міністерство Освіти й науки України

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Кафедра автоматизації проектування енергетичних процесів і систем

Звіт
«Візуалізація графічної та геометричної інформації»

Розрахунково-графічна робота
Варіант №23

Виконав:
студент 1-го курсу групи ТР-21мп
НН ІАТЕ Савонік Ю.В.
Перевірив: Демчишин А.А.

Київ-2022

Опис завдання:

Необхідно нанести текстуру на поверхню, що задана наступною формулою:

$$\begin{cases} x = \left(\frac{(|z|-h)^2}{2p}\right) * \cos(\alpha) \\ y = \left(\frac{(|z|-h)^2}{2p}\right) * \sin(\alpha) \\ z = z \end{cases}$$, де 2*p – радіус основи.

Формула 1. Явне задання поверхні.

Для текстури необхідно реалізувати масштабування текстури (координати текстури) та масштабування навколо визначеної користувачем точки.
Також повинна бути можливість переміщати точку вздовж простору поверхні (u,v).
Підготувати звіт, з наступним змістом:
- титульна сторінка;
- розділ з описом завдання (1 сторінка);
- розділ з описом теорії (2 сторінки);
- розділ з описом деталей впровадження (2 сторінки);
- розділ інструкції користувача зі скріншотами (2 сторінки);
- зразок вихідного коду (2 сторінки).

Опис теорії:

Відображення текстури — це техніка визначення унікального кольору для кожного фрагмента, який утворює трикутник. Кольори походять із відображення.

Огляд обладнання

Відображення текстур є фундаментальним для створення реалістичних візуалізацій, а апаратне забезпечення GPU містить блоки текстури для підтримки відображення текстури.

Блок текстури виконує обробку для відображення текстури. Об'єкт текстури зберігає дані, необхідні для відображення текстури. В загальному, можна створити багато текстурних об'єктів, але кількість текстурних одиниць у графічному процесорі визначає, скільки текстурних карт можна використовувати одночасно в шейдерній програмі.

Відображення текстури відображає місце на 2D-зображенні на місце на 3D-трикутнику. WebGL використовує координати текстури для виконання цього відображення. Як і в багатьох інших аспектах графіки, координати текстури є відсотками. Позначення координат текстури використовує (u,v) для представлення розташування зображення. Компонент u – це відсоток від ширини зображення, а компонент v – відсоток від висоти зображення. Кожній вершині моделі призначається координата текстури, і ці координати інтерполюються по всій поверхні трикутника, щоб визначити унікальне
розташування на зображенні для кожного фрагмента трикутника.

Основні кроки для створення текстурного відображення такі:

При побудові моделі:
Знайти зображення для накладання текстури, що не порушує CORS політику.
Призначити відповідну координату текстури (u,v) кожній вершині трикутника.
Попередньо модифікувати проект в JavaScript для візуалізації полотна:
Завантажити зображення текстурної карти з сервера.
Створити і заповнити об'єкт текстури GPU зображенням.
Встановити параметри, які керують використанням зображення карти текстури.
Отримати розташування однорідної змінної Sample2D із програми шейдера.
Налаштовувати JavaScript кожного разу, коли модель відображається за допомогою карти текстури:
Прив'язати об'єкт текстури до блоку текстури
Прив'язати блок текстури до однорідної змінної шейдера.
Шейдерна програма
У вершинному шейдері створення змінної, яка буде інтерполювати координати текстури по поверхні трикутника.
У фрагментному шейдері використовуються координати текстури, щоб знайти колір із зображення текстури.

Деталі впровадження:

Вершинний шейдер:

Спочатку в вершинному шейдері необхідно було додати новий атрибут типу vec2, щоб отримувати координати текстури. Так як текстурування відбувається в фрагментному шейдері, то було додано змінну, для передачі в фрагментний шейдер, з ідентифікатором varying. Окрім того, за варіантом, для масштабування текстури було додано змінну з ідентифікаторо uniform для, власне, значення масштабу (типу float) та точки, навколо якої буде проводитись масштабування (типу vec2). Для обчислення матриці масштабування створили функцію scale, в яку передається масштаб і, перед передачею, координати текстури в фрагментний шейдер, спочатку був зроблений масштаб за наступним алгоритмом:

1) Зсунення координати текстури на значення точки масштабування (від координати текстури відняли координати точки масштабування).
2) Дія масштабування (зсунену точку домножили на матрицю масштабува-ння).
3) Відмасштабовану точку зсунули на її початкове місце (до відмасштабова-ної точки додали точку масштабування).

