**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО"**

Кафедра цифрових технологій в енергетиці

Розрахунково-графічна робота з дисципліни

“Візуалізація графічної та геометричної інформації”

на тему “Операції з текстурними координатами”

Варіант 10

Виконaв студент групи ТР-32мп

Зуй Денис Олександрович

Київ - 2023

1. **Завдання**

* Нанести текстуру на поверхню з практичного завдання №2.
* Реалізувати масштабування/обертання текстури (координати текстури) масштабування/обертання навколо визначеної користувачем точки – непарні варіанти реалізують масштабування, парні варіанти реалізують обертання.
* Розробити функціонал для переміщення точки вздовж простору поверхні (u,v) за допомогою клавіатури. наприклад клавіші A і D переміщують точку вздовж параметра u, а клавіші W і S переміщують точку вздовж параметра v.

1. **Теорія**

WebGL

WebGL - це JavaScript API, який використовується для створення інтерактивної графіки 2D та 3D безпосередньо у веб-браузері. Воно дозволяє виконувати такі завдання, як текстурування, масштабування, обертання та інші інтерактивні дії з точками за допомогою API, що надає функції для роботи з шейдерами, текстурами і трансформаціями матриць. Шейдери використовуються для налаштування обробки вершин і фрагментів, включаючи координати текстур та трансформації. Текстури створюються і прикріплюються до текстурних одиниць, щоб шейдери могли їх використовувати. Трансформації матриці використовуються для управління положенням, масштабуванням і обертанням моделей.

Шейдери WebGL

У системі WebGL шейдери відіграють ключову роль у процесі візуалізації, дозволяючи детально налаштовувати зображення 3D графіки. Існують два основних типи шейдерів: вершинні і фрагментні. Вершинні шейдери обробляють інформацію кожної вершини моделі, таку як її положення, кольори та текстурні координати, та застосовують трансформації до них. Фрагментні шейдери, відомі також як піксельні шейдери, визначають колір кожного пікселя на основі інформації, отриманої від вершинних шейдерів, включаючи текстури та освітлення. Шейдери написані на мові GLSL та компілюються для використання у програмах шейдерів. Uniforms і Attributes використовуються для передачі даних шейдерам, що дозволяє зовнішнім даним впливати на візуалізацію.

Накладання текстур. Текстурні координати

Текстурування є ключовою технікою у комп’ютерній графіці для створення реалістичних поверхонь, де зображення або текстури накладаються на 3D моделі. У WebGL, кожна вершина об’єкта пов’язана з текстурними координатами, які визначають, як текстура буде нанесена на поверхню об’єкта. Ці координати, які зазвичай представлені парою значень

u і v, використовуються для точного визначення, як текстура розтягується або обертається на моделі, забезпечуючи точне покриття текстурою.

1. **Деталі розробки**

За варіантом мені було надано Astroidal Helicoid. При виконанні практичного завдання №2 було розроблено програму, що виводить поверхню у вигляді суцільних трикутників.

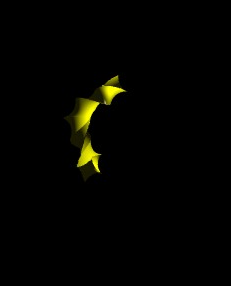


Рисунок 1 - Вигляд поверхні

Я обрав наступне зображення для подальшого виконання розрахунково-графічної роботи. Для підтримки на більшій кількості операційних систем та браузерів було обрано зображення розміром 512х512 пікселів. Формат зображення .jpg.



Рисунок 2 - Зображення текстури

Було накладено текстуру на поверхню. Текстура була застосована до поверхні за допомогою підготовленого буфера текстурних координат, де кожна координата кореспондує з елементом у буфері вершин. Для того, щоб забезпечити можливість обертання текстури, було введено спеціальний uniform, який задає кут обертання. У шейдерній програмі, для встановлення кольору кожного пікселя об'єкта з накладеною текстурою, застосовується функція texture2D(). Ця функція використовує два аргументи: об'єкт типу sampler2D, який містить інформацію про текстуру, та текстурну координату, що визначає, яка частина текстури буде відображена.

