Tracking Data Documentation

Lewis Higgins

lewis.higgins@postgrad.manchester.ac.uk

September 2019

Contents

1 Introduction

Tracking data comes from different suppliers depending on which competition the match is for. The raw data comes in different file formats which is a pain for analysis. The solution is to convert the tracking data from different suppliers to a common file format. It is also worth optimising this common file format for analysis to further improve performance.

A common feature of all the suppliers' formats is that all the tracking data is stored in one plain-text file. This

means the whole file must be read even if only part of the data is needed (e.g. only the ball). Moreover, the usual benefit of plain-text is that the files are human-readable. However, there is little to gain from reading a tracking file by eye so this is not a requirement.
The proposed format aims to improve file-read performance by storing the tracking data for the ball, and each team in separate files. In addition, the files are stored in binary format which both reduces the overall size of the files and makes them quicker to load into memory.

2 MessagePack (MsgPack)

2.1 JSON

JSON (JavaScript Object Notation) is a plain-text file format which can be used to store data objects comprised of arrays, maps (an example of a map is a Python `dict`), strings, integers, bools and floats. This is handy to store unusually shaped data structures with more than one data type.
Example (`https://json.org/example.html`)

```
{
    "menu": {
        "id": "file",
        "value": "File",
        "popup": {
            "menuitem": [
                {"value": "New", "onclick": "CreateNewDoc()"},
                {"value": "Open", "onclick": "OpenDoc()"},
                {"value": "Close", "onclick": "CloseDoc()"}
            ]
        }
    }
}
```

2.2 MsgPack

MessagePack[1] behaves like JSON but is stored in binary rather than plain-text. This means the files are smaller and quicker to read. Due to the similar structure, it is simple to describe MsgPack files with JSON representations.

3 Ball File

The ball file is saved as `${MATCH_ID}.BALL.msgpack`.
The file is structured as an array of `Ball` objects.

3.1 `Ball` Object (JSON representation)

The Ball object contains all the information about the ball in an array. The order of the data must be specified:

[1] `https://msgpack.org/`

```
[
    ${FRAME_ID},         # [32-bit int]
    ${OBJECT_POS_X},     # [16-bit int] relative to pitch centre (in cm)
    ${OBJECT_POS_Y},     # [16-bit int] relative to pitch centre (in cm)
    ${OBJECT_POS_Z},     # [16-bit int] relative to pitch centre (in cm)
    ${ALIVE},            # [bool]
    ${OWNING_TEAM},      # [char]
    ${OWNING_PLAYER_ID} # [32-bit uint]
]
```

4 Team File

The team file is saved as `${MATCH_ID}.${TEAM}.msgpack` with `TEAM` being `HOME`, `AWAY`, or `OFFICIALS` (if present). The file is structure as an array of `Team` objects.

4.1 Team Object (JSON representation)

The `Team` object contains the current Frame ID, whether the team owns the ball, and an array of players:

```
[
    ${FRAME_ID},         # [32-bit int]
    ${BALL_OWNED},       # [bool]
    ${PLAYERS_IN_TEAM}   # [Player array]
]
```

4.2 Player Object (JSON representation)

The `Player` object contains the current position, as well as their team, ID, and shirt number:

```
[
    ${TEAM},             # [char]
    ${PLAYER_ID}         # [32-bit uint]
    ${PLAYER_SHIRT_NUM} # [8-bit uint]
    ${OBJECT_POS_X},     # [16-bit int] relative to pitch centre (in cm)
    ${OBJECT_POS_Y},     # [16-bit int] relative to pitch centre (in cm)
    ${BALL_OWNED},       # [bool]
]
```

5 APIs

5.1 C++

Programs written and compiled in C++ will run much quicker when analysing large datasets like the tracking data. The C++ API is the main focus for analysing data.
The API will load a GamePack from a source directory and provides a set of classes and methods to access the loaded data.

