

CGA 2 Framework User Guide

Inhaltsverzeichnis

Das Framework	3
Installation	3
GitHub Repository	3
CMake Projekt	4
Projekt Struktur	4
Builden von Projekten	4
Anlegen von neuen Source Files im CMake-generierten Visual Studio Projekt	10
Öffnen des CMake Projektes mit QtCreator	11
Source Files in QtCreator hinzufügen	14
QtCreator: Debugger für MSVC hinzufügen (bei MinGW nicht nötig)	15
CMake Projekt mit Visual Studio direkt öffnen (nur VS2017)	18
Source Files zu CMake/Visual Studio hinzufügen	20
Build-Konfigurationen (Was sind Debug und Release?)	22
Benutzung des Frameworks	23
Das Basisfenster benutzen	23
Einstellungen für das Fenster	25
init(), shutdown(), render() und update()	26
Maus- und Tastatureingaben	27
Reagieren auf Verkleinerung/Vergrößerung des Fensters	30
Start des Programms / main.cpp	31
Beenden des Programms	31
OBJ-Dateien laden	32
OBJLoader Klassen und Datentypen	34
Verwenden der Daten	36
Nachbearbeitung	36
Texturen laden	37
Schnelles OpenGL Error Checking	39
Einstellungen für GLERR und checkglerror()	41
Grafikdebugger: RenderDoc	42
Links zu Tutorials, Referenzen, Bibliotheken,	49

Das Framework

Mit dem Framework wollen wir euch für die ersten Projekte einen einfachen Startpunkt bieten. Es erzeugt ein OpenGL Fenster, bietet Funktionen um auf Maus- und Tastatureingaben zu reagieren, OBJ Dateien und Texturen zu laden. In z.B. `update()` und `render()` Funktionen könnt ihr dann euren eigenen Code einhängen, wobei ihr dann native OpenGL Funktionen (`glDrawElements(...)` etc.) benutzt um eure Kreationen auf den Bildschirm zu rendern (mehr dazu später).





Installation

GitHub Repository

Link zum Repository: <https://github.com/FabianFriederichs/CGA2>

Ihr könnt das Repository entweder clonen, forken und dann clonen um eine Kopie des Repository für euer eigenes Projekt zu verwenden, oder einfach als .zip herunterladen.

Im Repository sind dann einige Ordner:

 CMakeProject	02.05.2017 10:24	Dateiordner
 Doc	26.04.2017 13:54	Dateiordner
 LibraryBuilds	30.04.2017 01:51	Dateiordner
 Preconfigured	26.04.2017 16:55	Dateiordner

Falls ihr Visual Studio 2013, 2015 oder 2017 benutzen wollt, findet ihr im Ordner *Preconfigured* fertig konfigurierte Projekte. Man kann leider nicht garantieren, dass diese Projekte auf Anhieb beschwerdefrei laufen (z.B. liegt die System-Bibliotheksdatei „`opengl32.lib`“ unter Windows oft nicht da wo sie liegen sollte...). Meistens kann man diese Probleme recht schnell beheben. Schaut euch dazu den Abschnitt „QtCreator: Debugger für MSVC hinzufügen“ an. Die fehlenden Dateien kann man über das Windows SDK nachinstallieren.

Falls gar nichts hilft, gibt es das CMake Projekt.

Unter *LibraryBuilds* findet ihr vorkompilierte Bibliotheksdateien für die Visual Studio Compiler MSVC2013, MSVC2015 und MSVC2017, jeweils in 32bit und 64bit.

Im Ordner *Doc* liegt diese User Guide.

Wenn ihr Visual Studio nicht benutzen wollt oder nicht auf Windows unterwegs seid, empfiehlt sich die IDE *QtCreator*. Damit könnt ihr das CMake Projekt direkt öffnen.

Hier kann man *QtCreator* herunterladen: <https://www.qt.io/ide/>

Die kostenlose Visual Studio 2017 Community Edition findet ihr hier: <https://www.visualstudio.com/de/downloads/>

Unter Linux müsst ihr mit hoher Wahrscheinlichkeit ein paar Dependencies nachinstallieren. Auf Debian-basierten Systemen (Ubuntu, Mint, ...) reicht es meist das Paket `xorg-dev` zu installieren:
`sudo apt-get install xorg-dev`

CMake Projekt








CMake ist ein Cross-Platform Build Tool. Es erzeugt Projektdateien für verschiedene IDEs oder einfach gute alte Makefiles.

Um das Projekt nutzen zu können müsst ihr CMake $\geq 3.2.0$ installieren.

<https://cmake.org/download/>

Projekt Struktur

Das Projekt enthält wiederum einige Ordner und Dateien:

 assets	03.05.2017 11:07	Dateiordner	
 bin	03.05.2017 11:07	Dateiordner	
 build	03.05.2017 11:07	Dateiordner	
 framework	03.05.2017 11:07	Dateiordner	
 libs	03.05.2017 11:09	Dateiordner	
 src	03.05.2017 11:09	Dateiordner	
 CMakeLists.txt	02.05.2017 00:53	Textdokument	9 KB

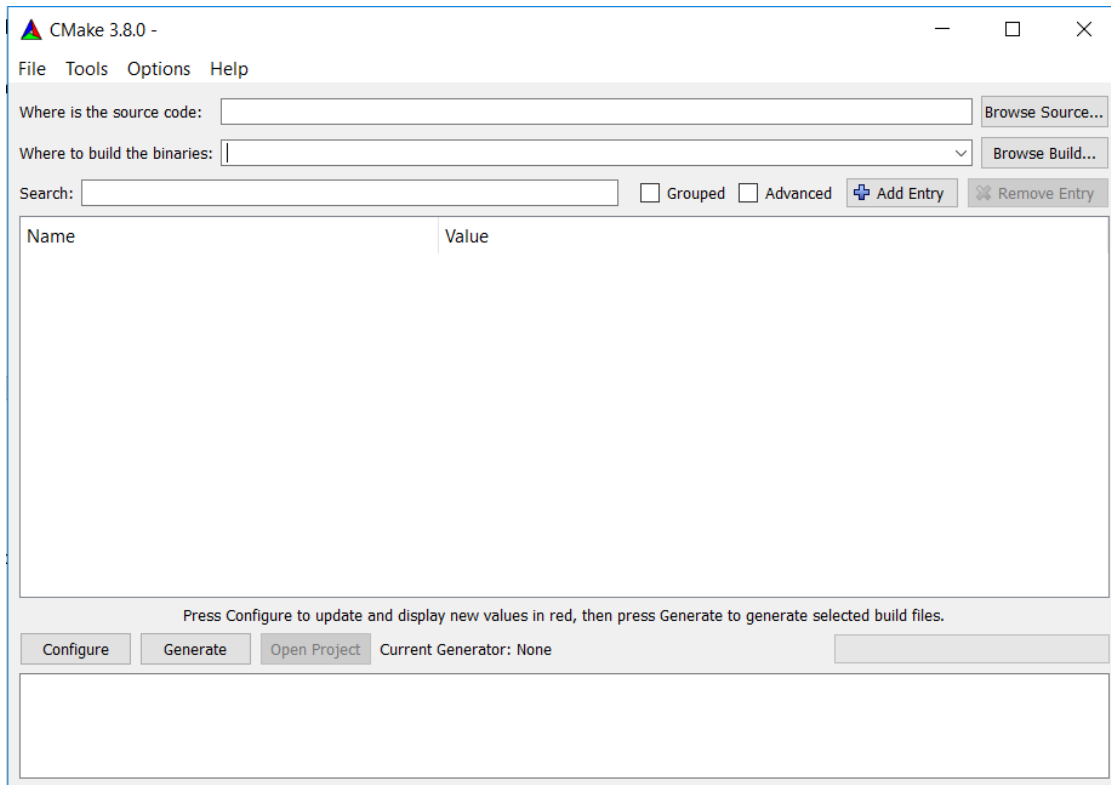
- assets:** Hier könntet ihr später eure Assets (Models, Texturen, Szenenbeschreibungen etc.) ablegen.
- bin:** Hier kommen standardmäßig die fertigen, ausführbaren Programme hin. Das könnt ihr aber auch noch in der DIE ändern, falls ihr es anders haben wollt.
- build:** Nur als Vorschlag: Hier könntet ihr das Projekt „hinbuilden“, dann habt ihr alles in einem Ordner.
- framework:** Hier befinden sich .cpp und .h Dateien in denen die Funktionalität des Frameworks umgesetzt ist.
- libs:** Hier befinden sich benötigte Bibliotheken, aber auch ein paar Bibliotheken die noch nicht eingebunden sind. Wenn ihr später mehr Funktionen braucht (zum Beispiel zum Laden von .fbx Dateien: Hier würde sich die Bibliothek „AssImp“ eignen.) könnt ihr diese auch noch einbinden.
- src:** Hier sollte später euer eigener Sourcecode liegen, damit alles richtig funktioniert. Neue Dateien, egal ob direkt in src oder in Unterordnern werden automatisch dem Projekt hinzugefügt.
- CMakeLists.txt:** Die Hauptprojektdatei. Aus dieser generiert CMake Projekte und Makefiles.

Builden von Projekten

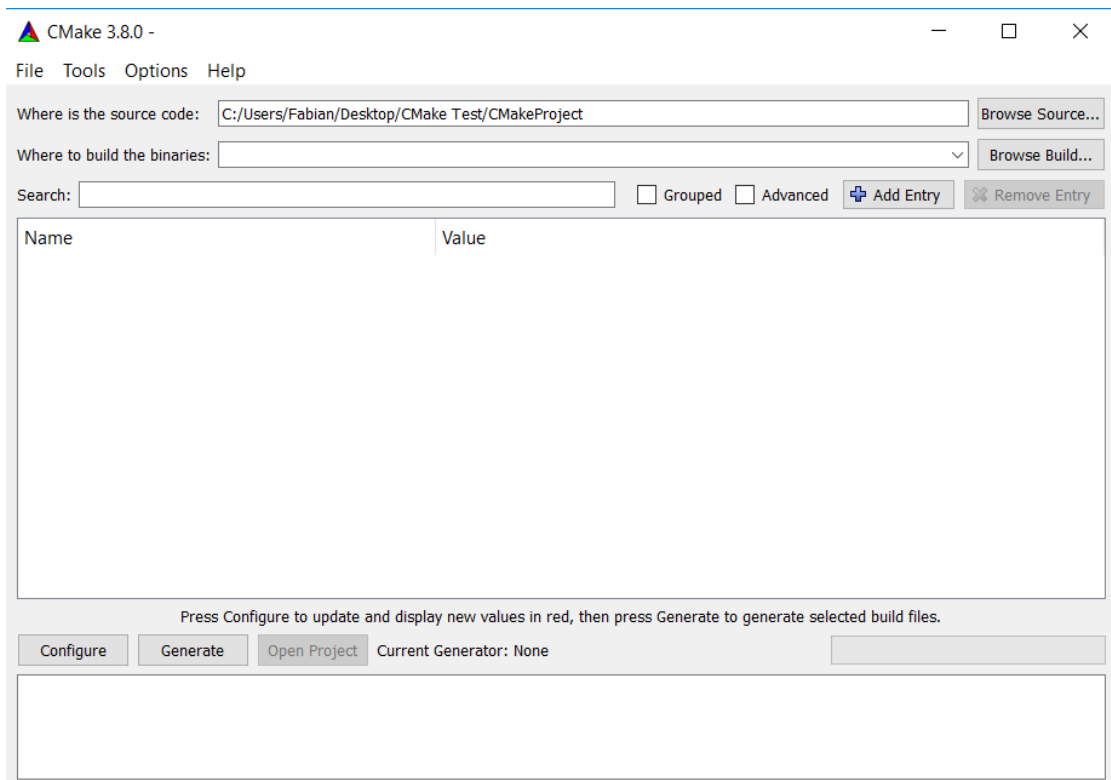
Am einfachsten lässt sich das mit der grafischen Benutzeroberfläche von CMake erledigen. Unter Windows sollte diese nach der Installation irgendwo im Startmenü sein. Unter Linux befindet sich im Installationsordner von CMake eine ausführbare Datei namens „cmake-gui“.

Als Beispiel hier einmal das Builden eines Visual Studio 2017 Projektes mit CMake:

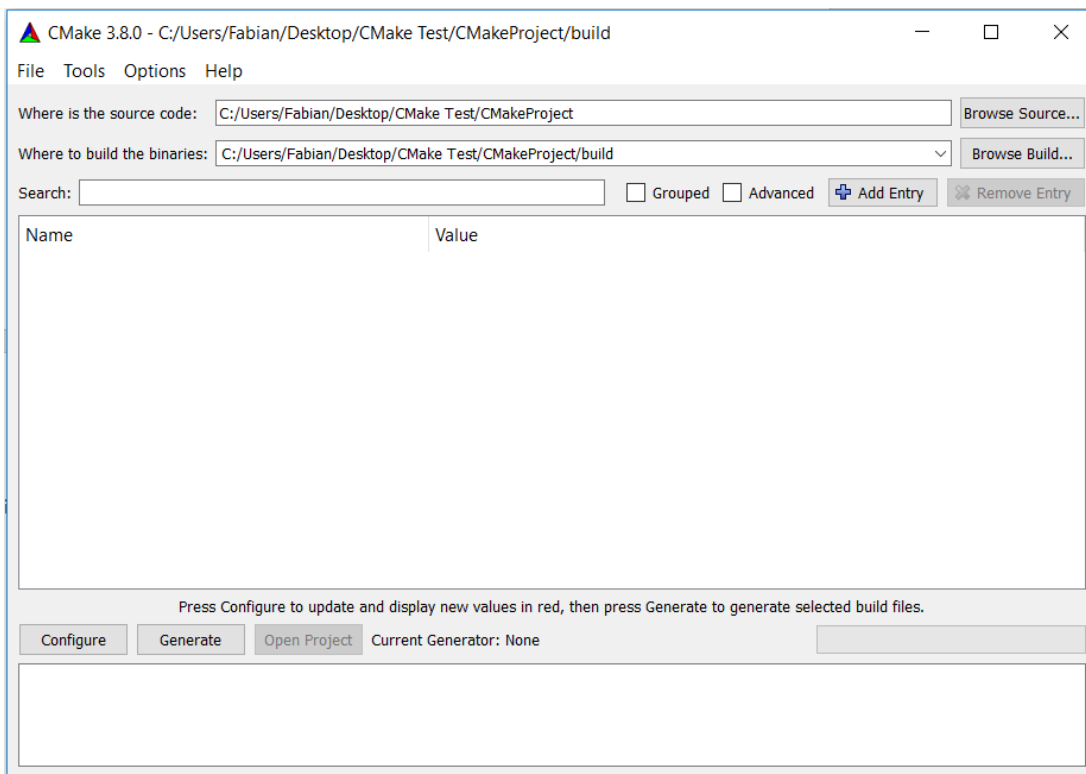
1. cmake-gui starten:



2. *Browse Source...* anklicken und den Ordner, der die CMakeLists.txt Datei enthält auswählen:

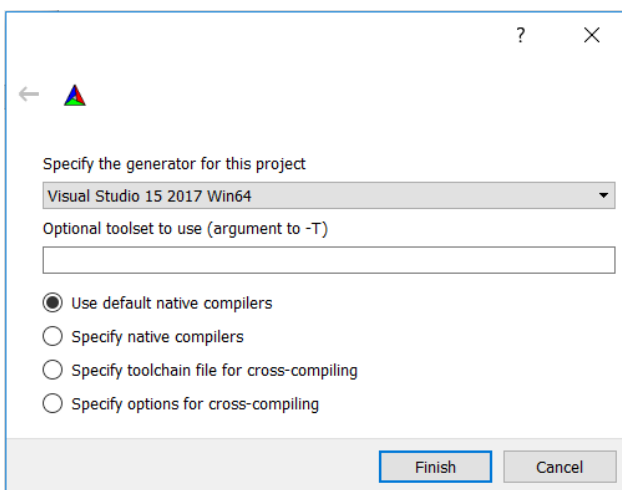


3. *Browse Build...* anklicken und einen Ordner auswählen, in dem das Projekt generiert werden soll:



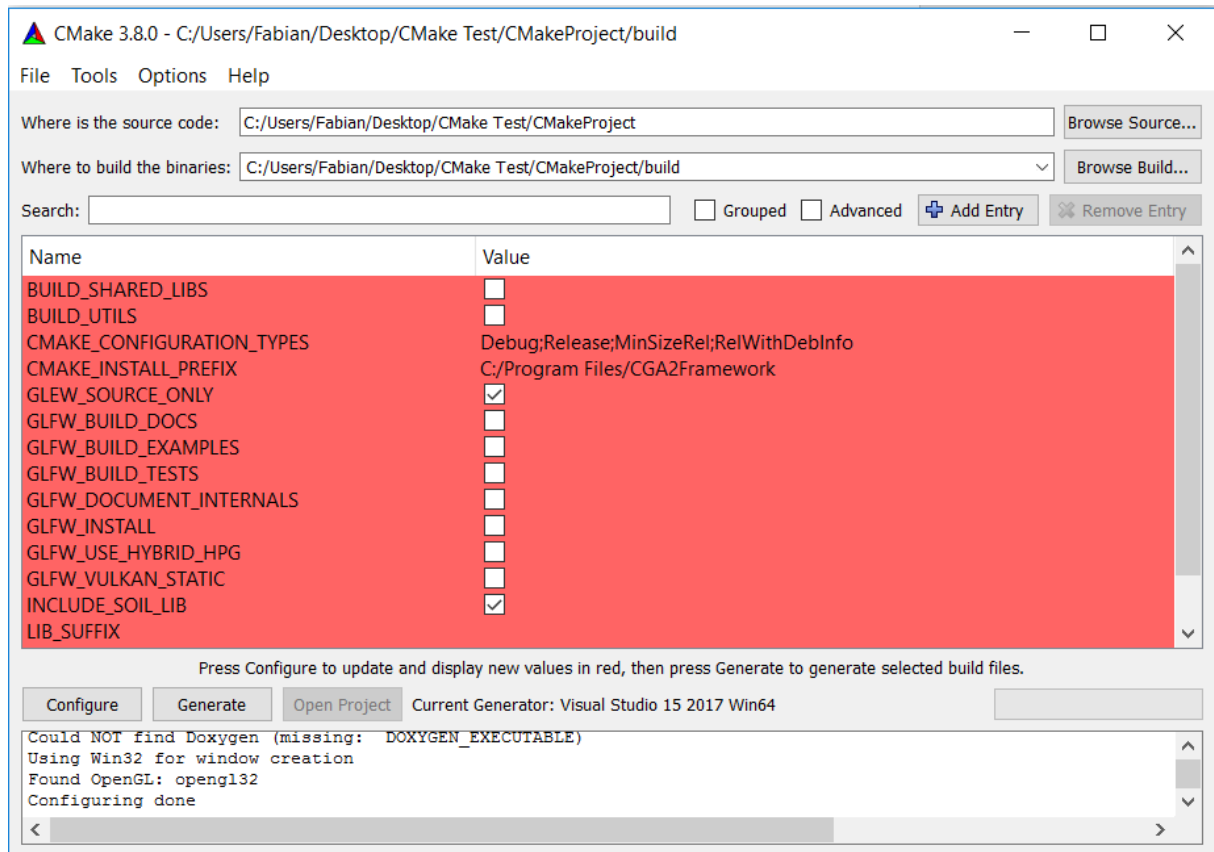
In diesem Fall haben wir den (zunächst leeren) Ordner „build“ im Projektordner ausgewählt. Das kann sinnvoll sein, wenn man die generierte Projektdatei mit dem ganzen Projektordner auf ein Repository pushen möchte.

