**סיכום פרויקט - אלגוריתמים מתקדמים**

עידן לוין 318516135 | לירז ליזי כהן 318865854 | אודיה הרפז 209517614

**רשימת לינקים**

* סרט הדגמה המצגת - <https://youtu.be/VjM4AN29R3M>
* סרט הדגמת המשחק - <https://youtu.be/SQ1SLwTpObM>
* קישור למשחק אונליין - <https://tic-tac-toe-3d-idan-lizi.vercel.app/>

**תיאור הבעיה**בחרנו לממש את המשחק איקס-עיגול בתלת-מימד כך שאנחנו נשחק נגד המחשב. המשחק יכלול כמה שדרוגים בנוסף למשחק המוכר.   
1. המשחק ישוחק על גבי קוביה. כך שכל פאה בקובייה תייצג לוח איקס עיגול רגיל. והמטרה היא

להשיג 3 רצפים על גבי פאות הקוביה. (רצף חוקי הוא שורה/טור/אלכסון מאותה צבע).

2. לכל שחקן תהיה פצצה מיוחדת אשר תוכל לבטל 3 תאים שעל הקוביה. כל משתמש יוכל

להשתמש בפצצה פעם אחת במשחק. שימוש בפצצה יחשב כתור.

**תיאור הפתרון**

ויזואליזציה: משום שרצינו ליצור משחק מושקע ולהשתמש באלמטים תלת מימדיים, השתמשנו בספריה 'THREE'. ספריה זו מאפשרת ליצור אלמנטים תלת מימדיים עם עומק. יצרנו את הקוביה וכל פאה בקוביה מייצגת לוח איקס עיגול. הקוד נמצא בקומפוננטה RaycasterHandler.js.

שחקן המחשב: על מנת שהמחשב יבצע מהלכים נבונים היינו צריכים להשתמש באלגוריתם Minimax שלמדנו בכיתה. בכל שלב של המשחק, המחשב חישב את המהלך הכי טוב עבורו על ידי חשיבה של כמה מהלכים קדימה. בסוף הריצה של האלגוריתם מוחזר המהלך הטוב ביותר שהמחשב העריך כמהלך הכי טוב עבורו.   
מכיוון שמדובר בקוביה ושעל כל פאה יש 9 תאים, נתונים 54 תאים לבחירה סך הכל (לא כולל מיפוי הקוביה. הסבר על מיפוי הקוביה יסופק בהמשך המסמך).

רצינו שהמחשב יחשוב כמה מהלכים קדימה ולקחנו בחשבון שזה יכול לקחת המון זמן. לכן היינו צריכים להוסיף לMinimax את הקונספט של Alpha Beta Pruning כך שהמחשב יוכל לבצע מהלך בזמן קצר הרבה יותר.

הערכת הלוח: כדי שהמחשב יוכל לבצע את המהלך הטוב ביותר הוא צריך לתת ציון לכל מהלך שהוא חושב לבצע. לכן יצרנו את הפונקציה EvaluateBoard שמחשבת לפי קריטריונים מסוימים כמה כדאי למחשב לבצע את המהלך הזה. הקריטריונים שעליהם הוא מסתמך הם: העדפה של תאים בפינות של הפאות (משום שתאים אלה שייכים למספר גדול יותר של פאות שונות ומאפשרות התפרסות "רחבה יותר" ומקנות יותר אופציות), חסימות של רצפים של היריב כדי שלא יוכל לנצח, יצירת רצפים משל עצמו.

מיפוי הקוביה: משום שאנחנו מדברים על קוביה, ברגע שאנחנו לוחצים על תא מסוים הנמצא בפינות הפאה, נצבע יותר מתא אחד, מאחר ויש לו תאים שצמודים אליו (ושייכים כאמור לפאות שונות). כדי להצליח למפות את כל הקוביה היינו צריכים ליצור מין מפה (AdjacentMap) ופונקציה שתקבל תא מסוים בפאה מסוימת והיא תחזיר לנו את השכנים שלו כדי שהם יצבעו גם (getAdjacentCells).

פצצה מיוחדת: בשביל לדעת מתי המחשב צריך להשתמש בפצצה יצרנו פונקציה shouldUseBomb. הפונקציה בודקת האם השחקן היריב נמצא במצב שהוא תכף מנצח ואם כן אז המחשב משתמש בפצצה שלו כדי להוריד תאים של היריב.

