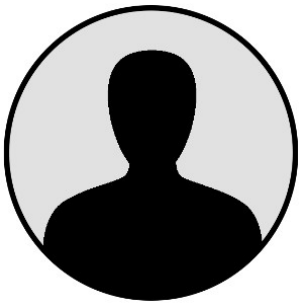
3 Week Poker Project

02321: Hardware/Software Programming



Suhail Abdi Nuur
s205135

Jacob Hornshøj
s193598

Philip Kildal Kristoffersen
s205117

Jonas Jensen
s205129

September 28, 2026

Contents

1 Requirements

Upon starting this 3 week project, the group decided together on creating a poker game from scratch. Before beginning, there was no research into what it would entail, only a general idea was set up, leading to a vague set of requirements that at a later stage could be defined to a more appropriate set of requirements once it was known precisely which restrictions creating such a program would require.

These were the requirements that was set out at that stage of the planning.

Minimum requires:

- IP Cores to control the screen.
- IP Cores to control a keyboard.
- Represent the layout with a grid-based graphical setup.
- Each player needs 2 cards.
- Requires 2 players.
- Interrupt based player input.
- Statistical calculations in run-time, if there is any significant statistical difference.

Expansion requirements:

- More advanced graphic, possibly by showing a picture from an SD card.
- More gadgets on the board (Undefined).
- AI implementation, so it can become a 1-player game.
- Points system - scoreboard/betting implementation.

2 Analysis

Following the initial requirements, and the setup of the projects, the requirements became refined into a specific set instead of something a bit more vague. A few requirements were not required at all and therefore removed, while others were added in as it turned out they were much larger parts than first anticipated. Furthermore, having been given the IP block design for a HDMI-Out setup from Diligent by the teacher, many of the IP requirements were no longer needed as that was the base from which the project began instead of from scratch.

It was further defined that minimum requirements would be focused on, while expansion requirements would be items that could be abandoned if time did not permit it or too many issues arose when trying it out, thus ensuring that a stable product could be produced that worked, while the rest would be nice-to-haves, instead of need-to-haves.

Minimum requirements then became:

- Grid-based layout for graphics by using arrays.
- 2-8 players (Maximum player amount was added).
- Specified game logic (2 cards per player, 3 cards in middle first, then a fourth card added and finally a fifth card added to the middle to follow game rules).
- Player input would need to be interrupt based.
- Player hands would need to be evaluated against each-other to ensure rules could be followed (changed from statistical calculations, as this entailed it and more).

Expansion requirements were turned into:

- Advanced graphics using pictures from an SD card and the ability to switch between "retro" (arrays) and "modern" (pictures) modes.
- Sounds through speakers from an SD card.
- Touch-pad through Arduino interface instead of using terminal.
- AI implementation, moving it from a 2-8 player game to a 1-8 player game.

3 Risk Analysis

There was five specific topics that were argued would be the primary risks for the project and thus focused upon for this situation, which were:

- Corona
- Adding files together
- Linking header files
- Size limitations
- Evaluation of hands

Furthermore, three additional topics were focused on could be problematic if it was decided to expand upon the project from the starting design, which were:

- Sound
- Touch display
- Animation

Going over each one by one, the risk would look as follows:

Risk Analysis			
What can go wrong	How likely is it to go wrong	How serious are the consequences if it goes wrong	Overall risk
Corona	Fairly likely	Very serious	High
Adding files together	Likely	Very serious	Very high
Linking header files	Unlikely	Not serious	Very low
Size limitations	Unlikely	Not serious	Low
Evaluation of hands	Very likely	Serious	High
Sound	Very likely	Not serious	Very low
Touch display	Not likely	Not serious	Very low
Animation	Likely	Not serious	Low

4 Design

4.1 Hardware

Hardware wise, we have implemented the game on the ZYNQ-7000 architecture, with two ARM CPU's representing the processing system plus a programmable logic part, where we have implemented a simple graphics card along with other I/O components. However the most important aspects of our hardware design is the processing system and video card, which enables the execution of C-code on the CPU, and display our game on the screen.
We have been given group of ipcores which can display the contents of a buffer as a frame on the screen. These consist of the VDMA which copies the entire frame directly from memory, the Video timing controller which controls the timings of when the pixels are displayed on screen, the AXI stream to RGB controller which used said timings signals to send the data. Finally RGB to DVI which is the interface.

4.2 Back-end software

With regards to the backend of our program, i.e. all poker logic, some general ideas might be worth explaining before going into depth with the different aspects of the logic itself. First of all, the logic can be divided into different parts - deck logic, turn/player logic, pool/bet logic and evaluation/comparison of hand logic. The snippet below encapsulates most of the base data structures, which the logic utilizes.

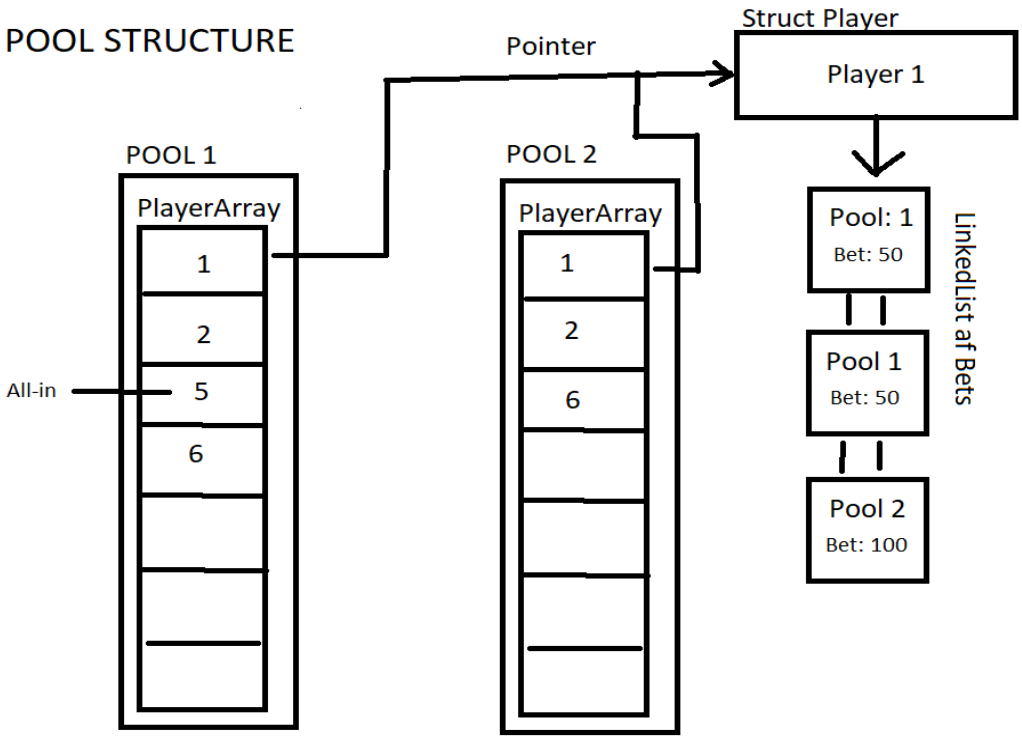


Figure 1: Array Structure

The picture above displays the overview of the pool/bet logic. The software itself is quite adaptable given the bets and the pools themselves are allocated and removed by us in memory depending on demand. In combination with our deck logic, which is also allocated by us in memory, we can wipe the entire deck, pools and bets to ensure a clean slate for each round. This makes sure we only use the specific amount of memory needed. The primary functionality of the pool/bet logic is to ensure that once a player goes all-in, any bets that my be higher are separated in different pools and only players present in these pools are evaluated for winning that pool. The deck logic itself is a linked list of cards in order. By the use of a randomizer we then proceed to allocate cards randomly into a new linked list until a fully shuffled deck is completed. The player/turn logic branches out into different scenarios with varied complexity, but the essence is simple. A for loop holds which players turn it is and depending on prior actions the game will stop the turn once certain criteria has been made.

4.3 Evaluation of hands - Backend

Part of the program which is responsible for evaluating hands is based upon treating cards as a 7 number sorted sequence of cards. Note that only 7 cards are present after the river (the last stage in a turn). Some examples below will showcase different combinations and their respective value from 1-10, with 1 being the worst (High Card) and 10 being the best (Royal Flush) - we i.e. rank-order the different hand combinations from 1 to 10. For simplification

Hand	Value	7 card sorted sequense	Best Combination	
Royal Flush	10	1 – 10 – 10 – 11 – 12 – 13 – 13	10 – 11 – 12 – 13 – 1	
Straight Flush	9	4 – 4 – 5 – 6 – 7 – 8 – 9	5 – 6 – 7 – 8 – 9	(same kind)
Four of A Kind	8	1 – 2 – 7 – 7 – 7 – 7 – 13	7 – 7 – 7 – 7	
Full House	7	1 – 4 – 4 – 4 – 9 – 9 – 9	4 – 4 – 9 – 9 – 9	
Flush	6	2 – 6 – 8 – 10 – 11 – 12 – 13 (all same kind)	8 – 10 – 11 – 12 – 13	
Straight	5	1 – 2 – 3 – 4 – 5 – 6 – 7	3 – 4 – 5 – 6 – 7	
Three of A Kind (Set)	4	1 – 2 – 3 – 3 – 3 – 10 - 13	3 – 3 – 3	
Two Pair	3	1 – 1 – 2 – 2 – 10 – 10 – 13	1 – 1 – 10 – 10	
One Pair	2	1 – 2 – 2 – 6 – 6 – 10 – 11	6 – 6	
High Card	1	1 – 5 – 6 – 10 – 11 – 12 – 13	1	

Figure 2: Examples of different possible 7-card combinations

only the card values are showed above (not the kind). Edge-cases are showcased to highlight, why the algorithm that must evaluate a hand is not all that trivial. E.g. straight combinations (5 cards connected) can also be 6 or 7 cards connected, but the best possible combination must still be found, since two players can have a straight but one might be higher than another. The same applies for the other hands - in the case of full house, the highest set must be found. Note that ace is the highest card in poker except for the lowest straight, where it counts as the lowest card (1 - 2 - 3 - 4 - 5).

