Developer's Image Library manual

Table of Contents

IL manual

This is a manual describing IL part of `DevIL` – handling images.

1 Introduction

1.1 General 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 into 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.

1.2 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.

2 Library setup

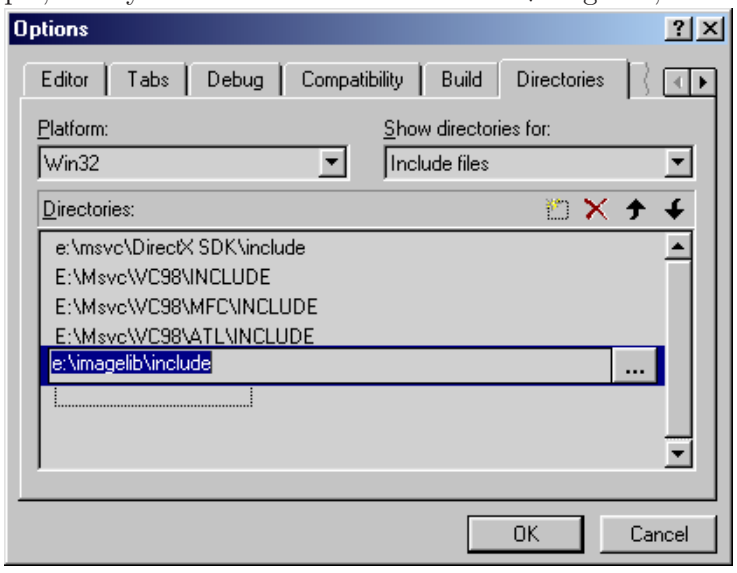
2.1 Microsoft Visual C++ setup

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.sln file in the install directory to load the DevIL workspace in Microsoft Visual C++ (MSVC++).

2.1.1 Directories

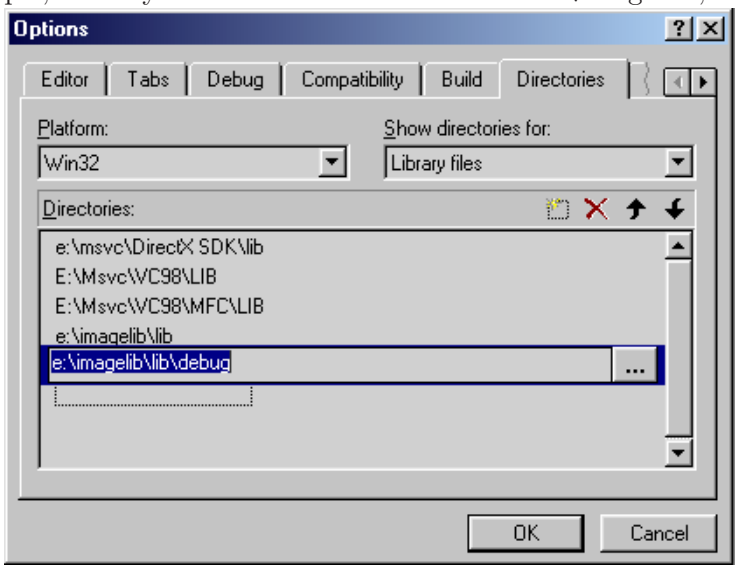
You will need to change some directory settings in MSVC++ to get DevIL working.

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'.



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 exam-

ple, 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.

2.1.2 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.

2.1.3 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.

Generally, DevIL is not thread safe. You should make sure that threads in your application do not use DevIL at the same time.

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.

2.2 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 '`-lilu`' and '`-lilut`', respectively.

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

Setting up DevIL in this environment requires the following steps:

1. Unpack DevIL to your favourite build area: Typically this is done by running `tar -xvvf devil-xxx.tar.gz`
2. Go to the root devil directory and run `./configure <your-favourite options>`

   TIP: Typically, you will want to specify `--prefix=/usr` (where to install DevIL), `--enable-ILU` and/or `--enable-ILUT` and probably also `--with-examples`. Running `configure --help` gives you an exhaustive list of possibilities how to tweak DevIL build.
3. If no errors occured and you are satisfied with the configure report, you can compile it by running `make`

   TIP: You can run `make -j3` if you have a dual-core processor. Generally, if you replace the number '3' with 'number of CPU cores' + 1, you are likely to get the job done in the shortest time.
4. Run `make check` to check whether everything works as it should. See [tests], page 6, for hints what to do if a test fails.
5. Install DevIL by running `make install` as superuser.

