Tool Development with C++

Koen Samyn

March 21, 2023

Introduction

Developing tools and integrating your tool in a professional environment can be big part of your professional career. A Tool Development professional has a career track is a bit more stable and does not depend as much on the hectic cycles of game development.

Furthermore, even as a game play or graphics programmer, the need will sometimes arise to **automate** parts of the development process.

This part of the Tool Development course focuses on using C++ as a language for the development of:

1. **Command line tools**: This is still a powerful way to add functionality to a tool chain, specifically in the context of CI/CD paradigm. (Continuous Integration/Continous Deployment).
2. **GUI tools**: GUI toolkits such as Qt, Imgui, Juce provide a way to develop tools with a graphical interface for tools that need more interaction from the user. Examples are node based tools to develop conversation systems, shader tools, ...

Furthermore you will learn to:

1. Setup build systems with CMake
2. Create modular projects with subprojects that consist of your own code and projects from github for example. (SDL, Qt, ...)
3. Setup unit testing within your project.
4. Create installers for your project.

For the intrepid and restless developer there is also an additional (but not mandatory) part about developing your own language with ANTLR4. Who knows, you might develop the next big thing in programming land.

Organization

The C++ part takes **6 weeks**, which is 5 weeks of content and a final week to hand in a small project which counts for 20% of the score for the C++ part.

The C# part (taught by Lies Pinket) is also 6 weeks with the same arrangement.

Normally there are 3 hours per session, with one hour for feedback. The feedback session will be used for help with the project or for help with the exercises depending on your own progress.

The final exam counts for 80% of the total score, evenly divided over the two parts of this module. This is a practical exam where you will need to program a small application with the focus on building with CMake.

Overview:

Part	*Percentage*
C++ assignment	10%
C++ final exam	40%
C# assignment	10%
C# final exam	40%
	100%

For the retake, only the final exam is organized (no assignment).

Assignment

For this part of the course you will make a command line tool and GUI that automates part of a production pipeline. In the course we will demonstratie the conversion of a file format into another file format (json to obj). You can continue on this example or you can formulate your own tool.

Functional Requirements

The command line tool has to satisfy these requirements:

1. it must be possible to specify what the input file is.
2. it must be possible to specify where to output the generated file.

3. at least one extra parameter to this command line tool.

The user interface has the same function thing as the command line tool, but allows the user to select the input file and output file with a file chooser (we will see this in the Qt section).
To be clear the user interface needs the following elements:

1. Choose the input file with a file chooser
2. Choose the output file with a file chooser
3. Ok/Cancel buttons
4. A report (in table format) of the output.

Build requirements

Als the source code must be built with CMake and the project must be clearly structured into subprojects.
The project must also contain unit tests to test if the functionality is correct:

1. MainProject
   (a) Common Code project
   (b) Command line project
   (c) GUI project
   (d) Unit tests

Finally, the project needs an installer that installs the tools on the PC of the user. As we will see this is very easy to do with CMake.

Chapter 1

Command line tools

The command line is still a powerful tool in today's world where we are used to graphical user interfaces. In the fast paced word of game development there is sometimes not enough time to develop a GUI and a tool that works with input from a file (or several files) may hit the sweet spot in terms of speed of development.

In this first chapter we will develop a tool that uses JSON input that contains a simplified minecraft **chunk** and convert this minecraft chunk into the object format. We will first perform a simple conversion with all **voxels** in the minecraft world represented as full cubes. As a second step we eliminate the connected faces in this small **voxel** world.

We start the explanation with the JSON file format, which is a really convenient way to store text data, and the rapidjson library which will be demonstrated in C++.

Then we move on to accepting a json file and command line parameters as input to a command line executable, parsing it with rapidjson and generating the output.

1.1 RapidJson library

We are in luck, the rapid json library is header only, so we can just include it in our project instead of having to link against a separate dll or lib file.

As first step, create a visual studio project that is a windows console application. In the new project dialog select **C++** as language (off course) and "**Console App**" as project type.

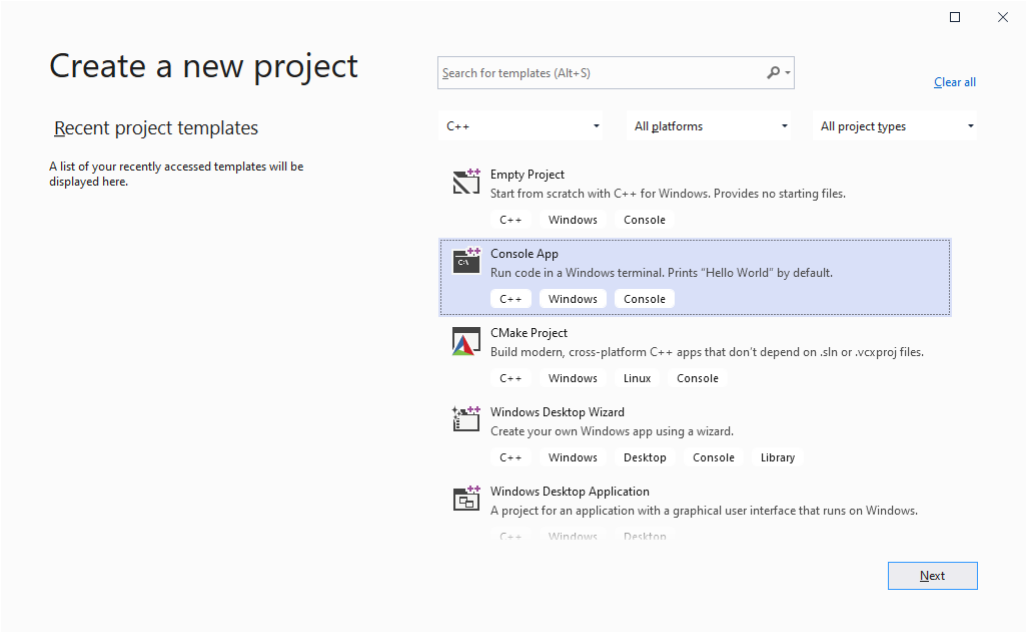


Figure 1.1: New C++ Console app project

Choose a convenient location for your new project. Typically for small projects I put the solution folder and C++ project folder in the same location, but you are free to choose what you want to do here.

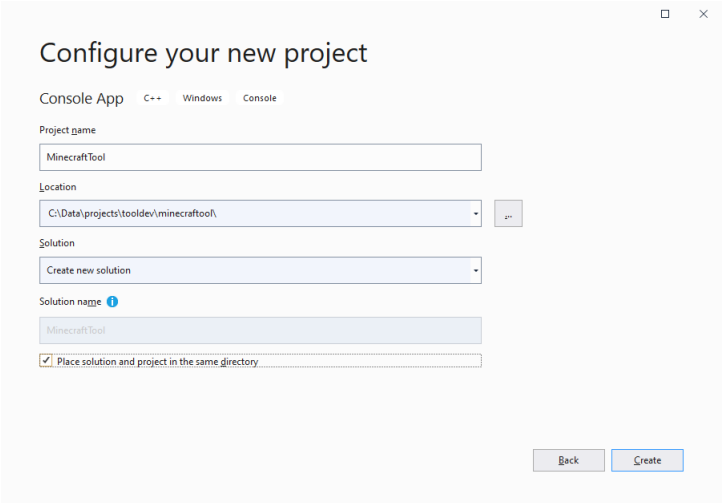


Figure 1.2: Project settings

Finally click on the **Create** button.

The project

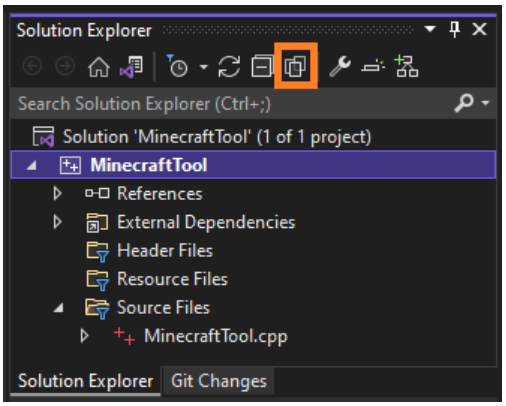


Figure 1.3: All files

A handy tip for working with visual studio files is the "**Show all files**" button. This button allows you the see the actual file structure on disk. This is useful to see files that you forgot to include or that maybe are in the wrong place, or even to open resources from within Visual Studio.

After clicking this button, you can see that the only file currently in the project is the `MinecraftTool.cpp` file. You can also see the build results in this view. In my case I built an x64 debug version of the command line tool. Verify that there is a `MinecraftTool.exe` file in the structure of this project.

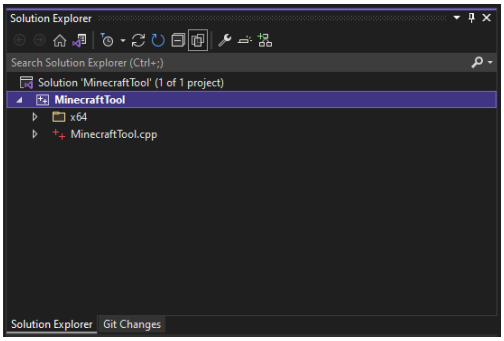


Figure 1.4: Project file structure

We now add the rapidjson files to our project. To know exactly where to put these files rightclick on the project (**MinecraftTool**) and select the following menu item:

Open Folder in File Explorer

Now you can dowload the RapidJson library from `https://github.com/Tencent/rapidjson/`. The documentation on `https://rapidjson.org` states that only the files in `include/rapidjson` are necessary if the files are included in the project. However, to download the project from github click the green Code button and select Download Zip.

Copy the zip file to the same location as the **solution** file of the minecraft tool and choose *extract here*.

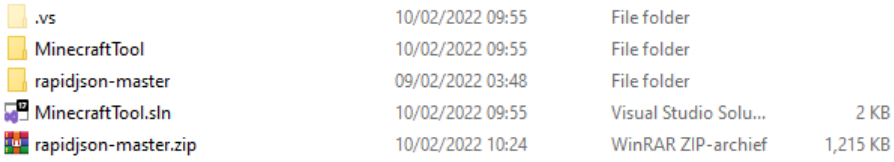


Figure 1.5: Rapidjson-master export location

To be able to use these header files, you now need to change the properties of the **MinecraftTool** project. Right click on the MinecraftTool project and choose *Properties*. Within the property pages of the project you can now navigate to the **C/C++** category and edit the **Additional Include Directories**:

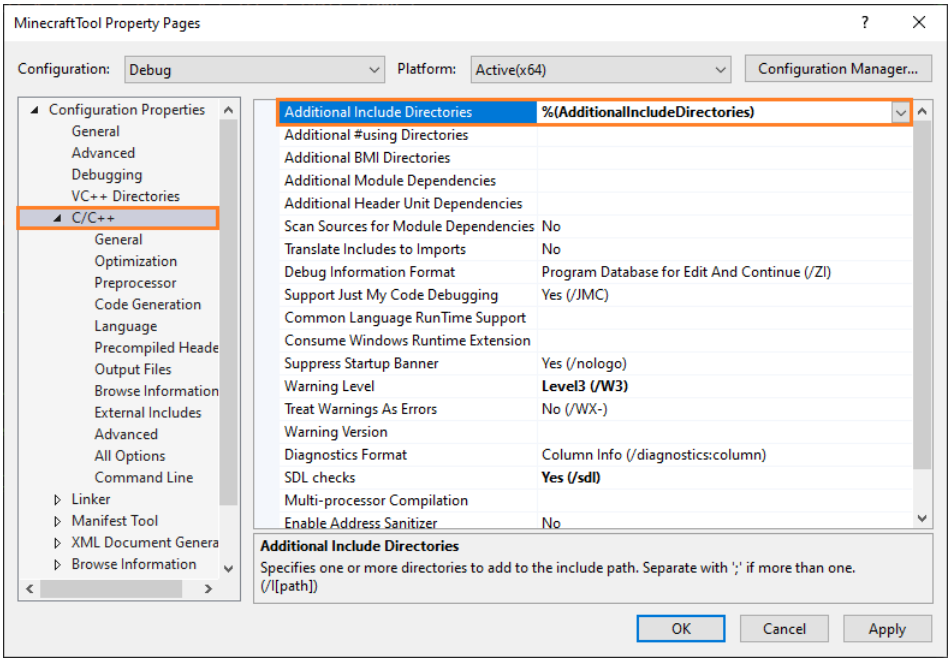


Figure 1.6: Project properties

Click on the drop down button of the **Additional Include Directories** property and select *<Edit...>*. You can now select the directory of the rapidjson-master project that contains the header files for rapidjson.

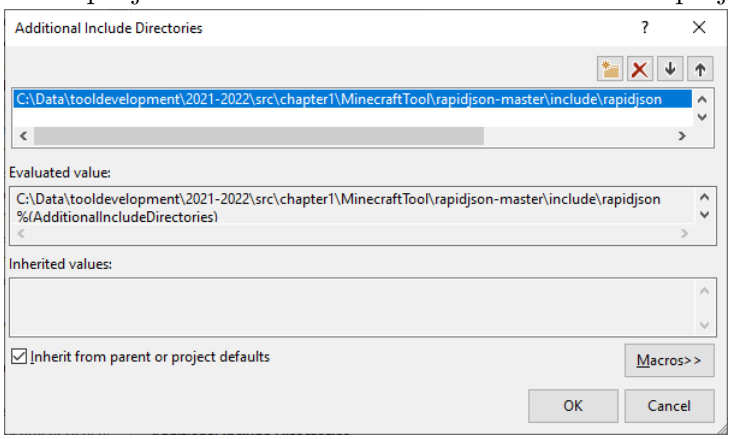
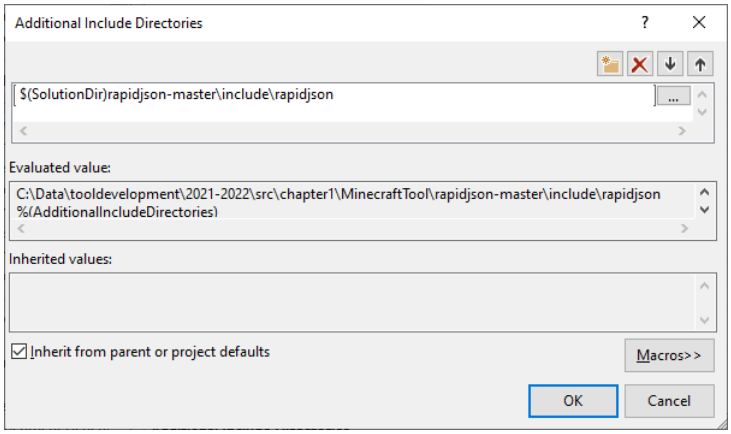


Figure 1.7: Project properties

A better way is to use relative paths that takes the location of the visual studio solution into account:



The $(SolutionDir) value is a Visual Studio **Macro** that contains the path of the current solution. This is a lot better because now we can 'give' the project to other programmers without problems.

> **Note**: This is one way to use the rapid json library in a project, but it is not the most **professional** way. We will later see how we can use a build system to be able to better structure these types of projects.

JSON itself

JSON is short for JavaScript Object Notation, and is a convenient format to write down an object structure in a textual format. Because the notation is based on the JavaScript language which is a C derivate, the notation should not pose too much problems. The complete structure of the json format is defined in `https://www.json.org/json-en.html`.

An **advantage** of this file format is that parsers and writers are available for almost every language including C++. In terms of ease and speed development this is a definite plus, because as a programmer you just need to think about the organization of the file and the how the data should be converted to C++ objects. Compared to managed languages (C# and Java) however, C++ has the distinct **disadvantage** that the process of converting data to objects can not be performed automatically.

Another advantage is that **arrays** are well supported in JSON, certainly in comparison with XML. It is easy to use arrays in a JSON file, but also (and maybe more important) it is easy to iterate over the contents of an array in the C++ code.

We will discuss the structure of this file format with some simple examples.

The smallest JSON examples

There are two versions of a very small json file.

This first version is an empty **object**. Objects are defined (as in C++) with curly braces:

```
1 {}
```

The second version is an empty **array**. The contents of an array is again defined as in C++ with **square** brackets:

```
1 []
```

A JSON object

This JSON example contains a single object with two members, x and y with values 100 and 50. Look closely at this file, what are some **details** that

attract your attention?

```
1 {
2 "x":100,
3 "y":50
4 }
```

On line 2 and 3 you can see that the **data members** are always defined as **strings**, meaning that they are always enclosed with **double quotes**. A second observation should be that the colon (:) symbol is used to assign a value to a data member. Decimal and integral values are supported as values for these datamembers.

Finally an object is defined as a **comma separated list** as members with their values.

> To check the contents of a json file you can go to the following webpage `https://jsonlint.com/`

A JSON array

As mentioned before, a JSON array is enclosed with square brackets and is also a comma separated list of values:

```
1 [true,false,true]
```

One major difference with the way that arrays in C++ work is that arrays in JSON can contain **mixed** types:

```
1 [true,100,"a string",28.3]
```

Now we can think about putting two arrays in a single JSON file. There are two ways that this can be done. An array is an object so it is possible to make an array of arrays. Secondly it is possible to create an object with two data members that contain an array object.

First solution:

```
[
    [true,false,true],
    [true,100,"a string",28.3]
]
```

Second solution:

```
{
    "array1":[true,false,true],
    "array2":[true,100,"a string",28.3]
}
```

Again, mind the small details (and differences) in the two solutions.

A JSON array with JSON objects

It is off course also possible to define a JSON array that consists of JSON objects:

```
[
    {
        "x":10,
        "y":15,
        "type":"dirt"
    },
    {
        "x":12,
        "y":14,
        "type":"water"
    },
    {
        "x":12,
        "y":14,
        "type":"rock"
    }
]
```

All though it is possible to create an array with different object types and values, I would not recommend it. You will have a bad time converting everything into objects if you mix and match objects too much.

> Model your JSON files as you would model a class in C++. Avoid inheritance structures because they make it harder to parse your file.

Further modelling

Text files typically contain more bloat than a binary equivalent. JSON is not too bad compared to other data model files, but we still need to think about **efficient** structures.

For example we can model a location as an array:

```
[
    {
        "loc":[10,15],
        "type":"dirt"
    },
    {
        "loc":[12,14],
        "type":"water"
    },
    {
        "loc":[17,-12],
        "type":"rock"
    }
]
```

1.2 Parsing of JSON files

We now turn our attention to parsing a JSON file in C++. We first just create a valid JSON file in **memory** as a string to make it a bit easier to understand what is going on.

```
1  #include <iostream>
2
3  #include "rapidjson.h"
4  #include "document.h"
5  #include "stream.h"
6  int main()
7  {
8    // be careful not to use json in the file itself
9    const char* jsonContents = R"json( {"x": 10, "y":15} )json";
10
11   rapidjson::Document jsonDoc;
12   jsonDoc.Parse(jsonContents);
13
14   const rapidjson::Value& xVal = jsonDoc["x"];
15   const rapidjson::Value& yVal = jsonDoc["y"];
16
17   std::cout << "Point : "
18     << xVal.GetDouble() << ","
19     << yVal.GetDouble() << std::endl;
20 }
```

On **line 9** we use a special C++ string. This string begins with `R"json(` and ends with `)json"`. The **actual** contents of the string is then :

```
{"x": 10, "y":15}
```

With a normal string the double quotes (") would have to be **escaped** with a backslash in the json code string, which makes it unreadable in my opinion. With this nifty feature code like this looks a lot cleaner and readable.

On **line 11** a Document object is created which will be used as storage for all the information in the JSON file. Since C++ doesn't do a lot of reflection intermediate objects (such as the Value object) will be used to store this information, and we need to convert this to our own data structures if necessary.

On **line 14 and 15** the x and y data members are retrieved from the root of the JSON structure. We store the result **by reference** to avoid unnecessary copying of the Value object.

Finally on **line 17-19** we output the actual by calling `GetDouble()` on `xVal` and `yVal`.

Parsing arrays

Arrays are the only way to get multiple instances of data into a file, so it is important to take a look at how arrays can be parsed.

The json content now has an **array** in the root with 3 json objects as elements. As in the first example, the objects contain a position defined by an x and y datamember.

```
1  #include <iostream>
2
3  #include "rapidjson.h"
4  #include "document.h"
5  #include "stream.h"
6  int main()
7  {
8    const char* jsonContents = R"json(
9    [
10     {"x": 10, "y":15},
11     {"x": 17, "y":21},
12     {"x": 41, "y":33}
13   ]
14   )json";
15
16   rapidjson::Document jsonDoc;
17   jsonDoc.Parse(jsonContents);
18
19   using rapidjson::Value;
20
21   for (Value::ConstValueIterator itr = jsonDoc.Begin();
22       itr != jsonDoc.End(); ++itr)
23   {
24     const Value& position = *itr;
25     const Value& xVal = position["x"];
26     const Value& yVal = position["y"];
27     std::cout << "Point : " <<
28       xVal.GetDouble() << "," <<
29       yVal.GetDouble() << std::endl;
30   }
31 }
```

After parsing , the `jsonDoc` objectc now contains an array. We can traverse this array with a const iterator that is also provided by the rapidjson library.

On line 24 we dereference the iterator and get a rapidjson::Value object as result. We again store this Value object by reference to avoid unnecessary copying.

On line 25 and 26 we can now get the datamembers of this object x and y in the same manner as we did with the first example.

Parsing arrays with indexed access

In the following listing the json file was **remodeled** for efficiency. Instead of using two datamembers for x and y, we create a single datamember for the location and model the coordinates as an array. Rapidjson also allows you to use an index to get an element of this array, as the following listing shows:

```
1  #include <iostream>
2
3  #include "rapidjson.h"
4  #include "document.h"
5  #include "stream.h"
6  int main()
7  {
8    const char* jsonContents = R"json(
9    [
10     {"loc": [10, 15] },
11     {"loc": [17, 21] },
12     {"loc": [41, 33] }
13   ]
14   )json";
15
16   rapidjson::Document jsonDoc;
17   jsonDoc.Parse(jsonContents);
18
19   using rapidjson::Value;
20
21   for (Value::ConstValueIterator itr = jsonDoc.Begin();
22       itr != jsonDoc.End(); ++itr)
23   {
24     const Value& position = *itr;
25     const Value& loc = position["loc"];
26
27     const Value& xVal = loc[0];
28     const Value& yVal = loc[1];
29
30     std::cout << "Point : " <<
31       xVal.GetDouble() << "," <<
32       yVal.GetDouble() << std::endl;
33   }
34 }
```

the major change is off course the json file itself from line 9 to 13 where an array is used to model a 2D location.

The Value that we get when we query the **loc** position for each element is now also an array.

On line 27 and 28 we now simply get the first (index 0) and second (index 1) element of this array. We know that these elements contain a double so we can again use the **`GetDouble`** method to get the value as a double.

1.3 Reading from files

Using a **hardcoded** `const char*` to try out the features of json is good, but in a tool or a game this is the exception. The information will typically come from a file somewhere in the file system.

Reading a text file is easy ... right? Not so fast, there are a number of issues you will have to deal with. The first major issue is the text file encoding which is a confusing area since a lot of older encodings are still supported.

Secondly, there are a number of ways to read a file into memory, and we shall take a look at a couple of them.

File encodings

Reading a text file sounds simple, but a major source of problems is the encoding of the file. The most commonly used file format is **`UTF-8`**, but other encodings are possible:

- `ASCII`
- `Windows-1252`
- `UTF-16`
- `UTF-32`

Something that you need to be aware of is the byte order mark (BOM). The byte order mark is **optional** but if a file starts with a byte order mark you can be relatively sure that the file is an **`UTF-8`** file. The fact that a BOM is optional makes it a bit of a hit and a miss, but if it is present it has to be handled.

See `https://en.wikipedia.org/wiki/Byte_order_mark` for more information.

For our purpose it is sufficient to know that an UTF-8 file **can** start with the following hexadecimal pattern: `EF BB BF`. For example Notepad++ uses a default UTF-8 encoding without the BOM header part.

The best way to try this out is to embed a unicode character into a json file and see what happens in terms of bytes.

Let's take a look at the following json file with a special character inside:

```
1 {
2 "name":"Ω"
3 }
```

When we look at this file in a hex editor (e.g. HxD) we can see the byte patterns that are used to include this character. The special character Ω is highlighted and is encoded with 3 bytes, namely `E2 84 A6`.

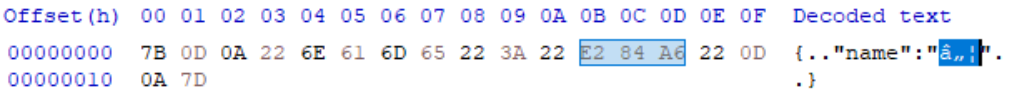


Figure 1.8: Binary representation of JSON file

First of all note that there is no BOM header for this file.

You might have been under the impression that all unicode file encodings use 2 bytes for every character. This is not the case however. The UTF-8 encoding uses special byte patterns to indicate that a special character (with character > 255) follows.

The unicode hexadecimal code for this character Ω is : `0x2126`. So how does the sequence `E2 84 A6` get converted to the correct unicode?

