {
if (window.DeviceOrientationEvent) {
    timeStamp = Date.now();
    let sensor = new Gyroscope({
        frequency: 60
    });
    sensor.addEventListener('reading', e => {
        x = sensor.x
        y = sensor.y
        z = sensor.z
    });
    sensor.start();
} else alert('Gyroscope not supported');

}
function init() {
  let canvas;
  readGyroscope();

  try {
    canvas = document.getElementById("webglcanvas");
    CanvasWidth = canvas.scrollWidth;
    CanvasHeight = canvas.scrollHeight;

    gl = canvas.getContext("webgl");

    if (!gl) {
      throw "Browser does not support WebGL";
    }
  } catch (e) {
    console.log(e);
    document.getElementById("canvas-holder").innerHTML =
      "<p>Sorry, could not get a WebGL graphics
context.</p>";
    return;
  }

  try {
    initGL();
```

```
  } catch (e) {
    console.log(e);
    document.getElementById("canvas-holder").innerHTML =
      "<p>Sorry, could not initialize the WebGL graphics
context: " + e + "</p>";
    return;
  }

  video_cam = document.createElement('video');
  video_cam.setAttribute('autoplay', true);
  window.vid = video_cam;

  navigator.getUserMedia({
      video: true
  }, function (stream) {
      video_cam.srcObject = stream;
      let stream_settings =
stream.getVideoTracks()[0].getSettings();
      let stream_width = stream_settings.width;
      let stream_height = stream_settings.height;
      canvas = document.querySelector('canvas');
      gl = canvas.getContext("webgl");
      canvas.width = stream_width;
      canvas.height = stream_height;
      gl.viewport(0, 0, stream_width, stream_height);

  }, function (e) {
      console.error('Rejected!', e);
  });

  SetUpWebCamTexture();
  setTimeout(function () {
      spaceball = new TrackballRotator(canvas, draw, 0);

      if (texture_type == "image") {
          var texture = loadTexture(gl, image_src);
      } else {
          var texture = initTexture(gl);
          var video = setupVideo(video_src);
          var then = 0;
```

```
            function render(now) {
                now *= 0.001;
                var deltaTime = now - then;
                then = now;
                if (copyVideo) {
                    updateTexture(gl, texture, video);
                }
                if (texture_type != "image") {
                    requestAnimationFrame(render);
                }
            }
            requestAnimationFrame(render);
        }

        audio = document.getElementById("audio");

        audio.addEventListener("pause", () => {
            audioContext.resume();
        });

        audio.addEventListener("play", () => {
          if (!audioContext) {
                audioContext = new(window.AudioContext ||
window.webkitAudioContext)();
                audioSource =
audioContext.createMediaElementSource(audio);

                audioPanner = audioContext.createPanner();
                audioFilter =
audioContext.createBiquadFilter();
                reverbNode = audioContext.createConvolver();

                audioPanner.panningModel = "HRTF";
                audioPanner.distanceModel = "linear";
                audioFilter.type = "lowpass";
                audioFilter.frequency.value =
centerFrequencyInput;
                audioFilter.Q.value = Q_value;

                audio.volume = Volume;
                audio.playbackRate = PlaybackRate;
```

```
            audioContext.resume();
        }
    });

    const filter =
document.getElementById("filter_check");

    filter.addEventListener("change", function () {
        if (filter.checked) {
            audioPanner.disconnect();
            audioPanner.connect(audioFilter);

audioFilter.connect(audioContext.destination);
        } else {
            audioPanner.disconnect();

audioPanner.connect(audioContext.destination);
        }
    });


    audio.play();


      playVideo();
  }, 500);
}

function playVideo() {
  draw();
  window.requestAnimationFrame(playVideo);
}

function SetUpWebCamTexture() {
  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 initTexture(gl) {
  var texture = gl.createTexture();
  gl.bindTexture(gl.TEXTURE_2D, texture);

  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, 1, 1, 0,
gl.RGBA, gl.UNSIGNED_BYTE,
      new Uint8Array([0, 0, 255, 255]));

  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);

  return texture;
}

function updateTexture(gl, texture, video) {
  gl.bindTexture(gl.TEXTURE_2D, SurfaceTexture);
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
gl.UNSIGNED_BYTE, video);
  draw();
}
function CreateSphereSurface(radius = 2) {
  const vertices = [];
  const increment = 0.5;
  const halfPi = Math.PI * 0.5;
  const fullPi = Math.PI;

  for (let longitude = -fullPi; longitude < fullPi;
longitude += increment) {
```

```
  for (let latitude = -halfPi; latitude < halfPi;
latitude += increment) {
  const lon1 = longitude;
  const lon2 = longitude + increment;
  const lat1 = latitude;
  const lat2 = latitude + increment;
  vertices.push(
    ...sphereSurfaceData(radius, lon1, lat1),
    ...sphereSurfaceData(radius, lon2, lat1),
    ...sphereSurfaceData(radius, lon1, lat2),
    ...sphereSurfaceData(radius, lon2, lat2),
    ...sphereSurfaceData(radius, lon2, lat1),
    ...sphereSurfaceData(radius, lon1, lat2)
  );
}
}
return vertices;
}


function sphereSurfaceData(radius, longitude, latitude) {
  const sinLon = Math.sin(longitude);
  const cosLon = Math.cos(longitude);
  const sinLat = Math.sin(latitude);
  const cosLat = Math.cos(latitude);

  const x = radius * sinLon * cosLat;
  const y = radius * sinLon * sinLat;
  const z = radius * cosLon;

  return [x, y, z];
  }
function draw() {
  if (audioPanner) {
    audioFilter.frequency.value = centerFrequencyInput;
    audioFilter.Q.value = Q_value;
    audio.volume = Volume;
    audio.playbackRate = PlaybackRate;

    if (GyroscopeRotate) {
        audioPanner.setPosition(
```

```
            (x * 1000).toFixed(2),
            (y * 1000).toFixed(2),
            (z * 1000).toFixed(2)
        );
    } else {
        audioPanner.setPosition(
            World_X * 1000,
            World_Y * 1000,
            World_Z * 1000
        );
    }
}

gl.clearColor(0, 0, 0, 1);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

if (camera_ind) {
    DrawWebCamVideo();
}

gl.clear(gl.DEPTH_BUFFER_BIT);
stereoCamera.ApplyLeftFrustum();
gl.colorMask(true, false, false, false);
DrawSurface();
DrawSphere();

gl.clear(gl.DEPTH_BUFFER_BIT);
stereoCamera.ApplyRightFrustum();
gl.colorMask(false, true, true, false);
DrawSurface();
DrawSphere();
gl.colorMask(true, true, true, true);
}
class Model {
  constructor(name) {
    this.name = name;
    this.iVertexBuffer = gl.createBuffer();
    this.iTextureBuffer = gl.createBuffer();
    this.count = 0;
  }
```

```
  bufferData(vertices, textures) {
    gl.bindBuffer(gl.ARRAY_BUFFER, this.iVertexBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(vertices), gl.STREAM_DRAW);

    gl.bindBuffer(gl.ARRAY_BUFFER, this.iTextureBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(textures), gl.STREAM_DRAW);

    this.count = vertices.length / 3;
  }

  bufferData(vertices) {
    gl.bindBuffer(gl.ARRAY_BUFFER, this.iVertexBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(vertices), gl.STREAM_DRAW);

        this.count = vertices.length / 3;
  }

  draw() {
    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.iTextureCoords, 2,
gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(shProgram.iTextureCoords);
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, this.count);
  }
  drawSphere() {
    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.count);
  }
}
```