National Technical University of Ukraine

"Ihor Sikorsky Kyiv Polytechnic Institute"

Institute of Atomic and Thermal Energy

Department of digital technologies in energy

Visualization of Graphical and Geometrical Information

Calculation and graphics work

Executed by:

Syshylnykov Kyrylo

TR-23mp

2023 year

Chapter describing the task

In this calculated graphic work, I need to learn and complete the task of adding texture coordinates to the designed figure in previous works.

Scale and rotate a texture around user specified point.

Requirements:

Map the texture over the surface from practical assignment #2.

Implement texture scaling (texture coordinates) scaling / rotation around user specified point- odd variants implement scaling, even variants implement rotation

It has to be possible to move the point along the surface (u,v) space using a keyboard. E.g. keys A and D move the point along u parameter and keys W and S move the point along v parameter.

Theory

Texture coordinates are typically provided to the shader program as an attribute variable. However, when texture coordinates are not available, it is possible to generate them in the shader program. While the results will not usually look as good as using texture coordinates that are customized for the object that is being rendered, they can be acceptable in some cases.

Generated texture coordinates should usually be computed from the object coordinates of the object that is being rendered. That is, they are computed from the original vertex coordinates, before any transformation has been applied. Then, when the object is transformed, the texture will be transformed along with the object so that it will look like the texture is attached to the object. The texture coordinates could be almost any function of the object coordinates. If an affine function is used, as is usually the case, then the texture coordinates can be computed in the vertex shader. Otherwise, you need to send the object coordinates to the fragment shader in a varying variable and do the computation there.

The simplest idea for generated texture coordinates is simply to use the *x* and *y* coordinates from the object coordinate system as the texture coordinates. If the vertex coordinates are given as the value of the attribute variable *a\_coords*, that would mean using *a\_coords.xy* as texture coordinates. This has the effect of projecting the texture onto the surface from the direction of the positive *z*-axis, perpendicular to the *xy*-plane. The mapping works OK for a polygon that is facing, more-or-less, in the direction of positive *z*, but it doesn't give good results for polygons that are edge-on to the *xy*-plane.

When using flat shading, so that all of the normals to a polygon point in the same direction, the computation can be done in the vertex shader. With smooth shading, normals at different vertices of a polygon can point in different directions. If you project texture coordinates from different directions at different vertices and interpolate the results, the result is likely to be a mess. So, doing the computation in the fragment shader is safer. Suppose that the interpolated normal vectors and object coordinates are provided to the fragment shader in varying variables named *v\_normal* and *v\_objCoords*.

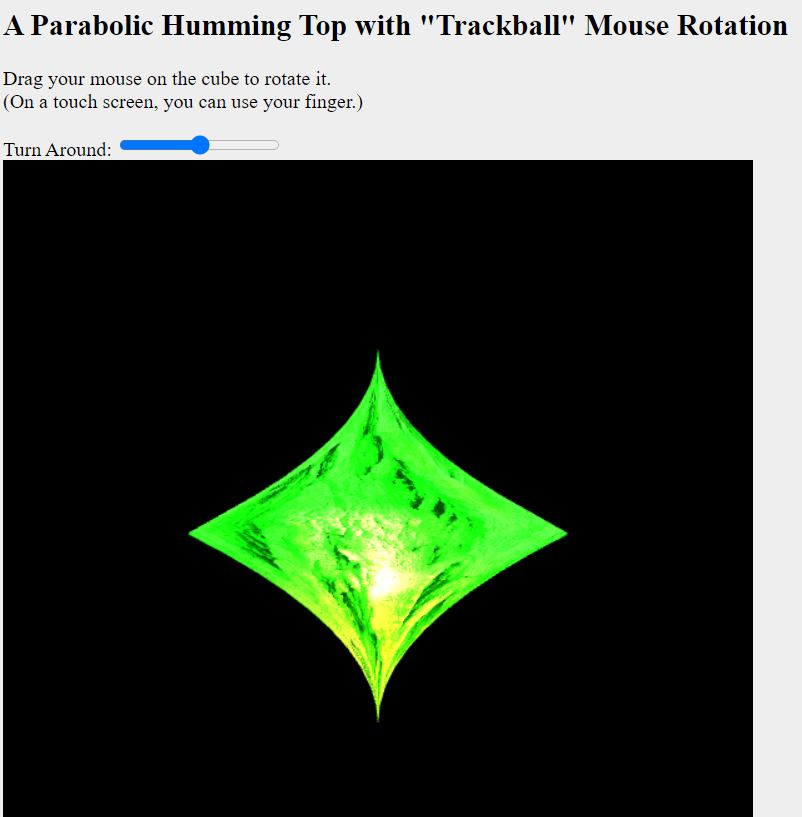
When projecting along the *x*-axis, for example, the *y* and *z* coordinates from *v\_objCoords* are used as texture coordinates. The coordinates are computed as either *v\_objCoords.yz* or *v\_objCoords.zy*, depending on whether the projection is from the positive or the negative direction of *x*. The order of the two coordinates is chosen so that a texture image will be projected directly onto the surface, rather than mirror-reversed.

In WebGL, procedural textures can be defined in the fragment shader. The idea is simple: Take a *vec2* representing a set of texture coordinates. Then, instead of using a *sampler2D* to look up a color, use the *vec2* as input to some mathematical computation that computes a *vec4* representing a color. In theory any computation could be used, as long as the components of the *vec4* are in the range 0.0 to 1.0.

