Design and implementation of a multiplayer game system on a big screen with human interaction using smartphones

Agustí Sobejano Ventayol

Abstract– This paper proposes a software design of a multiplayer game framework that allows many people to play together on the same screen by interacting with their own smartphones. The architecture for a flexible multiplatform game engine and a mobile app is described. It also proposes a simple event-oriented communication protocol that makes possible the human interaction with the games. Furthermore, in order to demonstrate the possible uses of this system, several games of distinct natures where implemented that uses different sensors from the device. Finally, there is a simulation of a significant number of players connected at the same time to make an analysis of the performance and the gameplay experience at those situations. It concludes in a working implementation that lets a couple of users to play some games together using a mobile application.

Keywords– Videogame, Multiplayer, Smartphone, Big Screen, Human Interaction, Leisure Time

1 Introdution

The videogame industry is nowadays a huge one that has been evolving for decades. It goes from the arcade machines to private home gaming on game consoles and computers, both with multiplayer capabilities and using many different kinds of controllers and a wide range of experiences.

We can frame some key technologies that made possible the videogame to reach to any person in the world. The handheld gaming is one of them and the online gaming is the other one. These two are flooding our free time thanks to the smartphones we carry in our pockets. We can play videogames anywhere, at anytime with anyone over the globe.

This paper focuses on develop a system that aims to merge the local multiplayer gaming with the powerful world of the outdoors smartphone interaction and socialization. I called this project Giant Play.

It is organized the following way. In section 2 I make a brief introduction of the current state of the art. In section 3 and 4, the objectives and the methodology of the work for this project. In section 5 the full design explanation including the architecture and software modules. In section 6, the algorithms and game implementations. And finally, in section 7 and 8, there are some results and a conclusion that finish the paper.

• E-mail de contacte: agusti.sobejano@uab.cat
• Menció realitzada: Computació
• Treball tutoritzat per: Dr. Fernando Luis Vilariño Freire (CVC)
• Curs 2018/19

2 State of art

Despite the big amount of teams and companies that produce videogames systems and research over new experiences of playing, there is not much investigation about this kind of local collaborative game.

The most meaningful project likely to Giant Play is a racing game made by the start-up company Screenreach Interactive [1]. It is oriented to the buyers of a mall and joins people into races of three or four cars each. The players should download an specific app, log into it, and connect with the game using a code. The car moves as the smartphone like a steering wheel and the communication goes by Internet. This project, actually, is not implemented yet in the society, it just was a prototype.

Another powerful project that is currently active on market is AirConsole [2]. This project uses the web to create a game console on the pc. Also by using a code, up to eight people can connect their smartphones to the console to play. There are a big catalog of games and you can pay to get some extra games. The benefit of this project is that its not necessary to download any app because the server and the client are web so the start up is very fast. Although that they have an app as a support or alternative. The communications are done by Internet so it is not recommended to play outdoors.

There is a product whose concept is relevant for this project, the EyePlay [3]. This product consists in a projector installed in the roof that projects multiple minigames on the floor. By using a movement detection sensor, the people can interact with it using their own bodies, in special, the children. This project does not use any kind of smartphone or controller to work but the multiplayer interaction that it can accomplish may be interesting to study. This game, unlike the others, let the people to join or leave the game whenever they want so it encourages the participation making it more attractive.

3 Objectives

The main objective of this project is to design and implement a full stack, multiplatform, flexible and scalable system prototype that let the people to play with a smartphone in multiplayer mode sharing the same game screen. This design must have a server-client orientation and should let developers to make its own games. The design also must be capable to handle any kind of event and any kind of interactive device.

The second objective is to develop several play modes and game paradigms to show all the different uses of this system. Furthermore, some game implementations that demonstrates its power too.

4 Methodology

The development has been taken in place using an agile methodology with some deliverable and meetings with my tutor.

I have been working fully independent. The first few weeks was devoted exclusively to documenting, designing and defining the system completely.

The implementation is done in an incremental way by testing each new feature. This methodology lets us to prioritize the tasks and hold such a contingency plan to always have a functional system despite the time spent or the resources.

The development and testing will be carried out on a PC using IDEs such as Android Studio or PyCharm.

5 Design

In this section the whole design of the system is presented, its modules and the technologies to be used.

The Figure 1 exposes the scheme of modules and communications of the architecure, which I will explain below and in the following subsections.

The architecture of Giant Play is client-server based. Where the clients are the smartphones, and the server is the main screen.

The server contains all the game logic and user handling. The clients (i.e., the devices) are connected to this module and authenticated as users. Then, the game instances a handler for them in order to take a context inside the game. This module also has an event distribution system to make the events from the clients to reach the games

The clients are dumb implementations whose function is to send events to the server that can describe the actions that the user is performing.

The whole system is based on events which were designed to be multi-purpose and independent of the system and the game that uses them. An event is a small message formed by a key or name and a list of parameters, usually numbers. They are the main method of communication between the client, the server, and the game that it is currently executing. Each system defines what events can generate and the meaning of its parameters. Each game listens for a set of specific events to interact. Furthermore, some modules whose objectives are to transform raw events to other virtual events that games can process may be present.

The following subsections explains in detail how these components and modules work.

5.1 Server

The server is the biggest and most complex component of this system. The server is a portable app that has graphics but not direct user interface. It contains the main loop of the program; the communications module, which handles the clients connections, the discovery handshake and the receives the events; the users module, which handles the connection and disconnection of the players; the event libraries and game utilities, that ease the development of games for the platform; and finally, the game infrastructure, that contains the basic components to build your own games, logic and graphics.

The following subsections explains in detail what are and how all of these modules of the server work.

5.1.1 Communications module

This module implements the interface between the server and the client devices. The Figure 2 shows an example of what messages have be sent from both in order to establish a successfully communication.

Firstly, the client needs to know where the server is located. The communications module has an independent sub-module that receives discovery UDP[1]

