# APCS project, A.Y. 2019-2020 Project proposal #50:"Development of an interactive 2D application with OpenFrameworks"

Antonio Pipita

September 2, 2020

# Contents

## 1 Introduction

This project consists in developing an application that enables the user to interact with a 2D environment via video feed.
In particular, the user will control via hand motion a component of a simple game: the goal of the player is to destroy a brick wall by making a ball bounce into it, while avoiding to make the ball fall off the screen. In order to do so, the player will move a bar at the bottom of the gameboard via motion control.
The video feed will be captured via a Microsoft Kinect, mod. 1414

## 2 Application architectural overview

This section introduces the application outlining its structure, the information flow and information regarding the development.

### 2.1 Design pattern

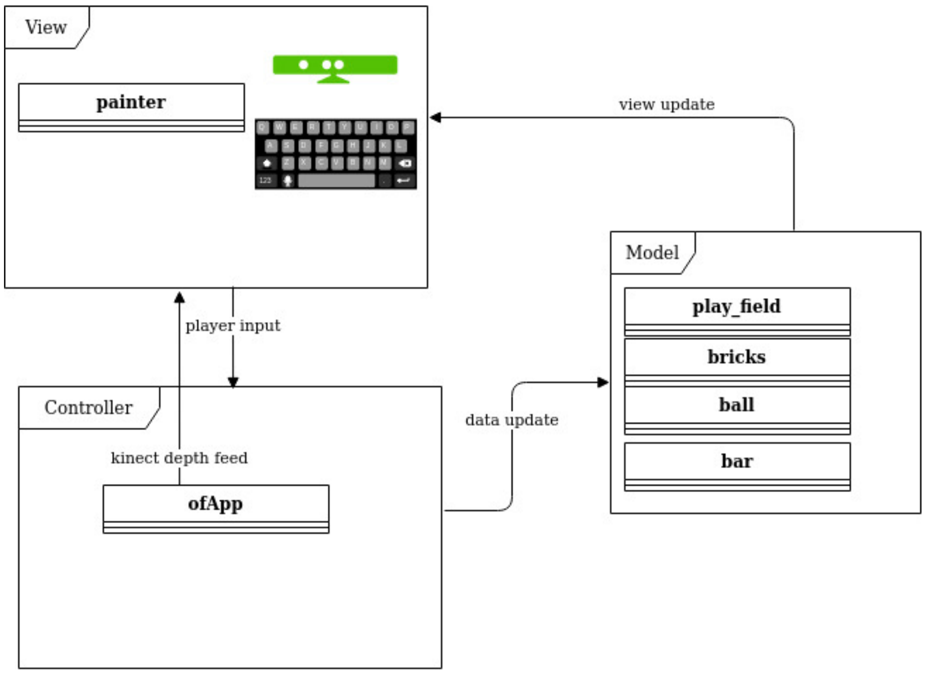


Figure 1: MVC pattern

This application has been designed following the Model-Controller-View pattern:

Model : contains the application data and manages its update. In this application it comprehends the classes play_field, ball, bar and brick.
It manages the movement of the ball on the gameboard, its interactions with the bricks, the bar and the gameboard edges. It also receives the updates on the bar position and angle, if the required action is possible.

Controller : manages the flow of information from the view to the controller. In this application it comprehends the class ofApp, which grabs input from the Kinect device, re-elaborates it and pushes it to the model.

View : manages the interaction with the user. In this application it comprehends the painter class, which represents on the device screen the status of the model, partially the ofApp class, since it directly draws the Kinect's depth view on the screen, and the hardware used by the user for the inputs, which comprises the Kinect itself and the keyboard.

The splitting of the view between the painter class and the ofApp class was done in order to keep the Kinect interaction just in the ofApp class, thus keeping the code a bit more tidy.

## 2.2 Development approach

An hybrid between the bottom-up and top-down approaches has been chosen for the development: once the application parts were designed following the bottom-up approach the application was built following the top-down approach. Methods were designed following the divide et impera paradigm.

## 2.3 Testing

Testing has been performed using VSCode and Intellisense debugging, focusing on limit cases.

## 2.4 Development environment

The project has been developed in an Ubuntu environment (Bionic Beaver), using VScode with C/C++ extensions and Intellisense to write, debug and test the code.

## 2.5 Information flow

The information that the user wants to transmit enters the application via the view (Kinect and keyboard); it is then interpreted and reorganized by the controller, then it is pushed to the model. The model applies the required changes to the application data, updates itself following the rules of ball movement and then sends the new info the view again, which prints on the screen the current state of the game and gives the user new information.

