Synthesizing Retro Game Screenshot Datasets for Sprite Detection

Chanha Kim, Jaden Kim, Joseph C. Osborn
Formal Analysis of Interactive Media Lab
Pomona College
185 East 6th Street
Claremont, California 91711
{chanha.kim, jaden.kim, joseph.osborn}@pomona.edu

Abstract

Scenes in 2D videogames generally consist of a static terrain and a set of dynamic *sprites* which move around freely. AI systems that aim to understand game rules (for design support or automated gameplay) must be able to distinguish moving elements from the background. To this end, we re-purposed an object detection model from deep learning literature, developing along the way *YOLO Artificial Retro-game Data Synthesizer*, or YARDS, which efficiently produces semi-realistic, retro-game sprite detection datasets without manual labeling. Provided with sprites, background images, and a set of parameters, the package uses sprite frequency spaces to create synthetic gameplay images along with their corresponding labels.

Introduction

Many videogames employ a visual language which presents a static terrain (with background and foreground elements) juxtaposed with dynamic, animated, and freely moving *sprites*. Sprites are often game characters or objects of interest: potential threats, powerups, or the player's character. Knowledge about these sprites (e.g. their type, location, or speed) is vital for AI systems meant to understand games, especially in systems such as automated game design learning (Osborn, Summerville, and Mateas 2017) and learning-based level generation (Guzdial and Riedl 2016; Summerville et al. 2016a) as well as general videogame playing and automated approaches to accessibility.

Besides manual image labeling, current methods for obtaining sprite segmentations generally involve game-specific image processing or deep instrumentation of game emulators, as in CHARDA (Summerville, Osborn, and Mateas 2017). These techniques can be difficult to generalize and may be expensive, slow, or potentially fragile (e.g. the locations of hardware sprites in memory do not correspond exactly to the positions of human-legible sprites). In this work, we instead generate synthetic images starting from readily accessible *spritesheets* and sprite-free background images. We can then use the synthetic datasets generated from these

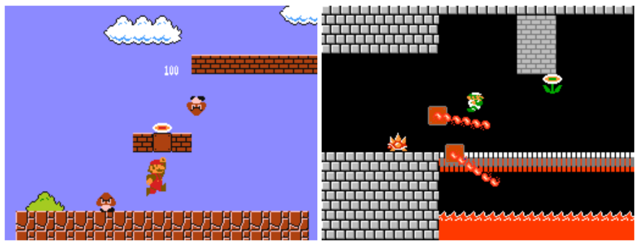

Figure 1: Real (left) vs. Synthetic (right) Screenshot from *Super Mario Bros.* on NES

resources to train sprite detection models for their corresponding games, which should generalize better than classical computer vision techniques or emulator instrumentation.

While our approach still requires that users collect spritesheets and background images, it eliminates the need for manually labeling images or writing computer vision code. We have packaged this generator in a Python package called *YOLO Artificial Retro-game Data Synthesizer* (YARDS). YARDS is a command-line tool which, given sprites and background images, generates synthetic training images that mimic patterns found in real game screenshots. YARDS pre-formats the synthetic data for integration with YOLO and can generate 1,000 labeled images in 10 seconds—a task which took the authors 10 long hours!

In this paper, we present our two key contributions. First, we apply recent progress made in deep learning and computer vision to aid in producing semi-realistic datasets suitable for training sprite detection models. Second, we introduce a software package that can rapidly generate large datasets. In the following sections, we will discuss our approach for synthesizing retro-game screenshots, demonstrate how YARDS works, and evaluate several models trained on synthetic data against those trained on manually labeled images.

Related Works

The intersection of computer vision and games is a growing area of research that can benefit both designers and players. In 2018, Luo et al. demonstrated how transfer learning

