POLITECNICO DI MILANO
Corso di Laurea MAGISTRALE in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Tecniche di data mining e generazione procedurale per lo sviluppo su smartphone

Relatore: Prof. Pier Luca Lanzi
Correlatore: Ing. Daniele Loiacono

Tesi di Laurea di:
Emanuele Parini, matricola 765491
Federico Sannicolò, matricola 766324

Anno Accademico 2011-2012

Everybody is a genius.
But, if you judge a fish by its ability to climb a tree,
it will spend its whole life believing that it is stupid.
Albert Einstein

Ringraziamenti

Desideriamo esprimere un sentito ringraziamento al Prof. Pier Luca Lanzi per la disponibilita che ci ha dimostrato, per l'aiuto fornito durante la stesura di questo lavoro e per le tutte le opportunità che ci ha presentato. Ringraziamo inoltre l'Ing. Daniele Loiacono per l'interesse e il tempo che ci ha dedicato nelle analisi di *data mining*. Grazie anche a Sara Campagna di Microsoft Italia per averci fornito gli *smartphone* su cui abbiamo sviluppato la nostra applicazione.

Un importante ringraziamento va ai nostri genitori che, con il loro incrollabile sostegno morale ed economico, ci hanno permesso di raggiungere questo traguardo di nuovo.

Un ringraziamento particolare va a Claudio e Davide per aver contribuito allo sviluppo di BadBlood, per tutte le giornate di lavoro spese in ottima compagnia e per aver contribuito a creare un gruppo unito e affiatato.

Non possiamo dimenticare, infine, i familiari, gli amici, i compagni di corso e tutte le altre persone che ci sono state vicino e che hanno condiviso con noi questo periodo della nostra vita.

Riconoscimenti

- **Partecipazione a Imagine Cup 2012 con BadBlood, sezione Windows Phone Challenge, unica app italiana classificata top 100 nel mondo**
- **Primi classificati al concorso Sharecare del Centro Nazionale Sangue con l'applicazione BadBlood "la giuria ha ritenuto l'idea inedita e molto originale e ha voluto premiare il lavoro di progettazione del videogioco che costituisce una interessante sfida comunicativa di cui il settore trasfusionale ha bisogno"**
- **Menzione speciale "Pubblicità Progresso" grazie a BadBlood "per aver saputo declinare il tema della donazione di sangue attraverso la linea didattica attiva e vivace dell'edugame"**
- **Vincitori del concorso "New Game Designer 2012" dell'Università Statale di Milano e Politecnico di Milano**

Indice

Elenco delle figure

Sommario

In questa tesi presentiamo l'applicazione di tecniche di *data mining* e di generazione procedurale di contenuti applicate a BadBlood, il videogioco per *smartphone* che abbiamo realizzato per il corso di Videogame Design and Programming del Politecnico di Milano e in seguito presentato al concorso Microsoft Imagine Cup 2012. Dato il poco tempo a disposizione per la fase di *playtesting*, dovuto alle scadenze del concorso e al divieto di pubblicazione imposto dal regolamento, abbiamo fatto provare il videogioco durante due eventi pubblici in modo da ottenere quanti più dati possibili riguardo i livelli giocati. Successivamente, attraverso alcune tecniche di *data mining,* abbiamo valutato l'analisi della giocabilità per rilevare eventuali difetti di *level design* e di interazione con il giocatore. Abbiamo individuato almeno tre problemi riguardanti il ritmo di gioco, la progettazione dei livelli e la gestione degli *input* del giocatore.

Nei videogiochi moderni è presente un'elevata significatività dei contenuti, elemento rilevante anche nei videogiochi che vengono distribuiti su dispositivi mobili (*smartphone* e *tablet*). In questo contesto si rendono necessarie tecniche automatiche per la generazione di contenuti che abbiamo applicato al nostro videogioco: per questo motivo abbiamo implementato una variante originale degli algoritmi di *Procedural Content Generation*, o PCG. Il PCG si occupa della generazione procedurale di contenuti di gioco usando un processo casuale che, tramite l'utilizzo di alcuni parametri, rende possibile la generazione di un numero molto elevato di contenuti

di gioco. Implementando il PCG abbiamo ridotto il lavoro di progettazione dei livelli, realizzando un generatore automatico che in base ad alcuni parametri crea un elevato numero di contenuti per una modalità di gioco.

Capitolo 1

Introduzione

Il mercato dei videogiochi su piattaforme mobili (*smartphone* e *tablet*) è in rapido aumento, infatti è stato registrato un incremento del 46% delle quote di mercato tra il 2009 e 2010 corrispondenti ad una crescita nella spesa di 1.6 miliardi di dollari [1]. I videogiochi su dispositivi mobili nascono nel 1994 quando *Tetris* diventa il primo gioco su telefono, l'Hagenuk[1] MT-2000. Tre anni dopo viene esportato *Snake* su dispositivi Nokia[2], diventando uno dei giochi più diffusi con più di 350 milioni di installazioni [2]. Nel 2000 viene introdotto sul mercato l'iPAQ[3], un *Pocket-PC* basato sul sistema Windows, che possiede una grafica migliore senza fornire la funzionalità di telefono. Nel 2002, Handspring's[4] Treo 180 diventa il primo *smartphone* che combina le funzionalità di un tipico *Personal Digital Assistant* (PDA) con quelle di un telefono mobile, facendo nascere una nuova linea di prodotti di successo che si è conclusa nel 2008 con il Palm Treo Pro. Questi dispositivi avevano una limitata capacità multimediale che rendeva possibile l'esecuzione di semplici giochi *puzzle* in due dimensioni. Nel 2007, il rilascio dell'Apple iPhone e il conseguente lancio di Apple AppStore a metà del 2008 crea uno standard de-facto per la distribuzione digitale di videogiochi per il *mobile*. Nello stesso periodo sono nati i primi motori grafici 3D basati su *OpenGL*, *Unity3D*[5] e

[1] http://www.hagenuk-germany.de/
[2] http://www.nokia.com
[3] http://en.wikipedia.org/wiki/IPAQ
[4] http://www.handspring.com/
[5] http://www.unity3d.com/

UDK[6], rendendo possibile lo sviluppo di giochi di alto livello su dispositivi mobili. Android e Windows Phone hanno presto seguito lo stesso percorso.

L'implementazione di videogiochi per dispositivi mobili richiede tempi di produzione stretti e brevi cicli di sviluppo rispetto ai tipici giochi per *console* portatili o titoli per PC e coinvolge un numero limitato di sviluppatori e grafici. La maggior parte delle compagnie operanti in questo campo tendono a sviluppare giochi semplici in breve tempo piuttosto che focalizzarsi su prodotti più grandi che richiedono tempi più lunghi [3]. Questo riduce il tempo a disposizione per il *play-test* rendendo necessaria una frequente distribuzione digitale per aggiornare e migliorare il gioco.

1.1 Il data mining nei videogiochi

Il *data mining* è un utile strumento per gli sviluppatori di videogiochi perché permette di analizzare i dati raccolti da un numero limitato di sessioni di gioco per ottenere *feedback* immediati sulla giocabilità e possibili difetti nel design. Diversi autori hanno utilizzato il *data mining* per studiare il comportamento dei giocatori sfruttando diverse tecniche di analisi dei dati per progettare e migliorare videogiochi [4]. Ad esempio, il *data mining* è stato impiegato nell'analisi dei dati raccolti dai giocatori durante le partite di *Tomb Raider Underworld* allo scopo di studiare ed eventualmente predire il comportamento dei giocatori stessi [5]. Un altro esempio è l'utilizzo del *data mining* in *Unreal Tournament* 3 per studiare le strategie di selezione delle armi [7] e per rilevare comportamenti scorretti [8]. Analogamente, Weber e Mateas [9] hanno inoltre analizzato grandi raccolte di dati per predire le strategie dei giocatori in *Starcraft*. Thurau e Bauckhage [10] hanno applicato il *data mining* per estrarre dati riguardanti le interazioni sociali nei MMORPG (*Massive Multiplayer Online Role-Playing Game*), ovvero giochi di ruolo che vengono svolti tramite Internet da più persone contemporaneamente.

Attualmente, non esistono lavori pubblicati che utilizzano il *data mining* per analizzare i dati raccolti dai videogiochi per piattaforme mobili (*smartphone* e *tablet*). In particolare non esistono applicazioni del *data mining*

[6] http://udk.com/

su *smartphone* che permettano l'analisi del *game design*. In questo lavoro, presentiamo un'applicazione del *data mining* per l'analisi della giocabilità in BadBlood, il videogioco che abbiamo sviluppato per il concorso Microsoft Imagine Cup 2012[7].

Il gioco è stato sviluppato in quattro mesi, formando un gruppo di lavoro composto da tre programmatori e un grafico. Il tempo limitato dovuto alle scadenze del concorso non ci ha permesso di svolgere una fase di *play-test* esaustiva, quindi abbiamo deciso di utilizzare delle tecniche di *data mining* per scoprire eventuali errori nel design di gioco, identificare problemi di giocabilità e assicurarci che non ci fossero significativi cambiamenti in base al dispositivo utilizzato (ricordiamo che, a differenza degli iPhone, i dispositivi Windows Phone sono dotati di hardware e dimensioni non uniformi perchè realizzati da case produttrici diverse). Inoltre, non abbiamo potuto rilasciare distribuzioni digitali per correggere eventuali *bug* o errori di design perché non era possibile pubblicare il gioco prima della valutazione finale dei giudici (come indicato dal regolamento del concorso[8]). Per questi motivi, per aumentare la qualità del nostro lavoro, abbiamo deciso di instrumentare il codice in modo da raccogliere quante più informazioni possibili riguardo la giocabilità allo scopo di eseguire dei *play-test* in occasione di due eventi pubblici.

Complessivamente non abbiamo rilevato differenze statisticamente significative nell'utilizzo dei dispositivi. Infine il processo di analisi ci ha permesso di identificare alcuni difetti nel *gameplay* del videogioco, i quali sono stati risolti, permettendoci di migliorare la giocabilità.

1.2 La generazione procedurale dei contenuti

La generazione procedurale dei contenuti, o PCG (*Procedural Content Generation*), si occupa della creazione di contenuti casuali in modo automatico[9]. Questi contenuti sono in generale relativi, ad esempio, ai livelli di gioco, ambientazione, personaggi e mappe. Lo sviluppo dei contenuti è stato un

[7] http://www.imaginecup.com/

[8] http://compete.imaginecup.com/docs/rules/ic13-official-rules-and-regulations—games-competition.pdf

[9] http://pcg.wikidot.com/

problema rilevante per l'industria dei videogiochi dai primi anni '80 quando le limitazioni di memoria delle piattaforme esistenti non permettevano la distribuzione di grandi quantità di contenuti pre-progettati come i livelli di gioco. Per questo motivo sono state largamente applicate delle procedure *ad hoc* per generare contenuti di gioco al momento dell'esecuzione, in modo da creare un numero molto elevato di livelli.

Oggi la generazione procedurale è largamente applicata nell'industria dei videogiochi e la sua importanza per lo sviluppo è radicalmente cresciuta per controllare i costi di progettazione e per permettere la nascita di nuovi tipi di gioco incentrati sulla generazione di contenuti. Questa è tipicamente basata su un approccio costruttivo nel quale il progettista sviluppa procedure algoritmiche *ad hoc* per generare una grande quantità di contenuti di gioco [11].

Uno dei primi esempi in quest'area è stato *Rogue*[10], un videogioco di ruolo con grafica ASCII in cui venivano generate proceduralmente un numero illimitato di mappe e nemici. *Elite*[11], un videogioco di commercio spaziale, forniva un ambiente espansivo con otto galassie, ciascuna contenente 256 stelle. Anch'esso usava un approccio procedurale per generare il contenuto di ogni galassia e richiedeva solo pochi *kilobyte* per memorizzare un intero universo. *Spore*[12] è un videogioco che tratta dell'evoluzione di una civiltà dalle prime molecole fino alla conquista delle galassie. In questo contesto vengono applicati degli algoritmi procedurali per creare e animare l'enorme varietà di creature possibili. *Borderlands*[13] è un videogioco sparatutto in prima persona che fornisce al giocatore un arsenale di tre milioni di armi generate usando un singolo vettore di parametri [12].

Il PCG che presenteremo nei prossimi capitoli è applicato in una modalità di gioco di BadBlood. Data la volontà di commercializzare il videogioco, ci siamo trovati nella necessità di creare un elevato numero di livelli. La generazione di contenuti procedurali ci ha portati a riprogettare in parte il *level design* di questa modalità, modificando alcuni elementi di gioco. Infatti

[10] http://en.wikipedia.org/wiki/Rogue_(computer_game)
[11] http://en.wikipedia.org/wiki/Elite_(video_game)
[12] http://www.spore.com
[13] http://www.borderlandsthegame.com

la versione originale permetteva eccessiva libertà al giocatore, il quale aveva a disposizione un numero elevato di soluzioni per ogni livello. Per questo motivo, prima di applicare il PCG, abbiamo introdotto ulteriori vincoli al gioco in modo da limitare il numero di soluzioni ottime. Gli stessi vincoli sono stati utilizzati come parametri per calibrare la difficoltà di un livello generato.

