

UNIVERSITÀ DEGLI STUDI DI UDINE

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Triennale in Informatica

Tesi di Laurea

**IL GIOCO “FORZA 4!”:
IMPLEMENTAZIONE
DELL’ALGORITMO MINIMAX
IN UN AMBIENTE GRAFICO 3D**

Relatore:

Prof. AGOSTINO DOVIER

Laureando:

FRANCESCO ANDREUSSI

ANNO ACCADEMICO 2015-2016

Indice

1	Introduzione	v
2	L'Interfaccia Grafica Tridimensionale	1
2.1	Strumenti usati	1
2.2	Basi teoriche e tecniche di realizzazione	2
2.2.1	La scena	2
2.2.2	La rappresentazione di oggetti 3D	4
2.2.3	La pipeline grafica	4
2.2.4	Lo skybox	5
2.2.5	Lo shading e il postprocessing	7
2.2.6	Il sistema particellare	16
2.2.7	I quaternioni	19
2.2.8	Le ombre	21
3	Rappresentazione della conoscenza e Ragionamento Automatico	23
3.1	Strumenti usati	23
3.2	Basi teoriche	24
3.2.1	La programmazione a vincoli	24
3.2.2	La programmazione logica	25
3.2.3	I solver	26
3.2.4	L'algoritmo minimax	28
3.2.5	Rappresentazione della conoscenza nel caso di studio	31
3.3	Tecniche di realizzazione	32
3.3.1	Il modello MiniZinc	32
3.3.2	Il modello ASP	44
3.3.3	L'algoritmo JavaScript	51
4	Conclusioni	61
5	Ringraziamenti	63

Capitolo 1

Introduzione

Questo elaborato è il risultato dell'incontro tra grafica tridimensionale e ragionamento automatico in uno degli ambiti più conosciuti e amati dell'informatica: lo sviluppo di programmi videoludici.

I videogiochi sono una grande passione per tantissime persone in tutto il mondo e il settore è in continua espansione, grazie anche alle nuove frontiere come il mobile gaming, la Virtual e la Mixed Reality, che hanno mosso nel solo 2016 circa 42,4 miliardi di dollari (40,6 il mercato già affermato dei giochi per dispositivi mobili e 1,8 quello in via di sviluppo delle realtà virtuale e mista) [4]. Io in primis sono un appassionato videogiocatore, in particolare di FIFA, il più famoso gioco di simulazione di calcio in commercio ed è anche a causa di questo mio interesse che mi sono avvicinato all'informatica: per imparare a capire e a creare qualcosa che possa coinvolgere, e divertire, giovani e meno giovani attraverso l'uso di tecniche sempre in evoluzione. Questo è evidente in particolar modo nei due campi che andrò ad analizzare nella mia tesi: la Grafica 3D Interattiva, in grado di visualizzare ambienti virtuali sempre più accattivanti e realistici, e il Ragionamento Automatico, che fa prendere ad una macchina decisioni il più possibile vicine a quelle di un giocatore umano. Sia il rendering grafico che il ragionamento automatico, in un'applicazione interattiva, devono essere computazionalmente efficienti e non richiedere quantità di memoria troppo ingenti per essere eseguiti, altrimenti si rischia di compromettere l'esperienza di utilizzo dell'utente.

Infatti, cedendo alle mie inclinazioni, ho frequentato i corsi di Interactive 3D Graphics e Automated Reasoning, nonostante non fossero inclusi nel mio piano di studi, proprio per poter dedicare del tempo all'approfondimento di tali discipline, poi proseguito nella preparazione del lavoro di tesi che esporrò nelle prossime pagine.

Grazie appunto alle conoscenze acquisite nel primo ho potuto realizzare un'ambiente tridimensionale che, seppur limitato nella resa grafica per poter man-

tenere un buon frame-rate anche su elaboratori non molto performanti (senza una scheda grafica dedicata), mostra molti elementi interessanti e abbastanza avanzati; invece, con le nozioni apprese nel secondo corso sono riuscito ad analizzare varie tecniche di implementazione di algoritmi di ragionamento automatico, che permettono di eseguire scelte a partire da una rappresentazione numerica della situazione reale.

Nel proseguo della tesi presenterò le tecniche, gli strumenti, i problemi e le relative soluzioni in merito allo sviluppo della versione virtuale di uno dei giochi da tavola più conosciuti e apprezzati del mondo: “Forza 4!”.

Capitolo 2

L’Interfaccia Grafica Tridimensionale

Ogni programma scritto con l’intenzione di essere fruibile da degli utenti deve possedere un’interfaccia, per questo progetto ho deciso di creare un ambiente virtuale grafico sfruttando gli strumenti e le conoscenze teoriche che esporrò in questo capitolo.

2.1 Strumenti usati

Per la realizzazione della grafica tridimensionale ho utilizzato:

- **three.js**: libreria JavaScript che sfrutta le potenzialità di WebGL, la riduzione web della maggiore libreria grafica per importanza: OpenGL, la quale, in alternativa a Direct3D/DirectX di Microsoft, è utilizzata da ogni applicativo eseguibile, che presenti una grafica tridimensionale;
- **GLSL (OpenGL Shading Language)**: è, insieme al linguaggio proprietario di Microsoft HLSL (High Level Shader Language), il linguaggio di programmazione principe per le schede grafiche e la sua sintassi è C-like, proprio come HLSL e Cg (C for graphics) che è un’ulteriore alternativa sviluppata dal produttore di GPU nVidia.

Il principale vantaggio rispetto al suo “concorrente” è la sua compatibilità anche con i sistemi operativi UNIX-based, come GNU/Linux e macOS. È utilizzato per creare shader, cioè programmi che computano gli effetti di una sorgente luminosa virtuale sulla superficie di un oggetto 3D applicandogli, se si vuole, anche una o più texture (approfondirò l’argomento più avanti);

- **blender:** programma open source di modellazione con il quale ho creato alcuni semplici modelli 3D che ho usato per arricchire la scena;

2.2 Basi teoriche e tecniche di realizzazione

In questa sezione mi accingo a riassumere le teoria matematica ed i meccanismi che supportano le interfacce grafiche tridimensionali, concentrandomi su quelli da me usati per questo progetto.

Durante e in seguito al corso di Interactive 3D Graphics ho modellato gli oggetti che potete vedere nella scena, i più semplici (la maggior parte) sono stati realizzati direttamente in three.js (esempio codice), alcuni creati con blender e poi importati e altri, i più complessi, scaricati da internet, a volte riadattati, sempre con blender e poi importati su three.js.

Dopo aver popolato la scena e creato lo skybox circostante ho assegnato i materiali ai vari oggetti usando shader GLSL visti a lezione, assegnando opportuni valori ai vari parametri per ottenere effetti il più vicino possibile al vero. Questo è stato possibile grazie allo sfruttamento anche di color e normal maps, e ho implementato le animazioni come le transizioni della camera e degli oggetti, dapprima tramite l'uso di trasformazioni lineari e rotazioni semplici, poi rispettivamente sostituite dai più raffinati comandi della libreria JavaScript tween.js e dai quaternioni.

2.2.1 La scena

Lo scenario è così composto: al centro di una stanza si trovano un tavolo circolare con due sedie e sopra il tavolo ci sono le pedine e il supporto del gioco “Forza 4!”, il resto della stanza è decorato con un mobile basso (stile buffet) sovrastato dal dipinto di Piet Mondrian *“Composizione II in rosso, blu, nero e giallo”* (1929), da una porta in legno e da una finestra che lascia vedere l’ambiente esterno costituito da uno skybox cubico a cui ho applicato una texture apposita che è fatta in modo da “mascherare” il cubo dando l’effetto che non ci siano spigoli e vertici. Le fonti luminose sono una luce direzionale che simula l’illuminazione solare, la luce di una plafoniera che si trova esattamente sopra il tavolo e una lampada a terra con un lungo stelo nero metallico e il vetro rosso satinato posizionato alla destra dell’osservatore rispetto a quadro e buffet.

Il supporto verticale e le pedine che servono per giocare hanno come materiale uno ShaderMaterial che gli assegna un “effetto plastica” grazie alla composizione di un microfacet e di un lambert shader (posta nel fragment shader). La composizione avviene attraverso l’operatore mix di glsl (interpolazione lineare

tra due valori) che prende come valore interpolante una uniform (una valore di GLSL costante per ogni vertice di un oggetto) che varia da materiale a materiale. Infine ho aggiunto (mediante una somma) un colore ambientale per rendere la zona non illuminata meno scura e il materiale più realistico, inoltre avere la parte in ombra di colore nero rendeva il gioco difficilmente utilizzabile poiché si distingueva difficilmente la differenza tra pedine rosse e gialle durante il gioco.

Il materiale del tavolo (un legno lucido) è stato realizzato in maniera analoga al primo, ma con un diverso fragment shader, in grado di gestire le texture. Il piedistallo dello stesso, invece è realizzato con un colore grigio/nero uniforme con una prevalenza di componente lambert rispetto a quella microfacet.

Il materiale di sedie, piedistallo del tavolo, mobile e cornice è un legno con una laccatura grigio-nera semi-lucida, mentre quello di porta e finestra è un legno scuro, meno lucido rispetto al tavolo realizzato con lo stesso shader, ma con valori e texture diverse. Porta e finestra condividono anche il materiale delle proprie maniglie: materiale realizzato con gli stessi shader degli altri materiali con colore uniforme, ma in questo caso non è presente la componente lambertiana, infatti l'effetto voluto è quello di un materiale metallico che ricordi l'ottone.

Muri, soffitto, tetto, e la parte di casa sotto il pavimento presentano anch'essi un comportamento "lambertiano". Questi elementi hanno un ambient color grigio medio e tutti, tranne il tetto a cui è applicata una texture, sono totalmente bianchi.

Il pavimento, invece, ha un effetto legno trattato, ma in maniera più lieve rispetto a quello del tavolo; infatti in questo caso si nota la rugosità (ottenuta tramite l'applicazione di una normal map) del materiale che si presenta comunque semi-lucido.

Tutte le texture sono applicate in maniera ripetuta alle superfici, infatti nello shader, nel comando texture2D che prende in input una texture e una coppia di coordinate uv (una coppia di float compresi tra 0 e 1) modifico le coordinate uv passate moltiplicandole prima per una potenza di due passata come uniform e poi passando a texture2D il resto della divisione tra il risultato della moltiplicazione e 1, cosicché, al variare della uniform, si possa modificare il numero di volte che la texture viene ripetuta.

I vetri delle lampade hanno una forte componente ambientale che li rende molto luminosi e il loro materiale è ottenuto sempre attraverso i soliti shader, ma in questo caso la componente lambertiana è nulla e l'alfa molto elevata (200) fa in modo che siano talmente lucidi che sembra addirittura si lascino attraversare dalla luce (nei prossimi mesi esplorerò soluzioni più raffinate).

Come già accennato le animazioni sono piuttosto semplici e si dividono in due

tipologie: transazioni (implementate appoggiandosi alle funzioni della libreria tween.js) e rotazioni (gestite tramite quaternioni).

Ho usato le transazioni in maniera piuttosto diffusa infatti vengono utilizzate per far muovere la camera durante i cambi di gioco e ai comandi play/pause/-stop, inoltre viene portata una pedina sopra il supporto del gioco e poi girata di 90° per metterla in posizione verticale, mediante l'uso di un quaternione. Per giocare l'utente può selezionare la colonna spostando la pedina a destra e a sinistra e poi, quando la lascia cadere la pedina è animata con un tweening con easing di tipo bounce.out per dare un effetto fisico di caduta e collisione che però non è calcolato in base alle leggi della fisica, bensì trovando la prima riga libera della colonna selezionata e ponendo come target le coordinate corrispondenti alla coppia (riga, colonna) selezionata. Durante la traslazione del cambio del turno viene effettuata in automatico da three.js anche una rotazione di 180° poiché la camera altrimenti inquadrerebbe la scena sottosopra.

Ho inserito anche un'opzione di post-processing attivabile con un che abilità/disabilità un effetto negativo, che cioè mostra a video il colore contrario a quello normalmente utilizzato, l'effetto è volutamente semplice così da mantenere il gioco a 60 fps.

Infine, quando si vince si potrà apprezzare una scritta dello stesso colore e materiale delle proprie pedine che annuncia la sua vittoria e un “effetto coriandoli” ottenuto grazie ad un sistema particellare.

2.2.2 La rappresentazione di oggetti 3D

La prima questione da affrontare nella grafica 3D è sicuramente quello di avere una struttura dati per memorizzare e poter processare oggetti tridimensionali su un calcolatore.

La soluzione universalmente adottata è quella di approssimare, semplificando le superfici continue reali in tante piccole superfici triangolari affiancate le une alle altre. La domanda sorge spontanea: perché proprio triangolari? Perchè dati tre punti nel piano si pu'o sempre determinare in modo univoco il triangolo del quale sono i vertici e perchè dall'accostamento di vari triangoli si possono creare tutte le ltre figure piane e solide (con un certo grado di approssimazione). Un solido, poi, è ottenuto dalla giustapposizione di più superfici e di conseguenza all'interno è vuoto, ma ciò non comporta grossi problemi perchè chi guarda un oggetto ne apprezza la superficie e non quello che c'è al di sotto.

2.2.3 La pipeline grafica

Altro grande problema posto dal rendering di uno scenario grafico che ha l'obiettivo di essere interattivo, e quindi anche quello appena descritto, è di

sicuro quello di eseguire un elevato numero di calcoli, per poter computare i “fotogrammi” (frame) da far vedere sullo schermo, molte volte al secondo così da mantenere la frequenza dei fotogrammi (frame-rate) abbastanza alta affinché l’occhio umano percepisca fluidità nel movimento: almeno 30 frame al secondo (frames-per-second, abbreviato praticamente sempre in fps), ma è preferibile cercare di averne 60 sia per un’immagine più fluida, sia per evitare cali drastici e fastidiosi (a 15 fps o meno) del frame-rate che potrebbero essere causati da un sovraccarico, anche solo momentaneo, della GPU (Graphics Processing Unit). Questo problema viene risolto dall’utilizzo di processori dedicati e altamente parallelizzati che riescono a svolgere in maniera assai efficiente un numero incredibilmente alto di operazioni semplici e simili fra loro come quelle riguardanti il rendering 3D e dalla scomposizione dell’operazione di rendering in passaggi più agevoli che costituiscono la pipeline grafica.

La pipeline grafica (Figura 2.2) è costituita da due macro stadi: il *geometry stage*, che si occupa di mappare i vertici degli oggetti presenti nella scena nelle coordinate dello schermo (dove dovranno essere visualizzate alla fine del rendering) e di eseguire operazione sui poligoni e sui vertici in input e il *rasterizer stage*, il quale si occupa di assegnare ad ogni pixel dello schermo un colore derivandolo dalle proprietà degli elementi che gli vengono passati dallo stadio precedente, il suo output è il fotogramma da far vedere sullo schermo.

I vari passaggi della pipeline possono essere classificati anche a seconda del loro grado di programmabilità: ci sono due stadi che non possono essere personalizzati (il Geometry shader, che si occupa principalmente dello Screen mapping e la Rasterization, nel quale vengono interpolati i valori degli attributi dei vari vertici assegnando ad ogni frammento i suoi attributi), due sono configurabili ma non programmabili (Shape assembly, che serve a appunto ad assemblare i poligoni ma anche a rimuovere ciò che non è nel *frustum* cioè nel campo visivo della videocamera e il Blending o Merging, dove vengono eseguiti alcuni test finali e viene computato finalmente il colore di ogni pixel dello schermo), e due completamente programmabili (Vertex shader, che prende gli attributi di un vertice e ne calcola la posizione finale e alcune *varyings* che verranno utilizzate in seguito e Fragment/Pixel shader, dove si calcola il colore basandosi anche sulle *varyings* calcolate in precedenza).[12]

2.2.4 Lo skybox

Il *cubic environment map* è il metodo più usato per creare lo sfondo di un ambiente grafico tridimensionale grazie alla sua efficacia ed efficienza. Infatti consiste semplicemente di un cubo molto grande a cui vengono assegnate sei texture (ovviamente una per ogni faccia interna del cubo) e al cui interno si trovano tutti gli oggetti che compongono la scena e la camera (il punto di

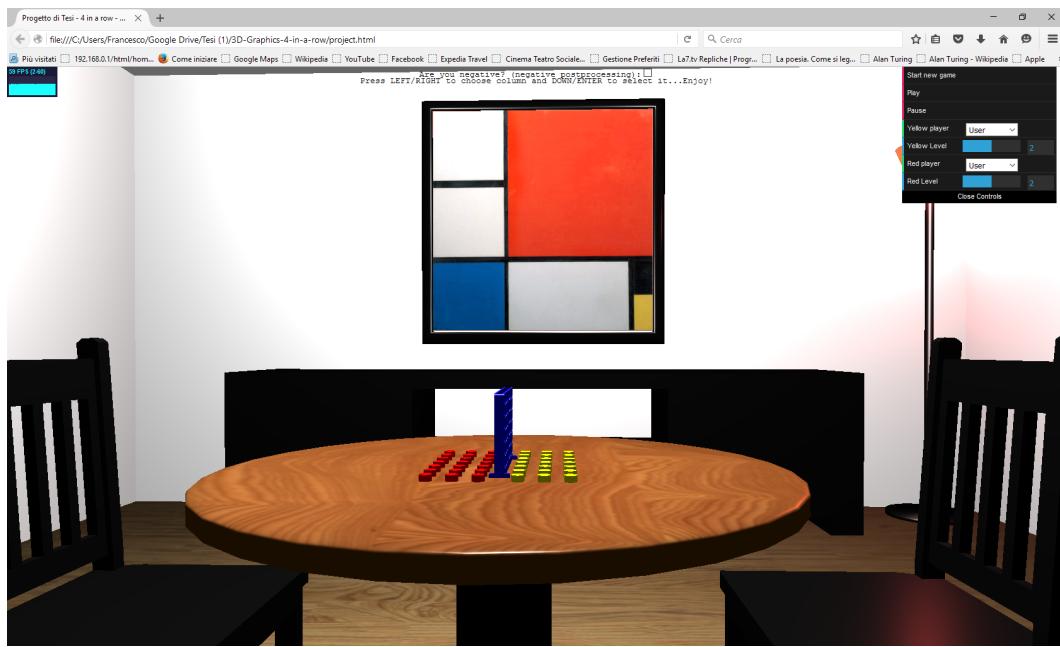


Figura 2.1: La scena

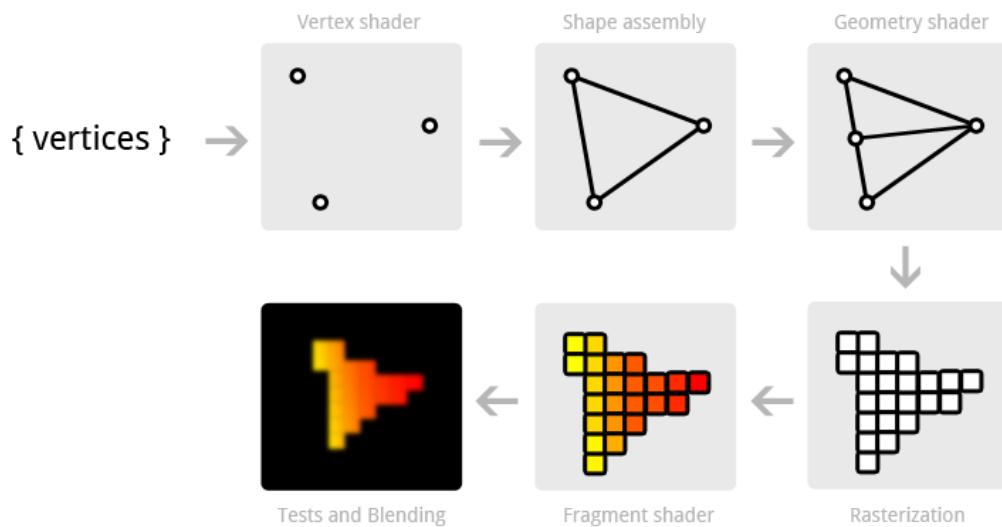


Figura 2.2: La pipeline grafica

vista) (Figure 2.3 e 2.4).

