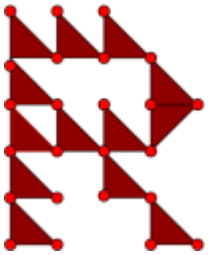

ANSI C Library for PLY file format input and output

Introduction

RPly is a library that lets applications read and write PLY files. The PLY file format is widely used to store geometric information, such as 3D models, but is general enough to be useful for other purposes.

There are other libraries out there, of course. I tried using them and finally decided to write my own. The result is RPly, and I hope you are as happy with it as I am.

RPly is easy to use, well documented, small, free, open-source, ANSI C, efficient, and well tested. I will keep supporting it for a while because all my tools use the library for input/output. The highlights are:

- A callback mechanism that makes PLY file input straightforward;
- Support for the full range of numeric formats though the user only deals with doubles;
- Binary (big and little endian) and text modes are fully supported;
- Input and output are buffered for efficiency;
- Available under the MIT license for added freedom.

The format was developed at Stanford University for use with their 3D scanning projects. Greg Turk's PLY library, available from Georgia Institute of Technology, seems to be the standard reference to the PLY file format, although there are some variations out there.

Whatever documentation and examples were found, were taken into consideration to create RPly. In theory, since RPly doesn't try to interpret the meaning of the data in a PLY file, it should be able to read any PLY file. In practice, the library works with all PLY files that I could find.

Contents

What is a PLY file?

A PLY file contains the description of one object. This object is composed by *elements*, each element type being defined by a group of *properties*. The PLY file format specifies a syntax for the description of element types and the properties that compose them, as well as comments and meta-information.

The element type descriptions come in a header, which is followed by element instances. Element instances come grouped by their type, in the order of declaration. Each element instance is defined by the value of its properties. Properties values also appear in the order of their declaration.

Here is a sample PLY file describing a triangle:

```
ply
format ascii 1.0
comment this is a simple file
obj_info any data, in one line of free form text
element vertex 3
property float x
property float y
property float z
element face 1
property list uchar int vertex_indices
end_header
-1 0 0
 0 1 0
 1 0 0
3 0 1 2
```

The header goes from the first line to the line marked by `end_header`. The first line contains only `ply\n` and is used to detect whether a file is in PLY format or not (RPly also accepts files that start with `ply\r\n`, in which case the end-of-line terminator is assumed to be `\r\n` throughout.) The second line specifies the `format` number (which is always `1.0`) and the storage mode (`ascii`, `binary_big_endian` or `binary_little_endian`).

Lines that start with `comment` are just comments, of course. Lines that start with `obj_info` contain meta-information about the object. `Comment`s and `obj_info`s are optional and their relative order in the header is irrelevant.

In the sample PLY file, the first element type is declared with name `vertex`, and on the same line we learn that there will be 3 instances of this element type. The properties following describe what a `vertex` element looks like. Each `vertex` is declared to consist of 3 scalar properties, named `x`, `y` and `z`. Each scalar property is declared to be of type `float`.

Scalar types can be any of the following: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `float32`, `float64`, `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `float`, `double`. They consist of signed and unsigned integer types of sizes 8, 16 and 32 bits, as well as floating point types of 32 and 64bits.

Next, the `face` element type is declared, of which only 1 instance will be given. This element consists of a `list` property, named `vertex_indices`. Lists are sequences on which the first value, the

length, gives the number of remaining values. List properties are described by the scalar type of their length field and the scalar type of the remaining fields. In the case of `vertex_indices`, the length field is of type `uchar` and the remaining values are of type `int`.

Following the header, come the elements, in the order they were declared in the header. First come the 3 elements of type `vertex`, each represented by the value of their properties `x`, `y` and `z`. Then comes the single `face` element, composed by a single list of type `vertex_indices` containing 3 values (0 1 2).

How to read a file with RPly

Most users that want to read a PLY file already know which elements and properties they are interested in. In the following example, we will implement a simple program that dumps the contents of a PLY file to the terminal, in a different, simpler format that only works for triangles.

