

Міністерство освіти й науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря
Сікорського» Кафедра цифрових технологій в
енергетиці

Звіт
«Візуалізація графічної та геометричної інформації»

Розрахунково графічна робота
на тему: «Просторове аудіо»
Варіант №25

Виконав:
студент 5-го курсу
групи ТР-22мп ІАТЕ
Снитко О.Д.
Перевірив: Демчишин А.А.

Задача	3
Загальна інформація	3
Хід роботи	5
Зображення виконання	7
Вихідний код	10

Задача

Використавши код із практичного завдання №2:

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.

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

Загальна інформація

WebGL — це кросплатформний низькорівневий графічний API, який дозволяє розробникам створювати інтерактивну 2D і 3D графіку у веб-переглядачах. Однією з ключових особливостей WebGL є можливість використовувати GLSL (OpenGL Shading Language) для написання власних шейдерів, які можна запускати на графічному процесорі (графічному процесорі). Одним із поширених застосувань GLSL у WebGL є виконання матричних перетворень 2D та 3D об'єктів. Матричні перетворення використовуються для переміщення, обертання, масштабування та нахилу об'єктів у 3D-сцені. Ці перетворення можна комбінувати та об'єднувати для створення більш складних перетворень. Відображення текстури — це техніка, яка часто використовується в WebGL для додання реалізму 3D-моделям. Текстури — це зображення, які наносяться на поверхню тривимірного об'єкта, щоб надати йому певного вигляду, наприклад текстуру дерева або каменю. У WebGL текстури зазвичай завантажуються в графічний процесор і доступ до них

здійснюється за допомогою UV (текстурних) координат. UV-координати використовуються для визначення положення точки на двовимірному зображенні текстури по відношенню до тривимірного об'єкта, до якого вона застосована. UV-координати використовуються для інтерполяції між пікселями в зображенні текстури, що дозволяє плавно застосовувати текстуру до 3D-об'єкта. Використовуючи GLSL, можна маніпулювати UV-координатами текстури для досягнення різних ефектів, таких як масштабування, обертання або зміщення текстури. Це може бути корисним для створення анімації або додавання деталей до 3D-моделей. Таким чином, WebGL і GLSL надають потужний набір інструментів для створення інтерактивної графіки та анімації у веб-браузерах. Матричні перетворення та відображення текстури є двома важливими техніками, які можна використовувати для додання реалізму та деталізації 3D-моделей. Використовуючи графічний процесор для виконання цих операцій, можна досягти швидкого та ефективного рендерингу складної графіки в реальному часі.

Web Audio API є потужним інструментом для обробки та синтезу звуку у веб-додатках.

Одним з ефективних елементів обробки звуку, доступних в Web Audio API, є шелфовий фільтр низьких частот (lowshelf filter).

Шелфовий фільтр низьких частот є типом фільтра, який може зменшити або підсилити сигнал залежно від його частоти. Він особливо корисний для контролю над низькими частотами в звуковому сигналі. Цей фільтр має параметри, які дозволяють змінювати точку зрізу (початкову точку підсилення або зменшення), підсилення або зниження та коефіцієнт нахилу.

Web Audio API надає можливість створювати шелфові фільтри низьких частот за допомогою AudioContext та AudioNode. Використовуючи Web Audio API, ви можете створити об'єкт BiquadFilterNode та встановити його тип на "lowshelf". Потім ви можете налаштувати параметри фільтра, такі як "frequency" (частота), "gain" (підсилення або зниження) та "Q" (коефіцієнт нахилу).

