ScreenShield: Computer Monitor Tracking and Blurring for Video

Niall Kehoe
Stanford University
nkehoe@stanford.edu

Aniket Mahajan
Stanford University
aniketm@stanford.edu

Abstract

ScreenShield is a pipeline for detecting, segmentation, tracking, and inpainting computer monitor screens in video, balancing both speed and accuracy for latency-sensitive scenarios such as live stream processing. It uses a finetuned YOLOv11 detector capable of 20 ms/frame, a YOLOv11 segmentation model running at 0.15 sec/frame, and Cutie to track previously identified masks between frames. We inpaint a blur of the content below the mask and optionally allow users to customize cosmetic preferences, like border and/or a logo.

Our finetuned pipeline observes an object detection F1 score of 0.87 and image segmentation F1 score of 0.78, all the while reducing the runtime from 1.5 sec/frame on zero-shot baselines to 0.15 sec/frame. By combining optimizations like fast detection, conditional segmentation, and mask propagation, ScreenShield demonstrates potential for real time screen blurring. This is particularly useful for video conferences, corporate livestreams, and matters of national security.

This low latency pipeline could be used to make on-device video classification possible, to eliminate possible leaks at their source, with no need for external storage/transfer which present extra security challenges and increases the number points of failure.

1. Introduction

Companies/agencies in today's environment try to appear more authentic in order to appeal to modern audiences by producing live streams and short-form relaxed content. While this format is growing in popularity, they pose a threat to the businesses they seek to promote by risking private information leaking from the screens of their employees. Traditionally, these regions are blurred manually by professional video editors in post. However, due to the legacy methods for preventing this are becoming impractical for more frequent releases, a new technology is needed to edit video footage in real time, protecting company secrets.

Though there have been many breakthroughs in object detection (YOLO), video segmentation (Meta's Segment Anything), and inpainting, there is no single system connecting these techniques together while still prioritizing latency. We aim to present a unified framework for object tracking and segmentation in video–in particular, tracking objects in video and inpainting on computer monitors.

This could also be applied to the realm of national security. When the press covers sensitive areas (e.g. SCIFFs), cameras are seized and held until the footage is digitally altered to remove security information. Our software would be able to automatically do these removals, further enhancing security and efficiency.

2. Literature Review

2.1. Detection

The You Only Look Once (YOLO) models, originally proposed in 2016 by Redmon et al [17], are a series of open-source models targeting the specific task of image detection. This CNN based architecture poses object detection as a regression problem to bounding boxes and class probabilities. Unlike prior R-CNN variants which requires multiple network stages, YOLO performs classification with just one pass.

Since then, the YOLO model has undergone many iterations, improving speed and accuracy. Recently, YOLOv11 added a spatial attention module that allows for the model to better focus on important regions in the image and improve accuracy on smaller and occluded objects [13]. It also explores optimizations in the cross stage partial (CSP) blocks–a technique used to improve the information encoded in a feature map. This is done by splitting the feature map into two paths along channels: one that undergoes transformation and another that bypasses it, reducing redundant computations and improving training efficiency. YOLOv11 improves computation efficiency further by using two smaller convolutions instead of a single large one.

2.2. Segmentation

Long et al's Fully Connected Networks for Image Segmentation paper proposed swapping out the fully connected layers with convolutions in a network, demonstrating that FCNs can be trained and output a one-to-one pixel label [15]. Additionally, this new architecture made it possible to handle arbitrary sized input dimensions. They also introduced a skip architecture that allows for merging long range context with short term detail.

Since then, Facebook Research built off these concepts and released DINO [11] and the Segment Anything Model (SAM) [14]. Caron et al. observed with their DINO model that a vision transformer trained without any image labels naturally learns attention maps that can identify the presence of an object. Paired with threshold cutoffs, they leveraged this property to perform segmentation tasks. SAM wraps this idea into a promptable interface–given point, box, or mask prompts, its transformer backbone and mask decoder output a segmentation. Grounded-SAM extends SAM by adding a grounding head that accepts text phrases around which, model generates masks [2].

2.3. Video Tracking

One-shot video object segmentation (OBVOS) proposed by Caelles et al explores the separation of an object from the background of a video given a mask of the object on the first frame [10]. Using a fully convolutional neural network architecture pretrained on ImageNet and finetuned on a video segmentation dataset, OBVOS learns foreground vs. background discrimination for a per pixel output and predicts per frame.

