VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA

Tutorial:

Programming tools/IDEs

CGRA 354 : Computer Graphics Programming

Instructor: Dr Alex Doronin
Cotton Level 3, Office 330
alex.doronin@vuw.ac.nz

With slides from: Holly Rushmeier (Yale), Marc Gluyas (VUW), Abigail Clennell (VUW)

Next five lectures

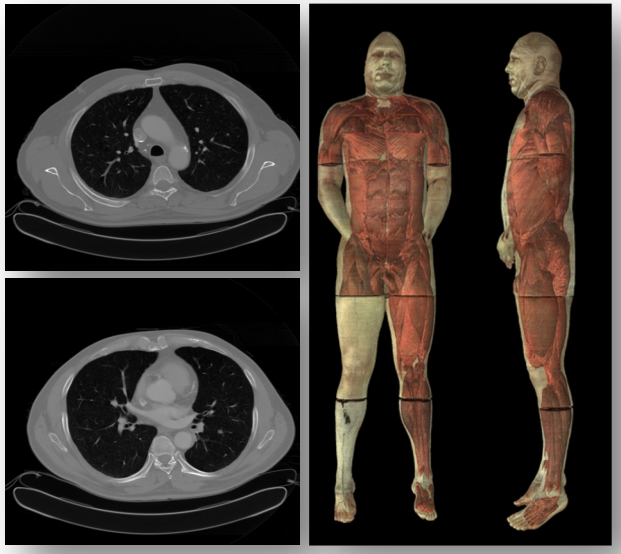
- Computer Graphics and C++ recap
- C++ Recap continued and introduction to OpenGL programming
- **C++/OpenGL programming continued:**
  - **3D Geometry and GUI**
  - **Shading and color**
  - **Introduction to Lighting**

Subsequently:

- 3D Transformations, etc.

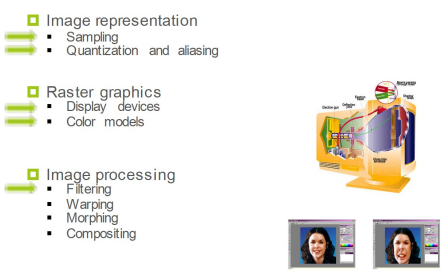
Recap: Applications: Discovery, Design, Communication, Expression

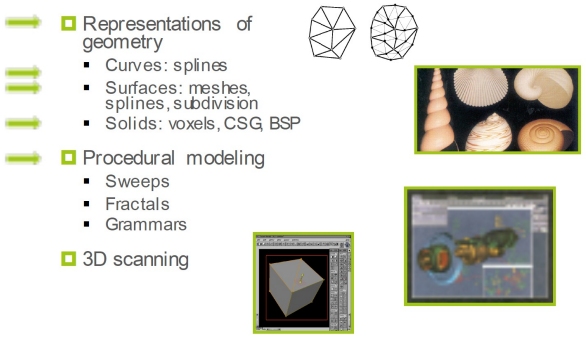
- Movies
- Games
- Computer-aided design
- Scientific visualization
- Medical imaging
- Training
- Education
- E-commerce
- Graphical User Interfaces

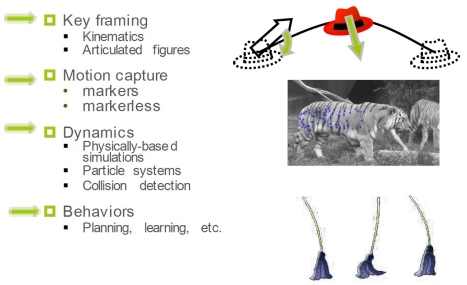


Areas in Computer Graphics

- **Imaging** = representing 2D images
- **Modeling** = representing 2D/3D objects
- **Rendering** = 2D images from 2D/3D models
- **Animation** = simulating changes over time

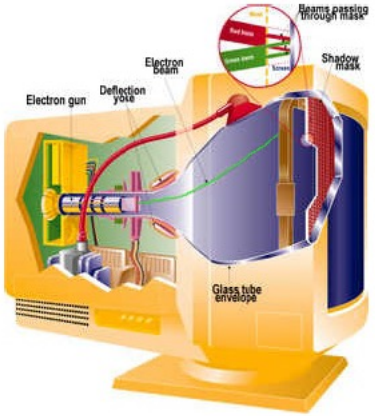


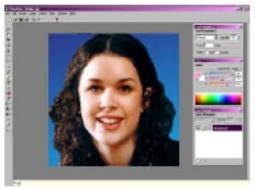




Imaging

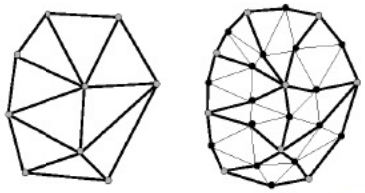
- Image representation
  - Sampling
  - Quantization and aliasing
- Raster graphics
  - Display devices
  - Color models
- Image processing
  - Filtering
  - Warping
  - Morphing
  - Compositing

