Sometimes, Aliens Are Not Your Friends: A WebGL Game Development Project

Aidan Mark, Brian Hu, Matt Cassidy

Introduction:

"Sometimes, Aliens Are Not Your Friends" is an exciting and challenging WebGL game that combines elements of classic space invaders and asteroid games. In this game, players navigate through space, battling waves of UFO enemies and avoiding asteroids. The project presents numerous implementation challenges, notably in developing smooth controls, realistic physics, and engaging enemy AI. The integration of advanced graphics, such as dynamic lighting and texture mapping, also posed significant complexity. This project serves as a testament to the intricacies of modern web-based game development and the application of theoretical concepts in a practical, interactive environment.

Methods:

Summary of Functionality
The game's core functionalities are implemented using JavaScript and WebGL. The player controls a spaceship, dodging obstacles and firing at enemies. Key features include:

Dynamic Enemy Encounters: Enemies appear in waves, each with increasing difficulty.
Player and Enemy Mechanics: Includes health systems, shooting mechanics, and collision detection.
Environmental Challenges: Asteroids and other obstacles create a dynamic, interactive space environment.
Lighting and Effects: Utilizes point lights for dynamic lighting effects on the spaceship, weapons, and enemies.
Scene Management: The game dynamically loads a scene from a JSON file (*scene.json*), creating an immersive environment for the player.
Skybox Implementation: A skybox is used to create an expansive background, enhancing the game's immersive experience and space feel. (Not quite implemented)

Implementation of Shaders

The game employs two primary shaders written in GLSL: a vertex shader (vertShaderSample) and a fragment shader (fragShaderSample). These shaders are pivotal in rendering objects with realistic lighting and textures.

Vertex Shader

Functionality: Transforms vertex positions from model space to clip space, calculates transformed normal vectors for lighting, and passes texture coordinates to the fragment shader.

Key Components:

- Inputs for vertex position, normal, and texture coordinates.
- Uniforms for transformation matrices and camera position.
- Outputs for texture coordinates, fragment position, normal vector, and camera position.

Fragment Shader

Functionality: Implements the Phong lighting model for ambient, diffuse, and specular lighting, and manages texture mapping and transparency.

Phong Lighting Model:

- Ambient Lighting: Simple, uniform lighting across the surface.
- Diffuse Lighting: Reflects the light based on the angle of the light source, using the dot product of the normal and light direction vectors.
- Specular Lighting: Simulates the shiny spots of light, particularly where the light source reflects directly into the camera.
- Attenuation: Reduces light intensity over distance, using linear and quadratic factors.

- Texture Mapping: Applies textures to surfaces when available, enhancing visual details.
- Transparency Control: Manages the transparency of objects using an alpha value.

Link to Theory:

The game's development is closely tied to the theoretical concepts taught in class:

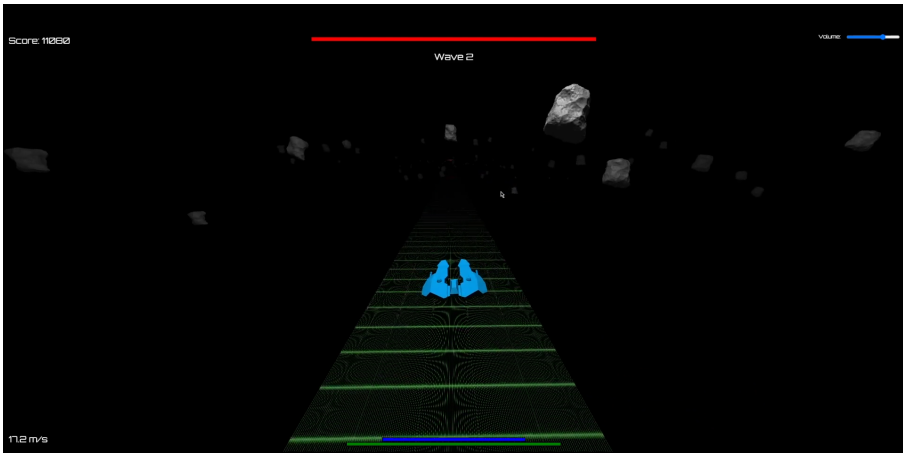
Modeling and Viewing: Utilizes WebGL's 3D rendering capabilities for creating and manipulating game objects in a virtual space. The game employs various transformations, such as translations, rotations, and scaling, to simulate realistic movement.
Shading and Light Models: Implements Phong shading model for realistic rendering of objects with attenuation. The game uses point light sources to create dynamic shadows and highlights, enhancing the visual quality.
Texture Mapping: Applies textures to game objects for added realism. Special textures, like bump maps, are used for asteroids to give them a more realistic, rugged look.
Visibility and Transparency: Manages the rendering order and uses alpha values to handle object visibility and transparency, crucial for effects like spaceship-asteroid impacts and laser beam collisions.

