Cave Invaders

The New Old Game Nobody Is Playing

`https://github.com/michaelmrose/caveInvaders`

1 Overview

Today I will present my game Cave Invaders, riffing on the name and aesthetic of old Atari games. We shall celebrate their aesthetic sensibilities, if not their economy, by using more computing power than some supercomputers of that era to render a grid of colored squares.

The game is simple in gameplay. You must score as many points as possible by eradicating alien invaders until you succumb or an alien ship reaches your base.

2 Design

The game is more complex in design. Every entity has a collection of one or more points dynamically calculated by base position and rotation in relation to their origin on a game-wide 2-dimensional metadata array that serves as a meta map of the level.

We accomplish this by scribbling an object reference on the point or points. "Alien wuz here." When we move or rotate, we clear the old points, claim new ones and check for collision. If we collide, we call foo.onCollide(bar) and bar.onCollide(foo), mostly deferring to the object as to how to handle the collision so this behavior can be easily customized. Most frequently implementing the default behavior of calling destroy on self.

3 Strategies

Strategies for autonomous units are simple functions pushed onto a stack of functions each unit carries to make logic configurable, stackable, and easy to customize. For example, one might use this snippet of code to tell an alien ship that it ought to follow the target but also continually evaluate priorities to determine the current target.

```
class AlienShip extends Ship {
    constructor(x, y, game, position) {
        super(x, y, game, position);
        this.colors = ["green", "lightgreen"];
        this.ticksToShoot = 80;
        this.ticksToMove = 7;
        this.target = undefined;
        this.strategies.push(() => chargeShot(this));
        this.strategies.push(() => shootPlayer(this));
        this.strategies.push(() => followTarget(this, this.target));
        this.strategies.push(() =>
            selectNearestTargetFromArrayWeightedByPriority(
                this,
                this.game.player,
                this.game.base
            )
        );
    }
    onDestroy() {
        this.dieSound.play();
    }
}
```

4 Progression of the Game

The logic moves forward one tick at a time; we apply every strategy for every entity, apply any actions the player has queued up by pressing buttons, move everyone, collide, everyone, handle collisions by class logic, and handle destruction by pushing entity to the graveyard and render the survivors. Then at the start of the next tick, we'll process everything in the graveyard and clear its positions to avoid them striking at their fellows from beyond the grave.

```
tick() {
    if (this.paused === false) {
        //clear the positions occupied by destroyed entities
        //a second tier "buried" shouldn't be needed added to avoid
        //double counting aliens for score should be fixed by just not
        //doing that in the future
        this.graveyard.forEach((e) => {
            if (e instanceof AlienShip && !this.buried.includes(e)) {
                this.score++;
                this.buried.push(e);
            }
            e.positions().forEach((p) => (this.board[p.y][p.x] = 0));
        });
        //to hold entities to be destroyed at start of tick
        this.graveyard = [];
        //this will be examined to allow destoryed enemy ships to be counted
        this.buried = [];
        this.generateEnemies();
        //ensure metadata map holds correct points for each entity
        this.entities.forEach((e) => e.claimPointsOnBoard());

        // each entity has an array of functions it should evaluate every tick herein termed
        // actions are strategies for player entity should simply be merged with strategies

        this.entities.forEach((e) => {
            e.strategies.forEach((s) => {
                s(e);
            });
            e.actions.forEach((a) => {
                a();
            });
            e.actions = [];
        });
    }
}
```

Handling the destruction of entities separately is necessary because otherwise, we end up trying to destroy things that are already gone or often not at all. In the worst case, entities become invisible as they drop off the game's entities array and aren't rendered but still collide.

5 Challenges

The previous example illustrates the most significant challenge. Namely, the meta map of the level can be out of sync with the reality presented by entities. This design is less than ideal, but it provides the game with the capability to have any number of entities on the screen and handle collision in the same amount of time per entity. This will later be used to form a more complex level out of destroyable rocks loaded from image files by examining the color values of pixels.

