WORCESTER POLYTECHNIC INSTITUTE

RBE 549 COMPUTER VISION
FALL 2015
FINAL PROJECT REPORT

Dinosaur Classifier

Team Crocodile

Member	Signature	Contribution (%)
Aditya Bhat	Aditya	25%
Joseph McMahon		25%
Peng He	(Peng's verbal consent)	25%
Renato Gasoto		25%

Grading:		
	Approach	_____/ 20
	Justification	_____/ 15
	Analysis	_____/ 20
	Testing & Examples	_____/ 15
	Documentation	_____/ 15
	Presentation	_____/ 10
	Difficulty	_____/ 05
	Total	_____/100

December 21, 2015

Contents

List of Figures

Executive Summary

Planar object classification with plain texture on a changing background is a challenging task for vision systems. As the object may be of the same color as the background, a simple color blob detection is not enough for a good quality image segmentation before processing for analysis and classification. As the object doesn't have texture, the only means for its identification is by its shape.

A background removal, followed by color segmentation allows for a more precise identification of the object on the screen. Passing the resulting sub-image into a cascaded algorithm that initially classifies the object on a subgroup of similar semantic features, such as the convexity ratio, then on a feature matching algorithm, allows a precise object classification and tracking, even on a cluttered background. Figure 1 is an example of an object identified on a video stream. The green contour shows the object's convex hull.

Figure 1: Output of the classification based on convexity method on the Triceratops cutout.

1 Introduction

Object classification is one of the core problems of computer vision. It can be used as a building block for many other tasks such as localization, detection and scene understanding. Computer vision systems perform extremely well at detecting objects with a lot of features under specific lighting conditions and poses. However, these systems have a higher error during classification when the object to classify has occlusions, varying lighting conditions and fewer features.

Our work deals with the problem of classifying planar objects which have very few features for a standard feature matching algorithm. To tackle this problem, the semantic attributes like color and convexity were used to obtain the type of object. Using semantic attributes provides a high rate of success for classifying specific predefined objects. The objects classified 3 types of dinosaur cutouts which are of the same size and color.

Convexity based classification provides better results with minimal computation when compared to the keypoint matching algorithm. Although it compromises on robustness and is unable to handle occlusions, it does provide a reasonably effective first pass to classify the type of object in the frame.

To successfully classify the object based on semantic features, preprocessing the image to reduce the amount of information to classify is key. The preprocessing steps involve normalizing the image, background removal, filtering and color based segmentation.

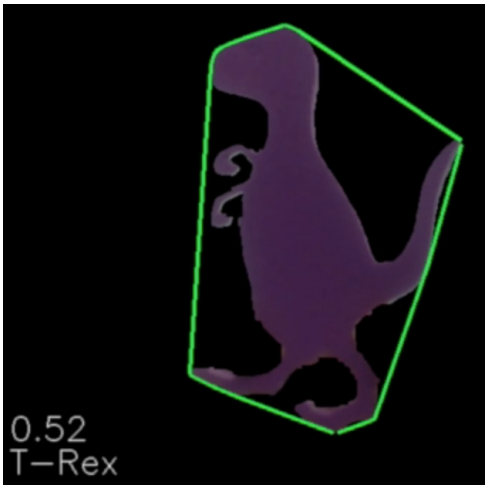


Figure 2: Classification of the object within the convex hull as a T-Rex based on the area ratio

This report shows how semantic attributes can be used for object classification. Section 2 deals with eliciting the problem statement. Next, section 3.1 deals with the preprocessing steps whose output is the input to the image classifier. In section 3.2 and 3.3 we talk about the two types of classifiers we

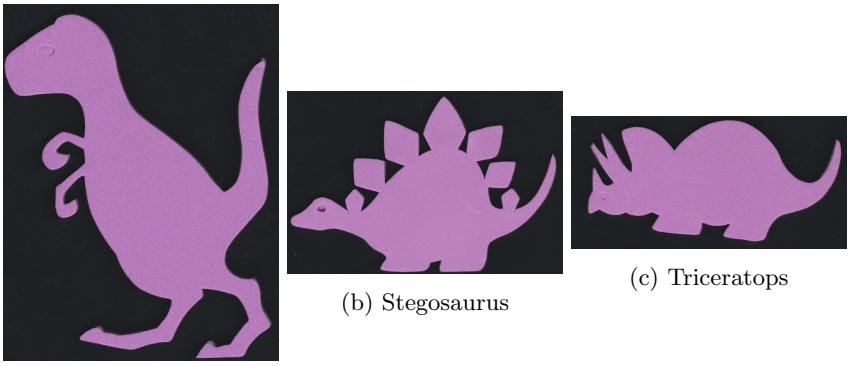
used for the purposes of classifying the planar object. The first classifier is a feature matching algorithm based on keypoints. The second algorithm uses the convexity of the outermost contour of the object to classify the type of object.

2 Problem Statement

The problem we are trying to solve is to classify a magenta shaped dinosaur cutout in a streaming set of images as one of three dinosaur cutouts. A major design assumption is that there is only one dinosaur cutout per image and that it is the largest magenta colored object in the frame. It is also assumed that the camera is stationary, there are no occlusions on the object and no obstructions between the camera and the object. We do consider the rotation of the object and scale/skew (or a perspective function on the dinosaur cutout). The system should work well under reasonable lighting conditions.

2.1 Inputs

The inputs for the system are a major factor in its design. The most notable reason being that the input had no features or variety in color and are flat objects as shown in Figure 3. The most reliable information to compare is the contour.



(a) Tyrannosaurus

(b) Stegosaurus

(c) Triceratops

Figure 3: The input to the system is an image of a dinosaur cutout. A dinosaur classifier should be robust enough to classify the dinosaur into one of these three categories given the object in a diverse set of backgrounds.

3 Approach

The required input is represented by the arrow labelled 'Image' on the left side of the system flowchart of the dinosaur Classifier shown in Figure 4. The input

can be passed as a single static frame or a stream of static frames. This allowed a webcam to be used to accept input.

The system preprocesses the webcam images and passes it to each of the two classification methods.

