COMP20050 - Software Engineering Project II

libGDX Framework

Ravi Reddy Manumachu
ravi.manumachu@ucd.ie

UCD School of Computer Science.

Scoil na Ríomheolaíochta
UCD.

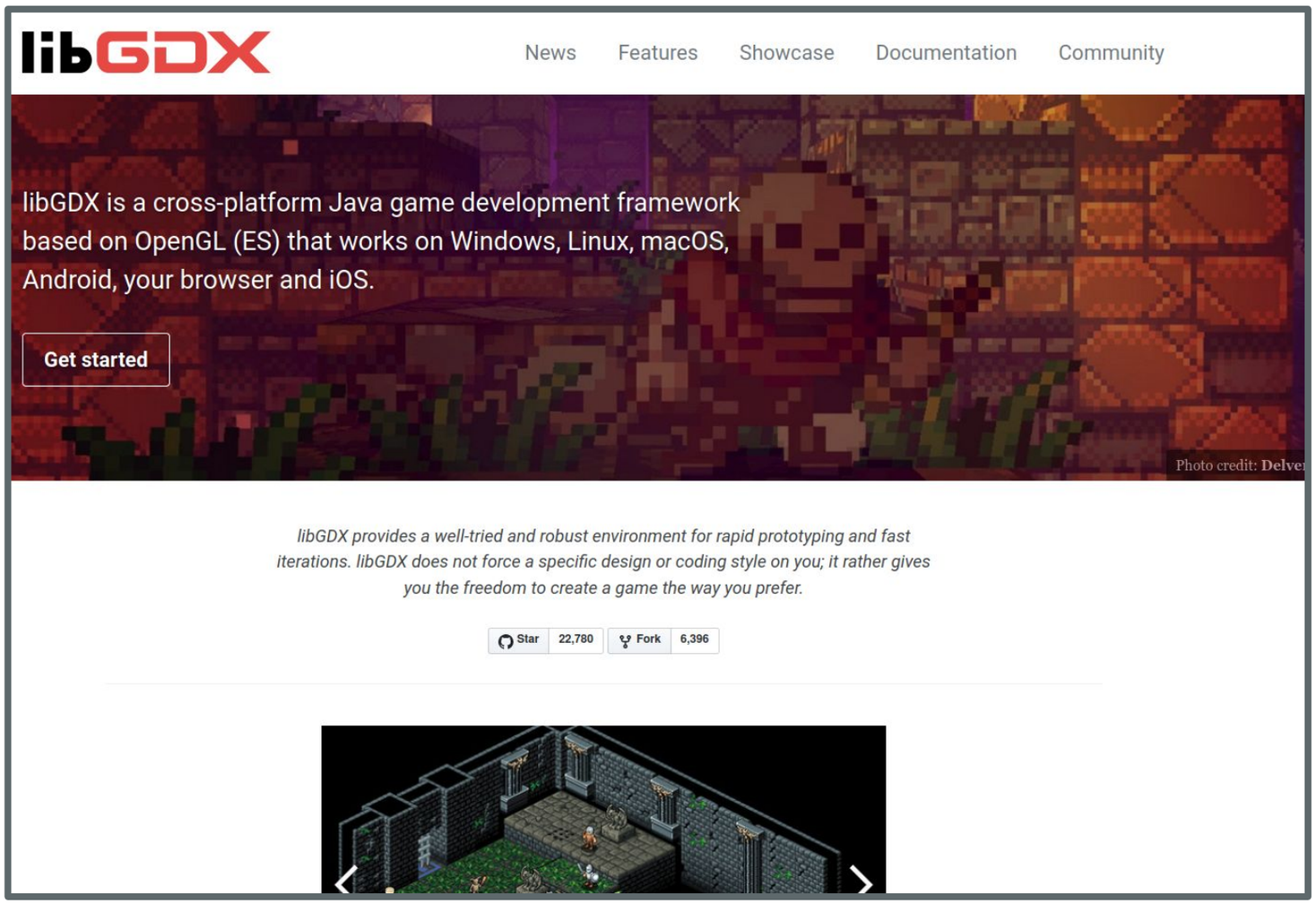
Outline (Learning Objectives)

- Understand the libGDX Features and Basic Tutorial.
- Become familiar with the libGDX Components.
- Understand the libGDX Application Life Cycle.

What is libGDX?

https://libgdx.com/

- **libGDX** is a cross-platform Java game development framework based on OpenGL (ES) that works on Windows, Linux, macOS, Android, your browser and iOS.



libGDX Features (1/3)

- **libGDX** offers a single API to target: Windows, Linux, macOS, Android, iOS and Web.
- **Rendering** is handled on all platforms through **OpenGL ES 2.0/3.0**.
- **libGDX** offers a very extensive third-party ecosystem. https://libgdx.com/dev/tools/
- **libGDX** is **open source** and is licensed under **Apache 2.0**, offering unrestricted usage in both commercial and non-commercial projects. https://github.com/libgdx/libgdx

libGDX Features (2/3)

https://libgdx.com/features/

- **Audio**: Streaming music and sound effect playback for **WAV**, **MP3** and **OGG**.
- **Input Handling**: Abstractions for mouse on the desktop/browser, touch screens on Android and keyboards.
- **Math and Physics**:
  - **Matrix**, **vector** and **quaternion** classes accelerated via native C code where possible.
  - libGDX has several geometric classes for dealing with shapes, areas, and volumes. These include: **Circle**, **Frustum**, **Plane**, **Spline**, **Polygon**, **Ray**, and **Rectangle**.
  - 2D and 3D physics.

libGDX Features (3/3)

https://libgdx.com/features/

- **File I/O and Storage**: File system abstraction for all platforms.

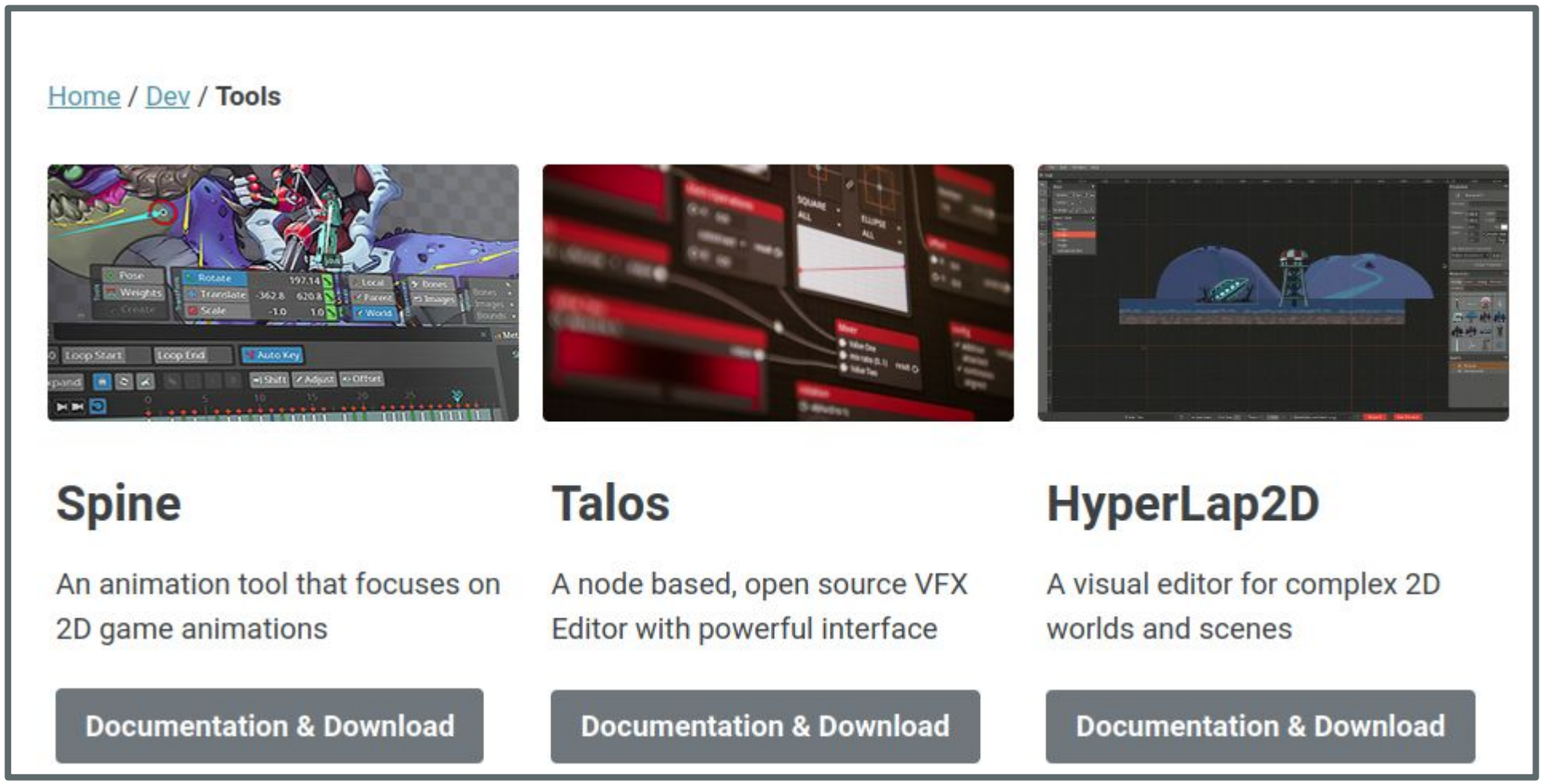
- **Graphics**:
  - Low-level OpenGL helpers, such as **Vertex arrays** and vertex buffer objects, **Meshes**, and **Textures**.
  - High-level 2D and 3D APIs.

- **Networking**: Gdx.net for simple networking (TCP sockets and HTTP requests).

LibGDX Tools

https://libgdx.com/dev/tools/

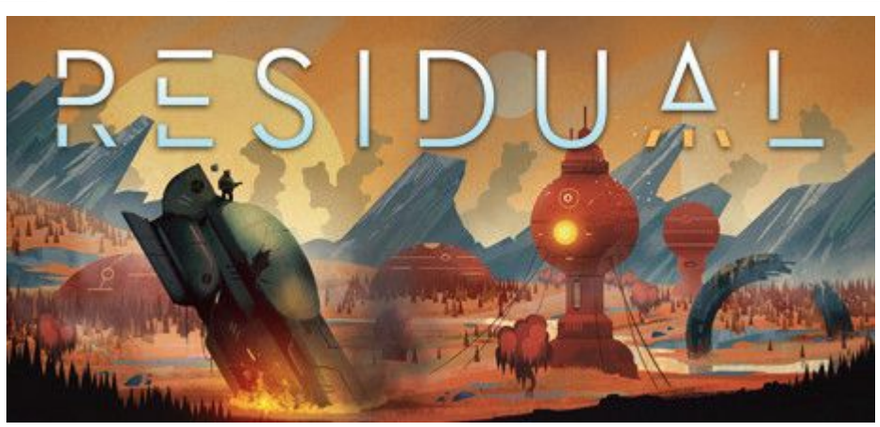
- Both official and community-made – that can help make the development process for libGDX much easier.



