Computational Visual Media
DOI 10.1007/s41095-016-0047-3 Vol. 2, No. 2, June 2016, 183–193

Research Article

Augmented reality system to assist inexperienced pool players

L. Sousa[1] (✉), **R. Alves**[1], **and J. M. F. Rodrigues**[2]

Abstract Pool and billiards are amongst a family of games played on a table with six pockets along the rails. This paper presents an augmented reality tool designed to assist unskilled or amateur players of such games. The system is based on a projector and a Kinect 2 sensor placed above the table, acquiring and processing the game on-the-fly. By using depth information and detecting the table's rails (borders), the balls' positions, the cue direction, and the strike of the ball, computations predict the resulting balls' trajectories after the shot is played. These results—trajectories, visual effects, and menus—are visually output by the projector, making them visible on the snooker table. The system achieves a shot prediction accuracy of 98% when no bouncing occurs.

Keywords computer vision; augmented reality (AR); Kinect; pool game

1 Introduction

The classical physics underpinning a game of pool can be hard to understand by a beginner or unskilled player, typically requiring many hours of practice to completely understand them.

In this paper, a visual application is introduced intended to help and assist amateur players by using the pool table as an interface, showing on-the-fly a prediction of what will happen when the player hits the white ball in its current position; menus or other visual effects can also be projected and accessed over the table or elsewhere. The system works for all varieties of tables and cues, regardless of their size, cloth or cue colour and material, or even the game type. It is based on a Kinect 2 sensor [1] and a projector placed above the table. The Kinect 2 sensor is responsible for capturing the game, which is then processed by a standard computer, enabling detection of game elements such as the table's rails (borders), cue direction, and balls' positions, which are all used to predict a trajectory. The output result is then forwarded in real time to a projector, showing what might be the final result of that shot on the pool table.

The contributions of this paper are firstly, introducing use of a depth sensor to augmented reality (AR) for a pool or billiards game system, and making the detection functions of the system more reliable: novel methods are proposed to detect the motion, the pool balls' centres (even when several balls are in contact), the cue position and direction, and the strike of the ball. This basic information is used to simulate the basic physics of the game based on depth information. Secondly, the system itself is an augmented reality pool application that works in real clubs, pubs, or exhibition environments, without the need for any changes to equipment including balls, cues, table, lighting, etc.

Section 2 presents the state of the art, while Section 3 explains in detail the system's implementation: detection of the table borders, balls, cue and strike, the physics computation, and how output is mapped to the projector. Section 4 presents tests and results in an exhibition environment. The final section presents a discussion, conclusions, and future work.

2 State of the art

Many examples of tools exist for the games of pool, snooker, and billiards. Many of them are focused

1 LARSyS and Institute of Engineering, University of the Algarve, 8005-139 Faro, Portugal. E-mail: L. Sousa, luiscarlosrsousa@outlook.com (✉); R. Alves, ricardo_alves_r16@hotmail.com.

2 LARSyS, CIAC and Institute of Engineering, University of the Algarve, 8005-139 Faro, Portugal. E-mail: jrodrig@ualg.pt.

Manuscript received: 2015-12-24; accepted: 2016-02-25

on analysing video footage mostly to give a 3D representation of the game [2–4]. Unlike those systems, e.g., Ref. [2], the application proposed in this paper is an AR tool that enhances on-the-fly the perception of what the player is currently doing by projecting a calculated trajectory onto the table. Other research uses a robot capable of choosing and executing shots on a real table, although these robotic systems have been tested under laboratory conditions [5–7].

Leckie and Greenspan [8] presented a paper about the physics in a game of pool; also see Refs. [9, 10]. One of these authors also presented a tool similar to the one proposed in this paper: ARPool is a projector–camera system that provides the player real time feedback on the surface of the table [11]. However, no publications appear to be available regarding this tool (only a web page). The present authors introduced in Ref. [12] an initial version of the system (PoolLiveAid), very similar to Refs. [11, 13], using a single Full HD webcam as a sensor to acquire what is occurring on the table. Despite the good results provided by the system, some limitations exist, e.g., it is very difficult to detect and individually distinguish each ball when two or more balls are in contact. Also when using the system in real pool clubs, some shortcomings were observed in ball detection due to the imposed lighting.

Other systems exist: Shih et al. [10] presented a system to compute the best sequence of shots given a starting cue ball position. Later, Shih [9] presented three novel game strategies to investigate the effect of cue shot planning on game performance. The above installations almost all use RGB cameras, but other sensors can be used. For instance, 3D sensors are gaining more attention currently, due to their greater functionality.

The Microsoft Kinect [1] is one of the best known and was popularized by the video gaming industry, but now many applications can be found using it; see, e.g., Refs. [14–16]. By using the above mentioned 3D sensor and the depth information it provides, in comparison to Refs. [11, 12], the tool proposed in this paper increases the robustness of the application with respect to suboptimal lighting conditions, relative to our previous tool [12].

The use of 3D sensors, instead of RGB cameras, doubtless provides benefits in this type of application, such as the aforementioned immunity to changes in lighting conditions, making cue and ball segmentation more robust than if done purely using colour, especially for separating balls that are in contact. By using depth information, as it will be shown, the segmentation of the cue and balls can be very precise. There are no main disadvantages in the use of the Kinect in this particular application, except if an infra-red source causes interference in the field of view of the sensor, stopping it working properly.

The most similar tool to the one presented in this paper is OpenPool [17], an open source system that uses a 3D depth sensor to detect balls' positions, and uses Unity software to compute animations that are mapped onto the table using a projector. This tool can also detect when a ball is successfully potted using auxiliary hardware installed in the table's pockets. For ball detection, it uses the same sensor technology as this paper (a Kinect 2), but the similarity stops there. In our system, the cue is also detected and the basic physics of the strike of the ball are computed. The goal of OpenPool seems (at least for now) to be an animated pool table, using the elements (the balls) that are on the table. Our system also allows animations, but the main difference is in purpose, allowing the inexperienced player to comprehend the basic physics of the game. By moving the cue near the ball, it is possible to see on-the-fly a projection of its expected trajectory.

