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: «Операції над координатами текстури»

Варіант №14

Виконав:

студент 5-го курсу

групи ТР-22мп ІАТЕ

Кузяк Н. І.

Перевірив: Демчишин А.А.

Київ-2022  
 **Постановка задачі**

1. Нанести текстуру на поверхню з практичного завдання №2.
2. Реалізувати масштабування текстури (координати текстури) масштабування/обертання навколо визначеної користувачем точки – непарні варіанти реалізують масштабування, парні варіанти реалізують обертання.
3. Повинна бути можливість переміщати точку 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.

Поставлене завдання полягає в тому, щоб відобразити текстуру на поверхні та реалізувати масштабування та обертання текстури навколо вказаної користувачем точки на поверхні. Користувач повинен мати можливість переміщати точку по поверхні за допомогою елементів керування на клавіатурі.

У комп’ютерній графіці відображення текстури — це техніка, яка використовується для додання реалізму та деталізації 3D-моделей шляхом нанесення зображення, відомого як текстура, на поверхню моделі. Текстури можна використовувати для створення зовнішнього вигляду різних матеріалів, таких як дерево, камінь або метал, і їх можна масштабувати та обертати для досягнення різних ефектів.

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

**Теоритичні відомості**

WebGL — програмна бібліотека для JavaScript призначена для візуалізації інтерактивної тривимірної графіки та двовимірної графіки в межах сумісності веб-браузера без використання плагінів. WebGL приносить у веб тривимірну графіку, вводячи API, який побудований на основі OpenGL ES 2.0, що дозволяє його використовувати в елементах canvas HTML5.

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.

Щоб повернути текстуру навколо 2D-точки в WebGL за допомогою GLSL потрібно:

1. Визначити центральну точку обертання як двовимірний вектор (центр).
2. Перемістити текстуру на обернену центральної точки, використовуючи матрицю translate. Це перемістить центр текстури до початку координат.
3. Обернути текстуру навколо початку координат за допомогою матриці обертання.
4. Перемістити текстуру назад у вихідне положення, додавши центральну точку за допомогою іншої матриці перекладу.
5. Об’єднайти матриці перетворення та матрицю обертання, щоб створити єдину матрицю перетворення.
6. Використати матрицю перетворення, щоб перетворити UV-координати текстури.

В GLSL реалізація цього буде виглядати наступним чином:

// concatenate the translation and rotation matrices

mat3 transform = translate \* rotate \* translateBack;

// transform the UV coordinates

vec2 uv = transform \* vec3(originalUV, 1.0);

Нанесення текстури – це техніка, яка зазвичай використовується в WebGL для додання реалізму та деталізації 3D-моделей. Текстура — це зображення, яке наноситься 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. Маніпулюючи UV-координатами, можна досягти різних ефектів, таких як масштабування, обертання або зміщення текстури. Це може бути корисним для створення анімації або додавання деталей до 3D-моделей. Крім того, кілька текстур можна комбінувати за допомогою техніки змішування для створення більш складних зовнішніх зображень.

**Хід роботи**

Перед початком написання коду потрібно створити нову гілку в git. Після створення нової гілки, можна одразу завантажити її на GitHub та зробити так, щоб вона відслідковувала віддалену гілку на origin.

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-Top дорівнює 2, UV координатами будуть як раз ці параметри, на основі яких ми будуємо поверхню. Нам залишається лише їх нормалізувати. Кут u в межах 0..180, його достатньо поділити на 180, так як і кут v в межах 0…360, можна 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 випидку 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)

);

