SPRAWOZDANIE
Zajęcia: Grafika komputerowa
Prowadzący: prof. dr. hab. Vasyl Martsenyuk

Laboratorium 10
06 V 2021 r.
Temat: „Podstawy WebGL/GLSL”
Wariant:
Liczba kątów: 5

Robert Laszczak
Informatyka I stopień
Stacjonarne, 4 semestr
Grupa 2B

1. Polecenie

Program Program w lab11.html pokazuje wiele ruchomych czerwonych kwadratów, które odbijają się od krawędzi płótna. Płótno wypełnia cały obszar zawartości przeglądarki internetowej. Kwadraty odpowiadają również myszą:
Jeśli klikniesz lewym przyciskiem myszy lub klikniesz lewym przyciskiem myszy i przeciągniesz na płótnie, cały kwadrat będzie kierowany w stronę pozycji myszy. Jeśli klikniesz lewym przyciskiem myszy, dane punktów zostaną ponownie zainicjowane, więc zaczną się od środka. Możesz wstrzymać i ponownie uruchomić animację, naciskając spację.
Kwadraty są w rzeczywistości częścią jednego prymitywu WebGL typu \texttt{gl.POINTS}. Każdy kwadrat odpowiada jednemu z wierzchołków pierwotnego. Oczywiście renderowanie jest wykonywane przez moduł shadera wierzchołków i moduł shadera fragmentu. Kod źródłowy shaderów jest w dwóch fałszywych „skryptach" w górnej części pliku html.
Będziesz modyfikował kod modułu shadera i kod JavaScript, aby zaimplementować kilka różnych stylów dla prymitywu punktu. Na przykład możliwe będzie rysowanie kwadratów w różnych kolorach, rysowanie wielokątów zamiast kwadratów i tak dalej. Użytkownik będzie kontrolował program, naciskając klawisze na klawiaturze. Do ciebie należy decyzja, których klawiszy użyć, ale proszę udokumentować interfejs w odpowiednim komentarzu do funkcji doKey() lub na górze programu.
Program ma dwie funkcje, nad którymi będziesz musiał pracować: Funkcja initGL() jest wywoływana, gdy program jest uruchamiany po raz pierwszy, a funkcje updateForFrame() i render() są wywoływane dla każdej ramki animacji. Ten sam zestaw poleceń byłby legalny we wszystkich tych poleceniach, ale initGL() jest najlepszym miejscem do ustawiania rzeczy, które nie zmienią się w trakcie działania programu, takich jak położenie zmiennych i zmiennych atrybutów w module shadera; updateForFrame() jest przeznaczony do aktualizacji zmiennych JavaScript, które zmieniają się z ramki na ramkę; i render() ma na celu wykonanie rzeczywistego rysunku WebGL ramki. Atrybut koloru
W oryginalnej wersji programu wszystkie kwadraty są czerwone. Pierwsze ćwiczenie polega na umożliwieniu przypisania innego koloru do każdego kwadratu. Ponieważ kwadraty są naprawdę wierzchołkami w pojedynczym prymitywie typu gl.POINTS, można użyć zmiennej atrybutu dla koloru. Atrybut może mieć inną wartość dla każdego wierzchołka.
Pierwszym zadaniem jest dodanie zmiennej kolorowej typu vec3 do modułu shadera wierzchołka i użycie wartości atrybutu do pokolorowania kwadratów. Będziesz także musiał pracować po stronie JavaScript. Będziesz potrzebował Float32Array do przechowywania wartości kolorów po stronie JavaScript, a będziesz potrzebował bufora WebGL dla tego atrybutu. Program ma już jeden atrybut, który jest używany do współrzędnych wierzchołków. Będziesz robił coś podobnego do atrybutu color (poza tym, że możesz to zrobić w initGL(), ponieważ wartości kolorów nie zmienią się po ich utworzeniu). Można użyć losowych wartości w zakresie od 0,0 do 1,0 dla składników koloru.
Po uruchomieniu wielokolorowych kwadratów powinieneś ustawić kolory jako opcjonalne. Możesz włączać i wyłączać użycie tablicy wartości atrybutów za pomocą następujących poleceń, gdzie a_color_loc to identyfikator atrybutu color w programie shader:

```
gl.enableVertexAttribArray (a_color_loc); // użyj bufora atrybutów kolorów
gl.disableVertexAttribArray (a_color_loc); // nie używaj bufora
```

