Object-Oriented Programming Project Report

20202865 윤수용

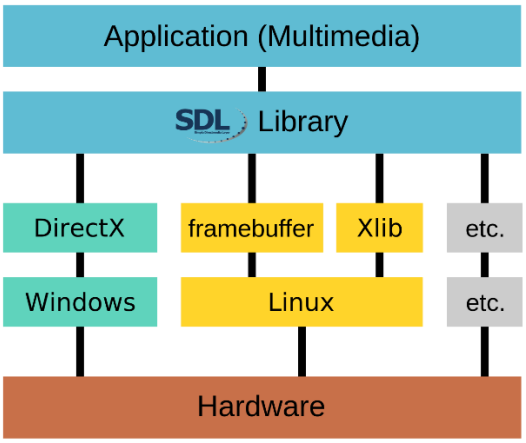
1. Project Introduction

A. Game Introduction

'Bong's Dungeon' is a roguelike game in its most basic form, based on the 'rogue'. The player character is represented by symbol '@' and the objective is to manipulate it and battle to gain experience (exp) to strengthen the character, collect items, defeat enemies, and descend as deep into the dungeon as possible. The game is played on a turn-by-turn basis based on the player's actions. Characters, including the player, have health (hp), damage (damage), defense (defense), and evasion (evasion), and if the player runs out of health, the game will be over, so the player must act appropriately depending on the situation.

B. Development Basis

'Bong's Dungeon' is based on the external library libtcod. libtcod is based on the Simple DirectMedia Layer (SDL), a cross-platform multimedia library that specializes in creating roguelike games. SDL abstracts the layers of video, audio, user input, etc. to make it possible to run on multiple operating systems. And libtcod outputs graphics to the screen via SDL.



This project is based on 'Complete roguelike tutorial using C++ and libtcod'[1]. The project is quite old, so while it works, there are many inconveniences. So, the goal of this project is to maintain the overall framework of the program and implemented the same functionality, but aimed to improve the internal structure and visibility of

[1] https://www.roguebasin.com/index.php/Complete_roguelike_tutorial_using_C%2B%2B_and_libtcod_-_part_1:_setting_up

the code, and to apply the project more oriented to the concept of object orientation.

C. Game Construct

When you run the game for the first time, the welcome screen displays 'New game' to start the game and 'Exit' to end the game. When executing a new game, the interface and map are displayed on the screen, and you can use the directional keys and 'G', 'I', and '/' to operate the game as shown below.

i. Movement

Move around the map by accepting up, down, left, and right keystrokes.

ii. Combat

Combat is performed by making vertical or horizontal contact with an enemy symbol (o, C, T, B) and pressing a key in that direction. Upon contact, the player and the enemy take damage equal to their respective Attack values, which can be reduced by their Defense values or avoided proportionally to their Evasion values. Defeating enemies in combat earns experience points, which, after accumulating a certain amount of experience points, increases your level and allows you to choose one of the stats to strengthen your player. If a player's health reaches zero during a battle, he/she will die and the player have to restart the game.

iii. Collect Items

If you go over an item symbol (!, #, ?) that exists in the field, a log will be displayed at the bottom saying that a specific item exists at this location, and you can get the item by typing 'G'.

iv. View Items

You can check your inventory by typing 'I' to see the list of items you have acquired. You can get up to 27 items, which is the number of letters in the alphabet, and they are sorted in order of acquisition. Acquired items can be used by entering the corresponding alphabet key in the inventory, and the inventory is closed by entering a key other than the alphabet.

v. Use Items

Items are randomly generated throughout the map and can be acquired.

Healing items (heal potion) restore the player's health by a certain amount, and tracking items (snap shot, confusion shot, wide shot) search for the nearest enemy within range and deal a certain amount of damage or grant an effect upon use.

vi. Movement Between Floors

As you explore the map and find the '>' symbol in a field, you can press the '/' key to move to the next level. Descending the floor will randomize the entire map and reset the placement of enemies and items. Enemies will also increase their stats in response to the floor level.

vii. Save & Run

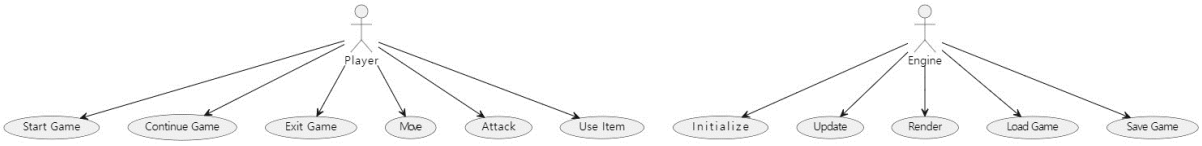
The game automatically saves your current progress. This creates a save file, 'save.sav', which, if present, will display 'Continue' on the welcome screen and allow you to execute the saved progress. This save.sav is automatically deleted when the player gets killed in the game.

viii. Exit

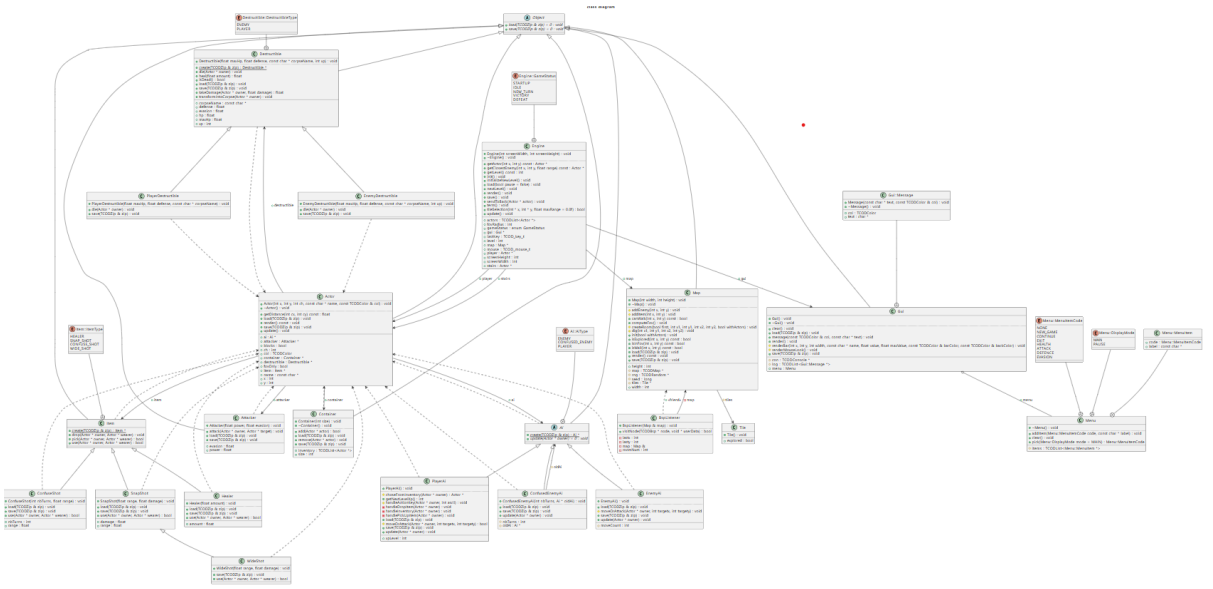
Selecting 'Exit' will save the current progress and exit the game.

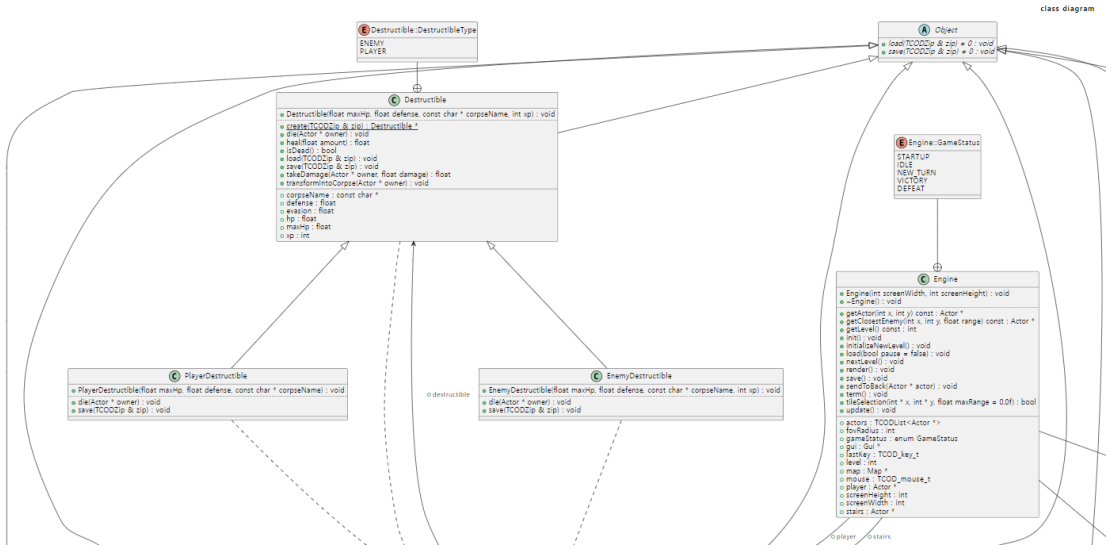
2. Object-Orient Implementation

A. Game Design Properties (Use Case Diagram)



B. Class Design Properties (Class Diagram)





i. Object

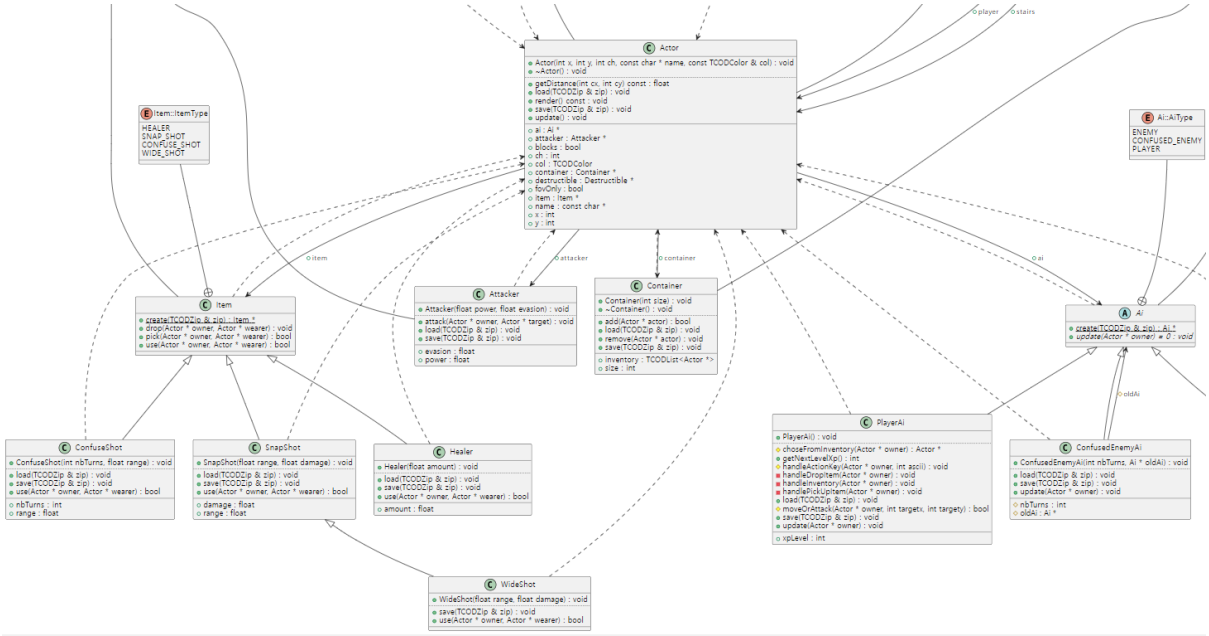
1. Anything that can be stored and executed
2. The basis of all classes