The following table shows how unicode characters are converted to a binary representation. The table is set up in a manner that allows for an **encoder** to work efficiently because all that has to be done for encoding is to look at the range of the character and distribute the bit pattern over the x's in the following table.

```
0x00000000 - 0x0000007F (0-127):
    0xxxxxxx

0x00000080 - 0x000007FF (128-2047):
    110xxxxx 10xxxxxx

0x00000800 - 0x0000FFFF (2048-65535):
    1110xxxx 10xxxxxx 10xxxxxx

0x00010000 - 0x001FFFFF (65536-2079151):
    11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

The x's mean that the bit pattern of the unicode character is copied over to these locations. For the first 127 characters it is simple, nothing changes. This is off course because the bit pattern for the first 127 characters can have at most 7 (**seven**) significant bits. This is already a major advantage for UTF-8. For most characters nothing changes.

Example for range 0-127

Let's take a look at the character '~' which has the character code `126` in decimal and `7E` in hexadecimal or finally in binary form : `111 1110`.

Note that there are **7** significant bits in this pattern and we now just copy these bits in the bit pattern `0xxx xxxx` to get the final UTF8 value for this character: `0111 1110`.

Example for range 128-2047

We now take a look at the following character '®'. This character has the unicode `0x00AE` which is `174` in decimal which is indeed within the range of 128-2047.

The bit pattern for this character in unicode is : `1010 1110`.

The table of conversions indicates that we need to copy this bit pattern into the x's of the following pattern:

`110x xxxx 10xx xxxx`

To do this we need to work from right to left and group the bit pattern for the unicode into a group of 6 bits and a group of 5 bits:

For the bit pattern `1010 1110` the result is :

`0 0010 10 1110`

Note that the character '®' can be written with 8 bits so we **pad** the bit pattern with zeros on the left to get the required number of bits.

And after copying this into the UTF-8 pattern:

`1100 0010 10 1110`

Example for range 2048-65535

Let's take a look at the Ω - character that was present in the example json file and see if we can convert it to the correct bit pattern.

The unicode for this character in hexadecimal `2126`, in decimal this is 8486 which is in the third range of the conversion table.

The binary bit pattern for this character is : `0010 0001 0010 0110`

We need to copy these bits into the following UTF-8 pattern:

`1110xxxx 10xxxxxx 10xxxxxx`

The bit pattern for this character correctly grouped becomes:

`0010 00 0100 10 0110`

And pasted into the UTF-8 pattern the result is:

`1110 0010 10 00 0100 10 10 0110`

And in hexadecimal : `E2 84 A6` and this corresponds to the result we got in figure 1.8

Encoder versus decoder

You now know how the UTF-8 encoder works.

How would you write a decoder? In other words suppose you know a file is encoded with the UTF-8 encoding, how can you load this file as a `std::wstring` ?

> Writing software that handles text files in a consistent manner is challenging. Read the following article for an example of how difficult it can be : `https://preshing.com/20200727/automatically-detecting-text-encodings-in-cpp/`
> The following website also shows the results of various encodings for unicode characters:
> `https://www.branah.com/unicode-converter`

Current working directory

For a command line tool the current directory is very important because users expect a command line tool to be able to find a file relative to the **current** directory. In the following command line example three command line tools are used:

- `cd` : short for change directory. This tool accepts one argument which is the folder to navigate to. This folder can be **relative** to the current folder, but also can be an absolute directory path.
- `dir` : Prints out the contents of the current directory.
- `copy` : This command line tool copies a file to another location. In this example the first file (`utf8example.json`) is copied to the second file (`utf8example2.json`). In this example both files are **relative** to the current location.

```
c:\Data\tooldevelopment>cd chapter1files

c:\Data\tooldevelopment\chapter1files>dir
 Volume in drive C is Windows
 Volume Serial Number is 7ED0-607C

 Directory of c:\Data\tooldevelopment\chapter1files

01/02/2021  10:55    <DIR>          .
01/02/2021  10:55    <DIR>          ..
01/02/2021  10:55                18 utf8example.json
               1 File(s)             18 bytes
               2 Dir(s)  117,545,840,640 bytes free

c:\Data\tooldevelopment\chapter1files>copy utf8example.json utf8exampl2.json
        1 file(s) copied.

c:\Data\tooldevelopment\chapter1files>dir
 Volume in drive C is Windows
 Volume Serial Number is 7ED0-607C

 Directory of c:\Data\tooldevelopment\chapter1files

02/02/2021  09:20    <DIR>          .
02/02/2021  09:20    <DIR>          ..
01/02/2021  10:55                18 utf8exampl2.json
01/02/2021  10:55                18 utf8example.json
               2 File(s)             36 bytes
               2 Dir(s)  117,546,913,792 bytes free

c:\Data\tooldevelopment\chapter1files>
```

What can we learn from this? First of all there is a **difference** between the location of the executable and the current directory in the command line. For command line tools we need to know the current location in the command

line or in other words the location from where the executable was started and not necessarily the location of the executable itself.

This is in contrast with a game, where the location of the executable is typically used to load the assets that are required to run the game.

The location from where a file was executed is called the **current working directory** or short `CWD`. Let's write a small program to test this out:

```
1  #include <direct.h> // _getwcwd
2  #include <iostream>
3
4  int main()
5  {
6
7    wchar_t * currentWorkingDir = _wgetcwd(nullptr, 0);
8    if (currentWorkingDir != nullptr) {
9      std::wstring current_working_dir(currentWorkingDir);
10     std::wcout << current_working_dir << std::endl;
11   }
12   return 0;
13 }
```

On line 7 the function `_getwcwd` is called which returns a wide character buffer. If the function should fail for some reason, the return value will be equal to `nullptr`. It is possible to call this function with your own buffer, but you need to be sure that the buffer will be big enough to store the entire path of the current working directory.

After executing this program the following output can be expected:

```
1  C:\Data\projects\tooldev\minecraftool\MinecraftTool
```

This is already a nice illustration of the current working directory concept. The location of the executable itself is `MinecraftTool/x64/Debug/MinecraftTool.exe`. Visual Studio sets a current working directory because that makes it easier when you develop an application to store **resources** (e.g. obj files or shader files) in the same location regardless of wether you are building a win32 target, or debug, or release, or other type of application.

Let's go a step further and see if we can use the MinecraftTool from any location.

```
1  c:\Data\tooldevelopment\chapter1files>MinecraftTool
2  'MinecraftTool' is not recognized as an internal or external command,
3  operable program or batch file.
4
5  c:\Data\tooldevelopment\chapter1files>
```

Oops, the dreaded 'not recognized' error. We can easily fix this by adding the MinecraftTool.exe to the Path environment variable in Windows. To open the environment variables enter the following magical incantation in the command line:

```
rundll32 sysdm.cpl,EditEnvironmentVariables
```

It's also a good idea for a developer to have this as a shortcut on the desktop. Right click on the desktop and select: New > Shortcut and type in the command:

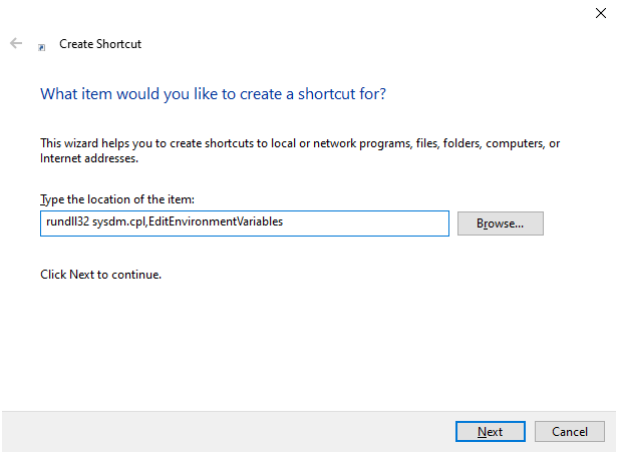


Figure 1.9: Create shortcut

Press Next and give a name to the shortcut:

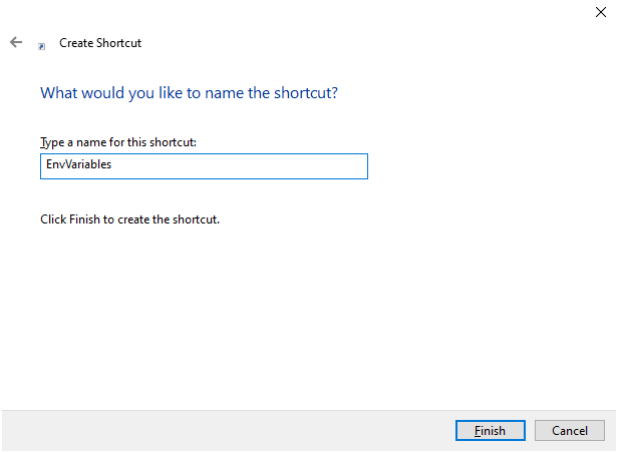


Figure 1.10: Give a name to the shortcut

You can now choose to remember the magical incantation or just click on the shortcut to start the environment variable editor. (I use the shortcut).

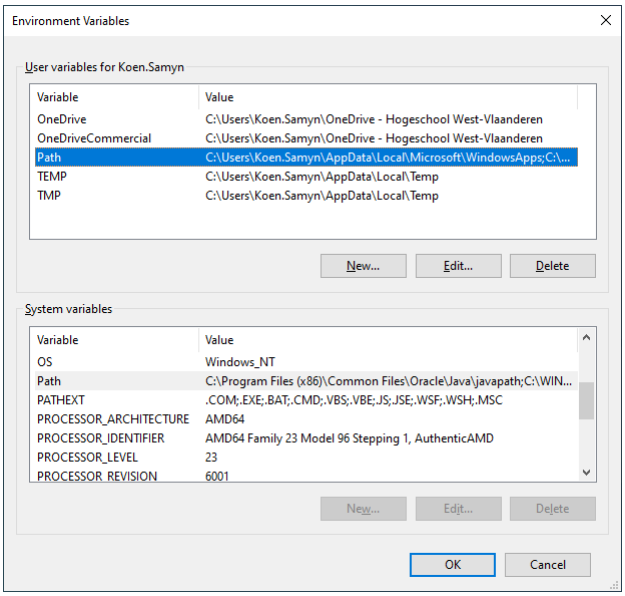


Figure 1.11: Environment variables

You can now edit the path for the current user (the user variables part). If you want to edit the path for all users and change the **System variables** you need to run the shortcut as an administrator. For our purposes it is sufficient to add the path of the executable to the user variables section.

As shown in the screenshot, select the `Path` variable and click Edit... . You now get an interface where you can add the location of the MinecraftTool.exe file:

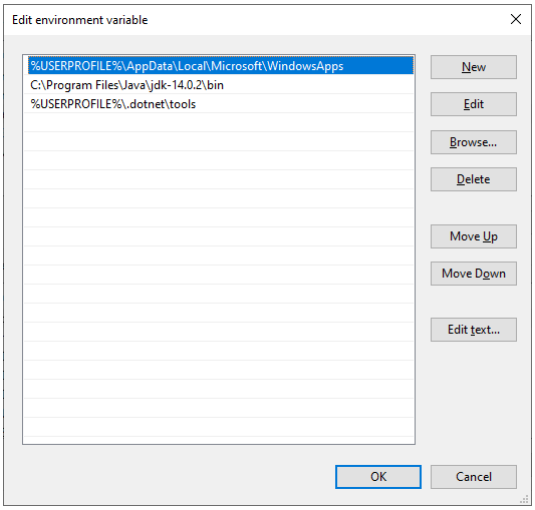


Figure 1.12: Edit Path variable

First click New and then Browse... then select the directory where the executable is stored:

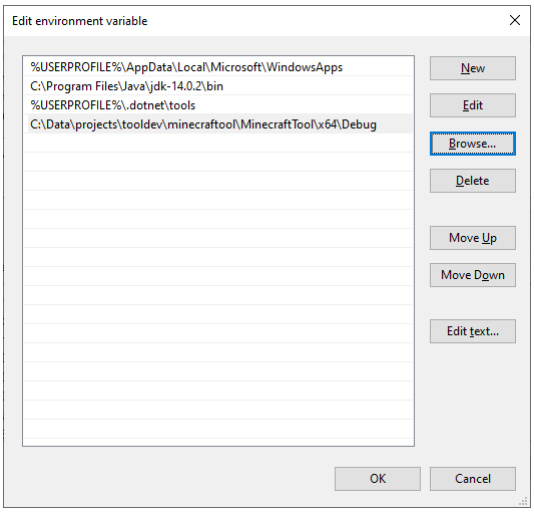


Figure 1.13: Result

Press OK and then OK again to close the environment variables editor.

We now try again to call our tool in the same command window:

```
c:\Data\tooldevelopment\chapter1files>MinecraftTool
'MinecraftTool' is not recognized as an internal or external command,
operable program or batch file.
```

Drats, it still doesn't work, what happened?

The command window makes a record of all the environment variables when it is started and does not get updates when the environment variables are changed. The solution is to start a new command window after environment variables are changed:

```
Microsoft Windows [Version 10.0.19041.572]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Koen.Samyn>MinecraftTool
C:\Users\Koen.Samyn
```

If we now call the tool, the correct current working directory is reported.

Read a json file

The json library provides some wrappers that can be used to efficiently read a json file and parse it:

```
1  #include <iostream>
2  #include <fstream>
3
4  #include "rapidjson.h"
5  #include "istreamwrapper.h"
6  #include "document.h"
7
8  int main()
9  {
10   if (std::ifstream is{ "utf8.json" })
11   {
12     rapidjson::IStreamWrapper isw{ is };
13
14     rapidjson::Document jsonDoc;
15     jsonDoc.ParseStream(isw);
16
17     const rapidjson::Value& valName = jsonDoc["name"];
18     const char* valString = valName.GetString();
19
20     std::cout << "name is : " << valString << std::endl;
21     return 0;
22   }
23   else {
24     return -2; // file not found.
25   }
26 }
```

On **line 10** we open an **std::ifstream** in the standard way (which defaults to interpreting the file as a *text* file). If the file does not exist or another error occurs, the main function returns the value **-2**.

Line 12-13: The rapid json library can work with `std::ifstream` objects if they are wrapped in an `IStreamWrapper` object.

Line 14-15 : Again we need to make a `rapidjson::Document` object and use the `ParseStream` method to parse it.

Ling 17-21 : We can now again use the `jsonDoc` object as an entry point to get all the data and output it again.

> Still to be done in the example is to apply the RAII principle. We will apply this principle in the next section, for now we focus on reading and parsing the JSON file.

Place the `utf8.json` file in the correct location based on the current working directory to make this example work.

```
C:\Data\projects\tooldev\minecraftool\MinecraftTool
name is : ├─

C:\Data\projects\tooldev\minecraftool\MinecraftTool\x64\Debug\MinecraftTool.exe (
    process 32636) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->
    Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

A new problem to solve. The GetString method returns a char* but the encoding was UTF-8 so we need to **decode** this character array to a unicode string. Notice that the characters that are shown correspond to the 3 byte UTF8 example that we calculate on page 21.

For command line tools we will use the easy way out, and change the way the console interprets the characters in the `std::string` variable.

```
// needed for SetConsoleOutput
#include <windows.h>


int main(void) {
// ....

// Set console code page to UTF-8 so console knows how to interpret string data
SetConsoleOutputCP(CP_UTF8);

// Enable buffering to prevent VS from chopping up UTF-8 byte sequences
setvbuf(stdout, nullptr, _IOFBF, 1000);

std::cout << "name is : " << utf8String << std::endl;

}
```

With these changes the string is rendered correctly:

```
C:\Data\projects\tooldev\minecraftool\MinecraftTool
name is : \omega
```

An advantage of keeping the string in its UTF8 format is that the number of bytes that is required to store strings is smaller in comparison with the number of bytes that is required if we would store the unicode code points for every character in the string. The **disadvantage** is that the conversion needs to be done when printing an UTF8 string to the console or other form of output.

UTF8 is the standard encoding for the rapidjson library. However, the rapidjson library does not convert the UTF8 strings to for example wstring objects. Instead the **raw** UTF8 encoded strings are kept in memory. If you need the unicode code points you will need to write a decode function yourself. In the C++ standard there used to be conversion routines but these routines are deprecated and the advice now is to rely on either third party libraries to perform these conversions, or write them yourself.

Command line arguments

The file to read is now **hardcoded** in the source code. This is obviously not the professional way so we need a way to pass parameters to the executable via the command line. In our sample application we accept one parameter, the filename of the json file we want to read.

Again we encounter the problem that the default for the main function is the old ASCII standard. Fortunately there is a `wmain` method with parameters that accept Unicode (wchar_t) characters.

Within the Visual Studio project we can define a command line parameter to make it easier to test the application. Right click on the project and select Properties. Within the Properties window of the project select the **Debugging** section and change the **command arguments**. Also check that you are changing the correct **configuration** (Debug).

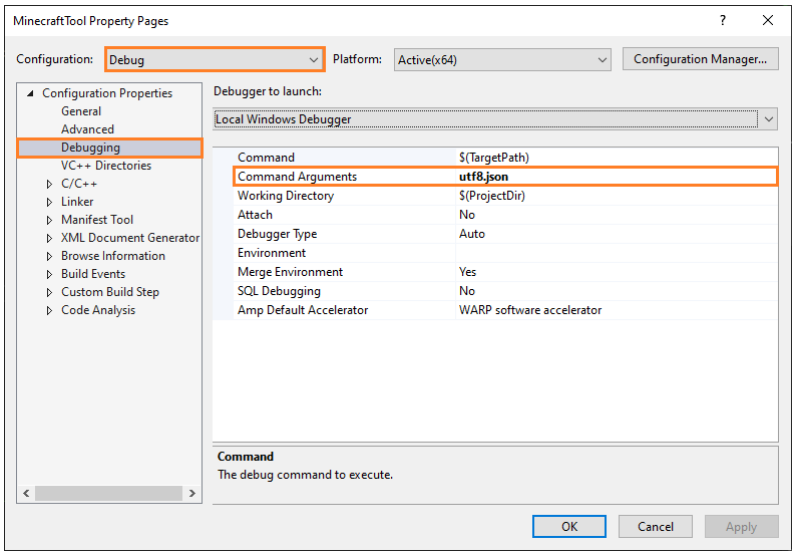


Figure 1.14: Command line arguments

We update the program with to use the `wmain` method (**line 10**). We can create the `ifstream` object with the element at position 1 in the `argv` array (the first element is the name of the executable itself).

Notable is that there is always at least one element in the argv array, namely the complete path of the executable.

```
1  #include <direct.h> // _getwcwd
2  #include <iostream>
3  #include <fstream>
4  #include <windows.h>
5
6  #include "rapidjson.h"
7  #include "istreamwrapper.h"
8  #include "document.h"
9
10 int wmain(int argc, wchar_t* argv[], wchar_t* envp[])
11 {
12   using rapidjson::Document;
13   Document jsonDoc;
14
15   SetConsoleOutputCP(CP_UTF8);
16
17   // Enable buffering to prevent VS from chopping
18   // up UTF-8 byte sequences
19   setvbuf(stdout, nullptr, _IOFBF, 1000);
20   std::cout << "number of arguments :" << argc << std::endl;
21   // argv[0] is the name of the executable
22   // argv[1] is the filename to read.
23   if (argc >= 2) {
24     std::wcout << "Executable itself : " << argv[0] << std::endl;
25     std::wcout << "File to read : " << argv[1] << std::endl;
26
27
28     if (std::ifstream is{ argv[1] })
```

```
29      {
30        rapidjson::IStreamWrapper isw{ is };
31        jsonDoc.ParseStream(isw);
32
33        using rapidjson::Value;
34
35        const Value& valName = jsonDoc["name"];
36        const char* valString = valName.GetString();
37
38        std::cout << "name is : " << valString << std::endl;
39        return 0;
40      }
41      else {
42        return -2;
43      }
44    }
45    else {
46      return -1; // file not found.
47    }
48  }
```

Our executable is still defined in the Path environment variable. Now it should work to open a command line window and use the tool with different files.

To test this I have a directory with 2 files `utf8example1.json` and `utf8example2.json`.

```
{
"name":"Ω"
}
```

Listing 1.1: utf8example1.json

```
{
"name":"Geen utf"
}
```

Listing 1.2: utf8example2.json

With these two files present in the directory we can try out the minecrafttool to see if it works:

```
C:\Data\tooldevelopment\chapter1files>dir
 Volume in drive C is Windows
 Volume Serial Number is 7ED0-607C

 Directory of C:\Data\tooldevelopment\chapter1files

02/02/2021  09:20    <DIR>          .
02/02/2021  09:20    <DIR>          ..
02/02/2021  18:08                23 utf8exampl2.json
01/02/2021  10:55                18 utf8example.json
               2 File(s)             41 bytes
               2 Dir(s)  117,200,912,384 bytes free

C:\Data\tooldevelopment\chapter1files>minecrafttool utf8example1.json
C:\Data\tooldevelopment\chapter1files
number of arguments :2
Executable itself : minecrafttool
File to read : utf8example1.json
name is : Ω

C:\Data\tooldevelopment\chapter1files>minecrafttool utf8example2.json
C:\Data\tooldevelopment\chapter1files
number of arguments :2
Executable itself : minecrafttool
File to read : utf8example2.json
name is : Geen utf
```

Our program will still work if the file that is entered is a relative file or an absolute file. The `ifstream` object handles this detail for us!

Writing a (utf8) text file with streams

The following program tries to output unicode characters that can be used to create boxes.

```
#include <iostream>
#include <fstream>
#include <locale>

int wmain(int argc, wchar_t* argv[], wchar_t* envp[])
{
  std::wstring boxTop = L"   ";
  std::wstring boxMid = L"     ";
  std::wstring boxBot = L"   ";

  std::wofstream fs;
  fs.open("utf8testwrite.txt", std::ios::out);
  fs.imbue(std::locale("en_US.UTF8"));
  fs << boxTop << std::endl;
  fs << boxMid << std::endl;
  fs << boxBot << std::endl;
  fs.close();
}
```

Line 1-9,18 : To start we just create some unicode string with special characters, to make sure that we are interpreting the situation correctly.

Line 11,12 : Next step is to create a wide character output stream and open it for writing.

Line 3,13 : The next step is to `imbue` the correct locale into the output stream. The keyword `en_US.UTF8` ensures that UTF8 output will be written. We need to include `<locale>` for this to work.

Line 14-17 : Finally, we write some output and close the stream. Closing the stream is not absolutely necessary because the `fs` object is declared on the stack and when it is deleted, it will **close** the associated output stream.

The unicode codepoint for the characters is:

1. : code 0x2550
2. : code 0x2551
3. : code 0x2554
4. : code 0x2557
5. : code 0x255A
6. : code 0x255D

After writing the file the following binary content is available.

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000   E2 95 94 E2 95 90 E2 95 90 E2 95 90 E2 95 90 E2  â•”â•.â•.â•.â•.â
00000010   95 97 0D 0A E2 95 91 20 20 20 20 E2 95 91 0D 0A  •—..â•‘    â•‘..
00000020   E2 95 9A E2 95 90 E2 95 90 E2 95 90 E2 95 90 E2  â•šâ•.â•.â•.â•.â
00000030   95 9D 0D 0A                                      •...
```

Let's check the first character to see if it is indeed utf8 encoded. The binary pattern for `E2 95 94` is:
`1110 0010 1001 0101 1001 0100`

The bit pattern starts with three 1 values, which in utf8 means that the character is encoded with 3 bytes. The two following byte will then start with the pattern 10. Let's clarify this pattern:

111 0 0010 10 01 0101 10 01 0100

The bit pattern that contains the unicode character is then formed by the green bits:

0 0010 0101 0101 0100 = 0x2554 = OK!

Writing a (utf8) text file with printf

The `printf` family of functions is pretty old but it still allows you to efficiently format text that for example contains a lot of numbers:

```
#include <iostream>

int wmain(int argc, wchar_t* argv[], wchar_t* envp[])
{
  std::wstring boxTop = L"   ";
  std::wstring boxMid = L"     ";
  std::wstring boxBot = L"   ";

  FILE* pOFile = nullptr;
  _wfopen_s(&pOFile, L"utf8testwrite2.txt", L"w+,ccs=UTF-8");
  if (pOFile != nullptr) {
    // it was possible to create the file for writing.
    fwprintf_s(pOFile, L"%s\n", boxTop.c_str());
    fwprintf_s(pOFile, L"%s\n", boxMid.c_str());
    fwprintf_s(pOFile, L"%s\n", boxBot.c_str());
    fclose(pOFile);
  }
}
```

line 1-7,19: We start with the same Unicode strings as the previous example.

line 9 : To use the `printf` family of functions we need to open a `FILE` object.

line 10 : To open a `FILE` object the `_wfopen_s` function is used. The first parameter is a pointer to the `FILE*`. The second parameter is the (relative) location of the file. The last parameter is the writing mode. This contains two parts:

- **w+** : erase the file or create a new one if it does not exist.
- **ccs=UTF-8** : encode the characters with UTF-8.

line 11,12,17 : If the file was opened succesfully (it can always happen that another application has a lock on the file), then we can start writing.

line 13 : The `fwprintf_s` method writes to a file (`f`) with wide characters (`w`) in a safe way (`_s`), which means it checks for buffer overruns. The parameters of this function are:

- first parameter: the `FILE` pointer.
- second parameter: the format string. This string can contains ordinary characters but also format commands (starting with %). In this case the %s gives the command to insert a string at this location.
- third parameter: Every format command needs to be matched by an additional parameter. In this case a `wchar_t*` needs to be inserted there.

line 14,15 : We repeat this for the two other unicode strings.

line 16 : Finally we close the file pointer.

After executing the program, the following output is generate in the file:

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000   EF BB BF E2 95 94 E2 95 90 E2 95 90 E2 95 90 E2  ï»¿â•”â•.â•.â•.â
00000010   95 90 E2 95 97 0D 0A E2 95 91 20 20 20 20 E2 95  •.â•—..â•‘    â•
00000020   91 0D 0A E2 95 9A E2 95 90 E2 95 90 E2 95 90 E2  ‘..â•šâ•.â•.â•.â
00000030   95 90 E2 95 9D 0D 0A                             •.â•...
```

What do you conclude? What is different from the previous output?

Writing a (utf8) text file

To write to a text file we again open a `FILE*`, but now we specify that we we want to open the file for writing, and that we want to output a UTF8 file:

```
_wfopen_s(&pOFile, L"scene.obj", L"w,ccs=UTF-8");
```

The first method to write to the file is with the `fwrite` function, as you can see on **line 18**.

However this method is efficient but a bit verbose. As an old school programmer myself I am more partial to the printf family of methods, as demonstrated on line 21 to line 53.

