PROGETTO PROGRAMMAZIONE

ANNO 2021/2022

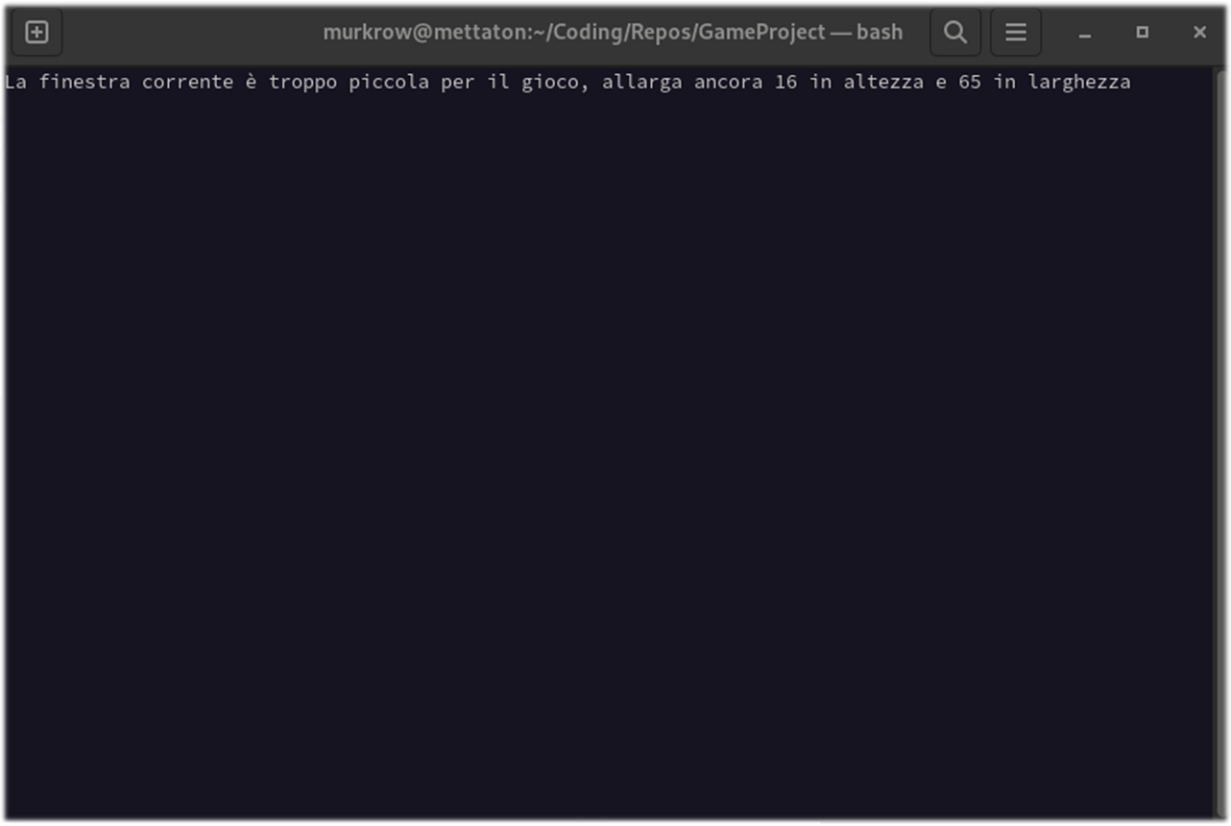
Marco Coppola 0001020433
Gabriele Randi 001021542

