General Robust Interface Tracking (GRIT) Meshes in 2D

Overview and Introduction
by
Kenny Erleben and Marek Krzysztof Misztal

8 Steps

- STEP 1: History of development of GRIT
- STEP 2: Getting the code
- STEP 3: Install dependencies
- STEP 4: Run CMake
- STEP 5: Compile, link and run
- STEP 6: Understanding an application using GRIT
- STEP 7: Write your own application
- STEP 8: How to make a movie

STEP 1 — A little history

Short History of GRIT

- The original idea of mixing implicit level-set interface tracking with explicit interface representation using simplicial complexes (triangle mesh) was the sole result of Andreas Bærentzen work.

- The idea where later extended to 3D in the deformable simplicial complex (DSC) method. The work was driven by Marek Misztal during his PhD and later post doc time at DTU. We strongly recommend his thesis as the main background source.

- As one of many demonstrator cases of DSC a multiphase liquid simulation was developed in collaboration between mainly researchers from DTU, UCPH and Alexandra Institute. This lead to a long series of simulation papers exploring and improving the DSC method. We perceive this work as helping mature the DSC method.

- After many years of experience with DSC the simulation researchers at UCPH meet the performance barrier of DSC in the simulation needs. This was the birth of GRIT. A parallel GRIT was first prototyped by master thesis student Mark Jensen.

History Contd.

- GRIT builds on the experiences from DSC, but is radical different in many aspects. Erleben and Misztal developed the algorithmic framework to overcome the sequential nature of the DSC method and the overhead in the DSC implementation at that time. Hence,
    - GRIT was designed to support domain decomposition through a copy-and-replace sub-domain strategy. Hence GRIT was born to support parallelism in at least a simple matter.
    - GRIT eliminated the sequential control flow in the original DSC algorithm by batching operations into categories. A movement operation batch, a vertex split operation batch, a relabelling operation batch, a refinement operation batch, a coarsening operation batch, smoothing operation batch, and optimisation operation batch. Leading to batching of similar computational operations that all could potentially be made parallel using a kind of red-black Jacobi blocking of the operations within a batch, but keeping batch execution sequential and synched.
    - GRIT was designed to support custom quality measures for all batch operations
    - GRIT was initially designed to support generic attribute vectors to support more easy adaption to various problems. Like defining a heat field, or a viscosity field or whatever field that needs to be included into the model.
    - GRIT abstracted over the actual mesh implementation and math types by defining a mesh interface in terms of topological algebra operations on simplicial complexes and using a math type binder for easily swapping math types.
    - DSC supported rollback mechanism which was abandoned in GRIT.

Future Focus

- GRIT started as a DSC method parallelization assignment. The domain decomposition and batching of the operations have lead to a generic framework for easily customising remeshing methods without the complicated sequential control flow from previous work.
    - Current ongoing work is exploring the interplay between different remeshing methods and needs by various simulation problems. We are particular interested in studying effects such as
        - Fracturing, separation and sliding of interfaces shared by multiple phases
    - Contributors to GRIT have made demonstrations of area maximization, Enright test, Zalesak Disk, hyper elastic deformable models, Newtonian liquids, Magnetostatics and more. We hope to study more problems
        - Rigid body motion, Mathematical Morphological Operations, Multiphase level-set segmentation, Meshing of distance fields, and many more
- The goal of Erleben and Misztal is that GRIT can become a computational paradigm for solving PDEs with complicated moving boundary conditions, such as the ones with dynamic solution dependence or inherent non-smoothness either in PDE model or in geometry representation.

References

- http://orbit.dtu.dk/en/publications/deformable-simplicial-complexes(df4bfb96-476b-4eee-8904-1cf70597cc28).html
- http://image.diku.dk/kenny/download/viinblad.jensen.14.pdf
- https://iphys.wordpress.com/2013/07/15/multiphase-flow-of-immiscible-fluids-on-unstructured-moving-meshes-2/
- https://iphys.wordpress.com/2012/07/18/multiphase-flow-of-immiscible-fluids-on-unstructured-moving-meshes/
- https://iphys.wordpress.com/2011/07/16/mathematical-foundation-of-the-optimization-based-fluid-animation-method/
- https://iphys.wordpress.com/2010/09/04/optimization-based-fluid-simulation-on-unstructured-meshes/
- https://iphys.wordpress.com/2014/09/27/finite-element-modeling-of-the-vocal-folds-with-deformable-interface-tracking/
- https://iphys.wordpress.com/2014/04/16/conforming-contact-manifolds-for-multibody-simulations/
- https://iphys.wordpress.com/2014/04/16/disjoint-domains-interactions-framework-for-hyperelastic-simulations/
- http://image.diku.dk/kenny/download/2012_SCA/
- http://image.diku.dk/kenny/download/2013_BELLAIRS/moving_meshes.pdf
- http://www2.compute.dtu.dk/~janba/DSC-webpage/

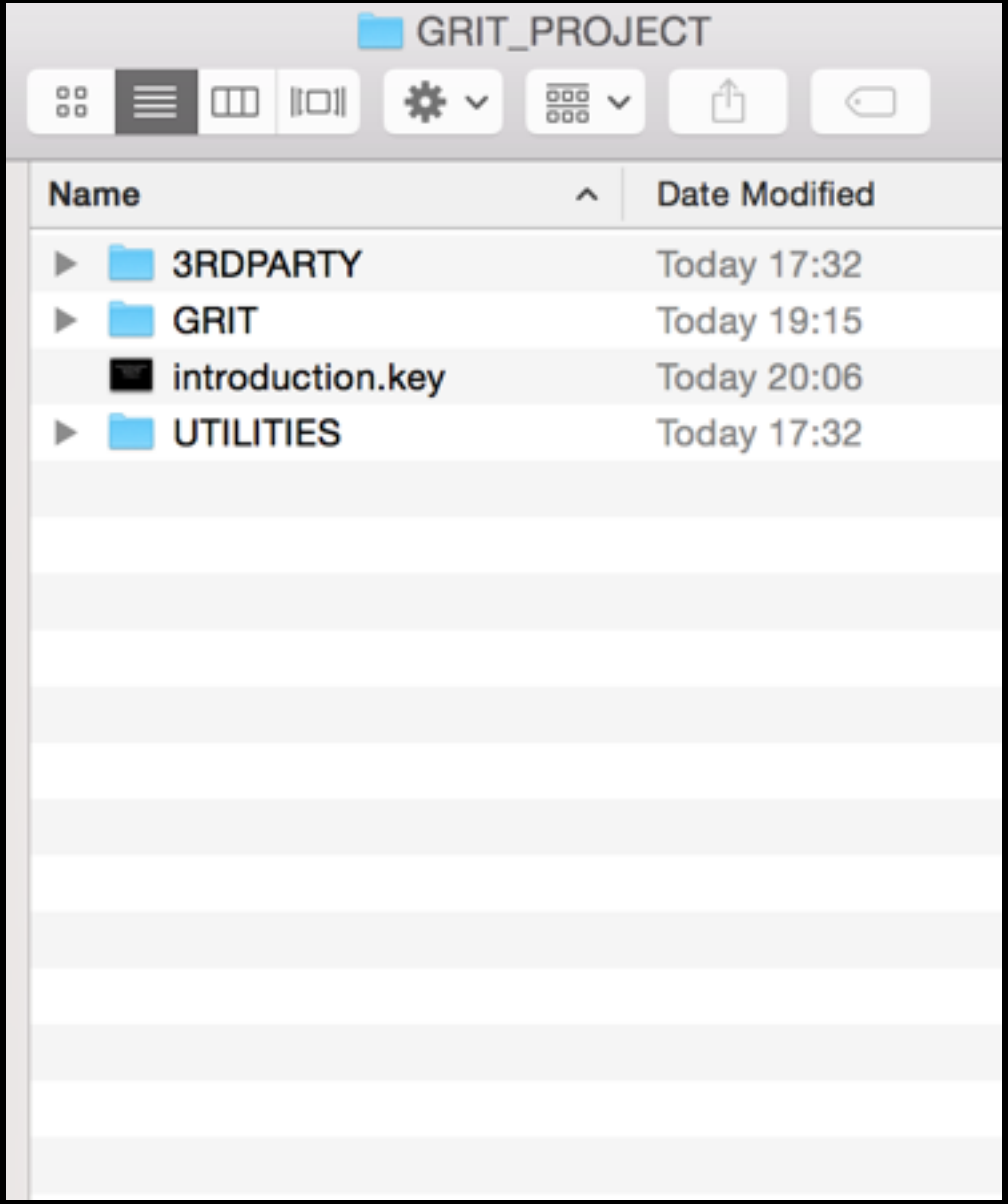
STEP 2 — Check out the code from svn

Use your favorite svn client and the svn URL

- Replace XXX below with your user account on the image.diku.dk server. If you do not have an account then write to kenny@di.ku.dk to obtained one

- svn+ssh://XXX@image.diku.dk/home/mmg/svn/CODE/GRIT_PROJECT

Will be replaced by github



Name	Date Modified
3RDPARTY	Today 17:32
GRIT	Today 19:15
introduction.key	Today 20:06
UTILITIES	Today 17:32

This is what you get from svn checkout

Now let us just briefly explain the content of this folder structure

Structure of Project Folder

- 3RDPARTY: Contains libraries not developed by us but used by us. In as much as possible we bundle other dependencies with our code to make it as self-contained as possible.
- GRIT: Contains the actual library code.
- UTILITIES: Contains a small collection of various tools for pre- or post processing.

Structure of GRIT Folder

- GLUE: Contains source code for the glue library.
- UTIL: Contains source code for the util library
- GRIT: Contains source code for the moving mesh library
- APPS: Contains demo applications showing examples of how to write an application that “glues” a simulator together with our mesh engine.
- SIMULATORS: Contains a small collection of various simulators.
- bin: Contains cfg-files for applications and binary executables, and mesh data files (See data subfolder).
- lib: Contains static binary library files
- unit_tests: Contains some unit tests for various functionality, you will likely only need to care about these if you wish to work on the core library functionality.

GRIT Library Overview

- GRIT library is the actual meshing library. It has a rich set of features and data types. Such as simplex sets, quality comparators, and much more.
- UTIL is a utility library making it easier to load config files, write log files and output matlab scripts etc. It contains many small utility tools, data types for making it convenient to write small demo applications. For instance a small tensor algebra library.
- GLUE is a helper library that makes it more simple to get started using GRIT. As the name suggest it is the “glue” between a simulator and a moving mesh. GLUE wraps GRIT interfaces and data structures into a simple array-based interface. GLUE may be simple to use but does not provide full functionality and comes with a performance penalty over using the raw GRIT directly from ones own application.
- SIMULATORS contains many black-box physics based simulators that we use in our demos and tutorials.
- APPS folder contains all our demos and tutorial examples.