OpenCV 3 [1] with Python was used because of its easily accessible documentation, easy of use (we run OS X and Ubuntu), and speed. For example, OpenCV's image processing algorithms are much faster than MATLAB's built in image processing.

```
import cv2
import numpy as np

c = cv2.VideoCapture(0)
```

Listing 1: Webcam frame

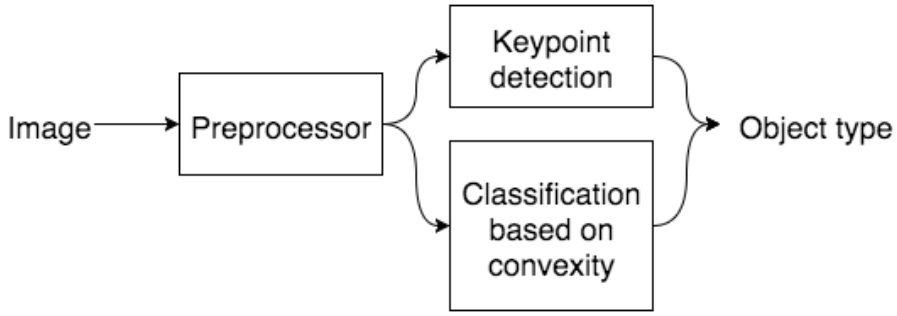


Figure 4: System Flowchart

3.1 Preprocessor

Figure 5 is the preprocessor flowchart. The preprocessor is required to separate a magenta blob from the image background. It also deals with different lighting by normalizing the blob first. Then the background is removed and color based segmentation is used to retrieve the shape of the object.

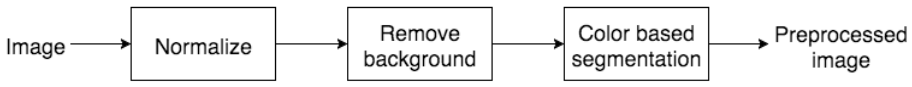


Figure 5: Preprocessor Flowchart

3.1.1 Normalize

The image must be normalized to compensate for various changes in lighting. Normalizing makes different tones of magenta appear as a more consistent share of magenta.

```
def normalized(b,g,r):
    clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(3,3))
    b1 = clahe.apply(b)
    g1 = clahe.apply(g)
    r1 = clahe.apply(r)
    return b1,g1,r1.
```

Listing 2: Normalize RGB Values

3.1.2 Remove background

In order to remove the background, first we let the camera load a series of frames to get the static background by applying a weighted sum of the captured frames.

```
def preprocessbackground(c, f, avg2):
    t=0
    gray=0
    while(t<50):
        _,f = c.read()
        # Get average of background with weight of 0.01 for new
    images
        cv2.accumulateWeighted(f,avg2,0.01)
        t+=1
    #normalize back to 8-bit values
    res2 = cv2.convertScaleAbs(avg2)
    #Remove noise with median blur
    res2=cv2.medianBlur(res2,5)

    return res2
```

Listing 3: Pre-process the background

The averaged image is then used to create a mask out of the difference between the background and the new frames.

```
def foregroundMask(color, background):
    b = cv2.split(background)
    im = cv2.split(color)
    mask = None
    out = None
    # For each channel (B-G-R) in the image:
    for i in range(len(b)):
        # Channel c gets a median blur of same size as the kernel
        # used in the preprocessed background
        c = cv2.medianBlur(im[i],5)
        # Gets difference between background and image (chanel-wise
    )
        imgs=cv2.absdiff(im[i], b[i])
        # Run a Binary OTSU threshold to get what is background and
     what is
        # not based on the absolute difference.
        (thresh, im_bw) = cv2.threshold(imgs, 250, 255, cv2.
    THRESH_BINARY | cv2.THRESH_OTSU)

        # Close the Thresholded image to remove
        # small noises in the threshold
        # Reopen the Threshold
```

```
        im_bw=cv2.morphologyEx(im_bw, cv2.MORPH_CLOSE, np.ones
    ((5,5), np.uint8))
        im_bw=cv2.morphologyEx(im_bw, cv2.MORPH_OPEN, np.ones((5,5)
    , np.uint8))
        # Merge Channel Masks
        if mask is None:
            # If it's the first channel being processed
            mask = im_bw
        else:
            # Else, merge channels by running an OR
            # (wherever it's above the threshold, it's an object)
            mask = cv2.bitwise_or(mask,im_bw)
    return mask
```

Listing 4: Pre-process the background

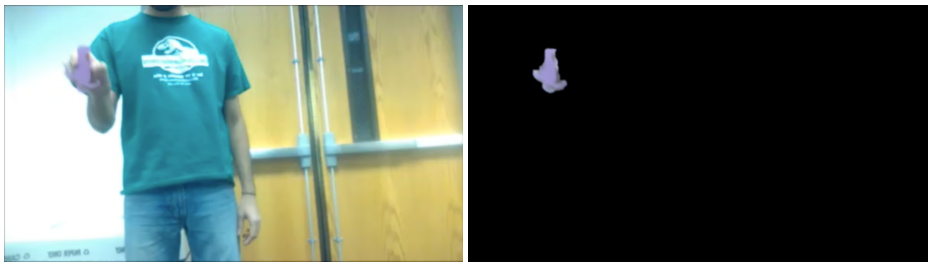
The mask is applied to each image channel individually, which is then normalized.

```
background=BackgroundRemoval.preprocessbackground(c, f, avg2)
# get new frame
_,f = c.read()
while True:
    _,f = c.read()
    o = copy.deepcopy(f)
    # gets mask for background removal
    mask=BackgroundRemoval.foregroundMask(f, background)
    # Split channels to remove backround and normalize color
    b,g,r = cv2.split(f)
    # Applies mask to each channel to remove background
    nb=np.minimum(mask, b)
    ng=np.minimum(mask, g)
    nr=np.minimum

    # normalize color
    bn,gn,rn = normalized(ng,nr,ng)

    # return to color image
    backgroundRemovedImage=cv2.merge((nb, ng, nr))
```

Listing 5: Remove the background

3.1.3 Color based segmentation

Color based segmentation is the process of selecting desired areas of the image based on the color represented [2]. As the image contains noise, the perceived color on each pixel may be distorted, so instead of having a single point on color space, it is more desired to have a range of colors that is accepted. To better represent the desired range, the color space was converted from BGR (Blue, Green, Red) to HSV (Hue, Saturation, Value). On the converted image, a threshold is applied such that a mask is created to discard every pixel that is not within the determined range. Figures 6a and 6b show the result of the color based segmentation.



