The CSI Journal on
Computer Science and Engineering
Vol. 17, No. 2, 2020
Pages 26-33
**Regular Paper**

# A Video Game Testing Method Utilizing Deep Learning

Mohammad Reza Taesiri[1] Moslem Habibi[2] MohammadAmin Fazli[1]

[1]Department of Computer Engineering, Sharif University of Technology, Tehran, Iran
[2]Department of Industrial Engineering, Sharif University of Technology, Tehran, Iran

## Abstract

Computer video games must pass different types of tests before release. Yet most products in this multibillion-dollar industry still exhibit various compatibility problems when run on end consumers' computers. In this work, we propose a new automated testing method which utilizes deep convolutional neural networks to test video game compatibility with target runtime environments. This will result in better support for various computing environments that run video games and a reduction of the effort needed for testing them. Our method executes tests both on local computers and the cloud. Locally, a game tester will test the video game with normal testing routines. After that, these tests are automatically replicated on the cloud, running the video game on different environments. With the help of two convolutional neural networks, corrupted frames of the game containing artifacts are automatically discerned, and by comparing the local execution to the ones on the cloud, the corresponding problematic Draw Calls are determined. These are then used as a basis for comparison in order to determine the root cause of the graphical issue.

**Keywords:** Video Game Testing, Automated Testing, Software Testing, Deep Learning, Convolutional Neural Networks.

## 1. Introduction

The development process of video games varies in many aspects compared to other software products. Different development teams usually use their own specific test processes based on the type of game they are creating. One common problem that all teams deal with after a game is published is incompatibility with consumers' personal computers. In recent years, there have been many instances of popular video games that needed so-called "day one patches" shortly after release because of such incompatibility problems. For example, in 2015, Steam temporarily discontinued sales of the PC version of 'Batman: Arkham Knight' due to such issues and bugs[1].

Video games can be evaluated from different viewpoints. A video game is typically created using a Game Engine composed of various components including the Rendering Engine, Physics Engine, Particles System, Audio & Video Libraries, etc. Such components can be tested individually using traditional software testing methods. Additionally, some components can use specialized testing procedures unique to them, such as graphics tests for the rendering engine.

Game developers who create their products by utilizing the components mentioned above have a hard time automatically detecting graphical bugs which occur on some specific runtime environment configuration. Identifying such visual bugs might be easy for a human, but because of the large number of elements that make up a PC's runtime configuration, such manual detection is time-consuming and tedious. Also, as many modern computer games consist of numerous graphical objects in each frame, visual glitches cannot all be detected by the naked human eye.

A typical method to detect such problems is to make use of traditional image quality metrics, but such metrics have been shown to possess shortcomings in detecting graphical glitches in many situations.

In this paper, we propose a novel solution for automated testing of video games based on deep learning methods. Our main goal is to detect graphical glitches, i.e., corrupted frames, that occur due to the different runtime environments in which a game is executed in.

In the next section, we will briefly introduce related works in this field. Then in Section 3, our proposed method is explained and its various steps are discussed. In the subsequent section, we will explain our evaluation method and

the obtained results. Finally, future work in this field is expanded upon in the Conclusion section.

## 2. Related Work

In this section, we introduce related works in literature dealing with video game testing. In [2] several new methods for usability testing in video games are introduced using data gathered from users, a technique also emphasized in [3]. Diah et al. [4] also deal with usability testing, utilizing the observation method in this regard for a specific category of video games, i.e., educational video games.

Automated testing of video games has also gathered much attention, with [5] introducing a Smoke Testing mechanism for video games, while the automated testing method proposed by [6] is process based and platform independent. A large part of the QA activities of any video game development outfit is beta testing. Schaefer et al. [7] propose an automated testing method which can reduce the time spent on beta testing, introducing a framework for this purpose called Crushinator. Testing games at earlier steps of the software development process is discussed in [8], which proposes a state-based technique to test games at the prototype stage.

An important recent trend in this field is the use of AI and deep learning methods in video game testing. [9] provides an overview of research in this field. The authors show that the vast majority of analysed approaches rely on playtesting to product results, with work in this field shown to fall into one of three categories, human imitation, scenario-based, and goal-based approaches. Researchers in [10] make use of artificial intelligence agents and machine vision algorithms to devise a semi-automated game testing technique, while Chan et al. [11] use genetic algorithms to detect undesirable behaviour in video games. Folan et al. [12] have proposed Wuji, an automated game testing framework, which works by using deep reinforcement learning alongside multi-object optimization to automatically detect different category of bugs, from crash bugs to user experience ones. Bergdahl et al. [13] also use DRL, but to augment traditional scripting methods, especially in edge cases hard for humans to simulate. Researchers in [14] propose an approach for detecting graphical anomalies in video game images. Their method, which makes use of CNNs, is able to detect 88.1% of the glitches with a false positive rate of 6.3%. In [15] the researchers use deep neural networks to improve video game graphics in real time. Their results show how graphical artifacts can be manipulated and enhanced by AI methods, which can be used in the context of testing computer game graphics.

