IL RELITTO SILENTE

DOCUMENTAZIONE TECNICA DI PROGETTO

Architettura, Design e Sviluppo di un'Avventura Testuale Fantascientifica in Lingua Italiana

Voce	Dato
Autore	Simone Pizzi
Anno	2026
Versione di riferimento	1.2.5 · Prima Edizione
Tipologia	Documentazione tecnica di progetto software / Design document

> *«Non c'è grafica che anticipi l'immaginazione. C'è solo il testo, che indica una direzione, e la mente che costruisce il paesaggio.»*
>
> dal corpus progettuale de Il Relitto Silente

Abstract

Abstract (Italiano)

Il presente documento descrive in maniera sistematica e approfondita l'architettura tecnica, le scelte di design e il processo di sviluppo de *Il Relitto Silente*, un'avventura testuale interattiva fantascientifica in lingua italiana, distribuita come applicazione desktop standalone per il sistema operativo Windows mediante il framework Electron.

Il progetto integra un motore testuale di parsing sviluppato in TypeScript con un sistema di narrazione ambientale distribuita su quindici ambienti, un sottosistema di sintesi audio procedurale basato sulle Web Audio API, un'interfaccia grafica che emula l'estetica dei terminali CRT degli anni Settanta e Ottanta, e un layer di persistenza duale compatibile sia con l'ambiente Electron sia con ambienti browser.

Vengono analizzati in dettaglio: il modello computazionale delle stanze, la funzione di normalizzazione dell'input testuale, il sistema di combinazione degli oggetti, il meccanismo di traduzione progressiva della lingua aliena K'tharr, la gerarchia dei suggerimenti contestuali e i profili di sintesi sonora ambientale.

Il documento esamina inoltre il processo di sviluppo assistito da modelli di linguaggio di grandi dimensioni (LLM), con una riflessione sulle modalità di collaborazione uomo–macchina adottate e sulle implicazioni metodologiche che ne derivano per lo sviluppo di software creativo a bassa numerosità di team.

Parole chiave: avventura testuale, interactive fiction, parser testuale, TypeScript, React, Electron, Web Audio API, sintesi procedurale, sviluppo assistito da IA, design narrativo, linguaggio K'tharr.

Abstract (English)

This document provides a systematic and detailed account of the technical architecture, design decisions, and development process of *Il Relitto Silente* (The Silent Wreck), a science-fiction interactive text adventure written in Italian, distributed as a standalone desktop application for Windows via the Electron framework.

The project integrates a TypeScript-based text parsing engine with an ambient narrative system distributed across fifteen rooms, a procedural audio synthesis subsystem built on the Web Audio API, a graphical interface emulating the CRT terminal aesthetic of the 1970s and 1980s, and a dual-mode persistence layer compatible with both the Electron environment and standard browser environments.

The document provides in-depth analysis of: the room computation model, the text input normalisation function, the object combination system, the progressive translation mechanism for the alien K'tharr language, the contextual hint hierarchy, and the ambient audio synthesis profiles.

The document also examines the development process as assisted by large language models (LLMs), reflecting on the human–machine collaboration methodologies adopted and on the methodological implications for creative software development within small or single-person teams.

Keywords: text adventure, interactive fiction, text parser, TypeScript, React, Electron, Web Audio API, procedural synthesis, AI-assisted development, narrative design, K'tharr language.

> **Nota sul documento.** Il presente volume è concepito come un design document esteso, con impostazione tecnico-scientifica e vocazione archivistica. Il testo è riportato integralmente, senza sintesi o compressioni redazionali, così da preservare continuità argomentativa, dettaglio implementativo e densità documentaria.

Indice generale

Sezione	Titolo
Sezione I	Introduzione
Sezione II	Inquadramento Teorico e Stato dell'Arte
Sezione III	Architettura del Sistema
Sezione IV	Il Motore Testuale e il Sistema di Parsing
Sezione V	Il Sistema delle Stanze e la Macchina a Stati
Sezione VI	Il Sistema degli Oggetti e il Design degli Enigmi
Sezione VII	Il Sistema di Traduzione Progressiva
Sezione VIII	La Sintesi Audio Procedurale
Sezione IX	L'Interfaccia Utente e il Layer di Persistenza
Sezione X	Il Processo di Sviluppo e lo Sviluppo Assistito da IA
Sezione XI	Analisi Critica, Metriche e Limitazioni
Sezione XII	Conclusioni e Sviluppi Futuri
Riferimenti	Riferimenti Bibliografici e Sitografici
Appendice A	Vocabolario Completo dei Comandi
Appendice B	Struttura del Progetto e Metriche di Codice

Sezione I — Introduzione

1.1 Motivazioni e Genesi del Progetto

Il Relitto Silente nasce dall'intenzione di dimostrare che il genere dell'avventura testuale — storicamente confinato ai margini della cultura videoludica contemporanea, considerato da molti un reperto museale dell'era pre-grafica — è ancora in grado di produrre esperienze narrative di qualità, purché affrontato con la stessa serietà tecnica e attenzione progettuale riservata a qualsiasi altro tipo di software creativo.

Il progetto è frutto del lavoro di un singolo sviluppatore e nasce da una doppia motivazione. La prima è affettiva: la volontà di ricreare e tributare l'esperienza del terminale a fosfori verdi, quell'interfaccia austera e immediata che negli anni Ottanta ha rappresentato per molti la prima porta di accesso a un mondo narrativo interattivo. La seconda è tecnica: la curiosità di esplorare fino a dove possono spingersi le tecnologie web moderne — React, TypeScript, Web Audio API — quando vengono applicate non a un'interfaccia commerciale, ma a un medium che pone al centro la parola scritta e l'immaginazione del giocatore.

Il risultato è un'applicazione desktop standalone per Windows, distribuita come file .exe autonomo senza dipendenze esterne, che integra un motore narrativo testuale, una sintesi audio procedurale in tempo reale e un'interfaccia grafica che emula fedelmente l'aspetto di un terminale a raggi catodici.

1.2 Obiettivi del Documento

Il presente documento si propone i seguenti obiettivi:

1. **Documentare l'architettura tecnica** del progetto nella sua interezza, fornendo una descrizione precisa di ogni componente software, delle sue responsabilità e delle interfacce che lo connettono al resto del sistema.
2. **Analizzare le scelte di design** che hanno guidato lo sviluppo, spiegando non soltanto *come* funziona il sistema, ma *perché* è stato progettato in un determinato modo e quali alternative sono state valutate e scartate.
3. **Descrivere il processo di sviluppo**, con particolare attenzione al ruolo svolto dai modelli di linguaggio di grandi dimensioni nella produzione del codice e dei testi narrativi, e alle riflessioni metodologiche che emergono da questa modalità di lavoro.
4. **Fornire una base di riferimento** per chiunque intenda comprendere, estendere o riprendere il progetto in futuro, con una granularità di dettaglio sufficiente a permettere la navigazione del codice sorgente senza ambiguità.

1.3 Struttura del Documento

Il documento è organizzato nelle seguenti sezioni principali:

La **Sezione II** fornisce l'inquadramento teorico, descrivendo la storia del genere dell'avventura testuale, le sue caratteristiche fondamentali e lo stato dell'arte della ricerca in questo dominio.

La **Sezione III** presenta l'architettura generale del sistema: stack tecnologico, struttura del repository, dipendenze e pipeline di build.

La **Sezione IV** analizza in profondità il motore testuale: la funzione di normalizzazione dell'input, il sistema di matching dei comandi, il meccanismo di fuzzy matching e il sistema di suggerimenti contestuali.

La **Sezione V** descrive il modello computazionale delle stanze: come sono strutturate, come le loro descrizioni cambiano in funzione dello stato, e come il sistema gestisce la transizione tra ambienti.

La **Sezione VI** esamina il sistema degli oggetti e degli inventari, con una digressione sul design degli enigmi e sulla filosofia che li governa.

La **Sezione VII** descrive il sistema di traduzione progressiva della lingua aliena K'tharr, una meccanica narrativa originale che intreccia la progressione del gioco con la decodifica linguistica.

La **Sezione VIII** analizza il sottosistema audio: la sintesi procedurale di ambience per cinque profili sonori distinti e il sistema di effetti sonori in tempo reale.

La **Sezione IX** tratta l'interfaccia utente — l'emulazione CRT, l'architettura dei componenti React, il meccanismo di scaling e il layer di persistenza su filesystem tramite IPC Electron.

La **Sezione X** descrive il processo di sviluppo, con un'analisi critica dell'uso dei modelli di linguaggio come strumenti di collaborazione tecnica e creativa.

Le **Sezioni XI e XII** presentano rispettivamente un'analisi critica del progetto — metriche, limitazioni, punti aperti — e le conclusioni con una prospettiva sugli sviluppi futuri.

Le **Appendici A e B** forniscono il vocabolario completo dei comandi riconosciuti dal parser e la struttura dettagliata del repository con le metriche di codice.

1.4 Convenzioni del Documento

Nel corso del testo vengono adottate le seguenti convenzioni tipografiche e redazionali:

- I **nomi propri del gioco** (stanze, oggetti, personaggi, comandi) sono riportati in monospace quando citati nella loro forma tecnica esatta, o in *corsivo* quando citati nel loro contesto narrativo.
- I **listati di codice** sono riportati in blocchi monospace con didascalia che ne indica la provenienza nel repository (nome file e numero di riga approssimativo).
- Le **note tecniche a margine** sono inserite in riquadri separati preceduti dalla dicitura *Nota tecnica:* e contengono approfondimenti non strettamente necessari alla comprensione del flusso principale.

- I **riferimenti incrociati** tra sezioni sono indicati esplicitamente con il numero di sezione tra parentesi, ad esempio: *(cfr. Sezione IV.3)*.
- Le **metriche di codice** (linee di codice, numero di funzioni, complessità ciclomatica) si riferiscono sempre alla versione 1.2.5 del progetto, che costituisce il punto di riferimento del presente documento.

1.5 Perimetro e Limitazioni del Documento

Il documento si concentra sul codice sorgente e sull'architettura del software. Non tratta in dettaglio:

- Il **contenuto narrativo** del gioco (testi delle stanze, echi temporali, dialoghi): questi sono documentati nel Manuale Esteso e nella guida alla soluzione presenti nella cartella docs/giocatore/.
- Le **procedure di distribuzione** e packaging (firme digitali, distribuzione tramite piattaforme di terze parti).
- Le **scelte estetiche** dell'interfaccia visiva oltre la loro dimensione tecnica.

«Un progetto che parte come esperimento tecnico e si trasforma, nel corso dei mesi, in qualcosa di più personale: un tentativo di capire cosa può fare la narrazione testuale che nessun altro medium può fare allo stesso modo.»

— dal Capitolo XI del Manuale Esteso

Sezione II — Inquadramento Teorico e Stato dell'Arte

2.1 Le Avventure Testuali: Origine e Definizione

Le avventure testuali — denominate in inglese *text adventures* o, con termine più ampio e teoricamente più preciso, *interactive fiction* (IF) — rappresentano il primo genere videoludico della storia. La loro origine è databile con precisione: nel 1975, Will Crowther, ingegnere informatico presso BBN Technologies e speleologo dilettante, sviluppò *ADVENTURE* (poi noto come *Colossal Cave Adventure*) su un mainframe PDP-10. Il programma simulava l'esplorazione di un sistema cavernicolo ispirato alle Mammoth Cave del Kentucky, consentendo all'utente di navigare l'ambiente digitando comandi in linguaggio naturale ristretto.

ADVENTURE era strutturato attorno a due elementi fondamentali che avrebbero definito il genere per i decenni successivi: una **rappresentazione spaziale discreta** del mondo (un grafo di «stanze» connesse da transizioni direzionali) e un **parser testuale** incaricato di interpretare i comandi dell'utente e tradurli in operazioni sul modello di mondo.

La versione estesa del programma, sviluppata da Don Woods nel 1977, aggiunse elementi fantastici e narrativi che ne amplificarono enormemente la diffusione. Da quel momento, il genere si sviluppò lungo due direttrici parallele: la **direttrice commerciale**, rappresentata da aziende come Infocom (fondata al MIT nel 1979) e Sierra On-Line, e la **direttrice hobbistica**, alimentata da una comunità di appassionati che sviluppavano e distribuivano titoli amatoriali.

2.2 L'Era d'Oro: Infocom e la Raffinatezza del Parser

La produzione di Infocom negli anni Ottanta rappresenta il punto più alto raggiunto dal genere nella sua forma classica. Titoli come *Zork* (1980), *The Hitchhiker's Guide to the Galaxy* (1984), scritto in collaborazione con Douglas Adams, e *A Mind Forever Voyaging* (1985) dimostrarono che un'avventura testuale poteva essere al contempo un prodotto di intrattenimento sofisticato e un oggetto letterario di valore.

L'elemento tecnico che distingue la produzione Infocom è la **Z-machine**: una macchina virtuale sviluppata internamente che consentiva di scrivere i giochi in un linguaggio ad alto livello (ZIL, Zork Implementation Language, un dialetto di LISP) e di eseguirli su qualsiasi piattaforma dotata di un interprete Z-machine. Questa architettura anticipò di circa quindici anni il concetto di runtime portabile poi popolarizzato da Java.

Il parser di Infocom era notevolmente più sofisticato dei suoi predecessori: comprendeva articoli, preposizioni complesse, comandi con oggetti multipli («metti la lanterna e il libro nello zaino»), e produceva messaggi di errore informativi che guidavano il giocatore verso la formulazione corretta del comando. Molti di questi principi sono stati ripresi e adattati nel parser di *Il Relitto Silente (cfr. Sezione IV)*.

2.3 Il Declino Commerciale e la Rinascita Amatoriale

Con la diffusione dei personal computer grafici nella seconda metà degli anni Ottanta — in particolare con il lancio del Commodore Amiga nel 1985 e della serie Macintosh — il pubblico si spostò progressivamente verso titoli con interfacce visive. Le avventure testuali persero rilevanza commerciale; Infocom fu acquisita da Activision nel 1986 e chiuse definitivamente nel 1989.

Tuttavia, il genere non morì. Nel 1987, Michael J. Roberts sviluppò TADS (Text Adventure Development System), il primo strumento dedicato specificamente alla creazione di avventure testuali. Nel 1993, Graham Nelson creò Inform, un linguaggio di programmazione dichiarativo che consentiva di descrivere il modello di mondo in modo quasi prosastico. Attorno a questi strumenti si formò una comunità internazionale — l'Interactive Fiction community — che ha continuato a produrre, distribuire e teorizzare il genere fino ai giorni nostri.

La Twine, nata nel 2009, rappresenta la più recente evoluzione significativa degli strumenti per l'IF: spostando il focus dal parser al sistema di scelte ipertestuali, ha reso il genere accessibile a un pubblico di autori senza background tecnico, contribuendo a un'ulteriore espansione della produzione amatoriale.

2.4 Il Parser come Interfaccia: Riflessioni Teoriche

Il parser testuale occupa una posizione peculiare nel panorama delle interfacce uomo–macchina. A differenza di un'interfaccia grafica — che presenta all'utente le opzioni disponibili sotto forma visiva — il parser impone all'utente di **formulare autonomamente** la propria intenzione in linguaggio (parzialmente) naturale. Questo requisito è stato storicamente una fonte di frustrazione per i giocatori meno esperti, ma è anche, secondo molti teorici dell'IF, una delle caratteristiche più preziose del medium.

Nick Montfort, nel suo *Twisty Little Passages: An Approach to Interactive Fiction* (2003), sostiene che il parser non è semplicemente un'interfaccia per il gioco ma una delle sue componenti narrative fondamentali: la negoziazione linguistica tra giocatore e sistema è essa stessa una forma di esplorazione del mondo finzionale. Il giocatore non soltanto naviga lo spazio diegetico; lo costruisce attraverso il linguaggio.

Emily Short — tra le autrici contemporanee di IF più rilevanti — ha approfondito nel suo lavoro critico la distinzione tra parser «opaco» (che punisce l'utente per formulazioni imprecise) e parser «trasparente» (che comprende l'intento anche quando l'espressione è imperfetta). *Il Relitto Silente* adotta esplicitamente la seconda filosofia *(cfr. Sezione IV.1)*.

2.5 L'Interactive Fiction in Lingua Italiana

La produzione italiana di avventure testuali, sebbene meno documentata rispetto a quella anglofona, ha una storia significativa. Negli anni Ottanta, aziende come Activision Italy e la Mastervision pubblicarono adattamenti di titoli internazionali e, in alcuni casi, produzioni originali. La comunità amatoriale italiana ha prodotto nel corso dei decenni un corpus di titoli scritti in Inform, TADS e Twine, raccolti parzialmente nel repository dell'Italian Interactive Fiction community.

La scelta di sviluppare *Il Relitto Silente* in italiano risponde a motivazioni sia narrative sia linguistiche. Dal punto di vista narrativo, la lingua italiana consente una densità semantica e una musicalità testuale difficilmente replicabili in altre lingue; dal punto di vista tecnico, il trattamento della lingua italiana pone sfide specifiche al parser — gestione degli articoli maschili e femminili, singolari e plurali, elisioni, preposizioni articolate — che hanno richiesto soluzioni dedicate nella fase di normalizzazione dell'input *(cfr. Sezione IV.2)*.

2.6 Tecnologie Web per lo Sviluppo di Giochi Testuali

La diffusione di JavaScript come linguaggio universale del web ha prodotto un ampio ecosistema di strumenti per lo sviluppo di IF nel browser. Twine genera HTML statico; Ink (sviluppato da Inkle Studios) è un linguaggio compilabile in JavaScript; la Z-machine è stata portata in JavaScript da diversi sviluppatori indipendenti.

Il Relitto Silente non utilizza nessuno di questi framework preesistenti. Il motore narrativo è sviluppato *ex novo* in TypeScript, senza dipendenze da librerie specializzate per l'IF. Questa scelta — non priva di costi in termini di tempo di sviluppo — ha consentito un controllo totale sull'architettura del sistema e ha permesso l'integrazione con componenti non standard come il sottosistema audio procedurale *(cfr. Sezione VIII)* e il layer di persistenza su filesystem *(cfr. Sezione IX.4)*.

L'impacchettamento tramite Electron 41 consente la distribuzione come applicazione desktop standalone, eliminando le dipendenze da ambienti browser specifici e garantendo prestazioni prevedibili su Windows 10 e 11.

2.7 Sviluppo Assistito da Modelli di Linguaggio

Lo sviluppo di *Il Relitto Silente* si inserisce nel contesto più ampio della diffusione dei modelli di linguaggio di grandi dimensioni (LLM) come strumenti di assistenza alla programmazione. Strumenti come GitHub Copilot (2021), Claude Code (2025) e altri hanno cominciato a ridefinire il processo di sviluppo software, con implicazioni metodologiche ancora in parte esplorate.

Il progetto è stato sviluppato in due fasi distinte. La prima fase ha utilizzato Google AI Studio App Builder come ambiente di prototipazione rapida. La seconda — quella che ha portato il progetto alla sua forma definitiva — è stata condotta in collaborazione con Claude Sonnet 4.6 tramite Claude Code, lo strumento di sviluppo assistito di Anthropic.