3 System implementation

As already mentioned in the introduction, the system consists of (a) a pool table, of any size, with the usual balls and cue, (b) a Microsoft Kinect 2 [1], (c) any ordinary laptop or desktop computer capable of analysing inputs from the Kinect, and (d) a projector to project the computed trajectories and balls' locations.

In terms of setup, the projector can be placed above the table (Fig. 1 left, fixed in the celling) or on a side wall, as long as it can project onto the whole table. A single Kinect 2 (Fig. 1 left, the black sensor on the white support) can cope with tables of size up to $2.5\,\mathrm{m} \times 1.4\,\mathrm{m}$ at a height of up to $1.75\,\mathrm{m}$; the dimension used for the experimental setup presented in this paper is $2.2\,\mathrm{m} \times 1.1\,\mathrm{m}$, with the Kinect placed

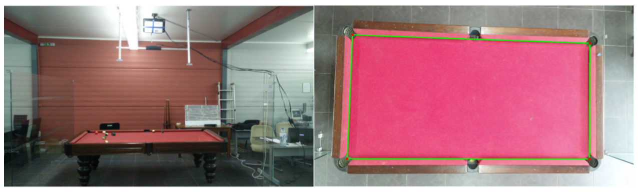
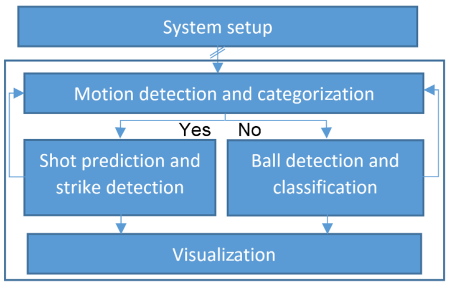


Fig. 1 Left: prototype. Right: a region of the colour frame acquired by the Kinect, with the 4 corners (and rails) marked.

at a height of about 1.6 m. For larger tables, two or more Kinect sensors could be used to acquire the entire game field, requiring an additional (trivial) algorithm to merge the acquired frames. All the algorithms presented in this paper could be used with more than one Kinect sensor (in Fig. 1 left, the ends of the support were used to test with two Kinect 1 sensors). Finally, the Kinect should be placed more or less above the centre of the table: while other positions could be used, if the Kinect were placed for instance on a side wall, this would severely impair the Kinect's depth resolution, hampering detection of game elements, as would the occlusion of some elements.

The system is divided into 5 main modules (Fig. 2). *System setup* computes the table boundaries and all transformation matrices. *Motion detection and categorization* detects motion on the table. If motion and thus game play has stopped (no motion was detected in the current frame at time t, but motion was detected at time $t-1$), then it executes the *ball detection and classification* module. However, if motion was detected in the current and previous frames, then two situations can exist: the balls are in movement on the table, or a sudden movement was detected, assuming that a player is approaching to play. In this case, using the balls already detected in a previous iteration, *shot prediction and strike detection* starts to execute. Again two situations can occur: cue detection is needed, and physics simulated, or after the cue ball has been struck, balls are still moving and need to be detected once they stop. Finally, the *visualization* module ensures all outputs, strike prediction, visual effects involving the balls, and menus, are projected onto the table.


System setup
Motion detection and categorization
Yes
No
Shot prediction and strike detection
Ball detection and classification
Visualization
Fig. 2 Block diagram of the system.

3.1 System setup

As mentioned, neither the Kinect nor the projector needs to be placed in the centre of the table, so two important preprocessing steps must compute: (i) the perspective transform of frames acquired by the sensor, and (ii) the perspective transformation of images that will be displayed by the projector. Plus, (iii) a reference depth frame (R) is determined. This information is saved to file, and loaded every time the system initializes; it only needs to be computed again if the table, the Kinect sensor, or the projector changes position.

The Kinect sensor provides at each time step an RGB frame, $I(x, y)$, and a depth frame, $D(x, y)$; in the latter each pixel (x, y) represents the distance of sensed objects to the sensor. Smaller pixel values correspond to points closer to the sensor. The Kinect depth frame has 16 bit resolution, so pixel values range from 0 to $N_D = 65535$.

For step (i) *perspective transformation*, the borders of the table's playing area are needed. These are computed using the RGB image I; a Canny edge detector [18], then a Hough transformation [18] are applied. As several lines are detected, only (almost) horizontal and vertical lines are selected, and their intersection points are shown to the user (see implementation details in Ref. [12]). The user is asked to validate these corners, adjusting them if necessary, as small errors in corner positions lead to larger trajectory errors.

Let $Ct_{\{1,\ldots,4\}}(x, y)$ be the positions of the corners of the playing area from top-left in clockwise order (see Fig. 1 right). Given the four points determined above and the four reference points of the mapping, i.e., $(0, 0)$, $(M, 0)$, (M, N) and $(0, N)$, with $M = 2 \times N$ and $N = [\mathrm{dist}(Ct_2, Ct_3) + \mathrm{dist}(Ct_1, Ct_4) + (\mathrm{dist}(Ct_1, Ct_2) + \mathrm{dist}(Ct_4, Ct_3))/2]/4$, where dist means Euclidean distance, then a transformation matrix [18] M^{PtK} can be computed. Thus, the initial depth frame D can now be transformed to a depth frame containing only the playing area, $D' = M^{\mathrm{PtK}} D$. We set $M = 2 \times N$ as a professional pool table has a length twice its width.