LibGDX Showcase

https://libgdx.com/showcase/

- Collection of games built with libGDX.



libGDX Get Started

libGDX Wiki

https://libgdx.com/wiki/

- **libGDX Wiki** is a place for comprehensive documentation on the libGDX API and features.

libGDX: Set Up a Dev Environment

https://libgdx.com/wiki/start/setup

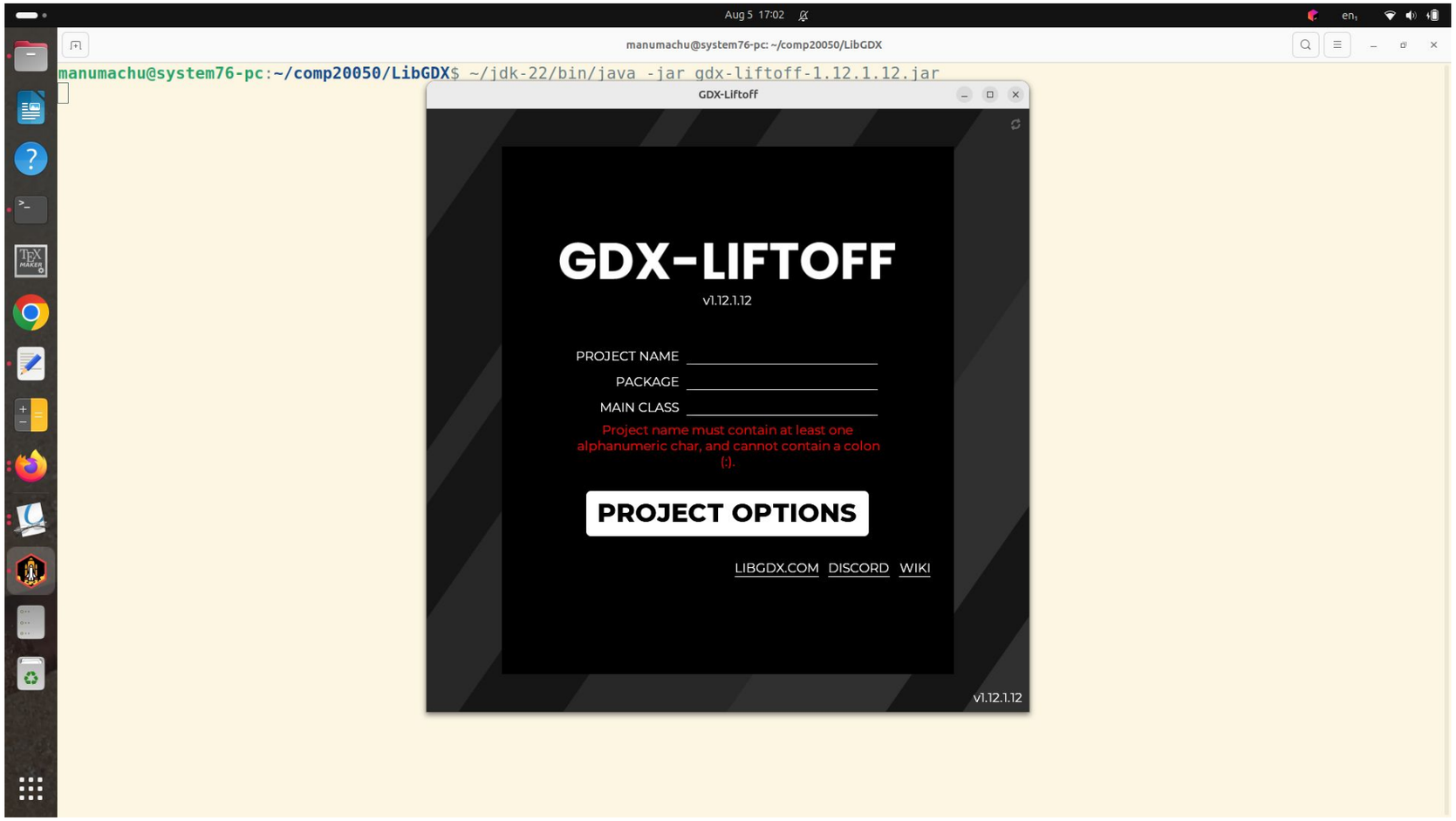
- Before you can get started with **libGDX**, you need to set up an **IDE** (Integrated Development Environment).
  - Android Studio.
  - IntelliJ IDEA (Community Edition).
  - Eclipse.
  - No IDE.
    - ❖ Use simple editors such as Notepad or Vim.
    - ❖ libGDX applications are Gradle applications that can be built and executed via the command line.

libGDX: Generate a Project (1/6)

https://libgdx.com/wiki/start/project-generation

- Download the libGDX Project Setup Tool (**gdx-liftoff**).
- Run the command:

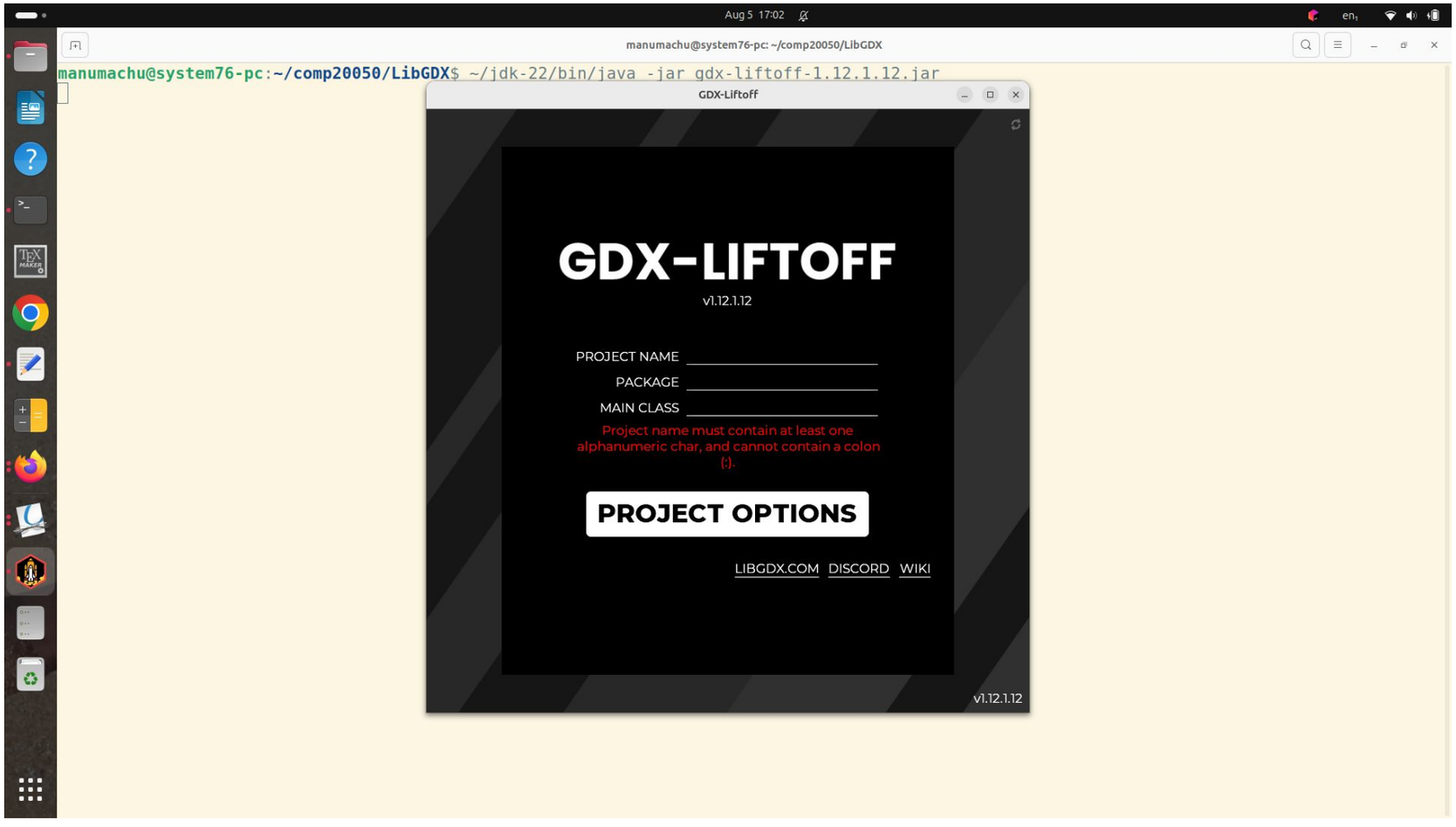
java -jar gdx-liftoff-x.x.x.x.jar



libGDX: Generate a Project (2/6)

- You are asked to provide the following:
  - **PROJECT NAME:** the name of the application.
  - **PACKAGE:** the Java package under which your code will reside.
  - **MAIN CLASS:** the name of the main game Java class of your app.

- Click **PROJECT OPTIONS.**



libGDX: Generate a Project (3/6)

- Clicking **PROJECT OPTIONS** takes you to the **Add-Ons** screen.

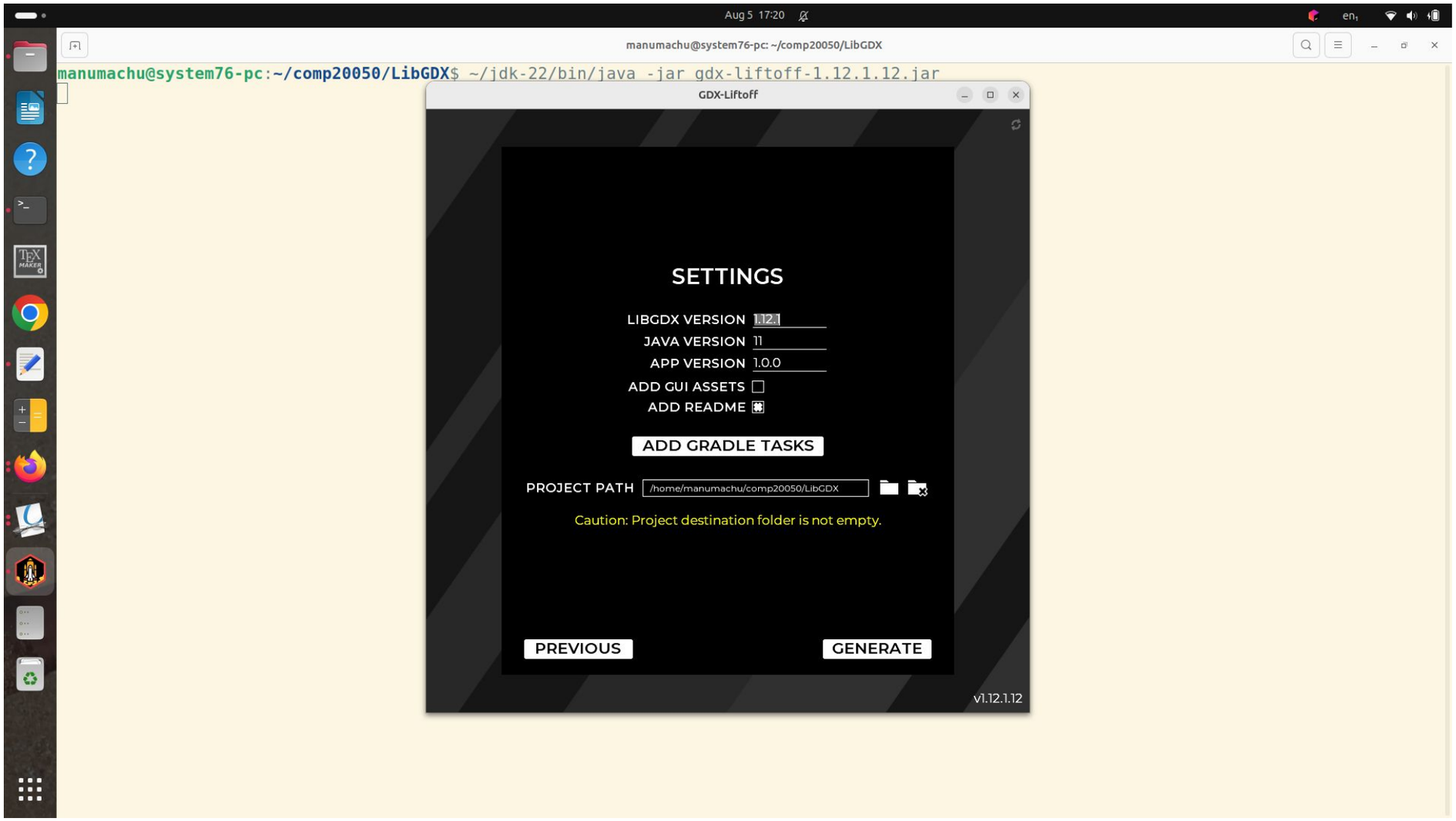
libGDX: Generate a Project (4/6)

- **Add-Ons:**
  - **Platforms:** The backends that your project will support. You will use Desktop.
  - **Languages:** The languages besides Java that you want to include in the project (Groovy, Kotlin, Scala).
  - **Extensions:** Officially supported add-ons that extend the functionality of libGDX.

libGDX: Generate a Project (5/6)

- Proceeding to the next screen takes you to the Third-Party screen. These are additional extensions that are not provided by the official libGDX maintainers.