Space time memory networks then added global memory banks [16], appending recognized features from subsequent frames into a global cache. This way, the model grows resilient in recognizing images in different permutations or illusions than the first frame. Cutie builds off that by establishing a "permanent" slot for objects, reducing complexity for persistent features and avoiding object drift over long video segments [12].

3. Datasets

We used datasets with bounding boxes and segmented images for image detection and segmentation respectively. We applied consistent preprocessing across all datasets: resizing images to a fixed resolution (640x640), image augmentation (flipping, color variants, etc.), and normalizing pixel values.

3.1. Image Detection

- **RoboFlow Computer Monitor Dataset**: 1,312 computer monitor images with bounding box annotations using a 69:19:12 training, validation, and test split [1].
- **RoboFlow TV, PC, Monitors Dataset**: 1,002 images spanning various screen types, though slightly outdated tech, using a 68:20:12 training, validation, and test split [5].
- **RoboFlow Office Monitor Dataset**: 2,003 desktop monitor images using a 72:19:9 training, validation, and test split [6].
- **RoboFlow Screen Dataset**: 350 neatly annotated images of modern laptops and desktop monitors using a 72:19:9 training, validation, and test split [7].

3.2. Image Segmentation

- **RoboFlow Screen Segmentation**: 350 images with mask annotations of primarily desk setups. We use a 88:8:4 training, validation, and test split [8].
- **Our Roboflow Dataset**: 534 images spanning YouTube videos and high-quality data from the image detection datasets. Hand annotated by us for screens. The dataset is split 70:20:10 [9].
- **RoboFlow Laptop Screen Detection**: 47 images focused on laptop screens in a range of real-world environments. Split used is 87:8:5 [3].
- **RoboFlow Laptop Screen Detection (Vivek version)**: 58 images of neatly annotated laptops. Split is 88:8:4 [4].

4. Method

To obscure key regions, we need to identify, refine, and apply a transformation to each frame of video. No existing out of the box methods existed for our use case so we created a custom pipeline divided into 4 steps: **detection**, **segmentation**, **tracking**, and **inpainting**. The **detection** and **segmentation** tasks could be done with existing open-vocabulary object models. However, both suffer from major latency issues which make them impractical for our use-case.

4.1. Image Detection

This step is key to identifying the first frame in which a monitor is present. We use a traditional, bounding-box detection model to detect the number of instances and locations for screens. Once any bounding box reaches our confidence threshold, we transition to the segmentation stage.

4.2. Image Segmentation

Given a frame of interest as input, we feed it into a segmentation model to create a mask for all the instances of screens within the image. Once complete, it begins the Image Tracking stage.

4.3. Image Tracking

The VOS model takes the last segmented frame and tracks the previous instances through to the following frame, adjusting the new mask which it uses as input when tracking the next frame transition. If the number of instances of screens (from 4.1), we restart stage 2 on that frame and re-run segmentation on all occurrences (this is done to include new monitors now in the field of view in the segmentation mask initialization). Additionally, if the existing items being tracked can no longer be tracked, we terminate the Cutie session to reduce overhead.

4.4. Inpainting

Each of the segmentation masks identified on a given frame represent the regions of a screen and consequently are inpainted using a Gaussian blur effect on the bounded pixels. This involves smoothing the image by taking the average RGB values of its neighbors following a normal distribution.

In addition, if the user wants to add a logo / border outline of the screens, they can set a CLI flag to do so. The logo is then mounted into mask by computing the centroid of the mask and scaling it to fit inside the segmentation mask.

5. Experiments

5.1. Evaluation Metrics

To evaluate our object detection and segmentation models, we utilize the following metrics.

- **Precision**: The fraction of all predicted bounding boxes that actually contain a computer monitor screen.
- **Recall**: The fraction of computer monitors in a scene that were successfully detected.
- **F1 Score**: The harmonic mean of precision and recall, providing a single measure that balances both:
- **AP@0.5** (Average Precision at IoU threshold 0.5): The average precision computed when a predicted bounding box is considered correct if its Intersection over Union (IoU) with the ground truth is at least 0.5.
- **AP@0.5:0.95**: The average precision computed over multiple IoU thresholds from 0.5 to 0.95 with a step size of 0.05, offering a more comprehensive evaluation of detector performance.
- **Jaccard Similarity** (also known as *mean Intersection over Union*) to measure how accurately we can identify the mask for a monitor screen.

$$\mathrm{J}(M, G) = \frac{|M \cap G|}{|M \cup G|} \tag{1}$$

where M is the pixels of the segmentation mask and G is the ground-truth pixels of the object.