[1]User Datagram Protocol

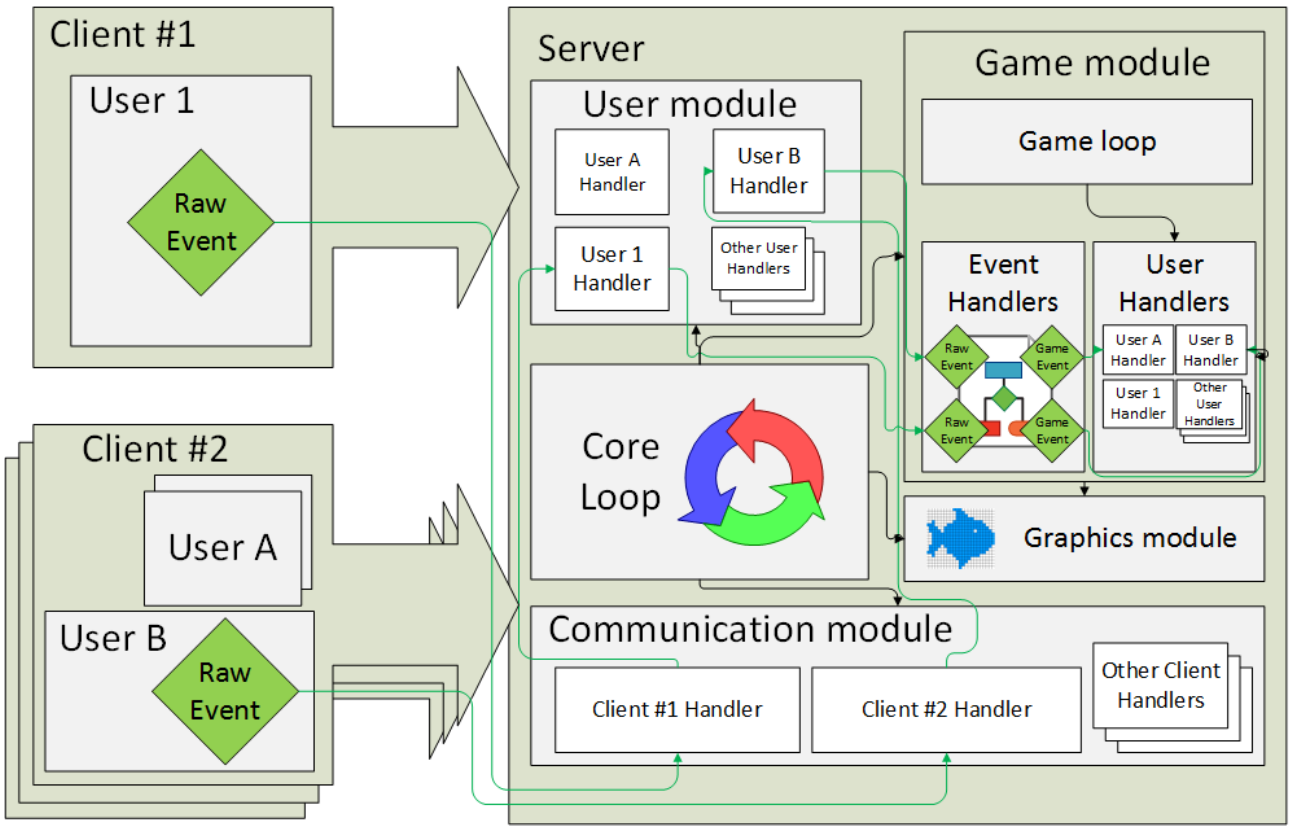


Fig. 1: Modules of the server and path of the events

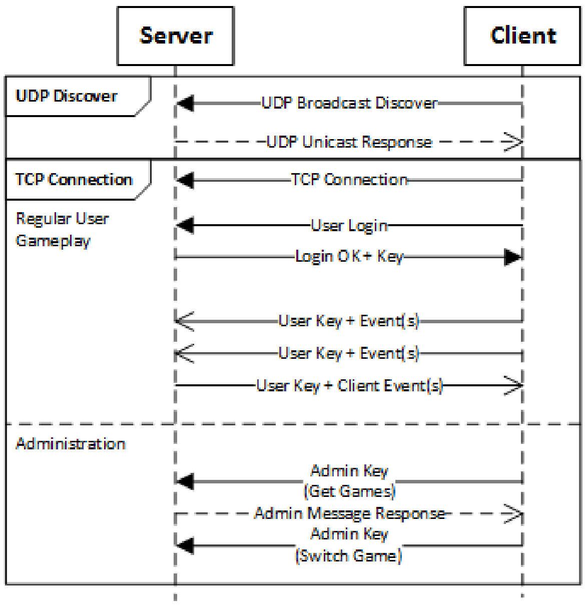


Fig. 2: Modules of the server and path of the events

broadcast messages in order to be responded to the askers with another UDP message [9]. This method of discovering is available thanks to the local area network, so there is not necessary to locate the server on a specific IP[2] but a port.

This last message let the client to know the IP of the server. Then, a TCP[3] socket communication is started and this module is responsible for handling it. The communications module manages the client connection and disconnection and the receiving and sending of messages, every action is delegated to the core module.

The typology of the messages is plain-text JSON[4] and follows a protocol that was designed specifically for this project (See the Appendix A.1). JSON format has some advantages over binary format: It is dynamic, easily debuggable and extensible and is not a problem of performance for this application.

5.1.2 Users module

Because of the fact that one device could run multiple users (i.e., players), it is needed a distinction between them. For example, in test purposes, a single smartphone can be used to test multiple users. This is why the client after connecting with the server has to log in a user to start sending events. A client can log in as much users as it want, the server will respond with a key for each one, and this key is what the client will use to distinguish between users at the moment of sending event messages.

The user is the entity that the games will understand and handle independent of the client that it uses. The users module is responsible for login and logout them, send or receive and distribute the user events and handle the disconnection of the users when a client device is disconnected.

[2]Internet Protocol address
[3]Transmission Control Protocol
[4]JavaScript Object Notation

In the login process of a user, there is some information that has to be sent to the server besides the name, the type of device and some parameters. E.g., the smartphones have to send their screen sizes.

5.1.3 Game module

The game module, as it is referenced above, contains the basic classes to let the developers implement their own games. There are four components that should be extended to make a game:

- **Game Loop**

  It has two methods: update and render. This class implements the logic of the global game, the game scenario or the interaction between users. These methods are programmatically called repeatedly by the core module to make the game run.

- **User Handler**

  This class is instanced and deleted for each user that has been connected or disconnected from the game. The events that the physical one sends to the server are attended on this object. It is used also to send small events back to the device to enable some actions on it.

- **Event Handler Tree**

  An event handler is an object that listens for an event and handles it. Sometimes it is not possible the games to understand the raw events the user have sent to the server. In these cases it is necessary to treat them by transforming them, by collecting them into groups or even by classifying them in order to be delivered, in good conditions, to the user handler any time now. This tree actually builds a path where the events will go across before reaching the game.

- **Game Builder**

  This class represents the game presence in the Giant Play server. It declares the game metadata, such as the name of the game, and instances it when the server requests it in order to play. It also instances user handlers and builds the event handler tree every time a user is connected.

5.1.4 Configuration File

This server is designed in such way that only one file have to be modified every time a new game implementation has to be linked in. Although the file is written in python language, it can be interpreted as a properties file. It contains such data as the maximum FPS[5], the list of available communication modules and the list of game builders that can be used during the execution.