## 2.6 Phases

The application works in three different phases:

Setup phase : this is the phase in which the application starts. During this phase the user can regulate the Kinect angle and move on to the game phase.

Game phase : in this phase the user plays the actual game. The phase ends when the player finishes all of its lives or destroys all the bricks. Once this phase ends the application moves on to the endgame phase.

Endgame phase : in this phase the user can either exit the application or play again, thus returning to the setup phase.

## 2.7 Acronyms and conventions

OF : openframeworks

gameboard : refers to the space in which the ball moves, the elements on it and the rules relative to the interaction between objects.

## 3 Classes overview

This section contains the class diagram and a description of the attributes and methods of each class.

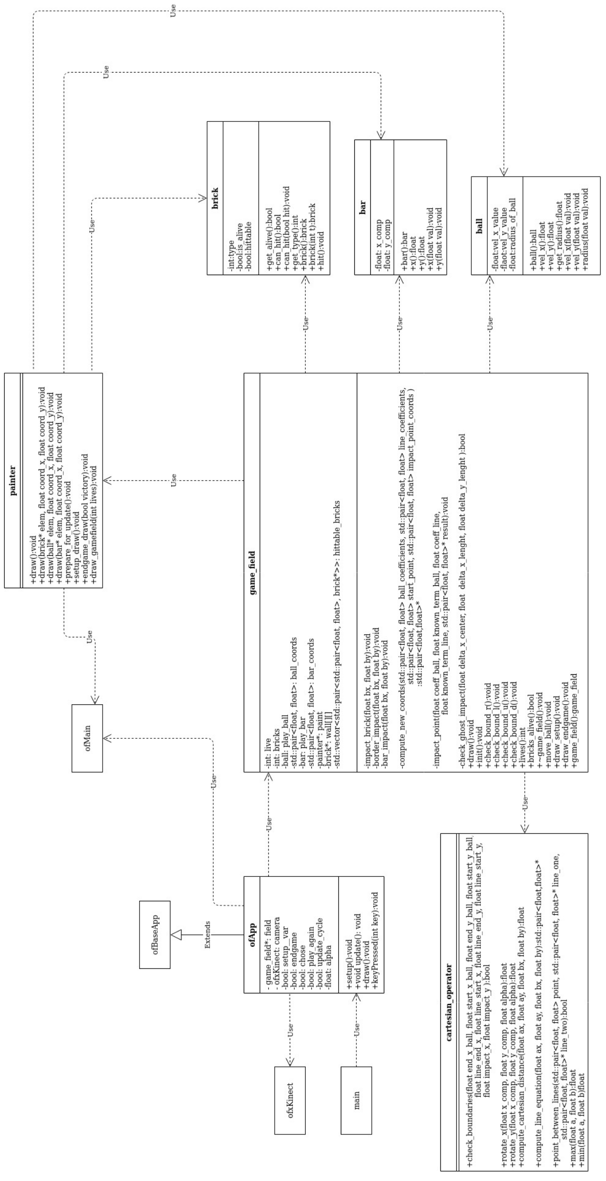

painter
+draw():void
+draw(brick* elem, float coord_x, float coord_y):void
+draw(ball* elem, float coord_x, float coord_y):void
+draw(bar* elem, float coord_x, float coord_y):void
+prepare_for_update():void
+setup_draw():void
+endgame_draw(bool victory):void
+draw_gamefield(int lives):void
ofMain
ofBaseApp
Extends
brick
-int:type
-bool:is_alive
-bool:hittable
+get_alive():bool
+can_hit():bool
+can_hit(bool hit):void
+get_type():int
+brick():brick
+brick(int t):brick
+hit():void
bar
-float: x_comp
-float: y_comp
+bar():bar
+x():float
+y():float
+x(float val):void
+y(float val):void
ball
-float:vel_x_value
-flaot:vel_y_value
-float:radius_of_ball
+ball():ball
+vel_x():float
+vel_y():float
+get_radius():float
+vel_x(float val):void
+vel_y(float val):void
+radius(float val):void
ofApp
- game_field*: field
- ofxKinect: camera
-bool: setup__var
-bool: endgame
-bool: chose
-bool: play_again
-bool: update_cycle
-float: alpha
+setup():void
+void update(): void
+draw():void
+keyPressed(int key):void
ofxKinect
main
game_field
-int: live
-int: bricks
-ball: play_ball
-std::pair<float, float>: ball_coords
-bar: play_bar
-std::pair<float, float>: bar_coords
-painter*: paint
-brick*: wall[][]
-std::vector<std::pair<std::pair<float, float>, brick*>>: hittable_bricks
-impact_brick(float bx, float by):void
-border_impact(float bx, float by):void
-bar_impact(float bx, float by):void
-compute_new_coords(std::pair<float, float> ball_coefficients, std::pair<float, float> line_coefficients, std::pair<float, float> start_point, std::pair<float, float> impact_point_coords ) :std::pair<float,float>*
-impact_point(float coeff_ball, float known_term_ball, float coeff_line, float known_term_line, std::pair<float, float>* result):void
-check_ghost_impact(float delta_x_center, float delta_x_lenght, float delta_y_lenght ):bool
+draw():void
+init():void
+check_bound_r():void
+check_bound_l():void
+check_bound_u():void
+check_bound_d():void
+lives():int
+bricks_alive():bool
+~game_field():void
+move_ball():void
+draw_setup():void
+draw_endgame():void
+game_field():game_field
cartesian_operator
+check_boundaries(float end_x_ball, float start_x_ball, float end_y_ball, float start_y_ball, float line_end_x, float line_start_x, float line_end_y, float line_start_y, float impact_x, float impact_y ):bool
+rotate_x(float x_comp, float y_comp, float alpha):float
+rotate_y(float x_comp, float y_comp, float alpha):float
+compute_cartesian_distance(float ax, float ay, float bx, float by):float
+compute_line_equation(float ax, float ay, float bx, float by):std::pair<float,float>*