Наприклад, ось як виглядає код для створення шелфового фільтра низьких частот у Web Audio API:

```
// Створення AudioContext
```

```
const audioContext = new AudioContext();
```

```
// Створення AudioNode, наприклад, AudioBufferSourceNode
```

```
const sourceNode = audioContext.createBufferSource();
```

```
// Завантаження аудіофайлу
```

```
// ...
```

```
// Створення BiquadFilterNode та налаштування типу на "lowshelf" const
lowshelfFilter = audioContext.createBiquadFilter();
lowshelfFilter.type = "lowshelf";

// Налаштування параметрів фільтра
lowshelfFilter.frequency.value = 200; // Частота 200 Гц
lowshelfFilter.gain.value = -10; // Зниження на 10 децибел
lowshelfFilter.Q.value = 1; // Коефіцієнт нахилу

// Підключення AudioNode до шелфового фільтра
sourceNode.connect(lowshelfFilter);
```

Хід роботи

Перед початком написання коду потрібно створити нову гілку в git, за допомогою команди `git switch -c CGW`, перед цим переконавшись базова гілка оновлена з віддаленим сервером, у нашому випадку `origin`. Після створення нової гілки, можна одразу завантажити її на GitHub та зробити так, щоб вона відслідковувала віддалену гілку на `origin`. Для цього потрібно використати команду `git push -u origin head`.

Після того як git налаштований для виконання розрахунково графічної роботи, перейдемо до редактора коду. Для того щоб нанести текстуру на поверхню, спочатку потрібно знайти саму текстуру, для цього можна використати будь який інтернет ресурс, але важливо щоб розміри цієї текстури були в степені двійки, так звана POT текстура. Щоб завантажити її на нашу сторінку, потрібно використати API браузера Image. Важливо вказати `crossOrigin anonymous`, для того щоб змінити налаштування безпеки і картинку можна було завантажити із зовнішнього ресурсу. Підпишемося на подію `onload`, для того, щоб перемалювати канвас вже з новою текстурою.

Для того, щоб передати текстуру до відеокарти нам потрібно зробити дві речі: 1. Передати саму картинку до буферу і в шейдерній програмі використати її як `uniform sampler2D`.

2. Порахувати UV координати та передати їх як атрибут до вершинного шейдеру.

Оскільки топологія поверхні Parabolic Humming-Тор дорівнює 2, UV координатами будуть як раз ці параметри, на основі яких ми будуємо поверхню. Нам залишається лише їх нормалізувати. Z координата в проміжку `-1..1`, тому потрібно її зсунути на 1 та поділити на 2; Кут `d` в межах `0..360`, його достатньо поділити на 360.

Отримавши текстуру та нормалізовані UV координати, їх потрібно передати з JavaScript у WebGL.

Для цього потрібно отримати посилання на контекст WebGL і розташування атрибута та юніформ змінної в програмі GLSL. По-перше, потрібно отримати посилання на контекст WebGL, викликавши метод `getContext` в елементі `canvas HTML`. Далі вам потрібно буде отримати розташування атрибута або уніфікованої змінної в програмі GLSL за допомогою функцій `getAttribLocation` і `getUniformLocation` відповідно. Ці функції приймають програму GLSL та назву атрибута або уніфікованої змінної як аргументи та повертають місце розташування змінної в програмі. Нарешті, можна встановити атрибут та юніформ значення за допомогою відповідної функції WebGL. Для атрибутів можна використовувати функції `vertexAttrib*`, наприклад `vertexAttrib2f` або `vertexAttrib3fv`. Для уніформи можна використовувати функції `uniform*`, наприклад `uniform4fv` або `uniformMatrix4fv`. Ці функції приймають розташування атрибута або уніфікованої змінної та значення як аргументи.

Перейдемо до перетворень переданої точки у GLSL. Нам знадобляться функції `translate()` та `rotate()`. Оскільки їх немає в стандартній бібліотеці, необхідно створити їх самим, за допомогою матриць перетворень та кватерніону (у випадку з `rotate()`).

Приклад матриці `translate`:

```
mat4 translate(mat4 m, vec3 v) {  
    return m * mat4(  
        vec4(1.0, 0.0, 0.0, v.x),  
        vec4(0.0, 1.0, 0.0, v.y),  
        vec4(0.0, 0.0, 1.0, v.z),  
        vec4(0.0, 0.0, 0.0, 1.0)  
    );  
}
```

Як було описано в теоритичній частині, щоб обернути текстуру навколо точки потрібно спочатку змістити позицію на цю точку, повернути її і змістити назад, на від'ємне значення цієї точки.

