Free supervision from video games

Philipp Krähenbühl
UT Austin
philkr@cs.utexas.edu

Abstract

Deep networks are extremely hungry for data. They devour hundreds of thousands of labeled images to learn robust and semantically meaningful feature representations. Current networks are so data hungry that collecting labeled data has become as important as designing the networks themselves. Unfortunately, manual data collection is both expensive and time consuming. We present an alternative, and show how ground truth labels for many vision tasks are easily extracted from video games in real time as we play them. We interface the popular Microsoft® DirectX® rendering API, and inject specialized rendering code into the game as it is running. This code produces ground truth labels for instance segmentation, semantic labeling, depth estimation, optical flow, intrinsic image decomposition, and instance tracking. Instead of labeling images, a researcher now simply plays video games all day long. Our method is general and works on a wide range of video games. We collected a dataset of 220k training images, and 60k test images across 3 video games, and evaluate state of the art optical flow, depth estimation and intrinsic image decomposition algorithms. Our video game data is visually closer to real world images, than other synthetic dataset.

1. Introduction

Supervised deep learning is the engine of computer vision, and good datasets are its fuel. Researchers use them to develop novel architectures and models, evaluate their performance, and measure the general progress of the field. Traditionally, most datasets were collected by human annotators, mimicking some part of the human perception system. Examples include boundary detection [1], segmentation [1], semantic labeling [29, 42], depth estimation [42], and many more. However, human annotations have two clear drawbacks: They are expensive to collect, and they bias research towards fields in which labels are readily available. Consider recent datasets image captioning [29], visual reasoning [24], image [12] or scene recognition [45] datasets. Each spurred a proliferation of algorithms, models and training objectives in their respective field. In other fields, such as depth estimation, optical flow, or intrinsic image decomposition, human labels are much harder to obtain. This led researchers to either focus on problems with plentiful human annotations, or abandon human labels and supervised learning all together [13,19,35]. Neither of these solutions is truly satisfying.

In this paper, we propose a different solution: we collect more labels. We use video games to produce ground truth labels without the need for human supervision. We obtain the ground truth labels through a wrapper around the commonly used DirectX® 11 API. Our wrapper intercepts and modifies any rendering calls by the video game. We reverse engineered the DirectX HLSL binary shader format, which allows us to inject arbitrary code into the rendering pipeline. We force the game to render our ground truth labels in addition to the final image. Our wrapper labels the images in real time without slowing the game down, allowing a seamless gaming experience in the name of science.

Our method collects a virtually infinite stream of tracked instance segmentations, semantic labels, depth, optical flow and albedo, paired with the gamer's actions, and a partial internal game state. For examples, see Figure 1. Very few of these labels are easily obtained through human annotations, and thus complement existing datasets. In an effort to increase the visual diversity of the collected data, we use multiple different video games, simulating a wide range of human visual experiences: starting with the survivalist life of an early cave men in Far Cry Primal [43], to the jours of a medieval warrior in The Witcher 3 [8], and culminating in the challenges facing a modern American man in Grand Theft Auto V [39]. Each game's features different environments, weather conditions, and a full day night cycle.

We collected a dataset of 220k training images, and 60k test images across our three video games, and evaluate state of the art optical flow and depth estimation algorithms. The new dataset is more challenging that existing ones. Finally, we evaluate the visual similarity between our dataset and other real world datasets, and show that our dataset is closest to real world benchmarks, while being absolutely free to collect.

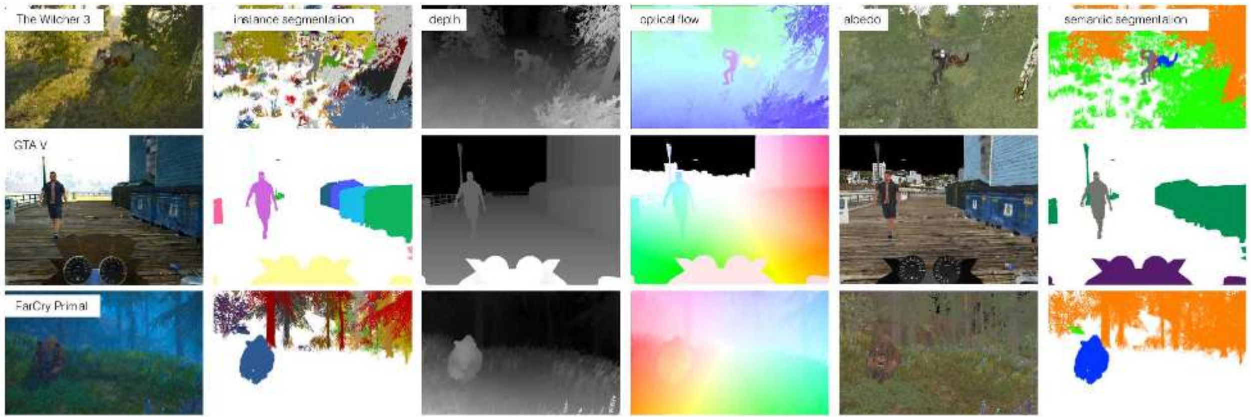


Figure 1: We extract free ground truth labels from DirectX video games as we play them. *Best viewed in color.*

2. Related work

Datasets drive computer vision. Early datasets, such as the Berkeley Segmentation Dataset and Benchmark (BSDS), brought machine learning into the field of computer vision [1]. The Pascal VOC benchmark fostered the development of object detection [15]. ImageNet lead to the current deep learning revolution [12] in image recognition, which morphed into a broader object centric challenge in Microsoft COCO [29]. The Places dataset pushed the state of the art in scene recognition [45]. This proliferation of datasets has been most dramatic in visual recognition, where human labels are relatively cheap to collect.

