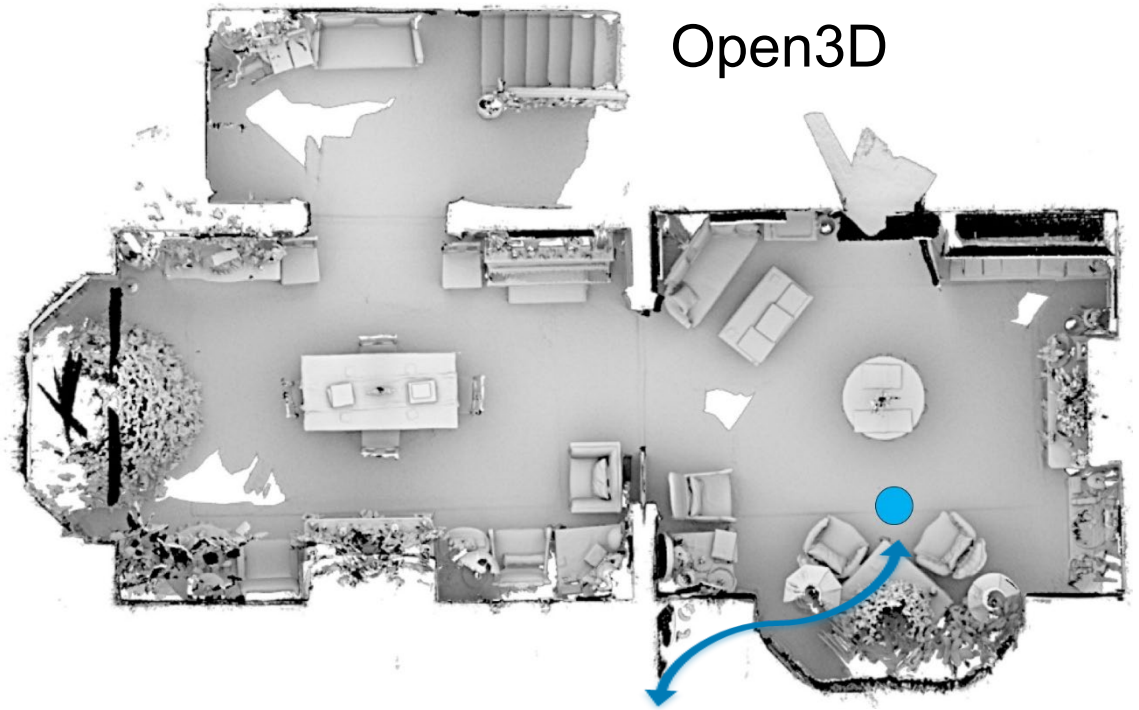
Robotic Mapping & Localization

Kaveh Fathian
Assistant Professor
Computer Science Department
Colorado School of Mines

Lab03: Useful C++ Libraries for SLAM;
Git *Courtesy of many people!

Lecture Outline

- **Useful C++ Libraries for SLAM/Robotics**
  - Eigen
  - OpenCV
  - PCL
  - Open3D
  - Other libraries
- **Intro to Git**



Eigen

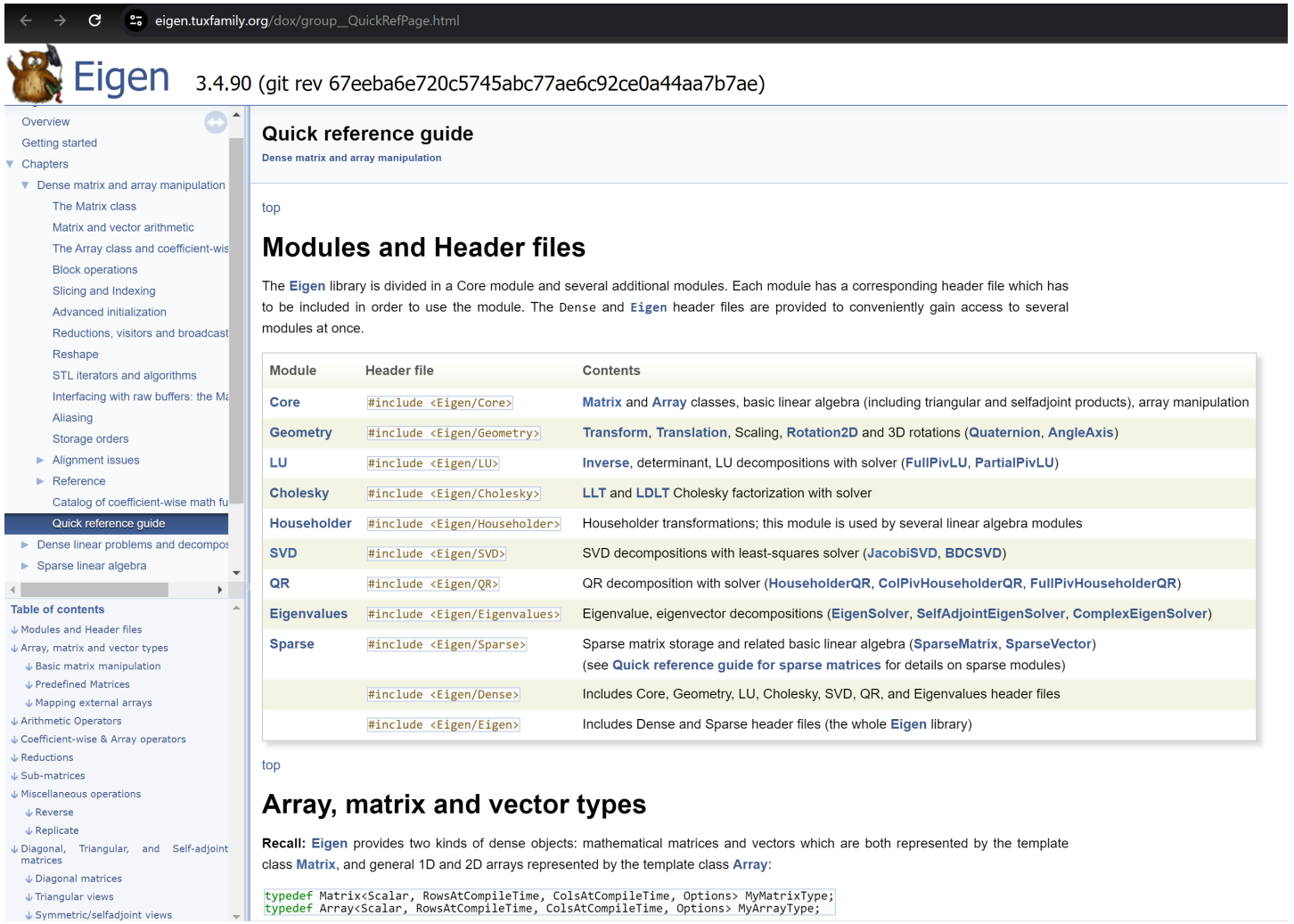
- Eigen is a C++ templated library for linear algebra (matrices, vectors, algorithms)
- Is header-only, meaning that it doesn't require compilation or linking
- Widely used in scientific computing, engineering, computer graphics, machine learning, robotics/SLAM

Eigen Library

```
18 #include <iostream>
19 #include <Eigen/Dense>
20 using namespace std;
21 using namespace Eigen;
22 int main()
23 {
24     Matrix3f A;
25     Vector3f b;
26     A << 1,2,3,  4,5,6,  7,8,10;
27     b << 3, 3, 4;
28     cout << "Here is the matrix A:\n" << A << endl;
29     cout << "Here is the vector b:\n" << b << endl;
30     Vector3f x = A.colPivHouseholderQr().solve(b);
31     cout << "The solution is:\n" << x << endl;
32 }
```

Eigen

- Eigen's documentation is a great resource – use it!
  https://eigen.tuxfamily.org/



Eigen

- Introduction slides from
  https://people.engr.tamu.edu/sueda/courses/CSCE489/2020F/misc/eigen.html

CSCE 489 – Eigen

people.engr.tamu.edu/sueda/courses/CSCE489/2020F/misc/eigen.html

Home

Eigen Intro

Matrix and Vector sizes

All matrix and vector classes are subclasses of the `Matrix` class. There are also built-in types for commonly used matrix and vector types.

```
Eigen::MatrixXd A(2,4); // 2x4 double matrix
Eigen::Matrix4d B;      // 4x4 double matrix
Eigen::Matrix3f C;      // 3x3 float matrix
Eigen::VectorXf d(9);   // 9x1 float vector
Eigen::Vector4f e;      // 4x1 float vector
```

If you declare `using namespace Eigen;` beforehand in your code, you don't need to say `Eigen::` in front of these and other Eigen types. If you do this, it is highly recommended that you do this in each `.cpp` file and not in a `.h` file, where it could potentially pollute multiple other files.

Initializing a matrix

Use the `<<` and `,` operators:

```
MatrixXd A(2,4); // 2x4 matrix
A << 1, 2, 3, 4,
     5, 6, 7, 8;
```

The elements are specified row by row.

You can also initialize matrices to the zero and identity matrices.

```
Matrix4d B = Matrix4d::Zero();     // 4x4 zero matrix
Matrix3d C = Matrix3d::Identity(); // 3x3 identity matrix
...
B.setZero();                       // back to zero
...
B.setIdentity();                   // back to identity
```

See http://eigen.tuxfamily.org/dox/group__TutorialAdvancedInitialization.html for more information.

Printing a matrix

You can print the matrix to the screen by using `cout`:

```
cout << A << endl;
```

will print out

```
1 2 3 4
5 6 7 8
```

Element access

You can access an element for read/write with the `()` operator. Rows and columns start with 0, not 1.

```
A(0,0) = 9;
double foo = A(1,3);
```

Arithmetic operations

Eigen overloads the common operators, so you can add, subtract, and multiply matrices easily.

```
Matrix4d A, B;      // 4x4 matrices
A << ...;
B << ...;
Matrix4d C = A + B; // 4x4 matrix
double s = 2.0;
Matrix4d D = s * C; // 4x4 matrix
MatrixXd E(4,3);    // 4x3 matrix
E << ...
MatrixXd F(4,3);    // 4x3 matrix
F = C * E;
```

The matrix sizes aren't checked at compile time. If there is a mismatch, you'll get a very complex looking runtime error - one drawback of template programming. *Eigen warns against using the `auto` keyword.* See http://eigen.tuxfamily.org/dox/TopicPitfalls.html. In general, it is a good idea to compile and run your code often so that Eigen errors are easier to find.

Block access

You can read/write blocks of elements with the `block` function.

```
Matrix2d B; // 2x2 matrix
B << -1, -2,
     -3, -4;
A.block<2,2>(0,0) = B;
```

The code above writes `B` into the upper 2x2 block of `A`. The numbers in the angled bracket indicate the size of the block, and the numbers in parenthesis indicate the starting location. You can also read a block in the same way.

```
Matrix2d C = A.block<2,2>(0,2); // 2x2 matrix
```

This sets `C` to be the right half of `A`.

For vectors, use the `segment` function.

```
Vector4d a; // 4x1 vector
a << 1, 2, 3, 4;
Vector2d b = a.segment<2>(0); // 2x1 vector
```

This sets `b` to be the first two elements of `a`.

Rotation matrix

You can create a rotation matrix from an axis-angle as follows. The input axis to the `AngleAxis` constructor must be normalized.

```
#include <Eigen/Geometry>
...
float angle = M_PI/6.0f;
Vector3f axis(1.0f, 1.0f, 1.0f);
axis.normalize();
Matrix3f R(AngleAxisf(angle, axis));
Matrix4f M;
M.setIdentity();
M.block<3,3>(0,0) = R;
cout << M << endl;
```

Similarly, you can create a rotation matrix from a unit quaternion.

```
#include <Eigen/Geometry>
...
float x, y, z, w;
x = y = z = w = 1.0f;
Quaternionf q(w, x, y, z);
q.normalize();
Matrix3f R(q);
Matrix4f M;
M.setIdentity();
M.block<3,3>(0,0) = R;
cout << M << endl;
```

Compilation Hints

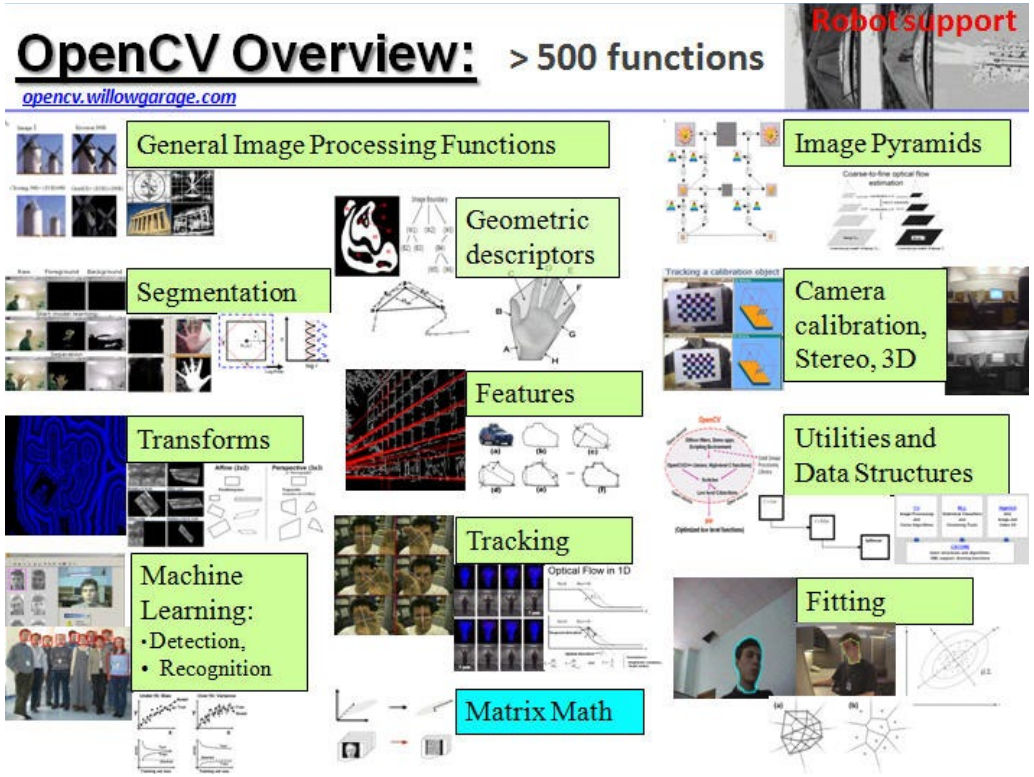
- If you have structures/classes that include Eigen members, put the following inside your class: `public: EIGEN_MAKE_ALIGNED_OPERATOR_NEW`.
- If you're having trouble running your application with Eigen, try `#define EIGEN_DONT_ALIGN_STATICALLY` before you `#include <Eigen/Dense>`.

For more hints, see http://eigen.tuxfamily.org/dox/group__DenseMatrixManipulation__Alignement.html.

Generated on Sun Aug 16 14:53:54 CDT 2020

OpenCV

- Open-source library for computer vision & machine learning
- Originally developed by Intel, now maintained by developers worldwide
- Written in C++, has interfaces for Python, Java, MATLAB, …
- Many functionalities & algorithms:
  - Image processing
  - Feature detection
  - Object detection
  - Camera calibration
  - Machine learning (deep learning, lassification, clustering, regression)



OpenCV

- Introduction slides from

  https://web.stanford.edu/class/cs231a/sessions/session4_opencv_tutorial.pdf

Introduction to OpenCV (3.1)

- Open source computer vision and machine learning library
- Contains implementations of a large number of vision algorithms
- Written natively in C++, also has C, Python, Java, and MATLAB interfaces
- Supports Windows, Linux, Mac OS X, Android, and iOS

Installation

- Download from http://opencv.org and compile from source
- Windows: Run executable downloaded from OpenCV website
- Mac OS X: Install through MacPorts
- Linux: Install through the package manager (e.g. `yum`, `apt`) but make sure the version is sufficiently up-to-date for your needs
- OpenCV uses the `cv` namespace (omitted during this presentation to save space)

Images

- Images in OpenCV are stored in a `Mat` object
- Consists of a matrix header and a pointer to the matrix containing the pixel values
- Header contains information such as the size of the matrix, the number of color channels in the image, etc.
- Access this information through functions:
  - `Mat.rows()`: Returns the number of rows
  - `Mat.columns()`: Returns the number of columns
  - `Mat.channels()`: Returns the number of channels

Reading, Writing, and Displaying Images

- Read the color image named `"foo.png"`
  - `Mat foo=imread("foo.png",IMREAD_COLOR);`
- Write the data contained in `foo` to `"bar.png"`
  - `imwrite("bar.png",foo);`
- Create a window
  - `namedWindow("display",WINDOW_AUTOSIZE);`
- Display the data stored in `foo`
  - `imshow("display",foo);`

Image Types

- The type of a `Mat` object specifies how to interpret the underlying binary data
- Represented as `CV_<Datatype>C<#Channels>`
- Example: `CV_8UC3` represents an image with three channels each represented by an 8-bit unsigned integer
  - `Mat.type()`: Returns an identifier that specifies the underlying arrangement of the data
  - `Mat.depth()`: Returns the number of bytes for a matrix element which depends on the datatype

Pixel Types

- Also need to know the underlying ordering of the channels
- OpenCV defaults to `BGR`
- Can also handle other types of channels/orderings such as `RGB`, `HSV` or `GRAY`
- Can access the value for a particular channel at a specific pixel once we have know how to interpret the underlying data
  - `Mat.at<datatype>(row,col)[channel]`: Returns a pointer to the image data at the specified location
- Convert between color spaces using `cvtColor(foo,bar,CV_BGR2GRAY)`

Copying Images

- In order to avoid making unnecessary copies of images, OpenCV uses a reference counting system to manage the underlying matrix data and *performs shallow copies by default*
- The copy constructor and assignment operator only copy the headers and the pointer to the underlying data
- Use `Mat::clone()` and `Mat::copyTo()` to perform a deep copy

Other Operations

- `Mat` object has many more capabilities
- Refer to the OpenCV Documentation for more details

Image Normalization and Thresholding

- Normalization remaps a range of pixel values to another range of pixel values
  - `void normalize(InputArray src,OutputArray dst,...)`
- OpenCV provides a general purpose method for thresholding an image
  - `double threshold(InputArray src,OutputArray dst,double thresh,double maxval,int type)`
  - Specify thresholding scheme specified by the `type` variable

Image Smoothing

- Reduces the sharpness of edges and smooths out details in an image
- OpenCV implements several of the most commonly used methods
- `void GaussianBlur(InputArray src,OutputArray dst,...)`
- `void medianBlur(InputArray src,OutputArray dst,...)`

Image Smoothing: Code

```
#include <cv.h>
#include <cvaux.h>
#include <highgui.h>

int main(int argc,char** argv){
    //Read in colored image
    cv::Mat image=cv::imread(argv[1]);
    cv::imwrite("photo.jpg",image);
    //Apply Gaussian blur
    cv::Mat image_gaussian_blur;
    image.convertTo(image_gaussian_blur,CV_8UC3);
    cv::GaussianBlur(image_gaussian_blur,image_gaussian_blur,cv::Size(0,0),9);
    cv::imwrite("photo_gaussian_blur.jpg",image_gaussian_blur);

    //Apply median blur
    cv::Mat image_median_blur;
    image.convertTo(image_median_blur,CV_8UC3);
    cv::medianBlur(image_median_blur,image_median_blur,17);
    cv::imwrite("photo_median_blur.jpg",image_median_blur);
}
```

Gaussian Blur: Original Image

Gaussian Blur: Result

Median Blur: Original Image

Median Blur: Result

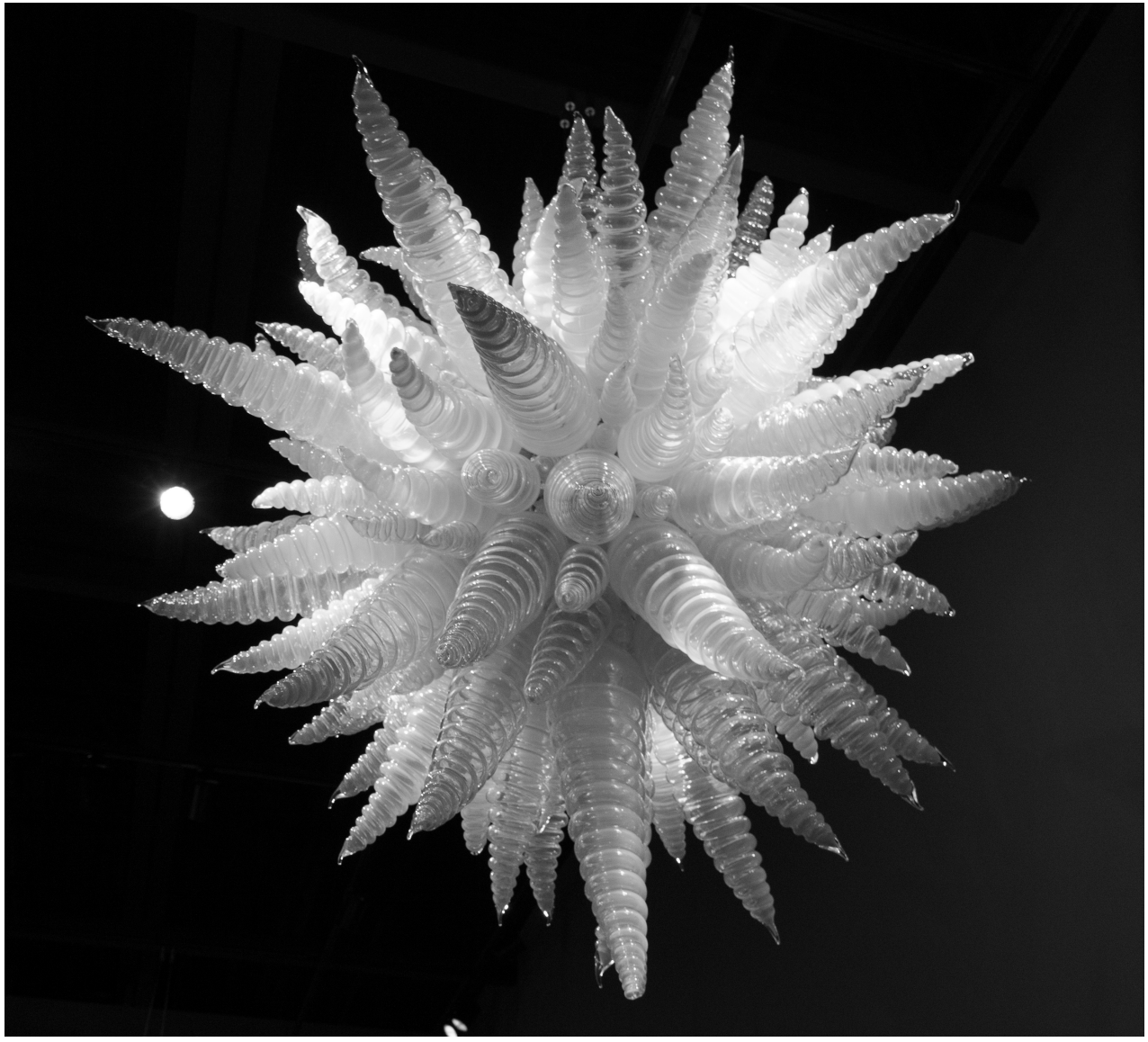
Edge Detection

- OpenCV implements a number of operators to help detect edges in an image
  - Sobel Operator
  - Scharr Operator
  - Laplacian Operator
- OpenCV also implements image detection algorithms such as Canny edge detection
- Tip: If your image is noisy, then edge detection will often exaggerate the noise
- Sometimes smoothing the image before running edge detection gives better results