[5]Frames Per Second

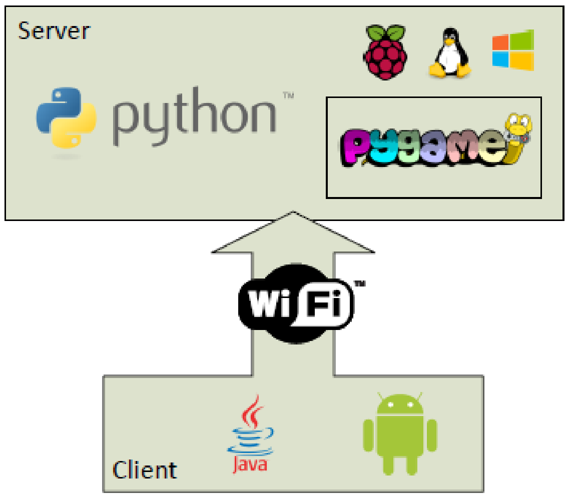


Fig. 3: Technologies used in the system.

5.2 Client

The client is a device that connects to the server, logs in a user and starts sending events. The main target device to be a client of Giant Play is the smartphone. Its main sensors are the touch screen and the orientation, and eventually the GPS[6]. The proximity sensor, NFC or the camera are not taken in account for now in order to simplify de development and not to overload the traffic. Furthermore, it has some features that can be used as event actuators such as the led screen, the rumble feature or the flash. The full list of implemented events for the smartphone client can be seen in the Appendix A.2.

6 Implementation

This section explains what technologies and programming languages have been used to implement this systeam (Figure 3). It also shows a couple of software that has been implemented for the system: a simple collision system and event handlers that generate virtual sensors. Finally, it also shows the three games that where developed in order to make use of them.

For the server, Python will be used due to its ease to develop quickly and the tidiness of this code. Python, in addition, has a lot of modules ready-to-use that can manage sockets [4, 5], game developing [6] and matrix manipulation easily. It is multiplatform and it can even be deployed on embedded systems, such as Raspberry Pi [7].

For the clients the chosen platform is Android because the application has to be capable to configure and test the system, so it will be a small administration application that is not targeted to the final public although is full functional to play. This application can do administration tasks at the server, log in multiple users, play physically with them or sim-

[6]Global Positioning System

ulate many users at the same time which ones can generate pretty much traffic to the server.

When this app boots, automatically does the communications handshake to discover and connect to the server. If it is not found, the app is finished. It also hardcodes the admin key in order to be able to get the list of available games in the server and switch between them.

As it is shown in the Figure 4 there is a screen that lists the available games in the server and another that shows the connected users from this client. Furthermore, there is a screen that collects the events from the sensors and sends them to the server in name of an activated user. This last one is not present in the Figure 4.

For the testing purposes, the app is able to send cloned events from multiple users concurrently. This feature follows the rule that any player that starts with the same word as the active one, will send the same messages but with a different delay each. The user can select which of them becomes active at any moment.

Finally, the communications will be satisfied using TCP sockets over a WLAN[7][8]. This method reduces as much as possible the latency of the messages, makes easier the task of testing, does not consume contract data of Internet and requires the client to be physically near to the server, which is the objective of the game.

6.1 Event Handlers

It has been implemented three useful reusable event handlers to be included in the event handler tree of the games.

6.1.1 Screen Bounds Adapter

This handler captures any kind of touch event and transforms mathematically its coordinates from the device coordinate space to the game screen coordinate space.

Because of the different devices and their variable screen sizes, it may be useful that every coordinate that enters the game its being transformed into its bounds. This fact makes the players to play at the same screen size conditions.

6.1.2 Orientation to Aim

This other handler takes the orientation in form of a quaternion and creates a new virtual sensor that represents an aim at the screen. As a gun style but using the smartphone.

The sensor is inspired by the all existing remote pointing devices that already exist in the market such as smart TV or game consoles. E.g., the Wii Motion Plus [10] controller by Nintendo or the PlayStation Move [11] by Sony. But, specially, the Wii Motion Plus which uses only a gyroscope which one is not necessary to be configured or calibrated.

The problem that has to be solved is that we do not want to calibrate the aim manually depending on the orientation of the main game screen. The Wii Motion Plus gives us the idea to solve it. As in the Figure 5 is shown, the server stores an orientation initially pointing to north that says the truth about where the screen is oriented at. This value can be changed simply moving the aim of the smartphone out of bounds of the logically screen. When this occurs, the reference is readjusted so now the playing screen has moved. Always, the player will be able to play in any orientation they want and even, in the real orientation where the screen is oriented at. This algorithm is explained in the Appendix A.3.

However, the calibration process is only effective in the yaw axis. The pitch one is always centred parallel to the floor. The roll one is not implemented yet due to simplify the development [12].

The output event of this handler has two coordinates, which are referencing the screen coordinates where it is pointing.

This new sensor is very powerful because it represents a pointer at the screen without using the touch input which one can be used for another events or actions. There are some examples of the usage in the Section 6.3.

6.1.3 Orientation to Axis

There is another virtual sensor that can be very useful for the usage of the games: an axis stick. This handler provides a 2d axis based on the orientation of the smartphone.

The axis points to zero when the screen of the device is face up. You can move the vertical axis by turning it forwards or backwards, the horizontal one, by tilting the phone to the sides.

The output of the handler is an array of two real axis that bounds from minus one to one.

This sensor can be used to move objects over the screen such as cars or characters.

6.2 Game Utilities

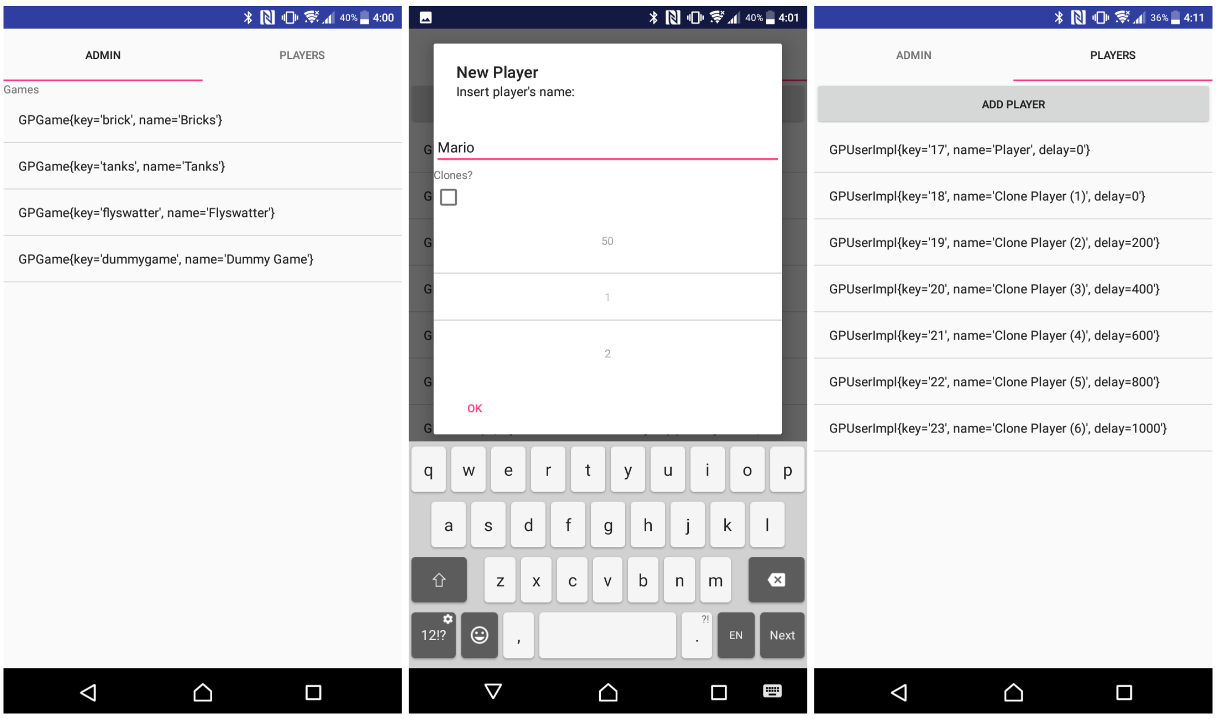
There are two little utilities that were been implemented for this project: a collision detection function and a bounce angle computation one.

