uzh | eth | zürich

Controlling a Car with an aVLSI Motion Detection Chip

Semester thesis

Roger Jäggi

Supervisors: Rico Möckel
Dr. Shih-Chii Liu

30th September 2007

Contents

List of Figures

1

Introduction

1.1 Motivation

We use the outputs of a new motion detection chip [**?**] to control a simple racing game. The controller is based on a single layer perceptron. We demonstrate the static and dynamic characterizations of the motion detection chip in a well controlled environment.

With a racing game for the motion detection chip we give a realistic impression to human users in a game like situation about the strengths of this new kind of technology. In particular, we create an environment to demonstrate properties like real time capabilities, invariance to low contrast stimuli and spacial and temporal frequencies. Furthermore allows us the game environment to test different and maybe more complex and accurate controller in the future.

1.2 Overview

In the following chapters the entire project is described. Chapter 2 gives an overview and introduction to the motion detection chip. Furthermore it describes the static and dynamic properties of the chip. Chapter 3 describes the racing game and its implementation. Chapter 4 is about the controller and the used neural network. Chapter 5 explains how learning is done with the controller presented in the preceding chapter. Finally, chapter 6 lists some conclusions and further thoughts on the work and proposes some improvements. Appendix A is a user manual for the racing game. It explains how to calibrate the motion detection chip and how to start a competition between a human player and the chip. Appendix B consists of the most important functions.

2

Motion Detection Chip

2.1 Introduction

We use the motion detection chip described in [?]. This motion detection circuit has 24 motion pixels arranged in a 1-dimensional array. Basically a contrast edge falling on a photoreceptor results in a voltage change. The chip outputs a high gain, high-pass filtered (to remove the effects of the refresh rate of displays) version of the photoreceptor signal. The possible output values of a pixel are in the range [-127, 127]. The value is negative for objects moving from the right to the left and positive for left to right. The chip has a serial port. For this semester thesis, a serial-to-USB adapter is used to connect the chip with a laptop.

2.2 Static Experiments

The first static experiment is to measure the output of one single motion pixel for a drifting sinus stimulus from the right to the left. Because of the chip design, the output must oscillate between two values. Figure 2.1 shows the output of motion pixel 12 for a stimulus drifting with approximately $30°/s$.

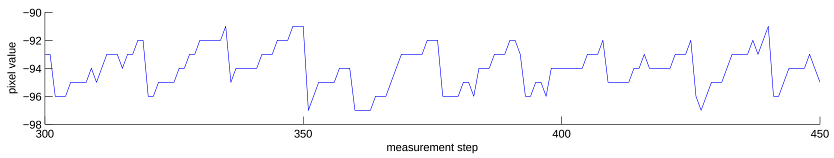

Figure 2.1: Output of pixel 12 for a drifting sinus stimulus at $30°/s$.

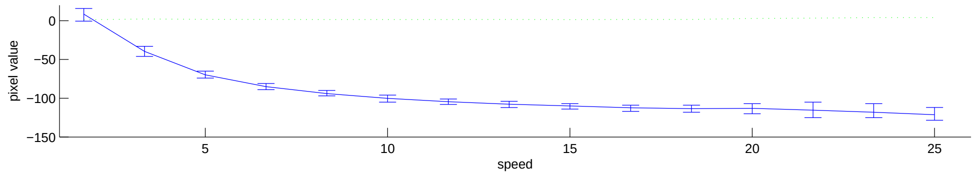

Figure 2.2: Output of pixel 12 for a drifing sinus stimulus at different speed levels. Error bars showing maximum and minimum motion values. The green, dotted line is the standard deviation (300 measurements at each speed level).

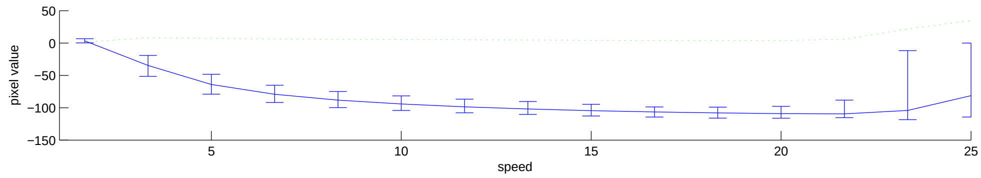

Figure 2.3: Summarized output of all motion pixels. The minimum and maximum bars refer to the minimum and maximum averages of the pixels. The green, dotted line is the standard deviation.

In the second static experiment, the speed of the stimulus is varied from 0 to $85°/s$. Figure 2.2 shows again the output of pixel 12. The error bars show the minimum and maximum values of 300 measurements at each speed level. The second, dotted line is the standard deviation.

Figure 2.3 shows a similar information as figure 2.2. Instead of showing the average value of a single pixel, the average value of all pixel is plotted. The minimum and maximum bars refer to the average value of each pixel. The green, dotted line is again the standard deviation. For greater speed levels, the output value of some pixels is zero. The reason for this effect is the high-pass filter of the motion detection chip. Instead of returning a value nearby -127, the chip returns the filtered value 0.

Figure 2.4 illustrates the information shown in figure 2.2 for all pixels in one 3 dimensional plot. It becomes visible that for some pixels the motion value is wrong for high speed.

2.3 Dynamic Experiments

The next two dynamic experiments are to verify the characteristics of the motion chip on abrupt and on slow changes of the stimulus speed. For both experiments, the stimulus is drifting from the right to the left.

In figure 2.5 is the speed after approximately 225 datasets doubled. The observable effect is a very fast adaptation without oscillations as expected. The same happens for the other way around.

The last experiment is very similar to the previous. The speed is now not abrupt changed but step-wise. The reaction of the chip is very accurate as it should be.

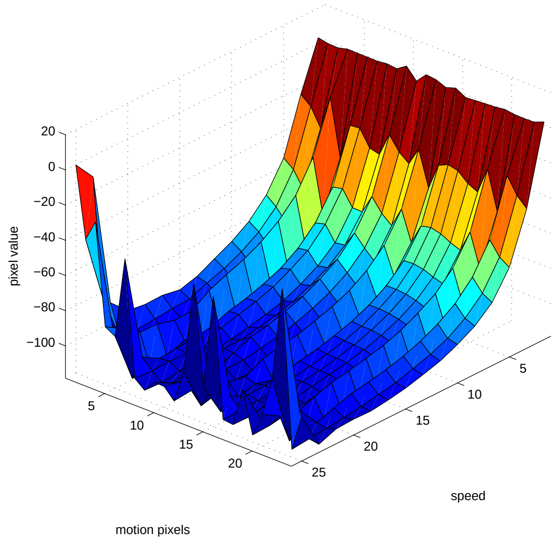

Figure 2.4: The outputs of all pixels for a drifting sinus stimulus at different speed levels.

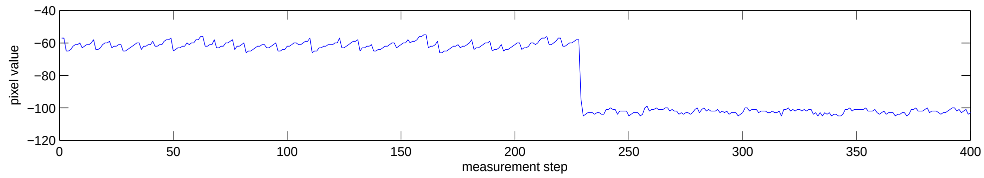

Figure 2.5: The stimulus speed is abrupt doubled. The motion pixel adapts to the new speed without oscillation.

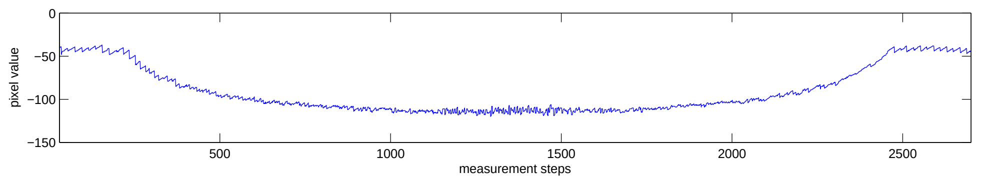

Figure 2.6: Speed is step-wise adjusted. The reaction of the motion pixel is accurate.

3

Racing Game

3.1 Idea

We want to give a more realistic impression to human users about the strengths of the motion detection chip. With a computer game we are able to demonstrate the properties of the chip in a dynamic but still well controlled environment.

The implemented game is a simple racing game where one or two players have to absolve a racing course. To avoid synchronization problems the players absolve the course consecutively and not concurrently. Each course is randomly generated before the game starts, but it is possible to create and choose one in advance. This is especially useful for debugging the learning behavior of the controller presented in chapter 4. The car can only shift vertically and as soon as it touches the roadside, the speed is reset to zero and the car is moved to the middle of the street. Finally, each crash adds one second to the final time. The player with the lower total time wins the competition.

To use the chip to control the car it must be in front of the display, pointing to the end of the street. In figure 3.1 on the left side is shown how the visible area of the chip should look like. The distance between chip and display has to be around 15 - 20 centimeter. See appendix A for more information about positioning and calibrating the chip.

If the chip is well positioned, the visible game scene is transformed into motion vectors as illustrated in figure 3.1 on the right side. This information is converted by the controller into a control signal for the racing car. After a learning phase, the controller can detect the middle of the motion field and aim for it. While a human player can change the speed of the car, the controller simply tries to speed up until a predefined speed limit is reached.

