

**Università di Padova**

DIPARTIMENTO DI MATEMATICA “TULLIO LEVI-CIVITA”

CORSO DI LAUREA IN INFORMATICA



**«Shots Up!»:**

**Sviluppo di un Videogioco Web 3D**

Tesi di Laurea

Laureando

Visentin Giovanni

Matricola 2101064

Relatore

Prof. Vardanega Tullio

---

ANNO ACCADEMICO 2025-2026



Al mio tre di denari.

# Ringraziamenti

Per primi, voglio ringraziare i miei genitori e la mia famiglia: mio padre, mia madre, mio fratello e mia sorella, per essere stati presenti, sempre e comunque, e per avermi dato fiducia e forza per continuare.

Voglio ringraziare Tiberio e Marco, per essere sempre stati vicini a sostenermi, per tutti i momenti, per tutte le risate e tutte le giornate che abbiamo vissuto e trascorso assieme.

Voglio poi ringraziare tutti gli altri miei amici, senza i quali non avrei passato questi anni così stupendi della mia vita, che porterò per sempre nel mio cuore, e senza i quali non avrei giocato così tante volte a carte.

Un ringraziamento va anche al professor Vardanega per avermi stimolato a impegnarmi e per la sua disponibilità dimostrata durante il percorso di *stage* e di tesi.

Grazie poi alla regia, ai fonici, ai *cameraman*, allo *staff*, al trucco e al parrucco, ma soprattutto grazie a voi, perché senza di voi difficilmente sarei qua.

Padova, Settembre 2026

*Visentin Giovanni*

# Indice

|          |                                                                   |           |
|----------|-------------------------------------------------------------------|-----------|
| <b>1</b> | <b>Sync Lab: contesto e processi</b>                              | <b>1</b>  |
| 1.1      | Come ho raccolto le informazioni di questo capitolo . . . . .     | 1         |
| 1.2      | L'azienda e la sua articolazione territoriale . . . . .           | 3         |
| 1.3      | Prodotti, servizi, tecnologie e tipologia di clientela . . . . .  | 4         |
| 1.4      | Processi interni di sviluppo e manutenzione . . . . .             | 5         |
| 1.5      | Il rapporto dell'azienda con l'innovazione . . . . .              | 8         |
| <b>2</b> | <b>Esplorare la grafica 3D nel <i>browser</i></b>                 | <b>10</b> |
| 2.1      | Gli <i>stage</i> come pratica ricorrente in Sync Lab . . . . .    | 10        |
| 2.2      | Il problema: un banco di prova formativo sulle tecnologie di casa | 12        |
| 2.3      | Obiettivi e vincoli . . . . .                                     | 14        |
| 2.4      | La pianificazione concordata . . . . .                            | 17        |
| 2.5      | Perché ho proposto questo progetto . . . . .                      | 19        |
| <b>3</b> | <b>Il progetto «Shots Up!»</b>                                    | <b>21</b> |
| 3.1      | Il metodo di lavoro . . . . .                                     | 21        |
| 3.2      | Analisi: dagli obiettivi ai requisiti . . . . .                   | 23        |
| 3.3      | Progettazione . . . . .                                           | 26        |
| 3.3.1    | Architettura del sistema . . . . .                                | 26        |
| 3.3.2    | I vincoli architetturali adottati . . . . .                       | 30        |
| 3.4      | Codifica: tre problemi degni di attenzione . . . . .              | 33        |
| 3.4.1    | La meccanica del tiro . . . . .                                   | 33        |
| 3.4.2    | Mappa di densità e stima locale . . . . .                         | 42        |
| 3.4.3    | Rendere pubblicabile il prodotto . . . . .                        | 49        |
| 3.5      | Verifica e validazione . . . . .                                  | 56        |

|          |                                                                   |           |
|----------|-------------------------------------------------------------------|-----------|
| 3.6      | I risultati . . . . .                                             | 60        |
| 3.6.1    | Il prodotto realizzato . . . . .                                  | 60        |
| 3.6.2    | Il prodotto in cifre . . . . .                                    | 63        |
| <b>4</b> | <b>Valutazione retrospettiva</b>                                  | <b>68</b> |
| 4.1      | Grado di raggiungimento degli obiettivi . . . . .                 | 68        |
| 4.2      | Consuntivo delle ore e scostamenti dalla pianificazione . . . . . | 72        |
| 4.3      | Conoscenze e abilità acquisite . . . . .                          | 75        |
| 4.4      | Distanza fra le competenze richieste e quelle erogate dal corso . | 78        |
|          | <b>Acronimi e abbreviazioni</b>                                   | <b>81</b> |
|          | <b>Glossario</b>                                                  | <b>82</b> |
|          | <b>Bibliografia</b>                                               | <b>86</b> |
|          | <b>Sitografia</b>                                                 | <b>87</b> |

# Elenco delle tabelle

|     |                                                                       |    |
|-----|-----------------------------------------------------------------------|----|
| 1.1 | Le tre fonti su cui poggia il capitolo . . . . .                      | 2  |
| 2.1 | Gli otto obiettivi e l'interesse a cui rispondono . . . . .           | 15 |
| 2.2 | La ripartizione preventiva delle 320 ore . . . . .                    | 17 |
| 3.1 | Dagli otto obiettivi ai requisiti . . . . .                           | 26 |
| 3.2 | L'interfaccia del servizio, per intero . . . . .                      | 30 |
| 3.3 | I tre strati del personaggio e ciò che ciascuno possiede . . . . .    | 34 |
| 3.4 | I nomi che un tiro assume lungo il percorso . . . . .                 | 45 |
| 3.5 | Il peso con cui un tiro concorre a stimarne un altro . . . . .        | 48 |
| 3.6 | Le quattro varianti di importazione del motore grafico, pesate .      | 52 |
| 3.7 | Le tre prove con cui accerto che il rilevatore sappia fallire . . . . | 59 |
| 3.8 | Il prodotto prima e dopo l'intervento sul peso . . . . .              | 67 |
| 4.1 | Lo stato degli otto obiettivi e l'evidenza che lo sostiene . . . . .  | 69 |
| 4.2 | Il preventivo delle ore a confronto con la ricostruzione . . . . .    | 73 |

## Elenco delle figure

|     |                                                                        |    |
|-----|------------------------------------------------------------------------|----|
| 1.1 | Le cinque sedi di Sync Lab . . . . .                                   | 3  |
| 1.2 | Le tecnologie di casa e quelle del progetto . . . . .                  | 5  |
| 1.3 | I due percorsi: la commessa riferita, il tirocinio documentato . .     | 6  |
| 2.1 | Le otto settimane del Piano di Lavoro . . . . .                        | 18 |
| 3.1 | I tre livelli e ciò che attraversa i loro confini . . . . .            | 28 |
| 3.2 | I moduli del gioco e il passaggio obbligato verso il motore . . . .    | 29 |
| 3.3 | L'angolo di lancio come bisettrice, e l'intervallo che lo vincola .    | 37 |
| 3.4 | I quattro stati della palla e il passaggio fra i due regimi . . . . .  | 40 |
| 3.5 | Il menu principale, sovrapposto alla scena già in movimento . .        | 61 |
| 3.6 | L'istante del tiro, con l'arco della traiettoria disegnato in anticipo | 62 |
| 3.7 | La mappa di densità dei tiri, con i due gruppi di filtri . . . . .     | 63 |

## Elenco dei codici sorgenti

|     |                                                             |    |
|-----|-------------------------------------------------------------|----|
| 3.1 | Le coordinate che il modello copia dalla capsula . . . . .  | 36 |
| 3.2 | Da dove il servizio prende l'identità di chi tira . . . . . | 44 |
| 3.3 | Lo stesso simbolo chiesto a due indirizzi diversi . . . . . | 51 |

# Capitolo 1

## Sync Lab: contesto e processi

In questo capitolo colloco lo *stage* nell'organizzazione che lo ha ospitato. Prima di descriverla dichiaro da dove provenga ciò che ne riporto, perché ho lavorato in prevalenza da remoto e ne ho quindi osservato soltanto una parte. Chiudo sul rapporto dell'azienda con le tecnologie non consolidate, che è la premessa del capitolo [2](#).

### 1.1 Come ho raccolto le informazioni di questo capitolo

Quanto segue proviene da tre fonti di natura diversa, che tengo distinte per tutto il capitolo e che la tabella [1.1](#) riassume.

La prima è il Piano di Lavoro, il documento concordato con il *tutor* aziendale e approvato prima dell'avvio del tirocinio. È una fonte scritta e verificabile, che riporta sulla propria carta intestata la forma societaria, le sedi e i recapiti dell'azienda, e nel corpo la cadenza concordata degli incontri, gli obiettivi del progetto e la ripartizione preventiva delle ore. Dove il capitolo attinge a esso, il dato è controllabile su un documento firmato dalle parti.

La seconda è la mia osservazione diretta, limitata però alle giornate trascorse presso la sede di Padova. Il tirocinio l'ho svolto in trecentoventi ore, che il piano articola in otto settimane da quaranta, in prevalenza da remoto, con una o due giornate a settimana in sede: una cadenza che avevamo concordato in anticipo

| Fonte                            | Natura                                                                     | Che cosa ne ricavo                                                                                                                                                      | Sezioni                                                                               |
|----------------------------------|----------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| Piano di Lavoro                  | Scritta, firmata dalle parti prima dell'avvio                              | Forma societaria, sedi e certificazioni; cadenza degli incontri, obiettivi, prodotti attesi, ore preventivate                                                           | <a href="#">1.2</a> , <a href="#">1.4</a>                                             |
| Osservazione diretta             | Vista di persona, nelle giornate in sede a Padova                          | Le persone presenti in sede; il quadro procedurale del tirocinio; una discussione tecnica con un collega                                                                | <a href="#">1.2</a> , <a href="#">1.4</a> , <a href="#">1.5</a>                       |
| Quanto riferito dal <i>tutor</i> | Testimonianza, citata con la sua autorizzazione. Non verificata di persona | Dimensione e specializzazione della sede; composizione dei gruppi; committenza e tecnologie; processi e manutenzione su commessa; studio delle tecnologie non impiegate | <a href="#">1.2</a> , <a href="#">1.3</a> , <a href="#">1.4</a> , <a href="#">1.5</a> |

**Tabella 1.1:** Le tre fonti del capitolo, dalla più verificabile alla meno. L'ultima colonna indica in quali sezioni impiego ciascuna.

e che rispetta la previsione del Piano di Lavoro, il quale fissava «almeno una volta la settimana» gli incontri diretti con il *tutor* aziendale. In alcune occasioni sono mancato, a volte per un mio impegno familiare, altre perché la sede era chiusa. Ogni giornata in ufficio comprendeva un passaggio con il *tutor* dedicato allo stato di avanzamento, ovvero i progressi compiuti dall'incontro precedente, gli eventuali scostamenti rispetto alla pianificazione e il modo di rientrarvi.

La terza è quanto mi ha riferito il *tutor* aziendale, in parte nel corso di quegli incontri e in parte in risposta scritta a una serie di domande che gli ho rivolto in fase di stesura, per non affidare alla mia memoria informazioni che sarebbero finite in un documento pubblico. Fra quelle domande ce n'era una sul riportarne le risposte in questa relazione, citandolo come fonte, e ha acconsentito senza riserve e senza escludere nulla di ciò che mi aveva detto.

Da ciò nasce un limite che enuncio subito. Quello che ho potuto osservare di persona riguarda l'ambiente della sede di Padova e il rapporto di lavoro con il *tutor*; non ho invece preso parte ad alcuna commessa, e non ho quindi visto all'opera i processi che descriverò nella sezione [1.4](#). Le sezioni [1.3](#), [1.4](#) e [1.5](#) riportano perciò informazioni riferite, e come tali le segnalo nel testo: dove scrivo che il *tutor* riferisce un fatto, il lettore deve intendere che non l'ho verificato di persona. Quelle sezioni le scrivo per dare al lettore un quadro dell'azienda, non



**Figura 1.1:** Le cinque sedi di Sync Lab su un profilo semplificato dell'Italia.

per documentare ciò che ho visto fare in sede.

## 1.2 L'azienda e la sua articolazione territoriale

Sync Lab è una società a responsabilità limitata con capitale sociale di centomila euro interamente versato, iscritta al registro delle imprese di Napoli. È distribuita su cinque sedi: Napoli, che è la sede legale, Roma, Milano, Padova e Verona.

Lo *stage* l'ho svolto presso la sede di Padova, in Galleria Spagna: è l'unica delle cinque che ho visto.

La figura 1.1 rende visibile quel limite: quattro sedi di cui parlo per averne letto o sentito, una sola vista di persona.

In Sync Lab lavorano all'incirca duecentocinquanta persone, e una venticinquina afferisce alla sede di Padova, mi ha riferito il *tutor*. Gli ambiti su cui quella sede è più focalizzata sono l'intelligenza artificiale, le integrazioni di sistemi e le *web application*.

I gruppi di lavoro, riferisce ancora, l'azienda li costituisce per competenza e ne decide la composizione effettiva all'occorrenza, di volta in volta, sul progetto da avviare. È un dettaglio organizzativo di cui la sezione 1.4 avrà bisogno,

perché da solo rende conto del fatto che la cadenza degli allineamenti non sia unica in tutta l'azienda.

Ciò che ho visto di persona è una porzione molto più piccola. Nelle giornate trascorse in sede ho lavorato insieme ad altri tre stagisti, quattro in tutto contando me nel periodo in cui mi trovavo lì, e al *tutor* aziendale; in altre stanze dell'ufficio, non sempre, ho incrociato altre due o tre persone.

Non tutti gli scambi che ho avuto lì riguardavano però il mio percorso di tirocinante. In una di quelle giornate ho avuto con un collega una discussione tecnica sul progetto che stavo sviluppando, stimolante e per me utile, dalla quale sono uscite osservazioni che ho poi impiegato per migliorarlo. Lo annoto qui perché è l'unico episodio del genere a cui ho assistito, e perché nella sezione [1.5](#) tornerà a servire.

### 1.3 Prodotti, servizi, tecnologie e tipologia di clientela

Di questa sezione dico subito il limite, che è il più stretto del capitolo. Non ho preso parte ad alcuna commessa e non ho visto un prodotto dell'azienda in nessuna fase della sua vita, quindi non descrivo qui né singoli prodotti né singoli clienti, dei quali non ho notizia. Riporto due cose che il *tutor* mi ha descritto e che bastano allo scopo per cui il capitolo esiste, ovvero il tipo di committenza a cui Sync Lab si rivolge e le famiglie di tecnologie con cui lavora.

La committenza è fatta di aziende *enterprise* di dimensione medio-grande, prevalentemente nazionali, alle quali negli ultimi tempi si sono aggiunti alcuni clienti internazionali. Chiamo *enterprise*, con il significato che il termine ha nel settore, un'organizzazione abbastanza grande e strutturata da avere già un proprio sistema informativo: chi commissiona non cerca un programma a sé stante, ma una parte che si innesti in ciò che ha già, ne rispetti i vincoli di sicurezza e di continuità del servizio e resti manutenibile nel tempo.

Il *tutor* nomina le tecnologie abituali per linguaggio e non per prodotto: Java, TypeScript e, in tempi recenti, molto Python.

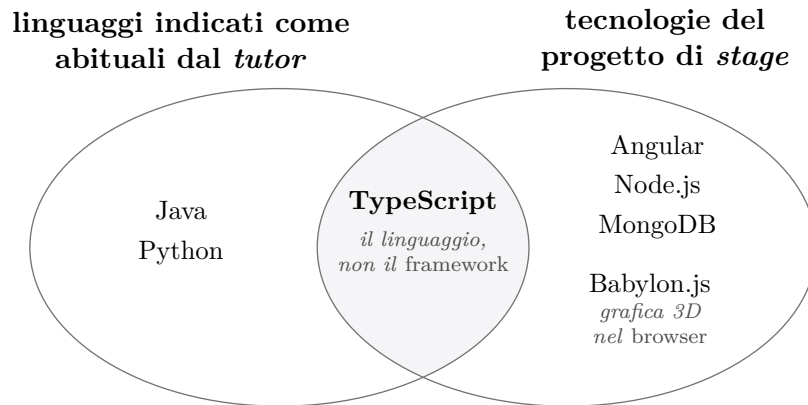


**Figura 1.2:** A sinistra i linguaggi che il *tutor* indica come abituali, a destra le tecnologie del progetto: i due insiemi si toccano in TypeScript e in nient'altro.

Da qui discende il vincolo tecnologico entro cui ho lavorato, di cui rendo conto nel capitolo 2. TypeScript è una tecnologia di casa per Sync Lab, e la scelta di Angular per l'applicazione nel *browser* ne è la ricaduta diretta. Su un punto devo però essere più preciso. Il *tutor* parla di linguaggi e non di *framework*, e non nomina Node.js, da me utilizzato; la continuità fra il mio progetto e il lavoro dell'azienda che posso affermare è quindi quella del linguaggio, non quella del singolo strumento di *back end*.

Resta fuori una cosa sola, ed è quella su cui il progetto è costruito. La grafica tridimensionale interattiva nel *browser* non compare in nessuna delle famiglie nominate, né fra gli ambiti su cui la sede di Padova è specializzata. È in questo senso, e non in un altro, che il tema dello *stage* è stato per Sync Lab un'esplorazione e non un'attività ordinaria. La figura 1.2 tiene insieme i due elenchi e il punto in cui si sovrappongono: è lì, e soltanto lì, che posso affermare una continuità fra il mio progetto e il lavoro dell'azienda.

## 1.4 Processi interni di sviluppo e manutenzione

In questa sezione riferisco due cose di natura diversa, e le tengo ben separate, come fa la figura 1.3. La prima è il modo in cui l'azienda lavora su una commessa, che non ho visto, non avendo preso parte ad alcun progetto per un cliente: quanto ne riporto me lo ha descritto il *tutor*. La seconda è il quadro procedurale entro cui ho svolto il tirocinio, che ho invece documentato per intero e che il

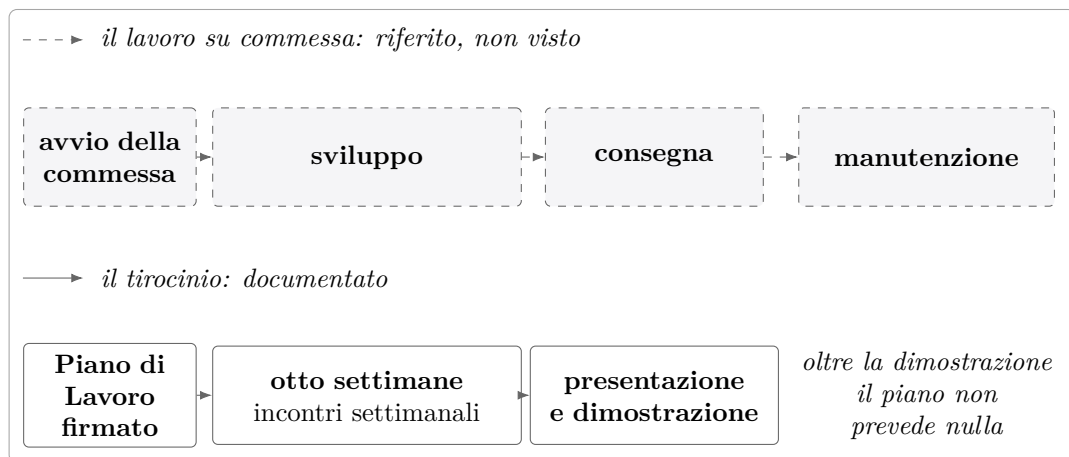


**Figura 1.3:** Sopra la commessa; sotto il tirocinio. Solo la prima prosegue oltre la consegna.

Piano di Lavoro fissa in parte per iscritto.

### Il lavoro su commessa

Del lavoro su commessa il *tutor* riferisce un metodo dichiarato, Agile con Scrum, adottato «più possibile»: la cadenza delle riunioni non è unica ma dipende dal progetto, e va da quella giornaliera a quella settimanale. Gli strumenti sono Git per il versionamento e ClickUp per il tracciamento delle attività; la revisione del codice l'azienda cerca di farla periodicamente, senza sempre riuscirci. È una variabilità che l'organizzazione descritta nella sezione 1.2 spiega da sé: gruppi costituiti per competenza, la cui composizione l'azienda decide di volta in volta sul progetto, adottano naturalmente il regime di allineamento che il progetto richiede.

A questa descrizione ne affianco una documentale, che viene dalla stessa carta intestata da cui la sezione 1.2 ricava le sedi. Vi compaiono quattro marchi di certificazione di sistema di gestione, rilasciati tutti dallo stesso ente, e due di essi riguardano quanto descrivo qui: ISO 9001 per la qualità e ISO 27001 per la sicurezza delle informazioni. Da un marchio ricavo una cosa sola, ovvero che procedure documentate esistono e che un ente terzo ne verifica periodicamente la conformità.

Sulla manutenzione dei prodotti consegnati il *tutor* riferisce due strumenti distinti: la copertura che la legge impone al fornitore e, oltre quella, contratti

di assistenza attivati con il cliente per la correzione dei difetti e per piccoli interventi evolutivi successivi. La manutenzione non è quindi un prolungamento indefinito della fornitura, ma un rapporto stipulato a parte.

### Il quadro del tirocinio

Il quadro entro cui ho svolto il tirocinio ha una parte scritta. Il Piano di Lavoro dedica un proprio paragrafo alla forma dell'interazione e vi fissa due canali distinti, l'incontro diretto, «regolarmente, almeno una volta la settimana», con il *tutor* aziendale e gli *stakeholder*, e l'interazione infrasettimanale per posta elettronica o su Discord, riservata ai dubbi e ai problemi che sorgano nel frattempo. Due dettagli di questa formulazione dicono più di quanto sembri. La cadenza settimanale vi è indicata come minimo, non come massimo, ed è uno dei due regimi che l'azienda pratica anche altrove. E fra gli scopi dell'incontro, accanto alla verifica dello stato di avanzamento e al chiarimento degli obiettivi, compare l'aggiornamento del piano stesso: è la cornice entro cui la sezione 4.2 legge lo scostamento fra ore preventivate ed effettive.

Gli incontri si appoggiavano a una bacheca Trello, che riportava gli obiettivi del periodo e il loro stato di avanzamento e che a ogni passaggio percorrevamo voce per voce. Ne discende una conseguenza pratica di cui rendo conto nella sezione 4.2: poiché il piano non prevedeva un registro delle ore per attività, la bacheca resta l'unica traccia scritta dell'andamento settimanale del lavoro.

Nel Piano di Lavoro il versionamento occupa una posizione che vale la pena osservare: l'uso di Git e la conformità alle *best practice* di programmazione non compaiono fra i mezzi con cui raggiungere il risultato, ma fra gli obiettivi da conseguire, come riporto nella sezione 2.3. Altrettanto esplicita è l'attesa documentale: dei tre prodotti che il piano si attende, uno solo è il videogioco funzionante, mentre gli altri due sono la documentazione tecnica con il manuale utente e il codice sorgente «ben strutturato e documentato». È un'attesa che la manutenzione descritta sopra rende concreta, perché un prodotto ripreso a distanza di mesi dev'essere leggibile da chi non lo ha scritto.

Il piano prevede in anticipo anche la chiusura, e le riserva ore proprie: un incontro di presentazione della piattaforma agli *stakeholder* e una dimostrazio-

ne dal vivo del lavoro svolto. La ripartizione delle ore in cui queste due voci compaiono la riporto nella sezione 2.4.

