University of West Bohemia in Pilsen
Faculty of Applied Sciences
Department of Computer Science and Engineering

DIPLOMA THESIS

Pilsen, 2006

Jiří Skála

University of West Bohemia in Pilsen
Faculty of Applied Sciences
Department of Computer Science and Engineering

Diploma Thesis

Masking Images for DTP Needs

Implemented as Adobe Photoshop Plug-in

Pilsen, 2006

Jiří Skála

Abstract

This thesis has two major aims. In the first part, creation of plug-in modules for Adobe Photoshop is described. It is explained how plug-ins work, how their interface looks like and how to write plug-ins for Adobe Photoshop. It is also presented how to create dialog windows using Adobe Dialog Manager. The second part of this thesis deals with image masking. Several existing algorithms are introduced. Since no one is suitable for our needs a new masking technique is proposed. It is then described in detail and implemented as Photoshop plug-in. Finally the method is evaluated and suggestions for future work are given.

Keywords: Adobe Photoshop, plug-in creation, Adobe Dialog Manager, alpha channel, image masking, image matting.

Contents

I hereby declare that this diploma thesis is completely my own work and that I used only the cited sources.

Pilsen, May 17, 2006 Jiří Skála

Preface and thesis aims

Adobe Photoshop is very popular application for editing raster images. It has great possibilities but can also be further extended by so-called plug-ins. These are program modules that add new functionality to current application. Photoshop was one of the first applications that have plug-in support and we can say that it was the program which made plug-in modules really popular. Adobe also profits from it very much. Thanks to skillful programmers from all over the world, Photoshop has almost unlimited possibilities and therefore it became something like a standard in raster image processing. Currently many other competitive applications have some support for Photoshop plug-ins. Demands on writing plug-in modules are growing since it has several advantages. Users can use all their favourite functions in a single application. There is no need to transfer data between various utility programs. Programmers can benefit from the environment provided by the plug-in host application. It takes care of user interface, format conversions, exception handling, etc. Developers can thus concentrate on the core of given assignment.

Image masking (also called matting) is a process of extracting image parts that are somehow interesting. Most often we want to separate objects in the foreground and suppress the background. This technique is frequently used especially in Desktop Publishing (DTP). Many masking algorithms exist, they usually fall into one of three major categories. *Constant color matting* assumes that objects are shot against a background of constant color (mostly blue, sometimes green). Therefore it is often referred to as blue screen matting. Of course the foreground object must not contain the backing color. This technique is most often used in film industry and usually employs special hardware devices. The second approach is *difference matting*. In this case the background may be potentially arbitrary but we need a picture of empty background (with no objects) as a reference. Later when some objects are added they can be extracted with respect to their difference from reference background. The last category is *natural image matting*. It's partly a combination of both previous techniques. The background may be arbitrary but we don't have any reference picture. First, approximate object boundaries must be specified. Alternatively, we can designate areas that are definitely foreground and definitely background. All these tasks are most often done by the user. The algorithm then tries to refine given

boundaries so that they correspond precisely to the object. All these matting techniques are discussed in more detail in sections 2.3, 2.4 and 2.5.

The main goal of this thesis is to explore how Photoshop plug-ins work and how to create them. This is discussed in detail in chapter 1. First a short summary regarding plug-in possibilities is made. Then a general interface is described with further focus on Filter type modules. Essential data structures are presented and data interchange between Photoshop and plug-in module is made clear. After that a plug-in calling scheme is explained. The host application provides modules with several utility functions, these are discussed with some hints how to use them. There is also one section about the Plug-in Property List, an essential set of information describing the plug-in module. Since building a plug-in is not absolutely easy, one section discusses how to set up a Microsoft Visual Studio project to compile plug-in successfully. The last section is dedicated to Adobe Dialog Manager, an interface for managing dialog windows.

The secondary submission of this thesis is to find an algorithm for difference masking. Because it is intended for use in DTP, we can expect some reasonable input in high resolution and objects contrasting with background. Details are discussed in chapter 3 on page 50 or in the conclusion. Since it turned out that none of currently known masking methods is well suitable for our needs a new method is developed in chapter 3. Section 3.2 briefly describes several unsuccessful attempts, while section 3.3 presents the technique that was finally implemented. Results of all proposed approaches are illustrated by example pictures. More of them may be found in the appendix or on the enclosed CD. Interesting details regarding the implementation are discussed in section 3.4. In section 3.5 there is a short users manual. Section 3.5 contains method evaluation and thesis conclusion.

Chapter 1

How to write plug-ins for Adobe Photoshop

1.1 Plug-ins in general

Adobe Photoshop is widely known as a very powerful and perhaps the most widespread program for creating and editing raster graphics. But it brought another great contribution to desktop computing. Thanks to Photoshop, plug-ins started to be popular among users. Such type of extension was first introduced in 1987 in a program called HyperCard[1]. In that time, new functions had to be pasted directly into the application.

Later, Silicon Beach in its products Digital Darkroom and SuperPaint introduced an improved architecture. Plug-ins were in separate files and version numbering ensured backwards compatibility. With some modifications, such approach remained actual up to the present day. Nevertheless plug-ins started to be really popular as lately as Photoshop introduction. It was also a great advantage for Photoshop itself, because skillful programmers made its possibilities almost unlimited. Competitive applications got into an unpleasant situation. Many of them were later forced to add Photoshop plug-in support in order to achieve some market profit. Nowadays almost every major application has some extension possibilities based on plug-in architecture.

First of all we should make clear what we are going to talk about. A *plug-in* is a program that extends functionality of already existing application. Such application is called a *plug-in host*. It takes care of loading the plug-in into memory and calling it. Plug-in host is usually a standalone application like Photoshop. However it may be also another plug-in. For example in Adobe Illustrator, there is a Photoshop Adapter plug-in, that serves as a plug-in host for Photoshop plug-ins. Plug-in modules may be created by the author of the application they are intended for. However the major advantage of plug-in modules is right that they

[1] A database application by Bill Atkinson with its own simple programming language.

may also be written by independent programmers. Plug-ins may be added and modified according to actual needs, independently of the application they are designed for.

We can divide plug-in advantages to user's and programmer's ones. The user certainly appreciates, when he has all the necessary functions available in a single application. He doesn't have to save the work and open it again in different programs and solve compatibility problems. The programmer can take advantage of plug-in host environment. There is no need for creating graphical user interface, memory management, format conversion, etc. Thus the developer can concentrate on solving the core of the task. Result presentation, *undo* commands and other utility functions are provided by the plug-in host.

1.2 Plug-ins for Adobe Photoshop

In this paper, we are going to concern with plug-in development for Adobe Photoshop on a Microsoft Windows platform. Originally Photoshop was developed for Apple Macintosh only. This makes itself felt also on today's Windows version. Fortunately the differences[2] between platforms do not cause any serious problems. We can encounter data structures that are not typical for Windows. Also some function calls and other constructions might look unfamiliar at first. The resulting source code will be easily portable to Macintosh. It is however unpleasant that if we are developing for Windows only, we still won't avoid some specialities. Certain reward may be that Photoshop plug-ins can be used also with other Adobe products like Premiere or PhotoDeluxe. Even some programs by other companies provide support for Photoshop plug-ins.

There are nine types of plug-in modules in Photoshop. A summary with brief description follows. Perhaps some of you will be surprised when you find out what all can be implemented as a plug-in.

Automation Plug-ins of this type can control Photoshop scripting. It is a powerful mechanism whereby we can invoke in fact any menu command or execute a tool such as plug-in. Most of user actions can be done by scripting. Automation plug-ins are often used in cooperation with other plug-in modules for tasks, that exceed capabilities of other plug-ins. A typical example is adding an alpha channel to the image. Automation modules can be found in the *File* | *Automate* or *Help* menu.

Color Picker These modules give the possibility to create alternative color selection palettes. By default Photoshop uses its own or the system palette. Color Picker plug-ins can implement color selection in arbitrary fashion,

[2]For example Intel processors store numbers in little endian format, while Motorola and PowerPC use big endian byte ordering.

for example a color catalogue. The desired color palette may be selected in Photoshop preferences.

Import Loads an image and opens it in a new window. Import plug-ins are mainly intended as an interface to scanners and other peripheral devices. They may also generate synthetic images. They are usually not used for reading files, since this is a task for Format modules. Import plug-ins can be found in the *File* | *Import* menu.

Export Deals with image output to a printer or other non-disc device. Writing images to files is done by Format modules. You will find Export plug-ins in the *File* | *Export* menu.

Extension Plug-in modules of this type are called once at Photoshop startup and once at Photoshop shutdown. They are used for various device initialization or custom cursor drawing. The interface of these modules is not public, it is for Adobe's internal use only.

Filter This is what most people imagine as Photoshop plug-in. It's also the most frequently used type of module. It is used to modify image data in selected areas. The modification may be anything from slight color correction to dramatic effects. As you can expect, filter plug-ins are located in the *Filter* menu.

Format Sometimes called also File Format or Image Format. Adobe Photoshop supports many image formats. This set can be even expanded by Format modules. It is possible to implement loading and saving various exotic or compressed image formats. Another interesting exercise would be e.g. TIFF metadata processing. Format plug-ins will show up as new options in drop-down menus of *Open...* or *Save As...* dialogs. A nice example of perfect plug-in to Photoshop interconnection.

Parser Similar to Import and Export modules. Parser plug-ins are used for data interchange between Photoshop and other, mainly vector, applications like Adobe Illustrator or Adobe PageMaker. Parser plug-in interface is not public.

Selection These plug-in modules can alter which image pixels are selected. In Selection modules it is possible to implement creation of various complex selections. The plug-in may work both with selection shape and the color of contained pixels. Modules can process selections, masks and even paths. They can be found in the *Selection* menu.

When we talk about plug-ins, most of users think of Filter plug-ins. As you can see from the summary above, there are much more possibilities. And it's not

all. A plug-in module is a program like any other. So it can use functions for playing sounds and video, rendering 3D graphics, etc. It would be possible to implement for instance an OCR program, spreadsheet or even a game. Well, this is probably not what anyone would try. I just wanted to show, that the plug-in doesn't have to limit itself just to the environment provided by the plug-in host. We will only deal with Filter plug-ins further in this paper.

1.3 Plug-in interface

Adobe Photoshop plug-ins are implemented as DLL libraries. When Photoshop starts, it scans the plug-in directory ("Plug-Ins" by default) for plug-in modules and makes them ready to use.

In most cases plug-ins are called upon user request. They have a single entry point with a function prototype of

```
DLLExport MACPASCAL void PluginMain(
    const short selector, void* pluginParamBlock,
    long* pluginData, short* result)
```

The `selector` parameter determines which operation should the module perform. If `selector` is zero, it always and for all modules means that the plug-in should display an about box. Details will be discussed later in section 1.5. Parameter `pluginParamBlock` is a pointer to an extensive structure, that is used for two-way information exchange between plug-in and host. In the `pluginData` pointer the module can store a reference to its internal data. After returning from plug-in call, this reference will be hold by the host. So when the module is executed next time, it will have its data available. The plug-in should always set the output parameter `result`, which has a meaning of an error code. It informs the plug-in host about the result of performed operation. The function body is usually a switch statement, that dispatches program control according to `selector` value. When executing a plug-in, the entry point is called several times. Detailed description is in section 1.5.

As mentioned before, errors are reported in the `result` parameter. Symbolic constant `noErr` (value zero) means that everything went all right. Any nonzero number indicates an error. When plug-in returns a positive value, it tells host that the module has already informed the user about an error. This is useful especially in case when only the module understands the error. On the other hand a negative return value means that a standard error message should be displayed by the host. This is the preferred way, since it spares programmer's work and the plug-in host will display the message in local language. When reporting errors it is generally recommended to use symbolic constants defined in the SDK or standard operating system error codes.

1.4 Filter plug-ins

This paper focuses on Filter plug-in modules. Let's have a look at the Filter Parameter Block that is passed as the `pluginParamBlock` parameter. It's an extensive structure, in current Photoshop version CS2 counts over one hundred entries. However it is by far not necessary to know all of them. So the following list summarizes only the important ones.

parameters If the plug-in has some parameters, it will store them in a block of memory. We will probably want the setting to be retained even after the plug-in ends. Therefore the module stores a reference to its parameters into `parameters` entry. Plug-in host will then maintain this block of memory until the plug-in is executed again.

imageSize The size of the whole image.

imageMode Image color space.

planes Number of image channels, it is the number of color components including alpha channels and/or masks.

depth Image bit depth (1, 8 or 16).

maxSpace In this entry Photoshop specifies the maximum amount of memory that will be available to the plug-in. If the module can estimate its memory requirements and those are lower, it should lower the `maxSpace` value accordingly. Thereby more memory will be left to Photoshop and we will avoid unnecessary swapping. How to estimate memory requirements is further discussed in section 1.5.2.

bufferSpace If the plug-in module is going to allocate large memory blocks, it should set `bufferSpace` to the number of bytes needed. Before the plug-in starts working, plug-in host will try to free up that amount of memory. If the module allocates memory using Buffer suite, `bufferSpace` can remain zero. More about Buffer suite functions may be found in section 1.6.2.

filterRect This is a bounding rectangle of the area the plug-in will process. If there is a non-rectangular selection in the image, `filterRect` is the smallest rectangle that encloses all the selected pixels. Right and bottom bounds are counted *noninclusive*.

inRect, outRect In these entries plug-in specifies input and output rectangles. These are subsets of the image being filtered. They determine areas from where the module will read data and where it will write. Right and bottom bounds are counted *noninclusive*. These rectangles don't need to be the same (not even the same size) although they usually are.

inLoPlane, inHiPlane, outLoPlane, outHiPlane Plug-in sets these entries according to image channels it wants to process. It's clear that `inLoPlane` and `inHiPlane` specify the range of input channels, while `outLoPlane` and `outHiPlane` give the range of channels for output. If the module wants to work with all the channels at once it sets `inLoPlane = outLoPlane = 0` and `inHiPlane = outHiPlane = planes - 1`.

inData, outData Pointers to blocks of memory with input and output data respectively. *Both these blocks are allocated by the plug-in host, not by the plug-in itself.* Description of how image data is organized can be found in section 1.4.2.

haveMask Determines whether the image has a mask. It is the case when there is a non-rectangular selection defined. The mask then specifies which pixels are selected. Mask manipulation is further discussed in section 1.4.1.

maskRect, maskData These entries are concerned with mask. They have similar meaning as `inRect` and `inData`.

inRowBytes, outRowBytes, maskRowBytes This determines the offset between rows of data. There may be padding bytes at the end of rows. Naturally these bytes should be skipped.

inputRate, maskRate While reading image data, Photoshop can resample it. Sampling rate values are entered as 32bit *fixed* point numbers. The integer part is stored in higher 16 bits, the fraction is in lower 16 bits. So if the plug-in module wants half-sized image (every second pixel), it sets `inputRate = (int32)(2 << 16)`.

inputPadding, outputPadding, maskPadding It may happen that plug-in will try to access data that are out of bounds of the image[3]. Padding is set to `plugInWantsErrorOnBoundsException` by default. It means that accessing pixels out of image bounds will cause an error and plug-in execution will be stopped. When we set padding to an arbitrary value from 0 to 255, all pixels outside image bounds will have this value. Setting `plugInDoesNotWantPadding` means that it is possible to access pixels out of the image, but their values will be entirely random. The most interesting option is `plugInWantsEdgeReplication`. This ensures that pixels at image borders will be copied beyond the bounds to extend the image as necessary. Padding may be very useful for example when applying a convolution mask. The programmer doesn't need to care about special cases at image borders and thus saves much work.

