

PixelLight Build Documentation



February 23, 2012
PixelLight 0.9.11-R1



The content of this PixelLight document is published under the Creative Commons
Attribution-NonCommercial-ShareAlike 3.0 Unported
Copyright © 2002-2012 by The PixelLight Team

Contents

| | |
|--|-----------|
| 1. Introduction | 5 |
| 2. External Dependencies | 9 |
| 2.1. Dependencies | 9 |
| 2.2. External Packages | 9 |
| 3. Windows | 13 |
| 3.1. CMake | 13 |
| 3.1.1. Prerequisites for Binaries | 13 |
| 3.1.2. Prerequisites for Documentation | 14 |
| 3.1.3. Prerequisites for Installable Software Development Kit (SDK) | 15 |
| 3.1.4. Create Solutions and Build | 15 |
| 3.2. Running from a Local Build | 16 |
| 3.3. Microsoft Visual C++ (MSVC) Solutions within the Git Repository | 17 |
| 3.4. Using the Build PixelLight Version in MSVC | 17 |
| 4. Linux | 19 |
| 4.1. Prerequisites | 19 |
| 4.2. External Packages | 22 |
| 4.3. CMake | 22 |
| 4.4. Maketool Script | 23 |
| 4.5. Build | 24 |
| 4.6. Create Documentation and Packages | 24 |
| 4.7. Running from a Local Build and Installing | 25 |
| 4.7.1. 1. Run from a your Local Source Directory | 25 |
| 4.7.2. 2. Install the PixelLight SDK Locally | 26 |
| 5. Mac OS X | 27 |
| 6. Android | 29 |
| 6.1. Prerequisites | 29 |
| 6.2. Create Makefiles and Build | 30 |
| 6.3. Using the Android Port | 31 |
| 7. Contact | 37 |
| A. FindPixelLight.cmake | 39 |

| | |
|--|-----------|
| B. Android Native Build Tutorial | 41 |
| B.1. Prerequisites | 41 |
| B.2. Android Emulator and Device | 43 |
| B.3. Android Native Development Kit (NDK) Build System | 44 |
| B.4. CMake Build System | 46 |
| B.4.1. First Experiment | 46 |
| B.4.2. Native Activity Experiment | 47 |
| B.5. Glossary | 49 |
| B.6. Command Glossary | 49 |
| B.7. Possible issues | 50 |
| C. Mac OS X GCC | 53 |
| Abbreviations | 55 |

1. Introduction

DON'T PANIC

(The Hitchhiker's Guide to the Galaxy by Douglas Adams)

Target Audience This document is meant for programmers. Please note that also trivial stuff will be mentioned. A lot.

Motivation This document describes how to build PixelLight from the sources. Don't be shocked when looking at the size of this document, this doesn't imply that it's highly complicated or near impossible to build the project. Right from the beginning, one of our goals was, that it should be as easy as possible to build the technology. A lot of efforts were and are put into this goal as one may see when looking at this document or the fact that such a document exists.

Sadly, across all the supported platforms and external dependencies there may be some pitfalls - especially for users without much experience within a certain target platform or tool set. The goal of this document is to provide as much information as possible to minimize the frustration of building PixelLight. Please note that this document can't avoid frustration completely, especially if you plan to compile PixelLight with a not officially supported compiler or for a new, untested platform.

Due to the really small development team compared to the dimension of the project, we can't support every compiler or Integrated Development Environment (IDE) existing out there (meaning adding support and especially maintaining it). We have to focus on the mainstream, or what we consider as mainstream. Currently we're using the following compilers and compiler versions:

- Microsoft (MS) Windows: Microsoft Visual Studio 10
- Linux: GNU Compiler Collection (GCC) 4.6
- Linux: Clang 3.0
- Mac OS X: GCC 4.2.1
- Android NDK Toolchain

If you stick to those, you should be on the safe side. Other compilers and compiler versions may work as well, but are untested. There's support for 32 bit and 64 bit.

1. Introduction

In general, if you encounter time consuming pitfalls not yet described in this document, please tell us by using e.g. the official forum at <http://dev.pixellight.org/forum/> or the bugtracker at http://sourceforge.net/tracker/?group_id=507544&atid=2063682. If there's no feedback, we can't improve things to make it even easier to use PixelLight in the future.

Linux Especially the Linux part is quite detailed due to the fact that it's the base of several other ports like Android and Mac OS X, even when a platform is no direct derivation of Linux. Additionally, this document should also enable MS Windows user, without or with just a little experience with Linux, to build PixelLight under Linux.

CMake To build PixelLight across all supported platforms in an uniform way, we're using CMake (<http://www.cmake.org/>).

Source Codes of Releases At <http://sourceforge.net/projects/pixellight/files/> are the packed sources codes of every PixelLight release available for download. For example, *PixellLight-0.9.11-R1-SourceCodes.tar.gz* are the packed source codes of the release *0.9.11-R1*. There are also packages containing all public external dependencies used to compile this release version.

Latest Source Codes We use Git¹ version control to manage our source code repository. This repository contains the main source code of the PixelLight framework. In case you want to access the latest source codes for the next release, you can use this repository. Please note that the latest commit(s) may have broken something or introduced not yet fixed or even recognized new issues. Although we always try keep the version within the Git repository usable, it may even not be compilable for a short time after a breaking commit. So, if you want to be on the safe side, use the sources codes of an official PixelLight release as mentioned in the paragraph above. These versions are tagged within the repository. For anonymous access, it is readable only. To ensure high quality source code, write access is only available to team members of the development team.

To checkout the current source code, use the following Git command line:

```
1 git clone git://pixellight.git.sourceforge.net/gitroot/pixellight/  
   pixellight
```

In case you're a PixelLight team member, you might want to clone by using your SourceForge username by writing

```
1 git clone ssh://<user name>@pixellight.git.sourceforge.net/gitroot/  
   pixellight/pixellight
```

Example:

¹<http://www.git-scm.com>

```
1 git clone ssh://bob42@pixellight.git.sourceforge.net/gitroot/pixellight/  
  pixellight
```

Using the *ssh*-protocol might reduce the risk of firewall problems.

Or use this Uniform Resource Locator (URL) to checkout the source code with your favourite Git client:

```
1 git://pixellight.git.sourceforge.net/gitroot/pixellight/pixellight
```

In case you get for instance the following error message during cloning the Git repository

```
fatal: Unable to look up pixellight.git.sourceforge.net (port 9418) (No such  
host is known. )
```

you might want to check your firewall settings. Additionally you can browse the Git repository within your web browser by opening pixellight.git.sourceforge.net/git/gitweb.cgi?p=pixellight/pixellight;a=summary to ensure that the server is not down (doesn't happen that often).

You can use the following Really Simple Syndication (RSS) feed to get informed automatically about changes within the public Git repository:

```
1 http://pixellight.git.sourceforge.net/git/gitweb.cgi?p=pixellight/  
  pixellight;a=rss
```


2. External Dependencies

In order to build the PixelLight engine for instance with MSVC, all the external packages used by the engine need to be in the right place for the build. When using the project files generated by CMake and there's an Internet connection, those external packages are downloaded and extracted automatically during the build. So, this chapter does only exist in case this automatic process is not working for you or you don't use the CMake way at all. Have a look at the *External*-directory inside the Git repository. There you can find a readme file for every library, that describes what files are needed.

2.1. Dependencies

When writing this documentation in November 2011, there are 38 external dependencies. Please note that this doesn't mean that you have to take care of 38 external libraries before you can start building and using PixelLight. In the minimal build, PixelLight only depends on PCRE used within PLCore for regular expressions. zlib for zip support within PLCore is highly recommended, but can be deactivated. The same is for jpeg and png support within PLGraphics. Nice to have but no must have, especially when you just want to do your first steps with building PixelLight. As you can see, the 12 base projects of PixelLight don't have a lot of external dependencies.

Everything else is completely optional. Most of the projects are just dynamically loaded plugins. If you need no Cg shaders, don't use Cg. If you need no Newton physics, don't use it. For scripting, you only need to compile PLCore and e.g. the Lua scripting plugin. That's it. For developing new plugins, you even don't need to compile PixelLight itself, just start a new plugin project and PixelLight will be able to use the new dynamic library providing e.g. Run-Time Type Information (RTTI) classes for new script languages.