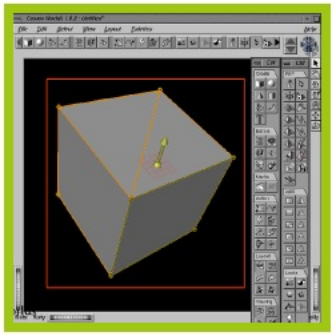




Modeling

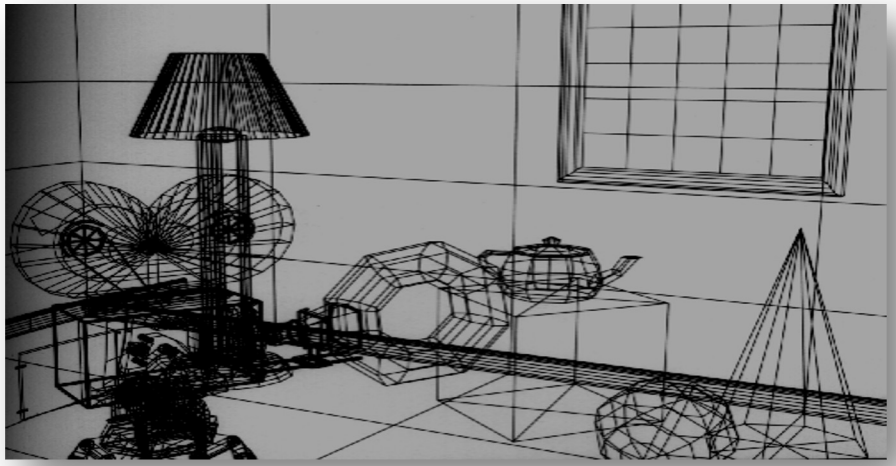
- Representations of geometry
  - Curves: splines
  - Surfaces: meshes, splines, subdivision
  - Solids: voxels, CSG, BSP
- Procedural modeling
  - Sweeps
  - Fractals
  - Grammars
- 3D scanning

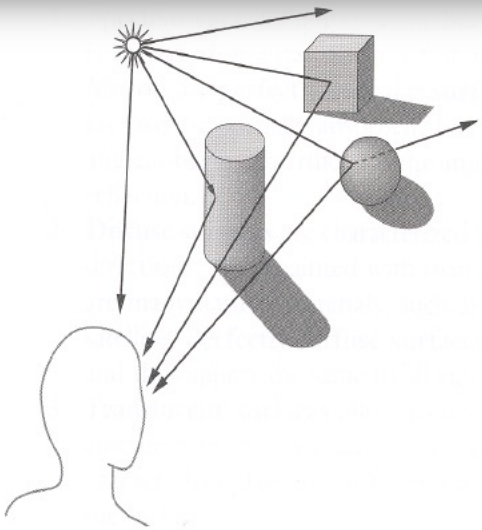




Rendering

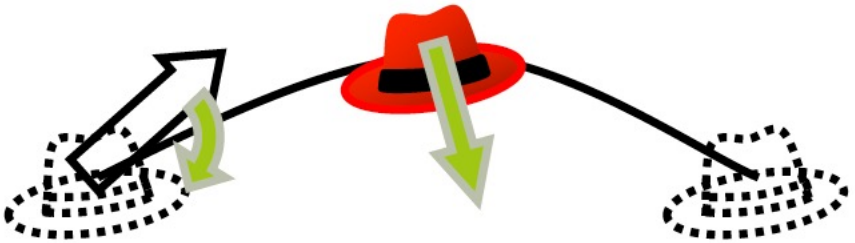
- 3D rendering pipeline
  - Modeling transformations
  - Viewing transformations
  - Hidden surface removal
  - Illumination, shading and textures
  - Scan conversion, clipping
  - Hierarchical scene graphics
  - OpenGL/WebGL
- Global Illumination
  - Ray/Path tracing
  - Radiosity

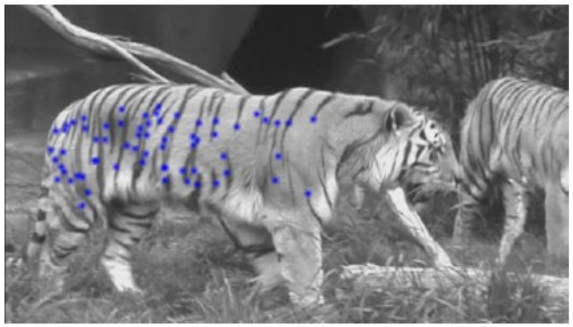




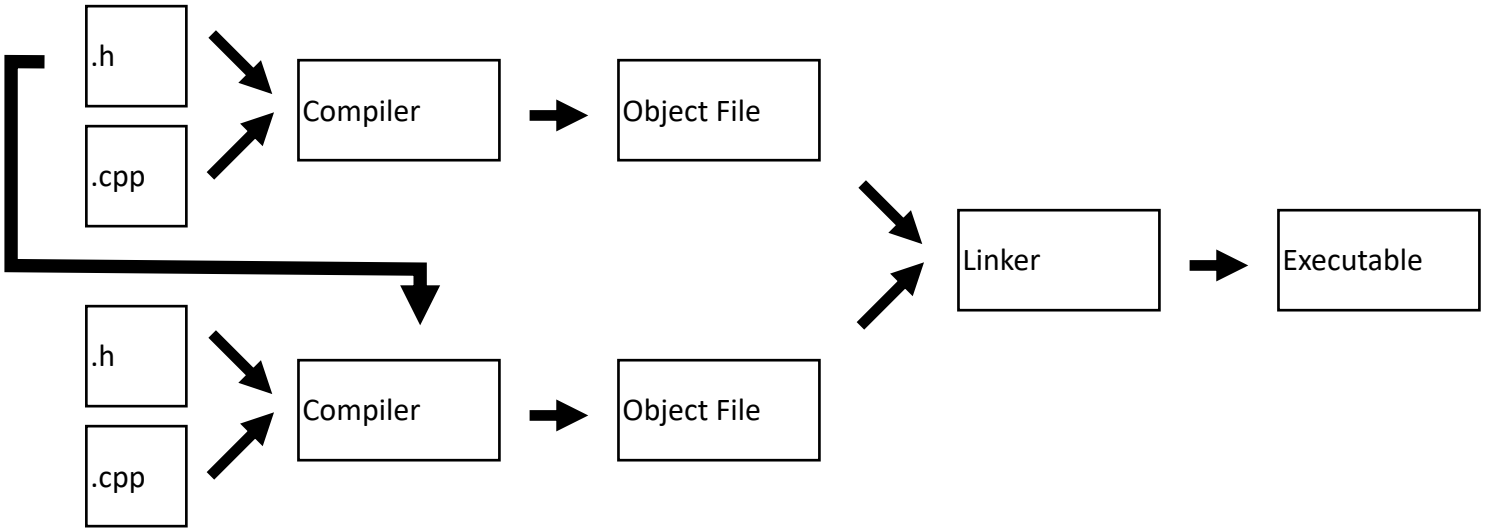
Animation

- Key framing
  - Kinematics
  - Articulated figures
- Motion capture
  - markers
  - markerless
- Dynamics
  - Physically-based simulations
  - Particle systems
  - Collision detection
- Behaviors
  - Planning, learning, etc.





