

Game Programming 2009

iPhone Yahtzee Game: iRoll

Name: Steven Roebert
E-mail: sroebert@science.uva.nl

1 Introduction

Mobile games are becoming more and more popular. The mobile devices used for gaming (e.g. mobile phones, handhelds, etc) are incredibly powerful compared to some years ago and the possibilities seem endless [5, 2]. The additions of touchscreen, camera's, GPS, accelerometer and other sensors make for interesting additions in games. The iPhone especially has become a huge platform for mobile games. One of the reasons for its success can be found in the App Store. The App Store is a service for the iPhone, iPod Touch and iPad which allows users to easily browse and find applications for their device. Furthermore it gives developers the opportunity to easily distribute their applications to the public. In a short time thousands of games have been created for these devices.

As a project for a Game Programming course, I have decided to build my own game for the iPhone. As I want to add knowledge from Artificial Intelligence and keep the application from becoming too complicated I decided to build my own version of a Yahtzee game. Using techniques from Artificial Intelligence, like edge detection [3] and component labeling [4], I will use the built-in camera of the iPhone to automatically detect dice rolls. Also suggestions based on what a player has rolled will be given, in order to help the player reach the highest score possible. The latter part will be generated by calculating the full state graph of the Yahtzee game and following the optimal strategy [6].

This paper consists of several sections explaining the challenges of building the application. Section 2 will explain the rules of Yahtzee, for those people who do not know the game. Section 3 goes into the details of dice roll detection. In section 4 I will talk about adding suggestions to the application. Next in section 5 the different views of the application will be discussed. Finally section 6 contains some conclusions I have drawn and section 7 discusses possible future work.



Figure 1: iPhone 3GS

2 Game Rules

The goal of Yahtzee is simple, get the highest possible score. To get points, a player has to roll dice to get specific combinations. At each turn a player can roll the dice up to 3 times. The first time all dice have to be rolled. For the second and third roll the player can decide to keep a number of dice and re-roll the remaining ones. At the end of a turn the player has to choose a category to place the score. Each score category can only be used once. If the player cannot choose a suitable category for the points, a score of zero has to be placed at one of the categories instead. At the end of the game all points are summed up to get the final score.

The scoring is divided into two sections, the upper and lower section. The upper section contains a category for each die face. The score for each category is the sum of all dice that have the corresponding value. So for example:



will result in a score of 9 if a player chooses the Threes category.

The upper section also contains a special bonus. This bonus will be awarded when the player has a total score of 63 or more in the upper section categories.

The lower section contains the following categories.

Three of a kind

At least three dice showing the same face

Score: Sum of all dice



Four of a kind

At least four dice showing the same face

Score: Sum of all dice



Full House

Two plus three dice showing the same face

Score: 25



Small Straight

Four sequential dice

Score: 30



Large Straight

Five sequential dice

Score: 40



Yahtzee

All five dice showing the same face

Score: 50



Chance

Any combination

Score: Sum of all dice



Now the rules are known, the next two sections will explain specific internal parts of the application.

3 Dice Roll Detection

One of the main ideas for the application has been to implement a method for detecting a dice roll automatically. Since the iPhone has a built-in camera, a user could simply make a picture of the dice he rolled and the application should be able to detect the faces of each die.

The iPhone does not have the processing power of desktop computer, but instead has to work with less processing power, significantly less memory and a battery. This makes it crucial for the code to be efficient and fast. It would not be ideal for a user to have to wait more than a few seconds before a dice roll has been detected. If that was the case, a user would obviously return back to the old method of simply writing down the score by hand.

3.1 Edge Detection

The algorithm to detect a dice roll consists of a few steps. The first step is finding the outlines of all the dice in the image. An example of an image taken by the iPhone's camera can be seen in Figure 2(a).