Style of Code

- GRIT borrowed heavily from the OpenTissue (http://www.opentissue.org)in regards to how to structure and write library code.
- We would like in particular to high light the OpenTissue documentation on
  - Code standards,
    - http://www.opentissue.org/mediawiki/index.php/Code_standards
  - Good Practice,
    - http://www.opentissue.org/mediawiki/index.php/Good_Practice
  - Design Patterns,
    - http://www.opentissue.org/mediawiki/index.php/Design_Patterns
- In general the OpenTissue website contains many good hints for setting up svn and ssh to make it easier for developers to contribute. We strongly recommend looking through these pages to make life easier.

STEP 3 — Install code dependencies

Dependencies Overview

- DevIL: Used by util library for reading and writing image files (such as png or jpeg files)
- CUSP: Used by some simulators (not all) for solving linear systems. Requires NVIDIA CUDA Toolkit. The cusp version used by GRIT is bundled with 3RDPARTY.
- OpenTissue: Used by GRIT as the actual underlining mesh data structure. The sub-parts needed by GRIT is bundled with 3RDPARY. OpenTissue uses a few things from Boost numerics.
- Unit-tests in GRIT are written using the Boost unit test framework. This means you need to make sure your boost installation will install the “binaries” from Boost that supports this framework.

Install Checklist

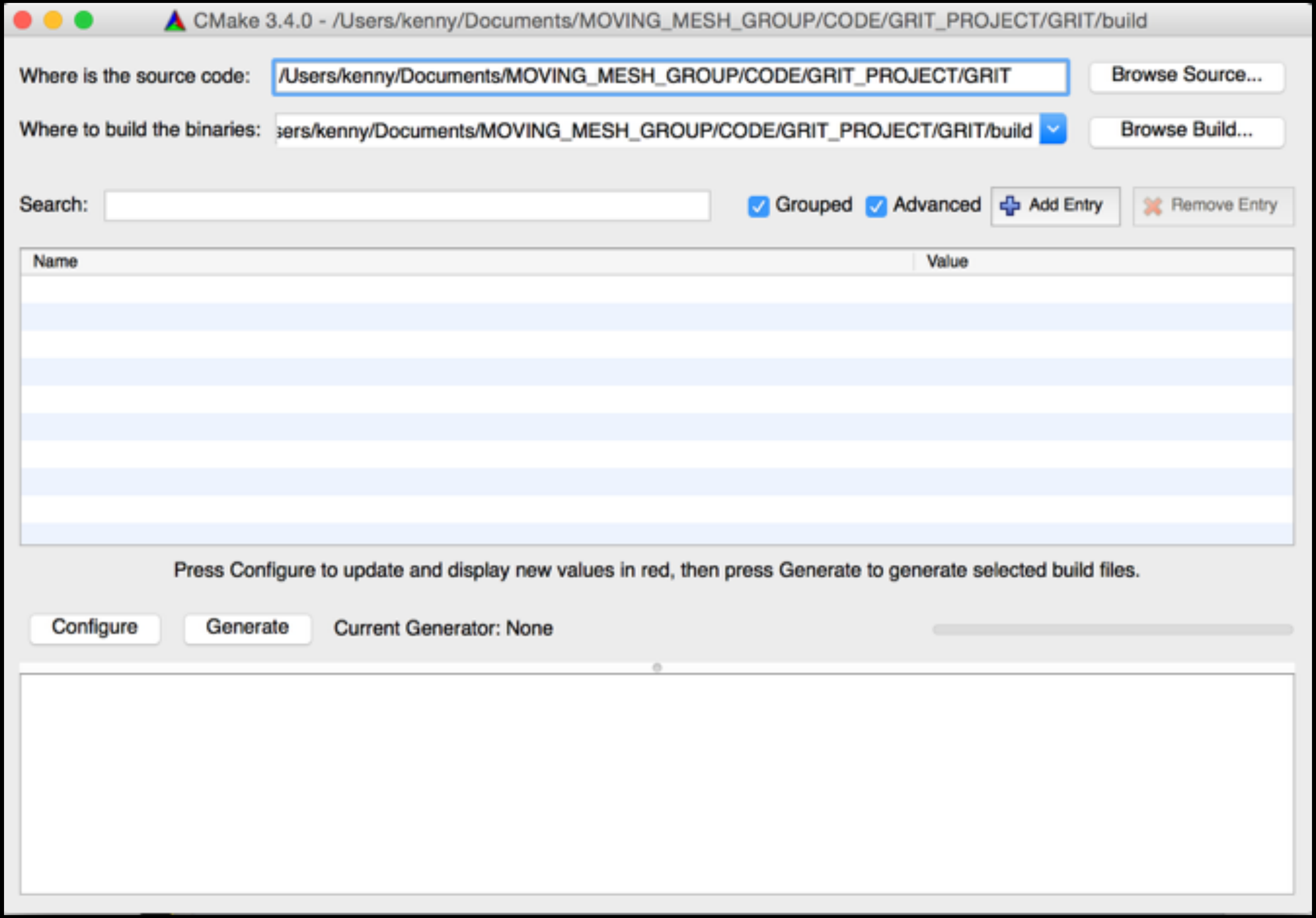
- DeVIL
    - http://openil.sourceforge.net
- NVIDIA CUDA TOOLKIT
    - https://developer.nvidia.com/cuda-toolkit
- Boost C++ Libraries
    - http://www.boost.org

STEP 4 — Run CMake

CMake Introduction

- Read about cmake here: http://www.cmake.org
- Observe: We use separate binary tree and source tree
- Remember to use ccmake if you run on command line or cmake-gui if not.
- Make sure all variables are found and set correctly before generating your make files.

CMake Step-by-Step Example



CMake 3.4.0 - /Users/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT/build

Where is the source code: /Users/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT

Where to build the binaries: sers/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT/build

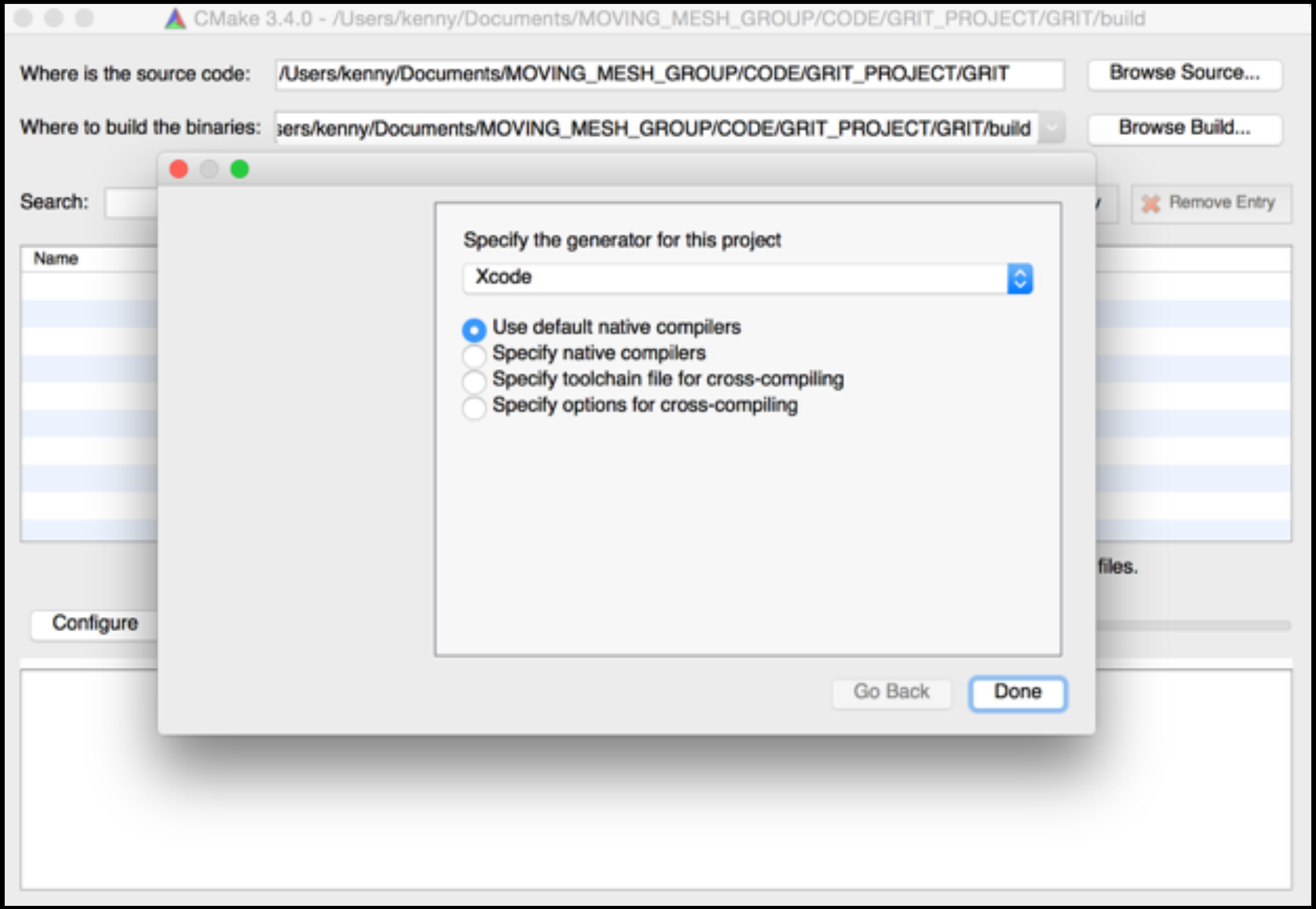
Search:

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: None

Specify folders

Observe source (where top-most CMakeLists.txt is) are different from the binary folder (that will contain your IDE project/make files)



CMake 3.4.0 - /Users/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT/build

Where is the source code: /Users/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT

Where to build the binaries: sers/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT/build