//app.js

import React, { useState, useEffect } from "react";

import { Canvas } from "@react-three/fiber"; *// Import for 3D rendering*

import { OrbitControls } from "@react-three/drei"; *// Controls to orbit around the 3D scene*

import TicTacToeBoard from "./TicTacToeBoard"; *// Custom component for the Tic Tac Toe board*

import RaycasterHandler from "./RaycasterHandler"; *// Handles 3D raycasting for mouse interactions*

import bombIcon from "./bomb.png"; *// Icon for the bomb power-up*

import GameModeModal from "./GameModeModal"; *// Modal component to choose the game mode*

const MAX\_DEPTH = 12; *// Maximum depth for the AI search algorithm*

const App = () => {

*// Initialize the game state for 6 faces, each with a 3x3 grid*

const initialState = Array(6).fill().map(() => Array(9).fill(null));

const [gameState, setGameState] = useState(initialState);

const [currentPlayer, setCurrentPlayer] = useState(null); *// Tracks the current player*

const [winner, setWinner] = useState(null); *// Tracks the winner of the game*

const [winningCells, setWinningCells] = useState([]); *// Stores the cells that form the winning line*

const [bombUsed, setBombUsed] = useState({ X: false, O: false }); *// Tracks whether each player has used their bomb*

const [bombMode, setBombMode] = useState(false); *// Flag to indicate if bomb mode is active*

const [bombCells, setBombCells] = useState([]); *// Cells affected by the bomb*

const [highlightedCells, setHighlightedCells] = useState([]); *// Cells to highlight during bomb mode*

const [gameMode, setGameMode] = useState(null); *// Game mode: single or multi-player*

const [modalOpen, setModalOpen] = useState(true); *// Controls visibility of the game mode modal*

const [aiMovePending, setAiMovePending] = useState(false); *// Indicates if an AI move is pending*

const [bombPending, setBombPending] = useState(false); *// Indicates if bomb action is pending*

*// Handles closing the game mode modal and setting up the game based on selected mode*

const handleModalClose = (*mode*) => {

console.log("Game mode selected:", *mode*);

setGameMode(*mode*);

setModalOpen(false);

if (*mode* === "single") {

setCurrentPlayer("O"); *// AI starts in single player mode*

setAiMovePending(true); *// Trigger AI move if it's the computer's turn*

} else {

setCurrentPlayer("X"); *// Player X starts in multiplayer mode*

}

};

*// Effect hook for handling AI moves*

useEffect(() => {

console.log("useEffect triggered", { gameMode, currentPlayer, winner });

if (aiMovePending && currentPlayer === "O") {

console.log("AI is making a move");

const bombDecision = shouldUseBomb(gameState);

if (bombDecision.useBomb) {

alert("AI wants to use the bomb!");

setBombPending(true);*// Set bomb as pending if AI decides to use it*

} else {

const [bestFace, bestCell] = findBestMove(gameState);

console.log("Best move found by AI:", { bestFace, bestCell });

setTimeout(() => handleCellClick(bestFace, bestCell), 500); *// Small delay to simulate thinking*

setAiMovePending(false); *// Reset the flag after AI makes its move*

}

}

}, [aiMovePending, currentPlayer, gameState]);

*// Effect hook for handling bomb actions*

useEffect(() => {

if (bombPending) {

const bombDecision = shouldUseBomb(gameState);

handleBombUsage(bombDecision.bombCells);

setBombPending(false);

setAiMovePending(false);

setCurrentPlayer("X"); *// Switch back to player's turn after AI uses bomb*

}

}, [bombPending]);

*// Handle clicks on the game cells*

const handleCellClick = (*face*, *cell*) => {

if (bombMode) {

handleBombCellSelection(*face*, *cell*);

return; *// If bomb mode is active, handle bomb selection instead of a normal mov*

}

console.log("handleCellClick triggered", { face, cell, currentPlayer });

if (winner || gameState[*face*][*cell*] !== null) return;

*// Update the game state with the new move*

let newGameState = gameState.map((*board*, *idx*) => {

if (*idx* === *face*) {

const newBoard = [...*board*];

newBoard[*cell*] = currentPlayer;

return newBoard;

}

return *board*;

});

if (isCornerCell(*face*, *cell*)) {

const adjacentCells = getAdjacentCells(*face*, *cell*);

adjacentCells.forEach(([*adjFace*, *adjCell*]) => {

newGameState = newGameState.map((*board*, *idx*) => {

if (*idx* === *adjFace*) {

const newBoard = [...*board*];

newBoard[*adjCell*] = currentPlayer;

return newBoard;

}

return *board*;

});

});

}

setGameState(newGameState);

*// Check for any winning combination after the move*

const { winningPlayer, winningCells } = checkCubeWin(newGameState);

if (winningPlayer) {

setWinner(winningPlayer);

setWinningCells(winningCells);

console.log("Winner found:", winningPlayer);