Queste sono le righe di codice necessarie a creare uno skybox con three.js [8].

```
// carico le texture
var path = "textures/Meadow/";
var format = '.jpg';
var urls =
    [path + 'posx' + format, path + 'negx' + format,
     path + 'posy' + format, path + 'negy' + format,
     path + 'posz' + format, path + 'negz' + format];
//creo il cubo e gli assegno le texture...
//...e gli altri parametri
var environmentCube =
    THREE.ImageUtils.loadTextureCube( urls );
var shader = THREE.ShaderLib[ "cube" ];
shader.uniforms[ "tCube" ].value = environmentCube;
var materialSkyBox = new THREE.ShaderMaterial( {
    fragmentShader: shader.fragmentShader,
    vertexShader: shader.vertexShader,
    uniforms: shader.uniforms,
    depthWrite: false,
    side: THREE.BackSide
} );
```

2.2.5 Lo shading e il postprocessing

Ora la nostra ambientazione ha uno skybox ed è popolata di oggetti, però questi oggetti hanno un colore totalmente nero perché non reagiscono con le fonti luminose presenti nello spazio, infatti alle loro superfici manca il *materiale*, cioè quell'insieme di proprietà e valori grazie ai quali gli shaders saranno poi in grado di computare i colori, i riflessi e le trasparenze di tali superfici. Ancor prima, però, di definire i vari materiali che renderanno questo scenario piacevole e utilizzabile, bisogna scrivere uno o più shaders.

three.js ne fornisce alcuni di default: il Basic che non interagisce con la luce e prende come parametro un colore, in sostanza non cambia molto dalla situazione sopra descritta, il Lambert che interagisce con la luce in modo da assomigliare ad una superficie molto ruvida come ad esempio un muro e il Phong, che interagisce anch'esso con la luce ma in maniera opposta al Lambert, infatti simula un materiale lucido e riflettente come la ceramica. Per questo progetto non ho utilizzato gli shader di base, bensì ne ho definito uno nuovo.

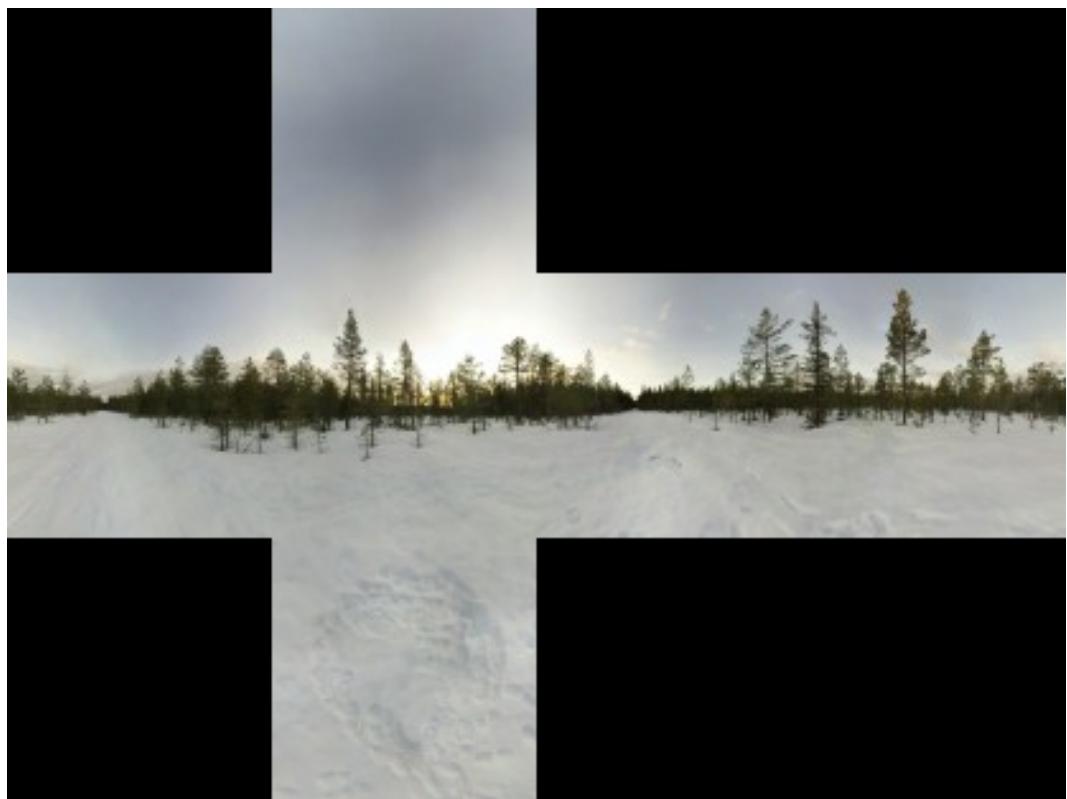


Figura 2.3: Assemblamento in piano delle sei texture per uno skybox

In realtà gli shader da me scritti sono due un vertex shader ed un fragment shader ma si comportano, visti dall'esterno, come un'unica entità e sono utilizzati dal resto del programma “a scatola chiusa”.

Lo shading è l'insieme di operazioni che, durante il rendering di un ambiente virtuale permettono alla scheda grafica di calcolare i colori degli oggetti di tale ambiente, quindi in teoria vorremmo che gli shader a nostra disposizione ci permettessero di riprodurre fedelissimamente tutte le interazioni della luce con le superfici da noi create. L'*equazione di rendering* fornisce un modo per calcolare la quantità di luce uscente da un punto in una certa direzione:

$$L_o(X, v) = \int_{\|\mathbb{S}^2\|} L_i(Y, w) |w \cdot n| f(w, v) dw$$

dove L_o è il colore e l'intensità della luce uscente da un punto X in direzione v è l'integrale sulla sfera cioè la somma infinitesimale di tutti i colori (e di conseguenza delle potenze) dei raggi “entranti” in quel punto e provenienti dal punto Y con direzione w moltiplicati per il modulo del prodotto scalare dell'angolo di incidenza di tale raggio e della normale della superficie in quel punto, perché la potenza dei raggi incidenti in un punto viene smorzata, per ragioni di fisica ottica, a seconda del loro angolo di incidenza e per il risultato di una funzione f che modella la riflessione della luce entrante dalla direzione w e uscente dalla direzione v . Sfortunatamente ci scontriamo ancora una volta con il fatto che la realtà è complessa e anche questa funzione (seppur già sia un'approssimazione) non ha una soluzione analitica, l'integrale è un integrale su dominio continuo (quindi infinito) ed è anche ricorsiva. Insomma, è molto più complessa di quanto una GPU possa elaborare poiché un qualsiasi raggio di luce prima di raggiungere l'occhio umano o l'obiettivo di una videocamera subisce un numero altissimo di riflessioni e rifrazioni interagendo con gli oggetti che ci circondano, ma la faccenda è ancora più complessa perché vogliamo calcolare le interazioni con il nostro ambiente di tutti i raggi di luce che raggiungono il nostro occhio/obiettivo, e tutto ciò andrebbe eseguito 30/60 volte al secondo...non basterebbe il computer più performante in commercio!

Per semplificare la situazione calcoliamo solo l'effetto della luce bisogna ridurre il numero di fonti luminose ad una manciata ottenendo così

$$L_o(X, v) = A + \sum_{Y \in lights} |l \cdot n| f(l, v) \beta(Y, X)$$

, dove A rappresenta la luce ambientale diffusa che migliora l'aspetto grafico della scena sopprimendo al mancato calcolo delle fonti luminose indirette e l la direzione per la fonte luminosa Y . Ora l'effetto che vogliamo dare al materiale dipende totalmente da f , la BRDF (*bidirectional reflectance distribution function*) che appunto calcola, come prima, la quantità di luce riflessa dal punto di

una superficie in una direzione a seconda da dove arriva la radiazione luminosa incidente.

La funzione di riflettanza (la quantità di energia luminosa viene emessa in una direzione dal punto di una superficie) più semplice e, al contempo, efficacissima nel far apparire in maniera realistica le superfici molto ruvide o con proprietà di subsurface scattering (la luce penetra i primi strati del materiale e subisce riflessioni e rifrazioni per cui una parte torna ad uscire distribuita equamente in tutte le direzioni), che diffondono equamente in ogni direzione l'energia luminosa che ricevono. La sua formula è questa:

$$f(l, v) = \frac{\rho}{\pi}$$

dove ρ è l'*albedo*, cioè l'energia emessa da un punto quando è raggiunto da una radiazione luminosa, la quale viene divisa per π (la superficie di mezza sfera) per ricavare l'intensità della radiazione emessa in una singola direzione.

I materiali non lambertiani sono invece formati da superfici lucide più o meno lisce, che quindi riflettono la luce e, se si vuole dare un effetto particolarmente lucido e si ha a disposizione una macchina sufficientemente performante, anche gli elementi dell'ambiente che lo circondano. Anche in questo caso, in realtà la faccenda è più complicata: poiché le superfici sono comunque ruvide e si vengono a creare a livello microscopico fenomeni di *shadowing* (certe zone sono lasciate in ombra perché coperte da una lieve irregolarità), di masking (certe zone non vengono viste dall'osservatore perché si trovano dietro ad un'a discontinuità della superficie) e di inter-reflections (un raggio raggiunge una piccola "depressione" del piano viene riflesso/assorbito/rifratto più volte prima di poter raggiungere la camera. A causa di queste e di altre complicazioni la BRDF di un materiale microfacet è questa:

$$f(l, v) = \frac{F(l, h)G(l, v, h)D(h)}{4(n \cdot l)(n \cdot v)}$$

, dove h è la normale della superficie in quel punto, l e v sono le stesse dell'equazione semplificata di rendering, F è la riflettanza di Fresnel (calcolata tramite la funzione approssimata di Schlick): una caratteristica di alcuni materiali che quando vengono guardati quasi a 90° aumentano esponenzialmente la loro riflettanza, G è il Geometry factor, che è una funzione che ci dà la probabilità che un punto sia visibile ed illuminato e D è una funzione che determina quante normali della superficie puntano verso il nostro obiettivo. Per G e D c'è una vasta possibilità di scelta [6].

Ovviamente pochi materiali hanno solo la componente diffusa o quella speculare, nella maggioranza dei casi sono entrambe presenti, in tal caso basta sommare le due componenti e normalizzando il risultato (facendo in modo che

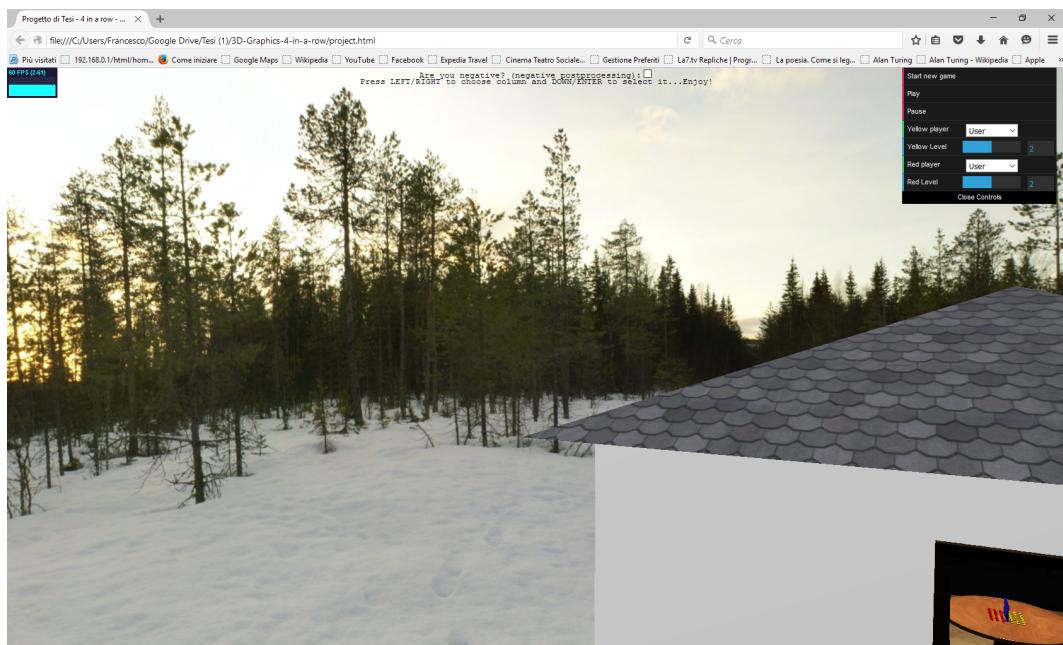


Figura 2.4: Uno skybox

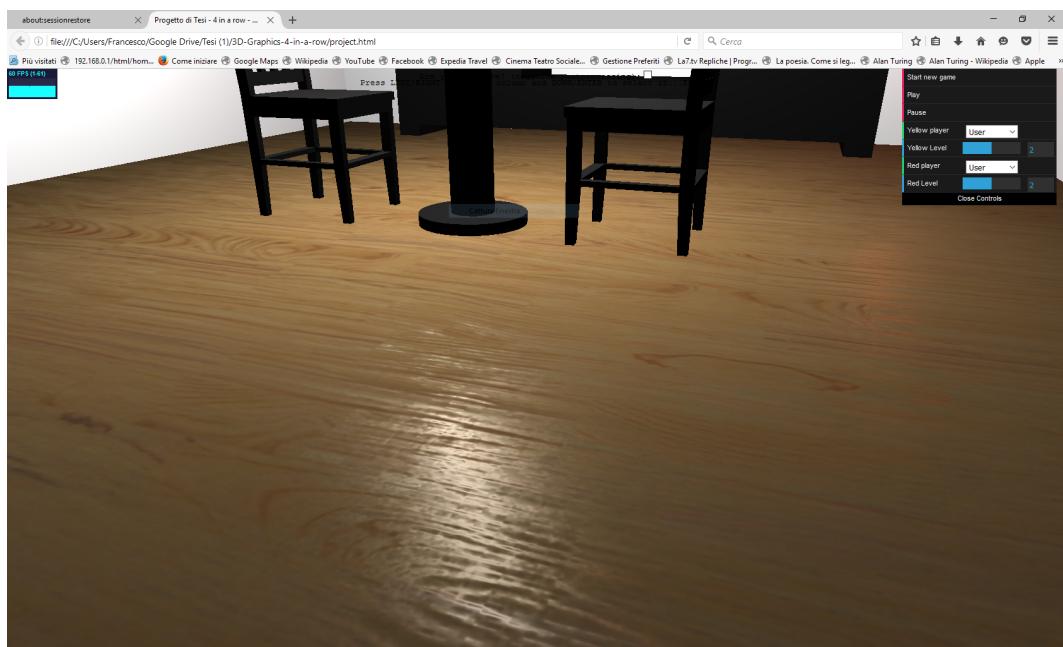


Figura 2.5: L'effetto dell'applicazione di textures sulla superficie del pavimento

il massimo possa essere 1).

