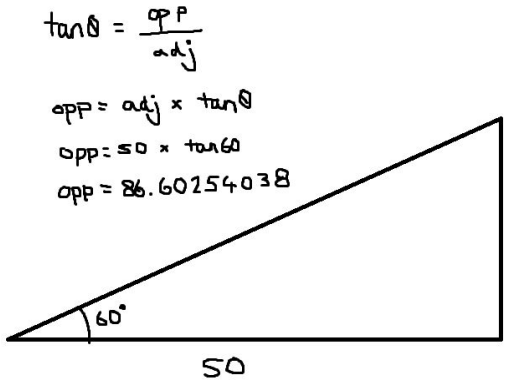
UP892525

There is a README inside of the coursework folder, which states how to run both tasks.

Task 1

Design / Implementation Decisions

Lighting




```
// Setup information about the light
gl.uniform3fv(pwgl.uniformLightPositionLoc, [50.0, 86.6, 0.0]);
```

The lighting in the specification states that the scene should have a light to the top right at a 60 degree angle. To calculate the x and y value of the light, I used SOHCAHTOA to work out the y value based on the degree and a distance to the right of the user (diagram shows calculations). Therefore when placing the light, the x value is 50 (assumption made about the x value) and the y value is 86.6. I made the assumption that the light is relative to the user, so when the user controls the scene, the light will always be to the top right of them.

Satellite

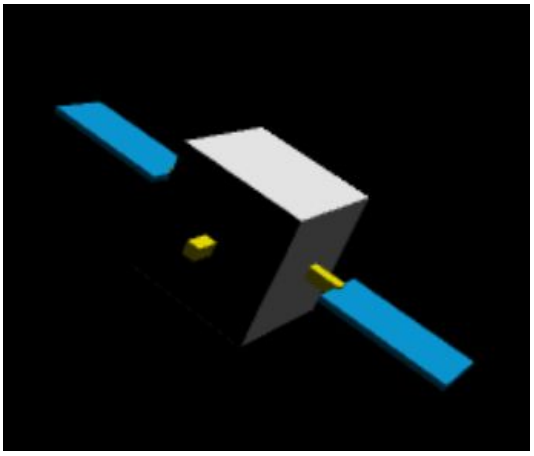
```
//Transforms the cube to create the right rod
changeCube([0.0, 0.0, 1.3], [0.2, 0.2, 0.5], pwgl.goldTexture);
//Transforms the cube to create the left rod
changeCube([0, 0.0, -1.3], [0.2, 0.2, 0.5], pwgl.goldTexture);
//Transforms the cube to create the right solar panel
changeCube([0.0, 0.0, 2.5], [1.0, 0, 2.0], pwgl.solarTexture);
//Transforms the cube to create the left solar panel
changeCube([0.0, 0.0, -2.5], [1.0, 0, 2.0], pwgl.solarTexture);
//Transforms the cube to create the rod at the front
changeCube([-1.3, 0.0, 0.0], [0.4, 0.2, 0.2], pwgl.goldTexture);
```

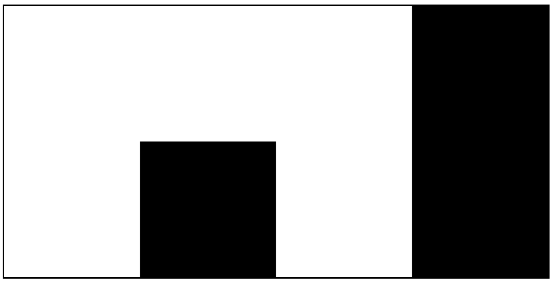
The satellite is made up of multiple transformations of a cube of size [1, 1, 1]. Each transformation is designed using the values given in the coursework specification. When creating each of the shapes, I noticed that I was repeating the same code, so the translation is handled in its own function, ‘changeCube(translation, scale, texture)’. As shown in the screenshot (note some values are in a mixed order due to the direction they are drawn in), the side rods are of size [0.2, 0.2, 0.5], solar panels are of size [1, 0, 2], and the front rod is

[0.2, 0.2, 0.4]. The arrays entered into the first parameter of the function determine the position of the cube in order to create the satellite.

```
mat4.translate(pwgl.modelViewMatrix, [-1.5, 0.0, 0.0], pwgl.modelViewMatrix);
mat4.scale(pwgl.modelViewMatrix, [0.0001, 0.2, 0.2], pwgl.modelViewMatrix);
```

When creating the dish, the specified diameter is to be 4. To achieve this the original globe sphere is changed to create the shape. As the diameter of the globe is 20, the shape is scaled by 0.2 to get a diameter of 4.