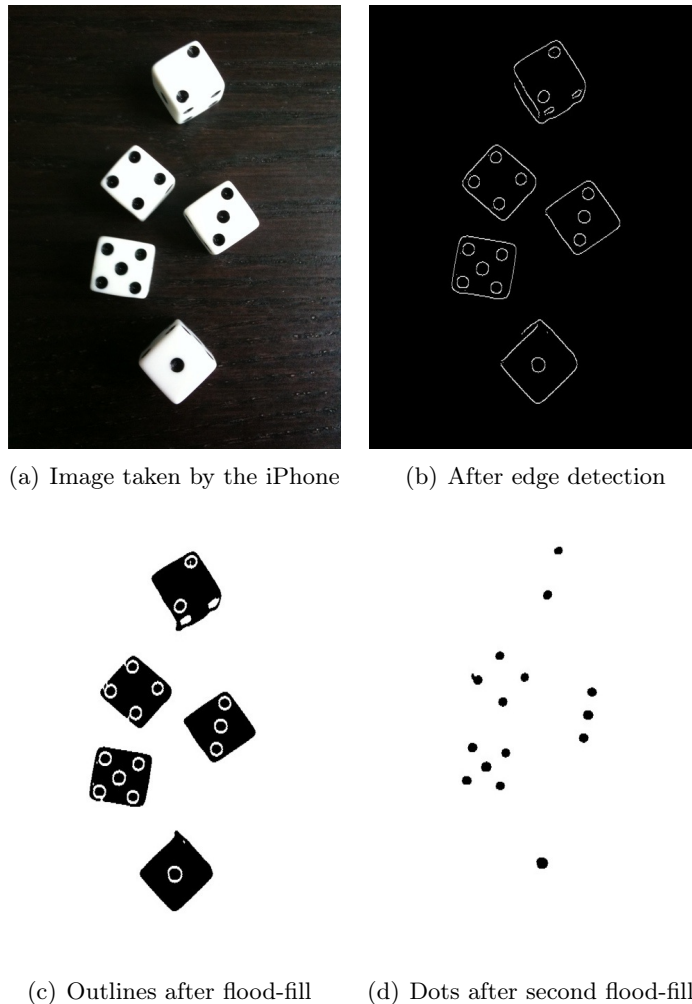


Figure 2: Four stages in the dice face detection algorithm

In order to find the dice, a Canny edge detector [3] was used. A Canny edge detector will detect sharp edges in image brightness. It does this by first convolving the image

with a Gaussian filter, removing the effect of noise in the image. After convolving a series of four filters are used to detect horizontal, vertical and diagonal edges in the image. This results in leaving only the most important objects of the image, which in our case should be the dice. If dice with a large contrast between dot and surface colour are used (e.g. white dice with black dots) the edge detection algorithm should also be able to find the edges of the dots of the dice. Although detecting edges is fast and efficient, it will require that a background with a clear contrast between the dice is used. For example, when using default white dice, a white background would result in poor edge detection, which in effect causes the detection algorithm to fail.

After performing the edge detection, some noise might exist. The edge detection could find some smaller edges in the background for example. This noise can easily be removed by eroding and dilating the image. This result in an image like Figure 2(b). As can be seen, all the outlines of the dice, including the dots inside them have been found. The next step in the algorithm is to find the actual dice locations from these outlines.

3.2 Connected Components Labeling

To find the die locations, a clear distinction between dice and background has to exist. The edge detection removes this (i.e. only white lines on a black background remain), but some assumptions can be made in order to get this distinction back in the image. For the edge detection algorithm to work correctly, all the dice in the image need to be completely visible (i.e. the dice should not overlap the borders of the image). If this is the case the outer border of the image will be part of the background. This property can be used by a flood fill algorithm (see Algorithm 1) to fill the background of the image. The resulting image after performing the flood fill algorithm can be seen in Figure 2(c). In this image, the interior of the dice are now the only black parts in the image.