(a) Input for Color segmentation (b) Output of Color Segmantation

Figure 6: Color Segmentation.

```
class ColorSegmenter:
    #Thresholds for Magenta blob
    lower_magenta = np.array([125,30,30])
    upper_magenta = np.array([160,255,255])

    @staticmethod
    def getMagentaBlob(bgrimg):
        #Convert image space to HSV
        hsvimg = cv2.cvtColor(bgrimg,cv2.COLOR_BGR2HSV)
        # Apply Bilateral Filter to remove noise
        blur = cv2.bilateralFilter(hsvimg,9,75,75)
        # Threshold the HSV image to get only magenta
        mask = cv2.inRange(blur, ColorSegmenter.lower_magenta,
    ColorSegmenter.upper_magenta)
        #Clean residual noise eroding, opening and closing image
        mask = cv2.erode(mask,np.ones((1,1),np.uint8),iterations =
    5)
        mask = cv2.morphologyEx(mask,cv2.MORPH_OPEN,np.ones((4,4),
    np.uint8))
        mask = cv2.morphologyEx(mask,cv2.MORPH_CLOSE,np.ones
    ((15,15),np.uint8))

        # Apply mask to original image
        res = cv2.bitwise_and(bgrimg,bgrimg, mask= mask)
        return res
```

Listing 6: Color segmentation

3.2 Classification based on features

Classification based on features is a common approach to object detection. There are many well known feature detection methods and algorithms. Most of them compute small gradients across and object and remember the significant ones.

3.2.1 Key Point Description

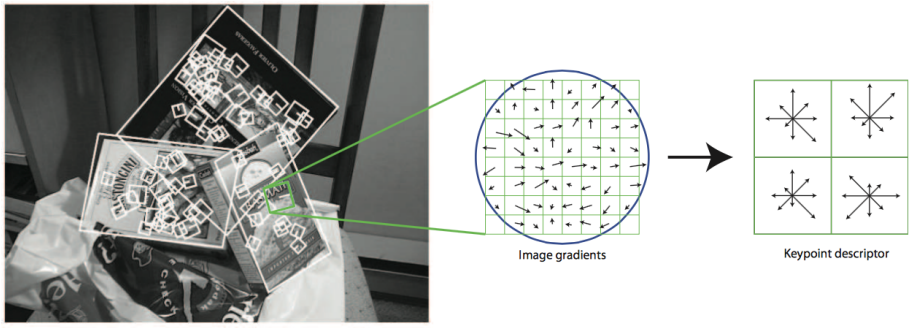
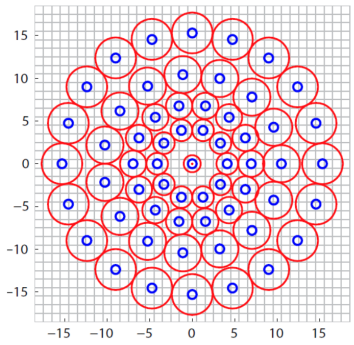


Figure 7: Use SIFT feature descriptor to create key points

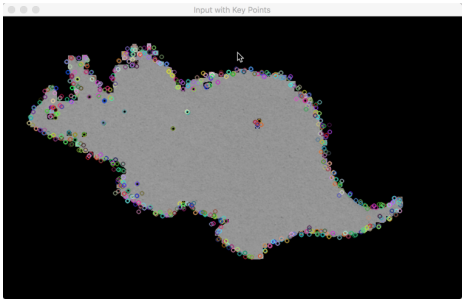
The feature description is nothing else than some gradients. For example, in the SIFT method, it will calculate the gradients on the image and base on a scale parameter that user sets to gather all the gradient vector within this region then make it a "Feature Vector" as shown in Figure7 [3]. By finding these feature vectors, we could tell if a query image will "matches" one of the image sample in our database.

In this project, we decide to choose two feature description methods: BRISK and AKAZE. This is due to our queries are planner objects which is lacking of 3D features on the object. These two methods are good at creating descriptors on a binary image or gray scale image. We will briefly talk about how they work.

The BRISK use a pattern to calculate the image intensity around a key-point's neighborhood. The red circle in Figure8a indicates the size of smoothing kernel. It use brightness and pixel orientation to create feature descriptor on a binary image[3].



(a) BRISK detecting pattern



(b) AKAZE interest points

Figure 8: Some feature generating methods

The A-KAZE is short for accelerated KAZE feature. It is good for creating feature vectors in a nonlinear scale space. According to the paper[4], they use AOS and variable conductance diffusion to build a maximum evolution time nonlinear scale space. Then exhibiting a maxima of the Hessian response through the nonlinear scale space, 2D features will be detected. Finally, the main orientation and rotation invariant descriptor in computed by first order derivatives of the 2D features. We apply the AKAZE feature vector generating method on our sample object, shown in Figure.8b.

```
def describe(self, image):
    # the A-KAZE is offered in CV 3.0
    # to use BRISK as well, replace with code cv2.BRISK_create()
    descriptor = cv2.AKAZE_create()

    # make image gray scaled for extract 2D features.
    self.gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # compute the features and return the key points with its
    descriptor
    (self.kps, descs) = descriptor.detectAndCompute(self.gray, None
    )

    # only take the (x,y) attribute and form a NP array.
    kps = np.float32([kp.pt for kp in self.kps])

    return (kps, descs)
```

Listing 7: Feature Creation Using A-KAZE

The code 7 explains how to use openCV functions to create features. Notice that there is a special version of key point descriptor function which we made for the query image. They are basically the same, the query one contains raw key point information for drawing the contours in results display part.

3.2.2 Sample Database

We use a "smart" way to store our samples in a .csv file as the database. Once the program is initiated, it will read the .csv file and retrieve sample image paths along with a given sample ID name. After having these information, the program call our feature describing methods to create feature vectors on the samples and same them in the memory for next use.

