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

Навчально-науковий інститут атомної і теплової енергетики
Кафедра цифрових технологій в енергетиці

ЗВІТ

з розрахункової графічної роботи

з дисципліни «Візуалізація графічної та геометричної інформації»

Тема:
«Розрахунки та робота з графікою (Операції з координатами текстури)»

Варіант № 7 – масштабування текстури (координати текстури) масштабування

Виконав:
студент групи ТР-52мп
Пархоменко Дем’ян
Перевірив:
к.т.н., Демчишин А. А.

Дата здачі _____

Київ – 2025

Постановка задачі

У рамках даного кредитного модуля попередніми роботами вже було реалізовано аналітичну поверхню типу «Parabolic Humming-Top» (параболічна дзиґа), виконано її рендеринг у вигляді трикутної сітки за допомогою WebGL та організовано освітлення Фонга з рухомим точковим джерелом світла, а також застосовано дифузну, спекулярну та нормал-текстури.

У цьому звіті розглядається завершальний етап, що стосується саме операцій над текстурними координатами. Завдання полягає у тому, щоб, використовуючи вже наявне текстурне відображення поверхні, реалізувати масштабування та обертання текстури навколо точки, яка задається користувачем у параметричному просторі поверхні. Центральна ідея полягає у тому, що для кожної точки поверхні з параметрами (u, v) існують відповідні текстурні координати, і над ними потрібно виконувати афінні перетворення, змінюючи сприйняття текстури без зміни геометрії.

Потрібно використати текстурне відображення, отримане в контрольному завданні, не змінюючи базову схему прив’язки текстури до параметрів поверхні. Поверх цього відображення необхідно додати можливість масштабування текстурних координат відносно фіксованої точки у просторі (u, v), а також можливість обертання текстурного шаблону навколо цієї точки на заданий кут.

Особливістю завдання є те, що ця точка в просторі параметрів повинна задаватися та змінюватися користувачем у реальному часі за допомогою клавіатури. Координата u має змінюватися при натисканні клавіш A та D, а координата v – при натисканні клавіш W та S. Таким чином, користувач отримує можливість пересувати центр перетворення текстури по поверхні, а ефект масштабування та обертання відбувається саме навколо обраної точки.

Результатом виконання роботи має стати інтерактивний WebGL-додаток, у якому користувач може обертати сцену мишею, змінювати детальність сітки поверхні, а також експериментально спостерігати, як змінюється розташування, масштаб і орієнтація текстури відносно геометрії при пересуванні точки центру та коригуванні параметрів трансформації.

2. Теоретичні відомості

У розділі викладено математичні основи роботи з текстурами: від параметризації поверхонь та розрахунку освітлення до використання тангенціального базису та опису координатних перетворень.

2.1. Параметричний опис поверхні «Parabolic Humming-Top»

Розглядається поверхня обертання, створена шляхом обертання параболічного профілю навколо вертикальної осі. У роботі як вертикальну вісь зручно використовувати вісь Y. Параметричні рівняння поверхні мають вигляд

$$x(u,v) = r(y)cos\beta,\ z(u,v) = r(y)sin\beta,\ y(u,v) = y,$$

де параметр βпов’язується з параметром $u \in [0,1]$співвідношенням $\beta = 2\pi u$, а параметр yзмінюється від $-h$до hі задається через параметр $v \in [0,1]$формулою

$y = -h + 2hv$. Функція радіуса в горизонтальній площині виражається параболою

$$r(y) = \frac{(h)^2}{2p},$$

де h визначає характерну висоту половини фігури, а pє параметром параболи. Така параметризація створює характерну форму дзиґи: при значеннях y поблизу полюсів радіус малий, а в околі «екватора» досягає максимуму.

2.2. Текстурні координати та їх трансформації

У WebGL текстурні координати напряму пов'язують вершину поверхні з точкою на зображенні. Для поверхні, параметризованої через u та v, природно брати текстурні координати як (v). У такому разі кожна точка поверхні отримує відповідний фрагмент текстури.

Коли виникає потреба виконувати операції масштабування та обертання текстури відносно певної точки у текстурному просторі, необхідний математичний опис цих перетворень. Спочатку введемо вектор текстурних координат $\vec{t} = (u, v)$. Позначимо через $\vec{c} = (u_0, v_0)$координати центру, навколо якого відбувається масштабування та обертання, а через s– множник масштабу, через θ– кут повороту. Одна композиція масштабування та обертання навколо точки $\vec{c}$може бути подана як

$$\vec{t}' = R(\theta)\cdot\left(s(\vec{t} - \vec{c})\right) + \vec{c},$$

де матриця повороту на кут θу двовимірному просторі має стандартний вигляд

$$R(\theta) = (\cos cos\ \theta \quad - sin\theta\ \sin sin\ \theta \quad \cos cos\ \theta\ ).$$

Спочатку координати $\vec{t}$зміщуються так, щоб центр перетворення переходив у початок координат $(\vec{t} - \vec{c})$, потім виконується масштабування множником s, далі застосовується обертання матрицею $R(\theta)$, після чого результуючі координати знову зміщуються на вектор $\vec{c}$.

У фрагментному шейдері WebGL таке перетворення можна виконувати для кожного фрагмента, використовуючи uniform-параметри, які визначають центр, масштаб і кут повороту. Важливо, що сама геометрія поверхні при цьому не змінюється, змінюється лише спосіб, у який поверхня «зразкує» текстуру.

