| **The Generalized Hough Transform for Player Model Recognition** | | | |
| --- | --- | --- | --- |
|  | ECE 532 - Spring 2023 Semester | |  |
|  | Professor Jeffrey Rodriguez | |  |
| *University of Arizona College of Engineering*  *Nicolas Blanchard* | | | |
|  | *May 2, 2023* | |  |
|  | | | |

**Table of Contents**

[**1.0 Introduction and Scope 2**](#_wkjepj3pcqsg)

[**2.0 Applications 3**](#_8gs04f8pq0fm)

[**3.0 Generalized Hough Transform Approach 4**](#_w10kd1kry0pq)

[3.1 Basic Generalized Hough Transform 4](#_p60mjc9qtc2i)

[3.2 Scale Invariant Generalized Hough Transform 7](#_t0gincavtz9d)

[3.3 Other Hough Transform Considerations 10](#_yy5161tlyvb1)

[3.4 Auxiliary Image Analysis Code 12](#_9590xqqyn9sq)

[3.5 Generating R-Table 16](#_dtdvoacup6vo)

[**4.0 Machine Learning Approach 18**](#_ahhamqdij8g1)

[**5.0 Comparison of Results 20**](#_qwm75xh9tvkx)

[5.1 White Training Map 22](#_e7yopdb5ch40)

[5.2 Dust II 23](#_c6bd03c0xnnw)

[5.3 Cache 24](#_pw0d217py2jv)

[**6.0 Discussion and Conclusion 25**](#_g4q6gc4qzc9v)

[**7.0 Further Research 26**](#_35msnmfp22o1)

[**8.0 Reproducibility 27**](#_jm0gyh7tlqzj)

[**9.0 References and Acknowledgements 29**](#_b3pt5qfpd7iu)

# 1.0 Introduction and Scope

My project focuses on target recognition in video games. Specifically, I implement Ballard’s generalized Hough transform from scratch to recognize a player model in the game *Counter-Strike: Global Offensive*. The generalized Hough transform is implemented using an R-Table in polar (*r*, 𝜙) format, and arbitrary scaling of the entries in the R-Table is taken into account to make the generalized Hough transform scale invariant. R-Tables describing the outline of a player model are used to recognize enemies in screenshots of the videogame.

The goals of my custom player model detector are as follows:

1. *Determine if there is an enemy currently on the screen.*
2. *If there is an enemy on the screen, determine the precise location of the enemy and highlight the model.*

In addition, this method is compared against the emerging standard in object recognition: a deep convolutional neural network (DCNN). The chosen DCNN is YOLOv3, an open source model that can be trained with the aid of a custom MATLAB script.

Although training the DCNN to recognize player models proved to be quite difficult due to the large amount of preparation and custom MATLAB code required for training, the majority of my work was focused on implementing the generalized Hough transform without any pre-existing code. There were many iterations that refined how the vectors stored in the R-Table were used to increment the accumulator bins, added considerations to control the precision of the detection, created a third dimension to support scale invariant detection, set proper thresholds for detection, and more.

# 2.0 Applications

*Counter-Strike: Global Offensive* is a game of quick reaction times and precision aim. Players compete to eliminate each other as fast as possible by shooting each other. Therefore, the players that are the fastest at recognizing enemies on their screen have a massive competitive advantage.

|  |
| --- |
| *Figure 2.0.1 - As rank (FACEIT ELO) increases, reaction time decreases (Słodownik, 2021).* |

Historically, exploits for *Counter-Strike: Global Offensive* have been executed within the game code itself. This means that the cheat will attach itself to the game and read memory being used by the game to gain information about player positions. The cheat can then notify the player of the enemy’s position. The downside is that the invasion into game memory is easily detectable. My proposed method of player model detection using image processing is undetectable and will only ever need an update if the visual appearance of the player model changes. Although my current implementation does not run in real time, it could theoretically be applied to run concurrently with the game to recognize player models during gameplay.

# 3.0 Generalized Hough Transform Approach

## 3.1 Basic Generalized Hough Transform

A generalized Hough transform is employed as the backbone of the player model detection. The generalized Hough transform was developed by Ballard and extends the traditional line-detection abilities of the Hough transform into the realm of curves and parabolas (Ballard, 1981). The algorithm is re-created based on his 1981 publication in *Pattern Recognition* titled ‘Generalizing The Hough Transform To Detect Arbitrary Shapes’.

