Image Processing Algorithms Integration System

Andrey A. Kravtsov[*], Dmitry V. Yurin[**]
[*] Moscow State University, Faculty of Computational Mathematics and Cybernetics, Lab. of Mathematical Methods of Image Processing.
[**] Institute of Computing for Physics and Technology
andrey.a.kravtsov@gmail.com, yurin_d@inbox.ru

Abstract

An approach of building complex image processing algorithms using independent blocks is presented. It is assumed that blocks are written in C++ programming language and the integration is performed due to means of Microsoft .NET Framework. A method of presentation of separate algorithm as a .NET dynamic loading library is described. The main advantage of this approach is ability of high-level algorithm construction and without the need of code recompilation. A new image object format suggested. This format is very convenient for data interchange between classes running in different dll's. It also eliminates the need of image copying from one container into another. For system demonstration purposes an implementation of straight line detection algorithm described.

Keywords: *image processing, image filtering, computer vision, algorithms integration, reusing of code.*

1. INTRODUCTION

Modern image processing algorithms are very complex and consist of numerous low-level subroutines. It is common practice that authors of articles describe their algorithms as pseudo code that has small size [4-7]. This shows that an algorithm can be presented as a super-algorithm consisting of simple blocks with low-level subprograms such as Fourier or Hartley transforms, interpolation, filtering, search of maximums, convolutions etc. An ordinary approach in programming such a super-algorithm is to implement blocks in C++ language and to integrate these blocks in common C++ program which can also contain graphical user interface code [8]. The choice of programming language for implementing low-level subroutines is in most cases obvious (C++ language implied). But it is not true for high-level algorithm because the most part of overall computational time relates to low-level subroutines. Implementation of super-algorithm is just call of several (not more than ten) such subroutines.

Combining of all blocks in a common C++ project requires common data formats as well as programming of all code parts in similar style. Moreover, various difficulties can occur by using ready components (algorithms) shared in internet for public access. These are different image representation types, name conflict etc. It is also often a problem to extract “pure algorithm” from the project, because low-level subroutines, super-algorithm, graphical user interface implementation details, data formats, exception handling are mixed.

Thus a solution for problems described above is to write low-level subroutines in C++ programming language (i.e. to make blocks of reusable code), combine them using a scripting language. GUI implementation depends on used operating system. At the moment the most natural approach for programming GUI for Windows OS is .NET Framework [3]. Integration with C++ modules is possible due to Managed C++ programming language. MC++ allows creating unified modules with .Net interface accessible from all .Net programming languages. But inside these modules developer can use all C++ native means like pointers, templates, polymorphism etc. It is important to note that JScript and VBasicScript (Microsoft.Vsa, Microsoft.JScript, Microsoft..VBScript namespaces) interpreters are available in .Net framework.

The goal of this paper is to develop an approach for programming low-level subroutines using efficient C++ language, combine these subroutines in a common super-algorithm using scripting languages and eliminate copying images from one container into another one only to provide compatibility between modules written by different people. Implementing high-level algorithm in a scripting language makes code more readable and accessible for editions without the need of recompilation.

2. SYSTEM ARCHITECTURE

Fig.1. depicts overall system architecture.

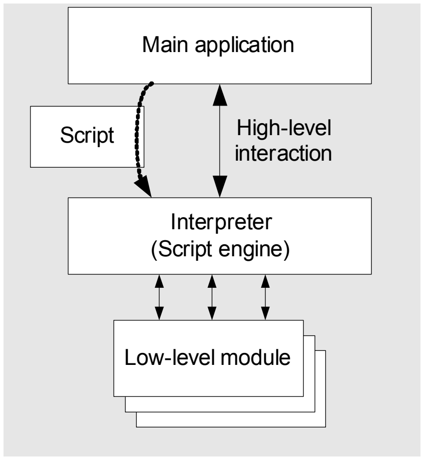

Fig.1. System architecture.

The **main application** is written in one of .Net programming languages. It can be either an existing program or newly developing solution.

The **interpreter**, or scripting engine, is a C# class which can interpret JScript (or VBScript.NET) scripts. Input data for the script engine are JScript-code, list of dynamic loading libraries, list of namespaces and variables if any. A big advantage is the possibility to pass a variable of *any* .Net framework object type.