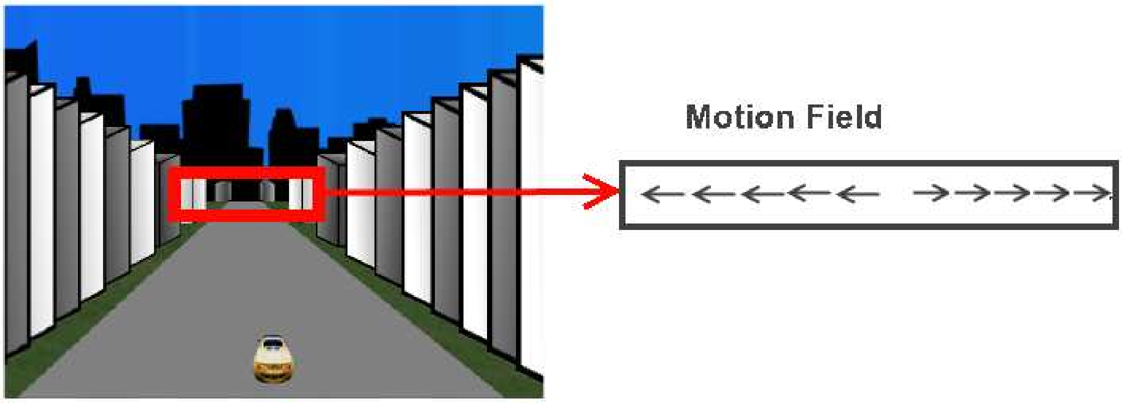

Figure 3.1: Visible area of the motion detection chip and what it really "sees".

3.2 Environment

The first step was to decide about the environment and programming language. The following points were crucial for this decision:

- The visual stimuli have to be accurate.
- The environment needs a simple but powerful plotting engine to display data for debugging and visualisation.
- It must be possible to deal easily with vectors and matrices.
- Serial port support is needed.
- It must be possible to implement an appropriate realistic racing game engine.

Based on this prerequisites we have chosen Matlab and Psychtoolbox[1]. Psychtoolbox is a set of Matlab functions for vision research and freely available under the GPL-2 licence[2]. It allows one to create and control visual stimuli within Matlab. For performance reasons, this Matlab extension is written in pure C and uses OpenGL to display visual stimuli. Furthermore, there are some very useful functions to handle user input and playing sound. Nevertheless it offers only a limited function set for graphical operations and the performance is not comparable to a pure OpenGL application.

To understand the program flow, it is important to know roughly how Psychtoolbox works. Psychtoolbox uses a double buffered drawing model: there is a visible frontbuffer and and invisible backbuffer. While the frontbuffer is drawn to the screen, the backbuffer is located in the video RAM of the system. All drawing operations are done in the invisible backbuffer. As soon as the $flip$ command is executed, the system replaces the frontbuffer with the content of the backbuffer. The timing of this $flip$ command depends on the framerate of the system. On a TFT the framerate is 60Hz and leads to a flip interval of $1/60 = 0.016667$ seconds. It is important to realize that the game engine must execute all calculations and operations of one update step within this interval - otherwise Psychtoolbox is not able to keep the timing conditions and the screen can flicker. A flickering screen can lead to unusable outputs from the motion detection chip.

[1] http://psychtoolbox.org/
[2] http://www.gnu.org/licenses/gpl-3.0.txt

Listing 3.1: A simple hello world for Psychtoolbox

```

% function to initialize Psychtoolbox
windowPointer = Screen('OpenWindow');

% load texture
img = imread('images/image.png');
texture = Screen('MakeTexture', windowPointer, img);

% get flip interval
framerate = Screen('NominalFramerate', windowPointer);
flipInterval = 1 / framerate;

% do initial flip
vbl = Screen('Flip', windowPointer);

% animation loop
while 1
  % put texture into invisible backbuffer
  Screen('DrawTexture', windowPointer, texture, [300, 600, 350, 450]);
  % write a red 'Hello World'
  Screen('DrawText', windowPointer, 'Hello World!', 300, 300, [255, 0, 0]);

  % flip back and frontbuffer, this command waits until vbl+flipInterval
  vbl = Screen('Flip', windowPointer, vbl+flipInterval);
end;

% close Psychtoolbox environment
Screen('CloseAll');
```

3.3 Implementation

This section explains the concrete implementation of the game. The full source code is available in appendix B.

3.3.1 Program Flow

Figure 3.2 shows how the game is implemented. There are only three main files: startGame.m, gameEngine.m and showFinalScreen.m.

startGame.m

startGame.m is the main function of the game. Input parameter handling is done here. After that, the starting script must perform some initialization steps:

- **Load map**: It is possible to create a map in advance. Then this map is loaded. Otherwise a new map is created with default parameters. How exactly a map is created is explained in detail in chapter 3.3.3.
- **Init Psychtoolbox**: Creates a new window for Psychtoolbox and sets some parameter.

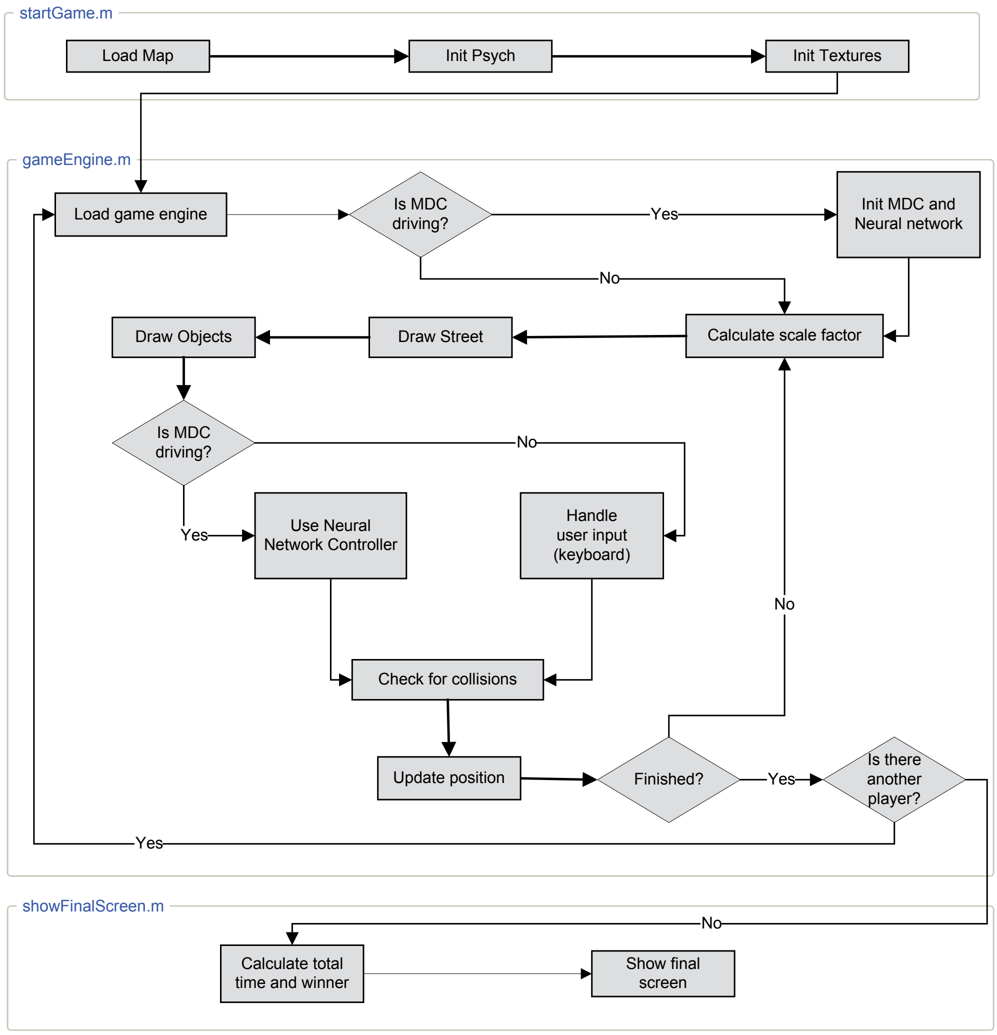

Figure 3.2: Program Flow

- **Init textures**: Because of performance issues, there are only 4 different objects: one black and one white colored house for each side of the street.

After these steps, the game engine is started with the appropriate parameters and finally showFinalScreen.m is called.

gameEngine.m

gameEngine.m is the actual racing game. All computational and graphical stuff during a game is done in this function. It must be called from startGame.m.

1. **Init game engine**: Sets a lot of engine and helper variables and loads remaining textures like the skyline, the car or the grassland. This is done only once.
2. **Init motion detection chip**: If there is a motion detection chip, the chip is initialized (this is basically just a $fopen$ command as known from C) and the neural network is set up. See chapter 4 for more information about the neural network.
3. **Draw street**: Drawing the street is the most expensive operation. The width of the street depends on its position - the street is wider in the foreground than in the background.
4. **Draw objects**: Currently there are only two types of objects in the game (beneath the car): black and white houses on both sides of the street. As the street, their scaling factor depends on their position. To scale the objects, the same scaling factor as for the street is used.
5. **Use neural network controller or handle user input**: After the game environment is drawn, either the controller or the user input is used to navigate the car. The controller is explained in full detail in chapter 4.
6. **Check for collisions**: The game engine has to check if there was a collision. Was this the case, speed is set to zero and the car is moved to the middle of the street. Moreover, an internal counter is increased to be able to compare two players when the game is over.
7. **Update position**: Now all information is collected to be able to update the current position. A game is over as soon as the driver has reached the finishing line.