ii. Engine

1. The underlying system that powers the game

iii. Destructible

1. Destructible objects
2. class PlayerDestructible: Player
3. class EnemyDestructible: Enemies



iv. Actor

1. Sets the state of each object

v. Item

1. How in-game items operates
2. class Healer: Healing items
3. class SnapShot: Attack items
4. class ConfuseShot: Effect items
5. class WideShot: AoE items

vi. Attacker

1. System that allows to perform combat

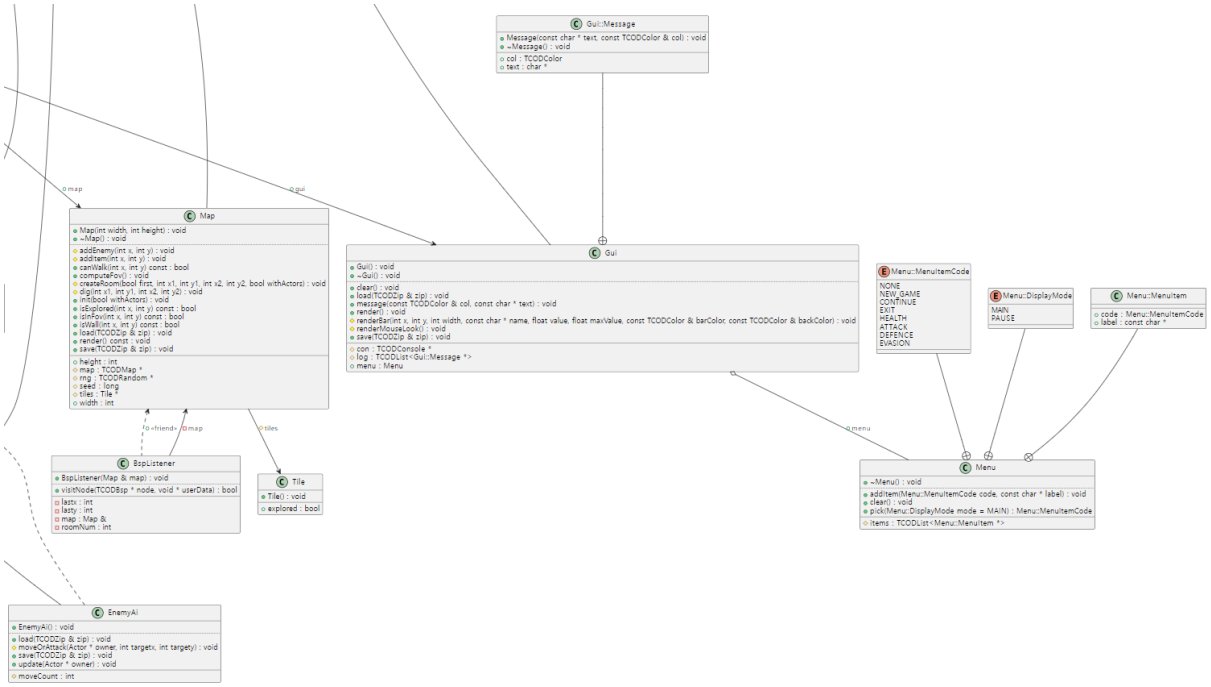
vii. Container

1. System that allows you to store acquired items

viii. Ai

1. How each Destructible operates
2. class PlayerAi: Player
3. class EnemyAi: Enemy

4. class ConfusedEnemyAI: Confused Enemy



ix. Map

1. Creates In-Game Map
2. class Tile: Game Map structure

x. Gui

1. Components of a screen
2. class Menu: Menu screen

C. Object-Orient Feature Implementation

i. Encapsulation

Encapsulation is achieved by separating headers and .cpp files, hiding data and behavior inside objects. For example, Actor encapsulates the location, appearance, and behavior of characters in the game; Destructible encapsulates the health, defense, and death process to represent something that can be destroyed; and Item represents an in-game item and encapsulates the type and behavior of the item.

ii. Inheritance

Inheritance is achieved by creating a hierarchy of objects, using the most basic

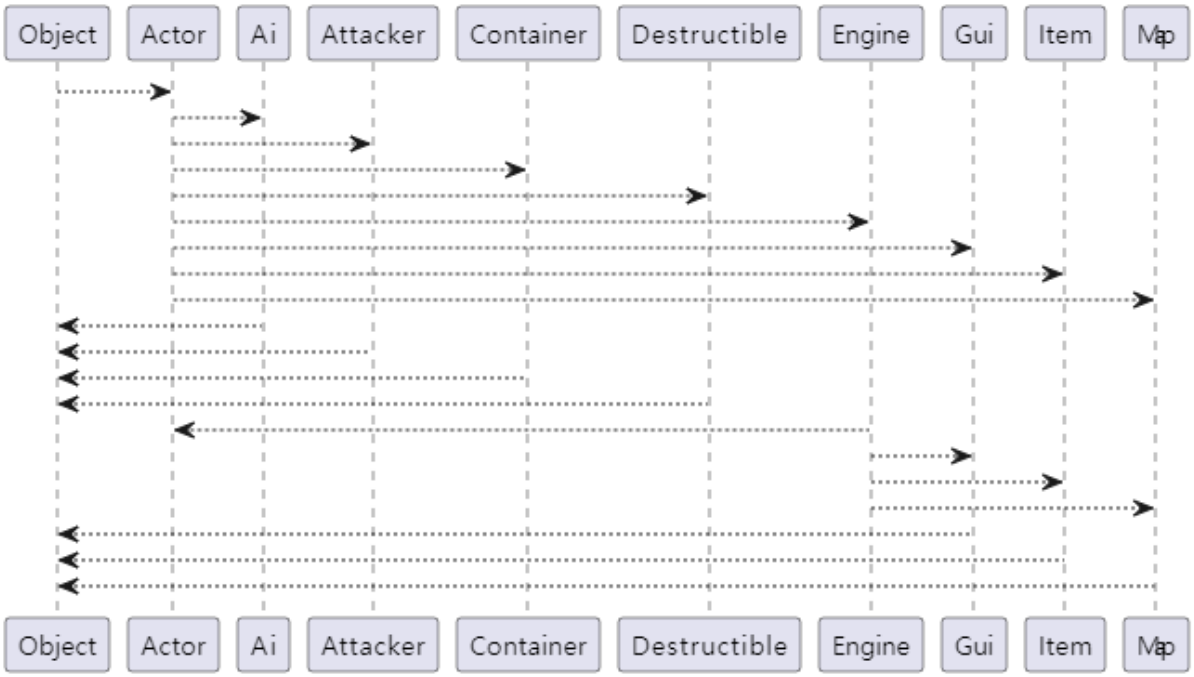
class and having other objects inherit from them. For example, most classes, such as Actor, Destructible, Attacker, Item, and Map, inherit from Object, the class that provides storage and loading functionality.

iii. Polymorphism

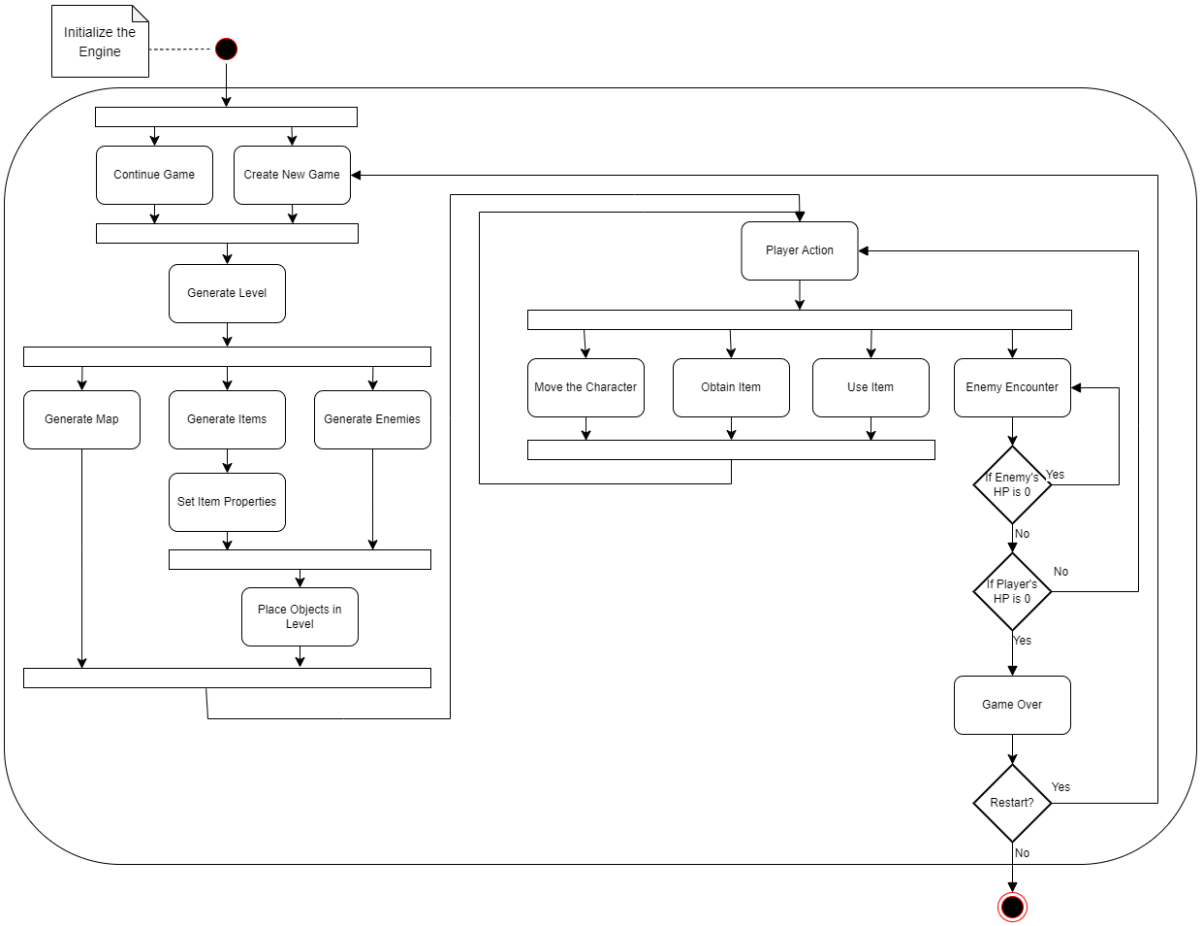
Polymorphism is achieved by allowing objects of different classes to respond to the same method call in different ways. For example, the update() method of an Ai adjusts the Ai state of each object, but it depends on which Ai player is dealing with. For an EnemyAi, update() will cause the enemy to move towards the player and attack, but for a ConfusedEnemyAi, it will cause the enemy to move in a random direction.