L'analisi di questa collaborazione e delle sue implicazioni metodologiche è trattata in dettaglio nella Sezione X. In questa sede è sufficiente osservare che il modello adottato è rigorosamente quello della **collaborazione tecnica consapevole**: le scelte architetturali, le decisioni di design e la direzione narrativa sono rimaste di esclusiva responsabilità dell'autore umano, mentre il modello ha fornito assistenza nella generazione, revisione e ottimizzazione del codice, nella stesura dei testi narrativi e nella risoluzione di problemi tecnici specifici.

2.8 Posizionamento del Progetto

Il Relitto Silente si posiziona in un punto di intersezione tra più tradizioni:

- La **tradizione classica del parser adventure** di stampo Infocom, con la sua enfasi sulla precisione narrativa e sul design degli enigmi coerente con il mondo finzionale.
- La **tradizione dell'IF letteraria contemporanea**, con l'attenzione alla qualità del testo, alla costruzione dell'atmosfera e all'esperienza del giocatore come lettore.
- La **tradizione tecnica del software indipendente**, con la scelta di costruire un motore proprietario piuttosto che adottare framework preesistenti, e di distribuire il prodotto come applicazione desktop autonoma.
- Il **contesto emergente dello sviluppo assistito da IA**, come caso di studio di una collaborazione tra autore umano e modello di linguaggio nella produzione di software creativo complesso.

Nessuna di queste tradizioni è sufficiente, da sola, a caratterizzare il progetto. È la loro combinazione — e le tensioni che essa genera — a definire la natura tecnica e culturale de *Il Relitto Silente*.

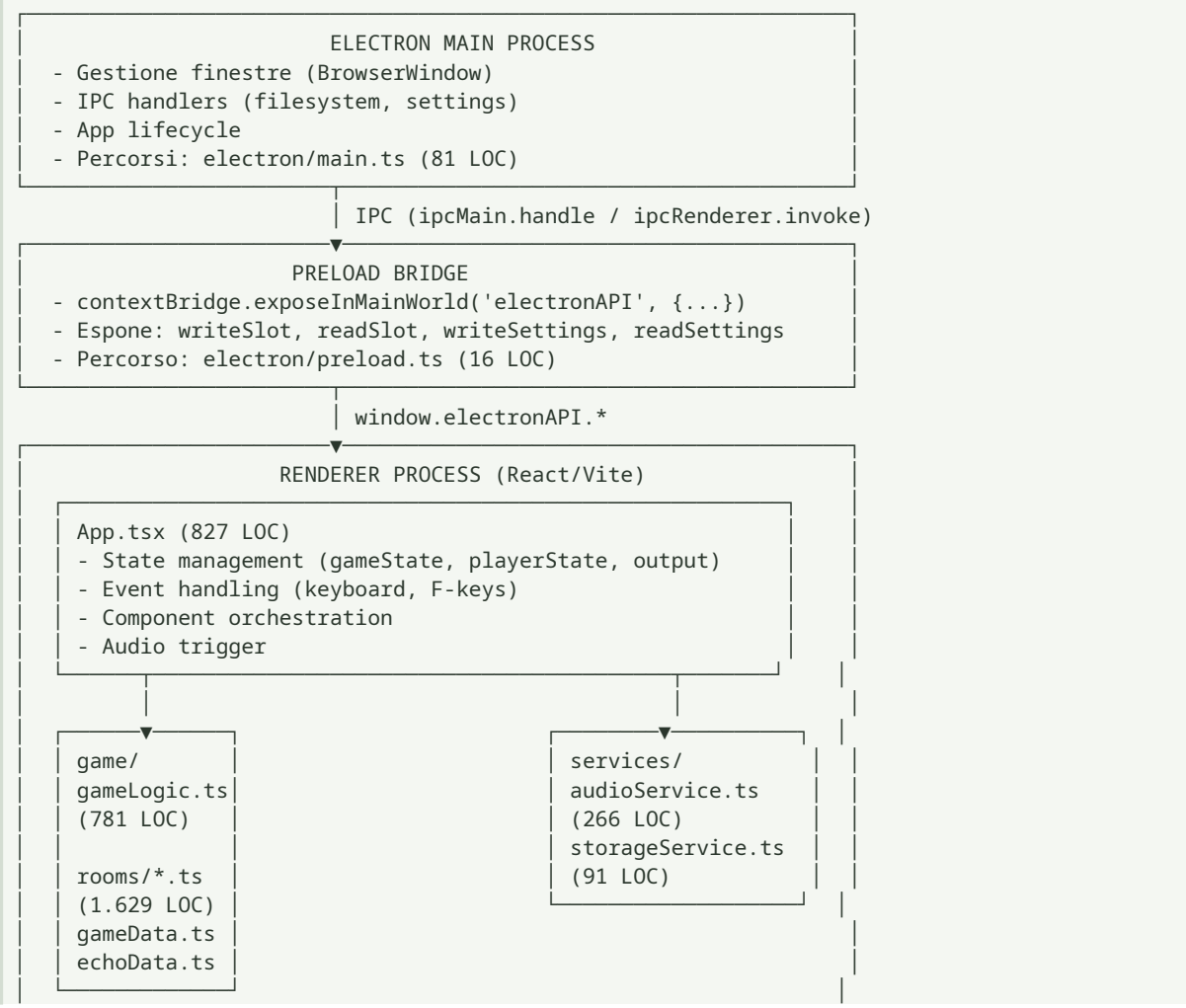
Sezione III — Architettura del Sistema

3.1 Visione d'Insieme

Il Relitto Silente è un'applicazione desktop sviluppata con tecnologie web e distribuita tramite il framework Electron. L'architettura generale del sistema segue il modello a **tre livelli** tipico delle applicazioni Electron moderne:

1. **Main Process** (Node.js): gestisce l'accesso al filesystem, le finestre dell'applicazione e la comunicazione IPC con il renderer.
2. **Renderer Process** (Chromium): ospita l'intera interfaccia utente e la logica di gioco, sviluppate in React e TypeScript.
3. **Preload Bridge**: uno script di sicurezza che espone al renderer un'API controllata per comunicare con il main process, senza che il renderer abbia accesso diretto alle API di Node.js.

Questa separazione è una scelta di sicurezza obbligata nelle versioni moderne di Electron: il renderer gira in un contesto di sicurezza analoga a quella di una pagina web (con contextIsolation: true e nodeIntegration: false), il che significa che non può accedere direttamente al filesystem o ad altre API native. Tutta la I/O avviene tramite canali IPC gestiti dal main process.



```
│                                                          │
│   ┌──────────────────────────────────────────────────┐   │
│   │ components/ (1.401 LOC totali)                   │   │
│   │ TerminalOutput, CommandLine, SaveLoadOverlay,    │   │
│   │ PauseOverlay, AudioSettingsOverlay, IntroScreen, │   │
│   │ BootScreen, StartScreen, GameOverScreen, ErrorBoundary │
│   └──────────────────────────────────────────────────┘   │
└──────────────────────────────────────────────────────────┘
```

3.2 Stack Tecnologico

La Tab. 3.1 riporta le dipendenze principali del progetto alla versione 1.2.5, con le versioni esatte rilevate dal file package.json.

Tab. 3.1 — Dipendenze principali del progetto

Libreria / Framework	Versione	Ruolo
React	19.2.4	Libreria UI, gestione componenti e stato
TypeScript	5.9.3	Linguaggio principale (superset di JavaScript)
Vite	8.0.1	Bundler, dev server con HMR
Electron	41.0.3	Wrapper desktop (Chromium + Node.js)
electron-builder	26.8.1	Packaging e distribuzione degli installer
Tailwind CSS	4.2.2	Framework CSS utility-first
@tailwindcss/vite	4.2.2	Integrazione Vite per Tailwind CSS
lodash.clonedeep	4.5.0	Deep clone dello stato di gioco
@fontsource/press-start-2p	5.2.7	Font pixel art per l'interfaccia retro
concurrently	9.2.1	Esecuzione parallela dei processi in sviluppo
cross-env	10.1.0	Variabili d'ambiente cross-platform

La scelta di React 19 (versione più recente disponibile al momento dello sviluppo) consente di sfruttare le ultime ottimizzazioni del motore di rendering, in particolare per quanto riguarda la gestione delle liste di output del terminale, che possono crescere significativamente durante una sessione di gioco prolungata.

TypeScript 5.9 con la configurazione strict mode garantisce il massimo livello di type safety disponibile nel linguaggio. La scelta di non usare Redux o altre librerie di state management esterne è una decisione deliberata: la gestione dello stato tramite useState di React è sufficiente per le esigenze del progetto e riduce la complessità architetturale.

lodash.clonedeep è l'unica dipendenza di runtime non riconducibile al framework UI. Il suo utilizzo è motivato dalla necessità di clonare profondamente il PlayerState ad ogni esecuzione di un comando *(cfr. Sezione IV.4)*, garantendo l'immutabilità dello stato e la possibilità di implementare rollback in futuro.

3.3 Struttura del Repository

Il repository è organizzato nella struttura riportata in Tab. 3.2. I percorsi sono relativi alla radice del progetto.

Tab. 3.2 — Struttura principale del repository

Percorso	Tipo	Descrizione
game/gameLogic.ts	Sorgente	Motore narrativo principale (781 LOC)
game/gameData.ts	Sorgente	Aggregatore delle definizioni di stanza (42 LOC)
game/echoData.ts	Sorgente	Testi e mapping degli echi temporali (36 LOC)
game/rooms/	Directory	15 file TypeScript, uno per stanza (1.629 LOC totali)
App.tsx	Sorgente	Componente React principale (827 LOC)
types.ts	Sorgente	Definizioni TypeScript centralizzate (72 LOC)
services/audioService.ts	Sorgente	Sintesi audio procedurale (266 LOC)
services/storageService.ts	Sorgente	Layer di persistenza duale (91 LOC)
components/	Directory	10 componenti React (1.401 LOC totali)
electron/main.ts	Sorgente	Processo principale Electron (81 LOC)
electron/preload.ts	Sorgente	Bridge di sicurezza IPC (16 LOC)
index.css	Stile	Foglio di stile principale (~7.000 LOC)
vite.config.ts	Config	Configurazione Vite (21 LOC)
tsconfig.json	Config	Configurazione TypeScript (30 LOC)
package.json	Config	Configurazione npm e build
docs/	Documentazione	Manuali e guide per il giocatore

Percorso	Tipo	Descrizione
MANUALE-ESTESO/	Documentazione	Manuale dettagliato del gioco
PAPER/	Documentazione	Il presente documento tecnico

3.4 Configurazione TypeScript

La configurazione TypeScript adottata privilegia la type safety e la compatibilità con le funzionalità ES2024. Il file tsconfig.json principale è riportato integralmente:

```
{
  "compilerOptions": {
    "target": "ES2024",
    "module": "ESNext",
    "lib": ["ES2024", "DOM", "DOM.Iterable"],
    "skipLibCheck": true,
    "types": ["node", "vite/client"],
    "moduleResolution": "bundler",
    "isolatedModules": true,
    "moduleDetection": "force",
    "allowJs": true,
    "jsx": "react-jsx",
    "paths": { "@/*": ["./*"] },
    "allowImportingTsExtensions": true,
    "noEmit": true
  }
}
```

L’opzione noEmit: true indica che il compilatore TypeScript non genera output JavaScript: la compilazione è interamente gestita da Vite, che utilizza esbuild internamente per la trascrizione da TypeScript a JavaScript ad alta velocità. TypeScript viene quindi usato esclusivamente come sistema di type checking.

L’opzione jsx: "react-jsx" attiva la trasformazione JSX automatica introdotta in React 17, che elimina la necessità dell’import esplicito di React in ogni file che utilizzi la sintassi JSX.

Il processo Electron ha una configurazione TypeScript separata (electron/tsconfig.json) che punta a CommonJS come formato di modulo, necessario per la compatibilità con Node.js nel main process.

3.5 Configurazione Vite

```
// vite.config.ts
import path from 'path';
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
import tailwindcss from '@tailwindcss/vite';

export default defineConfig({
  server: {
    port: 3000,
    host: '0.0.0.0',
  },
  plugins: [
    tailwindcss(),
    react(),
```

```
  ],
  base: './',
  resolve: {
    alias: {
      '@': path.resolve(__dirname, '.'),
    },
  },
});
```

L'opzione base: './' è fondamentale per il funzionamento corretto in Electron: il renderer carica i file statici dalla directory locale, non da un server remoto, e la base relativa garantisce che i percorsi delle risorse siano risolti correttamente indipendentemente dalla posizione dell'eseguibile sul filesystem dell'utente.

3.6 Pipeline di Build

Lo sviluppo e la distribuzione del progetto sono supportati da tre script npm distinti:

Modalità di sviluppo (npm run electron:dev): avvia il dev server Vite sulla porta 3000 (con Hot Module Replacement attivo) e il processo Electron in parallelo tramite concurrently. Lo script wait-on tcp:3000 garantisce che Electron si avvii soltanto dopo che il dev server Vite è pronto ad accettare connessioni.

Build di produzione (npm run electron:build): esegue in sequenza la build Vite (con ottimizzazioni e minificazione), la compilazione TypeScript del processo Electron, e infine electron-builder --win per generare l'installer NSIS per Windows. Il risultato è un singolo file .exe di dimensioni contenute che include l'intero runtime Chromium/Node.js, le risorse dell'applicazione e il codice del gioco.

Build solo web (npm run build): genera una versione web del gioco nella directory dist/, utilizzabile in un browser standard senza Electron. In questa modalità, il layer di persistenza ricade automaticamente su localStorage *(cfr. Sezione IX.4)*.

3.7 Configurazione Electron Builder

La distribuzione del pacchetto è configurata direttamente nel package.json tramite la chiave "build":

```
{
  "appId": "com.relittosilente.adventure",
  "productName": "Il Relitto Silente",
  "files": ["dist/**/*", "dist-electron/**/*"],
  "extraFiles": [
    { "from": "docs/giocatore/LEGGIMI.txt", "to": "LEGGIMI.txt" },
    { "from": "docs/giocatore/SOLUZIONE.md", "to": "SOLUZIONE.md" }
  ],
  "directories": { "output": "release" },
  "win": {
    "target": "nsis",
    "icon": "public/app-icon.png"
  }
}
```

La direttiva extraFiles copia i file di documentazione per il giocatore (LEGGIMI e guida alla soluzione) direttamente nella radice della directory di installazione, rendendoli accessibili senza dover aprire l'applicazione.

3.8 Il Modello dei Dati Centrale: types.ts

Il file types.ts (72 LOC) è il punto di riferimento unico per le definizioni di tipo condivise tra tutti i moduli. Le sue definizioni più rilevanti sono:

PlayerState: la struttura dati immutabile che rappresenta l'intero stato del gioco in un dato momento. Include la posizione corrente (location), l'inventario (inventory), il dizionario dei flag di avanzamento (flags), la lista degli echi captati (echoes) e le stanze visitate (visitedRooms).

Room: l'interfaccia che ogni definizione di stanza deve implementare. Contiene una funzione description che accetta uno PlayerState e restituisce una stringa (descrizione dinamica), un array commands di handler per i comandi specifici della stanza, e un array opzionale items con gli oggetti presenti.

Item: la struttura di un oggetto interattivo. Contiene identificatori e sinonimi, descrizioni per i comandi ESAMINA e ANALIZZA, flag di raccoglibilità, e handler opzionali per le azioni onUse, onAnalyze e onCombine.

GameResponse: la struttura di risposta restituita da processCommand ad ogni esecuzione. Include la descrizione testuale, il tipo di evento (usato per selezionare l'effetto sonoro appropriato), flag per la gestione del game over e del testo multi-pagina.

OutputLine: l'unione discriminata dei tre tipi di riga di output visualizzabili nel terminale: 'text' (testo statico), 'html' (markup HTML generato internamente) e 'typewriter' (testo animato carattere per carattere).

L'utilizzo di un file di tipi centralizzato — invece di definire i tipi localmente nei file che li utilizzano — è una scelta architetturale che facilita la navigazione del codice e garantisce la coerenza delle interfacce tra moduli.

Sezione IV — Il Motore Testuale e il Sistema di Parsing

4.1 Filosofia del Parser

Il parser è il fulcro di qualsiasi avventura testuale. È l'interfaccia attraverso cui il giocatore esprime la propria intenzione, e la qualità del suo funzionamento determina in modo diretto la qualità dell'esperienza. Un parser che respinge troppo facilmente i comandi dell'utente — che richiede una formulazione esatta, non tollerando varianti o approssimazioni — è percepito come un ostacolo, non come uno strumento. Questa percezione di «sbagliare le parole» pur sapendo cosa fare è stata storicamente una delle principali fonti di frustrazione nei giochi del genere.

Il Relitto Silente adotta una filosofia opposta. Il parser è progettato come **alleato del giocatore**: il suo compito non è filtrare i comandi non conformi a una lista predefinita, ma estrarre l'intenzione semantica da qualsiasi formulazione ragionevolmente vicina all'italiano. Questo obiettivo è perseguito attraverso un sistema di normalizzazione dell'input in due stadi: una fase di **pre-processing linguistico** (rimozione di articoli, pronomi, preposizioni superflue, normalizzazione di accenti e maiuscole) e una fase di **matching semantico** (corrispondenza dell'input normalizzato con i comandi riconosciuti, con supporto per sinonimi e abbreviazioni).

4.2 La Funzione normalizeCommand

Il cuore del sistema di parsing è la funzione normalizeCommand, definita in game/gameLogic.ts. La funzione riceve in input la stringa digitata dall'utente e restituisce una forma normalizzata utilizzata per il matching con i comandi riconosciuti.

```
// game/gameLogic.ts — normalizeCommand
export const normalizeCommand = (command: string): string => {
    let normalized = command
        .toLowerCase()
        .trim()
        // Rimozione accenti (NFD decomposition + strip diacritic marks)
        .normalize("NFD").replace(/[\u0300-\u036f]/g, "")
        // Articoli determinativi e indeterminativi
        .replace(/\b(il|lo|la|i|gli|le|un|uno|una|un'|l')\s+/g, '')
        // Dimostrativi
        .replace(/\b(quel|quello|quella|quei|quegli|quelle|questo|questa|questi|queste)
\s+/g, '')
        // Normalizzazione preposizioni
        .replace(/\b(con|sopra)\b/g, 'su')
        // Collasso spazi multipli
        .replace(/\s+/g, ' ')
        // Normalizzazione preposizioni articolate in comandi di entrata
        .replace(/^(entra|vai) (nel|nella|nell'|nello|nelle|in) /, '$1 ');

    // Abbreviazioni di movimento
    if (normalized === 'n') return 'nord';
    if (normalized === 's') return 'sud';
    if (normalized === 'e') return 'est';
    if (normalized === 'o' || normalized === 'w') return 'ovest';
```

```
    if (normalized === 'u') return 'alto';
    if (normalized === 'd' || normalized === 'giu') return 'basso';

    // Abbreviazioni di azione
    if (normalized === 'l') return 'guarda';
    if (normalized === 'i' || normalized === 'inv') return 'inventario';

    // Alias di verbo
    if (normalized.startsWith('indossa ')) return normalized.replace(/^indossa /, 'usa ');
    if (normalized.startsWith('leggi '))   return normalized.replace(/^leggi /',   'analizza
');
    if (normalized.startsWith('x '))       return normalized.replace(/^x /,        'esamina
');

    return normalized;
};
```

La sequenza delle operazioni di normalizzazione è significativa:

1. Lowercase e trim: operazioni preliminari che eliminano la dipendenza dalla capitalizzazione e dagli spazi iniziali/finali. Il parser è completamente insensibile alle maiuscole: GUARDA, guarda e Guarda producono lo stesso risultato.

