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»

Варіант 12

**Виконав:**

студент 5 курсу

групи ТР-23мп, ТЕФ

Кондратюк Ігор Віталійович

**Перевірив:**

Демчишин А. А.

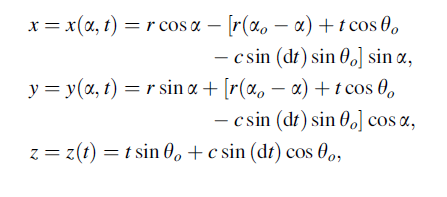
Київ – 2022

**Завдання**

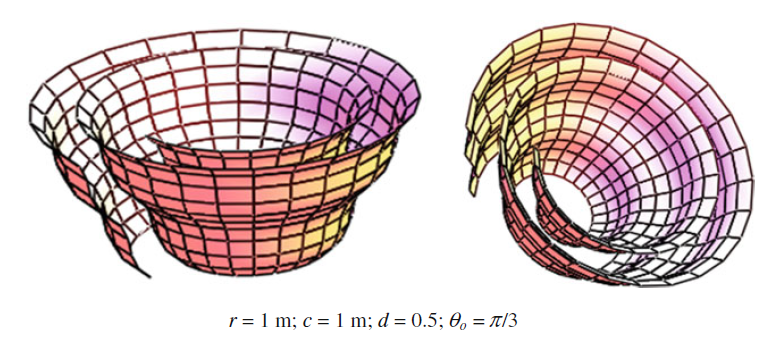
1. Нанести текстуру на поверхню з лабораторної роботи №2.
2. Реалізувати обертання текстури навколо визначеної користувачем точки.
3. Додати можливість переміщати точку вздовж простору поверхні (u, v) за допомогою клавіатури. Наприклад клавіші A і D переміщують точку вздовж параметра u, а клавіші W і S переміщують точку вздовж параметра v.

**Теоретична інформація**

Поверхню «Monge surface with a cylindrical directrix surface and a sinusoid curve as meridian» можна визначити як поверхню, утворену рухомою синусоїди, площина якої без ковзання котиться над правим круговим циліндром. Приймаючи синусоїду як твірну або меридіан, а евольвенту кола як директрису або паралель, можна дати інше визначення для досліджуваної поверхні: дана поверхня утворюється з циліндричною поверхнею директриси та синусоїдальною твірною сформованою плоскою синусоїдою, що рухається вздовж евольвенти директриси кола, так що площина з твірною синусоїдою весь час лежить на нормальній площині евольвенти директриси кола і міцно пов’язана з нею.



*Рис.1 Параметричні рівняння*



*Рис.2 «Monge surface with a cylindrical directrix surface and a sinusoid curve as meridian»*

Поверхня задана лініями кривизни t, α. Одне сімейство плоских ліній кривизни, позначених t, збігається з твірною синусоїди, а друге сімейство плоских ліній кривизни α є сімейством евольвент кола радіусом r.

Для того щоб отримати більший реалізм фігури, необхідно було б мати набагато більше вершин, щоб ми могли вказати більше кольорів. Даний підхід створює значну кількість додаткових накладних витрат, так як для кожного об'єкта, що створюється, знадобиться набагато більша кількість вершин, що використовують додаткову інформацію у вигляді атрибутів кольору.

Текстура – це 2D-зображення, що використовується для додавання деталей до об'єкта. Цей спосіб дозволяє додавати безліч деталей в одне зображення, а це означає, що у нас з'являється можливість створювати ілюзію надзвичайно деталізованого об'єкта без необхідності вказувати додаткові вершини.

Крім безпосередньо зображень, текстури також можуть використовуватися і для зберігання найрізноманітніших даних, призначених для відправки в шейдери.