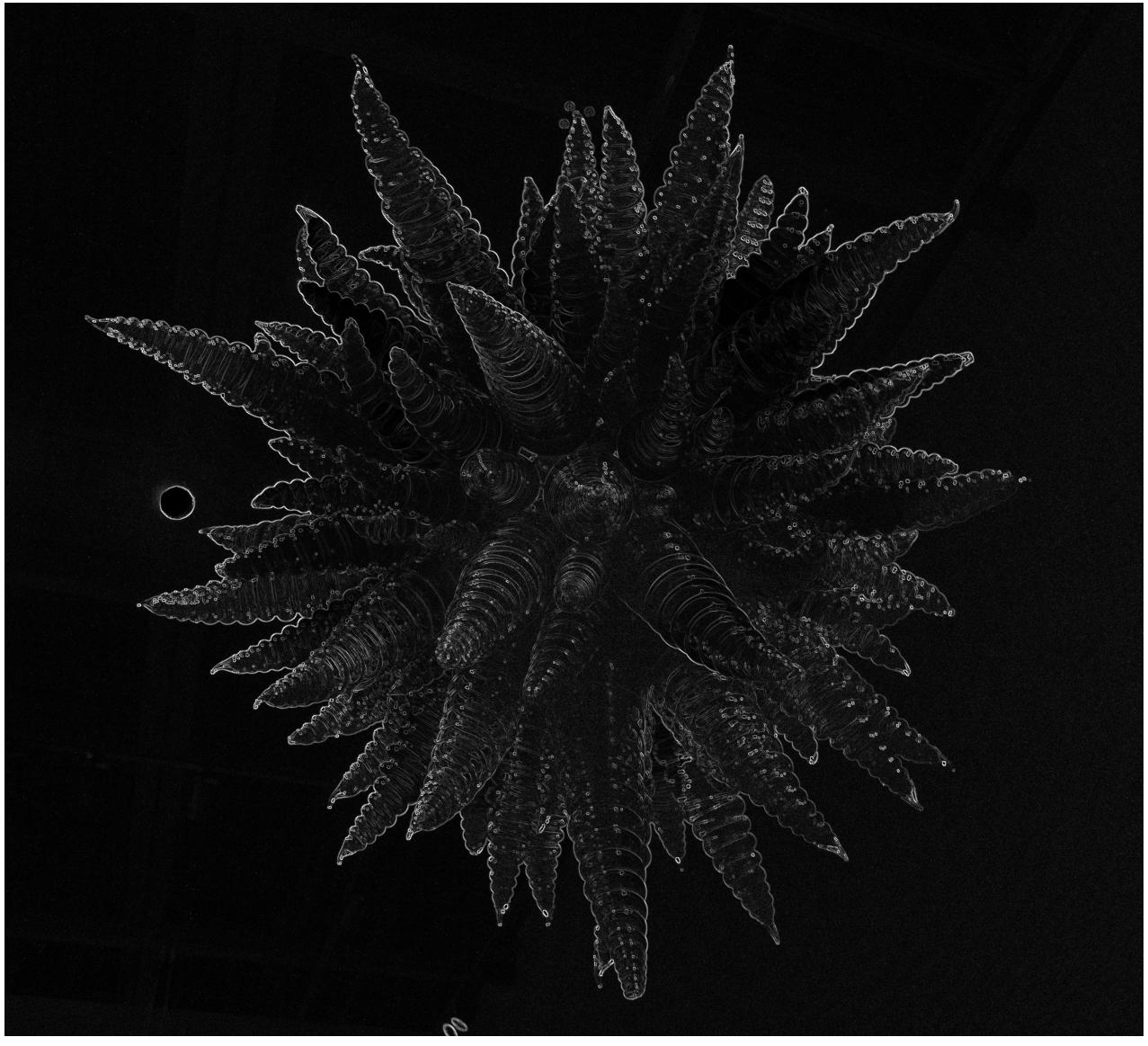
Edge Detection: Code

```
#include <cv.h>
#include <cvaux.h>
#include <highgui.h>

int main(int argc,char** argv){
    //Read image as grayscale, delete zero to read in color
    cv::Mat image=cv::imread(argv[1],0);
    cv::imwrite("photo_gray.jpg",image);

    //Calculate x-gradient using Sobel operator
    cv::Mat image_gradient_x;
    image.convertTo(image_gradient_x,CV_32FC1);
    cv::Sobel(image_gradient_x,image_gradient_x,CV_32FC1,0,1);
    //Absolute value and normalize
    cv::convertScaleAbs(image_gradient_x,image_gradient_x);

    //Calculate y-gradient using Sobel operator
    cv::Mat image_gradient_y;
    image.convertTo(image_gradient_y,CV_32FC1);
    cv::Sobel(image_gradient_y,image_gradient_y,CV_32FC1,1,0);
    //Absolute value and normalize
    cv::convertScaleAbs(image_gradient_y,image_gradient_y);

    //Average the x and y gradients into one image
    cv::Mat image_gradient;
    cv::addWeighted(image_gradient_x,0.5,image_gradient_y,0.5,0,image_gradient);
    cv::imwrite("photo_gradient.jpg",image_gradient);
}
```

Sobel Edge Detection: Grayscale Original Image

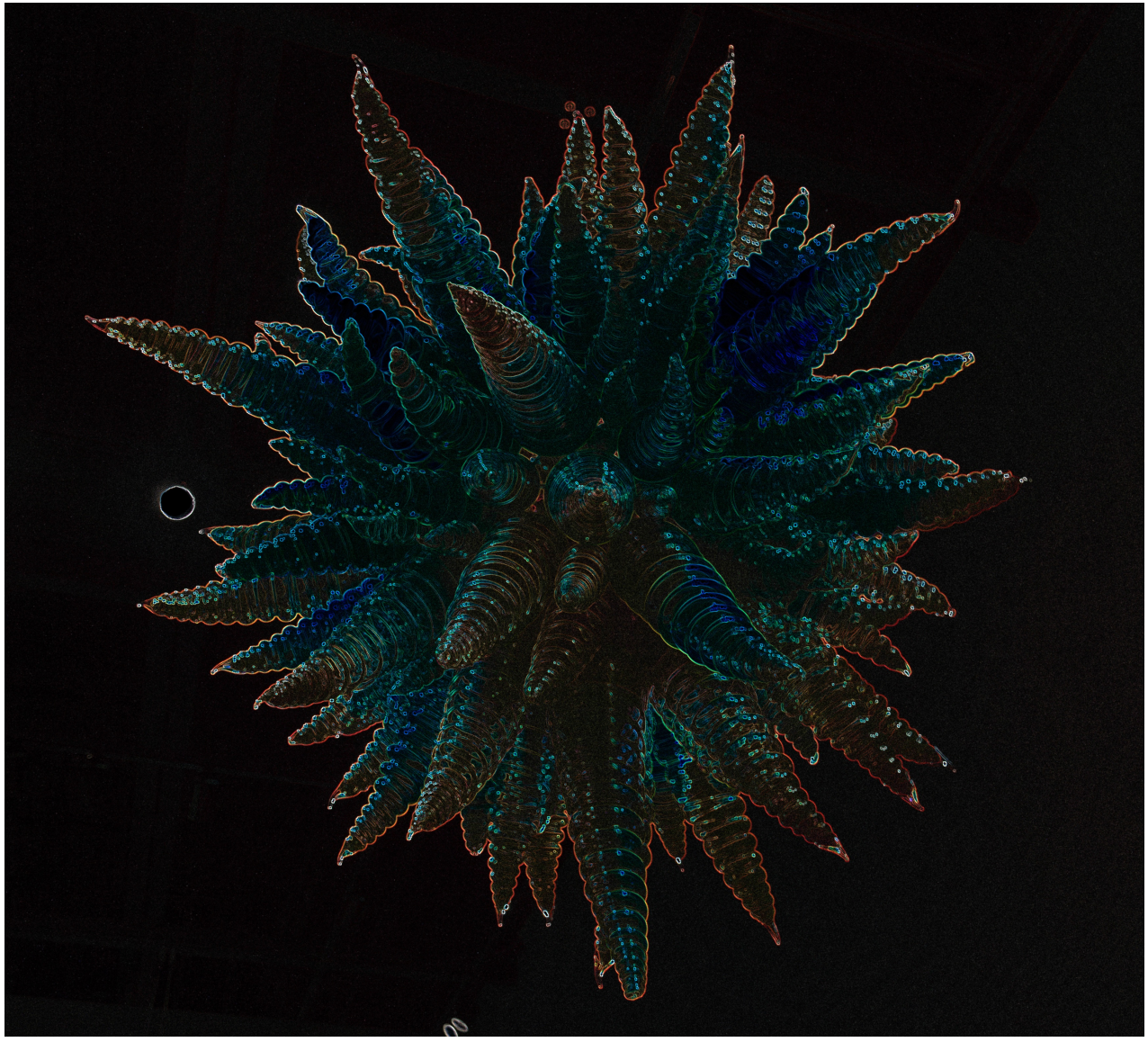


Sobel Edge Detection: Grayscale Result



Sobel Edge Detection: Color Original Image

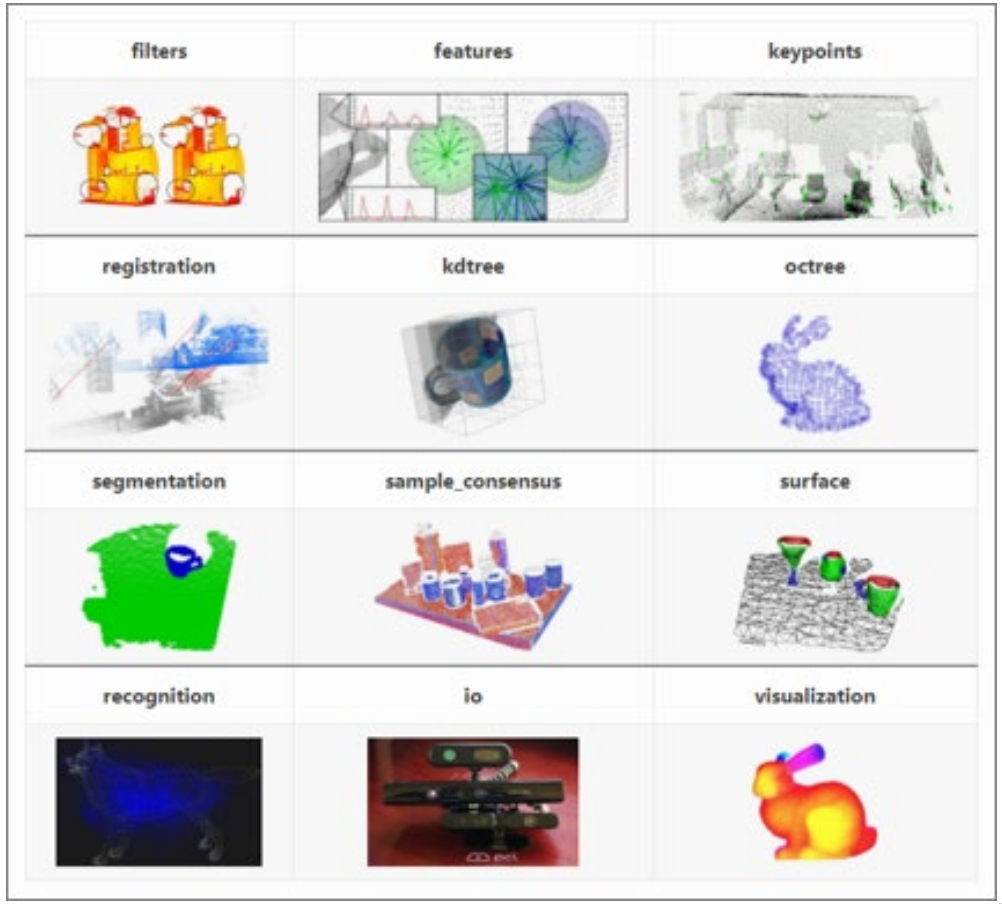
Sobel Edge Detection: Color Result



PCL

- Open-source library for point cloud processing
- Many functionalities & algorithms:
  - Point cloud processing
  - Feature extraction
  - Point cloud registration
  - Visualization

pointcloudlibrary



PCL

- Introduction slides from https://www.diag.uniroma1.it/~nardi/Didattica/LabAI/matdid/pcl_intro.pdf

Point clouds: a definition

A point cloud is a data structure used to represent a collection of **multi-dimensional points** and is commonly used to represent three-dimensional data.

The points usually represent the X, Y, and Z geometric coordinates of a sampled surface.

Each point can hold additional information: RGB colors, intensity values, etc...

Point Cloud Library

→ pointclouds.org

The Point Cloud Library (PCL) is a standalone, large scale, open source (C++) library for 2D/3D image and point cloud processing.

PCL is released under the terms of the BSD license, and thus free for commercial and research use.

PCL provides the 3D processing pipeline for ROS, so you can

also get the perception pcl stack and still use PCL standalone.

Among others, PCL depends on Boost, Eigen, OpenMP,...

PCL Basic Structures: PointCloud

A PointCloud is a templated C++ class which basically contains the following data fields:

- **width (int)** - specifies the width of the point cloud dataset in the number of points.
  → the total number of points in the cloud (equal with the number of elements in points) for unorganized datasets
  → the width (total number of points in a row) of an organized point cloud dataset
- **height (int)** - Specifies the height of the point cloud dataset in the number of points
  → set to 1 for unorganized point clouds
  → the height (total number of rows) of an organized point cloud dataset
- **points (std::vector <PointT>)** - Contains the data array where all the points of type PointT are stored.
-

PointCloud vs. PointCloud2

We distinguish between two data formats

for the point clouds:

- **PointCloud<PointType>** with a specific data type (for actual usage in the code)
- **PointCloud2** as a general representation containing a header defining the point cloud structure (e.g., for loading, saving or sending as a ROS message)

Conversion between the two frameworks is easy:

- **pcl::fromROSMsg** and **pcl::toROSMsg**

Important: clouds are often handled using smart pointers, e.g.:

- PointCloud<PointType> :: Ptr cloud_ptr;

Point Types

PointXYZ - float x, y, z

PointXYZI - float x, y, z, intensity

PointXYZRGB - float x, y, z, rgb

PointXYZRGBA - float x, y, z, uint32 t rgba

Normal - float normal[3], curvature

PointNormal - float x, y, z, normal[3], curvature

→ See pcl/include/pcl/point_types.h for more examples.

Building PCL Stand-alone Projects

```
# CMakeLists.txt
project( pcl_test )
cmake_minimum_required (VERSION 2.8)
cmake_policy(SET CMP0015 NEW)

find_package(PCL 1.7 REQUIRED )
add_definitions(${PCL_DEFINITIONS})

include_directories(... ${PCL_INCLUDE_DIRS})
link_directories(... ${PCL_LIBRARY_DIRS})

add_executable(pcl_test pcl_test.cpp ...)
target_link_libraries( pcl_test${PCL_LIBRARIES})
```

PCL structure

PCL is a collection of smaller, modular C++ libraries:

- **libpcl_features**: many 3D features (e.g., normals and curvatures, boundary points, moment invariants, principal curvatures, Point Feature Histograms (PFH), Fast PFH, ...)
- **libpcl_surface**: surface reconstruction techniques (e.g., meshing, convex hulls, Moving Least Squares, ...)
- **libpcl_filters**: point cloud data filters (e.g., downsampling, outlier removal, indices extraction, projections, ...)
- **libpcl_io**: I/O operations (e.g., writing to/reading from PCD (Point Cloud Data) and BAG files)
- **libpcl_segmentation**: segmentation operations (e.g.,cluster extraction, Sample Consensus model fitting, polygonal prism extraction, ...)
- **libpcl_registration**: point cloud registration methods (e.g., Iterative Closest Point (ICP), non linear optimizations, ...)
- **libpcl_range_image**: range image class with specialized methods

It provides unit tests, examples, tutorials, ...

Point Cloud file format

Point clouds can be stored to disk as files, into the **PCD (Point Cloud Data) format**:

- # Point Cloud Data (PCD) file format v .5
  FIELDS x y z rgba
  SIZE 4 4 4 4
  TYPE F F F U
  WIDTH 307200
  HEIGHT 1
  POINTS 307200
  DATA binary
  ...<data>...

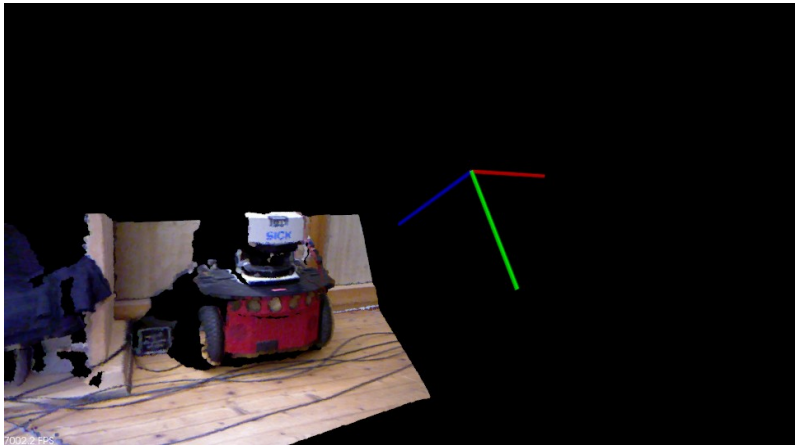
Funtions: **pcl::io::loadPCDFile** and **pcl::io::savePCDFile**

Example: create and save a PC

```
#include<pcl/io/pcd_io.h>
#include<pcl/point_types.h>
 //....
 pcl::PointCloud<pcl::PointXYZ>:: Ptr cloud_ptr (new pcl::PointCloud<pcl::PointXYZ>);
 cloud->width=50;
 cloud->height=1;
 cloud->isdense=false;
 cloud->points.resize(cloud.width*cloud.height);
 for(size_t i=0; i<cloud.points.size(); i++){
  cloud->points[i].x=1024*rand()/(RANDMAX+1.0f);
  cloud->points[i].y=1024*rand()/(RANDMAX+1.0f);
  cloud->points[i].z=1024*rand()/(RANDMAX+1.0f);
 }
 pcl::io::savePCDFileASCII("testpcd.pcd",*cloud);
```

Visualize a cloud

```
viewer->setBackgroundColor (0, 0, 0);
viewer->addPointCloud<pcl::PointXYZ> ( in_cloud, cloud_color,
                                                  "Input cloud" );
viewer->setPointCloudRenderingProperties
    (pcl::visualization::PCL_VISUALIZER_POINT_SIZE, 1, "Input cloud");
viewer->initCameraParameters ();
viewer->addCoordinateSystem (1.0);
while (!viewer->wasStopped ())   viewer->spinOnce ( 1 );
```



Basic Module Interface

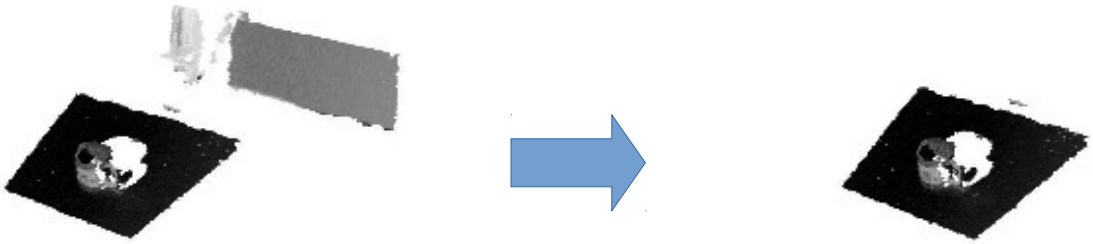
Filters, Features, Segmentation all use the same basic usage interface:

- use **setInputCloud()** to give the input
- set some parameters
- call **compute()** or **filter()** or **align()** or **...** to get the output

PassThrough Filter

Filter out points outside a specified range in one dimension.

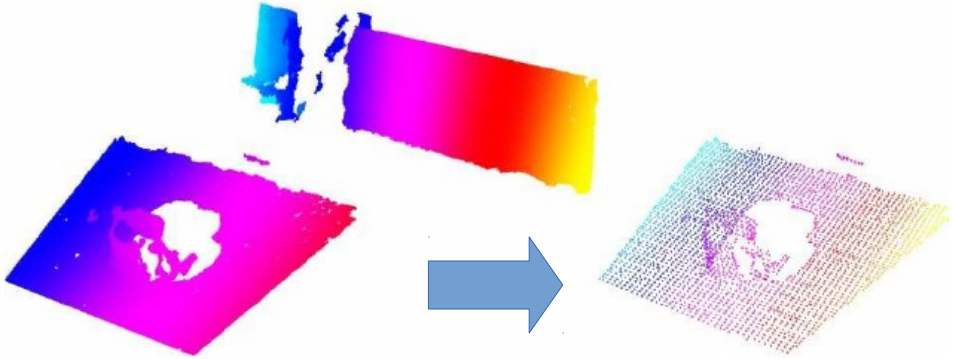
- pcl::PassThrough<T> pass_through;
  pass_through.setInputCloud (in_cloud);
  pass_through.setFilterLimits (0.0, 0.5);
  pass_through.setFilterFieldName ("z");
  pass_through.filter(*cutted_cloud);



Downsampling

Voxelize the cloud to a 3D grid. Each occupied voxel is approximated by the centroid of the points inside it.

- pcl::VoxelGrid<T> voxel_grid;
  voxel_grid.setInputCloud (input_cloud);
  voxel_grid.setLeafSize (0.01, 0.01, 0.01);
  voxel_grid.filter (*subsamp_cloud) ;



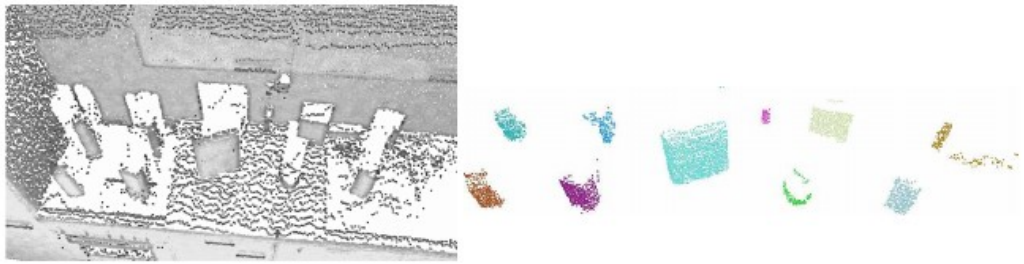
Features example: normals

```
pcl::NormalEstimation<T, pcl::Normal> ne;
ne.setInputCloud (in_cloud);
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new
                        pcl::search::KdTree<pcl::PointXYZ> ());
ne.setSearchMethod (tree);
ne.setRadiusSearch (0.03);
ne.compute ( *cloud_normals );
```

Segmentation example

A clustering method divides an unorganized point cloud int into smaller, correlated, parts. **EuclideanClusterExtraction** uses a distance threshold to the nearest neighbors of each point to decide if the two points belong to the same cluster.

- pcl::EuclideanClusterExtraction<T> ec;
  ec.setInputCloud (in_cloud);
  ec.setMinClusterSize (100);
  ec.setClusterTolerance (0.05); // distance threshold
  ec.extract (cluster_indices);



Iterative Closest Point - 1

ICP iteratively revises the transformation (translation,rotation) needed to minimize the distance between the points of two raw scans.

- **Inputs**: points from two raw scans, initial estimation of the transformation, criteria for stopping the iteration.
- **Output**: refined transformation.

Iterative Closest Point - 2

The algorithm steps are :

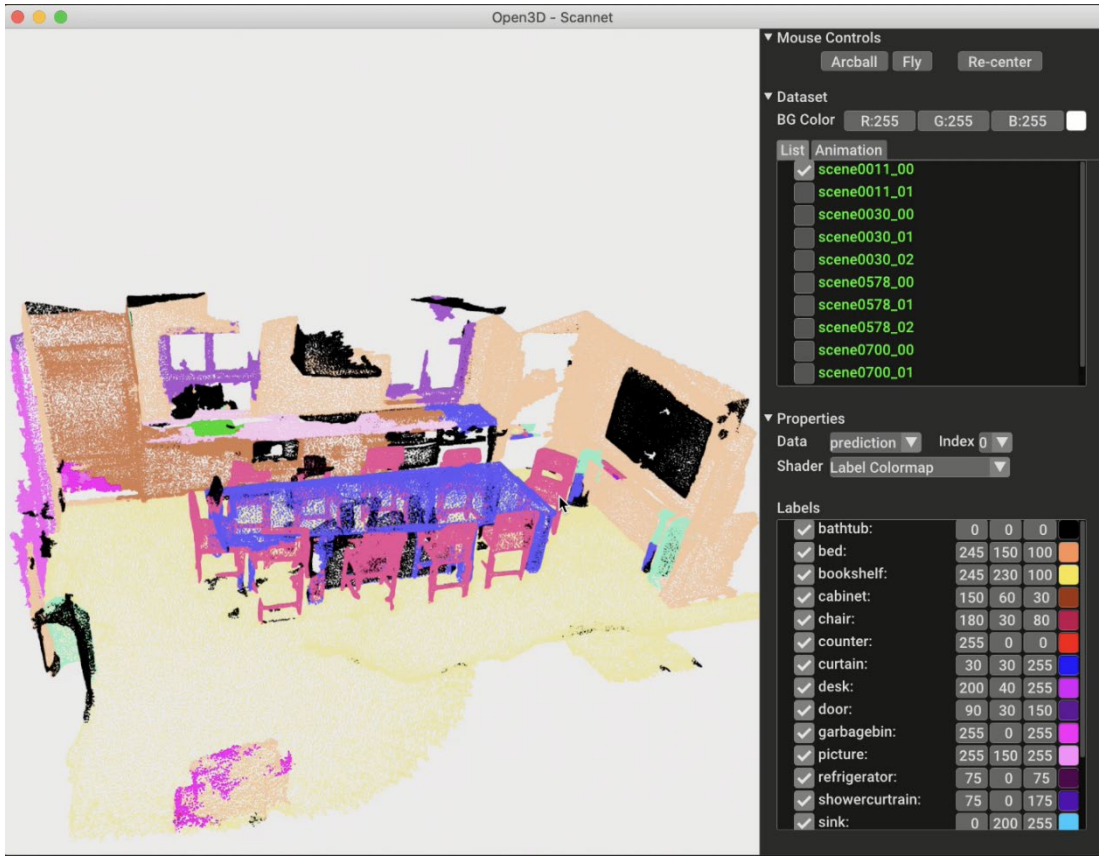
- 1. Associate points of the two cloud using the nearest neighbor criteria.
- 2. Estimate transformation parameters using a mean square cost function.
- 3. Transform the points using the estimated parameters.
- 4. Iterate (re-associate the points and so on).

