Contents

```
#include <cstdio>

class Foo {
    public:
    static int bla();
};
```

Figure 1: `foo.h`

```
#include "foo.h"

int Foo::bla() {
    return 4;
}
```

Figure 2: `foo.cpp`

1 Simple examples

1.1 A very simple example

We begin with a very simple example consisting of three files shown in figure 1, 2 and 3. We like to create a library called `foo`. This library offers the class `Foo`, defined in `foo.h` and implemented in `foo.cpp`. Then we want to create an executable `test`. The executable code is implemented in `test.cpp`.

To build this example with CMake, we create a file CMakeLists.txt in the source directory. The content of this file looks like this:

```
CMAKE_MINIMUM_REQUIRED (VERSION 2.6)
PROJECT (Test)

ADD_LIBRARY (foo SHARED
    foo.h
    foo.cpp
)
```

```
#include <cstdio>
#include "foo.h"

int main(int argc, char** argv)
{
    printf ("foo is %i\n", Foo::bla());
    return 0;
}
```

Figure 3: `test.cpp`

```
08
09 ADD_EXECUTABLE (test
10     test.cpp
11     foo.h
12 )
13
14 TARGET_LINK_LIBRARIES (test
15     foo
16 )
```

CMake commands are case insensitive. We will write them with big letters by default. The first two lines set the minimal version number of CMake required to build the project and the project name. The project name is used e.g. as the name for the Visual Studio solution.

Lines 4-7 define a library target. The first parameter sets the target name, the second one tells CMake that we like to build a shared object file[1]. If you prefer a static library, use `STATIC` instead.[2] The following parameters define the files included in the library. CMake will automatically keep track of header dependencies. Lines 9-12 define an executable. The parameters are just the same as in the library definition.

The command in the lines 14-16 does two things. First it tells CMake to link the target given in the first parameter against the following libraries, no matter if the target is a library or an executable. Second it allows CMake to automatically build a dependency structure. This way it is guaranteed that the build structure builds the defined targets in the right order.

1.2 Building the example

You should have the four files in one directory. Create another directory somewhere on your disk. You should NOT run CMake from your source directory.[3]

Unix makefiles Start a console and change to your newly created build directory. Now call `cmake pathto`, where `pathto` is the source directory. CMake should now build a makefile.

Visual Studio CMake comes with a program called cmake-gui. Specify your source and build directories and press Configure. CMake should now show you a list of supported generators. Choose your favorite Visual Studio version and press OK. Now press Configure again and then generate. CMake will now create the Visual Studio solution.

[1] `libfoo.so` on Unix or `foo.lib` and `foo.dll` on Windows

[2] If you don't specify if you want to use static or shared libraries, CMake will use the variable `BUILD_SHARED_LIBS`. You will see more about variables later.

[3] This is actually possible, but has its troubles. It is easier to keep your source tree untouched.

1.3 Subdirectories

Now we want to move the library and the executable in different subdirectories called `foolib` and `execute`.[4] Every subdirectory gets its own `CMakeLists.txt`, where most of the old file goes to. The files look like this:

```
Toplevel:
cmake_minimum_required(VERSION 2.6)
PROJECT (Test)

ADD_SUBDIRECTORY (foolib)
ADD_SUBDIRECTORY (execute)

foolib:
ADD_LIBRARY (foo SHARED
    foo.cpp
    foo.h
)

execute:
INCLUDE_DIRECTORIES (${CMAKE_SOURCE_DIR}/foolib)

ADD_EXECUTABLE (test
    test.cpp
)

TARGET_LINK_LIBRARIES (test
    foo
)
```

The `ADD_SUBDIRECTORY` command tells CMake to look for another `CMakeLists.txt` in the given subdirectory. The executable target needs an additional path to the include file `foo.h`. The library itself is not changed.

1.4 Variables and defines

In some cases it may happen, that one wants to make a library dependency optional. CMake should leave it to the user if he wants to build the foo library. If he decides to not build it, the executable should be built without using the class `Foo`. Please find the the `test.cpp` in figure 1.4. We change the `CMakeLists.txt` both on the top level and in the executable's directory. The foo library is not touched.

```
Toplevel:
cmake_minimum_required(VERSION 2.6)
PROJECT (Test)

SET (EXAMPLE_BUILD_FOO ON CACHE BOOL "Build foo library")
```

