Projekt MI - Treed

Jadran Kotnik, Tim Čas, Martin Konečnik

5. februar 2013

Kazalo

1 Uvod **3**
1.1 Kratek opis projekta . . . 3
1.2 Ideje in cilji . . . 3
1.3 Člani skupine . . . 3

2 Delo na projektu **4**
2.1 Opis dela . . . 4
2.2 Delitev dela . . . 4

3 Algoritmi **5**
3.1 Algoritem A* . . . 5
3.1.1 Implementacija . . . 5
3.1.2 Hevristika . . . 5
3.1.3 Možne izboljšave . . . 6
3.2 Logika, dogajanje med posameznimi stanji . . . 7
3.2.1 Odstranjevanje mrtvih entitet . . . 7
3.2.2 Izvajanje napadov posameznih entitet . . . 7
3.2.3 Iskanje najbližje enote/stolpa . . . 7
3.3 PNG . . . 8

1 Uvod

1.1 Kratek opis projekta

Treed je "tower defense" igra z odprtimi potmi - to pomeni, da se lahko stolpi gradijo na (skoraj) vseh točkah, prav tako pa lahko sovragi po le-teh hodijo. Tako lahko igralec s taktično postavitvijo stolpov poskrbi, da bodo sovragi morali vzeti daljšo pot do cilja (baze na sredini). Za razliko od "klasičnih" iger tega tipa je igralcu dovoljeno s stolpi popolnoma zazidati bazo, toda le-ti stolpi so uničljivi - sovragi jih lahko (in tudi jih) napadajo in tudi uničijo.

1.2 Ideje in cilji

Glavni cilj projekta je bil izdelati mobilno igro za naprave z operacijskim sistemom Android.

1.3 Člani skupine

- Jadran Kotnik
- Tim Čas
- Martin Konečnik

2 Delo na projektu

2.1 Opis dela

Letošnje leto smo celotni projekt začeli pisati znova, saj je bila prejšna različica 2D namizni demo, mi pa smo morali narediti 3D igro za mobilne naprave. Za enostavnejšo sihronizacijo najnovejše različice kode smo uporabili `git`, sistem za upravljanje izvorne kode. Na projektu smo večinoma delali izmenično (izjemoma tudi hkrati), da smo se izognili morebitnim problemom z združevanjem kode.

2.2 Delitev dela

Delo smo si razdelili približno tako[1]:

- Tim Čas: iskanje poti
- Jadran Kotnik: umetna inteligenca, entitete
- Martin Konečnik: ostalo (pomožni razredi, npr. `Vector2f`, `BoundingBox`)

Prvotno smo v projekt hoteli vključiti tudi particle engine, vendar pogon jPCT tega ne omogoča. Lahko bi implementirali svoj particle engine, vendar bi zaradi samega pogona to zelo upočasnilo delovanje igre, saj bi za vsak delec morali uporabiti nov objekt (pri testiranjih na telefonu se je pri 100 objektih na zaslonu framerate znižal na dobrih 20, pri 200 objektih pa se je igra zaradi obremenitve sesula). Ker žal tega ni nihče od nas prej preverjal smo pač morali particle engine izpustiti (prav tako smo morali odstraniti nekaj enot).

Prav tako smo v načrtu imeli prijateljske enote, vendar smo jih zaradi pogona odstraniti, saj je igra dokaj hitro postala neodzivna.

V kolikor bi vse to vedeli prej, bi sigurno izbrali drug pogon, saj smo kasneje na spletu našli mnogo pogonov, ki lahko renderirajo zelo kompleksne scene (npr: drevesa z listjem). Med vsemi pogoni je najbolj izstopal Rajawali, ki pa ga pred tem nikoli nismo opazili (dokaj popularen je postal v zadnjih mesecih).

[1] To je le okviren pregled kdo je vodil kakšno sekcijo, dejansko smo vsi delali na vsem.

3 Algoritmi

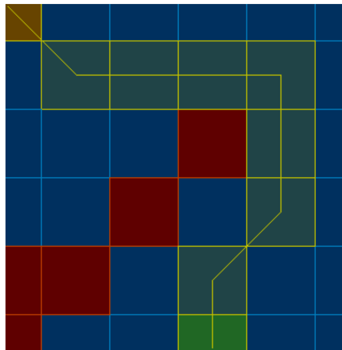
3.1 Algoritem A*

A* algoritem je zelo učinkovit algoritem za iskanje poti. Algoritem je zanesljiv, kar pomeni da vedno najde pot, kadar le-ta obstaja.

