FrameReject: Frame Rejection to Optimize Cloud Gaming Responsiveness

Lakshmi Sai Priyanka Boddapati
(A0248313J), School of Computing
e0925465@u.nus.edu

Muhammad Aldo Aditiya Nugroho
(A0248311M), School of Computing
e0925463@u.nus.edu

Saurabh Singh
(A0237499J), School of Computing
e0771095@u.nus.edu

Vaibhav Dasarahalli Ashoka
(A0237560E), School of Computing
e0771156@u.nus.edu

Varun Venkatesh Narayanan
(A0241600Y), School of Computing
e0787831@u.nus.edu

Abstract

We present FrameReject, a novel input prediction algorithm that is capable of frame prediction for a PC game at decent frame rates to send over a network. Our system performs evaluation of the probability of the game's future frames based on the player's inputs, and reject frames which probabilities are too low. Here, we describe both the model and our supporting game client/server architecture to support the model. The system requires a reliable network connection and is, to the best of our knowledge, the first cloud gaming input prediction method that incorporates frame rejection strategy. We have shared our implementation in github: https://github.com/Aldo-Aditiya/FrameReject.

1 Background

Mobile gaming accounts for a significant portion of the global consumer spending on video games [1]. The number of people gaming on mobile also accounts for a significant portion of digital gamers in the US [2]. This provides a sizeable incentive for game providers to give access of their games to the mobile market. The problem is, although the processing power of mobile devices gets stronger every year, modern video games are still more resource intensive - limiting the number of games that can be given access to mobile gamers. Recently, cloud gaming has emerged as a potential solution to this problem.

In cloud gaming, games are run and rendered on a capable remote server specifically constructed for game streaming. The server then streams the rendered frames to the players' devices, while also receiving game inputs from the player. As the frames are rendered on the server side, the player-side client only requires minimal processing power. This opens up the possibility of providing higher-end games to devices with weaker processing power, such as mobile phones.

But in reality, cloud gaming faces a number of problems - one of which is dealing with latency, while being able to stream game frames in real-time. As gaming is an interactive medium, players expect their input to be reflected quickly by the game. In a study by

Claypool et al. on online gaming [3], the performance of gamers tend to drop over an increase in latency, with the worst performance drop of 35% by an addition of 100 msec of latency. This drop is exacerbated in games that require fast reflexes such as first person shooters.

It is reasonable to find methods of solving this problem through similar, widely working applications, such as video streaming services (e.g. Netflix and Youtube). But because of the non-interactive and non-live nature of the video-based medium, service providers can resort to more time-consuming techniques to make video streaming more efficient - such as efficient video encoding before making the video available for viewing, and caching on the client side for more efficient repeat viewings [4]. Methods from livestream video services (e.g. Twitch) also do not fully fit to cloud gaming, as because of its non-interactive nature, "live" video feeds can actually be sent slightly slower than in real time - thus giving some leeway for heavier video processing. Another reasonable area to look into is multiplayer gaming - after all, like cloud gaming, it is both live and interactive. But methods to solve latency problems in multiplayer gaming often assumes that the game client is strong enough to render frames from the server-provided game states themselves [5], which negates the value proposition of cloud gaming.

Existing research to improve cloud gaming performance range from dynamic video encoding [6], dynamic virtual machine placement [7], to input prediction [8,9]. The focus of our approach will be in input prediction, with two notable existing research: CloudHide [8] and Outatime [9].

In CloudHide, Anand et al. [8] introduced the exponential prediction model, which generates frames for all the possible future game states that the player can input. In Outatime, Lee et al [9] introduced two separate methods for handling two separate types of inputs: (1) Navigational, and (2) Impulse. For navigational inputs such as movement and camera controls, they used a Markov Model which predicts the change in the translational and rotational vector based on the previous input, then they did shake reduction with Kalman Filter. Misprediction handling is then done through light image-based rendering. For impulse inputs such as firing a gun or using an item, they did not use a predictive model. But instead, they used a similar approach that is done by CloudHide [8], which sends all possible frames but with input shifting to reduce the number of possible states that can be generated.

In this project, we present FrameReject, where we modeled an alternative to the methods described in both Outatime and CloudHide. Instead of implementing Outatime's method of navigational input prediction and using image-based rendering for misprediction handling, we instead evaluated the probability of possible states (and with extension, frames) that can be generated over the consecutive time steps by our initial input. The frames with a low enough probability will be discarded, and we will only send the rest of the remaining frames. Compared to both Outatime's and CloudHide's approach, FrameReject has the benefit of minimizing the number of possible frames that needs to be sent to the client, thus potentially making transmissions more efficient.