[4]You need to change the include in the file `test.cpp` to `#include <foo.h>`

```
#include <cstdio>

#if USEFOO
#include <foo.h>
#endif

int main(int argc, char** argv)
{
#if USEFOO
    printf ("foo␣is␣%i\n", Foo::bla());
#else
    printf ("foo␣was␣not␣compiled\\n");
#endif
    return 0;
}
```

Figure 4: `test.cpp with optional foo support`

```

IF (EXAMPLE_BUILD_FOO)
    ADD_SUBDIRECTORY (foolib)
    ADD_DEFINITIONS (-DUSEFOO)
ENDIF (EXAMPLE_BUILD_FOO)
ADD_SUBDIRECTORY (execute)

INCLUDE_DIRECTORIES (${CMAKE_SOURCE_DIR}/foolib)
```

```
execute:
ADD_EXECUTABLE (test
    test.cpp
)

IF (EXAMPLE_BUILD_FOO)
    TARGET_LINK_LIBRARIES (test
        foo
    )
ENDIF (EXAMPLE_BUILD_FOO)
```

The `SET` command creates or changes a variable. In our case, we create a variable called `EXAMPLE_BUILD_FOO`. The second parameter is the value to which the variable is set. The key word `CACHE` tells CMake to create the variable in its cache. Cache variables will be saved, while other variables just exist in a single CMake run[5]. The remainder tells CMake the type of the variable and the description shown in the GUI.

The `IF` command is self-explanatory. Note that each `IF` needs a corresponding `ENDIF` with the same parameter. We now only include the `foolib` subdirectory,

[5]You may compare this to global and local variables

if out variable is set to `ON`[6]. The `ADD_DEFINITIONS` command tells CMake to use the given compiler flags. In this case, we define the compiler variable `USEFOO`, which is used by `test.cpp` to decide whether to use the foo library or not. CMake automatically rewrites the parameters in such a way that they are usable both on Windows and Unix systems without changing the `CMakeLists.txt`.

1.5 Installing the foo library

If the foo library is compiled, the user may want to install it to his default library path. Append the following code to the foolib's `CMakeLists.txt`:

```
INSTALL (TARGETS foo
         LIBRARY DESTINATION ${CMAKE_INSTALL_PREFIX}/lib
         ARCHIVE DESTINATION ${CMAKE_INSTALL_PREFIX}/lib
         RUNTIME DESTINATION ${CMAKE_INSTALL_PREFIX}/lib)
INSTALL (DIRECTORY .
         DESTINATION ${CMAKE_INSTALL_PREFIX}/include/foolib
         FILES_MATCHING PATTERN "*.h"
         PATTERN ".svn" EXCLUDE
         PATTERN "CMakeFiles" EXCLUDE)
```

CMake will generate either a `make install` target or a special install project in the Visual Basic solution. The first `INSTALL` tells CMake how to install the library itself. After the keyword `TARGETS`, you may specify arbitrary many targets. The other parameters determine where to put those targets. The cache variable `CMAKE_INSTALL_PREFIX` is used and set by CMake.

The second `INSTALL` copies the header structure. This is especially important for large libraries with many subdirectories. This command defines which headers to copy, where to put them and which subdirectories are to be excluded. Here, we exclude SVN directories[7] and everything that is created by CMake, if you use an in-source-build.

If you run CMake from the console, you have two possibilities to change cache variables. You can either set the variable via the parameter `-DEXAMPLE_BUILD_FOO=OFF` or you can use `ccmake` instead of `cmake`. `ccmake` shows all cache variables and allows the user to change them.

2 Vista default structures

2.1 Using Vista

The following is the `CMakeLists.txt` from Vista's geometry demo:

```
01 cmake_minimum_required(VERSION 2.6)
02 PROJECT (VISTADEMO_Geometry)
```

[6] You can also use `TRUE` or `1`. `FALSE` and `1` can be used instead of `OFF`.

[7] Depending on your project, you may want to ignore CVS or GIT directories here.

```
03
04 SET (EXEC_NAME VistaGeometryDemo)
05
06 FIND_PACKAGE(VISTA REQUIRED)
07
08 SET (files
09     main.cpp
10     GeometryDemoAppl.cpp
11     GeometryDemoAppl.h
12 )
13 SOURCE_GROUP ("Source Files" FILES ${files})
14
15 ADD_EXECUTABLE (${EXEC_NAME}
16     ${files}
17 )
18
19 TARGET_LINK_LIBRARIES (${EXEC_NAME}
20     ${VistaAspects}
21     ${VistaKernel}
22     ${VistaMath}
23 )
24
25 IF (UNIX)
26     INCLUDE (${VISTA_DIR}/VISTAShellScripts.cmake)
27     CREATE_SHELLSCRIPT (${CMAKE_CURRENT_BINARY_DIR}/start.sh
          "cd ${CMAKE_CURRENT_SOURCE_DIR}\n${CMAKE_CURRENT_BINARY_DIR}
          /${EXEC_NAME} $1 $2 $3 $4 $5 $6 $7 $8 $9")
28 ENDIF (UNIX)
```

