Building a Virtual 3D World Model for a Mobile Robot

by
Hong Yue

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Computer Science

at

Seidenberg School of Computer Science and Information Systems

Pace University

December 2015

Abstract

Building a Virtual 3D World Model for a Mobile Robot

by
Hong Yue

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Computer Science

December 2015

In this paper I render the real world into a 3D virtual world in real-time for comprehension of a cognitive mobile robot. I take a moving vehicle as an example and register it into a 3D model, to help the mobile robot comprehend the motions of the vehicle and generate its own task plan.

As an initial step, I build a 3D vehicle model using PhysX to represent the real world to reduce noise and interference. In the demo, the vehicle is controlled by keyboard, and the camera is fixed to make it easier to find the velocity of the vehicle. I record a video of the moving vehicle as the input of the system.

After that I train a set of cascade classifiers to detect and recognize the vehicle in frames of the video. I discuss Haar feature-based classifiers first and identify the Haar feature's problem in this specific case. Then I use Local Binary Patterns (LBP) to do the vehicle detection and recognition in image.

I also find the velocity of the vehicle by comparing the position of the vehicle in two or more frames. The distance traveled is divided by the time between the frames to give the approximate velocity.

Table of Contents

List of Figures

Chapter 1

Introduction: A Cognitive Approach to Robotics

Nowadays, robots are needed that can help people perform tasks in settings that are dangerous or repetitive and dull, in the home or the hospital or factory. This requires sophisticated human-robot interaction in which the robot can comprehend and predict human motions, and plan responses that are cooperative and avoid harm. I am rendering the real world into a 3D virtual world in real-time for comprehension of a cognitive mobile robot. I am taking a moving vehicle as an example and register it into a 3D model to help the mobile robot comprehend the motions of the vehicle and generate its own task plans as respond to vehicle movements.

The main research objectives in the whole picture are to demonstrate efficient use of stereovision to recognize objects, people and motions and render them accurately into the simulator, and demonstrate the use of a 3D simulator with realistic physics to navigate and generate task plans. The long-term goal of this project is to demonstrate smooth interaction and cooperation with humans and other robots in a range of settings. The settings to be used include navigating alongside a walking person, navigating through walking people, and crossing a busy street while avoiding moving vehicles

This project is part of a large multi-university multidisciplinary collaboration spanning

more than ten years. The ADAPT project (Adaptive Dynamics and Active Perception for Thought) is a collaboration of three university research groups at Pace University, Brigham Young University, and Fordham University to produce a robot cognitive architecture that integrates the structures designed by cognitive scientists with those developed by robotics researchers for real-time perception and control. The goal of ADAPT is to create a new kind of robot architecture capable of robust behavior in unstructured environments, exhibiting problem solving and planning skills, learning from experience, novel methods of perception, comprehension of natural language and speech generation.

1.1 Basic Concept of Machine Learning and Artificial Intelligence

Machine learning is a type of artificial intelligence (AI) that provides computers with the ability to learn without being explicitly programmed. Machine learning focuses on the development of computer programs that can teach themselves to grow and change when exposed to new data.

The process of machine learning is similar to that of data mining. Both systems search through data to look for patterns. However, instead of extracting data for human comprehension – as is the case in data mining applications – machine learning uses that data to improve the program's own understanding. Machine learning programs detect patterns in data and adjust program actions accordingly. For example, a recognition part of a perceptual and intelligent robot can recognize and detect a certain object in a new circumstance after "teaching" it what the object looks like in various circumstances; and a decision-making part can generate different task plans according to the objects around

and their states after learning about similar events for thousands of times. In this way, we can make a robot “perceptual” and “artificial-intelligent”.

1.2 Human-Robot Interaction of ADAPT

Tasks that involve robots interacting with humans require the robots to be able to comprehend the humans’ motions, to avoid misunderstanding or injury.

Nonverbal behaviors are extremely important to understand, because much of the interaction between actors in the real world is nonverbal. For example, walking alongside a person involves keeping a relatively constant distance from the person, which requires monitoring the person’s movements and responding appropriately to them. If the person speeds up or slows down or changes direction, there is probably a reason, and an intelligent robot should understand that reason in order to respond appropriately. If a team is composed of several people and several robots, it is not possible for the people to continually explain their movements to the robots; it is necessary for the robots to be able to generate their own explanations.

ADAPT models the world as a network of concurrent schemas, and models perception as a problem solving activity. Schemas are represented using the RS (Robot Schemas) language [9,10], and are activated by spreading activation. RS provides a powerful language for distributed control of concurrent processes. Also, the formal semantics of RS [8] provides the basis for the semantics of ADAPT's use of natural language. ADAPT have implemented the RS language in Soar [5,6], a mature cognitive architecture originally developed at Carnegie-Mellon University and used at a number of universities

and companies. Soar's subgoaling and learning capabilities enable ADAPT to manage the complexity of its environment and to learn new schemas from experience.

ADAPT generates the explanations to actors' movements by using the PhysX world model (https://developer.nvidia.com/gameworks-physx-overview) to simulate the observed behaviors of other actors. Each actor is represented as a Soar agent that is identical to the robot. ADAPT attempts to replicate the observed behavior by searching through sequences of Soar operator firings for that agent. When an appropriate sequence of Soar operators is found that replicates the observed behavior, ADAPT assumes that the agent has executed this sequence of operators, and attributes to the agent the goals and knowledge used by these operators. ADAPT then generates this sequence of behaviors.

For example, suppose the robot is traveling with a human teammate and the observed behavior is that the human looks ahead and slows down a bit because he sees a vehicle crossing his path and wishes to let it pass. This is a typical behavior that should not need to be explained to the robot. ADAPT can model the human and the vehicle in its 3D world model, and "imagine" the consequences of not slowing down, which will be a collision. This permits ADAPT to infer the reason for the behavior, and to create a new schema for slowing down to let vehicles pass. This schema is used both for generation (when the robot needs to slow down to permit vehicles to pass) and for comprehension.

1.3 A Single Vehicle's Behaviors

To accomplish the comprehension explained above I am trying to make the robot

understand a single vehicle's behaviors first. For example, assume a car is running above the robot. A cognitive robot is expected to see and recognize the car and then find the real-time velocity of the car; and it will able to predict that the car will keep the same velocity under a short time (we say, a quarter second) till the robot notices that the car's velocity has changed and the velocity-finding task needs to start again. As a sequence the robot can predict the events that are going to happen, for example, a collision. In this way the "comprehension" on the vehicle's behaviors is accomplished, after which the robot can generate its own sequence of behaviors.

Chapter 2

A Cognitive Approach to Vision in ADAPT

ADAPT's virtual world is not connected to its sensory processes. ADAPT's sensory data is placed directly in the reasoning engine (after some low-level processing); the reasoning engine's principal task in ADAPT is to reason about how to model the data. It does this using the following loop:

> It compares visual data from the camera with visual data from the corresponding virtual camera, using a least-squares measure, the Match-Mediated Difference System (MMD) to find areas of disagreement. Each disagreement causes a Soar operator to be proposed to attend to that disagreement.
>
> One Soar operator is selected, based on the robot's current goals. For example, if the goal is navigation, operators will be preferred that are for visual disagreements in the robot's current path. The selected operator fires, causing the cameras to saccade to the area of disagreement and fixate on it.
>
> Stereo disparity, color segmentation, and optical flow are computed only in the small region of focus. Restricting the computation to this small region permits the use of highly accurate but computationally expensive algorithms for these computations, e.g. disparity of disparities.

This information is input to the object recognition database, and a mesh model of the best match is rendered into the virtual world. If the information indicates that a current mesh model is inaccurate, that model is modified to incorporate the new information.

The Soar reasoning engine searches alternative combinations of virtual entities and behaviors to attempt to minimize the measured disagreement. In this way, perception becomes a problem-solving process. This enables all the knowledge of the system to be brought to bear on perception, and unifies the reasoning and learning processes of problem solving with those of perception.

The research hypothesis of this work is that the movements of the robot's cameras are only those that are necessary to build a sufficiently accurate world model for the robot's current goals. For example, if the goal is to navigate through a room, the model needs to contain any obstacles that would be encountered, giving their approximate positions and sizes. Other information does not need to be rendered into the virtual world, so this approach trades model accuracy for speed.

The search for what to examine in the camera input is interleaved with task search (the selection of task operators). ADAPT is expected to perceive only what is necessary for the task, pruning information at the perceptual stage instead of interpreting everything then deciding what to ignore. This requires the vision system to have a search strategy, which is executed by saccades and fixations.

ADAPT is evaluating a number of possible heuristics for this search:

Move to the area with the largest disagreement with the 3D model,

Move to the area with most vision data (e.g. densest concentration of keypoints),

Move to the closest area to current area that has a disagreement exceeding a set threshold,

Move to the closest area that is on a major line or curve from current area.

Sensory input goes directly to the Soar/RS reasoner, not through a world model. The reasoner explores how to interpret the world.

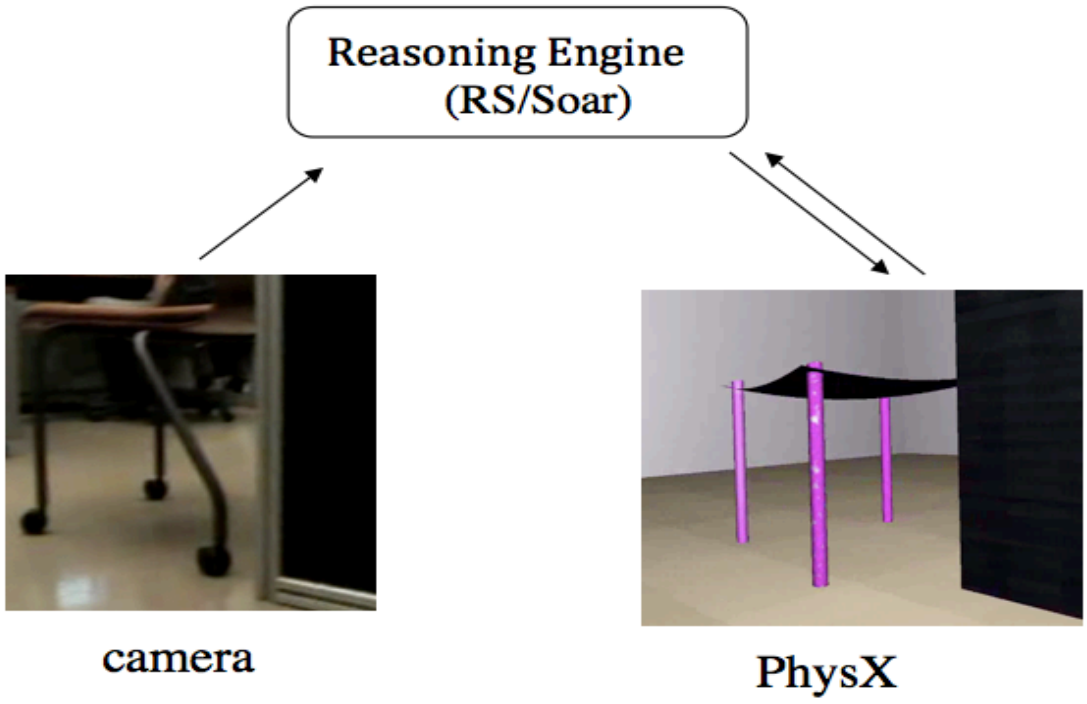


Figure 1The Soar/RS reasoning engine renders the camera input into PhysX using its knowledge about the world

Instead of rendering the entire scene in detail as a point cloud then registering it, high-accuracy rendering is taken only in a small region of the center of the scene. The camera moves from point to point in the scene rendering small parts that are combined by Soar into a mesh model.

This cognitive approach attempts to model the human vision system, particularly tracking strategies. High-accuracy algorithms are used only in the center of the visual field. The cognitive system must guide the cameras to parts of the visual field that provide information that is relevant to task goals, and especially to the goal of building the virtual model.

It is believed that this will lead to better performance on object recognition, tracking and activity comprehension.