We are using C++ and OpenGL



Program structure:

- Headers: *.h
- Source files: *.cpp
- Creates a binary file
- Program entry point: **main()**

Open Graphics Library:

- API for GPU-accelerated 2D / 3D rendering
- Cross-platform (Windows, OS X, Linux, iOS, Android, ...)

OpenGL SC™

Today:

- *Overview of Integrated development environments (IDEs) on Windows, Mac, Linux and Cmake tools*

Building a C++ program?

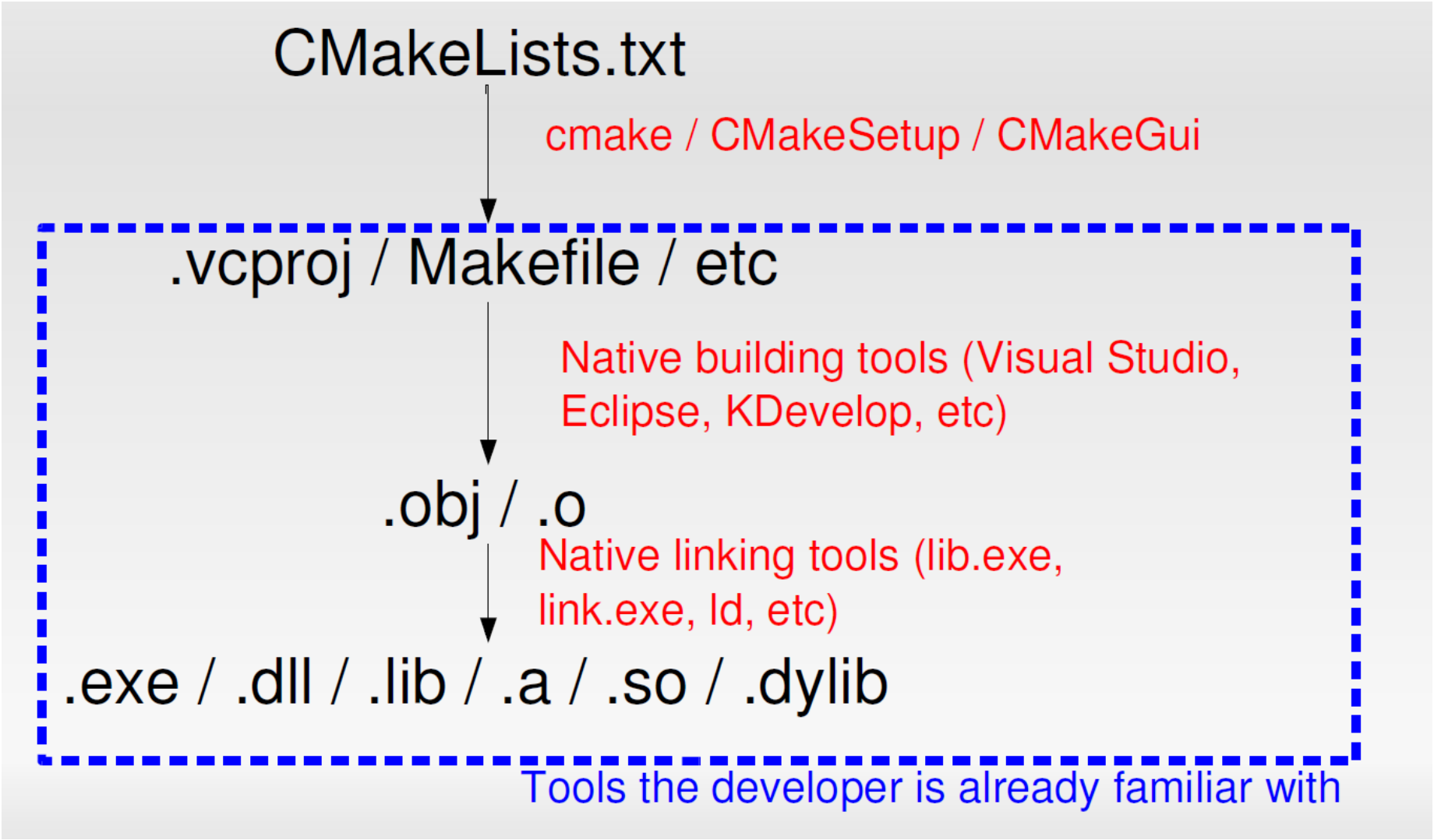
- You write an application (source code) and need to:
  - Compile the source
  - Link to other libraries
  - Distribute your application

What is Cmake?