Script has JScript.Net instructions: objects instantiation, function calls and so on. Some pluses of using scripting language should be noted: firstly, all low-level implementation details are invisible for **main application**-developer. So we get a black box which takes some input variables and returns a result. Implementation details are not important thus. Secondly, JScript.Net is an easy, but in the same time very powerful programming language. It provides all capabilities supported by

.Net framework. Thirdly, there is no need to recompile any part of the system after script editing. This means given approach allows user to program and chose algorithm implementation details. The only restriction which has user is the interface of interaction with the main application.

A **low-level module** is a dynamic loading library (dll) containing Managed C++ class – a wrapper for C/C++ code. Using of a managed class ensures module unification and algorithm implementing in the native language causes high speed efficiency.

3. IMAGE OBJECT

For modeling images a new format was developed. This format is compatible with ENVI system [9], which supports a wide range of data types and is easy to use. It should be noted that our format supports DIB and BMP image formats. For image processing algorithms the origin of images is not important, but access type to pixels is essential. Because of this an approach was selected that allows working with images created by various external tools. Particularly if you write a program in C# and use our system, you don't need to copy C# image into another container.

Inner structure of images is described by field of the **ImageDescriptor** class:

Width – image width in pixels;

Height – image height in pixels;

Bands – number of image color components (e.g. 1 for grayscale, 3 for color);

DataType – sample data type. Supported types: Byte (DataType = 1), unsigned short (16 bit, DataType = 12) and float32 (DataType = 4);

Interleaving – bands alternation way. Supported values: BSQ (bans sequentional) – all pixel data of first color band are followed by all pixel data of the second color band etc; BIL (band interleaving by line) – alternation of color bands line by line; BIP (band interleaving by pixel) – alternation of color bands pixel by pixel;

NLS (next line shift) – a value that should be added to pointer for shifting to the same position on image, but one line below. NLS can differ from Width.

NBS (next band shift) – a value that should be added to pointer for shifting to the same position on image, but on the next color band.

DataPtr – pointer to image data row.

Instances of **ImageDescriptor** class don't contain real data; they just store full description of image. The necessity of developing such a class is following: it is assumed that algorithms from different dynamic loading libraries pass to each other variables of various data types, for instance images. It is not efficient to pass megabytes-sized images. Some methods were developed to solve these problems:

int GetInforRow() returns handle to array of eight integers – fields of ImageDescriptor class;

ImageDescriptor(int inforow) constructor instantiates an object based on handle.

Data type **integer** is supported by both C/C++ language and .Net framework; it can also contain a handle. So the method described above allows passing images between different modules.

4. SAMPLE

For demonstration of the approach described above a straight line detection algorithm was selected [4]. Block diagram is presented in Fig.2.