Iterative Closest Point - 3

```
IterativeClosestPoint<PointXYZ, PointXYZ> icp;
// Set the input source and target
icp.setInputCloud (cloud_source);
icp.setInputTarget (cloud_target);
// Set the max correspondence distance to 5cm
icp.setMaxCorrespondenceDistance (0.05);
// Set the maximum number of iterations (criterion 1)
icp.setMaximumIterations (50);
// Set the transformation epsilon (criterion 2)
icp.setTransformationEpsilon (1e-8);
// Set the euclidean distance difference epsilon (criterion 3)
icp.setEuclideanFitnessEpsilon (1);
// Perform the alignment
icp.align (cloud_source_registered);
// Obtain the transformation that aligned cloud_source to cloud_source_registered
Eigen::Matrix4f transformation = icp.getFinalTransformation ();
```

Open3D

- Open source library for 3D data processing (point clouds, meshes, voxels)
- Has Python bindings for easy integration with Python projects
- Many functionalities & algorithms:
  - Point cloud processing
  - Feature extraction
  - Point cloud registration
  - Visualization
- Compared to PCL, Open3D is more user-friendly & modern, supports Python & recent ML/DL techniques



Open3D

- Introduction slides from https://goodgodgd.github.io/ian-flow/archivers/open3d-tutorial

Introduction to Open3D

goodgodgd.github.io/ian-flow/archivers/open3d-tutorial

1. What is Open3D?

Open3D is a library that collects tools for handling 3D data. There are various expressions of 3D data such as (RGB-)Depth image, point cloud, 3D voxel, and mesh. The major functions provided by Open3D are as follows.

- Provides data structures (classes) for various types of 3D data
- Ability to read and write different standard file formats for each data type
- Ability to visualize 2D/3D data
- 2D image processing algorithm (filtering, etc.)
- Various algorithms for 3D data
    - Local/Global Registration (ICP 등)
    - Normal Estimation
    - KD Tree
    - TSDF Volume Integration
    - Pose-graph Optimization

These features were originally also present in PCL (Point Cloud Library). However, Open3D has features that set it apart.

- Minimized dependency on 3rd party libraries: In short, **installation is easy.** To build PCL once, you had to install at least five or six 3rd party libraries or build it directly from source and link it with PCL. (This task alone takes a day.) However, Open3D has a small number of 3rd party libraries, and the necessary ones are included in the Open3D repository. So, you can build it together within Open3D without having to download a separate 3rd party.
- Lightweight: There are few 3rd party libraries and the internal structure is simple, so the number and size of library installation files are small.~~This is completely different from PCL, where only library files are in GB units.~~
- Python/C++ support: Open3D's official documentation explains it based on Python. Internally, it was implemented and optimized in C++, but it is easier to use in Python by creating a Python API. Of course, there is a C++ API, but there is no documentation explaining it, and you have to learn how to use it by looking at the API reference and various example files . For reference, PCL also has a Python API. (link)
- Convenience of using C++: PCL applies C++'s template syntax almost everywhere, so it has good optimization and scalability, but has many inconveniences in coding. Open3D makes it easier to read and write code by limiting it to one commonly used type.
- Cross platform: Works on Ubuntu/Windows/MacOS.

Because it is a new project first released in February 2018, there are some features (segmentation, etc.) that are present in PCL but not in Open3D. However, more recent algorithms for the same function are included, and Open3D also supports Graph Optimization and PointNet, which are not available in PCL. As it is open source, it is likely to develop further in the future.

The developer of Open3D is Intel, and it seems that Intel has also developed related software while making 3D sensors. It was developed by many people, but there is one person who contributed the most, a Korean named Park Jae-sik. It is said that he worked at Intel after receiving his doctorate from KAIST and recently went to POSTECH.

2. How to Install

If you use Open3D with Python, simply `pip install open3d-python`type:

However, when using C++, you must download the source code from https://github.com/intel-isl/Open3D and build it. It is not as complicated as OpenCV or PCL and does not take long. The guide below is based on Ubuntu (16.04 or 18.04).

2.1 Build Library

1. Get repository (note **the –recursive option)**

```
git clone --recursive https://github.com/intel-isl/Open3D
cd Open3D
```

2. install dependencies

```
sudo ./util/scripts/install-deps-ubuntu.sh
sudo apt install cmake-qt-gui
```

3. Run cmake-gui

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

4. “Configure” and “Finish” 클릭
5. “BUILD_xxx” option setting
    1. Uncheck: **PYBIND11** and **PYTHON_MODULE**
    2. Check: **SHARED_LIBS**, **TINYFILEDIALOGS** and **EIGEN3**
6. Specify the library installation path in “CMAKE_INSTALL_PREFIX”
7. Click “Configure”, “Generate” and exit cmake-gui

8. Build and Install

```
make -j3
make install
cp lib/* /path/to/install/lib/
```

2.2 Import Open3D to QtCreator Project

After creating a project in QtCreator, `.pro`add the following script to the file

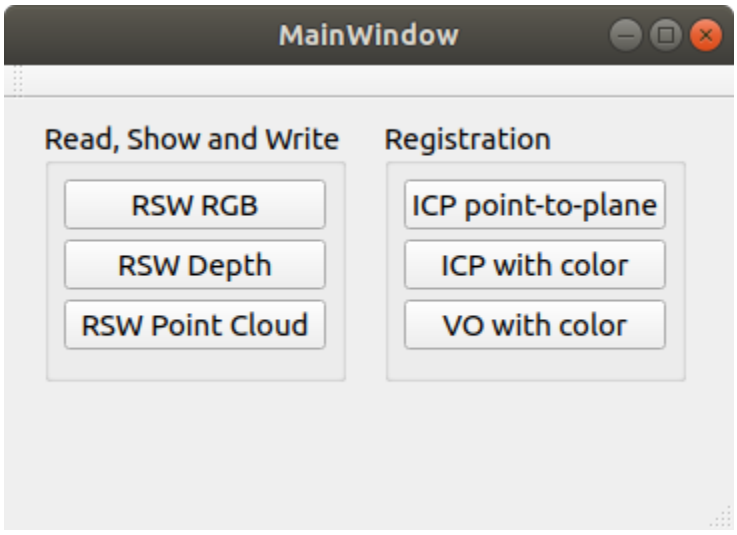
```
INCLUDEPATH += <install path>/open3d/include \
               <install path>/open3d/include/Open3D/3rdparty/fmt/include \
               <install path>/open3d/include/Open3D/3rdparty/Eigen

LIBS += -L<install path>/open3d/lib \
        -lOpen3D
```

3. Open3D Tutorial (C++)

The main topic of this post is how to use the C++ API for Open3D. There is already good official documentation for how to use Python, so you can view and follow it whenever necessary. (Python Basic Tutorial) Here, I would like to explain how to use C++, which is not covered in the official documentation but is more commonly used in companies. Although it is not documented, C++ also provides API references and examples to help you understand how to use it. There are two main topics covered here.

1. **File IO and Visualization** : Learn how to read data from a file, check it on the screen, and save it.
2. **Point Cloud Registration** : Register the point cloud through ICP and learn about ICP using RGB information.

The example program allows you to execute each function one by one through the following Qt GUI.



The main namespaces used in the example are as follows. You can know the function just by looking at the names of the classes and functions in this namespace.

- `open3d::geometry`: Contains data types for (3D) data such as Image or PointCloud.
- `open3d::io`: There are functions that read and write geometry data types to files.
- `open3d::visualization`: There are functions that visualize geometry data types.
- `open3d::registration`: There are registration functions between PointCloud.

3.1 File IO and Visualization

Open3D is worth using just to read and display the file format that stores the point cloud `.pcd`or the format that stores the mesh `.ply`.

The links below are related Python tutorials. Python tutorials are helpful because they explain related content and C++ also has similar classes and functions.

- FileIO: http://www.open3d.org/docs/release/tutorial/Basic/file_io.html
- Visualization: http://www.open3d.org/docs/release/tutorial/Basic/visualization.html

3.1.1 Depth Image

When you click “RSW Depth” in the GUI, the following function is executed. First, read the Depth image saved in 16bit PNG format. It shows two ways: 1) showing the depth image and saving it as a file, and 2) converting it to a point cloud and showing it. RGB images can also be read and displayed in almost the same way.

```
void MainWindow::on_pushButton_rsw_points_clicked()
{
    IoVis_Examples::ReadShowWrite_PointCloud("../samples/color1.png",
                        "../samples/depth1.png", "../results/pointcloud1.pcd");
}
```

a. Read depth file

`open3d::io::CreateImageFromFile()`You can read the video file by using and `IsEmpty()`check whether it was loaded properly using the member function.

```
// iovis_examples.h
void IoVis_Examples::ReadShowWrite_Depth(const char* srcname,
                                                                    const char* dstname, bool
write_scaled)
{
    o3ImagePtr depth_ptr = open3d::io::CreateImageFromFile(srcname);
    if(depth_ptr->IsEmpty())  {
        open3d::utility::LogError("Failed to read {}\n\n", srcname);
        return;
    }
    LogImageDimension(image_ptr, "color image");
    // 출력: [Open3D INFO] depth image size: 640 x 480 x 1 (2 bytes per channel)
        // ...
```

The type received as output `o3ImagePtr`has a long original type name, so a short alias is defined as follows. `o3ImagePtr`You `auto`can use instead of in the above code , but `auto`if you use type, the IDE will not autocomplete, making coding inconvenient. However, because it is cumbersome to write a long name, a short nickname is defined.

```
// definitions.h
typedef open3d::geometry::Image o3Image;
typedef std::shared_ptr<o3Image> o3ImagePtr;
typedef open3d::geometry::PointCloud o3PointCloud;
typedef std::shared_ptr<o3PointCloud> o3PointCloudPtr;
```

In the example, all types `std::shared_ptr`are wrapped with . This is because it is given when reading a file `shared_ptr`, and it is convenient in terms of variable passing and memory management.

`LogImageDimension()`shows the width, height, number of channels, and number of bytes of data `LogInfo()` through functions. `LogInfo()`Just like Python, `{}`variables are placed inside in order, so it can be conveniently used when printing results.

```
void IoVis_Examples::LogImageDimension(o3ImagePtr img_ptr, std::string name)
{
    open3d::utility::LogInfo("{} size: {:d} x {:d} x {:d} ({:d} bytes per channel)\n",
           name, img_ptr->width_, img_ptr->height_, img_ptr->num_of_channels_,
           img_ptr->bytes_per_channel_);
}
```

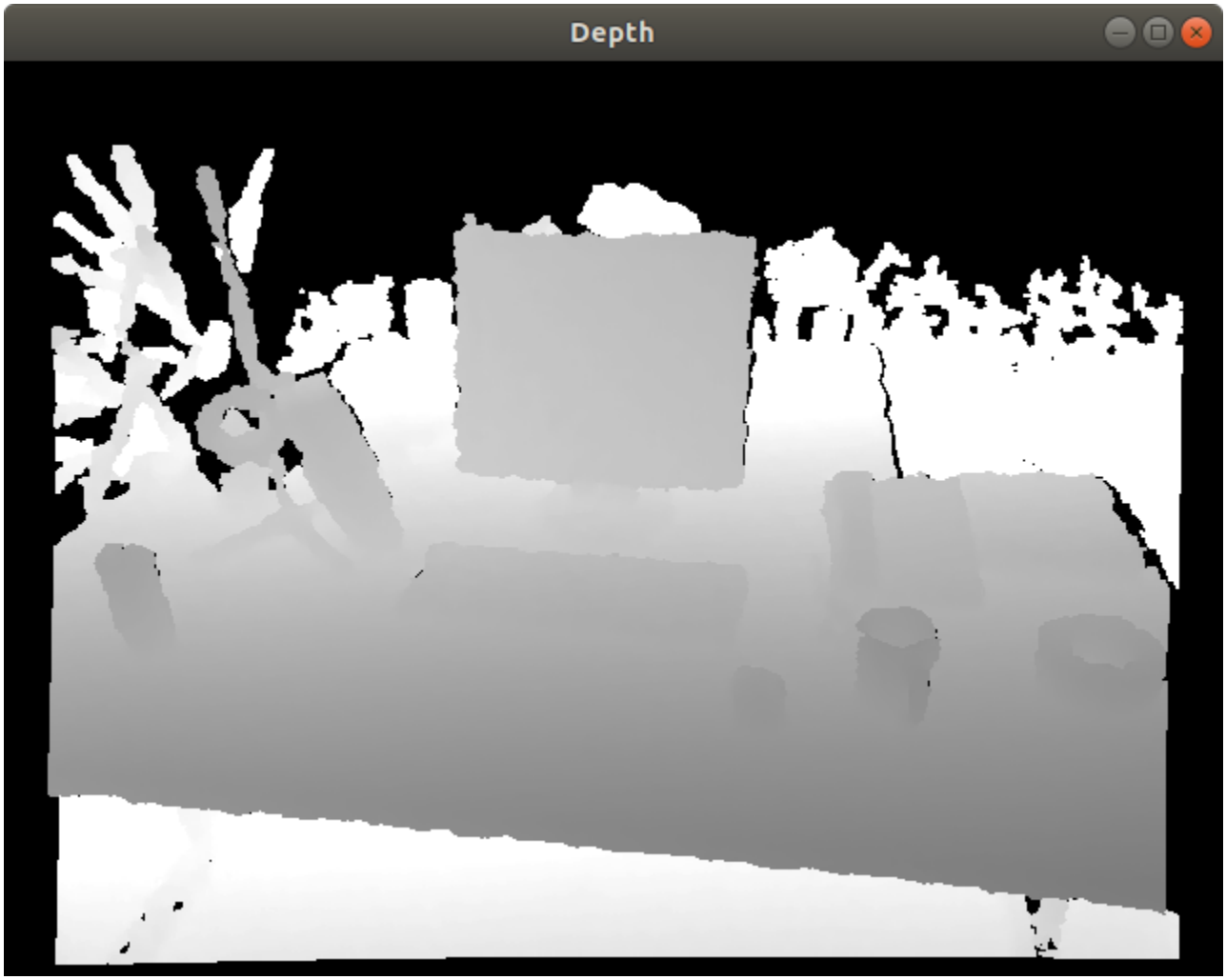
b. Visualize Depth

In order to display a depth image on the screen, it must be converted to a FloatImage. `ConvertDepthToFloatImage(scale);`It can be changed to a function, but the existing 16-bit integer depth must be divided by the scale value and an appropriate scale value must be specified so that most values fall between 0 and 1. If you set the scale too small, all pixels over 1 will look white, and if you set the scale too large, the overall look will be too dark, so you should set a value close to the maximum depth value. The code below reads and compares the original depth and the depth value converted to float. You can see that there is a 10000 times difference as you put it on the scale.

```
// convert depth scale
auto depth_float = depth_ptr->ConvertDepthToFloatImage(10000.0);
if(depth_float->IsEmpty())
{
    open3d::utility::LogError("Failed to convert to float image\n");
    return;
}
LogImageDimension(depth_float, "depth float image");

// compare depth
int raw_depth = *depth_ptr->PointerAt<uint16_t>(400, 320, 0);
float single_depth_float = depth_float->FloatValueAt(400, 320).second;
open3d::utility::LogInfo("raw depth={}, float depth={}\n",
                         raw_depth, single_depth_float);
    // 출력: [Open3D INFO] depth float image size: 640 x 480 x 1 (4 bytes per channel)
    // 출력: [Open3D INFO] raw depth=6122, float depth=0.6122
```

After converting the data to FloatImage, `DrawGeometries`you can easily display the image on the screen using a function.

```
// show depth
open3d::visualization::DrawGeometries({depth_float}, "Depth",
                      depth_ptr->width_, depth_ptr->height_);
```



c. Write depth image

Images `WriteImage()`can be written to a file using a function called, but if they are converted to float type, they must be converted back to integer type and saved.

```
if(write_scaled)
    open3d::io::WriteImage(dstname, *depth_float->CreateImageFromFloatImage<uint8_t>());
else
    open3d::io::WriteImage(dstname, *depth_ptr);
```

d. Visualize point cloud

If you want to view Depth as a point cloud, you must convert it to a point cloud. The information required for conversion is the intrinsic parameters of the camera. `SetIntrinsics()`Enter 6 parameters in the function.

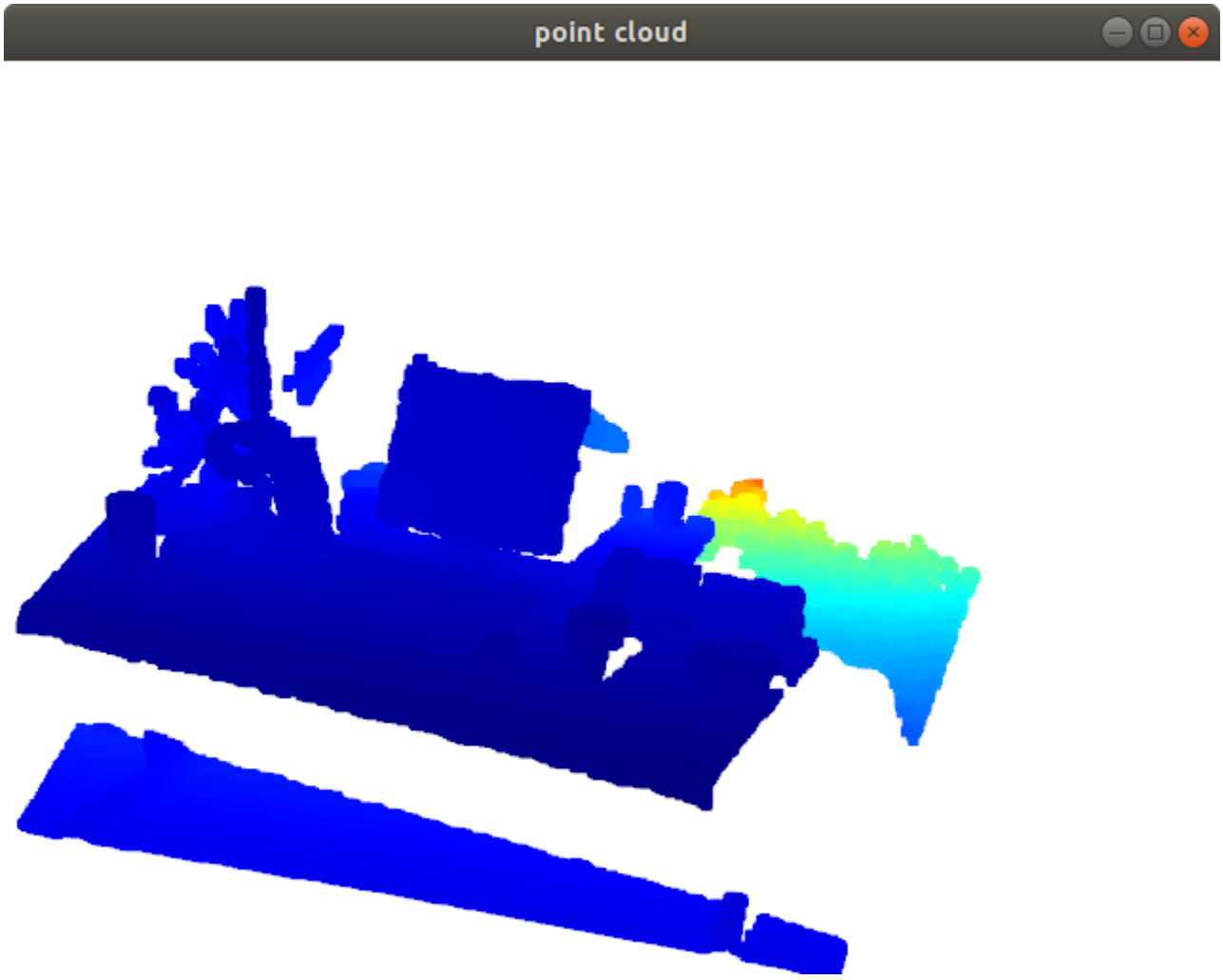
1. width: video width
2. height: video height
3. fx: horizontal focal length
4. fy: vertical focal length
5. cx: x-coordinate of the image center point

6. cy: y-coordinate of the image center point

Point clouds can also be displayed on the screen using the DrawGeometries function.

```
// convert depth to point cloud
open3d::camera::PinholeCameraIntrinsic camera;
camera.SetIntrinsics(640, 480, 575.0, 575.0, 319.5, 239.5);
o3PointCloudPtr pointcloud_ptr = o3PointCloud::CreateFromDepthImage(*depth_ptr, camera);

// show point cloud
    open3d::visualization::DrawGeometries({pointcloud_ptr}, "point cloud");
```

The following is a screen displaying a 3D point cloud. Depth is expressed in color. For reference, Open3D's default coordinate system is the camera coordinate system: x=bottom, y=left, z=front (depth).



3.1.2 RGB-Depth Image

When you click “RSW Point Cloud” in the GUI, the following function is executed.

```
void MainWindow::on_pushButton_icp_point_plane_clicked()
{
    RegistrationExamples::IcpPointCloud("../samples/depth1.png",
                                        "../samples/depth2.png");
}
```

a. Read rgb-d files

The method for reading RGB images and Depth images `CreateImageFromFile()`is the same.

```
void IoVis_Examples::ReadShowWrite_PointCloud(const char* colorname,
                     const char* depthname, const char* pcdname)
{
// read color and depth
    o3ImagePtr color_ptr = open3d::io::CreateImageFromFile(colorname);
    LogImageDimension(color_ptr, "color image");
    o3ImagePtr depth_ptr = open3d::io::CreateImageFromFile(depthname);
    LogImageDimension(depth_ptr, "depth image");
    if(color_ptr->IsEmpty() || depth_ptr->IsEmpty())
    {
        open3d::utility::LogError("Failed to read {} or {}\n\n",
                                  colorname, depthname);
        return;
    }
        // 출력
        // [Open3D INFO] color image size: 640 x 480 x 3 (1 bytes per channel)
        // [Open3D INFO] depth image size: 640 x 480 x 1 (2 bytes per channel)
```

b. Convert to RGB-D image

Open3D has a data type for RGB-D images `open3d::geometry::RGBDImage`.
`CreateFromColorAndDepth()`The arguments entered into the function are as follows.

1. RGB video and Depth video
2. depth_scale: Value divided when converting depth to float
3. depth_trunc: maximum valid depth value
4. convert_rgb_to_intensity: Determines whether to keep the color in RGB or convert it to gray scale when creating a point cloud.

```
    double depth_scale = 5000.0, depth_trunc = 3.0;
    bool convert_rgb_to_intensity = false;
    std::shared_ptr<open3d::geometry::RGBDImage> rgbd_ptr =
            open3d::geometry::RGBDImage::CreateFromColorAndDepth(
                *color_ptr, *depth_ptr, depth_scale, depth_trunc,
                convert_rgb_to_intensity);
```

c. Convert to point cloud and show

Point clouds can be created directly from depth as seen above, but to create colored point clouds, they must be created from rgbd images. (`CreateFromRGBDImage()`) In this case, intrinsic parameters are also needed, but in the case of commercial sensors, the values are stored in the library, so `enum`you can just enter the values.

```
    open3d::camera::PinholeCameraIntrinsic intrinsic(
            open3d::camera::PinholeCameraIntrinsicParameters::PrimeSenseDefault);
    o3PointCloudPtr ptcd_ptr = o3PointCloud::CreateFromRGBDImage(*rgbd_ptr, intrinsic);
```

The created `PointCloud`in this way contains three materials. `points_`must be present by default and `normals_`must `colors_`be entered or calculated for the value to be entered. Here `RGBDImage`, since it was created from , `colors_`is filled and `normals_`is empty.

