Writing Multiple Bit-Depth Graphics Code Using C++ Templates

Jonathan Shekter
Adobe Systems, Inc.

Starting with version 5.0, After Effects supports 16 bit per channel (16bpc) compositing. Effects plug-in writers can take advantage of this new functionality by writing 16bpc capable effects. However, all effects must also work in on 8bpc images. Therefore, effects which take advantage of the new higher bit depth must also contain code to perform the same image processing operation on two different pixel formats.

In most cases, the algorithm used for both 8 and 16 bit-per-channel processing will be identical. Ideally, one would like to write (and maintain) only one copy of any given algorithm. This white paper describes how C++ template programming techniques may be used to achieve this.

Why Templates?

There a number of approaches to duplicating program logic for different bit depths. Non-template approaches include:

- Copy and paste. This is the simplest of all possible methods, in terms of conceptual complexity. In practice it suffers in many ways. For instance, the conversion of the copied code from one bit depth to another must be done by hand, which is tedious and error prone. Worst of all, it requires maintenance of two logically identical bodies of code. Not only is this time-consuming, but it is problematic in that there is no guarantee that both copies really embody the same algorithm at any given point during development.
- Use of the C preprocessor. The most obvious technique in this class would be to describe an entire function as a #define macro, with a few parameters representing basic types and constants which differ based on bit-depth. Needless to say, this is very awkward.
- A more sophisticated preprocessor method involves writing a separate file which is #included in a main source file after defining macros for types, constants, accessor and arithmetic functions, etc. This overcomes the problems of the copy & paste approach, in that there is only one source level implementation of any given function, but it is still somewhat awkward. Besides being hard to read (in that the multiple bit-depth code cannot be written "normally") some debuggers cannot step through code written in this way.
- From an object-oriented viewpoint, pixels can be viewed as polymorphic objects which support a certain set of operations. C++ supports polymorphism through virtual functions, and one can define pixel objects through an abstract base class with virtual functions to perform basic processing and arithmetic operations. This is standard, simple, and elegant, but it exacts a heavy performance penalty. Namely, every pixel object must carry a virtual function table pointer, which is a huge overhead for a small pixel structure, and all arithmetic must be performed through virtual function calls, which are very expensive for operations which might otherwise require only a single machine instruction.

Thus, without templates, there is no technique which avoids duplicate code while remaining simple, standard, maintainable, and highly efficient.

Basic Template Usage

The basics of template syntax and usage can be found in any good C++ book. To apply this to the problem at hand, consider the following function which performs some image processing operation on a pixel buffer:

```
void ProcessPixels(PF_Pixel *in, int len, PF_Pixel *out)
{
  PF_Pixel *ptr = in;
  while (len--) {
            // Do something to pixels…
  }
}
```

This is what a scanline processing routine written for AE's usual 8 bit-per-channel PF_Pixel type might look like. The 16 bit version of this code should perform the logically identical operation, but on a different pixel type. Such a routine might look like this:

```
void ProcessPixels(PF_Pixel16 *in, int len, PF_Pixel16 *out)
{
  PF_Pixel16 *ptr = in;
  while (len--) {
            // Do something to pixels…
  }
}
```

(Note that we haven't changed the actual name of the routine, in other words were using function overloading here. The compiler will automatically determine the correct function to invoke whenever this symbol is called, based on the types of the arguments. This property will prove very useful in templated code.)

So far so good, but as discussed above we'd really like to avoid manually duplicating code. This can be accomplished by replacing both of the above functions with the following templated definition:

```
template <typename PIXTYPE>
void ProcessPixels(PIXTYPE *in, int len, PIXTYPE *out)
{
  PIXTYPE *ptr = in;
  while (len--) {
            // Do something to pixels…
  }
}
```

The PIXTYPE symbol is known as a "template parameter" and stands for an unknown type. After the "template" line, PIXTYPE acts just like a typedef. Parameters and return values may be declared to have this type, and it can be used for local variable definitions within the body of the function. However, no object code is actually generated when the compiler encounters this definition. Only when this function is actually called will code be generated. At that point, the type of the PIXTYPE parameter is determined based on the arguments to the call, and object code will be produced for a function which acts on that type. This process is called "instantiation" and is usually handled automatically by the compiler.

