File Input and Output using XML and YAML files

Goal

You'll find answers for the following questions:

- How to print and read text entries to a file and OpenCV using YAML or XML files?
- How to do the same for OpenCV data structures?
- How to do this for your data structures?
- Usage of OpenCV data structures such as FileStorage, FileNode or FileNodeIterator.

Source code

You can download this from here or find it in the samples/cpp/tutorial_code/core/file_input_output/file_input_output.cpp of the OpenCV source code library.

Here's a sample code of how to achieve all the stuff enumerated at the goal list.

```
#include <opencv2/core/core.hpp>
#include <iostream>
#include <string>

using namespace cv;
using namespace std;

class MyData
{
public:
    MyData() : A(0), X(0), id()
    {}
    explicit MyData(int) : A(97), X(CV_PI), id("mydata1234") // explicit to avoid implicit conversion
    {}
    void write(FileStorage& fs) const                        //Write serialization for this class
    {
        fs << "{" << "A" << A << "X" << X << "id" << id << "}";
    }
    void read(const FileNode& node)                          //Read serialization for this class
    {
        A = (int)node["A"];
        X = (double)node["X"];
        id = (string)node["id"];
    }
public:   // Data Members
    int A;
    double X;
    string id;
};

//These write and read functions must be defined for the serialization in FileStorage to work
static void write(FileStorage& fs, const std::string&, const MyData& x)
{
    x.write(fs);
}
static void read(const FileNode& node, MyData& x, const MyData& default_value = MyData()){
    if(node.empty())
        x = default_value;
    else
        x.read(node);
}

// This function will print our custom class to the console
static ostream& operator<<(ostream& out, const MyData& m)
{
    out << "{ id = " << m.id << ", ";
    out << "X = " << m.X << ", ";
    out << "A = " << m.A << "}";
    return out;
```

```
}

int main(int ac, char** av)
{
    if (ac != 2)
    {
        help(av);
        return 1;
    }

    string filename = av[1];
    { //write
        Mat R = Mat_<uchar>::eye(3, 3),
            T = Mat_<double>::zeros(3, 1);
        MyData m(1);

        FileStorage fs(filename, FileStorage::WRITE);

        fs << "iterationNr" << 100;
        fs << "strings" << "[";                              // text - string sequence
        fs << "image1.jpg" << "Awesomeness" << "baboon.jpg";
        fs << "]";                                           // close sequence

        fs << "Mapping";                              // text - mapping
        fs << "{" << "One" << 1;
        fs <<        "Two" << 2 << "}";

        fs << "R" << R;                                      // cv::Mat
        fs << "T" << T;

        fs << "MyData" << m;                                // your own data structures

        fs.release();                                       // explicit close
        cout << "Write Done." << endl;
    }

    {//read
        cout << endl << "Reading: " << endl;
        FileStorage fs;
        fs.open(filename, FileStorage::READ);

        int itNr;
        //fs["iterationNr"] >> itNr;
        itNr = (int) fs["iterationNr"];
        cout << itNr;
        if (!fs.isOpened())
        {
            cerr << "Failed to open " << filename << endl;
            help(av);
            return 1;
        }

        FileNode n = fs["strings"];                         // Read string sequence - Get node
        if (n.type() != FileNode::SEQ)
        {
            cerr << "strings is not a sequence! FAIL" << endl;
            return 1;
        }

        FileNodeIterator it = n.begin(), it_end = n.end(); // Go through the node
        for (; it != it_end; ++it)
            cout << (string)*it << endl;


        n = fs["Mapping"];                                // Read mappings from a sequence
        cout << "Two  " << (int)(n["Two"]) << "; ";
        cout << "One  " << (int)(n["One"]) << endl << endl;


        MyData m;
        Mat R, T;

        fs["R"] >> R;                                      // Read cv::Mat
        fs["T"] >> T;
        fs["MyData"] >> m;                                 // Read your own structure_

        cout << endl
            << "R = " << R << endl;
        cout << "T = " << T << endl << endl;
```

```
129           cout << "MyData = " << endl << m << endl << endl;
130
131           //Show default behavior for non existing nodes
132           cout << "Attempt to read NonExisting (should initialize the data structure with its default).";
133           fs["NonExisting"] >> m;
134           cout << endl << "NonExisting = " << endl << m << endl;
135       }
136
137       cout << endl
138           << "Tip: Open up " << filename << " with a text editor to see the serialized data." << endl;
139
140       return 0;
141   }
```