Infine, si pone ancora un problema: come simulare superfici non uniformi come il quadro, il tetto o gli oggetti in legno? Ed è qui che entrano in gioco le *textures* (*o texture maps*): array mono-, bi- o tridimensionali di colori che vengono applicati alle superfici tramite l'operazione di *texturing* o *texture mapping* che associa a ciascun punto di esse un *texel* (letteralmente tassello), cioè un pixel della texture. Le texture maps più frequentemente utilizzate sono quelle 2D, semplici immagini che però possono essere utilizzate per gli scopi più vari: per cambiare il colore diffuso di una superficie (diffuse color mapping, o più semplicemente color mapping) o quello speculare, cioè il colore che viene emesso da un materiale lucido al massimo della sua riflettanza, (specular mapping), ma anche la sua trasparenza (transparency mapping), la sua ruvidità (alpha mapping) e addirittura si possono modificare le normali di una superficie (normal mapping) in modo da simulare delle irregolarità senza complicare la geometria di un oggetto, gli effetti della luce su un'area, leggi anche ombre ma non solo, (shadow, parallax, ambient occlusion etc. mapping) e addirittura per creare a tempo d'esecuzione oggetti più complessi di quelli modellati offline, procedura però abbastanza pesante per la GPU (displacement mapping). Insomma, praticamente tutto ciò che si riesce a rappresentare con (triple di) numeri, può essere usato per creare una texture e poi applicato alla superficie di un oggetto[12, 8, 5].

```
<script type="text/x-glsl" id="vertex">
    varying vec3 tNormal;
    varying vec3 lVector [3];
    varying vec3 pointPosition;
    varying vec4 worldPosition;
    varying vec2 vUv;

    uniform vec3 pointLightPosition [3];

    void main() {
        tNormal = normalMatrix * normal;
        worldPosition = modelMatrix * vec4( position , 1.0 );
        pointPosition = ( viewMatrix * worldPosition ).xyz;
        gl_Position = projectionMatrix *
            vec4( pointPosition , 1.0 );
        vUv = uv;
        vec4 lPosition [3];
        for( int i = 0; i < 3; i++ ) {
```

```
    lPosition[ i ] = viewMatrix *  
        vec4( pointLightPosition[ i ], 1.0 );  
    lVector[ i ] = lPosition[ i ].xyz - pointPosition;  
}  
}  
</script>  
<script type="text/x-glsl" id="fragment">  
    varying vec3 tNormal;  
    varying vec3 lVector[3];  
    varying vec3 pointPosition;  
    varying vec4 worldPosition;  
    varying vec2 vUv;  
  
    uniform vec3 pointLightPosition[3];  
    uniform vec3 lightPower[3];  
    // surface specular color: equal to F(0)  
    uniform vec3 c_spec;  
    // surface diffuse color  
    uniform vec3 rho;  
    // material roughness (increase for rougher surface)  
    uniform float alpha;  
    // ratio of diffuse lighting  
    uniform float s;  
    uniform vec3 ambientLight;  
  
#define PI 3.14159265  
  
    // compute the geometry term  
    float G(float LdotH) {  
        return 1.0/pow(LdotH, 2.0);  
    }  
  
    // compute Fresnel reflection term with  
    // Schlick approximation  
    vec3 F(float LdotH) {  
        return c_spec +  
            (vec3(1.0) - c_spec)*pow(1.0-LdotH, 5.0);  
    }  
  
    // compute the normal distribution function ,
```

```

//based on Trowbridge-Reitz
float D( float NdotH ) {
    float A = pow( alpha , 2.0 );
    float B = PI * pow(
        pow( NdotH , 2.0 )*( A-1.0 ) + 1.0 , 2.0 );
    return A/B;
}

void main() {
    vec3 shadedColor = vec3( 0.0 , 0.0 , 0.0 );
    vec3 n = normalize( tNormal );
    vec3 v = normalize( -pointPosition );

    if( abs( worldPosition.x ) >= 1390.0 ||
        abs( worldPosition.z ) >= 1190.0 ||
        worldPosition.y >= 870.0 ){
        vec3 beta = lightPower[2];
        vec3 l = normalize( lVector[2] );
        vec3 h = normalize( v+l );
        float NdotL = max(0.000001, dot( n , l ));

        //SPECULAR TERM
        float NdotH = max(0.000001, dot( n , h ));
        float VdotH = max(0.000001, dot( v , h ));
        float NdotV = max(0.000001, dot( n , v ));
        //specular BRDF
        vec3 spec_term =
            F(VdotH) * G(VdotH) * D(NdotH) / 4.0;

        vec3 diff_term = rho;

        shadedColor += vec3( beta * NdotL *
            ( s * diff_term + (1.0-s) * spec_term ) );
    }else{
        for( int i = 0; i < 2; i++ ){
            //GENERAL OPERATIONS
            vec3 beta = lightPower[ i ] /
                ( 4.0 * PI * pow( length( lVector[ i ] ) , 2.0 ) );
            vec3 l = normalize( lVector[ i ] );
            vec3 h = normalize( v+l );

```

```

float NdotL = max(0.000001, dot( n, l ));

//SPECULAR TERM
float NdotH = max(0.000001, dot( n, h ));
float VdotH = max(0.000001, dot( v, h ));
float NdotV = max(0.000001, dot( n, v ));
//specular BRDF
vec3 spec_term =
    F(VdotH) * G(VdotH) * D(NdotH) / 4.0;
vec3 diff_term = rho;
shadedColor += vec3( beta * NdotL *
    ( s * diff_term + (1.0-s) * spec_term ) );
}
gl_FragColor =
    vec4( shadedColor + ambientLight , 1.0 );
}
</script>

```

Per implementare il post-processing si deve aggiungere un passo ulteriore in fondo alla pipeline grafica in modo da poter modificare a piacere i colori di ogni pixel sullo schermo. In questo caso ho implementato un'effetto negativo (Figura 2.) che non richiede computazioni particolari come magari l'effetto blur.[12, 8]

```

negShader = {
uniforms: {
    "tDiffuse": { type: "t", value: null },
    "width": { type: "f", value: 0.0 },
    "height": { type: "f", value: 0.0 },
},
vertexShader: [
    "varying vec2 vUv;" ,
    "void main() {" ,
    "    vUv = uv;" ,
    "    gl_Position = projectionMatrix *
        modelViewMatrix * vec4( position , 1.0 );" ,
    "}"
].join("\n") ,

fragmentShader: [
    "uniform sampler2D tDiffuse;" ,

```

```

"varying vec2 vUv;" ,
"void main() {",
"    vec3 original_color =
        texture2D( tDiffuse , vUv ).rgb ;",
"    vec3 new_color = vec3( 1.0 - original_color );",
"    gl_FragColor = vec4( new_color , 1.0);",
"}"
].join("\n")
};

...
var negEffect , composer , renderer ;
function updateOptions() {
var neg = document.getElementById('negative');
negEffect.enabled = neg.checked;
}
...
// nella funzione di inizializzazione
composer = new THREE.EffectComposer( renderer );
composer.addPass( new THREE.RenderPass( scene , camera ) );

negEffect = new THREE.ShaderPass( negShader );
negEffect.uniforms.width.value = window.innerWidth;
negEffect.uniforms.height.value = window.innerHeight;
composer.addPass( negEffect );

var effect = new THREE.ShaderPass( THREE.CopyShader );
effect.renderToScreen = true;
composer.addPass( effect );

updateOptions();

...
// nel ciclo di rendering
composer.render();

```

2.2.6 Il sistema particellare

I “coriandoli” del sistema particellare sono dei piccoli quadrati tutti uguali di colore rosso o giallo che vengono generati casualmente all’interno di un’area circoscritta (quella della stanza) e si muovono dall’alto in basso con la stes-

sa velocità, quelli che finiscono sotto il pavimento, poi vengono riportati alla posizione iniziale dando vita ad un ciclo che si interromperà quando l'utente chiuderà la pagina web soddisfatto per il successo, o amareggiato per la sconfitta, oppure quando si resetterà la situazione di gioco per la rivincita premendo “Start new game” [12].

```
var particles = new THREE.Geometry();
//Creo il sistema particellare per la sorpresa finale
function makeParticles( winner ){
    for ( var p = 0; p < pNum; p++ ) {
        var particle = new THREE.Vector3(
            Math.random() * 2800 - 1400,
            Math.random() * 1900 - 950,
            Math.random() * 2400 - 1200);
        particles.vertices.push(particle);
    }
    var particleMaterial;
    if ( winner == -1 ) {
        particleMaterial =
            new THREE.PointsMaterial(
                { color: 0xffff00, size: 10 })
    );
    } else {
        particleMaterial =
            new THREE.PointsMaterial(
                { color: 0xff0000, size: 10 })
    );
    }
    pointsSystem =
        new THREE.Points( particles, particleMaterial );
    scene.add(pointsSystem);
}
//...e lo faccio muovere
function updateParticles(delta){
    for ( var i = 0; i < particles.vertices.length; i++ ) {
        var p = particles.vertices[ i ];
        p.setY( p.getComponent( 1 ) -( t * 100 ) );
        if ( p.getComponent( 1 ) <= -950 ) {
            p.setY( 950 );
        }
    }
}
```

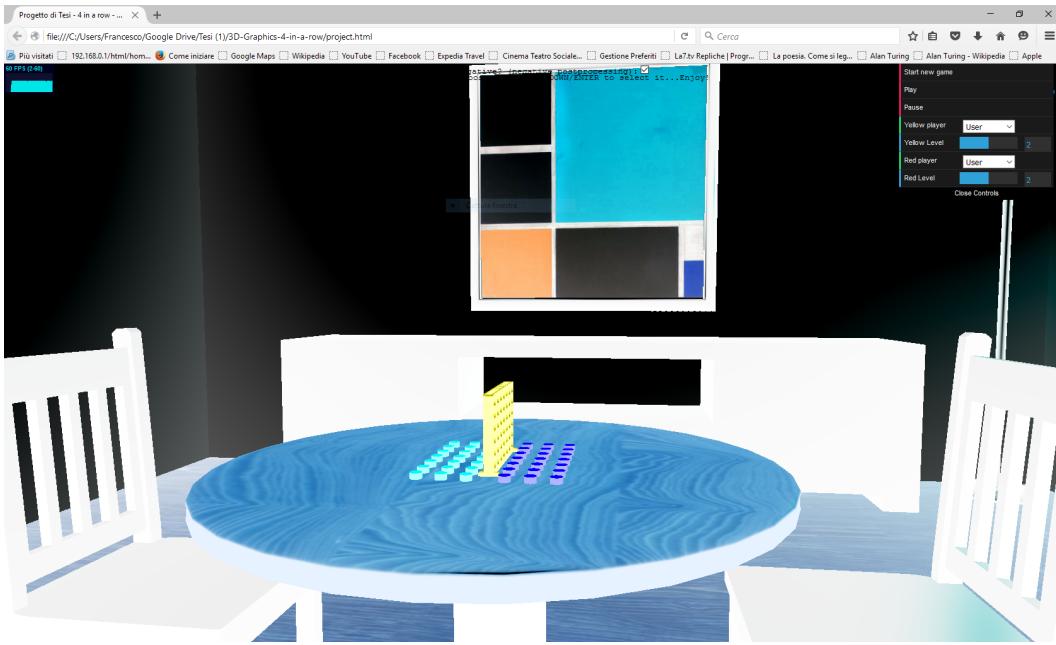


Figura 2.6: Come si presenta la scena se l'effetto negativo di post-processing è attivato

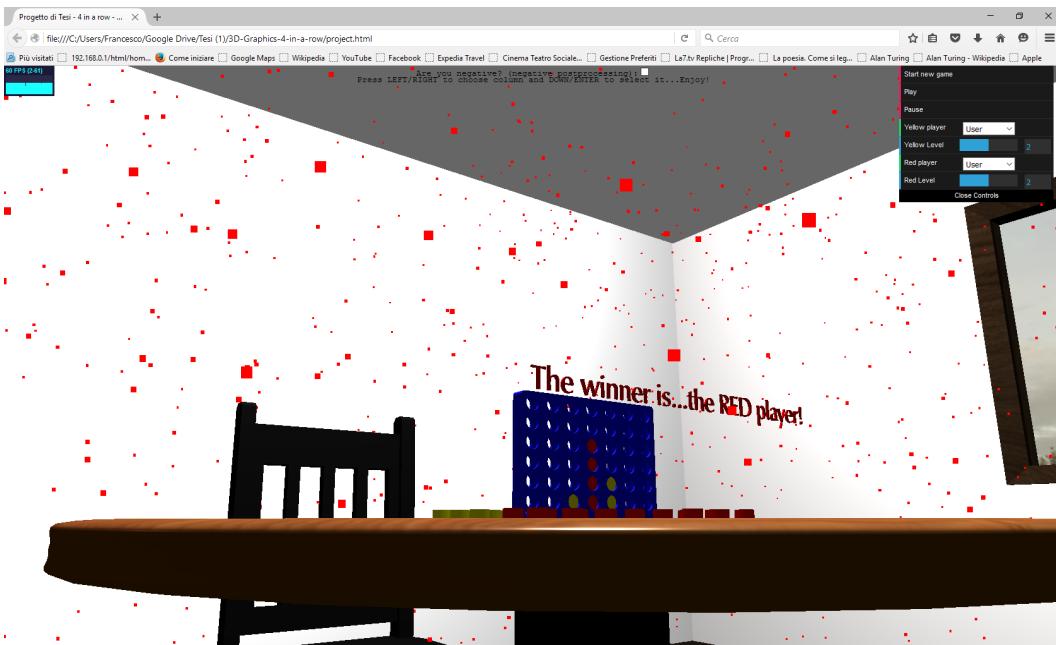


Figura 2.7: Il mio sistema particellare

```

    particles.verticesNeedUpdate = true;
}
...
// nel ciclo di rendering
t = clock.getDelta();
updateParticles(t);

```

2.2.7 I quaternioni

I quaternioni sono uno strumento matematico inventato da sir William Rowan Hamilton nel 1843 e applicato alla computer grafica nel 1985 per il fatto che sono davvero efficaci nel rappresentare le rotazioni, richiedono infatti un terzo della memoria per l'allocazione rispetto alle matrici di rotazione (un altro metodo per rappresentare e implementare le rotazioni), le operazioni fra quaternioni sono molto più efficienti di quelle fra matrici e, cosa più importante, esiste un metodo per interpolare due quaternioni riuscendo, così a realizzare l'animazione di una rotazione senza imbattersi in errori di approssimazione che potrebbero risultare fastidiosi all'utente.

Un quaternione (Figura 2.) \hat{q} è una quadrupla che rappresenta una rotazione di angolo θ rispetto ad un asse arbitrario a con le sue componenti a_x, a_y, a_z , dove

$$\hat{q} = \langle a_x \sin \frac{\theta}{2}, a_y \sin \frac{\theta}{2}, a_z \sin \frac{\theta}{2}, \cos \frac{\theta}{2} \rangle$$

. I primi tre termini indicano l'asse attorno a cui l'oggetto è ruotato, mentre il quarto contiene l'informazione riguardante l'angolo di rotazione.

Per animare una rotazione basta applicare una *spherical linear interpolation* (**Slerp**) a due quaternioni, uno iniziale e uno finale, dove

$$Slerp(\hat{q}_i, \hat{q}_f, t) = \frac{\sin(\theta(1-t))}{\sin \theta} \hat{q}_i + \frac{\sin \theta t}{\sin \theta} \hat{q}_f$$

,

$$\theta = \arccos(\hat{q}_i \cdot \hat{q}_f)$$

e t è il fattore di interpolazione, che nel nostro caso è il tempo [12, 8].

```

// Istanziò e setto il quaternione
var id = 0;
var qi = new THREE.Quaternion();
var qf = new THREE.Quaternion();
qf.setFromAxisAngle(
    new THREE.Vector3(1, 0, 0).normalize(), Math.PI / 2);

```

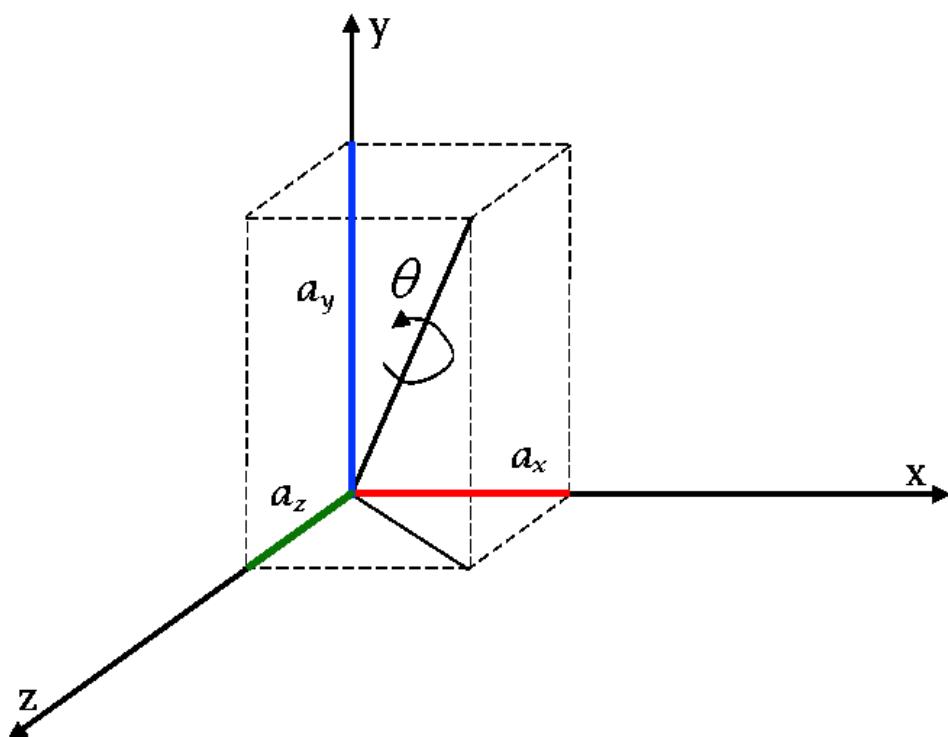


Figura 2.8: Rappresentazione grafica di un quaternione

```

var performRotation = false;
...
//lo assegno a uno o pi\‘u oggetti
tokens.children[ 20 - t ].quaternion.copy( qi );
tokens.children[ 41 - t ].quaternion.copy( qi );
//eseguo l'animazione nel ciclo di rendering
if ( performRotation && step <= 1 ) {
    tokens.children[ id ].quaternion.copy(
        qi.slerp( qf, step )
    );
    step += 0.01
} else {
    performRotation = false;
    step = 0;
}

```

2.2.8 Le ombre

Come già accennato al paragrafo 2.2.5 le ombre sono il risultato dall'applicazione di texture che possono essere anche calcolate offline (a priori) se tutti gli oggetti e le fonti luminose fossero statiche nella scena, nel mio caso, però ho degli elementi che si spostano quindi andrebbero calcolate *on-the-fly* (a tempo d'esecuzione) ed è troppo computazionalmente pesante da fare, ho fatto comunque in modo che le luci interne non influiscano sull'ambiente esterno alla casa e che la luce ambientale non illumini le facce interne dei muri o il pavimento o i mobili cosa che, non avendo abilitato il *casting*, cioè la proiezione delle ombre, non succedeva. Ciò è stato facilmente implementato grazie ad un controllo nel quale, se il frammento preso in considerazione ha una worldPosition tale da essere all'esterno della casa ($|x| \geq 1390$ o $|y| \geq 870$ o $|z| \geq 1190$) allora riceve solo la luce direzionale posta all'esterno della casa, altrimenti la luce direzionale non influisce sul suo colore finale sul quale, però, influiscono le due luci poste internamente a essa.

```

if( abs( worldPosition.x ) >= 1390.0 ||
    abs( worldPosition.z ) >= 1190.0 ||
    worldPosition.y >= 870.0 ){
    ...
} else { ... }

```


Capitolo 3

Rappresentazione della conoscenza e Ragionamento Automatico

In seguito ad aver creata l’interfaccia grafica tridimensionale ho implementato un’intelligenza artificiale che rispondesse in maniera sensata alle mosse di un giocatore umano. Dapprima l’ho sviluppata modellando il problema con MiniZinc e Answer Set Programming con lo scopo di sfruttare al meglio le conoscenze apprese con il corso di Automated Reasoning e per confrontare l’efficacia e l’efficienza dei rispettivi solver. Sfortunatamente, quando ormai i modelli erano pronti, mi sono accorto di non poter utilizzare su di essi l’algoritmo minimax e ciò rendeva il comportamento dell’intelligenza artificiale buono ma non ottimale.