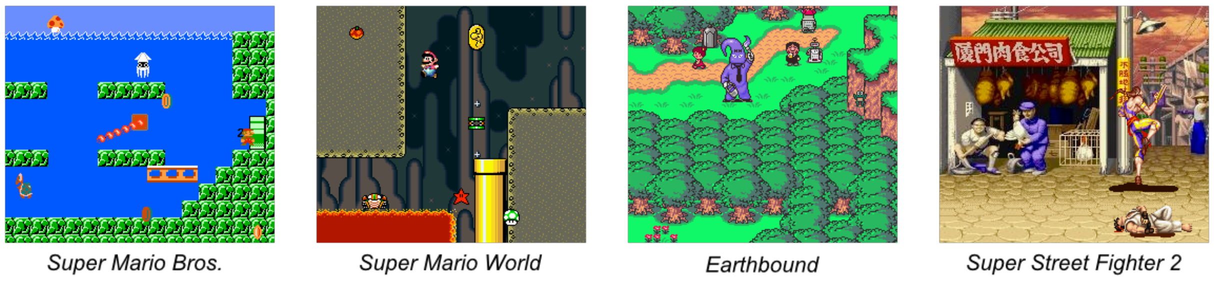

Figure 2: Examples of Synthetic Images Generated with YARDS

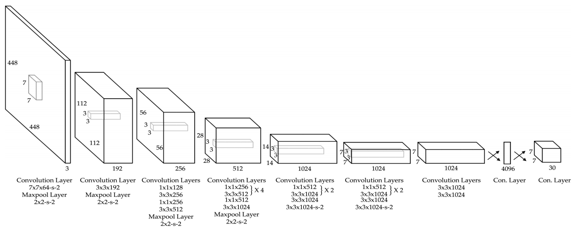

Figure 3: Adapted diagram of the original YOLO architecture (Koylu, Zhao, and Shao 2019)

could improve the task of extracting player experiences directly from gameplay videos. In the same year, Zhang et al. introduced the problem of content-based retrieval of game moments and presented a prototype search engine for retrieving such moments based on user-provided game screenshots (2018). Our research contributes to this area by applying synthetic data generation techniques to the task of training sprite detection models and by introducing a software package that supports end-users with synthesizing their own datasets.

In our research, we use YOLO (*You Only Look Once*) to detect sprites in four retro games: *Super Mario Bros.* (NES), *Earthbound* (SNES), *Super Street Fighter II* (SNES), and *Super Mario World* (SNES). YOLO is a well-known object detection model (Redmon et al. 2015) which has seen several iterations (Redmon and Farhadi 2016; 2018; Bochkovskiy, Wang, and Liao 2020). The original model's architecture is inspired by GoogLeNet (Szegedy et al. 2014) and has 24 convolutional layers followed by 2 fully connected layers (see Fig. 3). This end-to-end model allows for efficient training and detection of objects in both images and videos. The specific YOLO version that we are using is Ultralytic's YOLOv5 (Ultralytics 2020), a recent implementation of YOLO in PyTorch.

The use of synthetic data to train models is a well-known practice in computer vision. Motivations for generating synthetic datasets include the high cost of manually labelling real images (Roig et al. 2020), privacy concerns when using real user data (Triastcyn and Faltings 2018; Shaked and Rokach 2020), and the shortage of real training examples for rare cases (Beery et al. 2019). Other arguments for using synthetic data include the lack of extensive datasets in niche application domains (Wong et al. 2019) and the under-representation of the full target distribution in small datasets (Lateh et al. 2017). For such reasons, researchers have suggested numerous approaches to generating synthetic data over the past decade (Nikolenko 2019; Seib, Lange, and Wirtz 2020).

One common approach is to first generate a synthetic image and then stylize the image to be more realistic (Dwibedi, Misra, and Hebert 2017; Georgakis et al. 2017; Wong et al. 2019). For example, Wang et al. 2019 generated photorealistic synthetic images using a virtual 3D object-environment reconstruction method and style transfer techniques. Hinterstoisser et al. 2019 used 3D CAD models and pose curricula to generate foreground-background compositions, then made those compositions photorealistic via rendering techniques.

A significant issue arising from synthetic datasets is the synthetic-to-real domain gap (Tremblay et al. 2018; Yun et al. 2019b). This gap occurs when the synthetic images used to train a model are not representative of the target image distribution. In terms of model performance, research shows that models trained with both real and synthetic images achieve the best performance, followed by models trained with purely real images (Rozantsev, Lepetit, and Fua 2015; Dwibedi, Misra, and Hebert 2017; Georgakis et al. 2017; Rajpura, Bojinov, and Hegde 2017; Yun et al. 2019a). These studies also demonstrate that training with purely synthetic images seems to detract from model performance. However, comparable performance is achievable in image segmentation tasks (Di Cicco et al. 2017) and fine-tuning models trained on synthetic data with additional real images can yield better performance than mixed training (Nowruzi et al. 2019).