I understand that my audience (Professor Rodriguez) is already familiar with the generalized Hough transform, so my explanation will be brief. The idea of the generalized Hough transform is to match a predefined shape to the edge map of an image using a table of displacement vectors. This table, the R-Table, describes the shape that you are trying to detect. An accumulator array measures what areas of the image have the best matches to the shape described in the R-Table.

To start with, we allocate variables for our accumulator array (H) and find the size of the image we are working with.

|  |
| --- |
| *Figure 3.1.1 - Setting up the generalized Hough transform.* |

Next, we iterate over each pixel in the image. If that pixel is an edge pixel, then we loop through each vector stored in the R-Table and increment the corresponding bins in the accumulator array.

|  |
| --- |
| *Figure 3.1.2 - Executing the generalized Hough transform.* |

The next step is to detect where the peaks in the accumulator array occur. To do this, we iterate over every entry in the accumulator array, checking each one to see if:

1. The value is above a set, arbitrary threshold defined by the user.
2. The value is the highest in the region around it.

*Figure 3.1.3* below shows how each element on the interior of the accumulator array is checked. The same process is repeated for the edges of the accumulator array, but the coordinates of the surrounding elements are altered to prevent out-of-bounds errors.

|  |
| --- |
| *Figure 3.1.3 - Filtering for peaks within the accumulator array.* |

Each peak identified within the accumulator array represents a detected shape. We can then use this information to identify where in the image we have discovered our shape of interest (the player model). To test the implemented generalized Hough transform, I created the test image shown in *Figure 3.1.5*. It has two circles of different sizes, a square, and a line. Using an R-Table that describes a circle of radius 20 pixels, let’s apply the generalized Hough transform and observe the results.

|  |  |  |
| --- | --- | --- |
| *Figure 3.1.4 - Generating R-Table for a circle of radius 20 pixels.* | *Figure 3.1.5 - Test image containing a circle of radius 20 pixels (upper left), a circle of radius 10 pixels (bottom right), a straight line (upper right), and a square of side length 25 pixels (bottom left).* | *Figure 3.1.6 - Output from the generalized Hough transform. A gray circle of radius 20 pixels is drawn at the peak found within the accumulator array.* |

## 3.2 Scale Invariant Generalized Hough Transform

The next step is to improve the generalized Hough Transform to be scale invariant. Ballard describes a method for this - by adding another dimension to the accumulator array, we can make the generalized Hough transform scale invariant.

The code to set up the scale invariant generalized Hough transform is similar to what we saw before, but notice that we now have a vector *coeffs* storing different scaling coefficients. Also, the accumulator array has a third dimension to store the scale coefficient.

|  |
| --- |
| *Figure 3.2.1 - Setting up the scale invariant generalized Hough transform.* |

Our main Hough transform is also similar to what we saw before, but now we also check every single scale value for each entry within the R-Table. The vectors stored within the R-Table are multiplied by the scale value so that we are essentially checking many different sizes of the same shape. Note that previously, we were using Euclidian (*x, y*) coordinates in the R-Table, but in this iteration of the transform, polar (*r*, 𝜙) coordinates are used.

|  |
| --- |
| *Figure 3.2.2 - Executing the scale invariant generalized Hough transform.* |

Detecting peaks is accomplished in a similar manner as before, but now we need to also check the third dimension of the Hough array, the scale coefficient.

|  |
| --- |
| *Figure 3.2.3 - Filtering for peaks within the accumulator array (scale invariant). For the rest of the function (where the borders of the accumulator array are checked), see hough\_scale\_invariant.m.* |

The result of our modifications is that the generalized Hough transform is now able to detect circles of any size within our image.

|  |  |  |
| --- | --- | --- |
| *Figure 3.2.4 - Generating R-Table for a circle of radius 20 pixels.* | *Figure 3.2.5 - Test image containing a circle of radius 20 pixels (upper left), a circle of radius 10 pixels (bottom right), a straight line (upper right), and a square of side length 25 pixels (bottom left).* | *Figure 3.1.6 - Output from the generalized Hough transform. Gray circles are drawn at the peak coordinates and scales found within the accumulator array.* |

## 3.3 Other Hough Transform Considerations

Ballard’s generalized Hough transform also allows for rotation-invariant shape detection. This involves adding a rotation value as a fourth dimension to the accumulator array, similar to how scale coefficient as added as the third dimension. However, in my application, this is not useful; we know that the playermodel will always be in the same rotation. The player model is always standing straight up, as is the camera that is viewing the player model.