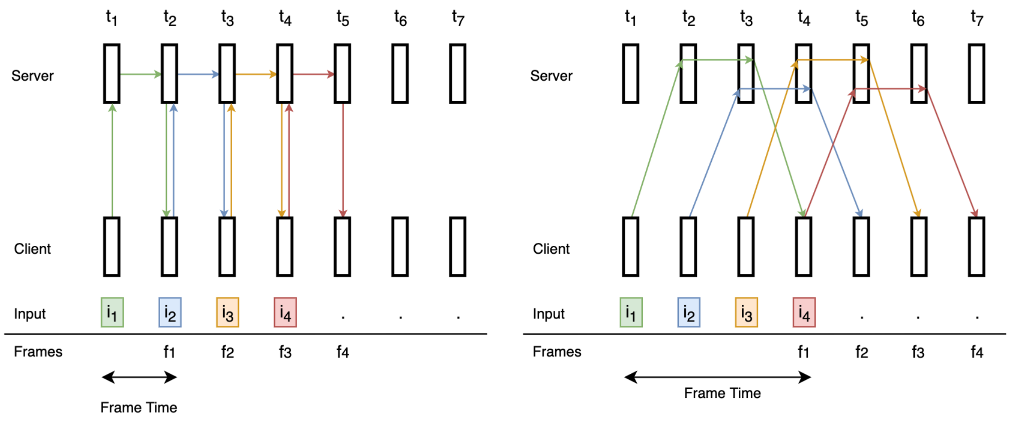
The rest of the report have been structured as such. Section 2 discusses the FrameReject architecture in detail. Section 3 discusses our experiments with various configurations of our method. Section 4 discusses the experimental results, potential flaws, and future improvements.

2 Method

2.1 Problem Formulation

To describe the problem of cloud gaming latency more formally, we will first compare client/server interaction between an ideal gaming scenario vs cloud gaming scenario. For this purpose, we illustrated the time frame for interaction between these two scenarios in Figure 1.

In Figure 1, t_k represents the game time step, i_k represents input at timestep t_k, and f_k represents the frame generated as a result of input i_k. The game time step itself depends on the framerate in which the game runs in, for example: 30 fps means we have $1/30 = 32$ ms delay between each time steps. We also defined frame time as the difference between the time step of the input and the time step of the generated frame corresponding to the input. A lower frame time means a lower perceived delay between what the user inputted and the corresponding frame, which is beneficial.



(a) Ideal Gaming Scenario

(b) Cloud Gaming Scenario

Figure 1: Client/Server Interaction Timeline between (a) Ideal Scenario and (b) Cloud Gaming Scenario

As shown in Figure 1a, in an ideal gaming scenario, the client and the server will be located on the same machine. Because of this, client input will be sent virtually without delay to the server, in which the server will then process said client input, and directly display the frames to the player. This results in a very short frame time of only 1 time step. Contrast this to the cloud gaming scenario, in which there is a delay in sending both the input to the server and the output back to the client. Assuming as in the case of the Figure 1b, that both input and output delay is 1 time step each, this results in a total frame time of 3 time steps.

Our goal is to minimize the frame time as perceived by the user. Modelling this mathematically, we want to minimize the frame time:

$$t(f_k) - t(i_k) = s(i_k) + ps(i_k) + s(f_k)$$

Where $t(f_k)$ and $t(i_k)$ are each the time for frame and input at time step k, $s(i_k)$ and $s(f_k)$ are the time taken to send input from client to server, and time taken to send frame from server to client respectively. Finally, $ps(i_k)$ is the time taken for processing the frames on the server. Our goal is to maintain frame rate of 30 fps, while achieving low frame times (approx 32 ms).

To model the above problem as a prediction problem, observe that to minimize frame time, we need to move input i_k as close as possible to frame f_k. But instead of manipulating the input or frame to get them as close in time as possible, what we can do is to predict the inputs i_l^* in between time steps of i_k and f_k, such that the generated frame at f_k is consistent with the actual player inputs at time steps t_l. Note that the number of inputs i_l^* that needs to be predicted is correlated to $t(f_k) - t(i_k)$.

As illustrated in Figure 2a, assuming that every prediction is correct, frame time for f_1 is measured with respect to t_3 rather than t_1, because the user perceives their input i_3 to be accounted for in f_1. Of course, this is because our predictions i_2^* and i_3^* are correct.