This means that the base of PixelLight is really slim when it comes to external dependencies. Those 38 external libraries only mean that there are a lot of optional plugins available for you so that you don't have to write them by yourself if you need them. It's a gift from us, to you.

2.2. External Packages

You can find the libraries pre-packed in the files-section on our homepage at <http://pixellight.sourceforge.net/externals/>. When opening this URL within a web browser, you'll see a file structure like

2. External Dependencies

- `Linux-ndk_armeabi-v7a_32/`
- `Linux-ndk_armeabi_32/`
- `Linux_x86_32/`
- `Linux_x86_64/`
- `MacOSX_x86_32/`
- `MacOSX_x86_64/`
- `Windows_x86_32/`
- `Windows_x86_64/`

There's one directory containing pre-packed external dependencies per platform variation. By default, the CMake based build process will download the packages from this location automatically. Within the CMake-GUI it's possible to change this external packages URL. You can also download the packages manually and extract them individually at the right place. Either way, it's always the same principle and data. Don't get confused.

Package examples Here's are a few package examples so you're able to get the idea:

- zlib package for MS Windows 32 bit: http://pixellight.sourceforge.net/externals/Windows_x86_32/zlib.tar.gz
- zlib package for Linux 32 bit: http://pixellight.sourceforge.net/externals/Linux_x86_32/zlib.tar.gz

There are also downloads available containing all public external packages. Examples:

- All public packages for MS Windows 32 bit: http://sourceforge.net/projects/pixellight/files/PixelLight%20v0.9/0.9.11-R1/PixelLight-0.9.11-R1-Externals-Windows_x86_32.tar.gz/download
- All public packages for Linux 32 bit: http://sourceforge.net/projects/pixellight/files/PixelLight%20v0.9/0.9.11-R1/PixelLight-0.9.11-R1-Externals-Linux_x86_32.tar.gz/download

Non-Public Packages Unfortunately, we can't provide some of those third party libraries due to their licensing terms. By convention, projects using such proprietary libraries are only optional, not mandatory in order to be able use PixelLight in the first place. Have a look at the according *Readme.txt* of each external dependency to determine where to obtain those libraries and where to put the resulting files in your source tree. When using the CMake based build process, projects depending on those private packages are excluded from the build by default.

In case you're inside the PixelLight development team, just enter your user name and password within the CMake-GUI and the build system will download those packages as well. Please note that this service is for the development team only. In order to avoid legal issues, we just can't give access to other persons. We're sorry.

The library packages must be unpacked and need to be at the right position for your specific build type, e.g. on Windows and 32 bit, put everything into the directory `<PixelLight root directory>/External/_Windows_x86_32`. Examples:

- `C:/PixelLight/External/_Windows_x86_32/zlib/include`
- `C:/PixelLight/External/_Windows_x86_32/zlib/lib`
- `C:/PixelLight/External/_Windows_x86_32/libpcre/include`
- `C:/PixelLight/External/_Windows_x86_32/libpcre/lib`

The same is true for the non-public packages. Everything goes into one and the same externals directory.

Conclusion The easiest way to install at least the public packages is to use the CMake based build system and make the project *External*. This will download and unpack all public externals for you in the right directory. The non-public externals must still be installed manually.

These packages are for the latest PixelLight version within the Git repository. So, if you're using an official PixelLight release, you might want to use the packed files from <http://sourceforge.net/projects/pixelight/files/> for the used release version in order to avoid compatibility issues.

3. Windows

If you want to build PixelLight on Windows, you have two choices:

- Use the CMake based build system
- Use the provided, manually created and tuned, MSVC solutions to build the engine

If you want to create SDK packages, build the documentation, don't want to care about the external packages or intend to e.g. use MinGW or something like that, stick to the CMake path. This way, you also have the same build process as on other platforms. If you never used CMake before, don't use this as argument to avoid using it - it's really worth a look and easy to use via a Graphical User Interface (GUI). It's just a few clicks.

3.1. CMake

Here's a list of required programs that you need to fully build the SDK on Windows. In case you only want to build, for example the binaries, this is possible as well. At bare minimum, you only need to install CMake and of course your compiler of choice.

3.1.1. Prerequisites for Binaries

In order to compile the binaries you need the prerequisites listed below.

CMake

- Cross platform build tool used to build the SDK
- Download CMake at <http://www.cmake.org>
- Tested with *cmake-2.8.6-win32-x86.exe*

Swiss File Knife *cmake/UsedTools/sfk/sfk.exe* is already within the Git repository and was downloaded from <http://sourceforge.net/projects/swissfileknife/>. It's only mentioned in here for completeness.

- *Swiss File Knife*: file management, search, text processing
- Directly used by the CMake scripts under MS Windows

3. Windows

Diff tools The *cmake/UsedTools/diff* directory is already within the Git repository. It's only mentioned in here for completeness.

- Some diff binaries directly used by the CMake scripts under MS Windows

3.1.2. Prerequisites for Documentation

In order to compile the documentation you need the prerequisites listed below.

Doxygen

- Used to create the code documentations
- Download Doxygen at <http://www.doxygen.org>
- Tested with *doxygen-1.7.6.1-setup.exe*

Graphviz

- Used from Doxygen to create diagrams
- Download Graphviz at <http://www.graphviz.org>
- Tested with *graphviz-2.28.0.msi*
- Ensure that the Graphviz binaries directory is correctly set in the *PATH* and/or *DOT_PATH* environment variables, otherwise Doxygen can't find Graphviz and as a result there will be no graphs within the generated document

MiKTeX

- Used to create the L^AT_EX documentations like the one you're currently reading
- Download MiKTeX at <http://miktex.org/>
- Tested with *setup-2.9.4321.exe*

Microsoft Hypertext Markup Language (HTML) Help Compiler

- Used to create chm documentations from the HTML help files created by Doxygen
- The CMake system automatically searches for an installed Microsoft HTML Help Compiler (*hhc.exe*)
- Download Microsoft HTML Help Workshop at <http://msdn.microsoft.com/en-us/library/ms669985> if it's not yet on your system

3.1.3. Prerequisites for Installable SDK

In order to compile the installable SDK you need the prerequisites listed below.

Nullsoft Scriptable Install System (NSIS)

- Used to create the Windows installer
- Download NSIS at <http://nsis.sourceforge.net/>
- Tested with *nsis-2.46-setup.exe*

3.1.4. Create Solutions and Build

Here's how to compile PixelLight by using the CMake-GUI:

- Start "CMake (cmake-gui)"
- "Where is the source code"-field: e.g. "C:/PixelLight"
- "Where to build the binaries"-field: e.g. "C:/PixelLight/CMakeOutput"
- Press the "Configure"-button
- Choose the generator, for instance "Visual Studio 10" for 32 bit or "Visual Studio 10 Win64" for 64 bit
- Press the "Generate"-button

The CMake part is done, you can close "CMake (cmake-gui)" now. The required external packages described within chapter 2 are automatically downloaded and extracted on-the-fly when building the projects created by the CMake based build system.

Visual Studio Command Prompt Here's how to compile PixelLight by using the Visual Studio command prompt:

- Open the Visual Studio command prompt ("Visual Studio Command Prompt (2010)" or "Visual Studio x64 Win64 Command Prompt (2010)")
- To build type "devenv <CMake output path>/PixelLight.sln /Build Release", example: "devenv C:/PixelLight/CMakeOutput/PixelLight.sln /Build Release"

An alternative is to change into the CMake output directory and typing

```
1 msbuild PixelLight.sln /t:Clean /p:Configuration=Release /p:Platform=
  Win32
2 msbuild PixelLight.sln /p:Configuration=Release /p:Platform=Win32
```

The first line removes everything from a previous build. The second line builds the solution in release mode for the defined platform. In case you want to compile for 64 bit instead of 32 bit as shown in the example above, replace */p:Platform=Win32* by */p:Platform=x64*.

3. Windows

Visual Studio IDE Here's how to compile PixelLight by using the Visual Studio IDE - be warned that the solution is comprehensive and the IDE might groan with pain:

- Open "CMakeOutput/PixelLight.sln" with Microsoft Visual Studio
- You may want to select *Release* as the configuration to build
- To create an installable PixelLight SDK, choose "Pack-SDK", please note that you are free to compile projects individually as well
- To build the documentation, choose "Docs"

3.2. Running from a Local Build

Please note that the following information should only be required during development using PixelLight. Release versions of your application should always contain everything so it can be run out-of-the-box. It should not be required to manipulate the *PATH environment variable* nor the *registry* on the system of end-users.