4. *Configure* anklicken. Darauf hin öffnet sich ein Dialog. Hier den gewünschten Projekttyp auswählen. Wir wählen hier Visual Studio 2017 in der 64-bit Version, da wir ein 64-bit Projekt bauen wollen:



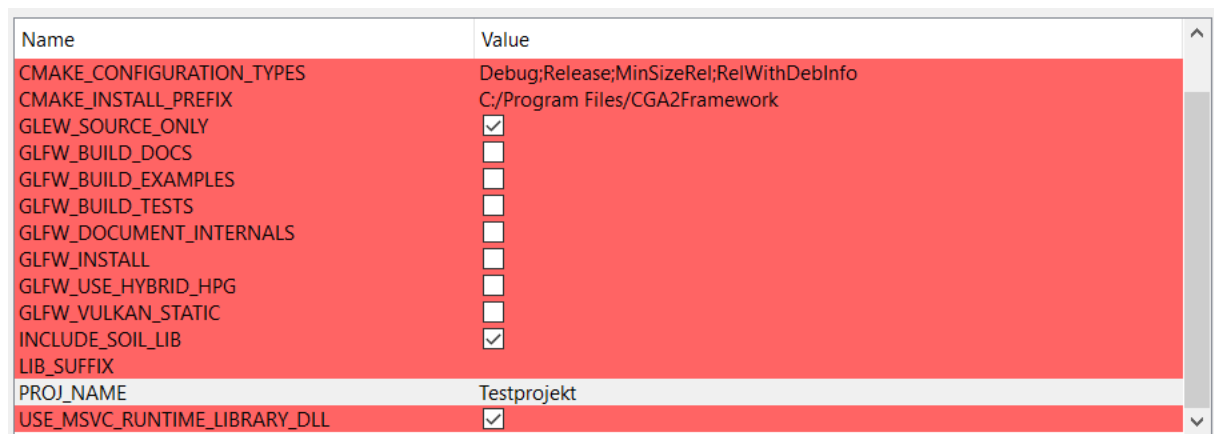
Danach einfach mit *Finish* bestätigen.

5. Jetzt dauert es einen Moment und wenn alles glatt gegangen ist steht unten im Log „Configuring done“. Wenn ein Fehler aufgetreten ist teilt CMake das mit und im Log erscheinen rote Fehlermeldungen. Wenn Fehler aufgetreten sind kann das verschiedene Ursachen haben. Meistens fehlen bestimmte Dependencies, z.B. OpenGL, Glu oder Sonstiges. Diese muss man dann einfach nachinstallieren und nochmal *Configure* drücken.



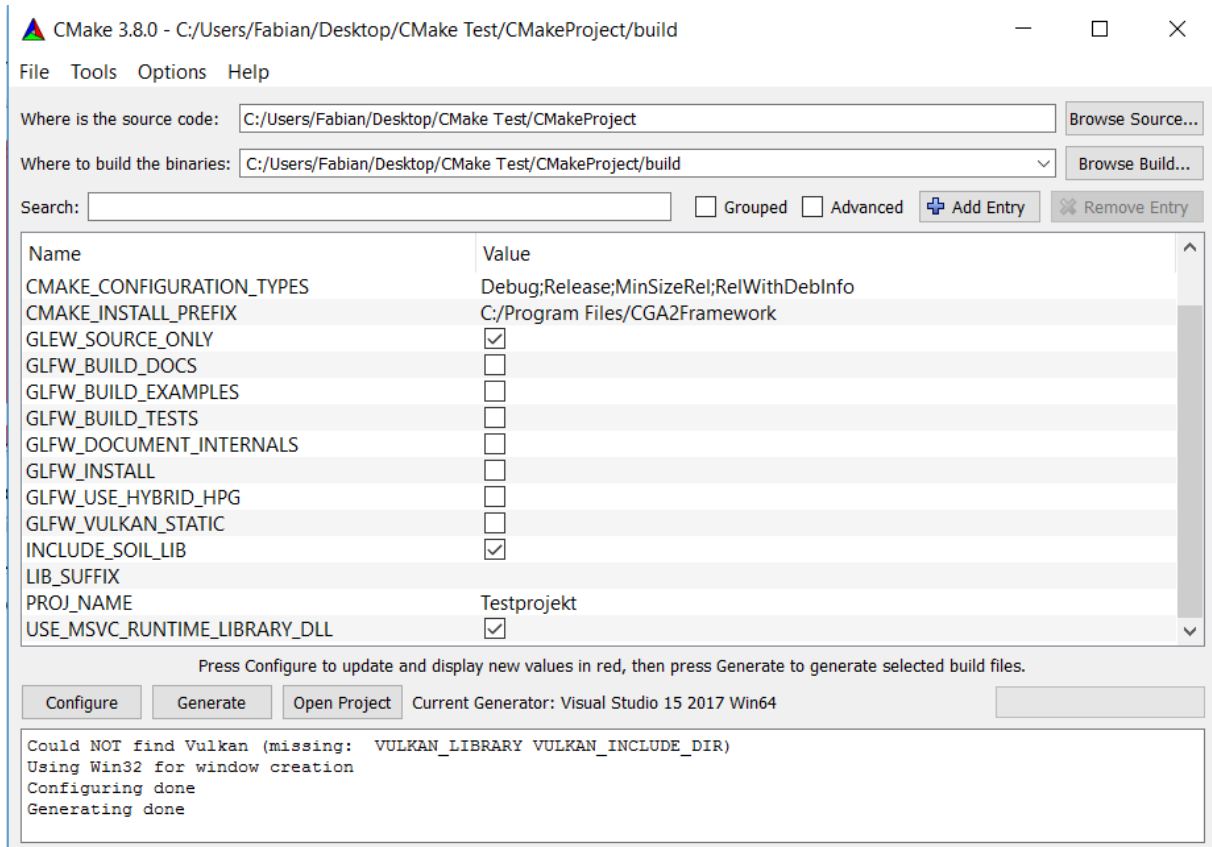
Jetzt erwartet CMake ein paar Einstellungen. Normalerweise kann man alles lassen wie es ist.

Weiter unten könnt ihr noch den Projektnamen ändern, falls ihr das wollt:



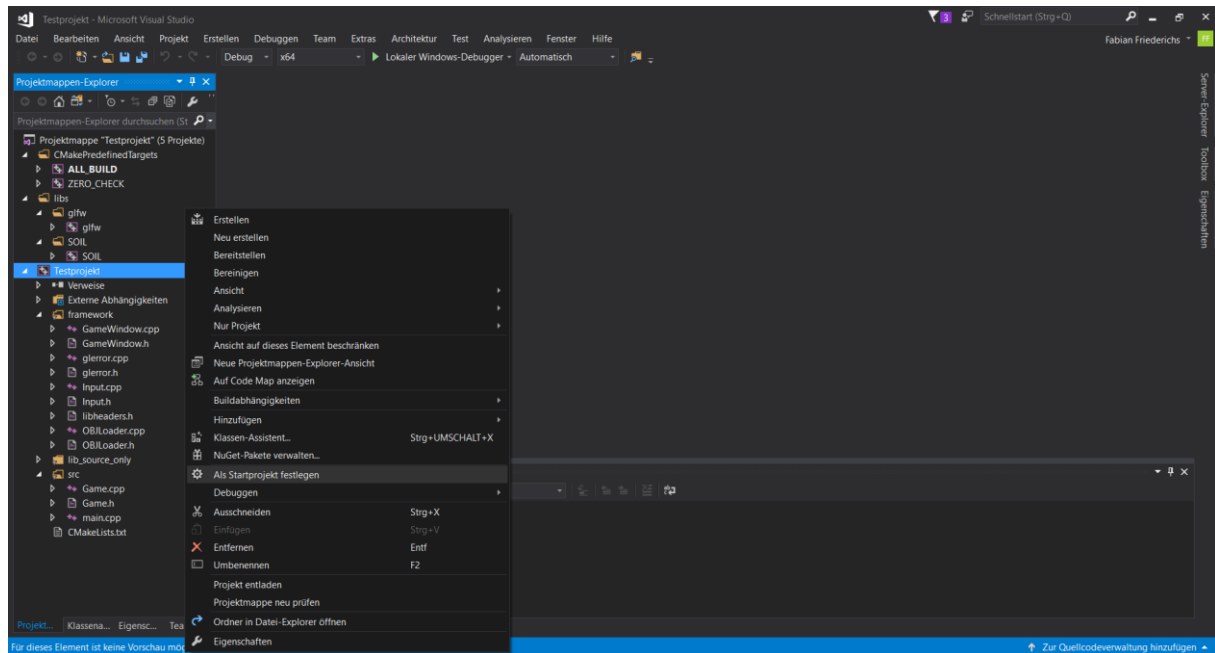
Wenn ihr zufrieden mit den Einstellungen seid klickt ihr nochmal auf *Configure*. Jetzt sollten die roten Zeilen verschwunden sein.

6. Jetzt könnt ihr auf *Generate* klicken. Damit wird das Projekt erstellt und befindet sich im oben angegebenen Build-Ordner.

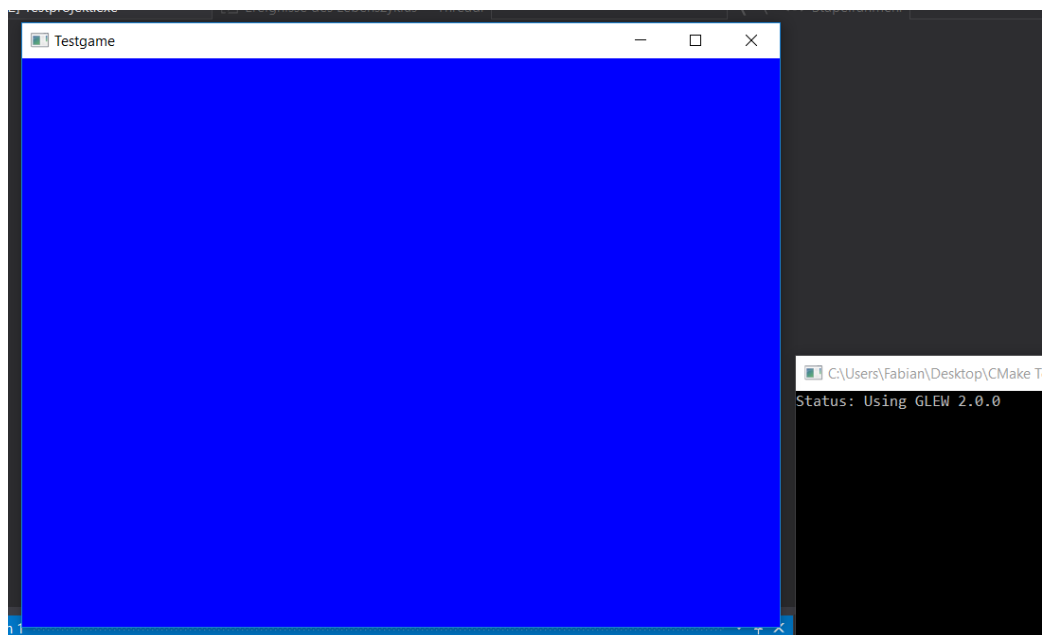


Wenn alles glatt ging steht unten im Log „Generating done“.

Das Projekt kann nun geöffnet werden. In Visual Studio kann das z.B. so aussehen:



Um die Projekte „ALL_BUILD“, „ZERO_CHECK“, „glfw“ und „SOIL“ muss man sich für gewöhnlich gar nicht kümmern. Diese wurden durch CMake erstellt und dienen zum Angleichen von CMake Projekt und Visual Studio Projekt bzw. zum automatischen Kompilieren von ein paar benötigten Bibliotheken. Rechtsklickt euer Projekt, in unserem Fall „Testprojekt“ und wählt *Als Startprojekt festlegen*. Damit wird, wenn man das Projekt startet (grüner Pfeil) nicht das „ALL_BUILD“ Projekt gestartet sondern das „richtige“ Projekt. Wenn alles funktioniert hat, das Projekt erfolgreich kompiliert und das Programm gestartet wurde solltet ihr ein blaues Fenster sehen, das mit Escape beendet werden kann. Unter „framework“ liegen wieder die Framework Source Files und unter „src“ eure Dateien.

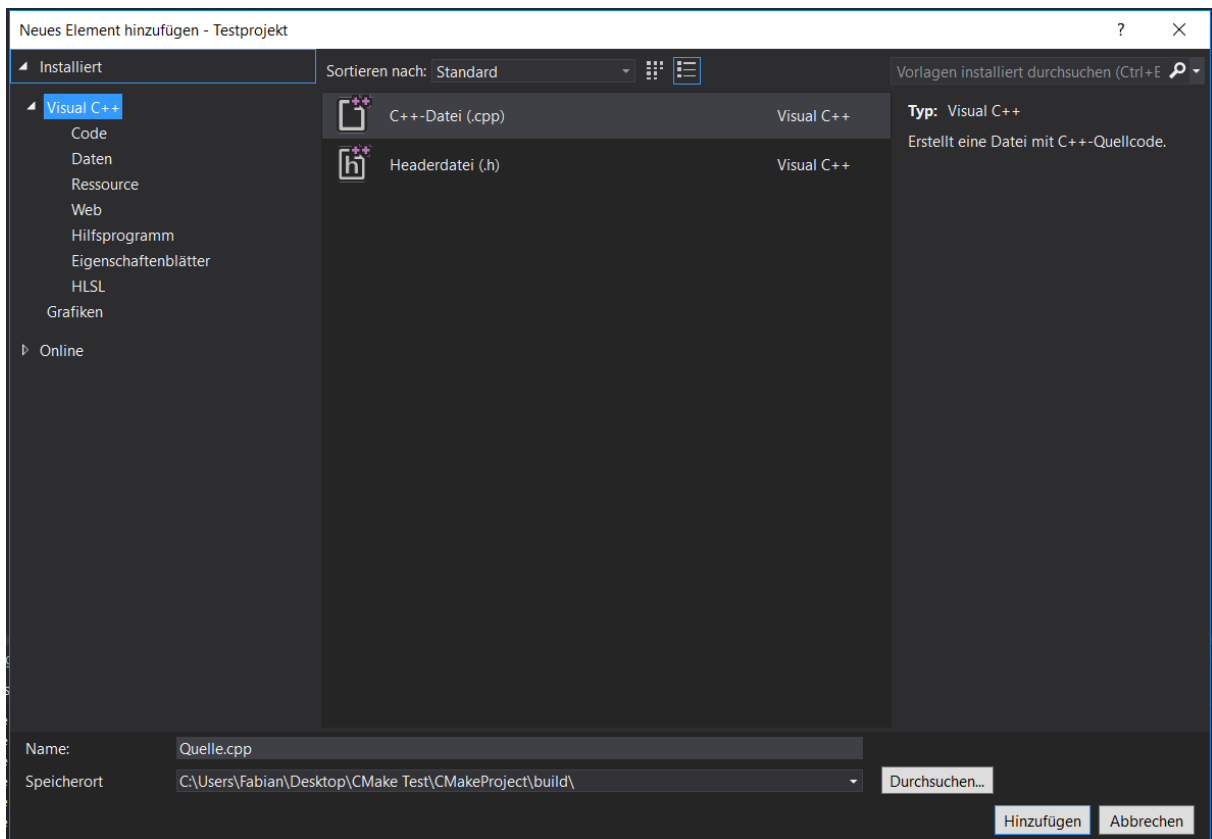


Unten in der Ecke seht ihr das zum Programm gehörende Konsolenfenster. Dort gehen auch alle Textausgaben hin.

Anlegen von neuen Source Files im CMake-generierten Visual Studio Projekt

Wenn ihr kein vorkonfiguriertes Visual Studio Projekt verwendet, sondern eines mit CMake erstellt, muss man noch ein paar Eigenarten von Visual Studio betrachten.

Zum Anlegen von neuen Source Files könnte man den von Visual Studio gewohnten Weg gehen: Rechtsklick auf „src“ -> Hinzufügen -> Neues Element -> etc. pp.



Wenn ihr das tut, solltet ihr darauf achten, dass ihr unten bei „Speicherort“ den „src“ Ordner aus dem CMake Projekt angebt (oder einen Unterordner davon, wenn das hilft die Übersicht zu behalten). Sonst werden beim Nächsten, der das CMake Projekt benutzt, die Dateien nicht automatisch zum Projekt hinzugefügt.

Belastet ihr CMake bei den gleichen Einstellungen wie beim Builden des Projektes und klickt erneut *Generate* werden eure Source Files auch wieder in den „src“ Filter des Visual Studio Projektes sortiert.

Wenn euch das CMake Projekt egal ist könnt ihr es natürlich auch einfach als Basis für euer Projekt benutzen und den Tipp ignorieren. Allerdings solltet ihr dann auf keinen Fall noch einmal *Generate* drücken, da dann die neuen Dateien zwar im Visual Studio Projektordner liegen aber nicht mehr im Projekt „registriert“ sind. Diese müsstet ihr dann manuell wieder hinzufügen, damit sie wieder angezeigt werden.

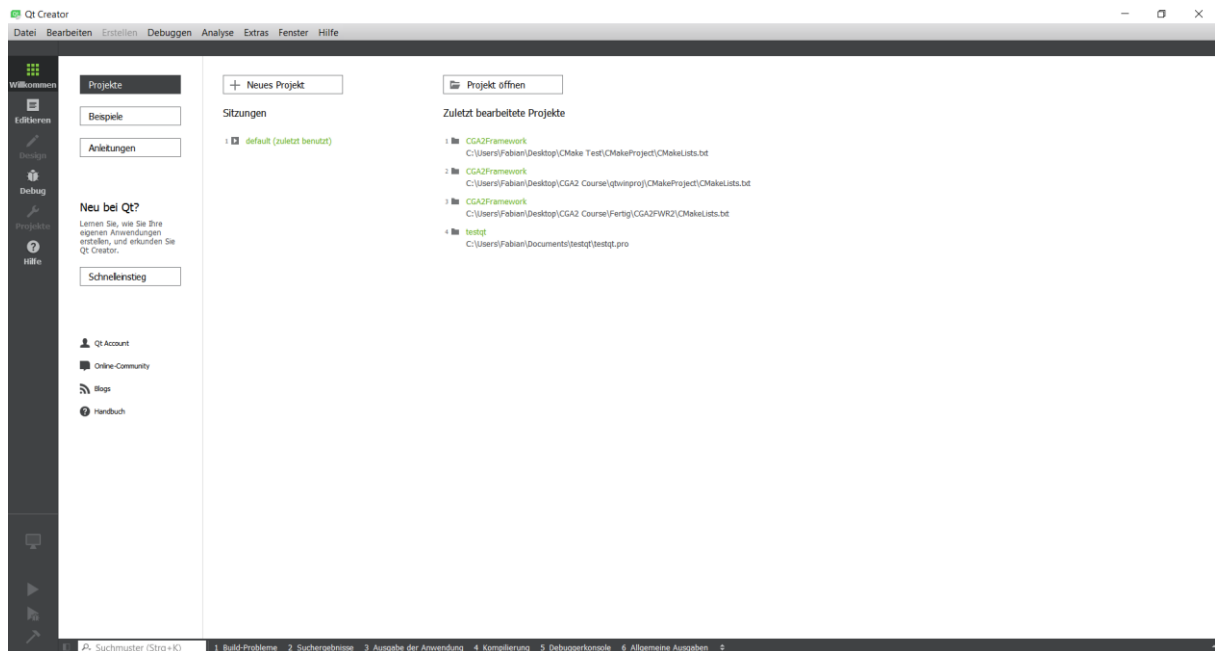
Eine letzte Eigenheit ist das Problem mit dem Hinzufügen von Klassen. Wenn ihr Hinzufügen -> Klasse wählt, könnt ihr nicht auswählen, wo .h und .cpp gespeichert werden sollen. Die Dateien werden

standardmäßig im Visual Studio Projektordner abgelegt. Beim Weitergeben des CMake Projektes müsstet ihr dann entweder die Dateien in den „src“ Ordner kopieren, oder aber .h und .cpp einzeln hinzufügen.