+point_between_lines(std::pair<float, float> point, std::pair<float, float>* line_one, std::pair<float, float>* line_two):bool
+max(float a, float b):float
+min(float a, float b)float
Use

Figure 2: The class diagram of the application

The main.cpp file will not have a dedicated section, since its only uses are to create a new instance of ofApp, execute an action aimed at setting up the application window and starting the actual application.

## 3.1 ofApp

The ofApp class acts as controller, extends ofBaseApp and is the class upon which the framework provided by OF acts, thus controlling the applicaiton execution.

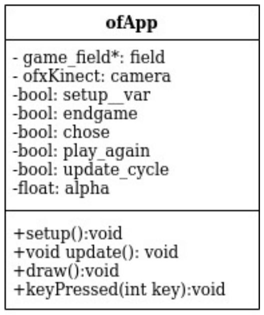


Figure 3: The ofApp class

### 3.1.1 Attributes

All of the following attributes are private

- game_field* playfield
  Pointer to the game_field instance that is currently active, enables the interaction between controller and model.
- ofxKinect camera
  The instance of ofxKinect we refer to in order to interact with the Kinect.
- bool setup__var
  Boolean variable used to represent whether the application is in the setup state.
- bool endgame
  Boolean variable used to represent whether the application is in the endgame state.
- bool chose
  Boolean variable used to represent whether the player has chosen to play again or hasn't taken that decision yet.

- bool play_again
  Boolean variable used to represent the preference of the player regarding playing again or exiting the application.
- bool update_cycle
  Switch used to choose whether to update the bar state or the ball state.
- float alpha
  Kinect angle.

### 3.1.2 Methods

All of the following methods are public. Note that only the overloaded methods have been reported.

- setup():void
  Prepares the application for the execution, initializing the playfield pointer and the gameboard, applying settings for the view and establishing a connection with the Kinect.
- update():void
  Main cycle of the application, its behaviour changes according to the game state:
  - Setup phase: waits for player input to start the game.
  - Game phase: asks the model to update the bar state according to the player's input and to update the gameboard state according to the rules of the gameboard.
  - Endgame phase: waits to the player's deccision to either exit the game or to play another round; then enforces that decision.

  It is worth noting that this method also regulates the game speed: if the updates were too quick that would result in the game speed being too high, thus a check is performed to make sure they take at least 20ms.
  If they take less, the thread will be paused for the difference between the update execution time and 20 ms.
- draw():void
  Method used to update the graphic representation of the game state, mostly by communicating to the model that it's time to do so. Its behaviour changes according to the game state:
  - Setup phase: tells the model to print the setup information and provides the Kinect's depth view feed.
  - Game phase: tells the model to update the visual representation of the gameboard state.
  - Endgame phase: tells the model to print the endgame information.

- keyPressed(int key):void
  Manages the player's input via keyboard, changes its behaviour according to the game phase
  - Up key: in the setup phase is used to increment the Kinect's angle, in the endgame phase to choose to play again.
  - Down key: in the setup phase is used to decrement the Kinect's angle, in the endgame phase to exit the application.
  - Right key: used only in the setup phase to start playing.

