<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Trinity Clash</title>

<link rel="manifest" href="manifest.json">

<meta name="mobile-web-app-capable" content="yes">

<meta name="apple-mobile-web-app-capable" content="yes">

<meta name="application-name" content="Trinity Clash">

<meta name="apple-mobile-web-app-title" content="Trinity Clash">

<meta name="theme-color" content="#2d2d2d">

<meta name="msapplication-navbutton-color" content="#2d2d2d">

<meta name="apple-mobile-web-app-status-bar-style" content="black-translucent">

<meta name="msapplication-starturl" content="/">

<link rel="preconnect" href="https://fonts.googleapis.com">

<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>

<link href="https://fonts.googleapis.com/css2?family=Cinzel:wght@400;700&family=MedievalSharp&display=swap" rel="stylesheet">

<style>

body {

margin: 0;

background-color: #2d2d2d; /\* Darker background \*/

background-image:

radial-gradient(circle at center, rgba(255, 255, 255, 0.05) 0%, rgba(255, 255, 255, 0) 60%),

linear-gradient(45deg, rgba(0,0,0,0.2) 25%, transparent 25%, transparent 75%, rgba(0,0,0,0.2) 75%),

linear-gradient(45deg, rgba(0,0,0,0.2) 25%, transparent 25%, transparent 75%, rgba(0,0,0,0.2) 75%);

background-size: 100% 100%, 60px 60px, 60px 60px;

background-position: 0 0, 0 0, 30px 30px;

font-family: 'MedievalSharp', cursive;

overflow: hidden; /\* Prevent scrolling \*/

}

canvas {

display: block;

}

</style>

</head>

<body>

<canvas id="gameCanvas"></canvas>

<script>

// ===================================

// 🎨 1. CONFIGURATION & CONSTANTS

// ===================================

const COLORS = {

background: '#2d2d2d',

text: '#EAE0C8',

cardBorder: '#1a1a1a',

might: '#9E2B25', // Deep Red

finesse: '#00695C', // Teal Green

magic: '#4A5D9A', // Royal Blue

accent: '#FFD700', // Brighter Gold

disabled: '#424242',

button: { start: '#8D6E63', end: '#5D4037' }, // Wood Brown Gradient

buttonHover: { start: '#A1887F', end: '#795548' },

buttonBorder: '#3E2723', // Darker Wood

damage: '#FF7043', // Orange-Red for damage

ability: '#4FC3F7', // Light Blue for abilities

info: '#BDBDBD' // Gray for general info

};

const HEROES = [

// Might Heroes

{ id: 0, name: 'Sir Reginald', type: 'Might', hp: 10, ap: 4, abilityId: 'lastStand', description: 'Last Stand: If knocked out, survives with 1 HP (once per game).' },

{ id: 1, name: 'Grak the Crusher', type: 'Might', hp: 8, ap: 6, abilityId: 'crush', description: 'Crush: His attacks ignore all damage reduction effects.' },

{ id: 6, name: 'Boro the Bulwark', type: 'Might', hp: 10, ap: 4, abilityId: 'tank', description: 'Tank: Boro absorbs all damage for his allies.' },

{ id: 9, name: 'Skeleton Warrior', type: 'Might', hp: 10, ap: 4, abilityId: null, description: 'A sturdy and reliable frontline soldier.' },

{ id: 12, name: 'Goblin Brawler', type: 'Might', hp: 9, ap: 5, abilityId: 'mobRule', description: 'Mob Rule: Gains +1 AP for each other living Goblin ally.' },

// Finesse Heroes

{ id: 2, name: 'Vex', type: 'Finesse', hp: 7, ap: 7, abilityId: 'firstStrike', description: 'First Strike: Attacks first. If she defeats the opponent, she takes no damage.' },

{ id: 3, name: 'Kaelen', type: 'Finesse', hp: 7, ap: 7, abilityId: 'longBow', description: 'Long Bow: Every other round, deals 2 damage to a random enemy.' },

{ id: 7, name: 'Lyra Nightwind', type: 'Finesse', hp: 8, ap: 6, abilityId: 'smokeBomb', description: 'Smoke Bomb: The first attack against Lyra misses.' },

{ id: 10, name: 'Skeleton Archer', type: 'Finesse', hp: 7, ap: 7, abilityId: null, description: 'A balanced archer with no special tricks.' },

{ id: 13, name: 'Goblin Cutpurse', type: 'Finesse', hp: 7, ap: 7, abilityId: 'ambush', description: 'Ambush: Deals 2 damage to the clashing enemy when on the bench.' },

// Magic Heroes

{ id: 4, name: 'Elara', type: 'Magic', hp: 6, ap: 8, abilityId: 'soulSiphon', description: 'Soul Siphon: The first time Elara defeats an enemy, she gains their max HP.' },

{ id: 5, name: 'Zoltan', type: 'Magic', hp: 5, ap: 9, abilityId: 'overload', description: 'Overload: 50% chance to deal 4 damage to another random enemy or himself.' },

{ id: 8, name: 'Master Theron', type: 'Magic', hp: 7, ap: 7, abilityId: 'fireball', description: 'Fireball: On round 5, deals 5 damage to all enemies.' },

{ id: 11, name: 'Skeleton Mage', type: 'Magic', hp: 4, ap: 10, abilityId: null, description: 'A frail but powerful spellcaster.' },

{ id: 14, name: 'Goblin Bomber', type: 'Magic', hp: 4, ap: 10, abilityId: 'unstableConcoction', description: 'Unstable Concoction: With advantage, explodes, dealing 3 damage to all enemies.' }

];

const BOSSES = [

{ id: 100, name: 'Minotaur', type: 'Might', hp: 50, ap: 5, abilities: ['cleave', 'ram', 'intimidation'], description: 'A fearsome beast with devastating attacks.' }

];

// ===================================

// ⚔️ 2. GAME CLASS

// ===================================

class Game {

constructor(canvasId) {

this.canvas = document.getElementById(canvasId);

this.ctx = this.canvas.getContext('2d');

this.scale = 1;

this.baseWidth = 1920;

this.baseHeight = 1080;

this.baseCardWidth = 180;

this.baseCardHeight = 260;

this.cardHitboxes = { player1: [], player2: [], selection: [] };

this.buttonHitboxes = {};

this.mousePos = { x: 0, y: 0 };

this.state = this.getInitialState();

this.init();

}

// -----------------------------------

// Initialization

// -----------------------------------

init() {

this.resizeCanvas();

window.addEventListener('resize', () => this.resizeCanvas());

this.canvas.addEventListener('click', (e) => this.handleCanvasClick(e));

this.canvas.addEventListener('mousemove', (e) => this.handleMouseMove(e));

this.gameLoop();

}

getInitialState() {

return {

gamePhase: 'mainMenu',

difficulty: 'normal',

teamSize: 3,

player1Team: [],

player2Team: [],

player1Selection: null,

player2Selection: null,

winner: null,

isClashing: false,

pendingClash: null,

animation: null,

impactAnimation: null,

fireballAnimation: null,

overloadAnimation: null,

round: 1,

heroViewerIndex: 0,

practiceSelectionState: 'player',

hoveredHero: null,

abilityAnimations: [],

battleLog: [[]],

confettiParticles: [],

bossActiveAbility: null,

};

}

resetGame() {

this.state = this.getInitialState();

this.log([{ text: 'Welcome! Select a hero to begin.', color: COLORS.info }]);

}

startBossBattle() {

this.state.teamSize = 5;

if (this.state.player1Team.length !== this.state.teamSize) {

this.state.gamePhase = 'heroSelection';

return;

}

this.state.player2Team = [JSON.parse(JSON.stringify(BOSSES[0]))];

this.state.gamePhase = 'bossBattle';

this.log([{ text: 'BOSS BATTLE! The Minotaur appears!', color: COLORS.might }]);

this.log([{ text: 'Select a hero to face the beast.', color: COLORS.info }]);

}

startGame(isPractice = false) {

if (this.state.player1Team.length !== this.state.teamSize) return;

if (!isPractice) {

const deepCopyHeroes = JSON.parse(JSON.stringify(HEROES));

if (this.state.difficulty === 'easy') {

this.state.player2Team = deepCopyHeroes.filter(hero =>

hero.name.includes('Skeleton')

).slice(0, this.state.teamSize);

} else {

const playerHeroIds = this.state.player1Team.map(h => h.id);

let remainingHeroes = deepCopyHeroes.filter(hero => !playerHeroIds.includes(hero.id));

if (this.state.difficulty === 'expert') {

const playerTypes = { Might: 0, Finesse: 0, Magic: 0 };

this.state.player1Team.forEach(hero => playerTypes[hero.type]++);

const counterType = { Might: 'Magic', Finesse: 'Might', Magic: 'Finesse' };

const typePriority = Object.keys(playerTypes)

.sort((a, b) => playerTypes[b] - playerTypes[a])

.map(type => counterType[type]);

this.state.player2Team = [];

const pickedIds = new Set();

for (let i = 0; i < this.state.teamSize; i++) {

let pickedHero = null;

for (const type of typePriority) {

const availableCounterHeroes = remainingHeroes.filter(h => h.type === type && !pickedIds.has(h.id));

if (availableCounterHeroes.length > 0) {

pickedHero = availableCounterHeroes[Math.floor(Math.random() \* availableCounterHeroes.length)];

break;

}

}

if (!pickedHero) {

const trulyRemaining = remainingHeroes.filter(h => !pickedIds.has(h.id));

if(trulyRemaining.length > 0) {

pickedHero = trulyRemaining[Math.floor(Math.random() \* trulyRemaining.length)];

}

}

if(pickedHero) {

this.state.player2Team.push(pickedHero);

pickedIds.add(pickedHero.id);

}

}

} else {

for (let i = remainingHeroes.length - 1; i > 0; i--) {

const j = Math.floor(Math.random() \* (i + 1));

[remainingHeroes[i], remainingHeroes[j]] = [remainingHeroes[j], remainingHeroes[i]];

}

this.state.player2Team = remainingHeroes.slice(0, this.state.teamSize);

}

}

}

this.state.gamePhase = 'playing';

this.log([{ text: 'Select one of your heroes to clash.', color: COLORS.info }]);

}

// -----------------------------------

// Game Loop & Drawing

// -----------------------------------

gameLoop() {

this.update();

this.draw();

requestAnimationFrame(() => this.gameLoop());

}

update() {

const speedMultiplier = 0.4;

if (this.state.animation) {

const anim = this.state.animation;

anim.progress += anim.speed \* speedMultiplier;

if (anim.progress >= 1) {

anim.progress = 1;

if (anim.onComplete) anim.onComplete();

this.state.animation = null;

}

}

if (this.state.impactAnimation) {

this.state.impactAnimation.progress += 0.05 \* speedMultiplier;

if (this.state.impactAnimation.progress >= 1) {

this.state.impactAnimation = null;

}

}

if (this.state.fireballAnimation) {

this.state.fireballAnimation.progress += 0.025 \* speedMultiplier;

if (this.state.fireballAnimation.progress >= 1) {

this.state.fireballAnimation = null;