Other work deal with Black Box methods for game testing, including [16] which uses scenario-based testing methods to evaluate video games without information on their inner workings. The authors of [17] concentrate on testing online video games by proposing an online game testing tool, EasyQA to create virtual gamer loads for the test environment. Others such as [18] make use of model-driven testing techniques, used to test a wide range of software products including enterprise software, targeting the popular platform genre of video games. Another important test category for software products is Regression Testing where developers make sure that new changes do not introduce side effects into previously checked components. An automated regression testing method customized for video game development is presented in [19]. Lastly, [20] makes use of $LTL - FO^{+}$, an extension of linear temporal logic, to find bugs in video games by monitoring their runtime behaviour.

## 3. Proposed Method

As mentioned before, the video game testing method we propose in this paper makes use of deep learning methods to detect graphical corruptions in rendered video game scenes. As we will show, this is done at the Draw Call level, which are commands that tell the Graphical Processing Unit (GPU) to draw a certain set of polygons. Initially, the game developers or testers will run the game on a local machine, testing the game with traditional methods. At this step, we capture the user's input commands (i.e., user activity) on the local machine. This must be done in such a way so that different runs of the game produce the same output, that is the graphical output must be deterministic. For this, we make use of the algorithm suggested by Cuervo et al. [21], which limits sources of non-determinism, specifically system time, random number generators, and the game's music.

For the next step, different instances of the same game must be run on different runtime environments. To efficiently and rapidly create such environments with different settings we use virtualization technologies, such as KVM and QEMU, which are prevalent in cloud computing solutions. These cloud game executions are also made deterministic using the above-mentioned algorithm.

After this step, our proposed method will then check the output of each video game instance, searching for possible corruptions in the rendered frames using two convolutional neural networks. If any corruption is found, we store the relevant Draw Calls at both the reference execution (local) and the faulty cloud one. For our last step, we compare the results in order to find the source of the encountered glitch at the Draw Call level. The overall structure of our proposed method can be seen in Figure 1. To summarize, our approach contains 3 main phases:

- A local reference instance of the video game alongside several cloud instances are run, where their graphical output is deterministic.
- Corrupted frames in each execution of the video game are detected using a convolutional neural network (CNN) and their Draw Calls are stored.
- The Draw Calls of the corrupted frames are compared to the Draw Calls of equivalent frames from the reference execution so that the source of the corruption can be identified.

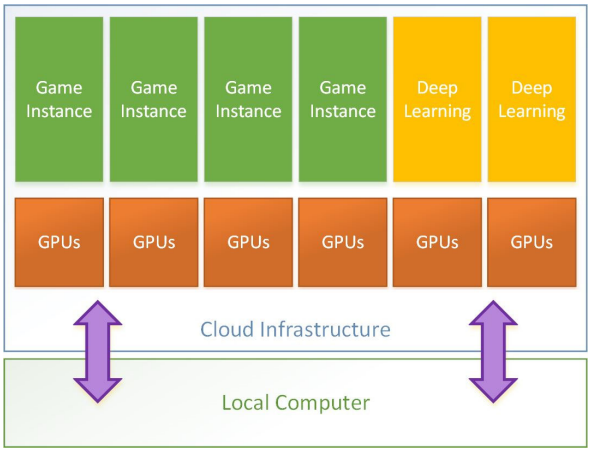


Figure 1: Top level view of proposed video game testing method.

An important question that arises here is why there is a need to use neural networks for this purpose and whether traditional image quality metrics could not be used in this regard. The short answer to this question is that the rendering of graphical frames in a video game is highly dependent on the game's user-defined graphics settings, and therefore a broad spectrum of valid frames for a given scene is possible based on the values given to these settings. As an example, we can see different renderings of the same scene in Figure 2, obtained from one of the Unity game engine's [22] sample projects. Here, rendering 1 is the reference rendering with the highest graphical fidelity. Renderings 2 and 3 are valid, but with graphical settings set to lower values (settings such as Shadow, Screen Space Ambient Occlusion, etc.), while rendering 4 is corrupted (the pink wooden objects).

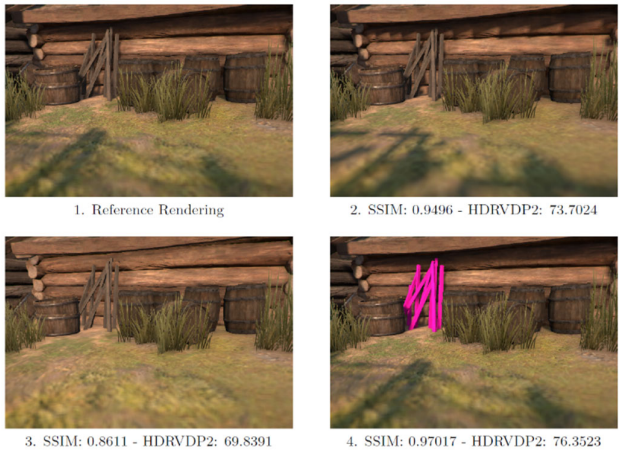


Figure 2: Different renderings of the same scene. Here renderings 1 to 3 are valid but at different quality settings. Rendering 4 is invalid as it contains corrupted artifacts.