Далі передамо отримане значення до фрагментного шейдеру. Є велика кількість режимів накладання, у нашому випадку було використано множення. Цей режим змішування часто використовується для створення більш темного, приглушеного ефекту шляхом множення кольорів верхнього шару на кольори нижнього. Це

досягається за рахунок того, що вихідні координати векторів завжди нормалізовані (в межах від 0 до 1).


Після вищеописаних дій, отримаємо накладене зображення текстури з освітленням на нашу поверхню.

Зображення виконання

Oleksandr Snytko

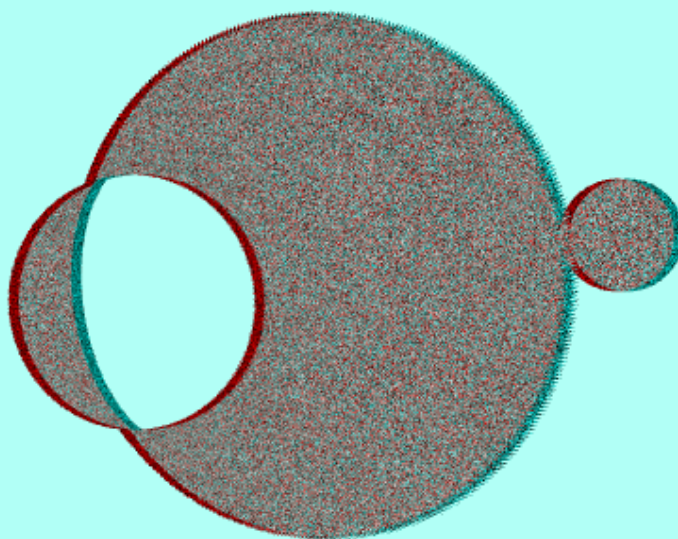
Separation 

Convergence 

FOV 

Near clipping 

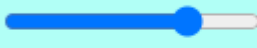
Low shelf filter ☒



Oleksandr Snytko

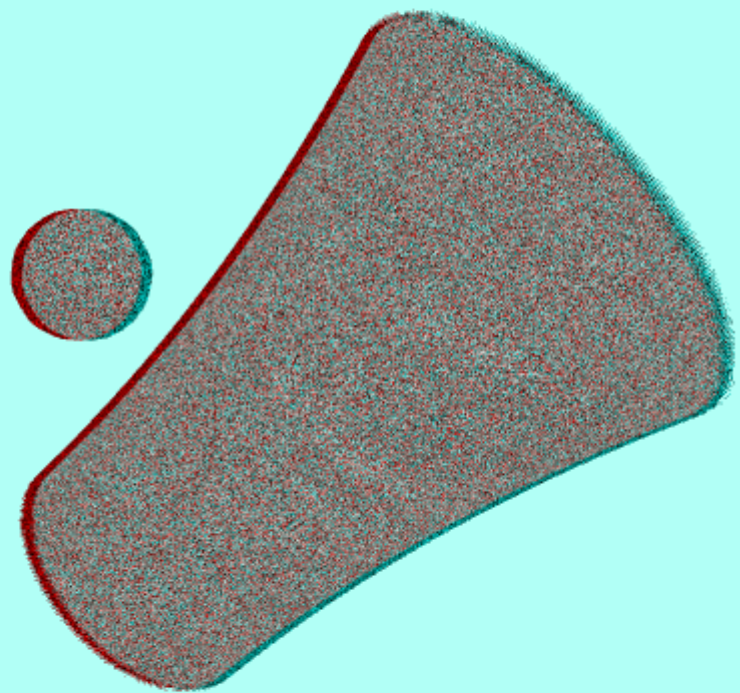
Separation 

Convergence 

FOV 

Near clipping 

Low shelf filter ☒



Oleksandr Snytko

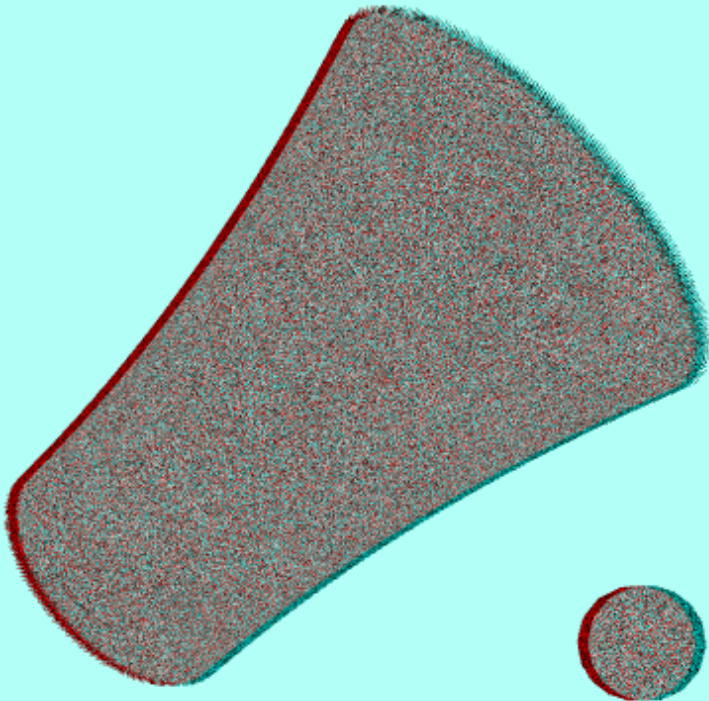
Separation

Convergence

FOV

Near clipping

Low shelf filter ☒



Вихідний код

```
function CreateSurfaceData() {  
  let vertices = [];  
  let uvs = [];  
  
  const l = 1;  
  const r1 = 0.5;  
  const r2 = 4;  
  
  function normUv(b, z) {  
    return [b / 360, z / (2 * l)];  
  }  
  
  function r(a) {  
    a = deg2rad(a);  
    return (r2 - r1) * Math.pow(Math.sin((Math.PI * a) / (4 * l)), 2) * 100 + r1; }  
  
  for (let b = 0; b <= 360; b += 1) {  
    for (let a = 0; a <= 2 * l; a += 0.1) {  
      const x = r(a) * Math.cos(deg2rad(b));  
      const y = r(a) * Math.sin(deg2rad(b));  