[3]It may be a programmer's mistake or also his intention.

wantsAbsolute The user may change the ordering of image channels in Photoshop. These must be then identified by the serial number. But the plug-in may also set `wantsAbsolute` entry. Photoshop will then pass image data in the standard order. Details about how image data is organised can be found in section 1.4.2.

inTileWidth, inTileHeight In order to optimize memory access, the plug-in should process the image in pieces. If possible it is best to work with block sizes suggested by Photoshop.

pointers to plug-in host functions Plug-in host provides plug-in modules with various utility functions. In Filter Parameter Block there are stored pointers to them. Some of them are direct and some are grouped in suites. For more about these functions please read section 1.6.

If you are going to write some considerably large plug-in, I certainly recommend you to read through the complete list of Filter Parameter Block entries. You may find it in the Adobe Photoshop API Guide. You will get a valuable knowledge about what the plug-in host may solve for you. Many times you will spare much programming.

1.4.1 Masking pixel modifications outside selection

In the input image there may be a non-rectangular selection. In such case plug-in will get to process the smallest rectangle that can hold the whole selected area. Also a mask will be available. From it the plug-in can determine which pixels are selected and which are not. The module needn't to care about the mask at all, since Photoshop masks all changes outside selection automatically. Only selected pixels will be modified, other changes are ignored. It is possible to disable this function so that plug-in can perform masking all by itself. However this is not very useful in practice.

Let's have a look at some better example when mask may be helpful. In some complicated image filters, processing each single pixel may be very time consuming. In this case it is worth first examining whether a pixel lies within a selection. We will then decide if the pixel needs processing or we can leave it alone. The mask may also be useful when we compute for example a histogram. In such case we should count only those pixels that are inside the selection. Photoshop automatically masks only writing, plug-in can read outside selection arbitrarily.

1.4.2 How image data is organized

Plug-in modules often process all image channels at once. In this case image data is interleaved. This means that in memory there are first stored all channels of

the first pixel consecutively, then all channels of the second pixel and so forth. If this kind of interleaving is for the plug-in unsuitable, it may be changed by setting the `wantLayout` entry of Filter Parameter Block. If standard interleaving is used channels are ordered is as follows: First all color components, then possible masks, finally possible alpha channels. This ordering may be however changed by the user. So it is useful to set `wantsAbsolute`. Photoshop will then ensure that image channels will be presented to the plug-in in the standard order.

1.5 Filter plug-in execution

As we already mentioned in section 1.3, while executing a plug-in the host calls module's entry point several times. The calling scheme is shown in Figure 1.1. The module performs operations according to the `selector` parameter. Following pages discuss all phases of plug-in execution in detail. We will refer to Filter Parameter Block as to FilterRecord for short. The same name has also the data structure where Filter Parameter Block is implemented.

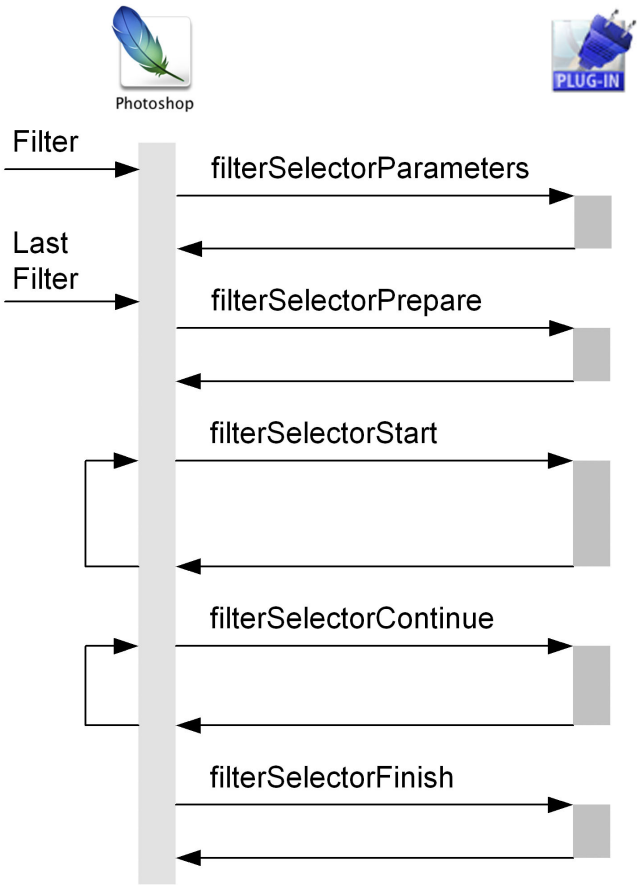

Figure 1.1: Filter plug-in calling scheme

We will start with a special case when the `selector` parameter is equal to `filterSelectorAbout` (value zero). It's a command for displaying an about box. In this case plug-in gets an AboutRecord instead of FilterRecord parameter. AboutRecord is a structure with basic information about the module. It can be used to show a simple informative dialog. In most cases this is fairly enough.

1.5.1 The Parameters phase

When the plug-in module is run for the very first time it is always with the `filterSelectorParameters` selector. Now the plug-in should allocate memory for parameters and set them to the default values. If the module stores its parameters in a configuration file or in Photoshop registry, it's a good time to load them now.

The plug-in will probably have some setup dialog. Therefore in the parameters structure there should be a flag that tells whether the dialog should be displayed. We will set the flag now because this is the first execution of the module so parameters need to be set up. The Parameters phase needn't to be called again later. Photoshop has a *Last Filter* function (keyboard shortcut Ctrl+F) that runs lastly used filter again with the same settings. Therefore plug-in parameters should be independent of image size or color space. Next time these values may be different. Moreover in the Parameters phase not all FilterRecord entries are valid, so the module doesn't know everything about the input image. Before the plug-in starts working it should always validate its parameters.

1.5.2 The Prepare phase

In the next step the plug-in is called with the `filterSelectorPrepare` parameter. This phase may (and may not) be preceded by the Parameters phase. If the module was run by the *Last Filter* command, the Parameters phase will be skipped. Not all FilterRecord entries are yet valid. But we already have `imageSize`, `filterRect` and `planes` available.

The plug-in should now estimate its memory requirements. Macintosh has poor memory management when allocating large blocks so Photoshop deals with it by itself. In FilterRecord there is a `maxSpace` entry where plug-in host indicates how many bytes of memory could be available for the plug-in. Take care, this memory need not to make up a single block. It is not efficient to always request all the free memory. It doesn't need to be free actually. Photoshop just says that it will be able to free it up if necessary. So if we want to avoid unnecessary swapping the plug-in should request only such amount of memory it will actually need. It will then inform Photoshop by setting `maxSpace` to appropriate value.

Memory requirements are usually estimated as follows. The module is going to process the image in pieces so we *estimate* the size of a single block. We can't use `inTileWidth` neither `inTileHeight` since these entries are not valid yet. The

number of pixels in a block is then multiplied by the number of image channels. The result is further multiplied by two since we have input data and output data. If there is a mask in the image we must count with additional space for it as well. In case of complicated algorithms it is sometimes not easy to estimate the amount of memory needed. If we don't want to bother we can just divide `maxSpace` by two. This way the plug-in will get half the memory available leaving the other half to Photoshop. This simple solution is often fully sufficient. But we may also experiment with `maxSpace` value. If program efficiency is critical it could be necessary to perform several tests and chose the best setting.

If the plug-in is going to allocate some additional tables, buffers or other large data structures it should set `bufferSpace` to the number of bytes needed. However it is more efficient to allocate memory using special functions provided by Photoshop for such situations. It's a task for Buffer suite which is discussed in detail later in section 1.6.2. When we use Buffer suite we leave `bufferSpace` set to zero and take advantage of sophisticated memory management in Photoshop.

1.5.3 The Start phase

The real filtering starts when plug-in is called with the `filterSelectorStart` parameter. First of all the module should validate it's parameters and in case of trouble return `filterBadParameters` error code. It should then verify that it can process image of given format. If not, the module should return `filterBadMode` error. According to PiPL, Photoshop enables plug-in execution only on suitable image formats but it is always better to make sure. We will discuss PiPL later in section 1.7. If the module is planning to use some plug-in host functions it should first verify whether these are available. Pointers to them must not be `NULL`. When needed functions are not provided by the host, the plug-in may try to get along without them or return `errPlugInHostInsufficient` error code.

When all tests passed well it's time to show the setup dialog. Of course only if it is needed. If the module was run "from the beginning" i.e., the Parameters phase was called, the flag for displaying setup dialog was set. When the user sets the parameters we will reset the flag. If the module would be later invoked with the *Last Filter* command, no setup dialog will be necessary. The user may of course change his mind about running the plug-in. He can hit the Cancel button. In such case the module should end with a `userCanceledErr` return value.

Next it is necessary to set up how the image will be processed. It's good to set `wantsAbsolute` to ensure standard channel ordering. We will probably want to sample the image 1:1 so we assign `inputRate = (int32)(1 << 16)`. If necessary we may also set `inputPadding` or other parameters.

The image will be processed in pieces. It's good because of lower memory requirements. Photoshop advises suitable block size in `inTileWidth` and `inTileHeight` entries. But we may chose other blocks that will be best suitable for us. For example we will compute horizontal motion blur by raster lines, while

for instance DCT for JPEG compression preferably by 8×8 blocks. For efficient computing on multiprocessor machines it is usually better to work with blocks rather than lines. Accessing memory by lines is often slower. The last thing to do is setting `inRect`, `outRect`, eventually `maskRect` rectangles to the first part of the image. Later during computing we must not forget that there may be padding bytes at the end of rows. We can find it out in FilterRecord from `inRowBytes`, `outRowBytes`, eventually `maskRowBytes` entries.

Now there are two possibilities how to continue. The conventional way is that we only make preparations in the Start phase. The image is then processed in the Continue phase which is being called repeatedly until the whole image is processed. Though this technique is not very efficient. In each turn the plug-in process just a single part of image. Then it returns to Photoshop which calls Continue phase again. So that program control still has to flip-flop back and forth between plug-in and the host. This causes excessive computation overhead.

Fortunately there is an elegant solution to this. We will use `advanceState` function. It updates the computing state (writes processed data and loads new input data) without having to return from plug-in call back to Photoshop. We shouldn't forget to always check the function's return value. If an error occurs, the plug-in should end with appropriate error code. It is also good to update the progress bar from time to time. The `progressProc` function does the job. Even when progress bar is growing the impatient user may decide to break the operation. This can be found out by the `abortProc` function. So it's also good to call it once in a while and test its return value. If user wants to halt, the module should stop computing and return `userCanceledErr` code. Photoshop will then automatically return the partially processed image into the state before plug-in invocation.

When the entire image has been processed the plug-in module sets `inRect`, `outRect` and `maskRect` rectangles (all of them) to zero. This way Photoshop will know that filtering has been finished. Setting to zero means setting to empty rectangles (0, 0, 0, 0), `NULL` value may be assigned as well.

1.5.4 The Continue phase

The module is called with `filterSelectorContinue` selector repeatedly until the whole image is processed. Plug-in will inform about it by zeroing `inRect`, `outRect` and `maskRect` rectangles. When we use `advanceState` technique, the Continue phase is actually not needed. Nevertheless it's good to respond to `filterSelectorContinue` for sure and zero all the rectangles.

If we don't use `advanceState` function we will process always a single image block in the Continue phase. We will then set rectangles to the next piece of image and return to Photoshop. If we have some unfinished work in the Continue phase we must remember that in case of error the Continue phase will not be called again. So we cannot rely that another Continue will always follow.

1.5.5 The Finish phase

After finishing the main operation, plug-in is called with `filterSelectorFinish` parameter. In this phase the module should do any clean-up needed. It should release allocated memory[4] and put everything else to the state where it's possible to end correctly. If there was no error in the Start phase, plug-in host guarantees that Finish will be called in the end. This applies even if there was some error in the Continue phase. But if an error occurs during the Start phase it is necessary to do any clean-up right there, Finish will not be called.

1.6 Plug-in host functions

The Filter Parameter Block Structure contains among others also pointers to function callbacks provided to plug-in module by the host. We will discuss these functions in detail in this section. First it should be reminded that not all of them must be always available. Those which are not supported by the plug-in host will have a `NULL` pointer in the FilterRecord. Before the module starts using a function it should first verify that plug-in host provides this function. If not, the module will have to get along without it. If the function is a must, plug-in should stop and return `errPlugInHostInsufficient` error code.

1.6.1 Directly accessible functions

Pointers to most often used functions are stored directly as FilterRecord entries. We will describe the most important ones in following paragraphs.

```
OSErr advanceState(void)
```

Upon calling this function, plug-in host updates all data structures used to communicate with plug-in module. In practice it means that a new block of input image will be loaded into `inData` according to `inRect` setting. Results in `outData` will be stored and according to `outRect` a new memory will be prepared for future results to come.

While using `advanceState`, communication between plug-in and the host is far more efficient. Unlike calling the Continue phase repeatedly, program control needn't to switch back and forth. If everything goes well, `advanceState` returns `noErr`. In case of error it returns appropriate error code. This applies even if the user wants to break plug-in operation (he pressed the Escape key). If the function ends up with an error, some FilterRecord entries may be invalid, so the module should stop computing. A typical `advanceState` call looks like this.

[4] Of course except module's parameters. These will be preserved until next plug-in execution.

```
*gResult = gFilterRecord->advanceState();
if (*gResult != noErr)
    return;
```

The `gFilterRecord` pointer holds a reference to the Filter Parameter Block. The `gResult` points to a global variable where an error code may be stored. This code will be passed to plug-in host when returning from plug-in entry point call.

`Boolean abortProc(void)`

This function should be called preferably several times per second. If it returns `true` it means that the user wants to break plug-in execution. As a side effect the function changes mouse cursor to hourglass.

`void progressProc(long done, long total)`

Plug-in should call this function once in a time to update progress bar during long operations. The first parameter `done` tells how many work units have been done so far. Second parameter `total` specifies how much work is there in total. The `progressProc` function should be used only during the main plug-in operation, not when e.g. computing a preview.

`OSErr colorServices(ColorServicesInfo *info)`

Provides services concerned with colors. There are four options to chose from. The function can show a palette and let the user select a color. The most frequently used service is color space conversion. Further it is possible to get current sample point. And finally we can read current foreground or background color.