Both two uses a square box model that simplifies either the implementation or the complexity and performance of the algorithm.

6.2.1 Collision Detection Model

This algorithm lets the game to set fixed objects on the screen (E.g., walls) in order to test quickly the collision between them and other mobile objects.

[7]Wifi Local Area Network



Games

GPGame{key='brick', name='Bricks'}

GPGame{key='tanks', name='Tanks'}

GPGame{key='flyswatter', name='Flyswatter'}

GPGame{key='dummygame', name='Dummy Game'}

New Player

Insert player's name:

Mario

Clones?

50

1

2

GPUserImpl{key='17', name='Player', delay=0'}

GPUserImpl{key='18', name='Clone Player (1)', delay=0'}

GPUserImpl{key='19', name='Clone Player (2)', delay=200'}

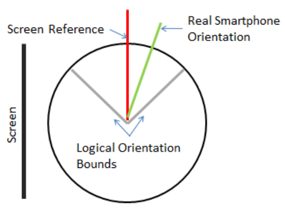
GPUserImpl{key='20', name='Clone Player (3)', delay=400'}

GPUserImpl{key='21', name='Clone Player (4)', delay=600'}

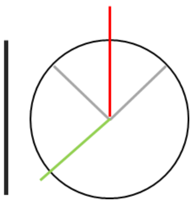
GPUserImpl{key='22', name='Clone Player (5)', delay=800'}

GPUserImpl{key='23', name='Clone Player (6)', delay=1000'}

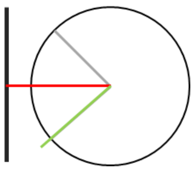
Fig. 4: Screens of the client app. At the left, the list of available games at the server; At the center, the form to login users; At the right, the logged in users and the delay of the clone ones.



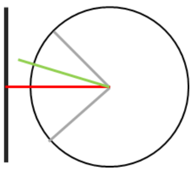
(a) The reference is pointing to north.



(b) The smartphone orientation goes out of bounds.



(c) The reference is recalculated.



(d) The playing orientation does not modify the reference.

Fig. 5: The process of recalibrating the aim to the screen.

The model consists in a grid of cells where these objects are located. There is a direct transformation of the screen coordinates and the cell that is located behind it. When a rectangle has to be tested, it computes all the cells which is overlapping and asks for a wall block in order to return a result.

6.2.2 Bounce Computation Model

There is a very fast bounce computation algorithm implemented in the utilities package. This one is able to compute the bounce angle of a bullet on a wall that is made of square-shaped blocks.

There are many factors what influence on the correctness result of this algorithm. It either takes in account the position of the bullet and its velocity or the surrounding blocks.

The algorithm follows these points: Which cell is the bullet at; Which cell is the bullet going at; Which cells are the bullet going through; And which face is going to collide on. Once it is everything in hands, the algorithm is ready to return the output angle in order to modify the velocity of the bullet. The algorithm is explained in detail in the Appendix A.4.

6.3 Games

The second objective of the project consists in the implementation of some games that demonstrates the usage possibilities of the system. There are three different gameplay paradigms defined:

- **All together competitive**

  Game mode where the players have the objective of reaching the best punctuation.

- **Champion competitive**

  Game mode where the players fight against others in order to be the last player to be up.

- **Cooperative**

Game mode mode where the more players are in the game, the easier it is to win. They play together to beat the machine.

These modes are the most meaningful ones that can be developed, so there is a game implementation for each one. They are exposed in the next subsections.

6.3.1 All together competitive game: Fly Swatter

The first game was implemented is a game that made use of the orientation sensor with the aim virtual sensor to create a fly swatter or a bird shooter game.

This game, whose screen is in the Figure 6 shows an aim as a circle for each player. Eventually, some black squares that represents the flies appears on the screen and moves randomly in any direction until someone shots them to get points.

The aim is moved thanks to the event handler that converts the orientation events to the aim ones. And the shots are done by touching anywhere on the screen.

When the user makes a shot whether it hit a fly or not a little amount of time has to be passed in order to shot the next. There are unlimited shots.

The game is inspired in the Fly Swatter for Mario Paint or the Duck Hunt both made by Nintendo for their old console systems.

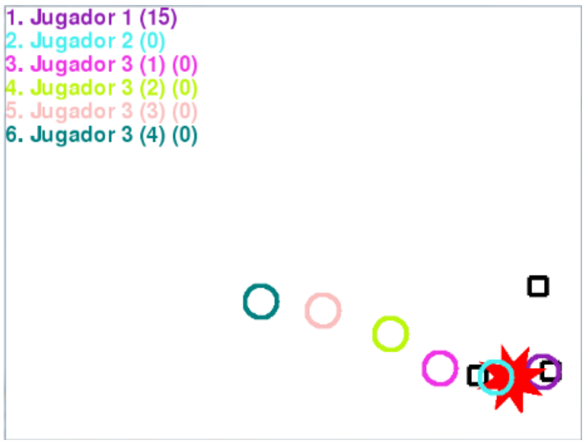


Fig. 6: Game screen of Fly Swatter. There are 6 players and shot that hit a fly, a black square.

6.3.2 Champion competitive game: Tanks

In this game, the player takes the form of a tank that can shoot bullets. It can both move in any direction and steer the barrel in any other direction, apart from firing it.

The background is a collision block layout. The tanks cannot through out these blocks and the bullets bounce on them. The collision system in the utilities package has been used for these two purposes.

The Figure 7 shows an scene of the game where two tanks battle and one is already dead. Over the tanks, the number of remaining lives is drawn.

