

A Visual Programming Language for Soccer

António Belguinha
ISE, University of the Algarve
Campus da Penha
Faro, Portugal
a34442@ualg.pt

Pedro J.S. Cardoso
ISE, University of the Algarve
Campus da Penha
Faro, Portugal
pcardoso@ualg.pt

Pedro Rodrigues
ISE, University of the Algarve
Campus da Penha
Faro, Portugal
pgrdrigues@ualg.pt

J.M.F. Rodrigues
ISE and Vision laboratory, LARSyS,
University of the Algarve
Faro, Portugal
jrodrig@ualg.pt

Domingos Paciência
University of the Algarve
Faro, Portugal

ABSTRACT

The use of Information Technologies (IT) in high competition sports is an instrument often used. However, the majority of the performers, including technical teams, do not have skills to program those IT tools to their needs. In this paper we present a visual programming language that allows the user without programming expertise to produce complex programs by drawing them on a web application, and we illustrated the development of this visual programming tool by applying it to situations from a soccer game. Although, this tool can be applied to other collective ball sports.

Categories and Subject Descriptors

D.1.7 [Visual Programming]

General Terms

Algorithms and Languages.

Keywords

Visual programming language, sports analysis.

1. INTRODUCTION

Aiming for excellence is the keyword to everyone involved in soccer. While players need to show their physical and tactical skills on the pitch, coaches and their remaining staff need to have their own tools so that they can perform at higher levels. Granting to all the parts a multi-functional information system (IS), with the objective of minimizing the adverse effects from the most critical and sensitive points of soccer is then critical. A match analysis, for example, can generate a huge amount of statistics and raw data. Consequently, it is very important to have a way to process that data, providing only the most important information to the coach, as soon as possible. Furthermore, a manual match analysis of soccer games represents a huge amount of work, which whenever possible should be an automatic process.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Conference '10, Month 1–2, 2010, City, State, Country.
Copyright 2010 ACM 1-58113-000-0/00/0010 ...\$15.00.

Footdata [1] is a project in development by Inesting, S.A., the University of the Algarve and the soccer coach Domingos Paciência. The goal is to build a web application product for soccer, which integrates two fundamental components of this sport's world: i) a social network (FootData-Social), which provides the typical features adapted to the soccer reality, and ii) the professional component (FootData-PRO), which can be considered as a Soccer Resource Planning (SRP), featuring a system for acquisition and processing of information to meet all the soccer management needs. The latter includes (between other things) an automated platform to gather information from the teams. This platform will be based on a system that will process live images acquired on-site, using a single or a group of cameras placed together in the stands, or images gathered from a Full HD Handycam. One of the main objectives is to automatically collect information and on-the-fly alert the technical staff about specific events. All the above should be presented in a web (browser) environment, accessible from a personal computer or a mobile device (smartphone or tablet).

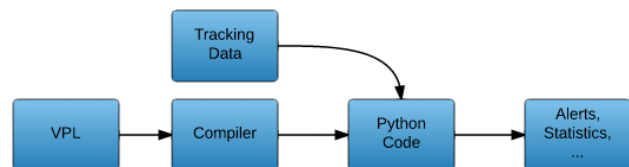


Figure 1 -- Architecture of the system.

Each technical staff has sometimes statistics that they would like to be treated and analyses but, in the majority of the cases, they do not have the programming skills to implement software capable of such. In order to detect the different statistics from the different coaches, we had to develop a tool that allow us to pass from a visual component to a programming (code) component, i.e., a visual programming language (VPL) framework for soccer (see Figure 1). The main feature consists in the design of an Interface Development Environment (IDE) and a compiler to transform the sketch into a language understandable by computers.

In more detail, a VPL is any programming language that allows users to create programs by using graphical elements rather than specifying them textually. Many VPLs are based on the idea of "boxes and arrows," where boxes or other screen objects are treated as entities, connected by arrows, lines or arcs which

represent relations. Usually, VPLs are used in the context where the programmer has limited programming knowledge.

