Developer's Image Library Manual

By Denton Woods
Abysmal Software
May 2002

Table of Contents

Introduction

Developer's Image Library was previously called OpenIL, but due to trademark issues, OpenIL is now known as DevIL. DevIL is an open source programming library for programmers to incorporate in to their own programs. DevIL loads and saves a large variety of images for use in a software developer's program. This library is capable of manipulating images in various ways and passing image information to display APIs, such as OpenGL and Direct3D.

The purpose of this manual is to guide users in coding with the Developer's Image Library. This manual is for users proficient in C and with competent knowledge of the integrated development environment (IDE) or compiler they are using.

Library Reference

Several times throughout this document, the three different sub-libraries of DevIL are referenced as IL, ILU and ILUT. IL refers to the base library for loading, saving and converting images. ILU refers to the middle level library for image manipulation. ILUT refers to the high level library for displaying images. Functions in IL, ILU and ILUT are prefixed by 'il', 'ilu' and 'ilut', respectively.

Library Setup

Microsoft Visual C++ Setup

DevIL setup for Windows is straightforward. Unzip DevIL in an empty directory. If using WinZip, check the “Use folder names” box before unzipping. Use the -d command line option if using pkunzip. Then double-click on the ImageLib.dsw file in the install directory to load the DevIL workspace in Microsoft Visual C++ (MSVC++).

Directories

You will need to change some directory settings in MSVC++ to get DevIL working.

1. Navigate to the *Tools* menu and select *Options*.
2. Click on the *Directories* tab.
3. Under *Show directories for*, select "Include files".
4. Click the *New* button (to the left of the red 'X')
5. Type the directory DevIL is installed in, plus "\Include". For example, if you installed DevIL to E:\ImageLib, enter "E:\ImageLib\Include".

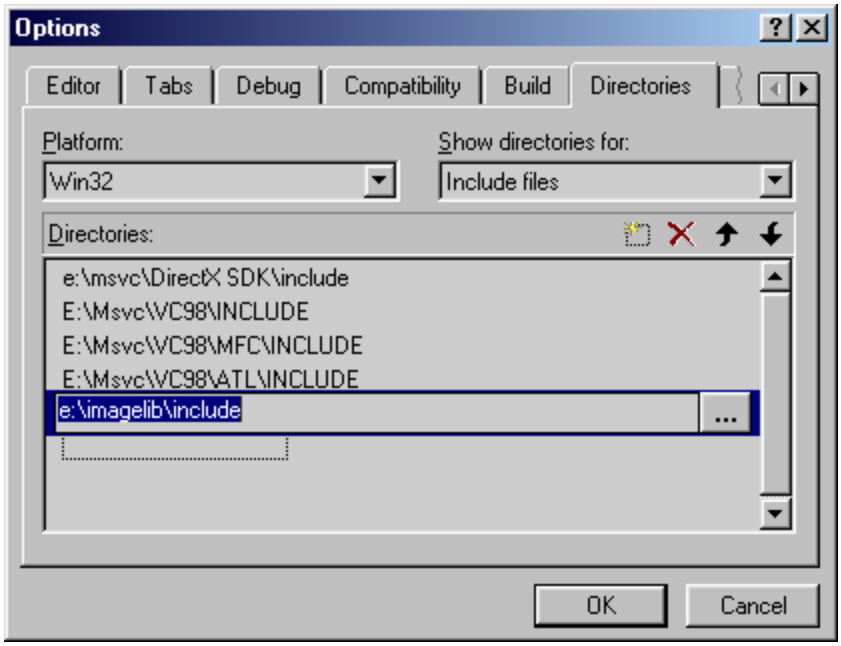


Figure 1-1. *Include Directory Settings Dialog*

6. Under *Show directories for*, click on "Library files".
7. Click the *New* button (to the left of the red 'X').
8. Type the directory DevIL is installed in, plus “\Lib". For example, if you installed DevIL to E:\ImageLib, enter "E:\ImageLib\Lib".
9. Click the *New* button (to the left of the red ‘X’).

10. Type the directory DevIL is installed in, plus "\Lib\Debug". In the previous example, you would enter "E:\ImageLib\Lib\Debug".
11. Choose OK.

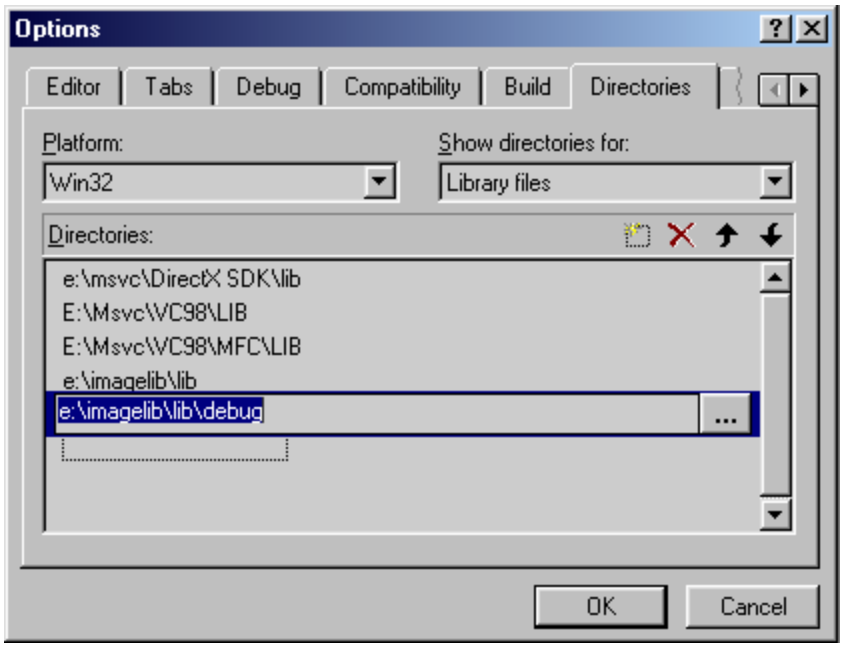


Figure 1-2. *Library Directory Settings Dialog*

MSVC++ Bug Workaround

Microsoft Visual C++ 6.0 has a bug that prevents debugging of a project. The bug appears to occur when you use a #pragma to link a .lib file and link it via another method. The header files il.h, ilu.h and ilut.h automatically link the .lib files in via a #pragma for convenience. To prevent this bug, check for and remove these:

- devil.lib, devil-d.lib, ilu.lib, ilu-d.lib, ilut.lib and ilut-d.lib in your project settings (*Project – Settings* menu).
- devil.lib, devil-d.lib, ilu.lib, ilu-d.lib, ilut.lib and ilut-d.lib in your project's workspace. Some people link libraries into their project this way, which really should be discouraged, due to the hardcoded paths.

