How to determine point cloud resolution?

Spatial X and Y

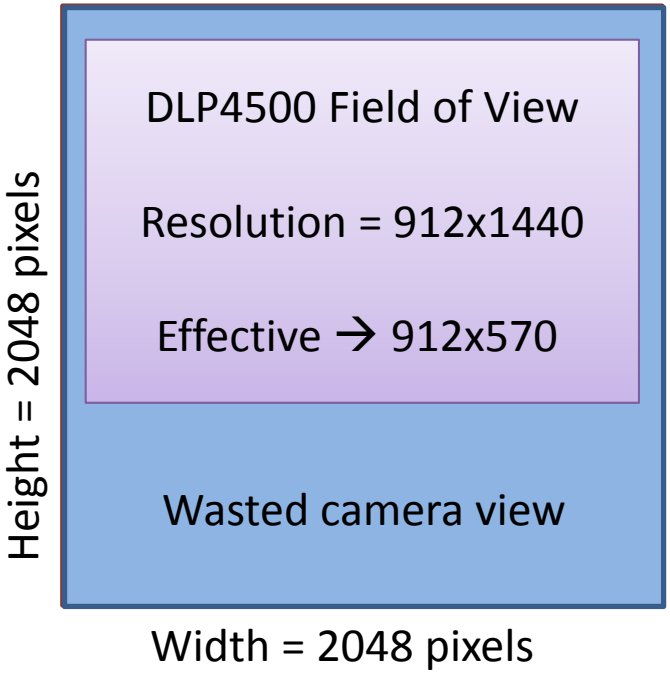
- Inversely proportional to field of view
  - Scanning larger areas worsens spatial resolution
  - Scanning smaller areas improves spatial resolution
- Proportional to camera and projector resolution
  - Increasing camera or projector resolution improves spatial resolution
  - Also increases number of points in cloud

Z-Depth

- Inversely proportional to focal length and baseline
  - Longer focal lengths improve accuracy
  - Increasing the baseline distance improves accuracy
- Proportional to the object distance and disparity resolution
  - Accuracy decreases as distance increases
  - Increasing camera and projector resolutions improves accuracy

System Resolutions & Field of Views

- Nyquist theorem requires at least 2x sampling
  - Camera width resolution must be double projector width resolution
  - Camera height resolution must be double projector height resolution
  - Camera pixel count should be at least 4 times larger than projector's!
- Field of view and "effective resolution" must be considered



Field of view mismatch means smaller effective resolution…

$$2048\ pxls * 60\% = 1228\ effective\ pxls$$

Check pixel sampling…

$$\frac{1228\ effective\ pxls}{1140\ projector\ pxls} = 1.07 < 2$$

Cannot resolve all projector rows!!