For example the Ktechlab [2] is a VPL for microcontrollers and electronics that uses flowcharts to program microcontrollers graphically. Another VPL example is Scratch [3] [4] which uses an intuitive interface with blocks shaped to fit together only in ways that make syntactic sense, allowing to program various actions easily without having programming knowledge. Because of this features, Scratch is successfully being used to teach new programmers and for educational propose. Another similar web-based VPL application is a google project called Blockly [5], which allows making short programs in a web environment by dragging and dropping programming blocks to interact with virtual objects on the computer screen.

Even for more advanced programmers, there are VPLs. For example, Simulink® is a block diagram environment for multi-domain simulation and Model-Based Design integrated with Matlab®. Simulink supports system-level design, simulation, automatic code generation, and continuous test and verification of embedded systems. With some similarity we have the Scicos [6], which is a graphical dynamical system modeler and simulator. With Scicos the user can create block diagrams to model and simulate the dynamics of hybrid dynamical systems and compile models into executable code. Scicos is used for signal processing, systems control, queuing systems, and to study physical and biological systems. Other examples of Visual processing Languages applied to specific situations are e.g. [7] [8] [9].

In this paper we describe a system that aims at creating an interface which allows coaches to “program” their team’s statistics using a web-browser VPL environment. Satisfying the prerequisites, and based on the transmitted experience of the consultant technical teams, the interface is prepared to be intuitive and easy to use, providing the necessary tools for the coaches to describe their team performance. Besides, the developed application is a web application, to run on the majority of the most common web browsers (e.g., Firefox, Chrome, Safari and Internet Explorer). The application includes tools to represent several aspects of soccer, namely areas, movements from players and ball, and passes. In this sense, there are obviously a large number of scenarios where VPL are useful. This paper main focus isn’t the design itself (i.e., the IDE building) but passing from the visual representation to code, and further testing in soccer game environment. Our main contribution in this paper is to propose VPL that can be applied to most collective ball sports.

The remaining document is structured as follows. Section 2 presents our soccer visual programming language. Section 3 and 4 show how the VPL is compiled and executed and Section 5 presents some experimental tests. Finally, we conclude the paper with a brief discussion and the presentation of some future work in Section 6.

2. FOOTDATA’S VISUAL PROGRAMMING LANGUAGE

The FootData’s interface development environment is called Field Editor. On the front end, the Field Editor is a visual programming web-based tool created using HTML 5, CSS, and Javascript. As intended, the Field Editor allows the coaches to compute his statistics. The features already implemented in the Field Editor include tools to draw and edit the configurations of players, ball, areas and distances. The distance tool can be used to set the distance between players, between player and other objects (e.g.,

areas, ball, line or points in the field), between objects or to measure the distance between midpoint of two players to any other object. On the other hand, areas can be any type of polygon or ellipse, and can be used to detect if a player or the ball is inside that area. It also includes areas inside areas allowing, for instance, to draw situation were two players can move inside an area, which can also move inside a larger area, limiting the distance between them.

Figure 2 shows an example of a situation drawn in the Field Editor. The figure contains four players (represented by the red circles), a ball, three distances lines between the players and two zones. The objective of this drawing is to verify that whenever the ball is inside the green zone, the players should be inside the red zone keeping a distance of 12 meters between them. We should

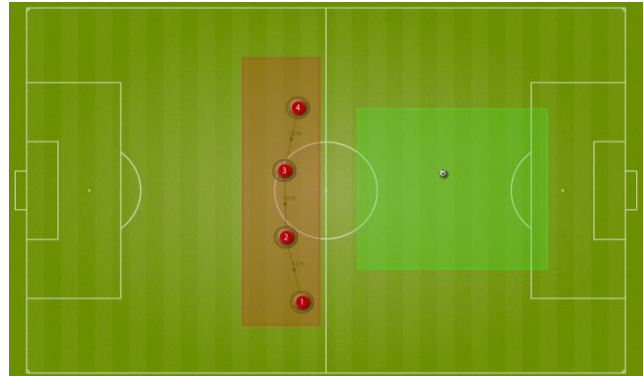


Figure 2 -- Example of a draw made on the Field Editor representing four players that should be at the red zone when the ball is inside the green one.

notice that although it is not shown in the picture, the distance between the players admits an error of 3 meters, value configurable in the Field Editor interface.