This simple format has a header that gives the number of vertices in the first line and the number of triangles in the second line. Following the header come the vertices, and finally the triangles. Here is the sample code for the program:

```
#include <stdio.h>
#include "rply.h"

static int vertex_cb(p_ply_argument argument) {
  long eol;
  ply_get_argument_user_data(argument, NULL, &eol);
  printf("%g", ply_get_argument_value(argument));
  if (eol) printf("\n");
  else printf(" ");
  return 1;
}

static int face_cb(p_ply_argument argument) {
  long length, value_index;
  ply_get_argument_property(argument, NULL, &length, &value_index);
  switch (value_index) {
    case 0:
    case 1:
      printf("%g ", ply_get_argument_value(argument));
      break;
    case 2:
```

```
      printf("%g\n", ply_get_argument_value(argument));
      break;
    default:
      break;
  }
  return 1;
}

int main(void) {
  long nvertices, ntriangles;
  p_ply ply = ply_open("input.ply", NULL, 0, NULL);
  if (!ply) return 1;
  if (!ply_read_header(ply)) return 1;
  nvertices = ply_set_read_cb(ply, "vertex", "x", vertex_cb, NULL, 0);
  ply_set_read_cb(ply, "vertex", "y", vertex_cb, NULL, 0);
  ply_set_read_cb(ply, "vertex", "z", vertex_cb, NULL, 1);
  ntriangles = ply_set_read_cb(ply, "face", "vertex_indices", face_cb, NULL, 0);
  printf("%ld\n%ld\n", nvertices, ntriangles);
  if (!ply_read(ply)) return 1;
  ply_close(ply);
  return 0;
}
```

RPly uses callbacks to pass data to an application. Independent callbacks can be associated with each property of each element. For scalar properties, the callback is invoked once for each instance. For list properties, the callback is invoked first with the number of entries in the instance, and then once for each of the data entries. *This is exactly the order in which the data items appear in the file.*

To keep things simple, values are always passed as `double`, regardless of how they are stored in the file. From its parameters, callbacks can find out exactly which part of the file is being processed (including the actual type of the value), plus access custom information provided by the user in the form of a pointer and an integer constant.

In our example, we start with a call to `ply_open` to open a file for reading. Then we get RPly to parse it's header, with a call to `ply_read_header`. After the header is parsed, RPly knows which element types and properties are available. We then set callbacks for each of the `vertex` element properties and the `face` property (using `ply_set_read_cb`). Finally, we invoke the main RPly reading function, `ply_read`. This function reads all data in the file, passing the data to the appropriate callbacks. After all reading is done, we call `ply_close` to release any resources used by RPly.

There are some details, of course. `Ply_set_read_cb` returns the number of instances of the target

property (which is the same as the number of element instances). This is how the program obtains the number of vertices and faces in the file.

RPly lets us associate one pointer *and* one integer to each callback. We are free to use either or both to link some context to our callbacks. Our example uses the integer placeholder to tell `vertex_cb` that it has to break the line after the `z` property (notice the last argument of `ply_set_read_cb`).

`Vertex_cb` gets the user data and the property value from it's argument and prints accordingly. The `face_cb` callback is a bit more complicated because lists are more complicated. Since the simple file format only supports triangles, it only prints the first 3 list values, after which it breaks the line.

The output of the program, as expected, is:

```
3
1
-1 0 0
 0 1 0
 1 0 0
 0 1 2
```

Writing files with RPly

The next example is somewhat more involved. We will create a program that converts our simple PLY file to binary mode. Besides showing how to write a PLY file, this example also illustrates the query functions. We do not know a priori which elements and properties, comments and obj_infos will be in the input file, so we need a way to find out. Although our simple program would work on any PLY file, a better version of this program is available from the RPly distribution. For simplicity, the simple version omits error messages and command line parameter processing.

