Technical Design Document – Outline

# Title Page

Sweets Stacker

# Document History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author(s) | Changes |
| 0.1 | April 14, 2024 | Morgan Lawrence | Table of Contents, Entities, Components, Behaviors and Technical Risks |
|  |  |  |  |
|  |  |  |  |

# Table of Contents

[1 Title Page 1](#_Toc380649073)

[2 Document History 1](#_Toc664624009)

[3 Table of Contents 1](#_Toc1037645858)

[4 Game Summary 1](#_Toc1341799109)

[5 Development Environment 1](#_Toc405094720)

[5.1 Development Hardware 1](#_Toc522028192)

[5.2 Programming Languages 1](#_Toc1842853032)

[5.3 Development Tools 1](#_Toc666893497)

[5.4 External Code 1](#_Toc326720430)

[5.5 Game Engine 1](#_Toc1225856755)

[6 Architectural Analysis 1](#_Toc775593681)

[6.1 Classes 2](#_Toc1733408212)

[6.2 Behavioral Analysis 2](#_Toc1682336108)

[6.3 Game Loop 2](#_Toc258976561)

[7 Technical Risks 2](#_Toc914284442)

# Game Summary

A brief description of the game. You may borrow content from your Game Design Document.

Overall, this game portrays a simple arcade adventure of a dog. The dog, Scoopy, uses his love for sweets to complete the challenge of stacking as many ice cream scops as possible using only his nose.

# Development Environment

## Development Hardware

List the hardware resources (including operating system) needed for game development.

PC Windows 10

## Programming Languages

Specify the primary programming language(s), including version (e.g. C++03), to be used for development. Also specify any scripting languages that are used (if applicable).

C++23 with SFML 2.6.1

## Development Tools

List the software needed for development, such as IDEs, compilers, debuggers, version control software, graphical editors, sound editors, etc.

Visual Studio

## External Code

List the libraries to be used, including the URLs of the official library site, and the version number of the library used in this project.

SFML (https://www.sfml-dev.org)

## Game Engine

Describe the game engine to be used (or implemented), as well as modifications that need to be done to it.

GEX game engine developed in prog1266. This is an Entity-Component-Systems based game engine.

# Architectural Analysis

## Classes

Describe the classes that will have to be implemented. For each class, provide:

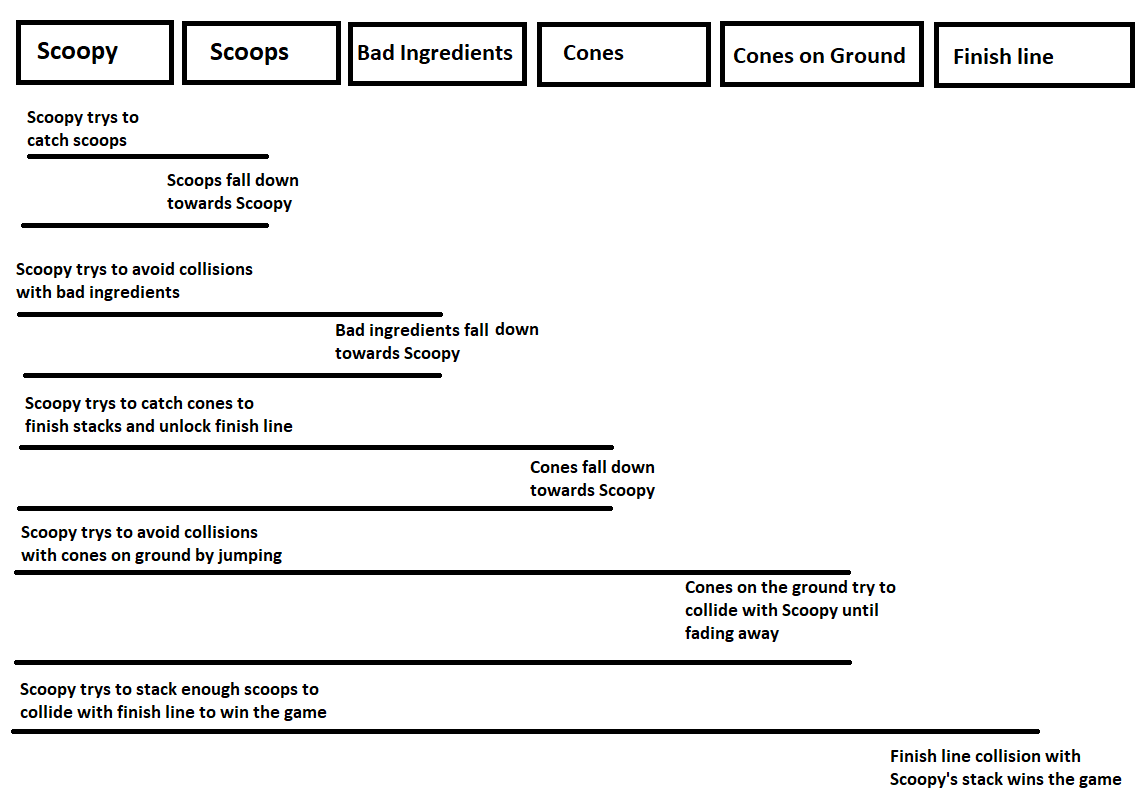
* Its responsibilities
* How it collaborates with other classes

Present class diagrams that show the relationships between classes. Show only the most essential attributes and methods for each class.