Nel nosto caso abbiamo scomposto il procedimento di PCG in tre sottoproblemi: generazione del livello, implementazione spaziale e ricerca delle soluzioni alternative. La prima fase consiste nella generazione di un livello seguendo le regole e i vincoli descritti dal gioco. L'implementazione spaziale riguarda la trasformazione del livello generato in una rappresentazione visiva, la quale condiziona la sua risolvibilità. L'ultimo sottoproblema analizzato, ovvero la ricerca di soluzioni alternative, è eseguito da un risolutore, ossia un insieme di algoritmi che analizza tutte le possibili soluzioni che il livello può possedere. Il risolutore rappresenta il componente più complesso della generazione dei contenuti.

1.3 Sommario

In questo capitolo abbiamo introdotto i concetti base trattati in questa tesi. Nei prossimi capitoli analizzeremo le tecniche di *data mining* e di generazione procedurale di contenuti applicate a BadBlood, il videogioco che abbiamo sviluppato.

La tesi è strutturata nel modo seguente:

Nel Capitolo 2 introduciamo la struttura e le principali caratteristiche di BadBlood.

Nel Capitolo 3 spieghiamo il concetto di *data mining* e commentiamo i risultati ottenuti dalla sua applicazione sui dati raccolti.

Nel Capitolo 4 analizziamo la generazione procedurale dei contenuti, le modifiche apportate a BadBlood e gli algoritmi utilizzati per il procedimento di generazione.

Nel Capitolo 5 illustriamo le tecnologie usate nell'implementazione di BadBlood, con particolare attenzione a Microsoft XNA e Silverlight.

Capitolo 2

BadBlood

In questo capitolo presentiamo la struttura e le principali caratteristiche di BadBlood, il videogioco per *smartphone* che abbiamo sviluppato per il corso di Videogame Design and Programming del Politecnico di Milano e in seguito modificato per esser presentato al concorso Microsoft Imagine Cup 2012. Al processo di sviluppo del videogioco han partecipato Davide Jones, Claudio Scamporlino, Alberto De Natale e Federico Pellegatta.

Il giocatore, servendosi del sistema immunitario, combatte contro le malattie più diffuse sulla Terra in diversi apparati del corpo umano (circolatorio, respiratorio, digerente e nervoso) attraverso quattro modalità di gioco (*Swipe, Tap, Free Drag e Puzzle*). In BadBlood ogni livello di gioco corrisponde ad una malattia da sconfiggere. I diversi livelli sono posizionati su sei continenti, uno per schermata; una volta selezionato il livello, il giocatore visualizza una schermata di istruzioni in cui vengono introdotti gli elementi di gioco.

Il nostro obiettivo è permettere al giocatore di imparare e condividere, attraverso i *social network* più diffusi, le informazioni apprese durante il gioco. Così come nel mondo si diffondono le malattie, allo stesso modo vogliamo rendere contagiosa la consapevolezza e la prevenzione delle malattie che si affrontano nel gioco e nella realtà quotidiana.

2.1 Selezione livello di gioco

Il giocatore sceglie il livello di gioco attraverso il menu di selezione livello che è composto da sei schermate, una per continente.

(a) Oceania (b) Nord America (c) Sud America

(d) Europa (e) Asia (f) Africa

Figura 2.1: Le schermate di selezione livello.

Nella figura 2.1 mostriamo le diverse schermate dei continenti da cui è possibile selezionare un livello del gioco. Su ogni continente sono posizionati i livelli raffigurati da un segnaposto di colore diverso a seconda della modalità di gioco:

- Rosso: modalità *Swipe*, ambientata nell'apparato circolatorio
- Verde: modalità *Tap*, ambientata nell'apparato respiratorio
- Giallo: modalità *Free Drag*, ambientata nell'apparato digerente
- Blu: modalità *Puzzle*, ambientata nell'apparato neurale

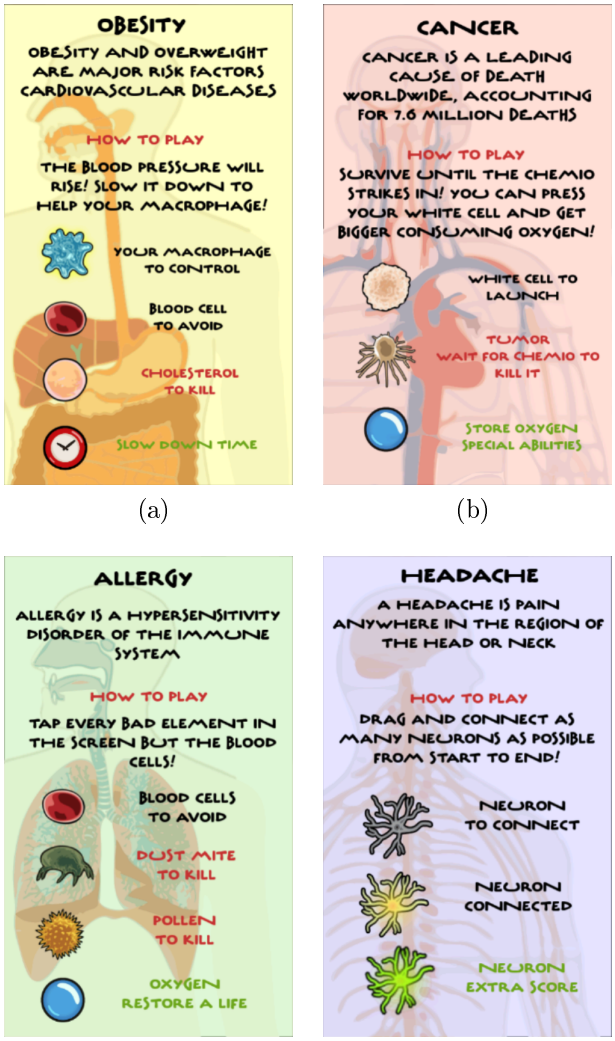
La posizione del segnaposto identifica in maniera approssimata una zona fortemente colpita dalla malattia; questa valutazione è stata fatta sulla base dei dati ottenuti dal *database* dell'WHO[1] (*World Healt Organization*). All'inizio del gioco tutti i continenti e i relativi livelli sono bloccati, eccetto l'Oceania che contiene i livelli di allenamento in cui il giocatore deve affrontare le malattie più comuni della Terra. I segnaposto su questo continente non indicano l'esatta zona di diffusione della malattia. Abbiamo scelto di mettere l'Oceania come primo continente poiché in un sondaggio fatto nel 2010 da "*The Economist Intelligence Unit*"[2], Australia e Nuova Zelanda sono risultate le migliori nazioni insieme alla Gran Bretagna nell'accesso ai servizi medici, qualità dei servizi e consapevolezza pubblica.

(a) (b)

Figura 2.2: Schermate informative sul continente

[1] http://www.who.int/gho/database/en/
[2] http://www.bbc.co.uk/news/health-10634371

Nella schermata in figura 2.2a, il giocatore ha la possibilità di visualizzare (premendo la "i" in basso a destra) le informazioni sugli stati del continente visualizzato riguardanti l'aspettativa di vita, il numero di medici ogni mille persone e la spesa del governo per i servizi sanitari. In BadBlood, abbiamo incluso due metriche di punteggio: la prima è basata su una valutazione numerica utilizzata per stilare una classifica mondiale dei giocatori, la seconda è basata su dei collezionabili ottenuti alla fine di ciascun livello il cui riepilogo viene visualizzato premendo sul riquadro in alto a sinistra, come mostra la figura 2.2b. Dopo aver selezionato il livello di gioco, viene visualizzata la schermata di istruzioni in cui vengono fornite informazioni riguardo la meccanica di gioco e i nemici da affrontare. In figura 2.3 mostriamo alcuni esempi di queste schermate.



(a) (b)

(c) (d)

Figura 2.3: Schermate di istruzioni.

2.2 Modalità di gioco

Dopo aver selezionato il livello e visualizzato le istruzioni, il giocatore inizia a giocare. Abbiamo sviluppato quattro modalità di gioco differenti per meccanica e ambientazione. Ciascuna modalità prende nome dalla *gesture* utilizzata per giocare.

2.2.1 Modalità Swipe

Questa modalità di gioco è ambientata nei vasi sanguigni dell'apparato circolatorio. Il giocatore deve combattere contro due tipologie di nemici (virus e batteri) controllando i globuli bianchi. L'obiettivo del gioco è colpire i nemici con i globuli bianchi che vengono generati dal sistema immunitario. Il giocatore deve affrontare ondate di nemici di varie dimensioni che attaccano con velocità e traiettorie diverse.

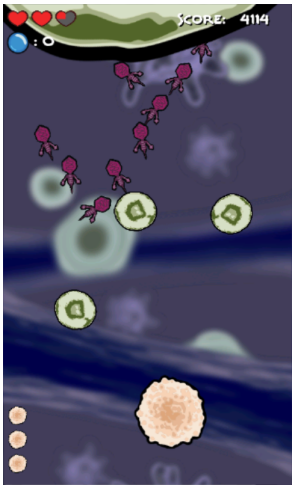


Figura 2.4: Modalità *Swipe*.

Nella figura 2.4 mostriamo la schermata di gioco della modalità *Swipe*. In alto a sinistra è presente un contatore di vite che viene decrementato ogni volta che un nemico raggiunge la parte inferiore dello schermo. Quando il contatore delle vite si azzera, il giocatore termina il livello con una sconfitta, se invece riesce mantenere il contatore delle vite positivo vince il livello. Sotto il contatore di vite è presente l'indicatore numerico del livello di ossigeno che, sopra un valore prefissato, permette al giocatore di far compiere l'ingrandimento o la mitosi al globulo bianco. Per ingrandimento intendiamo l'incremento di dimensione del globulo bianco, mentre per mitosi intendiamo la duplicazione del globulo che permette al giocatore di avere una capacità di fuoco maggiore. Nella parte inferiore dello schermo abbiamo posizionato un contatore di globuli bianchi che viene decrementato ad ogni lancio ed indica al giocatore quanti globuli può ancora lanciare. Il contatore serve a limitare

i lanci casuali: abbiamo posto un limite massimo sul lancio pari a tre globuli consecutivi, al superamento di questa soglia segue una penalizzazione in punti. Il punteggio viene visualizzato in alto a destra ed è calcolato in base alla posizione del nemico quando viene colpito: minore è la distanza rispetto al punto di generazione del nemico e maggiore sarà il punteggio.

In questa modalità di gioco abbiamo incluso tre tipologie di *powerup* (ovvero oggetti che danno al giocatore una particolare abilità temporanea quando vengono raccolti):

- ossigeno: serve per attivare le funzioni di ingrandimento e mitosi sul globulo bianco
- antivirale: permette di eliminare certe tipologie di virus
- antibiotico: permette di eliminare certe tipologie di batteri

2.2.2 Modalità Tap

Questa modalità di gioco è ambientata nei capillari degli alveoli polmonari nell'apparato respiratorio. In questa zona dei polmoni avviene lo scambio gassoso tra le sostanze inalate e il sangue. Il giocatore deve eliminare gli elementi nocivi inalati che variano a seconda del livello affrontato, ed evitare di colpire i globuli rossi.

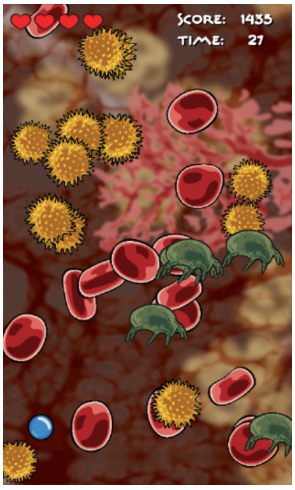


Figura 2.5: Modalità *Tap*.

Nella figura 2.5 mostriamo la schermata di gioco della modalità *Tap*. Abbiamo posizionato in alto a sinistra il contatore delle vite che si decrementa se un elemento dannoso raggiunge la parte inferiore dello schermo o se il giocatore elimina un globulo rosso. Il giocatore ha l'obiettivo di giocare per sessanta secondi mantenendo positivo il contatore delle vite. Il tempo rimanente viene visualizzato in alto a destra insieme all'indicatore del punteggio. Quest'ultimo è calcolato in base alla posizione del nemico quando viene colpito: minore è la distanza rispetto al punto di generazione del nemico e maggiore sarà il punteggio.

In questa modalità abbiamo incluso due *powerup*:

- ossigeno: permette di incrementare il contatore delle vite
- orologio: ferma il tempo e il movimento dei nemici in modo da facilitare la loro eliminazione

In alcuni livelli, per aumentare la difficoltà, abbiamo introdotto degli elementi di gioco, come ad esempio le nubi di fumo che molto lentamente tendono a coprire lo schermo, in modo da rendere più difficile l'individuazione dei nemici da eliminare.

