UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione

Sviluppo di un Serious Game nell'ambito del progetto Digital Water City sulla base di MicropolisJS

Development of a Serious Game in the context of the Digital Water City project based on MicropolisJS

Candidato:
Tommaso Coricelli

Relatore:
Prof. Adriano Mancini

Anno Accademico 2019-2020

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione

Sviluppo di un Serious Game nell'ambito del progetto Digital Water City sulla base di MicropolisJS

Development of a Serious Game in the context of the Digital Water City project based on MicropolisJS

Candidato:
Tommaso Coricelli

Relatore:
Prof. Adriano Mancini

Anno Accademico 2019-2020

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione
Via Brecce Bianche – 60131 Ancona (AN), Italy

Ringraziamenti

Giunto alla fine della stesura di questa tesi, voglio ringraziare innanzitutto il mio relatore, il prof. Adriano Mancini, che mi ha dato l'opportunitá di lavorare ad un progetto interessante riguardo una tematica da me mai trattata in precedenza, per essere sempre stato disponibile a risolvere qualsiasi dubbio e per aver stimolato il continuo miglioramento del progetto.
Doverosi sono i ringraziamenti alla mia famiglia, che mi ha sempre sostenuto, spronato quando c'é stato bisogno e aiutato nei momenti difficili. Loro mi hanno insegnato il vero valore del lavoro e la soddisfazione che ne consegue.
Ringrazio inoltre le mie zie Monia e Silvana per il contributo dato alla revisione della tesi, al fine di migliorare la comprensione degli argomenti tecnici.
Grazie a Eleonora, la mia fidanzata, con la quale ho condiviso gli ultimi sei anni, in lei ho trovato un sostegno e mi ha saputo trasmettere la positivitá necessaria per affrontare i momenti difficili. Come non ringraziare gli amici di una vita, con i quali ho condiviso esperienze indimenticabili.
Infine dedico la tesi all'amico che piú mi manca, Nicholas, scomparso all'inizio del mio percorso universitario e che mi ha accompagnato fino alla fine, rimanendo sempre nel mio cuore.

Ancona, Ottobre 2020

Tommaso Coricelli

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Il progetto

L'obbiettivo di questa tesi é quello di implementare nuove funzionalitá nello sviluppo di un *Serious Game* chiamato *MicropolisJS*. Il gioco in questione é di fatto il porting in linguaggio *javascript* del codice open-source di *SimCity*: storico videogioco gestionale rilasciato per la prima volta nel 1989.

Lo sviluppo del *Serious Game* entra nell'ambito del progetto europeo *digital-water.city*, il cui fine é quello di sensibilizzare la popolazione sull'uso di un bene primario come l'acqua tramite la tecnologia e i nuovi strumenti a nostra disposizione.

Lo scopo pratico del progetto, sviluppato durante il tirocinio, é l'introduzione nel gioco di elementi tipici del mondo del riuso delle acque (*reused water*), come ad esempio il *Waste-Water-Treatment-Plant*, ovvero un impianto di depurazione di acque nere che diventeranno utili per l'uso agricolo. La capacitá di questo impianto di depurare le acque anche per scopi agricoli consente di aumentare la sostenibilitá di una cittá.

In particolare, tale aspetto nel gioco é stato approfondito da me e dal mio collega Massimo Merla; insieme abbiamo arricchito il *gameplay* del gioco con l'introduzione dell'ambiente agricolo e nello specifico abbiamo modellato gli oggetti di tipo `field` e di *indfield*, due tipi di campo che sono poi collegati alla centrale `WWTP` tramite il connettore `channel`. Questa dinamica permetterá al videogiocatore di aumentare la sostenibilitá della cittá guadagnando punteggi piú alti.

1.2 Struttura della tesi

La struttura della tesi prevede una parte introduttiva riguardo ai temi trattati di *gamification* e di *Serious Game* e una parte successiva in cui vengono approfonditi gli strumenti utilizzati e l'approccio a livello di *coding* del progetto. Poi seguono le conclusioni, con uno sguardo ai possibili sviluppi futuri delle proposte avanzate nella tesi.

Capitolo 2

Serious Game e Gamification

2.1 Gamification

La parola *Gamification*, come é facile intuire, deriva dalla parola *game*, cioé gioco, ed esprime l'utilizzo videoludico come veicolo per messaggi e comportamenti trasmessi in maniera efficace all'utente.

A prima vista sembra un approccio leggero e sperimentale, ma in realtà si tratta di uno strumento consolidato, affidabile ed ormai largamente diffuso, perché tramite il suo coinvolgimento attivo l'utente puó raggiungere obbiettivi ed assimilare contenuti importanti.

Possiamo definire la *Gamification* come un insieme di regole mutuate dal mondo dei videogiochi aventi lo scopo di applicare meccaniche ludiche ad attività che non hanno direttamente a che fare con il gioco stesso; in questo modo è possibile influenzare e modificare il comportamento delle persone, favorendo la nascita ed il consolidamento di interesse attivo da parte degli utenti coinvolti verso il messaggio che si è scelto di comunicare, nell'ambito dell'incremento di performance personali o più in generale delle performance d'impresa. Ci sono molti contesti nei quali è possibile applicare quello che possiamo definire il "metodo" *Gamification*: un sito, un servizio, una comunità, un contenuto.[1]

Per raggiungere questi obbiettivi, il processo di *communication design* deve necessariamente essere ripensato in modo da introdurre meccaniche e dinamiche di gioco, aggiungendo ai fattori tradizionali altre componenti trainanti (ancora, mutuate dal mondo del *gaming*) che possano attirare l'interesse dell'utenza, spingendola a tornare su specifici contenuti proposti volontariamente e più volte nell'arco del tempo.

Tali relazioni motivano gli utenti al raggiungimento di obbiettivi predeterminati (es: il miglioramento delle proprie capacità, l'incremento delle performance, ecc...), modificando di fatto il loro comportamento.

2.2 Serious Game

Diversa invece é la definizione di *Serious Game*: ovvero un videogioco con uno scopo che non necessariamente deve essere collegato all'intrattenimento dell'utente, ma che possiede un fine didascalico relativo ai temi che tratta.

Figura 2.1: Grafico che mostra la differenza concettuale tra *Gamification* e *Serious Game* `https://www.1000minds.com/library/processofmcda.jpg`

2.2.1 Multi-Criteria Decision Analysis

L'*analisi decisionale a criteri multipli* o MCDA è una sotto-disciplina della ricerca operativa che valuta in modo esplicito i criteri più contrastanti nei processi decisionali.

L'MCDA si occupa di strutturazione e risoluzione dei problemi di decisione e di pianificazione che coinvolgono più criteri. Lo scopo è quello di sostenere i responsabili delle decisioni che affrontano questi problemi. In genere, non esiste un'unica ottimale soluzione ed è necessario utilizzare le preferenze del decisore a distinguere tra le varie soluzioni possibili.

Questo metodo é facilmente implementabile tramite un *Serious Game*, che potrebbe catturare l'attenzione dei giocatori rendendoli consci di difficoltá che esistono nel mondo reale, sviluppando in loro capacitá decisionali.

Ci sono molti metodi per tradurre una questione reale in un *Serious Game*: si potrebbe sviluppare un'applicazione ad alta verosimiglianza, oppure un'altra soluzione percorribile é quella di aumentare la complessitá step-by-step o, altrimenti, di trattare situazioni reali indipendenti tra loro, abbassando vertiginosamente la complessitá e la verosimiglianza del problema.

La scelta di implementazione dipende soprattutto dal pubblico a cui é rivolto lo sviluppo.[2]

Figura 2.2: *Serious Game* come strumento di apprendimento `https://ars.els-cdn.com/content/image/1-s2.0-S1364815217307661-fx1.jpg`

2.2.2 Vantaggi

I *Serious Games* offrono molteplici vantaggi:

- possono essere sviluppati in maniera molto flessibile e sono quindi estremamente adattabili al contesto all'interno del quale verranno utilizzati;

- consentono un attivo coinvolgimento del fruitore che pertanto diventa protagonista dell'esperienza formativa;

- permettono di veicolare contenuti che, attraverso la formazione classica, risulterebbero molto più difficoltosi (e noiosi) da apprendere;

- stimolano la riflessione e l'interiorizzazione di cambiamenti che solo attraverso l'esperienza diretta possono trovare radicale modificazione;

- forniscono la possibilità di ricreare un ambiente protetto nel quale il fruitore ha la possibilità di mettersi in gioco senza aver paura di sbagliare ma anzi può imparare dai propri errori per non commetterne nella vita reale.[3]

2.3 Digital-water.city

DWC é un progetto europeo che coinvolge piú realtá, sia universitarie che aziendali, e il cui scopo é quello di rivolgere investimenti e ricerca verso le infrastrutture e le soluzioni riguardanti il tema idrico all'interno di ambienti metropolitani[4].

Al momento lo studio è concentrato in cinque capitali europee: Berlino, Copenaghen, Parigi, Sofia e Milano.

Figura 2.3: Logo del progetto

Il capoluogo lombardo, nello specifico, giova di numerose soluzioni digitali, come sistemi di controllo con droni e sensori per l'efficienza delle irrigazioni nei campi e per rilevare eventuali problematiche nei terreni.

Proprio rivolto a Milano é lo sviluppo del *Serious Game* oggetto di questa tesi: il programma pone particolare attenzione alle logiche di *gameplay* in merito al riutilizzo delle acque reflue, tramite l'introduzione del *Waste-Water-Treatment-Plant*, ovvero un impianto di depurazione di acque nere, la cui produzione di fanghi viene utilizzata in campo agricolo come fertilizzante. Lo scopo generale del progetto é, quindi, quello di sensibilizzare circa le tematiche di sostenibilitá e rinnovabilitá delle energie per mezzo della componente ludica che sará introdotta tramite il videogioco.

2.4 MicropolisJS

MicropolisJS é la base su cui é stato costruito il progetto qui trattato;
Opera di **Graeme McCutcheon**, programmatore scozzese che ha contribuito anche alla programmazione del noto browser *Mozilla Firefox*, lo sviluppo del videogioco gestionale qui utilizzato é il *porting* in *JavaScript* del primo capitolo della celeberrima saga *SimCity*, rilasciato da *Elecrtronic Arts* nel 1989.[5]

Lo scopo principale del gioco é quello di costruire con una logica a tessere la propria cittá, riuscendo a farla prosperare con l'utilizzo di fondi limitati e rispettando dinamiche realistiche, come inquinamento e criminalitá. Il vantaggio del *porting* effettuato da *graememcc*, questo il suo nickname, é importante perché il progetto

Figura 2.4: Screenshot di gioco del primo *SimCity*
`https://archivio-gamesurf.tiscali.it/content/articoli/2013/20483/immagini_grandi/0.jpg`

open-source puó essere modificato da chiunque facilmente, grazie all'utilizzo di un linguaggio moderno come *JavaScript*.

