CamerAI: Chase Camera in a Dense Environment using a Proximal Policy Optimization-trained Neural Network

James Rucks
Department of Informatics
University of Hamburg
jamesleerucks@gmail.com

Nikolaos Katzakis
Core Technology Group
Deep Silver Fishlabs
n.katzakis@dsfishlabs.com

***Abstract*—CamerAI is an autonomous chase camera that uses a PPO-trained neural network. Our results suggest that a simple, fully connected network with only two hidden layers of 128 neurons can perform basic chase camera functionality. We contribute a set of inputs and outputs with their respective coordinate spaces, as well as a custom reward function that can be used to train the network with reinforcement learning. Our findings highlight the importance of correct coordinate spaces, the need for a continuous reward function, normalization as well as timely resets during training to allow the network to explore its environment. We additionally present an evaluation of the output of CamerAI during 10 hours of chasing a bot that is randomly exploring a commercial video game map in which CamerAI was able to keep the player visible 96% of the time.**

I. Introduction

Chase cameras are used in third-person video games as a means for players to enjoy the game while keeping the controlled player avatar in view. Keeping the player in view in a dynamic environment with obstacles and enemies is a challenging problem that is, to this day, occupying game developers.

Early work by Vazquez[1] tackles the viewpoint problem by weighing viewpoints by their entropy. The underlying concept is that the higher the entropy of the viewpoint, the more interesting it is to the user. Their definition of entropy is based on the projected area and the number of faces in the viewpoint. The focus of our work is chasing a single target and although viewpoint entropy does not accomplish this, it could be combined with our method as one input to the network.

Ozaki [2] presented a fully automated virtual camera system, which is able to chase a subject with unknown behaviour through a dense environment in real-time. They outlined three objectives: Avoid collisions of the camera with obstacles from the environment, avoid occlusion of the subject by geometry between the target and the camera, and finally choose a good viewpoint to look at the subject. The goodness of a viewpoint is evaluated by calculating the viewpoint entropy [1]. The approach by Ozaki et al. delivers good results, for a problem that is comparable to ours, yet with high complexity and real-time computational cost. In addition, it is unclear if their approach could scale to track multiple players.

Lino [3] developed an efficient technique and demonstrated how to reduce the problem of on-screen positioning of multiple subjects from a 6D search to only a 2D search by proposing a *Toric manifold*. Following up on their previous work, they introduce *Toric space* [4]. It extends the toric manifold to a three-dimensional space used to place cameras in 3D space to capture a scene with two subjects. Despite delivering good results, the overlapping constraints approach is challenging to implement and they do not take dynamic obstacle avoidance or occlusion of the target into account.

A different approach is the definition and maximisation of certain parameters evaluating the current viewport based on a specific target. This technique is used in various papers [5], [2], [6], [7], [8] and described in most detail by the work from Ranon and Urli [9]. They experiment with a variety of properties such as size, framing, relative position and occlusion and find different ways to weigh them.

Finally, Xie [10] developed a semi-automatic system for quadrocopter videography. It is intended to enable novice users to get good captures of landmarks, by only setting the start and end viewpoints of the camera. However, like other works in aerial cinematography [6], [11], it is not intended for dense environments due to minimal obstacle avoidance.

Christie [5] identified requirements for controlling cameras in computer graphics from interactive approaches to fully automated solutions. Their work divides automated solutions into three groups: reactive approaches, optimisation-based approaches and constraint-based approaches. Reactive approaches rely on techniques borrowed from robotics and sensor planning. These approaches compute the behaviour of the camera based solely on visual properties of the image without the use of any search process. In constraint-based and optimisation-based approaches, the user specifies desired image properties for which the camera behaviour is computed using general-purpose solving mechanisms. Constraint-based approaches model the desired properties as a set of constraints to be satisfied. Optimisation-based approaches express the desired image properties as a function to be maximised. Our proposed system can be thought of as a combination of the latter two.

978-1-6654-3886-5/21/$31.00

In this work, we tackle the problem using a simple neural network trained using Proximal Policy Optimization (PPO) [12], [13], [14]. We identify sets of appropriate inputs and outputs and a reward function to enable training of the network. The resulting network, CamerAI, controls the camera to traverse a 3D game level and keep the player in view. Our approach, i.e. rewarding the agent for satisfying certain visual properties using reinforcement learning, results in a model that performs well when exposed to maps of similar complexity to those seen during training.