- Think of it as a unified Make
- CMake is used to control the software compilation process using simple platform and compiler independent configuration files
- CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice: Visual C++, Kdevelop3, Eclipse, XCode, makefiles (Unix, NMake, Borland, Watcom, MinGW, MSYS, Cygwin), Code::Blocks etc
- Projects are described in CMakeLists.txt files (usually one per subdir)

https://cmake.org/

Build flow


CMakeLists.txt
cmake / CMakeSetup / CMakeGui
.vcproj / Makefile / etc
Native building tools (Visual Studio, Eclipse, KDevelop, etc)
.obj / .o
Native linking tools (lib.exe, link.exe, ld, etc)
.exe / .dll / .lib / .a / .so / .dylib
Tools the developer is already familiar with

Hello CMake

- PROJECT(helloworld)
- SET(hello_SRCS hello.cpp)
- ADD_EXECUTABLE(hello ${hello_SRCS})

Dependencies

- FIND_PACKAGE(Qt4 REQUIRED)
- Cmake includes finders (FindXXXX.cmake) for many software packages
- If using a non-CMake FindXXXX.cmake, tell Cmake where to find it by setting the CMAKE_MODULE_PATH variable
- Think of FIND_PACKAGE as an #include

Integrated Development Environment (IDE)

- Integrated Development Environment – allows the automation of many of the common programming tasks in one environment
  - Writing the code
  - Checking for Syntax (Language) errors
  - Compiling and Interpreting(Transferring to computer language)
  - Debugging (Fixing Run-time or Logic Errors)
  - Running the Application

Common IDEs: Visual Studio/Xcode

- Platforms that allows the development and deployment of desktop, mobile and web applications
- Allows user choice of many languages
  - May program in One of them
  - May create different parts of application in different languages

    C++

    C# (C Sharp)

    Java

    Etc.

Based on Project and Solution Concepts

User creates a new project in Visual Studio/Xcode

- A solution and a folder are created at the same time with the same name as the project
- The project belongs to the solution
- Multiple projects can be included in a solution

Solution

- Contains several folders that define an application's structure
- Solution files have a file suffix of e.g. .sln

Project: contains files for a part of the solution

- Project file is used to create an executable application
- A project file has a suffix
- Every project has a type (Console, Windows, etc.)

Creating and opening solutions: README.md

Windows

Visual Studio

This project requires at least Visual Studio 2015. You can get the latest, [Visual Studio Community 2017](https://www.visualstudio.com/downloads/), for free from Microsoft.

Product	XX
Visual Studio 2015	14
Visual Studio 2017	15

Run the `cmake` command for Visual Studio with the appropriate version number (XX).

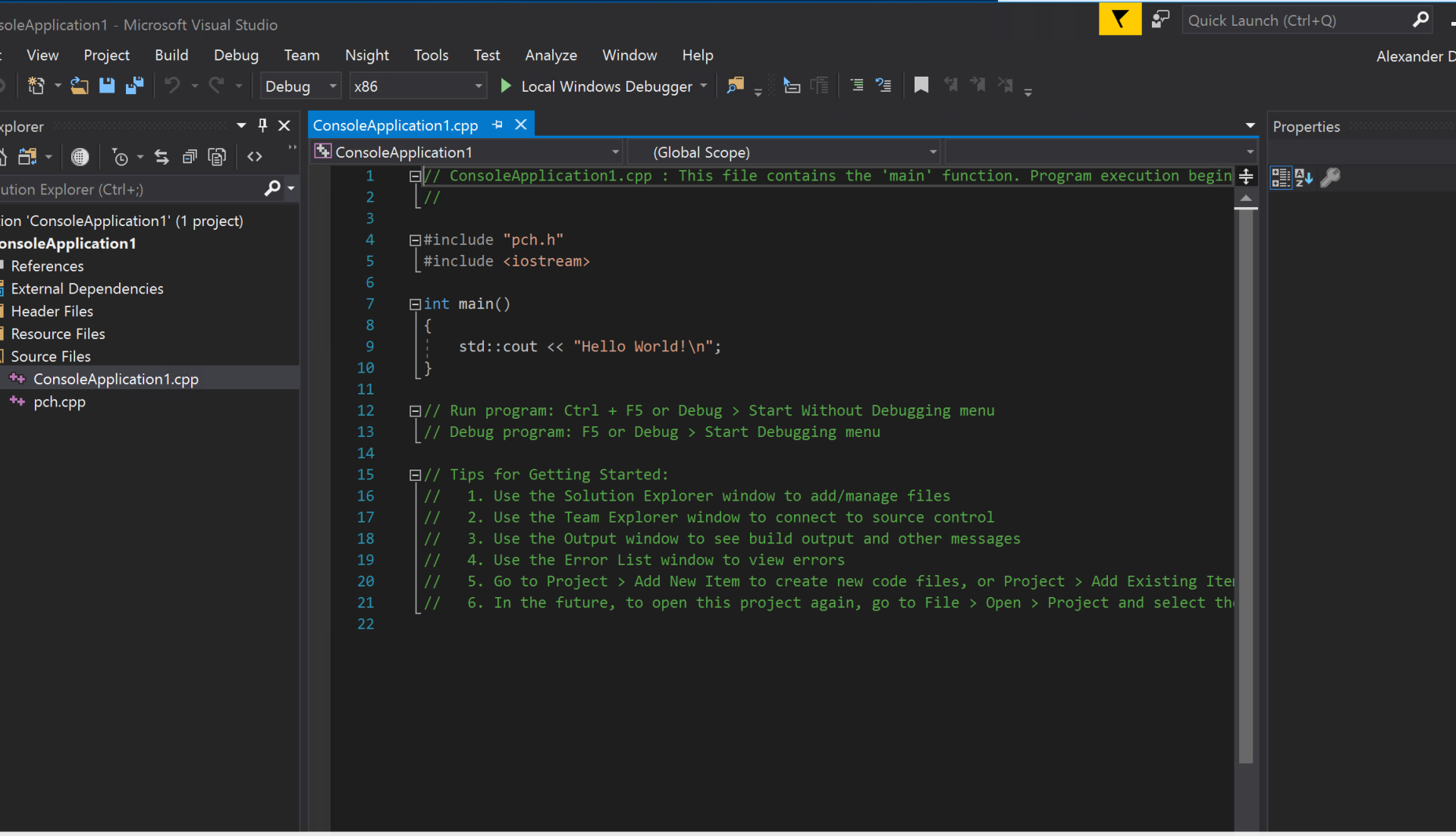
```sh
> cmake -G "Visual Studio XX" ..\work
```

Or if you are building for 64-bit systems.