Chapter 3

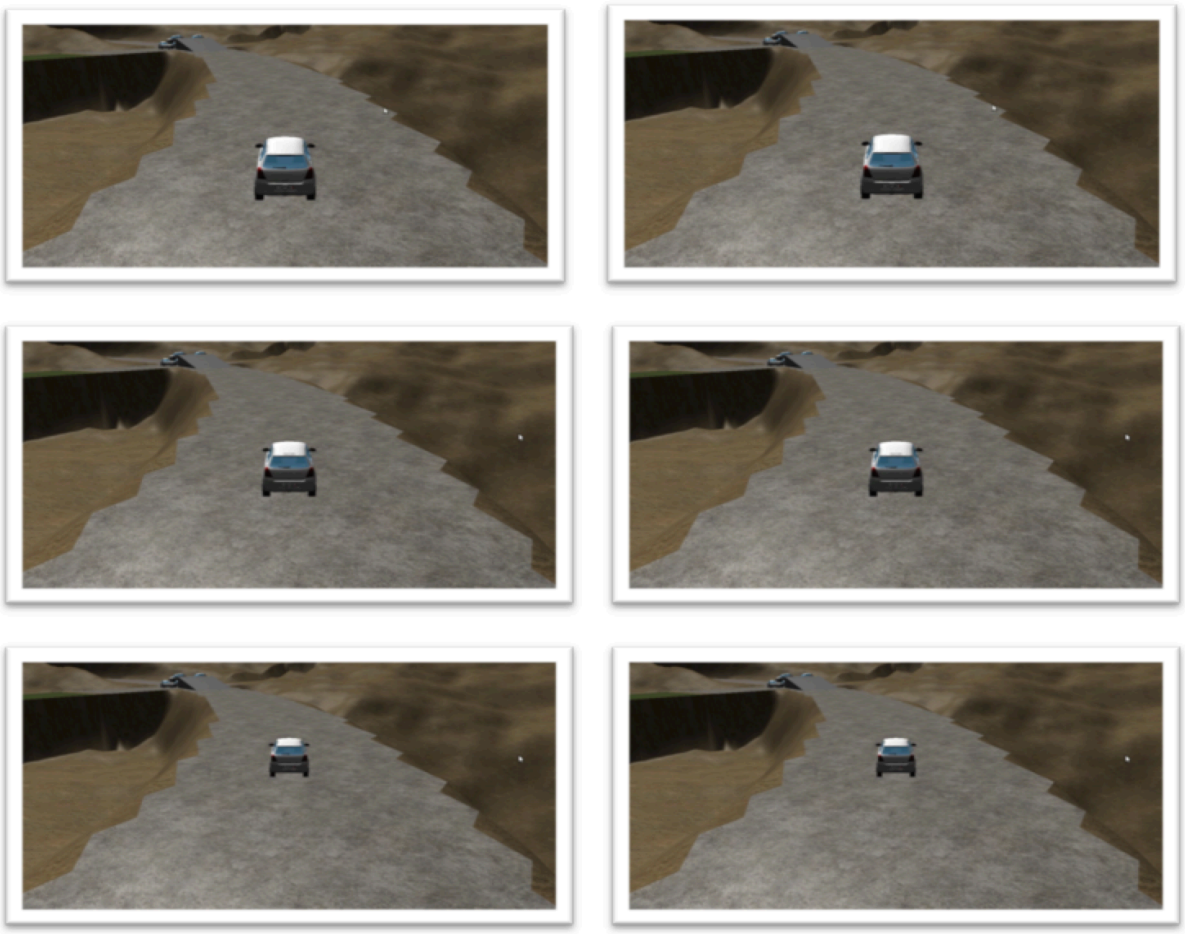
Example: Tracking a Moving Car

The current work is to monitor and predict the motion of a car so that we can drive another car beside it without any accidents. The whole 3D modeling project is patterning its work on the KITTI project because of its excellent data, and the goal is to render actual traffic into the 3D model in real time.

In this paper a 3D virtual world simulated by PhysX is built to represent the real world, in which a car is running in the road, and to create the output synthetic world, too. Since the Match-Mediated Difference (MMD) is an idea of Professor Lyons in Fordham University a couple of years ago, it needs to be re-written into class structure and changed to a new test bed version. So for the single vehicle project the MMD part is omitted and assumed working; and the car in the input is registered and is modeled by the corresponding car mesh in the output directly.

Since the target and task in this project is clarified, it is not necessary to interleave the video input and the task search to find what to examine in the camera. And the camera is fixed to simplify the calculation for the car's velocity, so the information that needs to be rendered into the output synthetic world is only about the car.

The initial step is to use the PhysX vehicle demo for both worlds:



Real World **Virtual world**

Figure 2 The PhysX demo is used as both the real world and the virtual world, to develop the tracking capability

Figure 2 shows the use of the PhysX demo. The left column is the real world and the right column is the rendered world. In the first pair of images, the cars start synchronized. In the second pair of images, the real car begins to move, and differences are created between the images. In this demo, the virtual camera is fixed, so the difference is centered on the car itself; this forces the system to reason about how the differences have been caused, and to attempt to register the car with the help of MMD (we assume MMD is working here). Then the system tries to calculate the velocity of the real car through the

difference between frames and adjust the velocity of the virtual car to match the real one, shown in the bottom image.

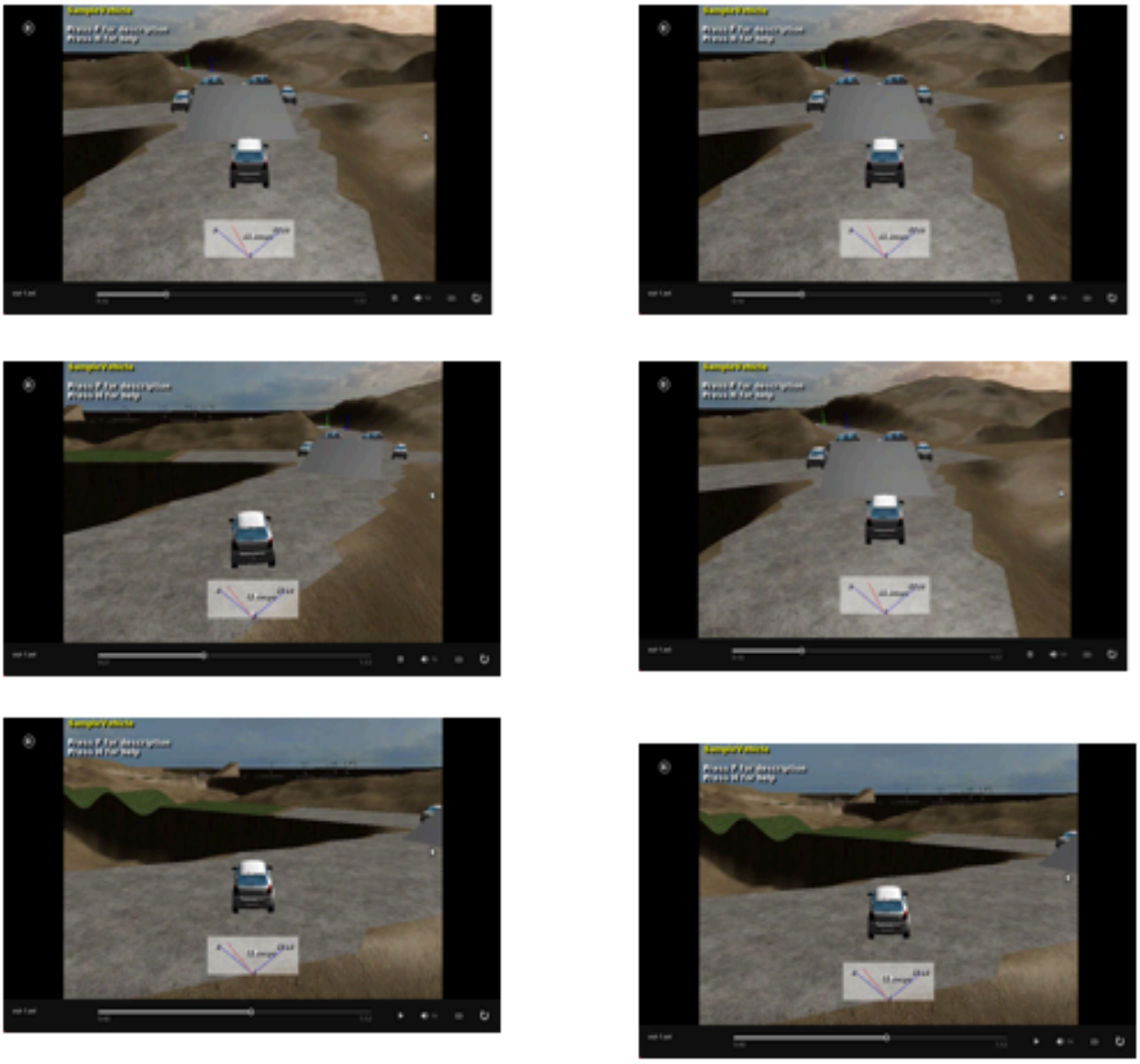


Figure 3 Another example showing the car turning

Figure 3 is another example, in which the real car turns left. In the middle pair of images, the difference caused by the turn is found. Once again, the reasoned searches to register the car (Again, assume MMD is working). And this time find the velocity of the real car in the input to derive the turn. The virtual car is repositioned and given the appropriate turning radius to match the real car at bottom.

Errors accumulate as the real car follows a path that is not exactly straight and the virtual car goes straight (similar to dead reckoning error). This is not comparable to other data because nobody else seems to be doing this kind of activity modeling/comprehension, e.g. the KITTI database has no data on this.

Chapter 4

Implementation

My current work is to monitor and predict the motion of a moving car, for demonstrating another autonomous vehicle in the future that can move alongside the moving car without any accidents, even as the moving car turns, stops and accelerates.

The longer - term goal is to add people (skeletonized) so that the vehicle can move with one or more people in its environment. Applications of this could include autonomous trucks and taxis, warehouse vehicles, autonomous retrieval of wounded soldiers from the battlefield, and humanoid robots that can move among people.

4.1 PhysX World Demo

To make an easier start, a 3D model created by PhysX is used to represent the real world to avoid noises and interference.

4.1.1 Scenes, Actors, and Materials

In PhysX, the most important objects are scenes and actors. PhysX represents the world using the *scene*, which is defined in PxScene object; *actors* are the individual elements of that world. In this demo, we adopt a PxScene object with the same immutable parameters with those in the PhysX official vehicle sample.

```
PxSceneDesc sceneDesc(mPhysics->getTolerancesScale());
sceneDesc.gravity = PxVec3(0.0f, -9.81f, 0.0f);
getDefaultSceneDesc(sceneDesc);
```

```
customizeSceneDesc(sceneDesc);

…

  mScene = mPhysics->createScene(sceneDesc);
```

(SampleBase/PhysXSample:line1080)

And the rigid actors are of two principal kinds: static and dynamic, corresponding to the PhysX classes PxRigidStatic and PxRigidDynamic. In this demo, static actors, which are immovable by the simulation, include ground, roads, hills, static obstacles and non-player vehicles; whereas dynamic actors such as the player vehicle and pendulous balls as dynamic obstacles have their positions updated by the simulation when simulate() is called.