$$\frac{1228\ effective\ pxls}{570\ projector\ pxls} = 2.15 > 2$$

Can resolve projector row pairs

Scalable Solutions with OOP

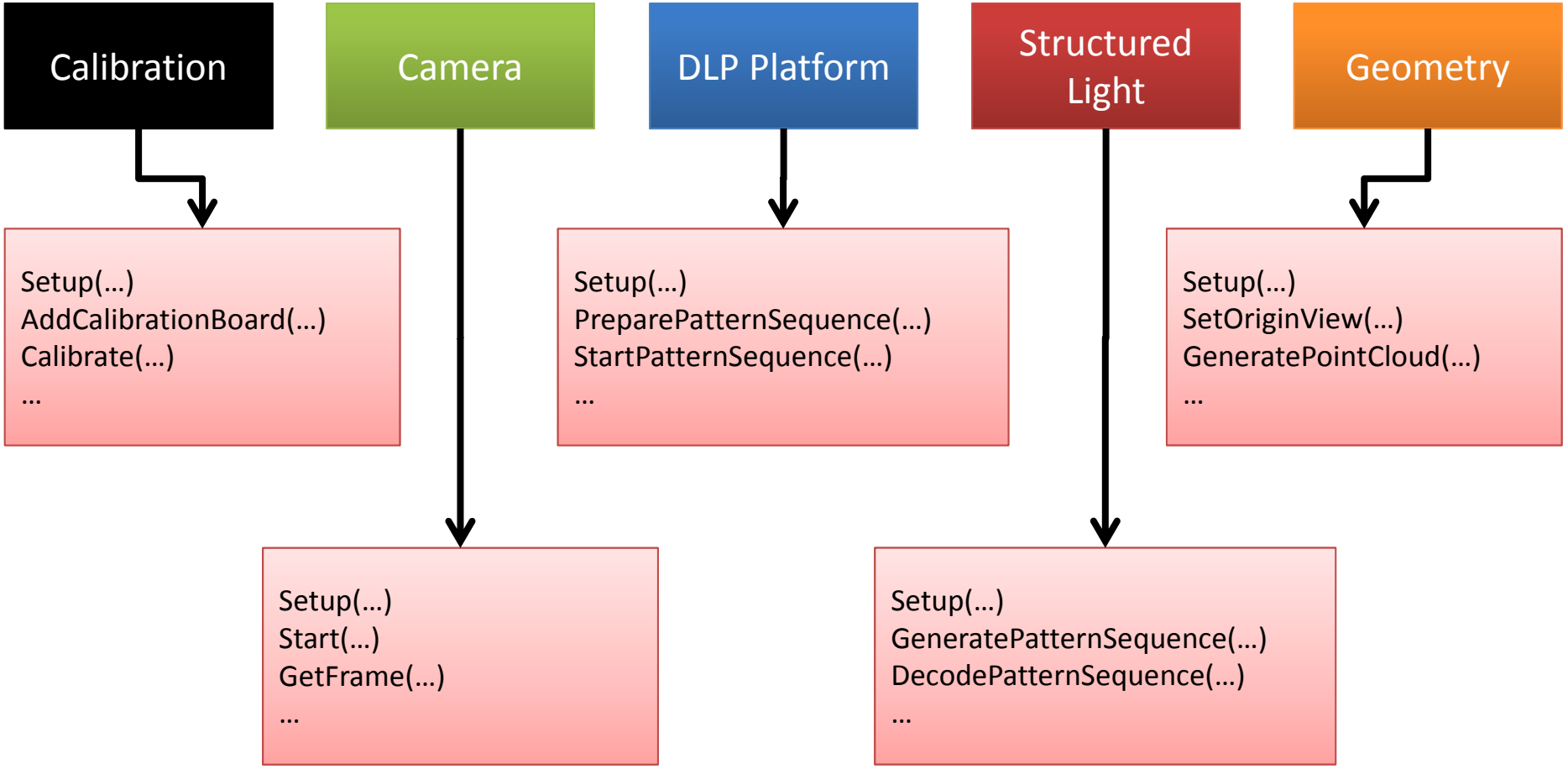
- Consider the LightCrafter 4500 and LightCrafter 6500 EVMS
  - Both chipsets have different API, resolutions, speeds, etc.
  - Did I need to rewrite each application with all of the chipset specific API ?
    - NO! On

```
1412    // System Variables
1413    dlp::PG_FlyCap2_C   camera;
1414    dlp::LCr4500        projector;
```

```
1412    // System Variables
1413    dlp::PG_FlyCap2_C   camera;
1414    dlp::LCr6500        projector;
```

  - **The LightCrafter 3000 has recently been added also!**
- How is this possible?
  - DLP Structured Light SDK contains modules which define interfaces
  - C++ allows you to reference sub-classes as their parent class

What are the primary abstract modules?



- Each module (base-class) defines an interface which all sub-modules (sub-classes) must follow

How to use abstracted modules?

```
void ScanObject(dlp::Camera          *camera,
                const std::string    &camera_calib_data_file,
                dlp::DLP_Platform    *projector,
                const std::string    &projector_calib_data_file,
                dlp::StructuredLight *structured_light_vertical,
                dlp::StructuredLight *structured_light_horizontal,
                const bool           &use_vertical,
                const bool           &use_horizontal,
                const std::string    &geometry_settings_file){
```