} else {

if (currentPlayer === "X" && gameMode === "single") {

setCurrentPlayer("O");

setTimeout(() => setAiMovePending(true), 500); *// Trigger AI move after a delay*

} else {

setCurrentPlayer(currentPlayer === "X" ? "O" : "X");

}

}

};

*// Determine if the AI should use a bomb based on game state analysis*

const shouldUseBomb = (*gameState*) => {

if (bombUsed["O"]) return { useBomb: false, bombCells: [] };

let playerSequences = 0;

let almostCompleteSequence = false;

let completedSequences = [];

let bombCells = [];

for (let face = 0; face < 6; face++) {

const lines = [

[0, 1, 2],

[3, 4, 5],

[6, 7, 8], *// Rows*

[0, 3, 6],

[1, 4, 7],

[2, 5, 8], *// Columns*

[0, 4, 8],

[2, 4, 6], *// Diagonals*

];

for (let line of lines) {

const values = line.map((*idx*) => *gameState*[face][*idx*]);

const XCount = values.filter((*v*) => *v* === "X").length;

const nullCount = values.filter((*v*) => *v* === null).length;

if (XCount === 3) {

playerSequences++;

completedSequences.push(line.map((*idx*) => ({ face, cell: *idx* })));

}

if (XCount === 2 && nullCount === 1) {

almostCompleteSequence = true;

}

}

}

if (playerSequences >= 2 && almostCompleteSequence) {

bombCells = completedSequences[0]; *// Select one of the completed sequences*

return { useBomb: true, bombCells };

}

return { useBomb: false, bombCells: [] };

};

*// Handle the actual bomb usage, updating the game state accordingly*

const handleBombUsage = (*bombCells*) => {

let cellsToBomb = [...*bombCells*];

*bombCells*.forEach(({ *face*, *cell* }) => {

const adjacentCells = getAdjacentCells(*face*, *cell*);

cellsToBomb = cellsToBomb.concat(

adjacentCells.map(([*adjFace*, *adjCell*]) => ({

face: *adjFace*,

cell: adjCell,

}))

);

});

let newGameState = gameState.map((*board*, *faceIdx*) => {

if (cellsToBomb.some((*c*) => *c*.face === *faceIdx*)) {

const newBoard = [...*board*];

cellsToBomb

.filter((*c*) => *c*.face === *faceIdx*)

.forEach(({ *cell* }) => {

newBoard[*cell*] = null;

});

return newBoard;

}

return *board*;

});

setGameState(newGameState);

setBombUsed({ ...bombUsed, [currentPlayer]: true });

setBombMode(false);

setBombCells([]);

setHighlightedCells([]);

const nextPlayer = currentPlayer === "X" ? "O" : "X";

setCurrentPlayer(nextPlayer);

if (nextPlayer === "O" && gameMode === "single") {

setTimeout(() => setAiMovePending(true), 500); *// Trigger AI move after a delay*

}

};

*// AI logic to determine the best move based on current game state*

const findBestMove = (*gameState*) => {

let bestMove = [-1, -1];

for (let depth = 1; depth <= MAX\_DEPTH; depth++) {

bestMove = findBestMoveAtDepth(*gameState*, depth);

}

return bestMove;

};

*// Additional functions like resetGame, getCurrentPlayerText, etc.*

const findBestMoveAtDepth = (*gameState*, *maxDepth*) => {

let bestVal = -Infinity;

let bestMove = [-1, -1];

const moves = getAllPossibleMoves(*gameState*);

for (let [face, cell] of moves) {

*gameState*[face][cell] = "O";

let moveVal = minimax(*gameState*, 0, false, -Infinity, Infinity, *maxDepth*);

*gameState*[face][cell] = null;

if (moveVal > bestVal) {

bestMove = [face, cell];

bestVal = moveVal;

}

}

return bestMove;

};

const getAllPossibleMoves = (*gameState*) => {

const moves = [];

for (let face = 0; face < 6; face++) {

for (let cell = 0; cell < 9; cell++) {

if (*gameState*[face][cell] === null) {

moves.push([face, cell]);

}

}

}

*// Move ordering: center, corners, edges*

const scoreCell = (*cell*) => {

if (*cell* === 4) return 3; *// Center*

if ([0, 2, 6, 8].includes(*cell*)) return 2; *// Corners*

return 1; *// Edges*

};

moves.sort((*a*, *b*) => scoreCell(*b*[1]) - scoreCell(*a*[1]));

return moves;