Le principali strutture contenute nel gioco sono le seguenti:

residential zona dove vivono i cittadini e dove sono presenti forti meccaniche di immigrazione ed emigrazione in base all'indice di criminalitá ed inquinamento; genera reddito in base a quante persone vivono nell'area;

commercial/industrial strutture che generano inquinamento e che crescono in base al livello di collegamento che hanno con le infrastrutture;

coal/nuclear powerplant stabilimenti che forniscono energia a tutte le costruzioni della cittá e che generano inquinamento, il quale dipende dal tipo di risorsa utilizzata;

wire strumento per collegare le centrali elettriche alle costruzioni;

road/rail infrastrutture che aumentano il punteggio delle strutture adiacenti e di conseguenza aumentano la crescita delle zone che collegano;

fire/police department strutture utili per controllare le variabili di crimine e per affrontare disastri come, ad esempio, incendi;

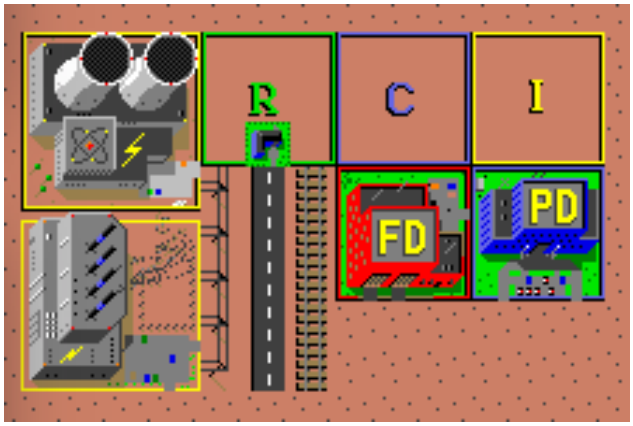

Figura 2.5: Screenshot che mostra le strutture descritte sopra

Come possiamo vedere in dettaglio nella figura 2.5 e piú in generale nella figura 2.6 l'interfaccia di gioco e i modelli utilizzati richiamano molto al *retrogaming*; questo aspetto potrebbe non entusiasmare i piú giovani, oggi abituati a video-giocare prediligendo la grafica spesso a discapito di altri aspetti tecnici.

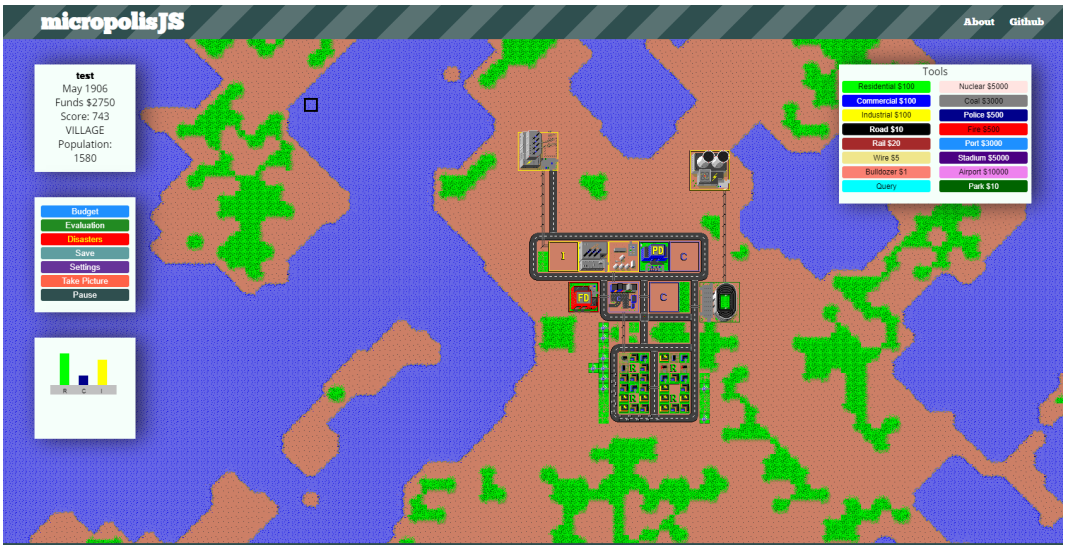

Figura 2.6: Screenshot dell'interfaccia di gioco

Capitolo 3

Strumenti utilizzati e inizializzazione progetto

3.1 Microsoft Visual Studio Code

VS Code é l'IDE scelto per questo tipo di progetto, un ambiente di sviluppo piú leggero rispetto all'originale *Visual Studio*, *cross-platform* compatibile con Windows, Linux e macOS e che permette di evidenziare la sintassi di ciascun linguaggio di programmazione (*Syntax highlighting*). L'IDE in questione integra il supporto per il debugging, un terminale a riga di comando, il controllo *Git*, *IntelliSense* (il completamento automatico delle istruzioni), oltre ad offrire la possibilitá di mantenere aperti piú file, affiancandone il contenuto in più schede e molto altro ancora. Inoltre si tratta di un ambiente di sviluppo orientato ad applicativi *web* e *cloud* con la possibilitá di installare numerose estensioni utili per la programmazione ed il *versioning*.

Figura 3.1: Logo di Visual Studio Code

3.2 Github

La condivisione é stata una parte molto importante dello sviluppo del programma, considerando che questo lavoro ha visto coinvolte due persone, e quindi per ogni progresso effettuato é stata, appunto, necessaria la condivisione del progetto nella sua versione piú aggiornata. Solo in questo modo, infatti, é possibile evitare sovrascritture di file e non si rischia di perdere i propri progressi.

Il principale portale utilizzato per mantenere la condivisione del progetto aggiornata é *Github*, la piattaforma maggiormente in uso per lo sviluppo collaborativo di software, ed oggi utilizzata in tutti quei settori in cui è richiesto il continuo aggiornamento e salvataggio di progetti di sviluppo portati avanti da due o più professionisti. Molti, ad esempio, sfruttano le potenzialità di *Github* per collaborare con colleghi di sedi distaccate nella stesura di documenti o report; altri per scrivere la tesi di laurea senza paura di perdere tutto il lavoro svolto a causa di un black out improvviso o per un disco rigido mal funzionante.

Per comprendere il funzionamento di *Github* é necessario introdurre il suo antenato, *Git*; si tratta di un software di controllo di versione: ciò vuol dire che controlla e gestisce gli aggiornamenti di un progetto senza sovrascrivere nessuna parte del progetto stesso. Venne creato dal papà di *Linux*, **Linus Torvalds**, e dai suoi collaboratori nel corso dello sviluppo del *kernel*: nel caso in cui qualche aggiornamento non avesse dato gli effetti sperati, si poteva sempre tornare indietro e recuperare la versione funzionante senza troppi problemi.[6]

Figura 3.2: Logo di GitHub

3.3 JavaScript

Il linguaggio utilizzato é *JavaScript*: si tratta di un linguaggio di programmazione orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione web lato *client* (esteso poi anche al lato *server*) per la creazione, in siti web e applicazioni web, di effetti dinamici interattivi tramite funzioni di script invocate da eventi innescati a loro volta in vari modi dall'utente sulla pagina web in uso.

Originariamente sviluppato da **Brendan Eich** della *Netscape Communications* con il nome di *Mochan* e successivamente di *LiveScript*, in seguito è stato rinominato *JavaScript* ed è stato formalizzato con una sintassi più vicina a quella del linguaggio *Java*. É stato standardizzato per la prima volta nel 1997.

Figura 3.3: Logo di JavaScript

JavaScript è un linguaggio di *scripting*: questo significa che la sintassi puó essere integrata dentro la pagina HTML, senza bisogno di produrre alcun file compilato. Con i linguaggi di programmazione come il C e il C++ si scrive, invece, la sintassi e poi la si passa ad un compilatore, che produce un file, appunto, "compilato", in cui la sintassi è scomparsa. Tutti i programmi di Windows, ad esempio, sono dei file compilati, in cui non c'è più traccia della sintassi originaria.

JavaScript, invece, non è compilato: é possibile, quindi, visualizzare in qualsiasi momento il codice di una pagina HTML e leggere le righe di sintassi.

Dire che è un linguaggio di *scripting* sottintende, dunque, il fatto che sia un linguaggio interpretato; come abbiamo visto, non esiste nessun compilatore, ma è direttamente il browser, tramite un apposito motore di *scripting* (cioè di visualizzazione), che legge le parti di codice *JavaScript*.[7]

3.4 Inizializzazione e debug

3.4.1 Nozioni importanti

Di seguito vengono introdotte informazioni di base riguardo alle componenti del progetto che sono essenziali per il corretto funzionamento del programma, ma che non saranno trattate nel prosieguo della tesi.

NPM

NPM è il gestore di pacchetti predefinito per l'ambiente di runtime *JavaScript Node.js*. Consiste di un *client* da linea di comando, chiamato anch'esso *npm*, e di un *database online* di pacchetti pubblici e privati, chiamato *npm registry*. L'*NPM* o *Node Package Manager* è la più grande libreria *opensource* al mondo di pacchetti *Javascript*; è uno strumento essenziale che permette di scaricare facilmente tutti i moduli sviluppati dalla *community Node.js*.

Tutti i moduli citati sono pubblicati nel sito web: `http://npmjs.org`; inoltre, l'*NPM* regola le dipendenze tra i vari moduli: questo significa che se un modulo ha bisogno di un altro modulo per funzionare, l'*NPM* lo scaricherà automaticamente. Un modulo è facilmente installabile, basta posizionarsi nel terminale all'interno della cartella del proprio progetto e digitare:

```
1    npm install nomemodulo
```

Il modulo verrà installato localmente, ovvero soltanto all'interno di quel progetto specifico, e lí verrà scaricata automaticamente l'ultima versione del modulo, che sará posizionata in una sottocartella `node_modules`. *NPM* per default installa i moduli localmente per ogni progetto, motivo per cui crea sottocartelle in `node_modules`.[8]

YARN

YARN, acronimo di *"Yet-Another-Resource-Negotiator"*, è un *package manager* parallelo a *NPM* e deriva, dalla versione del 2016, di quest'ultimo, ma con diverse funzionalità in più, allo scopo di sopperire alle carenze del sistema non deterministico di gestione dei pacchetti di *NPM*; per questo è stato introdotto il file `yarn.lock` accanto al file `package.json` (definito in seguito).

