PCL Documentation

Release 0.0.1

arzoo

Feb 29, 2020

Contents

The current document explains not only how to add your own *PointT* point type, but also what templated point types are in PCL, why do they exist, and how are they exposed. If you're already familiar with this information, feel free to skip to the last part of the document.

Contents

Note: The current document is valid only for PCL 0.x and 1.x. Note that at the time of this writing we are expecting things to be changed in PCL 2.x.

PCL comes with a variety of pre-defined point types, ranging from SSE-aligned structures for XYZ data, to more complex n-dimensional histogram representations such as PFH (Point Feature Histograms). These types should be enough to support all the algorithms and methods implemented in PCL. However, there are cases where users would like to define new types. This document describes the steps involved in defining your own custom PointT type and making sure that your project can be compiled successfully and ran.

CHAPTER 1

Why *PointT* types

PCL's *PointT* legacy goes back to the days where it was a library developed within ROS. The consensus then was that a *Point Cloud* is a complicated *n-D* structure that needs to be able to represent different types of information. However, the user should know and understand what types of information need to be passed around, in order to make the code easier to debug, think about optimizations, etc.

One example is represented by simple operations on *XYZ* data. The most efficient way for SSE-enabled processors, is to store the 3 dimensions as floats, followed by an extra float for padding:

```
struct PointXYZ
{
  float x;
  float y;
  float z;
  float padding;
};
```

As an example however, in case an user is looking at compiling PCL on an embedded platform, adding the extra padding can be a waste of memory. Therefore, a simpler *PointXYZ* structure without the last float could be used instead.

Moreover, if your application requires a *PointXYZRGBNormal* which contains *XYZ* 3D data, *RGBA* information (colors), and surface normals estimated at each point, it is trivial to define a structure with all the above. Since all algorithms in PCL should be templated, there are no other changes required other than your structure definition.

CHAPTER 2

What *PointT* types are available in PCL?

To cover all possible cases that we could think of, we defined a plethora of point types in PCL. The following might be only a snippet, please see point_types.hpp for the complete list.

This list is important, because before defining your own custom type, you need to understand why the existing types were created they way they were. In addition, the type that you want, might already be defined for you.

- *PointXYZ* - Members: float x, y, z;

  This is one of the most used data types, as it represents 3D xyz information only. The 3 floats are padded with an additional float for SSE alignment. The user can either access *points[i].data[0]* or *points[i].x* for accessing say, the x coordinate.

```
union
{
  float data[4];
  struct
  {
    float x;
    float y;
    float z;
  };
};
```

- *PointXYZI* - Members: float x, y, z, intensity;

  Simple XYZ + intensity point type. In an ideal world, these 4 components would create a single structure, SSE-aligned. However, because the majority of point operations will either set the last component of the *data[4]* array (from the xyz union) to 0 or 1 (for transformations), we cannot make *intensity* a member of the same structure, as its contents will be overwritten. For example, a dot product between two points will set their 4th component to 0, otherwise the dot product doesn't make sense, etc.

  Therefore for SSE-alignment, we pad intensity with 3 extra floats. Inefficient in terms of storage, but good in terms of memory alignment.

```
union
{
  float data[4];
  struct
  {
    float x;
    float y;
    float z;
  };
};
union
{
  struct
  {
    float intensity;
  };
  float data_c[4];
};
```

- *PointXYZRGBA* - Members: float x, y, z; std::uint32_t rgba;

  Similar to *PointXYZI*, except *rgba* contains the RGBA information packed into an unsigned 32-bit integer. Thanks to the *union* declaration, it is also possible to access color channels individually by name.

Note: The nested *union* declaration provides yet another way to look at the RGBA data–as a single precision floating point number. This is present for historical reasons and should not be used in new code.

```
union
{
  float data[4];
  struct
  {
    float x;
    float y;
    float z;
  };
};
union
{
  union
  {
    struct
    {
      std::uint8_t b;
      std::uint8_t g;
      std::uint8_t r;
      std::uint8_t a;
    };
    float rgb;
  };
  std::uint32_t rgba;
};
```

- *PointXYZRGB* - float x, y, z; std::uint32_t rgba;

  Same as *PointXYZRGBA*.

- *PointXY* - float x, y;

  Simple 2D x-y point structure.

```
struct
{
  float x;
  float y;
};
```

- *InterestPoint* - float x, y, z, strength;

  Similar to *PointXYZI*, except *strength* contains a measure of the strength of the keypoint.

```
union
{
  float data[4];
  struct
  {
    float x;
    float y;
    float z;
  };
};
union
{
  struct
  {
    float strength;
  };
  float data_c[4];
};
```

- *Normal* - float normal[3], curvature;

  One of the other most used data types, the *Normal* structure represents the surface normal at a given point, and a measure of curvature (which is obtained in the same call as a relationship between the eigenvalues of a surface patch – see the *NormalEstimation* class API for more information).