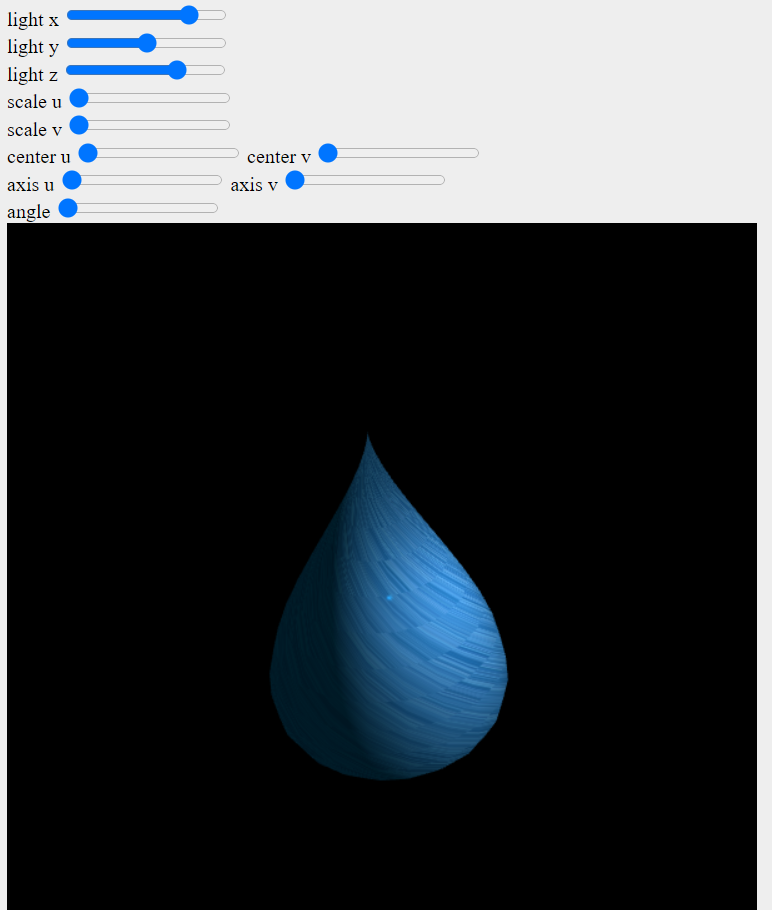
}

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.

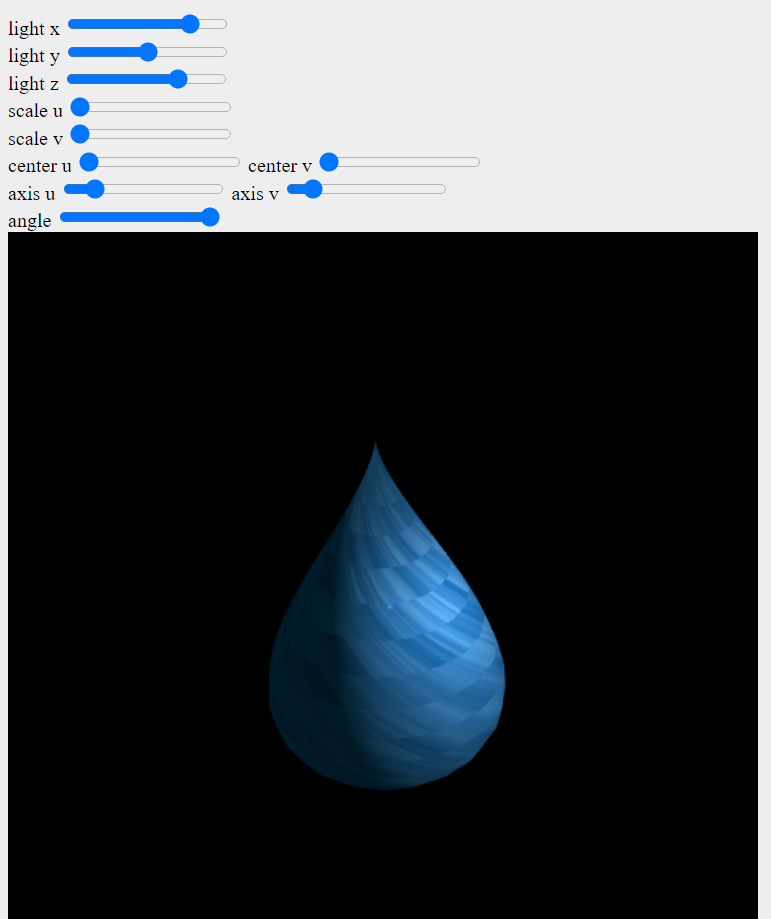
Далі передамо отримане значення до фрагментного шейдеру. Є велика кількисть режимів накладання, у нашому випадку було використано множення. Цей режим змішування часто використовується для створення більш темного, приглушеного ефекту шляхом множення кольорів верхнього шару на кольори нижнього. Це досягається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

****

Текстура нанесена на поверхню

Текстура обернена відносно точки

**Шейдер**

// Vertex shader

const vertexShaderSource = `

attribute vec3 vertex;

attribute vec2 texcoord;

uniform mat4 ModelViewProjectionMatrix;

uniform vec2 texScale;

uniform vec2 texCenter; //rotationCenter

uniform vec2 textureAxis; // rotation axis

uniform float textureRotAngleDeg; // rotation angle in degrees

varying vec3 vVertex;

varying vec3 vVertexPosition;

varying vec2 vTexcoord;

mat4 identity4() {

  return mat4(

    1.0, 0.0, 0.0, 0.0,

    0.0, 1.0, 0.0, 0.0,

    0.0, 0.0, 1.0, 0.0,

    0.0, 0.0, 0.0, 1.0

  );