   TIP: You can override variables at make time if you forgot to do that at configure time. For instance, you can install by `make install prefix=/usr`

In order to link to DevIL, you may use autotools with `libtool`, which means to link either with '`libIL.la`' or '`libILU.la`' or with '`libILUT.la`' depending what level of functionality you require. You may decide to use a monolithic build of the library and you then link with '`libDevIL.la`'. The best way is to have this code in your '`configure.ac`' file:

```
...
dnl Check for libtool (older macro for this is AC_PROG_LIBTOOL)
LT_INIT
...
# Check for libs we need for our program now
dnl Check that we have the header
AC_CHECK_HEADER([IL/il.h])
dnl Check for the IL part on DevIL on unix-like systems
PKG_CHECK_MODULES([DEVIL],
                  [IL ILU ILUT])
dnl Check that DevIL library exists (good for Windows)
AC_CHECK_LIB([DevIL], [main], [LIBS_WE_NEED="-lDevIL $LIBS_WE_NEED"])
...
dnl Now export the variable that contains libraries
dnl  so we can use it in makefiles
AC_SUBST([LIBS_WE_NEED])
AC_SUBST([DEVIL_CFLAGS DEVIL_LIBS])
...
```

If you are skilled with automake, you may link with the libtool files directly, which is more portable (`pkg-config` currently breaks cross-compilation).

If you use an IDE or if you don't like autotools (which is a big mistake, by the way :-), then you may use '`pkg-config`' as a program. This is not recommended if you intend to use `DevIL` in a cross-platform programs. You can get libraries you need to link to by running `pkg-config IL --libs`. Again, you decide whether you need `IL`, `ILU` or `ILUT`. Also pass `pkg-config IL --cflags` to the compiler!

If you are a happy IDE user, you have to write this commands in the backquotes like '`pkg-config ILUT --libs`' into some text boxes that allow you to specify additional `LDFLAGS`.

2.4 Test and examples

2.4.1 Test suite

`DevIL` now comes with a test suite. If you can use autotools to configure and compile it, you can also run the test suite by executing `make check` in the directory where you ran `configure`.

There are following tests available:

- Format test: An image is generated and saved to disc. It is loaded afterwards and compared to the original. If they are pretty much the same, the test is passed.
- Format test 2 (coming soon): Some image formats can't be saved. So a test images are provided and loaded and compared.

2.4.2 Examples

`IL` examples:

- Simple
- Read

- Override

Note: Those examples can be linked against `ILU`, but this is only because error report functions.

`ILUT` examples:

- Allegro
- C++ wrapper
- SDL
- Volume
- OpenGL

3 Basic usage

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 as well.

3.1 Initializing DevIL

3.1.1 IL Initialization

Simply call the ilInit function with no parameters:

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

3.1.2 ILU Initialization

Call the iluInit function with no parameters:

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

3.1.3 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);
```

3.2 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);
```

3.2.1 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.

3.2.2 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 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.

3.2.3 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.

3.3 File handling – loading images

DevIL's main purpose is to load images. DevIL's loading is designed to be extremely easy but very powerful.

See [file_formats], page 32, lists the image types DevIL is capable of loading.

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

```
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);
```

3.3.1 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 (see [registration], page 16).

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.

3.3.2 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.

3.3.3 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.

See [registration], page 16, for instructions on how to use your own file handling functions and file handles.

3.3.4 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.

3.3.5 Saving to Files

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);
ILuint ilSaveL(ILenum Type, ILvoid *Lump, ILuint Size);
```

DevIL's saving functions are identical to the loading functions, despite the fact that they save images instead of load images.

Typically, the user does not know exactly how large the output image will be. If you pass `NULL` for `Lump` and 0 for `Size` to `ilSaveL`, `ilSaveL` will return the buffer size needed to save an image of `Type`. When a buffer is passed for `Lump`, the return value is how many bytes were written to the buffer.

Note that not all formats that have load support have also save support (see [file_formats], page 32)

4 Image management

4.1 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);
```

`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. See [format #defines], page 31, for what you can pass.
- `Type`: The type of image data. Usually, this will be `IL_UNSIGNED_BYTE`, unless you want to utilize multiple bytes per colour channel. See [type #defines], page 31, for acceptable type.
- `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`.

4.2 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);
```

4.2.1 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.

4.2.2 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
- `Format, Type, Data`: These are basically the same as ones described above. see [ilTexImage reference], page 12.

4.3 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);
```

4.3.1 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`.

4.3.2 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`:

- Previous parameters are the same as in `ilTexImage`

– 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.

4.4 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);
```

4.4.1 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.

4.4.2 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.

4.4.3 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.

4.4.4 Blit/Overlay Behavior

By default, `ilBlit` and `ilOverlay` will blend the source with the destination image if the source has an alpha channel present. If you need to blit the image without blending, you can use the `IL_BLIT_BLEND` #define. This behavior can be toggled with `ilEnable` and `ilDisable`.