Multithreading

DevIL takes advantage of the multithreaded standard LIBC DLLs. To use file streams with DevIL, you must change the project settings of your project. If you do not perform these steps, your program will crash whenever you attempt to use a DevIL file stream.

1. Navigate to the *Project* menu and choose *Settings*.
2. Click the *C/C++* tab.
3. Change the *Category* drop-down menu to read *Code Generation*.

4. Change the *Use run-time library* drop-down menu to *Multithreaded DLL* if the *Settings For* menu says *Win32 Release*. Change the *Use run-time library* drop-down menu to *Debug Multithreaded DLL* if the *Settings For* menu says *Win32 Debug*.
5. Choose OK.

DJGPP Setup

Setting up DevIL in DJGPP requires the following steps:

1. Unzip DevIL in an empty directory. If using WinZip, check the "Use folder names" box before unzipping. Use the -d command line option if using pkunzip.
2. Create a new subdirectory called '*il*' in your DJGPP include directory.
3. Copy the files to their respective places:
   - To use the precompiled libraries, copy libil.a, libilu.a and libilut.a from ImageLib\lib\djgpp to your DJGPP lib directory. Then copy il.h, ilu.h and ilut.h from your ImageLib\lib\il directory to your DJGPP include\il directory.
   - To compile the library yourself, change directories to ImageLib\Makefiles\Djgpp. This folder contains only a makefile for DJGPP. Simply type 'make', and the makefile will compile DevIL and copy the files to their respective locations.

To compile with DevIL in DJGPP, add *–lil* to your command line. To also use ILU and ILUT, use *–lil* and *–lilut*, respectively.

General GCC-based (Linux, Cygwin, Max OS X, etc.) Setup

Setting up DevIL in this environment requires the following steps:

1. Unzip DevIL in an empty directory, by typing "gzip –d gzipname" then "tar –xvf tarname", where 'gzip' and 'tarname' are DevIL-x.x.x.tar.gz and DevIL-x.x.x.tar.
2. Unzip should automatically use the directory structure present in the DevIL zip file.
3. To use the precompiled libraries, copy libIL.so, libILU.so and libILUT.so to a place specified in your library path, or use the full path to the libraries when compiling.
4. To compile the library yourself:
   - Type 'configure'.
   - Type 'make'.
   - Type 'make install' to copy the .so files to /usr/lib and the headers to /usr/include/il.

Basic Usage

Initializing DevIL

You must initialize DevIL, or it will most certainly crash. You need to initialize each library (il, ilu, and ilut) separately. You do not need to initialize libraries you are not using, but keep in mind that the higher level libraries are dependent on the lower ones. For example, ilut is dependent on ilu and il, so you have to initialize il and ilu, also.

IL Initialization

Simply call the **ilInit** function with no parameters:

```
// Initialize IL
ilInit();
```

ILU Initialization

Call the **iluInit** function with no parameters:

```
// Initialize ILU
iluInit();
```

ILUT Initialization

ILUT initialization is slightly more complex than IL and ILU initialization. The function you will use is **ilutRenderer**. You must call **ilutRenderer** before you use any ILUT functions. This function initializes ILUT support for the API you desire to use by a single parameter:

- ILUT_OPENGL – Initializes ILUT's OpenGL support.
- ILUT_ALLEGRO – Initializes ILUT's Allegro support.
- ILUT_WIN32 – Initializes ILUT's Windows GDI and DirectX 8 support.

An example of using **ilutRenderer** follows:

```
// Initialize ILUT with OpenGL support.
ilutRenderer(ILUT_OPENGL);
```

Image Name Handling

Image names are DevIL's way of keeping track of images it is currently containing. Some other image libraries return structs, but they generally seem more cluttered than DevIL's image name handling.

```
ILvoid ilGenImages(ILsizei Num, ILuint *Images);
ILvoid ilBindImage(ILuint Image);
ILvoid ilDeleteImages(ILsizei Num, ILuint *Images);
```

Listing 2-1. *Syntax of the image name functions*

Generating Image Names

Use **ilGenImages** to generate a set of image names. **ilGenImages** accepts an array of ILuint to receive the generated image names. There are no guarantees about the order of the generated image names or any other predictable behaviour like this. If **ilDeleteImages** is called on an image name, **ilGenImages** will return that value afterward, until all deleted image names are used. This conserves memory and is generally quick. The only guarantee is that each member of the *Images* parameter (up to *Num* number of them) will have a new, unique value.

Binding Image Names

ilBindImage binds the current image to the image described by the image name in *Image*. DevIL reserves the number zero for the default base image. If you pass a value for *Image* that was not generated by **ilGenImages**, **ilBindImage** automatically creates an image specified by the image name passed. An image must always be bound before you call any functions that operate on images and their data.

When DevIL creates a new image, the image has the default properties of 1x1x1 with a bit depth of 8. DevIL creates a new image when you call **ilBindImage** with an image name that has not been generated by **ilGenImages** or when you call **ilGenImages** specifically.

Deleting Image Names

ilDeleteImages is the exact opposite of **ilGenImages** and even accepts the exact same parameters. **ilDeleteImages** deletes image names to free memory for subsequent operations. You should always call **ilDeleteImages** on images that are not in use anymore. When you delete an image, DevIL actually deletes all data and anything associate with it, so that **ilGenImages** can possibly use the image name later.