Each object has at least one *material* to define the friction and restitution properties during collisions with other objects. And the PhysX demo performs *raycasts* in "multiple hits" mode for collision queries. A raycast query detects collisions by casting a ray and also report multiple hits with objects in the scene.

```
sqDesc.queryMemory.userRaycastResultBuffer = mSqResults;
sqDesc.queryMemory.userRaycastTouchBuffer = mSqHitBuffer;
sqDesc.queryMemory.raycastTouchBufferSize = mNumQueries;
```

4.1.2 Simulation

PhysX uses the method PxScene::simulate() to advance the world forward in time. The sample's fixed stepper class is called from the sample framework whenever the app is done with processing events and is starting to idle. It accumulates elapsed real time until it is greater than 0.016666660 second, and then calls simulate(), which moves all objects in the scene forward by that interval.

```
mDebugStepper(0.016666660f),
mFixedStepper(0.016666660f, maxSubSteps),
mInvertedFixedStepper(0.016666660f, maxSubSteps),
```

(SampleBase/PhysXSample.cpp:line 708)

```
void FixedStepper::substepStrategy(const PxReal stepSize, PxU32& substepCount, PxReal& substepSize)
{
  if(mAccumulator > mFixedSubStepSize)
  mAccumulator = 0.0f;        // don't step less than the step size, just accumulate
  mAccumulator  += stepSize;
  if(mAccumulator < mFixedSubStepSize)
  {
        substepCount = 0;
        return;
  }
```

```
  substepSize = mFixedSubStepSize;
  substepCount = PxMin(PxU32(mAccumulator/mFixedSubStepSize), mMaxSubSteps);

  mAccumulator -= PxReal(substepCount)*substepSize;
}
```

(SampleBase/SampleStepper.cpp/line188)

```
bool DebugStepper::advance(PxScene* scene, PxReal dt, void* scratchBlock, PxU32 scratchBlockSize)
{
  mTimer.getElapsedSeconds();
        {
                PxSceneWriteLock writeLock(*scene);
                scene->simulate(mStepSize, NULL, scratchBlock, scratchBlockSize);
        }
  return true;
}
```

4.1.3 The Player Vehicle

PhysX provides a vehicles component in a manner similar to PhysX Extensions. The vehicle SDK can be thought of as having two separate components: the core vehicle SDK and an optional set of utility classes and functions that are provided as a reference

solution to common problems in game vehicle dynamics. We use most of the vehicle utility classes but with some modifications.

The PhysX Vehicle SDK models vehicles as collections of sprung masses, where each sprung mass represents a suspension line with associated wheel and tire data. These collections of sprung masses have a complementary representation as a rigid body actor whose mass, center of mass, and moment of inertia matches exactly the masses and coordinates of the sprung masses. The purpose of the PhysX Vehicle SDK update function is to compute suspension and tire forces using the sprung mass model and then to apply the aggregate of these forces to the PhysX SDK rigid body representation in the form of a modified velocity and angular velocity. Interaction of the rigid body actor with other scene objects and global pose update is then managed by the PhysX SDK.

The update of each vehicle begins with a raycast for each suspension line, with the raycast starting at the wheel center and casting downwards along the direction of suspension travel. The suspension force from each elongated or compressed spring is computed and added to the aggregate force to be applied to the rigid body. Additionally, the suspension force is used to compute the load that is bearing down on the tire. This load is used to determine the tire forces that will be generated in the contact plane and then added to the aggregate force to be applied to the rigid body. The tire force computation actually depends on a number of factors including steer angle, camber angle, friction, wheel rotation speed, and rigid body momentum. The aggregated force of all tire and suspension forces is then applied to the rigid body actor associated with the vehicle.

In addition to being collections of sprung masses, PhysX vehicles also support a variety of drive models. The center of the drive model is a torsion clutch, which couples together the wheels and the engine via forces that arise from differences in rotational speeds at both sides of the clutch. At one side of the clutch is the engine, a 1D rigid body that is powered directly from the accelerator pedal. At the other side of the clutch are the gearing system, the differential and the wheels. The effective rotational speed of the other side of the clutch can be computed directly from the gearing ratio and the rotational speed of the wheels that are coupled to the clutch through the differential. This model naturally allows engine torques to propagate to the wheels and wheel torques to propagate back to the engine, just as in a standard car[13].

To simulate the real world, we hide all the menu, description, screen texts and waypoints provided by the vehicle sample SDK.

```
SampleVehicle::SampleVehicle(PhysXSampleApplication& app) :
        …
        mHideScreenText                                         (true),
        …
```

(Samples/SampleVehicle/SampleVehicle.cpp: line 308)

```
void SampleVehicle::customizeSample(SampleSetup& setup)
{
        setup.mName  = " ";
         }
```

(Samples/SampleVehicle/SampleVehicle.cpp: line 336)

```
void SampleVehicle::onTickPostRender(PxF32 dtime)
{
   // Fetch results
   PhysXSample::onTickPostRender(dtime);
   if(mDebugRenderFlag)
   {
         drawWheels();
         drawVehicleDebug();
   }
   //Draw the next three way-points.
   //                                    const                                    RendererColor
colors[3]={RendererColor(255,0,0),RendererColor(0,255,0),RendererColor(0,0,255)};
   // PxVec3 v[3];
   // PxVec3 w[3];
   // PxU32 numPoints=0;
   //
mWayPoints.getNextWayPointsAndLineDirs(numPoints,v[0],v[1],v[2],w[0],w[1],w[2]);
   // for(PxU32 i=0;i<numPoints;i++)
   // {
   //      getDebugRenderer()->addLine(v[i],v[i]+PxVec3(0,5,0),colors[i]);
   //      getDebugRenderer()->addLine(v[i]-w[i],v[i]+w[i],colors[i]);
   // }
 }
```

(Samples/SampleVehicle/SampleVehicle.cpp: line 715)

```
void SampleVehicle::helpRender(PxU32 x, PxU32 y, PxU8 textAlpha)
{
}
```

(Samples/SampleVehicle/SampleVehicle.cpp: line 860)

```
 void SampleVehicle::descriptionRender(PxU32 x, PxU32 y, PxU8 textAlpha)
{
}
```

(Samples/SampleVehicle/SampleVehicle.cpp: line 942)

```
void SampleVehicle::customizeRender()
{
   drawHud();
   if(mDebugRenderFlag)
   {
          drawFocusVehicleGraphsAndPrintTireSurfaces();
   }
   const PxU32 width=getCamera().getScreenWidth();
   const PxU32 height=getCamera().getScreenHeight();
   const PxU32 xCentre=280*width/800;
   const PxU32 xRight=570*width/800;
   const PxU32 yBottom=600*height/600;
   const PxU32 yInc=18;
   Renderer* renderer = getRenderer();
```

```
  // char time[64];
  // sprintf(time, "Curr Lap Time: %1.1f \n", mWayPoints.getTimeElapsed());

  // renderer->print(xRight, yBottom - yInc*2, time);
  // sprintf(time, "Best Lap Time: %1.1f \n", mWayPoints.getMinTimeElapsed());

  // renderer->print(xRight, yBottom - yInc*3, time);
  // if(!mCameraController.getIsLockedOnVehicleTransform())
  // {
  //      renderer->print(xCentre, yBottom - yInc*4, "Camera decoupled from car");
  // }
}
```

(Samples/SampleVehicle/SampleVehicle.cpp: line 974)

4.1.4 Fixed Camera

In the PhysX vehicle SDK the camera follows the car all the time, but this brings more work to find the car's real velocity due to the relative velocity between the camera and the car. To make it easier we fix the camera at the back of the start point.

```
void SampleVehicle::updateCameraController(const PxF32 dtime, PxScene& scene)
{
  mCameraController.update(0,*mVehicleManager.getVehicle(mPlayerVehicle),scene);
}
```

(Samples/SampleVehicle/SampleVehicle.cpp:line2308)

4.2 Vehicle Detection & Recognition

To build a synthetic vehicle that moving in the same way with the one in real world, the moving vehicle in the real world is detected and recognized first. The general idea is to not look at the whole image as a high-dimensional vector, but describe only local textural features of an object. The feature extracted this way will have a low-dimensionality implicitly.

In this paper, a Haar feature-based cascade classifier in the OpenCV library is trained first to try the vehicle detection in frames.

4.2.1 Haar Feature-based Cascade Classification Algorithm

The word “cascade” in the classifier name means that the resultant classifier consists of several simpler classifiers (stages) that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed. The word “boosted” means that the classifiers at every stage of the cascade are complex themselves and they are built out of basic classifiers using one of four different boosting techniques (weighted voting). Currently Discrete Adaboost, Real Adaboost, Gentle Adaboost and Logitboost are supported. The basic classifiers are decision-tree classifiers with at least 2 leaves. Haar-like features are the input to the basic classifiers, and are calculated as described below. The current algorithm in OpenCV 2.4.11 uses the following Haar-like features [14]:

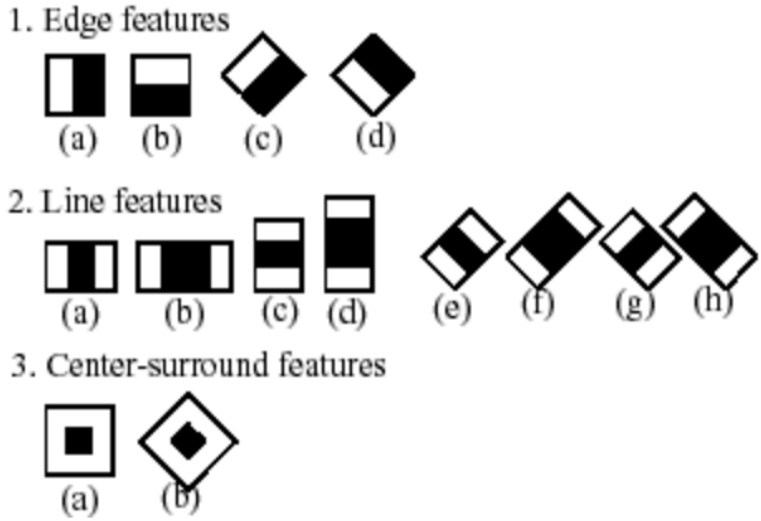


Figure 4 Haar-like features that are used as input

The feature used in a particular classifier is specified by its shape (1a, 2b etc.), position within the region of interest and the scale (this scale is not the same as the scale used at the detection stage, though these two scales are multiplied). For example, in the case of the third line feature (2c) the response is calculated as the difference between the sum of image pixels under the rectangle covering the whole feature (including the two white stripes and the black stripe in the middle) and the sum of the image pixels under the black stripe multiplied by 3 in order to compensate for the differences in the size of areas (I am using OpenCV 2.4.11 here. But in OpenCV 3.0.0, they changes both the Haar-like features shown in Figure 5 and the way to calculate the feature by subtracting sum of pixels under white rectangle from sum of pixels under black rectangle [15]). The sums of pixel values over rectangular regions are calculated rapidly using integral images.

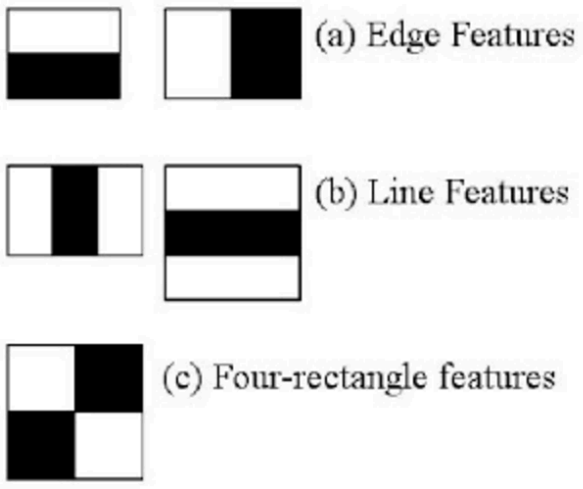


Figure 5 Latest Haar-like feature used in OpenCV

4.2.2 Detection with a Cascade Classifier

First, a Haar-feature classifier is trained with a few hundred sample views of a particular object (i.e., the car), called positive examples, that are scaled to the same size (e.g., 48 px* 48 px), and negative examples - arbitrary images of the same size.

After a Haar-feature classifier is trained, it is applied to a region of interest in an input image. The classifier outputs a "1" if the region is likely to show the object, and "0" otherwise. To search for the object in the whole image one can move the search window across the image and check every location using the classifier. The classifier is designed so that it can be easily "resized" in order to be able to find the objects of interest at different sizes, which is more efficient than resizing the image itself. So, to find an object of an unknown size in the image the scan procedure should be done several times at different scales.