Explanation

Here we talk only about XML and YAML file inputs. Your output (and its respective input) file may have only one of these extensions and the structure coming from this. They are two kinds of data structures you may serialize: mappings (like the STL map) and element sequence (like the STL vector). The difference between these is that in a map every element has a unique name through what you may access it. For sequences you need to go through them to query a specific item.

1. XML/YAML File Open and Close. Before you write any content to such file you need to open it and at the end to close it. The XML/YAML data structure in OpenCV is FileStorage. To specify that this structure to which file binds on your hard drive you can use either its constructor or the open() function of this:

   ```
   string filename = "I.xml";
   FileStorage fs(filename, FileStorage::WRITE);
   //...
   fs.open(filename, FileStorage::READ);
   ```

   Either one of this you use the second argument is a constant specifying the type of operations you'll be able to on them: WRITE, READ or APPEND. The extension specified in the file name also determinates the output format that will be used. The output may be even compressed if you specify an extension such as .xml.gz.

   The file automatically closes when the FileStorage objects is destroyed. However, you may explicitly call for this by using the release function:

   ```
   fs.release();                                       // explicit close
   ```

2. Input and Output of text and numbers. The data structure uses the same << output operator that the STL library. For outputting any type of data structure we need first to specify its name. We do this by just simply printing out the name of this. For basic types you may follow this with the print of the value :

   ```
   fs << "iterationNr" << 100;
   ```

   Reading in is a simple addressing (via the [] operator) and casting operation or a read via the >> operator :

   ```
   int itNr;
   fs["iterationNr"] >> itNr;
   itNr = (int) fs["iterationNr"];
   ```

3. Input/Output of OpenCV Data structures. Well these behave exactly just as the basic C++ types:

   ```
   Mat R = Mat_<uchar >::eye  (3, 3),
       T = Mat_<double>::zeros(3, 1);

   fs << "R" << R;                                   // Write cv::Mat
   ```

```
fs << "T" << T;

fs["R"] >> R;                                   // Read cv::Mat
fs["T"] >> T;
```

4. Input/Output of vectors (arrays) and associative maps. As I mentioned beforehand, we can output maps and sequences (array, vector) too. Again we first print the name of the variable and then we have to specify if our output is either a sequence or map.

   For sequence before the first element print the "[" character and after the last one the "]" character:

```
fs << "strings" << "[";                              // text - string sequence
fs << "image1.jpg" << "Awesomeness" << "baboon.jpg";
fs << "]";                                           // close sequence
```

   For maps the drill is the same however now we use the "{" and "}" delimiter characters:

```
fs << "Mapping";                              // text - mapping
fs << "{" << "One" << 1;
fs <<        "Two" << 2 << "}";
```

   To read from these we use the FileNode and the FileNodeIterator data structures. The [] operator of the FileStorage class returns a FileNode data type. If the node is sequential we can use the FileNodeIterator to iterate through the items:

```
FileNode n = fs["strings"];                         // Read string sequence - Get node
if (n.type() != FileNode::SEQ)
{
    cerr << "strings is not a sequence! FAIL" << endl;
    return 1;
}

FileNodeIterator it = n.begin(), it_end = n.end(); // Go through the node
for (; it != it_end; ++it)
    cout << (string)*it << endl;
```

   For maps you can use the [] operator again to acces the given item (or the >> operator too):

```
n = fs["Mapping"];                                // Read mappings from a sequence
cout << "Two  " << (int)(n["Two"]) << "; ";
cout << "One  " << (int)(n["One"]) << endl << endl;
```