3.3.2 Contrast Adjustment

To be able to demonstrate the constancy of the motion detection chip against contrast, a built-in contrast adjustment method is used. It is possible to control the contrast of all textures within the game with a simple start-up parameter.

Contrast adjustment is implemented as presented in [3]:

> In this process, pixel values below a specified value are mapped to black and pixel values above a specified value are mapped to white. The result is a linear mapping of a subset of pixel values to the entire range of display intensities (dynamic range).

The contrast can now be lowered by setting the pixel values above the upper threshold equal to the threshold and vice versa for the pixels below the lower threshold. Figure 3.3 illustrates this.

[3] http://matlab.izmiran.ru/help/toolbox/images/displa26.html

The implementation is very simple, it just iterates over all pixel values and sets them to the calculated threshold if necessary:

Listing 3.2: Function to adjust contrast

```
function [ result ] = adjustContrast( img, contrast )

contrast = 1 - contrast;
result = img;

upper_threshold = 255 - 255 * contrast / 2;
bottom_threshold = 255 * contrast / 2;
s = size(result);
for k = 1:s(3)
    for i = 1:s(1)
        for j = 1:s(2)
            if result(i, j, k) < bottom_threshold
                result(i, j, k) = bottom_threshold;
            elseif result(i, j, k) > upper_threshold
                result(i, j, k) = upper_threshold;
            end;
        end;
    end;
end;
```

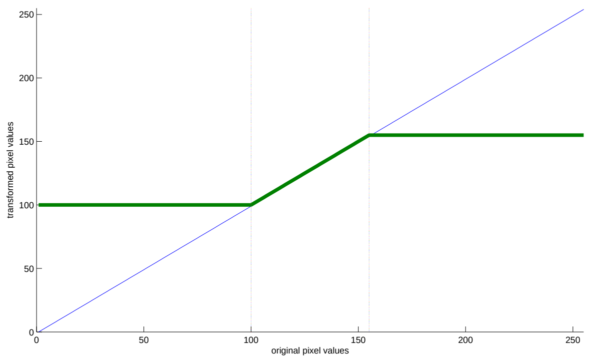

Figure 3.3: The contrast factor controls the size of the window. A higher value results in a smaller window size. The small blue line is the original pixel value, the bold green line the resulting value after contrast adjustment.

3.3.3 Racing Course

The most important part of a racing game is the racing course the player has to absolve. To simplify, the racing course is not a circuit but a route with distinct departure and destination. Therefore, as soon as the player has reached the destination point, the game is over.

The racing course is defined by an array of tuples $street = (dev, pos)$ where dev is a random deviation in pixels from the vertical middle of the screen at position pos. After the points have been created, the Matlab function $interp1$ is used to create a smooth and nice looking course:

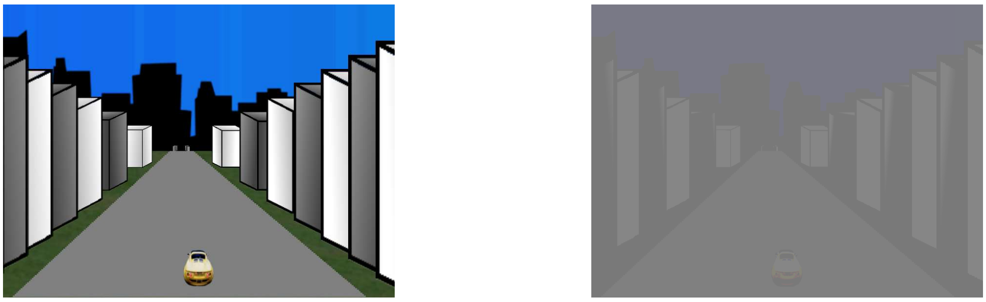

Figure 3.4: (a) contrast is set to 1.0. (b) The same scene again has been adjusted to 0.05

$$course \;=\; interp1(street(:,2),\; street(:,1)',\; x,\; 'spline');$$

It is possible to save the racing course on disk. The length and the maximum deviation of a course is changeable over parameters. A higher deviation results in a curvy course.

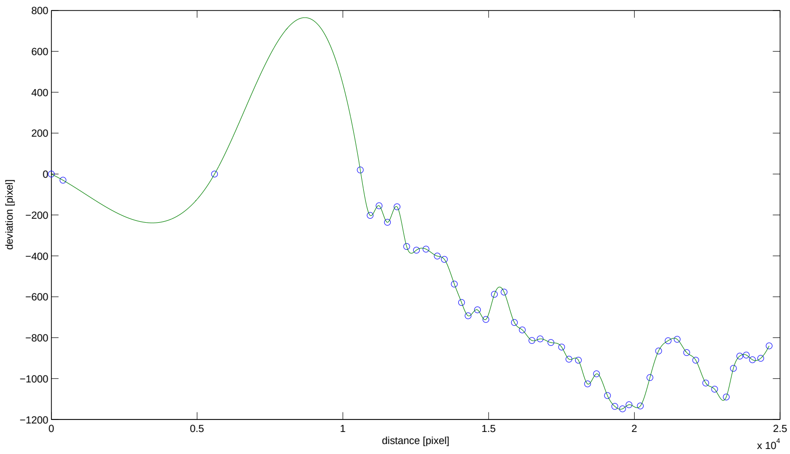

Figure 3.5: A possible race course with a maximum deviation of 80 pixels between two fix points.

3.3.4 Adaptation of Performance

The most expensive part of updating the screen is drawing the street. The game engine takes the course array presented in section 3.3.3 and draws for each visible point a line on the screen. With line depth one, this results in about 300 lines per frame. This can be to much for older systems. Therefore before starting the game engine, a script is executed which measures how many lines the system can draw in a given amount of time. This value is afterwards used to define the thickness of a single street line. For example if the system is capable of drawing 150 lines per interval, the line thickness has to be set to two.

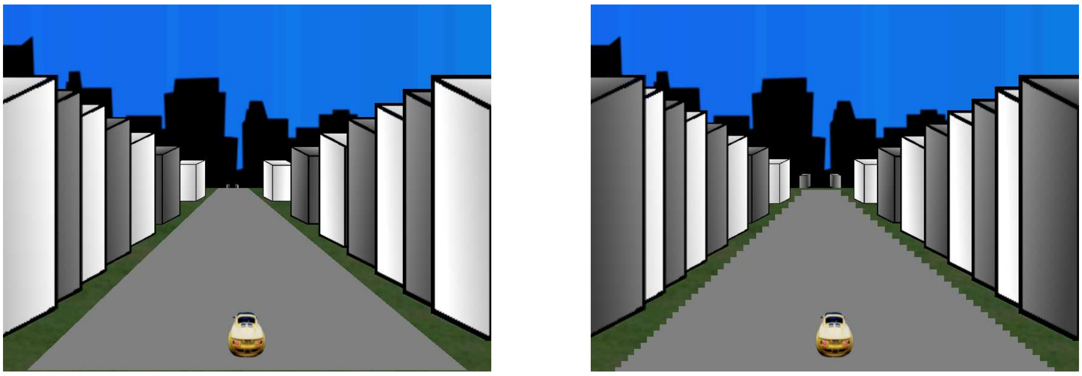

Figure 3.6: The same scene with a 1-pixel and a 10-pixel resolution.

Listing 3.3: Measure system performance

```
function resolution = measurePerformance(w, visibleRect)

    LENGTH_OF_VISIBLE_STREET = visibleRect(4) - visibleRect(2) - 150;
    black = BlackIndex(0);
    currentTime = 0; count = 0;
    maxTime = 0.006;

    tic
    while currentTime < maxTime
        Screen('DrawLine', w, black, 200, count, 400, count);
        currentTime = toc;
        count = count + 1;
    end;
    resolution = ceil(LENGTH_OF_VISIBLE_STREET / count) + 1;
```

3.3.5 Collision Detection

The car can only perform left or right shifts. It is not possible to rotate or to move forward or backward in the environment. Therefore, it is very easy to calculate the collision points. The system knows the current deviation within the course and the actual vertical shift. Also known is the street width, which is always equal (because the car can not change its horizontal position). A further simplification is the fact that the car is a rectangle and not a polynomial object - this reduces the possible collision points. Now the collision detection has just to check if one of the four corners is outside the street.

Listing 3.4: Simplified collision detection algorithm

```
% get collisionpoint on the right border of the street
collisionpoint = current_street_deviation + streetwidth / 2;

% check the right side of the car
if (half_carwidth - verticalshift) >= collisionpoint
  collision = 1;
end;
```

```
% get collisionpoint on the left border of the street
collisionpoint = current_street_deviation - streetwidth / 2;

% check the left side of the car
if (- half_carwidth - verticalshift) <= collisionpoint
  collision = 1;
end;
```

4

Controller

4.1 Idea

The motion vectors are not always accurate. There are a lot of possible influences as screen flickering because of performance problems or operation system issues, changing light, shadows and other effects. On the other hand it is very hard to calibrate the motion detection chip perfectly - a deviation of only some millimeters or degrees can have a huge impact. Therefore the controller has to work even if the chip is not perfectly pointing to the middle of the screen.