```
ilDisable(IL_BLIT_BLEND);  // Turns off blending
ilEnable(IL_BLIT_BLEND);   // Turns on blending
```

5 Image Characteristics

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

5.1 Origin

Depending on the file format, data can start in the upper left or the lower left corner of the image. By default, `DevIL` keeps the origin in the same place as the original image. This can cause your image to be flipped vertically if the image you are loading has an origin other than what you expect. To obtain the origin of the image, use `ilGetInteger`.

```
ilGetInteger(IL_IMAGE_ORIGIN);
```

To force `DevIL` to use just one origin, you need to use the following code:

```
ilEnable(IL_ORIGIN_SET);
ilSetInteger(Origin);
```

Origin is either `IL_ORIGIN_LOWER_LEFT` or `IL_ORIGIN_UPPER_LEFT`. Finally, if you need to find out which origin mode is currently set, use:

```
ilGetInteger(IL_ORIGIN_MODE);
```

5.2 Format

Format refers to the ordering of the bytes for each pixel.

5.3 Registration

6 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);
```

6.1 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.

See [error_codes], page 31, for error codes that can be returned by ilGetError.

6.2 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.

6.2.1 Languages

The ILU error messages have been translated into multiple languages: Arabic, Dutch, German, Japanese and Spanish. The default language is English.

6.2.2 Selecting a Language

iluSetLanguage will change the error string returned by iluErrorString to the language specified in its parameter. Languages supported are: English, Arabic, Dutch, German, Japanese and Spanish. See [language #defines], page 31, for a list of possible values.

Be aware that if the Unicode version of DevIL is not being used, some translations will not display properly. An example is Arabic, which uses characters outside of the standard ASCII character set.

7 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 the below figure.

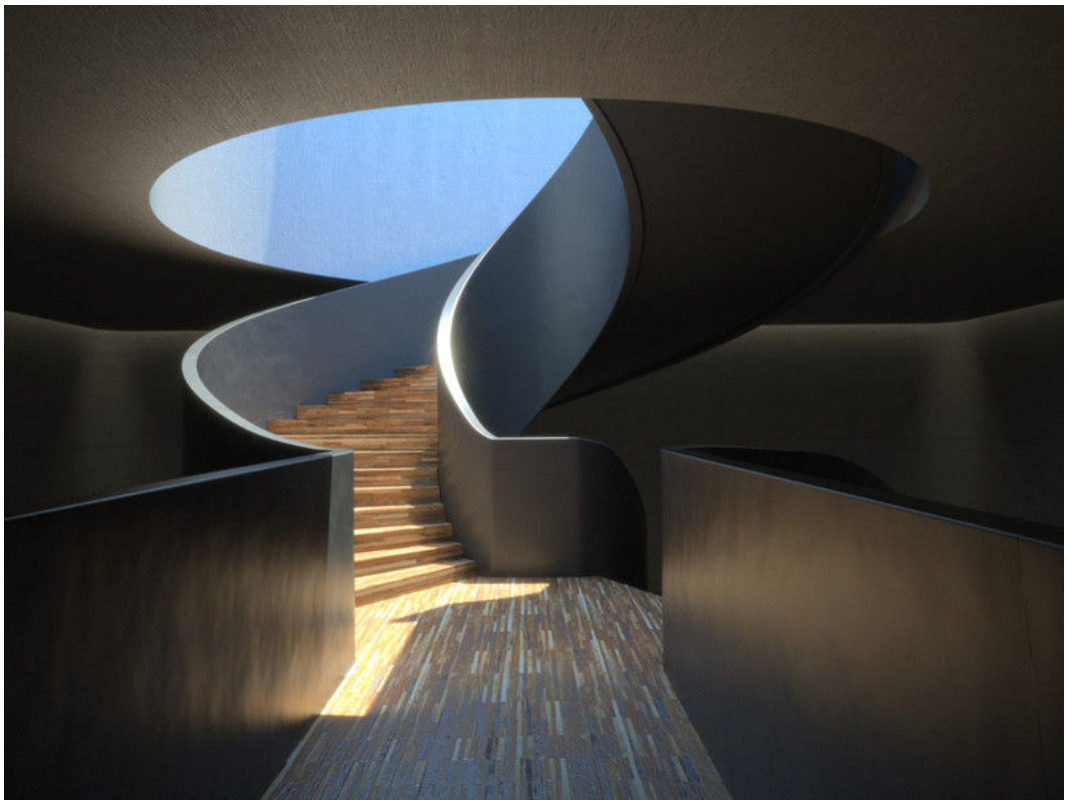


Figure 7.1: Original, unmodified image

This is a crop of a Bertrand Benoit's image taken from Blender art gallery, and Bertrand has kindly allowed us to use it for demonstrations. You can check out his website. Thank you, Bertrand!

The image samples here have a better-than-bad resolution, so you don't have to be afraid to zoom at them if you wish to see details.

7.1 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. The figure below illustrates this effect on the `DevIL` logo.

