Project Snowblind

Object detection from scratch without AI

A term project for Digital Media

Professor: Khavasi Aliasghar

By Gabriel Ott

Last update: 24.09.2021

Motivation

Object detection algorithms nowadays are made with AI. However, these Algorithms are not flawless. The problem is that this method is not capable of mimicking the object detection which goes on in the human brain. The AI compares the relation between all pixels and to get this right it needs a huge database to train with. Even then, the AI has not grasped the concept of the object it is supposed to detect. A human on the other hand can see an object for the first time and is afterwards able to not just find it in a big picture but also find it when it is partly covered with other objects. This indicates that there is clearly something different going on in the human brain than with the AI. Thus, I conclude that there must be another way to detect objects.

I want to try a different approach. It may not be a better one and I will certainly not be able to perfect it with the limited time given. However, it may find use later or just inspire to try out new approaches.

Idea

To keep it simple, let us look at lines, rectangles, triangles and spheres. How is a human able to clearly distinguish between these objects? I assume that if a human sees four lines which close around something it is some kind of rectangle. To get more information we can compare the length of the lines and the angle the lines form when they intersect each other. We are no longer comparing the relation between individual pixels but the relation between lines. If we manage that, then we can still use AI to make sense of it and hopefully get some better results, without having to train it thousands are even millions of times.

Let me explain the basic idea in a different way. Before there were rockets and satellites, which allowed to see the world from above, how did humans figure out the shape of a continent? Old maps may not be perfectly accurate. However, they clearly show the basic shapes. A more sophisticated example is a pixelart in Minecraft. Imagine you walk on this pixelart and cannot take a view from above. How can you determine, which object is shown on the pixelart? It is a similar situation to the mapping of continents. And there is a simple solution: just follow the outer border of a certain colour or just the coastline. Save the direction you are going to (and make sure you always measure your direction relative to a certain object like star formations). If you cannot follow your direction any longer, take a new direction and continue. This way a human can draw his own movement on a map and determine the shape of continents.

A computer model should be able to do the same thing to detect different objects. There is just one difference: the program cannot see multiple steps ahead. It can just reliably evaluate the pixels right

next to its current position. Like a thick fog which decreases you viewing distance. And to give it a better-sounding name: just like you are snowblind.

Basic plan

Making this idea reality is relatively simple. I need a program which follows the outer border of an object in an image and save certain things like direction. Afterwards I can conclude the overall object-shape and compare it with the object I know.

Images are usually big pixel-matrices with gradual colour changes. This can make it very hard to detect anything at all. To eliminate this problem, the program must first simplify a given image so that there is a hard border between differently coloured structures. Then this differently coloured structures can be analysed one by one.
You may think: that is the wrong way! There can be objects made up of multiple colours and this program will have a hard time detecting them!
This argument is not without merit. Let us take the colourful “G” for many Google-products. It clearly consists of many colours but humans can obviously see the capital “G”. This case should not pose a problem to my program, since it goes trough *all* colours in a picture and the background of this colourful “G” is white. The program is supposed to follow a trail of a colour where it borders to other colours. It can thus still get the G-shape.

After this simplification the program needs to go through each colour in the image and follow the borders. Just like a child does when it snips out a pre-drawn object from paper.

When that is done, the program needs to analyse what choices of movement it made to snip a certain structure out. These choices must be analysed for roundings or lines. Humans can easily recognize objects made up from simple shapes. This may be why most objects we make consist of straight lines and circular parts. The program needs to detect this to evaluate the shape. There are simple ways to detect certain objects with limited information. For example, when the structure is closed, that means that when you snip it out you end up at your starting position, and when the structure consists of three lines, then it can only be a triangle.

Why Octave?

For this project I used Octave. There are of course countless programming languages which are up to the task and I really love C++ and C. However, in C++ and C there are just so many little details to pay attention to that I might get bogged down just with reading images and picking the right pixels.

Octave has image tools by default. Besides, it is made to work with matrices and they are a wonderful way to do calculations with images. Moreover, in the Octave debug mode I am always able to check the path the program took for myself and can display it in a matrix.

Octave is the open-source version of Matlab. This enables even people without access to Matlab to use my program later. Frankly, the Octave documentation is not that good but it is so similar to Matlab that most of the time the Matlab documentation helps as well.

Project implementation

I wrote seven functions to make this program work. The first one is the start function. Important, general parameters like the path to the image of interest can be changed there. After that the start function just organizes the order in which the other functions are called.