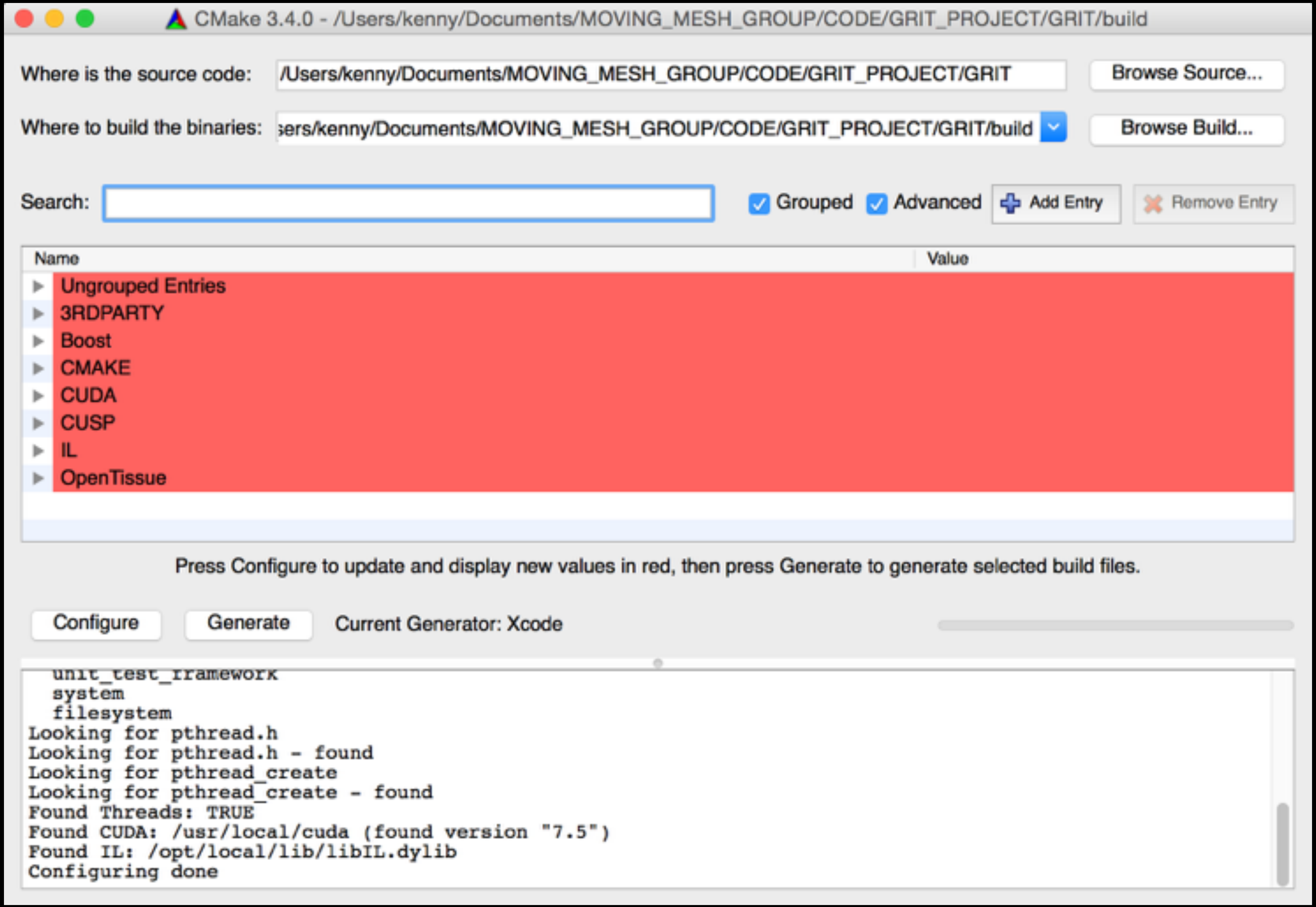
Specify the generator for this project

Xcode

[✓ Use default native compilers]
Specify native compilers
Specify toolchain file for cross-compiling
Specify options for cross-compiling

Click Configure and selected your generator

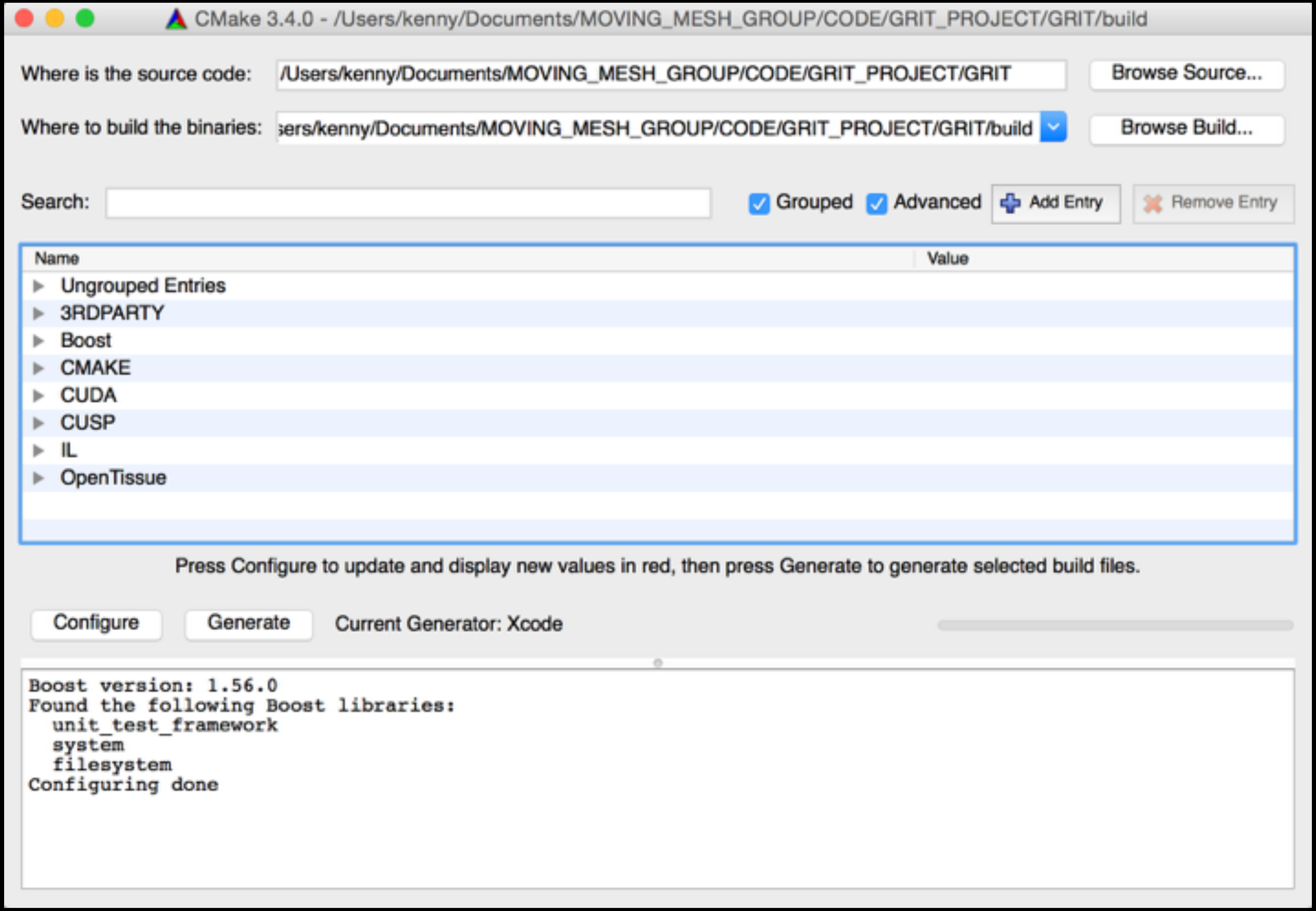
Im using macbook so in this example I picked xcode but you can pick anything from the drop down list


CMake 3.4.0 - /Users/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT/build
Where is the source code: /Users/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT
Browse Source...
Where to build the binaries: ɜers/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT/build
Browse Build...
Search:
Grouped
Advanced
Add Entry
Remove Entry
Name
Value
Ungrouped Entries
3RDPARTY
Boost
CMAKE
CUDA
CUSP
IL
OpenTissue
Press Configure to update and display new values in red, then press Generate to generate selected build files.
Configure
Generate
Current Generator: Xcode

```
  unit_test_framework
  system
  filesystem
Looking for pthread.h
Looking for pthread.h - found
Looking for pthread_create
Looking for pthread_create - found
Found Threads: TRUE
Found CUDA: /usr/local/cuda (found version "7.5")
Found IL: /opt/local/lib/libIL.dylib
Configuring done
```

Inspect the red stuff

CMake have now parsed all CMakeLists.txt files and inspected your system/computer. The red stuff is what it found in first iteration. Look stuff over to see if it is correct and fix the things you do not like.



Where is the source code: /Users/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT

Where to build the binaries: sers/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT/build

[✓ Grouped] [✓ Advanced]