2. Normalizzazione Unicode NFD: la decomposizione NFD (Normalization Form Canonical Decomposition) separa i caratteri composti nelle loro componenti base; la successiva sostituzione dei diacritici rimuove gli accenti. Questo consente di trattare analizza e analízzà come equivalenti. In pratica, questo significa che il parser tollera sia l'italiano con accenti corretti sia l'italiano digitato senza caratteri accentati — molto comune su tastiere configurate per layout non italiani.

3. Rimozione articoli e dimostrativi: una delle operazioni più importanti per la naturalità del parser. La frase usa il disco sull'altare viene ridotta a usa disco su altare; esamina quella maledetta teca diventa esamina teca. Questo elimina la necessità che il giocatore ricordi se un oggetto richiede l'articolo determinativo o indeterminativo, maschile o femminile.

4. Normalizzazione preposizioni: le preposizioni con e sopra vengono convertite nella forma canonica su (usata internamente dai comandi di combinazione). Questo consente di scrivere usa la chiave con la serratura in alternativa alla forma usa chiave su serratura.

5. Abbreviazioni direzionali: le singole lettere n, s, e, o, u, d (e le varianti w per ovest e giu per basso) vengono espanse nelle parole complete. Questo consente la navigazione rapida tipica delle avventure testuali classiche.

6. Alias di verbo: tre sostituzioni di prefisso trasformano verbi non canonici nelle loro forme canoniche: - indossa X → usa X (indossare un oggetto è semanticamente equivalente ad usarlo nella logica del gioco) - leggi X → analizza X (leggere un'iscrizione attiva il sistema di analisi) - x X → esamina X (abbreviazione tradizionale di EXAMINE nell'IF anglosassone)

4.3 Il Sistema di Matching dei Comandi

Una volta normalizzato l'input, il sistema cerca una corrispondenza attraverso due livelli di matching:

Livello 1 — Comandi globali: un insieme di comandi riconosciuti in qualsiasi stanza, indipendentemente dalla posizione del giocatore. Questi includono inventario, mappa, nota, aiuto, aspetta, salva, carica, audio, impostazioni, pulisci, echi, e i comandi di navigazione (nord, sud, ecc.).

Livello 2 — Comandi di stanza: ogni Room definisce un array di oggetti Command, ciascuno con una regex e un handler. I comandi di stanza vengono verificati dopo quelli globali e gestiscono le azioni specifiche dell'ambiente corrente.

Livello 3 — Sistema degli oggetti: se nessun comando globale o di stanza corrisponde, il sistema verifica se l'input contiene un riferimento a un oggetto dell'inventario o della stanza corrente, e instrada la richiesta all'handler appropriato dell'oggetto.

Livello 4 — Fuzzy matching e fallback: se nessun livello precedente produce una corrispondenza, il sistema tenta di identificare il verbo dell'input tramite distanza di Levenshtein e propone una correzione («Forse intendevi…?») prima di restituire il messaggio di errore.

4.4 La Funzione processCommand e l'Immutabilità dello Stato

La funzione principale del motore narrativo è processCommand, che orchestra l'intero ciclo di elaborazione di un comando:

```
// game/gameLogic.ts — processCommand (struttura semplificata)
export const processCommand = (
    command: string,
    currentState: PlayerState
): { response: GameResponse; newState: PlayerState } => {

    // 1. Clone profondo per garantire l'immutabilità
    const state = cloneDeep(currentState);

    // 2. Normalizzazione input
    const normalized = normalizeCommand(command);

    // 3. Matching e dispatch
    // ... (comandi globali, comandi di stanza, sistema oggetti)

    return { response, newState: state };
};
```

Il clone profondo iniziale è un elemento architetturale fondamentale. processCommand riceve lo stato corrente e deve restituire un nuovo stato che riflette le modifiche causate dal comando. Operare su una copia profonda garantisce che:

- Il componente React che chiama processCommand non vede modifiche «a sorpresa» allo stato attuale durante l'elaborazione.
- Le mutazioni che il sistema di stanze e oggetti applica allo stato (aggiunta di oggetti all'inventario, impostazione di flag, cambio di posizione) avvengono su una copia isolata.
- Il nuovo stato restituito da processCommand può essere confrontato con lo stato precedente per rilevare cambiamenti (ad esempio, per decidere se attivare l'autosave).

La funzione cloneDeep di lodash è preferita all'alternativa JSON.parse(JSON.stringify(...)) per ragioni di robustezza: gestisce correttamente tipi come undefined, funzioni (non presenti nello stato di gioco, ma presenti in futuro se lo stato fosse esteso) e riferimenti circolari.

4.5 La Cache delle Espressioni Regolari

Il sistema di matching dei comandi fa ampio uso di espressioni regolari per identificare pattern nei comandi normalizzati. La compilazione di un'espressione regolare è un'operazione computazionalmente non trascurabile; eseguirla ad ogni comando sarebbe inefficiente.

Il sistema adotta una cache basata su Map:

```
// game/gameLogic.ts — regex cache
const regexCache = new Map<string, RegExp>();

function getCachedRegex(pattern: string): RegExp {
    if (!regexCache.has(pattern)) {
        regexCache.set(pattern, new RegExp(pattern, 'i'));
    }
    return regexCache.get(pattern)!;
}
```

Le espressioni regolari vengono compilate soltanto alla prima esecuzione e poi recuperate dalla cache per le invocazioni successive. Considerando che processCommand viene chiamata ad ogni comando dell'utente — potenzialmente centinaia di volte durante una sessione di gioco — questa ottimizzazione è rilevante per la reattività percepita dell'interfaccia.

4.6 La Cache degli Oggetti Globali

Analogamente, il sistema degli oggetti mantiene una cache piatta di tutti gli oggetti presenti nel gioco:

```
// game/gameLogic.ts — items global cache
let _allItemsCache: Item[] | null = null;

function getAllItems(): Item[] {
    if (!_allItemsCache) {
        _allItemsCache = getAllRooms().flatMap(room => room.items ?? []);
    }
    return _allItemsCache;
}
```

Questa cache è necessaria per il matching degli oggetti dell'inventario: quando il giocatore usa un oggetto su un bersaglio, il sistema deve verificare sia gli oggetti della stanza corrente sia quelli dell'inventario (e, per alcune operazioni, gli oggetti fissi di qualsiasi stanza). Il flatmap di tutte le stanze viene eseguito una sola volta.

4.7 Il Fuzzy Matching: Distanza di Levenshtein

Quando l'input dell'utente non corrisponde a nessun comando riconosciuto, il sistema calcola la **distanza di Levenshtein** tra il verbo del comando e ciascuno dei verbi nel vocabolario noto (KNOWN_VERBS):

```
// game/gameLogic.ts — distanza di Levenshtein
function levenshtein(a: string, b: string): number {
    const m = a.length;
    const n = b.length;
    const dp: number[][] = Array.from({ length: m + 1 }, (_, i) =>
        Array.from({ length: n + 1 }, (_, j) => (i === 0 ? j : j === 0 ? i : 0))
    );
    for (let i = 1; i <= m; i++) {
        for (let j = 1; j <= n; j++) {
            dp[i][j] = a[i-1] === b[j-1]
                ? dp[i-1][j-1]
                : 1 + Math.min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]);
        }
    }
    return dp[m][n];
}
```

Se la distanza minima trovata è ≤ 2 (ovvero il comando digitato differisce dal comando più simile per non più di due inserimenti, cancellazioni o sostituzioni di carattere), il sistema propone una correzione:

```
> esamnia disco
Non ho capito. Forse intendevi "esamina"?
```

Questo meccanismo riduce significativamente la frustrazione causata da errori di battitura, uno dei problemi più frequenti nelle sessioni di gioco prolungate.

4.8 Il Sistema HINT: Suggerimenti Contestuali

La funzione getContextualHint implementa un sistema di suggerimenti che analizza lo stato corrente del gioco e restituisce il **passo logico successivo più rilevante**. A differenza di un sistema di suggerimenti generico, getContextualHint conosce la posizione del giocatore, il contenuto del suo inventario e i flag di avanzamento, e produce suggerimenti mirati.

La funzione è strutturata come una gerarchia di condizioni, valutate dall'alto verso il basso. L'ordine della gerarchia riflette la struttura narrativa del gioco: le condizioni più avanzate (finale, terza chiave) sono verificate per prime; le condizioni più iniziali (prima esplorazione, oggetti iniziali) sono verificate per ultime come fallback.

```
// game/gameLogic.ts — getContextualHint (struttura semplificata)
export function getContextualHint(state: PlayerState): string {
    const { location, inventory, flags } = state;

    // Fase finale
    if (location === 'Santuario Centrale') {
        return 'Hai raggiunto la fine del tuo viaggio. Parla con la figura luminosa che ti
attende.';
    }

    // Ha tutte e tre le Chiavi Simboliche
    if (hasAllThreeKeys(inventory)) {
        return suggestFinalDoor(location, flags);
```

```
    }

    // Manca il Nucleo di Memoria
    if (!inventory.includes('Nucleo di Memoria')) {
        return suggestNucleoPath(location, inventory, flags);
    }

    // Manca la Stele del Ricordo
    if (!inventory.includes('Stele del Ricordo')) {
        return suggestStelePath(location, inventory, flags);
    }

    // Manca il Seme della Continuità
    if (!inventory.includes('Seme della Continuità')) {
        return suggestSemePath(location, inventory, flags);
    }

    // Fase iniziale
    return suggestInitialPhase(location, inventory, flags);
}
```

Ogni funzione di suggerimento (suggestNucleoPath, suggestStelePath, ecc.) considera sia la posizione corrente sia lo stato dei flag rilevanti per quella specifica chiave, producendo un suggerimento granulare che indica non la soluzione completa ma il **prossimo passo utile**. La distinzione è importante: il sistema HINT non bypassa l'enigma, ma aiuta il giocatore a identificare dove si trova il blocco.

4.9 La Barra di Progresso della Traduzione

Un elemento visivo ricorrente nell'interfaccia testuale è la barra di avanzamento della traduzione K'tharr, generata dalla funzione:

```
// game/gameLogic.ts
function progressBar(pct: number): string {
    const filled = Math.round(pct / 10);
    return `[ MATRICE TRADUZIONE: ${'▓'.repeat(filled)}${'░'.repeat(10 - filled)} ${pct}
% ]`;
}
```

Questa funzione è un esempio di come il progetto utilizzi deliberatamente gli strumenti del testo puro per creare effetti visivi: non è un componente React, non è CSS, non è JavaScript che manipola il DOM. È una stringa ASCII che comunica un'informazione di stato in modo immediatamente leggibile nel contesto di un terminale. Questa coerenza estetica — mantenere la comunicazione visiva all'interno del medium testuale — è una scelta deliberata che riflette la filosofia del progetto.

4.10 Il Diario di Bordo

La funzione getDiario genera dinamicamente il contenuto del comando NOTA / DIARIO, producendo un resoconto in prima persona dei progressi del giocatore:

- Artefatti analizzati e percentuale di traduzione raggiunta
- Echi temporali captati e loro provenienza
- Scoperte narrative significative

Il diario non è un log di eventi generici ma un **documento narrativo**: i testi sono scritti in prima persona e in un registro coerente con quello del protagonista, mantenendo l'immersione anche durante la consultazione di uno strumento di gestione delle informazioni.

4.11 Le Statistiche di Fine Partita

Al termine del gioco, la funzione getStats calcola un punteggio composito basato su quattro metriche:

- Stanze esplorate (0–15), peso 25%
- Echi captati (0–11), peso 25%
- Percentuale di traduzione completata (0–100%), peso 25%
- Artefatti analizzati (0–7), peso 25%

Il punteggio finale determina uno dei seguenti titoli:

Punteggio	Titolo
≥ 80%	Custode della Memoria
≥ 60%	Archivista Stellare
≥ 40%	Esploratore Curioso
< 40%	Osservatore del Silenzio

Il titolo non ha effetti sul gioco e non sblocca o preclude contenuti: è esclusivamente un indicatore della profondità dell'esplorazione, destinato ai giocatori che desiderano una misura oggettiva del loro livello di completamento.

Sezione V — Il Sistema delle Stanze e la Macchina a Stati

5.1 Il Modello della Stanza

La struttura del mondo in *Il Relitto Silente* segue il modello classico delle avventure testuali: il mondo è rappresentato come un **grafo di ambienti discreti** (stanze), connessi da transizioni direzionali o contestuali. Ogni stanza è un nodo del grafo; ogni transizione è un arco con una direzione e, in alcuni casi, una precondizione (la transizione è percorribile soltanto se il giocatore soddisfa determinati requisiti).

In termini implementativi, ogni stanza è definita da un oggetto TypeScript conforme all'interfaccia Room:

```
// types.ts
export interface Room {
    description: (state: PlayerState) => string;
    commands: Command[];
    items?: Item[];
}
```

La stanza ha tre componenti:

1. **description**: una funzione — non una stringa statica — che accetta lo stato corrente del giocatore e restituisce la descrizione dell'ambiente. La natura funzionale di questo campo è la chiave della **dinamicità narrativa** del sistema *(cfr. §5.3)*.
2. **commands**: un array di oggetti Command, ciascuno con una regex e un handler, che definiscono le azioni specifiche eseguibili in quell'ambiente. Questi comandi integrano (e possono sovrascrivere) i comandi globali del motore.
3. **items**: un array opzionale di oggetti Item, che definisce gli oggetti fisici presenti nella stanza — sia raccoglibili sia fissi. La lista è opzionale perché alcune stanze molto semplici non hanno oggetti interattivi.

5.2 Le Quindici Stanze

Il relitto è composto da quindici ambienti distinti, distribuiti su due livelli connessi da passaggi verticali. La Tab. 5.1 riporta l'elenco completo con le dimensioni del file sorgente corrispondente.

Tab. 5.1 — Stanze del gioco e dimensioni del sorgente

Stanza	File sorgente	LOC	Profilo audio
Plancia della Santa Maria	plancia.ts	46	ship
Stiva	stiva.ts	119	ship
Corridoio Principale	corridoioPrincipale.ts	224	alien_quiet
Scafo Esterno	scafoEsterno.ts	80	alien_cold

Stanza	File sorgente	LOC	Profilo audio
Camera di Compensazione	cameraDiCompensazione.ts	86	alien_quiet
Santuario del Silenzio	santuarioDelSilenzio.ts	110	alien_quiet
Scriptorium	scriptorium.ts	75	alien_quiet
Arca della Memoria	arcaDellaMemoria.ts	101	alien_electric
Serra Morente	serraMorente.ts	154	alien_quiet
Arca Biologica	arcaBiologica.ts	143	alien_cold
Alloggi dell'Equipaggio	alloggiEquipaggio.ts	100	alien_quiet
Laboratori di Risonanza	laboratoriRisonanza.ts	96	alien_electric
Ponte di Comando	ponteDiComando.ts	190	alien_electric
Anticamera del Santuario	anticameraSantuario.ts	24	alien_quiet
Santuario Centrale	santuarioCentrale.ts	81	sacred

La stanza con il file sorgente più esteso è il **Corridoio Principale** (224 LOC): si tratta del nodo centrale del grafo, la stanza attraverso cui il giocatore transita più frequentemente e che contiene la maggior parte delle connessioni con le altre aree. La **Anticamera del Santuario** (24 LOC) è la più semplice: è una stanza di transizione narrativa senza oggetti interattivi complessi.

5.3 Descrizioni Dinamiche

Il tratto architetturale più significativo del sistema delle stanze è la natura funzionale del campo description. Invece di contenere una stringa fissa, ogni stanza espone una funzione che, ricevendo lo stato corrente, calcola la descrizione appropriata.

Questo consente alle descrizioni di riflettere i cambiamenti avvenuti nel mondo:

```
// Esempio semplificato da corridoioPrincipale.ts
description: (state: PlayerState): string => {
    let desc = "CORRIDOIO PRINCIPALE\n\n";
    desc += "Un lungo corridoio che si estende davanti a te...";

    // Se l'energia è stata ripristinata, la descrizione cambia
    if (state.flags['isPowerRestored']) {
        desc += "\nOra i pannelli lungo le pareti sono illuminati di una tenue luce
azzurra.";
    } else {
        desc += "\nI pannelli lungo le pareti sono spenti e freddi.";
    }

    // Se un oggetto non è ancora stato raccolto, compare nella descrizione
    if (!state.flags['picked_corridoio_lastra'] && !state.inventory.includes('Lastra Dati'))
{
        desc += "\nAppoggiata alla parete di sinistra, quasi invisibile tra i detriti, una
lastra di materiale traslucido.";
    }
```

```
    return desc;
}
```

Questo meccanismo produce due effetti narrativi di grande importanza:

Effetto 1 — La stanza risponde alle azioni del giocatore: tornare in una stanza dopo aver compiuto progressi altrove mostra una versione diversa dell'ambiente, che riflette le modifiche apportate. Il mondo non è statico; la presenza del giocatore lascia tracce visibili.

Effetto 2 — Gli oggetti non raccogliti sono visibili nella descrizione: quando un oggetto è presente nella stanza e non è ancora stato preso, la sua descrizione è incorporata nel testo ambientale. Una volta raccolto, scompare dalla descrizione. Questo elimina la necessità del classico meccanismo «items: [lista statica]» tipico di molti sistemi di avventure testuali, in cui gli oggetti presenti vengono elencati separatamente dalla descrizione dell'ambiente.

5.4 Il Sistema dei Comandi di Stanza

Ogni stanza definisce un array commands: Command[]. Ogni elemento dell'array è un oggetto con due campi:

```
// types.ts
export interface Command {
    regex: string;
    handler: CommandHandler;
}

export type CommandHandler =
    (state: PlayerState, match: RegExpMatchArray) => CommandHandlerResult;
```

Il campo regex è una stringa che verrà compilata in un'espressione regolare (e poi cachata) dal motore. La regex è valutata sull'input **già normalizzato**, il che significa che non deve gestire varianti di capitalizzazione, articoli o preposizioni — queste sono già state rimosse dalla fase di normalizzazione.