Another consideration is the precision of the accumulation array. Because potential shapes in an image do not perfectly match to the shape described by the R-Table, the entries into the accumulator array are often split between several neighboring bins, even though they would ideally be concentrated in a single bin for a perfect match. This hinders shape detection because if the entries are split between bins, it masks the fact that a match was detected. To combat this, the precision of the accumulator array can be reduced. Having fewer bins in the array means that more matches are concentrated together. When a match is positively identified, it cannot be pinpointed on the original image as precisely, but for bins that are only a few pixels wide, that is not an issue.

*Figure 3.3.1* shows how an accumulator array can be allocated given a set number of pixels that each bin corresponds to.

|  |
| --- |
| *Figure 3.3.1 - Allocating an accumulator array for variable bin sizes.* |

Then, when calculating which bin to increment, we need to translate between the pixel coordinates and the coordinates of the corresponding bin.

|  |
| --- |
| *Figure 3.3.2 - Translating pixel values into corresponding bin coordinates.* |

Note that the variable *pixels\_per\_bin* should be a perfect square for this implementation to function properly.

Lastly, edge orientation is also taken into consideration in this implementation of the generalized Hough transform. Ballard describes that using directional information does make the algorithm faster and more accurate (Ballard, 1981, p. 115). To implement directional information, we store the target gradient angle as an entry in the R-Table (alongside displacement vector magnitude and angle) and compare this target gradient angle with the gradient angle of each edge pixel. That way, for each edge pixel, we only need to increment the accumulator array bins that correspond to an R-vector that has a gradient angle matching the gradient angle of the current edge pixel. Based on my testing, I can confirm Ballard’s observation; this consideration did indeed massively increase both the accuracy and speed of the algorithm.

|  |
| --- |
| *Figure 3.3.3 - Checking the gradient direction of an edge pixel against entries in the R-Table.* |

## 3.4 Auxiliary Image Analysis Code

The most important auxiliary image analysis code that supports the generalized Hough transform is the edge detection. Edge detection is handled by a custom function that implements an extremely simple edge detector that estimates gradient magnitude. If this code seems familiar, that’s because it is! It is gratefully inspired by code provided by Professor Rodriguez’s fantastic ECE 532 lectures.

|  |
| --- |
| *Figure 3.4.1 - Gradient magnitude and direction are calculated for the input image.* |

|  |
| --- |
| *Figure 3.4.2 - Non-Maximum Suppression (NMS) is implemented using interpolation. Provided by Professor Rodriguez in his ECE 532 lectures.* |

|  |
| --- |
| *Figure 3.4.3 - Gradient magnitude thresholding.* |

Image segmentation is also utilized to assist in the labeling of detected player models. When the generalized Hough transform detects a player model within the frame, it tells us where the center of that player model is. So, if we can separate the image into its foreground segments, then whichever segment contains the center of the detected player model is the entire playermodel.

The implemented image segmentation is Haralick’s algorithm for connected component labeling, as discussed in lecture. To save time I will not go into detail, but see images of the implementation below.

|  |  |
| --- | --- |
| *Figure 3.4.4 - Establishing initial labels for background pixels.* | *Figure 3.4.5 - Scanning through foreground pixels to find connected regions.* |

Lastly, binary dilation is utilized to assist with segmentation. Segmentation performs best when there are clearly defined and separated segments in the image. To ensure that the provided edge map has strong edges and fully separated objects, the edge map is dilated before being sent off to segmentation. The dilation implemented uses a square of variable width as the structuring element. The foreground is assumed to have a gray level of 0.

|  |
| --- |
| *Figure 3.4. - Binary dilation.* |

## 3.5 Generating R-Table

The final step in detecting player models using our scale-invariant generalized hough transform is creating an R-Table for a playermodel. In *Counter-Strike, Global Offensive*, you are most likely to see an enemy when they are looking straight at you, ready to fight. Therefore, the R-Table is created for a player model that is oriented looking straight at the player.