3. Program Implementation

A. Function Implementation Properties(Sequence Diagram)



B. Function Implementation Properties(Activity Diagram)

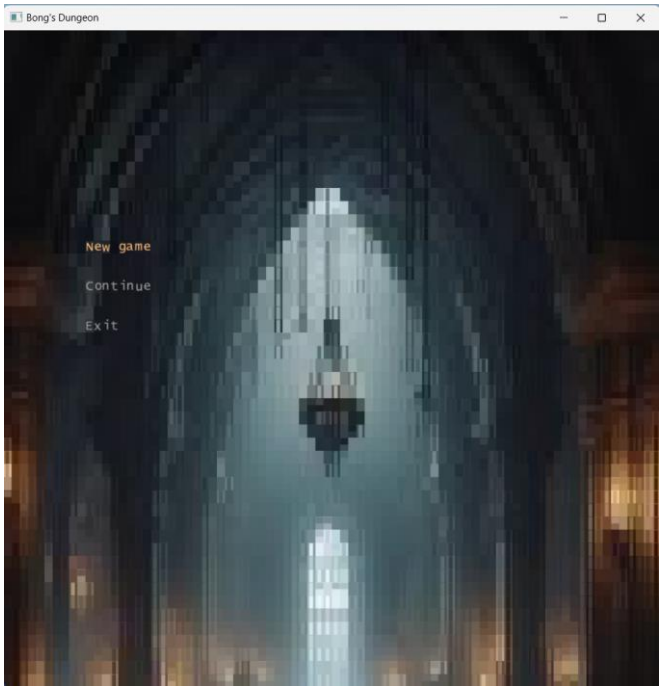


4. Project Result

A. Game Execution/Description

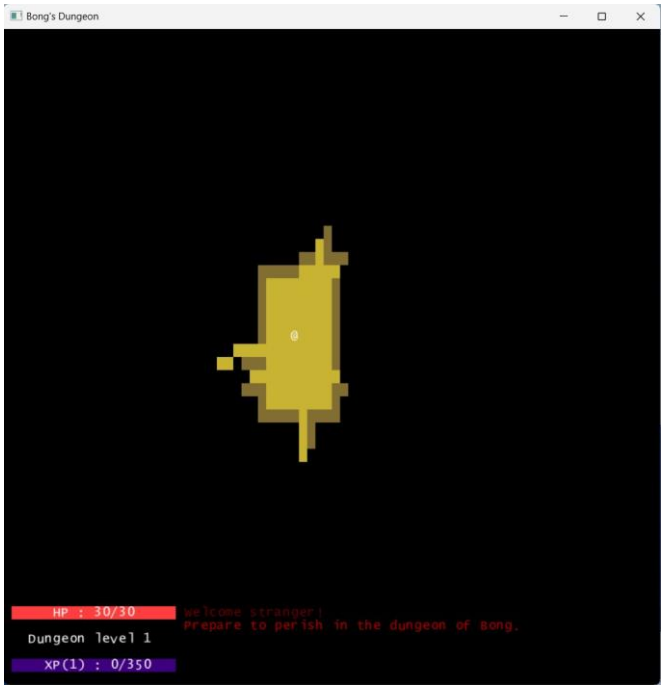
i. Display 1. Main menu

The first time you run the game, you will see a main menu screen. You can select New game to start a new game, Continue to continue if you have any progress (save file), or Exit to end the game.



ii. Display 2. Game Initialization

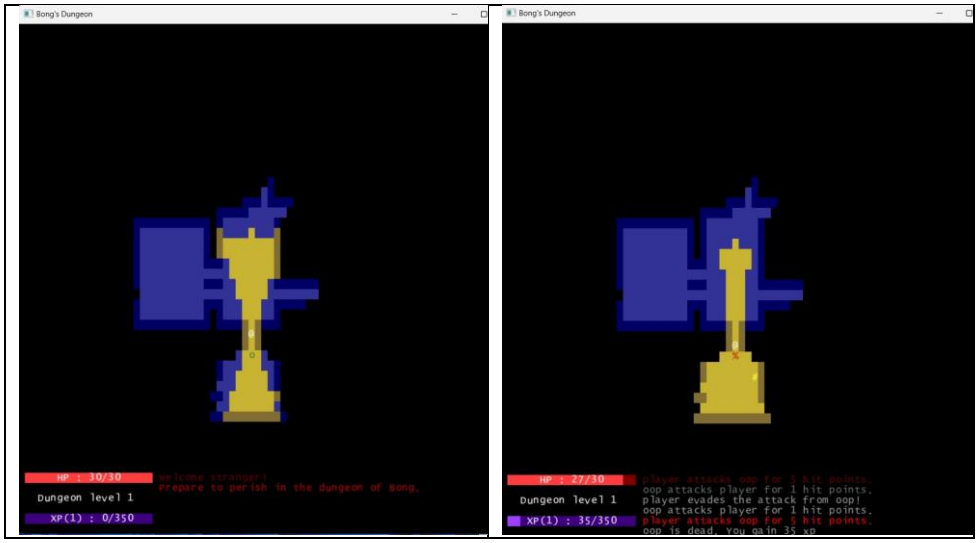
The bottom left corner of the screen displays the player's current health and required experience, while the bottom center displays a simple log of actions, interactions with objects, and the current situation.



iii. Display 3. 전투

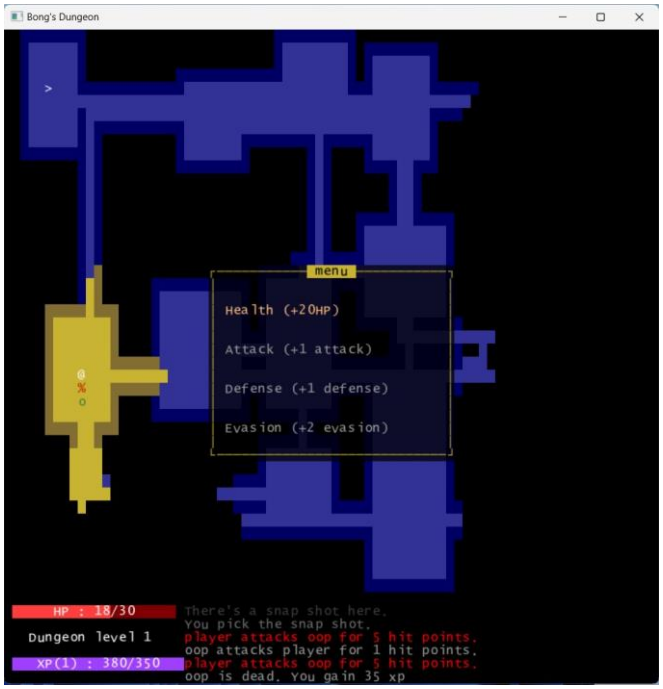
Enemies will track you as you enter their line of sight. When you and an enemy collide, you each take (your opponent's attack value - your defense

value) damage, and you gain experience for defeating them. When an enemy or player is killed, a log is printed at the bottom and a '%' symbol is left on the map.



iv. Display 4. Level Up

Gaining a certain amount of experience increases your level, and you can choose to enhance one of your player's stats. Health increases your health by 20, Attack increases your attack by 1, Defense increases your defense by 1, and Evasion increases your evasion by 2%.



v. Display 5. Collect Item

When the player moves over the tile where the item is located, a log is printed

at the bottom indicating that the item is located here. If the inventory is not full at this point, the item can be picked up and is stored in the inventory in the order it was picked up.

vi. Display 6. Use Item

Obtained items are available in your inventory.

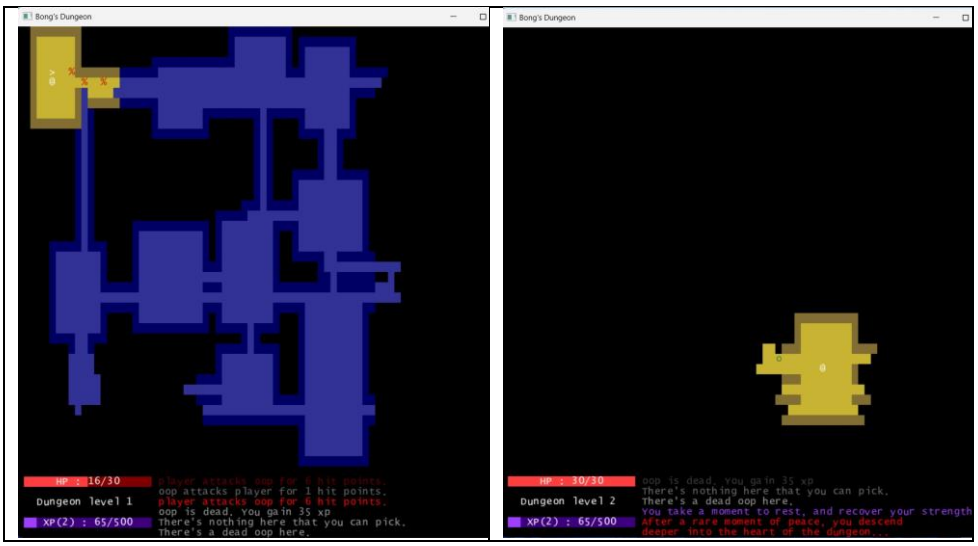
An example of a healing item is heal potion, which restores a certain amount of health and prints "You feel refreshed." to the log.

As an example of a tracking item, snap shot deals a certain amount of damage to the enemy closest to the player, and prints "The player fires a sudden shot to the %s! The damage is %g hit points." in the log.

vii. Display 7. Level Descension

When the player steps on a '<' symbol on the map and presses the '/' button, they move to the next floor. The new floor is completely reset, and traveling through it will result in a log being printed at the bottom and half of your

maximum health being restored.



viii. Display 8. Menu

Pressing the ESC button stops the game and displays a menu window. At this point, you can choose to start a new game, continue the game, or exit.

B. Demo Video

5. Appendix

A. How to Compile

i. This project is created through Code::Blocks.

First download the Nightly build CB_20230604_rev13311_win64-setup-MinGW.exe from the URL below and install it. This is because MinGW-w64 which is included in original Code::Blocks has problem supporting the C++17.

https://sourceforge.net/projects/codeblocks/files/Binaries/Nightlies/2023/

ii. Go to Windows Settings -> System -> Info -> Advanced System Settings -> Environment Variables, select Path, press Edit, and add MinGW₩bin where CodeBlocks is installed.

iii. Download SDL2-devel-2.28.5-mingw.zip from the URL below. After decompressing, copy the bin, include and lib folders under x86_64-w64-mingw32 to your working project folder.

https://github.com/libsdl-org/SDL/releases/tag/release-2.28.5

iv. Download libtcod-1.24.0-x86_64-mingw.tar.gz from the URL below. After decompressing, copy the include and data folders to your working project folder. Copy libtcod.dll to the project folder and libtcod.a to the project folder₩lib.

https://github.com/libtcod/libtcod/releases

v. Copy the following files into the Project folder and modify it as below.

Rogue.cbp

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<CodeBlocks_project_file>
	<FileVersion major="1" minor="6" />
	<Project>
		<Option title="rogue" />
		<Option pch_mode="2" />
		<Option compiler="gcc" />
		<Build>
			<Target title="Debug">
				<Option	output="rogue"	prefix_auto="1"
extension_auto="1" />
				<Option object_output="obj/" />
				<Option type="1" />
				<Option compiler="gcc" />
				<Compiler>
					<Add option="-g" />
				</Compiler>
			</Target>
		</Build>
		<Compiler>
			<Add option="-Wall" />
			<Add directory="include" />
			<Add directory="include/SDL2" />
		</Compiler>
		<Linker>
```

```
                        <Add option="-ltcod -lmingw32 -lSDL2main -lSDL2" />
                        <Add directory="lib" />
                </Linker>
                <Unit filename="rogue.cpp" />
                <Extensions>
                        <lib_finder disable_auto="1" />
                </Extensions>
        </Project>
</CodeBlocks_project_file>
```

B. Source Code

Actor.h

```
#ifndef _ACTOR_H_
#define _ACTOR_H_

class Actor : public Object {
public :
    int x, y; // position on map
    int ch; // ascii code
    TCODColor col; // color
        const char *name; // the actor's name
        bool blocks; // can we walk on this actor?
        bool fovOnly; // only display when in fov
        Attacker *attacker; // something that deals damages
        Destructible *destructible; // something that can be damaged
        Ai *ai; // something self-updating
        Item *item; // something that can be picked and used
        Container *container; // something that can contain actors

        Actor(int x, int y, int ch, const char *name, const TCODColor &col);
        ~Actor();
        void update();
    void render() const;
    float getDistance(int cx, int cy) const;
        void load(TCODZip &zip);
        void save(TCODZip &zip);
};
#endif // _ACTOR_H_
```

Ai.h