A perceptron can fusion the sensor data of the motion pixels and weight them to get a reliable control signal for the racing car.

The motion detection chip has 24 pixels arranged in a row. The value of each pixel is in the range $[-127, 127]$ where a negative value indicates a motion flow to the left and a positive value a motion flow to the right accordingly. A value nearby zero indicates that there was no motion.

For a perfectly trained controller the output of the neural network should be equal to the deviation of the street.

4.2 Single Layer Perceptron

A perceptron is a feedforward neuronal network. A single layer perceptron has only input and output neurons - especially are there no hidden units as in multi layer perceptron [1]. The input units are connected through weighted connections to the output units (see figure 4.1). The output is calculated

[1] http://en.wikipedia.org/wiki/Artificial_neural_network

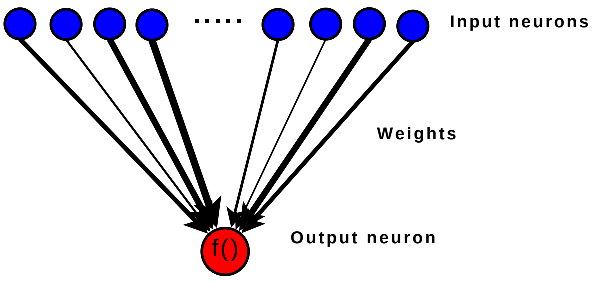

Figure 4.1: A single layer perceptron with different weights and one output neuron.

as follows:

$$output = f(\sum weights * input)$$

where f is an activation function. This can be the non-derivable sign function[2] or better a sigmoid function:

$$f(x) = \frac{1 - e^{-x}}{1 + e^{-x}}$$

A single layer perceptron is a linear classifier - it can only solve linear separable problems. Basically a single layer perceptron with one output unit tries to separate two classes as illustrated in figure 4.2

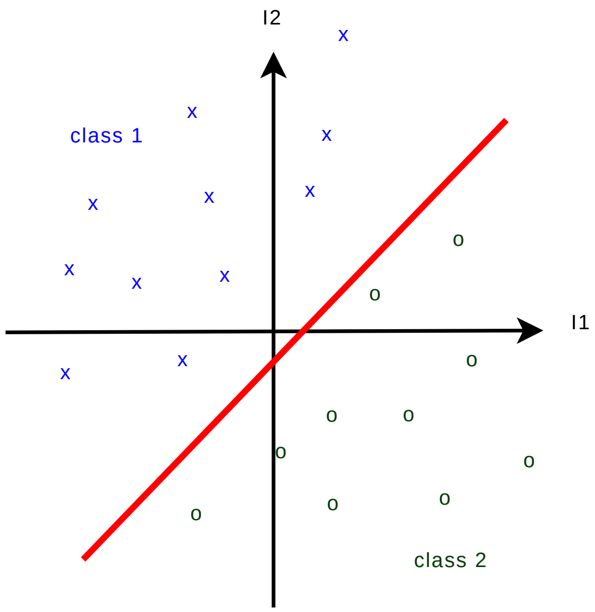

Figure 4.2: A linear classifier on a 2 dimensional input space.

[2]http://en.wikipedia.org/wiki/Sign_function

4.3 Implementation

This section is about the concrete implementation of the single layer perceptron in Matlab. The motion detection chip delivers a dataset all 10 ms roughly. The default flip interval for a TFT is about 16ms. Instead of using each dataset to perform an individual action, a queue buffer is used. The mean value of all datasets in the buffer becomes the input for the perceptron. The buffer is a FIFO queue[3], therefore the oldest value is always replaced by the newest one. With this technique, the controller does not depend to strong on the sample interval of the motion detection chip. And furthermore does this averaging flatten the movement of the car, especially if there are external error sources.

The output of the perceptron and the current speed are used to calculate the vertical shift:

$$\begin{aligned} \alpha &= 0.1 \\ shiftfactor &= \frac{1 - e^{-\alpha * output}}{1 + e^{-\alpha * output}} \\ shift &= constant * speed * shiftfactor \end{aligned}$$

Another important trick is the scaling of the input. The convergence behavior of a perceptron is better when the input values, weights and output values are approximately in the same range.

Listing 4.1: Fill buffer

```
% get datasets from motion detection chip and divide it through 127 to get a
    value between [-1, 1]
dataset = getMDCData(serialHandle, pixels)/127;
firstBuffer = [firstBuffer; dataset];

if size(firstBuffer) >= 3
  % append only last dataset to second buffer
  secondBuffer = [secondBuffer; firstBuffer(size(firstBuffer))];

  if size(secondBuffer) >= 3
    % calculate average of secondBuffer
    perceptronInput = average(secondBuffer);

    % calculate perceptron output
    output = sum(perceptronWeights .* perceptronInput);

    % calculate vertical shift based on perceptron output
    verticalshift = getVerticalShift(output);
  end;
end;
```

To avoid a jumping behavior of the car, a threshold value is used: if the shift factor is in the range [-threshold, threshold] then it is set to 0. In figure 4.3 is the threshold equal to 0.2.

[3] http://en.wikipedia.org/wiki/FIFO

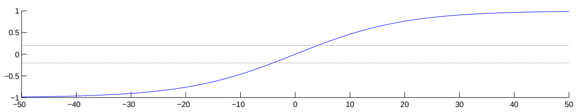

Figure 4.3: The activation function used to calculate the shift factor. Values between the two vertical threshold bars are mapped to zero.

5

Learning

5.1 Idea

The controller must learn the weights in advance. Per default learning is deactivated. To activate it, the startGame.m script needs the additional parameter learningMode = 1. This enables learning if the motion detection chip is connected.

To update the weights, the standard perceptron learning rule is used:

$$weights = weights + \eta * (expectedOutput - output) * perceptronInput$$

The expectedOutput is the known deviation of the street. η is the learning rate and should be very small, for example 0.0002. This rule is applied on every frame if new data is available from the motion detection chip. Figure 5.1 gives an overview over the whole process described in chapter 4 and 5. First the data is collected in a FIFO buffer then the average of this buffer is used to calculate the perceptron output. With this value the weights get updated and the controller uses the vertical shift value to steer the car. This process is repeated until the course is over. If learning mode is not active, the update step is omitted.

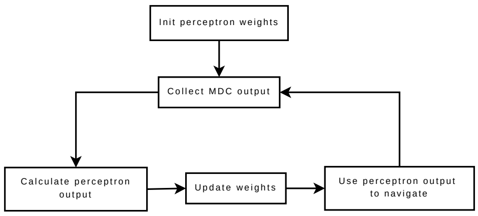

Figure 5.1: Illustrates the controller workflow. The weights get updated every frame.

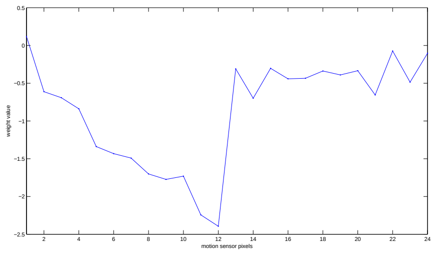

Figure 5.2: Learned weights after 8000 steps.

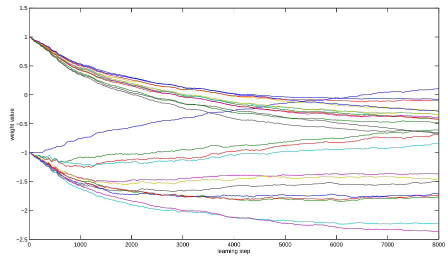

Figure 5.3: The evolution of the weights.

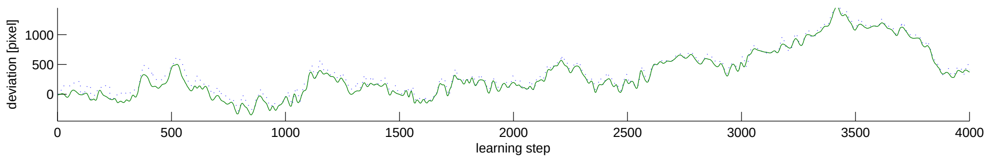

Figure 5.4: Comparison of the dotted perceptron output and the street deviation. For a perfect controller those values should be equal.

5.2 Results

Many measurements have shown, that the choice of initial values is very important for convergence. The learning rate η must be very small, otherwise the weights oscillate. Moreover, the selection of the initial weights is essential. We achieved the best learning behavior with the following initial values:

$$\begin{aligned} pixels &= 24 \\ \eta &= 0.00002 \\ weights &= [(-1 * ones(1, pixels/2)), ones(1, pixels/2)] \end{aligned}$$

Figure 5.2 shows the final weights after 8000 learning steps. The weights of the first five pixels are adjusted towards zero. Also the weights of the last 12 input neurons / pixels are adjusted from 1 to -1. In figure 5.3 the evolution of the weights is plotted.

Figure 5.4 compares the output of the perceptron with the street deviation. For a perfectly trained controller, those two values should be equal. The longer the perceptron is learning, the smaller is the measured difference. But anyway, the difference in 5.5 and the error in 5.4 are not very accurate, because it is not possible to determine the exact position where the motion detection chip is pointing to and ,therefore, only an approximation of the street deviation is possible.