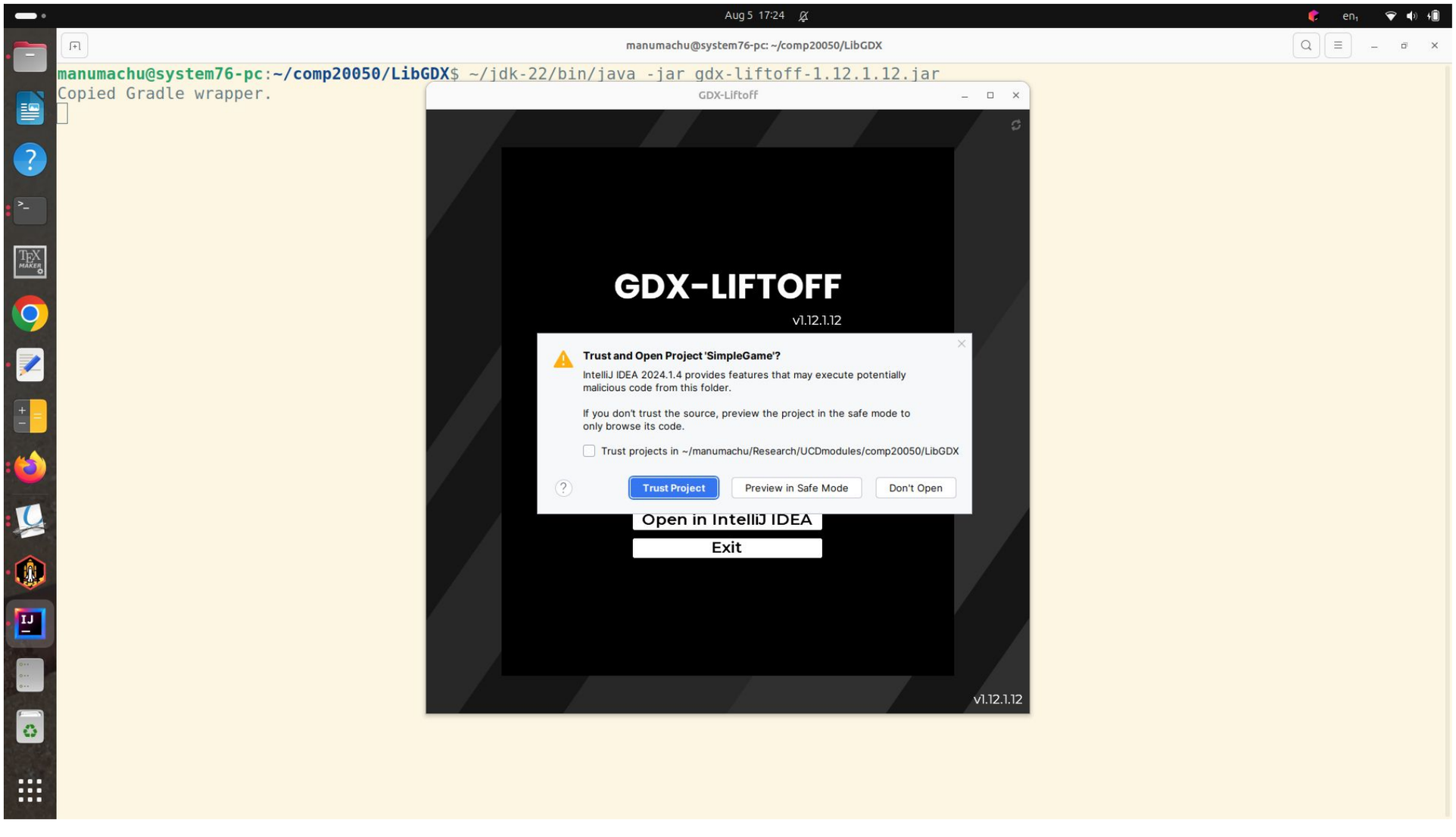
- The final screen allows you to set versions and other options.



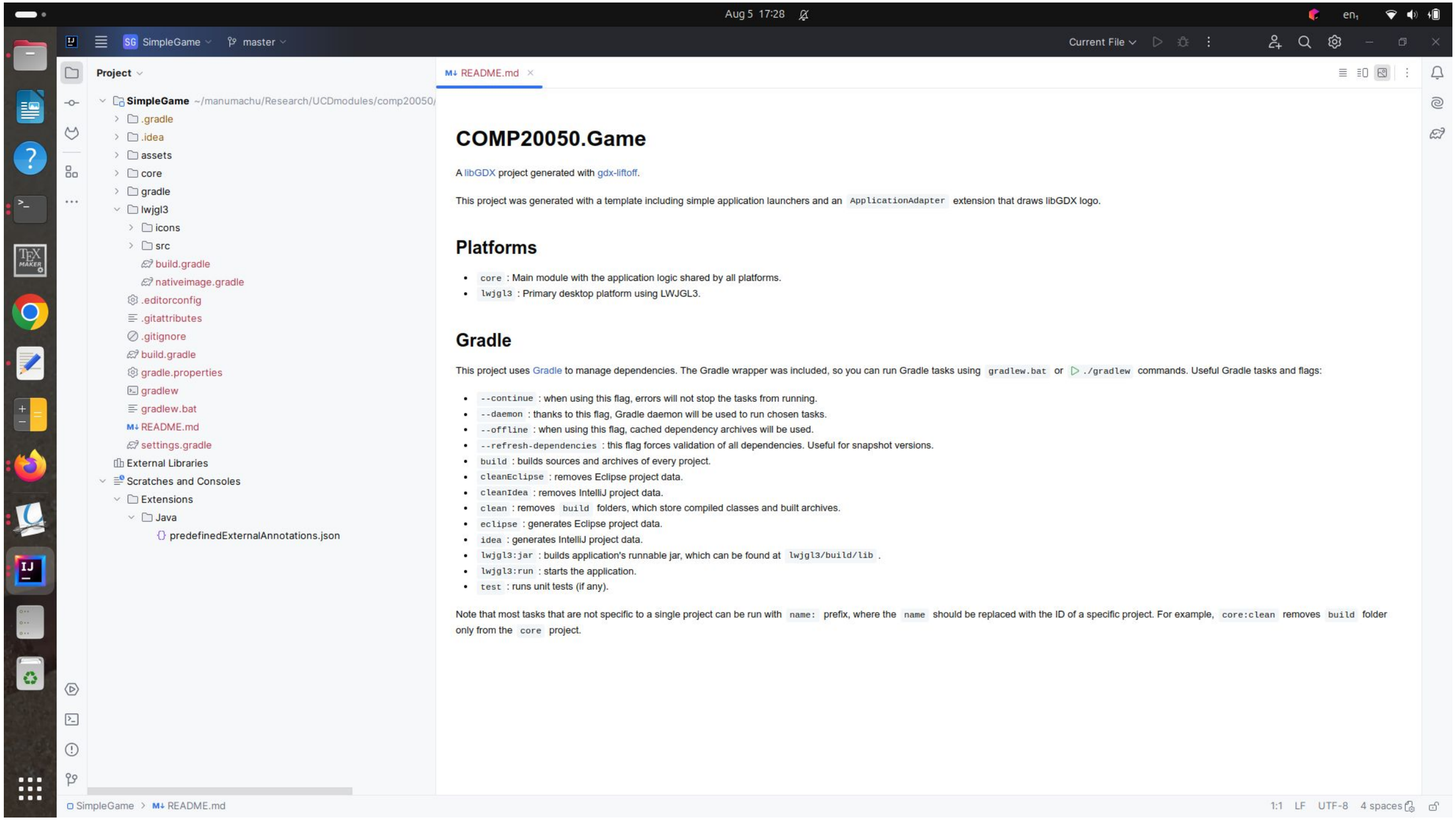
libGDX: Generate a Project (6/6)

- After you click **generate**, you will be presented with a project summary screen.

- You can **open** your project directly in IntelliJ IDEA if you have it installed.