Per questo motivo ho poi scritto un solver in JavaScript, successivamente integrato con la parte grafica, che implementasse questo algoritmo.

3.1 Strumenti usati

- **MiniZinc:** linguaggio ad alto livello pensato per modellare problemi di soddisfacibilità e di ottimizzazione con vincoli. Offre una sintassi pulita, spesso vicina alla notazione matematica, ma anche tipi di dato complessi (array multi-dimensionali, intervalli...) e la possibilità di gestire separatamente dati e modello (utile nell’esecuzione di batterie di test). La metodologia di programmazione è di tipo constraint+generate e lo stile per mettere i vincoli e’ abbastanza vicino a quello della programmazione imperativa [10].

- **Answer Set Programming:** forma di programmazione dichiarativa, che usa una variante della sintassi del linguaggio di programmazione Prolog, ma con la possibilità di utilizzare la negazione in modo libero così come costrutti di aggregazione e vincoli di integrità, orientata alla risoluzione di problemi di ricerca complessi. Nato come conseguenza delle ricerche su *non-monotonic reasoning* e *knowledge representation*, trova il suo campo di utilizzo nella ricerca di soluzioni in domini finiti [9].
- **JavaScript:** linguaggio di programmazione orientato agli oggetti introdotto nel 1995 e pensato per rendere possibile l'aggiunta di programmi (*script*) alle pagine web. Ora è supportato da tutti i web browser e conta innumerevoli estensioni.

3.2 Basi teoriche

3.2.1 La programmazione a vincoli

La programmazione a vincoli è un paradigma di programmazione **dichiarativa** molto adatta alla modellazione e alla risoluzione di problemi combinatorici.

Definizione 3.1. Un **CSP** (*Constraint Satisfaction Problem*) è una tripla $\langle \mathcal{V}, \mathcal{D}, C \rangle$, dove $\mathcal{V} = \{V_1, \dots, V_n\}$ è l'insieme delle variabili, con domini $\mathcal{D} = \{D_1, \dots, D_n\}$, e C è l'insieme dei vincoli.

Un **vincolo** (*constraint*) $c \in C$ sulle variabili V_{i_1}, \dots, V_{i_m} è una **relazione** su $D_{i_1} \times \dots \times D_{i_m}$, cioè $c \subseteq D_{i_1} \times \dots \times D_{i_m}$. Si dirà che una m -upla $V_{i_1}, \dots, V_{i_m} \in D_{i_1} \times \dots \times D_{i_m}$ soddisfa il vincolo c se $V_{i_1}, \dots, V_{i_m} \in c$.

La **soluzione** di un CSP è un assegnamento $\sigma : \mathcal{V} \rightarrow D_1 \cup \dots \cup D_n$ tale che:

- $(\forall i = 1, \dots, n)(\sigma(V_i) \in \mathcal{D}_i)$;
- $(\forall c \in C \text{ su } V_{i_1}, \dots, V_{i_m})(\langle \sigma(V_{i_1}), \dots, \sigma(V_{i_m}) \rangle \in C)$.

Cioè ogni valore assegnato ad una variabile deve essere all'interno del dominio di tale variabile e deve soddisfare tutti i vincoli definiti su di essa.

Definizione 3.2. Un **COP** (*Constraint Optimization Problem*) è un CSP 3.1 $\langle \mathcal{V}, \mathcal{D}, C \rangle$ con una funzione $f : V_{j_1}, \dots, V_{j_k} \in \mathcal{V} \rightarrow \mathbb{R}$. La soluzione di un COP è un assegnamento che soddisfi il CSP e che *minimizzi/massimizzi* f .

Modellare un problema a vincoli significa definire un CSP 3.1 (o un COP 3.2) da far poi risolvere ad un solver al cui si possono applicare euristiche, cioè metodi di ricerca delle soluzioni, che rendano più veloce tale ricerca.[1]

3.2.2 La programmazione logica

La programmazione logica nasce negli anni '70 come punto d'incontro dei lavori sul ragionamento deduttivo, sulla logica formale e sui linguaggi di programmazione dichiarativa per il problem solving.

Ciò rese possibile la formalizzazione di problemi basati su relazioni logiche di causa-effetto, scrivendo modelli risolvibili da un calcolatore. Oggigiorno la programmazione logica viene applicata soprattutto alla rappresentazione della conoscenza (*Knowledge Representation*) e al ragionamento non-monotono (*non-monotonic reasoning*), il quale fornisce meccanismi rigorosi che permettono a sistemi intelligenti di poter reagire adeguatamente quando si affrontano problemi dove l'informazione è incompleta o potrebbe cambiare riducendo l'insieme delle possibili soluzioni. [2]

Definizione 3.3. Un programma logico è formato da una serie di *clauses o rules* (clausole/regole) cioè di regole scritte in questa forma:

$H : -B_1, \dots, B_m, not B_{m+1}, \dots, B_n$, dove H è la *testa* della regola e il resto è il *corpo*. Se $m = n$ si ha una *definite rule/clause*, se $m = n = 0$ si ha, invece un *fact* (fatto).

H e B_1, \dots, B_n sono *atomi* (o formule atomiche).

Definizione 3.4. Sia \mathcal{P} l'insieme dei simboli dei predicati, ogni $p \in \mathcal{P}$ ha una sua arietà (numero di argomenti a cui può essere applicato), se p ha arietà n e t_1, \dots, t_n sono *termini* (terms) allora $p(t_1, \dots, t_n)$ è un atomo.

Definizione 3.5. Sono termini tutti i $c \in \mathcal{C}$, tutte le $X \in \mathcal{V}$ e tutte le $f(t_1, \dots, t_n)$ tali che $f \in \mathcal{F}$, f ha arietà n e t_1, \dots, t_n sono termini; dove \mathcal{C} è l'insieme delle costanti, \mathcal{V} quello delle variabili e \mathcal{F} quello delle funzioni.

Un *term* è un **ground term** se non presenta variabili.

A un tale programma P vogliamo assegnare un significato (o interpretazione) su un universo di oggetti, dove un atomo non è altro che una relazione fra oggetti di questo universo. Tra le varie interpretazioni possibili alcune possono essere dei *modelli* per il programma P , cioè delle interpretazioni che soddisfino il significato logico di tutte le clausole di P .

Definizione 3.6. Indicando $\bar{t}_1, \dots, \bar{t}_n$ l'interpretazione dei termini t_1, \dots, t_n su un certo universo di valori e Q quella di un predicato q , se in tutti i modelli di P si ha che $(\bar{t}_1, \dots, \bar{t}_n) \in Q$, allora $q(t_1, \dots, t_n)$ è **conseguenza logica** di P . Ciò si indica $P \models q(t_1, \dots, t_n)$.

La **soluzione** di un programma \mathcal{P} è l'insieme delle sue conseguenze logiche [1, 9]

26 Rappresentazione della conoscenza e Ragionamento Automatico

3.2.3 I solver

Per MiniZinc esistono diversi solver che però, a grandi linee, funzionano tutti allo stesso modo; infatti si basano sull'algoritmo **DP(LL)** che prende il nome da colore che l'hanno inventato nel 1960 (Davis-Putnam), e migliorato nel 1962 (Logemann-Loveland). Questo algoritmo è stato ideato per il *SAT solving*, la risoluzione di problemi di soddisfabilità di una formula logica posta in *forma normale congiuntiva (CNF)*, ma con i dovuti adattamenti, può essere usato per il Constraint Solving.

```
DP(formula , ass )
  ass '\leftarrow unit\_propagation(formula , ass )
  if (ok(ass )) return ass '
  else if (ko(formula , ass )) return false
  else X <- select_variable(formula , ass ')
    ass '' <- DP(formula , ass '[X/true ])
    if (ass ''? = false) return ass ''
    else return DP(formula , ass '[X/false ])
```

Il Davis-Putnam è consta di due parti fondamentali:

1. la **unit propagation** nella quale vengono assegnati in maniera deterministica dei valori ad alcune variabili che, nel contesto in cui si trovano, non possono che avere quel dato valore (la spiegazione delle tecniche con cui viene implementato questo passaggio esula dagli argomenti di questa tesi);
2. la **scelta non deterministica** del valore di una variabile X , che viene seguita da una chiamata ricorsiva dell'algoritmo sulla nuova formula. Questa chiamata può ritornare **false** se l'assegnamento non soddisfa la formula oppure restituisce l'assegnamento valido.

Nel Constraint Solving la fase di scelta non deterministica rimane pressoché immutata, mentre la unit propagation è sostituita dalla **constraint propagation**.

Definizione 3.7. Una **regola di prova** si rappresenta così

$$\frac{\varphi}{\psi}$$

dove φ e ψ sono due CSP e in ψ possono occorrere variabili non presenti in φ . La regola è detta *equivalence preserving* se $Sol(\varphi) = Sol(\psi) \parallel_{FV(\varphi)}$ cioè se la soluzione dei due problemi è la stessa, al netto delle nuove variabili introdotte in ψ .

Le regole di prova possono modificare i domini delle variabili, l'insieme dei vincoli e sono:

- **domain reduction rules:** nelle quali si cerca di ridurre la dimensione dei domini sfruttando l'informazione presente nei vincoli e rimuovendo i vincoli diventati inutili. Se l'insieme dei vincoli $\mathcal{C}' = \text{true}$ allora $\mathcal{D}_1 \times \dots \times \mathcal{D}_n$ rappresenta tutte le soluzioni possibili al CSP, se invece quest'ultimo è \emptyset allora il CSP $\psi = \text{fail.}$;
- **trasformation rules:** dove si trasforma, semplificando, i vincoli di φ ; questo processo può introdurre nuove variabili (con un proprio dominio) e può essere introdotto anche il **fail**;
- **introduction rules:** dove si sceglie, per comodità di inserire dei nuovi vincoli (solitamente più deboli dei precedenti).

Definizione 3.8. È detta constraint propagation l'applicazione ripetuta di un'insieme di *regole di prova* 3.7. La sequenza di applicazione di tali regole è invece detta *derivazione*, la quale, se è finita, può essere

- *failing*: porta ad un CSP **fail** (senza soluzioni);
- *stable*: non è failing e nessun'altra regola è applicata al CSP
- *successful*: è stable e il CSP finale è in **solved form**.

La tecnica di risoluzione dei modelli in Answer Set Programming è piuttosto diversa e affonda le sue radici nella logica formale.

Ora, si consideri l'insieme di tutti i termini ground ottenibili con i simboli delle funzioni e delle costanti in un programma P , questo insieme può essere utilizzato come universo per le interpretazioni e si chiama **universo di Herbrand** (H_P); questo insieme è univoco. Le interpretazioni su questo universo possono essere anche **modelli di Herbrand** di P .

Visto che esse si possono rappresentare come un insieme di atomi, possiamo dire che, sia $B_P = \{p(t_1, \dots, t_n) : p \in \mathcal{P} \wedge t_1, \dots, t_n \text{ are ground terms}\}$ la **base di Herbrand** di un programma P , ogni suo sottoinsieme determina univocamente un'interpretazione di Herbrand per P e, alcune di esse, saranno modelli di Herbrand per P .

Teorema 3.9. Sia P un programma formato da definite clauses (definite clauses program), non contenenti, cioè, atomi negati. Allora P ammette un (unico) modello di Herbrand che sia minimo M_P . Cioè tale che se I è un Herbrand model di P , allora $M_P \subseteq I$.

28 Rappresentazione della conoscenza e Ragionamento Automatico

Corollario 3.10. *Sia P un definite clause program e $A \in B_P$ un ground atom. Allora $P \models A$ (A è conseguenza logica di P) se e solo se $A \in M_P$.*

Perciò per calcolare l'insieme delle conseguenze logiche di un programma definito P “basta” trovare un minimal Herbrand model e ciò è resto possibile da Answer Set Programming tramite l'applicazione dell'operatore $\{T_P = (a : a \leftarrow b_1, \dots, b_n \in \text{ground}(P), \{b_1, \dots, b_n\} \subseteq I\}$ (conseguenza immediata) fino al raggiungimento di un punto fisso.

Questo metodo, però, non tratta i programmi che presentano negazioni (programmi generali), perciò dobbiamo ampliare il nostro concetto di soluzione per un modello in ASP ai **modelli stabili** (stable models) introdotti da Gelfond e Lifschitz nel 1988:

Definizione 3.11. Dato un general program P ed un modello *candidato* S , definiamo P^S , cioè il **ridotto di P rispetto a S** come segue:

1. rimuovo ogni regola che contenga nel corpo (almeno) un letterale che sia un atomo negato, un naf-literal, $\text{not } L$ tale che $L \in S$;
2. rimuovo ogni altro naf-literal dai corpi delle rimanenti regole.

Possiamo osservare ora che P^S è un definite program, che quindi ha un modello minimo M_{PS} e questo è computabile.

Se $M_{PS} = S$ allora S è un **modello stabile**, anche detto **answer set**.

In conclusione, applicando l'algoritmo di Gelfond-Lifschitz per ogni modello candidato e verificando che il ridotto sia uno modello stabile, si può trovare soluzione ai modelli di ASP [1].

3.2.4 L'algoritmo minimax

Possiamo dire che questo è il cuore, anzi il cervello, di questa tesi; infatti questo algoritmo è alla base di tutte le implementazioni di sistemi intelligenti in grado di giocare (e vincere) giochi come il “Forza 4”, cioè giochi a due persone ad informazione perfetta (dove entrambi i giocatori conoscono tutto sulla situazione di gioco corrente).

Per poter prendere una decisione riguardo ad un problema l'algoritmo richiede che gli siano fornite sia la configurazione iniziale rispetto alla quale prendere la decisione, sia una funzione con la quale valutare la bontà di una situazione *utility* o *payoff function*. Se si vuole implementare una ricerca perfetta, però, bisogna avere la possibilità di eseguire una ricerca esaustiva e visitare, quindi, tutto l'albero. La cosa non è fattibile anche per giochi tutto sommato semplici come appunto il “Forza 4!” o addirittura la tria (Tic-Tac-Toe).

Abbandoniamo quindi, l’idea che si possa compiere una scelta perfetta, ma non l’algoritmo minimax. Infatti, se si ha una funzione di payoff valida, si possono ottenere dei risultati validi anche con un numero di livelli esplorati contenuto. Il **minimax** funziona così: avendo un albero di ricerca, lo si visita con una DFS (*Depth-First Visit*) e si memorizza nei nodi ad altezza dispari il minimo fra i valori dei figli, mentre a quelli ad altezza pari si assegna il massimo, quindi la mossa prescelta sarà quella il cui valore è il massimo fra i minimi dei massimi ecc.. Ciò permette di simulare un comportamento del tipo: io giocatore opero la scelta migliore tenendo conto che anche il mio avversario farà lo stesso. Ci sono molti modi di rendere più efficiente questo algoritmo facendo in modo che non debba visitare tutto l’albero di ricerca. Il più famoso ed usato è sicuramente l’**alpha-beta pruning** (potatura alfa-beta) nel quale vengono scartati tutti i rami che non verranno usati. L’algoritmo è analogo a quello del minimax, però se sto esplorando i figli di un nodo MAX e il valore di uno dei suoi fratelli è minore del valore che esso ha in un certo momento (dopo aver visitato solo alcuni dei suoi figli) allora non serve che io finisca di visitare i sottoalberi in lui radicati perché, essendo un nodo MAX, potrà prendere solo valori ancora più alti, che verranno scartati dal nodo MIN genitore che prediligerà il valore di uno dei suoi fratelli (il minimo); se, invece sono in un nodo MIN la dinamica è opposta ma il concetto non cambia.