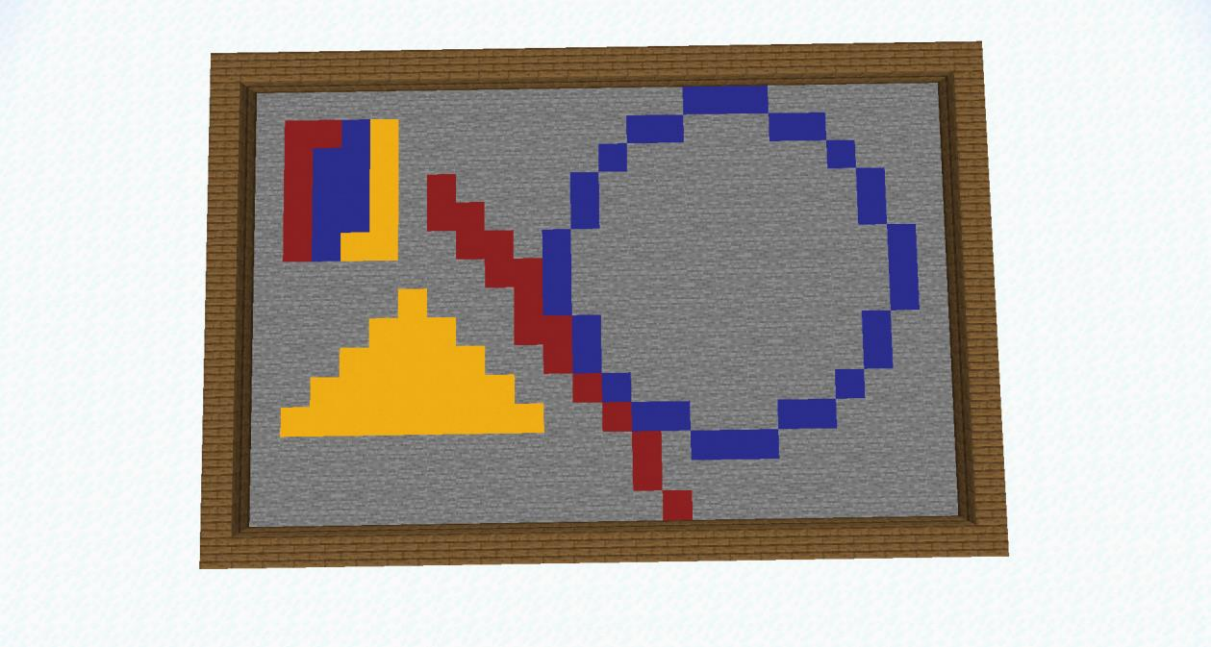
Picture 1: An example image in a wooden frame.

The load_and_simplify function is supposed to simplify the image by transforming it into grayvalues and only giving colours which appear commonly to the next function. If the image is not black and white from the beginning on, some simplification takes place. It may not be the best solution for pictures which are mostly in one colour but otherwise it should be enough. There is a maximum number of pixeltypes. If it is, for example 8, then that means the function shall reduce the image in a

way that there are at most 8 different pixels to be found excluding white. The colour-spectrum goes from 0 to 255. I can just divide 256 by 8 and can thus say multiples of 32 are the allowed pixeltypes. Then I need to map each value to a pixeltype. A value of 15 is supposed to be set so value 0, a value of 17 to 32 and so on. So the program takes the middle between to pixeltypes as a tolerance. There must be an exception for white since 256 does not exist as a value.

Picture 2 shows what that could look like for the example image. Take note of the transformation losses. After that is done the rgb-picture gets converted to grayscale. Then the program finds all the different gray-values and looks, if they are common enough to be of any relevance. The function gives back the relevant colours and the grayscale picture.

For the example in this document, I skip the grayscale conversion and keep to colours to clearly identify the pixels I am referring to. In this example, load_and_simplify is supposed to identify the “colours” grey, red, yellow and blue (to be more precise: the grayscale-values of these colours).



Picture 2: This is what the example image could look like simplified and before the conversion to grayscale.

In the next step, the snip_prepare function takes each relevant colour and looks for it in the simplified picture. The goal is to make visible where the structure described by this colour borders with other colours. Therefore, the program looks where pixels with this colour are and saves their

positions. I call them the relevant pixels. Then it creates a matrix which has ones everywhere but zeros where the relevant pixels are.

	0																							
									0	0	0	0	0	0										
			0						0	0	0	0	0	0										
		0	0	0					0	0			0	0					0	0	0			
	0	0	0	0	0				0	0			0	0				0	0	0	0	0		
0	0	0	0	0	0	0			0	0	0	0	0	0				0	0	0		0	0	0
									0	0	0	0	0	0				0	0				0	0
																		0	0	0		0	0	0
																			0	0	0	0	0	
																				0	0	0		

Table 1: Example of a simplified image-matrix. The zeros mark the relevant pixels. Empty space equals one.