**Деталі реалізації**

Для реалізації накладання текстури я використав мову програмування JavaScript та бібліотеку WebGL. Процес складається з таких етапів:

1. Перше, що потрібно зробити, це додати код для завантаження текстур та налаштувати параметри текстурування після завантаження самої текстури, що зробить функція LoadTexture(). У нашому випадку ми будемо використовувати одну текстуру, нанесену на фігуру. Розіб’ємо процес нанесення текстури на підпункти:
   1. Створюємо елемент Image, який буде виступати джерелом текстури та задаємо атрибут src текстури для даного елемента.
   2. Визначення методу onload, який міститиме логіку, яка спрацьовує при завантаженні зображення в елемент Image.
   3. Всередині методу onload визначаємо та створ’ємо об'єкта текстури за допомогою методу gl.createTexture().
   4. Далі виконуємо прив'язку текстури за допомогою методу gl.bindTexture().
   5. Налаштувуємо параметри текстури за допомогою методу gl.texParameteri().
   6. Після чого завантажуємо текстури у GPU за допомогою методу gl.texImage2D().

function LoadTexture() {  
 const image = new Image();  
 image.crossOrigin = 'anonymous';  
 image.src = 'https://www.the3rdsequence.com/texturedb/download/165/texture/jpg/1024/plastic+stripes-1024x1024.jpg';  
 image.onload = () => {  
 const 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*);   
 ***gl***.texImage2D(***gl***.*TEXTURE\_2D*, 0, ***gl***.*RGBA*, ***gl***.*RGBA*, ***gl***.*UNSIGNED\_BYTE*, image);  
 draw();  
 };  
}

1. Змінюємо шейдер, додаємо в нього інформацію про текстуру та додаємо функцію в шейдері для обертання текстури навколо точки.

mat4 getRotate(float angleRad) {  
 float c = cos(angleRad);  
 float s = sin(angleRad);  
 return mat4(  
 vec4(c, s, 0.0, 0.0),  
 vec4(-s, c, 0.0, 0.0),  
 vec4(0.0, 0.0, 1.0, 0.0),  
 vec4(0.0, 0.0, 0.0, 1.0)  
 );  
}

1. Для моделі фігури створюємо буфер для текстурних координат та додаєм вказівник на читання з буфера вершин.

***gl***.bindBuffer(***gl***.*ARRAY\_BUFFER*, this.iTextureBuffer);  
***gl***.vertexAttribPointer(***shProgram***.iTextureCoords, 2, ***gl***.*FLOAT*, false, 0, 0);  
***gl***.enableVertexAttribArray(***shProgram***.iTextureCoords);

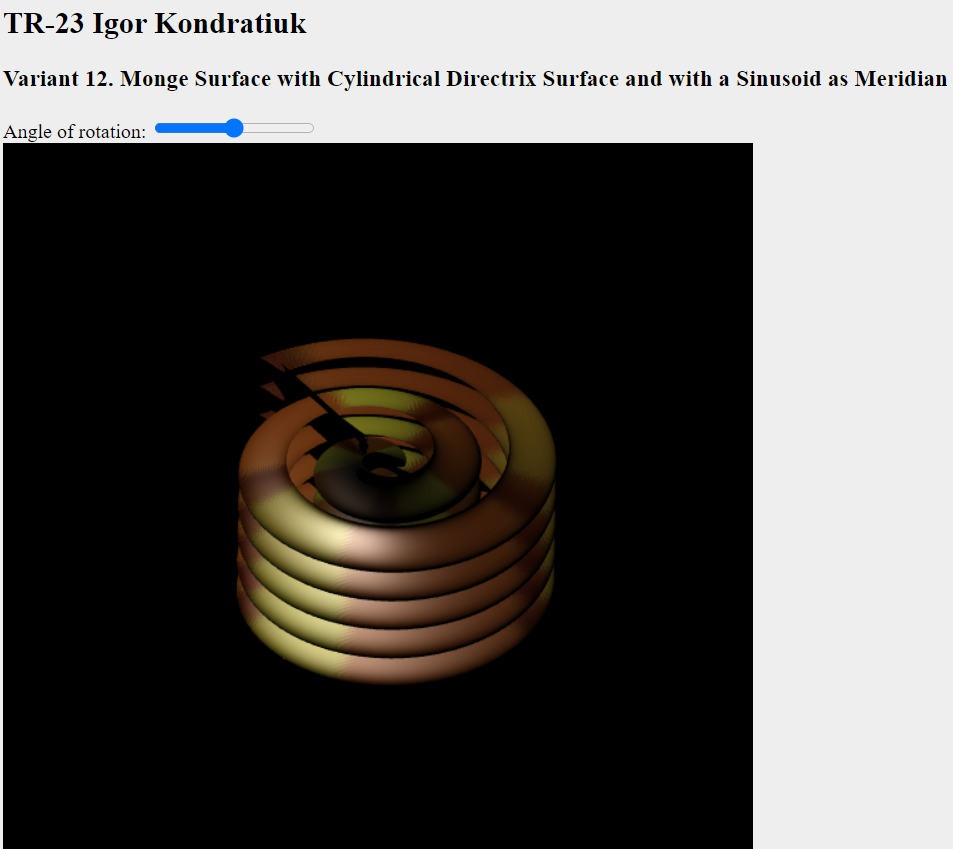
1. Додаємо зміну точки обертання через натиснення клавіш w, s, a, d:

let ***point*** = { x: 0, y: 0 };

***window***.addEventListener("keydown", (event) => {   
 const step = 0.1  
 switch (event.key) {  
 case 'w':  
 ***point***.y = ***point***.y + step;  
 draw();  
 break;  
 case 's':  
 ***point***.y = ***point***.y - step;  
 draw();  
 break;  
 case 'd':  
 ***point***.x = ***point***.x + step;  
 draw();  
 break;  
 case 'a':  
 ***point***.x = ***point***.x - step;  
 draw();  
 break;  
 default:  
 break;  
 }  
});

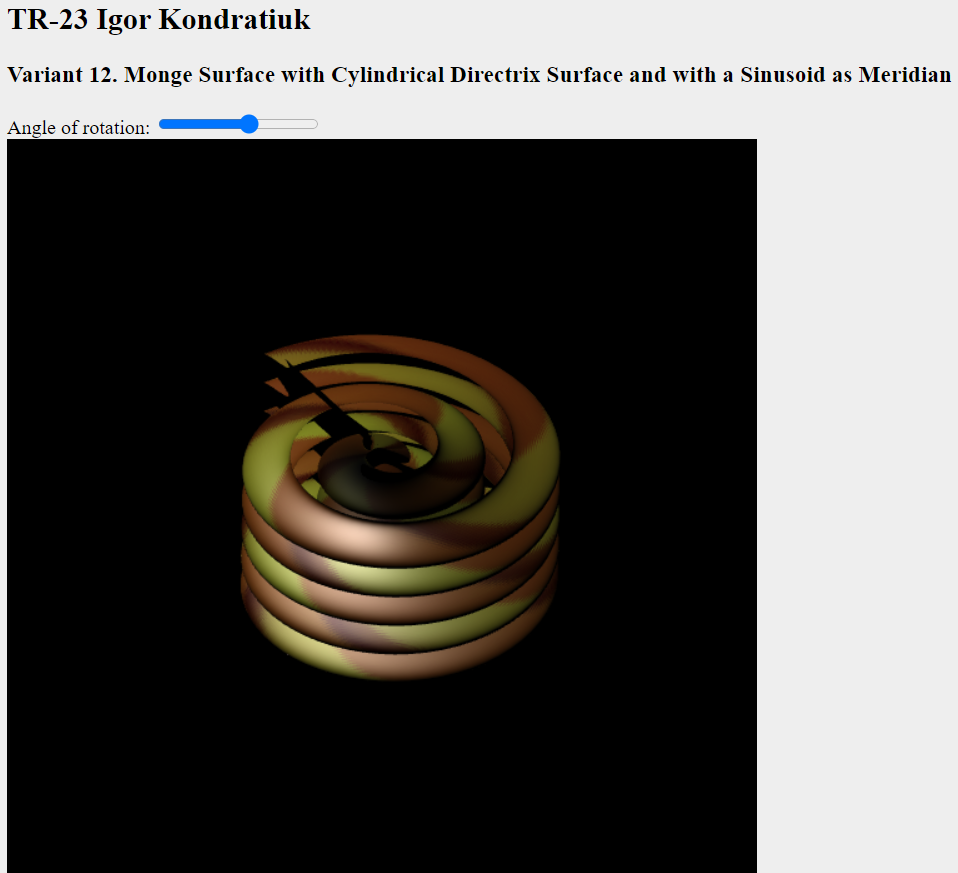
**Інструкція користувача**

Нижче наведені зображення для користування та демонстрації функціоналу програми.