## 3.2 painter

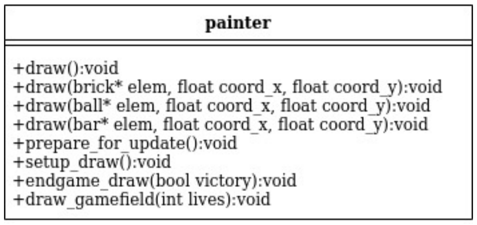

painter
+draw():void
+draw(brick* elem, float coord_x, float coord_y):void
+draw(ball* elem, float coord_x, float coord_y):void
+draw(bar* elem, float coord_x, float coord_y):void
+prepare_for_update():void
+setup_draw():void
+endgame_draw(bool victory):void
+draw_gamefield(int lives):void

Figure 4: The painter class

This class is used to print to the screen the gameboard state representation and the information the user needs. It's composed by public methods:

- draw():void
  General draw method.
- draw(brick* elem, float coord_x, float coord_y):void
  Draws a brick.
- draw(ball* elem, float coord_x, float coord_y):void
  Draws the ball.
- draw(bar* elem, float coord_x, float coord_y):void
  Draws the bar.
- prepare_for_update():void
  Clears the screen.
- setup_draw():void
  Draws the setup information.
- endgame_draw(bool victory)
  Draws the endgame information.
- draw_gamefield(int lives):void
  Draws the gamefield boundaries, the number of lives and other information.

It is worth noting that, while this class does not have any attribute, it presents hoever some global variables (situated in the .cpp file) used for the visual representation coordinates computation.

## 3.3 brick

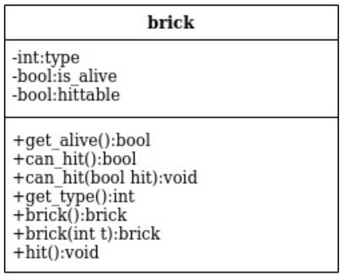


Figure 5: The brick class

This class represents the bricks

### 3.3.1 Attributes

All of the class attributes are private

- int:type
  Represents the remaining hitpoints of the brick.
- bool:is_alive
  Represents whether the brick has any hitpoints left.
- bool:hittable
  Represents whether the brick is currently reachable by the ball.

### 3.3.2 Methods

All of the class methods are public

- get_alive():bool
  is_alive getter.
- can_hit():bool
  hittable getter.
- get_type():int
  type getter.
- can_hit(bool hit):void
  hittable setter.
- hit():void
  Method called when the brick gets hit.
- brick():brick
  Default constructor.

- brick(int t):brick
  Constructor that sets the type to t.

## 3.4 ball

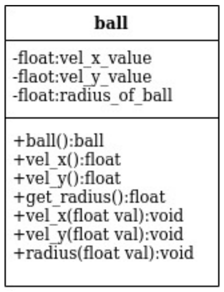


Figure 6: The ball class

This class represents the moving ball used to destroy the bricks

### 3.4.1 Attributes

All of the class attributes are private

- float:vel_x_value
  Represents the x speed component.
- float:vel_y_value
  Represents the y speed component.
- float:radius_of_ball
  Represents the radius of the ball.

### 3.4.2 Methods

- ball():ball
  Constructor.
- vel_x():float
  vel_x_value getter.
- vel_y():float
  vel_y_value getter.
- get_radius():float
  radius_of_ball getter.
- vel_x(float val):void
  vel_x_value setter.

- vel_y(float val):void
  vel_y_value setter.
- radius(float val):void
  radius_of_ball setter.

## 3.5 bar

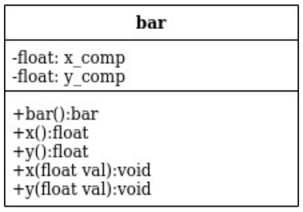


Figure 7: The bar class

This class represents the moving bar controlled by the player

### 3.5.1 Attributes

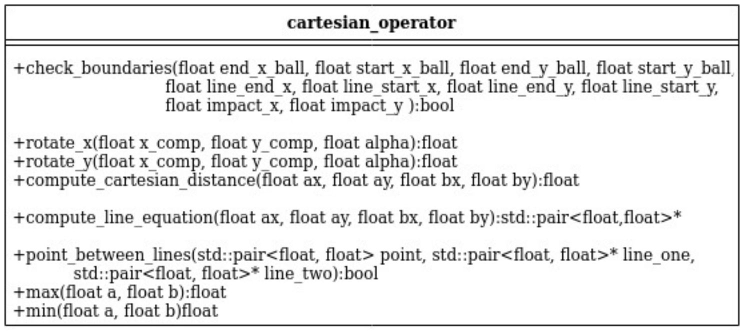
All of the class attributes are private

- float: x_comp
  Horizontal component of the distance between the bar center and the right extremity.
- float: y_comp
  Vertical component of the distance between the bar center and the right extremity.