Gdy tablica atrybutów jest włączona, każdy wierzchołek otrzymuje swój własny kolor z buforu atrybutów. Gdy tablica atrybutów jest wyłączona, wszystkie wierzchołki otrzymują ten sam kolor, a tę wartość można ustawić za pomocą rodziny funkcji gl.vertexAttrib *. Na przykład, aby ustawić wartość używaną, gdy tablica atrybutów kolorów jest wyłączona, można użyć

gl.vertexAttrib3f (a_color_loc, 1, 0, 0); // ustaw kolor attrbute na czerwony

Pozwól użytkownikowi na naciśnięcie określonego klawisza, aby włączyć lub wyłączyć losowe kolory. Program ma funkcję doKey(), która jest już skonfigurowana do reagowania na wprowadzanie z klawiatury. Będziesz dodawać do programu kilka typów interakcji z klawiaturą. Aby odpowiedzieć na klawiszę, musisz znać numeryczny kod klawiszy. Funkcja doKey() wysyła kod do konsoli za każdym razem, gdy użytkownik uderza klawisz, i możesz użyć tej funkcji, aby odkryć wszystkie inne kody klawiszy, których potrzebujesz.

Styl punktów
Powinieneś dodać opcję używania stylu wyświetlania dla punktów w postaci wielokąta. Pozwól użytkownikowi wybrać styl za pomocą klawiatury; na przykład, naciskając klawisze numeryczne. Style będą musiały zostać zaimplementowane w shaderze fragmentu, a będziesz potrzebował nowej zmiennej jednolitej, aby powiedzieć modułowi shadera fragmentu, którego stylu użyć. Dodaj jednolitą zmienną typu int do shadera fragmentu, aby kontrolować styl punktu, i dodaj kod do modułu cieniującego fragmentu, aby zaimplementować różne style. Będziesz także musiał dodać zmienną po stronie JavaScript dla lokalizacji zmiennej jednolitej, a będziesz musiał wywołać glUniform1i, gdy chcesz zmienić styl.

Naprzykład, żeby narysować punkt jako dysk, odrzucając niektóre piksele:

```
float dist = distance( vec2(0.5), gl_PointCoord );
if (dist > 0.5) {
discard;
}
```