Algorithm 1 *FloodFill(pixel, target_color, replacement_color)*

```

1: if pixel.color  $\neq$  target_color then
2:   return
3: end if
4: pixel.color  $\leftarrow$  replacement_color
5: FloodFill(pixel.north, target_color, replacement_color)
6: FloodFill(pixel.east, target_color, replacement_color)
7: FloodFill(pixel.south, target_color, replacement_color)
8: FloodFill(pixel.west, target_color, replacement_color)
9: return

```

For extracting the black dice parts, a component-labeling algorithm by Fu Chang et al. [4] is used. This algorithm scans a binary image for connected parts, while leaving out the background. Each non-background pixel will be assigned to a specific component. To remove noise, very small components will be filtered from this result. As these components might also include the dots of the dice and our interest is currently only with the dice itself, overlapping components are combined into one large component. So now we have all the contours for each die.

3.3 Dot Detection

Finally, the dots for each die separately have to be detected in order to identify the dice roll. One method for detecting dots would be to use the Circular Hough Transform [1]. This method successfully detects circles in a binary image and is often used together with edge detection. In our case however, the dots are rather small. Because of this, when

the image is not taken directly over the dice (e.g. with a small angle from the side), the dots will no longer be circles. What you will end up with instead are ellipsoids, which will be a lot less likely to be detected by the Circular Hough Transform.

An alternative method used in the application is to simply reproduce the steps for detecting the die locations. As the dots do not touch the outer edges of each die, the flood fill algorithm can be used to fill the die with a background colour. For the flood fill algorithm to work correctly, obviously the edges of each dot will have to be fully connected. To make sure of this, each die image is dilated by a small amount. After the flood fill algorithm (see Figure 2(d)), the component-labeling algorithm is used once again, this time to find all the dots inside the die. The resulting components are once again filtered to remove noise. Now the value of the die face is simply the number of found components.

3.4 Limitations

Even though the edge detection, combined with component labeling is an efficient and fast method, it does have its limitations. There has to be a good contrast between dice and background, but also between the dice face colour and the colour of the dots. Furthermore each die must be completely visible inside the image, without edges overlapping. Also the dice have to lie somewhat apart, otherwise the edge detection might result in one large die instead of two separate ones. Finally, the dice used, must contain sharp edges, instead of round ones which would cause the edge detection algorithm to fail. In the end, all these constraints can become quite a burden to an end-user of the application.

To try and overcome some of these limitations, I looked into Scale-invariant feature transform (SIFT) descriptors [8, 9] as an alternative to the method described above. SIFT descriptors are used to detect and describe local features in images. It is widely used in object recognition, as it is invariant to scale, orientation and affine distortion and partially invariant to illumination. Given these properties it would also be suitable for dice roll detection and would solve some of the current limitations.

Unfortunately after implementing SIFT descriptors, the first noticeable difference with the other algorithm is speed. Detecting SIFT descriptors is more computational intensive than edge and component detection. This problem could however be circumvented by optimizing the algorithm and searching for less descriptors. Recognition however does not directly work as expected. The most significant problem is that even though the SIFT descriptors can detect the presence of a die in the image, it does not seem to detect the difference between the different die faces. So a SIFT-only implementation will never be able to solve our problem.

4 Suggestions

Apart from detecting dice rolls using the camera, another feature of the application would be to present the user with suggestions. Suggestions could include which dice to keep and which to re-roll or what score to choose at the end of a turn. One way of doing this is to calculate the probabilities for each possible roll and choosing the most likely. For scores you could simply select the scores with the highest values, but this will result in suboptimal solutions. Instead a more thorough way would be to calculate the optimal strategy by calculating the values for all possible states in the game. For this part I have followed the approach used by James Glenn [6, 7], which I will discuss in more detail in this section.

