Slime Wars

- Inspired by the projectile-based Angry Birds and the bubble shooter and tower defense genres, Slime Wars tasks the player with fighting an onslaught of slimes that move towards you in waves.
- With the goal of surviving as long as possible and racking up the highest score, the player must shoot the slimes down with care to make the greatest use out of the combo system.
- BUT BEWARE! Shoot the slimes with the wrong colored projectile, and they will turn it into one of their own to charge on you!

Credits

- This game could not have been accomplished without the great artwork uploaded to the internet open to everyone :)

Assets

1. Background - Summer pixel art seamless background | OpenGameArt.org
2. Old Cannon - Old Cannon by KingKelpo (itch.io)
3. Slime - Animated Slime | OpenGameArt.org -> Calciumtrice
4. Projectile - Splash Effect [32x32] | OpenGameArt.org
5. Heart Pixel Art - heart pixel art | OpenGameArt.org, DontMind8.blogspot.com
6. Cannon SFX - Courtesy of the test files

Controls

Unfortunately, this game may be an exclusive controller-based game. I have only tested it with the controller and don't know what keyboard buttons may possibly be linked to the specific buttons…Apologies if this is an inconvenience to anyone! :(

- Left stick -> Position cannon
- A Button (South Button) -> Launch projectile
- Left/Right Shoulder Buttons (The ones above the triggers) -> Switch target lane

Debug Mode

- B Button -> Toggle Debug Mode

- Y Button -> Cancel Debug Mode

Game Features

- I'm very proud of what I did over the span of these last few days and would hope that you enjoy playing it as much as I enjoyed making it! :)

- Combo System
    - When launching a projectile of the same color at a slime, the slime that gets hit will find all the surrounding enemies of similar color and destroy them as well, recursively increasing the combo multiplier to score more points (like a bubble shooter!)
        - The combo multiplier is exponential, starting from x1 -> x2 -> x4 -> x8… and so on
    - With this, there are ways of racking up points more easily
        - Wait for a bunch of slimes to gather and then attack them; the combo chains will increase the points by a great amount!
        - Attacking the shoulder lanes rather than the center will start the combo chain origin on the side to expand the reaches of the combo multiplier!
- Enemy “Stickiness” (Bubble shooter)
    - Another aspect of bubble shooters that I adopted is the bubble “stickiness” when you shoot a bubble of a different color at another
        - In this case, the slimes will turn that projectile into a new slime that will move towards you at the same rate
- Lane Selection
    - I really wanted to add a 3-dimensional aspect to the game, so I made lanes that you can aim at and increased the scales of sprites to reflect how “close” they are to the screen
        - In actuality, there are parts to this where I did use 3D vectors and other parts where I didn't, so I'd like to call it pseudo-3D :)
    - Lane selection does support layer-masking! This means that any projectile to be shot at a certain targeted lane will only damage the enemies currently in that lane
- Projectile Aiming/Shooting
    - The player cannon will rotate according to the y-input received by the player
    - As the cannon rotates higher, the initial velocity in the y-direction for the projectiles when fired will be increased. The x-velocity will remain constant, according to SUVAT
- Game Progression
    - As the score racks up, the enemies will get faster and the maximum colors of the enemies and projectiles will increase.

        - This is part of the “bubble shooter” aspect of the game in which many popular bubble shooter titles get more difficult with more colors and lack of possible combos
- Title and Credits Screen
    - I really wanted a title and credits screen :)

Inner Workings Behind the Scenes

- Update Method / Game Loop
    - The application is comprised of a **World** that contains a set of **Levels**
        - In this case, I made a title **Level**, main **Level**, and Credits **Level**
        - The **World** cycles through them
    - Upon initialization, the **World** initializes the associated **Level** (which initializes its **GameObjects**) in the series it is to contain. The **World** is meant to only initialize one **Level** at a time to make the most of the memory available.
    - Upon update, the **World** delegates the work to the current running **Level**, which in turn updates all its associated **GameObjects**
        - **GameObjects** are treated as a collection of raw data and pointers to its respective components to link them. Otherwise, all logic associated with a “GameObject” is split into its different components that are stored separately for data locality reasons.
        - A compromise between data locality and coupling the components were met to support Object Pooling, which I found to be very integral to the inner workings of my game. Hopefully the game still runs well!
- Factory Method
    - The **GameObjectFactory** is in charge of initializing all GameObjects when provided with different parameters. This shoulders most of the work during **Level** initialization
- Physics Engine
    - There’s not enough to really call this an engine haha, but a class by the same name resolves all the collisions for the physics components
        - Supports AABB collisions and vector calculations to compare distances
- Object Pooling (Enabling/Disabling GameObjects)
    - The main component of this game is object pooling, which allows for the reuse of many of the **GameObjects**, which accounts for the projectiles (~10), enemies (~100), and UI score popups (~30)
    - I decided to put great importance in object pooling over other features particularly because I wanted memory usage to be efficient and optimized and my game heavily relies upon reusing the same components
        - Without it, my game would be a heavy mess of fragmentation and whatnot, so I’m glad I got it working in time! :)

- With Object Pooling comes enabling/disabling **GameObjects** as a side-effect, and this constitutes the ability to also enable/disable particular components and refer to their activity for optimized physics collisions
  - Unfortunately, I didn't have time to go further, but the game does currently loop through all components and update them, regardless of activity
    - Some changes to be made could be the movement of the active/inactive gameObjects to separate storage containers or separating them with a deque based on whether they're at the front of the container or back
    - However, this was decided against since I didn't have the time to implement a render "engine" to handle rendering the sprites on different layers, so the movement of components would have been one hot mess (similar to the physics components; order matters!)

Challenges

This part is kind of like a dev log (also for myself), so feel free to ignore it :)

- Components can get boilerplate over time
  - For almost every new **GameObject** I created, I found myself also creating new components for them. With better architecture and inheritance structures, this could have been avoided.
  - Another issue that inheritance could solve is that I found myself to mostly be changing every component possible when any new changes were made to the base **ControllerComponent**, **PhysicsComponent**, or **RendererComponent** classes.
- Object Pooling opened grounds for code stinks
  - The naming of the "Set Data" methods are absolutely horrendous and don't tell the reader anything about what the method does and continues to be carried down to the subclass level.
    - This gets especially hard to reason about within the subclasses and one could only guess what "Set Data" means as far as what data it concerns