II. Training Environment

Among the challenges in using neural networks for a chase camera task is to create a suitable training environment for the neural network. The requirements include an environment that captures inputs, feeds the outputs back to the agent, and most importantly the design of a reward function that evaluates the camera perspective and allows the neural network to find the gradient to a good solution as quickly as possible.

In our implementation, the viewpoint had to satisfy the following constraints:

1) The camera should not collide with map geometry.
2) The player should not be occluded by geometry between camera and player.
3) The player should not be outside the frustum of the camera.
4) The camera should keep a reasonable distance to the player.

A. Training Levels

As a test environment for the intelligent spectator camera, we developed four levels. The design of the levels is based on two essential requirements:

Firstly, they should resemble the structure of actual computer game maps.

Secondly, the maps should portray environments of different complexity that are increasingly more difficult for the camera to navigate.

To fulfil our requirements, we developed four distinct levels, which are displayed in Figure 1. The levels are similar to those of Burelli et al. [15].

B. Player Bots

As a target for the chase camera, we created a bot player whose movements contain a certain amount of randomness. The bot selects a random point in a sphere of 20m around itself ($wander\_radius$). From this random point, the closest position on the NavMesh[1] is calculated and the bot travels to that waypoint. Upon arrival at this waypoint, it generates a new one and so on.

[1]A mesh that specifies which places on the map computer-controlled enemies can access (see Fig. 1c)

C. Spectator Camera

The image that users see on their screens is the output of a virtual camera that is controlled by CamerAI. It can freely translate on the x,y and z axes. Rotation is limited; we allow free yaw, letting the camera rotate $360°$, as well as limited pitch of $180°$. We applied these restrictions to prevent the system from outputting camera shots that are vertically inverted, e.g. a pitch of more than $180°$. Roll was locked to ensure a stable view of the game without disorientation to the spectators. We set the field of view to $90°$.

This virtual camera provides input for the neural network and receives output in the form of translation and rotation. However, this can lead to many changes of direction in a short time, causing the camera image to jitter. In our current implementation of CamerAI, the output of the network exhibited a high amount of jitter.

We stabilize the image by using two cameras. We refer to the first one as "AI Camera". It provides the inputs for the neural network and is also directly controlled by the outputs of the neural network. The second camera, referred to as the "Lerp Camera", closely follows the "AI Camera", but significantly reduces jitter. Its new position $L'_{Position}$ is calculated each frame from its previous position $L_{Position}$, and the position of the AI Camera $A_{Position}$, dependent on an interpolant $t \in [0, 1]$.

From now on unless Lerp is explicitly stated, "camera" refers to the AI Camera.

$$L'_{Position} = (L_{Position} - A_{Position}) \cdot t \tag{1}$$

For this linear interpolation, we use "Vector3.Lerp()" [16]. Similarly "Quaternion.Slerp()" works for spherical interpolation, which treats the vectors as directions. We use the latter for the rotation of the Lerp Camera. The result of this smoothing is shown in Figure 2.

III. Neural Network

A. Inputs

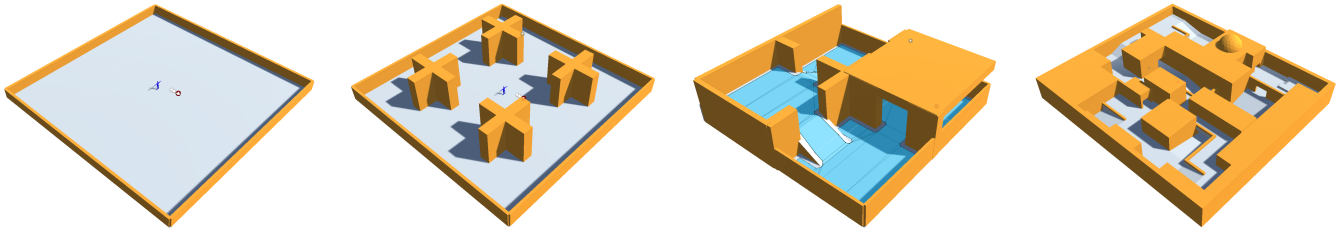
As input to the neural network, we collect observations from the training environment.

1) Player Location: Both *orientation* and *position* of the player needs to be transformed to the camera's coordinate space. The position is supplied as a vector $(x, y, z) \in \mathbb{R}^3$, while the orientation, of which we only need the Y-axis value, is collected as an Euler angle $0 \leqslant y \leqslant 360$.

2) Frustum Check: Another key piece of information is whether or not the player is inside the camera frustum.

