Team Name: Snowfall
Group Members: Justin Millette (jmille36), Stephanie Wilson (swilso17), and Rachel Bonanno (rbonan02)
Planned Weekly Meeting Time: Saturday sometime between 2 and 7pm
Project Description: Multiplayer synced vertical scrolling rhythm game.
Minimum Deliverable:

- Client application that connects to a server and waits for the server to start the song.
- Server ensures that both clients start at the same time, so there is no delay between the two players.
- Clients will confirm their "readiness" manually.
- During the song, scores will update atomically, and be housed on the Server, based on the accuracy of the player's performance.
- At the end, a screen containing information about the results will be displayed.
- This will be a cooperative game.
- "Charts" – at least one file that contain a list of {time, key} pairs. These are the notes that come down during gameplay. (This will probably be us writing some script to convert existing chart files from other games into something that works for us)

Maximum Deliverable:

- All of the above
- "Meter" – if a player correctly plays enough notes in a row, they gain "meter charge." This can be activated by pressing some button and increases the score gained for some amount of time, as it decays. One person can activate this and it will prompt other players to activate.
- Players vote to choose from multiple songs

What's your first step?

- **DONE** Charting: Make at least one "chart"
- **DONE** Gameplay: Create a simple, singleplayer version of this game.
- **DONE** Concurrency: Build the server and client architecture to test latency between the two

What's the biggest problem you foresee or question you need to answer to get started?

- How to ensure that songs start at the same time (syncing) – this needs testing to figure out how possible this is in its current state
    - We can sync the start of the song for each client by:
        - Clients all signal they're ready from user input
        - Once all clients are ready, server sends a message containing a timestamp some amount of time (~5 seconds) in the future
        - Clients all confirm to the server that they received this
        - If the server receives all confirmations, we proceed as normal. Otherwise, restart the process
- How do we convert existing rhythm game charts to our new charts
    - Should be able to write a Python script to convert these

Design Decisions:

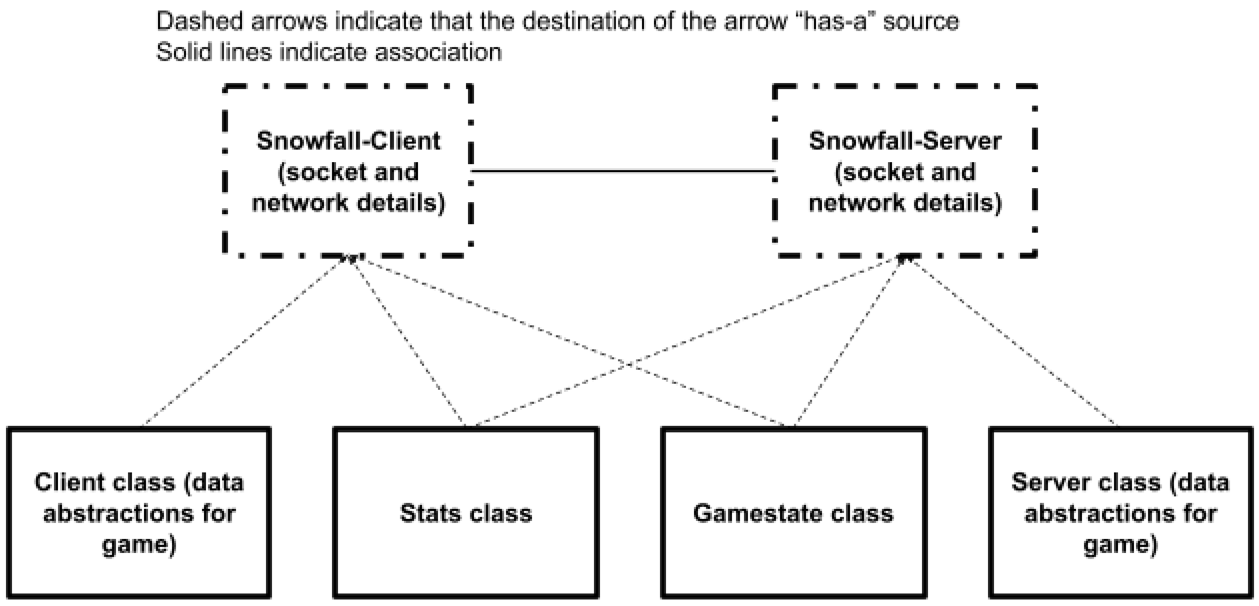
We ran into the concern: "how do we make our game interesting?" This is a very general question, but we define "interesting" as "using concurrency as a core element of gameplay." There were some ideas on how to address that:

- We initially were thinking about having cooperative and competitive game modes that would need to track score and update it accordingly.
    - But then the only element of concurrency was putting a mutex around the score variable and starting the song at the same time. That seems a bit simple, and it's not really interesting.
- Maybe we have a game where the players "ping-pong" notes back and forth to each other, a la Tetris Battle (send lines to your opponent, who has to deal with them or lose the game), and there could be power-ups to attack your opponent further (increase the number of notes to play, complexity of patterns, etc.)
    - This falls apart if we want this to be a rhythm game that cares about music, as increasing the number of notes is hard to do if the music (and thus note chart) is already pre-defined.

The idea we finally are landing on is different. Instead of four lanes per person, there are just eight lanes in total. Each of two players can play any 2 notes at the same time on any lane.

- This creates a more interesting experience gameplay wise and concurrency wise, because now we have to actively account for which notes each player is playing, and display that information on both players' screens.
- This allows for patterns in gameplay where one player is holding down a long held note, and the other player is playing something that would go into more than four lanes. An example of this can be seen here. The red player is holding notes, and the blue player is doing something in six lanes as opposed to just four, and then the players' roles swap. The interesting concurrency task comes up when trying to have this on multiple machines, as opposed to one giant arcade cabinet.

Class Diagram:


Dashed arrows indicate that the destination of the arrow “has-a” source
Solid lines indicate association
Snowfall-Client (socket and network details)
Snowfall-Server (socket and network details)
Client class (data abstractions for game)
Stats class
Gamestate class
Server class (data abstractions for game)

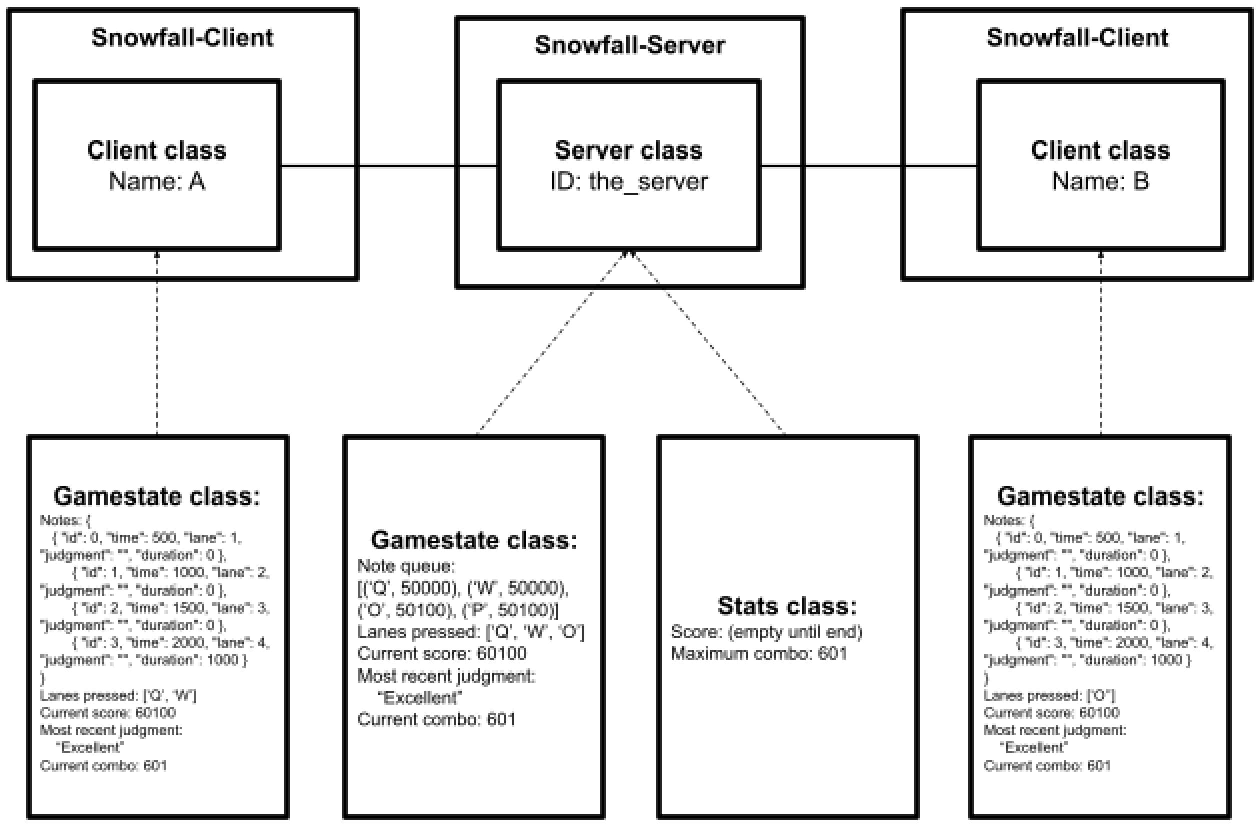
Classes:

- Snowfall_server and Snowfall_client files actually handle the message passing and information sharing between server and client. Server and Client are now for actual gameplay and display.
- Client
    - Is associated with a Server (Has a reference to the server’s socket)
    - Has a name
    - Has an instance of the Gamestate class
    - Has a start time that is sent from the server to the client for synchronization
    - Has a list of pressed keys
    - Has the last note that a score was announced for
    - houses the client loop module:
      Each pygame tick:
        - Receives new Gamestate from Server
        - Displays to screen (score, combo, recent judgment, current pressed keys, upcoming notes)
            - Notes are shown if the difference between Gamestate timestamp and note timestamp (in the chart) is small enough that it should be on the screen at this time
        - Check if there’s input from the player
            - If there is new input, we send updated Gamestate instance to the server
- Server
    - has an instance of the Stats class
    - has an instance of the Gamestate class, and a lock for it
    - houses the server loop module:

each tick (we expect that ticks are fast enough that scoring accuracy is doable):
        - Listens for clients to send over their Gamestate (which includes notes, their ids and judgements.)
        - Verify it's a valid input and then merge these into server Gamestate
            - Stores score data in the Stats object
        - With respect to latency, send merged Gamestate object to each connected client to make sure they both receive the most recent judgement

    After gameplay, Stats variables are printed to the terminal.
- Gamestate
    - Has a queue of upcoming notes/"Chart"
        - Notes are stored as a JSON object with "id, Time, Lane, Judgment, Duration"
        - Time and Lane are as we expect; time in milliseconds since the beginning of the song, and which lane the note will appear in
        - Judgment is updated as a note is hit by a Client. If a note is hit to have a better judgment than what is already stored (as in, one player hits the note too early, the other player hits it on time), it is updated to the better score. For scoring and stats tracking at the end of the song, we index through this field to see what score should be.
        - Duration variable is only used for Hold Notes. It is zero if the note is a normal tap note.
        - id exists so that we can ensure that we're accessing the same notes at all times from client to server – this allows us to make sure the state is aligned correctly.
    - Has a list of lanes pressed (8 of them - correspond to keyboard `QWER OP[]`)
    - Has the current score
    - Has the most recent judgment and the associated note ID
    - Has a queue of notes and judgments to be sent to the server
    - Has the current combo
- Stats object
    - Updated by Server during gameplay
    - Stores highest combo
    - Stores final score