The most important new command is the `FIND_PACKAGE` command in line 6. CMake will search for a file called `VISTAConfig.cmake` or `FindVISTA.cmake`. Vista comes with `VISTAConfig.cmake`, which sets up everything that is needed by CMake to link your program against the Vista libraries. To enable CMake to find this file, you need to specify its path in the variable `VISTA_DIR`, which has to be set via the console using `cmake` or `ccmake` or via the GUI before the first configuration step. Vista will install the file to `${CMAKE_INSTALL_PREFIX}/cmake`.

The `SOURCE_GROUP` command in line 13 puts the source files given in the variable `files` into a group called `Source Files`. This only concerns IDEs like Visual Studio. Makefiles don't mind source groups.

Lines 20-22 use variables that are defined by the `VISTAConfig.cmake`. The kernel library for example is called `VistaKernel` in release mode but `VistaKernelD` in debug mode. This is automatically reflected by the `VistaKernel` variable. Each Vista library has its corresponding variable.

For Unix systems[8], there is the possibility to create startup scripts for your executable. With these scripts you don't have to set environment variables any more. The command `CREATE_SHELLSCRIPT` takes the name of the script, that

[8]The key word `UNIX` is defined whenever CMake is run on a Unix system. For Windows systems, use the key word `WIN32` instead.

should be created, as the first parameter and the code to be executed as the second parameter. This command is defined in VISTAShellScripts.cmake, which is included in line 26.

2.2 Libraries with many directories

Sometimes you have a single library consisting of many source files grouped in a set of possibly nested directories. There are some different possibilities to reflect this in the CMakeLists.txt. We use a possibility that tries to minimize to effort necessary when source files change, but also reflects the directory structure in IDE projects. The following is a part of the kernel's CMakeLists.txt:

```
VistaKernel:
001 IF (WIN32)
002     ADD_DEFINITIONS(-DVISTAKERNEL_EXPORTS -DOSG_WITH_GIF -DOSG_WITH_TIF
             -DOSG_WITH_JPG -DOSG_BUILD_DLL -D_OSG_HAVE_CONFIGURED_H_ -wd4231)
003     SET (LIBRARIES
004         optimized OSGWindowGLUT
007         debug OSGWindowGLUTD
010     )
011     LINK_DIRECTORIES("${OSG_ROOT}/lib")
012 ELSEIF(UNIX)
013     SET (LIBRARIES
014         OSGWindowGLUT
017     )
018     LINK_DIRECTORIES("${OSG_ROOT}/lib/opt")
019 ENDIF(WIN32)
020
021 INCLUDE_DIRECTORIES("${OSG_ROOT}/include")
022
023 SET (dirFiles
024     VistaClientDataTunnel.cpp
025     VistaClusterAux.cpp
060 )
061 SOURCE_GROUP ("Source Files" FILES ${dirFiles})
062
072 ADD_SUBDIRECTORY (InteractionManager)
073 SOURCE_GROUP ("Source Files\\InteractionManager"
        FILES ${dirFiles_InteractionManager})
074 SOURCE_GROUP ("Source Files\\InteractionManager\\DfnNodes"
        FILES ${dirFiles_InteractionManager_DfnNodes})
088
089 ADD_LIBRARY (VistaKernel
090     ${dirFiles}
094     ${dirFiles_InteractionManager}
095     ${dirFiles_InteractionManager_DfnNodes}
101 )
102
103 TARGET_LINK_LIBRARIES (VistaKernel
    VistaDataFlowNet
    ${LIBRARIES}
)

# Include link paths for OpenSG so they will be found when linking against the kernel
IF (VISTA_USE_RPATH)
    IF (UNIX)
        SET_TARGET_PROPERTIES (VistaKernel PROPERTIES
            INSTALL_RPATH "${OSG_ROOT}/lib/opt"
        )
    ELSEIF (WIN32)
        SET_TARGET_PROPERTIES (VistaKernel PROPERTIES
            INSTALL_RPATH "${OSG_ROOT}/lib"
        )
    ENDIF (UNIX)
ENDIF (VISTA_USE_RPATH)

VistaKernel/InteractionManager:
SET (relDir "InteractionManager")
SET (dirFiles_in
    CMakeLists.txt
    VistaDriverWindowAspect.cpp
    VistaIntentionSelect.cpp
    VistaDriverWindowAspect.h
    VistaIntentionSelect.h
)

FOREACH (file ${dirFiles_in})
    LIST (APPEND dirFiles_InteractionManager "${relDir}/${file}")
ENDFOREACH (file)

ADD_SUBDIRECTORY (DfnNodes)

SET (dirFiles_InteractionManager ${dirFiles_InteractionManager} PARENT_SCOPE)
SET (dirFiles_InteractionManager_DfnNodes ${dirFiles_InteractionManager_DfnNodes} PARENT_SCOPE)

VistaKernel/InteractionManager/DfnNodes:
SET (relDir "InteractionManager/DfnNodes")
SET (dirFiles_in
    CMakeLists.txt
    VistaDfn3DMouseTransformNode.cpp
    VistaDfnClusterNodeInfoNode.cpp
    VistaDfn3DMouseTransformNode.h
    VistaDfnClusterNodeInfoNode.h
)

FOREACH (file ${dirFiles_in})
```

```
    LIST (APPEND dirFiles_InteractionManager_DfnNodes "${relDir}/${file}")
ENDFOREACH (file)

SET (dirFiles_InteractionManager_DfnNodes ${dirFiles_InteractionManager
    _DfnNodes} PARENT_SCOPE)
```