Фрагментний шейдер:

В фрагментому шейдері, був створений колір текстури через виклик функції texture2D в яку переданий поточний sampler (переданий через ідентифікатор uniform) та інтерпольовані координати текстури.

JavaScript:

В програмі JavaScript, спочатку була створена функція LoadTexture, в якій створюється текстура, завдяки вбудованій в WebGL функції gl.createTexture. Далі було зв'язано її через іншу вбудовану функцію bindTexture та, через texParameteri, було налаштовано фільтр збільшення текстури та фільтр мініміза-ції текстури. Після цього призначили текстурі зображення, взяте з інтернету, що не порушує cors політику. Виклик функції відбувається при ініціалізації контексту WebGL.

Через зміну шейдерної програми було додано зміні, що пов'язують значення (масштаб, sampler, координати текстури та точка масштабування) в коді JavaScript та в шейдері. Для масштабу створили змінну на початку JavaScript коду для швидкого і зручного редагування. Для точки масштабування було створено дві змінні: одна з них бере значення з повзунка в HTML, а інша трансформує це значення в uv координати. Для Samplery передано значення 0, що означає, що всі значення з пов'язаної текстури наносяться повністю, без змін.

Для розрахунку координат текстури була створена функція CreateTextureCoordinates. Так як поверхня залежить від двох змінних (куту та висоти), при чому кут змінюється від 0 до 2π, а висота від –height до height, то для того, аби отримати координати текстури для кожної вершини достатньо скористатись наступною формулою:

$$\begin{cases} u_t = \frac{u_s}{2\pi} \\ v_t = \frac{(v_{s+height})}{2\text{height}} \end{cases}$$, де u_t та v_t- координати текстури,

u_s та v_s – координати для

обчислення поверхні.

Формула 2. Обрахунок координати текстури.

Саме результат вищевказаної формули 2 передається в атрибут координатних текстур.

Для відображення місцезнаходження точки масштабування створюється, по аналогії з поверхнею, об’єкт Model, яка відмальовує відрізок з координатами (pointScale) та (2*pointScale), де pointScale – точка отримана з значення повзунка, що було обчислене по формулі 1, відповідно.

HTML:

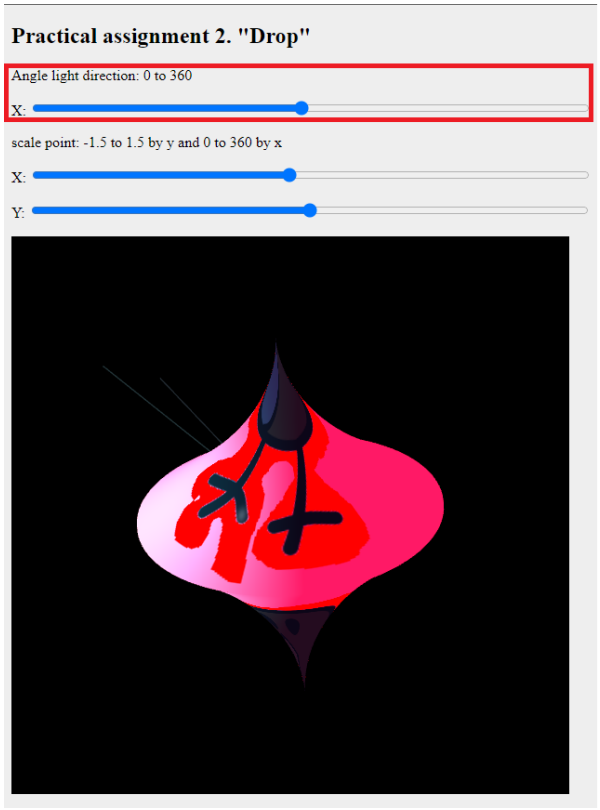
Для зміни точки машстабування користувачем було додано повзунки, що змінюються від 0 до 360 та від -1.5 до 1.5, відповідно. При цьому, при змінні значень повзунків, так само, як і при змінні значення точки з якої направляється світло, зворотнім викликом слугує функція Redraw, що заново буфферизує координати вершин, нормалей та текстури і викликається функція відмальовки поверхні.

