Salt Fighters

Final Report

Post-Paradigms

Sourav Banerjee, Emely Diaz, Kayla Han, Kenneth Le,
Davie Nguyen, Benjamin Pham, Ronghua Wang, Kaiden Zapanta

Trailer Link: https://www.youtube.com/watch?v=IZ2g8r53mQI&ab_channel=KennethLe
Repo: https://github.com/Post-Paradigms/Salt-Fighter
owo

Input Buffer

Kaiden Zapanta

- ★ **Controller inputs**
  - The player's movement inputs are read as a 2D vector. A circle is divided into 8 parts and the location of the vector within those parts determines what keypad input to push into the input buffer at each frame. A neutral input is detected when the magnitude of the vector is less than or equal to a set threshold (0.5 as of now)
- ★ **The buffer**
  - A circular buffer with a fixed size that stores enums that correspond to keypad notations. Overwrites input based on FIFO semantics.
- ★ **Input flushing**
  - Inputs become stale after a certain number of frames (currently 25) which causes the buffer to be flushed. The buffer is also flushed when a move is successfully performed.
- ★ **Input leniency**
  - The buffer allows for input leniency, meaning the player is allowed some number of wrong inputs before the sequence is considered invalid. (Currently allows 2 wrong inputs)
- ★ **Input mirroring**
  - Input mirroring is also handled in the case that the player is on the right side.
- ★ **Input priority**
  - Motion inputs are checked at each frame from the order of complex (largest number of inputs needed for the sequence) to simple so that if multiple valid sequences are read from the buffer, the most complex input will prevail.

State Machine

Kayla Han, Kenneth Le

- ★ **Features these states with valid transitioning:**
  - Neutral, Blocking (Walking Backwards), Jumping, Walking Forward, Dashing, Airdashing, Hit-stun, Block-stun, Knockdown, Startup, Active, Recovery, Crouching, and CrouchBlocking.
- ★ **Locking out states:**

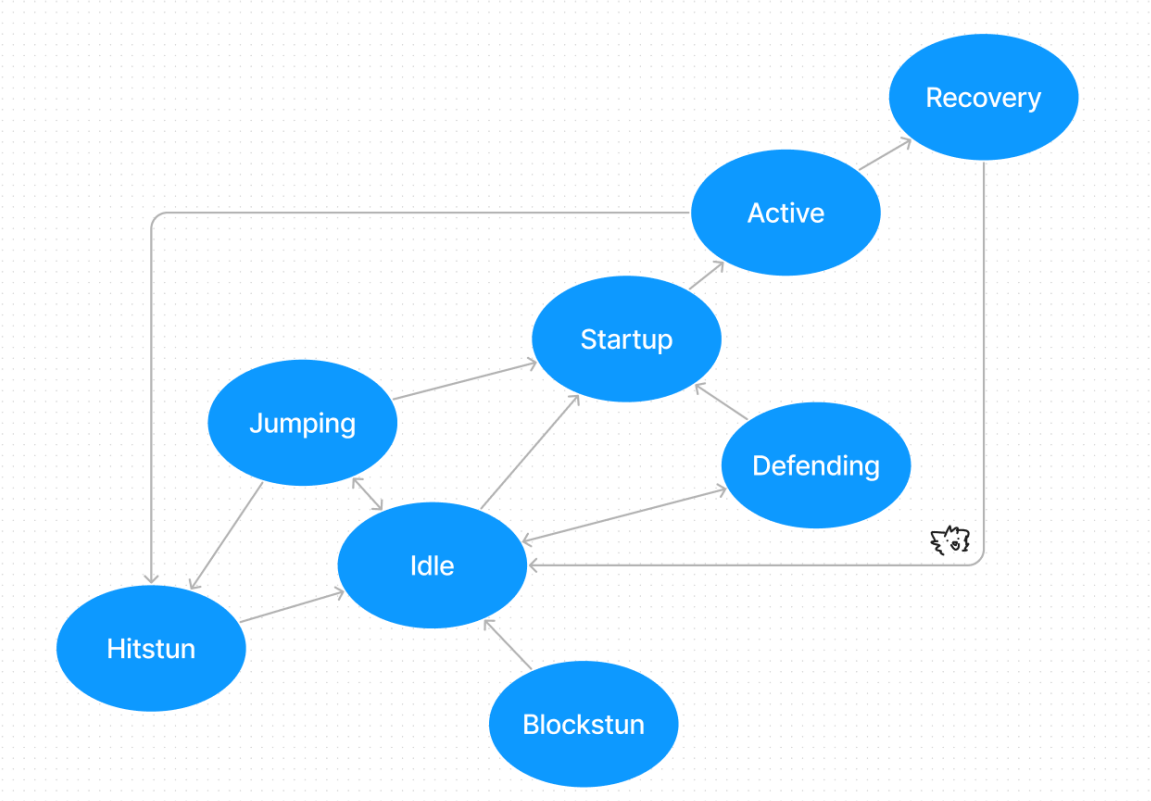
- When we are in certain states, we need to lock out transitioning to certain other states.
- The states that are locked out are most of the movement ones, such as neutral, walking forward, blocking, crouching, and crouch blocking.
- The states that acquire the lock are everything else, such as any attacks, dashing, jumping, the stuns, and knockdown.