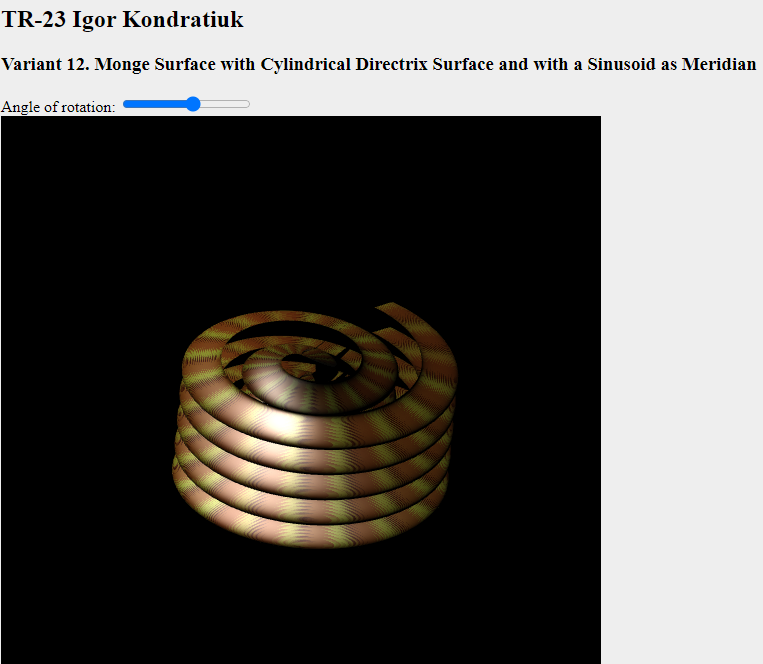


*Рис.3 Поверхня із стандартним значенням кута оберту текстури*

За допомогою клавіш «W», «S», «A», «D» користувач може змінювати позицію точки навколо якої буде здійснюватися обертання. Саме ж обертання здійснюється за допомогою зміни параметру «Angle of rotation» на сторінці.



*Рис.4 Змінений параметр кута оберту текстури*



*Рис.5 Змінений точки оберту та параметра кута оберту текстури*

**Вихідний код**

// Init data for calculation figure coordinates  
let ***r*** = 1;  
let ***c*** = 2;  
let ***d*** = 1;  
let ***teta*** = ***Math***.PI/2;  
let ***a0*** = 0;  
  
// Functions for calculation X,Y,Z coordinates for surface  
function getX (t,a, param = 15) {  
 return ((***r*** \* ***Math***.cos(a) - (***r*** \* (***a0*** - a) + t \* ***Math***.cos(***teta***) - ***c*** \* ***Math***.sin(***d*** \* t) \* ***Math***.sin(***teta***)) \* ***Math***.sin(a)) / param) \* ***scale***;  
}  
function getY (t,a, param = 15) {  
 return ((***r*** \* ***Math***.sin(a) + (***r*** \* (***a0*** - a) + t \* ***Math***.cos(***teta***) - ***c*** \* ***Math***.sin(***d*** \* t) \* ***Math***.sin(***teta***)) \* ***Math***.cos(a)) / param) \* ***scale***;  
}  
function getZ (t, height = 15) {  
 return ((t \* ***Math***.sin(***teta***) + ***c*** \* ***Math***.sin(***d*** \* t) \* ***Math***.cos(***teta***)) / (-height)) \* ***scale***;  
}  
  
function deg2rad(angle) { return angle \* ***Math***.PI / 180;}  
  
// Constructor  
function Model(name) {  
 this.name = name;  
 this.iVertexBuffer = ***gl***.createBuffer();  
 this.iNormalBuffer = ***gl***.createBuffer();  
 this.iTextureBuffer = ***gl***.createBuffer();  
 this.count = 0;  
  
 this.BufferData = function({ vertexList, normalsList, textureList }) {  
  
 ***gl***.bindBuffer(***gl***.*ARRAY\_BUFFER*, this.iVertexBuffer);  
 ***gl***.bufferData(***gl***.*ARRAY\_BUFFER*, new ***Float32Array***(vertexList), ***gl***.*STREAM\_DRAW*);  
   
