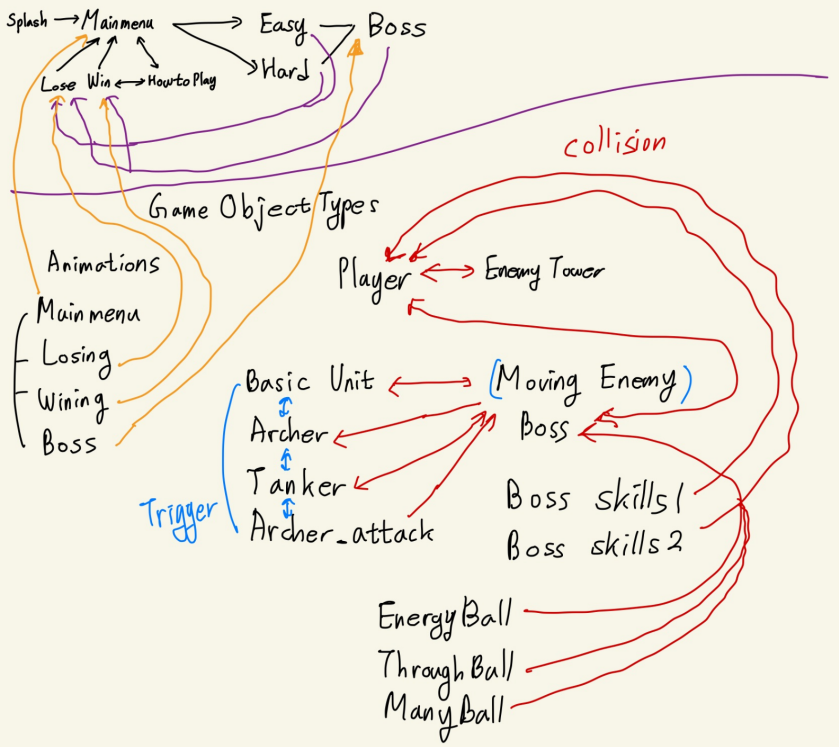
Technical Guide

Milestones 3

Team Name: **Celebration**
Game Name: **Palagang**
Milestone: **Milestone 3**

Total points: 170

1) Describe the overall architecture of this project. Include at least one diagram. (Example systems: Core, Factory, Physics, Graphics, Audio, etc.) Be sure to include file names in your diagram to help illustrate where the features live. (20)



2) What are the major technologies that are being used? (Example: DirectX 9.1, FMOD Designer, etc.) (5)

We used visual graphics for OpenGL 4.1 and the SDL for the Input and Window.

3) What 3rd-party libraries and assets are being used? Abide by the copyright notice and list them, along with their sources. (5)

In our project, Taehoon was responsible for creating all the assets, and as for the libraries, we included SFML for audio purposes. It's important to note that SFML is an open-source library.

```
////////////////////////////////////////////////////////////
//
// SFML - Simple and Fast Multimedia Library
// Copyright (C) 2007-2018 Laurent Gomila (laurent@sfml-dev.org)
//
// This software is provided 'as-is', without any express or implied
warranty.
// In no event will the authors be held liable for any damages arising
from the use of this software.
//
// Permission is granted to anyone to use this software for any purpose,
// including commercial applications, and to alter it and redistribute it
freely,
// subject to the following restrictions:
//
// 1. The origin of this software must not be misrepresented;
//    you must not claim that you wrote the original software.
//    If you use this software in a product, an acknowledgment
//    in the product documentation would be appreciated but is not
required.
//
// 2. Altered source versions must be plainly marked as such,
//    and must not be misrepresented as being the original software.
//
// 3. This notice may not be removed or altered from any source
distribution.
//
////////////////////////////////////////////////////////////
```

4) List the style conventions that are being used in this project for code formatting, documentation, file naming, etc. (10)

For our state machine implementation, we named the states in the format of State_Attack, and so on. We adhered to a coding style where if statements and curly braces {} were followed by a newline for clarity. For function names, we started with an uppercase letter, while variables were named in lowercase. In cases where two words were combined in a name, we inserted an underscore _ in the middle to maintain readability.