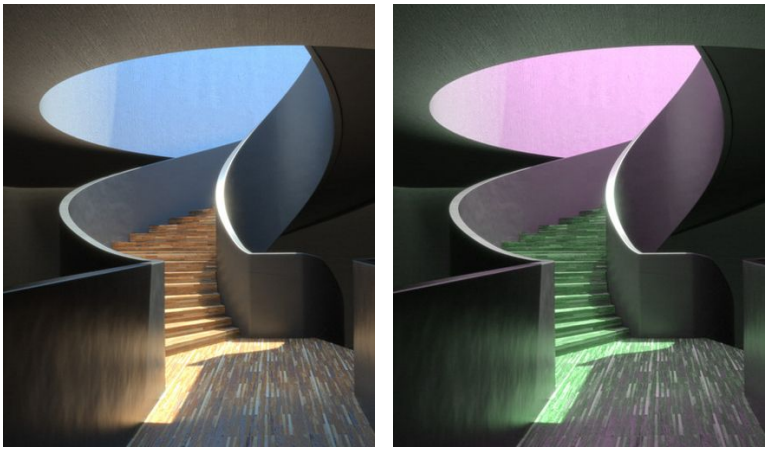


Figure 7.2: Original and "alienified" image

7.2 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 gamedev.net.

`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.

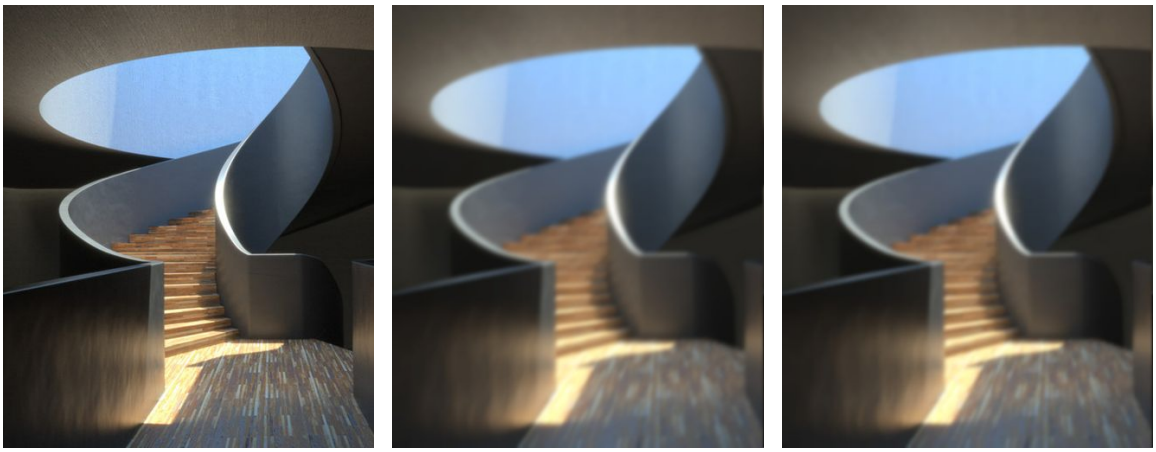


Figure 7.3: Original image, average blurred and gaussian blurred, both with 10 iterations applied

7.3 Contrast

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.

- values from 0.0 to 1.0 decrease the amount of contrast in the image.
- 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 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.

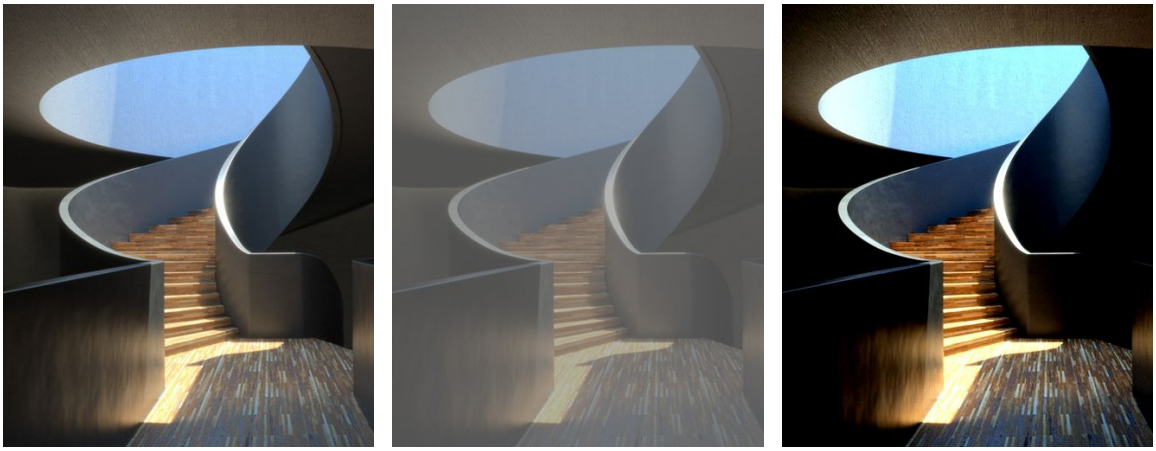


Figure 7.4: Original image, image with contrast of 0.4 and with contrast of 1.7

7.4 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. The below figure shows the results of applying `iluEqualize` to the `DevIL` image.

