İki Boyutlu Video Oyunlarında Sinir Stili Aktarımı Kullanarak Otomatik Oyun Mekaniği ve Estetiği Üretimi

Araştırma Makalesi/Research Article

Deniz ŞEN, Hasan Tahsin KÜÇÜKKAYKI, Elif SÜRER

Çokluortam Bilişimi, Modelleme ve Simülasyon Enstitü Anabilim Dalı, Enformatik Enstitüsü, Orta Doğu Teknik Üniversitesi, Ankara, Türkiye
deniz.sen_01@metu.edu.tr, tahsin.kucukkayki@metu.edu.tr, elifs@metu.edu.tr
(Geliş/Received:25.07.2020; Kabul/Accepted:02.07.2021)
DOI: 10.17671/gazibtd.706884

Özet— Video oyunu araştırması, karmaşık yöntemlerin ve algoritmaların geliştirildiği, sürekli değişmekte olan, dinamik bir alandır. Prosedürel içerik üretimi, kullanıcı tarafından oluşturulan parçaları video oyunu içeriğini otomatikleştirmek ve geliştirmek için algoritmalarla birleştirmeyi amaçlamakta ve bu yöntemlerin temelini oluşturmaktadır. Bununla birlikte, sonuçlar oyun mekaniğine ve oyunun oynanış biçimine değil, çoğunlukla oyun estetiğine yansımaktadır. Bu çalışmada, "tuval olarak oyun sahnesi" konsepti ile kullanıma hazır çarpıştırıcılar ve oyun estetiğini geliştiren, sanatsal açıdan farklı stiller kullanarak iki boyutlu oyun seviyesindeki bir görüntüyü basit bir prototip oyun geliştirme ortamına dönüştürebilen yöntem ve süreç sunulmaktadır. Bu amaçla, giriş oyun seviyesi görüntüsünün kenar ve renk bazlı özellikleri Canny kenar belirleme, basit doğrusal yinelemeli kümeleme ve Felzenszwalb segmentasyonu kullanılarak çıkarılmaktadır. Daha sonra, Unity oyun motoru, mekansal kontrol ile oyun seviyesinin stilinin aktarıldığı kenar ve renk özelliklerine göre çarpıştırıcılar oluşturmak için kullanılmaktadır. Farklı sinir stil transfer algoritmalarının sonuçları, Super Mario, Lode Runner ve Kid Icarus gibi oyunlar üzerinde karşılaştırılmakta ve tartışılmaktadır. Sonuçlar, bu çalışmanın oyun mekaniği ve oyun estetiğine odaklanarak iki boyutlu video oyunu geliştirmeyi kolaylaştırma potansiyeline sahip bir araç olduğunu göstermektedir.

Anahtar Kelimeler— sinir stili aktarımı, görüntü işleme, oyun mekaniği, video oyunu.

Automated Game Mechanics and Aesthetics Generation Using Neural Style Transfer in 2D Video Games

Abstract— Video game research is an ever-changing and dynamic area where sophisticated methods and algorithms are being developed. Procedural content generation (PCG), which aims to merge user-generated assets with algorithms to automate and improve video game content, has been the core of this sophistication. However, the outcomes are primarily reflected in game aesthetics, not in the game mechanics and gameplay. In this study, we introduce the "game scene as a canvas" concept where simple prototype game development pipelines, that can convert a 2D game-level image into a game development environment with ready-to-use colliders and artistically different styles that enhance the game aesthetics, are introduced. To do so, edge-based and color-based features of the input game level image are extracted using the Canny edge detector, Simple Linear Iterative Clustering, and Felzenszwalb segmentation. The Unity game engine is then used to generate colliders based on the provided edge and color features where the game level is style transferred with spatial control. Results of different neural style transfer algorithms are presented on benchmark games such as Super Mario and Kid Icarus. Results show that this study can become a promising tool to simplify 2D video game development, focusing on game mechanics and aesthetics.

Keywords— neural style transfer, image processing, game mechanics, video games.

1. INTRODUCTION

Recent developments in video game research are providing new experiences and challenges to game players. With the advancements of visualization, optimization, and automation algorithms, and with the help of the increase in computational power, the functionalities and the aesthetics of video games are getting more sophisticated and more

advanced [1-2]. The main objective of the games is to enable their players to enter the "flow" state where the skills of the player and the challenges provided by the game mostly match, causing a state where the player loses track of time [3]. To do so, the games should change, adapt, and create diverse, challenging, and entertaining experiences. Playing the same games over and over could become a repetitive experience in time if the game does not provide new levels, challenges, appearances, or user-centric adaptation.

Procedural content generation (PCG) is the systematic automation of producing content merging user-generated sprites, audio, and visuals using algorithmic approaches in order to create enhanced, diverse and automatic content in a fast and transitive way [4]. Neural style transfer is one of the PCG approaches that create diverse artistic styles [5-11] and get remarkable results, mainly in the image processing domain. Neural style transfer has become a highly researched topic in recent years due to deep learning practices [5,11]. So far, the neural style transfer experiments have been done to enhance the images' aesthetic values, and there have been other approaches rather than only deep learning algorithms.

Earlier works on non-parametric texture synthesis, such as [9], tried to address filling and scaling the images with different texture properties. The same approach was also used in [10], in which two images were merged to obtain a new styled image by overlapping and combining related and small patches of the images. Despite these attempts, deep learning has become the state-of-the-art approach in image generation, as most of the time, the deep models can learn the linear and non-linear relationships between their inputs and outputs automatically, which mainly eliminates the image pre-processing step. For instance, in [5], a convolutional neural network (CNN) was trained to extract content and style representations out of corresponding images, and these assets were used to create a new image, and the outcomes were successful. However, using CNNs may cause unnecessary and unrealistic distortions for accurate photo style transfer. One solution to this problem is to associate semantic labeling of the input and style images with maximizing the subregion mapping, as proposed in [6]. In [7], a different approach in which the foreground segmentation was combined with the neural style transfer was used. The main idea was to style only the user-specified object, which was done by first styling the whole image and then separately segmenting out the object to overlap the segmented and styled image further.

Although deep neural network approaches seemed to perform relatively well, they are mostly trained with artistic style images, which may cause them to overfit to these kinds of styling operations inherently. When it comes to using the neural style transfer approach in video games, these models may struggle to perform as intended. Thus, a more generic neural style transfer model can be more useful for the video game domain. The model proposed in [8] can style any arbitrary content image with any style image by using an auto-encoder network architecture, followed by correlating the content and style images to produce a new image. This model also has the ability to style hand-crafted parts of the image individually with different style images.

While PCG and neural style transfer can be used to create diverse content on image processing, audio signal processing, and text generation domains, generating game content by using machine learning models [12], which is called Procedural Content Generation via Machine Learning (PCGML), addresses mainly the video game research and introduces new types of experiences to gamers. PCGML uses the content within the game to change the effects of them, such as levels, characters, and maps, to name a few. PCGML [12] introduces groundbreaking developments such as generating game artifacts, auto-completion of missing game content, repair unplayable areas, recognize, analyze, and evaluate the game content. Snodgrass and Santiago Ontanon [13] used Markov Chains to map the game levels between different games while extracting a mapping on game tiles. They evaluated their outcomes with Super Mario Bros., Kid Kool, and Kid Icarus games. Guzdial and Riedl [14] applied the probabilistic models learned from the video gameplay and merged those models to create new game levels. They evaluated the outcome of their model with human participants. Gow and Corneli [15] also blended two games using Conceptual Blending on Video Game Description Language (VGDL) [16].