★ **Changing state split into two functions:**

- Validate state: Ensures the new state can be updated from the current state
- Update state: Changes to the new state

★ **Frame timing:**

- We fixed Unreal's timer to 60 fps, thus we can ensure that Tick() is consistent.
- To handle updating states that must last a certain number of frames, we used an int, that we set and then decrement with Tick(). When that int hits zero, then we update to the next state depending on our current state.



Projectiles

Kayla Han, Kenneth Le

- ★ For Projectiles we implemented a AProjectileBase class which essentially is the parent class for all projectiles. This ProjectileBase class this is then extended by specific projectiles spawned by the player, like fireball, to allow for unique behaviors while still being derived from one base
- ★ The fighter spawns a projectile, the projectile then spawns its own hitbox and attaches this hitbox to itself
    - The owner of this hitbox is the fighter so that the fighter is not damaged by their own projectile owo
- ★ Projectile data is stored in their own separate data table rather than adding 500 more columns to the frame data table (I wanted to but Kayla stopped me I guess)
    - The data table contains information about the projectile's hitbox, along with its lifespan, speed, attack type, etc.

Frame Data Table

Kayla Han, Kenneth Le

- ★ Contains info for player moves
    - Startup / Active / Recovery frame numbers
    - Damage
    - Animation associated with each move
    - Attack type (High, mid, low)
    - Hitbox information (location + size)
    - Blockstun / Hitstun
    - Causes a knockdown or not
    - Whether a move is a jump cancellable/special cancellable
    - Target combo (Next target button & name of next attack)
    - Knockback Info
    - Projectile Data (if move Spawns Projectile)
    - If Invincible during Startup/Active (DP)
- ★ Stored in a .csv 🙂

Hit/Hurtboxes

Emely Diaz, Kaiden Zapanta, Kayla Han

- ★ Features for Hurtboxes and Hitboxes:

- Both hitboxes and hurtboxes are being implemented within C++ and are inheriting from the Actor class. They are set visible within the C++ code as well so we can make sure each is being spawned correctly on the character and for debugging purposes. Both hitbox and hurtbox classes also contain a pointer that contains the particular fighter that they are attached to.

★ Hurtboxes:

- Hurtboxes are spawned at begin play and then are attached to the fighter. The hurtboxes will be always attached to their character throughout the game and will not be despawned. Along with being attached, each hurtbox contains a variable Hurtbox Owner which is a pointer to the specific fighter that they are attached to. This variable helps later on when we detect the collisions of a hitbox with the hurtboxes. The hurtboxes themselves deal with the detection of collisions from hitboxes that are coming from the opposing fighter. When a collision is detected from the opposing character's hitbox, the collision method will call OnOw() to the hurtbox owner, which deals with updating the state of the character that got hit. Then we also apply the damage based on the attack that the opposing character performed, which is applied to the hurtbox owner. Then within the method, we call OnHitOther() to the Owner of the hitbox, to update the state of the opponent as well.

★ Hitboxes:

- Hitboxes are only spawned in the active state of an attack that is being performed by a fighter. The hitboxes contain an Initialize method called whenever the player reaches the active state of an attack. Then they are set to last until the end of the active frames based on the number of active frames for each attack. After the frames for the active state are completed the hitbox is then destroyed. We then use the data table, which contains information about each possible attack for the player, the animation that will be played, the damage of each attack, and the location of where we want the hitbox to spawn. The hitboxes are spawned accordingly based on information from the data table and are attached to said fighter.