2.3. Нормалі, тангенти та тангенціальний базис

Для коректного освітлення та особливо для нормал-мапінгу потрібне розуміння нормалей і тангенціального простору. Нормаль до поверхні у певній точці описує орієнтацію мікроскопічної площадки і визначає, як поверхня взаємодіє зі світлом. Тангенти та бітангенти описують напрямки зростання текстурних координат.

У роботі для геометричних нормалей використовується підхід Facet average. Спочатку для кожного трикутника обчислюється векторна нормаль грані за формулою

$$\vec{n}_f = \frac{(\vec{b}-\vec{a})\times(\vec{c}-\vec{a})}{\|(\vec{b}-\vec{a})\times(\vec{c}-\vec{a})\|}.$$

Після цього до нормалей вершин, що належать грані, додається цей вектор, а у завершенні нормалі у вершинах нормалізуються, тобто діляться на власну довжину.

Для нормал-мапінгу необхідно мати ортонормований базис T, B, N, де T– тангент, B– бітангент, N– нормаль у просторі ока. Тангенти обчислюються аналітично як похідна за параметром β. Для фіксованого yпохідна від параметричного опису поверхні за β дає вектор

$$\frac{\partial \vec{P}}{\partial \beta} = (-\, r(y)sin\beta,\ 0,\ r(y)cos\beta),$$

який використовується як початковий тангентний вектор.

Згідно з вимогою, при ортогоналізації Грама–Шмідта пріоритет віддається саме тангенту. Це означає, що спочатку тангент нормалізується, а нормаль потім очищується від проєкції на цей тангент і нормалізується вдруге. Умовно це описується системою

$$\vec{T} := \frac{\vec{T}}{\|\vec{T}\|}, \vec{N} := \vec{N} - \vec{T}(\vec{N}\cdot\vec{T}), \vec{N} := \frac{\vec{N}}{\|\vec{N}\|},$$

Отриманий базис TBN дозволяє перетворити нормалі з normal map, що зберігаються у тангенціальному просторі, у простір ока.

2.4. Освітлення Фонга та роль текстур

Модель Фонга використовується для обчислення кольору поверхні з урахуванням навколишнього, розсіяного та дзеркального освітлення. Загальна інтенсивність у кожному каналі описується рівнянням

$$I(\vec{x}) = I_a + I_d + I_s.$$

Амб’є́нтна складова I_aмоделює фон, дифузна I_dзалежить від скалярного добутку між нормаллю та напрямком на джерело світла, а дзеркальна I_s– від скалярного добутку між відбитим вектором і напрямком на спостерігача, піднесеним до степеня, що відповідає блиску поверхні.

Цей варіант найкраще підходить для наукової роботи або технічного звіту. Він чіткий і використовує усталену термінологію.

«Застосування текстур дозволяє деталізувати візуальні властивості моделі. Дифузна карта (diffuse map) визначає базовий колір матеріалу шляхом модуляції коефіцієнта дифузного відбиття. Карта відблисків (specular map) регулює інтенсивність дзеркального відбиття в різних точках поверхні. Карта нормалей (normal map) модифікує вектори нормалей для кожного фрагмента, що дозволяє імітувати мікрорельєф без ускладнення геометричної сітки.

3. Опис реалізації

У цьому розділі описується, як теоретичні положення перенесено до коду WebGL-додатка. Розглядаються структура даних, алгоритми побудови трикутної сітки поверхні, обчислення нормалей і тангентів, організація шейдерів,

завантаження текстур, а також реалізація трансформації текстурних координат і керування з клавіатури.

3.1. Формування сітки поверхні

Програмно поверхня описується як регулярна сітка у параметричній області (u,v). У подвійно вкладеному циклі перебираються значення параметрів u та v, для кожної пари обчислюється β і y, а через функцію радіуса визначаються координати x та z. Для кожної вершини додається позиція, нульова нормаль, тангент, обчислений за аналітичною формулою, а також текстурні координати, що співпадають з u та v.

Потім параметрична сітка розбивається на трикутники. Кожен «квадрат» параметрів породжує два трикутники, номери вершин яких зберігаються в масиві індексів. Після побудови індексів запускається процедура обчислення нормалей методом Facet average: для кожного трикутника обчислюється нормаль грані, нормалізується й додається до трьох відповідних вершин, після чого нормалі у вершинах нормалізуються.

3.2. Шейдери та побудова TBN-базису

Вершинний шейдер приймає атрибути позиції, нормалі, тангенту та текстурних координат. Позиція множиться на модельно-видову матрицю, а результат передається до наступної стадії після множення на матрицю проєкції. Нормаль і тангент переносяться у простір ока за допомогою матриці ModelViewMatrix, після чого над ними виконується ортогоналізація Грама–Шмідта з пріоритетом тангента. Бітангент обчислюється як векторний добуток між нормаллю і тангентом. Всі три вектори T, B та N передаються у фрагментний шейдер як інтерпольовані змінні.

Фрагментний шейдер використовує TBN-базис, щоб перетворити нормалі з normal map (які зчитуються у просторі текстури) у простір ока. Спочатку нормаль із текстури переводиться з діапазону [0,1] у діапазон [-1,1], потім множиться на матрицю, складену з T, B та N. Далі, використовуючи цю нормаль, напрямок на джерело світла та наближений напрямок на спостерігача, обчислюються дифузна і спекулярна складові моделі Фонга. Колір дифузної складової модулюється дифузною текстурою, а інтенсивність спекулярної – значенням спекулярної текстури.