Examples :

Class : State_Attack

Variable : bool new_world

Function : GetNewworld(){ return new_world };

```
if(new_world)
{
        //Do this
}
```

5) Describe/show the support you have for debugging your game and how they are used: (10)

We have configured our system to be output-only through the console. We also have a debug drawing system in place. With the commands 'ShowCollision' and 'ShowTrigger', we can visually observe Triggers and Collisions. Collisions can be toggled on and off with '~', while Triggers can be controlled using 'T'

6) What other debug features has your team implemented? (+5)

[Debug Keys]
V : Decrease Palagang’s HP
B : Decrease Enemy Tower’s HP
R : Make the fast forward game play for debugging

7) How is debugging enabled/disabled? (10)

With the commands 'ShowCollision' and 'ShowTrigger', we can visually observe Triggers and Collisions. Collisions can be toggled on and off with '~', while Triggers can be controlled using 'T'

8) Explain your graphics API. Are you using fixed function or shaders? (10)

We used OpenGL which makes buffers and implements it.
We modified each one for the purpose.
We used a lot of shaders. We used font with the font_image.vert, font_image_attribute.frag. We used Texture with the texture_image.vert, texture_image_attribute.frag. We used shapes using shape.vert and basic_vtx_clr_attribute.frag.

9) Explain the process for loading graphical assets, such as sprites, models, textures, animations, etc. (10)

In our project, we employ OpenGL, a powerful graphics API, to handle the rendering aspects of our game. Specifically, we make use of framebuffers in OpenGL to manage our game's visual elements. This process involves a couple of key steps:

Creating Framebuffers in OpenGL: We start by creating framebuffers in OpenGL. A framebuffer is essentially a collection of buffers that can be used as the destination for rendering. This includes color buffers, depth buffers, and stencil buffers, among others. By using framebuffers, we gain greater control over rendering in our game, allowing us to implement various visual effects more efficiently.

Loading Assets as PNG Files: The next step involves loading our game assets, which are primarily in PNG format. PNG (Portable Network Graphics) is a popular image format known for its lossless compression and ability to handle transparent backgrounds, making it ideal for game assets like character sprites, backgrounds, and UI elements.

Utilizing Sprites for Rendering: Once our assets are loaded, we use sprites to render these PNG images. In the context of game development, a sprite is a 2D bitmap that is integrated into a larger scene, typically used to represent characters, objects, and other elements in a game. By using sprites, we can easily manipulate these images in our game world, such as moving them around, changing their size, or applying various transformations.

Integration with OpenGL's Rendering Pipeline: The sprites are then integrated into OpenGL's rendering pipeline. This process involves binding the textures from our PNG assets to the OpenGL context and then drawing them using the framebuffers. We can manipulate these textures using OpenGL's various functions to achieve the desired visual effects in our game.

Optimizing Performance: An important aspect of using framebuffers and sprites in OpenGL is performance optimization. We ensure that the rendering process is as efficient as possible, minimizing the load on the GPU and maintaining a smooth frame rate. This is crucial for providing an optimal gaming experience, especially for games with complex graphics and animations.

In summary, our use of frame buffers in OpenGL to render PNG assets as sprites is a key part of our game's graphics system. This approach provides us with the flexibility and control needed to create visually appealing and dynamic game scenes, enhancing the overall aesthetic and playability of our game.

10) Explain any process that your game might use to load and parse other game data, such as level files, enemy definitions, etc. (10)

In our game, we have a systematic approach for loading and parsing various game data, including level files, enemy definitions, and other related elements. This process is integral to how the game dynamically presents different levels and enemies, ensuring a varied and engaging gameplay experience. Here's an overview:
File Formats and Storage
Level Files and Enemy Definitions: Typically stored in custom file formats, which could be JSON, XML, or a proprietary format. These files contain structured data defining level layouts, enemy types, spawn points, and other game attributes.
Loading Process
Parsing Game Data: Upon game startup or level transition, the game engine reads these files. A parser is implemented to interpret the file contents. For instance, if using JSON, a JSON parser converts the text data into a format the game engine can use, like objects or arrays.
Dynamic Level Generation: The parsed data is then used to dynamically generate game levels. This includes placing enemies, setting up environments, and initializing any specific level attributes like background music or special conditions.
Enemy Definitions
Enemy Attributes: Enemy definition files detail characteristics such as health, attack patterns, sprites, AI behavior, and spawn logic.
Spawning Enemies: During gameplay, the game engine refers to these definitions to spawn enemies according to the level design. The AI behavior defined in these files dictates how enemies react to the player and the environment.
Data Integration
Game Objects and Environment: The parsed data is integrated into the game through GameObjects and environmental settings. For example, enemy data is used to instantiate enemy objects, and level data is used to set up the game environment.
Resource Management: The game engine manages resources like textures, sounds, and other assets required for the level, loading them into memory as needed.
Error Handling and Optimization
Error Checking: The loading process includes error checking to handle missing files, incorrect data formats, or parsing errors, ensuring the game remains stable.
Performance Considerations: Care is taken to optimize the loading process, minimizing loading times and ensuring smooth transitions between levels.
Extensibility
Adding New Content: To add new levels or enemies, developers can create or modify the respective files without altering the core game code. This extensibility allows for easy updates and expansions to the game.