}

}

if (this.state.overloadAnimation) {

this.state.overloadAnimation.progress += 0.075 \* speedMultiplier;

if (this.state.overloadAnimation.progress >= 1) {

this.state.overloadAnimation = null;

}

}

this.state.abilityAnimations.forEach(anim => anim.progress += anim.speed \* speedMultiplier);

this.state.abilityAnimations = this.state.abilityAnimations.filter(anim => anim.progress < 1);

this.state.player1Team.forEach(hero => this.updateHeroAnimation(hero, speedMultiplier));

this.state.player2Team.forEach(hero => this.updateHeroAnimation(hero, speedMultiplier));

if (this.state.gamePhase === 'expertVictory') {

this.state.confettiParticles.forEach(p => {

p.y += p.speed;

p.angle += p.rotationSpeed;

if (p.y > this.canvas.height) {

p.y = -p.height;

p.x = Math.random() \* this.canvas.width;

}

});

}

}

updateHeroAnimation(hero, speedMultiplier) {

if (hero.deathAnimation && hero.deathAnimation.progress < 1) {

hero.deathAnimation.progress += 0.015 \* speedMultiplier;

if (hero.deathAnimation.progress > 1) {

hero.deathAnimation.progress = 1;

}

}

}

draw() {

this.ctx.fillStyle = COLORS.background;

this.ctx.fillRect(0, 0, this.canvas.width, this.canvas.height);

this.cardHitboxes = { player1: [], player2: [], selection: [] };

this.buttonHitboxes = {};

switch(this.state.gamePhase) {

case 'mainMenu': this.drawMainMenuScreen(); break;

case 'difficultySelection': this.drawDifficultySelectionScreen(); break;

case 'heroSelection': this.drawHeroSelectionScreen(); break;

case 'practiceSelection': this.drawPracticeSelectionScreen(); break;

case 'heroViewer': this.drawHeroViewerScreen(); break;

case 'playing':

case 'bossBattle': this.drawPlayingScreen(); break;

case 'gameOver': this.drawGameOver(); break;

case 'expertVictory': this.drawExpertVictoryScreen(); break;

}

if (this.state.hoveredHero) {

const { x, y, hero, team } = this.state.hoveredHero;

this.drawCard(x, y, hero, 'hover', team, 1.5);

}

}

drawPlayingScreen() {

this.drawTeam(this.state.player2Team, 'player2');

this.drawTeam(this.state.player1Team, 'player1');

this.drawBattleLog();

this.initiateClashAnimation();

this.drawClashButton();

this.drawBackButton();

if (this.state.animation && this.state.animation.type === 'clash') {

this.drawClashAnimation();

}

if (this.state.impactAnimation) this.drawImpactAnimation();

if (this.state.fireballAnimation) this.drawFireballAnimation();

if (this.state.overloadAnimation) this.drawOverloadAnimation();

this.state.abilityAnimations.forEach(anim => {

if(anim.type === 'smokeBomb') this.drawSmokeBombAnimation(anim);

if(anim.type === 'longBow') this.drawLongBowAnimation(anim);

if(anim.type === 'ambush') this.drawAmbushAnimation(anim);

if(anim.type === 'lastStand') this.drawLastStandAnimation(anim);

if(anim.type === 'cleave') this.drawCleaveAnimation(anim);

if(anim.type === 'intimidation') this.drawIntimidationAnimation(anim);

if(anim.type === 'advantage') this.drawAdvantageIndicator(anim);

});

this.drawRoundCounter();

}

// -----------------------------------

// Drawing Helpers

// -----------------------------------

wrapText(text, x, y, maxWidth, lineHeight) {

const words = text.split(' ');

let line = '';

this.ctx.textAlign = 'center';

for (let n = 0; n < words.length; n++) {

const testLine = line + words[n] + ' ';

const metrics = this.ctx.measureText(testLine);

if (metrics.width > maxWidth && n > 0) {

this.ctx.fillText(line, x, y);

line = words[n] + ' ';

y += lineHeight;

} else {

line = testLine;

}

}

this.ctx.fillText(line, x, y);

}

drawBattleLog() {

const logHeight = 160 \* this.scale;

const width = this.canvas.width \* 0.7;

const x = (this.canvas.width - width) / 2;

const y = this.canvas.height - logHeight;

const height = logHeight - (10 \* this.scale);

this.ctx.fillStyle = 'rgba(0, 0, 0, 0.7)';

this.ctx.strokeStyle = COLORS.buttonBorder;

this.ctx.lineWidth = 2 \* this.scale;

this.ctx.beginPath();

this.ctx.roundRect(x, y, width, height, [15 \* this.scale]);

this.ctx.fill();

this.ctx.stroke();

this.ctx.fillStyle = COLORS.accent;

this.ctx.font = `bold ${20 \* this.scale}px 'Cinzel'`;

this.ctx.textAlign = 'left';

this.ctx.fillText('Battle Log', x + (15 \* this.scale), y + (25 \* this.scale));

let logY = y + height - (15 \* this.scale);

const lineHeight = 18 \* this.scale;

const padding = 15 \* this.scale;

const logsToDraw = this.state.battleLog.slice().reverse();

for (const [roundIndex, roundLog] of logsToDraw.entries()) {

if (logY < y + (35 \* this.scale)) break;

for (let i = roundLog.length - 1; i >= 0; i--) {

if (logY < y + (40 \* this.scale)) break;

const entry = roundLog[i];

let currentX = x + padding;

this.ctx.textAlign = 'left';

entry.forEach(part => {

this.ctx.fillStyle = part.color;

this.ctx.font = `${15 \* this.scale}px 'MedievalSharp'`;

this.ctx.fillText(part.text, currentX, logY);

currentX += this.ctx.measureText(part.text).width;

});

logY -= lineHeight;

}

if (logY < y + (35 \* this.scale)) break;

const roundNumber = this.state.round - roundIndex;

if (roundNumber > 0) {

this.ctx.fillStyle = COLORS.accent;

this.ctx.font = `bold ${16 \* this.scale}px 'Cinzel'`;

this.ctx.textAlign = 'center';

this.ctx.fillText(`--- Round ${roundNumber} ---`, x + width / 2, logY);

logY -= lineHeight \* 1.5;

}

}

}

drawTeam(team, playerKey) {

const isBoss = team.length === 1 && team[0].abilities;

const cardScaleFactor = isBoss ? 1.5 : (this.state.teamSize === 5 ? 0.8 : 1);

const cardWidth = this.baseCardWidth \* this.scale \* cardScaleFactor;

const cardHeight = this.baseCardHeight \* this.scale \* cardScaleFactor;

const spacing = 20 \* this.scale \* cardScaleFactor;

const teamSize = team.length;

const totalWidth = teamSize \* cardWidth + (teamSize - 1) \* spacing;

const availableWidth = this.canvas.width;

const startX = (availableWidth - totalWidth) / 2;

const logHeight = 160 \* this.scale;

const y = playerKey === 'player1'

? this.canvas.height - cardHeight - logHeight

: (80 \* this.scale);

const title = playerKey === 'player1' ? "Your Team" : (isBoss ? "THE MINOTAUR" : "Opponent's Team");

this.ctx.fillStyle = COLORS.text;

this.ctx.font = `bold ${30 \* this.scale}px 'Cinzel'`;

this.ctx.textAlign = 'center';

this.ctx.shadowColor = 'black';

this.ctx.shadowBlur = 5;

this.ctx.fillText(title, startX + totalWidth / 2, y - (20 \* this.scale));

this.ctx.shadowBlur = 0;

team.forEach((hero, index) => {

const x = startX + index \* (cardWidth + spacing);

const hitbox = { x, y, width: cardWidth, height: cardHeight, hero };

this.cardHitboxes[playerKey].push(hitbox);

const isAnimating = this.state.animation?.type === 'clash' &&

(this.state.animation.p1.id === hero.id || this.state.animation.p2.id === hero.id);

if (!isAnimating) {

this.drawCard(x, y, hero, playerKey, team, cardScaleFactor);

}

});

}

drawCard(x, y, hero, playerKey, team = [], customScale = 1) {

this.ctx.save();

const cardWidth = this.baseCardWidth \* this.scale \* customScale;

const cardHeight = this.baseCardHeight \* this.scale \* customScale;

const padding = 15 \* this.scale \* customScale;

if (hero.deathAnimation) {

const progress = hero.deathAnimation.progress;

const angle = (Math.PI / 2) \* progress;

this.ctx.translate(x + cardWidth / 2, y + cardHeight / 2);

this.ctx.rotate(angle);

this.ctx.translate(-(x + cardWidth / 2), -(y + cardHeight / 2));

this.ctx.globalAlpha = 1 - progress;

}

this.ctx.fillStyle = hero.hp > 0 ? COLORS[hero.type.toLowerCase()] : COLORS.disabled;

this.ctx.beginPath();

this.ctx.roundRect(x, y, cardWidth, cardHeight, [10 \* this.scale \* customScale]);

this.ctx.fill();

if (hero.hp > 0) {

const gradient = this.ctx.createRadialGradient(

x + cardWidth / 2, y + cardHeight / 2, 0,

x + cardWidth / 2, y + cardHeight / 2, cardWidth \* 0.8

);

gradient.addColorStop(0, 'rgba(255, 255, 255, 0.1)');

gradient.addColorStop(1, 'rgba(255, 255, 255, 0)');

this.ctx.fillStyle = gradient;

this.ctx.fill();

}

const isSelected = (this.state.player1Selection?.id === hero.id) ||

(this.state.player2Selection?.id === hero.id) ||

(this.state.gamePhase === 'heroSelection' && this.state.player1Team.some(h => h.id === hero.id)) ||

(this.state.gamePhase === 'practiceSelection' && (this.state.practiceSelectionState === 'player' ? this.state.player1Team : this.state.player2Team).some(h => h.id === hero.id));

if (isSelected && hero.hp > 0) {

this.ctx.strokeStyle = COLORS.accent;

this.ctx.lineWidth = 6 \* this.scale \* customScale;

this.ctx.shadowColor = COLORS.accent;

this.ctx.shadowBlur = 20 \* this.scale \* customScale;

} else {

this.ctx.strokeStyle = COLORS.cardBorder;

this.ctx.lineWidth = 4 \* this.scale \* customScale;

