

**Jashore University of Science and Technology**

**Department of Computer Science and Engineering**

**Course Title:** Computer Graphics Lab

**Course Code:** CSE-4104

**Project Report on**

**Development of a Classic Snake Game Using WebGL**

|  |  |
| --- | --- |
| **Submitted to** | **Submitted by** |
| **Mostafijur Rahman Akond**  **Assistant Professor**  Dept. of Computer Science and Engineering.  Jashore University of Science and Technology. | **Farhana Jesmin**  Student ID: 190151  4th year, 1st semester  Dept. of Computer Science and Engineering  Jashore University of Science and Technology. |

**Remarks:**

Date of Submission: 07-10-2024

## Table of Contents

1. [Abstract](#abstract)
2. [Introduction](#introduction)
3. [Objective](#objective)
4. [Materials and Methods](#materials-and-methods)
   * [Technologies Used](#technologies-used)
   * [Game Design and Mechanics](#game-design-and-mechanics)
   * [Rendering with WebGL](#rendering-with-webgl)
   * [User Input Handling](#user-input-handling)
   * [Game Loop and Logic](#game-loop-and-logic)
5. [Implementation](#implementation)
   * [Initial Setup](#initial-setup)
   * [Shader Programs](#shader-programs)
   * [Buffer Initialization](#buffer-initialization)
   * [Game State Management](#game-state-management)
   * [Rendering Functions](#rendering-functions)
   * [Collision Detection and Game Over Handling](#collision-detection-and-game-over-handl)
   * [Enhancements and Features](#enhancements-and-features)
6. [Results](#results)
7. [Discussion](#discussion)
   * [Challenges Faced](#challenges-faced)
   * [Enhancements and Future Work](#enhancements-and-future-work)
8. [Conclusion](#conclusion)
9. [References](#references)

## Abstract

This report details the development of a classic Snake game inspired by the Nokia-1100 version, implemented using HTML5, JavaScript, and WebGL for rendering. The game features keyboard-controlled movement, dynamic snake growth, collision detection, and real-time score tracking. Enhancements include color-changing mechanics upon consuming food and responsive game-over handling with restart capabilities. The project demonstrates the integration of WebGL for 2D game development, efficient handling of user inputs, and real-time game state management.

## Introduction

The Snake game is a quintessential example of early mobile gaming, beloved for its simplicity and engaging mechanics. This project aims to recreate the classic Snake game using modern web technologies, specifically leveraging WebGL for rendering graphics. The choice of WebGL provides a foundation for understanding low-level graphics programming within a web environment, offering opportunities to explore shader programming and GPU-accelerated rendering.

## Objective

The primary objective of this project is to develop a web-based Snake game that emulates the gameplay experience of the classic Nokia-1100 version. Key goals include:

* Implementing responsive and intuitive keyboard controls for snake navigation.
* Ensuring consistent and smooth snake movement with fixed speed and direction.
* Incorporating collision detection mechanisms for walls and self-collisions.
* Providing visual feedback through dynamic color changes and real-time score updates.
* Facilitating game-over scenarios with user-friendly restart options.

## Materials and Methods

### Technologies Used

* **HTML5**: For structuring the game interface.
* **CSS3**: For styling the game canvas and UI elements.
* **JavaScript**: For game logic, state management, and user input handling.
* **WebGL**: For rendering 2D graphics, utilizing shader programs for efficient drawing operations.

### Game Design and Mechanics

The game revolves around controlling a snake that moves across a grid, consuming food (referred to as "frog") to grow in length. The player navigates the snake using arrow keys, avoiding collisions with walls and the snake's own body. Each successful consumption of food increments the score and alters the snake's color, enhancing visual feedback.

### Rendering with WebGL

WebGL, a JavaScript API for rendering interactive 2D and 3D graphics, is employed to draw the game elements. Although primarily used for 3D graphics, WebGL is adapted here for efficient 2D rendering by leveraging shader programs to draw rectangles representing the snake and food.

### User Input Handling

Keyboard event listeners capture arrow key presses, translating them into directional changes for the snake. Direction management ensures that the snake cannot reverse directly, maintaining consistent movement and preventing immediate self-collisions.

### Game Loop and Logic

A game loop, orchestrated using setInterval, governs the snake's movement, state updates, and rendering cycles. This loop ensures that the game progresses at a steady pace, with each iteration handling movement, collision detection, and rendering.

## Implementation

### Initial Setup

The game's foundation is laid out using an HTML5 canvas element, styled with CSS to centralize it on the page and provide visual boundaries. The canvas dimensions are set to 600x600 pixels, creating a grid-based play area divided into cells of 20x20 pixels each.

html

Copy code

<canvas id="glCanvas" width="600" height="600"></canvas>

<div id="score">Score: 0</div>

<div id="gameOver">Game Over!</div>

<div id="restart">

<button onclick="startGame()">Restart</button>

</div>

### Shader Programs

Two shader programs are defined:

1. **Vertex Shader**: Transforms vertex positions from pixel coordinates to WebGL clip space.
2. **Fragment Shader**: Determines the color of each pixel.

javascript

Copy code

const vsSource = `

attribute vec2 aPosition;

uniform vec2 uResolution;

void main() {

vec2 zeroToOne = aPosition / uResolution;

vec2 zeroToTwo = zeroToOne \* 2.0;

vec2 clipSpace = zeroToTwo - 1.0;

gl\_Position = vec4(clipSpace \* vec2(1, -1), 0, 1);

}

`;

const fsSource = `

precision mediump float;

uniform vec4 uColor;

void main() {

gl\_FragColor = uColor;

}

### Buffer Initialization

A buffer is created to store vertex data for rendering rectangles. This buffer is reused for drawing both the snake segments and the food.