3.3. Завантаження текстур та їх застосування

Для трьох текстур створюються об’єкти WebGL-текстур. На початковому етапі у текстуру завантажується «заглушка» 1×1, яка дозволяє не чекати готовності зображення. Після того як зображення завантажується, воно передається у текстуру за допомогою виклику gl.texImage2D, генеруються міпмапи та встановлюються параметри фільтрації та обгортання.

Під час рендерингу кожній текстурі відповідає свій текстурний юніт. Фрагментному шейдеру через uniform-параметри вказується, з яких юнітів читати дифузну, нормальну та спекулярну карти. Таким чином, усі три текстури одночасно впливають на кінцевий колір фрагмента.

3.4. Реалізація трансформації текстурних координат

У фрагментному шейдері вводяться додаткові uniform-параметри, що відповідають центру трансформації у текстурному просторі, масштабу та куту повороту. У змінну uv записуються інтерпольовані текстурні координати фрагмента. Далі вони послідовно зміщуються на вектор, який переносить центр у початок координат, множаться на коефіцієнт масштабу, перетворюються матрицею повороту і, нарешті, зміщуються назад на координати центру. Уже трансформовані координати uv використовуються для зчитування всіх трьох текстур.

З боку JavaScript у глобальних змінних зберігаються поточні значення центру ($texCenterV$), масштабу texScale та кута texRotation. При кожному кадрі функція draw() передає ці значення у шейдер за допомогою відповідних uniform-викликів. Таким чином, будь-яка зміна цих параметрів негайно відображається на вигляді текстури.

3.5. Обробка керування з клавіатури

Керування точкою центру трансформації та параметрами масштабування і обертання реалізується за допомогою обробника подій клавіатури. При натисканні клавіш A і D змінюється координата u центру, при натисканні W і S змінюється координата v. Додатково можуть використовуватися клавіші для зміни масштабу та кута повороту. Кожне натискання коригує відповідну змінну на невелике значення. Після зміни параметри обмежуються допустимими діапазонами, щоб запобігти виходу за межі [0,1] для координат та уникнути нульового масштабу. Анімаційний цикл, який і так постійно оновлює зображення, автоматично відображає ці зміни.