The `fwprintf_s` method has the advantage that it converts numbers to strings without hassle, is readable, and for a command line tool the performance is certainly adequate.

```
1  #include <direct.h> // _getwcwd
2  #include <iostream>
3  #include <fstream>
4  #include <windows.h>
5
6  #include "rapidjson.h"
7  #include "document.h"
8  #include "stream.h"
9  #include "filereadstream.h"
10
11 int wmain(int argc, wchar_t* argv[], wchar_t* envp[])
12 {
13
14   FILE* pOFile = nullptr;
15   _wfopen_s(&pOFile, L"scene.obj", L"w+,ccs=UTF-8");
16   if (pOFile != nullptr) {
17     // it was possible to create the file for writing.
18     const wchar_t* text = L"# is the symbol for partial derivative.\n";
19     fwrite(text, wcslen(text) * sizeof(wchar_t), 1, pOFile);
20
21     fwprintf_s(pOFile, L"v %.4f %.4f %.4f\n", 0.0, 0.0, 0.0f);
22     fwprintf_s(pOFile, L"v %.4f %.4f %.4f\n", 0.0, 0.0, 1.0f);
23     fwprintf_s(pOFile, L"v %.4f %.4f %.4f\n", 0.0, 1.0, 0.0f);
24     fwprintf_s(pOFile, L"v %.4f %.4f %.4f\n", 0.0, 1.0, 1.0f);
25     fwprintf_s(pOFile, L"v %.4f %.4f %.4f\n", 1.0, 0.0, 0.0f);
26     fwprintf_s(pOFile, L"v %.4f %.4f %.4f\n", 1.0, 0.0, 1.0f);
27     fwprintf_s(pOFile, L"v %.4f %.4f %.4f\n", 1.0, 1.0, 0.0f);
28     fwprintf_s(pOFile, L"v %.4f %.4f %.4f\n", 1.0, 1.0, 1.0f);
29
30     fwprintf_s(pOFile, L"vn %.4f %.4f %.4f\n", 0.0f, 0.0f, 1.0f);
31     fwprintf_s(pOFile, L"vn %.4f %.4f %.4f\n", 0.0f, 0.0f, -1.0f);
32     fwprintf_s(pOFile, L"vn %.4f %.4f %.4f\n", 0.0f, 1.0f, 0.0f);
33     fwprintf_s(pOFile, L"vn %.4f %.4f %.4f\n", 0.0f, -1.0f, 0.0f);
34     fwprintf_s(pOFile, L"vn %.4f %.4f %.4f\n", 1.0f, 0.0f, 0.0f);
35     fwprintf_s(pOFile, L"vn %.4f %.4f %.4f\n", -1.0f, 0.0f, 0.0f);
36
37     fwprintf_s(pOFile, L"f %d//%d %d//%d %d//%d\n", 1, 2, 7, 2, 5, 2);
38     fwprintf_s(pOFile, L"f %d//%d %d//%d %d//%d\n", 1, 2, 3, 2, 7, 2);
39
40     fwprintf_s(pOFile, L"f %d//%d %d//%d %d//%d\n", 1, 6, 4, 6, 3, 6);
41     fwprintf_s(pOFile, L"f %d//%d %d//%d %d//%d\n", 1, 6, 2, 6, 4, 6);
42
43     fwprintf_s(pOFile, L"f %d//%d %d//%d %d//%d\n", 3, 3, 8, 3, 7, 3);
44     fwprintf_s(pOFile, L"f %d//%d %d//%d %d//%d\n", 3, 3, 4, 3, 8, 3);
```

```
45
46      fwprintf_s(pOFile, L"f %d//%d %d//%d %d//%d\n", 5, 5, 7, 5, 8, 5);
47      fwprintf_s(pOFile, L"f %d//%d %d//%d %d//%d\n", 5, 5, 8, 5, 6, 5);
48
49      fwprintf_s(pOFile, L"f %d//%d %d//%d %d//%d\n", 1, 4, 5, 4, 6, 4);
50      fwprintf_s(pOFile, L"f %d//%d %d//%d %d//%d\n", 1, 4, 6, 4, 2, 4);
51
52      fwprintf_s(pOFile, L"f %d//%d %d//%d %d//%d\n", 2, 1, 6, 1, 8, 1);
53      fwprintf_s(pOFile, L"f %d//%d %d//%d %d//%d\n", 2, 1, 8, 1, 4, 1);
54
55      fclose(pOFile);
56      return 0;
57    }
58    else {
59      return -1;
60    }
61  }
```

Listing 1.3: write utf8 file

Finally, we need to close the file (**line 55**). Not closing files is a major source of problems when dealing with input/output in an application.

If you run the tool from within Visual Studio, a `scene.obj` file will be created in the project folder. Use the "Show all files" button in the solution explorer to see this file. If you doubleclick on it in Visual Studio, you will see the cube visualized:

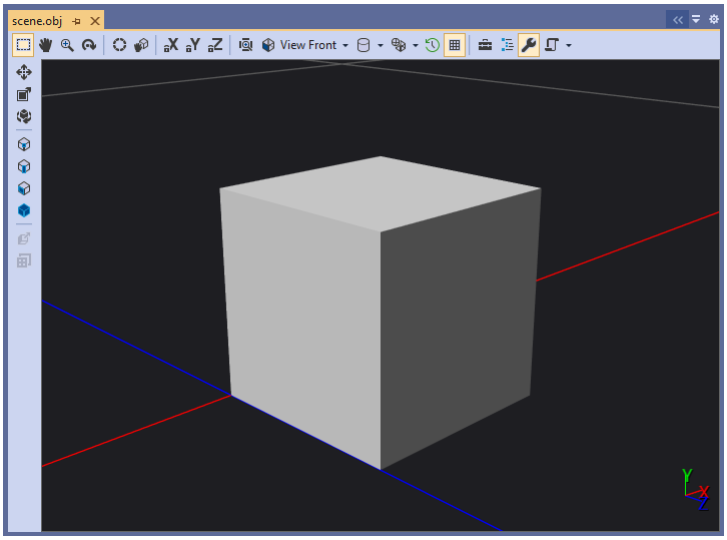


Figure 1.15: The resulting 3d cube from the obj file

1.4 Exercises

It is necessary to finish the exercises of this week because we will build upon them in the coming weeks.

Exercise : complex command line arguments

Command line arguments are typically more complicated and also allow the user to specify where the output of the file should be.

Change the program to make following use cases possible:

```
minecrafttool -i scene1.json
minecrafttool -i scene2.json -o minecraft.obj
```

In the first case only the input file is given and the output should be the `scene1.obj` file in the same folder. For now it is sufficient to output a simple obj file (e.g. just a cube), as shown on page 36.

Exercise : transform voxel scene

Given is a voxel scene in json which is a regular point cloud. The voxel scene has opaque (non transparent) and transparent blocks and the possible blocks are: dirt, stone, woord and glass.

The json file is somewhat optimized to be as compact as possible with text files and looks like this:

```
[
{
    "layer":"dirt",
    "opaque":true,
    "positions":[
        [-2,-2,0],[-1,-2,0],[0,-2,0],[1,-2,0],[2,-2,0]
          ,
        [-2,-1,0],[-1,-1,0],[0,-1,0],[1,-1,0],[2,-1,0]
          ,
        [-2,0,0],[-1,0,0],[0,0,0],[1,0,0],[2,0,0],
        [-2,1,0],[-1,1,0],[0,1,0],[1,1,0],[2,1,0],
        [-2,2,0],[-1,2,0],[0,2,0],[1,2,0],[2,2,0]
    ]
},
{
```

```
14      "layer":"stone",
15      "opaque":true,
16      "positions":[
17          [-1,-1,1], [-1,0,1],[-1,1,1],[0,1,1],[1,-1,1],
                [1,0,1],[1,1,1],
18          [-1,-1,2], [-1,0,2],[-1,1,2],[0,1,2],[1,-1,2],
                [1,0,2],[1,1,2],
19          [-1,-1,3], [-1,0,3],[-1,1,3],[0,1,3],[1,-1,3],
                [1,0,3],[1,1,3]
20      ]
21  },
22  {
23      "layer":"wood",
24      "opaque":true,
25      "positions":[
26              [-1,-1,4], [-1,0,4],[-1,1,4],
27              [0,-1,4] ,[0,0,4]  ,[0,1,4],
28              [1,-1,4] ,[1,0,4]  ,[1,1,4]
29      ]
30  },
31  {
32      "layer":"glass",
33      "opaque":false,
34      "positions":[[0,-1,3]]
35  }
36  ]
```

Take the following into account when parsing:

- The root element is an **array**, its children are json objects.
- The positions member is an array of arrays. The elements of the positions member is the position of the block in the world.

Transform the scene into the obj-format, you can reuse the code that was used to generate a single block. The resulting obj file does not have to be optimized, but all blocks should use the correct material.

Exercise: optimize the scene

One complete cube for each block is a bit excessive. Suppress all the faces that will not be visible in the scene. Take into account that the glass layer is opaque.

For this to work you will need a way to check if a block has a neighbour and on which side this neighbour is situated.

Chapter 2

Build tools introduction

2.1 Goal

It is often the case that a project can only be build on the laptop or desktop of the original developer, due to a combination of installed libraries, settings, ... This is obviously a situation we want to avoid in a professional environment.

Within the philosophy of build systems and continuous integration it is also a deadly sin if a developer has to perform a manual intervention (e.g. copying a file, change a setting, ...) to ensure that the build runs correctly. In principle, a build should be launched by invoking one single command.

It is important then to be able to use build tools. In this session we will see a build tool which is used a lot in the industry, namely CMake (`https://cmake.org/`)

The goal of this session is to build the exercise of last week with a professional build tool in such a way that it can easily be rebuilt/distributed. This knowledge is also useful (and in a company absolutely necessary) to build applications and games.

2.2 Installation

Download `cmake-3.22.2-win64-x64.msi` from `https://cmake.org/` and execute it.

It is important to select "Add CMake to the system PATH for all users".

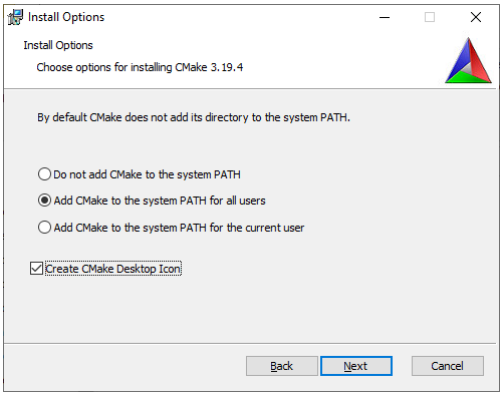


Figure 2.1: CMake installation

Press Next and choose the standard options after that.

2.3 A first example

We start with a very simple C++ file that we want to build:

```
1 #include <iostream>
2
3 int main()
4 {
5   std::cout<<"Hello World!\n";
6 }
```

To test this out, put this file into an empty directory (e.g. `exercise1`), and name this file `exercise1.cpp`.

The next step is to tell CMake what to do with this file. A very basic project definition is demonstrated here:

```
1 cmake_minimum_required(VERSION 3.10)
2
3 project(Exercise1)
4
5 add_executable(Exercise1 exercise1.cpp)
```

The first line defines the minimum required version of CMake itself.

On **line 3** the name of the project is defined.

Finally, on line 5 of this script the name of the executable and the name of the cpp file that contains the `main` function is defined.

Save this file as `CMakeLists.txt`, this is a convention that CMake uses for these types of files, and ensures that the project configuration gets automatically loaded by CMake.

To summarize you should now have a directory `exercise1` with just two files:

`exercise1‣exercise1.cpp`
`exercise1‣CMakeLists.txt`

Now we just need to invoke CMake to create a build system for this c++ program. Open a command prompt and navigate to the appropriate directory:

```
1 c:\Data\tooldevelopment\chapter2files\exercise1>cmake .
2 -- Selecting Windows SDK version 10.0.18362.0 to target Windows 10.0.19041.
3 -- Configuring done
4 -- Generating done
5 -- Build files have been written to:
      C:/Data/tooldevelopment/chapter2files/exercise1
6
7
8 c:\Data\tooldevelopment\chapter2files\exercise1>
```

On **line 1** we invoke the cmake command on the current directory (a current directory on many operatings systems is denoted by the '.'-character).

Check what the CMake system has generated for you:

```
1  c:\Data\tooldevelopment\chapter2files\oefening1>dir
2   Volume in drive C is Windows
3   Volume Serial Number is 7ED0-607C
4
5   Directory of c:\Data\tooldevelopment\chapter2files\exercise1
6
7  15/02/2021  18:06    <DIR>          .
8  15/02/2021  18:06    <DIR>          ..
9  15/02/2021  18:05            17,510 ALL_BUILD.vcxproj
10 15/02/2021  17:28               302 ALL_BUILD.vcxproj.filters
11 15/02/2021  18:05            14,018 CMakeCache.txt
12 15/02/2021  18:05    <DIR>          CMakeFiles
13 15/02/2021  18:05               101 CMakeLists.txt
14 15/02/2021  17:28             1,469 cmake_install.cmake
15 15/02/2021  17:24                70 exercise1.cpp
16 15/02/2021  18:05             3,118 Exercise1.sln
17 15/02/2021  18:05            27,093 exercise1.vcxproj
18 15/02/2021  18:05               612 exercise1.vcxproj.filters
19 15/02/2021  18:05            17,150 ZERO_CHECK.vcxproj
20 15/02/2021  17:28               539 ZERO_CHECK.vcxproj.filters
21              11 File(s)         81,982 bytes
```

```
               5 Dir(s)  119,901,257,728 bytes free
```

Visual studio was found and so an appropriate `solution` file and `vcxproj` file was generated.

To **build** the example we can now simply invoke the cmake program with the build option:

```
c:\Data\tooldevelopment\chapter2files\exercise1>cmake --build .
Microsoft (R) Build Engine version 16.8.3+39993bd9d for .NET Framework
Copyright (C) Microsoft Corporation. All rights reserved.

  Checking Build System
  Building Custom Rule C:/Data/tooldevelopment/chapter2files/exercise1/CMakeLists
      .txt
  exercise1.cpp
  exercise1.vcxproj -> C:\Data\tooldevelopment\chapter2files\exercise1\Debug\
      exercise1.exe

c:\Data\tooldevelopment\chapter2files\exercise1>
```

After executing this command a new folder called Debug should be available and it should contain the `Exercise1.exe` file.

We can now navigate to this directory and execute the program:

```
c:\Data\tooldevelopment\chapter2files\exercise1>cd Debug

c:\Data\tooldevelopment\chapter2files\exercise1\Debug>dir
 Volume in drive C is Windows
 Volume Serial Number is 7ED0-607C

 Directory of c:\Data\tooldevelopment\chapter2files\exercise1\Debug

15/02/2021  18:07    <DIR>          .
15/02/2021  18:07    <DIR>          ..
15/02/2021  18:07            57,856 exercise1.exe
15/02/2021  18:07           608,416 exercise1.ilk
15/02/2021  18:07         1,003,520 exercise1.pdb
15/02/2021  17:29            57,856 oef1.exe
15/02/2021  17:29           608,288 oef1.ilk
15/02/2021  17:29         1,003,520 oef1.pdb
               6 File(s)      3,339,456 bytes
               2 Dir(s)  119,903,113,216 bytes free

c:\Data\tooldevelopment\chapter2files\exercise1\Debug>exercise1
Hello World!
```

2.3.1 Out of source build

Making a build in this manner works, but it is not ideal because the build files are put next to the source files, and this makes it more difficult to for example delete the build when you want to distribute only your code.

Luckily we can create the necessary build files ”out of source”, we just need to create a directory and supply the initial cmake command with the location of the top level directory for our project. For example when we use CMake in Visual Studio, visual studio will put the build in the ”out” folder.

```
 Directory of c:\Data\tooldevelopment\chapter2files\exercise1

12/10/2021  05:11    <DIR>          .
12/10/2021  05:11    <DIR>          ..
15/02/2021  19:05               101 CMakeLists.txt
25/03/2021  15:36                92 exercise1.cpp
               2 File(s)            193 bytes
               2 Dir(s)  30,051,106,816 bytes free

c:\Data\tooldevelopment\chapter2files\exercise1>mkdir build

c:\Data\tooldevelopment\chapter2files\exercise1>cd build

c:\Data\tooldevelopment\chapter2files\exercise1\build>cmake ..
-- Building for: Visual Studio 16 2019
-- Selecting Windows SDK version 10.0.18362.0 to target Windows 10.0.19043.
-- The C compiler identification is MSVC 19.28.29914.0
-- The CXX compiler identification is MSVC 19.28.29914.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: C:/Program Files (x86)/Microsoft Visual
    Studio/2019/Enterprise/VC/Tools/MSVC/14.28.29910/bin/Hostx64/x64/cl.exe -
    skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual
    Studio/2019/Enterprise/VC/Tools/MSVC/14.28.29910/bin/Hostx64/x64/cl.exe -
    skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to:
    C:/Data/tooldevelopment/chapter2files/exercise1/build

c:\Data\tooldevelopment\chapter2files\exercise1\build>
```

To actually build the project we now need to execute the `cmake --build .` command from this **build** folder.

2.3.2 Opening the cmake project directly

There is support for CMake in Visual Studio, and it is possible to open a CMake project directly:

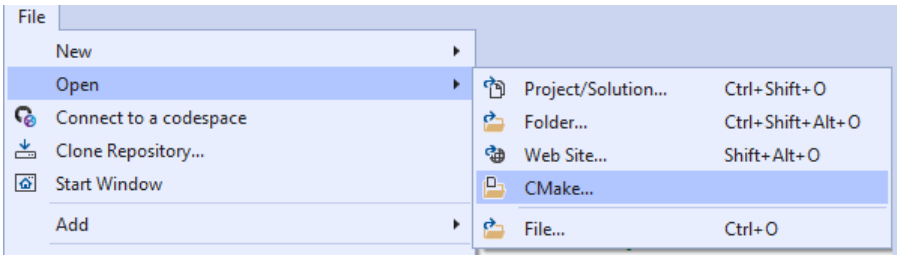


Figure 2.2: Open CMake Project in Visual Studio

The default view of a CMake project is a file view (but we can change this):

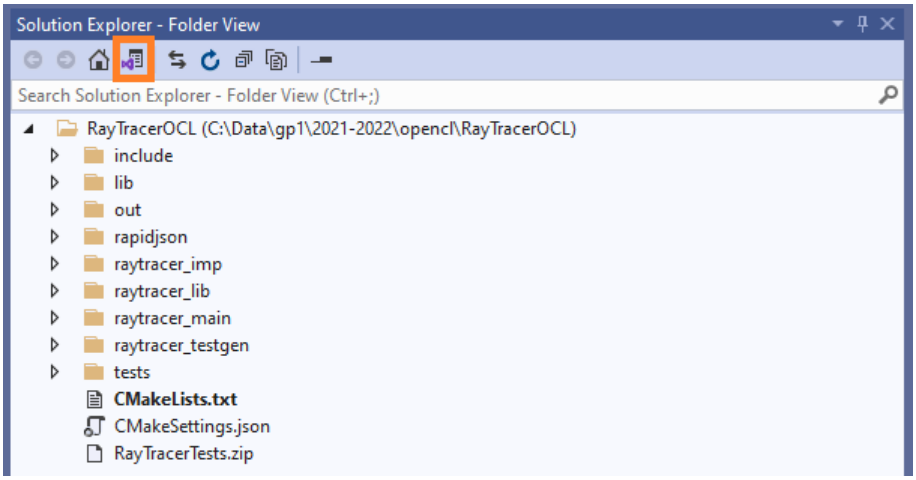


Figure 2.3: File view of the CMake project

To change the project view, click the button next to the home button in the solution explorer and doubleclick on the CMake Targets View:

This changes the view in visual studio to a more practical view and also shows which projects build an executable and which projects build library artefacts:

Executable projects can be selected and run in the usual manner:

Remark that there is an **install** target at the bottom which will create an installer for the project, this topic will be discussed later on.

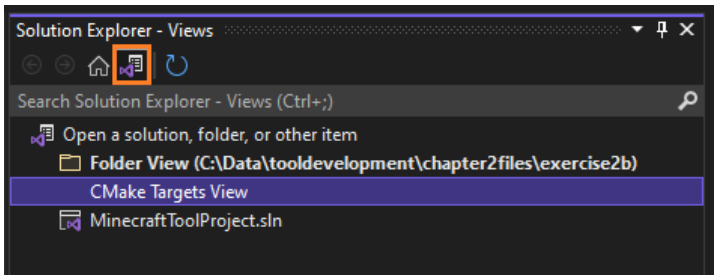


Figure 2.4: Change the project view

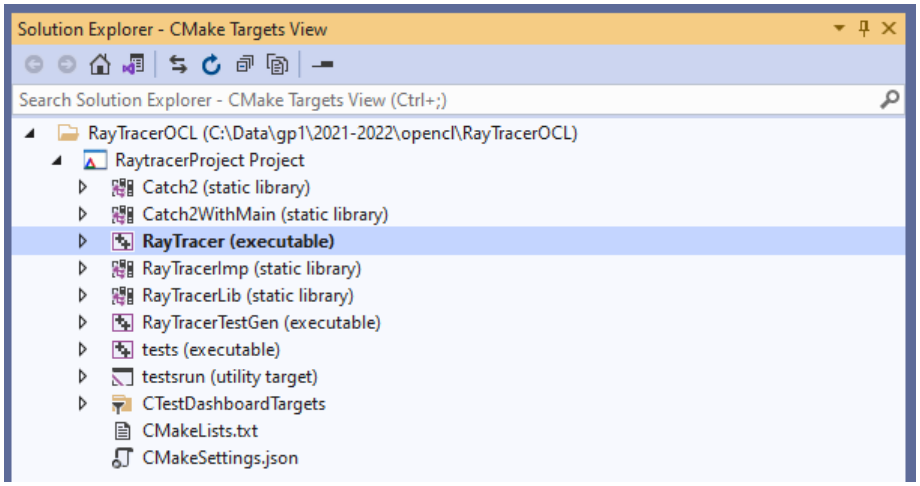


Figure 2.5: Change the project view

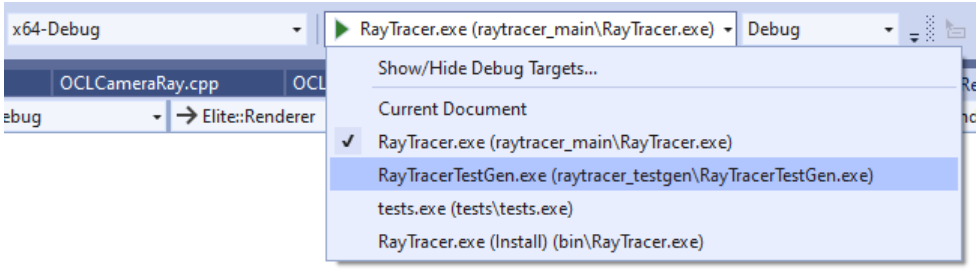


Figure 2.6: Select the executable or target to run

2.4 CMake as a programming language

CMake is actually a programming language, specialized in the domain of building C or C++ applications. Off course since the most important functionality is dealing with strings, there is a lot of syntax to make strings, lists of strings and easy ways to define and store paths.

2.4.1 Set

The set command is used to define a variable. This variable will be visible in the current CMakeLists.txt file and in the child CMakeLists.txt that are referenced in this file (see later).

```
1 set(Author "Koen Samyn")
2 set(CourseTitle "Tool Development by ${Author}")
```

The **CourseTitle** variable uses what is called a **string interpolation**. The contents of the **Author** variable is inserted at the position in the string where `${Author}` is placed.

There are predefined variables that are available in the CMakeLists.txt file which are the following:

- `CMAKE_CURRENT_SOURCE_DIR`: the exact location of the current CMakeLists.txt file. This is very convenient for defining include directories for example.
- `CMAKE_PROJECT_NAME`: the name of the current version.
- `CMAKE_PROJECT_VERSION`: the major version of the project.

2.4.2 Scope

Sometimes we want to set a variable in a build script and make this variable available in the parent build script. Let us first describe the situation, where a project is defined that is made up of several subprojects:

```
project‣CMakeLists.txt
project‣lib‣vld‣CMakeLists.txt
project‣lib‣sdl=2.0.9‣CMakeLists.txt
```

In this case the project defines a CMakeLists.txt file which will add several subprojects with the **add_subdirectory** command.
The **target_include_directories** command uses these variables to set the include directories for the RayTracer subproject (not shown in the listing):

```
cmake_minimum_required(VERSION 3.10)

project(RaytracerProject)

add_subdirectory(lib/vld)
add_subdirectory(lib/sdl2-2.0.9)

...

target_include_directories (RayTracer PUBLIC
    ${SDLIncludeDir}
    ${VLDIncludeDir}
)
```

In this script we depend on two variables that should be set in the CMakeLists.txt file in the **vld** subdirectory and the **sdl** subdirectory, namely the `SDLIncludeDir` and the `VLDIncludeDir` variable.

In the CMakeLists.txt file that is defined in the vld subdirectory we can set the `VLDIncludeDir` variable. The important keyword here is **PARENT_SCOPE** which will make this variable available in the parent script. The string interpolation functionality also makes it easy to compose directory paths:

```
add_library(VLD INTERFACE)
set(VLDIncludeDir "${CMAKE_CURRENT_SOURCE_DIR}/include" PARENT_SCOPE)
```

A similar script is inside the CMakeLists.txt file in the sdl subdirectory:

```
add_library(SDL INTERFACE)
set(SDLIncludeDir "${CMAKE_CURRENT_SOURCE_DIR}/include" PARENT_SCOPE)
```

2.5 Adding an include directory

In the previous chapter we developed a command line tool with the rapid json library.

Now we look at using **cmake** to build this project. We again start with an empty directory, call this directory `exercise2`.

The directory should contain:

```
exercise2‣MinecraftTool.cpp
exercise2‣rapidjson
```

Alternatively you can download the `exercise2.zip` file from Leho.