Indice

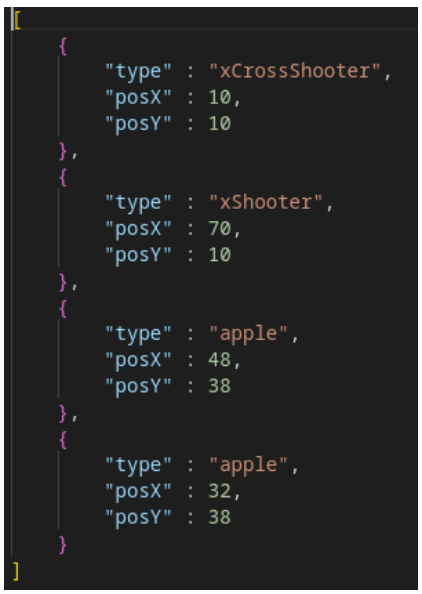
Inizializzazione

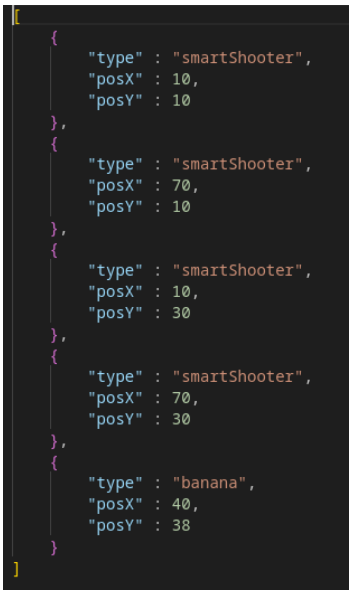
Il primo problema di cui ci siamo accorti durante lo sviluppo del gioco è stato che la dimensione del terminale non è sempre abbastanza grande da permettere alla mappa (e i vari elementi come mini-mappa o statistiche del gioco) di rientrare nei bordi.

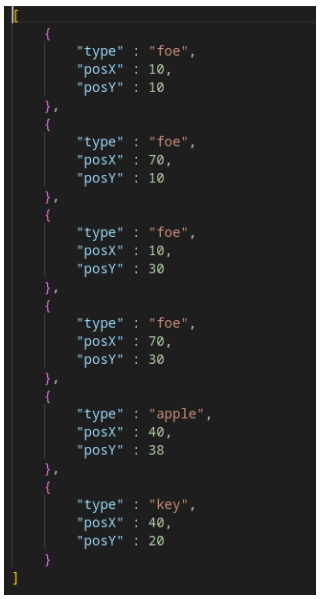
Per questo, una volta fissato le dimensioni della mappa, viene invocata una funzione che permette all'utente di ridimensionare il terminale affinché non raggiunga una dimensione accettabile.



Subito dopo parte la generazione della mappa e delle stanze, ognuna associata ad un file **.json,** nel quale sono descritti gli **artefatti** e le **entità** da creare e in che posizione.







Esempi di alcuni file .json che generano entità ed artefatti

La generazione della mappa

Per evitare che il gioco fosse ripetitivo abbiamo deciso di generare il layout della mappa in modo **casuale**, partendo da una struttura di base composta dalla stanza centrale e le quattro stanze adiacenti. Le successive cinque stanze vengono posizionate in modo casuale, decidendo come prima cosa a quale stanza saranno adiacenti, creando una lista delle possibili posizioni disponibili adiacenti alla stanza scelta e andando poi a scegliere una delle posizioni disponibili.

```
while(nRooms < 10){

    // resets variables
    options = 0;
    possibleCoords.clear();

    // picks random room to build from
    x = (rand()%nRooms) + 2;

    // puts every possible spot's coordinates to create a room in a vector
    for(int i = 0; i < 9; i++){
        for (int t = 0; t < 9; t++){
            if (floor[i][t] == x){
                if (floor[i-1][t] == -1){ //top
                    options++;
                    possibleCoords.push_back(pair<int, int>(i-1, t));
                }

                if (floor[i+1][t] == -1){ //bottom
                    options++;
                    possibleCoords.push_back(pair<int, int>(i+1, t));
                }
                if (floor[i][t+1] == -1){ //right
                    options++;
                    possibleCoords.push_back(pair<int, int>(i, t+1));
                }
                if (floor[i][t-1] == -1){ //left
                    options++;
                    possibleCoords.push_back(pair<int, int>(i, t-1));
                }
            }
        }
    }
    // picks one of the available spot and creates the room
    if (options != 0){
        x = (rand()%options) + 1;
        x--;
        // updates two-dimensional array with new room
        floor[possibleCoords[x].first][possibleCoords[x].second] = nRooms;
        // updates room coordinates with new room
        rooms.push_back(Room(nRooms, "default", possibleCoords[x], width, height, game_objects));
        nRooms++;
        }
    }
```

Il motore di gioco

Dopo aver configurato tutto e dopo aver generato stanze ed entità, il gioco vero e proprio è pronto per partire. Una delle sfide più grandi a livello implementativo è stato decidere come permettere alle varie entità di muoversi ed effettuare azioni tutte allo stesso tempo, per evitare in ogni **frame** di avere entità che si muovono prima delle altre oppure ancora peggio, non permettere al giocatore di evitare un proiettile in arrivo in quanto la sua posizione non è ancora stata aggiornata.