4. Інструкція користувача

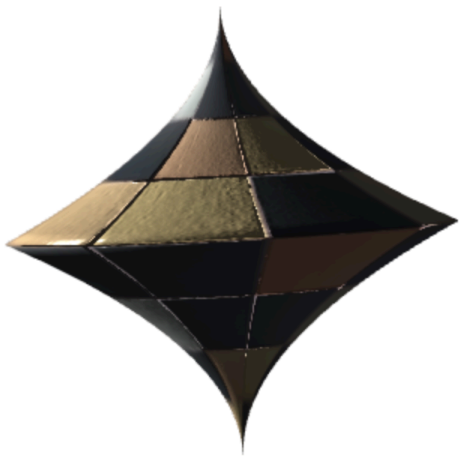
Parabolic Humming-Top – Phong Shading

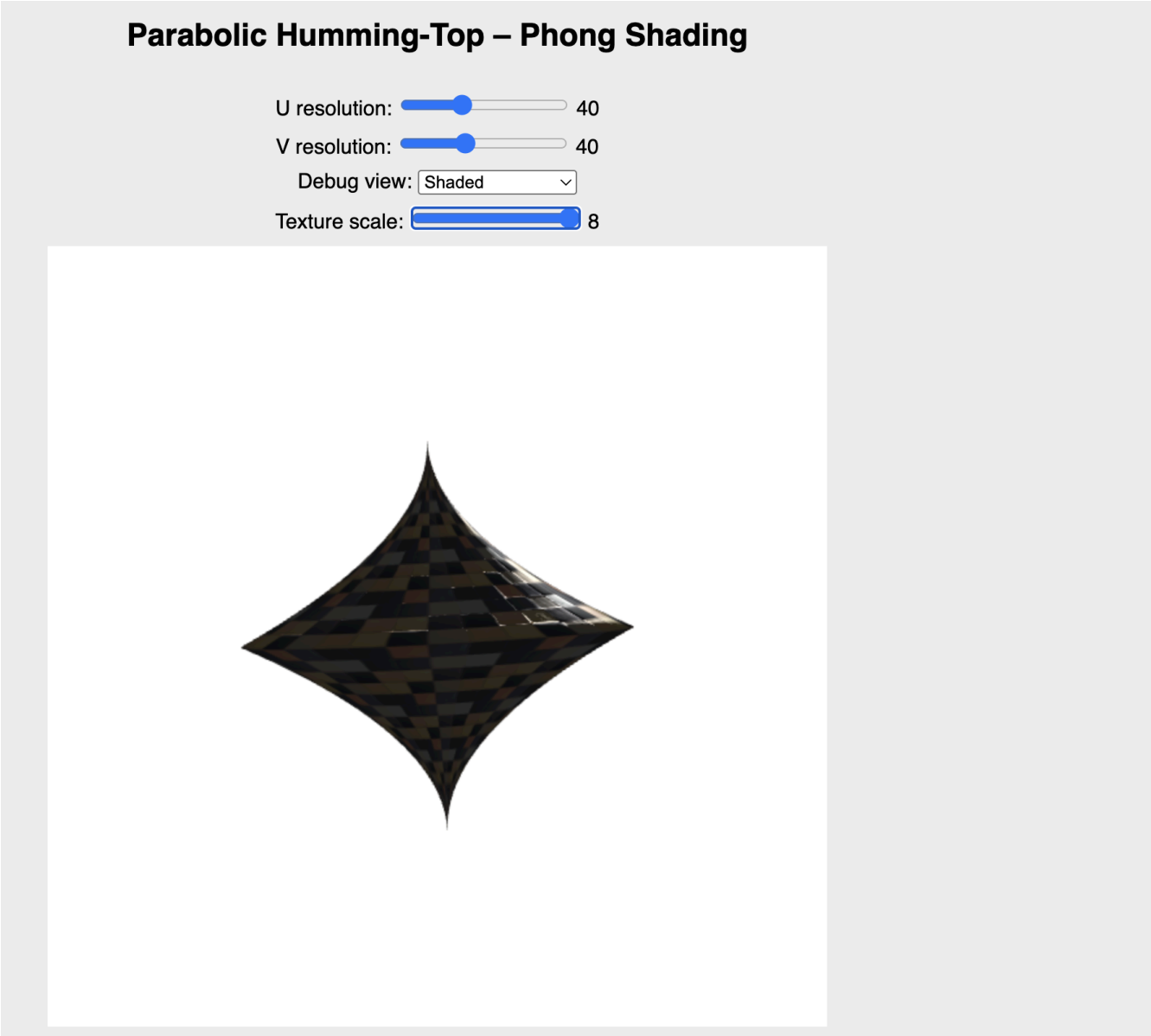
U resolution: 40

V resolution: 40

Debug view: Shaded

Texture scale: 2





Інструкція призначена для користувача, який запускає застосунок у браузері та взаємодіє з ним. У тексті залишені місця, де у фінальній версії звіту потрібно вставити скріншоти з короткими описами.

На першому етапі користувач повинен відкрити у браузері HTML-сторінку проєкту. Зазвичай це робиться через локальний HTTP-сервер, після чого у вікні браузера відображається полотно WebGL з темним фоном, текстом із назвою роботи та елементами керування у вигляді двох слайдерів під полем відображення.

Після завантаження сторінки в центрі полотна з’являється тривимірна поверхня, яка нагадує дзиґу, з накладеною бетоноподібною текстурою. Джерело світла рухається по колу навколо фігури, що створює змінні відблиски. Над поверхнею можна вільно виконувати обертання, використовуючи мишу. Для цього достатньо натиснути ліву кнопку миші на полотні і, не відпускаючи її, переміщувати курсор. На сенсорних пристроях для цієї дії використовується переміщення пальцем по полотну.

Користувач має змогу змінювати детальність сітки поверхні за допомогою двох слайдерів. Перший регулює кількість сегментів уздовж параметра U, другий – уздовж параметра V. При зменшенні значень сітка стає грубішою, окремі трикутники видно явно, а при збільшенні значення сітка стає більш щільною, поверхня виглядає гладкою. Зміна положення слайдерів призводить до перебудови геометрії, після чого поверхня негайно перемальовується з урахуванням нових параметрів.

Основна функціональність цієї роботи пов’язана з операціями над текстурними координатами. У центрі уваги знаходиться точка у просторі (u, v), яку можна вважати центром обертання та масштабування текстури. Користувач може змінювати положення цієї точки за допомогою клавіш A, D, W та S. При натисканні A центр рухається у напрямку зменшення координати u, при D – у напрямку її збільшення. При натисканні W центр зміщується у бік збільшення координати v, а при S – у бік її зменшення. У результаті користувач може вільно «ковзати» центром трансформації по поверхні.

За бажанням можна використовувати додаткові клавіші для зміни масштабу текстури та її обертання. Наприклад, клавіші Z і X змінюють масштаб, роблячи текстуру то більш дрібною і повторюваною, то більш великою. Клавіші Q та E можуть відповідати за обертання текстурного зображення навколо обраної точки, завдяки чому фактура бетону на поверхні «крутиться» незалежно від геометрії дзиґи. Усі ефекти зміни масштабу й орієнтації будуть помітні в режимі реального часу завдяки безперервному циклу анімації.

Користувачеві рекомендується спочатку поекспериментувати із слайдерами U та V, щоб підібрати бажану якість поверхні, а потім перейти до дослідження впливу переміщення центру та зміни параметрів трансформації на візуальне сприйняття текстури. Усі маніпуляції не змінюють саму геометрію, але істотно впливають на вигляд матеріалу, що дозволяє краще зрозуміти, як працює текстурний простір у WebGL.