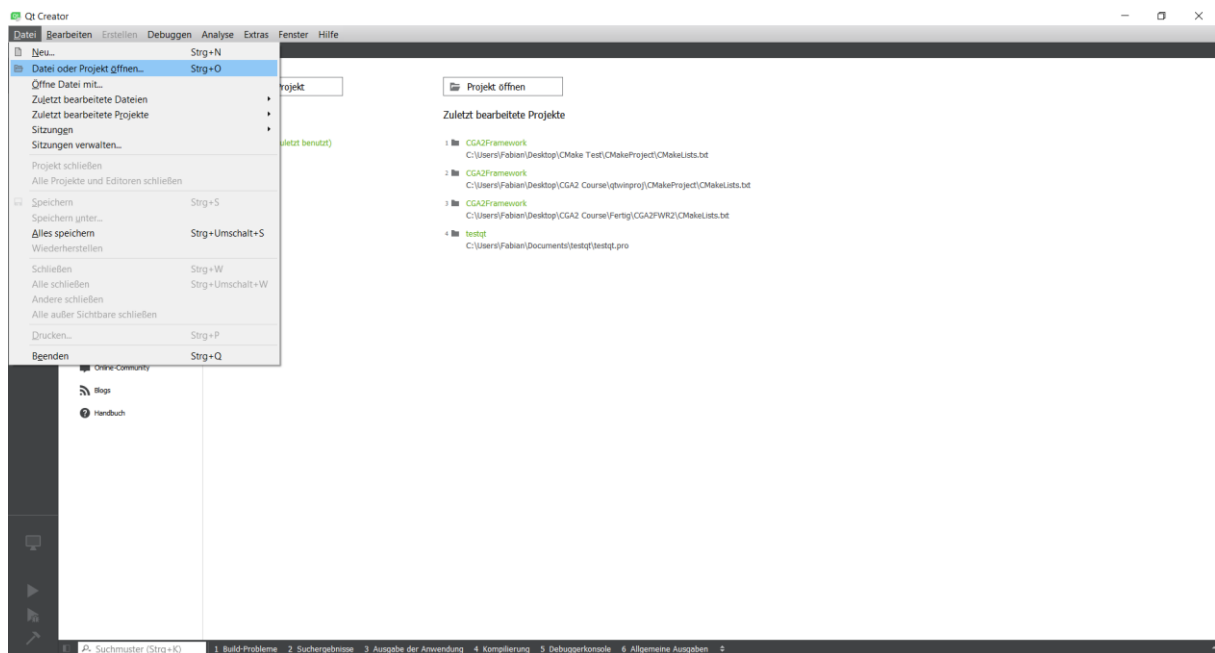
Öffnen des CMake Projektes mit QtCreator

Das Projekt kann auch direkt mit QtCreator geöffnet werden.

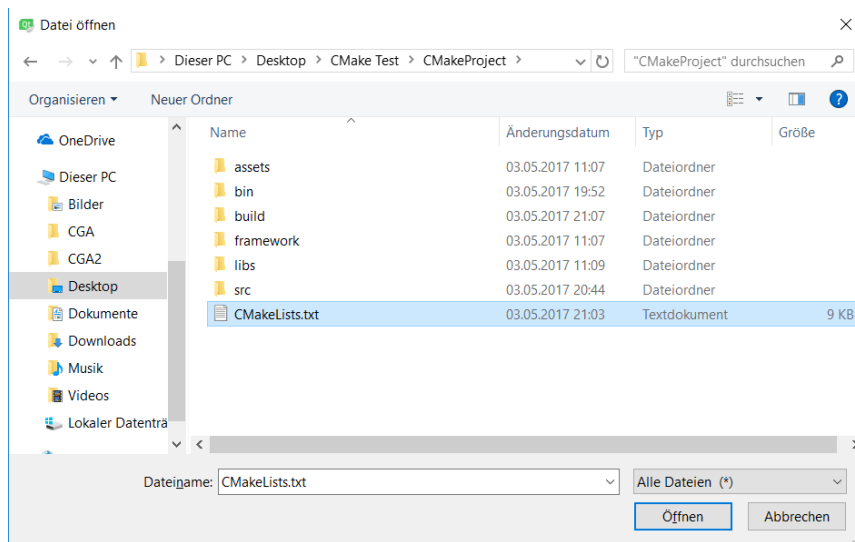
1. QtCreator starten:



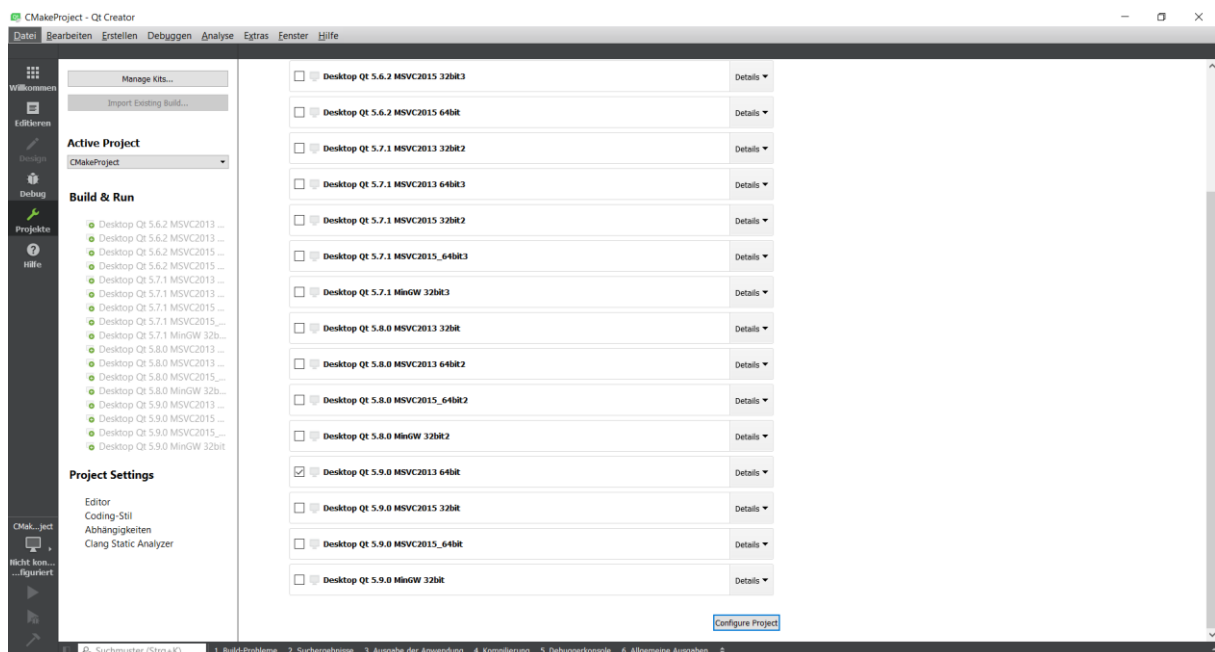
2. Datei -> Datei oder Projekt öffnen...



3. CMakeLists.txt Datei auswählen:



4. Wählt das Build Kit aus, welches ihr verwenden möchtet und klickt auf *Configure Project*:

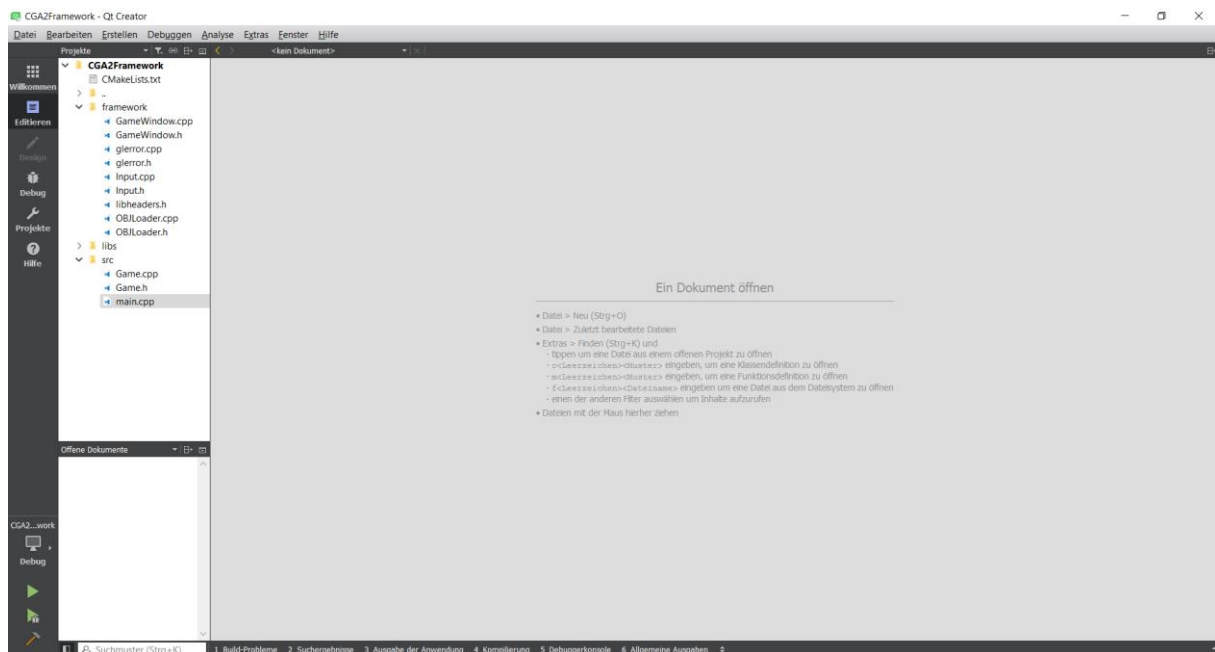


Der Name des Kits verrät unter anderem welcher Compiler verwendet wird, ob das Projekt als 32- oder 64-bit Anwendung kompiliert wird und welche Version der Qt Library zur Verfügung gestellt wird. Am besten wählt man das Kit mit dem neuesten Compiler, der auf dem System verfügbar ist.

Wenn ihr vorhabt große Mengen an Daten zu laden, wählt die 64-bit Version, damit ihr auch mal mehr als 2GB Arbeitsspeicher in eurem Programm benutzen könnt.

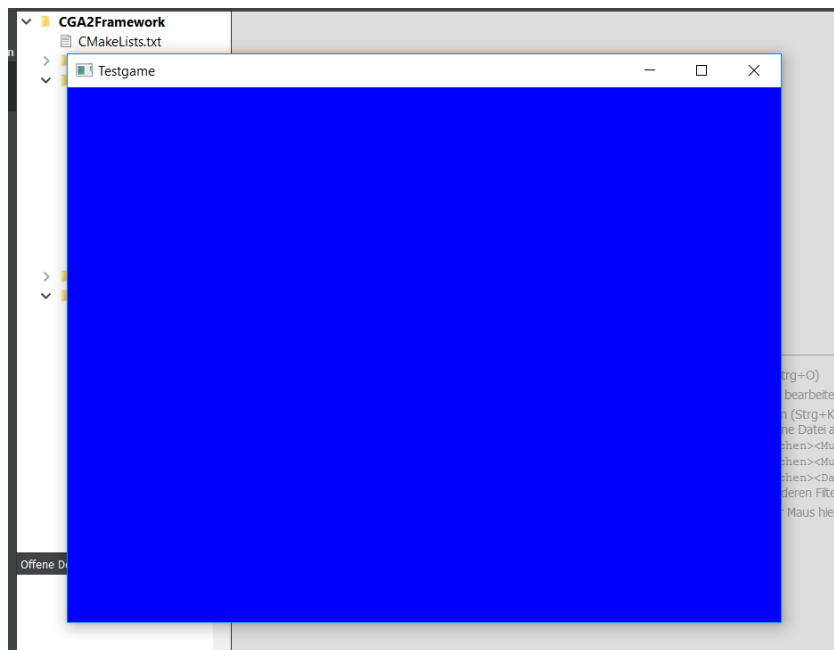
Bei den ersten Versuchen ist der Arbeitsspeicher nämlich schneller weg als man denkt, und wer sich fragt: „Warum 2GB? Mit 32-bit Adressen kann man doch 4GB adressieren!“ -> Zumindest unter Windows reserviert sich das Betriebssystem schon mal 2GB eures Adressraumes für Kernel Aufgaben.

Nach einer Weile ist das Projekt geladen und sieht in etwa so aus:



An der Seite erkennt man wieder die Ordner „framework“ und „src“.

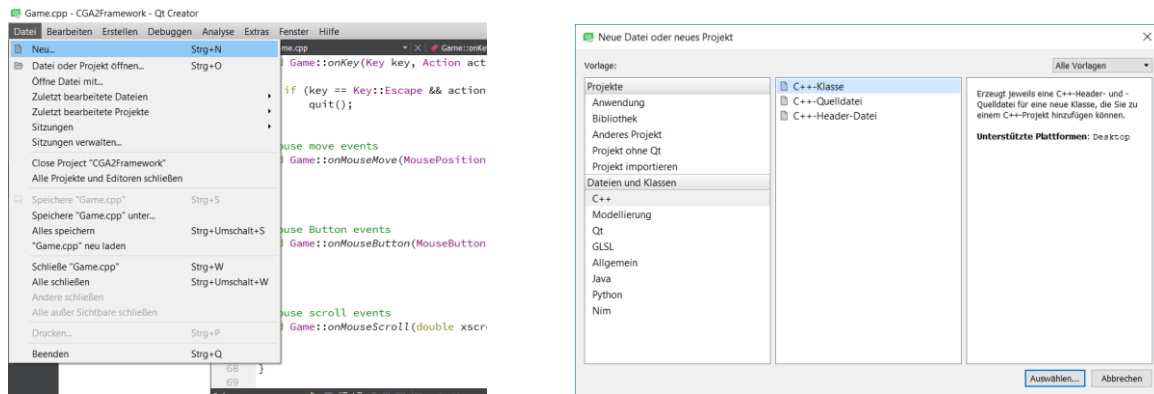
Der grüne Pfeil startet wie gehabt die Kompilierung und wenn alles glatt geht das Programm:



Source Files in QtCreator hinzufügen

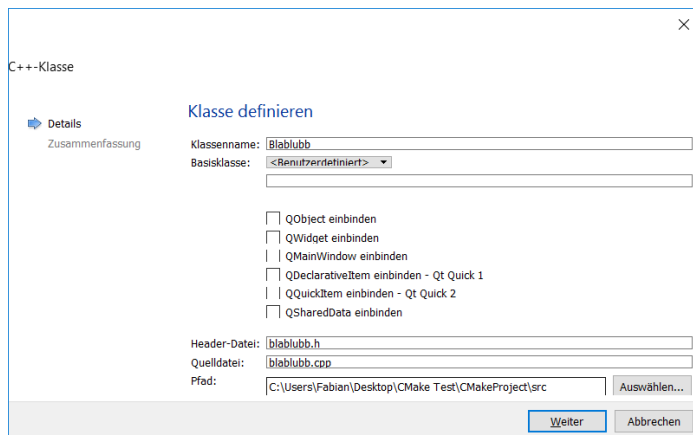
Habt ihr das CMake Projekt mit QtCreator geöffnet könnt ihr natürlich auch hier neue Source Files hinzufügen.

Geht dazu auf *Datei -> Neu...*

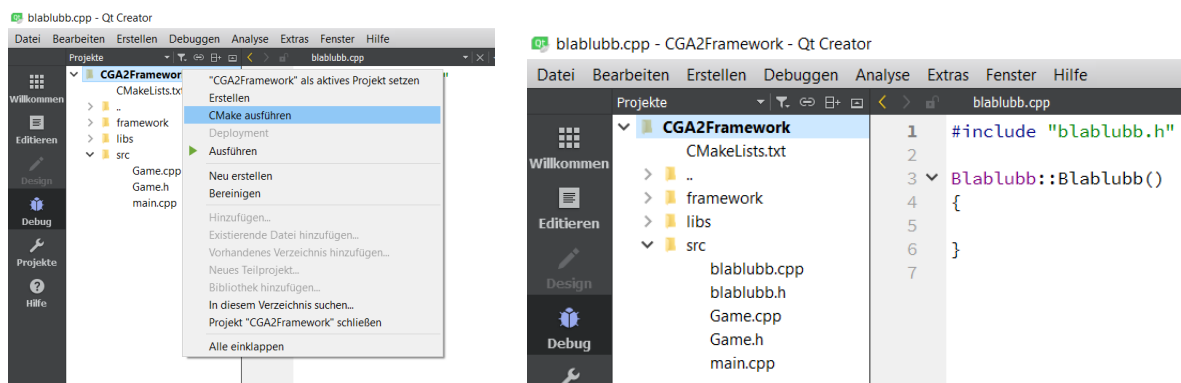


Und wählt unter „Dateien und Klassen“ -> „C++“ das aus, was ihr hinzufügen möchtet.

Vergebt einen Namen und achtet wieder darauf, dass die Dateien irgendwo unter dem „src“ Ordner des CMake Projekts abgelegt werden:



Danach stehen die neuen Dateien aber noch nicht im Projekt. Rechtsklickt dann einfach das Projekt und wählt *CMake ausführen...*



Jetzt sind die Dateien dem Projekt hinzugefügt.

QtCreator: Debugger für MSVC hinzufügen (bei MinGW nicht nötig)

Wenn ihr ein Visual Studio Kit gewählt habt, ist zunächst kein Debugger installiert. Den müsst ihr nachinstallieren, wenn ihr keine „Rate wo der Fehler ist“-Spiele spielen wollt.

Eine genaue Anleitung dazu findet ihr hier: <http://doc.qt.io/qtcreator/creator-debugger-engines.html>

Und als Kurzanleitung:

1. Je nachdem, welche Windows Version ihr installiert habt, müsst ihr euch das passende Windows SDK installieren:

[Windows 10](#)

[Windows 8/8.1](#)

[Windows 7](#)

Über das Windows SDK kommt man auch an fehlende Bibliotheksdateien und Header wie `<GL/gl.h>` oder `OpenGL32.lib`.