Our synthetic data generation approach is most similar to the one presented by Dwibedi et al. 2017. Since our games of interest use a pixelated art style, we can skip the step of increasing the realism of the generated images; it is enough to simply paste the sprites onto the backgrounds in a reasonable distribution. We therefore focus on developing an efficient technique for pasting sprites onto background images based on sprite frequency distributions observed in real gameplay images.

Synthetic Data Generation Approach

Our approach involves two steps: first, collect the sprites and background images for a given game; and second, paste sprites onto background images using their frequency distributions. Both sprites and background images are easily obtainable by extracting the data from an emulator, borrowing from archives compiled by fans, or utilizing approaches proposed by researchers like Summerville et al. 2016b. Additional methods for obtaining sprites and background images include scripting some game-specific image extraction code or providing the assets directly if the user is the one developing the game.

One reason why our approach is so effective in the videogame domain is because we are working with much smaller image spaces (sets of possible images) than the real-world image spaces typically used in computer vision tasks. While the immense complexity of real-world images can depend on virtually anything, from lighting conditions to object textures, our focus on low-resolution, retro-game screenshots allows us to synthesize realistic screenshots just by pasting sprites onto background images.

Sprite Frequency Spaces

Even though we are working with low-resolution images, synthesizing images that roughly mimic those seen in real gameplay is a nontrivial task. We generate synthetic images by providing the sprite frequency space for each class of sprites to be detected. We define the sprite frequency space for a given class as the discrete probability distribution over the frequency of appearances for that class on a given gameplay image. That is, it is a function mapping the numbers of sprites in a class to the probabilities that those numbers of sprites actually appear on a screenshot.

These sprite frequency spaces can either be defined by the user (as in our reported results) or approximated by feeding pre-labeled images into YARDS. At runtime, YARDS will approximate the sprite-frequency spaces by counting the frequencies of the desired sprite classes in the image labels. Using these sprite frequency spaces, we can determine the number of times each sprite class should appear on each generated image. For example, in *Super Mario Bros.* (NES), there should almost always be one player on the screen, while there may be any number of enemy sprites from zero to 16, each number appearing at a different rate.

For each output image, we choose the number of times each class of sprites should appear by sampling the class's corresponding sprite frequency space. We thereby incorporate the given or estimated frequencies with which our target sprite classes appear in authentic gameplay images. This allows the YOLO model to train on data that roughly mimics our target distribution and avoid the previously-discussed domain gap issues.

Edge Handling with Transparency Quadrants

Given that sprites often have transparent pixels, we must ensure that sprites are at least partially visible in screenshots no matter their shape. For example, an L-shaped sprite placed in the lower left corner of the screen might have no visible pixels (i.e., opaque pixels) in the screenshot (see Fig. 4). Training a model would be very difficult if our training set asserts that background pixels are in fact part of a character. To handle these edge cases, we propose an approach that relies on transparency quadrants for determining whether enough of the sprite is visible in the screenshot.

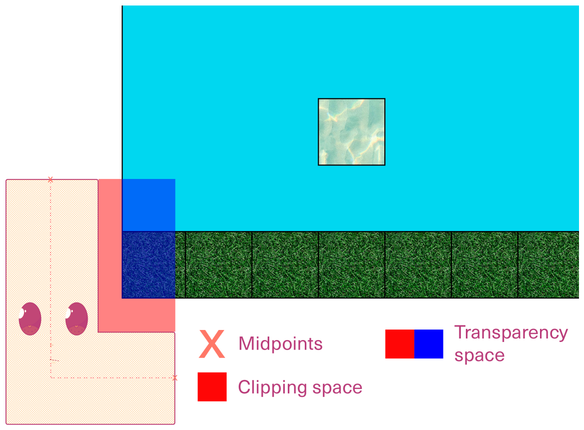

Figure 4: Example of sprite clipping on worst-case scenario of L-shaped sprite in bottom left corner of screen. Midpoints are where sprite will be cropped.