Typically, in the video game industry and also computer graphics literature, traditional image quality assessment metrics are used to compare and measure degradation, similarity and corruption in different versions of an image.

SSIM [23] & HDR-VDP-2 [24] are two full-reference quality metrics commonly used in this regard. We will give a brief description of these metrics below and show why they do not perform as desired in situations such as the example renderings described above:

- SSIM or structural similarity index is a classic algorithm used in computer graphics, especially in graphics regression tests which consider image degradation as a perceived change in structural information while also considering perceptual phenomena.
- HDR-VDP-2 is a visual metric that compares a pair of images (a reference and a test image) and predicts visibility (the probability that the differences between the images are visible for an average observer) & quality (the quality degradation with respect to the reference image).

In Figure 2 we can see the values of SSIM & HDR-VDP-2 for different renderings. A value closer to 1 for SSIM (and closer to 100 for HDR-VDP-2) shows a rendering that more closely matches the reference one. The problem here is that these metrics' values for renderings 2 & 3, which are valid renderings, are lower than rendering 4, which is not desirable, as rendering 4 contains corrupted pixels, being visually not acceptable. This shows us that existing traditional metrics have shortcomings in detecting such cases of graphics artifacts. This adheres to the findings of Zhang et al. [25] which show that deep learning methods can easily outperform traditional image quality metrics assessing the perceptual similarity between different images.

## 3.1. Deterministic Replays

As we need each instance of the game running on multiple computers to be identical, their graphics output should be deterministic. If in the process of duplicating user inputs from the local run to the cloud even one frame is missed (e.g., because of delays), the state of the two video game instances may be entirely different, each showing a different graphical output. This limits our method's ability to compare frames from different runs, as such frames must be identical to detect any corruptions. Depending on the game and the game engine it uses, the ability to replay runs might be supported. For the case where such a feature is not available, we use a method called “Deterministic Replays", based on an algorithm proposed by [21]. By making small changes in the game engine code, this method makes the output of a game deterministic. One source of indeterminism not considered there is the sensitivity of the output to the time between the rendering of two consecutive frames. This plays a vital role in calculations relating to a game's physics engine. To resolve this problem, we store all frames alongside their inputs and time-stamps in a table called the Events Table. Games being run on the cloud obtain this table at fixed intervals and advance the game execution based on its entries.

## 3.2. Comparing Two Runs

As we know, the output of video games is graphical images or frames. Our proposed testing method deals with bugs and issues that corrupt such rendered images. This may happen because of reasons such as the use of specific functions that are only available on certain VGAs or specific versions of an OS. Another reason might be problems with one or more buffers when rendering an image (i.e., corrupted rendering

buffers) due to programming mistakes. In our proposed testing method, we use DenseNet [26], a CNN, to classify frames as corrupted or healthy. Here we only make use of the final graphical output (Back Buffers). The process of analyzing such output frames is fast, but without considering other rendering data, finding the root cause of the corruption is difficult, e.g., objects not directly visible in the frame may impact the graphics output. Also, glitches relating to lighting and shading of scenes are very prevalent here. Therefore, our method relies on identifying Draw Calls that play a role in corrupting the graphics output.

To render a scene and create the final output image it is necessary to move data such as meshes, textures, materials, etc. from the main memory of the computer to the graphics card memory. Next, the CPU will command the graphics card to render an object on the scene using Draw Calls. The total number of Draw Calls needed to render a scene depends on the complexity of the scene itself, and in some cases, over 30000 draw calls might be used to fully render a scene. Figure 3 depicts the rendering process of a scene with around 4000 draw calls. An important point that affects the appearance of the scene is the order of the draw calls. This order is determined by the rendering algorithm and the game engine architecture. One of the major assumptions in our method is that the game engine uses a total order for the draw calls. Our observation shows that many of today's most widely used game engines such as the Unreal Engine [27] and Unity [22] have this property.

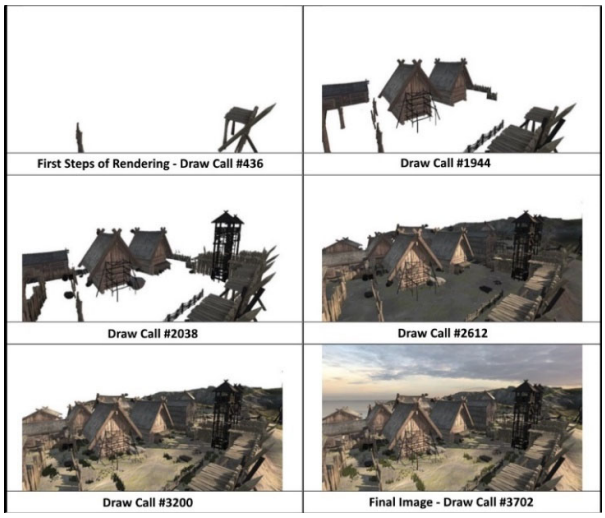


Figure 3: Rendering process of a frame by a game engine.