The wall layout is chosen randomly at the beginning of the game from a set of predefined ones. The bullets has zero to three bounces before they disappear, this number is randomly chosen as well.

When a bullet hits the tank's hit box it loses a life and it has immunity for a few seconds before it can be hit again.

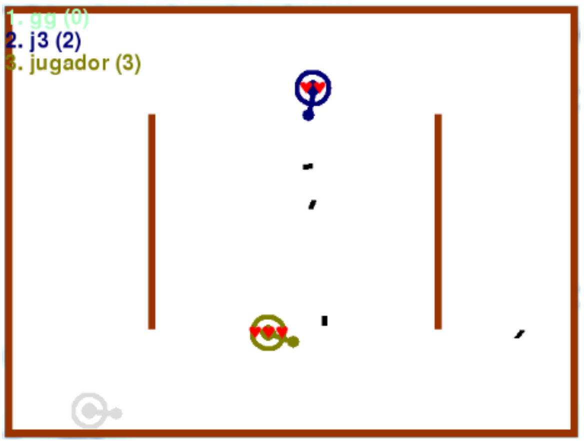


Fig. 7: Game screen of Tanks. There are 3 players, one of them has died. There also are some bullets going through the land.

This game uses all possible events that the smartphone is sending. The orientation one is used to get an axis sensor that moves the tank in function of the inclination. The touch screen to get the orientation of the barrel. And the second touch event to fire the bullets.

The Figure 8 is a graphical description of the event tree that is used in this game. The events of each type goes through its handlers that calls, at the end, the method that performs actions on the game.

This game is inspired on the Tanks game of Wii Play by Nintendo but there are some other online games which this game has referenced on: Diep.io[13] or SuperTanks.io[14].

6.3.3 Cooperative game: Circular Bricks

The last game that was implemented is a cooperative game. The idea is gotten form the old Arkanoid brick game. In this game there is a paddle located at the bottom of the screen that prevents the ball from going over. That ball bounces and crushes the bricks located at the top of the screen. The game is over when all the bricks are broken.

The Circular Bricks game that has been implemented in this project is a variant of the Arkanoid designed exclusively for this system that has been thought to be multiplayer. In this variant, the bricks are located in the center of the screen and the multiple paddles goes around them. Each paddle has only one ball assigned but it can make bounce any of the

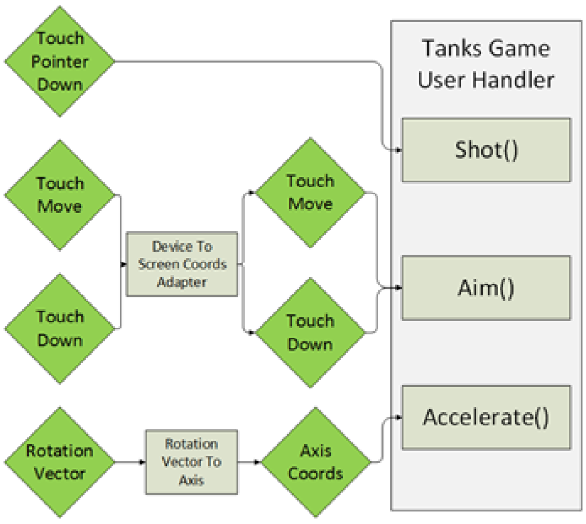


Fig. 8: Tree of events of Tanks.

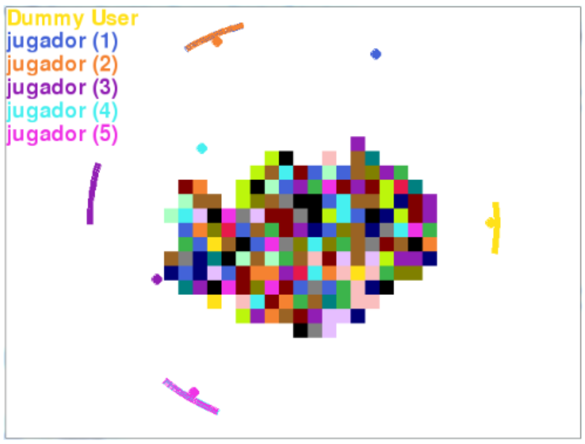


Fig. 9: Game screen of Circular Bricks. Six players are making team to break all the bricks.

others. When a ball goes out of bounds, that ball is re-spawned in the source paddle in order to be fired again. There is no dies or lives, the more players are playing together, the faster they will break all the bricks.

The Figure 9 displays the game in action.

7 Results

In this section it is exposed some results of testing trials. It shows the performance of the system by two points of view: the analytic one, where it shows, in numbers, the maximum capacity of this implementation; and the game experience one, which it is evaluated subjectively.

7.1 Analytic results

We are going to measure the speed of the game in terms of FPS in function of the game, the resolution and the number of players.

There are two data that have to be considered. The first one is that the smartphone has set a fixed event resolution of 15 messages per second which

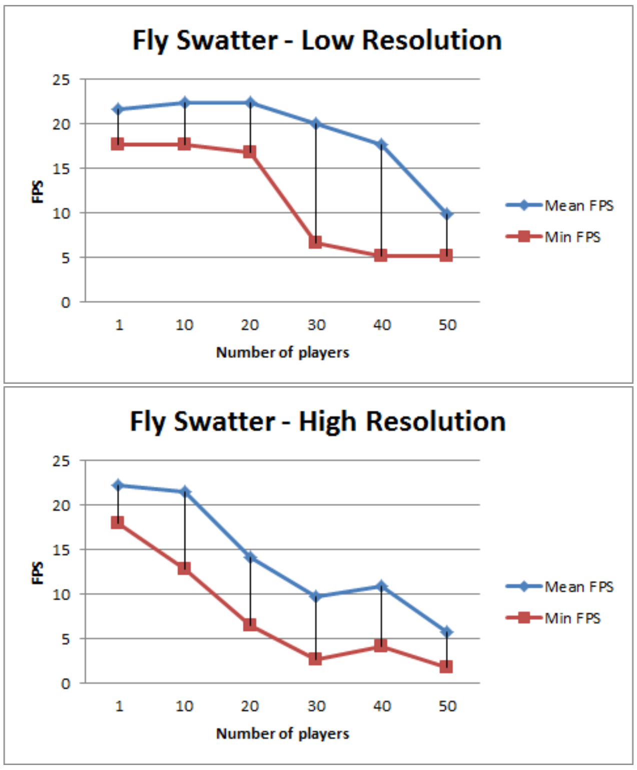


Fig. 10: Results in FPS for the game Fly Swatter.

we considered it's enough to have a good gameplay experience. The second one is that we also fixed a maximum FPS in the server of 24, this sufficient to lie the eye sight for a continuous movement.