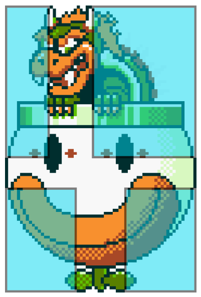

Figure 5: Example of top-left, top-right, bottom-left, and bottom-right quadrants in a clipped sprite

For a given sprite, we first determine its transparency quadrants by using the locations of the first and last visible (non-transparent) pixels along each of its border axes. The upper-left quadrant (Fig. 5) is determined by the first visible pixel in the first row of visible pixels and the first visible pixel in the first column of visible pixels. Corresponding horizontal and vertical lines are drawn from each pixel, and their intersection defines the quadrant. Similarly, the bottom-right quadrant (Fig. 5) is determined by the last visible pixel in the last row of visible pixels and the last visible pixel in the last column of visible pixels. The upper-right and bottom-left quadrants are formed analogously. In general, the quadrant is defined by the intersection of the horizontal and vertical lines drawn from each relevant pixel.

Once we have the sprite's transparency quadrants, we determine where the sprite is clipped by the boundaries of the screenshot. For example, if it is partially off the left side of

Table 1: YARDS Configuration Parameters

game_title	The game's title, which is prepended to each image's filename to avoid naming conflicts.
num_images	The total number of images to generate.
train_size	The proportion of total images which should be included in the train set.
mix_size	The proportion of total real images which should be included in the train set.
label_all_classes	Determines whether all classes should be labeled or if only specific classes should be.
labeled_classes	Determines which classes to label if `label_all_classes` is false. Useful for focusing attention on a single sprite and introducing noise in the form of other sprites or random images.
max_sprites_per_class	The maximum number of sprites per class which can appear in any given image. If set to -1, no cap will be set. Provides a means for limiting noise. Useful primarily when setting `classification_scheme` to `random`, as it allows for more control of the distribution.
transform_sprites	Another means for introducing noise. If set to true, transforms sprites by rotating a multiple of ninety degrees, mirroring, or scaling to twice their original size. The reason for the set scaling is because pixel art gets distorted by any non-double scaling.
clip_sprites[1]	Determines whether to keep all sprites entirely on screen or to allow some sprite clipping.
classification_scheme[2]	Determines the classification scheme by which to place sprites.

the screenshot, we check the right-most quadrants (Fig. 5). Then, if the larger width of the two quadrants (i.e., the *transparency space*) is greater than the width of the sprite that is visible after clipping (i.e., the *clipping space*), not enough of the sprite is within the boundaries of the screenshot. For instance, assume the transparency space is greater than or equal to the clipping space for a sprite being clipped off the left of an image. In this case, we average the x-positions of the inner vertical edges of all four transparency quadrants. Then, we crop the sprite to be from the resulting average x-position to the sprite's rightmost border and paste the cropped sprite into the screenshot with the sprite's left border aligned with the screenshot's left border. Because we use all four transparency edges to determine how to crop the sprite, we know that enough of the sprite's useful information will appear on the generated image. We perform an analogous procedure for each screen boundary that the sprite overlaps.

Using YARDS

Integrating YARDS into object detection projects is simple. The development pipeline with YARDS involves three main steps: preprocessing, synthetic data generation, and model training. In preprocessing, we gather sprite and background images for a given game and define the corresponding folder locations in the configuration file. During synthetic data generation, we define the parameters in the rest of the configuration file and run YARDS via command line to generate the images. The command-line package for YARDS takes up to two parameters. The configuration parameter (`--config` or `-c`) defines the path to the configuration file, and the visualize parameter (`--visualize` or `-v`) tells the package to draw bounding boxes for a sample of images. Finally, in model training, we train the model and validate it in a conventional machine learning pipeline.

Configuration Parameters

YARDS has multiple configuration parameters that the user must define prior to using the package. Table 1 summarizes what each parameter does, and the complete documentation is available in the project's source code repository.