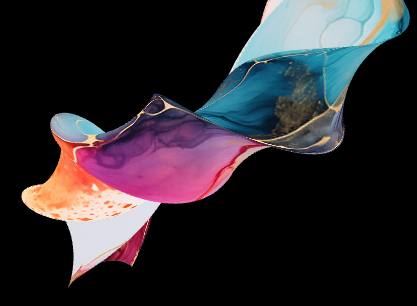


Рисунок 3 - Текстура накладена на поверхню

Для візуалізації точки трансформації текстури було створено новий екземпляр класу Model, який представляє собою сферу, розташовану на поверхні. Цей об'єкт дозволяє наочно показати локацію в просторі, довкола якої будуть відбуватись трансформації текстури, наприклад, обертання або масштабування. Сама сфера слугує графічним зображенням точки, навколо якої буде здійснюватись трансформація, надаючи зручний спосіб візуальної індикації для розуміння і налагодження процесу трансформації текстури.

Рисунок 4 - Точка на текстурованій поверхні

Згідно варіанту було імплементовано обертання текстури відносно точки на поверхні.

1. **Інструкції користувача**

Фігуру можна обертати відносно центру затиснувши ліву клавішу миші та потягнувши в сторону бажаного обертання.



Рисунок 5 - Вигляд фігури до та після обертання

Переміщувати точку відносно якої здійснюється обертання можна за допомогою клавіш WASD. Кожне натискання переміщує точку по поверхні на визначений крок. Переміщення здійснюється до визначеної межі.

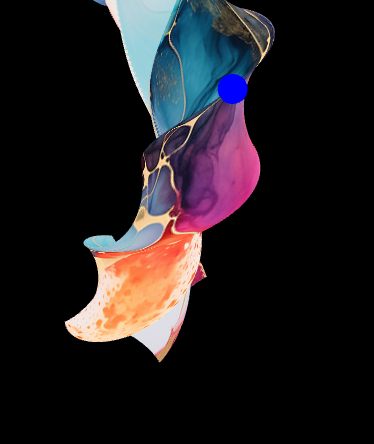
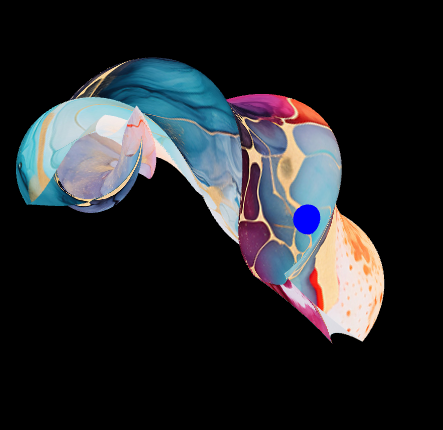


Рисунок 6 - Демонстрація переміщення точки відносно нерухомої фігури

За допомогою слайдера можливо регулювати кут обертання відносно уявної точки на поверхні. При зміні кута обертання, якщо він не є нульовим, спостерігається зміщення текстури. Це відбувається тому, що обертання проводиться відносно точки, розташованої на поверхні, яка відповідає певній текстурній координаті. Таким чином, при переміщенні цієї точки змінюється не тільки положення сфери, а й положення і орієнтація текстури на об'єкті, надаючи можливість динамічно контролювати і візуально наблюдати за процесом обертання текстури.

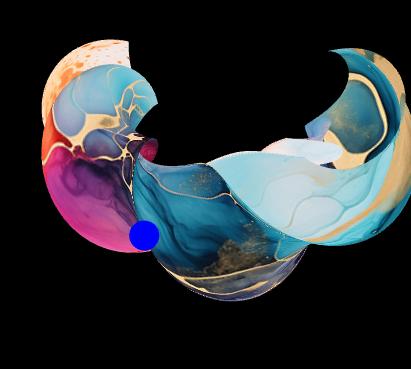


Рисунок 7 - Демонстрація зміни кута обертання текстури

При перезавантаженні сторінки обертання фігури, а також значення кута обертання буде встановлено по замовчуванню. Положення точки відносно поверхні також буде скинуто до значення по замовчуванню.