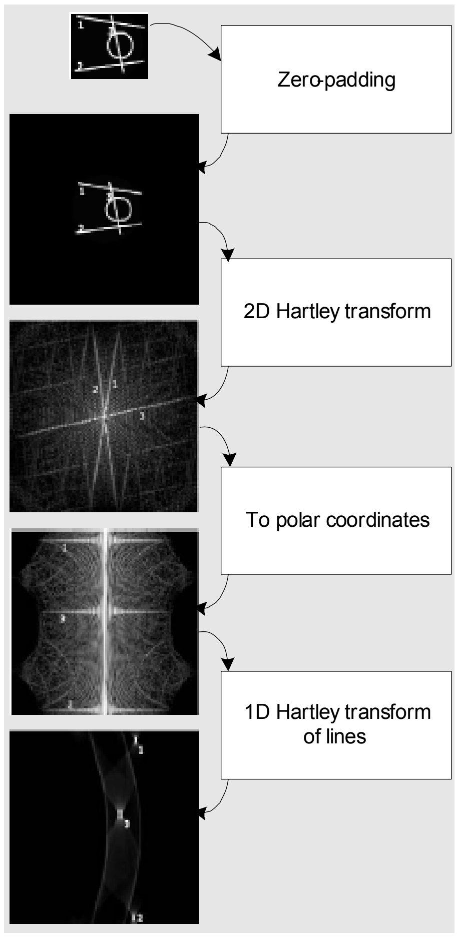

Fig.2. Straight line detection algorithm schema

4.1 Main application code

The following listing contains C# code with scripting engine instantiation, reading script text from file and script interpretation in it.

```
C#
string appPath = Application.StartupPath;

m_ScriptEngine = ScriptEngine.GetInstance(new
string[] {
    appPath + @"\DotNetImageWrapper.dll",
    appPath + @"\FilterPreprocessing.dll",
    appPath + @"\Hartley.dll",
    appPath + @"\Hartley1DWrapper.dll",
    appPath + @"\ToPolarWrapper.dll"});

string script = ScriptEngine.ReadScriptFromFile(
  Application.StartupPath + @"\LineDetection.js");

m_ScriptEngine.AddVariableAndInitialize(
          "_filterInput", new Bitmap("demo.bmp"));

m_ScriptEngine.Eval(script);
m_ScriptEngine.Eval("JLastLineObject.Run();");
```

4.2 Interpreter

The key role in this work has JScript.NET scripts interpreter. Microsoft.JScript.dll and Microsoft.Vsa.dll assemblies were used. Unfortunately, the standard help [10] doesn't provide details about classes from Microsoft.Vsa and Microsoft.JScript namespaces. Because of this we had to guess all needed information based on class, methods and properties names.

The interpreter is very important and difficult to implement, so it is worth to list the whole **ScriptEngine class** code.

C#

```
using System;
using System.Reflection;
using System.Collections;
using System.Collections.Generic;
using System.Windows.Forms;
using Microsoft.JScript;
using Microsoft.JScript.Vsa;
using Microsoft.Vsa;

public class ScriptEngine :
System.ComponentModel.Component
{
GlobalScope m_GlobalScope = null;

public GlobalScope GlobalScope
{
    get { return m_GlobalScope; }
}

// Constructor gets an array of assemblies names
// to load
public ScriptEngine(string[] assemblyReferences)
{
    GlobalScope gs =
       VsaEngine.CreateEngineAndGetGlobalScope(
          false, assemblyReferences);

    VsaEngine engine = gs.engine;

    GlobalScope newGS =new GlobalScope(gs,engine);

    engine.PushScriptObject(newGS);
    m_GlobalScope = newGS;
}

// Script evaluation
public object Eval(string scriptText)
{
    // return object
    object result = null;

    try
    {
        result = Eval.JScriptEvaluate(scriptText,
           m_GlobalScope.engine);
    }
    catch (JScriptException ex)
    {
        result = ex;
        MessageBox.Show(ex.Message, string.Format(
         "Error in script in position ({0}, {1})",
         ex.Line.ToString(),ex.Column.ToString()),
         MessageBoxButtons.OK,
         MessageBoxIcon.Error);
    }
    catch (Exception ex)
    {
      MessageBox.Show(ex.Message, "Error",
      MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
    return result;
}

// Reading script from file
public static string ReadScriptFromFile(
   string fileName)
{
    System.IO.StreamReader sr = new
       System.IO.StreamReader(fileName);

    string sScript = "";
    try
    {
        sScript = sr.ReadToEnd();
    }
    catch (Exception ex)
    {
        //TODO
    }
    finally
    {
        sr.Close();
    }

    return sScript;
}

// Method adds all namespaces from a given
// assembly
static void CollectNamespaces(Assembly assembly,
Hashtable hTable)
{
    Type[] types = assembly.GetTypes();