Questo algoritmo può essere ulteriormente migliorato tramite la *potatura euristica* che elimina parti più considerevoli dell’albero di ricerca rispetto all’alpha-beta pruning, permettendo così di andare più in profondità nell’albero di ricerca; vengono tralasciate soluzioni che potrebbero essere interessanti, ma con un vantaggio considerevole nella predizione nel gioco, se si sviluppasse davvero come previsto dall’euristica. Un altro raffinamento molto interessante è il cosiddetto *approfondimento progressivo*, il quale applica il pruning ad un numero ridottissimo di livelli, scartando subito rami che non possono dare risultati interessanti e poi lo riapplica ad un numero crescente di livelli, finché non viene raggiunto un timeout dopo il quale l’algoritmo si ferma e restituisce il risultato trovato.

La più grossa limitazione del minimax è il **problema dell’orizzonte**: in certi casi una situazione può sembrare ottima e quindi potrebbe farci propendere per la scelta del ramo che conduce a quella ma al passaggio successivo capiamo che quella situazione ottima in realtà può essere seguita da una situazione svantaggiosa a causa della mossa dell’avversario non esplorata precedentemente. Questo problema, sfortunatamente non può essere aggirato perciò l’unico modo di attenuarlo è eseguire la ricerca sul maggior numero di livelli possibili, ma la complessità computazionale del minimax per il gioco “Forza 4!” è $O(7^n)$ dove n è il numero di livelli che si vogliono esplorare. Quindi si rischia di

30 Rappresentazione della conoscenza e Ragionamento Automatico

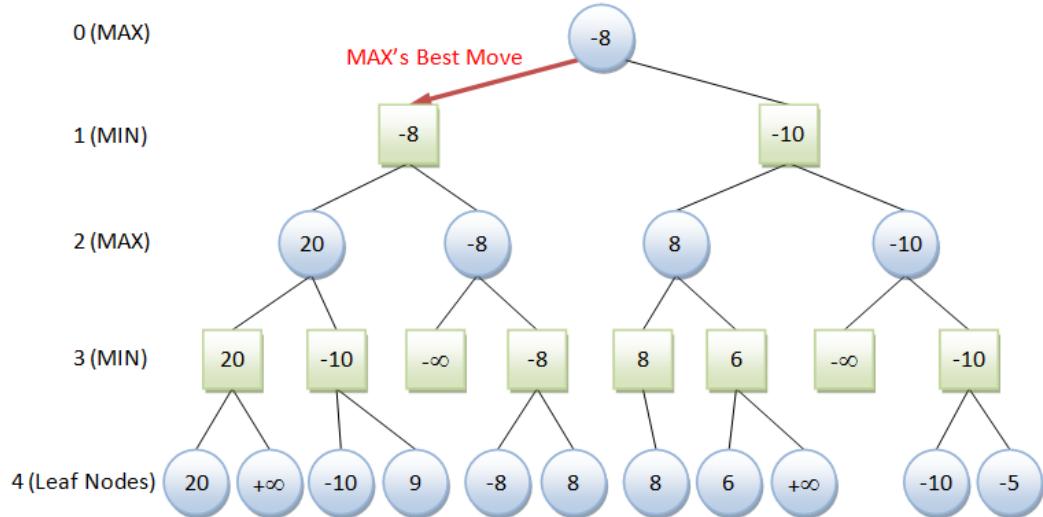


Figura 3.1: L’algoritmo minimax

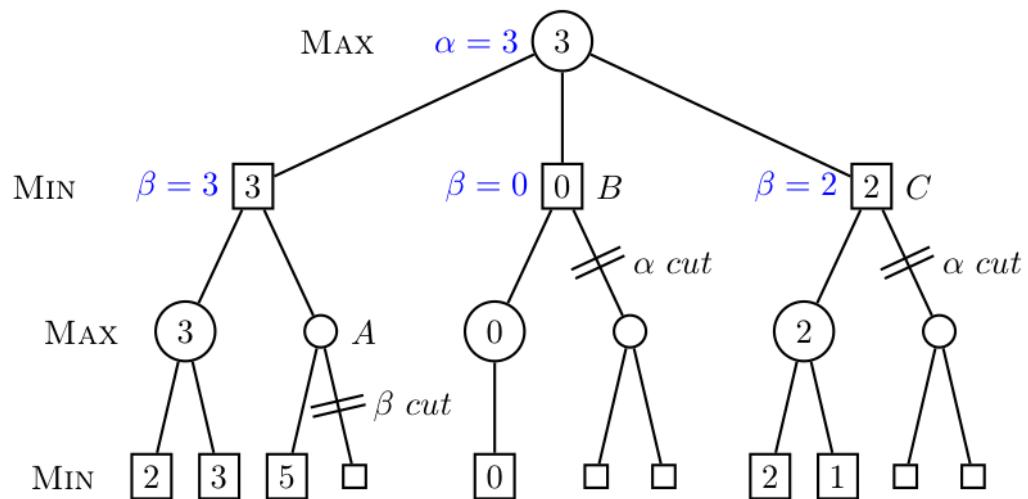


Figura 3.2: L'algoritmo alpha-beta pruning

avere, già con un albero di ricerca relativamente basso, un costo in termini sia di tempo di esecuzione che di memoria utilizzata, troppo alto [11, 3].

La mia euristica si basa sulla valutazione, per mezzo di una semplice funzione esponenziale, del numero di pedine di un giocatore disposte in linea e non interrotte da pedine avversarie. Mi permette di dare più peso a situazioni con poche serie vicine alla vittoria che a configurazioni dove ci sono tante pedine ma molto sparse, inoltre tiene conto del numero delle possibili vittorie visto che valuto le serie in ogni direzione a partire da ogni casella. Infine, per assicurare che il nucleo di ragionamento si accorga di situazioni critiche, come la possibilità di vincere con una mossa o il rischio di spianare la strada alla vittoria avversaria, ho aggiunto un controllo ulteriore: un minimax su un albero di altezza 2, che quindi considera una mossa per ogni giocatore. Ciò ha anche il pregio che, in caso venga scovata una “situazione critica” in un nodo, successivamente non venga esplorato il sottoalbero radicato in tale nodo, portando un leggero guadagno computazionale.

Confrontando la mia euristica con quella di alcuni studenti della Valparaiso University [7], mi sembrano abbastanza simili anche se loro applicano il minimax solo a valori -1,0,1 e poi aggiungono altri controlli, mentre la mia è più generica e modulare e di presta facilmente ad aggiustamenti nell’assegnamento dei punteggi.

3.2.5 Rappresentazione della conoscenza nel caso di studio

In tutti e tre le implementazioni il primo problema che mi sono trovato a dover affrontare è stato quello di rappresentare in maniera esaustiva la conoscenza della situazione di gioco.

Data la struttura “a scacchiera”, di 6 righe e 7 colonne, del supporto la scelta più naturale è stata quella di rappresentare la situazione di gioco con una matrice 6×7 con soli tre valori possibili: giocatore1, giocatore2 e vuoto (in JavaScript sono stati rappresentati da 1,-1 e 0, mentre in MiniZinc da 1, 2 e 0), purtroppo in ASP non sono presenti strutture dati come matrici o array, quindi ho dovuto definire un predicato `inpos` di arietà 4 che associasse ad una coppia (riga,colonna) e ad un tempo T, il valore della cella a quelle coordinate. In aggiunta a questa informazione di base bisognava evitare che alcune pedine “galleggiassero” in aria, perché, visto che il supporto del gioco è vericale, la forza di gravità agisce sulle pedine dei giocatori facendo prendere loro la posizione più bassa disponibile nella colonna prescelta. per questo motivo ho aggiunto un vettore di 7 elementi nel modello MiniZinc e nel codice JavaScript che memorizzasse, per ogni colonna, la prima posizione libera disponibile e nel

32 Rappresentazione della conoscenza e Ragionamento Automatico

modello ASP ho dichiarato un predicato che afferma che una cella del supporto è disponibile per una pedina al tempo di gioco T se e solo se non è occupata ed è sul fondo del supporto oppure la cella sotto di essa è occupata.

Queste strutture dati o predicati vengono usati come punto di partenza per generare l'albero di ricerca che esplorera tutte le possibili situazioni, che hanno origine da quella iniziale, dopo un numero limitato di passi (deciso dall'utente). Esse, infine, vengono valutate da una funzione (in JavaScript), da una serie di vincoli che costringono ad alcuni particolari valori gli elementi di un'array multidimensionale (in MiniZinc), da una serie di predicati dei quali mi interessa uno degli argomenti (ma solo se il predicato in questione è vero).

3.3 Tecniche di realizzazione

3.3.1 Il modello MiniZinc

A partire da una matrice iniziale di 6 righe e 7 colonne rappresentante la situazione di gioco corrente e un array di 7 elementi dal quale si ricavano le posizioni nelle quali, per ogni colonna, si può andare a collocare la prossima pedina inserita, dapprima viene vincolato il posizionamento della prossima pedina inserita nella posizione indicata da `firstFFP[i]` dove `i` è la colonna scelta al primo livello dell'albero di ricerca. Ai livelli 1 successivi al primo tale posizione è ricavata da `ffp[1, i]`. L'alternanza nei turni è garantita dall'array `player` che permette agevolmente di ricavare quale giocatore deve muovere in ciascun turno, sfortunatamente però la soluzione è decisamente naive e richiede l'aggiornamento manuale di `player` ogni qualvolta si vogliano cambiare il numero di livelli o il giocatore che fa la prima mossa, però viene controllato dal vincolo subito sotto che sia rispettata l'alternanza, se non viene rispettata il solver restituirà `UNSATISFIABLE`.

Una volta inserite tante pedine quante sono i livelli dell'albero di gioco che vogliono essere esplorati dall'utente e si ha quindi raggiunto una foglia dell'albero viene valutato lo `score` di ogni giocatore, cioè la somma dei punteggi parziali per ognuna delle 9 direzioni: su, giù, sinistra, destra, diagonali e quella centrale, che in realtà è una non-direzione e ha sempre punteggio 0. Tali punteggi parziali sono così calcolati: se nella cella (i,j) c'è una pedina del giocatore di cui stiamo valutando il punteggio, il parziale di ciascuna direzione è 10^n dove n è il numero di occorrenze di pedine di tale giocatore in suddetta direzione, oppure 0 se in tale direzione è impossibile fare 4 o perché si dovrebbe uscire dal supporto o perché ci sono pedine avversarie che lo impediscono. I punteggi parziali e il punteggio finale, come anche le varie situazioni di gioco, sono "memorizzati" tramite vincoli (`constraints`) in elementi di array

multidimensionali o in variabili dedicate. Questa “struttura dati” è formata da `config[livello,riga,colonna]`, che mantiene i valori (0,1 o 2) che sono presenti in ogni posizione del supporto del gioco (riga, colonna) ad ogni livello di profondità del ramo correntemente esplorato dell’albero di ricerca, da `ffp[livello,colonna]`, il quale memorizza per ogni livello del ramo esplorato le posizioni (righe) di ogni colonna in cui può collocarsi una pedina (una sola per ogni colonna con valori in 1..6), `move[livello]`, che per ogni livello ricorda la mossa fatta (la colonna scelta) da uno qualunque dei due giocatori e prende valori in 1..7, `eval[giocatore,riga, colonna,direzione,profondita’]` che ha il compito di registrare per ognuno dei due giocatori, per ogni cella della scacchiera e per ogni direzione che parta da tale casella il punteggio parziale che viene calcolato moltiplicando per 10 il valore della `profondita’` precedente (la profondità 1 ha sempre valore 1) se in quella direzione a quella profondità trovo una pedina del giocatore di cui sto valutando il punteggio, moltiplicando per 1 il valore precedente se trovo 0, moltiplicando per 0 se trovo una pedina avversaria, infine c’è `score` che salva la differenza della somma di tutti i valori di `eval` dei due giocatori a profondità 4.

La soluzione “migliore” è quella che massimizza la differenza tra il punteggio del primo giocatore (che il solver cerca di far vincere) e quello del secondo. Questa massimizzazione non è, purtroppo, realistica infatti sì nella configurazione finale che massimizza il guadagno del primo giocatore, il suo avversario ha messo le sue pedine in modo da spianargli la strada, cosa che raramente succede nel gioco reale. Per simulare più realisticamente il comportamento reale di due giocatori si sarebbe dovuto avere a disposizione (o implementare) una funzione aggregata di minimax.

```

include "globals.mzn";

int: levels;

set of int: LEVS = 1..levels;
set of int: FROW = 1..7;
set of int: ROWS = 1..6;
set of int: COLS = 1..7;
set of int: VALS = 0..2;
par int: coeff = 10;

%% EXTERNAL PARAMETERS
array [ROWS,COLS] of par VALS: firstConfig;
array [COLS] of par FROW: firstFFP;
array [LEVS] of par VALS: player;
```

34 Rappresentazione della conoscenza e Ragionamento Automatico

```
%%%%%
%% INPUTS
% esempio di input
firstConfig = [| 0, 0, 0, 0, 0, 0, 0, 0
                | 0, 0, 0, 0, 0, 0, 0, 0
                | 0, 0, 0, 0, 0, 0, 0, 0
                | 0, 0, 0, 0, 0, 0, 0, 0
                | 0, 0, 0, 0, 0, 0, 0, 0
                | 0, 0, 0, 0, 0, 0, 0, 0
                | 0, 0, 0, 0, 0, 0, 0, 0 |];
firstFFP = [ 1, 1, 1, 1, 1, 1, 1, 1 ];

player = [ 1, 2, 1, 2 ];
levels = 4;
%%%%%

%% INTERNAL VARIABLES
array [LEVS,ROWS,COLS] of var VALS: config ;
array [LEVS,COLS] of var FROW: ffp ;
array [LEVS] of var COLS: move;
array [1..2 ,ROWS,COLS,1..9 ,1..4] of var 0..100000: eval ;
var int: score;
%%%%%

%%%% THIS CONSTRAINT VERIFIES
%%%% IF THE ELEMENTS OF THE ARRAY
%%%% player ARE REALLY ALTERNATE
constraint forall( l in 1..levels-1 )((
    ( player[l] = 1 -> player[l+1] = 2 ) /\(
    ( player[l] = 2 -> player[l+1] = 1 ) ) );

%%%% CONSTRAINTS THAT REDUCE THE SEARCH SPACE...
constraint forall( l in LEVS )
(
    ( l = 1 -> forall( m1 in COLS, i1 in ROWS, j1 in COLS )
        (( move[1] = m1 /\ m1 != j1
            -> config[l,i1,j1] = firstConfig[i1,j1]
            /\ ffp[l,j1] = firstFFP[j1] ) /\(
        ( move[1] = m1 /\ m1 = j1 /\ i1 != firstFFP[j1]
            -> config[l,i1,j1] = firstConfig[i1,j1] ) /\
```

```

( move[1] = m1 /\ m1 = j1 /\ i1 = firstFFP[j1]
  -> config[l,i1,j1] = player[1]
    /\ ffp[l,j1] = firstFFP[j1]+1)
) /\ 
( l > 1 -> forall( m in COLS, i in ROWS, j in COLS )
  (( move[1] = m /\ m != j
    -> config[l,i,j] = config[l-1,i,j]
      /\ ffp[l,j] = ffp[l-1,j] ) /\ 
  ( move[1] = m /\ m = j /\ i != ffp[l-1,j]
    -> config[l,i,j] = config[l-1,i,j] ) /\ 
  ( move[1] = m /\ m = j /\ i = ffp[l-1,j]
    -> config[l,i,j] = player[1]
      /\ ffp[l,j] = ffp[l-1,j]+1 ))));
%%%%%
%% ...EVALUATING THE POSSIBLE CONFIGURATIONS...
%
% se nella cella non c'e' una pedina del giocatore p
% di cui stiamo valutando il punteggio metto 0
% in posizione 1 per tutte le direzioni
constraint forall( p in 1..2, i in ROWS, j in COLS )
( config[levels,i,j] != p
  -> forall( dir in 1..9 )( eval[p,i,j,dir,4] = 0 ));
%
% se nella cella non c'e' una pedina del giocatore
% p di cui stiamo valutando il punteggio metto
% 1 per tutte le direzioni di una qualunque casella
% dove c'e' una pedina del giocatore p di cui stiamo
% valutando il punteggio
%
% per la direzione 5 (central) metto 0
constraint forall( p in 1..2, i in ROWS, j in COLS )
( config[levels,i,j] = p
  -> forall( dir in 1..9 )
    ( eval[p,i,j,dir,1] = 1 )
      /\ eval[p,i,j,5,4] = 0 );
%
%%% i,j<=3: guardo SOLO le direzioni up, right, up-right
% le altre vengono messe a 0, se in tali direzioni
% trovo un valore non 0 ne' p, metto 0 in eval
% senno' metto il numero di occorrenze di p
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i <= 3 /\ j <= 3 /\ config[levels,i,j] = p )