File Handling

Loading Images

DevIL's main purpose is to load images. DevIL's loading is designed to be extremely easy but very powerful. Appendix B lists the image types DevIL is capable of loading.

```
ILboolean ilLoadImage(const char *FileName);
ILboolean ilLoad(ILenum Type, const char *FileName);
ILboolean ilLoadF(ILenum Type, ILHANDLE File);
ILboolean ilLoadL(ILenum Type, ILvoid *Lump, ILuint Size);
```

Listing 2-2. *Syntax of the loading functions*

DevIL contains four loading functions to support different loading styles and loading from several different image sources.

Loading from Files - ilLoadImage

ilLoadImage is the main DevIL loading function. All you do is pass **ilLoadImage** the filename of the image you wish to load. **ilLoadImage** takes care of the rest. **ilLoadImage** allows users to transparently load several different image formats uniformly. DevIL's most powerful function is **ilLoadImage** because of this feature.

Before loading the image, **ilLoadImage** must first determine the image format of the file. **ilLoadImage** performs the following steps:

1. Compares the filename's extension to any registered file handlers, allowing the registered file handlers to take precedence over the default DevIL file handlers. If the extension matches a registered file handler, **ilLoadImage** passes control to the file handler and returns. For more information on registering, refer to the section entitled "Registration".
2. Compares the filename's extension to the extensions natively supported by DevIL. If the extension matches a loading function's extension, **ilLoadImage** passes control to the file handler and returns.
3. Examines the file for a header and tries to match it with a known type of image header. If a valid image header is found, **ilLoadImage** passes control to the appropriate file hander and returns.
4. Returns IL_FALSE.

Loading from Files - ilLoad

DevIL's other file loading function is **ilLoad**. **ilLoad** is similar to **ilLoadImage** in many respects but different in other ways. **ilLoad** accepts two parameters: the type of image and the filename of the image.

ilLoad's type parameter is what differentiates it from **ilLoadImage**. *Type* can be any of the values listed in table B-2 in appendix B or the value IL_TYPE_UNKNOWN. If *Type* is a value from table B-1, **ilLoad** attempts to load the file as the specified type of image format. Only use this if you know what type of images you will be loading and want to bypass DevIL's checks.

If IL_TYPE_UNKNOWN is specified for *Type*, **ilLoad** behaves exactly like **ilLoadImage**. Refer to the previous section for detailed behaviour of these two functions.

Loading from File Streams - ilLoadF

DevIL's file stream loading function is **ilLoadF**. **ilLoadF** is exactly equivalent to **ilLoad**, but instead of accepting a const char pointer, **ilLoadF** accepts an ILHANDLE. DevIL defines ILHANDLE as a void pointer via a typedef. Under normal circumstances, *File* will be a FILE struct pointer defined in stdio.h.

Refer to the section entitled "Registration" for instructions on how to use your own file handling functions and file handles.

Loading from Memory Lumps - ilLoadL

DevIL's file handling is abstracted to allow loading images from memory called "lumps". **ilLoadL** handles loading from lumps. You must specify a valid type as the first parameter and the lump as the second parameter.

The third parameter that **ilLoadL** accepts is the total size of the lump. DevIL uses this value to perform bounds checking on the input data. Specify a value of zero for *Size* if you do not want **ilLoadL** to perform any bounds checking.

Saving Images

DevIL also has some powerful saving functions to fully complement the loading functions.

```
ILboolean ilSaveImage(const char *FileName);
ILboolean ilSave(ILenum Type, const char *FileName);
ILboolean ilSaveF(ILenum Type, ILHANDLE File);
ILboolean ilSaveL(ILenum Type, ILvoid *Lump, ILuint Size);
```

Listing 2-3. *Syntax of the saving functions*

DevIL's saving functions are identical to the loading functions, despite the fact that they save images instead of load images. Lists of possible values for *Type* and supported saving formats are located in Appendix B.

Image Management

Defining Images

ilTexImage is used to give the current bound image new attributes that you specify. Any image data or attributes previously in the current bound image are lost after a call to **ilTexImage**, so make sure that you call it only after preserving the image data if need be.

```
ILboolean ilTexImage( ILuint Width, ILuint Height, ILuint Depth, ILubyte Bpp,
                      ILenum Format, ILenum Type, ILvoid *Data);
```

Listing 2-4. *Syntax of the ilTexImage function*

ilTexImage has one of the longer parameter lists of the DevIL functions, so we will briefly go over what is expected for each argument.

- *Width:* The width of the image. If this is zero, DevIL creates an image with a width of one.
- *Height:* The height of the image. If this is zero, DevIL creates an image with a height of one.
- *Depth:* The depth of the image, if it is an image volume. Most applications should specify 0 or 1 for this parameter.
- *Bpp:* The bytes per pixel of the image data. Do not confuse this with bits per pixel, which is also commonly used. Common bytes per pixel values are 1, 3 and 4.
- *Format:* The format of the image data. Formats accepted are listed here and are self-explanatory:

```
IL_COLOUR_INDEX
IL_RGB
IL_RGBA
IL_BGR
IL_BGRA
IL_LUMINANCE
```

- *Type:* The type of image data. Usually, this will be IL_UNSIGNED_BYTE, unless you want to utilize multiple bits per colour channel. Type accepted are listed here:

```
IL_BYTE
IL_UNSIGNED_BYTE
IL_SHORT
IL_UNSIGNED_SHORT
IL_INT
IL_UNSIGNED_INT
IL_FLOAT
IL_DOUBLE
```

- *Data:* Mainly for convenience, if you already have image data loaded and ready to put into the newly created image. Specifying NULL for this

parameter just results in the image having unpredictable image data. You can specify image data later using **ilSetData** or **ilSetPixels**.

Getting Image Data

There are two ways to set image data: one is quick and dirty, while the other is more flexible but slower. These two functions are **ilGetData** and **ilCopyPixels**.

```
ILubyte*    ilGetData(ILvoid);
ILuint      ilCopyPixels( ILuint XOff, ILuint YOff, ILuint ZOff, ILuint Width,
                          ILuint Height, ILuint Depth, ILenum Format, ILenum
                          Type, ILvoid *Data);
```

Listing 2-6. *Syntax of the functions to get image data*

The Quick Method

Use **ilGetData** to get a direct pointer to the current bound image's data pointer. Do not ever try to delete this pointer that is returned. To get information about the image data, use **ilGetInteger**.

ilGetData will return NULL and set an error of IL_ILLEGAL_OPERATION if there is no currently bound image.