In low-level vision, the Middlebury dataset served as the gold standard to judge all optical flow, stereo and multi-view stereo algorithms [3]. It contains a few dozen images, and is too small to train effective deep models. The NYU RGBD dataset partially remedies this [42]. It provides paired color images, depth and semantic segmentation labels, but is limited to indoor environments. The KITTI benchmark [17] provides ground truth for visual odometry, stereo reconstruction, optical flow, scene flow, and object detection. It pushed the limits of human labeling, as optical and scene flow are obtained by manually fitting CAD models to moving objects [32]. Cityscapes [11] extends part of the KITTI benchmark to 50 European cities, with accurate depth and semantic labels available for each frame.

Our dataset extends the positive aspects of all the aforementioned datasets. It provides a diverse set of visual experiences and environmental conditions, spanning 3 video games and full day-night and weather cycles. It provides a large amount of paired labeled data, and is easily extended. In addition, it does not require any human labeling.

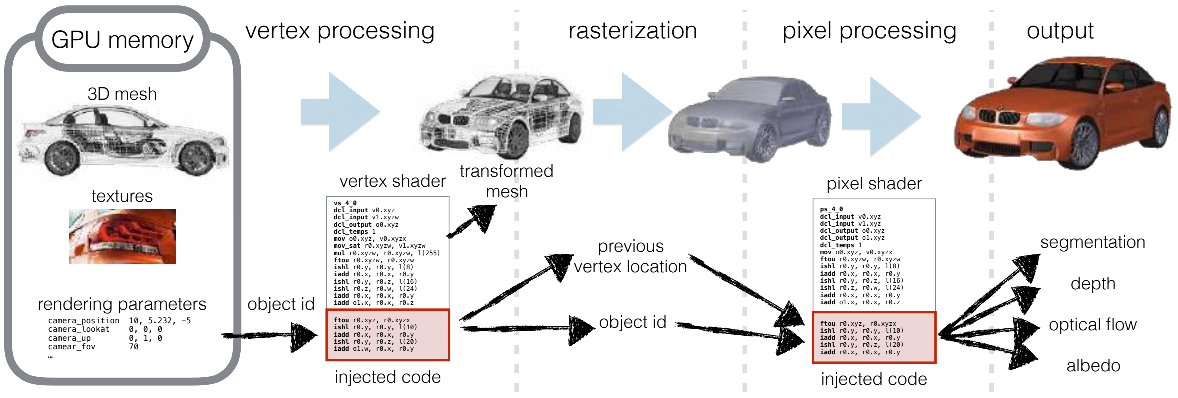
Synthetic data has existed for almost as long as computer vision itself. Horn and Schunck [22] evaluated their algorithm using synthetic two dimensional translation patterns. Barron *et al*. [4] and McCane *et al*. [31] extend this to include the three dimensional scene and rendered objects. Mayer *et al*. [30] include more complex scenes. Synthetic datasets are also used to benchmark depth estimation [21], instance segmentation [36], visual feature descriptors [26], or human-pose and hand tracking from depth images [40, 41]. However, these datasets still have a limited visual quality and diversity, as content is primarily hand crafted. In contrast, this paper extracts ground truth labels from modern video games created by professional artists, providing a richer and more diverse visual experience.

Buttler *et al*. [7] cleverly sidestepped the content creation issue, and built their dataset on an existing open-source movie. They modified the rendering engine of the movie and simplified scene elements. Due to the short length of the movie and the immense human effort involved, the resulting dataset is rather small.

We are not the first paper to use video games to extract ground truth labels for computer vision. Richter *et al*. [38] extract semantic segmentation labels from Grand Theft Auto V. They record all DirectX rendering commands, and group pixels according to similar texture, shader and geometry. They propagate human segmentation annotations between groups of pixels throughout all recorded frames. Concurrently to us, Richter *et al*. [37] improved their system to capture optical flow and instance segmentations. However, both of their algorithms rely on expensive post-processing to compute all modalities.

This work differs in three fundamental ways from prior synthetic datasets: Our system is real time, requires no human annotations, and generalizes across multiple games. We directly inject code into the rendering pipeline of a game to compute all labels in real time. This could enable policy learning algorithms to train on ground truth labels [6, 44], and open the possibility of real time style transfer [9]. Our system also requires no human annotations. We obtain free semantic labels by looking at the English language descriptions of resources in shader. Finally, we show that our approach works across multiple games with diverse visuals.

To the best of our knowledge we are the first paper to produce a large joint depth, optical flow, segmentation, tracking and intrinsic image decomposition dataset.



```
camera_position  10, 5.232, -5
camera_lookat    0, 0, 0
camera_up        0, 1, 0
camear_fov       70
…
```

```
vs_4_0
dcl_input v0.xyz
dcl_input v1.xyzw
dcl_output o0.xyz
dcl_temps 1
mov o0.xyz, v0.xyzx
mov_sat r0.xyzw, v1.xyzw
mul r0.xyzw, r0.xyzw, l(255)
ftou r0.xyzw, r0.xyzw
ishl r0.y, r0.y, l(8)
iadd r0.x, r0.x, r0.y
ishl r0.y, r0.z, l(16)
ishl r0.z, r0.w, l(24)
iadd r0.x, r0.x, r0.y
iadd o1.x, r0.x, r0.z

ftou r0.xyz, r0.xyzx
ishl r0.y, r0.y, l(10)
iadd r0.x, r0.x, r0.y
ishl r0.y, r0.z, l(20)
iadd o1.w, r0.x, r0.y
```

```
ps_4_0
dcl_input v0.xyz
dcl_input v1.xyz
dcl_output o0.xyz
dcl_output o1.xyz
dcl_temps 1
mov o0.xyz, v0.xyzx
ftou r0.xyzw, r0.xyzw
ishl r0.y, r0.y, l(8)
iadd r0.x, r0.x, r0.y
ishl r0.y, r0.z, l(16)
ishl r0.z, r0.w, l(24)
iadd r0.x, r0.x, r0.y
iadd o1.x, r0.x, r0.z

ftou r0.xyz, r0.xyzx
ishl r0.y, r0.y, l(10)
iadd r0.x, r0.x, r0.y
ishl r0.y, r0.z, l(20)
iadd r0.x, r0.x, r0.y
```

Figure 2: Overview of the DirectX rendering pipeline: 3D geometry, textures and rendering parameters are stored in GPU memory. During rendering, they undergo at least three transformations. First the geometry is transformed into screen space, using a vertex shader. Then the geometry is rasterized into a 2d images. Finally, a pixel shader post-processes every pixel. In this paper, we wrap the DirectX library and reverse engineered the proprietary binary shader format to inject arbitrary code into the game's shaders. This code allows us to extract both low and high level vision ground truth from the game engine.