In summary, our game employs a detailed and efficient process to load and parse various game data. This system not only enables the dynamic generation of levels and the integration of diverse enemy types but also ensures that the game can be easily extended and maintained over time.

11) Describe your physics API and processing from within your game engine. What kind of integration does your Physics API rely on? (Euler, Improved Euler, Verlet, Runge-Kutta, etc.) (10)

We employ Verlet Integration in our approach. This is because the movement of the player and the movement of the objects change over time.

12) What kind of space partitioning is used? (5)

We didn't use any specific space partitioning.

13) Explain collision detection in your engine. (10)

In our collision detection system, we only created rectangles (Rects) for the collision of game objects. The method of detection involves using the left, right, top, and bottom sides of these Rectangles to check if they can collide with other game objects, thereby confirming a collision.

14) Explain your project's AI systems and how they inform AI behavior. (10)

Our project's AI system encompasses the movement and attack mechanisms of allied and enemy units, along with the boss's attack patterns.

Allied Units: Upon the player's input, allied units are randomly spawned at one of two paths at the left edge of the screen. They move steadily towards the right. If an allied unit collides with an enemy unit, collision detection is triggered, leading to a 'Resolve Collision' state. This changes the unit's status to an attack state, halting its movement. In this state, the unit inflicts damage on the colliding enemy unit at regular intervals, reducing its health over time. Additionally, allied units check for other allied units in front using triggers and switch to an idle state if they find one. Units are removed from the game using a destroy function when their health falls below zero.

Enemy Units: Enemy units are similarly generated at random intervals from the right edge, moving towards the left. They follow the same collision and attack process as allied units.

Basic Unit Behavior: These units move from left to right at a constant speed. They attack any colliding enemy, gradually reducing the enemy's health by the amount of damage they deal.

Archer Unit Behavior: Archers also move from left to right but attack at regular intervals by shooting arrows. They can only attack with arrows and reduce the health of colliding enemies in the same manner as basic units.

Tanker Unit Behavior: Tankers move from left to right without an attack function. They reduce their health in response to the damage dealt by colliding enemies.

Boss Mechanics: The boss remains stationary on the right side of the screen and uses a set of five attack patterns randomly at regular intervals. These patterns continue until the boss is defeated, with skills being activated at specific time intervals within each pattern.

This AI system thus creates a dynamic and engaging gameplay experience, with different unit types and behaviors contributing to varied strategies and interactions in the game.

15) What types of algorithms were used to implement this behavior? Please describe any of the following features your game incorporates, in regards to AI: (10)
In our project, the AI system's focus is on the patterned movement of both allied and enemy units, as well as the distinct attack patterns of the boss. Here's a detailed explanation from this perspective:

[Allied Units]

Spawn and Movement: Allied units are randomly generated from two paths at the left side of the screen based on player input. They move consistently towards the right.
Collision and Combat: Upon colliding with enemy units, a collision check initiates a transition to a combat state, halting movement. In this state, units inflict damage on colliding enemies at regular intervals, mirroring the damage dealt by the enemy.
State Transition: Allied units also check for stationary allied units ahead using triggers and switch to an idle state via a 'ResolveTrigger' mechanism. Units are destroyed when their health drops below zero.

[Enemy Units]
Spawn and Movement: Enemy units appear at random intervals from the right, moving towards the left at a constant speed.
Combat Mechanics: Similar to allied units, enemy units engage in combat upon collision, attacking at regular intervals and receiving damage.

[Specific Unit Behaviors]
Basic Units: These units move from left to right and engage in direct combat with colliding enemies, receiving damage over time.
Archer Units: Moving similarly, archers attack at set intervals by shooting arrows, their only mode of attack.
Tanker Units: These units, devoid of attack capabilities, move from left to right and sustain damage from colliding enemies.