Once you have built PixelLight, you may want to run e.g. the sample applications. In order for this to work correctly, PixelLight must know where to search for data files and plugins. The simplest solution is to just start *PLInstall.exe*, which is inside your build PixelLight runtime directory (e.g. "C:/PixelLight/Bin/Runtime/x86/"). This tool automatically adds the required *PATH environment variable* and the *registry key*. In case you want to do this manually, continue reading.

PATH Environment Variable MS Windows must be able to find the PixelLight Dynamic-Link Library (DLL)s, meaning you have to add the directory of your PixelLight runtime build (for example "C:/PixelLight/Bin/Runtime/x86/") to the MS Windows PATH environment variable.

- 1. Right click on the "My Computer"-icon on the desktop
- 2. Click on the "Advanced"-tab
- 3. Click the "Environment Variables"-Button
- 4. Under the "User Variables"-header, select the PATH entry and click "Edit"
- 5. Scroll to the end of the text, add a new semi-colon and enter the location of your PixelLight runtime directory (for example "C:/PixelLight/Bin/Runtime/x86/")
- 6. Click "Ok" on all the panels to exit this dialog
- (7. You may have to restart your system, just try whether or not MS Windows is now able to find the PixelLight DLLs, if not, restart your system)

Registry Key You may also want to add a key to the registry, so that the path to the build PixelLight runtime can be found in case you're using the static version of *PLCore*. This key has to be at "HKEY_LOCAL_MACHINE/SOFTWARE/PixelLight/PixelLight-SDK/Runtime" or at "HKEY_LOCAL_MACHINE/SOFTWARE/Wow6432Node/PixelLight/PixelLight-SDK/Runtime" if you are using a 32 bit PixelLight SDK on a 64 bit MS Windows. This "Runtime"-key has e.g. the string value "C:/PixelLight/Bin/Runtime/x86/", the same as directory the PATH environment variable entry mentioned above.

3.3. MSVC Solutions within the Git Repository

If you want to build only a local version of PixelLight but do not intend to create documentations or SDK packages, you can just use the MSVC solution files which can be found directly within the Git repository. Most PixelLight developers also use those solutions in order to work on the engine itself. Those project files are manually created and tuned in order to be more comfortable to use as the project files automatically generated by CMake.

External Package Please note that you need to unpack the required external packages first. The Visual Studio solutions and projects don't do this automatically for you. See chapter 2 for details. Unlike the automatic CMake build process, it's not possible to exclude projects automatically in case an external package is not there. In this case, you will get errors when building everything. In practice, this is no problem because the most important components do only use external prebuild packages which can be published in public. If you have problems with a solution containing projects that can't be build out-of-the box, just stick to the automatic CMake build process.

Build Open the solution *PixellLight.sln* within the root directory of the cloned Git repository with Microsoft Visual Studio. This solution contains all projects of the PixelLight framework. *Build*. Done.

3.4. Using the Build PixelLight Version in MSVC

When using the build PixelLight version in MSVC there are two common approaches. The *dungeon demo project*¹ is used as an concrete example.

Environment Variable Please note that the PixelLight Git repository itself does not require any special PixelLight environment variable to be set in order to build it. Everything is relative to each other. On the other hand, when creating a custom project using PixelLight one might want to use an environment variable pointing to the location

¹git://pixellight.git.sourceforge.net/gitroot/pixellight/pldungeon/

3. Windows

of the cloned PixelLight Git repository. In case you prefer this way we would like to ask you to use the name *PL_ROOT* for the environment variable so that multiple projects are all using the same environment variable for this purpose. When telling MSVC where to find the build library you have to use in this example the following path

```
$(PL_ROOT)/Bin/Lib/x86/
```

for 32 bit. Same thing for the multiple include directories. MSVC will automatically look up your environment variable².

Relative Paths In case you don't want to manipulate your Windows system by adding an environment variable, there's always the option to use relative paths. Imagine you cloned the Git repositories into a directory which now has the following content:

- pixellight
- pldungeon

Meaning that the MSVC project of the dungeon demo is located at *<directory>/pldungeon/Source/Dungeon.vcxproj*. When telling MSVC where to find the build library you have to use in this example the following path

```
../../pixellight/Bin/Lib/x86/
```

for 32 bit. Same thing for the multiple include directories.

²Please note that MSVC needs to be restarted in case there was a change in the registry

4. Linux

To build PixelLight on Linux, use the CMake based cross-platform build system of PixelLight.

4.1. Prerequisites

You need to install¹ certain dependencies on your system that are needed to build PixelLight, including build tools and used development libraries.

The following package list applies to the Linux distribution "Ubuntu 11.10 - Oneiric Ocelot". Other required packages will be downloaded automatically by this listed packages. If you use another distribution, please have a look at their package repository to find out which packages you need. In most cases, the package names will not be that different, but we can't cover all Linux distributions, so we focus just on one.

One Line In case you don't want to read the following paragraphs, just copy'n'paste the following command line:

```
1 apt-get install git cmake cmake-qt-gui build-essential libncurses5-dev  
libxext-dev libxcursor-dev libdbus-1-dev libxxf86vm-dev libxrandr-dev  
libglu1-mesa-dev doxygen graphviz texlive texlive-science
```

Tools You might want to install the following tools:

- *git* (version control)
- *cmake* (build tool)
- *cmake-qt-gui* (comfortable GUI for CMake, start application via command line by typing *cmake-gui*)

To install all packages at once, just use:

```
1 apt-get install git cmake cmake-qt-gui
```

¹This has to be done as root. Use e.g. *sudo* or *su* for this task.

4. Linux

Compile Binaries The following packages are required in order to be able to compile the binaries:

- *build-essential* (GNU C++ compiler and other important build tools)
- *libncurses5-dev* (required by PLCore inside the *PLCore::ConsoleLinux*-class)
- *libxext-dev* (required by e.g. PLFrontendOS or PLGui)
- *libxcursor-dev* (required by e.g. PLFrontendOS or PLGui)
- *libdbus-1-dev* (required by PLInput)
- *libxxf86vm-dev* (required by PLRendererOpenGL)
- *libxrandr-dev* (required by PLRendererOpenGL)
- *libglu1-mesa-dev* (required by PLRendererOpenGL)

To install all required packages at once, just use:

```
1 apt-get install build-essential libncurses5-dev libxext-dev libxcursor-  
dev libdbus-1-dev libxxf86vm-dev libxrandr-dev libglu1-mesa-dev
```

"Could NOT find Qt4 (missing: QT_QMAKE_EXECUTABLE)" In case you get the error message

Could NOT find Qt4 (missing: QT_QMAKE_EXECUTABLE)

, ensure that the files within the *External/Optional/Qt/Linux_x86/* directory are executable (file permissions).

"fatal error: dbus/dbus-arch-deps.h: No such file or directory" In case you get the error message

fatal error: dbus/dbus-arch-deps.h: No such file or directory

and you're using a x86 version of Ubuntu, use the following command in order to copy the missing *dbus-arch-deps.h* file into the correct directory: (This has to be done as root. Use e.g. *sudo* or *su* for this task.)

```
1 cp /usr/lib/i386-linux-gnu/dbus-1.0/include/dbus/dbus-arch-deps.h /usr/  
include/dbus-1.0/dbus
```

Compile Documentation To compile the documentation, the following packages are required:

- *doxygen* (required to compile the automatic code documentation, warning: Also downloads L^AT_EX packages which may take a while)
- *graphviz* (required to create the diagrams of the automatic code documentation)
- *texlive-science* (L^AT_EX package for compiling the documentation)

To install all required packages at once, just use:

```
1 apt-get install doxygen graphviz texlive texlive-science
```

Compile Using System Libraries When using the *maketool* flag `--syslibs`, you also need the following packages:

- *libpcre3-dev* (required by PLCore)
- *libjpeg62-dev* (required by PLGraphics)
- *libpng12-dev* (required by PLGraphics)
- *libfreetype6-dev* (required by PLRendererOpenGL and PLRendererOpenGLES)
- *nvidia-cg-toolkit* (required by PLRendererOpenGLCg - `>= Cg Toolkit 3.1 - February 2012` required) - *nvidia-cg-toolkit* may be out-of-date, install a newer Cg SDK: http://developer.download.nvidia.com/cg/Cg_3.1/Cg-3.1_February2012_x86.deb
- *libopenal-dev* (required by PLSoundOpenAL)
- *libogg-dev* (PLSoundOpenAL)
- *libvorbis-dev* (PLSoundOpenAL)

To install all required packages at once, just use:

```
1 apt-get install libpcre3-dev libjpeg62-dev libpng12-dev libfreetype6-dev
  libopenal-dev libogg-dev libvorbis-dev
```

4.2. External Packages

Just like the Windows build, it is necessary to obtain all the external packages used by the engine and install them in the right place for the build. Have a look at the *External* directory, there you find a readme file for every library that describes what files are needed.