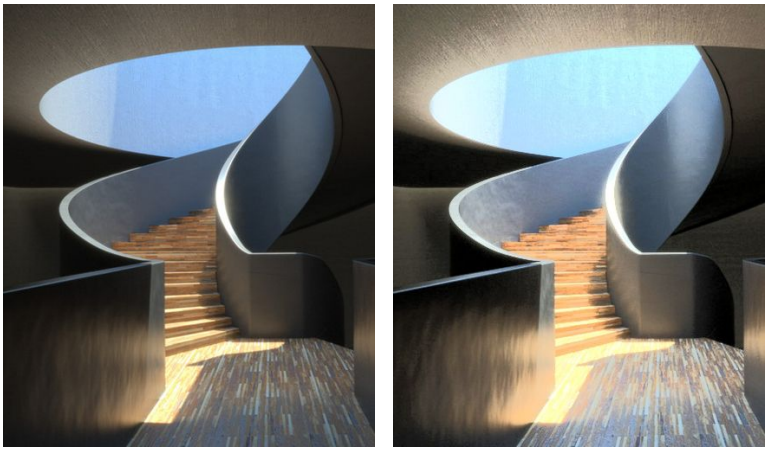


Figure 7.5: Original image and equalized image

7.5 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.

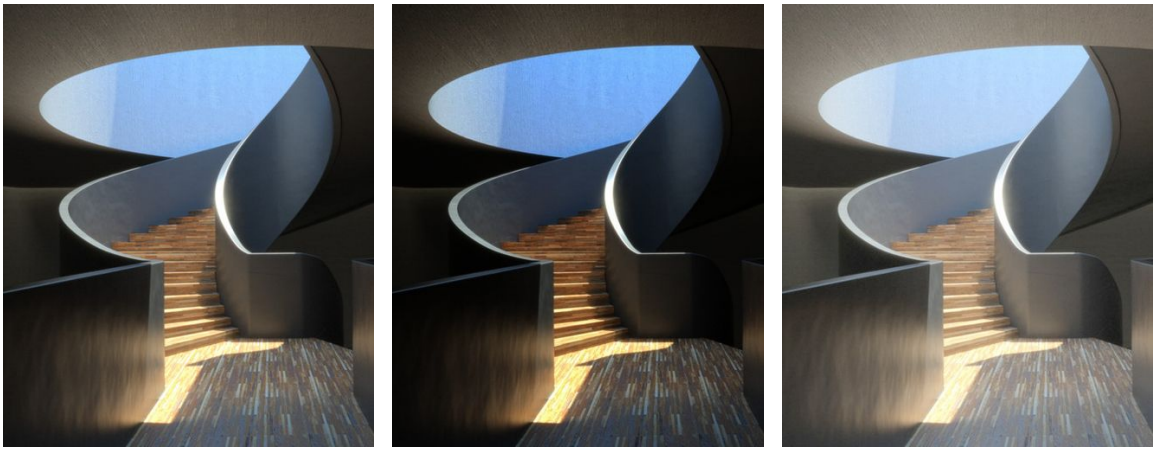


Figure 7.6: Original image, image with gamma 0.7 and with with gamma 1.6

7.6 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.

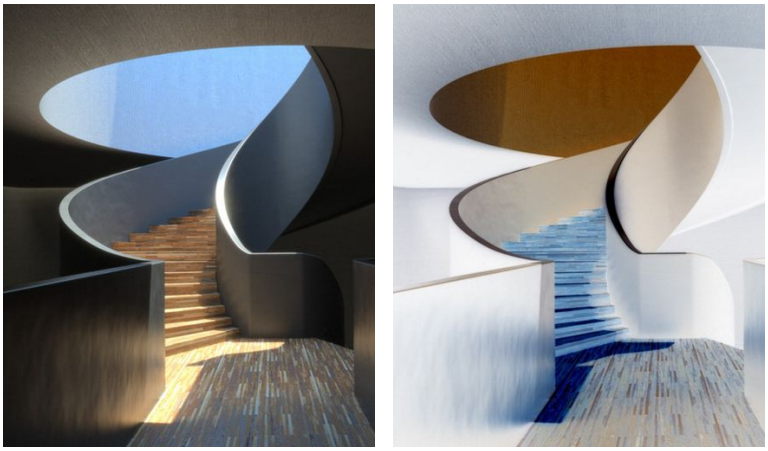


Figure 7.7: Original and negative image

7.7 Noise

ILU 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 ILU 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.