Powinieneś również wykorzystać przezroczystość alfa w niektórych stylach. Aby umożliwić korzystanie ze składnika alpha, musisz dodać następujące linie do funkcji initGL ():

```
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

Dzięki tym ustawieniom wartość alfa koloru będzie używana do przezroczystości w zwykły sposób. W szczególności jeden z twoich stylów powinien pokazywać punkt jako wielokąt, który zanika z całkowicie nieprzezroczystego w środku wielokątu do całkowicie przezroczystego na krawędzi.

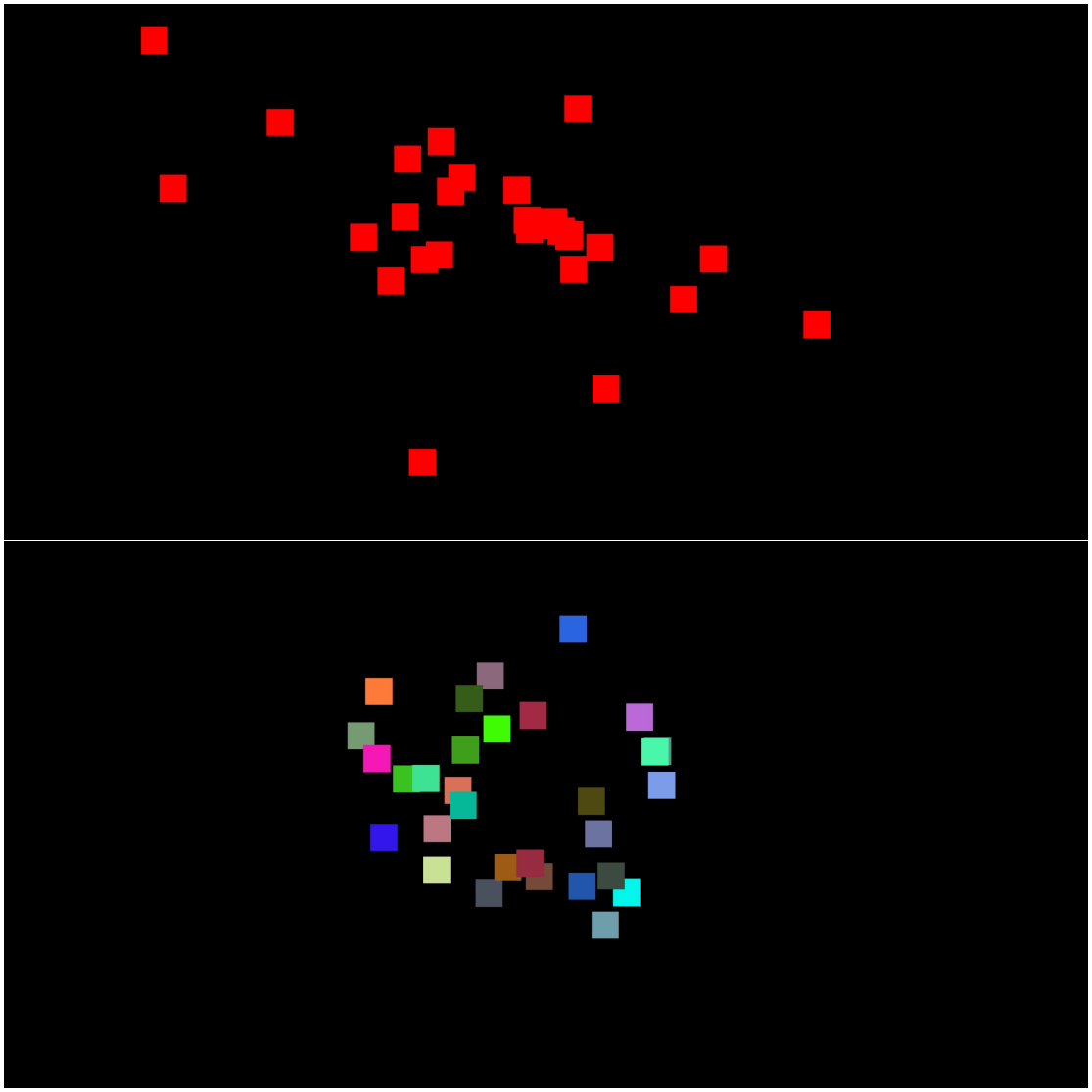
2. Wprowadzam dane:

Liczba kątów - 5

3. Wykorzystane komendy:

Link do zdalnego repozytorium:
https://github.com/RLaszczak/Lab_GK

4. Wyniki działania:



5. Wnioski

Na podstawie otrzymanego wyniku można stwierdzić, że:

- Możemy ustawiać ilość boków wielokątów,
- Możemy sterować kolorami wielokątów.

Kod:

```
<!DOCTYPE html>
<meta charset="UTF-8">
<html>
<head>
<title>WebGL Intro</title>
<style>
   html, body {
      margin: 0;  /* Make sure that there is no margin around the canvas */
      overflow: hidden;  /* Make sure we don't get scroll bars. */
   }
   canvas {
      display: block; /* The default display, inline, would add a small margin below the canvas */
   }
</style>
<!--
   A 2D WebGL app in which "points" move around in the browser window, bouncing
   off the edges.  The animation can be paused and restarted by pressing the
   space key.
      If the user clicks or clicks-and-drags with the mouse, all of the
   points head towards the mouse position, except if the user shift-clicks, the
   positions and velocities of the points are re-initialized.
-->