L'handler riceve sia lo stato corrente (per leggere flag e inventario) sia il match dell'espressione regolare (che può contenere gruppi di cattura per estrarre i parametri del comando). L'handler può modificare lo stato (il clone che gli è stato passato) e deve restituire un CommandHandlerResult contenente la descrizione della risposta.

Un esempio tipico è il comando di movimento:

```
// Semplificato da corridoioPrincipale.ts
{
    regex: '^(vai\\s+)?nord$|^n$',
    handler: (state, _match) => {
        if (!state.flags['isPowerRestored']) {
            return {
                description: "La porta verso nord è bloccata da un meccanismo di sicurezza.
I pannelli di controllo sono spenti.",
                eventType: 'error'
            };
        }
        state.location = 'Ponte di Comando';
        state.visitedRooms = [...(state.visitedRooms ?? []), 'Ponte di Comando'];
        return {
            description: getRoomDescription('Ponte di Comando', state),
```

```
            eventType: 'movement'
        };
    }
}
```

5.5 La Struttura Narrativa: Atto I, Atto II, Atto III

Il grafo delle stanze è progettato per supportare una struttura narrativa in tre atti, non imposta esplicitamente dal codice ma emergente dalla logica delle precondizioni:

Atto I — Arrivo e orientamento: il giocatore parte dalla Plancia della *Santa Maria*, attraversa la Stiva per equipaggiarsi, raggiunge lo Scafo Esterno per analizzare la struttura aliena, e penetra nel relitto attraverso la Camera di Compensazione. In questa fase, il sistema di traduzione è a 0% e il mondo alieno è pressoché incomprensibile.

Atto II — Raccolta delle tre Chiavi Simboliche: il cuore del gioco. Il giocatore deve recuperare tre artefatti (il Nucleo di Memoria, la Stele del Ricordo e il Seme della Continuità) distribuiti in stanze diverse, ciascuna con un percorso di accesso che richiede oggetti e conoscenze acquisiti altrove. In questa fase, la traduzione avanza e il mondo alieno acquista progressivamente un significato.

Atto III — Il Santuario Centrale: il finale narrativo. Il giocatore porta le tre chiavi alla Porta Circolare nel Ponte di Comando, accede all'Anticamera e infine al Santuario Centrale, dove si svolge la conclusione della storia.

Questa struttura non è implementata come un sistema esplicito di «atti» nel codice: è il risultato naturale dell'interlocking delle precondizioni tra le stanze e i loro oggetti.

5.6 La Mappa Progressiva

Il comando MAPPA genera una rappresentazione ASCII del relitto che rivela soltanto le stanze già visitate dal giocatore (meccanismo di **fog of war**). La funzione getMappa in gameLogic.ts accede al campo visitedRooms dello stato e genera una griglia dove:

- Le stanze visitate sono mostrate con il loro nome abbreviato
- La stanza corrente è marcata con i delimitatori [*...*]
- Le stanze non ancora visitate appaiono come spazi vuoti

Questo meccanismo serve a due scopi. Il primo è pratico: il giocatore può orientarsi nel relitto senza dover tenere traccia manualmente delle stanze esplorate. Il secondo è narrativo: la mappa che cresce gradualmente è una metafora visiva dell'esplorazione stessa — il mondo si rivela man mano che viene percorso.

5.7 La Gestione delle Transizioni e dell'Autosave

Ogni volta che un comando produce un cambio di location nello stato, il componente App.tsx rileva la modifica e attiva due operazioni: l'aggiornamento del profilo audio

(passaggio al profilo ambientale della nuova stanza) e l'autosave silenzioso (salvataggio del nuovo stato su disco senza notifica visiva all'utente).

Il rilevamento del cambio di stanza avviene tramite un hook React useEffect che osserva playerState.location:

```
// App.tsx — effetto autosave su cambio stanza
useEffect(() => {
    if (playerState.location !== previousLocation.current) {
        previousLocation.current = playerState.location;
        // Aggiorna ambience audio
        const profile = ROOM_AMBIENCE[playerState.location] ?? 'alien_quiet';
        startAmbience(profile);
        // Salva automaticamente
        saveSlotData(-1, JSON.stringify(playerState));
    }
}, [playerState.location]);
```

Il meccanismo garantisce che il giocatore non possa perdere i propri progressi per una chiusura accidentale dell'applicazione, con una perdita massima di una stanza visitata.

Sezione VI — Il Sistema degli Oggetti e il Design degli Enigmi

6.1 L'Interfaccia Item

In *Il Relitto Silente*, ogni entità del mondo con cui il giocatore può interagire è rappresentata come un oggetto conforme all'interfaccia Item:

```
// types.ts
export interface Item {
    id: string;
    name: string;
    synonyms: string[];
    description: string;
    details?: string;
    isPickable?: boolean;
    isFixed?: boolean;
    onUse?: (state: PlayerState) => CommandHandlerResult;
    onAnalyze?: (state: PlayerState) => CommandHandlerResult;
    onCombine?: (targetId: string, state: PlayerState) => CommandHandlerResult;
}
```

La struttura distingue chiaramente tra:

Dati descrittivi: id (identificatore interno unico), name (nome canonico dell'oggetto), synonyms (varianti accettate nel parser), description (risposta al comando ESAMINA), details (risposta alternativa al comando ANALIZZA, usata quando non è necessaria logica condizionale).

Flag di comportamento: isPickable indica se l'oggetto può essere inserito nell'inventario; isFixed indica che l'oggetto è fisso nell'ambiente e non può essere spostato (altari, pannelli, strutture architettoniche). In assenza di entrambi i flag, il comportamento di default è «non raccoglibile, non fisso» — una categoria residuale usata per oggetti che il giocatore può esaminare ma non portare con sé.

Handler comportamentali: tre funzioni opzionali che vengono invocate dal motore quando il giocatore esegue le corrispondenti azioni. Ciascuna riceve lo stato corrente (mutabile, in quanto è già il clone) e restituisce un CommandHandlerResult con la descrizione della risposta.

6.2 Il Matching degli Oggetti: Sinonimi e Ricerca per Contesto

Quando il parser identifica un comando del tipo VERBO OGGETTO o USA OGGETTO SU BERSAGLIO, il motore deve trovare l'oggetto corrispondente. La ricerca avviene in due fasi:

Fase 1 — Ricerca locale: il motore verifica gli oggetti presenti nella stanza corrente. La corrispondenza avviene per nome canonico o per uno qualsiasi dei sinonimi definiti nel campo synonyms. Il confronto avviene sull'input normalizzato.

Fase 2 — Ricerca nell'inventario: se l'oggetto non è trovato nella stanza, il motore cerca nell'inventario del giocatore. Questo è il comportamento atteso per le azioni su oggetti che il giocatore ha già raccolto.

I sinonimi sono fondamentali per la naturalità del parser: un oggetto chiamato «Lastra Dati» potrebbe avere sinonimi come lastra, lastra dati, tavoletta, piastra, frammento. Il giocatore non è tenuto a conoscere il nome canonico — qualsiasi termine descrittivo ragionevole produce il comportamento corretto.

6.3 La Logica di onCombine: il Nucleo degli Enigmi

L'handler onCombine è il meccanismo su cui si fondano la maggior parte degli enigmi del gioco. Viene invocato quando il giocatore esegue un comando del tipo USA X SU Y, dove X è l'oggetto che agisce e Y è il bersaglio.

```
// Flusso di onCombine nel motore (semplificato da gameLogic.ts)
const handleUsaXSuY = (source: string, target: string, state: PlayerState) => {
    // Cerca l'oggetto sorgente nell'inventario
    const sourceItem = findItemInInventory(source, state);
    if (!sourceItem) {
        return { description: "Non hai questo oggetto.", eventType: 'error' };
    }

    // Cerca il bersaglio (nella stanza o nell'inventario)
    const targetItem = findItemInContext(target, state);
    if (!targetItem) {
        return { description: "Non vedo qui un bersaglio adatto.", eventType: 'error' };
    }

    // Invoca onCombine sull'oggetto sorgente, passando l'id del bersaglio
    if (sourceItem.onCombine) {
        return sourceItem.onCombine(targetItem.id, state);
    }

    return { description: "Non sembra funzionare così.", eventType: 'error' };
};
```

La scelta di attribuire l'handler all'oggetto *sorgente* — quello che agisce — piuttosto che al bersaglio è una decisione di design deliberata: incoraggia il giocatore a pensare agli oggetti dell'inventario come strumenti con uno scopo, piuttosto che come chiavi da inserire in serrature. L'inventario è un insieme di capacità, non di credenziali di accesso.

Un handler onCombine tipico ha la struttura seguente:

```
// Esempio semplificato
onCombine: (targetId: string, state: PlayerState): CommandHandlerResult => {
    if (targetId === 'pannello-energia') {
        if (!state.flags['isPowerRestored']) {
            state.flags['isPowerRestored'] = true;
            state.inventory = state.inventory.filter(i => i !== 'Batteria di Riserva');
            return {
                description: "Inserisci la batteria nel vano aperto del pannello. " +
                             "Un ronzio si propaga attraverso le pareti. " +
                             "Qualcosa, da qualche parte nella nave, si riaccende.",
                eventType: 'magic'
            };
        } else {
            return {
```

```
                description: "Il pannello è già alimentato.",
                eventType: 'error'
            };
        }
    }
    return {
        description: "Questa batteria non si adatta qui.",
        eventType: 'error'
    };
}
```

6.4 Le Tre Chiavi Simboliche

Le cosiddette **Chiavi Simboliche** sono tre oggetti che costituiscono l'obiettivo centrale dell'Atto II. Ognuna di esse richiede un percorso di acquisizione dedicato, con precondizioni che si intrecciano con le meccaniche di altre stanze.

Il Nucleo di Memoria: conservato nell'Arca della Memoria, accessibile soltanto dopo aver ripristinato l'alimentazione elettrica della nave e attivato il terminale principale. L'analisi del Nucleo di Memoria porta la traduzione al 100% e innesca la sequenza narrativa più significativa del gioco *(cfr. Sezione VII.4)*.

La Stele del Ricordo: situata nel Santuario del Silenzio, protetta da un meccanismo di blocco che richiede l'uso di un oggetto specifico trovato altrove. La Stele è il pezzo più ricco di contenuto narrativo: la sua analisi porta la traduzione dal 18% al 75%, rivelando la lingua K'tharr quasi nella sua completezza.

Il Seme della Continuità: l'oggetto più difficile da ottenere. Si trova nella Serra Morente in uno stato di blocco fisico (chiuso in una teca sigillata) che richiede due operazioni sequenziali, la seconda delle quali è disponibile solo dopo aver completato un'azione in una stanza diversa.

Questa struttura di interlocking tra gli enigmi — dove la soluzione di un problema in una stanza abilita la soluzione di un problema in un'altra — è un principio fondamentale del game design delle avventure testuali di qualità. Impedisce al gioco di essere risolto linearmente e costringe il giocatore a tornare in stanze già visitate con nuove conoscenze e nuovi strumenti.

6.5 Filosofia del Design degli Enigmi

Il design degli enigmi de *Il Relitto Silente* segue quattro principi guida:

Principio 1 — La logica è interna al mondo: nessun enigma richiede conoscenze esterne al gioco. Tutto ciò che serve per risolvere qualsiasi problema è presente nell'ambiente — in una descrizione, in un'analisi, in un eco temporale. Il giocatore non deve indovinare: deve osservare.

Principio 2 — L'analisi è la chiave: il comando ANALIZZA rivela la natura tecnica degli oggetti e, spesso, indica esplicitamente con cosa sono compatibili. «Il meccanismo richiede un picco di energia ad alto voltaggio» è un'istruzione implicita: il giocatore deve trovare una

fonte di energia ad alto voltaggio. Questo meccanismo trasforma le analisi da semplice lore in strumenti di risoluzione.

Principio 3 — Gli enigmi hanno una dimensione narrativa: ogni soluzione è coerente con la storia del mondo. Non si usa un oggetto a caso su un bersaglio a caso; si usa uno strumento che ha senso usare in quel contesto, nel modo in cui lo userebbe chiunque nel mondo diegetico. Questa coerenza riduce la componente di trial-and-error e aumenta la soddisfazione della soluzione.

Principio 4 — Il fallimento è informativo: quando un'azione non funziona, il messaggio di risposta non è generico («Non puoi farlo») ma specifico («La teca è sigillata da un meccanismo che sembra richiedere una chiave energetica, non una meccanica»). Il fallimento è un'informazione sul problema, non una punizione.

6.6 L'Inventario e la sua Gestione

L'inventario è implementato come un semplice array di stringhe (string[]) nel PlayerState: ogni stringa è il nome canonico di un oggetto. La scelta di usare stringhe invece di riferimenti agli oggetti Item è deliberata: semplifica la serializzazione per il sistema di salvataggio (un array di stringhe si serializza e deserializza in JSON senza ambiguità) e riduce la complessità del codice di gestione dell'inventario.

Non esiste un limite massimo di oggetti in inventario. Questa scelta — che rompe con la tradizione di molti giochi testuali classici, che imponevano un peso massimo o un numero massimo di oggetti — è motivata dalla volontà di eliminare una fonte di frustrazione non produttiva. In *Il Relitto Silente*, il giocatore non deve preoccuparsi della logistica dell'inventario; può concentrarsi sull'esplorazione e sulla risoluzione degli enigmi.

La visualizzazione dell'inventario tramite INVENTARIO / I è generata dalla funzione getInventarioHtml, che produce markup HTML interno (non interpolazione di input utente) con formattazione CRT-coerente.

6.7 Il Sintonizzatore di Frequenza e gli Echi

Il Sintonizzatore di Frequenza è un oggetto speciale con un comportamento che non rientra nei pattern standard degli enigmi. Non è uno strumento per risolvere puzzle: è uno strumento per ascoltare. Il suo onUse handler verifica la stanza corrente nella mappa LOCATION_TO_ECHO (definita in echoData.ts) e, se presente, riproduce il testo dell'eco corrispondente:

```
// Struttura semplificata dell'handler del Sintonizzatore
onUse: (state: PlayerState): CommandHandlerResult => {
    const echoId = LOCATION_TO_ECHO[state.location];
    if (!echoId) {
        return {
            description: "Il Sintonizzatore analizza la frequenza ambientale. " +
                         "Nessuna traccia residua rilevata in questo luogo.",
            eventType: 'tech'
```

```
        };
    }
    if (state.echoes.includes(echoId)) {
        return {
            description: "Hai già registrato l'eco di questo luogo.",
            eventType: null
        };
    }
    // Prima volta: aggiunge l'eco alla lista e mostra il testo
    state.echoes.push(echoId);
    state.flags[`echoDetected_${state.location}`] = true;
    return {
        description: ECHO_TEXTS[echoId],
        eventType: 'echo',
        typewriter: true
    };
}
```

Il flag typewriter: true nella risposta attiva l'animazione di scrittura carattere per carattere nel TerminalOutput, che rallenta la comparsa del testo dell'eco e aumenta l'impatto narrativo del momento.

Sezione VII — Il Sistema di Traduzione Progressiva

7.1 La Lingua K'tharr come Meccanica di Gioco

Il sistema di traduzione progressiva è la meccanica narrativa più originale de *Il Relitto Silente*. L'idea di fondo è semplice: il giocatore entra in una nave aliena dove ogni cosa — iscrizioni, ologrammi, registrazioni vocali — è in una lingua incomprensibile. Man mano che il suo scanner analizza artefatti significativi, decodifica progressivamente questa lingua, e il mondo acquista senso davanti ai suoi occhi.

Questa meccanica ha una duplice funzione. Sul piano **ludico**, crea un arco di progressione che motiva l'esplorazione sistematica: ogni oggetto analizzato potrebbe aumentare la comprensione e rivelare nuovi significati. Sul piano **narrativo**, riproduce la struttura dell'archeologia e della linguistica applicata: la comprensione non arriva tutta in una volta, ma si accumula in strati, e ogni frammento decodificato getta nuova luce su quelli precedenti.

7.2 Il Flag translationProgress

La progressione della traduzione è tracciata da un singolo valore numerico nello PlayerState.flags:

```
state.flags['translationProgress'] = 0; // valore iniziale
```

Il valore è un numero intero da 0 a 100, che rappresenta la percentuale di decodifica della matrice linguistica K'tharr. Ogni analisi di un artefatto significativo incrementa questo valore di una quantità predefinita. Il flag è di tipo number all'interno del dizionario flags: Record<string, boolean | number | string>, che ammette valori eterogenei.

7.3 Gli Artefatti Linguistici e i loro Incrementi

Quattro oggetti del gioco contribuiscono alla traduzione, in ordine di scoperta tipica:

Tab. 7.1 — Artefatti linguistici e loro contributo alla traduzione

Artefatto	Stanza	Contributo	Totale cumulativo
Lastra Dati	Corridoio Principale	+4%	4%
Cilindro Mnemonico	Alloggi dell'Equipaggio	+14%	18%
Stele del Ricordo	Santuario del Silenzio	+57%	75%
Nucleo di Memoria	Arca della Memoria	+25%	100%

La distribuzione degli incrementi è narrativamente motivata. La Lastra Dati è un frammento minore — le prime lettere di un alfabeto incomprensibile. Il Cilindro Mnemonico aggiunge vocabolario di base. La Stele del Ricordo è il documento più significativo dell'intera

nave: è un'iscrizione progettata dai K'tharr per essere leggibile da civiltà aliene, un equivalente tecnologico della Stele di Rosetta. Il Nucleo di Memoria, infine, completa la decodifica portando la comprensione alla piena maturità.

7.4 L'Implementazione dell'Incremento

L'incremento della traduzione avviene nell'handler onAnalyze degli artefatti linguistici. La struttura tipica è la seguente:

```
// Struttura semplificata di onAnalyze per un artefatto linguistico
onAnalyze: (state: PlayerState): CommandHandlerResult => {
    const increment = 57; // contributo specifico dell'artefatto
    const currentPct = (state.flags['translationProgress'] as number) ?? 0;
    const newPct = Math.min(100, currentPct + increment);
    state.flags['translationProgress'] = newPct;
    state.flags['steleAnalizzata'] = true;

    const barBefore = progressBar(currentPct);
    const barAfter  = progressBar(newPct);

    return {
        description:
            "ANALISI IN CORSO...\n\n" +
            barBefore + "\n\n" +
            "Lo scanner elabora le iscrizioni. I simboli K'tharr si riorganizzano " +
            "nella matrice di traduzione. Alcune sequenze erano già note — " +
            "le strutture grammaticali di base — ma questa stele è diversa: " +
            "è un documento volontariamente bilingue, pensato per essere letto " +
            "da chi non conosce la lingua.\n\n" +
            "[PAUSE]\n\n" +
            "La traduzione avanza con una velocità inaspettata.\n\n" +
            barAfter + "\n\n" +
            "Testo tradotto: «Noi che partiamo lasciamo queste parole " +
            "non per essere ricordati, ma perché qualcuno sappia " +
            "che siamo stati qui, e che era importante esserci stati.»",
        eventType: 'tech',
        typewriter: true
    };
}
```

La struttura del testo prodotto da questi handler segue un formato narrativo preciso: 1. **Stato iniziale**: la barra di progresso prima dell'analisi 2. **Narrazione del processo**: una descrizione in prima persona di cosa sta succedendo 3. **Marcatore [PAUSE]**: divide il testo in due «pagine», creando una pausa drammatica 4. **Stato finale**: la barra di progresso aggiornata 5. **Contenuto tradotto**: una porzione del testo K'tharr decodificato

---

7.5 Il Marcatore [PAUSE] e la Paginazione

Il marcatore [PAUSE] inserito nel testo di risposta attiva un meccanismo di paginazione nel componente App.tsx. Il testo viene suddiviso nel punto del marcatore, e la seconda parte viene mostrata soltanto quando il giocatore preme Invio o qualsiasi altro tasto (esclusi i tasti funzione).

```
// App.tsx — gestione continuation
const paginateText = (text: string): string[] => {
```

```
    return text.split('[PAUSE]').map(p => p.trim()).filter(Boolean);
};
```

Questo meccanismo è particolarmente utile per le analisi degli artefatti linguistici, che producono testi lunghi. Invece di mostrare tutto il testo in una volta sola (il che richiederebbe lo scorrimento del terminale), la paginazione crea momenti narrativi distinti: prima il giocatore vede il processo di analisi, poi — dopo una pausa consapevole — vede il risultato.