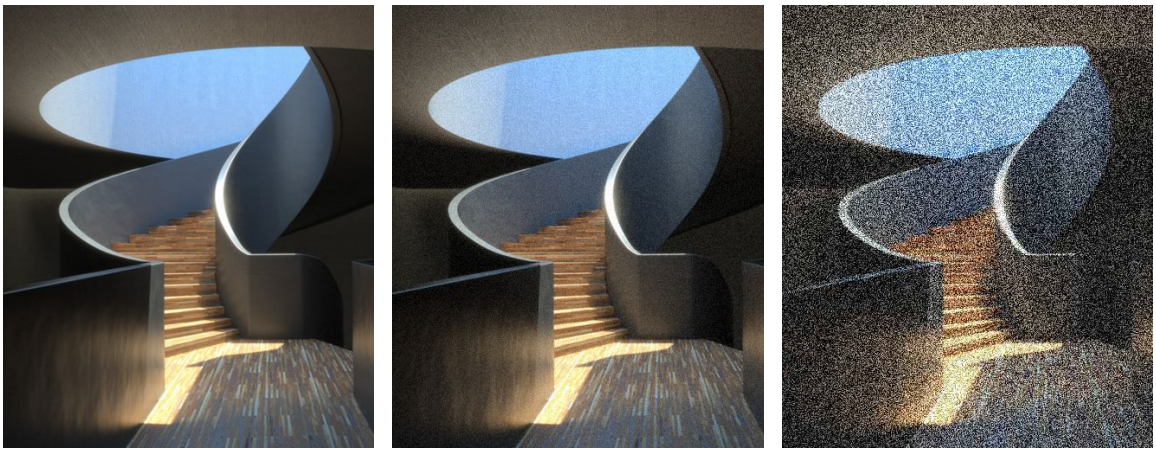


Figure 7.8: Original image, noisified image 0.1, noisified 0.8

`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.

7.8 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.

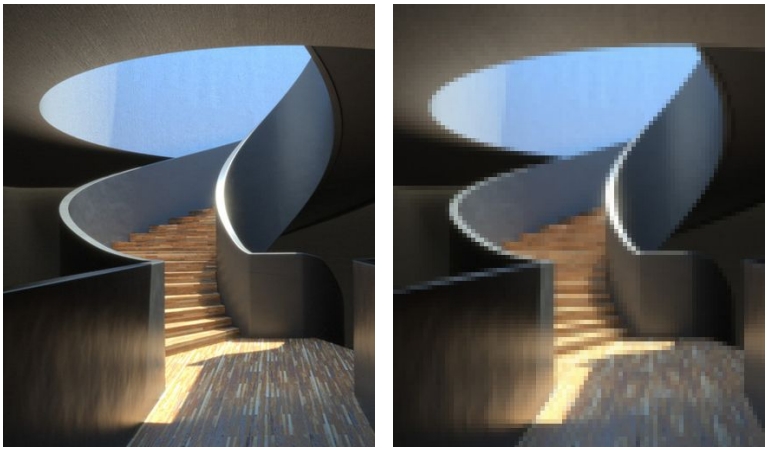


Figure 7.9: Pixelization of 5 pixels across

7.9 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.

- values from 0.0 to 1.0 do a type of reverse sharpening, blurring the image.
- 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 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.

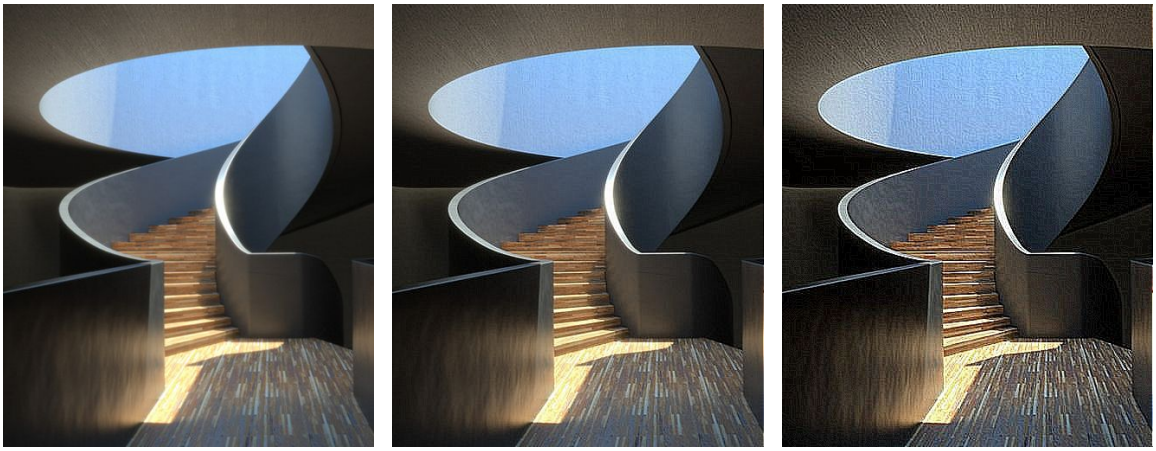


Figure 7.10: Original image, image sharpened by 1.8 in 2 iterations and sharpened by 2.1 in 3 iterations

8 Resizing Images

8.1 Basic Scaling

To resize images, use the `iluScale` function:

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

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.