`clip_sprites`[1] and `classification_scheme`[2] are the parameters that control our synthetic data generation approach. `clip_sprites`[1] determines whether or not the synthetic screenshots should have clipped sprites. `classification_scheme`[2] accepts one of four keywords that define different methods for characterizing the sprite frequency space: `distribution`, `mimic-real`, `random`, and `discrete`. The `distribution` method takes a set number of predefined classes such as `player`, `enemy`, or `item` and corresponding sprite frequency spaces for each class, represented by an array. For instance, `player: [0.20, 0.40, 0.40]` means that for the `player` class, zero sprites should appear twenty percent of the time, one sprite should appear forty percent of the time, and two sprites should appear forty percent of the time. The `mimic-real` method analyzes a set of pre-labeled images to approximate the sprite distribution in a dataset and takes as input an array of class numbers, which correspond to the class numbers in the image labels. It then uses the approximated distributions to generate the images. The `random` method samples each class with a uniform distribution, given the maximum number of sprites for each class. The `discrete` method takes inspiration from games like *Street Fighter II* where each screen has a constant number of sprites, and it takes a constant number of sprites to display on each screenshot.

Tests and Results

To evaluate our dataset synthesizer we compare model performance for different datasets from a single game, train binary classifiers to detect real versus synthetic data, and demonstrate the generalizability of our approach.

Training on Synthetic, Real, and Mixed Datasets

To compare the performance of models trained on synthetic data to those trained on real data, we trained nine YOLO

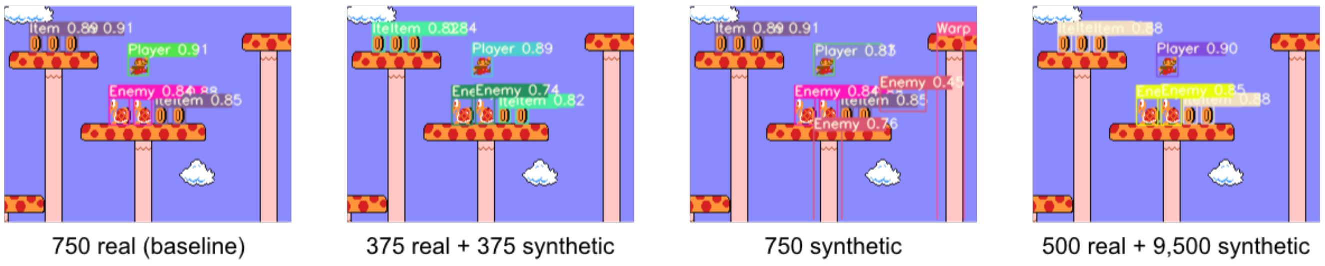

Figure 6: Detection Results of Models Trained on Various Datasets for *Super Mario Bros.*

models on various datasets for *Super Mario Bros.* (screenshot dimension: 256×192). Each YOLOv5 model (Ultralytics 2020) was trained for 200 epochs with batch-size 32, and the best weights for each model (i.e. the weights that yielded the best model performance in training) were validated on 250 real images. We used mAP@0.5, a mean average precision metric for measuring the performance of object detection models, as our single-valued evaluation metric.

Table 2: Mean average precision of YOLO models trained on various datasets for *Super Mario Bros.* (Better-than-baseline performance is in bold.)

Training Set Composition	mAP@0.5
750 real (baseline)	0.856
375 real + 375 synthetic	**0.886**
75 real + 675 synthetic	0.841
750 synthetic	0.677
750 synthetic + fine-tuning w/ 750 real	**0.881**
375 real + 3,375 synthetic	**0.956**
3,750 synthetic	0.816
500 real + 9,500 synthetic	**0.963**
10,000 synthetic	**0.935**

Table 2 confirms the results shown by researchers in other image recognition domains, suggesting that training mixed datasets of real and synthetic images yields the best model performance—although models trained on 3,750 and 10,000 synthetic images show that training with large purely synthetic datasets can also work well. The model trained on 750 synthetic images also improved significantly after fine-tuning with 750 real images for 41 epochs. We initialized fine-tuning to train the model's weights for 200 epochs, but the package fast-forwarded the fine-tuning process to the last 41 epochs. Based on these results, we recommend using YARDS to generate either very large synthetic datasets or smaller supplementary synthetic datasets to boost manually labeled images.