3. Preliminaries

Most modern video games use raster-based rendering on dedicated hardware (GPUs) to deliver stunning visuals. Figure 2 gives a high level overview of raster-based rendering. A game object consists of 3D meshes, textures and rendering parameters, all stored in buffers on the GPU memory inaccessible from the CPU. Geometry and texture buffers remain mostly fixed during the rendering process. The arrangement, deformation and appearance of objects is all controlled by the rendering parameters, stored in constant buffers (cbuffers). They contain camera parameters, object position, and even joint locations of animated objects.

Modern games render each object with several render calls, each of which draws part of an object with a constant texture and mostly continuous geometry. For example, a car comprises the wheels, tires and body, which each produce their own render call. In a render call the mesh undergoes several transformations: vertex processing, rasterization, and pixel processing.

In vertex processing projects the mesh into screen coordinates. First, optional animations deform the object. Next, the *world transformation* places the object in a common world reference frame. Finally, *view* and *projective transformation* project the object into the screen coordinate system, where it is then rasterized.

Rasterization takes the three dimensional description of an object, in the form of triangles, and converts them into pixels on the screen. For the purpose of this overview rasterization is fixed.

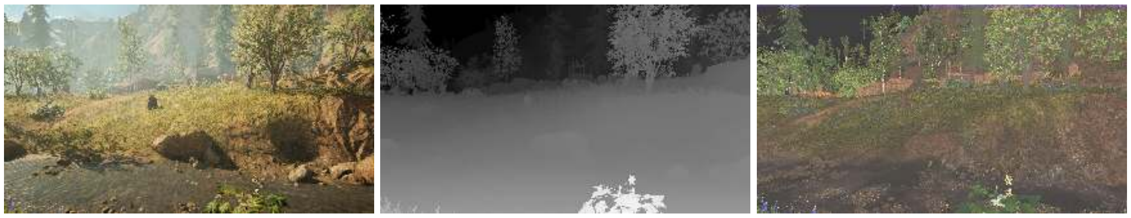
Finally, a pixel shader performs various post-processing operations, such as lighting computation, texturing, or shadow computations on the rasterized pixels. Vertex and pixel processing is fully programmable. A vertex or pixel shader is a small GPU program that computes these transformations.

The rendering parameters, together with vertex and pixel shaders control the entire rendering pipeline. Here, we modify these shaders, and intercept the rendering parameters to produce ground truth labels in real time, as the game is played. The next section goes into more details.

4. Supervision from video games

To access the DirectX pipeline for arbitrary video games, we wrote a wrapper around DirectX and the Direct X Graphics Infrastructure library (DXGI). We inject our wrapper into the video game through DLL injection using `dxgi.dll`. If placed into a game directory the operating system automatically loads our wrapper into the game's main memory. Once in main memory, we use function hooking to overwrite several key DirectX functions. Namely, we intercept shader creation, drawing functions, selection of render outputs, and any keyboard or mouse input. We further record all rendering parameters as they are uploaded to the GPU. We store them in CPU memory in a cbuffer cache. This allows us to access them more efficiently and at irregular access patterns.

Albedo and depth At this stage, the game already exposes a significant amount of information. For example, most games store both depth and albedo in some internal buffer. Unfortunately, each game stores these modalities slightly differently. While GTA V and The Witcher 3, store albedo as a RGB image, FarCry Primal stores it in a compressed two channel format, similar to a Bayer pattern. Similar differences apply to the depth buffer. We directly copy these buffers during the rendering process, and apply a game specific post processing step to transform them into a common format: A RGB image for albedo, and a disparity



(a) image (b) depth (c) albedo

Figure 3: For each frame in a game we directly capture: (a) the image before the GUI is drawn, (b) the depth map, and (c) the texture color as albedo. Depth is displayed in log-scale, and only converted to color for display. *Best viewed in color.*

image for depth, see Figure 3.

Other modalities such as segmentation or optical flow cannot be read out directly. To extract these modalities, we need to modify the actual rendering code, by injecting our own code into the game's shaders.

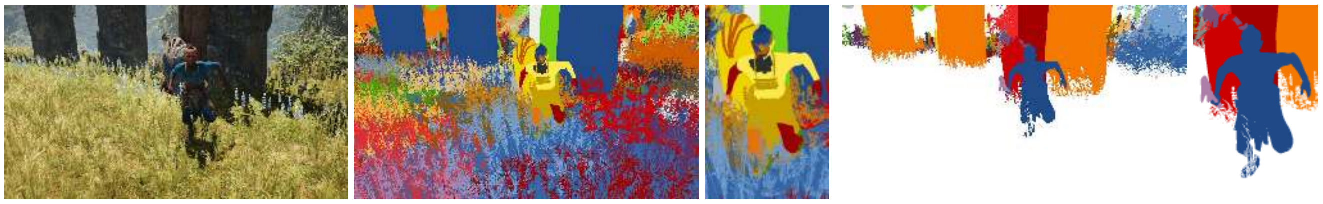
Shader injection DirectX shaders use a proprietary binary format, produced by the DirectX compiler. Games ship all shaders pre-compiled to hinder reverse engineering. However, significant parts of the DirectX shader format is reverse engineered for cross compilers to OpenGL, see [25] for a nice overview. We built on this work and wrote a tool to merge two different DirectX shader. It allows us to write arbitrary shader code, compile it, and execute it either before or after the game's shader code. To prevent this sort of injection, DirectX signs the bytecode using a hash function. If the hash does not match, the shader is not executed. Luckily, this hash function was recently reverse engineered by users on the GeForce forum [16]. It turns out to be an obfuscated implementation of MD5. Our shader injection gives us the ability to add arbitrary code into the game's shaders producing additional output without modifying or reverse engineering the core rendering code. We use shader injection to extract all remaining ground truth annotations from the game.