Within an After Effects plug-in, one might call the above function as follows:

```
PF_World *input_world, *output_world;
if (PF_WORLD_IS_DEEP(input_world))
            ProcessPixels(
                        PF_GET_PIXEL_DATA16(input_world),
                        input_world->width,
                        PF_GET_PIXEL_DATA16(output_world));
else
            ProcessPixels(
                        PF_GET_PIXEL_DATA8(input_world),
                        input_world->width,
                        PF_GET_PIXEL_DATA8(output_world));
```

Because the PF_GET_PIXEL_DATA8 and PF_GET_PIXEL_DATA16 macros evaluate to an expression of type PF_Pixel8* and PF_Pixel16*, respectively, the compiler instantiates ProcessPixels() on these two types. Note that each templated function is instantiated at most once for each type; all calls with the same argument types will bind to the same instantiation. The net result is that the same source code function generates exactly as many different object code functions as there are different pixel types, no matter how many times it is called, which is exactly the desired behavior.

Using an Unknown Type

Suppose one wanted to write a function that performs clipped addition of a span of pixel values. An 8-bit version might look like this:

```
void AddSpan(PF_Pixel *in, int len, PF_Pixel *out)
{
  while (len--) {
            out->red = min((int)in->red + (int)out->red, 255)
            .
            .
            .
  }
}
```

This can be templated easily enough, but different pixel types will have different maximum values, so the constant 255 in the above function will be incorrect. More generally, this sort of problem occurs whenever merely changing the type of the pixel parameters and variable is not sufficient to convert an algorithm from 8 to 16 bits.

Using Unknown Types Through Overloaded Functions

One solution to this problem involves defining overloaded functions to perform bit-depth specific manipulations. For example, one could write:

```
// 8 bit version
inline void AddPixel(PF_Pixel8 *src, PF_Pixel8 *dst)
{
  out->red   = min((int)in->red   + (int)out->red,   255)
  out->green = min((int)in->green + (int)out->green, 255)
  out->blue  = min((int)in->blue  + (int)out->blue,  255)
  out->alpha = min((int)in->alpha + (int)out->alpha, 255)
}

// 16 bit version
inline void AddPixel(PF_Pixel16 *src, PF_Pixel16 *dst)
{
  out->red   = min((int)in->red   + (int)out->red,   32768)
  out->green = min((int)in->green + (int)out->green, 32768)
  out->blue  = min((int)in->blue  + (int)out->blue,  32768)
  out->alpha = min((int)in->alpha + (int)out->alpha, 32768)
}
```

Then, a templated span addition routine can be written by exploiting C++'s function overload mechanism:

```
template <typename PIXTYPE>
void AddSpan(PIXTYPE *in, int len, PIXTYPE *out)
{
  while (len--) {
            AddPixel(in, out);
            in++;
            out++;
  }
}
```

When AddSpan() is instantiated for a particular pixel type, the compiler will automatically determine which version of AddPixel() to call based on the type of the pointers passed to it, which in turn will have type corresponding to the template parameter PIXTYPE.

This works, and is elegant and simple when the required code varies greatly between pixel types. A good example of this would be an algorithm where some computational step is performed by accessing a look-up-table when operating on 8 bit pixels, but computed directly for 16 bit pixels (presumably because a 32769 element table would take too much memory.) By encapsulating this step into two different overloaded functions, the algorithmic difference between the two bit depths can be hidden from the main processing.

However, when the 8 and 16 bit versions of some operation are very similar, as in the case above, it's not an entirely satisfactory solution. For instance, the same operation is often performed on multiple image channels, so it would be nice to encapsulate the differences in processing at the channel level, rather than the pixel level. One might try overloading on channel type:

```
// DANGEROUS! Don't do this! See text below.
inline uchar AddChannel(uchar a, uchar b)
{
  return min((int)a + (int)b, 255);  // 8 bit version
}

// DANGEROUS! Don't do this! See text below.
inline ushort AddChannel(ushort a, ushort b)
{
  return min((int)a + (int)b, 32768);          // 16 bit version
}
```

but this is a very bad idea. The problem is that channel values aren't always represented by the "obvious" type. Larger intermediate types are required for many types of computation (as in the casts to int in the additions above) and there's no easy way to guarantee that no one will ever store your 8-bit pixel value in a 16-bit type. Consider:

```
template <typename PIXTYPE>
void DoSomething(PIXTYPE *ptr)
{
  ushort stored_red  = ptr->red;
  ushort stored_blue = ptr->blue;
  ptr->green = AddChannel(stored_red, stored_blue);
}
```