Implementation Details

Scene Composition and Player Interaction

1. Dynamic Scene Elements

- Player Spaceship and Light Source: The spaceship, accompanied by a soft light source above, creates a sense of depth and dimension. This light source adds to the visual appeal and aids in spatial orientation.
- Camera Dynamics: The camera moves in unison with the player, maintaining a consistent viewpoint. This movement is critical for ensuring a seamless and immersive player experience.



2. Asteroid Field Implementation

- Spawn Mechanics: Asteroids are spawned within a randomized range, introducing unpredictability and variety to the gameplay.
- Repositioning Algorithm: A key optimization technique involves repositioning asteroids in front of the player, thereby reducing the computational overhead of generating new objects.

```
spawnAsteroidField() {
        const numAsteroids = 1;

        for (let i = 0; i < numAsteroids; i++) {
            const zPos = 200 + (i * (400 / numAsteroids));
            this.spawnAsteroid(0, 0, zPos);
        }
    }

    spawnAsteroid(zPos) {
        // Define the range for spawning asteroids
        const xMin = -15;
        const xMax = 20;
        const yMin = -6;
        const yMax = 7;
        const zOffsetAhead = 400;

        // Randomly choose an asteroid model
        const asteroidModels = ["asteroid3.obj", "asteroid4.obj", "asteroid5.obj",
                                "asteroid6.obj", "asteroid7.obj", "asteroid8.obj",
"asteroid9.obj", "asteroid10.obj"];
        const asteroidModel = asteroidModels[Math.floor(Math.random() *
asteroidModels.length)];

        // Random size variation
        const sizeFactor = 0.3 + Math.random() * 1.7; // Random scale

        // Define the asteroid configuration
        const asteroidConfig = {
            name: `Asteroid-${Date.now()}`,
            model: asteroidModel,
            type: "mesh",
            material: {
                diffuse: [0.1, 0.1, 0.1],
                ambient: [0.05, 0.05, 0.05],
                specular: [0.1, 0.1, 0.1],
                n: 10,
                alpha: 1,
                shaderType: 3
            },
            position: vec3.fromValues(
                xPos,
                yPos,
                zPos
            ),
            scale: vec3.fromValues(sizeFactor, sizeFactor, sizeFactor),
            diffuseTexture: "DefaultMaterial_albedo.jpg",
            normalTexture: "DefaultMaterial_normal.png"
```

```
        };

        // Add the asteroid to the game state
        const newAsteroid = spawnObject(asteroidConfig, this.state);
        this.spawnedObjects.push(newAsteroid);
        this.asteroidPool.push(asteroidConfig);

        if (newAsteroid && newAsteroid.model) { // Check if the model property exists ??
Doesn't work
             // Add to the asteroid pool
            console.log(this.asteroidPool[0]);
        }
    }

    repositionAsteroids() {
        const xMin = -80;
        const xMax = 80;
        const yMin = -10;
        const yMax = 10;
        const zMin = 400;
        const zMax = 800;

        this.state.objects.forEach((object) => {
            if (object.name.startsWith('Asteroid-')) {
                // Check if the asteroid is behind the spaceship
                if (object.model.position[2] < this.spaceship.model.position[2] - 17.5) {
                    const newZ = this.spaceship.model.position[2] + Math.random() * (zMax -
zMin) + zMin;

                    // Randomly determine new X and Y positions within a specified range
                    const newX = Math.random() * (xMax - xMin) + xMin - 20;
                    const newY = Math.random() * (yMax - yMin) + yMin + 5;

                    // Reposition the asteroid
                    vec3.set(object.model.position, newX, newY, newZ);
                    //console.log("Asteroid repositioned: ", object.model.position);
                }
            }
        });
    }
```

3. Infinite Scrolling Planes

- Plane Movement Logic: Two planes are employed to simulate an endless space environment. One plane repositions in front of the other as the player advances, creating an illusion of continuous movement.

```
updatePlane() {
    let planeLength = 1060;
    let halfPlaneLength = planeLength / 2;
    let repositionThreshold = -100; // Distance behind the camera to trigger reposition

    // Check if the end of tempPlane is behind the camera
    if (this.plane.model.position[2] + halfPlaneLength < this.state.camera.position[2] + repositionThreshold) {
        this.plane.model.position[2] = this.plane2.model.position[2] + planeLength - halfPlaneLength;
    }

    // Check if the end of tempPlane2 is behind the camera
    if (this.plane2.model.position[2] + halfPlaneLength < this.state.camera.position[2] + repositionThreshold) {
        this.plane2.model.position[2] = this.plane.model.position[2] + planeLength - halfPlaneLength;
    }
}
```