Although these attempts have begun to be used in video game research, to the best of the authors' knowledge, neural style transfer has not been merged and adapted with game mechanics transfer yet. Summerville et al. [12] summarized this phenomenon as: *"These approaches transfer and blend level styles, but do not attempt to address the game mechanics explicitly; both approaches ensure playable levels, but do not attempt transfer or blending between different mechanics."* Thus, transferring the style of well-known background images of the game levels and adding extra functionalities to the game objects could enhance the complexity and content of a game.

In this study, we present forward and backward game development pipelines using neural style transfer and game mechanics transfer and showcase the outcomes of our proposed method on five benchmark video games. The following sections describe the steps of our methodology in detail, followed by the visual and quantitative results of our game mechanics and game aesthetics transfer.

2. MATERIALS AND METHODS

Our game mechanics and game aesthetics transfer architecture can be divided into three main sections: feature mask generation, neural style transfer, and game mechanics generation. This study applies these steps to two different game development pipelines —forward and backward pipelines, as can be seen in Figure 1 and Figure 2— and provides a thorough analysis of five benchmark video games in terms of visual and quantitative outcomes.

2.1. Neural Style Transfer Pipelines

2.1.1 Forward Pipelines

In the forward pipeline, the game mechanics generation is done using the properties of the original level image. This generation includes three stages that are executed in the following order: mask generation, neural style transfer with region control, and game mechanics generation. The mask generation step is a series of operations that produces a binary bitmap —i.e., a mask— and the procedures are explained in Section 2.2. The style transfer uses this mask, the original level image, and two style images to produce a styled background image for the newly created game. The game mechanics generation part uses several features of the Unity game engine, which can produce colliders using the binary mask generated earlier and the styled background image. The details of this procedure are explained in the game mechanics generation section. The pipeline terminates by producing a new Unity project/game that contains a 2D game environment with new colliders and a styled background layer.

2.1.2 Backward Pipelines

The backward pipeline is different from the forward pipeline such that it first styles the input game level image, followed by the mask generation step. Thus, in this pipeline, the neural style transfer is not region controlled, so at the instance of styling the background, there is not any mask to indicate the regions to be styled using a different style image. The mask generation step uses the same algorithms that are used in the forward pipeline, but the input is styled as the background image. Furthermore, the mask and the styled image are put in the same operations in the Unity game engine and are used to create a new game environment with new colliders.

2.2. Mask Generation

2.2.1 Edge Based Mask Generation

The mask generation [17] operation consists of using different image processing techniques, which extract a binary mask from an input game level image and process its edge features. The process starts with converting the input RGB image into grayscale, which is in a single channel form. Then, the image goes through a morphological dilation operation with a disk kernel. This operation highlights the boundaries of the color changes, and it helps the edge detection step. In the absence of this operation, it becomes difficult to detect the edges of the image.

The Canny edge detection algorithm is a commonly used edge detection algorithm, and the mask generation pipeline also uses this algorithm to find the edges of the features [18]. Since the dilation operation creates sharp changes of grayscale colors, it equivalently adds sharper changes across the image, and Canny edge detection searches for higher changes between the pixel values. This algorithm creates a binary image with the detected edges indicated by the value one. This binary image is then used for contour detection to find the groups of 1-valued pixels. The contour detection algorithm used is described in [18], and it also finds the bundles that can be completed even though a bundle does not exactly describe a shape. To complete these contours, each line of these contours is thickened to form near-perfect shapes. Then, block-based connected component labeling with 8-neighbors is applied. The contour size is calculated based on the average number of pixels of a contour. The contours that have a smaller number of pixels than the average are eliminated. The output of this step is the generated mask, which will be used in the following steps of the pipeline.

2.2.2 Color-Based Mask Generation

The pipelines also generate mask images out of the color images based on the floating-point RGB values. Colors provide defined spatial properties of the image, which makes it a reliable index to determine the locations to pick the color values. Superpixels are groups of pixels that can be considered as one large chunk of space. This operation can be classified as a clustering operation that can be used in unsupervised segmentation practices. The two effective and easy-to-use segmentation algorithms —Simple Linear Iterative Clustering (SLIC) and Felzenszwalb— are used to detect the groups of pixels that have the same characteristics.

2.3. Segmentation

2.3.1 SLIC Segmentation

In this study, in order to segment the parts of the color-based pipelines, the SLIC algorithm [19,20] was chosen. SLIC uses the principle of the k-means clustering algorithm, which groups the data according to their similarity in content and their positions on the respective space. This algorithm requires a parameter to be chosen, mostly heuristically, that indicates the number of clusters to be formed on the image. However, the output may not always give the exact number of the parameterized cluster number in practice. As mentioned earlier, these clusters are mostly called superpixels as the ensemble of the groups of pixels creates a meaningful image.

In this study, a threshold is applied to the output of SLIC first. However, assigning a metric as a threshold and choosing a respective threshold value are rather difficult tasks as the numerical data of the contents of the superpixels are unreliable and unpredictable. On the other hand, in this study, after several iterations on the segmented images using SLIC, the regions with a higher number of color value changes are seen to contain important spatial features in the 2D video games. A metric that can be used to detect the amount of change inside the