5.1.1 `Football::Match`

`Football::Match` is external structure that contains all the tracking data for an entire game. `Football::Match` contains a `Football::Ball` object and two `Football::Team` objects for each frame of data. These are each stored in a `std::vector`: `Football::Match::BALL_FRAMES`, `Football::Match::HOME_FRAMES`, and `Football::Match::`
There is also an optional `std::vector` (`Football::Match::OFFICIALS_FRAMES`) for when the officials are tracked.
Outlined below are the `public` access member variables and functions of `Football::Match`.

```
std::uint32_t    FRAME_ID;
Ball             BALL;
Team             HOMETEAM;
Team             AWAYTEAM;
```

The `public` member-variables of `Football::Match`.

```
std::uint32_t number_of_frames() const {...}
```

Get the number of frames in the match.

```
Frame get_frame (std::uint32_t idx) const {...}
```

Gets a specific `Football::Frame`. The index argument refers to position in `std::vector` not Frame ID.

```
void reduce_to_5fps() {...}
```

Reduces a 25 FPS `Football::Match` to a 5 FPS `Football::Match`.

```
void remove_dead_frames(bool verbose = false) {...}
```

Removes all frames where the `Football::Ball` is not marked alive.

```
void mirror_alternate_periods () {...}
```

Rotates the pitch coordinates every-other period to stop the teams swapping sides at the beginning of each period.

```
void resetFrameIDs() {...}
```

Translates all frameIDs such that the first frame has frameID = 0.

```
void loadFromFile(std::string _data_dir, std::uint32_t _match_id, bool fps5 = true) {...}
```

Loads a full match from a given path into this `Football::Match` object. If fps5 option is true then the `5fps/` sub-directory is used to load data. Returns false if there was a problem loading the match.

```
template<typename T>
    static bool load_subfile(std::string path, T& store, bool required = true) {...}
```

This function can be used to load just one `MsgPack` file instead of a full game. e.g. to load just the ball data, prepare a `std::vector<Football::Ball>` to store the ball data then call this function passing `std::vector<Football::Ball>` as the store argument. It loads the `MsgPack` file from `path` and stores the data in `T store`. While `T` is a template it will only work with structures that have proper MsgPack definitions. Returns false if there was a problem loading the subfile. If required is true, a std::exception is thrown instead (will improve this one day).

```
static bool getMatchFromFile(Match& storage_match, std::string _data_dir, std::uint32_t
    _match_id, bool fps5 = true) {...}
```

Loads a full match from the given path into the `Football::Match` object passed as storage. If fps5 option is true then the `5fps/` sub-directory is used to load data. Returns false if there was a problem loading the match.

5.1.2 `Football::Frame`

A `Football::Frame` is a structure used to store a `Football::Ball`, and two `Football::Team` for an instant of the match. `Football::Match` does not store the data in a `std::vector` of `Football::Frame`, as storing them in separate `std::vector` provides the option to access them individually. `Football::Frame` is a `struct` so all members are `public` access.

```
Frame(std::uint32_t _frame_id, const Ball & _b = Ball(), const Team & _ht = Team(), const
    Team & _at = Team()) {...}
```

Parameterised constructor.

```
std::uint32_t    FRAME_ID;
Ball             BALL;
Team             HOMETEAM;
Team             AWAYTEAM;
```

The member variables of `Football::Frame`.

```
bool     isAlive() const {...}
```

Checks whether the ball is marked alive.

5.1.3 `Football::Ball`

This object contains the position of the ball, along with some extra information.

```
Ball(std::uint32_t frame_id) {...}
Ball(std::int16_t x, std::int16_t y, std::uint32_t frame_id = 0) {...}
Ball(std::pair<std::int16_t, std::int16_t> p) {...}
Ball(const Ball& b, std::uint32_t frame_id) {...}
```

Parameterised constructors.

```
std::uint16_t   get_posX() const {...}
void            set_posX(const std::uint16_t _x) {...}
std::uint16_t   get_posY() const {...}
void            set_posY(const std::uint16_t _y) {...}
std::uint16_t   get_posZ() const {...}
void            set_posZ(const std::uint16_t _z) {...}
std::array<std::int16_t, 3> get_pos () const {...}
void            set_pos (const std::array<std::int16_t, 3> & _pos) {...}
```

Getters and Setters for ball position.

```
bool            is_alive() const {...}
void            set_alive(const bool  _alive) {...}
```

Checks whether the ball is marked alive.

```
char            get_owningTeam() const {...}
void            set_owningTeam(const char _team) {...}
```

Single character representation of which team owns the ball. Accepted values: Home - `H`, Away - `A`, Undefined - `U`.

```
std::uint32_t   get_owningPlayerId()    const {...}
void            set_owningPlayerid(const std::uint32_t _player_id) {...}
```

The ID of the player that owns the ball (doesn't work currently).

```
std::uint32_t   get_frameId()           const {...}
void            set_frameId(const std::uint32_t _frame_id) {...}
```

The current frame ID.