}

this.ctx.stroke();

this.ctx.shadowBlur = 0;

this.ctx.globalAlpha = hero.deathAnimation ? 1 - hero.deathAnimation.progress : 1;

this.ctx.fillStyle = COLORS.text;

this.ctx.textAlign = 'center';

this.ctx.font = `bold ${24 \* this.scale \* customScale}px 'Cinzel'`;

this.ctx.fillText(hero.name, x + cardWidth / 2, y + padding + (10 \* this.scale \* customScale));

const iconSize = 30 \* this.scale \* customScale;

this.drawTypeIcon(hero.type, x + cardWidth / 2, y + padding + (45 \* this.scale \* customScale), iconSize);

const statY = y + cardHeight - padding - (55 \* this.scale \* customScale);

const iconStatSize = 24 \* this.scale \* customScale;

this.drawHeartIcon(x + padding, statY, iconStatSize, iconStatSize);

this.drawAPSwordIcon(x + cardWidth - padding - iconStatSize, statY, iconStatSize, iconStatSize);

const currentAP = this.getHeroCurrentAP(hero, team);

this.ctx.font = `bold ${28 \* this.scale \* customScale}px 'MedievalSharp'`;

this.ctx.textAlign = 'left';

this.ctx.fillText(`${hero.hp}`, x + padding + iconStatSize + (5 \* this.scale \* customScale), statY + iconStatSize \* 0.8);

this.ctx.textAlign = 'right';

this.ctx.fillText(`${currentAP}`, x + cardWidth - padding - iconStatSize - (5 \* this.scale \* customScale), statY + iconStatSize \* 0.8);

this.ctx.textAlign = 'center';

this.ctx.font = `italic ${14 \* this.scale \* customScale}px 'MedievalSharp'`;

this.wrapText(hero.description || '', x + cardWidth / 2, y + cardHeight / 2 - (10 \* this.scale \* customScale), cardWidth - padding \* 1.5, 16 \* this.scale \* customScale);

if (hero.hp <= 0) {

this.ctx.globalAlpha = hero.deathAnimation ? hero.deathAnimation.progress \* 0.8 : 0.8;

this.drawSkullIcon(x + cardWidth / 2, y + cardHeight / 2, cardWidth \* 0.5);

}

this.ctx.restore();

}

drawClashAnimation() {

const anim = this.state.animation;

if (!anim) return;

const { p1, p2, p1StartPos, p2StartPos } = anim;

const cardWidth = this.baseCardWidth \* this.scale;

const cardHeight = this.baseCardHeight \* this.scale;

const p1MeetX = this.canvas.width / 2 - cardWidth;

const p1MeetY = this.canvas.height / 2 - cardHeight / 2;

const p2MeetX = this.canvas.width / 2;

const p2MeetY = this.canvas.height / 2 - cardHeight / 2;

let p1CurrentX, p1CurrentY, p2CurrentX, p2CurrentY;

const rawProgress = anim.progress;

if (rawProgress < 0.5) {

const phaseProgress = rawProgress / 0.5;

const easeOut = t => t \* (2 - t);

p1CurrentX = p1StartPos.x + (p1MeetX - p1StartPos.x) \* easeOut(phaseProgress);

p1CurrentY = p1StartPos.y + (p1MeetY - p1StartPos.y) \* easeOut(phaseProgress);

p2CurrentX = p2StartPos.x + (p2MeetX - p2StartPos.x) \* easeOut(phaseProgress);

p2CurrentY = p2StartPos.y + (p2MeetY - p2StartPos.y) \* easeOut(phaseProgress);

} else {

p1CurrentX = p1MeetX;

p1CurrentY = p1MeetY;

p2CurrentX = p2MeetX;

p2CurrentY = p2MeetY;

const lungePhaseProgress = (rawProgress - 0.5) / 0.5;

const lungeFactor = Math.sin(lungePhaseProgress \* Math.PI);

const lungeDistance = 50 \* this.scale \* lungeFactor;

p1CurrentX += lungeDistance;

p2CurrentX -= lungeDistance;

if (lungePhaseProgress > 0.45 && lungePhaseProgress < 0.55 && !anim.impactTriggered) {

anim.impactTriggered = true;

this.triggerImpact();

this.calculateClashResult(p1, p2);

}

}

const lungeProgress = Math.max(0, (rawProgress - 0.5) / 0.5);

const shakeAngle = Math.sin(lungeProgress \* Math.PI \* 4) \* 0.05;

this.ctx.save();

this.ctx.translate(p1CurrentX + cardWidth/2, p1CurrentY + cardHeight/2);

this.ctx.rotate(shakeAngle);

this.drawCard(-cardWidth/2, -cardHeight/2, p1, 'player1', this.state.player1Team);

this.ctx.restore();

this.ctx.save();

this.ctx.translate(p2CurrentX + cardWidth/2, p2CurrentY + cardHeight/2);

this.ctx.rotate(-shakeAngle);

this.drawCard(-cardWidth/2, -cardHeight/2, p2, 'player2', this.state.player2Team, this.state.gamePhase === 'bossBattle' ? 1.5 : 1);

this.ctx.restore();

}

drawImpactAnimation() {

const anim = this.state.impactAnimation;

if (!anim) return;

const progress = anim.progress;

const flash = Math.sin(progress \* Math.PI);

this.ctx.save();

this.ctx.translate(anim.x, anim.y);

this.ctx.globalAlpha = flash;

this.ctx.strokeStyle = 'white';

this.ctx.lineWidth = (5 \* (1 - progress)) \* this.scale;

this.ctx.beginPath();

this.ctx.arc(0, 0, anim.radius \* progress, 0, 2 \* Math.PI);

this.ctx.stroke();

this.ctx.strokeStyle = COLORS.accent;

this.ctx.lineWidth = (15 \* (1 - progress)) \* this.scale;

const numSpikes = 12;

for (let i = 0; i < numSpikes; i++) {

const angle = (i / numSpikes) \* (2 \* Math.PI);

const startRadius = anim.radius \* 0.3 \* progress;

const endRadius = anim.radius \* progress;

this.ctx.beginPath();

this.ctx.moveTo(Math.cos(angle) \* startRadius, Math.sin(angle) \* startRadius);

this.ctx.lineTo(Math.cos(angle) \* endRadius, Math.sin(angle) \* endRadius);

this.ctx.stroke();

}

this.ctx.restore();

}

drawFireballAnimation() {

const anim = this.state.fireballAnimation;

if (!anim) return;

const cardWidth = this.baseCardWidth \* this.scale;

this.cardHitboxes[anim.targetTeamKey].forEach((hitbox, index) => {

const explosionProgress = Math.max(0, (anim.progress - (index \* 0.2)) / (1 - (index \* 0.2)));

if (explosionProgress > 0) {

const flash = Math.sin(explosionProgress \* Math.PI);

const centerX = hitbox.x + cardWidth / 2;

const centerY = hitbox.y + hitbox.height / 2;

this.ctx.save();

this.ctx.globalAlpha = flash;

this.ctx.fillStyle = COLORS.accent;

this.ctx.beginPath();

this.ctx.arc(centerX, centerY, (cardWidth \* 0.75) \* flash, 0, Math.PI \* 2);

this.ctx.fill();

this.ctx.restore();

}

});

}

drawOverloadAnimation() {

const anim = this.state.overloadAnimation;

if (!anim) return;

const flash = Math.sin(anim.progress \* Math.PI);

this.ctx.save();

this.ctx.strokeStyle = COLORS.accent;

this.ctx.lineWidth = 5 \* this.scale \* flash;

this.ctx.globalAlpha = flash;

this.ctx.beginPath();

this.ctx.moveTo(anim.startX, anim.startY);

const dx = anim.endX - anim.startX;

const dy = anim.endY - anim.startY;

for (let i = 0; i < 1; i += 0.1) {

this.ctx.lineTo(anim.startX + dx \* i + (Math.random() - 0.5) \* 60 \* this.scale, anim.startY + dy \* i + (Math.random() - 0.5) \* 60 \* this.scale);

}

this.ctx.lineTo(anim.endX, anim.endY);

this.ctx.stroke();

this.ctx.restore();

}

drawLastStandAnimation(anim) {

const heroBox = [...this.cardHitboxes.player1, ...this.cardHitboxes.player2].find(box => box.hero.id === anim.hero.id);

if (!heroBox) return;

const progress = anim.progress;

const opacity = Math.sin(progress \* Math.PI);

const radius = heroBox.width \* (1 + progress);

this.ctx.save();

this.ctx.globalAlpha = opacity;

this.ctx.strokeStyle = COLORS.accent;

this.ctx.lineWidth = 10 \* this.scale \* (1 - progress);

this.ctx.beginPath();

this.ctx.arc(heroBox.x + heroBox.width / 2, heroBox.y + heroBox.height / 2, radius, 0, Math.PI \* 2);

this.ctx.stroke();

this.ctx.restore();

}

drawSmokeBombAnimation(anim) {

const heroBox = [...this.cardHitboxes.player1, ...this.cardHitboxes.player2].find(box => box.hero.id === anim.hero.id);

if (!heroBox) return;

const progress = anim.progress;

const opacity = Math.sin(progress \* Math.PI);

this.ctx.save();

this.ctx.globalAlpha = opacity;

this.ctx.fillStyle = '#aaaaaa';

for(let i = 0; i < 5; i++) {

const xOffset = (Math.random() - 0.5) \* heroBox.width \* progress;

const yOffset = (Math.random() - 0.5) \* heroBox.height \* progress;

const radius = heroBox.width / 2 \* (1 - progress) \* (Math.random() \* 0.5 + 0.5);

this.ctx.beginPath();

this.ctx.arc(heroBox.x + heroBox.width / 2 + xOffset, heroBox.y + heroBox.height / 2 + yOffset, radius, 0, Math.PI \* 2);

this.ctx.fill();

}

this.ctx.restore();

}

drawLongBowAnimation(anim) {

const progress = anim.progress;

const easeOut = t => 1 - Math.pow(1 - t, 3);

const currentX = anim.startX + (anim.endX - anim.startX) \* easeOut(progress);

const currentY = anim.startY + (anim.endY - anim.startY) \* easeOut(progress);

const angle = Math.atan2(anim.endY - anim.startY, anim.endX - anim.startX);

this.ctx.save();

this.ctx.translate(currentX, currentY);

this.ctx.rotate(angle);

this.ctx.fillStyle = COLORS.accent;

this.ctx.strokeStyle = COLORS.buttonBorder;

this.ctx.lineWidth = 4 \* this.scale;

this.ctx.beginPath();

this.ctx.moveTo(20 \* this.scale, 0);

this.ctx.lineTo(-10 \* this.scale, -10 \* this.scale);

this.ctx.lineTo(-10 \* this.scale, 10 \* this.scale);

this.ctx.closePath();

this.ctx.fill();

this.ctx.stroke();

this.ctx.beginPath();

this.ctx.moveTo(-10 \* this.scale, 0);

this.ctx.lineTo(-50 \* this.scale, 0);

this.ctx.stroke();

this.ctx.restore();

}

drawAmbushAnimation(anim) {

const heroBox = [...this.cardHitboxes.player1, ...this.cardHitboxes.player2].find(box => box.hero.id === anim.target.id);

if (!heroBox) return;

const progress = anim.progress;

const opacity = Math.sin(progress \* Math.PI);

const length = heroBox.width \* 1.5 \* Math.sin(progress \* Math.PI \* 0.5);

this.ctx.save();

this.ctx.globalAlpha = opacity;

this.ctx.strokeStyle = COLORS.might;

this.ctx.lineWidth = 16 \* this.scale;

this.ctx.translate(heroBox.x + heroBox.width / 2, heroBox.y + heroBox.height / 2);

this.ctx.rotate(-Math.PI / 4);

this.ctx.beginPath();

this.ctx.moveTo(-length / 2, 0);

this.ctx.lineTo(length / 2, 0);

this.ctx.stroke();

this.ctx.restore();