};

const transpositionTable = new Map();

const minimax = (*gameState*, *depth*, *isMaximizing*, *alpha*, *beta*, *maxDepth*) => {

const stateKey = JSON.stringify(*gameState*);

if (transpositionTable.has(stateKey))

return transpositionTable.get(stateKey);

const { winningPlayer } = checkCubeWin(*gameState*);

if (winningPlayer === "O") return 100 - *depth*;

if (winningPlayer === "X") return *depth* - 100;

if (!isMovesLeft(*gameState*)) return 0;

if (*depth* >= *maxDepth*) return evaluateBoard(*gameState*, *depth*);

const moves = getAllPossibleMoves(*gameState*);

let best;

if (*isMaximizing*) {

best = -Infinity;

for (let [face, cell] of moves) {

*gameState*[face][cell] = "O";

best = Math.max(

best,

minimax(*gameState*, *depth* + 1, false, *alpha*, *beta*, *maxDepth*)

);

*gameState*[face][cell] = null;

*alpha* = Math.max(*alpha*, best);

if (*beta* <= *alpha*) break; *// Prune remaining branches*

}

} else {

best = Infinity;

for (let [face, cell] of moves) {

gameState[face][cell] = "X";

best = Math.min(

best,

minimax(gameState, depth + 1, true, alpha, beta, maxDepth)

);

gameState[face][cell] = null;

beta = Math.min(beta, best);

if (beta <= alpha) break; *// Prune remaining branches*

}

}

transpositionTable.set(stateKey, best);

return best;

};

const evaluateBoard = (*gameState*, *depth*) => {

let score = 0;

const { winningPlayer } = checkCubeWin(gameState);

if (winningPlayer === "O") return 100 - depth;

if (winningPlayer === "X") return depth - 100;

*// Additional heuristic: count potential winning lines for both players*

for (let face = 0; face < 6; face++) {

const lines = checkFaceWin(gameState[face]);

lines.forEach((*line*) => {

const values = line.map((*idx*) => gameState[face][idx]);

const OCount = values.filter((*v*) => v === "O").length;

const XCount = values.filter((*v*) => v === "X").length;

*// Reward AI for lines that are closer to completion*

if (OCount > 0 && XCount === 0) score += Math.pow(10, OCount);

if (XCount > 0 && OCount === 0) score -= Math.pow(10, XCount);

*// Strongly penalize potential opponent wins*

if (XCount === 2 && OCount === 0) score -= 1000; *// Increase penalty for potential opponent wins*

if (XCount === 1 && OCount === 0) score -= 10; *// Small penalty for potential opponent wins*

});

}

*// Check for imminent wins and prioritize blocking them*

for (let face = 0; face < 6; face++) {

for (let cell = 0; cell < 9; cell++) {

if (gameState[face][cell] === null) {

gameState[face][cell] = "X";

if (checkCubeWin(gameState).winningPlayer === "X") {

score -= 10000; *// Large negative score to prioritize blocking*

}

gameState[face][cell] = null;

}

}

}

*// Prioritize corners, middle edges, and center cells*

const cornerCells = [0, 2, 6, 8];

const middleEdgeCells = [1, 3, 5, 7];

const centerCell = [4];

for (let face = 0; face < 6; face++) {

cornerCells.forEach((*cell*) => {

if (gameState[face][cell] === "O") score += 5;

if (gameState[face][cell] === "X") score -= 5;

});

middleEdgeCells.forEach((*cell*) => {

if (gameState[face][cell] === "O") score += 3;

if (gameState[face][cell] === "X") score -= 3;

});

centerCell.forEach((*cell*) => {

if (gameState[face][cell] === "O") score += 1;

if (gameState[face][cell] === "X") score -= 1;

});

}

return score;

};

const isMovesLeft = (*gameState*) => {

for (let face = 0; face < 6; face++) {

for (let cell = 0; cell < 9; cell++) {

if (gameState[face][cell] === null) return true;

}

}

return false;

};

const handleBombClick = () => {

if (bombUsed[currentPlayer] || winner) return;

console.log(`${currentPlayer} is entering bomb mode`);

setBombMode(true);

};

const handleBombCellSelection = (*face*, *cell*) => {

if (bombCells.length >= 3) {

setBombMode(false);

setBombCells([]);

setHighlightedCells([]);

return;

}

const newBombCells = [...bombCells, { face, cell }];

const newHighlightedCells = [...highlightedCells, { face, cell }];

if (isCornerCell(face, cell)) {

const adjacentCells = getAdjacentCells(face, cell);

adjacentCells.forEach(([*adjFace*, *adjCell*]) => {

newHighlightedCells.push({ face: adjFace, cell: adjCell });

});