Computing (ii) the *perspective transformation of the images*, is necessary only for projectors that do not have a built-in function that lets the user choose the four corners of the projection. In such cases, a similar computation to (i) is necessary. The visualization component P, can be transformed to $P'' = M^{\mathrm{PtP}} P$, where M^{PtP} is the perspective transformation for the projector.

Also, during setup we compute (iii) the *table reference depth frame*, R. Depth frames acquired by the Kinect present small inconsistencies and noise, which need to be removed in order to improve detection and reliability. As during setup time, and while no motion is occuring on the table, a small delay of say 1 s in the computation is not important, and an average of the most recent frames is computed to remove noise. The filter used for time t, Da_t, averages the previous N_p depth frames: $Da_t(x,y) = \sum_{k=t-N_p}^{t} D_k(x,y)/N_p$. The result of this process is illustrated in Fig. 3, the 1st row, right (with $N_p = 25$), the middle showing an example of an original depth frame D, extracted from the empty table on the left. Choosing a higher N_p improves reliability but increases the delay. The reference depth average frame is the initial Da (empty table) found at setup, after applying the transformation $Ra' = M^{\mathrm{PtK}} Da$ (see Fig. 3, the 2nd row, left).

In the remaining text, the notations X' and X'' represent frames to which the transformations M^{PtK} and M^{PtP} have been applied, repsectively. For visualization purposes, all pixels from the figures representing the depth frames were divided by 16, clamping any values higher than 255, then they were brightened by 10% and their contrast was increased by 90%. This causes the images in Fig. 3 to show some banding, representing very small changes in depth that have been amplified for visualization purposes. This banding would be concentric with the sensor if it were centred and parallel to the table. In this case, it had a slight pan and tilt, but this does not affect any of our algorithms.

3.2 Motion detection and categorization

The motion detection and categorization module determines the phase of the game and what step of the algorithm should be exectuted next (see Fig. 2). To detect motion the depth frame D is used. For frames that are expected to have motion, or with motion, the noise removal is applied in real time using a Gaussian filter, G [18], with $\sigma = 2$, $Dg_t(x,y) = G(D_t(x,y))$.

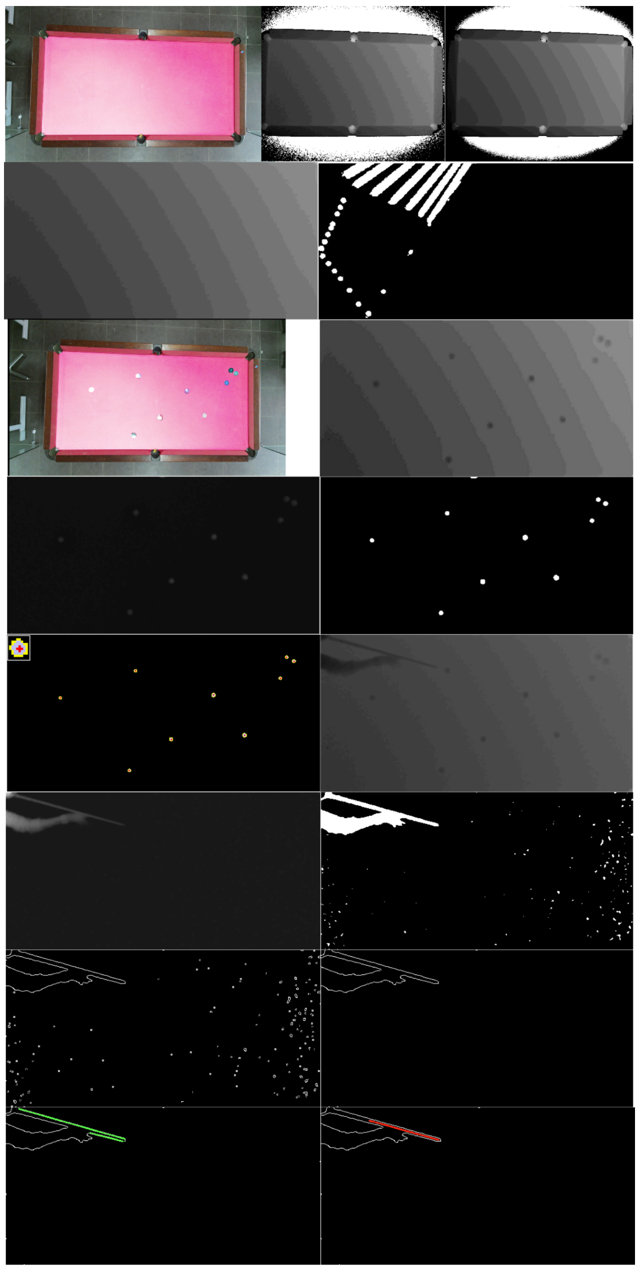


Fig. 3 Top row: left: empty table, middle: depth frame D, right: after noise removal Da. The 2nd row: left: reference frame Ra', right: example of Mb frame. The 3rd row: left: table with balls, right: Da'_t. The 4th row: left: B_t with blobs where possible balls might be, right: binary image Bb_t. The 5th row: left: Bc_t image with contours marked in yellow and the balls' centres in red, right: example of a current frame containing the player's hand and the cue Da'_t. The 6th row: C_t and Cb_t. The 7th row: left: Ce_t, right: a frame with blobs due to noise removed. Bottom row: left: result showing multiple lines found on the cue, right: a single line (in red) resulting from cue detection.

3.2.1 Motion detection

Motion detection can be implemented using the difference between two depth frames, $|Dg'_t - Dg'_{t-1}|$, but using this approach, small differences between two consecutive frames are almost undetectable and can be confused with noise. Thus, a comparison between the current frame and multiple other frames is used: $M_t(x,y) = \sum_{j=0}^{N_p} |Dg'_t(x,y) - Dg'_{t-j}(x,y)|$, where M_t is the motion detection frame at time t, which compares the current frame Dg'_t and the previous $N_p = 40$ frames (around 1.5 s). A binary image is computed indicating where motion exists: $Mb_t(x,y) = 1$, if $M_t(x,y) \geqslant T_m$, otherwise $Mb_t(x,y) = 0$, with $T_m = 0.05\% N_D$. Figure 3, the 2nd row, right, shows an example including a cue and a ball being struck.