Resta da dire a che cosa questa sezione serve, e perché l'ho scritta. Il capitolo 3 valuta il metodo di lavoro che ho adottato, e per farlo ha bisogno di un secondo termine di paragone. Quel termine è il quadro che l'azienda ha posto attorno al mio *stage*, ovvero cadenza degli allineamenti, tracciamento degli obiettivi, versionamento, attesa documentale e presentazione finale, ed è su questi cinque elementi che conduco il confronto nella sezione 3.1.

## 1.5 Il rapporto dell'azienda con l'innovazione

Vista la natura sperimentale del mio progetto, ho chiesto al *tutor* se l'azienda dedichi tempo o risorse allo studio di tecnologie che ancora non impiega, e in quale forma lo faccia. La risposta è breve e la riporto per intero, perché ogni sua parte porta un'informazione. In Sync Lab vi sono «alcune persone (fortemente orientate alla tecnologia) nei vari gruppi che nei momenti di vuoto lavoro si dedicano allo *scouting*/studio delle nuove tecnologie».

Ne ricavo tre cose. La prima riguarda la forma che questo studio ha: la risposta non nomina un programma, un tempo allocato allo scopo o un incarico assegnato a qualcuno, e mette al centro delle persone. La seconda riguarda chi se ne occupa, ovvero singole persone tecnicamente orientate, appassionate, distribuite nei vari gruppi anziché raccolte in un reparto che faccia ricerca per tutti gli altri. La terza riguarda il tempo in cui lo studio avviene, ovvero i momenti di vuoto lavoro.

Merita più attenzione l'ultimo punto, perché dice qualcosa sulle persone: l'interesse per la tecnologia arriva fin dove non c'è nessuno che lo chieda. Lo *scouting* non è assegnato a nessuno e qualcuno lo fa lo stesso per passione e interesse. Ne discende una conseguenza di misura, e la riporto perché è la premessa del capitolo 2: una tecnologia lontana dal perimetro descritto nella sezione 1.3 dipende, per essere esaminata, dal tempo che quel perimetro lascia libero.

Di questa disposizione ho visto un solo esempio diretto, ovvero la discussione tecnica con un collega di cui ho riferito nella sezione [1.2](#). È un episodio singolo e non dimostra nulla di generale, ma ha la forma di ciò che il *tutor* descrive in termini generali, ovvero l'interesse tecnico di chi si occupa di qualcosa anche fuori dai compiti che ha ricevuto.

Resta però un secondo modo con cui l'azienda si avvicina a ciò che non impiega, e non dipende dal tempo che il lavoro su commessa lascia libero: il tirocinio, che dello studio di una tecnologia nuova ha la stessa funzione, ovvero metterla alla prova per vedere se e come funziona. È da qui che apro il capitolo [2](#).

## Capitolo 2

# Esplorare la grafica 3D nel *browser*

Il capitolo precedente ha descritto l'organizzazione che ha ospitato lo *stage*; in questo dico perché quello *stage* sia esistito. La domanda ha due risposte distinte, che sono le motivazioni delle due parti: quella dell'azienda che lo ha accolto e la mia, che l'ho proposto. Le fonti restano quelle dichiarate nella sezione [1.1](#), ma qui pesano in modo diverso: il Piano di Lavoro porta gli obiettivi, i vincoli e le ore, il *tutor* porta la ragione per cui l'azienda ha accettato la mia proposta, e l'ultima sezione risponde di sé, perché riguarda una scelta mia.

### 2.1 Gli *stage* come pratica ricorrente in Sync Lab

Ho chiuso la sezione [1.5](#) sul tirocinio come secondo modo con cui Sync Lab si avvicina a ciò che non impiega. Devo ora mostrare che quel modo abbia una consistenza, ovvero che i tirocini siano molti, ricorrenti, e che di ciò che producono l'azienda faccia qualcosa. Quanto segue me lo ha riferito il *tutor* nella stessa risposta scritta da cui il capitolo precedente attinge.

La sola sede di Padova porta a termine in un anno una decina di tirocini curriculari. Nelle giornate in sede lavoravo accanto ad altri tre stagisti, come ho riferito nella sezione [1.2](#), e quattro percorsi attivi insieme sono compatibili

con una decina portati a termine in un anno. Quel che ho visto non misura la pratica, ma non la smentisce; e ciò che il dato riferito dimostra non è la quantità, è la ricorrenza: gli *stage* sono in Sync Lab qualcosa che si ripete, e non un episodio isolato.

Di ciò che quei percorsi producono il *tutor* dice che «quando possibile» l'azienda ne condivide internamente il codice per studiare tecnologie nuove. La condivisione non è quindi un passaggio previsto che avvenga sempre, ma ciò che l'azienda fa quando le condizioni lo permettono. Con questa misura l'anello si chiude. Le persone descritte nella sezione 1.5, quelle che studiano tecnologie nuove nei momenti di vuoto lavoro, trovano nel materiale uscito dai tirocini qualcosa di già scritto da cui partire, che non hanno dovuto scrivere loro e che non è costato ore di commessa. È in questo senso che il tirocinio è il secondo canale: non perché sostituisca lo *scouting*, ma perché gli fornisce materiale.

La domanda che più da vicino mi riguarda è però un'altra: se accogliere un tema proposto dallo studente sia una prassi consolidata. Il *tutor* dichiara di spingere «fortemente» lo studente a proporre un tema perché, a detta sua, è molto più motivante lavorare su qualcosa di cui si conosce già il dominio applicativo.

Su questo punto devo essere preciso, e distinguo il fatto dalla sua lettura. Il fatto non cambia: il tema di questo *stage* l'ho proposto io, e delle ragioni per cui l'ho fatto rendo conto nella sezione 2.5. Cambia il modo di intenderlo. Non è una concessione ottenuta da un'azienda che aveva altro in mente, è il funzionamento previsto della pratica di tirocinio in Sync Lab, dove è l'azienda a sollecitare la proposta dello studente, e la scelta del tema resta sua. Ne ricavo però che, se non è l'azienda a scegliere il tema, il suo interesse non può stare in quello, ma in ciò che il tirocinio produce e nel modo in cui lo produce. È la premessa della sezione 2.2, dove nomino quell'interesse, e della sezione 2.3, dove il piano lo scrive per esteso in otto obiettivi.

## 2.2 Il problema: un banco di prova formativo sulle tecnologie di casa

### Una spiegazione che non regge

Di questo *stage* esiste una spiegazione che viene per prima ed è anche la più lusinghiera per chi lo ha svolto: un'azienda che non ha la grafica tridimensionale nel proprio perimetro tecnologico ne affida l'esplorazione a un tirocinante, per sapere se convenga adottarla. Il capitolo precedente sembra sostenerla da due lati. La sezione 1.3 mostra un perimetro tecnologico in cui il 3D nel *browser* non compare, e la sezione 1.5 si chiude indicando nel tirocinio il modo con cui l'azienda mette alla prova una tecnologia che non impiega: messe insieme, le due cose farebbero del mio *stage* lo strumento con cui Sync Lab ha guardato a quella che le manca. Sarebbe una conclusione gradevole, perché darebbe al lavoro un destinatario dentro l'azienda e uno scopo che va oltre la formazione di chi lo ha svolto. Tuttavia, alla domanda se esistano progetti, in corso o previsti, che facciano uso di grafica tridimensionale interattiva nel *browser*, il *tutor* risponde in una riga: «No, in questo momento non abbiamo attività di questo tipo».

Non esiste dunque un progetto a cui il risultato di questo lavoro possa servire, né uno previsto. Messa accanto alla conclusione della sezione 2.1, dove ho mostrato che il tema non lo ha scelto l'azienda, la risposta restringe il campo abbastanza da rendere precisa la domanda: che cosa un'organizzazione si attenda da un percorso di cui non ha scelto l'argomento e di cui non le serve la tecnologia.

### La condizione, e il problema che ne risulta

La risposta sta nella condizione a cui l'azienda ha subordinato l'accettazione. La proposta era mia, e l'ha accolta a patto che fosse realizzabile e che costruissi l'applicazione con Angular e il servizio di *back end* su un *framework* già in uso in azienda. Le due parti dell'intesa rispondono a esigenze diverse. Il dominio, un gioco, è ciò che doveva tenere alta la motivazione di chi lo sviluppava, ed

è la stessa ragione per cui il *tutor*, nella sezione 2.1, dichiara di spingere lo studente a proporre un tema. Le tecnologie vincolate sono invece ciò che doveva rendere l'esperienza trasferibile: un percorso condotto per intero fuori da ciò con cui l'azienda lavora avrebbe prodotto un tirocinante esperto di qualcosa che in Sync Lab non serve a nessuno, e materiale che in azienda nessuno avrebbe saputo rileggere.

Il problema che lo *stage* ha inteso affrontare è dunque un problema formativo: costruire un banco di prova su cui esercitare le tecnologie di casa, tenuto interessante da un dominio che quelle tecnologie non hanno. Nella stessa distinzione stanno i due interessi in gioco: le tecnologie rispondono a quello di Sync Lab, la grafica tridimensionale al mio, del quale rendo conto nella sezione 2.5. Sull'espressione «tecnologie di casa» vale però la cautela della sezione 1.3: Angular la condizione lo nomina per nome, ed è la ricaduta diretta di TypeScript, che il *tutor* indica fra i linguaggi abituali, mentre il *back end* lo nomina per classe e non per prodotto, e Node.js è la scelta che ho fatto io dentro quella classe. La continuità che posso affermare resta quella della figura 1.2, ovvero quella del linguaggio e non quella del singolo strumento.

### **Perché proprio un simulatore**

Detto quale problema, la domanda successiva è perché proprio un simulatore di tiro ne fosse il banco di prova, e non un pretesto. La ragione principale è che obbliga ad attraversare per intero la verticale di un'applicazione *web*: la simulazione fisica del tiro e il *rendering* della scena stanno nel *browser*, l'autenticazione e la conservazione dei tiri stanno sul servizio, e la rilettura dello storico rimette insieme le due parti, perché un dato prodotto dal motore grafico deve tornare dal servizio per essere disegnato di nuovo. Nessuno di questi pezzi può restare simulato: se l'autenticazione non tiene, i tiri di un utente compaiono nella mappa di un altro, e chi guarda lo schermo se ne accorge.

La seconda ragione riguarda il rischio, ed è quella per cui la distanza fra il dominio del gioco e quello delle commesse è un pregio e non un difetto. Un tirocinio che sbaglia su un tema di lavoro reale sbaglia dove c'è un cliente, mentre

qui l'esito peggiore possibile era un gioco che non funzionava. Con questo ho nominato il problema; la sezione che segue ne è la forma scritta e verificabile.

## 2.3 Obiettivi e vincoli

Gli otto obiettivi del Piano di Lavoro non stanno tutti sullo stesso piano: il documento li distribuisce in tre gruppi e li identifica con un codice, quattro obbligatori da O01 a O04, due desiderabili, D01 e D02, e due facoltativi, F01 e F02. La tabella 2.1 li riporta nella codifica originale, che uso invariata per tutto il resto della relazione.

L'assegnazione di un obiettivo a un gruppo o a un altro non è una convenzione di scrittura del piano, ma un atto di progettazione: la scala fra obbligatorio, desiderabile e facoltativo è il risultato del vaglio di fattibilità che ho condotto con il *tutor* sulla proposta iniziale, e distribuisce in anticipo il rischio delle 320 ore, che sono la sola risorsa in gioco e non sono elastiche. In quel senso è la prima decisione di progetto dello *stage*: l'abbiamo presa prima che scrivessi una riga di codice, e stabilisce che cosa debba esistere perché il lavoro possa dirsi fatto, che cosa valga la pena di avere ma possa cadere senza che manchi, e che cosa sia dichiaratamente sacrificabile.

L'ultima colonna della tabella è la sola cosa che essa aggiunga al testo, e la leggo per intero perché mostra una regolarità. Fra gli obbligatori, tre su quattro rispondono all'interesse dell'azienda, e sono le tre forme in cui il lavoro diventa per lei riutilizzabile: la disciplina di scrittura e di versionamento del codice (O02), la documentazione tecnica e il manuale (O03), il servizio di *back end* che il vincolo tecnologico obbliga a costruire con ciò che in azienda è già in uso (O04). O03 in particolare è ciò che rende praticabile il riuso descritto nella sezione 2.1, perché di codice condiviso internamente per studiare tecnologie nuove nessuno fa molto, se non lo accompagna qualcosa che spieghi come è fatto. I due facoltativi rispondono invece per intero al mio interesse, dato che riguardano i modelli tridimensionali e la resa visiva dell'ambiente. I desiderabili si dividono uno per parte: D01 è la mappa di densità che mostra da dove il giocatore ha tirato e

| Codice              | Obiettivo                                                                                                                                 | Interesse    |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| <b>Obbligatori</b>  |                                                                                                                                           |              |
| O01                 | Funzionamento di base del videogioco: tiro a canestro, movimento del personaggio, calcolo del punteggio                                   | Mio          |
| O02                 | Codice pulito, ben strutturato e documentato, secondo le <i>best practice</i> di programmazione, con Git per il versionamento             | Dell'azienda |
| O03                 | Documentazione tecnica e manuale utente: architettura del sistema, scelte progettuali, istruzioni d'uso                                   | Dell'azienda |
| O04                 | Sistema di <i>back end</i> che consenta di salvare e gestire i dati degli utenti                                                          | Dell'azienda |
| <b>Desiderabili</b> |                                                                                                                                           |              |
| D01                 | Modulo di visualizzazione dei dati: <i>heatmap</i> interattiva dei tiri riusciti, in vista dall'alto, con filtro per intervallo temporale | Mio          |
| D02                 | Algoritmo di stima della probabilità di successo di ogni tiro, a partire dalla posizione e dai parametri di lancio                        | Dell'azienda |
| <b>Facoltativi</b>  |                                                                                                                                           |              |
| F01                 | Modelli tridimensionali e animazioni di base per il personaggio e per l'ambiente di gioco, con strumenti di modellazione come Blender     | Mio          |
| F02                 | Personalizzazione estetica dell'ambiente: luci, materiali, ciclo giorno/notte                                                             | Mio          |

**Tabella 2.1:** Gli otto obiettivi del Piano di Lavoro, nella codifica originale del documento e con il testo condensato. L'ultima colonna indica a quale dei due interessi in gioco ciascun obiettivo risponda: per O02 e D02 l'attribuzione è del *tutor*, per gli altri sei è mia.

con quale esito, ed è ancora resa grafica; D02 è un problema di stima, e su di esso l'azienda ha posto un'attesa esplicita.

Fa eccezione O01, che è obbligatorio ma risponde al mio interesse: è il funzionamento del gioco, cioè la parte che l'azienda non ha scelto e da cui non ricava nulla di direttamente riutilizzabile. L'eccezione è però apparente, e spiega la scala meglio di quanto farebbe una regola senza strappi. Senza un gioco che funziona non c'è nulla su cui esercitare gli altri tre obbligatori, perché il codice da tenere pulito, la documentazione da scrivere e i dati da conservare sono il

codice, la documentazione e i dati di quel gioco. O01 è obbligatorio in quanto condizione degli altri tre, non perché un videogioco interessasse a Sync Lab.

Su due delle otto righe l'attribuzione non è una mia lettura ma una dichiarazione, ed è la ragione per cui ho chiesto al *tutor* quali obiettivi contassero di più per l'azienda. I più importanti da raggiungere, risponde, erano O02 e, insieme a esso, D02, la stima della probabilità di successo del tiro, «per sviluppare il ragionamento “algoritmico” del laureando»; dei due facoltativi scrive che interessavano molto meno, e che «sono tali proprio per questo». Delle altre sei righe rispondo io, e le do per quello che sono, cioè una lettura.

Dovevo perseguire gli obiettivi entro tre vincoli, che il piano fissa insieme a loro. Il primo è tecnologico ed è la condizione a cui l'azienda ha subordinato l'accettazione della proposta: Angular per l'applicazione, e per il servizio un *framework* di *back end* già in uso in azienda. Ne ho reso conto nella sezione 2.2 e non ci torno; qui lo registro fra i vincoli, perché è quello che ha determinato la forma del prodotto più di ogni altro. Il secondo è di tempo: 320 ore su otto settimane da quaranta, ripartite fra le attività nel modo di cui dà conto la sezione 2.4. Il terzo è l'esecuzione nel *browser*, e sulla sua provenienza devo essere preciso, perché non l'ha posto l'azienda: stava già nella proposta che ho presentato. Era però una scelta anch'essa, e di fattibilità, come dico nella sezione 2.5. È nondimeno un vincolo, e non dei più leggeri, dato che fissa il motore grafico a quello che il *browser* mette a disposizione e obbliga ogni calcolo del gioco a stare nel tempo di un fotogramma.

Riporto nella relazione i codici del piano, invece di lasciarli nel documento che li ha fissati, perché sono il filo a cui si legano le due parti che devono rispondere di qualcosa: la sezione 3.2 mostra come da ciascuno di essi abbia derivato i requisiti, e il capitolo 4 li riprende uno per uno per dire a che punto il lavoro sia arrivato. Senza una codifica fissata prima, quella verifica non potrebbe essere che un giudizio a posteriori su obiettivi ricostruiti a posteriori. Con i codici viaggia il peso dei gruppi, che la risposta del *tutor* ha messo per iscritto prima dell'avvio: la sezione 4.1 è per questo un confronto con attese dichiarate e non presunte.

## 2.4 La pianificazione concordata

Il Piano di Lavoro non fissa soltanto che cosa dovessi fare, ma anche quanto dovesse costare, e ripartisce in anticipo due volte le 320 ore concordate. La prima ripartizione è per calendario e assegna a ciascuna delle otto settimane da quaranta ore un tema di lavoro; la seconda è per tipo di attività e distribuisce lo stesso totale fra formazione, analisi e progettazione, sviluppo e collaudo. È la seconda quella che porta un preventivo confrontabile, e la tabella 2.2 la riporta con l'articolazione interna che il piano le dà; della prima, che serve più avanti, dà conto la figura 2.1.

| Ore | Attività                                                                      | Quota       |
|-----|-------------------------------------------------------------------------------|-------------|
| 44  | <b>Formazione sulle tecnologie</b>                                            | 13,75%      |
| 38  | <b>Definizione dell'architettura di riferimento e relativa documentazione</b> | 11,875%     |
| 12  | Analisi del problema e del dominio applicativo                                |             |
| 22  | Progettazione della piattaforma e relativi <i>test</i>                        |             |
| 4   | Stesura della documentazione di analisi e progettazione                       |             |
| 200 | <b>Sviluppo e integrazione</b>                                                | 62,5%       |
| 80  | Sviluppo delle interfacce di <i>front end</i> e del modulo tridimensionale    |             |
| 60  | Sviluppo dei servizi di <i>back end</i>                                       |             |
| 60  | Integrazione dei sistemi                                                      |             |
| 38  | <b>Collaudo finale</b>                                                        | 11,875%     |
| 30  | Collaudo                                                                      |             |
| 5   | Stesura della documentazione finale                                           |             |
| 1   | Incontro di presentazione con gli <i>stakeholder</i>                          |             |
| 2   | Dimostrazione dal vivo del lavoro svolto                                      |             |
| 320 | <b>Totale</b>                                                                 | <b>100%</b> |

**Tabella 2.2:** La ripartizione preventiva delle 320 ore fissata dal Piano di Lavoro. Le quattro voci in evidenza sono quelle del documento, con le rispettive voci di dettaglio rientrate sotto ciascuna; l'ultima colonna, che nel piano non compare, è la quota di ciascuna voce sul totale.

Le quattro voci non pesano allo stesso modo. Cinque ottavi esatti delle ore stanno in una sola, quella dello sviluppo e dell'integrazione, mentre le altre tre si dividono il rimanente in parti quasi uguali, con la formazione appena sopra

| 1                    | 2                         | 3                        | 4                  | 5                        | 6                       | 7                            | 8           |
|----------------------|---------------------------|--------------------------|--------------------|--------------------------|-------------------------|------------------------------|-------------|
| Analisi e formazione | Sviluppo <i>front end</i> | Sviluppo <i>back end</i> | Sviluppo modulo 3D | Completamento interfacce | Integrazione di sistema | <i>Testing</i> e affinamento | Conclusione |
| Angular              | Angular                   | Node.js                  | Babylon.js         |                          |                         |                              |             |

**Figura 2.1:** La ripartizione per calendario: le otto settimane da quaranta ore, con il titolo che il Piano di Lavoro dà a ciascuna. In evidenza le quattro in cui il piano colloca uno studio, con la tecnologia studiata. Il totale è lo stesso della tabella 2.2, ripartito per tempo anziché per attività.

le altre due. Analisi, progettazione e relativa documentazione insieme valgono trentotto ore, tante quante il collaudo finale, cioè poco meno di una settimana per parte; alla formazione ne vanno quarantaquattro, poco più di una settimana. Il piano concentra dunque il proprio impegno sulla costruzione e tiene le altre tre fasi su un ordine di grandezza comune, che è quello della singola settimana di lavoro.

Dentro le duecento ore dello sviluppo la ripartizione è in tre parti, e la terza è quella che dice di più. Ottanta ore vanno alle interfacce e al modulo tridimensionale, sessanta ai servizi di *back end* e altre sessanta all'integrazione dei sistemi, ossia al solo mettere insieme due parti già scritte. Il piano assegna quindi al raccordo tante ore quante ne assegna alla costruzione dell'intero servizio. È la verticale descritta nella sezione 2.2, letta però dal lato del costo: il punto in cui il *browser* e il servizio si parlano è preventivato come un lavoro a sé, e non come la coda di quelli che lo precedono.

I due estremi della tabella meritano una nota ciascuno, perché nessuno dei due è quello che il suo titolo lascerebbe supporre. Le quarantaquattro ore di formazione non sono un blocco iniziale, e a dirlo è l'altra ripartizione, quella per calendario, che la figura 2.1 riporta per intero: lo studio ricorre in quattro settimane su otto, e ogni volta il piano lo affianca alla tecnologia che quella settimana introduce: Angular nella prima e nella seconda, Node.js nella terza, Babylon.js nella quarta. Le trentotto ore del collaudo finale, dall'altro capo, non sono tutte collaudo: trenta lo sono, cinque vanno alla stesura della documentazione finale e le ultime tre alla presentazione del lavoro, un'ora di incontro con gli *stakeholder* e due di dimostrazione dal vivo. Sono le sole ore del preventivo

destinate a mostrare il prodotto a qualcuno anziché a costruirlo.

Resta da dire come questa ripartizione vada letta, perché la sua grana non è quella degli obiettivi della sezione 2.3 e i due elenchi non si sovrappongono riga per riga. Il piano ripartisce per fasi, e due punti in particolare lo mostrano. Il primo è che il modulo tridimensionale non ha una voce propria e condivide le ottanta ore con le interfacce, sicché la parte tecnicamente più incerta del lavoro non ha un preventivo isolabile. Il secondo è che la stesura della documentazione compare due volte e in due voci diverse, quattro ore per quella di analisi e progettazione e cinque per quella finale: O03, che la sezione 2.3 indica come ciò che rende riutilizzabile il resto, non ha quindi un unico costo dichiarato a cui commisurarlo.

Un preventivo, però, vale per il confronto che rende possibile, ed è quello che la sezione 4.2 conduce fra le quattro voci di questa tabella e la spesa effettiva. Che di quest'ultima io dia una ricostruzione, e con quale approssimazione, lo dico lì. Qui basta osservare che il termine di confronto esiste, che è dettagliato fino al secondo livello e che il piano lo ha fissato prima che il lavoro cominciasse: è questa la condizione che rende la valutazione finale un esercizio su dati e non su impressioni.