5. Read and write your own data structures. Suppose you have a data structure such as:

```
class MyData
{
public:
      MyData() : A(0), X(0), id() {}
public:   // Data Members
   int A;
   double X;
   string id;
};
```

   It's possible to serialize this through the OpenCV I/O XML/YAML interface (just as in case of the OpenCV data structures) by adding a read and a write function inside and outside of your class. For the inside part:

```
void write(FileStorage& fs) const                        //Write serialization for this class
{
  fs << "{" << "A" << A << "X" << X << "id" << id << "}";
}

void read(const FileNode& node)                          //Read serialization for this class
{
  A = (int)node["A"];
  X = (double)node["X"];
  id = (string)node["id"];
```

```
}
```

Then you need to add the following functions definitions outside the class:

```
void write(FileStorage& fs, const std::string&, const MyData& x)
{
x.write(fs);
}

void read(const FileNode& node, MyData& x, const MyData& default_value = MyData())
{
if(node.empty())
    x = default_value;
else
    x.read(node);
}
```

Here you can observe that in the read section we defined what happens if the user tries to read a non-existing node. In this case we just return the default initialization value, however a more verbose solution would be to return for instance a minus one value for an object ID.

Once you added these four functions use the >> operator for write and the << operator for read:

```
MyData m(1);
fs << "MyData" << m;                                 // your own data structures
fs["MyData"] >> m;                                  // Read your own structure_
```

Or to try out reading a non-existing read:

```
fs["NonExisting"] >> m;   // Do not add a fs << "NonExisting" << m command for this to work
cout << endl << "NonExisting = " << endl << m << endl;
```

Result

Well mostly we just print out the defined numbers. On the screen of your console you could see:

```
Write Done.

Reading:
100image1.jpg
Awesomeness
baboon.jpg
Two  2; One  1


R = [1, 0, 0;
  0, 1, 0;
  0, 0, 1]
T = [0; 0; 0]

MyData =
{ id = mydata1234, X = 3.14159, A = 97}

Attempt to read NonExisting (should initialize the data structure with its default).
NonExisting =
{ id = , X = 0, A = 0}

Tip: Open up output.xml with a text editor to see the serialized data.
```

Nevertheless, it’s much more interesting what you may see in the output xml file:

```
<?xml version="1.0"?>
<opencv_storage>
<iterationNr>100</iterationNr>
<strings>
  image1.jpg Awesomeness baboon.jpg</strings>
<Mapping>
```

```
   <One>1</One>
   <Two>2</Two></Mapping>
<R type_id="opencv-matrix">
   <rows>3</rows>
   <cols>3</cols>
   <dt>u</dt>
   <data>
     1 0 0 0 1 0 0 0 1</data></R>
<T type_id="opencv-matrix">
   <rows>3</rows>
   <cols>1</cols>
   <dt>d</dt>
   <data>
     0. 0. 0.</data></T>
<MyData>
   <A>97</A>
   <X>3.1415926535897931e+000</X>
   <id>mydata1234</id></MyData>
</opencv_storage>
```

Or the YAML file:

```
%YAML:1.0
iterationNr: 100
strings:
   - "image1.jpg"
   - Awesomeness
   - "baboon.jpg"
Mapping:
   One: 1
   Two: 2
R: !!opencv-matrix
   rows: 3
   cols: 3
   dt: u
   data: [ 1, 0, 0, 0, 1, 0, 0, 0, 1 ]
T: !!opencv-matrix
   rows: 3
   cols: 1
   dt: d
   data: [ 0., 0., 0. ]
MyData:
   A: 97
   X: 3.1415926535897931e+000
   id: mydata1234
```

You may observe a runtime instance of this on the YouTube here .

You did not find what you were looking for?

Ask a question on the Q&A forum.
If you think something is missing or wrong in the documentation, please file a bug report.