The Flexible Method

Use **ilCopyPixels** to get a portion of the current bound image's data or to get the current image's data with in a different format / type. DevIL takes care of all conversions automatically for you to give you the image data in the format or type that you need. The data block can range from a single line to a rectangle, all the way to a cube.

ilCopyPixels has a long parameter list, like **ilTexImage**, so here is a description of the parameters of **ilCopyPixels**:

- *XOff:* Specifies where to start copying in the x direction.
- *YOff:* Specifies where to start copying in the y direction.
- *ZOff:* Specifies where to start copying in the z direction. This will be 0 in most cases, unless you are using image volumes.
- *Width:* Number of pixels to copy in the x direction.
- *Height:* Number of pixels to copy in the y direction.
- *Depth:* Number of pixels to copy in the z direction. This will be 1, unless you are using image volumes.
- *Format:* The format of the returned data that you desire. Acceptable formats are IL_RGB, IL_RGBA, IL_BGR, IL_BGRA and IL_LUMINANCE.

- *Type:* The type of the data block in *Data.* Acceptable types are IL_UNSIGNED_BYTE, IL_BYTE, IL_UNSIGNED_SHORT, IL_SHORT, IL_UNSIGNED_INT and IL_INT. IL_FLOAT and IL_DOUBLE will be supported shortly. For most purposes, IL_UNSIGNED_BYTE is always acceptable here.
- *Data:* A pointer to the data block that you wish to receive the specified image data. If this is NULL, DevIL will set an error of IL_INVALID_PARAM and return IL_FALSE (please refer to the section on error handling in DevIL).

Setting Image Data

There are two ways to set image data: one is quick and dirty, while the other is more flexible but slower. These two functions are **ilSetData** and **ilSetPixels**.

```
ILboolean  ilSetData(ILvoid *Data);
ILvoid     ilSetPixels( ILuint XOff, ILuint YOff, ILuint ZOff, ILuint Width,
                        ILuint Height, ILuint Depth, ILenum Format, ILenum
                        Type, ILvoid *Data);
```

Listing 2-5. *Syntax of the functions to set image data*

The Quick Method

Use **ilSetData** to set the image data directly. DevIL will copy the data provided in the *Data* parameter to the image's data, so you need not worry about DevIL trying to delete your pointer later on. This function is the counterpart to **ilGetData**.

You must provide image data in the exact same format, type, width, height, depth and bpp as the current bound image, since DevIL does no conversions here; it just does a simple memory copy.

ilSetData will return IL_FALSE and set an error of IL_INVALID_PARAM if *Data* is NULL.

The Flexible Method

Use **ilSetPixels** to set a portion of the current bound image's data or to set the current image's data with data of a different format / type. Specify the data block, where you want to put it and what kind of data it is, and DevIL takes care of all conversions automatically for you. The data block can range from a single line to a rectangle, all the way to a cube.

ilSetPixels has a long parameter list, like **ilCopyPixels**, so here is a description of the parameters of **ilSetPixels**:

- *XOff:* Specifies where to place the block of image data in the x direction.
- *YOff:* Specifies where to place the block of image data in the y direction.
- *ZOff:* Specifies where to place the block of image data in the z direction. This will be 0 in most cases, unless you are using image volumes.
- *Width:* The width of the data block in *Data.*
- *Height:* The height of the data block in *Data.*
- *Depth:* The depth of the data block in *Data*. This will be 1, unless you are using image volumes.
- *Format:* The format of the data block in *Data.* Acceptable formats are IL_RGB, IL_RGBA, IL_BGR, IL_BGRA and IL_LUMINANCE.
- *Type:* The type of the data block in *Data.* For acceptable types, refer to the documentation on **ilTexImage**. For most purposes, IL_UNSIGNED_BYTE is always acceptable here.
- *Data:* A pointer to the actual data block. If this is NULL, DevIL will set an error of IL_INVALID_PARAM and return IL_FALSE (please refer to the section on error handling in DevIL).

If you specify a combination of an offset with a width/height/depth that makes your data block overreach the edge of the currently bound image, DevIL will clip your data so that no crashes will occur and that the resulting image will be correctly produced.

Copying Images

DevIL has three functions to copy images: **ilCopyImage**, **ilOverlayImage** and **ilBlit**.

```
ILboolean ilCopyImage(ILuint Src);
ILboolean ilOverlayImage(ILuint Src, ILint XCoord, ILint YCoord, ILint
                         ZCoord);
ILboolean ilBlit(ILuint Src, ILint DestX, ILint DestY, ILint DestZ, ILuint SrcX,
                 ILuint SrcY, ILuint SrcZ, ILuint Width, ILuint Height, ILuint
                 Depth);
```

Listing 2-6. *Syntax of the functions to copy images*

Direct Copying

Use **ilCopyImage** to create a copy of an image. **ilCopyImage** will copy the image specified by the image name in *Src* to the currently bound image. **ilCopyImage** can be useful when you want to apply an effect to an image but want to preserve the original. The image bound before calling **ilCopyImage** will still be bound after **ilCopyImage** exits.

If you specify an image name in *Src* that has not been generated by **ilGenImages** or **ilBindImage**, **ilCopyImage** will set the IL_INVALID_PARAM error and return IL_FALSE.

Blitting

ilBlit copies a portion of an image over to another image. This is similar to blitting performed in graphics libraries, such as StretchBlt in the Windows API. You can copy a rectangular block from anywhere in a source image, specified by *Src*, to any point in the currently bound image. A description of the various **ilBlit** parameters follows:

- *Src:* The source image name.
- *DestX:* Specifies where to place the block of image data in the x direction.
- *DestY:* Specifies where to place the block of image data in the y direction.
- *DestZ:* Specifies where to place the block of image data in the z direction.
- *SrcX:* Specifies where to start copying in the x direction of the source image.
- *SrcY:* Specifies where to start copying in the y direction of the source image.
- *SrcZ:* Specifies where to start copying in the z direction of the source image.
- *Width:* How many pixels to copy in the x direction of the source image.
- *Height:* How many pixels to copy in the y direction of the source image.
- *Depth:* How many pixels to copy in the z direction of the source image.