Abbiamo deciso allora di prendere spunto dal funzionamento di molte librerie grafiche che suggeriscono di inserire una **classe astratta** (nel nostro caso GameObject) da cui tutti gli oggetti che hanno bisogno di essere aggiornati ereditano e che consiste di due metodi fondamentali:

- DoFrame: quando questa funzione viene chiamata, l'entità deve svolgere tutta la logica per quanto riguarda il suo movimento, se ha inflitto/subito danno ecc.
- Draw: effettua il render dell'entità su schermo

```
class GameObject
{

    public:

        virtual void DoFrame() = 0; // called once per frame
        virtual void Draw() = 0; // called once per frame, after DoFrame

        // position
        int x;
        int y;

        // game window size
        int xMax,yMax;

        char displayChar; // the char displayed on screen for this objec
        GameObjectList *gameItems; // easy access to all game objects

};
```

Abbiamo infine arricchito la classe GameObject con delle proprietà condivise da tutti gli oggetti di gioco, ovvero la posizione all'interno della mappa, il carattere identificativo dell'oggetto che comparirà sulla mappa una volta chiamato Draw() e la lista di tutti gli oggetti attualmente nella stanza di gioco per un accesso immediato (ad esempio calcolare la distanza di un nemico dal giocatore, determinare le varie collisioni ecc.)

Lo spostamento attraverso le varie stanze

Per muoversi attraverso le stanze è necessario aver prima sconfitto tutti i nemici presenti nella stanza in cui ci si trova, e questo controllo vieni eseguito utilizzando la funzione numberOfEnemies().

```
int GameObjectNode::numberOfEnemies(GameObjectList *gameObjects){

    Node *current = head;
    int number = 0;
    while (current != NULL){

        std::vector<char> included{CHAR_DUMMY, CHAR_CROSS_SHOOTER, CHAR_SMART_SHOOTER, CHAR_X_CROSS_SHOOTER, CHAR_X_SHOOTER};
        char toFind = current->data->displayChar;
        if (std::find(included.begin(), included.end(), toFind) != included.end())
        {
            number++;
        }

        current = current->next;
    }

    return number;
}
```

Ogni volta che si esce da una stanza, la lista di oggetti locale viene azzerata e resa identica a quella attualmente in aggiornamento, che a sua volta viene poi azzerata per poter leggere la lista di oggetti locale della stanza a cui si sta accedendo.

In questo modo, qualsiasi cambiamento avvenuto all'interno di una stanza verrà mantenuto per tutto il resto della partita.

```
void Room::freeze_room(){

    //Remove previous entities from local list
    roomObjects.reset();

    //Populate room objects with gameobjects from main
    Node* current = gameObjects->head;
    while(current != NULL){

        if(current->data != NULL)
            roomObjects.insert(current->data);
        current = current->next;
    }
    if (roomObjects.numberOfEnemies(gameObjects) == 0 && !cleared){
        gameObjects->roomsToClear--;
        cleared = true;
        if (gameObjects->roomsToClear == 0)
        {
            gameObjects->gameMap->rooms[0].roomObjects.insert(new Ladder(20, 40, CHAR_LADDER, gameObjects));
            message welcome = message("Complimenti!","Hai completato questo piano, ritorna alla stanza centrale per proseguire");
            welcome.wait_close.wait();
        }

    }
}
```