When an output frame is identified as corrupted by the DenseNet network, we need to obtain the Draw Calls used in the execution. This can be done using a specialized layer placed between the game and the graphics (API) which receives function calls from the game and stores them before passing them on. Many implementations of such layers exist mainly for troubleshooting purposes. In this work, we use the open source RenderDoc tool [28]. Now that we have access to the Draw Calls, we must match them in different executions. This is because, for a variety of reasons, a draw call in the local execution may not be called in some of the cloud instances. We name such draw calls as Missed Draw Calls. A missed draw call may be a sign of graphical problems, but this is not always the case. For example, reducing the graphic details setting of a game leads to such missed draw calls.

If in the game instances run via our testing method, a draw call is detected for which no corresponding draw call exists in the local execution, such a draw call is called an Orphan Draw Call. One reason for this such draw calls is the Level of Details (LOD) algorithms.

To properly match draw calls from two separate executions we can use special commands of the graphics API used for troubleshooting (e.g., D3DPERF_SetMarker in DirectX) and add extra information to each draw call. Such mechanisms are not applicable to all game engines. Therefore, we propose an automated method to find a match between draw calls; we iterate the draw calls in order of their number and insert them into a hash table. As the orphan & missed draw calls are present in one execution and no match exists for them, we use the final output image to compare them. We first detect the bounding box of these draw calls on the output frame and compare the pixels inside for local and cloud executions. If these executions use different settings when run, a large number of orphan & missed draw calls will be encountered. To reduce this problem all instances of the video game must run using the same settings. Also, as some draw calls affect a small amount of the pixels in a frame, we can omit them to speed things up.

## 3.4. Identifying Corrupted Frames Using CNNs

In a typical CNN architecture, several Convolutional and Pooling Layers are used in a sequence to reduce the width & height of the image and increase the depth of features. Such use of many layers can cause issues such as the Vanishing & Exploding Gradient problem when using backpropagation. This hinders the learning ability of the network. At the same time, as the number of layers grows, the network's capacity also increases, and we expect the network to be able to extract more features and learn the Identity Function. This is the main reasoning behind the ResNet network[29]. Here, by adding Skip Connections between different network layers, many of the issues mentioned above are mitigated. The DenseNet network builds on this idea by using several subnetworks where the skip connections continue beyond consecutive layers. In other words, not only is there an input to layer $L_i$ from layer $L_{i-1}$, but the output of previous layers are also present as inputs to this layer. This reduces the problem of vanishing & exploding gradients, enhances the propagation of features, enforces reuse of features, and decreases the total number of the network's parameters.

The DenseNet network is trained on the ImageNet dataset [30] and needs tuning for use in other areas. Therefore, 3 Affine layers measuring 2048, 1024, and 64 with the RELU (Rectified Linear Unit) activation function are added to the end of the neural network, with classification at the last layer done using the Softmax algorithm. To train the network, we freeze the first 169 layers of the network and only train the parameters of the newly added layers. To be able to do this, we first need to collect a large number of healthy & corrupted frame samples. Fortunately, social platforms such as Youtube and Twitch provide video content from which healthy frames can be collected. For corrupted frames, we crawled Youtube and Reddit to find video clips that have keywords showing they contain graphical bugs. All in all, we collected 5000

corrupted frames and 5000 healthy ones. The collected frames were high resolution (up to 1920x1080), therefore to increase accuracy & speed up processing each frame was sliced into subframes measuring 256x256 pixels. These were then labeled based on their pixel content. We also omitted subframes which did not contain any valuable features, such as the player webcam feed or overlaid text. An example can be seen in Figure 4, where the omitted subframes are shown in blue.

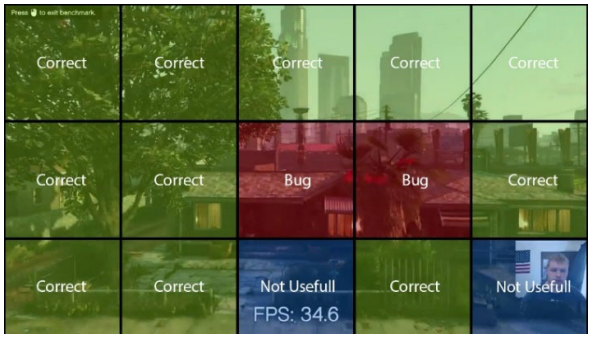


Figure 4: Slicing a single frame into multiple subframes. Subframes are either correct, corrupted (buggy) or are emitted for not including useful data.