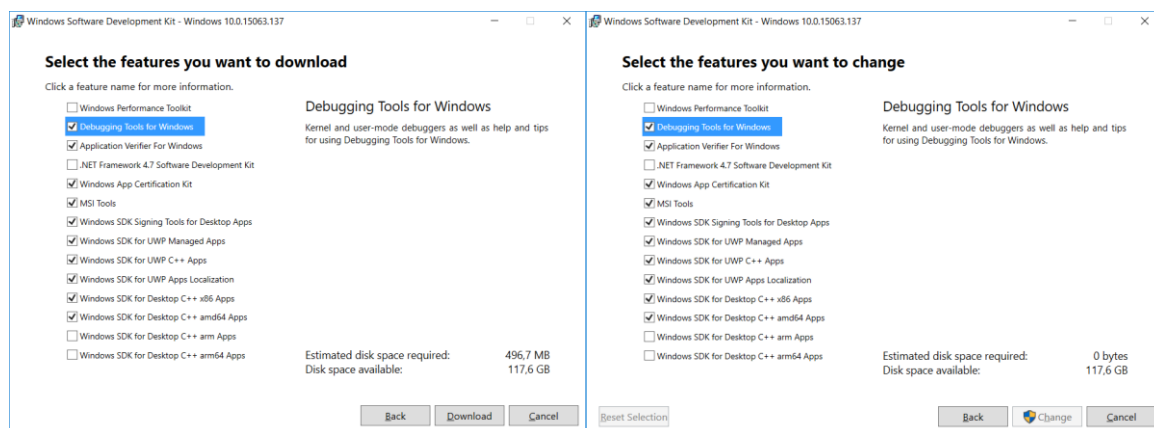
Falls ihr bereits Visual Studio installiert habt, ist das Windows SDK zwar schon installiert, aber Visual Studio benutzt einen eigenen Debugger und verzichtet deswegen auf die Installation der Standard-Debugging Tools. In dem Fall müsst ihr das SDK nicht erneut installieren.

Geht auf Systemsteuerung -> Programme und Features (oder wie auch immer das bei euch heißt) und sucht den Eintrag zu „Windows Software Development Kit“. Klickt dann auf *Ändern*. Im nächsten Dialog wählt *Change* aus.

2. Jetzt seht ihr, egal ob ihr das SDK neu installiert oder die *Ändern* Methode benutzt folgende Auswahl:

Neuinstallation:

Über Programme und Features -> Ändern:

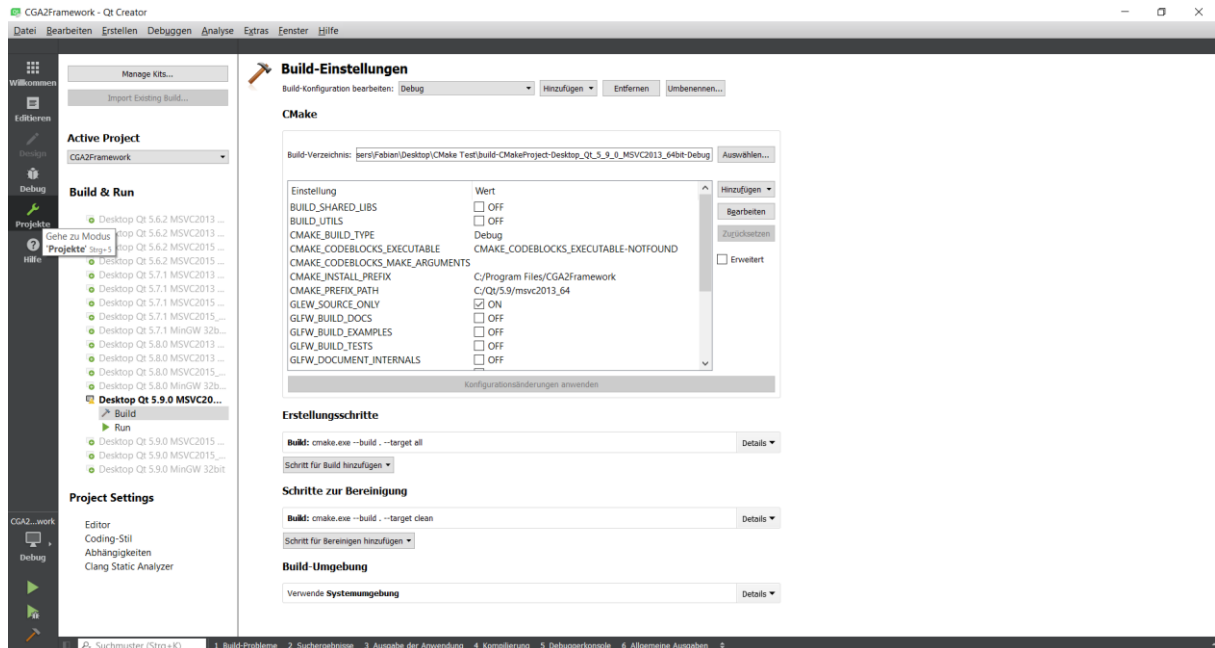


Lasst alles andere angehakt und setzt den Haken bei „Debugging Tools for Windows“.

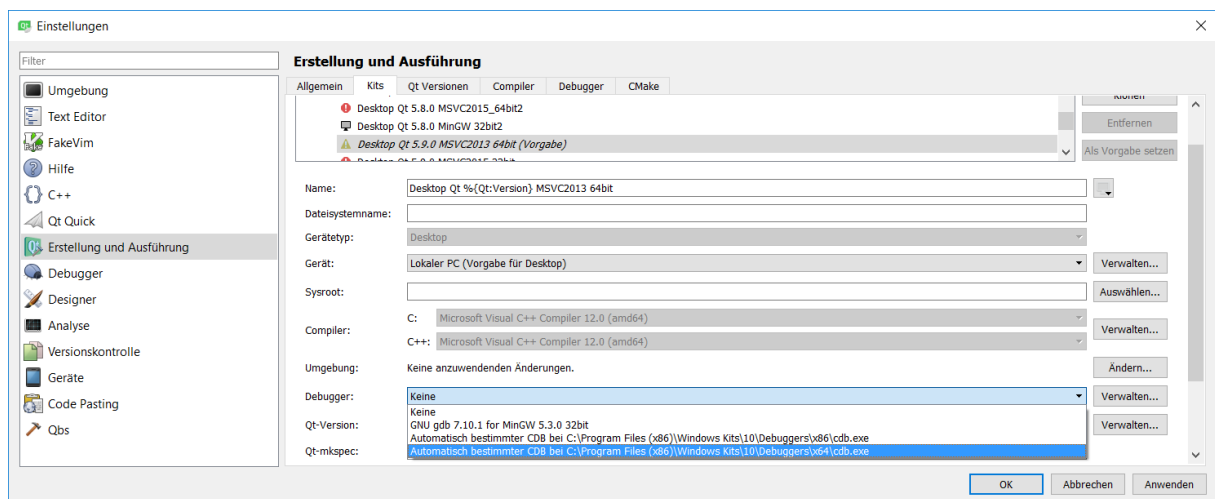
Wenn unten „Windows SDK for Desktop C++ x86“ oder „Windows SDK for Desktop C++ amd64“ nicht angehakt sind, könnt ihr das noch tun. Das installiert euch zum Beispiel die OpenGL Bibliotheken und Header nach. Klickt auf *Change* bzw. *Download*. Wenn Änderungen auszuführen sind oder Dinge herunterzuladen sind wird das nun getan.

Bei der Neuinstallation erhaltet ihr am Ende einen Hinweis, wo der Setup liegt. Den führt ihr aus und seid mit diesem Schritt fertig.

3. Wenn ihr jetzt mit QtCreator das Projekt erneut öffnet, geht zur Projekteinstellungs-Seite (linker Rand -> *Projekte*):

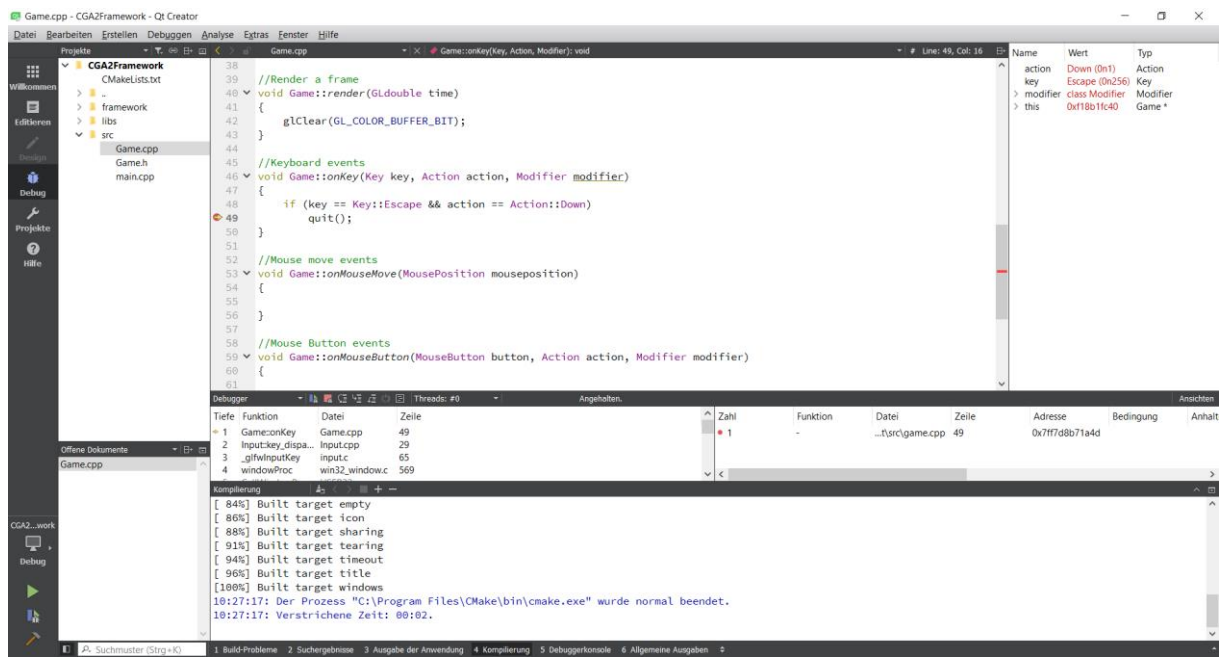


4. Danach auf *Manage Kits...* und wählt das Kit aus welches ihr zur Projekterstellung gewählt habt:



5. Unten bei „Debugger:“ steht jetzt noch „Keine“. Wählt hier den Debugger aus, der zu eurem Projekt passt (32-oder 64 bit) und klickt *OK*.

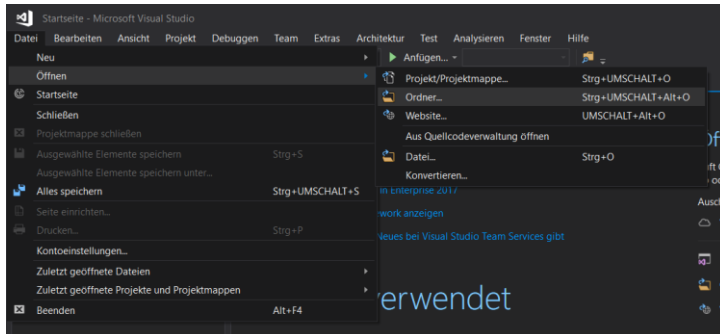
Jetzt könnt ihr zurück zum Editiermodus gehen und ganz normal Debuggen:



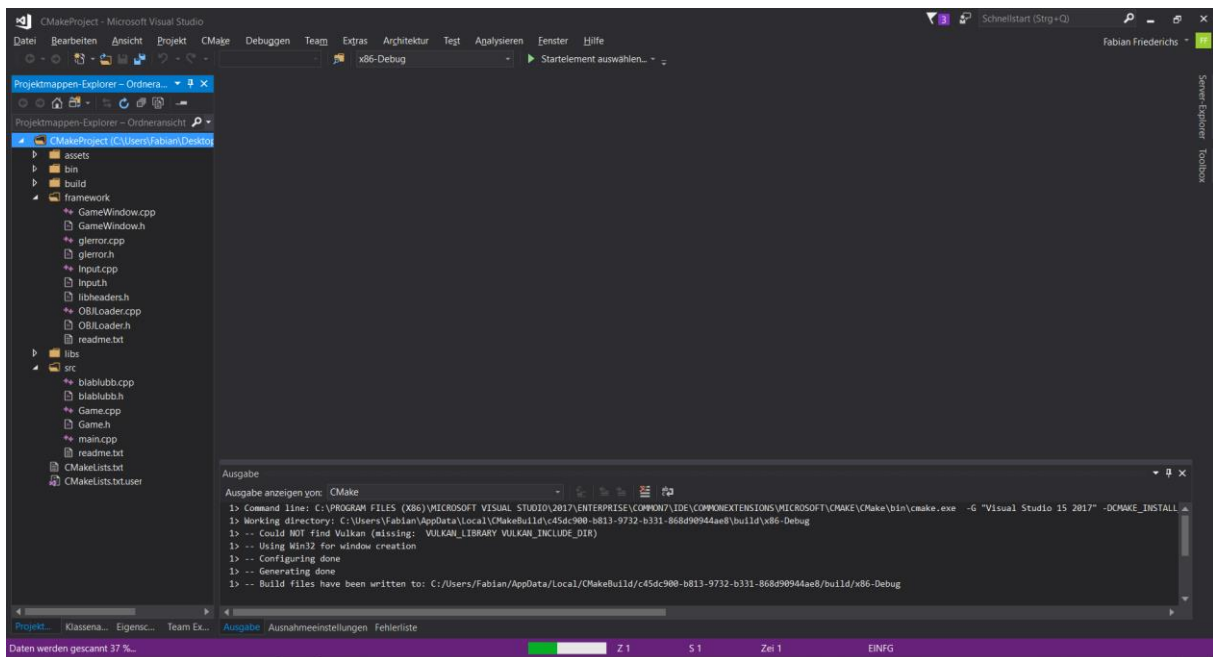
CMake Projekt mit Visual Studio direkt öffnen (nur VS2017)

Ab Visual Studio 2017 kann man CMake Projekte ebenfalls direkt öffnen.

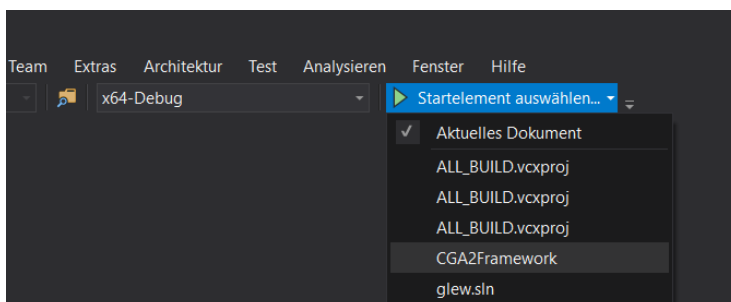
Dazu Visual Studio starten, *Datei -> Öffnen -> Ordner...*



Und den Ordner mit der CMakeLists.txt Datei öffnen. Heraus kommt das:

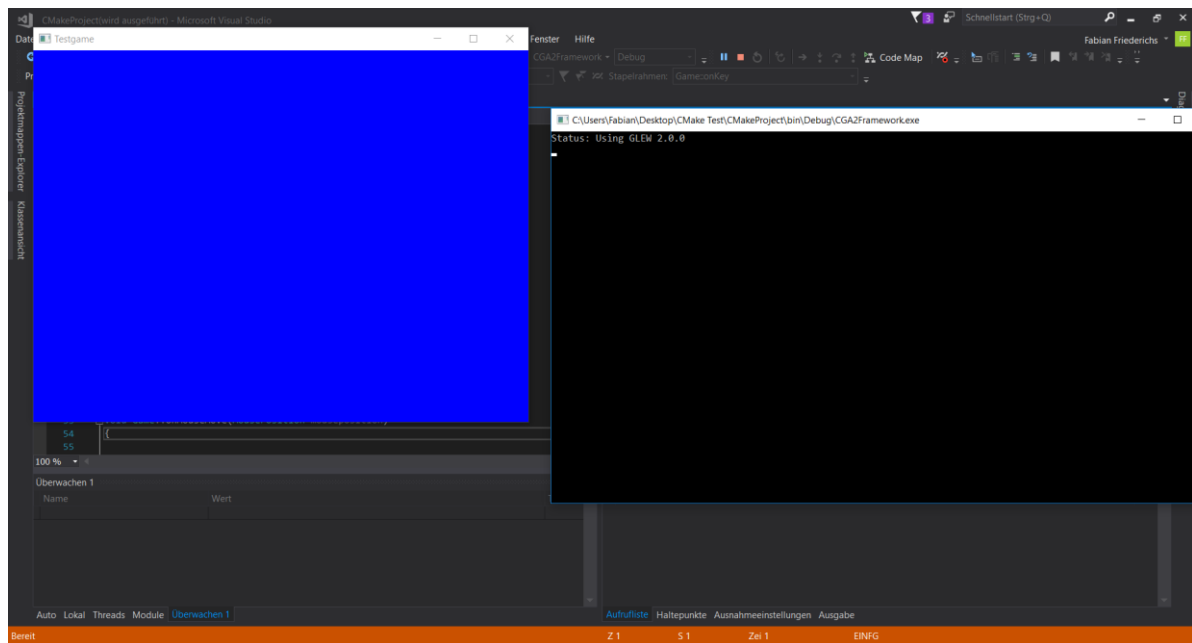


Als nächstes oben auswählen, welche Konfiguration ihr starten wollt (x86/x64 Debug/Release) und welches Projekt gestartet werden soll (*Startelement auswählen...*):



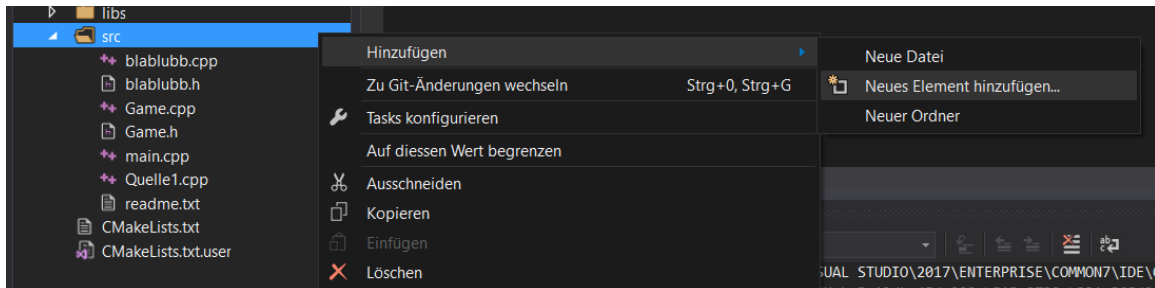
Kümmert euch nicht um die ganzen anderen Elemente, das sind Einzelprojekte, die dafür da sind, die benötigten Bibliotheken zu liefern.