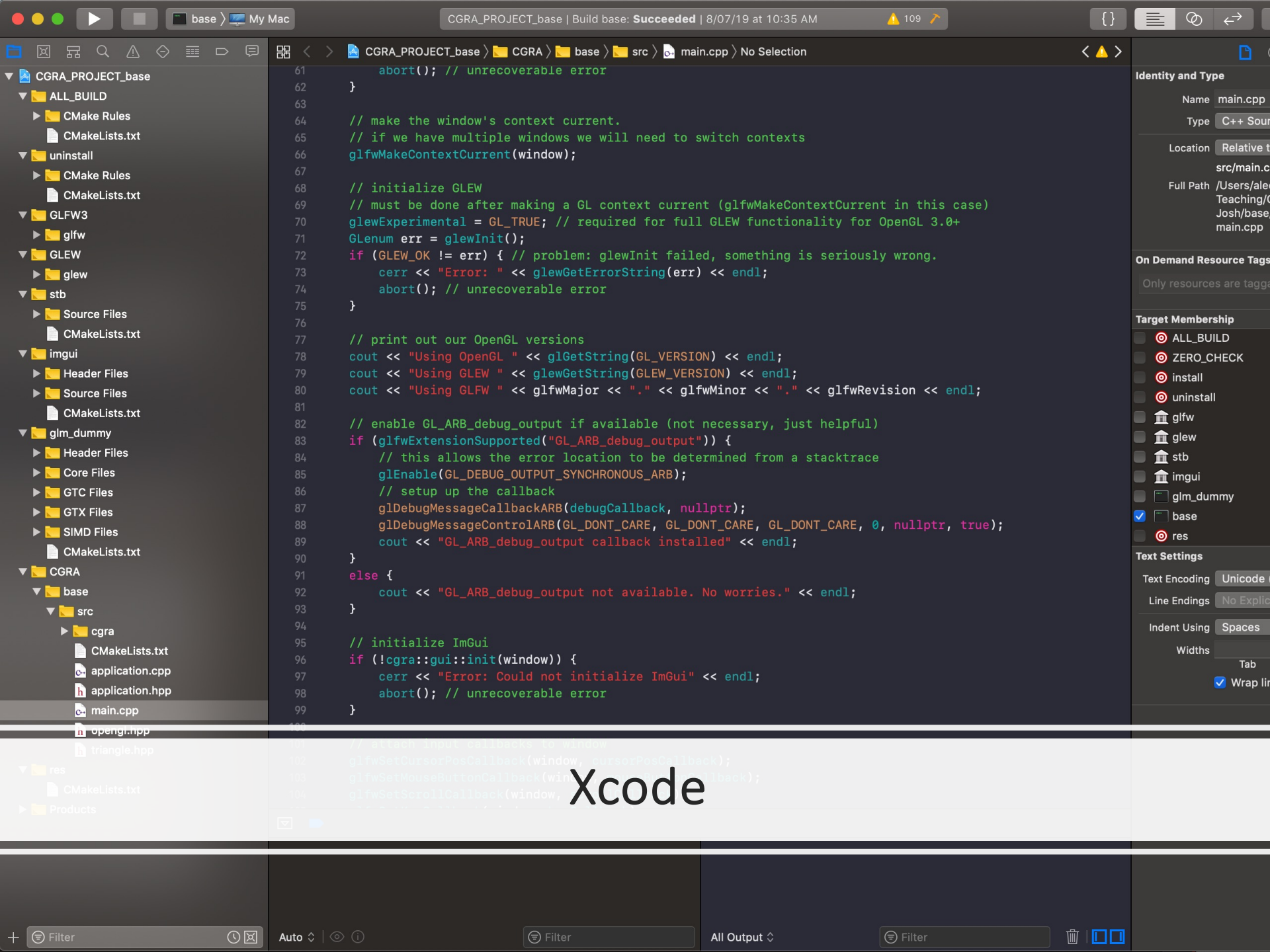
```sh
> cmake -G "Visual Studio XX Win64" ..\work
```