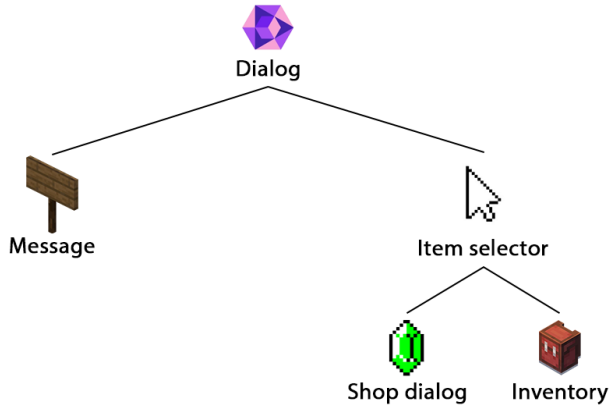
Una volta esplorate tutte le stanze è possibile accedere al livello successivo, che avrà un layout delle stanze differente e un livello di difficoltà progressivamente maggiore.

```
void user_interacted()
{
    message newFloor = message("Attenzione", "Stai per accedere al piano successivo");
    newFloor.wait_close.wait();

    clear();

    gameItems->gameMap->generateMap(gameItems->gameWindow, gameItems);
    gameItems->gameMinimap->drawMinimap(gameItems, 0);
    gameItems->gameMap->createRoom(0);
    gameItems->player->x = 40;
    gameItems->player->y = 20;
    gameItems->roomsToClear = 9;
    gameItems->difficultyLevel = gameItems->difficultyLevel*1.5;

    gameItems->gameStats->update_stats();
    mvwaddch(gameItems->gameWindow, y, x, ' ');

}
```

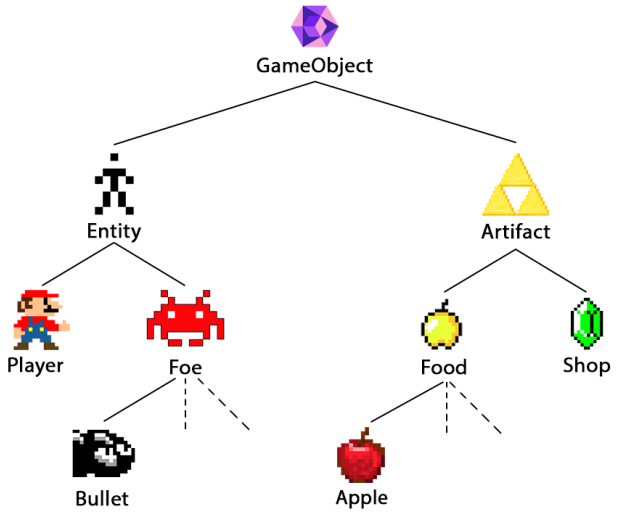
Ereditarietà

Il concetto di **ereditarietà** svolge un ruolo fondamentale all'interno del progetto, applicandosi alla perfezione per la **molteplicità** degli oggetti di gioco che, pur condividendo la maggior parte delle variabili e delle funzioni basilari, si distinguono ognuno per qualche particolarità che viene implementata in modo diverso.

I due ambiti principali in cui è stata applicata l'ereditarietà sono stati i vari **oggetti di gioco** (il giocatore stesso, i vari artefatti e i diversi tipi di nemici) e i **dialoghi** (che includono messaggi, inventario e il negozio di oggetti)



Gerarchia dei dialoghi



Gerarchia delle entità

Gli oggetti di gioco

Sia gli oggetti di ogni stanza che quelli attualmente in aggiornamento (sul campo di gioco) sono gestiti tramite una classe (GameObjectNode) che opera su una struttura personalizzata (Node):

```
struct Node
{
    GameObject *data;
    Node *next;
};

class GameObjectNode{

public:

    Node *head; // head is null by default

    // returns (if present) a reference to entity placed at point (x,y)
    Entity* findEntityAtPos(int x, int y, GameObject *excluded);

    // add a new gameobject to list
    void insert(GameObject *item);

    // removes gameobject from list
    void remove(GameObject *item);

    // clears the list
    void reset();

    // returns the total number of enemies in list
    int numberOfEnemies(GameObjectList *gameObjects);

    GameObjectNode();

};
```

Quando un elemento viene inserito nella lista principale generata nel **main**, l'oggetto viene inserito nel campo di gioco e ad ogni frame viene prima invocata la sua funzione DoFrame() e in seguito Draw(). Non appena un oggetto viene rimosso dalla lista principale (ad esempio se il giocatore si muove in un'altra stanza o se un'entità muore e viene chiamata la sua funzione Destroy()) i suoi metodi DoFrame() e Draw() non saranno più chiamati e pertanto non comparirà nella schermata di gioco.