In practice, writing a file is even easier than reading one. First we create a file in binary mode, with a call to `ply_create` (notice the argument `PLY_LITTLE_ENDIAN` that gives the storage mode). Then, we define the elements using `ply_add_element`. After each element, we define its properties using `ply_add_scalar_property` or `ply_add_list_property`. When we are done with elements and properties, we add comments and obj_infos. We then write the header with `ply_write_header` and send all data items. The data items are sent one by one, with calls to `ply_write`, *in the same order they are to appear in the file*. Again, to simplify things, this function receives data as `double` and performs the needed conversion. Here is the code for the example:

```
#include <stdio.h>
#include "rply.h"


static int callback(p_ply_argument argument) {
  void *pdata;
  /* just pass the value from the input file to the output file */
  ply_get_argument_user_data(argument, &pdata, NULL);
  ply_write((p_ply) pdata, ply_get_argument_value(argument));
```

```
  return 1;
}

static int setup_callbacks(p_ply iply, p_ply oply) {
  p_ply_element element = NULL;
  /* iterate over all elements in input file */
  while ((element = ply_get_next_element(iply, element))) {
    p_ply_property property = NULL;
    long ninstances = 0;
    const char *element_name;
    ply_get_element_info(element, &element_name, &ninstances);
    /* add this element to output file */
    if (!ply_add_element(oply, element_name, ninstances)) return 0;
    /* iterate over all properties of current element */
    while ((property = ply_get_next_property(element, property))) {
      const char *property_name;
      e_ply_type type, length_type, value_type;
      ply_get_property_info(property, &property_name, &type,
          &length_type, &value_type);
      /* setup input callback for this property */
      if (!ply_set_read_cb(iply, element_name, property_name, callback,
          oply, 0)) return 0;
      /* add this property to output file */
      if (!ply_add_property(oply, property_name, type, length_type,
          value_type)) return 0;
    }
  }
  return 1;
}

int main(int argc, char *argv[]) {
  const char *value;
  p_ply iply, oply;
  iply = ply_open("input.ply", NULL, 0, NULL);
  if (!iply) return 1;
```

```
  if (!ply_read_header(iply)) return 1;
  oply = ply_create("output.ply", PLY_LITTLE_ENDIAN, NULL, 0, NULL);
  if (!oply) return 1;
  if (!setup_callbacks(iply, oply)) return 1;
  /* pass comments and obj_infos from input to output */
  value = NULL;
  while ((value = ply_get_next_comment(iply, value)))
    if (!ply_add_comment(oply, value)) return 1;
  value = NULL;
  while ((value = ply_get_next_obj_info(iply, value)))
    if (!ply_add_obj_info(oply, value)) return 1;;
  /* write output header */
  if (!ply_write_header(oply)) return 1;
  /* read input file generating callbacks that pass data to output file */
  if (!ply_read(iply)) return 1;
  /* close up, we are done */
  if (!ply_close(iply)) return 1;
  if (!ply_close(oply)) return 1;
  return 0;
}
```

RPly uses iterators to let the user loop over a PLY file header. A function is used to get the first item of a given class (element, property etc). Passing the last returned item to the same function produces the next item, until there are no more items. Examples of iterator use can be seen in the `main` function, which uses them to loop over comments and obj_infos, and in the `setup_callbacks` function, which loops over elements and properties.

In the `setup_callbacks` function, for each element in the input, an equivalent element is defined in the output. For each property in each element, an equivalent property is defined in the output. Notice that the same callback is specified for all properties. It is given the output PLY handle as the context pointer. Each time it is called, it passes the received value to `ply_write` on the output handle. It is as simple as that.

A note on locale

ASCII PLY files are supposed to use the `C` locale for numeric formatting. RPly relies on library functions (such as `fprintf` and `strtod`) that are affected by the current locale. If your software modifies the locale (or if it uses another library/toolkit that does) and you use RPly under the modified locale, you may be unable to read or write properly formatted ASCII PLY files.