## 3.4. Testing Method Procedure

After different instances of the video game have been run on the cloud, the captured frames are sent to the DenseNet to detect corrupted frames and store their draw calls. As most games run at 30 to 60 FPS (frames per second), the performance of the DenseNet network must be high enough for such inferences. We can either use variations of this CNN, which increase performance at the cost of losing some accuracy, or reduce the number of frames. The time gap between the two frames is very small, and therefore, two consecutive frames might not have much difference among them. To evaluate the difference between two consecutive frames we can use metrics such as the Mean Square Error (MSE), Normalized Root Mean Square Error (NRMSE), Peak Signal to Noise Ratio (PSNR), and Sum of Absolute Differences (ABSDIFF). Each metric performs best under different circumstances. In [31], a similar problem to ours is considered, and MSE is shown to provide reasonable performance. We also use MSE to evaluate the difference between two frames, denoting the parameter $\delta_{diff}$ as the difference threshold between two frames. $\delta_{diff}$ is one of the parameters that we must decide on.

Eventually, when a frame is detected as corrupted, its related draw calls must be stored. But as the frame is fully rendered, we do not have access to its constituting draw calls. Two solutions exist for this; storing the draw calls inside the frame (or consecutive frames) or recovering the corrupted frame. In the first case, the command to store the draw calls in the next $\delta_f$ frames is scheduled. When the marked frame is encountered, the draw calls are stored in both the local video game instance as well as the corrupted cloud one so that they can be compared. $\delta_f$ is another parameter which must be specified in our method, which can be difficult to do. This solution is fast, but if only a single frame is corrupted, it cannot store its draw calls. The corruption of a single frame may be negligible in a real-world setting, but still using the first solution decreases our method's overall accuracy.

In the second solution, we mark the corrupted frames and store all input provided to the game in a table. After the testing phases are finished, the game is run for a second round with the input to the game read from the table. When encountering previously marked frames the draw calls are stored. This solution also has its drawbacks, i.e., the frames in the second round might not precisely match those rendered in the first round. To mitigate this, we can store a window $\delta_w$ of frames neighboring the marked frames. This can increase accuracy but results in an increased processing load. We must also decide on a value for the parameter $\delta_w$. Based on our observations using both proposed solutions at the same time increases accuracy. We will discuss value selection for $\delta_f$ and $\delta_w$ more in-depth in subsequent sections.

After storing the draw calls of the corrupted frames, the comparison phase starts. We first omit the draw calls which have updated pixels outside the area of the corrupted image. Next, by matching the remaining draw calls the difference between their output pixels is calculated and classified using the second DenseNet network. Here the DenseNet network determines the acceptable threshold for changes between two draw calls. In order to also be able to compare orphan and missing draw calls, we make use of the pixels in the final image. Orphan draw calls can be a sign of artifacts in the final image and missing draw calls lead to areas of this image not being rendered. To make sure of the health of the frame, we compare the final image from the local & cloud instances directly. Algorithm 1 shows the steps of our procedure to find different sets of draw calls, where $k_L \cap k_R$ is the set of matched draw calls, $k_L - k_R$ is the set of missing draw calls & $k_L - k_R$ is the set of orphan draw calls. Some draw calls are used for calculations, and their output has no direct impact on the output images pixels, but they can still be the root of some graphical bugs. The above procedure can also be used to compare such draw calls as their output is used by other draw calls which do change pixels.

**Algorithm 1** Draw Calls Comparison Algorithm

**Input**: Set of: reference draw calls $D_L$, secondary draw calls $D_R$ and the faulty portion of the frame $Sq_{Error}$

**Output**: Possible candidates for sets of Matched, Missed and Orphan draw calls.

Let $F_L = \emptyset$ and $F_R = \emptyset$
**for** $E \in \{L, R\}$ **do**
  **for** $i = 0$ **to** $|D_E|$ **do**
    Let $Sq_i$ be bounding box for $D_E[i]$
    **if** $Sq_i \cap Sq_{Error} \neq \emptyset$ **then**
      Add $Sq_i$ to $F_E$
    **end if**
  **end for**
**end for**
Let $T_L$ and $T_R$ be empty hash tables
**for** $E \in \{L, R\}$ **do**
  **for** $i = 0$ **to** $|F_E|$ **do**
    $Key = VertexCount(F_E[i])$
    Insert $F_E[i]$ into $T_E[Key]$
  **end for**
**end for**
Let $\kappa_L$ be all keys in $T_L$ and $\kappa_R$ all keys in $T_R$
Return $\kappa_L$, $\kappa_R$, $\kappa_L - \kappa_R$, $\kappa_R - \kappa_L$ and $\kappa_L \cap \kappa_R$

# 4. Evaluation & Results

Various experiments were performed to evaluate the accuracy of our method in identifying graphical problems. To simulate the cloud computing environment, we used KVM & QEMU with VGA passthrough. We created six virtual machines, each one having a dedicated graphics card. Each VM has 4 CPU cores and 6GB of RAM. The exact specification of these VMs can be seen in Table 1.