Jetzt könnt ihr das Projekt ganz normal starten und debuggen:

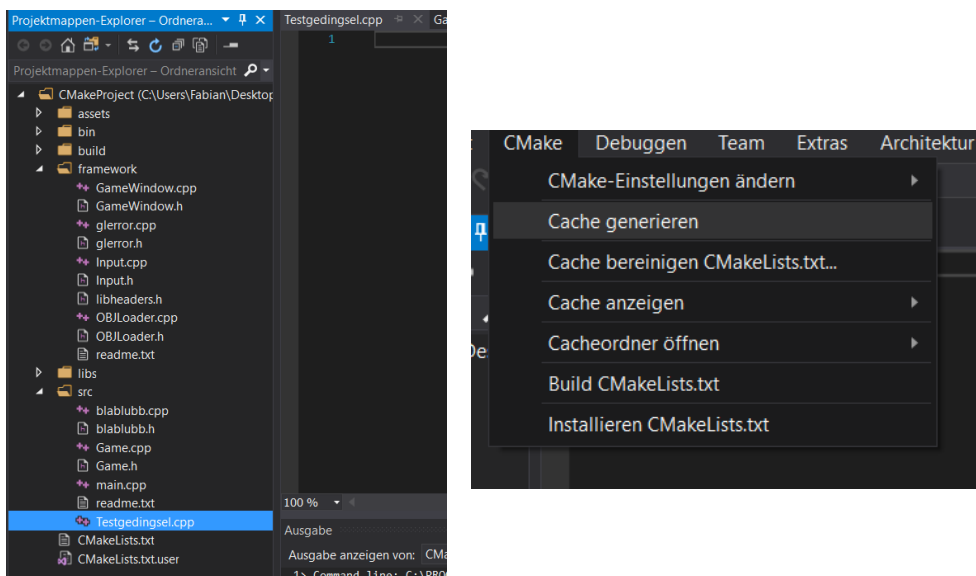


Source Files zu CMake/Visual Studio hinzufügen

Hab ihr das CMake Projekt direkt mit Visual Studio 2017 geöffnet könnt ihr neue Dateien einfach über Rechtsklick auf „src“ -> *Neues Element hinzufügen...* hinzufügen.



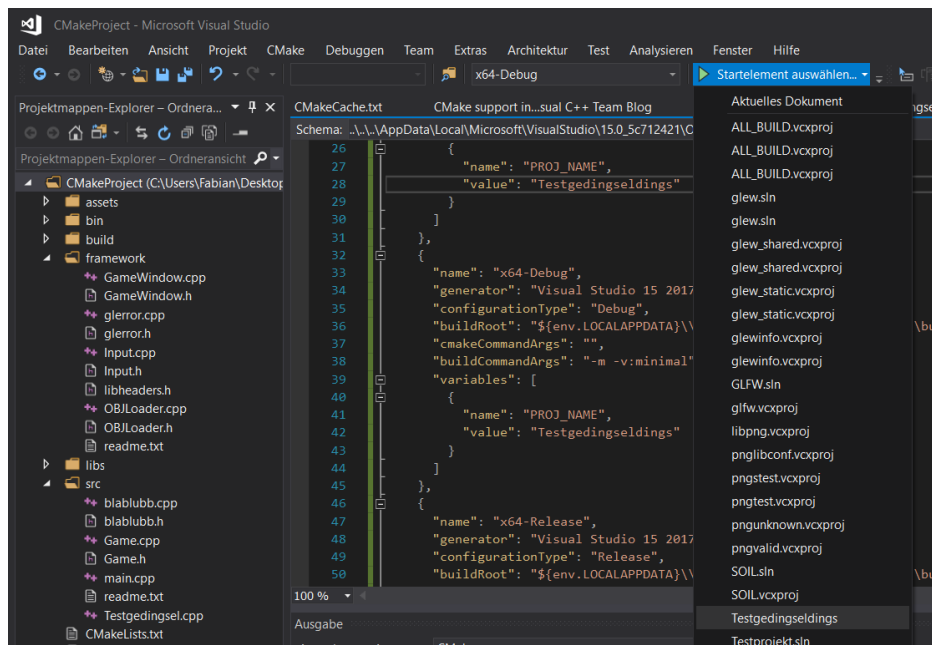
Die neue Datei ist dann sofort sichtbar. Wenn die Datei aus irgendeinem Grund nicht mitkompiliert werden sollte oder in einer `#include<...>` Anweisung nicht gefunden wird, sagt CMake, dass es das Projekt noch einmal aktualisieren soll (Klick auf *CMake -> Cache generieren*):



CMake Einstellungen, wie den Projektnamen können über *CMake-Einstellungen ändern* -> *CMakeLists.txt* geändert werden. Man bekommt eine JSON Datei heraus, die etwas umständlich aussieht aber funktioniert. Um zum Beispiel den Projektnamen zu ändern, tragt ihr in die gewünschte Konfiguration ein „variables“ Array ein und setzt „PROJ_NAME“ auf einen neuen Wert. Danach wird CMake automatisch neu ausgeführt.

```
{
  "name": "x64-Debug",                                <- Einstellung für Debug, 64-bit
  "generator": "Visual Studio 15 2017 Win64",
  "configurationType": "Debug",
  "buildRoot": "${env.LOCALAPPDATA}\\CMakeBuild\\${workspaceHash}\\build\\${name}",
  "cmakeCommandArgs": "",
  "buildCommandArgs": "-m -v:minimal",
  "variables": [                                       <- Hier „variables“ einfügen
    {
      "name": "PROJ_NAME",
      "value": "Testgedingseldings"
    }
  ]
},
```

Danach seht ihr unter *Startelement auswählen...* den neuen Projektnamen:



Visual Studio verwendet den Ansatz mit der JSON Datei, damit man die Konfiguration an andere Personen weitergeben kann. Sonst würde CMake die eingestellten Werte bei der nächsten Person die das Projekt verwendet „vergessen“. Das Feature ist noch ganz neu und etwas ungeschliffen, vielleicht kommt in einem zukünftigen Update eine übersichtlichere Möglichkeit zur Konfiguration von CMake Projekten dazu.

Build-Konfigurationen (Was sind Debug und Release?)

Moderne Compiler können euren Code bis zur Unkenntlichkeit verunstalten. Und das ist gut so. Durch Umsortierung von Befehlen, weglassen von „unnötigen“ Variablen usw. kann ein Programm in der Ausführung stark beschleunigt werden. Man nennt das Optimierung.

Wenn man allerdings Schritt-für-Schritt debuggen möchte ist die Optimierung eher hinderlich. Plötzlich werden Zeilen übersprungen, leere Zeilen tun seltsame Dinge, Werte von Variablen können nicht mehr ausgelesen werden usw. Das liegt daran, dass der Code den man geschrieben hat nicht mehr zu dem passt, was der Debugger vom Compiler bekommt.

Daher gibt es zwei (es gibt noch mehr, aber die sind in 99% der Fälle uninteressant) sogenannte Build-Konfigurationen:

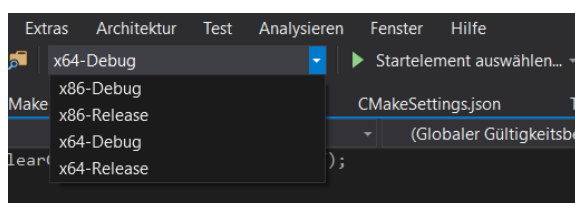
Release: In der Release-Konfiguration sind alle Optimierungen eingeschaltet. Das Programm, das herauskommt, ist das schnellste, was der Compiler aus dem Code machen kann.

Benutzt diese Option, wenn ihr das Programm fertig habt und z.B. jemandem vorführen wollt. Es wird einige Male schneller und flüssiger laufen, als das, was aus der Debug-Konfiguration herauskommt.

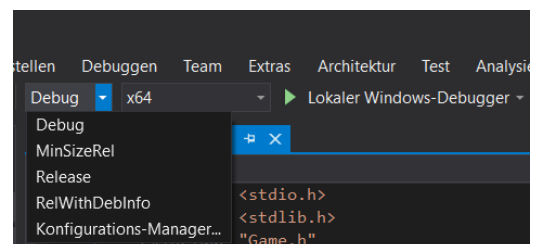
Debug: In der Debug-Konfiguration sind alle Optimierungen ausgeschaltet. Nur mit dieser Konfiguration kann man anständig debuggen, Werte von Variablen ansehen usw. Das Programm kann aber unter Umständen sehr langsam laufen, bei komplizierteren Grafikanwendungen kann das manchmal zu einer (wenn auch sehr hübschen, wenn man Alles richtig macht) Diashow führen.

Benutzt unbedingt diese Option, wenn ihr in der Debug-Phase seid und versucht Fehler in eurem Code zu finden.

In Visual Studio könnt ihr die Konfiguration oben neben dem grünen Pfeil auswählen:

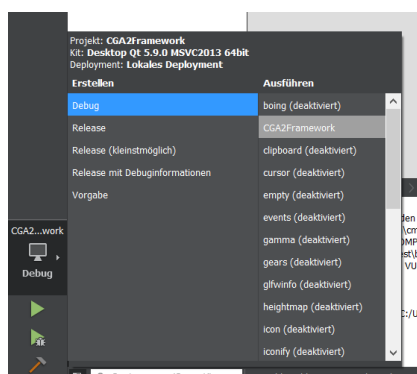


1: CMake Projekt in Visual Studio



2: "Echtes" Visual Studio Projekt

In QtCreator wählt ihr die zu startende Konfiguration über dem grünen Pfeil aus:



Benutzung des Frameworks

Das Basisfenster benutzen

Im „src“ Ordner befindet sich von Anfang an eine Beispiel Klasse, die nichts Anderes tut, als ein blaues Fenster anzuzeigen. Ihr könnt diese Klasse komplett neu schreiben oder einfach ausbauen.

Game.h:

```
#ifndef _GAME_H_
#define _GAME_H_
#include "GameWindow.h"

class Game : public GameWindow
{
public:
    Game();
    ~Game();

    void init() override;
    void shutdown() override;

    void update(GLdouble dt) override;
    void render(GLdouble dt) override;

    void onKey(Key key, Action action, Modifier modifier) override;
    void onMouseMove(MousePosition mouseposition) override;
    void onMouseButton(MouseButton button, Action action, Modifier modifier) override;
    void onMouseScroll(double xscroll, double yscroll) override;

    void onFramebufferResize(int width, int height) override;

private:
};
#endif
```

Das ist der Header der Beispielklasse. Die Klasse ist von der abstrakten Klasse „GameWindow“ abgeleitet. Die Methodendeklarationen sind alle optional, im diesem Beispiel sind sie alle definiert.

Wenn ihr eine neue Klasse anfangen wollt, müsst ihr diese ebenfalls von GameWindow ableiten und dann die Methoden überschreiben, die ihr braucht.

Die .cpp Datei dazu sieht so aus:

```
#include "Game.h"

Game::Game() :
GameWindow(      800,          //width
               600,          //height
               false,        //fullscreen
               true,         //vsync
               3,            //OpenGL Version Major
               3,            //OpenGL Version Minor => Here the OpenGL Version is 3.3
               "Testgame")   //Title of the window
{
}

Game::~Game()
{
}

//Initialization here. (i.e. load a scene, load textures ...)
void Game::init()
{
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
}

//cleanup. Free resources here.
void Game::shutdown()
{
}

//Update Game Logic here
void Game::update(GLdouble time)
{
}

//Render a frame
void Game::render(GLdouble time)
{
    glClear(GL_COLOR_BUFFER_BIT);
}

//Keyboard events
void Game::onKey(Key key, Action action, Modifier modifier)
{
    if (key == Key::Escape && action == Action::Down)
        quit();
}

//Mouse move events
void Game::onMouseMove(MousePosition mouseposition)
{
}

//Mouse Button events
void Game::onMouseButton(MouseButton button, Action action, Modifier modifier)
{
}

//Mouse scroll events
void Game::onMouseScroll(double xscroll, double yscroll)
{
}

//Window resize events
void Game::onFramebufferResize(int width, int height)
{
}
```


Einstellungen für das Fenster

Im Konstruktor der abgeleiteten Klasse wird der Konstruktor der Basisklasse mit bestimmten Parametern aufgerufen:

```
Game::Game() :
GameWindow(      800,          //width
                600,          //height
                false,        //fullscreen
                true,         //vsync
                3,            //OpenGL Version Major
                3,            //OpenGL Version Minor  => Here the OpenGL Version is 3.3
                "Testgame")   //Title of the window
{
}
```

Die Parameter bewirken Folgendes:

- | | | |
|---------------|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| #1 | (800) | Setzt die horizontale Auflösung des Fensters oder des Monitors, falls der Vollbildmodus verwendet wird. |
| #2 | (600) | Setzt die vertikale Auflösung des Fensters oder des Monitors, falls der Vollbildmodus verwendet wird. |
| #3 | (false) | Gibt an, ob der Vollbildmodus verwendet werden soll. |
| #4 | (true) | Gibt an, ob vertikale Synchronisation eingeschaltet werden soll. Am Anfang ist das sehr sinnvoll, da sonst bei sehr einfachen oder sogar leeren Szenen mehrere Tausend Bilder pro Sekunde gerendert werden. Manche Grafikkarten fangen bei solchen Bildraten an zu „singen“. |
| #5, #6 | (3, 3) | Geben zusammen die OpenGL Version an, die verwendet werden soll. Unterstützt das System die angegebene Version nicht, wird ein Fehler angezeigt. Die aktuelle Version ist 4.5 und bietet viele neue Features, wird aber erst von wenigen Notebooks unterstützt. Ab Version 3.3 kann man sinnvoll mit modernem OpenGL arbeiten. |
| #7 | („Testgame“) | Gibt den Text an, der in der Titelleiste des Fensters steht. |

Ihr könnt die Parameter entweder wie hier hardcoden, ihr könnt sie aber auch von einem anderen Ort holen. Z.B. später von einer fancy Konfigurationsdatei oder ähnlichem.

Wenn ihr für #1, #2 eine nicht unterstützte Auflösung wählt und den Vollbildmodus einschaltet, wird eine andere Auflösung benutzt, die den angegebenen Werten am nächsten ist.

`init()`, `shutdown()`, `render()` und `update()`

Diese Methoden können bei Bedarf überschrieben werden und werden automatisch vom Framework aufgerufen.

`init()`

Wird beim Programmstart aufgerufen. Hier könnt ihr Assets laden, Szenen anlegen, Kameras konfigurieren, und alles tun was sonst noch an Initialisierungsaufgaben ansteht.

`shutdown()`

Wird aufgerufen, wenn das Programm beendet wird. Hier solltet ihr alle geladenen Ressourcen freigeben, damit das Programm sauber beendet werden kann.

`update(GLdouble dttime)`

Wird in festen Zeitabständen aufgerufen. Hier sollte die Game Logik simuliert werden. Also Objekte bewegen, Kamera bewegen usw. damit das Spiel auf schnelleren Rechnern nicht plötzlich in einer anderen Geschwindigkeit läuft als auf langsamen.

`dttime:`

Das Zeit-Delta um das die Simulation weitergeführt werden soll. Bei der `update()` Methode sollte `dttime` immer annähernd gleich sein. Zeitwert in ms.

`render(GLdouble dttime)`

Wird aufgerufen, sobald der letzte Frame fertig gerendert wurde. Hier sollte das Rendering der Szene stattfinden.

`dttime`

Gibt an, wie lange der letzte Frame zur Fertigstellung brauchte. Zeitwert in ms.

Maus- und Tastatureingaben

Auf Events reagieren

Passiert etwas auf der Tastatur, wird die Maus bewegt, geklickt oder gescrollt, werden, sofern vorhanden diese Methoden entsprechend aufgerufen:

```
//Keyboard events
void Game::onKey(Key key, Action action, Modifier modifier)
{
    if (key == Key::Escape && action == Action::Down)
        quit();
}

//Mouse move events
void Game::onMouseMove(MousePosition mouseposition)
{
}

//Mouse Button events
void Game::onMouseButton(MouseButton button, Action action, Modifier modifier)
{
}

//Mouse scroll events
void Game::onMouseScroll(double xscroll, double yscroll)
{
}
```

onKey(...)	Wird aufgerufen, wenn eine Taste gedrückt, gehalten oder losgelassen wird.
key	Gibt an, welche Taste das Ereignis ausgelöst hat.
action	Gibt an, ob die Taste gedrückt, gehalten oder losgelassen wurde.
modifier	Gibt an, ob zum Zeitpunkt des Ereignisses Ctrl-, Alt-, Shift- oder Super-(Windows-) Tasten gedrückt waren.
onMouseMove(...)	Wird aufgerufen, wenn die Maus bewegt wird.
mouseposition	Gibt die neue und alte Mausposition an.
onMouseButton(...)	Wird aufgerufen, wenn eine Maustaste gedrückt wird.
button	Gibt an, welche Maustaste das Ereignis ausgelöst hat.
action	Gibt an, ob die Maustaste gedrückt oder losgelassen wurde.
modifier	Gibt an, ob zum Zeitpunkt des Ereignisses Ctrl-, Alt-, Shift- oder Super-(Windows-) Tasten gedrückt waren.
onMouseScroll(...)	Wird aufgerufen, wenn das Mauseisen bewegt wird.
xscroll	Wenn die Maus auch seitliches Scrollen unterstützt, gibt dieser Wert an, wie weit das Mauseisen nach links oder rechts bewegt wurde.
yscroll	Gibt an, wie weit das Mauseisen nach vorne oder hinten bewegt wurde. Positiver oder negativer Wert.

In diesem Beispiel wird das Programm beendet, sobald die Escape-Taste gedrückt wird.

Key- und Mousebutton States direkt abfragen

Wenn es um die Bewegung der Kamera mit z.B. den Tasten W, A, S, D geht, gibt es eine deutlich bessere Möglichkeit. Testet man nämlich nur ob eine Taste gehalten wird, kommt das Signal alle ~20 ms und das Ganze wird eine sehr ruckelige Angelegenheit.

Stattdessen könnt ihr einfach den Status einer Taste in z.B. der update() Methode abfragen:

```
void Game::update(GLdouble time)
{
    if (input->getKeyState(Key::W) == KeyState::Pressed)
    {
        glClearColor(1.0f, 0.0f, 0.0f, 1.0f);
    }
    else
    {
        glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
    }
}
```

Mit `input->getKeyState(...)` kann man den Status einer Taste abfragen. In diesem einfachen Beispiel wird die Clear Color des Fensters einfach auf Rot gesetzt, wenn die W Taste gedrückt ist.

Das geht natürlich auch mit Maustasten:

```
void Game::update(GLdouble time)
{
    if (input->getMouseButtonState(MouseButton::Left) == MouseButtonState::Pressed)
    {
        glClearColor(1.0f, 0.0f, 0.0f, 1.0f);
    }
    else
    {
        glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
    }
}
```

Auf das Input Objekt kann man in der gesamten abgeleiteten Klasse zugreifen. Aus technischen Gründen ist das Objekt leider ein Pointer und muss über den `->` Operator bedient werden.