5.1.4 `Football::Team`

Stores the Frame ID and a `std::vector` of `Football::Player`. Also stores whether this team is currently in possession.

```
Team(std::uint32_t frame_id) {...}
Team(std::vector<Player> plyrs, std::uint32_t frame_id, bool ball_owned = false) {...}
```

Parameterised constructors.

```
std::uint32_t    get_frameId()           const {...}
void             set_frameId(const std::uint32_t _frame_id) {...}
```

The current frame ID.

```
std::uint16_t number_of_players () const {...}
```

Counts the number of players in the team.

```
std::uint32_t       bool            get_ballOwned() const {...}
bool             ownsBalled()    const {...}
void             set_ballOwned(bool _ball_owned) {...}
```

If the team is in possession.

```
std::vector<Player>&    get_playersInTeam() {...}
void                    set_playersInTeam(const std::vector<Player>& _players_in_team)
    {...}
void                    add_player(const Player& _player) {...}
```

The `std::vector` that stores all the `Football::Player`.

```
Player&                 get_player(const std::uint16_t _player_array_index) {...}
void                    set_player(const std::uint16_t _player_array_index, const Player&
    _player) {...}
```

Access a specific `Football::Player` from the `std::vector`.

5.1.5 `Football::Player`

```
Player(std::int16_t x = 0.0, std::int16_t y = 0.0, std::uint8_t sn = 1) : PLAYER_SHIRT_NUM(
    sn) {...}
Player(std::pair<std::int16_t, std::int16_t> p, std::uint8_t sn = 1) : PLAYER_SHIRT_NUM(sn)
     {...}
```

Parameterised constructors.

```
std::uint16_t    get_posX() const {...}
void             set_posX(const std::uint16_t _x) {...}
std::uint16_t    get_posY() const {...}
void             set_posY(const std::uint16_t _y) {...}
std::array<std::int16_t, 2> get_pos () const {...}
void             set_pos (const std::array<std::int16_t, 2> & _pos) {...}
```

Getters and Setters for ball position.

```
std::uint8_t     get_shirtNumber() const {...}
void             set_shirtNumber(const std::uint8_t _sn) {...}
```

Shirt number of player.

```
char            get_team() const {...}
void            set_team(const char _team) {...}
```

Character representation of player's team. Accepted values are: Home - 'H', Away - 'A', Official - 'O', and Undefined - 'U'.

```
std::uint32_t   get_playerId() const {...}
void            set_playerId(const std::uint32_t _player_id) {...}
```

Opta Match ID of player.

```
bool            get_ballOwned() const {...}
bool            ownsBall()      const {...}
void            set_ballOwned(const bool _ball_owned) {...}
```

Whether this specific player owns the ball (not working right now).

Appendices

A API Examples

A.1 Download Link

https://github.com/hidgjens/ReadTrackingData

A.2 C++ Example

```
/*
    Lewis Higgins,
    City Football Group & The University of Manchester,
    September 2019

    E: lewis.higgins@postgrad.manchester.ac.uk

    W: https://github.com/hidgjens/ReadTrackingData

    Example for loading and analysing a GamePack.

    To build, please add "FOOTBALL/THIRDPARTY" to your include-dirs via the -I flag:
        g++ cpp_example.cpp -I"FOOTBALL/THIRDPARTY"

    Might also want to consider building a 64-bit binary using the -m64 flag.

*/

// Football.h will include the whole folder
#include "FOOTBALL/Football.h"

int main (int argc, char * argv[])
{
    // variables to locate game
    std::string     DATA_DIR    = "Data/";
    uint            MATCH_ID    = 1059714;
    bool            mode_5fps   = true;         // true for loading the 5fps version

    // Note that everything from the FOOTBALL folder is stored in namespace Football
    // Create match object
    Football::Match ex_match;

    // load game from file
    ex_match.loadFromFile(DATA_DIR, MATCH_ID, mode_5fps);

    // count the number of frames in possession
    uint        home_possession;
    uint        away_possession;
    uint        total_frames;       // only counting alive frames

    {