Table 1: Specs of VMs used for Simulation

| Component | Details |
|---|---|
| **CPU** | Dual Xeon E5-2620v3 |
| **VM1 VGA** | NVIDIA GTX 1080 |
| **VM2 VGA** | NVIDIA GTX 780 Ti |
| **VM3 VGA** | NVIDIA GT 740 |
| **VM4 VGA** | AMD Radeon RX 480 |
| **VM5 VGA** | AMD Radeon R9 270 |
| **VM6 VGA** | AMD Radeon HD6450 |
| **RAM** | 6x 8GB DDR4 ECC 2400MHz |
| **NVMe Storage** | Plextor 256GB PCIe NVMe |
| **SSD Storage** | SAMSUNG 850 EVO 1TB |
| **Hypervisor** | QEMU + KVM |
| **Guest OS** | Windows 10.0.15063 |

We use VM1 (housing an NVIDIA GTX 1080) for deep learning uses, utilizing the TensorFlow open-source library. The other five VMs are used to execute the video game instances.

For our evaluation to be meaningful, we needed to create examples of graphical bugs in the executions deterministically. As a collection of data relating to such bugs with the level of detail we required does not exist, we artificially created a collection of such bugs. This was done on video games based on the Unity game engine. We manipulated the Vertex Buffer, Textures & Materials to create more than 1000 graphical problems in different rendered objects. An example of such graphical problems can be seen in Figure 5.

As all the VMs are run on a single machine, the communication delay between the game instances & the corruption detection platform is nearly zero. Therefore, we set the $\delta_f$ parameter to 5ms and the $\delta_w$ parameter to 10 frames. Our observations show that increasing $\delta_f$ up to 30ms has a negligible impact on accuracy. This is due to the dataset used and also the fact that most issues last at least 1 second. For similar reasons, increasing the value of parameter $\delta_w$ to more than 30 frames does not considerably change accuracy.

To evaluate the first DenseNet network, we look at its ability to detect corrupted frames. After the extraction of video game frames from online platforms using the method described above, we classified them using the DenseNet network. 10% of the input data were used as the validation set and 10% held out for the Test set, with the rest used for the purpose of training. The DenseNet network's accuracy during the training period can be seen in Figure 6.

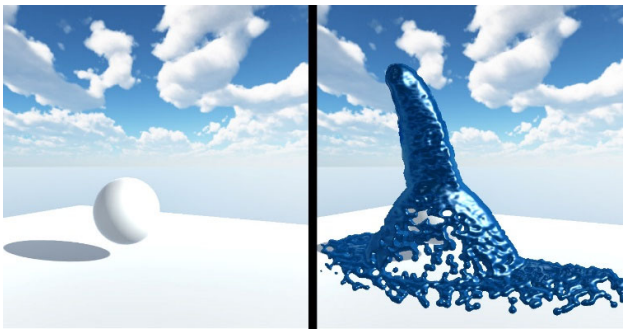

Figure 5: Inability of the DenseNet network in detecting graphical problems - Right: Correct rendering of a fluid - **Left**: The fluid has not been rendered at all [32].

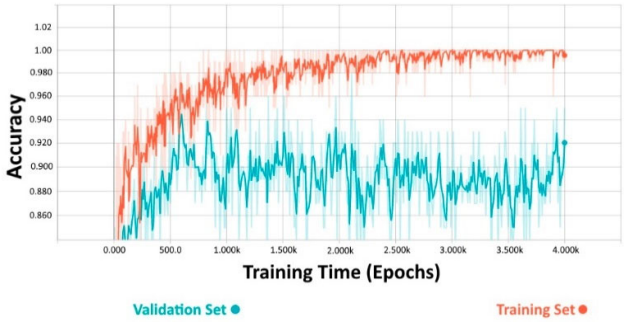


Figure 6: Training & validation accuracy of detecting graphical bugs using the proposed method.

As can be seen, the accuracy of detecting corrupted frames in our proposed method can reach up to 94% in the Validation set. The final accuracy of the network for the Test set is around 90%. These numbers show us that the DenseNet network works exceptionally well in most situations. Also, we can see that the gap between the accuracy of the validation set and the training set does not change in later epochs. This shows us that we are in an overfitting regime, and therefore more training will not increase the validation accuracy. This can be remediated by increasing the amount of training data.

In some scenarios, the first DenseNet network used in our method might not be able to detect a graphical glitch. Figure 5 shows such a situation, in which the absence of a graphical element (here a simulated fluid) cannot be detected. This represents the hardest class of graphical issues which even a human might not be able to identify unless they have specific knowledge on the scene being rendered. As another example, in Figure 7, an image of a special effects scene is shown. This scene might visually look like a glitch, but it is in fact intended to look distorted. Such images might be falsely classified as a corrupted frame by the first DenseNet network. Here the second DenseNet network comes to our aid. This network is trained on the input relating to the difference between the pixels of a frame from a faulty execution and a frame from the healthy reference execution. This allows it to estimate the boundary between corrupted frames and healthy ones. As our generated corrupted frame data is very small compared to the set of all possible graphical issues, in our observations, the CNN can reach an accuracy of 90% in this second part.