Overlaying

ilOverlay is essentially the same as **ilBlit**, but it copies the entire image over, instead of just a portion of the image. **ilOverlay** is more of a convenience function, since you can obtain the same results by calling **ilBlit** with *SrcX*, *SrcY* and *SrcZ* set to zero, with the *Width*, *Height* and *Depth* parameters set to the source image's height, width and depth, respectively. **ilOverlay** is missing six parameters that **ilBlit** has:

- *Src:* The source image name.
- *DestX:* Specifies where to place the block of image data in the x direction.
- *DestY:* Specifies where to place the block of image data in the y direction.
- *DestZ:* Specifies where to place the block of image data in the z direction.

Error Handling

DevIL contains error-handling routines to alert the users of this library to any internal problems in DevIL. The **ilGetError** function reports all errors in DevIL. **iluErrorString** converts error numbers returned from **ilGetError** to a human-readable format.

```
ILenum          ilGetError(ILvoid);
const char*     iluErrorString(ILenum Error);
```

Listing 3-1. *Syntax of the error functions*

Error Detection

Problems can always occur in any software application, and DevIL is no different. DevIL keeps track of all non-fatal errors that have occurred during its operation. All errors are kept on a stack maintained by **ilGetError**. Every time **ilGetError** is called, the last error is returned and pushed off the top of the stack. You should call **ilGetError** until IL_NO_ERROR is returned. IL_NO_ERROR signifies that there are no more errors on the error stack. Most errors reported are not harmful, and DevIL operation can continue, except for IL_OUT_OF_MEMORY.

All error codes that can be returned by **ilGetError** are listed in Appendix A.

Error Strings

iluErrorString returns a human readable error string from any error that **ilGetError** can return. This is useful for when you want to display what kind of error happened to the user.

Image Characteristics

All images have a certain set of characteristics: origin of the image, format of the image, type of the image, and more.

Origin

Image Manipulation

ILU (image library utilities) contains functions to manipulate any type of image in a variety of ways. Some functions filter images, while others perform a wider variety of operations, such as scaling an image. This section will give a comparison of the utility functions against figure 4-1.

Figure 4-1. *Original, unmodified image*

Alienifying

iluAlienify is a filter I created purely by accident, when I was attempting to write colour matrix code. The effect **iluAlienify** gives to an image is a green and purple tint. On images with humans in them, **iluAlienify** generally makes the people look green, hence the fabricated term "alienify". **iluAlienify** does not accept any parameters. Figure 4-2 illustrates this effect on the OpenIL logo.

Figure 4-2. *"Alienified" image*

Blurring

ILU has two blurring functions – **iluBlurAverage** and **iluBlurGaussian**. Blurring can be used for a simple motion blur effect or something as sophisticated as concealing the identity of a person in an image. Both of these functions use a convolution filter and multiple iterations to blur an image. Gaussian blurs look more natural than averaging blurs, because the center pixel in the convolution filter "weighs" more. For an in-depth description of convolution filters, see the excellent "Elementary Digital Filtering" article at http://www.gamedev.net/reference/programming/features/edf/.

iluBlurAverage and **iluBlurGaussian** are functionally equivalent. Both functions accept a single parameter. Call the desired function with the number of iterations of blurring

you wish to be performed on the image. Increase the number of iterations to increase the blurriness of an image.

Figure 4-3. *Average blurred with 10 iterations applied*

Figure 4-4. *Gaussian blurred with 10 iterations applied*

Contrast

The American Heritage Dictionary describes contrast as "The use of opposing elements, such as colors, forms, or lines, in proximity to produce an intensified effect in a work of art." ILU can apply more colour contrast to your image by brightening the lights and darkening the darks via **iluContrast**. This effect can make a dull image livelier and "stand out" more.

iluContrast accepts a single parameter describing the desired amount of contrast to modify the image by. A value of 1.0 does not affect the image. Values above 1.0 to 1.7 increase the amount of contrast in the image, with 1.7 increasing the contrast the most. Values from 0.0 to 1.0 decrease the amount of contrast in the image. Values outside of the 0.0 to 1.7 range will give undefined results. -0.5 to 0.0 will actually create a negative of the image and increase the contrast.

Figure 4-5. *Contrast of 1.6*

Figure 4-6. *Contrast of 0.2*

Figure 4-7. *Contrast of -.5*

Equalization

Sometimes it may be useful to equalize an image – that is, bring the extreme colour values to a median point. **iluEqualize** darkens the bright colours and lightens the dark colours, reducing the contrast in an image or "equalizing" it. Figure 4-8 shows the results of applying **iluEqualize** to the OpenIL image.

Figure 4-8. *Equalized image*

Gamma Correction

iluGammaCorrect applies gamma correction to an image using an exponential curve. The single parameter **iluGammaCorrect** accepts is the gamma correction factor you wish to use. A gamma correction factor of 1.0 leaves the image unmodified. Values in the range 0.0 - 1.0 darken the image. 0.0 leaves a totally black image. Anything above 1.0 brightens the image, but values too large may saturate the image.

Figure 4-9. *Result of gamma correction of 0.5*

Figure 4-10. *Result of gamma correction of 1.9*

Negativity

iluNegative is a very basic function that inverts every pixel's colour in an image. For example, pure white becomes pure black, and vice-versa. The resulting colour of a pixel can be determined by this formula: new_colour = ~old_colour (where the tilde is the negation of the set of bits). **iluNegative** does not accept any parameters and is reversible by calling it again.