2.2.3 Modalità Free Drag

Questa modalità di gioco è ambientata nell'apparato digerente. Il giocatore ha il compito di trascinare un monocita, ovvero una cellula del sangue che ingloba e ingerisce microrganismi e sostanze dannose per l'organismo. Il giocatore deve trascinare il monocita per colpire gli elementi infetti del sangue (virus e batteri), evitando il contatto con i globuli rossi. Quando un elemento nocivo esce dallo schermo, viene incrementata la percentuale di infezione nel sangue, indicata nella figura 2.6 in alto a sinistra.

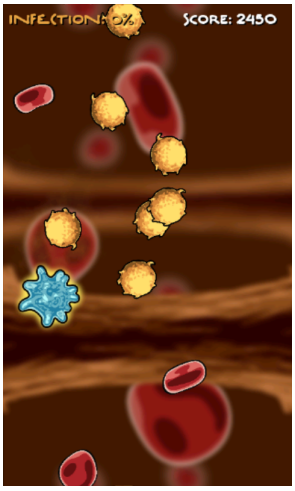


Figura 2.6: Modalità *Free Drag*.

Il gioco finisce quando il giocatore colpisce un globulo rosso oppure quando il livello di infezione del sangue raggiunge il 100%. In alcuni livelli, se il giocatore colpisce un virus senza aver equipaggato il monocita con un antivirale, il gioco termina con una sconfitta. Il punteggio viene visualizzato in alto a destra ed è calcolato in base alla posizione del nemico quando viene colpito: minore è la distanza rispetto al punto di generazione del nemico e maggiore sarà il punteggio.

In questa modalità abbiamo incluso tre diversi *powerup*:

- orologio *Up*: aumenta la velocità degli elementi presenti nel sangue
- orologio *Down*: diminuisce la velocità degli elementi in gioco
- antivirale: ha una durata di pochi secondi e può essere utilizzato per combattere specifiche malattie virali affrontate nel gioco.

2.2.4 Modalità Puzzle

Questa modalità di gioco è ambientata nel sistema nervoso. Il gioco consiste nel creare un percorso che parta dal neurone *START* e arrivi al neurone *END*, includendo il maggior numero possibile di neuroni. Il percorso è formato da una sequenza di collegamenti tra i neuroni: ciascun collegamento si ottiene toccando un neurone adiacente all'ultimo che è stato aggiunto alla sequenza.

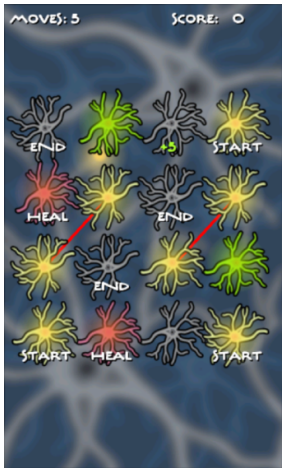


Figura 2.7: Modalità *Puzzle*.

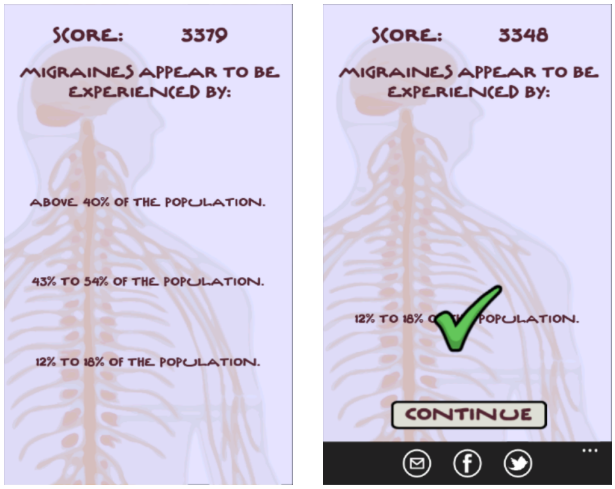
Nella parte superiore della schermata di gioco mostrata in figura 2.7, abbiamo posizionato un contatore di movimenti che si decrementa ogni volta che viene connesso un neurone. Il giocatore perde quando ha esaurito i movimenti oppure quando non è più in grado di raggiungere il neurone *END*. In questa modalità non abbiamo inserito *powerup* ma sono presenti diverse tipologie di neuroni:

- neurone classico: inizialmente non è illuminato, se attivato si illumina di giallo e decrementa il contatore dei movimenti
- neurone *waypoint*: è colorato di verde e permette di ottenere un punteggio maggiore se attivato
- neurone *heal*: è colorato di viola e se attivato elimina i collegamenti rossi che bloccano alcuni neuroni
- neurone bonus: decrementa o incrementa di un certo valore il contatore dei movimenti

2.3 Schermata di fine livello e quiz

Figura 2.8: Schermata di fine livello.

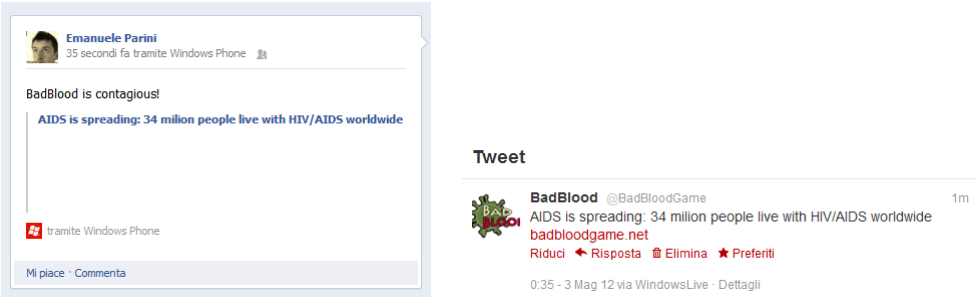
Dopo aver sconfitto una malattia viene visualizzata la schermata di fine livello, mostrata in figura 2.8. Nella parte inferiore dello schermo è possibile condividere su *Facebook*, *Twitter* oppure via *mail*, un insieme di informazioni riguardanti la malattia appena sconfitta. Queste informazioni sono tratte dal *database* dell'WHO. Scegliendo la voce *quiz* dalla schermata di fine livello è possibile rispondere a un quesito riguardante la malattia sconfitta per incrementare il punteggio numerico ottenuto nel gioco.



(a) (b)

Figura 2.9: Schermata di *quiz*.

Il *quiz*, mostrato nella figura 2.9, permette al giocatore di ottenere un incremento del 25% sul punteggio in caso di risposta corretta, altrimenti il punteggio viene abbassato in relazione al tempo impiegato: minore è il tempo trascorso prima di sbagliare la risposta, maggiore sarà la decurtazione dei punti. Questo metodo serve per penalizzare le risposte casuali. Dopo aver risposto, il giocatore può condividere il *quiz* con la risposta corretta attraverso i *social network*, pubblicando automaticamente domanda e risposta sul proprio *account*. In figura 2.10 mostriamo la pubblicazione delle informazioni riguardanti malattia appena sconfitta su *Facebook* e *Twitter*.



(a) (b)

Figura 2.10: Pubblicazione su *Facebook* e *Twitter*

2.4 Riconoscimenti

Con BadBlood abbiamo partecipato a Microsoft Imagine Cup 2012[3], una competizione Internazionale riguardante lo sviluppo di applicazioni che permettano di sensibilizzare e comunicare rilevanti tematiche sociali attraverso videogiochi o applicazioni per *smartphone*. Il tema del concorso è: "*Imagine a world where technology helps solve the toughest problems*". In questa competizione siamo giunti alla fase finale, posizionandoci tra i primi cento mondiali[4], primi in Italia.

Inoltre ci siamo classificati primi a *Sharecare*[5], un concorso promosso dal Centro Nazionale Sangue in collaborazione con l'Istituto Superiore di Sanità e il Ministero della Salute. Questo concorso consiste nella realizzazione di nuove forme di comunicazione per sottolineare l'importanza della donazione del sangue in Italia. Grazie a *Sharecare* abbiamo ottenuto una menzione speciale da Pubblicità Progresso "per aver saputo declinare il tema della donazione di sangue attraverso la linea didattica attiva e vivace dell'*edugame*".

Al momento BadBlood è disponibile gratuitamente sul *Marketplace* di Windows Phone[6].

2.5 Sommario

In questo capitolo abbiamo descritto le principali caratteristiche strutturali di BadBlood e le varie modalità di gioco che abbiamo sviluppato. Nel capitolo successivo descriviamo le analisi di *data mining* che abbiamo applicato alla modalità *Swipe* e *Tap*. La modalità *Puzzle* viene utilizzata come punto di partenza per l'applicazione di tecniche di generazione procedurale dei contenuti.

[3] http://www.imaginecup.com/
[4] http://www.imaginecup.com/Competition/Leaderboard.aspx
[5] http://www.centronazionalesangue.it/newsbox/share-care-condividere-la-cura
[6] http://www.windowsphone.com/it-IT/apps/cb9abd2b-0c21-461f-b547-7f9260f229a1

Capitolo 3

Datamining in BadBlood

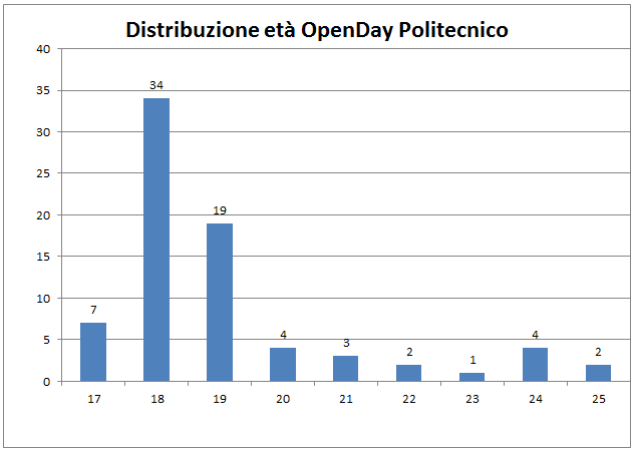
In questo capitolo illustriamo le metodologie di analisi e commentiamo i risultati ottenuti per identificare possibili problemi nel *design* di BadBlood. Abbiamo studiato come i giocatori interagiscono nelle diverse modalità di gioco e controllato se esiste una differenza significativa tra i dispositivi mobili che abbiamo utilizzato per i test: quattro telefoni HTC Surround forniti da Microsoft e un Samsung Omnia 7.

3.1 Raccolta dei dati di gioco

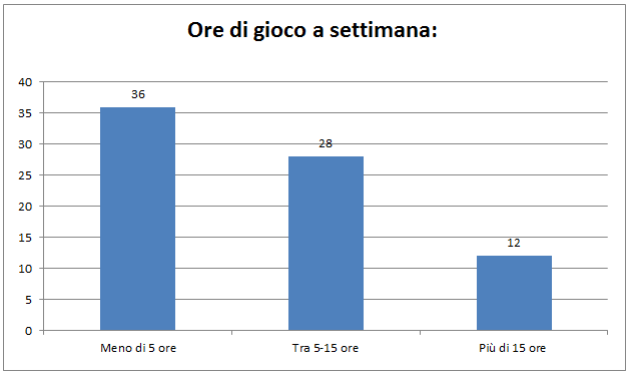
Per le analisi di *data mining*, ci siamo focalizzati su due principali modalità di gioco che richiedono un'elevata interattività con l'utente: le modalità *Swipe* e *Tap*. Abbiamo instrumentato il codice di BadBlood in modo da raccogliere ogni possibile informazione riguardo la giocabilità e l'interazione dell'utente in ogni livello. Prima di iniziare a giocare, venivano chiesti ad ogni utente età, sesso e un numero approssimato di ore di gioco a settimana. Per ogni modalità abbiamo registrato il livello giocato, i dati di tutte le *gesture* utilizzate (per esempio *tap*, *pinch*, *drag*) includendo anche la posizione di inizio e di fine di ogni *gesture*, la posizione e le caratteristiche di tutti i nemici, la posizione e natura di tutti i *powerup*, il numero di nemici uccisi/mancati in ogni istante di gioco, i dati registrati dall'accelerometro e infine tutte le informazioni riguardanti lo stato del giocatore (come per esempio il punteggio e l'energia rimanente). Tutte queste informazioni sono state campionate

ogni 200ms. Abbiamo utilizzato i dati raccolti per ricavare altre variabili come la distanza tra il punto di tocco e la posizione del nemico per valutare la precisione nella modalità *Tap*, l'angolo di lancio e il rispettivo rapporto tra elementi colpiti/mancati nella modalità *Swipe*. La raccolta dei dati è stata fatta durante Play! Modena[1], un importante evento italiano dedicato ai videogiochi e durante l'*Open day* del Politecnico di Milano il 31 Marzo 2012. Nelle figure 3.1a e 3.1b mostriamo rispettivamente le fasce d'età e le ore di gioco a settimana di ogni utente che ha testato BadBlood durante l'*Open day* del Politecnico di Milano. Durante i due eventi, abbiamo fatto giocare a circa duecento persone quattrocento livelli equamente distribuiti nelle due modalità di gioco. Notiamo che, sebbene quattrocento livelli sembrino fornire parecchie informazioni, la segmentazione degli utenti (bambini, adolescenti, adulti, etc.) soprattutto durante il Play! Modena, introduce un'enorme quantità di rumore. Per questo motivo, i duecento livelli giocati per ogni modalità sono a malapena sufficienti per ricavare alcuni *pattern* interessanti.