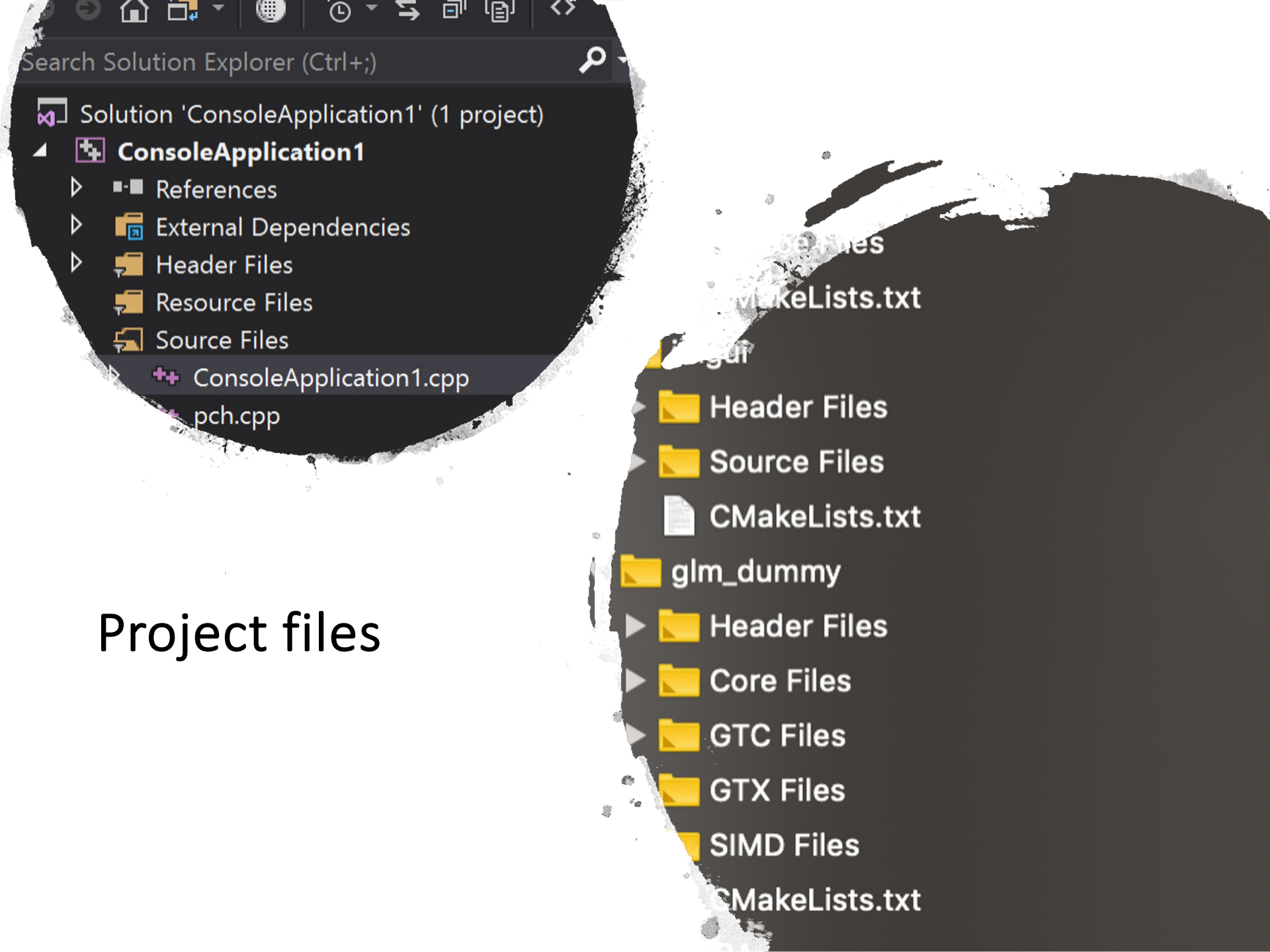
After opening the solution (`.sln`) you will need to set some additional variables before running.

- `Solution Explorer > base > [right click] > Set as StartUp Project`
- `Solution Explorer > base > [right click] > Properties > Configuration Properties > Debugging`
  - Select `All Configurations` from the configuration drop-down
  - Set `Working Directory` to `(SolutionDir)../work`
  - Set `Command Arguments` to whatever is required by your program



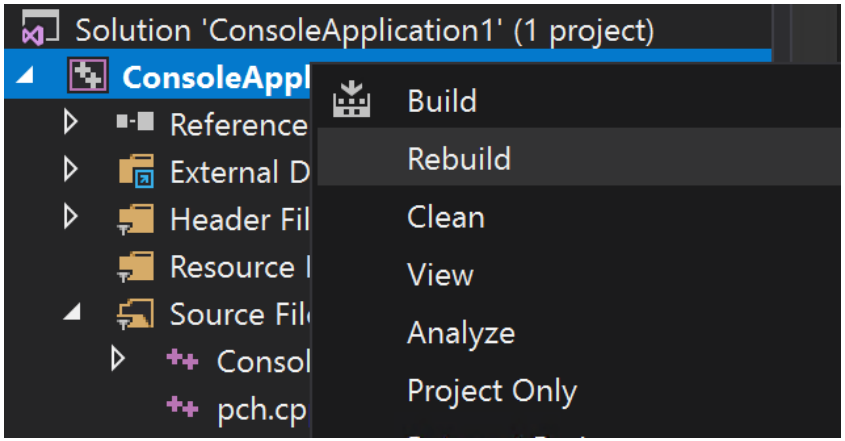
Visual studio


base My Mac
CGRA_PROJECT_base | Build base: Succeeded | 8/07/19 at 10:35 AM
CGRA_PROJECT_base CGRA base src main.cpp No Selection
CGRA_PROJECT_base
ALL_BUILD
CMake Rules
CMakeLists.txt
uninstall
CMake Rules
CMakeLists.txt
GLFW3
glfw
GLEW
glew
stb
Source Files
CMakeLists.txt
imgui
Header Files
Source Files
CMakeLists.txt
glm_dummy
Header Files
Core Files
GTC Files
GTX Files
SIMD Files
CMakeLists.txt
CGRA
base
src
cgra
CMakeLists.txt
application.cpp
application.hpp
main.cpp
triangle.hpp
res
CMakeLists.txt
Products
abort(); // unrecoverable error
}
// make the window's context current.
// if we have multiple windows we will need to switch contexts
glfwMakeContextCurrent(window);
// initialize GLEW
// must be done after making a GL context current (glfwMakeContextCurrent in this case)
glewExperimental = GL_TRUE; // required for full GLEW functionality for OpenGL 3.0+
GLenum err = glewInit();
if (GLEW_OK != err) { // problem: glewInit failed, something is seriously wrong.
cerr << "Error: " << glewGetErrorString(err) << endl;
abort(); // unrecoverable error
}
// print out our OpenGL versions
cout << "Using OpenGL " << glGetString(GL_VERSION) << endl;
cout << "Using GLEW " << glewGetString(GLEW_VERSION) << endl;
cout << "Using GLFW " << glfwMajor << "." << glfwMinor << "." << glfwRevision << endl;
// enable GL_ARB_debug_output if available (not necessary, just helpful)
if (glfwExtensionSupported("GL_ARB_debug_output")) {
// this allows the error location to be determined from a stacktrace
glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS_ARB);
// setup up the callback
glDebugMessageCallbackARB(debugCallback, nullptr);
glDebugMessageControlARB(GL_DONT_CARE, GL_DONT_CARE, GL_DONT_CARE, 0, nullptr, true);
cout << "GL_ARB_debug_output callback installed" << endl;
}
else {
cout << "GL_ARB_debug_output not available. No worries." << endl;
}
// initialize ImGui
if (!cgra::gui::init(window)) {
cerr << "Error: Could not initialize ImGui" << endl;
abort(); // unrecoverable error
}
Identity and Type
Name main.cpp
Type C++ Sour
Location Relative t
src/main.c
Full Path /Users/ale
Teaching/
Josh/base
main.cpp
On Demand Resource Tags
Only resources are tagg
Target Membership
ALL_BUILD
ZERO_CHECK
install
uninstall
glfw
glew
stb
imgui
glm_dummy
base
res
Text Settings
Text Encoding Unicode
Line Endings No Explic
Indent Using Spaces
Widths
Tab
Wrap li
Filter
Auto
Filter
All Output
Filter
Xcode