In the case where the inputs are not predicted correctly, as in Figure 2b, f_1 is incorrect, and we need to be able to substitute this frame with the correct frame so that player experience does not consist of jarring, unexpected frames.

Observe also that the time taken for the prediction process itself needs to be accommodated in the server process $ps(i_k)$. As the number of inputs that needs to be predicted depends on $t(f_k) - t(i_k)$, a sizeable increase in $ps(i_k)$ may result in an increased number of inputs required to be predicted. At worst, a significant increase in $ps(i_k)$ might make the delay too big, and itself affect the frame time.

To summarize the problems we will be solving:

- We need to minimize the frame time between the user's last input to its corresponding frame. To do this, we need to predict the user's last few inputs before displaying the frame, and use that to generate the corresponding frame.
- We need to handle cases when the prediction model fails to predict the correct input sequence, and thus predicts the wrong frame.
- We need to do this prediction in a short enough time frame, such that we do not meaningfully increase the processing time in the server.

The next subsection discusses in general our approach in solving these problems.

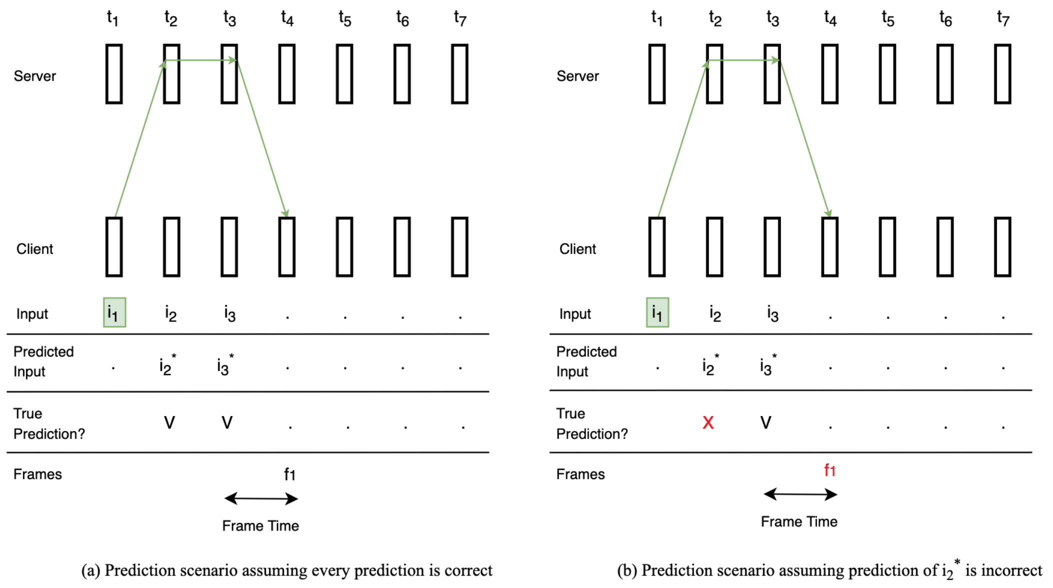

t1
t2
t3
t4
t5
t6
t7
Server
Client
Input
i1
i2
i3
Predicted Input
i2*
i3*
True Prediction?
V
V
X
Frames
f1
Frame Time
(a) Prediction scenario assuming every prediction is correct
(b) Prediction scenario assuming prediction of i2* is incorrect

Figure 2: Client/Server Interaction timeline, with added input prediction, assuming (a) All predicted inputs correct, and (2) Prediction at i_2^* is incorrect

2.2 Overview of Our Solution

As we experimented on cloud gaming, we implemented our own Client/Server interaction with each of its supporting components. In this subsection we will be describing each notable component of our architecture as shown in Figure 3.

Player Input and Show Frame For the game, we used an Atari game called Breakout, which we controlled by using the Arcade Learning Environment [10]. We also received player inputs and showed the frames using Pygame [11]. The reason we used Breakout is because of its simple movement of only three possible inputs. We will also mainly be focusing on the navigational inputs rather than impulse inputs. This is to limit the scope of this project, as handling impulse inputs can get quite complex.