Then the program sums up the values of the four pixels surrounding each relevant pixels and writes the result in a new matrix. 0 means that all bordering 4 pixels are relevant pixels. 4 means that all bordering pixels are non-relevant pixels. -2 is a special value for all the non-relevant pixels. Later “used” pixels will get the value -1 and this way we can tell them apart from the background pixels.

	4																							
									2	1	1	1	1	2										
			3						1	0	1	1	0	1										
		2	0	2					1	1			1	1					2	1	2			
	2	0	0	0	2				1	1			1	1				2	0	1	0	2		
3	1	1	1	1	1	2			1	0	1	1	0	1			2	0	2		2	0	2	
									2	1	1	1	1	2			1	1				1	1	
																	2	0	2		2	0	2	
																		2	0	1	0	2		
																			2	1	2			

Table 2: Example of a simplified image-matrix. Empty space equals zero.

There is a slight problem at the borders of the image because there a pixel does no necessarily have four neighbours and when you try adding up their values it may be out of range. To prevent this, the image-matrix gets a border filled with zeros and every position gets x++ and y++ to still address the relevant pixels in the right way.

As a summary up to this point, now the image is reduced to a matrix which is filled with numbers regarding how close these pixels are to the border of the structure they describe.

At this point, the “snipping” can happen by the do_snipping-function. First, this function must find a starting point. It just looks at the first position in the matrix which has a value greater than 0. Then it looks at the pixels which surround this position. In contrast to the calculation of the values in the snip_prepare functions, this function also looks at diagonal values. Each surrounding pixel of the current position R is given a value like this:

$$\begin{matrix} 1 & 2 & 3 \\ 8 & R & 4 \\ 7 & 6 & 5 \end{matrix}$$

Direction-Matrix

The program looks which surrounding pixel has the highest value and saves it. After that it jumps to this new position and continuous. To prevent jumping to previous pixels and to still see the outline when debugging, the current pixel R is set to value -1 at first contact. Moreover, the algorithm ends when it cannot jump to a value greater than 0. So far so simple.

	-1																								
									-1	-1	-1	1	1	2											
			3						1	0	1	1	0	1											
		2	0	2					1	1			1	1					2	1	2				
	2	0	0	0	2				1	1			1	1				2	0	1	0	2			
3	1	1	1	1	1	2			1	0	1	1	0	1				2	0	2		2	0	2	
									2	1	1	1	1	2				1	1				1	1	
																		2	0	2		2	0	2	
																			2	0	1	0	2		
																				2	1	2			

Table 3: In-process picture. The single pixel in the upper left corner is ignored. The dark red shows a problematic part.

However, this simple algorithm gets more complicated when there are multiple pixels which share the highest value and we are not at the start as marked with red in table 3. When the program choses randomly it may get stuck later without visiting the other part. That is why there is a function which tells the program to take the value which is closest to its last value. In other words: is should try to continue walking in the direction it came from. This evaluation can be tedious because of the border-values 1 and 8. For this reason, there is a unique function which calculates the proximity between to values, by just increasing/decreasing the current value at the same time and seeing

which value is first hits the target value: compare_direction_val. For later use it also evaluates the clock-direction in took like was it a turn clockwise or anti-clockwise? For example, if the program jumps on pixel 4 from its current location and this position offers next the options 4 or 5, then it will choose the 4.

									-1	-1	-1	-1	-1	-1											
			-1						-1	0	1	1	0	-1											
		2	0	-1					-1	1			1	-1							2	1	2		
	2	0	0	0	-1				-1	1			1	-1					2	0	1	0	2		
3	1	1	1	1	1	-1			-1	0	1	1	0	-1				2	0	2		2	0	2	
							4		-1	-1	-1	-1	-1	-1				1	1				1	1	
																		2	0	2		2	0	2	
																			2	0	1	0	2		
																				2	1	2			

Table 3: In-process picture. A rectangle has been successfully walked along. The green marked second rectangle can be identified in a second walkthrough. The dark red shows a problematic dead end.

Another interesting thing is, what happens when you jump to a pixel which has the value 4. That might just be a dead end because this pixel has no pixel of its own colours on its four sides. However, it can have diagonal partners but the program does not know. Whenever the program jumps to a value of 4 and there are multiple options, the current process is saved as a backup. If the taken way turn out to be a dead end, then the backup will be loaded. However, the way which lead to the dead-end will still be marked with -1-values so the program does not take it again. Instead, it is forced to take the other option. The backup gets deleted after use so no loop can occur.