By writing code this way, we can easily modify our sample database without a line changed in the code. Code 8

```
ap = argparse.ArgumentParser()
# requires user to give the path where our .csv file is
ap.add_argument("-d", "--db", required = True, help = "path to the
    object information database csv file")
# parseing the arguments
args = vars(ap.parse_args())
# create an empty array for the sample image paths
db = {}
# read the .csv file and get sample image paths in
for line in csv.reader(open(args["db"])):
    db[line[0]] = line[1:]
```

Listing 8: Creating database from command line arguments

3.2.3 Matching Queries

When all the steps above are done, we can finally recognize our query image by comparing our samples to it(notice the comparing direction). A K-NN network is applied here based on a method that can measure the distance between two images which is called "BruteForce-Hamming" method [5].

```
def match(self, kpsA, featuresA, kpsB, featuresB):
        matcher = cv2.DescriptorMatcher_create(self.distanceMethod)
        rawMatches = matcher.knnMatch(featuresB, featuresA, 2)
        matches = []
        # apply a ratio to get better results
        for m in rawMatches:
            if len(m) == 2 and m[0].distance < m[1].distance * self
    .ratio:
                matches.append((m[0].trainIdx, m[0].queryIdx))
```

Listing 9: Using KNN to match out the query

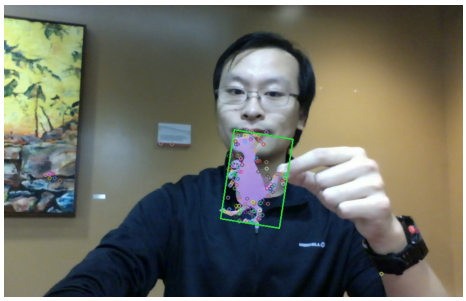
Notice that, in this project, we are classifying the query to its corresponding sample, so the neighbor number argument we give to KNN is only '2'; besides this, we apply a 0.7 ratio on the distance to make sure that we get a better result. However, due to the problem that these planner sample image are lacking keypoints in their center area. A high ratio like 0.7 is not always returning a accurate match.

```
if len(matches) > self.minMatches:
            ptsA = np.float32([kpsA[i] for (i, _) in matches])
            ptsB = np.float32([kpsB[j] for (_, j) in matches])
            (_, status) = cv2.findHomography(ptsA, ptsB, cv2.RANSAC
    , 4.0)
```

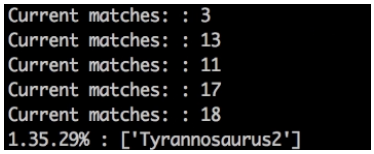
```
5
6                 return float(status.sum()) / status.size
7         db[line[0]] = line[1:]
8 # return -1 if no possible matches found
9 return -1.0
```

Listing 10: Classify based on area ratio

Now supposing we already have some good matches, the last thing we need to do is find the transformation between two matched keypoints which are in the query image and our sample image. We need at least four matched pairs to do this. The openCV has done this for us, just call the function shown in the code with a finding method RANSAC will give us the answer.



(a) A-KAZE Matching from a Web-cam



(b) A-KAZE matching results

Figure 9: Real time A-KAZE Key point matching to classify a query image from a Web-cam streaming input.

3.2.4 More Difficult Queries

We tested a more complex scenario where the queries are scaled, shrunk vertically or horizontally, and rotated as shown in Figure 10. A-KAZE keypoint matching method succesfully identifies all of the manipulated queries except for the horizontally shrunk volcano.

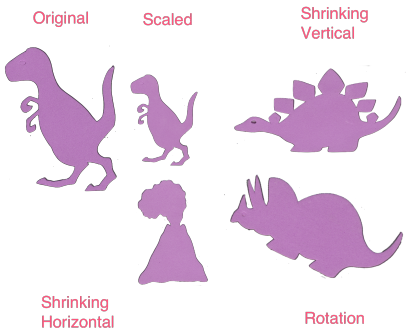


Figure 10: Manipulated Query Samples.

3.3 Classification based on convexity

3.3.1 Convexity

A two dimensional shape, represented by a continuous variables x y or discrete set of x y pixels is convex if the line drawn between any two points do not cross the border. Otherwise, it is concave - the object's shape looks like it has at least one dent in it, even though it may be a small dent [6].

3.3.2 Convex Hull

The convex hull B of a set of concave points A, is the minimal convex set where $A \subseteq B$. If the set A is already convex, $A = B$.

A simpler analogy is that the convex hull is the shape produced by putting a rubber band around the object. After obtaining the color segmented blob, Canny Edge detection was used to produce a contour. The contour is closed and its area calculated. And then the blob's convex hull is created and drawn, and its area calculated.

```
# Canny Edge Detection
objectdetection = cv2.cvtColor(res, cv2.COLOR_BGR2GRAY)
edged = cv2.Canny(objectdetection, 100, 250)
# Close edges
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (7, 7))
closed = cv2.morphologyEx(edged, cv2.MORPH_CLOSE, kernel)
# Get Contours out of edges
contours,hierarchy = cv2.findContours(closed,2,1)
areaContours=[]
res2 = copy.deepcopy(res)
# For each contour in image:
for cnt in contours:
    # Only if there are 2 contours or something
    area = cv2.contourArea(cnt)
    # Get Convex hull
    hull = cv2.convexHull(cnt,returnPoints = False)
    if(len(hull)>3 and len(cnt)>3):
        # Get Convexity Defects (where the contour is not convex)
```

```
        defects = cv2.convexityDefects(cnt,hull)
        if defects!=None:
            # Get contours areas
            areaContours=areaContours+[(area, len(defects),defects,
    cnt)]
            if area>1000: # Filter out small objects (noise)
                for i in range(defects.shape[0]):
                    s,e,f,d = defects[i,0]
                    start = tuple(cnt[s][0])
                    end = tuple(cnt[e][0])
                    cv2.line(res2,start,end,[0,255,0],2) # Draw
    convex Hull
# Now that convex hull is drawn on image, obtain area of convex
    hull as well
forareaofhull=cv2.cvtColor(res2, cv2.COLOR_BGR2GRAY)
contoursforhull,hierarchy = cv2.findContours(forareaofhull,2,1)
areaWithHull=[]
# For every contours in Hull group
for cntforhull in contoursforhull:
    # Only if there are 2 contours or something
    area = cv2.contourArea(cntforhull)
    # Gets list of convex hulls Areas
    areaWithHull=areaWithHull+[area]
# Select biggest Area for Hull
if len(areaContours)>0:
    if max(areaWithHull)>1000:
        # Draw Contour in output image
        defects = max(areaContours,key=itemgetter(1))
        for i in range(defects[2].shape[0]):
            s,e,f,d = defects[2][i,0]
            start = tuple(defects[3][s][0])
            end = tuple(defects[3][e][0])
            cv2.line(res,start,end,[0,255,0],2)
        # Calculate ratio between Hull and shape area
        ratioOfAreas=max(areaContours,key=itemgetter(1))[0]/float(
    max(areaWithHull))
```

Listing 11: Limit the results to our configuration

(a) Tyrannosaurus Convex Hull.