For each of the eight corner points of the player's bounding box, we check individually, if it lies in the visible screen area of the camera, as seen in Figure 3. To do so, we transform its position $CornerPos$ from world space into camera space to $CornerPos^T$.

In Unity3D screen space, $ScreenSpace_{width}$ is defined on the X-axis and $ScreenSpace_{height}$ is defined on the Y-Axis. The Z-Axis represents depth in camera space. $CornerPos^T$ is



(a) Level I (b) Level II (c) Level III with navmesh in cyan (d) Level IV - de_dust2

Fig. 1: Level Design

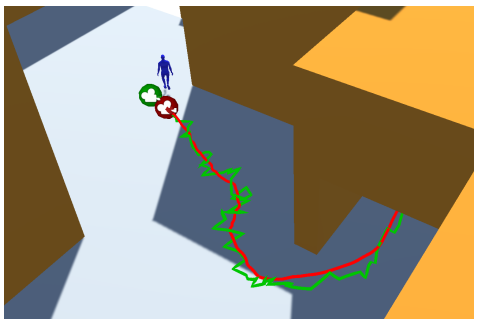


Fig. 2: Path of the AI Camera (Green) and Lerp Camera (Red)

Fig. 3: Frustum check

located in the visible screen area of the camera if the following three conditions are met:

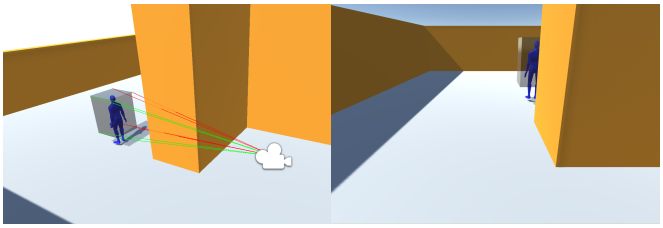
$$0 \leqslant CornerPos_x^T \leqslant ScreenSpace_{width} \quad (2)$$

$$0 \leqslant CornerPos_y^T \leqslant ScreenSpace_{height} \quad (3)$$

$$0 \leqslant CornerPos_z^T \quad (4)$$

We store the result of our visibility test in a list consisting of eight Boolean variables $V_{fru} = (r_1, \ldots, r_8)$ with $r_n \in \{0, 1\}$. During the iteration through the eight corner points of the bounding box we set each corner's respective variable in the list to 1 if it fulfils the Equations 2, 3 and 4, and to 0 if it does not.

3) Player occlusion by scene geometry: To prevent the player from being occluded by scene geometry, we test using the method of Ranon and Urli, by casting rays to the corners of a bounding box [9]. This should not be confused with the previously collected view frustum observations. Unlike the frustum check, all ray tests here will succeed even if the camera is facing away from the player (see Figure 4).



(a) Overview of the Scene (b) Camera Output

Fig. 4: Geometry occlusion test

Here we also store the result of our test in a list of eight boolean variables $V_{occ} = (r_1, \ldots, r_8)$ with $r_n \in \{0, 1\}$. During the iteration, through the eight corner points of the bounding box, we set each corner's respective entry in the list to 0, if the corner is hidden, and to 1, if it is not.

4) Object proximity rays: Information about nearby objects in the vicinity of the camera and target player is essential to find ways around obstacles.

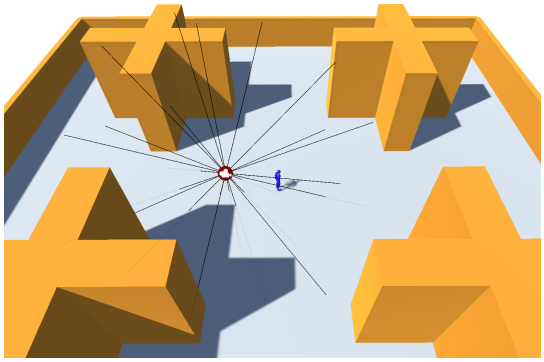


Fig. 5: Environment proximity Rays being cast.

First, we cast two rays of length l, one of them straight up and one straight down. Then we form a fan shape from n rays and duplicate it m times around the Y-axis. As a result, we get evenly distributed rays in all directions, as seen in Figure 5. In our implementation, values of $l = 10$, $n = 3$ and $m = 8$ resulted in the best performance.

Similar to rays from the camera, we cast rays from the player location to its surrounding environment.

We create two lists, one for $CameraEnvRays$ and one for $PlayerEnvRays$. Both are of length $2 + (n \cdot m)$ and each entry represents one ray. For each ray, we store the distance until the first impact with a collider. If the ray does not hit a collider, we record the maximum length l.