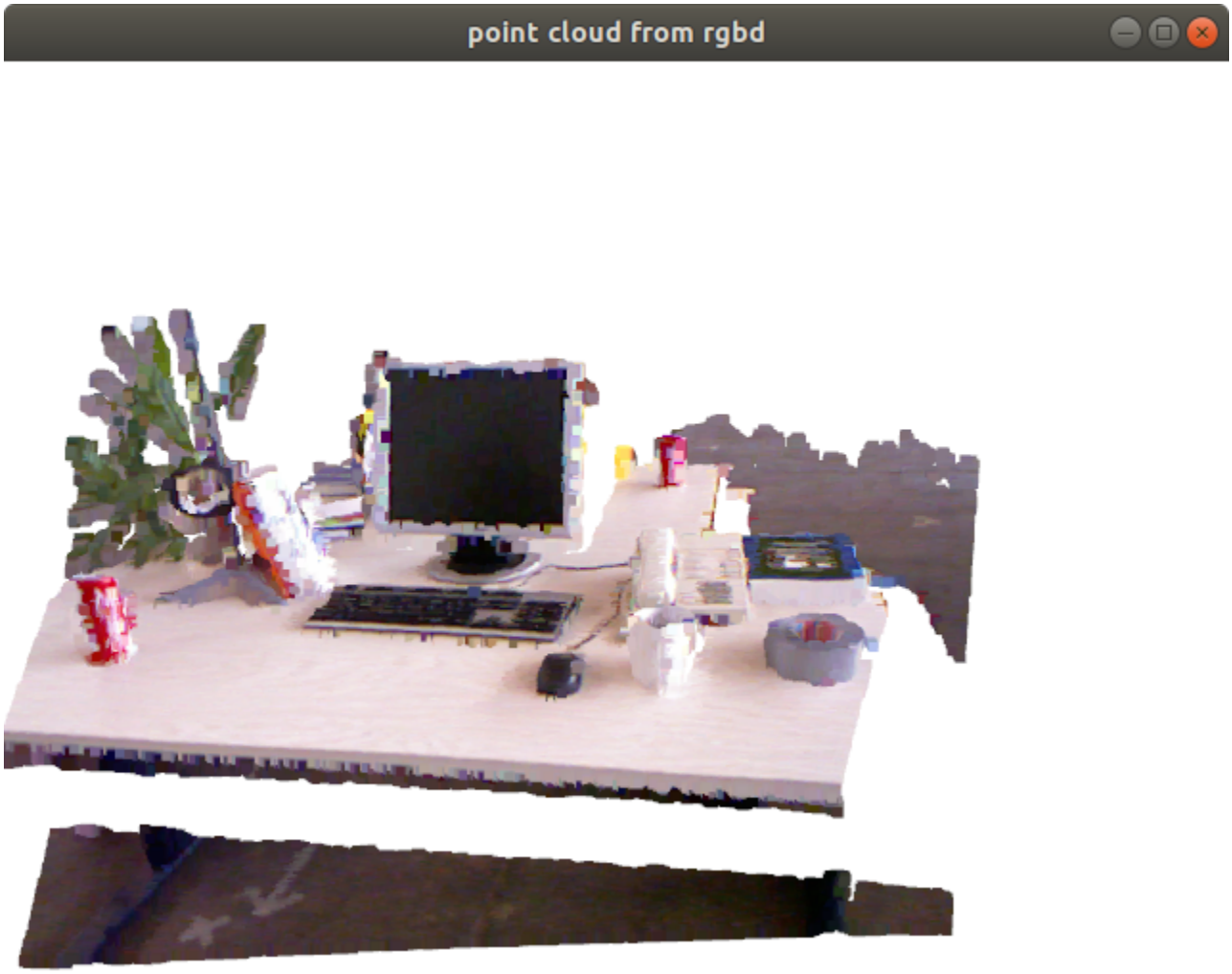
```
// PointCloud.h
class PointCloud : public Geometry3D {
public:
        std::vector<Eigen::Vector3d> points_;
    std::vector<Eigen::Vector3d> normals_;
    std::vector<Eigen::Vector3d> colors_;
```

Let's access the variable and check its value directly. Because the variable is public, it can be accessed directly from outside.

```
    uint32_t index = 240*480 + 400;
    open3d::utility::LogInfo("check point cloud values: point={} | color={}\n",
                             ptcd_ptr->points_[index].transpose(),
                             ptcd_ptr->colors_[index].transpose());
        // 출력: [Open3D INFO] check point cloud values:
        // point=0.259197 0.083068   0.6558 | color=0.811765 0.658824 0.654902
```

Afterwards, `DrawGeometries()`the 3D point cloud is shown as a function.

```
    open3d::visualization::DrawGeometries({ptcd_ptr}, "point cloud from rgbd");
```



d. Write, read and show point cloud

There is a function that stores the point cloud itself `WritePointCloud()`. If the input argument `write_ascii`is true, the data is saved as readable text, and if false, the data is saved in binary format with a high compression rate.

The function that reads the point cloud from a file is exactly what it says `CreatePointCloudFromFile()`. The first argument is the file name, the second is the file format, and the third is the argument that determines whether to print the process.

The last line shows the loaded point cloud again to visually check the data.

```
// write point cloud in ascii format without compression
bool write_ascii = true, compressed = false;
open3d::io::WritePointCloud(pcdname, *ptcd_ptr, write_ascii, compressed);

// read point cloud again
o3PointCloudPtr neo_ptcd_ptr;
neo_ptcd_ptr = open3d::io::CreatePointCloudFromFile(pcdname,
                                                    "pcd", true);

// show loaded point cloud
open3d::visualization::DrawGeometries({neo_ptcd_ptr}, "loaded point cloud");
```

3.2 Point Cloud Registration

Point cloud registration refers to registering two point clouds with similar shapes so that they overlap. From a computational perspective, it calculates a rigid transformation that can match the source point cloud to the target point cloud. An algorithm called ICP (Iterative Closest Points) is mainly used, and Open3D provides various registration algorithms.

1. Point-to-point ICP: Simply matches two point clouds to minimize the distance between points.
2. Point-to-plane ICP: Matches two point clouds to minimize the vertical distance between the source point and the target surface.
3. Colored ICP: This is an algorithm that converges more accurately and quickly by using color information when finding connections between points in ICP.
4. Global registration: The above ICP algorithms are local registration algorithms that can match two slightly different point clouds. However, if the poses of the two point clouds are significantly different, they cannot be matched through ICP. Global registration is an algorithm that can find global correspondence through feature matching and achieve registration regardless of the initial posture.

Here, we study examples using algorithms 2 and 3.

The links below are related Python tutorial links.

- ICP registration: http://www.open3d.org/docs/release/tutorial/Basic/icp_registration.html
- Colored point cloud registration: http://www.open3d.org/docs/release/tutorial/Advanced/colored_pointcloud_registration.html
- RGBD odometry: http://www.open3d.org/docs/release/tutorial/Basic/rgbd_odometry.html
- Global registration: http://www.open3d.org/docs/release/tutorial/Advanced/global_registration.html

3.2.0 Registration Parameters

The parameters required for registration `RegPar`have been entered in advance into the namespace.

```
// registration_examples.cpp
namespace RegPar
{
const double depth_scale = 5000.0;  // from TUM RGBD format
const double depth_trunc = 4.0;
const double max_correspondence_dist = 0.3;
const Eigen::Matrix4d init_pose = Eigen::Matrix4d::Identity();
const open3d::geometry::KDTreeSearchParamHybrid kdtree(0.01, 20);
const open3d::camera::PinholeCameraIntrinsic intrinsic(
        open3d::camera::PinholeCameraIntrinsicParameters::PrimeSenseDefault);
const open3d::registration::ICPConvergenceCriteria convergence(
        1e-6, 1e-6, 20);
}
```

- depth_scale: The value divided when converting integer depth to real depth, set to 5000 to create a meter unit.
- depth_trunc: Maximum depth converted to point cloud, depth beyond this is not used.
- max_correspondence_dist: Maximum distance range to find matches between points
- init_pose: Starting point of transformation calculation, usually 4x4 identity matrix.
- kdtree: Used for kd-tree objects, normal estimation, etc. The constructor argument indicates the size of the voxel grid and the maximum number of searches.
- intrinsic: An object that stores the intrinsic parameters of the camera.
- convergence: An object that stores the condition to stop repetition. The argument of the constructor indicates the reduction rate and size of the error to stop repetition, and the maximum number of repetitions.

3.2.1 Point-to-plane ICP

When you click “ICP point-to-plane” in the GUI, the following function is executed.

```
    RegistrationExamples::IcpPointCloud("../samples/depth1.png",
                                        "../samples/depth2.png");
```

a. Read rgb-d files

`CreateImageFromFile()`Read the two depth images.

```
void RegistrationExamples::IcpPointCloud(const char* srcdepthfile,
                                         const char* tgtdepthfile)
{
        // read
    o3ImagePtr src_depth = open3d::io::CreateImageFromFile(srcdepthfile);
    o3ImagePtr tgt_depth = open3d::io::CreateImageFromFile(tgtdepthfile);
    if(src_depth->IsEmpty() || tgt_depth->IsEmpty())
    {
        open3d::utility::LogError("Failed to read {} or {}", srcdepthfile, tgtdepthfile);
        return;
    }
```

b. Convert to point cloud

As was done in 3.1.1, the depth image is converted to a point cloud, but the process of converting to FloatDepth is omitted and converted directly to a point cloud. At this time , you must enter both the input parameters required for conversion to FloatDepth `RegPar::depth_scale, RegPar::depth_trunc`and the camera intrinsic parameters and initial pose relationship required for conversion to point cloud .`RegPar::intrinsic, RegPar::init_pose`

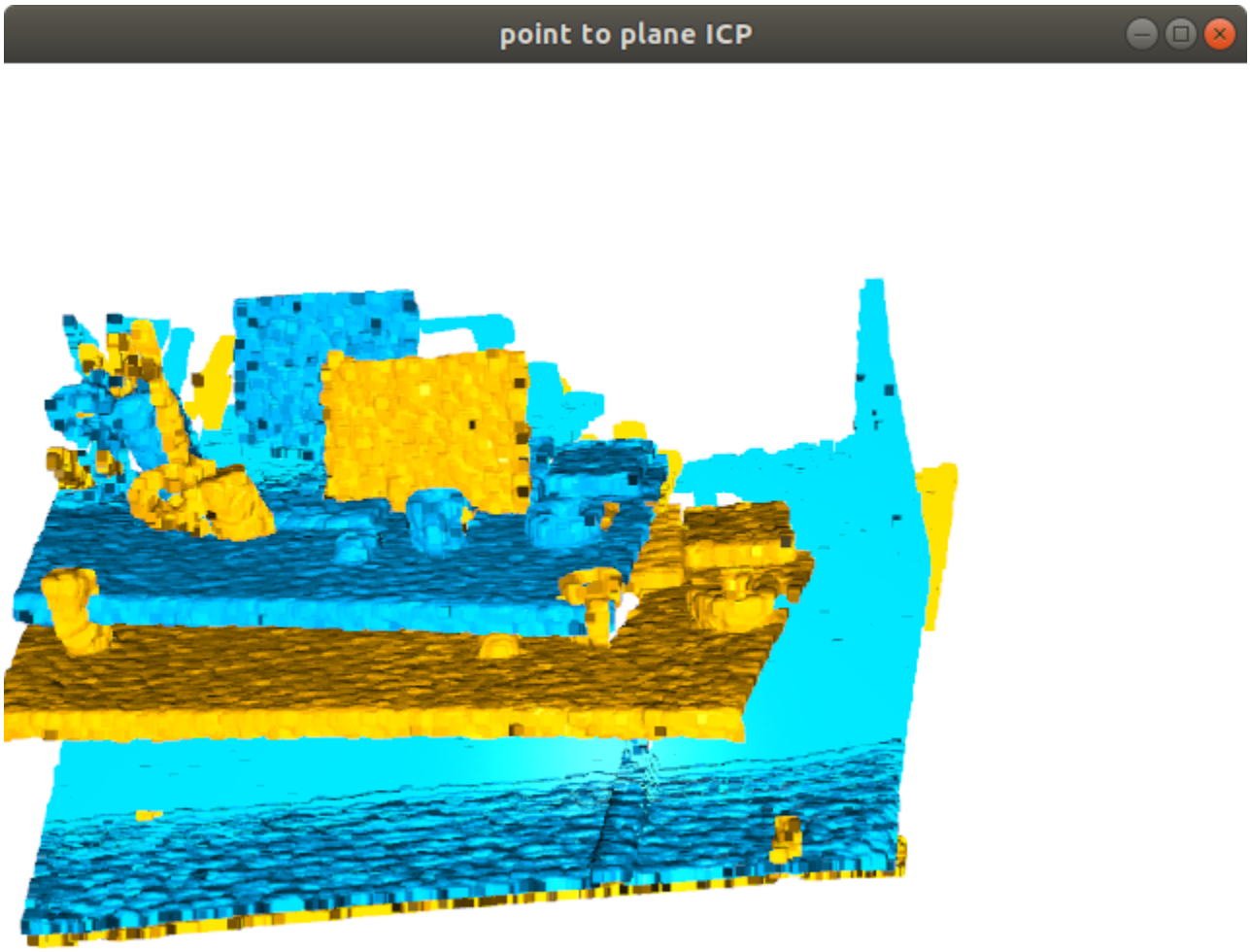
```
    // convert depth to point cloud
    o3PointCloudPtr src_cloud = o3PointCloud::CreateFromDepthImage(
        *src_depth, RegPar::intrinsic, RegPar::init_pose,
        RegPar::depth_scale, RegPar::depth_trunc);
    o3PointCloudPtr tgt_cloud = o3PointCloud::CreateFromDepthImage(
        *tgt_depth, RegPar::intrinsic, RegPar::init_pose,
        RegPar::depth_scale, RegPar::depth_trunc);
    // estimate normal
    src_cloud->EstimateNormals(RegPar::kdtree, true);
    tgt_cloud->EstimateNormals(RegPar::kdtree, true);
    // draw two point clouds
    ShowTwoPointClouds(src_cloud, tgt_cloud,
        RegPar::init_pose, "point to plane ICP");
```

Afterwards `ShowTwoPointClouds()`, compare the two point clouds by displaying them on the screen. If there is no color information, the specified color is painted. (`PaintUniformColor()`) In order to preserve the original data, you need to deep copy PointCloud data, but there is currently no such function. So, I cropped a wide area to make a copy of the original.

`ShowTwoPointClouds()`The function converts `source`the coordinates of `transform`to input parameters into a coordinate system and displays them on the screen. Now that we have entered the identity matrix, we see two point clouds that are not registered.

```
void RegistrationExamples::ShowTwoPointClouds(o3PointCloudPtr source,
                o3PointCloudPtr target, Eigen::Matrix4d_u transform,
        const std::string title)
{
    // since there is no deep copy function, use crop instead
    Eigen::Vector3d range(10000.0, 10000.0, 10000.0);
    o3PointCloudPtr source_tmp = source->Crop(-range, range);
    o3PointCloudPtr target_tmp = target->Crop(-range, range);
    // set uniform color to point cloud
    if(!source_tmp->HasColors())
        source_tmp->PaintUniformColor(Eigen::Vector3d(1, 0.7, 0));
    if(!target_tmp->HasColors())
        target_tmp->PaintUniformColor(Eigen::Vector3d(0, 0.7, 1));
    source_tmp->Transform(transform);
    open3d::visualization::DrawGeometries({source_tmp, target_tmp}, title);
}
```

Comparing the two point clouds, the target (light blue) appears to have moved to the left, up, and back compared to the source (yellow). If you interpret this as coordinates, the source must move in the -x, -y, and +z directions to attach to the target.



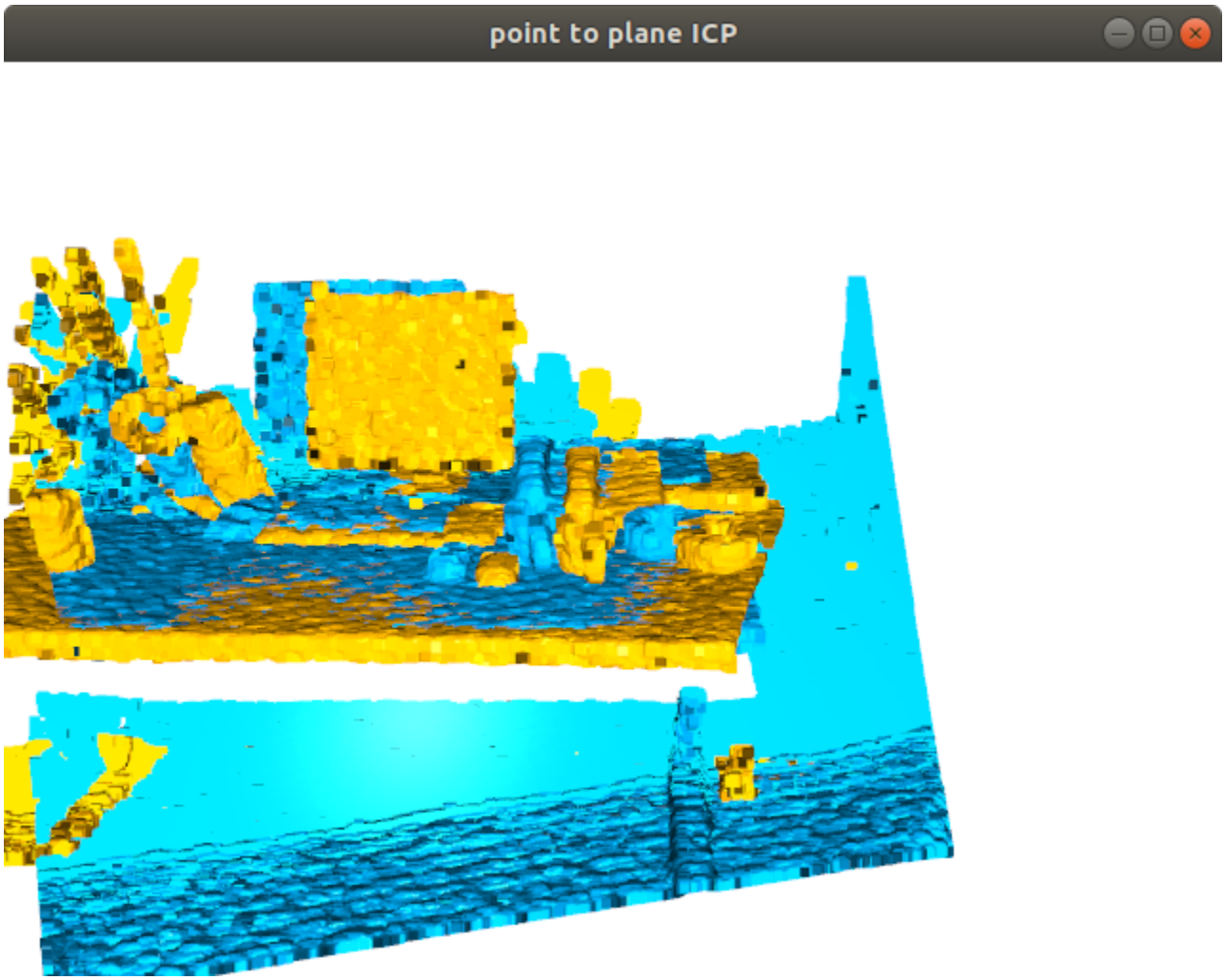
c. Run ICP

The ICP algorithm `open3d::registration::RegistrationICP`is implemented in . There are six input parameters, the first two are point clouds and most of the rest `RegPar`are pre-specified values `point_to_plane`but are new values. The 5th factor sets the distance to be minimized in ICP. If you use this, `TransformationEstimationPointToPoint`it becomes point-to-point ICP, and `TransformationEstimationPointToPlane`if you use like the code below, it becomes point-to-plane ICP.

The result is `open3d::registration::RegistrationResult result`displayed as , which contains information such as fitness, transformation, and correspondence.

```
// point-to-plane ICP
open3d::registration::TransformationEstimationPointToPlane point_to_plane;
open3d::registration::RegistrationResult result =
            open3d::registration::RegistrationICP(
                    *src_cloud, *tgt_cloud,
                    RegPar::max_correspondence_dist,
                    RegPar::init_pose, point_to_plane, RegPar::convergence);

open3d::utility::LogInfo("point-to-plane ICP result: \n"
                         "fitness={}, transformation=\n{}\n",
                         result.fitness_, result.transformation_);

// draw registration result
ShowTwoPointClouds(src_cloud, tgt_cloud, result.transformation_,
                   "point to plane ICP");
```

If you look at the results, you can see that it sticks well front and back, top and bottom, but less so on the left and right. This is because there are few reference surfaces (walls on both sides, etc.) that ICP can align the left and right with, so this result can occur even if the algorithm operates normally.



3.2.2 Colored ICP

When you click “ICP with color” in the GUI, the following function is executed. This time, ICP is performed by reading a PCD format file that stores a point cloud rather than an rgb-depth image. For this purpose `RegistrationExamples::RgbDepthToPCD()`, we implemented a function that converts rgb and depth images into point clouds and saves them . After this function is executed, the saved PCD file is read and colored ICP is executed.

```
void MainWindow::on_pushButton_icp_colored_clicked()
{
   if(!open3d::utility::filesystem::FileExists("../results/ptcloud1.pcd"))
        RegistrationExamples::RgbDepthToPCD("../samples/color1.png",
                                "../samples/depth1.png", "../results/ptcloud1.pcd");
   if(!open3d::utility::filesystem::FileExists("../results/ptcloud2.pcd"))
        RegistrationExamples::RgbDepthToPCD("../samples/color2.png",
                                "../samples/depth2.png", "../results/ptcloud2.pcd");
    RegistrationExamples::IcpColoredPointCloud("../results/ptcloud1.pcd",
                                               "../results/ptcloud2.pcd");
}
```

a. Read pcd files

Read the two point cloud files saved earlier and display them on the screen.

```
void RegistrationExamples::IcpColoredPointCloud(const char* srcpcdfile,
                                                const char* tgtpcdfile)
{
    // read
    o3PointCloudPtr src_cloud = open3d::io::CreatePointCloudFromFile(srcpcdfile);
    o3PointCloudPtr tgt_cloud = open3d::io::CreatePointCloudFromFile(tgtpcdfile);
    if(src_cloud->IsEmpty() || tgt_cloud->IsEmpty())
    {
        open3d::utility::LogError("Failed to read {} or {}", srcpcdfile, tgtpcdfile);
        return;
    }
    // draw initial state
    ShowTwoPointClouds(src_cloud, tgt_cloud,
                                       Eigen::Matrix4d_u::Identity(), "Before colored ICP");
```

b. Run colored ICP

`open3d::registration::RegistrationColoredICP`A more accurate transformation can be calculated using RGB information. Input arguments include two point clouds and `RegPar`variables defined in . The results are also presented in the same `open3d::registration::RegistrationResult result`format as point-to-plane ICP.

```
// colored ICP
open3d::registration::RegistrationResult result =
        open3d::registration::RegistrationColoredICP(
                            *src_cloud, *tgt_cloud,
                            RegPar::max_correspondence_dist,
                            RegPar::init_pose, RegPar::convergence);
open3d::utility::LogInfo("colored ICP result: \n"
                         "fitness={}, transformation=\n{}\n",
                         result.fitness_, result.transformation_);
// draw results
ShowTwoPointClouds(src_cloud, tgt_cloud,
                   result.transformation_, "After colored ICP");
```

If you look at the two matched point clouds, you can see that they overlap so well that the two are almost indistinguishable.

4. Other Useful Funcitons

Useful algorithms that were not introduced above are introduced through Python tutorials.

1. Multiway registration: Calculate the pose of each point cloud that can best match multiple point clouds through pose-graph optimization. In short, do SLAM.

   http://www.open3d.org/docs/release/tutorial/Advanced/multiway_registration.html

2. Point cloud outlier removal: Literally removes outliers from the point cloud.

   http://www.open3d.org/docs/release/tutorial/Advanced/pointcloud_outlier_removal.html

3. RGBD integration: Using TSDF technology, consecutive RGBD images and the camera pose at the time are collected to create a TSDFVolume. You can also convert this into a simpler mesh form and save it.

   http://www.open3d.org/docs/release/tutorial/Advanced/rgbd_integration.html

4. PointNet++: A project that processes point clouds based on Open3D and trains PointNet++ with Tensorflow has been released. There is code, but unfortunately, the already trained model is not visible.

   https://github.com/intel-isl/Open3D-PointNet2-Semantic3D

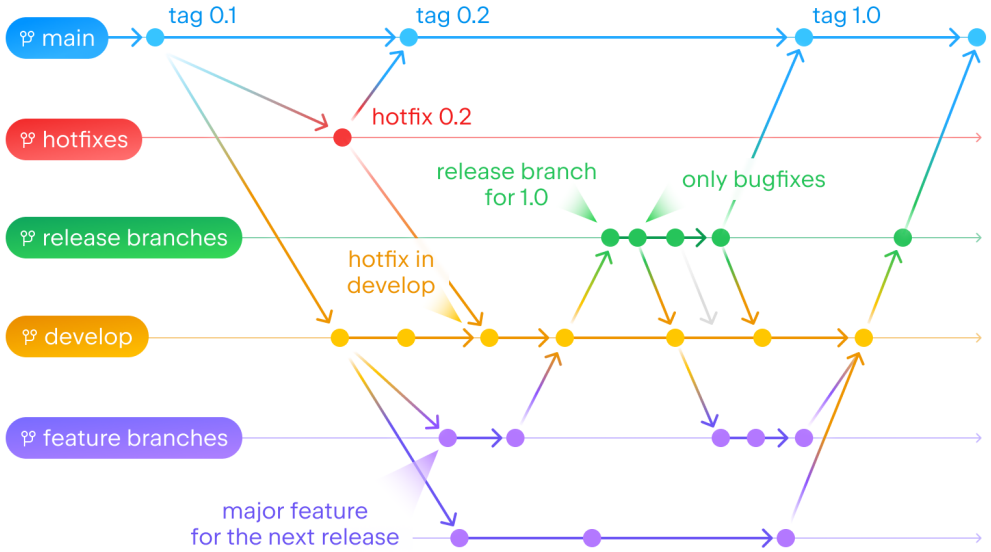
Other Libraries

- CCV: Implements many modern vision algorithms (Predator, DPMs,…)
- VLFeat: Mature and well-documented vision library, includes MATLAB bindings.
- LibCVD: Includes an optimized FAST corner detector implementation.
- Sophus: Lie groups library for Eigen. Useful for SLAM / Visual Odometry

Git

- Distributed version control system for tracking changes in code
- Widely used in the software development industry, & considered the standard version control system for managing source code
- Many web-based platforms provide hosting services for Git repositories (e.g., GitHub, GitLab)
- **Good practices**:
  - Keep code & data separate (only commit code, unless data is very small)
  - Commit small changes
  - Use descriptive commit messages
  - Use branches

Git flow



- The `main` branch is for production code only.
- The `develop` branch is for development code.
- `feature` branches are created from the `develop` branch.
- `hotfix` branches are created from the `main` branch.
- `release` branches are created from the `develop` branch.