<script type="x-shader/x-vertex" id="vshader-source"> //shadery
   attribute vec2 a_coords; // vertex position in standard canvas pixel coords
   attribute vec3 color;
   uniform float u_width;   // width of canvas
   uniform float u_height;  // height of canvas
   uniform float u_pointSize;
   uniform int u_type;
   varying vec3 outcolor;
   varying float type;
   void main() {
      float x,y;  // vertex position in clip coordinates
      x = a_coords.x/u_width * 2.0 - 1.0;  // convert pixel coords to clip coords
      y = 1.0 - a_coords.y/u_height * 2.0;
      gl_Position = vec4(x, y, 0.0, 1.0);
```

```
        gl_PointSize = u_pointSize;
        outcolor = vec3(color);
        type = float(u_type);
    }
</script>
<script type="x-shader/x-fragment" id="fshader-source"> //shadery fragmentów
    #ifdef GL_FRAGMENT_PRECISION_HIGH
      precision highp float;
    #else
      precision mediump float;
    #endif
    varying vec3 outcolor;
    varying float type;
    const float pi=3.141592653589793;
    float polygon(float s, float apotheme, vec2 p){
        float ang=atan(p.x,p.y);
        ang-=floor(ang/pi/5.*s)/s*pi*5.-pi/s;
        return cos(atan(p.x,p.y)-floor(atan(p.x,p.y)/pi/2.*s)/s*pi*2.-pi/s)*length(p)<apotheme?1.:0.;
    }

    void main() {
        float dist = distance( vec2(0.5), gl_PointCoord );
        gl_FragColor = vec4(outcolor, 1.0);
        if ( type > 4.0 ){
            if ( dist > polygon( type , 0.4, vec2(gl_PointCoord.x - 0.5, gl_PointCoord.y- 0.5))) {
                discard;
            }
        }
    }
</script>
<script>
"use strict";

var canvas; // The canvas that is used for WebGL drawing; occupies the entire window.
var gl;     // The webgl context.
var u_width_loc;        // Location of "width" uniform, which holds the width of the canvas.
var u_height_loc;       // Location of "height" uniform, which holds the height of the canvas.
var u_pointSize_loc;    // Location of "pointSize" uniform, which gives the size for point primitives.
var a_coords_loc;       // Location of the a_coords attribute variable in the shader program;
var a_color_loc;       // This attribute gives the (x,y) coordinates of the points
var a_coords_buffer;    // Buffer to hold the values for a_coords (coordinates for the points)
var a_color_buffer;
var u_type_loc;
var POINT_COUNT = 30;   // How many points to draw.
var POINT_SIZE = 64;    // Size in pixel of the square drawn for each point.
var nSides = 5;
var positions = new Float32Array( 2*POINT_COUNT );  // Position data for points.
var velocities = new Float32Array( 2*POINT_COUNT );
var color = new Float32Array( 3*POINT_COUNT );
 // Velocity data for points.
      // Note: The xy coords for point number i are in positions[2*i],position[2*i+1].
      // The xy velocity compontents for point number i are in velocities[2*i],velociteis[2*i+1].
```

```
      // Position coordinates are in pixels, and velocity components are in pixels per frame.

var isRunning = true;  // The animation runs when this is true; its value is toggled by the space bar.

function SetRandomColor(){
        for (let i = 0; i < color.length; i++) {
            color[i] = Math.random();
        }
 }

function changeType(){
              var is = false;
              var num;
    num = prompt("Jaki wielokąt chcesz zobaczyć?", "4");
               nSides = parseInt(num);
    gl.uniform1i(u_type_loc, nSides);
}

var isColorRandom = true;
/**
 *  Called by init() when the window is first opened, and by frame() to render each frame.
 */

function render() {
    gl.clear(gl.COLOR_BUFFER_BIT);  // clear the color buffer before drawing
    // The position data changes for each frame, so we have to send the new values
    // for the position attirbute into the corresponding buffer in the GPU here,
    // in every frame.

