Prolećni semestar 2019/2020

Rogue-like igra

CS103
Algoritmi i strukture

Projektna dokumentacija

Student:

Nikola Tasić 3698

Asistent:

Veljko Grković

Sadržaj

Uvod

Razvoj video igara je definitivno jedan od najboljih načina za demonstriranje rešavanja kompleksih problema. Bez obzira koliko naivno deluje, programiranje igrara na nižem nivou od komercijalnih alata zahteva veoma dobru optimizaciju samog koda i biranje najefikansnijih resenja kako ne samo rešiti probleme nego i struktuirati ceo projekat. Tema ovog projekta biće implementacija rogue-like igre u programskom jeziku C. Odabir jezika je takav jer se možda u njemu najbolje ogleda primena algoritama a naročlito struktura podataka a i performanse koje dobijamo ovim jezikom nikad nisu na odmet. Pored činjenice da koristimo C kao izabrani programski jezik navodim takodje da ćemo koristiti SDL2 biblioteku za rad sa grafičkim procesorom, fontovima i teksturama.

Analiza

Osvetljenje

Igra je predstavlja klasičan rouge-like lavirint u kome se lavirint, čudovišta i izlaz generišu nasumično. Prilikom pornalaska izlaza iz lavirinta igrač prelazi u sledeći nivo koji sadrži između ostalog više neprijatelja. Lavirint je tako dizajniran da se kretanje odvija u kvadratima 32x32 piksela. Takodje postoji primitivna implementacija tri razlicite vrste osvetljenja:

- Ceo lavirint je osvetljen podjednako.
- Osvetljeni su samo blokovi i entiteti u neposrednoj bilizini izvora svetlosti. Naravno u zavisnosti od udaljenosti. Izvori svetlosti su igrač i nasumično generisane baklje na zidovima.
- Primitivna implementacija direkcionog osvetljenja primenom *Bresenhamovog* algoritma za iscrtavanje linija u matrici. Impementacija je takva da ako postoji načini da se iscrta prava linija izmedju bloka i izvora svetlosti tako da se ne prelazi preko bloka koji ne propusta svetlost taj blok je osvetljen u zavisnosti od distance.

Lavirint

Generisanje lavirinta počinje sa dinamički alociranim jednodimenzionim nizom. Razlog biranja jednodimenzionog niza u odnosu na primer matricu je brzina nasumicnog pristupa memorijskim blokovima te strukture. Matrica predstavlja niz nizova gde svaki od nizova može da bude alociran na različitim delovima *heap* memorije programa i samim tim pristup susednim blokovima ne mora da traje isto sto nije slucaj kod jednodimenzionog niza. Algoritam za generisanje prolazi kroz neparne kolone matrice (reći ću matrice zbog jasnije slike o tome šta se dešava) i nasumično generiše hodnike lavirinta. Izlaz je postavljen nasumično van prvog kvadranta koji igrač vidi. O kvadrantima kasnije. Tip generisanog bloka i entiteta se čuva kao *enum.* Lavirint se u memoriji čuva kao strukura sa generisanom matricom koja se koristi za logiku i renderovanje, predefinisanom visinom i sirinom, kordiantama izlaza i graf reprezentacijom svih raskrsnica njemu samom. Taj graf se koristi za pronalaženje najkraćeg puta do izlaza pomocu specifične adaptacije ***A**** algoritma.

Entiteti

Ideja za implementaciju entiteta u igri vuče osnove iz objektno-orijentisanog programiranja ciju upotrebu kao stil programiranja vidimo kao veoma isforiranu u savremenom IT svetu, pocevsi od sve više i više OO jezika, preko OO servera sve do objektno orijentisanih embeded i operativnih sistema. Obzirom na to o kakvom problemu se radi OO pristup je verovatno najbolji za razvoj igara ali C kao programski jezik nije objektno orijentisan tako da ćemo morati da emuliramo klase preko struct-ova.

```
typedef struct entity {
    int x;
    int y;
    float hp;
    int next_move;
    union {
        player_t player;
        enemy_t enemy;
        spawner_t spawner;
        light_t light;
        pew_t pew;
    };
    enum entities type;
} entity_t;
```

Kao sto vidimo gore na slici ovo je minimum od onoga sto na treba da definišemo entitet. Šta entitet može da bude, gde se nalazi i da li može da napravi sledeći korak u smislu vremena proteklog u igri. Na osonovu ovakve strukure možemo izvesti jos jednu strukturu koji je član gore navedene anonimne unije tj našeg igrača.

```
typedef struct player {
    enum dir dir;
    int next_shot;
    float dmg;
} player_t;
```

Obzirom na to da je *player* deo unije on takođe ima sve što i svaki entitet i još po nešto. Klasičan primer nasleđivanja u OOP.

