3110 Final Project Design

Manish Saha: ms3236
Kenneth Li: kql3
Collin Montag: cm759

System description

Core vision
To create a game where the player must gather resources and use them to defend against waves of enemies.

Key features
- Standalone graphical top-down game
- User-controlled “ship” surviving against infinite waves of AI enemies
- Player can collect certain raw materials (bits) from areas of a large map and use them to either unlock new weapons or build structures to automatically attack
    - Player constructs a chassis holding the weapon they are currently using
- Player gains points from defeating enemies; successive waves will reward more points
- Weapon unlocks done in tech-tree fashion

Narrative Description
Our game consists of a WASD-controlled player on a finite, circular world. There are randomly generated features scattered throughout the world: resource fields and enemy spawn locations. The objective of the game is to survive as long as possible against enemies of increasing difficulty with the help of player-built structures and weapon unlocks.

System & Module Design

Our system implements the modules Input, Action, State, Ai, Canvas, Main, Js_utils, Utils, and Types.

Input
Input sets event listeners to collect user input and updates an internal input state. The exposed functions allow other modules to query it for the current input state.

Action
Action parses user inputs into discrete actions that communicate intent to change the state. These encompass any ability provided to the player in-game that can affect the information stored in the game state, such as movement, mouse clicks, and UI element interactions.

State & Ai

State houses all internal information and functions relevant to gameplay. This includes the player information, such as equipment, health, and upgrades; enemy and structure information, including attributes and locations; and other game state information, like points, mode, and other information to be rendered on the UI.

Ai contains a multitude of state helper functions that do everything from firing bullets to allowing structures to track enemies to allowing enemies to follow the player. Many of the functions are reusable and polymorphic (to the extent allowed by OCaml) — for example, the same bullet firing function is called for players, enemies, and structures.

Canvas
Canvas provides visual representations of everything within the game state. It uses the compiler js_to_ocaml to make more flexible and powerful graphical tools available. It draws all map details — player, enemies, structures, bullets, backgrounds — and the UI.

Main
Main initializes the game state when the user begins playing and calls the browser's `requestAnimationFrame` to register the next iteration of the game loop. The loop itself includes calls to Input to fetch input state, Action to parse actions, State to advance the game state, and finally Canvas to render the game.

Js_utils, Utils, and Types
Utils and Types contain helper functions and all the types needed between files to prevent circular dependencies between the other modules. Js_utils likewise contains JavaScript-related utils that cannot be loaded when running outside of a browser (such as when running `make test`).

Data

Single source of truth
We maintain a single game state passed from loop to recursive loop. This large game state object, of type `State.state` , is the single source of truth for everything, be it creating new states through `State.advance` or drawing to the canvas in `Canvas`. This single game state is the basis of new states, along with input from the user and the passage of time.

State organization
The game state contains information about the player, enemies, structures, and bullets. These element references contain attributes like health, size, attack, cooldown, and speed.

Data structures
We use lists extensively in order to communicate collections of enemies, structures, etc. These lists are iterated through when creating a new state or when rendering a state to the graphical canvas. Also, the upgrade system relies on a record structure that uses lists to determine upgrade dependencies. Records and variants are used to represent all the OCaml types we have implemented to run our game.

External Dependencies

We use js_of_ocaml in order to compile our OCaml source code into JS, thus giving us access to a robust graphics library in the Canvas API. We also use Python Flask to serve the static files required to run the game.