8.2 Advanced Scaling

ILU also allows you to specify which method you want to use to resize images. As you can see in the middle figure above, the enlarged image is very pixelized. 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.

ILU allows you to use different filters to produce better scaling results via `iluImageParameter`:

- 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);
```

8.3 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. ILU 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.

9 Sub-Images

9.1 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.

All mipmap levels down to 1x1

9.1.1 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.

9.1.2 Mipmap Access

Access mipmaps through the iluActiveMipmap function:

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

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.

9.2 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.

9.2.1 Animation Chain Creation

To be added...

9.2.2 Animation Chain Access

Access animations through the iluActiveImage function:

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

iluActiveImage sets the current image to the *ImageNum* animation frame of the current image. If there are no animation frames present, then iluActiveImage returns IL_FALSE, else it returns IL_TRUE. The base image is animation frame 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.

9.3 Layers

DevIL does not have a full layer implementation yet.

9.4 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 after iluActiveImage, a mipmap from the selected image in the animation chain is chosen.

10 DXTC/S3TC Notes

10.1 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.

10.1.1 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.

10.1.2 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.

10.1.3 Compression Method

DevIL can use the nVidia Texture Tools (NVTT) library, the libsquish library and its own internal compressor to generate DXTC data. By default, DevIL uses its internal compressor. This compressor is fast but is not very high quality. NVTT is usually CUDA-enabled, meaning that it can run quickly on computers with GeForce 8-series and higher cards. libsquish generates images with the highest quality possible, but it can be very slow.

To enable compression by NVTT or libsquish, use one of the following lines of code:

```
ilEnable(IL_NVIDIA_COMPRESS);
ilEnable(IL_SQUISH_COMPRESS);
```

You can also disable compression by these libraries by using ilDisable. If both are enabled, NVTT is used.

10.2 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);
```

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.

10.3 Compressing DXTC Data

In the previous section, it was mentioned that DevIL can compress the data of an image with DXT compression. If you have image data in your program that you want to compress, you can use the ilCompressDXT function.

```
ILubyte *ilCompressDXT(ILubyte *Data, ILuint Width, ILuint Height,
                       ILuint Depth, ILenum DXTCFormat, ILuint *DXTCSize);
```

Data must be in BGRA format for NVTT and DevIL's compressor, and it must be in RGBA format for libsquish. Please keep this in mind when calling this ilCompressDXT. Look at Compression Method in this section for information on how to use these libraries.

10.4 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.

10.4.1 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.

10.4.2 Direct3D DXTC Support

ILUT's Direct3D (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 #defines

Here goes lists of DevIL #defines used in functions that manipulate image data. As you can see, they are self-explanatory.

A.1 format-related #defines

IL_COLOUR_INDEX
IL_RGB
IL_RGBA
IL_BGR
IL_BGRA
IL_LUMINANCE

A.2 type-related #defines

IL_BYTE
IL_UNSIGNED_BYTE
IL_SHORT
IL_UNSIGNED_SHORT
IL_INT
IL_UNSIGNED_INT
IL_FLOAT
IL_DOUBLE

A.3 Language-related #defines

IL_ENGLISH
IL_ARABIC
IL_DUTCH
IL_GERMAN
IL_JAPANESE
IL_SPANISH

Appendix B 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	Hex 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_LIB_JP2_ERROR	0x5E6	1510
IL_LIB_EXR_ERROR	0x5E7	1511
IL_UNKNOWN_ERROR	0x5FF	1535

Appendix C Supported File Formats

DevIL supports loading and saving of a large number of image formats. Table lists the formats DevIL supports sorted according to #define.