Quanto troviamo scritto nel `yarn.lock` non è semplicemente una lista di dipendenze, ma una lista che, innanzitutto, non punta ai *server NPM* e che contiene anche un *hash* per controllare che il *package* scaricato sia precisamente quello che vogliamo. [9]

package.json

Tutti i pacchetti *NPM* contengono un file (di solito nella *directory* principale), chiamato `package.json`. Questo file contiene vari *metadati* rilevanti per il progetto. Esso viene utilizzato per fornire informazioni ad *NPM*, consentendo ad esso di identificare il progetto e gestire le sue dipendenze: per dipendenze si intendono altri pacchetti utili al suo funzionamento. Può anche contenere altri *metadati*, come la descrizione del progetto, la versione del progetto in una particolare distribuzione, le informazioni sulla licenza e persino dati di configurazione, che possono essere vitali sia per *NPM*, sia per gli utenti finali del pacchetto. [10]

webpack e webpack-dev-server

webpack è un pacchetto *node* e per questo installabile con *NPM*; in un progetto si trova nella cartella `node_modules`. La sua funzione è quella di essere un *bundler* di moduli; lo scopo principale è raggruppare i file *JavaScript* per l'utilizzo in un *browser*, ma *webpack* è anche in grado di trasformare, raggruppare o impacchettare praticamente qualsiasi risorsa. Di grande importanza anche il `webpack-dev-server`, che sostanzialmente permette di creare un server locale e fornisce il ricaricamento della pagina web in tempo reale; quest'ultima funzione dovrebbe essere usata solo

per lo sviluppo.
Nel nostro progetto è presente anche il file di configurazione, `webpack.config.js`, nel quale si possono definire i comportamenti dei pacchetti *webpack*, come vedremo in seguito. [11]

3.4.2 Messa in opera

Di seguito si riporta la procedura corretta per inizializzare il progetto e successivamente per configurare il *debug*. É importante seguire passo dopo passo le seguenti istruzioni, affinché il programma funzioni.

1. Installare correttamente *Visual Studio Code* per la propria architettura dal seguente *link*
   `https://code.visualstudio.com/download`
2. Fare il `git clone` del progetto al *link*
   `https://github.com/capatommy/micropolisJS`
3. Aprire la finestra delle estensioni e scaricare il *plugin* `yarn` come mostrato in figura 3.4. Questo *plugin* ci serve per poter installare le dipendenze tracciate nel file `package.json`. L'avvenuta installazione di `yarn` apparirà nella sezione *Extensions* di VS Code. Successivamente, cliccando col tasto destro su `package.json`, si avrá la possibilità di mandare in esecuzione il comando *Install yarn Packages*, il quale permette di integrare i pacchetti che compaiono in un formato chiave/valore ("nome" : "versione") nella sezione *devDependencies* di `package.json`. Di conseguenza verrá prodotto automaticamente il file `yarn.lock`, sotto la *directory* principale del progetto, che non dovrà essere modificato. Dopo l'installazione delle dipendenze andremo a modificare alcuni file.
4. `tsconfig.json`: in questo file, che istruisce il compilatore TS dei file .ts (TS sta per *TypeScript*, un linguaggio *front-end* robusto e adatto per applicazioni *JavaScript* complesse, compilato a *runtime* in *JavaScript*), andiamo a modificare il *"target"* da *"es5"* a *"es6"*;
5. `config.js`: questo file si trova sotto `src/`; esso permetterà di visualizzare più opzioni per il *debugging*. Ci si troverá davanti ad un oggetto con tre proprietà "settate" tutte a `false`: basterà cambiarle in `true`;
6. `index.html`: questa è la modifica più impegnativa; per avere la versione funzionante fare riferimento al file nella *repository* su *GitHub*:
   `https://github.com/capatommy/micropolisJS/blob/prova1/index.html`.
7. Aprire il file `package.json` e cliccare sul pulsante *Debug*; successivamente cliccare `start` (raffigurato in figura 3.5).

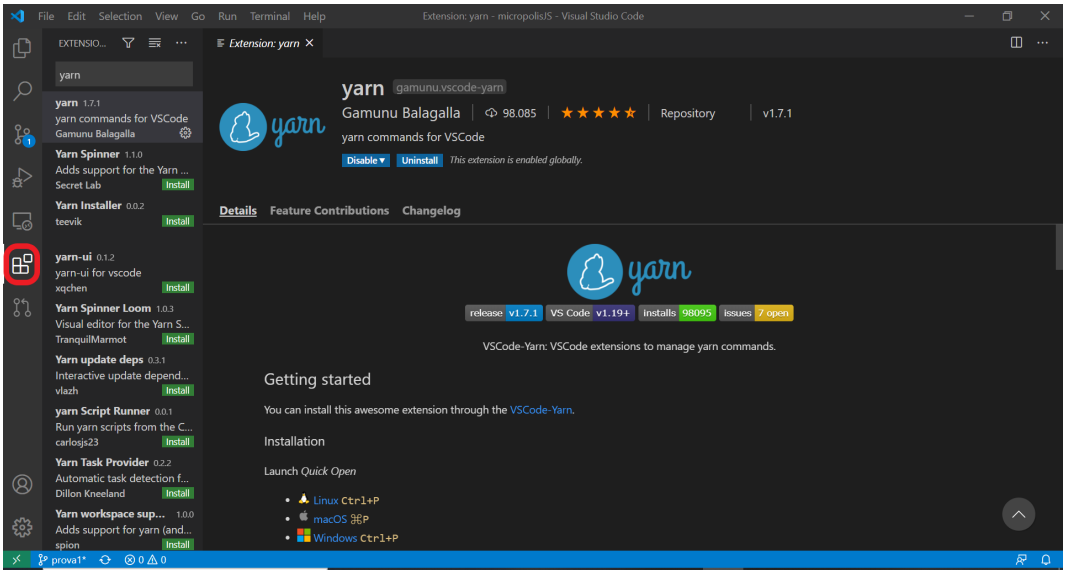

Figura 3.4: Screenshot della finestra *estensioni* dove installare il plugin `yarn`

Figura 3.5: Screenshot del pulsante *Debug* da cliccare nel file `package.json`

8. Cliccare nella finestra di *CONSOLE* raffigurata in figura 3.6 sul link `http://localhost:8080/` e attendere che si apra il *browser* predefinito.

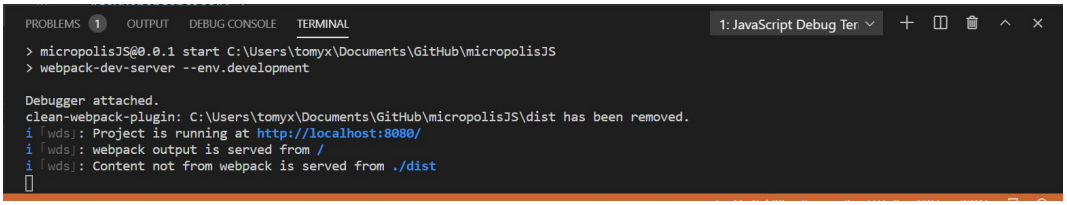

Figura 3.6: Screenshot della *CONSOLE*

Seguendo questi passi sará possibile fare il *debug* del programma sia tramite l'*IDE* che con il *browser*, semplificando di molto lo sviluppo del progetto.

Infatti, il *debug* permette di inserire in fase di esecuzione i cosiddetti *breakpoint*: sono istruzioni che generano una pausa del *runtime* del programma per dare modo allo sviluppatore di osservare meglio gli oggetti trattati in una determinata posizione del codice. Per esempio, se si posiziona il punto di *breakpoint* all'interno di un comando `if` si puó verificare se la condizione viene o meno accettata e inoltre é possibile leggere il valore delle variabili in gioco.

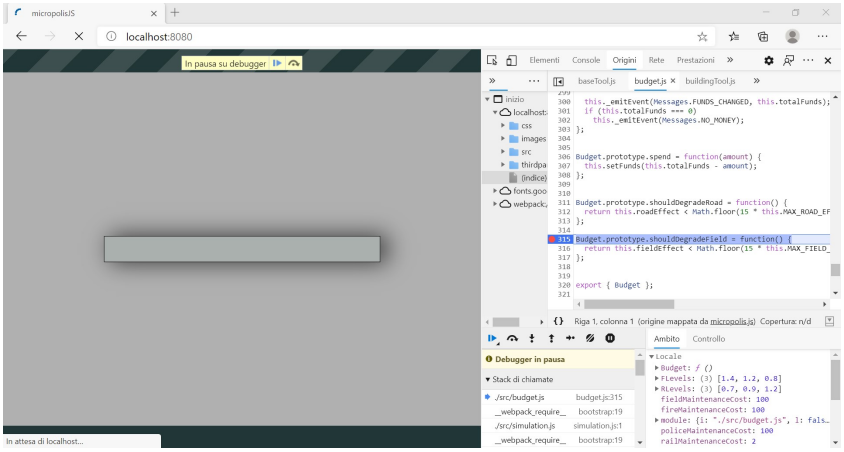

Figura 3.7: Screenshot della finestra del *browser* durante l'esecuzione di un *breakpoint*

Capitolo 4

Progettazione e Sviluppo

4.1 Descrizione degli intenti

L'obbiettivo che ha guidato lo sviluppo del nostro progetto é stato quello di introdurre quattro nuovi elementi di *gameplay*:

- `field`;

- `indfield`;

- `wwtp plant`;

- `water channel`

Gli elementi sono stati introdotti seguendo la logica che caratterizza le stutture giá presenti nel gioco, mantenendo peró indipendente dal resto il blocco di oggetti sopra elencati, che funziona anche in mancanza degli altri. L'utente potrá quindi posizionare un campo collegato con un canale all'impianto idrico senza avere altre strutture presenti.

Inoltre, sará presente un sistema per il mantenimento dei campi, i quali hanno un costo annuale che l'utente dovrá pagare insieme alle tasse. Se non verrá rispettato il pagamento per la manutenzione, il campo degraderá e non sará piú produttivo.