OpenCV comes with a detector as well as trainer and it works following the following steps:

First one needs to load the required XML classifiers, and then load the input image (or video) in grayscale mode.

```
…
String vehicle_cascade_name = "/home/hong/Documents/classifiers/cascade.xml";
CascadeClassifier vehicle_cascade;
...
char* imageName = argv[1];
   Mat frame;
   frame = imread( imageName, 1 );
…
 Mat frame_gray;
   cvtColor( frame, frame_gray, COLOR_BGR2GRAY );
   equalizeHist( frame_gray, frame_gray );
```

(detectionFromImage.cpp/detectionFromVideo.cpp line 28-29, 36-38, 65-68)

Then the classifier is applied to the input image (or video) by CascadeClassifier:detectMultiScale Class:

```
void CascadeClassifier::detectMultiScale(const Mat& image, vector<Rect>& objects,
double scaleFactor=1.1, int minNeighbors=3, int flags=0, Size minSize=Size(),
Size maxSize=Size())
```

In the parameters, “objects” refer to the vector of rectangles where each rectangle contains the detected object; “scaleFacor” specifies how much the image size is reduced

at each image scale; “minNeighbors” specifiers how many neighbours each candidate rectangle should have to retain it. And the minimum and maximum possible object sizes are also specified in the class which means objects smaller or larger than the range we set are ignored.

In this paper, the parameters used are:

```
vehicle_cascade.detectMultiScale( frame_gray, vehicles, 1.1, 7, 0, Size(200,
200),Size(250,250) );
```

(detectionFromImage.cpp/detectionFromVideo.cpp line 71)

4.2.3 Haar Feature-based Cascade Classifier Problem

The classifiers are trained by an application: opencv_traincascade in OpenCV 2.4.11, written in C++.

For training there is a set of samples that are of two types: negative and positive.

Negative samples (also called background samples) are taken from arbitrary images that don’t contain the vehicle. Generally thousands of negative samples are needed to train a haar feature-based cascade classifier. In this case 627 negative samples are enumerated in a text file manually, and each line in the text file contains an image filename (relative to the directory of the description file) of negative sample image, shown in Figure 6.

```
./negatives1/highlevel001.png_0000_0548_0007_0617_0337_11.png
./negatives1/highlevel022.png_0000_0295_0032_0290_0506_37.png
./negatives1/highlevel010.png_0000_0761_0078_0421_0292_26.png
./negatives1/highlevel003.png_0000_0450_0035_0530_0287_08.png
./negatives1/highlevel001.png_0000_0548_0007_0617_0337_72.png
./negatives1/highlevel001.png_0000_0548_0007_0617_0337_69.png
./negatives1/highlevel002.png_0000_0398_0449_0731_0234_49.png
./negatives1/highlevel003.png_0000_0450_0035_0530_0287_09.png
./negatives1/highlevel007.png_0000_0306_0039_0842_0290_101.png
./negatives1/highlevel007.png_0000_0306_0039_0842_0290_71.png
./negatives1/highlevel001.png_0000_0548_0007_0617_0337_58.png
./negatives1/highlevel004.png_0000_0741_0073_0611_0292_36.png
./negatives1/highlevel010.png_0000_0761_0078_0421_0292_48.png
./negatives1/highlevel002.png_0000_0398_0449_0731_0234_46.png
./negatives1/highlevel003.png_0000_0450_0035_0530_0287_06.png
./negatives1/highlevel019.png_0000_0870_0360_0407_0187_06.png
./negatives1/highlevel019.png_0000_0165_0223_0376_0420_03.png
./negatives1/highlevel010.png_0000_0761_0078_0421_0292_51.png
```

Figure 6 Negative sample txt file fragment

Described images can be of different sizes; but since the background images are cropped using Photoshop in batches, it takes less effort to make them of the same size. These images are used to subsample negative image to the training size, each image are larger than a training window size, which is 24 px *24 px, shown in Figure 7.

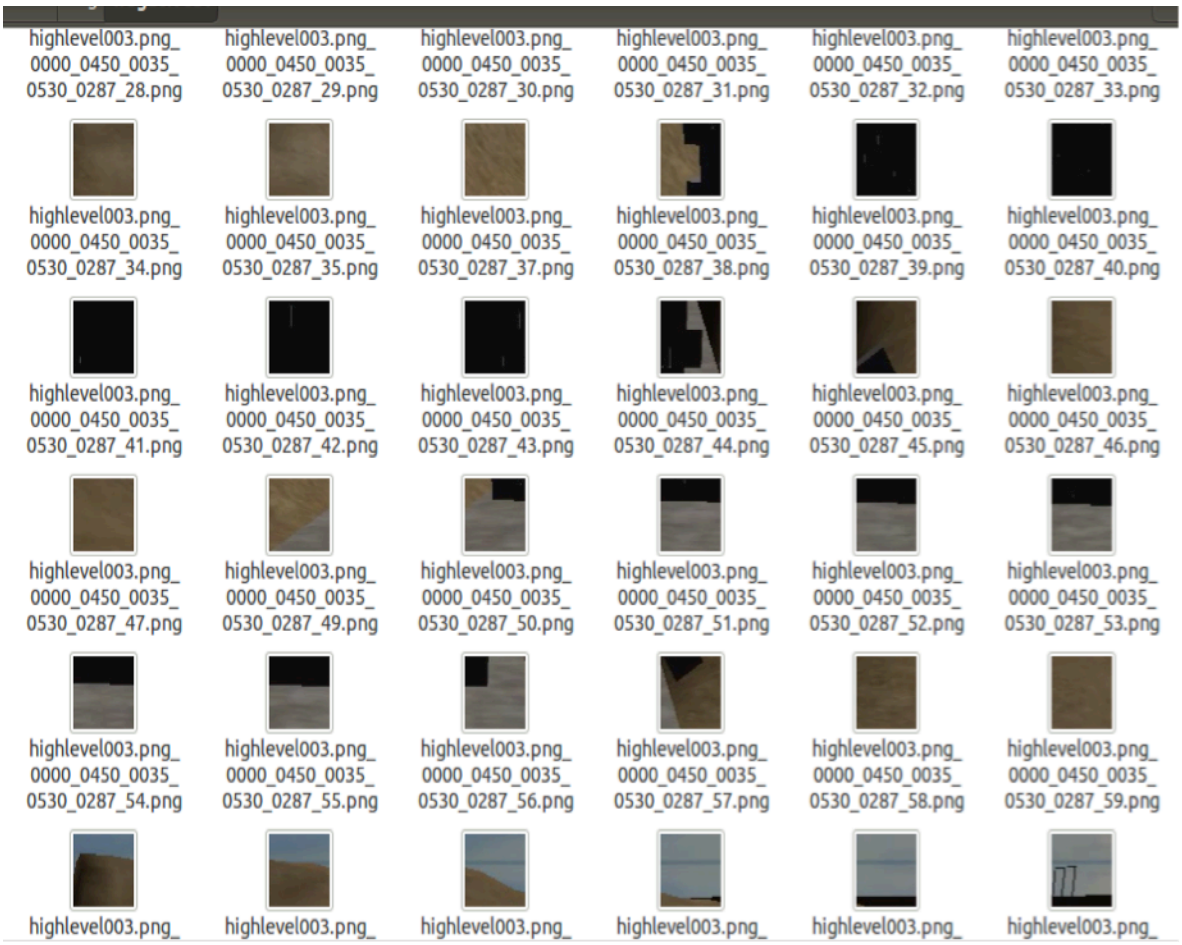


Figure 7 Some negative samples during Haar feature-based classifier training

Positive samples are created by opencv_createsamples utility from a single image with a size of 48*48 of the vehicle.

Figure 8 The single images used to train the positive samples

Since the white vehicle is obvious from the background, 300 positive samples are created from the above vehicle image by random rotating, changing the intensity as well as placing the vehicle on arbitrary background.

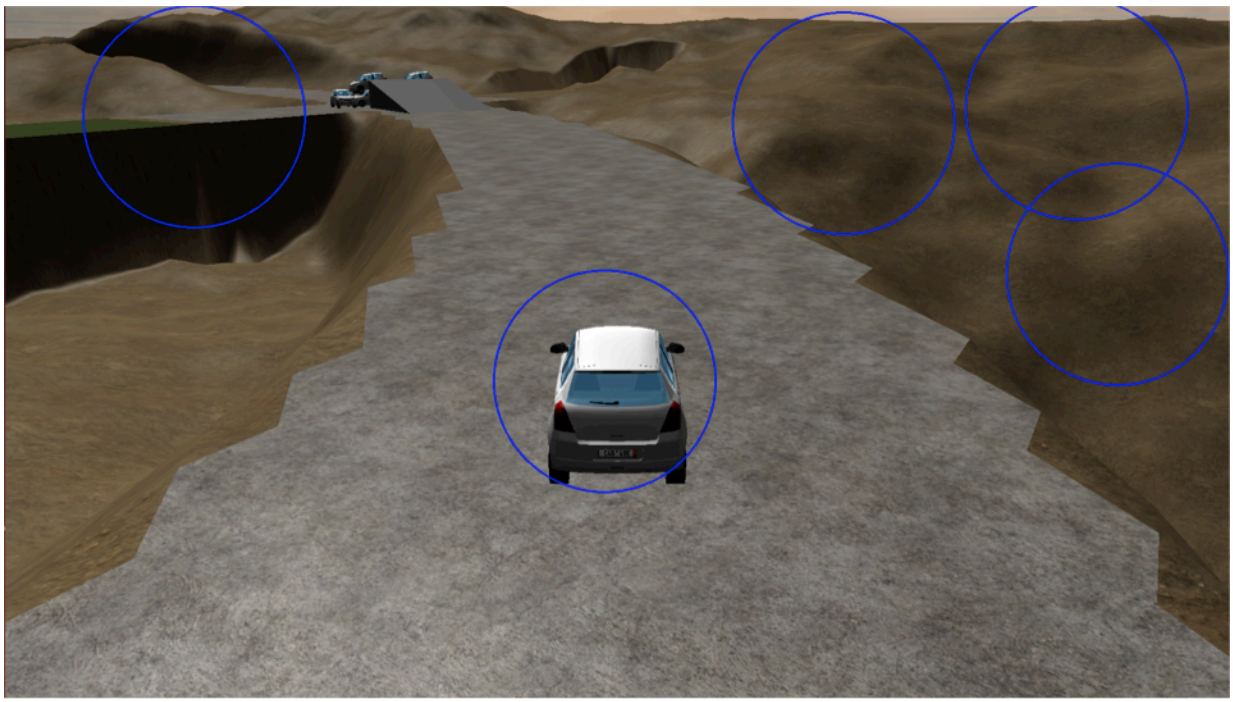


Figure 9 Haar cascade classifier detection result

But the Haar feature-based classifier trained cannot distinguish background and the vehicle ideally. (Figure 9)

To improve the performance I crop the misclassified parts in the background and put them in the background samples again to force the machine to learn these parts again. With an increase of 153 background samples, the Haar feature-based training algorithm still needs more samples to get a better result:

Train dataset for temp stage can not be filled. Branch training terminated.

4.2.4 Haar Feature-based Cascade Classifier Problem Analysis

One of the reasons why there is wrong classification is that the detection process is undertaken in grayscale (shown in Figure 10); for the classifier, the range of hills along the road has similar black-white outlook with the vehicle in the test image.

It is possible to solve this problem and get a better result by increasing negative samples to increase the ratio of negative samples to positive samples during classifier training, as the algorithm requires. But that is not that easy within the current project and its negative dataset. To meet the demand of Haar feature-based cascade classifiers, more than one thousand negative samples with a good diversity are needed. Since the positive samples are cropped out of the PhysX frames, the diversity of the negative samples is not large enough.

Another option is to use the same dataset, but increase the negative samples number during training to make a window for cropping more negative images from the dataset. However, this will make the negative images too small to contain the texture information of the background (e.g., the hills). As a result the system has a relatively high possibility to recognize the textures of the hills as the car by mistake, which is still the same problem.

A third option is to use another dataset where the png files have a great diversity, shown in Figure 11. Still by increasing the negative samples number and using cropping window during training we can get a large number of negative samples. But the system cannot get enough information to distinguish the background texture and the car due to the large demand of Haar feature-based cascade classifier for training samples.

Train dataset for temp stage can not be filled. Branch training terminated.