### 3.5.2 Methods

- bar():bar
  Constructor.
- x():float
  x_comp getter.
- y():float
  y_comp getter.
- x(float val):void
  x_comp setter.
- y(float val):void
  y_comp setter.

## 3.6 cartesian_operator



```
cartesian_operator

+check_boundaries(float end_x_ball, float start_x_ball, float end_y_ball, float start_y_ball,
                  float line_end_x, float line_start_x, float line_end_y, float line_start_y,
                  float impact_x, float impact_y ):bool

+rotate_x(float x_comp, float y_comp, float alpha):float
+rotate_y(float x_comp, float y_comp, float alpha):float
+compute_cartesian_distance(float ax, float ay, float bx, float by):float

+compute_line_equation(float ax, float ay, float bx, float by):std::pair<float,float>*

+point_between_lines(std::pair<float, float> point, std::pair<float, float>* line_one,
        std::pair<float, float>* line_two):bool
+max(float a, float b):float
+min(float a, float b)float
```

Figure 8: The cartesian_operator class

This class is used to perform cartesian computations that don't involve directly data of the objects of the game (bar, ball, bricks)

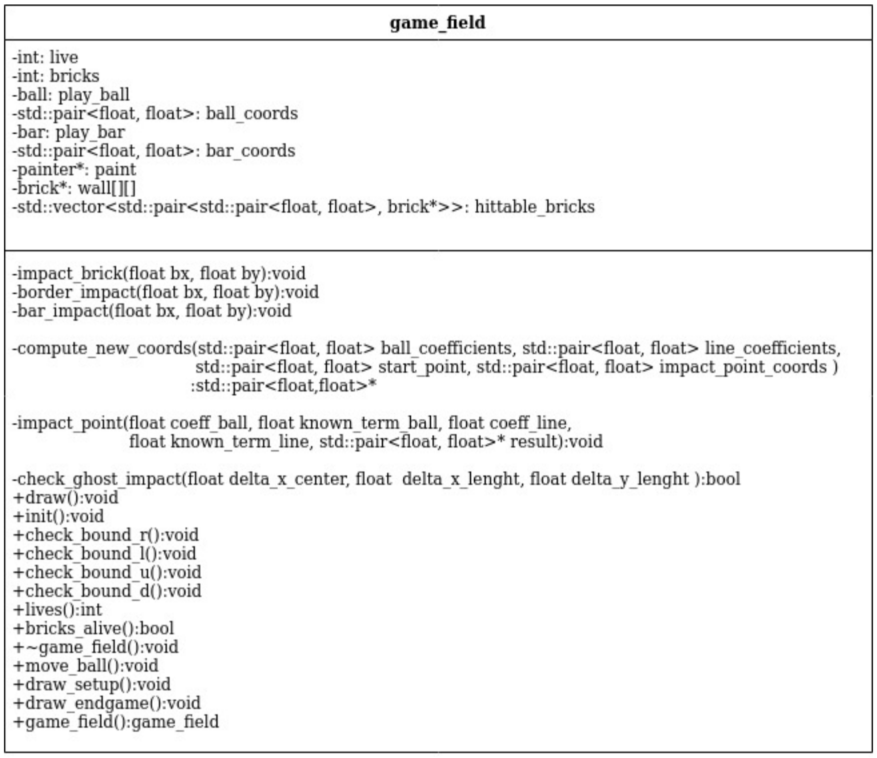
### 3.6.1 Methods

All methods are public

- rotate_x(float x_comp, float y_comp, float alpha):float
  This method executes a rotation of alpha degrees on the x coordinates.
- rotate_y(float x_comp, float y_comp, float alpha):float
  This method executes a rotation of alpha degrees on the y coordinates.
- compute_cartesian_distance(float ax, float ay, float bx, float by):float
  This method computes the cartesian distance betweeen two points.
- check_boundaries(float end_x_ball, float start_x_ball, float end_y_ball, float start_y_ball, float line_end_x, float line_start_x, float line_end_y, float line_start_y, float impact_x, float impact_y ):bool
  Given the points defining 2 segments, the lines on which they lay and the impact point between the two lines this method computes whether the impact point is within the boundaries of the segments.
- compute_line_equation(float ax, float ay, float bx, float by):std::pair<float,float>*
  Given two points this method will compute the coefficient and the known term of the line defined by those two points. Note that if the line is parallel to the y axis, the result returned will be (0,0). This doesn't lead to errors because of how the system has been built.
- bool point_between_lines(std::pair<float, float> point, std::pair<float, float>* line_one, std::pair<float, float>* line_two)
  This method, given a point and two lines, returns true if the point is situated between the two lines.