5) Collision of the Camera: Whether or not the camera collides with an object from the environment can be detected by checking the proximity rays cast from the camera, as described in Section III-A4. If the distance to the first collision is zero for every ray, then the camera must be colliding with geometry. This method does, however, require casting rays in all directions every time.

The reward function only uses the information of whether a collision occurs. Since this does not require the casting of multiple environment rays, we implement a second, more efficient method.

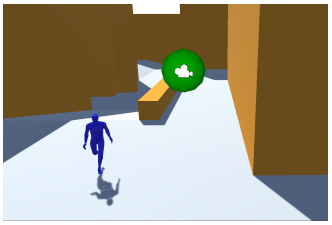


Fig. 6: Camera in Open Space

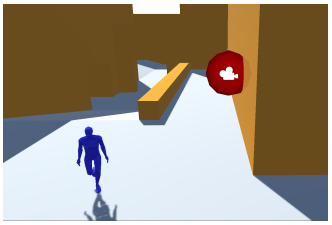


Fig. 7: Camera Colliding

We can check whether a sphere with centre p and a radius r touches or is inside of a wall of the map. We pass the current position of the camera to the collision checking method as p and $r = 1$. Based on the result, we set a Boolean variable C_{coll} to 1 if the camera collides with something (Figure 7), and to 0, if it does not (Figure 6).

6) Data Preprocessing: To reduce and stabilize training times, we preprocess the input data before feeding it into the neural network.

For rescaling a value x we use *min-max normalization* [17, p. 33-34]. This scales a value x with a minimum of x_{min} and maximum of x_{max} to a value $x' \in [a, b]$.

$$x' = a + \frac{(x - x_{min})(b - a)}{x_{max} - x_{min}} \tag{5}$$

For positions in world space, we scale each of the three components of the vectors to the interval [0, 1] with min-max normalization. First, we iterate through all colliders of the scene and extend a "Bounds" [16] object incrementally so that it encloses all colliders at the end. Then we obtain the minimum and maximum points of the bounds and use their components as a and b values for normalization.

Positions in camera space are not normalized.

We convert rotations from quaternions to Euler angles. Then, we normalize the components of the vectors, which are between 0 and 360 to values between 0 and 1 using min-max normalization.

Camera and player environment rays produce values between 0 and 20, which we also scale with min-max normalization. We additionally use one-hot encoding, which means that for every ray, we introduce a separate independent variable. Each variable represents an input neuron.

The checks for occlusion and frustum return lists of Boolean values represented by 0 and 1. For the eight values per list, we also use one-hot encoding, resulting in a total of sixteen variables.

7) Input Format: As inputs for the neural network, we use the observations of the environment in a format presented in Table I:

TABLE I: Inputs of the Neural Network

Observation	Value Domain	Coordinate Space
Camera Rotation	$\mathbb{R}^3$	World
Camera Environment Rays	$\mathbb{R}^{26}$	Camera
Player Position	$\mathbb{R}^3$	Camera
Player Rotation	$\mathbb{R}^1$	Camera
Player Occlusion Rays	$\{0, 1\}^8$	*independent*
Player Frustum Check	$\{0, 1\}^8$	*independent*

B. Outputs

The output of the neural network is applied as translation and rotation to the camera. Table II lists all CamerAI outputs:

TABLE II: Outputs of the Neural Network

Action	Value Domain	Coordinate Space
Translate Camera	$\mathbb{R}^3$	Camera
Rotate Camera	$\mathbb{R}^2$	Camera

It should be noted that PPO has been found to work best when the network outputs translation relative to the camera's coordinate system. This means that CamerAI tells the camera to "move left by x units.." or "translate upwards by Y units..". The network inference pass does not output positions in world coordinates. The output is directions in which the CamerAI agent should move.

C. Network Architecture

We chose an architecture that consists of two fully connected hidden layers, illustrated in Figure 8. In our final configuration, we use 48 input neurons. The hidden layers consist of 128 neurons each, and there are five output neurons. We use the default configuration of hyperparameters from the ML-Agents library [18].