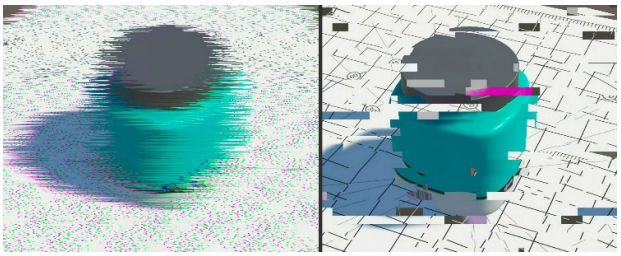

Figure 7: Example of incorrect classification of a frame related to a special effects scene[33].

## 5. Conclusion and Future Works

In this paper, we presented a novel video game testing method for graphical bugs that cause various visual artifacts in a video game's rendered frames. Our results show that the use of deep learning methods can be very effective in detecting graphical bugs. Here, after corrupted frames are identified, by processing the draw calls responsible for such frames and comparison to the draw calls of related healthy frames we can identify the root cause of the problem at the draw call level. As we described in previous sections, the total accuracy of our proposed method is 81%, with the first DenseNet network's accuracy near 90% and the second network's accuracy at around 90%. Further improvements in accuracy can be obtained by using a more extensive data set.

One future direction this work can take is to create the input to the game executions automatically. At the moment this input is given by a human game tester. Reinforced Learning methods can be useful in this regard by training a collection of agents that can play the role of the tester and create input commands for the game. Another promising direction is to extend our method for specific classes of video games. For example, a large number of video games today are made for various mobile platforms. Here cloud services that are tailor-made for mobile app testing, such as AWS device farm [34], can be used. This can pose its own challenges, such as the difference in the resolution of game instances run on various mobile devices.

## References

[1] Sales of batman: Arkham knight's pc version suspended on steam (update), http://www.polygon.com/2015/6/24/8842447/, [accessed 20-December-2017] (2015).

[2] Y. J. Choi, Providing novel and useful data for game development using usability expert evaluation and testing, in: Computer Graphics, Imaging and Visualization, 2009. CGIV'09. Sixth International Conference on, IEEE, 2009, pp. 129–132.

[3] P. Moreno-Ger, J. Torrente, Y. G. Hsieh, W. T. Lester, Usability testing for serious games: Making informed design decisions with user data, Advances in Human-Computer Interaction 2012 (2012) 4.

[4] N. M. Diah, M. Ismail, S. Ahmad, M. K. M. Dahari, Usability testing for educational computer game using observation method, in: Information Retrieval & Knowledge Management,(CAMP), 2010 International Conference on, IEEE, 2010, pp. 157–161.

[5] C. Buhl, F. Gareeboo, Automated testing: a key factor for success in video game development. case study and lessons learned, in: Proceedings of Pacific NW Software Quality Conferences, 2012, pp. 1–15.

[6] K. Peterson, S. Behunin, F. Graham, Automated testing on multiple video game platforms, uS Patent App. 13/020,959 (Feb. 4 2011).

[7] C. Schaefer, H. Do, B. M. Slator, Crushinator: A framework towards game-independent testing, in: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, IEEE, 2013, pp. 726–729.

[8] A. M. Smith, M. J. Nelson, M. Mateas, Computational support for play testing game sketches., in: Artificial Intelligence for Interactive Digital Entertainment (AIIDE), 2009 the Fifth International Conference on, AAAI, 2009, pp. 167–172.

[9] I. Zarembo, Analysis of Artificial Intelligence Applications for Automated Testing of Video Games, in: Environment Technologies Resources, 2019 Proceedings of the International Scientific and Practical Conference, Vol. 2, pp. 170-174.

[10] A. Nantes, R. Brown, F. Maire, A framework for the semi-automatic testing of video games., in: Artificial Intelligence for Interactive Digital Entertainment (AIIDE), 2008 The Fourth International Conference on, 2008.

[11] B. Chan, J. Denzinger, D. Gates, K. Loose, J. Buchanan, Evolutionary behavior testing of commercial computer games, in: Evolutionary Computation, 2004. CEC2004. Congress on, Vol. 1, IEEE, 2004, pp. 125–132.

[12] Y,Zheng, X. Xie, T. Su, L. Ma, J Hao, Z. Meng & C. Fan, Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. 2019, in: 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE , pp. 772-784.

[13] J. Bergdahl, C. Gordillo, K. Tollmar, & L. Gisslén, Augmenting automated game testing with deep reinforcement learning, 2020, in: 2020 IEEE Conference on Games (CoG) pp. 600-603.