```
#ifndef _AI_H_
#define _AI_H_

class Ai : public Object {
public :
        virtual void update(Actor *owner) = 0;
        static Ai *create(TCODZip &zip);
protected :
        enum AiType {
                ENEMY, CONFUSED_ENEMY, PLAYER
        };
};

class EnemyAi : public Ai {
```

```
public :
	EnemyAi();
	void update(Actor *owner);
	void load(TCODZip &zip);
	void save(TCODZip &zip);
protected :
	int moveCount;

	void moveOrAttack(Actor *owner, int targetx, int targety);
};

class ConfusedEnemyAi : public Ai {
public :
	ConfusedEnemyAi(int nbTurns, Ai *oldAi);
	void update(Actor *owner);
	void load(TCODZip &zip);
	void save(TCODZip &zip);
protected :
	int nbTurns;
	Ai *oldAi;
};

class PlayerAi : public Ai {
public :
	int xpLevel;
	PlayerAi();
	int getNextLevelXp();
	void update(Actor *owner);
	void load(TCODZip &zip);
	void save(TCODZip &zip);
protected :
	bool moveOrAttack(Actor *owner, int targetx, int targety);
	void handleActionKey(Actor *owner, int ascii);
	Actor *choseFromInventory(Actor *owner);
private:
    void handleDropItem(Actor *owner);
    void handlePickUpItem(Actor *owner);
    void handleInventory(Actor *owner);
};
#endif // _AI_H_
```

Attacker.h

```
#ifndef _ATTACKER_H_
#define _ATTACKER_H_

class Attacker : public Object {
public :
	float power; // hit points given
	float evasion; // evasion points given

	Attacker(float power, float evasion);
	void attack(Actor *owner, Actor *target);
	void load(TCODZip &zip);
	void save(TCODZip &zip);
};
```

```
#endif // _ATTACKER_H_
```

Container.h

```
#ifndef _CONTAINER_H_
#define _CONTAINER_H_

class Container : public Object {
public :
		int size; // maximum number of actors. 0=unlimited
		TCODList<Actor *> inventory;

		Container(int size);
		~Container();
		bool add(Actor *actor);
		void remove(Actor *actor);
		void load(TCODZip &zip);
		void save(TCODZip &zip);
};
#endif // _CONTAINER_H_
```

Destructible.h

```
#ifndef _DESTRUCTIBLE_H_
#define _DESTRUCTIBLE_H_

class Destructible : public Object {
public :
		float maxHp; // maximum health points
		float hp; // current health points
		float defense; // hit points deflected
		float evasion; // evasion chance points
		const char *corpseName; // the actor's name once dead/destroyed
		int xp; // XP gained when killing this monster (or player xp)

		Destructible(float maxHp, float defense, const char
*corpseName, int xp);
		inline bool isDead() { return hp <= 0; }
		float takeDamage(Actor *owner, float damage);
		float heal(float amount);
		virtual void die(Actor *owner);
		void transformIntoCorpse(Actor *owner);
		void load(TCODZip &zip);
		void save(TCODZip &zip);
		static Destructible *create(TCODZip &zip);
protected :
		enum DestructibleType {
			ENEMY, PLAYER
		};
};

class EnemyDestructible : public Destructible {
public :
		EnemyDestructible(float maxHp, float defense, const char
*corpseName, int xp);
		void die(Actor *owner);
		void save(TCODZip &zip);
```

```
};

class PlayerDestructible : public Destructible {
public :
                PlayerDestructible(float maxHp, float defense, const char
*corpseName);
                void die(Actor *owner);
                void save(TCODZip &zip);
};
#endif // _DESTRUCTIBLE_H_
```

Engine.h

```
#ifndef _ENGINE_H_
#define _ENGINE_H_

class Engine {
public :
        enum GameStatus {
                STARTUP,
                IDLE,
                NEW_TURN,
                VICTORY,
                DEFEAT
        } gameStatus;
        TCOD_key_t lastKey;
        TCOD_mouse_t mouse;
    TCODList<Actor *> actors;
    Actor *player;
    Actor *stairs;
    Map *map;
    int fovRadius;
    int screenWidth;
    int screenHeight;
    Gui *gui;
    int level;

    Engine(int screenWidth, int screenHeight);
    ~Engine();
    int getLevel() const;
    void update();
    void render();
    void sendToBack(Actor *actor);
    Actor *getActor(int x, int y) const;
    Actor *getClosestEnemy(int x, int y, float range) const;
    bool tileSelection(int *x, int *y, float maxRange = 0.0f);
    void nextLevel();
    void initializeNewLevel();
        void load(bool pause=false);
        void save();
    void init();
    void term();
};

extern Engine engine;
#endif // _ENGINE_H_
```

Gui.h

```
#ifndef _GUI_H_
#define _GUI_H_

class Menu {
public :
	enum MenuItemCode {
		NONE,
		NEW_GAME,
		CONTINUE,
		EXIT,
		HEALTH,
		ATTACK,
		DEFENCE,
		EVASION
	};
	enum DisplayMode {
		MAIN,
		PAUSE
	};
	~Menu();
	void clear();
	void addItem(MenuItemCode code, const char *label);
	MenuItemCode pick(DisplayMode mode = MAIN);
protected :
	struct MenuItem {
		MenuItemCode code;
		const char *label;
	};
	TCODList<MenuItem *> items;
};

class Gui : public Object {
public :
	Menu menu;

	Gui();
	~Gui();
	void render();
	void message(const TCODColor &col, const char *text, ...);
	void load(TCODZip &zip);
	void save(TCODZip &zip);
	void clear();
protected :
	TCODConsole *con;
	struct Message {
		char *text;
		TCODColor col;
		Message(const char *text, const TCODColor &col);
		~Message();
	};
	TCODList<Message *> log;

	void renderBar(int x, int y, int width, const char *name,
```

```
                float value, float maxValue, const TCODColor &barColor,
                const TCODColor &backColor);
        void renderMouseLook();
};
#endif // _GUI_H_
```

Item.h

```
#ifndef _ITEM_H_
#define _ITEM_H_

class Item : public Object {
public :
        bool pick(Actor *owner, Actor *wearer);
        void drop(Actor *owner, Actor *wearer);
        virtual bool use(Actor *owner, Actor *wearer);
        static Item *create (TCODZip &zip);
protected :
        enum ItemType {
                HEALER,SNAP_SHOT,CONFUSE_SHOT,WIDE_SHOT
        };
};

class Healer : public Item {
public :
        float amount; // how many hp

        Healer(float amount);
        bool use(Actor *owner, Actor *wearer);
        void load(TCODZip &zip);
        void save(TCODZip &zip);
};

class SnapShot : public Item {
public :
        float range, damage;
        SnapShot(float range, float damage);
        bool use(Actor *owner, Actor *wearer);
        void load(TCODZip &zip);
        void save(TCODZip &zip);
};

class ConfuseShot : public Item {
public :
        int nbTurns;
        float range;
        ConfuseShot(int nbTurns, float range);
        bool use(Actor *owner, Actor *wearer);
        void load(TCODZip &zip);
        void save(TCODZip &zip);
};

class WideShot : public SnapShot {
public :
        WideShot(float range, float damage);
        bool use(Actor *owner, Actor *wearer);
```

```
        void save(TCODZip &zip);
};

#endif // _ITEM_H_
```

Map.h

```
#ifndef _MAP_H_
#define _MAP_H_

struct Tile {
    bool explored; // has the player already seen this tile ?
    Tile() : explored(false) {}
};

class Map : public Object {
public :
    int width, height;

    Map(int width, int height);
    ~Map();
    bool isWall(int x, int y) const;
        bool isInFov(int x, int y) const;
    bool isExplored(int x, int y) const;
    bool canWalk(int x, int y) const;
    void computeFov();
    void render() const;

        void load(TCODZip &zip);
        void save(TCODZip &zip);
        void init(bool withActors);
protected :
    Tile *tiles;
    TCODMap *map;
    long seed;
    TCODRandom *rng;
    friend class BspListener;

    void dig(int x1, int y1, int x2, int y2);
    void createRoom(bool first, int x1, int y1, int x2, int y2, bool
withActors);
    void addEnemy(int x, int y);
    void addItem(int x, int y);
};
#endif // _MAP_H_
```

Object.h

```
#ifndef _OBJECT_H_
#define _OBJECT_H_

class Object {
public :
        virtual void load(TCODZip &zip) = 0;
        virtual void save(TCODZip &zip) = 0;
};
#endif // _OBJECT_H_
```

main.h

```
#ifndef _MAIN_H_
#define _MAIN_H_

#include "libtcod.h"
class Actor;
#include "Object.h"
#include "Destructible.h"
#include "Attacker.h"
#include "Ai.h"
#include "Item.h"
#include "Container.h"
#include "Actor.h"
#include "Map.h"
#include "Gui.h"
#include "Engine.h"
#endif // _MAIN_H_
```

Actor.cpp

```
#include <math.h>
#include "main.h"

Actor::Actor(int x, int y, int ch, const char *name,
    const TCODColor &col) :
    x(x), y(y), ch(ch), col(col), name(name),
    blocks(true), fovOnly(true), attacker(NULL), destructible(NULL), ai(NULL),
    item(NULL), container(NULL) {
}

Actor::~Actor() {
        if (attacker) delete attacker;
        if (destructible) delete destructible;
        if (ai) delete ai;
        if (item) delete item;
        if (container) delete container;
}

void Actor::load(TCODZip &zip) {
        x = zip.getInt();
        y = zip.getInt();
        ch = zip.getInt();
        col = zip.getColor();
        name = strdup(zip.getString());
        blocks = zip.getInt();
        fovOnly = zip.getInt();
        bool hasAttacker = zip.getInt();
        bool hasDestructible = zip.getInt();
        bool hasAi = zip.getInt();
        bool hasItem = zip.getInt();
        bool hasContainer = zip.getInt();
        if (hasAttacker) {
                attacker = new Attacker(0.0f, 0.0f);
                attacker->load(zip);
        }
```

```
        if (hasDestructible) {
                destructible = Destructible::create(zip);
        }
        if (hasAi) {
                ai = Ai::create(zip);
        }
        if (hasItem) {
                item = Item::create(zip);
        }
        if (hasContainer) {
                container = new Container(0);
                container->load(zip);
        }
}

void Actor::save(TCODZip &zip) {
        zip.putInt(x);
        zip.putInt(y);
        zip.putInt(ch);
        zip.putColor(&col);
        zip.putString(name);
        zip.putInt(blocks);
        zip.putInt(fovOnly);
        zip.putInt(attacker != NULL);
        zip.putInt(destructible != NULL);
        zip.putInt(ai != NULL);
        zip.putInt(item != NULL);
        zip.putInt(container != NULL);
        if (attacker) attacker->save(zip);
        if (destructible) destructible->save(zip);
        if (ai) ai->save(zip);
        if (item) item->save(zip);
        if (container) container->save(zip);
}

void Actor::render() const {
    TCODConsole::root->setChar(x, y, ch);
    TCODConsole::root->setCharForeground(x, y, col);
}

void Actor::update() {
        if (ai) ai->update(this);
}

float Actor::getDistance(int cx, int cy) const {
        int dx = x - cx;
        int dy = y - cy;
        return sqrtf(dx*dx + dy*dy);
}
```

Ai.cpp

```
#include <stdio.h>
#include <math.h>
#include "main.h"
```

```
Ai *Ai::create(TCODZip &zip) {
	AiType type = (AiType)zip.getInt();
	Ai *ai = NULL;
	switch(type) {
		case PLAYER:
		    ai = new PlayerAi();
		    break;
		case ENEMY:
		    ai = new EnemyAi();
		    break;
		case CONFUSED_ENEMY:
		    ai = new ConfusedEnemyAi(0, NULL);
		    break;
	}
	ai->load(zip);
	return ai;
}

// how many turns the monster chases the player
// after losing his sight
static const int TRACKING_TURNS = 3;

EnemyAi::EnemyAi() : moveCount(0) {}

void EnemyAi::load(TCODZip &zip) {
	moveCount = zip.getInt();
}

void EnemyAi::save(TCODZip &zip) {
	zip.putInt(ENEMY);
	zip.putInt(moveCount);
}

void EnemyAi::update(Actor *owner) {
    if (owner->destructible && owner->destructible->isDead()) {
        return;
    }

        if (engine.map->isInFov(owner->x, owner->y)) {
        // we can see the player. move towards him
        moveCount = TRACKING_TURNS;
    } else {
        moveCount--;
    }

        if (moveCount > 0) {
		moveOrAttack(owner, engine.player->x, engine.player->y);
        }
}

void EnemyAi::moveOrAttack(Actor *owner, int targetx, int targety) {
	int dx = targetx - owner->x;
	int dy = targety - owner->y;
	int stepdx = (dx > 0 ? 1 : -1);
```