Modifying RPly internally to hedge against different locales would be complicated, particularly in multi-threaded applications. Therefore, RPly leaves this as your responsibility. To protect against locale

problems in the simplest scenario, you should bracket RPly I/O as follows:

#include <locale.h>

```
/* Save application locale */
const char *old_locale = setlocale(LC_NUMERIC, NULL);
/* Change to PLY standard */
setlocale(LC_NUMERIC, "C");
/* Use the RPly library */
⋮
/* Restore application locale when done */
setlocale(LC_NUMERIC, old_locale);
```

Reference Manual

```
p_ply ply_open(const char *name, p_ply_error_cb error_cb, long idata, void *pdata)
```

Opens a PLY file for reading, checks if it is a valid PLY file and returns a handle to it.

`Name` is the file name, and `error_cb` is a function to be called when an error is found. Arguments `idata` and `pdata` are available to the error callback via the `ply_get_ply_user_data` function. If `error_cb` is NULL, the default error callback is used. It prints a message to the standard error stream.

Returns a handle to the file or NULL on error.

Note: `Error_cb` is of type `void (*p_ply_error_cb)(p_ply ply, const char *message)`.

```
p_ply ply_open_from_file(FILE *file_pointer, p_ply_error_cb error_cb, long idata,
            void *pdata)
```

Checks if the FILE pointer points to a valid PLY file and returns a handle to it. The handle can be used wherever a handle returned by `ply_open` is accepted.

`File_pointer` is the FILE pointer open for reading, and `error_cb` is a function to be called when an error is found. Arguments `idata` and `pdata` are available to the error callback via the `ply_get_ply_user_data` function. If `error_cb` is NULL, the default error callback is used. It prints a message to the standard error stream.

Returns a handle to the file or NULL on error.

Note: `Error_cb` is of type `void (*p_ply_error_cb)(p_ply ply, const char *message)`.

Note: This function is declared in header `rplyfile.h`.

```
int ply_get_ply_user_data(p_ply_ply ply, void *pdata, long *idata)
```

Retrieves user data from the ply handle.

`Ply` is the handle passed to the error callback. `Pdata` receives the user data pointer. `Idata` receives the user data integer. `Pdata` and `idata` can be NULL.

Returns 1 in case of success, 0 otherwise.

```
int ply_read_header(p_ply ply)
```

Reads and parses the header of a PLY file. After a call to this function, the query functions `ply_get_next_element`, `ply_get_next_property`, `ply_get_next_comment`, and `ply_get_next_obj_info` can be called. Callbacks can also be set with the `ply_set_read_cb` function.

`Ply` is a handle returned by `ply_open`.

Returns 1 in case of success, 0 otherwise.

```
long ply_add_comment(
            p_ply ply,
            const char *element_name,
            const char *property_name,
            p_ply_read_cb read_cb,
            void *pdata,
            long idata
            )
```

Sets up the callback to be invoked when the value of a property is read.

`Ply` is a handle returned by `ply_open`. `Element_name` and `property_name` are the names of the element and property of interest. `Read_cb` is the callback function. `Pdata` and `idata` are user data to be passed to the callback function.

Returns the number of instances of the element of interest.

Note: `Read_cb` is of type `int (*p_ply_read_cb)(p_ply_argument argument)`. The callback should return 1 to continue the reading process, or return 0 to abort.

```
int ply_add_comment(
            p_ply_argument argument,
            p_ply_element *element,
            long *instance_index
            )
```

Retrieves element information from the callback argument.

`Argument` is the handle passed to the callback. `Element` receives a handle to the element originating the callback. `Instance_index` receives the index of the instance of the element being read. `Element` and `instance_index` can be NULL.

Returns 1 in case of success, 0 otherwise.

Note: further information can be obtained from `element` with a call to ply_get_element_info.

```
int ply_get_argument_property(
            p_ply_argument argument,
            p_ply_property *property,
            long *length,
            long *value_index
            )
```

Retrieves property information from the callback argument.

`Argument` is the handle passed to the callback. `Property` receives a handle to the property originating the callback. `Length` receives the number of values in the list property (1 for scalar properties). `Value_index` receives the index of the current property entry (0 for scalar properties, -1 for the first value of a list property, the one that gives the number of entries). `Property`, `length` and `value_index` can be NULL.