The `rapidjson` directory contains the header files of the **rapidjson** library.

Configuring the rapidjson library for CMake

In the directory `rapidjson` we need to add the following file again with the name `CMakeLists.txt` :

```
1 add_library(rapidjson INTERFACE)
2 set(JSONIncludeDir "${CMAKE_CURRENT_SOURCE_DIR}" PARENT_SCOPE)
```

The `INTERFACE` keyword indicates that this is a header only library and that there will be no output for this library (the output could for example be a `lib` file or a `dll` file).

Configuring the main project for CMake

Go back to the `exercise2` directory and create a `CMakeLists.txt` file here as well:

```
1  cmake_minimum_required(VERSION 3.10)
2
3  project(MinecraftTool)
4  add_subdirectory(rapidjson)
5  add_executable(MinecraftTool MinecraftTool.cpp)
6
7  message(STATUS "${PROJECT_SOURCE_DIR}")
8
9  target_include_directories(MinecraftTool PUBLIC
10     "${PROJECT_BINARY_DIR}"
11     "${JSONIncludeDir}"
12     )
```

Line 1,3 and 5 are the same as in the previous exercise.

On **line 4** we tell CMake to also look in the rapidjson directory. Because this is the main project you can add as many subdirectories as you want.

On **line 7** we check if the source directory is correctly set.

Line 9 then defines the includes that are needed to build the project. **Line 10** is not strictly necessary for this project, but is important when a library produces a dll or lib file as input for the entire build process.

Line 11 defines the root directory for the includes. This makes it possible to use the following include statements in `MinecraftTool.cpp` :

```
1 #include "rapidjson.h"
2 #include "document.h"
3 #include "stream.h"
4 #include "filereadstream.h"
```

Building exercise 2

To build exercise 2 we will again create an **out of source** build:

```
c:\Data\tooldevelopment\chapter2files\exercise2>dir
 Volume in drive C is Windows
 Volume Serial Number is 7ED0-607C

 Directory of c:\Data\tooldevelopment\chapter2files\exercise2

17/02/2022  10:55    <DIR>          .
17/02/2022  10:55    <DIR>          ..
17/02/2022  10:47               572 CMakeLists.txt
15/02/2021  20:20             1,065 License.txt
15/02/2021  20:00             3,836 MinecraftTool.cpp
17/02/2022  10:44    <DIR>          rapidjson
               3 File(s)          5,473 bytes
```

```
               3 Dir(s)  182,587,613,184 bytes free

c:\Data\tooldevelopment\chapter2files\exercise2>mkdir build

c:\Data\tooldevelopment\chapter2files\exercise2>cd build

c:\Data\tooldevelopment\chapter2files\exercise2\build>cmake ..
-- Building for: Visual Studio 17 2022
-- Selecting Windows SDK version 10.0.19041.0 to target Windows 10.0.19044.
-- The C compiler identification is MSVC 19.30.30709.0
-- The CXX compiler identification is MSVC 19.30.30709.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
...
-- include directory: C:/Data/tooldevelopment/chapter2files/exercise2
-- Configuring done
-- Generating done
-- Build files have been written to:
    C:/Data/tooldevelopment/chapter2files/exercise2/build

c:\Data\tooldevelopment\chapter2files\exercise2\build>cmake --build .
Microsoft (R) Build Engine version 17.0.0+c9eb9dd64 for .NET Framework
Copyright (C) Microsoft Corporation. All rights reserved.

  Checking Build System
  Building Custom Rule C:/Data/tooldevelopment/chapter2files/exercise2/CMakeLists
      .txt
  MinecraftTool.cpp
  MinecraftTool.vcxproj -> C:\Data\tooldevelopment\chapter2files\exercise2\build\
      Debug\MinecraftTool.exe
  Building Custom Rule C:/Data/tooldevelopment/chapter2files/exercise2/CMakeLists
      .txt

c:\Data\tooldevelopment\chapter2files\exercise2\build>
```

The steps are:

1. Go into the root folder of `exercise2`.
2. Create a **build** directory
3. Go into the build directory with the cd command.
4. Give the command: `cmake ...` The .. means that the cmake command will one directory higher and use the `CMakeLists.txt` at that location as the **root** element of the build project.
5. Once this command is done all the configuration is stored into the current directory. So now we just need to give the command `cmake --build .` to build the executable.

2.6 Installeable version

Creating software is fun, but it even more fun if other people can use your software or play your game. In this section we first create a zip version of the software, and then we create an installer version.

Creating a zip file

Update the `CMakeLists.txt` in the `exercise2` directory:

```
1  cmake_minimum_required(VERSION 3.10)
2
3  project(MinecraftTool)
4  add_subdirectory(rapidjson)
5  add_executable(MinecraftTool MinecraftTool.cpp)
6
7  message(STATUS "include directory: ${PROJECT_SOURCE_DIR}")
8  target_include_directories(MinecraftTool PUBLIC
9      "${PROJECT_BINARY_DIR}"
10     "${JSONIncludeDir}"
11     )
12
13 install(TARGETS MinecraftTool DESTINATION bin)
14
15 include(InstallRequiredSystemLibraries)
16 set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
17 set(CPACK_PACKAGE_VERSION_MAJOR "0")
18 set(CPACK_PACKAGE_VERSION_MINOR "1")
19 include(CPack)
```

The installer needs to know what files need to be included in the final installation, so on **line 13** we define that the MinecraftTool **target** is one of the elements that need to be included in the **binary** distribution. Typically for a game we only distribute binaries, that is to say we do not typically include the source files for a game project, unless it is open source.

Line 13 makes sure that all the necessary system libraries are included in the distribution. This could include the visual studio runtimes for example.

Line 16 references a `License.txt` that you do not yet have in your `exercise2` directory, so download it from `Leho` and place it in the `exercise2` directory.

Line 17 and 18 set the version of the software. This is now set to a hardcoded number, but this can integrate with a source control system to automatically for example update the **minor version** number when a new build is created.

Finally, for Cpack, the last statement must be :

```
include(CPack)
```

To create a zip file for the installation we call the **cpack** command with the ZIP option:

```
c:\Data\tooldevelopment\chapter2files\exercise2\build>cpack -G ZIP -C Debug
CPack: Create package using ZIP
CPack: Install projects
CPack: - Install project: MinecraftTool [Debug]
CPack: Create package
CPack: - package: C:/Data/tooldevelopment/chapter2files/exercise2/MinecraftTool-0
    .1.1-win64.zip generated.

c:\Data\tooldevelopment\chapter2files\exercise2\build>
```

This generates a zip file with the MinecraftTool.exe include but also all the dependencies:

Naam	Grootte	Ingep...	Type	Gewijzigd	CRC32
..			File folder		
concrt140.dll	309,128	136,755	Application extension	04/12/2020 12:27	4F30F4AC
MinecraftTool.exe	159,232	42,738	Application	15/02/2021 20:06	33C15CF4
msvcp140.dll	585,096	186,406	Application extension	04/12/2020 12:27	DE413D2D
msvcp140_1.dll	23,944	11,839	Application extension	04/12/2020 12:27	CCDDF18F
msvcp140_2.dll	186,248	80,418	Application extension	04/12/2020 12:27	99275508
msvcp140_atomic_wait.dll	41,352	14,182	Application extension	04/12/2020 12:27	ACB9BA60
msvcp140_codecvt_ids.dll	20,360	10,463	Application extension	04/12/2020 12:27	608DF751
vcruntime140.dll	94,088	50,071	Application extension	04/12/2020 12:27	CDAE8E49
vcruntime140_1.dll	36,744	19,767	Application extension	04/12/2020 12:27	36920A2E

Figure 2.7: Contents of the generated zip file

Installer with NSIS

NSIS is short for Nullsoft Scriptable Install System. You can download NSIS from SourceForge or from Leho under the Tool section.

After installing we can try to run cpack:

```
c:\Data\tooldevelopment\chapter2files\exercise2>cpack -C Debug
CPack: Create package using NSIS
CPack: Install projects
CPack: - Install project: MinecraftTool [Debug]
CPack: Create package
CPack: - package: C:/Data/tooldevelopment/chapter2files/exercise2/MinecraftTool-0
    .1.1-win64.exe generated.

c:\Data\tooldevelopment\chapter2files\exercise2>
```

This generates an installer for you which will install the MinecraftTool on your system. Even better, the installer now also has an **uninstall** option.

Try this installer out, and check if the installer modified the **Path** environment variable.

To add the directory of the installation to the PATH variable on windows we need to change the cmake script:

```
1  cmake_minimum_required(VERSION 3.10)
2
3  project(MinecraftToolProject)
4  add_subdirectory(rapidjson)
5  add_executable(MinecraftTool MinecraftTool.cpp)
6
7  message(STATUS "include directory: ${PROJECT_SOURCE_DIR}")
8  target_include_directories(MinecraftTool PUBLIC
9      "${PROJECT_BINARY_DIR}"
10     "${JSONIncludeDir}"
11     )
12
13 install(TARGETS MinecraftTool DESTINATION bin)
14
15 include(InstallRequiredSystemLibraries)
16 set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
17 set(CPACK_PACKAGE_VERSION_MAJOR "0")
18 set(CPACK_PACKAGE_VERSION_MINOR "1")
19
20 set(CPACK_NSIS_MODIFY_PATH ON)
21 include(CPack)
```

On line 20 an option has been added to change the Path variable during installation.

Because we changed the cmake file, we need to rebuild the project:

```
1  c:\Data\tooldevelopment\chapter2files\exercise2\build>cmake --build .
2
3    MinecraftTool.vcxproj -> C:\Data\tooldevelopment\chapter2files\exercise2\build\
         Debug\MinecraftTool.exe
```

And then we can regenerate the installer:

```
1  c:\Data\tooldevelopment\chapter2files\exercise2\build>cpack -C Debug
2  CPack: Create package using NSIS
3  CPack: Install projects
4  CPack: - Install project: MinecraftTool [Debug]
5  CPack: Create package
6  CPack: - package:
       C:/Data/tooldevelopment/chapter2files/exercise2//build/MinecraftTool-0.1.1
       -win64.exe generated.
7
8  c:\Data\tooldevelopment\chapter2files\exercise2\build>
```

2.7 Project structure

I am still not happy with the build of exercise 2. The main problem right now is that there is source code in the root folder. The **directory structure** of the ideal solution would look like this:

```
Root
├── CMakeLists.txt
├── MinecraftTool
│   ├── CMakeLists.txt : instructions for MinecraftTool subproject
│   └── MinecraftTool.cpp : the main cpp file
└── Rapidjson
    ├── CMakeLists.txt : instructions for the rapidjson subproject
    └── *.h : all the rapidjson files
```

This also corresponds to the way Microsoft Visual Studio create a solution with multiple sub projects. The CMakeLists.txt file would then correspond to the functionality of the main solution file (.sln) and the CMakeLists.txt files in the subfolders correspond to the configuration for the subprojects.

As we will see later, this is also a good principle to make the entire solution more **modular**. Every subdirectory has a purpose and its unique set of instructions on how to build it.

Fortunately this is not a big change. We just need to make an extra CMakeLists.txt file in the MinecraftTool folder that tells CMake what cpp files are needed to build the executable.

MinecraftTool subdirectory

The `MinecraftTool.cpp` file needs to be placed in this folder, and there is a new CMakeLists.txt file which just describes this executable project:

```
1 add_executable(MinecraftTool MinecraftTool.cpp)
```

This line was originally found in the CMakeLists.txt file in the root directory. As an added benefit it is now a bit clearer that MinecraftTool is the name of this executable project.

Rapidjson subdirectory

No changes in this folder.

Root folder

We need to make a small change in the CMakeLists.txt of this folder:

```
cmake_minimum_required(VERSION 3.10)

project(MinecraftToolProject)
add_subdirectory(rapidjson)
add_subdirectory(MinecraftTool)

message(STATUS "include directory: ${PROJECT_SOURCE_DIR}")
target_include_directories(MinecraftTool PUBLIC
    "${JSONIncludeDir}"
    )

install(TARGETS MinecraftTool DESTINATION bin)

include(InstallRequiredSystemLibraries)
set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
set(CPACK_PACKAGE_VERSION_MAJOR "0")
set(CPACK_PACKAGE_VERSION_MINOR "1")
set(CPACK_GENERATOR NSIS)
set(CPACK_NSIS_MODIFY_PATH ON)
include(CPack)
```

Create the out of source build

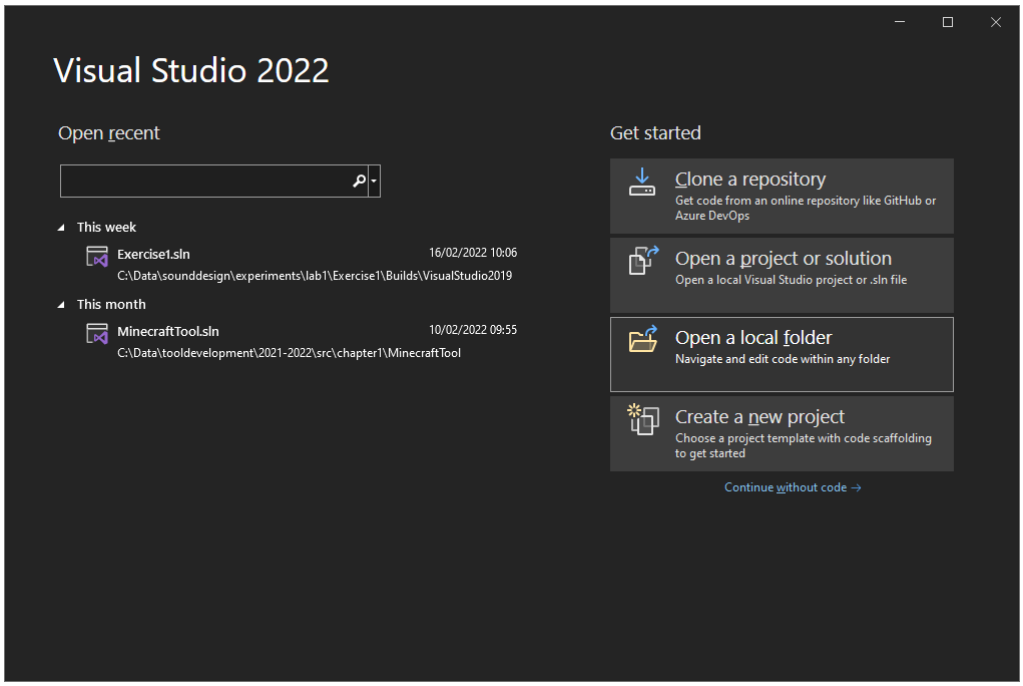
Start in the root of the project and give the following commands:

```
mkdir build
cd build
cmake ..
cmake --build .
```

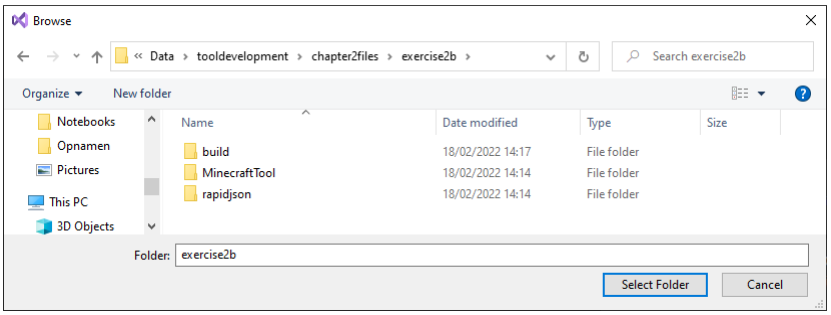
Visual Studio

Before opening the project in Visual Studio **remove** the build folder that we used to build the project via the command line.

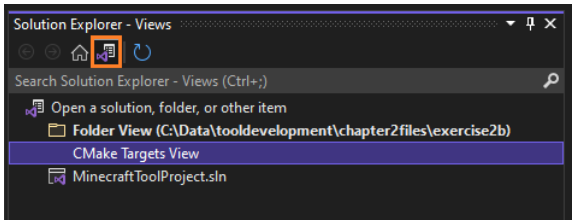
To open the cmake project in Visual Studio, start Visual Studio and choose ”Open a local folder”:



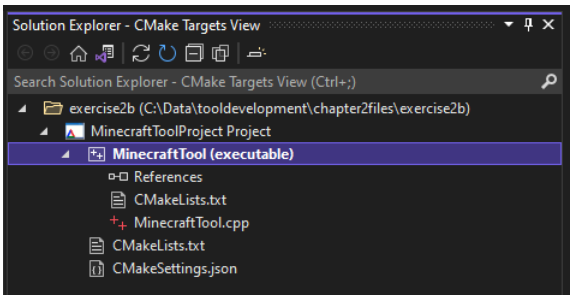
Select the root folder of the project (the MinecraftTool and rapidjson directory should be visible):



Visual Studio will recognize the project as a CMake project. The Solution Explorer will show a file view of the project, which is not what we want. To change this, you need to press the button next to the home button and select the CMake targets view:



You can now use Visual Studio to **build**, **debug** and also **install** the project:



We will discuss how to run the packager (which creates an installable version of the project) later on.

2.8 Static library

The rapidjson library is special because it is **header only**, which makes it easier to compile. This will off course not always be the case, so we study a simple example of a library next.

We start with the following files:

```
Exercise3
├── CMakeLists.txt
├── MatrixTest
│   ├── CMakeLists.txt : MatrixTest executable subproject
│   └── MatrixTest.cpp : the main cpp file
└── MatrixMath
    ├── CMakeLists.txt : MatrixMath library subproject
    ├── FMatrix.h : the header file for the FMatrix class
    └── FMatrix.cpp : the implementation of the FMatrix class
```

In this project `MatrixTest.cpp` contains the main function and we want to compile the header and cpp file in `MatrixMath` as a separate library.

There can be many reasons for keeping code in a separate library. The separation can make it easier to reuse this library, or maybe the library has to be compiled for different operating systems, platforms, ... As we will see later there are also speed of compilation advantages, because only the subprojects where changes were made need to be recompiled.

The `MatrixTest.cpp` file is a test of a simple matrix library (which was used for gameplay programming in the first semester. Note that this file needs to include `FMatrix.h` in the `MatrixMath` sub project.

```
#include "FMatrix.h"

#include <iostream>

void main(){
  std::cout<< "Matrix test" << std::endl;

  FMatrix matrix1(4,4);
  matrix1.Randomize(0,1);
  FMatrix matrix2(4,4);
  matrix2.Randomize(0,1);

  std::cout<< "Matrix 1" << std::endl;
  matrix1.Print();
  std::cout<< "Matrix 2" << std::endl;
  matrix2.Print();

  FMatrix product(4,4);
  matrix1.MatrixMultiply(matrix2,product);
  std::cout << "Result matrix : " << std::endl;
  product.Print();
}
```

We again start with the CMakeLists.txt for the main project in the `MatrixMath` **library** project, then we will look at the MatrixTest **main** project and finally look at the build instructions in the root directory.

CMakeLists.txt for the MatrixMath library directory

To define the library we give this library the following name: **MatrixMath**

In contrast with the previous example we now have something to build, the file `FMatrix.cpp`.

```
1 add_library(MatrixMath FMatrix.cpp)
2 set(MatrixMathIncludeDir "${CMAKE_CURRENT_SOURCE_DIR}" PARENT_SCOPE)
```

On **line 1** we give this library the name **MatrixMath** and add the `cpp` files that need to be compiled.

On **line 2** we record were the headers can be found and store that information in the variable `$MatrixMathIncludeDir`

Create this file and save it as `CMakeLists.txt` in the `matrixmath` directory.

CMakeLists.txt for the MatrixTest project

For the **MatrixTest** project we need to edit the file `MatrixTest/CMakeLists.txt`:

```
1 add_executable(MatrixTest MatrixTest.cpp)
2 target_link_libraries(MatrixTest PUBLIC MatrixMath)
```

Note that we again have to add the source code files to the MatrixTest executable project.

Line 2 is the **important** new statement. This line declares the dependency of the **MatrixText** project on the **MatrixMath** library! In other words the `lib` file produced by the MatrixMath library will be necessary to compile the MatrixTest executable.

CMakeLists.txt for the solution

We now have a different project name: MatrixMathProject, as can be seen in **line 3**.

```
1  cmake_minimum_required(VERSION 3.10)
2
3  project(MatrixMathProject)
4  add_subdirectory(matrixmath)
5  add_subdirectory(matrixtest)
6
7  message(STATUS "include directory: ${PROJECT_SOURCE_DIR}")
8  target_include_directories(MatrixTest PUBLIC
9      "${PROJECT_BINARY_DIR}"
10     "${MatrixMathIncludeDir}"
11     )
12
13 install(TARGETS MatrixMath DESTINATION bin)
14
15 include(InstallRequiredSystemLibraries)
16 set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
17 set(CPACK_PACKAGE_VERSION_MAJOR "0")
18 set(CPACK_PACKAGE_VERSION_MINOR "1")
19
20 set(CPACK_NSIS_MODIFY_PATH ON)
21 include(CPack)
```

On **line 4,5** we add the subdirectory **matrixmath** of this new project, and the **matrixtest** subdirectory. The `CMakeLists.txt` files of both subdirectories will be read and processed to create a build system.

Take note of **line 13** where the **TARGETS** of the **MatrixMath** library are added to the directory where all the files necessary for installation are gathered.

Create the project and build

To create the project files we invoke again the cmake command:

```
c:\Data\tooldevelopment\chapter2files\exercise3\build>cmake ..
-- Selecting Windows SDK version 10.0.18362.0 to target Windows 10.0.19041.
-- include directory: C:/Data/tooldevelopment/chapter2files/exercise3
-- Configuring done
-- Generating done
-- Build files have been written to:
    C:/Data/tooldevelopment/chapter2files/exercise3
```

To build the project we again invoke cmake with the build option:

```
c:\Data\tooldevelopment\chapter2files\exercise3\build>cmake --build .
Microsoft (R) Build Engine version 17.0.0+c9eb9dd64 for .NET Framework
Copyright (C) Microsoft Corporation. All rights reserved.

  MatrixMath.vcxproj -> C:\Data\tooldevelopment\chapter2files\exercise3\build\
      matrixmath\Debug\MatrixMath.lib
  MatrixTest.vcxproj -> C:\Data\tooldevelopment\chapter2files\exercise3\build\
      matrixtest\Debug\MatrixTest.exe

c:\Data\tooldevelopment\chapter2files\exercise3>
```

Notice how a `MatrixMath.lib` file was created by the compiler. This **.lib** file will then be **integrated** into the executable.

This is off course the main difference between a **lib** file and a **dll** file. A lib or library file will be merged or integrated with the executable at **compile time**.

A dll or dynamic link library will be available for the executable at run time. **DirectX** and **NVidia PhysX** are examples of dynamic link libraries.

Now, we can run the compiled program:

```
c:\Data\tooldevelopment\chapter2files\exercise3\build>cd matrixtest/Debug

c:\Data\tooldevelopment\chapter2files\exercise3\build\matrixtest\Debug>MatrixTest
Matrix test
Matrix 1
0.001   0.585   0.823   0.711
0.564   0.480   0.747   0.514
0.193   0.350   0.174   0.304
0.809   0.896   0.859   0.015
Matrix 2
0.091   0.989   0.009   0.602
0.364   0.446   0.378   0.607
0.147   0.119   0.532   0.166
```

```
0.166   0.005   0.571   0.663
Result matrix :
0.452   0.363   1.064   0.964
0.422   0.862   0.877   1.095
0.221   0.369   0.400   0.560
0.529   1.301   0.811   1.183

c:\Data\tooldevelopment\chapter2files\exercise3\Debug>
```

2.9 Command line arguments in Visual Studio

When opening a cmake project in Visual Studio, the command line arguments can be configured when debugging your project, but is has to be done in a different way.

To start click on

2.10 Exercises

Exercise 1

Make a new directory and follow the course material "A first example". Check if the executable is built and that it runs correctly.

Exercise 2

Download the exercise 2 zip folder from Leho and follow the steps in the document starting from the "Adding an include directory" paragraph on page 52. Ensure that you can create an installable version and that the executable can be run from the command line.

Exercise 3

Dowload the exercise 3 zip folder from Leho and follow the steps in the document starting from the "Static library directory" paragraph on page 62.

Use the second exercise as an example on how to create an installer for this tool. Also make sure that the tool is added to the path.

Exercise 4

Given on Leho is a json file that contains the definition of two 4x4 matrices. Change the `MatrixTest.cpp` file so that it accepts a json file as input (again with command line parameters).

The FMatrix class has the following function to set a value:

```
1 void Set(int r, int c, float val);
```

Use this function to set the two matrices and finally calculate the product of these two matrix.

Chapter 3

Build tools advanced

3.1 Goal

A first goal of this session is to dive deeper into the compilation process itself. The compiler is responsible for generating executables, dll (dynamic link library) files, lib (static or dynamic) files and obj files (not wavefront obj). It is important to have some idea of how these files are produced and how they interact with each other.

In this session we improve the build system and also take into account that maybe we want different builds for different situations. The typical example is a Debug build versus a Release build, or maybe a specific demo build is needed with only part of the game present in it.

Another issue is the usage of libraries where only the binaries are present. For example the FMod library has header files and provides precompiled lib and dll files. The header files are needed for the **compilation** process, the lib files are required during **linking** and the dll files are needed when **running** the application. The build definition has to take these different phases into account.

3.2 The compilation process

When we use the word compilation in the C++ context we usually imply the process of producing an executable program. To be pedantic, this is not what the term actually means. The complete process to produce an executable involves the **preprocessor** step, **compiling**, **assembling** and **linking**.

Also important to realize is that each c or cpp source file in the project first gets compiled separately and each c or cpp file results in an obj file after preprocessing, compilation and assembly. The job of the linker is then to link all the separate object files together to create one executable. Notice that libraries (e.g. FMod, NVidia PhysX, ...) also get linked in this step.