### 2.5 Perché ho proposto questo progetto

Le proposte di tirocinio già disponibili le ho scorse prima di formularne una mia: erano sia di Sync Lab sia di altre aziende, e portavano ciascuna un tema già delimitato. Nessuna copriva quello che volevo imparare, ed è per questa ragione, e non per insofferenza verso un argomento assegnato, che ho preferito scriverne una. La differenza fra le due strade non riguarda però soltanto l'argomento: di un tema ricevuto avrei risposto per la parte che mi fossi visto affidare, mentre di uno che ho proposto io rispondo per intero, dall'analisi alla verifica, senza una specifica che qualcun altro abbia già scritto.

Quello che volevo imparare aveva al centro un videogioco, e su questo voglio essere netto: il gioco non era il pretesto per arrivare a una tecnologia, era la cosa che volevo costruire, e le tecnologie, l'architettura e il servizio sono venuti dopo

di esso. Le ragioni erano tre. La prima è un interesse personale che precede di molto lo *stage*, e che aveva già una forma. La seconda è che un gioco obbliga ad attraversare competenze che il corso di studi non tocca, in particolare la grafica tridimensionale in tempo reale e la simulazione fisica. La terza è la meno tecnica e forse la più importante: il settore videoludico e la grafica tridimensionale sono ambiti che potrei voler esplorare più avanti nel mio percorso, e un tirocinio è il modo meno costoso di scoprire se quel mondo mi interessi davvero prima di sceglierlo.

L'idea però non era nata nella forma in cui l'ho proposta. Il gioco che avevo in mente era da *desktop*, e l'esecuzione nel *browser* è subentrata perché l'idea fosse proponibile a un'azienda e realizzabile una volta accettata. Un gioco da *desktop* non avrebbe avuto quasi nulla in comune né con l'azienda a cui stavo per proporlo né con ciò che il corso mi aveva dato: sarei partito da strumenti che non conoscevo e che non compaiono fra quelli che il *tutor* indica come abituali in azienda. Sul *web*, invece, esistevano tecnologie capaci di fare da ponte fra tre cose diverse, quelle di casa in azienda, quelle del percorso di studi e la grafica tridimensionale che volevo imparare. È in questo senso che l'esecuzione nel *browser* è una scelta e non un vincolo ricevuto, come anticipato nella sezione 2.3, ed è anche il punto in cui l'ecosistema del *web* moderno è entrato fra le competenze che contavo di acquisire. La condizione che l'azienda avrebbe poi posto, di cui rende conto la sezione 2.2, è caduta dentro questa scelta e non sopra di essa: Angular e un *framework* di *back end* già in casa sono tecnologie del *web*, cioè del terreno che avevo scelto perché fosse comune.

Accanto alle mie metto infine la ragione del *tutor*, che nella sezione 2.1 dichiara di spingere «fortemente» lo studente a proporre un tema perché si lavora meglio su un dominio già noto: porta allo stesso esito, ma non ha la stessa origine, dato che riguarda la motivazione di chi sviluppa e non ciò che chi sviluppa vuole imparare. È una distinzione da tenere ferma fino in fondo, perché il capitolo 4 deve tornare qui: gli otto obiettivi del piano non rispondono a un interesse solo, come mostra la tabella 2.1, e la sezione 4.1 deve pesare anche quelli che rispondono al mio.

## Capitolo 3

# Il progetto «Shots Up!»

### 3.1 Il metodo di lavoro

Di che cosa ho costruito rende conto il resto del capitolo; qui dico come l’ho condotto, e mi tengo a una regola per non scriverne un diario: di ogni pratica dico perché esisteva e che cosa mi ha evitato, non in quali giornate l’ho esercitata. Il termine di paragone è il quadro che l’azienda ha posto attorno allo *stage*, che la sezione 1.4 riassume in cinque elementi: cadenza degli allineamenti, tracciamento degli obiettivi, versionamento, attesa documentale e presentazione finale. Dei primi quattro rispondo qui; il quinto non è una pratica di lavoro ma il punto in cui il lavoro arriva davanti a qualcuno, e di ciò che lo sostiene, cioè il confronto fra il prodotto e i requisiti che mi ero dato, rendono conto la sezione 3.5 e il capitolo 4.

Con il *tutor* mi allineavo una volta la settimana, di persona. Ogni incontro aveva la stessa struttura, ed è la struttura a farne uno strumento anziché una consuetudine: esponevo che cosa avevo fatto, che cosa mi mancava e quali problemi fossero emersi, e su questi ultimi chiedevo consiglio su come mitigarli. La terza voce esisteva anche nelle settimane in cui restava vuota, ed è quella che conta, perché un problema che l’ordine del giorno prevede arriva prima di uno che aspetta di essere grande abbastanza da giustificare una richiesta d’aiuto. Il ruolo del *tutor* restava consultivo e circoscritto, dato che discutevamo come mitigare un problema e non che cosa dovessi fare, che il Piano di Lavoro aveva

già fissato negli obiettivi della sezione 2.3. E la cadenza settimanale non era un regime che l'azienda avesse adottato per lo stagista, ma uno dei due che la sezione 1.4 riferisce in uso sui suoi progetti.

Fra un incontro e l'altro l'avanzamento stava sulla bacheca Trello che la sezione 1.4 nomina, e ciò che conta non è lo strumento ma che cosa vi stesse sopra: era il Piano di Lavoro in forma di bacheca, gli obiettivi concordati e non altri, non un secondo piano che visse accanto al primo. Ne discende la proprietà per cui la tenevo: il tracciamento non poteva divergere dalla pianificazione, perché era la pianificazione con sopra il segno di ciò che avevo chiuso. Spuntavo un obiettivo a completamente avvenuto, e poiché la bacheca la vedeva anche il *tutor* la verifica settimanale consisteva nel percorrere insieme quel segno: non produceva un rapporto scritto, che non avrebbe avuto niente da riferire oltre a quello.

Prima di ogni intervento di qualche peso scrivevo un documento di piano, e ciascuno si apre dichiarando quale problema avesse davanti. Nessuno li approvava e nessuno li riceveva: sono una disciplina che mi ero imposto e non un processo, e sta tutta in una regola, mettere per iscritto che cosa stavo per fare, e perché così e non altrimenti, prima di farlo. Serviva a potermi correggere leggendo invece che ricordando, e a non riaprire una decisione già presa perché non ricordavo più contro quali alternative l'avessi presa. Quanti siano e quanto pesino lo dice la sezione 3.6.2. Quella disciplina me l'ero imposta io, ma è con quei documenti che rispondo di metà del requisito RV3, nel quale la sezione 3.2 traduce l'attesa documentale del piano, ovvero la documentazione dell'architettura e delle scelte.

Ogni intervento aveva poi un ramo di sviluppo suo, che entrava nel ramo principale con una richiesta di incorporazione. Il nome del ramo era esso stesso il tracciamento, perché nominava il problema che quel ramo apriva — la fisica del tiro, la mappa di densità, il peso da pubblicare —, sicché la storia del deposito dice a quale problema ciascuna modifica appartenga e non soltanto in quale ordine le abbia scritte. È anche la sola fra le mie pratiche di metodo di cui dovessi rendere conto a qualcuno: il Piano di Lavoro colloca la disciplina di versionamento fra gli obiettivi anziché fra i mezzi, come osservo nella sezione 1.4,

e da quell'obiettivo ho derivato il requisito RV2, che la sezione 3.6.2 verifica sul deposito.

L'ultima delle pratiche non è una consuetudine ma un principio, e non l'ho scelto a tavolino: misurare prima di intervenire l'ho imparato sbagliando. Avevo diagnosticato a mente la causa del peso eccessivo del prodotto, con una stima che pareva ragionevole e che la prima misura vera ha smentito. Il caso lo racconto per intero nella sezione 3.4.3; la regola che ne è rimasta è che su una grandezza misurabile non si discute a memoria, ed è il primo degli apprendimenti che riprendo nella sezione 4.3.

Resta da dire che cosa in questo metodo non c'era, perché è la domanda che il lettore si pone da sé. La revisione del mio codice da parte di altri è stata poca e non sistematica: ha riguardato l'impostazione dell'insieme più che il singolo intervento, e non aveva una cadenza propria come l'avevano gli allineamenti. Il limite non era una porta chiusa, perché il codice era a disposizione e avrei potuto portarlo a ogni incontro. Lo dichiaro perché è il punto in cui il mio metodo prevedeva meno, ed è anche un punto che i controlli della sezione 3.5 non raggiungono: quelli sorvegliano che il codice faccia ciò che intendevo, non che io l'abbia impostato bene, e sulla seconda domanda ho avuto quasi sempre il mio solo giudizio.

## 3.2 Analisi: dagli obiettivi ai requisiti

Un obiettivo dice che cosa deve esistere, non come si faccia a sapere se esiste. «Funzionamento di base del videogioco» è una frase su cui non si può litigare, ed è esattamente per questo che non serve a chiudere il lavoro: non contiene nulla che io possa mostrare a qualcuno per sostenere che è fatta. Il passaggio dagli otto obiettivi della sezione 2.3 ai requisiti è il punto in cui ho stabilito che cosa significasse «fatto» per ciascuno di essi, e ho cercato di stabilirlo in termini che potessi mostrare invece che sostenere.

La tabella 3.1 raccoglie i requisiti che ne ho ricavato, raggruppati sotto l'obiettivo da cui ciascuno discende. Li distinguo in due famiglie: i requisiti funzionali, che indico con RF, dicono che cosa il sistema fa; i requisiti di vincolo,

che indico con RV, dicono che cosa il sistema o il suo codice devono rispettare qualunque cosa facciano. Il raggruppamento è la ragione per cui la tabella è ordinata così e non per natura del requisito: la colonna di sinistra, letta dall'alto in basso, è la tracciabilità, ed è ciò su cui si appoggiano la sezione [3.6.2](#) e la sezione [4.1](#). Fanno eccezione le due righe finali, che non discendono da un obiettivo ma dai vincoli che il piano fissava insieme a essi, e che riporto qui perché hanno vincolato il prodotto quanto e più degli altri.

Tre derivazioni meritano una nota, perché in ciascuna ho deciso qualcosa che l'obiettivo non decideva. La prima riguarda O02 e O03, che non producono alcun requisito funzionale: sono obiettivi sul modo in cui il prodotto è fatto e su ciò che lo accompagna, non su ciò che fa, e da essi discendono vincoli sul codice e sui documenti. Vale la pena di segnarlo, perché sono due dei tre obiettivi che rispondono all'interesse dell'azienda, e sono anche i due che un catalogo di funzioni non registrerebbe affatto.

La seconda riguarda F02, l'unica riga in cui ho separato ciò che l'obiettivo teneva insieme. «Luci, materiali, ciclo giorno/notte» sono tre parole nella stessa frase, ma le prime due riguardano l'aspetto che l'ambiente ha, e la terza lo fa variare nel tempo: è un problema di natura diversa, perché richiede di governare una variazione e non di sceglierne un valore. Tenendoli su due righe, la sezione [4.1](#) può rispondere su ciascuno per conto suo; su una riga sola avrebbe dovuto dare una risposta unica a due domande diverse.

La terza riguarda O04, che produce due funzioni e due vincoli, e i due vincoli sono la parte interessante. «Salvare e gestire i dati degli utenti» non dice a chi i dati appartengano né chi possa vederli: quello l'ho dovuto decidere io, e RV4 e RV5 sono quella decisione. Li argomento come regole di progetto nella sezione [3.3.2](#), perché è ciò che sono.

Mi fermo a questo livello e non allego un catalogo esteso, con precondizioni, postcondizioni e casi d'uso per ciascuna voce: non lo ometto per brevità, ma perché un catalogo di quel genere serve a chi deve realizzare un sistema che non ha progettato, e qui chi progetta e chi realizza sono la stessa persona. Che i requisiti esistessero, da dove venissero e su quali il lavoro possa essere misurato sta tutto nella tabella.

| Obiettivo | Codice | Requisito                                                                                                                                    |
|-----------|--------|----------------------------------------------------------------------------------------------------------------------------------------------|
| O01       | RF1    | Il giocatore muove il personaggio sul campo e ne governa l'orientamento.                                                                     |
| O01       | RF2    | Il giocatore esegue un tiro modulandone la potenza; il sistema ne calcola la traiettoria da posizione, direzione e potenza di rilascio.      |
| O01       | RF3    | Il sistema riconosce l'esito di ogni tiro e aggiorna il punteggio della sessione.                                                            |
| O02       | RV1    | Il codice si articola in moduli con una sola responsabilità ciascuno, e il calcolo del tiro resta separato dal motore grafico.               |
| O02       | RV2    | Ogni intervento di rilievo passa per un ramo dedicato ed entra nel ramo principale con una richiesta di incorporazione.                      |
| O03       | RV3    | Il prodotto porta con sé la documentazione dell'architettura e delle scelte di progetto, e le istruzioni d'uso per chi gioca.                |
| O04       | RF4    | Un utente si registra e si autentica, e riceve un'identità che sopravvive alla sessione.                                                     |
| O04       | RF5    | Il sistema conserva ogni tiro concluso con la posizione di partenza, la potenza, l'esito e l'istante.                                        |
| O04       | RV4    | Il servizio espone i tiri solo a chi li ha effettuati, e ne ricava l'identità dalla credenziale firmata, non dalla richiesta.                |
| O04       | RV5    | L'identità di un <i>record</i> appartiene alla base di dati: l'applicazione nel <i>browser</i> non genera mai un identificativo.             |
| D01       | RF6    | L'applicazione sovrappone a una vista dall'alto del campo una mappa di densità delle posizioni da cui l'utente ha tirato.                    |
| D01       | RF7    | Il giocatore restringe a un intervallo temporale i tiri che concorrono alla mappa, senza alterare lo storico.                                |
| D02       | RF8    | Il sistema stima per ogni tiro la probabilità di segnare da quella posizione e con quella potenza, sugli altri tiri dello stesso utente.     |
| F01       | RF9    | Il personaggio, il canestro e l'ambiente hanno una rappresentazione tridimensionale; il personaggio ha le animazioni di movimento e di tiro. |
| F02       | RF10   | L'ambiente ha un'illuminazione e materiali propri che ne definiscono l'aspetto.                                                              |
| F02       | RF11   | Un ciclo giorno/notte fa variare nel tempo l'illuminazione dell'ambiente.                                                                    |

| Obiettivo | Codice | Requisito                                                                                                             |
|-----------|--------|-----------------------------------------------------------------------------------------------------------------------|
| Vincolo   | RV6    | L'applicazione si esegue in un <i>browser</i> recente, senza installazioni sulla macchina di chi gioca.               |
| Vincolo   | RV7    | L'applicazione è realizzata in Angular; il servizio su un <i>framework</i> di <i>back end</i> già in uso in Sync Lab. |

**Tabella 3.1:** I requisiti funzionali (RF) e di vincolo (RV) con l'obiettivo da cui ciascuno discende, nella codifica della tabella 2.1. Le ultime due righe non hanno un obiettivo di provenienza: discendono dai vincoli che il Piano di Lavoro fissava accanto agli obiettivi.

## 3.3 Progettazione

A quei requisiti do una forma, e la do in due passi: quale struttura ho scelto per soddisfarli e quali regole ho imposto a quella struttura perché reggesse. Le regole stanno per ultime perché si capiscono soltanto guardando la struttura su cui mordono, e quella struttura si giustifica soltanto davanti a ciò che deve reggere.

### 3.3.1 Architettura del sistema

Il sistema ha tre parti, e ciascuna gira in un posto diverso. La prima è l'applicazione che si apre nel *browser* di chi gioca, scritta in Angular, e contiene il gioco per intero: dentro di essa stanno anche il motore grafico, Babylon.js, e quello fisico, Havok. La seconda è un servizio che vive su un calcolatore remoto e risponde a richieste HTTP nello stile REST, in cui ogni richiesta nomina una risorsa e il verbo del protocollo dice che cosa farne; l'ho scritto in Node.js con Express. La terza è la base di dati che conserva quelle risorse, MongoDB.

Le chiamo livelli, e non semplicemente parti, perché stanno in fila e ciascuna si rivolge soltanto alla successiva: l'applicazione parla al servizio, il servizio parla alla base di dati, e nessuna delle due salta un passaggio. Conviene enunciarlo subito, perché è la proprietà su cui il resto della sezione si appoggia: se l'applicazione potesse rivolgersi alla base di dati, metà di ciò che segue non avrebbe argomenti.

Il gioco, per funzionare, non avrebbe avuto bisogno né del servizio né della base di dati. È un gioco a un giocatore: il campo, il personaggio, la palla e le regole del punteggio stanno tutti nella pagina che si apre, e nessuno di essi deve chiedere qualcosa a qualcun altro per andare avanti. Gli altri due livelli esistono per un requisito solo, RF5, cioè perché un tiro non svanisca insieme alla sessione. Nel momento in cui un dato deve durare più della pagina che l'ha prodotto, e deve per giunta appartenere a qualcuno, servono un luogo dove conservarlo e qualcuno che decida chi possa leggerlo.

Conservare e decidere sono però due compiti distinti, ed è il secondo la ragione per cui non li ho messi nello stesso posto. Un'applicazione che si rivolgesse direttamente alla base di dati porterebbe con sé, dentro la pagina, le credenziali per farlo, e la decisione su chi possa leggere che cosa si troverebbe dalla parte sbagliata: dalla parte di chi chiede, non da quella di chi custodisce. Il servizio esiste perché quel controllo stia dove chi lo subisce non lo può toccare. Il vincolo tecnologico della sezione 2.3 me lo imponeva comunque, ma preferisco poterlo giustificare anche senza appellarmi a esso.

I due motori, va detto, non sono un quarto livello, e nemmeno stanno accanto al primo: stanno dentro. Sono librerie che vivono nella stessa pagina e nello stesso filo di esecuzione dell'applicazione, e non hanno un processo proprio a cui addebitare qualcosa. La conseguenza non è terminologica. Ogni byte che quelle librerie pesano lo scarica il visitatore insieme all'applicazione, e ogni millisecondo che consumano lo tolgono ai sedici entro cui un fotogramma va consegnato: sono le due grandezze su cui la sezione 3.4.3 lavora per intero, e stanno entrambe a carico del primo livello.

Il primo livello ha però anche un confine interno, ed è il solo dei confini della figura 3.1 che non si vede guardando il sistema da fuori. Angular decide quando ridisegnare l'interfaccia sorvegliando le operazioni asincrone che accadono dentro la propria zona, cioè il contesto di esecuzione entro il quale il *framework* tiene traccia di ciò che potrebbe aver cambiato lo stato dell'applicazione. Il ciclo di *rendering* e tutto ciò che gli gira intorno stanno fuori da quella zona, e vi rientrano soltanto per portare all'interfaccia le poche grandezze che essa mostra. Perché ce li ho messi, e che cosa accadrebbe se non ci fossero, lo dico nella

sezione 3.3.2: qui basta sapere che quel confine non separa due livelli, ma taglia in due il primo.

Della figura mi interessano gli attraversamenti più dei riquadri, perché un'architettura si giudica da ciò che passa i suoi confini: è lì che si vede quali informazioni un livello si fida di ricevere e quali si tiene per sé. Ogni attraversamento porta perciò scritto il proprio contenuto e la propria direzione, e in due punti porta scritto anche ciò che non attraversa. Le sezioni che seguono si appoggiano a questa figura invece di ridescrivere ogni volta il contesto.

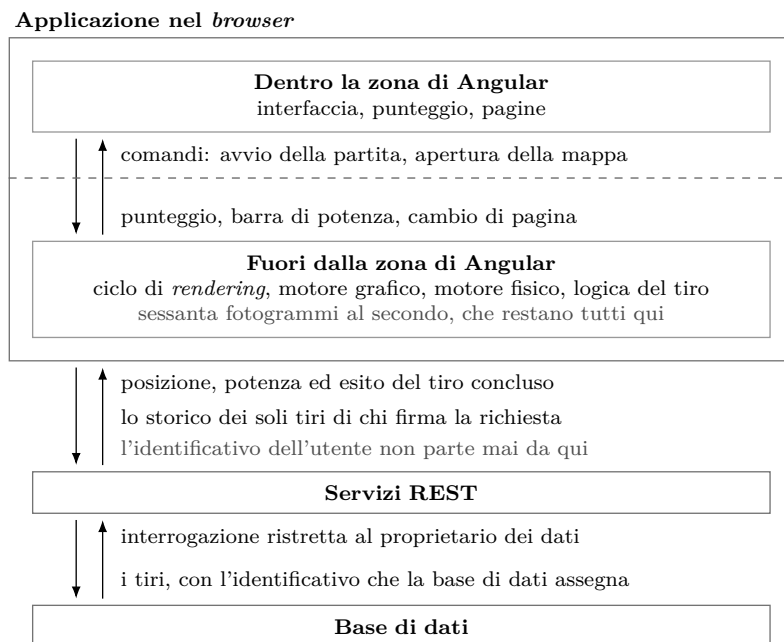


**Figura 3.1:** I tre livelli del sistema e i confini che li separano, con il contenuto di ogni attraversamento e la sua direzione. Il confine tratteggiato non separa due livelli: passa dentro il primo, e distingue ciò che Angular sorveglia da ciò che gira fuori dalla sua zona. Le due voci in grigio dicono ciò che un confine non lascia passare. Ogni richiesta verso i servizi porta con sé la credenziale firmata di chi la invia.

La figura 3.1 guarda il sistema da fuori, e da fuori il primo livello è un riquadro solo. Dentro sono trenta moduli, e il criterio con cui li ho separati conta più del numero: ho diviso per responsabilità e non per genere di oggetto, sicché un modulo costruisce una parte del mondo, un altro governa il personaggio, un altro calcola il tiro. La figura 3.2 li raggruppa così e dice quanti ne conti ciascun gruppo, perché è quel conto a mostrare dove stia la massa del codice: quasi la metà dei moduli serve a costruire il mondo in cui si gioca, che è la parte di cui,

giocando, non ci si accorge.

Il dato che la figura porta, però, è un altro, e riguarda un passaggio. Ventisei moduli hanno bisogno del motore grafico, e nessuno di essi glielo chiede: lo chiedono tutti a uno stesso modulo, che è la sola parte del gioco a nominarlo e che ne espone i soli pezzi effettivamente adoperati. Averlo reso un passaggio obbligato ha due conseguenze, e nessuna delle due è di ordine. La prima è che l'elenco di ciò che il gioco usa davvero del motore esiste in un punto e si legge; sparso su ventisei moduli non esisterebbe, e nessuno saprebbe dire quanto grande sia. La seconda è che cambiare il modo di chiedere quei pezzi è una modifica sola invece di ventisei, ed è precisamente ciò che nella sezione 3.4.3 è servito.

Tre moduli non nominano il motore in nessuna riga: i comandi di gioco, lo stato dell'interfaccia e i segnali che accompagnano la sessione. Li tengo separati per la ragione che la sezione 3.5 riprende, e che vale anche oltre questi tre: ciò che non ha bisogno del motore per funzionare non ne ha bisogno nemmeno per essere messo alla prova.