3.2.2 Motion categorization

The system uses a binary variable ω to indicate the motion status of the system, 1 meaning *motion* and 0 meaning *no motion*. The system starts by default with $\omega = 1$. The number of white pixels detected in Mb_t gives us information about whether motion is occurring in the current frame: $Cm_t = \sum_{x=0}^{M} \sum_{y=0}^{N} Mb_t(x,y)$. A counter, K_t, manages false movements detected by the procedure above:

(i) If the current value of the system state is *no motion*, then it is necessary to detect when motion starts, and for this the counter K_t is incremented if Cm_t is higher than $K_1 = 0.05\%M$, otherwise decremented (values lower than 0 are clamped to 0). If the counter K_t reaches $K_2 = 25$ (around 1 s; 25 frames), then it is considered that motion has started, changing the state to *motion* and we set $K_t = 0$.

(ii) On the other hand, if the system state is *motion* then the counter is incremented if Cm_t is lower than K_1, otherwise decremented (again, values lower than 0 are clamped to 0). If the counter K_t reaches K_2, then it is considered that motion has stopped, changing the state to *no motion* and we set $K_t = 0$. Using K_t, motion can be characterized as follows:

(ii.1) *Stopped*: If motion was not detected in the current frame, but was detected in the previous frame, $\omega_{t-1} = 1 \wedge K_t = K_2$, then $\omega_t = 0 \wedge K_t = 0$.

(ii.2) *Started*: If motion was detected in the current frame, but was not detected in the previous frame, $\omega_{t-1} = 0 \wedge K_t = K_2$, then $\omega_t = 1 \wedge K_t = 0$.

(ii.3) *Non-existent* (no motion): If motion was neither detected in the current or previous frames, $\omega_{t-1} = 0 \wedge K_t < K_2$, then $\omega_t = 0$.

(ii.4) *In motion*: If motion was detected in either the current or previous frame, $\omega_{t-1} = 1 \wedge K_t < K_2$, then $\omega_t = 1$.

Having characterized the motion, it is now possible to project different visual outputs concerning the motion event taking place, and to detect the balls' positions and the remaining elements of the game.

3.3 Ball detection and classification

Ball detection and classification is triggered after motion has *stopped*, and uses the reference depth average frame Ra' determined in the setup step (see Fig. 3, the 2nd row, left).

3.3.1 Ball detection

To detect the balls' positions on the table we compute $B_t(x,y) = |Ra'(x,y) - Da'_t(x,y)|$, where B_t is the frame containing blobs where balls might be (see Fig. 3, the 4th row, left). A binary threshold is applied to obtain a binary frame containing the balls (see Fig. 3, the 4th row, right), where $Bb_t(x,y) = 1$ if $B_t(x,y) > T_b$, and 0 otherwise; $T_b = 13/16 B_w$, where B_w is the ball height determined during system setup from depth information. It is important to stress that the threshold must be between $T1 = 7/8 B_w$ and $T2 = 3/4 B_w$ (see Fig. 4), so that $T_b \geqslant T2$, otherwise balls that are in contact with each other will forming a single blob, and thus be recognised as just one ball.

The next step consists of applying the morphological erosion operator E, with the goal of removing any remaining noise (small blobs) that still persist: $Be_t = E(Bb_t)$. Finally, we compute the ball centre and radius (r). A ball has a peak in the depth frame at its centre, giving the exact position (x,y) of the ball on the table. To find these peaks, allowing for noise in the Kinect depth frames, a contour finder C [19] is applied, $Bc_t = C(Be_t)$.

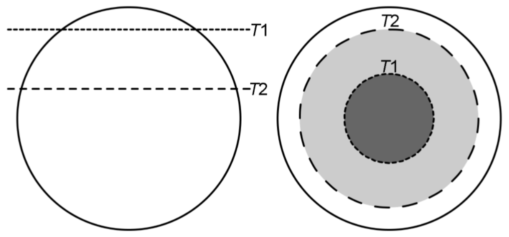


Fig. 4 Side and top view of a ball, with $T1$ and $T2$ reference depths.

For each contour a local maximum is computed in B_t, corresponding to the coordinates of the ball's centre. Figure 3, the 5th row, left, shows in yellow the contour of each ball, and in red its peak (at the top left of the image, for clarity, a zoomed view of one of the blobs/balls is shown).

3.3.2 *White ball classification*

Having detected all balls, the white ball has to be classified, as it is the only ball the cue may hit. The I' colour frame from the Kinect is converted to HSV colour space, I'_{HSV}, and the pixels' V component values inside each ξ_i contour in Bc_t are summed. The contour with the biggest sum is classified as the white ball, with centre (x_{bc}, y_{bc}), with respective contour index i, where $W(x,y,i) = \max_{\{i=0,\ldots,N_b\}}\{\sum_{\xi_i} I'_V(x,y)\}$, and N_b is the number of ball contours in Bc_t.

A similar process could be used to classify other balls.

3.4 Shot prediction and strike detection

With all balls detected and the white ball found, cue detection is the next step, followed by shot prediction and strike detection.

3.4.1 *Cue detection*

Cue detection is based on 5 steps: (i) waiting for all balls to stop moving, (ii) defining the depth reference frame of the table, (iii) computing the difference between the depth reference frame and the current depth frame, (iv) finding the largest blob, if any exists (removing all smaller ones), and (v) detecting the centre line for the largest blob from a starting point near the white ball.