Binary Classification of Synthetic vs. Real Data

To test whether computer vision models could distinguish between real and synthetic data, we trained LeNet-5 (Lecun et al. 1998), AlexNet (Krizhevsky, Sutskever, and Hinton 2012), and ResNet-50 (He et al. 2015)—three classic CNN architectures of increasing complexity—to classify real versus synthetic images. Each architecture was modified to take in inputs of $256 \times 256 \times 3$, configured with binary cross-entropy loss and the Adam optimizer, and trained for 50 epochs with batch-size 64. LeNet-5 was modified to use max-pooling and ReLU activation. We trained and tested the classifiers on 4000-image datasets composed of real and synthetic training images from *Super Mario Bros.*

Table 3: Accuracy of binary classifiers trained to classify synthetic versus real *Super Mario Bros.* gameplay images

Model	Accuracy
LeNet-5	0.5000
AlexNet	0.8490
ResNet-50	0.9690

Table 4: Mean average precision of datasets containing mixture of generated data from *Super Mario Bros.*, *Super Street Fighter II*, and *Earthbound*

Dataset	mAP@0.5
60k imgs w/ clipping	0.9525
60k imgs w/o clipping	0.9635

Contrary to our original hypothesis that each model would achieve roughly a 50% accuracy, Table 3 suggests that architectures with many weights (e.g. AlexNet and ResNet-50) are able to distinguish between real and synthetic images, while smaller ones like LeNet-5 are not. We therefore need to develop a training strategy to account for the discrepancy between synthetic and real data; in the future, we may be able to leverage these binary classifiers to guide further improvements to our synthetic data generation approach.

Generalization

We trained two very large datasets composed of synthetic data for three separate games: *Super Mario World*, *Super Street Fighter II*, and *Earthbound* (screenshot dimension: 256×224). We generated 20,000 images for each game, combining them into a total dataset of 60,000 images. We split the dataset at a 0.8 train-test ratio and generated two variants: one with clipping and one without. The results can be seen in Table 4. Model performance dropped for games outside of those the model was trained on, which may be due to the lack of sprites representing the larger sphere of

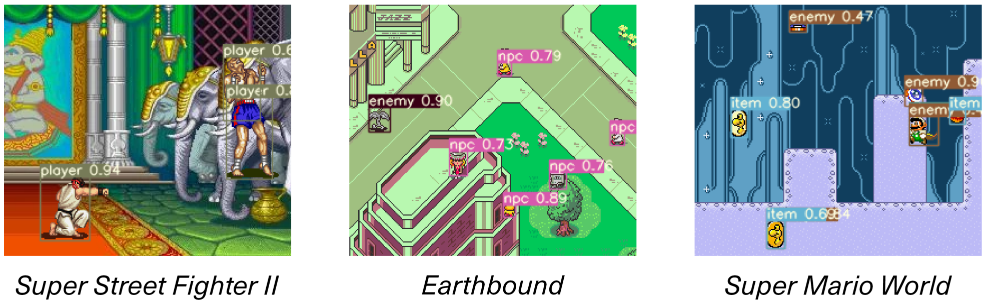

Figure 7: Detection Results of Models Trained on Large Dataset with Clipping for *Super Mario Bros.*, *Super Street Fighter II*, and *Earthbound*

NES/SNES games. Given a wider variety of sprites and games, however, we believe in the possibility of training a general detection model for most NES/SNES games.

Discussion

Although models trained on purely synthetic datasets do not perform as well as those trained on purely real datasets, our results suggest that training models with mixed synthetic-and-real datasets can increase overall sprite detection performance. Furthermore, synthesizing artificial data is much more efficient than collecting real gameplay screenshots and accelerates object-detection model development, ultimately allowing for better-performing models. Additionally, large synthetic datasets may outweigh the advantages of using real images and make sprite detection tools more accessible.

Training videogame object detectors on mixed datasets can be useful for many applications. For example, it may accelerate research in automated game design learning and help relax the requirement of deep visibility into the inner workings of emulated game hardware for distinguishing game sprites from the level geometry. We also envision methods for game developers to improve the accessibility of their games by verifying that a trained model recognizes sprites and their labels in a way which is consistent with a designer's intention (e.g. that enemies "read" as enemies, that a character is not easy to misinterpret as background texture, etc.). Such a model could help predict whether future players would be able to make the same assumptions and easily identify the playable parts of the game.