  Because operation on surface normals are quite common in PCL, we pad the 3 components with a fourth one, in order to be SSE-aligned and computationally efficient. The user can either access *points[i].data_n[0]* or *points[i].normal[0]* or *points[i].normal_x* for accessing say, the first coordinate of the normal vector. Again, *curvature* cannot be stored in the same structure as it would be overwritten by operations on the normal data.

```
union
{
  float data_n[4];
  float normal[3];
  struct
  {
    float normal_x;
    float normal_y;
    float normal_z;
  };
}
union
{
  struct
  {
```

(continues on next page)

(continued from previous page)

```
    float curvature;
  };
  float data_c[4];
};
```

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

  A point structure that holds XYZ data, together with surface normals and curvatures.

```
union
{
  float data[4];
  struct
  {
    float x;
    float y;
    float z;
  };
};
union
{
  float data_n[4];
  float normal[3];
  struct
  {
    float normal_x;
    float normal_y;
    float normal_z;
  };
};
union
{
  struct
  {
    float curvature;
  };
  float data_c[4];
};
```

- *PointXYZRGBNormal* - float x, y, z, normal[3], curvature; std::uint32_t rgba;

  A point structure that holds XYZ data, and RGBA colors, together with surface normals and curvatures.

Note: Despite the name, this point type does contain the alpha color channel.

```
union
{
  float data[4];
  struct
  {
    float x;
    float y;
    float z;
  };
};
```

(continues on next page)

(continued from previous page)

```
union
{
  float data_n[4];
  float normal[3];
  struct
  {
    float normal_x;
    float normal_y;
    float normal_z;
  };
}
union
{
  struct
  {
    union
    {
      union
      {
        struct
        {
          std::uint8_t b;
          std::uint8_t g;
          std::uint8_t r;
          std::uint8_t a;
        };
        float rgb;
      };
      std::uint32_t rgba;
    };
    float curvature;
  };
  float data_c[4];
};
```

- *PointXYZINormal* - float x, y, z, intensity, normal[3], curvature;

  A point structure that holds XYZ data, and intensity values, together with surface normals and curvatures.

```
union
{
  float data[4];
  struct
  {
    float x;
    float y;
    float z;
  };
};
union
{
  float data_n[4];
  float normal[3];
  struct
  {
    float normal_x;
    float normal_y;
```

(continues on next page)

(continued from previous page)

```
    float normal_z;
  };
}
union
{
  struct
  {
    float intensity;
    float curvature;
  };
  float data_c[4];
};
```

- *PointWithRange* - float x, y, z (union with float point[4]), range;

  Similar to *PointXYZI*, except *range* contains a measure of the distance from the acquisition viewpoint to the point in the world.

```
union
{
  float data[4];
  struct
  {
    float x;
    float y;
    float z;
  };
};
union
{
  struct
  {
    float range;
  };
  float data_c[4];
};
```

- *PointWithViewpoint* - float x, y, z, vp_x, vp_y, vp_z;

  Similar to *PointXYZI*, except *vp_x*, *vp_y*, and *vp_z* contain the acquisition viewpoint as a 3D point.

```
union
{
  float data[4];
  struct
  {
    float x;
    float y;
    float z;
  };
};
union
{
  struct
  {
    float vp_x;
    float vp_y;
```

(continues on next page)

(continued from previous page)

```
    float vp_z;
  };
  float data_c[4];
};
```

- *MomentInvariants* - float j1, j2, j3;

  Simple point type holding the 3 moment invariants at a surface patch. See *MomentInvariantsEstimation* for more information.

```
struct
{
  float j1, j2, j3;
};
```

- *PrincipalRadiiRSD* - float r_min, r_max;

  Simple point type holding the 2 RSD radii at a surface patch. See *RSDEstimation* for more information.

```
struct
{
  float r_min, r_max;
};
```

- *Boundary* - std::uint8_t boundary_point;

  Simple point type holding whether the point is lying on a surface boundary or not. See *BoundaryEstimation* for more information.

```
struct
{
  std::uint8_t boundary_point;
};
```

- *PrincipalCurvatures* - float principal_curvature[3], pc1, pc2;