Instance segmentation Grouping pixels by render calls already yields a pixel perfect over-segmentation of the scene. However, each individual object requires many different render calls, see Figure 4b. Thus we need a way to merge different render calls within the same object. Here we use the fact that artists model individual objects in a 3D editor. For convenience, all individual parts of objects are kept in the same reference frame. We use this to group render calls into instances.

Specifically, we read the world transformation matrix of each render call from the cbuffer cache, extract position and orientation of the object and use a KD-tree to group objects in real time. We assign each object a unique id, and pass it as an additional parameter into a modified pixel shader. This modified pixel shader produces an additional instance segmentation map. See Figure 4c for the result.

Next, we show how these instances are tracked over time.

Tracking To track instances over several frames we first extract a signature for each render call. The signature contains the unique memory locations of the geometry, the hash functions of all shaders, and the textures used. We match each render call at time t to all render calls at $t-1$ using this signature. We greedily track objects using the Euclidean distance in both position and quaternion rotation, as well as the matching signature. We normalize the position for each game to adjust for differences in measurement units. If the distance exceeds a certain threshold, we set the object as untracked. We experimented with a more expensive weighted bipartite matching, but found greedy matching to be sufficient. The key to the success of greedy matching is to update the tracker at full frame rate, even though the



(a) image (b) render call id (c) instance segmentation

Figure 4: Multiple render calls (b) are grouped together to form an instance segmentation (c). Instances are segmented according to a common reference frame. The grass was purposefully not segmented. However each individual tree is segmented in the background. *Best viewed in color.*

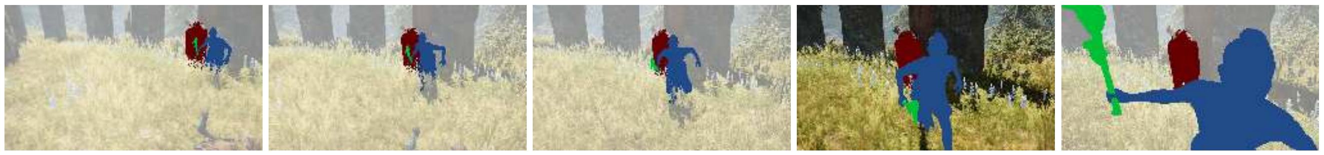


Figure 5: Tracking of the three largest objects in the scene. The weapon travels an average distance of 1.1 ± 0.6 between consecutive frames, the person travels 0.5 ± 0.1, while the object in the background stays stationary. All objects in the scene were tracked, but only the largest three are shown. *Best viewed on screen.*

game might not record every frame. For GTA V, we additionally use the games scripting engine to track cars and pedestrians, which is more efficient. Figure 5 shows the results of the tracking algorithm.

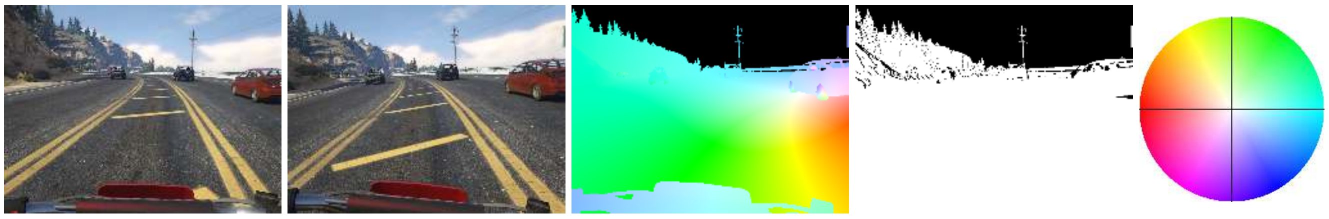
Optical flow Finally, we use the tracked instances to compute ground truth optical flow oriented backward in time. The optical flow $\vec{u}_t(\vec{x}) = \vec{x}_t - \vec{x}_{t-1}$ at pixel $\vec{x}_t$ measures the difference in pixel position of a point in the scene between the current and prior frame. Let $\vec{x}_t$ denote the current position and $\vec{x}_{t-1}$ the prior position. We now show how to extract this information directly from a vertex shader. Our key insight is that the deformation and projection of any point on a mesh solely depends on the rendering parameters. Let θ_{t-1} and θ_t be the rendering parameters at two consecutive frames. If we render a vertex $\vec{y}$ in frame t with its current rendering parameters θ_t, it is displayed at the current screen position $\vec{x}_t$, while using parameter θ_{t-1} will display it at the previous location $\vec{x}_{t-1}$. We use this to compute optical flow in real time. We duplicate each vertex shader to output two screen transformations for each object: One with θ_t, and one with θ_{t-1}. We rasterize $\vec{x}_t$ and compute the flow using $\vec{x}_{t-1}$. This optical flow estimate is exact and does not use any optimization, see Figure 6c for example. However, unlike traditional optical flow, our flow estimate is directed backward in time, from the current image to the previous one. It also does not yet deal with occlusion boundaries.