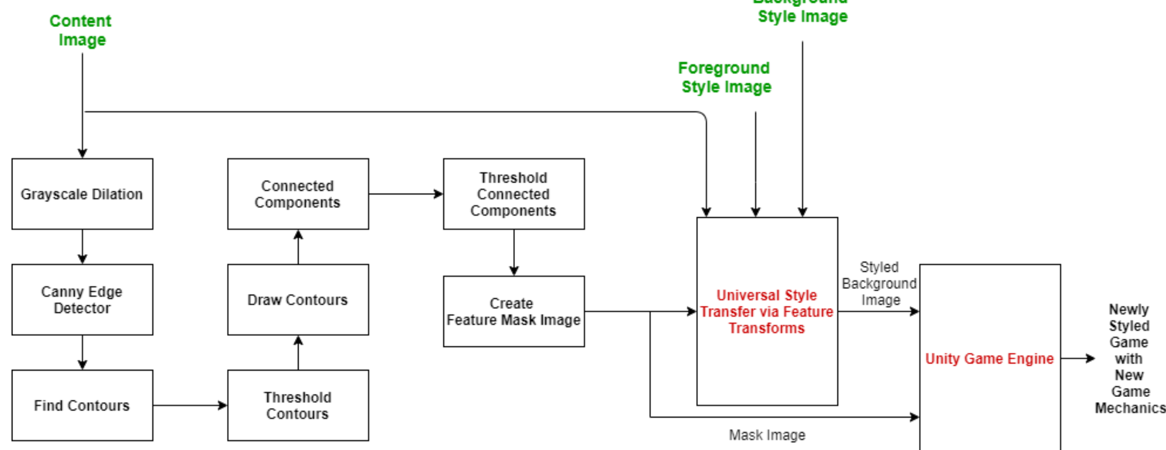

Figure 1. Forward edge pipeline

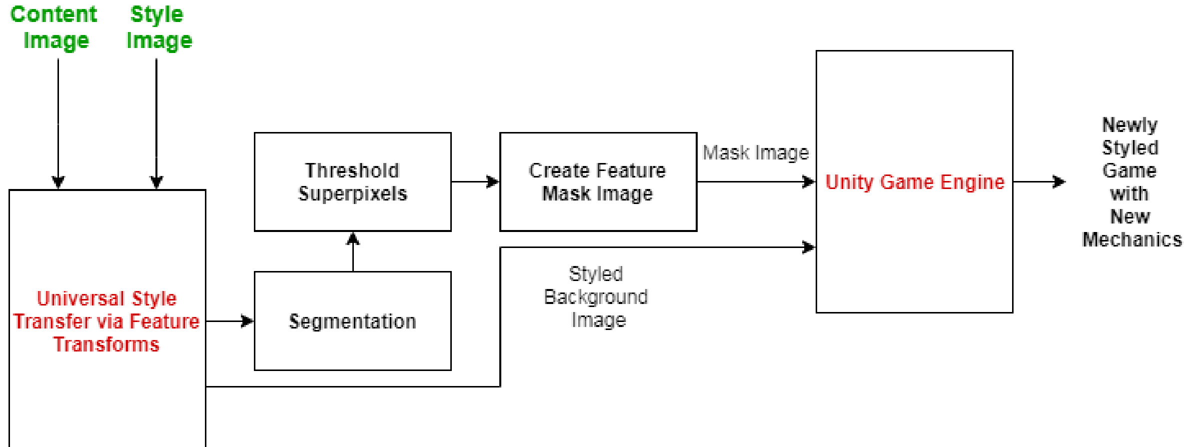

Figure 2. Backward color pipeline

clusters is the variance of the RGB values of the member pixels. Therefore, the separated variances of the three-color channels were averaged and saved.

Then, a threshold is applied to the mean variances of each superpixel. Then, the mean-variance values are then sorted, followed by a threshold operation. This process is parameterized with the elimination rate, indicating the index inside the unique values of the mean variances. The value corresponding to this index determines the superpixel that will be labeled as one, which indicates the presence of a collider on the pixels of this superpixel. This selection is made by comparing the mean-variance value of the superpixel —if the value is smaller than the threshold value, the pixels are labeled as zero and otherwise labeled as one. The binary mask is then generated based on the labels of the pixels.

2.3.2 Felzenszwalb Segmentation

The second segmentation algorithm used is Felzenszwalb [21], which is a graph-based approach that uses the color properties of the image. The procedure of generating a mask out of the segmentation of the input image is precisely the same as the SLIC segmentation; however, several parameters such as the minimum number of pixels that a segment can possess or the size of the Gaussian kernel are different. The rest of the pipeline is the same as the previous segmentation procedure.

2.4. Neural Style Transfer

The neural style transfer step is either done before or after the mask generation step. In such cases, the usage of the neural style transfer model changes drastically. The neural style transfer in this architecture is done via the implementation of [8], which is an auto-encoder that can transfer the style properties of one image to another to finally obtain a visually pleasant image (Figure 3, Table 1). The idea of this study is thoroughly based on the transfer of an aesthetically pleasing style transfer into the 2D video game domain. Therefore, in this case, the input content image, which is the commonly used term for the image to be styled, is a 2D game level image. Besides, the game objects can have distinctive appearances from which the player can understand that object's functionality. The base style transfer implementation also can have spatial control over the stylization process; in other words, the model can style indicated locations of the content image with a new style image, different from the original one. The spatial

control is done via the indication of the locations to be styled differently with a bitmap with the same size as the content image that has value one on the pixels that are preferred to be styled differently.

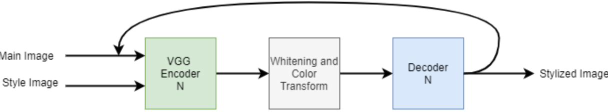

Figure 3. At each iteration, a different (less deep) pair of encoder and decoder is used and the procedure is done for five iterations (Adapted from Li et al. [8])

Table 1. The training procedure's hyperparameters and the total reconstruction loss function and common qualities between different layers of encoder and decoder networks

Parameter	Value
Optimizer	Adam
Learning rate	1e-4
Learning rate decay	5e-5
Beta1	0.9
Beta2	0.999
Maximum iteration	16000
Convolution kernel size	3x3
Convolution kernel number	Either 64, 128, 256 or 512, depending on the encoding and decoding layer
Padding	Reflection, 1 pixel in each side
Pooling in encoders	Maximum
Pooling window in encoders	2x2
Pooling stride in encoders	2x2
Activation function	ReLU
Upsampling in decoders	Nearest with a scaling factor of 2
Pixel reconstruction loss weight	1
Feature loss weight	1

As mentioned earlier, the styling step can be done on different timestamps, depending on the main pipeline. In the forward pipeline, the mask generation is done based on the content image before the style transfer step. Note that the output of the step before the neural style transfer step is a bitmap, which contains the information of the desired spatial control behavior. Neural style transfer with spatial control requires four inputs; the content image, the main style image, the mask style image, and the mask —which is a bitmap image. The resultant image contains different kinds of styles depending on the input mask. However, in the backward pipeline, the mask generation is done after the neural style transfer. In this pipeline, the neural style transfer works without spatial control as the mask is generated based on the styled image itself. Ultimately, the goal is to compare the results of the output of the pipelines when the mask is generated based on the original content image or the styled content image.

The images in Figure 4 (stone texture[1] and Marsden Hartley's Abstraction[2]) were used as the style images since they have the aesthetic features and the addition of a third dimension in the transfer.

Figure 4. From left to right: (a) Mask style image, and (b) main style image

2.5. Collider Generation

The last part of either of the pipelines is the collider generation, where the mask generated from the prior steps is put under several simple steps that result in obtaining a game sprite having colliders in appropriate locations so that the game objects become interactable. For this step, the study counts on several features that the Unity game engine [22] provides. The Unity game engine has the feature to generate several triangular colliders out of binary masks and requires a certain amount of tolerance [23] in terms of the difference between the collider places indicated in the mask and the generated colliders. For this experiment, the tolerance was kept as low as possible to obtain a collider map that matches the styled background components.

At this point, even though the colliders are ready to be played on, the game requires a background whose visual features overlap with the generated collider map [24]. It is a rather simple step as the Unity game engine only requires the styled background image to be imported and used as a sprite and to be located behind the colliders. However, it is relatively difficult to place the sprites on the specific positions of the game space. A simple solution to this problem is to put both the collider map and the background style image on the origin of the game scene, but both the background image and the collider map must match exactly in terms of the pixel placements; otherwise, the game might become unplayable. For instance, without any adjustment, the player may be placed on a certain location that does not have any visual features, such as a box that can be stepped on; however, because of the lack of scaling on the collider map, there can be a collider that should not be present. For this issue, the metric of the scaling is taken as the pixel per unit parameter that is defined while the background image

[1] Internet: Texture, Stone Walls - Image Source: https://www.sketchuptextureclub.com/textures/architecture/stones-walls/claddings-stone/exterior/wall-cladding-stone-texture-seamless-19009, 24.07.2020.

[2] Internet: Marsden Hartley's Abstraction - Image Source: https://commons.wikimedia.org/wiki/File:Marsden_Hartley_-_Abstraction_-_Google_Art_Project.jpg, 24.07.2020.

and the bitmap are imported into the Unity game project. When the heights and widths of both assets are inherently equal to each other, and if both assets are imported with one game unit that corresponds to the same number of pixels per unit parameter, their overlap will ultimately result in complete overlaps; thus, the desired outcome will be achieved.