5. Приклад вихідного коду

На завершення наведено фрагменти вихідного коду, які ілюструють побудову сітки поверхні, обчислення нормалей та тангентів, а також реалізацію вершинного та фрагментного шейдерів з операціями над текстурними координатами.

```
"use strict";

let gl; // WebGL context
let surface; // Surface model
let shProgram; // Shader program
let spaceball; // Trackball rotator
let currentTime = 0.0; // For rotating light

// Textures
```

```
let diffuseTex = null;
let normalTex = null;
let specularTex = null;

function deg2rad(angle) {
  return (angle * Math.PI) / 180;
}

// Helper: transform point by 4x4 matrix
function transformPoint(m, p) {
  const x = p[0],
    y = p[1],
    z = p[2];
  return [
    m[0] * x + m[4] * y + m[8] * z + m[12],
    m[1] * x + m[5] * y + m[9] * z + m[13],
    m[2] * x + m[6] * y + m[10] * z + m[14],
  ];
}


function Model(name) {
  this.name = name;

  this.vbo = gl.createBuffer(); // vertex positions
  this.nbo = gl.createBuffer(); // vertex normals
  this.tbo = gl.createBuffer(); // tangent vectors
  this.uvbo = gl.createBuffer(); // texture coordinates
  this.ibo = gl.createBuffer(); // indices

  this.indexCount = 0;

  /**
   * vertices:  [x,y,z,...]
   * normals:   [nx,ny,nz,...]
   * tangents:  [tx,ty,tz,...]
   * texCoords: [u,v,...]
   * indices:   [i0,i1,i2,...]
   */
  this.BufferData = function (vertices, normals, tangents, texCoords, indices) {
    this.indexCount = indices.length;

    // Positions
```

```
        gl.bindBuffer(gl.ARRAY_BUFFER, this.vbo);
                gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
gl.STATIC_DRAW);

        // Normals
        gl.bindBuffer(gl.ARRAY_BUFFER, this.nbo);
                gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(normals),
gl.STATIC_DRAW);

        // Tangents
        gl.bindBuffer(gl.ARRAY_BUFFER, this.tbo);
                gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(tangents),
gl.STATIC_DRAW);

        // Texture coordinates
        gl.bindBuffer(gl.ARRAY_BUFFER, this.uvbo);
                gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(texCoords),
gl.STATIC_DRAW);

        // Indices
        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, this.ibo);
        gl.bufferData(
          gl.ELEMENT_ARRAY_BUFFER,
          new Uint16Array(indices),
          gl.STATIC_DRAW
        );
      };

      this.Draw = function () {
        // Positions
        gl.bindBuffer(gl.ARRAY_BUFFER, this.vbo);
        gl.vertexAttribPointer(shProgram.iAttribVertex, 3, gl.FLOAT, false, 0, 0);
        gl.enableVertexAttribArray(shProgram.iAttribVertex);

        // Normals
        gl.bindBuffer(gl.ARRAY_BUFFER, this.nbo);
        gl.vertexAttribPointer(shProgram.iAttribNormal, 3, gl.FLOAT, false, 0, 0);
        gl.enableVertexAttribArray(shProgram.iAttribNormal);

        // Tangents
        gl.bindBuffer(gl.ARRAY_BUFFER, this.tbo);
        gl.vertexAttribPointer(shProgram.iAttribTangent, 3, gl.FLOAT, false, 0, 0);
        gl.enableVertexAttribArray(shProgram.iAttribTangent);
```

```
        // TexCoords
        gl.bindBuffer(gl.ARRAY_BUFFER, this.uvbo);
        gl.vertexAttribPointer(shProgram.iAttribTexCoord, 2, gl.FLOAT, false, 0, 0);
        gl.enableVertexAttribArray(shProgram.iAttribTexCoord);

        // Indices
        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, this.ibo);
          gl.drawElements(gl.TRIANGLES, this.indexCount, gl.UNSIGNED_SHORT,
0);
      };
    }


    function ShaderProgram(name, program) {
      this.name = name;
      this.prog = program;

      this.iAttribVertex = -1;
      this.iAttribNormal = -1;
      this.iAttribTexCoord = -1;
      this.iAttribTangent = -1;

      this.iModelViewMatrix = -1;
      this.iProjectionMatrix = -1;

      this.iLightPos = -1;
      this.iAmbientColor = -1;
      this.iDiffuseColor = -1;
      this.iSpecularColor = -1;
      this.iShininess = -1;

      this.iDiffuseMap = -1;
      this.iNormalMap = -1;
      this.iSpecularMap = -1;

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


    // parametric surface: (y is vertical axis)
```

```
// y ∈ [-h, h], beta ∈ [0, 2π]
function parabolicHummingTopVertex(y, beta, h, p) {
  let rBase = Math.abs(y) - h; // |y| - h
  let r = (rBase * rBase) / (2 * p); // (|y| - h)^2 / (2p)

  let x = r * Math.cos(beta);
  let z = r * Math.sin(beta);

  return [x, y, z];
}

/**
 * Create surface mesh data for given U/V granularity.
 * uSeg: segments along angle (U)
 * vSeg: segments along vertical (V)
 *
 * Returns { positions, normals, tangents, texCoords, indices }
 */
function CreateSurfaceData(uSeg, vSeg) {
  uSeg = uSeg || 40;
  vSeg = vSeg || 40;

  const h = 1.0;
  const p = 0.5;

  let positions = [];
  let normals = [];
  let tangents = [];
  let texCoords = [];
  let indices = [];

  // Build grid of vertices
  for (let j = 0; j <= vSeg; j++) {
    let v = j / vSeg;
    let y = -h + 2.0 * h * v; // from -h to h

    for (let i = 0; i <= uSeg; i++) {
      let u = i / uSeg;
      let beta = 2.0 * Math.PI * u;