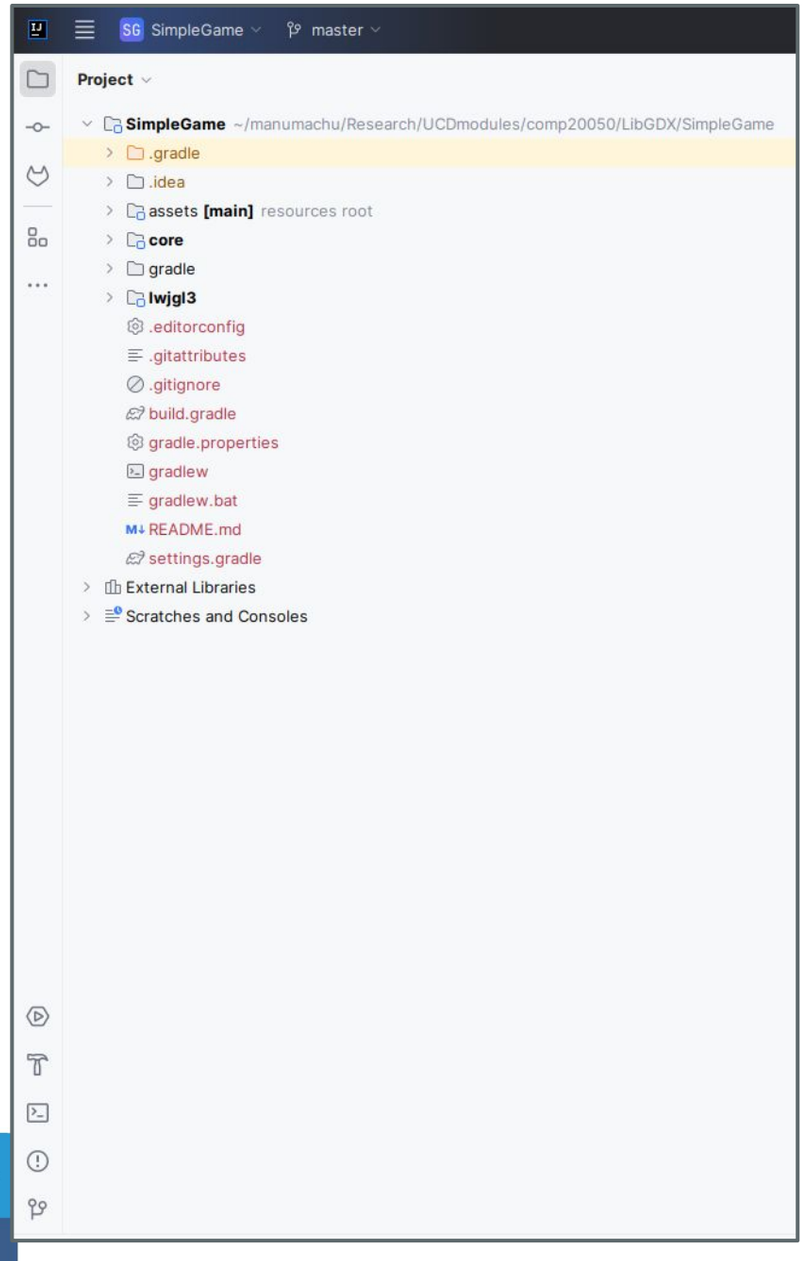


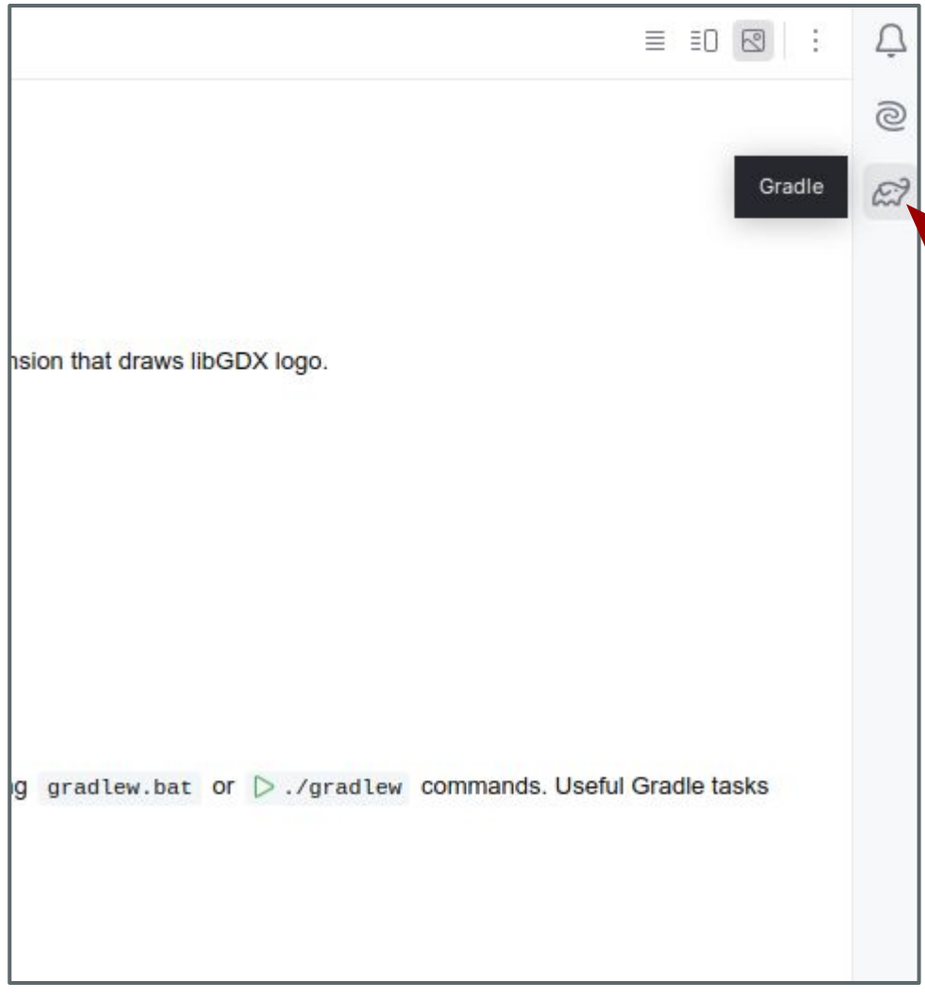
libGDX Project in IntelliJ IDEA



libGDX: Running in IntelliJ IDEA

- Extend the **gradle** tab on the right.

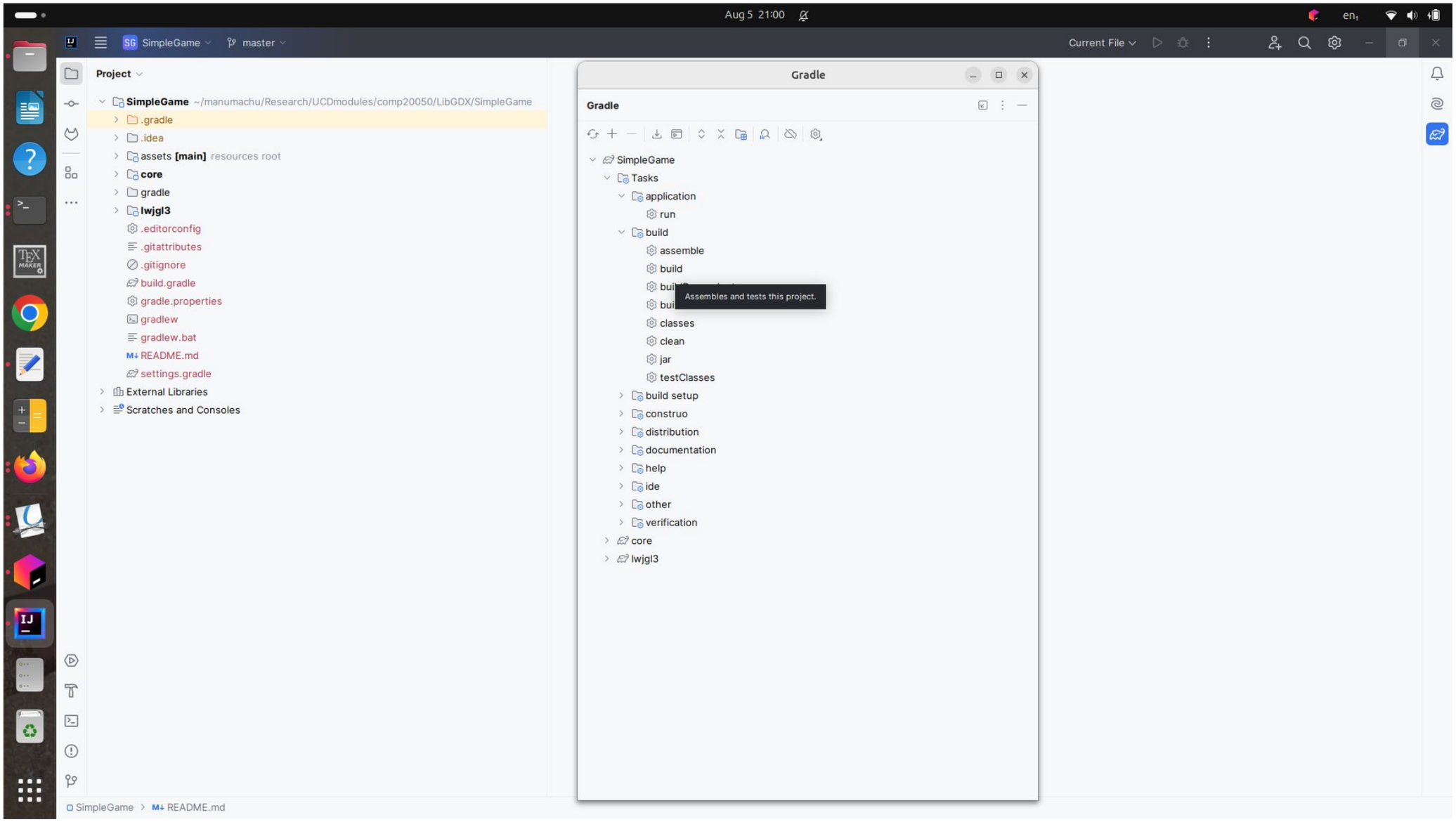




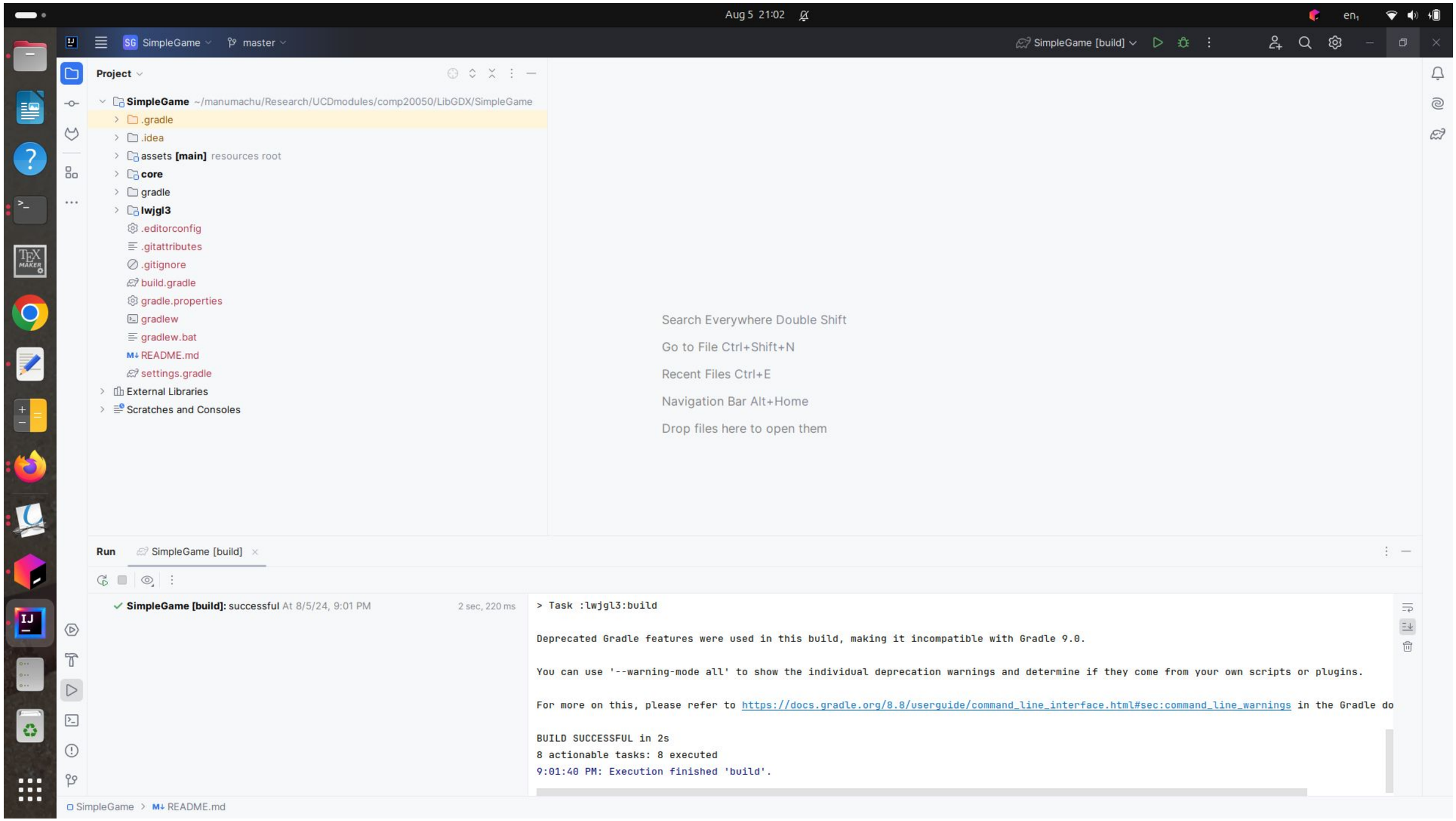
Gradle Tab

libGDX: Building App in IntelliJ IDEA

- Double click **build** to build the application.