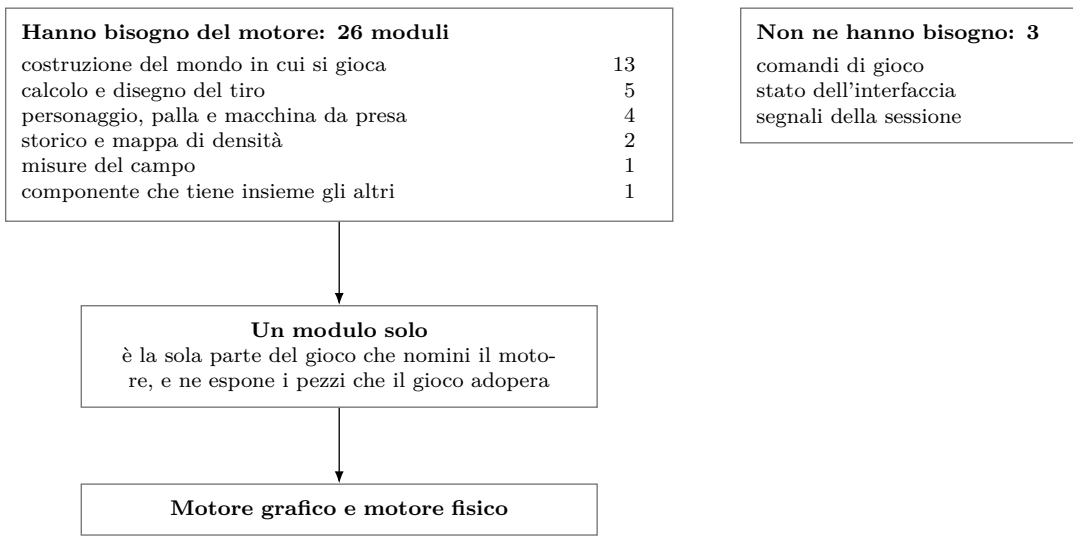


**Figura 3.2:** I trenta moduli in cui il gioco è diviso, raggruppati per responsabilità e contati. Fra i moduli e il motore esiste un solo passaggio; i tre del riquadro a destra non lo attraversano, perché del motore non hanno bisogno.

Del secondo livello ho detto finora che cosa gli spetta decidere, non che cosa sappia fare. La tabella 3.2 lo riporta per intero, e per intero significa quattro operazioni: due servono a ottenere una credenziale, due sono il salvataggio di un tiro e la lettura dello storico. È un'interfaccia piccola, e la sua piccolezza

non è modestia del prodotto: è la conseguenza di ciò che il livello deve fare, cioè conservare e decidere, e di nient'altro.

Vale la pena di guardare che cosa la tabella non contiene, perché è lì che il progetto si vede. Nessuna delle quattro operazioni accetta l'identificativo di un utente fra i dati che porta con sé, e nessuna consente di nominarne uno. Dall'esterno non si possono chiedere i tiri di qualcuno: si possono chiedere soltanto i propri, e a stabilire quali siano è la credenziale che firma la richiesta. È la terza regola della sezione 3.3.2 vista dal lato dell'interfaccia, ed è anche la ragione per cui l'interfaccia è così corta: un'operazione in più sarebbe stata un modo in più di sbagliare la stessa cosa.

| Richiesta                  | Credenziale | Che cosa porta                       | Che cosa restituisce                  |
|----------------------------|-------------|--------------------------------------|---------------------------------------|
| POST<br>/api/auth/register | no          | nome, indirizzo e parola d'accesso   | la credenziale firmata                |
| POST<br>/api/auth/login    | no          | nome o indirizzo, e parola d'accesso | la credenziale firmata                |
| POST<br>/api/shots         | sì          | posizione, potenza ed esito del tiro | il tiro come è stato archiviato       |
| GET<br>/api/shots          | sì          | nulla                                | i soli tiri di chi firma la richiesta |

**Tabella 3.2:** Le quattro operazioni che il servizio espone. Le due che non richiedono una credenziale sono quelle che servono a ottenerne una; nessuna delle quattro accetta l'identificativo di un utente fra i dati che porta con sé.

### 3.3.2 I vincoli architetturali adottati

Le tre regole che seguono non sono convenzioni di stile. Ciascuna vieta al codice qualcosa che avrebbe potuto fare, e ciascuna l'ho applicata senza eccezioni, perché una regola architetturale rispettata quasi sempre non è una regola ma un'abitudine, e su un'abitudine non posso contare quando rileggo codice che non ho appena scritto. Di ognuna dico che cosa impone, perché esiste e che cosa accade se la violo: è quest'ultimo punto a distinguere una decisione di progetto da una preferenza.

### **Il ciclo di *rendering* gira fuori dalla zona del *framework***

Angular ridisegna l'interfaccia quando sospetta che qualcosa sia cambiato, e per sospettarlo sorveglia le operazioni asincrone che accadono dentro la propria zona: un temporizzatore che scatta, una risposta che arriva, un evento del *mouse*. Ogni volta che una di esse termina, il *framework* avvia un ciclo di rilevamento delle modifiche, cioè ripercorre l'albero dei componenti e confronta ciò che ciascuno mostra con ciò che dovrebbe mostrare.

Il ciclo di *rendering* del motore grafico è, per il *framework*, un'operazione asincrona come le altre, e si ripete sessanta volte al secondo. Se lo lasciassi dentro la zona pagherei sessanta cicli di rilevamento al secondo, e li pagherei interamente a vuoto: ciò che cambia fra un fotogramma e l'altro sta nella scena tridimensionale, che non è fatta di componenti e che nessun confronto sull'albero può vedere. Il costo cadrebbe per giunta sul solo bilancio che in un gioco non è elastico, i sedici millisecondi entro cui un fotogramma va consegnato per restare a sessanta al secondo.

Tengo dunque fuori dalla zona tutto ciò che il motore tocca: il ciclo di *rendering*, la costruzione della scena, la simulazione fisica, i comandi da tastiera e da *mouse* del gioco, i richiami che il motore fisico esegue quando due corpi si toccano. Rientrano soltanto le grandezze che l'interfaccia mostra davvero (il punteggio, il livello della barra di potenza, il messaggio che accompagna un canestro, il cambio di pagina) e rientrano qualche volta per tiro invece che sessanta volte al secondo. È questo rapporto, più che la regola in sé, a dire perché la regola conviene.

### **Le collisioni si calcolano solo su primitive geometriche invisibili**

Nessun modello importato partecipa al calcolo delle collisioni. Per ciascuno degli oggetti che si devono urtare, come il personaggio, la palla, il ferro, il tabellone, il pavimento, i muri, esiste una forma invisibile, una capsula, una sfera o un riquadro, e il motore fisico conosce soltanto quella; il modello visibile la segue e non partecipa.

La ragione immediata è il costo. Un modello importato è una *mesh*, cioè un guscio descritto triangolo per triangolo, e i suoi triangoli sono migliaia; una capsula è descritta da due numeri, un raggio e un'altezza. Interrogare la seconda a ogni passo della simulazione costa una frazione trascurabile di ciò che costerebbe interrogare la prima, e quel passo ricade dentro i sedici millisecondi di poco fa.

La ragione che conta di più, però, è un'altra, e riguarda che cosa accade se la regola non si applica. Una *mesh* nata per essere guardata non è garantita né chiusa né coerente: può avere facce orientate al contrario, spessore nullo, buchi che nessuno nota perché da fuori non si vedono. Un motore fisico su una superficie del genere non si rifiuta di lavorare e non segnala nulla: produce compenetrazioni, scatti, corpi che attraversano un muro perché in quel punto il muro, per lui, non c'era. Il difetto non arriva come un errore ma come un comportamento sbagliato, ed è la categoria di guasti su cui questo capitolo torna più di una volta, perché è la sola che non si trova leggendo un registro.

La regola ha un prezzo, e vale la pena di dichiararlo: la forma fisica e la forma visibile di uno stesso oggetto possono divergere, e nulla nel codice impedisce che lo facciano. A tenerle allineate sono io, decidendo per ciascuna grandezza quale delle due la possiede. È il filo che la sezione [3.4.1](#) segue per intero.

### **L'identità di un *record* appartiene alla base di dati**

L'applicazione nel *browser* non genera mai un identificativo, nemmeno quando è lei a creare il *record*: quando registra un tiro invia la posizione, la potenza e l'esito, e nient'altro. L'identificativo lo assegna la base di dati, e torna indietro insieme al *record*.

Il motivo non è di eleganza. Un identificativo generato da chi chiede è un dato che chi custodisce deve accettare senza poterlo verificare, e da quel momento quel campo non è più un identificativo ma un parametro. Due richieste possono sceglierlo uguale, e allora un *record* ne sovrascrive un altro; una sola richiesta può sceglierlo pari a quello di un *record* che non le appartiene, e allora il danno smette di essere un incidente. Il costo di non concederlo è nullo, un campo

in meno da riempire, ed è questo a rendere la regola facile: non c'è nulla da bilanciare.

La stessa idea, applicata all'identità di chi tira invece che a quella del *record*, regge il solo punto in cui questo sistema fa una promessa di sicurezza; nella sezione [3.4.2](#) la si vede lavorare.

### 3.4 Codifica: tre problemi degni di attenzione

Non ho scelto questi tre problemi perché fossero i più grossi, ma perché da ciascuno ho ricavato qualcosa che vale anche fuori da questo progetto. Il primo mi ha insegnato ad attribuire ogni grandezza a un proprietario solo, e a riconoscere i difetti che nascono dove quell'attribuzione manca; il secondo che cosa comporti seguire un dato lungo un sistema intero, dal punto in cui nasce a quello in cui qualcuno lo guarda; il terzo che sui numeri conviene misurare invece che stimare, e che esiste un'intera classe di guasti di cui nessun controllo automatico si accorge. Il resto del lavoro non è meno lavoro: è lavoro da cui ho imparato meno, e tre affondi sono quanti ne regge un capitolo che deve anche dire com'è fatto il prodotto e come l'ho verificato.

#### 3.4.1 La meccanica del tiro

Mentre il gioco gira, un pugno di numeri cambia sessanta volte al secondo: dove si trova il personaggio sul pavimento, a che altezza, in che direzione guarda, dove si trova la palla. Chiamo grandezza ciascuno di questi numeri, e proprietario il pezzo di programma che è il solo autorizzato a scriverlo: tutti gli altri lo leggono quanto vogliono, ma assegnargli un valore nuovo spetta a lui e a nessun altro. È la regola annunciata in chiusura della sezione [3.3.2](#). Il tiro è il punto in cui il progetto è più denso di scelte, e quasi tutte discendono da lei.

La regola serve perché, quando due pezzi di programma scrivono la stessa grandezza nello stesso fotogramma, nessuno dei due è sbagliato per conto suo: ciascuno fa ciò per cui l'ho scritto, e quello che si vede sullo schermo è l'esito del tiro alla fune fra i due. Il difetto compare così in un terzo posto e non somiglia

a nessuna delle due cause: è il tipo di guasto che si cerca dove non è. Seguo qui la regola dal personaggio fino alla palla, e chiudo là dove la regola non arriva.

### I tre strati del personaggio

Il personaggio che corre, salta e tira non è un oggetto solo, ma tre sovrapposti, e la tabella 3.3 dice che cosa possiede ciascuno. Il primo è una capsula invisibile, una delle primitive geometriche di cui parla la sezione 3.3.2: possiede la posizione orizzontale ed è l'unica forma del personaggio che il motore confronti con i muri, quindi è lei che li urta ed è la sola a urtarli. Il secondo è il nodo a cui il modello visibile è appeso, cioè un oggetto senza forma né aspetto che porta con sé una posizione e un orientamento e li impone a tutto ciò che gli sta sotto: possiede la quota, che è quanto dire il salto, e la direzione verso cui il personaggio è girato. Il terzo sono le sequenze di animazione che arrivano insieme al modello, corsa, salto e tiro già registrati dentro il *file* del personaggio, e non possiedono niente: scelgono quale movimento far vedere, e nessun numero del gioco dipende da loro. Il confine fra il primo strato e il secondo passa dentro una riga di programma, e poco più avanti la si vede: a ogni fotogramma il modello copia dalla capsula le due coordinate orizzontali, e non la terza.

| Strato                                     | Che cosa possiede                                     | Che cosa non tocca                           |
|--------------------------------------------|-------------------------------------------------------|----------------------------------------------|
| La capsula invisibile                      | La posizione orizzontale e le collisioni con il mondo | La quota e l'orientamento                    |
| Il nodo a cui il modello visibile è appeso | La quota, cioè il salto, e l'orientamento             | Le collisioni, che non calcola e non subisce |
| Le sequenze di animazione del modello      | Nulla: sceglie soltanto quale sequenza mostrare       | Ogni grandezza del gioco                     |

**Tabella 3.3:** I tre strati che compongono il personaggio, con la grandezza di cui ciascuno è proprietario. La colonna di destra è quella che rende la tabella una regola invece che una descrizione: le tre righe non si sovrappongono in nessun punto, e lo strato che si vede non possiede niente.

**Il salto è calcolato, non recitato**

Il modello porta con sé una sequenza di salto, e quella sequenza non solleva nessuno: la quota la calcolo io, con due operazioni per fotogramma. Alla velocità verticale sottraggo la gravità, poi alla quota sommo la velocità appena aggiornata; ripetute sessanta volte al secondo, le due righe disegnano la parabola del salto, e con la spinta iniziale che ho scelto il culmine arriva a settantaquattro centimetri dopo poco meno di quattro decimi di secondo. Dev'essere così perché è il salto a decidere la fisica del tiro: la palla parte da dove si trova il personaggio nell'istante del rilascio, quindi l'origine del tiro sale insieme a lui, e la velocità verticale del salto è il dato che dice quanto quell'istante è vicino al culmine, che è ciò su cui si regola l'assistenza al rilascio. Un'animazione mi avrebbe dato un movimento che si vede ma di cui il programma non conosce i numeri: né la quota di adesso, né quanto manca al culmine.

La regola si vede meglio dove ho dovuto imporla. Il passo che muove il personaggio in orizzontale lo spinge anche un poco verso il basso a ogni fotogramma, di una quantità fissa, perché la capsula resti appoggiata al pavimento invece di staccarsene: non è una gravità, è una colla. Mentre il personaggio è in aria quella spinta la azzerò, altrimenti due pezzi di programma scrivono la stessa quota nello stesso fotogramma, e lo spostamento con collisioni tira verso il basso, fotogramma dopo fotogramma, ciò che il salto ha appena alzato. Il sintomo non somiglia a un errore di calcolo: si vede un personaggio che sprofonda invece di saltare, e, l'ho già detto, nessuno dei due pezzi di programma letto per conto suo ha qualcosa che non va.

Il codice [3.1](#) mette le due decisioni una accanto all'altra, perché nel programma stanno a poche righe di distanza e si leggono insieme. La prima è il condizionale in cima: una riga, ed è tutta la difesa contro il personaggio che sprofonda. La seconda sta nelle tre righe finali, dove il confine fra i due strati non è scritto da nessuna parte, perché è l'assenza della terza coordinata. Nessuno strumento me lo segnalerebbe se qualcuno la aggiungesse: il programma continuerebbe a compilare e a comportarsi bene in ogni situazione tranne una, il salto.

```
1 // tiene la capsula sul pavimento, tranne quando e' in aria
2 direction.y = this.jumping ? 0 : this.scene.gravity.y;
3 this.collider.moveWithCollisions(direction);
4
5 // [...] la capsula viene riportata dentro i limiti del campo
6
7 const pos = this.collider.position;
8 this.root.position.x = pos.x;
9 this.root.position.z = pos.z;
```

**Codice 3.1:** Estratto del passo che muove il personaggio, eseguito a ogni fotogramma. Le tre righe finali allineano il modello alla capsula sulle sole coordinate orizzontali; la prima decide se la capsula, in questo fotogramma, debba essere spinta verso il basso.

### L'angolo di lancio dipende dalla posizione

Nell'istante del rilascio devo dare alla palla una velocità, e una velocità sono tre cose: la direzione in cui punta, guardando il campo dall'alto, che è del giocatore; il modulo, cioè quanto forte la palla parte, che dipende da quanto a lungo ha caricato; e l'angolo di elevazione, cioè di quanto quella velocità punta verso l'alto rispetto al pavimento, che non gli chiedo affatto. All'angolo avevo dato un valore fisso, cinquantadue gradi, e un valore fisso dichiara che tutte le posizioni del campo si somigliano: da lontano spreca velocità a salire invece che a coprire terreno, da sotto il canestro non basta a guadagnare l'altezza che serve sulla distanza minuscola di cui dispone.

Al suo posto calcolo l'angolo per ogni posizione:

$$\theta = 45^\circ + \frac{1}{2} \arctan \frac{\Delta y}{d}$$

dove  $d$  è la distanza orizzontale fra l'origine del tiro e il centro del ferro e  $\Delta y$  la differenza di quota fra i due. Non l'ho inventata: è il risultato classico del moto del proiettile. Uno stesso bersaglio si raggiunge con molti angoli diversi, ciascuno a patto di tirare con la velocità che gli corrisponde, e questo è l'angolo a cui quella velocità è la più bassa di tutte. Vale quarantacinque gradi per un bersaglio alla stessa quota, si alza quando il ferro è alto rispetto alla distanza, si abbassa quando è la distanza a dominare. La figura 3.3 lo mostra per quello che è geometricamente, la bisettrice fra la verticale e la retta che dall'origine punta

al ferro. Non è un valore scelto a mano e riuscito giusto al primo tentativo: è la soluzione di un problema, e infatti non ho tarato niente. La mira laterale, che invece è tarata, lo è almeno lungo gli assi giusti: cresce con la distanza e con la potenza, le due grandezze da cui il gesto dipende davvero.

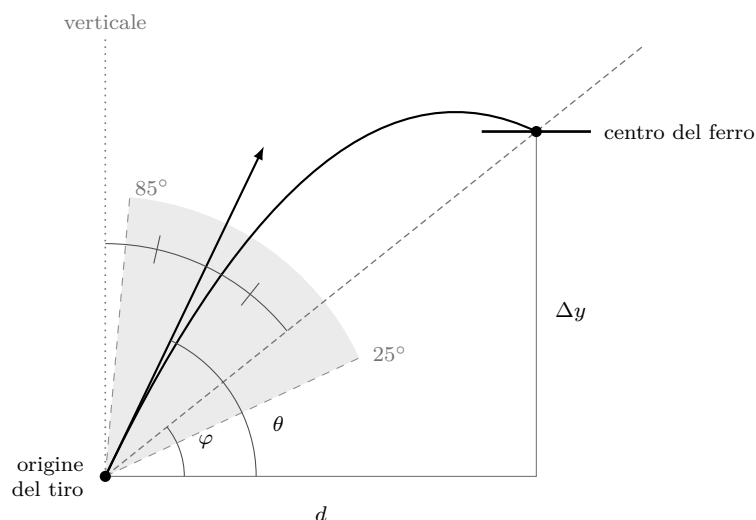


**Figura 3.3:** L'angolo di lancio  $\theta$  calcolato per una posizione: è la bisettrice fra la verticale e la retta che dall'origine del tiro punta al centro del ferro, e i due archi con il trattino segnano i due angoli uguali che la definiscono.  $\varphi$  è l'angolo sotto cui l'origine vede il ferro, cioè  $\arctan(\Delta y/d)$ . La zona grigia è l'intervallo entro cui riporto comunque l'angolo: la verticale ne resta fuori, ed è la ragione per cui un bersaglio quasi sopra la testa chiede più di quanto l'intervallo conceda.

L'angolo che la formula restituisce lo riporto comunque dentro l'intervallo fra venticinque e ottantacinque gradi, perché fuori di lì otterrei un pallonetto quasi verticale o un tiro tesissimo, e nessuno dei due somiglia a un tiro da pallacanestro. Il limite superiore è ottantacinque e non un più tondo ottanta, e la ragione merita il racconto. Da sotto il canestro l'origine della palla dista meno di trenta centimetri dal muro e il ferro sta quasi due metri e mezzo più in alto: lì l'angolo di minima velocità vale davvero ottantasei o ottantasette gradi, e la formula lo chiede a ragione, perché quella è la geometria di quel tiro. Un limite più basso non produce un tiro peggiore in quella zona: produce un tiro impossibile.

Impossibile alla lettera, e il modo in cui il programma lo dice merita attenzione. Fissato l'angolo, ricavo la velocità che porta la palla al centro del ferro in forma chiusa, cioè con una formula che la dà direttamente invece di cercarla

per tentativi; e quella formula ha senso solo se la retta lungo cui la palla parte passa sopra il ferro. Se l'angolo consentito è troppo basso per superare quell'altezza su quella distanza, non esiste alcuna velocità che risolva il problema, e la funzione restituisce un non numero. Nessuno se ne accorge: il programma non si ferma e non scrive niente da nessuna parte, e soltanto l'assistenza al rilascio, che su quella velocità si regola, da lì in avanti non fa più nulla, esattamente nel caso ravvicinato per cui l'angolo adattivo esiste. Non è un guasto che qualcuno segnala, è una funzionalità che c'è in una zona del campo e non in un'altra: la stessa classe di difetti su cui la sezione [3.4.3](#) torna per intero.

#### **L'assistenza al culmine corregge la potenza, non la mira**

Un tiro in salto è un problema di tempismo: fra lo stacco e l'atterraggio il giocatore sceglie un istante, ed è la sola cosa che possa sbagliare senza sapere di quanto. Calcolo allora, per la posizione da cui sta tirando, il modulo che manderebbe la palla al centro del ferro, e spostato verso quel valore il modulo con cui ha rilasciato davvero. Non lo sostituisco: lo sposto di una frazione, tanto più grande quanto più il rilascio è vicino al culmine del salto. La frazione vale zero allo stacco e all'atterraggio, il massimo esattamente al culmine, dove la velocità verticale passa per lo zero, e fra i due estremi cambia con continuità. Al suo posto avevo una finestra di tempo: dentro, l'aiuto c'era tutto; fuori, non c'era affatto. Il gradino si sentiva, perché due rilasci distanti un fotogramma, che nessun giocatore sa distinguere, davano esiti che non si somigliavano.

La direzione non la tocca mai, ed è questa, più della formula, la decisione di progetto: scegliendo che cosa correggere scelgo che cosa il gioco perdona. Perdona l'errore di potenza, che nessuno sa dosare a occhio, e non perdona l'errore di mira, che è l'abilità che chiede. L'angolo con cui calcolo l'arco ideale, poi, è lo stesso con cui calcolo il tiro vero, perché è una sola funzione chiamata da entrambi: se fossero due, l'assistenza spingerebbe la velocità verso un arco diverso da quello che la palla percorre. È la regola del proprietario unico applicata a una grandezza che non appartiene a un oggetto ma a un calcolo.

Della stessa decisione esiste il rovescio, ed è l'unico punto in cui il gioco punisce invece di perdonare: chi comincia a caricare in aria e tocca terra prima

di rilasciare non ha tirato né da fermo né in salto, e glielo faccio sentire ruotando di poco la direzione del tiro, con uno scarto che sorteggio al rilascio. È la sola cosa in tutto il gioco che sposti la mira, e la riservo a quel caso perché togliere il controllo dell'abilità che il gioco chiede è la penalità più eloquente di cui disponessi. I due trattamenti si escludono a vicenda, e insieme dicono che cosa premio.

### Quattro stati, due regimi

Anche la palla ha un proprietario, ma non sempre lo stesso: attraversa i quattro stati della figura 3.4, e a metà strada cambia chi le scrive la posizione. Appena rilasciata non la governa il motore fisico: la posizione la scrivo io, fotogramma per fotogramma, con la parabola in forma chiusa

$$\mathbf{p}(t) = \mathbf{p}_0 + \mathbf{v} t + \frac{1}{2} \mathbf{g} t^2$$

dove  $\mathbf{p}_0$  è l'origine del tiro,  $\mathbf{v}$  la velocità di rilascio,  $\mathbf{g}$  la gravità e  $t$  il tempo trascorso dal rilascio. Dato un istante, la formula dice dove si trova la palla, e lo dice senza sapere dove fosse al fotogramma precedente: è questo che si intende per forma chiusa. Appena tocca qualcosa il governo passa al motore fisico, e gli consegno lo stato esatto di quell'istante: la velocità che la palla ha in quel punto e la rotazione che le stavo imponendo, tradotta in velocità di rotazione vera, così che i rimbalzi mordano il pavimento e i giri sul ferro vengano da sé.