      let [x, yy, z] = parabolicHummingTopVertex(y, beta, h, p);
      positions.push(x, yy, z);
```

```
    // Initial normals: zero, will be facet-average later
    normals.push(0.0, 0.0, 0.0);

    // Tangent: derivative wrt beta at fixed y
    // r does not depend on beta, so:
    let rBase = Math.abs(y) - h;
    let r = (rBase * rBase) / (2 * p);
    let tx = -r * Math.sin(beta);
    let ty = 0.0;
    let tz = r * Math.cos(beta);
    tangents.push(tx, ty, tz);

    // Simple UV mapping: (u,v) from param domain
    texCoords.push(u, v);
  }
}

const vertsPerRow = uSeg + 1;

// Helper: accumulate facet normals (facet average)
function addFace(i0, i1, i2) {
  const ax = positions[3 * i0],
    ay = positions[3 * i0 + 1],
    az = positions[3 * i0 + 2];
  const bx = positions[3 * i1],
    by = positions[3 * i1 + 1],
    bz = positions[3 * i1 + 2];
  const cx = positions[3 * i2],
    cy = positions[3 * i2 + 1],
    cz = positions[3 * i2 + 2];

  const ux = bx - ax,
    uy = by - ay,
    uz = bz - az;
  const vx = cx - ax,
    vy = cy - ay,
    vz = cz - az;

  // face normal = cross(u, v)
  let nx = uy * vz - uz * vy;
  let ny = uz * vx - ux * vz;
  let nz = ux * vy - uy * vx;
```

```
  // normalize face normal before accumulating -> facet average
  let len = Math.hypot(nx, ny, nz);
  if (len > 1e-6) {
    nx /= len;
    ny /= len;
    nz /= len;
  }

  // accumulate to vertices
  normals[3 * i0] += nx;
  normals[3 * i0 + 1] += ny;
  normals[3 * i0 + 2] += nz;

  normals[3 * i1] += nx;
  normals[3 * i1 + 1] += ny;
  normals[3 * i1 + 2] += nz;

  normals[3 * i2] += nx;
  normals[3 * i2 + 1] += ny;
  normals[3 * i2 + 2] += nz;
}

// Build triangles (two per quad) and compute normals
for (let j = 0; j < vSeg; j++) {
  for (let i = 0; i < uSeg; i++) {
    const i0 = j * vertsPerRow + i;
    const i1 = i0 + 1;
    const i2 = i0 + vertsPerRow;
    const i3 = i2 + 1;

    // triangle 1
    indices.push(i0, i2, i1);
    addFace(i0, i2, i1);

    // triangle 2
    indices.push(i1, i2, i3);
    addFace(i1, i2, i3);
  }
}

// Normalize accumulated vertex normals
for (let k = 0; k < normals.length; k += 3) {
  let nx = normals[k];
```

```
        let ny = normals[k + 1];
        let nz = normals[k + 2];
        let len = Math.hypot(nx, ny, nz);
        if (len > 1e-6) {
          normals[k] = nx / len;
          normals[k + 1] = ny / len;
          normals[k + 2] = nz / len;
        } else {
          normals[k] = 0.0;
          normals[k + 1] = 1.0;
          normals[k + 2] = 0.0;
        }
      }

      return { positions, normals, tangents, texCoords, indices };
    }

    /*======================                    TEXTURE          LOADING
======================*/
    function isPowerOf2(value) {
      return (value & (value - 1)) === 0;
    }

    function loadTexture(url) {
      const texture = gl.createTexture();
      gl.bindTexture(gl.TEXTURE_2D, texture);

      // Temporary 1x1 pixel until image loads
      const tempPixel = new Uint8Array([128, 128, 128, 255]);
      gl.texImage2D(
        gl.TEXTURE_2D,
        0,
        gl.RGBA,
        1,
        1,
        0,
        gl.RGBA,
        gl.UNSIGNED_BYTE,
        tempPixel
      );

      const image = new Image();
      // Allow CORS for same-origin or servers that send ACAO headers
```

```
        image.crossOrigin = "anonymous";
        image.onerror = function () {
          console.error("Failed to load texture:", url);
        };
        image.onload = function () {
          gl.bindTexture(gl.TEXTURE_2D, texture);
          gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
                        gl.texImage2D(gl.TEXTURE_2D,     0,     gl.RGBA,     gl.RGBA,
gl.UNSIGNED_BYTE, image);

          const pot = isPowerOf2(image.width) && isPowerOf2(image.height);
          if (pot) {
            gl.generateMipmap(gl.TEXTURE_2D);
            gl.texParameteri(
              gl.TEXTURE_2D,
              gl.TEXTURE_MIN_FILTER,
              gl.LINEAR_MIPMAP_LINEAR
            );
                        gl.texParameteri(gl.TEXTURE_2D,   gl.TEXTURE_MAG_FILTER,
gl.LINEAR);
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
          } else {
            // Non-POT textures: no mipmaps and clamp to edge
                         gl.texParameteri(gl.TEXTURE_2D,   gl.TEXTURE_MIN_FILTER,
gl.LINEAR);
                        gl.texParameteri(gl.TEXTURE_2D,   gl.TEXTURE_MAG_FILTER,
gl.LINEAR);
                             gl.texParameteri(gl.TEXTURE_2D,    gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE);
                             gl.texParameteri(gl.TEXTURE_2D,    gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE);
          }
          console.log(
            "Loaded texture",
            url,
            image.width + "x" + image.height,
            "POT:",
            pot
          );
        };
        image.src = url;
```

```
  return texture;
}

/*====================== DRAW ======================*/

function draw() {
  gl.clearColor(1, 1, 1, 1);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  // Projection (perspective)