3. RESULTS AND DISCUSSION

In this section, visual and quantitative outputs from the previous steps will be given and interpreted individually. The benchmark games Super Mario Bros, Super Mario Kart, Rainbow Islands, Lode Runner, and Kid Icarus [25] were used to demonstrate the results. The games do not contain any 3D assets and complex visual features. In the examples, an abstract tone has been intentionally given to the well-known benchmark games to highlight the neural style transfer's potential in enhancing the game aesthetics. A more realistic output could have been possible by changing the hyperparameters and the number of iterations.

3.1. Edge Pipeline Results

In theory, edge-based features were expected to provide usable results. The human eye can, in fact, differentiate the game objects and possible colliders only by looking at the image constituted by only the edges. Figure 5c is an example of such an image; the human eye can imagine the possible appearance and shape of the colliders only by looking at that field.

The steps of the forward edge pipeline produce the fields that are shown in Figure 5. The dilation has some remarkable effects on the performance of the Canny edge detector, as the output of this step is able to give reliable clues about the way the colliders will be put in the later steps. The noise elimination process, which is the connected component thresholding step, did, in fact, eliminate the small chunks of the objects, which can create a smoother and better gaming experience on the map. In the last step, the Unity game engine generated a sprite that contains the colliders that the image processing steps have created. The collider generation is done via approximating the polygons that can be put on the binary image, and for this particular pipeline, the polygons have meaningful scales and positions. As mentioned earlier, the tolerance of not covering the features indicated in the mask was set to be low; therefore, the approximations tend to become more complex yet more accurate. On the other side, the backward edge pipeline, whose outputs are given in Figure 6, seems to have more noisy outputs in the intermediate steps. In the case of Figure 6b, and possibly in any case of styling with a different style image pair, the level image gains some 3D effects, such as subtle color gradient changes and alpha effects. The later steps then tend to struggle remarkably in detecting the image features. The colliders have also become more complex as the features tend not to have trivial shapes.

3.2. Color Pipeline Results

The overall performance of the color pipeline varies due to the pipelines preferred. In the forward pipeline with the SLIC algorithm having 1000 segments as in Figure 7, the separations seem to be well distributed. The segments are meaningful in terms of their contents and their neighborhood. When it comes to the elimination of the redundant segments, which mostly correspond to the background of the level image, the variance-based thresholding results in eliminating several unimportant parts that do not signify meaningful and potential game mechanics. Most of the segments have rectangular areas that decrease the complexity and make the game sprites not overlap completely with the game mechanics. On the other hand, the fact that the segments have low shape complexities creates an easier environment for the Unity game engine to fit the polygons. The polygons on the output are very close to the input mask. However, in some places, the collider generator connects and covers multiple segments to create a single polygon, which produces inaccurate outcomes. The backward pipeline using SLIC, which can be seen in Figure 8, has some major inaccuracies.

The style added before the segmentation process adds some complexity, better yet, a slight dimensionality, which affects the segmentation process. Besides, the variance-based thresholding seems to fail as the gradient on the image added by the styling step makes the variances distribute too evenly throughout the segments. After this point, the collider generation is seen to work poorly because it is too difficult to fit the polygons inside the selected regions.

In Figure 9, Felzenszwalb segmentation seems to separate the segments better than the SLIC algorithm since not every segment requires to have similar color properties. For instance, the sky in the game level is taken as a single segment, which simplifies the thresholding process. However, it is seen that some of the large objects are dropped out during the thresholding process, such as the green pipes. The collider generation step is also more effective in terms of the precision of the edges of the colliders. The backward pipeline in Figure 10 performs in a different way —the segments are not as well-defined as the forward pipeline. However, looking at the overall mask generated from the variance-based thresholding, the result is well-organized and well put together, apart from some cases where the clouds are too close to the platforms. The output of the collider generation shows that the estimated polygons seem to fit well to create an overall playable map (Figure 10 and Figure 11). Figure 12, Figure 13, and Figure 14 demonstrate the end-to-end pipeline outcomes of the games Rainbow Islands, Lode Runner, and Kid Icarus, respectively.

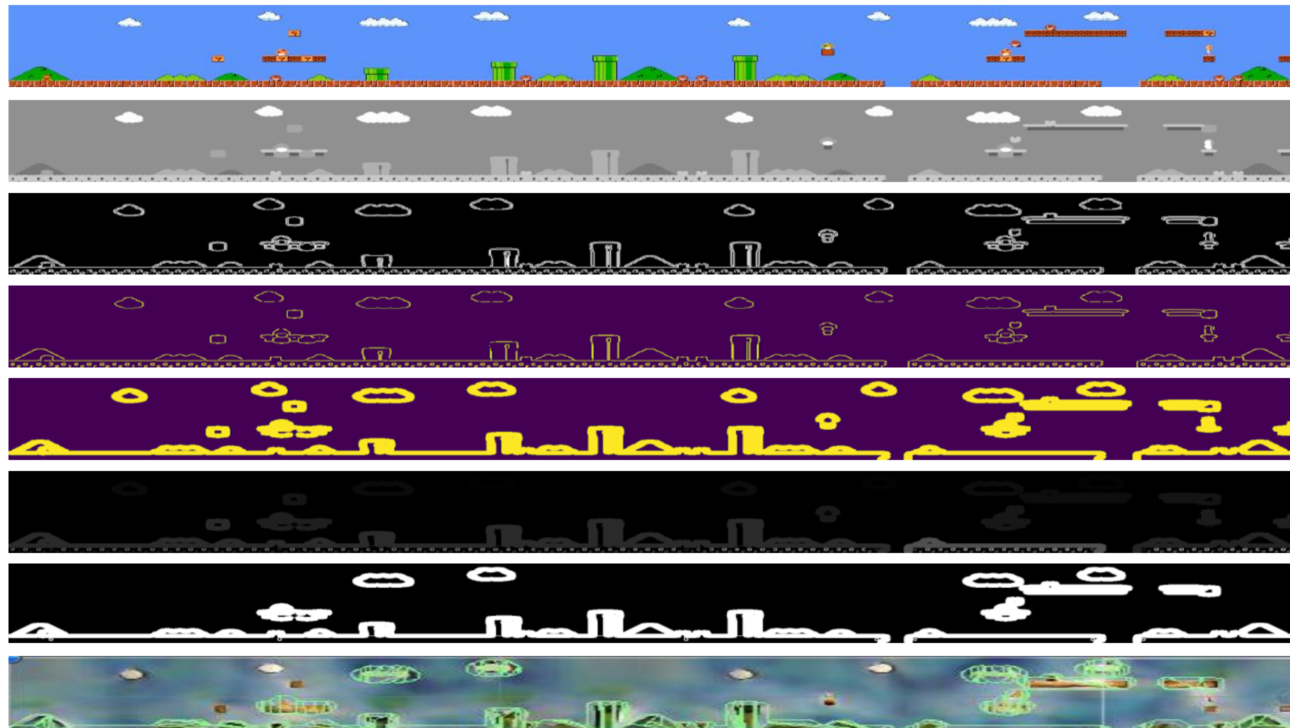

Figure 5. Results of the forward edge pipeline, from top to bottom, intermediate results: (a) Mario game content image, (b) Grayscale dilation, (c) Canny edge detector, (d) Contours, (e) Thickened contours, (f) Connected components, (g) Create feature mask, and (h) Collider generation