First, an image of the playermodel is put through our edge detector. Then, an R-Table is created, and the r, phi, and gradient orientation values for every pixel in the edge map are added to the R-Table.

|  |
| --- |
| *Figure 3.5.1 - Generating an R-Table based off of an edge map of a playermodel.* |

Finally, as validation, the player model is re-drawn using the data stored in the R-Table.

|  |  |  |
| --- | --- | --- |
| *Figure 3.5.2 - Original image of a playermodel.* | *Figure 3.5.3 - Edge map of the playermodel.* | *Figure 3.5.4 - Image of the playermodel re-created using the data stored in the R-Table.* |

# 

# 4.0 Machine Learning Approach

# 

To determine the relative effectiveness of my custom developed player model recognition method, a control method is required to test it against. I have selected a machine learning based convolutional deep neural network (CDNN) as this control method. CDNNs are becoming extremely popular within the sphere of image analysis, particularly on the topic of object recognition, and are currently being tried out in new applications. The specific machine learning algorithm that I will use is You Only Look Once, Version 3 (YOLOv3). YOLOv3 is open source and one of the most implemented computer vision algorithms of all time. In addition, it can be trained in MATLAB, which is where I did the majority of development for this project.

Before I started, I needed to outline some ground constraints for the algorithm. To maximize effectiveness while still maintaining its usefulness, I settled on the below constraints:

1. *The algorithm would be limited to a single map (location in the game). All of the training images and eventual testing occurred on the map Dust II.*
2. *The algorithm would be trained on both types of playermodels (CT and T). These playermodels have different textures and colors, but are the same shape and size. Training on both types allows the algorithm to be effective against either team.*
3. *A realistic dataset would be used to train the algorithm. The algorithm would not be trained on thousands of images; the dataset needs to be something that one person can realistically collect in a short period of time.*

I began training the algorithm by capturing 116 images of CS:GO player models at various orientations, light levels, and distances from the player. *Figure 4.0.1* shows an example image.

|  |  |
| --- | --- |
| *Figure 4.0.1. Example image captured in game.* | *Figure 4.0.2. Example image labeled; the player model resides inside the box.* |

Then, each image was labeled so that the YOLOv3 algorithm knows which part of the image contains the player model. The open-source software OpenLabeling was used for this task. *Figure 4.0.2* shows an example of a labeled image.

Finally, once all 116 images are labeled, they may be loaded into the YOLOv3 algorithm for training. To train the model, I used a MATLAB script. This script can be found in the file ‘CS.m’. Although there was a lot of tedious preparation, mostly translating the data describing the locations of each label on the images between different formats and data types, the training process is straightforward. After around 4 hours of training on an 8-core CPU, the model was ready to test. Note that the training time could be cut down if the model was trained on a GPU, but processing the 116 training images failed when my GPU ran out of VRAM (8 GB).

# 5.0 Comparison of Results

The effectiveness of both approaches are evaluated in a variety of environments. First, 20 test images are processed from a practice map that has all white surroundings. This means that each playermodel has fantastic contrast and is easy to separate from the background. Then, 20 test images are processed from Dust II. Dust II is the environment that the DCNN was trained on and represents a scenario that is true to real gameplay. Finally, 20 test images are processed from Cache. Cache is very different from the environment that the DCNN was trained on.

For all scenarios, a player model was ‘correctly’ identified if:

1. *The generalized Hough transform recognized the player model and pinpointed the predicted center to be somewhere within the true playermodel. This equates to the program being able to ‘aim’ at the player model and theoretically hit in. Only the highest peak above the user-set threshold is used.*
2. *The DCNN recognized the player model and captured both the torso and head of the true playermodel within the boundary box. Only objects detected with a confidence score of 0.60 or higher are considered.*

In each scenario, 10 of the 20 test images do not contain a playermodel, and the other 10 test images do contain a playermodel. This means that both methods must recognize whether a player model is present or not, as well as determine where the player model is in the frame. To measure effectiveness, two metrics are employed. They are defined by Slodownik (2001) and adapted for this situation as follows:

| ***Sensitivity =*** | ***Specificity =*** |
| --- | --- |

Sensitivity represents how effective the algorithm is in situations where there is a player model in the frame. Specificity represents how effective the algorithm is in situations where there is not a player model in the frame.