For the body of the satellite I assumed that the 5 other sides of the cube not mentioned in the coursework specification are white.
The body of the satellite is textured using the texture on the right. The texture is a 4x2, as it makes the texture mapping easier to manage (as values are done every 0.25, rather than 0.33 on a 3x2 texture), so the right two boxes are not used. The image on the left shows the satellite with the dish removed to show that the surface is black. Texturing on other surfaces (e.g rods and solar panels) is done by creating a 1x1 pixel colour in paint and using that colour as a texture.

Animations

The animations are very similar to the animations done in tutorial 10, so the base of the calculations are from that tutorial.

```
pwgl.orbitAngle = (currentTime - pwgl.animationStartTime)/ pwgl.orbitSpeed *2*Math.PI % (2*Math.PI);
```

The value pwgl.orbitSpeed, is a value that directly affects the speed of the orbit, so this value is controlled by the user using the arrow keys.

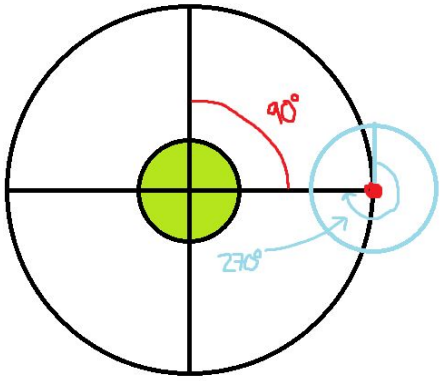
```
pwgl.x = Math.cos(pwgl.orbitAngle) * pwgl.orbitRadius;
pwgl.z = Math.sin(pwgl.orbitAngle) * pwgl.orbitRadius;
```

The value pwgl.orbitRadius is another value that can be changed by the user to affect the radius of the orbit by using the arrow keys.

```
//Calculations for the animation of the earth, the earth is set to rotate on its own very slowly
pwgl.spinEarth = ((currentTime - pwgl.animationStartTime) / 10) / 2000 * 2 * Math.PI % (2 * Math.PI);
```

To create the spin of the globe, I used the same calculation used for the orbit of the satellite, but is set to a non-changeable speed that is slow.

```
pwgl.satelliteAngle = 2 * Math.PI - pwgl.orbitAngle;
```



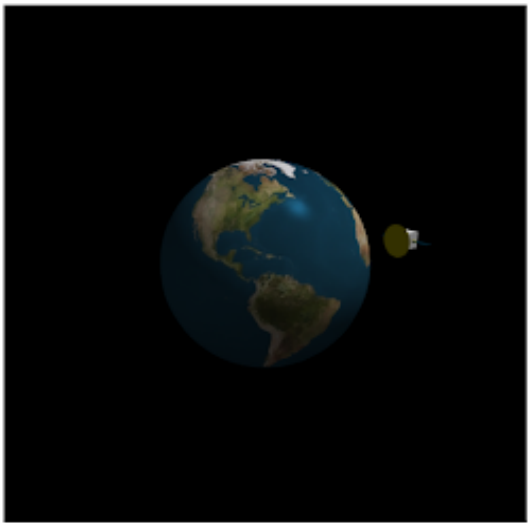
To create the spin of the satellite and make it stay pointing at the center of the globe, I use this calculation. This will take the angle that the satellite is, in relation to the globe, and face it in the exact opposite direction, which is the center of the globe. The diagram above shows this calculation, where the angle is 90, therefore the satellite will face towards 270.