Igra

Na slici vidimo igrača koji je uvek pozicioniran na koordinatama (1,1) u igri i delimično ovetljenje delove lavirinta. Kao deo svake igre postoji jedan glavni koncept koji se naziva game loop. Game loop uglavnom sastoji iz tri procedure koje se moraju izvršiti u svakom frejmu (frejm bi bio najmanja nedeljiva jedinica vremena u igri) a to su uzimanje input-a od igrača, izvršavanje logike igrice i prikaz rezultata prethodna dva na ekranu.

```
while (running) {
    Input(&state, &event);
    Event(&state, elapsed_time);
    Update(&state, elapsed_time);
    Render(&state, renderer);
}
```

Naša bazična implementacija bi izgledala ovako. Obraćamo pažnju na to da svaka od metoda prima trenutni *state* igre kao parametar jer se držimo principa funcionalnog programiranja a to je da ne koristimo globalne promenljive i da svaka metoda uvek menja ceo state programa. Ovde postoji jedan dodatak a to je *Event* procedura.

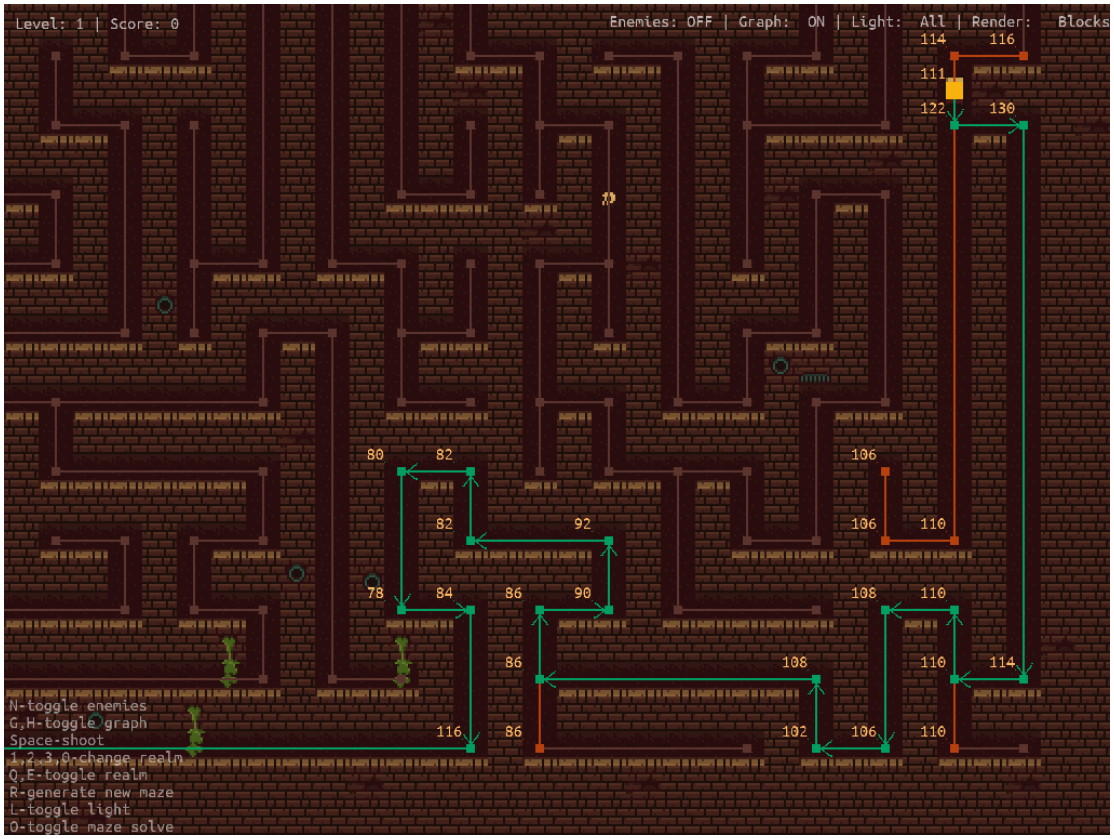
```
void Event(state_t* state, double delta_time) {
    event_t* ev;
    while (!queue_isempty(state->events)) {
        ev = queue_dequeue(state->events);
        ev->callback(state);
        free(ev);
    }
}
```