Occlusion boundaries To obtain occlusion boundaries, we first compute the scene flow, using the same procedure described above. Scene flow additionally computes the depth value d_{t-1} for each point $\vec{x}_{t-1}$. A point is occluded if there is another object in the target frame with a smaller depth value at location $\vec{x}_{t-1}$. Thus, a simple depth test is sufficient to compute occlusion boundaries. We add a small slack of $\varepsilon = 10cm$ to account for any numerical instability. Figure 6d shown an example.

Semantics Finally, the rendering engine also exposes some semantics to DirectX. Variables, buffers and texture are all named in English language in the shader bytecode. For example shaders that render trees consistently contain the word "tree" in their variable names, irrespective of the game. The same applies to animals, vegetation, or vehicles. We extract this semantic information by running a tokenizer over the variable names and finding rarely occurring tokens. However, not all tokens are semantic, some refer to shading computation or other internal rendering information.

We briefly considered using NLP techniques to extract the semantically meaningful tokens, but then quickly opted for a manual approach. We map these semantic tokens to seven different classes: Tree, Vegetation, Person, Weapon, Car, Object and Structure. The structure class contains any part of the game world that is not moving, or cannot be interacted with. Labeling took roughly 10 minutes per game.

This pipeline now allows us to extract a nearly infinite stream of tracked instance segmentations, semantic labels, depth, optical flow and albedo from a video game without any human annotations.



(a) frame $t-1$ (b) frame t (c) optical flow (d) occlusion boundaries (e) flow wheel

Figure 6: Visualization of optical flow (c) and occlusion boundaries (d) between two frames 100ms apart. The players vehicle is going at high speed, with displacements exceeding 300px. The sky is not tracked, hence there is no flow estimate. For reference we include the flow encoding in (e), colors are truncated at 100px displacement. *Best viewed in color.*

5. Evaluation

We collect our data from three different video games: FarCry Primal [43], GTA V [39] and The Witcher 3 [8]. We run out main evaluation on GTA V, and show preliminary results for FarCry Primal and The Witcher 3. Both FarCry and The Witcher heavily rely on instanced drawing calls, which we were not able to track at a satisfactory accuracy. We thus only report instance tracking and optical flow results for GTA V.

We collect a total of 220k training images, and 60k testing images. Training and testing images come from different play sessions, restart of game. For the Witcher 3 and FarCry primal we physically played the game, for GTA V we use the built in auto-pilot to navigate the environment. The auto-pilot starts at a random location in the world, drives or walks for 30 seconds ($\sim$ 180 frames), then gets reset. In total, we spent 2 hours physically playing The Witcher and FarCry, and collected 10 hours of GTA V auto-pilot data. The dataset contains $16-41$ instances per frame. In GTA, the average distance between consecutive training frames is $1m$ ($\sim \frac{1}{4}$ game car length). The average distance between a test frame and its closest training frame is $435m$ (median $540m$, 5th-percentile $3.9m$, 10th-percentile $26.8m$). Table 1 summarizes our dataset and compares it to MS COCO, the largest real world instance segmentation dataset. Note that our dataset contains two to five times more instances per image than COCO. At the same the labeling time was four orders of magnitude shorter, and a lot more fun.

Since Richter *et al*. [37] already show that modern video game images look realistic to the human eye, we will not repeat this experiment. We instead focus on how deep networks see our dataset. Specifically, we compute statistical similarities between our video game dataset and other datasets using a discriminatively trained classifier. We then provide baseline models for all tasks on our dataset. Finally, we provide some applications to robotics tasks, making full use of the real-time nature of our approach.

5.1. Statistical similarity to other datasets

Let $P_\mathcal{D}$ be the distribution of image patches X of size $W \times H$ in a dataset $\mathcal{D}$. We compare two distributions $P_{\mathcal{D}_1}$ and $P_{\mathcal{D}_2}$ using the KL-divergence. If two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$ have a low KL-divergence, they are similar. Computing the KL-divergence directly is hard, and involves estimating high dimensional probability densities. We instead compute the difference in KL-divergence from a query dataset $\mathcal{D}_3$, to two other datasets $\mathcal{D}_1$ and $\mathcal{D}_2$. The difference is negative if the query is close the first dataset, and positive otherwise. Mathematically the difference is defined as

$$\mathrm{KL}(P_{\mathcal{D}_3}|P_{\mathcal{D}_1}) - \mathrm{KL}(P_{\mathcal{D}_3}|P_{\mathcal{D}_2}) = \mathop{\mathrm{E}}_{X\sim P_{\mathcal{D}_3}}\left[\log\frac{P_{\mathcal{D}_2}(X)}{P_{\mathcal{D}_1}(X)}\right].$$

The resulting probability ratio is much easier to estimate.

We take inspiration from the Generative Adversarial Network literature [18], and train a discriminator D between the two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$. It should predict 0 for images in $\mathcal{D}_1$ and 1 for images in $\mathcal{D}_2$. By definition the optimal discriminator has the form

$$D(X) = \frac{P_{\mathcal{D}_2}(X)}{P_{\mathcal{D}_1}(X) + P_{\mathcal{D}_2}(X)}.$$

and the "logit" of the discriminator $\log \frac{D(X)}{1-D(X)} = \log \frac{P_{\mathcal{D}_2}(X)}{P_{\mathcal{D}_1}(X)}$ yields the probability ratio between the datasets. This allows us to efficiently approximate the difference in KL-divergence through the eyes of a convolutional network.

We train a Network in Network [28] model to distinguish between two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$ and evaluate it on a third $\mathcal{D}_3$. All patches are resized to 32×32, mean subtracted, and normalized. We mean subtract and normalize each dataset individually. We train a separate discriminator for each pair of datasets, and use ADAM [27] with a batch size of 32 and train for 10k iterations. The architecture and hyper parameters are the same for all pairs of datasets.