Implementation Plan

Specializations
Kenneth: Front-end (Graphics, Main, Input, Action)
Manish: Game logic (State)
Collin: State interactions (Ai, State)

Weekly Agenda
April 17th

- GUI mockups done, rudimentary graphics functions working (drawing shapes)
- Input capturing and typing implemented
- Simple enemies and player representations implemented
- Basic game initialization and looping done

April 24th

- Rudimentary graphic designs done, drawing functions work at basic level
- Action typing implemented, derivation from a given input state done
- Some weapons, world implemented

May 1st

- Drawing functions fleshed out
- Basic enemy AI implemented
- More weapons, some structures, shooting implemented
- Playtesting

May 8th

- Game loop logic improved with automatic updates/collision checks
- More work on enemy AI, building structures implemented
- Most graphic designs done
- Resource fields implemented
- Playtesting

May 15th

- Collision checks, entity AI vastly improved
- Drawing of structures streamlined for fluidity
- Structures fully implemented with AI algorithms
- Finalized graphics
- Stepping of bullets, enemies, and structures streamlined
- Randomization of enemy waves and resource fields done
- Points system and wave timers
- Playtesting

Changes

Between Milestones 2 and 3, we prioritized developing according to our game's guidelines. Along the process, we swapped some of our game's features for more fitting ones. For example, we swapped the experience and stats system for point counter, which is better suited to a survival game. In addition, this rescoping allowed us to focus much more on creating a more fun and robust game. This meant we were able to implement more weapons and enemies to vary with successive waves, as well as better AI to present a greater challenge.

Testing

Unit tests
We used end-to-end unit tests to validate all state advancements. In order to test the functionality of `advance` for a given timestep, we implemented unit tests for each supporting function. This allowed us to confirm that advancing each individual action would properly change the state; thus, in playtesting, we could focus on the correctness of the interactions of these advance helpers and the semantics of the underlying game engine.

Playtesting
Playtesting was a crucial part of our development. In many cases, unit tests were infeasible (assertions on float equality were very error prone, and many functions had complex computations that were unverifiable by hand). Thus, we relied heavily on the actual gameplay experience to find bugs and verify the overall correctness of collision detection, enemy spawning, game mechanics, and enemy/structure autonomy, as well as get a sense of game balance.

Known Problems

We know of several bugs in our collision detection algorithm. One is the random overlap of enemies from time to time, and their movement toward each other. We aren't sure why our algorithm doesn't work in these cases, but ultimately decided that the bug was not disruptive and too time-consuming to fix. The other bug is that the player can force through gaps smaller than should be possible, likely due to our sequential resolution of collisions. With more time, we would have implemented a more robust collision detection algorithm that hopefully would solve these problems.

In addition, due to a bash script mishap, ~200 commits were doubled and display erroneously.

Division of Labor

Manish
Primarily worked on `State`. This module handled all possible player interactions that were passed through `Canvas`. Some functions in state were also devoted to randomization (i.e. generating bitfield locations and bullets). Also worked on the process of firing and spawning bullets from all the available weapons in the game in `Ai`. Each weapon had a unique purpose, and hence, this required handling all weapon ids separately. For instance, the rocket required an extra `check_explosion` function in `Ai` to animate the explosion upon collision with a rocket. Another task was to create multiple checks for `Canvas` to recognize

when to draw an entity/structure/bitfield. This allowed the game to process smoothly and without any lag. `State` was also responsible for building structures and advancing the upgrades from player interactions. All in all, `State` allowed the front end to properly construct an instance of the game at any moment.

Kenneth
Primarily worked on `Canvas`, `Input`, `Action`, and `Main`. Prior experience with JavaScript facilitated understanding of the `Js_of_ocaml` library and the semantics of canvas animation and event handling. Handled graphic and UI design. Wrote Python static file server and main game loop. `Canvas` had to translate a given game state into all the necessary graphics and UI updates on every loop.

Collin
Primarily worked on `Ai` and `State`. This included designing advance functions to handle stepping all the entities in the game. Advancing structures, bullets, and enemies are all handled in `State`. Functions called within these advances are integrated in `Ai` as movement vector calculation, collision handling, and passive checks. Enemy movement specifically required thorough calculations for matching tracking patterns and customizing target priority. Generalized most functions to handle any passed entity type and systematically scanned entity lists in state to minimize time complexity. Also designed all enemy types and implemented the procedural wave spawning system.