Enemy Dynamics and Combat

1. Enemy Behavior

- Spawn and Approach Mechanics: The enemy UFO spawns at a distance and appears to approach the player. This simulated movement enhances the game's challenge and engagement.
- Update Movement Algorithm: The updateEnemyMovement function is invoked when the enemy is within a specific range, simulating a DVD logo-like random movement within a defined box, adding to the gameplay's unpredictability.

```
updateEnemyMovement(deltaTime) {
    // Define the range of movement along the x-axis and y-axis
    const minX = -40;
    const maxX = 40;
    const minY = -7;
    const maxY = 15;
    const minZ = 12;
    const maxZ = 35;

    // Movement speed along x-axis and y-axis
    const xSpeed = 10 * this.enemySpeed;
    const ySpeed = 3 * this.enemySpeed;
    const zSpeed = 8 * this.enemySpeed;

    // Check if the enemy is at or beyond the x-axis boundaries
    if (this.enemy1.model.position[0] >= maxX || this.enemy1.model.position[0] <= minX) {
        this.enemy1.xMovementFactor *= -1;
    }

    // Check if the enemy is at or beyond the y-axis boundaries
    if (this.enemy1.model.position[1] >= maxY || this.enemy1.model.position[1] <= minY) {
        this.enemy1.yMovementFactor *= -1;
    }

    // Check if the enemy is at or beyond the z-axis boundaries
    if (this.enemy1.model.position[2] >= maxZ || this.enemy1.model.position[2] <= minZ) {
        this.enemy1.zMovementFactor *= -1;
    }

    this.enemy1Velocity = {
        x: this.enemy1.xMovementFactor * xSpeed * deltaTime,
        y: this.enemy1.yMovementFactor * ySpeed * deltaTime,
        z: this.enemy1.zMovementFactor * zSpeed * deltaTime
    };

    // Update position
    this.enemy1.model.position[0] += this.enemy1.xMovementFactor * xSpeed * deltaTime;
    this.enemy1.model.position[1] += this.enemy1.yMovementFactor * ySpeed * deltaTime;
    this.enemy1.model.position[2] += this.enemy1.zMovementFactor * zSpeed * deltaTime;

    // Enemy AI unstuck mechanic
    let enemyStuckTimer = 0;
    const enemyStuckThreshold = 5; // Time limit until enemy gets forcefully repositioned
    const boundaryBuffer = 1; // buffer zone if their close but not touching the boundary. May need adjusting.
    const isStuck =
    (this.enemy1.model.position[0] >= maxX - boundaryBuffer|| this.enemy1.model.position[0] <= minX + boundaryBuffer) ||
    (this.enemy1.model.position[1] >= maxY - boundaryBuffer|| this.enemy1.model.position[1] <= minY) + boundaryBuffer||
    (this.enemy1.model.position[2] >= maxZ - boundaryBuffer|| this.enemy1.model.position[2] <= minZ + boundaryBuffer);


    if (isStuck) {···
    } else {
        // Reset the timer when the enemy is not stuck
        enemyStuckTimer = 0;
    }
```

2. Projectile System

- Fire Rate and Speed: Both the player and the enemy have specified fire rates and projectile speeds. Balancing these parameters is crucial for fair gameplay.
- Enemy Targeting System: The enemy's targeting algorithm includes a random variation, ensuring the player is not overwhelmed, thus balancing difficulty and fairness.

```
enemyAttack() {
    if (this.enemy1Killed || this.gameOver) {
        return;
    }

    if (!this.enemyVolleyInProgress) {
        this.enemyVolleyInProgress = true;
        let shotsFired = 0;

        const volleyInterval = setInterval(() => {
            if (shotsFired < this.enemyVolleyCount) {
                // Spawn point
                let spawnPosition = vec3.clone(this.enemy1.model.position);
                spawnPosition[1] -= 1;
                spawnPosition[2] += 45;

                // Target point
                let targetPosition = vec3.clone(this.spaceship.model.position);
                targetPosition[0] -= 0.25 + (Math.random() * 6 - 3);
                targetPosition[1] += (Math.random() * 6 - 3);
                targetPosition[2] += 15;


                // Calculate the direction
                let direction = vec3.create();
                vec3.subtract(direction, targetPosition, spawnPosition);
                vec3.normalize(direction, direction);

                // Spawn the projectile (cube) from the enemy
                this.spawnEnemyCube(this.state, spawnPosition, direction);
                shotsFired++;
            } else {
                clearInterval(volleyInterval);
                this.enemyVolleyInProgress = false;
                this.enemyVolleyTimer = this.enemyVolleyCooldown;
            }
        }, this.enemyShotInterval);
    }
}
```