La soluzione ibrida la motivo con ciò che ciascuno dei due regimi sa fare. Finché la palla non tocca niente non c'è nulla che un motore fisico possa aggiungere: applicherebbe la stessa gravità, ma un pezzetto alla volta, sommando a ogni fotogramma un tratto di traiettoria al tratto precedente, e da quelle somme ripetute nascono il piccolo errore che si accumula e il tremolio che ogni calcolo passo passo si porta dietro. La forma chiusa dà invece l'arco esatto, e l'anteprima della traiettoria che il giocatore ha sotto gli occhi non è un disegno somigliante: è la stessa formula, valutata in una manciata di istanti e unita in una linea. Nell'istante in cui la palla tocca il ferro, però, il rimbalzo, la rotazione e il giro che entra o che esce sono esattamente ciò che una formula non sa fare.

Il confine fra i due regimi si paga in due modi, e il primo me l'ha fatto scoprire il tiro più pulito che esista. Un canestro che entra senza sfiorare nulla non tocca nulla per definizione, quindi non fa mai scattare il passaggio: quando dichiaro concluso il tiro la formula governa ancora la palla, e da quel momento non la governa più nessuno, perché la formula smette di scriverne la posizione e il motore non l'ha mai presa in carico. Resta ferma a mezz'aria, per sempre. Forzo perciò il passaggio prima di chiudere il tiro, ed è la ragione per cui nella figura 3.4 dal volo calcolato non parte nessuna freccia diretta verso il tiro concluso.

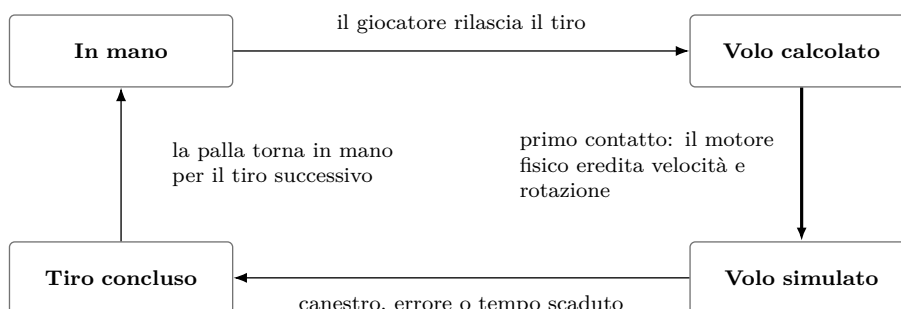


**Figura 3.4:** I quattro stati che la palla attraversa e la condizione che fa scattare ogni passaggio. La freccia più marcata è quella che separa i due regimi: prima di essa la posizione della palla la scrive una formula, dopo di essa la governa il motore fisico. Dal volo calcolato non esce alcuna freccia diretta verso il tiro concluso, perché faccio passare per la simulazione anche il canestro che non tocca nulla: senza quel passaggio resterebbe una palla che nessuno dei due regimi governa.

### Due difetti istruttivi

Il secondo modo in cui il confine si paga è il tempo. Nel volo calcolato la palla non percorre la parabola, la salta: a ogni fotogramma valuto la formula nell'istante corrente e sposto la palla lì, senza che nessuno guardi che cosa ci fosse in mezzo. Finché i fotogrammi si susseguono fitti quel salto è breve e non se ne accorge nessuno; se il calcolatore rallenta e fra un fotogramma e il successivo passa troppo tempo, il salto diventa lungo abbastanza da scavalcare il tabellone — il punto di prima sta davanti, quello di dopo sta dietro, nessuno dei due cade dentro il tabellone, che quindi nessuno dei due tocca. La palla lo attraversa come se non ci fosse. Il rimedio è non lasciare che quel salto cresca: la fase calcolata accetta al massimo un trentesimo di secondo per volta, così che

un fotogramma lungo produca un volo che rallenta invece di una palla che passa attraverso; dall'altro lato del confine il motore fisico si difende da sé, spezzando i fotogrammi lunghi in sottopassi da dieci millesimi di secondo. La lezione è che la fase calcolata non è una fisica più semplice: è un regime con un proprio dominio di validità, e quel dominio ha un limite superiore sul tempo.

L'altro difetto l'ho capito sbagliando, ed è per questo che lo racconto. Per il motore fisico il ferro non è un anello ma due capsule affiancate, ed è lì che la palla qualche volta si ferma: appoggiata a una sola, oppure posata esattamente nell'incavo fra le due. Resta immobile, e il tiro non si chiude più. La mia prima diagnosi è stata l'attrito, e ho cercato la coppia di valori, attrito e coefficiente di rimbalzo, che la lasciasse scendere. Quella coppia non esiste, per una ragione che nessuna taratura può aggirare: una palla posata nell'incavo fra due capsule è in equilibrio stabile, cioè si trova in un punto dove le forze si annullano e da cui ogni piccolo spostamento la riporta indietro invece di allontanarla. Non è l'attrito a tenerla ferma, è la forma; e abbassare l'attrito quanto basterebbe a liberarla renderebbe scivoloso ogni altro contatto del gioco. Avevo sbagliato la categoria del problema, non il valore: la domanda giusta non era quale attrito mettere, ma che cosa tenesse ferma la palla.

Riconosciuta la categoria, il rimedio è quello che si dà a un equilibrio: romperlo. Quando la palla resta quasi immobile all'altezza del ferro e dentro il suo ingombro per un quarto di secondo, riceve un impulso una tantum verso l'esterno, che rompe l'appoggio simmetrico fra le due capsule, e verso il basso, che rompe quello su una capsula sola. È una correzione locale, che agisce soltanto su una situazione che so riconoscere, e la preferisco a un parametro globale che avrebbe cambiato il comportamento di tutto il resto per curare un caso.

La regola con cui ho aperto ha retto per tutta la sezione, ma i difetti che ho raccontato non nascono dentro uno strato: nascono dove la proprietà passa di mano. Il personaggio sprofonda dove il salto e il movimento si contendono la stessa quota; la palla resta a mezz'aria, o attraversa il tabellone, nell'istante in cui il governo passa dalla formula al motore fisico. Attribuire una grandezza è la parte facile: quella delicata è il momento in cui l'attribuzione cambia, e conviene sapere in anticipo quali sono quei momenti, perché sono pochi e si

possono contare. Il difetto del ferro insegna invece un'altra cosa, ed è quella che porto più volentieri con me: prima di cercare il valore giusto conviene chiedersi se il problema è di valori.

### 3.4.2 Mappa di densità e stima locale

Questa sezione segue un dato solo, la posizione da cui è partito un tiro, dal punto in cui nasce dentro il motore grafico fino al *pixel* che lo rappresenta sulla mappa di densità, l'immagine a colori che il gioco sovrappone al campo per mostrare da quali zone il giocatore ha tirato di più. È il percorso che attraversa tutti e tre i livelli della sezione 3.3.1, e a ogni tappa il dato guadagna o perde qualcosa: un'identità, un istante, un nome nuovo, i campi che al gioco non servono. Quei guadagni e quelle perdite li ho decisi uno per uno. Sullo stesso storico poggia poi la stima della probabilità di segnare, che dagli stessi tiri ricava una cosa diversa.

#### Dove nasce il dato

Registro la posizione del giocatore nell'istante del rilascio, non il punto in cui la palla finisce, e ne tengo le due coordinate sul pavimento buttando via la quota, perché su un campo guardato dall'alto il salto non ha niente da dire. È una decisione di dominio, non tecnica, e cambia che cosa la mappa significhi: è una mappa di zone di tiro, non di rimbalzi. Registro anche l'esito, cioè se il tiro è entrato, e la potenza con cui è partito. La potenza alla mappa non serve: la chiede la stima, e prima ancora l'obiettivo D02, che la vuole condizionata anche ai parametri di lancio.

#### Scrivere senza fermare la partita

La richiesta che salva il tiro parte e il gioco prosegue senza aspettarne la risposta: un eventuale errore finisce soltanto nel registro diagnostico. Il prezzo lo dichiaro: l'archivio non è garantito completo, perché un tiro può andare perduto per una rete lenta senza che nessuno se ne accorga. L'alternativa era

far dipendere la partita dal tempo di risposta della rete, e un fotogramma speso ad aspettare un servizio remoto il giocatore lo vede come uno scatto.

### **La catena che il dato attraversa**

Sul servizio la richiesta percorre una catena di funzioni, ciascuna delle quali può fermarla o passarla alla successiva: le rotte, che riconoscono l'indirizzo a cui è arrivata; il controllo della credenziale; il controllore, che decide che cosa fare; il modello, che è la sola parte a parlare con la base di dati. Conta la posizione del secondo anello, montato prima del controllore su entrambe le operazioni, la scrittura e la lettura: controllare la credenziale non è un passo del trattamento della richiesta, è la condizione per comincerlo. Una richiesta senza credenziale valida non arriva mai al controllore e non tocca mai la base di dati, e lo si legge dalla forma della catena invece di doverlo verificare in ciascuna funzione.

### **L'isolamento per utente**

La credenziale che accompagna ogni richiesta è un piccolo documento che il servizio ha rilasciato al momento dell'accesso e ha firmato con una chiave che non esce da lì: dentro c'è, fra l'altro, l'identificativo dell'utente, e la firma permette al servizio di riconoscere che quel documento l'ha emesso lui e che nessuno ne ha cambiato il contenuto per strada. Il servizio prende da lì, e soltanto da lì, l'identificativo di chi tira: mai dal corpo della richiesta, mai dai suoi parametri. Alla lettura restringe l'interrogazione a quell'identificativo; alla scrittura intesta il tiro a esso prima di consegnarlo alla base di dati.

È l'unica proprietà di sicurezza sostanziale di questo sistema, e la dico per quello che è: il resto è un gioco a un giocatore, dove non c'è niente da proteggere, mentre lo storico di un utente è suo. Prendere l'identificativo dalla richiesta invece che dalla credenziale non sarebbe una differenza di stile: il servizio riceverebbe un numero scelto da chi chiede, e basterebbe cambiarlo, in una richiesta scritta a mano fuori dal gioco, per leggere lo storico di chiunque. La firma è ciò che sottrae quel campo alla scelta di chi chiede. È la stessa idea della terza regola della sezione [3.3.2](#), applicata all'identità di chi agisce invece che a quella

di un *record*: un dato che decide qualcosa non deve venire da chi ha interesse a deciderlo.

Il codice 3.2 affianca i due versi, e in mezzo non c'è altro. Ciò che conta non è quello che c'è scritto ma da dove viene: in entrambe le operazioni il proprietario è `req.user.id`, cioè il campo che il controllo della credenziale ha estratto dal documento firmato prima ancora che il controllore cominciasse. Nella scrittura i campi presi dal corpo della richiesta sono quattro, e l'utente non è fra loro; nella lettura il proprietario non è un filtro applicato ai risultati, ma una condizione dell'interrogazione, sicché non esiste alcun ordine di esecuzione in cui il servizio abbia in mano i tiri di qualcun altro.

```
1 // scrittura: il tiro nasce già' intestato
2 const { positionX, positionZ, isMade, power } = req.body;
3 const nuovoTiro = new Shot({
4   user: req.user.id, positionX, positionZ, isMade, power
5 });
6 await nuovoTiro.save();
7
8 // lettura: il proprietario condiziona l'interrogazione
9 const tiri = await Shot.find({ user: req.user.id })
10   .sort({ createdAt: -1 });
```

**Codice 3.2:** I due punti del servizio in cui l'identità dell'utente entra in gioco: la creazione di un tiro e il recupero dello storico. In nessuno dei due compare un identificativo che provenga dalla richiesta.

### Il punto in cui i nomi cambiano

Il tiro come sta in archivio e il tiro come lo adopera il gioco non hanno la stessa forma: il primo porta i nomi della base di dati e qualche campo che serve soltanto lì, il secondo i nomi del dominio del gioco. La traduzione avviene in un punto solo, nel servizio dell'applicazione che interroga il *server*: lì i nomi cambiano e i campi che al gioco non servono spariscono, l'identificativo dell'utente compreso, che da quel punto in poi il resto dell'applicazione non sa nemmeno che esista. Il guadagno non è l'eleganza: chi disegna la mappa riceve coordinate già del mondo tridimensionale, e il giorno in cui il formato conservato cambiasse a cambiare sarebbe una funzione sola. La tabella 3.4 raccoglie le tappe e il nome che il dato porta in ciascuna.

Lo stesso punto fa anche da filtro. Un *record* che arrivasse senza una potenza utilizzabile non lo correggo, lo scarto, e la ragione sta nella lettura che ne farò più avanti: nella stima ogni tiro pesa tutti gli altri, quindi un solo valore mancante si propagherebbe a ogni stima dello storico invece di restare confinato alla propria riga.

| <b>Tappa</b>                   | <b>Come si chiama il dato</b>                                                                       | <b>Che cosa cambia qui</b>                                                          |
|--------------------------------|-----------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| Cattura, nel motore grafico    | Una coppia di coordinate del mondo, con la quota azzerata                                           | Nasce; l'esito e la potenza gli si affiancano                                       |
| Corpo della richiesta          | <code>positionX</code> , <code>positionZ</code> , <code>isMade</code> , <code>power</code>          | Diventa un messaggio, e nessun identificativo lo accompagna                         |
| Ingresso del servizio          | gli stessi campi, più <code>user</code>                                                             | L'identità entra qui, e viene dalla credenziale firmata                             |
| Archivio                       | gli stessi, più <code>_id</code> e <code>createdAt</code>                                           | Identificativo e istante li assegna la base di dati                                 |
| Confine del modello di dominio | <code>x</code> , <code>z</code> , <code>isMade</code> , <code>power</code> , <code>timestamp</code> | I nomi cambiano, una volta sola; <code>user</code> e i campi di servizio spariscono |
| Disegno della mappa            | Due coordinate in <i>pixel</i> dell'immagine                                                        | Da metri sul campo a <i>pixel</i> ; qui cade l'inversione dell'asse                 |

**Tabella 3.4:** Le tappe che la posizione di un tiro attraversa e il nome che porta in ciascuna. La terza colonna dice dove il dato acquista qualcosa che prima non aveva: l'identità dell'utente entra in una riga sola e da una fonte sola, l'identificativo del *record* e l'istante li scrive soltanto la base di dati, e i nomi cambiano in una riga e non si toccano più.

### Dal dato al *pixel*

La mappa non è fatta di oggetti tridimensionali, uno per tiro. Disegno invece tutti i tiri su una sola immagine bidimensionale, che applico come *texture* a un piano sospeso tre centimetri sopra il campo. La stessa immagine governa sia il colore sia la trasparenza del piano, così dove non ci sono tiri le righe del campo si vedono sotto.

Il disegno è in due passate, e la prima fa il lavoro. Ogni tiro diventa una macchia sfumata. Ogni punto di un'immagine, oltre al colore, porta un numero che dice quanto è opaco, e di norma due macchie sovrapposte lo risolvono per copertura, cioè vince l'ultima disegnata; io chiedo invece che si sommino, e con la somma quel numero smette di essere un effetto grafico e diventa un conteggio: dove sono passati dieci tiri vale dieci volte che dove ne è passato uno, cioè misura la densità. La seconda passata rilegge l'immagine, ne cerca il valore massimo e rimappa ogni punto su una scala dal verde scuro al rosso, azzerando ciò che sta sotto una soglia perché il rumore non dipinga il campo intero. La conversione dalle coordinate del campo a quelle dell'immagine porta un'inversione sull'asse verticale, perché le due convenzioni non concordano su dove stia l'origine: sbagliarla non produce alcuna eccezione, produce una mappa specchiata attorno alla metà campo, che sembra plausibile finché il giocatore non la confronta con il punto da cui ha tirato davvero.

Rimappare sul massimo è a sua volta una decisione: la scala non è assoluta ma relativa allo storico che si sta guardando, quindi il rosso pieno non significa «molti tiri», significa «il punto più battuto di questa mappa», e quel punto esiste con cinque tiri come con cinquecento. Il guadagno è che la mappa si legge fin dalla prima apertura; il prezzo è che due mappe non sono confrontabili, perché il rosso di un utente e quello di un altro non valgono lo stesso numero di tiri.

#### **Che cosa il filtro decide**

Lo storico arriva intero e resta in memoria: i filtri non lo interrogano di nuovo e non lo ritoccano, decidono quali tiri concorrono alla mappa, e cambiarli ridisegna l'immagine senza toccare la rete. Sono due, l'esito e l'intervallo temporale, e un tiro deve soddisfarli entrambi. Il taglio temporale è a giorni di calendario e non a multipli di ventiquattro ore, perché è ciò che promette l'etichetta: con una finestra mobile un tiro di martedì pomeriggio uscirebbe da «oggi» nel pomeriggio di mercoledì, non a mezzanotte.

**La seconda lettura: stimare invece di contare**

Sullo stesso storico poggia la stima della probabilità di segnare, che risponde all’obiettivo D02: per ogni tiro, un numero fra zero e uno che dice quanto era probabile che entrasse, dedotto da tutti gli altri tiri del giocatore. La via ovvia sarebbe dividere il campo in zone e dare a ogni tiro la percentuale della sua zona. Non l’ho presa perché i confini sarebbero arbitrari: due tiri a mezzo metro l’uno dall’altro riceverebbero stime molto diverse soltanto perché fra loro passa una riga che ho disegnato io. Non conto dunque dentro una zona, ma peso tutti i tiri dello storico, ciascuno in proporzione a quanto somiglia a quello che sto stimando:

$$P_i = \frac{\sum_{j \neq i} w_{ij} y_j + k \bar{p}}{\sum_{j \neq i} w_{ij} + k} \quad (3.1)$$

dove  $P_i$  è la probabilità stimata per il tiro  $i$ ,  $y_j$  vale uno se il tiro  $j$  è entrato e zero altrimenti,  $\bar{p}$  è la percentuale media del giocatore e  $k$  il peso che le assegno. Senza i due termini che contengono  $k$  la formula sarebbe una media degli esiti degli altri tiri, in cui però ciascuno conta per il proprio  $w_{ij}$  invece che per uno. Quel peso è il prodotto di due gaussiane, cioè di due campane che valgono uno quando i due tiri coincidono e scendono verso lo zero via via che si allontanano: una misura la distanza sul campo fra i due tiri, l’altra lo scarto fra le loro potenze di rilascio.

Le due campane restano distinte invece di fondersi in una misura sola di somiglianza, perché metri e potenza non sono grandezze commensurabili: metterle insieme vorrebbe dire decidere quanti metri vale un decimo di barra, e non c’è ragione per scegliere un cambio di unità piuttosto che un altro. Tenendole separate, l’ampiezza di ciascuna resta leggibile nella propria unità, e le due ampiezze sono la sola cosa che dichiaro: un metro e mezzo sul campo, e un ottavo scarso dell’escursione della barra di carica, che sulla velocità di rilascio vale poco meno di un metro al secondo. Sono la traduzione in numeri di «da qui» e di «tirando così», ed è il secondo asse a distinguere quanto sono bravo da qui da quanto sono bravo da qui tirando così: è quella la domanda utile, perché a posizione fissata esiste una potenza giusta e il resto sono tiri corti o lunghi.

La tabella 3.5 dice che cosa comportano quelle due ampiezze. Sulla prima

| Distanza fra i due tiri | Scarto fra le potenze di rilascio |        |        |         |
|-------------------------|-----------------------------------|--------|--------|---------|
|                         | 0                                 | 0,12   | 0,24   | 0,36    |
| Stesso punto            | 1                                 | 0,61   | 0,14   | 0,011   |
| 1,5 m                   | 0,61                              | 0,37   | 0,082  | 0,0067  |
| 3 m                     | 0,14                              | 0,082  | 0,018  | 0,0015  |
| 4,5 m                   | 0,011                             | 0,0067 | 0,0015 | 0,00012 |

**Tabella 3.5:** Il peso con cui un tiro concorre alla stima di un altro, al variare della distanza fra i due sul campo (righe) e dello scarto fra le loro potenze di rilascio (colonne, espresso in frazione dell'escursione della barra di carica: 0,12 vale poco meno di un metro al secondo). Due tiri identici pesano uno. La prima riga e la prima colonna danno il costo di un solo scarto per volta; la diagonale mostra che i due pesi si moltiplicano, ed è la ragione per cui il vicinato che sostiene ciascuna stima è più piccolo di quanto i due assi, presi uno alla volta, lascino pensare.

riga e sulla prima colonna si legge il costo di un solo scarto per volta: uno scarto pari all'ampiezza riduce il peso a sei decimi, uno doppio a un settimo, uno triplo a un centesimo abbondante, e a cinque metri un tiro non partecipa più in alcun modo apprezzabile alla stima di un altro. La tabella è simmetrica perché la campana è la stessa sui due assi, non perché abbia tarato un'ampiezza sull'altra: fissare l'ampiezza è dire che cosa conta come vicino, alla forma del calo pensa la campana.

La diagonale è il dato che la prosa non sa dare. I due pesi non si sommano, si moltiplicano: due tiri che differiscono di un'ampiezza su entrambi gli assi non pesano sei decimi ma meno di quattro, e a due ampiezze per asse restano meno di due centesimi. È il prezzo dell'asse in più: per pesarsi a vicenda due tiri devono somigliarsi in tutte e due le dimensioni, sicché il vicinato si rarefa e quelli che sostengono ciascuna stima diventano pochi. È anche la ragione per cui le due correzioni che seguono non sono prudenza ma necessità.

### Le due correzioni

La prima è che un tiro non concorre a stimare se stesso. Senza, un canestro isolato in mezzo al campo si autopredirebbe al cento per cento, perché l'unico dato vicino sarebbe lui: il numero descriverebbe il dato invece di stimarlo, e sarebbe massimamente sicuro proprio dove non sa niente.

La seconda è il termine  $k\bar{p}$  dell'equazione 3.1, che tira la stima verso la media del giocatore. Senza di esso, in una zona poco frequentata, basterebbero i due o tre tiri più vicini a decidere il risultato, che uscirebbe uno zero o un cento per cento: non una stima, ma una coincidenza scritta in percentuale. Con quel termine la stima parte dalla media del giocatore e se ne allontana solo quando i tiri vicini pesano abbastanza da giustificarlo. Il peso  $k$  vale due tiri, cioè la media conta come un paio di tiri presi esattamente da lì: non è una soglia sotto la quale la stima si rifiuta di comparire, è un contrappeso che cede da sé quando le prove aumentano. Anche la media, a sua volta, la smorza con la regola di successione di Laplace, che conta un successo e un fallimento in più di quelli osservati, così che con zero tiri valga la metà invece di essere indefinita. E quando il peso complessivo dei tiri vicini resta sotto una soglia, accompagno il numero con l'avvertenza che è poco sostenuto.

La stima, infine, non la conservo mai in archivio, e non è una dimenticanza. Poiché ogni tiro prende la propria stima dagli altri, la sua probabilità cambia a ogni tiro nuovo: un numero salvato sarebbe vero soltanto nell'istante in cui l'ho salvato, e da lì in avanti sarebbe una vecchia opinione conservata accanto a un fatto, indistinguibile da esso.

Il percorso finisce qui, e le due letture che ne escono dicono la stessa cosa: la mappa non promette che il rosso valga un numero di tiri e la stima non promette una precisione che i dati non sostengono. Lungo la strada il dato ha guadagnato soltanto ciò che qualcuno poteva garantirgli, l'identità dalla firma del servizio e l'istante dalla base di dati, e mai un grado di certezza che nessuno era in condizione di dargli.

