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

Generazione Automatica di Scenari per TORCS

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

Tesi di Laurea di:
Stefano Suardi, matricola 715857

Anno Accademico 2010-2011

Ringraziamenti

Un primo ringraziamento va ai miei genitori, Emanuela e Giovanni, e a mio fratello Andrea che mi hanno sempre motivato e sostenuto durante gli anni trascorsi al Politecnico. Ringrazio poi tutti i miei parenti, in particolare mia nonna, gli zii e le zie.

Un grazie speciale va a tutti i miei amici con cui ho condiviso momenti speciali della mia vita, come dimenticare tutte le vacanze passate insieme. Un grazie sincero va poi ai vari "colleghi" e amici che mi hanno accompagnato nel corso di questi anni Andrea, Alessandro, Massimo e Paolo. Che non solo mi hanno aiutato a studiare ma anche ad ammazzare il tempo in modo piacevole.

Un ringraziamento ulteriore va al mio relatore Pier Luca Lanzi e ai due correlatori Luigi Cardamone e Daniele Loiacono che hanno saputo spronarmi e tirare fuori il meglio di me, soprattutto nelle situazioni più insidiose contro le quali ho spesso sbattuto la tesa. Inoltre mi hanno sempre dato una mano assieme a preziosi consigli per svolgere questo lavoro al meglio.

Indice

Capitolo 1

Introduzione

Questa tesi si inquadra nell'ambito dei videogiochi, un settore in grande crescita negli ultimi anni, soprattutto per l'aumento dei videogiocatori e della domanda in continua evoluzione.

L'obbiettivo di questa tesi e' la generazione automatica di contenuti, un tema molto rilevante in ambito videoludico che porta a notevoli riduzioni nei costi di sviluppo.

Il maggior impiego di risorse per lo sviluppo si è quindi spostata verso l'utilizzo di una grafica sempre più spettacolare e realistica. Per limitare i costi della creazione di ambientazioni grafiche si è cercato fin da subito di automatizzare il tutto, ciò è possibile sfruttando algoritmi in grado di generare i contenuti grafici in modo pseudo-casuale, questi tecniche sono chiamate di *generazione procedurale dei contenuti*, però spesso si preferisce usare l'acronimo di derivazione anglosassone *PCG (Procedural Content Generation)*.

Con questo termine possiamo riferirci a molti campi di applicazioni, quindi dobbiamo operare una prima distinzione tra ciò che tratteremo effettivamente e ciò che invece esula dai nostri scopi.

Dobbiamo inoltre vedere queste tecniche esclusivamente come un ausilio alla produzione di contenuti grafici e non come a degli algoritmi in grado di realizzare scenari a partire dal nulla senza un'adeguata programmazione, quindi il ruolo dell'uomo non è del tutto eliminabile nel processo di produzione.

Le tecniche studiate in quest'ambito, inoltre, hanno molteplici vantaggi, primo tra tutti è la riduzione drastica dei costi di sviluppo delle componenti grafiche, il che significa poter generare dei videogiochi sempre più vari e qualitativamente migliori. Un ulteriore pregio nell'utilizzo di PCG è legato alla scarsa quantità di memoria necessaria per poter rappresentare scenari anche complessi, infatti, è sufficiente memorizzare l'insieme di parametri atti a generare un particolare livello e non tutti i minimi dettagli. Un aspetto di non secondaria importanza è la possibilità offerta nello sviluppo di nuove tipologie di videogiochi basati su questi concetti e tecniche. Inoltre PCG può aumentare l'immaginazione umana in quanto può dare spunti per la costruzione di contenuti sempre più complessi e dettagliati, in più potenzialmente si possono generare ambientazioni praticamente infinite e ciascuna di essa differente dalle altre per qualche dettaglio e quindi unica.

Lo scopo di questa tesi è perciò quello di automatizzare la generazione dell'ambientazione del videogioco di guida *TORCS*, acronimo di: *The Open Racing Car Simulator*, mediante l'utilizzo di tecniche procedurali.

La tesi è strutturata nel modo seguente.

Nel Capitolo 2 mostriamo alcune tecniche principalmente utilizzate nell'ambito di PCG e conseguentemente lo stato dell'arte.

Nel Capitolo 3 illustriamo gli obbiettivi della tesi e i problemi riscontrati.

Nel Capitolo 4 poniamo l'accento sul conseguimento dell'obbiettivo dal punto di vista pratico e quindi l'implementazione del framework.

Nel Capitolo 5 effettuiamo alcune valutazioni sul lavoro svolto e alcuni sviluppi futuri.

Nell'appendice A riportiamo alcuni diagrammi delle classi delle sezioni più rilevanti.

Nell'appendice B è presente una porzione del codice nelle sue parti più significative.

Nell'appendice C troviamo il manuale utente.

Capitolo 2

Generazione procedurale dei contenuti

2.1 Definizione e tassonomie

Per generazione procedurale dei contenuti si intende la creazione di contenuti in modo automatico, cioè mediante algoritmi. Questi contenuti sono in generale relativi a tutto il campo della *computer grafica* e quindi anche al campo dei videogiochi, oggetto di questa tesi.

In letteratura sono molti gli articoli che trattano di PCG anche se non vi è una comunità accademica dedicata a questo campo particolare, per esempio non esiste nessun libro specifico, anche se negli ultimi anni si sta osservando una crescita di interesse come dimostrato dalla nascita di una lista di distribuzione [41], una pagina wiki [45] e una task force dell'ente *IEEE CIS GTC* [44].

Una prima distinzione sui vari aspetti di questo campo è stata effettuata da J. Togelius, G, Yannagis, K. Stanley, C. Browne [28] che individuano alcuni aspetti caratterizzanti delle varie tecniche, andando a creare una tassonomia.

Innanzitutto possiamo distinguere tra le tecniche che producono il contenuto durante lo sviluppo del videogioco, in questo caso parliamo di generazione

offline, oppure durante l'esecuzione da parte dell'utente, questo è il caso di una generazione detta *online*.

Dal punto di vista del contenuto creato è bene separare ciò che è necessario per il completamento del gioco, da ciò che è opzionale, questa distinzione ci permette di trattare in modo differente le due categorie, dando maggiore importanza alla prima che deve funzionare sempre, pena l'impossibilità di finire il gioco.

Un'ulteriore distinzione può essere fatta a seconda che l'algoritmo di PCG si basi su un numero casuale di partenza, detto *seme*, oppure su un insieme di parametri che definiscono una generazione. In ogni caso la generazione può avvenire in modo stocastico oppure deterministico, in quest'ultimo caso a parità di parametri o seme iniziale verrà prodotto sempre lo stesso contenuto, mentre con la versione stocastica ciò non è garantito.

2.2 Principali tecniche utilizzate nell'ambito di PCG

Per comprendere al meglio come funzionano gli algoritmi di PCG è necessario mostrare alcune tecniche generali che è possibile utilizzare per procedere con la generazione del contenuto. Vediamo quindi alcune di queste tecniche.

2.2.1 Casualità

Data la necessità di produrre molti oggetti con lo stesso algoritmo è praticamente indispensabile utilizzare in qualche modo una funzione casuale, esistono però molte distribuzioni comuni tra cui scegliere, spesso però, queste non bastano quindi ne sono state inventate altre per applicazioni più specifiche.

Per esempio, in alcuni casi, si richiede che i numeri casuali generati siano in qualche modo dipendenti gli uni dagli altri, queste considerazioni ci portano a parlare di un rumore detto *Perlin Noise* [21], questo rumore, infatti, ha la caratteristica di mantenere una sorta di dipendenza spaziale. Possiede inoltre

la peculiarità di essere molto flessibile, è cioè possibile applicargli funzioni di vario tipo con lo scopo di variare ulteriormente l'effetto finale. Tutte queste qualità lo rendono il rumore per eccellenza nell'ambito di PCG e nelle sue applicazioni.

Inoltre nell'attuale architettura del calcolatore la generazione di numeri casuali presenta non poche difficoltà dovute principalmente alla natura deterministica degli stessi. Parleremo quindi spesso di numeri pseudo-casuali proprio per sottolineare la non perfetta aleatorietà delle funzioni generatrici di numeri casuali.

2.2.2 Grammatica geometrica

Sono delle grammatiche, nell'accezione tipica dell'informatica teorica, definite in uno spazio n-dimensionale (tipicamente $n \in (2, 3)$). Vengono utilizzate per la creazione di nuove forme partendo da alcune già esistenti mediante l'utilizzo di regole predefinite. Hanno il difetto di essere molto complicate da realizzare, quindi vengono utilizzate solo per generate figure semplici.

Sono state introdotte da George Stiny e James Gips [25] nel 1971 ma senza grandi applicazioni pratiche, oggi [16], invece, ne esistono molte di più.

2.2.3 L-systems

È una tecnica simile ad una grammatica ma con una grande differenza, tutti i caratteri non terminali vengono riscritti contemporaneamente, al contrario di quanto accade in una grammatica in cui si riscrive un solo carattere non terminale alla volta.

Questa peculiarità li rende adatti ad essere utilizzati per simulare la crescita di vari organismi viventi, è quindi molto interessante nell'ambito di PCG. Possiede inoltre la caratteristica di poter generare alcune tipologie di frattali (che vedremo nel prossimo paragrafo). Maggiori informazioni possono essere reperite nei lavori seguenti [15, 1].

2.2.4 Frattali

Un frattale è un oggetto geometrico che si ripete nella sua struttura allo stesso modo su scale diverse, quindi permette di sviluppare oggetti con un livello di dettaglio teorico infinito.

È un termine che è stato coniato dal matematico francese Benoît Mandelbrot [2] che fu il primo a studiarne il comportamento e a darne una definizione, un frattale è quindi un oggetto matematico definito da una funzione ricorsiva che ben si presta ad imitare il comportamento di molti fenomeni naturali, si vedano anche [13, 30].

2.2.5 Algoritmi genetici

Gli algoritmi genetici rappresentano un metodo euristico di ricerca e di ottimizzazione inspirati alla teoria di selezione naturale di Darwin [6], occorre citare il primo lavoro che introdusse il concetto di computazione evolutiva [3], e il primo vero lavoro riguardante gli algoritmi genetici [14].

2.2.6 Diagrammi di Voronoi

Nascono per ben altri scopi, e vengono studiati nel loro caso generale in n dimensioni dal matematico Georgy Voronoi [29], poi utilizzati nell'ambito di PCG in quanto la loro rappresentazione grafica ricorda particolarmente la struttura delle cellule, ed è quindi molto utilizzato in varie applicazioni come vedremo nei paragrafi seguenti.

2.3 Stato dell'arte

Dato che gli oggetti da generare sono molti e ognuno prevede una tecnica particolare, è necessario procedere con una distinzione in base all'obbiettivo della generazione. Distinguiamo quindi:

2.3.1 Terreni

Quest'area di applicazione è molto trattata, in letteratura troviamo svariate tecniche per generare varie tipologie di terreni.

Le tecniche più semplici per generare il terreno si basano su vari modelli di rumore, spesso utilizzando proprio *Perlin Noise* opportunamente ricombinato ed adattato, mediante alcune funzioni, a seconda del tipo di terreno desiderato. Una variante è basata sulla sintesi dello spettro piuttosto che del rumore direttamente [8].

Un apporto considerevole proviene da alcuni algoritmi che introducono l'erosione degli agenti atmosferici per cercare di imitare la conformazione di alcuni luoghi semplicemente agendo sull'intensità delle piogge, del vento, delle eruzioni vulcaniche e così via [18], nel tempo sono stati effettuati vari miglioramenti [9, 19, 4, 5].

Ulteriori modalità, invece, si basano sulla definizione e costruzione di alcuni frattali, in particolare sono stati ideati due algoritmi [10]:

- algoritmo di dislocamento nel punto medio (o midpoint displacement algorithm);
- algoritmo diamante-quadrato (o diamond-square algorithm).

È presente inoltre il lavoro proposto da Teong Ong [27] che sfrutta algoritmi genetici, in seguito M. Frade ne prese spunto per la sua tesi [7], che gli permise di applicare queste tecniche per modificare il terreno del videogioco *Chapas*.

Vi è anche un esempio di come realizzare un terreno mediante la combinazione di più diagrammi di Voronoi opportunamente perturbati da rumore [8, 599].

2.3.2 Texture

Innanzitutto è bene definire questa parola, una *texture* è un'immagine atta a rappresentare un colore o un materiale particolare, viene utilizzata per rivestire un oggetto virtuale, quindi bidimensionale o tridimensionale, per renderlo verosimile.

Nell'ambito di PCG si è cercato di fornire strumenti utili per costruire le più svariate texture combinando l'utilizzo di filtri, immagini preesistenti e rumori.

Ken Perlin approfondì la ricerca in questo campo insieme a David Ebert ed altri, che nel 2002 pubblicarono un libro che tratta di generazione procedurale sotto molti punti di vista ed in particolare la parte relativa alle texture [8, 179-202].

In letteratura esiste anche un tentativo di creare delle texture mediante algoritmi genetici: "Gentropy: evolving2D textures" [32].

2.3.3 Vegetazione

Paragrafo che meriterebbe un intero capitolo per essere descritto al meglio, infatti, in letteratura sono molti i tentativi di creazione della vegetazione in modo automatico.

Tra gli studi più interessanti è necessario citare il lavoro svolto da Jason Weber e Joseph Penn [31], che mostrano un modello molto esaustivo per descrivere i parametri che governano il mondo della generazione, in particolar modo, di alberi. Lavoro che ha portato a varie implementazioni tra le quali il software in Java *Arbaro* [34] ed il plugin per Blender *gen3* [36] il cui autore è passato allo sviluppo di *NGPlants* [42].

Il lavoro sopraccitato propone un approccio basato sullo studio della struttura di un albero, cioè sull'individuazione di alcuni parametri fondamentali per poi ricostruirla in modo procedurale partendo esclusivamente da quest'ultimi. In passato, invece, altri studiosi avevano tentato di creare alberi seguendo degli approcci differenti.

Il biologo Lindenmayer nel 1968 [15] ideò un modello matematico in grado di simulare il comportamento di alcune piante, questa tecnica prese poi il nome di *L-System* in omaggio al suo ideatore. Molte migliorie sono poi state apportate nel lavoro svolto assieme a Prusinkiewicz [23].

Un altro approccio, esposto da Oppenheimer, si basava sull'utilizzo di frattali per la generazione di piante ed alberi [20].

Molti sforzi sono stati dedicati anche alla creazione di foglie realistiche, come nel lavoro di Peyrat, Terraz, Merillou e Galin [22].

Studi decennali hanno quindi portato allo sviluppo di software in grado di generare molte tipologie di vegetazione fino alla realizzazione di intere foreste costituite da varie tipologie di piante, erba, cespugli, fiori e così via [24].

2.3.4 Edifici e città

Di grande interesse è anche la creazione di edifici, inizialmente era motivata ed alimentata dal campo dell'architettura, recentemente è stata apprezzata anche in altri settori, tra i quali proprio PCG [17].

Attualmente la maggior parte di questi algoritmi si basa esclusivamente sull'aspetto esteriore degli edifici senza preoccuparsi degli interni. Le tecniche migliori, sono inoltre in grado, non solo di creare alcune case ma intere città come nel lavoro svolto da Kelly G. e McCabe H. [11] nel quale vengono utilizzate e miscelate assieme varie tecniche tra le quali L-system, frattali e persino grammatiche geometriche.

Del tutto simile ma introducendo una generazione in tempo reale è la pubblicazione di Greuter, Parker, Stewart e Leach [12].

Per poter realizzare città complesse è bene realizzare una rete stradale all'altezza come è stato fatto da Sun, Yu, Baciu e Green [26] utilizzando principalmente il diagramma di Voronoi, lavoro utilizzato anche per mettere a punto il progetto di Kelly e McCabe appena citato.

2.3.5 Armi

Sono state ideate tecniche per generare le armi più disparate, non solo dal punto di vista grafico ma soprattutto dal punto di vista della loro funzionalità in riferimento a parametri quali: danno apportato, raggio di azione, precisione e così via. Per poter creare armi in modo procedurale spesso si è fatto ricorso a tecniche basate su algoritmi genetici come nel videogioco *Galactic Arm Race* [35].