4.1.1 Field

Nonostante l'era tecnologica in cui viviamo, restiamo sempre legati alle tradizioni e alle tecniche introdotte secoli fa, pensiamo ad esempio all'agricoltura. La cittá da gestire in *MicropolisJS* non puó, quindi, rimanere senza la presenza di campi coltivati con diversi tipi di colture.

É stato nostro compito programmare l'oggetto `field` ed inserirlo nella dinamica di *gameplay*. Concretamente, io ed il mio collega siamo partiti dalla logica della parte residenziale, che é stata di seguito adattata all'utilizzo che ne fa il campo. Ma andiamo con ordine:

Il campo é un'entitá che a differenza delle altre strutture necessitanti di elettricitá, ha bisogno di essere collegato ad un corso d'acqua. La fonte d'acqua é rappresentata dallo stabilimento di depurazione di acque reflue, il `WWTP`, che funge da centrale idrica per la cittá. Il collegamento tra centrale `WWTP` e `field` avviene in questo modo: ogni struttura conduce a sua volta elettricitá o acqua, quindi, se il campo é adiacente alla `WWTP` o ad una struttura alimentata dalla centrale, anche il campo viene alimentato. Inoltre, esiste uno strumento di collegamento tra campo e centrale idrica: il canale (`channel`).

La valutazione in termini di punteggio del campo é effettuata tramite diversi fattori: l'adiacenza ad una strada, la presenza di irrigazione e quindi di alimentazione idrica, ed altri indicatori relativi ad inquinamento e bellezza della zona.

```
1  // Returns a score for the zone in the range -3000 - 3000
2  var evalField = function(blockMaps, x, y, traffic) {
3    if (traffic === Traffic.NO_ROAD_FOUND)
4      return -3000;
5
6    var landValue = blockMaps.landValueMap.worldGet(x, y);
7    landValue -= blockMaps.pollutionDensityMap.worldGet(x, y);
8
9    if (landValue < 0)
10     landValue = 0;
11   else
12     landValue = Math.min(landValue * 32, 6000);
13
14   return landValue - 3000;
15 };
```

Listing 4.1: Funzione di valutazione del campo nel file `field.js`

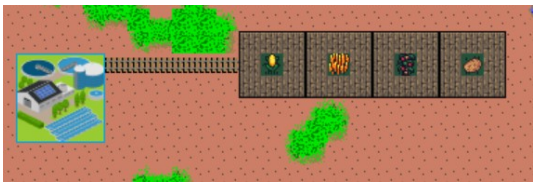

Figura 4.1: Screenshot della struttura `field` collegata alla centrale `WWTP`

Notiamo che nella funzione sopra mostrata si esegue una valutazione del campo con un indicatore che varia da -3000 a 3000; se intorno al campo non viene rilevata la costruzione di infrastrutture e in particolare di strade, la valutazione dell'indicatore è minima. In caso contrario, viene dichiarata una variabile `landValue`, il cui valore viene assegnato da una funzione esterna e modificato in base all'inquinamento. Infine, viene effettuato un controllo in base al quale se il `landValue` é negativo viene riportato a 0, altrimenti si esegue un calcolo in cui il valore massimo é 6000, dopodiché alla variabile viene sottratto il valore di 3000 in modo da rispettare l'intervallo desiderato.

4.1.2 IndField

L'inserimento di un campo non puó dipendere unicamente dalla presenza di una centrale di depurazione idrica nelle vicinanze, anche perché costruirne una richiede un esborso di denaro non indifferente. Quindi, nella logica di gioco abbiamo inserito l'oggetto `indfield`, un campo che genera risorse fin da subito, senza bisogno di collegamento con una `WWTP` e quindi ottimo per le fasi iniziali di gioco.

L'esigenza di creare un altro oggetto solo per ottenere l'indipendenza dal `WWTP` é un adattamento alla logica base del gioco: le *tiles*. Un approfondimento a riguardo viene trattato nella prossima sezione.

Figura 4.2: Screenshot dei diversi tipi di colture dell'`indfield`

Esiste, infatti, un oggetto chiamato `MapScanner`, che periodicamente viene richiamato dalla simulazione per effettuare lo *scan* della mappa e tenere il conto dei diversi parametri. Se la *tile* conduce corrente, quindi si tratta di una struttura o una centrale elettrica, viene richiamata la funzione `setTilePower`, che memorizza le coordinate di quella *tile* sapendo che esso conduce. Lo stesso discorso vale per le tessere che conducono acqua con la funzione `setTileIrrigate`. Inoltre, il gioco riconosce la tessera centrale di una struttura come *tile* di Zona; l'oggetto *scanner* si occupa di contare anche queste ultime. Un ulteriore controllo che fa `MapScanner` é relativo al tipo di coltivazione del campo.

```
1  MapScanner.prototype.mapScan = function(startX, maxX, simData) {
2    for (var y = 0; y < this._map.height; y++) {
3      for (var x = startX; x < maxX; x++) {
4        this._map.getTile(x, y, tile);
5        var tileValue = tile.getValue();
6
7        if (tileValue < Tile.FLOOD)
8          continue;
9
10       if (tile.isConductive())
11         simData.powerManager.setTilePower(x, y);
12
13       if (tile.isHydraulic())
14         simData.powerManager.setTileIrrigate(x, y);
15
16       if(tileutils.isFieldZone(tile) || tileutils.isIndFieldZone(tile)
           )
```

```
17          simData.powerManager.setCostCrop(x,y);
18
19        if (tile.isZone()) {
20          simData.repairManager.checkTile(x, y, simData.cityTime);
21          var powered = tile.isPowered();
22          if (powered)
23            simData.census.poweredZoneCount += 1;
24          else
25            simData.census.unpoweredZoneCount += 1;
26        }
```

Listing 4.2: Listato della funzione interessata in `mapScanner.js`

Il grande valore di `MapScanner` sta soprattutto nella possibilitá di eseguire lo *scan* di un preciso tipo di struttura in base ai parametri passati ad una determinata `action`, che sará poi richiamata dentro un ciclo in `MapScanner`. Le `actions` vengono aggiunte al vettore tramite la funzione `addAction`, mostrata di seguito.

```
1  MapScanner.prototype.addAction = function(criterion, action) {
2    this._actions.push({criterion: criterion, action: action});
3  };
```

Listing 4.3: Funzione `addAction` definita all'interno di `mapScanner.js`

```
1  for (var i = 0, l = this._actions.length; i < l; i++) {
2          var current = this._actions[i];
3          var callable = isCallable(current.criterion);
4
5          if (callable && current.criterion.call(null, tile)) {
6            current.action.call(null, this._map, x, y, simData);
7            break;
8          } else if (!callable && current.criterion === tileValue) {
9            current.action.call(null, this._map, x, y, simData);
10           break;
11         }
```

Listing 4.4: Listato del ciclo dove vengono eseguite le `actions`

La presenza delle `actions` per forza di cose ha portato alla generazione di due oggetti diversi: il `field` e l'`indfield`. Ognuno ha le proprie *tiles* ed ognuno ha le proprie funzioni di *scan*, in modo da differenziarli e non creare sovrapposizioni.

4.1.3 WWTP Plant

Centrale nello svolgimento del progetto é stato il ruolo del *Waste-Water-Treatment-Plant*, per semplicitá abbrieviato `WWTP`: é un tipo di stabilimento il cui fine é trattare le acque reflue generalmente industriali, eliminando le impuritá dannose per l'ambiente. Questo permette, quindi, di riutilizzare le acque nere senza rischi, anzi rendendole utili in ambiti come, per esempio, quello agricolo.

Figura 4.3: Screenshot delle *tiles* del `field` e dell'`indfield`

Ai fini del *gameplay* la stazione `WWTP` funge da centrale idrica, alimentando i campi coltivati. A livello di codice, abbiamo fatto un parallelo con le centrali elettriche giá presenti nel gioco e abbiamo integrato la logica idrica nel file chiamato `powerManager.js`.

All'inizio del file si dichiarano le costanti appartenenti al tipo di energia trattata, le quali sono utilizzate per calcolare l'efficienza della copertura energetica ed idrica se moltiplicate al numero di centrali presenti nella mappa. Per semplicitá, abbiamo usato lo stesso valore della centrale a carbone.

```
1 var COAL_POWER_STRENGTH = 700;
2 var WWTP_POWER_STRENGTH = 700;
3 var NUCLEAR_POWER_STRENGTH = 2000;
```

Listing 4.5: Costanti di efficienza

L'oggetto `PowerManager` eredita la classe `EventEmitter` e definisce al suo interno delle variabili importanti, come gli *stack* relativi all'energia e all'irrigazione. Questi elementi saranno poi riempiti dalle rispettive funzioni `doPowerScan, doIrrigateScan`, che realizzano operazioni di `push` e di `pop` in base alle centrali elettriche e alle stazioni `WWTP` trovate.

```
1 var PowerManager = EventEmitter(function(map) {
2   this._map = map;
3   this._powerStack = [];
4   this._setCropStack = [];
5   this.powerGridMap = new BlockMap(this._map.width, this._map.height,
        1);
6   this.irrigateGridMap = new BlockMap(this._map.width, this._map.
        height, 1);
7   this.costFieldMap = new BlockMap(this._map.width, this._map.height,
        1);
```

```
8 });
```

Listing 4.6: Inizializzazione dell'oggetto `PowerManager`

In ultimo vengono inizializzati tre oggetti `BlockMap`; si tratta di mappe relative solo ad una specifica caratteristica. In questo caso una é relativa alla copertura elettrica, una alla copertura idrica ed una serve per tenere traccia delle colture dei campi, le quali si differenziano per il costo di costruzione.