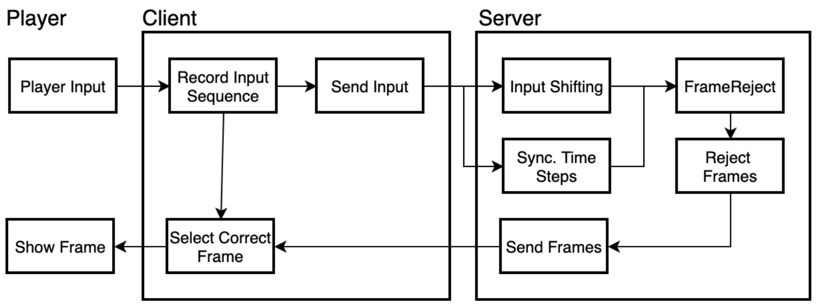


Figure 3: Overview of Our Solution

Send Input/Frames Input and Frame sending are implemented using a multiprocessed socket so that the data transmission runs separately from the main game loop. Frames are also compressed to make sending more efficient.

Input Shifting and Sync Time Steps To limit the number of inputs that can be sent by the client, we combined multiple individual time steps into a single sampling time step for a single input. We note that this is called input shifting, and is originally implemented by Outatime [9]. As for synchronization of time steps between client and server, this is required such that the server predicts the same number of inputs as the client is able to receive. This is done to prevent over or under-prediction of the number of inputs in the sequence.

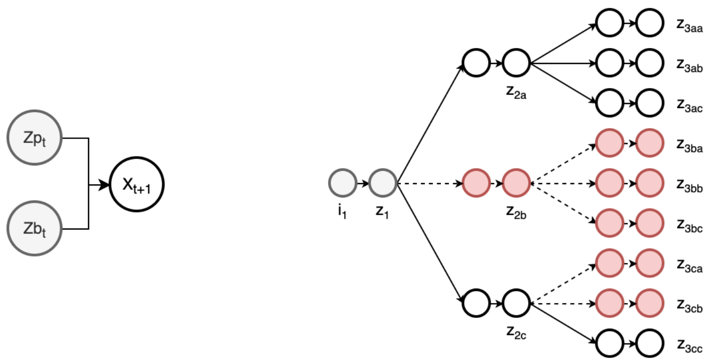
FrameReject The player input at the current timestamp, and consequently the game state in the current timestamp, is used to measure the probability of the input in the next timestamp, and consequently, the game state for the next timestamp. The probabilities will be measured for every possible states within a certain time frame. As we need to process an exponential number of states, this process is done in a multiprocessed way such that server processing time does not suffer. After acquiring the probability for all possible states, we then drop the states which has a low enough probability measure and generate frames from the remaining states. This solves the problem of input prediction handling, as we only send the frames which are likely enough to be the correct prediction. This can also solve the misprediction handling problem, provided we tuned our rejection threshold such that false rejection is miniscule.

Record Input Sequence and Select Correct Frames The saved input sequence will be used to select which of the frames sent to the client is the valid one by comparing the actual recorded input sequence vs the frame input sequence. We then choose that correct frame to be shown to the player.

2.3 FrameReject

To explain how FrameReject works, we will start by explaining how we designed the Probabilistic Graphical Model (PGM) for the related variables in our application. This is illustrated in figure 4a, where X_k is the player input at time step k and Z_k is the game state at time step k. In the case of Breakout, the game state Z_k is the paddle and ball states Zp_k and Zb_k.

We constructed the PGM at one timestep as shown in Figure 4a because player input in the next time step X_{k+1} is based on the current time step's state Z_k. We modeled the input X as a categorical distribution consisting of 3 different classes which corresponds to the navigational input: {Right, Left, No Operation}, while for the ball and paddle state Z, we quantized the coordinates of each of them into certain classes, and modeled them each as a



(a) PGM of FrameReject

(b) FrameReject state diagram over all possible states in three time steps.

Figure 4: PGM of FrameReject, where X is the player input and Z is the game state.

categorical distribution. How these states are acquired and quantized is described in more detail in the next subsection about data processing.

Through this PGM, we can see that to estimate the probability of the next timestep, we only need to evaluate the conditional probability:

$$P(X_{t+1}|Zp_t, Zb_t)$$

The parameters for this probability distribution is easily acquired through Maximum Likelihood Estimation.

Having acquired the parameters for the probability distribution represented by the PGM, we move to the next step - which is applying this representation for all the possible states in a certain number of time steps. This is illustrated in Figure 4b, where we plotted all possible state branches for 3 time steps. Note that each of the illustrated state branches is meant to be evaluated as an individual PGM.