Capitolo 3

Generazione contenuti nei videogiochi di guida

3.1 Videogiochi di guida

Un videogioco di guida è un videogioco in cui i giocatori gareggiano in una competizione mediante l'utilizzo di un veicolo, quindi rientrano in questa categoria sia mezzi terrestri che acquatici o aerei. La diffusione maggiore è relativa a quelli terrestri e in particolar modo alle automobili e alle moto, ma esistono una grande varietà di titoli che permettono la guida di motoscafi, navi, aerei, elicotteri, dei quali, però, non ci occuperemo. Il nostro interesse principale è legato al mondo delle automobili e ai videogiochi che vedono queste ultime come protagoniste.

I primi videogiochi di guida nascono negli anni '70 con quello che è considerato il primo titolo del genere cioè *Speed Race* prodotto dalla *Taito*, in quegli anni i videogiochi erano molto semplice, basati su una grafica molto basilare e poco dettagliata, anche la giocabilità non era delle migliori.

Con il passare degli anni, e con la possibilità di sviluppare giochi migliori da tutti i punti di vista incominciarono a formarsi due filoni, da una parte i giochi creati esclusivamente per affascinare e divertire, e dall'altra videogiochi in grado di simulare il più possibile il comportamento reale di un autovettura.

Questa è la differenza tra i titoli detti *arcade* e quelli definiti *simulatori di guida*. Attualmente non esiste una netta distinzione tra una o l'altra categoria in quanto per alcuni aspetti un titolo è prettamente di simulazione ma per altri potrebbe introdurre nuove caratteristiche spettacolari che lo rendono più entusiasmante da giocare ma non più realistico.

Occorre eseguire un'ulteriore distinzione per comprendere al meglio le tecniche di PCG applicate, dobbiamo infatti considerare la composizione di un livello, cioè del tracciato giocabile. Quindi individuiamo due zone principali, quelle che fanno parte del tracciato e quindi influenzano la giocabilità oltre che l'estetica del gioco stesso, stando alla definizione data nel capitolo 2.1 sono la parte di contenuto necessaria. Le altre zone, invece, sono esclusivamente di carattere estetico quindi facoltative, la loro presenza, però, è molto importante ai fini della bellezza di un gioco, rappresentano quindi un punto cruciale nella fase di sviluppo.

3.1.1 TORCS

Il videogioco di guida TORCS fa parte dei titoli di simulazione pura in cui si guidano automobili, appartiene a questa categoria in quanto implementa rigorose regole fisiche per aderire il più possibile alla realtà. Da questo punto di vista è molto ben fatto e compete direttamente con software commerciali, da quest'ultimi perde sotto il profilo della qualità grafica, soprattutto per il fatto che il gioco è abbastanza datato, infatti nacque sotto l'impulso dei due programmatori Eric Espié e Christophe Guionneau nel 1997 con una prima versione 2D che poi è diventata nel tempo l'attuale viodeogioco giunto alla versione 1.3.1, risalente al 2007.

Fornisce, inoltre, un buon supporto alla programmazione di robot autonomi permettendo di agire in modo abbastanza preciso su quasi tutti i comandi presenti in una automobile reale, provocando un effetto degno di un videogioco di simulazione.

Dobbiamo inoltre sottolineare il fatto che è disponibile per più piattaforme, quali Windows, MacOSX, FreeBSD e Linux, e soprattutto che è rilasciato

sotto licenza *GPL* il che ci permette di poter comprendere direttamente la sua struttura.

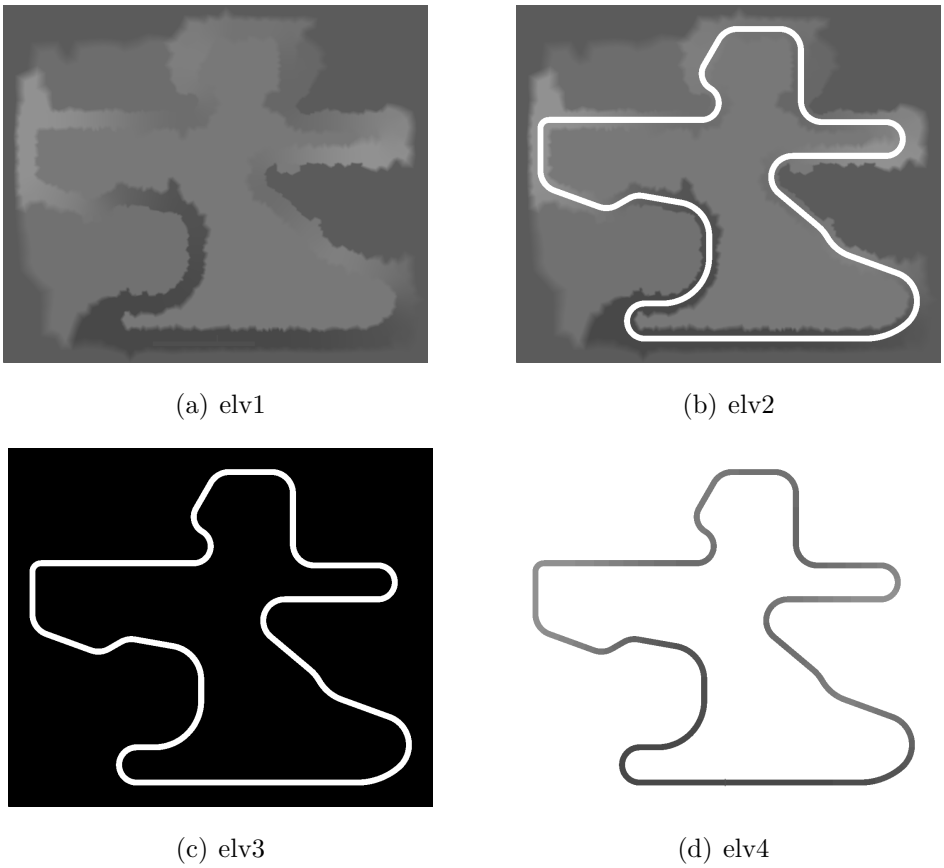
Trackgen

Per poter capire le funzioni implementate è bene descrivere le funzionalità offerte da TORCS e in modo particolare dallo strumento *trackegen*. Questo programma ci permette da un lato di ottenere alcune immagini importanti, e da un altro di procurarci il tracciato finale. Grazie a questo prezioso strumento, inoltre, possiamo gestire molte fasi della generazione a livello bidimensionale invece di trattare modelli tridimensionali, con una conseguente riduzione della complessità dell'approccio utilizzato ma con un incremento dei vincoli da rispettare e quindi con una minore libertà di azione, mostreremmo all'occorrenza le fasi in cui saremo affetti da queste limitazioni.

Tra le immagini più importanti ci sono quelle relative all'elevazione che sono tutte in scala di grigi nel formato *png*. In generale il colore bianco rappresenta la quota più alta, mentre il nero quella più bassa, qui di seguito vi è una descrizione per ciascuna di esse:

- elv1: è la mappa elevazione impostata di default, fig. 3.1(a);
- elv2: simile alla precedente ma con evidenziato il tracciato in bianco, in questo caso il bianco rappresenta anche il tracciato oltre che i punti ad altezza massima, fig. 3.1(b);
- elv3: mostra esclusivamente il tracciato in bianco e il resto dell'ambiente colorato di nero, il significato della colorazione riferita alla quota perde totalmente di significato, fig. 3.1(c);
- elv4: in cui si indica l'elevazione solamente del tracciato, al bianco associamo sia il valore di quota massima sia di semplice sfondo, fig. 3.1(d).

Questa doppia informazione portata dal colore bianco ci porta a dover effettuare delle ipotesi restrittive come meglio descritto nel paragrafo 4.10.3.



(a) elv1

(b) elv2

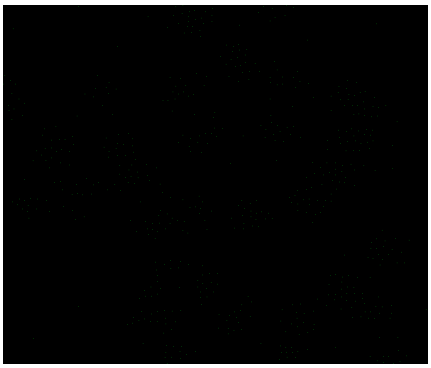
(c) elv3

(d) elv4

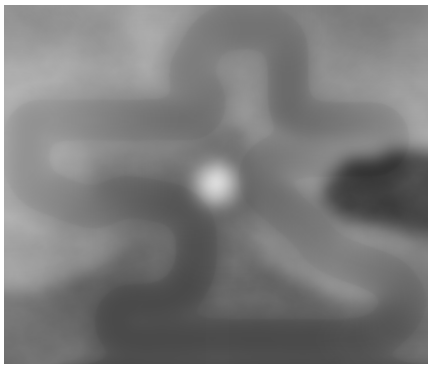
Figura 3.1: Immagini create da trackgen

Per la generazione del risultato finale, invece, possono essere utilizzate ulteriori due immagini:

- mappa degli oggetti: rappresentata da un'immagine a sfondo nero, in cui ad ogni pixel è associata una posizione nell'ambientazione, ed una colorazione di quel pixel diversa dal nero indica la presenza di un oggetto in quel punto. Ogni oggetto, infatti, possiede un ID che coincide con la codifica esadecimale del colore che lo rappresenta, quindi è possibile inserire un oggetto semplicemente colorando opportunamente un pixel 3.2(a);
- mappa elevazione: è un immagine del tutto simile a quelle definite all'inizio del paragrafo, quindi in scala di grigi in cui ad ogni tonalità di grigio è associata una quota differente, con l'unica eccezione che in questo caso si tratta di un immagine presa in input a cui il programma si attiene per generare l'elevazione finale 3.2(b).



(a) mappa oggetti



(b) mappa elevazione

3.2 Modifiche apportate a TORCS

Lo scopo principale di questo lavoro è quello di generare in modo automatico l'ambientazione del videogioco TORCS. Ambientazione è però un termine troppo generico, può infatti riferirsi a molti elementi ad un diverso livello di dettaglio, sicuramente utilizziamo un approccio macroscopico del problema,

cioè non siamo interessati a far sì che venga generato qualsiasi dettaglio ma solo un assetto generale. Per esempio non costruiamo un terreno erboso filo d'erba per filo d'erba ma utilizzando tecniche alternative atte a ridurre sia la complessità nel rappresentare il mondo, sia nel crearlo.

Possiamo inoltre affermare che le metodologie applicate sono specifiche dell'architettura di TORCS, ma la risoluzione concettuale di alcune problematiche affrontate è valida molto più in generale e di facile estensione a realtà differenti.

Le tecniche usate sono stazionarie, cioè non variano nel tempo e non si basano su nessun algoritmo genetico o di apprendimento automatico che ottimizzerebbero il risultato. L'obbiettivo è semplicemente quello di creare un'ambientazione realistica seguendo alcune regole euristiche determinate a priori ed eventualmente rappresentare la base per lavori futuri inerenti alle tecniche sopra citate.

Dobbiamo anche tener conto che il lavoro si basa sul videogioco TORCS, quindi spesso ci troviamo a dover limitare alcune scelte in base ai vincoli imposti dalla struttura del programma preesistente, ciò riduce la libertà e ci obbliga ad affrontare i vari problemi con le approssimazioni del caso che quindi ci allontanano dalla soluzione ideale (non da quella ottima in quanto non applichiamo nessuna ottimizzazione).

Gli aspetti sui quali ci concentriamo maggiormente non sono tutti quelli già citati ne "Lo stato dell'arte" (paragrafo 2.3) ma una loro piccola parte, anche se tutto ciò che non viene esplicitamente implementato mantiene comunque caratteristiche tipiche delle tecniche procedurali dato che molti componenti grafici sono stati creati mediante tali metodologie.

Diamo quindi maggior interesse alla generazione del terreno, al posizionamento degli oggetti ed alla modifiche delle superfici come mostrato nei prossimi paragrafi.

3.2.1 Terreno

Uno dei problemi da affrontare è quello relativo alla coerenza con il tracciato, cioè far sì che l'altitudine del terreno generato in prossimità della pista, non si

discosti di molto dalla quota relativa a quest'ultima. I primi tentativi di coerenza erano basati sulla scelta di alcuni parametri mediante delle assunzioni buone per alcune piste ma pessime per altre morfologie, quindi il principale difetto di questi metodi era che non si basavano sui valori esistenti in modo uniforme ma prendevano come riferimento dei valori di massima scelti a priori, nella fattispecie i valori di altitudine massima e minima del tracciato stesso. L'approccio attuale, che ha risolto definitivamente i problemi di coerenza, si basa sull'elevazione preesistente ed effettua una sorta di espansione dell'altitudine del tracciato verso lo spazio con altezza incognita.

Un altro problema è quello di variare l'altitudine per creare rilievi più o meno accentuati in base al tipo di ambientazione e soprattutto in modo pseudo-casuale.

Dobbiamo anche affrontare un effetto indesiderato che si è presentato applicando una prima bozza di algoritmo, cioè l'effetto che battezziamo con il nome di *effetto canyon*, nome abbastanza autoesplicativo, ma a scanso di equivoci, esso indica una situazione in cui sia a sinistra che a destra del tracciato è presente una salita molto pendente che limita la visuale al solo tracciato, impedendo cioè di scrutare l'orizzonte e regalando una sensazione visiva simile a quella che si avrebbe all'interno di un canyon.

3.2.2 Oggetti

Riguardo agli oggetti, intesi come oggetti tridimensionali da aggiungere all'ambientazione, il nostro obbiettivo è semplicemente quello di posizionarli nel modo migliore possibile, con ciò non vogliamo intendere che gli oggetti verranno disposti sempre nel luogo più adatto dato che sarebbe impossibile, non esiste infatti una regola razionale che possa determinarlo. La collocazione è solamente verosimile e tenta di seguire alcune regole specifiche create appositamente per aderire al senso estetico del programmatore.

Un aspetto non trattato è quindi la generazione automatica degli stessi, compito che è di per sè molto complesso e oneroso, quindi ci limitiamo ad

utilizzare, ove possibile, oggetti creati con tecniche procedurali mostrate ne "Lo stato dell'arte" e non ad implementarle direttamente.

Un altro problema che sorge utilizzando direttamente gli strumenti forniti da TORCS è l'inserimento di oggetti di modeste dimensioni in zone molto pendenti del terreno.

Altra questione da superare è la possibile intersezione tra gli oggetti, e soprattutto tra gli oggetti e la pista, quest'ultima evenienza renderebbe infatti il tracciato inutilizzabile.

3.2.3 Superfici

Un punto di particolare interesse è la modifica di tutte le superfici che fanno parte dell'ambientazione, quindi sia le parti del tracciato dall'asfalto ai cordoli, sia il terreno esterno alla pista.

Il vero punto cruciale di questa sezione è però capire perfettamente il significato dei tag presenti nel file xml e la conseguente implementazione delle operazioni di lettura e scrittura opportune, in modo da mantenere la validità del documento.

Capitolo 4

Implementazione del framework

4.1 Librerie esterne

Vediamo ora in dettaglio tutte le librerie utilizzate.

- *Java Image Filters*: è una libreria che fornisce una grande quantità di filtri ed effetti da applicare ad immagini, in particolar modo siamo interessati all'effetto di sfocatura gaussiano che è di ricorrente utilizzo nella tesi. Per ulteriori dettagli e guide di utilizzo rimandiamo al sito dell'autore *Jerry Huxtable* [39];

- *JTexJen*: di queste librerie faremo uso esclusivamente della parte relativa alla generazione di rumore pseudo-casuale, nello specifico ad una applicazione dell'algoritmo ideato da Ken Perlin. È distribuito sollo licenza *LGPL* ed ulteriori informazioni possono essere trovate sul sito del progetto [40];

- *VecMath*: sono librerie interne al progetto *Java 3D* e quindi sviluppate da Oracle, utilizzate in questo tesi solo per la nozione di punto tridimensionale non presente nelle librerie standard di Java. Maggiori informazioni sul sito presente in bibliografia [37];

- *JDOM*: utilizzate per la modifica dei vari file xml, preferite alle librerie native di Java poiché ritenute più potenti e di più immediato utilizzo. Altre informazioni, compresa la licenza e il codice sorgente, possono essere trovate direttamente sul sito degli sviluppatori di *JDOM Project* [38].