Alle the jump-values are written in a vector called vector_history. And the function analyze_vector analyses this vector. This vector contains the whole structure. It is comparable to “up, up, right, right, down, down, left, left” instructions. And these instructions can be analysed and you can see: they make up a triangle. A big picture, which describes an object can thus been reduced to a single vector. The task of the analyse-function is firstly, so compress this set of instructions even further. “up, up, up” can be summed up by 3x”up”.

									-1	-1	-1	-1	-1	-1											
			-1						-1	0	1	1	0	-1											
		-1	0	-1					-1	1			1	-1						2	1	2			
	-1	0	0	0	-1				-1	1			1	-1					2	0	1	0	2		
-1	-1	-1	-1	-1	-1	-1			-1	0	1	1	0	-1				2	0	2		2	0	2	
									-1	-1	-1	-1	-1	-1				1	1				1	1	
																		2	0	2		2	0	2	
																			2	0	1	0	2		
																				2	1	2			

Table 4: Vector-history for the triangle when start and end is at the blue marked spot: 5 5 5 8 8 8 8 8 8 3 3 3. The analysation function makes three lines with the direction 5, 8 and 3 out of this.

However, it is not always this simple. There can just be bad pixels because of the simplifications which were made before. That is why there is a ignore-value which determines how many out-of-the-line pixels can be tolerated before ending the current vector and starting a new one.

									-1	-1	-1	-1	-1	-1											
			-1						-1	0	1	1	0	-1											
		-1	0	-1					-1	1			1	-1						2	1	2			
	-1	0	0	0	-1				-1	1			1	-1					2	0	1	0	2		
-1	-1	-1	-1	-1	-1	-1			-1	0	1	1	0	-1				2	0	2		2	0	2	
						-1			-1	-1	-1	-1	-1	-1				1	1				1	1	
																		2	0	2		2	0	2	
																			2	0	1	0	2		
																				2	1	2			

Table 5: Vector-history for the triangle when start and end is at the blue marked spot: 5 5 5 **6 1** 8 8 8 8 8 3 3. The problematic pixel is marked red and the resulting divergence in the vector-history is printed boldly.

The analytics algorithm looks at the vector-history entry by entry and starts to form vectors by taking a few values and seeing if the next value does no differ to much from the mean of the values before. This way, even diagonals which have a vector history like

4 4 4 4 3 4 4 4 3 4 4 4 4 3 3

are understood as a single vector. The mean of the vector is between 0 and 9, because there can be calculation discrepancies with between 8 and 1. 0 and 8 point technically in the same direction and 1 and 9 as well. Depending on the general direction of the vector to far, the value is understood

differently. Thus, with the direction-matrix in mind and the total length of the vector the function can describe any straight line. When lines end, there start other lines and this is a sign of corners.

When everything is analysed to lines, what about circular shapes?

									-1	-1	-1	-1	-1	-1										
			-1						-1	0	1	1	0	-1										
		-1	0	-1					-1	1			1	-1						-1	-1	-1			
	-1	0	0	0	-1				-1	1			1	-1					-1	0	-1	0	-1		
-1	-1	-1	-1	-1	-1	-1			-1	0	1	1	0	-1				-1	0	-1		-1	0	-1	
									-1	-1	-1	-1	-1	-1				-1	-1				-1	-1	
																		-1	0	-1		-1	0	-1	
																			-1	0	-1	0	-1		
																				-1	-1	-1			

Table 6: Fully analysed matrix. The inner “circle” is identified in a second walkthough. Vector-history for the outer circle when start and end is at the blue marked spot: 4 4 5 5 6 6 7 7 8 8 1 1 2 2 3 3

That is, were the third analysation layer comes into play. It takes the collection of vectors and checks, if they gradually change and rotate in the same direction like clockwise or anticlockwise. In this way, roundings and even whole circular structures can be identified and looked at as one single unit instead of many small vectors.

After all this effort, the determination of the end result is hardly implemented in my program, due to the countless combinations of data which this method can create. If there is one line, which gradually closes in on itself, it has to be a circle-like structure. If there are 4 lines which close in on itself it must be something like a rectangle or trapezoid. All in all, the program can clearly distinguish between three test-pictures which show a circle, a triangle and a rectangle. The objects are not tilted and on quite small pictures, so that one can see the whole number-matrix for debugging. In respect to that this is just supposed to be a term project, the program is a success.

At the moment, the Snowblind-program is not very accurate and not really close to detecting any objects. To make it even better, there must be better ways to analyse the vector-history. The angle between different vector is very helpful when determining the exact shape instead of saying “it is something with 4 lines”. Moreover, shapes can be very complex and they can consist of multiple

other shape. The analytics options are endless and for real-world object detection it is also necessary to understand the context in which a certain shape is to understand what it is meant to represent. However, I am convinced that it can be done. Simple structures like letters, numbers and basic forms are certainly doable.