```

36 Rappresentazione della conoscenza e Ragionamento Automatico

```

( eval[p,i,j,1,4] = 0 /\ eval[p,i,j,2,4] = 0
  /\ eval[p,i,j,3,4] = 0 /\ eval[p,i,j,4,4] = 0
  /\ eval[p,i,j,7,4] = 0 );
% up - 8
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i <= 3 /\ j <= 3 /\ config[levels,i,j] = p )
  ( exists( k in 1..3 )( config[levels,i+k,j] != p
    /\ config[levels,i+k,j] != 0 )
    -> eval[p,i,j,8,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
  j in COLS, k in 2..4
  where i <= 3 /\ j <= 3 /\ config[levels,i,j] = p )
  (( config[levels,i+(k-1),j] = p
    -> eval[p,i,j,8,k] = coeff*eval[p,i,j,8,k-1] ) /\
  ( config[levels,i+(k-1),j] = 0
    -> eval[p,i,j,8,k] = eval[p,i,j,8,k-1] )));
% right - 6
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i <= 3 /\ j <= 3 /\ config[levels,i,j] = p )
  ( exists( k in 1..3 )( config[levels,i,j+k] != p
    /\ config[levels,i,j+k] != 0 )
    -> eval[p,i,j,6,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
  j in COLS, k in 2..4
  where i <= 3 /\ j <= 3 /\ config[levels,i,j] = p )
  ( ( config[levels,i,j+(k-1)] = p ->
    eval[p,i,j,6,k] = coeff*eval[p,i,j,6,k-1] ) /\
  ( config[levels,i,j+(k-1)] = 0 ->
    eval[p,i,j,6,k] = eval[p,i,j,6,k-1] ) );
% up-right - 9
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i <= 3 /\ j <= 3 /\ config[levels,i,j] = p )
  ( exists( k in 1..3 )( config[levels,i+k,j+k] != p
    /\ config[levels,i+k,j+k] != 0 )
    -> eval[p,i,j,9,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
  j in COLS, k in 2..4
  where i <= 3 /\ j <= 3 /\ config[levels,i,j] = p )
  ( ( config[levels,i+(k-1),j+(k-1)] = p
    -> eval[p,i,j,9,k] = coeff*eval[p,i,j,9,k-1] ) /\

```

```

( config[levels,i+(k-1),j+(k-1)] = 0
  -> eval[p,i,j,9,k] = eval[p,i,j,9,k-1] ) );
%% i<=3, j=4: guardo SOLO le direzioni up, right,
% up-right, left, up-left le altre vengono messe a 0,
% se in tali direzioni trovo un valore non 0 ne' p metto
% 0 in eval, senno' metto il numero di occorrenze di p
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i <= 3 /\ j = 4 /\ config[levels,i,j] = p )
  ( eval[p,i,j,1,4] = 0 /\ eval[p,i,j,2,4] = 0
    /\ eval[p,i,j,3,4] = 0 );
% up - 8
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i <= 3 /\ j = 4 /\ config[levels,i,j] = p )
  ( exists( k in 1..3 )( config[levels,i+k,j] != p
    /\ config[levels,i+k,j] != 0 )
    -> eval[p,i,j,8,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
  j in COLS, k in 2..4
  where i <= 3 /\ j = 4 /\ config[levels,i,j] = p )
  ( ( config[levels,i+(k-1),j] = p
    -> eval[p,i,j,8,k] = coeff*eval[p,i,j,8,k-1] ) /\
    ( config[levels,i+(k-1),j] = 0
      -> eval[p,i,j,8,k] = eval[p,i,j,8,k-1] ) );
% right - 6
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i <= 3 /\ j = 4 /\ config[levels,i,j] = p )
  ( exists( k in 1..3 )( config[levels,i,j+k] != p
    /\ config[levels,i,j+k] != 0 )
    -> eval[p,i,j,6,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
  j in COLS, k in 2..4
  where i <= 3 /\ j = 4 /\ config[levels,i,j] = p )
  ( ( config[levels,i,j+(k-1)] = p
    -> eval[p,i,j,6,k] = coeff*eval[p,i,j,6,k-1] ) /\
    ( config[levels,i,j+(k-1)] = 0
      -> eval[p,i,j,6,k] = eval[p,i,j,6,k-1] ) );
% up-right - 9
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i <= 3 /\ j = 4 /\ config[levels,i,j] = p )
  ( exists( k in 1..3 )( config[levels,i+k,j+k] != p
    /\ config[levels,i+k,j+k] != 0 )
    -> eval[p,i,j,9,k] = coeff*eval[p,i,j,9,k-1] ) /\
    ( config[levels,i+k,j+k] = 0
      -> eval[p,i,j,9,k] = eval[p,i,j,9,k-1] ) );

```

38 Rappresentazione della conoscenza e Ragionamento Automatico

```

/\ config [ levels ,i+k,j+k] != 0 )
-> eval[p,i,j,9,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
                   j in COLS, k in 2..4
    where i <= 3 /\ j = 4 /\ config [ levels ,i,j] = p )
    ( ( config [ levels ,i+(k-1),j+(k-1)] = p
        -> eval[p,i,j,9,k] = coeff*eval[p,i,j,9,k-1] ) /\
        ( config [ levels ,i+(k-1),j+(k-1)] = 0
        -> eval[p,i,j,9,k] = eval[p,i,j,9,k-1] ) );
% left - 4
constraint forall( p in 1..2, i in ROWS, j in COLS
    where i <= 3 /\ j = 4 /\ config [ levels ,i,j] = p )
    ( exists( k in 1..3 )( config [ levels ,i,j-k] != p
        /\ config [ levels ,i,j-k] != 0 )
        -> eval[p,i,j,4,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
                   j in COLS, k in 2..4
    where i <= 3 /\ j = 4 /\ config [ levels ,i,j] = p )
    ( ( config [ levels ,i,j-(k-1)] = p
        -> eval[p,i,j,4,k] = coeff*eval[p,i,j,4,k-1] ) /\
        ( config [ levels ,i,j-(k-1)] = 0
        -> eval[p,i,j,4,k] = eval[p,i,j,4,k-1] ) );
% up-left - 7
constraint forall( p in 1..2, i in ROWS, j in COLS
    where i <= 3 /\ j = 4 /\ config [ levels ,i,j] = p )
    ( exists( k in 1..3 )( config [ levels ,i+k,j-k] != p
        /\ config [ levels ,i+k,j-k] != 0 )
        -> eval[p,i,j,7,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
                   j in COLS, k in 2..4
    where i <= 3 /\ j = 4 /\ config [ levels ,i,j] = p )
    ( ( config [ levels ,i+(k-1),j-(k-1)] = p
        -> eval[p,i,j,7,k] = coeff*eval[p,i,j,7,k-1] ) /\
        ( config [ levels ,i+(k-1),j-(k-1)] = 0
        -> eval[p,i,j,7,k] = eval[p,i,j,7,k-1] ) );
%%% i<=3, j>=5: guardo SOLO le direzioni
% up, left, up-left le altre vengono messe a 0,
% se in tali direzioni trovo un valore non 0 ne' p metto
% 0 in eval, senno' metto il numero di occorrenze di p
constraint forall( p in 1..2, i in ROWS, j in COLS

```

```

where i <= 3 /\ j >= 5 /\ config[levels,i,j] = p )
( eval[p,i,j,1,4] = 0 /\ eval[p,i,j,2,4] = 0
  /\ eval[p,i,j,3,4] = 0 /\ eval[p,i,j,6,4] = 0
  /\ eval[p,i,j,9,4] = 0 );
% up - 8
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i <= 3 /\ j >= 5 /\ config[levels,i,j] = p )
  ( exists( k in 1..3 )( config[levels,i+k,j] != p
    /\ config[levels,i+k,j] != 0 )
    -> eval[p,i,j,8,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
  j in COLS, k in 2..4
  where i <= 3 /\ j >= 5 /\ config[levels,i,j] = p )
  ( ( config[levels,i+(k-1),j] = p
    -> eval[p,i,j,8,k] = coeff*eval[p,i,j,8,k-1] ) /\
    ( config[levels,i+(k-1),j] = 0
    -> eval[p,i,j,8,k] = eval[p,i,j,8,k-1] ) );
% left - 4
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i <= 3 /\ j >= 5 /\ config[levels,i,j] = p )
  ( exists( k in 1..3 )( config[levels,i,j-k] != p
    /\ config[levels,i,j-k] != 0 )
    -> eval[p,i,j,4,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
  j in COLS, k in 2..4
  where i <= 3 /\ j >= 5 /\ config[levels,i,j] = p )
  ( ( config[levels,i,j-(k-1)] = p
    -> eval[p,i,j,4,k] = coeff*eval[p,i,j,4,k-1] ) /\
    ( config[levels,i,j-(k-1)] = 0
    -> eval[p,i,j,4,k] = eval[p,i,j,4,k-1] ) );
% up-left - 7
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i <= 3 /\ j >= 5 /\ config[levels,i,j] = p )
  ( exists( k in 1..3 )( config[levels,i+k,j-k] != p
    /\ config[levels,i+k,j-k] != 0 )
    -> eval[p,i,j,7,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
  j in COLS, k in 2..4
  where i <= 3 /\ j >= 5 /\ config[levels,i,j] = p )
  ( ( config[levels,i+(k-1),j-(k-1)] = p
    -> eval[p,i,j,7,k] = coeff*eval[p,i,j,7,k-1] ) /\
    ( config[levels,i+(k-1),j-(k-1)] = 0
    -> eval[p,i,j,7,k] = eval[p,i,j,7,k-1] ) );

```

40 Rappresentazione della conoscenza e Ragionamento Automatico

```

    -> eval[p,i,j,7,k] = coeff*eval[p,i,j,7,k-1] ) /\ 
( config[levels,i+(k-1),j-(k-1)] = 0
    -> eval[p,i,j,7,k] = eval[p,i,j,7,k-1] ) );
%% i>=4, j<=3: guardo SOLO le direzioni
% down, right, down-right le altre vengono messe a 0,
% se in tali direzioni trovo un valore non 0 ne' p
% metto 0 in eval, senno' ci metto il numero
% di occorrenze di p
constraint forall( p in 1..2, i in ROWS, j in COLS
    where i >= 4 /\ j <= 3 /\ config[levels,i,j] = p )
( eval[p,i,j,7,4] = 0 /\ eval[p,i,j,8,4] = 0
    /\ eval[p,i,j,9,4] = 0 /\ eval[p,i,j,4,4] = 0
    /\ eval[p,i,j,1,4] = 0 );
% down - 2
constraint forall( p in 1..2, i in ROWS, j in COLS
    where i >= 4 /\ j <= 3 /\ config[levels,i,j] = p )
( exists( k in 1..3 )( config[levels,i-k,j] != p
    /\ config[levels,i-k,j] != 0 )
    -> eval[p,i,j,2,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
    j in COLS, k in 2..4
    where i >= 4 /\ j <= 3 /\ config[levels,i,j] = p )
( ( config[levels,i-(k-1),j] = p
    -> eval[p,i,j,2,k] = coeff*eval[p,i,j,2,k-1] ) /\ 
( config[levels,i-(k-1),j] = 0
    -> eval[p,i,j,2,k] = eval[p,i,j,2,k-1] ) );
% right - 6
constraint forall( p in 1..2, i in ROWS, j in COLS
    where i >= 4 /\ j <= 3 /\ config[levels,i,j] = p )
( exists( k in 1..3 )( config[levels,i,j+k] != p
    /\ config[levels,i,j+k] != 0 )
    -> eval[p,i,j,6,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
    j in COLS, k in 2..4
    where i >= 4 /\ j <= 3 /\ config[levels,i,j] = p )
( ( config[levels,i,j+(k-1)] = p
    -> eval[p,i,j,6,k] = coeff*eval[p,i,j,6,k-1] ) /\ 
( config[levels,i,j+(k-1)] = 0
    -> eval[p,i,j,6,k] = eval[p,i,j,6,k-1] ) );
% down-right - 3

```

```

constraint forall( p in 1..2, i in ROWS, j in COLS
  where i >= 4 /\ j <= 3 /\ config[levels,i,j] = p )
  ( exists( k in 1..3 )( config[levels,i-k,j+k] != p
    /\ config[levels,i-k,j+k] != 0 )
    -> eval[p,i,j,3,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
  j in COLS, k in 2..4
  where i >= 4 /\ j <= 3 /\ config[levels,i,j] = p )
  ( ( config[levels,i-(k-1),j+(k-1)] = p
    -> eval[p,i,j,3,k] = coeff*eval[p,i,j,3,k-1] ) /\
    ( config[levels,i-(k-1),j+(k-1)] = 0
    -> eval[p,i,j,3,k] = eval[p,i,j,3,k-1] ) );
% i>=4, j=4: guardo SOLO le direzioni
% down, right, down-right, left, down-left
% le altre vengono messe a 0, se in tali
% direzioni trovo un valore non 0 ne' p metto 0 in eval,
% senno' metto il numero di occorrenze di p
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i >= 4 /\ j = 4 /\ config[levels,i,j] = p )
  ( eval[p,i,j,7,4] = 0 /\ eval[p,i,j,8,4] = 0
  /\ eval[p,i,j,9,4] = 0 );
% down - 2
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i >= 4 /\ j = 4 /\ config[levels,i,j] = p )
  ( exists( k in 1..3 )( config[levels,i-k,j] != p
    /\ config[levels,i-k,j] != 0 )
    -> eval[p,i,j,2,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
  j in COLS, k in 2..4
  where i >= 4 /\ j = 4 /\ config[levels,i,j] = p )
  ( ( config[levels,i-(k-1),j] = p
    -> eval[p,i,j,2,k] = coeff*eval[p,i,j,2,k-1] ) /\
    ( config[levels,i-(k-1),j] = 0
    -> eval[p,i,j,2,k] = eval[p,i,j,2,k-1] ) );
% right - 6
constraint forall( p in 1..2, i in ROWS, j in COLS
  where i >= 4 /\ j = 4 /\ config[levels,i,j] = p )
  ( exists( k in 1..3 )( config[levels,i,j+k] != p
    /\ config[levels,i,j+k] != 0 )
    -> eval[p,i,j,6,4] = 0 );

```

42 Rappresentazione della conoscenza e Ragionamento Automatico

```

constraint forall( p in 1..2 , i in ROWS,
                   j in COLS, k in 2..4
    where i >= 4 /\ j = 4 /\ config [ levels ,i ,j ] = p )
    ( ( config [ levels ,i ,j+(k-1)] = p
        -> eval [p,i,j,6,k] = coeff*eval [p,i,j,6,k-1] ) /\ 
    ( config [ levels ,i ,j+(k-1)] = 0
        -> eval [p,i,j,6,k] = eval [p,i,j,6,k-1] ) );
% down-right - 3
constraint forall( p in 1..2 , i in ROWS, j in COLS
    where i >= 4 /\ j = 4 /\ config [ levels ,i ,j ] = p )
    ( exists( k in 1..3 )( config [ levels ,i-k,j+k] != p
        /\ config [ levels ,i-k,j+k] != 0 )
        -> eval [p,i,j,3,4] = 0 );
constraint forall( p in 1..2 , i in ROWS,
                   j in COLS, k in 2..4
    where i >= 4 /\ j = 4 /\ config [ levels ,i ,j ] = p )
    ( ( config [ levels ,i-(k-1),j+(k-1)] = p
        -> eval [p,i,j,3,k] = coeff*eval [p,i,j,3,k-1] ) /\ 
    ( config [ levels ,i-(k-1),j+(k-1)] = 0
        -> eval [p,i,j,3,k] = eval [p,i,j,3,k-1] ) );
% left - 4
constraint forall( p in 1..2 , i in ROWS, j in COLS
    where i >= 4 /\ j = 4 /\ config [ levels ,i ,j ] = p )
    ( exists( k in 1..3 )( config [ levels ,i ,j-k] != p
        /\ config [ levels ,i ,j-k] != 0 )
        -> eval [p,i,j,4,4] = 0 );
constraint forall( p in 1..2 , i in ROWS,
                   j in COLS, k in 2..4
    where i >= 4 /\ j = 4 /\ config [ levels ,i ,j ] = p )
    ( ( config [ levels ,i ,j-(k-1)] = p
        -> eval [p,i,j,4,k] = coeff*eval [p,i,j,4,k-1] ) /\ 
    ( config [ levels ,i ,j-(k-1)] = 0
        -> eval [p,i,j,4,k] = eval [p,i,j,4,k-1] ) );
% down-left - 1
constraint forall( p in 1..2 , i in ROWS, j in COLS
    where i >= 4 /\ j = 4 /\ config [ levels ,i ,j ] = p )
    ( exists( k in 1..3 )( config [ levels ,i-k,j-k] != p
        /\ config [ levels ,i-k,j-k] != 0 )
        -> eval [p,i,j,1,4] = 0 );
constraint forall( p in 1..2 , i in ROWS,

```

```

j in COLS, k in 2..4
where i >= 4 /\ j = 4 /\ config[levels,i,j] = p )
( ( config[levels,i-(k-1),j-(k-1)] = p
    -> eval[p,i,j,1,k] = coeff*eval[p,i,j,1,k-1] ) /\ 
    ( config[levels,i-(k-1),j-(k-1)] = 0
    -> eval[p,i,j,1,k] = eval[p,i,j,1,k-1] ) );
%% i>=4, j>=5: guardo SOLO le direzioni
% down, left, down-left le altre vengono
% messe a 0, se in tali direzioni
% trovo un valore non 0 ne' p metto 0
% in eval, senno' metto il numero di occorrenze di p
constraint forall( p in 1..2, i in ROWS, j in COLS
    where i >= 4 /\ j >= 5 /\ config[levels,i,j] = p )
( eval[p,i,j,7,4] = 0 /\ eval[p,i,j,8,4] = 0
  /\ eval[p,i,j,9,4] = 0 /\ eval[p,i,j,6,4] = 0
  /\ eval[p,i,j,3,4] = 0 );
% down - 2
constraint forall( p in 1..2, i in ROWS, j in COLS
    where i >= 4 /\ j >= 5 /\ config[levels,i,j] = p )
( exists( k in 1..3 )( config[levels,i-k,j] != p
  /\ config[levels,i-k,j] != 0 )
  -> eval[p,i,j,2,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
    j in COLS, k in 2..4
    where i >= 4 /\ j >= 5 /\ config[levels,i,j] = p )
( ( config[levels,i-(k-1),j] = p
    -> eval[p,i,j,2,k] = coeff*eval[p,i,j,2,k-1] ) /\ 
    ( config[levels,i-(k-1),j] = 0
    -> eval[p,i,j,2,k] = eval[p,i,j,2,k-1] ) );
% left - 4
constraint forall( p in 1..2, i in ROWS, j in COLS
    where i >= 4 /\ j >= 5 /\ config[levels,i,j] = p )
( exists( k in 1..3 )( config[levels,i,j-k] != p
  /\ config[levels,i,j-k] != 0 )
  -> eval[p,i,j,4,4] = 0 );
constraint forall( p in 1..2, i in ROWS,
    j in COLS, k in 2..4
    where i >= 4 /\ j >= 5 /\ config[levels,i,j] = p )
( ( config[levels,i,j-(k-1)] = p
    -> eval[p,i,j,4,k] = coeff*eval[p,i,j,4,k-1] ) /\ 

```

44 Rappresentazione della conoscenza e Ragionamento Automatico

```

( config [ levels ,i ,j -(k-1)] = 0
  -> eval [p ,i ,j ,4 ,k] = eval [p ,i ,j ,4 ,k-1] ) );
% down-left - 1
constraint forall( p in 1..2 , i in ROWS, j in COLS
  where i >= 4 /\ j >= 5 /\ config [ levels ,i ,j ] = p )
  ( exists( k in 1..3 )( config [ levels ,i-k ,j-k ] != p
    /\ config [ levels ,i-k ,j-k ] != 0 )
    -> eval [p ,i ,j ,1 ,4] = 0 );
constraint forall( p in 1..2 , i in ROWS,
  j in COLS, k in 2..4
  where i >= 3 /\ j >= 5 /\ config [ levels ,i ,j ] = p )
  ( ( config [ levels ,i-(k-1) ,j-(k-1)] = p
    -> eval [p ,i ,j ,1 ,k] = coeff*eval [p ,i ,j ,1 ,k-1] ) /\
    ( config [ levels ,i-(k-1) ,j-(k-1)] = 0
      -> eval [p ,i ,j ,1 ,k] = eval [p ,i ,j ,1 ,k-1] ) );

constraint score =
  sum([ eval [ player [1] ,i ,j ,dir ,4]
    | i in ROWS, j in COLS, dir in 1..9 ])
  - sum([ eval [ player [2] ,i ,j ,dir ,4]
    | i in ROWS, j in COLS, dir in 1..9 ]);

%% ... MAXIMIZING THE EVALUATION
solve :::
  int_search (move ,input_order ,indomain ,complete)
  maximize score;
%%%
%% PRINTING THE OUTPUT...