Git

- Introduction slides from
  - https://course.ccs.neu.edu/cs5010f15/Slides/Lesson%200.5%20Introduction%20to%20Git.pptx
  - Sumner Evans, Mines ACM, 2023

Git is a distributed version-control system

- You keep your files in a *repository* on your local machine.
- You synchronize your repository with a repository on a server.
- If you move from one machine to another, you can pick up the changes by synchronizing with the server.
- If your partner uploads some changes to your files, you can pick those up by synchronizing with the server.

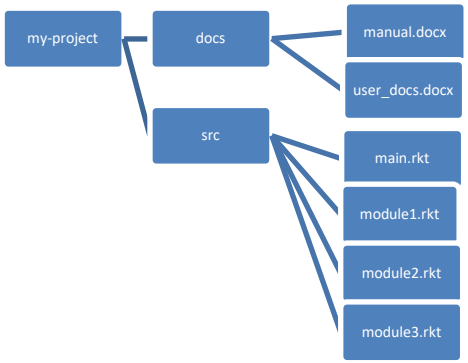
Git is a distributed version-control system

- Terminology: In git-speak, a “version” is called a “commit.”
- Git keeps track of the history of your commits, so you can go back and look at earlier versions, or just give up on the current version and go back some earlier version.

A simple model of git

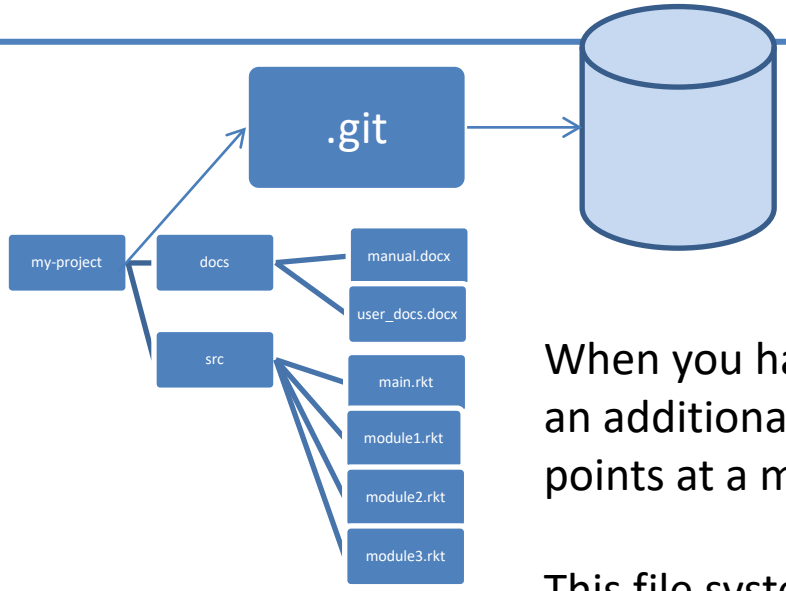
- Most git documentation gets into details very quickly.
- Here's a very simple model of what's going on in git.

Your files



Here are your files, sitting in a directory called my-project

Your files in your git repository

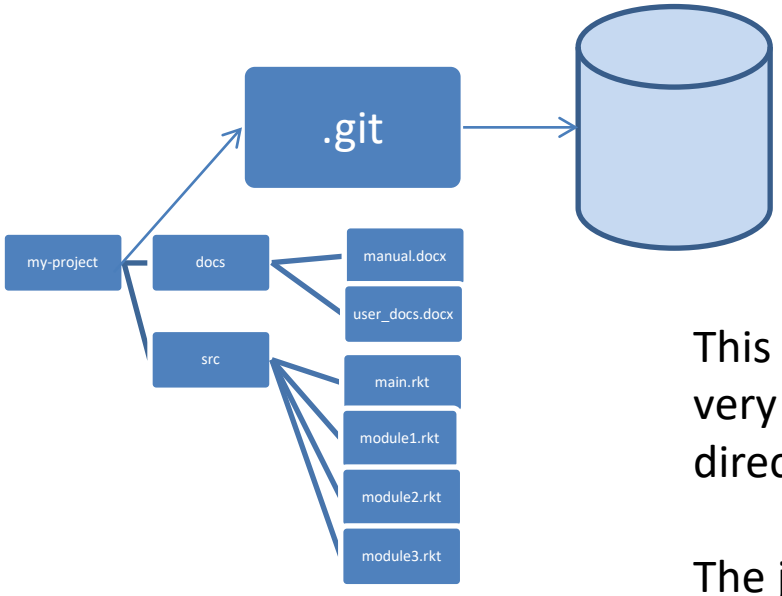


When you have a git repository, you have an additional directory called .git, which points at a mini-filesystem.

This file system keeps all your data, plus the bells and whistles that git needs to do its job.

All this sits on your local machine.

The git client



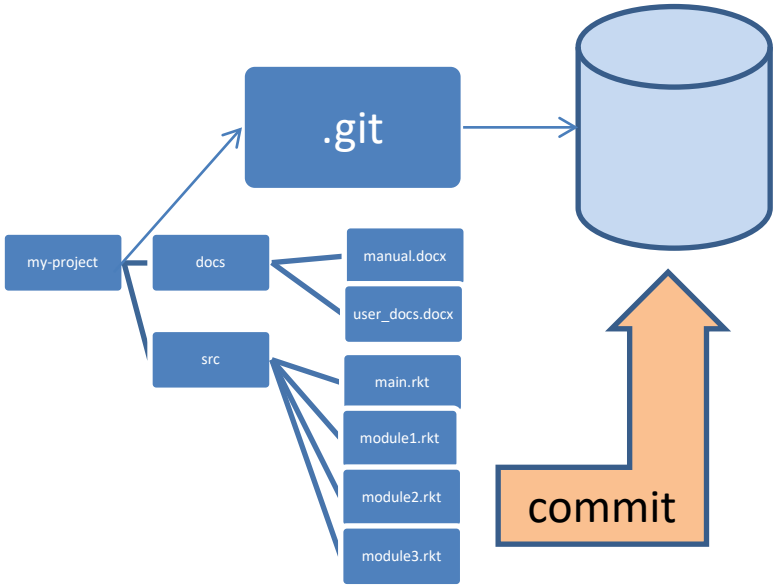
This mini-filesystem is highly optimized and very complicated. Don't try to read it directly.

The job of the git client (either Github for Windows, Github for Mac, or a suite of command-line utilities) is to manage this for you.

Your workflow (part 1)

- You edit your local files directly.
  - You can edit, add files, delete files, etc., using whatever tools you like.
  - This doesn't change the mini-filesystem, so now your mini-fs is behind.

A Commit



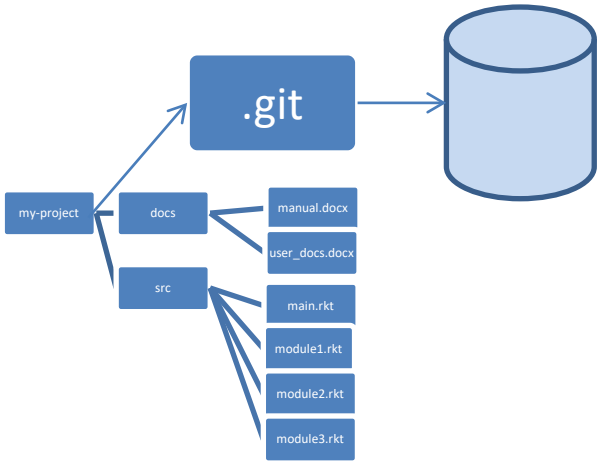
When you do a “commit”, you record all your local changes into the mini-fs.

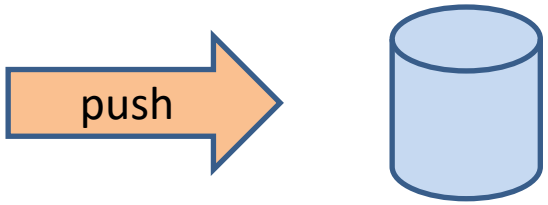
The mini-fs is “append-only”. Nothing is ever over-written there, so everything you ever commit can be recovered.

Synchronizing with the server (1)

your local machine

a server, somewhere on the internet, eg. github.com



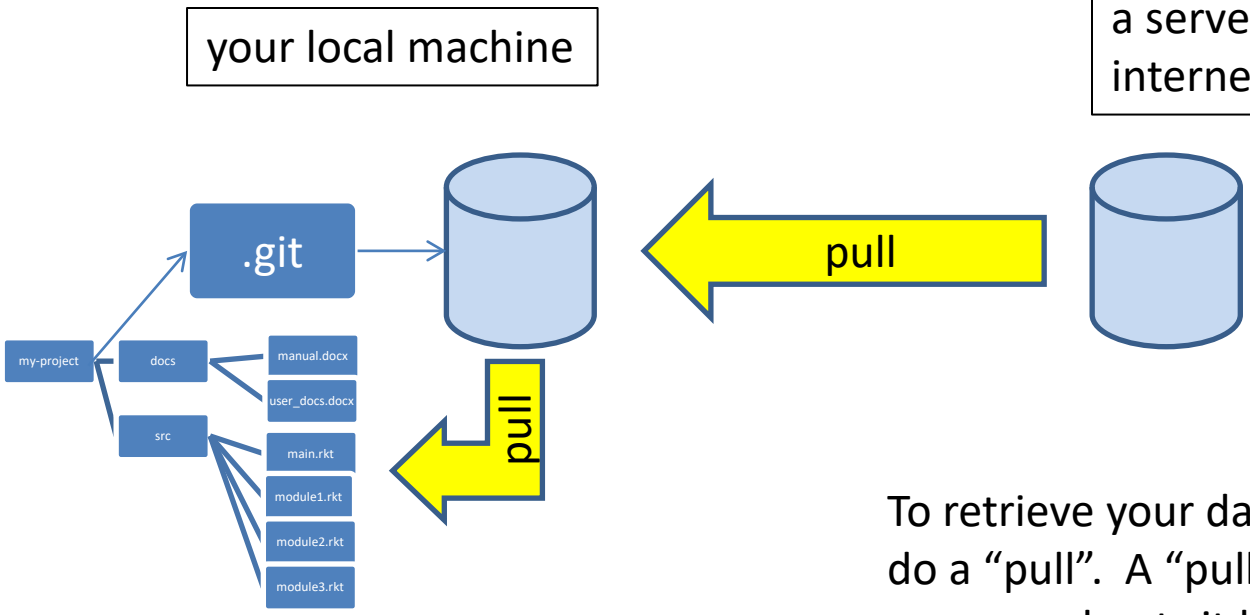


At the end of each work session, you need to save your changes on the server. This is called a "push".

Now all your data is backed up.

- You can retrieve it, on your machine or some other machine.
- We can retrieve it

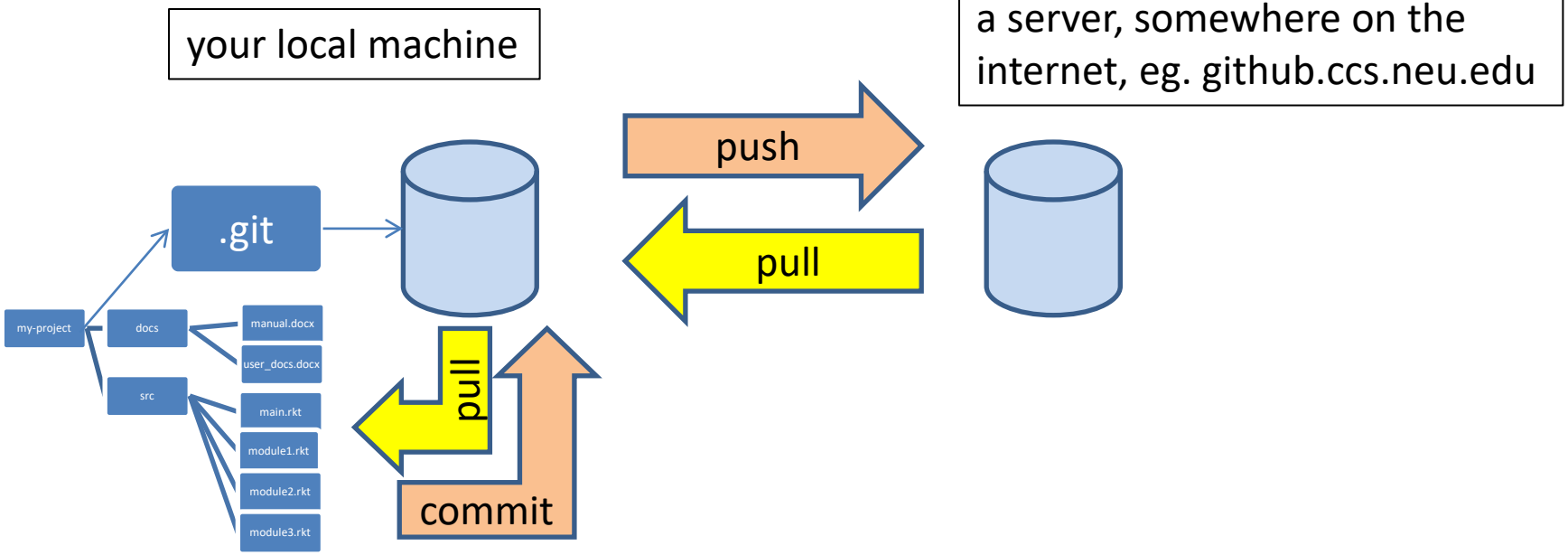
Synchronizing with the server (2)



To retrieve your data from the server, you do a “pull”. A “pull” takes the data from the server and puts it both in your local mini-fs and in your ordinary files.

If your local file has changed, git will merge the changes if possible. If it can’t figure out how to the merge, you will get an error message. We'll learn how to deal with these in the next lesson.

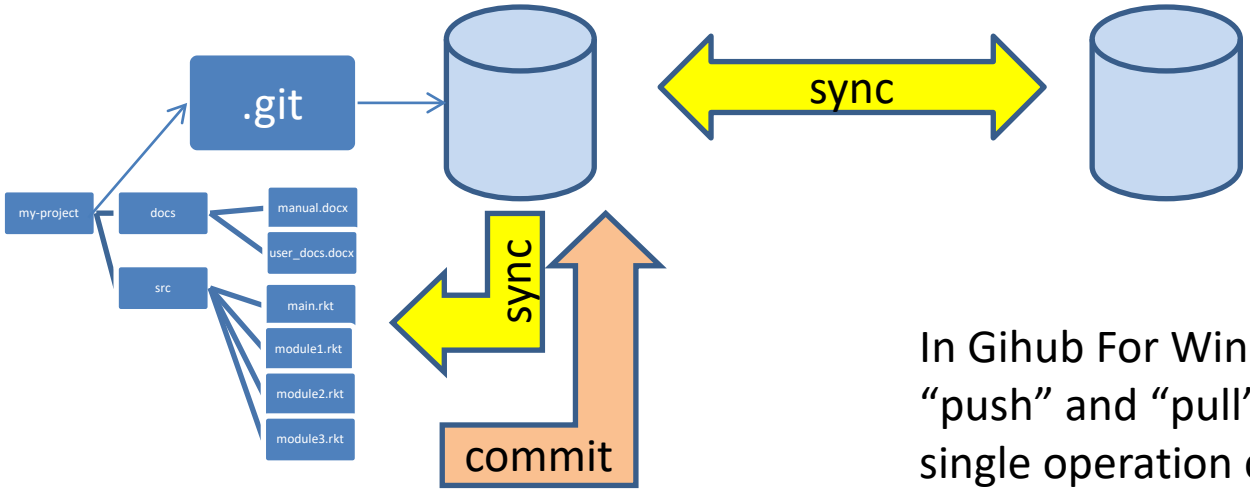
The whole picture



The whole picture using GHFW

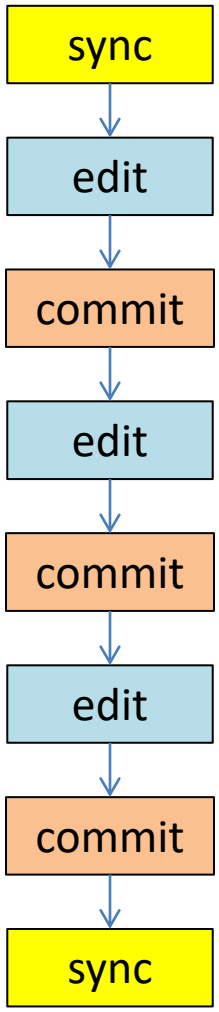
your local machine

a server, somewhere on the internet, eg. github.ccs.neu.edu



In Gihub For Windows or Github For Mac, “push” and “pull” are combined into a single operation called “sync”. So in these clients, there are only two steps (“commit” and “sync”) to worry about, not three.

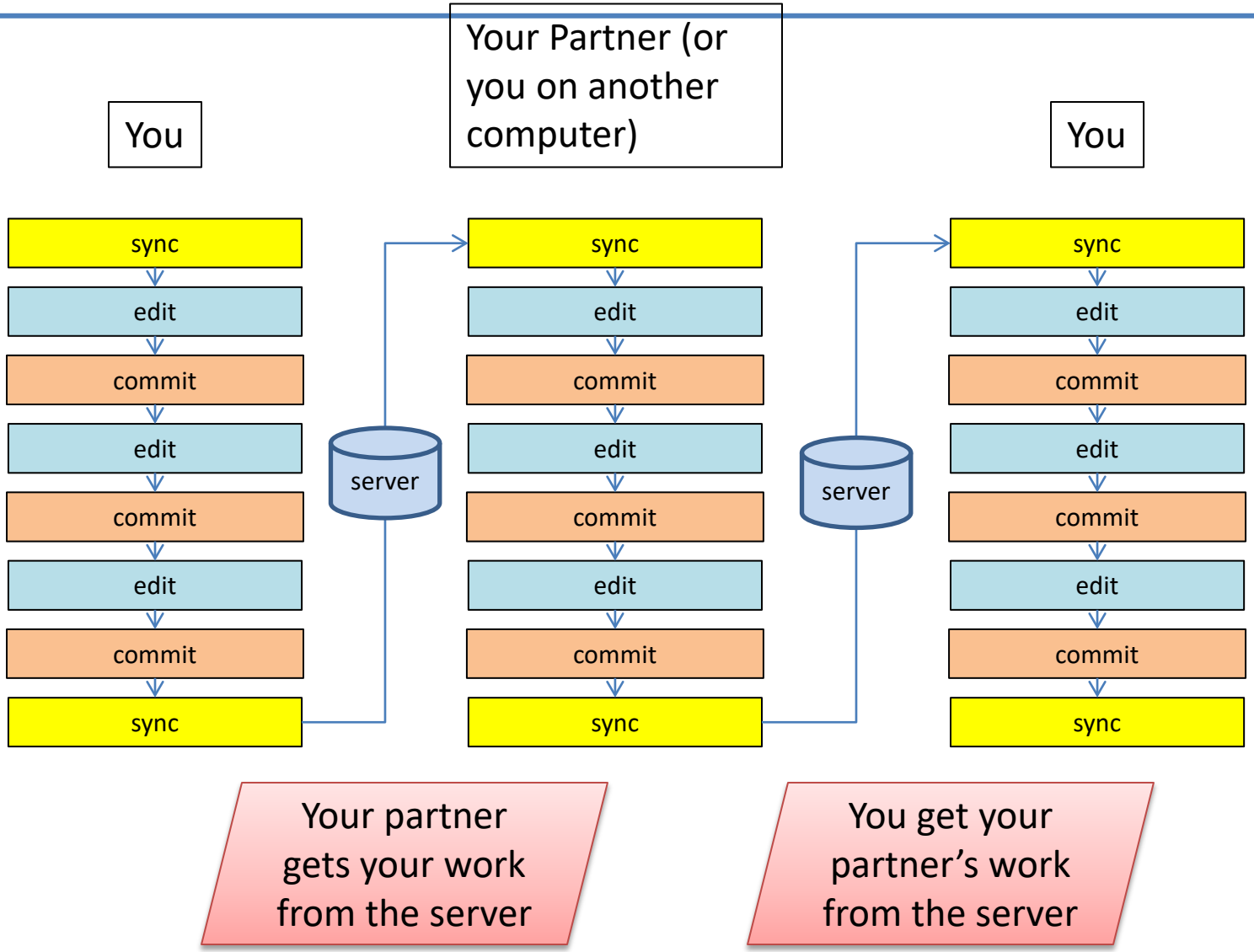
Your workflow (2)



Best practice: commit your work whenever you've gotten one part of your problem working, or before trying something that might fail.

If your new stuff is screwed up, you can always "revert" to your last good commit. (Remember: always "revert", never "roll back")

Your workflow with a partner



Overview

1. Why use Git?
2. Commits
3. Branches
4. Merging
5. Rebasing
6. Remotes
7. Advanced Tips

This talk is interactive!

If you have questions at any point, feel free to interrupt me.

Why use Git?

Why use Version Control? I

Example Scenario:

1. You start a project called ”my-proj” and write a ton of code.
2. You finally get it to (kinda) work.
3. You decide to make a copy of ”my-proj” for backup purposes.
4. You continue development on ”my-proj” but then screw something up really bad.
5. You decide to revert back to your copy.
6. Then you realize your copy doesn’t have a bug fix that you actually wanted.
7. You then proceed to manually compare the files in the backup to those in your new code and figure out what you still want to have.

This is terrible.

Why use Version Control? II

Another Scenario:

1. You start working on a project with a partner.
2. You write a bunch of code.
3. You email the code in a .zip file, then go home for the weekend.
4. You and your partner had decided to work on two separate tasks over the weekend so you make some changes to the code and your partner makes some changes to the code.
5. You come together and start copying files. Then you realize you both modified *`main()`*.
6. You then manually determine what changed in both files and reconcile them.

This is awful.

Version Control Systems

Version Control Systems (VCSs) such as Git solve these problems.

- VCS keeps track of *revisions*, changes in the code in entities called *changesets* or *commits*.
- Most VCS allow version merging. That means multiple people can be working on the same file and resolve discrepancies later. Git is very elegant in handling merge conflicts such as this.

Git is popular

Git is a very popular version control system.

Services such as GitHub and GitLab provide free hosting for Git repositories.

It has become the de-facto industry standard for source control.

Git is a distributed version control system

Distributed because you can use it without being connected to a central server. You have a full copy of the code on your own computer.

Version control because it keeps track of changes to files.