The factorization for each state branches can be acquired directly by observing the PGM. For example, the factorization for the state at the last time step is as such:

$$P(X_{3ij}|Zb_1, Zp_1) = P(X_{3ij}|Zb_{2i}, Zp_{2i})P(X_{2i}|Zb_1, Zp_1)$$

Note that in our case, the relationship between X_i and Z_i is deterministic, as the current input will directly affect the resulting game state. Thus, we can assume that $P(X_i|Z_i) = 1$ for any i.

Once we have calculated the probability for each of the states, we can set a certain probability threshold with which we can reject the states. This is illustrated in Figure 4b, where red-colored nodes are variables whose probabilities are lower than the threshold, and thus can be rejected. The remaining states are used to generate the frames, which are then sent to the client.

Finally, we also need to take into account the processing time. As can be seen from Figure 4b, by increasing the number of predicted time steps, the number of states explode exponentially. Thus, to increase the efficiency of our model, some methods were required:

- Instead of evaluating probability of every possible states at the last time step, we can instead start our evaluation from the root, to the next time step, up until the last time step. This way, if some state in the middle time step has a low probability, we can disregard the rest of the consecutive time steps of this state branch. For example, observe X_{2b} in Figure 4b, which was rejected, and consequently all of its child states are rejected.

- Multiple branches are evaluated in parallel through multiprocessing.
- Possible states per time steps is limited by input shifting.

2.4 Dataset Processing

We resort to computer vision to acquire the states of the ball and paddle, as we do not have direct access to the specific game states. In practice, the game state objects which are relevant to track are usually marked by the game developers [9].

The game frame at a particular time step is split into three segments: (1) the ball segment with black background, (2) the ball segment with colored bricks and (3) the paddle segment. As the ball and paddle are red, we identified the coordinates of red pixels in all of the frames with black background. In the segment with the colored bricks, we cannot use this method. So instead we identify the black pixels and compare them to the black pixels from prior frames to determine the current position of the ball.

After identifying the ball and paddle coordinates, we compare the current frame coordinates to the previous frame coordinates to determine if the ball and paddle are either moving left, right, or stagnant. We then use these movements as the ball and paddle states. Finally, we save all of the ball and paddle states for each frame, along with their corresponding user inputs.

The dataset was collected using the locally-run game client for 1, 3, 5, 7, and 10 episodes. We used a locally-run game client to get what players will input when there is no delay. Each episodes lasts for an average of 51.15 s and generated on average 1530 data points each.

3 Experiments

We note that the we did not finish the full FrameReject solution, as we met with some difficulties that will be discussed in the next section. As such, what we present here is our experiments on the separate components, specifically the game/client server and the model.

We profiled the game client/server which is capped to run at 60 fps. The most time consuming process was that of input sending and frame sending, which each takes 16 ms and 13.8 ms, for a round trip time of 29.8 ms.

To observe the effect of delay on player performance, we played the game 25 times for both the game was played 25 times for cases without delay, and cases with delay. We noted an average gameplay time of 51.15 sec and 43.01 sec respectively, or a percentage increase in gameplay time of about 16%. This means that players last longer when playing the game without delay. Though this is not a comprehensive enough experiment, we observe that there is some indication that delay affects player performance.

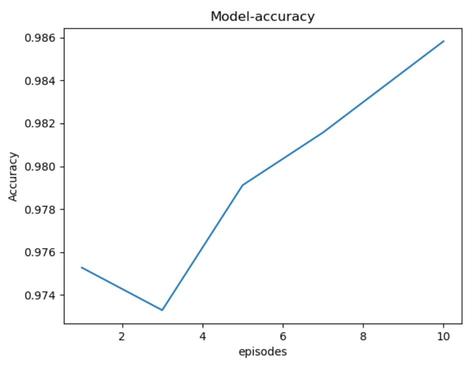


Figure 5: Model's accuracy of predicting future user's input

As for our model, we constructed the PGM as defined in the previous section. We trained the PGM using Maximum Likelihood Estimator, using a variable number of episodes as collected during the data collecting and processing step.

A test data from a separate episode was then used to check the accuracy of the model's prediction. This accuracy was calculated frame by frame by comparing the predicted user input with actual player input. The graph in Figure 5 shows that the accuracy of the model increased when the number of dataset episodes is increased, with the best accuracy of 0.986 using 10 episodes of training data.

4 Conclusion

4.1 Evaluation

Much of the difficulty with implementing the full FrameReject solution stems from the strict time frame requirement of the game loop, coupled with the need to stream the continuous game loop over the network in real time. This leads to the requirement of making the sending and receiving of the input and frames as efficient as possible, while not disturbing the main game loop. We had to explore compression and multiprocessing methods, but ultimately was not able to implement all of the required infrastructure on time.