### 3.4.3 Rendere pubblicabile il prodotto

Il problema da cui questo lavoro è partito non era che il sito fosse lento: era che il sito non c'era. La compilazione usciva in errore e la cartella di distribuzione restava vuota, quindi non esisteva alcuna versione del prodotto da pubblicare. Il compilatore di Angular consente infatti di dichiarare quanto al massimo possa pesare ciò che produce, e distingue due soglie: una che avverte

e una che ferma. Superata la seconda esce in errore e non scrive niente, e non è un avviso estetico ma un cancello: il risultato pesava otto megabyte e mezzo contro un massimo dichiarato di uno. Finché quel numero restava tale il prodotto non era distribuibile, per quanto bene funzionasse sulla macchina su cui l'avevo scritto.

Quello che segue è il ciclo intero, dalla misura delle alternative alla verifica di ciò che avevo cambiato, e la lezione che ne porto via non riguarda i byte ma ciò che gli strumenti non dicono.

### **Le due cause, e tre esclusioni**

La prima causa era il modo in cui chiedevo i pezzi del motore grafico. Tutti i *file* del gioco, ventotto, li chiedevano al punto di raccolta del pacchetto, cioè all'unico indirizzo che li espone tutti insieme, invece che ai moduli che li definiscono uno per uno. Un compilatore moderno sa togliere dal risultato il codice che nessuno adopera, operazione che va sotto il nome di *tree-shaking*, ma sa farlo soltanto dove il pacchetto gli dichiara quali dei propri *file* sono privi di effetti collaterali: un *file* con effetti collaterali cambia qualcosa nel programma per il solo fatto di essere caricato, quindi toglierlo non è mai sicuro e il compilatore per prudenza lo tiene. Babylon.js dichiara «con effetti» un insieme molto largo, ed è esattamente l'insieme che il mio codice attraversava. Chiedere un solo pezzo a quell'indirizzo costa perciò quanto chiederli tutti, e cinque *file* del gioco lo pagavano per niente: usano soltanto vettori e un po' di matematica, e si trascinavano dietro il motore intero.

La correzione è il codice [3.3](#), e sta tutta lì: lo stesso paio di simboli chiesto a due indirizzi diversi. La prima forma è più comoda da scrivere e non dichiara nulla su che cosa serva davvero, perché quell'indirizzo espone il motore intero; la seconda nomina il modulo che definisce ciascun simbolo, e solo a quel punto il compilatore ha il permesso di lasciare fuori il resto. Ventotto *file* l'hanno ricevuta, e nessuno di essi cambia di una riga in ciò che fa. È la ragione per cui questo intervento è misurabile e insieme invisibile: il programma è lo stesso, cambia soltanto che cosa il visitatore ne deve scaricare.

```
1 // prima
2 import { Vector3, MeshBuilder } from '@babylonjs/core';
3
4 // dopo
5 import { Vector3 } from '@babylonjs/core/Maths/math.vector';
6 import { MeshBuilder } from
    '@babylonjs/core/Meshes/meshBuilder';
```

**Codice 3.3:** La stessa coppia di simboli chiesta al punto di raccolta del pacchetto e ai due moduli che li definiscono. Per il programma le due forme si equivalgono; per il compilatore no.

La seconda causa era che le rotte dell'applicazione, cioè le pagine fra cui si naviga, chiedevano i propri componenti in modo statico: il compilatore non aveva modo di sapere che una pagina poteva servire più tardi, o non servire affatto, quindi le metteva tutte nel blocco iniziale, quello che il *browser* deve scaricare prima di mostrare qualunque cosa, ed è esattamente il blocco che la soglia misura.

Ho messo per iscritto anche che cosa avevo escluso, e conta quanto il resto. Lo strumento di ispezione della scena stava già dietro un'importazione che si risolve soltanto se e quando qualcuno lo apre, e perciò non pesava sul blocco iniziale; il motore fisico arriva come *file* a sé, scaricato mentre il gioco gira e perciò fuori dal conto; e i quattrocentosessantatré blocchi minuscoli che la compilazione produceva non erano una patologia ma il modo in cui il motore grafico si carica i propri programmi di disegno, uno alla volta. Un elenco di cause senza un elenco di esclusioni è una diagnosi a metà: dice dove intervenire, non dove ho già guardato.

### Misurare prima di riscrivere

Cambiare il modo di chiedere i pezzi significava toccare ventotto *file*, e prima di toccarli ho pesato le alternative. Le ho pesate con lo stesso strumento che il progetto usa per compilare davvero, contro le stesse librerie installate e chiedendo esattamente le trentacinque voci, fra classi e funzioni, che il codice adopera: una misura presa con uno strumento diverso avrebbe risposto a una domanda diversa dalla mia. La tabella 3.6 riporta le quattro varianti e il loro

peso in due momenti: dopo la minificazione, che accorcia i nomi interni e toglie tutto ciò che serve soltanto a chi legge il codice, e dopo la compressione che il *server* applica al trasferimento.

| Variante                                                                 | Minificato | Compresso |
|--------------------------------------------------------------------------|------------|-----------|
| Punto di raccolta, con tutti i caricatori                                | 6,78 MB    | 1,53 MB   |
| Punto di raccolta, con il solo caricatore dei modelli                    | 6,53 MB    | 1,45 MB   |
| Moduli singoli, con il solo caricatore dei modelli                       | 3,20 MB    | 0,74 MB   |
| <i>File</i> locale di ri-esposizione, con il solo caricatore dei modelli | 3,20 MB    | 0,74 MB   |

**Tabella 3.6:** Le quattro varianti pesate prima di riscrivere alcunché, con lo stesso strumento che il progetto usa per compilare e chiedendo esattamente i simboli che il codice adopera. La prima riga è il punto di partenza. La colonna di destra è quella che il giocatore paga davvero, ed è là che si legge il dato decisivo: le ultime due righe coincidono, a meno di un chilobyte e mezzo, quindi un solo *file* locale conserva l'intero guadagno senza che i percorsi vadano riscritti in ogni *file* del gioco.

La tabella dice tre cose. Sfoltire i caricatori dei modelli, che è la modifica più facile delle tre, vale poche decine di chilobyte: è un guadagno vero ed è piccolo. Chiedere i pezzi ai singoli moduli dimezza il motore grafico, ed è la sola modifica che tolga byte davvero dal carico di chi gioca. E le ultime due righe coincidono: un solo *file* locale che chiede i pezzi ai singoli moduli e li ri-espone a tutto il resto del gioco conserva l'intero guadagno, a meno di un chilobyte e mezzo. È la misura che ha reso il lavoro fattibile, perché trasforma trentacinque percorsi da riscrivere dentro ventotto *file* in un *file* nuovo e una riga cambiata per ciascuno.

### Il prezzo: guasti che il compilatore non vede

Quel *file* unico ha un prezzo, ed è il punto più istruttivo dell'intero intervento. Chiedendo i pezzi al punto di raccolta ricevevo senza saperlo anche gli effetti collaterali di tutto ciò che vi sta dentro, cioè le righe con cui ciascun modulo, appena caricato, si annuncia al motore grafico e si mette a disposizione di chi lo cercherà. Chiedendoli ai singoli moduli quelle registrazioni non arrivano più, e

la loro assenza non produce alcun errore. Il codice compila identico. Si rompe quando gira, e quasi sempre in silenzio: una funzionalità semplicemente non c'è.

Ne sono uscite cinque, e le ho trovate tutte allo stesso modo, giocando con la console del *browser* aperta. Due meritano il racconto. Nella prima i modelli tridimensionali non si caricavano più, perché registrare il lettore di un formato e registrare il modulo che ne interpreta la versione sono due operazioni distinte e io ne avevo chiesta una sola. La seconda è peggiore: mancava il componente che coordina le collisioni, e la mancanza si manifestava soltanto muovendo il giocatore. La scena si apriva perfetta. Nessun controllo automatico sul codice l'avrebbe vista, e nemmeno la prova più naturale del mondo, che è aprire la pagina e guardare.

Esiste anche il caso opposto, e l'ho verificato: un modulo che non va chiesto affatto, perché il motore lo registra da sé per evitare una dipendenza circolare. Ogni riga di quel *file* porta perciò annotato il punto del programma che la richiede, così che il giorno in cui quel punto sparisse si sappia che si può togliere anche la riga.

### **Il caso che si rompeva solo in sviluppo**

Il guasto più didattico dei cinque riguarda gli *shader*, i programmi che il motore grafico fa eseguire alla scheda video per calcolare il colore di ogni *pixel*. Il motore se li procura da sé, con richieste che risolve mentre gira; chiedendo i pezzi ai singoli moduli quelle richieste non si risolvevano più, e il motore ripiegava sul recupero attraverso la rete. Il *server* di sviluppo, che a ogni indirizzo sconosciuto risponde con la pagina dell'applicazione, restituiva HTML, e il motore provava a compilare HTML come codice per la scheda video.

Due dettagli lo rendono istruttivo. Il primo è che il guasto non toccava la compilazione di produzione, perché lì quelle richieste si risolvono davvero: stavo dunque confrontando due situazioni diverse, e ne ho concluso a torto che la causa fosse un'altra. Il secondo è il sintomo. La scena appariva mezza vuota, con le superfici semplici al loro posto e le altre no, e somigliava a un problema di immagini mancanti. Non lo era: fra quei programmi ce n'è uno che prepara l'immagine con cui illumino l'ambiente, e senza di esso il motore non disegnava

affatto le superfici del tipo più realistico. «Immagini rotte» e «materiali mai disegnati» danno la stessa fotografia e non sono lo stesso guasto, e il tempo che ho perso lì l'ho perso perché ho creduto alla fotografia. La correzione è chiedere quei programmi insieme al resto, così che siano già in casa quando il motore li cerca.

### **Onestà sui numeri**

La seconda causa l'ho corretta caricando le rotte a richiesta, e qui devo essere preciso, perché è facile scambiare una compilazione che passa per un sito più veloce. Caricare a richiesta la rotta del gioco fa passare la compilazione e non rende il sito più rapido: la rotta radice rimanda al gioco, quindi il visitatore tipico scarica quel blocco ugualmente e subito. L'operazione sposta byte dall'etichetta «iniziale» a quella «a richiesta», ed è l'etichetta che la soglia conta. Un beneficio reale c'è, ed è più stretto: chi apre la pagina di accesso o quella dello storico smette di pagare il motore grafico. Ma i byte tolti a tutti li ha tolti l'altra correzione, non questa, e scriverlo al contrario sarebbe stato comodo.

### **Il settantotto per cento che nessuno scaricava**

Sistemato il motore ho misurato di nuovo la cartella prodotta, e il conto diceva una cosa che non mi aspettavo: quasi otto decimi di tutto il codice prodotto erano lo strumento di ispezione della scena, con l'intera libreria di interfaccia che si porta dietro. Circa dieci megabyte che nessun giocatore scaricava mai, perché quel ramo si raggiunge soltanto dalla console del *browser*. Non costava niente a chi giocava: costava peso da pubblicare, e il peso da pubblicare non si vede da nessuna parte finché non lo si misura.

Una guardia su una costante che vale «vero» soltanto nella compilazione di produzione rende quel ramo irraggiungibile, e il compilatore smette di emetterlo. Verificandolo ho trovato due cose che non mi aspettavo: che la guardia svuoti il risultato e non il lavoro, perché il compilatore analizza il grafo intero delle dipendenze e solo dopo elimina ciò che è irraggiungibile, sicché di una compilazione più rapida non rivendico niente; e che il pacchetto debba restare installato,

perché togliendolo la compilazione muore prima del taglio, sul controllo dei tipi, che il compilatore esegue anche sul ramo morto.

#### **Le immagini, e una regola imparata da un caso**

Tolto il peso dal codice, ciò che restava era fatto di immagini. Ho ricodificato in un formato più recente le sette mappe che rivestono le superfici, a risoluzione invariata, e sono passate da undici megabyte e mezzo a poco più di tre; il cielo l'ho invece dimezzato per lato, da due megabyte a mezzo.

Il motivo per cui tratto il cielo diversamente è la sola cosa che mi porto via da questo passaggio. Ricodificarlo nel formato più recente lo faceva diventare più grande, non più piccolo: quel formato conviene sulle immagini ricche di dettaglio e lavora male su una sfumatura liscia, dove la compressione tradizionale è già quasi ottimale. Su un cielo a gradiente l'unica leva che resta è la risoluzione. È il caso particolare di una regola generale: un formato non è migliore di un altro, lo è per un certo contenuto, e stabilire quale sia il contenuto fa parte del lavoro. Anche qui, del resto, il guasto in agguato sarebbe stato muto: i percorsi delle immagini li compongo mentre il programma gira, quindi sostituire un *file* senza cambiare la riga che lo nomina non produce alcun errore, il motore riceve un «non trovato» e disegna la superficie senza immagine.

#### **La diagnosi che avevo sbagliato**

Chiudo con l'errore, perché è la parte da cui ho imparato di più. La mia prima diagnosi non era nessuna delle due cause: incolpavo i fogli di stile dei componenti, che credevo oltre soglia. Era sbagliata due volte. La prima perché citavo il peso del sorgente invece di quello del *file* compilato, che è ciò che la soglia misura. La seconda, più grave, perché il peso compilato non l'avevo misurato affatto: l'avevo stimato. Eseguendo davvero la compilazione, di errori sui fogli di stile non ce n'era nessuno.

La lezione è banale e mi è costata cara: su una soglia si misura, non si stima. Ed è la ragione per cui ho pesato quattro varianti prima di toccare ventotto *file*.

Quel lavoro non è prudenza: è la conseguenza diretta dell'essermi accorto che la stima precedente era campata in aria.

In questa sezione ho tolto codice che nessuno adoperava e immagini più grandi del necessario, ma il guadagno che porto via non si misura in megabyte: è aver capito che il silenzio degli strumenti non è un'informazione. Un compilatore che non protesta non sta dicendo che il programma funziona, sta dicendo che ha finito; una stima fatta a mente non è una misura approssimata, è una misura che non c'è. Le due volte in cui ho preso quel silenzio per una risposta mi sono costate più del problema che stavo risolvendo.

### 3.5 Verifica e validazione

Verificare e validare rispondono a due domande diverse, e le tengo distinte perché il resto della sezione ha senso solo se lo sono. La verifica chiede se ho costruito bene il prodotto, cioè se ciò che il codice fa coincide con ciò che intendevo facesse; la validazione chiede se è il prodotto giusto, cioè se ciò che intendevo coincide con ciò che serviva. La prima la conduco contro il codice e la posso automatizzare, la seconda contro i requisiti e non la automatizza nessuno. Rendo conto qui della prima, e chiudo sulla seconda.

#### **Il criterio: ho separato per poter verificare**

La domanda da cui sono partito non è che cosa fosse importante provare, ma che cosa fosse provabile senza costruire un'impalcatura più grande di ciò che regge. In un programma che vive dentro un motore grafico la risposta non è scontata: quasi tutto ciò che il gioco fa ha bisogno di una scena aperta e di una scheda video che la disegni, e un caso di prova che debba procurarsele non prova più il mio codice, prova il motore.

Il calcolo del tiro sfugge a quella condizione, e non per fortuna: il vincolo enunciato nella sezione [3.3.2](#) lo tiene separato dal motore, e la conseguenza è che una funzione che riceve una posizione, una direzione e una potenza e restituisce una velocità non ha bisogno di nulla per essere interrogata. Le si passano tre numeri e si guarda che cosa risponde. La verificabilità non è dunque un'attività

che ho aggiunto a un prodotto già scritto: è una proprietà che quel prodotto ha perché l'ho progettato così, ed è l'osservazione che riprendo nella sezione 4.3. Sottopongo perciò a *test* di unità ciò che è funzione del solo ingresso: l'angolo di lancio, la velocità che raggiunge il canestro, la correzione che il salto concede al giocatore, il filtro che restringe lo storico a un intervallo di tempo, la stima di probabilità della sezione 3.4.2 e la lettura della risposta del servizio. La scena, i materiali, le animazioni e la simulazione fisica restano fuori, non perché contino meno, ma perché per provarli dovrei riprodurre il motore, e ciò che misurerei sarebbe il motore.

### **Asserire proprietà invece di valori**

Un caso di prova che asserisce il valore atteso di un calcolo come questo è già un *test* che passa sempre. Se scrivo che a una certa distanza la velocità deve valere un certo numero, quel numero non l'ho ricavato da nessuna parte: l'ho letto dal codice che sto provando, e il caso di prova diventa una fotografia di ciò che il programma fa, che resta verde anche quando il programma sbaglia nello stesso modo di sempre. Asserisco allora le proprietà che il calcolo deve avere qualunque numero produca: che a mire opposte corrispondano traiettorie speculari, che la correzione concessa dal salto cresca senza scalini fino all'apice e sia nulla allo stacco e all'atterraggio, che una probabilità stimata non esca dall'intervallo fra zero e uno né ne raggiunga gli estremi sulla scorta di due soli tiri vicini. E dove una proprietà vale su tutto il dominio la controllo su tutto il dominio: il caso di prova nato dai tiri da sotto il canestro, che l'angolo fisso adoperato in precedenza rendeva geometricamente irrisolvibili, non verifica qualche punto scelto a mano ma percorre l'intero campo e pretende che da ogni posizione raggiungibile una soluzione esista. Un controllo su qualche punto notevole avrebbe detto che andava tutto bene, perché la zona guasta era piccola e non era lì che avrei guardato.

### Il rilevatore in un *browser* reale

I *test* di unità non vedono la classe di guasti che la sezione 3.4.3 racconta, e non possono vederla: un'importazione che manca non cambia il risultato di alcuna funzione, cambia ciò che il motore ha in casa quando qualcuno glielo chiede. L'unico posto in cui quei guasti si annunciano è la console del *browser*, e ho scritto un controllo automatico che la legga al posto mio.

Il controllo pubblica la versione compilata del prodotto, apre la pagina di gioco in un Chrome senza finestra, aspetta che la rete si acquieti e conta ciò che è davvero arrivato: i modelli tridimensionali, le immagini che rivestono le superfici e il modulo del motore fisico. Poi fallisce su qualunque errore o avviso del motore grafico, su qualunque eccezione non gestita, su qualunque risorsa che non si sia caricata; ignora invece i rifiuti che il servizio oppone a chi non ha fatto accesso, perché senza credenziali sono la risposta corretta. È quello che si chiama uno *smoke test*, e il nome dichiara da sé la sua ambizione: un rilevatore di fumo dice che qualcosa brucia, non che l'impianto funziona.

### Un *test* che passa sempre è peggio di nessun *test*

Un rilevatore di cui si sappia soltanto che tace non serve a niente, perché tacere è ciò che farebbe anche se fosse guasto, e nel frattempo autorizza a non guardare. Ho verificato allora che sapesse fallire, guastando io il prodotto in due modi diversi e controllando che se ne accorgesse: la tabella 3.7 riporta le tre prove. La terza è quella che vale. Togliere il lettore che interpreta il formato dei modelli non produce alcun errore di compilazione, perché nessuno lo chiama per nome: lo cerca il motore, quando gli si dà un modello da aprire. Quella versione del prodotto si compila pulita e non mostra nulla, e un controllo sul codice continuerebbe ad approvarla.

C'è un'ultima cosa da dire, ed è un difetto mio. La prima versione di questo controllo passava anche con un'immagine cancellata, per la stessa ragione incontrata negli *shader* della sezione 3.4.3: il *server* rispondeva con la pagina dell'applicazione a qualunque indirizzo non trovasse, dichiarando di averla trovata, e il rilevatore non aveva niente da leggere. Un ripiego troppo generoso non

| Scenario                                                      | Esito atteso                                     | Esito osservato                                                                     |
|---------------------------------------------------------------|--------------------------------------------------|-------------------------------------------------------------------------------------|
| Prodotto integro                                              | Nessuna segnalazione                             | Nessun problema; sei modelli, nove immagini e il modulo del motore fisico scaricati |
| Un'immagine rinominata senza aggiornare la riga che la nomina | Fallimento, con la risorsa mancante identificata | Fallimento, con il nome del <i>file</i> che il <i>server</i> non trova              |
| Il lettore del formato dei modelli tolto dalle importazioni   | Fallimento, con i modelli non caricati           | Fallimento, otto problemi, fra cui l'assenza di un lettore per quel formato         |

**Tabella 3.7:** Le tre prove con cui accerto che la verifica automatica non si limiti a passare. Nella prima il prodotto è integro; nelle altre due l'ho guastato di proposito, ciascuna volta in un modo che nessun controllo sul codice segnala. La colonna di destra riporta ciò che il rilevatore dice, non ciò che dovrebbe dire.

rende il sistema più robusto: rende cieco chi lo sorveglia, e l'ho ristretto ai soli indirizzi che non nominano un *file*.

### Che cosa questa verifica non dice

Il limite del rilevatore sta scritto in testa allo *script* che lo esegue, perché è il genere di cosa che si dimentica: carica la scena e basta, non gioca. La più insidiosa delle rotture che ho raccontato, quella del componente che coordina le collisioni, passerebbe di qui indisturbata, perché si manifestava soltanto muovendo il giocatore, ed estendere il controllo a un tiro intero coprirebbe la metà del motore che resta fuori. Lo dichiaro perché un controllo di cui si conosca il perimetro vale più di uno di cui si presuma la copertura.

Resta la seconda domanda, che nessuna di queste prove tocca: i controlli confrontano il codice con ciò che intendevo, e nulla in quel confronto verifica ciò che intendevo. Il termine di paragone l'ho fissato prima di scrivere il codice, ed è la tabella 3.1: ho tradotto gli otto obiettivi in requisiti proprio perché «fatto» avesse per ciascuno un significato che potessi mostrare invece che sostenere. Validare consiste allora nel percorrere quella tabella riga per riga e dire, di ogni requisito, se il prodotto lo soddisfa. Quel percorso lo compiono la sezione 3.6.2 sui requisiti e la sezione 4.1 sugli obiettivi da cui discendono, che è il modo in

cui avevo posto la domanda.

## 3.6 I risultati

I risultati li do in due forme, e nessuna delle due sostituisce l'altra: prima il prodotto come lo incontra chi lo apre, poi la misura di ciò che ne è venuto fuori. Le tengo distinte perché rispondono a domande diverse — che cosa ho fatto e quanto — e perché una cifra non dice se un gioco si può giocare, né un'immagine quanta parte del lavoro concordato copra.

### 3.6.1 Il prodotto realizzato

Questa sezione guarda il prodotto da fuori: che cosa incontra chi apre l'applicazione e che cosa può farci. Del come sia costruito non dico nulla, perché è materia della sezione 3.4. Tre immagini lo mostrano, e le commento una per una, perché un'immagine senza commento non dice quale delle cose che si vedono conti.

La prima cosa che chi apre l'applicazione incontra è che non c'è nulla da attendere. La figura 3.5 mostra il menu principale, e ciò che conta in quell'immagine è lo sfondo: il campo dietro il ventaglio non è un'illustrazione, è la scena tridimensionale già viva e già in movimento, con la telecamera parcheggiata su un'inquadratura di riposo. Il menu e il gioco sono la stessa pagina, quindi premere «Play» non avvia nessun caricamento: il ventaglio si apre verso il bordo e la telecamera scende sul campo. Ciò che c'è da attendere lo si attende una volta sola, aprendo la pagina, e lo si attende mentre si legge il menu.

I tre settori sono le tre cose che si possono fare, e una delle tre cambia quanto valgono le altre due. Chi gioca senza avere un'identità tira, segna e vede il proprio punteggio salire, ma i suoi tiri non entrano in nessuno storico; e l'applicazione glielo dice mentre gioca, invece di lasciarglielo scoprire quando apre la mappa e la trova vuota. È la faccia visibile della regola che ho enunciato nella sezione 3.3.2, per cui il servizio consegna a ciascuno soltanto i tiri di chi firma la richiesta: senza un'identità non c'è nessuno a cui attribuirli.