[1] http://www.play-modena.it/



(a)



(b)

Figura 3.1: Dati recuperati da PlayModena.

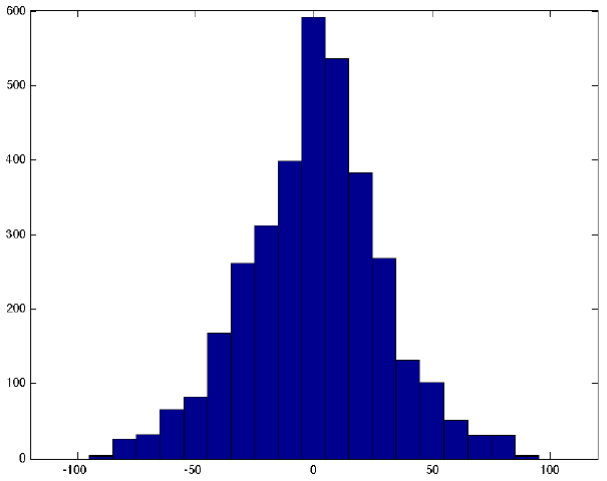
3.2 Analisi di base

Inizialmente, abbiamo eseguito un'analisi per controllare se, per esempio, lo schermo è stato utilizzato in maniera uniforme da tutti gli utenti. La nostra analisi non ha mostrato particolari differenze tra le due tipologie di piattaforme mobili utilizzate e fondamentalmente c'è un utilizzo uniforme degli schermi dei dispositivi. Abbiamo inoltre analizzato i dati ottenuti dagli accelerometri, notando una significativa differenza statistica tra la quantità di sollecitazioni sull'asse x (il meno sollecitato), y e z (il più sollecitato secondo l'accelerometro). Il risultato è coerente con la disposizione dei livelli nel

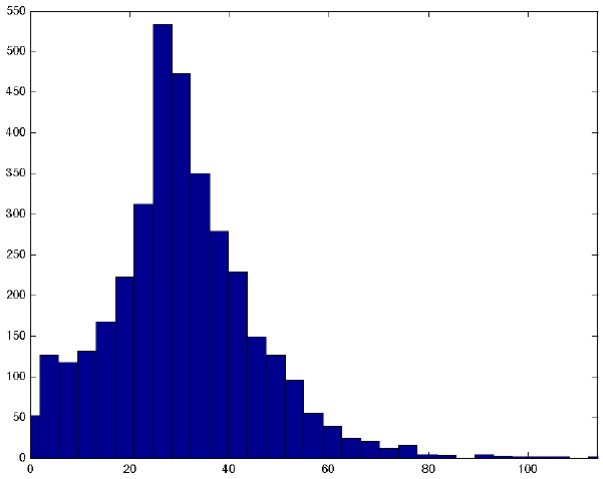
gioco i quali obbligano il giocatore a mantenere il dispositivo in verticale. Infine, non abbiamo registrato una significativa differenza statistica tra i due tipi di dispositivi (HTC Surround e Samsung Omnia 7). Questo conferma che, nonostante la differenza di marchio, i telefoni Windows Phone hanno essenzialmente le stesse caratteristiche *hardware* (una buona notizia per gli sviluppatori).

3.3 Modalità Swipe

In questo paragrafo ci focalizziamo sulla modalità *Swipe*, nella quale il giocatore combatte contro ondate di virus e batteri lanciando globuli bianchi. Il lancio viene effettuato per mezzo della *gesture* di *swipe* che consiste nel trascinare il dito da una posizione iniziale ad una finale. Per iniziare, abbiamo analizzato le interazioni degli utenti con l'interfaccia di gioco. Nelle figure 3.2 mostriamo la distribuzione delle *gesture* di *swipe*; i dati sono normalizzati assumendo che tutte le *gesture* partano dal centro della parte inferiore dello schermo. In particolare, in figura 3.2a mostriamo la distribuzione degli angoli di *swipe* in cui 0 corrisponde ad una direzione di lancio verticale, angoli negativi corrispondono a *swipe* verso la parte superiore di sinistra e angoli positivi corrispondono a *swipe* alla destra della parte superiore dello schermo. Dalla figura 3.2a notiamo che è presente del rumore nella parte destra dello schermo, ma questo non è statisticamente significativo per valutare l'indice di asimmetria sul test. In figura 3.2b mostriamo la distribuzione delle lunghezze delle *gesture* di *swipe* eseguite dagli utenti. Come si può notare, il movimento tende ad essere molto corto (tipicamente meno di 40 *pixel* in lunghezza) e le *gesture* lunghe sono rare. Dalle nostre osservazioni deduciamo che i movimenti lunghi sono tipicamente utilizzati da utenti inesperti che impugnano il telefono con una mano e utilizzano l'indice dell'altra mano per giocare.



(a)

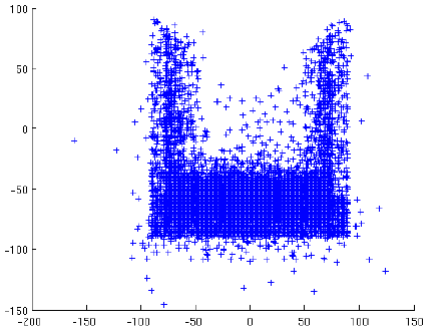


(b)

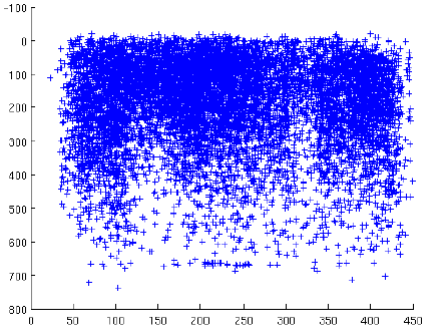
Figura 3.2: Distribuzione dell'angolo (a) delle traiettorie e lunghezza (b) dello *swipe*.

Successivamente, abbiamo analizzato la distribuzione dei colpi a segno su nemici e *powerup* per valutare possibili disturbi nel loro posizionamento che potrebbero influenzare in qualche modo la giocabilità. In particolare abbiamo studiato la distribuzione dei colpi sia rispetto al bordo dei nemici e sia rispetto alla localizzazione degli *hit* sullo schermo. Per posizione di *hit* intendiamo la coordinata in cui il rettangolo di collisione del globulo

bianco interseca quello del nemico o del *powerup*. Il rettangolo di collisione rappresenta la superficie occupata da un oggetto e il suo scopo è segnalare la collisione tra due oggetti quando i loro rettangoli di collisione si intersecano. In figura 3.3a mostriamo le posizioni di *hit* tra i globuli bianchi lanciati e ogni singolo nemico (il cui centro si trova in <0,0>). Possiamo notare che la maggior parte degli *hit* avvengono sul bordo inferiore del nemico e molti meno avvengono lateralmente. Non notiamo disturbo nella direzione di lancio e nei contatti con i nemici (il grafico in figura 3.3a è simmetrico), infatti anche gli *hit* laterali hanno una distribuzione uniforme. In alcuni rari casi, i nemici sono stati colpiti dall'alto a causa dei rimbalzi dei globuli lanciati sui bordi laterali dello schermo (cosa che non avevamo previsto). In figura 3.3b mostriamo la dispersione delle posizioni di *hit* rispetto allo schermo del telefono. Ci sono tre zone a maggior densità: una centrale e una per ogni lato (che probabilmente identificano due direzioni di lancio preferenziali). La parte superiore è più densa perché i giocatori tendono a colpire i nemici in maniera continua e costante non appena appaiono a schermo. Anche le zone laterali hanno elevata densità di *hit* a causa dei rimbalzi del globulo sui bordi dello schermo.



(a)



(b)

Figura 3.3: Distribuzione degli *hit* rispetto (a) alla posizione dei nemici (in <0,0>) e (b) allo schermo del dispositivo

Abbiamo inoltre applicato le stesse tipologie di analisi ai tre tipi di *target*, cioè batteri, virus e ossigeno, quest'ultimo visto come *powerup*. Abbiamo notato che l'ossigeno viene raramente colpito su rimbalzo dai bordi dello schermo e che i batteri sono leggermente più spostati verso la parte sinistra dello schermo. Questa considerazione ci fa intuire la presenza di uno squilibrio riguardante la generazione dei nemici.

Infine, abbiamo analizzato il ritmo di gioco: in particolare abbiamo studiato come il comportamento del giocatore viene influenzato dalle ondate di nemici. In figura 3.4 riportiamo il numero medio di nemici (i batteri sono rappresentati dalla linea rossa e i virus da quella blu), di globuli bianchi (rappresentati dai cerchi) e di *powerup* (linea verde). Nella modalità *Swipe*, il livello inizia solo con batteri e successivamente a metà partita arrivano i

virus. Come si può vedere dalla figura 3.4, l'utente tende ad avere un ritmo di fuoco stabile e nello schermo ci sono sempre due globuli bianchi in vita indipendentemente dal numero di nemici. Questo ci suggerisce che nella modalità *Swipe* l'utente tende a mantenere un ritmo di gioco costante.

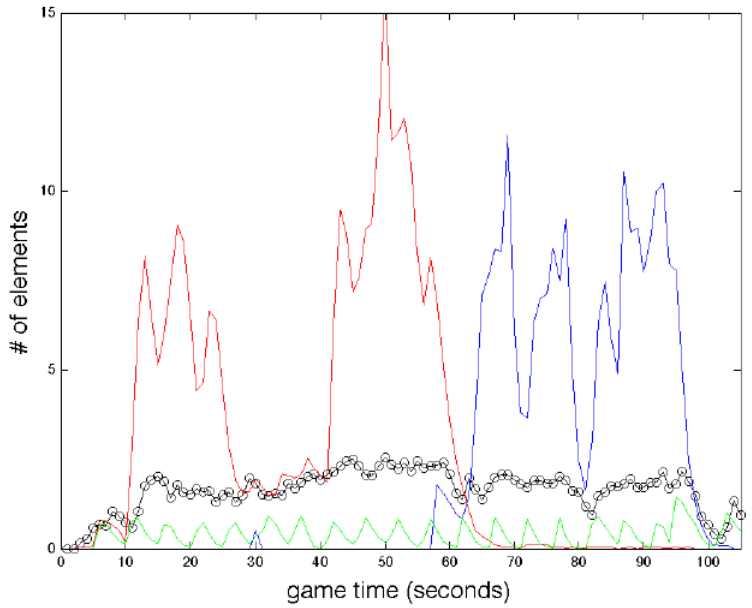


Figura 3.4: Paragone tra il numero medio di nemici, *powerup* e globuli bianchi.

Sulla base di questa analisi, abbiamo deciso di modificare il *design* dei livelli prima della sottomissione finale al concorso aggiungendo, in ogni livello della modalità *Swipe*, un combattimento contro un *boss*, ovvero un nemico di dimensioni notevolmente maggiori rispetto agli altri che richiede numerosi attacchi per essere eliminato. In questa situazione, il giocatore deve aumentare istantaneamente il ritmo di fuoco per distruggere il bersaglio evitando che il nemico raggiunga la parte inferiore dello schermo, infliggendo danno.

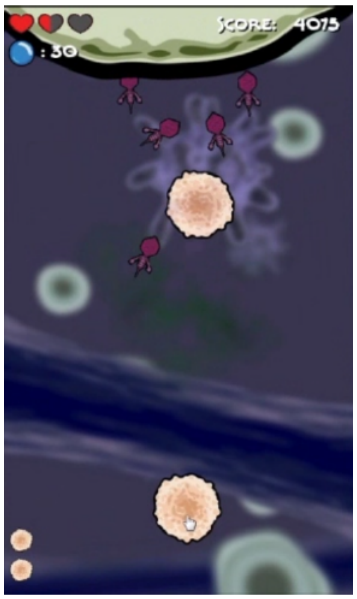
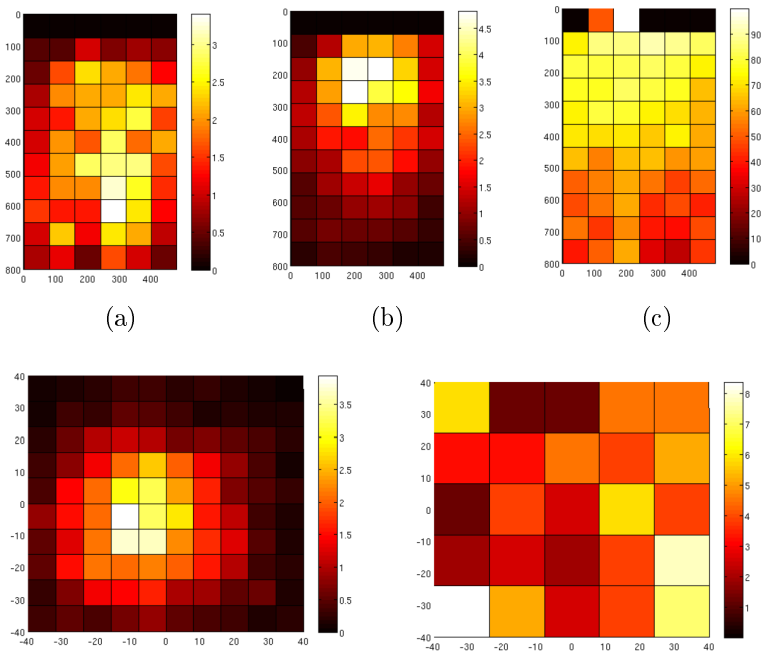


Figura 3.5: Introduzione del combattimento contro un *boss* nel gioco.