You can find the libraries pre-packed in our files-section on our homepage: <http://pixellight.sourceforge.net/externals/>. Unfortunately, we can't provide some of those libraries due to their licensing terms. Have a look at the according *Readme.txt* to determine where to obtain those libraries and where to put the resulting files into your source tree.

The library packages need to be unpacked and put at the right position for your specific build type, e.g. on Linux and 32 bit, put everything in the directory *External/_Linux_x86_32*.

The CMake based build will try to download the needed packages automatically from our homepage when you build a project that depends on an external. It will download and unpack all public externals for you in the right directory. The non-public externals must still be installed manually. You should also use the maketool flag `--syslibs` to use the libraries installed on your system rather than our own external packages whenever that is possible (however, not all externals can be used that way).

For more information about the external dependencies, please have a look at chapter 2.

4.3. CMake

Here's how to compile PixelLight by using the CMake-GUI:

- Start "CMake (cmake-gui)"
- "Where is the source code"-field: e.g. "~/PixelLight"
- "Where to build the binaries"-field: e.g. "~/PixelLight/CMakeOutput"
- Press the "Configure"-button
- Choose the generator "Unix Makefiles"
- Press the "Generate"-button

The CMake part is done, you can close "CMake (cmake-gui)" now. All required external packages are downloaded automatically, see chapter 2.

- Open a terminal and change into e.g. "~/PixelLight/CMakeOutput"
- Type "make" (example: "make -j 4 -k" will use four Central Processing Unit (CPU) cores and will keep on going when there are errors)

4.4. Maketool Script

In case you don't want to use CMake directly, you can also use the shell script *maketool*. The shell script *maketool* performs the required build steps automatically.

After cloning the Git repository, you first have to call

```
1 chmod +x maketool
```

in order to make the script executable.

To generate the project files just call

```
1 ./maketool setup [--release] [--syslibs]
```

On Linux, it is generally recommended to use the flag `--syslibs`, this will cause the build system to use and depend on the libraries found on your system, rather than using our own externals. Although this may cause troubles when distributing an application, you have to ensure that the end user has the required dependencies installed.

After the project files were generated, the project can be compiled by writing

```
1 ./maketool build [--release]
```

Here's a list of the most important *maketool*-commands:

- `./maketool setup` - Create project files
- `./maketool build` - Compile the project
- `./maketool docs` - Compile the documentation
- `./maketool pack` - Generate an installable package
- `./maketool clean` - Delete the build directories

Here's a list of the most important *maketool*-options:

- `--debug` - Create a debug version (default)
- `--release` - Create a release version
- `--suffix <suffix>` - Add a suffix to all library names
- `--externals` - Repository URL were to download the packages with the external dependencies from (e.g. `http://pixellight.sourceforge.net/externals/`)
- `--username` - User name for access to restricted packages within the repository
- `--password` - User password for access to restricted packages within the repository
- `--arch` - Architecture (e.g. *x86*, *arm*)
- `--bitsize` - Bit size (e.g. *32* or *64*)

4. Linux

- `--syslibs` - Use system libraries
- `--minimal` - Do only compile the most important projects

To avoid setting the parameters `--externals`, `--username` and `--password` over and over again, create a file `pl_config.cfg` in your home directory (Perl-Script, included and just executed) with the following content:

```
1 $pl_external_url = "";
2 $pl_external_user = "";
3 $pl_external_pass = "";
```

4.5. Build

To build PixelLight, either use maketool to do everything automatically:

```
1 ./maketool build [--release]
```

Or change into the build-directory for your configuration (e.g. `build-Debug` or `build-Release`) and run make yourself:

```
1 cd build-Release
2 make
```

The latter option has the advantage that you can also build individual projects rather than the whole SDK. To build projects individually, change into the `build-Debug` or `build-Release` directory and type

```
1 make <project name>
```

(for example: `make PLCore`)

In order to use make options, change into the `build-Debug` or `build-Release` directory and type for example

```
1 make
```

In order to make all projects using four CPU cores to significantly speed up the make process, add the option `-j 4`. You may also add the option `-k` to tell the compiler "Keep on going" if there are errors within the build - useful if you're going for a walk or having some fresh coffee while the build is running. Example:

```
1 make -j 4 -k
```

4.6. Create Documentation and Packages

To create the documentation, build project `Docs`:

```
1 cd build-Release
2 make Docs
```

Or use maketool to do it for you:

```
1 ./maketool docs
```

To create the PixelLight SDK and create a Debian installation file, build project *Pack*:

```
1 cd build-Release
2 make Pack
```

Or use maketool to do it for you:

```
1 ./maketool pack
```

4.7. Running from a Local Build and Installing

Please note that the following information should only be required during development using PixelLight. Release versions of your application should always contain everything so it can be run out-of-the-box. It should not be required to manipulate an *environment variable* on the system of end-users.

Once you have built PixelLight, you may want to run e.g. the sample applications. In order for this to work correctly, PixelLight must know where to search for data files and plugins, which is not always an easy task on Linux systems. You have several options here.

4.7.1. 1. Run from a your Local Source Directory

This means that everything stays in your source directory. All libraries and applications have already been copied into the directory *Bin-Linux* by means of post-build commands.

Now you only have to tell PixelLight where it can find the runtime and data files. This can be done by setting the environment variable *PL_RUNTIME* to the *Runtime* directory.

If you're in the root of the source tree, you can use the script *profile* to do this for you:

```
1 source ./profile
```

or

```
1 . ./profile
```

(sets the environment variable *PL_RUNTIME*)

To inspect the content of *PL_RUNTIME*, call

```
1 echo $PL_RUNTIME
```

4. Linux

To delete the environment variable `PL_RUNTIME`, call

```
1 unset PL_RUNTIME
```

Of course you can also set this variable e.g. inside your profile or bash-scripts so that they are always available. Within your home (`~`) directory, open the hidden `~/.bashrc` file and add:

```
1 export PL_RUNTIME="<PixelLight root path>/Bin-Linux/Runtime/x86"
```

This is the most comfortable way, because you only have to do this once, and then you can use the PixelLight runtime even across multiple terminal instances.

LD_LIBRARY_PATH You may also want to have a look into the `LD_LIBRARY_PATH` environment variable of Linux. It's comparable to the `PATH` environment variable under Windows and enables the Operating System (OS) to find, for example, the `PLCore` shared library if one is requesting it by just using the library name.

4.7.2. 2. Install the PixelLight SDK Locally

Installing means copying the files built by the project into your local Linux system so that the libraries and applications can be found there. The CMake build script therefore provides you with an `install` target that installs everything on your local machine into the `/usr/local` directory.

Change into the build directory, e.g.:

```
1 cd build-Release
```

Install: (This has to be done as root. Use e.g. `sudo` or `su` for this task.)

```
1 make install
```

Update libraries: (This has to be done as root. Use e.g. `sudo` or `su` for this task.)

```
1 ldconfig /usr/local/lib
```

(if this is not done, new dynamic libs may not be found correctly)

If you have installed PixelLight, you will find the runtime in `/usr/local/share/pixel-light/Runtime`, and the samples e.g. in `/usr/local/share/pixel-light/Samples`.

Now you should be able to run the applications built by the PixelLight project, e.g. run one of the samples:

```
1 cd Bin-Linux/Samples/x86
2 ./50RendererTriangle
```

5. Mac OS X

This chapter explains how to compile PixelLight for *Mac OS X 10.6*.

Under Construction The Mac OS X port is currently under construction.