```
	int stepdy = (dy > 0 ? 1 : -1);
	float distance=sqrtf(dx*dx + dy*dy);
	if (distance >= 2) {
		dx = (int)(round(dx / distance));
		dy = (int)(round(dy / distance));
		if (engine.map->canWalk(owner->x+dx, owner->y+dy)) {
			owner->x += dx;
			owner->y += dy;
		} else if (engine.map->canWalk(owner->x+stepdx, owner->y)) {
			owner->x += stepdx;
		} else if (engine.map->canWalk(owner->x, owner->y+stepdy)) {
			owner->y += stepdy;
		}
	} else if (owner->attacker) {
		owner->attacker->attack(owner, engine.player);
	}
}

ConfusedEnemyAi::ConfusedEnemyAi(int nbTurns, Ai *oldAi) : nbTurns(nbTurns),
oldAi(oldAi) {}

void ConfusedEnemyAi::load(TCODZip &zip) {
	nbTurns = zip.getInt();
	oldAi = Ai::create(zip);
}

void ConfusedEnemyAi::save(TCODZip &zip) {
	zip.putInt(CONFUSED_ENEMY);
	zip.putInt(nbTurns);
	oldAi->save(zip);
}

void ConfusedEnemyAi::update(Actor *owner) {
	if (!owner->destructible || ! owner->destructible->isDead()) {
		TCODRandom *rng = TCODRandom::getInstance();
		int dx = rng->getInt(-1, 1);
		int dy = rng->getInt(-1, 1);
		if (dx != 0 || dy != 0) {
			int destx = owner->x+dx;
			int desty = owner->y+dy;
			if (engine.map->canWalk(destx, desty)) {
				owner->x = destx;
				owner->y = desty;
			} else {
				Actor *actor = engine.getActor(destx, desty);
				if (actor) {
					owner->attacker->attack(owner, actor);
				}
			}
		}
	}

	nbTurns--;
```

```
	if (nbTurns == 0) {
		owner->ai = oldAi;
		delete this;
	}
}

PlayerAi::PlayerAi() : xpLevel(1) {}

void PlayerAi::load(TCODZip &zip) {
	xpLevel = zip.getInt();
}

void PlayerAi::save(TCODZip &zip) {
	zip.putInt(PLAYER);
	zip.putInt(xpLevel);
}

const int LEVEL_UP_BASE = 200;
const int LEVEL_UP_FACTOR = 150;


int PlayerAi::getNextLevelXp() {
	return LEVEL_UP_BASE + xpLevel * LEVEL_UP_FACTOR;
}

void PlayerAi::update(Actor *owner) {
	int levelUpXp = getNextLevelXp();
	if (owner->destructible->xp >= levelUpXp) {
		xpLevel++;
		owner->destructible->xp -= levelUpXp;
		engine.gui->message(TCODColor::yellow, "Your battle skills grow
stronger! You reached level %d", xpLevel);
		engine.gui->menu.clear();
		engine.gui->menu.addItem(Menu::HEALTH, "Health (+20HP)");
		engine.gui->menu.addItem(Menu::ATTACK, "Attack (+1 attack)");
		engine.gui->menu.addItem(Menu::DEFENCE, "Defense (+1 defense)");
		engine.gui->menu.addItem(Menu::EVASION, "Evasion (+2 evasion)");
		Menu::MenuItemCode menuItem = engine.gui->menu.pick(Menu::PAUSE);
		switch (menuItem) {
			case Menu::HEALTH :
				owner->destructible->maxHp += 20;
				owner->destructible->hp += 20;
				break;
			case Menu::ATTACK :
				owner->attacker->power += 1;
				break;
			case Menu::DEFENCE :
				owner->destructible->defense += 1;
				break;
		    case Menu::EVASION :
				owner->destructible->evasion += 2;
				break;
			default:
			    break;
```

```
                }
        }
    if (owner->destructible && owner->destructible->isDead()) {
        return;
    }
        int dx = 0,dy = 0;
        switch(engine.lastKey.vk) {
                case TCODK_UP:
        case TCODK_KP8:
            dy = -1;
            break;
                case TCODK_DOWN:
        case TCODK_KP2:
            dy = 1;
            break;
                case TCODK_LEFT:
        case TCODK_KP4:
            dx = -1;
            break;
                case TCODK_RIGHT:
        case TCODK_KP6:
            dx = 1;
            break;
                case TCODK_KP7:
                    dy = dx = -1;
                    break;
                case TCODK_KP9:
                    dy = -1;
                    dx = 1;
                    break;
                case TCODK_KP1:
                    dx = -1;
                    dy = 1;
                    break;
                case TCODK_KP3:
                    dx = dy = 1;
                    break;
                case TCODK_KP5:
                    engine.gameStatus = Engine::NEW_TURN;
                    break;
                case TCODK_CHAR:
                    handleActionKey(owner, engine.lastKey.c);
            break;
        default:
            break;
    }
    if (dx != 0 || dy != 0) {
        engine.gameStatus = Engine::NEW_TURN;
        if (moveOrAttack(owner, owner->x+dx, owner->y+dy)) {
                engine.map->computeFov();
        }
        }
}
```

```
bool PlayerAi::moveOrAttack(Actor *owner, int targetx,int targety) {
	if (engine.map->isWall(targetx,targety)) return false;
	// look for living actors to attack
	for (Actor **iterator=engine.actors.begin();
		iterator != engine.actors.end(); iterator++) {
		Actor *actor=*iterator;
		if (actor->destructible && !actor->destructible->isDead() &&
			actor->x == targetx && actor->y == targety) {
			owner->attacker->attack(owner, actor);
			return false;
		}
	}
	// look for corpses or items
	for (Actor **iterator=engine.actors.begin();
		iterator != engine.actors.end(); iterator++) {
		Actor *actor=*iterator;
		bool corpseOrItem = (actor->destructible && actor->destructible-
>isDead()) || actor->item;
		if (corpseOrItem && actor->x == targetx && actor->y == targety) {
			engine.gui->message(TCODColor::lightGrey, "There's a %s
here.", actor->name);
		}
	}
	owner->x = targetx;
	owner->y = targety;
	return true;
}

void PlayerAi::handleActionKey(Actor *owner, int ascii) {
    switch(ascii) {
        case 'd': // drop item
            handleDropItem(owner);
            break;
        case 'g': // pickup item
            handlePickUpItem(owner);
            break;
        case 'i': // display inventory
            handleInventory(owner);
            break;
        case '/': // handle stairs
            if (engine.stairs->x == owner->x && engine.stairs->y == owner->y) {
                            engine.nextLevel();
                        } else {
                            engine.gui->message(TCODColor::lightGrey, "There
are no stairs here.");
                        }
            break;
        default:
            break;
    }
}

void PlayerAi::handleDropItem(Actor *owner) {
    Actor *actor = choseFromInventory(owner);
```

```
    if (actor) {
        actor->item->drop(actor, owner);
        engine.gameStatus = Engine::NEW_TURN;
    }
}

void PlayerAi::handlePickUpItem(Actor *owner) {
    bool found = false;

    for (Actor **iterator = engine.actors.begin(); iterator !=
engine.actors.end(); iterator++) {
        Actor *actor = *iterator;

        if (actor->item && actor->x == owner->x && actor->y == owner->y) {
            if (actor->item->pick(actor, owner)) {
                found = true;
                engine.gui->message(TCODColor::lightGrey, "You pick the %s.",
                                    actor->name);
                break;
            } else if (!found) {
                found = true;
                engine.gui->message(TCODColor::red, "Your inventory is full.");
            }
        }
    }

    if (!found) {
        engine.gui->message(TCODColor::lightGrey, "There's nothing here that you
can pick.");
    }

    engine.gameStatus = Engine::NEW_TURN;
}

void PlayerAi::handleInventory(Actor *owner) {
    Actor *actor = choseFromInventory(owner);
    if (actor) {
        actor->item->use(actor,owner);
        engine.gameStatus = Engine::NEW_TURN;
    }
}

Actor *PlayerAi::choseFromInventory(Actor *owner) {
        static const int INVENTORY_WIDTH = 50;
        static const int INVENTORY_HEIGHT = 28;
        static TCODConsole con(INVENTORY_WIDTH, INVENTORY_HEIGHT);

        // display the inventory frame
        con.setDefaultForeground(TCODColor(200, 180, 50));
        con.printFrame(0, 0, INVENTORY_WIDTH, INVENTORY_HEIGHT, true,
TCOD_BKGND_DEFAULT, "inventory");

        // display the items with their keyboard shortcut
```

```
        con.setDefaultForeground(TCODColor::white);
        int shortcut = 'a';
        int y = 1;
        for (Actor **it = owner->container->inventory.begin();
                it != owner->container->inventory.end(); it++) {
                Actor *actor = *it;
                con.print(2, y, "(%c) %s", shortcut, actor->name);
                y++;
                shortcut++;
        }

        // blit the inventory console on the root console
        TCODConsole::blit(&con, 0, 0, INVENTORY_WIDTH, INVENTORY_HEIGHT,
                TCODConsole::root, engine.screenWidth/2 - INVENTORY_WIDTH/2,
                engine.screenHeight/2 - INVENTORY_HEIGHT/2);
        TCODConsole::flush();

        // wait for a key press
        TCOD_key_t key;
        TCODSystem::waitForEvent(TCOD_EVENT_KEY_PRESS, &key, NULL, true);
        if (key.vk == TCODK_CHAR) {
                int actorIndex = key.c - 'a';
                if (actorIndex >= 0 && actorIndex < owner->container-
>inventory.size()) {
                        return owner->container->inventory.get(actorIndex);
                }
        }
        return NULL;
}
```

Attacker.cpp

```
#include <stdio.h>
#include <cstdlib>
#include "main.h"

Attacker::Attacker(float power, float evasion) : power(power), evasion(evasion)
{}

void Attacker::attack(Actor *owner, Actor *target) {
        if (target->destructible && ! target->destructible->isDead()) {
        // Generate a random value between 0 and 99 for evasion check
                int evasionRoll = rand() % 100;

                if (evasionRoll < evasion) {
                        // Evade the attack
                        engine.gui->message(TCODColor::lightGrey, "%s evades the
attack from %s!", owner->name, target->name);
                } else {
            if (power - target->destructible->defense > 0) {
                engine.gui->message(owner == engine.player ? TCODColor::red :
TCODColor::lightGrey,
                    "%s attacks %s for %g hit points.", owner->name, target-
>name,
                    power - target->destructible->defense);
            } else {
```

```
                engine.gui->message(TCODColor::lightGrey, "%s attacks %s but it
has no effect!", owner->name, target->name);
            }
            target->destructible->takeDamage(target, power);
                }
        } else {
                engine.gui->message(TCODColor::lightGrey, "%s attacks %s in
vain.", owner->name, target->name);
        }
}

void Attacker::load(TCODZip &zip) {
        power = zip.getFloat();
}

void Attacker::save(TCODZip &zip) {
        zip.putFloat(power);
}
```

Container.h

```
#include "main.h"

Container::Container(int size) : size(size) {}

Container::~Container() {
        inventory.clearAndDelete();
}

void Container::load(TCODZip &zip) {
        size = zip.getInt();
        int nbActors = zip.getInt();
        while (nbActors > 0) {
                Actor *actor = new Actor(0, 0, 0, NULL, TCODColor::white);
                actor->load(zip);
                inventory.push(actor);
                nbActors--;
        }
}

void Container::save(TCODZip &zip) {
        zip.putInt(size);
        zip.putInt(inventory.size());
        for (Actor **it = inventory.begin(); it != inventory.end(); it++) {
                (*it)->save(zip);
        }
}

bool Container::add(Actor *actor) {
        if (size > 0 && inventory.size() >= size) {
                // inventory full
                return false;
        }
        inventory.push(actor);
        return true;
}
```