- Ungrouped Entries
- 3RDPARTY
- Boost
- CMAKE
- CUDA
- CUSP
- IL
- OpenTissue

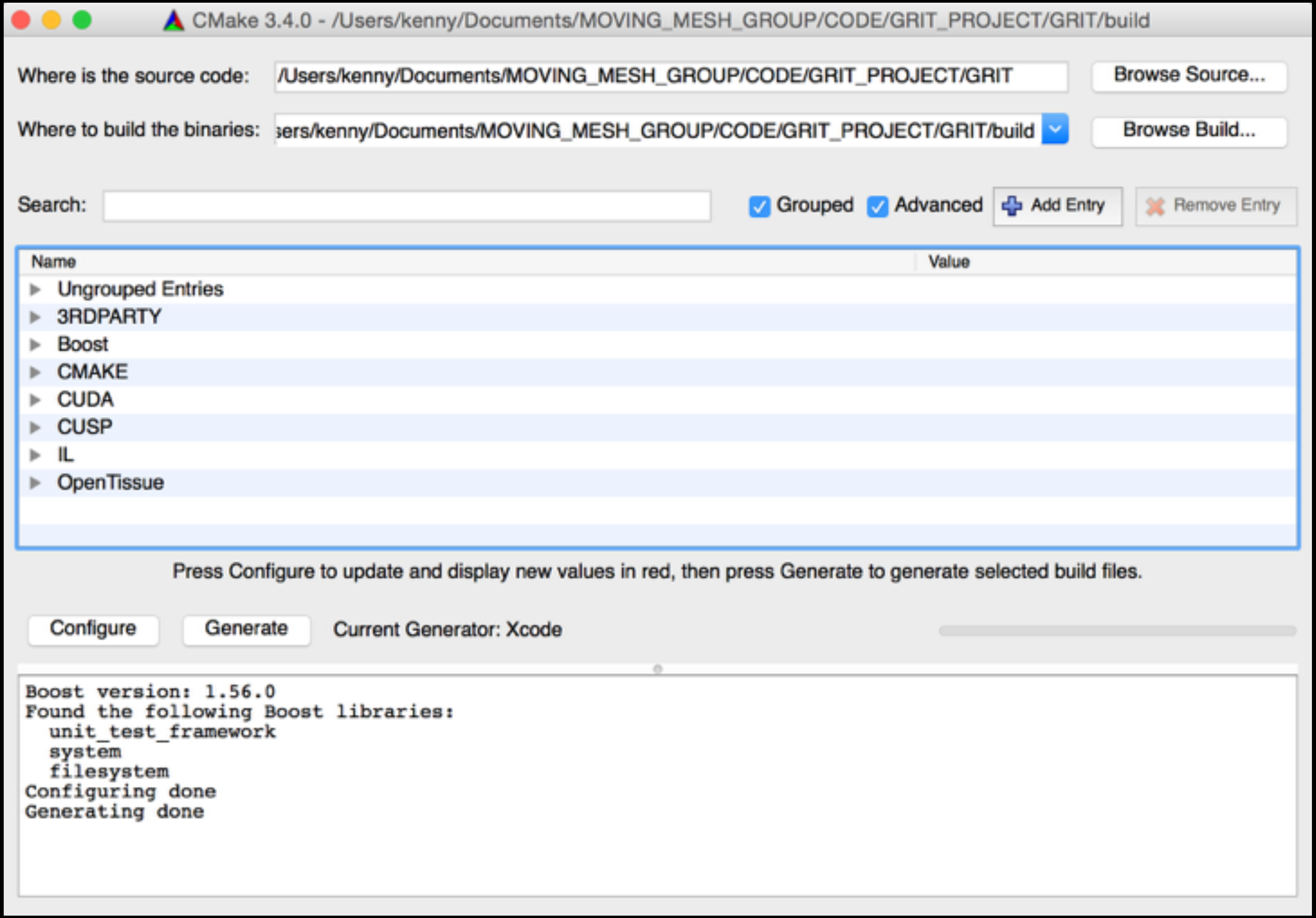
Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Xcode

```
Boost version: 1.56.0
Found the following Boost libraries:
  unit_test_framework
  system
  filesystem
Configuring done
```

Click Configure (2nd Time)

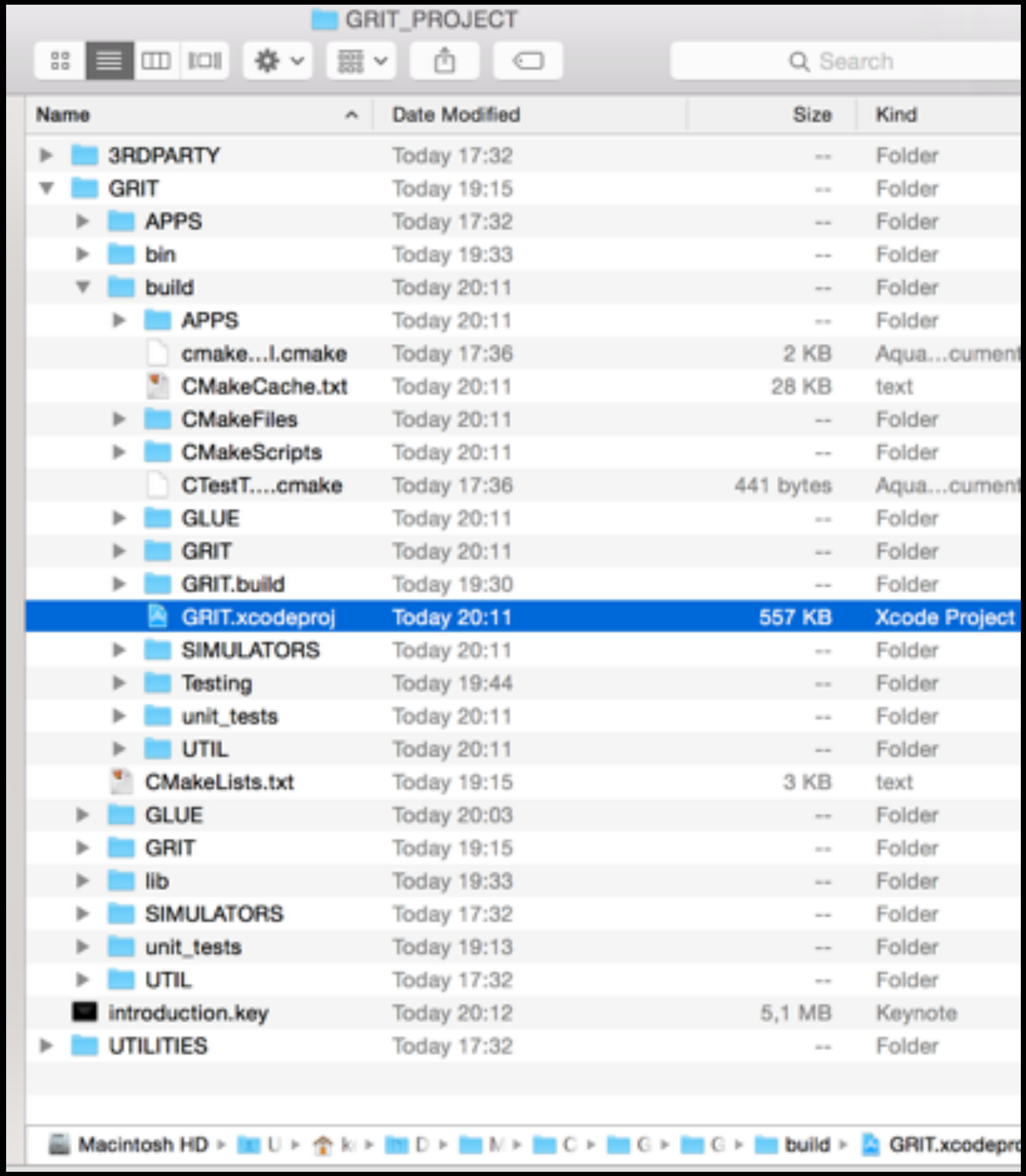
Now all red stuff should have gone unless you have a conflict or error that CMake do not know how to resolve for your system. If you still got red stuff at this point, then fix it and re-run configure until all red stuff has gone.


CMake 3.4.0 - /Users/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT/build
Where is the source code: /Users/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT
Browse Source...
Where to build the binaries: sers/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT/build
Browse Build...
Search:
Grouped
Advanced
Add Entry
Remove Entry
Name
Value
Ungrouped Entries
3RDPARTY
Boost
CMAKE
CUDA
CUSP
IL
OpenTissue
Press Configure to update and display new values in red, then press Generate to generate selected build files.
Configure
Generate
Current Generator: Xcode

```
Boost version: 1.56.0
Found the following Boost libraries:
  unit_test_framework
  system
  filesystem
Configuring done
Generating done
```

Press Generate

That’s it, now I got my xcode IDE project files.



GRIT_PROJECT

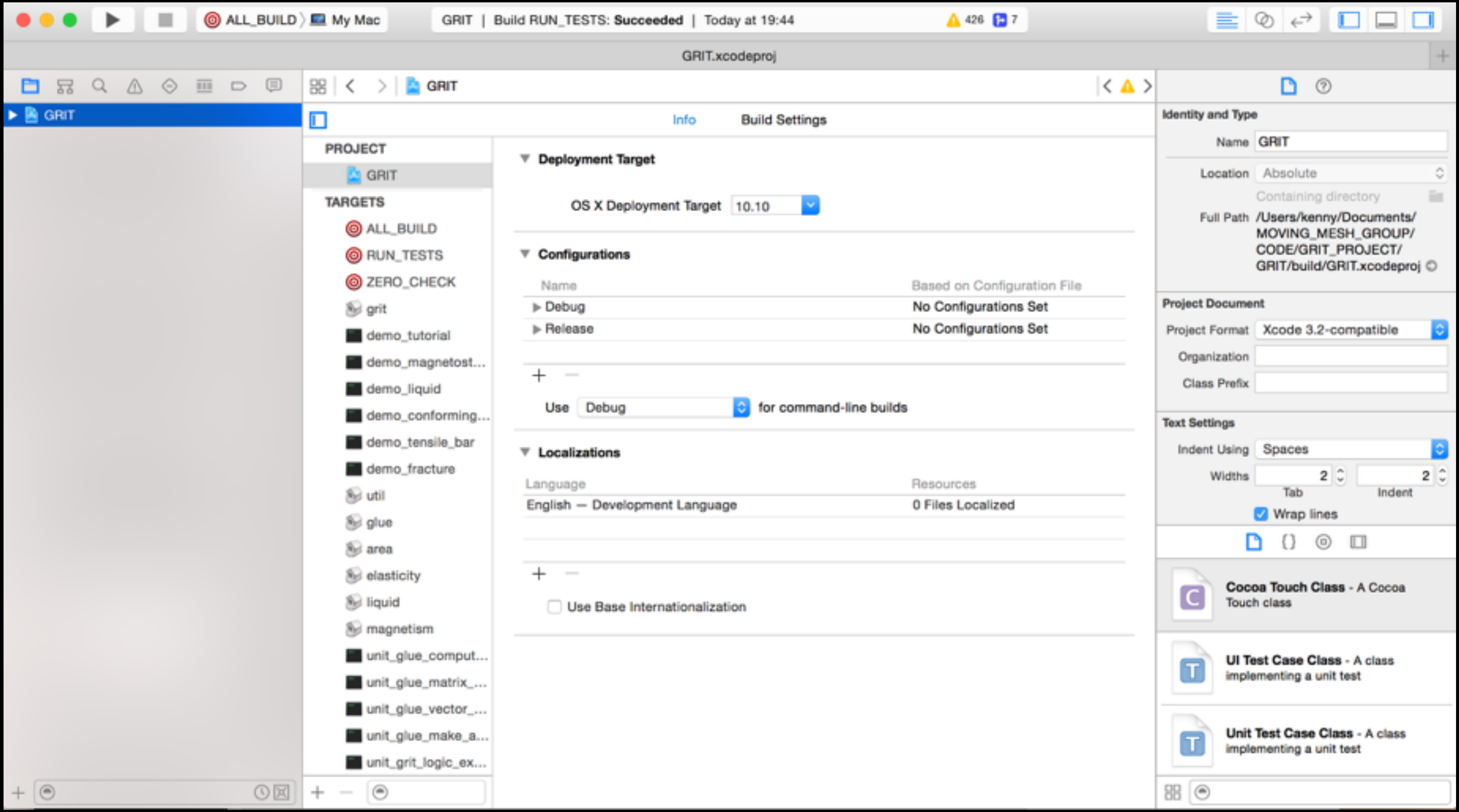
Name	Date Modified	Size	Kind
3RDPARTY	Today 17:32	--	Folder
GRIT	Today 19:15	--	Folder
APPS	Today 17:32	--	Folder
bin	Today 19:33	--	Folder
build	Today 20:11	--	Folder
APPS	Today 20:11	--	Folder
cmake...l.cmake	Today 17:36	2 KB	Aqua...cument
CMakeCache.txt	Today 20:11	28 KB	text
CMakeFiles	Today 20:11	--	Folder
CMakeScripts	Today 20:11	--	Folder
CTestT....cmake	Today 17:36	441 bytes	Aqua...cument
GLUE	Today 20:11	--	Folder
GRIT	Today 20:11	--	Folder
GRIT.build	Today 19:30	--	Folder
GRIT.xcodeproj	Today 20:11	557 KB	Xcode Project
SIMULATORS	Today 20:11	--	Folder
Testing	Today 19:44	--	Folder
unit_tests	Today 20:11	--	Folder
UTIL	Today 20:11	--	Folder
CMakeLists.txt	Today 19:15	3 KB	text
GLUE	Today 20:03	--	Folder
GRIT	Today 19:15	--	Folder
lib	Today 19:33	--	Folder
SIMULATORS	Today 17:32	--	Folder
unit_tests	Today 19:13	--	Folder
UTIL	Today 17:32	--	Folder
introduction.key	Today 20:12	5,1 MB	Keynote
UTILITIES	Today 17:32	--	Folder