In more detail, when *motion stops* and triggers ball detection, a reference frame Qa' is captured (see Fig. 3, the 3rd row, right). Since the reference frame contains the balls at the instance motion stopped, the difference between it and any current frame (see Fig. 3, the 5th row, right) can only be a player, a cue, or both. Thus, cue detection can be achieved by computing $C_t(x,y) = |Qa'(x,y) - Da'_t(x,y)|$ (see Fig. 3, the 6th row, left), then creating a binary frame, removing small inconsistencies due to noise: $Cb_t(x,y) = 1$, if $C_t(x,y) \geqslant T_c$; 0, otherwise, where $T_c = 0.05\% N_D$ (see Fig. 3, the 6th row, right).

Using a contour finder [19] on Cb_t, the contours of all blobs are found (γ_i), one being the cue (usually with the hand and arm attached), and all others being noise. If the cue exists in Cb_t, then it has a larger area than the other blobs found (see Fig. 3, the 7th row, left). To avoid false cue positives, first we find the blob with the largest area, $A_l = \max(A_i)$, where A_i is the area of each blob γ_i in frame Cb_t. We then compute the average area of the remaining blobs $\overline{A} = (\sum_{i=0}^{N_c} A_i - A_l)/(N_c - 1)$, where N_c is the total number of blobs. The next step consists of removing from Cb_t all blobs A_i whose area is less than $100 \times \overline{A}$. At this point, if a blob still exists in frame $Cc_t(x,y)$ it could be a hand, a cue, or more probably a cue with a hand (see Fig. 3, the 7th row, right).

To detect the cue, and later its direction, a Hough line transform [20] is computed for $Cc_t(x,y)$. Now, it is necessary to select only lines that belong to the cue. All lines that both start and end more than 5 times the ball diameter from the cue ball are discarded, thus removing possible lines detected due to the user's arm. All remaining lines having the same angle, $\pm 5°$ (see Fig. 3, bottom row, left, in green), are then used to determine an average line (see Fig. 3, bottom row, right, red line), defined by (x_c, y_c), a point on the line, and (c_x, c_y), the direction vector of the line.

3.4.2 *Shot prediction*

With the white ball located and the cue detected, it is possible to predict the trajectory of the cue ball after it has been struck. To check if the cue is being aimed at the white ball (see Fig. 5), (i) the equation of the cue line, $\boldsymbol{v}(x,y) = (x_c, y_c) + k_c(c_x, c_y)$ is computed (see Section 3.4.1), as well as (ii) the equation of the ball, $r^2 = (x - x_{bc})^2 + (y - y_{bc})^2$ (see Section 3.3.2), and (iii) we check if they intersect, $(x_c - k \times c_x - x_{bc})^2 + (y_c + k \times c_y - y_{bc})^2 = r^2$.

If k_c is a real number, then the cue is aimed at the white ball, and the physics can now be simulated. Knowing the vector representing the direction of the cue ($\boldsymbol{v}$), it is necessary to compute vectors at each table boundary in order to calculate the respective reflected trajectory (see Fig. 5). Taking into consideration that the centre of the ball, due to the ball's finite radius, does not reach the table boundary, the table boundary vectors are computed as follows, clockwise around the table: $\boldsymbol{b}^i(x,y) = (x_b^i, y_b^i) + k_b^i((y_b^i - y_c), b_y^i)$, with $i = \{1,\ldots,4\}$, $(x_b^1, y_b^1) = (r,r)$, $(x_b^2, y_b^2) = (M-r, r)$, $(x_b^3, y_b^3) = (M-r, N-r)$, and $(x_b^4, y_b^4) = (r, N-r)$. The

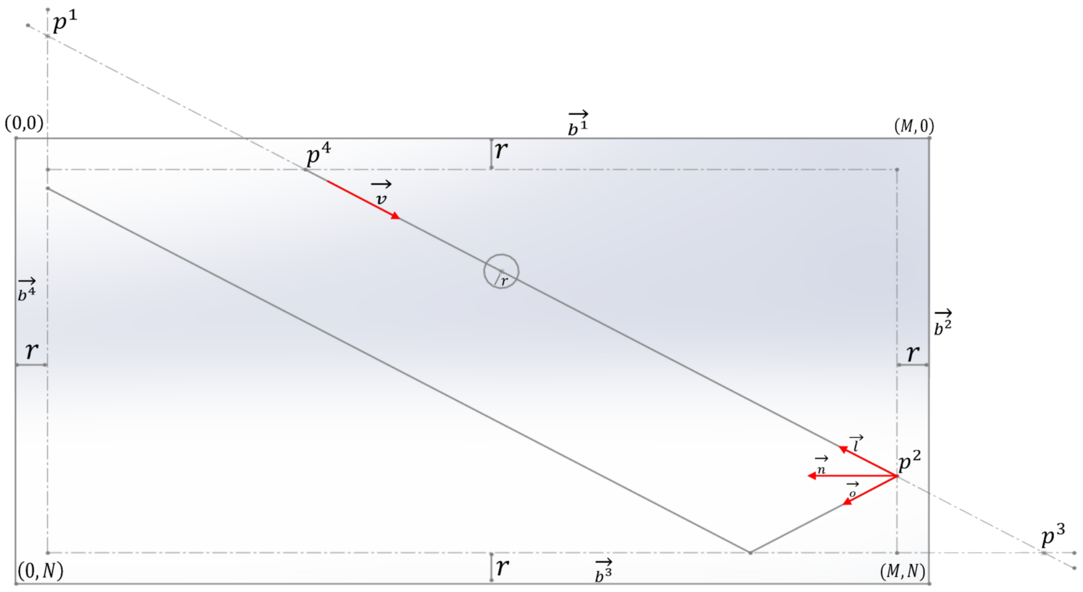


Fig. 5 Shot preview, trajectory computation.

directions are respectively, $(b_x^{\{1,3\}}, b_y^{\{1,3\}}) = (M, 0)$ and $(b_x^{\{2,4\}}, b_y^{\{2,4\}}) = (0, N)$.