Before `colorServices` can be used we have to fill in the `ColorServicesInfo` structure. A simple example follows showing how to convert color from image space to RGB.

```
ColorServicesInfo info;
info.infoSize = sizeof(info);
info.selector = plugInColorServicesConvertColor;
info.sourceSpace
    = CSModeToSpace(gFilterRecord->imageMode);
info.resultSpace = plugInColorServicesRGBSpace;
for (int a = 0; a < 4; a++)
    info.colorComponents[a] = color[a];

if (gFilterRecord->colorServices(&info) == noErr)
    for (int b = 0; b < 4; b++)
        color[b] = (uint8)info.colorComponents[b];
```

The `info` structure size is may be reasonless assigned to the `infoSize` entry. It's a trick how to easily ensure version numbering. The `selector` specifies which service we want to use. In our example it is a color space conversion. We set the `sourceSpace` parameter to the image color space. The `imageMode` value from FilterRecord has to be converted to a `colorSpace` value. Therefor we use the `CSModeToSpace` function. We will then set the `resultSpace` to RGB. Lastly we just copy the color we want to convert into the `colorComponents` array.

The `ColorServicesInfo` structure is ready, we can call the `colorServices` function. If everything went all right the converted color is now stored in the `colorComponents` array. Take care while reading the color components. They must be cast to `uint8` since `colorComponents` array is of type `int16`! If plug-in host supports it, from `info.resultInGamut` we may find out whether resulting color is inside gamut for current printer setting. This is possible as well when using the other services not just color space conversion.

```
OSErr displayPixels(const PSPixelMap *source,
    const VRect *srcRect, int dstRow, int dstCol,
    unsigned platformContext)
```

This function serves for drawing image data into a dialog window. Thus it is useful for displaying a preview. Plug-in host automatically performs color space conversions as necessary. The image is then drawn to given place in the dialog. The use of `displayPixels` is a bit more complicated and needs some knowledge of Win32API. So interested readers should consult the Adobe Photoshop SDK documentation and sample programs.

1.6.2 Buffer suite functions

Additional functions are grouped together into suites according to common functionality. The Buffer suite could be perhaps the most useful. It offers possibility to use Photoshop's efficient memory management. Photoshop was originally created for Apple Macintosh, where allocation of large blocks is not very efficient. So Photoshop has its own memory management. When plug-ins want to take advantage of it they can use the Buffer suite. It's functions can be accessed through the `bufferProcs` entry in FilterRecord. Let's have a look at individual functions.

```
int spaceProc(void)
```

Returns the number of free bytes available. The free memory needn't to be a continuous block.

`OSErr allocateProc(int size, BufferID *buffer)`

Allocates a memory block of given size. The block identifier is returned as the `buffer` parameter.

`void freeProc(BufferID buffer)`

Releases given block of memory.

`Ptr lockProc(BufferID buffer, Boolean moveHigh)`

Locks specified block thus preventing it from moving in memory. The `moveHigh` parameter is significant only on Macintosh. It determines whether the block being locked should be moved to the high end of memory to reduce fragmentation. The function returns a pointer to the beginning of locked block. Take care, numbers are stored as `int8`. So when reading image data we must not forget to cast it to `uint8`.

`void unlockProc(BufferID buffer)`

Unlocks given memory block. Buffer suite uses a cumulative lock so one block may be locked several times. To unlock it, the same number of `unlockProc` calls is necessary.

1.7 The PiPL resource

Originally Adobe Photoshop recognized plug-ins by their filename extension, eventually by file type on Macintosh. In Photoshop version 3.0 the plug-in interface was considerably revised. Many new functions and possibilities have come. From Photoshop version 3.0 plug-in information are extracted from so-called PiPL (Plug-in Property List). It is however recommended to keep standard filename extensions. Plug-in Property List is a set of information added to a plug-in as a (platform dependant) resource. Plug-in host can read module type, abilities and static settings from PiPL. On the other hand the module can specify its relation to various hosts.

A PiPL is usually written in a *Rez* format that originates from Macintosh. It's highly recommended to keep this format even when developing for Windows. As mentioned many times before, Photoshop comes from Macintosh and many things are affected by this. It is necessary to comply with byte ordering (Photoshop uses big endian), padding bytes, etc. There is a simple utility to convert PiPL to a Windows resource. Detailed description can be found in section 1.8.2.

The PiPL structure is very complex and there's not point in studying it in detail. There is a comprehensive description in Adobe Photoshop SDK documentation. But at the same place there is recommended to use a PiPL from some sample project as a template and then just fine tune necessary settings. So now a typical PiPL will be shown with detailed description of all used entries.

```
resource 'PiPL' (ResourceID, "Invert PiPL", purgeable)
{ {
    Kind {Filter},
    Category {"Invert"},
    Name {"Invert..."},
    Version {(latestFilterVersion << 16)
                | latestFilterSubVersion},
    CodeWin32X86 {"PluginMain"},

    SupportedModes
    {
        noBitmap, doesSupportGrayScale,
        noIndexColor, doesSupportRGBColor,
        doesSupportCMYKColor, ...
    },

    EnableInfo
    {
        "in (PSHOP_ImageMode, GrayScaleMode,"
            "RGBMode, CMYKMode, ...)"
    },

    FilterCaseInfo
    { {
        inStraightData, outStraightData,
        doNotWriteOutsideSelection,
        doesNotFilterLayerMask,
        doesNotWorkWithBlankData,
        doNotCopySourceToDestination,
        ...
    } }
} };
```

There is a `resource` keyword at the first line. Follows a 4byte code `'PiPL'` determining type. In parentheses there are three parameters. The symbolic constant `ResourceID` is resource identifier. The `"Invert PiPL"` string is a name, it can be arbitrary. The last parameter is a `purgeable` keyword. That means

if Photoshop would run out of memory it can remove this resource if it's not actually needed.

Now we will describe all used PiPL entries. Their ordering in the source code doesn't matter. The `Kind` attribute designates module type. When we are developing a Filter plug-in we will use the `Filter` keyword. The `Category` determines into which category the module belongs to. *This is the name of submenu in Photoshop Filter menu where the plug-in will show up.* Obviously the `Name` attribute is a name of the module. *The same name will also have appropriate submenu entry in the Filter menu.* The `Version` attribute is not module version. It's a version of SDK which was used to build the plug-in. The value is a 32bit number with higher 16 bits for major version and lower 16 bits for minor version. Plug-in host gets module's entry point name from the `CodeWin32X86` attribute.

The plug-in might not be able to work with all possible image modes. This is determined by `SupportedModes` and `EnableInfo`. Photoshop will enable or disable plug-in invocation accordingly. Though we shouldn't blindly rely on this. A correct plug-in always checks color space of the image it's going to process. The `SupportedModes` attribute contains just a list of color modes. The `doesSupport` prefix means that the plug-in is capable of processing such color space, prefix `no` means the opposite.

The `EnableInfo` entry contains an expression. If it's evaluated as true, plug-in will be enabled. The expression can be quite complicated so that we can check many image parameters, e.g. bit depth. But mostly we get along just with the `in` function. It determines whether it's first parameter is equal to at least one of those following. The `PSHOP_ImageMode` parameter represents color space of current image. Complete description of `EnableInfo` expression possibilities may be found in SDK documentation. Attributes `SupportedModes` and `EnableInfo` should always match, they should produce the same results for given image mode.

Specially for Filter plug-ins there is the `FilterCaseInfo` attribute. It contains a list of settings which specify how image data will be presented to the plug-in. Seven cases may occur depending on whether the image contains transparency, selection and/or layer mask. Settings for all those cases are stored in one list consecutively so here the *ordering is significant*. In the above PiPL example there are settings just for the first case i.e., image with no transparency and no selection. Remaining variants may be found in the documentation and in sample projects.

In each case the setting has six parameters. First two of them control preprocessing and postprocessing of image data. Keywords `inStraightData` and `outStraightData` means that plug-in will get the data as is. Neither after processing, the host will do any extra adjustments. Alternatively Photoshop may for example fill transparent areas by defined color. When processing is done, the color can be removed so as to restore the transparency. Please consult documentation for details.

The next attribute `doNotWriteOutsideSelection` specifies that Photoshop will automatically mask modification of pixels that lie outside selection. Choosing `writeOutsideSelection` disables this feature.

The `doesNotFilterLayerMask` means that plug-in module will not process layer mask. The opposite option is `filtersLayerMask` which ensures that the mask will be added to other image channels.

Setting `doesNotWorkWithBlankData` indicates that the module is not capable of working with completely transparent pixels. They actually don't have any color defined. If such pixels would appear in the image, an error will be signaled. If plug-in can handle transparent pixels, we will set `worksWithBlankData`.

The `doNotCopySourceToDestination` attribute specifies that plug-in host will not copy input pixels to the output image automatically. Thereby we spare some computational time. However if the plug-in module is going to modify just a few pixels, it's better to set `copySourceToDestination`. This way plug-in won't have to copy all the unaltered pixels.

1.8 Writing plug-ins in Microsoft Visual Studio

Compiling a plug-in module might be a bit tricky so here is a step by step guide. Microsoft Visual Studio (also referred to as MSVS further in text) is probably the most widespread development environment on Windows. Therefore following instructions are right for MSVS.

1.8.1 Setting up the project

Project settings may be found in the *Project* | *Properties...* menu. First of all the module should compile as DLL (Dynamic Link Library). This is preferably set already when creating the project. It can also be done later in the *General* tab as shown in Figure 1.2. When we are developing a Filter plug-in the resulting file should have a `.8BF` extension. This can be set in the *Linker* | *General* tab as shown in Figure 1.3. It's also necessary to specify directories where Adobe Photoshop SDK header files are located. We can set it in *C/C++* | *General* tab, see Figure 1.4.

If we use any utility functions from SDK samples, we have to include source files, where these functions are programmed, to the project. Figure 1.5 shows an example. Mentioned source files are located in the *Utilities* folder but this is not decisive. They may also be in the standard *Source Files* folder as well.

To compile all utility functions successfully it may be necessary to set the project to link with `version.lib` library. We can set it up in the *Linker* | *Input* tab. There's an example shown in Figure 1.6.