Figure 6. Results of the backward edge pipeline, from top to bottom, intermediate results: (a) Mario game content image, (b) Style transferred image, (c) Grayscale dilation, (d) Canny edge detector, (e) Contours, (f) Thickened contours, (g) Connected components, (h) Create feature mask, and (i) Collider generation

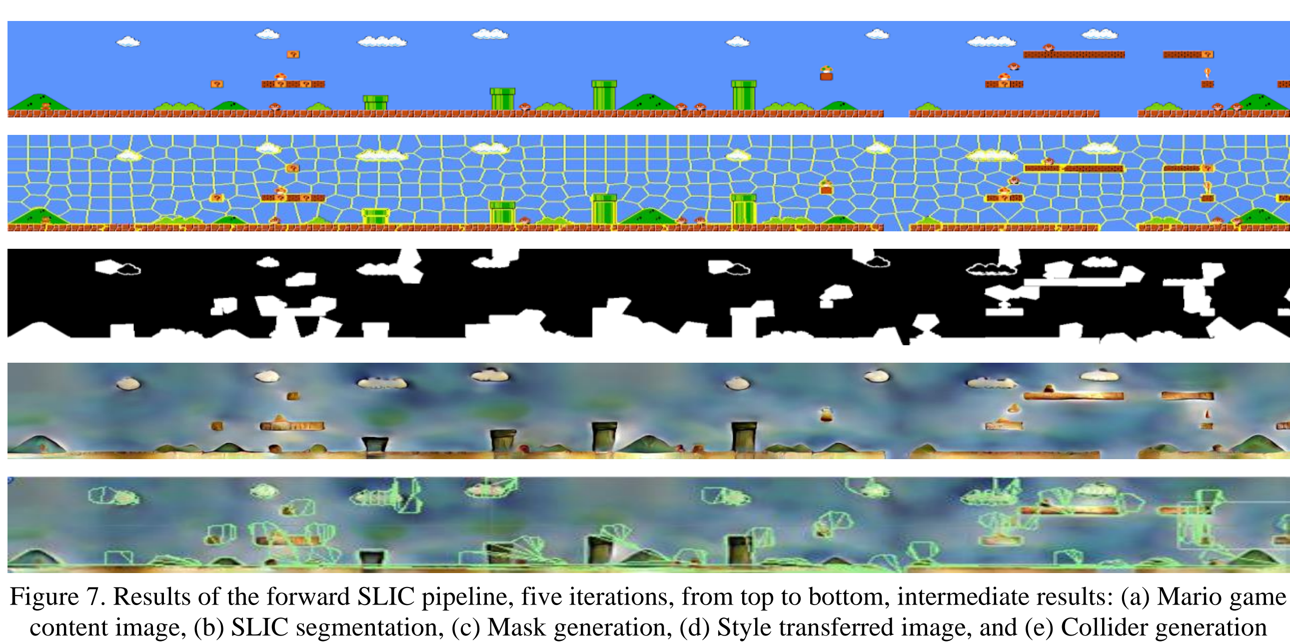

Figure 7. Results of the forward SLIC pipeline, five iterations, from top to bottom, intermediate results: (a) Mario game content image, (b) SLIC segmentation, (c) Mask generation, (d) Style transferred image, and (e) Collider generation

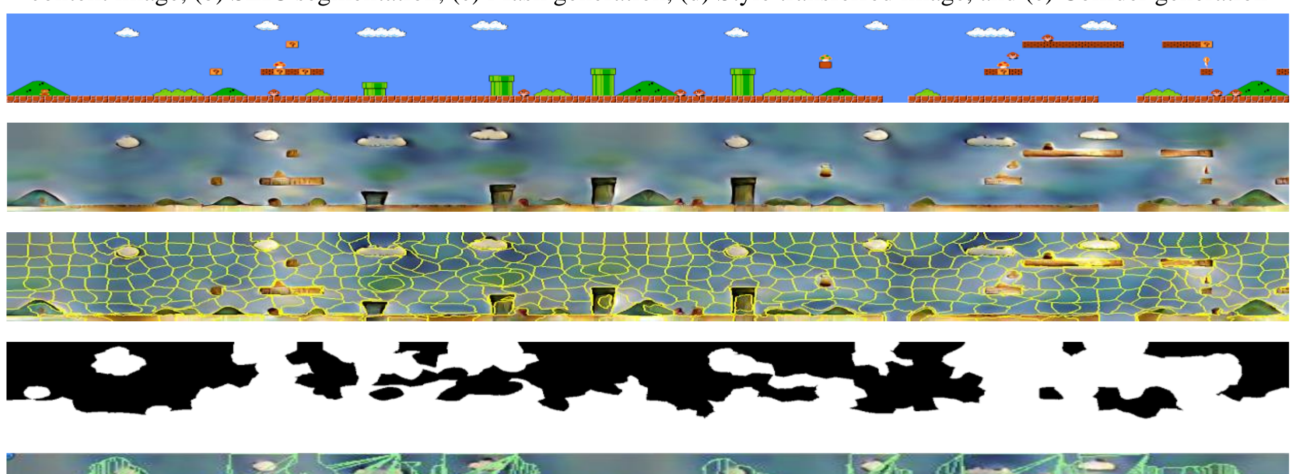

Figure 8. Results of the backward SLIC pipeline, five iterations, from top to bottom, intermediate results: (a) Mario game content image, (b) Style transfer, (c) SLIC segmentation, (d) Mask generation, and (e) Collider generation

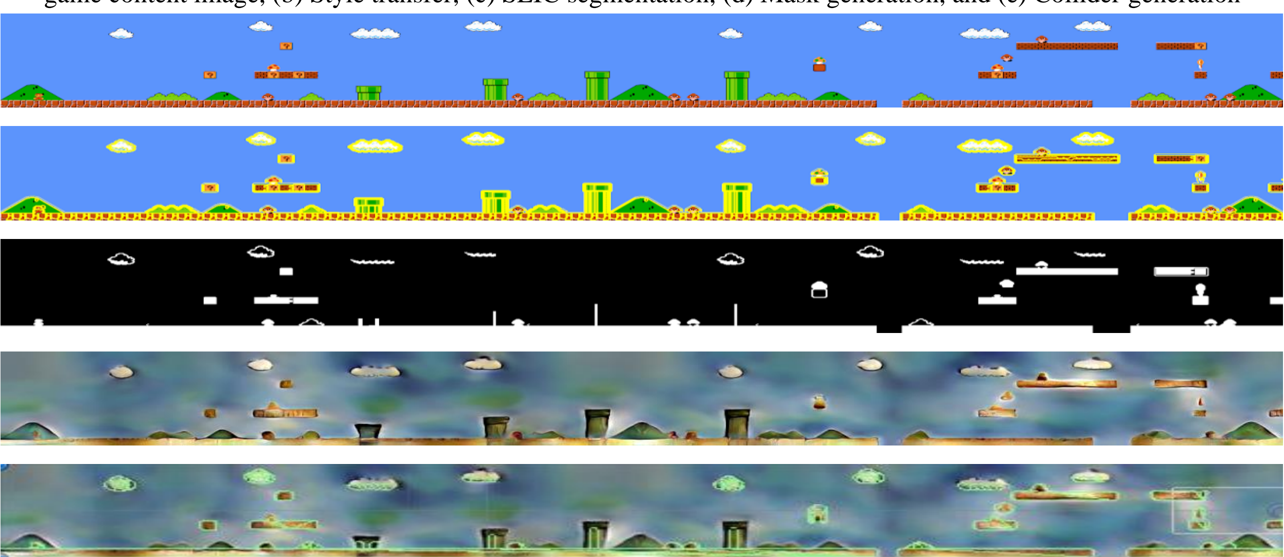

Figure 9. Results of the forward Felzenszwalb pipeline, five iterations, from top to bottom, intermediate results: (a) Mario game content image, (b) Felzenszwalb segmentation, (c) Mask generation, (d) Style transferred image, and (e) Collider generation

