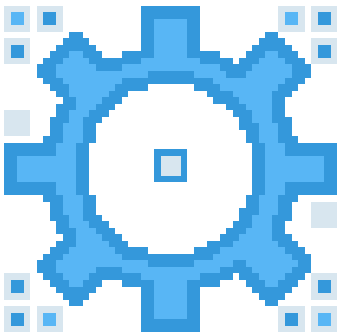
JTileEngine - User Manual

Java 2D Game Engine



Petr Chalupa, 2025

Table of contents

Installation

Prerequisites

- **Java Development Kit (JDK) 21**
- **Apache Maven**

Maven automatically manages JavaFX, so no separate installation is necessary.

Building the project

1. Clone the repository and navigate into the root directory
2. Then build the entire project using: `mvn clean install`

This will compile both modules: `engine` and `ui`.

Running the application

From the root folder, navigate to the `ui` module:

```
cd ui
mvn javafx:run
```

This launches the application with the main class `ui.App`.

Technology

The main technologies used include:

- Java
- Maven
- JavaFX
- JUnit Jupiter
- JSON (org.json)

Assets

Assets such as the logo and sprites, used in both the `ui` and `engine`, were manually created for the engine using the free online editor **Piskel**. Sprites are defined as 64×64 pixel images. Examples are shown below:

Figure 1: Engine logo

Figure 2: Sword sprite

The bug report icon was taken from **Google Material Symbols and Icons** website.

Project structure

The project is organized as a multi-module Maven project:

```
jtileengine/
├── pom.xml
├── engine/          # Game & Engine logic
│   └── pom.xml
├── ui/              # JavaFX GUI
│   └── pom.xml
```

This design was chosen because it allows the graphical user interface to be changed more easily, while not interfering with the engine itself.

User interface

- Contains the JavaFX front-end logic and layout
- Depends on the `engine` module
- The main entry point is `ui.App`

Layout of the GUI was built by **Gluon Scene Builder** and therefore different screens are also different `fxml` files. The interface is mostly styled by `CSS` for its convenience and better scalability (sharing styles across files etc.). Every screen has its own style sheet, and there is one style sheet containing general shared styles.

Main Menu

This screen is the application's landing page.
Functions:

- Start game (opens the **Level Menu** with built-in levels pre-selected)
- Access **Settings** and **Level Editor**

Level Menu

This screen is where user can select which level to play. There is a toggle button to switch between built-in and custom (imported or created) levels.

Level Editor

This screen consists of two parts. Firstly, there is a level selection box and secondly the actual editor. User can select from already loaded custom levels, import or create a new level. The editor itself is divided into configuration settings and map editor.

Settings

This screen allows user to change the engine-wide settings regarding rendered tile size, logging or rendering debug information. The settings are used directly without the need to restart the app and they are also reloaded on startup.

Game view

This screen is where the actual game is rendered to. The game can be paused by pressing the `ESC` key or by clicking on the pause button in the left upper corner of the screen.

Figure 3: Game view

Controls:

- Movement: `W`, `A`, `S`, `D`
- Interacting with the environment: `E`
- Changing active inventory slot: `mouse scrolling`
  - `Player` inventory also allows `number keys`
- Changing the active inventory (f.ex. `Player` ↔ `Chest`): `Tab`
- Transferring item from the active inventory: `Space`
- Using the active item: `LMB`
- Dropping the active item: `Q`

Engine architecture

- Contains the core game engine logic
- Exposes classes used by the UI module
- Also includes tests using `junit-jupiter`

Engine consists of multiple directories that group classes by purpose:

```
engine/
├── Engine.java     # The main engine class
├── core/           # Core engine classes
├── gameobjects/    # Game object definitions
├── ui/             # Engine-side UI data
├── utils/          # Utility classes
```

Engine.java

This class is the main entry point of the engine. It exposes engine resources and methods for easy life-cycle control such as `init`, `loadLevel` and `shutdown`. While it is not strictly enforced, accessing engine components via this main class is strongly advised (rather than accessing them directly).

Core

This directory holds the most important engine classes responsible for the game loop, input handling, rendering, game camera etc. Game loop is executed on a dedicated thread handled by the `Renderer` class.

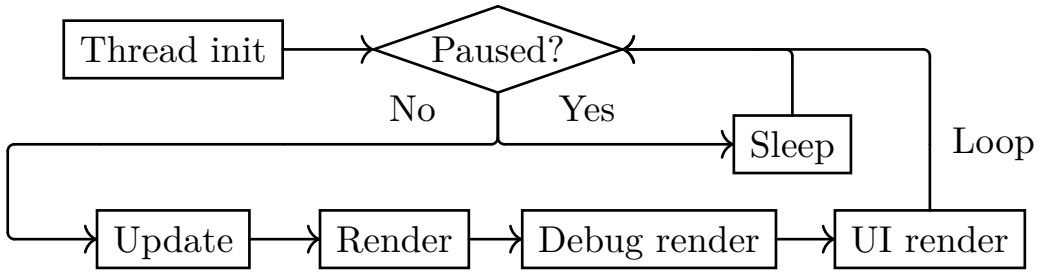


Figure 4: The game loop diagram

All loaded level data is stored inside `LevelData`. It is important that every request for game objects returns a copy of list of those objects ensuring asynchronous list update safety. Loaded engine settings are kept in the `GameSettings`. State of the engine is handled by `GameStateManager`.