The tests are made using a traffic-free wireless lan, the client that has been developed in this project, which one is able to simulate many users and their traffic, and a Windows PC of an I5 CPU and 8GB of RAM. A quick extension of the server exports continuously some statistics into a file, which one has been used to read and analyse them. During the test, from about 15 seconds each sample, the client is playing actively, so all the users are sending events as much as possible.

There are two graphics per game, one in a low resolution (in pixels: 640x480) and the other in a high resolution (in pixels: 960x720). In addition, each chart has two functions, one for the minimum FPS that the simulation reached and the other, the mean value. We interpret that, a low value of the minimum one, means that the game is decreasing its speed eventually due to a high processing input on the game (i.e., collision detection process or event processing). The mean values point to the global game performance.

As it can be seen in the Figure 10, in the game Fly Swatter, which is a game that makes use of the aiming algorithm and the collision detection is minimum, the speed decreases different as of the number of players in low and in high resolution. Apparently, this may not have sense because the number of objects is not increased between both cases.

The game Tanks is, in my opinion, the best game

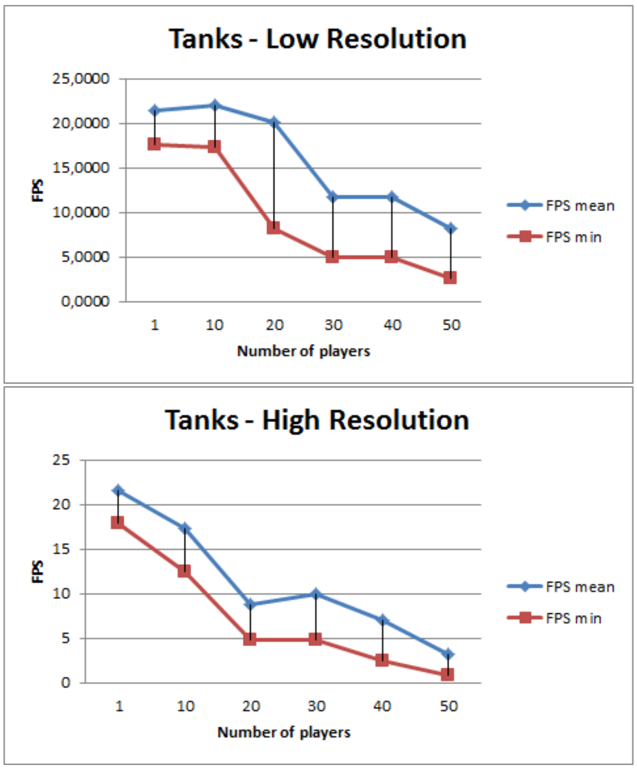


Fig. 11: Results in FPS for the game Tanks.

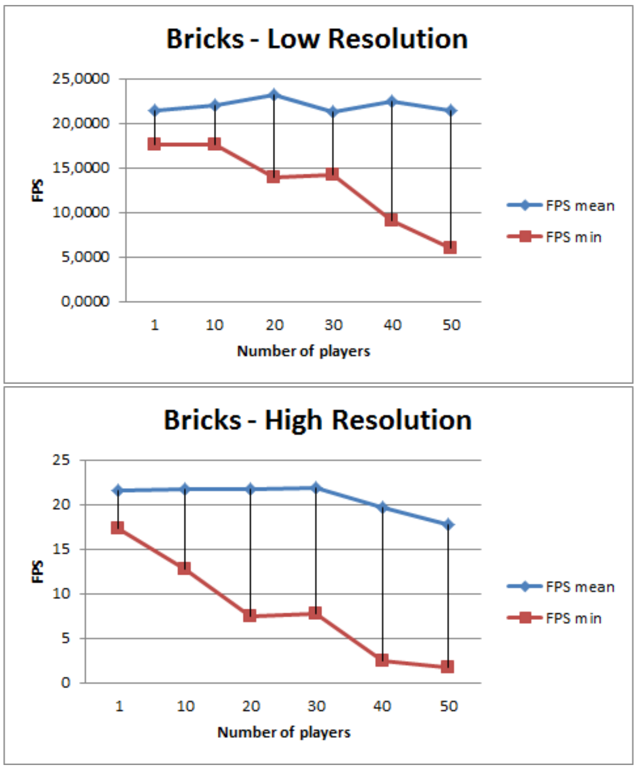


Fig. 12: Results in FPS for the game Circular Bricks.

to stress the server because, not only makes use of every event in the game but also the use of a complete collision and bouncing system. This game, whose results are in the Figure 11, is unplayable when there are more than 20 players because the values are below 10 FPS. There also are no distinction between the two resolution sets since both decrease the speed at the same time.

Finally, the Circular Bricks game (which results are displayed in the Figure 12) demonstrates the best stability in terms of the mean FPS. As of the other games, the minimum FPS values are decreasing in the same way. This game uses the collision system but the number of bricks are not increased in function of the resolution, so the results are very similar.

Following this last argument, these results make me think that the bottleneck is not found in the resolution of the screen nor the complexity of the game but in the number of events that must be processed per second and, therefore, the number of connected users.

7.2 Game Experience results

There is also a subjective analysis that shows where the game looses its capacity of entertaining and turns it into an unplayable game.

We took some shots of the games at +20 players (Figure 13 and we realized that not only the game turns slower but we lose the track of the character in the screen, so the player gets lost and starts to move randomly the smartphone in order to find it out or do something good with no success.

There is a game that this fact is not present: the Circular Bricks. In that case when the number of paddles is very large, they all make a giant wall that any ball can escape so they are able to beat the game very soon (see the Figure 13).

8 Conclusions

In conclusion, this work has been carried out successfully. There is a full working videogame system, open to be extended, that demonstrates many new applications to the market.

The hypothesis of the initial client-server architecture and the event handling works well. The optimal number of players is found between 10 and 15 playing together so its deployment cannot be feasible in very large places with tones of people but in mall halls or squares where they are passing through.

This architecture proposal could be consolidated and converted to a standard or even to a product that can be implemented on these target social places.

The code of this project is stored on a repository on GitHub [18] and use a CC [8] license.