Object Diagram:



```
Notes: {
  { "id": 0, "time": 500, "lane": 1,
"judgment": "", "duration": 0 },
      { "id": 1, "time": 1000, "lane": 2,
"judgment": "", "duration": 0 },
      { "id": 2, "time": 1500, "lane": 3,
"judgment": "", "duration": 0 },
      { "id": 3, "time": 2000, "lane": 4,
"judgment": "", "duration": 1000 }
}
Lanes pressed: ['Q', 'W']
Current score: 60100
Most recent judgment:
  "Excellent"
Current combo: 601
```

```
Note queue:
[('Q', 50000), ('W', 50000),
('O', 50100), ('P', 50100)]
Lanes pressed: ['Q', 'W', 'O']
Current score: 60100
Most recent judgment:
   "Excellent"
Current combo: 601
```

```
Score: (empty until end)
Maximum combo: 601
```

```
Notes: {
  { "id": 0, "time": 500, "lane": 1,
"judgment": "", "duration": 0 },
      { "id": 1, "time": 1000, "lane": 2,
"judgment": "", "duration": 0 },
      { "id": 2, "time": 1500, "lane": 3,
"judgment": "", "duration": 0 },
      { "id": 3, "time": 2000, "lane": 4,
"judgment": "", "duration": 1000 }
}
Lanes pressed: ['O']
Current score: 60100
Most recent judgment:
  "Excellent"
Current combo: 601
```

Dashed arrows indicate that the destination of the arrow "has-a" source
Solid lines indicate association

This object diagram is a snapshot just after a message from a client has been received in the server loop (an "Excellent" was scored), but before the next note has been hit. Two clients, "A" and "B," are connected to the server, "the_server." Each client has a Gamestate class which originated from the same source that the Server sent out – the only modification is the current pressed lanes. The other information in the client Gamestate classes is used for displaying the game on the screen, which is not pictured here. A player would see on either client screen that the 'Q', 'W', and 'O' lanes are pressed, that the current score is 60100, that the current combo is 601, that the most recent note was judged to be Excellent, and that there are four notes coming up – the notes in the 'Q' and 'W' lanes are on top of the judgment line, and the notes on the 'E' and 'R' lanes are 100 ms above (notes fall down as time progresses).

The server owns a Gamestate class that has all of the pressed lanes combined, but is otherwise identical. After this snapshot, it will note that there are two notes in range of being judged, assign the judgment "Excellent" to both of them, increase the current combo by 2, increase the score, modify the max combo in the server's Stats object, increase the "Excellent" count in the Stats object by 2, remove the notes from the queue, update the timestamp, and send out the new Gamestate to the clients, then repeat.

At the end of the game, the Server's Stats object will be populated with the final score, and displayed on the terminal.
One thing worth noting is how notes in the lanes will sync to the music. The start time of the music is what will determine the timestamps, so notes falling down the screen adhere to the music.

Development Plan:
The artwork is done by Steph and Rachel. Steph is making the notes, score graphic, and precision bar; while Rachel is working on creating the background. In terms of coding, the work will be divided at weekly meetings. While some of the code may be worked out together at a weekly meeting, most of the coding will happen outside of meetings. Justin will be working on making the charts and ensuring they are being read correctly. For the client and server side operations, these will be distributed in a way to prevent dependencies that if not communicated could cause issues. Code pushes will be communicated in order to further prevent these issues.