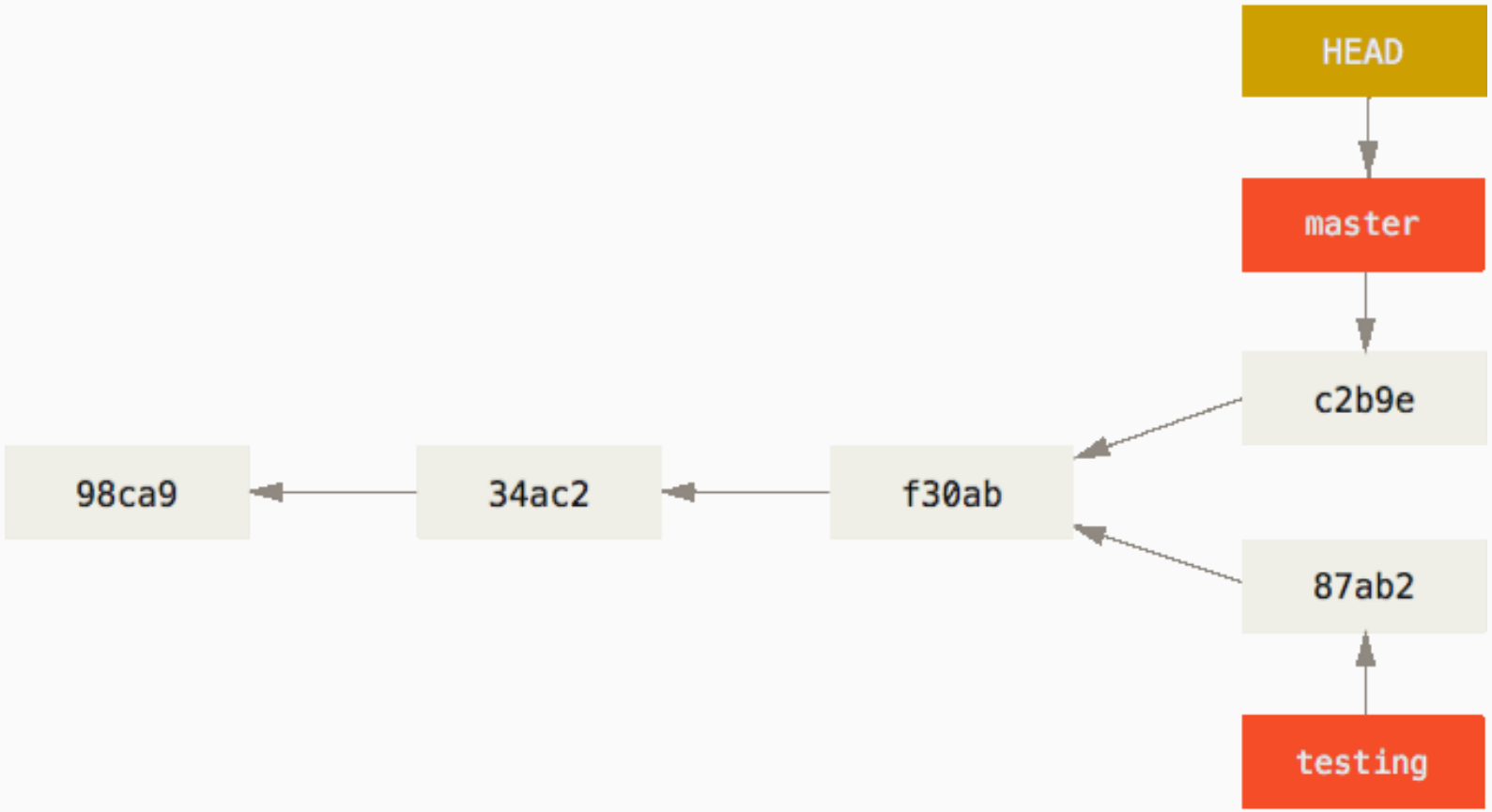
But how does it keep track of all of the changes?

Commits

Commits: what are they?

Commits are sets of differences (diffs) in files.[1]

Commits reference their parent(s) and contain information about the changes made in the repo since that parent commit.



[1]This is a bit of a lie, more on that later.

Commits: they form a DAG, and are stored on the “heap”

Commits form a Directed Acyclic Graph (DAG). There are no loops in the graph, and every commit points to its parent(s).

Every single commit is stored in the `.git` directory of your repository[2] which can be thought of like a “heap” for commits. The parents of a commit are also stored in this “heap”.

We will return to this fact when we talk about *branches*.

[2] Technically another lie.

Commits: creating

You can create a commit by running `git commit`. This will create a commit based on the checked-out commit with the changes that you have "staged".

- You can stage changes by running `git add` and passing a list of files.
- You can stage all changes by running `git add -A`.
- **Pro tip**: If you want to add specific parts of files, use `git add -p`.

If you want to remove from the stage, you can run `git reset` (optionally passing a list of files to unstage).

Commits: what will be committed?

If you ever want to know what will be included in your commit, you can run *`git status`* to show the list of files staged for commit.

```
> git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   git.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
  ↪  directory)
        modified:   git.pdf
        modified:   git.tex
```


The *`git.tex`* file has only some lines staged for commit.

Commits: what will be committed, but with more detail?

To see the details of the changes that are staged (that is, will be committed), you can run *`git diff --cached`*.

If you want to see the details of the changes that are *not* staged, you can run *`git diff`*.

`git diff` optionally accepts a list of files to diff.

```
diff --git a/git.tex b/git.tex
index 2c01a7b..91148d1 100644
--- a/git.tex
+++ b/git.tex
@@ -33,9 +33,7 @@

 \section{Why use Git?}

-\begin{frame}{Why use Git? I}
-
-    Example Scenario:
+\begin{frame}{Example Scenario 1}

     \begin{enumerate}[<+->]
         \item You start a project called ``my-proj'' and write a ton of code.
```

Commits: showing existing commits

You can view a commit using *`git show <commit hash>`*.

```
commit cb1b9610e4d34aa66b52e7fb722654679b4529e2
Author: Sumner Evans <me@sumnerevans.com>
Date:   Tue Jan 31 10:42:58 2023 -0700

    matrix-synapse: 1.76.0rc2 -> 1.76.0

    Signed-off-by: Sumner Evans <me@sumnerevans.com>

diff --git a/modules/services/matrix/synapse/default.nix
↪  b/modules/services/matrix/synapse/default.nix
index b142cf0..675ab77 100644
--- a/modules/services/matrix/synapse/default.nix
+++ b/modules/services/matrix/synapse/default.nix
@@ -7,20 +7,20 @@ let
   # Custom package that tracks with the latest release of Synapse.
   package = pkgs.matrix-synapse.overridePythonAttrs (old: rec {
     pname = "matrix-synapse";
-    version = "1.76.0rc2";
+    version = "1.76.0";
     format = "pyproject";

     src = pkgs.fetchFromGitHub {
       owner = "matrix-org";
       repo = "synapse";

...
```

Commits: summary

Use `git add` and `git reset` (or variants) to stage/unstage changes for your.

Use `git commit` to create a commit from the currently staged changes (which you can see by running `git diff --cached`).

Branches

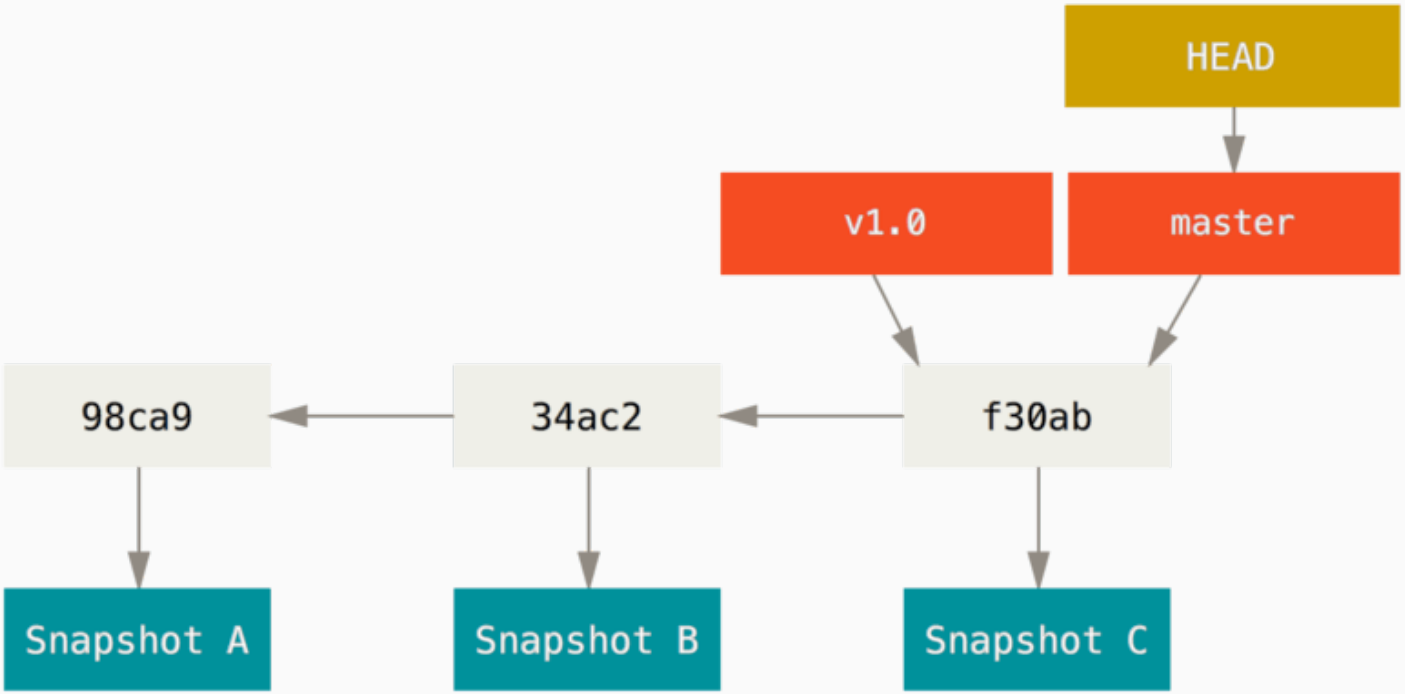
Branches: what are they?[3]

Remember how we said that the `.git` directory is like a "heap" for commits?

Branches are pointers to a specific commit in that heap. You can then follow that commit's parent pointers to reconstruct the graph.

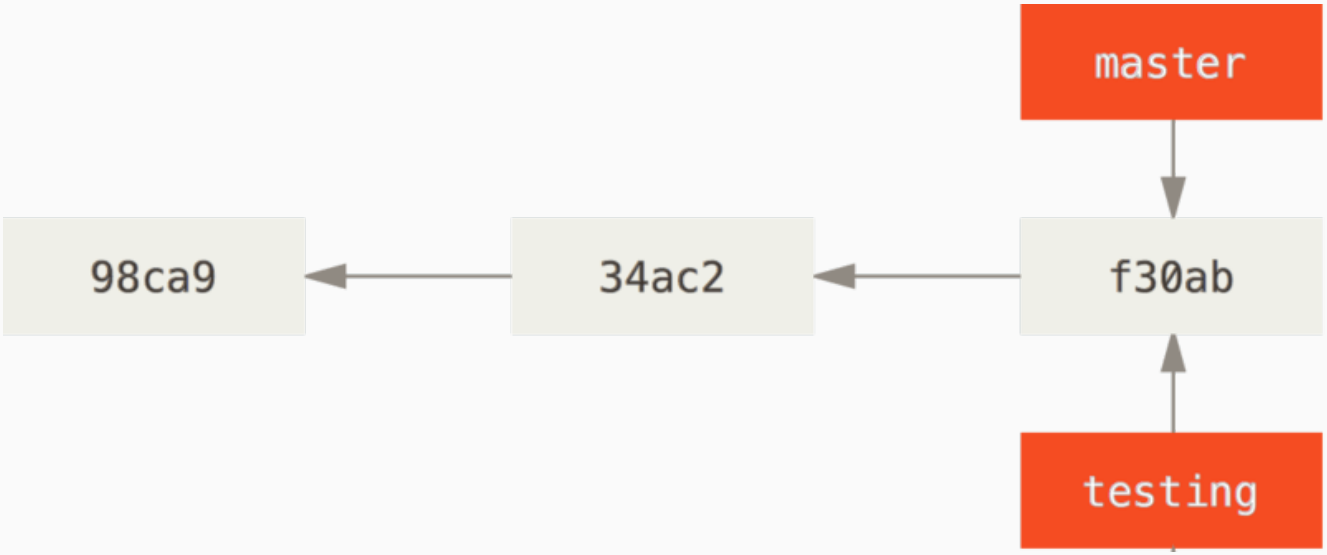
[3] Info in the rest of the *Branches* section is mainly from *https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell*

Branches: pointers



Branches: creating them

Branches can be created using the *`git branch <branch name>`* command. **This will not change your HEAD pointer.**



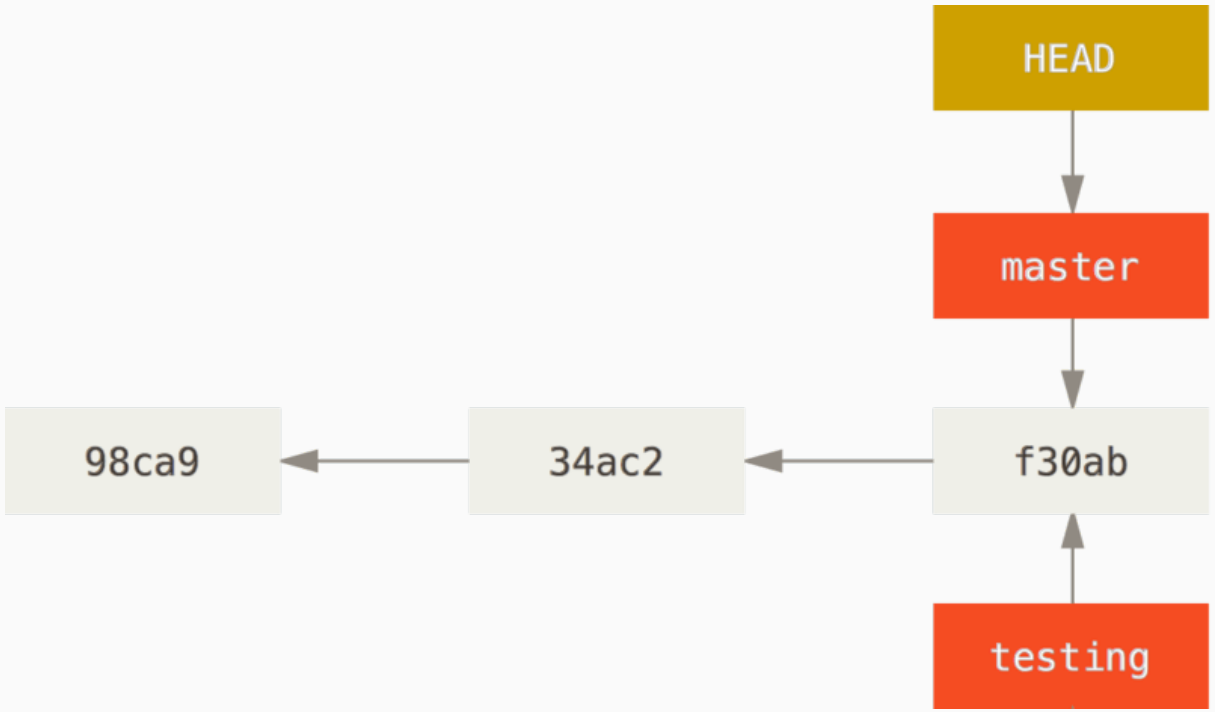
If you want to create a branch and also change the *HEAD* pointer to the newly created branch, you can use: *`git checkout -b <branch name>`*.

You can use *`git branch [-a]`* to list (all) branches.

Branches: moving HEAD around

HEAD is a special pointer to the current repository state. Checking out a commit/branch will update the files in your working directory.

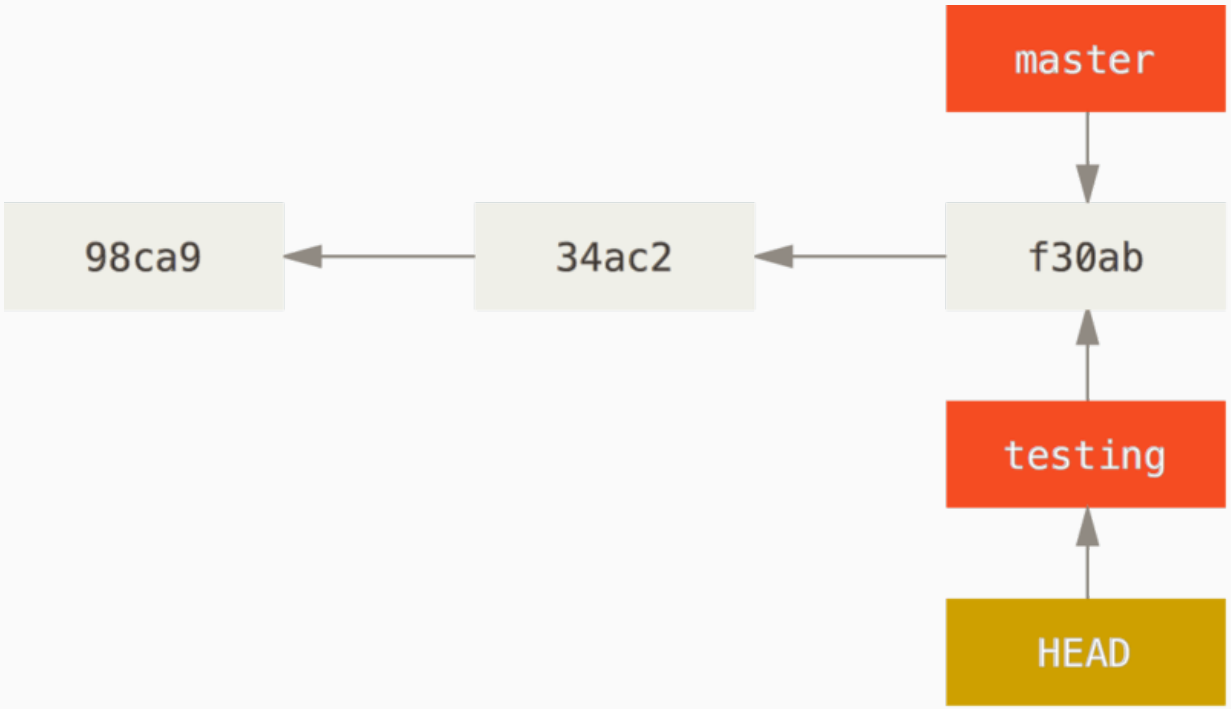
You can move the *HEAD* pointer to a different commit using *`git checkout <commit hash or branch name>`*.



Branches: moving HEAD around

HEAD is a special pointer to the current repository state. Checking out a commit/branch will update the files in your working directory.

You can move the *HEAD* pointer to a different commit using *`git checkout <commit hash or branch name>`*.



Branches: moving other branches around

If you want to move the branch that *HEAD* is pointing to to a different location, you can use *git reset*.

git reset --hard <commit hash> will move the branch that *HEAD* is pointing to to the specified commit, **discarding all changes in your working directory**.

git reset --soft <commit hash> will move the branch that *HEAD* is pointing to to the specified commit, **leaving all changes since the specified commit as staged changes in your working directory**.

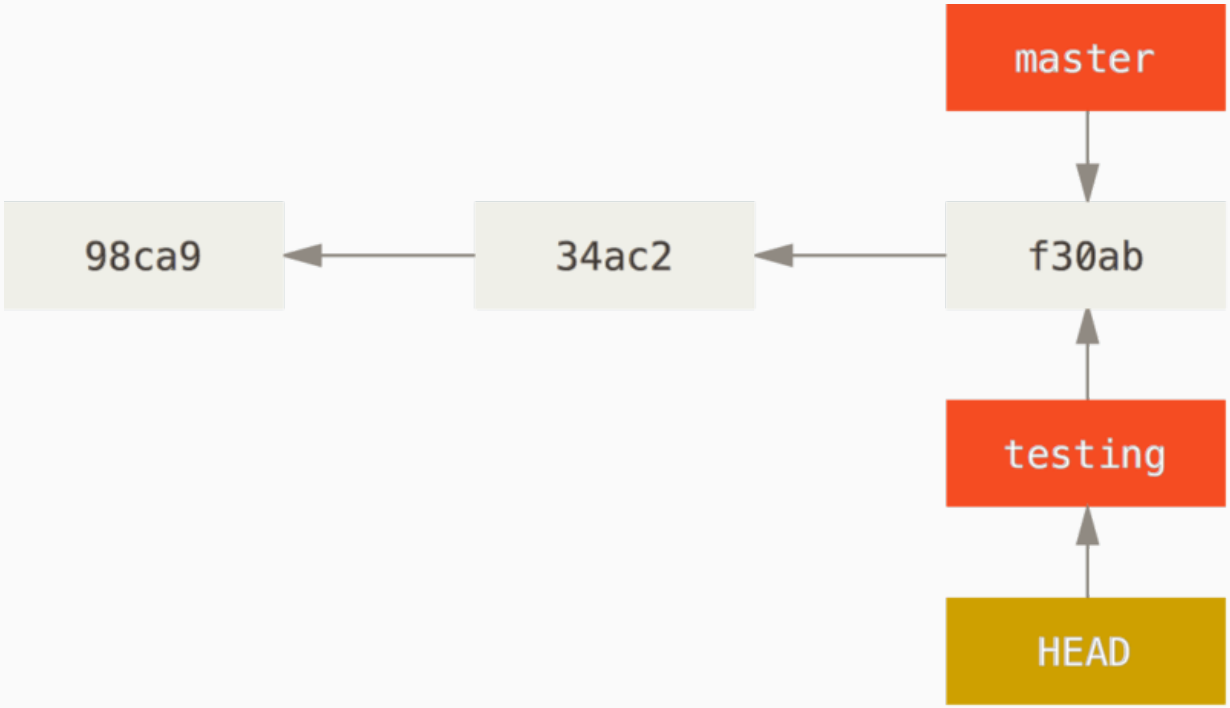
Branches: `checkout` gotchas and pro-tips

- If you checkout a commit hash, you will be in a *`detached HEAD`* state because your *`HEAD`* pointer is not pointing to a branch.
- If you have uncommitted changes, switching branches *might* fail.
  You can use *`git stash`* to save the changes in your working directory, then *`checkout`* the other branch, and then *`git stash pop`* to restore the changes.
  Alternatively, you can just create a WIP commit and then switch to the other branch.

Pro tip: You can use `-` to refer to the previously checked out object.

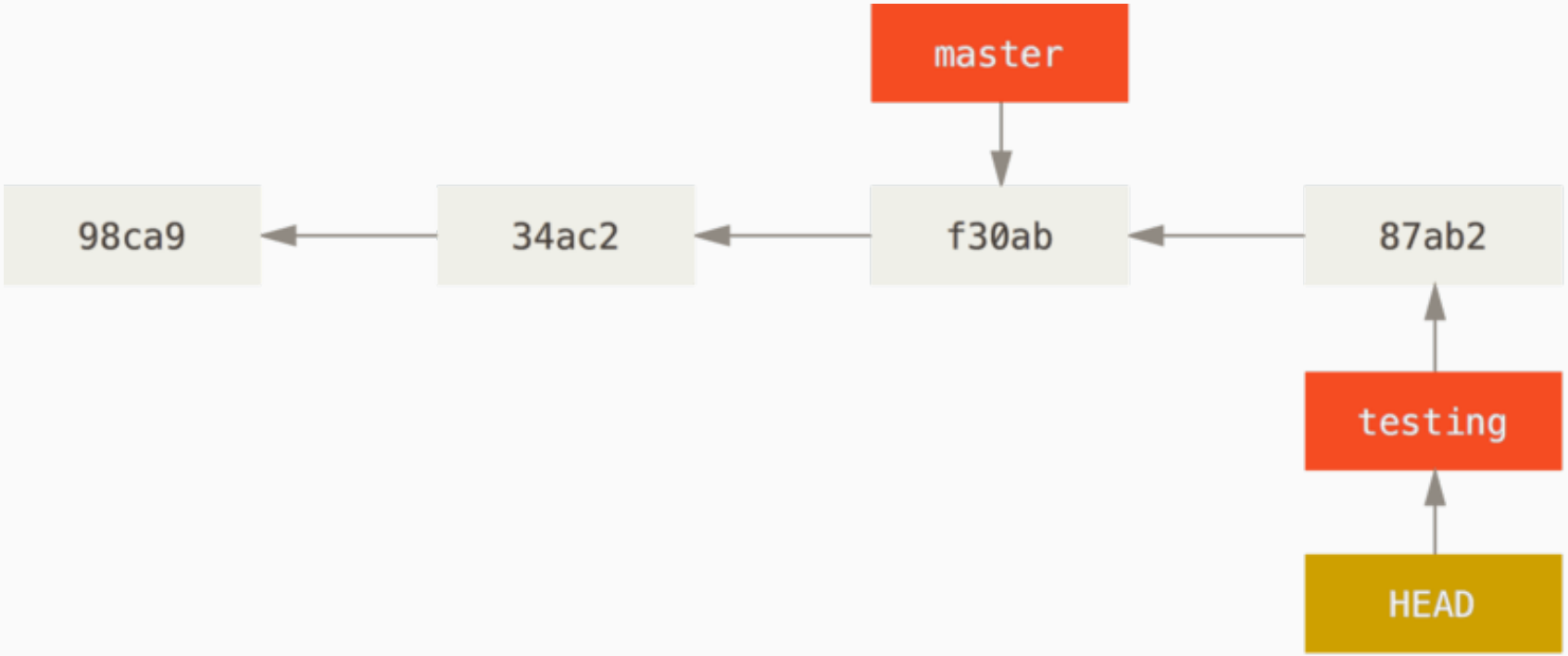
Branches: making commits

If you commit something while *HEAD* is pointed to a branch, both *HEAD* and your branch will move to the new commit.