}

drawCleaveAnimation(anim) {

const bossBox = this.cardHitboxes.player2[0];

if (!bossBox) return;

const progress = anim.progress;

const arcProgress = Math.sin(progress \* Math.PI \* 0.5);

this.ctx.save();

this.ctx.strokeStyle = COLORS.might;

this.ctx.lineWidth = 15 \* this.scale;

this.ctx.globalAlpha = Math.sin(progress \* Math.PI);

const radius = this.canvas.width \* 0.4 \* arcProgress;

this.ctx.beginPath();

this.ctx.arc(bossBox.x + bossBox.width / 2, bossBox.y + bossBox.height / 2, radius, Math.PI \* 0.7, Math.PI \* 1.3);

this.ctx.stroke();

this.ctx.restore();

}

drawIntimidationAnimation(anim) {

const bossBox = this.cardHitboxes.player2[0];

if (!bossBox) return;

const progress = anim.progress;

const opacity = Math.sin(progress \* Math.PI);

const radius = bossBox.width \* (0.5 + progress);

this.ctx.save();

this.ctx.globalAlpha = opacity \* 0.5;

this.ctx.fillStyle = '#333333';

this.ctx.beginPath();

this.ctx.arc(bossBox.x + bossBox.width / 2, bossBox.y + bossBox.height / 2, radius, 0, Math.PI \* 2);

this.ctx.fill();

this.ctx.restore();

}

drawAdvantageIndicator(anim) {

const { hitbox, color } = anim;

const progress = anim.progress;

const opacity = Math.sin(progress \* Math.PI);

const scale = 1 + (0.5 \* progress);

this.ctx.save();

this.ctx.globalAlpha = opacity;

this.ctx.strokeStyle = color;

this.ctx.lineWidth = 8 \* this.scale \* (1 - progress);

this.ctx.translate(hitbox.x + hitbox.width / 2, hitbox.y + hitbox.height / 2);

this.ctx.scale(scale, scale);

for (let i = 0; i < 3; i++) {

this.ctx.beginPath();

this.ctx.moveTo(-20 \* this.scale, (10 - i \* 15) \* this.scale);

this.ctx.lineTo(0, (-5 - i \* 15) \* this.scale);

this.ctx.lineTo(20 \* this.scale, (10 - i \* 15) \* this.scale);

this.ctx.stroke();

}

this.ctx.restore();

}

drawButton(hitboxKey, text, colorScheme) {

const hitbox = this.buttonHitboxes[hitboxKey];

if (!hitbox) return;

const isHovered = this.isClickInHitbox(this.mousePos.x, this.mousePos.y, hitbox);

const colors = isHovered ? COLORS.buttonHover : colorScheme;

const gradient = this.ctx.createLinearGradient(hitbox.x, hitbox.y, hitbox.x, hitbox.y + hitbox.height);

gradient.addColorStop(0, colors.start);

gradient.addColorStop(1, colors.end);

this.ctx.fillStyle = gradient;

this.ctx.beginPath();

this.ctx.roundRect(hitbox.x, hitbox.y, hitbox.width, hitbox.height, [10 \* this.scale]);

this.ctx.fill();

this.ctx.strokeStyle = COLORS.buttonBorder;

this.ctx.lineWidth = 4 \* this.scale;

this.ctx.stroke();

this.ctx.fillStyle = COLORS.text;

this.ctx.font = `bold ${hitbox.height \* 0.4}px "Cinzel"`;

this.ctx.textAlign = 'center';

this.ctx.textBaseline = 'middle';

this.ctx.shadowColor = 'black';

this.ctx.shadowBlur = 5;

this.ctx.fillText(text, hitbox.x + hitbox.width / 2, hitbox.y + hitbox.height / 2);

this.ctx.shadowBlur = 0;

this.ctx.textBaseline = 'alphabetic';

}

drawClashButton() {

if (!this.state.player1Selection || this.state.isClashing || this.state.winner) return;

const buttonWidth = 200 \* this.scale;

const buttonHeight = 60 \* this.scale;

const x = (this.canvas.width - buttonWidth) / 2;

const y = (this.canvas.height - buttonHeight) / 2;

this.buttonHitboxes.clash = { x, y, width: buttonWidth, height: buttonHeight };

this.drawButton('clash', 'CLASH!', {start: COLORS.accent, end: '#B8860B'});

}

drawBackButton() {

const buttonWidth = 200 \* this.scale;

const buttonHeight = 60 \* this.scale;

const x = this.canvas.width - buttonWidth - (20 \* this.scale);

const y = 20 \* this.scale;

this.buttonHitboxes.backToMenu = { x, y, width: buttonWidth, height: buttonHeight };

this.drawButton('backToMenu', 'Main Menu', {start: COLORS.might, end: COLORS.might});

}

drawRoundCounter() {

this.ctx.fillStyle = COLORS.text;

this.ctx.font = `bold ${30 \* this.scale}px "Cinzel"`;

this.ctx.textAlign = 'center';

this.ctx.shadowColor = 'black';

this.ctx.shadowBlur = 5;

const x = this.canvas.width / 2;

this.ctx.fillText(`Round ${this.state.round}`, x, 40 \* this.scale);

this.ctx.shadowBlur = 0;

}

drawGameOver() {

this.ctx.fillStyle = 'rgba(0, 0, 0, 0.8)';

this.ctx.fillRect(0, 0, this.canvas.width, this.canvas.height);

this.ctx.fillStyle = COLORS.accent;

this.ctx.font = `bold ${60 \* this.scale}px "Cinzel"`;

this.ctx.textAlign = 'center';

const message = this.state.winner === 'p1' ? 'VICTORY!' : 'DEFEAT';

this.ctx.fillText(message, this.canvas.width / 2, this.canvas.height / 2 - (100 \* this.scale));

const buttonWidth = 250 \* this.scale;

const buttonHeight = 70 \* this.scale;

const x = (this.canvas.width - buttonWidth) / 2;

const y = this.canvas.height / 2;

this.buttonHitboxes.playAgain = { x, y, width: buttonWidth, height: buttonHeight };

this.drawButton('playAgain', 'Main Menu', COLORS.button);

}

drawExpertVictoryScreen() {

this.ctx.fillStyle = 'rgba(0, 0, 0, 0.8)';

this.ctx.fillRect(0, 0, this.canvas.width, this.canvas.height);

this.state.confettiParticles.forEach(p => {

this.ctx.save();

this.ctx.fillStyle = p.color;

this.ctx.translate(p.x, p.y);

this.ctx.rotate(p.angle);

this.ctx.fillRect(-p.width / 2, -p.height / 2, p.width, p.height);

this.ctx.restore();

});

this.ctx.fillStyle = COLORS.accent;

this.ctx.shadowColor = '#000';

this.ctx.shadowBlur = 15;

this.ctx.font = `bold ${70 \* this.scale}px "Cinzel"`;

this.ctx.textAlign = 'center';

this.ctx.fillText('EXPERT VICTORY!', this.canvas.width / 2, this.canvas.height / 2 - 50 \* this.scale);

this.ctx.fillStyle = COLORS.text;

this.ctx.font = `bold ${30 \* this.scale}px "MedievalSharp"`;

this.ctx.fillText('You have mastered Trinity Clash!', this.canvas.width / 2, this.canvas.height / 2 + 20 \* this.scale);

this.ctx.shadowBlur = 0;

const buttonWidth = 250 \* this.scale;

const buttonHeight = 70 \* this.scale;

const x = (this.canvas.width - buttonWidth) / 2;

const y = this.canvas.height / 2 + 100 \* this.scale;

this.buttonHitboxes.playAgain = { x, y, width: buttonWidth, height: buttonHeight };

this.drawButton('playAgain', 'Main Menu', COLORS.button);

}

drawTitleSwordIcon(x, y, width, height) {

this.ctx.save();

this.ctx.strokeStyle = COLORS.buttonBorder;

this.ctx.lineWidth = 3 \* this.scale;

const pommelRadius = (width \* 0.4) / 2;

this.ctx.fillStyle = COLORS.might;

this.ctx.beginPath();

this.ctx.arc(x + width / 2, y + pommelRadius, pommelRadius, 0, Math.PI \* 2);

this.ctx.fill();

this.ctx.stroke();

const hiltWidth = width \* 0.4;

const hiltHeight = height \* 0.2;

const hiltY = y + pommelRadius \* 2;

this.ctx.fillRect(x + (width - hiltWidth) / 2, hiltY, hiltWidth, hiltHeight);

this.ctx.strokeRect(x + (width - hiltWidth) / 2, hiltY, hiltWidth, hiltHeight);

const guardHeight = height \* 0.1;

const guardY = hiltY + hiltHeight;

this.ctx.fillStyle = COLORS.accent;

this.ctx.fillRect(x, guardY, width, guardHeight);

this.ctx.strokeRect(x, guardY, width, guardHeight);

const bladeWidth = width \* 0.4;

const bladeHeight = height \* 0.65;

const bladeY = guardY + guardHeight;

this.ctx.fillStyle = '#BDBDBD';

this.ctx.fillRect(x + (width - bladeWidth) / 2, bladeY, bladeWidth, bladeHeight);

this.ctx.strokeRect(x + (width - bladeWidth) / 2, bladeY, bladeWidth, bladeHeight);

const tipY = bladeY + bladeHeight;

this.ctx.beginPath();

this.ctx.moveTo(x + (width - bladeWidth) / 2, tipY);

this.ctx.lineTo(x + width / 2, tipY + height \* 0.1);

this.ctx.lineTo(x + (width + bladeWidth) / 2, tipY);

this.ctx.closePath();

this.ctx.fill();

this.ctx.stroke();

this.ctx.restore();

}

drawMainMenuScreen() {

this.ctx.fillStyle = COLORS.text;

this.ctx.font = `bold ${80 \* this.scale}px "Cinzel"`;

this.ctx.shadowColor = 'black';

this.ctx.shadowBlur = 10;

const titleY = this.canvas.height / 2 - 350 \* this.scale;

const remainingText = 'rinity Clash';

const textMetrics = this.ctx.measureText(remainingText);

const swordWidth = 60 \* this.scale;

const swordHeight = 80 \* this.scale;

const totalTitleWidth = swordWidth + textMetrics.width;

const startX = (this.canvas.width - totalTitleWidth) / 2;

const swordY = titleY - swordHeight;

this.drawTitleSwordIcon(startX, swordY, swordWidth, swordHeight);

this.ctx.textAlign = 'left';

this.ctx.textBaseline = 'alphabetic';

this.ctx.fillText(remainingText, startX + swordWidth, titleY);

this.ctx.textAlign = 'center';

this.ctx.shadowBlur = 0;

const menuItems = ['3v3 Clash', '5v5 Clash', 'Boss Battle', 'Practice', 'Hero Viewer'];

const buttonWidth = 300 \* this.scale;

const buttonHeight = 80 \* this.scale;

const spacing = 30 \* this.scale;

const totalHeight = menuItems.length \* buttonHeight + (menuItems.length - 1) \* spacing;

const menuStartY = (this.canvas.height - totalHeight) / 2;

menuItems.forEach((item, index) => {

const x = (this.canvas.width - buttonWidth) / 2;

const y = menuStartY + index \* (buttonHeight + spacing);

const key = item.toLowerCase().replace(/ /g, '');

this.buttonHitboxes[key] = { x, y, width: buttonWidth, height: buttonHeight };

this.drawButton(key, item, COLORS.button);