7.6 Il Momento del 100%: Una Sequenza Speciale

Quando la traduzione raggiunge il 100% — ovvero quando il giocatore analizza il Nucleo di Memoria — avviene qualcosa di insolito. Il testo prodotto dall'handler non è una risposta tecnica. È una risposta umana.

La scena è progettata come un momento di pausa narrativa nel mezzo dell'avventura: lo scanner, che fino a quel momento ha parlato il linguaggio asciutto dell'analisi tecnica, riporta una traduzione che cambia la natura del viaggio. Il giocatore non è più un esogrado in missione di ricognizione; è diventato, involontariamente, il destinatario di un messaggio lasciato millenni prima.

Il momento del 100% può essere **perso**. Se il giocatore analizza il Nucleo di Memoria troppo tardi — dopo aver già attivato certi meccanismi che cambiano lo stato della stanza — la sequenza speciale non si attiva. Il gioco non avverte il giocatore di questo rischio; è un segreto che il gioco custodisce per chi esplora con attenzione. Il file SOLUZIONE.md nella documentazione per il giocatore indica la sequenza esatta per non perderlo.

7.7 La Traduzione come Cambiamento Percettivo

Un dettaglio implementativo importante è che la traduzione non modifica soltanto il testo dell'analisi. Alcune descrizioni ambientali cambiano in funzione del valore di translationProgress. Stanze in cui le iscrizioni sono descritte come «simboli incomprensibili» quando la traduzione è a 0% diventano progressive in termini di leggibilità:

- 0–17%: «simboli incomprensibili, forse una forma di scrittura»
- 18–74%: «caratteri che il tuo scanner sta cominciando a decifrare — alcune parole sono già distinguibili»
- 75–99%: «testo quasi interamente leggibile — la lingua K'tharr emerge dalla pietra in tutta la sua complessità»
- 100%: «il testo è ora completamente comprensibile»

Questo cambiamento percettivo progressivo è la realizzazione narrativa della meccanica: il mondo non è diverso, ma il giocatore lo vede diversamente. È un esempio di come una meccanica di gioco possa diventare, con la giusta implementazione, un dispositivo narrativo.

7.8 Il Diario come Finestra sulla Traduzione

Il comando NOTA / DIARIO rispecchia la progressione della traduzione nella sua sezione dedicata:

```
MATRICE DI TRADUZIONE
[ MATRICE TRADUZIONE: ███████▒▒ 75% ]

Artefatti analizzati:
  - Lastra Dati (Corridoio Principale): +4%
  - Cilindro Mnemonico (Alloggi): +14%
  - Stele del Ricordo (Santuario del Silenzio): +57%

Artefatto mancante: Nucleo di Memoria
```

Il diario funge da cruscotto della comprensione linguistica: il giocatore può verificare in qualsiasi momento a che punto si trova la decodifica e quali artefatti ha già analizzato, senza dover ricordare i dettagli di sessioni di gioco precedenti.

Sezione VIII — La Sintesi Audio Procedurale

8.1 La Scelta del Silenzio come Punto di Partenza

Il Relitto Silente non contiene un solo file audio preregistrato. Nessun MP3, nessun WAV, nessun OGG. Tutta la dimensione sonora del gioco — sia le atmosfere ambientali sia gli effetti sonori — è generata in tempo reale dal processore del computer dell'utente, tramite le Web Audio API native del browser (e di Electron).

Questa scelta è, nelle intenzioni del progetto, una scelta **estetica prima che tecnica**. Un gioco che non usa grafica pre-renderizzata, che chiede all'immaginazione del giocatore di costruire lo spazio visivo, doveva fare lo stesso con il suono. Non registrazioni di «rumori spaziali» scaricate da una libreria — qualcosa di più fondamentale. Frequenze. Vibrazioni. Il suono come fenomeno fisico, non come rappresentazione.

C'è una coerenza profonda, in questo, con la natura del medium testuale: così come il testo indica una direzione e l'immaginazione costruisce lo spazio visivo, le frequenze indicate dalla sintesi indicano un'atmosfera e l'immaginazione (e il sistema uditivo) costruisce l'ambiente sonoro.

8.2 Architettura del Servizio Audio

Il sistema audio è implementato nel file services/audioService.ts (266 LOC). Il file esporta funzioni pure che operano su un AudioContext singleton, mantenuto come variabile di modulo:

```
// services/audioService.ts
let audioCtx: AudioContext | null = null;
let ambienceGain: GainNode | null = null;
let ambienceStoppable: AudioScheduledSourceNode[] = [];
```

L'AudioContext viene inizializzato in modo **lazy** — soltanto al primo gesto dell'utente (click, keydown) — per rispettare la politica di autoplay dei browser moderni e di Electron, che richiedono un'azione dell'utente prima di permettere la riproduzione audio.

```
const initializeAudio = (): void => {
    if (!audioCtx) {
        audioCtx = new AudioContext();
    }
    if (audioCtx.state === 'suspended') {
        audioCtx.resume();
    }
};

const ensureAudioInitialized = (): void => {
    if (!audioCtx || audioCtx.state === 'suspended') {
        initializeAudio();
    }
};
```

Ogni funzione audio pubblica chiama ensureAudioInitialized() come prima operazione, garantendo che l'AudioContext sia sempre in stato attivo prima dell'uso.

8.3 La Funzione Primitiva: playTone

La funzione playTone è il mattone elementare su cui è costruito l'intero sistema di effetti sonori:

```
// services/audioService.ts — playTone
const playTone = (
    frequency: number,
    duration: number,
    type: OscillatorType = 'sine'
): void => {
    if (!audioCtx) return;
    const oscillator = audioCtx.createOscillator();
    const gainNode   = audioCtx.createGain();

    oscillator.type      = type;
    oscillator.frequency.setValueAtTime(frequency, audioCtx.currentTime);

    gainNode.gain.setValueAtTime(getSfxVol() * 0.3, audioCtx.currentTime);
    gainNode.gain.exponentialRampToValueAtTime(0.0001, audioCtx.currentTime + duration);

    oscillator.connect(gainNode);
    gainNode.connect(audioCtx.destination);

    oscillator.start(audioCtx.currentTime);
    oscillator.stop(audioCtx.currentTime + duration);
};
```

La struttura è quella classica della sintesi sottrattiva semplificata: un oscillatore che genera la forma d'onda, un nodo GainNode che controlla l'ampiezza con un inviluppo di decadimento esponenziale. L'inviluppo esponenziale (da getSfxVol() * 0.3 a 0.0001 in duration secondi) produce un suono con un attacco immediato e un decadimento naturale, coerente con l'estetica elettronica dell'interfaccia CRT.

8.4 Gli Effetti Sonori

Il sistema di effetti sonori (SFX) definisce sei funzioni pubbliche, ciascuna associata a un tipo specifico di evento di gioco:

```
// Tastiera — pressione di un tasto durante la digitazione
export const playKeystrokeSound = (): void => {
    ensureAudioInitialized();
    playTone(220, 0.05, 'square');
};

// Invio — conferma di un comando
export const playSubmitSound = (): void => {
    ensureAudioInitialized();
    playTone(440, 0.1, 'square');
};

// Raccolta oggetto — feedback positivo
export const playItemSound = (): void => {
    ensureAudioInitialized();
```

```
    playTone(880, 0.05, 'triangle');
    setTimeout(() => { if (audioCtx) playTone(1046, 0.05, 'triangle'); }, 60);
};

// Evento magico / scoperta significativa
export const playMagicSound = (): void => {
    ensureAudioInitialized();
    playTone(1046.50, 0.1, 'sine');
    setTimeout(() => { if (audioCtx) playTone(1396.91, 0.1, 'sine'); }, 100);
    setTimeout(() => { if (audioCtx) playTone(1567.98, 0.2, 'sine'); }, 200);
};

// Movimento — cambio di stanza
export const playMoveSound = (): void => {
    ensureAudioInitialized();
    playNoise(0.15);
};

// Errore — comando non riconosciuto
export const playErrorSound = (): void => {
    ensureAudioInitialized();
    playTone(110, 0.15, 'sawtooth');
};
```

Le frequenze degli SFX non sono scelte casualmente. playKeystrokeSound usa 220 Hz (La3) con forma d'onda quadra, che riproduce il click secco di un terminale meccanico. playSubmitSound usa 440 Hz (La4, la frequenza di riferimento universale), un semitono sopra, per segnalare la conferma. playItemSound usa due note in successione rapida (Do6 e Do#6) con forma d'onda triangolare (più morbida), producendo un «cling» piacevole. playMagicSound costruisce un accordo arpeggiato (Do6, Mi6, Sol6 — un accordo di Do maggiore) con oscillatori sinusoidali puri, per segnalare una scoperta significativa.

playErrorSound usa 110 Hz (La2) con forma d'onda a dente di sega, che produce un suono aspro e dissonante — intenzionalmente sgradevole per rinforzare il feedback negativo senza essere fastidioso.

8.5 I Cinque Profili di Ambience

Il sistema di ambience è la componente audio più complessa del progetto. Ogni area del gioco è associata a uno dei cinque profili sonori, ciascuno costruito da una combinazione di oscillatori e generatori di rumore:

Profilo SHIP — La nave umana

```
// Profilo 'ship': suono di motori e ventilazione
// Oscillatore sawtooth 55 Hz (gain 0.04) + sawtooth 110 Hz (gain 0.02)
// + Noise bianco passa-banda 300 Hz (BW 0.8)
```

Tre sorgenti sonore simulate: - Un basso profondo a 55 Hz (Si0) con forma d'onda a dente di sega simula il rombo dei motori principali - Un armonico a 110 Hz (La2) aggiunge spessore al suono meccanico - Un rumore bianco filtrato attorno a 300 Hz simula il sibilo di ventilazione e pressione dell'aria

Il risultato è un suono familiare e rassicurante — il suono del mondo conosciuto dal giocatore, prima di attraversare la soglia verso il relitto alieno.

Profilo ALIEN QUIET — L'interno alieno in stato di quiete

```
// Profilo 'alien_quiet': vibrazione singola con modulazione lenta
// Oscillatore sine 432 Hz + LFO sine 0.1 Hz (modulazione frequenza ±8 Hz)
```

Una singola frequenza di 432 Hz (una scelta deliberata: 432 Hz è un'accordatura alternativa al La standard di 440 Hz, con proprietà psicoacustiche spesso associate alla sensazione di «spazio» e «apertura») modulata da un LFO a 0,1 Hz — un ciclo ogni dieci secondi. Il vibrato resultante è quasi impercettibile nella sua lentezza, ma presente.

Questa lentezza è intenzionale: la stanza non ha fretta. Ha aspettato millenni. Non ha bisogno di segnalare urgenza.

Profilo ALIEN COLD — Il gelo dello spazio vuoto

```
// Profilo 'alien_cold': frequenza subsonica + rumore tenue
// Oscillatore sine 28 Hz (gain molto basso, sul limite dell'udibile)
// + Noise bianco passa-banda 180 Hz (BW 1.2)
```

28 Hz è al limite inferiore della percezione uditiva umana standard (approssimativamente 20 Hz). Non viene «sentito» con le orecchie nel senso tradizionale del termine: viene percepito come pressione, vibrazione, una sensazione fisica diffusa. Combinato con un rumore tenue a frequenze medie, produce la sensazione di vuoto gelido che caratterizza le stanze dello scafo esterno e dell'arca biologica.

Profilo ALIEN ELECTRIC — La tensione latente

```
// Profilo 'alien_electric': frequenze irregolari, dissonanti
// Oscillatore square 60 Hz + square 120 Hz (armonico)
// + Noise bianco passa-banda 600 Hz (BW 2.0)
```

60 Hz è la frequenza della corrente alternata standard nei sistemi elettrici (50 Hz in Europa, 60 Hz negli USA; la scelta di 60 Hz è coerente con l'ambientazione futuristica internazionale). La forma d'onda quadra produce armonici dispari, caratteristici dei dispositivi elettrici. Il rumore a 600 Hz simula il crepitio di sistemi ancora attivi ma instabili. Questo profilo è usato nei Laboratori di Risonanza, nell'Arca della Memoria e nel Ponte di Comando — stanze in cui i sistemi K'tharr sono ancora parzialmente funzionanti.

Profilo SACRED — Il Santuario

```
// Profilo 'sacred': frequenza pura con modulazione minima
// Oscillatore sine 528 Hz + LFO sine 0.05 Hz (modulazione frequenza ±3 Hz)
```

528 Hz è una frequenza con storia nella cultura popolare musicale e nella pseudoscienza del «healing sound»: indipendentemente dalla validità di queste teorie, la frequenza produce un suono puro e chiaro che molti ascoltatori associano a uno stato di calma e attenzione. Il LFO a 0,05 Hz (un ciclo ogni venti secondi) produce un vibrato talmente lento da essere percepito più come respirazione che come modulazione.

Il profilo SACRED è usato esclusivamente nel Santuario Centrale — il luogo della conclusione narrativa del gioco. Il cambiamento percepito dal giocatore quando entra in questa stanza per la prima volta, dopo aver attraversato tutta la nave con i profili «alieni», è uno dei momenti più significativi dell'esperienza.

8.6 Il Fade-in e il Fade-out

Un dettaglio implementativo importante è il sistema di transizione tra profili audio:

```
// services/audioService.ts — startAmbience (estratto)
export const startAmbience = (
    type: 'ship' | 'alien_quiet' | 'alien_cold' | 'alien_electric' | 'sacred' | null
): void => {
    stopAmbience();    // Prima ferma il profilo corrente con fade-out
    if (!isAmbienceEnabled() || !type) return;
    ensureAudioInitialized();
    if (!audioCtx) return;

    ambienceGain = audioCtx.createGain();
    const targetVol = getAmbienceVol();
    // Fade-in esponenziale in 1.5 secondi
    ambienceGain.gain.setValueAtTime(0.0001, audioCtx.currentTime);
    ambienceGain.gain.exponentialRampToValueAtTime(targetVol, audioCtx.currentTime + 1.5);
    ambienceGain.connect(audioCtx.destination);
    // ... configurazione oscillatori per il profilo specifico
};

export const stopAmbience = (): void => {
    const nodesToStop = [...ambienceStoppable];
    ambienceStoppable = [];
    const gainToFade  = ambienceGain;
    ambienceGain = null;
    if (audioCtx && gainToFade && nodesToStop.length > 0) {
        const now = audioCtx.currentTime;
        const cur = Math.max(gainToFade.gain.value, 0.0001);
        // Fade-out esponenziale in 0.7 secondi
        gainToFade.gain.setValueAtTime(cur, now);
        gainToFade.gain.exponentialRampToValueAtTime(0.0001, now + 0.7);
        setTimeout(() => {
            nodesToStop.forEach(n => { try { n.stop(); } catch { /* ignorato */ } });
            try { gainToFade.disconnect(); } catch { /* ignorato */ }
        }, 750);
    }
};
```

Il fade-in dura 1,5 secondi; il fade-out dura 0,7 secondi. La scelta di rampe esponenziali (invece che lineari) riflette la risposta logaritmica dell'orecchio umano all'ampiezza: una rampa esponenziale produce una transizione percettivamente più uniforme. L'uso di 0.0001 come valore minimo invece di 0 evita il problema matematico del logaritmo di zero nelle funzioni di interpolazione della Web Audio API.

8.7 La Gestione dei Settaggi Audio

Il volume e l'attivazione dell'audio sono configurabili separatamente per SFX e ambience, tramite quattro chiavi persistenti:

```
relitto_sfx_on       (boolean, default: true)
relitto_sfx_vol      (float 0.0–1.0, default: 0.7)
relitto_ambience_on  (boolean, default: true)
relitto_ambience_vol (float 0.0–1.0, default: 0.5)
```

I settaggi sono memorizzati tramite storageService e letti **in modo sincrono** attraverso la cache in memoria di settingsCache. Questo è fondamentale: le funzioni audio di audioService.ts devono essere sincrone per funzionare correttamente con la Web Audio API, che non supporta await. Il layer di storage asincrono (IPC Electron) viene usato soltanto al

caricamento iniziale dell’applicazione; da quel momento in poi, tutti gli accessi ai settaggi avvengono sulla cache in memoria.

Sezione IX — L'Interfaccia Utente e il Layer di Persistenza

9.1 La Filosofia dell'Interfaccia

L'interfaccia de *Il Relitto Silente* è fondata su un principio che potremmo chiamare **autenticità dell'estetica**: non si tratta di una skin sovrapposta a un'applicazione moderna, ma di un'interfaccia progettata dall'interno per replicare la coerenza visiva, la logica di interazione e le limitazioni estetiche di un terminale CRT degli anni Settanta e Ottanta.

Questo significa che ogni scelta visiva — il verde fosforescente su nero assoluto, il font pixel art, il cursore lampeggiante, la barra dei tasti funzione stile DOS — ha una controparte funzionale nella struttura dell'applicazione. L'estetica non è decorazione: è l'interfaccia stessa.

9.2 Il Sistema di Colori CRT

Il foglio di stile principale (index.css, circa 7.000 LOC incluso Tailwind) definisce le variabili CSS che governano l'intera palette:

```
:root {
    --bg-screen:  #030b02;    /* Nero assoluto con minima tinta verde */
    --p-main:     #33ff00;    /* Verde fosforescente principale */
    --p-bright:   #66ff00;    /* Verde luminoso (hover, selezioni) */
    --p-dim:      #1a4d1a;    /* Verde scuro (elementi secondari) */
    --p-glow-a:   #33ff00aa; /* Verde con trasparenza (effetti glow) */
    --border-crt: #1a4d1a;    /* Verde scuro per bordi e cornice */
}
```

Il colore #33ff00 è una calibrazione deliberata: non è il verde standard del CSS (#00ff00), né il verde comune nelle ricreazioni del look CRT (#4AF626). È una tinta leggermente più fredda, con meno giallo, che replica più fedelmente il fosforo P1 usato nei monitor a raggi catodici monocromatici degli anni Settanta. La componente rossa ridotta (0x33 invece di 0x00) aggiunge una lieve lucentezza che riduce l'affaticamento visivo durante sessioni prolungate.