Figure 4-11. *iluNegative example*

Noise

DevIL can add "random" noise to any image to make it appear noisy. The function, **iluNoisify**, simply uses the standard libc **rand** function after initializing it with a seed to **srand**. If your program depends on a different seed to **rand**, reset it after calling

iluNoisify. The seed DevIL uses is the standard **time(NULL)** call. Of course, the noise added to the image is not totally random, since no such thing exists, but there should be no repeating, except in extremely large images.

iluNoisify accepts a single parameter – the tolerance to use. This parameter is a clamped (float) value that should be in the range 0.0f - 1.0f. Lower values indicate a lower tolerance, while higher values indicate the opposite. The tolerance indicates just how much of a mono intensity that **iluNoisify** is allowed to apply to each pixel. A "random" mono intensity is applied to each pixel so that you will not end up with totally new colours, just the same colours with a different luminance value. Colours change by both negative and positive values, so some pixels may be darker, some may be lighter, and others will remain the same.

Figure 4-12. *Result of iluNoisify with a 0.50 tolerance*

Pixelization

iluPixelize creates pixelized images by averaging the colour values of blocks of pixels. The single parameter passed to **iluPixelize** determines the size of these square blocks. The result is a pixelized image.

Call **iluPixelize** with values greater than 1 to pixelize the image. The larger the values, the larger the pixel blocks will be. A value of 1 will leave the image unchanged. Values less than 1 generate an error.

Figure 4-13. *Pixelization of 10 pixels across*

Sharpening

Sharpening sharply defines the outlines in an image. **iluSharpen** performs this sharpening effect on an image. **iluSharpen** accepts two parameters: the sharpening factor and the number of iterations to perform the sharpening effect.

The sharpening factor must be in the range of 0.0 - 2.5. A value of 1.0 for the sharpening factor will have no effect on the image. Values in the range 1.0 - 2.5 will sharpen the image, with 2.5 having the most pronounced sharpening effect. Values from 0.0 to 1.0 do a type of reverse sharpening, blurring the image. Values outside of the 0.0 - 2.5 range produce undefined results.

The number of iterations to perform will usually be 1, but to achieve more sharpening, increase the number of iterations. This parameter is similar to the *Iterations* parameter of the two blurring functions. The time it takes to run this function is directly proportional to the number of iterations desired.

Figure 4-14. *Sharpening of 1.5 with 5 iterations*

Resizing Images

Basic Scaling

To resize images, use the **iluScale** function:

```
ILboolean iluScale(ILuint Width, ILuint Height, ILuint Depth);
```

Listing 5.1. *Syntax of the iluScale function*

The three parameters are relatively explanatory. Any image can be resized to a new width, height and depth, provided that you have enough memory to hold the new image. The new dimensions do not have to be the same as the original in any way. Aspect ratios of the image do not even have to be the same. The currently bound image is replaced entirely by the new scaled image.

If you specify a dimension greater than the original dimension, the image enlarges in that direction. Alternately, if you specify a dimension smaller than the original dimension, the image shrinks in that direction.

Figure 5.1. Original image

Figure 5.2. Enlarged image

Figure 5.3. Shrunk image

Advanced Scaling

DevIL also allows you to specify which method you want to use to resize images. As you can see in figure 5.2, the enlarged image is very pixelized. In figure 5.3, the shrunk image is also blocky. This is because a nearest filter was applied to the image in figure 5.1 to produce figures 5.2 and 5.3.

DevIL allows you to use different filters to produce better scaling results:

- Nearest filter - `ILU_NEAREST`
- Linear filter - `ILU_LINEAR`
- Bilinear filter - `ILU_BILINEAR`
- Box filter - `ILU_SCALE_BOX`
- Triangle filter - `ILU_SCALE_TRIANGLE`
- Bell filter - `ILU_SCALE_BELL`
- B Spline filter - `ILU_SCALE_BSPLINE`
- Lanczos filter - `ILU_SCALE_LANCZOS3`
- Mitchell filter - `ILU_SCALE_MITCHELL`

Just use the `ILU_FILTER` define as *PName* in **iluImageParameter** with the appropriate filter define as *Param*.

```
ILvoid iluImageParameter(ILenum PName, ILenum Param);
```

Listing 5.1. *Syntax of the iluImageParameter function*

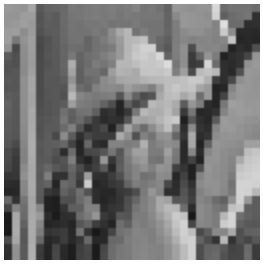
Filter Comparisons

The first three filters (nearest, linear and bilinear) require an increasing amount of time to resize an image, with nearest being the quickest and bilinear being the slowest of the three. All the filters after bilinear are considered the "advanced" scaling functions and require much more time to complete, but they generally produce much nicer results.

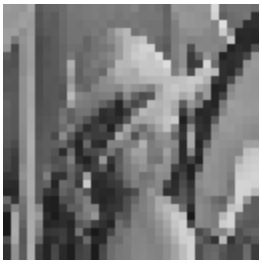
When minimizing an image, bilinear filtering should be sufficient, since it uses a four-pixel averaging scheme to create every destination pixel. Minimized images do not generally have to use higher sampling schemes to achieve a reasonable image.

Enlarging an image, though, depends quite heavily on how good the sampling scheme is. DevIL provides several filtering functions to let you choose which one best fits your needs: speed versus image quality. Below is a comparison of the different types of filters when enlarging an image.

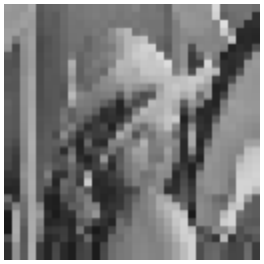
Figure 5.4. Original 'Lena' image



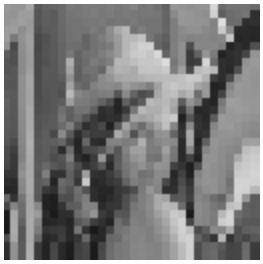
Nearest filter



Linear filter



Bilinear filter



Box filter

Triangle filter

Bell filter

B spline filter | Lanczos filter | Mitchell filter

Figure 5.5. Filter comparisons

Sub-Images

Mipmaps

Mipmaps in DevIL are successive half-dimensioned power-of-2 images. The dimensions do not have to be powers of 2 if you generate them manually, but DevIL's mipmap generation facilities assume power-of-2 images.