Alla fine del file vengono aggiunte le `actions` relative al `MapScanner`, che fungono da *trigger*. Infatti, quando l'oggetto `MapScanner` esegue il controllo su tutte le *tiles* della mappa, se incontra delle tessere di tipo `WWTP, POWERPLANT` o `NUCLEAR` richiama le rispettive funzioni. Queste funzioni eseguono operazioni di gestione delle animazioni e aumentano lo *stack* della copertura relativa.

```
1 PowerManager.prototype.registerHandlers = function(mapScanner,
      repairManager) {
2   mapScanner.addAction(Tile.POWERPLANT, this.coalPowerFound.bind(this)
        );
3   mapScanner.addAction(Tile.NUCLEAR, this.nuclearPowerFound.bind(this)
        );
4   mapScanner.addAction(Tile.WWTP, this.wwtpPowerFound.bind(this));
5   repairManager.addAction(Tile.POWERPLANT, 7, 4);
6   repairManager.addAction(Tile.NUCLEAR, 7, 4);
7   repairManager.addAction(Tile.WWTP, 7, 4);
8 };
```

Listing 4.7: Listato del `registerHandlers`

```
1  PowerManager.prototype.coalPowerFound = function(map, x, y, simData) {
2    simData.census.coalPowerPop += 1;
3
4    this._powerStack.push(new map.Position(x, y));
5
6    // Ensure animation runs
7    var dX = [-1, 2, 1, 2];
8    var dY = [-1, -1, 0, 0];
9
10   for (var i = 0; i < 4; i++)
11     map.addTileFlags(x + dX[i], y + dY[i], Tile.ANIMBIT);
12 };
```

Listing 4.8: Listato della funzione `coalPowerFound`

4.1.4 Channel

Lo strumento `channel` viene definito come oggetto figlio della classe `ConnectingTool`, che appunto definisce la sua natura; si tratta di uno strumento che serve a collegare due strutture, permettendo loro di interagire. Infatti, viene utilizzato per portare

l'acqua dalla centrale `WWTP` ai campi, rendendoli produttivi.

Abbiamo introdotto questo oggetto sulla falsa riga del codice di un altro strumento, il `wireTool`. La logica nel concreto é la stessa: il `channel` alimenta i campi collegandoli alla `WWTP`, come il `wire` collega la centrale elettrica alle altre strutture.

Il codice del file `channelTool.js` si articola in questo modo: la funzione principale, `layChannel`, ha il compito di pulire la *tile* selezionata tramite lo strumento *Bulldozer* per rimuovere eventuali piante.

Successivamente, a seconda del tipo di *tile* su cui deve costruire il canale (terra, acqua), utilizza una tessera diversa. Questo vale anche se deve modulare il proprio percorso; esistono, infatti, diverse tessere per gestire cambi di direzione nel percorso e sovrapposizioni ad altri tipi di connettori (`wire`, `road`).

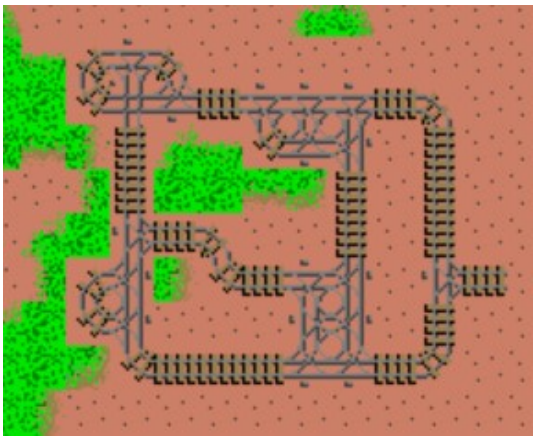

Figura 4.4: Screenshot delle tessere dello strumento `channel`

Infine, si valorizza il *BITMASK* della tessera in modo che il `channel` possa condurre acqua nel suo percorso; si valorizza, quindi, l'`HYDRABIT`.

Graficamente abbiamo utilizzato le *tiles* giá presenti dello strumento `rail`; questo ci ha permesso di provare in tempi brevi il componente e affinarne l'efficienza. Nella sezione successiva verrá spiegato in modo piú approfondito il funzionamento delle *tiles* e del loro relativo `BITMASK`.

```
1 import { ConnectingTool } from './connectingTool';
2 import { Tile } from './tile';
3 import { TileUtils } from './tileUtils';
4
5 var ChannelTool = ConnectingTool(function(map) {
6   this.init(5, map, true, true);
7 });
```

```
8
9
10 ChannelTool.prototype.layChannel = function(x, y) {
11   this.doAutoBulldoze(x, y);
12   var cost = this.toolCost;
13
14   var tile = this._worldEffects.getTileValue(x, y);
15   tile = TileUtils.normalizeRoad(tile);
16
17   switch (tile) {
18     case Tile.DIRT:
19       this._worldEffects.setTile(x, y, Tile.LHTUBE, Tile.HYDRABIT |
             Tile.BURNBIT | Tile.BULLBIT);
20       break;
21
22     case Tile.RIVER:
23     case Tile.REDGE:
24     case Tile.CHANNEL:
25       cost = 25;
26
27       if (x < this._map.width - 1) {
28         tile = this._worldEffects.getTile(x + 1, y);
29         if (tile.isHydraulic()) {
30           tile = tile.getValue();
31           tile = TileUtils.normalizeRoad(tile);
32           if (tile != Tile.HTUBEROAD && tile != Tile.TUBEHPOWERV &&
                 tile != Tile.HTUBE) {
33             this._worldEffects.setTile(x, y, Tile.VTUBE, Tile.HYDRABIT
                   | Tile.BULLBIT);
34             break;
35           }
36         }
37       }
38
39       if (x > 0) {
40         tile = this._worldEffects.getTile(x - 1, y);
41         if (tile.isHydraulic()) {
42           tile = tile.getValue();
43           tile = TileUtils.normalizeRoad(tile);
44           if (tile != Tile.HTUBEROAD && tile != Tile.TUBEHPOWERV &&
                 tile != Tile.HTUBE) {
45             this._worldEffects.setTile(x, y, Tile.VTUBE, Tile.HYDRABIT
                   | Tile.BULLBIT);
46             break;
47           }
48         }
49       }
50
51       if (y < this._map.height - 1) {
52         tile = this._worldEffects.getTile(x, y + 1);
53         if (tile.isHydraulic()) {
```

```
54          tile = tile.getValue();
55          tile = TileUtils.normalizeRoad(tile);
56          if (tile != Tile.VTUBEROAD && tile != Tile.TUBEVPOWERH &&
                tile != Tile.VTUBE) {
57            this._worldEffects.setTile(x, y, Tile.HTUBE, Tile.HYDRABIT
                  | Tile.BULLBIT);
58            break;
59          }
60        }
61      }
62
63      if (y > 0) {
64        tile = this._worldEffects.getTile(x, y - 1);
65        if (tile.isHydraulic()) {
66          tile = tile.getValue();
67          tile = TileUtils.normalizeRoad(tile);
68          if (tile != Tile.VTUBEROAD && tile != Tile.TUBEVPOWERH &&
                tile != Tile.VTUBE) {
69            this._worldEffects.setTile(x, y, Tile.HTUBE, Tile.HYDRABIT
                  | Tile.BULLBIT);
70            break;
71          }
72        }
73      }
74
75      return this.TOOLRESULT_FAILED;
76
77    case Tile.ROADS:
78      this._worldEffects.setTile(x, y, Tile.HTUBEROAD, Tile.HYDRABIT |
            Tile.BURNBIT | Tile.BULLBIT);
79      break;
80
81    case Tile.ROADS2:
82      this._worldEffects.setTile(x, y, Tile.VTUBEROAD, Tile.HYDRABIT |
            Tile.BURNBIT | Tile.BULLBIT);
83      break;
84
85    case Tile.LHPOWER:
86      this._worldEffects.setTile(x, y, Tile.TUBEVPOWERH, Tile.HYDRABIT
            | Tile.BURNBIT | Tile.BULLBIT);
87      break;
88
89    case Tile.LVPOWER:
90      this._worldEffects.setTile(x, y, Tile.TUBEHPOWERV, Tile.HYDRABIT
            | Tile.BURNBIT | Tile.BULLBIT);
91      break;
92
93    default:
94      return this.TOOLRESULT_FAILED;
95  }
96
```

```
97    this.addCost(cost);
98    this.checkZoneConnections(x, y);
99    return this.TOOLRESULT_OK;
100 };
101
102
103 ChannelTool.prototype.doTool = function(x, y, blockMaps) {
104   this.result = this.layChannel(x, y);
105 };
106
107
108 export { ChannelTool };
```

Listing 4.9: Listato del codice di `channelTool.js`

4.2 Funzionamento delle Tiles

La logica dominante di *MicropolisJS* é quella delle *tiles*, ovvero tessere. Il mondo generato nel gioco si basa su quadrati di 16 *pixels*, ognuno codificato con un numero che va da 1 a 1028. La simulazione periodicamente "scannerizza" la mappa di gioco, riconosce il valore di ogni *tile* nella mappa e agisce di conseguenza, aumentando il carico energetico se trova una *tile* residenziale o aumentando la capacitá idrica se riconosce una tessera relativa alla centrale `WWTP`.

Ogni *tile* possiede una `BITMASK`, che ne caratterizza la funzione: se conduce corrente il suo 14esimo bit sará valorizzato a 1, se puó prendere fuoco sará il bit numero 13 ad essere 1 e cosí via.

```
1  // Bit-masks for statusBits
2  Tile.HYDRABIT = 0x20000; // bit 17, tile can run water. AGG
3  Tile.IRRIGBIT = 0x10000; // bit 16, tile is irrigated. AGGIUNTO
4  Tile.POWERBIT  = 0x8000; // bit 15, tile has power.
5  Tile.CONDBIT = 0x4000; // bit 14. tile can conduct electricity.
6  Tile.BURNBIT = 0x2000; // bit 13, tile can be lit.
7  Tile.BULLBIT = 0x1000; // bit 12, tile is bulldozable.
8  Tile.ANIMBIT = 0x0800; // bit 11, tile is animated.
9  Tile.ZONEBIT = 0x0400; // bit 10, tile is the center tile of the zone.
10 Tile.BLBNBIT   = Tile.BULLBIT | Tile.BURNBIT;
11 Tile.BLBNCNBIT = Tile.BULLBIT | Tile.BURNBIT | Tile.CONDBIT;
12 Tile.BNCNBIT   = Tile.BURNBIT | Tile.CONDBIT;
13 Tile.ASCBIT   = Tile.ANIMBIT | Tile.CONDBIT | Tile.BURNBIT;
14 Tile.BNHYBIT   = Tile.BURNBIT | Tile.HYDRABIT; //
15 Tile.BLBNHYBIT = Tile.BULLBIT | Tile.BURNBIT | Tile.HYDRABIT; //
16 Tile.ALLBITS = Tile.HYDRABIT | Tile.IRRIGBIT | Tile.POWERBIT | Tile.
       CONDBIT | Tile.BURNBIT | Tile.BULLBIT | Tile.ANIMBIT | Tile.
       ZONEBIT ;
17 Tile.BIT_START = 0x400;
18 Tile.BIT_END = 0x20000;
```

```
19 Tile.BIT_MASK = Tile.BIT_START - 1;
```

Listing 4.10: Definizione `BITMASK` nel file `tile.js`

Le tessere totali sono quindi limitate a 1028 (figura 4.5), fattore che ha reso necessaria la sovrascrittura di alcune *tiles*, come fatto per l'oggetto `channel` che rimpiazza totalmente l'oggetto `rail` ovvero i binari del treno. Per cambiare la grafica delle tessere si é modificato direttamente il file `tiles.png` con un manipolatore di immagini (GIMP[12] per esempio), inserendo le nuove *tiles* nella giusta posizione.