Compiler Version When porting PixelLight to Mac OS X in 2011, there were several issues with the GCC 4.2.1 used on the available system. During the port process, Xcode 4.2 was released introducing C++11 support. At the time of writing this document this new version was not yet successfully installed on the used *Mac OS X 10.6* system. PixelLight also compiles without C++11 features, but you might want to use a more up-to-date compiler like GCC 4.6. Have a look at appendix C for more details about this topic.

Linux On the terminal level, Mac OS X is nearly identical to Linux. So, there's no point in copy'n'paste the Linux part. Instead, this section will refer to the relevant Linux sections.

External Packages The management of external packages is the same on all platforms. Refer to to the Linux section 4.2 or directly to chapter 2. On Mac OS X and 32 bit, put everything in the directory *External/_MacOSX_x86_32*.

CMake CMake as described in 4.3 can be used in the same way under Mac OS X.

Maketool The maketool script as described in 4.4 can be used in the same way under Mac OS X.

Build and Run You may also want to have a look at the Linux sections 4.5 and 4.7.

6. Android

In principle, there's no difference in building PixelLight for Android on a MS Windows or Linux host. So, the following description is e.g. not MS Windows only.

Warning: We noticed several issues when using the NDK under MS Windows (tested with *ndk r6b*)

- The linker eat characters resulting in errors like "CMakeFiles/PLCore.dr/src/File/FileSearchLinux.cpp not found" while the filename was "CMakeFiles/PLCore.dir/src/File/FileSearchLinux.cpp"
- Performance issues, the build process is slow, really slow

In case you never ever did native Android development before, you may have a look at the appendix B, first. Under Linux, in case you're using environment variables as described within the appendix, don't forget that you have to start CMake-GUI out of a terminal. If you don't, the CMake-GUI doesn't know anything about your environment variables.

6.1. Prerequisites

- PixelLight requires at least Android 2.3 (Gingerbread, NDK Application Programming Interface (API) level 9), previous Android versions had no decent native support
- In case you want to use OpenGL ES 2.0, you'll need a physical device because currently the Android Emulator has no support for OpenGL ES 2.0

Install the usual Android development tools. The following versions are those used when writing this documentation, other versions may work as well:

- Android SDK Platform Android 2.3, API 9¹
- Android NDK (tested with *ndk r6b* and *ndk r7*)

Have a look at appendix B for details about setting up the Android development tools.

¹See <http://developer.android.com/guide/appendix/api-levels.html>

6. Android

make (for Windows)

- *Make for Windows*: Make: GNU make utility to maintain groups of programs
- Directly used by the CMake scripts under MS Windows when using the NDK toolchain
- *cmake/UsedTools/make/make.exe* was downloaded from <http://gnuwin32.sourceforge.net/packages/make.htm>

This tool can't be set within a CMake file automatically, there are several options:

- Add `<PixelLight root path>/cmake/UsedTools/make` to the MS Windows *PATH* environment variable **recommended**
- Use a MinGW installer from e.g. <http://www.tdragon.net/recentgcc/> which can set the *PATH* environment variable **overkill because only the 171 KiB make is required**
- Use CMake from inside a command prompt by typing for example (*DCMAKE_TOOLCHAIN_FILE* is only required when using a toolchain)
not really comfortable when working with it on a regular basis

```
1 cmake.exe -G"Unix Makefiles" -DCMAKE\ _MAKE\ _PROGRAM="<PixelLight root path>/cmake/UsedTools/make/make.exe" -DCMAKE\ _TOOLCHAIN\ _FILE="<PixelLight root path>/cmake/Toolchains7Toolchain-ndk.cmake"
```

6.2. Create Makefiles and Build

Here's how to compile PixelLight by using the CMake-GUI:

- Ensure "make" (GNU make utility to maintain groups of programs) can be found by CMake (add for instance "`<PixelLight root path>/cmake/UsedTools/make`" to the MS Windows *PATH* environment variable)
- Start "CMake (cmake-gui)"
- "Where is the source code"-field: e.g. "C:/PixelLight"
- "Where to build the binaries"-field: e.g. "C:/PixelLight/CMakeOutput"
- In case you want to use the Android Emulator instead of a physical Android device: Click on "Add Entry", add a variable named *ARM_TARGET* of the type *STRING* and assign the value *armeabi* to it (default is *armeabi-v7a* for a physical Android device)
- Press the "Configure"-button

- Choose the generator "Unix Makefiles" and select the radio box "Specify toolchain file for cross-compiling"
- Press the "Next"-button
- "Specify the Toolchain file": e.g. "C:/PixelLight/cmake/Toolchains/Toolchain-ndk.cmake"
- Press the "Generate"-button

The CMake part is done, you can close "CMake (cmake-gui)" now. All required external packages are downloaded automatically, see chapter 2.

- Open a command prompt and change into e.g. "C:/PixelLight/CMakeOutput" (MS Windows: by typing "cd /D C:/PixelLight/CMakeOutput" -> "/D" is only required when changing into another partition)
- Type "make" (example: "make -j 4 -k" will use four CPU cores and will keep on going when there are errors)
- (You now should have the ready to be used Android shared library files)

6.3. Using the Android Port

The content within this section was tested using "Ubuntu 11.10 - Oneiric Ocelot". Just mentioned the used version to be on the safe side, other, newer versions may probably work as well. Usually, the build PixelLight version for Android can't be used in the same way as a version build for the host system like Linux. That's the reason this section, showing how to get a PixelLight sample onto your Android device, exists.

Java Development Kit (JDK) and Java Runtime Environment (JRE) The following is required for building Android Android Application Package file (APK)-files.

- Install JDK ("sudo apt-get install default-jdk") and JRE ("sudo apt-get install default-jre")
- Install "ant" ("sudo apt-get install ant"), required to create the APK files

To install all packages at once, just use:

```
1 apt-get install default-jdk default-jre ant
```

6. Android

Files Overview Content of the *project*-directory (name it e.g. *PixelLightAndroid*):

- FindPixelLight.cmake
- CMakeLists.txt
- *src*-directory

Content of the *src*-directory:

- AndroidMain.cpp
- CMakeLists.txt

FindPixelLight.cmake Within the *<PixelLight root path>/Tools* directory is a CMake file named *FindPixelLight.cmake*. This section is using this script file and assumes that it just has been copied into your project directory. It's kind of a link to PixelLight. Have a look at the appendix A for more information about this script.

CMakeLists.txt within Project Directory Here's an example for a simple *CMakeLists.txt* file within your project directory:

```
1 cmake_minimum_required(VERSION 2.8.3)
2
3 # Add the directory this "CMakeLists.txt" file is in as CMake module
4   path in order to make it possible to find "FindPixelLight.cmake"
5   set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} ${CMAKE_CURRENT_LIST_DIR})
6
7 # Define the project, use the current directory name as project name
8   get_filename_component(CURRENT_TARGET_NAME ${CMAKE_CURRENT_LIST_DIR}
9     NAME_WE)
10  project(${CURRENT_TARGET_NAME})
11
12 # Add the source code directory
13 add_subdirectory(src)
```

AndroidMain.cpp Here's an example for a simple *AndroidMain.cpp* file within your project source code directory:

```
1 #include <PLFrontendOS/FrontendAndroid.h>
2
3 /**
4  * @brief
5  *   Program entry point
6  *
7  * @remarks
```

```

8 *   This is the main entry point of a native application that is using
9 *   android_native_app_glue. It runs in its own thread, with its own
10 *   event loop for receiving input events and doing other things.
11 */
12 void android_main(struct android_app* state)
13 {
14     // Make sure glue isn't stripped
15     app_dummy();
16
17     // Create a frontend instance on the C runtime stack
18     PLFrontendOS::FrontendAndroid cFrontendAndroid(*state);
19
20     // Finish the given activity
21     ANativeActivity_finish(state->activity);
22 }

```

CMakeLists.txt within Project Source Code Directory Here's an example for a simple *CMakeLists.txt* file within your project source code directory:

```

1 cmake_minimum_required(VERSION 2.8.3)
2
3 # Find packages
4 find_host_package(PixelLight) # "find_package"-variant provided by "
   android.toolchain.cmake", required for PixelLight CMake variables
   like "PL_PLCORE_INCLUDE_DIR"
5
6 # Includes
7 include_directories(${CMAKE_CURRENT_SOURCE_DIR})           # The current
   source directory
8 include_directories(${ANDROID_NDK}/sources/android/native_app_glue) #
   For the "android_native_app_glue.h"-header include
9 include_directories(${PL_PLCORE_INCLUDE_DIR})             # PLCore headers
10 include_directories(${PL_PLFRONTENDOS_INCLUDE_DIR})      #
   PLFrontendOS headers
11
12 # Source codes
13 set(CURRENT_SRC
14     ${ANDROID_NDK}/sources/android/native_app_glue/android_native_app_glue
15     .C
16     AndroidMain.cpp
17 )
18
19 # Shared libraries
20 set(CURRENT_SHARED_LIBRARIES