Advanced Player Mechanics

1. Aim Assist and Shooting Mechanics

- Aim Assist Functionality: The game employs a glmatrix lerp function for aim assist, subtly guiding player shots for a more rewarding experience without undermining the skill component.

```
// Update the firing direction based on the current mouse position
updateFiringDirection() {
    let mouseWorldPosition = this.screenToWorld(this.mousePosition);

    if (this.isFirstPersonCamera) {
        mouseWorldPosition = this.rotatePositionBySpaceshipOrientation(mouseWorldPosition);
    }

    // Target point with adjusted offset for the enemy's actual position
    let targetPosition = vec3.clone(this.enemy1.model.position); // TODO: add multiple enemies, make this change targeting somehow
    targetPosition[2] += 40;

    // Iterate over all cubes and update their direction
    this.state.objects.forEach(object => {
        if (object.name.startsWith('Cube-') && object.model.position[2] < this.spaceship.model.position[2] + this.shootingCurve) {
            let direction = vec3.create();

            // Calculate direction towards the enemy
            let enemyDirection = vec3.create();
            vec3.subtract(enemyDirection, targetPosition, this.spaceship.model.position);
            vec3.normalize(enemyDirection, enemyDirection);

            // Calculate player's current firing direction
            let playerDirection = vec3.create();
            vec3.subtract(playerDirection, vec3.fromValues(-mouseWorldPosition[0] - 5, mouseWorldPosition[1] - 5, mouseWorldPosition[2] + 25), this.spaceship.model.position);
            vec3.normalize(playerDirection, playerDirection);

            // Blend the directions based on aim assist factor
            let aimAssistFactor = 0;
            if (!this.intermission) {
                aimAssistFactor = 0.7;
            }
            vec3.lerp(direction, playerDirection, enemyDirection, aimAssistFactor);

            // vec3.subtract(direction, vec3.fromValues(-mouseWorldPosition[0] - 5, mouseWorldPosition[1] - 5, mouseWorldPosition[2] + 25), this.spaceship.model.position);
            // vec3.normalize(direction, direction);
            object.direction = direction;
        }
    });
}
```

- Mouse-based Trajectory Calculation: The player's shot trajectory is determined using the mouse position, with a Z-axis offset to bridge the 2D-3D space conversion. This implementation encourages players to focus on shot impact rather than mouse location.