Finding if an intersection exists, and getting the contact point, for every boundary i, is then determined using $(x_c, y_c) + k_c^i(c_x, c_y) = (x_b^i, y_b^i) + k_b^i(b_x^i, b_y^i)$, i.e., $k_b^i = (c_x(y_b^i - y_c) - c_y((x_b^i - x_c))/(b_x^i c_x - b_y^i c_y)$. An intersection between the boundary and the cue trajectory only occurs when $(b_x^i c_x - b_y^i c_y) \neq 0$, i.e., $(c_y/c_x) \neq (b_x^i/b_y^i)$.

If an intersection occurs, the possible contact points p^1 to p^4 (with (p_x^i, p_y^i) the coordinates of contact with the table boundary) with the table boundary are calculated using $(p_x^i, p_y^i) = (x_b^i, y_b^i) + k_b^i(y_b^i - y_c)$; see Fig. 5.

Since the line of the cue can intersect more than one boundary, the true physical intersection needs to be found. Out of all boundary intersection points, the nearest point of contact to the ball that also has the same direction as the vector (c_x, c_y) is found. The point where the contact occurs, (x_f, y_f), is the one satisfying $(x_f, y_f) = \min \mid (p_x^i - x_{bc}, p_y^i - y_{bc}) \mid \wedge((c_x \times b_x^i) > 0) \wedge ((c_y \times b_y^i) > 0)$.

The reflection between the boundary and the current direction also needs to be calculated, for prediction of the trajectory afterwards. Using the boundary f selected by the process above, a normal vector to that boundary is calculated, $\boldsymbol{n}(x, y) = \boldsymbol{v}(-b_y^f, b_x^f)$ as well as the vector with the opposite direction to the current trajectory $(\boldsymbol{v})$, $\boldsymbol{l} = (x_c - x_{bc}, y_c - y_{bc})$. The reflection trajectory is then calculated to be $\boldsymbol{o} = 2\boldsymbol{n}(\boldsymbol{l} \cdot \boldsymbol{n}) - \boldsymbol{l}$.

This process can be repeated any number of times and should be repeated just enough times as the number of boundaries the ball would hit in its trajectory. Figure 6 shows examples of 2 boundary collisions in the 1st row and 3 collisions in the bottom row, left.

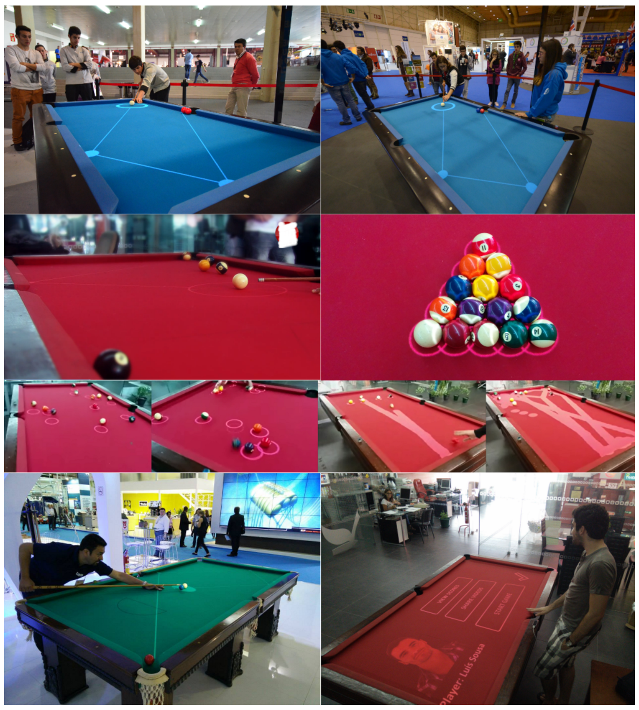


Fig. 6 Examples of the application working at different exhibitions and sites. Bottom right: menu example.

3.4.3 Strike detection

With a predicted trajectory computed, strike detection is necessary to detect when the white ball starts moving, in order to stop calculating (and projecting onto the table) trajectories. The detection of this movement uses the reference image Qa' (see Section 3.4.1), extracting a region of interest (RoI) τ, centred at the current position of the white ball, with size $d \times d$, where $d = 2r$.

For every new depth frame Da'_t, a new RoI (υ) centred on the white ball and with size τ is obtained. Subtraction is then perfromed, $S(x,y) = |\tau(x,y) - \upsilon(x,y)|$, and a threshold is applied to remove any noise, giving $Sb(x,y)=1$, if $S(x,y) \geqslant T_s$; 0, otherwise, where $T_s = 0.01\% N_D$. Summing all pixels in the image Sb determines if the ball has left its place: $C_S = \sum_{x=0}^{d}\sum_{y=0}^{d} Sb(x,y)$, thus if $C_S > \pi r^2/3$, then it is considered that the white ball has been struck.

3.5 Visualization

Having found all game elements and categorised the motion (see Section 3.2.2), it is now possible to project visual information onto the table. An image P is dynamically created with several options depending on the game stage (see Fig. 6): (i) circles centred on the balls, (ii) predicted trajectories, (iii) animations, and (iv) menus. If necessary (see Section 3.1), the projector transformation is applied to P, returning the projected table image P''. Details of the menus and effects, and the corresponding interface are outside the scope of this paper.

4 Tests and results

Figure 6 shows some examples of the system working in real crowded environments, at 3 different exhibitions, each lasting 1 week. All tests were performed and statistics gathered during these weeks with real users, all of whom were first-time application users and unknown to the development team. Two hours (evenly distributed between the exhibitions) of recordings were made, in random 10 minute slots at various time of day, on 4 different days, for a total of 163 shots. In each case, the ground truth was manually created. Table 1 summarises the results. Video extracts from the exhibitions can be seen on Ref. [21] (2014 onwards postings).