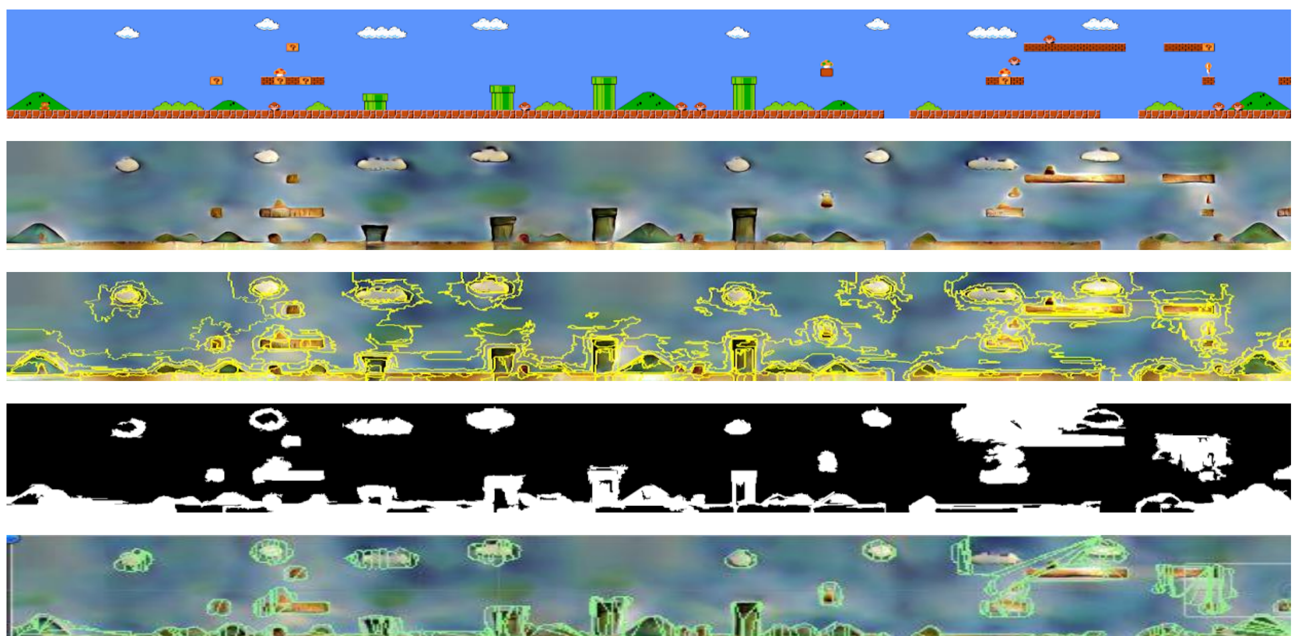

Figure 10. Results of the backward Felzenszwalb pipeline, five iterations, from top to bottom, intermediate results: (a) Mario game content image, (b) Style transferred image, (c) Felzenszwalb segmentation, (d) Mask generation, and (e) Collider generation

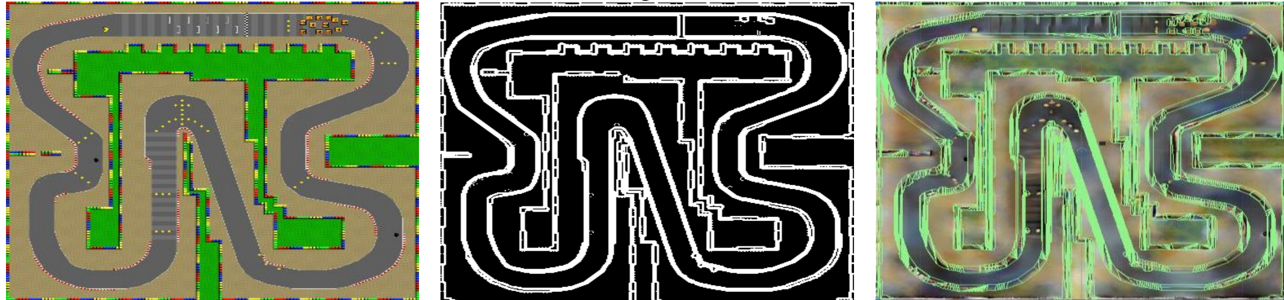

Figure 11. From left to right, one iteration: (a) Original Mario Kart game level image, (b) Output of the mask generation with the forward edge pipeline, and (c) Colliders generated by the Unity game engine

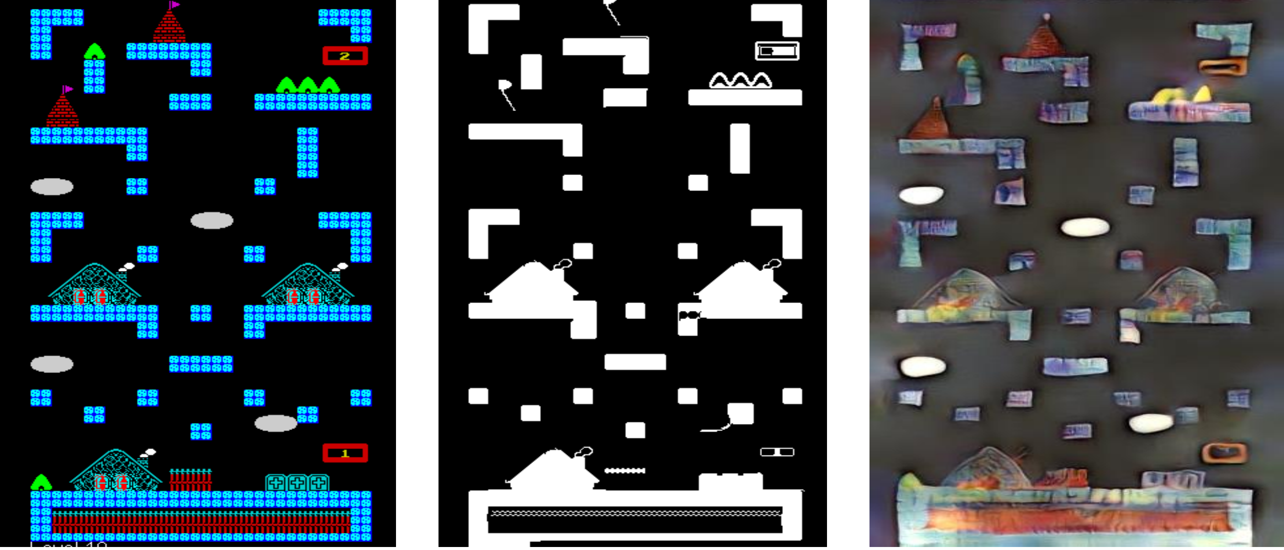

Figure 12. From left to right, four iterations: (a) Original of Rainbow Islands, (b) Output of the mask generation with the forward Felzenszwalb pipeline, and (c) Styled game level image

Figure 13. From top to bottom, left to right, three iterations: (a) Original level image of Lode Runner, (b) Binary mask generated with forward the SLIC pipeline, and (c) Styled game level image

Figure 14. From left to right, two iterations: (a) Original game level of Kid Icarus, (b) Binary mask generated with forward Felzenszwalb pipeline, and (c) Styled game level image

3.3. Quantitative Results

The outputs of the pipelines differ drastically based on the game they were run with. On the visual outputs, the colliders are the most informative pieces of data in terms of the overall performances of the pipelines and their game mechanics generation processes. On the other hand, the colliders are generated based on the outputs of the pipelines, which are the binary masks and the components inside them. Each pipeline was run to obtain the collider coverage, the connected component number, the path sizes, and the RAM usages of the final game.