```

```
    // create frame object as temporary storage
    Football::Frame _frame;

    // iterate through match frames
    for (uint i = 0 ; i < ex_match.number_of_frames() ; i++ )
    {
        // store current frame in temporary storage
        _frame = ex_match.get_frame(i);

        /*
            Analysis for this frame.
        */

        // check if ball is alive in this frame
        if (_frame.isAlive())
        {
            // increment alive frames counter
            total_frames ++;

            // check who is in possession
            switch (_frame.BALL.get_owningTeam())
            {
            // home team
            case 'H':
                // increment home counter
                home_possession ++;
                break;
            // away team
            case 'A':
                // increment away counter
                away_possession ++;
                break;
            // officials
            case 'O':
                // officials in possession of the ball?
                std::cerr << "Frame " << i << " official possession?" << std::endl;
                // discount this frame
                total_frames --;
                break;
            // undefined
            case 'U':
                // undefined possession - unlikely to occur, but not necessarily an error
                std::cout << "Frame " << i << " undefined possesion" << std::endl;
                // discount this frame
                total_frames --;
            default:
            // default case is none of the above
                std::cerr << "Frame " << i << " Default case on switch" << std::endl;
                // discount this frame
                total_frames --;
                break;
            } /* end of switch */
        } /* endif ball alive */
```

```
    } /* for loop ends */

    } // the extra set of curly braces is limiting the scope of _frame, beyond here it is no longer in scope. Good practice as _frame was temporary storage for the loop and no longer needed

    // compute fraction of possession from extracted data
    float home_pos_frac = home_possession / ((float) total_frames); // explicitely casting one of these numbers to float to avoid integer result i.e. 1/2 = 0 vs 1/2.0 = 0.5
    float away_pos_frac = away_possession / ((float) total_frames);

    // print result to console
    printf ("\nHome team possession %4.1f%%, Away team possession %4.1f%%\n", home_pos_frac * 100.0, away_pos_frac * 100.0);


    // print starting player line-up

    printf ("\nInitial team line-ups:\n");

    // get the first frame
    auto first_frame = ex_match.get_frame(0);

    // get the Football::Team objects stored in the frame
    auto& initial_home_team = first_frame.HOMETEAM;
    auto& initial_away_team = first_frame.AWAYTEAM;

    printf ("\tHome Team\n");

    // iterate through the players in team
    for (const auto& player : initial_home_team.get_playersInTeam()) // currently, Football::Team is not iterable, but the std::vector\<Football::Player> contained within is
    {
        printf("\t\t%s\n", player.get_summaryString().c_str());
    }

    printf ("\tAway Team\n");
    // iterate through the away players
    for (const auto& player : initial_away_team.get_playersInTeam())
    {
        printf("\t\t%s\n", player.get_summaryString().c_str());
    }


    return EXIT_SUCCESS;
}

/*
    Goal:
    I want to implement the Football::Match object to be iterable, i.e.

    for (auto frame_ : match)
    {
```

```
        //  analyse frame_
    }

    likewise for Football::Team:

    for (auto& player : team)
    {
        //  analyse player
    }
*/
```

A.3 Python Example

```
'''
    Lewis Higgins,
    City Football Group & The University of Manchester,
    September 2019

    E: lewis.higgins@postgrad.manchester.ac.uk

    W: https://github.com/hidgjens/ReadTrackingData

    Example for loading and analysing a GamePack.

    tqdm is an optional module used in Football.py which provides progress bars while
    loading data. (install via python pip)
'''

# import Football.py to use it
import Football as FB

# variables used to locate and load game
data_dir        = "Data/"       #   relative path to data directory
match_id        = 1059714       #   integer MatchID, not "g1059714"
mode_5fps       = True          #   whether to load the 5FPS version

# load the match using the @staticmethod in Match class
loaded_match    = FB.Match.getMatchFromFile(data_dir, match_id, mode_5fps)

# count the number of frames in possession
home_possession = 0
away_possession = 0
total_frames    = 0 # only wish to count alive frames

# match object is iterable
for ball, home_team, away_team in loaded_match:
    '''
        Just to note, an alternative arrangment is to write:
            for frame in match:

            where frame is a tuple = (ball, home_team, away_team)
    '''

```

```
    # check if ball is alive
    if ball.alive:
        # count the frame
        total_frames += 1

        # check who is in possession
        if ball.owning_team == 'H':
            home_possession += 1
        elif ball.owning_team == 'A':
            away_possession += 1
        else:
            print('Undefined possession')

# compute results
home_fraction   = home_possession/total_frames
away_fraction   = away_possession/total_frames

print("\nHome team possession %.1f%%, Away team possession %.1f%%" % (home_fraction * 100,
    away_fraction * 100))

# print starting player lineups:
hometeam        = loaded_match.home_frames[0]   # FB.Team objects
awayteam        = loaded_match.away_frames[0]

print("\nInitial team line-ups:\n")
# Team objects are iterable
print("\tHome Team")
for player in hometeam:
    print('\t\t%s' % player)

print("\tAway Team")
for player in awayteam:
    print('\t\t%s' % player)
```