Для того, щоб користувач міг бачити початкове зображення текстури – знизу було додано елемент зображення.

Інструкція користувача:

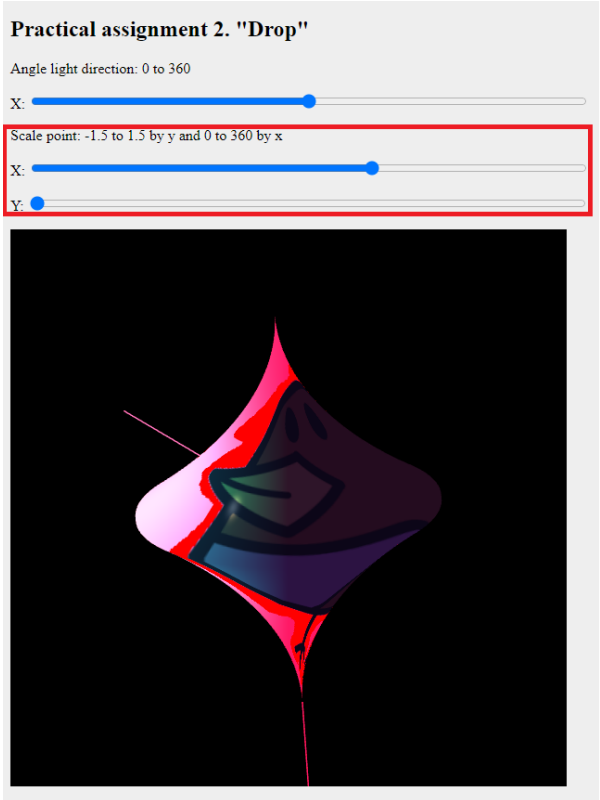
Користувач має можливість:

1) Змінювати вектор напрямку направленого світла завдяки повзунку:



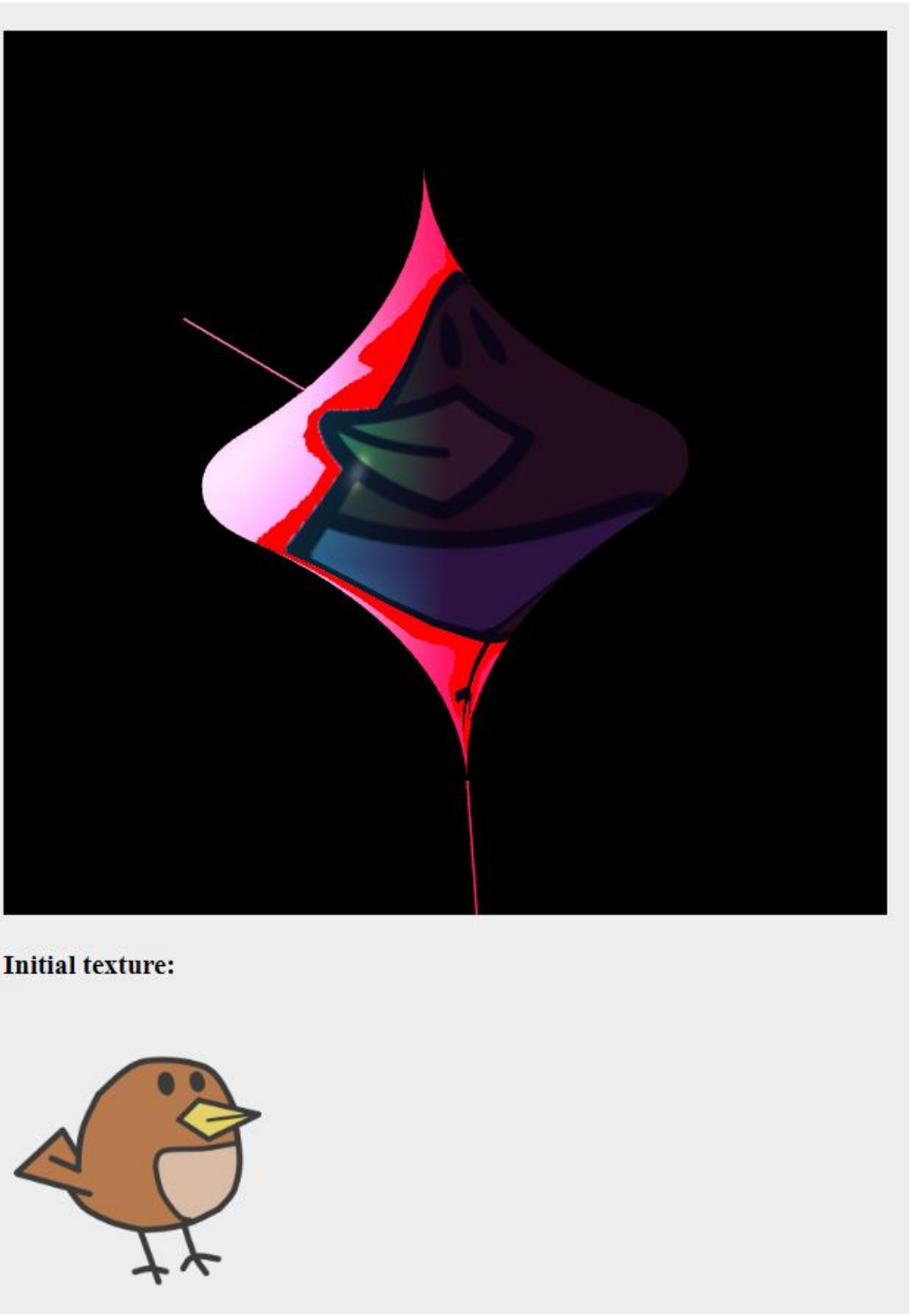
Для того, щоб змінити вектор напрямку, необхідно використовувати повзунок, що виділений на скріншоті. При змінені розташування повзунка, вектор напрямку рухається по колу. Для відображення напрямку світла використовується лінія.