Den Mauszeiger zeigen und verstecken

Ist der Mauszeiger sichtbar, werden Mausbewegungen nicht mehr erkannt, wenn er außerhalb des Fensters ist. Um den Mauszeiger unsichtbar zu machen und zu verhindern, dass er aus dem Fenster herauskommt ruft man die `setCursorVisible()` Methode auf:

```
void Game::init()
{
    input->setCursorVisible(false);
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
}
```

Übergibt man `false`, wird der Mauszeiger versteckt, `true` macht ihn wieder sichtbar.

Key ist eine Enumeration, die für jede Taste einen Namen definiert. Auf die Werte greift man mit `Key::<Name>` zu. Key definiert folgende Tasten:

Unknown	Space	Apostrophe	Comma
Minus	Period	Slash	K0
K1	K2	K3	K4
K5	K6	K7	K8
K9	Semicolon	Equal	A
B	C	D	E
F	G	H	I
J	K	L	M
N	O	P	Q
R	S	T	U
V	W	X	Y
Z	LeftBracket	Backslash	RightBracket
GraveAccent	World1	World2	Escape
Enter	Tab	Backspace	Insert
Delete	Right	Left	Down
Up	PageUp	PageDown	Home
End	CapsLock	ScrollLock	NumLock
PrintScreen	Pause	F1	F2
F3	F4	F5	F6
F7	F8	F9	F10
F11	F12	F13	F14
F15	F16	F17	F18
F19	F20	F21	F22
F23	F24	F25	NumPad0
NumPad1	NumPad2	NumPad3	NumPad4
NumPad5	NumPad6	NumPad7	NumPad8
NumPad9	NumPadDecimal	NumPadDivide	NumPadMultiply
NumPadSubtract	NumPadAdd	NumPadEnter	NumPadEqual
LeftShift	LeftCtrl	LeftAlt	LeftSuper
RightShift	RightCtrl	RightAlt	RightSuper
Menu	Last		

Werte und Namen wurden direkt aus der *GLFW3* Bibliothek übernommen und in C++11 enum classes verpackt, um Code Completion besser ausnutzen zu können. Laut der GLFW3-Dokumentation sind die Tasten auf das amerikanische Tastaturlayout gemappt, daher muss man bei exotischeren Tasten manchmal etwas rumprobieren bis man die richtige Taste gefunden hat.

MouseButton benennt die Maustasten. Auf die Werte greift man mit `MouseButton::<Name>` zu. MouseButton definiert folgende Maustasten:

MouseButton1	MouseButton2	MouseButton3	MouseButton4
MouseButton5	MouseButton6	MouseButton7	MouseButton8
Last	Left	Right	Middle

Action definiert Aktionen für Tastatur- und Maustasten

Down	Taste wurde gedrückt
Up	Taste wurde losgelassen
Repeat	Taste wird gehalten (kommt nur bei der Tastatur vor)

Auf die Werte von Action greift man wieder mit `Action::<Name>` zu.

Modifier ist eine kleine Klasse mit vier `bool` Membern:

<code>Alt</code>	<code>true</code> , wenn eine Alt Taste gedrückt ist.
<code>Ctrl</code>	<code>true</code> , wenn eine Strg Taste gedrückt ist.
<code>Shift</code>	<code>true</code> , wenn eine Shift Taste gedrückt ist.
<code>Super</code>	<code>true</code> , wenn eine Super(Windows) Taste gedrückt ist.

Auf einen dieser Member greift man mit `modifierobjekt.<Name>` zu.

MousePosition ist eine Klasse, deren Objekte die alte und neue Mausposition enthalten.

`MousePosition` hat vier `double` Member:

<code>X</code>	Neue X-Position
<code>Y</code>	Neue Y-Position
<code>oldX</code>	Alte X-Position
<code>oldY</code>	Alte Y-Position

Auf einen dieser Member greift man mit `mousepositionobjekt.<Name>` zu.

KeyState ist wieder eine Enumeration und hat nur zwei Werte:

<code>Pressed</code>	Die Taste ist gedrückt
<code>Released</code>	Die Taste ist nicht gedrückt

Zugriff mit `KeyState : <Name>`.

MouseButtonState ist exakt das Gleiche wie `KeyState`, etwas redundant aber der Code bleibt verständlicher:

<code>Pressed</code>	Die Maustaste ist gedrückt
<code>Released</code>	Die Maustaste ist nicht gedrückt

Zugriff mit `MouseButtonState : <Name>`.

Reagieren auf Verkleinerung/Vergrößerung des Fensters

Verändert sich die Größe des Fensters und damit die Größe des zugrundeliegenden Framebuffers, wird die Methode `onFramebufferResize(..)` aufgerufen. Hier könnt ihr auf die Größenänderung reagieren und z.B. `glViewport(...)` aufrufen um den gerenderten Bereich der neuen Fenstergröße anzupassen:

```
//Window resize events
void Game::onFramebufferResize(int width, int height)
{
    glViewport(0, 0, width, height);
}
```

`width` und `height` beschreiben die neue Größe des Framebuffers.

Start des Programms / main.cpp

Die main.cpp Datei kann stark vernachlässigt werden. Hier solltet ihr nur eine Instanz eurer Game Klasse erzeugen und run() aufrufen. Denkt daran, dass das starten des Programms auch fehlschlagen kann, z.B. wenn das System die angegebene OpenGL Version nicht unterstützt. In so einem Fall solltet ihr den Fehler mit einem try/catch Block abfangen und berichten, das irgendetwas schief gegangen ist:

```
#include <iostream>
#include "Game.h"

int main(void)
{
    try
    {
        Game mg;
        mg.run();
    }
    catch(const std::exception& ex)
    {
        std::cout << ex.what() << "\nDruecke eine Taste zum beenden." << std::endl;
        getchar();
    }

    return 1;
}
```

Beenden des Programms

Möchtet ihr das Programm beenden, ruft einfach an irgendeiner Stelle in eurer Game Klasse die Methode quit() auf:

```
void Game::onKey(Key key, Action action, Modifier modifier)
{
    if (key == Key::Escape && action == Action::Down)
        quit();
}
```

Tut ihr das, wird zuerst eure shutdown() Methode aufgerufen und danach das Programm kontrolliert beendet.

OBJ-Dateien laden

Im Framework enthalten ist auch ein recht simpler Loader für .obj Dateien. Um ihn zu nutzen, inkludiert als erstes „OBJLoader.h“, in der Klasse, in der ihr ihn benutzen wollt:

```
#ifndef _GAME_H_
#define _GAME_H_
#include "GameWindow.h"
#include "OBJLoader.h"

class Game : public GameWindow
{
...
}
```

Danach könnt ihr den Loader benutzen:

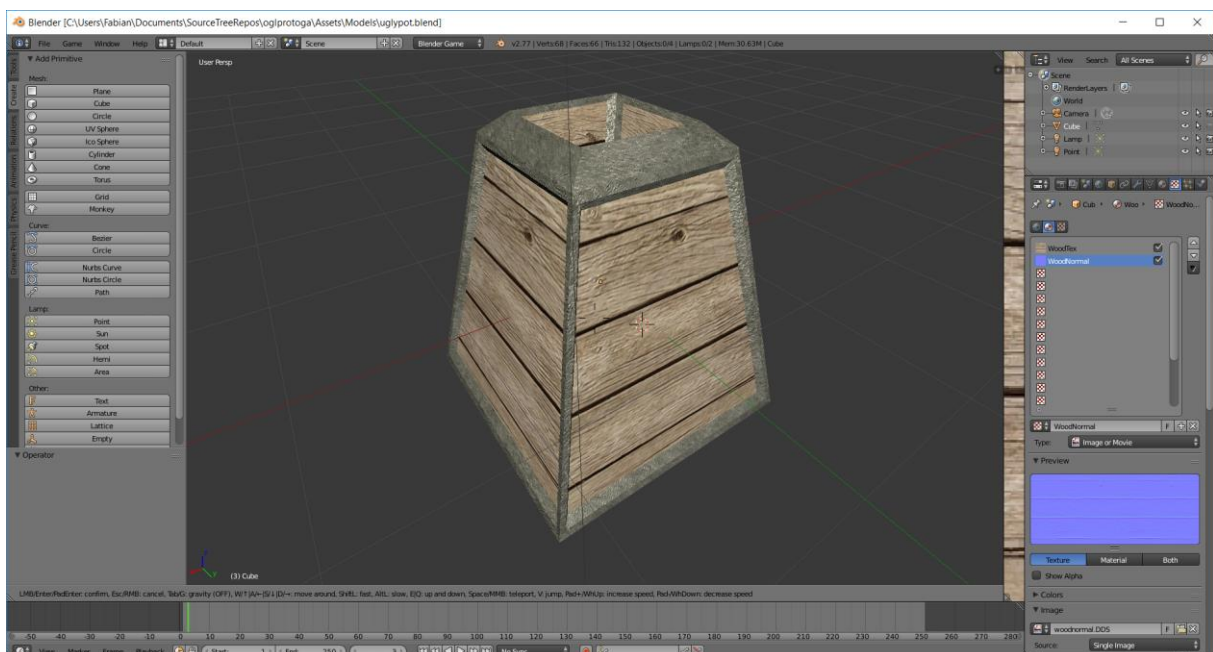
```
void Game::init()
{
    input->setCursorVisible(false);
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);

    try
    {
        OBJResult res = OBJLoader::loadOBJ("../assets/models/uglypot.obj");
    }
    catch (const OBJException& ex)
    {
        std::cout << ex.what() << "\n";
    }
}
```

Wenn beim Laden etwas schief geht, wird eine Exception geworfen. Die solltet ihr abfangen und ausgeben.

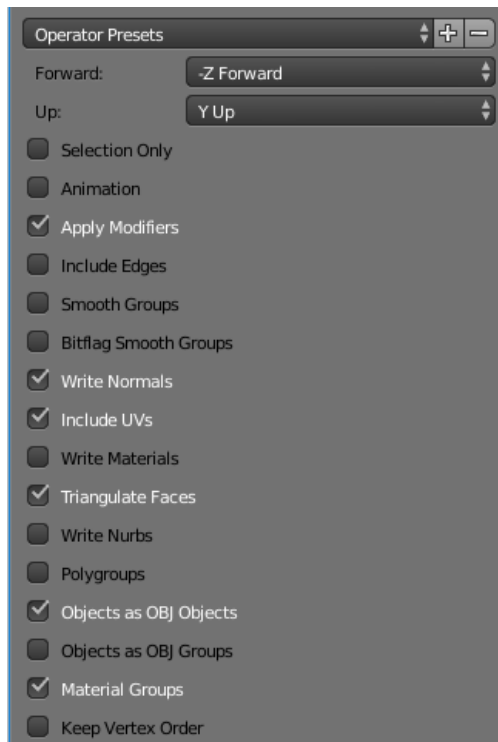
loadOBJ(...) nimmt den Pfad zur .obj Datei entgegen und gibt ein Objekt vom Typ OBJResult zurück.

Ein OBJResult enthält OBJObjects, ein OBJObject enthält OBJMeshes und ein OBJMesh enthält schließlich Vertices und Indices. Zunächst aber die .obj Datei, dann versteht man besser was am Ende herauskommt. So sieht diese wunderschöne *hust* Kreation in Blender aus:



Einige Faces haben ein Holz-Material bekommen, die anderen eine Metall-Material.

Exportiert man in Blender dann nach .obj muss man ein paar Häkchen setzen:



Wichtig sind hier „Write Normals“ und „Write UVs“, wenn man möchte, dass die Normalen und Texturkoordinaten mit exportiert werden.

„Triangulate Faces“ sollte angehakt werden. Der OBJLoader kommt nicht mit Vierecken klar. Wenn man den Haken nicht setzt, sieht das Model später aus wie ein Schweizer Käse (mit dreieckigen Löchern).

„Objects as OBJ Objects“ bewirkt, dass für jedes einzelne Objekt eine „o“-Sektion in der .obj Datei angelegt wird.

„Material Groups“ fasst pro Objekt Faces mit gleichem Material zu Gruppen zusammen und packt diese in eine „g“-Sektion in der .obj Datei.

Hakt man die letzten beiden nicht an, bekommt man vom OBJLoader ein Model, das nur aus einem Objekt mit einem Mesh besteht. (Kann durchaus sinnvoll sein, wenn man Models mit nur einer Textur hat oder die Fähigkeit Texturatlant zu erstellen)

Materialien kann der Loader nicht verarbeiten und ignoriert diese. Mit gesetztem „Material Groups“ kann man dann aber im Programm den einzelnen Meshes wieder Materials zuweisen.

Exportiert man den hässlichen Blumenkübel von oben mit diesen Einstellungen, kommt so eine .obj Datei heraus:

```
# Blender v2.77 (sub 0) OBJ File: 'uglypot.blend'      <- Kommentare
# www.blender.org
o Cube                                                <- Objekt
v 1.000000 -1.000000 -1.000000                        <- Positionen
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
...
vt 0.5297 0.1097                                     <- Texturkoordinaten
vt 0.8329 0.3654
vt 0.6441 0.3654
...
vn -0.0000 0.8132 0.5820                             <- Normalen
vn -0.9748 0.2230 -0.0000
vn 0.0000 0.2230 -0.9748
...
g Cube_Cube_Wood                                     <- Mesh/Material Gruppe „Cube_Cube_Wood“
s off
f 15/1/1 50/2/1 52/3/1                                <- Faces der Gruppe „Cube_Cube_Wood“
f 19/4/2 18/5/2 17/6/2
f 21/7/3 24/8/3 23/9/3
...
g Cube_Cube_Metal                                    <- Mesh/Material Gruppe „Cube_Cube_Metal“
f 2/53/14 4/54/14 1/55/14                             <- Faces der Gruppe „Cube_Cube_Metal“
f 8/56/15 11/57/15 12/58/15
f 2/59/16 55/60/16 54/61/16
...
```

Das OBJResult enthält damit für jedes „o“ ein OBJObject und jedes OBJObject enthält für jedes „g“ ein Mesh. OBJResult bekommt zusätzlich noch den Pfad zur Datei, OBJObject den Namen neben „o“, also hier „Cube“ und OBJMesh den Namen neben „g“, also „Cube_Cube_Wood“ bzw. „Cube_Cube_Metal“.

loadOBJ kann man auch noch zwei weitere Parameter übergeben:

```
try
{
    OBJResult res = OBJLoader::loadOBJ(
        "../../assets/models/uglypot.obj",    //Pfad zur .obj Datei
        true,                                  //Normalen generieren
        true,                                  //Tangenten generieren
    );
}
catch (const OBJException& ex)
{
    std::cout << ex.what() << "\n";
}
```

Der zweite Parameter, wenn true, bewirkt, dass alle Normalen neu berechnet werden.

Der dritte Parameter, wenn true und wenn die .obj Texturkoordinaten enthält, bewirkt, dass Tangenten für alle Meshes generiert werden. Diese kann man später beim Normal Mapping gebrauchen.

OBJLoader Klassen und Datentypen

OBJResult

```
class OBJResult
{
    /*
        Konstruktoren etc.
    */
public:
    std::string objname;
    std::vector<OBJObject> objects;
};
```

objname	Pfad zur .obj Datei
objects	Array der Objekte aus der .obj Datei

OBJObject

```
class OBJObject
{
    /*
        Konstruktoren etc.
    */
public:
    std::string name;
    std::vector<OBJMesh> meshes;
};
```

name	Name des Objektes
meshes	Array der Meshes dieses Objektes

OBJMesh

```
class OBJMesh
{
    /*
        Konstruktoren etc.
    */
public:
    bool hasPositions;
    bool hasUVs;
    bool hasNormals;
    bool hasTangents;

    std::string name;

    std::vector<Vertex> vertices;
    std::vector<Index> indices;
};
```

hasPositions	Die Vertices des Meshes haben gültige Positionen
hasUVs	Die Vertices des Meshes haben gültige Texturkoordinaten
hasNormals	Die Vertices des Meshes haben gültige Normalen
hasTangents	Die Vertices des Meshes haben gültige Tangenten
name	Der Name des Meshes
vertices	Vertices des Meshes
indices	Indices des Meshes

Vertex

```
struct Vertex
{
    glm::vec3 position;
    glm::vec2 uv;
    glm::vec3 normal;
    glm::vec3 tangent;
};
```

position	Positionsvektor
uv	Texturkoordinate
normal	Normalenvektor
tangent	Tangentenvektor

Index

```
typedef GLuint Index;
```

Aus Bequemlichkeit einfach ein Alias für den Typ `GLuint`. Das dient der Wartbarkeit/Änderbarkeit des Codes. Mit dem Alias muss man nur eine einzige Stelle den Code anfassen um den Datentyp für Indices zu ändern.

Verwenden der Daten

Die letztendlichen Rohdaten, Vertex und Index, sind beide keine komplexen Datentypen. Daher sollte man sie ohne Probleme in einen Vertex- oder Elementbuffer kopieren können.

Als Pointer auf die Daten würde man dann z.B. einfach `mesh.vertices.data()` übergeben. Indices und Vertices sind so gefüllt, dass man sie als `GL_TRIANGLES` rendern kann.

Sinnvoller wäre es natürlich, wenn ihr die Daten in eure eigenen Klassen und Datenstrukturen übernehmt.

Nachbearbeitung

Für einzelne OBJMeshes könnt ihr auch manuell Normalen und Tangenten neu berechnen:

```
try
{
    OBJResult res = OBJLoader::loadOBJ("../assets/models/uglypot.obj");
    OBJMesh mesh1 = res.objects[0].meshes[0];
    OBJMesh mesh2 = res.objects[0].meshes[1];

    OBJLoader::recalculateNormals(mesh1);
    OBJLoader::recalculateTangents(mesh1);

    OBJLoader::reverseWinding(mesh1);
}
catch (const OBJException& ex)
{
    std::cout << ex.what() << "\n";
}
```

`recalculateNormals(mesh1)` Berechnet die Normalen in mesh1 neu.

`recalculateTangents(mesh1)` Berechnet die Tangenten in mesh1 neu.

Wenn die Dreiecke eines Meshes mal in der falschen Winding Order in der OBJ Datei sind, kann man diese mit `reverseWinding(...)` umdrehen. Das sollte aber fast nie nötig sein, die meisten Programme wie Blender exportieren die Dreiecke richtigerum.

Texturen laden

Zum Laden von Texturen ist eine sehr einfache Bibliothek namens *SOIL* eingebunden. Das steht für *Simple OpenGL Image Library*. SOIL kann ein paar Texturformate laden:

BMP, PNG, JPG, TGA, DDS, PSD, HDR

Genauerer dazu auf der Seite des Entwicklers: <http://www.lonesock.net/soil.html>

Zuerst müsst ihr SOIL.h includieren:

```
#ifndef _GAME_H_
#define _GAME_H_
#include "GameWindow.h"
#include "OBJLoader.h"
#include <SOIL.h>

class Game : public GameWindow
{
...
}
```

An die Rohdaten einer Textur kommt man so:

```
GLuint width;
GLuint height;
GLuint channels;

unsigned char* image = SOIL_load_image("../assets/textures/burned_wood_diff.png", &width, &height,
&channels, SOIL_LOAD_RGBA);

if (image == 0)
{
    std::cerr << "Fehler: Textur konnte nicht geladen werden.\n";
}
else
{
    //Hier OpenGL Textur mit den Rohdaten aus image bauen

    SOIL_free_image_data(image);
}
```

`image` ist dann ein Pointer auf die Rohdaten der Textur, in `width`, `height` und `channels` stehen anschließend Höhe, Breite und Anzahl der Farbkanäle. Wenn ihr die Daten schlussendlich in eine OpenGL Textur verfrachtet habt, denkt daran `SOIL_free_image_data(...)` aufzurufen um den Speicher, den SOIL reserviert hat wieder freizugeben.