});

}

drawDifficultySelectionScreen() {

this.ctx.fillStyle = COLORS.text;

this.ctx.textAlign = 'center';

this.ctx.font = `bold ${48 \* this.scale}px "Cinzel"`;

this.ctx.fillText('Select Difficulty', this.canvas.width / 2, this.canvas.height / 2 - 200 \* this.scale);

const difficulties = ['Easy', 'Normal', 'Expert'];

const buttonWidth = 300 \* this.scale;

const buttonHeight = 80 \* this.scale;

const spacing = 30 \* this.scale;

const totalHeight = difficulties.length \* buttonHeight + (difficulties.length - 1) \* spacing;

const startY = (this.canvas.height - totalHeight) / 2;

difficulties.forEach((level, index) => {

const x = (this.canvas.width - buttonWidth) / 2;

const y = startY + index \* (buttonHeight + spacing);

const key = level.toLowerCase();

this.buttonHitboxes[key] = { x, y, width: buttonWidth, height: buttonHeight };

this.drawButton(key, level, COLORS.button);

});

this.drawBackButton();

}

drawHeroSelectionScreen() {

const cardWidth = this.baseCardWidth \* this.scale;

const cardHeight = this.baseCardHeight \* this.scale;

const spacing = 20 \* this.scale;

const allHeroes = JSON.parse(JSON.stringify(HEROES));

this.ctx.fillStyle = COLORS.text;

this.ctx.textAlign = 'center';

this.ctx.font = `bold ${48 \* this.scale}px "Cinzel"`;

this.ctx.fillText('Choose Your Team', this.canvas.width / 2, 80 \* this.scale);

this.ctx.font = `${24 \* this.scale}px "MedievalSharp"`;

this.ctx.fillText(`Select ${this.state.teamSize} heroes (${this.state.player1Team.length} / ${this.state.teamSize})`, this.canvas.width / 2, 130 \* this.scale);

const cardsPerRow = Math.max(1, Math.floor((this.canvas.width - spacing) / (cardWidth + spacing)));

const numRows = Math.ceil(allHeroes.length / cardsPerRow);

const rowSpacing = 30 \* this.scale;

const gridWidth = cardsPerRow \* cardWidth + (cardsPerRow - 1) \* spacing;

const gridHeight = numRows \* cardHeight + (numRows - 1) \* rowSpacing;

const startX = (this.canvas.width - gridWidth) / 2;

const topMargin = 150 \* this.scale;

const bottomMargin = 150 \* this.scale;

const availableHeight = this.canvas.height - topMargin - bottomMargin;

const startY = topMargin + (availableHeight - gridHeight) / 2;

this.cardHitboxes.selection = [];

allHeroes.forEach((hero, index) => {

const col = index % cardsPerRow;

const row = Math.floor(index / cardsPerRow);

const x = startX + col \* (cardWidth + spacing);

const y = startY + row \* (cardHeight + rowSpacing);

this.cardHitboxes.selection.push({ x, y, width: cardWidth, height: cardHeight, hero });

this.drawCard(x, y, hero, 'selection');

});

if (this.state.player1Team.length === this.state.teamSize) {

const buttonWidth = 300 \* this.scale;

const buttonHeight = 80 \* this.scale;

const btnX = (this.canvas.width - buttonWidth) / 2;

const btnY = this.canvas.height - (150 \* this.scale);

const buttonText = this.state.difficulty === 'boss' ? 'Face the Boss!' : 'Start Game';

this.buttonHitboxes.startGame = { x: btnX, y: btnY, width: buttonWidth, height: buttonHeight };

this.drawButton('startGame', buttonText, {start: COLORS.finesse, end: COLORS.finesse});

}

this.drawBackButton();

}

drawPracticeSelectionScreen() {

const cardWidth = this.baseCardWidth \* this.scale;

const cardHeight = this.baseCardHeight \* this.scale;

const spacing = 20 \* this.scale;

const allHeroes = JSON.parse(JSON.stringify(HEROES));

this.ctx.fillStyle = COLORS.text;

this.ctx.textAlign = 'center';

const title = this.state.practiceSelectionState === 'player' ? 'Choose Your Team' : 'Choose Opponent\'s Team';

const team = this.state.practiceSelectionState === 'player' ? this.state.player1Team : this.state.player2Team;

this.ctx.font = `bold ${48 \* this.scale}px "Cinzel"`;

this.ctx.fillText(title, this.canvas.width / 2, 80 \* this.scale);

this.ctx.font = `${24 \* this.scale}px "MedievalSharp"`;

this.ctx.fillText(`Select ${this.state.teamSize} heroes (${team.length} / ${this.state.teamSize})`, this.canvas.width / 2, 130 \* this.scale);

const cardsPerRow = Math.max(1, Math.floor((this.canvas.width - spacing) / (cardWidth + spacing)));

const numRows = Math.ceil(allHeroes.length / cardsPerRow);

const rowSpacing = 30 \* this.scale;

const gridWidth = cardsPerRow \* cardWidth + (cardsPerRow - 1) \* spacing;

const gridHeight = numRows \* cardHeight + (numRows - 1) \* rowSpacing;

const startX = (this.canvas.width - gridWidth) / 2;

const topMargin = 150 \* this.scale;

const bottomMargin = 150 \* this.scale;

const availableHeight = this.canvas.height - topMargin - bottomMargin;

const startY = topMargin + (availableHeight - gridHeight) / 2;

this.cardHitboxes.selection = [];

allHeroes.forEach((hero, index) => {

const col = index % cardsPerRow;

const row = Math.floor(index / cardsPerRow);

const x = startX + col \* (cardWidth + spacing);

const y = startY + row \* (cardHeight + rowSpacing);

this.cardHitboxes.selection.push({ x, y, width: cardWidth, height: cardHeight, hero });

this.drawCard(x, y, hero, 'selection');

});

if (team.length === this.state.teamSize) {

const buttonWidth = 300 \* this.scale;

const buttonHeight = 80 \* this.scale;

const btnX = (this.canvas.width - buttonWidth) / 2;

const btnY = this.canvas.height - (150 \* this.scale);

const buttonText = this.state.practiceSelectionState === 'player' ? 'Confirm Your Team' : 'Start Practice';

this.buttonHitboxes.confirmTeam = { x: btnX, y: btnY, width: buttonWidth, height: buttonHeight };

this.drawButton('confirmTeam', buttonText, {start: COLORS.finesse, end: COLORS.finesse});

}

this.drawBackButton();

}

drawHeroViewerScreen() {

const cardWidth = this.baseCardWidth \* this.scale \* 2;

const cardHeight = this.baseCardHeight \* this.scale \* 2;

const x = (this.canvas.width - cardWidth) / 2;

const y = (this.canvas.height - cardHeight) / 2 - 50 \* this.scale;

const hero = HEROES[this.state.heroViewerIndex];

this.drawCard(x, y, hero, 'viewer', [], 2);

const buttonWidth = 200 \* this.scale;

const buttonHeight = 70 \* this.scale;

const btnY = y + cardHeight + 50 \* this.scale;

const prevX = x;

this.buttonHitboxes.previous = { x: prevX, y: btnY, width: buttonWidth, height: buttonHeight };

this.drawButton('previous', 'Previous', COLORS.button);

const nextX = x + cardWidth - buttonWidth;

this.buttonHitboxes.next = { x: nextX, y: btnY, width: buttonWidth, height: buttonHeight };

this.drawButton('next', 'Next', COLORS.button);

const backX = (this.canvas.width - buttonWidth) / 2;

const backY = btnY + buttonHeight + 30 \* this.scale;

this.buttonHitboxes.back = { x: backX, y: backY, width: buttonWidth, height: buttonHeight };

this.drawButton('back', 'Back', {start: COLORS.might, end: COLORS.might});

}

drawHeartIcon(x, y, width, height) {

this.ctx.save();

this.ctx.fillStyle = '#E53935';

this.ctx.strokeStyle = '#C62828';

this.ctx.lineWidth = 2 \* this.scale;

this.ctx.beginPath();

this.ctx.moveTo(x + width / 2, y + height \* 0.35);

this.ctx.bezierCurveTo(x + width / 2, y + height \* 0.1, x, y, x, y + height / 2.5);

this.ctx.bezierCurveTo(x, y + height, x + width, y + height, x + width, y + height / 2.5);

this.ctx.bezierCurveTo(x + width, y, x + width / 2, y + height \* 0.1, x + width / 2, y + height \* 0.35);

this.ctx.fill();

this.ctx.stroke();

this.ctx.restore();

}

drawAPSwordIcon(x, y, width, height) {

this.ctx.save();

this.ctx.fillStyle = '#78909C';

this.ctx.strokeStyle = '#455A64';

this.ctx.lineWidth = 2 \* this.scale;

this.ctx.beginPath();

this.ctx.moveTo(x + width \* 0.5, y);

this.ctx.lineTo(x + width \* 0.6, y + height \* 0.6);

this.ctx.lineTo(x + width \* 0.4, y + height \* 0.6);

this.ctx.closePath();

this.ctx.fill();

this.ctx.stroke();

this.ctx.fillStyle = '#A1887F';

this.ctx.fillRect(x, y + height \* 0.6, width, height \* 0.15);

this.ctx.strokeRect(x, y + height \* 0.6, width, height \* 0.15);

this.ctx.fillRect(x + width \* 0.35, y + height \* 0.75, width \* 0.3, height \* 0.25);

this.ctx.strokeRect(x + width \* 0.35, y + height \* 0.75, width \* 0.3, height \* 0.25);

this.ctx.restore();

}

drawTypeIcon(type, x, y, size) {

this.ctx.save();

this.ctx.strokeStyle = 'rgba(0,0,0,0.5)';

this.ctx.lineWidth = size \* 0.1;

this.ctx.font = `bold ${size \* 0.8}px "Cinzel"`;

this.ctx.beginPath();

if (type === 'Might') {

this.ctx.roundRect(x - size / 2, y - size / 2, size, size, size \* 0.2);

this.ctx.stroke();

this.ctx.fillText('✊', x, y + size \* 0.2);

} else if (type === 'Finesse') {

this.ctx.ellipse(x, y, size / 2, size / 3, 0, 0, Math.PI \* 2);

this.ctx.stroke();

this.ctx.fillText('🍃', x, y + size \* 0.2);