Figure 4-1. *All mipmap levels down to 1x1*

Mipmap Creation

You generate mipmaps for any image using **iluBuildMipmaps**. If the image already has mipmaps, the previous mipmaps are erased, and new mipmaps are generated. Otherwise, **iluBuildMipmaps** generates mipmaps for the image.

The mipmaps built are always powers of 2. If the original image does not have power-of-2 dimensions, **iluBuildMipmaps** resizes the original image via **iluScale** to have power-of-2 dimensions.

Mipmap Access

Access mipmaps through the **iluActiveMipmap** function:

```
ILboolean ilActiveMipmap(ILuint MipNum);
```

Listing 4-1. Syntax of the mipmap access function

iluActiveMipmap sets the current image to the *MipNum* mipmap level of the current image. If there are no mipmaps present, then **iluActiveMipmap** returns IL_FALSE, else it returns IL_TRUE. The base image is mipmap level 0, so specify 0 for *MipNum* to return to the base image. The only other method for setting the current image to the base image is to call **ilBindImage** again.

Animations

Animations are similar to mipmaps, but instead of being smaller successive images, the images are the same size but have different data. The successive animation chains in DevIL can be used to create animations in your programs. File formats that natively support animations are .gif and .mng. You can also create your own sub-images as animations.

Animation Chain Creation

Animation Chain Access

Access animations through the **iluActiveImage** function:

```
ILboolean ilActiveImage(ILuint ImageNum);
```

Listing 4-2. Syntax of the mipmap access function

iluActiveImage sets the current image to the *ImageNum* mipmap level of the current image. If there are no mipmaps present, then **iluActiveImage** returns IL_FALSE, else it returns IL_TRUE. The base image is mipmap level 0, so specify 0 for *ImageNum* to return to the base image. The only other method for setting the current image to the base image is to call **ilBindImage** again.

iluActiveImage is functionally equivalent to **iluActiveMipmap**, except that it deals with animations and not mipmaps.

Layers

DevIL does not have a full layer implementation yet.

Sub-Image Mixing

An image can have both mipmaps and animations at the same time. Every image in an animation chain can have its own set of mipmaps, though it is not necessary by any means. If you "activate" an animation image in the base image's animation chain, the active image becomes the new "base" image. Therefore, if you call **iluActiveMipmap**, the new active image will become the new active image,

DXTC/S3TC Notes

DDS Loading/Saving

DevIL supports loading and saving of Microsoft .dds files. DDS files can either be compressed or uncompressed. If they are compressed, DDS files use DirectX Texture Compression (DXTC). DXTC is also known as S3TC, since Microsoft licensed the compression technology from S3.

Keeping DXTC Data

When loading, DevIL uncompresses the DXTC. If you call ilEnable with the IL_KEEP_DXTC_DATA parameter, DevIL will keep an uncompressed copy of the DXTC data along with the image. Functions that deal with DXTC data can use this data without having to recompress the uncompressed data, making these functions operate faster. The only drawback is the use of more memory.

Controlling Saving

DevIL's DXTC support consists of three different compression formats: DXT1, DXT3 and DXT5. DXT2 and DXT4 use premultiplied alpha, which not even OpenGL supports. DevIL loads DXT2 and DXT4 textures but immediately converts them to formats that do not use premultiplied alpha. To set what format to save DDS files in, use this line:

```
ilSetInteger(IL_DXTC_FORMAT, Format);
```

Format can be IL_DXT1, IL_DXT3 or IL_DXT5.

Retrieving DXTC Data

To retrieve a copy of the DXTC data, use **ilGetDXTCData**. To determine how large *Buffer* should be, first call **ilGetDXTCData** with the *Buffer* parameter as NULL. This function will then return the number of bytes that are required to completely store the DXTC data. Call it a second time to actually retrieve the data.

```
ILuint ilGetDXTCData(ILvoid *Buffer, ILuint BufferSize, ILenum
                     DXTCFormat)
```

Listing 5-1. *Syntax of the ilGetDXTCData function*

If the DXTC data does not exist in the format that you request, DevIL will automatically compress the data. If **ilGetDXTCData** returns 0, then the data could not be compressed. To see if a certain format of DXTC data already exists for the currently bound image, call **ilGetInteger** with the IL_DXTC_DATA_FORMAT parameter.

OpenGL/Direct3D DXTC Support

ILUT allows you to directly send the DXTC data to OpenGL or Direct3D. Several modes in ILUT directly control this behavior.

OpenGL S3TC Support

OpenGL can use S3TC (DXTC) textures via extensions. If a computer does not support the S3TC texture extension, DevIL will just send the data normally through **glTexImage2D**, as always. Please keep in mind that DDS files store their data in a top-down format, so if you enable the OpenGL S3TC support, make certain to set the origins of all images in the upper left:

```
ilEnable(IL_ORIGIN_SET);
ilSetInteger(IL_ORIGIN_MODE, IL_ORIGIN_UPPER_LEFT);
```

To enable the OpenGL S3TC support, use the **ilutEnable** function with the ILUT_GL_USE_S3TC parameter:

```
ilutEnable(ILUT_GL_USE_S3TC);
```

Setting this parameter means that ILUT will only use DXTC data from images that are already compressed with DXTC (e.g. DDS files). To force ILUT to compress any image it sends to OpenGL, use **ilutEnable** again:

```
ilutEnable(ILUT_GL_GEN_S3TC);
```

This can adversely affect your performance while loading textures, though, so use it with caution, especially if you are running a performance-critical application.

Direct 3D DXTC Support

ILUT's Direct 3D (D3D) support works exactly like the OpenGL support, except you use the ILUT_D3D_USE_DXTC and ILUT_D3D_GEN_DXTC defines instead of ILUT_GL_USE_S3TC and ILUT_GL_GEN_S3TC, respectively.