We can even extend the idea to 3D textures. 2D textures use a *vec2* as texture coordinates. For 3D texture coordinates, we use a *vec3*. Instead of mapping points on a plane to color, a 3D texture maps points in space to colors. It's possible to have 3D textures that are similar to image textures. That is, a color value is stored for each point in a 3D grid, and the texture is sampled by looking up colors in the grid. However, a 3D grid of colors takes up a lot of memory. On the other hand, 3D procedural textures use no memory resources and use very little more computational resources than 2D procedural textures.

Chapter of user's instruction with screenshots

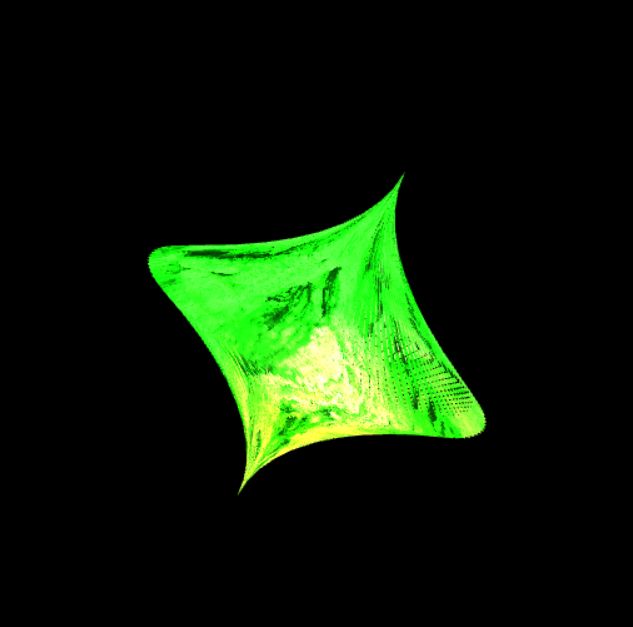
When you get to the main screen of the running program, you can see the canvas window with the corresponding shape:



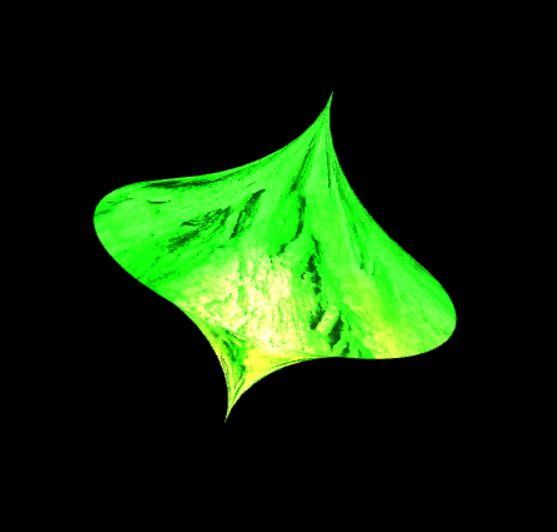
Using a standard input field “Turn Around” with a range type, we can rotate the texture around accordingly:

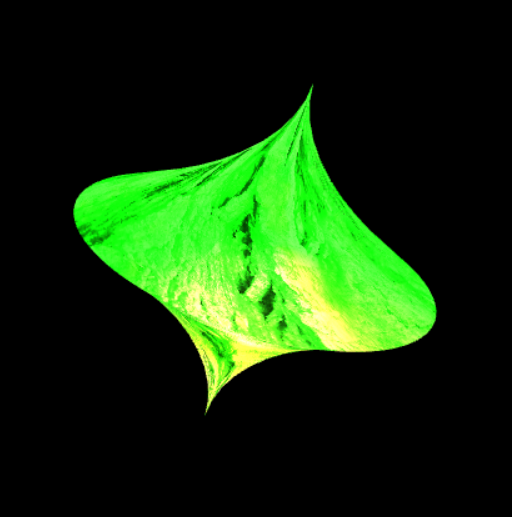


Use the mouse to rotate the figure:



Using the WASD keys, we change the position of the texture on the figure and see the corresponding results below:





Source code:

function CreateSurfaceData()

{

    let vertexList = [];

    let textureList = [];

    let splines = 100;

    let zStep = 0.1;

    let BStep = 0.1;

    let h = 1;

    let p = 0.5;

    let b = 360

    const step = (max, splines = 30) => {

        return max / (splines - 1);

    };

    let stepI = step(b, splines);

    let stepJ = step(h, splines);

     let getb = (i) => {

        return i / b;

    };

    let geth = (j) => {

        return j / h;

    };

    for  (let B = 0; B <= b; B += BStep) {

         for (let z = -h; z <= h; z += zStep) {

            vertexList.push(

                (((Math.pow(Math.abs(z) - h, 2))/(2\*p))\*Math.cos(deg2rad(B))),

                (((Math.pow(Math.abs(z) - h, 2))/(2\*p))\*Math.sin(deg2rad(B))),

                z

            );

            textureList.push(getb(B), geth(z));

            vertexList.push(

                (((Math.pow(Math.abs(z + stepJ) - h, 2))/(2\*p))\*Math.cos(deg2rad(B + stepI))),

                (((Math.pow(Math.abs(z + stepJ) - h, 2))/(2\*p))\*Math.sin(deg2rad(B + stepI))),

                z

            );

            textureList.push(getb(B + stepI), geth(z + stepJ))

        }

    }

    return {vertexList,textureList};

};