5.2. Image Detection

As a baseline for detection, we used GroundingDINO with a zero-shot prompt of "screen" to predict the bounding boxes of the screens. While this was able to draw reasonable bounding boxes around monitors, the major drawback of this approach is latency. On a single T-4 chip, a single frame takes 0.3 secs per inference.

To solve this issue, we finetuned a YOLOv11 model for the task. This sped up inference to take only 0.02 seconds per frame—a **15x improvement**. In addition, this model outperformed zero-shot DINO in every metric by a large margin, indicating that despite advanced attention based architectures showing promise in literature, when it comes to domain specific tasks, finetuning on a convolutional model yields better performance.

The full performance metrics of the two models are reported in Table 1.

5.3. Object Segmentation

As a baseline for segmentation, we used DINO-SAM – a combination of both Grounding DINO and Segment Anything. Again, similar to its performance in object detection, this worked reasonably, but involved expensive inference—we observed an inference of 1.2 seconds per frame on a single T4 chip.

To solve the latency issue here, we finetuned a YOLOv11-segmentation model to generate a precise mask for the screens. This sped up inference significantly, observing an inference of 0.15 seconds per frame—an **8x improvement**.

Unlike image detection, YOLOv11's segmentation performance shows mixed results. The finetuned YOLO model achieves higher AP scores for both 0.5 and 0.95, indicating that its confidence scores are more reliable across different thresholds. However, DINO-SAM still generates tighter masks when evaluated at a fixed threshold, as seen with the higher precision, recall, and mean IoU scores.

The full performance metrics of the two models are reported in Table 2.

5.4. Object tracking

Using a Cutie session, we were able to take steps between frames without having to recompute the mask in each frame.

We also experimented with using frame-by-frame segmentation and found higher latency as expected with negligible performance benefits. The increase in time for inference on these images is extremely important for our desired use case.

Table 1: Performance of object detection models

Model	Dataset	#Images	#Instances	Precision	Recall	F1 Score	AP@0.5	AP@0.5:0.95	Mean IoU
GroundingDINO (zero-shot)	Validation	903	1472	0.7750	0.8424	0.8073	0.7750	0.5677	0.8319
YOLOv11 (finetuned)				**0.8418**	**0.9110**	**0.8750**	**0.8418**	**0.7099**	**0.8963**
GroundingDINO (zero-shot)	Testing	401	648	0.7709	0.8410	0.8044	0.7709	0.5778	0.8453
YOLOv11 (finetuned)				**0.8333**	**0.9028**	**0.8667**	**0.8333**	**0.7083**	**0.9009**

Table 2: Performance of image segmentation models

Model	Dataset	#Images	#Instances	Precision	Recall	F1 Score	AP@0.5	AP@0.5:0.95	Mean IoU
DINO-SAM (zero-shot)	Validation	262	391	**0.8680**	**0.8211**	**0.8198**	0.7721	0.5850	**0.7541**
YOLO (finetuned)				0.7301	0.7915	0.7395	**0.7774**	**0.5995**	0.6866
DINO-SAM (zero-shot)	Testing	134	203	**0.8236**	0.8002	**0.7903**	0.7640	0.4914	0.7091
YOLO (finetuned)				0.7572	**0.8531**	0.7781	**0.7732**	**0.6087**	**0.7220**

Additionally, due to hardware constraints we had to reduce the Cutie's max_internal_size from 480 to 240, slightly degrading the accuracy of cutie tracking.