The tests were divided into 8 categories. For *table boundary detection*, two different tables were used, both different from the one used during development (a red one), also having different lighting conditions. Table boundaries were always automatically detected (100%) with less than 3-pixel errors. It is important to stress that more tests were conducted during development, the final ones always result in 100% success rate with less than 3-pixel errors. The algorithm was tested 3 times, once at each exhibition.

Motion detection: in the 163 shots made, motion detection worked 100% of the time, correctly triggering the ball detection algorithm. The maximum delay obtained between the motion actually stopping and the ball detection was around 2–3 s, for $N_p = 40$ (most users did not notice this delay). A further set of tests with $N_p = 25$ was also made (with 1 s delay), but in this case the motion detection success rate dropped to 90%.

For *ball detection*, in the 163 shots made, there were a total of 605 balls to be detected (not all balls were placed on the table in every shot). All balls were successfully detected but there were 17 false positives, due to failure to filter noise correctly. The *white ball* was successfully detected 91% of the time. 19 times of failure were due to a striped ball occuring with its white part facing upwards in an area with higher luminosity, while the others (16 times) were due to the white ball being potted. Balls were detected with a maximum of 10 mm error of their true positions, due to noise in the depth frames and distortion of the projector, with the error increasing for balls further from the centre of the table.

Cue detection always worked in the above 163 shots, and so did *stroke detection*. Nevertheless, some errors were noticed in the cue detection outside these tests, when a player did not behave in accordance with pool rules or other expectations, e.g., putting two cues in the pool table area or there several people were present with their hands moving near the table border. During stroke detection, a small delay of around 1 s could sometimes be noticed, mostly due to the cue occupying the position the white ball previously occupied.

Finally, *shot prediction* was the most difficult test to quantify. Since the goal of the application is to assist inexperienced players, and since they may not know how to hold the cue and take a shot, hitting

Table 1 Test results

Algorithm	Sub-test	Number of repetitions	Succeeded repetitions	Success rate (%)
Table boundary detection	NA	3	3	100.0*
Motion detection	NA	163	163	100.0*
Ball detection	NA	605	588	97.2
White ball classification	NA	389	354	91.0
Cue detection	NA	163	163	100.0*
Strike detection	NA	163	163	100.0*
	No bouncing	82	80	97.6
Shot prediction	1 bounce	49	39	79.6
	2 bounces	32	17	53.1

*Although achieving 100% success rate, these tests suffered imprecisions, as explained in the text.

the white ball on the side and giving it spin was not counted in these statistics. Ball motions that did not bounce were successfully predicted in 98% of cases, while balls that bounced once were successfully predicted 80% of the time and balls that bounced twice were successfully predicted 53% of the time. More bounces were not included in this test, due to their poor results (a bounce being each contact the white ball makes with a table boundary).

5 Conclusions

We have presented an application that aids a beginner to play pool, based on a Microsoft Kinect 2 sensor, allowing the detection of table boundaries, the balls, and the cue with high accuracy. All the algorithms have been demonstrated to be very robust against changes in lighting conditions and noise. A projector, placed above the table, shows in real time the computed trajectory in order to give a player a perception of what will happen on that particular turn.

The system works in real time, and all testing was done in real environments, showing very good results. A comparison with previous systems is difficult, as to the best of our knowledge there is no suitable test data or ranking method. However, our system works in real time and in real conditions, whereas systems like Refs. [2, 4, 22] work with video (or video streams) taken from pool or snooker championships, with very stable and controlled conditions (lighting, player positions, etc.), or expect a controlled environment because of the use of robots [5–7].

The most similar works are Refs. [9–13]. There is no technical publication available to make any type of comparison to Ref. [11]. In comparison to Ref. [12] (our previous work), the system has improved in terms of reliability by about 10%–20%, depending on the test considered, while improving reliability by 1%–5% over using two Kinect 1 sensors (in unpublished work). Considering Ref. [12] in more depth, in terms of lighting, the Kinect enables precise detection of game elements even when lighting conditions change drastically, which was detrimental to the results when using a webcam. Secondly, since the colour segmentation used in Ref. [12] is now replaced by balls and cue detection, balls or cues that have very similar colour to the cloth on the table are now more easily detected, leading to a 30% improvement. Finally, Shih et al. [9, 10] presented very interesting work in terms of physics of the game (better simulation than that presented here), but they used a very small table under controlled conditions. They used an RGB camera to extract the balls and cue, which when applied in real situations, e.g., under different lighting in pubs and exhibitions, with different table cloths, etc., is unlikely to be as reliable as our present approach (see our previous work [12], and the discussion above). In terms of augmented reality, they only showed their output on a computer screen.

In summary, the most important contribution of our paper is the complete system, that by using the Kinect sensor, has turned out to be very reliable and can work in any environment using any table cloth, balls, or cues. In the near future, work will focus on increasing the number of menu options, improving the augmented reality menu, and implementing an automatic scoring system allowing us to collect more statistics. An important focus will be the prediction of the movement of the coloured balls after they are hit by the white ball. Also, the physics can be improved if the stroke force is estimated, as well as determining the exact position at which the cue hits the white ball. This last point will be for sure a very challenging goal.

Finally, after this, tests with established professional players should be performed in order to validate the implemented physics, by tracking the struck ball and comparing with the previous system prediction. Improving the physics and validating it with the aid of professional players will enable us to implement a set of tests to show whether the application can also teach beginners how to play or improve their skills.

Acknowledgements

This work was partly supported by the Portuguese Foundation for Science and Technology (FCT), project LARSyS UID/EEA/50009/2013, and the INALUX company (http://www.inalux.com/). We also thank the anonymous reviewers for their very significant and useful contributions to the paper.