```

6. Android

```
20 # Base
21 ${PL_PLCORE_LIBRARY}
22 ${PL_PLMATH_LIBRARY}
23 ${PL_PLGRAPHICS_LIBRARY}
24 ${PL_PLINPUT_LIBRARY}
25 ${PL_PLRENDERER_LIBRARY}
26 ${PL_PLMESH_LIBRARY}
27 ${PL_PLSCENE_LIBRARY}
28 ${PL_PLPHYSICS_LIBRARY}
29 ${PL_PLENGINE_LIBRARY}
30 # Plugins
31 ${PL_PLFRONTENDOS_LIBRARY}      # The PixelLight Android frontend
32 ${PL_PLRENDEREROPENGLS2_LIBRARY}
33 ${PL_PLCOMPOSITING_LIBRARY}
34 # Application
35 "${PL_SAMPLES_BIN_DIR}/lib60Scene.so"
36 )
37
38 # Assets
39 set(CURRENT_ASSETS
40     "${PL_RUNTIME_DATA_DIR}/Standard.zip"
41     "${PL_SAMPLES_DATA_DIR}/*.*"      # Copy all sample data
42 )
43
44 # Build
45 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x -ffor-scope -fno-rtti
46     -fno-exceptions -pipe -ffunction-sections -fdata-sections -ffast-
47     math -Wnon-virtual-dtor -Wreorder -Wsign-promo -fvisibility=hidden -
48     fvisibility-inlines-hidden -Wstrict-null-sentinel -O3 -funroll-all-
49     loops -fpeel-loops -ftree-vectorize")
50
51 set(LINKER_FLAGS "${LINKER_FLAGS} -Wl,--as-needed -Wl,--gc-sections -Wl
52     ,--no-undefined -Wl,--strip-all -Wl,-rpath-link=${ANDROID_NDK_SYSROOT
53     }/usr/lib/ -L${ANDROID_NDK_SYSROOT}/usr/lib/")
54
55 add_library(${CURRENT_TARGET_NAME} SHARED ${CURRENT_SRC})
56 target_link_libraries(${CURRENT_TARGET_NAME} log android ${
57     CURRENT_SHARED_LIBRARIES})
58
59 set_target_properties(${CURRENT_TARGET_NAME} PROPERTIES
60     COMPILE_DEFINITIONS "__STDC_INT64__;LINUX;ANDROID") # PLCore needs
61     the preprocessor definitions "LINUX" and "ANDROID"
62
63 # Post build
64
65 # Copy the build shared library as well
66 set(CURRENT_SHARED_LIBRARIES
```

```

55  ${CURRENT_SHARED_LIBRARIES}
56  "${CMAKE_BINARY_DIR}/libs/${ARM_TARGET}/lib${CURRENT_TARGET_NAME}.so"
57  )
58
59  # Create Android apk file (macro from "<PixelLight>/cmake/Android/Apk.
    cmake")
60  android_create_apk("${CURRENT_TARGET_NAME}" "${CMAKE_BINARY_DIR}/apk" "$
    {CURRENT_SHARED_LIBRARIES}" "${CURRENT_ASSETS}" "Data")

```

Build The build process for this project is the same as described within section 6.2, except that the standalone toolchain *<PixelLight root path>/cmake/Android/android.toolchain.cmake* instead of *<PixelLight root path>/cmake/Toolchains/Toolchain-ndk.cmake* is used. By using the provided macro *android_create_apk()*, the project is automatically packed into an APK-file, installed on our device and started at once.

At first, this may look a lot - but you usually need to do write this only once per project. During the development you just have to type *make* in order to let your project run on the Android device, without doing everything manually as described within appendix B over and over again.

7. Contact

contact@pixellight.org

<http://www.pixellight.org>

A. FindPixelLight.cmake

Within the `<PixelLight root path>/Tools` directory is a CMake file named `FindPixelLight.cmake`. You can use this file in your CMake build scripts in order to find and use PixelLight. You only have to tell this PixelLight find script where it can find the PixelLight runtime.

Windows Registry Key You have to add a key to the registry, so that the path to the build PixelLight runtime can be found. This key has to be at "HKEY_LOCAL_MACHINE/SOFTWARE/PixelLight/PixelLight-SDK/Runtime" (or at "HKEY_LOCAL_MACHINE/SOFTWARE/Wow6432Node/PixelLight/PixelLight-SDK/Runtime" if you are using a 32 bit PixelLight SDK on a 64 bit MS Windows). This "Runtime"-key has e.g. the string value "C:/PixelLight/Bin/Runtime/x86/" (same as the PATH environment variable entry).

Linux On Linux, this can be done by setting the environment variable `PL_RUNTIME` to the `Runtime` directory.

If you're in the root of the source tree, you can use the script `profile` to do this for you:

```
1 source ./profile
```

or

```
1 . ./profile
```

(sets the environment variable `PL_RUNTIME`)

To inspect the content of `PL_RUNTIME`, call

```
1 echo $PL_RUNTIME
```

To delete the environment variable `PL_RUNTIME`, call

```
1 unset PL_RUNTIME
```

Of course you can also set this variable e.g. inside your profile or bash-scripts so that they are always available. Within your home (`~`) directory, open the hidden `~/.bashrc` file and add:

```
1 export PL_RUNTIME="<PixelLight root path>/Bin-Linux-ndk/Runtime/armeabi-v7a"
```

This is the most comfortable way, because you only have to do this once, and then you can use the PixelLight runtime even across multiple terminal instances.

B. Android Native Build Tutorial

During the implementation of the Android port, a lot of notices were written down. Although there's couple of information about how to do native Android development available, it's often out-of-date, confusing or just not working. A lot of trivial looking information had also be figured out in many frustrating hours of work. So, because there were already a lot of notices written down and the possibility is high that someone using PixelLight for native Android development is requiring the same or quite similar information, those notices were put into this appendix in a revised form.

The good news is, when you build PixelLight for Android, you don't really need to know everything written down in here. The process is heavily automated by using CMake scripts. In case you just find this appendix unnecessary or you hate spoilers and want to figure out all by yourself, ignore this appendix. For all others, this information in here is hopefully useful for you for the first steps in native Android development without the automated CMake build system PixelLight is using.

Please note that this tutorial is using Linux because using MS Windows for native Android development is really painful.

Some Hints on how to Read this Tutorial The tutorial is written in brief sentences to keep it compact. But this doesn't mean there's no additional information, so, the most important information is marked by using **bold text**. Additional, nice to know information is marked by using \rightarrow . If you want to use the fast path, just focus on the **bold texts** like terminal commands.

B.1. Prerequisites

- Used OS: "Ubuntu 11.10 - Oneiric Ocelot" (just mentioned the used version to be on the safe side, other, newer versions may probably work as well)
- Install JDK ("**sudo apt-get install default-jdk**") and JRE ("**sudo apt-get install default-jre**")
- Install "ant" ("**sudo apt-get install ant**"), required to create the APK files

To install all packages at once, just use:

```
1 apt-get install default-jdk default-jre ant
```

Install Android NDK

- **Download** from <http://developer.android.com/sdk/ndk/index.html> → "android-ndk-r6b-linux-x86.tar.bz2"
- **Extract** to for example "~/android-ndk-r6b" ("~/ " is your home directory)

Android NDK (*ndk r6b*) - MS Windows

- Extract it and set the MS Windows PATH environment variable *ANDROID_NDK* to the NDK root directory
- Set the MS Windows *PATH* environment variable *ANDROID_NDK_TOOLCHAIN_ROOT* to the NDK toolchain root directory (e.g. "C:/android-ndk-r6b/toolchains/arm-linux-androideabi-4.4.3/prebuilt/windows/arm-linux-androideabi")
- (Those variables can also be added/set within the CMake-GUI)