  let projection = m4.perspective(Math.PI / 8, 1, 2, 20);

  // View
  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); // ModelViewMatrix

  // Send matrices
  gl.uniformMatrix4fv(shProgram.iModelViewMatrix, false, matAccum1);
  gl.uniformMatrix4fv(shProgram.iProjectionMatrix, false, projection);

  // Rotating light position in model space (circle around top)
  const lightRadius = 5.0;
  const lightHeight = 2.0;
  const lightSpeed = 0.5; // radians per second

  let angle = currentTime * lightSpeed;
  let lightPosModel = [
    lightRadius * Math.cos(angle),
    lightHeight,
    lightRadius * Math.sin(angle),
  ];

  // Transform light to eye space
  let lightPosEye = transformPoint(matAccum1, lightPosModel);
  gl.uniform3fv(shProgram.iLightPos, new Float32Array(lightPosEye));

  // Bind textures to texture units
```

```
        gl.activeTexture(gl.TEXTURE0);
        gl.bindTexture(gl.TEXTURE_2D, diffuseTex);
        gl.uniform1i(shProgram.iDiffuseMap, 0);

        gl.activeTexture(gl.TEXTURE1);
        gl.bindTexture(gl.TEXTURE_2D, normalTex);
        gl.uniform1i(shProgram.iNormalMap, 1);

        gl.activeTexture(gl.TEXTURE2);
        gl.bindTexture(gl.TEXTURE_2D, specularTex);
        gl.uniform1i(shProgram.iSpecularMap, 2);

        // Draw surface
        surface.Draw();
    }

    /*======================= ANIMATION LOOP =======================*/

    function animate(time) {
        currentTime = time * 0.001; // ms -> s
        draw();
        requestAnimationFrame(animate);
    }

    /*======================= INIT GL =======================*/

    function initGL() {
        let prog = createProgram(gl, vertexShaderSource, fragmentShaderSource);

        shProgram = new ShaderProgram("TexturedPhong", prog);
        shProgram.Use();

        // Attributes
        shProgram.iAttribVertex = gl.getAttribLocation(prog, "vertex");
        shProgram.iAttribNormal = gl.getAttribLocation(prog, "normal");
        shProgram.iAttribTexCoord = gl.getAttribLocation(prog, "texCoord");
        shProgram.iAttribTangent = gl.getAttribLocation(prog, "tangent");

        // Uniforms
        shProgram.iModelViewMatrix = gl.getUniformLocation(prog, "ModelViewMatrix");
```

```
        shProgram.iProjectionMatrix = gl.getUniformLocation(prog, "ProjectionMatrix");

        shProgram.iLightPos = gl.getUniformLocation(prog, "uLightPos");
        shProgram.iAmbientColor = gl.getUniformLocation(prog, "uAmbientColor");
        shProgram.iDiffuseColor = gl.getUniformLocation(prog, "uDiffuseColor");
        shProgram.iSpecularColor = gl.getUniformLocation(prog, "uSpecularColor");
        shProgram.iShininess = gl.getUniformLocation(prog, "uShininess");

        shProgram.iDiffuseMap = gl.getUniformLocation(prog, "uDiffuseMap");
        shProgram.iNormalMap = gl.getUniformLocation(prog, "uNormalMap");
        shProgram.iSpecularMap = gl.getUniformLocation(prog, "uSpecularMap");
        shProgram.iDebugMode = gl.getUniformLocation(prog, "uDebugMode");
        shProgram.iTexScale = gl.getUniformLocation(prog, "uTexScale");

        // Lighting constants (can tweak)
        gl.uniform3fv(shProgram.iAmbientColor, new Float32Array([0.2, 0.2, 0.2]));
        gl.uniform3fv(shProgram.iDiffuseColor, new Float32Array([1.0, 1.0, 1.0]));
        gl.uniform3fv(shProgram.iSpecularColor, new Float32Array([1.0, 1.0, 1.0]));
        gl.uniform1f(shProgram.iShininess, 32.0);

        surface = new Model("ParabolicHummingTop");

        gl.enable(gl.DEPTH_TEST);
        gl.enable(gl.CULL_FACE);
        gl.cullFace(gl.BACK);

        // Load textures
        diffuseTex = loadTexture("./textures/diffuse.png");
        normalTex = loadTexture("./textures/normal.png");
        specularTex = loadTexture("./textures/specular.png");

        // Default debug mode: shaded rendering
        gl.uniform1i(shProgram.iDebugMode, 0);
        // Default texture tiling
        gl.uniform1f(shProgram.iTexScale, 1.0);
      }

      /*====================== SHADER CREATION ======================*/

      function createProgram(gl, vShader, fShader) {
        let vsh = gl.createShader(gl.VERTEX_SHADER);
```

```
  gl.shaderSource(vsh, vShader);
  gl.compileShader(vsh);
  if (!gl.getShaderParameter(vsh, gl.COMPILE_STATUS)) {
    throw new Error("Error in vertex shader:  " + gl.getShaderInfoLog(vsh));
  }

  let fsh = gl.createShader(gl.FRAGMENT_SHADER);
  gl.shaderSource(fsh, fShader);
  gl.compileShader(fsh);
  if (!gl.getShaderParameter(fsh, gl.COMPILE_STATUS)) {
    throw new Error("Error in fragment shader:  " + gl.getShaderInfoLog(fsh));
  }

  let prog = gl.createProgram();
  gl.attachShader(prog, vsh);
  gl.attachShader(prog, fsh);
  gl.linkProgram(prog);
  if (!gl.getProgramParameter(prog, gl.LINK_STATUS)) {
    throw new Error("Link error in program:  " + gl.getProgramInfoLog(prog));
  }
  return prog;
}


function init() {
  let canvas;
  try {
    canvas = document.getElementById("webglcanvas");
    gl = canvas.getContext("webgl");
    if (!gl) {