1. **Код програми**
2. // Constructor
3. function ShaderProgram(name, program) {
4. this.name = name;
5. this.prog = program;
6. // Location of the attribute variable in the shader program.
7. this.iAttribVertex = -1;
8. // Location of the uniform specifying a color for the primitive.
9. this.iColor = -1;
10. // Location of the uniform matrix representing the combined transformation.
11. this.iModelViewProjectionMatrix = -1;
12. this.Use = function () {
13. gl.useProgram(this.prog);
14. }
15. }
16. /\* Draws a colored cube, along with a set of coordinate axes.
17. \* (Note that the use of the above drawPrimitive function is not an efficient
18. \* way to draw with WebGL.  Here, the geometry is so simple that it doesn't matter.)
19. \*/
20. function draw() {
21. gl.clearColor(0, 0, 0, 1);
22. gl.clear(gl.COLOR\_BUFFER\_BIT | gl.DEPTH\_BUFFER\_BIT);
23. /\* Set the values of the projection transformation \*/
24. // let projection = m4.perspective(Math.PI / 8, 1, 8, 12);
25. let projection = m4.orthographic(-2, 2, -2, 2, -2, 2);
26. /\* Get the view matrix from the SimpleRotator object.\*/
27. let modelView = spaceball.getViewMatrix();
28. let rotateToPointZero = m4.axisRotation([0.707, 0.707, 0], 0.7);
29. let translateToPointZero = m4.translation(0, 0, -0);
30. let matAccum0 = m4.multiply(rotateToPointZero, modelView);
31. let matAccum1 = m4.multiply(translateToPointZero, matAccum0);
32. /\* Multiply the projection matrix times the modelview matrix to give the
33. combined transformation matrix, and send that to the shader program. \*/
34. let modelViewProjection = m4.multiply(projection, matAccum1);
35. gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false, modelViewProjection);
36. /\* Draw the six faces of a cube, with different colors. \*/
37. gl.uniform4fv(shProgram.iColor, [1, 1, 0, 1]);
38. gl.uniform3fv(shProgram.iPosition, [Math.cos(Date.now() \* 0.001), Math.sin(Date.now() \* 0.001), 1]);
39. gl.uniform2fv(shProgram.iP, pos);
40. gl.uniform1f(shProgram.iR, document.getElementById('r').value);
41. surface.Draw();
42. gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false, m4.multiply(modelViewProjection,
43. m4.translation(...get3D(pos[0] \* 2 \* Math.PI, pos[1] \* 2 \* Math.PI))));
44. gl.uniform4fv(shProgram.iColor, [1, 1, 0, -1]);
45. sphere.Draw()
46. }
47. function animate() {
48. draw()
49. window.requestAnimationFrame(animate)
50. }
51. function CreateSurfaceData() {
52. const numUSteps = 150;
53. const numVSteps = 150;
54. const vertices = [];
55. for (let i = 0; i <= numUSteps; i++) {
56. for (let j = 0; j <= numVSteps; j++) {
57. const u = i \* (2 \* Math.PI) / numUSteps;
58. const v = j \* (2 \* Math.PI) / numVSteps;
59. const uInc = 1 \* (2 \* Math.PI) / numUSteps;
60. const vInc = 1 \* (2 \* Math.PI) / numVSteps;
61. let v1 = get3D(u, v);
62. let v2 = get3D(u + uInc, v);
63. let v3 = get3D(u, v + vInc);
64. let v4 = get3D(u + uInc, v + vInc);
65. vertices.push(...v1, ...v2, ...v3, ...v3, ...v2, ...v4);
66. }
67. }
68. return vertices;
69. }