4.2 Percorsi e cartelle

Prima di eseguire qualsiasi operazione dobbiamo ottenere alcune informazioni utili, tra cui: le cartelle relative all'installazione di TORCS, il tipo di generazione e ambientazione che desideriamo creare, il percorso in cui sono salvati i contenuti grafici e infine il nome del tracciato e il tipo di quest'ultimo.

Per far ciò utilizziamo un file di testo la cui struttura è spiegata più dettagliatamente nell'appendice C.2.2.

4.3 Tipi di generazione

Sono previste 4 tipologie, ognuna di queste differisce dal modo in cui viene implementata la generazione, in pratica i metodi delle classi, quindi possiamo aggiungere altre categorie solamente modificando il codice.

- city: adatta ai terreni pianeggianti, che prevedono un grande numero di oggetti;
- desert: adeguata a terreni collinari, con pochi oggetti;
- hill: opportuno per terreni collinari tendenzialmente montuosi, con un discreto numero di oggetti;
- mountain: idoneo a morfologie montuose con pochissimi oggetti.

4.4 Ambientazioni

Di seguito ci sono alcune ambientazioni disponibili, che si basano su un insieme di contenuti grafici predefiniti ed alcuni parametri ottenuti sperimentalmente

che ben si prestano al paesaggio da creare. È possibile aggiungerne altre senza modificare il codice a patto di rispettare alcune convenzioni, cioè bisogna rispettare rigidamente l'organizzazione in cartelle come meglio specificato nel paragrafo 4.5, oltre a rispettare tutti i vincoli sui formati dei files definiti nel manuale utente C.3.

Attualmente sono previste le seguenti ambientazioni:
city-1, desert, desert-orange, hill, lava, mountain, mountain-snow, polimi, simpson.

4.5 Suddivisione dei contenuti

Per facilitare la comprensione e per non appesantire troppo i nomi dei file decidiamo di suddividere i vari contenuti grafici seguendo una struttura ad albero, quindi risulta comodo sfruttare la gerarchia a cartelle dei più comuni sistemi operativi.

Sono previste altre cartelle per sviluppi futuri ma attualmente sono utilizzate solo le seguenti, individuabili in quattro categorie principali:

- background: contiene i possibili sfondi dello scenario;
- env: dove inserire l'immagine utilizzata dal motore grafico per simulare i riflessi dei materiali lucidi, è un immagine che mostra i tratti tipici dell'ambiente in cui si trova;
- surface: in cui sono presenti tutte le texture relative alle superfici da modificare, è suddiviso in:
  - asphalt: vari tipi di superfici della carreggiata, per esempio asfalto, terra, sabbia e così via;
  - barrier: le barriere laterali che dividono il tracciato dal resto dell'ambiente, possono essere muretti, staccionate, guardrail ...;
  - border: indica lo spazio tra la carreggiata e la zona esterna;

  - side: la zona più esterna calpestabile, dopo di questa c'è solo una barriera;
  - terrain: tutto il resto dell'ambiente.

- object: in questa cartella sono inseriti tutti gli oggetti da aggiungere, comprese le texture a questi associate, è suddiviso in:
  - arch: archi sospesi sopra la pista;
  - billboard: cartelloni pubblicitari tipicamente posizionati ai lati;
  - building: tutti i vari tipi di edifici;
  - vegetation: qualsiasi oggetti relativo alla vegetazione: alberi, cespugli, fiori ...;
  - vehicle: veicoli in generale, può contenere: automobili, camion, camper, elicotteri ...;

È presente inoltre il file *pattern.xml* che permette di definire alcuni parametri tipici dell'ambientazione in cui è contenuto, per una spiegazione dettagliata e per la struttura del file si veda l'appendice C.2.3.

4.6 Modifica superfici

In accordo a quanto già definito all'interno di TORCS suddividiamo le superfici in categorie, le stesse descritte nell'elenco puntato "surfaces" presente nel paragrafo 4.5. Questa ripartizione ci permette di assegnare ad ogni superficie una sua categoria precisa così da poter modificare la zona interessata, ciò avviene semplicemente andando a scrivere sul file xml del tracciato, la cui struttura è mostrata dettagliatamente nell'appendice C.1.

Se di ogni categoria esistono più contenuti il programma ne sceglie uno a caso ed utilizza sempre quello per le varie modifiche che lo richiedono. Una scelta differente dovrebbe necessariamente sfruttare informazioni aggiuntive, per esempio assegnare una certa priorità ai vari contenuti in modo manuale oppure automaticamente per esempio scegliendo quelli con color medio simile;

ma per semplicità non imbocchiamo questa strada che rimane uno dei problemi in sospeso.

Ogni contenuto grafico relativo a superfici va aggiunto nella sezione *Surface* impostando alcuni attributi soprattutto *texture name* che indica il nome dell'immagine. Altri attributi, invece, sono quelli relativi ai parametri dell'asfalto e sono scelti utilizzando le informazioni presenti nel file *pattern.xml* (un esempio di questo file relativo ad una generazione cittadina è disponibile nell'appendice C.2.3).

La variazione effettiva delle superfici nel tracciato consiste solamente nel modificare la corretta sezione dell'xml del tracciato, in particolare l'attributo *surface* presente nella sezione *Main Track* (vedi appendice C.1.5).

Fa eccezione la superficie relativa al terreno che invece viene modificata nella sezione *Terrain Generation*, sempre con attributo *surface* come esposto nell'appendice C.1.4.

4.7 Ulteriori modifiche all'xml del tracciato

Una corretta generazione prevede che vengano effettuate ulteriori modifiche al file xml, tra le quali è necessario indicare il file relativo alla mappa degli oggetti, quello previsto per l'elevazione ed altri parametri importanti come l'altezza minima e massima del terreno. La scelta di questi ultimi due valori è spiegata più avanti nel paragrafo 4.10.1.

4.8 Fattore di scala

Per ottenere dei risultati validi ad ogni tracciato, dobbiamo anche fare attenzione al problema causato dalla scelta degli sviluppatori di TORCS di generare le varie immagini riferite al tracciato con una dimensione fissa, cioè tutte hanno larghezza pari a 1024 pixel mentre l'altezza è variabile, mantenendo però la proporzionalità . Ciò implica che la dimensione di un pixel in unità di misura a noi più congeniali non è sempre la stessa, siamo quindi obbligati a considerare

un fattore di scala, usiamo cioè un valore che indica a quanti metri corrisponde un pixel (*meterPerPixel*) facendo il rapporto tra la larghezza reale in metri (*dimX*) e 1024:

$$meterPerPixel = \frac{dimX}{1024}$$

Individuiamo quindi due strati, uno più alto che lavora con soli valori in metri, tipicamente anche l'utente, mentre uno più basso che ragiona su valori espressi in pixel, in quest'ultimo caso troviamo essenzialmente tutti gli algoritmi che manipolano le immagini. Passando da uno strato all'altro è ovviamente necessaria una conversione, questo compito viene fatto all'interno del livello basso quindi l'utilizzatore finale non dovrà mai preoccuparsi di questo problema che è tutto sulle spalle del programmatore.

4.9 Manipolazione immagini

A causa di un largo utilizzo di immagini adottiamo delle classi specifiche per la manipolazione di queste ultime, di seguito sono mostrate alcune tra le più importanti con le funzioni svolte.

4.9.1 Classe Elv3

Utilizziamo una classe per implementare tutti quei metodi strettamente legati all'immagine omonima, quindi questa classe fornisce alcuni metodi importanti, tra i quali:

- getBounds(boolean internal): restituisce tutti i punti posti sul confine del tracciato ordinati seguendo la direzione dello stesso, mediante l'unico parametro d'ingresso scegliamo se considerare il bordo interno o esterno (vedi appendice B.2.2);
- getCouples(int spacing): restituisce le coppie di punti che fanno parte del bordo pista e che tra di loro hanno distanza minima rispetto a tutte le altre possibili combinazioni (vedi appendice B.2.2);

4.9.2 Classe Elv4

In più rispetto alla classe precedente prevede anche il valore di altitudine di ogni punto in analogia da quanto previsto da trackgen, quindi è possibile ottenere informazioni riguardo a questo aspetto come nel caso di:

- getNearestColor(IntPoint point): partendo dal punto preso come parametro *point* restituisce il punto della pista più vicino e il suo valore di altitudine salvato come colore (vedi appendice B.2.3).

4.10 Generazione terreno

Riferendoci alle varie tecniche mostrate ne "Lo stato dell'arte" (Capitolo 2.3), utilizziamo quelle che sfruttano il rumore, nello specifico il *Perlin Noise*.

Per ottenere una morfologia del terreno casuale, e adatta al nostro scopo, creiamo una mappa elevazione, cioè un'immagine in cui il colore di un pixel rappresenta l'altitudine di quel punto.

Matematicamente abbiamo una funzione in due variabili che chiamiamo $h(x, y) : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}$ il cui grafico rappresenta l'elevazione del terreno che è appunto rappresentabile in uno spazio tridimensionale $z = h(x, y)$, dove z_0 è la quota relativa al punto $P_0(x_0, y_0)$. Per poter memorizzare i valori di una funzione all'interno di un computer è conveniente utilizzare una struttura dati tipo matrice, nel nostro caso utilizziamo un'immagine in scala di grigi poiché è necessaria per sfruttare alcuni strumenti già forniti da TORCS stesso. Questa scelta ci permette di sfruttare al massimo 256 valori, quindi accettiamo una perdita di precisione causata dall'approssimazione, dobbiamo infatti passare da una funzione, in generale, con dominio definito su $\mathbb{R}^2$ a una rappresentazione discreta cioè con dominio incluso in $\mathbb{N}^2$, o meglio, dato che vi è un limite intrinseco in TORCS, del tipo $(0, 1024) \times (0, n)$ con $n \in \mathbb{N}$.

Inoltre lavorando con immagini in scala di grigi il codominio diventa un sottoinsieme di $\mathbb{N}$, cioè $(0, 255) \subset \mathbb{N}$.

La struttura dati utilizzata, inoltre, ci impone di generare solo alcune tipologie di terreni, cioè tutti e soli quelli che possono essere grafici di una funzione, quindi si devono escludere caverne, tunnel e così via.

Per i nostri scopi è un vincolo ragionevolmente accettabile ed è del tutto necessario se vogliamo utilizzare il codice già scritto in TORCS senza dover creare direttamente la struttura tridimensionale del tracciato.

É prevista una generazione specifica del terreno in base alla tipologia scelta, quindi per esempio un paesaggio collinare avrà insenature lievi e poco frastagliate, viceversa un terreno montagnoso sarà più scolpito. Per fare ciò abbiamo sfruttato il principio dell'ereditarietà, in cui una classe astratta implementa un'interfaccia(*Elevating*) che possiede tutti i metodi necessari per la modifica del terreno. Questa classe astratta(*ElevationMap*) a sua volta aggiunge dei metodi generali che saranno utili a tutte le sue sottoclassi ed effettua un *override* dei metodi dell'interfaccia implementando una versione di default comune a tutti. Le classi specifiche che estendono quest'ultima possono reimplementare i metodi ereditati così da rendere specifica la generazione di un dato paesaggio dal punto di vista dell'elevazione. Questo particolare diagramma è mostrato in appendice A.1.

4.10.1 Scelta intervallo altezze

Per creare correttamente il terreno, prima di tutto, è necessario scegliere degli opportuni valori di altitudine massima e minima basandoci su quelle del tracciato, cioè andando a valutare l'altezza di ogni segmento, e se non specificata ricavandola grazie alla pendenza ed alla lunghezza del tratto che ci permettono di ottenere il dislivello e quindi anche la variazione locale.

Una volta determinate l'intervallo di altezza effettivo procediamo ad una sua modifica per permetterci di avere una morfologia più realistica, l'intensità della variazione, definita nel file "pattern.xml", dipende dal tipo di ambientazione desiderata. In ogni caso la variazione è sempre positiva per quanto riguarda l'altezza massima e negativa per quanto riguarda quella minima.

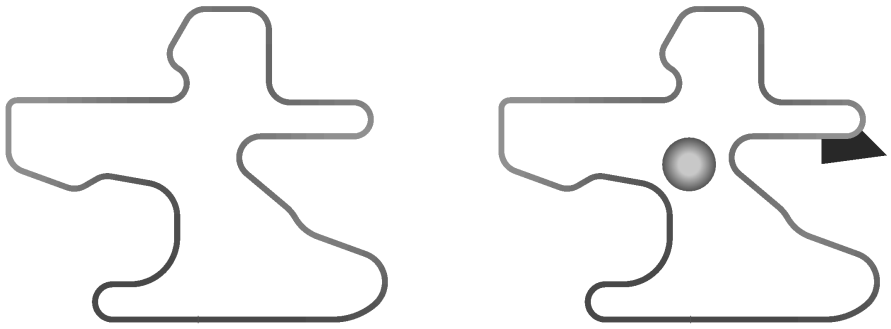
Inoltre non può essere pari a zero per evitare il problema della mancanza di informazioni già citato e che sarà descritto nel paragrafo 4.10.3.

4.10.2 Modifiche terreno a priori

Partendo dalla mappa elevazione del solo tracciato (fig. 4.2(a)) aggiungiamo delle piccole variazioni al terreno secondo alcune regole definite in base al tipo di generazione, possiamo distinguere queste variazioni in due categorie: burroni e montagne a seconda che diminuiscono o aumentano l'altezza.

Nella figura 4.1 mostriamo la mappa elevazione e le modifiche apportate nella fase preliminare con l'aggiunta di un burrone (in nero) e una montagna al centro (colorata con un gradiente radiale in scala di grigi).

Figura 4.1: Mappa elevazione:



(a) solamente del tracciato (b) con le modifiche preliminari

Burroni

Per esempio, inseriamo un burrone colorando un poligono di grigio scuro i cui vertici sono un insieme di bordi pista successivi uniti da due punti perpendicolari al tracciato così da formare una figura simile ad un rettangolo.

Montagne

La variazione positiva dell'altitudine è invece demandata ad un ulteriore algoritmo che, dato un punto e un altezza desiderata, genera una piccola insenatura disegnando un cerchio riempito con un gradiente di scala di grigi in direzione radiale, cioè un cono nella versione tridimensionale.

Scegliamo in base al tipo di generazione i parametri richiesti, per esempio in un paesaggio collinare vengono inseriti dei rilievi nei quattro angoli della mappa, oppure nella generazione di una morfologia montagnosa immettiamo un picco al centro del tracciato.

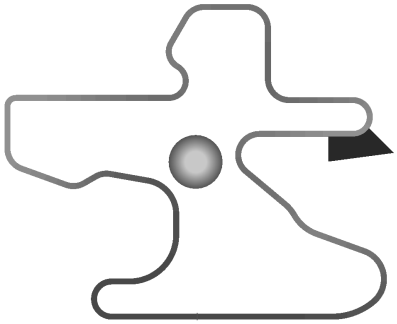
4.10.3 Espansione

Per ottenere una mappa di elevazione di partenza dobbiamo basarci sulle informazioni contenute nell'immagine appena elaborata cioè sull'elevazione del solo tracciato più le informazioni di burroni e montagne. Procediamo quindi ad una sorta di espansione dell'immagine esistente come mostrato nelle figure 4.2

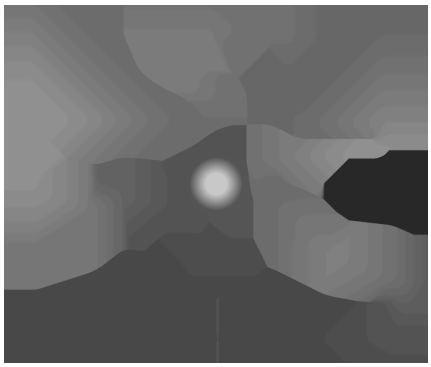
Applichiamo un algoritmo che va a colorare le zone bianche basandosi sulla media dei colori adiacenti (dove per colori adiacenti intendiamo tutti gli otto pixel che circondano quello centrale); ovviamente per poter riempire tutta l'immagine occorre eseguire molte iterazioni, l'algoritmo termina quando non vi sono più pixel bianchi. Nelle figure 4.3 è presente un esempio di espansione di una piccola porzione di 9 pixel nei quali è scritto il valore riferito al colore per un immagine in scala di grigi.

Non è possibile effettuare le iterazioni sempre sulla stessa matrice quindi è necessario lavorare su una copia per evitare di influenzare il colore dei pixel ancora da esaminare.