}

setBombCells(newBombCells);

setHighlightedCells(newHighlightedCells);

if (newBombCells.length === 3) {

if (isValidTriple(newBombCells)) {

explodeBomb(newHighlightedCells);

handleBombUsage(newBombCells); *// Handle the bomb usage immediately after selection*

} else {

alert("Invalid triple selection. Please select a valid triple.");

setBombCells([]);

setHighlightedCells([]);

}

}

};

const isValidTriple = (*cells*) => {

if (*cells*.length !== 3) return false;

const faces = *cells*.map((*c*) => *c*.face);

const positions = *cells*.map((*c*) => *c*.cell);

const uniqueFaces = new Set(faces);

if (uniqueFaces.size !== 1) return false;

const face = faces[0];

const lines = [

[0, 1, 2],

[3, 4, 5],

[6, 7, 8], *// Rows*

[0, 3, 6],

[1, 4, 7],

[2, 5, 8], *// Columns*

[0, 4, 8],

[2, 4, 6], *// Diagonals*

];

return lines.some((*line*) => *line*.every((*pos*) => positions.includes(*pos*)));

};

const explodeBomb = (*cells*) => {

let newGameState = gameState.map((*board*, *faceIdx*) => {

if (*cells*.some((*c*) => *c*.face === *faceIdx*)) {

const newBoard = [...*board*];

*cells*

.filter((*c*) => *c*.face === *faceIdx*)

.forEach(({ *cell* }) => {

newBoard[*cell*] = null;

});

return newBoard;

}

return *board*;

});

setGameState(newGameState);

setBombUsed({ ...bombUsed, [currentPlayer]: true });

setBombMode(false);

setBombCells([]);

setHighlightedCells([]);

if (currentPlayer === "X" && gameMode === "single") {

setTimeout(() => setAiMovePending(true), 500); *// Trigger AI move after a delay*

} else {

setCurrentPlayer(currentPlayer === "X" ? "O" : "X");

}

};

const checkFaceWin = (*board*) => {

const lines = [

*// Rows*

[0, 1, 2],

[3, 4, 5],

[6, 7, 8],

*// Columns*

[0, 3, 6],

[1, 4, 7],

[2, 5, 8],

*// Diagonals*

[0, 4, 8],

[2, 4, 6],

];

const triples = [];

for (let line of lines) {

const [a, b, c] = line;

if (*board*[a] && *board*[a] === *board*[b] && *board*[a] === *board*[c]) {

triples.push(line);

}

}

return triples;

};

const checkCubeWin = (*gameState*) => {

let triplesX = 0;

let triplesO = 0;

let winningCellsX = [];

let winningCellsO = [];

for (let face = 0; face < 6; face++) {

const triples = checkFaceWin(*gameState*[face]);

triples.forEach((*line*) => {

if (*gameState*[face][*line*[0]] === "X") {

triplesX++;

winningCellsX.push(...*line*.map((*cell*) => ({ face, cell })));

} else if (*gameState*[face][*line*[0]] === "O") {

triplesO++;

winningCellsO.push(...*line*.map((*cell*) => ({ face, cell })));

}

});

}

if (triplesX >= 3) {

return { winningPlayer: "X", winningCells: winningCellsX };

}

if (triplesO >= 3) {

return { winningPlayer: "O", winningCells: winningCellsO };

}

return { winningPlayer: null, winningCells: [] };

};

const isCornerCell = (*face*, *cell*) => {

const corners = {

0: [0, 1, 2, 3, 5, 6, 7, 8],

1: [0, 1, 2, 3, 5, 6, 7, 8],

2: [0, 1, 2, 3, 5, 6, 7, 8],

3: [0, 1, 2, 3, 5, 6, 7, 8],

4: [0, 1, 2, 3, 5, 6, 7, 8],

5: [0, 1, 2, 3, 5, 6, 7, 8],

};

return corners[*face*].includes(*cell*);

};