In this example, the arguments to the AddChannel() call will always be of type ushort, no matter what size pixels this function is actually instantiated on. Thus, the 16-bit version of AddChannel() will be called even when processing 8 bit pixels! The situation gets even more confusing when the arguments to AddChannel() have different types. In such cases, at best you'll get an error message from the compiler complaining about ambiguous function overload resolution. At worst, C/C++'s integral type promotion rules will be invoked silently and the wrong function will be called. Of course, any individual line of code can be fixed to overcome these problems, but the general technique is very error-prone and fragile.

In summary, performing some operation on an unknown (template parameter) type by function overloading is an excellent strategy when entire pixels are being processed substantially differently depending on bit depth, but is inelegant or downright dangerous for more atomic operations such as channel arithmetic.

Traits Classes

The above problems with AddSpan() could be solved if there were some way to know the maximum channel value within the function in a parameterized way. More generally, there are many things one would like to associate with a given pixel type, such as:

- bit depth
- minimum, maximum, and half channel values
- channel type
- intermediate types (useful for channel arithmetic)
- optimized functions for basic channel arithmetic (multiplication, etc.)

Although not currently supported in After Effects, this list could even include things like number of channels, color-space, whether the alpha is straight or premultiplied, etc.

All of this "stuff" can be stored in a "traits" class, a concept first developed for use in the C++ standard library (formerly known as the Standard Template Library.) This involves defining a structure which holds all these constants, types, and functions:

```
struct PF_Pixel8_Traits {
  enum { bit_depth = 8; }
  enum { max_value = 255; }
  typedef uchar channel_type;
  static uchar Multiply(uchar a, uchar b);
  ...
};

struct PF_Pixel16_Traits {
  enum { bit_depth = 16; }
  enum { max_value = 32768; }
  typedef ushort channel_type;
  static ushort Multiply(ushort a, ushort b);
  ...
};
```

Note that these structures have no data and therefore no non-static methods. They are merely a description of a particular type. Because of this, passing a traits object to a function is usually a no-op, depending on the compiler's optimizer. With such a structure in hand, all the relevant particulars about a pixel type can be conveniently packaged into a single parameter, like this:

```
template <typename PIXTYPE, typename TRAITS>
void ProcessPixels(PIXTYPE *in, int len, PIXTYPE *out, TRAITS traits_obj)
{
  // Declare local variables of channel type
  // (C++ standard says we need "typename" keyword here to denote
  //  a template-derived type, and some compilers complain without it.)
  typename TRAITS::channel_type red, green, blue, alpha;
  .
  .
  while (len--) {
    .
    .
    // Clip to pixel type's max value
    out->red = min((int)in->red + (int)out->red, TRAITS::max_value);
    .
    .
    // Multiply two channels together
    product = TRAITS:Multiply(in->red, in->alpha);
    .
    .
    // etc.
  }
}
```

This shows declaration of variables which have type dependent on the traits class, and access of constants and functions within the traits class. Because all channel arithmetic function calls are now methods of the traits class, there is no integral type ambiguity as there was with overloaded functions in the global scope. Note that the actual traits_obj is never used, only its type matters. This routine would be called as follows:

```
PF_World *input_world, *output_world;
if (PF_WORLD_IS_DEEP(input_world))
        ProcessPixels(
                PF_GET_PIXEL_DATA16(input_world),
                input_world->width,
                PF_GET_PIXEL_DATA16(output_world),
                PF_Pixel16_Traits());
else
        ProcessPixels(
```

```
                PF_GET_PIXEL_DATA8(input_world),
                input_world->width,
                PF_GET_PIXEL_DATA8(output_world),
                PF_Pixel8_Traits());
```

Associating Traits With Types

This is effective, but still somewhat awkward. In particular a traits object must be passed down the call chain through all templated functions. This makes things like callbacks or function pointers difficult to use. Also, there is no guarantee that traits object will always match the pixel type; it's perfectly possible to pass a PF_Pixel8_Traits object to a routine which is processing 16 bit pixels. The resulting object code will be totally incorrect but will probably compile without generating any warnings.