9 Future Work

There is a whole researching field to make this system stronger and working better. For example, it

[8]Creative Commons

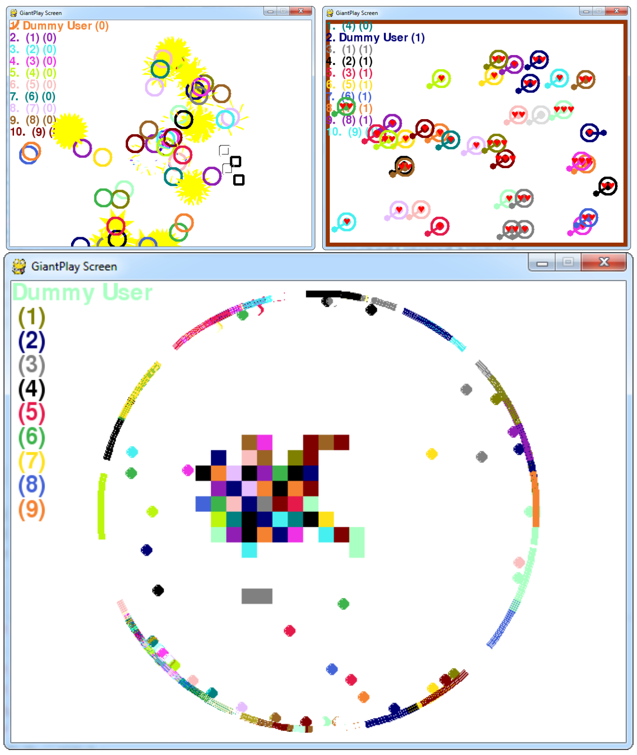


Fig. 13: Screenshots of the games at a high number of connected players. At the top, Fly Swatter and Tanks. At the bottom, Circular Bricks.

can be extended to new mobile platforms and other communication technologies such as bluetooth, the faster UDP or Internet. In addition, the server module could be implemented in other not such user-friendly but faster programming languages like Java or C++. This work would make the server faster and more stable. In this project the security of messaging and DoS[9] attacks are no taken into account so this could be another point at which to develop. On the other hand, an open API[10] could be defined and published to let developers creating their own games for the platform, so it could become popular across them and the players.

Acknowledgments

I would like to thank my tutor to guide me on doing this project. Also to some of my friends, that gave me powerful ideas to make this project grater and my family, parents, brother and uncles, to make me support on studying my vocational field.

References

[1] TheNextWeb.com: Multiplayer game by screenreach news *https://thenextweb.com/uk/2011/10/11/london-mall-gets-big-screen-multiplayer-driving-game-you-control-with-your-phone/*

[2] AirConsole (Official Website) *https://www.airconsole.com/*

[3] Eyeplay (Official Website) *http://www.eyeplay.es/*

[4] Python API: Udp Communication *https://wiki.python.org/moin/UdpCommunication*

[5] StackOverflow: Python TCP Socket Handle Example *https://stackoverflow.com/questions/21027949/python-tcp-disconnect-detection*

[6] Python PyGame: Documentation *https://www.pygame.org/docs/*

[7] Raspberry Pi (Official Website) *https://www.raspberrypi.org/*

[8] Matthias Friedrich's Blog: Using TCP for Low-Latency Applications *https://blog.mafr.de/2010/03/14/tcp-for-low-latency-applications/*

[9] Baeldung: Broadcasting and Multicasting in Java *https://www.baeldung.com/java-broadcast-multicast*

[10] Wikipedia: The Wii Motion Plus *https://en.wikipedia.org/wiki/Wii_MotionPlus*

[11] Wikipedia: The PlayStation Move *https://en.wikipedia.org/wiki/PlayStation_Move*

[12] Wikipedia: Aircraft principal axes. An explanation about yaw, pitch and roll in an axis context. *https://en.wikipedia.org/wiki/Aircraft_principal_axes*

[13] Online Game: Diep.io *http://diep.io/*

[14] Online Game: SuperTanks.io *https://www.supertanks.io/*

[15] DevelperBlog: Stop Worrying and Love Quaternions *https://developerblog.myo.com/quaternions/*

[16] Github: An impletentation of the quaternion *https://github.com/KieranWynn/pyquaternion*

[17] Stackverflow: An impletentation of the raytrace algorithm *https://stackoverflow.com/questions/35807686/find-cells-in-array-that-are-crossed-by-a-given-line-segment*

[18] Github: Repository where all the code of Giant Play is stored. *https://github.com/achiikun/giantplay-tfg*

[9]Denial of Service
[10]Application Programming Interface

Appendices

A.1 JSON Messaging Protocol

The Table 1 shows the format of the protocol messages in JSON and a short description of them.

A.2 Smartphone Events

The Table 2 shows the parameters that an smartphone must send when does login to the server. The Table 3 the events that the server sends to the client and the Table 4 shows the events that a smartphone sends to the server.

A.3 Aim Event Handler Algorithm

The Algorithm 1 it that which the server uses to convert the rotation vector to an aim. This algorithm uses Quaternions [15] to represent the orientation. Part of the implementation of the quaternion object is extracted from github [16].

A.4 Bounce Computation Algorithm

The algorithm 2 was implemented to detect the ball collision with a brick and compute the bouncing angle. This algorithm uses an implementation of the raytrace [17] in order to detect which cells goes the ball through .

Algorithm 1 Quaternion to Aim

```
1: $w \leftarrow$ screen width
2: $h \leftarrow$ screen height
3: $qref \leftarrow$ reference quaternion    ▷ Stored globally
4: $qinput \leftarrow$ input quaternion
5: $qrotated \leftarrow qinput * qref$
6: $vlook \leftarrow Vector(0, 0, -1)$
7: rotate $vlook$ with $qrotated$
8: if $vlook.y > 0$ then    ▷ Looking front
9:     $ibounds \leftarrow Interval(-0.3, 0.3)$
10:    $iscreenw \leftarrow Interval(0, w)$
11:    $iscreenh \leftarrow Interval(0, h)$
12:    $x \leftarrow$ map $-vlook.z$ from $ibounds$ to $iscreenw$
13:    $y \leftarrow$ map $vlook.x$ from $ibounds$ to $iscreenh$
14:    if $vlook.x < ibounds.min$ then    ▷ Out of bounds by the left
15:        $angle \leftarrow$ angle between $vlook$ and $Vector(ibounds.min, vlook.y)$    ▷ Angle which to correct the ref
16:        $qref \leftarrow qref * Quaternion(Vector(0, 0, 1), -angle)$
17:    end if
18:    return $Event("aim", Point(x, y))$
19:    if $vlook.x > ibounds.max$ then    ▷ Out of bounds by the right
20:        $angle \leftarrow$ angle between $vlook$ and $Vector(ibounds.max, vlook.y)$    ▷ Angle which to correct the ref
21:        $qref \leftarrow qref * Quaternion(Vector(0, 0, 1), angle)$
22:    end if
23: else    ▷ Looking back
24:    if $vlook.x < 0$ then
25:        $angle \leftarrow \pi/2 -$ (angle between $vlook$ and $Vector(ibounds.min, vlook.y)$)
26:        $angle \leftarrow angle +$ (angle between $vlook$ and $Vector(-1, vlook.y)$)
27:        $qref \leftarrow qref * Quaternion(Vector(0, 0, 1), -angle)$
28:    end if
29:    if $vlook.x > 0$ then
30:        $angle \leftarrow \pi/2 -$ (angle between $vlook$ and $Vector(ibounds.max, vlook.y)$)
31:        $angle \leftarrow angle +$ (angle between $vlook$ and $Vector(1, vlook.y)$)
32:        $qref \leftarrow qref * Quaternion(Vector(0, 0, 1), angle)$
33:    end if
34: end if
```