(b) Stegosaurus Convex Hull
.

(c) Triceratops Convex Hull.

Figure 11: The convex hull is the shape produced by stretching a metaphorical rubber band around an object.

3.3.3 Area Ratio Detection

The ratio of the area A_R of the object's image area A over the area of the object's convex hull A_{CH} remains constant. This is intuitive - the size of an object is inversely related to its distance from the camera, so when the object is skewed, scaled, or projected, both the contour and its convex hull will be proportionally transformed. So when two areas are divided by each other, this inverse relationship cancels out and a constant value remains. The area ratios for the three dinosaurs are shown in Table 1.

$$A_R = \frac{A}{A_{CH}}$$

Dinosaur	Area Ratio
Trex	0.50-0.65
Steg	0.70-0.77
Tric	0.78-0.84

Table 1: Area Ratios of the Dinosaurs.

```
def classifyObject(ratioOfAreas):
    if ratioOfAreas>0.5 and ratioOfAreas<0.65:
        return 'T-Rex'
    elif ratioOfAreas>0.70 and ratioOfAreas<0.77:
        return 'Stegosaurus'
    elif ratioOfAreas>0.78 and ratioOfAreas<0.84:
        return 'Triceratops'
    elif ratioOfAreas>0.86:
        return 'Volcano'
```

```
10 return ''
```

Listing 12: Limit the results to our configuration

4 Justification

The justification of the primary Dinosaur Classifier is in the system's robust detection and classification, and its minimal overhead and fast runtime. The criteria analyzed are: accuracy, robustness, speed, and range as discussed in the Analysis section.

The required input was a major factor in the justification of the system design. The dinosaur cutouts realistically only have two pieces of information: their magenta color and their shape as shown in Figure 12. So in general, the system's methods used the magenta color to detect a dinosaur color, and then used the shape to classify it.

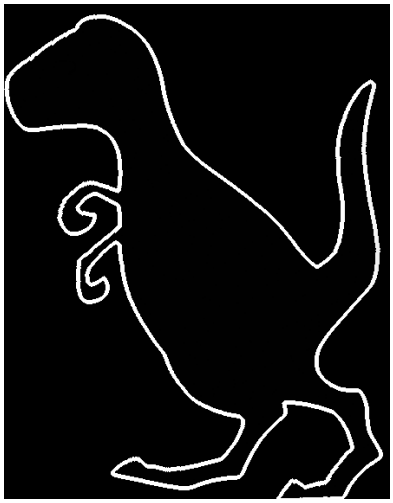


Figure 12: Contour of the Trex dinosaur cutout.

The system's software components were subtly optimized for these factors. However, the range and robustness could still be improved at the cost of time complexity, and perhaps accuracy. This is because the system will take longer to compute more robust results across a wider range, which could result in more false positives.

5 Results & Analysis

All of the dinosaurs were successfully detected and classified. There were a few false positives, especially between classifying the stegosaurs and triceratops

because their area ratios and shapes are similiar. The classification based on convexity was particularly interesting because it is still an experimental method. The keypoint matching worked well, but the feature vectors the algorithms generate are limited because the dinosaur cutouts are so similar to one another as shown in Figure 13.

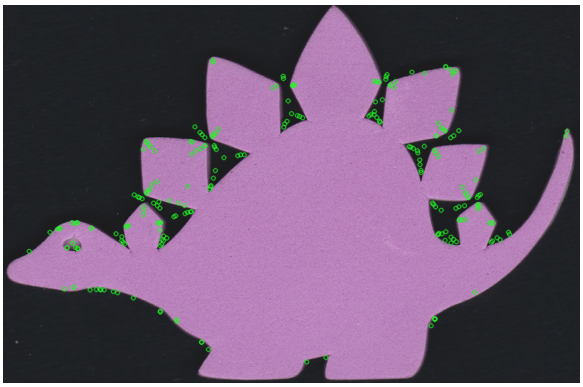


Figure 13: AKAZE keypoints of the Stegosaurus.

The results were analyzed across four factors: accuracy, robustness, speed, and range. It was determined that the feature matching was not suitable for classifying the dinosaur cutout images.

5.1 Accuracy

The system accurately classifies the three dinosaurs. It has trouble classifying them when a dinosaur is rotated more than 15 degrees parallel to the camera. The system was accurate within the assumed range of the dinosaur cutout being roughly planar with the camera plane, and within a distance of 10cm-2m from the camera.

5.2 Robustness

The system robustly detects magenta objects against a busy background. The classification based on features and based on convexity are very robust when applied on a preprocessed image. Furthermore, the preprocessor is able to normalize the colors in an image so that it would in a variety of lighting. Without a this step, neither method would perform nearly as well.

5.3 Speed

The system is very fast and runs at about 30 fps. A laptop's webcam serves as a nice camera to produce a stream of images. This is about the same rate a computer screen refreshes (60fps) and the average human eye can see (45fps).

The speed really could not get any better unless it were running on a real robot that requires a higher refresh rate.

5.4 Range

The system can only detect the largest magenta blob, so it cannot classify multiple dinosaurs. The system only works for reasonable ranges and lighting scenarios. The preprocessor is able to normalize a wide range of daily lighting scenarios, like white light, yellow light, or dim light.

6 Conclusion

The system worked very well given the design time provided. However, like most systems, it could improve from more features, more detection and classification methods, and general continuous improvement.

6.1 Future Work

There are many additional features and projects that were considered, but rejected due to time constraints. They are still very valuable systems to discuss.

6.1.1 Deep Learning using Caffe

Unfortunately real dinosaurs are extinct. Now all that is left of them bones. What if we could classify images or drawings of dinosaur bones as their respective dinosaurs too?