Appendix A: Common DevIL Error Codes

Errors sometimes occur within DevIL. To get the error code of the last error that occurred, call **ilGetError** with no parameters. To get a human-readable string of an error code, call **iluErrorString** with the error code. A table of error codes follows:

Error Code #define	Hexadecimal Value	Decimal Value
IL_NO_ERROR	0x000	0
IL_INVALID_ENUM	0x501	1281
IL_OUT_OF_MEMORY	0x502	1282
IL_FORMAT_NOT_SUPPORTED	0x503	1283
IL_INTERNAL_ERROR	0x504	1284
IL_INVALID_VALUE	0x505	1285
IL_ILLEGAL_OPERATION	0x506	1286
IL_ILLEGAL_FILE_VALUE	0x507	1287
IL_INVALID_FILE_HEADER	0x508	1288
IL_INVALID_PARAM	0x509	1289
IL_COULD_NOT_OPEN_FILE	0x50A	1290
IL_INVALID_EXTENSION	0x50B	1291
IL_FILE_ALREADY_EXISTS	0x50C	1292
IL_OUT_FORMAT_SAME	0x50D	1293
IL_STACK_OVERFLOW	0x50E	1294
IL_STACK_UNDERFLOW	0x50F	1295
IL_INVALID_CONVERSION	0x510	1296
IL_BAD_DIMENSIONS	0x511	1297
IL_FILE_READ_ERROR	0x512	1298
IL_LIB_JPEG_ERROR	0x5E2	1506
IL_LIB_PNG_ERROR	0x5E3	1507
IL_LIB_TIFF_ERROR	0x5E4	1508
IL_LIB_MNG_ERROR	0x5E5	1509
IL_UNKNOWN_ERROR	0x5FF	1535

Table A-1. *DevIL error codes*

Appendix B: Supported File Formats

DevIL supports loading and saving of a large number of image formats. Table B-1 lists the formats DevIL supports.

Formats Supported by DevIL			
Loading		***Saving***	
Type	**Extension(s)**	**Type**	**Extension(s)**
Windows Bitmap	.bmp	Windows Bitmap	.bmp
Dr. Halo Cut File	.cut	C-style header	.h
DirectDraw Surface	.dds	DirectDraw Surface	.dds
Graphics Interchange Format	.gif	Jpeg	.jpg, .jpe, .jpeg
Icons	.ico, .cur	Palette	.pal
Jpeg	.jpg, .jpe, .jpeg	ZSoft PCX	.pcx
Interlaced Bitmap	.lbm	Portable Network Graphics	.png
Homeworld File	.lif	Pnm	.pbm, .pgm, .ppm
Doom Walls / Flats	.lmp	Raw Data	.raw
Half-Life Model	.mdl	Silicon Graphics	.sgi, .bw, .rgb, .rgba
Mng Animation	.mng	Targa	.tga
PhotoCD	.pcd	TIF	.tif, .tiff
ZSoft PCX	.pcx		
PIC	.pic		
PIX	.pix		
Portable Network Graphics	.png		
Pnm	.pbm, .pgm, .ppm, .pnm		
PhotoShop	.psd		
Pixar	.pxr		
Silicon Graphics	.sgi, .bw, .rgb, .rgba		
Targa	.tga		
TIF	.tif, .tiff		
Quake2 Texture	.wal		
X Pixel Map	.xpm		

Raw Data	Any		

Table B-1. *Types of image file formats DevIL supports*

ilLoad, **ilLoadF** and **ilLoadL** all accept a *Type* parameter. Possible values for the *Type* parameter for each function are listed below. If you use an unsupported value for *Type*, then the function generates an IL_INVALID_ENUM error and returns IL_FALSE.

Supported Loading Types					
DevIL #define	**Hex Value**	**Decimal Value**	**Supported by ilLoad**	**Supported by ilLoadF**	**Supported by ilLoadL**
IL_BMP	0x420	1056	v	v	v
IL_CUT	0x421	1057	v	v	v
IL_DCX	0x438	1080	v	v	v
IL_DDS	0x437	1079	v	v	v
IL_DOOM	0x422	1058	v	v	v
IL_DOOM_FLAT	0x423	1059	v	v	v
IL_GIF	0x436	1078	v	v	v
IL_ICO	0x424	1060	v	v	v
IL_JNG	0x435	1077	v	v	v
IL_JPG (IJL)	0x425	1061	v		v
IL_JPG (libjpeg)	0x425	1061	v	v	v
IL_LBM	0x426	1062	v	v	v
IL_LIF	0x434	1076	v	v	v
IL_MDL	0x431	1073	v	v	v
IL_MNG	0x435	1077	v	v	v
IL_PCD	0x427	1063	v	v	v
IL_PCX	0x428	1064	v	v	v
IL_PIC	0x429	1065	v	v	v
IL_PIX	0x43C	1084	v	v	v
IL_PNG	0x42A	1066	v	v	v
IL_PNM	0x42B	1067	v	v	v
IL_PSD	0x439	1081	v	v	v
IL_PSP	0x43B	1083	v	v	v
IL_PXR	0x43D	1085	v	v	v
IL_RAW	0x430	1072	v	v	v

IL_SGI	0x42C	1068	v	v	v
IL_TGA	0x42D	1069	v	v	v
IL_TIF	0x42E	1070	v	v	v
IL_WAL	0x432	1074	v	v	v
IL_XPM	0x43E		v	v	v

Table B-2. *Values for the* Type *parameter*

ilSave, **ilSaveF** and **ilSaveL** all accept a *Type* parameter. Possible values for the *Type* parameter for each function are listed below. If you use an unsupported value for *Type*, then the function generates an IL_INVALID_ENUM error and returns IL_FALSE.

Supported Saving Types					
DevIL #define	**Hex Value**	**Decimal Value**	**Supported by ilSave**	**Supported by ilSaveF**	**Supported by ilSaveL**
IL_BMP	0x420	1056	v	v	v
IL_CHEAD	0x42F	1071	v	v	v
IL_DDS	0x437	1079	v	v	v
IL_JPG (IJL)	0x425	1061	v		v
IL_JPG (libjpeg)	0x425	1061	v	v	v
IL_PCX	0x428	1064	v	v	v
IL_PNG	0x42A	1066	v	v	v
IL_PNM	0x42B	1067	v	v	v
IL_RAW	0x430	1072	v	v	v
IL_SGI	0x42C	1068	v	v	v
IL_TGA	0x42D	1069	v	v	v
IL_TIF	0x42E	1070	v		

Table B-3. *Values for the* Type *parameter*

Index