The main focus of this paper is not to explore the Field editor interface. So, for more details about the Field Editor interface please refer to [10].

When the user completes the draw of the desired statistics, typically the coaches or some other member of the technical team, this has to be passed to the FootData datacenter so that the depicted statistics are committed to the database (allowing posterior load and re-edition), and transformed into code which is then matched against the acquired/tracked images.

Also outside the scope of this paper is the tracking framework, developed inside the FootData project. In resume the tracking framework is composed by a set application capable of doing the tracking of the players and ball, given images of the full size pitch. Please refer to [1] for further details.

3. SERIALIZATION OF THE DRAWN SCHEMAS

Returning to the communication between the Field Editor (web application) and the FootData datacenter, a serialization process was thought tacking into consideration the fact that the Field Editor is implemented over JavaScript and the FootData’s libraries used for the verification of the statistics were programmed in Python. In this case, the drawn is converted to JSON (JavaScript Object Notation) documents [11], which is a lightweight data-interchange format. Using the JSON type of format allowed us to take advantage of (1) the JavaScript facilities

```

{
  "name": "schema 1",
  "description": "BP",
  "type": "DP",
  "field_dimension": [105, 68],
  "field_scale": [0.1, 0.1],
  "players": [
    {"id": 1, "number": 1, "team": "A",
     "points": [514, 559], "radius": 3},
    {"id": 2, "number": 2, "team": "A",
     "points": [486, 438], "radius": 3},
    {"id": 3, "number": 3, "team": "A",
     "points": [483, 314], "radius": 3},
    {"id": 4, "number": 4, "team": "A",
     "points": [507, 197], "radius": 3}
  ],
  "ball": {"id": 5, "points": [762, 320],
           "owner": "None"},
  "zones": [
    {"id": 7, "name": "Zone 7",
     "points": [[945, 498], [945, 198],
                [611, 198], [611, 498]]},
    {"id": 6, "name": "Zone 6",
     "points": [[546, 603], [546, 103],
                [409, 103], [409, 603]]}
  ],
  "lines": [ {"id": 10, "objects": [3, 4]},
             {"id": 9, "objects": [2, 3]},
             {"id": 8, "objects": [1, 2]}
  ],
  "distances": [
    {"objects": [3,4], "distance": 12,
     "margin": 3},
    {"objects": [2, 3], "distance": 12,
     "margin": 3},
    {"objects": [1, 2], "distance": 12,
     "margin": 3}
  ],
  "conditions": [
    [
      {"verify_location": [1, "IN", 6]},
      {"verify_location": [2, "IN", 6]},
      {"verify_location": [3, "IN", 6]},
      {"verify_location": [4, "IN", 6]},
      {"verify_location": [5, "IN", 7]},
      {"verify_distances": true}
    ],
    [{"alert": "True", msg: "Pressing the ball"}],
    [{"alert": "False", msg: ""}]
  ]
}

```

Figure 3 -- Example of JSON document generated at the FE.

to treat JSON documents, (2) the fact that those documents are very similar to Python dictionaries avoiding intricate parsing procedures, and (3) the fact that the database used to store the

document is MongoDB [12]. MongoDB is a cross-platform document-oriented database system, classified as a noSQL database, which eschews the traditional table-based relational database structure in favor of JSON-like documents.

Figure 3Figure 4 shows an example of the JSON document generated by the Field Editor to translate the drawing depicted in Figure 2. The JSON document has a predefined structure, that includes the name of the scheme, the type, the field dimensions, the field scale, the players, the ball, the lines, the distances and a list of conditions.

In more detail, the *name* attribute refers to the name given by the coach to the drawn schema and the *type* is used to know in which type of statistics is inserted. The attributes *field_dimension* and *field_scale* have also to be passed, because the soccer pitch image used in the field editor has been designed to a normalized field dimension, which in general will not match the dimensions of the majority of the stadiums, who have slightly varying sizes. Knowing the field dimensions where the game will occur and drawn the statistical schema to be detected, it is then possible to adjust the tactic to a particular stadium.