Each subdirectory has its own `CMakeLists.txt` which contains information about all needed files in this directory[9] along with the actual directory name, relative to the library's top directory. Subdirectories first prepend the file names with this directory. We do not want to create file name lists in the cache, so all variables are defined locally. To be able to access them on the above layer, we have to re-define them with the key word `PARENT_SCOPE`. In this example, we have a subsubdirectory. The `InteractionManager` directory has to keep track of this by re-defining the variable that was re-defined by the `DfnNodes` directory.

The toplevel `CMakeLists.txt` can now access all file name lists. It creates source groups for IDEs and merges the files for the library target. Most of this work can be done automatically by our Python script, which you can find in the directory `VistaCoreLibs/VistaBuild/CMake`

The cache variable `VISTA_USE_RPATH` is defined in the toplevel `CMakeLists.txt`. If it is set to `ON`, the kernel library will be built with rpath support. This tells the system where to search for built-in libraries. In our case, we want the kernel library to find the OpenSG libraries.

2.3 Resetting the cache

There is no reasonable way to reset the cache without deleting the whole build directory. For this reason we have introduced the variable `RESET_CACHE`. Many of Vista's CMake files accept this variable and reset their cache variables, if `RESET_CACHE` is set to `ON`. To use this variable, you may simply add it to your own executable's CMake file. If your `CMakeLists.txt` does not offer this variable, you can also set it on your own. See that you set `RESET_CACHE` to `OFF` after resetting the cache!

2.4 Config files and prototypes

Vista uses a prototyped Config file. Config files are provided by libraries built with CMake for easy access to them. Each Config file has the form `FOOConfig.cmake`, if the corresponding project is named `FOO`. A Config file is included with the `FIND_PACKAGE` command. If `FOOConfig.cmake` is not in a standard install directory, you may specify its path by setting the variable `FOO_DIR`.

Prototype in our case means, that we provide an unfinished file, which is completed by CMake within a configure run. The prototype files have prototype variables in the form `<<VARNAME>>`[10]. These variables are instantiated regarding

[9] For convenience, we add the `CMakeLists.txt` here, so that CMake includes them in IDE projects. This allows IDEs to automatically rebuild their projects if the CMake files have changed.

[10] Please note, that this is not a CMake standard. We use this form for our own software.

the user given cache variables and environment settings. An instantiated Config file prototype is specific to the system on which it was created.