4.3.1 Evaluation algorithm

The design of the actual algorithm that detects the best possible hand is displayed below.

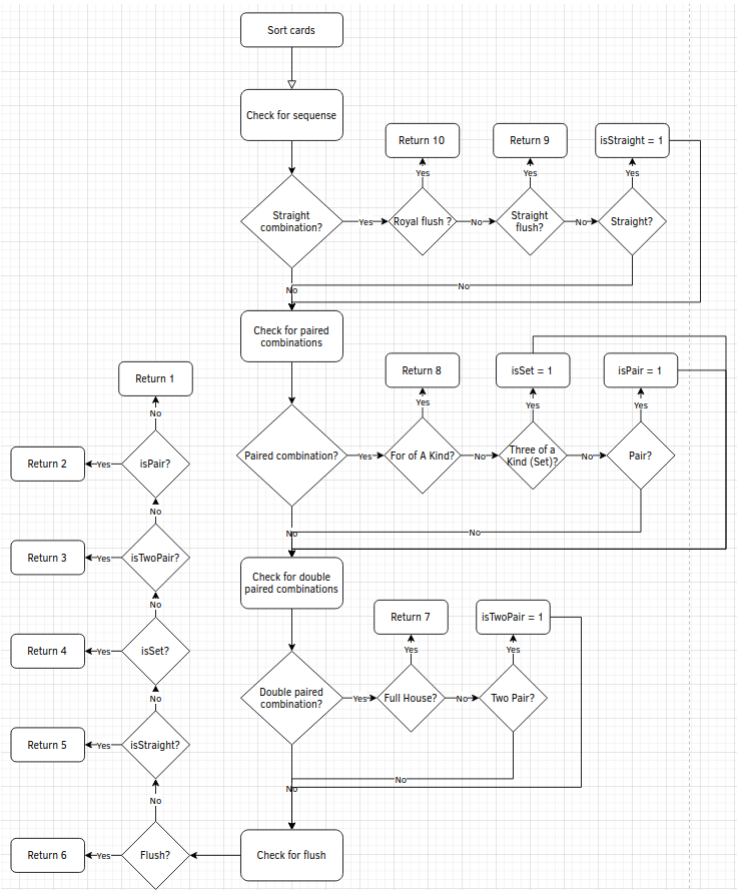


Figure 3: Evaluation flowchart

Note that this displays the general design of the algorithm, and is fairly complicated when actually implemented. This is because, as mentioned earlier that multiple combinations of the same type can occur in a hand, why the best possible combination must be made. The figure above only encapsulates the detection of a combination - not necessarily the best combination. What can be said about the algorithm is that a top-down divide and conquer approach has been taken. Similar combinations are checked in one go - e.g. Four of A Kind, Set, and Pair are of similar type, why it makes sense to check them in one go. Since we are checking from best to worst combinations, we can return certain values and not bother with checks for worse hand combinations. For some detection the checks must however finish since e.g. a full house is better than a straight but the straight check is made before the full house check - remember, we are trying to return the best possible hand value. The reason for sorting the cards, is that combinations are easier to check this way - every check has to do with comparing connected or equal value cards. As mentioned, we are not only giving a hand a certain value - we are also making the best hand from the 7 cards and if multiple combinations of the same kind occur we must find the best which becomes much simpler with sorted cards. The general idea to achieve this is displayed in the flowchart above where we are making the best hand based on an offset - e.g. a straight combination is contained in

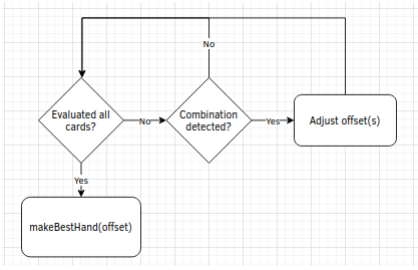


Figure 4: Make best possible hand flowchart

2 - 2 - 3 - 4 - 5 - 6 - 12, where the offset is from card 1 (the second 2). For double paired combinations, two offsets are necessary, one for first pair or first set and one for second pair or second set. It should be noted, that evaluations happen on a turn basis so players can see the best hand hand they currently have, why the algorithm isn't limited to 7-card combinations.

4.3.2 Comparison of hands

The reason we need to evaluate hands is that we need to determine who has the best hand if multiple players are in play at the end. Therefor we need a design for an algorithm, which compares all players' hands, who are still in play for the pot. A flowchart is provided below, to highlight the important aspects of the design.

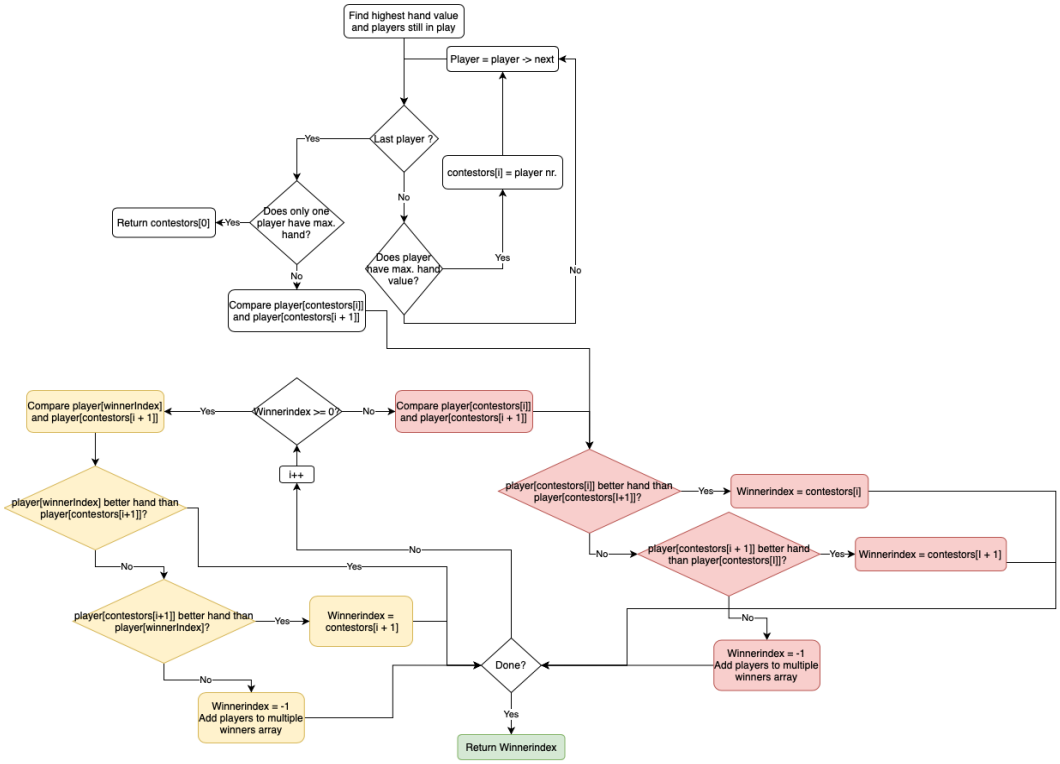


Figure 5: Comparison of hands flowchart

The general idea behind this algorithm, is that we start out by finding the maximum hand value. If only one player has this, we already have a winner *contestors[0]*, and no hands actually need to be compared, since this has been done implicitly by our evaluation. However, if multiple players have the winning hand value, we need to compare hands, and find the player with the best combination of cards, why we save exactly this combination in the evaluation part of the logic (**4**). The way these hands are compared is in a sequential iterative manner, where we keep track of a winner index (the player number of the current winner). The two parts of the flowchart marked with yellow and red represent the two different scenarios in which two hands should be compared. If a better hand is found, i.e. two hands are not equal, we compare the winning index with the next player, who has not been compared yet (yellow part). If winner index is below 0 (red part), i.e. first iteration or if two hands are equal, we can simply compare compare the two next adjacent hands. Note that if the best hands are equal, the returned winner index is -1, why we also maintain an array with multiple winners which can be accessed in other parts of the backend logic. How two hands with equal hand-value are compared card-wise is fairly trivial, since it simply involves checking e.g. for higher pairs/sets for paired/double paired combinations, and for highest card in a straight or flush.

4.3.3 Outs and statistics

We have also decided to incorporate some simple statistics in the game, where each player can get a hint which is how many cards in the deck will improve their current hand value, and what the chance is of hitting that card. These numbers are actually based upon the evaluation algorithm, where every card in the current combination is replaced with a card from the deck and evaluated in between. This is repeated for every card in the deck, which means we e.g. for a given player with 5 players around the table on the flop (39 cards left in the deck) are doing $39 \cdot 7$ evaluations because we are replacing every card in the current combination (7 cards) with a given card from the deck, which is also repeated for every other remaining cards in the deck. Note that redundant checks occur, because e.g. on a flop only a 5-card combination is present which leaves room for optimization. The chance of hitting a card is simply a matter of calculating the proportion $\frac{outs}{cards\ in\ deck}$.

4.4 Front-end software

The design of the graphic revolves around a static background and a layer of items that can be placed upon said background. Many of the external functions meant for use by the game loop that manipulate the screen will be based around placing numbers into the data structure that decides what will be shown on screen.