In figura 3.5 mostriamo una schermata di gioco in cui è stato aggiunto il combattimento con il *boss*, introdotto in maniera casuale durante un livello di questa modalità.

3.4 Modalità Tap



(a) (b) (c)

(d) (e)

Figura 3.6: Distribuzione dei tocchi degli utenti nella modalità *Tap*.

In questo paragrafo analizziamo la modalità *Tap*, in cui il giocatore deve toccare gli elementi pericolosi del sangue per distruggerli, evitando di eliminare i globuli rossi. In figura 3.6a mostriamo la distribuzione dei *miss* sullo schermo, ovvero tutti quei tocchi che non sono andati a buon fine perché non hanno eliminato il bersaglio. I colori più chiari indicano un elevato numero di *miss*; i dati sono riportati in percentuale rispetto a tutti i tocchi registrati. Notiamo che nella parte inferiore dello schermo ci sono delle posizioni che registrano una alta percentuale di *miss*, mentre le posizioni nell'intorno registrano una quantità di *miss* molto inferiore. In figura 3.6b notiamo la distribuzione degli *hit* sullo schermo, ovvero tutti quei tocchi che hanno avuto successo perché hanno eliminato il bersaglio. Come prima, i colori più chiari indicano un numero maggiore di *hit* e in questo caso consideriamo solo gli *hit* sui nemici, non sui globuli rossi che sono elementi positivi. Notiamo che la

maggior parte degli *hit* avviene al centro della parte superiore dello schermo. Comparando la figura 3.6b con la figura 3.6a precedente riguardante i *miss*, notiamo che la distribuzione degli *hit* è molto più spostata verso la parte alta dello schermo, da cui possiamo dedurre che il giocatore cerca di colpire i nemici non appena appaiono a schermo e se non ci riesce tende a tralasciarli, preoccupandosene in un secondo momento.

In figura 3.6c mostriamo il rapporto tra *hit* e *miss*, il colore più chiaro rappresenta una maggior percentuale di *hit*. Notiamo che c'è una maggior percentuale di *hit* nella parte alta dello schermo rispetto alla parte più bassa. Questo avviene sia perché la parte superiore dello schermo è più raggiungibile rispetto a quella inferiore (in cui il giocatore posiziona la mano) e sia per la meccanica di gioco (gli oggetti arrivano dalla parte superiore dello schermo). Sulla base di questa considerazione, abbiamo pensato di aumentare la difficoltà del gioco cambiando semplicemente la direzione e il punto di entrata dei nemici, facendoli per esempio entrare dal basso. Notiamo inoltre un leggero squilibrio alla destra della parte inferiore dello schermo (dove abbiamo un rapporto di *hit* e *miss* peggiore) che corrisponde a posizioni difficilmente raggiungibili dalle persone destrorse che usano il pollice per giocare.

In figura 3.6d mostriamo le posizioni in cui i nemici vengono colpiti rispetto al loro centro (posizionato in <0,0>). Notiamo che la distribuzione dei tocchi non è centrata nell'origine ma ha il suo picco nella parte in basso a sinistra rispetto al centro del bersaglio, infatti i giocatori raramente coprono il *target* con tutto il dito. Dal momento che i nemici arrivano dalla parte superiore dello schermo, gli utenti tendono a toccare i *target* leggermente più in basso rispetto al loro centro effettivo. Basandoci su questa considerazione abbiamo apportato una correzione alla giocabilità, allargando il rettangolo di collisione dei nemici a sinistra della parte inferiore del bordo in modo da migliorare la precisione dei giocatori.

In figura 3.6e mostriamo la distribuzione degli *hit* sui globuli rossi che penalizzano il giocatore (rispetto al loro centro in <0,0>). Questo può includere due tipi di errori: tocchi volontari sul bersaglio sbagliato (il giocatore non ha capito correttamente la meccanica di gioco) oppure involontari (il giocatore voleva colpire qualcos'altro ma ha colpito il bersaglio sbagliato).

Dalla distribuzione in figura 3.6e notiamo che la maggior parte dei tocchi sbagliati sono involontari poiché la distribuzione non è centrata sul bersaglio ma sparsa intorno al suo perimetro, con picchi di frequenze sparsi sul bordo. Abbiamo inoltre ripetuto l'analisi dei dati ottenuti dagli accelerometri nella modalità *Tap*, ottenendo risultati simili a quelli della modalità *Swipe*. Non ci sono significative differenze tra i dati dei due dispositivi che abbiamo utilizzato per testare le varie modalità.

3.5 Sommario

In questo capitolo abbiamo applicato tecniche di *data mining* ai dati ottenuti in seguito a numerose sessioni di gioco. In particolare ci siamo focalizzati sulla modalità *Swipe* e *Tap*, evidenziando alcuni problemi relativi al *level design*, apportando di conseguenza dei cambiamenti alla giocabilità. Nella modalità *Swipe* abbiamo introdotto uno scontro finale con un *boss* per variare il ritmo di gioco che altrimenti sarebbe rimasto troppo costante. Nella modalità *Tap* abbiamo modificato i movimenti dei nemici e abbiamo allargato il loro rettangolo di collisione in modo da aumentare la precisione dei giocatori, rendendo il gioco meno frustrante.

Capitolo 4

Generazione procedurale di contenuti

In questo capitolo introduciamo i principi della generazione procedurale di contenuti applicata ai *puzzle game*, in particolare ci focalizziamo sulla modalità *Puzzle* estratta da BadBlood. Nella prima parte del capitolo discutiamo del design di due versioni della modalità *Puzzle*, la prima inclusa in BadBlood, la seconda modificata per permettere la generazione procedurale dei livelli. La seconda parte del capitolo illustra la formalizzazione del metodo utilizzata per applicare la generazione procedurale e la relativa divisione in sottoproblemi, in particolare spiegheremo gli algoritmi scelti nelle varie fasi.

4.1 Introduzione

La generazione procedurale dei contenuti, o PCG (*Procedural Content Generation*), si occupa della creazione di contenuti casuali in modo automatico. Questo procedimento di generazione riscontra alcuni problemi nei videogiochi che coinvolgono molti vincoli, come nel nostro caso i *puzzle*, in cui bisogna garantire la risolvibilità del livello generato.

Prendendo come esempio *Refraction*[1], un videogioco educativo disponibile *online*, il lavoro svolto dagli sviluppatori dimostra che i vincoli del *puzz-*

[1] http://kongregate.com/games/GameScience/refraction

le possono essere facilmente incorporati nei processi generativi. Il potere espressivo dei vincoli, inoltre, ha reso possibile la generazione di contenuti di gioco affidabili e controllabili [14]. Abbiamo sfruttato queste conoscenze per progettare il PCG per la modalità *Puzzle* di BadBlood.

4.2 Procedural Content Generation

I generatori di contenuti possono essere progettati come processi direttamente costruttivi oppure come sistemi di generazione e test. I processi costruttivi garantiscono per costruzione alcune proprietà dei loro *output*, perché sono generati seguendo precise regole; tuttavia altre proprietà possono essere imposte selezionandole attentamente da alcuni *output* campione. I sistemi di generazione e test cercano invece di automatizzare questo processo, tuttavia sono spesso implementati come processi di ottimizzazione flessibili (come gli algoritmi genetici), i quali richiedono l'intervento umano per decidere precisamente quando un artefatto generato sia sufficientemente adatto all'uso. Nel nostro caso abbiamo scelto la prima strada, ovvero un processo costruttivo che fosse in grado di generare contenuti validi in base a pochi parametri da noi selezionati, senza l'intervento umano che validi i contenuti generati.

Per quanto riguarda il *level design*, dobbiamo considerare tre diversi problemi: la generazione di livelli (sotto specifici vincoli derivati dal *gameplay*), la trasformazione del livello in un problema spaziale (il quale deve essere sempre risolvibile) e l'implementazione di un risolutore per trovare soluzioni alternative. Parliamo di questo più avanti nel capitolo.

A livello di progettazione, lo scopo del risolutore è quello di eseguire un processo ripetitivo di ricerca di soluzioni, deducendo le loro proprietà, testandone la validità ed eventualmente scartandole se questa non è verificata. Più precisamente, i risolutori propagano i vincoli del *puzzle* durante il processo di ricerca delle soluzioni e questo spesso ne include la creazione di nuovi. Per ottimizzare la fase di ricerca il risolutore individua, tra tutte le possibili soluzioni, quelle non valide e le elimina prima di completarle.

4.2.1 Design modalità Puzzle

Lo scopo del giocatore nella modalità *Puzzle* di BadBlood è quello di ripristinare le connessioni tra alcuni neuroni spenti disposti a griglia, evitando di esaurire le risorse disponibili (indicate da *Moves* in alto a sinistra sullo schermo). Il giocatore deve creare un percorso che parta dal neurone identificato come *START* e termini al neurone *END*. L'insieme dei collegamenti neurali crea una sequenza di neuroni accesi; ogni nuovo collegamento neurale costa al giocatore una unità di risorsa. Maggiore sarà il numero di neuroni utilizzati nel cammino, maggiore sarà il punteggio finale. Il giocatore completa il livello quando raggiunge il neurone finale, altrimenti sarà costretto a ripartire in caso di esaurimento delle risorse residue oppure in assenza di mosse valide.

Ogni livello si svolge in una griglia 5x4 che presenta un numero variabile di neuroni, i quali possono essere colorati in modo diverso e accompagnati da una parola o un numero che ne identifica la natura, come visibile in figura 4.1.

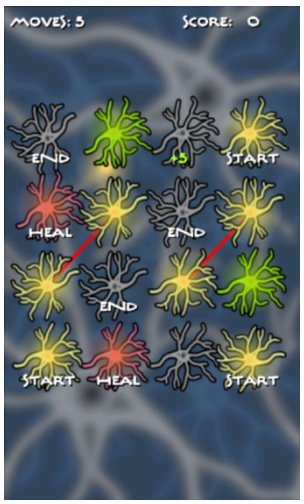


Figura 4.1: Modalità *Puzzle* in BadBlood.

Nella modalità originale utilizzata in BadBlood sono presenti quattro tipi di neuroni: classico, bonus, *waypoint* e *heal*. I neuroni classici si presentano come dei neuroni spenti grigi, i bonus sono identificati da un numero colorato che indica l'energia che verrà aggiunta (o sottratta) alla risorsa totale, i

waypoint sono colorati di verde e incrementano il punteggio del giocatore, gli *heal* hanno una tinta viola e permettono di curare i collegamenti rossi che bloccano alcuni neuroni.

Gli elementi elencati finora permettono eccessiva libertà al giocatore, il quale ha a disposizione un numero elevato di cammini per completare ogni livello. Per questo motivo, prima di applicare il PCG, la modalità *Puzzle* è stata modificata limitando il numero di soluzioni ottime e questo è stato possibile aggiungendo e cambiando il comportamento di alcuni neuroni, come visibile in figura 4.2.

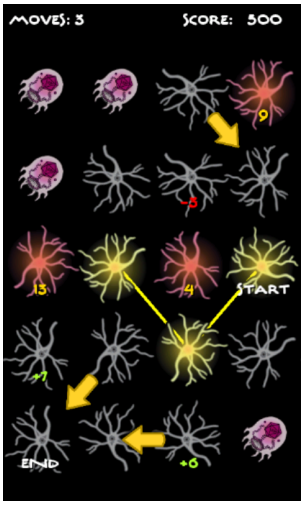


Figura 4.2: Modalità *Puzzle* modificata in versione PCG.

La nuova modalità *Puzzle*, a cui faremo riferimento per il resto del capitolo, presenta ancora i neuroni classici e quelli bonus, ai quali è stato aggiunto un neurone di tipo energetico, mentre i *waypoint* sono stati modificati. I neuroni energetici sono colorati di viola e per essere attivati richiedono che il giocatore abbia a disposizione un certo valore di risorsa, indicato da un numero giallo sottostante; in caso contrario non sarà possibile creare il collegamento neurale. I neuroni *waypoint*, invece, non hanno più una tinta verde, ma sono identificati da una freccia dorata che ne indica il successore: attivando un neurone *waypoint* il giocatore guadagnerà automaticamente un collegamento neurale anche con il suo successore, evento che consumerà un'altra unità di risorsa. Se la mossa non dovesse essere valida o se avrà

provocato l'esaurimento delle risorse, il giocatore dovrà ricominciare da capo il livello.

Il nostro obiettivo è produrre un generatore che possa ricreare autonomamente dei livelli che seguano il *level design* appena descritto, secondo uno stile e una complessità definita dal programmatore.