D. Rewards

To calculate the reward that is awarded to the network at every training step, we rely on the observations of the environment that are already collected for the input of the neural network. The calculation of the reward is based on four core principles:

1) A high penalty should be given if the camera collides with geometry, as this scenario provides the viewer with

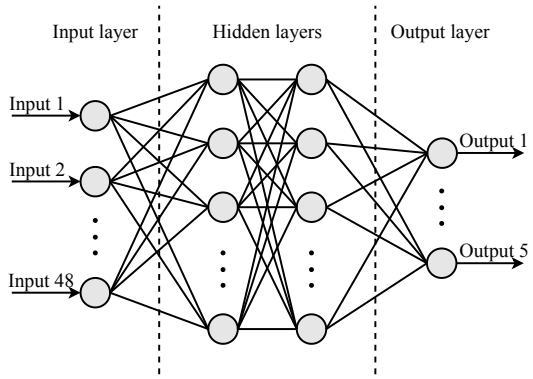


Fig. 8: Structure of the Neural Network

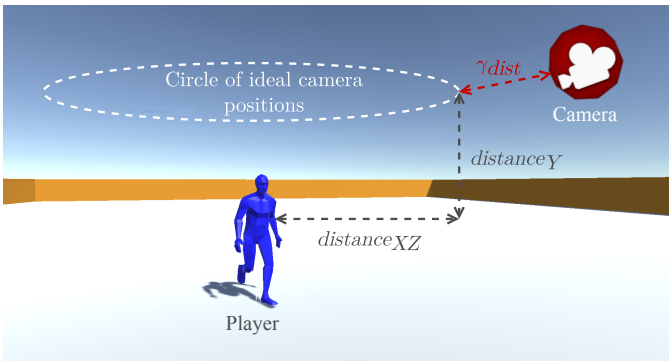


Fig. 9: Ideal Camera Position and γ_{dist}

an even worse shot than, for example, a viewpoint from which the player is partially occluded.

2) A minor penalty should be given if parts of the player are not in the camera's frustum, or are occluded by geometry, as this is not a desired viewpoint either. To check this, we count the number of corners of the player's bounding box that are in the camera's frustum or unoccluded.
3) A small reward should already be given when the player is fully visible, as this provides an acceptable shot.
4) A high reward should be given when the player is fully visible, and the camera is at the ideal distance from the player.

We introduce a tuple $\Gamma = (\gamma_{coll}, \gamma_{fru}, \gamma_{occ}, \gamma_{dist})$ containing the information of whether the camera is colliding $\gamma_{coll} \in \{0, 1\}$, how many bounding box vertices are in the frustum $\gamma_{fru} \in \{0, \dots, 8\}$, how many bounding box vertices are visible $\gamma_{occ} \in \{0, \dots, 8\}$ and the distance between the camera and its ideal position $\gamma_{dist} \in \mathbb{R}$ at the current time step.

Section III-A already describes how $C_{coll} = \gamma_{coll}$ is obtained from the environment for the neural network.

V_{fru} and V_{occ} are also described in Section III-A. γ_{fru} and γ_{occ} can easily be derived from them:

$$\gamma_{fru} = \sum_{x \in V_{fru}} x \quad \text{and} \quad \gamma_{occ} = \sum_{x \in V_{occ}} x \tag{6}$$

In order to calculate γ_{dist}, we first define an ideal position of the camera. We define an ideal distance of the camera to the player on the Y-axis $distance_Y = 1.5$, as well as an ideal distance on the XZ-plane $distance_{XZ} = 3.5$. These two specifications result in a circle of ideal camera positions above and around the player, as shown in Figure 9.

The reward should therefore be a continuous function proportional to the distance from the ideal camera position. For this purpose we define the function $R_{dist}(\Gamma) : \Gamma \mapsto [0, 1]$, which is a Gaussian function.

In an early implementation of the reward function, the distance reward function R_{dist} only specified a fixed area where the camera was supposed to position itself. If the camera was far away from its ideal position, it was not easy for the PPO algorithm to learn (via gradient descent) in *which* direction it had to move to get closer to the ideal position (and therefore highest reward). Only by introducing the Gaussian function for R_{dist} the camera learned to stay within the desired XZ- and Y-distance.

The function value is 1 when $\gamma_{dist} = 0$ and declines with increasing γ_{dist}:

$$R_{dist}(\Gamma) = a \cdot \exp\left(-\frac{(\gamma_{dist} - b)^2}{2c^2}\right) \quad \text{with} \quad a = 1,\ b = 0,\ c = 2 \tag{7}$$

Making the reward function continuous helps the neural network find an appropriate gradient during training and therefore learning how to reach a higher reward. Another option to make the reward function more continuous is to check the corner points of the player bounding box for being in the view frustum or for not being occluded by geometry. Instead of just measuring full occlusion or visibility with true or false, we look at each corner of the bounding box individually.