Macintosh HD ▸ U ▸ k ▸ D ▸ M ▸ C ▸ G ▸ G ▸ build ▸ GRIT.xcodepr

There it is! Just open it

Now you are ready to code!

STEP 5 — Compile, link and run the code


ALL_BUILD 〉 My Mac

GRIT | Build RUN_TESTS: **Succeeded** | Today at 19:44

426 7

GRIT.xcodeproj

GRIT

Info Build Settings

PROJECT
- GRIT

TARGETS
- ALL_BUILD
- RUN_TESTS
- ZERO_CHECK
- grit
- demo_tutorial
- demo_magnetost...
- demo_liquid
- demo_conforming...
- demo_tensile_bar
- demo_fracture
- util
- glue
- area
- elasticity
- liquid
- magnetism
- unit_glue_comput...
- unit_glue_matrix_...
- unit_glue_vector_...
- unit_glue_make_a...
- unit_grit_logic_ex...

Deployment Target

OS X Deployment Target 10.10

Configurations

Name	Based on Configuration File
Debug	No Configurations Set
Release	No Configurations Set

Use Debug for command-line builds

Localizations

Language	Resources
English — Development Language	0 Files Localized

Use Base Internationalization

Identity and Type

Name	GRIT
Location	Absolute
	Containing directory
Full Path	/Users/kenny/Documents/MOVING_MESH_GROUP/CODE/GRIT_PROJECT/GRIT/build/GRIT.xcodeproj

Project Document

Project Format	Xcode 3.2-compatible
Organization	
Class Prefix	

Text Settings

Indent Using	Spaces	
Widths	2	2
	Tab	Indent

✓ Wrap lines

Cocoa Touch Class - A Cocoa Touch class

UI Test Case Class - A class implementing a unit test

Unit Test Case Class - A class implementing a unit test

This is how your IDE might look like

When first opening up the project you get a all into your face at once experience. Try to either build the "build all" target or select a specific demo_XXX target as your initial playground.

bin & lib

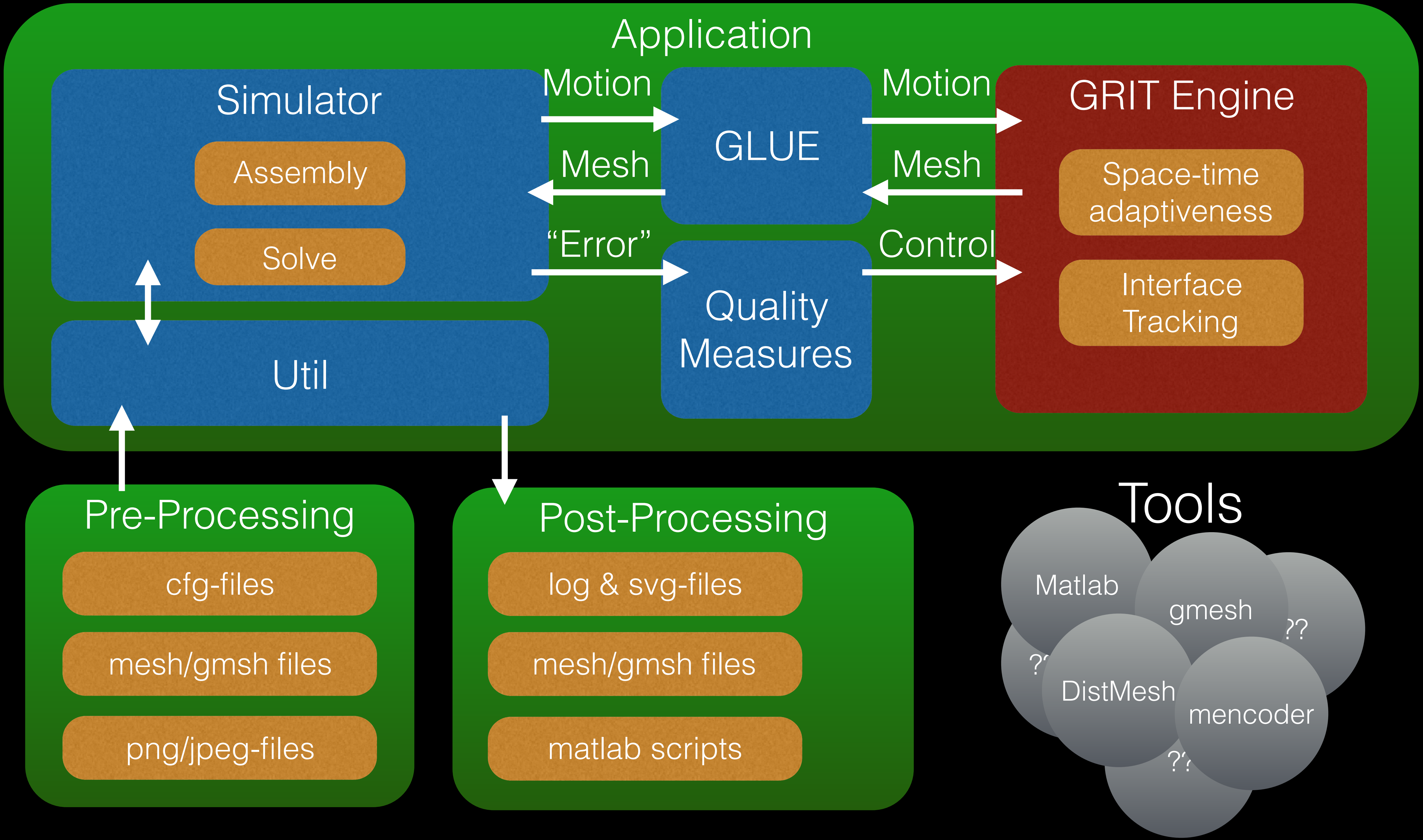
- Now just compile and link using your favorite compiler/IDE
- Remember that our cmake is setup such that binary executables are put into the bin-folder and that libraries are put into the lib folder
- Code is mostly header only with a few exceptions and all libraries build uses static linking.

Running an APP

- When running an APP there is no windows opening up. All code is command line tools, outputting to the prompt or using files for loading/saving etc..
- Look for corresponding cfg-file in bin folder to fine tune control of the APP you wish to work with.