- F d
  b

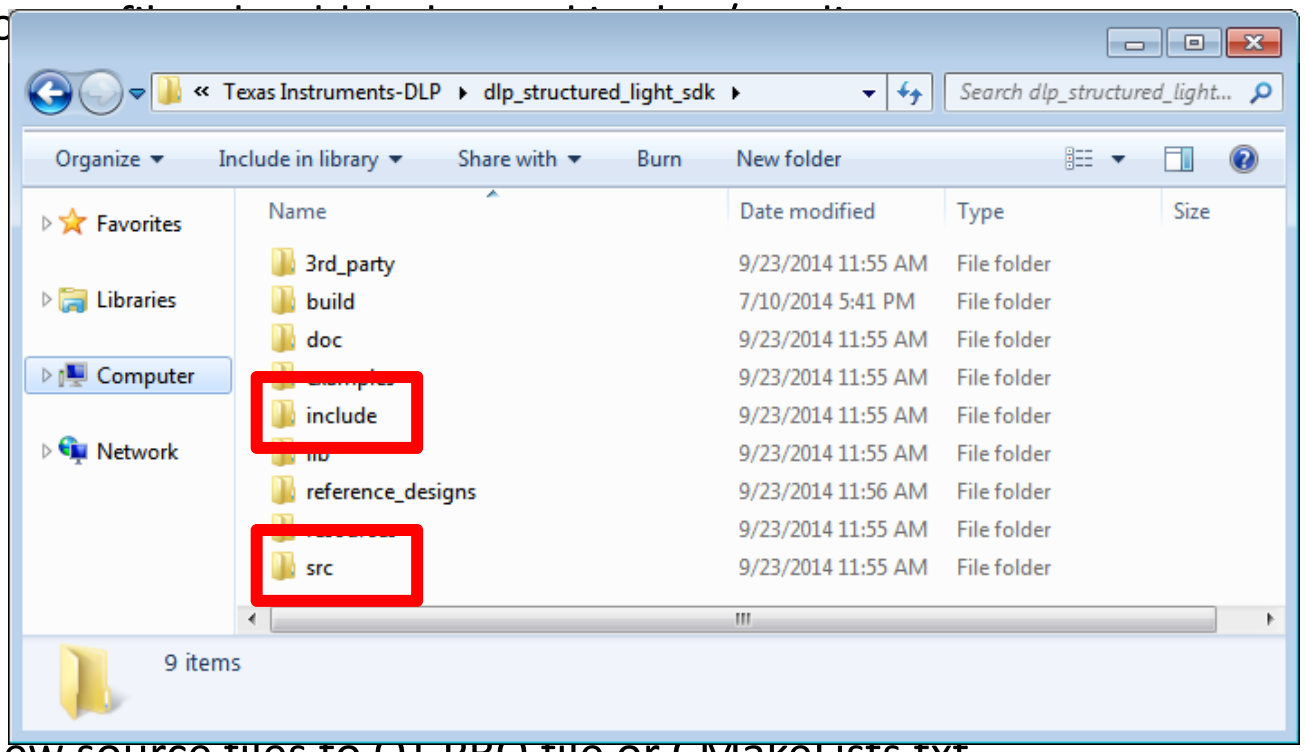
```
// System Variables
dlp::PG_FlyCap2_C   camera;
dlp::LCr6500        projector;

ScanObject(&camera,
           calibration_data_file_camera,
           &projector,
           calibration_data_file_projector,
           &structured_light_vertical,
           &structured_light_horizontal,
           true,
           true,
           geometry_settings_file);
```

- S e passed as the
  p

Where should source code go?

- Use the current sub-modules for reference
  - Header files should be located in the /include directory
  - So
- Add new source files to QT PRO file or CMakeLists.txt



How to creating a new camera module?

- Reference the module base-class header files to identify what functions need to be written for a sub-class
  - All virtual functions must be written by the sub-class!

```
class Camera: public dlp::Module{
    public:

    // Define by subclass
    virtual ReturnCode Connect(int camera_id) = 0;
    virtual ReturnCode Disconnect() = 0;

    virtual ReturnCode Start() = 0;
    virtual ReturnCode Stop() = 0;

    virtual ReturnCode GetFrame(Image* ret_frame) = 0;
}
```

```
class PG_FlyCap2_C : public Camera
{
public:

    // Define pure virtual functions
    ReturnCode Connect(int camera_id);
    ReturnCode Disconnect();

    ReturnCode Start();
    ReturnCode Stop();

    ReturnCode GetFrame(Image* ret_frame);
}
```

NOTE: This is only an example!! Please reference camera.hpp for the complete camera module declaration

Notice that the sub-class contains the exact same methods