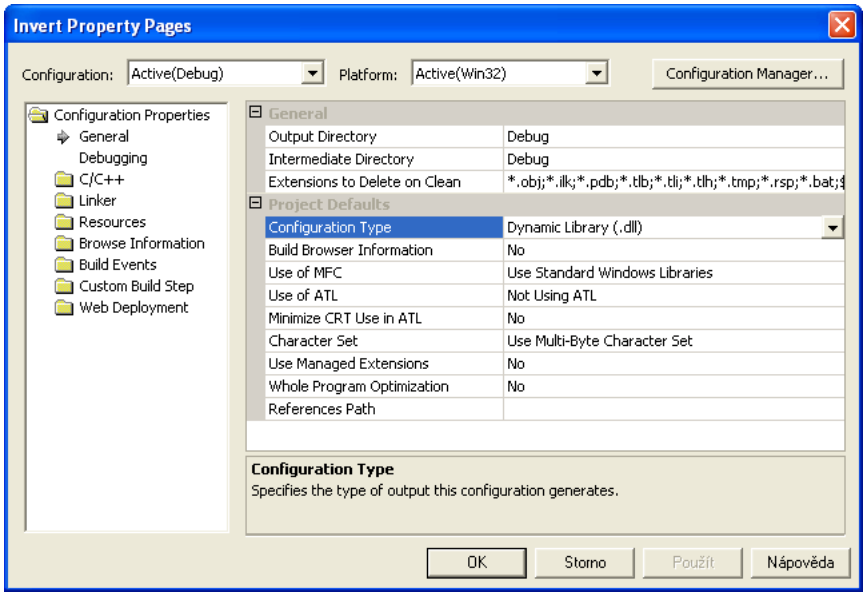

Figure 1.2: Setting compilation preferences

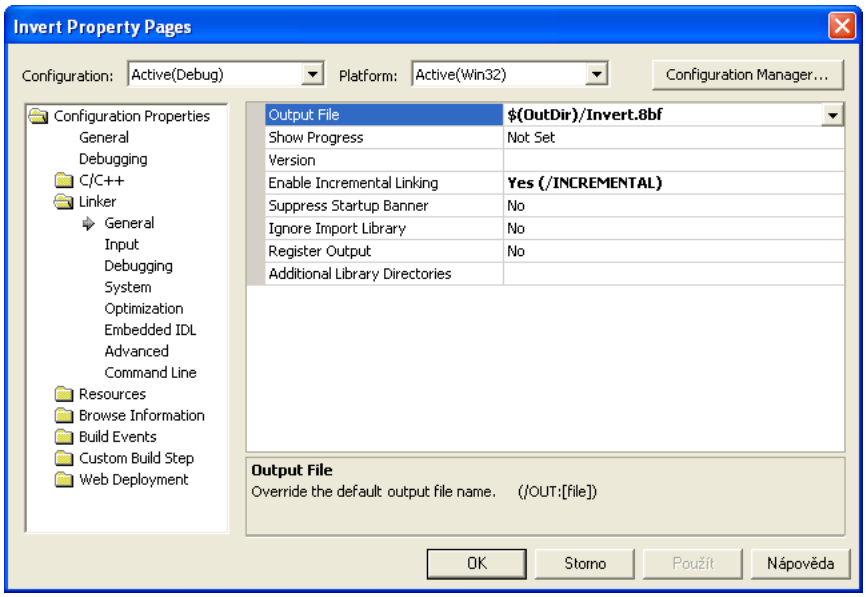

Figure 1.3: Setting output filename extension

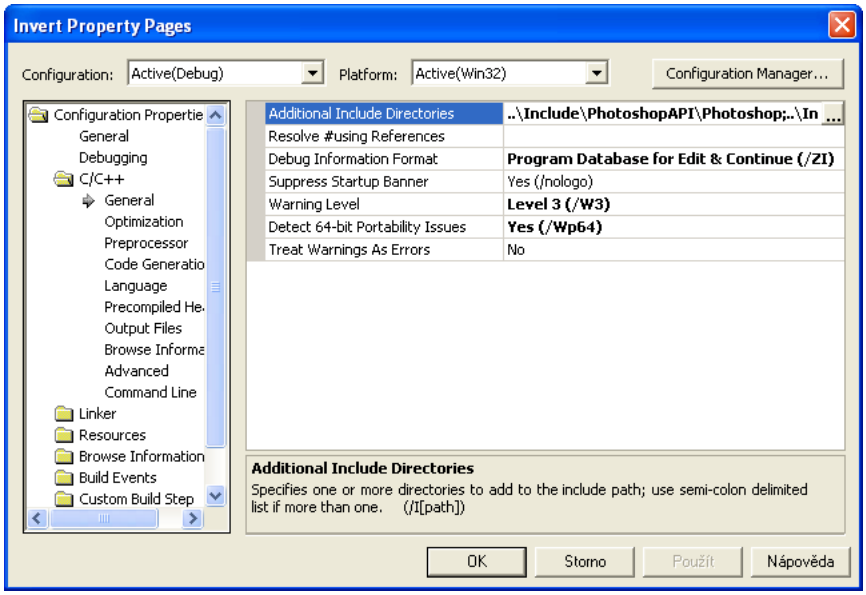

Figure 1.4: Designation of directories with header files

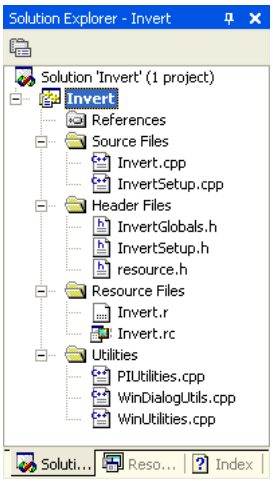

Figure 1.5: An example of arranging files into the project

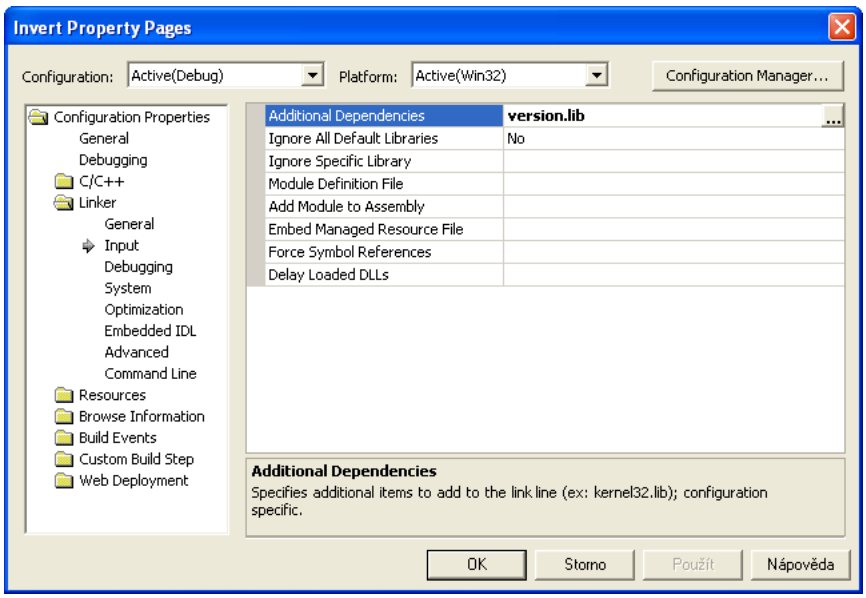

Figure 1.6: Setting the linker

1.8.2 Compiling PiPL resource

We will also add the PiPL source file (it usually has a `.r` extension) into the project. Situation is illustrated in Figure 1.5. We have to use a Custom Build Step to compile a PiPL since it's in the Rez format that MSVS doesn't recognize. Therefore we must exactly specify how to handle such file. We can open the Custom Build Step options by selecting the `.r` file and choosing *Project* | *Properties* from the menu. A dialog similar to Figure 1.7 will appear.

First we have to prepare the PiPL source file by passing it through a standard C preprocessor.

```
cl.exe /I<file1.h> /I<file2.h> /EP /DWIN32=1 /Tc
    "$(InputPath)" > "$(IntDir)\$(InputName).rr"
```

The cl.exe program is a standard C compiler. All necessary header files are specified as parameters with `/I` prefix. The `/EP` switch runs the preprocessor. The result will be written to the standard output. Further it's necessary to define the `WIN32` symbolic constant, this is done by `/DWIN32=1`. The `/Tc` switch indicates that the input file is in the C language. The `$(InputPath)` macro stands for input filename. Preprocessor's standard output is redirected to a file with `.rr` extension.

We will now compile the preprocessed file by a utility program from Adobe Photoshop SDK.

```
Cnvtpipl.exe "$(IntDir)\$(InputName).rr" "$(InputName).pipl"
```

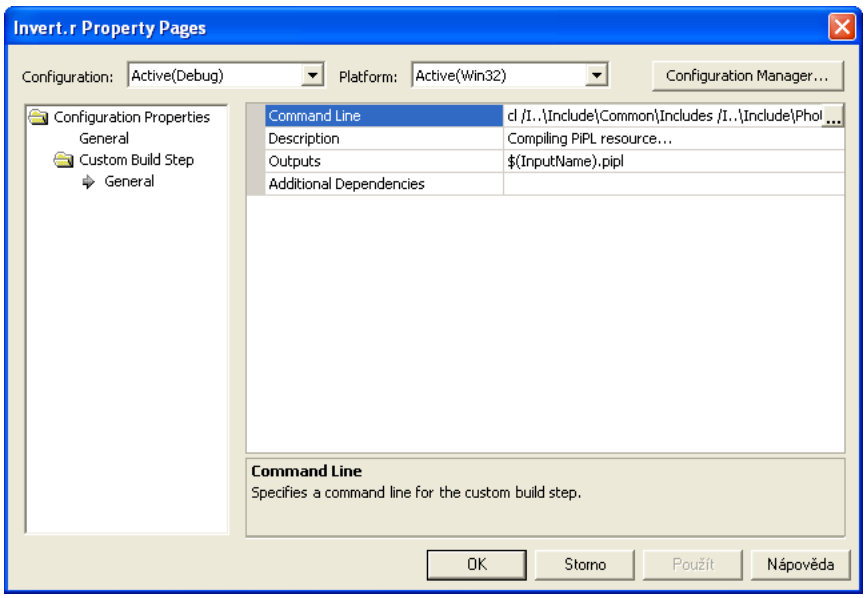

Figure 1.7: Setting a Custom Build Step for PiPL resource

The first parameter is input filename, the second one is output filename. As a result we will get PiPL as Windows resource. Unfortunately programs can't be placed in a pipe since Cnvtpipl.exe doesn't read data from standard input.

Lastly we add some resource to the project. The resource may be empty but we will probably have a setup dialog definition. In the *Resource View* window we then invoke (by the right mouse button) a context menu for the resource and we select *Resource Includes...* There's an example in Figure 1.8. A dialog window will show up. In the *Compile-time directives* field we include the PiPL into the resource by `#include` command. Figure 1.9 shows an example.

1.8.3 Debugging a plug-in module

For comfortable debugging it's good to automatically copy the compiled module into a Photoshop directory. A Post-Build Even is useful for this. We can set it up in the *Build Events* | *Post-Build Event*. There's an example in Figure 1.10. May be the `$(TargetPath)` macro is worth explaining. It represents the resulting filename. When updating an existing plug-in there is no need to restart Photoshop. Before any module starts it's always reloaded in a current form.

Of course we can debug plug-ins using the standard Visual Studio debugger. When running the project for the first time, MSVS will ask in which application the library should be debugged. So we chose Photoshop.exe. The setting may be later revised in the *Debugging* tab as shown in Figure 1.11.

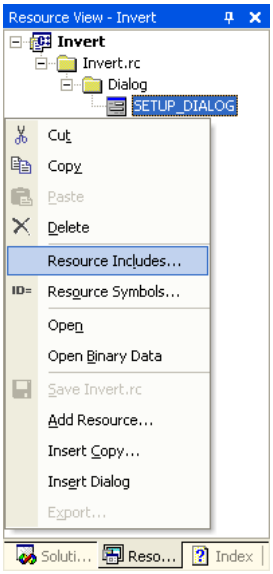

Figure 1.8: Context menu for a resource

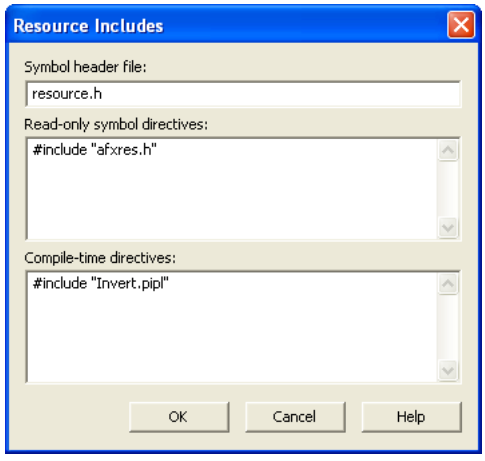

Figure 1.9: Including a PiPL into a resource

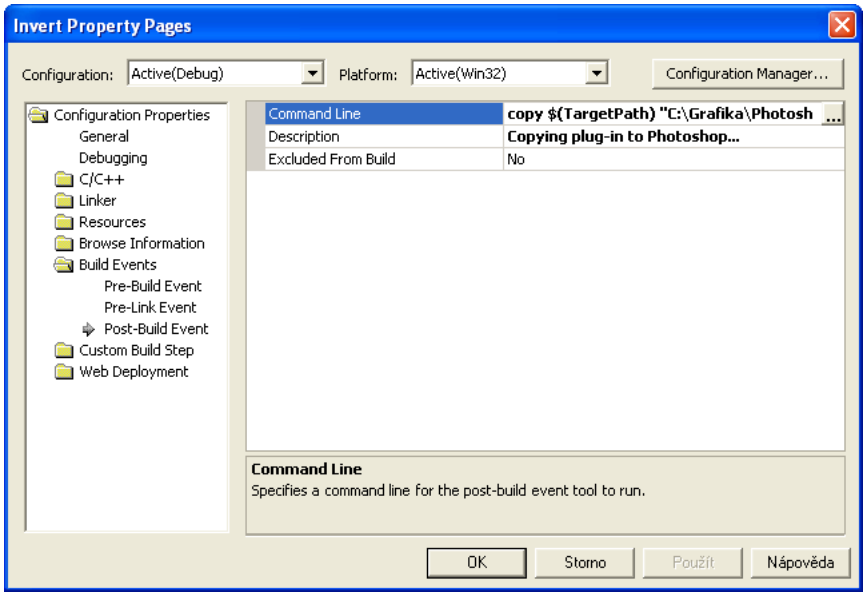

Figure 1.10: Copying compiled module to Photoshop

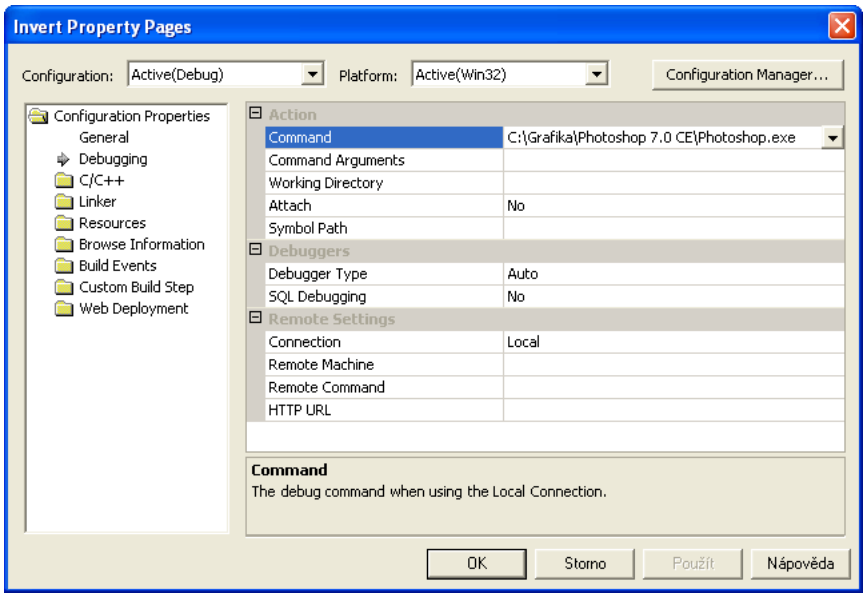

Figure 1.11: Choosing application where to debug the module

Let's have a brief look at some of the common problems you may encounter while writing a plug-in. If Photoshop does not recognize a module at all, the problem is most probably with PiPL. If the project was compiled successfully, PiPL was probably not included correctly to the resulting program. When processing images we use arrays very often. Hence if we are looking for an error in the program, it's probable that some index exceeded out of array bounds. When writing plug-ins we must remember that not all images are in RGB color space. The module should be tested with various image formats, with and without selection, including a non-rectangular selection.

1.9 Adobe Dialog Manager

Adobe Dialog Manager (further referred to as ADM) is a platform independent interface for managing dialog windows in Adobe applications. Dialogs created by ADM have consistent Adobe look and feel. May be you are now wandering whether ADM is worth using or it's better to use Win32API. I would recommend you the next. If you have a good knowledge of Win32API, use it. You will have full control over all components. Program portability to Macintosh can be the last thing to trouble you. If Win32API is unfamiliar to you, I would recommend you to consider learning at least the basics. It's by far not an anachronism[5] and you will make use of it certainly more often than ADM. However if you like discovering new things, let's start revealing the ADM. It isn't difficult and in contrast to Win32API it's quite comfortable.

For those not discouraged by the previous paragraph a light description of ADM follows. The entire interface is very extensive so we will discuss just the most important functions. Those should be fully sufficient for creating basic dialogs. Further details may be found in the documentation and sample projects. First we will need a dialog design. This is the only platform dependent part. Today there are many graphic applications where a dialog can be easily designed. Microsoft Visual Studio is probably the most frequently used on Windows. Arranging components onto a form isn't anything complicated so we won't deal with it here. Adobe Dialog Manager loads dialog design from a resource. All common components work well. However we should be careful when using some specialities that ADM might not recognize.

Adobe Dialog Manager offers a range of functions that are grouped to suites according to application field. Every such suite has to be acquired before use and released at the end. The whole ADM is implemented according to PICA (Plug-In Component Architecture). In Filter Parameter Block there is a `sSPBasic` entry where we can find basic functions for using PICA. The following example shows how to acquire ADM suite for working with dialogs.

[5]Originally I thought, as many others did, Win32API is a prehistoric thing. Later I was surprised how many times it was useful to me.

```
gFilterRecord->sSPBasic->AcquireSuite(
    kADMDialogSuite, kADMDialogSuiteVersion, (void**)&sADMDialog);
```

The `gFilterRecord` variable contains a pointer to the Filter Parameter Block. The `kADMDialogSuite` constant determines which ADM suite we want to acquire, `kADMDialogSuiteVersion` is its version. A pointer to the acquired suite will be assigned to `sADMDialog`. Releasing a suite is even more easy.

```
gFilterRecord->sSPBasic->ReleaseSuite(
    kADMDialogSuite, kADMDialogSuiteVersion);
sADMDialog = NULL;
```

It isn't necessary to explain anything more here. No thorough programmer would forget the command at the last line.