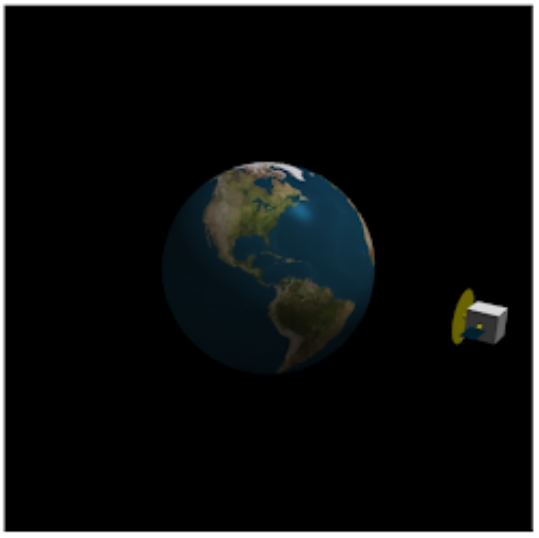
Difficulties or Problems

There were problems when getting the animation of the satellite increasing and decreasing its speed. This is caused by the incrementation being too high when decreasing, causing the satellite to go backwards, rather than slowing down. This can be fixed, by decreasing the value that the speed is decreased by, but however this means that it takes too much time for it to slow down to the point where it doesn't look like it is slowing down. The value that it is currently set to (15 per press) still has the issue, but is severely reduced to stop it looking too bad.

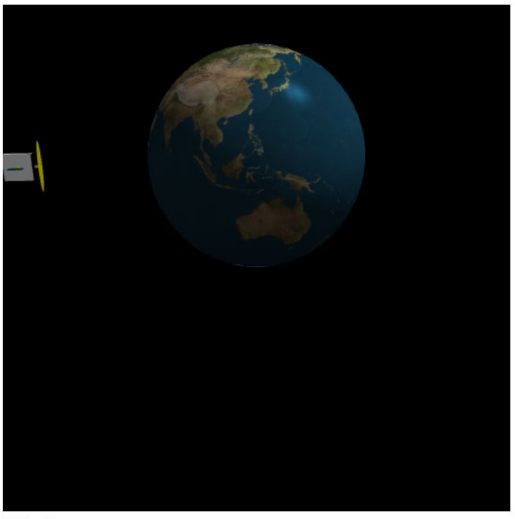
When creating the dish for the satellite, I was unsure of how to create a 2D circle in a 3D space, so I decided to just transform the shape of a sphere. This choice, however, means that the dish is more the shape of an M&M.

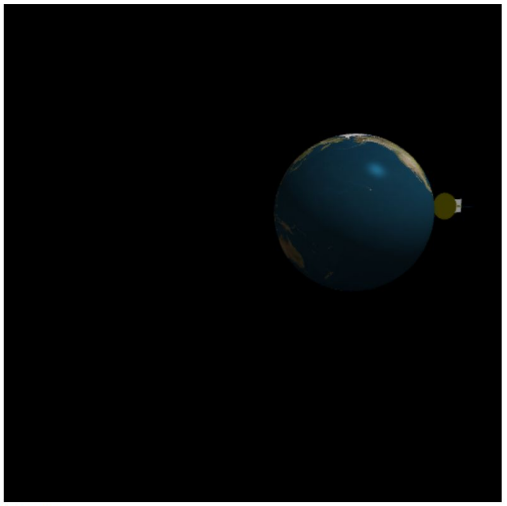
Tests / Evaluation





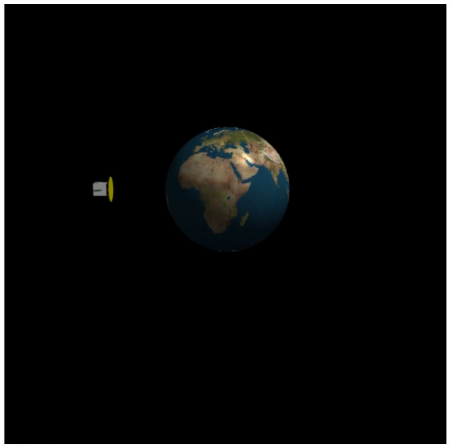
These screenshots are taken right after eachother, showing the movement of the satellite and that the satellite will always face the center of the globe. No issues.
The screenshot also shows the glare from the light that it is to the top right of the user facing towards the globe. There are no issues with either feature.