|  |  |  |
| --- | --- | --- |
| Class | Responsibilities | Collaborations |
| Animation | Animation class manages sprite animations, frame information, timing, and sprite information. It updates the animation, checks if it has ended, and access properties like the animation name and sprite | Animation collaborates with the Assets class to load and manage sprites / animations. It also collaborates with entities that use animations, with the Entity class |
| Entity Manager | EntityManager class is responsible for managing game entities, like adding, retrieving, and updating them | Entity Manager collaborates with the Entity class to manage game entities |
| Entity | Entity class is responsible for representing individual entities in the game, managing their components, and providing functionality to add, remove, and access components | Entity collaborates with the EntityManager class to be managed in the game |
| Assets | Assets class is responsible for managing game assets like fonts, textures, sprites, sounds, and animations. It loads assets from files, add them to the asset repository, and retrieve them | Assets collaborates with Animation, Scene, and SoundPlayer, to provide access to game assets textures, fonts, sounds, and animations |
| Scene\_SweetsStacker | Scene\_SweetsStacker class is responsible for managing the gameplay scene for the game. It handles player movement, collision detection, entity spawning, score tracking, level progression, and rendering. It has helper functions for managing player actions, spawning entities, and initializing the scene | Scene\_SweetsStacker collaborates with the EntityManager, Assets, Animation and other classes to manage entities, assets, and animations within the game |
| Scene\_Menu | Scene\_Menu class is responsible for managing the menu scene in the game. It handles displaying menu options, managing user input, loading levels, and transitioning to other scenes based on user actions. It allows for initializing the menu, registering actions, and rendering menu elements | Scene\_Menu collaborates with the GameEngine class to handle scene transitions and manage user input. It also interacts with the Assets class to load menu assets |
| SoundPlayer | SoundPlayer class is responsible for managing and playing sound effects in the game. It handles playing sounds with or without position information, removing stopped sounds, and managing the player's position and direction. It ensures that sound effects are played correctly throughout the game | SoundPlayer collaborates with the Assets class to access sound assets and play sound effects in the game. It also can interact with the Entity class to associate sound effects with specific entities |
| Scene\_Controls | Scene\_Controls class is responsible for managing and displaying control options. It handles user interactions related to controls and responding to user input. | Scene\_Controls collaborates with the GameEngine class to handle user input and manage scene transitions. It also interacts with the Assets class to load menu assets |
| Scene | Scene class is responsible as a base class for all scenes in the game. It encapsulates functionalities and properties shared through different scenes like managing entities, handling game updates, rendering scenes, and processing user commands. It defines virtual functions that subclasses can implement to customize behavior specific to each scene, like updating, rendering, and responding to actions | Scene collaborates with various other classes like EntityManager, Assets, and GameEngine to manage game state, update logic, and render scenes |
| GameEngine | GameEngine class is responsible for managing the game's functionality, like initializing the game window, handling scene changes, managing user input, and running the game loop. It maintains a map of scenes, allowing for transitions between different parts of the game. It provides methods for loading configuration settings, updating game statistics, and controlling the game's execution flow | GameEngine collaborates with all other classes to manage the game's execution, including scene management, user input, asset loading, and running the game loop |
| MusicPlayer | MusicPlayer class is responsible for managing music playing in the game. It provides functionality to add, play music, stop music, adjust volume, and pausing. It uses Singleton to make sure that only one instance of the MusicPlayer exists throughout the game | MusicPlayer collaborates with the Assets class to access music assets and play background music within the game. It can also interact with the Scene class to control music playback based on the current scene |
| Scene\_Credits | Scene\_Credits class is responsible for displaying the credits scene. It manages the layout and rendering of text displaying credit information. It handles user input and scene transition | Scene\_Credits collaborates with the GameEngine class to manage scene transitions and user input. It also interacts with the Assets class to load menu assets |

## Behavioral Analysis

Present statecharts, flow charts (activity diagrams), sequence diagrams, etc. that model complicated behavior. If your game has actors that implement a state machine, this would be the section where you’d present the statechart.



## Game Loop

Describe, in order, the sequence of activities that happen during each game loop. You must document this even if you’ll be using the “Clown Cannon” game engine.

1. Start the game.
2. Display the Main Menu.
3. Wait for player input.
4. If player selects "Start Game," proceed to the game loop.
5. Game Loop:
   * Handle input.
   * Update game state.
   * Draw/Render game world.
   * Check for collisions.
   * Repeat loop until conditions are met.
6. End of Game:
   * Display game over screen or victory screen.
   * Return player to the Main Menu.

# Technical Risks

List all technical risks that could make it difficult or impossible to complete the game. Examples:

* Uncertainty on how to implement a certain feature
* Uncertainty on if a certain feature can be executed fast enough in real time
* First time using a certain library

|  |  |  |
| --- | --- | --- |
| Risk | Severity | Mitigation (what is to be done to eliminate or minimize this risk) |
| PlayerNose bounding box | High | Map the playerNose movement to the player movement with an offset to have the right position while following player position |
| Player Animations | Low | Set the animation based on the direction of the input (Left, Right) |
| End Screens | Medium | Render a new end screen (win or lose) based on state and player lives |
| Collisions with player and cones on the ground | High | Update a physics and collisions for player and cone to offset the players bounding box so the player wont get hurt if players tail is over a cone but only players parts touching the ground |
| Stacking scoops in order of catch | High | Place the caught scoops in a vector based on the collision with a falling scoop to add a sprite of correlating flavor |
| Uncertainty on player death animation | Medium | Understand the animation components to check when a collision happens to check state of animation |
| Uncertainty on pause menu functionality with mosue | Low | Understand the actions available to use and base on game state isPaused |