# Iron Choir

Final Project Report

---

A single-player, player-versus-environment turn-based strategy game with an API designed for adversarial AI integration.

---

| | |
|---|---|
| **Client:** | Lockheed Martin Advanced Technology Laboratories |
| **Instructor:** | Simanta Mitra |
| **Teaching Assistant:** | Haniyeh Fekrmandi |
| **Institution:** | Iowa State University |

## Team Members


| | |
|---|---|
| Noah Shepardson | Game Loop and API |
| Derek Louden | Entities |
| Christian Borek | User Interface |
| Carter Scott | Map |


May 11, 2026

# Contents

**Client Letter** **1**

**1 Requirements** **2**
1.1 Project Purpose . . . . . 2
1.2 Functional Requirements . . . . . 2
1.3 Nonfunctional Requirements . . . . . 3

**2 Design** **4**
2.1 System Overview . . . . . 4
2.2 Architecture . . . . . 4
2.3 Game Loop Design . . . . . 5
2.4 API and State Model . . . . . 5
2.5 Map Design . . . . . 6
2.6 Entity and Combat Design . . . . . 6
2.7 User Interface Design . . . . . 7

**3 Work Done** **8**
3.1 Noah Shepardson - Game Loop and API . . . . . 8
3.2 Derek Louden - Entities . . . . . 8
3.3 Christian Borek - User Interface . . . . . 9
3.4 Carter Scott - Map . . . . . 9

**4 Results Achieved** **10**
4.1 Playable Game Loop . . . . . 10
4.2 AI-Ready State and Action Spaces . . . . . 10
4.3 Procedural Tactical Maps . . . . . 10
4.4 Entity, Combat, and Enemy Behavior . . . . . 11
4.5 Functional User Interface . . . . . 11
4.6 Development Outcome . . . . . 11

# Client Letter

**Attachment Instructions:** Attach the client letter here.

# Chapter 1

# Requirements

## 1.1 Project Purpose

Iron Choir was developed as a single-player, player-versus-environment turn-based strategy game for Lockheed Martin Advanced Technology Laboratories. The central project requirement was to create a game environment that can expose individual player strategy to an adversarial artificial intelligence model. Instead of relying only on conventional scripted difficulty, the game provides structured state and action data so that a separate adversarial AI can observe play, reason about player behavior, and influence enemy decisions.

## 1.2 Functional Requirements

The project requirements can be summarized as follows:

1. **Turn-based tactical gameplay.** The game must support a grid-based tactical loop where the player deploys and commands mechs against enemy Vek units.
2. **Procedural level generation.** The game must generate an 8 by 8 battlefield with varied biomes, terrain types, destructible objects, buildings, and spawn zones.
3. **Objective and failure conditions.** The player must be able to win by eliminating enemies and lose by losing all mechs or the map's power grid.
4. **Clear state representation.** The backend must expose the current turn, phase, result, map state, entity state, enemy intents, and player/UI selection state.
5. **Clear action representation.** The game must define a consistent action space for deployment, movement, attacks, ending turns, quitting, enemy moves, enemy intent selection, enemy intent execution, and enemy spawning.
6. **Bidirectional AI integration interface.** The game must support interaction with a separate modeling team by exposing game state and accepting commands without

requiring model-side changes to core game logic.

7. **Backend-controlled validation.** User interface actions must not mutate the authoritative game state directly. Player intents must be converted into backend commands and validated before any state change occurs.
8. **Playable user interface.** The game must show the board, units, terrain, health, power grid state, valid moves, valid attacks, current phase, and clear player prompts.

## 1.3 Nonfunctional Requirements

The system was also designed around several nonfunctional requirements:

- **Determinism at phase boundaries:** turn phases should advance predictably so the adversarial AI can reason over discrete game states.
- **Modularity:** the frontend, map logic, entity logic, game loop, and API representation should be separable to support parallel team development.
- **Extensibility:** enemy behavior weights and state/action spaces should be tunable or replaceable by the adversarial AI team.
- **Validation and safety:** illegal moves, illegal attacks, invalid deployments, and unsupported commands should be rejected by the backend.
- **Replayable variety:** procedural maps and biome-specific terrain should create different tactical conditions across sessions.

# Chapter 2

# Design

## 2.1 System Overview

Iron Choir is implemented in Python. The backend owns the authoritative game state, including the finite-state game loop, map, entity manager, combat resolution, and command validation. The frontend uses pygame to render the current state, collect player intent, and forward commands to the backend. This separation prevents rendering code from directly changing game logic and allows backend features to evolve without requiring substantial frontend rewrites.