4.2.2 Formalizzazione

Per automatizzare la creazione dei livelli del nostro *puzzle*, abbiamo scomposto il problema in tre fasi. In questo contesto bisogna fare una distinzione tra missione e spazio. Una missione è un ordine logico degli obiettivi che il giocatore deve compiere per completare il livello, uno spazio è invece l'effettivo *layout* fisico del livello. Quindi le prime due fasi riguardano la produzione di missioni per i *puzzle* e il successivo incorporamento di queste missioni in una griglia, procedimento definito *grid embedding*. La fase finale riguarda la ricerca di tutte le soluzioni per i *puzzle* generati.

Generatore missione

L'obiettivo del generatore di missioni è quello di cogliere i dettagli di alto livello relativi alla progettazione del *puzzle*. Data la logica che compone i *puzzle* che dobbiamo generare, abbiamo scelto di rappresentare la missione come una lista di interi, i quali indicano univocamente i neuroni che compongono il livello. L'unico *input* di cui ha bisogno il generatore di missioni è l'intero che indica quanti neuroni comporranno il livello, valore che per motivi spaziali dovuti alla dimensione della griglia deve essere inferiore o uguale a 20.

A questo punto la lista viene inizializzata con il numero precedentemente fissato di neuroni identici, tra i quali il primo di essi verrà identificato come neurone *START*, mentre l'ultimo della lista sarà impostato come neurone *END*. La sequenza di neuroni generata rappresenterà sicuramente una soluzione ottima.

Una volta creata la struttura base del livello introduciamo i primi vincoli rappresentati dai neuroni bonus, ovvero quei neuroni che, quando attivati,

incrementano o decrementano la quantità di risorsa residua. Questi neuroni sono generati casualmente, sia per quanto riguarda la posizione nella lista sia per la quantità di risorsa che possiedono, ma sempre controllando la risolvibilità del livello, ovvero che sia possibile raggiungere il neurone *END* partendo da quello *START* senza mai esaurire la risorsa.

Il passo successivo è quello di impostare eventuali neuroni energetici, ovvero quei neuroni che possono essere attivati solo se la risorsa residua del giocatore è pari al valore indicato dal neurone. Posizione nella lista e numerosità di questi neuroni sono generate casualmente. Questo vincolo si calcola iniziando dal primo neurone e scorrendo la lista: partendo dalla risorsa iniziale, ad ogni passo e per ogni neurone attraversato si aggiorna la risorsa residua fino ad arrivare al neurone candidato, la cui energia richiesta, memorizzata nel parametro `EnergyRequired`, viene impostata allo stesso valore della risorsa residua.

Il vincolo finale riguarda i *waypoint*, ovvero quei neuroni che, alla loro attivazione, generano un collegamento automatico con il loro successore, indicato nel gioco da una freccia dorata. Questo vincolo, anch'esso generato casualmente per posizione e numerosità, non richiede nessun controllo specifico in quanto, quando sarà attivato, il successore del neurone *waypoint* sarà subito individuato incrementando l'indice numerico della lista (e.g. se il neurone *waypoint* è identificato dall'intero n, il suo successore avrà come indice n+1).

Grid Embedding

Il *grid embedding* risolve i vincoli spaziali ignorati nella generazione delle missioni, ovvero si occupa di posizionare la struttura ottenuta in una griglia bidimensionale. L'*input* principale per questo problema è la struttura generata al passo precedente. A livello teorico si potrebbe ottenere un numero molto elevato di soluzioni valide al problema del *grid embedding*, ma a noi interessa produrne solamente una. Per questo motivo la strategia utilizzata sfrutta un algoritmo di ricerca chiamato DFS, *Depth-First Search*[2] (ricerca in profondità).

[2] http://en.wikipedia.org/wiki/Depth-first_search

L'algoritmo di ricerca DFS espande progressivamente i nodi dell'albero di ricerca partendo dal primo figlio del nodo che appare nell'albero, andando sempre più in profondità finché un nodo obiettivo non viene rilevato. Se invece l'algoritmo si trova in un nodo senza figli, la ricerca risale lungo l'albero ritornando al nodo più recente sul quale non è stata terminata l'esplorazione e ne espande un altro figlio. La lista dei figli estratti da ogni nodo viene gestita in modalità LIFO (*Last In First Out*)[3], in cui i nuovi nodi vengono inseriti in cima alla struttura dati, causando la progressiva discesa della ricerca fino alla rilevazione di una soluzione. L'algoritmo 4.1 mostra un esempio in pseudocodice del DFS applicato a un grafo G dove v è un vertice di G.

Listato 4.1 *Depth-First Search*

```
1:  procedure DFS(G,v) :
2:    label v as expolored
3:    for all edges e in G.incidentEdges(v) do
4:     if edge e is unexplored then
5:      w ← G.opposite(v,e)
6:      if vertex w is unexplored then
7:        label e as a discovery edge
8:        recursively call DFS(G,w)
9:      else
10:       label e as a back edge
```

Il *grid embedding* prevede che tutte le soluzioni siano alla stessa profondità nell'albero di ricerca, infatti l'algoritmo dovrà compiere un numero di passi pari a quanti sono i neuroni che compongono una missione. Per questo motivo l'utilizzo dell'algoritmo DFS risulta più efficiente, in quanto la prima soluzione trovata sarà anche ottima. La profondità dell'albero di ricerca è definita, quindi non si possono verificare casi di non-terminazione dell'algoritmo [15].

Abbiamo implementato il *grid embedding* usando tre strutture dati: una lista di nodi per memorizzare il cammino attuale, una lista per i nodi ancora da espandere e una matrice bidimensionale di interi della stessa dimensione della griglia di gioco per tener traccia del posizionamento dei nodi. Ogni

[3] http://en.wikipedia.org/wiki/LIFO_(computing)

numero intero positivo contenuto nella matrice fa riferimento alla posizione di un neurone nella relativa lista (e.g. la cella della matrice contenente l'intero 1 rappresenta il neurone iniziale). Il posizionatore all'inizio genera casualmente due coordinate per selezionare la cella in cui posizioniamo il primo nodo, dal quale inizia il cammino. A questo punto, per procedere con l'algoritmo, bisogna espandere il nodo iniziale, ovvero si devono analizzare le celle della griglia adiacenti al nodo. Ciascuna di queste celle conterrà un nodo che verrà aggiunto alla lista dei nodi da espandere. Data la ripetitività dell'algoritmo di generazione delle celle adiacenti, per evitare di creare sempre lo stesso cammino, abbiamo utilizzato un algoritmo chiamato *Fisher-Yates shuffle*[4] per mescolare il vettore dei figli dell'ultimo nodo estratto. Lo pseudocodice di *Fisher-Yates* è mostrato nell'algoritmo 4.2, il quale prende in *input* un vettore a di n elementi.

Listato 4.2 *Fisher-Yates shuffle*

```
1: for i from n - 1 downto 1 do
2:   j ← random integer with 0 ≤ j ≤ i
3:   exchange a[j] and a[i]
```

Una volta inseriti tutti gli interi nella matrice di posizionamento, la soluzione è ritenuta valida. Questa matrice è sfruttata dal *grid embedding* per impostare le coordinate a schermo dei neuroni che compongono il livello.

Risolutore

L'ultimo sottoproblema del PCG riguarda l'individuazione di tutte le soluzioni partendo dal *puzzle* generato precedentemente dall'*embedder*. A differenza del *grid embedding*, il risolutore deve ricavare tutte le possibili soluzioni al problema e non solo una, per questo abbiamo utilizzato un approccio diverso sfruttando l'algoritmo di ricerca BFS, *Breadth-First Search*[5] (ricerca in ampiezza).

L'algoritmo BFS mira ad espandere ed esaminare tutti i nodi di un albero, ricercando ogni soluzione. Esso analizza l'intera struttura dati, partendo dal

[4] http://en.wikipedia.org/wiki/Fisher-Yates_shuffle
[5] http://en.wikipedia.org/wiki/Breadth-first_search

nodo radice ed espandendo progressivamente tutti i nodi figli posizionati allo stesso livello nell'albero di ricerca. Se il nodo analizzato possiede dei nodi figli, questi vengono aggiunti alla lista dei futuri nodi da espandere. A differenza del DFS, la lista dei figli estratti da ogni nodo viene gestita in modalità FIFO (*First In First Out*)[6], in cui i nuovi nodi vengono inseriti in fondo alla struttura dati, visitando ed eventualmente espandendo tutti i nodi di un certo livello prima di passare ai figli del livello inferiore. L'algoritmo 4.3 mostra un esempio in pseudocodice del BFS applicato a un grafo G dove v è un vertice di G.

Listato 4.3 *Breadth-First Search*

```
1:  procedure BFS(G,v) :
2:   create a queue Q
3:   enqueue v onto Q
4:   mark v
5:    while Q is not empty :
6:     t ← Q.dequeue()
7:     if t is what we are looking for :
8:       return t
9:     for all edges e in G.incidentEdges (t) do
10:      o ← G.opposite(t,e)
11:      if o is not marked :
12:        mark o
13:        enqueue o onto Q
```

Il risolutore non prevede che le soluzioni siano tutte alla stessa profondità nell'albero di ricerca, in quanto la lunghezza di una soluzione per il *puzzle* in questione può essere variabile. Infatti è possibile connettere i neuroni *START* ed *END* attraverso cammini di diversa lunghezza senza necessariamente connettere tutti i neuroni del livello: l'algoritmo deve analizzare tutti i possibili cammini dell'albero che portano ad una soluzione, ovvero un cammino che termina con il neurone finale. Per questo motivo abbiamo utilizzato l'algoritmo BFS data la sua completezza, ovvero la proprietà di trovare una soluzione se ne esiste una. Dopo aver analizzato tutti i nodi da espandere, l'algoritmo termina con la certezza di aver ricavato tutte le soluzioni esistenti.

[6] http://en.wikipedia.org/wiki/FIFO

In questo caso le soluzioni ottime saranno quelle di lunghezza pari al numero di neuroni presenti nel livello, ovvero che hanno attraversato tutti i neuroni senza ripetizioni.

Per quanto riguarda l'implementazione del risolutore, utilizziamo due liste: una per i cammini parziali (le soluzioni candidate) e una per i cammini validi che hanno raggiunto il neurone finale (le soluzioni). Con cammino si intende una lista di interi senza occorrenze che parte dal neurone iniziale (identificato dal valore 0), se inoltre termina con il neurone finale (identificato dal valore n-1, con n numero di neuroni nel livello) è anche una soluzione.

Il risolutore inizialmente genera un cammino contenente solo il nodo iniziale da cui dovranno essere cercati i nodi figli. In pratica, ad ogni passo, l'algoritmo dovrà analizzare le celle adiacenti all'ultimo nodo posizionato che rappresentano una possibile mossa valida da poter eseguire. Nella lista delle soluzioni candidate andranno inseriti tutti i nuovi percorsi che saranno il risultato del cammino attuale con la nuova mossa aggiunta in coda ad esso. Come spiegato precedentemente, ogni nuovo cammino deve essere aggiunto in coda alla lista delle soluzioni candidate per preservare la logica di scansione "in ampiezza" dell'algoritmo. Una volta analizzate le celle adiacenti che rappresentano mosse valide, il risolutore rimuove dalla lista delle soluzioni candidate il cammino che le ha generate. Stesso procedimento accade ai cammini che non sono in grado di generare mosse valide, i quali vengono scartati.

Data la caratteristica del problema, la ricerca potrebbe scoprire una soluzione in un qualunque punto dell'albero. Per questo motivo, appena il risolutore rileva una mossa valida che include il neurone finale, il cammino che la contiene sarà sicuramente una soluzione valida per il *puzzle*. Essendo una soluzione, il cammino non potrà essere ulteriormente allungato, per cui quest'ultimo sarà aggiunto dal risolutore alla lista delle soluzioni e non a quella dei percorsi da espandere.

Quando la ricerca avrà analizzato ed espanso tutti i nodi dell'albero, la lista delle soluzioni candidate risulterà vuota, quindi i cammini contenuti nella lista delle soluzioni saranno tutte le soluzioni possibili del *puzzle*. Questa lista è molto numerosa, quindi per analizzarla abbiamo deciso di attribuire

un valore numerico ad ogni soluzione.

4.2.3 Risultati

Le soluzioni sono valutate da una funzione obiettivo che calcola il livello di complessità della singola soluzione, generando un valore `float` che varia da 1 (soluzione banale) a 10 (soluzione ottima). La funzione obiettivo valuta la lunghezza della soluzione e gli eventuali neuroni-vincoli (energetici e *waypoint*) attraversati da essa. Approssimando i valori delle soluzioni alla loro parte intera, abbiamo diviso i risultati ottenuti in dieci classi, ognuna di intervallo 1. Analizzando i dati ottenuti abbiamo osservato come la distribuzione sia approssimabile ad una gaussiana con le soluzioni banali e quelle ottime posizionate sulle due code.