Another important note is that the edge mapping threshold and generalized Hough transform peak threshold were adjusted manually to find an appropriate value on a per-map basis. The parameters used were determined through trial and error to allow the transform to have the best true positive and true negative rates. The complete parameters are shown below.

|  | **White** | **Dust II** | **Cache** |
| --- | --- | --- | --- |
| **Edge Mapping Threshold** (*thresh*) | 30 | 50 | 50 |
| **Generalized Hough Transform Bin Size** (*pixels\_per\_bin*) | 9 | 9 | 9 |
| **Generalized Hough Transform Peak Threshold** (*threshold*) | 50 \* *pixels\_per\_bin*  *=*  50 \* 9  *=*  450 | 111 \* *pixels\_per\_bin*  *=*  111 \* 9  *=*  999 | 115 \* *pixels\_per\_bin*  *=*  115 \* 9  *=*  1035 |
| *Table 5.0.1 - Generalized Hough transform parameters used for each map.* | | | |

To ensure that I wasn’t missing any false positives (the generalized Hough transform identifies a player model where there shouldn’t be one), I confirmed the exact location of the detected peaks (if any) for each test image. A detected peak on an image that does not contain a player model, for example, was proof of a false positive.

And, as a final note, all test images were 950x520 pixel resolution. This is exactly ¼ resolution of a full 1920x1080 screen capture.

## 5.1 White Training Map

|  |  |  |
| --- | --- | --- |
| *Figure 5.1.1 - Test image from the white training map. Horizontally cropped to better fit into this figure.* | *Figure 5.1.2 - Player model correctly identified by generalized Hough transform.* | *Figure 5.1.3 - Player model correctly identified by DCNN.* |

There were no false positives for either method. The DCNN failed to detect a playermodel in 3 out of the 10 test images that contained player models.

|  | **Generalized Hough Transform** | **DCNN** |
| --- | --- | --- |
| **Sensitivity (Player Model Present)** | 10 / 10 | 7 / 10 |
| **Specificity (Player Model Not Present)** | 10 / 10 | 10 / 10 |
| **% Accuracy (Overall)** | 100 % | 85 % |
| *Table 5.1.1 - Results from the white practice map.* | | |

## 

## 5.2 Dust II

|  |  |  |
| --- | --- | --- |
| *Figure 5.2.1 - Test image from the map Dust II. Horizontally cropped to better fit into this figure.* | *Figure 5.2.2 - Player model correctly identified by generalized Hough transform.* | *Figure 5.2.3 - Player model correctly identified by DCNN.* |

In the case of false negatives, both methods simply failed to recognize any player model on the screen. In the case of false positives, the location of a potential player model was placed incorrectly in the frame. Some examples of false positives are shown below.

|  |  |
| --- | --- |
| *Figure 5.3.4 - DCNN false positive; gray circle in top right corner.* | *Figure 5.2.5 - DCNN false positive; yellow label in bottom left corner.* |

|  | **Generalized Hough Transform** | **DCNN** |
| --- | --- | --- |
| **Sensitivity (Player Model Present)** | 3 / 10 | 6 / 10 |
| **Specificity (Player Model Not Present)** | 8 / 10 | 9 / 10 |
| **% Accuracy (Overall)** | 55 % | 75 % |
| *Table 5.2.1 - Results from the map Dust II.* | | |

## 5.3 Cache

|  |  |  |
| --- | --- | --- |
| *Figure 5.3.1 - Test image from the map Cache. Horizontally cropped to better fit into this figure.* | *Figure 5.3.2 - Player model correctly identified by generalized Hough transform.* | *Figure 5.3.3 - Player model correctly identified by DCNN.* |

As was included in the section on *Dust II*, some examples of false positives are shown below.

|  |  |
| --- | --- |
| *Figure 5.3.4 - DCNN false positive; gray circle in top right corner.* | *Figure 5.3.5 - DCNN false positive; yellow label in bottom left corner.* |

|  | **Generalized Hough Transform** | **DCNN** |
| --- | --- | --- |
| **Sensitivity (Player Model Present)** | 5 / 10 | 7 / 10 |
| **Specificity (Player Model Not Present)** | 9 / 10 | 9 / 10 |
| **% Accuracy (Overall)** | 70 % | 80 % |
| *Table 5.3.1 - Results from the map Cache.* | | |

## 5.4 Results Utilizing Segmentation

Segmentation can also be used in conjunction with the generalized Hough transform to better identify the precise boundary of each detected player model. However, for the segmentation and connected component labeling to function properly, there must be a completely closed edge around the player model. In scenarios with complex and noisy edge maps, such as on Dust II and Cache, the edge around each player model is often not fully closed. As a result, segmentation cannot be applied in those cases.