1.9.1 Creating a dialog window

Now we will finally learn how to put ADM functions to practice. First of all we have to create a dialog. This can be done using a function from ADM Dialog Suite.

```
int dismissButton = sADMDialog->Modal(
    (SPPluginRef)gFilterRecord->plugInRef, "SetupDialog",
    SETUP_DIALOG, kADMModalDialogStyle, DialogInit, NULL);
```

The `Modal` function creates and shows a modal dialog window. Modal means that the user can't return to the main window of application until he closes the dialog. Modal windows is exactly what we need in plug-ins. Photoshop not even supports other ADM dialog types. The first parameter of the function is a plug-in module reference. We get it from the Filter Parameter Block. Second parameter is dialog name for ADM internal use. The window caption is determined by design loaded from a resource. When designing a dialog, every component and even the window itself gets a unique identifier. Appropriate dialog definition will be found in the resource according to the `SETUP_DIALOG` constant. Another constant `kADMModalDialogStyle` means that a modal dialog will be created. The `DialogInit` is a name of initialization function. It will be called immediately upon dialog creation. This function is described later. The last parameter of the `Modal` function is a pointer to data that should be bundled with the dialog. This data can be later accessed using the `sADMDialog->GetUserData` function. The `Modal` function returns identifier of the component that was used to dismiss the dialog. It's usually either the OK button or the Cancel button. This way we can easily find out whether the user affirmed settings or he wants to cancel.

The initialization function serves for initial setting of the dialog and its components. Adobe Dialog Manager creates all components and positions them at the correct place. So we usually just set them to the state corresponding to actual

plug-in parameters. In documentation you can find for each component an exact description of what all should be initialized. The initialization function has the following prototype.

```
ASErr ASAPI DialogInit(ADMDialogRef dlgRef)
```

The `ASAPI` macro is just another name for Pascal (stdcall) calling convention. The parameter is a reference to the dialog just created. The function returns an error code. If everything was all right the code should be `kNoErr`.

1.9.2 Accessing dialog components

It's time to learn how to manipulate individual dialog components. First we have to get the component reference.

```
item = sADMDialog->GetItem(dlgRef, ID_CHECKBOX);
```

The `dlgRef` parameter is a reference to the dialog window where the desired item is located. The `ID_CHECKBOX` identifier designates the component we want. We can then manipulate it using ADM Item Suite functions. The example below shows how to set a check box.

```
sADMItem->SetBooleanValue(item, true);
```

Adobe Dialog Manager may look a bit confusing for the first time since ADM Item Suite functions are designed very generally. For example in this case the `SetBooleanValue` function sets a check box. But when applied to other component type it may have different meaning. In documentation you can find for each function explained what effect is has on which component. The ADM Item Suite has about 140 functions. So it's up to the reader to find those he will need.

1.9.3 Event handling

The dialog window should of course respond to user input. This is, as everywhere else, ensured by a system of events. For accessing events at a higher level there is so-called *notifier*. It is called when user finishes interaction with a component. Let's have a look at a simple example. Consider a track bar. User can grab the slider by mouse and drag it. As he scrolls to desired position he releases the slider. At this time a notifier will be called.

In the previous section it was explained how dialog component properties can be set up. We assign a notifier in the same manner.

```
item = sADMDialog->GetItem(dlgRef, ID_OK);
sADMItem->SetNotifyProc(item, ButtonOkNotify);
```

The meaning of all functions and parameters should be clear now. The OK button events will be handled by the `ButtonOkNotify` function. The following example shows its general scheme.

```
void ASAPI ButtonOkNotify(
    ADMItemRef itemRef, ADMNotifierRef notifier)
{
    sADMItem->DefaultNotify(itemRef, notifier);
    if (sADMNotifier->IsNotifierType(
        notifier, kADMUserChangedNotifier))
    { ... }
}
```

The function prototype must comply. The `itemRef` parameter is a reference to the component that caused this event. This way one function can handle events of several components. The `notifier` parameter is a pointer to information about current event.

In the function body we should first call the default event handler. This can be done by the `sADMItem->DefaultNotify` function. Usually a conditional statement follows where we determine what type of event occurred. Thereby we can chose to which events we will respond. The `sADMNotifier->IsNotifierType` function has two parameters. The first one is a pointer to the event, the second one is a symbolic constant determining what kind of event we want to handle. If such event occurred, the `IsNotifierFunction` returns `true`.

For accessing events at a lower level we can use so-called *tracker*. Consider again the example with a track bar. As explained, the notifier will be called when the user releases the slider. On the other hand the tracker will be called as soon as user grabs the slider. It will be then called every time the slider moves by a single tick. And for the last time when the slider is released. All these events are placed into a queue. So if their service would take a long time it's guaranteed they will be processed in the same order as they occurred.

Tracker usage is similar to notifier. A tracker is assigned to a component by a `sADMItem->SetTrackProc` command. The event handling function may then look like the example below.

```
ASBoolean SliderTrack(ADMItemRef itemRef, ADMTrackerRef tracker)
{
    ASBoolean sendNotify
        = sADMItem->DefaultTrack(itemRef, tracker);
    if (sADMTracker->TestAction(
        tracker, kADMMouseMovedDownAction))
    { ... }
    return sendNotify;
}
```

At the beginning we will call the default event handler as well. It will return information whether current event requires calling a notifier. So when dragging the track bar slider it will return `false` while when the slider is released it will return `true`.

After returning from the default handler we usually test what kind of event occurred. This time we use the `sADMTracker->TestAction` function to it. A `kADMMouseMovedDownAction` constant is used in the example above. It doesn't mean we respond just in case when mouse moved downwards. It stands for the event when a mouse moved with a button pressed (this is sometimes called mouse dragging). In Win32API we would have to check separately whether the mouse moved and whether a button is pressed. The handler function should return information whether a notifier should be called. We will generally use the return value of the default event handler but we can also control when notifier will be called.

Chapter 2

Theory of masking images

2.1 Image registration

When we want to work with two or more pictures of the same or similar scenes we first have to find a correspondence between pixels of individual images. We don't concern about synthetically generated images in this work since there is usually no need to perform any masking. Our main objective are pictures taken from a digital camera. It is supposed that pictures will have good quality, that means that they are well lit, focused and that the camera didn't move between individual shots. Especially the last requirement is not easy to assure. Of course a tripod should be used when taking pictures, though this is often not enough since even in this case the camera can move slightly.

To achieve reasonable results from further processing we first have to find a match between images. That is to determine which pixels of the first image correspond to which pixels in the second image. This step is often called image registration. In the research literature methods for automatic image matching fall broadly into two categories: direct and feature based. Feature based methods begin by establishing correspondences between points, lines or other geometrical entities. For example, a typical approach would be to extract Harris corners and use a normalised cross-correlation of the local intensity values to match them. Direct methods attempt to estimate the registration by minimising an error function based on the image intensity difference.

Feature based methods are capable of successfully registering images even with high differences in camera position including rotation and zoom. However the algorithms are rather complicated and require considerable computational time. Direct methods have the advantage that they use all of the available data and hence can provide very accurate registration. They usually assume just small changes between the images being registered. But then algorithms are quite fast and simple to implement.

2.1.1 Scale Invariant Feature Transform

Recently there has been great progress in the use of invariant features for object recognition and matching. In this section we discuss Scale Invariant Feature Transform (SIFT features) as presented by D. G. Lowe in [4]. These features are geometrically invariant under similarity transforms and invariant under affine changes in intensity. This method can extract distinctive invariant features from images that can be used to perform reliable matching between different views of an object or scene.

The features are invariant to image scale and rotation, and are shown to provide robust matching across a substantial range of affine distortion, change in 3D viewpoint, addition of noise, and change in illumination. The features are highly distinctive, in the sense that a single feature can be correctly matched with high probability against a large database of features from many images. Following are the major stages of computation used to generate the set of image features.

1. *Scale-space extrema detection*: The first stage of computation searches over all scales and image locations. It can be implemented efficiently by using a difference-of-Gaussian function to identify potential interest points that are invariant to scale and orientation.

2. *Key point localization*: At each candidate location, a detailed model is fit to determine location and scale. Key points are selected based on measures of their stability.

3. *Orientation assignment*: One or more orientations are assigned to each key point location based on local image gradient directions. All future operations are performed on image data that has been transformed relative to the assigned orientation, scale, and location for each feature, thereby providing invariance to these transformations.

4. *Key point descriptor*: The local image gradients are measured at the selected scale in the region around each key point. These are transformed into a representation that allows for significant levels of local shape distortion and change in illumination.

This approach has been named the Scale Invariant Feature Transform (SIFT), as it transforms image data into scale-invariant coordinates relative to local features.

2.1.2 Direct image registration

Given two input images A and B we are able to compute a mean square error (MSE) between them as

$$MSE = \frac{1}{xy} \sum_{i=0}^{x-1} \sum_{j=0}^{y-1} (A[i,j] - B[i,j])^2, \tag{2.1}$$

where x and y are image dimensions. If we have images with multiple channels we usually interpret color components as coordinates in space. Pixel difference is then computed as Euclidean distance.

When registering images directly we have to first specify which image transformations we are going to take into account. In most cases it will be translation (in x and y axes), we may add rotation, zoom, etc. We then search for such transformation parameters that minimize the MSE. We have to specify some bounds i.e., set some range and sampling rate for parameter values. This designates the search space. The algorithm is then quite simple. A MSE is computed for each possible combination of transform parameters. Those that yield least MSE value are the best solution for registration.

2.2 Alpha channel

As described in [8] a separate component is needed to retain the matte information, the extent of coverage of an element at a pixel. In a full color rendering of an element, the RGB components retain only the color. In order to place the element over an arbitrary background, a mixing factor is required at every pixel to control the linear interpolation of foreground and background colors. In general, there is no way to encode this component as part of the color information. For anti-aliasing purposes, this mixing factor needs to be of comparable resolution to the color channels. Such mixing factor is called an *alpha channel*, while an alpha of 0 indicates no coverage, 1 means full coverage, with fractions corresponding to partial coverage.

In an environment where the compositing of elements is required, we see the need for an alpha channel as an integral part of all pictures. Because mattes are naturally computed along with the picture, a separate alpha component in the frame buffer is appropriate. Off-line storage of alpha information along with color works conveniently into run-length encoding schemes because the alpha information tends to abide by the same runs.

Each pixel is then described by a quadruple (r, g, b, a). How do we express that a pixel is half covered by a full red object? One obvious suggestion is to assign (1, 0, 0, 0.5) to that pixel. The 0.5 indicates the coverage and the (1, 0, 0) is the color. There are a few reasons to dismiss this proposal, the most severe being that all compositing operations will involve multiplying the 1 in the red channel

by the 0.5 in the alpha channel to compute the red contribution of this object at this pixel. The desire to avoid this multiplication points up a better solution, storing the *premultiplied* value in the color component, so that (0.5, 0, 0, 0.5) will indicate a full red object half covering a pixel.

The quadruple (r, g, b, a) indicates that the pixel is covered by the color (r/a, g/a, b/a). A quadruple where the alpha component is less than a color component indicates a color outside the [0, 1] interval, which is somewhat unusual. Though luminescent objects can be usefully represented in this way. For the representation of normal objects, an alpha of 0 at a pixel generally forces the color components to be 0. Thus the RGB channels record the true colors where alpha is 1, linearly darkened colors for fractional alphas along edges, and black where alpha is 0. Silhouette edges of RGBA elements thus exhibit their anti-aliased nature when viewed on an RGB monitor. It is important to distinguish between two key pixel representations: black = (0, 0, 0, 1) and clear = (0, 0, 0, 0). The former pixel is an opaque black, the latter pixel is transparent.

The color of the composite can be computed on a component basis by adding the color of the picture A times its fraction to the color of picture B times its fraction. To see this, let c_A, c_B, and c_O be some color components of pictures A, B and the composite. Let C_A, C_B, and C_O be the true color components before premultiplication by alpha. Then we have $c_O = \alpha_O C_O$. Now C_O can be computed by averaging contributions made by C_A and C_B, so

$$c_O = \alpha_O \frac{\alpha_A F_A C_A + \alpha_B F_B C_B}{\alpha_A F_A + \alpha_B F_B}. \tag{2.2}$$

But the denominator is just α_O, so

$$c_O = \alpha_A F_A C_A + \alpha_B F_B C_B = \alpha_A F_A \frac{c_A}{\alpha_A} + \alpha_B F_B \frac{c_B}{\alpha_B} = c_A F_A + c_B F_B. \tag{2.3}$$

Factors F_A and F_B are determined by the compositing operation. The most common is the *over* operation when foreground A is placed over background B. In this case $F_A = 1$ and $F_B = 1 - \alpha_A$ resulting in

$$c_O = c_A + c_B(1 - \alpha_A). \tag{2.4}$$

Because each of the input colors is premultiplied by its alpha, and we are adding contributions from non-overlapping areas, the sum will be effectively premultiplied by the alpha value of the composite just computed. The pleasant result that the color channels are handled with the same computation as alpha can be traced back to the decision to store premultiplied RGBA quadruples. Thus the problem is reduced to finding fractions F_A and F_B which indicate the extent of contribution of A and B. When we want to blend overlapping images together we have to introduce a dissolve factor α. The composite is then computed as

$$c_O = c_A \alpha + c_B(1 - \alpha). \tag{2.5}$$

2.3 Constant color matting

The basic problem of constant color matting is: Given an image of a foreground object shot in front of a constant backing color, obtain a matte (alpha channel) so that the object can be blended onto a new background.

2.3.1 Blue screen matting

Originally matting was a domain of film industry and it was done on a blue background only. In addition to blue, other colors can be used, green is the most common. Red, green and blue channels have all been used, but blue has been favored for several reasons. It is the complementary color to flesh tone. Since the most common color in most scenes is flesh tone, the opposite color is the logical choice to avoid conflicts. Sometimes (usually) the background color reflects onto the foreground talent creating a slight blue tinge around the edges. This is known as blue spill. It doesn't look nearly as bad as green spill, which one would get from green.

Green has it's own advantages, beyond the obvious one of greater flexibility in matting with blue foreground objects. Green paint has greater reflectance than blue paint which can make matting easier. Also, video cameras are usually most sensitive in the green channel, and often have the best resolution and detail in that channel. A disadvantage is that green spill is almost always objectionable and obvious even in small amounts, whereas blue can sometimes slip by unnoticed.

A sophisticated television process is Ultimatte; also the name of the company that manufactures Ultimatte equipment. It has been the ultimate in video compositing for 30 years. With an Ultimatte unit it is possible to create composites that include smoke, transparent objects, different shades of blue, and shadows. It is useful to think of the Ultimatte process as a mixing process, not a keying process. This makes it possible to matte with shadows, hair, water etc. An Ultimatte uses the intensity and purity of the blue signal as a function to determine how much blending to perform between the foreground and background images. Another useful feature of the Ultimatte is blue spill removal. Other circuits deal with glare, uneven or dirty blue backings, etc.