**Figura 3.5:** Il menu principale: tre settori aperti a ventaglio attorno all'angolo inferiore destro — Play in basso, Heatmap al centro, Account in alto. Heatmap porta il disegno di un tabellone dei punteggi e Account quello di un contratto da firmare; Play non copre nulla e lascia vedere ciò che sta dietro il ventaglio, che non è uno sfondo ma il campo di gioco già disegnato, con il giocatore e la palla al loro posto.

La figura 3.6 è l'istante in cui si decide un tiro. Tenendo premuto si carica la potenza, e mentre si carica l'applicazione disegna l'arco che la palla percorrerebbe rilasciando in quel momento. Non è un'indicazione approssimativa della direzione: è la parabola calcolata sulla posizione, sulla mira e sulla potenza di quell'istante, campionata punto per punto, e l'arco disegnato e il tiro vero la ricavano dallo stesso calcolo. Dice però che cosa il giocatore ha mirato, non dove la palla andrà a finire: al rilascio il tempismo può ancora spostarla, nell'un senso o nell'altro, e come lo faccia l'ho detto nella sezione 3.4.1. È una promessa sulla mira, non sull'esito, ed è la sola che un'anteprima disegnata prima del rilascio possa mantenere.

Intorno c'è ciò che serve mentre si gioca e nient'altro: i canestri sui tentativi, la barra che cresce con la potenza caricata, e una scritta che commenta l'esito nominando ciò che la palla ha toccato, con un moltiplicatore quando i canestri si succedono.

Come si tiri, chi gioca non lo deve indovinare. La prima volta che preme «Play» l'applicazione apre da sé le istruzioni e sospende il gioco finché restano aperte, e da quel momento il punto di domanda in cima allo schermo le richiama quando si vuole. Con pochi comandi e una sola cosa da capire, che è come si



**Figura 3.6:** L'istante del tiro: mentre si carica la potenza, una linea punteggiata disegna l'arco che la palla percorrerà rilasciando in quel momento, dal punto di rilascio fin dentro il canestro. In alto a sinistra i canestri sui tentativi e il richiamo alle istruzioni, in basso la barra che riempiendosi misura la potenza caricata, in alto a destra la palla che riporta al menu.

modula la potenza, una pagina che si rilegge costa a chi già sa meno di un addestramento guidato da attraversare ogni volta.

La figura 3.7 risponde a una domanda che il punteggio non tocca: non quanti tiri ho segnato, ma da dove tiro. Il campo visto dall'alto porta sovrapposta una mappa della densità delle posizioni da cui ho tirato, e due gruppi di filtri la restringono combinandosi fra loro, l'esito da una parte e l'intervallo di tempo dall'altra, così che «i canestri degli ultimi sette giorni» diventi una domanda che si compone con due tocchi. Nessuno dei filtri tocca lo storico: cambiarli ridisegna la mappa, e quando l'intervallo scelto resta senza tiri l'applicazione lo dice con parole diverse da quelle che usa per uno storico ancora vuoto, perché le due cose non sono la stessa e chi guarda deve poterle distinguere. Un cassetto laterale mostra gli stessi tiri in forma di elenco, raggruppati per giornata, per chi voglia leggere i tiri uno per uno invece della loro distribuzione.

Della mappa si vede anche il compromesso che la sezione 3.4.2 argomenta: la scala di colore è relativa a ciò che la mappa sta mostrando, quindi la zona più battuta è rosso pieno sia con cinque tiri sia con cinquecento. È la scelta giusta per la domanda che questa mappa serve, che è «da dove tiro di più», e sarebbe quella sbagliata per «tiro più di prima».



**Figura 3.7:** La mappa di densità sul campo visto dall'alto: le zone più battute virano al caldo, quelle senza tiri lasciano vedere attraverso di sé le righe del campo. In alto i due gruppi di filtri, che si combinano fra loro, l'esito del tiro e l'intervallo di tempo: qui entrambi sono sulla scelta più larga, quindi la mappa mostra lo storico intero. Sul fianco sinistro la linguetta che apre l'elenco degli stessi tiri.

Chiudo su una decisione che in nessuna delle tre immagini si vede, e che riguarda che cosa ci sia da pubblicare. Il servizio non espone soltanto le proprie interfacce: consegna anche l'applicazione compilata, sicché chi apre il prodotto riceve tutto dalla stessa origine, l'applicazione chiede i propri dati a un indirizzo relativo invece che a un *host* da riscrivere a ogni pubblicazione, e non c'è nessun permesso di accesso fra origini diverse da configurare. I tre livelli della figura 3.1 restano quelli, ma ciò che si pubblica è un artefatto solo invece di due componenti da tenere allineate. È la condizione che rende il prodotto pubblicabile, e pubblicabile è quanto mi ero proposto: in esercizio non l'ho messo, e non era fra gli obiettivi.

### 3.6.2 Il prodotto in cifre

I numeri di questa sezione rispondono a tre domande distinte, e li tengo separati perché mescolarli li renderebbe tutti meno leggibili: quanto di ciò che avevo concordato il prodotto copre, quanto prodotto è venuto fuori, e quanta parte di esso resta sotto un controllo che non dipende da me. Una cifra senza il criterio con cui leggerla non è un risultato ma materiale grezzo, e dico perciò

di ogni gruppo che cosa dimostra e dove smette di dimostrare.

### Che cosa dei requisiti il prodotto soddisfa

La tabella 3.1 elenca diciotto requisiti, undici funzionali e sette di vincolo. Il prodotto ne soddisfa diciassette. Quello che non soddisfa è RF11, il ciclo giorno/notte: l'ambiente ha l'illuminazione e i materiali che RF10 chiede, e non ha nulla che li faccia variare nel tempo. È la sola riga della tabella su cui la risposta è no, ed è la ragione per cui, derivando i requisiti, avevo tenuto RF11 distinto da RF10: su una riga sola avrei dovuto rispondere con una parola a due domande diverse. Che cosa questo significhi per l'obiettivo da cui il requisito discende lo discuto nella sezione 4.1: qui rispondo sui requisiti, che sono il termine di paragone che mi ero dato.

I sette requisiti di vincolo meritano una nota, perché non si controllano aprendo l'applicazione: nessuno di essi produce qualcosa che si veda sullo schermo. Di RV1, che chiede moduli con una sola responsabilità e il calcolo del tiro separato dal motore grafico, la prova è la separazione su cui poggia tutta la sezione 3.5: senza, quei casi di prova non esisterebbero. Di RV2, che chiede un ramo dedicato e una richiesta di incorporazione per ogni intervento di rilievo, la prova è il deposito del codice, che porta centoquarantaquattro *commit* su diciotto rami remoti, ciascuno integrato per quella via. RV3 chiede la documentazione dell'architettura e delle scelte, e le istruzioni per chi gioca: i dodici documenti di piano sono la prima, la pagina che il gioco apre da sé alla prima partita è la seconda. RV4 e RV5 li ho argomentati come regole di progetto nella sezione 3.3.2, e il servizio li applica su ogni richiesta. RV6 e RV7 li soddisfa il prodotto per il solo fatto di esistere nella forma in cui esiste.

### Quanto prodotto c'è

Il codice sorgente ammonta a circa diecimila righe. Di queste, 9.693 stanno nell'applicazione che gira nel *browser*, distribuite su sessantotto *file*: 7.129 di TypeScript, 1.477 di fogli di stile e 1.087 di marcatura. Trecentotrentasei stanno nel servizio, su otto *file*. Il rapporto fra i due numeri è esso stesso un dato, ed è

il più onesto riassunto di dove sta il peso di questo lavoro: il servizio riceve tiri, li conserva e li restituisce a chi li ha fatti, e per farlo non ha bisogno di molto codice, mentre la simulazione, la geometria del tiro, la lettura dello storico e l'interfaccia vivono nella macchina di chi gioca. I trenta moduli della figura 3.2, separati per responsabilità come RV1 chiede, sono la ragione per cui ho potuto raccontare la meccanica del tiro, la rilettura dello storico e il peso da pubblicare come tre problemi indipendenti: lo erano anche nel codice.

Accanto al codice stanno dodici documenti di piano, per 2.925 righe complessive: quasi ogni tre righe di programma ce n'è una che spiega perché quel programma è fatto così. Non è un merito in sé, ma è la condizione senza la quale questo capitolo non si sarebbe potuto scrivere.

### **Che cosa resta sotto controllo automatico**

I casi di prova sono quarantacinque, distribuiti in sei *file*: venticinque interrogano il calcolo del tiro, otto il filtro temporale della mappa, otto la stima di probabilità, quattro il resto. Accanto a essi sta il rilevatore che apre il prodotto in un *browser* vero, di cui ho accertato la capacità di fallire su tre scenari, uno integro e due guasti da me. Il numero che conta non è il totale ma la distribuzione: più della metà dei casi grava su un solo argomento, ed è quello dove un errore non si annuncia. Un tiro che parte con un'inclinazione sbagliata di due gradi non produce nessun sintomo — la palla vola, il gioco continua — e la sola cosa che possa accorgersene è un caso di prova che sappia che cosa quell'inclinazione dovrebbe valere.

Il perimetro lo dichiaro invece di presumerlo, perché è la sola cosa che lo renda utilizzabile da chi non l'ha scritto. Sotto controllo automatico stanno le funzioni che dipendono unicamente dai loro argomenti; fuori restano la scena, i materiali, le animazioni e la simulazione fisica, per la ragione detta nella sezione 3.5, e fuori resta anche il servizio, del quale questo capitolo non rende conto: la verifica che ho raccontato è quella del prodotto che gira nel *browser*, ed è là che stanno più di nove righe di codice su dieci. Di nessuna di queste componenti dichiaro una percentuale di copertura: misurata su un sistema che è per gran

parte grafico direbbe quante righe qualche esecuzione ha attraversato, non quali proprietà qualcuno ha controllato.

### **Le misure dell'intervento sul peso**

La tabella 3.8 mette a confronto i due stati del prodotto fra cui sta l'intervento della sezione 3.4.3. Le righe non hanno tutte lo stesso valore: le altre dicono di quanto una grandezza è diminuita, la prima dice che una cosa che non si poteva fare si può fare, perché un prodotto la cui compilazione di produzione fallisce non è lento da scaricare, è un prodotto che non esiste in forma distribuibile. È la sola riga per cui rivendichi più di un miglioramento.

Fra le righe numeriche non tutte pesano su chi gioca, e la distinzione l'ho argomentata dove ho ottenuto le misure. Quella che il giocatore paga davvero è il codice che il gioco chiede aprendosi, sceso da poco meno di un megabyte e mezzo trasferito a tre quarti di megabyte; il crollo del blocco iniziale è in buona parte uno spostamento di byte da un'etichetta all'altra, e a contarli è la soglia, non il giocatore; le ultime due righe non toccano chi gioca affatto, perché pesavano sulla cartella da pubblicare, e la cartella da pubblicare non si vede finché non la si misura.

Chiudo dichiarando che cosa questi numeri non sono. Non sono una misura della qualità del prodotto: dicono quanto codice c'è, quanto pesa e quanta parte è sorvegliata, e nessuna di queste tre cose garantisce che il gioco sia buono da giocare. Quello lo dice la sezione 3.6.1, e lo dice mostrando invece che contando. E non sono nemmeno un giudizio sul lavoro: il giudizio chiede di confrontare ciò che ho fatto con ciò che il piano chiedeva, e quel confronto, obiettivo per obiettivo, è il primo compito del capitolo che segue.

| Misura                                        | Prima    | Dopo      |
|-----------------------------------------------|----------|-----------|
| Compilazione di produzione                    | fallisce | passa     |
| Blocco iniziale, prodotto                     | 8,42 MB  | 272,76 kB |
| Blocco iniziale, trasferito                   | 1,46 MB  | 73,33 kB  |
| Codice richiesto aprendo il gioco             | 8,10 MB  | 2,90 MB   |
| Codice richiesto aprendo il gioco, trasferito | 1,39 MB  | 0,74 MB   |
| Codice prodotto nella cartella da pubblicare  | 13,00 MB | 3,78 MB   |
| <i>File</i> in cui è prodotto                 | 801      | 190       |
| Immagini che rivestono le superfici           | 13,42 MB | 3,68 MB   |
| Cartella da pubblicare, in tutto              | 34 MB    | 14 MB     |

**Tabella 3.8:** Il prodotto nei due stati fra cui sta l'intervento sul peso. «Prodotto» è ciò che la compilazione scrive su disco, «trasferito» ciò che arriva a chi apre la pagina, cioè lo stesso codice dopo la compressione che il *server* applica. Le due righe sul codice richiesto aprendo il gioco sono arrotondate. La prima riga non è omogenea alle altre perché non misura un grado: è la sola che cambi lo stato del prodotto.

# Capitolo 4

## Valutazione retrospettiva

### 4.1 Grado di raggiungimento degli obiettivi

La sezione 3.6.2 ha risposto sui requisiti; questa risponde sugli obiettivi, e le due risposte non sono la stessa cosa. Un requisito dice che cosa il prodotto ha, e si controlla guardandolo; un obiettivo dice anche che cosa io dovessi farne, e per verificarlo il prodotto non basta. È la ragione per cui il conto non torna uguale nei due posti: i requisiti soddisfatti sono diciassette su diciotto, gli obiettivi raggiunti per intero sei su otto, e uno dei due che restano indietro non lo si potrebbe prevedere dalla tabella 3.1, perché il requisito che ne discende è fra quelli soddisfatti.

Dichiaro lo stato di ciascuno nella tabella 4.1, con accanto l'evidenza su cui lo fondo, e qui sotto commento i gruppi invece delle righe: sugli obbligatori c'è poco da argomentare, ed è sui due raggiunti in parte che vale la pena di spendere lo spazio.

#### I quattro obbligatori

Li ho raggiunti tutti e quattro, e ne dico poco perché il capitolo 3 li ha già raccontati per esteso. O01 è il gioco che funziona: si muove il personaggio, si tira modulandone la potenza, il sistema riconosce l'esito e tiene il punteggio, e la figura 3.6 mostra l'istante in cui tutto questo si decide. O02 e O03 non producono invece nulla che si veda giocando, ed è la ragione per cui la tabella

| Codice              | Stato              | Evidenza                                                                                                                             |
|---------------------|--------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <b>Obbligatorî</b>  |                    |                                                                                                                                      |
| O01                 | Raggiunto          | RF1–RF3; la meccanica del tiro della sezione 3.4.1 e i venticinque casi di prova che la interrogano                                  |
| O02                 | Raggiunto          | RV1–RV2; i trenta moduli della figura 3.2, separati per responsabilità, e centoquarantaquattro <i>commit</i> su diciotto rami remoti |
| O03                 | Raggiunto          | RV3; dodici documenti di piano per 2.925 righe e la pagina di istruzioni che il gioco apre da sé alla prima partita                  |
| O04                 | Raggiunto          | RF4–RF5, RV4–RV5; il servizio della sezione 3.3.1 e le due regole di identità argomentate nella sezione 3.3.2                        |
| <b>Desiderabili</b> |                    |                                                                                                                                      |
| D01                 | Raggiunto          | RF6–RF7; la mappa della figura 3.7, con i due gruppi di filtri che si combinano, e otto casi di prova sul filtro temporale           |
| D02                 | Raggiunto          | RF8; la stima locale della sezione 3.4.2, con i pesi della tabella 3.5, e otto casi di prova                                         |
| <b>Facoltativi</b>  |                    |                                                                                                                                      |
| F01                 | Raggiunto in parte | RF9 soddisfatto; mia l’attrezzatura di gioco, e mio il salto perché calcolato e non registrato; adottati gli altri modelli           |
| F02                 | Raggiunto in parte | RF10 soddisfatto, RF11 no: l’ambiente ha luci e materiali propri, e nulla che li faccia variare nel tempo                            |

**Tabella 4.1:** Gli otto obiettivi della tabella 2.1, nella stessa codifica, con lo stato che dichiaro per ciascuno e l’evidenza su cui lo fondo. I codici RF e RV sono quelli della tabella 3.1. L’evidenza rimanda al capitolo 3, che la espone: qui non misuro nulla che non sia già misurato là.

ne cerca l’evidenza altrove, nella forma in cui il codice è scritto e nei documenti che lo accompagnano; della separazione in moduli aggiungo una cosa sola, che senza di essa i casi di prova della sezione 3.5 non esisterebbero. O04 è il servizio: chi gioca si registra, riceve un’identità che sopravvive alla sessione, e i suoi tiri gli tornano indietro perché il servizio li attribuisce alla credenziale firmata e non a ciò che la richiesta dichiara di essere.

### **I due desiderabili**

Li ho raggiunti entrambi, e nessuno dei due nella versione più stretta che il piano avrebbe consentito. D01 chiedeva una mappa di densità dei tiri riusciti in vista dall'alto, con filtro per intervallo temporale, e la figura 3.7 ne porta uno in più di quelli chiesti, perché all'intervallo di tempo si affianca l'esito e i due si combinano fra loro. D02 chiedeva un algoritmo di stima della probabilità di successo a partire dalla posizione e dai parametri di lancio, e di come lo calcolo rende conto la sezione 3.4.2: qui conta un solo aspetto di quel calcolo, perché è l'unico che possa mettere in dubbio l'obiettivo. La stima non la conservo in archivio e la rifaccio a ogni lettura, e non è una mancanza: la probabilità di un tiro cambia a ogni tiro nuovo, e un numero conservato sarebbe una vecchia opinione riposta accanto a un fatto.

### **F01, dove la mia parte è più stretta del prodotto**

F01 chiedeva modelli tridimensionali e animazioni di base per il personaggio e per l'ambiente di gioco, con strumenti di modellazione come Blender, e lo dichiaro raggiunto in parte: il prodotto ha tutto ciò che RF9 domanda, ma non tutto quello che ha l'ho modellato io. Ho modellato l'attrezzatura di gioco, cioè il canestro su cui tutto il resto si regge, mentre per la palla e per la vegetazione ho adottato modelli esistenti, che uso come sono e di cui riporto altrove autore e licenza; il personaggio arriva con le proprie sequenze di corsa, di salto e di tiro. Rivendico però il salto, che modellato non è: la sezione 3.4.1 lo documenta come calcolato e non registrato, ed è codice e non geometria; è la scelta che mi lascia conoscere in ogni istante la quota del rilascio, che una sequenza registrata non avrebbe saputo dare. È qui che la distinzione da cui sono partito si vede all'opera, ed è la ragione per cui un requisito soddisfatto può convivere con un obiettivo raggiunto a metà.

### **F02, l'ambiente sì e il ciclo giorno e notte no**

F02 chiedeva la personalizzazione estetica dell'ambiente, cioè luci, materiali e ciclo giorno e notte, e dei tre resta fuori l'ultimo: è RF11, la sola riga della

tabella 3.1 su cui la risposta è no. Non è che non ci abbia pensato: l'ho progettato in un documento dedicato, che ne fissa il funzionamento nel dettaglio, e non l'ho realizzato, sicché nel prodotto la direzione del sole resta una costante. La ragione è la precedenza che ho dato a rendere il prodotto distribuibile, che la sezione 3.4.3 racconta e che la tabella 3.8 misura: fra un ambiente che cambia colore nel corso della giornata e un prodotto che si possa pubblicare ho scelto il secondo, e con le 320 ore che erano la sola risorsa in gioco le due cose non stavano insieme.

### **Letti con il peso che l'azienda dava a ciascuno**

I due obiettivi raggiunti in parte sono i due facoltativi, e sono anche i due su cui il *tutor* ha dichiarato l'interesse minore: dei facoltativi scrive che interessavano molto meno, e che «sono tali proprio per questo». I due che indicava invece come i più importanti, O02 e D02, li ho raggiunti entrambi, e D02 nella forma argomentata della sezione 3.4.2 e non in quella di un numero qualunque. Non lo osservo per attenuare le due parzialità, che restano dove sono e che ho appena scritte senza sconti, ma perché ripartire gli obiettivi fra obbligatori, desiderabili e facoltativi è il modo consueto di scrivere un piano e serve esattamente a questo: a stabilire in anticipo che cosa possa cadere. Ciò che è caduto è ciò che quella ripartizione dichiarava sacrificabile, e non è un merito della mia esecuzione.

### **Gli obiettivi che rispondevano al mio interesse**

Restano i miei, che la sezione 2.5 dichiara e che le direttive chiedono di valutare accanto a quelli dell'organizzazione ospitante; del resto l'elenco è uno solo, e a dividerlo per interesse è l'ultima colonna della tabella 2.1. Il primo era costruire il gioco che volevo costruire, e il gioco esiste: la sezione 3.6.1 lo mostra da fuori, come lo incontra chi apre l'applicazione. Il secondo erano le competenze che il corso di studi non tocca, la grafica tridimensionale in tempo reale, la simulazione fisica e l'ecosistema del *web* moderno: di quelle rendono conto la sezione 4.3 e la sezione 4.4, e qui non le anticipo.

Il terzo l'ho raggiunto solo in parte, e la parte mancante insegna più di quella raggiunta. Volevo capire se il settore videoludico e la grafica tridimensionale siano un ambito in cui proseguire più avanti, e la risposta che ho è a metà. Il mondo videoludico non mi è dispiaciuto, ma questo lavoro non si è quasi mai mosso nella direzione che avrebbe risposto alla domanda: non ho progettato meccaniche di gioco, non ho costruito un racconto, non ho lavorato sulla giocabilità. Ho lavorato per dimostrare che un gioco tridimensionale nel *browser* si può fare, con un'identità per chi gioca, i suoi dati conservati e restituiti a lui soltanto, e un artefatto che si possa pubblicare: sono problemi di ingegneria del software applicati a un dominio videoludico, non i problemi del mestiere di chi i videogiochi li progetta. La domanda resta dunque aperta per la parte che riguarda quel mestiere, e questo è un esito e non un fallimento: un tirocinio è il modo meno costoso di scoprire dove si voglia andare, e ciò che ho scoperto è che la mia domanda ne conteneva due, se mi interessi il dominio e se mi interessi il mestiere che lo costruisce. Alla prima ho risposto, e la risposta non è negativa; alla seconda risponderà semmai un lavoro che di quel mestiere si occupi, perché questo non se ne è occupato.

## 4.2 Consuntivo delle ore e scostamenti dalla pianificazione

Il Piano di Lavoro fissa il preventivo, e la tabella [2.2](#) lo riporta; questa sezione gli mette accanto la spesa effettiva. Fra i due termini c'è però un'asimmetria che dichiaro prima dei numeri, perché ne governa la lettura: il preventivo è un documento, la spesa effettiva è una ricostruzione. Un registro delle ore per attività non esiste, e non esiste perché nessuno lo chiedeva: l'avanzamento si verificava una volta a settimana sulla bacheca della sezione [3.1](#), che portava gli obiettivi concordati con il segno di quelli chiusi, e un obiettivo chiuso non dice quante ore sia costato.

Ricostruisco quindi da ciò che esiste, e sono due cose. La prima è quella bacheca, che scandisce il periodo in otto settimane da quaranta ore e registra

per ciascuna che cosa vi avessi portato a termine; la seconda è la cronologia del versionamento, che dentro una settimana mostra su che cosa il lavoro stesse insistendo. Della prima dico subito il limite, perché è la fonte che sembra più forte e non lo è: quella bacheca era il Piano di Lavoro in forma di bacheca, come racconta la sezione 3.1, e perciò non poteva divergere dal piano. Conferma che il calendario ha tenuto, non come siano andate le ore dentro le settimane.