2) Змінювати точку масштабування:



Для того, щоб змінити точку масштабування, необхідно використовувати повзунки, що виділені на скріншоті. При змінені розташування повзунків, точка масштабування рухається в заданих границях.

Також, крім цього підполем відображається початкова текстура, для розуміння користувачем, яке зображення використовується, в якості текстури:



Вихідний код:

JavaScript:

```
function draw() {
```

```
    gl.clearColor(0,0,0,1);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    /* Set the values of the projection transformation */
    let orthographic = m4.orthographic(-2, 2, -2,2, 8, 12);
    /* Get the view matrix from the SimpleRotator object.*/
    let modelView = spaceball.getViewMatrix();
    let rotateToPointZero = m4.axisRotation([0.707,0.707,0], 0.7);
    let translateToPointZero = m4.translation(0,0,-10);
    let matAccum0 = m4.multiply(rotateToPointZero, modelView );
    let matAccum1 = m4.multiply(translateToPointZero, matAccum0 );
    const modelviewInv = m4.inverse(matAccum1, new Float32Array(16));
    const normalMatrix = m4.transpose(modelviewInv, new Float32Array(16));
    /* Multiply the projection matrix times the modelview matrix to give the
       combined transformation matrix, and send that to the shader program. */
    let modelViewProjection = m4.multiply(orthographic, matAccum1 );
    gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false,modelViewProjection );
    gl.uniformMatrix4fv(shProgram.iNormalMatrix, false, normalMatrix);
    let angle = Array.from(lightPositionEl.getElementsByTagName('input')).map(el =>
+el.value)[0];
    lightPos = GetCirclePoint(angle);
    gl.uniform3fv(shProgram.iLightPos, lightPos);
    const scaleWorldPosition =
Array.from(scalePositionEl.getElementsByTagName('input')).map(el => +el.value);
    let scalePos = [];
    scalePos[0] = GetRadiansFromDegree(scaleWorldPosition[0]) / (2 * Math.PI);
    scalePos[1] = (scaleWorldPosition[1] + height) / (2*height);
    gl.uniform2fv(shProgram.iScalePoint, scalePos);
    angle = GetRadiansFromDegree(scaleWorldPosition[0]);
    let currentZ = GetCurrentZPosition(scaleWorldPosition[1]);
    scalePosition = [currentZ * Math.cos(angle), scaleWorldPosition[1], currentZ *
Math.sin(angle)]
    gl.uniform1f(shProgram.iShininess, 80.0);
    gl.uniform1f(shProgram.iAmbientCoefficient, 1);
    gl.uniform1f(shProgram.iDiffuseCoefficient, 1);
    gl.uniform1f(shProgram.iSpecularCoefficient, 1);
    gl.uniform3fv(shProgram.iAmbientColor, [0.2, 0.1, 0.4]);
    gl.uniform3fv(shProgram.iDiffuseColor, [0, 0.8, 0.8]);
    gl.uniform3fv(shProgram.iSpecularColor, [1.0, 1.0, 1.0]);
    gl.uniform1i(shProgram.iTMU, 0);
    gl.uniform1f(shProgram.iScaleValue, scaleValue);
    gl.uniform4fv(shProgram.iColor, [0,0,0.8,1] );
    surface.Draw(gl.TRIANGLE_STRIP);
    light.Draw(gl.LINES);
    scale.Draw(gl.LINES);
}
```

```
function LoadTexture()
{
    var 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);
```

```
    var image = new Image();
    image.crossOrigin = 'anonymous';
    image.src = "https://upload.wikimedia.org/wikipedia/commons/6/63/Icon_Bird_512x512.png";
    image.addEventListener('load', () => {
        gl.bindTexture(gl.TEXTURE_2D, texture);
        gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
        draw();
    }
    );
}
```