The following is a part of Vista's Config file prototype:

```
# Create cache variables
IF (RESET_CACHE)
    SET (VISTA_INC_DIR <<VISTA_INC_DIR>> CACHE FILEPATH
    "Path where ViSTA includes are installed, relative to
    VISTA_INSTALL_PREFIX" FORCE)
    SET (VISTA_LIB_DIR <<VISTA_LIB_DIR>> CACHE FILEPATH
    "Path where ViSTA libraries are installed, relative to
    VISTA_INSTALL_PREFIX" FORCE)
    SET (VISTA_INSTALL_PREFIX <<VISTA_INSTALL_PREFIX>>
    CACHE FILEPATH "Root path of your ViSTA installation"
    FORCE)
ELSE ()
    SET (VISTA_INC_DIR <<VISTA_INC_DIR>> CACHE FILEPATH
    "Path where ViSTA includes are installed, relative to
    VISTA_INSTALL_PREFIX")
    SET (VISTA_LIB_DIR <<VISTA_LIB_DIR>> CACHE FILEPATH
    "Path where ViSTA libraries are installed, relative to
    VISTA_INSTALL_PREFIX")
    SET (VISTA_INSTALL_PREFIX <<VISTA_INSTALL_PREFIX>>
    CACHE FILEPATH "Root path of your ViSTA installation")
ENDIF (RESET_CACHE)

# Add the includes and libraries to the compiler/linker
# search directories
INCLUDE_DIRECTORIES(${VISTA_INSTALL_PREFIX}/${VISTA_INC_DIR})
LINK_DIRECTORIES(${VISTA_INSTALL_PREFIX}/${VISTA_LIB_DIR})

# Add platform defines
IF(UNIX)
    ADD_DEFINITIONS(-DLINUX)
ELSEIF(WIN32)
    ADD_DEFINITIONS(-DWIN32)
ENDIF(UNIX)

# Add library name variables
SET (VistaAspects
    optimized VistaAspects
    debug VistaAspectsD
)

# Set OpenSG paths for usage in other projects
SET (OPENSG_INC_DIR <<OPENSG_INC_DIR>>)
SET (OPENSG_LIB_DIR <<OPENSG_LIB_DIR>>)
# Set OpenSG as link path
LINK_DIRECTORIES(${OPENSG_LIB_DIR})
```

Here, we use the following prototype variables:

- `<<VISTA_INC_DIR>>`, the directory, where Vista include files are installed
- `<<VISTA_LIB_DIR>>`, the directory, where Vista library files are installed
- `<<VISTA_INSTALL_PREFIX>>`, the main directory of the Vista installation
- `<<OPENSG_INC_DIR>>`, the directory, where OpenSG include files are installed
- `<<OPENSG_LIB_DIR>>`, the directory, where OpenSG library files are installed

When Vista is configured, this prototype gets instantiated. The resulting file will then be installed. Including this file does the following:

- The variables `VISTA_LIB_DIR`, `VISTA_INC_DIR`, `VISTA_INSTALL_PREFIX`, `OPENSG_LIB_DIR` and `OPENSG_INC_DIR` are set according to your installation.
- Both the Vista and the OpenSG library path are set as link directories. Those are directories in which the linker will search for libraries.
- The Vista include path is added to CMake's include directories so that you can access all the headers.
- For each Vista library there is a variable with the same name. These variables include both the release mode and the debug mode library so that the makefile or the IDE project will link your code against the right file.

The following code instantiates the prototype:

```
# First read the prototype
FILE (READ VistaBuild/CMake/VISTAConfig.cmake_proto CONFIG_FILE)
# Then replace the prototype variables
STRING (REPLACE "<<VISTA_INC_DIR>>"  "${VISTA_INSTALL_INC_DIR}"
    CONFIG_FILE ${CONFIG_FILE})
STRING (REPLACE "<<VISTA_LIB_DIR>>"  "${VISTA_INSTALL_LIB_DIR}"
    CONFIG_FILE ${CONFIG_FILE})
STRING (REPLACE "<<VISTA_INSTALL_PREFIX>>"  "${CMAKE_INSTALL_PREFIX}"
    CONFIG_FILE ${CONFIG_FILE})
STRING (REPLACE "<<OPENSG_INC_DIR>>" "${OSG_ROOT}/include"
    CONFIG_FILE ${CONFIG_FILE})
IF (UNIX)
    STRING (REPLACE "<<OPENSG_LIB_DIR>>" "${OSG_ROOT}/lib/opt"
          CONFIG_FILE ${CONFIG_FILE})
ELSEIF (WIN32)
    STRING (REPLACE "<<OPENSG_LIB_DIR>>" "${OSG_ROOT}/lib"
          CONFIG_FILE ${CONFIG_FILE})
ENDIF (UNIX)
# And write the file to the build directory
FILE (WRITE ${CMAKE_CURRENT_BINARY_DIR}/cmake/VISTAConfig.cmake
     ${CONFIG_FILE})
```