[Boss Mechanics]
Positioning: The boss remains stationary on the right side of the screen.

Attack Patterns: It randomly selects from five different attack patterns at regular intervals. Each pattern activates skills at set times, continuously cycling through these patterns until the boss's defeat.

From a pattern movement standpoint, the AI system in our game introduces a strategic layer where positioning, timing, and the type of units significantly impact the gameplay dynamics. The patterned movement and behavior of units create a complex and engaging combat system, requiring players to think tactically about unit placement and timing to effectively counter the enemy's strategies and the boss's varied attack patterns.

16) Explain any networking that your engine supports and uses for this game. What is the method for hosting and connecting to games? What kind of network protocols does it use? What features are included in it? (Like encryption, data compression, host migration, rollback techniques, etc.) (+10)

Unfortunately, we did not implement networking in our game. If we were to implement it, I think we would create a score leaderboard.

17) Does your game support multiplayer in any way? If so, how? (+10)

Our game is a single-player game. The mechanics are not quite suitable for multiplayer.

18) Explain the content development pipeline for your game. If someone wanted to create another level or more enemies, what would be the process for that? (10)

[Design and Conceptualization:]

Start with designing the new level or enemy types. This involves conceptualizing the gameplay mechanics, visual style, and how they fit into the existing game narrative and difficulty balance.

[Asset Creation:]

For new levels: Create or modify level assets like backgrounds, platforms, obstacles, etc. This might involve using a tool like Piskel for pixel art or other graphic design software.
For new enemies: Design and create enemy sprites, animations, and any special effects they might have.

[Coding and Integration:]

Add the new assets to the game's asset directory (e.g., assets/).
Update the game state file (like Mode1.cpp) to include the new levels or enemies.
For a new level, this might involve creating a new game state class similar to Mode1, with its own Load, Update, Draw, and Unload methods.

For new enemies, define their behavior and properties, then instantiate them in the game world, likely in the Load or Update methods of the relevant game state.

[Game Logic Updates:]

Adjust the game logic to incorporate the new content. This includes updating collision detection, enemy AI, level progression logic, and any other relevant systems.

[Testing and Debugging:]

Thoroughly test the new level or enemies to ensure they integrate seamlessly with the existing game. Check for bugs, performance issues, and ensure that the gameplay experience is balanced and enjoyable.

[Optimization and Polishing:]

Based on testing feedback, optimize the new content for performance and refine any details for better player experience. This might include tweaking enemy behavior, adjusting level design, or improving visuals and sound effects.

[Documentation and Version Control:]

Document the changes made and update the project documentation to reflect the new content. Use version control (like GitHub) to manage the changes and ensure that the team has access to the latest version of the game.

19) Describe/show all tools that are used in your project's development. (10)

For the development of our project, we utilized a variety of tools to streamline our workflow and enhance collaboration. We primarily used Visual Studio as our integrated development environment (IDE) for writing and managing our code. It provided a comprehensive set of tools for coding, debugging, and testing our application, ensuring a robust development process.

In addition, GitHub played a crucial role in our project for code integration and version control. It allowed us to collaboratively work on the codebase, track changes, and manage different versions of our project seamlessly. This tool was essential for maintaining a synchronized workflow among team members and for ensuring that all code contributions were properly integrated and stored.

Furthermore, for the creation of graphics, Taehoon utilized the online platform Piskel. This website offered a user-friendly interface for designing pixel art and animations. It was particularly beneficial for creating and editing sprites in a format that was compatible with our game's aesthetic and technical requirements. Piskel’s intuitive tools and accessible online

platform made it an ideal choice for our graphic design needs, allowing Taehoon to efficiently produce high-quality visual assets for our project.

18) Describe/show the structure of how your editors work, along with: (+10)

In our game development project, we have a structured approach to object management and editor functionality, which is crucial for an efficient and streamlined development process.

We start with object creation, primarily using Visual Studio to generate new source and header files. The objects in our game are designed with a strong emphasis on class inheritance, allowing for extended functionality through deriving from components and GameObjects. This is particularly evident in skill-related objects, where we implement particles and enable customization through various constructor parameters. Managing these objects is a key part of our process, and we rely on ParticleManager and GameObjectManager for this. For instance, adding a player object is done using GetGSComponent<GameObjectManager>()->Add(player_ptr);, and for particle management, we use something like AddGSComponent(new ParticleManager<Particles::Flower_particle>());.