2. Spaceship Movement and Control

- Velocity and Boundary Interaction: The spaceship's movement incorporates a velocity component, giving it a fluid, slidey feel. When hitting the play area's boundaries, it elastically bounces back, a mechanic that enhances the game's realism and player immersion.
- Roll Effect on Movement: Rolling the spaceship adds to the visual aesthetics and affects its movement dynamics, particularly in side-to-side maneuvers.

```
updateSpaceshipPosition(deltaTime) {
        let moveDirection = vec3.create();

        // Movement keys
        const movementIncrement = 0.003;
        if (this.isFirstPersonCamera) { // For first person mode
            // Calculate the direction vectors based on the ship's up
vector
            let forward = vec3.fromValues(0, 0, 1);
            let right = vec3.create();
            vec3.cross(right, this.shipUp, forward);
            vec3.normalize(right, right);

            if (this.keyPressed.w) {
                vec3.add(moveDirection, moveDirection, this.shipUp);
            }
            if (this.keyPressed.s) {
                vec3.subtract(moveDirection, moveDirection, this.shipUp);
            }
            if (this.keyPressed.a) {
                vec3.add(moveDirection, moveDirection, right);
                this.targetRollAngle = this.maxRollAngle;
            }
            if (this.keyPressed.d) {
                vec3.subtract(moveDirection, moveDirection, right);
                this.targetRollAngle = -this.maxRollAngle;
            }
```

```
            // Apply the movement
            vec3.scale(moveDirection, moveDirection, movementIncrement);
            this.spaceshipVelocity.x += moveDirection[0];
            this.spaceshipVelocity.y += moveDirection[1];
        } else {
            if (this.keyPressed.w) {
                this.spaceshipVelocity.y += movementIncrement;
            }
            if (this.keyPressed.s) {
                this.spaceshipVelocity.y -= movementIncrement;
            }
            if (this.keyPressed.a) {
                this.spaceshipVelocity.x += movementIncrement;
                this.targetRollAngle = this.maxRollAngle;
            }
            if (this.keyPressed.d) {
                this.spaceshipVelocity.x -= movementIncrement;
                this.targetRollAngle = -this.maxRollAngle;
            }
        }
        if (this.keyPressed.q) {
            this.targetRollAngle = this.maxRollAngle;
        }
        if (this.keyPressed.e) {
            this.targetRollAngle = -this.maxRollAngle;
        }

        // Clamp the currentRollAngle to prevent excessive rolling
        this.currentRollAngle = Math.max(-this.maxRollAngle,
Math.min(this.maxRollAngle, this.currentRollAngle));
        if (this.isFirstPersonCamera) {
            this.rollAngle = Math.max(-0.005, Math.min(0.005,
this.rollAngle));
        } else {
            this.rollAngle = Math.max(-0.1, Math.min(0.1,
this.rollAngle));
        }

        // Gradually adjust the roll angle towards the target roll angle
```

```
        if (this.isFirstPersonCamera && (this.keyPressed.a ||
this.keyPressed.d)) {  // Slower roll speed in first person A and D keys
            const firstPersonRollSpeed = this.rollSpeed / 9;
            if (this.rollAngle < this.targetRollAngle) {
                this.rollAngle = Math.min(this.rollAngle +
firstPersonRollSpeed, this.targetRollAngle);
            } else if (this.rollAngle > this.targetRollAngle) {
                this.rollAngle = Math.max(this.rollAngle -
firstPersonRollSpeed, this.targetRollAngle);
            }
        } else {
            // Original logic for outside first-person mode
            if (this.rollAngle < this.targetRollAngle) {
                this.rollAngle = Math.min(this.rollAngle + this.rollSpeed,
this.targetRollAngle);
            } else if (this.rollAngle > this.targetRollAngle) {
                this.rollAngle = Math.max(this.rollAngle - this.rollSpeed,
this.targetRollAngle);
            }
        }

        // Gradually return the roll angle to zero when no left/right
movement
        if (!this.keyPressed.a && !this.keyPressed.d) {
            if (this.rollAngle > 0) {
                this.rollAngle = Math.max(this.rollAngle -
this.rollReturnSpeed, 0);
            } else if (this.rollAngle < 0) {
                this.rollAngle = Math.min(this.rollAngle +
this.rollReturnSpeed, 0);
            }
            this.targetRollAngle = 0; // Reset target roll angle
        }

        // Apply rotation
        this.spaceship.rotate('z', this.rollAngle);

        // Update the total rotation of the spaceship
        const scaleFactor = 180 / 1.5; // Calibrated to match the actual
rotation
```

```
        this.spaceshipTotalRotation += (this.rollAngle * deltaTime) *
scaleFactor;
        this.spaceshipTotalRotation = (this.spaceshipTotalRotation + 2 *
Math.PI) % (2 * Math.PI); // Normalize the total rotation

        // Calculate the up vector components based on the scaled rotation
        let upVectorX = Math.cos(this.spaceshipTotalRotation - Math.PI /
2);
        let upVectorY = -Math.sin(this.spaceshipTotalRotation - Math.PI /
2);

        // Normalize the up vector
        const length = Math.sqrt(upVectorX * upVectorX + upVectorY *
upVectorY);
        upVectorX /= length;
        upVectorY /= length;

        // Set the up vector for the spaceship
        this.shipUp = vec3.fromValues(upVectorX, upVectorY, 0);

        // Limit the spaceship's velocity
        const spaceshipVelocityMaximum = 0.1;
        this.spaceshipVelocity.x = Math.max(-spaceshipVelocityMaximum,
Math.min(spaceshipVelocityMaximum, this.spaceshipVelocity.x));
        this.spaceshipVelocity.y = Math.max(-spaceshipVelocityMaximum,
Math.min(spaceshipVelocityMaximum, this.spaceshipVelocity.y));

        // Apply deceleration when movement keys are not pressed
        const decelerationFactor = 0.99; // More pronounced deceleration
        if (!this.keyPressed.w && !this.keyPressed.s) {
            this.spaceshipVelocity.y *= decelerationFactor;
        }
        if (!this.keyPressed.a && !this.keyPressed.d) {
            this.spaceshipVelocity.x *= decelerationFactor;
        }

        // Define clamping bounds with a slight overstep allowance
        let minX = -7, maxX = 7, minY = -3, maxY = 5;
        if (this.isFirstPersonCamera) { // Extra room for first person
            minX -= 12;
```

```
            maxX += 12;
            minY -= 5;
            maxY += 5;
        }

        // Update spaceship's position based on its velocity
        let newX = this.spaceship.model.position[0] +
this.spaceshipVelocity.x;
        let newY = this.spaceship.model.position[1] +
this.spaceshipVelocity.y;
        vec3.scale(moveDirection, moveDirection, this.spaceshipSpeed);

        // Check if spaceship is out of bounds and adjust position
accordingly
        let outOfBoundsX = newX < minX || newX > maxX;
        let outOfBoundsY = newY < minY || newY > maxY;

        const maxReturnSpeed = 0.06; // Maximum speed for returning to
bounds
        const overstepMargin = 0.2; // Allow some overstepping
        const returnAcceleration = 0.002; // Acceleration when returning
to bounds

        if (outOfBoundsX) {
            let boundaryX = newX < minX ? minX : maxX;
            let distanceX = Math.abs(newX - boundaryX) - overstepMargin;
            let directionX = newX < minX ? 1 : -1;
            this.spaceshipVelocity.x += directionX * Math.max(0,
distanceX) * returnAcceleration;
            this.spaceshipVelocity.x = Math.min(maxReturnSpeed,
Math.max(-maxReturnSpeed, this.spaceshipVelocity.x));
            newX += this.spaceshipVelocity.x;
        } else {
            this.spaceshipVelocity.x *= decelerationFactor;
        }

        if (outOfBoundsY) {
            let boundaryY = newY < minY ? minY : maxY;
            let distanceY = Math.abs(newY - boundaryY) - overstepMargin;
            let directionY = newY < minY ? 1 : -1;
```