} else if (type === 'Magic') {

this.ctx.moveTo(x, y - size / 2);

this.ctx.lineTo(x + size \* 0.15, y - size \* 0.15);

this.ctx.lineTo(x + size / 2, y);

this.ctx.lineTo(x + size \* 0.15, y + size \* 0.15);

this.ctx.lineTo(x, y + size / 2);

this.ctx.lineTo(x - size \* 0.15, y + size \* 0.15);

this.ctx.lineTo(x - size / 2, y);

this.ctx.lineTo(x - size \* 0.15, y - size \* 0.15);

this.ctx.closePath();

this.ctx.stroke();

this.ctx.fillText('✨', x, y + size \* 0.2);

}

this.ctx.restore();

}

drawSkullIcon(x, y, size) {

this.ctx.save();

this.ctx.fillStyle = 'rgba(20, 20, 20, 0.8)';

this.ctx.strokeStyle = 'rgba(0, 0, 0, 0.9)';

this.ctx.lineWidth = size \* 0.05;

// Skull

this.ctx.beginPath();

this.ctx.arc(x, y - size \* 0.1, size \* 0.25, Math.PI \* 0.8, Math.PI \* 2.2);

this.ctx.lineTo(x + size \* 0.15, y + size \* 0.25);

this.ctx.arc(x, y + size \* 0.25, size \* 0.15, 0, Math.PI);

this.ctx.closePath();

this.ctx.fill();

this.ctx.stroke();

// Eyes

this.ctx.fillStyle = '#000';

this.ctx.beginPath();

this.ctx.arc(x - size \* 0.1, y - size \* 0.15, size \* 0.06, 0, Math.PI \* 2);

this.ctx.arc(x + size \* 0.1, y - size \* 0.15, size \* 0.06, 0, Math.PI \* 2);

this.ctx.fill();

// Crossbones

this.ctx.save();

this.ctx.translate(x, y + size \* 0.1);

this.ctx.rotate(Math.PI / 4);

this.ctx.fillRect(-size \* 0.3, -size \* 0.05, size \* 0.6, size \* 0.1);

this.ctx.strokeRect(-size \* 0.3, -size \* 0.05, size \* 0.6, size \* 0.1);

this.ctx.rotate(Math.PI / 2);

this.ctx.fillRect(-size \* 0.3, -size \* 0.05, size \* 0.6, size \* 0.1);

this.ctx.strokeRect(-size \* 0.3, -size \* 0.05, size \* 0.6, size \* 0.1);

this.ctx.restore();

this.ctx.restore();

}

// -----------------------------------

// Game Logic

// -----------------------------------

log(messageParts) {

if (this.state.battleLog.length > 0) {

this.state.battleLog[this.state.battleLog.length - 1].push(messageParts);

}

}

startNewRound() {

this.state.round++;

this.state.battleLog.push([]);

if (this.state.battleLog.length > 2) {

this.state.battleLog.shift();

}

this.state.isClashing = false;

this.state.bossActiveAbility = null;

const p1Theron = this.state.player1Team.find(h => h.abilityId === 'fireball' && h.hp > 0);

if (p1Theron && this.state.round === 5) this.handleFireball(p1Theron, this.state.player2Team, 'player2');

const p2Theron = this.state.player2Team.find(h => h.abilityId === 'fireball' && h.hp > 0);

if (p2Theron && this.state.round === 5) this.handleFireball(p2Theron, this.state.player1Team, 'player1');

const p1Kaelen = this.state.player1Team.find(h => h.abilityId === 'longBow' && h.hp > 0);

if (p1Kaelen && this.state.round % 2 === 0) this.handleLongBow(p1Kaelen, this.state.player2Team);

const p2Kaelen = this.state.player2Team.find(h => h.abilityId === 'longBow' && h.hp > 0);

if (p2Kaelen && this.state.round % 2 === 0) this.handleLongBow(p2Kaelen, this.state.player1Team);

if (!this.state.winner) this.log([{ text: 'Select a hero to clash.', color: COLORS.info }]);

}

handleFireball(caster, targetTeam, targetTeamKey) {

let livingTargets = targetTeam.filter(h => h.hp > 0);

if (livingTargets.length === 0) return;

this.log([

{ text: caster.name, color: COLORS[caster.type.toLowerCase()] },

{ text: ' unleashes a ', color: COLORS.text },

{ text: 'Fireball!', color: COLORS.ability }

]);

this.state.fireballAnimation = { progress: 0, targetTeamKey };

const taunter = livingTargets.find(h => h.abilityId === 'tank' && h.hp > 0);

livingTargets.forEach(target => {

let damage = 5;

let finalTarget = taunter || target;

if(caster.type === 'Magic' && finalTarget.type === 'Might') damage \*= 2;

if (finalTarget.abilityId === 'smokeBomb' && !finalTarget.smokeBombUsed) {

this.log([

{ text: finalTarget.name, color: COLORS[finalTarget.type.toLowerCase()] },

{ text: "'s ", color: COLORS.text },

{ text: 'Smoke Bomb', color: COLORS.ability },

{ text: ' evades the fireball!', color: COLORS.text }

]);

finalTarget.smokeBombUsed = true;

} else {

finalTarget.hp -= damage;

this.log([

{ text: 'Fireball hits ', color: COLORS.ability },

{ text: finalTarget.name, color: COLORS[finalTarget.type.toLowerCase()] },

{ text: ` for ${damage} damage.`, color: COLORS.damage }

]);

this.checkHeroKO(finalTarget);

}

});

this.checkGameOver();

}

handleLongBow(kaelen, targetTeam) {

let livingTargets = targetTeam.filter(h => h.hp > 0);

if (livingTargets.length === 0) return;

let randomTarget = livingTargets[Math.floor(Math.random() \* livingTargets.length)];

const taunter = targetTeam.find(h => h.abilityId === 'tank' && h.hp > 0);

let finalTarget = taunter || randomTarget;

const kaelenBox = [...this.cardHitboxes.player1, ...this.cardHitboxes.player2].find(box => box.hero.id === kaelen.id);

const targetBox = [...this.cardHitboxes.player1, ...this.cardHitboxes.player2].find(box => box.hero.id === finalTarget.id);

if (kaelenBox && targetBox) {

this.state.abilityAnimations.push({

type: 'longBow', progress: 0, speed: 0.04,

startX: kaelenBox.x + kaelenBox.width / 2, startY: kaelenBox.y + kaelenBox.height / 2,

endX: targetBox.x + targetBox.width / 2, endY: targetBox.y + targetBox.height / 2,

});

}

let damage = 2;

if (finalTarget.abilityId === 'smokeBomb' && !finalTarget.smokeBombUsed) {

damage = 0;

finalTarget.smokeBombUsed = true;

this.log([

{ text: kaelen.name, color: COLORS[kaelen.type.toLowerCase()] },

{ text: "'s arrow was dodged by ", color: COLORS.text },

{ text: finalTarget.name, color: COLORS[finalTarget.type.toLowerCase()] },

{ text: '!', color: COLORS.text }

]);

} else {

finalTarget.hp -= damage;

this.log([

{ text: kaelen.name, color: COLORS[kaelen.type.toLowerCase()] },

{ text: "'s arrow hits ", color: COLORS.text },

{ text: finalTarget.name, color: COLORS[finalTarget.type.toLowerCase()] },

{ text: ` for ${damage} damage.`, color: COLORS.damage }

]);

this.checkHeroKO(finalTarget);

}

this.checkGameOver();

}

initiateClashAnimation() {

if (!this.state.pendingClash) return;

const { p1, p2 } = this.state.pendingClash;

const p1StartPos = this.cardHitboxes.player1.find(box => box.hero.id === p1.id);

const p2StartPos = this.cardHitboxes.player2.find(box => box.hero.id === p2.id);

if (p1StartPos && p2StartPos) {

this.state.animation = {

type: 'clash', progress: 0, speed: 0.02, p1, p2,

p1StartPos: { ...p1StartPos }, p2StartPos: { ...p2StartPos },

impactTriggered: false,

onComplete: () => {

setTimeout(() => {

this.state.player1Selection = null;

this.state.player2Selection = null;

this.startNewRound();

}, 1000);

}

};

this.state.pendingClash = null;

}

}

resolveClash() {

if (!this.state.player1Selection || this.state.isClashing) return;

const p1 = this.state.player1Selection;

let p2;

if (this.state.gamePhase === 'bossBattle') {

p2 = this.state.player2Team[0];

const abilities = p2.abilities;

this.state.bossActiveAbility = abilities[Math.floor(Math.random() \* abilities.length)];

this.log([

{ text: 'The Minotaur uses ', color: COLORS.might },

{ text: `${this.state.bossActiveAbility.toUpperCase()}!`, color: COLORS.ability }

]);

} else {

let p2Team = this.state.player2Team.filter(h => h.hp > 0);

if (p2Team.length === 0) { this.checkGameOver(); return; }

if (this.state.difficulty === 'easy' || this.state.difficulty === 'normal') {

p2 = p2Team[Math.floor(Math.random() \* p2Team.length)];

} else {

let bestChoice = null;

let maxScore = -Infinity;

p2Team.forEach(opponent => {

const [p1Dmg, p2Dmg] = this.simulateClash(p1, opponent);

const score = p2Dmg - p1Dmg;

if (score > maxScore) {

maxScore = score;

bestChoice = opponent;

}

});

p2 = bestChoice;

}

}

this.state.isClashing = true;

this.state.player2Selection = p2;

this.log([

{ text: p1.name, color: COLORS[p1.type.toLowerCase()] },

{ text: ' clashes with ', color: COLORS.text },

{ text: p2.name, color: COLORS[p2.type.toLowerCase()] },

{ text: '!', color: COLORS.text }

]);

if (this.state.gamePhase === 'bossBattle') {

if (this.state.bossActiveAbility === 'cleave') {

this.state.abilityAnimations.push({ type: 'cleave', progress: 0, speed: 0.03 });

this.state.player1Team.forEach(hero => {

if (hero.hp > 0) {

hero.hp -= 4;

this.log([

{ text: 'Cleave hits ', color: COLORS.ability },

{ text: hero.name, color: COLORS[hero.type.toLowerCase()] },

{ text: ' for 4 damage!', color: COLORS.damage }

]);

this.checkHeroKO(hero);

}

});

} else if (this.state.bossActiveAbility === 'intimidation') {

this.state.abilityAnimations.push({ type: 'intimidation', progress: 0, speed: 0.03 });

}

} else {

this.state.player1Team.forEach(hero => {

if (hero.hp > 0 && hero.abilityId === 'ambush' && hero.id !== p1.id) {

p2.hp -= 2;

this.log([

{ text: hero.name, color: COLORS[hero.type.toLowerCase()] },

{ text: "'s ", color: COLORS.text },

{ text: 'Ambush', color: COLORS.ability },

{ text: ' hits ', color: COLORS.text },

{ text: p2.name, color: COLORS[p2.type.toLowerCase()] },

{ text: ' for 2 damage!', color: COLORS.damage }

]);

this.state.abilityAnimations.push({ type: 'ambush', target: p2, progress: 0, speed: 0.05 });

this.checkHeroKO(p2);