The *players* attribute has an array that takes an embed JSON object, a subdocument, for each player inserted in the field editor, following the structure: {"id": 1, "number": 1, "team": "A", "point": [514, 559]}. As we will see, a different *id* is attributed to each object inserted in the Field Editor, so we can refer to any object by its *id*. The subdocument has also the player's *team*, its *number*, and *point* which are the player's coordinates in the pitch and radius that is a value in meters in which the player position can vary.

When inserting the distances between two players, a line object is created containing an *id* and the ids of the *objects* where the distance line starts and where it ends. This results in the following structure: {"id": 8, "objects": [1, 2]} which indicates that there is a line between player with *id* 1 and 2. The distances information is then placed in another attribute called *distances*, which contains an array with objects having the information about the *objects* ids, and the *distance* and a (error) *margin* in meters. The object with that information has the following structure: {"objects": [3, 4], "distance": 12, "margin": 3} which, for the presented case, says that objects with *id* 3 and 4 (corresponding to the player with *id* 3 and 4, respectively) should maintain a distance of 12 meters, allowing an error of 3 meters.

As already mentioned the *zones* inserted in the Field Editor can be elliptical, rectangular or free draws. Instead of having a label in the JSON document for each those geometric objects, we consider them all as polygons (the elliptical objects are approximated through the computation of a certain number of points so that the representation as a polygon is as close as the original figure). Those points (vertices), that represent coordinates in the soccer field, are then inserted into a *zones* array in the JSON document. As before, each zone as also a unique *id* and a representation that follows the structure {"id": 6, "name": "Zone 6", "points": [[546, 603], [546, 103], [409, 103], [409, 603]]}.

Having all this information gathered, we added one last attribute to the JSON document to allow the program to know what is to be detected, that attribute is called *conditions*. As the name suggest, in this field it is possible to create conditions, more specifically, *if-then-else statements*. To do this we defined a structure with an array with 3 sub-arrays: the first one is the condition of the *if*, the second one is *then* statement and the last one is what to do if the condition is false (i.e., the *else* statement).

```

from lib import gpm
def run():
    result = False
    if pm.verify_location(frame,['ball','IN',zones7]):
        if gpm.verify_location(frame,
                                [players2,'IN',zones6]):
            if gpm.verify_location(frame,
                                    [players3,'IN',zones6]):
                if gpm.verify_location(frame,
                                        [players4,'IN',zones6]):
                    if gpm.verify_location(frame,
                                            [players1,'IN',zones6]):
                        if gpm.verify_distances(frame,
                                                distances) :
                            gpm.sendmsg('Pressing the ball')
                            gpm.sendalert('True')
                            result = True
                        else:
                            gpm.sendmsg('')
                            gpm.sendalert('False')
                            result = False
    return result, gpm

```

Figure 4 - Python code generated from the JSON document.

Some combinations providing more complex programs are also possible. For example: (1) it is possible to program chained *if-then-else* statements, by adding inside the *then* or *else* statements others *if-then-else* statement, i.e., including more “three element arrays” that follow the previous presented structure. Furthermore, it is also possible to (2) program conjunctions (“*and*”s) and (3) disjunction (“*or*”s). The conjunctions are represented as single conditions in a JSON document, as for example: [{"*verify_location*": [1, "IN", 6]}, {"*verify_location*": [2, "IN", 6]}]. For the previous case, the results in true if “*verify_location*”: [1, "IN", 6] and “*verify_location*”: [2, "IN", 6] are both true. Disjunctions are made by inserting various conditions in the same JSON object, e.g., [{"*verify_location*": [1, "IN", 6], "*verify_distances*": true}] results in true if at least one of the conditions is true. As expected it’s also possible to combine both “*and*”s and “*or*”s, e.g: [{"*condition1*, *condition2*}, {"*condition3*, *condition4*}], which results in the (*condition1* OR *condition2*) AND (*condition3* OR *condition4*) logical expression.