			Assignment Released	**Due Dates**	**Goals** ● To Do ● In Progress ● Completed ●		
Week 1	**3/13**	**T**	**Project Initial Design**		Create a Proof of Concept (read in chart, notes come down the screen, and player accuracy can be detected).		Create Game Sprites
	3/14	**F**					
	3/15	**S**					
Spring Break							
Week 2	**3/23**	**S**				Do Project Initial Design	
	3/24	**M**					
	3/25	**T**		**Project Initial Design**	Add a Second Player		
	3/26	**W**					Add Sound
	3/27	**T**					
	3/28	**F**					
	3/29	**S**				Background Layout Template	
Week 3	**3/30**	**S**					Create Background
	3/31	**M**					
	4/1	**T**	**Project Refined Design**		Do Project Refined Design	Add Concurrency of Play for a Second Player	
	4/2	**W**					
	4/3	**T**					
	4/4	**F**					

	4/5	**S**			🟩	🟩	🟩
Week 4	**4/6**	**S**			🟩		🟩
	4/7	**M**			🟩		🟩 Add in Sprites and Background
	4/8	**T**		**Project Refined Design**		🟩	🟩
	4/9	**W**				🟩 Bug Fixing/ Overflow/ Make it Pretty	
	4/10	**T**				🟩	
	4/11	**F**				🟩	
	4/12	**S**			🟩	🟩	
Week 5	**4/13**	**S**			🟩		
	4/14	**M**			🟩		
	4/15	**T**			🟩 Do Presentation Slides		
	4/16	**W**			🟩		
	4/17	**T**			🟩		
	4/18	**F**			🟩		
	4/19	**S**			🟩		
Week 6	**4/20**	**S**					
	4/21	**M**	**Project Final Report**		🟩		
	4/22	**T**	**Team Presentations**		🟩		
	4/23	**W**			🟩 Do Project Final Report		
	4/24	**T**	**Team Presentations**		🟩		
	4/25	**F**			🟩		
	4/26	**S**			🟩		
Week 7	**4/27**	**S**			🟩		
	4/28	**M**		**Project Final Report**			

Outcome Analysis

We completed all parts of our minimum deliverable except for an ending screen that showed the results. Instead of having a screen that shows the results, we just display them on the server's standard out. We did not have enough time to create the graphics for this screen and instead opted for a terminal print out in order to prioritize working through timing issues (due to game delay between players). We did not implement any of our maximum deliverables, but adding both a final screen and song selection are something we hope to add to the game in the future, especially since we already have the information necessary to make these features happen (the score that would be displayed on the end screen, and the capability for different songs to be played). All that said, we are happy with the result given the time we had and we completed our main aim for the deliverable.

Design Reflection

Snowfall allows players to hit any of the notes. This gives players the ability to modify the difficulty of the game based on what columns each player decides they want to play. With this additional functionality it introduces different cases that need to be handled; players can hit the same exact note (which can create a race condition) or opt to only handle notes that come down one side of the board. In order to handle this we implemented a game rule that states that the best accuracy between each player is stored for the overall score. This is done by both client threads sending messages to the server when they hit a note with the accuracy and note ID. The server receives this information and then compares the judgment/accuracy score of both clients for a given note. The best accuracy gets stored in the server and the server becomes the single point of truth. The server also sends messages to the client when the other client hits a note to tell the client that the other player has hit this note, so it stops being drawn on the second player's screen, as to minimize the chance of this race condition occurring in the first place. This one design choice to let players hit any note they please led us down more paths to concurrency and taught us more about client-server communication.

In terms of concurrency design decisions, one thing we particularly struggled with was how to handle the difference in timing between the two clients for displaying "No Credit" judgements. For example, if one client is ahead of the other, that means they register judgments into the server for notes faster than the other client. This means we do not truly know if a note is missed until both clients tell the server "hey, we missed!" We decided that when the server receives the "No Credit" judgment for a note from both clients, only then does it broadcast what the judgement was to both players. However, this decision still has its oddities. Since both players are most likely at different times in the song, one client will often see "No Credit" judgments more than the other. The faster player will hit the next note faster (and thus replace the "No Credit" with their new judgement) while the other will see "No Credit" for a longer time. This can be odd at times but it doesn't affect gameplay too much, just an interesting concurrency problem we tried our best at solving.

If we were to do it all again we would probably use the calculated ping in other places, rather than just syncing up the song at the beginning. If we had used the calculated ping when removing played notes from the other player's screen, we could have made those notes disappear later, as to give the illusion that it's all happening at exactly the same time (which is impossible as data does not travel instantaneously). What we actually did was: if the note has already been hit or missed by Player A, it waits until crossing the judgment line on Player B's screen before it stops being drawn.