Name	Direction	JSON	Description
Login	C-S	`{ { "action": "login", "name": "myname", "type":"smartphone" "props":{ "screenw": 500, "screenh": 600 } }`	Message sent by a user to sign in. It includes its name, the device type and its specific properties.
Login Ack	S-C	`{ "action": "login", "key": "a" }`	Message returned by the server after the login. A key is sent in order to identify this user in future messages.
Logout	C-S	`{ "action": "logout" "key": "a" }`	Message sent by a user to do log out.
Admin	C-S	`{ "action": "admin", "adminkey": "YYY"  // Recievable "games":[{ "key":"game1", "name":"Game 1" }, ...], "devices":[{ "ip": "192.168.0.44", "id": "phoneid" "users":[{ "key": "user1", "name": "name" }, ...], }, ...],  // Editable "game":"game1" }`	Special message that requires an admin key to do some admin tasks. Some parameters are read-only and other are write-only.
Event	C-S, S-C	`[ "a", ["type1", 3, 4, 5], ["type2", 4, 5] ]`	Tiny message to communicate the server for user events.

Table 1: JSON messages which define the system protocol.

Algorithm 2 Bounce Computation Algorithm. Part 1/2.

```
1: $cellsize \leftarrow$ size of the cells
2: $from \leftarrow Vector($ball position$)$
3: $to \leftarrow Vector($ball position$) + Vector($ball velocity$)$
4: $lastcellisblock \leftarrow false$
5: $lastcellidx \leftarrow null$
6: $lastcell \leftarrow null$
7: for all $cell \leftarrow RayTrace(from, to)$ do    ▷ Compute and iterate all the cells where the bullet will cross through by
8:     $cellidx \leftarrow Point(cell.x/cellsize, cell.y/cellsize)$
9:     $cellisblock \leftarrow IsBlock(cellidx)$
10:    if $\neg lastcellisblock\ \&\ \neg cellisblock$ then
11:        return $null$
12:    else if $\neg lastcellisblock\ \&\ cellisblock$ then
13:        if $cellidx.x > lastcellidx.x$ then
14:            if $cellidx.y == lastcellidx.y$ then
15:                return $LEFT$
16:            else if $cellidx.y > lastcellidx.y$ then
17:                $isblockleft \leftarrow IsBlock(Point(cellidx.x - 1, cellidx.y))$
18:                $isblocktop \leftarrow IsBlock(Point(cellidx.x, cellidx.y - 1))$
```

Propietats	**Tipus**	**Descripció**
screenw	enter	Width of the client screen in pixels in a vertical orientation.
screenh	enter	Height of the client screen in pixels in a vertical orientation.

Table 2: Parameters that a smartphone must send when does login to the server.

Property	**Type of Data**	**Description**
vibrate	Array of integers: milliseconds to switch the vibration mode on and off	Creates a pattern to switch the vibration of the smartphone on and off.
color	Array of three integers: RGB	Sets the color of the screen.
flash	Array of integers: milliseconds to switch the flash on and off	Creates a pattern to switch the flash of the smartphone on and off.

Table 3: Events the server can send to a smartphone.

Property	**Type of Data**	**Description**
tdown	position x: integer; position y: integer	The user has touched the screen.
tmove	position x: integer; position y: integer	The user has moved the finger over the screen.
tup	position x: integer; position y: integer	The user has just released the finger from the screen.
tpdown	position x: integer; position y: integer	The user has touched the screen with a second finger.
tpmove	position x: integer; position y: integer	The user has moved the second finger over the screen.
tpup	position x: integer; position y: integer	The user has just released the second finger from the screen.
rotvec	Attributes of a Quaternion, in this order: x, y, z, w	The smartphone has changed its orientation.

Table 4: Events the smartphone sends to the server.

Algorithm 3 Bounce Computation Algorithm. Part 2/2.

```
19:                 **if** $isblockleft\ \ \&\ \ \neg isblocktop$ **then**
20:                     **return** $UP$
21:                 **else if** $isblocktop\ \ \&\ \ \neg isblockleft$ **then**
22:                     **return** $LEFT$
23:                 **else**
24:                     **return** $CORNER$
25:                 **end if**
26:             **else**
27:                 $isblockleft \leftarrow IsBlock(Point(cellidx.x - 1, cellidx.y))$
28:                 $isblockbottom \leftarrow IsBlock(Point(cellidx.x, cellidx.y + 1))$
29:                 **if** $isblockleft\ \ \&\ \ \neg isblockbottom$ **then**
30:                     **return** $DOWN$
31:                 **else if** $isblockbottom\ \ \&\ \ \neg isblockleft$ **then**
32:                     **return** $LEFT$
33:                 **else**
34:                     **return** $CORNER$
35:                 **end if**
36:             **end if**
37:         **else if** $cellidx.x == lastcellidx.x$ **then**
38:             **if** $cellidx.y == lastcellidx.y$ **then**
39:                 **return** $null$
40:             **else if** $cellidx.y > lastcellidx.y$ **then**
41:                 **return** $UP$
42:             **else**
43:                 **return** $DOWN$
44:             **end if**
45:         **else**
46:             **if** $cellidx.y == lastcellidx.y$ **then**
47:                 **return** $RIGHT$
48:             **else if** $cellidx.y > lastcellidx.y$ **then**
49:                 $isblockright \leftarrow IsBlock(Point(cellidx.x + 1, cellidx.y))$
50:                 $isblocktop \leftarrow IsBlock(Point(cellidx.x, cellidx.y - 1))$
51:                 **if** $isblockright\ \ \&\ \ \neg isblocktop$ **then**
52:                     **return** $UP$
53:                 **else if** $isblocktop\ \ \&\ \ \neg isblockright$ **then**
54:                     **return** $RIGHT$
55:                 **else**
56:                     **return** $CORNER$
57:                 **end if**
58:             **else**
59:                 $isblockright \leftarrow IsBlock(Point(cellidx.x + 1, cellidx.y))$
60:                 $isblockbottom \leftarrow IsBlock(Point(cellidx.x, cellidx.y + 1))$
61:                 **if** $isblockright\ \ \&\ \ \neg isblockbottom$ **then**
62:                     **return** $DOWN$
63:                 **else if** $isblockbottom\ \ \&\ \ \neg isblockright$ **then**
64:                     **return** $RIGHT$
65:                 **else**
66:                     **return** $CORNER$
67:                 **end if**
68:             **end if**
69:         **end if**
70:     **end if**
71:     $lastcellidx \leftarrow cellidx$
72:     $lastcell \leftarrow cell$
73:     $lastcellisblock \leftarrow cellisblock$
74: **end for**
75: **return** $null$
```