Install Android SDK

- **Download** from <http://developer.android.com/sdk/index.html> → "android-sdk_r12-linux_x86.tgz"
- **Extract** to for example "~/android-sdk-linux_x86" ("~/ " is your home directory)
- Tested with Android SDK Tools, revision 12
- Tested with Android SDK Platform-tools, revision 6
- Tested with SDK Platform Android 2.3, API 9¹

Optional but Highly Recommended for a Decent Workflow This example assumes that the data has been extracted directly within the home (~) directory. Open hidden "~/.bashrc"-file and add:

```
1 # Important Android SDK and NDK paths
2 export ANDROID_SDK=~/.android-sdk-linux_x86
3 export ANDROID_NDK=~/.android-ndk-r6b
4 export PATH=${PATH}:${ANDROID_SDK}/tools:${ANDROID_SDK}/platform-tools
   :~/${ANDROID_NDK}
```

- Open a new terminal so the changes from the step above have an effect

¹See <http://developer.android.com/guide/appendix/api-levels.html>

Android SDK and AVD Manager Type "**android**" to open the "Android SDK and AVD Manager"-GUI:

- "Available packages" → disable the "Display updates only"-checkbox (you may need to enlarge the window to see this checkbox) → install at least the following:
- "Android SDK Tools, revision 12"
- "Android SDK Platform-tools, revision 6"
- "SDK Platform Android 2.3.1, API 9, revision 2" (it's marked "Obsolete", but within the NDK r6b there's only up to API level 9 available and we don't want to mix)

B.2. Android Emulator and Device

- Android emulator² → Type "**android**" to open the "Android SDK and AVD Manager"-GUI and do the emulator configuration in here
- Android device³ → Configuration: Just connect your device to your computer and it should work at once... if you have enabled "USB-Debugging" on your device (launch "Settings", tap "Applications", tap "Development" and enable the checkbox for USB debugging)

Not Required but Nice to Know Check available devices: Type "**adb devices**" → You should see at least one entry when your device is connected. The result may look like the following:

```
1 "List of devices attached
2 028842074300d157 device
3 emulator-5554 device"
```

- The output for each instance is formatted like this: → "[serialNumber] [state]"
- In this case "028842074300d157 device" is my connected smartphone
- In this case "emulator-5554 device" is the emulator I created and started

In case you don't see your device, try:

```
1 adb kill-server
2 adb start-server
3 adb devices
```

²General information: <http://developer.android.com/guide/developing/devices/emulator.html>

³General information: <http://developer.android.com/guide/developing/device.html>

B.3. NDK Build System

The NDK build system is not used by PixelLight, but for the first steps it's nice to know how to use it.

Time for a first experiment. Sadly, as on September 2011, it appears that some of the information at <http://developer.android.com/sdk/ndk/overview.html> is out-of-date and something like "I just try out native-activity to get the idea" isn't working as "just" as thought. So, here's an updated version with some additional handy information for MS Windows users like myself: (yes, there are people out there don't knowing that "~/" is your home directory, so I just mention such stuff)

build.xml Run the following command to generate a *build.xml* file:

- **Change** to the "~/android-ndk-r6b/samples/native-activity" **directory**
- → Don't follow the official sample instructions: "android update project -p . -s"
→ "Error: The project either has no target set or the target is invalid. Please provide a --target to the 'android update' command."
- → Type "android --help" to see available options
- → Type "android list targets" so see available targets (= API levels)
- Type "android update project -t android-9 -p ." ("." means "the current directory")
- → You now have a "build.xml"-file in the same directory as the "AndroidManifest.xml"-file, this is required for the APK creation step

Compile Compile the native code using the *ndk-build* command.

- "**ndk-build**"
- → You now have "~/android-ndk-r6b/samples/native-activity/libs/armeabi/libnative-activity.so"

APK Create APK file

- Type "**ant debug**" ("debug" is only for developing and testing using easy automatically signing, see <http://developer.android.com/guide/publishing/app-signing.html>)
- → You now have "~/android-ndk-r6b/samples/native-activity/bin" with some files in it
- Start the emulator, or connect your device (ensure that it has Android 2.3 > on it, else the app will just crash)

- Type **"adb install -r bin/NativeActivity-debug.apk"** ("-r"-option to avoid "Failure [INSTALL_FAILED_ALREADY_EXISTS]"-error when the APK is already installed, default destination is "/data/local/tmp/NativeActivity-debug.apk")
- → The app is now available and ready to be started within your emulator or on your device
- → To uninstall the app, type **"adb uninstall com.example.native_activity"**

Release APK Create release APK file⁴

- Type **"ant release"** (you now have "bin/NativeActivity-unsigned.apk")
- → Don't try **"adb install -r bin/NativeActivity-unsigned.apk"**, this will just result in "Failure [INSTALL_PARSE_FAILED_NO_CERTIFICATES]"
- → The JDK tools Jarsigner and Keytool will be used (ensure they're available, if you installed JRE they are usually available)
- → You need a private key to sign your APK file with, if you don't have any: Type e.g. **"keytool -genkey -v -keystore my-release-key.keystore -alias myalias -keyalg RSA -keysize 2048 -validity 10000"** (you now have a "my-release-key.keystore"-file)
- For signing the APK file, type: **"jarsigner -verbose -keystore my-release-key.keystore bin/NativeActivity-unsigned.apk myalias"** (no new file, it's still "bin/NativeActivity-unsigned.apk" and you may rename it later, but for now we don't touch the name)
- → Type **"jarsigner -verify bin/NativeActivity-unsigned.apk"** to verify that everything went fine and **"jarsigner -verify -verbose -certs bin/NativeActivity-unsigned.apk"** to get additional information
- Finally, align your APK file by typing **"zipalign -v 4 bin/NativeActivity-unsigned.apk bin/NativeActivity.apk"** (you now have the ready to be released file "bin/NativeActivity.apk")
- To install this file right now, type **"adb install -r bin/NativeActivity.apk"**
- → If you receive a "Failure [INSTALL_PARSE_FAILED_INCONSISTENT_CERTIFICATES]" you need to remove the previously installed application by typing **"adb uninstall com.example.native_activity"**
- → To uninstall the app, type **"adb uninstall com.example.native_activity"**

⁴See <http://developer.android.com/guide/publishing/app-signing.html>fordetailedinformation

B.4. CMake Build System

We want to use the universal CMake build system, not the special NDK build system.

- **Download and extract** CMake toolchain "android-cmake" from <http://code.google.com/p/android-cmake/> (we're using it's "android.toolchain.cmake"-file) and extract it to e.g. "~/android-cmake"

Optional but Highly recommended for a Decent Workflow Open hidden "~/.bashrc"-file and add:

```
1 # CMake toolchain "android-cmake" from \url{http://code.google.com/p/
   android-cmake/}
2 export ANDROID_CMAKE=~/.android-cmake
3 export ANDTOOLCHAIN=$ANDROID_CMAKE/toolchain/android.toolchain.cmake
4 alias android-cmake='cmake -DCMAKE_TOOLCHAIN_FILE=$ANDTOOLCHAIN '
```

- Open a new terminal so the changes from the step above have an effect

B.4.1. First Experiment

Time for a first experiment by using "hello-gl2" of *android-cmake*.

- **Change** into the "hello-gl2"-**directory** of *android-cmake*

Build Type:

- "mkdir build"
- "cd build"
- "android-cmake -DARM_TARGET=armeabi .."
- → The Android SDK emulator supports only "armeabi", but "android-cmake" has "armeabi-v7a" as default. If you don't change this, the application will just crash when you try to start it within the emulator. For more complex 3D applications, "armeabi-v7a" is highly recommended due to hardware floating point support. So, for more advanced stuff you really need a real device instead of the emulator.
- "make"
- → You now have "~/hello-gl2/libs/armeabi/libgl2jni.so"

Keystore

- **Copy "my-release-key.keystore"** from your NDK build system experiment into the "hello-gl2"-directory, or keep it within e.g. your home directory and update the entries below

APK Back to the "hello-gl2"-directory and type:

- `"sh project_create.sh"` (you may need to open this file first and replace `"android update project --name HelloGL2 --path ."` through `"android update project -t android-8 --name HelloGL2 --path ."`)
- `"ant release"` (create the APK file)
- `"jarsigner -verbose -keystore my-release-key.keystore bin/HelloGL2-unsigned.apk myalias"` (sign the APK file)
- `"zipalign -v 4 bin/HelloGL2-unsigned.apk bin/HelloGL2.apk"` (align the APK file)
- `"adb uninstall com.android.gl2jni"` (in order to ensure that we don't get problems when doing the following install step)
- `"adb install -r bin/HelloGL2.apk"` (install the APK file on the device)
- `"adb shell am start -n com.android.gl2jni/.GL2JNIActivity"` (start the installed APK file automatically)

B.4.2. Native Activity Experiment

Another experiment, "native-activity" for a native activity of the NDK.