Per quanto riguarda la relazione tra numero di neuroni-vincoli e soluzioni ottime, abbiamo constatato che maggiore è il numero di neuroni-vincoli utilizzati nella generazione, maggiore sarà la complessità del livello stesso: questo indica un minor numero di soluzioni, lo stesso discorso vale per le soluzioni ottime. Una maggiore complessità del livello può garantire un coinvolgimento elevato per il giocatore e questo non è un fattore da sottovalutare. Tuttavia, la relazione tra neuroni-vincoli e complessità non è valida in caso di un numero di neuroni-vincoli troppo elevato, se così fosse il *puzzle* sarebbe talmente complesso da garantire un numero esiguo di soluzioni, per cui la soluzione ottima (e probabilmente unica) sarebbe facilmente identificabile dal giocatore. Diventa quindi necessario un *tradeoff* tra la complessità del livello e il numero di neuroni-vincoli utilizzati per la generazione. Lo scopo è quello di generare dei livelli che possiedano poche soluzioni ottime e un discreto numero di soluzioni subottime, in modo da distogliere il giocatore dalla soluzione ottima, la quale risulterebbe evidente se il *puzzle* generato ne possedesse solo una.

4.3 Sommario

In questo capitolo abbiamo analizzato i principi della generazione procedurale di contenuti applicata ai *puzzle*. Abbiamo mostrato come la modalità *Puzzle* di BadBlood sia stata modificata per l'applicazione del metodo PCG, in modo da ottenere risultati più significativi. La procedura PCG ha richiesto una formalizzazione che ci ha portato a suddividere il processo di generazione dei livelli in tre fasi: generatore di missioni, *grid embedding* e risolutore. Per ogni sottoproblema abbiamo illustrato i vari algoritmi utilizzati e i motivi per cui li abbiamo scelti. Nella parte finale del capitolo abbiamo analizzato i risultati ottenuti dalla generazione, in particolare quello che ha prodotto il risolutore e il legame osservato tra i vincoli presenti nel livello generato e la sua complessità.

Capitolo 5

Tecnologie e implementazione

In questo capitolo illustriamo le tecnologie usate per l'implementazione di BadBlood. La principale è Microsoft XNA, un *framework* che fornisce librerie e classi per lo sviluppo di videogiochi e che permette di semplificare il processo di sviluppo di un'applicazione. La seconda parte del capitolo riguarda l'implementazione di alcuni elementi del videogioco.

5.1 Tecnologie utilizzate

In questa sezione descriviamo Microsoft XNA e Silverlight, due strumenti gratuiti che vengono messi a disposizione degli sviluppatori per facilitare la realizzazione di applicazioni per Windows Phone.

5.1.1 Microsoft XNA

Microsoft XNA (*Xna is Not an Acronym*) è un *framework* gratuito che fornisce librerie di classi C# e strumenti per la progettazione e lo sviluppo di videogiochi distribuibili per le piattaforme Windows, Windows Phone 7 e Xbox360 [16]. In particolare XNA, oltre alla programmazione, fornisce strumenti per la gestione dei contenuti grafici e il controllo degli *input* di cui parliamo in questo capitolo.

Ogni videogioco in XNA si basa su una classe chiamata `GameComponent`, la quale si occupa dell'esecuzione delle *routine* che rappresentano il nucleo

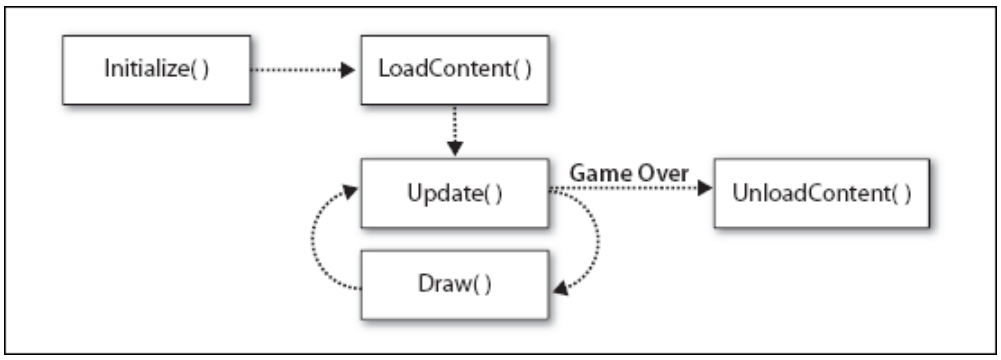


Figura 5.1: R*outine* che compongono il ciclo di controllo del videogioco.

del videogioco e hanno il compito di inizializzare le componenti di gioco e controllare l'esecuzione dell'applicazione.

Per spiegare il significato delle *routine*, consideriamo il "ciclo di controllo" di un progetto XNA. Ogni videogioco in XNA segue un flusso, rappresentato da una sequenza di chiamate a dei metodi, i quali eseguono compiti specifici. La prima operazione effettuata è definita inizializzazione e consiste nel preparare l'ambiente nel quale il gioco verrà eseguito, ad esempio inizializzando variabili o componenti di gioco. Una volta terminata l'inizializzazione, la successiva operazione ha lo scopo di caricare gli *asset* di gioco, quali le *texture* (ovvero gli elementi grafici, sfondi e *sprite*) e l'audio. A questo punto il gioco entra nel ciclo di controllo, che consiste in due metodi chiamati in sequenza all'infinito: il primo si occupa dell'aggiornamento degli elementi e della logica di gioco, il secondo disegna le *texture* e gli elementi grafici a schermo. Quando finisce il gioco viene eseguita l'ultima operazione che libera dalla memoria gli *asset* e altre risorse riservate in precedenza. La figura 5.1 mostra una semplificazione del ciclo di controllo.

Il metodo `Initialize` è utilizzato per inizializzare le variabili necessarie all'esecuzione e istanziare gli oggetti della classe `GameComponent`; l'operazione di inizializzazione è eseguita non appena ogni componente di gioco viene istanziato. Il gestore grafico `SpriteBatch`, i generatori di numeri casuali e il *parser* XML vengono istanziati in questo metodo.

Lo `SpriteBatch` si occupa della gestione dei contenuti grafici e fornisce diversi metodi di disegno e visualizzazione delle *texture*. Attraverso questo

strumento è possibile modificare l'aspetto delle *texture* tramite l'utilizzo di alcuni parametri, che permettono di variare la scala, aggiungere delle rotazioni e applicare un colore di tinta o delle trasparenze. È inoltre possibile controllare l'ordine di visualizzazione delle *texture* in caso di sovrapposizioni.

Il metodo `LoadContent` ha lo scopo di caricare gli *asset*, incluse le *texture* e i suoni. I contenuti vengono caricati tutti insieme in modo da essere sempre disponibili al momento dell'utilizzo, senza dover rallentare l'esecuzione. Questo metodo viene eseguito dopo il procedimento di inizializzazione.

Dopo il caricamento dei contenuti di gioco, l'esecuzione entra nel *game cycle* (o ciclo di gioco) che consiste nel richiamare continuamente le *routine* di aggiornamento e disegno: `Update` e `Draw`.

Il metodo `Update` esegue tutti i calcoli e i controlli necessari all'esecuzione del gioco, come i movimenti e la generazione delle *sprite*, gestione degli *input*, controllo delle collisioni tra gli oggetti, aggiornamento dei punteggi e tutto quello che riguarda la logica di gioco. Per poter gestire l'intera logica di gioco è necessario che questo metodo venga eseguito con una frequenza di almeno 30 volte al secondo.

Il metodo `Draw`, invece, ha il compito di disegnare le schermate di gioco e viene chiamato appena termina la *routine* di `Update`. Questo metodo sovrascrive il *frame* precedente e ne disegna uno nuovo ad ogni chiamata, dando l'illusione di un'animazione. Anche il metodo di `Draw` viene eseguito alla stessa frequenza del metodo `Update`, quindi se quest'ultimo subisce dei ritardi, la visualizzazione a schermo sarà ritardata e il giocatore avrà la sensazione di un gioco rallentato e poco fluido.

Non appena il gioco esce dal ciclo di chiamate di aggiornamento e disegno, viene eseguito il metodo `UnloadContent`. Questo metodo è usato per liberare le risorse caricate nel metodo `LoadContent` che richiedono una gestione diversa degli altri contenuti. Normalmente, XNA libera automaticamente le risorse, utilizzando un componente chiamato *garbage collector* che si occupa di eliminare le variabili e le allocazioni di memoria inutilizzate. Se il programmatore modifica le allocazioni di memoria in un modo particolare, sarà necessario liberarle manualmente nel metodo `UnloadContent`.

Terminata l'analisi del ciclo di gioco, definiamo uno strumento impor-

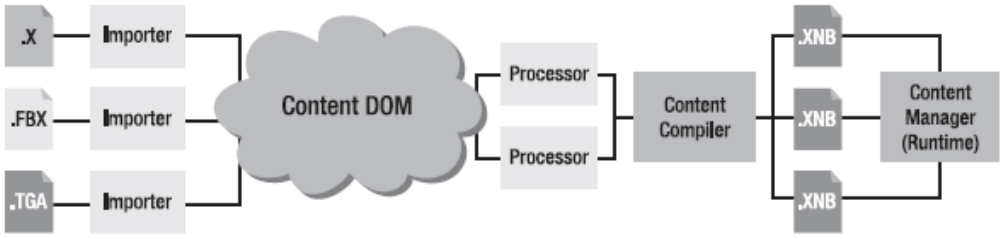


Figura 5.2: Componenti del *Content Pipeline Manager*.

tante che semplifica la gestione dei contenuti: il *Content Pipeline Manager*. Senza l'uso di XNA, dovremmo preoccuparci del caricamento dei componenti come audio e *texture*, ciascuno avente diversi formati ed estensioni. Il *Content Pipeline Manager* semplifica la gestione dei contenuti di gioco, in modo da poterli utilizzare con facilità: esso analizza i componenti presenti nella cartella dei contenuti del progetto, li compila e genera dei nuovi contenuti ognuno convertito in un formato a lui noto (*.xnb*).

Ogni volta che un contenuto è aggiunto al progetto, viene trasformato in un formato riconoscibile da XNA e spostato in una cartella all'interno del progetto adibita ai contenuti convertiti, in modo che il programma sappia sempre dove trovarlo e come utilizzare le informazioni al suo interno. Questo procedimento, mostrato in figura 5.2, semplifica l'accesso ai contenuti da parte del programmatore, il quale deve solo specificare il tipo dell'oggetto da caricare (sia esso una *texture* o un *file* audio) e il nome del *file*; sarà poi compito del *Content Pipeline Manager* gestirne il caricamento ed associarlo ad una variabile.

5.1.2 Microsoft Silverlight

L'ultima versione di XNA sfrutta la sinergia con Silverlight, un ambiente che permette di visualizzare contenuti ad alta interattività per piattaforme Windows e Mac. In Silverlight la gestione dei contenuti è basata su XAML (*eXtensible Application Markup Language*), un linguaggio basato su XML, utilizzato per descrivere l'interfaccia grafica delle applicazioni. Silverlight fornisce un *editor* il quale permette di modificare le componenti grafiche senza dover scrivere i *tag* che ne descrivono il codice [17].

Ora che abbiamo spiegato le componenti principali di XNA e il funzionamento di Silverlight, illustriamo alcuni aspetti riguardanti l'implementazione di BabBlood, il gioco per *smartphone* che abbiamo sviluppato.

5.2 Implentazione

In questa sezione spieghiamo l'implementazione dei principali elementi che compongono Badblood, dai menu fino alle quattro modalità di gioco, con riferimento alle tecnologie introdotte finora.

5.2.1 Menu in BadBlood

BadBlood combina l'utilizzo di Silverlight, per la gestione dei menu, e di XNA, per lo sviluppo del videogioco. Abbiamo scelto di utilizzare Silverlight per la facilità nel progettare i menu, per le funzionalità di navigazione disponibili e per le prestazioni.

Un esempio importante riguarda la schermata di selezione livello, nella quale mostriamo i sei continenti e le loro relative malattie da sconfiggere, ognuna associata ad un segnaposto colorato che ne rappresenta l'apparato nel corpo umano. La navigazione tra i continenti sfrutta una funzionalità fornita da Silverlight chiamata *Panorama*, ovvero lo scorrimento ciclico laterale tra schermate dello stesso menu. In questo modo è stato possibile posizionare i continenti in sequenza in una mappa a scorrimento. Sempre nello stesso menu, abbiamo utilizzato dei pulsanti per rilevare il tocco del giocatore in particolari aree della mappa, quali i segnaposto delle malattie e i pulsanti informativi che comprendono il riassunto dei collezionabili raccolti nei livelli di gioco e le informazioni del continente.

Anche il menu degli *Achievements* sfrutta il principio di scorrimento laterale della schermata di selezione livello. In questa schermata sono mostrati i progressi del giocatore nelle due metriche utilizzate, una è relativa al punteggio numerico ottenuto nei vari livelli, l'altra riguarda gli *achievement* sbloccati, ovvero dei trofei che il giocatore ottiene come premio in un particolare momento del gioco. Lo scorrimento laterale permette di navigare tra

la pagina dei punteggi e le tre categorie (*Basic, Advanced, Expert*) utilizzate per catalogare gli *achievement*.