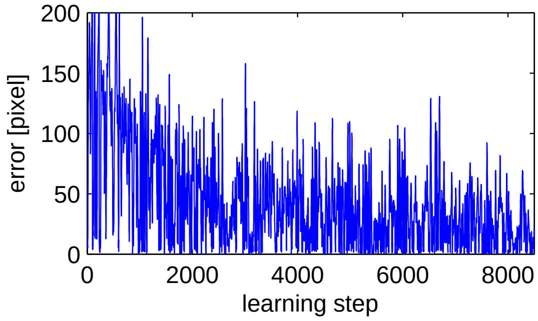

Figure 5.5: The error in pixels as difference between perceptron output and current street deviation.

Finally, figure 5.5 visualizes the error in pixels. In this case the error is defined as the difference between the controller output and the current street deviation. The error is decreasing over time, but still very high. It was not possible to significantly decrease the error with longer race courses nor with different initial parameters. The single layer perceptron is just too weak to learn so many different situations as we have in a curvy street. For a more or less straight street, the error was nearby zero. The more complex the street, the bigger the error.

6

Conclusions

In this semester thesis, we describe the use of a motion detection chip to control a car with a neural network in a simple racing game. The game is implemented with Matlab and Psychtoolbox and uses OpenGL to draw the graphics. We use a single layer perceptron as a controller. The perceptron is able to control the car, but it is not possible to achieve a perfect behavior with this simple neural network. Nevertheless we have provided a basis for further experiments with the motion detection chip. Furthermore we have characterized the chip in a well controlled but still dynamic environment.

The following list summarizes points for future work or improvements:

- The game is still very simple. A better car handling and a realistic physic engine is missing.
- The chip can handle very low contrasts. It is possible to reduce the contrast over a parameter. This feature was not tested because of lack of time.
- An improved controller is necessary for a better car handling. The controller does not consider recent situations. For example, the controller should detect a turn in the road and behave accordingly.

A

Manual

1. Connect the motion detection chip with a serial-to-USB adapter to your computer or laptop.
2. Start Matlab and change the current directory to racinggame/tools.
3. Put the chip about in the middle and 15-20 centimeter in front of the display. The lens has to head for the middle of the display.
4. Execute $twoSidedVertSinStim$ without an argument. Move the chip until you have approximately as much green as red squares. You can change the speed of the stimulus by pressing f or s. Switch the direction of the stimulus by pressing c. To exit press escape.
5. Execute $squareSinStim$ without an argument. Calibrate the chip. The visible area must be slightly above the middle of the screen. To exit press escape.
6. Change the current folder to racinggame/game and create a new racing course by executing

   $$map = makeRaceCourse(100, 80, 1);$$

   The first parameter is the length of the course, the second parameter controls how curvy the course should be and the third one instructs the function to save a copy of the course on disk.
7. Execute $startGame(1, 1, map, 1, 1)$ to start learning. A green square shows the decision of the controller. If there are only green squares in the middle of the street, try to recalibrate the chip.
8. Load the learned weights with the command $load('perceptronWeights')$. Start a real race with $startGame(2, 1, map, 1, 0, perceptronWeights)$. It is possible, of course, to create a new course for the race.
9. It is possible to take a screen shot during the game by pressing s. The screen shot is the saved in screenshot.mat. Write $load('screenshot.mat'); imshow(screenshot);$ to display the image.

B

Matlab Code

Appendix B introduces the most important functions. All other functions and testing code is available on request.

B.1 Game

startGame.m

Listing B.1: Start Game

```
function startGame(numOfPlayers, chipIsDriving, mapname, contrast, learningMode,
    perceptronWeights)

  % set some default arguments
  if nargin < 1
      numOfPlayers = 1;
  end;

  if nargin < 2
      chipIsDriving = 0;
  end;

  if nargin < 3
      % init new map
      [map, maplength] = makeRaceCourse(80, 80, 0);
  else
      % load existing map from filesystem
      load(strcat('maps/', mapname), '-mat');
  end;

```

```
if nargin < 4
    contrast = 1;
end;

if nargin < 5
    learningMode = 0; % 1 == learning
end;

if nargin < 6
  pixels = 24;
    perceptronWeights = [(-1*ones(1,pixels/2)), ones(1, pixels/2)];
end;

try
    KbName('UnifyKeyNames');

    % init Screen
    [w, screenRect, screenNumber] = initPsych();

    % init inner screen
    visibleRect = [800, 600];
    if screenRect(3) < visibleRect(1) || screenRect(4) < visibleRect(1)
        error('Your resolution is too small. Minimal resoultion is 800x600!');
    end;
    x1 = round((screenRect(3) - visibleRect(1)) / 2);
    x2 = screenRect(3) - x1;
    y1 = round((screenRect(4) - visibleRect(2)) / 2);
    y2 = screenRect(4) - y1;
    visibleRect = [x1, y1, x2, y2];

    % load availables enviroment textures, return format of textures: [texture
         width height]
    textures = initTextures(w, contrast, 'textures/1024x768/env/');

    % init elements we want to display
    tmp = fliplr(50:30+round(rand*20):maplength);
    objects = zeros(length(tmp), 2);
    objects(:, 1) = tmp';
    for i = 1: length(tmp)
        objects(i, 2) = mod(i, 2) + 1;
    end;

    % start 1 or 2 player game. In 2 player game, player number one is
    % the motion detection chip, player number two is the human player.
    if numOfPlayers == 1
        [timeP1, collisionsP1, abortedP1] = gameEngine(w, visibleRect,
             screenNumber, map, maplength, textures, objects, contrast,
             chipIsDriving, learningMode, perceptronWeights);
          if abortedP1 == 0 && learningMode == 0
              showFinalScreen(w, visibleRect, screenNumber, timeP1,
                   collisionsP1, 0, 0);
          elseif abortedP1 == 1
              disp ('Player 1 has aborted ... :-(');
          end;
    elseif numOfPlayers == 2
        [timeP1, collisionsP1, abortedP1] = gameEngine(w, visibleRect,
             screenNumber, map, maplength, textures, objects, contrast, 1, 0,
```

```
               perceptronWeights);
          if abortedP1 == 0
              Screen('FillRect', w, BlackIndex(screenNumber), visibleRect);
              Screen('DrawText', w, 'PLAYER 2, press SPACE to continue ...', 300,
                    500, [0, 255, 255, 255]);
              Screen('DrawingFinished', w);
              Screen('Flip', w, 0);

              while 1
                  [keyIsDown, secs, keyCode] = KbCheck;
                  if keyIsDown && keyCode(KbName('space'))
                      while KbCheck; end; % Flush all keyboard events...
                           FlushEvents() seems not to work
                      break;
                  end;
              end;

              [timeP2, collisionsP2, abortedP2] = gameEngine(w, visibleRect,
                   screenNumber, map, maplength, textures, objects, contrast, 0,
                   perceptronWeights);
              if abortedP2 == 0
                  showFinalScreen(w, visibleRect, screenNumber, timeP1,
                       collisionsP1, timeP2, collisionsP2);
              end;
          end;

          if abortedP1 == 1
              disp('Player 1 has aborted ... :-(');
          elseif abortedP2 == 1
              disp('Player 2 has aborted ... :-(');
          end;
      else
          error('Not supported');
      end;

      % close Screen environment
      closePsych();
  catch
      % close Screen environment if there was an error.
      closePsych();
      psychrethrow(psychlasterror);
  end;
```