Event sistem sluzi da umesto da objekti direktno menjaju state igre oni zapravo salju evente u event sistem koji je implementiran kao strukutra red(queue) i akcije koje se aktiviraju prilikom poziva ove procedure zapravo menjaju state. Ova implementacija u ovakvoj single-threaded aplikaciji je redundantna ali može da bude veoma korisna prilikom uvođenja više threadova za izvršavanje logike igre.

Pronalaženje izlaza

Na slici vidimo dva prikaza pronađenog puta od trenutne igračeve lokacija do izlaza iz lavirinta koji se nalazi van trenutnog vidokruga. Put je nađen preko adaptacije A* algoritma za pretragu najkraćeg puta u grafu. Na slici svetlo zelene linije predstavljaju put do izalaza, crvene pretrazeni putevi koji nisu doveli do izlaza a braon linije one koje nisu ni bile pretražene jer nisu deo optimalnog rešenja. A* algoritam obilazi sve dostupne čvorove iz početnog čvora vrši evaulaciju njihove heuristike (koliko su validni kao deo potencijalnog puta) i smešta ih u strukturu prioritetnog reda (priority queue). Iz tog reda uvek izlazi kao prvi čvor koji ima najbolju heuristiku kao prvi potencijalni deo puta do izlaza i tako sve dok se ne pronadje put. Adaptacija heuristike za ovaj problem je bila takva da iako A* koristi udaljenost od izlaza za evaluaciju to mozda nije najbolje rešenje jer u lavirintima ima dosta udaljavanja od samog cilja pre nego sto zapravo dodjemo do njega. Tako da u sustini vise posmatramo svaki čvor zasebno.

```
n->h = (int) euclidean_dist(n->x, n->y, end->x, end->y) * 2;
n->g = manhattan_dist(n->x, n->y, (*node)->x, (*node)->y);
// apparently better results by not factoring in the previous node cost
n->f = /*(*node)->f +*/ n->g + n->h;
if (!n->visited) {
    // leave some breadcrumbs of how we got here
    n->came_from = *node;
    pqueue_enqueue(visited, &n, &n->f);
}
```



Na ovoj slici vidimo kako zapravo izgleda heuristika za svaki od pretraženih čvorova grafa. U evaulaciju heuristike ulaze direktna udaljenost (Euclidian distance) i pravougaona distanca (Mahattan distance).

Između ostalog isti algoritam koriste i čudovišta koja u svakom trenutku aktivno traže igrača, s tim što je njihovor rešenje tj put do cilja pretvoreno u stek (stack) svih koordinata blokova koji moraju da posete da bi došli do njega. Tako da se u svakom sledećem trenutku kada čudovište može da načini korak skida sledeća koordinata sa steka i ono se pomera na tu lokaciju.

Zakljucak

Kao što je već navedeno u uvodu implementacija video igre je veoma plodno tlo na kome niču razne ideje za rešavanje kompleksnih programerskih problema a programiranje u C-u je pružilo malo svežine u inače dosadan svet OO programiranja i glomaznih frejmvorkova. S druge strane A* algoritam iako je jedan od najefikasnijih *pathfinding* algoritama zahtevao je malu adaptaciju da bi funcionisao optimalno u okruženju u kakvom smo se našli.

Bibliografija

Bresenham's Line Algorithm. (n.d.). Wikipedia: https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

Computerphile. (n.d.). *A* (A-Star) Search Algorithm*. YouTube: https://www.youtube.com/watch?v=ySN5Wnu88nE

geeksforgeeks. (n.d.). *Priority Queue in CPP STL*. GeeksForGeeks: https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/

javidx9. (n.d.). *Path Planning - A* (A-Star)*. Youtube: https://www.youtube.com/watch?v=icZj67PTFhc

joewing. (n.d.). *Maze*. GitHub: https://github.com/joewing/maze/blob/master/maze.c

lazyfoo. (n.d.). *SDL*. LazyFoo: https://lazyfoo.net/tutorials/SDL/index.php

Pound, M. (n.d.). *mazesolving*. GitHub: https://github.com/mikepound/mazesolving