`SOIL_LOAD_RGBA` sagt SOIL, dass es RGB und Alpha Channel Daten laden soll. Das geht natürlich nur wenn die Textur diese Daten hat. Es ist sinnvoll sich auf ein Format zu einigen, z.B. RGBA mit vier Kanälen pro Textur. Man überlegt sich, wie man seine Materialien aufbauen möchte. Beispiel:

Jedes Model bekommt zwei RGBA Texturen:

Textur1: Diffuse Farbe des Models in RGB, Gloss Map in A

Textur2: Normalen in RGB, Alpha Map des Models in A

=> 4 Texture Maps in 2 Texturen!

SOIL hat auch noch ein paar anderen Funktionen. Mehr dazu gibt es auf der Seite des Entwicklers.

Aus den Rohdaten (image) von oben erzeugt man dann eine OpenGL Textur:

```
GLsizei width;
GLsizei height;
GLsizei channels;

unsigned char* image = SOIL_load_image("../assets/textures/burned_wood_diff.png", &width, &height,
                                     &channels, SOIL_LOAD_RGBA);

GLuint texid = 0;           //Handle auf die OpenGL Textur
if (image == 0)
{
    std::cerr << "Fehler: Textur konnte nicht geladen werden.\n";
}
else
{
    //Hier OpenGL Textur mit den Rohdaten aus image bauen

    glGenTextures(1, &texid);
    if (texid == 0)
    {
        std::cerr << "gl texture object creation failed." << std::endl;
    }
    glBindTexture(GL_TEXTURE_2D, texid);
    glTexImage2D(GL_TEXTURE_2D,
                0,
                GL_RGBA8,
                width,
                height,
                0,
                GL_RGBA,
                GL_UNSIGNED_BYTE,
                image,           //Texturdaten an OpenGL übergeben
                );
    glGenerateMipmap(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, 0);

    SOIL_free_image_data(image);           //den Speicher den SOIL reserviert hat freigeben.
}
```

Zu Texturformaten und glTexImage2D gibt es in der OpenGL Referenz weitere Informationen:

<https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexImage2D.xhtml>

<https://www.khronos.org/opengl/wiki/Texture>

SOIL ist insgesamt sehr einfach gestrickt. Ihr könnt natürlich eure eigenen Texture Loader bauen (was sehr viel Aufwand bedeuten kann) oder später mächtigere Bibliotheken einbinden wie zum Beispiel OpenCV: <http://opencv.org/>

Beim Umgang mit OpenGL Texturformaten macht man sehr schnell Fehler, da es unglaublich viele Möglichkeiten gibt Texturen anzulegen. Ein Fehler äußert sich meistens darin, dass die Objekte die man mit seinen Texturen bespannt einfach schwarz sind. Das ist natürlich wenig hilfreich, deswegen wird im nächsten Abschnitt ein sehr simples Hilfsmittel vorgestellt das einem das Leben aber stark erleichtern kann.

Schnelles OpenGL Error Checking

Damit man einen Hinweis bekommt, wo denn überhaupt der OpenGL Fehler aufgetreten ist, gibt es zwei Makros die man benutzen kann. Diese sind in der Datei „glerror.h“ im „framework“ Ordner definiert.

GLERR

Dieses Makro kann man einfach hinter alle möglichen OpenGL Befehle schreiben.

Ein Beispiel:

```
//Render a frame
void Game::render(GLdouble time)
{
    //glClear(GL_COLOR_BUFFER_BIT);
    glClear(42); GLERR
}
```

Das ist natürlich ein sehr offensichtlicher Fehler, aber als Beispiel in Ordnung.

Die mit der Konstante „42“ kann glClear(...) kann damit logischerweise nichts anfangen und wird nicht funktionieren. Ohne GLERR würde einfach gar nichts passieren. Das Gerenderte sieht nicht richtig aus aber das war es auch schon.

Mit GLERR wird eine Fehlermeldung ausgegeben und zusätzlich in die Datei „glerrorlog.txt“ geloggt:

```
An OpenGL error occured at file: "C:\Users\Fabian\Desktop\CMake Test\CMakeProject\src\Game.cpp", line:
43:
invalid value
```

Datei und Zeile geben den Ort an, an dem GLERR stand, als der Fehler aufgetreten ist. Deswegen ist es sinnvoll GLERR immer direkt hinter den OpenGL Befehl zu schreiben, weil sonst die Zeilenangabe nicht stimmt.

„invalid value“ ist der Name des OpenGL Fehlers. Oft sagt der in bestimmten Situationen nicht viel aus. In diesem Fall geht man zur OpenGL Referenz-Seite für die Funktion und sieht nach was den Fehler auslösen kann:

<https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glClear.xml>

Man findet unter „Errors“:

Errors

GL_INVALID_VALUE is generated if any bit other than the four defined bits is set in *mask*.

GL_INVALID_OPERATION is generated if glClear is executed between the execution of [glBegin](#) and the corresponding execution of [glEnd](#).

Und kann nun prüfen, welche der Bedingungen für den Fehlercode zutrifft und wenn ja, warum.

GLERR kann man getrost hinter jeden OpenGL Befehl schreiben den man im Code benutzt. Möchte man das Projekt releasen, kann man GLERR „deaktivieren“ um Performance zu sparen.

GLERR wirft standardmäßig eine Exception die man irgendwo im Code abfangen kann, z.B. ganz oben in der main.cpp. Auch das kann man noch einstellen.

Jetzt erst mal das zweite Makro:

checkglerror()

Anders als GLERR ist dieses Makro dazu gedacht auch im Release-Programm eingesetzt zu werden. Es wirft keine Exceptions und kann nicht deaktiviert werden. Wie GLERR gibt es die Fehlermeldung aus, gibt aber zusätzlich noch einen bool-Wert zurück, der angibt ob ein OpenGL Fehler aufgetreten ist oder nicht. Das kann man dazu benutzen um auf Fehler zu reagieren die auch im Release-Programm auftreten könnten:

```
//Render a frame
void Game::render(GLdouble time)
{
    //glClear(GL_COLOR_BUFFER_BIT);
    glClear(42);
    if (checkglerror())
    {
        std::cerr << "glClear failed.\n";
    }
}
```

Eine sinnvollere Anwendung dafür wäre, zu überprüfen ob die Textur von oben erfolgreich an OpenGL gegeben werden konnte (Lädt man eine Texturdatei mit einem anderen Format als man angibt, ist das ein Fehler den man nicht im Programm verhindern kann):

```
glGenTextures(1, &texid);
if (texid == 0)
{
    std::cerr << "gl texture object creation failed." << std::endl;
}
glBindTexture(GL_TEXTURE_2D, texid);
glTexImage2D(GL_TEXTURE_2D,
    0,
    GL_RGBA8,
    width,
    height,
    0,
    GL_RGBA,
    GL_UNSIGNED_BYTE,
    image //Texturdaten an OpenGL übergeben
);
if(checkglerror()) //wenn das fehlschlägt gebe eine Fehlermeldung aus und
{                //werfe eine Exception
    std::cerr << "Textur hat das falsche Format." << std::endl;
    throw std::exception();
}
glGenerateMipmap(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, 0);

SOIL_free_image_data(image); //den Speicher den SOIL reserviert hat freigeben.
```


Einstellungen für GLERR und checkglerror()

Im „framework“ Ordner befindet sich noch eine Datei namens „fw_config.h“. Hier könnt ihr noch ein paar Einstellungen setzen:

```
#ifndef _FW_CONFIG_H
#define _FW_CONFIG_H

//Debug settings for glerror.h
#define THROW_ON_GL_ERROR 1           //throw exception when OpenGL error occurs
#define HOLD_ON_GL_ERROR 0           //print error and wait for a key when OpenGL error occurs
#define LOG_GL_ERRORS 1              //Log all errors to "glerrorlog.txt"
#define CGA2_DEBUG                    //if not defined, GLERR does nothing

#endif
```

THROW_ON_GL_ERROR

- 1** GLERR wirft Exceptions, wenn OpenGL Fehler auftreten
- 0** GLERR wirft keine Exceptions

HOLD_ON_GL_ERROR

- 1** GLERR wartet auf einen Tastendruck wenn ein OpenGL Fehler aufgetreten ist. Das gibt einem Zeit einen Breakpoint in der Nähe des Fehlers zu setzen um direkt Werte von Variablen etc. zu überprüfen.
- 0** GLERR wartet nicht auf einen Tastendruck

LOG_GL_ERRORS

- 1** Fehlermeldungen werden von GLERR und checkglerror() in die Datei „glerrorlog.txt“ geschrieben.
- 0** Es werden keine Fehlermeldungen geloggt.

CGA2_DEBUG

- definiert** GLERR arbeitet wie oben beschrieben
- nicht definiert** GLERR wird an allen Stellen im Code durch Nichts ersetzt. Das spart Performance in der Releaseversion.

Möchte man GLERR deaktivieren, kommentiert man am besten einfach nur die Zeile aus:

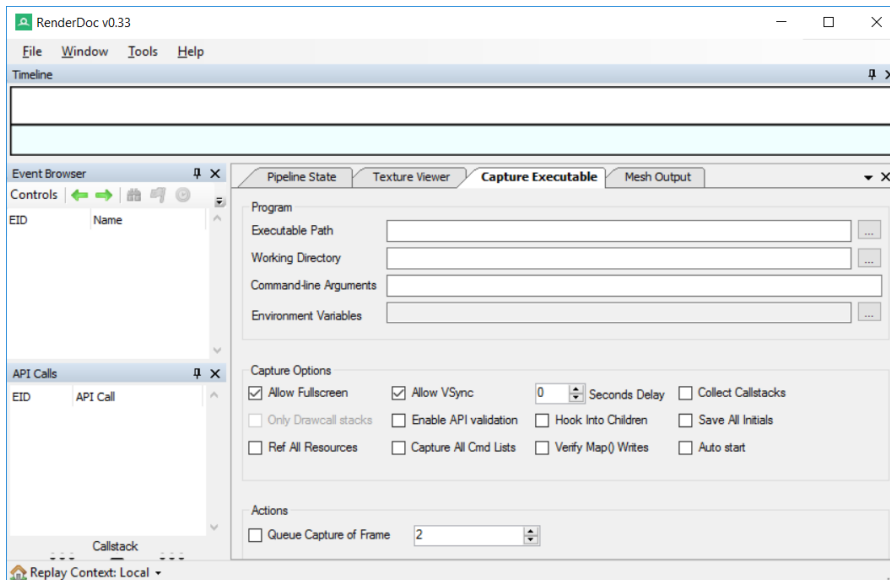
```
//Debug settings for glerror.h
#define THROW_ON_GL_ERROR 1           //throw exception when OpenGL error occurs
#define HOLD_ON_GL_ERROR 0           //print error and wait for a key when OpenGL error occurs
#define LOG_GL_ERRORS 1              //Log all errors to "glerrorlog.txt"
//#define CGA2_DEBUG                    //if not defined, GLERR does nothing
```

Grafikdebugger: RenderDoc

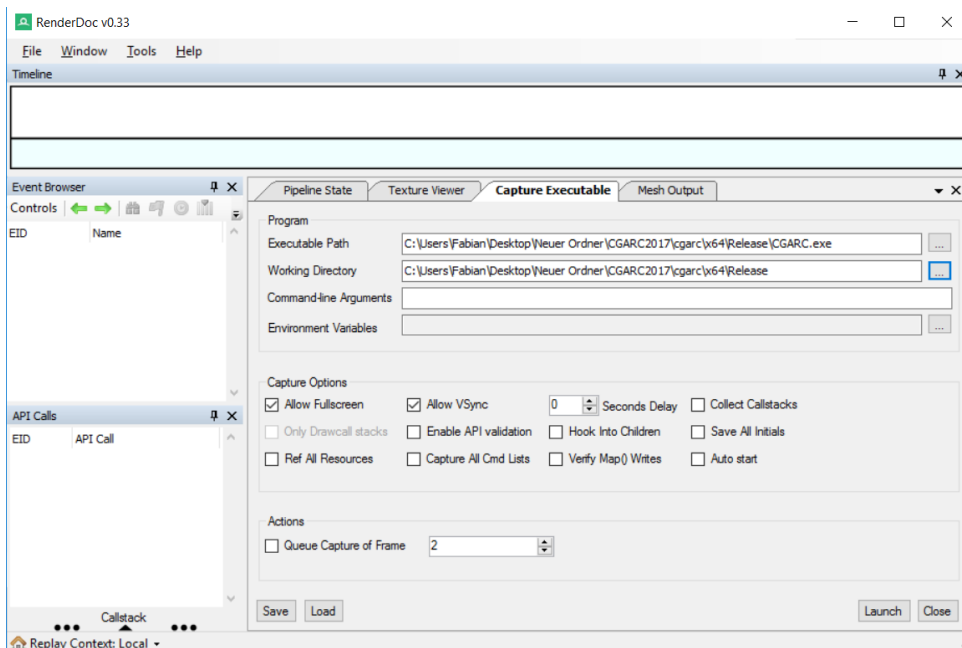
Hier <https://renderdoc.org/builds> findet ihr die Downloads zum Grafikdebugger *RenderDoc*.

Er funktioniert unter Windows und Linux. Auf OS X soll es auch laufen aber das Feature „capture and replay“ wird noch nicht unterstützt: <https://github.com/baldurk/renderdoc/wiki/Linux-and-OS-X-support>

Wenn ihr RenderDoc startet geht zum Reiter *Capture Executable*:

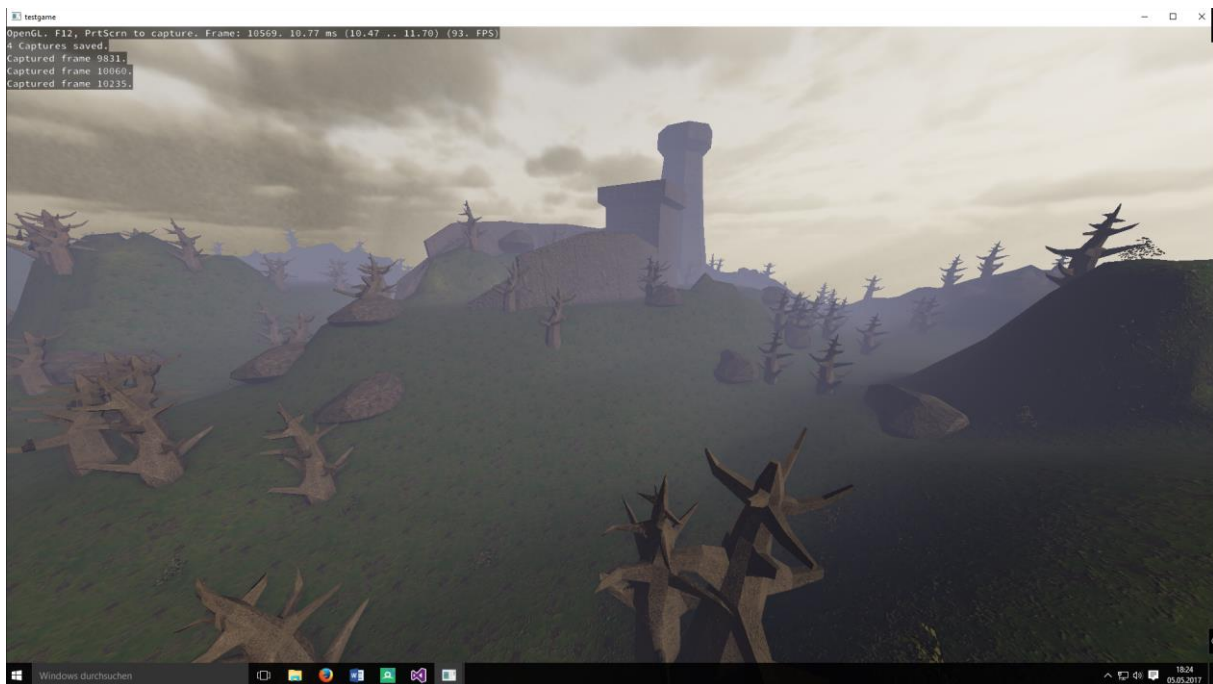


Dort wählt ihr den Pfad zum kompilierten, ausführbaren Programm aus, sowie das Arbeitsverzeichnis (Auf diesen Pfad beziehen sich alle relativen Pfade im Programm):

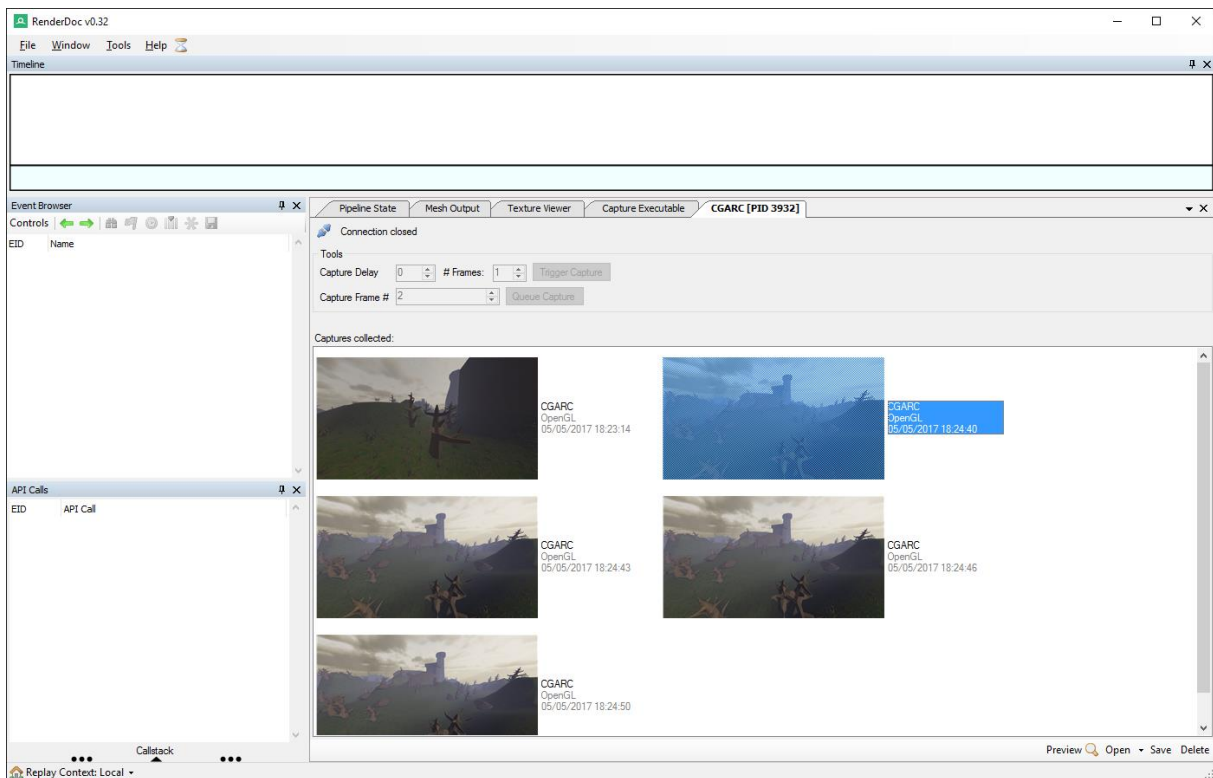


Drückt auf *Launch*.