## 2.2 Architecture

The project is divided into four main technical areas:

| **Area** | **Responsibility** |
|---|---|
| Game Loop and API | Controls phase transitions, validates commands, exposes state and action spaces, and provides the interface needed by the adversarial AI team. |
| Map | Generates the 8 by 8 tactical grid, biomes, terrain, building placement, spawn zones, validation, terrain destruction, and power grid interactions. |
| Entities | Defines mechs and enemy Veks, movement validation, attack sets, combat resolution, knockback, environmental damage, and heuristic enemy behavior. |
| User Interface | Renders the board, sidebar, entities, terrain, prompts, buttons, highlights, and enemy-turn animations through pygame primitives. |

## 2.3 Game Loop Design

The game loop uses a finite-state machine with nine phases:

1. `BOOT`: instantiate required services.
2. `SETUP`: initialize the map, entity manager, and enemy placement.
3. `DEPLOY`: allow the player to place the initial mech squad.
4. `ENEMY_PRETURN`: choose enemy moves and planned attacks.
5. `PLAYER_TURN`: allow the player to move, attack, end the turn, or quit.
6. `ENEMY_EXECUTE`: execute previously selected enemy intents.
7. `ENVIRONMENT`: apply environmental damage and tile effects.
8. `CLEANUP`: evaluate win/loss conditions, reset transient data, and advance the turn.
9. `END`: cleanly stop the game.

This design was selected because a turn-based strategy game benefits from strict control over when actors are allowed to modify state. The discrete phases also make the game easier for an adversarial AI model to consume than a continuous real-time simulation.

## 2.4 API and State Model

The API design is organized around explicit state and command objects. The game state contains the turn number, current phase, running flag, result, world object, map grid, entity manager, and enemy intents. The command model provides a compact action interface for the frontend and future AI agents.

Listing 2.1: Command types used by the backend API.

```
class CommandType(Enum):
    END_TURN = auto()
    QUIT = auto()
    MOVE = auto()      # payload: entity_id, row, col
    ATTACK = auto()    # payload: entity_id, row, col
    DEPLOY = auto()    # payload: entity_id, row, col
```

The player-facing command path is intentionally indirect: the frontend detects input, translates that input into an intent, converts the intent into a backend command, and submits it for validation. The backend rejects invalid commands with structured reasons such as `entity_not_found`, `invalid_move`, `invalid_attack`, `already_moved`, `already_attacked`, or `unsupported_command`. This keeps the authoritative rules in one place.

## 2.5 Map Design

The map is an 8 by 8 tile-based tactical environment. Each tile has terrain attributes such as movement cost, passability, destructibility, and biome-specific effects. Supported biome structures include:

- **Default:** grass, mountains, and water.
- **Island:** grass and water, with the map fully waterlocked.
- **Jungle:** grass, spikes, and hornet nests.
- **Desert:** sand, quicksand, and cacti.
- **Snow:** snow, mountains, and ice.

The generator combines clustered growth and probability-based placement. Larger formations such as lakes, mountains, ice fields, and quicksand are grown from seed points through breadth-first expansion with stochastic pruning. Smaller features such as cacti, spikes, and hornet nests are placed through budget-controlled clustering. A global tile-cost budget prevents maps from becoming overloaded with high-impact terrain.

After terrain generation, the map is validated through breadth-first search over passable tiles. If any passable region is isolated, the map is discarded and regenerated. Buildings are then placed on valid biome-compatible terrain. The number of active buildings contributes to the power grid, so terrain destruction and building damage have direct consequences for the loss condition.

## 2.6 Entity and Combat Design

Iron Choir includes five entity types split across two factions. Player mechs include the CombatMech, ArtilleryMech, and CannonMech. Enemy Veks include the Firefly and Hornet. All entities share a base class with common attributes such as ID, hit points, damage, movement range, straight knockback, radial knockback, and attack pattern behavior.

The `EntityManager` mediates map/entity interaction. Movement is calculated through breadth-first expansion up to the entity's movement range, while filtering invalid coordinates, impassable terrain, and occupied tiles. Combat resolves direct damage, straight knockback, radial knockback, collision damage, and tile damage. Environmental effects are applied after enemy execution; for example, Fireflies on water are killed, and entities on quicksand are killed.

Enemy behavior is implemented through a heuristic AI with tunable weights. Movement decisions minimize weighted distance to important targets when no attack is available. Attack decisions use a cost function that rewards damaging or killing mechs and buildings, penalizes friendly fire, and accounts for knockback and collision outcomes. These tunable

parameters provide a baseline opponent and a useful control point for the adversarial AI team.

## 2.7 User Interface Design

The user interface is implemented in pygame using primitive shapes rather than external sprites. The frontend is organized around data translation, adaptation, and rendering:

- `data.py`: frontend data classes.
- `adapter.py`: translation from backend state to frontend-readable state.
- `ui.py`: visual rendering and input handling.
- `main.py`: main loop coordination, rendering, re-rendering, input polling, and command dispatch.

The UI presents the grid on the left, a sidebar on the right, and a top status bar. Tile types are drawn separately and may include animations, biome-specific variants, and contextual building rendering. Entities are drawn by layering pygame primitives, and idle animations use a sine-wave bobbing effect.