  Simple point type holding the principal curvatures of a given point. See *PrincipalCurvaturesEstimation* for more information.

```
struct
{
  union
  {
    float principal_curvature[3];
    struct
    {
      float principal_curvature_x;
      float principal_curvature_y;
      float principal_curvature_z;
    };
  };
  float pc1;
  float pc2;
};
```

- *PFHSignature125* - float pfh[125];

  Simple point type holding the PFH (Point Feature Histogram) of a given point. See *PFHEstimation* for more information.

```
struct
{
  float histogram[125];
};
```

- *FPFHSignature33* - float fpfh[33];

  Simple point type holding the FPFH (Fast Point Feature Histogram) of a given point. See *FPFHEstimation* for more information.

```
struct
{
  float histogram[33];
};
```

- *VFHSignature308* - float vfh[308];

  Simple point type holding the VFH (Viewpoint Feature Histogram) of a given point. See *VFHEstimation* for more information.

```
struct
{
  float histogram[308];
};
```

- *Narf36* - float x, y, z, roll, pitch, yaw; float descriptor[36];

  Simple point type holding the NARF (Normally Aligned Radius Feature) of a given point. See *NARFEstimation* for more information.

```
struct
{
  float x, y, z, roll, pitch, yaw;
  float descriptor[36];
};
```

- *BorderDescription* - int x, y; BorderTraits traits;

  Simple point type holding the border type of a given point. See *BorderEstimation* for more information.

```
struct
{
  int x, y;
  BorderTraits traits;
};
```

- *IntensityGradient* - float gradient[3];

  Simple point type holding the intensity gradient of a given point. See *IntensityGradientEstimation* for more information.

```
struct
{
  union
  {
    float gradient[3];
    struct
    {
      float gradient_x;
```

(continues on next page)

(continued from previous page)

```
      float gradient_y;
      float gradient_z;
    };
  };
};
```

- *Histogram* - float histogram[N];

  General purpose n-D histogram placeholder.

```
template <int N>
struct Histogram
{
  float histogram[N];
};
```

- *PointWithScale* - float x, y, z, scale;

  Similar to *PointXYZI*, except *scale* contains the scale at which a certain point was considered for a geometric operation (e.g. the radius of the sphere for its nearest neighbors computation, the window size, etc).

```
struct
{
  union
  {
    float data[4];
    struct
    {
      float x;
      float y;
      float z;
    };
  };
  float scale;
};
```

- *PointSurfel* - float x, y, z, normal[3], rgba, radius, confidence, curvature;

  A complex point type containing XYZ data, surface normals, together with RGB information, scale, confidence, and surface curvature.

```
union
{
  float data[4];
  struct
  {
    float x;
    float y;
    float z;
  };
};
union
{
  float data_n[4];
  float normal[3];
  struct
  {
```

(continues on next page)

(continued from previous page)

```
    float normal_x;
    float normal_y;
    float normal_z;
  };
};
union
{
  struct
  {
    std::uint32_t rgba;
    float radius;
    float confidence;
    float curvature;
  };
  float data_c[4];
};
```

CHAPTER 3

How are the point types exposed?

Because of its large size, and because it's a template library, including many PCL algorithms in one source file can slow down the compilation process. At the time of writing this document, most C++ compilers still haven't been properly optimized to deal with large sets of templated files, especially when optimizations (*-O2* or *-O3*) are involved.

To speed up user code that includes and links against PCL, we are using *explicit template instantiations*, by including all possible combinations in which all algorithms could be called using the already defined point types from PCL. This means that once PCL is compiled as a library, any user code will not require to compile templated code, thus speeding up compile time. The trick involves separating the templated implementations from the headers which forward declare our classes and methods, and resolving at link time. Here's a fictitious example:

```
// foo.h

#ifndef PCL_FOO_
#define PCL_FOO_

template <typename PointT>
class Foo
{
  public:
    void
    compute (const pcl::PointCloud<PointT> &input,
             pcl::PointCloud<PointT> &output);
}

#endif // PCL_FOO_
```

The above defines the header file which is usually included by all user code. As we can see, we're defining methods and classes, but we're not implementing anything yet.

```
// impl/foo.hpp

#ifndef PCL_IMPL_FOO_
#define PCL_IMPL_FOO_

```

(continues on next page)

(continued from previous page)

```
#include "foo.h"

template <typename PointT> void
Foo::compute (const pcl::PointCloud<PointT> &input,
              pcl::PointCloud<PointT> &output)
{
  output = input;
}

#endif // PCL_IMPL_FOO_
```