Caffe is an open source deep learning framework that makes this possible. A caffemodel is a belief model made from hundreds or thousands of images of the dinosaur's bones. This model can be matched to the object until we find a match or run out of models and it is fast to check if a model matches an input image.

6.1.2 Handle camera movement

The system cannot handle camera movement because the background removal tool will fail. It would be a considerable improvement to the system if it could still detect and classify dinosaurs while the camera moves.

6.1.3 Cascade detection methods

Although the detection and classification methods on their own are robust, by cascading them together and assigning confidence values to each of their results, the result can be improved.

Furthermore, successfully cascading detection methods may make it easier to detect multiple dinosaurs.

6.1.4 Detect multiple dinosaurs

The system cannot detect multiple dinosaurs in a single image. It only classifies the largest magenta blob found.

6.1.5 Person holding a dinosaur classification

It was considered to classify the person holding a particular dinosaur. For example, if Aditya were holding the Stegosaurus, the system would output "Steg:Aditya". This is an interesting problem because it gives the ability to detect ownership, a uniquely human conceived abstraction.

6.1.6 Human dinosaur pose classification

Finally, recognizing human poses and classifying them as a dinosaur was considered. This involved a heavy human pose classification component that was decided to be out of scope for the given design time. It is a very worthwhile problem to solve, especially as humans and robots have more intimate interactions. For example, robots easily induce psychological discomfort, and if they could recognize human body language as "discomfort" then they could do something to fix that perception.

References

[1] G. Bradski *Dr. Dobb's Journal of Software Tools.*

[2] R. Szeliski, *Computer Vision: Algorithms and Applications.* New York, NY, USA: Springer-Verlag New York, Inc., 1st ed., 2010.

[3] L. B. Grauman K., "Local features: Detection and description,"

[4] B. A. Alcantarilla1 P. and D. A., "Akaze features,"

[5] R. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal, The*, vol. 29, pp. 147–160, April 1950.

[6] S. Boyd and L. Vandenberghe, *Convex Optimization.* New York, NY, USA: Cambridge University Press, 2004.

7 Appendix

```
import cv2
import numpy as np
import cv

class BackgroundRemoval:
    @staticmethod
    def preprocessbackground(c, f, avg2):
        t=0
        gray=0
        while(t<50):
            _,f = c.read()
            # Get average of background with weight of 0.01 for new images
            cv2.accumulateWeighted(f,avg2,0.01)
            t+=1
        #normalize back to 8-bit values
        res2 = cv2.convertScaleAbs(avg2)
        #Remove noise with median blur
        res2=cv2.medianBlur(res2,5)

        return res2

    @staticmethod
    def foregroundMask(color, background):
        b = cv2.split(background)
        im = cv2.split(color)
        mask = None
        out = None
        # For each channel (B-G-R) in the image:
        for i in range(len(b)):
            # Channel c gets a median blur of same size as the kernel
            # used in the preprocessed background
            c = cv2.medianBlur(im[i],5)
            # Gets difference between background and image (chanel-wise)
            imgs=cv2.absdiff(im[i], b[i])
            # Run a Binary OTSU threshold to get what is background and what is
            # not based on the absolute difference.
            (thresh, im_bw) = cv2.threshold(imgs, 250, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)

            # Close the Thresholded image to remove
            # small noises in the threshold
            # Reopen the Threshold
            im_bw=cv2.morphologyEx(im_bw, cv2.MORPH_CLOSE, np.ones((5,5), np.uint8))
            im_bw=cv2.morphologyEx(im_bw, cv2.MORPH_OPEN, np.ones((5,5), np.uint8))
            # Merge Channel Masks
            if mask is None:
                # If it's the first channel being processed
                mask = im_bw
            else:
```

```
                # Else, merge channels by running an OR
                # (wherever it's above the threshold, it's an object)
                mask = cv2.bitwise_or(mask,im_bw)
        return mask
```

Listing 13: Background removal

```
import cv2
import numpy as np

class ColorSegmenter:
    lower_magenta = np.array([125,30,30])
    upper_magenta = np.array([160,255,255])

    @staticmethod
    def getMagentaBlob(bgrimg):
        hsvimg = cv2.cvtColor(bgrimg,cv2.COLOR_BGR2HSV)
        blur = cv2.GaussianBlur(hsvimg,(7,7),0)
        # Threshold the HSV image to get only magenta
        mask = cv2.inRange(blur, ColorSegmenter.lower_magenta, ColorSegmenter.upper_magenta)
        mask = cv2.erode(mask,np.ones((1,1),np.uint8),iterations = 5)
        # mask = cv2.dilate(mask,np.ones((3,3),np.uint8),iterations = 3)
        mask = cv2.morphologyEx(mask,cv2.MORPH_OPEN,np.ones((4,4),np.uint8))
        mask = cv2.morphologyEx(mask,cv2.MORPH_CLOSE,np.ones((15,15),np.uint8))
        # Bitwise-AND mask and original image
        res = cv2.bitwise_and(bgrimg,bgrimg, mask= mask)
        return res
```

Listing 14: Color Segmentation

```

import cv2
import numpy as np
import cv
from ColorSegmenter import ColorSegmenter
from backgroundremoval import BackgroundRemoval
from operator import itemgetter
import copy

def classifyObject(ratioOfAreas):
    if ratioOfAreas>0.5 and ratioOfAreas<0.65:
        return 'T-Rex'
    elif ratioOfAreas>0.70 and ratioOfAreas<0.77:
        return 'Stegosaurus'
    elif ratioOfAreas>0.78 and ratioOfAreas<0.84:
        return 'Triceratops'
    elif ratioOfAreas>0.86:
        return 'Volcano'
    return ''