```
void Container::remove(Actor *actor) {
	inventory.remove(actor);
}
```

Destructible.h

```
#include <stdio.h>
#include "main.h"

Destructible::Destructible(float maxHp, float defense, const char *corpseName,
int xp) :
	maxHp(maxHp), hp(maxHp), defense(defense), corpseName(corpseName),
xp(xp) {
}

Destructible *Destructible::create(TCODZip &zip) {
	DestructibleType type = (DestructibleType)zip.getInt();
	Destructible *destructible = NULL;
	switch(type) {
		case ENEMY:
		    destructible = new EnemyDestructible(0, 0, NULL, 0);
		    break;
		case PLAYER:
		    destructible = new PlayerDestructible(0, 0, NULL);
		    break;
	}
	destructible->load(zip);
	return destructible;
}

void Destructible::load(TCODZip &zip) {
	maxHp = zip.getFloat();
	hp = zip.getFloat();
	defense = zip.getFloat();
	evasion = zip.getFloat();
	corpseName = strdup(zip.getString());
	xp = zip.getInt();
}

void Destructible::save(TCODZip &zip) {
	zip.putFloat(maxHp);
	zip.putFloat(hp);
	zip.putFloat(defense);
	zip.putFloat(evasion);
	zip.putString(corpseName);
	zip.putInt(xp);
}

float Destructible::takeDamage(Actor *owner, float damage) {
	if (damage <= 0) {
	damage = 0;
    } else {
	damage -= defense;
	if (damage > 0) {
	    hp -= damage;
```

```
            if (hp <= 0) {
                die(owner);
            }
        }
    }
        return damage;
}

float Destructible::heal(float amount) {
        hp += amount;
        if (hp > maxHp) {
                amount -= hp-maxHp;
                hp = maxHp;
        }
        return amount;
}

void Destructible::die(Actor *owner) {
    transformIntoCorpse(owner);
    // Make sure corpses are drawn before living actors
    engine.sendToBack(owner);
}

void Destructible::transformIntoCorpse(Actor *owner) {
    // transform the actor into a corpse!
    owner->ch = '%';
    owner->col = TCODColor::darkRed;
    owner->name = corpseName;
    owner->blocks = false;
}

EnemyDestructible::EnemyDestructible(float maxHp, float defense, const char
*corpseName, int xp) :
        Destructible(maxHp, defense, corpseName, xp) {
}

void EnemyDestructible::save(TCODZip &zip) {
        zip.putInt(ENEMY);
        Destructible::save(zip);
}

void EnemyDestructible::die(Actor *owner) {
        // transform it into a nasty corpse! it doesn't block, can't be
        // attacked and doesn't move
        engine.gui->message(TCODColor::lightGrey, "%s is dead. You gain %d xp",
owner->name, xp);
        engine.player->destructible->xp += xp;
        Destructible::die(owner);
}

PlayerDestructible::PlayerDestructible(float maxHp, float defense, const char
*corpseName) :
        Destructible(maxHp, defense, corpseName, 0) {
}
```

```
void PlayerDestructible::die(Actor *owner) {
	engine.gui->message(TCODColor::red, "You died!");
	Destructible::die(owner);
	engine.gameStatus = Engine::DEFEAT;
}

void PlayerDestructible::save(TCODZip &zip) {
	zip.putInt(PLAYER);
	Destructible::save(zip);
}
```

Engine.cpp

```
#include <math.h>
#include "main.h"

Engine::Engine(int screenWidth, int screenHeight) : gameStatus(STARTUP),
	player(NULL), map(NULL), fovRadius(10),
	screenWidth(screenWidth), screenHeight(screenHeight), level(1) {
    TCODConsole::initRoot(screenWidth, screenHeight, "Bong's Dungeon", false);
    gui = new Gui();
}

void Engine::init() {
    player = new Actor(40, 25, '@', "player", TCODColor::white);
    player->destructible = new PlayerDestructible(30, 2, "your dead body");
    player->attacker = new Attacker(5, 1);
    player->ai = new PlayerAi();
    player->container = new Container(26);
    actors.push(player);
    stairs = new Actor(0, 0, '>', "stairs", TCODColor::white);
    stairs->blocks = false;
    stairs->fovOnly = false;
    actors.push(stairs);
    map = new Map(80, 43);
    map->init(true);
    gui->message(TCODColor::red, "Welcome stranger!\nPrepare to perish in the
dungeon of Bong.");
    gameStatus = STARTUP;
}

Engine::~Engine() {
	term();
    delete gui;
}

int Engine::getLevel() const {
    // get the level from the Engine.h
    return level;
}

const int SAVEGAME_VERSION=0x1100;
void Engine::load(bool pause) {
	TCODZip zip;
	engine.gui->menu.clear();
```

```
	engine.gui->menu.addItem(Menu::NEW_GAME, "New game");
	if (TCODSystem::fileExists("game.sav")) {
		zip.loadFromFile("game.sav");
		int version = zip.getInt();
		if (version == SAVEGAME_VERSION) {
			engine.gui->menu.addItem(Menu::CONTINUE, "Continue");
		}
	}
	engine.gui->menu.addItem(Menu::EXIT,"Exit");

	Menu::MenuItemCode menuItem = engine.gui->menu.pick(pause ?
Menu::PAUSE : Menu::MAIN);
	if (menuItem == Menu::EXIT || menuItem == Menu::NONE) {
		// Exit or window closed
		exit(0);
	} else if (menuItem == Menu::NEW_GAME) {
		// New game
		engine.term();
		engine.init();
	} else {
		// continue a saved game
		engine.term();
		// load the map
		level = zip.getInt();
		int width = zip.getInt();
		int height = zip.getInt();
		map = new Map(width, height);
		map->load(zip);
		// then the player
		player = new Actor(0, 0, 0, NULL, TCODColor::white);
		actors.push(player);
		player->load(zip);
		// the stairs
		stairs = new Actor(0, 0, 0, NULL, TCODColor::white);
		stairs->load(zip);
		actors.push(stairs);
		// then all other actors
		int nbActors = zip.getInt();
		while (nbActors > 0) {
			Actor *actor = new Actor(0, 0, 0, NULL, TCODColor::white);
			actor->load(zip);
			actors.push(actor);
			nbActors--;
		}
		// finally the message log
		gui->load(zip);
		// to force FOV recomputation
		gameStatus = STARTUP;
	}
}

void Engine::save() {
	if (player->destructible->isDead()) {
		// Remove the save file if the player is dead
```

```
        TCODSystem::deleteFile("game.sav");
        } else {
                TCODZip zip;
                zip.putInt(SAVEGAME_VERSION);
                zip.putInt(level);
                // save the map first
                zip.putInt(map->width);
                zip.putInt(map->height);
                map->save(zip);
                // then the player
                player->save(zip);
                // then the stairs
                stairs->save(zip);
                // then all the other actors
                zip.putInt(actors.size() - 2);
                for (Actor **it = actors.begin(); it != actors.end(); it++) {
                        if (*it != player && *it != stairs) {
                                (*it)->save(zip);
                        }
                }
                // finally the message log
                gui->save(zip);
                zip.saveToFile("game.sav");
        }
}

void Engine::term() {
    actors.clearAndDelete();
    if (map) delete map;
    gui->clear();
}

void Engine::update() {
        if (gameStatus == STARTUP) map->computeFov();
        gameStatus = IDLE;
    TCODSystem::checkForEvent(TCOD_EVENT_KEY_PRESS|TCOD_EVENT_MOUSE, &lastKey,
&mouse);
    if (lastKey.vk == TCODK_ESCAPE) {
        save();
        load(true);
    }
    player->update();
    if (gameStatus == NEW_TURN) {
            for (Actor **iterator = actors.begin(); iterator != actors.end();
iterator++) {
                Actor *actor =* iterator;
                if (actor != player) {
                    actor->update();
                }
            }
        }
}

void Engine::render() {
```

```
	TCODConsole::root->clear();
	// draw the map
	map->render();
	// draw the actors
	for (Actor **iterator = actors.begin(); iterator != actors.end();
iterator++) {
		Actor *actor = *iterator;
		if (actor != player && ((!actor->fovOnly &&
	    map->isExplored(actor->x, actor->y)) ||
	    map->isInFov(actor->x, actor->y))) {
		    actor->render();
		}
	}
	player->render();
	// show the player's stats
	gui->render();
}

void Engine::sendToBack(Actor *actor) {
	actors.remove(actor);
	actors.insertBefore(actor, 0);
}

Actor *Engine::getActor(int x, int y) const {
	for (Actor **iterator = actors.begin(); iterator != actors.end();
iterator++) {
		Actor *actor = *iterator;
		if (actor->x == x && actor->y == y &&
	    actor->destructible && !actor->destructible->isDead()) {
			return actor;
		}
	}
	return NULL;
}

Actor *Engine::getClosestEnemy(int x, int y, float range) const {
	Actor *closest = NULL;
	float bestDistance = 1E6f;
	for (Actor **iterator = actors.begin(); iterator != actors.end();
iterator++) {
		Actor *actor = *iterator;
		if (actor != player && actor->destructible &&
			!actor->destructible->isDead()) {
			float distance = actor->getDistance(x, y);
			if (distance < bestDistance && (distance <= range || range
== 0.0f)) {
				bestDistance = distance;
				closest = actor;
			}
		}
	}
	return closest;
}
```

```
bool Engine::tileSelection(int *x, int *y, float maxRange) {
	while (!TCODConsole::isWindowClosed()) {
		render();
		// highlight the possible range
		for (int cx = 0; cx < map->width; cx++) {
			for (int cy = 0; cy < map->height; cy++) {
				if (map->isInFov(cx,cy) &&
					(maxRange == 0 || player->getDistance(cx,
cy) <= maxRange)) {
					TCODColor col = TCODConsole::root-
>getCharBackground(cx, cy);
					col = col * 1.2f;
					TCODConsole::root->setCharBackground(cx,
cy, col);
				}
			}
		}

		TCODSystem::checkForEvent(TCOD_EVENT_KEY_PRESS|TCOD_EVENT_MOUSE,
&lastKey, &mouse);

		if (map->isInFov(mouse.cx, mouse.cy) &&
			(maxRange == 0 || player->getDistance(mouse.cx,mouse.cy)
<= maxRange)) {
			TCODConsole::root->setCharBackground(mouse.cx, mouse.cy,
TCODColor::white);
			if (mouse.lbutton_pressed) {
				*x = mouse.cx;
				*y = mouse.cy;
				return true;
			}
		}

		if (mouse.rbutton_pressed || lastKey.vk != TCODK_NONE) {
			return false;
		}

		TCODConsole::flush();
	}
	return false;
}

void Engine::nextLevel() {
	level++;
	gui->message(TCODColor::lightViolet, "You take a moment to rest, and
recover your strength.");
    player->destructible->heal(player->destructible->maxHp / 2);
	gui->message(TCODColor::red, "After a rare moment of peace, you
descend\ndeeper into the heart of the dungeon...");

	// Extract the map creation and actor initialization code into a
separate function
    initializeNewLevel();
```

```
        gameStatus = STARTUP;
}

void Engine::initializeNewLevel() {
    delete map;
    // delete all actors but player and stairs
    for (Actor **it = actors.begin(); it != actors.end(); it++) {
        if (*it != player && *it != stairs) {
                delete *it;
                it = actors.remove(it);
        }
    }
    // create a new map
    map = new Map(80, 43);
    map->init(true);
}
```

Gui.cpp

```
#include <stdio.h>
#include <stdarg.h>
#include "main.h"

static const int PANEL_HEIGHT = 7;
static const int BAR_WIDTH = 20;
static const int MSG_X = BAR_WIDTH+2;
static const int MSG_HEIGHT = PANEL_HEIGHT-1;

Gui::Gui() {
          con = new TCODConsole(engine.screenWidth, PANEL_HEIGHT);
}