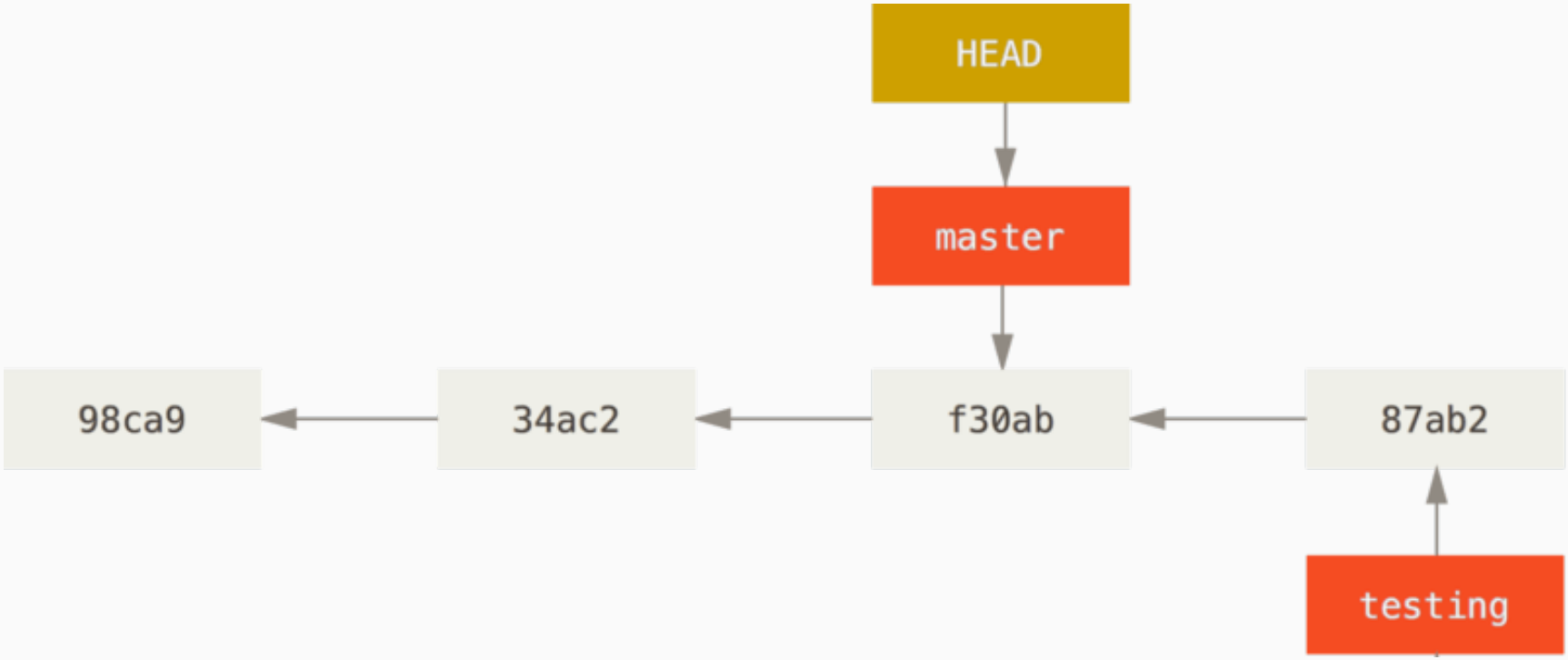
Branches: making commits

If you commit something while *HEAD* is pointed to a branch, both *HEAD* and your branch will move to the new commit.



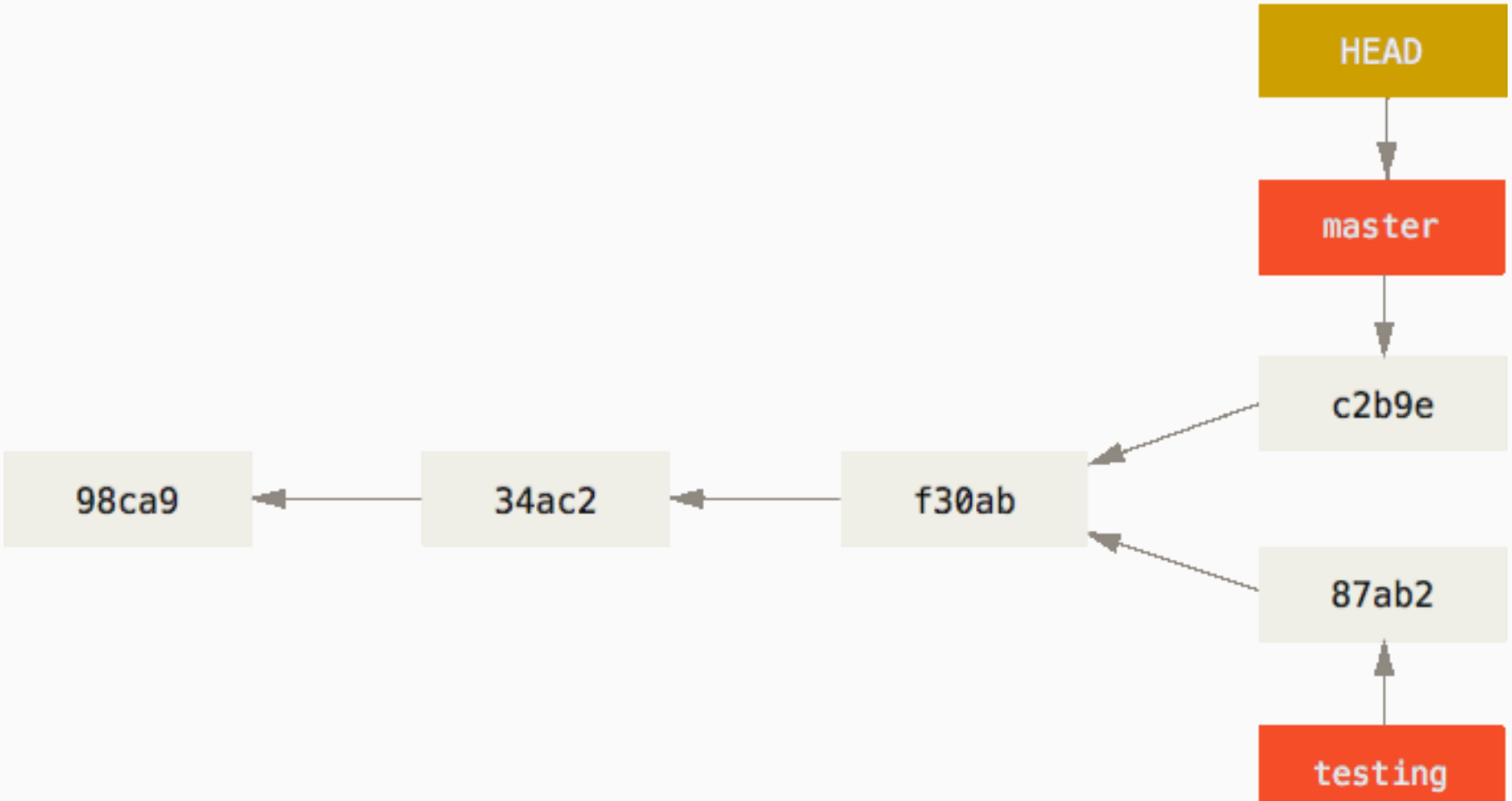
Branches: using multiple branches

Of course, you can always switch back to *`master`* using *`git checkout master`*.



Branches: divergence

If you make a commit on the master branch, the *`master`* pointer moves to that new commit creating **divergent** branch histories.



Branches: where am I?

Often, you want to get a summary of where you are in the repository. That's where ***`git log`*** comes in.

```
> git log
commit b08107c144003ba42495995d59234595d2d875b4 (HEAD -> master, origin/master,
↪  origin/HEAD)
Author: Sumner Evans <me@sumnerevans.com>
Date:   Mon Feb 13 14:35:57 2023 -0700

    fix some things

    Signed-off-by: Sumner Evans <me@sumnerevans.com>

commit 6c1f8b53ac774dc6b0376810b4745951bd572519
Merge: 47bc626 67f0f4d
Author: Ethan Richards <42894274+ezrichards@users.noreply.github.com>
Date:   Mon Feb 13 14:06:08 2023 -0700

    Merge branch 'master' of github.com:ColoradoSchoolOfMines/mineshspc.com

...
```

This is mostly useless. Let's make it better.

`git log`: but actually good

For *`git log`* to be useful, you want it to show *all* branches, show a graph, and get rid of most of the details.

```
> git log --all --graph --decorate --oneline
* b08107c (HEAD -> master, origin/master, origin/HEAD) fix some things
*   6c1f8b5 Merge branch 'master' of github.com:ColoradoSchoolOfMines/mineshspc.com
|\
| * 67f0f4d fix background color bug
* | 47bc626 Fix accordion
|/
* 186b52a Archive overhaul
* 385666b Fix accordion arrows
* 3d0f64a Archive preliminary updates
* aaae02b Update FAQ and archive pages
*   b2a64f7 Merge branch 'master' of github.com:ColoradoSchoolOfMines/mineshspc.com
|\
| * 1450745 make footer reveal
* | f7ddfa2 Fix alt text
|/
* 0f89721 created student confirm registration page
* 7b15e86 editing teams: ensure that you can't change from in-person to remote or
↪  vice versa
* e15bda6 save team below member list
* 853dcd1 add ability to add team members
```

Branches: summary

- Branches are **pointers** to commits.
- Use `git checkout` to move between branches.
- Use `git log` to see where you are.

Merging

Merging: resolving divergent histories[4]

If you want to merge the changes from branch *A* into another branch *B*, you need to:

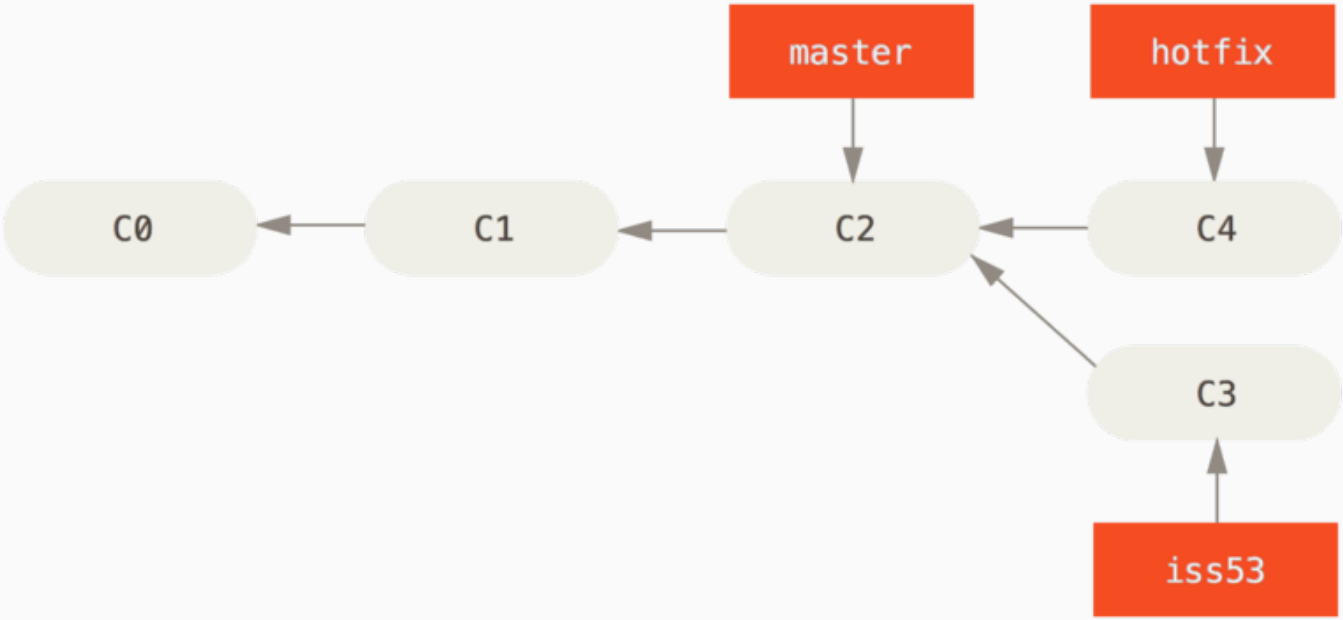
1. Switch to branch *B* (`git checkout B`)
2. Run `git merge A`.

This will do one of two things: fast-forward or create a merge commit.

[4] Info in the rest of the *Merging* section is mainly from `https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging`

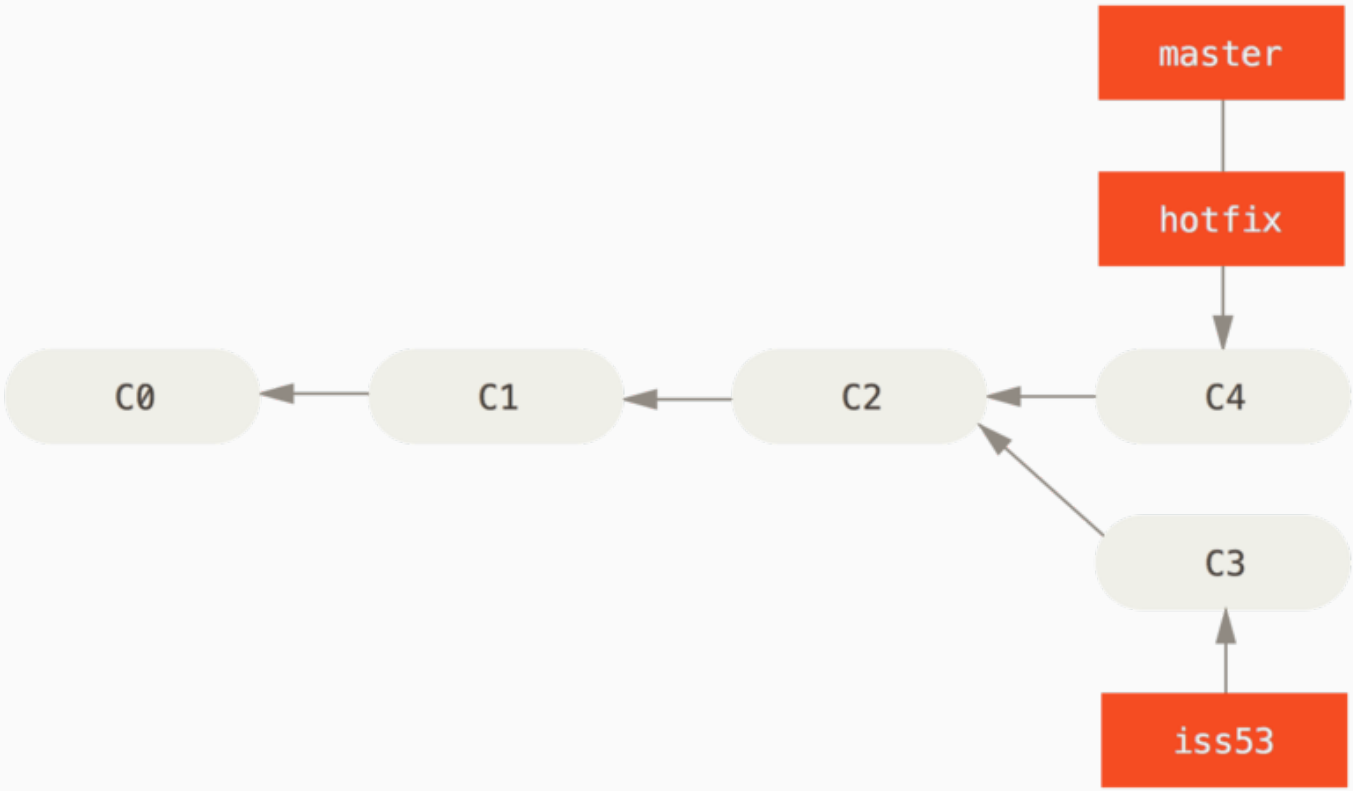
Merging: fast-forwarding

If the branch you are merging is directly ahead of the branch you are merging into, Git will just move the pointer in a **fast-forward merge**.



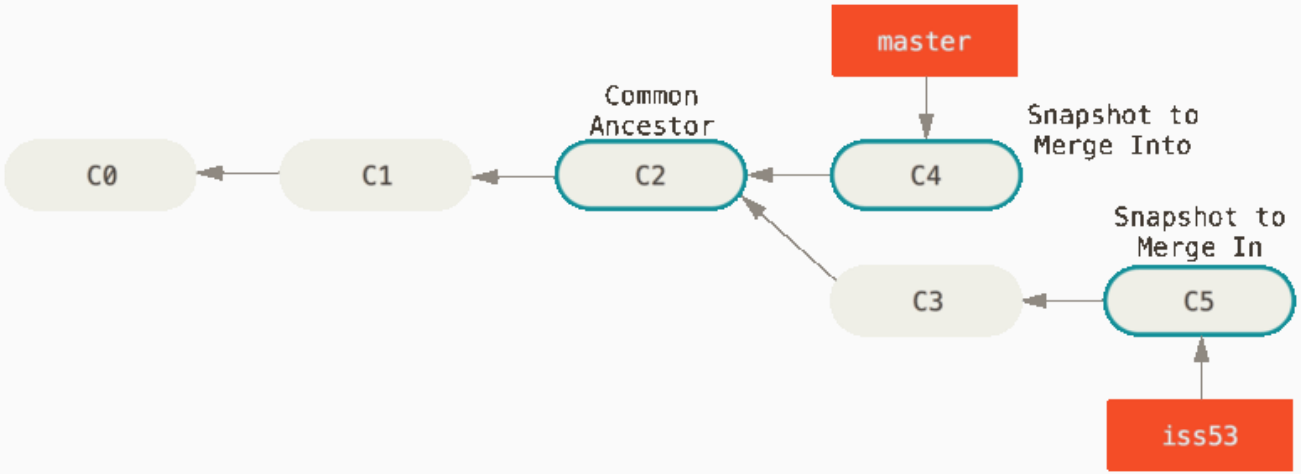
Merging: fast-forwarding

If the branch you are merging is directly ahead of the branch you are merging into, Git will just move the pointer in a **fast-forward merge**.



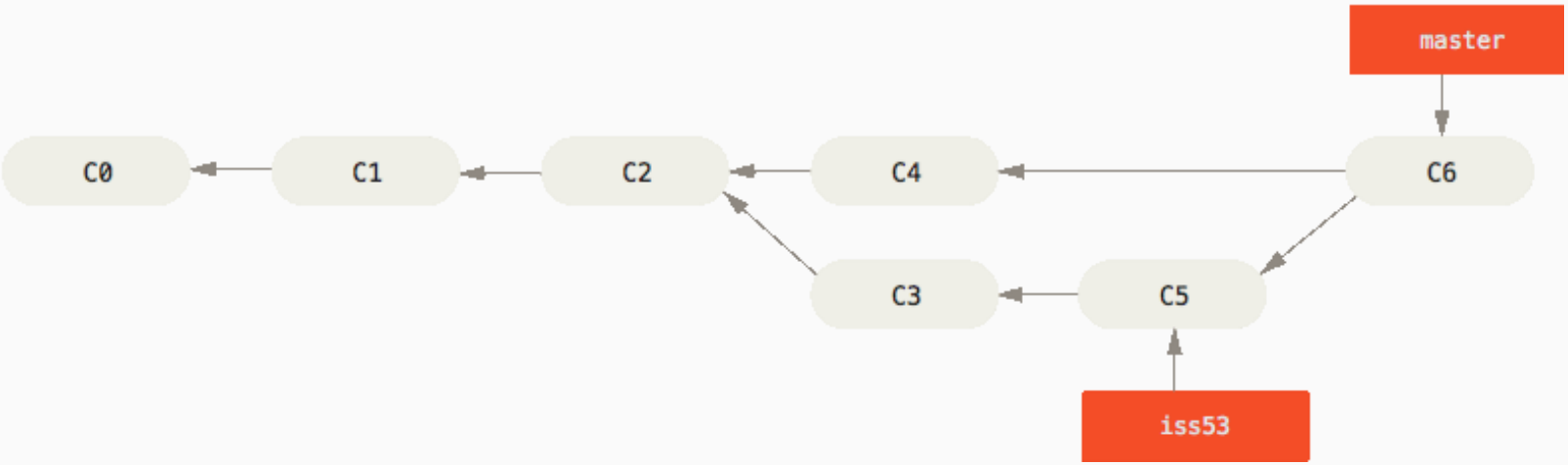
Merging: creating a merge commit

If the branch you are merging has diverged from the one you are merging into, Git will create a merge commit through a **three-way merge**.



Merging: creating a merge commit

If the branch you are merging has diverged from the one you are merging into, Git will create a merge commit through a **three-way merge**.



Merging: resolving conflicts

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging, Git won't be able to merge them cleanly.

```
> git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

You can always use `git status` to see what has been automatically merged and what files have conflicts.

```
> git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Merging: editing files to resolve merge conflicts

You can use a merge tool to resolve conflicts, however I find that it's easier to just manually resolve the conflicts.

Visual Studio Code has a good UI for this. The process you should follow is as follows:

1. Open a file with the conflict.
2. Find one of the conflict-resolution markers.
3. Make edits to resolve the conflict.
4. Run `git add` on the file.
5. Repeat steps 1-4 until all conflicts are resolved.
6. Run `git commit` to commit the merge.

Merging: understanding conflict-resolution markers

In order to find the conflict-resolution markers, search for `<<<<<<<`. Each conflict-resolution block should look something like this:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

The first part (between `<<<<<<<` and `=======`) is what the branch you are merging *into* has. The second part (between `=======` and `>>>>>>>`) is what the branch you are merging *from* has.

Sometimes, you just one one side of the conflict, other times you need to be more nuanced in your merge.

Merging: summary

- Merging allows you to pull changes from one branch into another branch.
- Use *`git merge A`* to merge branch *A* into the current branch.
- Git will do a fast-forward merge if possible, otherwise it will create a merge commit.
- You might have to resolve merge conflicts.

Rebasing

Rebasing: maintaining linear histories

Most merge commits are useless. They clutter the history, and normally don't add anything of value to the understanding of how the codebase evolved.

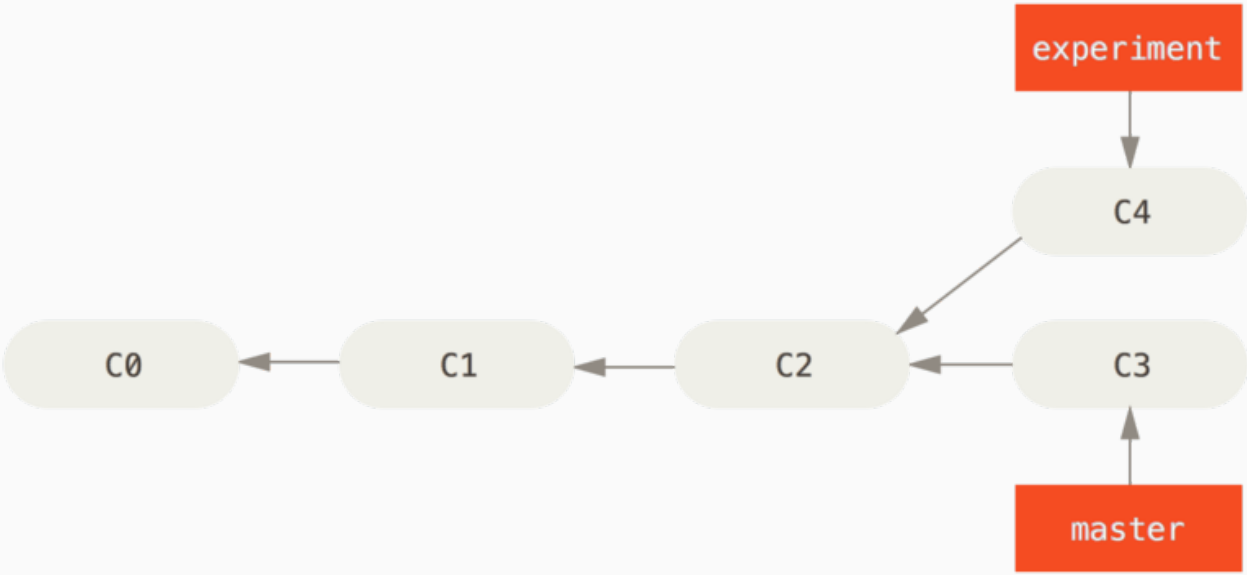
```
* b08107c (HEAD -> master, origin/master, origin/HEAD) fix some things
*   6c1f8b5 Merge branch 'master' of github.com:ColoradoSchoolOfMines/mineshspc.com
|\
| * 67f0f4d fix background color bug
* | 47bc626 Fix accordion
|/
* 186b52a Archive overhaul
* 385666b Fix accordion arrows
```

Ideally the above history would be linear:

```
* b08107c (HEAD -> master, origin/master, origin/HEAD) fix some things
* 47bc626 Fix accordion
* 67f0f4d fix background color bug
* 186b52a Archive overhaul
* 385666b Fix accordion arrows
```

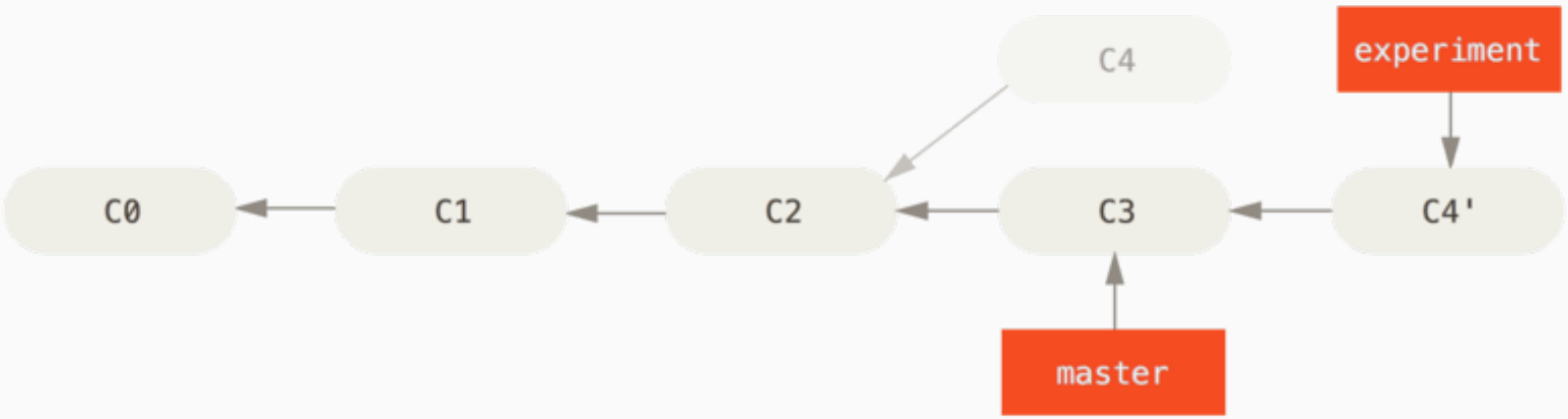
Rebasing: what it does

Rebasing allows you to take the commits from one branch and reapply them on top of another branch.