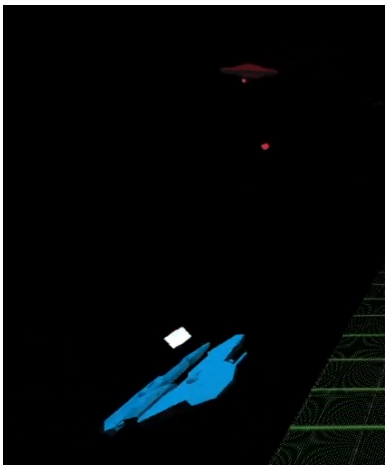
```
            this.spaceshipVelocity.y += directionY * Math.max(0,
distanceY) * returnAcceleration;
            this.spaceshipVelocity.y = Math.min(maxReturnSpeed,
Math.max(-maxReturnSpeed, this.spaceshipVelocity.y));
            newY += this.spaceshipVelocity.y;
        } else {
            this.spaceshipVelocity.y *= decelerationFactor;
        }

        // Allow slight overstep before clamping
        this.spaceship.model.position[0] = Math.max(minX - overstepMargin,
Math.min(maxX + overstepMargin, newX));
        this.spaceship.model.position[1] = Math.max(minY - overstepMargin,
Math.min(maxY + overstepMargin, newY));

        // Translate the position of the spaceship
        this.spaceship.translate(vec3.fromValues(
            newX - this.spaceship.model.position[0],
            newY - this.spaceship.model.position[1],
            0
        ));
    }
```

3. First-Person Perspective Complexity

- View Toggle and Roll Interaction: Pressing the 'v' key toggles a first-person perspective, introducing complexities in movement and camera alignment, especially when the ship rolls.
- Camera Alignment: The camera's dynamic offset ensures it feels locked onto the ship, even when the ship rolls or inverts.
- Press the '~' key for Freecam mode (debugging feature)

```
updateFirstPersonCamera() {
    // Constant z-offset
    const zOffset = -3;

    // Calculate the dynamic offset based on the ship's up vector
    let dynamicOffset = vec3.create();
    vec3.scale(dynamicOffset, this.shipUp, 1); // Scale the ship's up vector by 1 unit
    dynamicOffset[2] = zOffset; // Set the z component to the constant offset

    // Add the dynamic offset to the spaceship's position for the camera's world position
    let cameraWorldPosition = vec3.create();
    vec3.add(cameraWorldPosition, this.spaceship.model.position, dynamicOffset);
    this.state.camera.position = cameraWorldPosition;

    this.state.camera.front = vec3.fromValues(0, 0, 1);
    this.state.camera.up = vec3.clone(this.shipUp);

    // Update the camera view matrix
    mat4.lookAt(
        this.state.camera.viewMatrix,
        this.state.camera.position,
        vec3.add(vec3.create(), this.state.camera.position, this.state.camera.front),
        this.state.camera.up
    );
}
```

Collision Mechanics and Effects



1. Collision Detection and Response

- Collision Effects: Collisions with enemies and asteroids trigger distinct visual effects, such as bouncing off and color changes to white, enhancing visual feedback.
- Player-Asteroid Collision Mechanics: When the player's ship collides with an asteroid, it temporarily becomes transparent, and its weapons go offline, adding a strategic element to navigation and combat.

```
processCollisionEffect(object, deltaTime, duration, isEnemyCube = false) {
    object.collisionTime += deltaTime;
    let progress = object.collisionTime / duration;
    let scale = (progress < 0.4) ? 1.07 : 0.97;

    // Stop the cube
    object.direction = [0, 0, 0];

    // Customize effect based on cube type
    if (isEnemyCube) {
        // Specific effects for enemy cube collision
        object.scale(vec3.fromValues(scale, scale, scale));
        object.model.position[2] += this.forwardDistance - 0.5;
        object.material.ambient = [progress, progress, progress];
    } else {
        // Effects for player cube collision
        object.material.ambient = [1, 1, 1];
        object.model.position[2] += this.forwardDistance - 0.02;
        object.scale(vec3.fromValues(scale, scale, scale));
    }

    if (object.collisionTime >= duration) {
        removeObjectFromState(object, state);
    }
}
```

2. Collider Shape and Limitations

- Collider Shape Optimization: The current rectangular collider could be optimized with a spherical collider for more accurate and realistic collision detection, especially given the game's 3D nature.

Performance and Configuration

1. Asteroid Field Optimization

- Pre-Spawn Strategy: A Python script is used to pre-spawn asteroids before the game for better performance, demonstrating an innovative approach to resource management.
- Scene Configuration Flexibility: Players can adjust the number of asteroids in the scene.json file, allowing the game to be tailored to different hardware capabilities.

User Interface and Additional Features

1. Comprehensive User Interface

- Gameplay Metrics Display: The game features a sleek UI that tracks crucial metrics like score, speed, player health, ammo, enemy health, and wave number, providing players with essential information at a glance.
- Pause and Restart Functions: The ability to pause and restart the game adds to the user-friendly experience, allowing players to take breaks or retry challenges.

