Technical Design Document

Detailed specifications and guidelines for the technical implementation of the video game. Include sections covering:

- The target platform (hardware baseline, resolution)
    - Resolution: 1280x720@60fps
    - Windowed & Fullscreen modes
    - Maximum memory usage must be under 256 MB (RAM)
    - Primary target platform: PC (GitHub & Itch.io)
    - Secondary target platform: Nintendo Switch homebrew.

- Architecture (overall diagram, classes, and description, external libraries)
    - We will be using the L16_Profiling_Solution branch of the SDL project we were given previously as a base everyone on the team is familiar with.
    - Libraries:
        - SDL2: base library for rendering and general cross-platform code
        - SDL_image, SDL_ttf: image and text rendering
        - Box2D: Used for easy collision detection
        - SDL_mixer: Audio
        - pugi_xml: Used to interact with configuration files, maps (.tmx files in XML format), save files and other assets also in XML format.
        - OpTick: Performance profiler for games which will be used to find bottlenecks in game logic, memory leaks and other performance-impacting bugs.
        - ImGui: GUI library for implementing most of the menus and HUD.
    - Changes to App: added the new modules, and boolean variables for pausing. Moved the debug variable from the Physics module to App for general access without having to include the physics header in every file where the variable is used.
    - Entities
        - Player: Player character on the overworld. Basic movement, interacting with objects and NPCs. Menus and other specifics are handled by their own modules.
        - Trigger: When the Player collides or otherwise interacts with a trigger, a variety of actions can be triggered, like opening a path, starting a cutscene or a dialogue.
        - NPC: Like Triggers, but can move around the area. Generally only used for dialogues but can also trigger full cutscenes.
        - CombatEntity: Entity mainly used for graphical display of characters during combat.
    - CombatController: Manages the game's state while in battle. Rewards the player if the battle is won. Mainly calls the menu module to show the correct menu elements for the player to be able to play, along with storing battle-related data and passing said data to the GUI for it to be displayed. CombatEntity and CharData objects are created for each character in battle.
        - CharData: Holds a given character's stats, skills and active status effects.

        - Skill: Base class for various actions a character can perform. Attacks are derived from this class. Skills can be used on enemies, the player, or any. This is determined by the targetType variable, while the amount of targets is set by the maxTargets variable.
        - StatusEffect: Abstract base class for status effects. The effect itself is implemented in a derived class.
    - MenuManager: Manages menu screens and their transitions. Also manages UI elements directly related to menus.
        - Menu: Struct that holds and renders the objects needed to display a specific menu.
    - Inventory: Holds and manages the player's inventory and equipment.
        - Item: Base item class to implement multiple types of items with various effects.
    - CutsceneManager: Manages and plays triggered cutscenes. advancing them as conditions are fulfilled (i.e: pressing a button, or a character reaches its destination).
        - CutsceneData: The data for a cutscene as a whole. contains the id for the cutscene, the current step and the list of steps.
        - CutsceneStep: Represents a single action to be performed during the cutscene, along with the condition to continue to the next step. Classes may derive from this one for more advanced behavior, as this class just advances to the next step whenever it is executed.
        - Dialogue: contains the needed data to display a character talking. Includes the name of the character, an image path and the text to be displayed, along with the type of dialogue and possible choices. The choice made by the player is handled in the CutsceneStep object that contains the dialogue data.
    - Additions to the Input module: Support for both controller and keyboard inputs simultaneously through the use of keymapping. each action can be mapped to a keyboard key and/or a controller button (or axis in the case of movement). Near Unity-style mapping. The module now stores a list of controllers, the currently active controller and the mapped buttons. This is a work-in-progress design and will be changed during implementation, but the output will be in the form of an array accessed using enum values associated with each action the player can do (interact/confirm, cancel, move in 4 directions, open menu, etc.)
        - Controller: Struct that stores controller data. Stores all buttons in an array, and 2D analogue inputs in a separate array.
    - GuiControl types added:
        - GuiControlDialogue: displays a character's dialogue box with the supplied values. Also holds a set of GuiControlButton for dialogues with choices.
        - GuiControlGroup: Initially designed to be used in menus. Groups GUI elements to move them around more easily if needed, as it acts like an anchor point for nested GUI elements.

- Technology choices (programming language, code base, programming style)
    - Programming language: C++ 14 because we think that it is a good programming language, modern and easy to compute, and the language control is better and has access to tools and libraries like SDL, BOX2D and Tiled.
    - Base code: We will develop our game based on this handout because we think it is a good and complete template with advanced features to start developing our game.
    - Programming style: We will program using OOP because it offers a set of advantages that make it a powerful and flexible programming paradigm for developing robust, scalable, and adaptable software.
    - We will develop a turn-based combat RPG as we think is a good way of showing the full potential of game art and the essence of an RPG game.