Returns 1 in case of success, 0 otherwise.

Note: further information can be obtained from `property` with a call to ply_get_property_info.

```
int ply_get_argument_user_data(p_ply_argument argument, void *pdata, long *idata)
```

Retrieves the user data from the callback argument.

`Argument` is the handle passed to the callback. `Pdata` receives the user data pointer. `Idata` receives the user data integer. `Pdata` and `idata` can be NULL.

Returns 1 in case of success, 0 otherwise.

```
double ply_get_argument_value(p_ply_argument argument)
```

Retrieves the property value from the callback argument.

`Argument` is the handle passed to the callback.

Returns the property value.

```
int ply_read(p_ply ply)
```

Reads all data in file, calling appropriate callbacks.

`Ply` is a handle returned by `ply_open`.

Returns 1 in case of success, 0 otherwise.

```
p_ply_element ply_get_next_element(p_ply ply, p_ply_element last)
```

Iterates over all elements on the header of a PLY file.

`Ply` is a handle returned by `ply_open`. `Ply_read_header` must have been called on the handle otherwise no elements will be found. `Last` is NULL to retrieve the first element, and an element to retrieve the next element.

Returns the next element, or NULL if no more elements.

Note: further information can be obtained from an element with a call to ply_get_element_info.

```
p_ply_property ply_get_next_property(p_ply_element element, p_ply_property last)
```

Iterates over all properties of an element.

`Element` is an element handle. `Last` is NULL to retrieve the first property, and a property to retrieve the next property.

Returns the next property, or NULL if no more properties.

Note: further information can be obtained from a property with a call to ply_get_property_info.

```
const char *ply_get_next_comment(p_ply ply, const char *last)
```

`Iterates over all comments on the header of a PLY file.`

`Ply` is a handle returned by `ply_open`. `Ply_read_header` must have been called on the handle otherwise no comments will be found. `Last` is NULL to retrieve the first comment, and a comment to retrieve the next comment.

Returns the next comment, or NULL if no more comments.

```
const char *ply_get_next_obj_info(p_ply ply, const char *last)
```

Iterates over all obj_infos on the header of a PLY file.

`Ply` is a handle returned by `ply_open`. `Ply_read_header` must have been called on the handle otherwise no obj_infos will be found. `Last` is NULL to retrieve the first obj_info, and a obj_info to retrieve the next obj_info.

Returns the next obj_info, or NULL if no more obj_infos.

```
int ply_get_element_info(p_ply_element element, const char** name, long
            *ninstances)
```

Retrieves information from an element handle.

`Element` is the handle of the element of interest. `Name` receives the internal copy of the element name. `Ninstances` receives the number of instances of this element in the file. Both `name` and `ninstances` can be NULL.

Returns 1 in case of success, 0 otherwise.

```
int ply_get_property_info(
            p_ply_property property,
            const char** name,
            e_ply_type *type,
            e_ply_type *length_type,
            e_ply_type *value_type
            )
```

Retrieves information from a property handle.

`Property` is the handle of the property of interest. `Name` receives the internal copy of the property name. `Type` receives the property type. `Length_type` receives the scalar type of the first entry in a list property (the one that gives the number of entries). `Value_type` receives the scalar type of the remaining list entries. `Name`, `type`, `length_type`, and `value_type` can be NULL.

Returns 1 in case of success, 0 otherwise.

Note: `Length_type` and `value_type` can receive any of the constants for scalar types defined in `e_ply_type`. `Type` can, in addition, be `PLY_LIST`, in which case the property is a list property and the fields `length_type` and `value_type` become meaningful.

```
p_ply ply_create(const char *name, e_ply_storage_mode storage_mode, p_ply_error_cb
            error_cb)
```

Creates a PLY file for writing.