The game camera (`Camera` class) handles what should be viewed and therefore rendered at any given time. It has a target (`Player` by default) that it follows and keeps in the center of the screen. It also ensures that the view does not overflow off the map, so the camera may not intentionally move when the player is near the map boundary (especially in a corner).

In addition to controlling the game loop itself, `Renderer` also renders visible game objects. It should also be noted that for objects which are on the edge of the view, only the visible part is rendered. Rendering is based on layering, so objects in a lower layer may be overdrawn by objects in a higher layer (background is drawn first).

Rendering UI elements is delegated to the `UIManager` class. Every UI element has to be registered for rendering inside one of the specified UI regions (or float by its parent game object). UI elements are also layered.

Input is handled by `InputHandler` class. It is registered onto a scene and allows binding specific keys with callback functions to events. It also accepts callbacks for mouse events.

Physics for objects are simulated with the `Collider` class. It can calculate intersection between two objects after one of them moves (`deltaX`, `deltaY`; 0 for immediate intersection). It also calculates distance between the centers of two objects as:

$$\overrightarrow{\mathrm{AB}} = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}$$

Game objects

Every game object inherits the main properties and methods from `GameObject`. Position of game object is relative to the world and has to be converted to screen position if needed. This is intentional because it allows to separate actual render scaling (tile size) and object coordinates. Game object may implement the interface `Interactable`.

```
gameobjects/
├── GameObject.java       # The main class
├── Interactable.java     # Interface
├── tiles/
├── blocks/
├── entities/
├── items/
```

Objects are then divided into `tiles`, `blocks`, `entities` and `items`. Every object then has its respective superclass like `Entity` is for `Player` and `Enemy`.

Tiles are the primary building blocks of the map allowing player to move around based on their `walkable` property.

Blocks are similar to tiles because they can influence navigation on the map by being `solid`. On the other side, they are in a higher layer and they often provide some interactability. They can also have an inventory. Example of a block can be `Chest`.

Entities have the ability to move freely on the map and interact with the environment. They can also have an inventory. The most important is `Player`, being the playable character. Enemies have a very simple algorithm for following player if they are close enough.

Items have the ability to be in two states - dropped on the ground or stored in an inventory. They can also have specific action triggered by using them or by acquiring them. Every item has properties like uses, stack size, price and name defined. Some of those properties are rendered inside the player inventory.

UI

Elements like player stats, inventories and health bars are rendered on top of everything. Every UI element has other game object as its parent and every element has to be registered in `UIManager` to some `UIRegion` to be rendered. Special region is `FLOAT` because it actually is not a region because such elements will have their rendering position influenced by their parent - that is especially useful for enemy health bars.

Utils

These classes assist others in the engine life-cycle. `LevelLoader` handles level data serialization and deserialization, `ResourceManager` loads needed resources and serves as a cache for subsequent requests. There is also `EngineLogger` with custom log formatting and `DebugManager` handling rendering of enabled debug information like game object colliders, enemy search range etc.

Engine logging

The engine logs are both shown in console and saved to a local file `engine.log` inside `JTileEngine` folder. The logs are appended chronologically. Example of such logs are shown below:

```
2025 15.05. [23:32:16] - JTileEngine (EngineLogger.info):
 [INFO] Initializing Engine
2025 15.05. [23:32:16] - JTileEngine (EngineLogger.info):
 [INFO] Loading level: d0c83376-1e08-4a0c-bccd-f0a8a7d0af3
2025 15.05. [23:32:16] - JTileEngine (EngineLogger.info):
 [INFO] Engine paused
2025 15.05. [23:32:16] - JTileEngine (EngineLogger.info):
 [INFO] Engine resumed
2025 15.05. [23:32:23] - JTileEngine (EngineLogger.info):
 [INFO] Shutting down Engine
```

Logging level is customizable in the settings. The default value is `ALL`.

Engine settings file format

The engine settings file is saved locally into the `JTileEngine` folder as `settings.json`. As the file extension suggests, it is stored in JSON format. The format of this file is pretty straightforward:

```
{
    "sound_enabled": Boolean,
    "debug_info": Object,
    "logging": String,
    "tile_size": Integer
}
```

Where:

- `debug_info` is a dictionary of debug flags and their state
- `logging` is a string value of the logging level
- `tile_size` should be ≥ 0

Level file format

Each level is stored inside a `config.json` file in a dedicated folder. The folder name is rather irrelevant, but levels created inside the engine will inherit the levels `UUID` as this name. Level data is also stored in the `JSON` format as follows:

```
{
    "id": String,
    "name": String,
    "thumbnail": String,
    "author": String,
    "builtin": Boolean,
    "completed": Boolean,
    "rows": Integer,
    "cols": Integer,
    "tiles": Array,
    "blocks": Array,
    "entities": Array,
    "created": Integer,
    "updated": Integer
}
```

Where:

- `id` is a string representation of `UUID`
- `rows` & `cols` should be ≥ 0
- `tiles` & `blocks` & `entities` are arrays of objects of their respective types
- `created` & `updated` are integer values representing `Date`