4.4.1 Background layer and foreground layer 2d arrays

The background is a simple 1920*1080*2 size buffer that can contain the static background. This will be copied into the frame whenever we want to clear the screen for example for an update which moves the foreground elements.
The foreground will be represented by a 2d array.The values on the foreground array referred to as location array will be used to indicate what will be drawn on top the background. The size of the array will be 640x540 in order for the array to cover all of the 1920x1080 screen in "blocks". Below is an picture which illustrates this.

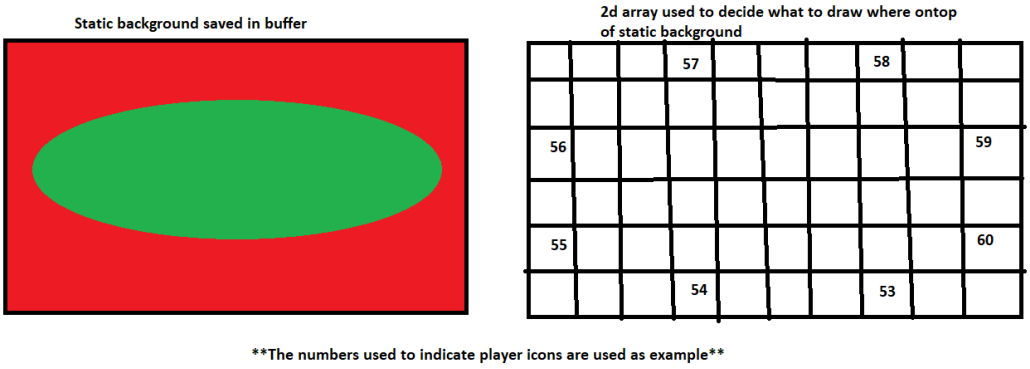


Figure 6: background and foreground

With this example i used to numbers to signify player icons so this will result in this:

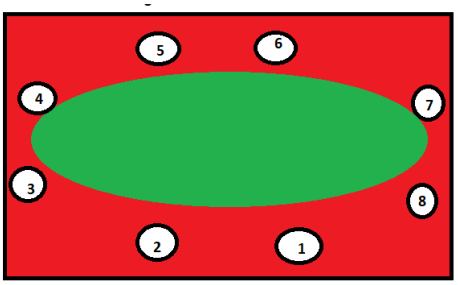


Figure 7: resulting image on screen

4.4.2 Drawing figures

Retro look

The retro look will be achieved by drawing different colored squares based on the values of a 2d array.This 2d array is separate from the 2d array which specifies location and type of object. This 2d array will be used as a blueprint for a drawing.An example being if for example you find value 1 at array[0][0] you will draw a black block at location (x+0*blockwidth, y+0*blockwidth). As you can see we use the location of the figure as an offset along with the location within the array time the size of the block. If we want to achieve transparency in

the pixelated designs we will just use a number in the 2d array not used in the function hence no block will be drawn there. Here is an example of this:

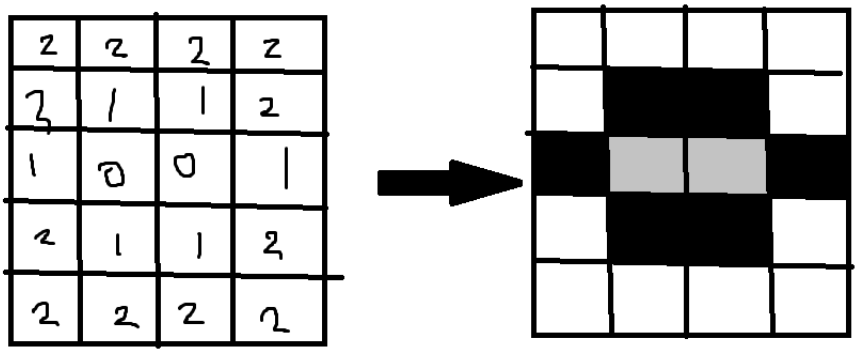


Figure 8: resulting image on screen

HQ look and alpha channel

The high quality look will achieved by using actual pictures acquired from the internet. We will use the said pictures and pre-process them into 4byte groups of BGRA(BLUE, GREEN, RED, ALPHA). The byte files will be stored into a storage media and be accessed in the program via libraries that can utilize said storage media and saved into buffers. The pictures will be shown via copying the pictures into the frame buffer in the desired location line by line or pixel by pixel.

Whenever we want to remove the background of an image we make it a solid color and set the alpha of pixels of that color to max when pre-processing it. That way if we encounter it while moving it into a frame we can ignore it and achieve a transparent background.

4.4.3 Screen updates using vsync, double buffering and movement

In order to avoid screen tearing we will use the vsync signal which indicates whether the screen has finished updating. We will also utilize double-buffering. Whenever we update the screen we will first check the flag that indicates whether we have a frame waiting to be swapped in. As long as that is true we will wait. When no other frame is waiting we write to the back-buffer which is not currently shown on the screen. When we are are done we will set the flag to true so that can be swapped at the next vsync signal. The program will continue execution then. What swaps the frames will be an interrupt handler connected to the vsync signal. The swap will be fast as it will simply swapping pointers.However since vsync is active high it will need to be negated.

Movement will be achieved by adding every currently moving object do a dynamic data structure. The data structure will contain the current location of the object, the end location, the figure type and whether they are currently moving or not. The location will be updated every

frame update. The moving figures are displayed separately from the constant figures.Once a figure has reached its destination it is added to the 2d array for immovable objects. When every figure in the dynamic data structure has stopped moving it is cleared.

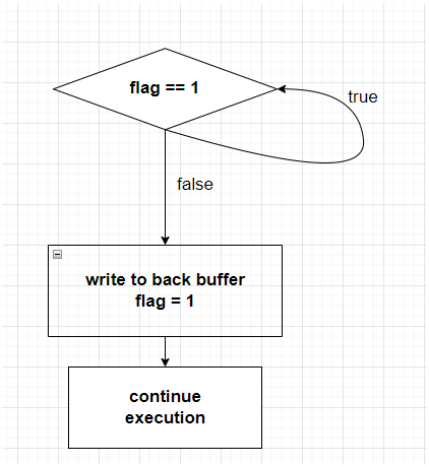


Figure 9: writing to back-buffer

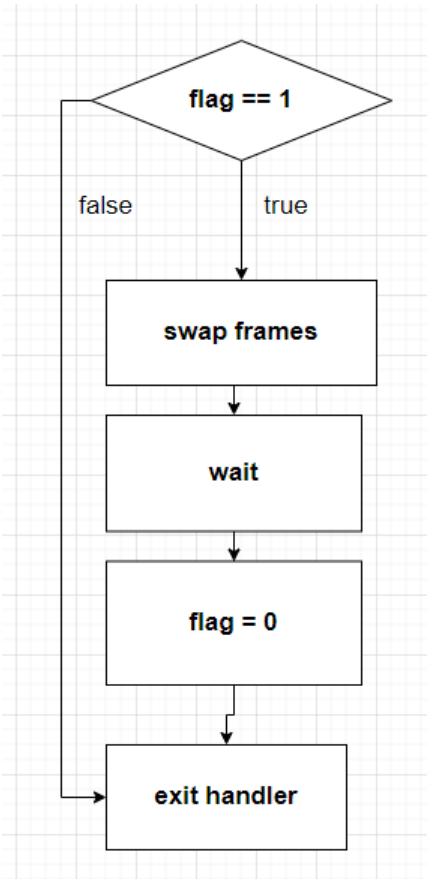


Figure 10: interrupt handler for swapping frames

5 Implementation

5.1 Evaluation of hands

How the evaluation of hands have been implemented is only a matter of implementing different functions searching for different card combinations, i.e. Straights, paired, double paired and flushes as described earlier. The different functions are different, but for simplicity only one is showcased below:

```
    for (int i = 0; i < 7; i++) {
    if (tmpCard[i].cardVal == tmpCard[i + 1].cardVal) {
      if (tmpCard[i].cardVal == tmpCard[i + 2].cardVal) {
        houseVal = tmpCard[i].cardVal;
        offsetHouse = i;
        set = 1;
        if (i == 4) {
          break;
        }
        continue;
      }
      if (pair1 == 1 && pair2 == 1 && pair3 == 0
      && tmpCard[i].cardVal != houseVal) {
        offsetPair3 = i;
        pair3 = 1;
      }
      else if (pair1 == 1 && pair2 == 0 && pair3 == 0
      && tmpCard[i].cardVal != houseVal) {
        offsetPair2 = i;
        pair2 = 1;
      }
      else if (pair1 == 0 && pair2 == 0 && pair3 == 0
      && tmpCard[i].cardVal != houseVal) {
        offsetPair1 = i;
        pair1 = 1;
```

Listing 1: Evaluation of hand

Here we look for cards with the same value. We can compare adjacent cards with index *i* and *i + 1* because they are sorted. If we do get a match, we also need to check for a set, since we are checking for two pair and full house combinations. If not a set, and previous pairs haven't been found, we execute the last if statement, and save the offset, which is important when forming the best hand - remember **(4)**. Otherwise if later pairs are found, they must be higher due to the low-to-highest sorting.

```
    else if (set && pair1) {
    makeBestHand(offsetPair1, 0, 0, 0, 1, offsetHouse, 0, 0);
    return 7;
  }
  else if (pair2 && pair3 && !isStraight) {
    makeBestHand(offsetPair2, 0, 0, 0, 1, offsetPair3, 0, 0);
```

```
7       return 3;
```

Listing 2: set hand

As mentioned earlier, we are forming the best hand based on offset(s) - in this case offsets, because we have two potentially unconnected pairs or pair and set. Different variations when forming the best hand are based on two offsets - the first being the first pair or set, and the second being the second pair or set, is provided above. There are actually six cases in total in this function but for simplicity only a couple is shown. *makeBestHand* also takes other control parameters indicating the kind of combination. Note that all other combinations only need one offset.