Ne segue l'approssimazione, che dichiaro invece di lasciarla intuire. La ricostruzione non scende sotto la settimana, sicché le cifre della tabella 4.2 le ho arrotondate a multipli di cinque e nessuna va letta come una misura: dentro una stessa settimana non sono in grado di separare un'ora di studio da un'ora di progettazione, e delle attività che non lasciano traccia (gli incontri, la lettura, il collaudo condotto a mano) non ho che il mio ricordo. Ciò che il confronto sostiene non è il valore di ciascuna riga: è il verso degli scostamenti e il loro ordine di grandezza.

| Attività                                                               | Preventivo | Ricostruzione | Scostamento |
|------------------------------------------------------------------------|------------|---------------|-------------|
| Formazione sulle tecnologie                                            | 44         | 45            | +1          |
| Definizione dell'architettura di riferimento e relativa documentazione | 38         | 55            | +17         |
| Sviluppo e integrazione                                                | 200        | 195           | -5          |
| Collaudo finale                                                        | 38         | 25            | -13         |
| <b>Totale</b>                                                          | <b>320</b> | <b>320</b>    | <b>0</b>    |

**Tabella 4.2:** Le quattro voci della tabella 2.2 con accanto la ricostruzione della spesa effettiva e lo scostamento fra le due. La colonna del preventivo è documentale; quella della ricostruzione è arrotondata a multipli di cinque per non attribuirle una precisione che le fonti non hanno, e lo scostamento eredita quella stessa approssimazione. Il totale è vincolato: le 320 ore sono quelle e la ricostruzione le ripartisce, non le ricalcola.

### Le due righe che tengono

La formazione torna, e la bacheca lo mostra meglio di quanto sappia fare una cifra: i quattro blocchi di studio stanno nelle quattro settimane in cui il piano li colloca, ciascuno con la tecnologia che quella settimana introduce, e risultano chiusi tutti e quattro. Lo sviluppo e l'integrazione restano in quota, ma è la

riga di cui il totale dice meno, perché è quella che il piano stesso aveva lasciato più indeterminata: la sezione 2.4 osservava che il modulo tridimensionale non ha una voce propria e divide le ottanta ore con le interfacce, e il consuntivo chiude quel cerchio nel modo che ci si poteva attendere. La parte priva di un preventivo isolabile è quella che al proprio interno ha assorbito di più, e non perché sia andata storta: perché era la meno nota delle tre, ed è il genere di lavoro che un preventivo non sa dimensionare finché non lo si è fatto una volta.

### **La documentazione, contata una volta e richiesta ogni volta**

Il preventivo assegna quattro ore alla stesura della documentazione di analisi e progettazione, dentro una voce che di ore ne vale trentotto. I documenti di piano sono dodici e la sezione 3.6.2 li conta in più di duemila righe: in quattro ore non ci stanno, e soprattutto non stanno dove il piano li colloca. Il piano li tratta come un adempimento da sbrigare in coda all'analisi, mentre nel lavoro sono diventati una disciplina ricorrente, uno prima di ogni intervento di qualche peso, secondo la regola che la sezione 3.1 enuncia. Lo scostamento non è dunque un eccesso di zelo né un ritardo: è una voce collocata nel posto sbagliato, che il piano contava una volta e che il lavoro ha richiesto ogni volta.

### **Il collaudo ha ceduto ore a un'attività che il piano non nominava**

Le trentotto ore del collaudo finale non sono andate tutte al collaudo, ed è questo lo scostamento che vale la pena di discutere. Verso la fine del periodo è entrata un'attività che il piano non prevede in nessuna delle quattro voci e nemmeno fra quelle di dettaglio: rendere il prodotto distribuibile. Non era rifinitura, era una condizione che mancava, e la sezione 3.6.2 lo dice nei termini più netti che ho saputo trovare, cioè che un prodotto la cui compilazione di produzione fallisce non è un prodotto lento da scaricare ma un prodotto che in forma distribuibile non esiste. Quelle ore, non essendoci una voce da cui prenderle, le ha date la voce contigua, che è appunto il collaudo finale.

### **Un costo che si presenta tutto in fondo**

Non lo osservo per rimproverare il piano, che di quel costo non poteva avere notizia, ma perché il modo in cui si è presentato non riguarda soltanto questo lavoro. Finché si costruisce, si prova e si corregge, ogni cosa si può fare per intero sulla macchina su cui il programma si scrive, e niente in quel percorso obbliga a chiedersi se il prodotto sappia formarsi fuori di lì. Quella domanda si pone una volta sola, e porta con sé tutto il proprio costo in una volta: non distribuito lungo le settimane, ma raccolto in fondo, dove di settimane ne restano meno. È in questo senso che una voce simile diventa visibile solo in prossimità del rilascio, e che un preventivo scritto all'inizio difficilmente può contenerla.

## **4.3 Conoscenze e abilità acquisite**

Delle tecnologie con cui ho lavorato rende conto il capitolo 3, che le mostra mentre servono a qualcosa; ripeterne qui l'elenco non aggiungerebbe nulla. Raccolgo invece quattro cose che restano quando gli strumenti cambiano, ciascuna nata da un episodio che il capitolo 3 ha già raccontato e a cui rimando. Tre delle quattro le ho imparate sbagliando.

### **Su una soglia si misura, non si stima**

La sezione 3.4.3 racconta una diagnosi sbagliata: incolpavo del peso i fogli di stile dei componenti, ed è bastato eseguire la compilazione per vedere che non c'entravano. Il punto non è il bersaglio sbagliato: è che quel peso non l'avevo mai misurato, l'avevo immaginato, e adoperavo quel numero come se me lo avesse dato uno strumento. Una stima fatta a mente non è una misura meno precisa: è una misura che non c'è. Enunciata così la distinzione sembra ovvia, quasi scontata, e infatti non l'ho imparata enunciandola.

La regola ha una seconda metà, che riguarda lo strumento. Le quattro varianti le ho pesate con lo stesso compilatore con cui il progetto si costruisce e chiedendo esattamente i simboli che il gioco adopera: un altro strumento mi avrebbe dato un numero altrettanto esatto, ma di un'altra domanda.

Alla stessa cautela appartiene un episodio che con le misure non ha nulla a che vedere. Nella sezione [3.4.1](#) ho cercato a lungo la coppia di valori, attrito e rimbalzo, che liberasse una palla rimasta ferma sul ferro: quella coppia non esisteva, perché a tenerla ferma non era l'attrito ma la forma dell'appoggio. Non c'era alcun numero da misurare, e lo sbaglio stava un gradino prima: conviene chiedersi se il problema sia di valori, prima di cercare il valore giusto.

### **Il costo di un guasto dipende da quanto tarda ad annunciarsi**

Dalla sola modifica che la sezione [3.4.3](#) racconta sono usciti cinque difetti, e tutte e cinque le volte il compilatore ha approvato il codice. Il più insidioso era l'assenza del componente che coordina le collisioni: la scena si apriva intatta, e che mancasse qualcosa si vedeva soltanto muovendo il giocatore. Un errore di compilazione lo si legge e lo si corregge in un minuto; quei cinque li ho cercati a lungo, e non perché fossero più gravi, ma perché nessuno li segnalava.

Non ne ricavo di diffidare degli strumenti, ma di sapere che cosa ciascuno guarda: un'importazione che manca non cambia il risultato di alcuna funzione, quindi né il compilatore né i *test* di unità potevano vederla, e l'unico posto in cui quel guasto si annuncia è la console del *browser*. Il rilevatore della sezione [3.5](#) è lì per leggerla. E prima di fidarmene ho guastato il prodotto di proposito, per vedere se se ne accorgeva: di un controllo conta ciò che avrebbe fermato, ed è una cosa che si prova, non che si presume.

C'è poi un secondo modo di nascondersi, e non è il silenzio: è la somiglianza. Uno dei cinque mostrava una scena a metà, con alcune superfici al loro posto e altre no, che è l'aspetto di un problema di immagini mancanti; erano invece gli *shader* a non arrivare più. La spiegazione che avevo pronta mi ha fermato prima della causa vera, ed è il modo più caro che conosca di perdere tempo.

### **La verificabilità si decide progettando, non collaudando**

È l'unica delle quattro che non ho imparato sbagliando. Quando la sezione [3.5](#) ha dovuto stabilire che cosa fosse provabile, la risposta l'avevo già scritta altrove: il calcolo del tiro si interroga passandogli tre numeri perché un vincolo

della sezione 3.3.2 lo tiene fuori dal motore grafico. Se l'avessi scritto dentro il ciclo di *rendering*, per provarlo avrei dovuto aprire una scena e disporre di una scheda video che la disegnasse, e nessun lavoro successivo me lo avrebbe risparmiato. La separazione si difende di solito in nome dell'ordine e della leggibilità, che sotto pressione sono le prime cose che si concedono; qui ha deciso che cosa avrei potuto provare e che cosa no.

La stessa scelta stabilisce anche che cosa resta fuori: la scena, i materiali e la simulazione fisica non li sottopongo a prova, non perché continuo meno, ma perché per provarli dovrei riprodurre il motore e finirei per misurare quello.

### **Un compromesso dichiarato è conoscenza; taciuto, è un'insidia**

Il quarto è il meno tecnico ed è quello che ho applicato più spesso. Caricare le rotte a richiesta fa passare la soglia sul blocco iniziale e non rende più rapido il sito per chi arriva dalla radice: la sezione 3.4.3 lo scrive, e scriverlo mi è costato un risultato che sarebbe stato comodo rivendicare. Il rilevatore della sezione 3.5 porta annotato in testa che carica la scena e non gioca, cioè che non vedrebbe il più insidioso dei cinque. La stima della sezione 3.4.2 accompagna con un'avvertenza i numeri che pochi tiri sostengono. E la sezione 4.2, in questo stesso capitolo, dichiara di quale approssimazione siano le proprie cifre prima di esporle.

Ogni volta il limite lo conoscevo io e nessun altro, e soltanto mentre lo accettavo. Tacerlo non lo toglie: lo lascia trovare a chi si fida, più tardi e a caro prezzo. Scriverlo costa tre righe e rende controllabile ciò che altrimenti si può soltanto credere.

Le quattro cose non sono indipendenti: sono quattro modi di tenere separato ciò che so da ciò che suppongo. Un numero misurato da un numero immaginato, un programma che ho provato da uno che nessuno strumento ha ancora contraddetto, e la riga che dice dove passa il confine fra i due. È la parte di questo lavoro che mi aspetto di riusare dove non ci sarà né un gioco né un *browser*.

## 4.4 Distanza fra le competenze richieste e quelle erogate dal corso

Il confronto che chiude il capitolo mette da una parte ciò che il corso di studi mi aveva dato e dall'altra ciò che questo lavoro ha richiesto, e una cosa va detta prima, perché non sembri un conto da presentare: una laurea triennale in informatica non promette la grafica tridimensionale né lo sviluppo di videogiochi, e il Piano di Lavoro lo dava per scontato, tanto da riservare alla formazione sulle tecnologie quarantaquattro delle 320 ore. Distinguo perciò tre casi: ciò che il corso mi ha dato, ciò che ho imparato da zero perché nessuno aveva promesso di darmelo, e un terzo che non è una tecnologia.

### Ciò che il corso ha dato, e che ho adoperato

La parte che ha retto meglio non è una tecnologia, è un metodo. Il capitolo 3 è costruito come l'ingegneria del software mi ha insegnato a costruirlo: gli obiettivi diventano requisiti nella sezione 3.2 perché «fatto» abbia un significato controllabile, verifica e validazione restano due domande distinte nella sezione 3.5, e ogni scelta porta accanto la ragione per cui l'ho presa.

Il resto sono nozioni che ho riconosciuto mentre le usavo. La programmazione a oggetti sta nella separazione in moduli e nella regola del proprietario unico della sezione 3.4.1. Delle basi di dati ho adoperato la nozione più elementare e più utile, cioè che l'identità di un *record* la assegna l'archivio e non chi lo scrive, ed è uno dei tre vincoli della sezione 3.3.2. Di tecnologie *web* ho preso le nozioni che hanno dato forma al servizio: le quattro rotte della tabella 3.2 sono un'interfaccia REST come la si disegna a lezione.

### Ciò che non c'era, e che nessuno aveva promesso

Tre cose le ho imparate da zero. La grafica tridimensionale in tempo reale, cioè che cosa sia una *mesh*, che cosa faccia uno *shader* e perché una scena si disegni in più passate: la mappa della sezione 3.4.2 ne è la conseguenza più visibile, perché non è fatta di oggetti ma di due passate su una sola immagine. La

simulazione fisica, e con essa la distinzione fra ciò che governa una formula e ciò che governa un motore, che la sezione 3.4.1 racconta per intero. E l'ecosistema del *web* moderno, che è il più largo dei tre: come si compila un'applicazione, che cosa significhi che un pacchetto dichiara i propri effetti collaterali, perché due modi di scrivere la stessa importazione producano artefatti che differiscono di alcuni megabyte.

Il corso mi aveva però insegnato come si affronta una tecnologia che non si conosce. Davanti al peso del prodotto ho misurato quattro varianti prima di riscrivere e ho messo per iscritto anche le cause che avevo escluso, invece di procedere per tentativi: le tecnologie erano nuove, il modo di affrontarle no.

### **Ciò che non avevo previsto: dal programma al prodotto**

Il terzo caso è di natura diversa, e la sezione 4.2 lo ha già incontrato dal lato delle ore. Dei programmi che avevo scritto sapevo se funzionassero, non quanto pesassero né come si consegnassero, e la sezione 3.4.3 racconta la prima volta che ho dovuto chiedermelo. È la sola cosa che aggiungerei al percorso di studi, e non sarebbe un corso di grafica: basterebbe che di un programma corretto venisse chiesto, qualche volta, anche come lo si consegna.

La distanza che ho trovato non è insomma fra ciò che il corso insegna e ciò che un'azienda adopera, che è un divario previsto e che si colma studiando: è fra un programma che funziona e un prodotto che esiste anche fuori dalla macchina che lo ha scritto. Che a mostrarmela sia stato un tirocinio non mi pare un caso, perché è il primo posto in cui mi è stato chiesto un prodotto e non un programma.

Concludo dicendo quindi che lo *stage* è stata per me una forte esperienza di apprendimento, e che la differenza fra ciò che il corso di studi mi ha dato e quello che il progetto raccontato mi ha richiesto non è un divario fra università e azienda, ma fra un programma che fornisce certe risposte, e un prodotto che ne fornisce altre.



# Acronimi e abbreviazioni

**HTML** HyperText Markup Language.

**HTTP** HyperText Transfer Protocol.

**ISO** International Organization for Standardization.

**REST** Representational State Transfer.

**RF** Requisito Funzionale.

**RV** Requisito di Vincolo.

# Glossario

**capsula** Primitiva geometrica formata da un cilindro chiuso alle due estremità da una semisfera, descritta da un raggio e da un'altezza. È la forma con cui si rappresenta un corpo allungato e all'incirca verticale, come una persona, quando serve soltanto sapere che cosa urta: non ha spigoli su cui un contatto possa incastrarsi, e interrogarla costa una frazione di ciò che costerebbe interrogare la superficie visibile dello stesso oggetto.

**credenziale firmata** Documento che un servizio rilascia a chi ha superato l'accesso e sigilla con una chiave che non lascia il servizio stesso. Contiene l'identità di chi lo ha ottenuto e viaggia insieme a ogni richiesta successiva; la firma permette al servizio di riconoscere di averlo emesso lui e che nessuno ne ha alterato il contenuto per strada. Il suo effetto di progetto è sottrarre l'identità alla scelta di chi chiede, perché il servizio la legge dalla firma e non dalla richiesta.

**enterprise** Detto di un'organizzazione abbastanza grande e strutturata da avere già un proprio sistema informativo. Chi commissiona software in questo contesto non cerca un programma a sé stante, ma una parte che si innesti in ciò che ha già, ne rispetti i vincoli di sicurezza e di continuità del servizio e resti manutenibile nel tempo.

**forma chiusa** Detto di una formula che, dato un ingresso, restituisce direttamente il risultato, invece di cercarlo per approssimazioni successive o di ricavarlo dal valore precedente. Applicata al moto di un corpo, dice dove esso si trova a un istante qualunque senza sapere dove fosse all'istante

prima: ne segue una traiettoria esatta, priva dell'errore che ogni calcolo passo passo accumula, ma anche incapace di rispondere a un urto.

**mappa di densità** Immagine a colori sovrapposta a una pianta, in cui l'intensità di ogni punto misura quanti eventi sono avvenuti lì. Rende visibile una distribuzione che un elenco non mostrerebbe. La scala di colore può essere assoluta oppure relativa al massimo di ciò che si sta guardando: nel secondo caso la mappa si legge fin dalla prima apertura, ma due mappe diverse non sono confrontabili fra loro.

**mesh** Guscio che descrive la superficie di un oggetto tridimensionale triangolo per triangolo, ed è la forma in cui i modelli si distribuiscono e si disegnano. Nata per essere guardata, non è garantita né chiusa né coerente: può avere facce orientate al contrario, spessore nullo o buchi che da fuori non si vedono, difetti innocui per il disegno e non per il calcolo delle collisioni.

**minificazione** Riscrittura automatica di un programma che ne accorcia i nomi interni e ne toglie tutto ciò che serve soltanto a chi lo legge, spazi e commenti compresi, senza cambiare ciò che il programma fa. Riduce il peso del risultato e lo rende illeggibile a una persona, il che è senza conseguenze, perché a quel punto il destinatario è una macchina.

**primitiva geometrica** Forma elementare descritta da pochi numeri, come una sfera, un riquadro o una capsula, usata al posto della superficie visibile di un oggetto per calcolarne gli urti. Resta invisibile, e il modello visibile la segue senza partecipare al calcolo. Il prezzo della sostituzione è che le due forme possono divergere, e a tenerle allineate è chi le governa.

**regressione locale** Metodo di stima che, per ogni osservazione, pesa tutte le altre in proporzione a quanto le somigliano, invece di raggrupparle in classi dai confini stabiliti in anticipo. Evita che due osservazioni quasi identiche ricevano stime molto diverse per il solo fatto di cadere ai due lati di una linea, e in cambio chiede di dichiarare che cosa conti come vicino su ciascuna dimensione.

**rendering** Calcolo con cui una scena tridimensionale diventa l'immagine bidimensionale che compare sullo schermo. In un gioco si ripete di continuo, e il ritmo a cui si ripete fissa il tempo entro cui ogni altro calcolo deve stare: a sessanta immagini al secondo restano poco più di sedici millesimi di secondo per ciascuna.

**shader** Programma che il motore grafico fa eseguire alla scheda video per calcolare il colore dei punti di un'immagine. Non viene compilato insieme al resto dell'applicazione ma preparato quando serve, e questo lo rende una dipendenza che può mancare a programma già avviato, con un guasto che si manifesta come una scena incompleta anziché come un errore.

**smoke test** Controllo automatico che avvia il prodotto per intero e verifica soltanto che non dia segni di guasto evidenti, senza esercitarne le funzioni. Il nome viene dalla prova di un impianto appena acceso e ne dichiara da sé l'ambizione: un rilevatore di fumo dice che qualcosa brucia, non che l'impianto funziona. Il suo valore dipende dall'essere stato visto fallire almeno una volta.

**soglia di compilazione** Limite di peso che si dichiara al compilatore per ciò che esso produce. Di norma se ne fissano due, una che avverte e una che ferma: superata la seconda la compilazione esce in errore e non scrive nulla, sicché il prodotto non esiste in forma distribuibile, per quanto bene funzioni sulla macchina su cui è stato scritto.

**texture** Immagine applicata alla superficie di un oggetto tridimensionale per darle un aspetto. Una stessa superficie ne riceve di norma più d'una, ciascuna responsabile di una proprietà diversa, come il colore, la ruvidità o il modo in cui la luce la colpisce. Sono spesso la parte più pesante di ciò che va distribuito.

**tree-shaking** Eliminazione dal risultato della compilazione del codice che nessuno adopera. Funziona soltanto dove il compilatore ha modo di stabilire che togliere un *file* sia sicuro: un *file* con effetti collaterali cambia qualcosa

nel programma per il solo fatto di essere caricato, quindi il compilatore per prudenza lo tiene, e con esso tutto ciò che si porta dietro.

**zona del *framework*** Contesto di esecuzione entro il quale un *framework* sorveglia le operazioni asincrone, per sapere quando lo stato dell'applicazione possa essere cambiato e quindi quando convenga ridisegnare l'interfaccia. Ciò che gira dentro la zona costa un ciclo di controllo a ogni operazione conclusa; ciò che gira fuori non ne costa nessuno e, per la stessa ragione, non aggiorna l'interfaccia da sé.

# Bibliografia

Il tutor aziendale. «Risposte scritte alle domande sull'organizzazione, sui processi e sugli obiettivi del tirocinio». Comunicazione personale, riportata e citata con l'autorizzazione espressa dell'autore. Non pubblica. 2026.

Sync Lab S.r.l. «Piano di Lavoro per il tirocinio curriculare di Visentin Giovanni». Documento concordato e sottoscritto dalle parti prima dell'avvio del tirocinio. Non pubblico. 2026.

# Sitografia

- ambientCG. *PaintedPlaster017 e PavingStones109*. Mappe di colore, rugosità, normali e occlusione ambientale impiegate per le superfici della scena. URL: <https://ambientcg.com/> (visitato il giorno 11/09/2026).
- Angular. *Angular. Documentazione ufficiale*. URL: <https://angular.dev/> (visitato il giorno 11/09/2026).
- Babylon.js. *Babylon.js Documentation*. URL: <https://doc.babylonjs.com/> (visitato il giorno 11/09/2026).
- Creative Commons. *Attribution 4.0 International (CC BY 4.0)*. URL: <http://creativecommons.org/licenses/by/4.0/> (visitato il giorno 11/09/2026).
- Daniel Petrov. *Stylized Bush*. Modello tridimensionale, licenza CC BY 4.0. URL: <https://sketchfab.com/3d-models/stylized-bush-9d9ce79d3ae040619e96d5b22c7de1a6> (visitato il giorno 11/09/2026).
- daniel.2U. *Pine Tree Low Poly*. Modello tridimensionale, licenza CC BY 4.0. URL: <https://sketchfab.com/3d-models/pine-tree-low-poly-9cde1025662e4954b1f803c2d4bb719d> (visitato il giorno 11/09/2026).
- Ghalek. *Basket Ball*. Modello tridimensionale, licenza CC BY 4.0. URL: <https://sketchfab.com/3d-models/basket-ball-7d185fa512604b3aaeca8a820c500661> (visitato il giorno 11/09/2026).
- Manifesto Agile*. URL: <http://agilemanifesto.org/iso/it/> (visitato il giorno 27/08/2026).

MongoDB. *MongoDB Documentation*. URL: <https://www.mongodb.com/docs/> (visitato il giorno 11/09/2026).

Mongoose. *Mongoose. Modellazione dei dati per MongoDB*. URL: <https://mongoosejs.com/docs/> (visitato il giorno 11/09/2026).

OpenJS Foundation. *Express. Framework per applicazioni web su Node.js*. URL: <https://expressjs.com/> (visitato il giorno 11/09/2026).

— *Node.js. Documentazione ufficiale*. URL: <https://nodejs.org/> (visitato il giorno 11/09/2026).

Sahir Virmani. *Giant Low Poly Tree*. Modello tridimensionale, licenza CC BY 4.0. URL: <https://sketchfab.com/3d-models/giant-low-poly-tree-acfd2b7f80894848b56c2ac8e7e59572> (visitato il giorno 11/09/2026).