4.1 Using a state graph

The general idea of this approach is to envision the game as a graph. In the case of Yahtzee, this graph will consist of so called widgets (this is the term used in the article by James Glenn). Each widget will represent a player's turn. For each turn there will be two parameters, namely the currently used score categories and the total score for the upper section. The latter part is used for checking whether the player has gained a bonus of 35 points, which is when you have 63 or more points in the upper section. Each individual widget will have a unique combination of these two parameters (we leave out any Yahtzee bonuses, as this will not be part of the application). This will result in a total of 2^{19} widgets.

Each widget in turn consists of all the states for a player's turn. These will obviously be the same for each widget, as the rolls for each Yahtzee turn are independent of the rolls in previous turns. Figure 4.1 shows the structure of the state graph, it shows three connected widgets in total as an example. Each widget contains numbered states, which I will now explain in more detail.

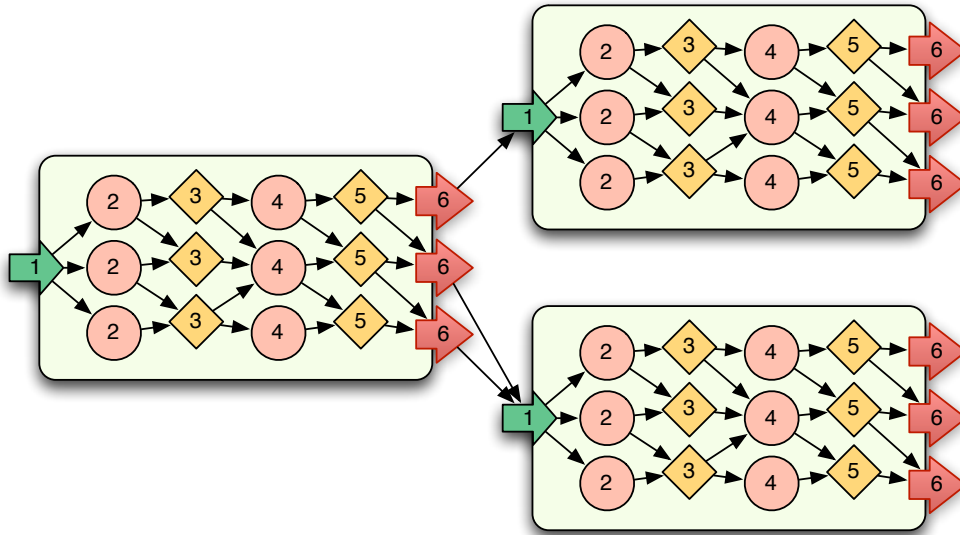


Figure 3: Graph structure showing three connected widgets

A widget consists of the following groups:

1. An entry point, representing the beginning of the turn.
2. Each state for the outcome of the first dice roll (roll states).
3. Each state for the dice a player keeps after the first roll (keep states).
4. Each state for the outcome of the second dice roll (roll states).
5. Each state for the dice a player keeps after the second roll (keep states).
6. Each state for the outcome of the final dice roll (roll states).

Each group of states in a widget has edges to the next group. The final group of each widget can be connected to the first group of possible next widgets. If you do this for all widgets you get one big graph which will be used for calculating the best possible strategy.

4.2 Computation

To get to the best possible Yahtzee strategy, we will start our calculations at the end of the state graph (i.e. the states in group 6 of the widgets where all score categories are used). From there we will work our way back to the beginning of the graph, each time assigning a specific value to each state we pass. I will call these values state potentials. When all calculations are finished, the best strategy can be found by choosing the keep states which have the highest potential and choosing the scores that will make you end up in the next widget with the highest potential.

I will use the same notations as used by James Glenn, but for clarity I will give a short explanation for each notation.