9.3 Gli Effetti CRT

Il foglio di stile implementa quattro effetti visivi che simulano le caratteristiche fisiche di un monitor a raggi catodici:

Flicker (crt-flicker): una leggera variazione dell'opacità ciclica (da 100% a 98% in 0,05 secondi) simula il refresh rate tipico dei CRT a 50/60 Hz. L'effetto è abbastanza sottile da non essere coscio ma abbastanza presente da contribuire alla percezione complessiva di «calore» dell'immagine.

Noise (crt-noise): una texture di rumore pseudo-casuale animata simula le impurità del fosforo e il rumore elettronico del segnale. È implementata come un overlay semi-trasparente con animazione CSS.

Vignette (crt-vignette): un gradiente radiale scuro ai bordi dello schermo simula la curvatura e l'oscuramento perimetrale tipici dei monitor sferici. Contribuisce a focalizzare l'attenzione al centro dello schermo.

Scanline (scanline): righe orizzontali sottili, semi-trasparenti, simulate tramite un gradiente ripetuto, replicano la struttura a linee di scansione dei CRT.

9.4 Il Sistema di Scaling

Il Relitto Silente è progettato per una risoluzione nativa di 1920×1080 pixel. Per garantire un'esperienza visiva ottimale su qualsiasi risoluzione — da 800×600 a monitor 4K — il componente App.tsx calcola dinamicamente un fattore di scala:

```
// App.tsx — useGameScale hook
const useGameScale = () => {
    const [scale, setScale] = useState(1);

    useEffect(() => {
        const updateScale = () => {
            const scaleX = window.innerWidth  / 1920;
            const scaleY = window.innerHeight / 1080;
            // Usa il fattore minore per non uscire dallo schermo
            setScale(Math.min(scaleX, scaleY));
        };
        updateScale();
        window.addEventListener('resize', updateScale);
        return () => window.removeEventListener('resize', updateScale);
    }, []);

    return scale;
};
```

La «cornice CRT» (l'area di gioco) viene poi renderizzata con una trasformazione CSS scale(), mantenendo l'aspect ratio 16:9 e centrando il contenuto nello spazio disponibile. Questo approccio garantisce che il gioco sia visivamente identico su qualsiasi schermo, senza distorsioni.

9.5 La Macchina a Stati dell'Applicazione

Lo stato globale dell'applicazione è modellato come una macchina a stati esplicita tramite l'enum GameState:

```
// types.ts
export enum GameState {
    Boot,         // Sequenza di avvio (BootScreen)
    StartMenu,    // Menu principale (StartScreen)
    Instructions,// Istruzioni (PaginatedScreen)
    Intro,        // Sequenza introduttiva (IntroScreen)
    Playing,      // Partita in corso (terminale principale)
    GameOver,     // Schermata di fine gioco
    Credits,      // Crediti (PaginatedScreen)
}
```

La transizione tra stati è gestita dalla funzione setGameState di React. Ogni stato corrisponde a un componente React diverso visualizzato nell'area principale dello schermo. I

componenti sovrapposti (overlay di pausa, inventario, mappa, salvataggio, impostazioni audio) sono gestiti separatamente come boolean nello stato locale di App.

9.6 Il Componente TerminalOutput

Il TerminalOutput (109 LOC) è il componente responsabile della visualizzazione dell'output del gioco. Gestisce tre tipi di riga:

kind: 'text': Testo puro renderizzato come elemento React standard. I titoli delle stanze (rilevati dalla funzione isRoomTitle, che verifica che la stringa sia in maiuscolo con lunghezza compresa tra 4 e 40 caratteri) ricevono uno stile evidenziato.

kind: 'html': Markup HTML generato internamente (inventario, mappa) renderizzato tramite dangerouslySetInnerHTML. L'uso di questa proprietà React è sicuro nel contesto del progetto perché il markup è sempre generato da funzioni interne (getInventarioHtml, getMappa) e mai da input dell'utente.

kind: 'typewriter': Testo animato carattere per carattere a 10 ms per carattere. Il componente TypewriterLine gestisce questa animazione tramite un intervallo che aggiunge progressivamente caratteri al testo visualizzato. Qualsiasi pressione di tasto (eccetto i tasti funzione) completa istantaneamente l'animazione, mostrando l'intero testo.

9.7 Il Componente CommandLine e la Storia dei Comandi

Il CommandLine (123 LOC) gestisce il campo di input del terminale. Implementa una navigazione della storia dei comandi tramite i tasti Freccia Su e Freccia Giù:

```
// components/CommandLine.tsx — navigazione storia
const handleKeyDown = (e: React.KeyboardEvent<HTMLInputElement>) => {
    if (e.key === 'ArrowUp') {
        e.preventDefault();
        if (historyIndex < history.length - 1) {
            const newIndex = historyIndex + 1;
            setHistoryIndex(newIndex);
            setInput(history[history.length - 1 - newIndex]);
        }
    } else if (e.key === 'ArrowDown') {
        e.preventDefault();
        if (historyIndex > 0) {
            const newIndex = historyIndex - 1;
            setHistoryIndex(newIndex);
            setInput(history[history.length - 1 - newIndex]);
        } else if (historyIndex === 0) {
            setHistoryIndex(-1);
            setInput('');
        }
    }
};
```

La storia dei comandi è mantenuta in un array nello stato di App e passata al CommandLine come prop. I comandi vengono aggiunti alla storia dopo ogni invio, consentendo al giocatore di richiamare e modificare comandi precedenti senza riscriverli.

9.8 La Barra dei Tasti Funzione

La barra dei tasti funzione nella parte inferiore dello schermo è un elemento dell'interfaccia che richiama direttamente i menu stile DOS dei sistemi operativi degli anni Ottanta. È sempre visibile durante il gioco e fornisce accesso rapido alle funzioni principali:

Tasto	Etichetta	Funzione
F1	ISTRUZIONI	Apre la pagina delle istruzioni
F2	GIOCA	Avvia una nuova partita
F3	CARICA	Apre il selettore di caricamento
F4	SALVA	Apre il selettore di salvataggio
F5	ESCI	Chiude l'applicazione
F6	INVENTARIO	Mostra l'inventario corrente
F7	MAPPA	Mostra la mappa progressiva
F8	CREDITI	Apre la schermata dei crediti
F9	PAUSA	Apre la schermata di pausa

I tasti vengono abilitati o disabilitati in funzione del GameState corrente: F3 e F4, ad esempio, sono attivi soltanto durante la partita. L'implementazione è tramite un event listener globale su keydown nel componente App.

9.9 Il Sistema di Salvataggio: Architettura IPC

Il layer di persistenza è implementato come una collaborazione tra tre file:

electron/main.ts: definisce i gestori IPC nel processo principale. Due famiglie di handler: storage:writeSlot / storage:readSlot per i dati di gioco, storage:writeSettings / storage:readSettings per le impostazioni audio.

```
// electron/main.ts — percorsi di salvataggio
function getSavesDir(): string {
    const dir = path.join(app.getPath('userData'), 'saves');
    if (!fs.existsSync(dir)) fs.mkdirSync(dir, { recursive: true });
    return dir;
}

// I file di salvataggio: slot-0.json, slot-1.json, ..., slot-4.json, autosave.json
// Percorso Windows: %APPDATA%\Il Relitto Silente\saves\
```

electron/preload.ts: espone al renderer un'API controllata tramite contextBridge:

```
// electron/preload.ts
contextBridge.exposeInMainWorld('electronAPI', {
    writeSlot:     (id: number, data: string)  => ipcRenderer.invoke('storage:writeSlot',
id, data),
    readSlot:      (id: number)                => ipcRenderer.invoke('storage:readSlot',
id),
    writeSettings: (data: string)              =>
ipcRenderer.invoke('storage:writeSettings', data),
    readSettings:  ()                          =>
ipcRenderer.invoke('storage:readSettings'),
});
```

services/storageService.ts: fornisce funzioni di alto livello al renderer, con logica di fallback su localStorage per l'esecuzione in modalità browser:

```
// services/storageService.ts — dualità Electron/Web
export async function writeSlotData(id: number, data: string): Promise<void> {
    if (window.electronAPI) {
        await window.electronAPI.writeSlot(id, data);
    } else {
        const key = id === -1 ? 'relitto-autosave' : `relitto-slot-${id}`;
        localStorage.setItem(key, data);
    }
}
```

Questa architettura garantisce che il codice dell'applicazione sia completamente indipendente dall'ambiente di esecuzione: lo stesso identico codice React funziona sia in Electron (con persistenza su filesystem) sia in un browser standard (con persistenza su localStorage), senza alcuna modifica.

9.10 Il Formato dei Salvataggi

I dati di gioco vengono serializzati tramite JSON.stringify(playerState). Il PlayerState, essendo composto da tipi JSON-serializzabili (stringhe, numeri, booleani, array e oggetti semplici), non richiede trasformazioni particolari per la serializzazione.

Un file di salvataggio tipico ha dimensioni dell'ordine di 1–3 KB, a seconda della lunghezza dell'inventario, del numero di flag impostati e del numero di stanze visitate. Questo significa che sei file di salvataggio (cinque slot + autosave) occupano complessivamente meno di 20 KB su disco — trascurabile rispetto alla dimensione dell'applicazione.

La deserializzazione avviene tramite JSON.parse(savedData) as PlayerState. Non è presente validazione dello schema al momento del caricamento: questa è una limitazione nota *(cfr. Sezione XI.3)*.

9.11 La Schermata di Pausa

La schermata di pausa (PauseOverlay, 112 LOC) è accessibile tramite F9 o ESC in qualsiasi momento durante la partita. Offre cinque opzioni navigabili sia con i tasti numerici 1–5 sia con il mouse:

1. **Continua** — chiude la pausa e torna al gioco
2. **Salva** — apre il selettore di slot per il salvataggio manuale

3. **Carica** — apre il selettore di slot per il caricamento
4. **Ricomincia** — avvia una nuova partita (con richiesta di conferma)
5. **Esci** — chiude l'applicazione (con richiesta di conferma)

La schermata di pausa può essere chiusa anche ripremendo F9 o ESC, ripristinando il comportamento atteso dagli utenti di sistemi di pausa standard.

Sezione X — Il Processo di Sviluppo e lo Sviluppo Assistito da IA

10.1 Panoramica del Processo

Lo sviluppo de *Il Relitto Silente* ha attraversato due fasi tecnicamente e metodologicamente distinte, ciascuna caratterizzata da strumenti, approcci e risultati diversi. La progressione da prototipo a prodotto finito riflette non soltanto la maturazione del progetto, ma anche l'evoluzione degli strumenti di sviluppo assistito da intelligenza artificiale disponibili nel periodo 2024–2026.

La documentazione di questo processo ha un valore che va oltre la specificità del progetto: *Il Relitto Silente* è uno dei primi esempi documentati di software creativo complesso — un'avventura testuale con meccaniche originali, narrazione strutturata e audio procedurale — sviluppato in modo sistematico con l'ausilio di modelli di linguaggio di grandi dimensioni. Le osservazioni metodologiche che emergono da questa esperienza potrebbero essere utili per chi affronta progetti simili.

10.2 Fase I: Prototipazione con Google AI Studio App Builder

Il progetto ha preso forma inizialmente nell'ecosistema di Google, utilizzando lo strumento AI Studio App Builder come ambiente di prototipazione rapida. In questa fase, l'obiettivo era esplorare la fattibilità del concept — verificare che un'avventura testuale con le caratteristiche desiderate potesse essere implementata in React/TypeScript con un motore proprietario — senza impegnarsi in scelte architetturali definitive.

La prototipazione rapida supportata dall'AI Builder consentì di:

- Definire la struttura fondamentale del modello dati (PlayerState, Room, Item)
- Implementare le prime stanze del relitto con descrizioni dinamiche
- Verificare la fattibilità del sistema di normalizzazione dell'input
- Prototipare l'interfaccia CRT di base con i suoi effetti visivi

Il risultato di questa fase era un prototipo funzionante ma architetturalmente fragile: il codice cresceva in modo non controllato, le stanze erano definite in file di dimensioni eccessive, e alcune scelte architetturali iniziali — in particolare nella gestione dello stato e nel sistema di matching — mostravano problemi di scalabilità che sarebbero diventati evidenti man mano che il numero di stanze e oggetti aumentava.

10.3 Fase II: Sviluppo con Claude Sonnet 4.6 tramite Claude Code

La seconda fase — quella che ha trasformato il prototipo nel prodotto finale — è stata condotta in collaborazione con Claude Sonnet 4.6 tramite Claude Code, lo strumento di sviluppo assistito di Anthropic.

Claude Code opera come un agente in grado di leggere, scrivere e modificare file nel repository, eseguire comandi, interpretare output di errore e mantenere una comprensione contestuale del progetto attraverso l'intera sessione di lavoro. A differenza di un assistente di completamento del codice (come GitHub Copilot), Claude Code gestisce task di sviluppo completi: refactoring architetturale, scrittura di nuove funzionalità, debugging, documentazione.

In questa fase, le attività principali sono state:

Refactoring architetturale: le stanze monolitiche del prototipo sono state suddivise in file separati (game/rooms/*.ts), ciascuno con una responsabilità chiara. Il sistema di matching dei comandi è stato riprogettato per supportare la cache delle espressioni regolari e il fuzzy matching. La gestione dello stato è stata resa immutabile tramite cloneDeep.

Scrittura del contenuto narrativo: i testi delle quindici stanze, delle analisi degli oggetti, degli echi temporali e delle risposte del parser sono stati sviluppati in collaborazione. Il modello ha prodotto testi in italiano in un registro coerente con lo stile del progetto; l'autore ha rivisto, corretto e in molti casi riscritto le proposte, mantenendo la direzione editoriale.

Implementazione del sottosistema audio: il servizio audio procedurale è stato progettato e implementato in questa fase, con una serie di iterazioni sulla scelta delle frequenze, dei profili e dei parametri di fade.

Debugging sistematico: diversi bug architetturali — in particolare nella gestione dei flag, nel sistema di matching degli oggetti con sinonimi, e nella sincronizzazione tra il layer di storage e il sistema audio — sono stati identificati e risolti attraverso sessioni di debugging condivise.

Documentazione: il Manuale Esteso, la guida alla soluzione e il presente documento tecnico sono stati sviluppati con il supporto del modello.

10.4 Il Modello di Collaborazione

La collaborazione con Claude Sonnet 4.6 ha seguito un modello che potremmo definire **direzione umana con esecuzione assistita**. Le caratteristiche di questo modello sono:

L'autore mantiene la direzione: ogni decisione di design — le meccaniche di gioco, la struttura narrativa, la scelta delle stanze, il tono dei testi — è presa dall'autore umano. Il modello non propone autonomamente nuove funzionalità o cambiamenti narrativi non richiesti.

Il modello gestisce la complessità tecnica: la produzione di codice TypeScript corretto, la gestione delle interfacce tra componenti, la risoluzione di errori TypeScript, l'ottimizzazione delle funzioni — queste sono le aree in cui il supporto del modello è stato più prezioso, liberando l'autore dalla pressione tecnica e consentendogli di concentrarsi sulle scelte di design.

La revisione è sistematica: ogni blocco di codice prodotto dal modello viene esaminato, verificato rispetto ai requisiti e, quando necessario, corretto. Il modello fa errori — a volte sottili, come la citazione di sinonimi non implementati nel parser, o l'uso erroneo di flag non esistenti — e la revisione è parte integrante del processo.

L'iterazione è rapida: la capacità del modello di produrre, modificare e riprodurre blocchi di codice o testi narrativi in risposta a feedback è il punto di forza principale del processo. Un ciclo di revisione che richiederebbe ore in uno sviluppo tradizionale si completa in minuti.

10.5 Vantaggi Osservati

Alcune osservazioni specifiche sui vantaggi del processo:

Riduzione del costo cognitivo delle implementazioni di routine: la scrittura di codice TypeScript ripetitivo (handler di comandi, definizioni di stanza, strutture di dati) ha un costo cognitivo non trascurabile. Con il supporto del modello, questo costo è drasticamente ridotto: l'autore descrive la funzionalità desiderata, il modello produce il codice, l'autore verifica e integra. L'attenzione cognitiva viene liberata per le decisioni non routine.

Qualità del codice prodotto: il codice generato da Claude Sonnet 4.6 rispetta le convenzioni TypeScript, segue i pattern architetturali stabiliti nel progetto e produce raramente errori di tipo. La qualità è paragonabile a quella di un programmatore esperto.

Capacità di mantenere il contesto: Claude Code mantiene la comprensione dell'intera base di codice attraverso sessioni di lavoro prolungate, il che significa che le modifiche prodotte in una sessione sono coerenti con le scelte architetturali di sessioni precedenti.

10.6 Limitazioni e Criticità Osservate

L'onestà metodologica richiede di documentare anche le limitazioni e i problemi incontrati:

Allucinazioni su fatti specifici del codice: in alcune occasioni, il modello ha prodotto documentazione o testi che affermavano l'esistenza di funzionalità non implementate nel codice (comandi non riconosciuti dal parser, sinonimi inesistenti). Questi errori richiedono una verifica sistematica rispetto al codice sorgente — un processo che è stato condotto nella fase di revisione della documentazione.

Limitazioni di contesto nelle sessioni lunghe: nelle sessioni di lavoro molto estese, la compressione del contesto prodotta dal modello può portare a perdita di coerenza con decisioni prese nelle sessioni precedenti. Il progetto ha mitigato questo problema mantenendo file di memoria persistenti (nel sistema di memoria di Claude Code) che riassumono le decisioni architetturali fondamentali.

Tendenza all'overengineering: il modello, in assenza di indicazioni specifiche sulla semplicità desiderata, tende a proporre soluzioni più elaborate del necessario. La filosofia di sviluppo del progetto — «la giusta quantità di complessità, non di più» — ha richiesto una supervisione attiva per mantenere il codice snello.

10.7 La Versione del Progetto: da 1.0.1 a 1.2.5

La progressione del numero di versione riflette la struttura dello sviluppo iterativo:

v1.0.x: il prototipo iniziale sviluppato con Google AI Studio. Funzionante ma architetturalmente fragile. Circa 5 stanze implementate.

v1.0.10 – **v1.0.16**: prima serie di refactoring in Claude Code. Suddivisione in file separati per stanza, implementazione del sistema audio, correzione dei bug architetturali maggiori. Completamento delle 15 stanze.

v1.1.x: seconda serie di iterazioni. Implementazione del sistema di traduzione progressiva, del Sintonizzatore di Frequenza e degli echi temporali. Raffinamento del parser con fuzzy matching. Prima versione del sistema HINT.

v1.2.x: fase di polish. Aggiunta della schermata di statistiche, del diario di bordo, del sistema di salvataggio con slot multipli. Revisione sistematica di tutti i testi narrativi. Prima Edizione pubblica: v1.2.5.