Nevertheless, we were able to train our FrameReject model, and evaluate its performance over non-live game data. Our implementation resulted in an accuracy of 0.986 in the best case. We note that this experiment is not comprehensive enough, as we still need to test by different users, different delay conditions, etc. But, this result prove that there is potential in our approach. We deduct that our model resulted in a high accuracy because we are predicting navigational inputs, which tend to be the same over a period of movement. Implementation on impulse inputs might not result in accuracy as good as we achieved.

Although we have not tested FrameReject on the branching future states, the high accuracy that we received suggests that the model will not reject the correct frame most of the time.

With that said, we notice a number of potential problems with the FrameReject approach:

- Misrejection may cause visual inconsistencies.
- Since this method does not negate the need for sending of multiple frames, a reasonably good internet connection is still required.
- As it depends on specific game's states, remodelling is needed for different games. More complex games with complex state-input interaction might result in a higher rate of misrejection.

4.2 Improvements

As we did not successfully complete the application we have designed, there are a number of things that still needs to be done:

- Implement selective rendering of frames, compression and sending of multiple possible frames, input shifting, and multiprocessing of the FrameReject branching states prediction.
- Implement FrameReject on the branching states.
- Test the solution on multiple different delay settings, comparing the user experience between using FrameReject vs not using it for different delays.
- Test the solution with training and testing data from different users.

We can also further extend the scope into a more complex application context, such as:

- Implement FrameReject on impulse inputs too rather than only on navigational inputs.
- Implement FrameReject on more complex games.

References

[1] B. Matthew, "Cloud Gaming: Why It Matters And The Games It Will Create — MatthewBall.vc", MatthewBall.vc, 2020. [Online]. Available: https://www.matthewball.vc/all/cloudmiles. [Accessed: 22- Apr- 2022].

[2] "US Video Gaming Industry in 2022: Gaming Devices Video Game Content Viewership Trends", Insider Intelligence, 2022. [Online]. Available: https://www.insiderintelligence.com/insights/us-gaming-industry-ecosystem/. [Accessed: 22- Apr- 2022].

[3] C. Mark and C. Kajal, "Latency and player actions in online games — Communications of the ACM", Communications of the ACM, 2006. [Online]. Available: https://dl.acm.org/doi/10.1145/1167838.1167860. [Accessed: 22- Apr- 2022].

[4] "Stadia Streaming Tech: A Deep Dive (Google I/O'19)", Youtube.com, 2019. [Online]. Available: https://www.youtube.com/watch?v=9Htdhz6Op1I. [Accessed: 22- Apr- 2022].

[5] G. Gabriel, "Client-Side Prediction and Server Reconciliation - Gabriel Gambetta", Gabrielgambetta.com, 2022. [Online]. Available: https://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html. [Accessed: 22- Apr- 2022].

[6] S. Jarvinen, J. Laulajainen, T. Sutinen and S. Sallinen, "QoS-Aware real-time video encoding How to Improve the User Experience of a Gaming-on-Demand Service", Ieeexplore.ieee.org, 2006. [Online]. Available: https://ieeexplore.ieee.org/document/1593187. [Accessed: 22- Apr- 2022].

[7] H. Hong, D. Chen, C. Huang, K. Chen and C. Hsu, "QoE-aware virtual machine placement for cloud games", Ieeexplore.ieee.org, 2013. [Online]. Available: https://ieeexplore.ieee.org/document/6820610. [Accessed: 21- Apr- 2022].

[8] B. Anand and P. Wenren, "CloudHide — Proceedings of the on Thematic Workshops of ACM Multimedia 2017", ACM Conferences, 2017. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3126686.3126777. [Accessed: 20- Apr- 2022].

[9] K. Lee et al., "Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming", Microsoft Research, 2015. [Online]. Available: https://www.microsoft.com/en-us/research/publication/outatime-using-speculation-to-enable-low-latency-continuous-interaction-for-mobile-cloud-gaming/. [Accessed: 22- Apr- 2022].

[10] M. Bellemare, Y. Naddaf, J. Veness and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents", 2012. [Online]. Available: https://arxiv.org/abs/1207.4708. [Accessed: 20- Apr- 2022].

[11] Pygame.org. [Online]. Available: https://www.pygame.org/. [Accessed: 22- Apr- 2022].