Gameplay Framework

Davie Nguyen

- ★ GameMode:
  - The FightGameMode class manages the player characters and controllers as well as the general flow of the match. It spawns and initializes characters and controllers at the beginning and resets them at the end of each round.
- ★ GameState:
  - The FightGameState class manages the state of the game during the match such as the round time, current round, and rounds won. This class also manages the UI and updates it as the state of the game changes.
- ★ PlayerState:
  - The FightPlayerState class manages the state of the player at any given time in the match. This class keeps track of player health and the number of rounds a player has won.

UI

Davie Nguyen

- ★ Input Buffer Widget:
  - The input buffer widget displays each player's inputs on the side of the screen. The widget has its own input buffer that is updated alongside the player controller's input buffer, however, there is no flushing and the size of the buffer is limited to the amount of displayable area on the edge of the screen.
- ★ Gameplay Widgets:
  - There are additional widgets that display each player's health, the time left in the round, a countdown before the round begins, as well as the number of rounds each player has won.
- ★ Main Menu Widget:
  - The main menu widget allows the player to begin a match, view the credits, or quit the game. It supports mouse, keyboard, and controller input.
- ★ Pause Menu Widget: (*Kayla Han*)
  - Players can pause/unpause the game by pressing the esc/controller start button. This pauses the game for both viewers and brings up a bridget containing the list of moves the player can perform.

Camera

Sourav Banerjee

- ★ **Shared camera**
  - The camera uses the average position between both players and zooms in and out according to how distant they are. We also placed invisible walls at the camera's boundaries to limit the distance between both players.
- ★ **Invisible walls**
  - A wall object on each side of the screen prevents players from moving too far and keeps players within view of the camera.

Sound

Sourav Banerjee

- ★ **Background Music**
  - The battle theme was written using Ableton Live 11, in the spirit of themes from other fighting games (particularly drawing upon themes from Under Night In-Birth), and written to sync up with the game start. Played (and looped) in-game through a SoundCue.
- ★ **Sound Effects**
  - Sound effects are created and processed through Ableton Live 11, creating custom sounds for jumping and landing, light attacks, heavy attacks, light hits, and heavy hits. Sounds are stored as SoundCues and broadcast using GameplayStatics.

Toon Shader

Benjamin Pham

- ★ Toon Shader Features
  - Hard Shadows
    - Uses a post-process material to check the luminance of each pixel. If a pixel is above a certain luminance, the pixel is tinted one way, and if it is below, it is tinted another.
    - Allows artists to adjust the threshold of shadows
    - Allows artists to adjust both dark tint and light tint
  - Outline
    - Accesses the depth buffer of neighboring pixels, and if the difference is above a certain threshold, it colors the pixel the outline color.

Modeling

Ronghua Wang

- ★ Character Design and Mesh
    - After exploring various designs relating to food, bakery, and medieval times, a character was sketched out on paper: she wears a chef hat with a metal chest plate. This design is then translated into a 3D model using Blender: the body of the character is first modeled, and then its clothing elements and additional details are placed on top. The character also features curly hair that is built from Blender's built-in curve feature.

- ★ Rigging and Retargeting
    - Taking advantage of its auto rig features, the character is exported into Mixamo. The hair and the fork are later attached manually in Unreal as it creates major errors in the skeleton from Mixamo's auto rig. Before the completion of the final character mesh, test meshes were used to implement animations. Retargeting was scrapped from the previous checkpoint. Instead, the model and all of its respective animations had to go through a converter to add a root bone to its skeleton. That way, moving animations that do not stay in place, like the hurricane kick, will also move the character hitbox as well.

Animation State and Montages

Ronghua Wang, Kenneth Le owo

- ★ Animation State
    - All states—walking, idle, jumping, blocking, knockback, and crouching, are implemented by states in the animation blueprint through the state machine. Walking and idle are merged into one state. Using blendspaces, the animation runs depending on the velocity of the character: negative velocity is walking backward, positive velocity is walking forward, and zero velocity is idle. Jumping itself features 3 states: start jump, falling, and landing. Having these states makes the animation more smooth.
- ★ Montages
    - All attacks are implemented and called through montages. This is because montages not only override state animations but also can

be called in C++. Since there is a wide range of attacks, there are multiple montages for each attack.