10.8 Riflessioni sull'Autorialità nello Sviluppo Assistito da IA

Lo sviluppo assistito da modelli di linguaggio pone domande di autorialità che meritano una riflessione esplicita, in particolare nel contesto di un'opera creativa come un'avventura testuale.

Il modello è un interlocutore di straordinaria utilità tecnica. Non ha un'agenda creativa propria; produce soluzioni coerenti con il contesto e le istruzioni ricevute, ma non impone un punto di vista narrativo, non difende una scelta di design, non ha preferenze estetiche autonome. Questo lo rende uno strumento potente ma, in un certo senso, neutro: la sua produzione è funzione diretta della qualità e della specificità delle istruzioni che riceve.

L'autorialità, in questo modello di collaborazione, rimane interamente nell'intenzione e nella direzione. Chi decide quali stanze esistono, quale storia raccontano, quale atmosfera comunicano, quale finale hanno — queste sono decisioni dell'autore umano. Il modello produce il codice che realizza queste decisioni; non le prende.

Questa distinzione, ovvia nella pratica quotidiana dello sviluppo, diventa meno ovvia quando si considerano i testi narrativi. Le descrizioni delle stanze, i testi degli echi temporali, le risposte del parser — questi sono scritti in collaborazione, con proposte del modello e revisioni dell'autore. Chi ne è l'autore? La risposta pragmatica è: entrambi, con la direzione creativa che rimane all'autore umano. La risposta filosofica è più complicata e rimane aperta.

Quello che sembra chiaro, dal punto di vista di questo progetto, è che l'uso consapevole e critico dei modelli di linguaggio nello sviluppo di software creativo non diluisce l'autorialità umana: la re-indirizza. L'autore non scrive ogni riga di codice o ogni frase di testo, ma decide cosa esiste, come funziona e cosa significa. Questo è — è sempre stato — il lavoro del progettista.

Sezione XI — Analisi Critica, Metriche e Limitazioni

11.1 Metriche del Progetto

La Tab. 11.1 riepiloga le metriche quantitative principali del progetto alla versione 1.2.5.

Tab. 11.1 — Metriche del progetto v1.2.5

Categoria	Metrica	Valore
Codice sorgente	LOC TypeScript/TSX totali	~5.200
	LOC motore narrativo (gameLogic.ts)	781
	LOC applicazione principale (App.tsx)	827
	LOC definizioni stanze (rooms/*.ts)	1.629
	LOC componenti UI (components/*.tsx)	1.401
	LOC servizio audio (audioService.ts)	266
	LOC servizio storage (storageService.ts)	91
	LOC processo Electron (main.ts)	81
	LOC tipi centrali (types.ts)	72
	LOC foglio di stile (index.css)	~7.000
Contenuto	Stanze implementate	15
	Echi temporali	11
	Profili audio ambientali	5
	Slot di salvataggio (escluso autosave)	5
	Chiavi Simboliche (obiettivi principali)	3
	Artefatti linguistici (traduzione)	4
Dipendenze	Dipendenze di runtime	4
	Dipendenze di sviluppo	10
Build	Versione	1.2.5

Categoria	Metrica	Valore
	Dimensione installazione (Windows)	~120 MB
	Formato distribuzione Windows	NSIS Installer

La metrica delle dipendenze di runtime merita un commento. Con sole quattro dipendenze (React, React DOM, lodash.clonedeep, @fontsource/press-start-2p), il progetto mantiene un footprint esterno minimo. Tutte le funzionalità core — parser, audio, storage, interfaccia — sono implementate senza librerie specializzate. Questa scelta aumenta la manutenibilità a lungo termine e riduce il rischio di incompatibilità tra dipendenze.

11.2 Punti di Forza del Progetto

Coerenza estetica e narrativa: l'integrazione tra interfaccia visiva, sistema audio e contenuto narrativo produce un'esperienza coerente dall'inizio alla fine. L'estetica CRT non è una scelta superficiale: è supportata da un sistema audio progettato per lo stesso medium, da un'interfaccia funzionale coerente con quella estetica, e da testi narrativi che tematizzano esplicitamente il rapporto tra tecnologia e memoria.

Qualità del parser: la funzione di normalizzazione dell'input, la gestione dei sinonimi, il fuzzy matching e il sistema di feedback sugli errori producono un parser che minimizza la frustrazione senza sacrificare la precisione. La capacità del parser di gestire articoli, pronomi dimostrativi e preposizioni dell'italiano è particolarmente significativa per una produzione in lingua italiana.

Sistema audio originale: la sintesi procedurale al posto dei file audio preregistrati è una scelta tecnica non comune e artisticamente coerente. Il risultato è un'atmosfera sonora che non ha precedenti nel genere per originalità del metodo.

Architettura scalabile: la separazione tra definizioni di stanza, motore narrativo e interfaccia garantisce che nuove stanze o nuove funzionalità possano essere aggiunte senza modificare il codice core. L'aggiunta di una sedicesima stanza richiederebbe soltanto la creazione di un nuovo file in game/rooms/ e l'aggiornamento di gameData.ts.

Dualità Electron/Web: la capacità di eseguire il progetto sia come applicazione desktop sia nel browser, senza modifiche al codice, è un vantaggio significativo per il testing e per potenziali usi futuri.

11.3 Limitazioni Tecniche

Assenza di validazione dei salvataggi: i dati di gioco vengono deserializzati con un semplice JSON.parse senza validazione dello schema. Un file di salvataggio corrotto o manipolato potrebbe produrre comportamenti imprevedibili. L'aggiunta di una funzione di

validazione che verifichi la presenza e il tipo corretto di tutti i campi del PlayerState prima del caricamento sarebbe un miglioramento significativo.

Nessun sistema di undo: una volta che il giocatore esegue un'azione che modifica lo stato del gioco, non c'è modo di tornare indietro senza caricare un salvataggio. Il meccanismo di cloneDeep a ogni comando renderebbe relativamente semplice l'implementazione di un buffer di stati per l'undo.

Mappa ASCII non interattiva: la mappa progressiva è visualizzata come testo ASCII nel terminale, il che la rende difficile da leggere su schermi con risoluzione inferiore. Una mappa grafica — anche minimalista, coerente con l'estetica CRT — migliorerebbe significativamente la navigabilità per i giocatori meno familiari con le avventure testuali.

Nessuna accessibilità per non-vedenti: il gioco è interamente testuale, ma l'interfaccia grafica (effetti CRT, font pixel, colori) non è compatibile con i reader per screen. Un tema ad alto contrasto e la compatibilità con ARIA potrebbero estendere significativamente la platea di utenti.

Dimensione del bundle: la dimensione dell'installazione (~120 MB) è dominata dal runtime Electron (Chromium + Node.js), non dal codice dell'applicazione. Questa è una limitazione strutturale di Electron che non può essere eliminata senza cambiare il framework di distribuzione.

11.4 Limitazioni Narrative e di Design

Linearità della fase finale: l'Atto III del gioco (dal Ponte di Comando al Santuario Centrale) è quasi completamente lineare, senza enigmi o ramificazioni. Questa scelta è narrativamente motivata — la fase finale è un momento di resa dei conti, non di esplorazione — ma riduce la densità interattiva dell'esperienza conclusiva.

Echi non uniformemente distribuiti: quattro delle quindici stanze (Scafo Esterno, Arca della Memoria, Anticamera del Santuario, Santuario Centrale) non hanno echi temporali. La distribuzione attuale lascia alcune aree del relitto più «silenziose» di quanto la premessa narrativa suggerirebbe.

Sistema HINT non localizzato: i messaggi del sistema HINT sono scritti in un registro leggermente diverso dal testo narrativo delle stanze — più diretto e meno letterario. Questo crea una dissonanza percepibile tra il tono del gioco e quello dei suggerimenti.

11.5 Confronto con lo Stato dell'Arte

Il Relitto Silente si confronta con i prodotti contemporanei del genere su alcune dimensioni rilevanti:

Rispetto ai giochi Twine: Twine offre una ramificazione narrativa molto più sviluppata (sistemi di scelta multipla, variabili di testo, link ipertestuali) ma non ha un sistema di parser.

Il Relitto Silente mantiene la profondità interattiva del parser classico, rinunciando alla ramificazione narrativa fine.

Rispetto ai giochi Inform: Inform produce avventure testuali con sistemi di simulazione del mondo molto più completi (fisica degli oggetti, relazioni spaziali dettagliate, NPC con comportamento autonomo). *Il Relitto Silente* adotta un modello deliberatamente più limitato, che paga il prezzo della profondità simulativa con la semplicità architetturale e la controllabilità dell'esperienza narrativa.

Rispetto alla produzione IF contemporanea in inglese: il settore anglofono dell'IF ha prodotto opere di grande complessità tecnica e narrativa negli ultimi dieci anni. La produzione italiana è sensibilmente più ridotta. *Il Relitto Silente* si posiziona come uno dei pochi titoli italiani contemporanei con un motore parser, un'identità estetica definita e una narrazione originale.

11.6 Sicurezza e Robustezza

Injection nel parser: il parser opera su input testuale dell'utente, ma la normalizzazione e il matching avvengono interamente in memoria, senza esecuzione di codice dinamico (eval o new Function). L'input dell'utente non può influenzare l'esecuzione del programma al di là di quanto previsto dall'interfaccia del parser. Non esiste rischio di injection.

XSS nell'output HTML: le funzioni che producono markup HTML (getInventarioHtml, getMappa) operano su dati interni dello stato (nomi di oggetti e stanze definiti nel codice sorgente), non su input dell'utente. Non esiste rischio di XSS.

IPC Electron: il bridge IPC espone soltanto operazioni di lettura e scrittura di file JSON in directory controllate (%APPDATA%\Il Relitto Silente\). Non espone API per l'esecuzione di comandi shell o l'accesso a percorsi arbitrari del filesystem.

Sezione XII — Conclusioni e Sviluppi Futuri

12.1 Sintesi del Contributo

Il Relitto Silente ha dimostrato che è possibile realizzare, nell'arco di un periodo di sviluppo ragionevole e con le risorse di un singolo sviluppatore, un'avventura testuale in lingua italiana con un motore proprietario, un sottosistema audio originale e una narrativa coerente e approfondita. Il progetto si è configurato come un caso pratico di convergenza tra tradizione del medium testuale e tecnologie web contemporanee.

Le contribuzioni tecniche principali del progetto sono:

1. **Un sistema di parsing per l'italiano** che gestisce articoli, preposizioni, sinonimi e fuzzy matching in modo robusto, con un codice compatto (~50 righe di normalizzazione effettiva) e facilmente estendibile.
2. **Un modello di stanza con descrizioni dinamiche** che permette alle descrizioni ambientali di riflettere lo stato del gioco in modo narrativamente coerente, senza ricorrere a sistemi di template separati.
3. **Un sistema di traduzione progressiva** che intreccia la meccanica del gioco con la dimensione linguistica del mondo alieno, producendo un arco narrativo che motiva l'esplorazione sistematica.
4. **Un sottosistema audio procedurale** basato esclusivamente sulle Web Audio API, senza file preregistrati, con cinque profili ambientali distinti e un sistema di fade coerente con le transizioni narrative.
5. **Un layer di persistenza duale** che astrae le differenze tra l'ambiente Electron e quello browser, consentendo lo sviluppo e il testing in entrambi i contesti senza modifiche al codice.
6. **Una documentazione tecnica sistematica** che descrive l'architettura del sistema con il livello di dettaglio necessario per la comprensione, la manutenzione e l'eventuale estensione del progetto.

12.2 Riflessioni sul Medium

L'avventura testuale è un medium che impone al giocatore di partecipare attivamente alla costruzione del mondo. Non c'è grafica che anticipi l'immaginazione; c'è soltanto il testo, che indica una direzione, e la mente che costruisce il paesaggio. Questo è, al tempo stesso, la limitazione più evidente e la forza più grande del genere.

Il risultato di questa costruzione è, per definizione, unico: ogni giocatore abitua lo spazio in modo diverso, attribuendo colori, odori, proporzioni che nessun designer ha imposto. L'avventura testuale è, tra tutti i generi videoludici, il più fondamentalmente collaborativo: il designer fornisce le coordinate, il giocatore costruisce il mondo.

Questa riflessione non è solo una considerazione estetica. Ha implicazioni pratiche per il design: ogni elemento testuale che viene scritto nella descrizione di una stanza non è soltanto informazione, è materiale per l'immaginazione del giocatore. La scelta di descrivere una stanza come «fredda» invece di «frigida», di usare «silenzio» invece di «assenza di suono», di menzionare «l'odore di ozono» invece di «un odore elettrico» — queste sono scelte che plasmano la costruzione mentale del giocatore in modo preciso e non reversibile.

Il Relitto Silente è stato progettato con questa consapevolezza. Ogni parola nei testi del gioco è lì per un motivo.

12.3 Sviluppi Futuri

Diverse direzioni di sviluppo sono state identificate nel corso del progetto ma non implementate nella versione 1.2.5:

Localizzazione in lingua inglese: la struttura del motore è sufficientemente modulare da supportare una versione inglese del gioco con modifiche limitate al codice. Le sfide principali sarebbero la riscrittura della funzione di normalizzazione (che gestisce specificamente l'italiano) e la traduzione di tutti i testi narrativi, mantenendo il tono e l'atmosfera dell'originale.

Validazione dei salvataggi: l'aggiunta di una funzione di validazione dei dati caricati risolverebbe la limitazione documentata nella Sezione XI.3, aumentando la robustezza del sistema.

Sistema di undo: l'implementazione di un buffer degli ultimi N stati consentirebbe al giocatore di annullare gli ultimi comandi, una funzionalità standard in molti giochi Inform moderni.

Mappa grafica interattiva: la sostituzione della mappa ASCII con una rappresentazione grafica minimalista (coerente con l'estetica CRT) migliorerebbe significativamente l'orientamento dei giocatori meno familiari con il genere.

Contenuto aggiuntivo: il sistema delle stanze è progettato per accogliere nuove stanze senza modifiche al motore. Una eventuale espansione narrativa — nuove aree del relitto, nuovi echi, una trama secondaria — richiede soltanto la creazione di file aggiuntivi in game/rooms/ e l'aggiornamento dei punti di connessione nel grafo delle stanze.

Opzioni di accessibilità: l'aggiunta di un tema ad alto contrasto, una dimensione del testo regolabile e la compatibilità con screen reader consentirebbe di estendere la platea di utenti a chi ha difficoltà visive.

12.4 Il Progetto come Caso di Studio

Al di là del suo valore come gioco, *Il Relitto Silente* si propone come caso di studio per alcune questioni metodologiche aperte:

Sviluppo mono-sviluppatore con assistenza IA: il progetto dimostra che un singolo sviluppatore, con l'ausilio sistematico di modelli di linguaggio, può completare in tempi ragionevoli un progetto software di complessità media-alta. Le condizioni di questo risultato — la qualità della direzione progettuale, la sistematicità della revisione, la coerenza delle scelte architetturali — meritano di essere studiate in modo più rigoroso.

Autorialità creativa nello sviluppo assistito: la questione di dove risieda l'autorialità in un progetto sviluppato con l'ausilio significativo di un modello generativo rimane aperta. Il presente progetto offre un caso concreto su cui articolare questa discussione.

Il medium testuale nel contesto dell'IA generativa: c'è una certa ironia nel fatto che un'avventura testuale — un medium fondato sul testo — venga sviluppata con strumenti di generazione testuale. La riflessione su questa convergenza, e sulle implicazioni per il futuro del medium, è lasciata al lettore.

12.5 Nota Finale

Il testo che chiude il Capitolo XI del Manuale Esteso de *Il Relitto Silente* dice:

«Il relitto sembrava silente. Non lo era mai stato davvero.»

È una frase sul gioco. Ma potrebbe essere una frase su qualsiasi cosa si pensi dimenticata o superata: un linguaggio, un medium, un modo di fare le cose. Il silenzio, a volte, è soltanto l'attesa di qualcuno che si prenda il tempo di ascoltare.

Il Relitto Silente è stato fatto per ascoltare, e per far ascoltare. Che ci sia riuscito è una domanda la cui risposta dipende, come sempre, da chi lo attraversa.

Riferimenti Bibliografici e Sitografici

Riferimenti Bibliografici

Avventura Testuale e Interactive Fiction

Anderson, T. & Galley, M. (2004). *The History of Zork*. New Zork Times, Infocom fan newsletter archive. Ristampa digitale: Interactive Fiction Archive.

Bernstein, M. (1998). *Patterns of Hypertext*. Proceedings of the Ninth ACM Conference on Hypertext and Hypermedia (HyperText '98), pp. 21–29. ACM Press.

Crowther, W. (1975). *ADVENTURE* (Colossal Cave Adventure). Programma per PDP-10. Versione estesa: Don Woods, 1977.

Montfort, N. (2003). *Twisty Little Passages: An Approach to Interactive Fiction*. MIT Press. ISBN 978-0-262-13436-5.

Nelson, G. (2001). *The Inform Designer's Manual* (quarta edizione). The Interactive Fiction Library. Disponibile in forma digitale presso: inform-fiction.org.

Ong, W. J. (1982). *Orality and Literacy: The Technologizing of the Word*. Methuen. ISBN 0-416-71380-0. (Per la riflessione sul testo come medium di costruzione del pensiero.)

Short, E. (2009). *Writing for the Interactive Fiction Parser*. Essay disponibile online. (Nota: Emily Short è autrice di interactive fiction e teorica del genere; i suoi saggi sono distribuiti sul suo blog personale.)

Wardrip-Fruin, N. (2009). *Expressive Processing: Digital Fictions, Computer Games, and Software Studies*. MIT Press. ISBN 978-0-262-23263-0.

Tecnologie Web e Architettura Software

Abramov, D. & Clark, A. (2015–). *React: A JavaScript library for building user interfaces*. Meta Open Source. Documentazione ufficiale: react.dev.

Chromium Project (2008–). *Chromium Browser*. Google LLC. Codice sorgente: chromium.googlesource.com.

Electron (2013–). *Electron: Build Cross-Platform Desktop Apps with JavaScript, HTML, and CSS*. OpenJS Foundation. Documentazione: electronjs.org.

Mozilla Developer Network (2013–). *Web Audio API specification and documentation*. MDN Web Docs: developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API.

Vite (2020–). *Vite: Next Generation Frontend Tooling*. Evan You e collaboratori. Documentazione: vitejs.dev.

W3C Audio Working Group (2012–2021). *Web Audio API: W3C Recommendation*. World Wide Web Consortium.

Sviluppo Assistito da IA

Anthropic (2024–). *Claude Code: Agentic coding in your terminal*. Anthropic PBC. Documentazione: docs.anthropic.com/claude-code.