    if (types == null || types.Length == 0)
        return;

    foreach (Type type in types)
    {
        if (!type.IsSubclassOf(typeof(Object)))
            continue;

        if (type.Namespace != null)
            hTable[type.Namespace] = type;
    }
}

// Method adds hTable assemblies referenced by
// a given assembly
static void CollectReferences(Assembly assembly,
Hashtable hTable)
{
    hTable[assembly.FullName] = assembly;

    AssemblyName[] aNames =
       assembly.GetReferencedAssemblies();

    if (aNames == null || aNames.Length == 0)
        return;

    foreach (AssemblyName aName in aNames)
    {
        string fullName = aName.FullName;

        if (hTable.ContainsKey(fullName))
            continue;

        try
        {
            Assembly assemb =
               Assembly.Load(aName);
            hTable.Add(fullName, assemb);
        }
        catch { }
    }
}
// Namespaces loading
void Import(params string[] list)
{
    foreach (string listItem in list)
    {
        Import.JScriptImport(listItem,
           GlobalScope.engine);
    }
}

// Method adds a new variable with name name
public void AddVariable(string name)
{
    m_GlobalScope.AddField(name);
}

// Assigning value to the variable name
public object SetVariable(string name,
                          object value)
{
    return m_GlobalScope.InvokeMember(name,
       BindingFlags.SetField, null, m_GlobalScope,
       new object[] { value }, null, null, null);
}

// Declaring and intialization of a new variable
public void AddVariableAndInitialize(string
variableName, object variableValue)
{
    AddVariable(variableName);
    SetVariable(variableName, variableValue);
}

// Method gets value of variable varName
public object GetVariable(string varName)
{
    return m_GlobalScope.InvokeMember(varName,
       BindingFlags.GetField, null, m_GlobalScope,
       null, null, null, null);
}

// Method returns instance of ScriptEngine class
// It gets array of paths to *.dll files
public static ScriptEngine GetInstance(
   string[] dlls)
{
    List<Assembly> aAssemblies =
       new List<Assembly>();

    // add to collection loaded assemblies
    aAssemblies.AddRange(
       AppDomain.CurrentDomain.GetAssemblies());

    foreach (string dll in dlls)
    {
        try
        {
        Assembly assembly =Assembly.LoadFrom(dll);

            if (!aAssemblies.Contains(assembly))
                aAssemblies.Add(assembly);
        }
        catch { }
    }

    Hashtable tableAssemblies = new Hashtable();

    // Creating collection of needed assemblies
    foreach (Assembly assembly in aAssemblies)
    {
        ScriptEngine.CollectReferences(assembly,
           tableAssemblies);
    }

    Hashtable tableNamespaces = new Hashtable();

    // Create namespaces collection
    foreach (DictionaryEntry entry in
       tableAssemblies)
    {
        ScriptEngine.CollectNamespaces(
          (Assembly)entry.Value, tableNamespaces);
    }

    string[] aStringDll = (string[])(
    new ArrayList(tableAssemblies.Keys)).ToArray(
       typeof(string));

    string[] aStringNS = (string[])(
    new ArrayList(tableNamespaces.Keys)).ToArray(
       typeof(string));

    ScriptEngine engine = new
       ScriptEngine(aStringDll);

    engine.Import(aStringNS);

    return engine;
}
}
```