gameEngine.m

Listing B.2: Game Engine

```
function [endtime, collisionCount, aborted] = gameEngine(w, visibleRect,
     screenNumber, street, tracklength, textures, objects, contrast, chipIsDriving
     , learningMode, perceptronWeights)
try
    % init some constants
    % speed cannot be negative
    MIN_SPEED = 0;
    MAX_SPEED = 80;
    SPEED_STEP = 1;
    % speed for MDC in learning phase
```

```
SPEED_LEARNING_PHASE = 55;
% speed for MDC in race phase, with deactivated learning
SPEED_RACE_PHASE = 70;
VERTICALSHIFT_STEP = 5;
% acceleration of the car. The smaller, the higher the influence of
% crashes on the final time
ACCELERATION = 0.002;
% for scaling. indicates where the zero point is. overhead = 20 means
% that scaling is zero 20 pixels 'after' the drawn scene.
OVERHEAD = 20;
STREET_WIDTH = 140;
% maximum size of objects
MAX_OBJECT_SCALING = 5;
% street resolution. higher for older systems
RESOLUTION = measurePerformance(w, visibleRect);


% is set to 1 if escape is pressed
aborted = 0;

% load default background
imageGras = imread('textures/1024x768/gras.png');
if contrast ~= 1
    imageGras = adjustContrast(imageGras, contrast);
end;
grastex = Screen('MakeTexture', w, imageGras);

% load skyline
imageSkyline = imread('textures/1024x768/skyline.png');
if contrast ~= 1
    imageSkyline = adjustContrast(imageSkyline, contrast);
end;
skylinetex = Screen('MakeTexture', w, imageSkyline);

% load car
[imageCar map alpha] = imread('textures/1024x768/car1.png');
if contrast ~= 1
    imageCar = adjustContrast(imageCar, contrast);
end;
imageCar(:,:,4) = alpha(:,:);
dimCar = size(imageCar);
cartex = Screen('MakeTexture', w, imageCar);

% position of car (usefull for drawing and collision detection)
x1 = (visibleRect(1) + visibleRect(3)) / 2 - dimCar(1) / 2;
y1 = visibleRect(4) - 100;
x2 = x1 + dimCar(1);
y2 = y1 + dimCar(2);
carRect = [x1, y1, x2, y2];

% try to preload textures (usually it works, but performance does not
     increase dramatically - lets do it anyway)
Screen('PreloadTextures', w);

% init perceptron if MDC is driving the car
if chipIsDriving == 1
    % init serial connection
```

```
        serialHandle = serial('COM1', 'BaudRate', 38400);
        fopen(serialHandle);
        % number of pixels, usually 24
        pixels = length(perceptronWeights);
        % learning rate
        ETA = 0.00002;
        % buffers for averaging
        firstBuffer = [];
        secondBuffer = [];
%         perceptronWeights = [-0.9370   -1.6345   -1.2339   -1.0772   -1.4571
     -1.4065   -1.4134   -1.4346   -1.5688   -1.5139   -1.8261   -1.8956    0.2083
      -0.1377    0.0808   -0.0356   -0.0424    0.0496   -0.0196   -0.0178
     -0.3488    0.1620   -0.2876    0.0572];

        % perceptron output before activation function
        output = 0;
        weightHistory = perceptronWeights;
        perceptronOutputHistory = output;
    end;

    % init colors
    white = (WhiteIndex(screenNumber) - BlackIndex(screenNumber)) * (1-contrast)
         / 2;
    black = (WhiteIndex(screenNumber) - BlackIndex(screenNumber)) - white;
    gray = (white + black) / 2;
    if round(gray) == white
        gray = black;
    end

    % query duration of monitor refresh interval:
    framerate = Screen('NominalFramerate', w);
    if (framerate == 0)
        framerate = 60;
    end;
    ifi = 1 / framerate;
    waitframes = 1;
    waitduration = waitframes * ifi;

    % set 'speed' of car. The faster the car is, the more pixels we have to
    % shift between to frames.
    speed = MIN_SPEED;
    shiftPerFrame = speed * waitduration;

    % perform initial Flip to sync us to the VBL and for getting an initial VBL-
         Timestamp
    vbl = Screen('Flip', w);

    % init some game parameter
    verticalshift = 0;
    collisionCount = 0;
    starttime = now;

    % calculate finishing line
    finishline = tracklength - round(carRect(2)) + visibleRect(2) - 30;

    i  = visibleRect(4) - visibleRect(2);
    while i < tracklength
```

```

        % draw background
        Screen('DrawTexture', w, grastex, [], visibleRect);

        % calculate scale factor : scale = a*x + b
        a = -MAX_OBJECT_SCALING / (visibleRect(4) - visibleRect(2) + OVERHEAD);
        b = (MAX_OBJECT_SCALING * (i - 300 + OVERHEAD)/(visibleRect(4) -
             visibleRect(2) + OVERHEAD));

        % draw street
        j = i - visibleRect(4) + visibleRect(2) + 1;
        while j <= (i - 300)
            scale = a * j + b;
            % left side ...
            x1 = (visibleRect(1) + visibleRect(3)) / 2 + street(j) - STREET_WIDTH
                  * scale + verticalshift;
            if x1 < visibleRect(1)
                x1 = visibleRect(1);
            end;
            % right side ...
            x2 = (visibleRect(1) + visibleRect(3)) / 2 + street(j) + STREET_WIDTH
                  * scale + verticalshift;
            if x2 > visibleRect(3)
                x2 = visibleRect(3);
            end;
            y1 = i + visibleRect(2) - j - 1;

            % draw finish line
            if j == finishline || j == (finishline - 1)
                Screen('DrawLine', w, black, x1, y1, x2, y1, RESOLUTION);
            else
                Screen('DrawLine', w, gray, x1, y1, x2, y1, RESOLUTION);
            end;
            j = j + RESOLUTION;
        end;

        % draw skyline
        Screen('DrawTexture', w, skylinetex, [], [visibleRect(1), visibleRect(2),
              visibleRect(3), visibleRect(2) + 300]);

        % draw objects on the left and right side of the street
        sObject = size(objects);
        for j = 1:sObject(1)
            scale = a * objects(j, 1) + b;
            if objects(j, 1) < (i - 300 + OVERHEAD) && objects(j, 1) > (i - 300 -
                  visibleRect(4) + visibleRect(2) - textures(objects(j, 2), 3) *
                 scale)
                % src rectangle - defines the parts of the original texture
                % we want to display.
                srcRectL = [0, 0, textures(objects(j, 2), 3), textures(objects(j,
                      2), 2)];
                srcRectR = [0, 0, textures(objects(j, 2), 3), textures(objects(j,
                      2), 2)];

                % x values are different for objects on the left side and
                % on the right side of the street