Chen, M., et al. (2021). *Evaluating Large Language Models Trained on Code*. arXiv:2107.03374. (Fondazione del campo Copilot / codegen con LLM.)

Vaswani, A., et al. (2017). *Attention Is All You Need*. Advances in Neural Information Processing Systems 30 (NeurIPS 2017). (L'architettura Transformer su cui si basano i moderni LLM.)

Psicoacustica e Sintesi Sonora

Backus, J. (1977). *The Acoustical Foundations of Music* (seconda edizione). Norton. ISBN 0-393-09071-0. (Fondamenti acustici su cui si basano le scelte di frequenza per i profili audio.)

Roads, C. (1996). *The Computer Music Tutorial*. MIT Press. ISBN 978-0-262-68082-0.

Rossing, T. D., Moore, F. R. & Wheeler, P. A. (2001). *The Science of Sound* (terza edizione). Addison-Wesley. ISBN 0-8053-8565-7.

Riferimenti Sitografici

Interactive Fiction

Interactive Fiction Archive — ftp.ifarchive.org Archivio storico del software di interactive fiction, inclusi i classici Infocom e la produzione hobbistica degli ultimi quarant'anni.

Interactive Fiction Community Forum — intfiction.org Forum della comunità internazionale di IF: discussioni su design, sviluppo, teoria e critica.

IFDB — Interactive Fiction Database — ifdb.org Database curato delle opere di interactive fiction con recensioni, classifiche e metadati.

Emily Short's Interactive Storytelling — emshort.blog Blog di Emily Short: saggi di design, critica e teoria dell'interactive fiction.

The Digital Antiquarian — filfre.net Storia dettagliata e critica dell'industria videoludica, con copertura approfondita dell'era delle avventure testuali.

Tecnologie Utilizzate nel Progetto

React Documentation — react.dev Documentazione ufficiale di React 19.

TypeScript Documentation — typescriptlang.org Documentazione ufficiale di TypeScript 5.9.

Vite Documentation — vitejs.dev Documentazione di Vite 8.

Electron Documentation — electronjs.org/docs Documentazione di Electron 41.

Web Audio API — MDN — developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API Specifica e documentazione della Web Audio API.

Tailwind CSS Documentation — tailwindcss.com Documentazione di Tailwind CSS 4.

lodash.clonedeep — lodash.com/docs/#cloneDeep Documentazione della funzione cloneDeep di lodash.

Produzione Italiana di Interactive Fiction

GiocoIF — Comunità italiana di Interactive Fiction — giocoif.org Portale della comunità italiana di interactive fiction: archivio di titoli, strumenti, tutorial in italiano.

Tutti i riferimenti sitografici sono stati verificati alla data di completamento del presente documento (marzo 2026). La disponibilità di risorse online è soggetta a variazione nel tempo.

Appendice A — Vocabolario Completo dei Comandi

Elenco sistematico di tutti i comandi riconosciuti dal parser de Il Relitto Silente v1.2.5, con le relative abbreviazioni, sinonimi e note di comportamento.

A.1 Comandi di Osservazione

Comando	Abbreviazione	Sinonimi	Descrizione
GUARDA	L	—	Ridescrive la stanza corrente
ESAMINA [oggetto]	X [oggetto]	—	Mostra la descrizione visiva di un oggetto
ANALIZZA [oggetto]	—	LEGGI [oggetto]	Esegue un'analisi tecnica con lo scanner
TOCCA [oggetto]	—	—	Risposta tattile contestuale all'oggetto

Note: LEGGI viene convertito internamente in ANALIZZA dalla funzione normalizeCommand. Non è un alias della funzione ESAMINA. I due comandi producono risposte diverse: ESAMINA descrive l'aspetto visivo; ANALIZZA rivela composizione, struttura e dati tecnici. In molti casi, l'analisi include informazioni non rilevabili visivamente.

A.2 Comandi di Navigazione

Comando	Abbreviazione	Note
VAI NORD	N	Si sposta verso nord se l'uscita è disponibile
VAI SUD	S	Si sposta verso sud
VAI EST	E	Si sposta verso est
VAI OVEST	O	Si sposta verso ovest. Riconosce anche W (convenzione anglosassone)
VAI ALTO	U	Sale al livello superiore (upstairs)
VAI BASSO	D	Scende al livello inferiore. Riconosce anche GIU
ENTRA [luogo]	—	Alternativa per ingressi contestuali

Note: i comandi di navigazione riconoscono varianti con articoli e preposizioni: VAI VERSO NORD, VAI A NORD, VAI AL NORD producono tutti lo stesso risultato. Se l'uscita richiesta è bloccata da una precondizione non soddisfatta, il sistema restituisce un messaggio descrittivo che indica perché il passaggio non è accessibile.

A.3 Comandi di Inventario e Oggetti

Comando	Abbreviazione	Sinonimi	Descrizione
INVENTARIO	I	INV	Mostra la lista degli oggetti in possesso
PRENDI [oggetto]	—	RACCOGLI [oggetto]	Inserisce un oggetto nell'inventario
USA [oggetto]	—	INDOSSA [oggetto]	Usa un oggetto (senza bersaglio specifico)
USA [oggetto] SU [bersaglio]	—	CON, SOPRA come alternative a SU	Usa un oggetto su un bersaglio

Note: INDOSSA viene convertito internamente in USA. La scelta progettuale è che «indossare un oggetto» e «usare un oggetto» producono lo stesso effetto nella logica del gioco: attivare l'handler onUse dell'oggetto. Le preposizioni alternative riconosciute sono: SU, CON, SOPRA, NELL', NELLA, SUL, SULLA, e le loro varianti con articolo. Tutte vengono normalizzate a SU.

A.4 Comandi di Informazione e Diario

Comando	Abbreviazione	Sinonimi	Descrizione
NOTA	—	DIARIO, APPUNTI, LOG	Apre il diario delle scoperte
ECHI	—	—	Mostra la lista degli echi captati
MAPPA	—	—	Mostra la mappa progressiva del relitto
AIUTO	—	HINT, SUGGERIMENTO, CONSIGLIO	Mostra il suggerimento contestuale
ASPETTA	Z	—	Lascia passare un turno senza azioni

Note: il comando AIUTO è identico a HINT, SUGGERIMENTO e CONSIGLIO nella logica interna. Il sistema HINT analizza posizione, inventario e flag di avanzamento per produrre un suggerimento contestuale, non un messaggio generico.

A.5 Comandi di Gestione del Gioco

Comando	Funzione equivalente	Nota
SALVA	Tasto F4	Apre il selettore di slot per il salvataggio manuale

Comando	Funzione equivalente	Nota
CARICA	Tasto F3	Apre il selettore di slot per il caricamento
AUDIO	Overlay audio	Toggle rapido dell'audio ambientale
MUSICA	Overlay audio	Sinonimo di AUDIO
IMPOSTAZIONI	Overlay audio	Apre il pannello di controllo del volume
PULISCI	—	Cancella l'output del terminale

Sinonimi di PULISCI: CLEAR.

A.6 Comandi Speciali e Contestuali

Comando	Descrizione
PARLA (o PARLA CON [personaggio])	Interagisce con un personaggio presente nella stanza
APRI [oggetto]	Tenta di aprire un oggetto. Produce un risultato specifico contestualmente
USA SINTONIZZATORE	Attiva il Sintonizzatore di Frequenza per captare echi (disponibile solo con lo strumento in inventario)

A.7 Tabella di Normalizzazione degli Articoli

La Tab. A.1 riporta gli articoli e i dimostrativi rimossi dalla funzione normalizeCommand prima del matching.

Tab. A.1 — Token rimossi dalla normalizzazione

Categoria	Token rimossi
Articoli determinativi	il, lo, la, i, gli, le, l'
Articoli indeterminativi	un, uno, una, un'
Dimostrativi	quel, quello, quella, quei, quegli, quelle, questo, questa, questi, queste
Preposizioni normalizzate a su	con, sopra

A.8 Esempi di Normalizzazione

La Tab. A.2 illustra il comportamento della normalizzazione su input reali.

Tab. A.2 — Esempi di input normalizzati

Input originale	Output normalizzato	Nota
USA IL DISCO SULL'ALTARE	usa disco su altare	Articolo rimosso, preposizione articolata normalizzata
usa quel disco su quell'altare	usa disco su altare	Dimostrativo rimosso
ESAMINA la teca di cristallo	esamina teca di cristallo	Articolo rimosso, resto invariato
Leggi l'iscrizione	analizza iscrizione	Articolo rimosso, LEGGI → ANALIZZA
indossa la tuta spaziale	usa tuta spaziale	Articolo rimosso, INDOSSA → USA
x disco	esamina disco	x → esamina
n	nord	Abbreviazione espansa
esamnia disco	—	Non normalizzato; fuzzy matching suggerisce «esamina»
analizzà pannello	analizza pannello	Accento rimosso dalla normalizzazione Unicode

A.9 Verbi nel Vocabolario di Fuzzy Matching

Il fuzzy matching (distanza di Levenshtein ≤ 2) opera sull'insieme dei verbi riconosciuti:

guarda, esamina, analizza, tocca, prendi, raccogli, usa, indossa, leggi, vai, nord, sud, est, ovest, alto, basso, entra, inventario, mappa, nota, diario, echi, aiuto, hint, aspetta, salva, carica, audio, impostazioni, pulisci, parla, apri

Un input come prnedi (distanza 2 da prendi) produrrà il suggerimento «Forse intendevi "prendi"?» prima del messaggio di errore.

Appendice B — Struttura del Progetto e Metriche di Codice

B.1 Albero del Repository

L'albero seguente riporta la struttura completa del repository alla versione 1.2.5, con l'esclusione delle directory generate (node_modules/, dist/, dist-electron/, release/) e dei file di lock.

```
gioco/
│
├── game/                              # Logica di gioco
│   ├── gameLogic.ts                   # Motore narrativo principale      [781 LOC]
│   ├── gameData.ts                    # Aggregatore stanze                [42 LOC]
│   ├── echoData.ts                    # Testi e mapping echi              [36 LOC]
│   └── rooms/                         # Definizioni delle 15 stanze
│       ├── plancia.ts                 # Plancia della Santa Maria         [46 LOC]
│       ├── stiva.ts                   # Stiva                            [119 LOC]
│       ├── corridoioPrincipale.ts     # Corridoio Principale             [224 LOC]
│       ├── scafoEsterno.ts            # Scafo Esterno                     [80 LOC]
│       ├── cameraDiCompensazione.ts   # Camera di Compensazione           [86 LOC]
│       ├── santuarioDelSilenzio.ts    # Santuario del Silenzio           [110 LOC]
│       ├── scriptorium.ts             # Scriptorium                       [75 LOC]
│       ├── arcaDellaMemoria.ts        # Arca della Memoria               [101 LOC]
│       ├── serraMorente.ts            # Serra Morente                    [154 LOC]
│       ├── arcaBiologica.ts           # Arca Biologica                   [143 LOC]
│       ├── alloggiEquipaggio.ts       # Alloggi dell'Equipaggio          [100 LOC]
│       ├── laboratoriRisonanza.ts     # Laboratori di Risonanza           [96 LOC]
│       ├── ponteDiComando.ts          # Ponte di Comando                 [190 LOC]
│       ├── santuarioCentrale.ts       # Santuario Centrale                [81 LOC]
│       └── anticameraSantuario.ts     # Anticamera del Santuario          [24 LOC]
│
├── components/                        # Componenti React
│   ├── AudioSettingsOverlay.tsx       # Overlay impostazioni audio       [170 LOC]
│   ├── BootScreen.tsx                 # Schermata di boot                [129 LOC]
│   ├── CommandLine.tsx                # Input da riga di comando         [123 LOC]
│   ├── ErrorBoundary.tsx              # Gestore errori React              [87 LOC]
│   ├── GameOverScreen.tsx             # Schermata di fine gioco           [55 LOC]
│   ├── IntroScreen.tsx                # Sequenza cinematica introduttiva [254 LOC]
│   ├── PauseOverlay.tsx               # Menu di pausa                    [112 LOC]
│   ├── SaveLoadOverlay.tsx            # Selettore slot salvataggio        [313 LOC]
│   ├── StartScreen.tsx                # Menu principale                   [49 LOC]
│   └── TerminalOutput.tsx             # Output del terminale             [109 LOC]
│
├── services/                          # Servizi trasversali
│   ├── audioService.ts                # Sintesi audio procedurale        [266 LOC]
│   └── storageService.ts              # Layer di persistenza duale        [91 LOC]
│
├── electron/                          # Codice processo principale Electron
│   ├── main.ts                        # Gestore eventi e IPC              [81 LOC]
│   ├── preload.ts                     # Bridge di sicurezza               [16 LOC]
│   ├── build-helper.js                # Script di supporto alla build
│   └── tsconfig.json                  # Configurazione TypeScript Electron
│
├── docs/
│   └── giocatore/
│       ├── LEGGIMI.txt                # Istruzioni per il giocatore
│       ├── MANUALE.md                 # Manuale rapido
│       ├── CARATTERISTICHE.txt        # Caratteristiche del gioco
│       └── SOLUZIONE.md               # Guida completa alla soluzione
│
├── MANUALE-ESTESO/                    # Manuale esteso (12 capitoli .md)
```

```
│
├── PAPER/                          # Il presente documento tecnico
│
├── App.tsx                         # Componente React principale      [827 LOC]
├── index.tsx                       # Entry point React
├── types.ts                        # Definizioni TypeScript             [72 LOC]
├── index.css                       # Foglio di stile principale     [~7.000 LOC]
├── index.html                      # Template HTML
├── vite.config.ts                  # Configurazione Vite                [21 LOC]
├── tsconfig.json                   # Configurazione TypeScript          [30 LOC]
├── package.json                    # Configurazione npm e build
├── metadata.json                   # Metadati applicazione
└── favicon.png                     # Icona applicazione
```

B.2 Metriche di Codice per Modulo

Tab. B.1 — Distribuzione delle linee di codice per modulo

Modulo	File	LOC	% del totale TS/TSX
Componente principale	App.tsx	827	15,9%
Motore narrativo	game/gameLogic.ts	781	15,0%
Definizioni stanze	game/rooms/*.ts	1.629	31,3%
Componenti UI	components/*.tsx	1.401	26,9%
Servizio audio	services/ audioService.ts	266	5,1%
Servizio storage	services/ storageService.ts	91	1,7%
Processo Electron	electron/main.ts	81	1,6%
Dati gioco	game/gameData.ts + echoData.ts	78	1,5%
Tipi centrali	types.ts	72	1,4%
Bridge Electron	electron/preload.ts	16	0,3%
Totale		~5.200	100%

Il modulo con la percentuale più alta è l'insieme delle definizioni di stanza (31,3%), il che riflette il fatto che il contenuto narrativo del gioco è rappresentato come codice. Ogni stanza è un file TypeScript che combina testi, logica e strutture dati in modo non separabile senza una refactorizzazione significativa del sistema.

B.3 Dipendenze: Grafo delle Importazioni Principali

Il grafo seguente rappresenta le dipendenze tra i moduli principali. Le frecce indicano «importa da».

```
App.tsx
  ├── types.ts
```

```
  ├── game/gameLogic.ts
  │     ├── types.ts
  │     ├── game/gameData.ts
  │     │     └── game/rooms/*.ts (15 file)
  │     │           └── types.ts
  │     └── game/echoData.ts
  ├── services/audioService.ts
  │     └── services/storageService.ts
  ├── services/storageService.ts
  └── components/*.tsx (10 componenti)
        └── types.ts
```

Il grafo è aciclico (nessuna dipendenza circolare), con types.ts come foglia radice condivisa da tutti i moduli. App.tsx è il nodo di integrazione: importa tutti i servizi e tutti i componenti, ma non viene importato da nessun altro modulo.

B.4 Configurazione di Build Electron-Builder

```
{
  "build": {
    "appId": "com.relittosilente.adventure",
    "productName": "Il Relitto Silente",
    "files": [
      "dist/**/*",
      "dist-electron/**/*"
    ],
    "extraFiles": [
      { "from": "docs/giocatore/LEGGIMI.txt",  "to": "LEGGIMI.txt"  },
      { "from": "docs/giocatore/SOLUZIONE.md", "to": "SOLUZIONE.md" }
    ],
    "directories": { "output": "release" },
    "win": {
      "target": "nsis",
      "icon": "public/app-icon.png"
    },
    "mac": {
      "target": "dmg",
      "icon": "public/app-icon.png"
    },
    "linux": {
      "target": "AppImage",
      "icon": "public/app-icon.png"
    }
  }
}
```

B.5 Script npm

Script	Comando	Descrizione
dev	vite	Dev server Vite su porta 3000 (HMR)
build	vite build	Build produzione per browser
preview	vite preview	Anteprima della build di produzione
electron	wait-on tcp:3000 && electron .	Avvia Electron (dopo che Vite è

Script	Comando	Descrizione
		pronto)
electron:dev	Concurrently: dev + electron	Sviluppo con Vite + Electron in parallelo
electron:build	build + tsc + node build-helper + electron-builder –win	Build installer Windows completo

B.6 Percorsi di Salvataggio su Filesystem (Windows)

Dato	Percorso
Salvataggio slot 0	%APPDATA%\Il Relitto Silente\saves\slot-0.json
Salvataggio slot 1	%APPDATA%\Il Relitto Silente\saves\slot-1.json
Salvataggio slot 2	%APPDATA%\Il Relitto Silente\saves\slot-2.json
Salvataggio slot 3	%APPDATA%\Il Relitto Silente\saves\slot-3.json
Salvataggio slot 4	%APPDATA%\Il Relitto Silente\saves\slot-4.json
Autosave	%APPDATA%\Il Relitto Silente\saves\autosave.json
Impostazioni	%APPDATA%\Il Relitto Silente\settings.json

Il percorso %APPDATA% corrisponde tipicamente a C:\Users\[NomeUtente]\AppData\Roaming\ su sistemi Windows 10/11.

B.7 Struttura del File di Salvataggio

Un file di salvataggio serializza un oggetto PlayerState in JSON. Esempio:

```
{
  "location": "Corridoio Principale",
  "inventory": ["Taglierina Magnetica", "Lastra Dati"],
  "flags": {
    "translationProgress": 4,
    "lastraAnalizzata": true,
    "picked_corridoio_lastra": true,
    "isWearingSuit": true,
    "kitAperto": true
  },
  "echoes": ["echo_plancia"],
  "visitedRooms": [
    "Plancia della Santa Maria",
    "Stiva",
    "Scafo Esterno",
    "Camera di Compensazione",
    "Corridoio Principale"
  ]
}
```

Nota: i flag sono un dizionario di tipo Record<string, boolean | number | string>. Le chiavi dei flag seguono una convenzione non formale: flag booleani usano nomi descrittivi (isWearingSuit, kitAperto); flag di oggetti raccolti usano il prefisso picked_ seguito da

identificatore (picked_corridoio_lastra); flag di echi usano il prefisso echoDetected_ seguito dal nome della stanza.