4.3 Low-level modules

How it was already mentioned above, all algorithms are encapsulated into .Net assemblies. The following listing contains code example for managed wrapper for two-dimensional Hartley transform:

```
Managed C++
// include section
#include "integral_transforms.h"

using namespace System;
using namespace System::Drawing;
using namespace ScriptImaging;

public ref class Hartley2DWrapper
{
public:
   Hartley2DWrapper(){}
   ~Hartley2DWrapper(){}
   !Hartley2DWrapper() {}

public:
   void DoJob(int imginfo)
   {
        ImageDescriptor img_id(imginfo);

        size_t n[] = {img_id.Width,img_id.Height};

        float* fPtr = (float*)(img_id.DataPtr);

        SwapQuadrants<2, false>(fPtr, n);
        fht2D(fPtr, n[0], n[1]);
        SwapQuadrants<2, false>(fPtr, n);
   }
};
```

This example illustrates easiness of creation managed wrapper for a C++ functions (SwapQuadrants and fht2D).

4.4 Script

The following listing contains script where algorithm presented in Fig.2 is implemented:

```
JScript.NET
public class JLineDetector
{
    public function Run(): void
    {
        var bmpW:int = _filterInput.Width;
        var bmpH:int = _filterInput.Height;

        var maxWH:int = Math.max(bmpW, bmpH);
        var twoPow:int = 1;

        while (twoPow < maxWH)
            twoPow *= 2;

        var img1:ScriptImage =
            new ScriptImage(twoPow, twoPow, 1, 3);
        var img2:ScriptImage =
            new ScriptImage(twoPow, twoPow, 1, 3);

        var filter1:ZeroPadding =
            new ZeroPadding();
        filter1.DoJob(_filterInput, img1.InfoRow);

        var filter2:Hartley2DWrapper =
            new Hartley2DWrapper();
        filter2.DoJob(img1.InfoRow);

        var filter3:ToPolarWrapper =
            new ToPolarWrapper();
        filter3.DoJob(img1.InfoRow, img2.InfoRow);

        var filter4:Hartley1DWrapper =
            new Hartley1DWrapper();
        filter4.DoJob(img2.InfoRow);

        // Save images
        img1.SaveAsBitmap(false, "out1.png");
        img2.SaveAsBitmap(false, "out2.png");
    }
};

var JLastLineObject:JLineDetector =
    new JLineDetector();
```

4.5 Passing images between different environments

An importation image representation and transmission (between different programming languages) feature should be noted. An instance of ScriptImage type is created inside the script (JScript.NET), than it is passed into function from assembly as integer (i.e. handle). Inside the function an image descriptor is created on the basis of this handle. ImageDescriptor provides programmer with convenient access to such image data as width, height, data pointer etc.

5. CONCLUSION

A system allowing constructing complex algorithms from independent C++ reusable code blocks is described. Presented image description format brings such advantages as ability to access data of images created by third-party programmers and eliminate the need of copying big amounts of data from one container into another. It also allows passing images between different environments. Using of scripting engine and JScript.NET language facilitates program implementation details editing process and eliminates the need to recompile any part of program after editing.

6. ACKNOWLEDGEMENTS

The research is done under financial support of RFBR, grants 06-01-00789-a, 08-07-00362-a.

7. REFERENCES

[1] A. Alexandrescu. Modern C++ design. Addison-Wesley, 2001. ISBN 0-201-70431-5

[2] B. Stroustrup. The C++ Programming Language, Special Edition, Addison-Wesley, 2000. ISBN 0-201-70073-5.

[3] A. Troelsen. C# and the .Net Platform, Springer New York Inc, ISBN: 9781590590553

[4] D.B. Volegov, V.V. Gusev, D.V. Yurin. Straight Line Detection on Images via Hartley Transform. Fast Hough Transform //GraphiCon'06, Novosibirsk, Russia, June 2006

http://www.graphicon.ru/2006/proceedings/papers/fr11_35_Volegov_Gusev_Yurin.pdf.

[5] B.S. Reddy, B.N. Chatterji, "An FFT-based technique for translation, rotation, and scale-invariant image registration", IEEE PAMI, Vol 5(8), pp. 1266-1271, August, 1996.

[6] James Davis. Mosaics of scenes with moving objects. In Proc. Computer Vision and Pattern Recognition Conf., pages 354--360, 1998. http://citeseer.ist.psu.edu/davis98mosaics.html.

[7] Siavash Zokai, George Wolberg. Image Registration Using Log-Polar Mappings for Recovery of Large-Scale Similarity and Projective Transformations //IEEE Transactions on Image Processing, Vol. 14, No. 10, October 2005. http://www-cs.engr.ccny.cuny.edu/~wolberg/pub/tip05.pdf

[8] B. Georgescu, P. Meer. Point Matching under Large Image Deformations and Illumination Changes //IEEE Transactions On Pattern Analysis and Machine Intelligence, June 2004, -V. 26, -No. 6, -P. 674-688. Code available at http://www.caip.rutgers.edu/riul/research/code.html

[9] Site of company ITT Visual Information Solutions, (before May 15, 2006 – RSI - Research Systems, Inc.), разработчик пакета ENVI, http://www.ittvis.com/index.asp.

[10] Microsoft Developer Network (MSDN)

About the authors

Andrey A. Kravtsov, a fifth year student, Moscow State University, Department of Computational Mathematics and Cybernetics.

E-mail: andrey.a.kravtsov@gmail.com

Dmitry V. Yurin, PhD, is a senior scientist at Institute of Computing for Physics and Technology and at Laboratory of Mathematical Methods of Image Processing, Chair of Mathematical Physics, Faculty of Computational Mathematics and Cybernetics, Moscow Lomonosov State University. His contact email is yurin_d@inbox.ru