`Name` is the file name, `storage_mode` is the file storage mode (`PLY_ASCII`, `PLY_LITTLE_ENDIAN`, `PLY_BIG_ENDIAN`, or `PLY_DEFAULT` to automatically detect host endianess). `Error_cb` is a function to be called when an error is found. Arguments `idata` and `pdata` are available to the error callback via the `ply_get_ply_user_data` function. If `error_cb` is NULL, the default error callback is used. It prints a message to the standard error stream.

Returns a handle to the file or NULL on error.

Note: `Error_cb` is of type `void (*p_ply_error_cb)(const char *message)`

```
p_ply ply_create_to_file(FILE *file_pointer, e_ply_storage_mode storage_mode,
            p_ply_error_cb error_cb)
```

Creates a PLY file to be written to a FILE pointer and returns a handle to it. The handle can be used wherever a handle returned by `ply_create` is accepted.

`File_pointer` a pointer to a file open for writing, `storage_mode` is the file storage mode (`PLY_ASCII`, `PLY_LITTLE_ENDIAN`, `PLY_BIG_ENDIAN`, or `PLY_DEFAULT` to automatically detect host endianess). `Error_cb` is a function to be called when an error is found. Arguments `idata` and `pdata` are available to the error callback via the `ply_get_ply_user_data` function. If `error_cb` is NULL, the default error callback is used. It prints a message to the standard error stream.

Returns a handle to the file or NULL on error.

Note: `Error_cb` is of type `void (*p_ply_error_cb)(const char *message)`

Note: This function is declared in header `rplyfile.h`.

```
int ply_add_element(p_ply ply, const char *name, long ninstances)
```

Adds a new element to the ply file.

`Ply` is a handle returned by `ply_create`, `name` is the element name and `ninstances` is the number of instances of this element that will be written to the file.

Returns 1 in case of success, 0 otherwise.

```
int ply_add_property(
            p_ply ply,
            const char *name,
            e_ply_type type,
            e_ply_type length_type,
            e_ply_type value_type
            )
```

Adds a new property to the last element added to the ply file.

`Ply` is a handle returned by `ply_create` and `name` is the property name. `Type` is the property type. `Length_type` is the scalar type of the first entry in a list property (the one that gives the number of entries). `Value_type` is the scalar type of the remaining list entries. If `type` is not `PLY_LIST`, `length_type` and `value_type` are ignored.

Returns 1 in case of success, 0 otherwise.

Note: `Length_type` and `value_type` can be any of the constants for scalar types defined in `e_ply_type`. `Type` can, in addition, be `PLY_LIST`, in which case the property is a list property and the fields `length_type` and `value_type` become meaningful.

```
int ply_add_list_property(
            p_ply ply,
            const char *name,
            e_ply_type length_type,
            e_ply_type value_type
            )
```

Same as ply_add_property if type is PLY_LIST.

```
int ply_add_scalar_property(p_ply ply, const char *name, e_ply_type type)
```

Same as ply_add_property if type is *not* PLY_LIST.

```
int ply_add_comment(p_ply ply, const char *comment);
```

Adds a comment to a PLY file.

Ply is a handle returned by ply_create and comment is the comment text.

Returns 1 in case of success, 0 otherwise.

```
int ply_add_obj_info(p_ply ply, const char *obj_info);
```

Adds a obj_info to a PLY file.

Ply is a handle returned by ply_create and obj_info is the obj_info text.

Returns 1 in case of success, 0 otherwise.

```
int ply_write_header(p_ply ply);
```

Writes the PLY file header to disk, after all elements, properties, comments and obj_infos have been added to the handle.

Ply is a handle returned by ply_create and comment is the comment text.

Returns 1 in case of success, 0 otherwise.

```
int ply_write(p_ply ply, double value);
```

Passes a value to be stored in the PLY file. Values must be passed in the order they will appear in the file.

Ply is a handle returned by ply_create and value is the value to be stored. For simplicity, values are always passed as double and conversion is performed as needed.

Returns 1 in case of success, 0 otherwise.

```
int ply_close(p_ply ply);
```

Closes the handle and ensures that all resources have been freed and data have been written.

Ply is a handle returned by ply_create or by ply_open.

Returns 1 in case of success, 0 otherwise.