}

mat4 genScaleMat(vec3 v) {

  return mat4(

    vec4(v.x, 0.0, 0.0, 0.0),

    vec4(0.0, v.y, 0.0, 0.0),

    vec4(0.0, 0.0, v.z, 0.0),

    vec4(0.0, 0.0, 0.0, 1.0)

  );

}

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)

  );

}

mat4 genTranslateMat(vec3 v) {

  return 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)

  );

}

mat4 rotate(mat4 m, vec3 axis, float angle) {

  // Normalize the axis vector

  axis = normalize(axis);

  // Convert the angle to radians

  float radians = radians(angle);

  // Calculate the sine and cosine of the angle

  float c = cos(radians / 2.0);

  float s = sin(radians / 2.0);

  // Create a quaternion from the axis and angle

  vec4 q = vec4(s \* axis, c);

  // Convert the quaternion to a rotation matrix

  return m \* mat4(

    1.0 - 2.0 \* q.y \* q.y - 2.0 \* q.z \* q.z,

    2.0 \* q.x \* q.y - 2.0 \* q.z \* q.w,

    2.0 \* q.x \* q.z + 2.0 \* q.y \* q.w,

    0.0,

    2.0 \* q.x \* q.y + 2.0 \* q.z \* q.w,

    1.0 - 2.0 \* q.x \* q.x - 2.0 \* q.z \* q.z,

    2.0 \* q.y \* q.z - 2.0 \* q.x \* q.w,

    0.0,

    2.0 \* q.x \* q.z - 2.0 \* q.y \* q.w,

    2.0 \* q.y \* q.z + 2.0 \* q.x \* q.w,

    1.0 - 2.0 \* q.x \* q.x - 2.0 \* q.y \* q.y,

    0.0,

    0.0,

    0.0,

    0.0,

    1.0

  );

}

vec2 scaleWithPoint(vec2 point, vec2 pivot, vec2 scale) {

    mat4 tran = genTranslateMat(vec3(pivot, 0.));

    mat4 tranBack = genTranslateMat(-vec3(pivot, 0.));

    mat4 scaleMat = genScaleMat(vec3(scale, 0.));

    vec4 point4 = vec4(point, 0., 0.);

    vec4 tr = tran \* point4;

    vec4 sc = tr \* scaleMat;

    vec4 trb = sc \* tranBack;

    return vec2(trb);

}

vec2 rotateWithPoint(vec2 point, vec2 pivot, vec2 axis, float angle\_deg) {

  mat4 tr = translate(identity4(), vec3(pivot, 0.0));

  mat4 rt = rotate(tr, vec3(axis, 1.0), angle\_deg);

  mat4 trb = translate(rt, vec3(-pivot, 0.0));

  return vec2(trb \* vec4(point, 0.0, 0.0));

}

void main() {

    vec4 position = ModelViewProjectionMatrix \* vec4(vertex,1.0);

    vVertex = vertex;

    vVertexPosition = vec3(position) / position.w;

    vec2 point = vec2(0, 0);

    vec2 tex\_scaled = scaleWithPoint(texcoord, point, texScale);

    vTexcoord  = rotateWithPoint(tex\_scaled, texCenter, textureAxis, textureRotAngleDeg);

    gl\_Position = position;

}`;

// Fragment shader

const fragmentShaderSource = `

#ifdef GL\_FRAGMENT\_PRECISION\_HIGH

   precision highp float;

#else

   precision mediump float;

#endif

varying vec3 vVertex;

varying vec3 vVertexPosition;

varying vec2 vTexcoord;

uniform mat4 normalMatrix;

uniform vec3 lightPosition;

uniform sampler2D uTexture;

vec3 calculate\_light(vec3 position) {

    vec3 shape\_color = vec3(1., 1., 1.);

    vec3 light\_color = vec3(0., 1., 1.);

    vec3 ambient = light\_color \* 0.2;

    vec3 normal = normalize(vec3(normalMatrix \* vec4(vVertex, 0.)));

    vec3 light\_direction = normalize(lightPosition - position);

    float dot\_light = max(dot(normal, light\_direction), 0.0);

    vec3 diffuse = shape\_color \* dot\_light;

    float specular\_strength = 0.5;

    float spec = 0.;

    if (dot\_light > 0.) {

        vec3 view\_dir = normalize(-position);

        vec3 reflect\_dir = reflect(-light\_direction, normal);

        float spec\_angle = max(dot(view\_dir, reflect\_dir), 0.0);

        spec = pow(spec\_angle, 32.);

    }

    vec3 specular = specular\_strength \* spec \* light\_color;

    return ambient + diffuse + specular;

}

void main() {

    vec4 texture = texture2D(uTexture, vTexcoord);

    vec4 lightColor = vec4(calculate\_light(vVertexPosition), 1.);

    gl\_FragColor = texture \* lightColor;

}`;