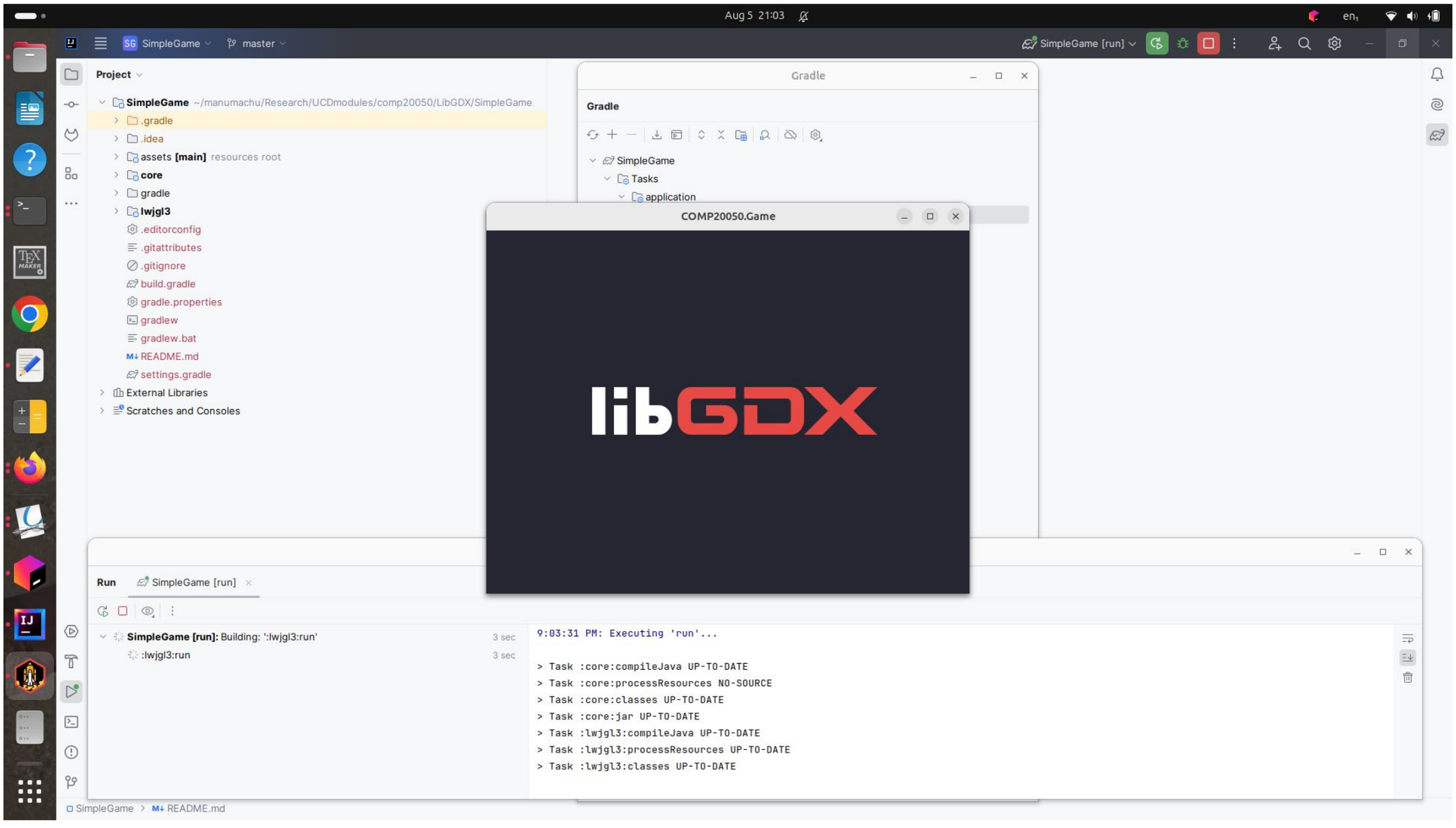


libGDX: Building App in IntelliJ IDEA

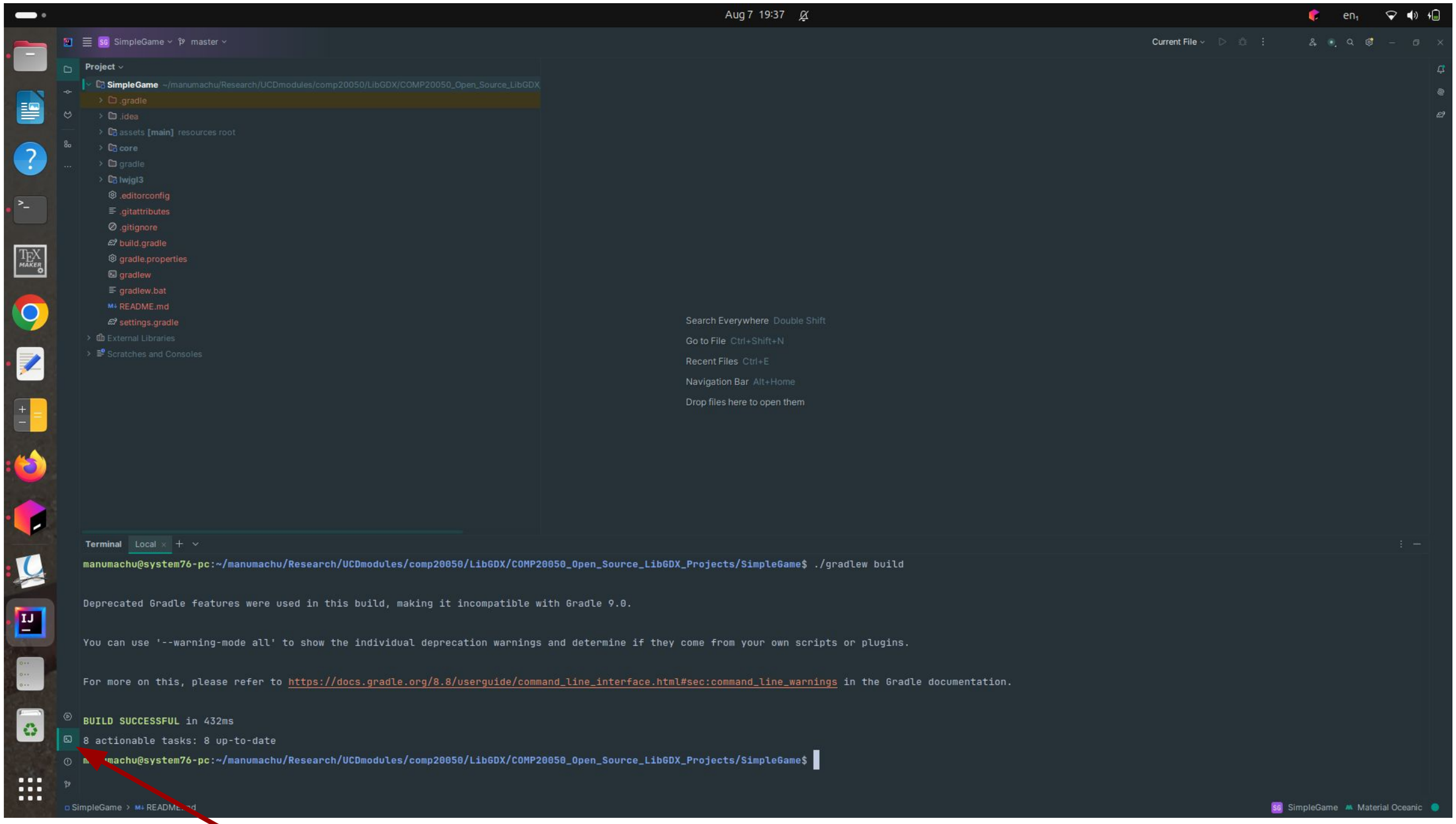


libGDX: Running App in IntelliJ IDEA

- Double click **run** to run the application.



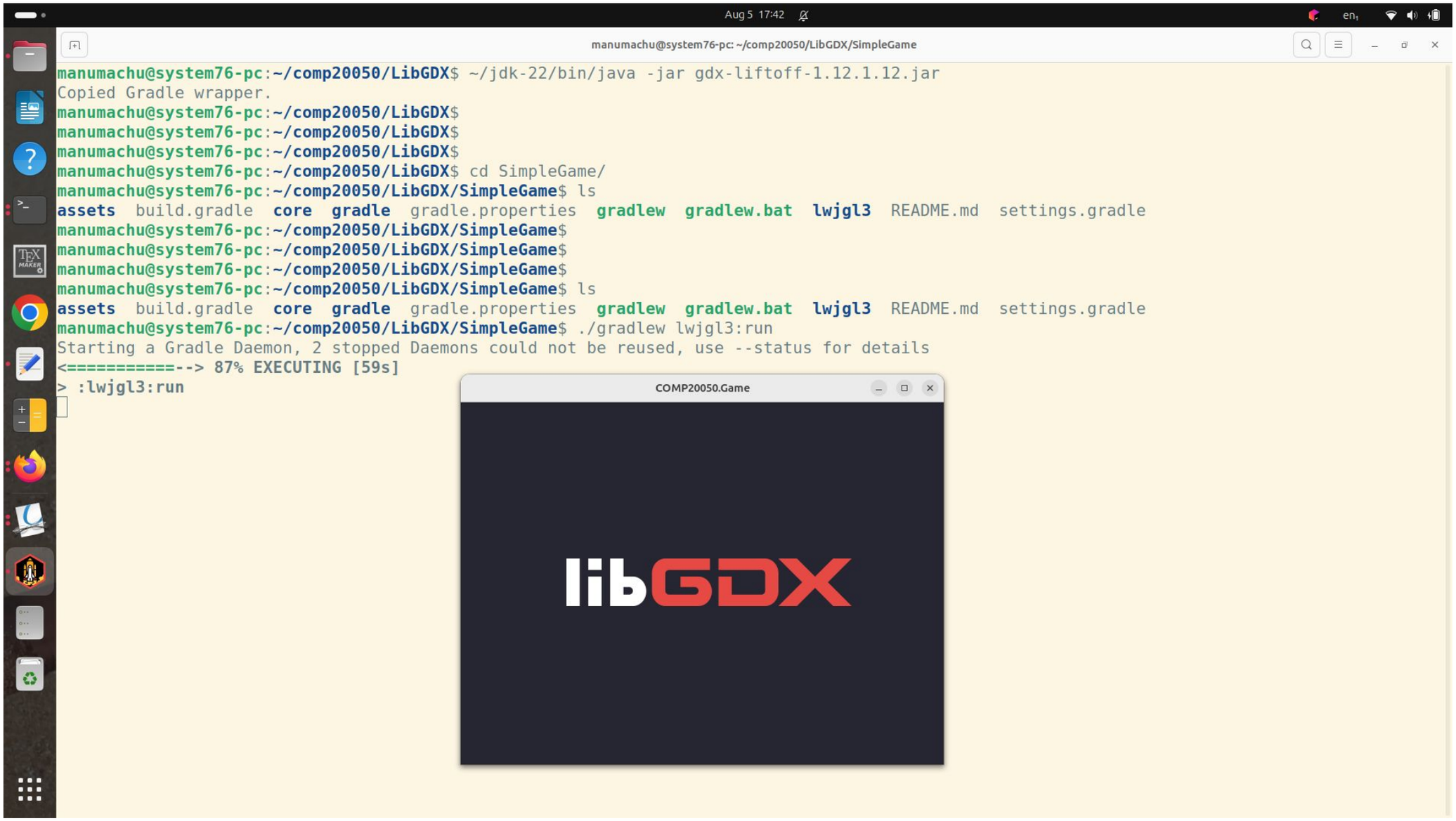
libGDX: Building and Running in Terminal



Terminal Tab

libGDX: Building and Running in Command Line

shell$ ./gradlew build
shell$ ./gradlew lwjgl3:run



Basics of Game Architecture

Game Framework Vs Game Engine

- A **game engine** is a complete development environment that provides all the tools and features needed to create and run a game.
- It is an all-in-one solution for game development.
- **Examples:** Unity, Unreal Engine, Godot, CryEngine.