When it comes to object deletion, we ensure this is handled efficiently through the same management systems, particularly focusing on the unloading of objects at the end of each stage. This is achieved with commands like GetGSComponent<GameObjectManager>()->Unload(); and ClearGSComponents();.

Our current project setup doesn't involve copying objects for reuse, which means each object is unique in its lifecycle within the game.

Object movement and rotation form another crucial aspect of our game's dynamics. Each GameObject can adjust its rotation using functions like SetRotation(), and we ensure that collision detection is robust and adapts with rotation by setting it as poly collision. For particles, such as those used in skills, we utilize the capabilities of ParticleManager for handling rotation, along with functions like SetVelocity() and UpdateVelocity() to manage movement. Again, poly is set for collision detection to ensure accuracy.

In terms of visual elements like images and animations, tools like Photoshop and Aseprite are our go-to choices. We create and then load these visuals into the game through .spt files, which contain all the necessary paths and animation frames. Additionally, .anm files are used to specify the details of animations, such as frame usage, duration, and instructions for post-animation frame return.

This methodical approach not only streamlines the development process but also enhances the overall gameplay experience by ensuring that each object and its interactions within the game are well-defined and efficiently managed.

19) Describe/show the audio pipeline. (10)

We made a SoundEffect.h and SoundEffect.cpp to implement the Sound.
First, make it static and load it. Load it using the SFML library.
I made the Sounds into a list with sf::Sound and saved them one by one.
In the Play we set the buffer and the volume and play it.

There is a loop and loop_play difference between two is that Looping or not.

20) Describe/show the art pipeline. (10)

From an art pipeline perspective, our project utilizes OpenGL and its features such as framebuffers to manage and render game graphics, ensuring a seamless integration of art assets into the game. This process is a critical part of our art pipeline, transforming raw graphical assets into dynamic game elements.

Art Asset Creation and Preparation
PNG Files for Assets: Our artists create assets like character sprites, backgrounds, and UI elements, primarily in PNG format. The choice of PNG is due to its lossless compression and support for transparent backgrounds, which are essential for game graphics.
Asset Optimization: Before these assets are imported into the game, they undergo optimization to ensure they are game-ready. This includes adjusting resolutions, ensuring consistent palettes, and optimizing for performance.
Integration into the Game Engine
Creating Framebuffers in OpenGL: At the core of our rendering process is the use of OpenGL framebuffers. These framebuffers are collections of buffers that store various components of an image, like colors, depth, and stencils.
Sprites for Dynamic Rendering: Once the assets are loaded into the game, they are rendered as sprites. This allows for dynamic manipulation within the game environment, such as movement, scaling, and rotation.

21) Does your engine leverage any kind of scripting langauge? If so, then please describe/show that! (+10)

We only used C++ and for the shader GLSL.

22) For the following features, please mark if they were implemented within your project. (0)

Feature	Implemented?
Voice-over audio	X

In-game SFX	O
Background music	O
Spatial audio	X
UI/menu SFX	O
Advanced/dynamic audio filters	X
Dynamic lighting/shading	X
Vector graphics	X
Sprite-based animation system	O
Sprite scaling	O
Sprite rotation	O
Particle effects	O
Image masks	X
Partial-transparency image blending or alpha blending methods	X
Parallax backgrounds	O
Multiple layers of background graphics	O
Kinematic/skeletal art	X
Scripted motion using vector paths	X
Animation tweens	X
Multiple levels/environments	O
Advanced physics simulations	O
Local multiplayer	X
Networked multiplayer	X
Downloadable content or online content fetching	X
Any other network features	X
Integration into web technologies (Such as web portals)	X
Component-based architecture	O
Game objects use C++ interfaces	O

Game objects are data-driven from factories	O
Menu systems	O
In-game HUD	O
File parsing for gameplay content, such as levels, scenarios, enemies, etc.	O
In-game level editor	X
Scripting language integration	X
Metaclasses, reflection, data-binding	X
Live object property inspection	X
Art pipeline tools	O
Audio pipeline tools	O
Engine/game build (compilation) tools	O
External gameplay editor	X
In-game gameplay editor	X
Testing tools and advanced debugging features	O
Ability to jump into the game in a given scenario for testing	O
Fast-forward and rewinding gameplay	O
Extensive/complex debug drawing	O
In-game performance visualization	X