Project files

Building projects



Details

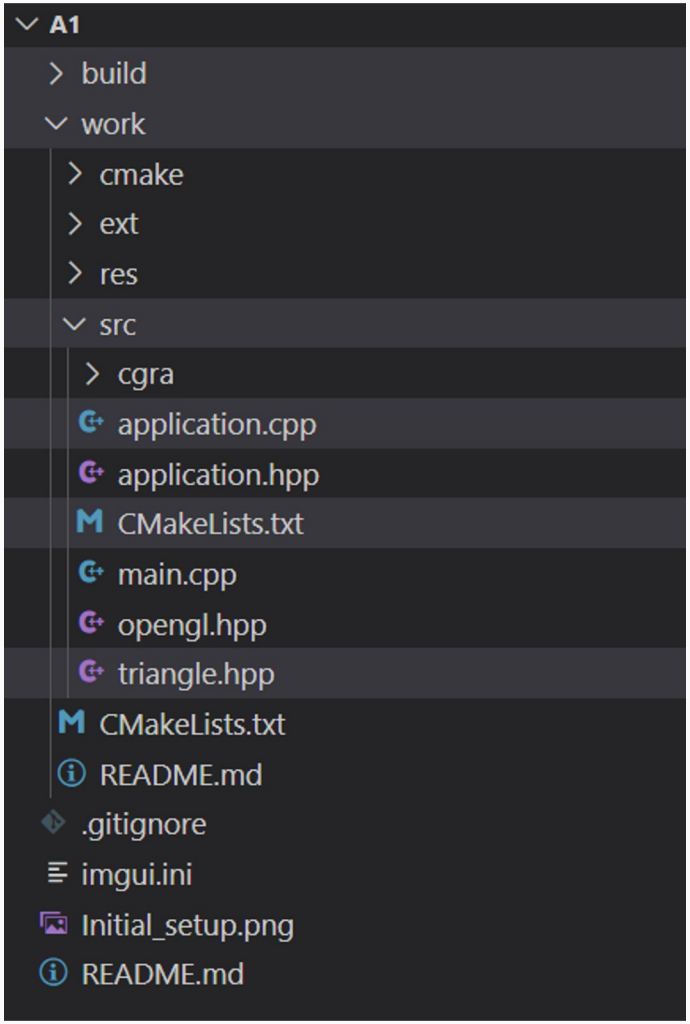
First Steps:

1. Install required tooling (if working on non-ECS machine)
   - C++ Compiler / IDE
   - Update OpenGL 3.30 drivers (if needed, but probably already installed)
   - CMake
2. Download and compile the CGRA framework
3. Create your source files (src/`objfile.h`, src/`objfile.cpp`) and add them to src/`CMakeLists.txt`
4. Implement your class with hard-coded placeholder geometry (eg: a tetrahedron) before attempting `loadObj`
   - Use `src/triangle.hpp` as example code for `setup()`, `draw()`, and `destroy()`

CGRA Framework Overview

CGRA Framework Key Components

`README.md`: Very detailed and incredibly helpful!
`build`: Where you will generate your platform-specific project files with CMake. Delete the contents of this when you submit your assignment.
`work`: Contains all the actual content.
`work/src`: Put all your source files here.
`work/res`: Contains a teapot asset and shader code.
`src/CMakeLists.txt`: Tells CMake what files to include in the build; update this *every time* you add create a new source file or folder. Be careful not to confuse this with the `CMakeLists.txt` in the root folder.
`src/application.cpp`: This is the main class you'll be modifying to instantiate and call out to your `.obj` parser.
`src/triangle.hpp`: Provides a very simple geometry implementation. You can use this as a base for your `build()`, `draw()`, and `destroy()` methods, but note that it may structure its data in a different way to you.



Adding Files and Folders to CMake

Every folder which contains C++ code contains a `CMakeLists.txt` file, which you need to update every time you add a new file or folder.
For Assignment 1, you only need to add code to `work/src`, and so you only need to work with `work/src/CMakeLists.txt`.
When creating a new file, find the `SET(sources ...)` section of the file, and add your filename.
When creating a new folder, add an `add_subdirectories()` statement with the name of your folder to the `CMakeLists.txt` in its parent directory. Then, create a `CMakeLists.txt` in your folder. You can use the code on the right as a template.

```
#############################################################
# Source Files
#############################################################

# ----TODO------------------- #
# list your source files here #
# --------------------------- #
SET(sources
          "application.hpp"
          "application.cpp"

          "opengl.hpp"

          "main.cpp"

          "triangle.hpp"

          "CMakeLists.txt"
)

# ----TODO---------------------- #
# list your subdirectories here #
# ----------------------------- #
add_subdirectory(cgra)
```

Building with CMake

We use CMake to allow cross-platform compilation. CMake uses files called `CMakeLists.txt` in multiple folders to describe how to compile the project.
CMake does not compile the program. Instead, it generates native build configurations (makefiles, project/solution files, etc.) for your computer, which you can then use with the compiler you have installed.