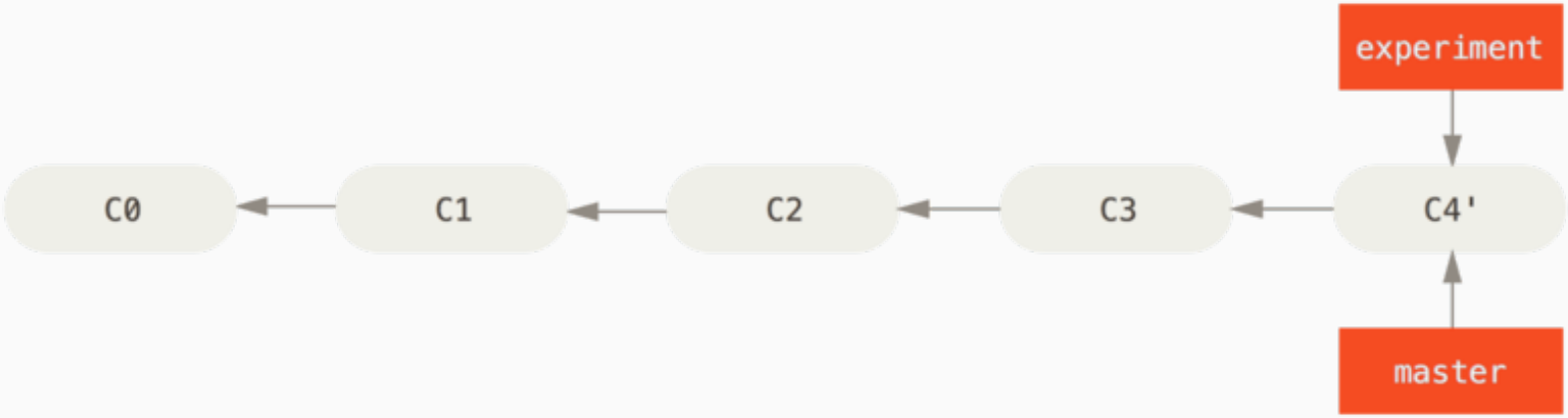
Rebasing: what it does

Rebasing allows you to take the commits from one branch and reapply them on top of another branch.



Rebasing: what it does

Rebasing allows you to take the commits from one branch and reapply them on top of another branch.



Rebasing: how to rebase

To rebase, follow this procedure:

1. Checkout the branch that you want to rebase.
2. Run *`git rebase A`*, where *A* is the branch or commit you want to rebase onto.

Note that **rebasing rewrites history**. Generally you should only rebase *local* commits, or branches that you are the only one using.

Rebasing: interactive rebase

Say you are working on a feature branch and you've made the following commits:

```
* 1a2b3c4 (HEAD -> my-feature) add more buzz
* 8a9b0c1 fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

But you realize that your *`fix fizzing`* commit introduced a bug! So you fix the issue but you don't want to end up with a "fix the fix" commit.

Interactive rebasing can help! Go ahead and create a new "fix the fix" commit:

```
* a81abe1 (HEAD -> my-feature) fixup! fix fix fizzing
* 1a2b3c4 add more buzz
* 8a9b0c1 fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

Rebasing: interactive rebase (continued)

Interactive rebasing can help! Go ahead and create a new “fix the fix” commit:

```
* a81abe1 (HEAD -> my-feature) fixup! fix fizzing
* 1a2b3c4 add more buzz
* 8a9b0c1 fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

Now, run ***`git rebase -i master`*** to start an interactive rebase. This will open an editor with the following (as well as instructions):

```
pick 8a9b0c1 fix fizzing
pick 1a2b3c4 add more buzz
pick a81abe1 fixup! fix fizzing
```

Now, you can move the commit order around by editing the file, and also ***`fixup`*** the commit, which will squash the two commits together into one!

```
pick 8a9b0c1 fix fizzing
fixup a81abe1 fixup! fix fizzing
pick 1a2b3c4 add more buzz
```

Rebasing: interactive rebase result

Now, you can move the commit order around by editing the file, and also *`fixup`* the commit, which will squash the two commits together into one!

```
pick 8a9b0c1 fix fizzing
fixup a81abe1 fixup! fix fizzing
pick 1a2b3c4 add more buzz
```

After saving and exiting the editor, Git will rewrite the history of your branch to look like this:

```
* a9bb207 (HEAD -> my-feature) add more buzz
* 4e89d4f fix fizzing
* 1d2e3f4 (master) fizz the buzz
```

and the *`fix fizzing`* commit will contain the changes from both commits!

Note that you now have different commit hashes! The old commits are still in tact (you can *`checkout`* them), but you have moved your branch (pointer) to the new commit.

Rebasing: interactive rebase shortcut

The workflow I described in the previous slides is so common that there are tools built-in to Git to help you accomplish them.

- `git commit --fixup <commit>` automatically marks your commit as a fix of a previous commit. (It uses that `fixup!` syntax from the previous slide.)
- `git rebase -i --autosquash` opens the editor and automatically reorders the fixup commits when the interactive rebase editor opens and sets them to `fixup` instead of `pick`.

See `https://fle.github.io/git-tip-keep-your-branch-clean-with-fixup-and-autos.html` for more details.

Rebasing: bailing out

If you ever need to cancel a rebase, use `git rebase --abort`.

This will restore your repository state to what it was before you started the rebase process.

Rebasing: other applications

There are lots of other reasons to rebase.

- You want to pull in changes from another branch without merging.
- You want to modify a commit that added a file by accident.
- You want to reorder your commits to make it more clear to a code reviewer what you changed.
- Somebody else pushed a commit to the *remote* branch, and you want to add your commit on top of theirs.

Remotes

Remotes: what are they?[5]

Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you.

GitHub is a popular choice for where to host your remote repositories, but other options exist such as GitLab, sourcehut, Bitbucket, and many other self-hosted options.

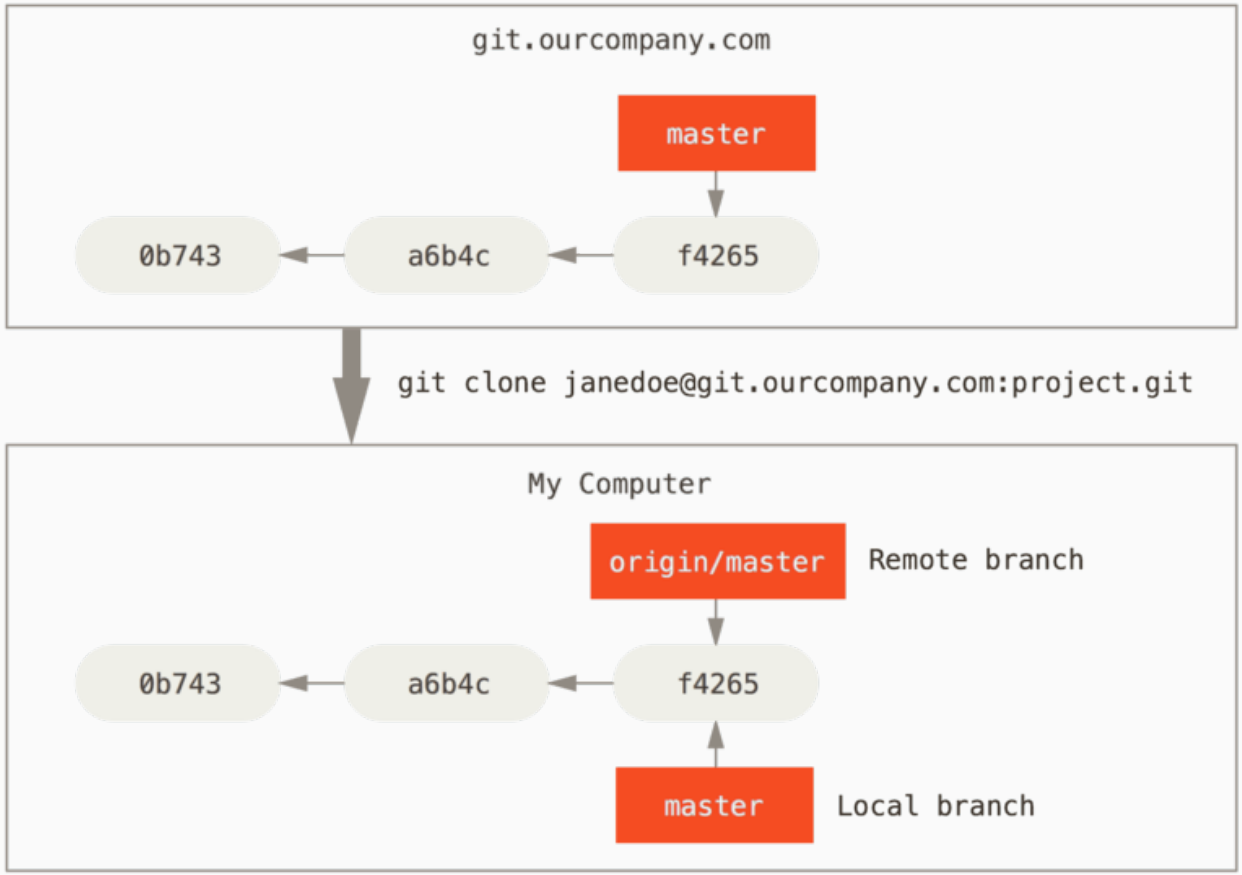
[5] Much of the content of this section is from https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes

Remotes: managing them

- View all of your remotes with:
  ```
  > git remote -v
  origin  git@github.com:sumnerevans/acm-git-good.git
  ↪  (fetch)
  origin  git@github.com:sumnerevans/acm-git-good.git
  ↪  (push)
  ```
- You can add remotes using:
  ```
  > git remote add other git@github.com:other/repo.git
  ```
- You can change the URL of a remote:
  ```
  > git remote set-url origin
  ↪  git@github.com:other/repo.git
  ```
- When you clone a repository, it automatically creates a remote called `origin` that points to your remote repository.
- **Note**: `origin` is not special, it's just a default.

Remote Branches: what are they?

The remote repository is **entirely separate from your local repository**. The remote repository has its own set of branches, commits, etc.

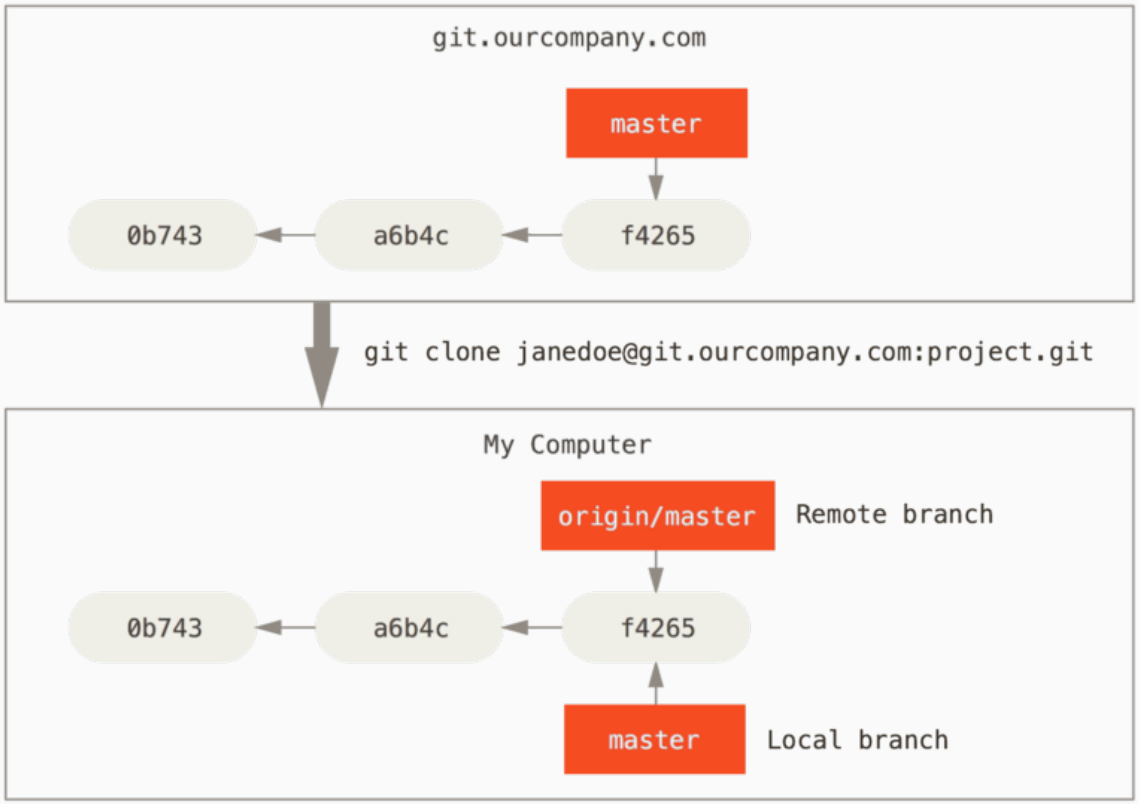


Remote Branches: three new concepts

1. **Fetch**: to download the current state of the remote repository, use the *`git fetch`* command.
   Git does not automatically fetch the state of the remote repository!
2. **Push**: to update the remote repository with your local state, use the *`git push`* command.
3. **Tracking branches**: tracking branches are local branches that have a direct relationship to a remote branch.

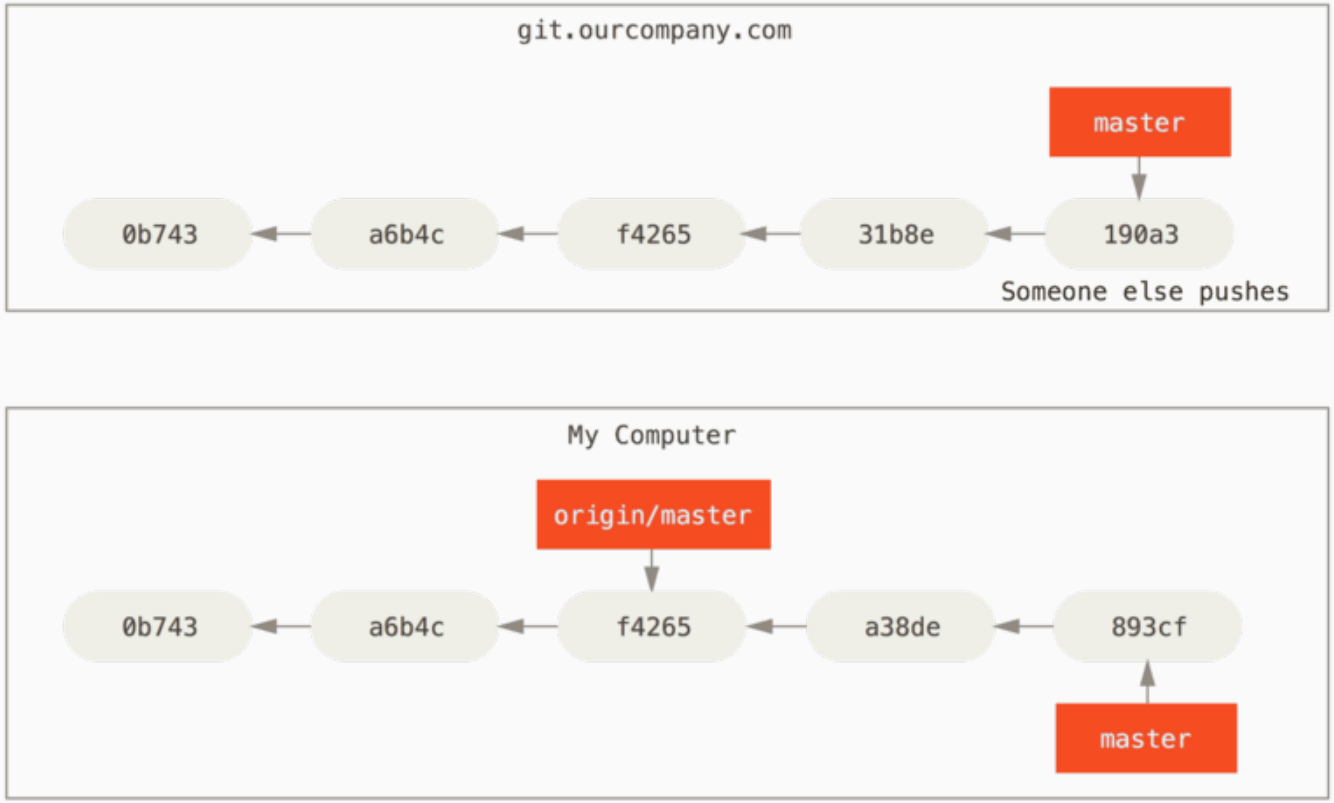
Remote Branches: divergence

The state of the local and remote repositories can diverge.



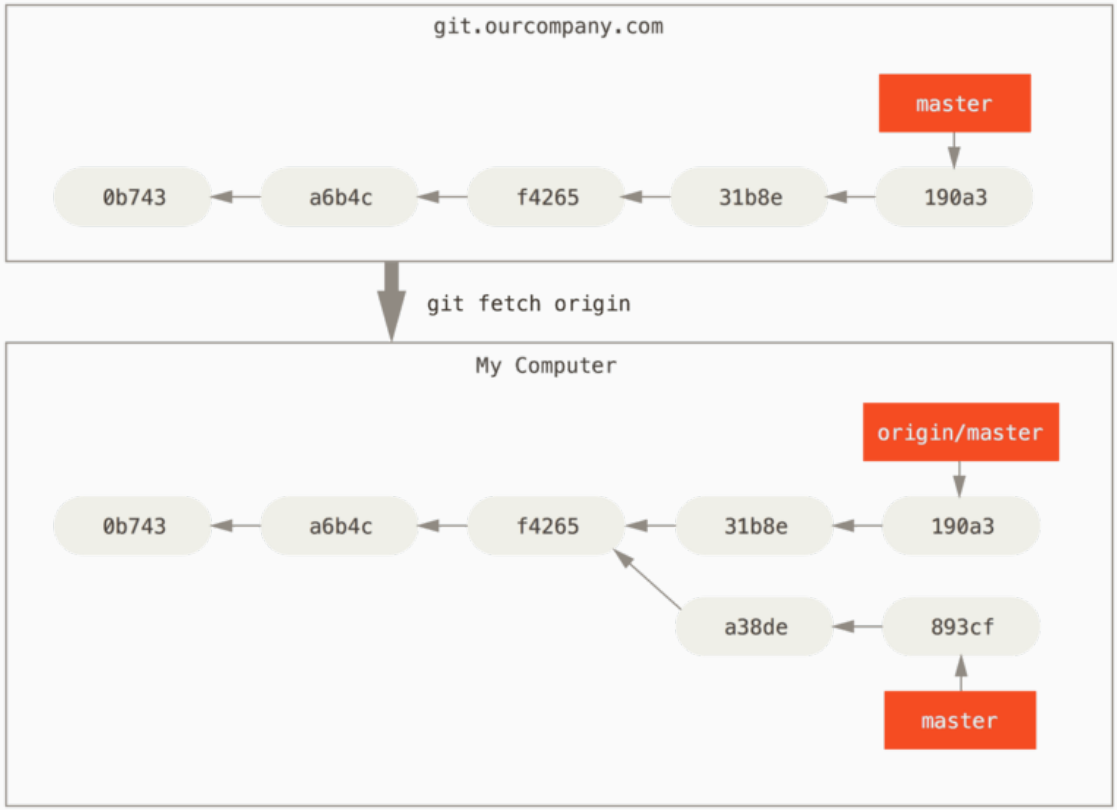
Remote Branches: divergence

The state of the local and remote repositories can diverge.



Remote Branches: divergence

The state of the local and remote repositories can diverge. Use *`git fetch`* to get the latest state of the remote repository.



Now use the tools we already know for divergent branches.

Remote Branches: divergence (continued)

Git provides a command to fetch and then merge the remote branch into the local branch called *`git pull`*.

Warning: by default, *`git pull`* will create a merge commit if the remote branch has diverged from the local one!

Merge commits are ugly! Use *`git pull --rebase`* to tell *`git pull`* to rebase your local changes on the remote changes instead.

Tell Git to never merge when pulling

You can configure Git to fail instead of making a merge commit by setting the *`pull.ff`* configuration option to *`only`*.

```
> git config --global pull.ff only
```

Remote Branches: pushing

Pushing can be thought of merging your local tracking into the remote tracking branch.

If your branch is already configured to track a remote branch, you can just use *`git push`*.

If you have a new branch that doesn't have a corresponding remote tracking branch, use

```
> git push -u origin <branch name>
```

Remote Branches: pushing harder

In certain circumstances, you want to push a divergent branch to the remote. Examples include:

- You did an interactive rebase and need to push the newly rebased changes.
- You reset your branch to a previous commit and want it to be the latest commit on the branch.

In these cases, you can use the `--force` or `--force-with-lease` options to tell Git to push your local state to the remote regardless of what is currently there.

`--force-with-lease` is less dangerous than `--force` because it will check that your current view of the remote state is up-to-date.

Advanced Tips

Undoing Things

Commits are kept around in the `.git` directory (remember, it's like a heap), which means that even if you loose a pointer to a commit (for example by moving all branch pointers away from it), it still exists!

You can use `git reflog` to see the history of when the tips of branches and other references were updated in the local repository.

If you screwed something up and you don't know what to do, `https://ohshitgit.com/` is a great resource.

Cherry-picking

You can apply arbitrary commits to your current branch by cherry-picking them.

This is similar to rebasing.

Aliases

Git supports the concept of *aliasing* one command to another name.

For example, you can alias the pretty log command I showed earlier to `git l` by using the following command:

```
> git config --global alias.l "log --oneline --graph --all
↪  --decorate"
```

Now you can just type `git l` to get the pretty version of the log output.

Ignoring Files

You can tell Git to not ever commit certain files by ignoring them using the `.gitignore` file.

Any file that matches one of the patterns in `.gitignore` will not be tracked by Git.

Warning: if you have already committed the file, it will still be tracked.

If you want to delete a file from Git, but keep it on your computer, use `git rm --cached <file>`.

Blaming

Want to see the last commit that modified each line of a file? Use *`git blame`*.

This is often a helpful way to see which commit cause a bug or see what needs to be changed when making a similar change.

Resources/Tips

I obviously was unable to tell you about everything you can do with Git. There are hundreds of options on every single command.

- `man git *`: The man pages on Git are good. Use them as your first line of defense.
- git-scm.com/book/en/v2: A huge resource about how to do everything Git.
- gitignore.io: Generates a `.gitignore` file for a given project type, OS, and IDE.
- delta: provides a much prettier `diff` interface.

References

- Eigen: https://eigen.tuxfamily.org/
- PCL: https://pointclouds.org/
- OpenCV: https://opencv.org/
- Open3D: https://www.open3d.org/
- Git: https://git-scm.com/
- MIT Visual Navigation: https://vnav.mit.edu/
- Modern C++ for Computer Vision: https://www.ipb.uni-bonn.de/modern-cpp/index.html