    gl.bindBuffer(gl.ARRAY_BUFFER, a_coords_buffer);            // Select the buffer we want to use.
    gl.bufferData(gl.ARRAY_BUFFER, positions, gl.STREAM_DRAW);  // Send the data.
    gl.vertexAttribPointer(a_coords_loc, 2, gl.FLOAT, false, 0, 0);  // Describes the data format.

    if ( isColorRandom ){
        gl.enableVertexAttribArray(a_color_loc);
    } else {
        gl.disableVertexAttribArray(a_color_loc);
        gl.vertexAttrib3f (a_color_loc, 1, 0, 0)
    }
    // Now, draw the points as a primitive of type gl.POINTS

 gl.drawArrays(gl.POINTS, 0, POINT_COUNT);
    if (gl.getError() != gl.NO_ERROR) {
        console.log("During render, a GL error has been detected.");
    }
} // end render()

/**
 * Called once in init() to create the data for the scene. Creates point positions and
 * velocities.  All points start at the center of the canvas, with random velocity.
 * The speed is between 2 and 6 pixels per frame.
 */
```

```
function createData() {
    SetRandomColor();
    for (var i = 0; i < POINT_COUNT; i++) {
        positions[2*i] = canvas.width/2;
        positions[2*i+1] = canvas.height/2;
        var speed = 2 + 4*Math.random();
        var angle = 2*Math.PI*Math.random();
        velocities[2*i] = speed*Math.sin(angle);
        velocities[2*i+1] = speed*Math.cos(angle);
    }
} // end createData()

/**
 * Called by frame() before each frame is rendered.  Adds velcities
 * to point positions.  If the point moves past the edge of the canvas,
 * it bounces.
 */
function updateData() {
    for (var i = 0; i < POINT_COUNT; i++) {
       positions[2*i] += velocities[2*i];
       if ( positions[2*i] < POINT_SIZE/2 && velocities[2*i] < 0) {
           positions[2*i] += 2*(POINT_SIZE/2 - positions[2*i]);
           velocities[2*i] = Math.abs(velocities[2*i]);
       }

       else if (positions[2*i] > canvas.width - POINT_SIZE/2 && velocities[2*i] > 0){
           positions[2*i] -= 2*(positions[2*i] - canvas.width + POINT_SIZE/2);
           velocities[2*i] = - Math.abs(velocities[2*i]);
       }

       positions[2*i+1] += velocities[2*i+1];
       if ( positions[2*i+1] < POINT_SIZE/2  && velocities[2*i+1] < 0) {
           positions[2*i+1] += 2*(POINT_SIZE/2 - positions[2*i+1]);
           velocities[2*i+1] = Math.abs(velocities[2*i+1]);
       }

       else if (positions[2*i+1] > canvas.height - POINT_SIZE/2  && velocities[2*i+1] > 0){
           positions[2*i+1] -= 2*(positions[2*i+1] - canvas.height + POINT_SIZE/2);
           velocities[2*i+1] = - Math.abs(velocities[2*i+1]);
       }
    }
} // end updateData()

/* Called when the user hits a key */
function doKey(evt) {
    var key = evt.keyCode;
    console.log("key pressed with keycode = " + key);
    if ( key == 49){
        isColorRandom == false ? isColorRandom = true : isColorRandom = false;
    }
```

```
    if ( key == 50) {
        changeType();
    }

    if (key == 32) {  // space bar
        if (isRunning) {
            isRunning = false;  // stops the animation
        }
        else {
            isRunning = true;
           requestAnimationFrame(frame);  // restart the animation
        }
    }
} // end doKey();