If the player is partially outside the frustum or hidden by geometry, we want to give a negative reward. Hence, we define the linear function $R_{vis} : \Gamma \mapsto [-1, 0]$ which scales γ_{fru} and γ_{occ} to 0 when the player is fully in the frustum and not occluded, and to -1 when the player is outside the frustum and fully occluded:

$$R_{vis}(\Gamma) = \left(\frac{\gamma_{fru} + \gamma_{occ}}{16}\right) - 1 \tag{8}$$

This continuous reward function is *key* for the learning process of the neural network. In $R : \Gamma \mapsto [-1, 1]$ we combine the previously defined subfunctions and calculate the final reward, which the network learns to maximize during training with the help of PPO:

$$R(\Gamma) = \begin{cases} -1 & \text{if } C_{coll} = 1 \\ \frac{2}{5} \cdot R_{vis}(\Gamma) & \text{if } C_{coll} = 0 \wedge (\gamma_{fru} < 8 \vee \gamma_{occ} < 8) \\ \frac{1}{2} \cdot R_{dist}(\Gamma) & \text{otherwise} \end{cases} \tag{9}$$

The summands and factors are used to express the "weight" of the subfunctions and were determined through experimentation.

E. Training Process

We conducted simultaneous single-agent training. This means that we train independent agents in parallel, who learn the same behaviour. We always train with ten agents at the same time as shown in the examples of the Unity3D ML-Agents framework. The advantage of this training method is that parallel training accelerates the training process and stabilizes the results.

For all actions, we use time steps of 0.02 seconds playing time and train at a rate of 100 times the game speed. Every agent independently calculates rewards, takes actions and if necessary, resets itself.

Every four time steps a decision is made, what action to take next. In each time step, the action selected by the last decision is executed and then evaluated by the reward function. Every 500 time steps a *training interval* is completed and the neural network is updated with new weights.

Following the completion of a training interval, it is not always necessary to perform a reset. We reset only if at the end of the training interval:

- the camera collides with an object from the environment,
- the player is not in the frustum of the camera, or
- or the player is occluded by geometry.

A reset results in the camera being automatically placed at the ideal Y- and XZ-distance behind the player. If none of the three reset criteria is met, the camera is not reset.

The motivation behind this is that the camera always has the same time to collect both positive and negative rewards within the set time intervals and can therefore easily recognize improvements. Avoiding a reset of the camera immediately after one of the three reset criteria is met is intentional. It is meant to give the network a chance to learn how to escape unfavourable situations before learning how to avoid them altogether.

IV. Evaluation

Part of the goals of this work was understanding how CamerAI performs in levels of increasing complexity (Figure 1). Therefore we considered the following training levels:

"Level I" only consists of a flat floor, which is bordered by walls. The camera only needs to turn in the right direction and stay within the playing area, to follow a player on this level. Keeping a suitable distance to the player can already be tested.

"Level II" has increased complexity. Four plus-shaped obstacles block the view of the camera to the player in certain positions. They can also lead to a collision of the camera with them.

"Level III" increases the complexity further. Since the ground was plane at all previous levels and there were no vertical obstacles, the problem of tracking the player could be solved solely through navigating the camera on a 2D plane. By introducing a plateau that can be reached by ramps and limiting the area to the top by a roof, vertical movement of the camera becomes necessary. This means that in order to track the player, navigation in 3D space is required in contrast to the previous two levels. Level III already contains all challenges which CamerAI was expected to master.

"Level IV" is a fully-fledged model of an original computer game map, with all the challenges of the previous levels, as well as other tricky elements such as broad slopes, narrow tunnels and thin walls. The geometry of the map is a replica of the Counter-Strike map "de_dust2". It is, to this day, one of the most popular Counter-Strike maps [19].

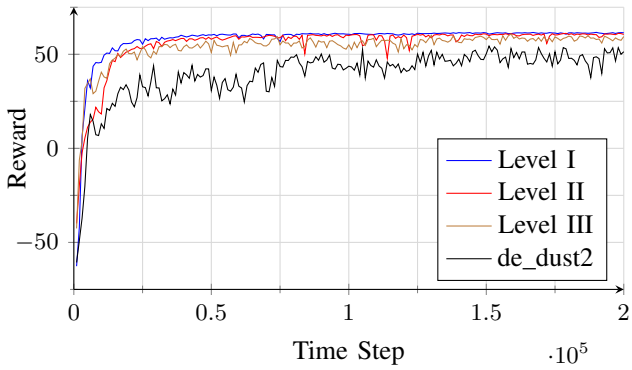


Fig. 10: Training Results on Levels I-IV. 200k time steps (i.e. far right of the plot) equals to 45 minutes training time.