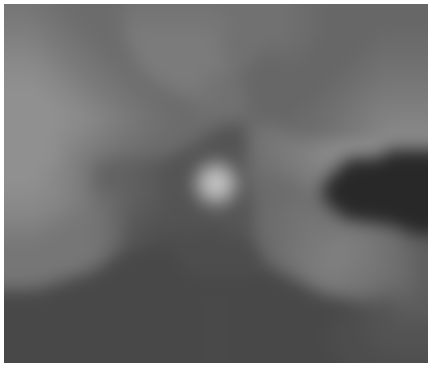
Per maggiore chiarezza su come è costruito l'algoritmo si faccia riferimento al codice presente in appendice B.4.



(c) elv4-mod

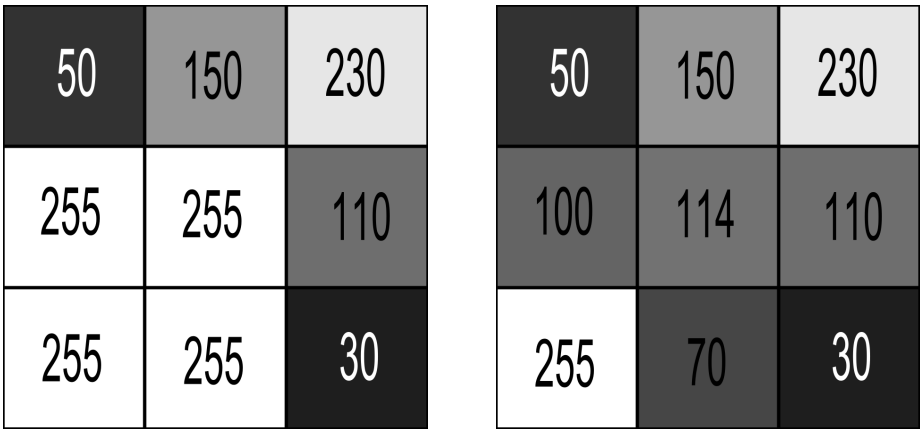


(d) Base



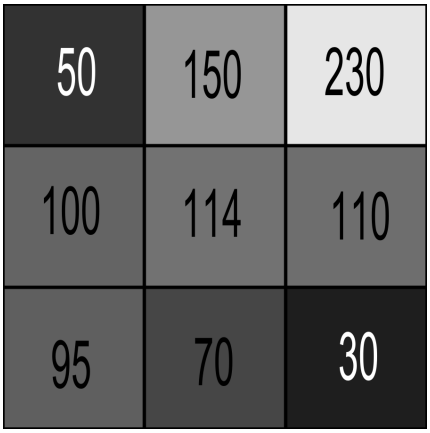
(e) Base sfocata

Figura 4.2: Espansione



(a) situazione iniziale

(b) primo passo



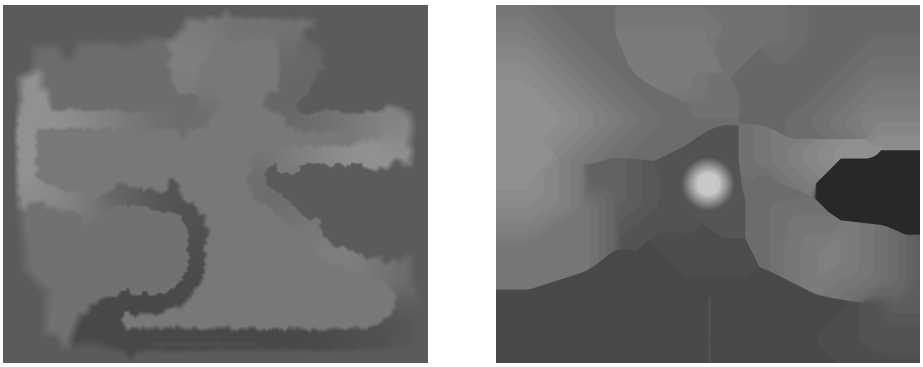
(c) secondo passo, fine

Figura 4.3: Espansione, zoom nove pixel

Confronto approcci

Un vecchio approccio consisteva nell'utilizzare direttamente l'immagine elv1 fornita da trackgen, si è poi rivelata una strada scorretta in quanto questa immagine non considera la variazione dell'altitudine del tracciato ma crea delle zone isoipse molto grandi e poco fedeli al reale profilo altimetrico.

Nella figura 4.4 mostriamo la differenza dei due approcci, (a) indica la strada abbandonata, mentre (b) è l'immagine utilizzata.



(a) Elv1 (b) Base

Figura 4.4: Confronto approcci

Mancanza di informazioni

Trackgen genera il file elv4 assumendo che è il bianco il colore destinato a rappresentare una zona che non fa parte del tracciato, ciò può generare dei problemi quando un punto del tracciato ha elevazione tale da meritarsi la massima quota che è rappresentata dal colore bianco.

Quindi vi è un conflitto: si usa lo stesso colore per fornire due informazioni diverse. Nel caso in cui ciò avvenisse l'algoritmo di espansione produrrebbe un risultato errato tanto più sono i pixel affetti da questo problema, cioè quei pixel che sono parte del tracciato ma hanno un elevazione massima.

Nei casi concreti ciò avviene raramente e soprattutto può essere evitato imponendo un limite superiore di altezza che deve essere maggiore all'altezza

massima del tracciato di qualche unità. Ciò può essere fatto mediante la scelta di opportuni valori di altezza minima e massima come mostrato nel paragrafo 4.10.1.

4.10.4 Mappa distanza

Per *mappa della distanza* indichiamo un immagine in scala di grigi che mostra uniformemente, cioè per ogni punto, la distanza dal tracciato. In realtà l'immagine serve solo per avere un riferimento grafico molto più significativo rispetto a una matrice bidimensionale di valori numerici.

Utilizziamo una convenzione che impone una precisione massima di $\frac{1}{256}$, dove $256 = 2^8$ sono i valori disponibili nella rappresentazione standard di un immagine in scala di grigi. Per convenzione associamo al colore bianco (valore 255) la distanza massima ed al colore nero (valore 0) quella minima.

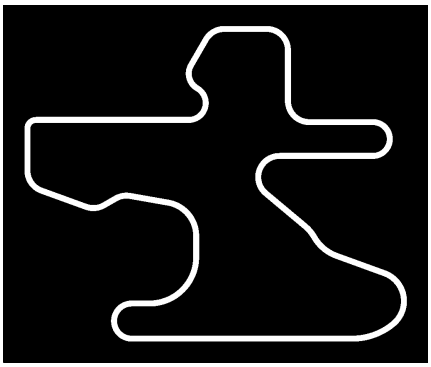
Come abbiamo detto per creare questa mappa dobbiamo calcolare per ogni punto la sua distanza dal tracciato, dove per distanza indichiamo quella minima, resta quindi il problema di come calcolare questa grandezza.

Prima versione

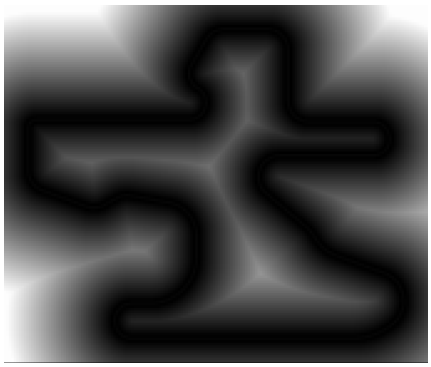
Possiamo trovare una soluzione analizzando ogni punto, e per ciascuno di questi allontanarci in tutte le direzioni fino a trovare un bordo pista, quando lo troviamo otteniamo direttamente la distanza minima che ovviamente coincide con il numero di iterazioni necessarie per raggiungere questa condizione.

Questo algoritmo però non è molto efficiente, infatti se consideriamo un immagine quadrata di dimensione x nel caso pessimo dobbiamo effettuare un numero di controlli pari a $O(x^2)$, in quanto non potremmo effettuare più controlli di quanti necessari ad esaurire l'immagine e nella peggiore delle ipotesi dobbiamo controllarli tutti.

Assumiamo poi che il calcolo della distanza tra due punti abbia complessità costante nel tempo, troviamo quindi, iterando su ogni pixel, una complessità asintotica totale di $O(x^4)$, come si può vedere nel codice in appendice B.3.1.



(a) elv3



(b) mappa distanza

(c) mappa distanza sfocata

Figura 4.5: Costruzione mappa distanza

Versione migliorata: analisi dei bordi

L'algoritmo è ancora troppo lento, cerchiamo quindi una soluzione alternativa considerando il fatto che siamo a conoscenza di tutti i punti che sono bordo pista.

Uniamo quindi i due bordi, uno interno e l'altro esterno, rappresentandoli come una lista di punti, e procediamo a valutare la distanza di ogni punto dell'immagine con ogni punto presente in questa lista.

Supponendo che la lunghezza del tracciato, ovvero il numero di pixel del bordo, sia lineare con la dimensione x definita nel paragrafo precedente, possiamo concludere che la complessità asintotica in questa versione vale $O(x^3)$, che è inferiore di quella prevista per l'algoritmo precedente.

La nuova versione è presente in appendice B.3.2.

Approssimazione con intorni quadrati

L'algoritmo del paragrafo precedente rimane valido, ma la grande dimensione delle immagini richiede un tempo ancora troppo elevato di esecuzione, quindi procediamo con un metodo che approssima l'algoritmo precedente.

Andiamo a considerarne uno che, invece di analizzare ogni pixel nell'immagine, valuti la distanza tra un sottoinsieme di punti ed ogni bordo pista e ne ottenga la distanza minima.

Dobbiamo inoltre associare ad ogni insieme di punti un intorno quadrato a cui riferirci ed assegnare la stessa distanza in blocco. Il risultato fin qui ottenuto risente dell'approssimazione effettuata, graficamente appare come un insieme di quadratini ($n \times n$, con, nel nostro caso, $n = 6$) visibili ad occhio nudo, di fatto è semplicemente cambiata la risoluzione.

Ovviamo a questo inconveniente sfocando l'immagine utilizzando il filtro *Gaussian Blur*, presente nella librerie esterne 4.1.

Questo algoritmo è mostrato nel listato B.3.3.

4.10.5 Perlin Noise

Vediamo ora la tecnica utilizzata per aggiungere degli elementi di casualità alla generazione dell'elevazione, sfruttiamo cioè il *Perlin Noise* e in particolare l'implementazione delle librerie JTexGen 4.1 la quale permette di definire alcuni parametri per poterne controllare in qualche misura il risultato.

Grazie alla possibilità di scegliere dei parametri risulta abbastanza semplice differenziare il rumore in base all'ambientazione scelta, infatti questi valori vengono prelevati direttamente dal file *pattern.xml* C.2.3.

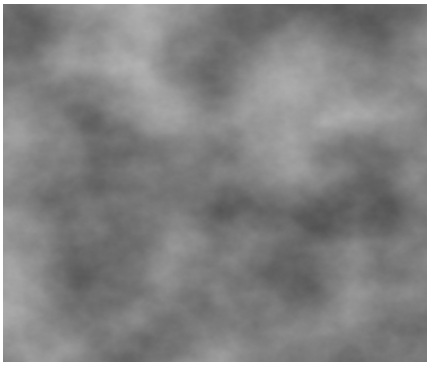


Figura 4.6: Rumore Perlin (Perlin Noise)

4.10.6 Elevazione finale

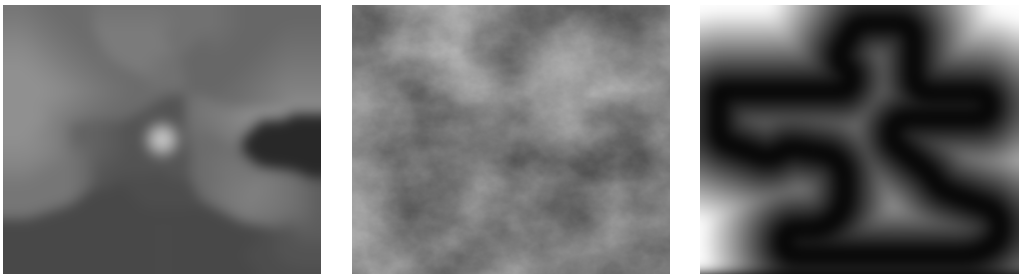
Abbiamo tutti gli elementi e le informazioni necessarie per generare il file di elevazione finale. Come accennato prima il rumore serve a introdurre un elemento aleatorio e quindi ci permette di generare ogni volta un terreno differente, mentre, la mappa distanza permette di applicare il rumore con diversa intensità in funzione della distanza dal tracciato, questo per evitare l'effetto indesiderato detto *effetto canyon*. L'applicazione del rumore si basa sulla seguente formula ottenuta sperimentalmente:

$$f(x) = \begin{cases} 1 & x \geq \frac{1}{3} \\ \frac{1}{2}\left[(1-6x)^{\frac{37}{61}} + 1\right] & x < \frac{1}{3} \end{cases}$$

In cui x è la distanza dal tracciato normalizzata sull'intervallo $(0, 1)$ ed $f(x)$ rappresenta l'intensità con cui applicare il rumore.

Il rumore si applica partendo dall'immagine di base 4.8(a), in cui ogni pixel subisce una modifica di colorazione verso quote superiori o inferiori a seconda dell'immagine del rumore 4.8(b) e della distanza dal tracciato 4.8(c) nel pixel esaminato. La variazione massima inferiore e superiore è definita dai parametri (indicati nel file *pattern.xml*) *startRange* e *endRange* rispettivamente. Quindi per esempio se scegliamo questi due parametri a $(-20, +30)$ e ci troviamo in un pixel con valore 127, che supponiamo per semplicità essere molto distante per non essere influenzati dal fattore distanza, e con un rumore di valore 200 il risultato finale sarà di 146, poiché normalizzando il rumore sull'insieme $(-20, +30)$ otteniamo un valore di circa 19 che sommato al valore della base ci porta al risultato.

Figura 4.7: Immagini da unire



(a) Base sfocata (b) Rumore (c) Mappa distanza

4.11 Posizionamento oggetti

Il posizionamento degli oggetti è un altro punto molto importante per una generazione il più verosimile possibile, la plausibilità è l'obbiettivo che tentiamo di raggiungere. Per fare ciò abbiamo a disposizione alcune tecniche create appositamente e specifiche per una tipologia di oggetti ma tutte seguono il medesimo approccio generale di inserimento descritto in seguito.

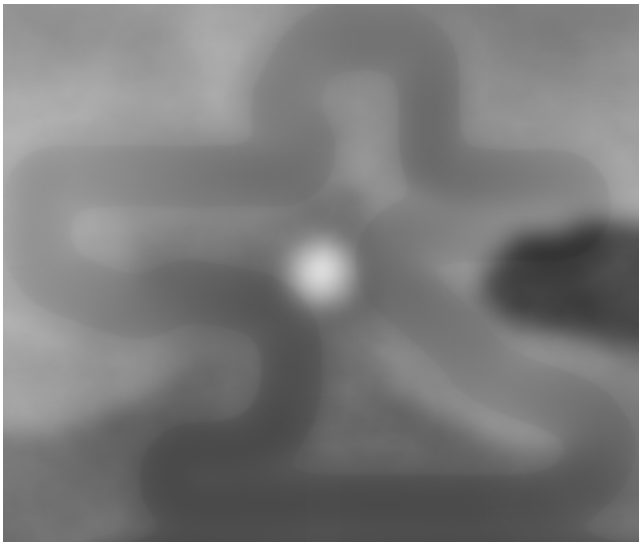


Figura 4.8: Risultato finale

Per quanto riguarda l'implementazione di questa parte facciamo riferimento alla gerarchia delle classi mostrata in appendice A.1, in cui è prevista una classe per ogni tipologia di generazione, ciò consente di implementare con buona flessibilità i metodi ereditati con la possibilità di raffinare l'inserimento in base alle caratteristiche peculiari di quel panorama.

L'immagine 4.9 rappresenta la struttura dati in cui individuiamo i punti di inserimento degli oggetti, l'immagine originale richiesta da TORCS dovrebbe essere su sfondo nero ma per motivi tipografici è mostrata con sfondo bianco. I punti, invece, dovrebbero essere del colore corrispondente all'ID che identifica univocamente un oggetto, ma per aumentarne la visibilità sono tutti neri e sempre per lo stesso motivo mostriamo solo una porzione ingrandita dell'immagine reale.