The conditions have their own structure. For example, *verify_location* will call a function, implemented in the already mentioned Python library, which verifies if a certain object is inside or outside a zone. In this particular case, this is done by passing a three element array which (1) receives de object *id* to be verified, (2) the condition “*IN*” or “*OUT*” and (3) the zone *id*. The *verify_distances* has a similar behavior, i.e., it calls a function which returns a boolean value upon the verification of the distances defined in the JSON document (see Figure 3).

4. CONVERTING THE SERIALIZED DOCUMENT TO PYTHON

Having the serialization as JSON structure it is now necessary to pass it to Python code. First the variables corresponding to the players, distances, lines and zones objects (everything except the

conditions) are instantiated from the pre-implemented Python classes. These Python instances are saved to a file using a serializer, so that the objects will be simply loaded avoiding their creation every time the verification of the schema is run. This process is also justified by the fact that data from the Field Editor is saved in a MongoDB and having the files with the objects already instantiated will speed up the initialization process. The next step is to convert the *conditions* to Python code, which is done by accessing the *conditions* array, and creating the *if-then-else* code from the sent statements. In more detail, after processing the *if* condition each *then* and *else* statements are checked to see if there is another *if-then-else* arrays inside. In this last case, of nested conditions, the arrays are accessed recursively. Then each condition is examined looking for “*and*” and/or “*or*” statements, by checking the structures explained previously. The conditions names are matched to their correspondent methods already implemented in our Python library. For example the matching method to the “*verify_location*”: [5, "IN", 7]” in the JSON document (Figure 3) corresponds to the function *verify_location(frame, ['ball', 'IN', zones7])* in our Python library which receives a *frame* from the tracking system and checks if the ball is inside the region defined as *zone7*.

While the conditions are processed the Python file with the *if else statements* is created on the fly using our Python generator library. Figure 4 shows the Python code which corresponds to the JSON document presented in Figure 3.

Now that we have the code generated, information about the players and the ball position is needed in order to test in game environment. As already mentioned, this information is obtained using a tracking system [13] that returns, for each frame, the players’ and ball’s positions in meters from the top left corner of the pitch using an homography [14] to place the objects (players and ball) in the model field. The frame’s data follows the structure: {"*frame_id*": t, "*teamA*": {"*player_1*": (x₁, y₁), "*player_2*": (x₂, y₂), ..., "*teamB*": {...}, "*ball*": (x,y)}.

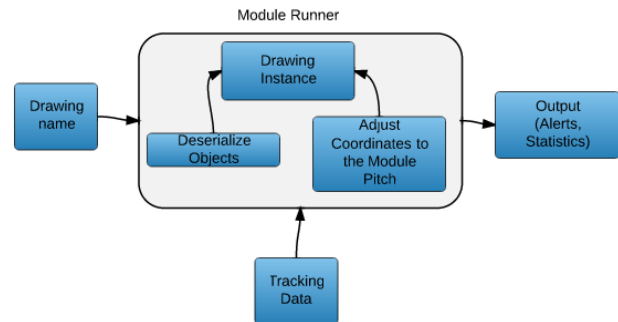


Figure 5 - Architecture used to run the game schemas.

The module *gpm* imported in Figure 4 has a set of pre-implemented methods used to analyze the game, and also records the information gathered by each method that is executed. To run the code, first the file name is passed by parameter to a module that is responsible to import the previously generated Python file (Figure 4), and the previously serialized objects. The objects are deserialized and loaded in to the already imported module with the code from the game schema using the *setattr* built-in function from Python.

```

'verify_distances': [
{
  'players':(<Player object>,<Player object>),
  'False': None,
  'distance_def': 12,
  'distance': 8.04
},
{
  'players':(<Player object>,<Player object>),
  'True': None,
  'distance_def': 12,
  'distance': 9.95
},
{
  'players':(<Player object>,<Player object>),
  'True': None,
  'distance_def': 12,
  'distance': 11.65
}
]

```

Figure 6 - Output from *verify_distances* function.