Figure 7 analyzes the visual similarity of datasets through the eyes of the network in network model for low to high-level patch sizes: 32px, 128px, and 512px. We compare our GTA V dataset with our other two datasets and 9 prior datasets: Citiscapes [11], Microsoft COCO [29], Flying Things [30], Freiburg driving sequence [30], Intrinsic Image in the Wild (IIW) [5], KITTI [17], NYU Depth v2 [42], and Sintel (final) [7]. To our surprise our synthetic dataset seems closer to most datasets than KITTI. We suspect this might have to do with saturation issues in KITTI images, which give them a distinct signature biasing the KL-divergence measure. On Cityscapes the situation is as expected: Natural image datasets are closer to each other than to our dataset, while most synthetic datasets are closer to GTA V. The more interesting experiment is the comparison of the GTA V dataset to other synthetic datasets. While Sintel seems close to real world datasets in low level patch statistics, GTA V clearly wins on medium and high level patches, capturing the overall appearance of the scene. The Freiburg dataset is farther in appearance

Dataset	#train	#test	time (h)	inst./img
GTA V	200k	50k	10	15.5
FarCry Primal	10k	5k	< 1	36.2
The Witcher 3	10k	5k	< 1	41.2
MS COCO	120k	40k	100+k	7.2

Table 1: A comparison of our game dataset to MS COCO in terms of size, instance density and annotation time.

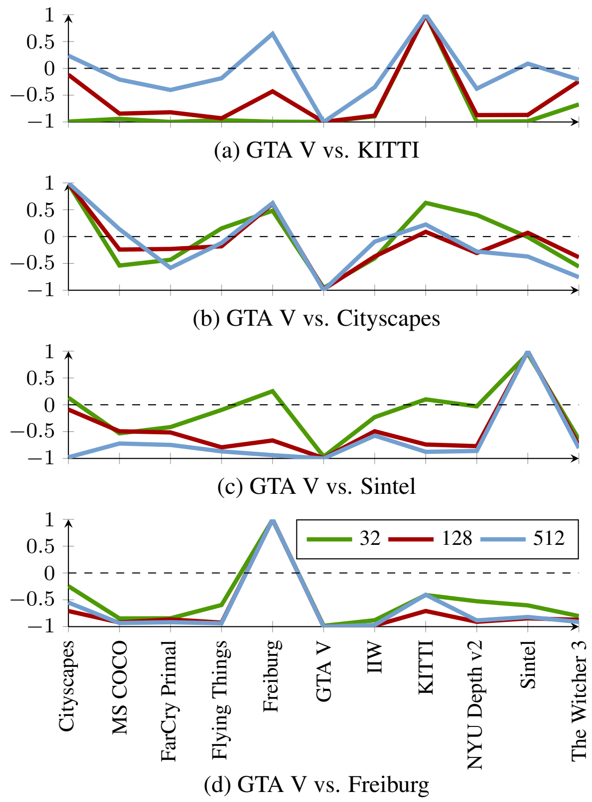


Figure 7: Comparison of our GTA V dataset to other datasets in terms of KL-divergence over patches of size 32×32, 128×128 and 512×512. Values below the dashed line indicate the dataset is closer to GTA V than the alternative.

than GTA V throughout all competing datasets. Note, that the Freiburg results is not symmetric, Freiburg is close to KITTI, but KITTI is not close to Freiburg. This comes from the fact that the KL-divergence is non-symmetric, and strongly favours visual diversity in its entropy term.

5.2. Baselines

We begin our evaluation with optical flow. We split the dataset into five parts according the the activity performed: walking, riding a bicycle, driving a car, riding a motorbike, and riding a quadbike. All vehicles scenes are very challenging. They feature high speed dirt bike races, passing cars on freeways, and relatively few slow moving scenes. The quad and car are easier, as these vehicles are large and cover part of the screen. Most optical flow algorithm do well on tracking the the players vehicle. Bikes and bicycles cover little of the screen, and feature quickly moving handles. Both walking and cycling scenes contain a significant amount of camera shaking and turns as the game simulates head motion. Table 2 measures the average flow magnitude for all valid pixels in a scene. The flow magnitude in GTA is significantly larger than KITTI, his is in part due to the larger images size (2x), and the faster travel speed and camera motion.

walk	bicycle	bike	quad	car	KITTI
51.3	96.0	87.7	60.0	70.0	27.7

Table 2: Average pixel displacement for different activities.

method	walk	bicl.	bike	quad	car	mean
Mean flow	3.25	6.04	4.15	5.77	3.11	4.46
FlowNet 2 (I)	83.11	36.02	38.65	50.52	35.99	48.86
FlowNet 2 (A)	**86.75**	**38.48**	**40.12**	**51.45**	**39.71**	**51.30**
FlowFields (I)	66.07	29.49	27.56	36.93	26.97	37.40
FlowFields (A)	76.65	34.47	32.96	41.26	36.94	44.46

Table 3: Flow accuracy in percent for a threshold of 5px, for four splits of our dataset, higher is better. I refers to flow computed on the original image, A the albedo image.

Table 3 compares state of the art optical flow algorithms on our dataset. We compare FlowNet 2 [23] and FlowFields [2], using the authors code and KITTI pre-trained models. We additionally evaluate the performance of the mean flow over the entire training set, as a constant baseline. This constant flow baseline performs poorly and serves as a lower bound. Analogous to KITTI [17], we evaluate the flow accuracy at a threshold of 5px for all valid pixels, occluded or not. We run each algorithm on both the original image, and the albedo image. Both algorithms perform slightly better with albedo. FlowNet 2 performs significantly better on all metrics.