Euer Programm öffnet sich und ihr könnt mit F12 ein paar Frames aufnehmen:



Beendet euer Programm, damit es nicht stört und geht wieder zu RenderDoc. Hier seht ihr jetzt die aufgenommenen Frames:



Ladet einen Frame indem ihr auf ihn doppelklickt.

Auf der linken Seite seht ihr jetzt den *Event Browser*:

The screenshot shows two panels from a graphics debugger. The top panel, 'Event Browser', lists events for 'Frame #9831'. It includes a 'Frame Start' event, followed by a 'Colour Pass #1 (5 Targets + Depth)' which contains several 'glClear' and 'glDrawArrays' calls. The bottom panel, 'API Calls', lists the specific OpenGL calls made during the frame, including 'glBindFramebuffer', 'glDisable', 'glDepthFunc', 'glActiveTexture', 'glBindTexture', 'glUseProgram', and multiple 'glProgramUniform' calls, ending with 'glDrawArrays'.

EID	Name
0	Frame Start
2	glClear(Color = <0.000000, 0.000000, 1.000000, 1.000000...>)
12-13...	Colour Pass #1 (5 Targets + Depth)
13662	glClear(Color = <1.000000, 1.000000, 1.000000, 1.000000>)
13754	glDrawArrays(6)
13762	glClear(Color = <1.000000, 1.000000, 1.000000, 1.000000>)
13773	glDrawArrays(6)
1378...	Colour Pass #2 (1 Targets + Depth)
13782	glClear(Color = <0.000000, 0.000000, 1.000000, 1.000000...>)
13923	glDrawArrays(6)
13927	glBlitFramebuffer(ResID_2449, ResID_0)
13940	glDrawArrays(36)
13942	SwapBuffers()

EID	API Call
13928	glBindFramebuffer
13929	glDisable
13930	glDepthFunc
13931	glActiveTexture
13932	glBindTexture
13933	glUseProgram
13934	glProgramUniformVector*
13935	glProgramUniformVector*
13936	glProgramUniformMatrix*
13937	glProgramUniformMatrix*
13938	glProgramUniformVector*
13939	glBindVertexArray
139...	glDrawArrays

Oben seht ihr alle generellen Kommandos, die ihr für den Frame abgesetzt habt und speziell eure Render Passes. Die könnt ihr aufklappen und nachschauen welche Kommandos in dem Pass ausgeführt wurden.

Weiter unten seht ihr die dazugehörigen *API-Calls*. Diese könnt ihr aufklappen und seht die Parameter, die ihr übergeben habt.

Auf der rechten Seite habt ihr den Reiter *Pipeline State*. Hier könnt ihr euch die gesamte Grafikkipeline ansehen, Inputs, Outputs und States untersuchen.

The screenshot shows the 'Pipeline State' window. At the top, there's a tab bar with 'Pipeline State' selected. Below it, a 'Display Controls' section has checkboxes for 'Show Disabled Items', 'Show Empty Items', and 'Export'. The main part of the window displays a sequence of pipeline stages: VTX, VS, TCS, TES, GS, Rasterizer, FS, FB, and CS. The 'VTX' stage is highlighted with a red box. Below the stages, there are two tables: 'Vertex Attribute Formats' and 'Buffer bindings'. The 'Vertex Attribute Formats' table shows attributes 'vposition' and 'texCoords' with their respective formats and buffer slots. The 'Buffer bindings' table shows two buffers, both pointing to 'Buffer 2461'. At the bottom right, there are two preview windows: 'Mesh View' showing a 3D wireframe model of a complex object, and 'Primitive Topology' showing 'TriangleList' with three triangles.

Index	Enabled	Name	Format/Generic Value	Buffer Slot	Relative Offset	Go
0	Enabled	vposition	GL_FLOAT2	0	0	→
1	Enabled	texCoords	GL_FLOAT2	1	0	→

Index	Buffer	Stride	Offset	Divisor	Byte Len	Go
0	Buffer 2461	16	0	0	96	→
1	Buffer 2461	16	8	0	96	→

Auf den Programmierbaren Stages könnt ihr euch den Shader Code ansehen:

The screenshot shows a pipeline editor with stages: VTX, VS (highlighted with a red box), TCS, TES, GS, and Rasterizer. Below the pipeline, the 'Shader' section shows 'Vertex Shader 11' with a 'View' button circled in red. The 'source0.glsl' tab is active, displaying the following GLSL code:

```
1 #version 330 core
2 layout (location = 0) in vec2 vposition;
3 layout (location = 1) in vec2 texCoords;
4
5 out vec2 UV;
6
7
8
9 void main()
10 {
11     gl_Position = vec4(vposition.x, vposition.y, 0.0f, 1.0f);
12     UV = texCoords;
13 }
```

Ihr könnt euch die Uniforms für eine Shader Stage anzeigen lassen (klick auf Go):

The screenshot shows the same pipeline editor with 'VS' highlighted. The 'Vertex UBO 0' tab is active, displaying a list of uniform variables:

Name	V...	Type
model_matrix	{...}	float4x4
model_matrix.row0	1...	float4
model_matrix.row1	0...	float4
model_matrix.row2	0...	float4
model_matrix.row3	0...	float4
projection_matrix	{...}	float4x4
projection_matrix...	0...	float4
projection_matrix...	0...	float4
projection_matrix...	0...	float4
projection_matrix...	0...	float4
view_matrix	{...}	float4x4
view_matrix.row0	0...	float4
view_matrix.row1	0...	float4
view_matrix.row2	...	float4
view_matrix.row3	0...	float4

Below the list, there is a 'Go' button. The 'Uniforms and UBOs' section at the bottom shows 'Uniforms' with '3 Variables' and a 'Go' button.

Die *FB* Stage zeigt euch eure Framebuffer und den State der damit zusammenhängt. Über *Go* könnt ihr euch die Inhalte der Buffer im *Texture Viewer* anzeigen lassen.

Display Controls | Show Disabled Items | Show Empty Items | Export

VTX → VS → TCS → TES → GS → Rasterizer → FS → **FB** → CS

Draw Buffers

Slot	Resource	Type	Width	Height	Depth	Array Size	Format	Go
0	Texture2DMS 1000000000000000007 (GL_FRAMEBUFFER_SRGB = 0)	2D MS	1920	1061	1	1	SRGB8_ALPHA8	⇒
Depth	Texture2DMS DSV 1000000000000000008	2D MS	1920	1061	1	1	DEPTH24_STENCIL8	⇒
Stencil	Texture2DMS DSV 1000000000000000008	2D MS	1920	1061	1	1	DEPTH24_STENCIL8	⇒

Target Blends

Slot	Enabled	Colour Src	Colour Dst	Colour Op	Alpha Src	Alpha Dst	Alpha Op	Write Mask
0	False	ONE	ZERO	ADD	ONE	ZERO	ADD	RGBA

Blend State

Blend Factor:	
	0.00, 0.00, 0.00, 0.00

Depth State

Enabled:	Write:
✓	✓
Bounds: ✗ 0.000 - 0.000	
Func:	LESS

Stencil State

Enabled:	Face	Func	Fail	Depth Fail	Pass	Ref	Write Mask	Val Mask
✗	Front	ALWAYS	KEEP	KEEP	KEEP	00	FF	FF
	Back	ALWAYS	KEEP	KEEP	KEEP	00	FF	FF

Das ist sehr hilfreich, wenn ihr später deferred rendert und einfach nichts so ist wie es sein sollte.

Mesh Output zeigt euch was mit euer Geometrie passiert. Ihr könnt Input und Output der Geometrie Stages untersuchen:

Pipeline State | **Mesh Output** | Texture Viewer | Capture Executable | CGARC [PID 3932] | main() | main()

Controls | Sync Views | Highlight Vertices | Row Offset: 0 | Instance: 0

VS Input

VTX	IDX	position	uv
0	0	49.34252 13.00411 23.19499	0.6467
1	1	65.25054 12.94103 8.19992	0.6465
2	2	52.45707 14.72829 19.15776	0.6346
3	3	42.49758 21.20934 7.26221	0.3796
4	4	45.24244 27.83049 4.71555	0.3362
5	5	45.24244 21.20935 4.71555	0.3808
6	3	42.49758 21.20934 7.26221	0.3796
7	6	42.49758 34.45163 7.26221	0.2906

VS Output

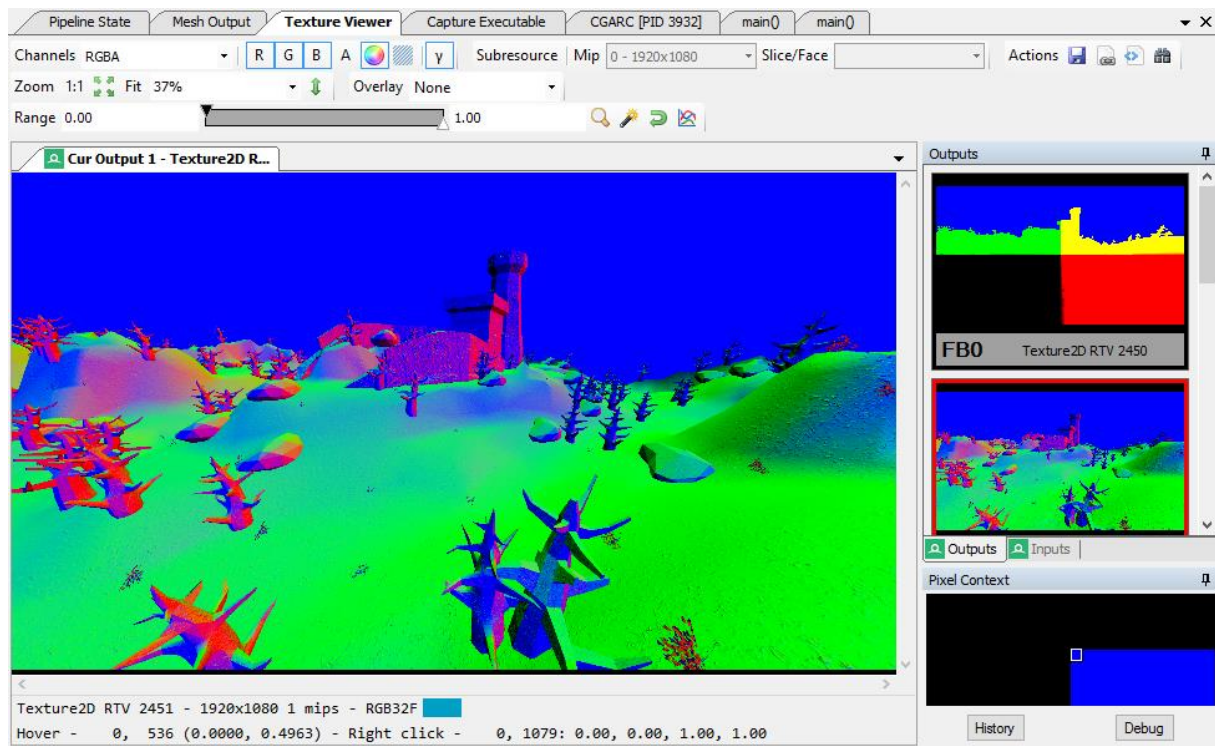
VTX	IDX	gl_Position
0	0	4.64829 10.91468 67.11384
1	1	9.59471 16.04301 85.30618
2	2	5.45941 13.58319 71.10332
3	3	-0.73545 21.01195 76.7671
4	4	0.13826 27.15089 77.71987
5	5	0.12401 21.8949 79.86599
6	3	-0.73545 21.01195 76.7671
7	6	-0.70694 31.52394 72.47485

Preview

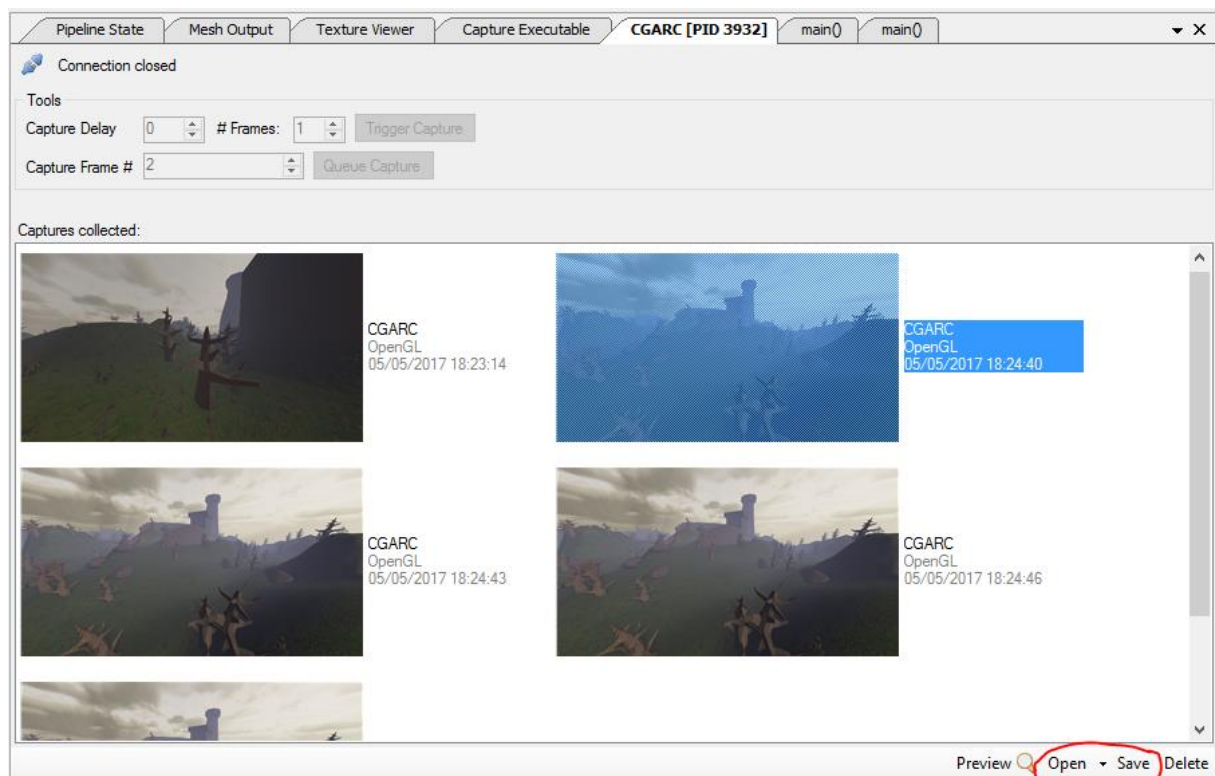
VS Input | VS Output | GS/DS Output

Arcball | Only this draw | Solid Shading: None | Wireframe

Im *Texture Viewer* könnt ihr Input-Texturen, also das was ihr den Shadern als Sampler mitgebt, und Output-Texturen, das was in diesem Pass bis jetzt auf dem Framebuffer gelandet ist, untersuchen. Hier sieht man zum Beispiel einen Output der die Normalen aus Sicht der Kamera enthält:



Die ganze Capture-Session kann man schlussendlich auch speichern und später wieder laden, wenn man die gesammelten Daten irgendwo oder irgendwann anders untersuchen möchte:



Alles in Allem ist RenderDoc ein extrem nützliches Tool, wenn es darum geht unerklärliche Grafik Bugs aufzudecken.

Sehr hilfreich in Kombination mit dem Tool sind Debug Renderings. Dabei kommentiert ihr in eurem Code alle anderen Renderings aus und farb-codiert bestimmte Daten die ihr genauer untersuchen wollt, wie zum Beispiel verrücktspielende UV-Koordinaten.

Manchmal kann es passieren, das RenderDoc eine Anwendung zum Absturz bringt, weil es sich in den Grafiktreiber einhakt und dadurch manchmal unvorhergesehene Dinge passieren.

Links zu Tutorials, Referenzen, Bibliotheken,...

Hier noch ein paar Links die vielleicht hilfreich sein könnten:

Tutorials:

Das wahrscheinlich beste Tutorial im Moment: <https://learnopengl.com/>

Auch nicht schlecht: <http://www.opengl-tutorial.org/>

Das OpenGL Tutorial auf dieser Seite ist etwas alt und auch recht leer, aber von den DirectX Tutorials kann man oft auch etwas mitnehmen. Das einzig schlechte an der Seite ist die schlechte Lesbarkeit des Codes: <http://www.rastertek.com/tutindex.html>

Bücher:

Ein gutes Standardwerk, Rückwärtskompatibilität zu älteren OpenGL Versionen wird allerdings stark vernachlässigt:

https://www.amazon.de/OpenGL-Programming-Guide-Official-Learning/dp/0134495497/ref=sr_1_1?ie=UTF8&qid=1494004814&sr=8-1&keywords=opengl+programming+guide

Referenzen:

C++: <http://www.cplusplus.com/reference/>
<http://en.cppreference.com/w/>

Aufpassen! Recht verständlich, aber nicht immer ganz standardkonform:

<https://msdn.microsoft.com/de-de/library/3bstk3k5.aspx>

<https://msdn.microsoft.com/de-de/library/csc687y.aspx>

OpenGL: <https://www.khronos.org/registry/OpenGL-Refpages/gl4/>

Eine ziemlich gute Idee ist es, sich diese handliche Reference Card auszudrucken:

https://www.opengl.org/sdk/docs/reference_card/opengl45-reference-card.pdf

Bibliotheken:

DIE Bibliothek für die Arbeit mit Bilddaten und Allem was dazu gehört:

<http://opencv.org/>

AssImp ist eine Bibliothek die fast alle geläufigen 3D-Model-Formate laden kann:

<http://assimp.sourceforge.net/>

Falls ihr irgendetwas findet, was C++ nicht kann, boost kann es:

<http://www.boost.org/>

Standard Bibliotheken für einzelne Bildformate:

KTX: <https://www.khronos.org/opengles/sdk/tools/KTX/doc/libktx/>

PNG: <http://www.libpng.org/pub/png/libpng.html>

JPEG: <http://libjpeg.sourceforge.net/>

Tools:

Es gibt noch ein interessantes Tool, das man zum Erzeugen von komprimierten und unkomprimierten Texturformaten aller Art aus beliebigen Bilddateien verwenden kann. Das Tool kommt aus der *GPUOpen* Initiative von *AMD*. Mit dem Tool kann man z.B. DXT-komprimierte DDS und KTX Dateien erstellen.

<http://gpuopen.com/gaming-product/compressorator/>

Und der direkte GitHub-Link:

<https://github.com/GPUOpen-Tools/Compressorator>