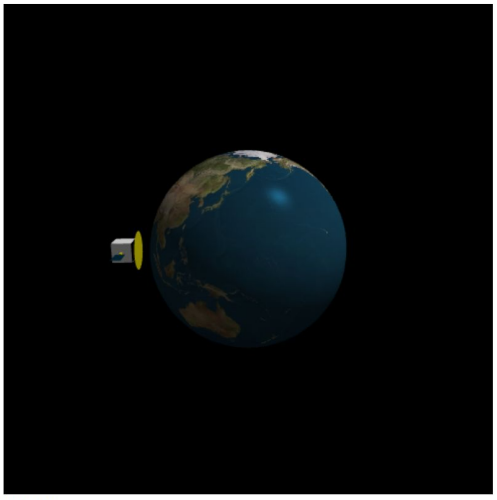
These screenshots are done by refreshing (resetting the globe to the center of the screen), then doing the action and immediately taking the screenshot to show movement.

Using alt-mouse, will move the globe up and down in the y direction (left screenshot moving up). No issues.
Using ctrl-mouse, will move the globe left and right in the x direction (right screenshot moving right). No issues.

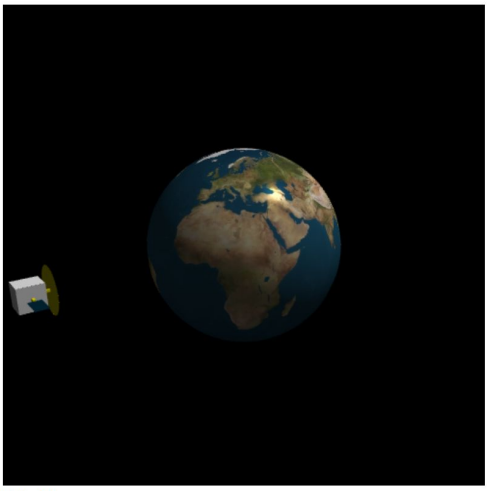


FPS: 60

Using the scroll wheel, will move the globe forward and backward in the z direction (screenshot shows moving backward). No issues.



FPS: 60



FPS: 60

These two screenshots show the control of the satellite's radius using the arrow keys (left arrow to decrease and right arrow to increase the radius). No issues.

It's impossible to show the speed increasing on screenshots, but using the up and down arrow keys does increase the speed of the orbit. The only issue is the visuals of it speeding up and slowing down that was mentioned in the difficulties / problems section.