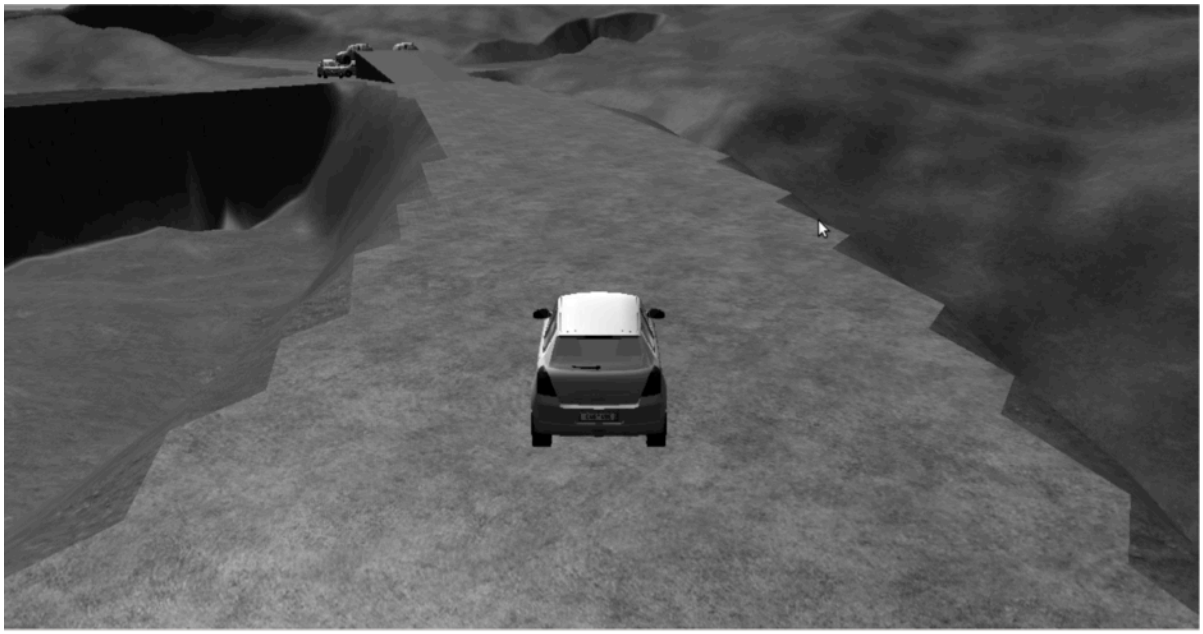


Figure 10 Test image in grayscale

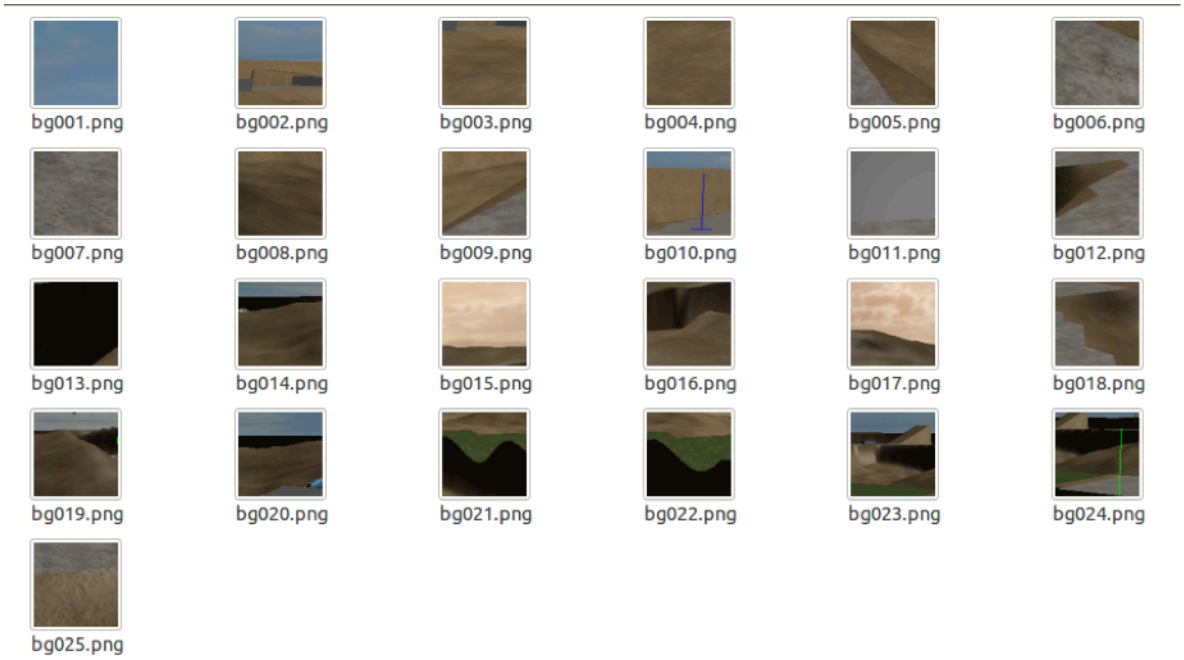


Figure 11 Negative dataset with a great diversity

To solve the problem Local Binary Patterns (LBP) features are used, which acquires both less samples and less time for cascade classifier training.

4.2.5 Local Binary Patterns Feature-based Cascade Classifiers Algorithm

Local binary patterns(LBP), introduced by Ojala et al, is another type of feature used for classification in computer vision. In contrast to Haar features, LBP features are integer, so both training and detection with LBP are several times faster then with Haar features. Regarding the LBP and Haar detection quality, it depends on training: the quality of training dataset first of all and training parameters too. It's possible to train a LBP-based classifier that will provide almost the same quality as Haar-based one but with smaller dataset and less time.

The basic local binary pattern is based on the assumption that texture has locally two complementary aspects, a pattern and its strength. LBP describes the local textual patterns as a two-level version of the texture unit [19]. The basic idea is to summarize the local structure in an image by comparing each pixel with its neighborhood.

The original version of LBP works in a 3 * 3 pixel block of an image. Each pixel is assigned a value by its gray value (Figure 12). The pixels in the block are threshholded by its center pixel value. If the intensity of the center pixel is greater-equal its neighbor, then denote it with 1 and 0 if not. (Figure 13) So with 8 surrounding pixels there are 2^8 possible combinations, called Local Binary Patterns or sometimes referred to as LBP codes.

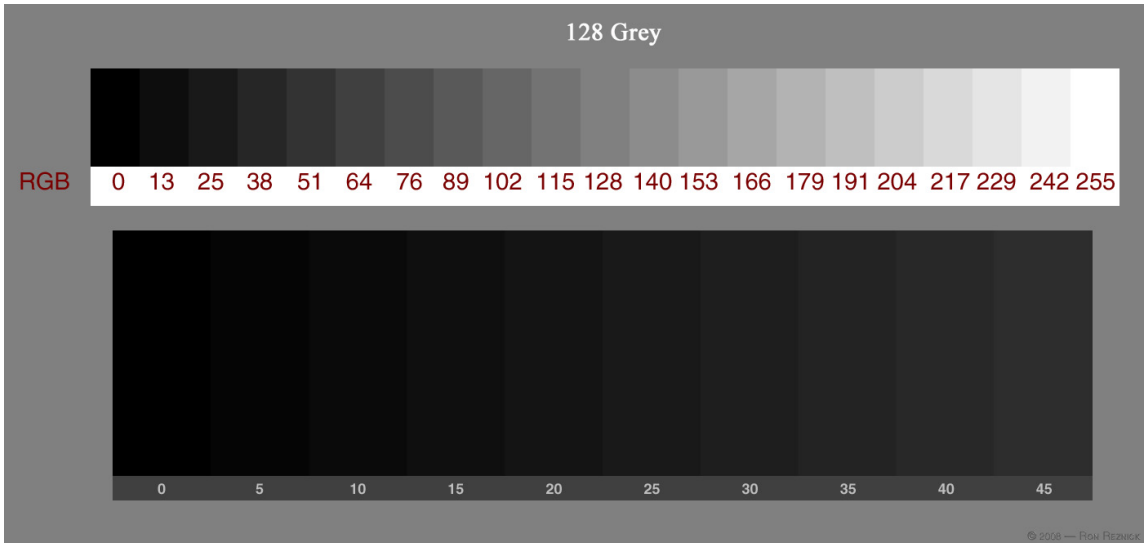


Figure 12 Gray value

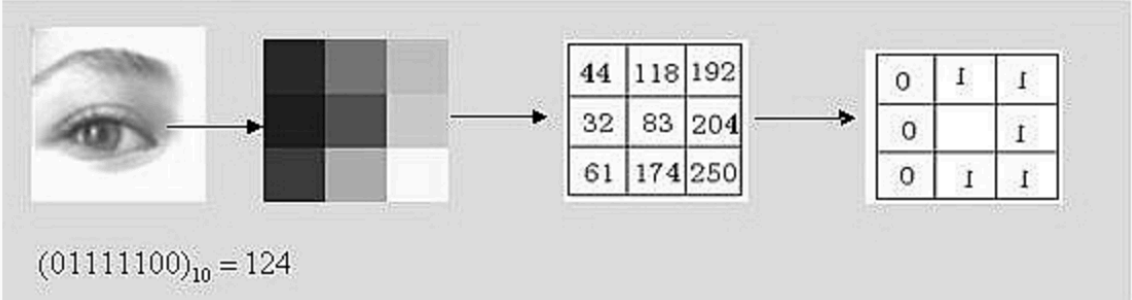


Figure 13 Basic Local Binary Pattern operator

Since LBP operator gets a LBP code (Figure 13) from every pixel, by definition it is obvious that the result of LBP process is still an "image"; and the LBP operator is robust against monotonic gray scale transformations. Following is a face image example provided by OpenCV [19](Figure 14):

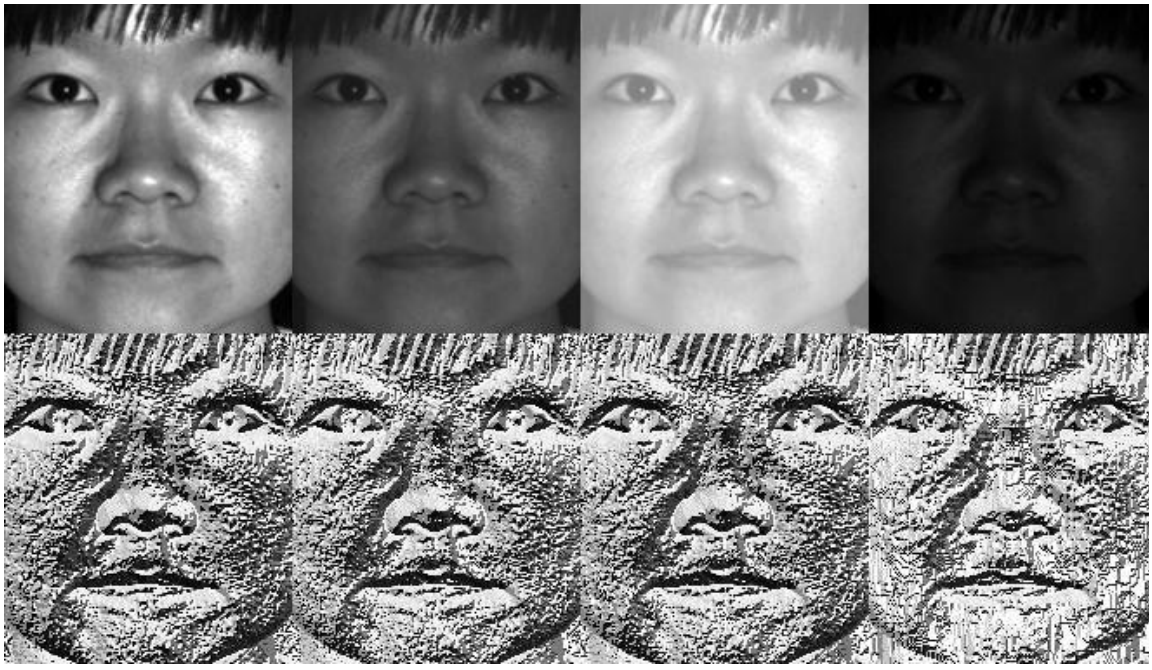


Figure 14 Robustness of LBP operator against monotonic scale transformations (the second line are LBP images)

Several years after its original publication, the local binary pattern operator was presented in a more generic revised form by Ojala et al to encode details differing in scale. In contrast to the basic LBP using 8 pixels in a 3*3 pixel block, this generic formulation of the LBP operator puts no limitations to the size of the neighborhood or to the number of sampling points. If a point's coordinate on the circle doesn't correspond to image coordinates, the point gets interpolated, as Figure 15 shows.