Shader:

```
// Vertex shader
const vertexShaderSource = `
attribute vec3 vertex;
attribute vec3 normal;
attribute vec2 texCoord;

uniform mat4 ModelViewProjectionMatrix, normalMat;
uniform vec2 scalePoint;
uniform float scaleValue;

varying vec3 normalInterp;
varying vec3 vertPos;
varying vec2 textureInterpolate;

mat4 scale(float value)
{
  mat4 scaleMatrix;

  for(int i = 0; i < 4;i++)
  {
    for(int j = 0; j < 4;j++)
    {
      if(i != j)
      {
        scaleMatrix[i][j] = 0.0;
      }
      else
      {
        scaleMatrix[i][j] = value;
      }
    }
  }
  scaleMatrix[3][3] = 1.0;
  return scaleMatrix;
}

void main() {
    mat4 scale = scale(scaleValue);

    vec4 vertPos4 = ModelViewProjectionMatrix * vec4(vertex, 1.0);
    vertPos = vec3(vertPos4) / vertPos4.w;
    normalInterp = vec3(normalMat * vec4(normal, 0.0));
```

```
    vec4 texCoordTranslated = vec4(texCoord -scalePoint ,0.0,0.0);
    vec4 texCoordScaled = texCoordTranslated * scale;
    vec4 texCoordTranslatedToBack = vec4(texCoord + scalePoint ,0.0,0.0);
    textureInterpolate = texCoordTranslatedToBack.xy;
    gl_Position = vertPos4;

}`;
// Fragment shader
const fragmentShaderSource = `
#ifdef GL_FRAGMENT_PRECISION_HIGH
   precision highp float;
#else
   precision mediump float;
#endif
varying vec4 color;
precision mediump float;
uniform float ambientCoefficient;   // Ambient reflection coefficient
uniform float diffuseCoefficient;   // Diffuse reflection coefficient
uniform float specularCoefficient;   // Specular reflection coefficient
uniform float shininess; // Shininess
uniform vec3 ambientColor;
uniform vec3 diffuseColor;
uniform vec3 specularColor;
uniform vec3 lightPosition;
uniform sampler2D sampler;
varying vec2 textureInterpolate;
varying vec3 normalInterp;  // Surface normal
varying vec3 vertPos;       // Vertex position
void main() {
  vec3 vNormal = normalize(normalInterp);
  vec3 light = normalize(lightPosition);
  float dotProduct = max(dot(vNormal, light), 0.0);
  float specular = 0.0;
  if(dotProduct > 0.0) {
    vec3 reflect = reflect(-light, vNormal);
    vec3 n_vertPos = normalize(-vertPos);
    float specAngle = max(dot(reflect, n_vertPos), 0.0);
    specular = pow(specAngle, shininess);
  }
  vec4 lightColor = vec4(ambientCoefficient * ambientColor
                + diffuseCoefficient * dotProduct * diffuseColor
                + specularCoefficient * specular * specularColor, 1.0);
  vec4 texColor = texture2D(sampler, textureInterpolate);
  gl_FragColor = texColor * lightColor;
}`;
```