- float max(float a, float b)
  Given two numbers returns the maximum
- float min(float a, float b)
  Given two numbers returns the minimum

## 3.7 game_field



```
game_field
-int: live
-int: bricks
-ball: play_ball
-std::pair<float, float>: ball_coords
-bar: play_bar
-std::pair<float, float>: bar_coords
-painter*: paint
-brick*: wall[][]
-std::vector<std::pair<std::pair<float, float>, brick*>>: hittable_bricks

-impact_brick(float bx, float by):void
-border_impact(float bx, float by):void
-bar_impact(float bx, float by):void

-compute_new_coords(std::pair<float, float> ball_coefficients, std::pair<float, float> line_coefficients,
                     std::pair<float, float> start_point, std::pair<float, float> impact_point_coords )
                    :std::pair<float,float>*

-impact_point(float coeff_ball, float known_term_ball, float coeff_line,
              float known_term_line, std::pair<float, float>* result):void

-check_ghost_impact(float delta_x_center, float  delta_x_lenght, float delta_y_lenght ):bool
+draw():void
+init():void
+check_bound_r():void
+check_bound_l():void
+check_bound_u():void
+check_bound_d():void
+lives():int
+bricks_alive():bool
+~game_field():void
+move_ball():void
+draw_setup():void
+draw_endgame():void
+game_field():game_field
```

Figure 9: The game_field class

This class contains all the game objects and the rules for their updates and interaction, thus representing the gameboard.

### 3.7.1 Attributes

All of the class attributes are private

- int: live
  Represents the lives the player currently has.
- int: bricks
  Number of bricks still alive.
- ball: play_ball
  Ball class instance.
- std::pair<float, float>: ball_coords
  Coordinates of the ball.

- bar: play_bar
  Bar class instance.

- std::pair<float, float>: bar_coords
  Coordinates of the center of the bar.

- painter*: paint
  Pointer to a painter class instance, grants the possibility to represent data via the view.

- brick*: wall[ ][ ]
  Bidimensional matrix of references to the bricks on the gameboard.

- std::vector<std::pair<std::pair<float, float>, brick*>>: hittable_bricks
  A vector of pairs whose first element are the indexes of the brick in the wall matrix and the second element is a pointer to said brick. This structure is used to represent the bricks that can be reached by the ball.

### 3.7.2 Methods

**Private methods**

- impact_brick(float bx, float by):void
  This function is called to check whether the ball will hit any brick with its next movement. If it is so, ball coordinates and speed components will be updated accordingly, the brick will be hit and, if destroyed, the data structures will be updated accordingly.

- border_impact(float bx, float by):void
  This method checks if, with the next movement, the ball will impact the sides or the ceiling of the playflield. If it is so, ball coordinates and speed component are updated accordingly.

- bar_impact(float bx, float by):void
  This method checks if, with the next movement, the ball will impact with the bar. If it is so, the ball coordinates and speed components are updated accordingly.

- compute_new_coords(std::pair<float, float> ball_coefficients, std::pair<float, float> line_coefficients, std::pair<float, float> start_point, std::pair<float, float> impact_point_coords ):std::pair<float,float>*
  Given the line along which the ball moves, the line upon which the bar resides, the ball start point and the computed impact point this method computes the new coordinates of the ball.

- impact_point(float coeff_ball, float known_term_ball, float coeff_line, float known_term_line, std::pair<float, float>* result):void
  Given the bar line and the ball line this method computes the impact point between the two lines.

- check_ghost_impact(float delta_x_center, float delta_x_lenght, float delta_y_lenght ):bool
  This method checks whether the bar will skip over the ball with its next movement. If it is so the method returns true.

