Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»
Інститут атомної та теплової енергетики
Кафедра цифрових технологій в енергетиці

Розрахунково-графічна робота
З дисципліни «Методи синтезу віртуальної реальності»

Виконав:
студент групи ТР-22 мп
Варіант 14
Кузяк Назарій

Київ 2023

Завдання

Варіант 14 імплементувати режекторний фільтр.

1. Завдання повинно бути завантажено в репозиторій на GitHub
2. Завдання повинно міститися в гілці, що має назву CGW
3. В репозиторії повинен міститися звіт до розрахунково-графічної роботи

1. реалізувати обертання джерела звуку навколо геометричного центру ділянки поверхні за допомогою інтерфейсу сенсора (цього разу поверхня залишається нерухомою, а джерело звуку рухається). Відтворити улюблену пісню у форматі mp3/ogg, маючи просторове розташування джерела звуку, кероване користувачем;
2. візуалізувати положення джерела звуку за допомогою сфери; додайте звуковий фільтр (використовуйте інтерфейс BiquadFilterNode) для кожного варіанту.
3. додати шельфовий фільтр низьких частот.
4. додати перемикач, який вмикає або вимикає фільтр. Встановіть параметри фільтра на свій смак.

Web Audio API та Режекторний фільтр

Режекторний фільтр, також відомий як notch-фільтр, є електронним пристроєм, який застосовується для пригнічення або відкидання сигналів у вузькому діапазоні частот. Його основною функцією є зниження або пригнічення вибраних частот для видалення не потрібних сигналів або шумів з сигналу.

Основні елементи режекторного фільтру включають операційні підсилювачі, резистори, конденсатори та індуктивності. Фільтр може бути реалізований як активний (з використанням операційних підсилювачів) або пасивний (з використанням резисторів, конденсаторів та індуктивностей).

Основний принцип роботи режекторного фільтру полягає у створенні підсилювача з негативним зворотним зв'язком, який має високу посилення на всіх частотах, крім діапазону частот, що підлягає пригніченню. Це досягається шляхом використання фільтруючих компонентів, таких як конденсатори та індуктивності, для створення вузькосмугового резонансного кола, яке піддаватиметься пригніченню.

Режекторний фільтр може бути налаштованим на певну частоту, яка потребує пригнічення. Це досягається шляхом правильного підбору значень компонентів фільтру. Вузькосмугове резонансне коло режекторного фільтру відбиває сигнали на налаштованій частоті, тим самим знижуючи їх амплітуду або видаляючи їх зовсім.

Одна з поширених конфігурацій режекторних фільтрів - це фільтр з активним дослідом (active twin-T filter). Він складається з двох гілок, що містять конденсатори і резистори, з'єднаних через операційний підсилювач. Цей тип фільтру може бути налаштований на певну частоту за допомогою змінних резисторів або конденсаторів.

Однак, варто зауважити, що точні характеристики режекторного фільтру залежатимуть від використовуваних компонентів, конфігурації фільтру та параметрів налаштування. Існують різні варіанти режекторних фільтрів, такі як активні, пасивні, аналогові та цифрові, кожен з яких має свої унікальні особливості та характеристики.

Узагальнюючи, режекторний фільтр є електронним пристроєм, призначеним для пригнічення сигналів у вузькому діапазоні частот. Він може бути використаний для видалення шумів або не потрібних сигналів зі сигналу і

може бути налаштований на певну частоту за допомогою відповідних компонентів фільтру.

Web Audio - це API (Application Programming Interface) веб-браузера, яке надає можливість створювати, маніпулювати та відтворювати аудіо на веб-сторінках. Це потужний інструмент, який дозволяє розробникам створювати різноманітні аудіо-додатки та ефекти безпосередньо у веб-браузері без необхідності встановлення додаткових плагінів чи залежностей.

Основні компоненти Web Audio API:

1. Аудіо контекст (AudioContext): Це основний об'єкт, який управляє всім аудіо веб-проекту. Він служить контейнером для всіх звукових джерел, ефектів та аудіо-вихідних пристроїв.
2. Звукові джерела (Audio Sources): Web Audio API надає різні типи звукових джерел, таких як аудіо-елемент HTML5, буферизований звук, генератори звуку (наприклад, звукові хвилі) та зовнішні звукові потоки (наприклад, мікрофон).
3. Ефекти та обробники (Effects & Processors): API має широкий набір вбудованих ефектів та обробників, таких як еквалайзер, реверберація, фільтри, компресори, довбання та багато інших. Ці компоненти дозволяють змінювати та обробляти звукові дані в реальному часі.
4. Підключення та маршрутизація (Connections & Routing): Звукові джерела та ефекти можна підключати до аудіо вузлів та маршрутизувати звукові дані від одного компонента до іншого. Це дозволяє створювати складні аудіо-ланцюжки та забезпечує гнучкість в обробці звуку.
5. Аудіо-вихідні пристрої (Audio Output Devices): Web Audio API підтримує відтворення звуку через різні вихідні пристрої, такі як динаміки комп'ютера, навушники або зовнішні аудіо-інтерфейси.