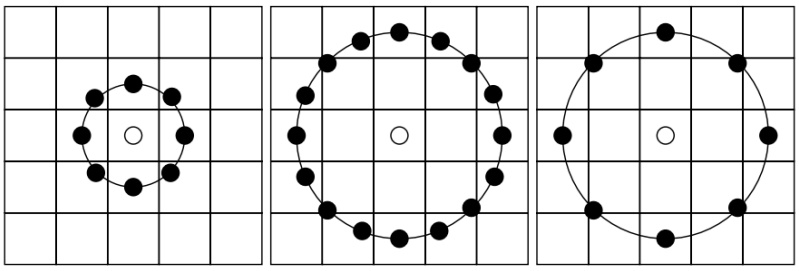


Figure 15 The circular (8,1), (16,2) and (8,2) neighborhoods. The pixel values are bilinearly interpolated whenever the sampling point is not in the center of a pixel

Consider a monochrome image I(x, y) and let gc denote the gray level of an arbitrary pixel (x, y), i.e. gc = I(x, y); gp denote the gray value of a sampling point in an evenly spaced circular neighborhood of P sampling points and radius R around point (x, y):

$$g_p = I(x_p, y_p), \quad p = 0, \ldots, P-1 \quad \text{and}$$
$$x_p = x + R\cos(2\pi p/P),$$
$$y_p = y - R\sin(2\pi p/P).$$

$$\mathrm{LBP}_{P,R}(x_c, y_c) = \sum_{p=0}^{P-1} s(g_p - g_c)2^p. \qquad [17]$$

where s(z) is the thresholding (step) function:

$$s(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0. \end{cases}$$

The derivation of the above generic LBP follows that of [20][21][22]. OpenCV supports LBP feature-based cascade classifier training by itself so the derivation details is not discussed here.

4.2.6 LBP feature-based cascade classifier training

In cascade training, as stated above, the resultant classifier consists of several simpler classifiers (stages) that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed. And positive and negative (background) samples are still needed for each stage.

So we use $numStages$ to denote a stages count, which a cascade classifier will have after the training; $numPose$ denotes a count of positive samples which is used to train each stage (is is NOT a count of all samples in vec-file.); and $minHitRate$ is a training constraint for each stage. Suppose a positive samples subset of size $numPose$ was selected to train current i stage (i is a zero-based index). After the training of current stage, at least $minHitRate * numPose$ samples from this subset have to pass this stage, i.e. current cascade classifier with $i+1$ stages has to recognize this part ($minHitRate$) of the selected sample as positive.

If some positive samples ($falseNegativeCount$ pieces) from the set (of size c) which was used to train i-stage were recognized as negative (i.e. background) after the stage training, then $numPose - falseNegativeCount$ pieces of correctly recognized positive samples will be kept to train $i+1$-stage and $falseNegativeCount$ pieces of new positive samples (unused before) will be selected from vec-file to get a set of size $numPose$ again. Another important note is, to train next $i+1$-stage we select only the samples that are passed a current cascade classifier with $i+1$ stages. (Figure 16)

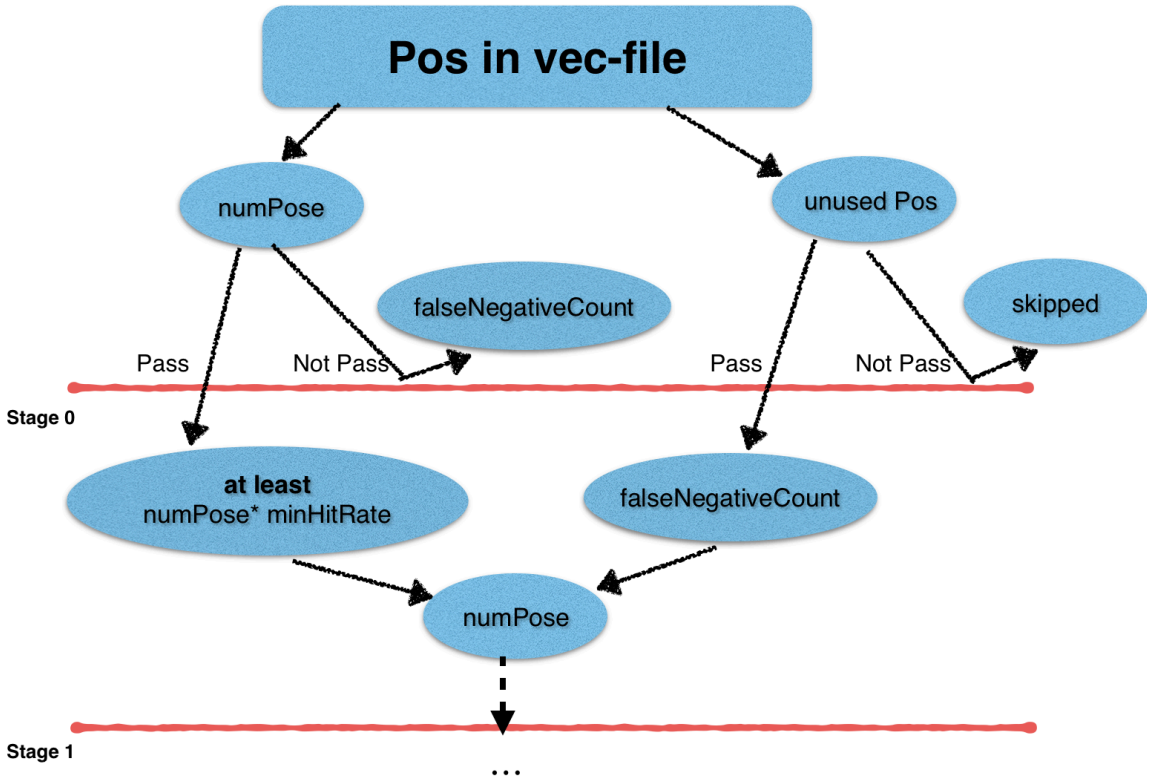


Figure 16 Cascade classifier algorithm

For the 0-stage training $numPose$ positive samples are taken from vec-file. In the worse case $(1-minHitRate)*numPose$ of these samples are recognized as negative by the cascade with 0-stage only. So in this case to get a training set of positive samples for the 1-stage training we have to select $(1-minHitRate)*numPose$ new samples from vec-file that are recognized as positive by the cascade with 0-stage only. While the selection, some new positive samples from vec-file can be recognized as background by the current cascade and these samples are skipped right away. The count of skipped sample depends on the vec-file (how different samples are in it) and other training parameters. By analogy, for each i -stage training (i =1,.. $numStages-1$) in the worse case

$(1 - minHitRate) * numPose$ new positive samples have to be selected and several positive samples will be skipped in the process. As result to train all stages $numPose + (numStages - 1) * (1 - minHitRate) * numPose + S$ positive samples are needed, where S is a count of all the skipped samples from vec-file (for all stages).

S depends on vec-file samples properties but its value can be estimated. Suppose that the samples in vec-file have the equal probabilities to be rejected by a given cascade (be recognized as background). (Of course it's not true in reality but it's suitable for the estimation of vec-file size.) In uniform case when one tries to select $falseNegativeCount$ positive samples to train i-stage one will select every new sample from vec-file with probability $minHitRate^i$. So to select $falseNegativeCount$ samples in average $falseNegativeCount / minHitRate^i$ samples will be tried from vec-file. The increasing factor is small. E.g., if i = 1, $numPose$ = 1000, $minHitRate$ = 0.99, $falseNegativeCount$ = 10 then in average ~10.1 samples need to be tried from vec-file.

The negative or background samples are created manually by cropping background images from the PhysX demo frames. Since LBP featured-based classifier can do the classification with a smaller dataset than Haar feature-based classifier does, I decrease the number of negative samples dramatically. In the end, there are only 25 negative samples with a great diversity used when training the classifier successfully. (Figure 17)

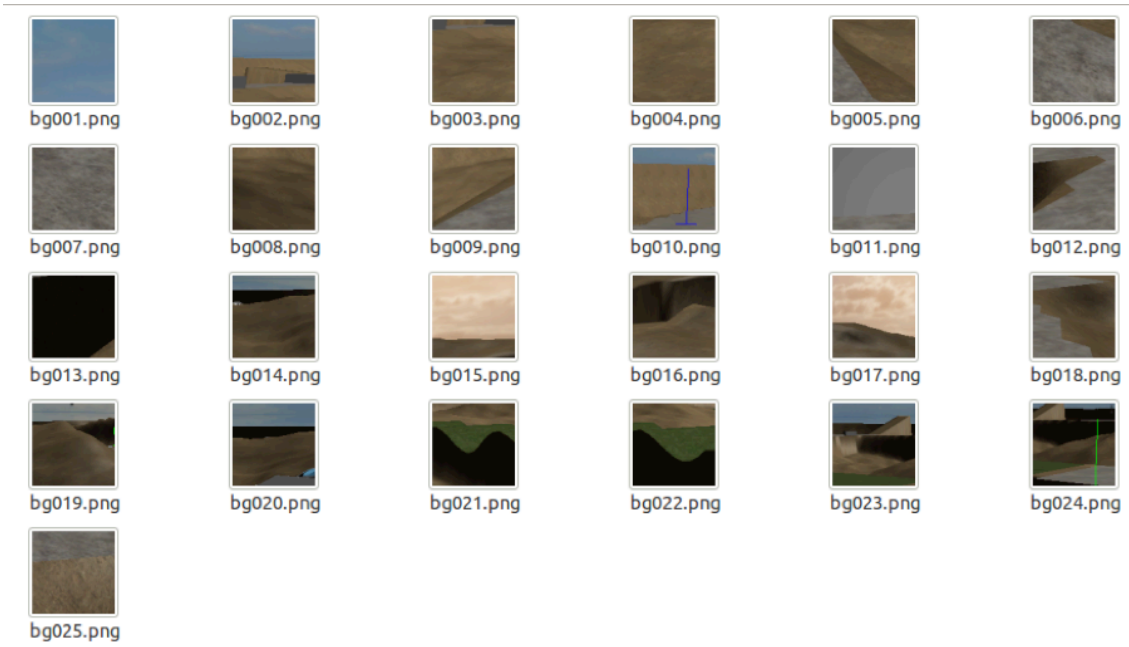


Figure 17 Negative samples used during LBP feature-based cascade classifier training

As to positive samples, they are created from a single image of the vehicle by opencv_createsamples utility. The source image is rotated randomly around all three axes. The chosen angle is limited my -max?angle. Then pixels having the intensity from [bg_color-bg_color_threshold; bg_color+bg_color_threshold] range are interpreted as transparent. White noise is added to the intensities of the foreground. If the -inv key is specified then foreground pixel intensities are inverted. If -randinv key is specified then algorithm randomly selects whether inversion should be applied to this sample. Finally, the obtained image is placed onto an arbitrary background from the background description file, resized to the desired size specified by -w and -h and stored to the vec-file, specified by the -vec command line option.