```

```
22 def normalized(b,g,r):
23     clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(3,3))
24     #b,g,r = cv2.split(img)
25     b1 = clahe.apply(b)
26     g1 = clahe.apply(g)
27     r1 = clahe.apply(r)
28     #bgr = cv2.merge((b1,g1,r1))
29     return b1,g1,r1
30 
31 c = cv2.VideoCapture(0)
32 _,f = c.read()
33 avg2 = np.float32(f)
34 # Get clean - unmoving background
35 background=BackgroundRemoval.preprocessbackground(c, f, avg2)
36 # get new frame
37 _,f = c.read()
38 while True:
39     _,f = c.read()
40     o = copy.deepcopy(f)
41     # gets mask for background removal
42     mask=BackgroundRemoval.foregroundMask(f, background)
43     # Split channels to remove backround and normalize color
44     b,g,r = cv2.split(f)
45     # Applies mask to each channel to remove background
46     nb=np.minimum(mask, b)
47     ng=np.minimum(mask, g)
48     nr=np.minimum(mask, r)
49 
50     # normalize color
51     bn,gn,rn = normalized(ng,nr,ng)
52 
53     # return to color image
54     backgroundRemovedImage=cv2.merge((nb, ng, nr))
55     # Color segmentation - Filter out non magenta
56     res = ColorSegmenter.getMagentaBlob(backgroundRemovedImage)
57 
58     # Canny Edge Detection
59     objectdetection=cv2.cvtColor(res, cv2.COLOR_BGR2GRAY)
60     edged = cv2.Canny(objectdetection, 100, 250)
61 
62     # Close edges
63     kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (7, 7))
64     closed = cv2.morphologyEx(edged, cv2.MORPH_CLOSE, kernel)
65 
66     # Get Contours out of edges
67     contours,hierarchy = cv2.findContours(closed,2,1)
68     areaContours=[]
69     res2 = copy.deepcopy(res)
70     #For each contour in image:
71     for cnt in contours:
72         #Only if there are 2 contours or something
73         area = cv2.contourArea(cnt)
74         #Get Convex hull
75         hull = cv2.convexHull(cnt,returnPoints = False)
76         if(len(hull)>3 and len(cnt)>3):
77             #Get Convexity Defects (where the contour is not convex
    )
```

```
            defects = cv2.convexityDefects(cnt,hull)
            if defects!=None:
                #get contours Areas
                areaContours=areaContours+[(area, len(defects),
    defects,cnt)]
                if area>1000: # Filter out small objects (noise)
                    for i in range(defects.shape[0]):
                        s,e,f,d = defects[i,0]
                        start = tuple(cnt[s][0])
                        end = tuple(cnt[e][0])
                        cv2.line(res2,start,end,[0,255,0],2) #Draw
    convex Hull
                    # cv2.fillConvexPoly(res2, np.array([convexpoly
    ]), (255, 255, 255))

    # Now that coonvex hull is drawn on image, obtain area of
    convex hull as well
    forareaofhull=cv2.cvtColor(res2, cv2.COLOR_BGR2GRAY)
    contoursforhull,hierarchy = cv2.findContours(forareaofhull,2,1)
    areaWithHull=[]
    # for every contours in Hull group
    for cntforhull in contoursforhull:
        #Only if there are 2 contours or something
        area = cv2.contourArea(cntforhull)
        #gets list of convex hulls Areas
        areaWithHull=areaWithHull+[area]

    #Select biggest Area for Hull
    if len(areaContours)>0:
        if max(areaWithHull)>1000:
            #Draw Contour in output image
            defects = max(areaContours,key=itemgetter(1))
            for i in range(defects[2].shape[0]):
                s,e,f,d = defects[2][i,0]
                start = tuple(defects[3][s][0])
                end = tuple(defects[3][e][0])
                cv2.line(o,start,end,[0,255,0],2)
            # Calculate ratio between Hull and shape area
            ratioOfAreas=max(areaContours,key=itemgetter(1))[0]/
    float(max(areaWithHull))
            # Put on text ratio, Classified Object
            cv2.putText(o,"{:0.2f}".format(ratioOfAreas), (0, o.
    shape[0]-80), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))
            cv2.putText(o, classifyObject(ratioOfAreas), (0, o.
    shape[0]-50), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))

    # Show Output
    cv2.imshow('Output',o)
    k = cv2.waitKey(20)
    if k == 27:
        break
cv2.destroyAllWindows()
c.release()
```

Listing 15: Classification based on convexity

```
from __future__ import print_function
import cv2
```

```
from dinoDescriptor import DinoDescriptor
from dinoMatcher import  DinoMatcher
from ColorSegmenter import  ColorSegmenter
from dinoResultsHandler import DinoResultsHandler
import utils2
import numpy as np
import argparse
import glob
import csv
import time


ap = argparse.ArgumentParser()
ap.add_argument("-d", "--db", required = True, help = "path to the object information database csv file")
ap.add_argument("-s", "--samples", required = True, help = "path to the sample(training) data folder")
ap.add_argument("-q", "--query", required = True, help = "path to the query image")
ap.add_argument("-f", "--sift", type = int, default = 0, help = "use SIFT = 1, not use = 0")

args = vars(ap.parse_args())

db = {}

for line in csv.reader(open(args["db"])):
    db[line[0]] = line[1:]

useSIFT = args["sift"] > 0
useHamming = args["sift"] == 0
ratio = 0.7
minMatches = 15

if useSIFT:
    minMatches = 50

descriptor = DinoDescriptor(useSIFT = useSIFT)
dinoMatcher = DinoMatcher(descriptor, glob.glob(args["samples"] + "/*.png"), ratio = ratio, minMatches = minMatches, useHamming = useHamming)
dinoResultsHandler = DinoResultsHandler(db)

# capture from web cam
cap = cv2.VideoCapture(0)
while True:
    ret, queryImage = cap.read()
    if ret == True:
        queryImage = utils2.resize(queryImage, width = 1000)
        # Segment the pink area out
        segImage = ColorSegmenter.getMagentaBlob(queryImage)
        # Describe the query image
        (queryKps, queryDescs, queryKpdRaw) = descriptor.describeQuery(segImage)
        # It is really important to handle the camera idling time.
        if len(queryKps) == 0:
            print("Place The Object In The Camera!")
```

```
            cv2.imshow("Query", queryImage)
        # else let's start matching our samples to the query
        else:
            # To show the key points on query image
            kpImage = cv2.drawKeypoints(queryImage, queryKpdRaw,
    None)
            # let's also add the cool green box
            greenBoxImg = dinoResultsHandler.drawGreenBox(
    queryImage, segImage, kpImage)
            # showing the box must have a timer sleep, otherwise it
     will be flushed
            cv2.imshow("Query", greenBoxImg)
            time.sleep(0.025)
            # Matching, the key step
            results = dinoMatcher.search(queryKps, queryDescs)
            # print out our matching rates
            dinoResultsHandler.showTexts(results)

        # Nicer function, press 'q' to quite the program
        if cv2.waitKey(20) & 0xFF == ord("q"):
            break
    # break when camera is wrong.
    else:
        break
# Release the camera and close all opened windows
cap.release()
cv2.destroyAllWindows()
```