- Change into the "native-activity"-directory of the NDK

CMakeLists.txt within the "native-activity"-directory Within the "native-activity"-directory, create a text file named *CMakeLists.txt* with the following content:

```
1 cmake_minimum_required(VERSION 2.8)
2 project(native-activity)
3 add_subdirectory(jni)
```

CMakeLists.txt within the "native-activity/jni"-directory Within the "native-activity/jni"-directory, create a text file named *CMakeLists.txt* with the following content:

```
1 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fPIC")
2 include_directories("${ANDROID_NDK}/sources/android/native_app_glue")
3 include_directories(${CMAKE_CURRENT_SOURCE_DIR})
4 set(LIBRARY_DEPS log android EGL GLESv1_CM)
5 set(MY_SRCS
6     ${ANDROID_NDK}/sources/android/native_app_glue/
7     android_native_app_glue.c
```

B. Android Native Build Tutorial

```
7     main.c
8     )
9 add_library(native-activity SHARED ${MY_SRCS})
10 target_link_libraries(native-activity ${LIBRARY_DEPS})
```

Build We're going to use the native activity which was introduced in Android 2.3 (Gingerbread, "android-9" NDK API level). Within the "native-activity"-directory, type:

- "mkdir build"
- "cd build"
- "android-cmake -DARM_TARGET=armeabi -DANDROID_API_LEVEL=9 .."
- "make"

Nice to Know In case you want to write "android-cmake -DARM_TARGET=armeabi .." instead of "android-cmake -DARM_TARGET=armeabi -DANDROID_API_LEVEL=9 .." → Open hidden "~/.bashrc"-file and add:

```
1 export ANDROID_API_LEVEL=9
```

Keystore

- **Copy** "my-release-key.keystore" from your NDK build system experiment into the "native-activity"-directory, or keep it within e.g. your home directory and update the entries below

APK Back to the "native-activity"-directory and type:

- "android update project -t android-9 --name native-activity --path .."
- "ant release"
- "jarsigner -verbose -keystore my-release-key.keystore bin/native-activity-unsigned.apk myalias"
- "zipalign -v 4 bin/native-activity-unsigned.apk bin/native-activity.apk"
- "adb uninstall com.example.native_activity"
- "adb install -r bin/native-activity.apk"

All in one single command line row:

```

1 mkdir build;cd build;android-cmake -DARM_TARGET=armeabi ../.;make;cd ../;
  android update project -t android-9 --name native-activity --path .;
  ant release;jarsigner -verbose -keystore my-release-key.keystore bin/
  native-activity-unsigned.apk myalias;rm bin/native-activity.apk;
  zipalign -v 4 bin/native-activity-unsigned.apk bin/native-activity.
  apk;adb uninstall com.example.native_activity;adb install -r bin/
  native-activity.apk

```

B.5. Glossary

Glossary (only terms I wasn't really familiar with)

| Short | Long | Information |
|--------|------------------------------|---|
| ndk | Native Development Kit | http://developer.android.com/sdk/ndk/index.html |
| adb | Android Debug Bridge | http://developer.android.com/guide/developing/tools/adb.html !important! |
| avd | Android Virtual Device | http://developer.android.com/guide/developing/devices/emulator.html |
| aapt | Android Asset Packaging Tool | |
| JNI | Java Native Interface | |
| logcat | | Android system central log buffer |

B.6. Command Glossary

Command glossary (only terms I wasn't really familiar with)

B. Android Native Build Tutorial

| Command | Result |
|---------------------------|---|
| android | Start "Android SDK and AVD Manager" (e.g. to start the emulator) |
| adb devices | See all available emulators/devices |
| adb logcat | Show Android log (called "logcat"), this is realtime, so the command prompt will block and show new upcoming log entries at once, more information: http://developer.android.com/guide/developing/tools/adb.html#logcat |
| adb logcat -s <tag> | Show only messages with the given tag within the Android log, example: "adb logcat -s PixelLight" |
| adb push <local> <remote> | Copy <local> (file or directory recursively) to the emulator/device to destination <remote>, example: "adb push foo.txt /sdcard/foo.txt" |
| adb pull <remote> <local> | Copy <remote> (file or directory recursively) from the emulator/device to destination <local>, example: "adb pull /sdcard/foo.txt foo.txt" |
| adb install <APK file> | Install APK file on emulator/device (but doesn't start it automatically) |
| adb -d install <APK file> | Install APK file on device (but doesn't start it automatically) |
| adb -e install <APK file> | Install APK file on emulator (but doesn't start it automatically) |
| adb uninstall <package> | Uninstall an APK, example: "adb uninstall com.example.native_activity" |
| adb shell am start <app> | Start an app, example: "adb shell am start -n com.android.gl2jni/.GL2JNIActivity" |

NDK build system related:

| Command | Result |
|--|--|
| android update project -t android-9 -p . | Create/update "build.xml" (required for Ant) |
| ndk-build | Compile native code |
| ant debug | Create debug APK |
| ant release | Create release APK |

B.7. Possible issues

"Android can't load in my shared library"

- Is the build target correct?
- The Android Emulator is only able to deal with "armeabi", not e.g. "armeabi-v7a"

- → When using CMake-GUI, add the string entry "ARM_TARGET"="armeabi"

How to Start a Native Activity Automatically? I wasn't able to figure this one out. When having a Java file as main program entry point, can can e.g. just type "adb shell am start -n com.android.gl2jni.GL2JNIActivity" and the application starts automatically. Well, a pure native activity without a single Java source file... I have no glue...

C. Mac OS X GCC

Within a terminal, type

```
1 gcc -v
```

in order to see the currently used GCC version. In case you see e.g.

```
1 gcc version 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
  2335.15.00)
```

you're using an out-of-date GCC version not capable of any C++11 features.

One way to get a new GCC version like GCC 4.6 is to just compile it yourself. Please note that explaining how to compile GCC for Mac OS X is out of the scope of this document. There are many good instructions like <http://solarianprogrammer.com/2011/05/28/compiling-gcc-4-6-0-on-mac-osx/> explaining in detail how to compile GCC for Mac OS X. The process isn't that complicated, but time consuming. Do not forget to include Objective-C++ support when compiling your GCC. When compiling a new GCC, the GCC of the system will not be touched. In order to use the build GCC 4.6 type

```
1 export CXX=$HOME/my_gcc/bin/g++-4.6.1
2 export CC=$HOME/my_gcc/bin/gcc-4.6.1
3 export CPP=$HOME/my_gcc/bin/cpp-4.6.1
```

in case you put your build GCC into the *\$HOME/my_gcc/* directory.

In case you don't want to do this over and over again, put this overwritten environment variables into your profile file. To put this into the profile file, type

```
1 pico $HOME/.profile
```

and copy'n'paste the export lines above.

When typing

```
1 gcc -v
```

you will still get *4.2.1* as GCC because you just set the GCC environment variables CMake is using in order to figure out which compiler to use. Type

```
1 $CXX -v
```

and the output will be

```
1 gcc-Version 4.6.1 (GCC)
```

, you're now ready to continue.

Abbreviations

DLL Dynamic-Link Library, under Linux it's called Shared Object (SO)

SO Shared Object, under MS Windows it's called DLL

API Application Programming Interface

SDK Software Development Kit, also known as *devkit*

OS Operating System

CPU Central Processing Unit

GUI Graphical User Interface

RTTI Run-Time Type Information, or Run-Time Type Identification

NSIS Nullsoft Scriptable Install System

MSVC Microsoft Visual C++, also known as *Visual C++* or *VC++*

GCC GNU Compiler Collection, formerly the "GNU C Compiler"

HTML Hypertext Markup Language

URL Uniform Resource Locator

IDE Integrated Development Environment

MS Microsoft

RSS Really Simple Syndication

JDK Java Development Kit

JRE Java Runtime Environment

NDK Android Native Development Kit

APK Android Application Package file