Хід роботи

У ході виконання роботи було розроблено програмний продукт, який реалізовує просторове аудіо з використанням Web Audio API та WebGL. Описуючи процес виконання цієї роботи, я розповім про додаткові деталі, що не були вказані у загальному описі.

Почавши з Web Audio API, першим кроком було створення аудіо-контексту за допомогою конструктора new AudioContext(). Цей контекст був основою для подальшої обробки аудіо. Потім було необхідно завантажити аудіо-файл, який використовувався для відтворення звуку. Це можна було зробити, використовуючи fetch або <audio> елемент.

Далі, було необхідно створити вузли Web Audio API для обробки звуку. Наприклад, створено AudioBufferSourceNode, який брав завантажений аудіо-файл як джерело звуку для відтворення. Щоб забезпечити просторовий звук, створено PannerNode, який відповідав за розміщення звуку в просторі. Позицію цього вузла можна було налаштувати, передавши в нього координати сфери, отримані з магнетометра.

Для візуалізації фігури було використано WebGL. Починаючи з налаштування <canvas> елементу у HTML-сторінці, було створено контекст WebGL за допомогою canvas.getContext('webgl'). Потім було необхідно створити шейдери - програми, які обробляють вершини та фрагменти графіки. Для цього були створені та компільовані вершинний та фрагментний шейдери, а потім було створено програму шейдера, яка поєднувала їх.

Після цього було необхідно налаштувати буфери вершин та атрибути вершин у WebGL. Це включало створення буфера вершин, завантаження координат вершин у цей буфер, а також налаштування атрибутів вершин, щоб WebGL знав, як інтерпретувати дані вершин.

Для відображення фігури на екрані було використано функцію gl.drawArrays(), яка використовувала вершинний буфер та шейдерну програму для рендерингу фігури. Це дозволяло відображати фігуру з використанням WebGL.

Нарешті, було додано обробники подій для отримання даних з магнетометра та оновлення позиції сфери, використовуючи PannerNode. Це забезпечувало

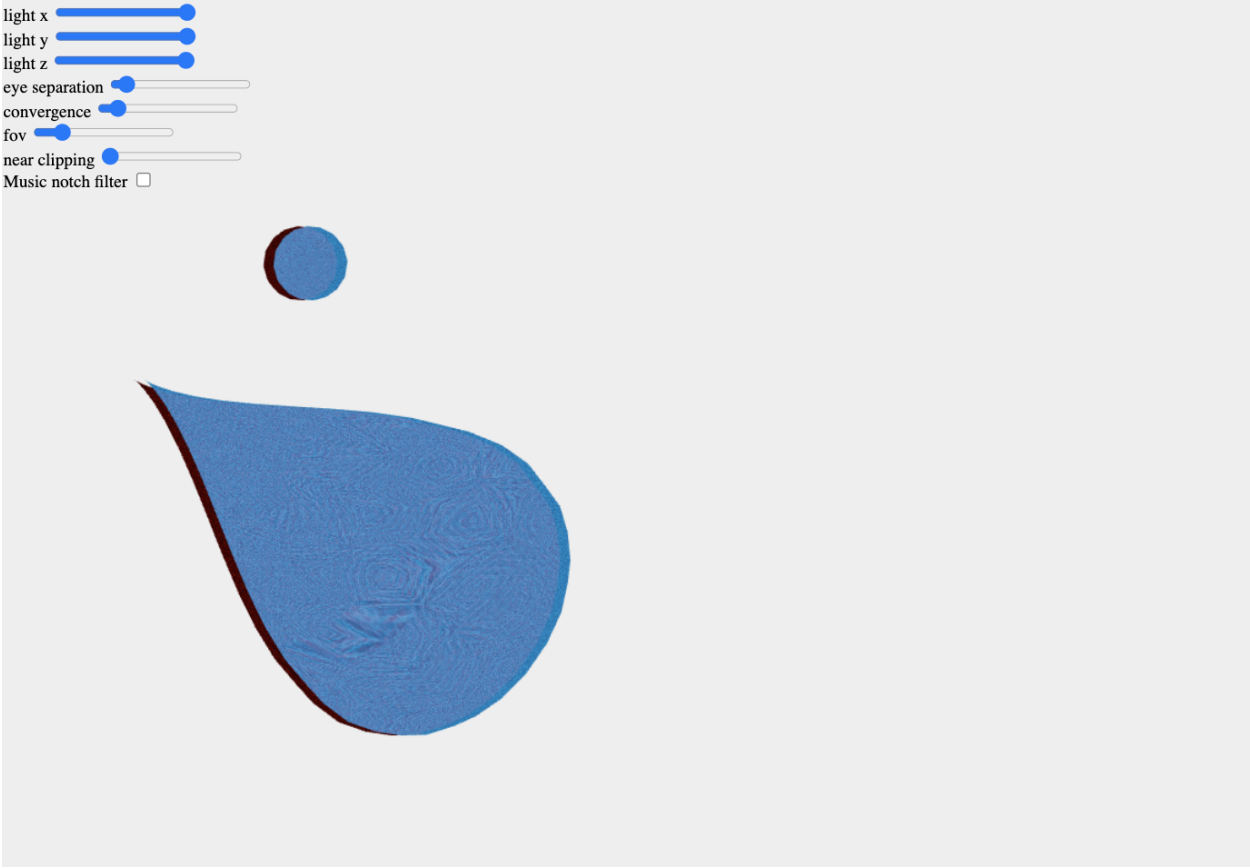
рух сфери відповідно до отриманих даних з магнетометра, що відображало позицію звуку у просторі.