Listing 16: Keypoint detection

```
import numpy as np
import cv2

class DinoDescriptor:
    def __init__(self, useSIFT = False):
        self.useSIFT = useSIFT
        self.kpsRaw = 0
        self.gray = 0

    def describe(self, image):
        # descriptor = cv2.BRISK_create() #we can use BRISK as well
        descriptor = cv2.AKAZE_create() #we can use BRISK as well

        if self.useSIFT:
            descriptor = cv2.xfeatures2d.SIFT_create()
        # make it gray scaled for 2D features
        self.gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        (self.kps, descs) = descriptor.detectAndCompute(self.gray,
    None)
        kps = np.float32([kp.pt for kp in self.kps]) #only take the
     (x,y) attribute and form a NP array.

        return (kps, descs)

    def describeQuery(self, image):
        # descriptor = cv2.BRISK_create() #we can use BRISK as well
        descriptor = cv2.AKAZE_create() #we can use BRISK as well

```

```
        if self.useSIFT:
            descriptor = cv2.xfeatures2d.SIFT_create()
        # make it gray scaled for 2D features
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        (kpsRaw, descs) = descriptor.detectAndCompute(gray, None)
        kps = np.float32([kp.pt for kp in kpsRaw]) #only take the (x,y) attribute and form a NP array.

        return (kps, descs, kpsRaw)
```

Listing 17: Keypoint descriptor

```
import numpy as np
import cv2
from ColorSegmenter import ColorSegmenter

class DinoMatcher:
    def __init__(self, descriptor, samplePaths, ratio = 0.7, minMatches = 30, useHamming = True):
        self.descriptor = descriptor
        self.samplePaths = samplePaths
        self.ratio = ratio
        self.minMatches = minMatches
        self.distanceMethod = "BruteForce"

        if useHamming:
            self.distanceMethod += "-Hamming"

    # now search the thing
    def search(self, queryKps, queryDescs):
        results = {}

        for samplePath in self.samplePaths:
            obj = cv2.imread(samplePath)

            # segmenter = DinoSegmenter2()
            segImage = ColorSegmenter.getMagentaBlob(obj)

            # gray = cv2.cvtColor(segImage, cv2.COLOR_BGR2GRAY)
            (kps, descs) = self.descriptor.describe(segImage)

            score = self.match(queryKps, queryDescs, kps, descs)
            results[samplePath] = score

        if len(results) > 0:
            # sort the result for having the most possible match at the first
            results = sorted([(a,b) for (b,a) in results.items() if a > 0], reverse = True)

        return results

    # now match the query obj (B) with our samplebase (A)
    def match(self, kpsA, featuresA, kpsB, featuresB):
        matcher = cv2.DescriptorMatcher_create(self.distanceMethod)
        rawMatches = matcher.knnMatch(featuresB, featuresA, 2)
        matches = []

```

```
        for m in rawMatches:
            if len(m) == 2 and m[0].distance < m[1].distance * self.ratio:
                matches.append((m[0].trainIdx, m[0].queryIdx)) #be careful with the train/query orders, I'm not sure

        # Test  Info
        print("{}: {}".format("Current matches: ", len(matches)))

        if len(matches) > self.minMatches:
            ptsA = np.float32([kpsA[i] for (i, _) in matches])
            ptsB = np.float32([kpsB[j] for (_, j) in matches])
            (_, status) = cv2.findHomography(ptsA, ptsB, cv2.RANSAC, 4.0)

            return float(status.sum()) / status.size # matching ratio against the object in samplebase

        return -1.0 # no possible match
```

Listing 18: Keypoint classification

```
import numpy as np
import cv2

class DinoResultsHandler:
    def __init__(self, database):
        self.database = database

    def drawGreenBox(self, queryImage, segImage, kpImage):
        # green box
        gray = cv2.cvtColor(segImage,cv2.COLOR_BGR2GRAY)
        ret,thresh = cv2.threshold(gray,127,255,0)
        (_, cnts, _) = cv2.findContours(thresh,
                                        cv2.RETR_EXTERNAL,
                                        cv2.CHAIN_APPROX_SIMPLE)
        if len(cnts) > 0:
            cnt = sorted(cnts, key = cv2.contourArea, reverse = True)[0]

            rect = np.int32(cv2.boxPoints(cv2.minAreaRect(cnt)))
            cv2.drawContours(kpImage, [rect], -1, (0,255,0),2)

        return kpImage

    def showTexts(self, matchedResults):

        if len(matchedResults) == 0:
            print("No samples are matched to the query !")
        else:
            for(i, (score, samplePath)) in enumerate(matchedResults):
                description = self.database[samplePath[samplePath.rfind("/") + 1:]]
                print("{}.{:.2f}% : {}".format(i + 1, score * 100, description))

```

```
                results = cv2.imread(samplePath) # only show the highest matching image
                cv2.imshow("Right: Matched Sample", results)
                cv2.waitKey(5000)
```

Listing 19: Keypoint classification output

```
import numpy as np
import cv2

def resize(image, width = None, height = None, inter = cv2.INTER_AREA):
  # initialize the dimensions of the image to be resized and
  # grab the image size
  dim = None
  (h, w) = image.shape[:2]

  # if both the width and height are None, then return the
  # original image
  if width is None and height is None:
    return image

  # check to see if the width is None
  if width is None:
    # calculate the ratio of the height and construct the
    # dimensions
    r = height / float(h)
    dim = (int(w * r), height)

  # otherwise, the height is None
  else:
    # calculate the ratio of the width and construct the
    # dimensions
    r = width / float(w)
    dim = (width, int(h * r))

  # resize the image
  resized = cv2.resize(image, dim, interpolation = inter)

  # return the resized image
  return resized
```

Listing 20: Utils functions output