As mention before, it is possible that the field size in the field editor is different than the field size from the game we are analysing, so the coordinates from the field editor are adjusted to the new pitch size. Since we have the pitch size used in the field editor, the pitch size from the game and the coordinates in the field editor, this is a direct calculation where it's only necessary to do a cross-multiplication to adjust the coordinates from the field editor to the new pitch size. The code can now be run for each frame or for a set of n of frames. At the end a Python dictionary is returned with the result for every frame analyzed. Figure 5 resumes the architecture explained above.

In our example, sketched in Figure 4, first is verified if the ball is in the green zone. If the ball is in the green zone then it is verified individually if the players are in the red zone and at last if the distances are correct. Those verifications are made individually because if one fails (returns false), the next verifications are no longer made and processing resources are spared. For each frame the verifications result are stored, enabling us to create statistics for what or who is failing in this schema. Figure 6 shows an example of the output obtained from the *verify_distances* method, which is used to verify the distances between players. The output is composed by an array with a Python dictionary for each distance verified. The dictionary has the players' objects, the defined distance, the distance between the objects in that frame and a key with the name *True* or *False* for easily verify if the distance is correct or not.

5. COMPUTATIONAL TESTS

In this section we present some tests and results from the application of the visual programming language. The first test is a simple example of a player inside a zone (Figure 7). Figure 8 shows the corresponding JSON document and the Python code obtained after compiling is presented in Figure 9. The second test analyzes the results obtained from the example in Figure 2, explained along the previous sections of this paper. The data used

to analyze the soccer game is returned by the tracking software and has a length of 2.30 minutes.

With the output from each method is possible to statistically analyze how a player performed in the game and analyze many other possibilities with the data acquired.

5.1 Test 1

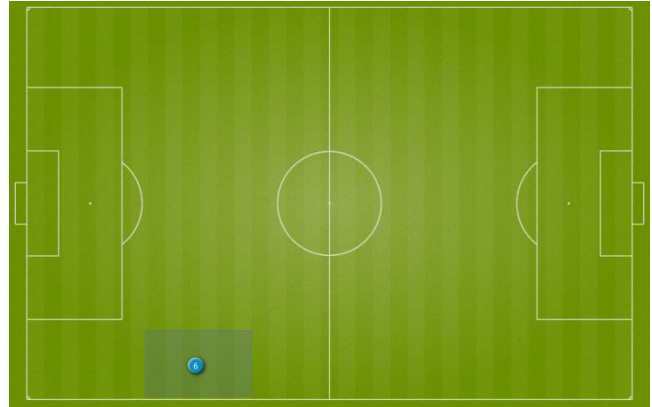


Figure 7 – Situation drawn in the FE, player inside a zone.

In this example is shown a simple application for the feature “player inside zone” (Figure 7), in which will be verified if player number ‘6’, from team ‘B’, is inside that zone. Figure 8, represents the JSON document generated from de FE, and Figure 9 the compiled Python code.

```

{
  "name": "Test1",
  "description": "",
  "type": "PO",
  "field_dimension": [105, 68 ],
  "field_scale": [0.1, 0.099 ],
  "players": [{
    "id": 2,
    "number": 6,
    "team": "B",
    "points": [324, 633 ]
  }],
  "ball": {},
  "zones": [{
    "id": 1,
    "name": "Zona 1",
    "points": [[421,686],[421,571],[236,571],
              [236,686]]
  }],
  "conditions": [
    [{"verify_location": [2, "IN", 1]} ],
    [{"alert": "True", "msg": "Event Y"}],
    [{"alert": "False"} ]
  ]
}

```

Figure 8 - JSON document for Test1 example (Figure 7).

Two examples of the output for this situation will be `{'verify_location': {False: None, 'first_arg': 'Number: 6, 'Team': 'B', 'second_arg': <Zone object >, 'condition': 'IN'}, 't': 258}` if the

player is outside the defined zone; and {'verify_location': {True: None, 'first_arg': 'Number: 6, 'Team': B ', 'second_arg': <Zone object >, 'condition': 'IN'}, 't': 538} if the player is inside the zone.

