SPACE-INVADERS GAME LOGIC

Created by Chrinovic Raya Tshiwaya

[DATE]
[COMPANY NAME]
[Company address]

1 Utilities

- The code was written in webgl using javascript
- The code uses a scaling matrix and a translation matrix that are only applied to the ship
- The ship has its own vertex shader and fragement shader including it own program
- The bullets and aliens share the same vertex shader and fragement shader and the same program
- You control movement and shooting with keyboard keys

1. ALIENS

In this code, aliens are created and managed through several steps:

1. Definition of Alien Vertices : Initially, the vertices for a single alien (a square) are defined in the `singleAlienData` array. This array contains the coordinates that define the shape of the alien.

```
// Vertices for a single alien (square)
let singleAlienData = [
    0.5, 0.5, 0,
    -0.5, 0.5, 0,
    -0.5, -0.5, 0,
    -0.5, -0.5, 0,
    0.5, -0.5, 0,
    0.5, 0.5, 0,
];
```

2. Texture Coordinate: Texture coordinates are defined for mapping textures onto the alien shape. The `textureCoordinate` array holds these coordinates

// Texture coordinates for a single alien

```
const textureCoordinate = new Float32Array([
  1.0, 1.0, // Bottom right
  0.0, 1.0, // Bottom left
  0.0, 0.0, // Top left
  0.0, 0.0, // Top left
  1.0, 0.0, // Top right
  1.0, 1.0, // Bottom right
]);
```

.

3. Alien Creation: The `addAlien()` function is responsible for creating new aliens. It first defines the initial position for the new alien. Then, it checks if this position overlaps with any existing bullets. If there's no overlap, a new alien object is created and added to the `aliens` array.

```
let aliens = [];
function addAlien() {
    // Define the initial position of the new alien
    let newAlienPosition = [0.5, 0.5, 0];
```

To specify how long you have to wait for a another alien to appear.

```
// Add a new alien every second
setInterval(addAlien, 1000);
```

4. Alien Buffer: A buffer (`alienBuffer`) is created to store the vertex data of the aliens. This buffer is initialized with the vertex data of a single alien.

```
//Buffer for aliens
```

```
const alienBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, alienBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(singleAlienData),
gl.STATIC_DRAW);
```

5. Alien Texture: Similar to the alien buffer, a texture is created (`alienTexture`) to hold the image representing the alien. This texture is initialized using the image loaded from `alienImage`.

```
const alienTexture = initTexture(gl, alienImage);
```

6. Drawing Aliens: The `drawAlien()` function iterates over all aliens in the `aliens` array. For each alien, it updates the buffer data with the alien's vertex coordinates and texture coordinates. Then, it draws the alien using the provided shader program (`aleinProgam`).

```
// Modify the drawAlien function to draw all aliens
function drawAlien() {
    for (let alien of aliens) {
        // Update the buffer data for alien vertices
        gl.bindBuffer(gl.ARRAY_BUFFER, alienBuffer);
        gl.bindTexture(gl.TEXTURE_2D, alienTexture);
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(alien.data),
gl.STATIC_DRAW);
        gl.enableVertexAttribArray(alienLocation);
        gl.vertexAttribPointer(alienLocation, 3, gl.FLOAT, false, 0, 0);

        // Update the buffer data for texture coordinates
        gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordinateBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(alien.textureCoordinate),
gl.STATIC_DRAW);
        gl.enableVertexAttribArray(texCoordLocation);
        gl.vertexAttribPointer(texCoordLocation, 2, gl.FLOAT, false, 0, 0);
        // Draw the alien
        gl.drawArrays(gl.TRIANGLES, 0, 6);
    }
}
```

7. Alien Movement: The `updateAlein()` function is responsible for updating the position of aliens. It checks if the alien goes past the bottom of the canvas and stops the animation if it does. It also handles the horizontal movement of aliens, reversing their direction if they reach the canvas boundaries.
   The let velocity specifies how fast the aliens move in the x axis.
   In the third for-loop the way the alein move when they hit that canvas is specified in these lines.

```
let velocity = [0.0, 0.5, 0.0];
function updateAlein() {
    for (let alien of aliens) {
        if (alien.data[1] < -8) {
            // If alien goes past the bottom of the canvas, stop the animation
            // document.getElementById(".game-over").style.display = "flex";
            let gamestats = document.getElementById("gamestats");
            gamestats.style.display = 'flex';
            console.log(gamestats);
            console.log('hello');
            return;
        }

        if (alien.data[0] + alien.velocity[0] > 5.2 || alien.data[0] +
alien.velocity[0] < -4.2) {
            // Reverse the direction of the alien's velocity in the x-axis
            alien.velocity[0] = -alien.velocity[0];

            // Move down by a unit in the y-axis
            for (let j = 1; j < 18; j += 3) {
                alien.data[j] -= 0.5;
            }
        }
        // Update the point's position
        for (let j = 0; j < 18; j += 3) {
            alien.data[j] += alien.velocity[0];
        }
    }
}
```