Enemy animations are handled through a custom queue system. The frontend snapshots state before and after a backend enemy phase, compares the snapshots, and renders derived events such as projectiles, impact flashes, shake effects, fade-outs, and sliding movement. Player input is blocked while the animation queue is active.

# Chapter 3

# Work Done

## 3.1 Noah Shepardson - Game Loop and API

Noah's work focused on the finite-state game loop, phase transitions, command validation, and the API structure needed for the modeling team. The major contributions include:

- Implemented the nine-phase game loop from boot through end state.
- Defined the state space used by the game and adversarial AI integration.
- Defined the action space for deployment, movement, attacks, ending turns, quitting, enemy movement, enemy intent selection, and enemy execution.
- Implemented backend command validation so invalid UI or AI commands cannot directly corrupt game state.
- Coordinated the separation between frontend input and backend authority.
- Learned and applied finite-state-machine design, command-pattern style interfaces, Python dataclasses/enums, and API boundary design for AI integration.

## 3.2 Derek Louden - Entities

Derek's work focused on unit representation, entity management, movement, combat, environmental interactions, and enemy behavior. The major contributions include:

- Implemented the shared entity base design for player mechs and enemy Veks.
- Implemented CombatMech, ArtilleryMech, CannonMech, Firefly, and Hornet behavior.
- Built movement validation using breadth-first search expansion.
- Implemented occupancy rules by tile and entity type.
- Implemented combat resolution, knockback, collision damage, tile damage, and environmental damage.

- Developed heuristic enemy movement and attack selection using tunable weights.
- Learned and applied grid-search algorithms, turn-based combat resolution, class inheritance, and AI heuristic evaluation functions.

## 3.3 Christian Borek - User Interface

Christian's work focused on the pygame frontend, state adaptation, rendering, input handling, and animations. The major contributions include:

- Implemented pygame rendering using primitive shapes rather than external sprites.
- Built frontend data classes and an adapter layer that translates backend state into UI-readable state.
- Designed the visual layout with grid, sidebar, and top status bar.
- Added tile rendering, biome-specific visuals, entity visuals, idle animations, health bars, and power grid display.
- Implemented input intent polling and command forwarding while keeping the UI stateless with respect to backend game rules.
- Implemented custom enemy-turn animations using before/after state snapshots and an animation queue.
- Learned and applied pygame rendering, event polling, custom animation sequencing, UI state adaptation, and frontend/backend separation.

## 3.4 Carter Scott - Map

Carter's work focused on procedural map generation, terrain types, biome rules, validation, destructible terrain, buildings, and spawn zones. The major contributions include:

- Implemented the 8 by 8 tactical map structure.
- Designed five biome types with distinct terrain rules and gameplay effects.
- Implemented clustered terrain growth through breadth-first expansion and stochastic pruning.
- Implemented budget-controlled terrain placement to balance map complexity.
- Implemented connectivity validation so all passable playable spaces remain reachable.
- Implemented building placement, power grid interaction, spawn-zone filtering, and destructible terrain updates.
- Learned and applied procedural generation, graph traversal, tile-cost balancing, terrain validation, and grid-based level design.

# Chapter 4

# Results Achieved

The final Iron Choir prototype achieved the core requirements for a playable tactical game and an AI-ready backend structure. The most important results are listed below.

## 4.1 Playable Game Loop

The team produced a working turn-based gameplay loop with deployment, enemy planning, player action, enemy execution, environmental effects, cleanup, and end conditions. The player can deploy three mechs, respond to visible enemy intents, move and attack with validated commands, and complete matches under explicit win/loss conditions.

## 4.2 AI-Ready State and Action Spaces

The game exposes a structured state model and action model suitable for adversarial AI integration. The modeling team can reason over map state, entity state, phase, turn, result, enemy intents, and valid commands without modifying the core game logic.

## 4.3 Procedural Tactical Maps

The team implemented five biome families and a procedural generation system that creates varied 8 by 8 battlefields. Connectivity validation ensures generated maps remain playable, while tile costs and biome-specific rules keep maps strategically varied without being uncontrolled.

## 4.4 Entity, Combat, and Enemy Behavior

The team implemented a complete entity system with multiple player and enemy unit types, movement validation, attack patterns, knockback, collision damage, tile destruction, building destruction, environmental hazards, and a baseline heuristic AI. This gives the game enough tactical complexity for player behavior to be observable.

## 4.5 Functional User Interface

The pygame frontend renders the game board, terrain, units, animations, valid movement and attack highlights, player prompts, health information, and power grid state. The UI supports player deployment and turn execution while respecting backend state ownership.

## 4.6 Development Outcome

The project produced a cohesive prototype demonstrating the intended architecture: a playable game shell, strict game-state authority in the backend, a frontend that acts as a view/input layer, and a structured integration path for adversarial AI experimentation.