function initBuffer(gl) {

const positionBuffer = gl.createBuffer();

gl.bindBuffer(gl.ARRAY\_BUFFER, positionBuffer);

return positionBuffer;

}

const positionBuffer = initBuffer(gl);

### Game State Management

The game maintains several state variables:

* **Grid Configuration**: Defines the size and dimensions of the game grid.
* **Snake**: An array representing the snake's segments.
* **Direction**: Current movement direction of the snake.
* **Frog**: Position of the food item.
* **Score**: Tracks the player's score.
* **Color**: Dynamic color of the snake.
* **Game Over Flag**: Indicates whether the game has ended.

const gridSize = 20;

const gridWidth = Math.floor(canvas.width / gridSize);

const gridHeight = Math.floor(canvas.height / gridSize);

let snake = [];

let direction = { x: 1, y: 0 };

let nextDirection = { x: 1, y: 0 };

let frog = { x: 0, y: 0 };

let score = 0;

let snakeColor = [0, 1, 0, 1];

let gameOver = false;

let moveInterval = null;

const moveSpeed = 200; // Milliseconds per move

### Rendering Functions

The drawRect function utilizes WebGL to render rectangles representing the snake segments and food.

function drawRect(x, y, size, color) {

const vertices = [

x, y,

x + size, y,

x, y + size,

x, y + size,

x + size, y,

x + size, y + size,

];

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(vertices), gl.STATIC\_DRAW);

gl.useProgram(programInfo.program);

gl.enableVertexAttribArray(programInfo.attribLocations.vertexPosition);

gl.bindBuffer(gl.ARRAY\_BUFFER, positionBuffer);

gl.vertexAttribPointer(

programInfo.attribLocations.vertexPosition,

2,

gl.FLOAT,

false,

0,

0

);

gl.uniform2f(programInfo.uniformLocations.resolution, canvas.width, canvas.height);

gl.uniform4fv(programInfo.uniformLocations.color, color);

gl.drawArrays(gl.TRIANGLES, 0, 6);

}

### Collision Detection and Game Over Handling

Collision detection ensures that the snake does not collide with walls or itself. Upon detecting a collision, the game over state is triggered, displaying relevant messages and offering a restart option.

function update() {

direction = nextDirection;

const head = { ...snake[0] };

head.x += direction.x;

head.y += direction.y;

// Collision with walls

if (head.x < 0 || head.x >= gridWidth || head.y < 0 || head.y >= gridHeight) {

gameOver = true;

return;

}

// Collision with itself

if (snake.some(segment => segment.x === head.x && segment.y === head.y)) {

gameOver = true;

return;

}

// Check if snake eats the frog

if (head.x === frog.x && head.y === frog.y) {

score += 1;

document.getElementById('score').innerText = 'Score: ' + score;

snakeColor = [Math.random(), Math.random(), Math.random(), 1];

snake.unshift(head);

placeFrog();

} else {

snake.unshift(head);

snake.pop();

}

}

### Enhancements and Features

Several enhancements were incorporated to elevate the gaming experience:

* **Dynamic Color Changes**: The snake changes color each time it consumes food, providing visual feedback.
* **Real-Time Score Tracking**: The score updates in real-time, encouraging continuous play.
* **Game Over Interface**: Upon game termination, a "Game Over" message and a restart button appear, allowing players to quickly begin a new game.
* **Keyboard Controls**: Implemented responsive arrow key controls for intuitive navigation.

## Results

The final implementation successfully replicates the core mechanics of the classic Snake game. Players can control the snake using arrow keys, consume food to grow longer, and observe real-time score updates. The snake changes color upon each successful consumption of food, enhancing visual appeal. Collision detection accurately identifies game-over scenarios, prompting users with appropriate messages and providing an easy restart mechanism.

**Key Outcomes:**

* **Smooth Movement**: The snake moves consistently at a fixed speed, adhering to grid-based movement.
* **Responsive Controls**: Arrow key inputs result in immediate and accurate directional changes.
* **Dynamic Visuals**: Random color changes upon eating food add a layer of visual engagement.
* **Robust Collision Detection**: The game reliably detects and handles collisions with walls and self, maintaining game integrity.

## Discussion

### Challenges Faced

1. **WebGL for 2D Rendering**: While WebGL is primarily designed for 3D graphics, adapting it for 2D rendering required careful management of shader programs and coordinate transformations. Ensuring efficient rendering of simple rectangles without overcomplicating the shader logic was a balancing act.
2. **Input Handling and Direction Management**: Preventing the snake from reversing direction directly was essential to avoid immediate self-collisions. Managing the current and next direction inputs to ensure smooth and logical movement demanded meticulous state management.
3. **Collision Detection Efficiency**: As the snake grows longer, efficient detection of self-collisions becomes critical. Implementing a method that scales well with the snake's length was necessary to maintain performance.
4. **Dynamic Color Changes**: Managing dynamic color transitions required ensuring that color data was correctly passed to the shader programs without introducing rendering artifacts.

## Results

The final implementation successfully replicates the core mechanics of the classic Snake game. Players can control the snake using arrow keys, consume food to grow longer, and observe real-time score updates. The snake changes color upon each successful consumption of food, enhancing visual appeal. Collision detection accurately identifies game-over scenarios, prompting users with appropriate messages and providing an easy restart mechanism.

**Key Outcomes:**

* **Smooth Movement**: The snake moves consistently at a fixed speed, adhering to grid-based movement.
* **Responsive Controls**: Arrow key inputs result in immediate and accurate directional changes.
* **Dynamic Visuals**: Random color changes upon eating food add a layer of visual engagement.
* **Robust Collision Detection**: The game reliably detects and handles collisions with walls and self, maintaining game integrity.