Format name	Extension	IL #define	Loading?	Saving?
Blizzard texture	.blp	IL_BLP	yes	no
Windows bitmap	.bmp	IL_BMP	yes	yes
C-style header	.h	IL_CHED	no	yes
Dr. Halo Cut File	.cut	IL_CUT	yes	no
ZSoft Multi-PCX	.dcx	IL_DCX	yes	no
DICOM	.dicom, .dcm	IL_DCM	yes	no
DirectDraw surface	.dds	IL_DDS	yes	yes
Digital Picture Exchange	.dpx	IL_DPX	yes	no
DOOM walls/flats	.lmp	IL_DOOM, IL_DOOM_FLAT	yes	no
OpenEXR	.exr	IL_EXR	yes	yes
Flexible Image Transport System	.fits, .fit	IL_FITS	yes	no
Heavy Metal: FAKK 2 Texture	.ftx	IL_FTX	yes	no
Graphics Interchange Format	.gif	IL_GIF	yes	no
Radiance High Dynamic Range	.hdr	IL_HDR	yes	yes
Macintosh Icons	.icns	IL_ICNS	yes	no
Windows Icons	.ico, .cur	IL_ICO	yes	no
Interchange File Format	.iff	IL_IFF	yes	no
Infinity Ward Image	.iwi	IL_IWI	yes	no
Jpeg Network Graphics	.jng	IL_JNG	yes	no
Jpeg 2000	.jp2	IL_JP2	yes	yes
Jpeg	.jpg, .jpe, .jpeg	IL_JPG	yes	yes
Interlaced Bitmap	.lbm	IL_LBM	yes	no
Homeworld File	.lif	IL_LIF	yes	no
Half-Life Model	.mdl	IL_MDL	yes	no
Mng Animation	.mng	IL_MNG	yes	no
Mp3	.mp3	IL_MP3	yes	no
PhotoCD	.pcd	IL_PCD	yes	no
ZSoft PCX	.pcx	IL_PCX	yes	yes
Softimage PIC	.pic	IL_PIC	yes	no
PIX	.pix	IL_PIX	yes	no
Portable Network Graphics	.png	IL_PNG	yes	yes
Pnm	.pbm, .pgm, .ppm, .pnm	IL_PPM	yes	yes
Adobe PhotoShop	.psd	IL_PSD	yes	yes
PaintShop Pro	.psp	IL_PSP	yes	no

Pixar	.pxr	IL_PXR	yes	no
Raw Data	*	IL_RAW	yes	yes
Homeworld 2 Texture	.rot	IL_ROT	yes	no
Silicon Graphics	.sgi, .bw, .rgb, .rgba	IL_SGI	yes	yes
Sun RAS	.sun, .ras, .rs, .im*	IL_SUN	yes	no
Creative Assembly Texture	.texture	IL_TEXTURE	yes	no
Targa	.tga	IL_TGA	yes	yes
Tagged Image File Format	.tif, .tiff	IL_TIF	yes	yes
Gamecube Texture	.tpl	IL_TPL	yes	no
Unreal Texture	.utx	IL_UTX	yes	no
Valve Texture	.vtf	IL_VTF	yes	yes
Quake2 Texture	.wal	IL_WAL	yes	no
HD Photo	.wdp, .hdp	IL_WDP	yes	no
X Pixel Map	.xpm	IL_XPM	yes	yes

Exception: IL_JPG (IJL) type is not supported by ilLoadF nor by ilSaveF. IL_JPG (libjpeg) is supported by both.

Appendix D Sample DevIL program

If you are not used to this approach, you may be grateful for a short program demonstrating how to actually use DevIL:

```
#include<IL/il.h>
#include<stdlib.h>   /* because of malloc() etc. */

int main(int argc, const char * argv[])
{
    ILuint handle, w, h;
    /* First we initialize the library. */
    /*Do not forget that... */
    ilInit();
    /* We want all images to be loaded in a consistent manner */
    ilEnable(IL_ORIGIN_SET);
    /* In the next section, we load one image */
    ilGenImages(1, & handle);
    ilBindImage(handle);
    ILboolean loaded = ilLoadImage("original_file.jpg");
    if (loaded == IL_FALSE)
       return -1; /* error encountered during loading */
    /* Let's spy on it a little bit */
    w = ilGetInteger(IL_IMAGE_WIDTH);    // getting image width
    h = ilGetInteger(IL_IMAGE_HEIGHT);   // and height
    printf("Our image resolution: %dx%d\n", w, h);
    /* how much memory will we need? */
    int memory_needed = w * h * 3 * sizeof(unsigned char);
    /* We multiply by 3 here because we want 3 components per pixel */
    ILubyte * data = (ILubyte *)malloc(memory_needed);
    /* finally get the image data */
    ilCopyPixels(0, 0, 0, w, h, 1, IL_RGB, IL_UNSIGNED_BYTE, data);
    /* We want to do something with the image, right? */
    int i;
    for(i = 0; i < memory_needed; i++)
       if(i % 31 == 0) /* vandalise the image */
          data[i] = i % 255;
    /* And maybe we want to save that all... */
    ilSetPixels(0, 0, 0, w, h, 1, IL_RGB, IL_UNSIGNED_BYTE, data);
    /* and dump them to the disc... */
    ilSaveImage("our_result.png");
    /* Finally, clean the mess! */
    ilDeleteImages(1, & handle);
    free(data); data = NULL;
    return 0;
}
```

Functions index