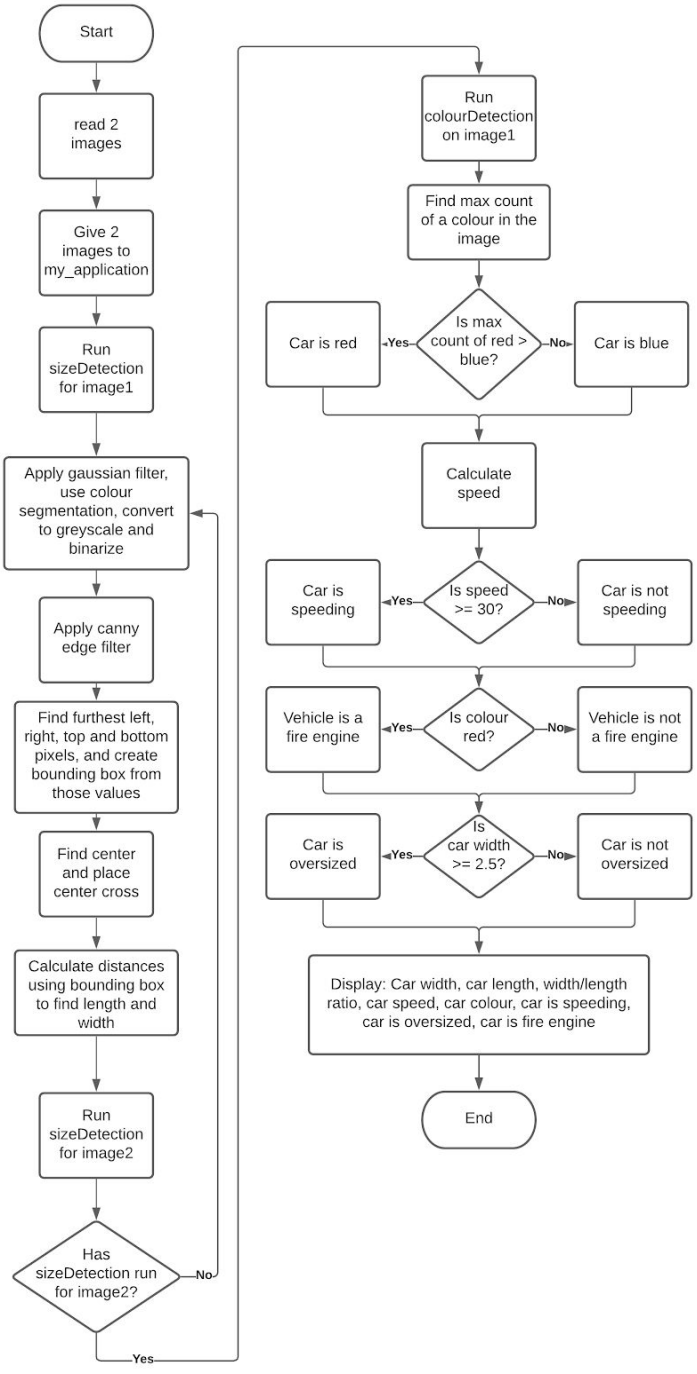
UP892525

Task 2

Conditions / Assumptions

-Distance will be measured by the distance directly to the camera.
-Fire engines are not oversized, based on the results that running the program has shown.

Flow Diagram



Features

Smoothing the image

```
% apply gaussian filter
gaus = fspecial('gaussian', [10,10], 3);
g = imfilter(image, gaus);
```

To achieve this I used the gaussian filter. I found that using [10, 10] as the filter size, and 3 as the sigma value worked to achieve the goal.

Colour Segmentation

```
% colour segmentation using K-Means clustering
lab = rgb2lab(g);   % covert to L*a*b colour space
ab = lab(:,:,2:3);
ab = im2single(ab);
pixel_labels = imsegkmeans(ab, 2, 'NumAttempts', 2);
mask = pixel_labels == 2;
cluster = g .* uint8(mask);
```

To segment colours in the image, I found a method called K-Means and how it could be used to segment colours. I used this as a guide, https://www.mathworks.com/help/images/color-based-segmentation-using-k-means-clustering.html. To achieve the best result in finding the car's shape, the image is split into 2 colour layers, where the blue segments are found in the 2nd layer, which is then used as a mask. The mask is then applied to the image to get the cluster image which isolates the vehicle in the image.

Edge Detection

```
% canny edges
BW1 = edge(bi, 'canny');
```

To find the edges, I used the canny method. The cluster image was grey scaled and binarized, then given to this method to find the edges of the car.

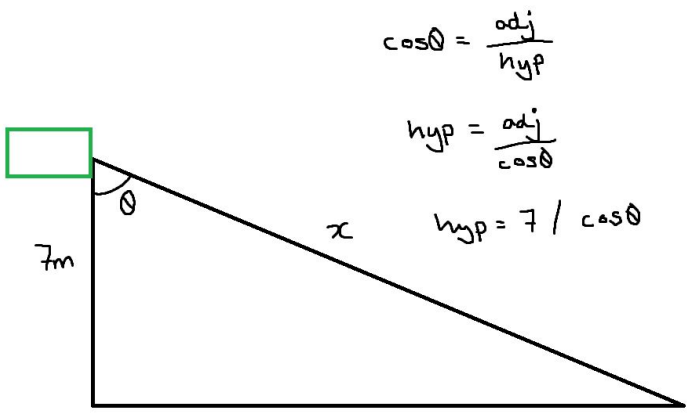
Bounding Box

```
% find the leftmost, rightmost, upper and lower pi
[rows, columns] = find(bi);
leftMost = min(columns);
rightMost = max(columns);
highest = min(rows);
lowest = max(rows);

% width and height of bounding box
width = rightMost - leftMost;
height = lowest - highest;

% rectangle array for the bounding box
boundingBox = [leftMost, highest, width, height];
```

To create a bounding box, I find the white pixels in the binary image and note the furthest left, furthest right, lowest and highest pixels. These pixel locations are then used to create a bounding box around the car.