**Public methods**

- draw():void
  This method is called by ofApp during the game phase in order to print the state of the gameboard.
- init():void
  Initializes the class. Its main use is for it to be overloaded in future class extensions, since the initialization of this class is done in the constructor method.
- check_bound_r():void
  Checks if it is possible to move the bar right, also checking for ghost impacts. If the action is possible the data is updated accordingly.
- check_bound_l():void
  Checks if it is possible to move the bar left, also checking for ghost impacts. If the action is possible the data is updated accordingly.
- check_bound_u():void
  Checks if it is possible to rotate the bar counter-clockwise, also checking for ghost impacts. If the action is possible the data is updated accordingly.
- check_bound_d():void
  Checks if it is possible to rotate the bar clockwise, also checking for ghost impacts. If the action is possible the data is updated accordingly.
- lives():int
  Returns the number of lives left.
- bricks_alive():bool
  Returns true if there is at least one brick left.
- game_field():void
  Deallocates painter instance and cleans the pointers.
- move_ball():void
  Moves the ball, checking for for all the different kinds of impacts and for life loss. Data is updated according to the results obtained.
- draw_setup():void
  Calls the painter methods relative to the visual representation of the setup phase.

- draw_endgame():void
  Calls the painter methods relative to the visual representation of the endgame phase.
- game_field():game_field
  Class constructor and initializer. The bricks are generated randomly everytime.

# 4 Application workflow

The application is controlled by the loop provided by OF thus entering setup() when the application starts; then the methods update and draw will alternate for the duration of the application execution time. When it is time to terminate the application the method ofBaseApp::exit() will be called.
In order to understand the algorithms there are two things worth noting:

- The area on which the game elements are positioned will be treated as a cartesian plain, with the origin at the bottom left corner, x coordinates grow when moving to the right and y coordinates grow while moving upwards. The dimension of the field is hardcoded as 20 by 30.
- Most of the times std::pair<float, float> is used to represent points coordinates. The first element is the x coordinate and the second one is the y coordinate.

In this section will discuss the information flow and the application mechanisms in each of the three phases

## 4.1 Setup phase

As mentioned in the introduction, this is the state in whih the application starts. During this phase the player has the possibility to regulate the Kinect's angle by using the up and down arow keys. When one of these two keys is pressed the method ofApp::keyPressed(int key) is called and, if the new angle is lesser than 30 and greater then -30 the changes are applied to the Kinect. The conditions on the angle are there in order to protect the Kinect's motor from breaking while trying to reach unreachable positions.
Until the right arrow key is pressed the update() method simply refreshes the camera feed and prints the depth view on the screen, while the painter class is asked (via the gameboard pointer) to print on the screen the instruction for the user, as well as an overlay over the Kinect depth view to indentify visually the areas of interaction.
When the user presses the right arrow key the variable setup__var is set to false, thus prompting the switch from the setup phase to the game phase.

## 4.2 Game phase

The game phase constitues the most important and temporally long phase.
The first thing done is a check over the number of bricks left and of live left: if either of those is 0 the endgame variable is set to true and the control cycle will not be performed.
While in the game phase the controller alternates between updating the bar data unsing the Kinect depth feed and prompting the updates on the ball position (and subsequent changes to the data) to the element pointed at by playfield element.
Using the binary variable update_cycle, the ofApp::update() method alternates between the two aformentioned activities

### 4.2.1 update_cycle==false

When in this case the ofApp::update() method prompts the game_field instance to move the ball by calling the method game_field::move_ball().

**Hit bricks** The first thing done during the execution of this method is to compute the new coordinates by adding to the current coordinates, contained in ball_coords, by adding to the x coordinate the x component of the speed of the ball and by adding to the y coordinate the y componentof the speed of the ball.
If y component of the start point or of the end point are above the lower limit of the lowest row of bricks then it is possible that the ball will hit a brick on its trajectory, thus the method game_field::brick_impact is called, passing as parameters the coordinates of the ball after the movement.
The hittable_bricks vector is then scanned, searching for impact points that are actually on the bricks, in the right direction and close enough to the ball to be reached. Of all the possible points the closest one is considered to be the right one.
If any impact is found the brick gets hit and the ball updates its speed components: if the brick was hit on the side then the x component is inverted, if instead the brick is hit on the high or the low side the y component is inverted; while the ball position is updatedd according to the impact point.

**Brick destruction** If a brick gets destroyed its is_alive attribute is set to false, then all the surrounding bricks in the wall matrix are evaluated: if a brick is adjacent, alive and not hittable it is set as hittable and added to the hittable_bricks vector. Then the destroyed brick is erased from the hittable_bricks vector.

**Bar collision** If no brick gets hit the coordinates of the ball won't change, thus a check on the ball_coords is perfomed by game_field::move_ball. If the

coordinates have changed the method returns, otherwise it calls the method bar_impact, passing the ball coordinates after the movement as argouments. This method checks whether the ball will collide with the bar. This is done by computing the impact point of the two lines defined by the bar itself and by the ball speed. If the computed impact point is invalid the method will return, otherwise a change of coordinates is done by performing a rotation, in order to find a x' axis parallel to the bar line.
The ball position are rotated accordingly, then the rotated y velocity is inverted and the counter rotation is performed, thus finding the new speed components. The new ball coordinates are then computed by finding the intersection point between the ball speed line and the line paralle to the bar line and whose distance from the bar line is equal to the bar radius.