[14] C. Ling, K. Tollmar & L. Gisslén, Using Deep Convolutional Neural Networks to Detect Rendered Glitches in Video Games, 2020, in: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, Vol. 16, No. 1, pp. 66-73.

[15] A. Watson, Deep Learning Techniques for Super-Resolution, in Video Games, 2020, arXiv preprint arXiv:2012.09810.

[16] C.-S. Cho, K.-M. Sohn, C.-J. Park, J.-H. Kang, Online game testing using scenario-based control of massive virtual users, in: Advanced Communication Technology (ICACT), 2010 The 12th International Conference on, Vol. 2, IEEE, 2010, pp. 1676–1680.

[17] Y. Choi, H. Kim, C. Park, S. Jin, A case study: Online game testing using massive virtual player, in: Control and Automation, and Energy System Engineering, Springer, 2011, pp. 296–301.

[18] S. Iftikhar, M. Z. Iqbal, M. U. Khan, W. Mahmood, An automated model based testing approach for platform games, in: Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on, IEEE, 2015, pp. 426–435.

[19] M. Ostrowski, S. Aroudj, Automated regression testing within video game development, GSTF Journal on Computing (JoC) 3 (2) (2013) 60.

[20] S. Varvaressos, K. Lavoie, A. B. Massé, S. Gaboury, S. Hallé, Automated bug finding in video games: A case study for runtime monitoring, in: Software Testing, Verification and Validation, 2014 IEEE Seventh International Conference on, IEEE, 2014, pp. 143–152.
[21] E. Cuervo, A. Wolman, L. P. Cox, K. Lebeck, A. Razeen, S. Saroiu, M. Musuvathi, Kahawai: High-quality mobile gaming using gpu offload, in: Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, ACM, 2015, pp. 121–135.
[22] Unity - game engine, https://unity3d.com/, [accessed 20-December-2016] (2016).
[23] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli, Image quality assessment: from error visibility to structural similarity, IEEE transactions on image processing 13 (4) (2004) 600–612.
[24] R. Mantiuk, K. J. Kim, A. G. Rempel, W. Heidrich, Hdr-vdp-2: a calibrated visual metric for visibility and quality predictions in all luminance conditions, in: ACM Transactions on graphics (TOG), Vol. 30.4, ACM, 2011, p. 40.
[25] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, O. Wang, The unreasonable effectiveness of deep features as a perceptual metric, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 586–595.
[26] G. Huang, Z. Liu, L. Van Der Maaten, K. Q. Weinberger, Densely connected convolutional networks, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2017, pp. 4700–4708.
[27] Unreal engine technology, https://www.unrealengine.com/, [accessed 20-December-2016] (2016).
[28] Renderdoc, a stand-alone graphics debugging tool., https://github.com/ baldurk/renderdoc/, [accessed 20-December-2017] (2016).
[29] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.
[30] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: A largescale hierarchical image database, in: 2009 IEEE conference on computer vision and pattern recognition, Ieee, 2009, pp. 248–255.
[31] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, M. Zaharia, Noscope: optimizing neural network queries over video at scale, Proceedings of the VLDB Endowment 10 (11) (2017) 1586–1597.
[32] uflex - asset store, goo.gl/gpaUJQ, [Online; accessed 18-December-2017] (2017). URL goo.gl/gpaUJQ
[33] Kinoglitch: Video glitch effects for unity, https://github.com/keijiro/KinoGlitch, [accessed 18-December-2017] (2017).
[34] Mobile app testing on devices - aws device farm, https://aws.amazon.com/device-farm/, [accessed 26-December-2017] (2017).

**Mohammad Reza Taesiri** received his BSc degree in Mathematics from Amirkabir University in 2015, and his MSc degree in Software Engineering from Sharif University in 2017. His research interests include video games, explainable, and interpretable machine learning.

**Email:** mtaesiri@gmail.com

**Moslem Habibi** received his BSc, MSc and PhD degrees in Computer Engineering from Sharif University of Technology. He is currently an assistant professor at Sharif University's Industrial Engineering department where his research centers on Digital Transformation, Organizational Agility and Application Lifecycle Management.

**Email:** mhabibi@sharif.edu

**MohammadAmin Fazli** received his BSc in hardware engineering and MSc and PhD in software engineering from Sharif University of Technology, in 2009, 2011 and 2015 respectively. He is currently a Lecturer at Sharif University of Technology and R&D Supervisor at Sharif's Intelligent Information Center resided in this university. His research interests include Game Theory, Combinatorial Optimization, Computational Business and Economics, Graphs and Combinatorics, Complex networks and Dynamical Systems.

**Email:** fazli@sharif.edu

**Paper Handling Data:**

Submitted: 12.27.2020
Received in revised form: 05.12.2021
Accepted: 05.20.2021
Corresponding author: Dr. Moslem Habibi
Industrial Engineering Department, Sharif University of Technology, Tehran, Iran