2.3.2 Formal presentation

In [10] or [11] there is a nice formal presentation of the problem. The color $C = [R, G, B, \alpha]$ at each point of a desired composite will be some function of the color C_f of the foreground and color C_b of the new background at the corresponding points in the two elements. Each of the first three primary color coordinates is assumed to have been premultiplied by the alpha coordinate. We shall sometimes refer to just these coordinates with the abbreviation $c = [R, G, B]$, for color C. For any subscript i, $C_i = [R_i, G_i, B_i, \alpha_i]$ and $c_i = [R_i, G_i, B_i]$. Each

of the four coordinates is assumed to lie on [0, 1]. We shall always assume that $\alpha_f = \alpha_b = 1$ for C_f and C_b i.e., the given foreground and new background are opaque rectangular images.

The foreground element C_f can be thought of as a composite of a special background, all points of which have the (almost) constant backing color C_k, and a foreground C_o that is the foreground object in isolation from any background and which is transparent, or partially so, whenever the backing color would show through. Thus $C_f = f(C_o, C_k)$ expresses the point-by-point foreground color as a given composite f of C_k and C_o. We shall always take $\alpha_k = 1$ for C_k.

This facilitates the matting problem. Given C_f and C_b at corresponding points, and C_k a known backing color, and assuming $C_f = C_o + (1 - \alpha_o)C_k$, determine C_o which then gives composite color $C = C_o + (1 - \alpha_o)C_b$ at the corresponding point, for all points that C_f and C_b share in common.

We know that R_f is an interpolation from R_k to R_o with weight α_o. In other words $R_f = R_o + (1 - \alpha_o)R_k$, similar relations hold for G_f and B_f. That means $c_f = c_o + (1 - \alpha_o)c_k$ in our abbreviated notation. A complete solution requires R_o, G_o, B_o, and α_o. Thus we have three equations and four unknowns, an incompletely specified problem and hence an infinity of solutions, unsolvable without more information. Fortunately, there are some special cases where a solution to the matting problem does exist and is simple.

2.3.3 No blue color

If c_o is known to contain no blue, $c_o = [R_o, G_o, 0]$, and c_k contains only blue, $c_k = [0, 0, B_k]$, then

$$c_f = c_o + (1 - \alpha_o)c_k = [R_o, B_o, (1 - \alpha_o)B_k] \tag{2.6}$$

Thus, solving the $B_f = (1 - \alpha_o)B_k$ equation for α_o gives solution

$$C_o = \left[R_f, G_f, 0, 1 - \frac{B_f}{B_k}\right], \tag{2.7}$$

if $B_k \neq 0$. This example is exceedingly ideal. The restriction to foreground objects with no blue is quite serious, excluding all grays but black, about two-thirds of all hues, and all pastels or tints of the remaining hues (because white contains blue). Basically, it is only valid for one plane of the 3D RGB color space, the RG plane.

2.3.4 Gray or flesh color

There is a solution to the matting problem if R_o or $G_o = aB_o + b\alpha_o$, and if c_k is pure blue with $aB_k + b \neq 0$. To show this, we derive the solution C_o for the green case, since the solution for red can be derived similarly: The conditions,

rewritten in color primary coordinates, are $c_f = [R_o,\, aB_o + b\alpha_o,\, B_o + (1-\alpha_o)B_k]$. Eliminate B_o from the expressions for G_f and B_f to solve for α_o:

$$C_o = \left[R_f, G_f, B_\Delta + \alpha_o B_k, \frac{G_f - aB_\Delta}{aB_k + b}\right], \tag{2.8}$$

if $aB_k + b \neq 0$. Here we have introduced a very useful definition $C_\Delta = C_f - C_k$. The special case C_o gray clearly satisfies given conditions, with $a = 1$ and $b = 0$ for both R_o and G_o. Thus it is not surprising that science fiction space movies effectively use the blue screen process (the color-difference technique) since many of the foreground objects are neutrally colored spacecraft. As we know from practice, the technique often works adequately well for desaturated (towards gray) foreground objects, typical of many real-world objects. A particularly important foreground element in film and video is flesh which typically has color $[d,\, 0.5d,\, 0.5d]$. Flesh of all races tends to have the same ratio of primaries, so d is the darkening or lightening factor. This is a non-gray example satisfying given conditions, so it is not surprising that the blue screen process works for flesh.

2.4 Difference matting

Another class of methods extract foreground objects based on a difference between input image and a reference image of background. It is a more general case of constant color matting. This time the background color is not constant, it is rather defined pixel by pixel by the reference picture.

2.4.1 Triangulation matting

An interesting solution is presented in [11]. Suppose c_o is known against two different shades of the backing color. Then a complete solution (so-called triangulation matting) exists as stated formally below. It does not require any special information about c_o.

Let B_{k1} and B_{k2} be two shades of the backing color i.e., $B_{k1} = cB_k$ and $B_{k2} = dB_k$ for $0 \leq d < c \leq 1$. Assume c_o is known against these two shades. Then there is a solution C_o to the matting problem. Note that c_{k2} could be black i.e., $d = 0$. The assumption that c_o is known against two shades of B_k is equivalent to the following:

$$c_{f1} = [R_o, G_o, B_o + (1-\alpha_o)B_{k1}] \tag{2.9}$$

$$c_{f2} = [R_o, G_o, B_o + (1-\alpha_o)B_{k2}]. \tag{2.10}$$

The expressions for B_{f1} and B_{f2} can be combined and B_o eliminated to show

$$\alpha_o = 1 - \frac{B_{f1} - B_{f2}}{B_{k1} - B_{k2}}, \tag{2.11}$$

where the denominator is not 0 since the two backing shades are different. Then

$$R_o = R_{f1} = R_{f2} \tag{2.12}$$

$$G_o = G_{f1} = G_{f2} \tag{2.13}$$

$$B_o = \frac{B_{f2}B_{k1} - B_{f1}B_{k2}}{B_{k1} - B_{k2}} \tag{2.14}$$

completes the solution. In [11] it is further shown that the triangulation matting can be used even in more general case when the foreground is known against two arbitrary backgrounds that differ in every pixel.

2.5 Natural image matting algorithms

Other approaches attempt to pull mattes from natural (arbitrary) backgrounds, using statistics of known regions of foreground or background in order to estimate the foreground and background colors along the boundary. Once these colors are known, the opacity value is uniquely determined. Many of these algorithms are nicely described in [13]. In most cases, the process begins by segmenting the image into three regions: definitely foreground, definitely background, and unknown. This must be done by a human. The algorithms then estimates contributions of object and background to the composite. Hence an alpha channel for all pixels in the unknown region can be determined.

2.5.1 Mishima method

More recently, Mishima developed a matting technique based on representative foreground and background samples, see Figure 2.1(e). In particular, the algorithm starts with two identical polyhedral (triangular mesh) approximations of a sphere in RGB space centered at the average value B of the background samples. The vertices of one of the polyhedra (the background polyhedron) are then repositioned by moving them along lines radiating from the center until the polyhedron is as small as possible while still containing all the background samples. The vertices of the other polyhedron (the foreground polyhedron) are similarly adjusted to give the largest possible polyhedron that contains no foreground pixels from the sample provided. Given a new composite color C, then, Mishima casts a ray from B through C and defines the intersections with the background and foreground polyhedra to be B and F, respectively. The fractional position of C along the line segment BF is α.

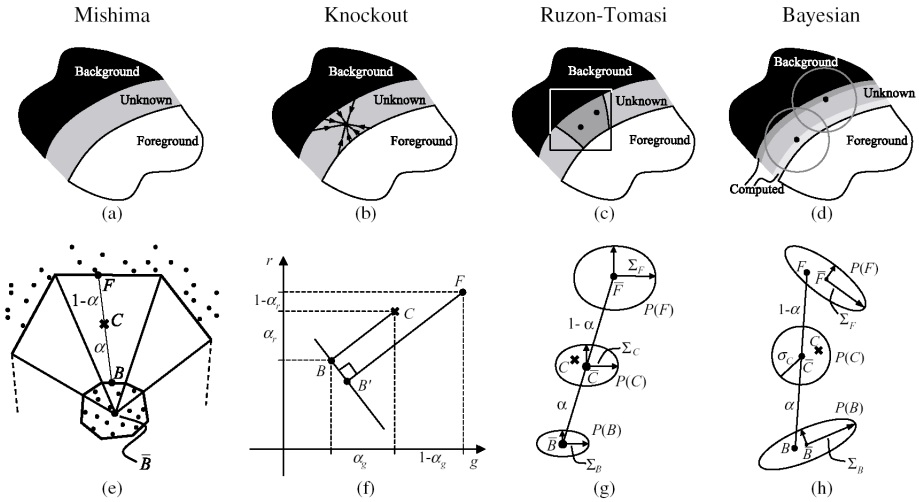

Figure 2.1: Summary of matting algorithms. Each of them requires some specification of background and foreground pixels. Figures (e)–(h) show how matte parameters are computed

2.5.2 Knockout system

For Knockout[1], after user segmentation, the next step is to extrapolate the known foreground and background colors into the unknown region. In particular, given a point in the unknown region, the foreground F is calculated as a weighted sum of the pixels on the perimeter of the known foreground region. The weight for the nearest known pixel is set to 1, and this weight tapers linearly with distance, reaching 0 for pixels that are twice as distant as the nearest pixel. The same procedure is used for initially estimating the background B' based on nearby known background pixels. Figure 2.1(b) shows a set of pixels that contribute to the calculation of F and B' of an unknown pixel.

The estimated background color B' is then refined to give B. using one of several methods that are all similar in character. One such method establishes a plane through the estimated background color with normal parallel to the line $B'F$. The pixel color in the unknown region is then projected along the direction of the normal onto the plane, and this projection becomes the refined guess for B. Figure 2.1(f) illustrates this procedure. Finally, Knockout estimates α according to the relation

$$\alpha = \frac{f(C) - f(B)}{f(F) - f(B)}, \tag{2.15}$$

where f projects a color onto one of several possible axes through RGB space (e.g., onto one of the R, G, or B axes). Figure 2.1(f) illustrates alphas computed with respect to the R and G axes. In general, α is computed by projection

[1]Developed by Ultimatte company.

onto all of the chosen axes, and the final α is taken as a weighted sum over all the projections, where the weights are proportional to the denominator in equation 2.15 for each axis.

2.5.3 Ruzon-Tomasi method

Ruzon and Tomasi [12] take a probabilistic view that is. First, they partition the unknown boundary region into sub-regions. For each sub-region, they construct a box that encompasses the sub-region and includes some of the nearby known foreground and background regions, see Figure 2.1(c). The encompassed foreground and background pixels are then treated as samples from distributions $P(F)$ and $P(B)$, respectively, in color space. The foreground pixels are split into coherent clusters, and unoriented Gaussians (i.e., Gaussians that are axis-aligned in color space) are fit to each cluster, each with mean $\bar{F}$ and diagonal covariance matrix Σ_F. In the end, the foreground distribution is treated as a mixture (sum) of Gaussians. The same procedure is performed on the background pixels yielding Gaussians, each with mean $\bar{B}$ and covariance Σ_B, and then every foreground cluster is paired with every background cluster. Many of these pairings are rejected based on various "intersection" and "angle" criteria. Figure 2.1(g) shows a single pairing for a foreground and background distribution.

After building this network of paired Gaussians, Ruzon and Tomasi treat the observed color C as coming from an intermediate distribution $P(C)$, somewhere between the foreground and background distributions. The intermediate distribution is also defined to be a sum of Gaussians, where each Gaussian is centered at a distinct mean value $\bar{C}$ located fractionally (according to a given alpha) along a line between the mean of each foreground and background cluster pair with fractionally interpolated covariance Σ_C. The optimal alpha is the one that yields an intermediate distribution for which the observed color has maximum probability; i.e., the optimal α is chosen independently of F and B. As a post-process, the F and B are computed as weighted sums of the foreground and background cluster means using the individual pairwise distribution probabilities as weights. The F and B colors are then perturbed to force them to be endpoints of a line segment passing through the observed color and satisfying the compositing equation.

2.5.4 Bayesian approach

The Bayesian approach described in [13] solves the problem in part by building foreground and background probability distributions from a given neighborhood. The method uses a continuously sliding window for neighborhood definitions, marches inward from the foreground and background regions, and utilizes nearby *computed* F, B, and α values (in addition to these values from "known" regions) in constructing oriented Gaussian distributions, as illustrated in Figure 2.1(d). Bayesian approach formulates the problem of computing matte parameters in

a well-defined Bayesian framework and solves it using the maximum a posteriori (MAP) technique.

In MAP estimation, the method tries to find the most likely estimates for F, B, and α, given the observation C. We can express this as a maximization over a probability distribution P and then use Bayess rule to express the result as the maximization over a sum of log likelihoods:

$$\begin{aligned}\arg\max_{F,B,\alpha} P(F,B,\alpha|C) &= \arg\max_{F,B,\alpha} P(C|F,B,\alpha)P(F)P(B)P(\alpha)/P(C) \\ &= \arg\max_{F,B,\alpha} L(C|F,B,\alpha) + L(F) + L(B) + L(\alpha), \quad (2.16)\end{aligned}$$

where L is the log likelihood $L = \log P$. The $P(C)$ term may be dropped because it is a constant with respect to the optimization parameters. Figure 2.1(h) illustrates the distributions over which we solve for the optimal F, B, and α parameters.

The problem is now reduced to defining the log likelihoods $L(C|F,B,\alpha)$, $L(F)$, $L(B)$, and $L(\alpha)$. We can model the first term by measuring the difference between the observed color C and the color that would be predicted by the estimated F, B, and α. The spatial coherence of the image is used to estimate the foreground term $L(F)$. That is, the color probability distribution is built using the known and previously estimated foreground colors within each pixels neighborhood. Given a set of foreground colors, we first partition colors into several clusters. For each cluster, we calculate the weighted mean color $\bar{F}$ and the weighted covariance matrix Σ_F. The log likelihoods for the foreground $L(F)$ can then be modeled as being derived from an oriented elliptical Gaussian distribution. The definition of the log likelihood for the background $L(B)$ depends on which matting problem is being solved. For natural image matting, an analogous term to that of the foreground is used. For constant color matting, the mean and covariance is calculated for the set of all pixels that are labelled as background. For difference matting, we have the background color at each pixel; we therefore use the known background color as the mean and a user-defined variance to model the noise of the background. The log likelihood for the opacity $L(\alpha)$ can be assumed constant (and thus omitted from the maximization).

2.6 Discrete Cosine Transform

The discrete cosine transform (DCT) is a Fourier-related transform converting input data from a spatial space into a frequency space. It's similar to the discrete Fourier transform (DFT), but using only real numbers. It is equivalent to a DFT of roughly twice the length, operating on real data with even symmetry (since the Fourier transform of a real and even function is real and even). In some DCT variants the input and/or output data are shifted by half a sample. There are eight standard variants, of which four are common. The most common variant

of discrete cosine transform is the type-II DCT, which is often called simply "the DCT". Its inverse, the type-III DCT, is correspondingly called "the inverse DCT" or "the IDCT".