- Physics and collision detection systems
    - We will not be using a physics system because the character will move up and down, taking into account the direction in which the player decides to go. The collision system will be designed in order to the character's collision cannot be mixed with those places that he cannot pass, preventing him from passing through these same ones. This will allow entities such as npc or objects not to move even if we push them, and it also helps us prevent the character from getting stuck in a place that shouldn't be due to collision. So, we will use the library of Box2d for the collision system because it is easy to use and we have experience with that one.

- AI and pathfinding algorithms
    - Pathfinding algorithms:
        - We will use A* pathfinding method because is the most efficient, complete and fast method to implement and has the best optimization for the project.
        - If an enemy detects you and this enemy is far from you, the enemy will pathfind you and will start the battle.
    - Enemies Movement:
        - NPC’s in the game have patterned movement in the game to make the sensation of not being a statue and giving the sensation of a live world to the player.
    - AI algorithms:
        - NPC’s in the game react depending on the character you play because in the game the different characters have a status in the world and a different behavior depending on the area you are in.
        - The turn-based combat system implements a randomness algorithm for the selection of the enemy attack to have an unpredictable experience in-game.

- Optimization strategies
    - **Level Streaming:**
        - Level streaming refers to a technique used in game development to dynamically load and unload game assets as the player navigates through the game world. Instead of loading the entire game world into memory at once, which can lead to excessive memory usage and longer loading times, level streaming divides the game world into manageable segments or chunks.
        - By dividing the game world into segments, only the portions of the game world that are relevant to the player's current location are loaded into memory. This not only reduces loading times but also optimizes memory usage, as only the necessary assets are kept in memory at any given time. As the player moves through the game world, segments are loaded and unloaded dynamically, ensuring a seamless and uninterrupted gaming experience.
    - **Code Optimization:**
        - Code optimization involves the periodic review and refinement of the game's codebase to improve performance and efficiency. This process typically involves identifying areas of the code that can be optimized, such as removing unnecessary loops, reducing redundant calculations, and optimizing algorithms.
        - By optimizing the game code, developers can reduce resource consumption, such as CPU and memory usage, resulting in improved overall performance and responsiveness. Code optimization is an ongoing process throughout the development lifecycle, as developers continuously strive to refine and improve the performance of the game.
    - **Control of Objects on Screen:**
        - Controlling the number of objects displayed on the screen at any given time is essential for optimizing rendering performance. Rendering too many objects simultaneously can lead to rendering overhead, causing slowdowns and decreased frame rates.
        - To mitigate rendering overhead, developers employ techniques to limit the number of objects displayed on the screen. This may involve implementing culling techniques to hide objects that are outside the player's view or reducing the complexity of objects based on their distance from the camera.
        - Additionally, developers may implement techniques to minimize the physics and logic calculations associated with off-screen objects, further reducing the strain on the GPU and CPU.
    - **Game Logic Optimization:**
        - Optimizing game logic involves refining the algorithms and processes responsible for handling gameplay mechanics, such as damage calculations, enemy AI, and interaction mechanics. By identifying inefficient or resource-intensive algorithms, developers can optimize game logic to reduce the load on the CPU and improve overall performance.

        - This optimization process may involve streamlining algorithms, reducing unnecessary calculations, and employing more efficient data structures and algorithms. By optimizing game logic, developers can ensure that the game runs smoothly and efficiently, providing players with a more immersive and enjoyable gaming experience.

- Tooling and workflow recommendations
    - **2D Art Tools:** In the development of 2D games, utilizing the right tools is crucial for creating visually appealing graphics and animations. Software like Tiled and Adobe Photoshop are invaluable for various aspects of 2D game development. Tiled is commonly used for designing game maps by arranging tiles, defining collision areas, and setting up object placement. It provides a user-friendly interface for level design and allows for easy integration with game engines.
      On the other hand, Adobe Photoshop is a versatile tool for editing 2D sprites, creating animations, and crafting concept art. Its robust features for image manipulation, layering, and animation timelines make it a preferred choice for many game artists. From character design to background art, Photoshop offers a comprehensive suite of tools to bring game assets to life with stunning visuals.
    - **Version Control:** Version control systems such as Git, combined with hosting platforms like GitHub, play a pivotal role in facilitating collaboration and managing code changes in game development projects. Git enables developers to track modifications to the source code, manage different branches for features or experiments, and merge changes seamlessly.
      GitHub, as a hosting platform for Git repositories, provides additional features for collaboration, including issue tracking, pull requests, and code review. It serves as a central hub for developers to collaborate on game development, share code, and maintain a comprehensive history of changes. By leveraging version control systems and hosting platforms, teams can work efficiently together, ensuring that everyone is on the same page and conflicts are resolved smoothly.
    - **Integrated Development Environment (IDE):** An integrated development environment (IDE) is essential for writing and debugging code efficiently. Visual Studio Code is a popular choice for C++ programming in game development due to its lightweight yet powerful features.
      Visual Studio Code provides a customizable and user-friendly interface, making it suitable for developers of all levels. Its extensive library of extensions further enhances productivity by adding support for various programming languages, frameworks, and tools. With Visual Studio Code, developers can streamline their workflow and focus on writing high-quality code for their games.
    - **Efficient Work Pipeline:** Establishing an efficient work pipeline is essential for ensuring smooth and organized development throughout the game development process. This pipeline encompasses various stages, including task planning, resource allocation, development, testing, and deployment.
      Task planning involves breaking down the project into manageable tasks, setting priorities, and assigning responsibilities to team members. Resource

allocation ensures that the necessary tools, assets, and manpower are available to complete tasks effectively and on schedule.