Загалом, ця лабораторна робота включала розробку програмного продукту, який поєднував Web Audio API та WebGL для створення просторового аудіо. Вона вимагала налаштування аудіо-контексту, завантаження аудіо-файлу, створення та налаштування вузлів Web Audio API, налаштування контексту WebGL, створення та компіляції шейдерів, налаштування буферів та атрибутів вершин у WebGL, використання gl.drawArrays() для рендерингу фігури та обробки даних з магнетометра для оновлення позиції сфери.

```
// Створення аудіо-контексту
const audioContext = new AudioContext();

// Завантаження аудіо-файлу
fetch('audio-file.mp3')
  .then(response => response.arrayBuffer())
  .then(arrayBuffer => audioContext.decodeAudioData(arrayBuffer))
  .then(audioBuffer => {
    // Створення AudioBufferSourceNode
    const sourceNode = audioContext.createBufferSource();
    sourceNode.buffer = audioBuffer;

// Створення PannerNode
    const pannerNode = audioContext.createPanner();
    pannerNode.panningModel = 'HRTF';
    pannerNode.distanceModel = 'inverse';
    pannerNode.refDistance = 1;
    pannerNode.maxDistance = 10000;

    // Підключення вузлів
    sourceNode.connect(pannerNode);
    pannerNode.connect(audioContext.destination);

    // Відтворення звуку
    sourceNode.start(0);
```

Скриншоти

На першому скрншоті зображено фігуру зі сферами та вимкненим режекторним фільтром.



На другому скриншоті видно як сфера обертається навколо фігури. Також тут увімкнено фільтр.

Вихідний код

```
async function initAudio() {
  await new Promise(resolve => setTimeout(resolve, 2500));
  const audioContext = new AudioContext();
  const decodedAudioData = await fetch("/VR-KPI/music.mp3")
          .then(response => response.arrayBuffer())
          .then(audioData => audioContext.decodeAudioData(audioData));

  const source = audioContext.createBufferSource();
  source.buffer = decodedAudioData;
  source.connect(audioContext.destination);
  source.start();
  const panner = audioContext.createPanner();
  const volume = audioContext.createGain();
  volume.connect(panner);
  const highpass = audioContext.createBiquadFilter();
  highpass.type = "notch";
  highpass.frequency.value = 500;

  audio.panner = panner;
  audio.context = audioContext;
  audio.filter = highpass;
  audio.source = source;
  source.connect(highpass);

  window.setAudioPosition = (x, y, z) => {
          panner.positionX.value = x;
          panner.positionY.value = y;
          panner.positionZ.value = z;
  }
}

function initCheckBox() {
  const toggle = document.querySelector("#toggleFilter");

  toggle.onchange = e => {
          if (e.target.checked) {
          audio.filter.connect(audio.context.destination);
          } else {
          audio.filter.disconnect();
          }
  }
}


function setTexture(gl, image) {
          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.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
}


// Constructor
function Model() {
        this.BufferData = function() {
        const allVertices = vertices.concat(sphereVertices);
        const allUvs = uvs.concat(sphereUvs);
        const allBuffer = allVertices.concat(allUvs);

        // vertices
        const vBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(allBuffer), gl.STREAM_DRAW);

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

        gl.enableVertexAttribArray(shProgram.iAttribTexcoord);
        gl.vertexAttribPointer(shProgram.iAttribTexcoord, 2, gl.FLOAT, false, 0, allVertices * 4);
        }
}



// 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.iTextureAxis = -1;
        this.ITextureRotAngleDeg = -1;

        this.Use = function() {
        gl.useProgram(this.prog);
        }
}
```