5.5. Pipeline

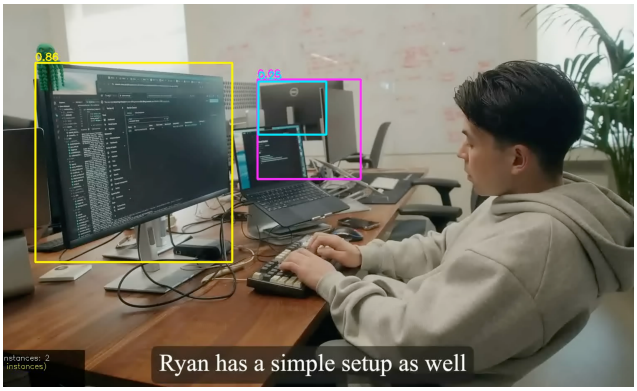


Figure 1: YOLOv11 bounding-box detection

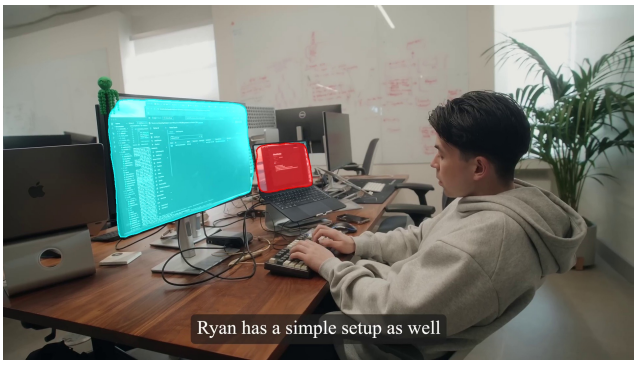
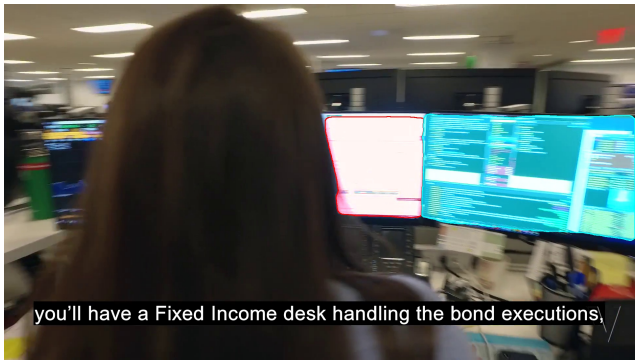


Figure 2: YOLOv11 segmentation mask

Above we show qualitatively the outputs of our pipeline through the steps of object detection, image segmentation, and inpainting.

(a) Stationary camera tracking



(b) Moving camera tracking

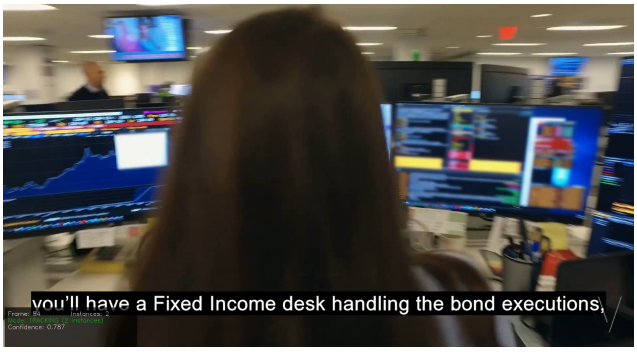
Figure 3: YOLOv11 segmentation after mask propagation

First, we run fine-tuned YOLOv11 detector on each in-

put frame to create monitor bounding boxes. Figure 1 displays the multiple identified bounding boxes identified in a single frame.

Next, once the detector finds at least one monitor, we pass this into our YOLOv11 segmentation model. In 2 we see the initial segmentation mask produced on the first frame with a detected monitor and in 3 the mask for a later frame via Cutie. Image (a) demonstrates Cutie's tracking ability on a stationary camera, while image (b) shows off Cutie's tracking on a moving camera. This means not requiring any re-segmentation to create the second frame which saves computation.

Finally, we inpaint each detected screen region with a Gaussian blur under the mask. 4 (a) shows output when just blurring the masks. 5 (a) demonstrates user-supplied logo overlaid at the mask center and 5 (b) white border outline.

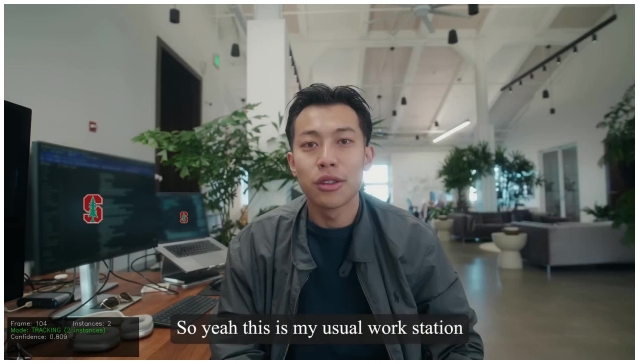


(a) Gaussian-blur inpainting only.