STEP 6 — Understanding how an APP works

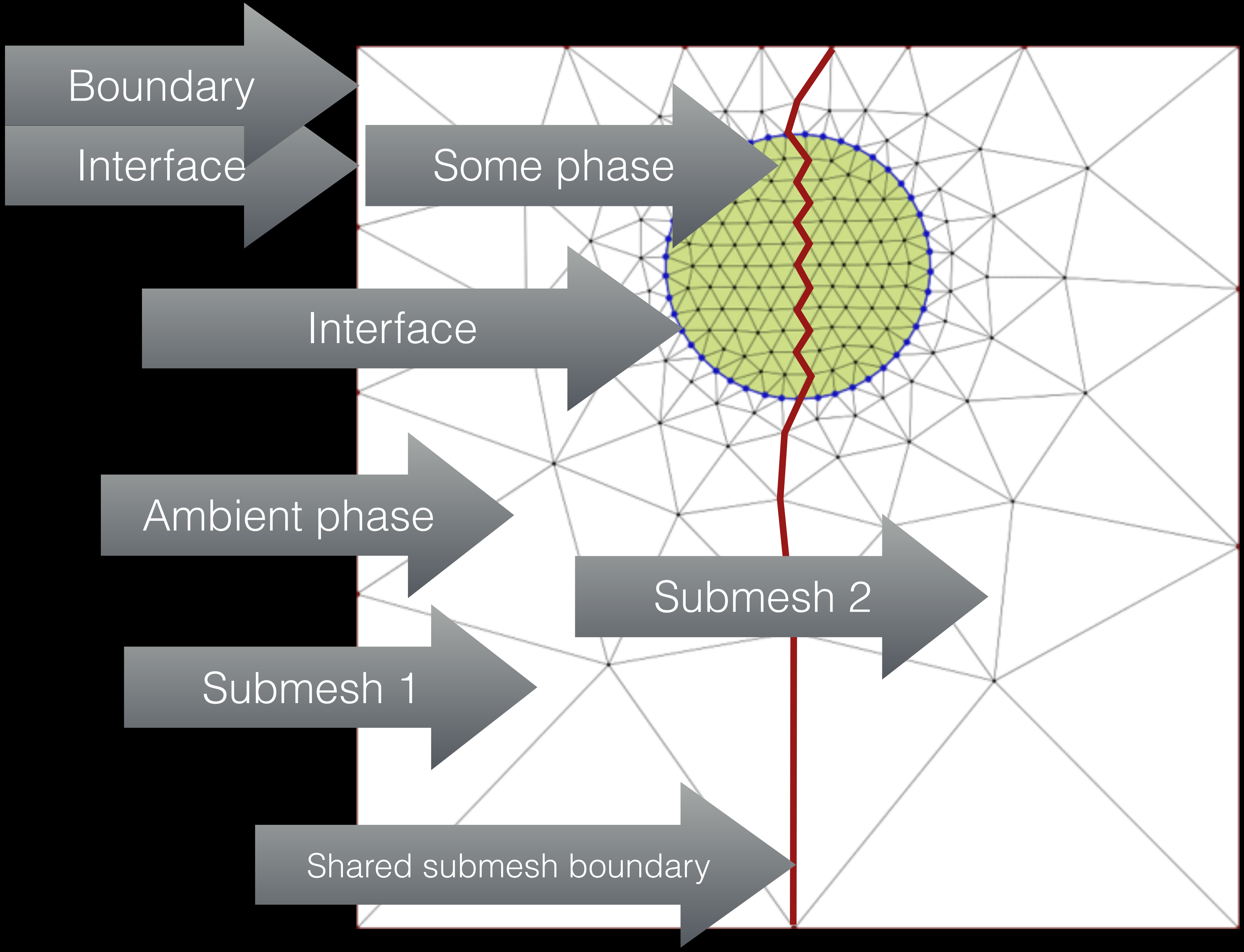
Overview


Application
Simulator
Assembly
Solve
Util
Motion
Mesh
"Error"
GLUE
Quality Measures
Motion
Mesh
Control
GRIT Engine
Space-time adaptiveness
Interface Tracking
Pre-Processing
cfg-files
mesh/gmsh files
png/jpeg-files
Post-Processing
log & svg-files
mesh/gmsh files
matlab scripts
Tools
Matlab
gmesh
??
??
DistMesh
mencoder
??

Terminology

- Domain is the whole mesh.
    - The boundary of the whole mesh is simply termed the “boundary”.
- Sub-mesh is a subset of the whole mesh and corresponds to a computational sub-domain.
    - The boundary of a sub-mesh is called a sub-mesh boundary. Subdomains have disjoint “faces” but shares vertices and edges on their boundaries with adjacent sub-domains.
    - Sub-domains are used to spatially decompose the whole mesh into smaller computational units of work. This allows for coarse grain parallelization.
- A phase is a subset of the whole mesh.
    - The boundary of a phase is called an interface
    - Phases can have shared interfaces but can never share a “face”.
    - The boundary is also an interface, but a sub mesh boundary is not an interface.
    - Phases can be decomposed into several sub-meshes

Terminology illustrated



STEP 7 — How to write your own first APP

The Tutorials

- The demo “tutorial” shows how to setup GRIT and how one can use GLUE to get and set mesh information. The demo demonstrates how simple kinematic/geometric deformations can be applied to the mesh engine
- The demo “tutorial_multiphase” shows how to control multiple phases with the GLUE library and how one can specify a simple sizing field for user specified control of refinement and coarsening

Code a Simple Demo

Details comes later first we just look at code structure

Include files and global instances

```
#include <grit.h>
#include <util.h>

grit::engine2d_type  engine;
grit::param_type     parameters;

util::ConfigFile      settings;
util::Log             logging;
```

Main

```
int main()
{
  settings.load("my_config_file.cfg")

  std::string  const txt_filename = settings.get_value("txt_filename");
  std::string  const output_path  = settings.get_value("output_path",    ""                  );

  util::LogInfo::on()            = util::to_value<bool>(settings.get_value("logging","true"));
  util::LogInfo::console()       = util::to_value<bool>(settings.get_value("console","true"));
  util::LogInfo::filename()      = output_path
                                 + "/"
                                 + settings.get_value("log_file","log.txt");

  logging << "### " << util::timestamp() << newline;

  parameters = grit::make_parameters_from_config_file(settings);

  assert(grit::is_valid(parameters));

  grit::init_engine_with_mesh_file(
                                      util::get_data_file_path(txt_filename)
                                      , parameters,engine
                                      );

  for (unsigned int i = 0; i < max_steps; ++i)
  {
    write_svg_files(output_path, i);

    do_simulation_step();

    engine.update(parameters);
 }

  logging << "Done" << newline;

  return 0;
}
```

Write SVG files

```
void write_svg_files(
                     std::string const & output_path
                     , unsigned int const & frame_number
                     )
{
  std::string const filename = output_path
                             + util::generate_filename(
                                                        "/my_own_file_name"
                                                       , frame_number
                                                       , "svg"
                                                       );

  glue::svg_draw(filename, engine, parameters);
}
```

Do Simulation Step

```
void do_simulation_step()
{
 unsigned int const object_label = util::to_value<unsigned int>(
                                          settings.get_value(“object_label","1")
                                          );

  glue::Phase  const object = glue::make_phase(engine, object_label);

  std::vector<double> px;
  std::vector<double> py;

  glue::get_sub_range_current(engine, object, px, py);

  std::vector<double> px_new = px;
  std::vector<double> py_new = py;

  for (unsigned int i = 0u; i != py_new.size(); ++i)
  {
    px_new[i] += 0.01;
    py_new[i] += 0.01;
  }

 glue::set_sub_range_target(engine, object, px_new, py_new);
}
```

How to create svg files

This utility feature is used in almost all demos

Why svg files?

- Writing svg files are convenient as these are ascii based vector graphics file. Hence, to make pretty pictures of how the mesh deforms during simulation on a server without a visual display is much more easier with svg files.

- One simply include the header file glue_svg_draw.h file and then at appropriate place in the code write

  - std::string filename = util::generate_filename("frame", i, "svg");

  - glue::svg_draw(filename, engine, parameters);

- Here “i” is the frame number being drawn. The util::generate_filename function is quite useful for adding a trailing number with padded zeros to the filename one desires to use. That is pretty much it. More details can be found in the header file glue_svg_draw.h

- Most internet browsers supports viewing of svg-files. Otherwise inkscape and GIMP are there good choices. See also STEP 8 for advice on how to convert svn-files to other formats.

How to use util::ConfigFile

This utility feature is used in almost all demos

Why getting parameters from cfg-files?

- There are many scenarios where this is beneficial. For instance, it is very convenient not having to re-compile and link an application when playing around with parameter values. Other times it is nice to store whole tests/experiments in single cfg-files, this way it is easier to repeat experiments later on. Application behaviour might also be changed from playing with parameter mode to run on a server mode.

- Hence we provide a utility tool for working with configuration files. So called cfg-files.

Example of cfg-syntax

- A cfg file may look like this
    - # I am a comment
    - on = true   #I am a comment too
    - list = item1 item2 item3 item4
    - filename = script.m
    - max_iterations = 100
    - stop_threshold = 0.01
- The syntax is simple and hopefully quite trivial. One can write any setting one wants into a config file. Our config utility will extract all variable names and associate their values with these names. The data is stored in a kind of runtime lookup table and can be queried at runtime.

Using util::ConfigFile

- One must include the util_config_file.h header file and then write
  - util::ConfigFile settings;
  - ...
  - settings.load("my_config_file.cfg");
  - bool on = util::to_value<bool>(settings.get_value(“on”,”true”));
  - …
- Observe that one simply invokes the load method with the path for the cfg-file that should be loaded. After this one can get values calling the get_value method. This method returns the value as a std::string. The first argument is the name of the variable, the second argument is the default value if no such variable exist in the cfg-file. Notice that above we use the util::to_value functionality to convert the string-value into the actual correct type that we wish to have.

Using multi-valued names

- Recall the syntax “list = item1 item2 item3”. To obtain all values one must write

  - std::vector<std::string> values = settings.get_values(“list”);

- If one only wrote settings.get_value(“list”) one would just get “item1” as the result. The above “values” vector contains all three string values “item1”, “item2” and “item3”.

How to use util::Log

This utility feature is used in almost all demos

Why not using std::cout?

- Depending on when you run your application you may wish to see messages on the console or store them into a file or neither of those behaviours. To make logging behaviour controllable by users of the application we provide the utility log feature. One must include the header file util_log.h
    - util::ConfigFile settings;
    - util::Log   logging;
    - std::string const newline = util::Log::newline();
    - ...
    - settings.load("my_config_file.cfg");
    - util::LogInfo::on()   = util::to_value<bool>(settings.get_value("logging"));
    - util::LogInfo::console() = util::to_value<bool>(settings.get_value("console"));
    - util::LogInfo::filename() = settings.get_value("log_file");
    - logging << "hello" << newline;
- Here we use a configuration file to obtain end-users decided logging behaviour. The behaviour is controlled by setting “on”, “console” and “filename” properties on the LogInfo class. Here after one can use a util::Log instance in much the same way as one would use std::cout or std::cerr streams.

Sprinkles on top

- The utility library contains other features that are nice in combination with logging. For instance util_timestamp.h contains a convenience tool to mark ones log with the time of the day
    - logging << util::timestamp() << newline;
- The util_string_helpers.h is quite useful too. For instance the util::to_value and util::to_string functions are convenient.