const getAdjacentCells = (*face*, *cell*) => {

const adjacentMap = {

"0-0": [

[4, 6],

[3, 2],

],

"0-1": [[4, 7]],

"0-2": [

[4, 8],

[2, 0],

],

"0-3": [[3, 5]],

"0-5": [[2, 3]],

"0-6": [

[5, 0],

[3, 8],

],

"0-7": [[5, 1]],

"0-8": [

[5, 2],

[2, 6],

],

"1-0": [

[2, 2],

[4, 2],

],

"1-1": [[4, 1]],

"1-2": [

[4, 0],

[3, 0],

],

"1-3": [[2, 5]],

"1-5": [[3, 3]],

"1-6": [

[2, 8],

[5, 8],

],

"1-7": [[5, 7]],

"1-8": [

[5, 6],

[3, 6],

],

"2-0": [

[4, 8],

[0, 2],

],

"2-1": [[4, 5]],

"2-2": [

[1, 0],

[4, 2],

],

"2-3": [[0, 5]],

"2-5": [[1, 3]],

"2-6": [

[5, 2],

[0, 8],

],

"2-7": [[5, 5]],

"2-8": [

[1, 6],

[5, 8],

],

"3-0": [

[4, 0],

[1, 2],

],

"3-1": [[4, 3]],

"3-2": [

[0, 0],

[4, 6],

],

"3-3": [[1, 5]],

"3-5": [[0, 3]],

"3-6": [

[5, 6],

[1, 8],

],

"3-7": [[5, 3]],

"3-8": [

[0, 6],

[5, 0],

],

"4-0": [

[3, 0],

[1, 2],

],

"4-1": [[1, 1]],

"4-2": [

[2, 2],

[1, 0],

],

"4-3": [[3, 1]],

"4-5": [[2, 1]],

"4-6": [

[0, 0],

[3, 2],

],

"4-7": [[0, 1]],

"4-8": [

[0, 2],

[2, 0],

],

"5-0": [

[0, 6],

[3, 8],

],

"5-1": [[0, 7]],

"5-2": [

[2, 6],

[0, 8],

],

"5-3": [[3, 7]],

"5-5": [[2, 7]],

"5-6": [

[1, 8],

[3, 6],

],

"5-7": [[1, 7]],

"5-8": [

[2, 8],

[1, 6],

],

};

return adjacentMap[`${*face*}-${*cell*}`] || [];

};

const resetGame = () => {

setGameState(initialState);

setCurrentPlayer(null); *// Reset the game with no player*

setWinner(null);

setWinningCells([]);

setBombUsed({ X: false, O: false });

setBombMode(false);

setBombCells([]);

setHighlightedCells([]);

setModalOpen(true); *// Open the modal when resetting the game*

setAiMovePending(false); *// Reset AI move pending flag*

};

const getCurrentPlayerText = () => {

if (currentPlayer === "X") return "Your turn";

if (currentPlayer === "O") return "Computer's turn";

return null;

};

return (

<>

<GameModeModal

*open*={modalOpen}

*handleClose*={handleModalClose}

*setGameMode*={setGameMode}

/>

<div

*style*={{

position: "absolute",

top: "1%",

left: "50%",

transform: "translateX(-50%)",

}}

>

<button

*onClick*={resetGame}

*style*={{

backgroundColor: "#ff64ab",

fontSize: "1rem",

padding: "10px 20px",

border: "none",

borderRadius: "10px",

cursor: "pointer",

color: "white",

fontWeight: "bold",

}}

>

Reset Game

</button>

</div>

{winner && (

<div

*style*={{

position: "absolute",

top: "12%",

left: "50%",

transform: "translateX(-50%)",

fontSize: "5.3rem",

color: "#6bc9ff",

fontWeight: "bold",

}}

>

{winner === "X" ? "You" : "The computer"} win!

</div>

)}

{!winner && currentPlayer && (

<div

*style*={{

backgroundColor: "#ffba46",

position: "absolute",

top: "12%",

left: "50%",

transform: "translateX(-50%)",

fontSize: "2.8rem",

borderRadius: "20px",

padding: "8px 17px",

color: "white",

fontWeight: "bold",

fontFamily: "calibri",

}}

>

{getCurrentPlayerText()}

</div>

)}

<div

*style*={{

position: "absolute",

top: "10%",

left: "10%",

cursor: "pointer",

zIndex: 1,

textAlign: "center",

}}

>

<div

*style*={{ color: "#9a8eff", fontSize: "1.5em", fontWeight: "bold" }}

>

Your Bomb

</div>

{bombUsed["X"] ? null : (

<img

*src*={bombIcon}

*alt*="Bomb Icon X"

*style*={{ width: 50, height: 50 }}

*onClick*={() => {

if (currentPlayer === "X") handleBombClick();

}}

/>

)}

</div>

<div

*style*={{

position: "absolute",

top: "10%",

right: "10%",

cursor: "pointer",

zIndex: 1,

textAlign: "center",

}}

>

<div

*style*={{ color: "#00ffc1", fontSize: "1.5em", fontWeight: "bold" }}

>

Computer's Bomb

</div>

{bombUsed["O"] ? null : (

<img

*src*={bombIcon}

*alt*="Bomb Icon O"

*style*={{ width: 50, height: 50 }}

*onClick*={() => {

if (currentPlayer === "O") handleBombClick();