 ***gl***.bindBuffer(***gl***.*ARRAY\_BUFFER*, this.iNormalBuffer)  
 ***gl***.bufferData(***gl***.*ARRAY\_BUFFER*, new ***Float32Array***(normalsList), ***gl***.*STREAM\_DRAW*);  
  
 ***gl***.bindBuffer(***gl***.*ARRAY\_BUFFER*, this.iTextureBuffer);  
 ***gl***.bufferData(***gl***.*ARRAY\_BUFFER*, new ***Float32Array***(textureList), ***gl***.*STREAM\_DRAW*);  
  
 this.count = vertexList.length/3;  
 }  
  
 this.Draw = function() {  
  
 ***gl***.bindBuffer(***gl***.*ARRAY\_BUFFER*, this.iVertexBuffer);  
 ***gl***.vertexAttribPointer(***shProgram***.iAttribVertex, 3, ***gl***.*FLOAT*, false, 0, 0);  
 ***gl***.enableVertexAttribArray(***shProgram***.iAttribVertex);  
  
 ***gl***.bindBuffer(***gl***.*ARRAY\_BUFFER*, this.iNormalBuffer);  
 ***gl***.vertexAttribPointer(***shProgram***.iNormalVertex, 3, ***gl***.*FLOAT*, true, 0, 0);  
 ***gl***.enableVertexAttribArray(***shProgram***.iNormalVertex);  
  
 ***gl***.bindBuffer(***gl***.*ARRAY\_BUFFER*, this.iTextureBuffer);  
 ***gl***.vertexAttribPointer(***shProgram***.iTextureCoords, 2, ***gl***.*FLOAT*, false, 0, 0);  
 ***gl***.enableVertexAttribArray(***shProgram***.iTextureCoords);  
 ***gl***.drawArrays(***gl***.*TRIANGLE\_STRIP*, 0, this.count);  
 }  
}  
  
// Constructor  
function ShaderProgram(name, program) {  
  
 this.name = name;  
 this.prog = program;  
  
 // Location of the attribute variable in the shader program.  
 this.iAttribVertex = -1;  
 // Location of the uniform specifying a color for the primitive.  
 this.iColor = -1;  
 // Location of the uniform matrix representing the combined transformation.  
 this.iModelViewProjectionMatrix = -1;  
 this.iNormalVertex = -1;  
 this.iWorldMatrix = -1;  
 this.iWorldInverseTranspose = -1;  
  
 this.iLightWorldPosition = -1;  
 this.iLightDirection = -1;  
 this.iViewWorldPosition = -1;  
 this.iLimit = -1;  
  
 // textCoords  
 this.iTextureCoords = -1;  
 this.iTMU = -1;  
  
 this.iFAngleRad = -1;  
 this.iFPoint = -1;  
  
 this.Use = function() {  
 ***gl***.useProgram(this.prog);  
 }  
}  
  
function draw() {  
 const angle = ***document***.getElementById('rotAngle').value;  
 ***gl***.clearColor(0,0,0,1);  
 ***gl***.clear(***gl***.*COLOR\_BUFFER\_BIT* | ***gl***.*DEPTH\_BUFFER\_BIT*);  
  
 ***gl***.enable(***gl***.*CULL\_FACE*);  
 ***gl***.enable(***gl***.*DEPTH\_TEST*);  
  
 /\* Set the values of the projection transformation \*/  
 let projection = m4.orthographic(-25, 25, -25, 25, -25, 25);  
  
 /\* Get the view matrix from the SimpleRotator object.\*/  
 let modelView = ***spaceball***.getViewMatrix();  
  
 let WorldMatrix = m4.translation(0, 0, -15);  
  
 let matAccum1 = m4.multiply(WorldMatrix, modelView );  
 let modelViewProjection = m4.multiply(projection, matAccum1 );  
  
 var worldInverseMatrix = m4.inverse(matAccum1);  
 var worldInverseTransposeMatrix = m4.transpose(worldInverseMatrix);  
  