Gui::~Gui() {
          delete con;
          clear();
}

void Gui::load(TCODZip &zip) {
          int nbMessages = zip.getInt();
          while (nbMessages > 0) {
               const char *text = zip.getString();
               TCODColor col = zip.getColor();
               message(col, text);
               nbMessages--;
          }
}

void Gui::save(TCODZip &zip) {
          zip.putInt(log.size());
          for (Message **it = log.begin(); it != log.end(); it++) {
               zip.putString((*it)->text);
               zip.putColor(&(*it)->col);
          }
}

void Gui::clear() {
```

```
	log.clearAndDelete();
}

void Gui::render() {
	// clear the GUI console
	con->setDefaultBackground(TCODColor::black);
	con->clear();

	// draw the health bar
	renderBar(1, 1, BAR_WIDTH, "HP", engine.player->destructible->hp,
       engine.player->destructible->maxHp,
		TCODColor::lightRed,TCODColor::darkerRed);

	// draw the XP bar
	PlayerAi *ai = (PlayerAi *)engine.player->ai;
	char xpTxt[128];
	sprintf(xpTxt, "XP(%d)", ai->xpLevel);
	renderBar(1, 5, BAR_WIDTH, xpTxt, engine.player->destructible->xp,
		ai->getNextLevelXp(),
		TCODColor::lightViolet,TCODColor::darkerViolet);

	// draw the message log
	int y = 1;
	float colorCoef = 0.4f;
	for (Message **it = log.begin(); it != log.end(); it++) {
		Message *message = *it;
		con->setDefaultForeground(message->col * colorCoef);
		con->print(MSG_X, y, message->text);
		y++;
		if (colorCoef < 1.0f) {
			colorCoef += 0.3f;
		}
	}

	// mouse look
	renderMouseLook();

	// dungeon level
	con->setDefaultForeground(TCODColor::white);
	con->print(3, 3, "Dungeon level %d", engine.level);

	// blit the GUI console on the root console
	TCODConsole::blit(con, 0, 0, engine.screenWidth, PANEL_HEIGHT,
TCODConsole::root, 0, engine.screenHeight - PANEL_HEIGHT);
}

void Gui::renderBar(int x, int y, int width, const char *name,
	float value, float maxValue, const TCODColor &barColor,
	const TCODColor &backColor) {
	// fill the background
	con->setDefaultBackground(backColor);
	con->rect(x, y, width, 1, false, TCOD_BKGND_SET);

	int barWidth = (int)(value / maxValue * width);
```

```
          if (barWidth > 0) {
               // draw the bar
               con->setDefaultBackground(barColor);
               con->rect(x, y, barWidth, 1, false, TCOD_BKGND_SET);
          }
          // print text on top of the bar
          con->setDefaultForeground(TCODColor::white);
          con->printEx(x + width / 2, y, TCOD_BKGND_NONE, TCOD_CENTER,
"%s : %g/%g", name, value, maxValue);
}

Gui::Message::Message(const char *text, const TCODColor &col) :
text(strdup(text)), col(col) {}

Gui::Message::~Message() {
          free(text);
}

void Gui::renderMouseLook() {
          if (!engine.map->isInFov(engine.mouse.cx, engine.mouse.cy)) {
               // if mouse is out of fov, nothing to render
               return;
          }
          char buf[128] = "";
          bool first = true;
          for (Actor **it = engine.actors.begin(); it != engine.actors.end();
it++) {
               Actor *actor = *it;
               // find actors under the mouse cursor
               if (actor->x == engine.mouse.cx && actor->y == engine.mouse.cy) {
                       if (!first) {
                               strcat(buf,", ");
                       } else {
                               first = false;
                       }
                       strcat(buf, actor->name);
               }
          }
          // display the list of actors under the mouse cursor
          con->setDefaultForeground(TCODColor::lightGrey);
          con->print(1, 0, buf);
}

void Gui::message(const TCODColor &col, const char *text, ...) {
          // build the text
          va_list ap;
          char buf[128];
          va_start(ap, text);
          vsprintf(buf, text, ap);
          va_end(ap);

          char *lineBegin = buf;
          char *lineEnd;
          do {
```

```
                // make room for the new message
                if (log.size() == MSG_HEIGHT) {
                        Message *toRemove = log.get(0);
                        log.remove(toRemove);
                        delete toRemove;
                }

                // detect end of the line
                lineEnd=strchr(lineBegin,'\n');
                if (lineEnd) {
                        *lineEnd = '\0';
                }

                // add a new message to the log
                Message *msg = new Message(lineBegin, col);
                log.push(msg);

                // go to next line
                lineBegin = lineEnd+1;
           } while (lineEnd);
}


Menu::~Menu() {
           clear();
}

void Menu::clear() {
           items.clearAndDelete();
}

void Menu::addItem(MenuItemCode code, const char *label) {
           MenuItem *item = new MenuItem();
           item->code = code;
           item->label = label;
           items.push(item);
}

const int PAUSE_MENU_WIDTH = 30;
const int PAUSE_MENU_HEIGHT = 15;
Menu::MenuItemCode Menu::pick(DisplayMode mode) {
           int selectedItem = 0;
           int menux, menuy;
           if (mode == PAUSE) {
                menux = engine.screenWidth / 2 - PAUSE_MENU_WIDTH / 2;
                menuy = engine.screenHeight / 2 - PAUSE_MENU_HEIGHT / 2;
                TCODConsole::root->setDefaultForeground(TCODColor(200, 180, 50));
                TCODConsole::root->printFrame(menux, menuy, PAUSE_MENU_WIDTH,
PAUSE_MENU_HEIGHT, true, TCOD_BKGND_ALPHA(70), "menu");
                menux += 2;
                menuy += 3;
           } else {
                static TCODImage img("menu_background1.png");
                img.blit2x(TCODConsole::root, 0, 0);
```

```
                menux = 10;
                menuy = TCODConsole::root->getHeight()/3;
          }

          while(!TCODConsole::isWindowClosed()) {
                int currentItem = 0;
                for (MenuItem **it = items.begin(); it != items.end(); it++) {
                      if (currentItem == selectedItem) {
                            TCODConsole::root-
>setDefaultForeground(TCODColor::lighterOrange);
                      } else {
                            TCODConsole::root-
>setDefaultForeground(TCODColor::lightGrey);
                      }
                      TCODConsole::root->print(menux, menuy + currentItem*3,
(*it)->label);
                      currentItem++;
                }
                TCODConsole::flush();

                // check key presses
                TCOD_key_t key;
                TCODSystem::checkForEvent(TCOD_EVENT_KEY_PRESS,&key,NULL);
                switch (key.vk) {
                      case TCODK_UP :
                            selectedItem--;
                            if (selectedItem < 0) {
                                  selectedItem=items.size() - 1;
                            }
                      break;
                      case TCODK_DOWN :
                            selectedItem = (selectedItem + 1) % items.size();
                      break;
                      case TCODK_ENTER :
                            return items.get(selectedItem)->code;
                      default : break;
                }
          }
          return NONE;
}
```

Item.cpp

```
#include <stdio.h>
#include <stdarg.h>
#include "main.h"

static const int PANEL_HEIGHT = 7;
static const int BAR_WIDTH = 20;
static const int MSG_X = BAR_WIDTH+2;
static const int MSG_HEIGHT = PANEL_HEIGHT-1;

Gui::Gui() {
       con = new TCODConsole(engine.screenWidth, PANEL_HEIGHT);
}
```

```
Gui::~Gui() {
	delete con;
	clear();
}

void Gui::load(TCODZip &zip) {
	int nbMessages = zip.getInt();
	while (nbMessages > 0) {
		const char *text = zip.getString();
		TCODColor col = zip.getColor();
		message(col, text);
		nbMessages--;
	}
}

void Gui::save(TCODZip &zip) {
	zip.putInt(log.size());
	for (Message **it = log.begin(); it != log.end(); it++) {
		zip.putString((*it)->text);
		zip.putColor(&(*it)->col);
	}
}

void Gui::clear() {
	log.clearAndDelete();
}

void Gui::render() {
	// clear the GUI console
	con->setDefaultBackground(TCODColor::black);
	con->clear();

	// draw the health bar
	renderBar(1, 1, BAR_WIDTH, "HP", engine.player->destructible->hp,
	engine.player->destructible->maxHp,
		TCODColor::lightRed,TCODColor::darkerRed);

	// draw the XP bar
	PlayerAi *ai = (PlayerAi *)engine.player->ai;
	char xpTxt[128];
	sprintf(xpTxt, "XP(%d)", ai->xpLevel);
	renderBar(1, 5, BAR_WIDTH, xpTxt, engine.player->destructible->xp,
		ai->getNextLevelXp(),
		TCODColor::lightViolet,TCODColor::darkerViolet);

	// draw the message log
	int y = 1;
	float colorCoef = 0.4f;
	for (Message **it = log.begin(); it != log.end(); it++) {
		Message *message = *it;
		con->setDefaultForeground(message->col * colorCoef);
		con->print(MSG_X, y, message->text);
		y++;
		if (colorCoef < 1.0f) {
```

```
                        colorCoef += 0.3f;
                }
        }

        // mouse look
        renderMouseLook();

        // dungeon level
        con->setDefaultForeground(TCODColor::white);
        con->print(3, 3, "Dungeon level %d", engine.level);

        // blit the GUI console on the root console
        TCODConsole::blit(con, 0, 0, engine.screenWidth, PANEL_HEIGHT,
TCODConsole::root, 0, engine.screenHeight - PANEL_HEIGHT);
}

void Gui::renderBar(int x, int y, int width, const char *name,
        float value, float maxValue, const TCODColor &barColor,
        const TCODColor &backColor) {
        // fill the background
        con->setDefaultBackground(backColor);
        con->rect(x, y, width, 1, false, TCOD_BKGND_SET);

        int barWidth = (int)(value / maxValue * width);
        if (barWidth > 0) {
                // draw the bar
                con->setDefaultBackground(barColor);
                con->rect(x, y, barWidth, 1, false, TCOD_BKGND_SET);
        }
        // print text on top of the bar
        con->setDefaultForeground(TCODColor::white);
        con->printEx(x + width / 2, y, TCOD_BKGND_NONE, TCOD_CENTER,
"%s : %g/%g", name, value, maxValue);
}

Gui::Message::Message(const char *text, const TCODColor &col) :
text(strdup(text)), col(col) {}

Gui::Message::~Message() {
        free(text);
}

void Gui::renderMouseLook() {
        if (!engine.map->isInFov(engine.mouse.cx, engine.mouse.cy)) {
                // if mouse is out of fov, nothing to render
                return;
        }
        char buf[128] = "";
        bool first = true;
        for (Actor **it = engine.actors.begin(); it != engine.actors.end();
it++) {
                Actor *actor = *it;
                // find actors under the mouse cursor
                if (actor->x == engine.mouse.cx && actor->y == engine.mouse.cy) {
```

```
                        if (!first) {
                                strcat(buf,", ");
                        } else {
                                first = false;
                        }
                        strcat(buf, actor->name);
                }
        }
        // display the list of actors under the mouse cursor
        con->setDefaultForeground(TCODColor::lightGrey);
        con->print(1, 0, buf);
}

void Gui::message(const TCODColor &col, const char *text, ...) {
        // build the text
        va_list ap;
        char buf[128];
        va_start(ap, text);
        vsprintf(buf, text, ap);
        va_end(ap);

        char *lineBegin = buf;
        char *lineEnd;
        do {
                // make room for the new message
                if (log.size() == MSG_HEIGHT) {
                        Message *toRemove = log.get(0);
                        log.remove(toRemove);
                        delete toRemove;
                }

                // detect end of the line
                lineEnd=strchr(lineBegin,'\n');
                if (lineEnd) {
                        *lineEnd = '\0';
                }

                // add a new message to the log
                Message *msg = new Message(lineBegin, col);
                log.push(msg);

                // go to next line
                lineBegin = lineEnd+1;
        } while (lineEnd);
}