CRYENGINE®

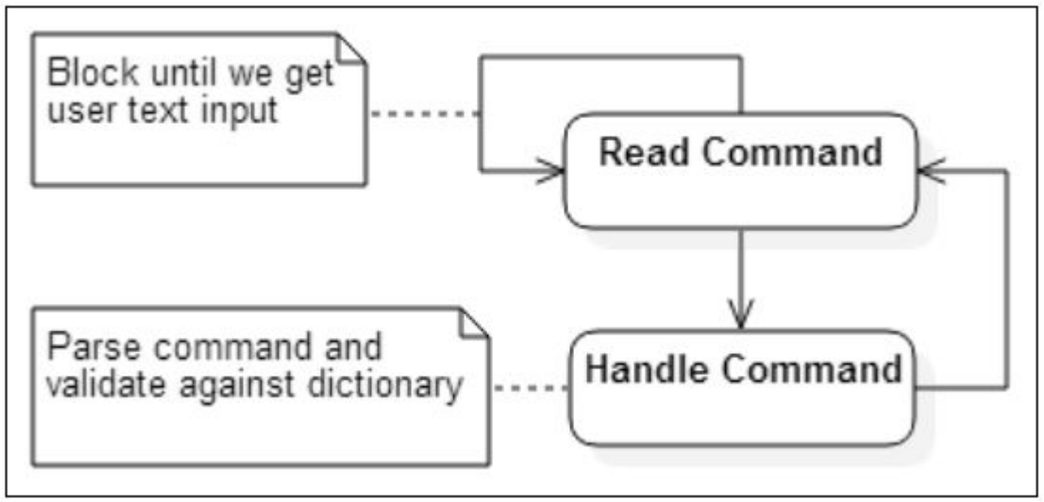
Game Framework

- A **game framework** is a collection of libraries with exposed application program interfaces (APIs) into **modules** that help developers create games.
- It does not come with a full-suite of pre-built tools or editors.
- **Examples:** Monogame, SFML, Phaser, libGDX.

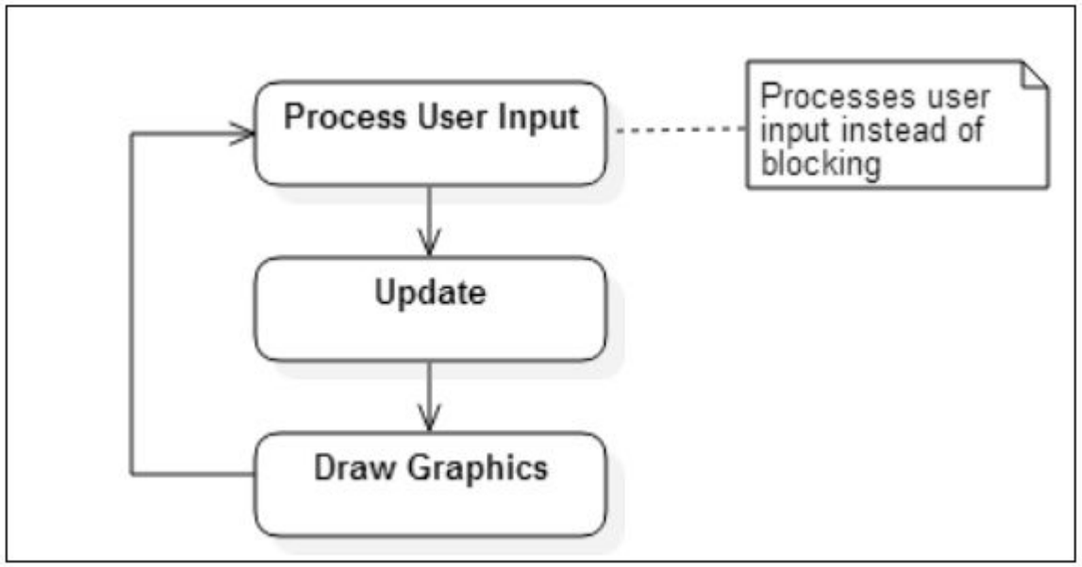
Game Framework Vs Engine: Key Differences

Key Differences		
Feature	**Game Engine**	**Game Framework**
Pre-Built Tools	Yes (editors, asset management, etc.)	No (only libraries and APIs)
Ease of Use	Easier for beginners	Requires more programming expertise
Flexibility	Less flexible (opinionated design)	Highly flexible
Development Speed	Faster (due to pre-built systems)	Slower (more manual work required)
Examples	Unity, Unreal Engine	MonoGame, SDL, Phaser

Text-based Vs Video Game Loops (1/2)



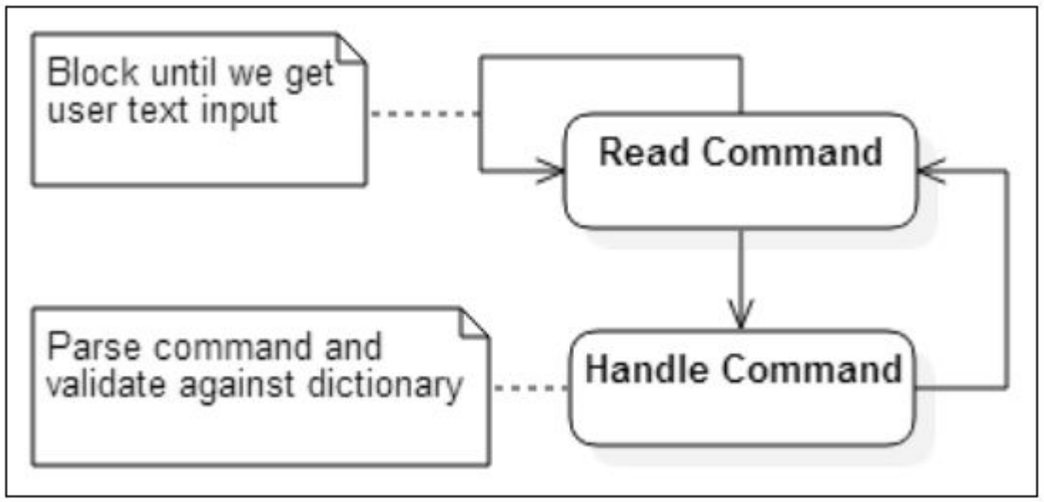
Text-based Game Loop



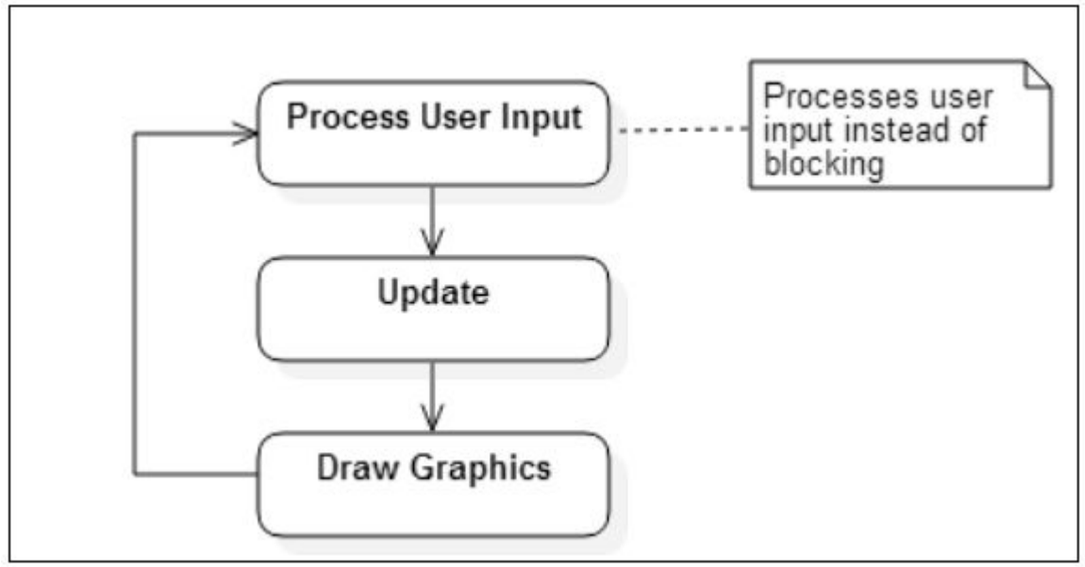
Graphics-based Video Game Loop

- In **text-based games**, the game would block all updates until it receives user-based text input from the command-line.
- The game would process the user input string, break it up into word chunks, and compare against its verb and noun dictionaries (key-value mappings).
- The difficulty with a text-driven system are the complexities associated with equally valid variations of an English sentence.

Text-based Vs Video Game Loops (2/2)



Text-based Game Loop



Graphics-based Video Game Loop

- However, in **graphics-based games**, there is always something that needs to be updated every cycle in the loop even if the player is idle.
- Instead of waiting for user input, a game loop **polls** for events, processing all user input available at that time.

The Life Cycle of a Video Game (1/3)

- **Startup:** Any files such as images or sounds are loaded, game objects are created, and values are initialized.

- **Game Loop:** Repeats continuously while the game is running
- and that consists of the following three sub-stages:
  - **Process Input:** The program checks to see if the user has performed any action
    - Pressing keyboard keys.
    - Moving the mouse or clicking mouse buttons.

The Life Cycle of a Video Game (2/3)

- **Game Loop:** Repeats continuously while the game is running and that consists of the following three sub-stages:
  - **Update:**
    - Performs tasks that involve the state of the game world and the entities within it.
    - This could include changing the positions of entities based on user input, performing collision detection or selecting actions for nonplayer characters.
  - **Render:**
    - Draws all graphics on the screen, such as background images, game-world entities, and the user interface.

The Life Cycle of a Video Game (3/3)

- **Shutdown:**
  - The player indicates intent to **Quit** the game application, which may involve removing images or data from memory.
  - Saving player data or the game state.
  - Closing any windows that were created by the game.

Frame Rate (1/2)

- **One cycle of the game loop** is generally referred to as a **frame**.
- **Frame rate** measured in frames per second (FPS) is the number of cycles that can be completed in a fixed amount of time.
- The **higher** the FPS, the better the **perceived experience** will be for the player.
- The game will feel more responsive, there will be better collision detection and enemy movement, and the graphics rendering will be much smoother.
- The **lower** the FPS, the more **degraded** the game experience will be for the player.
- In modern games, a frame rate of **30 FPS** is standard for a good gameplay experience.

Frame Rate (2/2)

- Two factors affect the frame rate.
- The **processing speed of the CPU** and whether there is a **dedicated GPU** (graphics processing unit) for rendering.
- How much logic there is to process in each frame?
  - Calculations for physics (collision detection) and rendering high-fidelity graphics for lots of game objects can lead to a frame taking longer to render.
  - Therefore, fewer frames are completed every second.

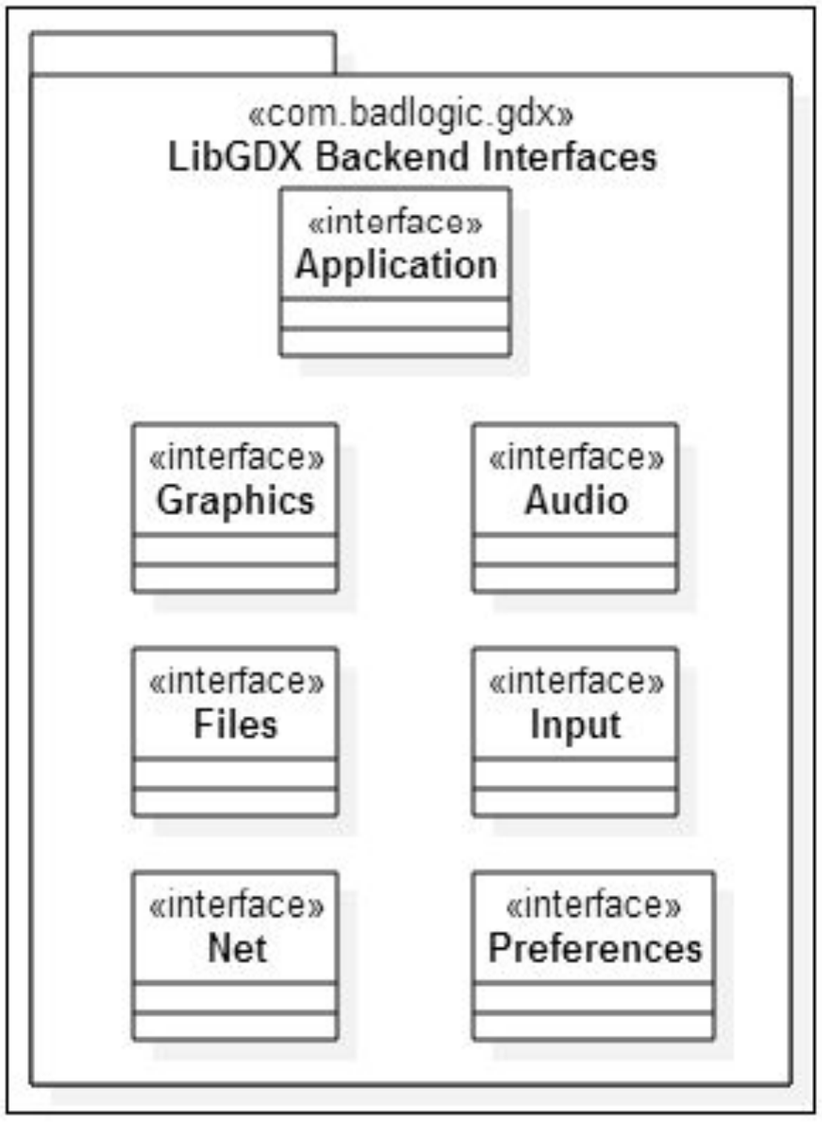
libGDX deltaTime

- The typical, brute force solution for dealing with the factors that affect the frame rate is to lock the frame rate.
- However, this is not an optimal solution.
- **LibGDX** addresses the problem of varying frame rates depending on the device, by passing in a **deltaTime** value during each render call for a frame.
- **deltaTime** is the total time in seconds that the game took to render the last frame.
- By updating calculations using the **deltaTime** value, the gameplay elements should be synchronized running more consistently across the different devices.