My source vehicle image is 48*48px and is scaled to 20 24*24 px positive samples. The background color in the source image is set to 0 as default and the threshold is 80 (In gray scale, 0 is black and 255 is white, shown in Figure 12.); the output file containing the positive samples for training is in vec format; the maximal intensity deviation of pixels in foreground samples is set to 40 and the maximum rotation angles are given in radians, which are 1.1, 1.1, 0.5 on x, y, z axes separately.

```
hong@hong-VirtualBox:~/workspace/levelup/bigsam$ opencv_createsamples -img back1.png -num 20 -bg ./negatives.txt -vec back1.vec -show

Info file name: (NULL)
Img file name: back1.png
Vec file name: back1.vec
BG  file name: ./negatives.txt
Num: 20
BG color: 0
BG threshold: 80
Invert: FALSE
Max intensity deviation: 40
Max x angle: 1.1
Max y angle: 1.1
Max z angle: 0.5
Show samples: TRUE
Scale applied to display : 4
```

```
Original image will be scaled to:
Width: $backgroundWidth / 24
Height: $backgroundHeight / 24
Create training samples from single image applying distortions...
Done
```

The positive samples in vec file looks like (Figure 18):

Figure 18 Positive sample examples

During LBP Classifier Training, based on the above calculation I make 5 stages for training and use 10 positive samples and 25 negative samples in each stage:

```
hong@hong-VirtualBox:~/workspace/levelup/bigsam$ opencv_traincascade -data
../bigsam -vec ./back1.vec -bg ./negatives.txt -numPos 10 -numNeg 25 -numStages 5 -w
24 -h 24 -featureType LBP


PARAMETERS:
cascadeDirName: ../bigsam
vecFileName: ./back1.vec
bgFileName: ./negatives.txt
numPos: 10
```

```
numNeg: 25
numStages: 5
precalcValBufSize[Mb] : 256
precalcIdxBufSize[Mb] : 256
stageType: BOOST
featureType: LBP
sampleWidth: 24
sampleHeight: 24
boostType: GAB
minHitRate: 0.995
maxFalseAlarmRate: 0.5
weightTrimRate: 0.95
maxDepth: 1
maxWeakCount: 100
===== TRAINING 0-stage =====
<BEGIN
POS count : consumed   10 : 10
NEG count : acceptanceRatio    25 : 1
Precalculation time: 0
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|   1|       1 |        0|
```

```
+----+---------+---------+
END>
Training until now has taken 0 days 0 hours 0 minutes 0 seconds.
===== TRAINING 1-stage =====
<BEGIN
POS count : consumed   10 : 10
NEG count : acceptanceRatio    25 : 0.543478
Precalculation time: 0
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|    1|          1|          0|
+----+---------+---------+
END>
Training until now has taken 0 days 0 hours 0 minutes 0 seconds.
===== TRAINING 2-stage =====
<BEGIN
POS count : consumed   10 : 10
NEG count : acceptanceRatio    25 : 0.675676
Precalculation time: 0
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
```

```
|    1|          1|          0|
+----+---------+---------+
END>
Training until now has taken 0 days 0 hours 0 minutes 0 seconds.
===== TRAINING 3-stage =====
<BEGIN
POS count : consumed   10 : 10
NEG count : acceptanceRatio    25 : 0.271739
Precalculation time: 0
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|    1|          1|          0|
+----+---------+---------+
END> Training until now has taken 0 days 0 hours 0 minutes 0 seconds.
===== TRAINING 4-stage =====
<BEGIN
POS count : consumed   10 : 10
NEG count : acceptanceRatio    25 : 0.290698
Precalculation time: 0
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
```

```
|    1|          1|         0|
+----+---------+---------+
END>
Training until now has taken 0 days 0 hours 0 minutes 0 seconds.
```

4.2.7 LBP Feature-based Cascade Classifier Recognition

In detection session the input frame is converted into gray scale first. Then opencv uses a detectMultiScale class to detect vehicles of different sized in the input image. At each image scale the image size is reduced with a factor of 1.1, and it can detect a vehicle with a size from 200*200 to 250 *250 px.

```
…
   cvtColor( frame, frame_gray, COLOR_BGR2GRAY );
   equalizeHist( frame_gray, frame_gray );
       vehicle_cascade.detectMultiScale( frame_gray, vehicles, 1.1, 7, 0, Size(200,
200),Size(250,250) );
…
```

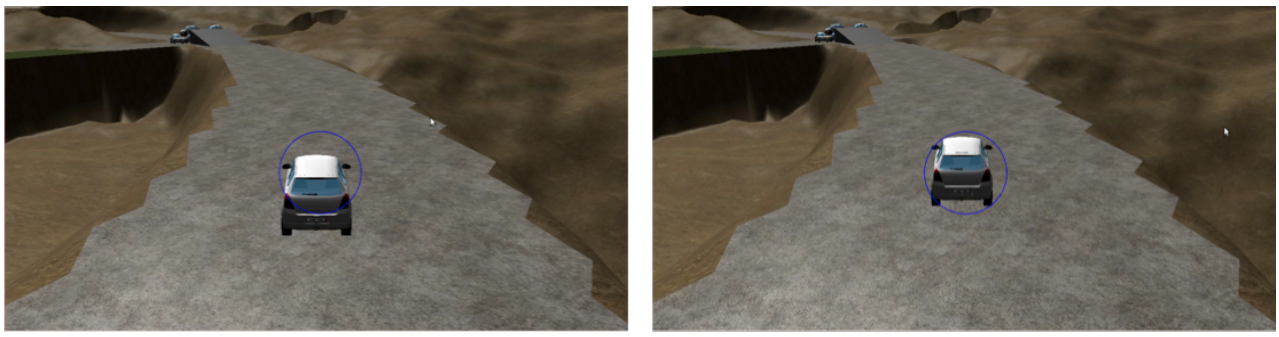


Figure 19 Recognition result of LBP feature-based cascade classifier

The classifiers discussed above are all from the back view. Generally in a multi-view recognition approach some method shall be used to deal with the consistent regions. A good method for that was developed by Scott Helmer of University of British Columbia [16]. There are two steps in their method: the first step is a sampling procedure that draws a finite set of candidate 3D locations in order to avoid the high computational cost of considering every potential location; the second step scores these potential locations based on how well they explain the outputs of the image-based classifier in all available view points.

However, in this project the target car is symmetric and each view has similar windows and textures, a decent cascade classifier from only one view works well for recognition from other views. So a classifier-merging algorithm is not included here.

4.3 Velocity Finding

4.3.1 Finding the Velocity of the Real Vehicle

After the recognition of the vehicle in the first frame of the video input, the velocity of the vehicle is found by comparing the position of the vehicle in two or more images. The distance traveled is divided by the time between the images to give the approximate velocity.

After getting the velocity through the first two or more frames the simulated car in the synthetic world is driven as same velocity. The velocity of the simulated car keeps constant till the MMD finds there are differences between the two worlds, which means the car in the real world has changed its velocity. With the help of MMD, the system registers the car and calls the velocity finding process again as stated above.

But through detection and recognition part, the distance of the car between frames we got is in two-dimensional level. It is necessary to translate the length in two-dimensional space into the real distance traveled in the stereo world. To do the calculation, there is a hypothesis that the car and the camera from which we look at the PhysX world can be seen as a particle.

Assume the car is moving forward along the road, from the back view we can see (Figure 20):

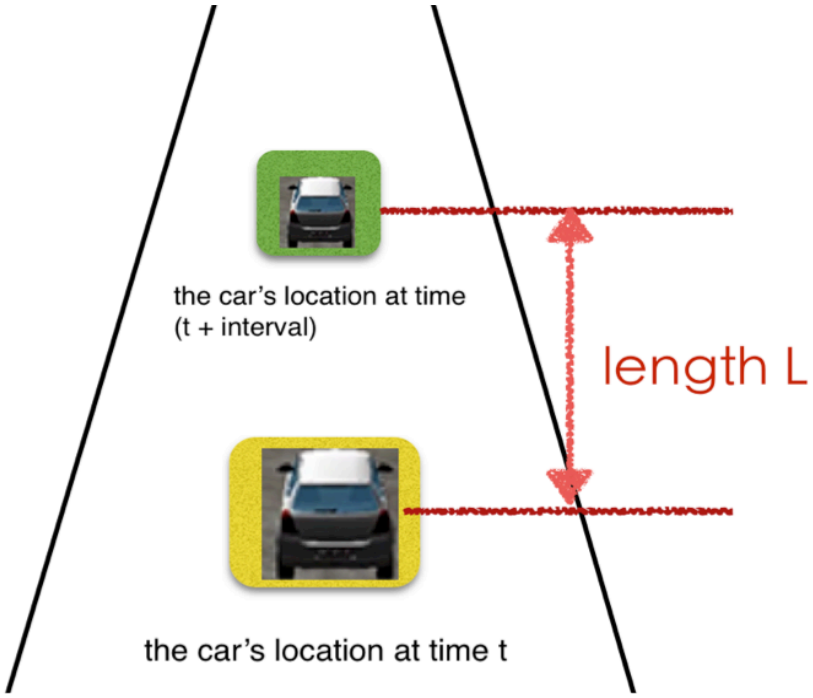


Figure 20 Geometric model of the vehicle's movement – back view

From the side view (Figure 21):

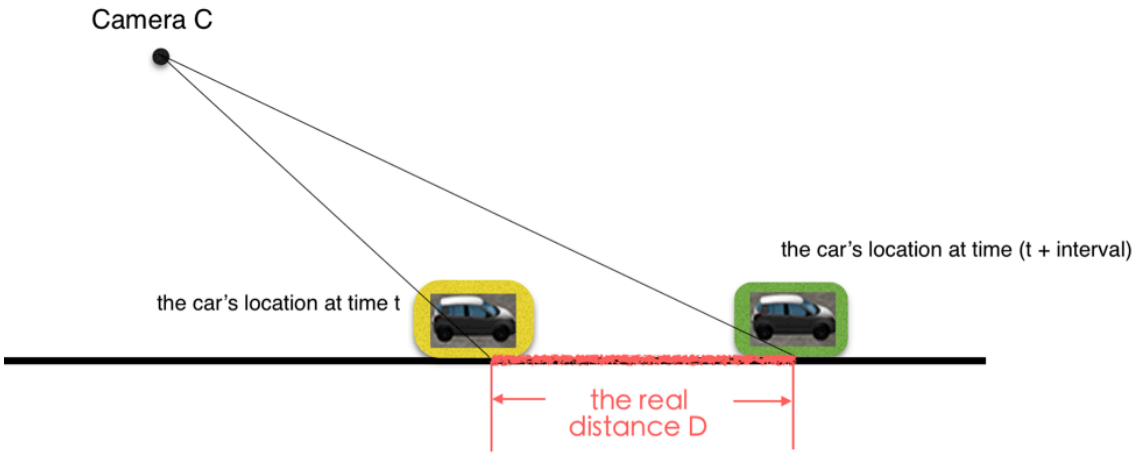


Figure 21 Geometric model of the vehicle's movement – side view

The relationship between L and D is shown in Figure 22:

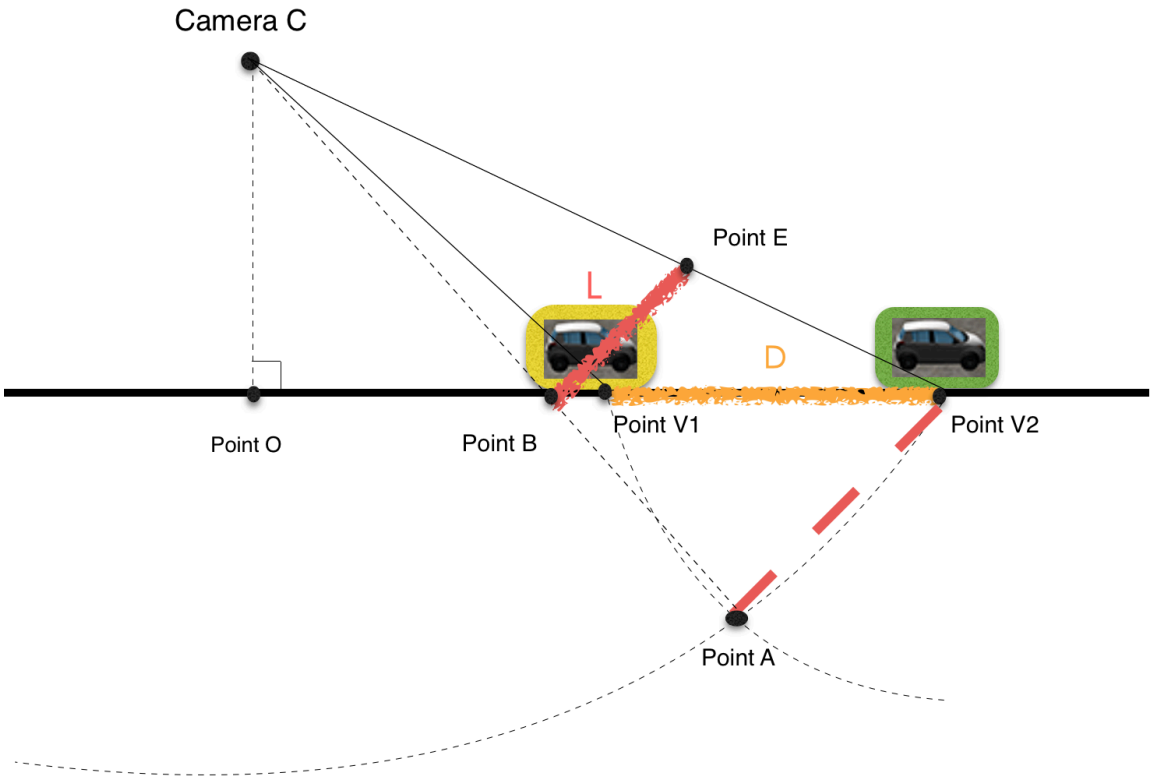


Figure 22 The distance between frames L and the real movement D

To find the relationship between L and D, select a point A, which makes $|AV_2|=|V_1V_2|$, $|AC|=|CV_2|$; so it can be considered that we rotate V_1V_2 to AV_2's position to adjust the

angle of view. However the light from point A hits the ground at point B, so from the camera's view the distance of the car between two frames is $|BE|$, where $BE//AV_2$.