In this case, what change between the two outputs is the existence of a dictionary key with the name *True* or *False* to validate or not the situation.

```

from lib import gpm
def run():
    result = False
    if gpm.verify_location(frame,[players2,'IN',zones1]):
        gpm.sendmsg('Event Y')
        gpm.sendalert('True')
        result = True
    else:
        gpm.sendalert('False')
        result = False
    return result, gpm

```

Figure 9 – Compiled Python code from Test1 example (Figure 7).

A simple analysis of the output can be the counting of how long the player stayed in the area. In our game example of 2.3 minutes, the player was 39,36% of the time in that zone.

5.2 Test 2

In this section we present the results obtained from the analysis of the example shown in the Figure 2. Figure 10 graphically shows the data for frame 947 of a soccer game, returned by the tracking software, along with the zones and distances between players in the pitch. The image is a screenshot from a development tool used for the visualization of what is being processed.

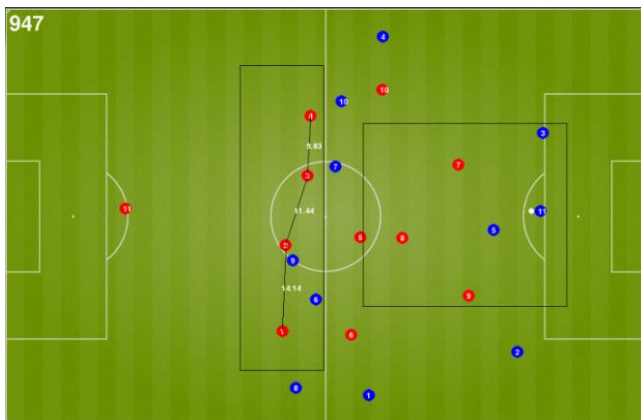


Figure 10 – Visual scheme being processed.

As mentioned before, for each frame we will have a Python dictionary with the information gathered by the methods executed to detect if the “drawing” is being accomplished.

Figure 11 shows the output for frame 947, in which is presented the result from the *verify_location* and *verify_distances*. Each time a method is run, the result is recorded in the dictionary key with the same name as the method, and if the same method is run more

```

{'verify_location': [
    {'True': None, 'first_arg': 'ball',
     'second_arg': Zones7,'condition': 'IN' },
    {'True': None, 'first_arg': 'Number: 2, Team: A '
    ,
     'second_arg': Zones6, 'condition': 'IN'},
    {True: None, 'first_arg': 'Number: 3, Team: A ',
     'second_arg': Zones6, 'condition': 'IN'},
    {True: None, 'first_arg': 'Number: 4, Team: A ',
     'second_arg': Zones6, 'condition': 'IN'},
    {True: None, 'first_arg': 'Number: 1, Team: A ',
     'second_arg': Zones6, 'condition': 'IN'}
],
't': 947,
'verify_distances': [
    {'players': ('Number: 3, Team: A ', 'Number: 4,
    Team: A '),
     'True': None,
     'distance_def': 12,
     'distance': 9.83
    },
    {'players': ('Number: 2, Team: A ', 'Number: 3,
    Team: A '),
     'True': None,
     'distance_def': 12,
     'distance': 11.44
    },
    {'players': ('Number: 1, Team: A ', 'Number: 2,
    Team: A '),
     'True': None,
     'distance_def': 12,
     'distance': 14.14
    }
]
}

```

Figure 11 – Output from frame 947.

than once in the same frame, the results are recorded inside an array, as shown in Figure 11. In *verify_location* each array element contains the result from one object, in our case the ball and four players, and as each element has a key with the name *True*, that means that the players and the ball are inside their predefined zone. The *verify_distances* follows the same principle, having one element of the array for each distance verified.

As all the players are complying the distances between them, being inside the predefined zone when the ball is at the other zone, the output from Figure 11 confirms that they are doing what was drawn in the FE.