Menu::~Menu() {
        clear();
}

void Menu::clear() {
        items.clearAndDelete();
}
```

```
void Menu::addItem(MenuItemCode code, const char *label) {
	MenuItem *item = new MenuItem();
	item->code = code;
	item->label = label;
	items.push(item);
}

const int PAUSE_MENU_WIDTH = 30;
const int PAUSE_MENU_HEIGHT = 15;
Menu::MenuItemCode Menu::pick(DisplayMode mode) {
	int selectedItem = 0;
	int menux, menuy;
	if (mode == PAUSE) {
		menux = engine.screenWidth / 2 - PAUSE_MENU_WIDTH / 2;
		menuy = engine.screenHeight / 2 - PAUSE_MENU_HEIGHT / 2;
		TCODConsole::root->setDefaultForeground(TCODColor(200, 180, 50));
		TCODConsole::root->printFrame(menux, menuy, PAUSE_MENU_WIDTH,
PAUSE_MENU_HEIGHT, true, TCOD_BKGND_ALPHA(70), "menu");
		menux += 2;
		menuy += 3;
	} else {
		static TCODImage img("menu_background1.png");
		img.blit2x(TCODConsole::root, 0, 0);
		menux = 10;
		menuy = TCODConsole::root->getHeight()/3;
	}

	while(!TCODConsole::isWindowClosed()) {
		int currentItem = 0;
		for (MenuItem **it = items.begin(); it != items.end(); it++) {
			if (currentItem == selectedItem) {
				TCODConsole::root-
>setDefaultForeground(TCODColor::lighterOrange);
			} else {
				TCODConsole::root-
>setDefaultForeground(TCODColor::lightGrey);
			}
			TCODConsole::root->print(menux, menuy + currentItem*3,
(*it)->label);
			currentItem++;
		}
		TCODConsole::flush();

		// check key presses
		TCOD_key_t key;
		TCODSystem::checkForEvent(TCOD_EVENT_KEY_PRESS,&key,NULL);
		switch (key.vk) {
			case TCODK_UP :
				selectedItem--;
				if (selectedItem < 0) {
					selectedItem=items.size() - 1;
				}
			break;
```

```
                        case TCODK_DOWN :
                                selectedItem = (selectedItem + 1) % items.size();
                        break;
                        case TCODK_ENTER :
                                return items.get(selectedItem)->code;
                        default : break;
                }
        }
        return NONE;
}
```

Map.cpp

```
#include <cmath>
#include "main.h"

static const int ROOM_MAX_SIZE = 12;
static const int ROOM_MIN_SIZE = 6;
static const int MAX_ROOM_ENEMIES = 3;
static const int MAX_ROOM_ITEMS = 2;

class BspListener : public ITCODBspCallback {
private :
    Map &map; // a map to dig
    int roomNum; // room number
    int lastx, lasty; // center of the last room
public :
    BspListener(Map &map) : map(map), roomNum(0) {}
    bool visitNode(TCODBsp *node, void *userData) {
        if (node->isLeaf()) {
                int x, y, w, h;
                        // dig a room
                        bool withActors = (bool)userData;
                        w = map.rng->getInt(ROOM_MIN_SIZE, node->w-2);
                        h = map.rng->getInt(ROOM_MIN_SIZE, node->h-2);
                        x = map.rng->getInt(node->x+1, node->x+node->w-w-1);
                        y = map.rng->getInt(node->y+1, node->y+node->h-h-1);
                        map.createRoom(roomNum == 0, x, y, x+w-1, y+h-1,
withActors);
                        if (roomNum != 0) {
                            // dig a corridor from last room
                            map.dig(lastx, lasty, x+w/2, lasty);
                            map.dig(x+w/2, lasty, x+w/2, y+h/2);
                        }
            lastx = x+w/2;
            lasty = y+h/2;
            roomNum++;
        }
        return true;
    }
};

Map::Map(int width, int height) : width(width), height(height) {
        seed = TCODRandom::getInstance()->getInt(0, 0x7FFFFFFF);
}
```

```
void Map::init(bool withActors) {
        rng = new TCODRandom(seed, TCOD_RNG_CMWC);
    tiles = new Tile[width * height];
    map = new TCODMap(width, height);
    TCODBsp bsp(0, 0, width, height);
    bsp.splitRecursive(rng, 8, ROOM_MAX_SIZE,ROOM_MAX_SIZE, 1.5f, 1.5f);
    BspListener listener(*this);
    bsp.traverseInvertedLevelOrder(&listener, (void *)withActors);
}

Map::~Map() {
    delete [] tiles;
    delete map;
}

void Map::load(TCODZip &zip) {
        seed = zip.getInt();
    init(false);
        for (int i = 0; i < width * height; i++) {
                tiles[i].explored = zip.getInt();
        }
}

void Map::save(TCODZip &zip) {
        zip.putInt(seed);
        for (int i = 0; i < width * height; i++) {
                zip.putInt(tiles[i].explored);
        }
}

void Map::dig(int x1, int y1, int x2, int y2) {
    if (x2 < x1) {
        int tmp = x2;
        x2 = x1;
        x1 = tmp;
    }
    if (y2 < y1) {
        int tmp = y2;
        y2 = y1;
        y1 = tmp;
    }
    for (int tilex=x1; tilex <= x2; tilex++) {
        for (int tiley=y1; tiley <= y2; tiley++) {
            map->setProperties(tilex, tiley, true, true);
        }
    }
}

void Map::addEnemy(int x, int y) {
        TCODRandom *rng = TCODRandom::getInstance();
        int mobGenerator = rng->getInt(0, 200); // random generator
        int levelScaler = engine.getLevel(); // level scaler
        if (levelScaler >= 50) levelScaler = 50; // maximum limit of level scale
    if (mobGenerator < 160) {
```

```
        // create an oop
        Actor *oop = new Actor(x, y, 'o', "oop", TCODColor::desaturatedGreen);
        oop->destructible = new EnemyDestructible(10+floor(levelScaler/3),
0+floor(levelScaler/6), "dead oop", 35);
        oop->attacker = new Attacker(3+floor(levelScaler/6),
0+floor(levelScaler/6));
        oop->ai = new EnemyAi();
        engine.actors.push(oop);
    } else if (mobGenerator >= 160 && mobGenerator <= 195) {
        // create a c++
        Actor *cplus = new Actor(x, y, 'C', "c++", TCODColor::darkerGreen);
        cplus->destructible = new EnemyDestructible(16+floor(levelScaler/3),
1+floor(levelScaler/5), "c++ remains", 100);
        cplus->attacker = new Attacker(4+floor(levelScaler/5),
1+floor(levelScaler/5));
        cplus->ai = new EnemyAi();
        engine.actors.push(cplus);
    } else {
        if (levelScaler >= 5 && mobGenerator == 199) {
            // create a bong
            Actor *cplus = new Actor(x, y, 'B', "BONG", TCODColor::darkerGreen);
            cplus->destructible = new EnemyDestructible(230+floor(levelScaler-4),
3+floor(levelScaler-4), "no BONG", 500);
            cplus->attacker = new Attacker(7+floor(levelScaler-4),
3+floor(levelScaler-4));
            cplus->ai = new EnemyAi();
            engine.actors.push(cplus);
        } else {
            // create a test
            Actor *cplus = new Actor(x, y, 'T', "test", TCODColor::darkerGreen);
            cplus->destructible = new EnemyDestructible(20+floor(levelScaler/3),
1+floor(levelScaler/4), "destroyed test", 150);
            cplus->attacker = new Attacker(5+floor(levelScaler/3),
2+floor(levelScaler/3));
            cplus->ai = new EnemyAi();
            engine.actors.push(cplus);
        }
    }
}

void Map::addItem(int x, int y) {
        TCODRandom *rng = TCODRandom::getInstance();
        int dice = rng->getInt(0, 200);
        if (dice < 140) {
                // create a health potion
                Actor *healthPotion = new Actor(x, y, '!', "health potion",
TCODColor::violet);
                healthPotion->blocks = false;
                healthPotion->item = new Healer(4);
                engine.actors.push(healthPotion);
        } else if (140 <= dice && dice < 140+20) {
                // create a snap shot
                Actor *snapShot = new Actor(x, y, '#', "snap shot",
TCODColor::lightYellow);
```

```
                snapShot->blocks = false;
                snapShot->item = new SnapShot(5, 20);
                engine.actors.push(snapShot);
        } else if (140+20 <= dice && dice < 140+20+10) {
                // create a wide shot
                Actor *wideShot = new Actor(x, y, '#', "wide shot",
TCODColor::lightYellow);
                wideShot->blocks = false;
                wideShot->item = new WideShot(7, 12);
                engine.actors.push(wideShot);
        } else if (140+20+10 <= dice && dice < 140+20+10+10) {
                // create a confuse shot
                Actor *confuseShot = new Actor(x, y, '#', "confuse shot",
TCODColor::lightYellow);
                confuseShot->blocks = false;
                confuseShot->item = new ConfuseShot(10, 8);
                engine.actors.push(confuseShot);
        } else {
        // create a super health potion
                Actor *superhealthPotion = new Actor(x, y, '?', "super health
potion", TCODColor::violet);
                superhealthPotion->blocks = false;
                superhealthPotion->item = new Healer(20);
                engine.actors.push(superhealthPotion);
        }
}

void Map::createRoom(bool first, int x1, int y1, int x2, int y2, bool
withActors) {
    dig (x1, y1, x2, y2);
    if (!withActors) {
        return;
    }
    if (first) {
        // put the player in the first room
        engine.player->x = (x1 + x2) / 2;
        engine.player->y = (y1 + y2) / 2;
    } else {
                TCODRandom *rng = TCODRandom::getInstance();
                // add monsters
                int nbEnemies = rng->getInt(0, MAX_ROOM_ENEMIES);
                while (nbEnemies > 0) {
                    int x = rng->getInt(x1, x2);
                    int y = rng->getInt(y1, y2);
                if (canWalk(x, y)) {
                                addEnemy(x, y);
                        }
                    nbEnemies--;
                }
                // add items
                int nbItems = rng->getInt(0, MAX_ROOM_ITEMS);
                while (nbItems > 0) {
                    int x = rng->getInt(x1, x2);
                    int y = rng->getInt(y1, y2);
```

```
                if (canWalk(x, y)) {
                        addItem(x, y);
                    }
                nbItems--;
                }
                // set stairs position
                engine.stairs->x = (x1 + x2) / 2;
                engine.stairs->y = (y1 + y2) / 2;
    }
}

bool Map::isWall(int x, int y) const {
    return !map->isWalkable(x, y);
}

bool Map::canWalk(int x, int y) const {
    if (isWall(x, y)) {
        // this is a wall
        return false;
    }
    for (Actor **iterator = engine.actors.begin();
        iterator != engine.actors.end(); iterator++) {
        Actor *actor = *iterator;
        if (actor->blocks && actor->x == x && actor->y == y) {
            // there is a blocking actor here. cannot walk
            return false;
        }
    }
    return true;
}

bool Map::isExplored(int x, int y) const {
    return tiles[x + y * width].explored;
}

bool Map::isInFov(int x, int y) const {
        if (x < 0 || x >= width || y < 0 || y >= height) {
                return false;
        }
    if (map->isInFov(x, y)) {
        tiles[x + y * width].explored=true;
        return true;
    }
    return false;
}

void Map::computeFov() {
    map->computeFov(engine.player->x, engine.player->y, engine.fovRadius);
}

void Map::render() const {
    static const TCODColor darkWall(0, 0, 100);
    static const TCODColor darkGround(50, 50, 150);
        static const TCODColor lightWall(130, 110, 50);
```

```
        static const TCODColor lightGround(200, 180, 50);

        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {
                if (isInFov(x, y)) {
                    TCODConsole::root->setCharBackground(x, y, isWall(x, y) ?
lightWall : lightGround);
                } else if (isExplored(x, y)) {
                    TCODConsole::root->setCharBackground(x, y, isWall(x, y) ?
darkWall : darkGround);
                }
            }
        }
}
```

main.cpp

```
#include "main.h"

Engine engine(80, 50);

int main() {
        engine.load();
    while (!TCODConsole::isWindowClosed()) {
        engine.update();
        engine.render();
                TCODConsole::flush();
    }
    engine.save();
    return 0;
}
```