8. Collision Detection: The `checkCollisions()` function checks for collisions between bullets and aliens. If a bullet hits an alien, both the bullet and the alien are removed from their respective arrays.

```
// Check collisions and remove bullets that hit aliens
function checkCollisions() {
    let bulletsToRemove = [];
    let aliensToRemove = [];

    for (let i = 0; i < bulletData.length; i += 3) {
        for (let j = 0; j < aliens.length; j++) {
            let alienLeft = aliens[j].data[0] - 0.5;
            let alienRight = aliens[j].data[0] + 0.5;

            if (bulletData[i] >= alienLeft && bulletData[i] <= alienRight) {
                bulletsToRemove.push(i);
                aliensToRemove.push(j);
                break;
            }
        }
    }
    // Sort the indices in descending order
    bulletsToRemove.sort((a, b) => b - a);
    aliensToRemove.sort((a, b) => b - a);
    // Remove the marked bullets and aliens
    for (let i of bulletsToRemove) {
        bulletData.splice(i, 3);
    }
    for (let j of aliensToRemove) {
        aliens.splice(j, 1);
    }
}
```

These steps together facilitate the creation, rendering, and movement of aliens within the WebGL environment.

SPACESHIP

To create and render the ship, the following steps are involved:

1. Definition of Ship Vertices: Define the vertices for the ship. This typically involves defining a polygon or a set of vertices that represent the ship's shape.

```
// Vertices for static shapes
let shapeData = [

    0.5, 0.5, 0,
    -0.5, 0.5, 0,
    -0.5, -0.5, 0,
    -0.5, -0.5, 0,
    0.5, -0.5, 0,
    0.5, 0.5, 0,

];
```

2. Texture Coordinates: Similar to aliens, you may want to map a texture onto the ship. Texture coordinates are necessary for this mapping.

```

function initTexture(gl, image) {

    const texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true); // This flips the image
orientation to be upright.

    if (isPowerOfTwo(image.width) && isPowerOfTwo(image.height)) {
        gl.generateMipmap(gl.TEXTURE_2D);
    } else {
        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);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
    }
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);

    return texture;
}


const alienTexture = initTexture(gl, alienImage);
const shipTexture = initTexture(gl, shipImage);
```

3. Buffer Creation: Create a buffer to hold the ship's vertex data.

```
// Buffer for static shapes

const shapeBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, shapeBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(shapeData), gl.STATIC_DRAW);
```

4. Texture Initialization: Load and initialize a texture for the ship.

```
  gl.bindTexture(gl.TEXTURE_2D, shipTexture);
```

5. Shader Definition: Define a vertex shader and a fragment shader specifically tailored for rendering the ship.

```
// shape and bullets vertex shader
const vsSource = `
    precision mediump float;
    attribute vec3 pos;

    attribute vec3 aPos;
    attribute vec2 stexCoord;
    varying vec2 vTexCoord;

    uniform mat4 u_ScaleMatrix;
    uniform mat4 u_TranslateMatrix;

    void main() {
        vTexCoord = stexCoord;
        gl_Position = u_TranslateMatrix * u_ScaleMatrix * vec4(pos, 1.0);
        gl_PointSize = 25.0;
    }
`;

// shape and bullets fragment shader
const fsSource = `
    precision mediump float;
    varying vec2 vTexCoord;
    uniform sampler2D texture;

    void main() {
        //gl_FragColor = vec4(0.2, 0.3, 0.8, 1);
        gl_FragColor = texture2D(texture, vTexCoord);
    }
`;
```