Distance from camera to the car

```
% work out the distances (front, middle and
frontDistance = calcDist(highest);
middleDistance = calcDist(centerY);
backDistance = calcDist(lowest);

% work out the distance to the camera
function dist = calcDist(numOfPixels)
angle = 60 + ((320 - numOfPixels) * 0.042);
% cos(x) = adj / hyp
dist = 7 / cosd(angle);
end
```

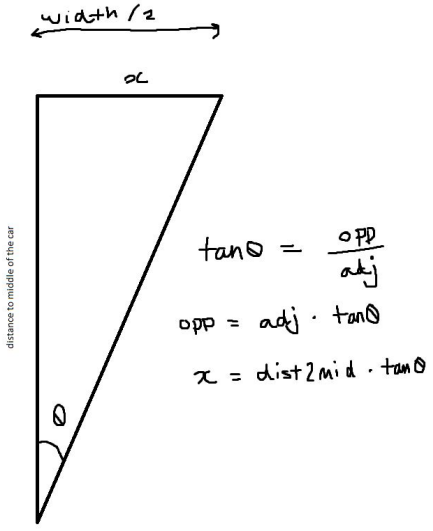


To calculate the distance to the top, middle and bottom of the car, I used the function calcDist(). The function used SOHCAHTOA (shown in the diagram) to calculate the distance from the camera to the car. The calculation places the angle to the center of the image at 60 degrees. When the line is above or below the center of the image, then the angle will be adjusted accordingly. The angle is then placed in the cos formula noted in the diagram.

```
% work out the cars length
carLength = frontDistance - backDistance;
```

The cars length is then calculated from these values.

```
% work out the cars width
widthAngle = (width * 0.042) / 2;
carHalfWidth = middleDistance * tand(widthAngle);
carWidth = carHalfWidth * 2;
```



The car's width is also calculated using a similar method, but calculating half of the width then multiplying the number by 2 to get the full width of the car.

Speed Detection

```
function speed = speedDetection(dist1, dist2)
%dist in meters and time in seconds
distDiff = dist2 - dist1;
%speed in m/s
speedMS = distDiff / 0.1;
%convert to mph
speed = convvel(speedMS, 'm/s', 'mph');
end
```

To calculate the speed of the car, the function takes two distances, which are the bottom y value of the bounding box from 2 different images. It then calculates the difference and divides by the time in seconds it takes to go that distance. It then uses convvel() to convert m/s to mph.

Colour Detection

```
function colour = colourDetection(boundingBox, image)
% select just the bounding box as an image
Imagecrop = imcrop(image, boundingBox);

% colour histograms
red = Imagecrop(:,:,1);
blue = Imagecrop(:,:,3);

[blueCount, ~] = imhist(blue);
blueCount = max(blueCount);
[redCount, ~] = imhist(red);
redCount = max(redCount);

% more dominant colour is the colour of the car
if blueCount > redCount
    colour = 'Red';
else
    colour = 'Blue';
end
end
```

To identify the colour of the car, I use the bounding box to crop the image to the size of the bounding box, and find the histogram with the highest count of that colour, then compare the two values. The value with the highest number is the most dominant colour, therefore the colour of the car.

Is the car speeding?

```
% work out if the car is speeding
if carSpeed >= 30
    isSpeeding = 'Y';
else
    isSpeeding = 'N';
end
```

This is a simple feature to implement, and looks at whether the speed is above or below 30mph. This identifies if the car is above or below the speed limit.

Is the car oversized?

```
% work out if the car is oversized
if carWidth >= 2.5
    isOversized = 'Y';
else
    isOversized = 'N';
end
```

Another simple feature, it looks at whether the car's width is above or below 2.5 meters which identifies if the car is oversized or not.