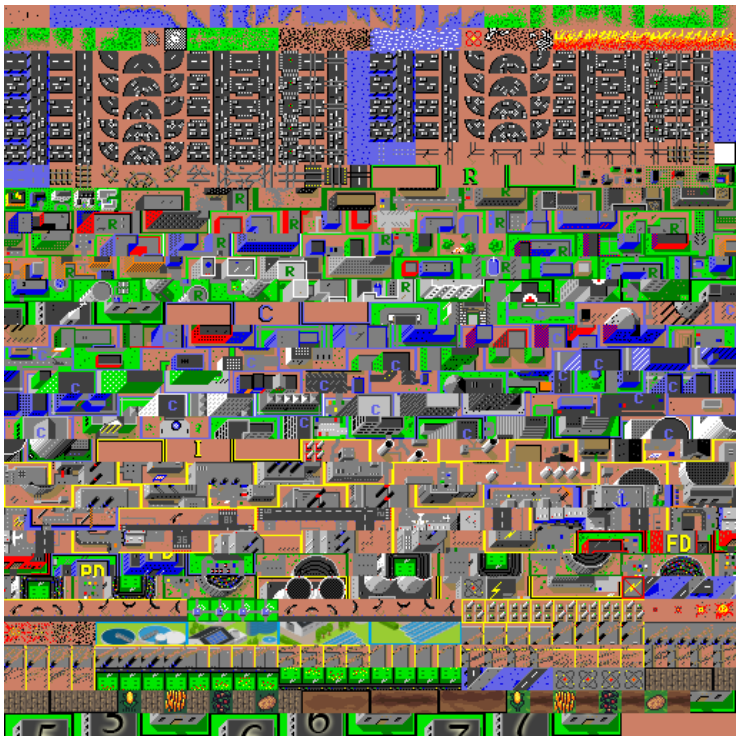

Figura 4.5: Insieme delle *tiles* del gioco

4.3 Scelta del campo e del tipo di coltura

Una parte impegnativa nello sviluppo del nostro progetto é stato rendere possibile la scelta da parte dell'utente del tipo di campo e della coltura relativa ad esso, tramite una finestra di *dialog*.

4.3.1 fieldWindow.js

Il file relativo alla finestra di scelta eredita i metodi di `ModalWindow` e permette due scelte: una relativa alla dipendenza dalla `WWTP` e l'altra relativa al tipo di coltura.

Vengono definite due azioni, `WWTP` e `CROP`, che restituiranno due valori una volta premuto il pulsante *submit* nella finestra. `shouldWWTP` é una funzione che ritorna un valore `bool` che, se positivo indica dipendenza da `WWTP`, invece `cropSelect` é un intero che va da 0 a 3 e indica quattro tipi di coltura.

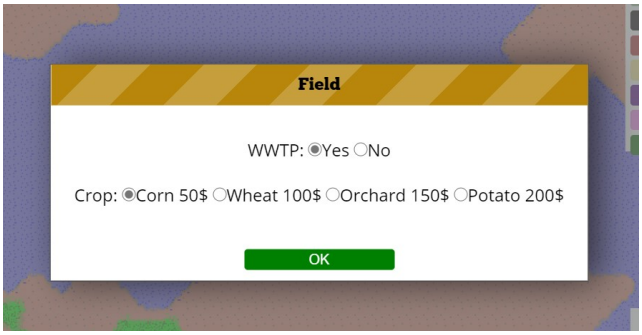

Figura 4.6: Screenshot della finestra di scelta del campo

```
1  import { Messages } from './messages';
2  import { ModalWindow } from './modalWindow';
3  import { MiscUtils } from './miscUtils';
4  import { Simulation } from './simulation';
5  import { Game } from './game';
6
7  var FieldWindow = ModalWindow(function() {
8    $(fieldFormID).on('submit', submit.bind(this));
9  },);
10
11
12 var cropCornID = '#cropCorn';
13 var cropPotatoID = '#cropPotato';
14 var cropWheatID = '#cropWheat';
15 var cropOrchardID = '#cropOrchard';
16 var fieldFormID = '#fieldForm';
17 var fieldOKID = '#fieldOK';
18 var WWTPYesID = '#WWTPYes';
19 var WWTPNoID = '#WWTPNo';
20
21
22
23 FieldWindow.prototype.close = function(actions) {
24   actions = actions || [];
25   this._emitEvent(Messages.FIELD_WINDOW_CLOSED, actions);
26   this._toggleDisplay();
27 };
28
29 var submit = function(e) {
30   e.preventDefault();
31
32   var actions = [];
```

```
33
34    var shouldWWTP = $('.WWTPField:checked').val();
35    if (shouldWWTP === 'true')
36      shouldWWTP = true;
37    else
38      shouldWWTP = false;
39    actions.push({action: FieldWindow.WWTP, data: shouldWWTP});
40
41    var cropSelect = $('.cropSetting:checked').val() - 0;
42    actions.push({action: FieldWindow.CROP, data: cropSelect});
43    this.close(actions);
44  };
45
46
47  FieldWindow.prototype.open = function(fieldData) {
48      $(WWTPYesID).prop('checked', true);
49
50      $(cropCornID).prop('checked', true);
51
52    this._toggleDisplay();
53  };
54
55
56  var defineAction = (function() {
57    var uid = 0;
58
59    return function(name) {
60      Object.defineProperty(FieldWindow, name, MiscUtils.
           makeConstantDescriptor(uid));
61      uid += 1;
62    };
63  })();
64
65
66  defineAction('WWTP');
67  defineAction('CROP');
68
69  export { FieldWindow };
```

Listing 4.11: Listato del file FieldWindow.js

4.3.2 game.js

In questo file sono gestiti gli *Handler* di apertura e chiusura relativi alla finestra; da qui verranno gestiti i dati ricevuti e saranno richiamate le funzioni caratterizzanti la costruzione del tipo di `field` desiderato.

Nel file `baseTool.js` sono state definite due funzioni, spiegate nella sezione successiva; si tratta di `setWWTP` e `setCropCost`, le quali vengono richiamate in questo file, passando loro come parametri le variabili ottenute dalla scelta dell'utente tramite la

finestra.

La funzione che gestisce la chiusura della finestra con un ciclo `for` scandaglia le varie azioni definite prima e con uno `switch case` richiama le funzioni relative al file `baseTool.js`.

```
Game.prototype.handleFieldWindowClosure = function(actions) {
  this.dialogOpen = false;

  for (var i = 0, l = actions.length; i < l; i++) {
    var a = actions[i];

    switch (a.action) {
      case FieldWindow.WWTP:
       BaseTool.setWWTP(a.data);
        break;

      case FieldWindow.CROP:
        this.setCrop(a.data);
        break;

      default:
        console.warn('Unexpected action', a);
    }
  }
};

Game.prototype.setCrop = function(c){
  if (c !== Simulation.CROP_CORN &&
    c !== Simulation.CROP_WHEAT &&
    c !== Simulation.CROP_ORCHARD &&
    c !== Simulation.CROP_POTATO)
  throw new Error('Invalid crop!');

  switch (c) {
    case Simulation.CROP_CORN:

      BaseTool.setCropCost(BaseTool.CORN_COST);
      break;

    case Simulation.CROP_WHEAT:

      BaseTool.setCropCost(BaseTool.WHEAT_COST);
      break;

    case Simulation.CROP_ORCHARD:

      BaseTool.setCropCost(BaseTool.ORCHARD_COST);
      break;

    case Simulation.CROP_POTATO:
```

```
46
47        BaseTool.setCropCost(BaseTool.POTATO_COST);
48        break;
49
50      default:
51          console.warn('Unexpected action', a);
52    }
53 };
```

Listing 4.12: Listato delle funzioni interessate nel file `game.js`

4.3.3 baseTool.js

Le funzioni `setWWTP` e `setCropCost` sono definite in questo file, ció perché lo strumento che poi cambierá la *tile* con quella del nostro campo, il `BuildingTool`, é una classe figlia di `BaseTool` e ne eredita variabili e metodi.

Quindi, definendo due variabili `wwtp` e `cropcost` nel costruttore di `Basetool` sará possibile, grazie a funzioni di `get` e `set`, modificare e leggere i valori desiderati.

```
1  var BaseToolConstructor = {
2    addCost: addCost,
3    autoBulldoze: true,
4    wwtp: true,
5    cropcost: cropcost,
6    bulldozerCost: 1,
7    clear: clear,
8    doAutoBulldoze: doAutoBulldoze,
9    init: init,
10   modifyIfEnoughFunding: modifyIfEnoughFunding,
11   TOOLRESULT_OK: TOOLRESULT_OK,
12   TOOLRESULT_FAILED: TOOLRESULT_FAILED,
13   TOOLRESULT_NO_MONEY: TOOLRESULT_NO_MONEY,
14   TOOLRESULT_NEEDS_BULLDOZE: TOOLRESULT_NEEDS_BULLDOZE
15 };
```

Listing 4.13: Costruttore della classe `BaseTool`

```
1  var BaseTool = {
2    makeTool: makeTool,
3    setCropCost: function(value){
4      BaseToolConstructor.cropcost = value;
5  },
6    getCropCost: function() {
7    return BaseToolConstructor.cropcost;
8  },
9    setWWTP: function(value){
10       BaseToolConstructor.wwtp = value;
11   },
12   getWWTP: function() {
13     return BaseToolConstructor.wwtp;
```

```
14    },
15    setAutoBulldoze: function(value) {
16      BaseToolConstructor.autoBulldoze = value;
17    },
18    getAutoBulldoze: function() {
19      return BaseToolConstructor.autoBulldoze;
20    },
21    save: save,
22    load: load,
23    CORN_COST : CORN_COST,
24    WHEAT_COST : WHEAT_COST,
25    ORCHARD_COST : ORCHARD_COST,
26    POTATO_COST : POTATO_COST
27  };
```

Listing 4.14: Definizione delle funzioni di set e di get per `cropcost` e `wwtp`

4.3.4 buildingTool.js

La gestione di questo strumento é stata una delle parti piú complesse nello sviluppo del progetto. Infatti `BuildingTool` é responsabile del cambio di *tile* nella mappa e quindi della costruzione delle strutture del gioco.