- S is a widget, which consists of two values (i.e. the currently used score categories and the total score for the upper section)
- (C, m) is a widget, C is the currently used score categories, m is the total score for the upper section
- $E(S, r, n)$ is the potential for the roll state in widget S where the player just rolled r and has n re-rolls remaining
- $E(S, r', n)$ is the potential for the keep state in widget S where the player has chosen to keep r' and has n re-rolls remaining
- $P(r' \rightarrow r)$ is the probability of rolling r after keeping r'
- $P(\perp \rightarrow r)$ is the probability of rolling r as the initial roll
- $R_{i,j}$ is the set of outcomes when rolling i j -sided dice
- $s(r, c)$ denotes the score obtained by scoring the roll r in category c
- $n(S, r, c)$ denotes the new widget after starting in widget S and rolling the final roll r and using category c

The potentials for the final states, will either be 0 or 35, depending on whether the player has received the bonus for the upper score or not.

Next are the states in group 5, which are the states where a player has just chosen which dice to keep after the second roll. The potentials for these states are calculated by taking the weighted sum of each possible roll state in group 6 you can reach after rolling for the third time.

So for example, you have rolled [1 3 4 5 6] and you decide to keep [3 4 5 6] and re-roll the 1. To get the potential for that keep state, you calculate the sum of the potential of each next group state with [3 4 5 6 *] (where * can be any number) as the dice roll, times the probability of getting into this state when re-rolling the 1 (which in this case will be $\frac{1}{6}$ for each state).

$$E(S, r', n) = \sum_{r \in R_{5,6}} P(r' \rightarrow r) \cdot E(S, r, n - 1) \quad (1)$$

Then for group 4, which is when a player has just rolled for the second time, you simply take the maximum potential of all possible next states in group 5. Because a player can choose which dice to keep, you do not have to weight this potential as the player can always choose the best possible state.

$$E(S, r, n) = \max_{r' \subseteq r} E(S, r', n) \quad (2)$$

Group 3 will be the same as group 5 and group 2 will be the same as group 4, since these are also keep and roll states respectively. Next the potential for the state in group 1 has to be calculated. This will also be the potential for the current widget. As for the first roll a player has to roll all dice, every possible roll state can be reached. This means to get the potential we simply take the weighted sum of all the state potentials in group 2.

$$E(S) = \sum_{r \in R_{5,6}} P(\perp \rightarrow r) \cdot E(S, r, 2) \quad (3)$$

Finally when group 1 has been reached, we have to make the step to group 6 of the previous widget. Once again, the player can take a choice here. The player can decide which score category to use. So to calculate the potential for group 6 of non final states, we take the maximum for the score of each unused category plus the potential of the next widget.

$$E((C, m), r, 0) = \max_{c \notin C} s((C, m), r, c) + E(n((C, m), r, c)) \quad (4)$$

Now we are in the next widget and the process can be repeated until each widget in the graph has been reached and all potentials are calculated.

4.3 In the application

As James Glen discussed, it takes quite some time to calculate all these potentials. It can definitely not be done in real time. In order to solve this problem, one can simply calculate all the widget potentials once and store them in a file. For 2^{19} widgets, this will take up 2.1MB of disk space (each widget value will be stored as a float). Even for the iPhone this will not be considered a large amount of space. The application then reads in this data when needed and only has to calculate the potentials for all state groups inside a widget, which will take only a fraction of a second. This also holds for the iPhone, which has a powerful enough processor to do the widget potential calculations in an unnoticeable amount of time. This means the optimal strategy can directly be used for suggestions on the iPhone. Another optimization I have added to the application is to save all the states and edges for one widget in a file, which can be loaded at runtime. Since all states inside a widget are the same for all widgets, it will be unnecessary to recalculate these every time.

For the application there will be two kinds of suggestions. First there are suggestions for which dice to keep. Second when choosing the score category, the best score will be suggested to the player. However, for both suggestions the same calculations are used. First the current state in the widget is found. Then all next states are taken and sorted by potential. The states with the highest potentials will be on top. For keep states, only the top three states will be suggested to the user. For score suggestions, all scores are shown, sorted based on their potential.