```

```
                % lets start on the left side of the street ..
                x2L = round((visibleRect(1) + visibleRect(3)) / 2 + round(street(
                     objects(j, 1))) - STREET_WIDTH * scale + verticalshift);
                x1L = round(x2L - textures(objects(j, 2), 3) * scale);

                % .. and now the right side of the street
                x1R = round((visibleRect(1) + visibleRect(3)) / 2 + round(street(
                     objects(j, 1))) + STREET_WIDTH * scale + verticalshift);
                x2R = round(x1R + textures(objects(j, 2), 3) * scale);

                % handle some special cases
                if x1L < visibleRect(1)
                    srcRectL(1) = (visibleRect(1) - x1L) / scale;
                    x1L = visibleRect(1);
                end;
                if x1R < visibleRect(1)
                    srcRectR(1) = (visibleRect(1) - x1R) / scale;
                    x1R = visibleRect(1);
                end;
                if x2L > visibleRect(3)
                    srcRectL(3) = (visibleRect(3) - x1L) / scale;
                    x2L = visibleRect(3);
                end;
                if x2R > visibleRect(3)
                    srcRectR(3) = (visibleRect(3) - x1R) / scale;
                    x2R = visibleRect(3);
                end;

                % y values are equal for objects on the left and on the
                % right side of the street
                y2 = round(i - objects(j, 1) + visibleRect(2));
                y1 = round(y2 - textures(objects(j, 2), 2) * scale);
                if y2 > visibleRect(4)
                    srcRectL(4) = (visibleRect(4) - y1) / scale;
                    srcRectR(4) = (visibleRect(4) - y1) / scale;
                    y2 = visibleRect(4);
                end;
                if y1 < visibleRect(2)
                   srcRectL(2) = (visibleRect(2) - y1) / scale;
                   srcRectR(2) = (visibleRect(2) - y1) / scale;
                   y1 = visibleRect(2);
                end;

                if y2 < (visibleRect(2) + 300)
                    y2 = visibleRect(2) + 300;
                    y1 = round(y2  - textures(objects(j, 2), 2) * scale);
                end;

                destRectL = [x1L, y1, x2L, y2];
                destRectR = [x1R, y1, x2R, y2];

                % lets draw the objects ..
                if srcRectL(1) < srcRectL(3) && x2 > visibleRect(1) && srcRectL
                     (2) < srcRectL(4) && y2 >= (visibleRect(2) + 300)
                    Screen('DrawTexture', w, textures(objects(j, 2), 1), srcRectL
                         , destRectL);
                end;
```

```
                if srcRectR(1) < srcRectR(3) && x2 > visibleRect(1) && srcRectR
                     (2) < srcRectR(4) && y2 >= (visibleRect(2) + 300)
                    Screen('DrawTexture', w, textures(objects(j, 2), 1) + 2,
                         srcRectR, destRectR);
                end;
            end;
        end;


        % draw car
        Screen('DrawTexture', w, cartex, [], carRect);

        % write some debugging and info output to the screen
        % DrawText is very slow, use with caution!!!!!!!!!!!!!!
        Screen('DrawText', w, strcat('Progress: ', num2str(ceil((i - visibleRect
             (4) + visibleRect(2)) / (tracklength - visibleRect(4) + visibleRect
             (2)) * 100)), '%'), 50, 150, [0, 255, 255, 255]);

        % collision point lines
        carPos = round(i + visibleRect(2) - carRect(2));
        scale = a * carPos + b;
        collPointLeft = (visibleRect(1) + visibleRect(3)) / 2 + street(carPos) -
             STREET_WIDTH * scale + verticalshift;
        collPointRight = (visibleRect(1) + visibleRect(3)) / 2 + street(carPos) +
              STREET_WIDTH * scale + verticalshift;
        % comment out to visualize collision points
%         Screen('DrawLine', w, black, collPointLeft, visibleRect(2),
     collPointLeft, visibleRect(4));
%         Screen('DrawLine', w, black, collPointRight, visibleRect(2),
     collPointRight, visibleRect(4));

        % tell PTB that there are no more drawing actions until next flip..
        Screen('DrawingFinished', w);

        % take screenshot if user presses 's'
        [keyIsDown, secs, keyCode] = KbCheck;
        if keyIsDown
            if keyCode(KbName('s'))
                % Take screenshot of GPU converted image:
                screenshot = Screen('GetImage', w, visibleRect, 'backBuffer');
                save('screenshot');
            elseif keyCode(KbName('Escape'))
                aborted = 1;
                break;
            end;
        end;

        % Flip 'waitframes' monitor refresh intervals after last redraw.
        vbl = Screen('Flip', w, vbl + (waitframes - 0.5) * ifi);

        % controller logic
        if chipIsDriving == 1
            % get dataset
            dataset = getMDCDataNoMemory(serialHandle, pixels);
            % append dataset to first buffer
            firstBuffer = [firstBuffer; dataset];
            s = size(firstBuffer);
```

```
            if s(1) >= 3
                secondBuffer = [secondBuffer; firstBuffer(s(1), :)/127];
                firstBuffer = [];
            end;

            sB = size(secondBuffer);
            if sB(1) >= 3
                perceptronInput = zeros(1, pixels);
                for l = 1:sB(1)
                    perceptronInput = perceptronInput + secondBuffer(l,:);
                end;
                perceptronInput = perceptronInput / sB(1);
                secondBuffer = secondBuffer(sB(1)-1:sB(1), :);

                % use perceptron
                output = sum(perceptronWeights .* perceptronInput);
                center = street(i-400) + verticalshift;
                if learningMode == 1
                    perceptronWeights = perceptronWeights + ETA * (center -
                         output) * perceptronInput;
                end;

                perceptronOutputHistory = [perceptronOutputHistory; output-
                     verticalshift, street(i-400)];
                weightHistory = [weightHistory; perceptronWeights];
            end;

            % shift..
            THRESHOLD = 0.10;
            if activationFunction(output) >= THRESHOLD
                verticalshift = verticalshift - round(VERTICALSHIFT_STEP*speed
                     /25*abs(activationFunction(output)));
                % show decision of perceptron
                Screen('FillRect', w, [0, 255, 0], [975, 100, 1025, 150]);
            elseif activationFunction(output) < -THRESHOLD
                verticalshift = verticalshift + round(VERTICALSHIFT_STEP*speed
                     /25*abs(activationFunction(output)));
                % show decision of perceptron
                Screen('FillRect', w, [0, 255, 0], [375, 100, 425, 150]);
            else
                Screen('FillRect', w, [0, 255, 0], [675, 100, 725, 150]);
            end;

            % maximize speed.
            if learningMode == 1
                if speed < SPEED_LEARNING_PHASE
                    speed = speed + SPEED_STEP;
                end;
                shiftPerFrame = ACCELERATION * speed * speed / 2;
            else
                if speed < SPEED_RACE_PHASE
                    speed = speed + SPEED_STEP;
                end;
                shiftPerFrame = ACCELERATION * speed * speed / 2;
            end;

        else
```

```
            % handle user input
            [keyIsDown, secs, keyCode] = KbCheck;
            if keyIsDown
                if keyCode(KbName('UpArrow'))
                    if speed + SPEED_STEP <= MAX_SPEED
                        speed = speed + SPEED_STEP;
                        shiftPerFrame = ACCELERATION * speed * speed / 2;
                    end;
                elseif keyCode(KbName('DownArrow'))
                    if speed - SPEED_STEP >= MIN_SPEED
                        speed = speed - SPEED_STEP;
                        shiftPerFrame = ACCELERATION * speed * speed / 2;
                    end;
                elseif keyCode(KbName('LeftArrow'))
                    verticalshift = verticalshift + VERTICALSHIFT_STEP;
                elseif keyCode(KbName('RightArrow'))
                    verticalshift = verticalshift - VERTICALSHIFT_STEP;
                end;
            end;
        end;


        % autopilot
%         verticalshift = - street(round(carPos));

        % was there a collision?
        collision = collisionDetection((collPointRight - collPointLeft), street(
             carPos), verticalshift, carRect);
        if collision == 1
            verticalshift = -street(carPos);
            speed = MIN_SPEED;
            shiftPerFrame = speed * waitduration;
            collisionCount = collisionCount + 1;
        end;
        i = ceil(i + shiftPerFrame);
    end;

    endtime = now - starttime;

    if chipIsDriving == 1
        fclose(serialHandle);
    end;
catch
    if chipIsDriving == 1
        fclose(serialHandle);
    end;
    psychrethrow(psychlasterror);
end;

% plot some debuging output
if chipIsDriving == 1
    figure(1)
    plot(perceptronWeights);
    figure(2)
    plot(weightHistory);
    figure(3)
    hold all
```

```
    plot(perceptronOutputHistory(:, 1), 'DisplayName', 'Perceptron Output');
    plot(perceptronOutputHistory(:, 2), 'DisplayName', 'Real Value');
    legend('show');
    hold off
    figure(4)
    plot(abs(perceptronOutputHistory(:, 2)-perceptronOutputHistory(:, 1)),'
        DisplayName', 'Error');
    hold off
    % learned weights
    % use load('perceptronWeights'); to get them
    save('perceptronWeights');
end;
```

makeRaceCourse.m

Listing B.3: Make Race Course

```
function [map, maplength, mapname] = makeRaceCourse( points, max_deviation,
     saveToDisk)
  if nargin < 3
      points = 100;
      max_deviation = 50; % from 0 to 150 makes sense
      saveToDisk = 1;
  end;

  tmpMap = zeros(points, 2);

  devx = 0;
  tmpMap(1, :) = [devx, 0];
  tmpMap(2, :) = [0, 400]; % lets start with a straight road
  maplength = 400;
  for i = 3:points
      deltax = round(randn*max_deviation);
      devx = devx + deltax;

      maplength = maplength + round(rand * 200) + 200;
      tmpMap(i, :) = [devx, maplength];
  end;

  x = 0:1:maplength;
  map = interp1(tmpMap(:, 2)', tmpMap(:, 1)', x, 'spline');

  mapname = '';
  if saveToDisk == 1
      numOfMaps =  length(dir('maps/')) - 2; % ignore '.' and '..'
      digits = floor(log10(numOfMaps) + 1);
      if digits == -Inf
          number = '0000';
      else
          number = num2str(numOfMaps);
          for i = 1:(4-digits)
              number = strcat('0', number);
          end;
      end;
      mapname = strcat('map', number, '.mat');
      filename = strcat('maps/map', number, '.mat');
      save(filename, 'map', 'maplength');
```

```
    end;

    % debugging output;
    if saveToDisk == 1
        plot(tmpMap(:, 1)’, tmpMap(:, 2)’, ’o’, map, x) % show profile
    end;
```

measurePerformance.m

Listing B.4: Measure Performance

```
function resolution = measurePerformance(w, visibleRect)
    LENGTH_OF_VISIBLE_STREET = visibleRect(4) - visibleRect(2) - 150;
    black = BlackIndex(0);
    currentTime = 0; count = 0;
    maxTime = 0.006;

    tic
    while currentTime < maxTime
        Screen(’DrawLine’, w, black, 200, count, 400, count);
        currentTime = toc;
        count = count + 1;
    end;
    resolution = ceil(LENGTH_OF_VISIBLE_STREET / count) + 1;
```

adjustContrast.m

Listing B.5: Adjust Contrast

```
function [ result ] = adjustContrast( img, contrast )

    contrast = 1 - contrast;
    result = img;

    upper_threshold = 255 - 255 * contrast / 2;
    bottom_threshold = 255 * contrast / 2;
    s = size(result);
    for k = 1:s(3)
        for i = 1:s(1)
            for j = 1:s(2)
                if result(i, j, k) < bottom_threshold
                    result(i, j, k) = bottom_threshold;
                elseif result(i, j, k) > upper_threshold
                    result(i, j, k) = upper_threshold;
                end;
            end;
        end;
    end;
```