6. CONCLUSIONS

This paper presented our Visual Programming Language for soccer statistics analysis, built to help the coaches and technical teams in the improvement of their players and team’s performance by allowing them to draw his team wonted statistics and verifying if the players are complying with them. This VPL can also be applied to other collective ball sports.

This web environment tool allows, without previous programming knowledge, to make fairly complex analysis just by using some of the implemented features. The tool analyses the pretended drawings for each frame, giving the chance to the coach to analyze thoroughly a great number of game aspect that previously would require traditional programming skills or exhaustive man work. Using this structure is also possible to receive alerts to a mobile device during the game if some particular event happens or not, making it possible to send real time adjustments to the team, in order to accomplish the pretended performance.

It is also important to refer that this tool can be also used to study the opposite team statistics after or during a game. This gives the coaches an opportunity to be better prepared when playing against other teams, or once more, if used during a game, to be notified in the bench and adjust how the team is performing. This tool can become a game changer for the ones who use it bringing soccer to a whole new level.

For future work, we plan to continue to develop our visual programming language by implementing more features. Improve our game statistics output, and since we have a huge quantity of data it is possible to implement some data mining algorithm to detect team patterns or some other interesting patterns.

7. ACKNOWLEDGMENTS

This work was supported by FCT project PEst-OE/EEI/LA0009/2013 and project FootData QREN I&DT, n.º 23119. We also thanks to project leader Inesting, S.A. [www.inesting.com], and the consultant soccer coach Domingos Paciência and our colleague Carlos Gomes.

8. REFERENCES

- [1] P. Rodrigues, A. Belguinha, C. Gomes, P. Cardoso, T. Vilas, R. Mestre and J. M. F. Rodrigues, "Open Source Technologies Involved in Constructing a Web-Based Football Information System," in *In Á. Rocha, A. M. Correia, T. Wilson, & K. A. Stroetmann (Eds.), Advances in Information Systems and Technologies SE - 66, vol. 206, pp. 715–723*, Springer Berlin Heidelberg, 2013.
- [2] "ktechlab," [Online]. Available: <https://github.com/ktechlab>. [Accessed 04 04 2014].
- [3] "scratch," [Online]. Available: <http://scratch.mit.edu/>. [Accessed 04 04 2014].
- [4] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman and Y. Kafai, "Scratch: programming for all," *Commun. ACM*, vol. 52, no. 11, pp. 60-67, 11 2009.
- [5] "blockly," [Online]. Available: <http://code.google.com/p/blockly>. [Accessed 04 04 2014].
- [6] "scicos," [Online]. Available: <http://www.scicos.org>. [Accessed 04 04 2014].
- [7] Z. Dobesova, "Visual programming language in geographic information systems," in *2nd Int. Conf. on Applied informatics and computing theory. World Scientific and Engineering Academy and Society (WSEAS)*, 2011.
- [8] E. Marchiori et al., "A visual language for the creation of narrative educational games," *Journal of Visual Languages & Computing*, vol. 22, pp. 443-452, 2011.
- [9] G. Tekli, R. Chbeir and F. Jacques, "A visual programming language for XML manipulation.," *Journal of Visual Languages & Computing*, vol. 24, pp. 110-135, 2013.
- [10] P. Rodrigues, P. Cardoso and J. Rodrigues, "A Field, Tracking and Video Editor Tool for a Football Resource Planner," in *In Proc. 8th Iberian Conf. on Information Systems and Technologies*, Lisbon, Portugal, 2013.
- [11] "JSON," [Online]. Available: <http://www.json.org>. [Accessed 04 04 2014].
- [12] "MongoDB," [Online]. Available: <https://www.mongodb.org>. [Accessed 04 04 2014].
- [13] J. Rodrigues, P. Cardoso, T. Vilas, S. Bruno, P. Rodrigues, A. Belguinha and C. Gomes, "A computer vision based web application for tracking soccer players," in *accepted for 16th Int. Conf. on Human-Computer Interaction*, Crete, Greece., 2014.
- [14] N. Sebe and M. Lew, "Robust computer vision: Theory and applications," The Netherlands: Kluwer Academic Publishers, 2003.