      const z = a;  
      vertices.push(x, y, z);  
      uvs.push(...normUv(b, a));  
  
      const a1 = a + 0.2;  
      const b1 = b + 5;  
      const x1 = r(a1) * Math.cos(deg2rad(b1));  
      const y1 = r(a1) * Math.sin(deg2rad(b1));  
      const z1 = a1;  
      vertices.push(x1, y1, z1);  
      uvs.push(...normUv(b1, a1));  
    }  
  
    for (let a = 2*l; a > 0; a -= 0.1) {  
      const x = r(a) * Math.cos(deg2rad(b));  
      const y = r(a) * Math.sin(deg2rad(b));
```

```

const z = a;

vertices.push(x, y, z);
uvs.push(...normUv(b, a));

const a1 = a + 0.2;
const b1 = b + 5;
const x1 = r(a1) * Math.cos(deg2rad(b1));
const y1 = r(a1) * Math.sin(deg2rad(b1));
const z1 = a1;
vertices.push(x1, y1, z1);
uvs.push(...normUv(b1, a1));
}
}

return {vertices, uvs};
}

/* 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.iTexCoord =
gl.getAttribLocation(prog, "texcoord"); shProgram.iModelViewProjectionMatrix =
gl.getUniformLocation(prog, "ModelViewProjectionMatrix");
    shProgram.iColor = gl.getUniformLocation(prog, "color");
    shProgram.iNormal = gl.getAttribLocation(prog, 'normal');
    shProgram.iNormalMatrix = gl.getUniformLocation(prog, 'normalMat'); shProgram.iLightColor =
gl.getUniformLocation(prog, 'lightColor'); shProgram.iShininess = gl.getUniformLocation(prog,
'shininess'); shProgram.iLightPos = gl.getUniformLocation(prog, 'lightPosition'); shProgram.iLightVec
= gl.getUniformLocation(prog, 'lightVec');
    gl.enable(gl.DEPTH_TEST);
}

```