However, on the white practice map, the segmentation performs much better. Shown below are examples of segmentation being used in conjunction with the generalized Hough transform to highlight the entire detected player model.

|  |  |
| --- | --- |
|  |  |
| *Figures 5.4.1-5.4.4 - Labeled player models at various distances.* | |

# 6.0 Discussion and Conclusion

Overall, both the generalized Hough transform and the DCNN are both effective methods of locating player models inside of gameplay frames, but they work best in different environments. In the scenario with the all white background, the generalized Hough transform was able to easily detect edges and pinpoint where the player model was in the frame. This allowed it to detect the player model with 100% accuracy. The DCNN, on the other hand, was not trained in this environment and failed to recognize several of the player models. The generalized Hough transform performed better than the DCNN in this situation.

On Dust II, the environments were more complicated, and this degraded the performance of the generalized Hough transform. The added complexity to the edge maps inflates the entries into the accumulator array and makes background features more prominent in the detected peaks. The DCNN performed better than the generalized Hough transform in this situation.

Cache presented a similar problem where background details were sometimes mistaken for player models. However, the map features better overall contrast (brighter backgrounds) and the generalized Hough transform performed better than on Dust II. However, the DCNN still slightly outperformed the generalized Hough transform in this situation.

In summary, the generalized Hough transform works best on images with simple backgrounds and high contrast, while the DCNN works best on images similar to the training environment. The generalized Hough transform is a viable and even superior method of player model detection in the white practice map in *Counter-Strike: Global Offensive*.

# 7.0 Further Research

One avenue for further research is exploring different player model orientations. Although the most common orientation is looking straight at the player (as tested in this project), there are some scenarios where the player model could be facing to the side. Creating multiple R-Tables to cover multiple possible orientations would increase the robustness of this approach.

In addition, an intriguing next step is to run the generalized Hough transform in real time. Running the transform in real time would allow the user to actually apply the player model recognition in live gameplay. However, in its current form, the generalized Hough transform does not run fast enough for real time processing. Processing a single frame takes between 2 and 15 seconds on my computer, depending on the specific image. Using low resolution images, extremely bare edgemaps, and a small number of possible scaling factors all increase execution speed.

# 8.0 Reproducibility

To reproduce my results, please refer to *main.m*. This calls all of the appropriate functions to run the scale-invariant generalized Hough transform on an input image. You can change the input image by editing the file being opened on line 56. Alternatively, change the file path on line 48 to the folder storing your test images.

|  |
| --- |
| *Figure 8.0.1 - Area where image is loaded in main.m.* |

By default, the code will use an R-Table describing a Counter-Terrorist model from *Counter-Strike: Global Offensive*. To generate a different R-Table, please see *Generate\_R\_Table.m*. In this file, you can load in any image of your choice and it will generate an R-Table. Note that high contrast images containing the image of interest against a white background perform best.

|  |
| --- |
| *Figure 8.0.2 - Area where image is loaded in Generate\_R\_Table.m.* |

Depending on the contrast, overall brightness, and complexity of the image, different parameters may need to be changed to achieve favorable results using the generalized Hough transform. To start with, adjust *threshold* on line 67 in *main.m* to the point where the output edge map captures the entire player model but the minimum amount of background noise.

In addition, adjust *thresh* on line 90 in *main.m*. If too many peaks are being detected, then the algorithm runs slower and *thresh* should be increased. If no peaks are being detected, decrease this value.

# 9.0 References and Acknowledgements

D.H. Ballard, *Generalizing the Hough transform to detect arbitrary shapes*, Pattern Recognition, Volume 13, Issue 2, 1981, Pages 111-122, ISSN 0031-3203, https://doi.org/10.1016/0031-3203(81)90009-1.

F. Jurie and C. Schmid, *Scale-invariant shape features for recognition of object categories*, CVPR 2004.

Trevethan R. *Sensitivity, Specificity, and Predictive Values: Foundations, Pliabilities, and Pitfalls in Research and Practice*. Front Public Health. 2017 Nov 20;5:307. doi: 10.3389/fpubh.2017.00307. PMID: 29209603; PMCID: PMC5701930.

Słodownik, B. (2021, May 26). *How your reaction time decreases as your rank increases*. GGPredict sp. z o.o. Retrieved April 29, 2023, from https://ggpredict.io/blog/how-your-reaction-time-decreases-as-your-rank-increases.