Synthetic data generation could also be useful as a feature extraction tool for reinforcement learning or other general game-playing agents. By training models that can accurately identify features of sprites belonging to classes like *helpful*, *harmful*, *item*, *enemy*, etc. (perhaps borrowed from an affordance grammar like that of Bentley and Osborn 2019), sprite detection models may help reinforcement learning agents train faster and generalize more effectively.

That being said, there are numerous ways to improve our approach. First, we suggest using model visualization techniques (e.g. class activation maps, occlusion sensitivity, gradient ascent) to visualize what the models see when trained with synthetic versus real data. Second, an addition that could greatly improve our current approach would be to define spatial curves in addition to sprite frequency spaces. Using the spatial curves to paste sprites into regions where they would appear in real gameplay images can serve as a way of increasing the realism of the synthetic images. Third, trying out existing approaches, such as domain randomization (Liu, Liu, and Luo 2020; Borrego et al. 2018; Tremblay et al. 2018), increasing the accuracy of images in relation to natural data (Liu, Liu, and Luo 2020), using generative models or GANS (Goodfellow et al. 2014; Liu, Liu, and Luo 2020; Bailo, Ham, and Shin 2019; Triastcyn and Faltings 2018), and procedural content generation (Nikolenko 2019), may provide further insight into how to refine our approach.

Our YARDS implementation can also benefit from additional features. First, incorporating multiprocessing would greatly increase the speed of synthetic data generation. Our current package generates 80 images per second on a single core with no GPU acceleration for *Super Mario Bros.*, and parallelizing this task would increase the package's efficiency. Second, adding basic image rendering and filtering functions such as blurring or pixelating sprites may be useful for videogames that do not use the pixelated style and resolution common to the four games we examined in this work. Third, color filtering functions may help the object detection models learn the sprites' essential features and avoid overfitting to their color patterns. Fourth, we would like to see added support for games in a wider variety of gameplay styles and genres. Finally, adding text detection functions may help with including basic user-interface elements.

In summary, this paper has introduced an application of existing synthetic data generation research to the problem of sprite detection and a software package that enables an end-user to rapidly generate large, synthetic training images based on sprite frequency spaces and edge-handling. An open-source and working prototype of YARDS is available at https://github.com/faimSD/yards. We hope that our paper and software package will inspire further research in sprite detection and in computer vision and games.

References

Bailo, O.; Ham, D.; and Shin, Y. M. 2019. Red blood cell image generation for data augmentation using conditional generative adversarial networks.

Beery, S.; Liu, Y.; Morris, D.; Piavis, J.; Kapoor, A.; Meister, M.; Joshi, N.; and Perona, P. 2019. Synthetic examples improve generalization for rare classes.

Bentley, G. R., and Osborn, J. C. 2019. The videogame affordances corpus. In *2019 Experimental AI in Games Workshop*.

Bochkovskiy, A.; Wang, C.-Y.; and Liao, H.-Y. M. 2020. Yolov4: Optimal speed and accuracy of object detection.

Borrego, J.; Dehban, A.; Figueiredo, R.; Moreno, P.; Bernardino, A.; and Santos-Victor, J. 2018. Applying domain randomization to synthetic data for object category detection.

Di Cicco, M.; Potena, C.; Grisetti, G.; and Pretto, A. 2017. Automatic model based dataset generation for fast and accurate crop and weeds detection. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Dwibedi, D.; Misra, I.; and Hebert, M. 2017. Cut, paste and learn: Surprisingly easy synthesis for instance detection.

Georgakis, G.; Mousavian, A.; Berg, A. C.; and Kosecka, J. 2017. Synthesizing training data for object detection in indoor scenes.

Goodfellow, I. J.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; and Bengio, Y. 2014. Generative adversarial networks.

Guzdial, M., and Riedl, M. 2016. Toward game level generation from gameplay videos.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Deep residual learning for image recognition. *CoRR* abs/1512.03385.

Hinterstoisser, S.; Pauly, O.; Heibel, H.; Marek, M.; and Bokeloh, M. 2019. An annotation saved is an annotation earned: Using fully synthetic training for object instance detection.