Quindi per effettuare il posizionamento di un singolo oggetto è necessario colorare un pixel nella posizione scelta e del colore corrispondente a ciò che si vuole inserire, il codice è presentato in appendice B.5.1; non è però sufficiente, infatti, bisogna effettuare un controllo per verificare se vi è abbastanza spazio disponibile, cioè se non si creeranno sovrapposizioni, come descritto nel prossimo paragrafo.

Figura 4.9: Mappa ogggetti

4.11.1 Mappa occupazione

Per risolvere il problema dell'intersezione usiamo un approccio bidimensionale, calcoliamo quindi, per ogni oggetto inserito, il suo riquadro minimo di delimitazione (o bounding box) riferito alle sole componenti x e y.

Nella figura 4.10 vediamo l'effetto finale di un'applicazione pratica, in bianco tutte le zone occupate, mentre in nero quelle ancora libere. Per facilitare l'operazione di controllo decidiamo di approssimare ogni oggetto ad un quadrato, è inoltre possibile notare che i quadrati bianchi tra di loro non si toccano rispettando il vincolo imposto, mentre in alcuni casi è possibile che siano sovrapposti alla pista. Di norma ciò non dovrebbe essere possibile poiché, per esempio potrebbe significare che c'è una casa in mezzo al tracciato, ma in questo caso vi è un'eccezione, infatti, gli oggetti in questione sono esclusivamente archi sospesi sulla pista che quindi non recano alcun problema.

4.11.2 Assunzioni sugli oggetti

Riuscire a posizionare gli oggetti correttamente impone di dover fare delle assunzioni su come gli oggetti stessi debbano essere costruiti, quindi dob-

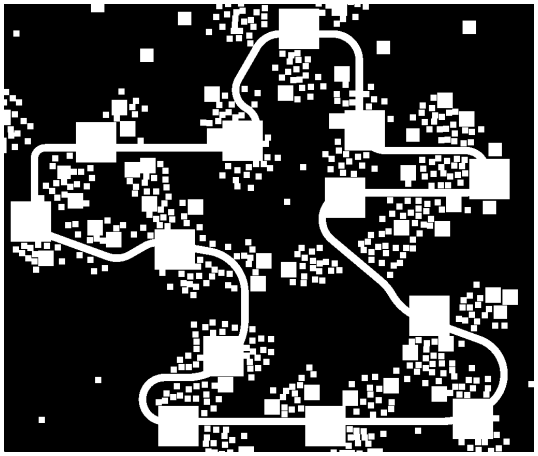


Figura 4.10: Mappa occupazione

biamo assumere implicite alcune informazione che altrimenti non potremmo determinare.

Per esempio la scala di ogni oggetto deve essere 1:1, ovvero deve avere le stesse dimensioni che avrebbe nella realtà così da rispettare il modello di TORCS che esprime tutte le misure nella stessa scala, inoltre, le distanze sono espresse in metri.

In generale ogni oggetto deve avere l'origine coincidente con il suo baricentro, così da poter essere sicuri che l'inserimento sarà effettivamente fatto alla coordinata desiderata e non in un punto traslato. Vi è però un eccezione: gli archi non devono rispettare questa condizione ma il loro centro deve essere spostato di circa il 20% da un'estremità, verso il centro, ciò perché questi non vengono inseriti specificando il punto centrale ma mediante un punto lontano dalla pista. Se così non fosse verrebbero ignorati poiché cadrebbe proprio sulla pista.

Anche la rotazione dovrebbe aderire ad una convenzione particolare; ciò però non vale in generale ma solo per alcune categorie particolari, le restanti infatti tipicamente vengono aggiunte con un angolo casuale quindi non vi è alcuna differenza concettuale se partire con un angolo o un altro. Esclusivamente per i cartelloni e gli archi utilizziamo una convenzione per la quale il lato

Tabella 4.1: Tecniche di posizionamento

Tecnica	*Oggetti*
Casuale	edifici e veicoli
Espansione a foresta	vegetazione
Perpendicolare al tracciato	archi e cartelloni

più lungo si sviluppa sull'asse delle x (positive nel caso degli archi), mentre il lato dell'altezza lungo l'asse delle z e quindi perpendicolare a x.

4.11.3 Tecniche di posizionamento

Classifichiamo ora le varie tipologie di posizionamento degli oggetti, suddivise come mostrato in tabella 4.1. Ciò è necessario dato che alcuni di essi richiedono un algoritmo particolare per poter essere inseriti in modo realistico.

Casuale

La tecnica più semplice in assoluto, presente in appendice B.5.3, permette di aggiungere gli oggetti in un punto casuale. Sorge però un problema, infatti, non sempre è possibile inserire il numero desiderato di oggetti, in quanto può capitare che tutto lo spazio sia occupato, quindi risolviamo inserendo un numero massimo di tentativi fissato sperimentalmente a due volte il numero di oggetti da inserire; così facendo non vi è il rischio di entrare in un ciclo infinito ma allo stesso tempo ha il difetto di non garantire l'inserimento del numero esatto richiesto nonostante esista ancora dello spazio libero.

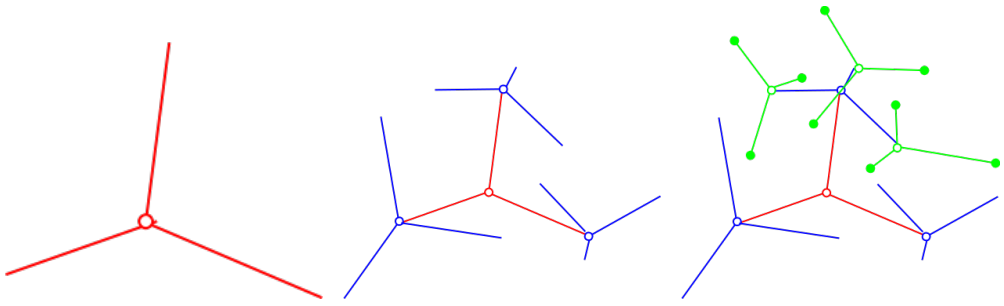
Questo difetto non è però di grande rilievo poiché il numero di oggetti da inserire è scelto in modo del tutto arbitrario e non rappresenta un risultato da raggiungere ma solo un indicazione di massima, cioè non richiede una precisa implementazione.

Espansione a foresta

Per simulare la creazione di una foresta facciamo riferimento all'algoritmo mostrato nel listato B.5.2, come si vede è un algoritmo ricorsivo basato su alcuni parametri tra cui il punto centrale, il numero di generazioni (o profondità), il numero di figli per generazione e altri che stabiliscono la distanza tra il punto attuale e il successivo impostando un intervallo di valori definito tra il valore minimo e quello massimo. Ogni passo la funzione esamina tutti i punti memorizzati e per ognuno di questi chiama ricorsivamente sé stessa decrementando la profondità, quando la profondità è nulla termina restituendo i punti ottenuti nell'ultima generazione.

Quindi di fatto generiamo un albero (nel senso di struttura dati) i cui nodi sono sparsi in uno spazio bidimensionale ma di cui memorizziamo solo le foglie. Le foglie dell'albero rappresentano i punti finali in cui inseriamo gli oggetti, nell'immagine 4.11 rappresentati dai pallini pieni, mentre i pallini vuoti rappresentano i punti necessari alla costruzione ma che di fatto alla fine vengono scartati.

Figura 4.11: Fasi generazione



(a) Primo passo (b) Secondo passo (c) Terzo passo

Perpendicolare al tracciato

Questa tecnica, il cui codice si trova in appendice B.5.4, permette di inserire un oggetto in un punto lontano dal tracciato e da esso perpendicolare. Per poter effettuare ciò dobbiamo prima ottenere la direzione normale uscente

al tracciato, che realizziamo mediante il metodo *getCouples* presentato nel paragrafo 4.9.1.

Ricaviamo facilmente la direzione normale al tracciato osservando che questo valore è approssimabile con la direzione tra un punto del bordo esterno e il punto del bordo interno che abbia distanza minima dal primo, cioè le coppie di punti ottenute con il metodo appena citato.

Trovato l'angolo possiamo determinare il punto in cui inserire l'oggetto, cioè seguendo la stessa direzione ci muoviamo con verso uscente in misura della distanza desiderata. Otteniamo quindi il punto in cui inserire l'oggetto, cosa che facciamo con la procedura generale di inserimento.

Questo è possibile grazie alle assunzioni fatte sugli oggetti presenti, quindi l'angolo di inserimento coincide con l'angolo della direzione normale al tracciato a cui si aggiunge l'angolo relativo alla rotazione richiesta (in genere $\pm 90°$), che non dipende da uno specifico oggetto ma assume sempre lo stesso valore, ciò che cambia è solo il segno che è positivo se siamo sul bordo interno, negativo per quello esterno.

Capitolo 5

Risultati e conclusioni

Una valutazione oggettiva dei risultati raggiunti non è affatto semplice in quanto quello che dovremmo misurare è un parametro estetico che richiederebbe una valutazione prettamente soggettiva, quindi per poter esprimere un giudizio sul lavoro svolto è importante mostrare alcune immagini dei risultati finali, lasciando al lettore il compito di criticarle.

5.1 Confronto

Di seguito alcune immagini 5.1 che mostrano una comparazione tra diverse ambientazioni applicate al tracciato *e-track-4*, di cui mostriamo come prima immagine la versione generata di default.

Siamo quindi in grado di valutare la validità dell'approccio usato, anche se non fa uso di tecniche particolarmente complesse, produce un risultato dignitoso e il tracciato finale risulta gradevole da giocare, sicuramente rende disponibili ambientazioni differenti e quindi la monotonia dell'unica ambientazione prevista di default viene a mancare.

Figura 5.1: Confronto ambientazioni, e-track-4

(a) Default

(b) Mountain-snow

(c) Hill

(d) Desert-orange

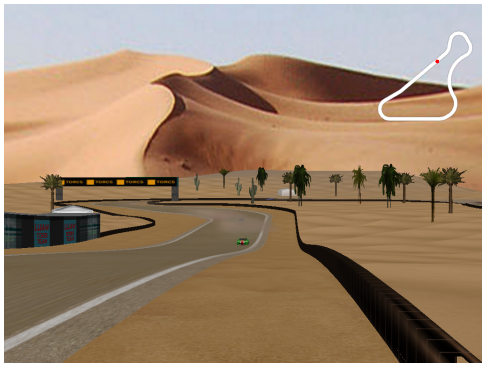
5.2 Scenari

Mostriamo quindi, a titolo d'esempio, un'immagine per ogni ambientazione esistente 5.2.

Il tracciato *HillRoad* 5.3(d) è stato inoltre utilizzato all'interno della competizione *The 2011 Simulated Car Racing (SCR) Championship @ Evo*-2011* organizzata da alcuni docenti del Politecnico di Milano e delle University of W*ü*rzburg di cui è stato girato un video visibile al seguente link [43].

Figura 5.2: Risultati

(a) City, *g-track-3*



(b) Desert, *g-track-1*

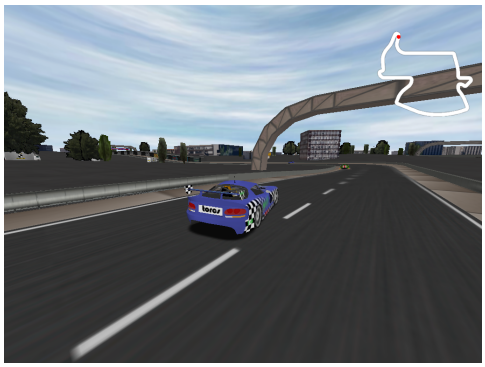
(c) Desert-orange, *e-track-2*

(d) Hill, *HillRoad*

(e) Lava, *e-track-1*

(f) Mountain, *eroad*

(g) Mountain-snow, *mango*



(h) Polimi, *aalboarg*

(i) Simpson, *eroad*

5.3 Tempi di esecuzione

Per eseguire alcuni test sui tempi necessari a ottenere una generazione completa abbiamo utilizzato un computer con processore AMD Turion X2 Ultra Dual-Core Mobile ZM-80 2,1 GHz e memoria di 4 GB. I test sono eseguiti sia con sistema operativo Windows 7 che Fedora 15 entrambi a 64 bit. I sono risultati mostrati in tabella 5.1.

Risulta evidente che in generale sotto il sistema operativo GNU-linux si ottengono risultati migliori, ciò è probabilmente dovuto da un minor consumo di risorse da parte del sistema operativo e in parte anche da una migliore ripartizione sulle due CPU da parte della JVM.

Inoltre, è importante evidenziare che in molti tracciati quasi l'80% del tempo impiegato per la generazione è da imputare alle due chiamate dello strumento esterno trackgen, quindi ottimizzando queste due procedure si potrebbe ridurre ulteriormente il tempo di esecuzione.

Tabella 5.1: Tempi esecuzione

Tracciato	*Windows 7*	*Fedora 15*
aalborg<hill, hill>	2'27"	1'59"
modifiche apportate	44"	43"
mappa distanza	15"	15"
espansione	58"	33"
trackgen elv4	17"	15"
trackgen -a	13"	13"
alpine-2<mountain, mountain>	6'27"	5'36"
modifiche apportate	17"	31"
mappa distanza	9"	14"
espansione	40"	20"
trackgen elv4	2'38"	2'25"
trackgen -a	2'41"	2'04"
e-track-4<desert, desert-orange>	6'58"	6'03"
modifiche apportate	31"	30"
mappa distanza	17"	18"
espansione	46"	28"
trackgen elv4	2'43"	2'31"
trackgen -a	2'40"	2'16"

5.4 Conclusioni

Nel progetto descritto l'obbiettivo è stato raggiunto, attualmente partendo da un qualsiasi tracciato ed alcuni parametri presi in input siamo in grado di generare un ambientazione abbastanza verosimile. Allo stesso tempo abbiamo dovuto accettare alcune limitazioni, le quali potrebbero essere superate in lavori futuri, nei quali si potrà utilizzare un approccio improntato direttamente alla modifica dell'ambientazione dal punto di vista tridimensionale piuttosto che utilizzare gli strumenti forniti da TORCS, prettamente bidimensionali. L'ideale sarebbe partire da un tracciato abbellito con il programma descritto in questa tesi ed arricchirlo di nuovi particolari agendo direttamente sullo spazio 3D.

Probabilmente un approccio simile soffrirebbe ugualmente dello stesso difetto del programma descritto nella tesi, cioè non vi sarebbe alcuna valutazione sulla qualità della generazione. Riguardo a ciò, si potrebbero introdurre alcuni giudizi da parte degli utenti in modo da poter migliorare sempre di più la generazione eliminando i tracciati peggio valutati. Si potrebbe quindi ipotizzare un'ambientazione che evolve nel tempo grazie ad algoritmi genetici, e che cerchi quindi di raggiungere, di volta in volta, una qualità superiore.

Un'altra possibile prospettiva sarebbe quella di svincolare il programma da TORCS e renderlo compatibile con altri videogiochi di guida, o addirittura ad altri videogiochi in genere che posseggano caratteristiche simili.

Bibliografia

[1] *The mathematical theory of L systems*. Academic Press, 1980.

[2] *The Fractal Geometry of Nature*. W. H. Freeman, 1983.

[3] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies - a comprehensive introduction. 1:3–52, May 2002.

[4] N. Chiba, K. Muraoka, and K. Fujita. An erosion model based on velocity fields for the visual simulation of mountain scenery. *Journal of Visualization and Computer Animation*, 9:185–194, 1998.

[5] D. D'Ambrosio, S. Di Gregorio, S. Gabriele, and R. Gaudio. a cellular automata model for soil erosion by water. *Physics and Chemistry of the Earth, Part B: Hydrology, Oceans and Atmosphere*, 26(1):33 – 39, 2001.

[6] Charles Darwin. *The Origin of Species*. Gramercy, May 1995.

[7] Miguel Monteiro de Sousa Frade. *Genetic Terrain Programming*. PhD thesis, Universidad de Extremadura, Spagna, 2008.

[8] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.

[9] Rafael Forsbach and Bedrich Beneš. Visual simulation of hydraulic erosion. *Journal of WSCG*, 10:2002, 2002.

[10] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25:371–384, June 1982.

[11] Kelly G. and McCabe H. A survey of procedural techniques for city generation. *ITB Journal*, 2006.