5.1.1 Comparison of hands

When it comes to the comparison of hands, this section will only cover, the cases where one hand is better than another for simplicity. As seen in the flowchart, which encapsulates all the functionality, we have the first case, where our winnerindex is below 0 (first iteration or if no better hand has been found, i.e. *retValue = -1*). The contestors[i] array is essentially initialized with all -1's, because 0 also can be a winnerindex. The reason we are giving *findQualifyingHand()* as an argument to *compareTwoHands()* is that it takes to parameters of type *struct Player*. The function returns 1 if first player passed as first argument has a better hand and 2 if second player hand is better. It returns -1 if they are equal. Equal hands is a matter of filling up an array with multiple winner indexes and can be seen in the source code.

```
1  int retValue = 0;
2  int winnerIndex = -2;
3  for (int i = 0; i < g - 1; i++) {
4  if (contestors[i] != -1 && contestors[i + 1] != -1) {
5    if (winnerIndex < 0){
6      retValue = compareTwoHands(findQualifyingHand(contestors[i],
7          maxPlayerAmount, player),
8        findQualifyingHand(contestors[i+1], maxPlayerAmount, player));
9    }
10   if (winnerIndex >= 0) {
11     retValue = compareTwoHands(findQualifyingHand(winnerIndex,
12         maxPlayerAmount, player),
13       findQualifyingHand(contestors[i+1], maxPlayerAmount, player));
14   }
15   if (retValue == 1 && (winnerIndex == -2)) {
16     winnerIndex = contestors[i];
17   }
18   else if (retValue == 1 && (winnerIndex >= -1)) {
19     continue;
20   }
21   else if (retValue == 2) {
22     winnerIndex = contestors[i + 1];
```

```
23      }
```

Listing 3: Comparison of Hand

5.2 Graphical implementation

5.2.1 Displaying background and figures on top of background

Background

Both the HQ background and the retro background are kept in buffers as they are static and unchanging. The retro background is generated from a giant 640x540 2d int array. The process of generating blocky images form 2d arrays is explained in the *"Displaying blocky retro figures"* section .This is written to an empty buffer and saved. The HQ buffer is simply loaded in from the SD card. Both images result in 1920x1080 images that fill a 1920x1080x4 buffer. Whenever the screen needs to updated and its contents cleared we simply memcpy the contents of the background buffer to the frame. We copy to the back-buffer for seamless updating.

```
1  void updateBackground(){
2    if(retro == 1){
3      memcpy(dispCtrl.framePtr[nextFrame()], background1, DEMO_MAX_FRAME
       );
4    }else{
5      memcpy(dispCtrl.framePtr[nextFrame()], background2, DEMO_MAX_FRAME
       );
```

Listing 4: Update Background

Depending on the on the visual mode we are operating under we either update using the retro background or the high quality background. This variable is set early in the execution of the program via user prompt or can be swapped between by the button interrupt.

Draw layer

The drawlayer function is meant to be run after updating the background. This function draws on top of the background and works as a sort of foreground. The area on top of the screen as described in the design section is represented by a 2d array. The 2d array called location array is 540x640 dividing the screen into that amount of blocks. whenever you want to draw something at a specific location on the screen you can place its corresponding item number into the location array at that location. The drawlayer function will then iterate though the 2d array and will then run the drawing function corresponding to the item number found in the array. The example below being the icon for player 1. The row and column index multiplied with blockwidth and blockheight respectively is the coordinate of the upper left corner of the figure that will be drawn(j*blockwidth, i*blockheight).
Below is an incomplete snippet of the switch case in drawlayer function which gives an idea on how it works:

```
void drawLayer(){
  XTime start;
  XTime end;

  int blockwidth = dispCtrl.vMode.width/GRANUALITYWIDTH;
  int blockheight = dispCtrl.vMode.height/GRANUALITYHEIGHT;

  for(int i = 0; i < GRANUALITYHEIGHT; i++){
    for(int j = 0; j < GRANUALITYWIDTH; j++){

      if(locationArray[i][j] == 0 ){continue;}//if zero no item so skip rest
      if(locationArray[i][j] >= 53 && locationArray[i][j] <= 60 ){ //show icons
        switch(locationArray[i][j]){
        case 53: //player 1
          if(currentPlayer == 0){
            showICON(0, j,  i, 1);
          }else{
           showICON(0, j,  i, 0);
          }
          break;
```

Listing 5: Draw Layer

Figures that do not need to be relocated or moved can be drawn outside the double for-loop and selection structure and need no index. Examples of this is the menu and the text box. These need to be in the foreground but are static never need to be relocated or removed. Cards that are currently moving are also not placed in the location array as they constantly need to be updated. They are drawn using a separate function outside the double for-loops. These 3 functions are at the bottom of the draw layer function.

```
drawMenu(579, 449-30, callOrCheck);
printText(3, 465);
drawMovingCard();
```

Listing 6: Draw Layer 2

5.2.2 Displaying blocky retro figures

Whenever we display retro pictures we use a "small" 2d array as a blueprint. The entire screen which is 1920x1080 pixels is divided up into 640x540 blocks. This makes the size of a block 3x2 pixels. We use a smaller 2d array which size is defined by the size of the figure we want, for example if we use a 2d array that is 15x15 we end up with a figure the size (15*3)*(15*2) pixels. We iterate through the indexes of the 2d array and drawing blocks at locations offset by the array index times the block size. The blockwidth times the row and the block height times the column.The color of the block is determined by the number and is

dependant on the function. An example below is the general draw function where you can specify the colors used when the index 0 and 1 are encountered in the 2d array.

```
1  void generalDraw(int arraywidth, int arrayheight,
2  int diagram[arrayheight][arraywidth], int color1, int color2, int x,
     int y){
3  int blockwidth = dispCtrl.vMode.width/GRANUALITYWIDTH;
4  int blockheight = dispCtrl.vMode.height/GRANUALITYHEIGHT;
5
6  for(int i = 0; i < arrayheight; i++){
7    for(int j = 0; j < arraywidth; j++){
8      if(diagram[i][j]== 0){
9        drawSquare2(blockwidth,blockheight,
10       (x + j)*blockwidth, (y + i)*blockheight,
11       color2, dispCtrl.framePtr[nextFrame()]);
12
13     }else if(diagram[i][j] == 1){
14       drawSquare2(blockwidth,blockheight,
15       (x + j)*blockwidth, (y + i)*blockheight,
16       color1, dispCtrl.framePtr[nextFrame()]);
```

Listing 7: General Draw Layer

The function drawsquare2 draws a square of given dimensions at a given location. The function uses the function drawpixel inside two for-loops to draw a square in the given frame. Drawpixel being a function meant for making it easier to alter a specific pixel at a specific coordinate.

5.2.3 HQ pictures, loading pictures from SD card, using the alpha channel

The pictures we use are contained in buffers the size being width x length x 4. We get these pictures from the internet and use the program irfanviewer to process them into a byte file in the form RGB with an ASCII header. This is further processed with a small java program into the form BGRA and removing the ASCII header. The alpha channel we add is very basic and is essentially just adding an alpha value of 255 to a specific color in order to remove that in software.

We are able to access files in our program via the use of the generic fat file system library(xiffs) provided by Xillinx. This allows us to access the file system on the SD-card inserted into the SD-card slot. Thus we have stored every picture needed by the program on there. On program startup these are loaded into buffers in global memory.
In order to easier understand how we currently print pictures with an alpha channel we need to understand the earlier version without. Since the picture is stored in a 1d array i.e. a buffer, the dimension of the picture is not known, therefore in order to print it we need the width and height of the picture in order to understand at which offsets every line of this picture starts within this buffer.

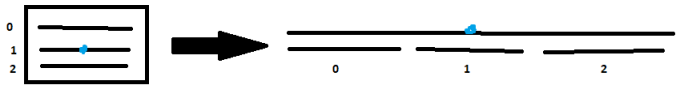


Figure 11: image in buffer

Thus we copy the picture a line at a time using memcpy into the correct offset in the frame. As you can see the pointer to the frame is offset by the the x coordinate times 4(every pixel is represented by 4 bytes). It is also has as offset the y coordinate with the current line number as offset times 4 times pixel stride. The pixel stride being the amount of pixels in a line. This is due to the 1d nature of the buffer, in order to get the location within the buffer that is one y coordinate down you need to offset by the length of one line.

```
1  void printPicture(int width, int height, int size ,u8 pictureBuffer[
       size],
2  int x, int y){
3    u8 *bufferPointer = pictureBuffer;
4
5    for(int i = 0; i < height; i++){
6      memcpy(dispCtrl.framePtr[nextFrame()]+x*4 +
7      ((y+i)*1920*4) , (pictureBuffer + (i*(width*4))),(width*4));
```

Listing 8: Print Picture

This is further complicated by the functions second iteration which ignores pixels with an alpha channel of 255. Because due to this we cannot copy a line at a time, but we need to do per pixel basis which is represented by 4 bytes.

```
1  //updated to look for alpha not just green
2  void printPictureWithGreenAlpha(int width, int height, int size ,u8
       pictureBuffer[size],
3  int x, int y){
4    u8 *bufferPointer = pictureBuffer;
5
6    for(int i = 0; i < height; i++){
7      for(int j = 0; j < width; j++){
8        if(!(pictureBuffer[(i*(width*4)) + (j*4) + 3] == 255)){
9          memcpy(dispCtrl.framePtr[nextFrame()] + ((j+x)*4) +
10         ((y+i)*1920*4), (pictureBuffer + (i*(width*4)) + (j*4)),4);
```