 ***gl***.uniform3fv(***shProgram***.iViewWorldPosition, [0, 0, 0]);  
  
 ***gl***.uniform1f(***shProgram***.iLimit, ***Math***.cos(deg2rad(45)));  
 ***gl***.uniform3fv(***shProgram***.iLightWorldPosition, calcParabola());  
 ***gl***.uniform3fv(***shProgram***.iLightDirection, [0, -1, 0]);  
  
 ***gl***.uniformMatrix4fv(***shProgram***.iWorldInverseTranspose, false, worldInverseTransposeMatrix);  
 ***gl***.uniformMatrix4fv(***shProgram***.iModelViewProjectionMatrix, false, modelViewProjection );  
 ***gl***.uniformMatrix4fv(***shProgram***.iWorldMatrix, false, matAccum1 );  
   
 ***gl***.uniform4fv(***shProgram***.iColor, ***generalColor*** );  
  
 ***gl***.uniform1f(***shProgram***.iFAngleRad, deg2rad(+angle));  
   
 ***gl***.uniform2fv(***shProgram***.iFPoint, [getX(***point***.x, ***point***.y), getY(***point***.x, ***point***.y)]);  
   
 ***gl***.uniform1i(***shProgram***.iTMU, 0);  
  
 ***surface***.Draw();  
}  
  
function CreateSurfaceData() {  
 let vertexList = [];  
 let normalsList = [];  
 let textureList = [];  
  
 let deltaT = 0.0005;  
 let deltaA = 0.0005;  
  
 const step = 0.1  
  
 for (let t = -15; t <= 15; t += step) {  
 for (let a = 0; a <= 15; a += step) {  
 const tNext = t + step;   
 vertexList.push(getX(t, a, 10), getY(t, a, 10), getZ(t, 20));  
 vertexList.push(getX(tNext, a, 10), getY(tNext, a, 10), getZ(tNext, 20));  
  
 // Normals  
 let result = m4.cross(calcDerT(t, a, deltaT), calcDerA(t, a, deltaA));  
 normalsList.push(result[0], result[1], result[2])  
  
 result = m4.cross(calcDerT(tNext, a, deltaT), calcDerA(tNext, a, deltaA));  
 normalsList.push(result[0], result[1], result[2]);  
  
 textureList.push(...calcTextureTA(t, a));  
 textureList.push(...calcTextureTA(tNext, a + step));  
 }  
 }  
  
 return { vertexList, normalsList, textureList };  
}  
  
const calcDerT = (t, a, tDelta) => ([  
 (getX(t + tDelta, a, 10) - getX(t, a, 10)) / deg2rad(tDelta),  
 (getY(t + tDelta, a, 10) - getY(t, a, 10)) / deg2rad(tDelta),  
 (getZ(t + tDelta, a) - getZ(t, a)) / deg2rad(tDelta),  
])  
  
const calcDerA = (t, a, aDelta) => ([  
 (getX(t, a + aDelta, 10) - getX(t, a, 10)) / deg2rad(aDelta),  
 (getY(t, a + aDelta, 10) - getY(t, a, 10)) / deg2rad(aDelta),  
 (getZ(t, a + aDelta) - getZ(t, a)) / deg2rad(aDelta),  
])  
  
***window***.addEventListener("keydown", (event) => {   
 const step = 0.1  
 switch (event.key) {  
 case 'w':  
 ***point***.y = ***point***.y + step;  
 draw();  
 break;  
 case 's':  
 ***point***.y = ***point***.y - step;  
 draw();  
 break;  
 case 'd':  
 ***point***.x = ***point***.x + step;  
 draw();  
 break;  
 case 'a':  
 ***point***.x = ***point***.x - step;  
 draw();  
 break;  
 default:  
 break;  
 }  
});  
  
function LoadTexture() {  
 const image = new Image();  
 image.crossOrigin = 'anonymous';  
 image.src = 'https://www.the3rdsequence.com/texturedb/download/165/texture/jpg/1024/plastic+stripes-1024x1024.jpg';  
 image.onload = () => {  
 const 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*);   
 ***gl***.texImage2D(***gl***.*TEXTURE\_2D*, 0, ***gl***.*RGBA*, ***gl***.*RGBA*, ***gl***.*UNSIGNED\_BYTE*, image);  
 draw();  
 };  
}