How to use util::Profiling

See for instance elasticity or liquid demos for example code

Easy profiling using macros

- The include header util_profiling.h contains utilities for profiling your code. For instance during a simulation loop one wants to time a certain computation or monitor say the kinetic energy, then one writes
- #include <util_profiling.h>
- while(true)
- {
    - START_TIMER(“MY_TIMER”);
    - …. do the computation….
    - STOP_TIMER(“MY_TIMER”);
    - RECORD(“ENERGY”, my_value);
- }
- Notice that all that is needed are the macros START_TIMER, STOP_TIMER and RECORD

Writing Profiling Data

- Once simulation is over one would want to write the profiled data. This is done by obtaining the corresponding monitor and getting the values stored during simulation. Here is the outline of the code for doing this:
  - std::ofstream file;
  - ……
  - Profiling::Monitor * E = Profiling::get_monitor("MY_ENERGY");
  - file << "E = " << util::matlab_write_vector(E->get_values()) << ";" << std::endl;
  - Profiling::TimerMonitor * T = Profiling::get_timer_monitor("MY_TIMER");
  - file << "T = " << util::matlab_write_vector(T->get_values()) << ";" << std::endl;
- Notice that string values are used to uniquely identify both Monitors and TimerMonitors. The application programmer can name these as he/she pleases. In the example code above we use the matlab write vector utility function from the header file util_matlab_write_vector.h file

How to implement a matrix assembly

For example code see magnetostatic or liquid demos

Typical FEM discretization

- Let us work through an example to make things more clear. Consider the PDE

$$\nabla^2 \vec{u} = 0, \quad \nabla = \begin{bmatrix} \partial_x \\ \partial_y \end{bmatrix}, \quad \text{and} \quad \vec{u} \in \Re^2$$

- Doing FEM gives the counter part discrete version for triangle t with nodes i,j,k as

$$\underbrace{\begin{bmatrix} E_{ii} & E_{ij} & E_{ik} \\ E_{ji} & E_{jj} & E_{jk} \\ E_{ki} & E_{kj} & E_{kk} \end{bmatrix}}_{E_t} \begin{bmatrix} u_{x,i} \\ u_{y,i} \\ u_{x,j} \\ u_{y,j} \\ u_{x,k} \\ u_{y,k} \end{bmatrix} = \vec{0}$$

Initialize Space

- First we need to get all triangles
    - glue::Phase omega = glue::make_phase(…);
- Next we must allocate space for the discrete FEM element matrix array
    - unsigned int T = omega.m_triangles.size();
    - std::vector<util::Block3x3Tensor2> Es;
    - Es.resize(T);

Compute Element Matrix Array

- Next we compute the element matrices and fill them into the array
- for(unsigned int t = 0; t < T; ++t)
- {
    - Es[t].m_blocks[0][0] = … compute E_ii….;
    - Es[t].m_blocks[0][1] = … compute E_ij….
    - …
    - Es[t].m_blocks[2][2] = … compute E_kk….
- }

Do Matrix Assembly

- For getting easy started a matrix data type is provided in util_coo_matrix.h header file and for working with GLUE the header file glue_matrix_assembly.h makes it easy to transform a element matrix array into a global matrix
  - bool interlaced = true;
  - util::COOMatrix<double> A = glue::matrix_assembly<double>(omega, Es,interlaced)
- The last argument controls how the memory layout is generated. That is how x and y coordinates are mapped to global matrix indices.

How to design a flexible simulation loop

For example code see liquid or elasticity demos

Desiderata for a Typical Simulation Loop

- One wants to have
  - control of the total simulated time, T>0, to run
  - control of the smallest and largest possible time step,dt_min and dt_max. Observe 0 < dt_min << dt_max < T. Setting these are convenient for ensuring upper and lower bounds on the computing time.
  - control of the frame rate, fps, of any images generated to be able to produce movie playback running in simulated time. Observe that T > dt_max >= 1/fps >= dt_min.
- Hence, T, dt_min, dt_max and fps are user input parameters to be set/read from for instance a config file.

The Simulation Loop

```
Algorithm simulation loop(T,dt_min,dt_max,fps)
    T_left = T
    while T_left > 0 do
        dt_wanted = 1 / fps
        dt_left = dt_wanted
        while dt_left > 0 do
            dt_adaptive = min( dt_left,
              compute_step_size( dt_left ));
            dt = min(dt_max,max(dt_min, dt_adaptive)
            compute_time_step( dt )
            dt_left = dt_left - dt
        end
        draw frame
        T_left = T_left - dt_wanted
    end
```

- Notes
  - compute_step_size
    - This is a simulation specific function that will try and estimate and adaptive time step size. For instance using a CFL condition or some time integration error measure to reduce/enlarge the step size
  - compute_time_step
    - This is a simulation specific function that will advance the state of the simulation system with the specified time step.

How to make matlab output

Tips for working with Matlab with out too much trouble

- When generating matlab scripts from your GRIT applications then it may be worthwhile to remember the following tips and tricks:
    - Add matlab to your path environment so you can run matlab from command line. On OSX one adds to the file ".profile"
        - export PATH=/Applications/MATLAB_R2014b.app/bin:$PATH
    - Now on the command line one can now write
        - matlab -nodesktop
    - When generating matlab scripts for making plots then it is convenient to write
        - figure(1, 'Visible', 'off')
    - This way when running a script from the command line figures will not pop up and disturb one.
    - Split your generated matlab script into several files and the generate one script that runs all other scripts. Go the the folder where you generated your matlab scripts then write on the bash command line
        - for f in *.m; do y=${f%.m}; echo ${y##*/}; done >> run_all.m
    - Now in the no desktop command line version of matlab write run_all to start processing all matlab script files as one "background" batch job.

Easy matlab output

- To make it easier to get started writing code for generating a matlab script we have provided several utility functions to help
  - util::write_matlab_vector
  - util::write_matlab_matrix
  - util::to_string(util::COOMatrix<T> …)
  - util::generate_filename(…)
- To help generate stript data for meshes GLUE provides
  - glue::matlab_write_mesh(…)

Example (1/2)

- Here we show code to generate a matlab script for drawing the mesh. First you write data as matlab arrays, like this
  - std::ofstream script;
  - ....
  - script << "close all;" << std::endl;
  - script << "clear all;" << std::endl;
  - script << "clc;" << std::endl;
  - script << "T_"   << count << " = " << matlab_write_mesh(triangles)   << ";" << std::endl;
  - script << "px_" << count << " = " << util::matlab_write_vector(px) << ";" << std::endl;
  - script << "py_" << count << " = " << util::matlab_write_vector(py) << ";" << std::endl;
- Here “count” is supposed to be a integer variable that counts the number of frames that you have generated to far.

Example (2/2)

- Next you must generate script code for making a matlab figure. That could look like this
  - script << "figure('Visible','off');" << std::endl;
  - script << "clf;" << std::endl;
  - script << "triplot(T_" << count << ",px_" << count << ",py_" << count << ",'b');" << std::endl;
  - script << "xlabel('x');" << std::endl;
  - script << "ylabel('y');" << std::endl;
  - script << "title('Computational Mesh');" << std::endl;
  - script << "print(gcf,'-depsc2','" << "mesh_" << std::setw(8) << std::setfill('0') << count << "');" << std::endl;
- More advanced matlab scripts can be observed in various demos.

How to use glue::set_sub_range and glue::get_sub_range

```
void do_simulation_step()
{
  double       const move_x            = 0.01;
  double       const move_y            = 0.01;
  unsigned int const my_object_label   = 1u;

  glue::Phase  const my_phase          = glue::make_phase(engine, my_object_label);

  std::vector<double> px;
  std::vector<double> py;

  glue::get_sub_range_current(engine, my_phase, px, py);

  std::vector<double> px_new (px.begin(), px.end());
  std::vector<double> py_new (py.begin(), py.end());

  for (unsigned int i = 0u; i != py_new.size(); ++i)
  {
    px_new[i] += move_x;
    py_new[i] += move_y;
  }

  glue::set_sub_range_target(engine, my_phase, px_new, py_new);
}
```

Step 1: Get a phase object

Step 2: Get current coordinates

Step 3: Set target coordinates for phase

Current and target attributes are in-built named attributes… Target attributes must be set for all vertices in phase. If no target attributes are specified then it is implicitly assumed target is equal to current attribute values.

Step 1:Create and clear a user custom field named phi

One can have custom attributes for vertices (0u), edges (1u) and faces (2u).

```
inline void setup_fields( grit::engine2d_type &  engine )
{
  engine.attributes().create_attribute( "phi",   2u );

  glue::clear_attribute( engine, "phi",  0.0, glue::FACE_ATTRIBUTE() );
}
```

```
inline void do_simulation_step(
                                grit::engine2d_type          & engine
                                , util::ConfigFile      const & settings
                                )
{
  std::vector<double> phi;    // Resulting potential

  glue::Phase const domain = glue::make_phase(engine);

  glue::get_sub_range(engine, domain, "phi",   phi,  glue::FACE_ATTRIBUTE()   );

  //… Compute a new phi field value …

  glue::set_sub_range(engine, domain, "phi",  phi,  glue::FACE_ATTRIBUTE()   );
}
```

Step 2: Get the field values

Step 3: Set the field values

Use glue::VERTEX_ATTRIBUTE, glue::EDGE_ATTRIBUTE or glue::FACE_ATTRIBUTE to tell GLUE what kind of attributes you are working on in glue::clear_attribute, glue::copy_attribute, and glue::set_sub_range & glue::get_sub_range.

How to use and setup a custom sizing field

```
inline void setup_fields( grit::engine2d_type &  engine, grit::param_type    & parameters)
{
  engine.attributes().create_attribute( "lower_field", 1u);
  engine.attributes().create_attribute( "upper_field", 1u);

  glue::clear_attribute( engine, "lower_field", lower_value, glue::EDGE_ATTRIBUTE() );
  glue::clear_attribute( engine, "upper_field", upper_value, glue::EDGE_ATTRIBUTE() );

  parameters.set_lower_threshold_attribute( "refinement",           “lower_field");
  parameters.set_lower_threshold_attribute( "interface_refinement", “lower_field");

  parameters.set_upper_threshold_attribute( "coarsening",           “upper_field");
  parameters.set_upper_threshold_attribute( "interface_coarsening", “upper_field");
}
```

Step 1: Connect custom edge fields to threshold limits on named operations

```
inline void do_simulation_step(
                               grit::engine2d_type          & engine
                               , util::ConfigFile      const & settings
                               )
{
  std::vector<double> L;
  std::vector<double> U;

  glue::Phase const my_area = glue::make_phase(engine,….);

  glue::get_sub_range(engine, my_area, “lower_field", L,  glue::EDGE_ATTRIBUTE()
  glue::get_sub_range(engine, my_area, “upper_field", U,  glue::EDGE_ATTRIBUTE()

  //… Update L and U with new values

  glue::set_sub_range(engine, my_area, “lower_field",  L,  glue::EDGE_ATTRIBUTE()
  glue::set_sub_range(engine, my_area, “upper_field",  U,  glue::EDGE_ATTRIBUTE()
}
```

Step 2: Set the field values using glue::get_sub_range & glue::set_sub_range. GRIT takes care of everything else.

If sizing fields are not specified for all attributes GRIT is just going to default back to the default value set with glue::clear_attribute or if glue::glue_attribute is omitted then the lower/upper threshold value that is read from the cfg files are used as default value.