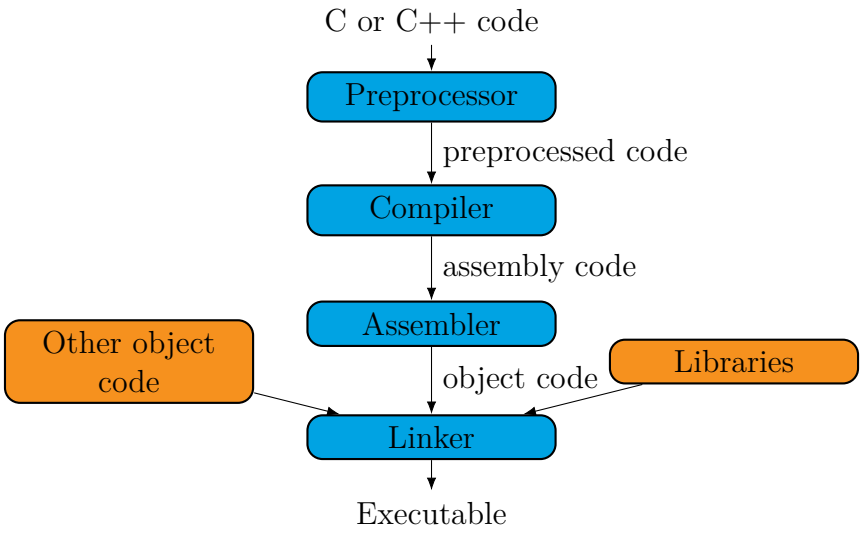


Figure 3.1: Compilation process

An executable is not always the end goal of the build step. There are two other types of files that can be generated by a C/C++ build process:

1. **Library file**: A library file with extension .lib can be **static** or **dynamic**. A **static** library contains actual executable code and can be **integrated** in a final executable. A dynamic library just contains the information to access a dynamic link library (dll file) when an executable that needs it is executed.

2. **Dynamic link library**: A dynamic link library contains executable code that can be called from an executable at **runtime**.

Adding external code as a **static** library has the advantage that there is no runtime dependency on third party code, but has the disadvantage that the executable will be bigger in size.

Preprocessing

The preprocessing step transforms all the preprocessor **directives** into actual C or C++ code. A preprocessor directive is every line of code that starts with the hash-character (#), this means that an include statement is also a preprocessor directive.

Let's take a look at what the preprocessor will do for a simple file (`helloworld.cpp`). There is one include directive, and one define in this file:

```
#include <iostream>
#define HELLO "Hello World!\n"
int main()
{
  std::cout<< HELLO;
}
```

The result of the preprocessing step can be seen with the visual studio compiler by invoking the visual studio compiler with the **/P** flag:

```
cl /P helloworld.cpp
```

The compiler generates a code file named `helloworld.i`. Prepare perhaps for a little shock, because this little program is expanded into a file of about **1.5Mb**. How can we explain this? The culprit is off course the following statement:

```
#include <iostream>
```

This innocent looking statement will include all the source code inside the iostream **header** file into the result of the preprocessing step. This can also potentially lead to **circular** includes where two files for example include each other and explains why C and C++ programmers place a `pragma once` statement or an `ifndef` statement at the top of the header file.

The `iostream` header file for example has the following code at the start, where again other preprocessor directives will be processed and code will be included:

```
#pragma once
#ifndef _IOSTREAM_
#define _IOSTREAM_
#include <yvals_core.h>
#if _STL_COMPILER_PREPROCESSOR
#include <istream>
```

At the very end of the generated `helloworld.i` file we finally see that the simple `#define` statement is also expanded:

```
...
#pragma warning(pop)
#pragma pack(pop)
#line 74 "C:\\Program Files (x86)\\Microsoft Visual Studio\\2019\\Enterprise\\VC
    \\Tools\\MSVC\\14.28.29333\\include\\iostream"
#line 75 "C:\\Program Files (x86)\\Microsoft Visual Studio\\2019\\Enterprise\\VC
    \\Tools\\MSVC\\14.28.29333\\include\\iostream"
#line 2 "exercise1.cpp"

int main()
{
  std::cout<< "Hello World!\n";
}
```

Every cpp file that includes other header files will be **expanded** into a much bigger file by this preprocessor step. There is a trend towards header only libraries at the moment, because they are easier to drop into an existing project. However, the header files will be compiled for every cpp file that references (includes) them, and they lead to longer compile times.

Let us also take a look at a much simpler header and cpp file and what happens with these two files:

A first file is a simple header file with the definition of a Float2 class:

```
class Float2{
  private:
    float m_x;
    float m_y;

  public:
    Float2(float x, float y);
    ~Float2();

    float GetX();
    float GetY();
};
```

Listing 3.1: Float2.h

The second file contains the implementation of this header, and also an include statement:

```
#include "Float2.h"
Float2::Float2(float x, float y): m_x(x),m_y(y){
}

Float2::~Float2(){
}

float Float2::GetX(){
  return m_x;
}

float Float2::GetY(){
  return m_y;
}
```

Listing 3.2: Float2.cpp

```
cl /P Float2.cpp
```

The resulting preprocessed file is a much smaller file now, with newly generated defines that make it possible to report compilation errors to the programmer:

```
#line 1 "Float2.cpp"
#line 1 "c:\\Data\\tooldevelopment\\chapter3files\\buildprocess\\Float2.h"
class Float2{
  private:
    float m_x;
    float m_y;

  public:
    Float2(float x, float y);
    ~Float2();

    float GetX();
    float GetY();
};
#line 2 "Float2.cpp"

Float2::Float2(float x, float y): m_x(x),m_y(y){
}

Float2::~Float2(){
}

float Float2::GetX(){
  return m_x;
}

float Float2::GetY(){
  return m_y;
}
```

Compiler

The compiler converts the code in the preprocessed file into assembly code. This assembly is human readable and still contains the original names of the classes and functions. It is possible to show this assembly code with the /FAs flag:

```
cl /FAs Float2.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.28.29336 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Float2.cpp
Microsoft (R) Incremental Linker Version 14.28.29336.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:Float2.exe
Float2.obj
LINK : fatal error LNK1561: entry point must be defined
```

The error here is due to the fact that there is no main function, we will correct this in the discussion of the linker step.

The file that was generated in this step is: `Float2.asm`, and the following listing shows only part of this asm file, the GetX() method:

```
1  ; Function compile flags: /Odtp
2  ; File c:\Data\tooldevelopment\chapter3files\buildprocess\Float2.cpp
3  _TEXT   SEGMENT
4  _this$ = -4                     ; size = 4
5  ?GetX@Float2@@QAEMXZ PROC               ; Float2::GetX
6  ; _this$ = ecx
7
8  ; 9    : float Float2::GetX(){
9
10     push    ebp
11     mov ebp, esp
12     push    ecx
13     mov DWORD PTR _this$[ebp], ecx
14
15 ; 10   :    return m_x;
16
17     mov eax, DWORD PTR _this$[ebp]
18     fld DWORD PTR [eax]
19
20 ; 11   : }
21
22     mov esp, ebp
23     pop ebp
24     ret 0
25 ?GetX@Float2@@QAEMXZ ENDP             ; Float2::GetX
26 _TEXT   ENDS
```

Important here is **line 5** where a unique identifier for this method (or procedure) is generated. This identifier will be used later in the linking process.

Assembler

The job of the assembler is to generate the obj files that will be used in the linking process. As stated before, every cpp file will result in a separate obj file for further linking.

To demonstrate this, we add a main.cpp file, where we simple create a **Float2** object without printing something to file:

```
#include "Float2.h"

int main()
{
  Float2 f(3,2);
  return 0;
}
```

We now have the following files:

- `main.cpp`
- `Float2.h`
- `Float2.cpp`

To compile the program we need to provide the Microsoft compiler with all the cpp files that need to be compiled. Header files are searched for in the current directory! The command to compile and link is then :

```
cl /FAs Float2.cpp main.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.28.29336 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Float2.cpp
main.cpp
Generating Code...
Microsoft (R) Incremental Linker Version 14.28.29336.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:Float2.exe
Float2.obj
main.obj
```

The process generates the following files:

- `main.cpp` → `main.asm` → `main.obj`
- `Float2.cpp` → `Float2.asm` → `Float2.obj`

The generated obj-files contain executable code, but do not yet have to correct memory addresses to execute procedure calls.

Linker

In the final step, the obj files are combined in one executable.

{`main.obj`,`Float2.obj`} → `Float2.exe`

A curious thing is that the name of the executable is determined by the name of the first source file if the name of the executable was not defined on the command line.

3.3 Compilation with static lib

For very small projects (e.g. less than 20 source files) it makes sense to make a single executable in **one step**. However for bigger projects it can be advantageous to split the project up in several smaller projects that can be compiled separately. Remember however that there will still be **header** files that will be included across projects. For example in the Graphics Programming 2 engine we need to include the fmod header file if we want to use the capabilities of that library.

Let's pretend that our simple example with `main.cpp`, `Float2.h` and `Float2.cpp` is a big project and we want to split it up.

We split the project into a **main** project (a project that can be executed) and a **library** project:

1. **Main** project
    - `main.cpp`
2. **Static library** project
    - `Float2.h`
    - `Float2.cpp`

Here we also need to take the dependencies into account. We cannot first build the main project, because it depends on the library project. By necessity we need to build the library project first, and this build process takes two steps. The first step compiles the code into obj files but does not generate an executable, because the `/c` command line option is specified:

```
1 cl /c Float2.cpp
```

This step generates the `Float2.obj` file, with the preprocessor ,compile and assembly as intermediate build step. Important to note is that just with the

regular build process, an obj file will be generated for each `cpp` file.

The second step is to gather all the obj files of the library (in this case only one) and generate the `.lib` file. This step is easily done with the `lib` command where all the obj files that need to be present in the lib are supplied as command line arguments:

```
lib Float2.obj
```

This step creates the `.lib` file and the two steps of the library build process are shown in the following figure:

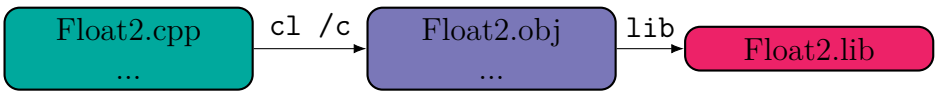


Figure 3.2: Build process for a static library

For the main project we need to again compile the `main.cpp` file. The `Float2` "library" has already been compiled as a `.lib` file but the build process needs to now that it exists. This is done by adding an extra command line parameter, namely the `/link` command line option, followed by the name of the `.lib` to add to the linking process:

```
cl main.cpp /link Float2.lib
Microsoft (R) C/C++ Optimizing Compiler Version 19.28.29336 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

main.cpp
Microsoft (R) Incremental Linker Version 14.28.29336.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:main.exe
Float2.lib
main.obj
```

The build for the final executable is visualized in the following figure:

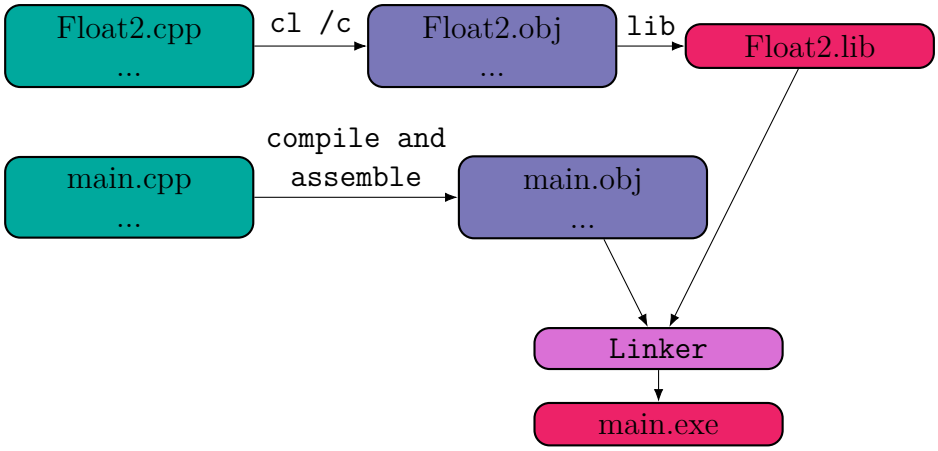


Figure 3.3: Build executable with lib as input

Remember that a static library contains all the necessary code to execute the library. The linker **integrates** a static library **completely** in the final executable!

3.4 Compilation with dynamic library

Another way to build and create an executable is by creating a dynamic link library (dll). However, this method requires that the classes and or functions that are **visible** from an executable have to be defined in one of the following two ways:

1. Add the `__dclspec(dllexport)` definition to the class or function.
2. Create a module or definition file (`.def`) to define the classes and or functions that should be exported.

In terms of organization, there is not a big change, but the Float2 header file has to be modified:

1. **Main** project
   - `main.cpp`
2. **Dynamic library** project
   - `Float2.h`
   - `Float2.cpp`

For the Float2 header file, we will just mark the Float2 class as an exported class. The only change is on **line 1** where the entire Float2 class will be exported:

```
1  class __declspec(dllexport) Float2{
2    private:
3      float m_x;
4      float m_y;
5
6    public:
7      Float2(float x, float y);
8      ~Float2();
9
10     float GetX();
11     float GetY();
12 };
```

Again, we need to build the dynamic library first:

```
1  cl /LD Float2.cpp
2  Microsoft (R) C/C++ Optimizing Compiler Version 19.28.29336 for x86
3  Copyright (C) Microsoft Corporation.  All rights reserved.
4
5  Float2.cpp
6  Microsoft (R) Incremental Linker Version 14.28.29336.0
7  Copyright (C) Microsoft Corporation.  All rights reserved.
8
9  /out:Float2.dll
10 /dll
11 /implib:Float2.lib
12 Float2.obj
13    Creating library Float2.lib and object Float2.exp
```

The build process for a dynamic library is demonstrated in following figure:

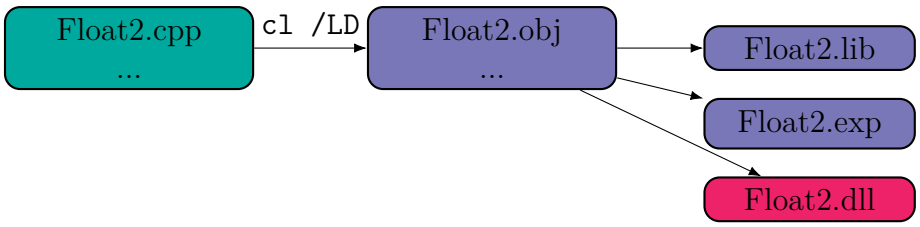


Figure 3.4: Build process for a dynamic library

Notice that again a `.lib` file is produced, this file will be needed to compile the main.cpp file:

```
1  cl main.cpp /link Float2.lib
2  Microsoft (R) C/C++ Optimizing Compiler Version 19.28.29336 for x86
3  Copyright (C) Microsoft Corporation.  All rights reserved.
4
5  main.cpp
```

```
Microsoft (R) Incremental Linker Version 14.28.29336.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:main.exe
Float2.lib
main.obj
   Creating library main.lib and object main.exp

c:\Data\tooldevelopment\chapter3files\buildprocess\dynlib>
```

3.5 Project with static lib in CMake

We can now create a CMake project to compile the simple `Float2` library as a static library.

The directory structure of the project is the following:

`Static lib directory:`

- `CMakeLists.txt` : the CMakeLists.txt file for the project.
- `library`
  - `CMakeLists.txt`
  - `Float2.h`
  - `Float2.cpp`
- `main`
  - `CMakeLists.txt`
  - `main.cpp`

Project CMakeLists.txt file

As always, the root CMakeLists.txt file is kept as simple as possible:

```
cmake_minimum_required(VERSION 3.21)

project(StaticLibraryProject)

add_subdirectory(library)
add_subdirectory(main)
```

/CMakeLists.txt

The project is called `StaticLibraryProject` and the two directories `library` and `main` are added to this project.

Library CMakeLists.txt file

For the library project we call the function `add_library` and instruct this library to compile `Float2.cpp` as part of this library. All the cpp files that are part of this library need to be added to this CMake instruction.

```
1 add_library(Vector Float2.cpp)
2 set(VectorIncludeDir "${CMAKE_CURRENT_SOURCE_DIR}" PARENT_SCOPE)
```

library/CMakeLists.txt

To make it easier to set the include directory for other projects to use we define a variable `VectorIncludeDir` which contains the directory where the header files of this library project are.

Main CMakeLists.txt file

For the main project the `add_library` CMake function is used together which defines the name of the executable project and all the `cpp` files that are needed for this project.

```
1 add_executable(Main main.cpp)
2 message(STATUS "linking with directory : ${VectorIncludeDir}")
3 target_include_directories(Main PUBLIC ${VectorIncludeDir})
4 target_link_libraries(Main PUBLIC Vector)
```

main/CMakeLists.txt

Again, as usual when we use a library project, we need to set the include directories for this project. On **line 3** the include directory for the **Main** project is set via the `target_include_directories` command and uses the `VectorIncludeDir` variable that was in the build file for the library project.

Finally the `target_link_libraries` declares the dependency of the **Main** project on the **Vector** library project.

3.6 Project with dynamic lib in CMake

Compared to the static lib project, the project structure and directories are the same:

`Dynamic lib directory:`

- `CMakeLists.txt` : the CMakeLists.txt file for the project.
- **library**

  - CMakeLists.txt
  - Float2.h
  - Float2.cpp
- main
  - CMakeLists.txt
  - main.cpp

Library CMakeLists.txt file

For the library project we call the function `add_library` and instruct this library to compile `Float2.cpp` as part of this library. All the cpp files that are part of this library need to be added to this CMake instruction.

Compared to a static library, a **dynamic** library project just need to use the `SHARED` keyword to declare the project as a dynamic library.

```
1 add_library(Vector SHARED Float2.cpp)
2 set(VectorIncludeDir "${CMAKE_CURRENT_SOURCE_DIR}" PARENT_SCOPE)
```

library/CMakeLists.txt

To make it easier to set the include directory for other projects to use we define a variable `VectorIncludeDir` which contains the directory where the header files of this library project are.

Don't forget that you need to use `__declspec(dllexport)` to declare the `Float2` class as an exported class:

```
1  class __declspec(dllexport) Float2{
2    private:
3      float m_x;
4      float m_y;
5
6    public:
7      Float2(float x, float y);
8      ~Float2();
9
10     float GetX();
11     float GetY();
12 };
```

library/Float2.h

Main CMakeLists.txt file

We start with the same CMakeLists.txt file as we used in the static lib project:

```
1 add_executable(Main main.cpp)
2 message(STATUS "linking with directory : ${VectorIncludeDir}")
3 target_include_directories(Main PUBLIC ${VectorIncludeDir})
4 target_link_libraries(Main PUBLIC Vector)
```

main/CMakeLists.txt

However when you build this project there is a problem. When we look at the results of the build process we get the following:

Build directory:

- library
  - Debug
    - Vector.dll
- main
  - Debug
    - Main.exe

The Vector.dll needs to be in the same folder as executable Main.exe, but this is clearly not the case. We can solve this by adding a custom command that will be executed after everything is built:

```
1  add_executable(Main main.cpp)
2  message(STATUS "linking with directory : ${VectorIncludeDir}")
3  target_include_directories(Main PUBLIC ${VectorIncludeDir})
4
5  target_link_libraries(Main PUBLIC Vector)
6
7  add_custom_command(TARGET Main POST_BUILD
8      COMMAND ${CMAKE_COMMAND} -E copy
9          "$<TARGET_FILE_DIR:Vector>/Vector.dll"
10         "$<TARGET_FILE_DIR:Main>"
11 )
```

main/CMakeLists.txt

On line 7 a custom command is defined which will be executed after the main executable is built (POST_BUILD) keyword. A simple **copy** is performed of the Vector.dll file here. It is important to note that we use a different type of variable here:

$<TARGET_FILE_DIR:Vector>

Note that this variable does not use curly braces, but instead is defined with the smaller than and greater than characters. This is what is called a **cmake generator expression**.

These expressions are evaluated when the build is actually executed, which means that in this case this folder can be different when a **Debug** build is created versus when a **Release** build is created. In principle, the build system for this project should still work when we do a Release build.

We can test this out off course:

```
1 c:\projects\dynamiclib>mkdir build
2 c:\projects\dynamiclib>cd build
3 c:\projects\dynamiclib>cmake ..
4 c:\projects\dynamiclib>cmake --build .
5 c:\projects\dynamiclib>cmake --build . --config Release
```

On **line 4** the default build is executed, which is the **Debug** build.

On **line 5** the build is executed with the option `--config Release`, which will create the Release build.

3.7 Use case: Compiling a graphics engine with CMake

The GP2 engine has the following structure:

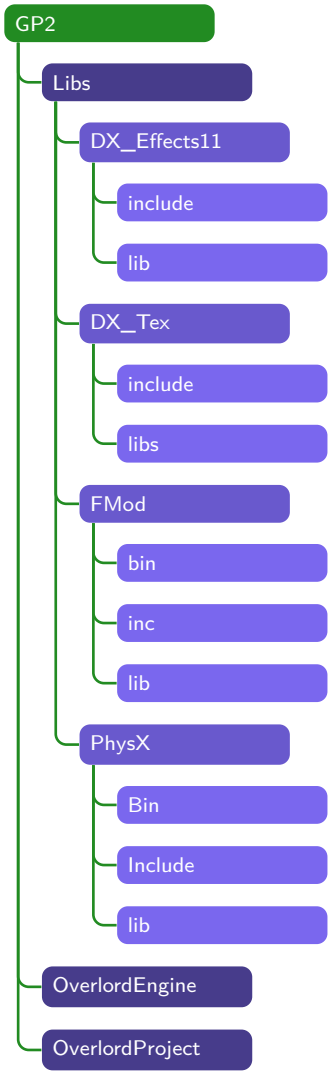


Figure 3.5: Project directory structure

The `include` or `inc` directories contain header files that are needed to compile the **OverlordEngine** and **OverlorProject** projects.

The `lib` or `libs` directories contain the `.lib` files. These files can be **static** or **dynamic** libraries.

Finally, the `Bin` directory contains the `.dll` files that are necessary at runtime.

To compile the project, we will define one master cmakelists.txt file in the GP2 directory that will just define what subdirectories need to be included for the entire buildprocess.

GP2 Directory

In the GP2 directory, add a CMakeLists.txt file with the following contents:

```
1  cmake_minimum_required(VERSION 3.10)
2
3  project(OverlordGame)
4
5  add_subdirectory(OverlordProject)
6  add_subdirectory(OverlordEngine)
7
8  add_subdirectory(Libs/PhysX)
9  add_subdirectory(Libs/Fmod)
10 add_subdirectory(Libs/DX_Tex)
11 add_subdirectory(Libs/DX_Effects11)
```

The important thing here is to define the name of project (line 3). In this case the name is **OverlordGame** (chosen to not collide with directory names that are in the project).

This way of structuring cmake files is important to keep the organization of your build sane and understandable for other people. As build files grow (and they will) they tend to become disorganized and in the end only the developer that maintains the build file can understand it.

The main CMakeLists.txt adds the directories of the six subprojects that need to be included in the build to create the final executable.

The four projects that are in the `Libs` directory do not provide the c or cpp sources that are necessary to build the `.lib` or `.dll` files ourself. Instead, we need to use the provided `.lib` and `.dll` files in those libraries.

- **DX_Effects11**: A helper library to help with the compilation process of shaders, and adds the possibility to define techniques. This library is statically included in the build, so there are only headers and `.lib` files.
- **DX_Tex**: A directory that helps with the DirectX textures. This library is statically included in the build, so there are only headers and `.lib` files.

- **PhysX** : This library adds the physics simulation capabilities. This library is a dynamic library and has header files, `.lib` files and `.dll` files.
- **FMod** : This library adds the fmod sound library simulation capabilities. This library is also a dynamice library and has header files, `.lib` files and `.dll` files.

The **OverlordEngine** and **OverlordProject** directories have source files that will be compiled and linked during the build process.

DX_Effects11 CMakeLists.txt

The location for this file is : `GP2/Libs/DX_Effects11`

The CMakeLists.txt file is relatively simple:

```
1 add_library(DXEffects11 INTERFACE)
2 target_include_directories (OverlordEngine PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/
      include)
3 target_include_directories (OverlordGame PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/
      include)
```

The first line of the file defines the library in this folder with the name DXEffects11 and defines it as an INTERFACE type library. For CMake this means that only the header files will be included in the build when this library is needed for the compilation of other projects.

Both the OverlordEngine and the OverlordGame projects use this library, so the following two lines tell CMake that these two projects need to use the include directory in this folder.

It is important to also realize that the variable `$CMAKE_CURRENT_SOURCE_DIR` is the directory where the CMakeLists.txt file is located.

DX_Tex CMakeLists.txt

The location for this file is : `GP2/Libs/DX_Tex`

This make file is similar to the DX_Effects11 library, because this is also a header only project with provided static libraries.

The only difference is the first line where the name of the library is different.

```
1 add_library(DXTex INTERFACE)
2 target_include_directories (OverlordEngine PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/
      include)
```

```
3 target_include_directories (OverlordGame PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/
      include)
```

FMod CMakeLists.txt

The location for this file is : `GP2/Libs/FMod`

This make file is also similar to the DX_Effects11 library, because this is also a header only project with provided **dynamic** libraries. But for this library there is also a dll file that is needed to run the final executable.