/* Initialize the WebGL context.  Called from init() */
function initGL() {
    var prog = createProgram(gl,"vshader-source", "fshader-source", "a_coords");
    gl.useProgram(prog);

    /* Get locations of uniforms and attributes. */

    u_width_loc = gl.getUniformLocation(prog,"u_width");
    u_height_loc = gl.getUniformLocation(prog,"u_height");
    u_pointSize_loc = gl.getUniformLocation(prog,"u_pointSize");
    a_coords_loc = gl.getAttribLocation(prog,"a_coords");
    a_color_loc = gl.getAttribLocation(prog, "color");
    u_type_loc = gl.getUniformLocation(prog, "u_type");

    /* Assign initial values to uniforms. */
    gl.uniform1f(u_width_loc, canvas.width);
    gl.uniform1f(u_height_loc, canvas.height);
    gl.uniform1f(u_pointSize_loc, POINT_SIZE);

    /* Create and configure buffers for the attributes. */
    a_coords_buffer = gl.createBuffer();
    gl.enableVertexAttribArray(a_coords_loc); // data from the attribute will come from a buffer.
    a_color_buffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, a_color_buffer);
    gl.bufferData(gl.ARRAY_BUFFER, color, gl.STATIC_DRAW);
    gl.vertexAttribPointer(
        a_color_loc,
        3,
        gl.FLOAT,
        false,
        0,
        0);

    /* Configure other WebGL options. */
    gl.clearColor(0,0,0,1);  // gl.clear will fill canvas with black.
    if (gl.getError() != gl.NO_ERROR) {
        console.log("During initialization, a GL error has been detected.");
```

```
    }
} // end initGL()
/**
 * Creates a program for use in the WebGL context gl, and returns the
 * identifier for that program.  If an error occurs while compiling or
 * linking the program, an exception of type String is thrown.  The error
 * string contains the compilation or linking error.  If no error occurs,
 * the program identifier is the return value of the function.
 * The second and third parameters are the id attributes for <script>
 * elements that contain the source code for the vertex and fragment
 * shaders.
 *     If the third parameter is present, it should be the name of an
 * attribute variable in the shader program, and the attribute should be
 * one that is always used.  The attribute will be assigned attribute
 * number 0.  This is done because it is suggested that there should
 * always be an attribute number 0 in use.
 */

function createProgram(gl, vertexShaderID, fragmentShaderID, attribute0) {
    function getTextContent( elementID ) {
            // This nested function retrieves the text content of an
            // element on the web page.  It is used here to get the shader
            // source code from the script elements that contain it.
        var element = document.getElementById(elementID);
        var node = element.firstChild;
        var str = "";
        while (node) {
            if (node.nodeType == 3) // this is a text node
                str += node.textContent;
            node = node.nextSibling;
        }
        return str;
    }
    try {
        var vertexShaderSource = getTextContent( vertexShaderID );
        var fragmentShaderSource = getTextContent( fragmentShaderID );
    }
    catch (e) {
        throw "Error: Could not get shader source code from script elements.";
    }
    var vsh = gl.createShader( gl.VERTEX_SHADER );
    gl.shaderSource(vsh,vertexShaderSource);
    gl.compileShader(vsh);
    if ( ! gl.getShaderParameter(vsh, gl.COMPILE_STATUS) ) {
        throw "Error in vertex shader:  " + gl.getShaderInfoLog(vsh);
     }
    var fsh = gl.createShader( gl.FRAGMENT_SHADER );
    gl.shaderSource(fsh, fragmentShaderSource);
    gl.compileShader(fsh);
    if ( ! gl.getShaderParameter(fsh, gl.COMPILE_STATUS) ) {
       throw "Error in fragment shader:  " + gl.getShaderInfoLog(fsh);
    }
```

```
    var prog = gl.createProgram();
    gl.attachShader(prog,vsh);
    gl.attachShader(prog, fsh);
    if (attribute0) {
      gl.bindAttribLocation(prog,0,attribute0);
    }
    gl.linkProgram(prog);
    if ( ! gl.getProgramParameter( prog, gl.LINK_STATUS) ) {
      throw "Link error in program:  " + gl.getProgramInfoLog(prog);
    }
    return prog;
}