How to set and control GRIT parameters in cfg files

A typical cfg-file example

Step 1: First specify what operations to use

Step 2: Create some namespaces from other cfg-files

Step 3: Assign values from namespaces to operations when working on specific phases

Step 4: Override assigned values to customise behaviour

```
operations = vertex_split move merge coarsening interface_coarsening refinement interface_refinement smoothing interface_smoothing optimization

import params_vertex_split         = remeshing_params/vertex_split.cfg
import params_move                 = remeshing_params/move.cfg
import params_merge                = remeshing_params/merge.cfg
import params_coarsening           = remeshing_params/coarsening.cfg
import params_interface_coarsening = remeshing_params/interface_coarsening.cfg
import params_refinement           = remeshing_params/refinement.cfg
import params_interface_refinement = remeshing_params/interface_refinement.cfg
import params_smoothing            = remeshing_params/smoothing.cfg        # Laplacian smoothing of non-interface vertices
import params_interface_smoothing  = remeshing_params/interface_smoothing.cfg
import params_optimization         = remeshing_params/optimization.cfg     # edge flip optimization

#syntax: assign = operation_nameX label_valueY scope_name;

assign = vertex_split         0 vertex_split
assign = vertex_split         1 vertex_split

assign = move                 1 params_move

assign = merge                0 params_merge
assign = merge                1 params_merge

assign = coarsening           0 params_coarsening
assign = coarsening           1 params_coarsening

assign = interface_coarsening 0 params_interface_coarsening
assign = interface_coarsening 1 params_interface_coarsening

assign = refinement           0 params_refinement
assign = refinement           1 params_refinement

assign = interface_refinement 0 params_interface_refinement
assign = interface_refinement 1 params_interface_refinement

assign = smoothing            0 params_smoothing
assign = smoothing            1 params_smoothing

assign = interface_smoothing  0 params_interface_smoothing
assign = interface_smoothing  1 params_interface_smoothing

assign = optimization         0 params_optimization
assign = optimization         1 params_optimization

#syntax: override = operation_nameX label_valueY parameter_nameX parameter_valueY

override = interface_refinement 0 max_iterations 0        # Turn off interface refinement for ambient space
override = interface_coarsening 0 max_iterations 0        # Turn off interface coarsening for ambient space

override = interface_refinement 1 lower_threshold  0.04
override = interface_coarsening 1 upper_threshold  0.001

override = refinement           1 lower_threshold  0.05
override = coarsening           1 upper_threshold  0.001

override = refinement           0 lower_threshold  0.08
override = coarsening           0 upper_threshold  0.02

override = merge                0 angle_threshold  175.0
override = merge                1 angle_threshold  175.0

override = move                 0 strength  0.99
override = move                 1 strength  0.99
```

Step 1 — Specify Operations

- One use the command syntax
  - operations = {operation_name}
  - operation_name = vertex_split | move | merge coarsening | interface_coarsening | refinement | interface_refinement | smoothing | interface_smoothing | optimization
- The operation names are hard-wired into GRIT as keywords and only those names are allowed.
- If a name is omitted then it means that GRIT will not perform this type of operation. It is effectively turned of.
- The names appear unordered and their order has nothing to do with the order that GRIT will perform the corresponding operation batches in.

Step 2 — Create Namespaces

- To make it easier to quickly setup parameters for operations we have created cfg-files for each operation that contains default values. The default values can be loaded into a named scope by writing
    - import scope_name = “path to cfg file”
- The scope_name can in principle be any name you decide. There are no rules for what you can call a scope. You may think of a scope like a read-only kind of record/struct.
- GRIT comes with default cfg scope-files for all operation types. One can explore the parameters inside these scope files for learning how to tweak and tune each operation type.

Step 3 — Assign Scopes to Operations

- Scopes do not do anything, one must assign their values to a given operation for a given phase (ie. label). The syntax is as follows

  - assign = operation_name label scope_name

- Notice that the scope is not merely assigned to an operation, but it is assigned to the pairing of an operation and a label. This is because operations can behave differently depending on what phase in the mesh they are invoked on.

- As an example one often turn of mesh optimisation in the ambient phase of the mesh, and have high quality optimisation on the phase representing the object of interest.

Step 4 — Override default values

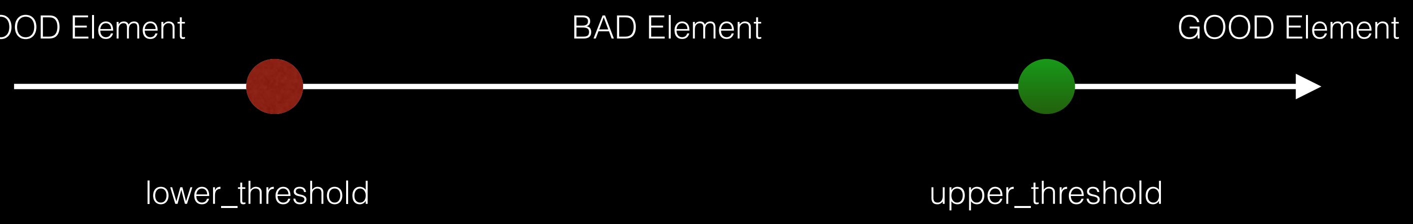
- The syntax for overriding the default scope assigned parameter values is as follows
  - override = operation_name label parameter_name value
- Here parameter_name is a parameter from the scope file and value is the new value that should be assigned to the given operation when working on the mesh phase with the specified label.
- As an example consider this
  - override = interface_refinement 0 max_iterations 0
- Here we tell GRIT to turn off interface refinement on phase with label=0. This is done by setting the scope parameter name “max_iterations” to the new value 0. Hence GRIT will perform 0 iterations of the interface_refinement for phase 0.

More Override Examples

- Often one wish to override refinement and coarsening thresholds to control the element sizes in the mesh. This may look as follows
    - override = refinement   1 lower_threshold 0.05
    - override = coarsening   1 upper_threshold 0.001
- Notice that this specify refinement and coarsening for phase with label = 1. The parameter names lower_threshold and upper_threshold requires some explanation.
- Operations using lower/upper threshold values are based on what GRIT calls a threshold quality measure. It means that if the current quality, q, of a given mesh element (edges for coarsening and refinement) are such that
    - q < lower_threshold   then we have a good mesh element and do nothing
    - q > upper_threshold then we have a good mesh element and do nothing
    - lower_threshold <= q <= upper_threshold then we have a bad mesh element and perform the operation
- For refinement upper_threshold is always set to infinity, and for coarsening lower_threshold is set to -infinity. For refinement and coarsening it is important that refinement lower_threshold is sufficiently larger than coarsening upper_threshold. As a rule of thumb make refinement lower_threshold > 2 coarsening upper_threshold. This is not guaranteed to work but usually prevents refinement and coarsening operations to counter act each other.

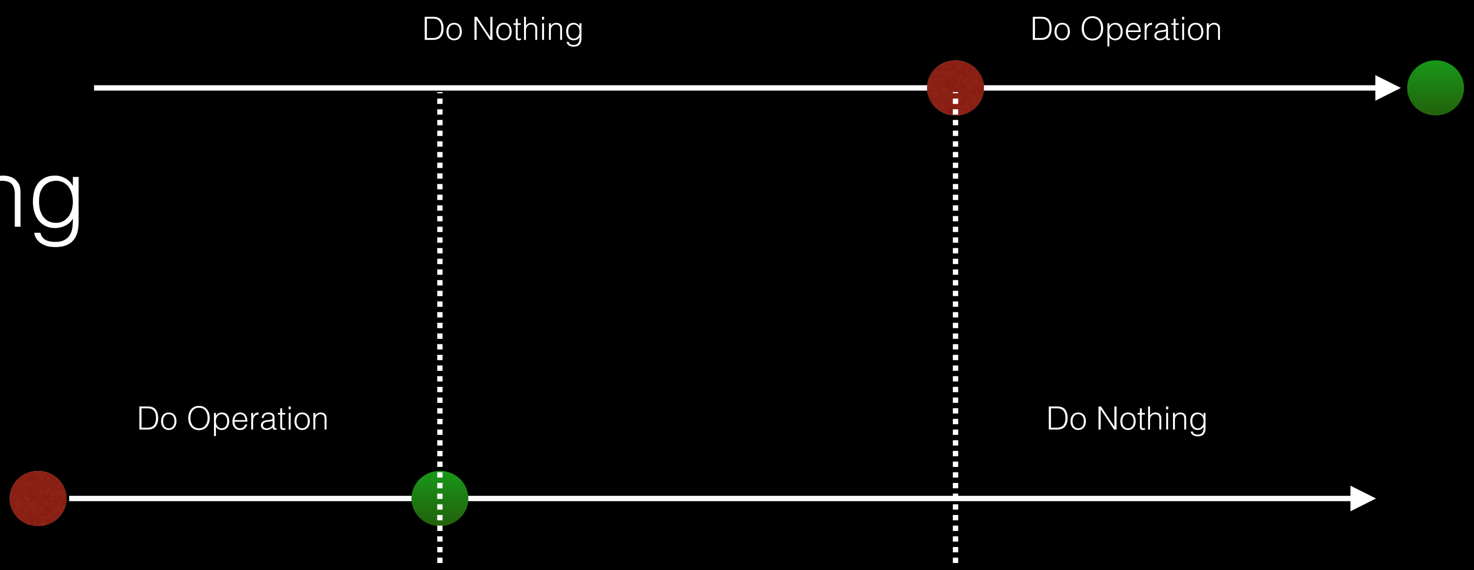
Threshold Summary

Threshold Quality Measure



Refinement

Coarsening



STEP 8 — How to make a movie on mac OS X

Install ImageMagick and Movie player

- sudo port install ImageMagick

- sudo port install mplayer

Convert to png and run mencoder

- mogrify   -format png   *.svg

- mencoder mf://*.png -mf w=800:h=800:fps=25:type=png -ovc lavc -lavcopts vcodec=mpeg4:mbd=2:trell -oac copy -o output.avi

A double pass divx encoding

- Here is a bash script for it
  - opt=“vbitrate=2160000:mbd=2:keyint=132:v4mv:vqmin=3:lumi_mask=0.07:dark_mask=0.2:mpeg_quant:scplx_mask=0.1:tcplx_mask=0.1:naq"
  - mencoder -ovc lavc -lavcopts vcodec=mpeg4:vpass=1:$opt -mf type=png:fps=25 -nosound -o /dev/null mf://*.png
  - mencoder -ovc lavc -lavcopts vcodec=mpeg4:vpass=2:$opt -mf type=png:fps=25 -nosound -o output.avi mf://*.png

Merging movies

- Here is example using mplayer
  - mencoder -oac copy -ovc copy video1.avi video2.avi -o final.avi
- Here is another command line tool
  - avimerge -o final.avi -i vidoe1.avi video2.avi

Bash Scripts for Movies

- Look in folder UTILITIES for useful bash scripts
- Copy and use scripts as boiler plate scripts for making your own custom solution

Enjoy