}}

/>

)}

</div>

<Canvas *style*={{ height: "100vh", marginTop: "5em" }}>

<ambientLight *intensity*={0.5} />

<pointLight *position*={[10, 10, 10]} />

<group>

{gameState.map((*board*, *idx*) => (

<TicTacToeBoard

*key*={*idx*}

*face*={*idx*}

*board*={*board*}

*position*={getBoardPosition(*idx*)}

*rotation*={getBoardRotation(*idx*)}

*onCellClick*={handleCellClick}

*winningCells*={winningCells

.filter((*cell*) => *cell*.face === *idx*)

.map((*cell*) => *cell*.cell)}

*bombMode*={bombMode}

*bombCells*={bombCells}

*highlightedCells*={highlightedCells}

/>

))}

</group>

<OrbitControls />

<RaycasterHandler *onCellClick*={handleCellClick} />

</Canvas>

</>

);

};

*// Define 3D positions for each face of the cube*

const getBoardPosition = (*idx*) => {

const positions = [

[0, 0, 1.5],

[0, 0, -1.5],

[1.5, 0, 0],

[-1.5, 0, 0],

[0, 1.5, 0],

[0, -1.5, 0],

];

return positions[*idx*];

};

*// Define rotation for each face to orient it correctly in the 3D space*

const getBoardRotation = (*idx*) => {

const rotations = [

[0, 0, 0],

[0, Math.PI, 0],

[0, Math.PI / 2, 0],

[0, -Math.PI / 2, 0],

[-Math.PI / 2, 0, 0],

[Math.PI / 2, 0, 0],

];

return rotations[*idx*];

};

export default App;

//TicTacToeBoard.js

import React, { useState } from 'react';

import { extend } from '@react-three/fiber';

import { Line } from '@react-three/drei';

import \* as THREE from 'three';

extend({ Line\_: Line });

const getAdjacentCells = (*face*, *cell*) => {

const adjacentMap = {

'0-0': [[4, 6], [3, 2]],

'0-1': [[4, 7]],

'0-2': [[4, 8], [2, 0]],

'0-3': [[3, 5]],

'0-5': [[2, 3]],

'0-6': [[5, 0], [3, 8]],

'0-7': [[5, 1]],

'0-8': [[5, 2], [2, 6]],

'1-0': [[2, 2], [4, 2]],

'1-1': [[4, 1]],

'1-2': [[4, 0], [3, 0]],

'1-3': [[2, 5]],

'1-5': [[3, 3]],

'1-6': [[2, 8], [5, 8]],

'1-7': [[5, 7]],

'1-8': [[5, 6], [3, 6]],

'2-0': [[4, 8], [0, 2]],

'2-1': [[4, 5]],

'2-2': [[1, 0], [4, 2]],

'2-3': [[0, 5]],

'2-5': [[1, 3]],

'2-6': [[5, 2], [0, 8]],

'2-7': [[5, 5]],

'2-8': [[1, 6], [5, 8]],

'3-0': [[4, 0], [1, 2]],

'3-1': [[4, 3]],

'3-2': [[0, 0], [4, 6]],

'3-3': [[1, 5]],

'3-5': [[0, 3]],

'3-6': [[5, 6], [1, 8]],

'3-7': [[5, 3]],

'3-8': [[0, 6], [5, 0]],

'4-0': [[3, 0], [1, 2]],

'4-1': [[1, 1]],

'4-2': [[2, 2], [1, 0]],

'4-3': [[3, 1]],

'4-5': [[2, 1]],

'4-6': [[0, 0], [3, 2]],

'4-7': [[0, 1]],

'4-8': [[0, 2], [2, 0]],

'5-0': [[0, 6], [3, 8]],

'5-1': [[0, 7]],

'5-2': [[2, 6], [0, 8]],

'5-3': [[3, 7]],

'5-5': [[2, 7]],

'5-6': [[1, 8], [3, 6]],

'5-7': [[1, 7]],

'5-8': [[2, 8], [1, 6]],

};

return adjacentMap[`${*face*}-${*cell*}`] || [];