6. Drawing the Ship: Implement the logic to draw the ship using WebGL.

```
// Draw the static shapes
function drawShapes() {
    gl.bindBuffer(gl.ARRAY_BUFFER, shapeBuffer);
    gl.enableVertexAttribArray(positionLocation);
    gl.vertexAttribPointer(positionLocation, 3, gl.FLOAT, false, 0, 0);
    gl.bindBuffer(gl.ARRAY_BUFFER, shipTextureCoordinateBuffer);
    gl.bindTexture(gl.TEXTURE_2D, shipTexture);

    gl.enableVertexAttribArray(shipTexCoordLocation);
    gl.bindBuffer(gl.ARRAY_BUFFER, shipTextureCoordinateBuffer);
    gl.vertexAttribPointer(shipTexCoordLocation, 2, gl.FLOAT, false, 0, 0);

    gl.drawArrays(gl.TRIANGLES, 0, shapeData.length / 3);
}
```

7. movement and shooting of the ship using keyboard button

```
document.addEventListener('keydown', function(event) {
    if (event.key === " ") {
        bulletData.push(0, 0.01, 0);
    }
});

document.addEventListener('keydown', function(event) {
    switch (event.key) {
        case "ArrowLeft":
            translatedMatrix[12] -= 0.1;
            break;
        case "ArrowRight":
            translatedMatrix[12] += 0.1;
            break;

    }
});

// checks if its to power of two
function isPowerOfTwo(value) {
    return (value & (value - 1)) === 0;
}
```

This setup will allow you to define, render, and texture the ship within your WebGL environment.

Bullets

To create and render bullets in the WebGL environment, the following steps are involved:

1. **Definition of Bullet Vertices**: Define the vertices for the bullet. This typically involves specifying a single point representing the bullet.

2. **Buffer Creation**: Create a buffer to hold the bullet's vertex data.

3. **Shader Definition**: Define a vertex shader and a fragment shader specifically tailored for rendering the bullet.

4. **Drawing the Bullets**: Implement the logic to draw the bullets using WebGL.

Here's the relevant code for the bullet creation and rendering:

```javascript
// Vertices for bullets (a single point)
let bulletData = [];

// Buffer for bullets
const bulletBuffer = gl.createBuffer();
```

```
gl.bindBuffer(gl.ARRAY_BUFFER, bulletBuffer);

gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(bulletData), gl.DYNAMIC_DRAW);


// Vertex shader for bullets

const bulletVertexShader = gl.createShader(gl.VERTEX_SHADER);

gl.shaderSource(bulletVertexShader, vsSource); // Using the same vertex shader as shapes

gl.compileShader(bulletVertexShader);


// Fragment shader for bullets

const bulletFragmentShader = gl.createShader(gl.FRAGMENT_SHADER);

gl.shaderSource(bulletFragmentShader, fsSource); // Using the same fragment shader as
shapes

gl.compileShader(bulletFragmentShader);


// Program for bullets

const bulletProgram = gl.createProgram();

gl.attachShader(bulletProgram, bulletVertexShader);

gl.attachShader(bulletProgram, bulletFragmentShader);

gl.linkProgram(bulletProgram);


// Get the attribute location for the bullet position

const bulletPositionLocation = gl.getAttribLocation(bulletProgram, "pos");


// Draw the bullets

function drawBullets() {

  // Update the buffer data for bullets
  gl.bindBuffer(gl.ARRAY_BUFFER, bulletBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(bulletData), gl.DYNAMIC_DRAW);

  // Apply any necessary transformations

  // Enable attribute array for bullets
  gl.enableVertexAttribArray(bulletPositionLocation);

  // Specify attribute pointer for bullets
  gl.vertexAttribPointer(bulletPositionLocation, 3, gl.FLOAT, false, 0, 0);

  // Draw the bullets (as points)
  gl.drawArrays(gl.POINTS, 0, bulletData.length / 3);
}
```

This setup allows you to define, render, and update the positions of bullets within your WebGL environment.

The shader code for bullets (which is already defined in the `shaders.js` file) can be reused, as it's the same as the one used for shapes:

```javascript
// Vertex shader for bullets (same as shapes)
const vsSource = `
  precision mediump float;
  attribute vec3 pos;
```

~~~~
  void main() {
    gl_Position = vec4(pos, 1.0);
    gl_PointSize = 25.0; // Adjust point size as needed
  }
`;

// Fragment shader for bullets (same as shapes)
const fsSource = `
  precision mediump float;

  void main() {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0); // White color for bullets
  }
`;
```
~~~~

This shader code will render bullets as white points on the canvas. You can customize the appearance by modifying the fragment shader code.