The DCT, and in particular the DCT-II, is often used in signal and image processing, especially for lossy data compression, because it has a strong energy compaction property. Most of the signal information tends to be concentrated in a few low-frequency components of the DCT. For example, a DCT is used in JPEG image compression, MJPEG, MPEG, and DV video compression. There, the two-dimensional DCT-II of $N \times N$ blocks is computed and the results are quantized and entropy coded. In this case, N is typically 8 and the DCT-II formula is applied to each row and column of the block. The result is an 8×8 transform coefficient array. The (0, 0) element is the DC (zero-frequency) component corresponding to average pixel intensity in the block. Entries with increasing vertical and horizontal index values represent higher vertical and horizontal spatial frequencies. A related transform, the modified discrete cosine transform, or MDCT, is used in AAC, Vorbis, and MP3 audio compression. DCTs are also widely employed in solving partial differential equations by spectral methods, where the different variants of the DCT correspond to slightly different even/odd boundary conditions at the two ends of the array.

The discrete cosine transform is formally defined as a linear, invertible function $F : \mathbf{R}^N \to \mathbf{R}^N$ (where $\mathbf{R}$ denotes the set of real numbers), or equivalently an N×N square matrix. There are several variants of the DCT with slightly modified definitions. The N real numbers $x_0, \ldots, x_{N-1}$ are transformed into the N real numbers $X_0, \ldots, X_{N-1}$ according to one of the formulas. The DCT-II is probably the most commonly used form

$$X_k = \sqrt{\frac{2}{N}} \cdot c_k \sum_{n=0}^{N-1} x_n \cos \frac{k(2n+1)\pi}{2N}, \tag{2.17}$$

where $c_k = \frac{1}{\sqrt{2}}$ for $k = 0$ and $c_k = 1$ otherwise. In image processing we need a 2D DCT which is defined as

$$X_{u,v} = \sqrt{\frac{4}{MN}} \cdot c_{u,v} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_{i,j} \cos \frac{u(2i+1)\pi}{2N} \cdot \cos \frac{v(2j+1)\pi}{2M}, \tag{2.18}$$

where $c_{u,v} = \frac{1}{\sqrt{2}}$ for $u = v = 0$ and $c_{u,v} = 1$ otherwise. Since DCT is separable, the 2D transform is usually computed as a 1D transform of rows followed by 1D transform of columns. One of the fastest implementations may be found in [7].

Chapter 3

Masking algorithm implementation

In this chapter we will present a solution to our matting problem. Perhaps it should be reminded now. We have two input images – one is a reference image of background, second is an object positioned over the same background. Given these two pictures we shall find a matte (alpha channel) so that the object in the foreground can be extracted. It is important to realize that we don't have any information about the shape of that object. All we have is the difference between background (reference image) and foreground (the image with object) in each pixel. So our task falls into the difference masking category, for details see section 2.4. The method should work rather automatically with minimal user intervention i.e., the user is supposed just to set up some initial parameters.

In chapter 2 a majority of all known matting techniques is described. However none of those is applicable to solve our problem. Thus we will develop a new method that would be suitable right for our task. Since the masking method is intended primary for use in DTP (Desktop Publishing) we don't have to create any absolutely general system. Therefore we can accept some reasonable assumptions. That is in particular that both input images will be taken under very similar conditions in regard of camera position and scene lighting. Additionally the object that is in the center of interest should make reasonable contrast with the background. Of course the method will work even when some of these conditions are broken however then perfect results cannot be expected. In following pages we will consider a pair of input images as shown in Figures 3.1 and 3.2. All computations will be performed with these two images and subsequent results will be shown. Please note that all pictures will be presented as they were computed by appropriate masking method. No further corrections were done so that you can see actual results. But in practice mattes are supposed to be fine tuned. We are developing a Photoshop plug-in so there is no problem in further editing resulting images. You can see all the images in full resolution on the enclosed CD. There are also some more samples.

Figure 3.1: Reference background

Figure 3.2: Object positioned over background

3.1 Image registration

Before we start masking we have to register input images so that corresponding background parts match well. With respect to assumptions made above we can use direct image registration as described in section 2.1.2. Only a movement in x and y axes was chosen as registration parameters. This should be enough by far since zoom can be considered constant. Actually there may occur some slight rotation between images nevertheless we can suppose it to be insignificant. Taking rotations into account will cost more computational time. Moreover image rotations may cause quality loss thus giving poor results.

We should put some approximate constraints to registration parameters. Experiments showed that images taken from a tripod are shifted no more than 2 pixels in each axis. When taking pictures from a hand shift values approach hundreds of pixels. Such images are unacceptable for some reasonable work.

Areas covered by the foreground object are of course different from background thus introducing some registration error. As there is no way how to identify such areas now, we have to rely that all of the other pixels (where there is background in both images) outweigh this error.

3.2 First masking attempts

This section describes several unsuccessful masking algorithms that were proposed while working on this thesis. Although following methods do not perform well they provide valuable experience regarding difference masking and problems concerned with it.

3.2.1 Difference in RGB color space

The first idea is simple. We compute absolute difference between background and foreground. Having three color components we can interpret them as a point in 3D space. The difference can then be defined as Euclidean distance between these points. By thresholding the difference we find out which pixels belong to background and which to the object. Naturally there will be some small deviation even between corresponding pixels that both belong to the background. But pixels belonging to the object should make much bigger difference. However this is not always true. In Figure 3.3 you can see a difference image with it's histogram. Applying a reasonable threshold to the difference we get the result shown in Figure 3.4. There is noticeable noise from the background showing through. The line on the left is caused by imperfect registration. The flower pot is full of holes and at the bottom there are some bad protrusions caused by flower pot reflection on the table. In this case the outcome is not absolutely bad however we would like to achieve better results.

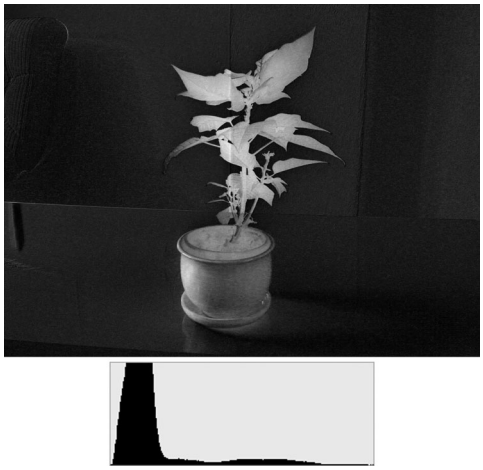

Figure 3.3: Difference in RGB space; values stretched to [0; 255] interval, histogram shown below

Figure 3.4: Result of RGB difference matte

We can try to threshold differences in all channels independently and then combine results together by some AND/OR operation. Experiments showed that the output is none the better. Another idea was to use some adaptive thresholding. First a highly restrictive threshold was used to isolate only those parts of image that definitely belong to the object. Then a low threshold was applied to find parts with low contrast to background. Finally we accept all the parts that are above the higher threshold and we add parts above the lower threshold only if they are connected to areas above the high threshold.[1] This way we can ignore noise while still preserving parts of object that have low contrast. This method requires more computing time but gives almost the same results as the ordinary approach with just one threshold. Another idea was to employ edge detection to determine object boundaries. Unfortunately it was proved to be ineffective because in general, both object and background can contain many strong edges. Detecting them does not bring much useful information and may rather lead to confusion.

3.2.2 Difference in HSV color space

We can try to convert input images into the HSV color space. It stands for Hue, Saturation, Volume (HSL – Hue, Saturation, Lightness is very similar). So instead of color components we have separate information about pixel "color properties". Differences in these properties may be seen in Figures 3.5 and 3.6.
The difference in hue contains strong noise. This is caused by presence of gray shades – majority of background is a white wall. Since grays have nearly zero saturation their hue is poorly defined thus very sensitive to noise. You may also notice bad resolution in Hue and Saturation components. It is because of image compression in digital cameras where only luminance is preserved at full resolution, chrominance is downsampled.

Since the Hue component is practically unusable because of the excessive noise we used just Saturation and Volume to compute the matte. Both components were thresholded and results combined by OR function. You can see the output in Figure 3.7. The flower pot is masked more correctly and the reflection at the bottom was partially eliminated. However several leaves have broken peaks and considerable noise appeared in the left part of image. Masking in HSV space have some advantages but also brings some problems. The end-result is more or less the same as in RGB masking.

[1] Similar technique is used in Canny edge detector.

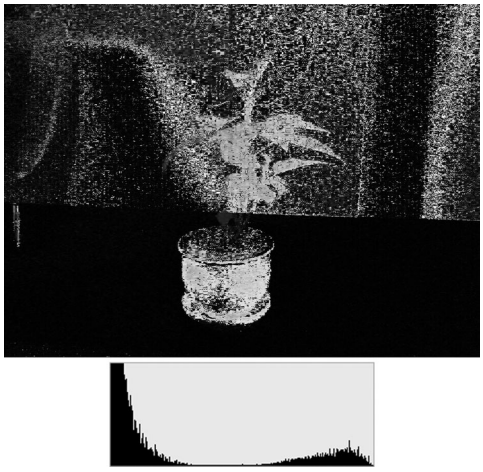

Figure 3.5: Difference in Hue component; values stretched to [0; 255] interval, histogram shown below

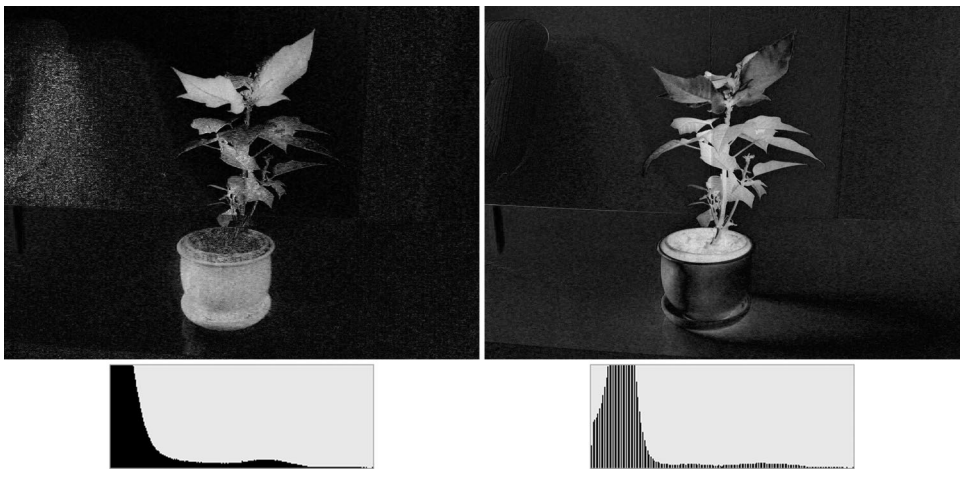

Figure 3.6: Differences in Saturation and Volume components respectively; values stretched to [0; 255] interval, histograms shown below

Figure 3.7: Result of matte based on Saturation and Volume

3.3 The resulting technique

3.3.1 Difference in L*a*b space

The HSV color space didn't bring much progress but it looks promising. So at last we try a L*a*b color space. The L component determines luminance while a and b describe chrominance. Component a is green/red ratio, b is blue/yellow ratio. The L*a*b space have several advantages. First, as already mentioned, it has separate lightness and chroma information. We say that L*a*b is perceptual that means it is based on the same scheme as color perception in human eye. Finally it's a linear color space i.e., the distance between two points corresponds to their color difference.

The masking algorithm is quite simple as those described before. We compute differences in luminance and chrominance, results are shown in Figure 3.8. The final output may be seen in Figure 3.9. As you can see there is very little noise. The line on the left is much thinner, the flower pot is extracted quite well and the bottom reflection was almost eliminated. Unfortunately there is still a hole in the flower's stalk.

Separate lightness and chroma in L*a*b space have great advantages. We can successfully mask shadows. Although they have lover brightness they still still the same hue. On the other hand we can mask objects of the same color

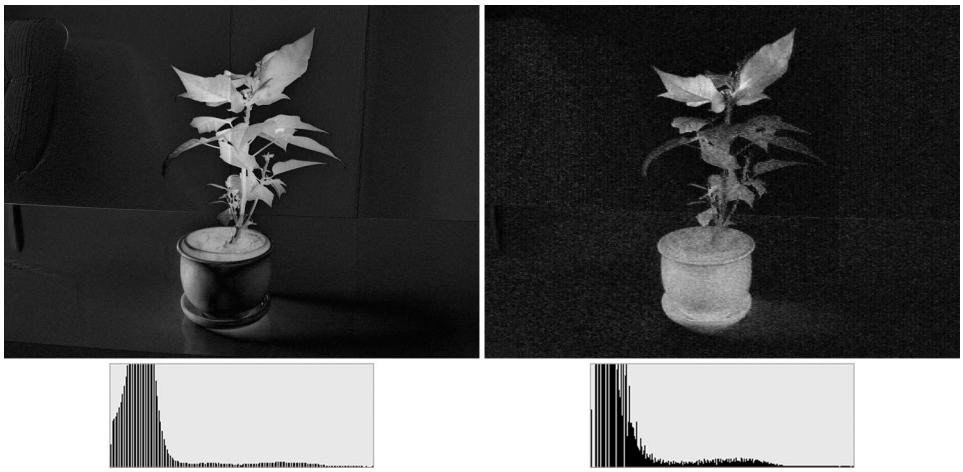

Figure 3.8: Differences in luminance and chrominance respectively; values stretched to [0; 255] interval, histograms shown below

Figure 3.9: Result of matte using L*a*b color space

as background as long as they differ in brightness. Downsampled chrominance doesn't cause problems because a and b are always well defined.

3.3.2 Difference in structure

In addition to separate pixel differencing we can also consider variations in image structure (texture). This way we will be able to discriminate between smooth and grainy areas. That means we can distinguish even pixels with similar lightness and chroma. Image structure is best analyzed by a Discreet Cosine Transform (DCT), please see section 2.6 for details. Also a Discrete Fourier Transform (DFT) could be used but DCT is more suitable since it produces real numbers (unlike DFT which produces imaginary results). We chose 8×8 blocks of images to compute frequency analysis. There are two reasons for it. First, all common DCT implementations use such block size. Second, we need some good compromise between sample size and it's locality. That means we need the block to be large enough to describe image structure. But we also need it small enough so that it relates just to the examined pixel and his small surrounding. A block size of 8×8 seems to be the best choice.

It is enough to compute frequency analysis of the L component. We might add also a and b but it wouldn't bring much improvement because chrominance has poor resolution. After we compute the frequency analysis we discard the DC term. It corresponds to zero frequencies in both axes so it has nothing to do with image structure. It regards average image intensity. Experiments showed that all other frequencies are approximately same significant. It's natural because the image can contain a gradient fill (low frequency) as well as something like a chessboard (high frequencies). So we compute differences between corresponding frequencies and then calculate an average of the block (excluding the DC term). Finally a threshold is applied to get a mask. It can be later combined with masks obtained from luminance and chrominance differencing.