Listing 9: Alpha Channel

5.2.4 Updating the screen

In order to avoid flickering of the graphics we use a double buffered approach and use vsync to get the right timing to swap the frames. Every drawing function we use update the back buffer. After we have written to the back-buffer we set the flag that indicates a new update to 1. Based on this flag an interrupt handler that operates on the vsync signal swaps the

back-buffer and front-buffer at the next vsync signal. The interrupt is useful so that we do not swap whilst in the process of showing a frame in the front buffer.After swapping the frames we set the flag to 0 indicating that there are no more updates. This is the code block we use when we want to update while in the game logic:

```
while(getFrameFlag() == 1){};//wait until screen can be updated
    updateBackground();
    drawLayer();
    setFrameFlag(1);
```

Listing 10: Updating

It wont write to the back-buffer if there is already data in it that has not been shown yet. The interrupt handler will not swap the buffers if there is no new content in the back-buffer. We have used the integer as a sort of semaphore and treat the double buffers as a mutual resource in need of mutual exclusion.

```
void swap_handler(void *InstancePtr){
    if(getFrameFlag() == 1 ){
        updateMovingCard();
        swapFrame();
        TimerDelay(1000);
        setFrameFlag(0);
        }
```

Listing 11: vsync interrupt handler

5.2.5 Movement

We have made it possible to move cards. Whenever we want to move cards we add them to a linked list data structure. The struct contains the current location, end location, card index and whether they are actively moving as well as a pointer to the next card.

```
struct MovingObject{
    int currentx;
    int currenty;
    int endx;
    int endy;
    int active;
    int cardId;
    struct MovingObject* nextMovingObject;
};
```

Listing 12: Moving Cards

The cards are drawn by a function a the bottom of the drawlayer function separately from the static figures which locations are found from the 2d array.The function traverses the linked list and draws the cards at their current locations. The position of the cards are moved in the vsync interrupt handler if there is a new update. Thus by continuously updating you can move the cards. When the cards reach their location the number representing the card is

placed into the 2d location array as they now do not need to continuously move and active is set to 0. When every card has active == 0 the linked list is not needed anymore as every card has been placed in the location array and is thus freed to avoid memory leak.

5.3 Hardware

5.3.1 VDMA

Our VDMA is responsible for reading a frame from main memory. However, when using the processor to draw some pictures only a part of the whole frame may reside in main memory, why we must flush the cache before displaying.

```
1 void flush(u8 *frame){
2   Xil_DCacheFlushRange((unsigned int) frame, DEMO_MAX_FRAME);
3 }
```

Listing 13: Flush

5.3.2 Video timing controller

The video timing controller is responsible for informing whenever the screen is ready to display a new line or a new frame, i.e. the *hsync* and *vsync* signals respectively. We specifically use the vsync signal in software as an interrupt to only start displaying a new frame whenever the screen is done displaying, meaning whenever it is done displaying vertically. This way, we avoid swapping frames while in the midst of already displaying one.Note that we negate the *vsync* signal, since it is active-high.

5.3.3 AXI stream to RGB controller

Because our VDMA essentially just feeds its output into an AXI-stream converter, the signal must be pre-processed to meet the HDMI interface. Essentially HDMI is the same as VGA as seen below but a digital signal where the bytes send respond to RGB plus the vsync and hsync signals. The digitalized version of VGA is exactly achieved in our case by converting the AXI-stream to parallel video data via the AXI-Stream to Video Out component. This signal is sent on to the RGB to DVI (DVI and HDMI use same transmitting/receiving interface).

5.3.4 interrupts

We utilize different interrupts in our system. This allows for better utilization of the processor, since we avoid polling and other aspects of our program are quite processor demanding. How the interrupts specifically is implemented on the FPGA, is by connecting interrupt sources from various GPIO sources and also the *vsync* signal in our case (both part of the PL), to the GIC (generic interrupt controller), which is a part of the processing system and is responsible for registering interrupt handlers for various devices and setting priority and trigger types.

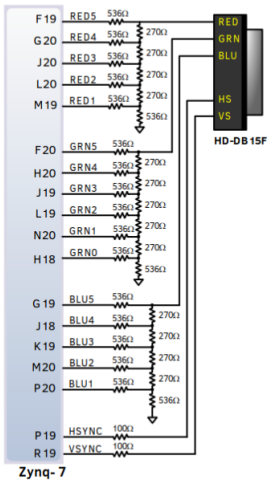


Figure 12: ZYBO VGA circuit [1, s. 6]

6 Tests

One part of the program, which can be tested isolated on the backend is the evaluation and comparison of hands algorithms. A test program has been made with the logic from generation of the deck and players. The program outputs the test results to a text file with all phases of the game (pre-flop, flop, turn and river) with the best possible combination a player has and how many outs he/she has. The winner is also found at the end. An example of the test can be seen in the appendix, where 8 players are evaluated through to the end. The expected results are met, i.e. right evaluations occur and the right winner is found. The test program generates random hand and table cards and is attached as source code (*src_eval_test.zip*). Note that with 7-card sorted sequences pre-river, there are less than 7 cards, why other values than actual cards dealt are initialized with "dummy" values, so the algorithm does not detect any combination including these non-existing cards yet. Furthermore outs on the river are not actual useful, since no more cards are dealt but are still displayed.

The attached video to the hand-ins showcase the program in action and worked as a running test of the system.

7 Discussion

Following the design and the implementation of the project, and while a few bugs are still present, a working product have been completed. An, almost, fully functional poker game can now be played through a terminal with the Zybo. All it requires is a PC to interface with the Zybo and act as a terminal, the Zybo itself as well as a screen connected to the Zybo through an HDMI cable to show the game.

Looking back at what the refined requirements were, all the minimum requirements were added into the game as well as one of the expansion requirements.
The graphical design was achieved using arrays to give it a retro look, as well as using it to assign a location to everything on the table, effectively fulfilling the first requirement. All arrays were drawn manually by using 2d arrays and nested for loops to ensure visibility was proper once implemented.
A player amount was added, as well as player icons and tokens, indicating which player was currently playing, and a visualization of active player as well as current bet and current pool.
Full game logic was implemented, barring minor bugs, the game could now run full Texas Hold'Em Poker including all game rules and proper pot splitting in case of all ins and possible several winners of a hand. One major change was the decision to run with a single kicker instead of several, but it was deemed that the chance was too small for two kickers to be relevant, compared with the amount of time it would take to create.
Player inputs was decided to be implemented through a serial connection instead of being interrupt based, as it made more sense once a working implementation was achieved, and therefore did not fulfill the initial requirement although player inputs was achieved through other means.
A full evaluation of each hand against each other was completed to both create the statistics required to compare winning chances, which was implemented in a text box in the lower left corner of the screen, as well as ensure the correct winning hand was chosen in the game logic implemented.
Advanced graphics was acquired through the use of pictures on an SD card and an alpha channel, which enabled the ability to use either an old-school visualization of the game, or a modern visualization of the game. The switch between the two modes was achieved with an interrupt based button on the FPGA resulting in expansion requirement being solved.

However, while these requirements were all completed and implemented, there was also attempts at implementing two other expansion requirements, namely the sounds through speakers from an SD card and the touch-pad through an Arduino interface to use instead of the terminal. These efforts were however abandoned after a while due to issues with the implementation and it was decided that a focus on the currently working interface was a

better solution due to the overall time constraints of the project.

An attempt to create an AI to move the game from a 2-8 player game into a 1-8 player game was abandoned entirely due to time constraints.

8 Conclusion

After a project like this, we could conclude that creating a poker game was much bigger than anticipated. Not due to the obvious "it's hard to create a game" but more due to the sheer amount of work that goes into creating the code that makes up for the game logic and the game rules. It was a very fun experience for all four of us, and at the end of the project, we had a firm grasp of what we had done, what went right, and what went wrong, and for most of it, how we would do it should we do it all over again.
A difference in focus would have been chosen, with more effort put into the game logic from the beginning, and less effort into the graphics to begin with, would probably have been a better solution. Working with a single graphical interface instead of two, giving us more time to flesh out the edge cases in the logic that either broke the game or created weird situations that were not supposed to happen.
We all agree that the final product delivered was something we were proud of having created, having it work very well with very few bugs, and only really regret that we could not get the touch-pad and sound to work in time before the deadline, simply due to the amount of time spent and how nice it would be to have these things in addition to the program, although they are in no way required for it to work or be enjoyable.

9 Appendix

9.1 References

References

[1] L. H. Crocket, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart. *ZYBO Reference Manual*. Strathclyde Academic Media, 2014.

9.2 User Manual

Start program by loading it in through SDK. Use the terminal from PuTTY to set up the game.
Setup is done by first entering the starting pool, any amount can be chosen here, finish with enter.
After starting pool has been chosen, enter the amount of players (minimum 2, maximum 8), finish with enter.
Once the amount of players have been chosen, the game begins. Small blind and big blind are automatically dealt, and first player turn is the player after big blind (player 4 in first round, since player 1 is automatically the dealer in first round, player 2 deals small blind, player 3 deals big blind).
Each player can now follow the rules of the game by choosing the appropriate response:
1. to fold their hand.
2. to call the previous opponents hand.
3. to raise the current bet (after which a number that is raised to needs to be input and finished with enter. This input can only be up to a maximum of what you currently hold in your own pool).
4. to get a hint about your hand.
Once you lose all your money, you are out of the game.
The game then continues until there is only 1 player left which has won the entire starting pool from all X players.
Game can be run in two graphical modes, retro and modern, done by pressing button 0 on the FPGA.