}

});

this.state.player2Team.forEach(hero => {

if (hero.hp > 0 && hero.abilityId === 'ambush' && hero.id !== p2.id) {

p1.hp -= 2;

this.log([

{ text: hero.name, color: COLORS[hero.type.toLowerCase()] },

{ text: "'s ", color: COLORS.text },

{ text: 'Ambush', color: COLORS.ability },

{ text: ' hits ', color: COLORS.text },

{ text: p1.name, color: COLORS[p1.type.toLowerCase()] },

{ text: ' for 2 damage!', color: COLORS.damage }

]);

this.state.abilityAnimations.push({ type: 'ambush', target: p1, progress: 0, speed: 0.05 });

this.checkHeroKO(p1);

}

});

}

if (this.checkGameOver()) return;

this.state.pendingClash = { p1, p2 };

}

simulateClash(p1, p2) {

let p1Damage = this.getHeroCurrentAP(p1, this.state.player1Team);

let p2Damage = this.getHeroCurrentAP(p2, this.state.player2Team);

const p1HasAdvantage = (p1.type === 'Might' && p2.type === 'Finesse') || (p1.type === 'Finesse' && p2.type === 'Magic') || (p1.type === 'Magic' && p2.type === 'Might');

const p2HasAdvantage = (p2.type === 'Might' && p1.type === 'Finesse') || (p2.type === 'Finesse' && p1.type === 'Magic') || (p2.type === 'Magic' && p1.type === 'Might');

if (p1HasAdvantage) { p1Damage \*= 2; p2Damage = Math.ceil(p2Damage / 2); }

else if (p2HasAdvantage) { p2Damage \*= 2; p1Damage = Math.ceil(p1Damage / 2); }

return [p1Damage, p2Damage];

}

triggerImpact() {

this.state.impactAnimation = { x: this.canvas.width / 2, y: this.canvas.height / 2, progress: 0, radius: 240 \* this.scale };

}

triggerOverloadAnimation(caster, target) {

const casterBox = [...this.cardHitboxes.player1, ...this.cardHitboxes.player2].find(box => box.hero.id === caster.id);

const targetBox = [...this.cardHitboxes.player1, ...this.cardHitboxes.player2].find(box => box.hero.id === target.id);

if (casterBox && targetBox) {

this.state.overloadAnimation = {

progress: 0,

startX: casterBox.x + this.baseCardWidth \* this.scale / 2, startY: casterBox.y + this.baseCardHeight \* this.scale / 2,

endX: targetBox.x + this.baseCardWidth \* this.scale / 2, endY: targetBox.y + this.baseCardHeight \* this.scale / 2,

};

}

}

getHeroCurrentAP(hero, team) {

let currentAP = hero.ap;

if (hero.abilityId === 'mobRule') {

currentAP += team.filter(h => h.id !== hero.id && h.name.includes('Goblin') && h.hp > 0).length;

}

return currentAP;

}

calculateClashResult(p1, p2) {

let p1Damage = this.getHeroCurrentAP(p1, this.state.player1Team);

let p2Damage = this.getHeroCurrentAP(p2, this.state.player2Team);

if (this.state.gamePhase === 'bossBattle' && this.state.bossActiveAbility === 'ram') p2Damage += 5;

const p1HasAdvantage = (p1.type === 'Might' && p2.type === 'Finesse') || (p1.type === 'Finesse' && p2.type === 'Magic') || (p1.type === 'Magic' && p2.type === 'Might');

const p2HasAdvantage = (p2.type === 'Might' && p1.type === 'Finesse') || (p2.type === 'Finesse' && p1.type === 'Magic') || (p2.type === 'Magic' && p1.type === 'Might');

if (p1HasAdvantage) {

p1Damage \*= 2; p2Damage = Math.ceil(p2Damage / 2);

this.log([{ text: p1.name, color: COLORS[p1.type.toLowerCase()] }, { text: ' has the advantage!', color: COLORS.accent }]);

const p1Hitbox = this.cardHitboxes.player1.find(h => h.hero.id === p1.id);

if(p1Hitbox) this.state.abilityAnimations.push({ type: 'advantage', hitbox: p1Hitbox, color: COLORS.accent, progress: 0, speed: 0.05 });

} else if (p2HasAdvantage) {

p2Damage \*= 2; p1Damage = Math.ceil(p1Damage / 2);

this.log([{ text: p2.name, color: COLORS[p2.type.toLowerCase()] }, { text: ' has the advantage!', color: COLORS.accent }]);

const p2Hitbox = this.cardHitboxes.player2.find(h => h.hero.id === p2.id);

if(p2Hitbox) this.state.abilityAnimations.push({ type: 'advantage', hitbox: p2Hitbox, color: COLORS.accent, progress: 0, speed: 0.05 });

}

let p1Target = p1, p2Target = p2;

if (this.state.gamePhase !== 'bossBattle') {

const p1Taunter = this.state.player1Team.find(h => h.abilityId === 'tank' && h.hp > 0);

if (p1Taunter && p1Taunter.id !== p1.id) { p1Target = p1Taunter; this.log([{ text: p1Taunter.name, color: COLORS[p1Taunter.type.toLowerCase()] }, { text: ' absorbs the attack!', color: COLORS.info }]); }

const p2Taunter = this.state.player2Team.find(h => h.abilityId === 'tank' && h.hp > 0);

if (p2Taunter && p2Taunter.id !== p2.id) { p2Target = p2Taunter; this.log([{ text: p2Taunter.name, color: COLORS[p2Taunter.type.toLowerCase()] }, { text: ' absorbs the attack!', color: COLORS.info }]); }

}

let finalP1Damage = p1Damage, finalP2Damage = p2Damage;

if (this.state.gamePhase === 'bossBattle' && this.state.bossActiveAbility === 'intimidation') finalP1Damage = 0;

if (p1.abilityId === 'firstStrike' && finalP1Damage >= p2Target.hp) { finalP2Damage = 0; this.log([{ text: p1.name, color: COLORS.finesse }, { text: "'s First Strike is lethal!", color: COLORS.ability }]); }

if (p2.abilityId === 'firstStrike' && finalP2Damage >= p1Target.hp) { finalP1Damage = 0; this.log([{ text: p2.name, color: COLORS.finesse }, { text: "'s First Strike is lethal!", color: COLORS.ability }]); }

if (p1Target.abilityId === 'smokeBomb' && !p1Target.smokeBombUsed) { finalP2Damage = 0; p1Target.smokeBombUsed = true; this.log([{ text: p1Target.name, color: COLORS.finesse }, { text: "'s Smoke Bomb evades!", color: COLORS.ability }]); this.state.abilityAnimations.push({ type: 'smokeBomb', hero: p1Target, progress: 0, speed: 0.04 }); }

if (p2Target.abilityId === 'smokeBomb' && !p2Target.smokeBombUsed) { finalP1Damage = 0; p2Target.smokeBombUsed = true; this.log([{ text: p2Target.name, color: COLORS.finesse }, { text: "'s Smoke Bomb evades!", color: COLORS.ability }]); this.state.abilityAnimations.push({ type: 'smokeBomb', hero: p2Target, progress: 0, speed: 0.04 }); }

if (p1.abilityId !== 'crush' && p2Target.abilityId === 'sturdy') finalP1Damage = Math.max(0, finalP1Damage - 1);

if (p2.abilityId !== 'crush' && p1Target.abilityId === 'sturdy') finalP2Damage = Math.max(0, finalP2Damage - 1);

if (p1Target.hp <= 0) {

finalP1Damage = 0;

if (p1.hp > 0) this.log([{ text: p1.name, color: COLORS[p1.type.toLowerCase()] }, { text: "'s attack is cancelled!", color: COLORS.info }]);

}

if (p2Target.hp <= 0) {

finalP2Damage = 0;

if (p2.hp > 0) this.log([{ text: p2.name, color: COLORS[p2.type.toLowerCase()] }, { text: "'s attack is cancelled!", color: COLORS.info }]);

}

if (p1.hp <= 0) finalP1Damage = 0;

if (p2.hp <= 0) finalP2Damage = 0;

if(finalP2Damage > 0) { p1Target.hp -= finalP2Damage; this.log([{ text: p2.name, color: COLORS[p2.type.toLowerCase()] }, { text: ` deals ${finalP2Damage} damage to `, color: COLORS.text }, { text: p1Target.name, color: COLORS[p1Target.type.toLowerCase()] }]); this.checkHeroKO(p1Target); }

if(finalP1Damage > 0) { p2Target.hp -= finalP1Damage; this.log([{ text: p1.name, color: COLORS[p1.type.toLowerCase()] }, { text: ` deals ${finalP1Damage} damage to `, color: COLORS.text }, { text: p2Target.name, color: COLORS[p2Target.type.toLowerCase()] }]); this.checkHeroKO(p2Target); }

if (p1.abilityId === 'unstableConcoction' && p1HasAdvantage && p1.hp > 0) {

this.log([{ text: p1.name, color: COLORS.magic }, { text: "'s concoction explodes!", color: COLORS.ability }]);

this.state.player2Team.forEach(e => { if (e.hp > 0) { e.hp -= 3; this.log([{ text: `Explosion hits ${e.name} for 3 damage.`, color: COLORS.damage }]); this.checkHeroKO(e); } });

}

if (p2.abilityId === 'unstableConcoction' && p2HasAdvantage && p2.hp > 0) {

this.log([{ text: p2.name, color: COLORS.magic }, { text: "'s concoction explodes!", color: COLORS.ability }]);

this.state.player1Team.forEach(e => { if (e.hp > 0) { e.hp -= 3; this.log([{ text: `Explosion hits ${e.name} for 3 damage.`, color: COLORS.damage }]); this.checkHeroKO(e); } });

}

if (p1.abilityId === 'overload' && p1.hp > 0 && Math.random() < 0.5) this.handleOverload(p1, p2);

if (p2.abilityId === 'overload' && p2.hp > 0 && Math.random() < 0.5) this.handleOverload(p2, p1);

if (p1.abilityId === 'soulSiphon' && p2Target.hp <= 0 && p1.hp > 0 && !p1.soulSiphonUsed) {

const stats = HEROES.find(h => h.id === p2Target.id);

if (stats) { p1.hp += stats.hp; this.log([{ text: p1.name, color: COLORS.magic }, { text: ` siphons ${stats.hp} life!`, color: COLORS.ability }]); p1.soulSiphonUsed = true; }

}

if (p2.abilityId === 'soulSiphon' && p1Target.hp <= 0 && p2.hp > 0 && !p2.soulSiphonUsed) {

const stats = HEROES.find(h => h.id === p1Target.id);

if (stats) { p2.hp += stats.hp; this.log([{ text: p2.name, color: COLORS.magic }, { text: ` siphons ${stats.hp} life!`, color: COLORS.ability }]); p2.soulSiphonUsed = true; }

}

this.checkGameOver();