References

[1] Kinect2. Kinect for Windows. 2015. Available at http://www.microsoft.com/en-us/kinectforwindows/.

[2] Höferlin, M.; Grundy, E.; Borgo, R.; Weiskopf, D.; Chen, M.; Griffiths, I. W.; Griffiths, W. Video visualization for snooker skill training. *Computer Graphics Forum* Vol. 29, No. 3, 1053–1062, 2010.

[3] Jiang, R.; Parry, M. L.; Legg, P. A.; Chung, D. H. S.; Griffiths, I. W. Automated 3-D animation from snooker videos with information-theoretical optimization. *IEEE Transactions on Computational Intelligence and AI in Games* Vol. 5, No. 4, 337–345, 2013.

[4] Ling, Y.; Li, S.; Xu, P.; Zhou, B. The detection of multi-objective billiards in snooker game video. In: Proceedings of the 3rd International Conference on Intelligent Control and Information Processing, 594–596, 2012.

[5] Archibald, C.; Altman, A.; Greenspan, M.; Shoham, Y. Computational pool: A new challenge for game theory pragmatics. *AI Magazine* Vol. 31, No. 4, 33–41, 2010.

[6] Landry, J.-F.; Dussault, J.-P.; Mahey, P. Billiards: An optimization challenge. In: Proceedings of the 4th International C* Conference on Computer Science and Software Engineering, 129–132, 2011.

[7] Nierhoff, T.; Kourakos, O.; Hirche, S. Playing pool with a dual-armed robot. In: Proceedings of IEEE International Conference on Robotics and Automation, 3445–3446, 2011.

[8] Leckie, W.; Greenspan, M. An event-based pool physics simulator. In: *Lecture Notes in Computer Science, Vol. 4250*. Van den Herik, H. J.; Hsu, S.-C.; Hsu, T.-S.; Donkers, H. H. L. M. Eds. Springer Berlin Heidelberg, 247–262, 2006.

[9] Shih, C. Analyzing and comparing shot planning strategies and their effects on the performance of an augment reality based billiard training system. *International Journal of Information Technology & Decision Making* Vol. 13, No. 3, 521–565, 2014.

[10] Shih, C.; Koong, C.-S.; Hsiung, P.-A. Billiard combat modeling and simulation based on optimal cue placement control and strategic planning. *Journal of Intelligent & Robotic Systems* Vol. 67, No. 1, 25–41, 2012.

[11] ARPool. Augmented reality: Pool. 2015. Available at http://rcvlab.ece.queensu.ca/qridb/ARPOOL.html.

[12] Alves, R.; Sousa, L.; Rodrigues, J. M. F. PoolLiveAid: Augmented reality pool table to assist inexperienced players. In: Proceedings of the 21st International Conference on Computer Graphics, Visualization and Computer Vision, 184–193, 2013.

[13] Larsen, L. B.; Jensen, R. B.; Jensen, K. L.; Larsen, S. Development of an automatic pool trainer. In: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology, 83–87, 2005.

[14] Ahmed, F.; Paul, P. P.; Gavrilova, M. L. DTW-based kernel and rank-level fusion for 3D gait recognition using Kinect. *The Visual Computer* Vol. 31, No. 6, 915–924, 2015.

[15] Song, X.; Zhong, F.; Wang, Y.; Qin, X. Estimation of Kinect depth confidence through self-training. *The Visual Computer* Vol. 30, No. 6, 855–865, 2014.

[16] Abedan Kondori, F.; Yousefi, S.; Liu, L.; Li, H. Head operated electric wheelchair. In: Proceedings of IEEE Southwest Symposium on Image Analysis and Interpretation, 53–56, 2014.

[17] OpenPool. OpenPool. 2015. Available at http://www.openpool.cc/.

[18] Russ, J. C. *The Image Processing Handbook*, 6th edn. CRC press, 2011.

[19] Suzuki, S.; KeiichiA be. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing* Vol. 30, No. 1, 32–46, 1985.

[20] Duda, R. O.; Hart, P. E. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM* Vol. 15, No. 1, 11–15, 1972.

[21] PoolLiveAid. PoolLiveAid Facebook. 2015. Available at https://www.facebook.com/Poolliveaid.

[22] Legg, P. A.; Parry, M. L.; Chung, D. H. S.; Jiang, R. M.; Morris, A.; Griffiths, I. W.; Marshall, D.; Chen, M. Intelligent filtering by semantic importance for single-view 3D reconstruction from Snooker video. In: Proceedings of the 18th IEEE International Conference on Image Processing, 2385–2388, 2011.

L. Sousa is a researcher at the University of the Algarve with a master degree in electrical and electronic engineering. He is a member of the LARSyS (ISR-Lisbon) laboratory and he is the co-author of 14 scientific publications. His major interests lie in electronic systems, embedded systems, and computer vision.

R. Alves has a master degree in electric and electronic engineering. He is a researcher at the University of the Algarve working with depth sensors. He is a member of the LARSyS (ISR-Lisbon) laboratory and he is the co-author of 9 scientific publications. He also spends some of his time developing other electronics and programming projects.

J. M. F. Rodrigues graduated in electrical engineering in 1993, got his M.Sc. degree in computer systems engineering in 1998, and achieved a Ph.D. degree in electronics and computer engineering in 2008 from the University of the Algarve, Portugal. He is an adjunct professor at Instituto Superior de Engenharia, also in the University of the Algarve, where he has lectured computer science and computer vision since 1994. He is a member of the LARSyS (ISR-Lisbon) laboratory, CIAC and the associations APRP, IAPR and ARTECH. He has participated in 14 financed scientific projects, and he is the co-author of more than 120 scientific publications. His major research interests lie in computer and human vision, assistive technologies, and human–computer interaction.