Il file `gametools.js` inizializza tutti i pulsati visibili nell'interfaccia relativi alla costruzione di strutture; essi sono oggetti `BuildingTool`, ognuno provvisto di due parametri caratteristici: il prezzo della struttura e la sua *tile* centrale.

```
1  import { BuildingTool } from './buildingTool';
2  import { BulldozerTool } from './bulldozerTool';
3  import { EventEmitter } from './eventEmitter';
4  import { Messages } from './messages';
5  import { MiscUtils } from './miscUtils';
6  import { ParkTool } from './parkTool';
7  import { RailTool } from './railTool';
8  import { RoadTool } from './roadTool';
9  import { QueryTool } from './queryTool';
10 import { Tile } from './tile';
11 import { WireTool } from './wireTool';
12 import { ChannelTool } from './channelTool';
13
14 function GameTools(map) {
15   var tools = EventEmitter({
16     airport: new BuildingTool(10000, Tile.AIRPORT, map, 6, false),
17     bulldozer: new BulldozerTool(map),
18     coal: new BuildingTool(3000, Tile.POWERPLANT, map, 4, false),
19     commercial: new BuildingTool(100, Tile.COMCLR, map, 3, false),
20     fire: new BuildingTool(500, Tile.FIRESTATION, map, 3, false),
21     industrial: new BuildingTool(100, Tile.INDCLR, map, 3, false),
22     wwtp: new BuildingTool(3000, Tile.WWTP, map, 4, false),
```

```
23      nuclear: new BuildingTool(5000, Tile.NUCLEAR, map, 4, true),
24      park: new ParkTool(map),
25      police: new BuildingTool(500, Tile.POLICESTATION, map, 3, false),
26      port: new BuildingTool(3000, Tile.PORT, map, 4, false),
27      rail: new RailTool(map),
28      residential: new BuildingTool(100, Tile.FREEZ, map, 3, false),
29      field: new BuildingTool(0, Tile.FREEF, map, 3, false),
30      road: new RoadTool(map),
31      query: new QueryTool(map),
32      stadium: new BuildingTool(5000, Tile.STADIUM, map, 4, false),
33      wire: new WireTool(map),
34      channel: new ChannelTool(map),
35    });
36
37    tools.query.addEventListener(Messages.QUERY_WINDOW_NEEDED, MiscUtils
          .reflectEvent.bind(tools, Messages.QUERY_WINDOW_NEEDED));
38
39    return tools;
40  }
41
42
43  export { GameTools };
```

Listing 4.15: Listato di `gameTools`

Ma come utilizzare lo strumento di costruzione se ogni volta i campi scelti saranno di tipo diverso?

La soluzione applicata sta nell'inizializzare l'oggetto con valori di *default* (costo di costruzione 0 e *tile* centrale del campo uguale a `FREEF`) e ogni volta che si apre la finestra di scelta del campo sta nell'aggiornare i valori delle variabili `wwtp` e `cropcost` tramite le funzioni definite in `BaseTool`. É cosí possibile aggiornare il `BuildingTool`, che in questo modo imposta il giusto costo del campo e il giusto tipo di campo prima di costruirlo.

```
1   BuildingTool.prototype.putBuilding = function(leftX, topY) {
2     var posX, posY, tileValue, tileFlags;
3    // var baseTile = this.centreTile - this.size - 1;
4     var baseTile;
5     var b = BaseTool.getWWTP();
6     var c = BaseTool.getCropCost();
7
8     if(this.centreTile == Tile.FREEF || this.centreTile == Tile.FREEINDF
          ) {
9         if(b) {this.centreTile = Tile.FREEF;}
10        else this.centreTile = Tile.FREEINDF;
11        this.addCost(c);
12    }
13
14    baseTile = this.centreTile - this.size - 1;
15
```

```
16  for (var dy = 0; dy < this.size; dy++) {
17    posY = topY + dy;
18
19    for (var dx = 0; dx < this.size; dx++) {
20      posX = leftX + dx;
21      tileValue = baseTile;
22
23      if (TileUtils.isIndField(tileValue))
24      {
25        if (dx === 1 && dy === 1 && (tileValue === Tile.FREEINDF))
26        {
27          switch (c){
28            case BaseTool.CORN_COST:
29              tileValue = Tile.INDFCORN;
30              break;
31
32            case BaseTool.WHEAT_COST:
33              tileValue = Tile.INDFWHEAT;
34              break;
35
36            case BaseTool.ORCHARD_COST:
37              tileValue = Tile.INDFORCHARD;
38              break;
39
40            case BaseTool.POTATO_COST:
41              tileValue = Tile.INDFPOTATO;
42              break;
43
44            default: break;
45          }
46        }
47      tileFlags = Tile.BURNBIT;
48      }
49        else if (TileUtils.isField(tileValue) ||(tileValue>=Tile.
              WWTPBASE && tileValue<=Tile.LASTWWTP))
50        {
51        if (dx === 1 && dy === 1 && (tileValue === Tile.FREEF))
52        {
53          switch (c){
54            case BaseTool.CORN_COST:
55              tileValue = Tile.FCORN;
56              break;
57
58            case BaseTool.WHEAT_COST:
59              tileValue = Tile.FWHEAT;
60              break;
61
62            case BaseTool.ORCHARD_COST:
63              tileValue = Tile.FORCHARD;
64              break;
65
```

```
66              case BaseTool.POTATO_COST:
67                tileValue = Tile.FPOTATO;
68                break;
69
70              default: break;
71               }
72             }
73          tileFlags = Tile.BNHYBIT;
74        }
75        else
76          tileFlags = Tile.BNCNBIT;
77
78        if (dx === 1) {
79          if (dy === 1){
80            tileFlags |= Tile.ZONEBIT;
81          }
82          else if (dy === 2 && this.animated)
83            tileFlags |= Tile.ANIMBIT;
84        }
85
86        this._worldEffects.setTile(posX, posY, tileValue, tileFlags);
87
88        baseTile++;
89      }
90    }
91  };
92
93
94
95  BuildingTool.prototype.buildBuilding = function(x, y) {
96    // Correct to top left
97    x--;
98    y--;
99
100   var prepareResult = this.prepareBuildingSite(x, y);
101   if (prepareResult !== this.TOOLRESULT_OK)
102     return prepareResult;
103
104   this.addCost(this.toolCost);
105
106   this.putBuilding(x, y);
107
108   this.checkBorder(x, y);
109
110   return this.TOOLRESULT_OK;
111 };
112
113
114 BuildingTool.prototype.doTool = function(x, y, blockMaps) {
115   this.result = this.buildBuilding(x, y);
```

```
116 };
```

Listing 4.16: Listato delle funzioni interessate in `buildingTool.js`

Si puó notare che all'interno della funzione `putBuilding` si eseguono due `switch case` relative ai due tipi di campo: uno per `field` e l'altro per `indfield`. Per gestire le colture all'interno dei due campi abbiamo utilizzato, per ciascun tipo, 4 *tile* di coltivazione e 4 *tile* al fine di segnalare che il campo é in degrado, per un totale di 16 tessere complessive.

Figura 4.7: Insieme delle *tiles* delle coltivazioni

4.3.5 powerManager.js

Nel file `powerManager.js` é presente una funzione importante chiamata `setCropCost`, che prende come parametri le coordinate della *tile* trattata e tramite uno `switch case` aggiorna la mappa relativa ai tipi di coltivazione (`costFieldMap`). Questi valori verrano utilizzati dal file `field.js` per poter impostare la *tile* centrale come produttiva.

```
1  PowerManager.prototype.setCostCrop = function(x, y) {
2    var tile = this._map.getTile(x, y);
3    var tileValue = tile.getValue();
4
5    switch (tileValue) {
6      case Tile.CORN:
7      case Tile.FCORN:
8      case Tile.INDCORN:
9      case Tile.INDFCORN:
10       this.costFieldMap.set(x, y, BaseTool.CORN_COST); break;
11     case Tile.WHEAT:
12     case Tile.FWHEAT:
13     case Tile.INDWHEAT:
14     case Tile.INDFWHEAT:
15       this.costFieldMap.set(x, y, BaseTool.WHEAT_COST); break;
16     case Tile.ORCHARD:
17     case Tile.FORCHARD:
18     case Tile.INDORCHARD:
19     case Tile.INDFORCHARD:
20       this.costFieldMap.set(x, y, BaseTool.ORCHARD_COST); break;
21     case Tile.POTATO:
22     case Tile.FPOTATO:
23     case Tile.INDPOTATO:
24     case Tile.INDFPOTATO:
25       this.costFieldMap.set(x, y, BaseTool.POTATO_COST); break;
```

```
26      default: break;
27    }
28  };
```

Listing 4.17: Listato relativo a SetCostCrop

4.3.6 field.js

Il file `field.js` completa la caratterizzazione del campo nella seguente modalitá: la funzione `fieldFound`, il cui ruolo é centrale, verifica se il campo é di fatto irrigato, quindi produttivo. Se questa condizione si verifica, viene dichiarata una variabile `cost` che ottiene il tipo di coltivazione, prendendolo direttamente dalla prima citata `costFieldMap` con una chiamata `get`. A questo punto viene impostata la *tile* centrale con il logo della coltura associata, simbolo di produttivitá.