Is the car a fire engine?

```
% work out if vehicle is fire engine
if contains(carColour, 'Blue')
    isFE = 'N';
else
    isFE = 'Y';
end
```

This is another simple feature that looks at whether the vehicle's colour is blue, or red, and decides whether the vehicle classifies as a fire engine or not.

Showing the outputs

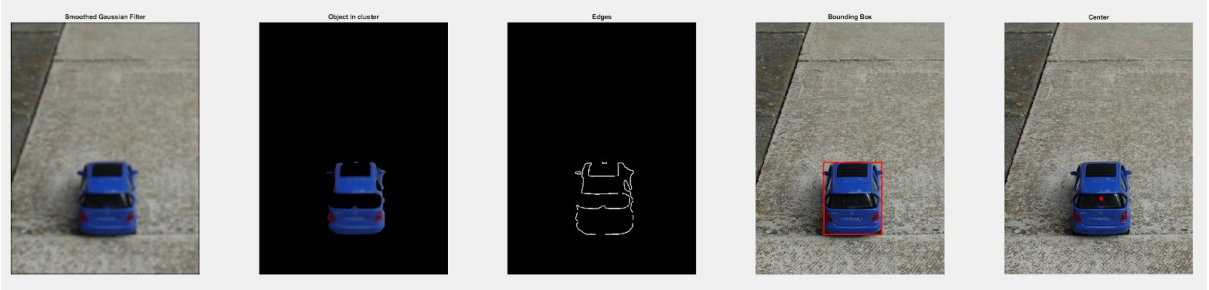
```
% creating the figure to show all of the stages of an image
figure, hold on
subplot(1,5,1)
imshow(g)
title('Smoothed Gaussian Filter')

subplot(1,5,2)
imshow(cluster)
title('Object in cluster')

subplot(1,5,3)
imshow(BW1)
title('Edges')

subplot(1,5,4)
imshow(image)
title('Bounding Box')
rectangle('Position', boundingBox, 'EdgeColor', 'r', 'LineWidth', 2)

subplot(1,5,5)
imshow(image)
title('Center')
hold on
plot([centerX, centerX], [centerY - 6, centerY + 6], 'LineWidth', 2, 'Color', 'r');
plot([centerX - 6, centerX + 6], [centerY, centerY], 'LineWidth', 2, 'Color', 'r');
hold off

hold off
```

Within the main sizeDetection function, a figure is made with 5 subplots, each showing major steps within the process and displays them in one figure. The results of this are shown below in the testing screenshots.

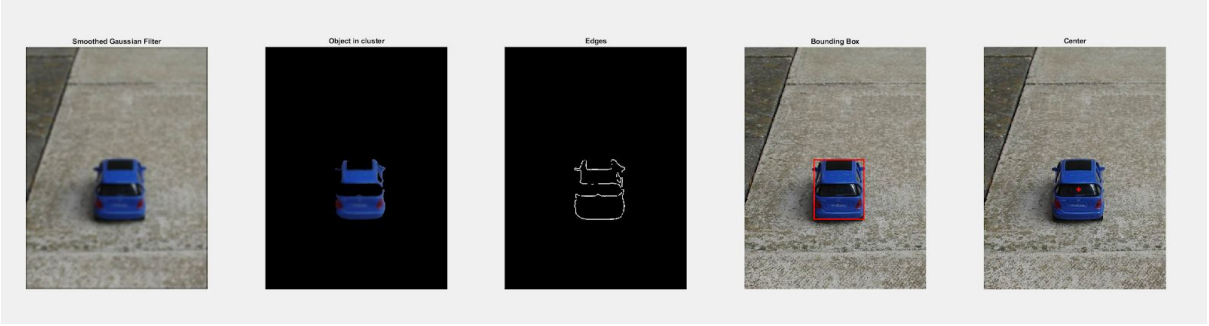
Testing

001.jpg

002.jpg



003.jpg



When running the program with 001.jpg and 002.jpg, the results are the following:

```
>> test()
Car width: 1.3608 meters
Car length: 2.1648 meters
Car width/length ratio: 1.3608 : 2.1648
Car speed: 14.3381 mph
Car colour: Blue
Car is speeding (Y/N): N
Car is oversised (Y/N): N
Car is fire engine (Y/N): N
```

This correctly identifies that the vehicle is not speeding, is not oversized, is not a fire engine, and is the colour blue.