[12] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Real-time procedural generation of 'pseudo infinite' cities. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '03, pages 87–ff, New York, NY, USA, 2003. ACM.

[13] Hartmut Jurgens Heinz-Otto Peitgen and Dietmar Saupe. *Chaos and Fractals - New Frontiers of Science*. Springer, 2004.

[14] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[15] A. Lindenmayer. Mathematical models for cellular interactions in development. *Journal of Theoretical Biology*, 1968.

[16] Lionel March. Forty years of shape and shape grammars, 1971 - 2011. *Nexus Network Journal*, 13:5–13, 2011.

[17] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 614–623, New York, NY, USA, 2006. ACM.

[18] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Comput. Graph.*, 23:41–50, July 1989.

[19] Jacob Olsen. Realtime procedural terrain generation - realtime synthesis of eroded fractal terrain for use in computer games. *Department of Mathematics And Computer Science IMADA University of Southern Denmark*, 2004.

[20] Peter E. Oppenheimer. Real time design and animation of fractal plants and trees. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 55–64, New York, NY, USA, 1986. ACM.

[21] Ken Perlin. Improving noise. *Media Research Laboratory*, 2001.

[22] Alexandre Peyrat, Olivier Terraz, Stephane Merillou, and Eric Galin. Generating vast varieties of realistic leaves with parametric 2gmap l-systems. *Vis. Comput.*, 24:807–816, July 2008.

[23] P. Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[24] Armando Re, Francisco Abad, Emilio Camahort, and M. C. Juan. Tools for procedural generation of plants in virtual scenes. In *Proceedings of the 9th International Conference on Computational Science*, ICCS 2009, pages 801–810, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] George Stiny and James Gips. Shape grammars and the generative specification of painting and sculptur. *Information Processing*, 1971.

[26] Jing Sun, Xiaobo Yu, George Baciu, and Mark Green. Template-based generation of road networks for virtual city modeling. In *Proceedings of the ACM symposium on Virtual reality software and technology*, VRST '02, pages 33–40, New York, NY, USA, 2002. ACM.

[27] John Keyser Teong Joo Ong, Ryan Saunders and John J. Leggett. Terrain generation using genetic algorithms. *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, 2005.

[28] Julian Togelius, Georgios Yannakakis, Kenneth Stanley, and Cameron Browne. Search-Based Procedural Content Generation. In e. c. i. l. i. a. Di, Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna Esparcia-Alcazar, Chi-Keong Goh, Juan Merelo, Ferrante Neri, Mike Preuß, Julian Togelius, and Georgios Yannakakis, editors, *Applications*

of Evolutionary Computation, volume 6024 of *Lecture Notes in Computer Science*, chapter 15, pages 141–150. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.

[29] Georgy Voronoi. Nouvelles applications des paramàtres continus à la thèorie des formes quadratiques. *Journal für die Reine und Angewandte Mathematik*, 1907.

[30] Richard Voss. Fractals in nature: characterization, measurement, and simulation. *SIGGRAPH*, 1987.

[31] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 119–128, New York, NY, USA, 1995. ACM.

[32] Andrea L. Wiens and Brian J. Ross. Gentropy: evolving 2D textures. *Computers and Graphics*, 26(1):75–88, February 2002.

[33] Ac3d file format. `http://www.inivis.com/ac3d/man/ac3dfileformat.html`.

[34] Arbaro. `http://arbaro.sourceforge.net/`.

[35] GAR. `http://gar.eecs.ucf.edu/`.

[36] gen3. `http://wiki.blender.org/index.php/Extensions:2.4/Py/Scripts/Wizards/Gen3`.

[37] Java 3D vecmath. `http://java.net/projects/vecmath/`.

[38] JDOM. `http://www.jdom.org/`.

[39] JHLABS filters. `http://www.jhlabs.com/`.

[40] JTexGen. `http://jtexgen.sourceforge.net/`.

[41] lista distribuzione (mailing list). http://groups.google.com/proceduralcontent.

[42] NG Plants. http://ngplant.sourceforge.net/.

[43] The 2011 Simulated Car Racing (SCR) Championship @ Evo*-2011. http://www.youtube.com/watch?v=52T8t7UQNrk&feature=player_embedded.

[44] Trask Force IEEE CIS GTC. http://game.itu.dk/pcg/.

[45] wiki su PCG. http://pcg.wikidot.com.

Appendice A

Documentazione del progetto logico

A.1 Diagrammi UML

All'interno della figura A.1, relativa all'elevazione, possiamo vedere come viene implementata l'interfaccia con i metodi necessari, mentre la classe astratta aggiunge altri metodi comuni e utili a tutte le sottoclassi.

Per quanto riguarda la gerarchia delle classi inerenti all'inserimento di oggetti abbiamo la figura A.2, la cui struttura è simile alla precedente ma leggermente più complessa in quanto l'inserimento di un nuovo oggetto deve basarsi anche sulla cosiddetta mappa occupazioni (cioè la classe *BusyMap*), per evitare sovrapposizioni.

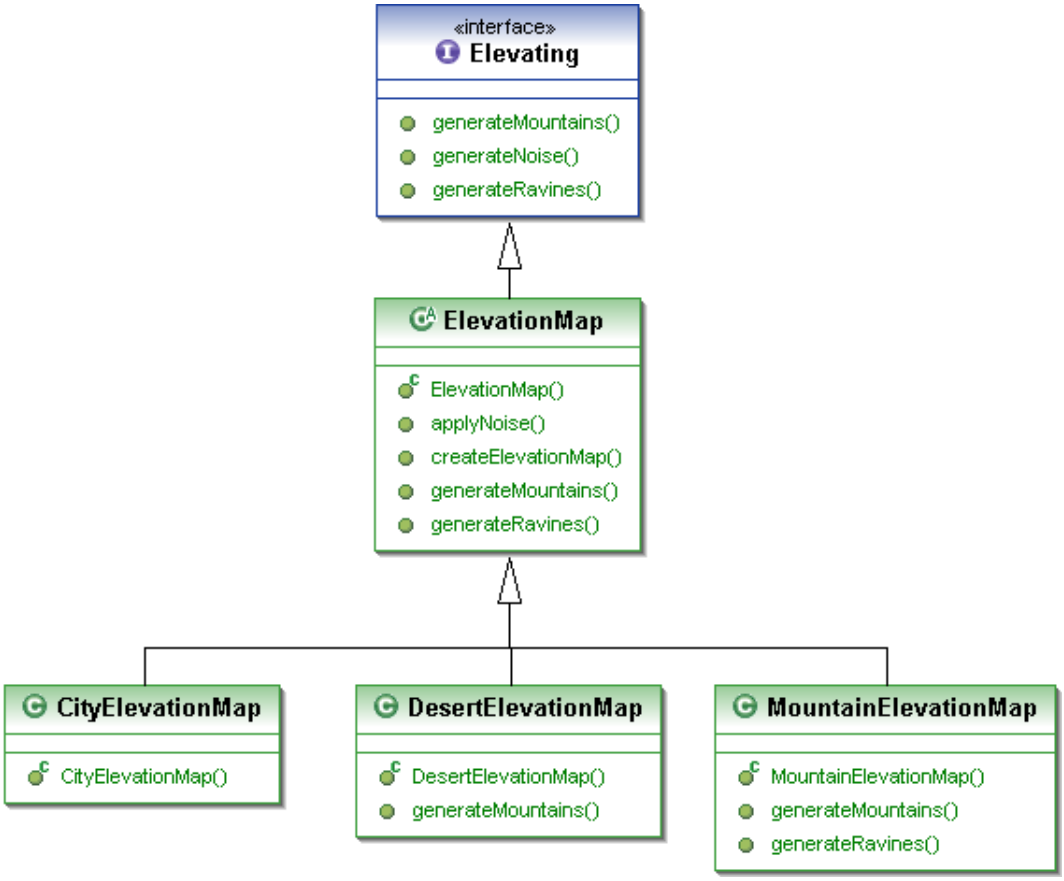


Figura A.1: Diagramma delle classi relativo all'elevazione

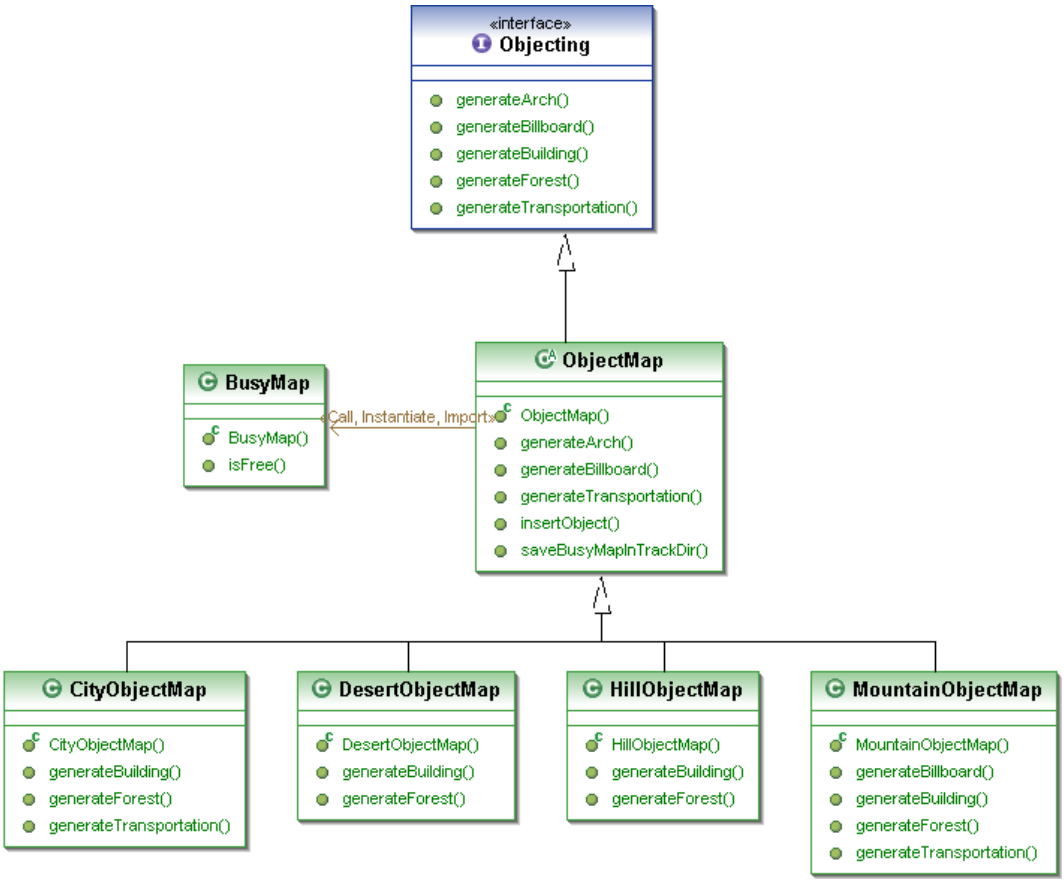


Figura A.2: Diagramma delle classi relativo agli oggetti

Appendice B

Listato

Di seguito riportiamo le porzioni di codice più interessanti e maggiormente citate all'interno dell'intera trattazione.

B.1 Utilità

B.1.1 IntPoint

```
public class IntPoint implements Cloneable{
  private int x;
  private int y;

  public IntPoint(int x, int y) {
    super();
    this.x = x;
    this.y = y;
  }

  public IntPoint(){
    this(0,0);
  }
  ...
}
```

Listato

getAdjacentsClockwise

```
//restituisce gli otto punti adiacenti in senso orario, partendo
    da nord-ovest
public IntPoint[] getAdjacentClockwise(){
  IntPoint[] adjacentPoints = new IntPoint[8];//
  adjacentPoints[7] = new IntPoint(x - 1, y );
  adjacentPoints[6] = new IntPoint(x - 1, y + 1);
  adjacentPoints[5] = new IntPoint(x, y + 1);
  adjacentPoints[4] = new IntPoint(x + 1, y + 1);
  adjacentPoints[3] = new IntPoint(x + 1, y);
  adjacentPoints[2] = new IntPoint(x + 1, y - 1);
  adjacentPoints[1] = new IntPoint(x, y - 1);
  adjacentPoints[0] = new IntPoint(x - 1, y - 1);
  return adjacentPoints;
}
```

getAdjacentsCounterClockwise

```
//restituisce gli otto punti adiacenti in senso antiorario,
    partendo da nord-ovest
public IntPoint[] getAdjacentCounterClockwise(){
  IntPoint[] adjacentPoints = new IntPoint[8];
  adjacentPoints[0] = new IntPoint(x - 1, y - 1);
  adjacentPoints[1] = new IntPoint(x - 1, y );
  adjacentPoints[2] = new IntPoint(x - 1, y + 1);
  adjacentPoints[3] = new IntPoint(x, y + 1);
  adjacentPoints[4] = new IntPoint(x + 1, y + 1);
  adjacentPoints[5] = new IntPoint(x + 1, y);
  adjacentPoints[6] = new IntPoint(x + 1, y - 1);
  adjacentPoints[7] = new IntPoint(x, y - 1);
  return adjacentPoints;
}
```

getAdjacentXYFromPrevious

```
//restituisce un vettore di punti adiacenti ordinati partendo dal
    punto precedente in senso orario o antiorario a seconda del
    parametro
public IntPoint[] getAdjacentXYFromPrev(IntPoint previousPoint,
    boolean clockwise){
  IntPoint[] adjacentPoints;
  if (clockwise) adjacentPoints = getAdjacentClockwise();
  else adjacentPoints = getAdjacentCounterClockwise();
  //cerca previous in adjacent
  int foundedPos = searchPoint(adjacentPoints,previousPoint);
  //shift del vettore
  if (foundedPos >= 0) {
    if (!clockwise) foundedPos = foundedPos * -1;
    Utilities.shiftArray(adjacentPoints, foundedPos);
  }
  else{
    System.out.println("ERROR");
  }
  return adjacentPoints;
}
```

B.2 Elaborazioni immagini

B.2.1 ImageWithName

```
public abstract class ImageWithName {
  private String name;
  protected BufferedImage image;
  ...
}
```

B.2.2 Classe Elv3

```
public class Elv3 extends ImageWithName {
  private boolean firstLoop = true;
```

Listato

```
  private ArrayList<IntPoint> last3;

  public Elv3() {
    this(Paths.elvPath[3], Paths.elvName[3]);
  }

  public Elv3(String elv3Path, String name){
    super();
    super.setName(name);
    last3 = new ArrayList<IntPoint>(3);
    try {
      super.setImage(ImageIO.read(new File(elv3Path)));
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
  ...
}
```

getBounds

```
//restituisce una lista dei punti del bordo pista interno o
    esterno a seconda del parametro
public ArrayList<IntPoint> getBounds(boolean internal){
  ArrayList<IntPoint> boundaryPoints = new ArrayList<IntPoint>();
  firstLoop = true;
  IntPoint tempPoint;
  IntPoint startPoint;
  IntPoint currentPoint;
  IntPoint previousPoint;

  //bordo interno
  if (internal) {
    //punto di partenza, primo punto bianco trovato
    startPoint = this.findInternalWhite();
  }
  //bordo esterno
  else {
```

```
    //punto di partenza, primo punto bianco trovato
    startPoint = this.findFirstWhite();
  }
  currentPoint = (IntPoint) startPoint.clone();
  //iniziallizzo punto precedente
  previousPoint = this.findPrevious(internal, startPoint);
  if (previousPoint == null){
    System.err.println("ERROR");
    System.exit(-1);
  }
  firstLoop = true;
  while(!(currentPoint.equals(startPoint) && firstLoop == false)){
    IntPoint tempIntPoint = (IntPoint) currentPoint.clone();
    boundaryPoints.add(tempIntPoint);
    firstLoop = false;
    tempPoint = (IntPoint) currentPoint.clone();
    //calcolo punto successivo
    currentPoint = nextBoundPoint(internal, previousPoint,
        startPoint, currentPoint);
    if (currentPoint == null){
      System.err.println("ERROR");
      System.exit(-1);
    }
    previousPoint = (IntPoint) tempPoint.clone();
  }
  return boundaryPoints;
}
```

getCouples

```
//restituisce una lista di coppie di punti che tra di loro sono
    bordi pista a distanza minima
public ArrayList<IntCouple> getCouples(int spacingMeter){
  //Conversione
  int spacingPixel = MathUtilities.fromMeterToPixel(spacingMeter);

  ArrayList<IntPoint> internalBoundPoint = this.getBounds(true);
  ArrayList<IntPoint> externalBoundPoint = this.getBounds(false);
```