9.3 Contribution to project work

–Contribution to project work
Jonas - Implementation: Evaluation algorithm, comparison of hands algorithm, outs-statistics and a little help on graphics.
Jonas - Report: Design and implementation of Evaluation of hands - Back-end and Hardware

design/implementation with Suhail (50/50).

Suhail - Implementation: Graphical implementation(Drawing functions using 2d arrays, loading/showing pictures from SD card, double-buffered frame updates, movement), utilization of interrupts and functions that manipulate the game visuals, touch-pad (failed implementation).
Suhail - Report: Graphical implementation(Implementation section), Front-end software(Design section).

Philip - Implementation: The overall game-loop structure(player turns, choosing options, giving them cards etc), player linked-list data structure, pool split algorithm, deck data structure generation and shuffling, touch-pad (failed implementation).
Philip - Report: Back-end Software.

Jacob - Implementation: 2d Array design, sound (failed implementation). Jacob - Report: Requirements, analysis, risk analysis, discussion, conclusion, appendix.

9.4 Test reports

```
##########################################################################################
######################################## PRE-FLOP ########################################
##########################################################################################
## Player nr. 1 ##
7-card sorted sequense:
Card 1: ##VALUE: -2 ##TYPE: -2
Card 2: ##VALUE: -4 ##TYPE: -4
Card 3: ##VALUE: -6 ##TYPE: -6
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -10 ##TYPE: -10
Card 6: ##VALUE: 4 ##TYPE: 3
Card 7: ##VALUE: 8 ##TYPE: 2
Best hand combination:
Card 1: ##VALUE: 8 ##TYPE: 2
Card 2: ##VALUE: 4 ##TYPE: 3
Card 3: ##VALUE: -10 ##TYPE: -10
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -6 ##TYPE: -6
## High card! ##

Outs: 5
Chance of hitting an out: 10%
-----------------------------------------
```

```
## Player nr. 2 ##
7-card sorted sequense:
Card 1: ##VALUE: -2 ##TYPE: -2
Card 2: ##VALUE: -4 ##TYPE: -4
Card 3: ##VALUE: -6 ##TYPE: -6
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -10 ##TYPE: -10
Card 6: ##VALUE: 10 ##TYPE: 3
Card 7: ##VALUE: 13 ##TYPE: 4
Best hand combination:
Card 1: ##VALUE: 13 ##TYPE: 4
Card 2: ##VALUE: 10 ##TYPE: 3
Card 3: ##VALUE: -10 ##TYPE: -10
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -6 ##TYPE: -6
## High card! ##

Outs: 4
Chance of hitting an out: 8%
-----------------------------------------
## Player nr. 3 ##
7-card sorted sequense:
Card 1: ##VALUE: -2 ##TYPE: -2
Card 2: ##VALUE: -4 ##TYPE: -4
Card 3: ##VALUE: -6 ##TYPE: -6
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -10 ##TYPE: -10
Card 6: ##VALUE: 1 ##TYPE: 1
Card 7: ##VALUE: 3 ##TYPE: 4
Best hand combination:
Card 1: ##VALUE: 3 ##TYPE: 4
Card 2: ##VALUE: 1 ##TYPE: 1
Card 3: ##VALUE: -10 ##TYPE: -10
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -6 ##TYPE: -6
## High card! ##

Outs: 3
Chance of hitting an out: 6%
-----------------------------------------
```

```
## Player nr. 4 ##
7-card sorted sequense:
Card 1: ##VALUE: -2 ##TYPE: -2
Card 2: ##VALUE: -4 ##TYPE: -4
Card 3: ##VALUE: -6 ##TYPE: -6
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -10 ##TYPE: -10
Card 6: ##VALUE: 1 ##TYPE: 2
Card 7: ##VALUE: 10 ##TYPE: 2
Best hand combination:
Card 1: ##VALUE: 10 ##TYPE: 2
Card 2: ##VALUE: 1 ##TYPE: 2
Card 3: ##VALUE: -10 ##TYPE: -10
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -6 ##TYPE: -6
## High card! ##

Outs: 3
Chance of hitting an out: 6%
-----------------------------------------
## Player nr. 5 ##
7-card sorted sequense:
Card 1: ##VALUE: -2 ##TYPE: -2
Card 2: ##VALUE: -4 ##TYPE: -4
Card 3: ##VALUE: -6 ##TYPE: -6
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -10 ##TYPE: -10
Card 6: ##VALUE: 3 ##TYPE: 2
Card 7: ##VALUE: 5 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 5 ##TYPE: 1
Card 2: ##VALUE: 3 ##TYPE: 2
Card 3: ##VALUE: -10 ##TYPE: -10
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -6 ##TYPE: -6
## High card! ##

Outs: 4
Chance of hitting an out: 8%
-----------------------------------------
```

```
## Player nr. 6 ##
7-card sorted sequense:
Card 1: ##VALUE: -2 ##TYPE: -2
Card 2: ##VALUE: -4 ##TYPE: -4
Card 3: ##VALUE: -6 ##TYPE: -6
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -10 ##TYPE: -10
Card 6: ##VALUE: 4 ##TYPE: 4
Card 7: ##VALUE: 7 ##TYPE: 2
Best hand combination:
Card 1: ##VALUE: 7 ##TYPE: 2
Card 2: ##VALUE: 4 ##TYPE: 4
Card 3: ##VALUE: -10 ##TYPE: -10
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -6 ##TYPE: -6
## High card! ##

Outs: 4
Chance of hitting an out: 8%
-----------------------------------------
## Player nr. 7 ##
7-card sorted sequense:
Card 1: ##VALUE: -2 ##TYPE: -2
Card 2: ##VALUE: -4 ##TYPE: -4
Card 3: ##VALUE: -6 ##TYPE: -6
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -10 ##TYPE: -10
Card 6: ##VALUE: 3 ##TYPE: 3
Card 7: ##VALUE: 7 ##TYPE: 4
Best hand combination:
Card 1: ##VALUE: 7 ##TYPE: 4
Card 2: ##VALUE: 3 ##TYPE: 3
Card 3: ##VALUE: -10 ##TYPE: -10
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -6 ##TYPE: -6
## High card! ##

Outs: 3
Chance of hitting an out: 6%
-----------------------------------------
```

```
## Player nr. 8 ##
7-card sorted sequense:
Card 1: ##VALUE: -2 ##TYPE: -2
Card 2: ##VALUE: -4 ##TYPE: -4
Card 3: ##VALUE: -6 ##TYPE: -6
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -10 ##TYPE: -10
Card 6: ##VALUE: 9 ##TYPE: 2
Card 7: ##VALUE: 10 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 10 ##TYPE: 1
Card 2: ##VALUE: 9 ##TYPE: 2
Card 3: ##VALUE: -10 ##TYPE: -10
Card 4: ##VALUE: -8 ##TYPE: -8
Card 5: ##VALUE: -6 ##TYPE: -6
## High card! ##

Outs: 4
Chance of hitting an out: 8%
-----------------------------------------
################################################################################################
######################################### FLOP ###############################################
################################################################################################
## Player nr. 1 ##
7-card sorted sequense:
Card 1: ##VALUE: -8 ##TYPE: -8
Card 2: ##VALUE: -10 ##TYPE: -10
Card 3: ##VALUE: 1 ##TYPE: 3
Card 4: ##VALUE: 2 ##TYPE: 1
Card 5: ##VALUE: 3 ##TYPE: 1
Card 6: ##VALUE: 4 ##TYPE: 3
Card 7: ##VALUE: 8 ##TYPE: 2
Best hand combination:
Card 1: ##VALUE: 8 ##TYPE: 2
Card 2: ##VALUE: 4 ##TYPE: 3
Card 3: ##VALUE: 3 ##TYPE: 1
Card 4: ##VALUE: 2 ##TYPE: 1
Card 5: ##VALUE: 1 ##TYPE: 3
## High card! ##
```

```
Outs: 14
Chance of hitting an out: 29%
-----------------------------------------
## Player nr. 2 ##
7-card sorted sequense:
Card 1: ##VALUE: -8 ##TYPE: -8
Card 2: ##VALUE: -10 ##TYPE: -10
Card 3: ##VALUE: 1 ##TYPE: 3
Card 4: ##VALUE: 2 ##TYPE: 1
Card 5: ##VALUE: 3 ##TYPE: 1
Card 6: ##VALUE: 10 ##TYPE: 3
Card 7: ##VALUE: 13 ##TYPE: 4
Best hand combination:
Card 1: ##VALUE: 13 ##TYPE: 4
Card 2: ##VALUE: 10 ##TYPE: 3
Card 3: ##VALUE: 3 ##TYPE: 1
Card 4: ##VALUE: 2 ##TYPE: 1
Card 5: ##VALUE: 1 ##TYPE: 3
## High card! ##

Outs: 10
Chance of hitting an out: 21%
-----------------------------------------
## Player nr. 3 ##
7-card sorted sequense:
Card 1: ##VALUE: -8 ##TYPE: -8
Card 2: ##VALUE: -10 ##TYPE: -10
Card 3: ##VALUE: 1 ##TYPE: 1
Card 4: ##VALUE: 1 ##TYPE: 3
Card 5: ##VALUE: 2 ##TYPE: 1
Card 6: ##VALUE: 3 ##TYPE: 4
Card 7: ##VALUE: 3 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 1 ##TYPE: 1
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 3 ##TYPE: 4
Card 4: ##VALUE: 3 ##TYPE: 1
Card 5: ##VALUE: 0 ##TYPE: 0
## Two Pair! ##
```

```
Outs: 2
Chance of hitting an out: 4%
-----------------------------------------
## Player nr. 4 ##
7-card sorted sequense:
Card 1: ##VALUE: -8 ##TYPE: -8
Card 2: ##VALUE: -10 ##TYPE: -10
Card 3: ##VALUE: 1 ##TYPE: 2
Card 4: ##VALUE: 1 ##TYPE: 3
Card 5: ##VALUE: 2 ##TYPE: 1
Card 6: ##VALUE: 3 ##TYPE: 1
Card 7: ##VALUE: 10 ##TYPE: 2
Best hand combination:
Card 1: ##VALUE: 1 ##TYPE: 2
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 0 ##TYPE: 0
Card 4: ##VALUE: 0 ##TYPE: 0
Card 5: ##VALUE: 0 ##TYPE: 0
## Pair! ##

Outs: 7
Chance of hitting an out: 14%
-----------------------------------------
## Player nr. 5 ##
7-card sorted sequense:
Card 1: ##VALUE: -8 ##TYPE: -8
Card 2: ##VALUE: -10 ##TYPE: -10
Card 3: ##VALUE: 1 ##TYPE: 3
Card 4: ##VALUE: 2 ##TYPE: 1
Card 5: ##VALUE: 3 ##TYPE: 2
Card 6: ##VALUE: 3 ##TYPE: 1
Card 7: ##VALUE: 5 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 3 ##TYPE: 2
Card 2: ##VALUE: 3 ##TYPE: 1
Card 3: ##VALUE: 0 ##TYPE: 0
Card 4: ##VALUE: 0 ##TYPE: 0
Card 5: ##VALUE: 0 ##TYPE: 0
## Pair! ##
```

```
Outs: 11
Chance of hitting an out: 23%
-----------------------------------------
## Player nr. 6 ##
7-card sorted sequense:
Card 1: ##VALUE: -8 ##TYPE: -8
Card 2: ##VALUE: -10 ##TYPE: -10
Card 3: ##VALUE: 1 ##TYPE: 3
Card 4: ##VALUE: 2 ##TYPE: 1
Card 5: ##VALUE: 3 ##TYPE: 1
Card 6: ##VALUE: 4 ##TYPE: 4
Card 7: ##VALUE: 7 ##TYPE: 2
Best hand combination:
Card 1: ##VALUE: 7 ##TYPE: 2
Card 2: ##VALUE: 4 ##TYPE: 4
Card 3: ##VALUE: 3 ##TYPE: 1
Card 4: ##VALUE: 2 ##TYPE: 1
Card 5: ##VALUE: 1 ##TYPE: 3
## High card! ##

Outs: 13
Chance of hitting an out: 27%
-----------------------------------------
## Player nr. 7 ##
7-card sorted sequense:
Card 1: ##VALUE: -8 ##TYPE: -8
Card 2: ##VALUE: -10 ##TYPE: -10
Card 3: ##VALUE: 1 ##TYPE: 3
Card 4: ##VALUE: 2 ##TYPE: 1
Card 5: ##VALUE: 3 ##TYPE: 3
Card 6: ##VALUE: 3 ##TYPE: 1
Card 7: ##VALUE: 7 ##TYPE: 4
Best hand combination:
Card 1: ##VALUE: 3 ##TYPE: 3
Card 2: ##VALUE: 3 ##TYPE: 1
Card 3: ##VALUE: 0 ##TYPE: 0
Card 4: ##VALUE: 0 ##TYPE: 0
Card 5: ##VALUE: 0 ##TYPE: 0
## Pair! ##
```

```
Outs: 8
Chance of hitting an out: 17%
-----------------------------------------
## Player nr. 8 ##
7-card sorted sequense:
Card 1: ##VALUE: -8 ##TYPE: -8
Card 2: ##VALUE: -10 ##TYPE: -10
Card 3: ##VALUE: 1 ##TYPE: 3
Card 4: ##VALUE: 2 ##TYPE: 1
Card 5: ##VALUE: 3 ##TYPE: 1
Card 6: ##VALUE: 9 ##TYPE: 2
Card 7: ##VALUE: 10 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 10 ##TYPE: 1
Card 2: ##VALUE: 9 ##TYPE: 2
Card 3: ##VALUE: 3 ##TYPE: 1
Card 4: ##VALUE: 2 ##TYPE: 1
Card 5: ##VALUE: 1 ##TYPE: 3
## High card! ##

Outs: 10
Chance of hitting an out: 21%
-----------------------------------------
################################################################################
######################################## TURN ####################################
################################################################################
## Player nr. 1 ##
7-card sorted sequense:
Card 1: ##VALUE: -10 ##TYPE: -10
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 2 ##TYPE: 1
Card 4: ##VALUE: 3 ##TYPE: 1
Card 5: ##VALUE: 4 ##TYPE: 3
Card 6: ##VALUE: 8 ##TYPE: 2
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 11 ##TYPE: 1
Card 2: ##VALUE: 8 ##TYPE: 2
Card 3: ##VALUE: 4 ##TYPE: 3
Card 4: ##VALUE: 3 ##TYPE: 1
```

```
Card 5: ##VALUE: 2 ##TYPE: 1
## High card! ##

Outs: 17
Chance of hitting an out: 36%
-----------------------------------------
## Player nr. 2 ##
7-card sorted sequense:
Card 1: ##VALUE: -10 ##TYPE: -10
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 2 ##TYPE: 1
Card 4: ##VALUE: 3 ##TYPE: 1
Card 5: ##VALUE: 10 ##TYPE: 3
Card 6: ##VALUE: 11 ##TYPE: 1
Card 7: ##VALUE: 13 ##TYPE: 4
Best hand combination:
Card 1: ##VALUE: 13 ##TYPE: 4
Card 2: ##VALUE: 11 ##TYPE: 1
Card 3: ##VALUE: 10 ##TYPE: 3
Card 4: ##VALUE: 3 ##TYPE: 1
Card 5: ##VALUE: 2 ##TYPE: 1
## High card! ##

Outs: 17
Chance of hitting an out: 36%
-----------------------------------------
## Player nr. 3 ##
7-card sorted sequense:
Card 1: ##VALUE: -10 ##TYPE: -10
Card 2: ##VALUE: 1 ##TYPE: 1
Card 3: ##VALUE: 1 ##TYPE: 3
Card 4: ##VALUE: 2 ##TYPE: 1
Card 5: ##VALUE: 3 ##TYPE: 4
Card 6: ##VALUE: 3 ##TYPE: 1
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 1 ##TYPE: 1
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 3 ##TYPE: 4
Card 4: ##VALUE: 3 ##TYPE: 1
```

```
Card 5: ##VALUE: 0 ##TYPE: 0
## Two Pair! ##

Outs: 10
Chance of hitting an out: 21%
-----------------------------------------
## Player nr. 4 ##
7-card sorted sequense:
Card 1: ##VALUE: -10 ##TYPE: -10
Card 2: ##VALUE: 1 ##TYPE: 2
Card 3: ##VALUE: 1 ##TYPE: 3
Card 4: ##VALUE: 2 ##TYPE: 1
Card 5: ##VALUE: 3 ##TYPE: 1
Card 6: ##VALUE: 10 ##TYPE: 2
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 1 ##TYPE: 2
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 0 ##TYPE: 0
Card 4: ##VALUE: 0 ##TYPE: 0
Card 5: ##VALUE: 0 ##TYPE: 0
## Pair! ##

Outs: 10
Chance of hitting an out: 21%
-----------------------------------------
## Player nr. 5 ##
7-card sorted sequense:
Card 1: ##VALUE: -10 ##TYPE: -10
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 2 ##TYPE: 1
Card 4: ##VALUE: 3 ##TYPE: 2
Card 5: ##VALUE: 3 ##TYPE: 1
Card 6: ##VALUE: 5 ##TYPE: 1
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 3 ##TYPE: 2
Card 2: ##VALUE: 3 ##TYPE: 1
Card 3: ##VALUE: 0 ##TYPE: 0
Card 4: ##VALUE: 0 ##TYPE: 0
```

```
Card 5: ##VALUE: 0 ##TYPE: 0
## Pair! ##

Outs: 20
Chance of hitting an out: 43%
-----------------------------------------
## Player nr. 6 ##
7-card sorted sequense:
Card 1: ##VALUE: -10 ##TYPE: -10
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 2 ##TYPE: 1
Card 4: ##VALUE: 3 ##TYPE: 1
Card 5: ##VALUE: 4 ##TYPE: 4
Card 6: ##VALUE: 7 ##TYPE: 2
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 11 ##TYPE: 1
Card 2: ##VALUE: 7 ##TYPE: 2
Card 3: ##VALUE: 4 ##TYPE: 4
Card 4: ##VALUE: 3 ##TYPE: 1
Card 5: ##VALUE: 2 ##TYPE: 1
## High card! ##

Outs: 16
Chance of hitting an out: 34%
-----------------------------------------
## Player nr. 7 ##
7-card sorted sequense:
Card 1: ##VALUE: -10 ##TYPE: -10
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 2 ##TYPE: 1
Card 4: ##VALUE: 3 ##TYPE: 3
Card 5: ##VALUE: 3 ##TYPE: 1
Card 6: ##VALUE: 7 ##TYPE: 4
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 3 ##TYPE: 3
Card 2: ##VALUE: 3 ##TYPE: 1
Card 3: ##VALUE: 0 ##TYPE: 0
Card 4: ##VALUE: 0 ##TYPE: 0
```

```
Card 5: ##VALUE: 0 ##TYPE: 0
## Pair! ##

Outs: 11
Chance of hitting an out: 23%
-----------------------------------------
## Player nr. 8 ##
7-card sorted sequense:
Card 1: ##VALUE: -10 ##TYPE: -10
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 2 ##TYPE: 1
Card 4: ##VALUE: 3 ##TYPE: 1
Card 5: ##VALUE: 9 ##TYPE: 2
Card 6: ##VALUE: 10 ##TYPE: 1
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 11 ##TYPE: 1
Card 2: ##VALUE: 10 ##TYPE: 1
Card 3: ##VALUE: 9 ##TYPE: 2
Card 4: ##VALUE: 3 ##TYPE: 1
Card 5: ##VALUE: 2 ##TYPE: 1
## High card! ##

Outs: 19
Chance of hitting an out: 41%
-----------------------------------------
################################################################################
######################################## RIVER ########################################
################################################################################
## Player nr. 1 ##
7-card sorted sequense:
Card 1: ##VALUE: 1 ##TYPE: 3
Card 2: ##VALUE: 2 ##TYPE: 1
Card 3: ##VALUE: 3 ##TYPE: 1
Card 4: ##VALUE: 4 ##TYPE: 3
Card 5: ##VALUE: 8 ##TYPE: 2
Card 6: ##VALUE: 9 ##TYPE: 3
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 1 ##TYPE: 3
```

```
Card 2: ##VALUE: 11 ##TYPE: 1
Card 3: ##VALUE: 9 ##TYPE: 3
Card 4: ##VALUE: 8 ##TYPE: 2
Card 5: ##VALUE: 4 ##TYPE: 3
## High card! ##

Outs: 19
Chance of hitting an out: 42%
-----------------------------------------
## Player nr. 2 ##
7-card sorted sequense:
Card 1: ##VALUE: 1 ##TYPE: 3
Card 2: ##VALUE: 2 ##TYPE: 1
Card 3: ##VALUE: 3 ##TYPE: 1
Card 4: ##VALUE: 9 ##TYPE: 3
Card 5: ##VALUE: 10 ##TYPE: 3
Card 6: ##VALUE: 11 ##TYPE: 1
Card 7: ##VALUE: 13 ##TYPE: 4
Best hand combination:
Card 1: ##VALUE: 1 ##TYPE: 3
Card 2: ##VALUE: 13 ##TYPE: 4
Card 3: ##VALUE: 11 ##TYPE: 1
Card 4: ##VALUE: 10 ##TYPE: 3
Card 5: ##VALUE: 9 ##TYPE: 3
## High card! ##

Outs: 19
Chance of hitting an out: 42%
-----------------------------------------
## Player nr. 3 ##
7-card sorted sequense:
Card 1: ##VALUE: 1 ##TYPE: 1
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 2 ##TYPE: 1
Card 4: ##VALUE: 3 ##TYPE: 4
Card 5: ##VALUE: 3 ##TYPE: 1
Card 6: ##VALUE: 9 ##TYPE: 3
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 1 ##TYPE: 1
```

```
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 3 ##TYPE: 4
Card 4: ##VALUE: 3 ##TYPE: 1
Card 5: ##VALUE: 0 ##TYPE: 0
## Two Pair! ##

Outs: 10
Chance of hitting an out: 22%
-----------------------------------------
## Player nr. 4 ##
7-card sorted sequense:
Card 1: ##VALUE: 1 ##TYPE: 2
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 2 ##TYPE: 1
Card 4: ##VALUE: 3 ##TYPE: 1
Card 5: ##VALUE: 9 ##TYPE: 3
Card 6: ##VALUE: 10 ##TYPE: 2
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 1 ##TYPE: 2
Card 2: ##VALUE: 1 ##TYPE: 3
Card 3: ##VALUE: 0 ##TYPE: 0
Card 4: ##VALUE: 0 ##TYPE: 0
Card 5: ##VALUE: 0 ##TYPE: 0
## Pair! ##

Outs: 12
Chance of hitting an out: 26%
-----------------------------------------
## Player nr. 5 ##
7-card sorted sequense:
Card 1: ##VALUE: 1 ##TYPE: 3
Card 2: ##VALUE: 2 ##TYPE: 1
Card 3: ##VALUE: 3 ##TYPE: 2
Card 4: ##VALUE: 3 ##TYPE: 1
Card 5: ##VALUE: 5 ##TYPE: 1
Card 6: ##VALUE: 9 ##TYPE: 3
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 3 ##TYPE: 2
```

```
Card 2: ##VALUE: 3 ##TYPE: 1
Card 3: ##VALUE: 0 ##TYPE: 0
Card 4: ##VALUE: 0 ##TYPE: 0
Card 5: ##VALUE: 0 ##TYPE: 0
## Pair! ##

Outs: 21
Chance of hitting an out: 46%
-----------------------------------------
## Player nr. 6 ##
7-card sorted sequense:
Card 1: ##VALUE: 1 ##TYPE: 3
Card 2: ##VALUE: 2 ##TYPE: 1
Card 3: ##VALUE: 3 ##TYPE: 1
Card 4: ##VALUE: 4 ##TYPE: 4
Card 5: ##VALUE: 7 ##TYPE: 2
Card 6: ##VALUE: 9 ##TYPE: 3
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 1 ##TYPE: 3
Card 2: ##VALUE: 11 ##TYPE: 1
Card 3: ##VALUE: 9 ##TYPE: 3
Card 4: ##VALUE: 7 ##TYPE: 2
Card 5: ##VALUE: 4 ##TYPE: 4
## High card! ##

Outs: 18
Chance of hitting an out: 40%
-----------------------------------------
## Player nr. 7 ##
7-card sorted sequense:
Card 1: ##VALUE: 1 ##TYPE: 3
Card 2: ##VALUE: 2 ##TYPE: 1
Card 3: ##VALUE: 3 ##TYPE: 3
Card 4: ##VALUE: 3 ##TYPE: 1
Card 5: ##VALUE: 7 ##TYPE: 4
Card 6: ##VALUE: 9 ##TYPE: 3
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 3 ##TYPE: 3
```

```
Card 2: ##VALUE: 3 ##TYPE: 1
Card 3: ##VALUE: 0 ##TYPE: 0
Card 4: ##VALUE: 0 ##TYPE: 0
Card 5: ##VALUE: 0 ##TYPE: 0
## Pair! ##

Outs: 13
Chance of hitting an out: 28%
-----------------------------------------
## Player nr. 8 ##
7-card sorted sequense:
Card 1: ##VALUE: 1 ##TYPE: 3
Card 2: ##VALUE: 2 ##TYPE: 1
Card 3: ##VALUE: 3 ##TYPE: 1
Card 4: ##VALUE: 9 ##TYPE: 2
Card 5: ##VALUE: 9 ##TYPE: 3
Card 6: ##VALUE: 10 ##TYPE: 1
Card 7: ##VALUE: 11 ##TYPE: 1
Best hand combination:
Card 1: ##VALUE: 9 ##TYPE: 2
Card 2: ##VALUE: 9 ##TYPE: 3
Card 3: ##VALUE: 0 ##TYPE: 0
Card 4: ##VALUE: 0 ##TYPE: 0
Card 5: ##VALUE: 0 ##TYPE: 0
## Pair! ##

Outs: 18
Chance of hitting an out: 40%
-----------------------------------------
############################
### Winner is Player 3!! ###
############################
```