2. Audio Integration

- Sound Effects: The game integrates sound effects to represent various in-game actions and events, contributing to an immersive auditory experience.

```
<audio id="startup" src="sounds/welcome.wav" preload="auto"></audio>
<audio id="hitmarker" src="sounds/hitmarker.wav" preload="auto"></audio>
<audio id="playerHit" src="sounds/player_hit.wav" preload="auto"></audio>
<audio id="playerKilled" src="sounds/player_killed.wav" preload="auto"></audio>
<audio id="enemyKilled" src="sounds/enemy_killed.wav" preload="auto"></audio>
<audio id="playerShot" src="sounds/player_shot.wav" preload="auto"></audio>
<audio id="enemyShot" src="sounds/enemy_shot.wav" preload="auto"></audio>
<audio id="newWave" src="sounds/new_wave.wav" preload="auto"></audio>
<audio id="click" src="sounds/click.wav" preload="auto"></audio>
<audio id="impact" src="sounds/impact.wav" preload="auto"></audio>
<audio id="asteroidHit" src="sounds/asteroid_hit.wav" preload="auto"></audio>
```

Future Development Considerations

1. Planned Features and Limitations

- Skybox/Spacebox: Can’t have a space game with no stars.
- Sphere Collisions: The planned implementation of sphere collisions for more accurate interactions.
- Projectile Lights: Adding lights to projectiles for enhanced visual feedback.
- Expanded Enemy and Multiplayer Mechanics: Introducing multiple enemies and multiplayer capabilities could significantly expand the game's scope and appeal.

Analysis and Discussion

The project "Sometimes, Aliens Are Not Your Friends" has been a super fun exercise in applying theoretical knowledge to practical game development. Through this project, we learned the importance of efficient code structuring and the challenges of optimizing 3D graphics for web browsers. The most successful aspects were the implementation of dynamic lighting and realistic looking physics, which significantly enhanced the game's immersive experience. However, challenges were faced in optimizing collision detection and maintaining smooth gameplay across different machines. Future improvements could include more sophisticated AI for enemies, a wider variety of obstacles, and perhaps a multiplayer mode. This project has laid a strong foundation for further exploration and innovation in game development.

Colors(Brian):

```
const fragShaderSample =
    `#version 300 es
    #define MAX_LIGHTS 100
    precision highp float;

    struct PointLight {
        vec3 position;
        vec3 colour;
        float strength;
        float linear;
        float quadratic;
    };

    in vec2 oUV;
    in vec3 oNormal;
    in vec3 oFragPosition;
    in vec3 oCameraPosition;

    uniform PointLight pointLights[MAX_LIGHTS];
    uniform int numLights;
    uniform vec3 diffuseVal;
    uniform vec3 ambientVal;
    uniform vec3 specularVal;
    uniform float nVal;
    uniform float uAlpha;
    uniform int samplerExists;
    uniform sampler2D uTexture;

    out vec4 fragColor;

    void main() {
        vec3 normal = normalize(oNormal);
        vec3 viewDir = normalize(oCameraPosition - oFragPosition);
        vec3 result = ambientVal; // Start with ambient lighting

        vec3 textureColor = vec3(1.0); // Default to white if no texture
        if (samplerExists == 1) {
            textureColor = texture(uTexture, oUV).rgb;
        }

        for (int i = 0; i < numLights; i++) {
            PointLight light = pointLights[i];
            vec3 lightDir = normalize(light.position - oFragPosition);

            // Diffuse shading
            float diff = max(dot(normal, lightDir), 0.0);
            vec3 diffuse = light.colour * light.strength * diff * (samplerExists == 1 ? textureColor : diffuseVal);

            // Specular shading
            vec3 reflectDir = reflect(-lightDir, normal);
            float spec = pow(max(dot(viewDir, reflectDir), 0.0), nVal);
            vec3 specular = light.colour * light.strength * spec * specularVal;

            // Attenuation
            float distance = length(light.position - oFragPosition);
            float attenuation = 1.0 / (1.0 + light.linear * distance + light.quadratic * (distance * distance));

            diffuse *= attenuation;
            specular *= attenuation;

            result += diffuse + specular;
        }

        fragColor = vec4(result, uAlpha);
    }`;
```

The fragment shader uses a Phong lighting model which is a commonly used shading model in computer graphics for simulating the interaction of light with surfaces. It computes the

surface normal ‘normal’ and view direction ‘viewDir’ to determine how light interacts with the surface. ‘oFragPosition’ and ‘oCameraPosition’ represent the fragment and camera positions respectively, aiding in calculating lighting effects based on their relative position. It implements the Phong lighting model by considering ambient, diffuse, and specular components for each light source. The diffuse and specular components are attenuated over distance ‘attenuation’ to simulate a darker appearance as objects move away from light sources. The shader includes texture mapping using a sampled texture ‘uTexture’ at specified UV coordinates ‘oUV’. This aligns with our theory of applying textures to surfaces to enhance their appearance. The ‘textureColor’ variable determines the final color based on the sampled texture or a default color ‘diffuseVal’ if no texture is present. The shader also calculates the interaction between surface normals, light directions, and view directions to determine what is visible to the viewer. Transparency can also be achieved by modifying the alpha component of the object., incorporates texture mapping if available, and calculates ambient, diffuse, and specular lighting for each light source, considering attenuation over distance to make the game darker in the diffuse and specular component.

$$\text{Color} = k_a \cdot \text{La} + \left(\frac{k_d \cdot (\mathbf{N} \cdot \mathbf{L})}{\text{attenuation}}\right) + \left(\frac{k_s \cdot (\mathbf{R} \cdot \mathbf{V})^n}{\text{attenuation}}\right)$$

Alpha(Brian)

We set the pixel blending equation for when objects are transparent or overlap to use the source alpha to determine the blending factor for its color. The destination pixel’s alpha is subtracted from 1-src alpha to also determine a blending factor.