B.2 Tools

TwoSidedVertSinStim.m

Listing B.6: Two Sided Vertical Stimulus

```
function twoSidedVertSinStim( p, visiblesize, cyclespersecond, contrast )
```

```
% Show a two sided vertical stimulus which is move- and resizeable.
%   Use the following keys to change the stimulus:
%     f/s : speed up / slow down the speed of the stimulus
%     c : swich stimulus direction
%     escape : quit program

if nargin < 1
    p = 128;
end;

if nargin < 2
    visiblesize = 2048;
end;

if nargin < 3
    cyclespersecond = 500;
end;

if nargin < 4
    contrast = 1;
end;

try
    % enable all key names
    KbName('UnifyKeyNames');

    % init PsychToolbox
    [w, screenRect, screenNumber] = initPsych();

    % set constants
    SPEEDSTEPSIZE = 5;

    % init MDC
    serialHandle = serial('COM1', 'BaudRate', 38400);
    fopen(serialHandle);
    pixels = 24;
    MDCData = zeros(1, pixels);

    % get color gray and inc
    wi = WhiteIndex(screenNumber);
    bi = BlackIndex(screenNumber);
    white = (wi - bi) / 2 * (1 + contrast);
    black = (wi - bi) - white;
    gray = (white + black) / 2;
    if round(gray) == white
        gray = black;
    end
    inc = white-gray;

    % create meshgrid
    stimulus = createDefaultStimulus(p, inc, visiblesize, gray);
    stimulusTexture = Screen('MakeTexture', w, stimulus, [], 1);

    % query duration of monitor refresh interval:
    framerate = Screen('NominalFramerate', w);
    if (framerate == 0) % OSX FIX !
        framerate = 60;
```

```
    end;
    ifi = 1 / framerate;

    % stimulus 'speed'
    shiftperframe = cyclespersecond * ifi;

    % initial flip
    vbl = Screen('Flip', w);

    % direction of stimulus. -1 == left, 1 == right
    direction = 1;

    % set to 1 after a flip direction operation until a KeyUp Event
    locked = 0;

    srcRectLeft = [0, 0, screenRect(3) / 2, screenRect(4)];
    srcRectRight = [screenRect(3) / 2, 0, screenRect(3), screenRect(4)];

    while 1

        % draw left and right stimulus
        Screen('DrawTexture', w, stimulusTexture, srcRectLeft, [0, 0, screenRect
             (3) / 2, screenRect(4)]);
        Screen('DrawTexture', w, stimulusTexture, srcRectRight, [screenRect(3) /
             2, 0, screenRect(3), screenRect(4)]);

        % draw MDC output
        [MDCData, data] = getMDCData(serialHandle, MDCData);
        widthx = 20;
        startx = round((screenRect(3) - pixels * widthx - pixels * 10) / 2);
        for k = 1:pixels
            if data(1, k) < 0
                color = [0, - 2 * data(1, k), 0];
            else
                color = [2 * data(1, k), 0, 0];
            end;
            Screen('FillRect', w, color, [startx + (k - 1) * widthx + k * 10,
                 100, startx + k * widthx + k * 10, 120]);
        end;

        % tell PTB that there are no more drawing actions until next flip..
        Screen('DrawingFinished', w);

        % Flip 'waitframes' monitor refresh intervals after last redraw.
        vbl = Screen('Flip', w, vbl + 0.5 * ifi);

        % handle user input
        [keyIsDown, secs, keyCode] = KbCheck;
        if keyIsDown
            if keyCode(KbName('f'))
                cyclespersecond = cyclespersecond + SPEEDSTEPSIZE;
                shiftperframe = cyclespersecond * ifi;
            elseif keyCode(KbName('s'))
                if cyclespersecond > 0
                    cyclespersecond = cyclespersecond - SPEEDSTEPSIZE;
                    shiftperframe = cyclespersecond * ifi;
                end;
```

```
            elseif keyCode(KbName('c')) && locked == 0
                direction = 0 - direction;
                locked = 1;
            elseif keyCode(KbName('Escape'))
                break;
            end;
        else
            locked = 0;
        end;

        srcRectLeft(1) = srcRectLeft(1) + direction * shiftperframe;
        srcRectLeft(3) = srcRectLeft(3) + direction * shiftperframe;

        srcRectRight(1) = srcRectRight(1) - direction * shiftperframe;
        srcRectRight(3) = srcRectRight(3) - direction * shiftperframe;

    end;

    fclose(serialHandle);
    closePsych();
catch
    fclose(serialHandle);
    closePsych();
    psychrethrow(psychlasterror);
end;
```

squareSinStim.m

Listing B.7: Square Sinus Stimulus

```
function squareSinStim( p, visiblesize, cyclespersecond, contrast )
%Show a square stimulus which is move- and resizeable.
%   Use the following keys to change the stimulus:
%     f/s : speed up / slow down the speed of the stimulus
%     up/down/left/right key : move the stimulus
%     c : swich stimulus direction
%     i/d : increase / decrease size of stimulus
%     escape : quit program

if nargin < 1
    p = 64;
end;

if nargin < 2
    visiblesize = 2048;
end;

if nargin < 3
    cyclespersecond = 500;
end;

if nargin < 4
    contrast = 1;
end;

try
    % enable all key names
```

```
    KbName('UnifyKeyNames');

    % init PsychToolbox
    [w, screenRect, screenNumber] = initPsych();

    % set constants
    MOVESTEPSIZE = 5;
    SPEEDSTEPSIZE = 5;
    STIMSIZE_MIN_THRESHOLD = 0;
    STIMSIZE_MAX_THRESHOLD = 10;

    % init MDC
    serialHandle = serial('COM1', 'BaudRate', 38400);
    fopen(serialHandle);
    pixels = 24;
    MDCData = zeros(1, pixels);

    % get color gray and inc
    wi = WhiteIndex(screenNumber);
    bi = BlackIndex(screenNumber);
    white = (wi - bi) / 2 * (1 + contrast);
    black = (wi - bi) - white;
    gray = (white + black) / 2;
    if round(gray) == white
        gray = black;
    end
    inc = white-gray;

    % create meshgrid
    stimulus = createDefaultStimulus(p, inc, visiblesize, gray);
    stimulusTexture = Screen('MakeTexture', w, stimulus, [], 1);

    % query duration of monitor refresh interval:
    framerate = Screen('NominalFramerate', w);
    if (framerate == 0) % OSX FIX !
        framerate = 60;
    end;
    ifi = 1 / framerate;

    % stimulus 'speed'
    shiftperframe = cyclespersecond * ifi;

    % initial stimulus size
    stimSize = 5;
    % set stimulus position
    x1 = 0; x2 = x1 + 2^stimSize;
    y1 = 0; y2 = y1 + 2^stimSize;
    destRect = [x1, y1, x2, y2];

   vbl = Screen('Flip', w);

    % direction of stimulus. -1 == left, 1 == right
    direction = -1;

    % set to 1 after a resize operation until a KeyUp Event
    locked = 0;

```

```
    srcRect = [0, 0, 2^stimSize, 2^stimSize];
    while 1

        % draw stimulus
        Screen(’DrawTexture’, w, stimulusTexture, srcRect, destRect);

        % draw MDC output
        [MDCData, data] = getMDCData(serialHandle, MDCData);
        widthx = 20;
        startx = round((screenRect(3) - pixels * widthx - pixels * 10) / 2);
        for k = 1:pixels
            if data(1, k) < 0
                color = [0, - 2 * data(1, k), 0];
            else
                color = [2 * data(1, k), 0, 0];
            end;
            Screen(’FillRect’, w, color, [startx + (k - 1) * widthx + k * 10,
                 100, startx + k * widthx + k * 10, 120]);
        end;

        % tell PTB that there are no more drawing actions until next flip..
        Screen(’DrawingFinished’, w);

        % Flip ’waitframes’ monitor refresh intervals after last redraw.
        vbl = Screen(’Flip’, w, vbl + 0.5 * ifi);
        % handle user input
        [keyIsDown, secs, keyCode] = KbCheck;
        if keyIsDown
            if keyCode(KbName(’UpArrow’))
                if y1 - MOVESTEPSIZE >= 0
                    y1 = y1 - MOVESTEPSIZE;
                    y2 = y2 - MOVESTEPSIZE;
                end;
            elseif keyCode(KbName(’DownArrow’))
                if y2 + MOVESTEPSIZE <= screenRect(4)
                    y1 = y1 + MOVESTEPSIZE;
                    y2 = y2 + MOVESTEPSIZE;
                end;
            elseif keyCode(KbName(’LeftArrow’))
                if x1 - MOVESTEPSIZE >= 0
                    x1 = x1 - MOVESTEPSIZE;
                    x2 = x2 - MOVESTEPSIZE;
                end;
            elseif keyCode(KbName(’RightArrow’))
                if x2 + MOVESTEPSIZE <= screenRect(3)
                    x1 = x1 + MOVESTEPSIZE;
                    x2 = x2 + MOVESTEPSIZE;
                end;
            elseif keyCode(KbName(’f’))
                cyclespersecond = cyclespersecond + SPEEDSTEPSIZE;
                shiftperframe = cyclespersecond * ifi;
            elseif keyCode(KbName(’s’))
                if cyclespersecond >= 0
                    cyclespersecond = cyclespersecond - SPEEDSTEPSIZE;
                    shiftperframe = cyclespersecond * ifi;
                end;
            elseif keyCode(KbName(’c’)) && locked == 0
```

```
                direction = 0 - direction;
                locked = 1;
            elseif keyCode(KbName('i'))
                if stimSize + 1 <= STIMSIZE_MAX_THRESHOLD && locked == 0
                    stimSize = stimSize + 1;
                    x2 = x1 + 2^stimSize;
                    y2 = y1 + 2^stimSize;
                    srcRect(3) = srcRect(1) + 2^stimSize;
                    srcRect(4) = srcRect(2) + 2^stimSize;
                    locked = 1;
                end;
            elseif keyCode(KbName('d'))
                if stimSize - 1 >= STIMSIZE_MIN_THRESHOLD && locked == 0
                    stimSize = stimSize - 1;
                    x2 = x1 + 2^stimSize;
                    y2 = y1 + 2^stimSize;
                    srcRect(3) = srcRect(1) + 2^stimSize;
                    srcRect(4) = srcRect(2) + 2^stimSize;
                    locked = 1;
                end;
            elseif keyCode(KbName('Escape'))
                break;
            end;
        else
            locked = 0;
        end;

        destRect = [x1, y1, x2, y2];

        % we can do that because the rotatingFlage of Screen MakeTexture is
        % set
        srcRect(1) = srcRect(1) + direction * shiftperframe;
        srcRect(3) = srcRect(3) + direction * shiftperframe;

    end;

    fclose(serialHandle);
    closePsych();
catch
    fclose(serialHandle);
    closePsych();
    lasterror
end;
```

createDefaultStimulus.m

Listing B.8: Create Default Stimulus

```
function stimulus = createDefaultStimulus(p, inc, visiblesize, gray)

    fr = 1 / p * 2 * pi;
    [x, y] = meshgrid(0:visiblesize-1, 0:visiblesize-1);
    stimulus = gray + inc * cos(fr * x);
```