```

3.3.2 Il modello ASP

Inizialmente dichiaro tutti i domain predicates che mi serviranno in seguito, poi dichiaro che il giocatore che cerco di far vincere è 1, poi definisco queste regole: una cella ha un valore finale che è quello che è presente al tempo t (il tempo finale), definisco `final` perché può essere utile nell'output; dichiaro anche che per ogni accoppiata riga-colonna c'è una coordinata con i loro valori, in ogni posizione del supporto ad ogni istante di gioco ci può essere un solo valore, una cella al tempo T è occupata se in tale posizione il valore non è 0 (quindi c'è la pedina di un giocatore), una cella è disponibile al tempo T se non

è occupata e si trova sul fondo o se non è occupata e si trova immediatamente sopra una cella occupata, dico che ad ogni tempo T ci può essere la mossa di un solo giocatore in una cella qualunque (l'alternanza è garantita dal fatto che muove sempre il giocatore $P : P \bmod 2 != T \bmod 2$ dove T è il tempo corrente di gioco (il turno), per invertire i turni bisogna cambiare $!=$ con $=$; inoltre vincolo il fatto di non poter mettere una pedina in una posizione che non sia disponibile, se ho messo in una casella al tempo T una pedina, tale casella al tempo T sarà `moved`. Poi ho inserito le regole per tenere conto dell'inerzia (al tempo T si devono mantenere tutte le modifiche fatte ai tempi precedenti) quindi al tempo $T+1$, se una cella non è `moved` allora avrà il valore che aveva a tempo T , altrimenti avrà il valore del giocatore che ha eseguito tale mossa. Il predicato `firstMove(colonna)` è vero se il `currentPlayer` ha eseguito una mossa in una cella di tale colonna al tempo 0. In seguito è definito `possible4(tempo,giocatore,riga,colonna,direzione)` che è vero se in una direzione a partire da una cella ad un tempo T è possibile fare 4 in un turno futuro cioè in quella cella c'è una pedina del giocatore che voglio, e nelle posizioni successive (per ogni direzione) non c'è una pedina avversaria.

Il predicato `partialscore(giocatore,riga,colonna,direzione,punteggio)` è soddisfatto se, al tempo finale, a partire da una cella in una direzione è possibile fare 4 e il punteggio è 10 quando c'è una sola pedina di quel giocatore, è 100 se ce ne sono due oppure è 1000 se ce ne sono tre; la direzione 5 (la non-direzione) vale 1 se in tale cella c'è una pedina del giocatore di cui stiamo valutando i punteggi.

Infine ho dichiarato `score(giocatore,totale)` dove `totale` è la somma di tutti i punteggi di `partialscore` per un giocatore; sfortunatamente la somma è eseguita da `clingo` su una set comprehension e quindi somma solo elementi diversi tra loro, perciò in realtà `score` conta il numero massimo di pedine in linea di ogni giocatore (se `score` è 1 ce n'è al massimo una, se è 111 ce ne sono due, se è 1111 ce ne sono tre e se è 11111 ce ne sono quattro). Infine viene massimizzata la differenza tra punteggio totale di `currentPlayer` e quello del suo avversario.

```
% Definizioni dei range delle variabili
row(1..6). col(1..7). val(0..2). player(1..2). dir(1..9).
time(0..t).
currentPlayer(1).

%%%%
% input base (nessuna pedina in gioco)
inpos(0,0,X,Y) :- coord(X,Y).
```

46 Rappresentazione della conoscenza e Ragionamento Automatico

```
%%%  
final(V,X,Y) :- inpos(t,V,X,Y), coord(X,Y), val(V).  
coord(X,Y) :- row(X), col(Y).  
  
% ogni cella un solo valore  
1 { inpos(T,V,X,Y) : val(V) } 1  
:- coord(X,Y), time(T), T < t.  
  
occupied(T,X,Y)  
:- coord(X,Y), val(V), V > 0, inpos(T,V,X,Y), time(T).  
  
% il predicato avail/2, indica le coordinate dove  
% possono essere messe  
% le pedine ad ogni passo  
avail(T,1,Y) :- coord(1,Y), time(T), not occupied(T,1,Y).  
avail(T,X,Y).  
  
:- coord(X,Y), time(T), X > 1, not occupied(T,X,Y),  
occupied(T,X-1,Y).  
  
1{ move(T,P,X,Y) : coord(X,Y), player(P), T\2 != P\2 }1  
:- time(T), T < t.  
:- move(T,P,X,Y), not avail(T,X,Y), time(T), coord(X,Y),  
player(P).  
  
moved(T,X,Y)  
:- move(T,P,X,Y), player(P), coord(X,Y), time(T).  
  
inpos(T+1,P,X,Y)  
:- move(T,P,X,Y), time(T), player(P), row(X), col(Y).  
inpos(T+1,V,X,Y)  
:- not moved(T,X,Y), inpos(T,V,X,Y), time(T), val(V),  
row(X), col(Y).  
  
firstMove(Y)  
:- move(0,P,X,Y), currentPlayer(P), coord(X,Y).  
  
%%%  
possible4(T,P,X,Y,1)
```

```

:- inpos(T,P,X,Y) , not inpos(T,P1,X-1,Y-1) ,
   not inpos(T,P1,X-2,Y-2) , not inpos(T,P1,X-3,Y-3) ,
   player(P) , player(P1) , P != P1 , time(T) , coord(X,Y) .
possible4(T,P,X,Y,2)
:- inpos(T,P,X,Y) , not inpos(T,P1,X-1,Y) ,
   not inpos(T,P1,X-2,Y) , not inpos(T,P1,X-3,Y) ,
   player(P) , player(P1) , P != P1 , time(T) , coord(X,Y) .
possible4(T,P,X,Y,3)
:- inpos(T,P,X,Y) , not inpos(T,P1,X-1,Y+1) ,
   not inpos(T,P1,X-2,Y+2) , not inpos(T,P1,X-3,Y+3) ,
   player(P) , player(P1) , P != P1 , time(T) , coord(X,Y) .
possible4(T,P,X,Y,4)
:- inpos(T,P,X,Y) , not inpos(T,P1,X,Y-1) ,
   not inpos(T,P1,X,Y-2) , not inpos(T,P1,X,Y-3) ,
   player(P) , player(P1) , P != P1 , time(T) , coord(X,Y) .
%non ha senso analizzare la direzione 5
possible4(T,P,X,Y,6)
:- inpos(T,P,X,Y) , not inpos(T,P1,X,Y+1) ,
   not inpos(T,P1,X,Y+2) , not inpos(T,P1,X,Y+3) ,
   player(P) , player(P1) , P != P1 , time(T) , coord(X,Y) .
possible4(T,P,X,Y,7)
:- inpos(T,P,X,Y) , not inpos(T,P1,X+1,Y-1) ,
   not inpos(T,P1,X+2,Y-2) , not inpos(T,P1,X+3,Y-3) ,
   player(P) , player(P1) , P != P1 , time(T) , coord(X,Y) .
possible4(T,P,X,Y,8)
:- inpos(T,P,X,Y) , not inpos(T,P1,X+1,Y) ,
   not inpos(T,P1,X+2,Y) , not inpos(T,P1,X+3,Y) ,
   player(P) , player(P1) , P != P1 , time(T) , coord(X,Y) .
possible4(T,P,X,Y,9)
:- inpos(T,P,X,Y) , not inpos(T,P1,X+1,Y+1) ,
   not inpos(T,P1,X+2,Y+2) , not inpos(T,P1,X+3,Y+3) ,
   player(P) , player(P1) , P != P1 , time(T) , coord(X,Y) .

%%%%%
%calcolo il punteggio dato da ogni
%cella in una certa direzione
% (left-down)
partialscore(P,X,Y,1,10)
:- inpos(t,P,X,Y) , player(P) ,

```

48 Rappresentazione della conoscenza e Ragionamento Automatico

```
coord(X,Y), possible4(t,P,X,Y,1),
1{ inpos(t,P,X-1,Y-1); inpos(t,P,X-2,Y-2);
   inpos(t,P,X-3,Y-3) }1.
partialscore(P,X,Y,1,100)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,1),
   2{ inpos(t,P,X-1,Y-1); inpos(t,P,X-2,Y-2);
      inpos(t,P,X-3,Y-3) }2.
partialscore(P,X,Y,1,1000)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,1),
   3{ inpos(t,P,X-1,Y-1); inpos(t,P,X-2,Y-2);
      inpos(t,P,X-3,Y-3) }3.
% (down)
partialscore(P,X,Y,2,10)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,2),
   1{ inpos(t,P,X-1,Y); inpos(t,P,X-2,Y);
      inpos(t,P,X-3,Y) }1.
partialscore(P,X,Y,2,100)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,2),
   2{ inpos(t,P,X-1,Y); inpos(t,P,X-2,Y);
      inpos(t,P,X-3,Y) }2.
partialscore(P,X,Y,2,1000)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,2),
   3{ inpos(t,P,X-1,Y); inpos(t,P,X-2,Y);
      inpos(t,P,X-3,Y) }3.
% (right-down)
partialscore(P,X,Y,3,10)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,3),
   1{ inpos(t,P,X-1,Y+1); inpos(t,P,X-2,Y+2);
      inpos(t,P,X-3,Y+3) }1.
partialscore(P,X,Y,3,100)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,3),
   2{ inpos(t,P,X-1,Y+1); inpos(t,P,X-2,Y+2);
      inpos(t,P,X-3,Y+3) }2.
```

```

partialscore(P,X,Y,3,1000)
:- inpos(t,P,X,Y), player(P), coord(X,Y),
possible4(t,P,X,Y,3),
3{ inpos(t,P,X-1,Y+1); inpos(t,P,X-2,Y+2);
    inpos(t,P,X-3,Y+3) }3.

% (left)
partialscore(P,X,Y,4,10)
:- inpos(t,P,X,Y), player(P),
coord(X,Y), possible4(t,P,X,Y,4),
1{ inpos(t,P,X,Y-1); inpos(t,P,X,Y-2);
    inpos(t,P,X,Y-3) }1.

partialscore(P,X,Y,4,100)
:- inpos(t,P,X,Y), player(P),
coord(X,Y), possible4(t,P,X,Y,4),
2{ inpos(t,P,X,Y-1); inpos(t,P,X,Y-2);
    inpos(t,P,X,Y-3) }2.

partialscore(P,X,Y,4,1000)
:- inpos(t,P,X,Y), player(P),
coord(X,Y), possible4(t,P,X,Y,4),
3{ inpos(t,P,X,Y-1); inpos(t,P,X,Y-2);
    inpos(t,P,X,Y-3) }3.

% (central => invalid direction)
partialscore(P,X,Y,5,1)
:- inpos(t,P,X,Y), player(P), coord(X,Y).

% (right)
partialscore(P,X,Y,6,10)
:- inpos(t,P,X,Y), player(P),
coord(X,Y), possible4(t,P,X,Y,6),
1{ inpos(t,P,X,Y+1); inpos(t,P,X,Y+2);
    inpos(t,P,X,Y+3) }1.

partialscore(P,X,Y,6,100)
:- inpos(t,P,X,Y), player(P),
coord(X,Y), possible4(t,P,X,Y,6),
2{ inpos(t,P,X,Y+1); inpos(t,P,X,Y+2);
    inpos(t,P,X,Y+3) }2.

partialscore(P,X,Y,6,1000)
:- inpos(t,P,X,Y), player(P), coord(X,Y),
possible4(t,P,X,Y,6),
3{ inpos(t,P,X,Y+1); inpos(t,P,X,Y+2);
    inpos(t,P,X,Y+3) }3.

```

50 Rappresentazione della conoscenza e Ragionamento Automatico

```
% (left-up)
partialscore(P,X,Y,7,10)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,7),
   1{ inpos(t,P,X+1,Y); inpos(t,P,X+2,Y-2);
       inpos(t,P,X+3,Y-3) }1.

partialscore(P,X,Y,7,100)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,7),
   2{ inpos(t,P,X+1,Y); inpos(t,P,X+2,Y-2);
       inpos(t,P,X+3,Y-3) }2.

partialscore(P,X,Y,7,1000)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,7),
   3{ inpos(t,P,X+1,Y); inpos(t,P,X+2,Y-2);
       inpos(t,P,X+3,Y-3) }3.

% (up)
partialscore(P,X,Y,8,10)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,8),
   1{ inpos(t,P,X+1,Y); inpos(t,P,X+2,Y-2);
       inpos(t,P,X+3,Y) }1.

partialscore(P,X,Y,8,100)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,8),
   2{ inpos(t,P,X+1,Y); inpos(t,P,X+2,Y-2);
       inpos(t,P,X+3,Y) }2.

partialscore(P,X,Y,8,1000)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,8),
   3{ inpos(t,P,X+1,Y); inpos(t,P,X+2,Y-2);
       inpos(t,P,X+3,Y) }3.

% (right-up)
partialscore(P,X,Y,9,10)
:- inpos(t,P,X,Y), player(P),
   coord(X,Y), possible4(t,P,X,Y,9),
   1{ inpos(t,P,X+1,Y-1); inpos(t,P,X+2,Y-2);
       inpos(t,P,X+3,Y+3) }1.

partialscore(P,X,Y,9,100)
:- inpos(t,P,X,Y), player(P),
```

```

coord(X,Y) , possible4(t,P,X,Y,9) ,
2{ inpos(t,P,X+1,Y-1); inpos(t,P,X+2,Y-2);
  inpos(t,P,X+3,Y+3) }2.
partialscore(P,X,Y,9,1000)
:- inpos(t,P,X,Y) , player(P),
  coord(X,Y) , possible4(t,P,X,Y,9),
  3{ inpos(t,P,X+1,Y-1); inpos(t,P,X+2,Y-2);
    inpos(t,P,X+3,Y+3) }3.

score(P,TOT) :- TOT =
  #sum{ PS : partialscore(P,X,Y,D,PS) } , player(P).
#maximize
{ N-M : score(P,N) , score(Q,M) , player(P) , player(Q) ,
  not currentPlayer(Q) , currentPlayer(P) }.

%% #show move/4.   %togliere i commenti se si vogliono
                  %avere maggiori info
%% #show final/3.  %sul risultato
#show score/2.
#show firstMove/1.

```

3.3.3 L'algoritmo JavaScript

Presi in input 5 parametri, dove il primo rappresenta a che livello siamo all'interno dell'albero di ricerca, il secondo l'altezza di suddetto albero, il terzo quale giocatore sta “pensando” alla mossa da eseguire, il quarto la configurazione all'istante considerato e il quinto le prime posizioni libere di ogni colonna. Se si è raggiunto il livello massimo di profondità richiesto dall'utente (`searchDepth == 0`) allora chiamo la funzione `evaluator` con parametri la configurazione raggiunta in quella foglia del sotto-albero di gioco e il giocatore per cui si deve valutare la situazione.

Tale funzione ritorna un numero così calcolato: su una matrice tridimensionale $6 \times 7 \times 9$, che mantiene tutti i valori di ogni direzione che parte da ogni possibile riga e colonna, metto per ogni cella dove non è presente una pedina del giocatore che mi interessa metto 0 ad ogni direzione; se invece la pedina c’è allora metto 1 alla direzione 5 e 10^n alle altre direzioni, dove n è il numero di pedine di quel giocatore presenti, in una stessa direzione, nelle tre caselle adiacenti a quella presa in considerazione; a meno che in quelle tre caselle non ci sia una tessera avversaria o si esca dal supporto, in tal caso metto 0; infine somma tutti gli elementi di questa matrice.