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

Score Display (BONUS)(Brian):

```
#scoreDisplay {
    position: absolute;
    top: 20px;
    left: 20px;
    font-size: 24px;
    color: white;
}
```

```
<div id="scoreDisplayContainer">
    <div id="scoreDisplay">Score: 0</div>
</div>
```

```
// A function to update the displayed score
updateScoreDisplay() {
    const scoreDisplay = document.getElementById('scoreDisplay');
    if (scoreDisplay) {
        scoreDisplay.innerText = `Score: ${this.playerScore}`;
    }
}
```

Score is displayed in the top left of the screen and gets updated upon an enemy death which gets checked every frame.

Enemy AI Teleport(Brian):

```
1134        // Enemy AI unstuck mechanic
1135        let enemyStuckTimer = 0;
1136        const enemyStuckThreshold = 5; // Time limit until enemy gets forcefully repositioned
1137        const boundaryBuffer = 1; // buffer zone if their close but not touching the boundary. May need adjusting.
1138        const isStuck =
1139        (this.enemy1.model.position[0] >= maxX - boundaryBuffer|| this.enemy1.model.position[0] <= minX + boundaryBuffer) ||
1140        (this.enemy1.model.position[1] >= maxY - boundaryBuffer|| this.enemy1.model.position[1] <= minY) + boundaryBuffer||
1141        (this.enemy1.model.position[2] >= maxZ - boundaryBuffer|| this.enemy1.model.position[2] <= minZ + boundaryBuffer);
1142
1143
1144        if (isStuck) {
1145            enemyStuckTimer += deltaTime; // Increment the timer when the enemy is stuck
1146            //console.log(`Enemy stuck timer: ${enemyStuckTimer}`); // dev tool
1147            if (enemyStuckTimer >= enemyStuckThreshold) {
1148                // Teleport the enemy x units in front of the spaceship
1149                const spaceshipPosition = getObject(this.state, "SpaceShip").model.position;
1150                const teleportDistance = 100;
1151                this.enemy1.model.position = vec3.fromValues(
1152                    spaceshipPosition[0],
1153                    spaceshipPosition[1],
1154                    spaceshipPosition[2] + teleportDistance
1155                );
1156
1157                // Reset the stuck timer after teleportation
1158                enemyStuckTimer = 0;
1159                console.log("Enemy has been teleported due to being stuck")
1160            }
1161        } else {
1162            // Reset the timer when the enemy is not stuck
1163            enemyStuckTimer = 0;
1164        }
1165        }
```

Since the enemy bounces back after hitting a boundary, sometimes the enemy gets stuck near the boundary permanently but is not quite touching the boundary. The game checks if it's stuck near the boundary for longer than 5 seconds then it copies the position of the player and replaces the enemies position with it to set the enemy in front of the player again. This prevents the enemy from getting stuck in a place the player cannot reach.

Object Iteration Loop (Matt)

```
// Loop for objects in gamestate
this.state.objects.forEach((object) => {

    if (object.name.startsWith('Enemy')) {
        object.rotate('z', Math.random() * 0.1); // Rotate the saucers
    }

    // Update the position of the light source to follow the enemy
    this.enemy1light.position[0] = this.enemy1.model.position[0];
    this.enemy1light.position[1] = this.enemy1.model.position[1];
    this.enemy1light.position[2] = this.enemy1.model.position[2] + 47.5;

    // Player Projectiles
    if (object.name.startsWith('Cube-')) {
        object.rotate('z', Math.random() * 1.5);

        /*
        if (this.frame % 500 == 0){
            console.log(object.model.position[0] + ' ' + object.model.position[1] + ' ' + object.model.position[2]);
        }
        */

        if ( this.is_between(this.enemy1.model.position[0], object.model.position[0], 3)
            && this.is_between(this.enemy1.model.position[1], object.model.position[1], 2.5)
            && this.is_between(this.enemy1.model.position[2] + 37, object.model.position[2], 2) ){

            if (!this.intermission) {
                this.enemy1Health -= 10;
                this.updateHealthBar()
            }
            console.log("UFO HIT!");
            object.model.position = vec3.fromValues(0, 0, 1000);

        }
```

This is a part of the onUpdate() game loop, where we iterate through each game object. We can identify each object by type, and apply any functions we need to on each object. Some objects need to rotate only on one axis, and others may need to rotate on all axes. We can also check for collisions on each object with other objects. Once collisions take place, we can update variables such as player health and the associated UI elements.

Collision Logic (Matt)

```
is_between(A, B, distance){
      var min = B - distance;
      var max = B + distance;

      if( A > min && A < max){
         return true;
      }else{
         return false;
      }
   }
```

This is a simple collision detection calculation. This function checks if a point is between two other points. When this calculation is done three times, on the x, y, and z axes, a 3 dimensional point can be checked if it is inside a 3d rectangle. Since the distance value can be changed for each dimension individually, we can create a rectangle of any size.