libGDX High-Level Components

libGDX backend modules (1/4)

- **libGDX** consists of **six modules**, which are the high level abstractions that provide most of the functionality you need to create your game.
- These interfaces are implemented for each of the currently supported target platforms (backends).



libGDX backend modules (2/4)

- **Application:**
  - The entry point that the platform OS uses to load your game.
  - Responsible for setting up a window, handling resize events, rendering to the surfaces, and managing the application during its lifetime.
  - Provides logging facilities and querying methods such as memory usage.

- **Graphics:**
  - Contains helper methods for communicating with the platform's graphics processor.
  - Such as rendering to the screen and querying for available display modes such as graphics resolution and color depth.

libGDX backend modules (3/4)

- **Audio:**
  - Contains helper methods for creating and managing various audio resources.
  - Helps to create sound effects, play music streams, and give direct access to the audio hardware for PCM audio input and output.

- **Files:**
  - Contains helper methods for accessing the platform's file system when managing game assets such as reading and writing files.

- **Input:**
  - Contains helper methods to poll (or process events) for user input from keyboard key presses and mouse button clicks.

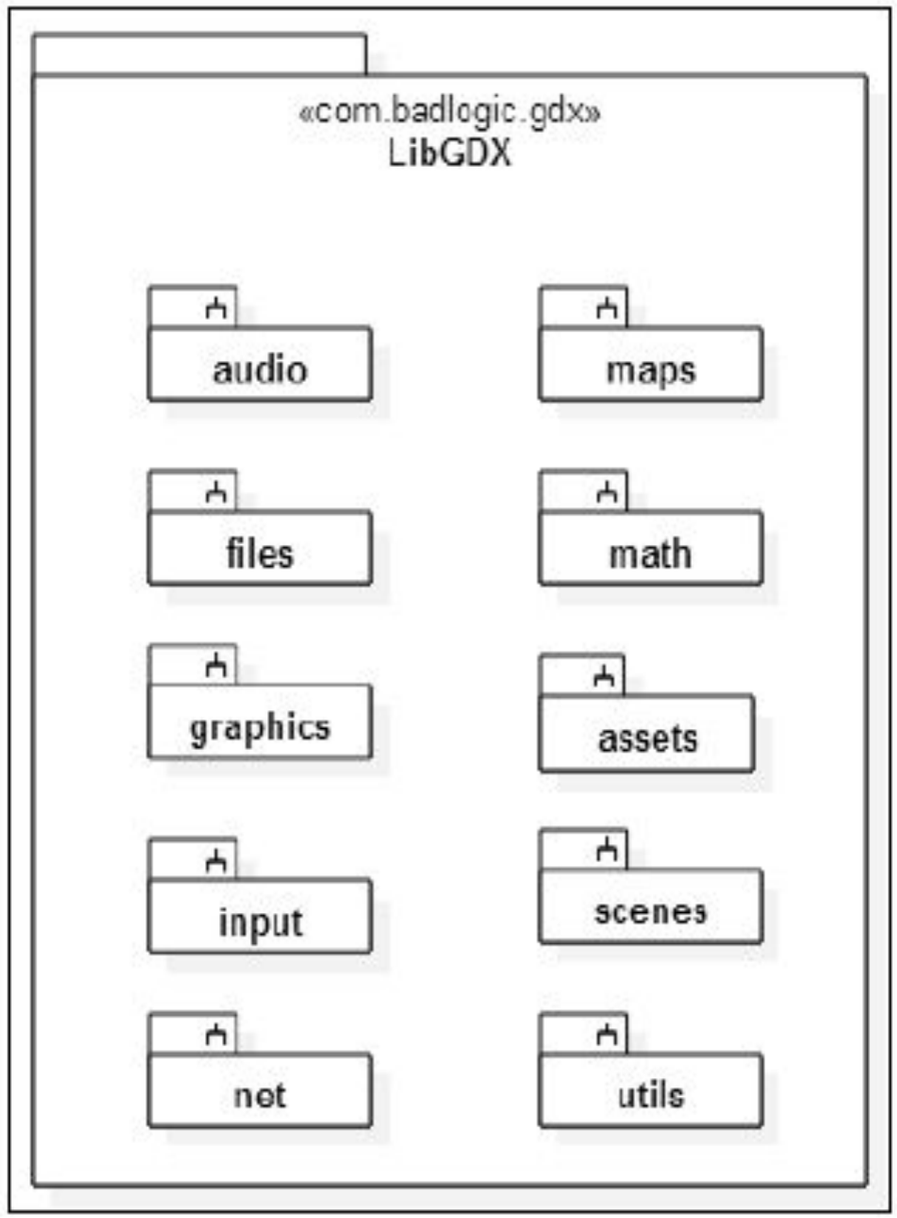
libGDX backend modules (4/4)

- **Net:**
  - Contains numerous helper methods for performing certain network-related operations.
  - Managing HTTP/HTTPS GET and POST requests and creating TCP server/client socket connections.

- **Preferences:**
  - Contains helper methods for storing and accessing application game setting values as a lightweight setting storage mechanism.

libGDX Core Modules (1/3)

- **libGDX** consists of some other core modules implemented within the framework (and not platform-specific).



libGDX Core Modules (2/3)

- **Maps:**
  - Contains classes for dealing with different level map implementations, such as maps generated from **Tiled** (an XML-based format called TMX).

- **Math:**
  - Contains classes (with utility methods) for dealing with various mathematical calculations such as trigonometry, linear algebra, and probability.
  - Geometric classes for dealing with shapes, areas, and volumes.
  - Collision detection tests such as intersection and overlap, and interpolation algorithms.

libGDX Core Modules (3/3)

- **Assets:**
  - Contains classes for managing the loading and storing of assets such as textures, bitmap fonts, particle effects, pixmaps, UI skins, tile maps, sounds, and music.

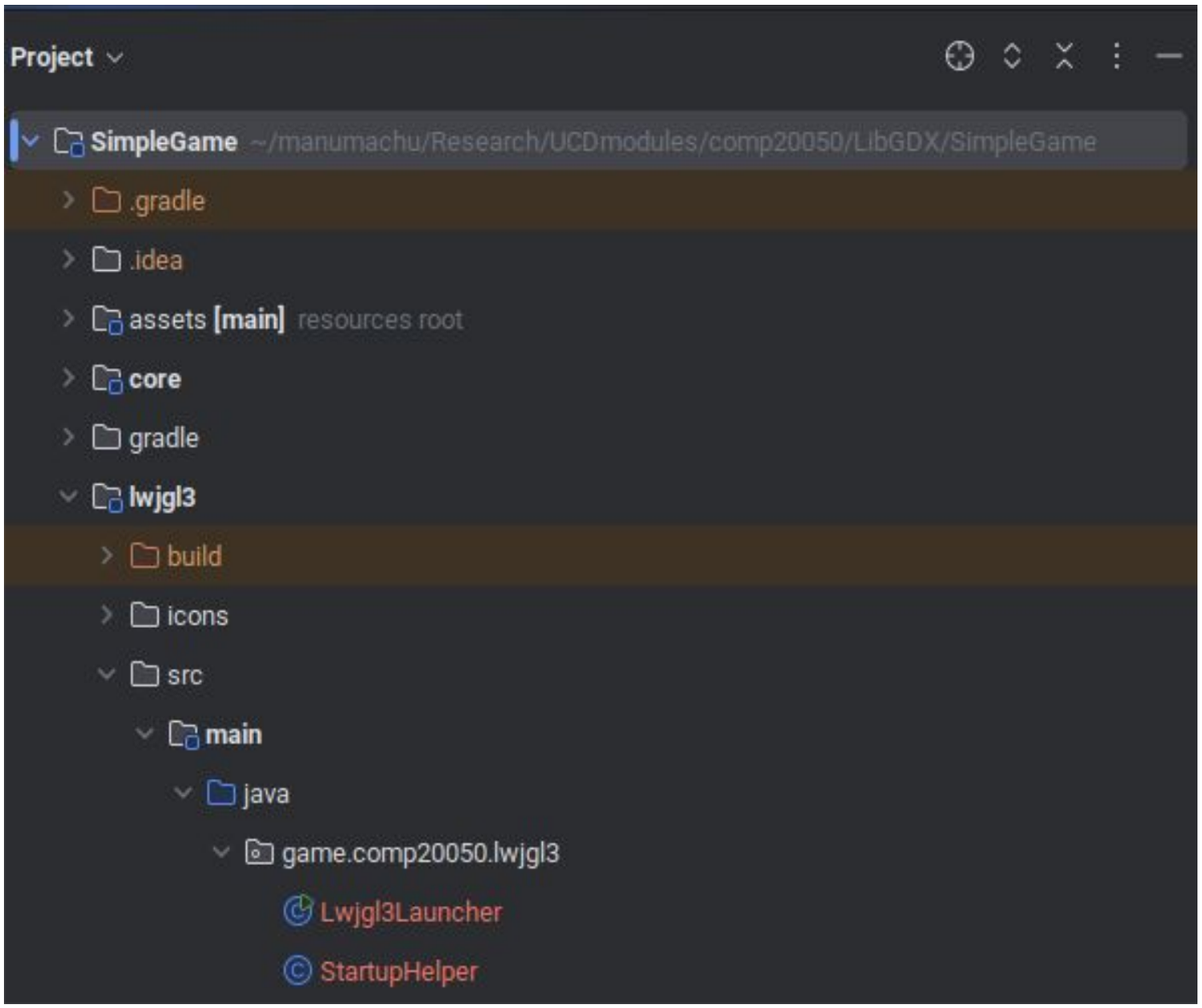
- **Scenes:**
  - Contains classes for building 2D scene graphs used in creating UIs such as game menus.