5 The Application

Before getting into details of the application, a bit of information about the iPhone SDK might be useful. The iPhone SDK can be downloaded for free from the Apple website¹. With it you get the Xcode developer tool, Interface Builder and the iPhone Simulator, all of which I have used for building the iRoll application. As with the Mac the default

¹<http://developer.apple.com/iphone>

programming language for the iPhone is Objective-C. This language is an extension for the C language, adding object oriented programming.

The iPhone Simulator lets you test your applications directly on your Mac. However, there are some limitations. The Simulator for obvious reasons does not have the accelerometer. It is also not possible to use a camera, even though your computer might have a webcam built-in. This makes testing the detection of dice quite difficult. In order to test your applications on your own device, you will have to participate in the iPhone Developer Program². This will grant you the certificates needed for installing your own applications on the iPhone, iPod Touch or iPad.

Apart from the iPhone SDK, I have also used the computer vision library OpenCV. This library makes it easy to perform specific computer vision techniques, like edge detection. By default it does not directly work on the iPhone, but a variety of tutorials³ can be found on the Internet on how to get it to work.

I will now be talking about the specific views in the application, explaining how the user can get through a game.

5.1 Startup

When starting the application, there are few choices for the player (see Figure 4(a)). Apart from starting a new game and continuing the last game, the player can read the rules of the game and view the about page. Continuing a game will be available as soon as the player has started a new game. When quitting the application, the game is automatically saved. So when the application is relaunched, the player can continue where he or she left off.



(a) The main menu

(b) Game setup view

Figure 4: Startup views of the application

If the player has chosen to start a new game, the player first has to choose how many

²<http://developer.apple.com/programs/iphone>

³<http://niw.at/articles/2009/03/14/using-opencv-on-iphone/en>

players will be joining the game (see Figure 4(b)). Up to eight different players can take part in a game. If more than one player will be playing, the players will take turns, each time filling in one score, in the order that was specified during setup. If you want, you can enter names for the players, which will make it easier to see whose turn it is.

5.2 Rolling the dice

At the beginning of a players turn, if there are multiple players playing, a message will briefly show, notifying whose turn it is (see Figure 5(a)). Now the player can roll the dice by clicking the "Roll" button. As an addition, the dice can also be rolled by shaking the device. As mentioned in section 2, after the first and second roll the player can choose which dice to keep and which dice to re-roll. By clicking on the individual dice a player can specify which dice to keep. If a die's face has turned green, it means the die will be kept during the next roll (see Figure 5(c)).

Instead of deciding which dice to keep by selecting them individually, a player can also decide to follow one of the suggestions. There will be up to three suggestions (see Figure 5(b), 5(c)), which will show up in a list underneath the dice. By clicking on a suggestion, the dice to keep will automatically be selected.

Another way of playing the game, of which the technique was discussed in section 3, is to take pictures of real dice you rolled. The application will then detect the faces of each die in the picture and you can use this dice roll to fill in a score. This way the player can use the application merely as a way of saving the score of the game and follow suggestions. If the device of the user has a built-in camera, a camera icon will show up in the top right corner of the application (during the rolling of the dice). If the player clicks this icon, a picture can be taken. Once the picture has been taken, the recognition will be done automatically, which will take up to a second.

5.3 Saving your score

At any time during a players turn, the player can check his or her score, by clicking the score button. The score view is a simple table containing all the categories, divided into the upper and lower sections (see Figure 6(a)). You can also find the total score for the upper and lower section, whether you have received the bonus and the combined total score. When a category has already been used, the score is grayed out and contains a checkmark on the right. If the category has not been used, the value on the right will indicate the score for the current roll. To fill in a category, the player simply selects the category and clicks the save button.

The score view also contains suggestions, as mentioned in section 4. The player can switch between default and suggested view mode, by clicking the appropriate button above the scores. When in suggested mode (see Figure 6(b)), the top category will be the correct category to pick if you were to follow the best strategy. The lower the category the less wise it would be to choose it. At the bottom you can still find the total score and the categories already used.