What is needed is a way to associate a traits object with pixel types. Then, this association could be invoked within each function, removing the need to pass an empty object down the call stack and preventing type mismatch errors.

This can be accomplished through the mechanism of class template specialization. Rather than defining the traits class for each pixel type as a separate class, they are defined as specializations of some (never used, undefined) templated class definition:

```
// template class declaration; no default definition provided
template <typename PIXTYPE>
struct PixelTraits;

// Specialize on PF_Pixel8 to define 8 bpc traits
template <>
struct PixelTraits<PF_Pixel8> {
  enum { bit_depth = 8; }
  enum { max_value = 255; }
  typedef uchar channel_type;
  static uchar Multiply(uchar a, uchar b);
  ...
};

// Specialize on PF_Pixel16 to define 16 bpc traits
template <>
struct PixelTraits<PF_Pixel16> {
  enum { bit_depth = 16; }
  enum { max_value = 32768; }
  typedef ushort channel_type;
  static ushort Multiply(ushort a, ushort b);
  ...
};
```

We can now refer to the traits class for a template parameter PIXTYPE by writing "PixTraits<PIXTYPE>". This name is valid at any scope, including inside a function, so
there is no need to actually pass an object of this type into pixel processing functions; the passed object wasn't actually used anyway. AddSpan() now looks like this:

```
template <typename PIXTYPE>
void AddSpan(PIXTYPE *in, int len, PIXTYPE *out)
{
  while (len--) {
            out->red   = min((int)in->red   + int)out->red,
                                    PixelTraits<PIXTYPE>::max_value);
            out->green = min((int)in->green + int)out->green,
                                    PixelTraits<PIXTYPE>::max_value);
            out->blue  = min((int)in->blue  + int)out->blue,
                                    PixelTraits<PIXTYPE>::max_value);
            out->alpha = min((int)in->alpha + int)out->alpha,
                                    PixelTraits<PIXTYPE>::max_value);
```

```
            in++;
            out++;
  }
}
```

Ideally, we'd add an "AddChannel" method to the traits class, which performs the appropriate clipped addition. Along with shorter symbol names ("PT" for "PixelTraits" and "P" for "PIXTYPE") complex multiple bit-depth code can become quite readable:

```
template <typename P>
void AddSpan(P *in, int len, P *out)
{
  while (len--) {
            out->red   = PT<P>::AddChannel(in->red   , out->red  );
            out->green = PT<P>::AddChannel(in->green , out->green);
            out->blue  = PT<P>::AddChannel(in->blue  , out->blue );
            out->alpha = PT<P>::AddChannel(in->alpha , out->alpha);
            in++;
            out++;
  }
}
```

When this function is instantiated, the compiler will resolve the symbol "PT<P>" to the correct traits class, and use this to select the types, constants, and functions appropriate to the current pixel type. Because all of this is done based on static type at compile time, this process invokes no run-time overhead. With the inlining of pixel arithmetic methods, the code generated is perfectly optimal, in the sense that it is just as efficient as pre-processor generated functions or hand-coded cut & paste routines.

Summary

C++'s function template mechanism seems ideally suited to the problem of writing high-performance low-level graphics code which operates on multiple bit-depths. In particular, templates are specifically designed to produce multiple, individually optimized object-code routines to be generated from a single readable source-code function.

Unfortunately, applying templates to basic pixel structures is not as simple as it seems, because the concept of "bit depth" involves a lot more than just a single pixel type. In particular, numerous types, constants, and basic arithmetic functions are bit-depth specific.

Large algorithmic differences between bit depths can be effectively handled through function overloading. However this method is inadequate for handling for finer-grained operations, such as channel arithmetic, where the base pixel type is not explicitly used. The problem of inferring types, constants, and methods associated with the base pixel type is therefore best solved through the use of traits classes.

The resulting syntax is a little awkward in places, but overall templates provide the best tradeoff between performance and clarity. That is, the performance is optimal, and templates are much easier to write, read, debug, and maintain than other techniques such as preprocessor tricks.