      throw "Browser does not support WebGL";
    }
  } catch (e) {
    document.getElementById("canvas-holder").innerHTML =
      "<p>Sorry, could not get a WebGL graphics context.</p>";
    return;
  }
  try {
    initGL();
  } catch (e) {
    document.getElementById("canvas-holder").innerHTML =
      "<p>Sorry, could not initialize the WebGL graphics context: " +
      e +
```

```
    "</p>";
  return;
}

spaceball = new TrackballRotator(canvas, draw, 0);

// Sliders from PA#2
const uSlider = document.getElementById("uResolution");
const vSlider = document.getElementById("vResolution");
const uVal = document.getElementById("uVal");
const vVal = document.getElementById("vVal");
const debugSelect = document.getElementById("debugMode");
const texScaleSlider = document.getElementById("texScale");
const texScaleVal = document.getElementById("texScaleVal");

function updateSurfaceFromSliders() {
  const uSeg = parseInt(uSlider.value);
  const vSeg = parseInt(vSlider.value);

  uVal.textContent = uSeg.toString();
  vVal.textContent = vSeg.toString();

  const data = CreateSurfaceData(uSeg, vSeg);
  surface.BufferData(
    data.positions,
    data.normals,
    data.tangents,
    data.texCoords,
    data.indices
  );

  draw();
}

uSlider.oninput = updateSurfaceFromSliders;
vSlider.oninput = updateSurfaceFromSliders;
debugSelect.onchange = function () {
  const mode = parseInt(debugSelect.value, 10) || 0;
  gl.uniform1i(shProgram.iDebugMode, mode);
  draw();
};

texScaleSlider.oninput = function () {
```

```
    const scale = parseFloat(texScaleSlider.value);
    texScaleVal.textContent = scale.toString();
    gl.uniform1f(shProgram.iTexScale, scale);
    draw();
  };

  updateSurfaceFromSliders();

  // Ensure debug uniform reflects initial selector value
  gl.uniform1i(shProgram.iDebugMode, parseInt(debugSelect.value, 10) || 0);
  gl.uniform1f(shProgram.iTexScale, parseFloat(texScaleSlider.value));

  requestAnimationFrame(animate);
}

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

uniform mat4 ModelViewMatrix;
uniform mat4 ProjectionMatrix;

varying vec3 vPosition;   // position in eye space
varying vec2 vTexCoord;
varying vec3 vT;
varying vec3 vB;
varying vec3 vN;

void main() {
    vec4 posEye = ModelViewMatrix * vec4(vertex, 1.0);
    vPosition = posEye.xyz;
    vTexCoord = texCoord;

    mat3 MV3 = mat3(ModelViewMatrix);

    // Transform normal and tangent to eye space
    vec3 T = MV3 * tangent;
    vec3 N = MV3 * normal;

    // === Gram–Schmidt, giving PRIORITY to tangent ===
```

```
    // First, normalize tangent -> primary basis vector
    T = normalize(T);

    // Then make normal orthogonal to tangent
    N = normalize(N - T * dot(N, T));

    // Bitangent completes TBN
    vec3 B = cross(N, T);

    vT = T;
    vB = B;
    vN = N;

    gl_Position = ProjectionMatrix * posEye;
}`;


// Fragment shader
const fragmentShaderSource = `
#ifdef GL_FRAGMENT_PRECISION_HIGH
precision highp float;
#else
precision mediump float;
#endif

varying vec3 vPosition;
varying vec2 vTexCoord;
varying vec3 vT;
varying vec3 vB;
varying vec3 vN;

uniform vec3 uLightPos;      // light position in eye space

uniform sampler2D uDiffuseMap;
uniform sampler2D uNormalMap;
uniform sampler2D uSpecularMap;

uniform vec3 uAmbientColor;
uniform vec3 uDiffuseColor;
uniform vec3 uSpecularColor;
uniform float uShininess;

// Debug mode: 0=shaded, 1=diffuse tex, 2=normal tex, 3=specular tex
```

```
uniform int uDebugMode;
// Texture tiling multiplier
uniform float uTexScale;

void main() {
    // Build TBN matrix (eye space)
    vec3 T = normalize(vT);
    vec3 B = normalize(vB);
    vec3 N = normalize(vN);
    mat3 TBN = mat3(T, B, N);

    // UV with tiling
    vec2 uv = vTexCoord * uTexScale;

    // Normal from normal map (tangent space -> eye space)
    vec3 nTex = texture2D(uNormalMap, uv).rgb;
    nTex = nTex * 2.0 - 1.0;   // [0,1] -> [-1,1]
    vec3 Neye = normalize(TBN * nTex);

    // Lighting vectors
    vec3 L = normalize(uLightPos - vPosition);
    float lambert = max(dot(Neye, L), 0.0);

    // Viewer direction: from fragment to eye (eye at origin in eye space)
    vec3 V = normalize(-vPosition);
    vec3 R = reflect(-L, Neye);
    float spec = 0.0;
    if (lambert > 0.0) {
        spec = pow(max(dot(R, V), 0.0), uShininess);
    }

    // Sample textures
    vec3 texDiffuse   = texture2D(uDiffuseMap,   uv).rgb;
    float texSpecular = texture2D(uSpecularMap, uv).r;

    // Debug outputs
    if (uDebugMode == 1) {
        gl_FragColor = vec4(texDiffuse, 1.0);
        return;
    } else if (uDebugMode == 2) {
        // Show normal map in RGB (tangent space), remapped to [0,1]
        gl_FragColor = vec4(0.5 * (nTex + 1.0), 1.0);
        return;
```

```
    } else if (uDebugMode == 3) {
        gl_FragColor = vec4(vec3(texSpecular), 1.0);
        return;
    }

    // Final shaded color
    vec3 color =
        uAmbientColor * texDiffuse +
        uDiffuseColor * lambert * texDiffuse +
        uSpecularColor * spec * texSpecular;

    gl_FragColor = vec4(color, 1.0);
}`;
```