**Border collision** If no bar hit is detected (once again by checking whether the ball has updated its coordinates), the border_impact method is called, once again passing the computed new coordinates as argouments.
The collisions with the border are checked analizing the argouments: if the x ccoordinate is <0 there is an impact on the left, if it is >20 there is an impact on the right and if the y component is >30 ther's an impact on the roof.
If a collision is detected the ball coordinates are updated accordingly and the speed gets modified (an impact no the left or the right prompts a x component inversion, while an impact on the roof promts a y component inversion).

**Life lost** If, again, the coordinates of the ball haven't changed a check is performed on the y component of the new coordinates: if it is lesser than 0 a life is lost and the speed and coordinates of the ball are reset. Otherwise the ball coordinates simply get updated and the method returns.

### 4.2.2 update_cycle==true

In this situation the ofApp::update method will set the update_cycle variable back to false and will then proceed to elaborate the Kinect feed

**Centroid computation** The algorithm will sample some points (1 every 100) from the depth field and discard the ones that are not inside the desired range (from 1 meter to 1 meter and 15 centimeters). The coordinates of the undiscarded points are then averaged, thus obtaining the coordinates of the centroid of the cluster composed by valid points.

**Command recognition** The coordinates of the centroid are then compared to che coordinates of the center of the area seen by the Kinect, plus or minus a treshold in order to grant a dead zone, both vertically and horizontally. A high centroid will make the bar rotate counter-clockwise, a low one will make it

rotate clockwise, a centroid on the left will make the bar move to the right and a centroid to the right will make the bar move to the left. This is so since the Kinect depth feed is not specular, thus the left-right axis had to be inverted.
During the updates both the checks on the boundaries and the modifications to the bar attributes are performed

**Ghost impact check** In order to avoid the bar skipping over the ball and thus missing to register an impact a check is performed: if the ball is on the intersection between the two semi-planes defined by the lines upon which the bar lays before and after the rototranslation, and the ball is close enough to the bar, the rototranslation is not allowed.

### 4.2.3 Visual representation

At each ofApp:draw execution the game_field prompts the painter to uppdate the visual representation, passing information about the bricks(whose colors change according to their lifepoints left), ball position, bar position and attributes and lives left.
The painter also draws the boundaries of the field and information about the meaning of the different brick colors.

## 4.3 Endgame phase

In this phase the player has to choose whether to play again or to exit the application. This is done by using the arrow keys: up means the player wants to play again, down means the player wants to exit the game.

### 4.3.1 New round

When the up key is pressed both the variables chose and play_again are set to true; thus ofApp::update will reset all the variables, delete the current game_field instance and create a new one and the phase will go back to setup.

### 4.3.2 Exit application

When the down key is pressedd the fariable chose will be set to true, while the variable play_again will be set to false; thus ofApp::update will delete the current element pointed by gameboard, close the Kinect feed and exit the application.

### 4.3.3 Video representation

Painter will display a different message according to whether the player has won or has lost the game; it will also display the instructions on how to play again or exit the game

# 5 User manual

## 5.1 Setup

In order for the application to work OF is needed, as well as the OpenKinect libraries. The README file contains the links to their webpage, but a quick search on internet should get you there as well.
Once you have downloaded the repository from git, modify the Makefile by expliciting the rocation of the root folder of OF and the the path to the default OF makefile. The latter one is present as a comment in the project Makefile, if you followed the standard install procedure should work just fine.
Open then the directory containing the makefile in a terminal and type "sudo make" to compile the release version. The sudo command is needed, otherwise it won't be possible to create the needed folders.
Once you have compiled the code, type "make RunRelease" to start the application (note that the command is case sensitive).

## 5.2 Playing the game

The Kinect will be set to only consider objects between 1 meter and 1 meter and 15 centimeters of distance from the device, so during the setup take into consideration the distance at which the Kinect stops seeing your hands.
Given how the code works you can use anything that can be picked up by the Kinect sensor to play.

## 6 Resources and reference material

- APSC slides and material for the academic year 2019-2020
- ”A tour of C++”, by Bjarne Stroustrup
- Openframeworks documentation and community support
- OFbook
- cppreference.com
- cplusplus.com
- OpenKinect documentation