- **Continuous Testing:** Continuous testing is an integral part of modern game development practices, enabling developers to detect bugs and performance issues early in the development cycle. By implementing automated testing processes that run continuously throughout development, developers can identify and address issues promptly, reducing the risk of encountering critical issues later in the project.

  Automated testing frameworks and tools allow developers to create and execute test cases automatically, covering various aspects of the game, including functionality, performance, and compatibility. Continuous integration and deployment pipelines further streamline the testing process by automatically running tests whenever changes are made to the codebase, ensuring that new features or modifications do not introduce regressions or breaking changes.

- Version control and collaboration practices
  - Using git technology together with the Github online repository system we will do the version control of the project as it is the most comfortable way to work in big groups and have access to all the project.
  - There will be a main branch and each new feature in a secondary branch, there will be a person in charge of all the pull requests. This decision is made due to prevention of further errors with the main branches.

- Delivery process
  - It will be uploaded to itch.io as is the best way to upload the game for free and other platforms as required, like Nintendo Switch homebrew.
  - It will be the release version of the Github main branch.

- Technical considerations relevant to the development process (requirements for different consoles & platforms)

- **1. Nintendo Switch Homebrew:**
  - Homebrew development on the Nintendo Switch involves creating games and applications for the console without official approval from Nintendo.
  - You will need to access the homebrew development community resources, including forums, tutorials, and development tools.
  - Ensure compliance with any legal and ethical guidelines established by the homebrew community.
  - Distribution typically occurs through unofficial channels such as forums, websites, or homebrew app stores.
  - Be aware of potential risks associated with homebrew development, including security vulnerabilities and the potential for console bans.

- **2. GitHub:**
  - GitHub is primarily a code repository and collaboration platform, but you can also distribute software through GitHub Releases.
  - Upload your game files, assets, and documentation to a GitHub repository.

    - Use GitHub Releases to package and distribute your game's executable files and assets.
    - Provide clear instructions for users on how to download and run your game.
    - Utilize version control features to manage updates and revisions of your game.
    - Consider open sourcing your game code to encourage collaboration and community contributions.

- **3. Itch.io:**
    - Itch.io is a popular platform for indie game distribution and hosting.
    - Create an account on itch.io and set up a project page for your game.
    - Upload your game files, including executables, assets, and documentation.
    - Provide a description, screenshots, and other information about your game on the project page.
    - Set pricing and distribution options as desired (including pay-what-you-want, free, or fixed pricing).
    - Utilize Itch.io's community features to engage with players, receive feedback, and promote your game.
    - Consider participating in game jams and Itch.io's various community events to increase visibility for your game.

TDD Tasklist

- Pau Mena Torres
    - The target platform (hardware baseline, resolution)
    - AI and pathfinding algorithms
    - Version control and collaboration practices
    - Delivery Process
- Edgar Mesa Domínguez
    - The target platform (hardware baseline, resolution)
    - Technology choices
    - Version control and collaboration practices
    - Delivery Process
- Roger Puchol
    - Architecture
- Eric Palomares
    - Physics and collision detections systems
- Rafael Esquius
    - Optimization strategies
- Bernat Cifuentes
    - Tooling and workflow recommendations
- Pau Hernandez
    - Technical considerations relevant to the development process
- Ferran Llobet
    - Version control and collaboration practices
    - Delivery Process

Production Plan Tasklist

- Pau Mena Torres
    - Gantt Chart
- Edgar Mesa Domínguez
    - Gantt Chart
- Roger Puchol
    - Gantt Chart
- Eric Palomares
- Rafael Esquius
    - Gantt Chart
- Bernat Cifuentes
- Pau Hernandez
- Ferran Llobet

Pitch Tasklist

- Pau Mena Torres
    - Audio
    - Production plan
    - Introduction
    - Conclusion
    - Guionist
- Edgar Mesa Domínguez
    - Video editing
    - Camera Recording
    - Guionist
- Eric Palomares
    - Investor
    - Guionist
- Rafael Esquius
    - Gameplay
    - Guionist
    - Conclusion
- Marc Gil
    - Guionist
    - Premise
    - Conclusión
- Erika Sanchez
    - Guionist
    - Narrative
    - Conclusión