}

handleOverload(caster, originalTarget) {

let potentialTargets = [

...this.state.player1Team.filter(h => h.hp > 0 && h.id !== caster.id && h.id !== originalTarget.id),

...this.state.player2Team.filter(h => h.hp > 0 && h.id !== caster.id && h.id !== originalTarget.id)

];

if (potentialTargets.length > 0) {

const overloadTarget = potentialTargets[Math.floor(Math.random() \* potentialTargets.length)];

this.triggerOverloadAnimation(caster, overloadTarget);

overloadTarget.hp -= 4;

this.log([

{ text: caster.name, color: COLORS.magic },

{ text: "'s Overload zaps ", color: COLORS.ability },

{ text: overloadTarget.name, color: COLORS[overloadTarget.type.toLowerCase()] },

{ text: ' for 4 damage!', color: COLORS.damage }

]);

this.checkHeroKO(overloadTarget);

} else {

this.log([{ text: caster.name, color: COLORS.magic }, { text: "'s Overload fizzles!", color: COLORS.info }]);

}

}

checkHeroKO(hero) {

if (hero.hp <= 0 && !hero.deathAnimation) {

if (hero.abilityId === 'lastStand' && !hero.lastStandUsed) {

hero.hp = 1;

hero.lastStandUsed = true;

this.log([{ text: hero.name, color: COLORS.might }, { text: "'s Last Stand activates!", color: COLORS.ability }]);

this.state.abilityAnimations.push({ type: 'lastStand', hero: hero, progress: 0, speed: 0.04 });

} else {

hero.hp = 0;

hero.deathAnimation = { progress: 0 };

this.log([{ text: hero.name, color: COLORS[hero.type.toLowerCase()] }, { text: ' has been defeated!', color: COLORS.info }]);

}

}

}

initConfetti() {

this.state.confettiParticles = [];

for (let i = 0; i < 200; i++) {

this.state.confettiParticles.push({

x: Math.random() \* this.canvas.width,

y: Math.random() \* -this.canvas.height,

width: 10 \* this.scale \* (Math.random() \* 0.5 + 0.5),

height: 20 \* this.scale \* (Math.random() \* 0.5 + 0.5),

color: [COLORS.accent, COLORS.might, COLORS.finesse, COLORS.magic][Math.floor(Math.random() \* 4)],

speed: (Math.random() \* 3 + 2) \* this.scale,

angle: Math.random() \* Math.PI \* 2,

rotationSpeed: (Math.random() - 0.5) \* 0.2

});

}

}

checkGameOver() {

const p1HasHeroes = this.state.player1Team.some(h => h.hp > 0);

const p2HasHeroes = this.state.player2Team.some(h => h.hp > 0);

if (!p1HasHeroes) this.state.winner = 'p2';

else if (!p2HasHeroes) this.state.winner = 'p1';

if (this.state.winner) {

setTimeout(() => {

if (this.state.winner === 'p1' && this.state.difficulty === 'expert') {

this.state.gamePhase = 'expertVictory';

this.initConfetti();

} else {

this.state.gamePhase = 'gameOver';

}

this.log([{ text: '--- GAME OVER ---', color: COLORS.accent }]);

this.log([{ text: this.state.winner === 'p1' ? 'VICTORY!' : 'DEFEAT!', color: this.state.winner === 'p1' ? COLORS.accent : COLORS.damage }]);

}, 1000);

return true;

}

return false;

}

// -----------------------------------

// Event Handling

// -----------------------------------

handleCanvasClick(event) {

const rect = this.canvas.getBoundingClientRect();

const mouseX = event.clientX - rect.left;

const mouseY = event.clientY - rect.top;

if (this.state.gamePhase !== 'mainMenu' && this.state.gamePhase !== 'gameOver' && this.state.gamePhase !== 'expertVictory') {

const backBtn = this.buttonHitboxes.backToMenu;

if (backBtn && this.isClickInHitbox(mouseX, mouseY, backBtn)) { this.resetGame(); return; }

}

switch (this.state.gamePhase) {

case 'mainMenu': this.handleMainMenuClick(mouseX, mouseY); break;

case 'difficultySelection': this.handleDifficultySelectionClick(mouseX, mouseY); break;

case 'heroSelection': this.handleHeroSelectionClick(mouseX, mouseY); break;

case 'practiceSelection': this.handlePracticeSelectionClick(mouseX, mouseY); break;

case 'heroViewer': this.handleHeroViewerClick(mouseX, mouseY); break;

case 'playing': case 'bossBattle': this.handlePlayingClick(mouseX, mouseY); break;

case 'gameOver': case 'expertVictory': this.handleGameOverClick(mouseX, mouseY); break;

}

}

handleMouseMove(event) {

const rect = this.canvas.getBoundingClientRect();

this.mousePos.x = event.clientX - rect.left;

this.mousePos.y = event.clientY - rect.top;

if (!['heroSelection', 'practiceSelection', 'playing', 'bossBattle'].includes(this.state.gamePhase)) {

this.state.hoveredHero = null;

return;

}

let foundHero = null;

const checkHitboxes = (hitboxes, team) => {

for (const hitbox of hitboxes) {

if (this.isClickInHitbox(this.mousePos.x, this.mousePos.y, hitbox)) {

foundHero = { ...hitbox, team };

break;

}

}

};

if (['heroSelection', 'practiceSelection'].includes(this.state.gamePhase)) {

checkHitboxes(this.cardHitboxes.selection, []);

} else if (['playing', 'bossBattle'].includes(this.state.gamePhase)) {

checkHitboxes(this.cardHitboxes.player1, this.state.player1Team);

if (!foundHero) checkHitboxes(this.cardHitboxes.player2, this.state.player2Team);

}

this.state.hoveredHero = foundHero;

}

handleMainMenuClick(mouseX, mouseY) {

if (this.buttonHitboxes['3v3clash']?.width && this.isClickInHitbox(mouseX, mouseY, this.buttonHitboxes['3v3clash'])) { this.state.teamSize = 3; this.state.gamePhase = 'difficultySelection'; }

if (this.buttonHitboxes['5v5clash']?.width && this.isClickInHitbox(mouseX, mouseY, this.buttonHitboxes['5v5clash'])) { this.state.teamSize = 5; this.state.gamePhase = 'difficultySelection'; }

if (this.buttonHitboxes.bossbattle?.width && this.isClickInHitbox(mouseX, mouseY, this.buttonHitboxes.bossbattle)) { this.state.difficulty = 'boss'; this.state.teamSize = 5; this.state.gamePhase = 'heroSelection'; }

if (this.buttonHitboxes.practice?.width && this.isClickInHitbox(mouseX, mouseY, this.buttonHitboxes.practice)) { this.state.teamSize = 3; this.state.gamePhase = 'practiceSelection'; }

if (this.buttonHitboxes.heroviewer?.width && this.isClickInHitbox(mouseX, mouseY, this.buttonHitboxes.heroviewer)) { this.state.gamePhase = 'heroViewer'; }

}

handleDifficultySelectionClick(mouseX, mouseY) {

for (const level of ['easy', 'normal', 'expert']) {

const btn = this.buttonHitboxes[level];

if (btn && this.isClickInHitbox(mouseX, mouseY, btn)) {

this.state.difficulty = level;

this.state.gamePhase = 'heroSelection';

return;

}

}

}

handleHeroSelectionClick(mouseX, mouseY) {

for (const hitbox of this.cardHitboxes.selection) {

if (this.isClickInHitbox(mouseX, mouseY, hitbox)) {

const hero = hitbox.hero;

const index = this.state.player1Team.findIndex(h => h.id === hero.id);

if (index > -1) this.state.player1Team.splice(index, 1);

else if (this.state.player1Team.length < this.state.teamSize) this.state.player1Team.push(JSON.parse(JSON.stringify(hero)));

return;

}

}

const startBtn = this.buttonHitboxes.startGame;

if (startBtn && this.isClickInHitbox(mouseX, mouseY, startBtn)) {

if (this.state.difficulty === 'boss') this.startBossBattle();

else this.startGame();

}

}

handlePracticeSelectionClick(mouseX, mouseY) {

for (const hitbox of this.cardHitboxes.selection) {

if (this.isClickInHitbox(mouseX, mouseY, hitbox)) {

const hero = hitbox.hero;

const team = this.state.practiceSelectionState === 'player' ? this.state.player1Team : this.state.player2Team;

const index = team.findIndex(h => h.id === hero.id);

if (index > -1) team.splice(index, 1);

else if (team.length < this.state.teamSize) team.push(JSON.parse(JSON.stringify(hero)));

return;

}

}

const confirmBtn = this.buttonHitboxes.confirmTeam;

if (confirmBtn && this.isClickInHitbox(mouseX, mouseY, confirmBtn)) {

if (this.state.practiceSelectionState === 'player') this.state.practiceSelectionState = 'opponent';

else this.startGame(true);

}

}

handleHeroViewerClick(mouseX, mouseY) {

if (this.buttonHitboxes.previous?.width && this.isClickInHitbox(mouseX, mouseY, this.buttonHitboxes.previous)) { this.state.heroViewerIndex = (this.state.heroViewerIndex - 1 + HEROES.length) % HEROES.length; }

if (this.buttonHitboxes.next?.width && this.isClickInHitbox(mouseX, mouseY, this.buttonHitboxes.next)) { this.state.heroViewerIndex = (this.state.heroViewerIndex + 1) % HEROES.length; }

if (this.buttonHitboxes.back?.width && this.isClickInHitbox(mouseX, mouseY, this.buttonHitboxes.back)) { this.state.gamePhase = 'mainMenu'; }

}

handlePlayingClick(mouseX, mouseY) {

if (this.state.isClashing) return;

const clashBtn = this.buttonHitboxes.clash;

if (clashBtn && this.isClickInHitbox(mouseX, mouseY, clashBtn)) { this.resolveClash(); return; }

for (const hitbox of this.cardHitboxes.player1) {

if (this.isClickInHitbox(mouseX, mouseY, hitbox) && hitbox.hero.hp > 0) {

this.state.player1Selection = hitbox.hero;

this.log([{ text: 'You selected ', color: COLORS.info }, { text: hitbox.hero.name, color: COLORS[hitbox.hero.type.toLowerCase()] }]);

return;

}

}

}

handleGameOverClick(mouseX, mouseY) {

const btn = this.buttonHitboxes.playAgain;

if (btn && this.isClickInHitbox(mouseX, mouseY, btn)) this.resetGame();

}

isClickInHitbox(mouseX, mouseY, hitbox) {

return mouseX >= hitbox.x && mouseX <= hitbox.x + hitbox.width &&

mouseY >= hitbox.y && mouseY <= hitbox.y + hitbox.height;

}

resizeCanvas() {

this.canvas.width = window.innerWidth;

this.canvas.height = window.innerHeight;

this.scale = Math.min(this.canvas.width / this.baseWidth, this.canvas.height / this.baseHeight);

}

}

// ===================================

// ▶️ 3. INITIALIZE

// ===================================

window.onload = () => {

const game = new Game('gameCanvas');

};

</script>

</body>

</html>