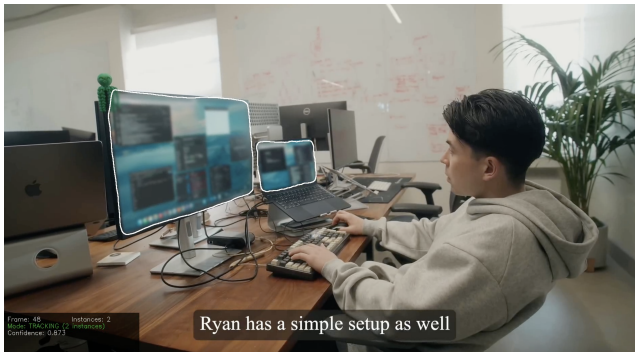
Figure 4: Inpainting stages: (a) blurred screen region

We integrate all these stages continuously, creating a low-latency pipeline. A demo video output of the pipeline can be viewed at:

- Original input clip: https://youtu.be/QP8muGb9pRc
- Pipeline output with blurred inpainting: https://youtu.be/-lSK1XjPA-g
- Pipeline output with logo and blurred inpainting: https://youtu.be/nOqEOoKYxv0



(a) Blur + logo overlay.



(b) Blur + border outline.

Figure 5: Inpainting stages: (a) logo addition, (b) border outline

6. Conclusions

Overall, the pipeline works as intended with high accuracy and low latency. YOLO detection models outperform zero-shot prompt models in accuracy and speed. YOLO segmentation finetuned models are slightly less accurate than their open-vocabulary peers. This is likely due to the shortage of high-quality training data for the segmentation task. Cutie performed very well at stepping the segmentation maps between frames, even with many instances of screens being tracked simultaneously.

Future areas of work could include:

- **Mask Flickering**: Occasionally, there is 1 frame in which the mask disappears, leaving the screen visible before resuming tracking. This could be solved by adding a post-processing step to mask a frame where there are masks immediately before and after it sequentially.
- **Device Types**: refining the detection and segmentation models to track specific types of screens, for example recognizing phones/tablets as distinct categories to exempt from obfuscation.

- **On Device Application**: Applying this pipeline between the capture and save to storage steps of a vision-enabled device. This would increase security in high-risk scenarios, as the only copy of a video / photo taken has had the sensitive material removed. With newer phone models shipped with in-built GPUs (e.g. iPhone 16), this could possibly be built in to employer-issued work phones to prevent the photographing and leaking of sensitive / intellectual property materials.
- **Logo Realism**: Re-orienting and blending the logo so that it matches the orientation of the screen and the lighting of the original scene. Currently, we are just dropping the logo on top of the blurred inpainting within the segmentation mask, but thoughtfully integrating the logo makes the final result less distracting and adds a polished, professional appearance.

7. Acknowledgments

7.1. Contributions

- **Niall Kehoe**:
  - Pre-processing and collation of the object detection and segementation datasets.
  - Training of our own image detection and segmentation models.
  - Evaluation of our finetuned YOLOv11 detection and segmentation models.
  - Implementation of pipeline from detection to segmentation and tracking.
- **Aniket Mahajan**:
  - Data set collection and hand annotating our custom segmentation dataset.
  - Evaluation of GroundingDINO and GroundingDINO-SAM.
  - Implementation of inpainting

Both of us contributed to completing the final writeup.

7.2. Acknowledgements

We thank the open-source contributors that built out the resources and models that we utilize throughout the project. In particular,

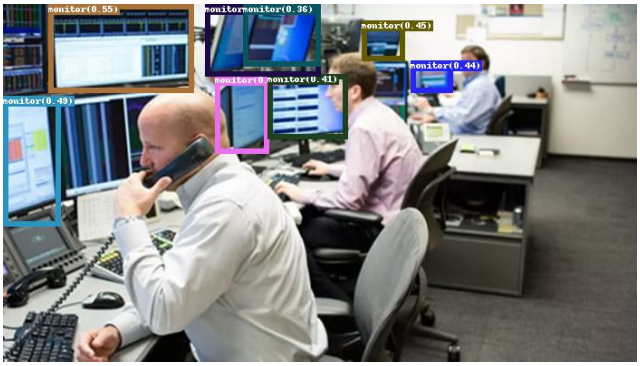
- Grounding DINO contributors at https://github.com/IDEA-Research/GroundingDINO
- YOLO contributors at https://github.com/ultralytics/ultralytics
- DINO SAM contributors at https://github.com/IDEA-Research/Grounded-Segment-Anything
- Cutie contributors at https://github.com/hkchengrex/Cutie

We also thank those who curated and annotated the many datasets we used on the Roboflow platform.

In addition, we express gratitude to the YouTube channel Linq (https://www.youtube.com/@thelinqapp) and Justin Tse (https://www.youtube.com/@JustinTse) for their office-space footage used to build our custom dataset. We also thank Dose of Devy (https://www.youtube.com/@doseofdevy) and Will Wang (https://www.youtube.com/@WillWang) for providing the video clips featured in our final pipeline demonstrations.