To calculate the real distance D more conditions are needed than the length of L. Since the real world demo is simulated by PhysX manually, the height of camera $|CO|$ and the car's distance from camera at the start point in the first frame $|OV_1|$ can be known. Also, the velocity finding process is called by system whenever the MMD finds there is a change in the velocity of the real car.

So if D is calculated in the first frame, than the real displacements in some certain frame can be derived from the displacement that has been made. But after calculation about the first frame, I got equations about D^2 and $|CB|^4$. To keep things simplest at this initial step, L is used as an approximate value of the real distance D to calculate an approximate velocity.

In the cases that the car is turning, the angle of view doesn't affect the calculation of the turning angle.

As stated above, when MMD finds differences between the real world and synthetic world, it helps the system register the object that causes the difference. Then the system calls the velocity finding process to modify the velocity of the corresponding object in the synthetic world. In this project, assuming MMD is working, an approximate velocity is calculated through the first two frames. If MMD finds differences in the following frames, we can imagine the velocity finding process will work again.

Since my system is run in a virtual box which makes the PhysX world react slow, the car in the video input doesn't move fast at the beginning. However, the calculation starts in the middle of the video, the car is relative small in the image. As a result the detection window has to be small, in which case the computing speed will drop down dramatically. To improve the computing performance, the displacement is calculated every ten frames, starting from the first frame (Figure 23).

```
hong@hong-VirtualBox:~/workspace/DetectionFromVideo$ ./detectionFromVideo
reading the next frame..
Frame 0
ready for detection..
detecting..
drawing the result..
the car is at (555, 248)
the displacement is (0, 0)
the velocity is (0.000000, 0.000000, 0.000000)
reading the next frame..
reading the next frame..
reading the next frame..
reading the next frame..
reading the next frame..
reading the next frame..
reading the next frame..
reading the next frame..
reading the next frame..
reading the next frame..
Frame 10
ready for detection..
detecting..
drawing the result..
the car is at (559, 246)
the displacement is (4, -2)
the velocity is (6.000000, -3.000000, 0.000000)
```

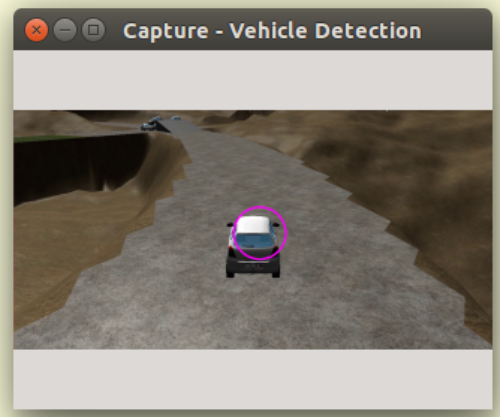


Figure 23 Velocity finding from the first two frames

As shown in Figure 23, the detectMultiScale class of OpenCV makes the coordinate of detected objects integer, which affects the accuracy further. And for the time calculation, it is 15 frames per second (Figure 24) when the video is recorded. So the sampling interval is 10/15 = 0.666666 second.

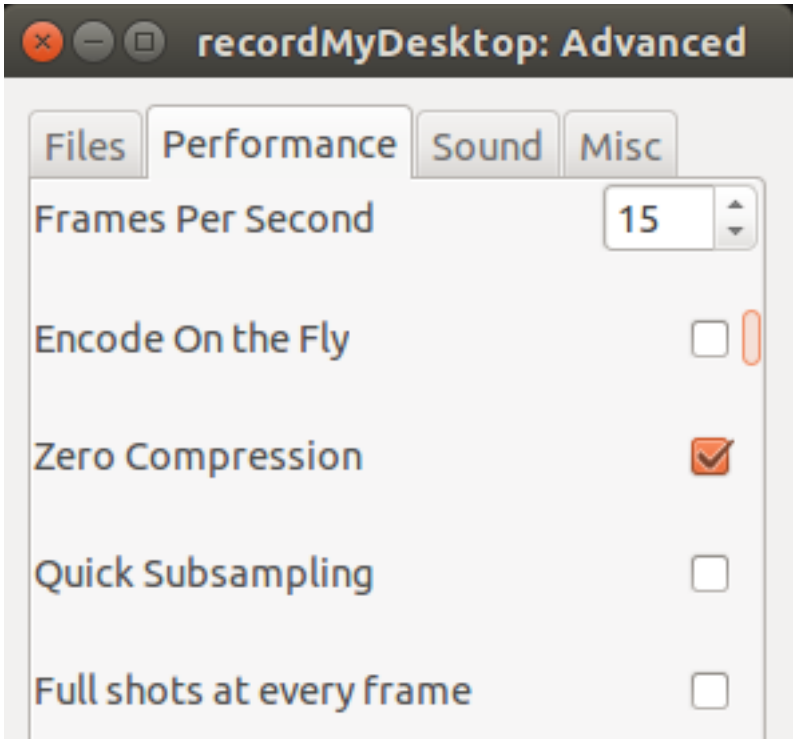


Figure 24 Sampling interval for video input

4.4 Vehicle Driving in the Rendered World

Once the approximate velocity has been computed in the real world, the velocity of the rendered vehicle is set to that value, causing the rendered car to move approximately the same path as the real car.

4.5 Connection between Real World and Rendered World

For connection between the real world and the render world in real-time, opening up socket is not a solution since PhysX needs every thread to be returned between each frame. So a program with a timestamp is needed for running openCV into a file and reading the file with PhysX to mitigate the race condition.

Chapter 5

Conclusions

We are building a 3D visual system in real time to the future.

A PhysX demo is build to represent the real world and the real vehicle; then a detection and recognition process is undertaken to find the specific vehicle using cascade classifiers. Cascade classification is a type of machine learning in artificial intelligence area and the proper algorithm and parameters depend greatly on the specific project and its dataset. In this paper Haar feature-based and Local Binary Pattern feature-based classifier are discussed and compared. And an approximate velocity of the real vehicle is calculated for driving the rendered vehicle.

This is initial work on this project. Much remains to be done, including using videos of real cars, and using stereo to find their distances, and adding multiple moving vehicles.

References

[1] D. Paul Benjamin, Damian Lyons, and Christopher Funk, "A Cognitive Approach to Vision for a Mobile Robot", SPIE Conference on Multisensor, Multisource Information Fusion: Architectures, Algorithms, and Applications, April 2013.

[2] Damian Lyons, Paramesh Nirmal and D. Paul Benjamin, "Navigation of Uncertain Terrain by Fusion of Information from Real and Synthetic Imagery", SPIE Conference on Multisensor, Multisource Information Fusion: Architectures, Algorithms, and Applications, April 2012.

[3] D. Paul Benjamin, Damian Lyons, John V. Monaco, Yixia Lin, and Christopher Funk, "Using a Virtual World for Robot Planning", SPIE Conference on Multisensor, Multisource Information Fusion: Architectures, Algorithms, and Applications, April 2012.

[4] Benjamin, D. Paul, Damian Lyons and Deryle Lonsdale, " Cognitive Robots: Integrating Perception, Action and Problem Solving in Behavior-Based Robots", AAMAS-2004 Proceedings, pp. 1308-1309, (2004).

[5] Laird, J.E., Newell, A. and Rosenbloom, P.S., “Soar: An Architecture for General Intelligence”, Artificial Intelligence 33, pp.1-64, (1987).

[6] Newell, Allen, “Unified Theories of Cognition”, Harvard University Press, Cambridge, Massachusetts, (1990).

[7] Blake, A. and Yuille, A., eds, “Active Vision”, MIT Press, Cambridge, MA, (1992).

[8] Lyons, D.M. and Hendriks, A., “Exploiting Patterns of Interaction to Select Reactions”, Special Issue on Computational Theories of Interaction, Artificial Intelligence 73, (1995), pp.117-148.

[9] Lyons, D.M., “Representing and Analysing Action Plans as Networks of Concurrent Processes”, IEEE Transactions on Robotics and Automation, June (1993).

[10] Lyons, D.M. and Arbib, M.A., “A Formal Model of Computation for Sensory-based Robotics”, IEEE Transactions on Robotics and Automation 5(3), Jun. (1989).

[11] Nicolescu, M. and Mataric, M>, “Extending Behavior-based System Capabilities Using an Abstract Behavior Representation”, Working Notes of the AAAI Fall

Symposium on Parallel Cognition, pages 27-34, North Falmouth, MA, November 3-5, (2000).

[12] Rosenbloom, P.S., Johnson, W.L., Jones, R.M., Koss, F., Laird, J.E., Lehman, J.F., Rubinoff, R., Schwamb, K.B., and Tambe, M., "Intelligent Automated Agents for Tactical Air Simulation: A Progress Report", Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation, pp.69-78, (1994).

[13] PhysX official documentation, http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Index.html

[14] OpenCV 2.4 documentation, "Haar Feature-based Cascade Classifier for Object Detection", http://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html

[15] OpenCV 3.0.0 –dev documentation, "Face Detection using Haar Cascades", http://docs.opencv.org/master/d7/d8b/tutorial_py_face_detection.html#gsc.tab=0

[16] Scott Helmer, David Meger, Marius Muja, James J. Little, David G. Lowe University of British Columbia, "Multiple Viewpoint Recognition and Localization", (2011)

[17] Pietikäinen, M., Hadid, A., Zhao, G., Ahonen, T., "Computer Vision Using Local Binary Patterns", Chapter 2 Local Binary Patterns for Still Images, p14-16, (2011)

[18] OpenCV 2.4 documentation, "Face Recognition with OpenCV", http://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec_tutorial.html#local-binary-patterns-histograms-in-opencv

[19] Wang, L., He, D.C.: Texture classification using texture spectrum. Pattern Recognit. 23, 905– 910 (1990)

[20] Mäenpää, T.: The local binary pattern approach to texture analysis—extensions and applications. PhD thesis, Acta Universitatis Ouluensis C 187, University of Oulu (2003)

[21] Mäenpää, T., Pietikäinen, M.: Texture analysis with local binary patterns. In: Chen, C.H., Wang, P.S.P. (eds.) Handbook of Pattern Recognition and Computer Vision, 3rd edn., pp. 197–216. World Scientific, Singapore (2005)

[22] Ojala, T., Pietikäinen, M., Mäenpää, T.: Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. IEEE Trans. Pattern Anal. Mach. Intell. 24(7), 971–987 (2002)