To evaluate CamerAI we first train a neural network for each level for $2 \cdot 10^5$ steps (Figure 10). Following that, we let CamerAI follow a player bot around the map for ten hours. For each time step, we record whether the player is visible on the screen completely, partially or not at all. To fasten the process, we simulate the ten hours of game time at a hundredfold speed, just like during network training.

TABLE III: CamerAI's Performance after 10 Hours of Filming

Player	Level I	Level II	Level III	de_dust2
Fully in Sight	99.14%	98.72%	97.93%	96.41%
Partially in Sight	0.86%	1.26%	2.05%	3.52%
Not in Sight	0.00%	0.02%	0.02%	0.07%

The result, which is presented in Table III, shows that CamerAI is able to keep the player on screen for the vast majority of the tested sessions. On Level IV (de_dust2), CamerAI keeps the player fully on-screen over 96% of the time and spends less than 0.1% of the time in a viewpoint where the player cannot be seen at all.

In addition to this, CamerAI delivers superior results to state of the art 3rd person cameras when map geometry abruptly occludes the view (c.f. Video Figure).

Also, we discovered that the normalization of inputs presented in Section III-A6 led to a significant reduction of training time in Level IV (Figure 11).

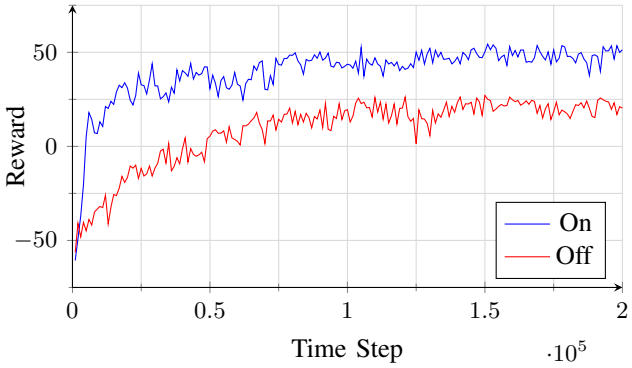


Fig. 11: Effect of Normalization on Training on Level IV. 200k time steps (i.e. 2 on the x axis) equals to 45 minutes training time.

V. Discussion

The evaluation results suggest that CamerAI performs well across our given set of environments in real-time. CamerAI is able to reliably keep a target player running around on a complex map on screen and avoid occlusion or collisions.

Our implementation does not divide the environment into cubic cells and run resource-intensive path-finding algorithms in 3D space. All inputs for the neural network can be broken down into primitive operations such as getting positions or casting rays, which are possible in all common 3D engines with low implementation and performance effort [16], [20], [21], [22].

Also, we see that in popular games, 3rd person cameras are unable to gracefully handle abrupt occlusions. We present two common scenarios in which CamerAI consistently delivers superior results (c.f. Video).

Despite the acceptable results, CamerAI also has some limitations. The network adapts well to the environment it has already been exposed to and learns to respond competently to small variations (i.e. a network trained on Level III - performs well on Level IV). Despite that, there are limitations to this which stem from the type of neural network configuration and type of training algorithm (PPO). Other types of configurations or training algorithms might be able to modulate their output faster to adapt to faster varying environments (abrupt stairs, pits, rapid onset-offset of occlusions).

CamerAI is only trained to keep a single player on screen while the environment in the background or interaction with other players is not taken into account. This remains as future work.

Finally, the algorithm's ability to provide enjoyable camera movement for the spectator is estimated using heuristics. The quality of the resulting camera movements should as a next step be evaluated with real users.

VI. Conclusion

CamerAI is a fully connected network with only two hidden layers of 128 neurons that can perform chase camera functionality. The CamerAI network architecture delivers acceptable results with only 8 minutes of training on a laptop computer, with maximum performance following 45 minutes of training. This work contributes a set of inputs and outputs, their appropriate coordinate spaces, as well as a carefully designed reward function that can be used to train the network with reinforcement learning. Our findings highlight the importance of correct coordinate spaces for the inputs and outputs, a continuous reward function, normalization of the inputs, as well as timely resets during training to allow the network to explore its options and its surroundings using raycasts. CamerAI was able to keep the player fully visible 96% of the time during a long run in a complex game map.