6 Fun

To make the game challenging, we generate aliens on the board's periphery, and we do so by randomly generating positions, discarding the ones not within scope, and feeding them to our ship constructor function. We decide how many to generate and how fast by mutating values that control how many ticks between generations and how many ships to generate while keeping these values to semi-sane maximums.

```
randomPositionWithinBoard() {
    let x = rand(3, this.width - 3);
    let y = rand(3, this.height - 3);
    return [x, y];
}

randomPositionOnPeripheryOfBoard() {
    while (true) {
        let r = this.randomPositionWithinBoard();
        let x = r[0];
        let y = r[1];
        let xRange = range(3, 21).concat(this.width - 20, this.width - 2);
        let yRange = range(3, 21).concat(this.height - 20, this.height - 2);
        if (xRange.includes(x) || yRange.includes(y)) {
            return r;
        }
    }
}

// ticks both get closer together and spawn more enemies as time goes by
generateEnemies() {
    this.nthTick++;
    if (this.nthTick % this.ticksToGenerateEnemies === 0) {
        for (let i = 0; i < this.enemiesToSpawn; i++)
            new AlienShip(
                ...this.randomPositionOnPeripheryOfBoard(),
                this,
                "up"
            );
        this.ticksToGenerateEnemies = Math.max(
            10,
            this.ticksToGenerateEnemies - 1
        );
        this.enemiesToSpawn = Math.min(this.enemiesToSpawn + 1, 8);
    }
}
```

7 Future

I look forward to adding more types of ships with their own logic, weapons, hp, power-ups, and more complex levels larger than one screen.

8 Code Examples

8.1 AlienShip Class

```
class AlienShip extends Ship {
    constructor(x, y, game, position) {
        super(x, y, game, position);
        this.colors = ["green", "lightgreen"];
        this.ticksToShoot = 80;
        this.ticksToMove = 7;
        this.target = undefined;
        this.strategies.push(() => chargeShot(this));
        this.strategies.push(() => shootPlayer(this));
        this.strategies.push(() => followTarget(this, this.target));
        this.strategies.push(() =>
            selectNearestTargetFromArrayWeightedByPriority(
                this,
                this.game.player,
                this.game.base
            )
        );
    }
    onDestroy() {
        this.dieSound.play();
    }
}
```

8.2 Game.tick function

```
tick() {
    if (this.paused === false) {
        //clear the positions occupied by destroyed entities
        //a second tier "buried" shouldn't be needed added to avoid
        //double counting aliens for score should be fixed by just not
        //doing that in the future
        this.graveyard.forEach((e) => {
            if (e instanceof AlienShip && !this.buried.includes(e)) {
                this.score++;
                this.buried.push(e);
            }
            e.positions().forEach((p) => (this.board[p.y][p.x] = 0));
        });
        //to hold entities to be destroyed at start of tick
        this.graveyard = [];
        //this will be examined to allow destoryed enemy ships to be counted
        this.buried = [];
        this.generateEnemies();
        //ensure metadata map holds correct points for each entity
        this.entities.forEach((e) => e.claimPointsOnBoard());

        // each entity has an array of functions it should evaluate every tick herein termed
        // actions are strategies for player entity should simply be merged with strategies

        this.entities.forEach((e) => {
            e.strategies.forEach((s) => {
                s(e);
            });
            e.actions.forEach((a) => {
                a();
            });
            e.actions = [];
        });
    }
}
```

8.3 Randomly Generate Increasing Number of Enemies

```
randomPositionWithinBoard() {
    let x = rand(3, this.width - 3);
    let y = rand(3, this.height - 3);
    return [x, y];
}

randomPositionOnPeripheryOfBoard() {
    while (true) {
        let r = this.randomPositionWithinBoard();
        let x = r[0];
        let y = r[1];
        let xRange = range(3, 21).concat(this.width - 20, this.width - 2);
        let yRange = range(3, 21).concat(this.height - 20, this.height - 2);
        if (xRange.includes(x) || yRange.includes(y)) {
            return r;
        }
    }
}

// ticks both get closer together and spawn more enemies as time goes by
generateEnemies() {
    this.nthTick++;
    if (this.nthTick % this.ticksToGenerateEnemies === 0) {
        for (let i = 0; i < this.enemiesToSpawn; i++)
            new AlienShip(
                ...this.randomPositionOnPeripheryOfBoard(),
                this,
                "up"
            );
        this.ticksToGenerateEnemies = Math.max(
            10,
            this.ticksToGenerateEnemies - 1
        );
        this.enemiesToSpawn = Math.min(this.enemiesToSpawn + 1, 8);
    }
}
```