Listato

```
  ArrayList<IntCouple> out = new ArrayList<IntCouple>();

  //offset iniziale, valore casuale
  int randomOffset = (int) (new Random().nextInt(spacingPixel));

  for (int i = randomOffset; i < externalBoundPoint.size(); i = i
      + spacingPixel) {
    IntPoint extPoint = externalBoundPoint.get(i);
    Point2D.Double extTemp = new Point2D.Double(extPoint.getX(),
        extPoint.getY());
    IntPoint minDisPoint = new IntPoint();
    double min = Double.MAX_VALUE;
    for (IntPoint intPoint : internalBoundPoint) {
      //passo da punto di interi a punto di double
      Point2D.Double pointTemp = new Point2D.Double(intPoint.getX
          (), intPoint.getY());
      //calcolo distanza
      double dist = extTemp.distance(pointTemp);
      if (dist < min){
        min = dist;    //distanza minima
        //punto del bordo interno a distanza minima
        minDisPoint = new IntPoint((int)pointTemp.getX(), (int)
            pointTemp.getY());
      }
    }
    IntCouple tempC = new IntCouple(extPoint, minDisPoint);
    out.add(tempC);
  }
  return out;
}
```

getMinDistancePoint

```
//restituisce il punto del tracciato a distanza minima dal punto
    preso come parametro
public IntPoint getMinDistancePoint(IntPoint point){
  int min = Integer.MAX_VALUE;
  IntPoint minPoint = null;
```

```
  for (int theta = 0; theta < 360; theta = theta + 45){
    int distance = getDinstance(point, theta);
    if (distance < min) {
      min = distance;
      IntPoint unitVector = point.getUnitVector(theta);
      int newX = unitVector.getX() * distance + point.getX();
      int newY = unitVector.getY() * distance + point.getY();
      minPoint = new IntPoint(newX, newY);
    }
  }
  return minPoint;
}
```

findInternalWhite

```
//cerca il primo pixel del bordo pista interno, che corrisponde ad
    un pixel bianco
private IntPoint findInternalWhite(){
  IntPoint firstEx = this.findFirstWhite();
  ArrayList<IntPoint> extBound = this.getBounds(false);
  IntPoint previous = findPrevious(false, firstEx);
  int firstExX = firstEx.getX();
  int firstExY = firstEx.getY();
  ArrayList<IntPoint> tempPoints = new ArrayList<IntPoint>();
  //vado in tutte le direzioni finchè non trovo un bordo pista che
      non è contenuto in external
  for (int k = 1; k < Math.max(getHeight(), getWidht()); k++){
    tempPoints = firstEx.getAdjacents(firstEx, k);
    for (IntPoint point : tempPoints){
      if (isBoundOfTrack(point) && !extBound.contains(point)){
        return point;
      }
    }
  }
  System.err.println("### ERROR ###");
  return null;
}
```

Listato

findFirstWhite

```
//cerca il primo pixel del bordo pista esterno, cioè un pixel di
    confine di colore bianco
private IntPoint findFirstWhite(){
  for (int x = 1; x < image.getWidth(); x++){
    for (int y = 1; y < image.getHeight(); y++){
      if ((image.getRGB(x, y) == (new Color(255,255,255).getRGB())
          ) &&
           (isBoundOfTrack(new IntPoint(x, y)))){
        return new IntPoint(x,y);
      }
    }
  }
  return null;
}
```

findPrevious

```
//restituisce il punto del tracciato a distanza minima dal punto
    preso come parametro
public IntPoint getMinDistancePoint(IntPoint point){
  int min = Integer.MAX_VALUE;
  IntPoint minPoint = null;
  for (int theta = 0; theta < 360; theta = theta + 45){
    int distance = getDinstance(point, theta);
    if (distance < min) {
      min = distance;
      IntPoint unitVector = point.getUnitVector(theta);
      int newX = unitVector.getX() * distance + point.getX();
      int newY = unitVector.getY() * distance + point.getY();
      minPoint = new IntPoint(newX, newY);
    }
  }
  return minPoint;
}
```

nextBoundPoint

```
//cerca il prossimo punto bordo pista
public IntPoint nextBoundPoint(Boolean internal, IntPoint
    previousPoint, IntPoint startPoint, IntPoint currentPoint){
  IntPoint[] adjacents;
  if (internal) adjacents = currentPoint.getAdjacentXYFromPrev(
      previousPoint, false);
  else adjacents = currentPoint.getAdjacentXYFromPrev(
      previousPoint, true);
  for (int k = 0; k < 8; k++){
    if ((isBoundOfTrack(adjacents[k]) == true) &&
        !(adjacents[k].equals(previousPoint)) && (last3.contains(
            adjacents[k]) == false)){
      if (last3.size() > 2 ) last3.remove(0);
      last3.add(adjacents[k]);
      return (IntPoint) adjacents[k].clone();
    }
  }
  return null;
}
```

B.2.3 Classe Elv4

```
public class Elv4 extends ImageWithName {
  private Elv3 elv3;

  public Elv4(Elv3 elv3) {
    this(elv3, Paths.elvPath[4], Paths.elvName[4]);
  }

  public Elv4(Elv3 elv3, String elv4Path, String name) {
    super();
    this.elv3 = elv3;
    super.setName(name);
    try {
      super.setImage(ImageIO.read(new File(elv4Path)));
```

```
        height = this.image.getHeight();
        widht = this.image.getWidth();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
    ...
}
```

getNearestColor

```
//restituisce il colore corrispondente al punto più vicino al
    bordo pista che abbia minima distanza dal punto preso come
    parametro
public Color getNearestColor(IntPoint point){
  //Ottiene il punto del bordo pista più vicino
  IntPoint tempPoint = elv3.getMinDistancePoint(point);
  //tutti i punti adiacenti a quest'ultimo
  IntPoint[] adjacents = tempPoint.getAdjacentClockwise();
  for (IntPoint intPoint : adjacents) {
    try {
      int tempColor = image.getRGB(intPoint.getX(), intPoint.getY
          ());
      if (tempColor != (Color.WHITE).getRGB()){
        return new Color(tempColor);
      }
    }
    catch (ArrayIndexOutOfBoundsException e) {
    }
  }
  return Color.BLACK;
}
```

B.3 Mappa distanza

B.3.1 Prima versione

```
//crea la mappa delle distanze
public BufferedImage createDistanceMap(int maxDistance){
  if (maxDistance < 0) maxDistance = 0;
  if (maxDistance > 255) maxDistance = 255;
  BufferedImage distanceMap = new BufferedImage(image.getWidth(),
      image.getHeight(), image.getType());
  for (int i = 0; i < image.getWidth(); i++) {
    for (int j = 0; j < image.getHeight(); j++) {
      IntPoint tempPoint = new IntPoint(i, j);

      //ottengo la distanza minima dal tracciato, complessità O(x
          ^2)
      int distance = this.getMinDistance(tempPoint);
      if (distance > maxDistance) distance = maxDistance;
      distanceMap.setRGB(i, j, new Color(distance, distance,
          distance).getRGB());
    }
  }
  try {
    ImageIO.write(distanceMap, "png", new File(Paths.
        distanceMapPath));
    System.out.println("Fine generazione");
  } catch (IOException e) {
    System.out.println("### ERROR ###");
    System.exit(-1);
  }
  return distanceMap;
}

//restituisce il valore di distanza minima dal bordo pista, solo
    in 8 direzioni
public int getMinDistance(IntPoint point){
  int min = Integer.MAX_VALUE;
  for (int theta = 0; theta < 360; theta = theta + 45){
```

```
    int distance = getDinstance(point, theta);
    if (distance < min) min = distance;
  }
  return min;
}
```

B.3.2 Seconda versione: riduzione di complessità

```
//crea la mappa delle distanze
public BufferedImage createDistanceMap2(int maxDistance){
  //ottengo tutti i punti in corrispondenza dei bordi pista interi
      ed esterni
  ArrayList<IntPoint> internalBounds = this.getBounds(true);
  ArrayList<IntPoint> externalBounds = this.getBounds(false);

  //unisco i bordi pista in un'unica struttura dati
  ArrayList<IntPoint> allBounds = new ArrayList<IntPoint>();
  allBounds.addAll(externalBounds);
  allBounds.addAll(internalBounds);

  if (maxDistance < 0) maxDistance = 0;
  if (maxDistance > 255) maxDistance = 255;
  BufferedImage distanceMap = new BufferedImage(image.getWidth(),
      image.getHeight(), image.getType());
  for (int i = 0; i < image.getWidth(); i++) {
    for (int j = 0; j < image.getHeight(); j++) {
      IntPoint tempPoint = new IntPoint(i, j);
      double distance = this.getMinDistFromBound(tempPoint,
          allBounds);
      if (distance > maxDistance) distance = maxDistance;
      int intDist = (int) distance;
      distanceMap.setRGB(i, j, new Color(intDist, intDist, intDist
          ).getRGB());
    }
  }
  try {
```

```
    ImageIO.write(distanceMap, "png", new File(Paths.
        distanceMapPath));
    System.out.println("Fine generazione distance-map");
    System.out.println(new Time().toString());
  } catch (IOException e) {
    System.out.println("### ERROR ###");
    System.exit(-1);
  }
  return distanceMap;
}

//restituisce il valore di distanza minima dal bordo pista
private double getMinDistFromBound(IntPoint point, ArrayList<
    IntPoint> points){
  double minDist = Double.MAX_VALUE;
  double x = point.getX();
  double y = point.getY();
  Point2D.Double dPoint = new Point2D.Double(x, y);
  for (IntPoint intPoint : points) {
    double tempX = intPoint.getX();
    double tempY = intPoint.getY();
    Point2D.Double tempPoint = new Point2D.Double(tempX, tempY);
    double tempDist = dPoint.distance(tempPoint);
    if (tempDist < minDist) minDist = tempDist;
  }
  return minDist;
}
```

B.3.3 Versione attuale: approssimazione

```
//crea la mappa delle distanze
public BufferedImage createDistanceMap3(int maxDistance){
  //ottengo tutti i punti in corrispondenza dei bordi pista interi
      ed esterni
  ArrayList<IntPoint> internalBounds = this.getBounds(true);
  ArrayList<IntPoint> externalBounds = this.getBounds(false);
```

Listato

```
//unisco i bordi pista in un'unica struttura dati
ArrayList<IntPoint> allBounds = new ArrayList<IntPoint>();
allBounds.addAll(externalBounds);
allBounds.addAll(internalBounds);

int w = image.getWidth();
int h = image.getHeight();
BufferedImage distanceMap = new BufferedImage(w, h, image.
    getType());
Graphics2D graDist = distanceMap.createGraphics();

//approssimazione adottata, deve essere pari per avere una metà
    intera
int spacing = 6;
int halfSpacing = spacing / 2;

for (int i = halfSpacing; i < image.getWidth(); i = i + spacing)
    {
  for (int j = halfSpacing; j < image.getHeight(); j = j +
      spacing) {
    IntPoint tempPoint = new IntPoint(i, j);
    double distance = this.getMinDistFromBound(tempPoint,
        allBounds);
    if (distance > maxDistance) distance = maxDistance;
    int intDist = (int) distance;
    if (intDist < 0) intDist = 0;
    graDist.setColor(new Color(intDist, intDist, intDist));
    graDist.fillRect(i - halfSpacing, j - halfSpacing, spacing,
        spacing);
  }
}
//sfoca l'immagine appena generata
GaussianFilter gaus = new GaussianFilter(60);
BufferedImage distanceBlurred = new BufferedImage(w, h,
    distanceMap.getType());
distanceBlurred = gaus.filter(distanceMap, distanceBlurred);
try {
```

```
    ImageIO.write(distanceBlurred, "png", new File(Paths.
        distanceMapPath));
    System.out.println("Fine generazione distance-map");
    System.out.println(new Time().toString());
  } catch (IOException e) {
    System.out.println("### ERROR ###");
    System.exit(-1);
  }
  return distanceBlurred;
}

//restituisce il valore di distanza minima dal bordo pista
private double getMinDistFromBound(IntPoint point, ArrayList<
    IntPoint> points){
  double minDist = Double.MAX_VALUE;
  double x = point.getX();
  double y = point.getY();
  Point2D.Double dPoint = new Point2D.Double(x, y);
  for (IntPoint intPoint : points) {
    double tempX = intPoint.getX();
    double tempY = intPoint.getY();
    Point2D.Double tempPoint = new Point2D.Double(tempX, tempY);
    double tempDist = dPoint.distance(tempPoint);
    if (tempDist < minDist) minDist = tempDist;
  }
  return minDist;
}
```

B.4 Espansione

```
//espande la matrice che rappresenta la mappa elevazione iniziale
    riempiendo le celle vuote(bianche)
int tempCounter = 1;
//non esistono più punti bianchi se tempCounter = 0
while (tempCounter > 0) {
  tempCounter = 0;
  for (int i = 0; i < h; i++) {
```

```
    for (int j = 0; j < w; j++) {
      int tempVal = mat[j][i];
      if (tempVal == 255 && getNumberOfAdjacentsMat(j, i) > 1){
        int avg = this.getAdjacentsAvgMat(j, i);
        mat[j][i] = avg;
        tempCounter ++;
      }
    }
  }
  //copia una matrice in un altra matrice
  for (int i = 0; i < imageMat.length; i++) {
    for (int j = 0; j < imageMat[i].length; j++) {
      imageMat[i][j] = mat[i][j];
    }
  }
}
```

B.5 Posizionamento Oggetti

B.5.1 Inserimento

```
//restituisce "true" se e solo se l'inserimento va a buon fine
public boolean insertObject(IntPoint point, int idColor, boolean
    notCheck){
  boolean out = false;
  try {
    Mesh mesh = xmlTrack.getMeshById(idColor);
    if (busyMap.isFree(point, mesh, notCheck)){
      try{
        this.image.setRGB(point.getX(), point.getY(), idColor);
        out = true;
      }
      catch (ArrayIndexOutOfBoundsException e) {
      }
    }
    return out;
  //se non vi è alcun oggetto solleva un eccezione
```

```
  } catch (NoObjectException e) {
    System.out.println("### WARNING ###");
  }
  return out;
}
```

B.5.2 Espansione a foresta

```
//avvia la generazione di una foresta
public void startGeneratePoints(){
  clearPoints();
  generatePoints(0, startingPoint);
  //    System.out.println("### DEBUG numero di punti: " + points.
      size());
}
```

```
//genera i punti che costituiranno la foresta aggiungendoli alla
    lista "points"
public void generatePoints(int actualDeep, MyPoint fatherPoint){
  Vector<MyPoint> nextPoints;
  nextPoints = nextPoints(fatherPoint);
  Iterator<MyPoint> pointsIterator = nextPoints.iterator();

  if (deep <= actualDeep) {
    //mi fermo, caso base
    while (pointsIterator.hasNext()) {
      MyPoint myPoint = (MyPoint) pointsIterator.next();
      points.add(myPoint);
    }
  }
  else {
    while (pointsIterator.hasNext()) {
      MyPoint myPoint = (MyPoint) pointsIterator.next();
      actualDeep++;
      generatePoints(actualDeep, myPoint);
    }
  }
```

```
}
```

```
//inserisce nella mappa oggetti tutti i punti presenti nella lista
    "points" assegnando un colore casuale tra quelli previsti per
   la vegetazione
public void drawForest(){
  Iterator<MyPoint> iteratorOfPoints = points.iterator();
  ArrayList<Integer> colors;
  try {
    colors = xmlTrack.getIdColor(ObjectType.VEGETATION);
    while (iteratorOfPoints.hasNext()) {
      MyPoint myPoint = (MyPoint) iteratorOfPoints.next();
      //traslo e mi porto nel sistema di riferimento dell'immagine
          e converto in interi
      int tempX = (int)((myPoint.getX()));
      int tempY = (int)((myPoint.getY()));
      try {
        int randCol = (Integer) Utilities.randomFromCollection(
            colors);
        objectMap.insertObject(new IntPoint(tempX, tempY), randCol
            , false);
      }
      catch (ArrayIndexOutOfBoundsException e) {
        //do nothing
      }
    }
  } catch (NoObjectException e1) {
    System.out.println("### WARNING ###");
  }
}
```