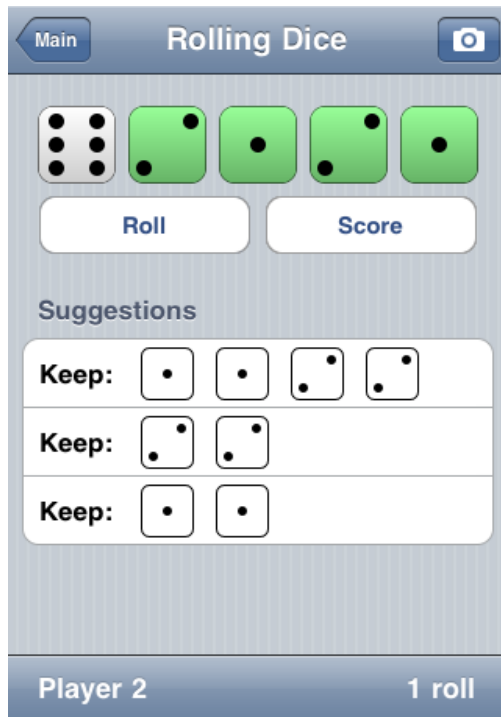
5.4 Final scores

Finally after filling in the last score category, the final score screen will popup. This view will simply show an overview of the scores for a player (see Figure 5.4). To view the scores of another player you can scroll horizontally through all the scores. Also if multiple players have participated in the game, a brief message will popup indicating the winner.