};

const TicTacToeBoard = ({ *face*, *board*, *position*, *rotation*, *onCellClick*, *winningCells*, *bombMode*, *bombCells*, *highlightedCells* }) => {

const [hoveredCell, setHoveredCell] = useState(null);

const [hoveredFace, setHoveredFace] = useState(null);

const renderCell = (*cell*, *idx*) => {

const isWinningCell = *winningCells*.includes(*idx*);

const isSelectedForBomb = *bombCells*.some(*bc* => *bc*.face === *face* && *bc*.cell === *idx*);

const isHighlightedCell = *highlightedCells*.some(*hc* => *hc*.face === *face* && *hc*.cell === *idx*);

const isHovered = hoveredFace === *face* && hoveredCell === *idx*;

const adjacentCells = hoveredFace !== null && hoveredCell !== null ? getAdjacentCells(hoveredFace, hoveredCell) : [];

const isAdjacentHovered = adjacentCells.some(([*adjFace*, *adjCell*]) => *adjFace* === *face* && *adjCell* === *idx*);

return (

<group *key*={*idx*} *position*={getCellPosition(*idx*)}>

<mesh

*userData*={{ face, cell: *idx* }}

*onPointerOver*={(*e*) => { setHoveredCell(*idx*); setHoveredFace(*face*); *e*.stopPropagation(); }}

*onPointerOut*={(*e*) => { setHoveredCell(null); setHoveredFace(null); *e*.stopPropagation(); }}

*onDoubleClick*={(*e*) => { *// Change this line*

onCellClick(*face*, *idx*);

*e*.stopPropagation();

}}

>

<boxGeometry *args*={[1, 1, 0.1]} />

<meshBasicMaterial *color*={

(isHovered || isAdjacentHovered) && *cell* === null ? '#8c8a89' :

isSelectedForBomb || isHighlightedCell ? '#ff64ab' :

isWinningCell ? '#40ff00' :

*cell* ? (*cell* === 'X' ? '#9a8eff' : '#00ffc1') : 'white'} />

</mesh>

</group>

);

};

const renderCellBorders = () => {

const lines = [];

const lineMaterial = new THREE.LineBasicMaterial({ color: 'black' });

const halfSize = 1.5;

*// Create vertical lines*

for (let i = -1; i <= 1; i++) {

lines.push(

<line *key*={`v-${i}`} *position*={[0, 0, 0.06]}>

<bufferGeometry>

<bufferAttribute

*attachObject*={['attributes', 'position']}

*array*={new Float32Array([i, -halfSize, 0, i, halfSize, 0])}

*itemSize*={3}

*count*={2}

/>

</bufferGeometry>

<primitive *attach*="material" *object*={lineMaterial} />

</line>

);

}

*// Create horizontal lines*

for (let j = -1; j <= 1; j++) {

lines.push(

<line *key*={`h-${j}`} *position*={[0, 0, 0.06]}>

<bufferGeometry>

<bufferAttribute

*attachObject*={['attributes', 'position']}

*array*={new Float32Array([-halfSize, j, 0, halfSize, j, 0])}

*itemSize*={3}

*count*={2}

/>

</bufferGeometry>

<primitive *attach*="material" *object*={lineMaterial} />

</line>

);

}

return lines;

};

return (

<group *position*={*position*} *rotation*={*rotation*}>

{*board*.map((*cell*, *idx*) => renderCell(*cell*, *idx*))}

{renderCellBorders()}

</group>

);

};

const getCellPosition = (*idx*) => {

const x = (*idx* % 3) - 1;

const y = 1 - Math.floor(*idx* / 3);

return [x \* 1, y \* 1, 0];

};

export default TicTacToeBoard;

//RaycasterHandler.js

*//src/RaycasterHandler.js*

import { useThree, useFrame } from '@react-three/fiber';

import \* as THREE from 'three';

import React, { useRef } from 'react';

const RaycasterHandler = ({ *onCellDoubleClick* }) => {

const { gl, camera, scene } = useThree();

const raycaster = useRef(new THREE.Raycaster());

const mouse = useRef(new THREE.Vector2());

const handlePointerMove = (*event*) => {

mouse.current.x = (*event*.clientX / window.innerWidth) \* 2 - 1;

mouse.current.y = -(*event*.clientY / window.innerHeight) \* 2 + 1;

};

const handlePointerDown = (*event*) => {

if (*event*.detail === 2) { *// Detect double-click*

raycaster.current.setFromCamera(mouse.current, camera);

const intersects = raycaster.current.intersectObjects(scene.children, true);

if (intersects.length > 0) {

const intersect = intersects[0];

const { face, cell } = intersect.object.userData;

if (face !== undefined && cell !== undefined) {

onCellDoubleClick(face, cell);

}

}

}

};

useFrame(() => {

raycaster.current.setFromCamera(mouse.current, camera);

});

React.useEffect(() => {

gl.domElement.addEventListener('pointermove', handlePointerMove);

gl.domElement.addEventListener('pointerdown', handlePointerDown);

return () => {

gl.domElement.removeEventListener('pointermove', handlePointerMove);

gl.domElement.removeEventListener('pointerdown', handlePointerDown);

};

}, [gl.domElement]);

return null;

};

export default RaycasterHandler;