/**
 *  A function to drive the animation, which runs continuously while the global
 *  variable isRunning is true.  The value of this variable is toggled by pressing
 *  the space bar.  If the animation is still running, this fucntion calls
 *  updateData(), then calls render(), then calls requestAnimationFrame to
 *  schedule the next call to the same function.
 */

function frame() {
    if (isRunning) {
       updateData();
       render();
       requestAnimationFrame(frame);  // Arrange for function to be called again
    }
}

/**
 * When the window is resized, we need to resize the canvas, reset the
 * OpenGL viewport to match the size, and reset the values of the uniform
 * variables in the shader that represent the canvas size.
 */
 function doResize() {
     canvas.width = window.innerWidth;
     canvas.height = window.innerHeight;
     gl.viewport(0, 0, canvas.width, canvas.height);
     gl.uniform1f(u_width_loc, canvas.width);
     gl.uniform1f(u_height_loc, canvas.height);
     if (!isRunning) {
         render();
     }
 }

/**
 * Responds to left mouse click on canvas; points all head toward mouse location
 * when mouse is clicked and as it is dragged.  However if shift key is down,
 * all the data is reinitialized instead.
 */
function doMouse(evt) {
    function headTowards(x,y) {
```

```
        for (var i = 0; i < POINT_COUNT; i++) {
            var dx = x - positions[2*i];
            var dy = y - positions[2*i+1];
            var dist = Math.sqrt(dx*dx + dy*dy);
            if (dist > 0.1) { // only if mouse and point are not too close.
                var speed = Math.sqrt( velocities[2*i]*velocities[2*i] + velocities[2*i+1]*velocities[2*i+1] );
                velocities[2*i] = dx/dist * speed;
                velocities[2*i+1] = dy/dist * speed;
            }
        }
    }
    function move(evt) {
        headTowards(evt.clientX,evt.clientY);
    }
    function up() {
        canvas.removeEventListener("mousemove", move, false);
        document.removeEventListener("mouseup", up, false);
    }
    if (evt.which != 1) {
        return;  // only respond to left mouse down
    }
    if (evt.shiftKey) {
        createData();
        return;
    }
    headTowards(evt.clientX,evt.clientY);
    canvas.addEventListener("mousemove", move);
    document.addEventListener("mouseup", up);
}

/**
 * initialization function that will be called when the page has loaded.
 */
function init() {
    try {
        canvas = document.createElement("canvas");
        canvas.width = window.innerWidth;
        canvas.height = window.innerHeight;
        var options = {
             alpha: false,   // The color buffer doesn't need an alpha component
             depth: false,   // No need for a depth buffer in this 2D program
             stencil: false  // This program doesn't use a stencil buffer
           };
        gl = canvas.getContext("webgl", options);
        if ( ! gl ) {
            throw "Browser does not support WebGL";
        }
    }
    catch (e) {
        var message = document.createElement("p");
        message.innerHTML = "Sorry, could not get a WebGL graphics context.  Error: " + e;
        document.body.appendChild(message);
```

```
        return;
    }
    try {
        createData();  // create data for points (in case it's needed in initGL())
        initGL();  // initialize the WebGL graphics context.
    }
    catch (e) {
        var message = document.createElement("p");
        message.innerHTML =
            "<pre>Sorry, could not initialize graphics context.  Error:\n\n" + e + "</pre>";
        document.body.appendChild(message);
        return;
    }
    document.body.appendChild(canvas);
    window.addEventListener("resize", doResize);
    canvas.addEventListener("mousedown",doMouse);
    document.addEventListener("keydown",doKey);
    requestAnimationFrame(frame);
}

</script>
</head>
<body onload="init()">
<noscript>Sorry, this page requires JavaScript.</noscript>
</body>
</html>
```