Questa fusione tra Silverlight e XNA richiede che ogni schermata sia descritta da due *file* distinti: uno rappresenterà l'interfaccia grafica descritta utilizzando i *tag* e gli attributi XAML, l'altro conterrà il codice C# che dovrà gestire le funzionalità della schermata (definita come `PhoneApplicationPage`) e potrà interfacciarsi con le classi che compongono il videogioco.

Un esempio di questa dualità si riscontra nelle due già citate schermate di *Achievements* e selezione livello. La prima infatti, per visualizzare il progresso del giocatore attraverso i componenti grafici in XAML (e.g. pulsanti, caselle di testo), deve potersi interfacciare con una classe chiamata `AchievementsManager` che tiene traccia dei punteggi nei vari livelli e degli *achievement* sbloccati. Lo stesso discorso vale per la selezione livello, in questo caso la schermata si dovrà interfacciare anche con una ulteriore classe, ovvero il `ProgressManager`, che memorizza i continenti e le malattie sbloccate dal giocatore, le quali risulteranno colorate diversamente nella schermata del gioco.

Una particolare schermata di menu è la `GamePage`: è anch'essa divisa in due *file* distinti, ma non è dotata di contenuti XAML in quanto non possiede un contenuto statico. Essendo questa la schermata di gioco vera e propria, il suo intero funzionamento è descritto nel *file* contenente il codice C#, il quale gestisce il passaggio tra Silverlight e XNA.

5.2.2 Gesture

Per quanto riguarda l'implementazione del gioco, abbiamo sfruttato le diverse *gesture* che Windows Phone mette a disposizione, creando quattro diversi *gameplay*. BadBlood, infatti, si divide in quattro modalità di gioco: *Swipe, Tap, Free Drag, Puzzle*. Ogni modalità è gestita da un diverso `SpriteManager`, una classe che si occupa di gestire il *game cycle* il quale si occupa della logica di gioco e ne gestisce la visualizzazione delle *texture*. BadBlood sfrutta la varietà di *gesture* disponibili fornite dal *touchscreen* che è supportata da una libreria di XNA chiamata `Input.Touch`, la quale mette

a disposizione del programmatore diverse classi e metodi per la rilevazione delle interazioni con il `TouchPanel` del dispositivo. Queste interazioni possono essere dei tocchi generici, registrati come coordinate a schermo e raccolte dall'oggetto `TouchCollection`, oppure possono essere catalogate sfruttando le diverse *gesture* fornite da Windows Phone, le quali sono definite dall'enumerativo `GestureType`. Queste si differenziano principalmente in tocchi (singoli, doppi e prolungati) e trascinamenti (direzionali e a due dita). Le diverse *gesture* rilevabili dal dispositivo ci hanno permesso di creare diversi *gameplay* per le varie modalità di gioco che ora andiamo a spiegare.

Gesture modalità Swipe

La modalità *Swipe* si svolge nel sistema circolatorio del sangue e il giocatore deve combattere contro virus e batteri controllando i globuli bianchi.

In questa modalità vengono utilizzate diverse tipologie di *gesture* che permettono al giocatore di interagire con la cellula. Il *drag* (ovvero il trascinamento) permette di spostare il globulo orizzontalmente per mirare e verticalmente per effettuare un lancio contro i nemici che stanno attaccando. Il *pinch* (un movimento a due dita divergente) esegue la mitosi sdoppiando il globulo, operazione possibile previo raccoglimento di una prestabilita quantità di ossigeno, la risorsa presente in questa modalità. Tenendo invece premuto il dito sul globulo si attiva la *gesture* di *hold* che permette di aumentarne la dimensione, operazione che consuma dell'ossigeno.

Gesture modalità Tap

La modalità *Tap* si svolge nell'apparato respiratorio e il compito del giocatore è quello di toccare gli elementi dannosi (quali acari, polline, virus e altre particelle) per eliminarli. La caratteristica di questa modalità è l'assenza di un personaggio da controllare.

L'unica *gesture* utilizzata in questa modalità è il *tap*, che permette di rilevare le coordinate del tocco del giocatore sullo schermo. Se la coordinata corrisponde all'area di collisione di un elemento, questo viene eliminato.

Gesture modalità Free Drag

La modalità *Free Drag* si svolge nell'apparato digerente e il giocatore deve far sopravvivere un monocita raccogliendo gli elementi nocivi del sangue (come colesterolo, virus e particelle di alcool) evitando la collisione con i globuli rossi, elementi positivi.

L'unica *gesture* utilizzata in questa modalità è il *free drag*, che permette al giocatore, trascinando il dito, di muovere liberamente il monocita su tutta la superficie dello schermo.

Gesture modalità Puzzle

La modalità *Puzzle* si svolge nel sistema nervoso e il compito del giocatore è quello di ripristinare i collegamenti neurali tra alcuni neuroni spenti disposti a griglia, evitando di esaurire le risorse disponibili.

Anche in questa modalità si utilizza una sola *gesture*, il *tap*, che permette al giocatore di connettere i neuroni adiacenti toccando quello che si vuole connettere.

Su questa modalità di gioco è basata la generazione procedurale dei contenuti, in quanto è l'unica modalità che richiede dei precisi criteri sulla generazione dei livelli. Al contrario, nella modalità *Tap* e in parte nella modalità *Swipe*, i parametri che ne definiscono gli elementi di gioco sono generati casualmente.

5.2.3 Parallasse

Dopo aver descritto le *gesture* utilizzate nelle varie modalità, approfondiamo ora l'implementazione di alcuni elementi secondari del gioco. Il primo di essi è la parallasse.

La parallasse è il fenomeno per cui un oggetto sembra spostarsi rispetto allo sfondo se si cambia il punto di osservazione. Sfruttando questo principio è possibile dare la sensazione di profondità utilizzando delle *texture* sovrapposte dotate di trasparenza, facendole muovere a velocità differenti. La velocità sarà massima per la *texture* superiore e minima per quella inferiore. La classe che si occupa di tutto ciò è chiamata `ParallaxBackground`, la quale ad

ogni ciclo di `Update` ha il compito di aggiornare la posizione delle *texture* che compongono la scena di sfondo. Questo metodo è usato in tutte le modalità che richiedono un ambiente dinamico, tranne nella modalità *Puzzle* a causa della staticità dei neuroni. Per quest'ultima modalità la classe di parallasse è stata modificata per animare i neuroni sullo sfondo, illuminandoli ad intermittenza e simulando la trasmissione sinaptica tra di essi.

5.2.4 Effetti particellari

Un altro elemento di gioco che abbiamo implementato è rappresentato dagli effetti particellari, utilizzati per animare l'esplosione degli elementi presenti nel sangue, simulando la dissolvenza di una nuvola di particelle colorate. Questo effetto utilizza una *texture* che riproduce il fumo ed è disegnata con dei particolari effetti implementati nella classe `Particle` che si occupa di scala, durata, rotazione e velocità dell'effetto. Una ulteriore classe chiamata `ParticleSystem` si occupa della generazione degli effetti di particelle, ovvero ne disegna la *texture*, ne imposta il colore e ne inizializza i parametri definiti in precedenza. Questa classe gestisce anche la dissolvenza dell'effetto che si verifica con l'avanzare del tempo.

5.2.5 Rilevazione degli accelerometri per il data mining

Per poter applicare le tecniche di *data mining* abbiamo utilizzato i dati ottenuti dalle registrazioni degli accelerometri posti sugli *smartphone*. Per fare questo XNA ci fornisce una libreria chiamata `Microsoft.Device.Sensors`. Non appena il sensore individua un cambiamento nell'inclinazione notifica un evento, il quale deve essere rilevato da un gestore di eventi (un `EventHandler`) che lo assocerà ad un metodo definito dal programmatore. Una volta rilevato l'evento, il gestore di eventi chiamerà il metodo specificato precedentemente. Nel nostro caso, il suddetto metodo ha il compito di campionare su un intervallo specifico i dati raccolti dagli accelerometri durante il gioco ed aggiungerli ad un *file* in formato *.csv* insieme ad altre informazioni sulla partita.

5.3 Sommario

In questo capitolo abbiamo presentato le tecnologie utilizzate per la realizzazione di BadBlood e approfondito alcuni aspetti implementativi facendo riferimento alle librerie C# di Microsoft XNA e Silverlight.

Capitolo 6

Conclusioni e sviluppi futuri

In questo lavoro abbiamo illustrato l'applicazione di tecniche di *data mining* e di generazione procedurale dei contenuti su BadBlood. Non avendo potuto svolgere una fase di *play-test* esaustiva a causa delle scadenze del concorso Imagine Cup, abbiamo analizzato tramite *data mining* i dati raccolti in occasione di due eventi pubblici per scoprire eventuali errori o problemi nel design di gioco. Il processo di analisi ci ha permesso di rilevare e correggere alcuni difetti nel *gameplay* del videogioco in modo da migliorare la giocabilità.

Data la volontà di commercializzare il videogioco, ci siamo trovati nella necessità di creare un numero elevato di livelli. L'automatizzazione si è rivelata complessa a causa dei numerosi vincoli che contraddistinguono i *puzzle*. Per questo motivo abbiamo applicato la generazione procedurale dei contenuti che ci ha permesso di creare in maniera automatica i livelli di gioco, impostandone la difficoltà tramite la calibrazione di alcuni parametri.

Da Luglio BadBlood è disponibile sul *Marketplace* di Microsoft ed è in fase di approvazione una nuova versione che comprende dei miglioramenti di prestazioni per i nuovi *smartphone* con 256MB di RAM e ulteriori modifiche al *gameplay*.

Ad Ottobre rilasceremo una versione modificata di BadBlood con cui parteciperemo al "Festival della scienza" di Genova e a "Globulandia", iniziativa promossa dal Centro Nazionale Sangue.

Infine stiamo sviluppando una nuova applicazione basata su una variante della modalità *Puzzle* di BadBlood, che fornirà ampio supporto alla genera-

zione procedurale dei contenuti. Questa ci permetterà di studiare il comportamento dei giocatori davanti ai livelli procedurali e ci consentirà eventualmente di modificare da remoto i parametri di generazione dei livelli impostandone la complessità, rendendo più avvincente e appagante l'esperienza di gioco.

Bibliografia

[1] Newzoo. 2011. Newzoo games market report: Consumer spending on key platforms and business models. Available at http://corporate.newzoo.com/press/GamesMarketReport_FREE_030510.pdf.

[2] NCNokiaConnect. 2011. 7 nokia world records that will blow your mind! http://nokiaconnects.com/2011/02/15/ .

[3] T. Cheshire. 2011. How rovio made angry birds a winner (and what's next). *Wired.* http://www.wired.co.uk/magazine/archive/2011/04/features/how-rovio-made-angry-birds-a-winner

[4] D. Kennerly. 2003. Better game design through data mining. *Gamasutra.* http://www.gamasutra.com/view/feature/2816/better_game_design_through_data_.php.

[5] T. Mahlmann; A. Drachen; J. Togelius; A. Canossa; G.N. Yannakakis. 2010. Predicting player behavior in tomb raider: Underworld. In Yannakakis, G.N., and Togelius, J., eds., CIG, 178-185. IEEE

[6] A. Drachen; A. Canossa; G.N. Yannakakis. 2009. Player modelling using self-organization in tomb raider: Underworld. In Lanzi (2009), 1-8.

[7] L. Galli; D. Loiacono; P.L. Lanzi. 2009. Learning a context-aware weapon selection policy for unreal tournament iii. In Lanzi (2009), 310–316.

[8] L. Galli; D. Loiacono; L. Cardamone; P.L. Lanzi. 2011. A cheating detection framework for unreal tournament iii: A machine learning approach. In Cho,S.-B.; Lucas,S.M.; and Hingston, P., eds.,CIG, 266-272. IEEE.

[9] B.G. Weber; M. Mateas. 2009. A data mining approach to strategy prediction. In Lanzi (2009), 140-147.

[10] C. Thurau; C. Bauckhage. 2010. Analyzing the evolution of social groups in world of warcraft(r). In *Computational Intelligence and Games* (CIG), 2010 IEEE *Symposium on*, 170-177.

[11] L. Cardamone; D. Loiacono; P.L. Lanzi. 2011. Interactive Evolution for the Procedural Generation of Tracks in a High-End Racing Game.

[12] L. Cardamone; D. Loiacono; P.L. Lanzi. 2010. Automatic Track Generation for High-End Racing Games Using Evolutionary Computation.

[13] P.L. Lanzi. 2009. *Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Games*, CIG 2009, *Milano, Italy, 7-10 September, 2009*. IEEE.

[14] A. M. Smith; E. Anderseny; M. Mateas; Z. Popovic. 2012. A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game.

[15] S. Russell; P. Norvig. 2010. Artificial Intelligence: A modern approach, 3rd edition.

[16] A. Reed; 2010. Learning XNA 4.0, Game development for the PC, Xbox360, and Windows Phone 7.

[17] C. Petzold. 2010. Programming Windows Phone 7, Microsoft Silverlight Edition.