With the install command we can copy this `FMod.dll` file into that ”final destination” or in other words the directory where all the build results will be gathered, in order to make the installation fie:

```
1 add_library(FMod INTERFACE)
2 target_include_directories (OverlordEngine
3     PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/inc)
4 target_include_directories (OverlordGame
5     PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/inc)
6
7 install(FILES ${CMAKE_CURRENT_SOURCE_DIR}/bin/FMod.dll DESTINATION bin)
```

GP2/Libs/FMod/CMakeLists.txt

Physx CMakeLists.txt

The location for this file is : `GP2/Libs/PhysX`

Similar to the FMod library, but the PhysX library has more dll files to copy to the final installation directory:

```
1 add_library(PhysX INTERFACE)
2 target_include_directories (OverlordEngine PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/
      Include)
3 target_include_directories (OverlordGame PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/
      Include)
4
5 install(FILES ${CMAKE_CURRENT_SOURCE_DIR}/bin/Debug/d3dcompiler_46.dll
      DESTINATION bin)
6 install(FILES ${CMAKE_CURRENT_SOURCE_DIR}/bin/Debug/nvToolsExt32_1.dll
      DESTINATION bin)
7 install(FILES ${CMAKE_CURRENT_SOURCE_DIR}/bin/Debug/PhysX3
      CharacterKinematicDEBUG_x86.dll DESTINATION bin)
8 install(FILES ${CMAKE_CURRENT_SOURCE_DIR}/bin/Debug/PhysX3CommonDebug_x86.dll
      DESTINATION bin)
9 install(FILES ${CMAKE_CURRENT_SOURCE_DIR}/bin/Debug/PhysX3Debug_x86.dll
      DESTINATION bin)
```

GP2/Libs/PhysX/CMakeLists.txt

OverlordEngine CMakeLists.txt

The location for this file is : `GP2/OverlordEngine`

This file is a bit bigger but not necessarily more complex. This project is again a library so we need to use the `add_library` command again. This command also needs to know what all the cpp files are that it needs to compile:

```
1  add_library(OverlordEngine
2      #base
3      GameTime.cpp
4      InputManager.cpp
5      SoundManager.cpp
6      OverlordGame.cpp
7      #Components
8      BaseComponent.cpp
9      CameraComponent.cpp
10     ColliderComponent.cpp
11     MeshDrawComponent.cpp
12     MeshIndexedDrawComponent.cpp
13     RigidBodyComponent.cpp
14     TransformComponent.cpp
15     #Content
16     ContentManager.cpp
17     EffectLoader.cpp
18     #Diagnostics
19     DebugRenderer.cpp
20     GameSpecs.cpp
21     Logger.cpp
22     #Graphics
23     RenderTarget.cpp
24     #Helpers
25     BinaryReader.cpp
26     EffectHelper.cpp
27     PhysXHelper.cpp
28     VertexHelper.cpp
29     #PhysX
30     PhysXErrorCallback.cpp
31     PhysXManager.cpp
32     PhysXProxy.cpp
33     #Prefabs
34     CubePrefab.cpp
35     FixedCamera.cpp
36     FreeCamera.cpp
37     SpherePrefab.cpp
38     TorusPrefab.cpp
39     #SceneGraph
40     GameObject.cpp
41     GameScene.cpp
42     SceneManager.cpp
43     )
44 target_include_directories (OverlordGame PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
45 add_definitions(-DUNICODE)
```

The OverlordGame still needs the know the location of the header files, so on line 44 the `target_include_directories` command is again called to make the OverlordGame subproject aware of this directory.

Finally, if windows APIs are used, there is a choice to use the ASCII style windows functions or the Unicode windows function. The `add_definitions` function makes it possible to add a `#define` preprocessor command to the build process.

OverlordGame CMakeLists.txt

The location for this file is : `GP2/OverlordProject`

Finally we have the main executable that we want to create. Make a note of the first line, where the `add_executable` command is called, in contrast with the `add_library` command for the other projects.

On **line 10** we declare the this project needs the OverlordEngine library, and that this `.lib` file will be built by cmake itself.

Line 11 to 16 declare the locations of the existing (precompiled) lib files that are also needed to build the final project.

Line 18 declares that everything that is build for the OverlordGame has to be part of the installation.

Line 19 then states that the folder Resources (with meshes, fx-files and textures) should also be copied to be part of the installation.

Line 21 to 27 are then again needed to define how the installation process should work.

```
1  add_executable(OverlordGame
2      Main.cpp
3      MainGame.cpp
4      "CourseObjects/Week 3/MinionScene.cpp"
5
6  )
7  add_definitions(-DUNICODE)
8  message(STATUS "build dir: ${CMAKE_CURRENT_SOURCE_DIR}")
9
10 target_link_libraries(OverlordGame PUBLIC OverlordEngine)
11 target_link_directories(OverlordGame
12     PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/../Libs/PhysX/Lib
13     PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/../Libs/DX_Effects11/Lib
14     PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/../Libs/FMod/Lib
15     PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/../Libs/DX_Tex/Lib
16     )
17
18 install(TARGETS OverlordGame DESTINATION bin)
19 install(DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/Resources DESTINATION bin)
20
21 include(InstallRequiredSystemLibraries)
22 set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
23 set(CPACK_PACKAGE_VERSION_MAJOR "1")
24 set(CPACK_PACKAGE_VERSION_MINOR "0")
25
```

```
26 set(CPACK_NSIS_MODIFY_PATH ON)
27 include(CPack)
```

Creating the cmake files

The final step is to create a build folder where everything will be stored. This makes it easier to separate the build process from the actual source, library and dll files, and to ”clean” the project.

For example create a directory : `GP2/build`

and navigate to this directory in the command shell, and generate all the necessary build files:

```
c:\Data\tooldevelopment\chapter3files\exercise1\GP2\build>cmake .. -A Win32
-- Selecting Windows SDK version 10.0.18362.0 to target Windows 10.0.19041.
-- build dir: C:/Data/tooldevelopment/chapter3files/exercise1/GP2/OverlordProject
-- Configuring done
-- Generating done
-- Build files have been written to:
    C:/Data/tooldevelopment/chapter3files/exercise1/GP2/build

c:\Data\tooldevelopment\chapter3files\exercise1\GP2\build>
```

When this is done succesfully , it is possible to actually build the project. Notice that we are still in the build folder but use the current directory (indicated by the .-character) to build the system. It is for example possible to create a second build system with other parameters (e.g. Release mode) to create another installation.

```
c:\Data\tooldevelopment\chapter3files\exercise1\GP2\build>cmake --build .
Microsoft (R) Build Engine version 16.8.3+39993bd9d for .NET Framework
Copyright (C) Microsoft Corporation. All rights reserved.

  OverlordEngine.vcxproj -> C:\Data\tooldevelopment\chapter3files\exercise1\GP2\
      build\OverlordEngine\Debug\OverlordEngi
  ne.lib
  OverlordGame.vcxproj -> C:\Data\tooldevelopment\chapter3files\exercise1\GP2\
      build\OverlordProject\Debug\OverlordGame.
  exe

c:\Data\tooldevelopment\chapter3files\exercise1\GP2\build>
```

And then finally, the installation executable can be generated with the following command:

```
c:\Data\tooldevelopment\chapter3files\exercise1\GP2\build>cpack -C Debug
CPack: Create package using NSIS
CPack: Install projects
CPack: - Install project: OverlordGame [Debug]
CPack: Create package
CPack: - package:
    C:/Data/tooldevelopment/chapter3files/exercise1/GP2/build/OverlordGame-1.0.1
    -win32.exe generated.

c:\Data\tooldevelopment\chapter3files\exercise1\GP2\build>
```

3.8 Exercises

Exercise 1

Recreate the static lib compilation with CMake. (the exercise with the main.cpp, Float2.h and Float2.cpp).

Hint: also lookup what #pragma comment lib can do for you.

Exercise 2

The subprojects in the case study directly define the include directories. As we have seen last week we can set variables for later use in the sub projects that need them for the include directories.

Change the case study to work with this scheme.

Exercise 3

Recreate the dynamic lib compilation with CMake. (the exercise with the main.cpp, Float2.h and Float2.cpp).

Hint: also lookup what#pragma comment lib can do for you.

Hint 2: You will probably run into some difficulties trying to debug this project within Visual Studio. Try to find a way, we will look at this again next week.

Chapter 4

User interfaces with JUCE

4.1 Goal

Building user interfaces can also be a valid way to build tools that can be used in a game development pipeline.

There are lots of potential UI SDKs but the one that is going to be discussed in this chapter is the JUCE toolkit. This toolkit can be compiled with CMake and also offers CMake functions (which will be discussed in this chapter) to make the creation of GUI projects with JUCE easier.

As you can see from the above screenshot, JUCE is a gui toolkit that is geared towards sound applications. However, it has a full set of user interface components and can even be used to create simple animations.

4.2 A Juce CMake project

Download JUCE from `https://juce.com/download/`. The latest version is JUCE 7.0.5 and the software for Windows comes in the form of a zip file that can be extracted to a convenient location. We will need the **JUCE** directory inside a CMake project to be able to build a JUCE Gui application from within said project.

We begin with the directory layout of the Juce project, in a way that fits the assignment itself. Next, we explain the concept of functions within CMake to conclude with a simple GUI application that is not dependent on another project.

Finally, an additional library project is added to the mix to showcase how to use this library project from within the Juce GUI project.

4.2.1 Project structure

For the demonstration we will use the following project structure:

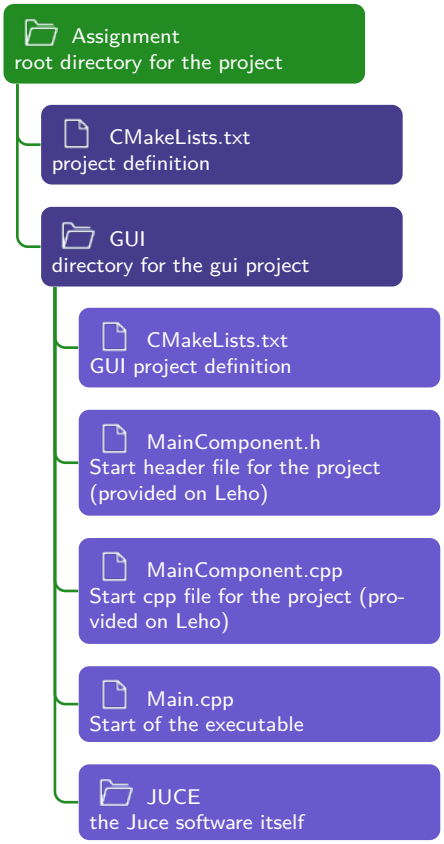


Figure 4.1: Project directory structure

Create this structure and copy the entire JUCE software into the GUI directory. This could be considered wasteful, but it is the most robust way to include a JUCE gui into the project.

4.2.2 CMake files

CMakeLists.txt - project definition

The first CMakeLists.txt file is in the root directory and is used to setup the project. As usual this corresponds to a solution and all the projects are stored in a subfolder to this directory.

```
1 cmake_minimum_required(VERSION 3.12)
2
3 project(MyJUCEApp VERSION 1.0.0)
4 # add the JUCE software as library in the project.
5 add_subdirectory("JUCE")
6 # add the JuceGUI folder with your own code to the project.
7 add_subdirectory("JuceGUI")
```

GUi sub project definition - CMakeLists.txt

Here we use two functions that were defined in the JUCE subproject.

- juce_add_gui_app : define a cmake project for a JUCE GUI app.
- juce_generate_juce_header : generate a header that includes all the JUCE header files that are necessary for a JUCE GUI app.

```
set(SOURCES
    Main.cpp
    MainComponent.cpp
)

# Add an executable target for the project
juce_add_gui_app(MyJUCEApp
    PRODUCT_NAME "My JUCE App"
    VERSION ${PROJECT_VERSION}
    COMPANY_NAME "My Company"
    DOCUMENT_EXTENSIONS "myext"
    DOCUMENT_NAMES "My Document"
    DOCUMENT_DESCRIPTIONS "My Document Description"
)

juce_generate_juce_header(MyJUCEApp)

target_sources(MyJUCEApp PRIVATE ${SOURCES})

# Link against the JUCE module
target_link_libraries(MyJUCEApp
    PRIVATE
        # GuiAppData            # If we'd created a binary data target, we'd link
            to it here
        juce::juce_gui_extra
    PUBLIC
        juce::juce_recommended_config_flags
        juce::juce_recommended_lto_flags
        juce::juce_recommended_warning_flags
)
```

Chapter 5

User interfaces with Qt

5.1 Goal

Building user interfaces can also be a valid way to build tools that can be used in a game development pipeline.

There are lots of potential UI SDKs but the one that was chosen for this course is the Qt toolkit. This toolkit is cross platform (even for Android for example) and as an added bonus can also be compiled with CMake, or integrated into a CMake build.

5.2 Installation of Qt

Download the latest version of Qt from the following url:

`https://download.qt.io/official_releases/qt/6.2/6.2.3/single/`

To build Qt we follow the instructions on the following page:

`https://doc.qt.io/qt-6/windows-building.html`

When installing Python make sure to add the location of the Python executable to the `Path` variable during execution. I also recommend not to accept the default path for the installation, as it is some weird directory in the user folder. Choose a more sensible path to avoid unnecessary pain and suffering.

The `Ninja` tool is just a simple executable, and you need to add it to the Path environment variable yourself.

Finally, the active `perl` installer is pretty well behaved, and it also offers the option of adding the executable to the Path environment variable.

Command environment for Qt

To facilitate the build process it is recommended to create a command environment for Qt. The following script declares the directories that are in use, it is off course necessary to change the directories in this script to your situation:

```
REM Set up Microsoft Visual Studio 2022 for 64 bit projects
CALL "C:\Program Files\Microsoft Visual Studio\2022\Enterprise\VC\Auxiliary\Build
    \vcvarsall" x64
SET _ROOT=C:\Data\Qt\qt-everywhere-src-6.2.3
SET PATH=%_ROOT%\qtbase\bin;%PATH%
SET _ROOT=
```

Listing 5.1: qt.cmd file

A `cmd` file is useful to perform some common actions before starting a command shell.

To actually launch a command shell with this cmd file **create** a **short cut**. The shortcut contains the following value for the target property:

```
%SystemRoot%\system32\cmd.exe /E:ON /V:ON /k C:\data\projects\qt\qt.cmd
```

The options for this command are:

- `/E:ON` → Enable command extensions. A command extensions means that you can type the name of a file in the command shell and the associated application will automatically open.
- `/V:ON` → Enables delayed environment variable expansion. Delayed expansion will cause variables within a batch file to be expanded at execution time rather than at parse time. This means that each time a variable is used it will be re-evaluated.
- `/k` → Executes the commands in the given file.

Build and install

Finally, we can build the Qt system and also install. The install script only works if the command shell has administrator privileges. So right click on the short cut you created and select Run as Administrator:

```
C:\WINDOWS\system32>REM Set up Microsoft Visual Studio 2019, where <arch> is
    amd64, x86, etc.

C:\WINDOWS\system32>set PYTHONHOME=C:\Data\projects\qt\Python

C:\WINDOWS\system32>set PYTHONPATH=C:\Data\projects\qt\Python\Lib

C:\WINDOWS\system32>CALL "C:\Program Files (x86)\Microsoft Visual Studio\2019\
    Enterprise\VC\Auxiliary\Build\vcvarsall.bat" x64
**********************************************************************
** Visual Studio 2019 Developer Command Prompt v16.8.4
** Copyright (c) 2020 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

c:\Data\projects\qt\src\qt601>
```

In this environment you can now first configure the Qt system:

```
c:\Data\projects\qt\src\qt601>configure
```

Followed by a parallel build of the Qt system:

```
c:\Data\projects\qt\src\qt601>cmake --build . --parallel
```

And to install:

```
c:\Data\projects\qt\src\qt601>cmake --install .
```

Take note of where the Qt system gets installed because we need it for the Qt examples of this chapter.

5.3 A first Qt example

For this example we just take the analog clock example of Qt itself and create a build script for it. The result of the analog clock is the following:

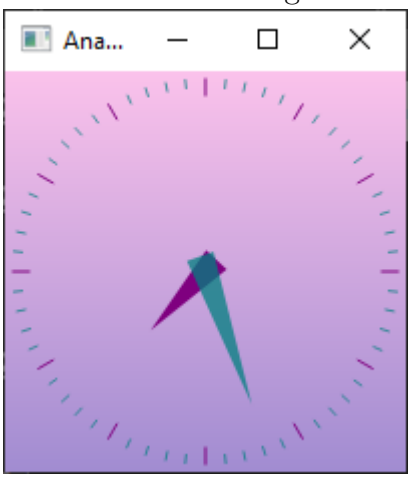


Figure 5.1: Q
t Analog clock example.

The contents of the sample is the following:

- `main.cpp` : The cpp file that starts the application.
- `rasterwindow.cpp` : A helper class to help with drawing the hands and the tick marks of the clock.
- `License.txt` : A license for the installation of this clock.
- `CMakeLists.txt` : The CMake file to build this sample.

Because this is a small example we will just create one single cmake file and configure it to build our application.

A very important **change** must be made on **line 4**. This line tells the cmake build system where to find the build instructions for linking with the Qt modules. If this path is wrong, the build will not work.

Lines 8-10 are specific Qt preprocessing steps. For example, MOC stands for Meta-Object compiler and replaces Qt specific macros with C++ code, before the preprocessor runs.

More information about this system can be found on : `https://doc.qt.io/archives/qt-4.8/moc.html#:~:text=The%20Meta%2DObject%20Compiler%2C%20moc,object%20code%20for%20those%20classes.`

RCC stands for Resource compiler and helps to **embed** icon files for example in the final executable.

Finally UIC, is the UI compiler which compiles a `.ui` file into a C++ coded user interace.

Line 12 then searches for the widgets package which instructs the CMake build system how to build the ui components.

The rest of the script is familiar terrain, where also the necessary instructions for creating an installer are specified.

```
1  cmake_minimum_required(VERSION 3.16.0)
2
3  project(helloworld VERSION 1.0.0 LANGUAGES CXX)
4  LIST(APPEND CMAKE_PREFIX_PATH "c:/program files (x86)/Qt/lib/cmake/")
5  set(CMAKE_CXX_STANDARD 17)
6  set(CMAKE_CXX_STANDARD_REQUIRED ON)
7
8  set(CMAKE_AUTOMOC ON)
9  set(CMAKE_AUTORCC ON)
10 set(CMAKE_AUTOUIC ON)
11
12 find_package(Qt6 COMPONENTS Widgets REQUIRED)
13
14 add_executable(helloworld
15     rasterwindow.cpp
16     main.cpp
17 )
18 target_link_libraries(helloworld PRIVATE Qt6::Widgets)
19 install(TARGETS helloworld DESTINATION bin)
20 install(FILES "c:/Program Files (x86)/Qt/bin/Qt6Cored.dll" DESTINATION bin)
21 install(FILES "c:/Program Files (x86)/Qt/bin/Qt6Guid.dll" DESTINATION bin)
22 install(FILES "C:/Program Files (x86)/Qt/plugins/platforms/qwindowsd.dll"
       DESTINATION bin/platform)
23 include(InstallRequiredSystemLibraries)
24 set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
25 set(CPACK_PACKAGE_VERSION_MAJOR "1")
26 set(CPACK_PACKAGE_VERSION_MINOR "0")
27
28 set(CPACK_NSIS_MODIFY_PATH ON)
29 include(CPack)
```

5.4 Exercises

5.5 Prerequisite

Build and install Qt as instructed in this chapter

5.6 Exercise 1

Build the first exercise on Leho. Verify that the executable works. Study the code in this sample to get a feeling for how Qt works.

5.7 Exercise 2

Build the second exercise on Leho. Verify that the executable works. This is a small text editor. Study the code and look at how you can react to events.

5.8 Exercise 3

This exercise is optional for this week, we will also discuss this next week. Do this if you have time to spare.

There are a lot of tutorials about Qt6 on the website. Follow the following tutorial to create a text editor Qt6 application: `https://doc-snapshots.qt.io/qt6-dev/qtwidgets-mainwindows-application-example.html`

Chapter 6

Qt GUI applications

6.1 Goal

With the Qt build completed we can now turn our attention to actually making applications with Qt.

The easiest way to create a GUI is with the Qt user interface designer. We will use this designer to create a `.ui` file and load it into our GUI application. The user interface components can then be found at startup with the help of `findChild` methods. To load a ui-file dynamically it is also best to add it as a resource that is **embedded** in the final executable.

We will first look at an example of a Qt application that uses an embedded `ui`-file and then work our way to a user interface that displays useful information about a 3D file.

Another goal is to start the project with some sources and a cmake file, to generate the visual studio solution and to work within this visual studio solution to create a working user interface.

6.2 Containers and widgets

Most UI framework have the concept of widget **containers** and widgets themselves. A container has a list of child widgets, and it is the responsibility of the container to perform the **layout** of its children.

A **container** is itself again a **widget**, so this follows the file/directory pattern. This means that a container widget can contain other container widgets, and can form a nested hierarchy of widgets.

The container widgets can be found on the left hand side of the Qt designer:

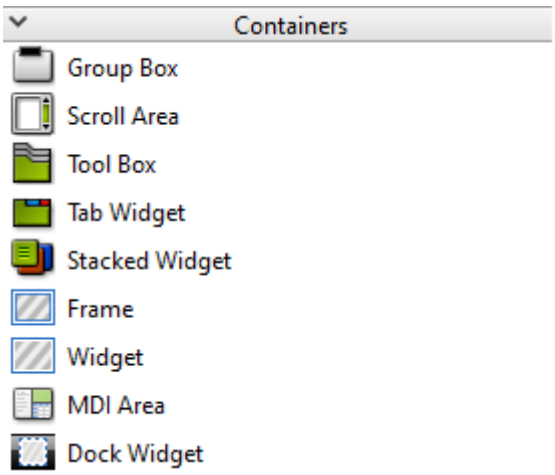


Figure 6.1: Widget containers

You can find more information about widgets at the following url : `https://doc.qt.io/archives/qt-4.8/designer-using-containers.html`.

Layouts

There are a couple of layouts available in Qt designer. These layouts are not regarded as container object, all though they can have children.

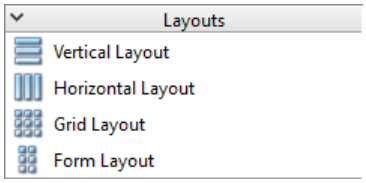


Figure 6.2: Layouts

Layout are more considered as helper classes that help within a container to ”lay out” the widgets in for example a horizontal row, or vertical column.

6.3 The calculator example

In the source folder the calculator example can be found at :`qttools/examples/designer/calculatorbuilder`. This is a perfect little example of using resources and embedding them into the executable, some simple user interface components, and how to use events to interact with them. All the source code for this example is bundled into `exercise1.rar` on the learning environment.

We will add some components to this form and also capture some simple events from this components.

The UI designer

Open the designer and open the `calculatorform.ui` :

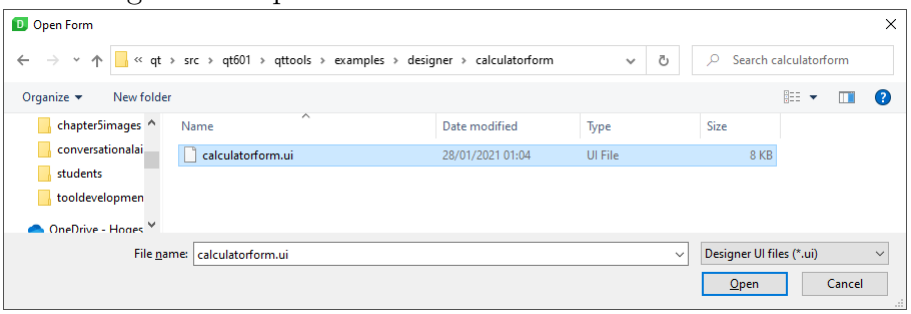


Figure 6.3: Open a ui-file

Open the designer and open the `calculatorform.ui`. Typically a user interface will be referred to as a **form** and the components are referred to as **widgets**. The configurable parts of these widgets or components are called **properties**.

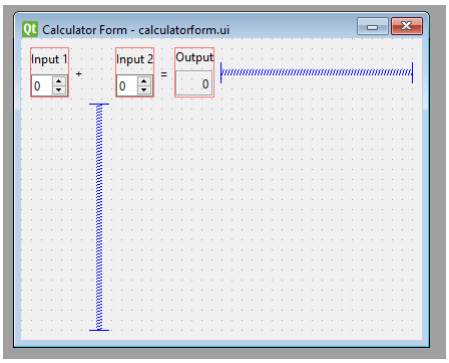


Figure 6.4: UI - Form

The ”springy” looking elements are **spacers**. **Spacers** ensure that other gui elements remain at a fixed location. When this form is resized, the spacers will grow vertically or horizontally depending on their **orientation** setting, which can only be **horizontal** or **vertical**.

A spacer also has a **sizeType** property which can be the following:

- **Expanding** : This is the setting that is used the most. The spacer will take as much space as it can, by **compressing** other components into their minimum size.
- **Fixed** : The spacer has a fixed size. Can be useful to guarantee a fixed size between components or a fixed `margin` of a component with a boundary of the form. This fixed size is determined by the `sizeHint`
- **Minimum** : The `sizeHint` property defines the **minimum** size of this spacer. The spacer can still ”grow” beyond this size.
- **Maximum** : The `sizeHint` property defines the **maximum** size of this spacer. The spacer can still ”shrink” beyond this size.
- **Preferred** : The `sizeHint` defines the **preferred** size of this spacer. This **preferred** size is the size that will be used when the application is first shown on the screen, if possible. The spacer can still shrink and grow beyond this preferred size.
- **MinimumExpanding** : The `sizeHint()` is minimal, and sufficient. The widget can make use of extra space, so it should get as much space as possible (e.g. the horizontal direction of a slider).
- **Ignored** : the `sizeHint()` is ignored. The widget will get as much space as possible.