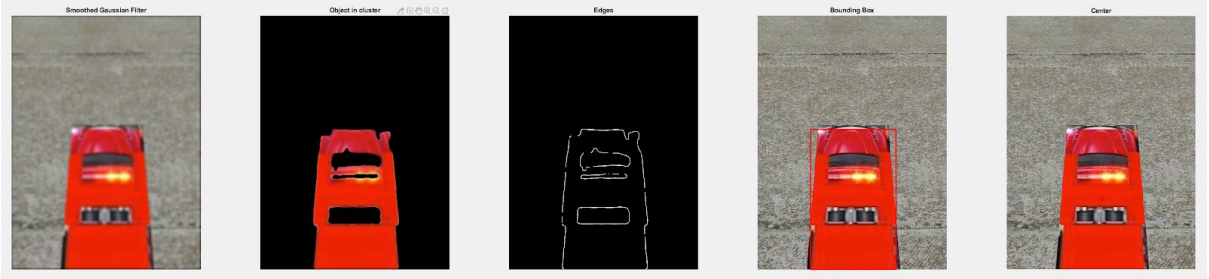
UP892525

When running the program with 001.jpg and 003.jpg, the results are the following:

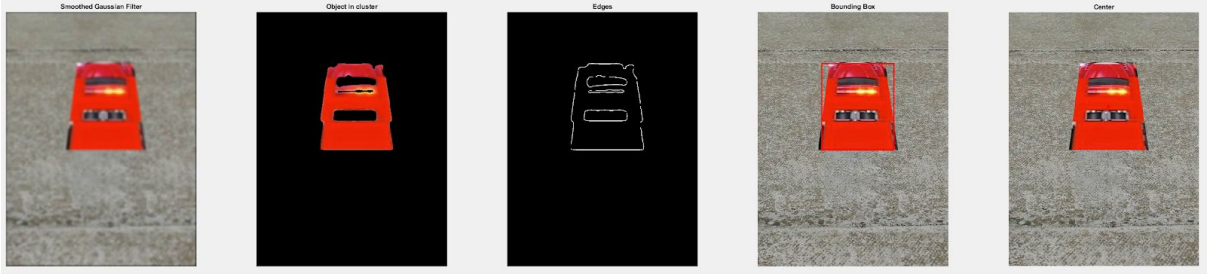
```
>> test()
Car width: 1.3608 meters
Car length: 2.1648 meters
Car width/length ratio: 1.3608 : 2.1648
Car speed: 35.0586 mph
Car colour: Blue
Car is speeding (Y/N): Y
Car is oversised (Y/N): N
Car is fire engine (Y/N): N
```

This correctly identifies that the vehicle is speeding (at 35mph), but the rest of the features are still the same about the car.
When comparing 001.jpg to any images above 002.jpg, the speed continues to increase, but all other features remain the same.

fire01.jpg



fire02.jpg



When running the program with fire01.jpg and fire02.jpg, the results are the following:

```
>> test()
Car width: 1.8952 meters
Car length: 4.4245 meters
Car width/length ratio: 1.8952 : 4.4245
Car speed: 74.9309 mph
Car colour: Red
Car is speeding (Y/N): Y
Car is oversised (Y/N): N
Car is fire engine (Y/N): Y
```

This correctly identifies the vehicle as a fire engine.

UP892525

oversized.jpg

As there is only one oversized image, the picture is run by it self, resulting in this output:

```
>> test()
Car width: 2.7713 meters
Car length: 5.7968 meters
Car width/length ratio: 2.7713 : 5.7968
Car speed: 0 mph
Car colour: Blue
Car is speeding (Y/N): N
Car is oversised (Y/N): Y
Car is fire engine (Y/N): N
```

This identifies that the vehicle is oversized, as its width is above 2.5 meters. The correct colour is also noted. The speed is noted as 0mph as this picture is being compared to itself.