The above defines the actual template implementation of the method *Foo::compute*. This should normally be hidden from user code.

```
// foo.cpp

#include "pcl/point_types.h"
#include "pcl/impl/instantiate.hpp"
#include "foo.h"
#include "impl/foo.hpp"

// Instantiations of specific point types
PCL_INSTANTIATE(Foo, PCL_XYZ_POINT_TYPES));
```

And finally, the above shows the way the explicit instantiations are done in PCL. The macro *PCL_INSTANTIATE* does nothing else but go over a given list of types and creates an explicit instantiation for each. From *pcl/include/pcl/impl/instantiate.hpp*:

```
// PCL_INSTANTIATE: call to instantiate template TEMPLATE for all
// POINT_TYPES

#define PCL_INSTANTIATE_IMPL(r, TEMPLATE, POINT_TYPE) \
  BOOST_PP_CAT(PCL_INSTANTIATE_, TEMPLATE)(POINT_TYPE)

#define PCL_INSTANTIATE(TEMPLATE, POINT_TYPES)        \
  BOOST_PP_SEQ_FOR_EACH(PCL_INSTANTIATE_IMPL, TEMPLATE, POINT_TYPES);
```

Where *PCL_XYZ_POINT_TYPES* is (from *pcl/include/pcl/impl/point_types.hpp*):

```
// Define all point types that include XYZ data
#define PCL_XYZ_POINT_TYPES   \
  (pcl::PointXYZ)             \
  (pcl::PointXYZI)            \
  (pcl::PointXYZRGBA)         \
  (pcl::PointXYZRGB)          \
  (pcl::InterestPoint)        \
  (pcl::PointNormal)          \
  (pcl::PointXYZRGBNormal)    \
  (pcl::PointXYZINormal)      \
  (pcl::PointWithRange)       \
  (pcl::PointWithViewpoint)   \
  (pcl::PointWithScale)
```

Basically, if you only want to explicitly instantiate *Foo* for *pcl::PointXYZ*, you don't need to use the macro, as something as simple as the following would do:

```cpp
// foo.cpp

#include "pcl/point_types.h"
#include "pcl/impl/instantiate.hpp"
#include "foo.h"
#include "impl/foo.hpp"

template class Foo<pcl::PointXYZ>;
```

Note: For more information about explicit instantiations, please see *C++ Templates - The Complete Guide*, by David Vandervoorde and Nicolai M. Josuttis.

CHAPTER 4

How to add a new *PointT* type

To add a new point type, you first have to define it. For example:

```
struct MyPointType
{
  float test;
};
```

Then, you need to make sure your code includes the template header implementation of the specific class/algorithm in PCL that you want your new point type *MyPointType* to work with. For example, say you want to use *pcl::PassThrough*. All you would have to do is:

```
#define PCL_NO_PRECOMPILE
#include <pcl/filters/passthrough.h>
#include <pcl/filters/impl/passthrough.hpp>

// the rest of the code goes here
```

If your code is part of the library, which gets used by others, it might also make sense to try to use explicit instantiations for your *MyPointType* types, for any classes that you expose (from PCL our outside PCL).

Note: Starting with PCL-1.7 you need to define PCL_NO_PRECOMPILE before you include any PCL headers to include the templated algorithms as well.

CHAPTER 5

Example

The following code snippet example creates a new point type that contains XYZ data (SSE padded), together with a test float.

```
#define PCL_NO_PRECOMPILE
#include <pcl/pcl_macros.h>
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <pcl/io/pcd_io.h>

struct MyPointType
{
  PCL_ADD_POINT4D;                  // preferred way of adding a XYZ+padding
  float test;
  PCL_MAKE_ALIGNED_OPERATOR_NEW     // make sure our new allocators are aligned
} EIGEN_ALIGN16;                    // enforce SSE padding for correct memory␣
↪alignment

POINT_CLOUD_REGISTER_POINT_STRUCT (MyPointType,           // here we assume a XYZ +
↪"test" (as fields)
                                   (float, x, x)
                                   (float, y, y)
                                   (float, z, z)
                                   (float, test, test)
)


int
main (int argc, char** argv)
{
  pcl::PointCloud<MyPointType> cloud;
  cloud.points.resize (2);
  cloud.width = 2;
  cloud.height = 1;

```

(continues on next page)

(continued from previous page)

```
  cloud.points[0].test = 1;
  cloud.points[1].test = 2;
  cloud.points[0].x = cloud.points[0].y = cloud.points[0].z = 0;
  cloud.points[1].x = cloud.points[1].y = cloud.points[1].z = 3;

  pcl::io::savePCDFile ("test.pcd", cloud);
}
```