The other elements in the UI form are self explanatory. Interesting to note is the tree structure of the UI is visible on the right hand side:

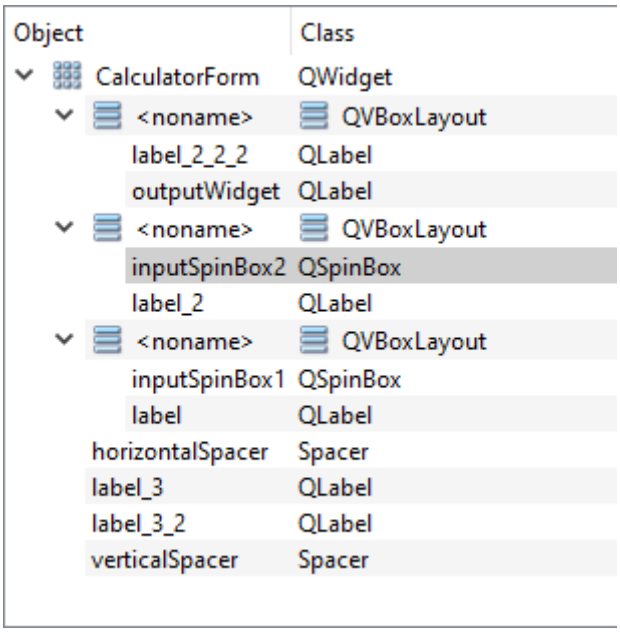


Figure 6.5: UI tree structure

In this tree structure we can see the following UI Widgets and layouts:

- **QVBoxLayout** : A layout that arranges its children in a vertical column.
- **QLabel** : A label
- **QSpinBox** : A spinner widget that allows the user to change a number.

The CMake file

Again a typical CMake file that will be able to compile our little example.

```
1  # Generated from calculatorform.pro.
2
3  cmake_minimum_required(VERSION 3.14)
4  project(calculatorform LANGUAGES CXX)
5  set( QtBaseDir "c:/program files (x86)/Qt")
6  LIST(APPEND CMAKE_PREFIX_PATH "${QtBaseDir}/lib/cmake/")
7  set(CMAKE_INCLUDE_CURRENT_DIR ON)
8
9  set(CMAKE_AUTOMOC ON)
10 set(CMAKE_AUTORCC ON)
11 set(CMAKE_AUTOUIC ON)
12
13 find_package(Qt6 COMPONENTS Core)
14 find_package(Qt6 COMPONENTS Gui)
15 find_package(Qt6 COMPONENTS Widgets)
16
17 qt_add_executable(calculatorform
18     calculatorform.cpp calculatorform.h calculatorform.ui
19     main.cpp
20 )
21 set_target_properties(calculatorform PROPERTIES
22     WIN32_EXECUTABLE TRUE
23     MACOSX_BUNDLE TRUE
24 )
25 target_link_libraries(calculatorform PUBLIC
26     Qt::Core
27     Qt::Gui
28     Qt::Widgets
29 )
30
31 install(TARGETS calculatorform DESTINATION bin)
32 install(FILES   "${QtBaseDir}/bin/Qt6Cored.dll"
33                 "${QtBaseDir}/bin/Qt6Guid.dll"
34                 "${QtBaseDir}/bin/Qt6Widgetsd.dll"
35         DESTINATION bin)
36
37 install(FILES "${QtBaseDir}/plugins/platforms/qwindowsd.dll"
38         DESTINATION bin/platforms )
39
40 include(InstallRequiredSystemLibraries)
41 set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
42 set(CPACK_PACKAGE_VERSION_MAJOR "1")
43 set(CPACK_PACKAGE_VERSION_MINOR "0")
44
45 set(CPACK_NSIS_MODIFY_PATH ON)
46 include(CPack)
```

A couple of finer points:

- **line 5** : The set command is used to define a variable. Here we define our own variable name `QtBaseDir` is defined which will be set to the base directory of the Qt install. To reference this variable (for example in a string), the following syntax is used : `${QtBaseDir}`.

- **line 6** : The variable `QtBaseDir` is used to help define the path where the Qt6 cmake files can be found.
- **line 18** : the `calculatorform.ui` file is included in the build process. This file will be compile by the **UIC compiler** which is enabled on **line 11**.
- **line 32-35** : The variable `QtBaseDir` is used to define the copy operations for the installer package.
- **line 37** : The `qwindowsd.dll` is also copied for the installation process.

Improvement with lists

CMake is actually a programming langue, it is possible to define lists and also for loops to create operations on lists. We can use these capabilities to improve the CMake file, and we will focus on the install commands.

We **replace lines 32-35** with the following:

```
1 set( DLLS "Qt6Core" "Qt6Gui" "Qt6Widgets" )
2 foreach(dll IN LISTS DLLS )
3 install(FILES "${QtBaseDir}/bin/${dll}d.dll" DESTINATION bin)
4 endforeach()
```

Line 1 defines a **list** with the name `DLLS`. We can iterate over this list with the foreach command where (in this case) **dll** is the loop variable and with the `IN LISTS` syntax we specify that we want to iterate over the `DLLS` list.

`dll` is now a **variable** so on **line 3** we can now use this variable inside a string. Notice that the character 'd' is added to each dll to specify that we want the debug **version** of the dll.

Only one platform dll is necessary for this example so we leave line 38 of the original script unmodified:

```
1 install(FILES "${QtBaseDir}/plugins/platforms/qwindowsd.dll"
2         DESTINATION bin/platforms )
```

Windows dlls and CMake

On most operating system a shared library (dll) is a serious matter. On Linux for example, the location and version of shared libraries is registered and it is easy to make use of these shared libraries in an executable.

As we all know, on Windows anything goes, and copies of the exact same dlls litter many installations. However, if we want to develop with Visual Studio

this poses a bit of a problem. The easiest way to be able to run the program with Visual Studio is to add the location of the dll files via the `PATH` environment variable.

main.cpp

The main file is fairly simple. QApplication contains the main event loop, where all events from the window system and other sources are processed and dispatched. It also handles the application's initialization, finalization, and provides session management. In addition, QApplication handles most of the system-wide and application-wide settings.

For any GUI application using Qt, there is precisely one QApplication object, no matter whether the application has 0, 1, 2 or more windows at any given time.

```
1  #include <QApplication>
2
3  #include "calculatorform.h"
4
5  int main(int argc, char *argv[])
6  {
7      QApplication app(argc, argv);
8      CalculatorForm calculator;
9      calculator.show();
10     return app.exec();
11 }
```

On line 8 and 9 the CalculatorForm object is created and shown on the screen.

CalculatorForm.h

You should notice that there is no file `ui_calculatorform.h` in the exercise1 source directory. This file will be generated from the `ui` file by the UIC (User Interface compiler). And this file also generates the form class in the namespace Ui with as class name the name of the form as defined inside the UI file.

The generated files for this exercise can be found in the folder `calculatorform_autogen`.

On **line 11** the `Q_OBJECT` macro is used, this is used to indicate that this is an object that should be preproccessed with the MOC (Meta Object Compiler) system.

The constructor is defined as an explicit constructor, more information about

the explicit keyword can be found here: `https://en.cppreference.com/w/cpp/language/explicit`

Line 16 to 18 define the events that we want to capture in our user interface. The **slots** keyword is not a C++ keyword but will also be processed by the MOC system.

```
1  #ifndef CALCULATORFORM_H
2  #define CALCULATORFORM_H
3
4  //! [0]
5  #include "ui_calculatorform.h"
6  //! [0]
7
8  //! [1]
9  class CalculatorForm : public QWidget
10 {
11     Q_OBJECT
12
13 public:
14     explicit CalculatorForm(QWidget *parent = nullptr);
15
16 private slots:
17     void on_inputSpinBox1_valueChanged(int value);
18     void on_inputSpinBox2_valueChanged(int value);
19
20 private:
21     Ui::CalculatorForm ui;
22 };
23 //! [1]
24
25 #endif
```

CalculatorForm.cpp

The implementation of the CaculatorForm class is then responsible to initialize the ui with the `setupUi` method.

The two event methods capture changes on the spin boxes and set the result of the addition in the outputWidget.

```
#include "calculatorform.h"

//! [0]
CalculatorForm::CalculatorForm(QWidget *parent)
    : QWidget(parent)
{
    ui.setupUi(this);
}
//! [0]

//! [1]
void CalculatorForm::on_inputSpinBox1_valueChanged(int value)
{
    ui.outputWidget->setText(QString::number(value + ui.inputSpinBox2->value()));
}
//! [1]

//! [2]
void CalculatorForm::on_inputSpinBox2_valueChanged(int value)
{
    ui.outputWidget->setText(QString::number(value + ui.inputSpinBox1->value()));
}
//! [2]
```

6.4 Table example

For the table example we create a QtMainWindow, which is different from the first example. The table widget fills the entire client area, and we also take a look at how to create this form in the Qt designer.

UI

Follow along with the demonstration to see how this example was created.

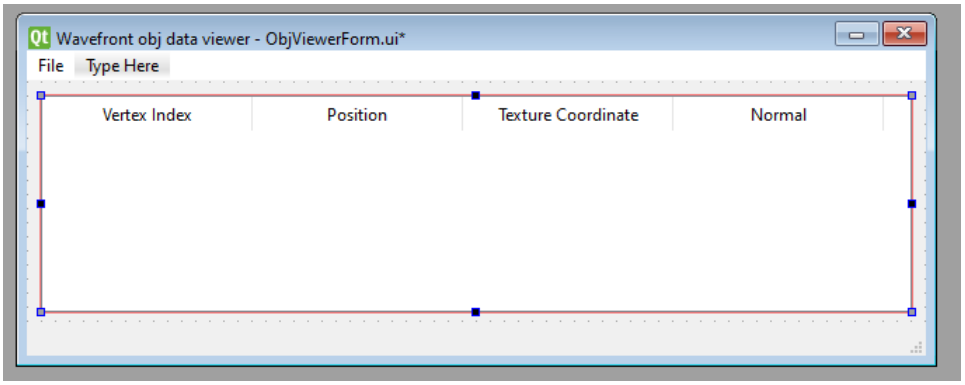


Figure 6.6: UI Definition for the table example

The structure of this user interface is as follows:

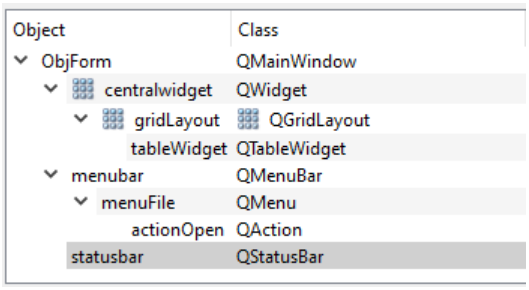


Figure 6.7: UI Definition tree

Notice that the root object in this user interface is now a QMainWindow object. This will have implications for the code later on.

Also notable is that there is one menu item, which will be used to open a file.

Last but not least the name of ui file is `ObjViewerForm.ui` and this will have to correspond with the name of the header and cpp source file.

CMake file

The CMake file is very similar to the CMake file of the first example.

main.cpp

Again, the main function is responsible to create one (and only one) Qapplication object.

The next step is then the creation of the ObjViewerForm object which will be shown on the screen.

```
#include <QApplication>

#include "ObjViewerForm.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    ObjViewerForm objViewer;
    objViewer.show();
    return app.exec();
}
```

Listing 6.1: main.cpp

ObjViewerForm.h

Notice that the name of this header file corresponds with the name of the ui file (`ObjViewerForm.ui`)

```
#ifndef OBJVIEWERFORM_H
#define OBJVIEWERFORM_H

#include "ui_objviewerform.h"
class ObjViewerForm : public QMainWindow
{
    Q_OBJECT

public:
    explicit ObjViewerForm();

private slots:
    void on_actionOpen_triggered(void);

private:
    Ui::ObjForm ui;
};

#endif
```

Listing 6.2: ObjViewerForm.h

In contrast with the first example the ObjViewerForm is a subclass of QMainWindow. This is necessary because we created a QMainWindow in the user interface designer.

The `Q_OBJECT` macro indicates that the MOC compiler should preprocess this header file.

The event of the file menu item (open menu) is handled by defining a method in the **private slots** part of the header. If the naming convention is followed then the event will automatically be routed into this method.

The naming convention is simple:

```
on_<name of ui element>_<name of event>
```

One final detail is that all though the user interface file is name `ObjViewerForm.ui`, the ui datamember in this header file has the type `Ui::ObjForm`. This corresponds to the name of the QMainWindow object in the UI designer (as show in figure 6.7.

ObjViewerForm.cpp

The implementation of the `ObjViewerForm` class has to setup the ui again, but since the top level element is a `QMainWindow`, the `setupUi` method now expects a `QMainWindow` object as parameter.

The `on_actionOpen_triggered` method will be called when the user clicks on the Open menu action.

The commented **lines 15-19** show a small demonstration of a message box which can be useful at times to debug applications.

Line 21 shows the usage of a `QFileDialog` to open a file, where also `QStandardPaths` is used to get the location of the home directory (user directory) of the current user. The `getOpenFileName` function returns a QString object with the location of the opened file (only if the user clicks Ok off course on the file dialog user interface). With this method it is also possible to specify a **file filter** that will only allow the selection of a file with extension **obj**.

Finally , **line 28-43** insert a row into the tableWidget object and also set an item into the table with the help of `QTableWidgetItem` object and a `QString`.

The include files (lines 2-5) are pretty sane and basically allow you to just type in the class name of the object you want to use.

```
1
2  #include "ObjViewerForm.h"
3  #include <QMessageBox>
4  #include <QFileDialog>
5  #include <QStandardPaths>
6
7  ObjViewerForm::ObjViewerForm()
8  {
9      ui.setupUi(this);
10 }
11
12
13 void ObjViewerForm::on_actionOpen_triggered()
14 {
15     /*
16     QMessageBox msgBox;
17     msgBox.setText("The document has been modified.");
18     msgBox.exec();
19     */
20
21     QString file = QFileDialog::getOpenFileName(this,
22         tr("Open Wavefront obj"),
23         QStandardPaths::writableLocation(QStandardPaths::HomeLocation),
24         tr("Obj Files (*.obj)"));
25
26     if (!file.isEmpty())
27     {
28         ui.tableWidget->insertRow(ui.tableWidget->rowCount());
29         ui.tableWidget->setItem(ui.tableWidget->rowCount() - 1,
30             0,
31             new QTableWidgetItem(QString("0")));
32
33         ui.tableWidget->setItem(ui.tableWidget->rowCount() - 1,
34             1,
35             new QTableWidgetItem(QString("[0,1,1]")));
36
37         ui.tableWidget->setItem(ui.tableWidget->rowCount() - 1,
38             2,
39             new QTableWidgetItem(QString("[0,1]")));
40
41         ui.tableWidget->setItem(ui.tableWidget->rowCount() - 1,
42             3,
43             new QTableWidgetItem(QString("[0,1,0]")));
44     }
45 }
```

Listing 6.3: ObjViewerForm.cpp

6.5 QOpenGLWidget example

Qt supports the usage of OpenGL, where it is possible to create a 3D scene programmatically. We will again create a `QMainWindow` widget with a Grid-Layout. One change is that we have to subclass the `QOpenGLWidget` to be able to make a scene.

UI

The main window is now empty save for a grid layout. The `QOpenGLWidget` will be subclassed as `DAEOpenGLWidget` with a spinning colored triangle and this widget will be added at runtime to the user interface (to the grid layout).

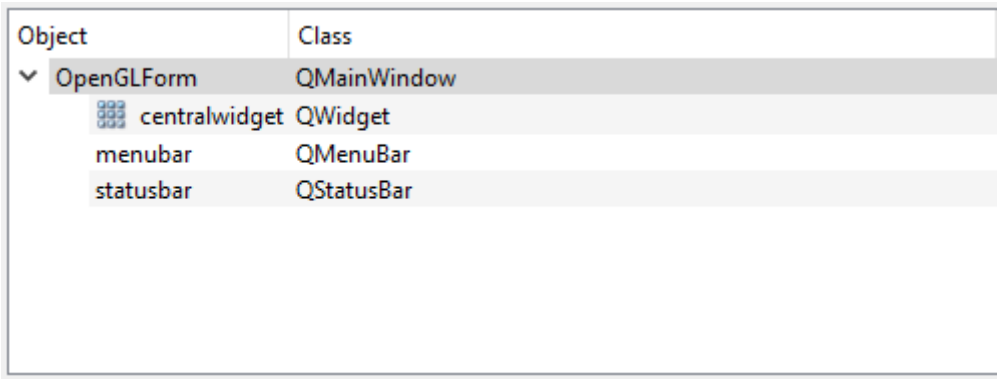


Figure 6.8: UI tree for the OpenGL example

CMake

There is a small addition to the CMake file, because we need the OpenGL functionality.

Line 14 and 15 find the extra packages that are needed to compile and run OpenGL examples, and **line 30 and 31** define the libraries (.lib) files that are needed to actually link to for the final executable.

Line 36 then also adds the two **dlls** that are necessary at runtime to run the executable.

```
1 cmake_minimum_required(VERSION 3.14)
2 project(ObjViewerForm LANGUAGES CXX)
3 set( QtBaseDir "c:/program files (x86)/Qt")
4 LIST(APPEND CMAKE_PREFIX_PATH "${QtBaseDir}/lib/cmake/")
5 set(CMAKE_INCLUDE_CURRENT_DIR ON)
6
7 set(CMAKE_AUTOMOC ON)
```

```
8  set(CMAKE_AUTORCC ON)
9  set(CMAKE_AUTOUIC ON)
10
11 find_package(Qt6 COMPONENTS Core)
12 find_package(Qt6 COMPONENTS Gui)
13 find_package(Qt6 COMPONENTS Widgets)
14 find_package(Qt6 COMPONENTS OpenGL)
15 find_package(Qt6 COMPONENTS OpenGLWidgets)
16
17 qt_add_executable(ObjViewerForm
18     DAEOpenGLWidget.h DAEOpenGLWidget.cpp
19     OpenGLForm.cpp OpenGLForm.h OpenGLForm.ui
20     main.cpp
21 )
22 set_target_properties(ObjViewerForm PROPERTIES
23     WIN32_EXECUTABLE TRUE
24     MACOSX_BUNDLE TRUE
25 )
26 target_link_libraries(ObjViewerForm PUBLIC
27     Qt::Core
28     Qt::Gui
29     Qt::Widgets
30     Qt::OpenGL
31     Qt::OpenGLWidgets
32 )
33
34 install(TARGETS ObjViewerForm DESTINATION bin)
35
36 set( DLLS "Qt6Core" "Qt6Gui" "Qt6Widgets" "Qt6OpenGL" "Qt6OpenGLWidgets")
37 foreach(dll IN LISTS DLLS )
38 install(FILES "${QtBaseDir}/bin/${dll}d.dll" DESTINATION bin)
39 endforeach()
40
41 install(FILES "${QtBaseDir}/plugins/platforms/qwindowsd.dll"
42         DESTINATION bin/platforms )
43
44 include(InstallRequiredSystemLibraries)
45 set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
46 set(CPACK_PACKAGE_VERSION_MAJOR "1")
47 set(CPACK_PACKAGE_VERSION_MINOR "0")
48
49 set(CPACK_NSIS_MODIFY_PATH ON)
50 include(CPack)
```

OpenGLForm.cpp

The OpenGLForm class is again derived from the QMainWindow base class. In the constructor a `DAEOpenGLWidget` is created which is added at runtime to the gridLayout object of the main window.

```
#include "OpenGLForm.h"
#include "DAEOpenGLWidget.h"
#include <QOpenglWidget>

OpenGLForm::OpenGLForm()
{
    ui.setupUi(this);
    DAEOpenGLWidget* widget = new DAEOpenGLWidget();
    ui.gridLayout->addWidget(widget);
}

OpenGLForm::~OpenGLForm()
{

}
```

Listing 6.4: OpenGLForm.cpp

DAEOpenGLWidget header

The DAEOpenGLWidget is a subclass of QOpenGLWidget and has to define 3 methods:

- `initializeGL` : initialize the OpenGL context for this window.
- `resizeGL` : deal with resized windows.
- `paintGL` : paint the open gl 3d or 2d primitives.

A `QScopedPointer` is a smart pointer that will delete the pointer it contains when the variable goes out of scope.

The `setClearColor` method is a convenience method to change the background color of the OpenGL window.

The extra parameters for the constructor define the speed of the rotation triangle and the rotation axis around which the triangle rotates.

```
#ifndef DAEOpenGLWidget_H
#define DAEOpenGLWidget_H

#include <QWidget>
#include <QOpenGLWidget>

#include <QVector3D>
#include <QScopedPointer>
class OpenGLWidgetPrivate;
class DAEOpenGLWidget : public QOpenGLWidget
{
public:
    DAEOpenGLWidget(int interval = 30,
      const QVector3D& rotAxis = QVector3D(0, 1, 0),
      QWidget *parent = nullptr);
    ~DAEOpenGLWidget();

    void initializeGL();
    void resizeGL(int w, int h);
    void paintGL();
    void setClearColor(const float* c);

private:
    QScopedPointer<OpenGLWidgetPrivate> d;
};

#endif // DAEOpenGLWidget_H
```

Listing 6.5: DAEOpenGLWidget.h

6.6 Exercise

6.6.1 Exercise 1

Build and run `w5\_exercise1.rar`

6.6.2 Exercise 2

Build and run `w5\_exercise2.rar`

1. When the users selects an obj file, clear the tableWidget and fill the rows with the data of the **optimized** obj vertices.
2. Add a second table to the user interface. This table will show all the indices but in groups of three (each row 3 triangle indices). This second table should be on the right side of the user interface.

6.6.3 Exercise 3

Build and run `w5\_exercise3.rar`

1. When the users selects an obj file, show this obj file as a 3D object in the OpenGLWidget. The mesh can be shown unshaded.
2. Add a color global variable to the shader and use per pixel lighting for the mesh.

6.6.4 Exercise 4

Add a menu item that exports the obj file as an ovm file. The menu item should be unavailable when no mesh has been loaded.

Chapter 7

Reading a Wavefront Obj file

7.1 Goal

This short chapter explains the reasoning and logic behind **normalizing** a 3d wavefront obj file. With the process of normalizing we mean that the vertex data (position, texture coordinates and normals) should be as minimal as possible.

To do this we need to find duplicate vertices in the list of faces and store these duplicates as one vertex. For this purpose an optimal solution will be discussed that is capable of finding the duplicate vertices in **linear time**. Finding duplicates can be done for example with a double for-loop, however in that case the complexity of the algorithm is O^2. One might also think to store the face vertices in a `map` object and store a list of duplicate vertices as a list (i.e. vector) in that same `map` object. However as the number of vertices grows, the performance of this solution also suffers.

The algorithm I present here was stumbled upon after encountering massive performance issues with the `map` solution. I hope it may provide food for thought, and I encourage you as reader to consider that an unoptimized good algorithm will always beat an optimized bad algorithm.

7.2 Example obj file

A small obj file that represents a quad is used to demonstrate the concept:

```
1  v 2 0 0
2  v 0 2 0
3  v 0 2 2
4  v -2 0 2
5
6
7  vt 0 0
8  vt 1 0
9  vt 0 1
10 vt 1 1
11
12 vn 0 1 0
13
14 f 1/1/1 2/2/1 3/3/1
15 f 3/3/1 2/2/1 4/4/1
```

Listing 7.1: Simple obj file

This obj-file results in the following quad:

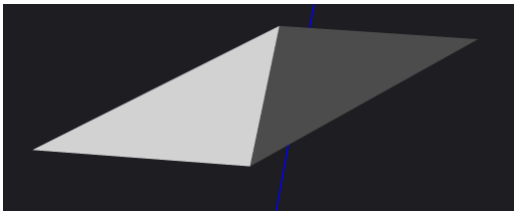


Figure 7.1: A simple quad

As an end result we want to create a vertex buffer with a minimal number of vertices. From the obj-file we see that it should be possible to create a vertex buffer with four vertices. All the vertices have the same normal, but different positions and texture coordinates.

Optimal solution

Before we start optimizing let's take a look at what the optimal solution should look like:

index	*Position*			*Tex coord*		*Normal*			*Face*
0	2	0	0	0	0	0	1	0	1/1/1
1	0	2	0	0	1	0	1	0	2/2/1
2	0	2	2	1	0	0	1	0	3/3/1
3	-2	0	2	1	1	0	1	0	4/4/1

Notable here is that the vertex buffer does not contain duplicate information.

We can now look up the faces in the original face definition to create the list of triangles:

- 0 1 2
- 1 2 3

Unoptimized solution

A naive approach would be to create a vertex buffer with the information in the face definitions (line 14 and 15) as is.

```
1 f 1/1/1 2/2/1 3/3/1
2 f 3/3/1 2/2/1 4/4/1
```

Listing 7.2: Face definitions

The resulting vertex buffer is then :

index	*Position*			*Tex coord*		*Normal*			*Face*
0	2	0	0	0	0	0	1	0	1/1/1
1	0	2	0	0	1	0	1	0	2/2/1
2	0	2	2	1	0	0	1	0	3/3/1
3	0	2	2	1	0	0	1	0	3/3/1
4	0	2	0	0	1	0	1	0	2/2/1
5	-2	0	2	1	1	0	1	0	4/4/1

And the triangle list is:

- 0 1 2
- 3 4 5

In this vertex buffer, vertex 1 and 3 are perfectly identical, as are vertex 2 and 4, so that seems like an inefficient use of bytes.

So this poses us two problems:

1. **Merge** all vertices that are exactly the same. We take a closer look at the merging process to find the best algorithm for this task.
2. Remember the triangle that the vertices came from. One vertex can now be reused in multiple triangles, so to build the indexed triangle list we need to keep track of the original triangle the vertex originated from.

7.3 Merge algorithm

We take a new look at the **unoptimized** vertex buffer and try to work out a way to find the duplicates. This implies that we have a well defined **equality operation** for two vertices and that we have an efficient way to **find** a duplicate in a **list** of vertices.

Equality of vertices

To find duplicates we must also discuss when two vertices are **equal**. It is true that two vertices are equal when their coordinates, texture coordinate and normal coordinates are equal. This means however, that we need to perform **8 floating point comparisons** per equality operation! The equal comparison for floats itself can also be problematic in its own right, but it can work here because the floats will be generated from the same textual representation so the bit pattern should indeed be the same.