Given the proliferation and ease of use of machine learning frameworks, we believe that extending this work as well as designing better agents holds promise for the future of automated cinematography for 3D environments and physical drones. The future of game cinematography could potentially comprise of a number of specialized CamerAI-type agents, orchestrated by a "director" neural network.

Acknowledgment

The authors would like to thank Christophe Lino for advice and guidance early in the project. Also, Frank Steinicke for offering help with editing the paper.

References

[1] P.-P. Vázquez, M. Feixas, M. Sbert, and W. Heidrich, "Viewpoint selection using viewpoint entropy." in *VMV*, vol. 1, 2001, pp. 273–280.

[2] M. Ozaki, L. Gobeawan, S. Kitaoka, H. Hamazaki, Y. Kitamura, and R. Lindeman, "Camera movement for chasing a subject with unknown behavior based on real-time viewpoint goodness evaluation," *The Visual Computer*, vol. 26, pp. 629–638, 06 2010.

[3] C. Lino and M. Christie, "Efficient composition for virtual camera control," in *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2012.

[4] ——, "Intuitive and Efficient Camera Control with the Toric Space," *ACM Transactions on Graphics*, vol. 34, no. 4, pp. 82:1–82:12, Jul. 2015.

[5] M. Christie, P. Olivier, and J.-M. Normand, "Camera control in computer graphics," *Computer Graphics Forum*, vol. 27, no. 8, pp. 2197–2218, 2008.

[6] R. Bonatti, C. Ho, W. Wang, S. Choudhury, and S. Scherer, "Towards a robust aerial cinematography platform: Localizing and tracking moving targets in unstructured environments," *CoRR*, vol. abs/1904.02319, 2019. [Online]. Available: http://arxiv.org/abs/1904.02319

[7] H. Jiang, B. Wang, X. Wang, M. Christie, and B. Chen, "Example-driven virtual cinematography by learning camera behaviors," *ACM Trans. Graph.*, vol. 39, no. 4, Jul. 2020. [Online]. Available: https://doi.org/10.1145/3386569.3392427

[8] L. Burg, C. Lino, and M. Christie, "Real-time anticipation of occlusions for automated camera control in toric space," *Computer Graphics Forum*, vol. 39, no. 2, pp. 523–533, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13949

[9] R. Ranon and T. Urli, "Improving the efficiency of viewpoint composition," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 5, pp. 795–807, May 2014.

[10] K. Xie, H. Yang, S. Huang, D. Lischinski, M. Christie, K. Xu, M. Gong, D. Cohen-Or, and H. Huang, "Creating and Chaining Camera Moves for Quadrotor Videography," *ACM Transactions on Graphics*, vol. 37, pp. 1–14, Aug. 2018.

[11] B. F. Jeon and H. J. Kim, "Online trajectory generation of a MAV for chasing a moving target in 3d dense environments," *CoRR*, vol. abs/1904.03421, 2019. [Online]. Available: http://arxiv.org/abs/1904.03421

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[13] S. Rudolph, S. Edenhofer, S. Tomforde, and J. Hähner, "Reinforcement learning for coverage optimization through ptz camera alignment in highly dynamic environments," in *Proceedings of the International Conference on Distributed Smart Cameras*, 2014, pp. 1–6.

[14] W. Hönig and N. Ayanian, "Dynamic multi-target coverage with robotic cameras," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 1871–1878.

[15] P. Burelli and G. N. Yannakakis, "Combining local and global optimisation for virtual camera control," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. IEEE, 2010, pp. 403–410.

[16] Unity Technologies. (2019) Unity user manual (2019.2). [Online]. Available: https://docs.unity3d.com/Manual/

[17] M. Kantardzic, *Data Mining: Concepts, Models, Methods and Algorithms*, 2nd ed. John Wiley & Sons, Inc., 2011.

[18] Unity Technologies. (2019) Unity ml-agents toolkit (beta). [Online]. Available: https://github.com/Unity-Technologies/ml-agents

[19] David Johnston. (2019) Making of: Dust 2. [Online]. Available: https://www.johnsto.co.uk/design/making-dust2/

[20] Epic Games. (2019) Unreal engine 4 documentation. [Online]. Available: https://docs.unrealengine.com/

[21] Crytek. (2019) Cryengine v manual. [Online]. Available: https://docs.cryengine.com/

[22] J. P. Freiwald, N. Katzakis, and F. Steinicke, "Camera time warp: compensating latency in video see-through head-mounted-displays for reduced cybersickness effects," in *Proceedings of the 24th ACM Symposium on Virtual Reality Software and Technology*, 2018, pp. 1–7.