Next we compare monocular depth prediction baselines: Depth in the Wild (DiW) [10] and Eigen *et al*. [14]. Both are trained on the KITTI dataset. We follow the evaluation metric used in DiW, and compute root mean squared error (RMSE), log scale RMSE, and the scale invariant log RSME. Table 4 summarizes the results. Neither of the two models transferred well to our GTA data. We suspect this might have to do with a slight viewpoint change between KITTI and GTA. The KITTI camera is mounted on a car, roughly two meters above ground, while the GTA camera is hood mounted one meter above ground, or at the players head position, one and a half meter above ground. Both depth models consistently predict the ground plane wrong.

method	RMSE	log-RMSE	s.i. log-RMSE
Mean depth	7.12	**0.575**	**0.084**
Eigen *et al*. [14]	7.49	0.740	0.161
DiW [10]	**7.09**	0.624	0.085

Table 4: Monocular depth estimation errors: root mean squared (RMSE), log RSME, and scale invariant log RSME.

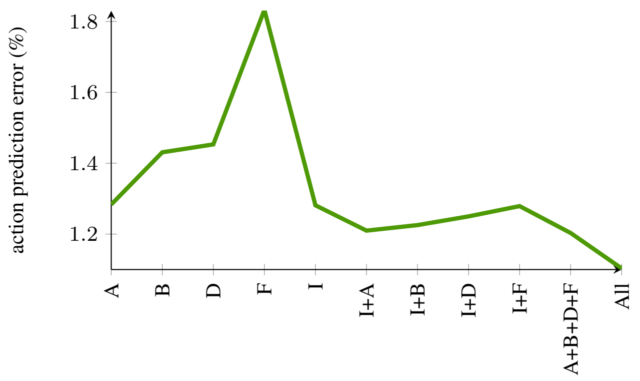


Figure 8: Action prediction using different modalities: Albedo (A), Instance Boundaries (B), Depth (D), Optical Flow (F) and the original Image (I). We measure classification error in percent. Lower is better.

Finally, we evaluate state of the art intrinsic image decomposition using the LMSE metric of Grosse*et al*. [20]. Direct Intrinsics [33] performed quite well with an LMSE of 0.077. Reflectance filtering [34] on the other hand didn't, at an LMSE of 0.086 with the original parameters, and 0.076 with highly tuned filter parameters. We suspect that the original parameters were tuned for the keypoint based evaluation of the IIW dataset [5]. None of the keypoints in IIW are selected close to discontinuities, hence excessive blurring of the albedo image is less penalized in IIW.

In summary, the novel dataset is slightly more challenging for both depth, and optical flow estimation. However, the data is easy to collect which will inspire larger and more powerful depth, and optical flow prediction networks.

6. Applications

Since our data collection is real time, it can be used to train autonomous driving agents. We follow Xu *et al*. [44] and Bojarski *et al*. [6], and train an imitation learning agent to act in GTA V. We limit ourself to $40k$ training images that involve driving a car in GTA V. During the gameplay, we record all actions performed by the auto-pilot: steer-left, steer-right, brake, accelerate. We binarize all actions and predict them using a classifier. Actions are heavily biased, most time is spent driving forward, and relatively little time goes into steering. This skewed distribution makes action prediction easy. A blind classifier, that does not consider the input, achieves an error rate of $\sim 4\%$. However looking at the input, still improved accuracy significantly.

Here, we are particularly interesting in figuring out which modalities help control and which ones don't. We use the albedo, depth, flow and color image as they were provided by our wrapper. Unfortunately, since there is not good way to feed instance segmentation directly into a deep neural network, we use instance boundaries instead. For each combination of modalities, we train a Network in Network model [28] on images resized to 256×256. We follow Xu *et al*., and use sigmoid cross entropy to predict the actions, and measure classification error at test time. We train each model from scratch with a batch size of 32 and 100k training iterations.

Figure 8 shows a summary of the results. To learn to act in GTA V, the original image, or the albedo image are most important. Depth or instance boundaries perform slightly worse. Optical flow alone produces a poor driving agent. This is in part due to the fact that optical flow is heavily influenced by camera motion, which at times can be distracting. Combining the image with any other modality slightly improved the imitation learning agent. Image and albedo lead to the largest improvement, and depth and boundaries closely follow. Combining all modalities performs best.

7. Discussion

In conclusion, we present a framework to extract ground truth supervision from video games in real-time. The supervisory signal is free and complementary to human annotations. Our approach is general, and is not tied to first person games. Thus far, we tried it on five video games: Fallout 4, The Witcher 3, GTA V, FarCry Primal, Total War - Warhammer. On all we were able to extract the image, albedo, and depth. On four, we can obtain instance segmentation and a semantic labeling. For GTA we additionally obtain optical flow, and occlusion boundaries. This can help research for general video game AIs. Figure 9 shows an example from the strategy game Total War - Warhammer. It took 30 minutes to hack the game and capture instance segmentations at 30 Hz.

Acknowledgment

We would like to thank Angela Lin and Thomas Crosley for helpful discussions. This work was supported in part by Berkeley DeepDrive and an equipment grant from Nvidia.

References

[1] P. Arbelaez, M. Maire, C. Fowlkes, and J. Malik. Contour detection and hierarchical image segmentation. *PAMI*, 2011.