The collider coverage can be defined as the percentage of the pixels that are generated to be a collider over the complete game level image. In Table 1, it is possible to see the collider coverage values of each iteration of pipelines on five games. The collider coverage values deviate across the pipelines within a range between 7.34 and 15.55, which is a relatively small difference in the context of video games considering their versatility in visual features. This situation can also imply that one can have some accurate predictions about the incoming collider coverage by looking at the visual features of the raw level image before the game mechanics generation is applied. Conversely, the average collider coverage percentages vary drastically, such that they are spread between 19.6 and 66.4 —i.e., a difference of 46.4. This implies the fact that the collider coverage is highly dependent on the visual features and is not a normalized metric across the pipelines. Backward SLIC pipeline produced extreme situations as the coverages vary between 57 and 78%, which are very high percentages, and in such maps, a player can struggle to find the places to move around. Another intermediate metric is the number of connected components in the generated mask. Although the final number of colliders can give more information about the overall performance of the pipelines, we are using the number of the connected components because the Unity game engine treats the colliders inside a collider map as a single collider, which does not signify any information. However, the collider generation is essentially fitting some polygons inside the given mask to highlight the parts to be filled (Figure 11). Thus, combining the collider coverage along with the number of connected components can give a good estimation of the overall performance. For instance, Figure 11b signifies the output of the game mechanics generation process with the forward edge pipeline applied on a level image of Mario Kart.

The collider coverage of this image with the forward edge pipeline is 32%, and the number of components is one, according to Table 1. This single map is converted into a collider map, which can be seen in Figure 10c. Therefore, a single large component is converted into a high number of colliders in the Unity environment. Connected component values are directly related to the image features that are relatively far from each other such that they are not clustered together or connected during the dilation process. According to the standard deviation values of the connected component numbers for different pipelines in Table 2, there can be some extreme situations, such as a component occupying 78% of the image or a component occupying 0.25%. Again, the backward SLIC pipeline has some high results where the colliders occupy a great portion of the whole image, and the number of components varies between 1 and 11. Thus, each component occupies bigger portions of the map rather than the components of the other outputs of the pipelines.

In Table 3, the results of the path size and the RAM usage of the five games according to different pipelines are presented. Game pipeline-based evaluation is meaningful since the path sizes are found according to each pipeline's binary masks. The path size signifies the number of significant colliders generated by the Unity game engine. Each significant collider has smaller regions located inside them; therefore, a smaller path size can signify a less complex collider graph. For instance, if the mask generated from any pipeline possesses a connected component that has a high density, it is likely that a single large polygon can be fit into that component, thus reducing the number of polygons to be generated in order to cover each component. Therefore, it is possible that there is an inverse correlation between the amount of space each component occupies and the path size. The backward SLIC pipeline produced high-density components, as can be seen in Table 2, and the path size of this pipeline is the smallest of all the outputs, as can be seen in Table 3.

On RAM usage of the generated games, the expectation would be in the way that a denser collider map, which is directly related to the collider coverage, would use more memory. However, such a correlation is not present in the results shown in Table 2 and Table 3. On the other hand, memory usage seems to be related to the area of the map, which is valid in theory, as in the simplest terms, the more the size of the area means more data to keep in RAM. Therefore, the memory usage is more correlated with the dimensions of the game level image, regardless of the pipeline, given that the segments are overlapped completely in terms of their shapes.

The collider generation step is also more effective in terms of the precision of the edges of the colliders. The backward pipeline in Figure 10 performs in a different way; the segments are not as well-defined as the forward pipeline. However, looking at the overall mask generated from the variance-based thresholding, the result is well-organized and well put together, apart from some cases, for instance, the places where the clouds are too close to the platforms.

The output of the collider generation is also unusual because the estimated polygons seem to fit well in order to create an overall playable map. It is not easy to come to a conclusion on the superiority of either of the pipelines, as there are cases where they perform better or worse. However, because styling an image creates new dimensions and breaks the 2D properties of the games, backward pipelines tend to create unstable colliders. Although it can be argued that having unexpected colliders on the map can have some ways of entertainment, it is likely to be frustrating for the players as well.

Table 2. Collider coverages and connected component counts

Generated Games and Their Mechanics	Game Mechanics Generation Pipelines											
	Forward Edge		*Backward Edge*		*Forward SLIC*		*Backward SLIC*		*Forward Felzenszwalb*		*Backward Felzenszwalb*	
	Collider Coverage (%)	*Number of Connected Components*	*Collider Cov. (%)*	*Conn. Comp. Count*	*Collider Cov. (%)*	*Conn. Comp. Count*	*Collider Cov. (%)*	*Conn. Comp. Count*	*Collider Cov. (%)*	*Conn. Comp. Count*	*Collider Cov. (%)*	*Conn. Comp. Count*
Super Mario	14	10	14	3	34	30	59	6	17	33	22	30
Super Mario Kart	32	1	23	3	79	2	78	1	19	49	56	61
Rainbow Islands	6	3	35	11	62	14	77	8	37	76	40	59
Lode Runner	27	1	34	1	43	12	57	3	30	4	50	2
Kid Icarus	19	53	22	20	54	19	61	11	24	95	32	60
Std. Dev.	9.2	20	8	7.1	15.6	9.2	9.2	3.5	7.3	31.9	12.2	23.3
Mean	19.6	13.6	25.6	7.6	54.4	15.4	66.4	5.8	25.4	51.4	40	42.4

Table 3. Path sizes and RAM usages of the generated games

Generated Games and Their Performance Outcomes	Game Mechanics Generation Pipelines													
	Level Image Width (pixels)	*Level Image Height (pixels)*	*Forward Edge*		*Backward Edge*		*Forward SLIC*		*Backward SLIC*		*Forward Felzenszwalb*		*Backward Felzenszwalb*	
			Path Size	*RAM (MB)*	*Path Size*	*RAM (MB)*	*Path Size*	*RAM (MB)*	*Path Size*	*RAM (MB)*	*Path Size*	*RAM (MB)*	*Path Size*	*RAM (MB)*
Super Mario	3232	208	24	495	31	502	34	507	12	509	151	518	66	512
Super Mario Kart	1024	1024	97	541	134	495	58	558	52	559	40	561	90	518
Rainbow Islands	297	1404	17	562	212	572	27	573	29	579	113	566	128	586
Lode Runner	275	214	21	587	48	585	16	594	22	594	5	598	13	604
Kid Icarus	256	2780	52	603	156	603	33	613	28	611	105	613	100	617
Std. Dev.	1144.9	948.5	30.1	37.8	67.8	44.4	13.8	36.2	13.2	35.2	52.8	33	38.7	43.9
Mean	1016.8	1126	42.2	557.6	116.2	551.4	33.6	569	28.6	570.4	82.8	571.2	79.4	567.4

Particularly, in the backward edge pipeline, most of the edges are, in fact, detected, but their thickened versions and their passes from the thresholding based on the pixel numbers create unpleasing results; however, the places that are kept fit on the collider space in a nice way. In the backward color pipelines, the qualities of the outputs vary. For instance, in the Felzenszwalb backward pipeline (Figure 6e), the colliders on the left side of the level image are as expected, but the ones on the right side are not playable. Overall, it can be concluded that the generation of the collider bitmap before the stylization process is a more reliable way of processing the level image.