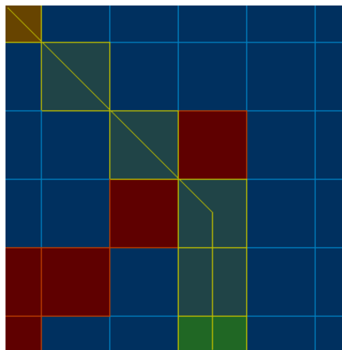
3.1.1 Implementacija

Algoritem je napisan v istem jeziku kot igra — jezik Java. To omogoča preprosto uporabo te implementacije iz igre same.

Igralno polje se razdeli v polje velikosti NxM (enako kot sama igra). Privzeto implementacija dovoli diagonalne premike, toda ne skozi zidove (gl. sliki).



Slika 1: Primer poti v okolici zida. Kot se vidi, algoritem dovoli diagonalno pot, razen kadar je agent blizu zida, saj bi takrat šel delno skozi zid.



Slika 2: Primer poti ko je diagonalno potovanje v okolici zida dovoljeno. Kot je videno, se dolžina (ter obstoj) poti lahko zelo spremeni v odvisnosti od te nastavitve.

Ker so stolpi uničljivi, lahko, namesto da prepovemo vso potovanje okoli stolpov, le-tem damo zelo visoko ceno — tako bodo enote raje uničile stolpe, namesto da vzamejo zelo dolgo pot okoli.

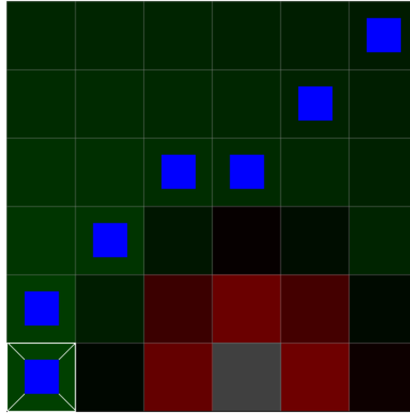
Implementacija omogoča več ciljev hkrati — tako lahko za cilje določimo npr. vse stolpe, nato pa algoritem določi pot do najbližjega stolpa.

3.1.2 Hevristika

Za hevristiko (H score) ter ceno (G score) se uporablja evklidska razdalja:

$$\mathrm{d}(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2} \quad (1)$$

Implementacija omogoča poljubno hevristiko in točkovanje – tako lahko npr. dodamo polja z različnimi cenami, kot je npr. 1× za vodo, 2× za blato, 3× za vodo. Prav tako lahko povečamo ceno polj, ki so blizu sovražnim stolpom ali enotam — tako lahko omogočimo, da se npr. taktične enote izogibajo stolpom kolikor se le da, namesto da bi šle po najbližji poti do cilja. Paziti moramo le da je hevristika še vedno



Slika 3: Primer potencialnega polja (zeleno - privlak, rdeče - odboj, modro - pot)

konsistentna, sicer je možno, da algoritem ne najde optimalne poti. Konsistentna ali monotona hevristika je tista, za katero velja:

$$\mathrm{h}(N) \leq \mathrm{c}(N, P) + \mathrm{h}(P) \tag{2}$$

$$\mathrm{h}(G) = 0 \tag{3}$$

Kjer so:

- N – točka v grafu (v našem primeru NxM mreži)
- P – naslednik od N
- G – ciljna točka

Vsaka konsistentna hevristika je optimistična, kar pomeni da nikoli ne preceni cene do cilja.

Možna je tudi hevristika $\mathrm{h}(N) = 0$, pri čemer se algoritem "prelevi" v iskanje v širino.

3.1.3 Možne izboljšave

Implementacija bi se dala optimizirati predvsem v seznamih — namesto linearnega iskanja v zaprti množici bi lahko uporabili binarno iskalno drevo, hash tabelo ipd.