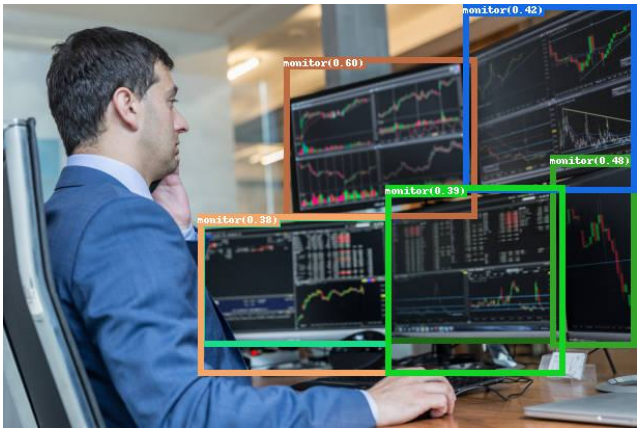
Lastly, we thank the CS231n course staff for their teaching this quarter and our project mentor Iris Xie for the support and advice along the way.

Appendix

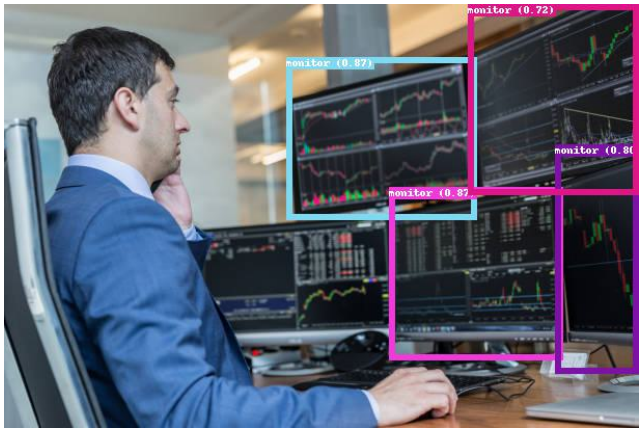


DINO Detections

YOLO Detections



DINO Detections



YOLO Detections

Figure 6: A side-by-side comparison of DINO and YOLO object detection performance on examples in which YOLO fails.

Our YOLO model struggles mainly with background monitors, more blurry and out of focus. This is permissible for our usecase as false negatives on already blurry images do not cause security concerns.

References

[1] Computer monitor dataset. universe.roboflow.com/n-j7ohx/computer-monitor-0cbhd.

[2] Grounded-segment-anything. https://github.com/IDEAResearch/GroundedSegmentAnything.

[3] Laptop screen detection dataset. https://universe.roboflow.com/laptop-screen-detection/laptop-screen-detection-lohtq.

[4] Laptop screen detection (vivek) dataset. https://universe.roboflow.com/vivek-kumar-kirw1/laptop-screen-detection.

[5] Monitors (tvs, pc, monitor, etc) dataset. universe.roboflow.com/energy-chaser/monitors-tvs-pc-monitors-etc.

[6] Office monitor dataset. universe.roboflow.com/4-52p2c/office-monitor-r7oge.

[7] Screen dataset. universe.roboflow.com/pavement-wwadi/screen-7i6h8.

[8] Screen segmentation dataset (screen-1). https://universe.roboflow.com/pavement-wwadi/screen-7i6h8.

[9] Screens segmentation dataset. https://universe.roboflow.com/myworkspace-mvnb3/screens-segmentation.

[10] S. Caelles, K.-K. Maninis, J. Pont-Tuset, L. Leal-Taixé, D. Cremers, and L. Van Gool. One-shot video object segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 221–230, 2017. arXiv:1611.05198.

[11] M. Caron, I. Misra, J. Mairal, P. Goyal, P. Bojanowski, and A. Joulin. Emerging properties in self-supervised vision transformers. https://ai.meta.com/blog/dino-v2-computer-vision-self-supervised-learning/, 2021.

[12] H. K. Cheng, S. W. Oh, B. Price, J.-Y. Lee, and A. Schwing. Putting the object back into video object segmentation. 2023.

[13] R. Khanam and M. Hussain. Yolov11: An overview of the key architectural enhancements.

[14] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollár, and R. Girshick. Segment anything. 2023.

[15] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. 2015.

[16] S. W. Oh, J.-Y. Lee, N. Xu, and S. J. Kim. Video object segmentation using space-time memory networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9226–9235, 2019. arXiv:1904.00607.

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