The outcomes of the segmentation types also differ given that both color- and edge-based segmentation have flawed parts, which is understandable considering the fact that the game levels have varying parts with shapes and colors in it. Edge-based segmentation seems to be accurate in terms of the positions and shapes of the objects. However, the colliders seem to be larger than they should be because of the thickening process, which is a necessary procedure for connected components. Color-based segmentation does not have such a problem as the extracted segments are used as they are. However, because the elimination is done via the variances, the output of this thresholding process may not always give the desired output since it can eliminate the segments that could be meaningful when overlapped with the colliders.

4. CONCLUSION

Video game research has become a highly active research field where the steps on the simplification and automation

of the game development processes —i.e., procedural content generation— had a huge impact. This study has been a first attempt on merging the neural style transfer and game mechanics generation in a single game development pipeline. Even with some of the most basic edge detection and segmentation algorithms and the use of already available collider tools, it is shown that video game automatization could be possible, and our study provides a fast, effective and enhanced game mechanics generation approach. Thus, this study underlines the potential usage, feasibility, and effectiveness of the neural style transfer with a focus on game mechanics generation.

This study has been a prototype in the field of neural style and game mechanics transfer, and there are methods and techniques yet to be tried or discovered to elaborate this proposed game generation pipeline forward. In this case, for the game mechanics generation part, only some of the most basic yet effective image processing methods were used; however, deep learning methods are also suitable for such practices, for instance, generative models [26]. They can also be more adaptable to the unseen types of scenes as in versatile domains such as video games, and non-linear relationships are likely to represent their generic properties rather than a series of image processing techniques.

Although the authors have tested the updated games intensively, future work will focus on the playability and usability of the games by a diverse group of participants. These tests will include the well-known benchmark games that were introduced in this study as well as the authors' unique games that will be enhanced with the current study's algorithmic approach.

Another case that can be improved using generative models is the complex game mechanics generation. In this study, the addition of basic colliders has been the most suitable solution in terms of game mechanics generation. However, 2D games can have different types of game mechanics, such as rotation, tilt, and move. In future studies, new models can also be trained in order to produce more complex game mechanics.

REFERENCES

[1] H. Ragib, S. Chakraborti, M. Z. Hossain, T. Ahamed, M. A. Hamid, M. F. Mridha, "Character and Mesh Optimization of Modern 3D Video Games", **Advances in Data and Information Sciences**, 655–666, Springer, Singapore, 2020.

[2] S. Bart, P. Dobrowolski, M. Skorko, J. Michalak, A. Brzezicka, "Issues and advances in research methods on video games and cognitive abilities", *Frontiers In Psychology*, 6(1451), 1–7, 2015.

[3] M. Csikszentmihalyi, M. Csikzentmihaly, **Flow: The psychology of optimal experience**, New York: Harper & Row, 1990.

[4] N. Shaker, J. Togelius, M. J. Nelson, **Procedural Content Generation in Games: A Textbook and an Overview of Current Research**, NewYork, NY, USA: Springer-Verlag, 2016.

[5] L. A. Gatys, A. S. Ecker, M. Bethge, "Image style transfer using convolutional neural networks", **IEEE Conference on Computer Vision and Pattern Recognition**, Las Vegas, NV, USA, 2016.

[6] F. Luan, S. Paris, E. Shechtman, K Bala, "Deep photo style transfer", **IEEE Conference on Computer Vision and Pattern Recognition**, Honolulu, HI, USA, 2017.

[7] J. J. Virtusio, A. Talavera, D. S. Tan, K. Hua, A. Azcarraga, "Interactive style transfer: Towards styling user-specified object", **IEEE Visual Communications and Image Processing (VCIP)**, Taichung, Taiwan, 2018.

[8] Y. Li, C. Fang, J. Yang, Z. Wang, X. Lu, M. Yang, "Universal style transfer via feature transforms", **The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)**, Long Beach, CA, USA, 2017.

[9] K. Ziga, J. Bagchi, J. P. Allebach, F. Zhu, "Non-parametric texture synthesis using texture classification", *Electronic Imaging*, 17, 136–141, 2017.

[10] A. A. Efros, W. T. Freeman, "Image quilting for texture synthesis and transfer", **28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'01)**, Los Angeles, CA, USA, 2001.

[11] J. Johnson, A. Alahi, L. Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution", **European Conference on Computer Vision (ECCV 2016)**, Amsterdam, The Netherlands, 2016.

[12] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, J. Togelius, "Procedural content generation via machine learning (PCGML)", *IEEE Transactions on Games*, 10(3), 257–270, 2018.

[13] S. Snodgrass, S. Ontanón, "Learning to generate video game maps using markov models", *IEEE Transactions on Computational Intelligence and AI in Games*, 9 (4), 410–422, 2016.

[14] M. Guzdial and M. Riedl, "Learning to blend computer game levels", **7th International Conference on Computational Creativity (ICCC 2016)**, Paris, France, 2016.

[15] J. Gow, J. Corneli, "Towards generating novel games using conceptual blending", **Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-15)**, Santa Cruz, CA, USA, 2015.

[16] A. J. Summerville, S. Snodgrass, M. Mateas, S. Ontanón, "The vglc: The video game level corpus", *arXiv preprint arXiv:1606.07487*, 2016.

[17] A. Polesel, G. Ramponi, V. J. Mathews, "Image enhancement via adaptive unsharp masking", *IEEE Transactions on Image Processing*, 9(3), 505–510, 2000.

[18] P. Bao, L. Zhang, X. Wu, "Canny edge detection enhancement by scale multiplication", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(9), 1485–1490, 2005.

[19] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, S. Süsstrunk, "SLIC superpixels compared to state-of-the-art superpixel methods", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(11), 2274–2282, 2012.

[20] E. B. Alexandre, A. Shankar Chowdhury, A. X. Falcao, P. A. V. Miranda, "IFT-SLIC: A general framework for superpixel generation based on simple linear iterative clustering and image foresting transform", **28th SIBGRAPI Conference on Graphics, Patterns and Images**, Salvador, Bahia, Brazil, 2015.

[21] P. Felzenszwalb, D. Huttenlocher, "Efficient Graph-Based Image Segmentation", *International Journal of Computer Vision*, 59 (2), 167–181, 2004.

[22] Internet: Unity Technologies–Unity 3d., http://unity3d.com/, 24.07.2020.

[23] W. Goldstone, **Unity 3. x game development Essentials**, Packt Publishing Ltd, 2011.

[24] C. Ericson, **Real-time collision detection**, CRC Press, 2004.

[25] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P., P. Mawhorter, G. Takahashi, G. Smith, "The 2010 Mario AI championship: Level generation track", *IEEE Transactions on Computational* Intelligence and AI in Games, 3(4), 332–347, 2011.

[26] D. J. Rezende, S. Mohamed, D. Wierstra, "Stochastic backpropagation and approximate inference in deep generative models", *arXiv preprint arXiv:1401.4082*, 2014.