(a) Player notification

(b) After rolling the dice



(c) After selecting some dice to keep

Figure 5: The views for a players turn

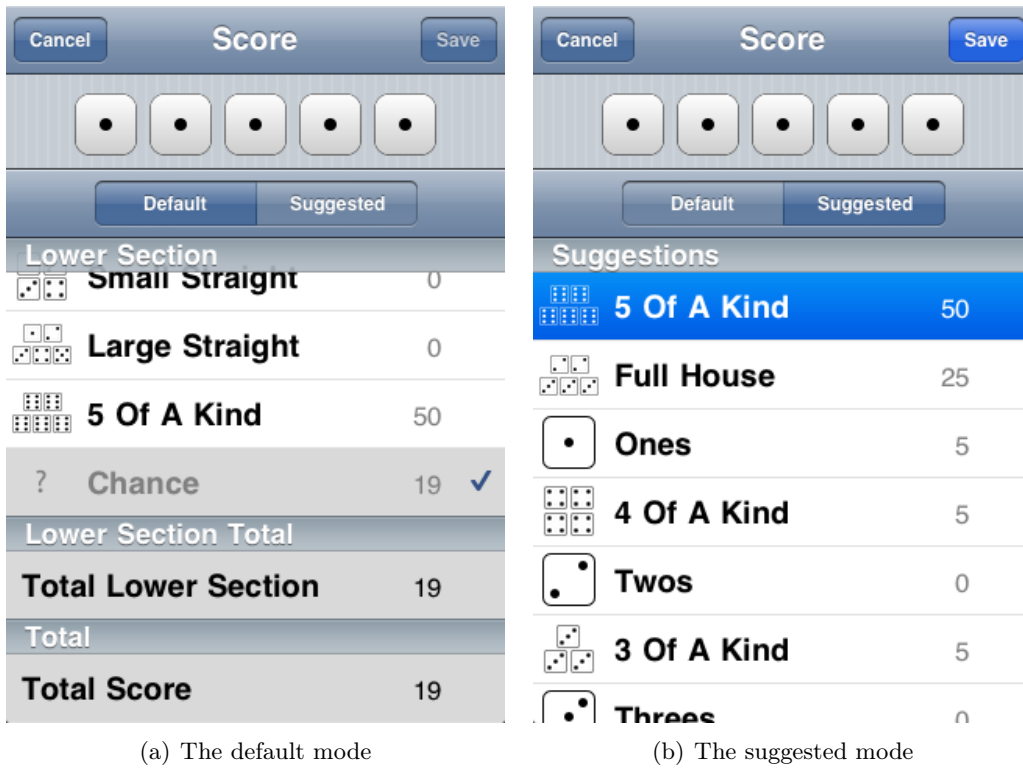


Figure 6: The views for selecting a score

Final Scores

Player 1

Ones:	2	Three of a kind:	23
Twos:	2	Four of a kind:	0
Threes:	12	Full House:	25
Fours:	16	Small Straight:	30
Fives:	10	Large Straight:	40
Sixes:	24	Five of a kind:	0
Bonus:	35	Chance:	18
Total upper section:		101	
Total lower section:		136	
Total:		237	

Figure 7: The final score view

After viewing the scores, the application will return back to the main menu and it will again be possible to start a new game. Also note that anytime during a game, the player can return to the main screen by simply clicking the main button in the top left corner. The game will then automatically be saved and the player can return by clicking the "Continue" button.

6 Conclusion

I have successfully created a working iPhone application from the ground up. The application allows the user to play a game of Yahtzee, with up to eight players at a time. The game itself not only follows the default rules of Yahtzee, it also gives suggestions to the player, based on the best possible strategy (either for choosing which dice to keep before a re-roll or which score to fill in at the end of the turn).

Furthermore, I have also implemented a way to recognize dice rolls from an image taken by the device's camera. This method is implemented with edge and contour detection algorithms, using the OpenCV library. Although the detection does not work flawlessly, it works under controlled conditions (section 3.4) and finishes almost instantly on the mobile device.

7 Future Work

Definitely the most room for improvement will be in the dice recognition part of the application. Right now the detection algorithm has quite some limitations, as discussed in section 3.4. As mentioned I have tried using SIFT to improve this, but SIFT alone does not seem to be able to detect a dice roll. Maybe combining SIFT with the current algorithm or a completely different approach might result in better dice recognition.

Other improvements in the application could be found in game extensions. A player might find it interesting to play against an artificial opponent. This could easily be realized by using the code for suggestions and instead letting the artificial player follow these suggestions to a certain degree. Also the possibility for connecting multiple iPhone for playing a multiplayer game of Yahtzee where every player can roll at the same time, seems like a nice extension.

Currently the application does not have any settings. It might also be an idea to add user settings, which for example could turn suggestions on or off. Right now, always playing with suggestions might take all the fun out of the game, as the user always knows what the best strategy will be. If suggestions can be turned off, the game might stay a bit more challenging.

References

- [1] DH Ballard. Generalizing the Hough transform to detect arbitrary shapes. *Pattern recognition*, 13(2):111–122, 1981.
- [2] M. Bell, M. Chalmers, L. Barkhuus, M. Hall, S. Sherwood, P. Tennent, B. Brown, D. Rowland, and S. Benford. Interweaving mobile games with everyday life. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, page 426. ACM, 2006.
- [3] J. Canny. A computational approach to edge detection. *Readings in computer vision: issues, problems, principles, and paradigms*, page 184, 1987.

- [4] F. Chang, C.J. Chen, and C.J. Lu. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision and Image Understanding*, 93(2):206–220, 2004.
- [5] D. Eriksson, J. Peitz, and S. Bjork. Enhancing board games with electronics. In *Proceedings of the 2nd International Workshop on Pervasive Games-PerGames*. Citeseer, 2005.
- [6] J. Glenn. An optimal strategy for Yahtzee. *Loyola College in Maryland, Tech. Rep. CS-TR-0002*, 2006.
- [7] J. Glenn. Computer strategies for solitaire yahtzee. In *IEEE Symposium on Computational Intelligence and Games (CIG 2007)*, pages 132–139. Citeseer, 2007.
- [8] D.G. Lowe. Object recognition from local scale-invariant features. In *iccv*, page 1150. Published by the IEEE Computer Society, 1999.
- [9] D.G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.