B.5.3 Casuale

```
//iserisce "number" oggetti appartenenti alla lista idColors in
   posizioni casuali sulla mappa oggetti
```

```
protected void insertRandomObjects(int number, ArrayList<Integer>
    idColors){
  boolean succes;
  int j = 0;

  //conversione
  number = MathUtilities.fromMeterToPixel(number);

  for (int i = 0; (j < number && i < 2 * number); i++) {
    Integer tempIdColor = (Integer) Utilities.randomFromCollection
        (idColors);
    if (tempIdColor != null){
      Random rand = new Random();
      int tX = rand.nextInt(widht);
      int tY = rand.nextInt(height);
      IntPoint tempPoint = new IntPoint(tX, tY);
      succes = this.insertObject(tempPoint, tempIdColor, false);
      if (succes) j++;
    }
    else {
      System.out.println("### WARNING ###");
    }
  }
}
```

B.5.4 Perpendicolare al tracciato

```
//inserisce un oggetto in direzione della coppia di punti presa
    come parametro a una certa distanza
protected void insertNormalDistPoint(IntCouple couple, boolean
    internal, ArrayList<Integer> colors, int distance, boolean
    notCheck) throws NoObjectException {
  //conversione
  distance = MathUtilities.fromMeterToPixel(distance);

  MyPoint out = null;
  IntCouple tempC = couple;
```

```
IntPoint extRandPoint = tempC.getFirst();
IntPoint intRandPoint = tempC.getSecond();

if (intRandPoint != null){
  int idColor;
  // uno a caso
  idColor = (Integer) Utilities.randomFromCollection(colors);

  Mesh mesh = xmlTrack.getMeshById(idColor);
  String objectName = mesh.getName();
  File objectPath = new File(mesh.getFilePath());
  String objPathName = objectPath.getName();
  MyPoint intP = intRandPoint.toMyPoint();
  MyPoint extP = extRandPoint.toMyPoint();

  double distExtInt = Math.abs(intP.distance(extP));
  // angolo in radianti tra il punto esterno e quello interno
  double angRad = extP.getRadDegree(intP, true);
  double deg = Math.toDegrees(angRad);
  deg =  deg + 180;

  //esterno
  if (!internal){
    idColor = this.getRightId(objPathName, deg);
    out = extP.getDistFromPoint((distance), angRad);
  }

  //interno
  if (internal){
    idColor = this.getRightId(objPathName, deg);
    out = intP.getDistFromPoint((-1) * (distance), angRad);
  }
  boolean witness = this.insertObject(out.toIntPoint(), idColor,
      notCheck);
}
else {
  System.out.println("### WARNING ###");
}
```

```
}
```

Appendice C

Il manuale utente

Il programma è scritto in Java 6.0 SE quindi è multipiattaforma cioè compatibile con più sistemi operativi, sicuramente è funzionante sia sotto ambiente Windows che Linux.

C.1 Struttura xml del tracciato

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE params SYSTEM "../../../../src/libs/tgf/params.dtd" >

<params name="test" type="param" mode="mw">
  <section name="Surfaces">
  </section>

  <section name="Objects">
  </section>

  <section name="Header">
  </section>

  <section name="Graphic">
  </section>

  <section name="Main Track">
```

```
    <section name="Track Segments">
    </section>

    <section name="Pits">
    </section>
  </section>

  <section name="Cameras">
  </section>
</params>
```

Il codice appena mostrato è la struttura del file xml del tracciato, è suddiviso in sezioni ognuna delle quali specifica un elemento differente del tracciato.

Le sezioni più importanti vengono descritte dettagliatamente nei seguenti paragrafi.

C.1.1 Objects

Contiene gli oggetti presenti nel circuito a cui viene assegnato un colore univoco che potrà essere utilizzato quante volte si vuole all'interno della mappa degli oggetti indicando così la posizione esatta di posizionamento nell'ambiente.

Qui di seguito mostriamo un esempio di oggetto:

```
<section name="arch-g1SD-1-13.0">
  <attstr name="object" val="arch-g1SD-1.ac" />
  <attnum name="color" val="0x00FF01" />
  <attstr name="orentation type" val="standard" />
  <attnum name="orientation" val="13.0" unit="deg" />
  <attnum name="delta height" val="4.0" />
  <attnum name="delta vert" val="10.0" />
</section>
```

C.1.2 Surfaces

Permette di specificare tutte le superfici richieste nel circuito impostando i vari parametri delle grandezze fisiche, texture da utilizzare e altri parametri secondari.

```
<section name="nome-logico-superficie">
  <attnum name="color R1" val="0.1" />
  <attnum name="color G1" val="0.1" />
  <attnum name="color B1" val="0.1" />
  <attnum name="color R2" val="0.2" />
  <attnum name="color G2" val="0.2" />
  <attnum name="color B2" val="0.2" />
  <attstr name="texture name" val="nome-immagine-superficie" />
  <attstr name="texture type" in="discrete, continous" val="
      continous" />
  <attnum name="texture size" val="20.0" unit="m" />
  <attnum name="texture mipmap" val="4.0" />
  <attstr name="bump name" val="" />
  <attnum name="bump size" val="2.0" unit="m" />
  <attnum name="friction" val="1.0" />
  <attnum name="rolling resistance" val="0.0050" />
  <attnum name="roughness" val="0.0" />
  <attnum name="roughness wavelength" val="1.0" />
</section>
```

C.1.3 Header

In questa sezione vengono raccolte alcune informazioni come il tipo di tracciato, il suo nome, l'autore e la descrizione del tracciato.

```
<section name="Header">
  <attstr name="name" val="alpine-2-mountain mountain-snow" />
  <attstr name="category" val="road" />
  <attnum name="version" val="4" />
  <attstr name="author" val="D. Schellhammer" />
  <attstr name="description" val="Slow mountain road" />
<\section>
```

C.1.4 Graphic

Contiene tutte le informazioni utili per generare il file del tracciato della pista, cioè lo sfondo, i parametri per la generazione del terreno quindi la mappa oggetti e quella elevazione, nonché l'altezza minima e massima dell'ambientazione stessa.

```
<section name="Graphic">
  <attstr name="3d description" val="alpine-2-mountain_mountain-
      snow.ac" />
    <attstr name="background image" val="background.png" />
    <attnum name="background type" val="4" />
    <attnum name="background color R" val="0.82" />
    <attnum name="background color G" val="0.85" />
    <attnum name="background color B" val="0.95" />
    <attnum name="light position x" val="-900" />
    <attnum name="light position y" val="3220" />
    <attnum name="light position z" val="1543" />
    <attnum name="ambient color R" val="0.08" />
    <attnum name="ambient color G" val="0.08" />
    <attnum name="ambient color B" val="0.1" />
    <attnum name="diffuse color R" val="1" />
    <attnum name="diffuse color G" val="1" />
    <attnum name="diffuse color B" val="1" />
    <attnum name="specular color R" val="0.6" />
    <attnum name="specular color G" val="0.6" />
    <attnum name="specular color B" val="0.6" />
    <attnum name="fov factor" val="1.0" />
    <section name="Terrain Generation">
      <attnum name="track step" unit="m" val="20" />
      <attnum name="border margin" unit="m" val="50" />
      <attnum name="border step" unit="m" val="30" />
      <attnum name="border height" unit="m" val="15" />
      <attstr name="orientation" val="counter-clockwise" />
      <attstr name="elevation map" val="alpine-2-mountain_mountain
          -snow-elevation.png" />
      <attnum name="maximum altitude" val="150.0" />
      <attnum name="minimum altitude" val="-60.0" />
```

```
      <attnum name="group size" val="100" />
      <attstr name="surface" val="terrain-snow4" />
      <section name="Object Maps">
        <section name="map1">
          <attstr name="object map" val="alpine-2-
              mountain_mountain-snow-object-map.png" />
        </section>
      </section>
    </section>
    <section name="Environment Mapping">
      <section name="general">
        <attstr name="env map image" val="env.rgb" />
      </section>
    </section>
</section>
```

C.1.5 Main Track

Al suo interno sono specificati tutti i segmenti e le curve che andranno a costruire il tracciato vero e proprio, quindi anche tutte le informazioni legate a un tratto di pista come la superficie da applicare.

```
<section name="Main Track">
  <attnum name="width" unit="m" val="10.0" />
  <attnum name="profil steps length" unit="m" val="4.0" />
  <attstr name="surface" val="asphalt-road" />

  <section name="Left Side">
    <attnum name="start width" unit="m" val="4.0" />
    <attnum name="end width" unit="m" val="4.0" />
    <attstr name="surface" val="side-terrain-snow1" />
    <attstr name="type" in="tangent, level" val="tangent" />
    <attnum name="width" val="0.1" />
  </section>
  <section name="Left Border">
    <attnum name="width" unit="m" val="2.0" />
    <attnum name="height" unit="m" val="0.05" />
    <attstr name="surface" val="border-dt4-border01" />
```

```
    <attstr name="style" val="plan" />
  </section>
  <section name="Left Barrier">
    <attnum name="width" unit="m" val="0.0" />
    <attnum name="height" unit="m" val="1.0" />
    <attstr name="surface" val="barrier-wall-1" />
    <attstr name="style" val="fence" />
  </section>
  <section name="Right Side">
  ...
  </section>
  <section name="Right Border">
  ...
  </section>
  <section name="Right Barrier">
  ...
  </section>

  <section name="Track Segments">
    <!--****************************-->
    <!--      Segment 1             -->
    <!--****************************-->
    <section name="straight 1">
      <attstr name="type" val="str" />
      <attnum name="lg" unit="m" val="90.0" />
      <attnum name="z start" unit="m" val="0.0" />
      <attnum name="z end" unit="m" val="0.0" />
      <attnum name="profil start tangent" unit="%" val="0" />
      <attnum name="profil end tangent" unit="%" val="0" />
    </section>
    ...
  </section>
  <section name="Pits">
    <attstr name="side" val="right" />
    <attstr name="entry" val="straight 37" />
    <attstr name="start" val="straight 38" />
    <attstr name="end" val="straight 1" />
    <attstr name="exit" val="straight 2" />
```

```
    <attnum name="length" unit="m" val="15.0" />
    <attnum name="width" unit="m" val="5.0" />
  </section>
</section>
```

C.1.6 Cameras

Permette di modificare la posizione e il comportamento delle telecamere fisse.

```
<section name="Cameras">
    <section name="cam 0">
      <attstr name="segment" val="straight 2" />
      <attnum name="to right" val="15" />
      <attnum name="to start" val="0" />
      <attnum name="height" val="4" />
      <attstr name="fov start" val="straight 37" />
      <attstr name="fov end" val="curve 4" />
    </section>
  ...
</section>
```

C.2 Parametri

L'utilizzo del programma prevede la possibilità di modificare alcuni parametri relativamente alla generazione di un ambientazione, questi sono prelevati direttamente da file:

C.2.1 Categorie circuiti

Le categorie sono le stesse previste da TORCS.

- dirt;
- oval;
- road.

C.2.2 Path file testuale

In ordine troviamo:

- percorso di torcs, data files, cioè dove sono salvati i tracciati;
- categoria del circuito;
- nome del circuito;
- percorso di torcs, bin files, cioè l'eseguibile;
- tipologia di generazione 4.3;
- percorso ambientazione, tipicamente ./env/[ambientazione], dove tra quadre uno degli scenari previsti 4.4;

C.2.3 Pattern, file xml

È il file che contiene informazioni relative ad una specifica ambientazione:

```
<section name="Pattern">
  <attnum name="minVar" val="3.0" />
  <attnum name="maxVar" val="1.5" />
  <attnum name="minNeed" val="-20.0" />
  <attnum name="maxNeed" val="40.0" />
  <attnum name="startRange" val="-10.0" />
  <attnum name="endRange" val="+30.0" />

  <!-- Asphalt-->
  <attnum name="friction" val="1.2" />
  <attnum name="roughness" val="0.0" />
  <attnum name="roughnessWavelength" val="1.0" />
  <attnum name="rollingRes" val="0.001" />

  <!-- Side -->
  <attnum name="sideFriction" val="0.6" />
  <attnum name="sideRoughness" val="0.03" />
  <attnum name="sideRoughnessWavelength" val="4.0" />
  <attnum name="sideRollingRes" val="0.03" />
```

```
  <!-- Border -->
  <attnum name="borderFriction" val="1.1" />
  <attnum name="borderRoughness" val="0.0" />
  <attnum name="borderRoughnessWavelength" val="1.0" />
  <attnum name="borderRollingRes" val="0.001" />

  <attnum name="damage" val="2" />

  <!-- NOISE -->
  <attnum name="speed" val="3" />
  <attnum name="octave" val="4" />
  <attnum name="size" val="1.0" />
</section>
```

Parametri del rumore

I parametri che permettono di ottenere varie tipologie di rumore sono i seguenti:

- speed: indica il grado di irregolarità del terreno;
- octave: rappresenta il numero di iterazione per raggiungere il rumore finale;
- size: cioè il grado di luminosità del rumore generato.

Purtroppo una loro spiegazione chiara sugli effetti non è semplice, quindi un metodo valido per sceglierli è effettuare vari tentativi per vedere quale meglio si adatta ad un'ambientazione specifica, per comprenderli al meglio conviene basarsi su uno dei tanti file pattern.xml d'esempio previsti per ogni ambientazione.

Parametri superfici

Per ogni superficie sono previsti i seguenti parametri che specificano una particolare grandezza fisica:

- friction: è l'attrito posseduto dal terreno, tipicamente compreso tra 0.5 e 1.5, un valore maggiore indica migliore aderenza;
- roughness: indica la rugosità della superficie, cioè la presenza di microimperfezioni geometriche superficiali che quindi riducono l'aderenza, tipicamente ha ordine di grandezza pari a 10^{-2};
- rolling resistance: cioè l'attrito volvente della superficie in questione, un valore maggio determina una minore stabilità del veicolo, tipicamente con ordine di grandezza pari a 10^{-3}.

Parametri altitudine

- minVar: rappresenta la percentuale di variazione dell'altitudine inferiore;
- maxVar: indica la percentuale di variazione dell'altitudine superiore;
- minNeed: sommato alla quota inferiore determina il valore massimo in metri da associare a quest'ultima, quindi tipicamente è un valore negativo;
- maxNeed: sommato alla quota superiore determina il valore minimo in metri che quest'ultima dovrà assumere;
- startRange: il primo valore dell'intervallo entro cui applicare il rumore;
- endRange: l'estremo superiore dell'intervallo di applicazione del rumore.

C.3 Vincoli sui formati

Dato che utilizziamo strumenti già esistenti dobbiamo attenerci ai loro vincoli sui i formati dei file.

Le texture, sia quelle delle superfici che quelle relative agli oggetti, devono essere nel formato *IRIS Silicon Graphics (SGI)* spesso lo troviamo con estensione *.rgb* o *.sgi*, in linea teorica anche il formato *Portable Network Graphics*

(PNG) dovrebbe essere supportato, ma in alcuni casi causa dei problemi quindi non garantiamo il corretto funzionamento utilizzando quest'ultimo formato.

Le ulteriori immagini utilizzate, quindi anche gli sfondi , invece, possono essere in entrambi i formati sopraccitati.

Gli oggetti 3D invece devono avere formato *AC3D* con estensione tipica *.ac* [33].

C.4 Esecuzione

Una volta scelti i parametri si può procedere a lanciare il programma con il seguente comando:

java -jar -Xmx512m TCG[version].jar [path.txt] {*pattern.xml*}

Spiegando in breve: *java* indica l'utilizzo della JVM a cui passiamo un file .jar eseguibile *-jar* e che forziamo ad eseguire con una memoria massima disponibile per lo heap di 512MB con l'argomento *Xmx512m*, è necessario aumentare la quantità di memoria massima (quella di default è pari a 64MB) poiché altrimenti il programma terminerebbe con un'eccezione per mancanza di memoria.

Poi va indicato il nome del programma, che è caratterizzato da una versione utile per eventuali aggiornamenti futuri.

Il primo parametro effettivo del programma, cioè *path.txt* è il file mostrato nei primi paragrafi di questo capitolo C.2.2.

L'ultimo parametro, quello tra parentesi graffe, invece, è opzionale e si riferisce al file *pattern.xml* citato anch'esso in questo capitolo C.2.3, se non viene fornito il programma sceglierà di default quello presente nella cartella specifica del tipo di ambientazione.