Kot že zgoraj omenjeno, bi lahko stolpom dodali visoko ceno (implementacija bi potrebovala manjšo spremembo da lahko išče skozi stolpe, kljub temu da se štejejo kot neprehodni) in v UI agentov dodali to, da se v tem primeru ustavijo pri stolpu (namesto da poskušajo iti skozi stolpe).

Namesto (ali pa v kombinaciji z) algoritma A* bi lahko uporabili potencialna polja, ki bi tudi omogočala agentom premikanje – navadno je problem v potencialnih poljih je v lokalnih maksimumih, saj agenti tam obtičijo. Tukaj bi bilo to manjši problem, saj bi agenti namesto baze šli napasti stolpe ali druge sovražne enote.

3.2 Logika, dogajanje med posameznimi stanji

Ob vsaki posodobitvi stanja v igri se najprej dodajo sovražne in prijateljske enote. Sovražne enote imajo 2×večjo verjetnost da se dodajo v igro kot pa prijateljske enote. Za tem sledi posodobitev celotne stopnje, ki pa je razdeljena na naslednje dele:

- odstranjevanje mrtvih entitet
- dodajanje nove ciljne točke prijateljskim enotam (dodeli se naključna točka iz okolice baze)
- posodobitev vseh entitet (lokacija, smer)
- izvajanje napadov posameznih entitet

3.2.1 Odstranjevanje mrtvih entitet

Za mrtve entitete se štejejo vse entitete katerim je zdravje padlo pod 0 in imajo zastavico `isImmortal` postavljeno na `false`. Poleg samega odstranjevanja entitet je potrebno izvesti tudi čiščenje stopnje v primeru da gre za stolp (polje označimo kot prosto), ter poženemo A* algoritem, da osvežimo poti posameznih entitet. Prav tako pa se igralcu dodeli določeno število cekinov, odvisno od tega koliko je bila enota vredna. Na koncu se entiteta le še odstrani iz seznama vseh entitet.

3.2.2 Izvajanje napadov posameznih entitet

Preden se napad izvede najprej poiščemo najbližjo entiteto, ki jo lahko napademo. Iskanje najbližje entitete je podrobneje opisano v 3.2.3. Če smo našli entiteto, ki je v dosegu trenutne entitete glede na njen tip izvedemo naslednje akcije:

- kamikaze
  - napade vse enote v dosegu in se razstreli (nastavimo življenje na 0)
- splash tower
  - napade vse enote v dosegu
- vse ostale enote in stolpi
  - napade najdeno entiteto

Vsi napadi se izvedejo s pomočjo izstrelkov. Zato se ob uspešnem napadu generira izstrelek glede na tip entitete, ki je napad sprožila, in se doda v seznam vseh entitet. Izstrelek bo naredil tarči škodo takrat, ko bo njegova lokacija enaka lokaciji tarče (pri tem upoštevamo računsko napako epsilon, ki je odvisna od hitrosti).

3.2.3 Iskanje najbližje enote/stolpa

Pri iskanju najbližje enote oziroma stolpa se upošteva prioriteta posameznih enot. Prijateljski stolpi bodo tako najprej napadali kamikaze enote, medtem ko bodo sovražne enote izbirale ali stolpe ali pa enote. Vse sovražne enote bodo najprej poskušali napasti glavno bazo. V trenutni implementaciji imajo enote samo eno prioritetno tarčo (zaradi manjšega števila enot seznam ni potreben), kar se bi lahko enostavno razširilo za poljubno število enot s pomočjo prioritetnega seznama. Za iskanje najbližje tarče se sprehodimo po seznamu vseh entitet in iščemo najbližjo. Če je ta entiteta označena kot prioritetna jo shranimo v prvi element polja, sicer pa v drugi element polja. Na koncu vrnemo prvi element polja, če ni prazen, sicer pa drugega.

3.3 PNG

Za zajemanje zaslonskih slik smo implementirali lastno različico algoritma PNG (`http://www.faqs.org/rfcs/rfc2083.html`). implementirali smo le dele, ki so kritični za shranjevanje slik v formatu PNG, torej glavo in podatkovna odseka IHDR (podatki o sliki) ter IDAT (podatki o pikslih). Za stiskanje podatkov v podatkovnem delu IDAT smo uporabili Android različico algoritma Deflate, saj nam po sprecifikacijah ni uspelo implementirati stiskanja (kombinacija huffmana in LZ77).