–The commented code is delivered separately

9.5 Small ppm preprocessor function

Due to ppm files not having an alpha channel and having a superflous ascii header we wrote a small javaprogram to preprocess images.

```
public static void convertFromPPmToTXTspecAlFinal(String inputPath,
   String OutputPath, int fileSize,int digitsInAspectratio, int col1,
    int col2, int col3){
        FileInputStream fileInputStream;
        FileOutputStream fileOutputStream;
        File file = new File(inputPath);
        File file2 = new File(OutputPath);
        byte[] trashbuffer = new byte[32+digitsInAspectratio];
        byte[] rgb = new byte[3];
        byte[] fullpicturebuffer = new byte[fileSize];
        int currentloc = 0;
        try {
            //read past the trash
            fileOutputStream = new FileOutputStream(file2);
            fileInputStream = new FileInputStream(file);
            fileInputStream.read(trashbuffer);
            for(int i = 0; i < 32+digitsInAspectratio; i++){
                System.out.print((char)trashbuffer[i]);
            }
            //fill up the actual buffer
            while(fileInputStream.read(rgb) != -1){
                if((Byte.toUnsignedInt(rgb[0])  == col1) && (Byte.
   toUnsignedInt(rgb[1])  == col2) && (Byte.toUnsignedInt(rgb[2])  ==
    col3)){
                    fullpicturebuffer[currentloc] = rgb[2];
                    fullpicturebuffer[currentloc+1] = rgb[1];
                    fullpicturebuffer[currentloc+2] = rgb[0];
                    fullpicturebuffer[currentloc+3] = (byte)255;
                    System.out.println("alfa match");
                }else{
                    fullpicturebuffer[currentloc] = rgb[2];
                    fullpicturebuffer[currentloc+1] = rgb[1];
                    fullpicturebuffer[currentloc+2] = rgb[0];
                    fullpicturebuffer[currentloc+3] = 0;
                }
                System.out.println("" + Byte.toUnsignedInt(rgb[0]) +
   " "+ Byte.toUnsignedInt(rgb[1])  + " "+ Byte.toUnsignedInt(rgb[2])
    );
                currentloc = currentloc + 4;
            }
            fileOutputStream.write(fullpicturebuffer);

            fileOutputStream.close();
            fileInputStream.close();

        }catch (IOException ioException){
            ioException.printStackTrace();
        }

```

```
44          }
```

Listing 14: ppmconverter