There is indeed a simpler way to perform the equality operation. The *Face* column in the following table has the information about the position index, texture coordinate index and normal index. These indices are **integers** and **comparing integers** is off course fool proof.

index	*Position*			*Tex coord*		*Normal*			*Face*
0	2	0	0	0	0	0	1	0	1/1/1
1	0	2	0	0	1	0	1	0	2/2/1
2	0	2	2	1	0	0	1	0	3/3/1
3	0	2	2	1	0	0	1	0	3/3/1
4	0	2	0	0	1	0	1	0	2/2/1
5	-2	0	2	1	1	0	1	0	4/4/1

For example if we compare vertex one with vertex four, we compare the face information of the two vertices to check if they are equal:

```
2/2/1 == 2/2/1 ?
```

Finding duplicate vertices

The equal operation is not enough, we must now also be able to find the duplicate vertices in a list so we can **merge** them. Here we must be careful to take the **algorithmic** complexity into account.

A first thought could be to start with vertex 0 and **iterate** over the list to find duplicates of this vertex with the equality operation as defined in the previous paragraph.

Next, start with vertex 1 and **iterate** over the list from that vertex to the end and find the duplicates of that vertex.

Next, start with vertex 2 and ... you hopefully get the point.

The problem with this method is that the algorithmic complexity is comparable to the algorithmic complexity of **bubblesort**, which is $O(n^2)$. This means that it will work reasonably well for small vertex buffers, but the performance will be abysmal.

Speaking of **sort algorithms**, let's see what would happen if we sort the *Face* information, in **ascending** order, first by position **index**, then by texture coordinate **index**, and then by normal **index**:

Position			*Tex coord*		*Normal*			*Face*
2	0	0	0	0	0	1	0	1/1/1
0	2	0	0	1	0	1	0	2/2/1
0	2	0	0	1	0	1	0	2/2/1
0	2	2	1	0	0	1	0	3/3/1
0	2	2	1	0	0	1	0	3/3/1
-2	0	2	1	1	0	1	0	4/4/1

After this sort operation, the vertices with equal *Face* information will be next to each in the sorted list, which means that finding duplicates has become a lot easier.

To find duplicates we can now iterate over the list, check the *Face* information of the previous vertex and see if it matches or not. That's it! We can now easily merge all equal vertices in the list.

The last remaining problem is to create a correct triangle list from this merged vertex buffer. We will discuss this problem in the next paragraph.

7.4 Merging and creating the triangle list

To create the triangle list we need to keep track of the original triangle that each vertex came from, otherwise this information is lost after sorting.

To do this we first create a C++ struct with the *Face* information and the triangle information:

```
/**
* vi : vertex index in m_ObjVertices
* ni : normal index in m_ObjNormals
* tci : texcoord index in m_ObjTexcoords
* ti : the index of the triangle
* iti : the vertex index in the triangle itself (i.e. 0,1,2)
*/
struct facevertex {
  int vi, tci, ni, ti, iti;
};
```

As stated in the comment of this struct, the face information is represented by `vi/tci/ni`. The original triangle information is represented by `ti` and `iti`, where `ti` is the index of the triangle and `iti` is the index of the vertex **within** the triangle.

The reader now needs to create a list (`std::vector`) of facevertex values. This list just stores the face definitions as they are encountered in the `obj` file. The triangle information just records the number of the triangle and the index of the vertex within that triangle.

Face			*Triangle*	
v_i	tc_i	n_i	t_i	it_i
1	1	1	0	0
2	2	1	0	1
3	3	1	0	2
3	3	1	1	0
2	2	1	1	1
4	4	1	1	2

The next step is to **sort** this list ascending by v_i first, then tc_i and then n_i, resulting in the following sorted facevertex vector:

Face			*Triangle*	
v_i	tc_i	n_i	t_i	it_i
1	1	1	0	0
2	2	1	0	1
2	2	1	1	1
3	3	1	0	2
3	3	1	1	0
4	4	1	1	2

We can now iterate over this facevertex list **one time** to simultaneously build the vertex and index buffer for this mesh. We emulate what this for loop does in the following steps.

C++ code

The C++ code for the algorithm is the following:

```
1  bool compareFace(const facevertex& f1 ,const facevertex& f2) {
2    if (f1.vi == f2.vi) {
3      if (f1.tci == f2.tci) {
4        return (f1.ni < f2.ni); // f1 : 2/2/3 and f2 : 2/2/4
5      }
6      else {
7        return f1.tci < f2.tci; // f1 : 2/3/4 and f2 : 2/7/4
8      }
9    }
10   else {
11     return f1.vi < f2.vi;
12   }
13 }
14
15 void ObjReader::BuildOutputMesh() {
16   // first sort the m_ObjFVertices by vertex index, texture index and normal index
17   std::sort(m_ObjFVertices.begin(), m_ObjFVertices.end(), compareFace);
18   int currentVertexIndex = -1;
19   int fvi = -1; // current face vertex index;
20   int fti = -1; // current texcoord index
21   int fni = -1; // current normal index
22   // same number of triangles as the m_ObjFVertices
23   m_OIndexBuffer.resize(m_ObjFVertices.size());
24   for (auto const& face : m_ObjFVertices) {
25     if (face.vi != fvi || face.tci != fti || face.ni != fni) {
26       fvi = face.vi;
27       fti = face.tci;
28       fni = face.ni;
29       m_OVertexBuffer.push_back(m_ObjVertices[fvi-1]);
30       m_OTexcoordBuffer.push_back(m_ObjTextcoords[fti-1]);
31       m_ONormalBuffer.push_back(m_ObjNormals[fni-1]);
32       ++currentVertexIndex;
33     }
34     int fIndex = face.ti * 3 + face.iti;
35     m_OIndexBuffer[fIndex] = currentVertexIndex;
36   }
37 }
```

The compareFace method compares two facevertex objects for sorting, first by vertex index, then by texcoord index and the by normal index.

On **line 17** this method is used to sort the facevertex objects that were read from the obj file.

Line 25 contains the simple equality check that was discussed earlier.

The variable `m_ObjVertices`, `m_ObjTexcoords` and `m_ObjNormals` are `std::vector` objects that contain the data that was read from the `obj` file.

In the following paragraphs a further explanation of the `BuildOutputMesh` function is presented.

Step 0

We set the following values all to `-1`.

```
int currentVertexIndex = -1;
int fvi = -1; // current face vertex index;
int fti = -1; // current texcoord index
int fni = -1; // current normal index
```

The **currentVertexIndex** is set to -1 to ensure that the first valid vertex index is zero. The variable `fvi`, `fti` and `fni` are set to -1 to ensure that the first equality check in the loop fails.

The final output **vertex buffer** is still **empty** in this step:

index	*Position*	*Tex coord*	*Normal*

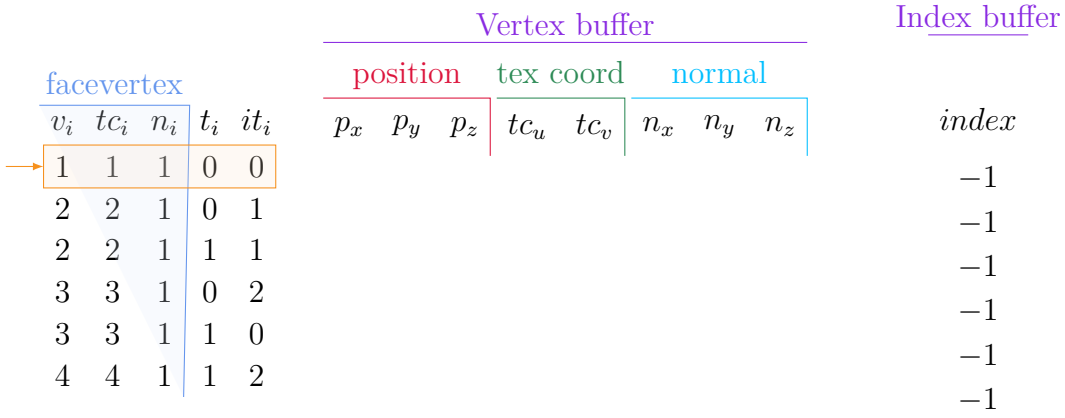
And the **index buffer** is off course also **empty**:

index
-1
-1
-1
-1
-1
-1

We know the size of this list because the output will contain the same number of triangles as the input `obj` file.

Step 1

In this first step, we compare the v_i, tc_i and n_i with the current indices, which were set to -1 in Step 0.

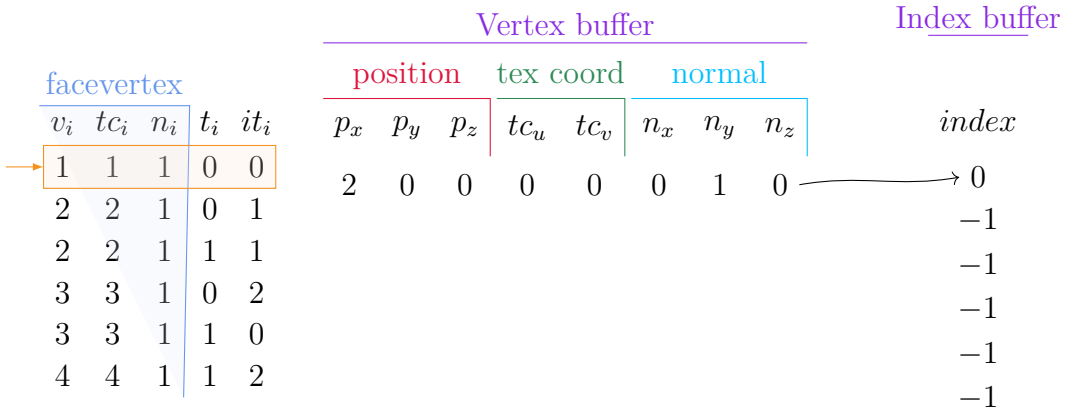


As a result we can say that this is a new combinations of indices, so we change currentVertexIndex and store this information in the vertex buffer at this new position:

```
currentVertexIndex = currentVertex+1;
```

With $t_i = 0$ and $it_i = 0$ the index into the IndexBuffer is:

```
currentIndexBufferIndex = 3 * ti + iti; // 0
indexBuffer[currentIndexBufferIndex] = currentVertexIndex;
```

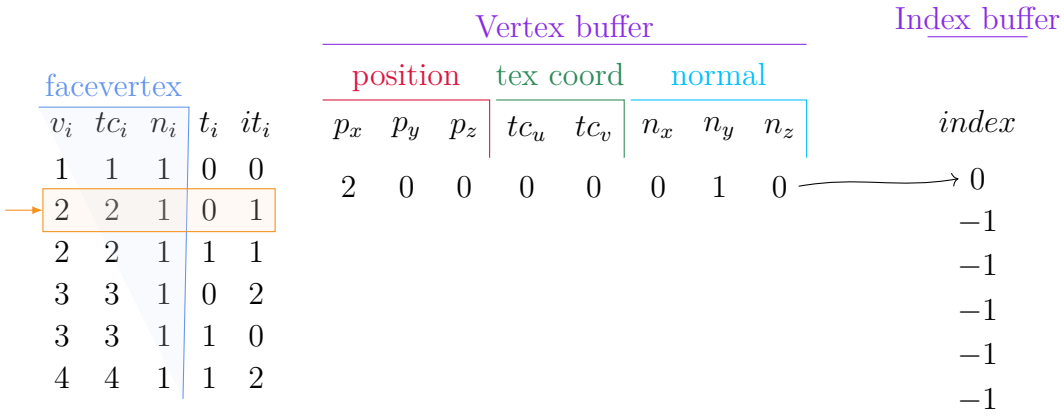


After this step we set the current face indices to the correct values:

```
fvi = 1; // current face vertex index;
fti = 1; // current texcoord index
fni = 1; // current normal index
```

Step 2

We start with the end result of the previous step, but now the current row of the facevertex list is the second row:

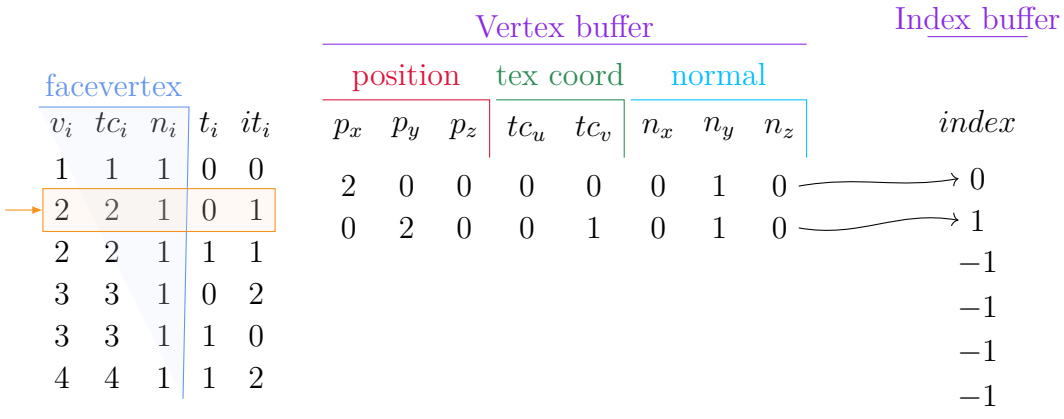


We again have a new combination of face indices:

```
currentVertexIndex = currentVertex+1; // 1
```

With $t_i = 0$ and $it_i = 1$ the index into the IndexBuffer is:

```
currentIndexBufferIndex = 3 * ti + iti; // 1
indexBuffer[currentIndexBufferIndex] = currentVertexIndex;
```

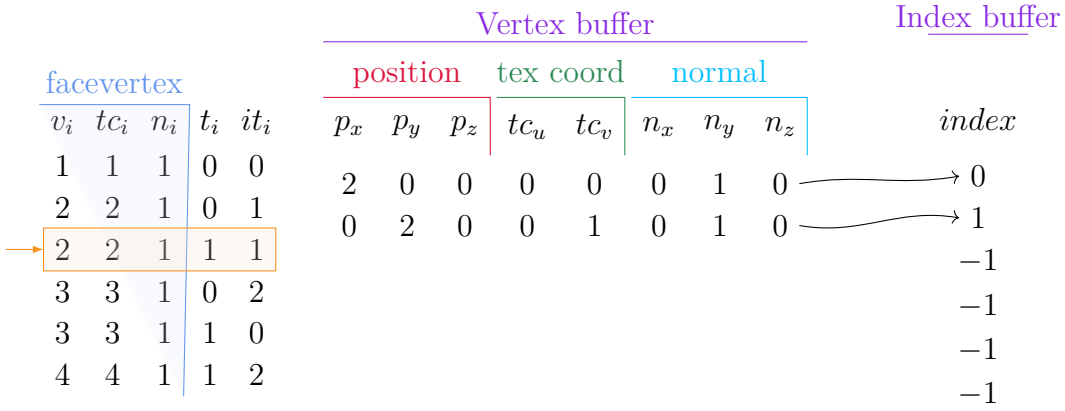


After this step we set the current face indices to the correct values:

```
fvi = 2; // current face vertex index;
fti = 2; // current texcoord index
fni = 1; // current normal index
```

Step 3

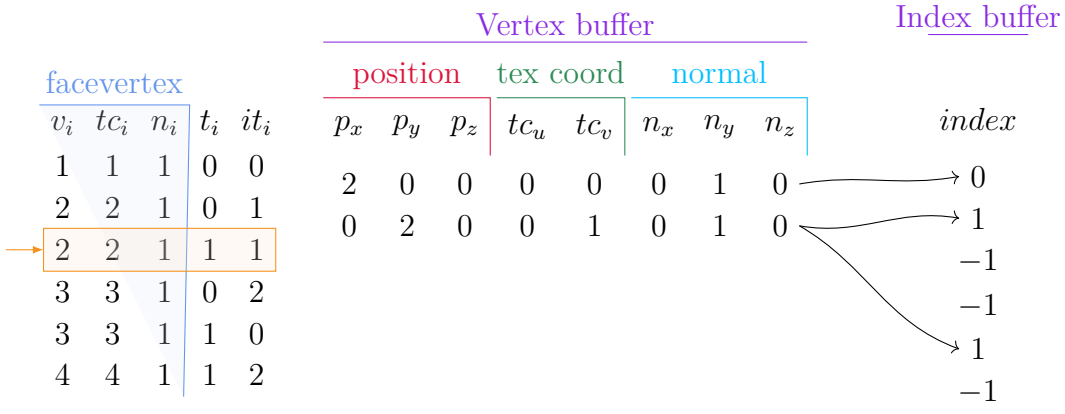
We start with the end result of the previous step, but now the current row of the facevertex list is the third row:



The previous faces indices were 2/2/1 and now we again have 2/2/1. This is not a new combination so we do **not update** the current vertex index, and we do **not** store new information into the vertex buffer.

However, this facevertex was part of a triangle with $t_i = 1$ and $it_i = 1$, and we must update the indexbuffer at the calculated index **4** with the currentVertexIndex (still 1).

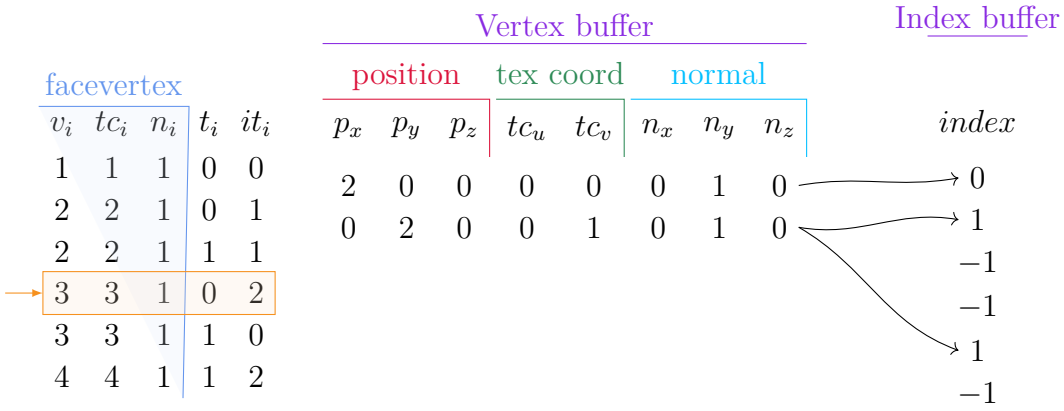
```
currentIndexBufferIndex = 3 * ti + iti; // 4
indexBuffer[currentIndexBufferIndex] = currentVertexIndex;
```



The `fvi` ,`fti` and `fni` variables remain the same.

Step 4

We start with the end result of the previous step, but now the current row of the face vertex list is the fourth row:

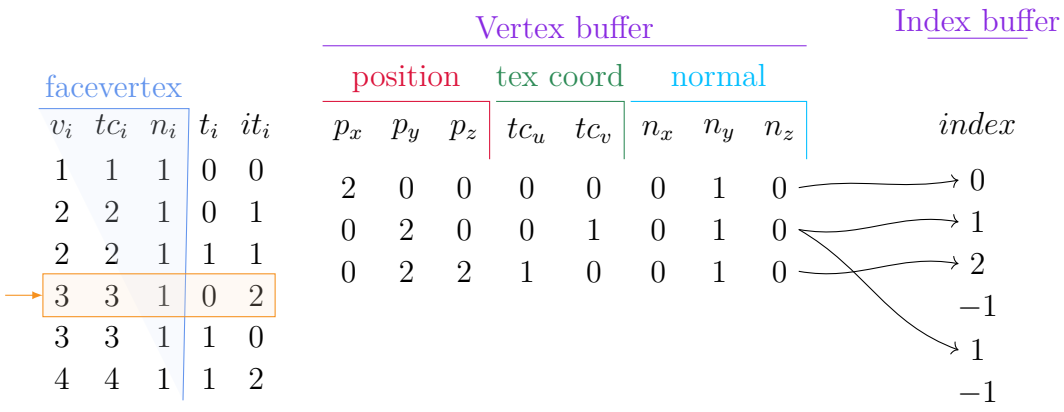


The new combination of indices is 3/3/1 which is different from the previous combination 2/2/1 , so we update the currentVertexIndex and store the correct information in the vertex buffer:

```
currentVertexIndex = currentVertex+1; // 2
```

With $t_i = 0$ and $it_i = 2$ the index into the IndexBuffer is:

```
currentIndexBufferIndex = 3 * ti + iti; // 2
indexBuffer[currentIndexBufferIndex] = currentVertexIndex;
```

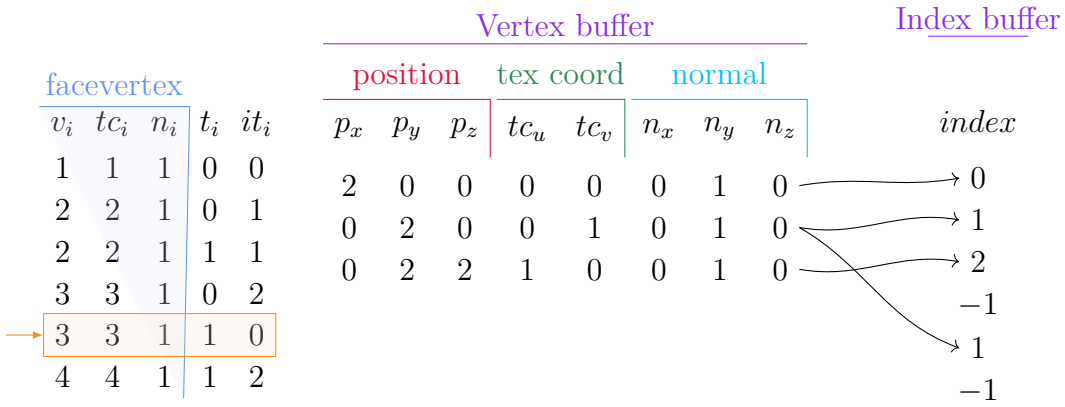


The indices have changed so we must store them to compare in the next step:

```
fvi = 3; // current face vertex index;
fti = 3; // current texcoord index
fni = 1; // current normal index
```

Step 5

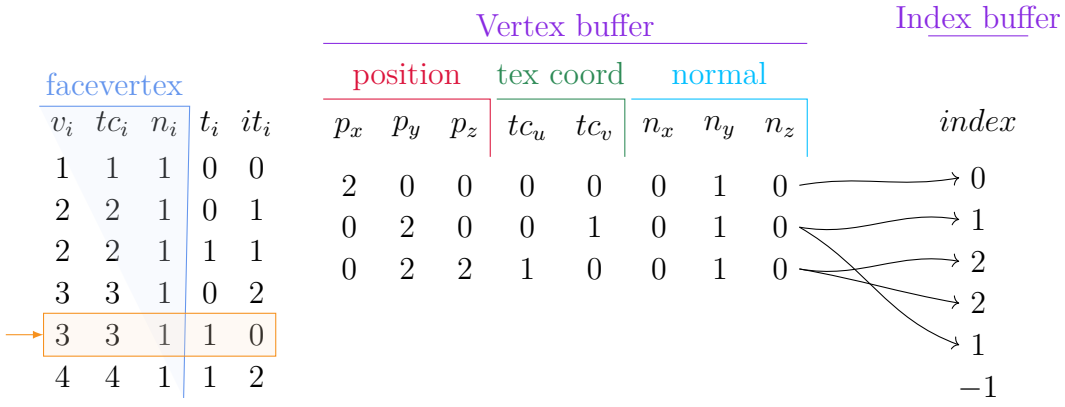
We start with the end result of the previous step, but now the current row of the face vertex list is the fifth row:



The new combination of indices is 3/3/1 which is equal to the old combination , so we do **not** update the currentVertexIndex and its value is still **2**.

We do need to update the index buffer so we calculate the index into the index buffer with the help of the t_i and it_i values:

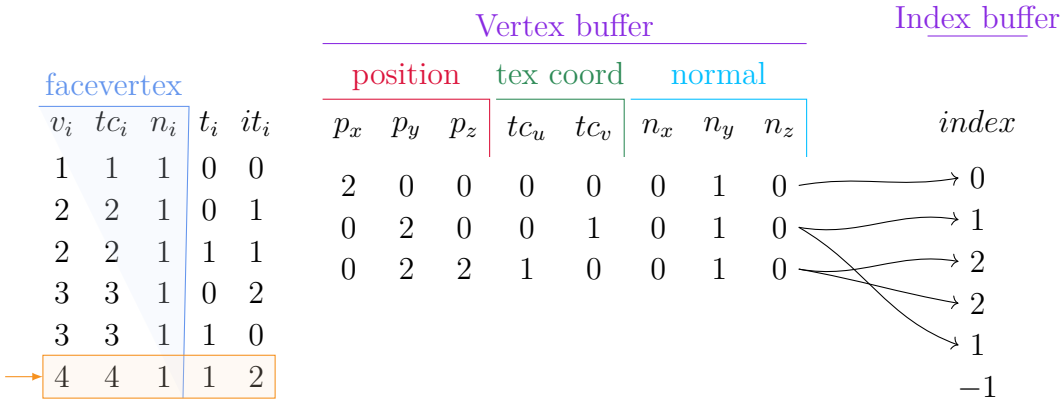
```
currentIndexBufferIndex = 3 * ti + iti; // 3
indexBuffer[currentIndexBufferIndex] = currentVertexIndex;
```



The face indices have not changed, so they stay at 3/3/1.

Step 6

We start with the end result of the previous step, but now the current row of the face vertex list is the sixth row:



The new combination of indices is 4/4/1 which is different from the previous combination 3/3/1 , so we update the currentVertexIndex and store the correct information in the vertex buffer:

```
1 currentVertexIndex = currentVertex+1; // 3
```

With $t_i = 0$ and $it_i = 2$ the index into the IndexBuffer is:

```
1 currentIndexBufferIndex = 3 * ti + iti; // 5
2 indexBuffer[currentIndexBufferIndex] = currentVertexIndex;
```