Division of Labor Reflection

Division of labor worked out really well, we think. As our code was modularized into six files, we were each able to do some individual work without too many weird merge conflicts (there were a couple, but they were easily resolved). Steph and Rachel contributed the background (which unfortunately went unused as we switched from a four-column game to an eight-column game), all of the sprites, and the code to put those visuals on the screen. Rachel was responsible for a lot of the logistics as well, including setting up the Git repo, orchestrating the presentation, and dividing up work in the development plan above. All parties worked on the code, where Steph and Rachel implemented the message passing protocols, and Justin implemented the main gameplay loop and input handling, the script to rip .osu files, and the handling of messages on both the client and server. The final touches (read: last six hours of coding) were done all-together, as was the majority of the testing.

Bug Report

There was a bug in which one player's notes were disappearing halfway up the screen, as the notes had already been counted as No Credit on the other player's screen. It took 45 minutes and a lot of repeated print statements to determine what was actually causing this – as it was technically intended behavior at the time. But we decided (rightly) that that would make for poor gameplay, so we implemented the change mentioned at the bottom of the design reflection to fix this. The fix for this still allowed for player 2 to score a "better" accuracy on each note, which was our intent. To find this faster, we should have realized that this would happen when factoring in latency – this is what prompted us to include the ping during setup to ensure that gameplay happened at the right time.

Code Overview

<u>How to run our code:</u>

1. Run
   `snowfall_server.py [--host HOST] [--port PORT] [--chart CHART]`
   a. Arguments are not required, chart will default to basic.chart.
      i. NOTE about basic.chart – there is no associated audio with this chart. By default it just plays the song from the demo.
   b. The argument of chart must be a file path to a valid json formatted .chart file (examples supplied with the code)
      i. Additionally, osurip.py allows you to make .chart files with a given .osu file, the details of how to run this program will be in the following section.
2. On another computer or terminal, run
   `snowfall_client.py [--host HOST] [--port CHART] [--chart CHART] [--name NAME]`

    a. Chart argument MUST match the chart file run with the server. It is a contract violation to fail to do this, and it will have unintended behavior. Will default to basic.chart if omitted.
3. Two clients must connect to the server before the game runs.
4. Play and have fun!

We recommend using the "imprinting.chart" chart as it's what we tested on and is music that is not too insane.

<u>How to make a new song with osurip.py and set the song for the players (note that nothing concurrency-related happens in here):</u>

1. Run the .osu file through osurip.py by running
   ```
   python osurip.py song.osu path/to/out.chart
   ```
2. Go to the bottom of the file and ensure that the following arguments exist. If they don't, add them:
    a. Audio: Should be a file path to the MP3 file you want to play during the chart.
    b. Offset: A number that's used to offset when the notes should fall by a certain amount if the chart starts faster than the song, in milliseconds. 450 is usually close to correct in our testing.
3. Supply the wanted .chart file to snowfall_server in program arguments
4. Supply the SAME wanted .chart file to snowfall_client in program arguments
5. Play with your new song!

Obtaining .osu files is a different process, and they can be downloaded from the game's ("osu!") website.

<u>Files:</u>

For user use:

snowfall_client.py: Is run to start the client side of the program– handles all concurrency and message passing and receiving on the client's side.

snowfall_server.py: Is run to start the server side of the program– handles all concurrency and message passing and receiving on the server's side.

osurip.py: Can be used to create a new .chart file using .osu files.

Backend:

client.py: The Client class. Handles running and logic of the game.

server.py: The Server class. Stores data that the clients give it in regards to the game.

gamestate.py: The Gamestate class. Holds time-sensitive data about the game. Is passed between the client and server for communication.

stats.py: The stats class. Holds data that is only relevant until the end of the game for the final score and final combo. Is printed to the server's terminal at the end of the game.