Koylu, C.; Zhao, C.; and Shao, W. 2019. Deep neural networks and kernel density estimation for detecting human activity patterns from geo-tagged images: A case study of birdwatching on flickr. *ISPRS International Journal of Geo-Information* 8(1).

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In Pereira, F.; Burges, C. J. C.; Bottou, L.; and Weinberger, K. Q., eds., *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc. 1097–1105.

Lateh, M. A.; Muda, A. K.; Yusof, Z. I. M.; Muda, N. A.; and Azmi, M. S. 2017. Handling a small dataset problem in prediction model by employ artificial data generation approach: A review. *Journal of Physics: Conference Series* 892:012016.

Lecun, Y.; Bottou, L.; Bengio, Y.; and Haffner, P. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 2278–2324.

Liu, W.; Liu, J.; and Luo, B. 2020. Can synthetic data improve object detection results for remote sensing images?

Luo, Z.; Guzdial, M.; Liao, N.; and Riedl, M. 2018. Player experience extraction from gameplay video. *CoRR* abs/1809.06201.

Nikolenko, S. I. 2019. Synthetic data for deep learning.

Nowruzi, F. E.; Kapoor, P.; Kolhatkar, D.; Hassanat, F. A.; Laganiere, R.; and Rebut, J. 2019. How much real data do we actually need: Analyzing object detection performance using synthetic and real data.

Osborn, J. C.; Summerville, A.; and Mateas, M. 2017. Automated game design learning.

Rajpura, P. S.; Bojinov, H.; and Hegde, R. S. 2017. Object detection using deep cnns trained on synthetic images.

Redmon, J., and Farhadi, A. 2016. Yolo9000: Better, faster, stronger.

Redmon, J., and Farhadi, A. 2018. Yolov3: An incremental improvement.

Redmon, J.; Divvala, S.; Girshick, R.; and Farhadi, A. 2015. You only look once: Unified, real-time object detection.

Roig, C.; Varas, D.; Masuda, I.; Riveiro, J. C.; and Bou-Balust, E. 2020. Unsupervised multi-label dataset generation from web data.

Rozantsev, A.; Lepetit, V.; and Fua, P. 2015. On rendering synthetic images for training an object detector. *Computer Vision and Image Understanding* 137:24–37.

Seib, V.; Lange, B.; and Wirtz, S. 2020. Mixing real and synthetic data to enhance neural network training – a review of current approaches.

Shaked, S., and Rokach, L. 2020. Privgen: Preserving privacy of sequences through data generation.

Summerville, A.; Guzdial, M.; Mateas, M.; and Riedl, M. 2016a. Learning player tailored content from observation: Platformer level generation from video traces using lstms. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Summerville, A. J.; Snodgrass, S.; Mateas, M.; and Ontanón, S. 2016b. The vglc: The video game level corpus. *arXiv preprint arXiv:1606.07487*.

Summerville, A.; Osborn, J.; and Mateas, M. 2017. Charda: Causal hybrid automata recovery via dynamic analysis. *arXiv preprint arXiv:1707.03336*.

Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; and Rabinovich, A. 2014. Going deeper with convolutions.

Tremblay, J.; Prakash, A.; Acuna, D.; Brophy, M.; Jampani, V.; Anil, C.; To, T.; Cameracci, E.; Boochoon, S.; and Birchfield, S. 2018. Training deep networks with synthetic data: Bridging the reality gap by domain randomization.

Triastcyn, A., and Faltings, B. 2018. Generating artificial data for private deep learning.

Ultralytics. 2020. Yolov5.

Wong, M. Z.; Kunii, K.; Baylis, M.; Ong, W. H.; Kroupa, P.; and Koller, S. 2019. Synthetic dataset generation for object-to-model deep learning in industrial applications.

Yun, K.; Nguyen, L.; Nguyen, T.; Kim, D.; Eldin, S.; Huyen, A.; Lu, T.; and Chow, E. 2019a. Small target detection for search and rescue operations using distributed deep learning and synthetic data generation.

Yun, W.; Lee, J.; Kim, J.; and Kim, J. 2019b. Balancing domain gap for object instance detection.

Zhang, X.; Zhan, Z.; Holtz, M.; and Smith, A. M. 2018. Crawling, indexing, and retrieving moments in videogames. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, FDG '18. New York, NY, USA: Association for Computing Machinery.