Nel caso dell'`indfield` la procedura appena descritta per il campo collegato alla rete idrica é essenzialmente la stessa.Le due procedure si differenziano principalmente per quest'ultimo passaggio, infatti il campo indipendente diventa produttivo nel momento della sua costruzione a causa della sua natura.

```
1  var fieldFound = function(map, x, y, simData) {
2    // If we choose to grow this zone, we will fill it with an index in
         the range 0-3 reflecting the land value and
3    // pollution scores (higher is better). This is then used to select
         the variant to build
4    var lpValue;
5    var tile = map.getTileValue(x, y);
6    var zoneIrrigate = map.getTile(x, y).isIrrigated();
7
8    var cost = 0;
9    if(zoneIrrigate) {
10     // Notify the census
11     simData.census.fieldZonePop += 1;
12     cost = simData.powerManager.costFieldMap.get(x, y);
13     switch(cost){
14       case BaseTool.CORN_COST:
15         tile = Tile.CORN;
16         break;
17
18       case BaseTool.WHEAT_COST:
19         tile = Tile.WHEAT;
20         break;
21
22       case BaseTool.ORCHARD_COST:
23         tile = Tile.ORCHARD;
24         break;
25
26       case BaseTool.POTATO_COST:
```

```
27          tile  = Tile.POTATO;
28          break;
29
30        default: break;
31      }
32    }
33    else{
34      tile = Tile.FREEF;
35      }
36
37      map.setTile(x, y, tile, Tile.BLBNHYBIT | Tile.ZONEBIT);
```

Listing 4.18: Parte interessata della funzione `fieldFound`

4.4 Mantenimento, crescita e degrado

Parte fondamentale del *gameplay* é stata l'introduzione del campo dentro la simulazione di gioco: io ed il mio collega abbiamo infatti predisposto un ciclo di mantenimento del campo, per riuscire a rendere meno statico l'oggetto `field`.

Questa parte viene trattata in maniera approfondita nella tesi del collega Massimo Merla, trattandosi della sezione in cui egli ha operato maggiormente. Di seguito, invece, un riassunto degli aspetti piú importanti della logica.

L'impostazione data alla natura delle strutture `field` ed `indfield` é la seguente: a differenza degli altri tipi di costruzioni, la produttivitá dei campi non dipende dalla popolazione che risiede in essi, ma dal pagamento annuale di una tassa che ne garantisce il funzionamento.

Al fine di implementare il tutto, abbiamo sfruttato delle funzioni giá esistenti per il degrado automatico della strada; questo avviene quando non si paga la dovuta tassa alla fine di ogni anno. Siamo partiti, quindi, dalla modifica del file `budget.js`.

```
1  Budget.prototype.collectTax = function(gameLevel, census) {
2    this.cashFlow = 0;
3
4    // How much would it cost to fully fund every service?
5    this.policeMaintenanceBudget = census.policeStationPop *
         policeMaintenanceCost;
6    this.fireMaintenanceBudget = census.fireStationPop *
         fireMaintenanceCost;
7    this.fieldMaintenanceBudget = ( census.fieldZonePop + census.
         indfieldZonePop ) * fieldMaintenanceCost;
8
9    var roadCost = census.roadTotal * roadMaintenanceCost;
10   var railCost = census.railTotal * railMaintenanceCost;
```

```
11    this.roadMaintenanceBudget = Math.floor((roadCost + railCost) *
          RLevels[gameLevel]);
12
13    this.taxFund = Math.floor(Math.floor(census.totalPop * census.
          landValueAverage / 120) * this.cityTax * FLevels[gameLevel]);
14
15    if (census.totalPop > 0) {
16      this.cashFlow = this.taxFund - (this.policeMaintenanceBudget +
            this.fireMaintenanceBudget + this.roadMaintenanceBudget + this
            .fieldMaintenanceBudget);
17      this.doBudgetNow(false);
18    } else {
19      // We don't want roads etc deteriorating when population hasn't
            yet been established
20      // (particularly early game)
21      this.roadEffect   = this.MAX_ROAD_EFFECT;
22      this.policeEffect = this.MAX_POLICESTATION_EFFECT;
23      this.fireEffect   = this.MAX_FIRESTATION_EFFECT;
24      this.fieldEffect  = this.MAX_FIELD_EFFECT;
25    }
26  };
```

Listing 4.19: Listato della funzione `collectTax` definita all'interno del file `budget.js`

Come si nota, la variabile `fieldMaintananceBudget` é il valore della tassa totale da pagare, che dipende dal numero dei campi indipendenti o legati alla rete idrica presenti nella mappa, il quale sará poi moltiplicato per un costo fisso impostato, per semplicitá, a 100$. Dopo di che si procede al calcolo degli altri costi di manutenzione: quelli relativi alla strada, al dipartimento di polizia e alla caserma dei pompieri.

La variabile `taxFund` indica il ricavo ottenuto dall'utente tramite le tasse; successivamente si verifica un controllo relativo alla popolazione: non avverrá il pagamento delle tasse se la popolazione é nulla. Si nota, infatti, che i diversi valori di `effect` vengono valorizzati al massimo, questo indica che l'efficienza di tali strutture non calerá e non ci sará degrado.

Al contrario, se una popolazione é presente nella cittá gestita dall'utente verrá effettuato il calcolo all'interno della variabile `cashFlow`, che rappresenta il flusso di denaro formato dalle entrate meno le uscite. Successivamente viene effettuato il pagamento delle tasse tramite la funzione `doBudgetNow`.

La situazione di degrado o mantenimento del campo viene definita dalla funzione `shouldDegradeField` che ritorna un valore *booleano* `true` nel momento in cui l'efficienza del campo (`fieldEffect`) é minore di un certo valore costante.

```
1   Budget.prototype.shouldDegradeField = function() {
```

```
2   return this.fieldEffect < Math.floor(15 * this.MAX_FIELD_EFFECT /
        25);
3 };
```

L'inefficienza del campo viene notificata dalla funzione `fieldFound` all'interno del file `field.js`. Tramite un controllo `if`, se `shouldDegradeField` risulta positivo si esegue la funzione `degradeZone`, che cambia la *tile* in campo non coltivato. Ogni volta che verrá chiamata la funzione `fieldFound`, si eseguiranno sempre le istruzioni di degrado, almeno fino a quando il campo avrá bassa efficienza.

```
1 var population = 0;
2   if(tile != Tile.FREEF) {
3   if(simData.budget.shouldDegradeField()){
4     lpValue = ZoneUtils.getLandPollutionValue(simData.blockMaps, x, y)
          ;
5     degradeZone(map, x, y, simData.blockMaps, population, lpValue,
          zoneIrrigate);
6     return;
```

Listing 4.20: Estratto della funzione `fieldFound` all'interno di `field.js`

```
1  var degradeZone = function(map, x, y, blockMaps, population, lpValue,
       zoneIrrigate) {
2    var tileValue = map.getTileValue(x, y);
3    switch(tileValue){
4      case Tile.CORN:
5        tileValue = Tile.FCORN;
6        map.setTile(x, y, tileValue, Tile.BLBNHYBIT | Tile.ZONEBIT);
7        break;
8
9      case Tile.WHEAT:
10       tileValue = Tile.FWHEAT;
11       map.setTile(x, y, tileValue, Tile.BLBNHYBIT | Tile.ZONEBIT);
12       break;
13
14     case Tile.ORCHARD:
15       tileValue = Tile.FORCHARD;
16       map.setTile(x, y, tileValue, Tile.BLBNHYBIT | Tile.ZONEBIT);
17       break;
18
19     case Tile.POTATO:
20       tileValue = Tile.FPOTATO;
21       map.setTile(x, y, tileValue, Tile.BLBNHYBIT | Tile.ZONEBIT);
22       break;
23
24     default:
25       return;
26
27   }
28
29   return;
```

```
30  };
```

Listing 4.21: Listato della funzione `degradeZone`

Capitolo 5

Conclusioni e sviluppi futuri

5.1 Conclusioni

Lo sviluppo del videogioco non si puó dire certo concluso; quello da noi svolto é da considerarsi un primo passo dentro un codice sicuramente complesso e molto articolato. Nonostante ció, il lavoro di collaborazione ha prodotto un buon risultato, con l'introduzione delle strutture trattate nell'ambito agricolo, strutture che, nonostante scarne logiche e con basso impatto sull'economia della cittá, risultano funzionanti e completamente indipendenti dagli altri elementi presenti nel gioco. Infatti, il blocco di elementi che sono stati implementati puó sopravvivere senza problemi all'interno della simulazione.

5.2 Sviluppi futuri

I piú probabili sviluppi riguardanti il programma presentato potrebbero concernere l'introduzione di costi annuali diversi per ogni tipo di coltivazione, contribuendo anche all'abbassamento generale dell'inquinamento per ogni campo presente nella mappa. Riteniamo, inoltre, che sarebbe opportuno far sí che la costruzione e il mantenimento di un campo si accompagnino anche a una crescita della popolazione totale concomitante con il miglioramento del terreno. Sicuramente si dovrebbe anche rendere piú dinamico il ruolo del `WWTP` magari introducendo piú strutture che necessitano di fabbisogno idrico.

Le vie che saranno percorse andranno nella direzione della sostenibilitá ambientale e delle energie rinnovabili, magari cambiando anche l'obbiettivo totale del *Serious Game* dal gestire una cittá sempre piú ampia e produttiva al raggiungimento di una societá a zero emissioni.

Bibliografia

[1] Cos'è la gamification. https://www.gamification.it/gamification/introduzione-alla-gamification/. Accessed: 2020-10-12.

[2] Analisi decisionale a piú criteri. https://www.hisour.com/it/multiple-criteria-decision-analysis-35202/. Accessed: 2020-10-12.

[3] Serious game: metodologie innovative per l'apprendimento esperienziale. https://www.diversity-management.it/2015/12/24/serious-game-definizione-vantaggi/. Accessed: 2020-10-12.

[4] Digital-water.city. https://www.digital-water.city/. Accessed: 2020-10-12.

[5] Graeme mccutcheon. http://www.graememcc.co.uk/. Accessed: 2020-10-12.

[6] Che cos'è e come funziona github. https://www.fastweb.it/web-e-digital/che-cos-e-e-come-funziona-github/. Accessed: 2020-10-12.

[7] Javascript. https://it.wikipedia.org/wiki/JavaScript. Accessed: 2020-10-12.

[8] Node Academy. Cos'è l'npm e come usarlo. https://www.nodeacademy.it/cose-npm-installazione-locale-globale-aggiornamento/, 2018.

[9] Giacomo Cerquone. Yarn vs npm – in cosa si differenziano oggi? https://italiancoders.it/yarn-vs-npm/, 2018.

[10] nodejs.org. What is the file 'package.json'? https://nodejs.org/en/knowledge/getting-started/npm/what-is-the-file-package-json/, 2011.

[11] GitHub Contributors. webpack. https://github.com/webpack, 2020.

[12] Gimp. https://www.gimp.org/. Accessed: 2020-10-12.