For the flower picture we use here as an example, image structure analysis does not give good results. The method has a strong response along flower's boundary. This causes that a background veil is added to flower's contour. In this case structure differencing should be omitted. On the enclosed CD you can see better examples of using this technique.

3.4 Implementation details

The masking algorithm described in section 3.3 is implemented as a plug-in module for Adobe Photoshop. The program is written in C++ language, Microsoft Visual Studio 2003 was used as development environment. Creation of plug-ins is discussed in large detail in chapter 1 so we won't deal with it here again. Instead we concentrate on how the masking algorithm is implemented.

First we need to convert input image into the L*a*b color space. This is done very easily since Photoshop provides a utility function for it. Please see section 1.6 for details. In the first stage we load the background (reference) image and store it into a buffer for further use. In the next stage we store the image with foreground object into memory too, because we are going to work with it frequently.

Once we have both images ready it's time to proceed with registration. As described in section 3.1 we consider just axial movement. Shift values may be entered by the user or computed automatically by the program. In such case a maximal offset must be specified. The program then tries all possible shifts in given range to find the best one.

As soon as registration is done the program can compute differences in luminance, chrominance and image structure (texture). These values are stored because they depend only on registration parameters. When the user starts adjusting masking parameters the program just thresholds the differences that are already prepared. There is no need to compute them over and over again.

The program uses very fast Discreet Cosine Transform implemented by Pascal Massimino, see [7]. As mentioned before 8×8 blocks are used. This introduces a small issue – we would like to position currently examined point in the center of the block. However it's not possible since 8×8 grid has no exact center. We turn this into our advantage. After computing DCT of a block we assign the result to all four pixels in the middle. Figure 3.10 illustrates the situation. This way we

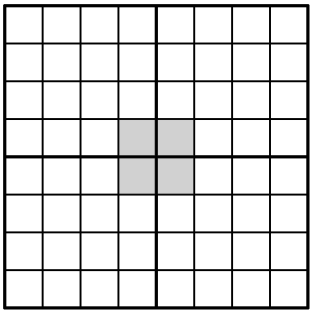

Figure 3.10: A block for DCT computing. The result will be assigned to all four pixels in the middle (shown as gray)

eliminate the issue of no exact center. Moreover we reduce the number of DCT computations to 1/4! Because a single transform gives solution for four points at once.

The program can also do some mask antialiasing so that object boundaries are smoother. This is done very easily by examining the resulting mask. In Photoshop the mask can gain values as any other channel thus from 0 to 255. Zero stands for transparency (image masked out), 255 means full opacity (image fully retained). So the program assigns a half transparent value to all pixels lying

at mask edges (in terms of four-connectivity). The four-connectivity was chosen because it's easy to compute and it does not soften diagonal edges excessively.

3.5 Users manual

To install the plug-in module you just have to copy the DiffMask.8bf file into Photoshop's plug-ins directory ("Plug-Ins" by default) or any of it's subdirectories. The plug-in will then be available through the *Filter* | *Masking* | *Difference Mask...* menu in Photoshop. Whenever the plug-in is launched it first converts the input image internally into the L*a*b color space. This may take several seconds so please be patient. You can break any operation by pressing Escape at any time.

Once the input image is converted, a setup dialog shows up. You can see a screenshot in Figure 3.11. When running the plug-in for the first time there

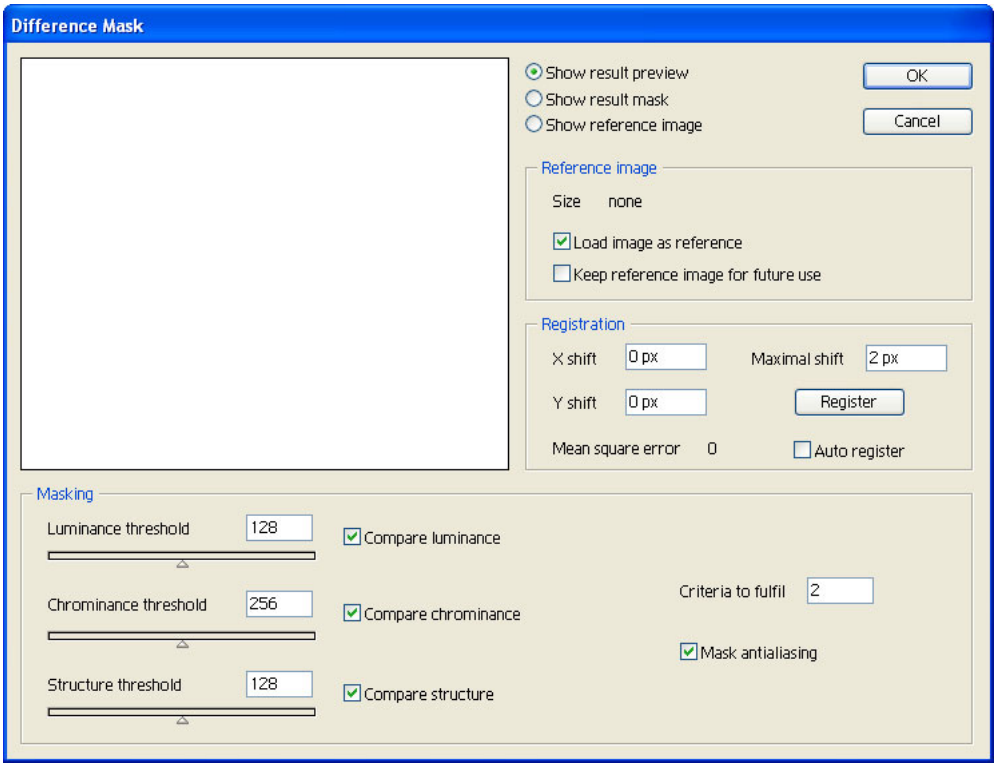

Figure 3.11: Plug-in setup dialog

is no reference image loaded. So you cannot do any masking. Just make sure the *Load image as reference* check box is checked and hit the *OK* button. The plug-in will load current image as reference background. You don't have to care about the *Keep reference image for future use* check box since when loading a new image it will always be kept.

The plug-in writes the mask into the last image channel. So before proceeding with masking you should add a channel to store result there. There are several ways how to do it, it depends on what do you prefer to use. You can just convert the image into a layer, it has a transparency implicitly. This can be done by the *Layer | New | Layer From Background* command. Another possibility is to add an alpha channel to the image. You can do it in the *Channels* palette. *Don't forget to select all the channels before running the plug-in.* The last and perhaps the most preferred way is to add a mask. First you must convert the image into a layer. Then a mask can be added in the *Layers* palette. *Don't forget to select the image layer, not the layer mask, before running the plug-in.*

As you have everything ready you can run the plug-in to proceed with masking. Naturally, the image you want to mask must have the same size as the reference background (it's size is shown in the *Reference image* panel). There is also a preview in the setup dialog. The image inside may be panned using the *right*[2] mouse button. Radio buttons to the right control what will be shown as preview. It may be the real result, just a black&white mask or the reference (background) image.

Before you start masking you should register input image with the reference one. This may be done in the *Registration* panel. The mean square error value shows average difference between images for current settings. You can control shifting manually or hit the *Register* button and let the plug-in find the best registration in given interval. The interval is specified in the *Maximal shift* field. The value means how big shifts will be tried during registration. When you enter 2 for example, the plug-in will try all shifts within [-2; 2] interval in both x and y axes (25 possible shifts in total). Checking the *Auto register* check box ensures that registration will be performed automatically every time the plug-in starts. This is useful for some batch processing.

At last we get to the masking itself. It can be mastered by controls in the *Masking* panel. You can select which criteria to use by appropriate check boxes. Thresholds may be set either by a slider or by entering a value directly into corresponding field. You can also use up/down arrow keys to increment/decrement current value. This holds for all editable fields in the setup dialog. In addition you can set how much criteria must be fulfilled. It's something like putting AND or OR operators between them. Finally there is an option for mask antialiasing. If you set the *Mask antialiasing* check box, the resulting mask will have smoothed edges i.e, boundary pixels will be half transparent. Everything ready, hit the *OK* button and see the result.

[2]The Adobe Dialog Manager experience some problems catching events from the left button.

Conclusion

A detailed description of creation plug-in modules for Adobe Photoshop was presented. It was explained how plug-ins work, how they are called and how they communicate with the host. Plug-in types and possibilities were discussed as well as what functions are provided by plug-in host. Data structures for information interchange between module and host were described. Also a setup of Microsoft Visual Studio project was shown since building a plug-in is not as straightforward as compiling an ordinary program. Adobe Dialog Manager was introduced including description of how to use it for creating and managing dialog windows. To see actual realization, an image masking algorithm was implemented as a plug-in for Adobe Photoshop.

Two different methods for image registration were presented. Alpha channel was introduced as essential descriptor of image transparency. It was explained how images can be blended together to form a composite scene. Then several approaches to image masking were introduced. These should represent majority of all known matting techniques. Constant color matting was described as being the most common in film industry. Difference masking was presented and it was found out that there are few algorithms that can deal with it. Finally a natural image masking was introduced as a method of extracting objects from arbitrary background. However this technique requires some human assistance in detection of approximate object boundaries. Lastly a Discrete Cosine Transform was described as a means to frequency analysis.

A new method was proposed that solves given problem of difference masking. Having an image of background as reference and an image of object positioned over the same background, the method is able to extract the matte. It can also perform constant color matting since it's just a special case with uniform background. It is quite hard to evaluate the method in sense of correctness since there is in fact no reference solution. We can say that correct masking result is what most people would agree to be the real object separated from background.

Developed method gives good results providing that several conditions are fulfilled. Both input images (background and object over background) must be taken under similar conditions. The most important is that the background may not alter. The camera must have constant zoom and should not move excessively. Registration algorithm can handle axial image shifts in range of

several pixels. There should be no rotation between images. The scene must be lit approximately equally when taking both shots. The object we want to mask should make good contrast with background regarding luminance, chrominance and/or texture. There should be no additional reflections introduced by the object we want to mask, nor the object can be transparent. The method can deal with shadows to certain extent. If one or more conditions are violated the method is still capable of producing reasonable results. However the better input is provided the better results may be expected.

The matting method is implemented as a plug-in for Adobe Photoshop. It is therefore available as a part of complex image editing application. If the masking algorithm fails in some areas, it's quite easy to correct the result by one of many Photoshop's tools. Since it is hard to rate masking results, the time necessary to create a nice matte was measured. Implemented method was compared against Vertus Fluid Mask. It is a Photoshop plug-in too, intended for natural image masking. Creating the mask completely by hand was tried as well. Magic Wand and Magnetic Lasso tools were often used to simplify the task. In all cases the matte was finally refined by hand as necessary to create a nice object mask.

Experiments with several images were performed and time was recorded. It is perhaps useless to present some hard numbers here since they strongly depend on complexity of the object being masked and, indeed, on user's skills. The time needed to create a mask varies from several minutes to tens of minutes. Experiments showed that creating the mask only by hand is a frustrating work. It takes from 8 to 30 minutes. The score of Vertus Fluid Mask was a bit surprise. To make a good quality matte, from 10 to 30 minutes were needed. The time is approximately equal to hand masking. Nevertheless, Fluid Mask is a powerful plug-in that can handle even hair, fur and other problematic things. These would be very hard to mask only by hand. Finally, using the algorithm introduced in this thesis, matte can be extracted in time from 4 to 20 minutes. It means that if a reference background image is available, the proposed method can shorten masking time nearly by one half. Thus the new technique proved itself to be useful.

Regarding suggestions for future work, masking algorithm may be extended by some morphological operations to smoothen the resulting mask in sense of suppressing noise and eliminating speckles. Also a more powerful registration algorithm may be used to make it possible to match images taken without a tripod. Current masking algorithm can deal with shadows quite well. The problem of reflections seems to be very hard. At least, as long as just two images (background and background with object) are available. Nevertheless, some additional research may be dedicated to the problem of transparent objects. Under some reasonable assumptions it might be solved somehow. It is a challenging task, but it will be certainly extremely useful to have a solution for it.

Bibliography

[1] *Adobe Photoshop Application Programming Interface Guide*. Available on demand at www.adobe.com

[2] *Adobe Dialog Manager Programmer's Guide and Reference*. Available on demand at www.adobe.com

[3] Kas, T.: *How to Write a Photoshop Plug-In, Part 1, 2*. Available at www.mactech.com/articles/mactech/Vol.15/15.04/PhotoshopPlug-InsPart1/index.html

www.mactech.com/articles/mactech/Vol.15/15.05/PhotoshopPlug-InsPart2/index.html

[4] Lowe, D. G.: *Distinctive Image Features from Scale-Invariant Keypoints*. In International Journal of Computer Vision, 2004

[5] Váša, L.: *Resolution improvement of digitized images*. Diploma thesis at the University of West Bohemia, 2004

[6] *Wikipedia, the free encyclopedia*. Available at www.wikipedia.com

[7] Massimino, P.: *implementing a fast DCT / IDCT with SIMD instructions*. Available at http://skal.planet-d.net/coding/dct.html

[8] Porter, T., Duff, T.: *Compositing Digital Images*. In Proceedings of SIGGRAPH 84, pp. 253-259, 1984

[9] Bradford, S.: *The Blue Screen Page*. Available at www.seanet.com/∼bradford/bluscrn.html

[10] *Theory of Blue Screen Matting*. Downloadable from www.comp.nus.edu.sg/∼cs5245/lecture/matte.pdf

[11] Smith, A. R., Blinn J. F.: *Blue Screen Matting*. In Proceedings of SIGGRAPH 96, pp. 259-268, 1996

[12] Ruzon, M. A., Tomasi, C.: *Alpha Estimation in Natural Images*. In Proceedings of the IEEE Conference on CVPR, Vol. 1, pp. 18-25, 2000

[13] Chuang Y.-Y., Curless, B., Salesin, D. H., Szeliski, R.: *A Bayesian Approach to Digital Matting*. In Proceedings of the IEEE Conference on CVPR, Vol. 2, pp. 264-271, 2001

[14] Hillman, P., Hannah, J., Renshaw, D.: *Alpha Channel Estimation in High Resolution Images and Image Sequences*. In Proceedings of the IEEE Conference on CVPR, Vol. 1, pp. 1063-1068, 2001

Appendix A

More masking examples

In following pictures you can see some other examples of masking results. All of them were generated by methods described in sections 3.2 and 3.3. In first two images you can see the reference background and the object respectively.

Figure A.1: Reference background

Figure A.2: Object over background

Figure A.3: RGB difference matte. Foreground object is full of holes. However if we would lower the difference threshold, shadows from background will come up

Figure A.4: Matte based on Saturation and Volume differences. The result is better then the one from RGB differencing. Though there are still many gaps

Figure A.5: Matte computed from L*a*b difference. Definitely the best one. Although the matte is still not perfect, you can notice that fighter's fists and feet start to appear