52 Rappresentazione della conoscenza e Ragionamento Automatico

Altrimenti la funzione `minimax` dapprima istanzia le sette configurazioni che si possono avere immediatamente dopo la mossa di un giocatore e i relativi array delle prime posizioni disponibili e li riempie. Successivamente va in ricorsione sul livello successivo dell'albero di ricerca dando come input alla chiamata una delle configurazioni appena generate, il relativo array delle posizioni e il parametro `searchDepth` decrementato di 1.

Il risultato di tale chiamata verrà messo in un array di supporto insieme alle altre sei chiamate ricorsive (una per ognuna delle altre possibili configurazioni) e a quel punto se siamo ad un livello di altezza pari allora ritorniamo il valore massimo fra quelli nell'array, se invece è un nodo ad altezza dispari bisognerà tornare il minimo.

Il livello più alto, infine, comunicherà all'interfaccia grafica la colonna dove ha deciso di mettere la pedina.

```
function evaluator( conf , player ){  
    var score1 = 0;  
    var score2 = 0;  
    for (var row = 0; row < 6; row++) {  
        for (var col = 0; col < 7; col++) {  
            //per ogni casella controllo una direzione solo se  
            //dopo tre caselle resto ancora dentro i limiti della  
            //scacchiera  
            if (conf[row][col] == player) {  
                //valuto la bonta' della disposizione delle tessere  
                //di player  
                if (row < 3 && col < 3) {  
                    //caselle in basso a destra nella scacchiera , con-  
                    //trollo verso l'alto , verso sx e in diagonale su-sx  
                    if (conf[row][col+1] != player*(-1) &&  
                        conf[row][col+2] != player*(-1) &&  
                        conf[row][col+3] != player*(-1)) {  
                        var incr = 1;  
                        if (conf[row][col+1] == player) { incr *= 100; }  
                        if (conf[row][col+2] == player) { incr *= 100; }  
                        if (conf[row][col+3] == player) { incr *= 100; }  
                        if (conf[row][col+1] == player &&  
                            conf[row][col+2] == player &&  
                            conf[row][col+3] == player){  
                            return 1000000000; }  
                        score1 += incr;  
                    }  
                }  
            }  
        }  
    }  
}
```

```

if ( conf [ row+1][ col+1] != player*(-1) &&
    conf [ row+2][ col+2] != player*(-1) &&
    conf [ row+3][ col+3] != player*(-1)) {
    var incr = 1;
    if (conf [ row+1][ col+1] == player) { incr *= 100; }
    if (conf [ row+2][ col+2] == player) { incr *= 100; }
    if (conf [ row+3][ col+3] == player) { incr *= 100; }
    if (conf [ row+1][ col+1] == player &&
        conf [ row+2][ col+2] == player &&
        conf [ row+3][ col+3] == player){
        return 1000000000; }
    score1 += incr;
}
if (conf [ row+1][ col ] != player*(-1) &&
    conf [ row+2][ col ] != player*(-1) &&
    conf [ row+3][ col ] != player*(-1)) {
    var incr = 1;
    if (conf [ row+1][ col ] == player) { incr *= 100; }
    if (conf [ row+2][ col ] == player) { incr *= 100; }
    if (conf [ row+3][ col ] == player) { incr *= 100; }
    if (conf [ row+1][ col ] == player &&
        conf [ row+2][ col ] == player &&
        conf [ row+3][ col ] == player){
        return 1000000000; }
    score1 += incr;
}
} else if (row < 3 && col == 3) {
//caselle in basso centrali: guardo a sx, a dx, in
//su e sulle due diagonali
if (conf [ row ][ col+1] != player*(-1) &&
    conf [ row ][ col+2] != player*(-1) &&
    conf [ row ][ col+3] != player*(-1)) { ... }
if (conf [ row+1][ col+1] != player*(-1) &&
    conf [ row+2][ col+2] != player*(-1) &&
    conf [ row+3][ col+3] != player*(-1)) { ... }
if (conf [ row+1][ col ] != player*(-1) &&
    conf [ row+2][ col ] != player*(-1) &&
    conf [ row+3][ col ] != player*(-1)) { ... }
if (conf [ row ][ col -1] != player*(-1) &&
    conf [ row ][ col -2] != player*(-1) &&
    conf [ row ][ col -3] != player*(-1)) { ... }

```

54 Rappresentazione della conoscenza e Ragionamento Automatico

```

        conf [ row ] [ col -3] != player*(-1)) { ... }

if ( conf [ row+1][ col -1] != player*(-1) &&
      conf [ row+2][ col -2] != player*(-1) &&
      conf [ row+3][ col -3] != player*(-1)) { ... }

} else if (row < 3 && col > 3) {
    ...
} else if (row >= 3 && col < 3) {
    ...
} else if (row >= 3 && col == 3) {
    if (conf [ row ] [ col+1] != player*(-1) &&
        conf [ row ] [ col+2] != player*(-1) &&
        conf [ row ] [ col+3] != player*(-1)) { ... }

    if ( conf [ row-1][ col+1] != player*(-1) &&
          conf [ row-2][ col+2] != player*(-1) &&
          conf [ row-3][ col+3] != player*(-1)) { ... }

    if ( conf [ row-1][ col ] != player*(-1) &&
          conf [ row-2][ col ] != player*(-1) &&
          conf [ row-3][ col ] != player*(-1)) { ... }

    if ( conf [ row ][ col-1] != player*(-1) &&
          conf [ row ][ col-2] != player*(-1) &&
          conf [ row ][ col-3] != player*(-1)) { ... }

    if ( conf [ row-1][ col-1] != player*(-1) &&
          conf [ row-2][ col-2] != player*(-1) &&
          conf [ row-3][ col-3] != player*(-1)) { ... }

} else if (row >= 3 && col > 3) {
    ...
} else {
    alert ("ERROR_in_evaluation");
}
} else if (conf [ row ] [ col ] == player*(-1)) {
    //valuto la bonta' della disposizione delle tessere
    //dell'avversario di player
    if (row < 3 && col < 3) {
        if (conf [ row ] [ col+1] != player &&
            conf [ row ] [ col+2] != player &&
            conf [ row ] [ col+3] != player) {

            var incr = 1;
            if (conf [ row ] [ col+1]==player*(-1)){ incr *= 100; }
            if (conf [ row ] [ col+2]==player*(-1)){ incr *= 100; }
            if (conf [ row ] [ col+3]==player*(-1)){ incr *= 100; }
        }
    }
}
```

```

if (conf[row][col+1]==player*(-1) &&
    conf[row][col+2]==player*(-1) &&
    conf[row][col+3]==player*(-1)){
    return -999999999999;
}
score2 += incr;
}
if (conf[row+1][col+1] != player &&
    conf[row+2][col+2] != player &&
    conf[row+3][col+3] != player) {
var incr = 1;
if (conf[row+1][col+1]==player*(-1)){incr *= 100;}
if (conf[row+2][col+2]==player*(-1)){incr *= 100;}
if (conf[row+3][col+3]==player*(-1)){incr *= 100;}
if (conf[row+1][col+1]==player*(-1) &&
    conf[row+2][col+2] == player &&
    conf[row+3][col+3] == player*(-1)){
    return -999999999999;
}
score2 += incr;
}
if (conf[row+1][col] != player &&
    conf[row+2][col] != player &&
    conf[row+3][col] != player) {
var incr = 1;
if (conf[row+1][col]==player*(-1)){incr *= 100; }
if (conf[row+2][col]==player*(-1)){incr *= 100; }
if (conf[row+3][col]==player*(-1)){incr *= 100; }
if (conf[row+1][col]==player*(-1) &&
    conf[row+2][col]==player*(-1) &&
    conf[row+3][col]==player*(-1)){
    return -999999999999;
}
score2 += incr;
}
} else if (row < 3 && col == 3) {
if (conf[row][col+1] != player &&
    conf[row][col+2] != player &&
    conf[row][col+3] != player) {...}
if (conf[row+1][col+1] != player &&
    conf[row+2][col+2] != player &&
    conf[row+3][col+3] != player) {...}
if (conf[row+1][col] != player &&
    conf[row+2][col] != player &&
    conf[row+3][col] != player) {...}

```

56 Rappresentazione della conoscenza e Ragionamento Automatico

```
        conf[row+2][col] != player &&
        conf[row+3][col] != player) {...}
    if (conf[row][col-1] != player &&
        conf[row][col-2] != player &&
        conf[row][col-3] != player) {...}
    if (conf[row+1][col-1] != player &&
        conf[row+2][col-2] != player &&
        conf[row+3][col-3] != player) {...}
} else if (row < 3 && col > 3) {
    ...
} else if (row >= 3 && col < 3) {
    ...
} else if (row >= 3 && col == 3) {
    if (conf[row][col+1] != player &&
        conf[row][col+2] != player &&
        conf[row][col+3] != player) {...}
    if (conf[row-1][col+1] != player &&
        conf[row-2][col+2] != player &&
        conf[row-3][col+3] != player) {...}
    if (conf[row-1][col] != player &&
        conf[row-2][col] != player &&
        conf[row-3][col] != player) {...}
    if (conf[row][col-1] != player &&
        conf[row][col-2] != player &&
        conf[row][col-3] != player) {...}
    if (conf[row-1][col-1] != player &&
        conf[row-2][col-2] != player &&
        conf[row-3][col-3] != player) {...}
} else if (row >= 3 && col > 3) {
    ...
} else {
    alert("ERROR_in_evaluation");
}
}
}
}
return score1-score2;
}

function generateNextConfig(player, config, ffps) {
```

```

var configs = new Array(7);
for (var mov = 0; mov < 7; mov++) {
    configs[mov] = new Array(6);
    for (var row = 0; row < 6; row++) {
        configs[mov][row] = new Array(7);
        for (var col = 0; col < 7; col++) {
            if (col == mov && row == ffps[mov][col]-1) {
                configs[mov][row][col] = player;
            } else {
                configs[mov][row][col] = config[row][col];
            }
        }
    }
}
return configs;
}

function generateNextFFPs(curFFP) {
    var ffps = new Array(7);
    for (var i = 0; i < 7; i++) {
        ffps[i] = new Array(7);
        for (var j = 0; j < 7; j++) {
            if (i == j) {
                ffps[i][j] = curFFP[j]+1;
            } else {
                ffps[i][j] = curFFP[j];
            }
        }
    }
    return ffps;
}

function minimax(searchDepth, maxDepth, player,
                 curConfig, curFFP) {
    var vals = new Array(7);
    if (searchDepth > 0) {
        // alterno fra mossa di 1 e mossa di -1
        if (searchDepth % 2 == 0) {
            // genero il prossimo step

```

58 Rappresentazione della conoscenza e Ragionamento Automatico

```
var ffps = generateNextFFPs(curFFP);
var configs =
    generateNextConfig(player, curConfig, ffps);
//ricorsione
for (var i = 0; i < 7; i++) {
    vals[i] = minimax(searchDepth - 1, maxDepth, player,
        configs[i], ffps[i]);
}
//scelgo il max
vals.sort(function(a, b){return a-b});
return vals[6];
} else {
    //genero il prossimo step
    var ffps = generateNextFFPs(curFFP);
    var configs =
        generateNextConfig(player * (-1), curConfig, ffps);
    //ricorsione
    for (var i = 0; i < 7; i++) {
        vals[i] = minimax(searchDepth - 1, maxDepth, player,
            configs[i], ffps[i]);
    }
    //scelgo il min
    vals.sort(function(a, b){return a-b});
    return vals[0];
}
} else {
    //valuto la config che mi arriva
    //la divido per maxDepth per dare peso maggiore
    //alle valutazioni fatte piu' in alto nell'albero
    //servira' nel raffinamento iterativo
    return evaluator(curConfig, player) / maxDepth;
}

function move( searchDepth, firstPlayer, firstConfig,
    firstFFP ) {
    //passo 0 (caso base) del minimax, ritorna la colonna
    //che da il punteggio migliore dato dal minimax
    var vals = new Array( 7 );
```

```
var ffps = generateNextFFPs(firstFFP);
var configs =
    generateNextConfig(player, firstConfig, ffps);
var earlyEval = 0;
var level = 1;
for (var i = 0; i < 7; i++) {
    //faccio un controllo rapido per cercare situazioni
    //critiche, tipo vittoria o sconfitta in una mossa
    earlyEval = minimax(1, 1, firstPlayer,
                        configs[i], ffps[i]);
    //il minimax ritorna 1.000.000.000.000 se una certo
    //ramo porta alla vittoria in una mossa
    //e 999.999.999.999 se fa evitare una sconfitta sicura
    if (earlyEval == 1000000000) {
        vals[i] = earlyEval;
    } else if (earlyEval == -99999999999) {
        vals[i] = earlyEval;
    } else {
        //senno' valuto tutte le possibilita'
        vals[i] = minimax(searchDepth - 1, searchDepth,
                           firstPlayer, configs[i], ffps[i]);
    }
}
//supp serve per poter trovare il massimo in vals
//senza modificare vals, cosi' da poter capire in quale
//posizione si trova
var supp = new Array(7);
for (var i = 0; i < 7; i++) {
    supp[i] = vals[i];
}
supp.sort(function (a, b) { return a - b; });
for (var i = 6; i >= 0; i--) {
    var chosenCol = vals.indexOf(supp[i]);
    //se la miglior colonna dove mettere la pedina e' piena
    //prendo la seconda migliore e cosi' via
    if (firstFFP[chosenCol] <= 5) {
        return chosenCol;
    }
}
//se tutti gli elementi di vals sono uguali si sceglie
```

60 Rappresentazione della conoscenza e Ragionamento Automatico

```
//una colonna qualsiasi
while (true) {
    var randomCol = Math.round(Math.random() * 6);
    if (firstFFP[randomCol] <= 5) {
        return randomCol;
    }
}
```

Capitolo 4

Conclusioni

Ho portato a termine lo sviluppo di un videogioco funzionante nato dall'integrazione di un'interfaccia grafica tridimensionale e di un nucleo di ragionamento basato sull'algoritmo *minimax*, dopo aver esplorato le soluzioni dichiarative offerte da MiniZinc e da Answer Set Programming.

Come spesso accade, però la conclusione di un progetto lascia sempre qualche questione in sospeso. Nel prossimo futuro quindi ci lavorerò ancora sopra al fine di migliorare ulteriormente il comparto grafico, aggiungendo le ombre (magari combinando shadowing statico e dinamico) e shader per materiali particolari come le superfici in vetro (anche opaco), riflettenti e anisotropiche, inoltre vorrei fare in modo che la caduta delle pedine non sia una semplice animazione ma che risponda alle leggi della fisica reale e, infine, sull'intelligenza artificiale, ho intenzione di implementare l'alpha-beta pruning, il raffinamento iterativo (per cui ho già gettato alcune basi) e altre migliorie per rendere questo progetto davvero completo.

Il codice sorgente completo lo potete trovare su GitHub al seguente link <https://github.com/francesco-graphics/Graphics-4-in-a-row.git> (codice in fase di aggiornamento).

Capitolo 5

Ringraziamenti

Per essere riuscito a raggiungere questo importante traguardo devo ringraziare molte persone:

- i miei genitori che mi hanno permesso di portare a termine gli studi nelle migliori condizioni possibili e che, insieme a mio fratello, hanno sopportato anche le mie intemperanze dell'ultimo, durissimo, mese e nonostante ciò mi vogliono ancora bene;
- la mia ragazza che mi ha sostenuto sempre, incitandomi e spronandomi a dare sempre di più, spero che tu voglia continuare a farlo sempre;
- tutto il resto della mia famiglia;
- i miei amici (di più o meno lunga data) che con una risata hanno alleggerito le mie fatiche
- ultimo, ma decisamente non meno importante, il mio relatore, il Prof. Dovier, che mi ha aiutato moltissimo ed è stato sempre più che disponibile anche quando la mia insistenza era forse eccessiva.

Capitolo 6

Bibliografia

- [1] Agostino Dovier, Andrea Formisano. *Programmazione Dichiarativa in Prolog, CLP e ASP*. Università di Udine, Università di Perugia.
- [2] Grigoris Antoniou. *Nonmonotonic Reasoning*. The MIT Press, mar 1997.
- [3] Ivan Bratko. *Programmare in PROLOG per l'intelligenza artificiale*. Addison-Wesley/Masson, 1988.
- [4] Leon Chen. Can't stop, won't stop: the 2016 mobile games market report. *Unity Analytics*, Febbraio 2017.
- [5] Naty Hoffman. Physics and math of shading. In *SIGGRAPH conference*, 2015.
- [6] James T. Kajiya. The rendering equation. *SIGGRAPH Computer Graphics*, 20(4):143–150, 1986.
- [7] Kirk Baly, Andrew Freeman, Andrew Jarratt, Kyle Kling, Owen Prough. Teaching computers to ink: Analysis of artificial intelligence and connect four. Technical report, Valparaiso University, 2012.
- [8] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*. Third Edition. Cengage Learning, 2012.
- [9] Vladimir Lifschitz. *What Is Answer Set Programming?* University of Texas at Austin, 2008.
- [10] The MiniZinc Team. *Specifications of MiniZinc*, nov 2016.
- [11] Stuart Jonathan Russell, Peter Norvig. *Artificial Intelligence, A Modern Approach*, chapter 5: Game Playing, pages 122–145. Prentice-Hall, dec 1994.

- [12] Tomas Akenine Möller, Eric Haines, Naty Hoffmann. *Real-Time Rendering*. Third Edition. A K Peters, 2008.