Several recent versions of IDEs (Visual Studio, CLion) have built-in support for CMake projects, so you don't have to manually re-run CMake when you add a new file to the project.

There is a platform-specific way for each platform (covered in readme.md), but in general the way it works is:

- Navigate your terminal to `./build`
- Run `cmake` with the path `../work`
  - This will generate the files in `./build`
- Run your normal compiler/build system
- `cd ..` to return to the root directory, then call `./build/base` to run your project

CLion

1. Cross-platform Windows/Mac/Linux, available on ECS computers
2. Based on IntelliJ - familiar
3. CMake integration, don’t need to manually generate projects, can auto-add files to CMakeLists.txt
4. Free for university students

CLion setup

Sign up with your uni email https://www.jetbrains.com/shop/eform/students

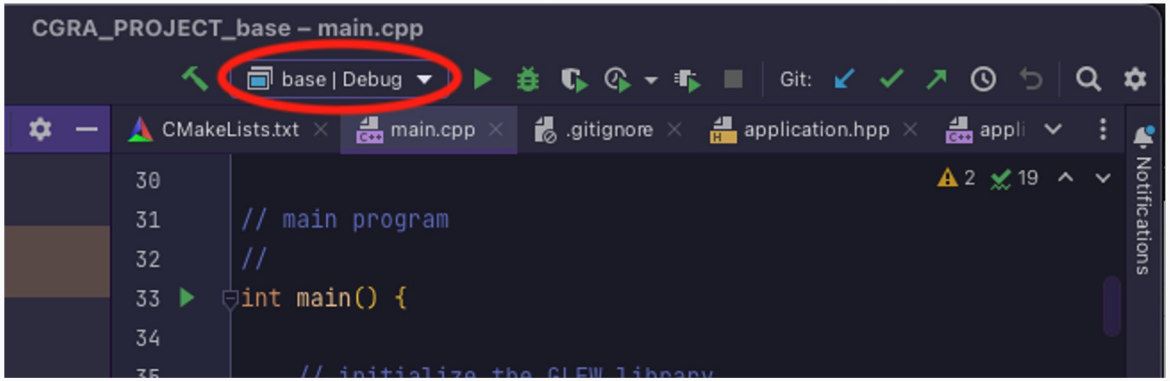
Compiler configuration:

Windows https://www.jetbrains.com/help/clion/quick-tutorial-on-configuring-clion-on-windows.html

Mac https://www.jetbrains.com/help/clion/quick-tutorial-on-configuring-clion-on-macos.html

Open the work/ folder of the framework download, and make sure the “base” target is selected

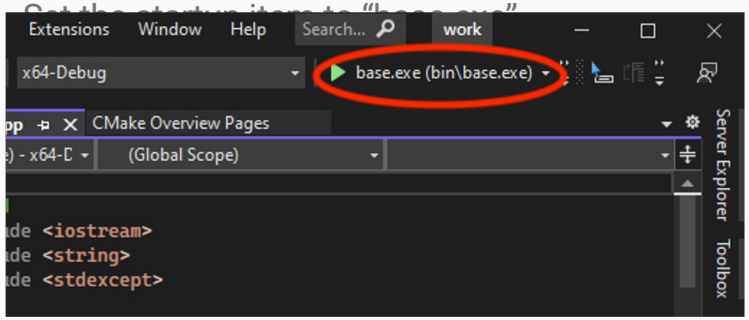
This is the one with your code in it

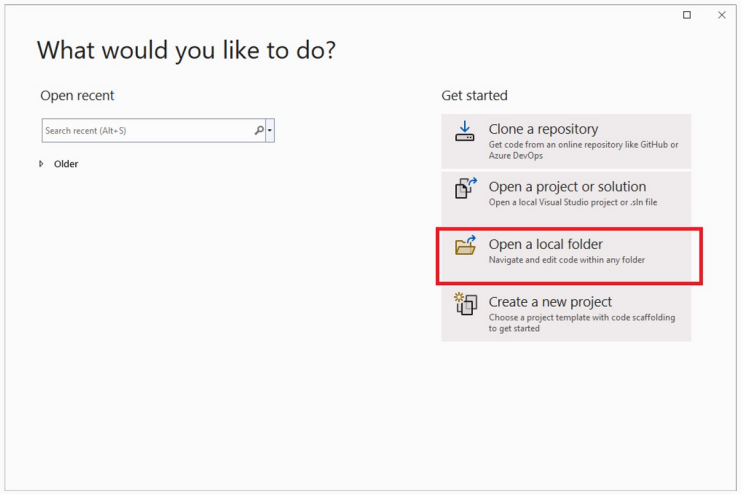


CMake in Visual Studio (Windows)

CMake in VS was introduced in VS17 but support has changed in new versions
https://docs.microsoft.com/en-us/cpp/build/cmake-projects-in-visual-studio?view=msvc-170

1. Install CMake tools when installing Visual Studio
2. Open the work/ folder of the framework with the “Open a local folder option”
3. Set the startup item to “base.exe”





Building for XCode (MacOS)

1. Open a Terminal window and navigate to the "`.../base/build`" directory. Use "`cd`" to change directory.
2. Call the command "`cmake -G "XCode" ../work`". This will build your framework so that it can be opened in XCode.
3. Change the scheme to "base"

Useful links

https://docs.microsoft.com/en-us/visualstudio/ide/getting-started-with-cpp-in-visual-studio?view=vs-2019

https://www.geeksforgeeks.org/introduction-to-visual-studio/

https://codewithchris.com/xcode-tutorial/