- **Utils:**
  - Supports reading and writing in XML and JSON.
  - Timers and object pools.

libGDX Application Lifecycle

Starter Classes: Lwjgl3Launcher.java

lwjgl3 -> src -> main -> java -> game.comp20050.lwjgl3 -> Lwjgl3Launcher.java



Starter Classes: Lwjgl3Launcher.java

lwjgl3 -> src -> main -> java -> game.comp20050.lwjgl3 -> Lwjgl3Launcher.java

Lwjgl3Launcher.java

```java
package game.comp20050.lwjgl3;

import com.badlogic.gdx.backends.lwjgl3.Lwjgl3Application;
import com.badlogic.gdx.backends.lwjgl3.Lwjgl3ApplicationConfiguration;
import game.comp20050.SimpleGame;

/** Launches the desktop (LWJGL3) application. */
public class Lwjgl3Launcher {
    public static void main(String[] args) {
        if (StartupHelper.startNewJvmIfRequired()) return; // This handles macOS support and helps on Windows.
        createApplication();
    }

    private static Lwjgl3Application createApplication() {  1 usage
        return new Lwjgl3Application(new SimpleGame(), getDefaultConfiguration());
    }

    private static Lwjgl3ApplicationConfiguration getDefaultConfiguration() {  1 usage
        Lwjgl3ApplicationConfiguration configuration = new Lwjgl3ApplicationConfiguration();
        configuration.setTitle("COMP20050.Game");
        configuration.useVsync(true);
        //// Limits FPS to the refresh rate of the currently active monitor.
        configuration.setForegroundFPS(Lwjgl3ApplicationConfiguration.getDisplayMode().refreshRate);
        //// If you remove the above line and set Vsync to false, you can get unlimited FPS, which can be
        //// useful for testing performance, but can also be very stressful to some hardware.
        //// You may also need to configure GPU drivers to fully disable Vsync; this can cause screen tearing.
        configuration.setWindowedMode( width: 640,  height: 480);
        configuration.setWindowIcon("libgdx128.png", "libgdx64.png", "libgdx32.png", "libgdx16.png");
        return configuration;
    }
}
```

Starter Classes: Desktop (LWJGL3)

- For each target platform, a starter class has to be written.
- This class instantiates a back-end specific **Application** implementation and the **ApplicationListener** that implements the application logic.

Starter Classes: Desktop (LWJGL3)

```
private static Lwjgl3ApplicationConfiguration getDefaultConfiguration() {  1 usage
    Lwjgl3ApplicationConfiguration configuration = new Lwjgl3ApplicationConfiguration();
    configuration.setTitle("COMP20050.Game");
    configuration.useVsync(true);
    //// Limits FPS to the refresh rate of the currently active monitor.
    configuration.setForegroundFPS(Lwjgl3ApplicationConfiguration.getDisplayMode().refreshRate);
    //// If you remove the above line and set Vsync to false, you can get unlimited FPS, which can be
    //// useful for testing performance, but can also be very stressful to some hardware.
    //// You may also need to configure GPU drivers to fully disable Vsync; this can cause screen tearing.
    configuration.setWindowedMode( width: 640,  height: 480);
    configuration.setWindowIcon("libgdx128.png", "libgdx64.png", "libgdx32.png", "libgdx16.png");
    return configuration;
}
```

- First an **Lwjgl3ApplicationConfiguration** is instantiated.
- This class lets one specify various configuration settings, such as the initial **screen resolution**, whether to use **OpenGL ES 2.0** or **3.0**.

Starter Classes: Desktop (LWJGL3)

```
private static Lwjgl3Application createApplication() {  1 usage
    return new Lwjgl3Application(new SimpleGame(), getDefaultConfiguration());
}
```

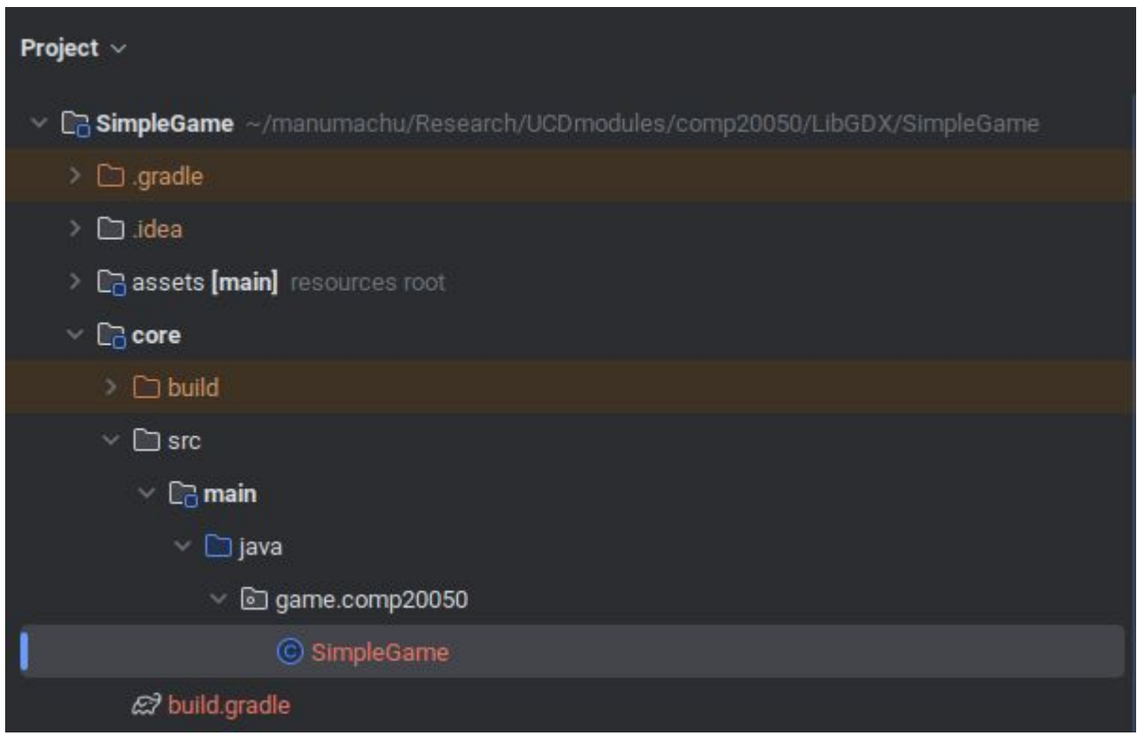
- Once the configuration object is set, an **Lwjgl3Application** is instantiated.
- The **SimpleGame()** class is the **ApplicationListener** implementing the game logic.
- From there on a window is created and the **ApplicationListener** is invoked as described in The Life Cycle.

The Life Cycle (1/4)

```java
public class MyGame implements ApplicationListener {
    public void create () {
    }

    public void render () {
    }

    public void resize (int width, int height) {
    }

    public void pause () {
    }

    public void resume () {
    }

    public void dispose () {
    }
}
```

- A libGDX application has a well defined life-cycle, governing the states of an application, like **creating**, **pausing** and **resuming**, **rendering** and **disposing** the application.

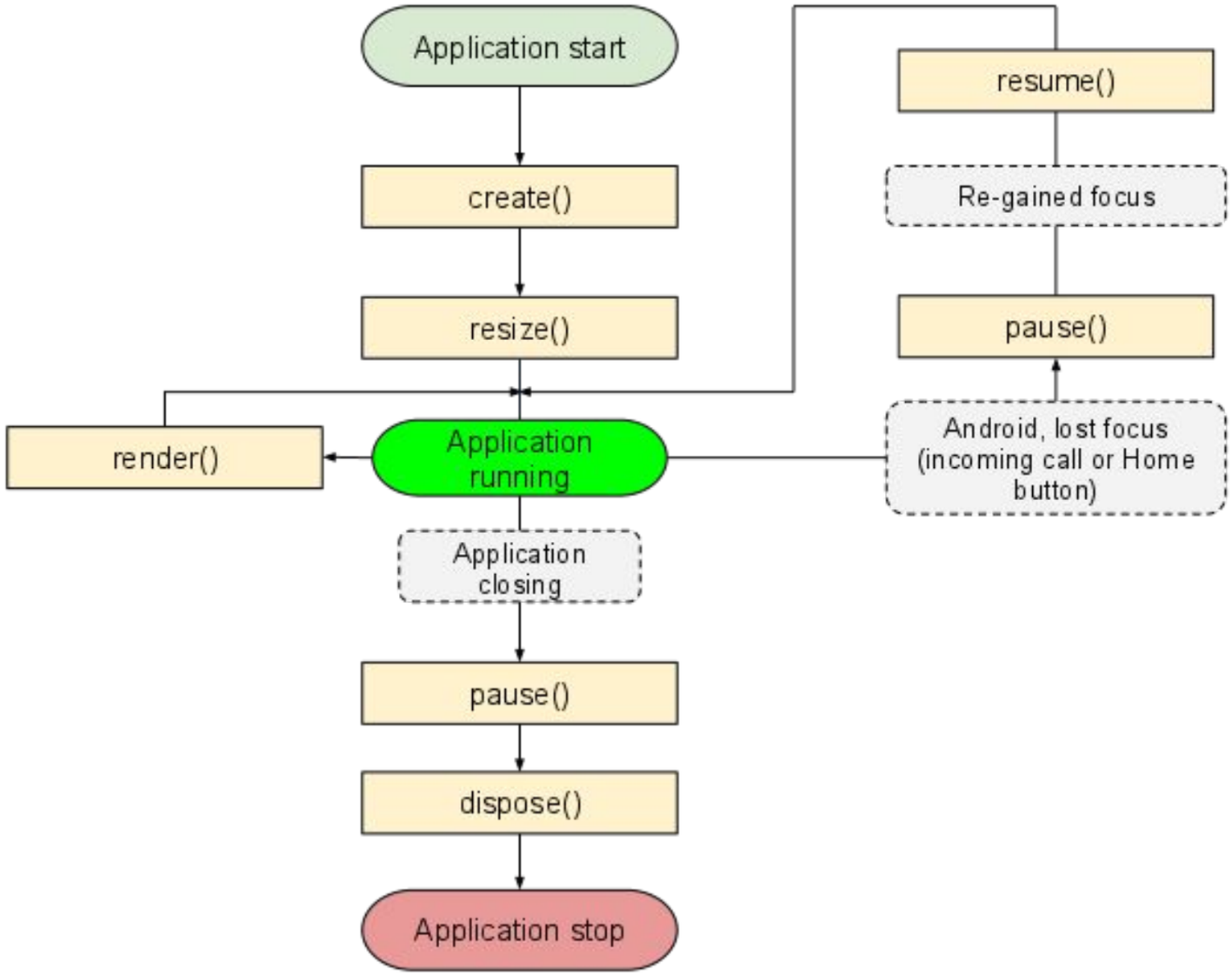
SimpleGame Application Listener




```java
/** {@link com.badlogic.gdx.ApplicationListener} implementation shared by all platforms. */
public class SimpleGame extends ApplicationAdapter {  2 usages
    private SpriteBatch batch;  5 usages
    private Texture image;  3 usages

    @Override
    public void create() {
        batch = new SpriteBatch();
        image = new Texture( internalPath: "libgdx.png");
    }

    @Override
    public void render() {
        ScreenUtils.clear( r: 0.15f,  g: 0.15f,  b: 0.2f,  a: 1f);
        batch.begin();
        batch.draw(image,  x: 140,  y: 210);
        batch.end();
    }

    @Override
    public void dispose() {
        batch.dispose();
        image.dispose();
    }
}
```

SimpleGame -> core -> src -> main -> java -> game.comp20050 -> SimpleGame.java

The Life Cycle (2/4)



Visual Illustration of the Life Cycle

The Life Cycle (3/4)

- **create ():** Method called once when the application is created.

- **resize (int width, int height):**
  - This method is called every time the game screen is re-sized and the game is not in the paused state.
  - It is also called once just after the **create()** method.
  - The **parameters** are the new **width** and **height** the screen has been **resized** to in **pixels**.

- **render ():**
  - Method called by the game loop from the application every time rendering should be performed.
  - Game logic updates are usually also performed in this method.

The Life Cycle (4/4)

- **pause ():**
  - On desktop this is called when the window is minimized and just before dispose() when exiting the application.
  - A good place to save the game state.

- **resume ():** This method is called on desktop when unminimized.

- **dispose ():**
  - Called when the application is destroyed.
  - It is preceded by a call to pause().

Q&A

To follow...

libGDX Tilemap Editor