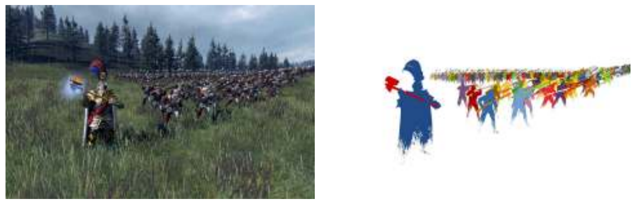


Figure 9: Our method easily generalizes to other types of video games, for example Total War - Warhammer. *Best view on screen*

1, 2

[2] C. Bailer, B. Taetz, and D. Stricker. Flow fields: Dense correspondence fields for highly accurate large displacement optical flow estimation. In *CVPR*, 2015. 7

[3] S. Baker, D. Scharstein, J. Lewis, S. Roth, M. J. Black, and R. Szeliski. A database and evaluation methodology for optical flow. *IJCV*, 2011. 2

[4] J. L. Barron, D. J. Fleet, and S. S. Beauchemin. Performance of optical flow techniques. *IJCV*, 1994. 2

[5] S. Bell, K. Bala, and N. Snavely. Intrinsic images in the wild. *ACM Transactions on Graphics (TOG)*, 2014. 6, 8

[6] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. 2, 8

[7] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black. A naturalistic open source movie for optical flow evaluation. In *ECCV*, 2012. 2, 6

[8] CD Project RED. The Witcher 3: Wild Hunt. http://thewitcher.com/en/witcher3. 1, 6

[9] Q. Chen and V. Koltun. Photographic image synthesis with cascaded refinement networks. In *ICCV*, 2017. 2

[10] W. Chen, Z. Fu, D. Yang, and J. Deng. Single-image depth perception in the wild. In *NIPS*, 2016. 7

[11] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In *CVPR*, 2016. 2, 6

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009. 1, 2

[13] J. Donahue, P. Krähenbühl, and T. Darrell. Adversarial feature learning. *ICLR*, 2017. 1

[14] D. Eigen, C. Puhrsch, and R. Fergus. Depth map prediction from a single image using a multi-scale deep network. In *NIPS*, 2014. 7

[15] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *ICCV*, 2010. 2

[16] GeForce forums. https://forums.geforce.com/default/topic/949101. Online; accessed 11-11-2017. 4

[17] A. Geiger, P. Lenz, and R. Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *CVPR*, 2012. 2, 6, 7

[18] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *NIPS*, 2014. 6

[19] R. Goroshin, M. F. Mathieu, and Y. LeCun. Learning to linearize under uncertainty. In *NIPS*, 2015. 1

[20] R. Grosse, M. K. Johnson, E. H. Adelson, and W. T. Freeman. Ground truth dataset and baseline evaluations for intrinsic image algorithms. In *ICCV*, 2009. 8

[21] A. Handa, T. Whelan, J. McDonald, and A. J. Davison. A benchmark for rgb-d visual odometry, 3d reconstruction and slam. In *ICRA*, 2014. 2

[22] B. K. Horn and B. G. Schunck. Determining optical flow. *Artificial intelligence*, 1981. 2

[23] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, and T. Brox. Flownet 2.0: Evolution of optical flow estimation with deep networks. *CVPR*, 2017. 7

[24] J. Johnson, B. Hariharan, L. van der Maaten, L. Fei-Fei, C. L. Zitnick, and R. Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *CVPR*, 2017. 1

[25] T. Jones. Parsing Direct3D shader bytecode. http://timjones.io/blog/archive/2015/09/02/parsing-direct3d-shader-bytecode. Online; accessed 11-11-2017. 4

[26] B. Kaneva, A. Torralba, and W. T. Freeman. Evaluation of image features using a photorealistic virtual world. In *ICCV*, 2011. 2

[27] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 6

[28] M. Lin, Q. Chen, and S. Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013. 6, 8

[29] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *ECCV*, 2014. 1, 2, 6

[30] N. Mayer, E. Ilg, P. Hausser, P. Fischer, D. Cremers, A. Dosovitskiy, and T. Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *CVPR*, 2016. 2, 6

[31] B. McCane, K. Novins, D. Crannitch, and B. Galvin. On benchmarking optical flow. *CVIU*, 2001. 2

[32] M. Menze and A. Geiger. Object scene flow for autonomous vehicles. In *CVPR*, 2015. 2

[33] T. Narihira, M. Maire, and S. X. Yu. Direct intrinsics: Learning albedo-shading decomposition by convolutional regression. In *CVPR*, 2015. 8

[34] T. Nestmeyer and P. V. Gehler. Reflectance adaptive filtering improves intrinsic image estimation. In *CVPR*, 2017. 8

[35] D. Pathak, P. Krähenbühl, J. Donahue, T. Darrell, and A. A. Efros. Context encoders: Feature learning by inpainting. In *CVPR*, 2016. 1

[36] W. Qiu and A. Yuille. Unrealcv: Connecting computer vision to unreal engine. *arXiv preprint arXiv:1609.01326*, 2016. 2

[37] S. R. Richter, Z. Hayder, and V. Koltun. Playing for benchmarks. In *ICCV*, 2017. 2, 6

[38] S. R. Richter, V. Vineet, S. Roth, and V. Koltun. Playing for data: Ground truth from computer games. In *ECCV*, 2016. 2

[39] Rockstar Games. Grand Theft Auto V. http://www.rockstargames.com/V. 1, 6

[40] T. Sharp, C. Keskin, D. Robertson, J. Taylor, J. Shotton, D. Kim, C. Rhemann, I. Leichter, A. Vinnikov, Y. Wei, et al. Accurate, robust, and flexible real-time hand tracking. In *ACM CHI*, 2015. 2

[41] J. Shotton, R. Girshick, A. Fitzgibbon, T. Sharp, M. Cook, M. Finocchio, R. Moore, P. Kohli, A. Criminisi, A. Kipman, et al. Efficient human pose estimation from single depth images. *PAMI*, 2013. 2

[42] N. Silberman, D. Hoiem, P. Kohli, and R. Fergus. Indoor segmentation and support inference from rgbd images. In *ECCV*, 2012. 1, 2, 6

[43] Ubisoft Montreal. Far Cry Primal. `https://far-cry.ubisoft.com/primal`. 1, 6

[44] H. Xu, Y. Gao, F. Yu, and T. Darrell. End-to-end learning of driving models from large-scale video datasets. *CVPR*, 2017. 2, 8

[45] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva. Learning deep features for scene recognition using places database. In *NIPS*, 2014. 1, 2