Using PDL for Code Design and Documentation

by Drew Sikora

Introduction

After much procrastination, I finally got around to purchasing Code Complete at my local Barnes & Noble (gotta love that store...). Of course, in my usual habit of digesting about 4-5 computer books at once, I've only read about 125 pages. Anyway I was cruising through Section 4 on building routines when I came across something called PDL. Reading further I was amazed that someone had actually taken this technique and named it for professional use. We'll get to what PDL is in a moment, but first a little background.

Before this article I also wrote The Art of Code Documentation, which, I can't help but say, got rave reviews including an email from a guy at LithTech saying how they needed more people who followed these rules in the industry. If you read this article and disagree, please, please email me and let me know why. This includes all my other articles. I am extremely open to critiscm and have a seperate email account for my articles, so don't be afraid to clutter up my inbox. Anyway in this article I laid down the basics of how to comment and then document your code so others could read it. The best part about this article is (I think) the fact that it was from experience only, rather than a take-off of a section of a book, like this is. I highly suggest you read it first - I won't directly reference anything from it, but I will assume you know, are familiar with, and accept common documenting practices.

Now that we are all ready, let's begin!

What is PDL?

What is PDL? Well that's a reasonable enough question. I sure as hell had no idea what PDL was when I first started reading about it. In fact, the meaning of its name can be quite misleading. PDL stands for Program Design Language. Now you may notice how short this article is if you glance to the menu on the right. Obviously you can assume that PDL really isn't a full-fledged language with it's own functions and variables and such - no way could I teach it in such a simple lesson. There is a reference you can make to PDL - it is a common coding technique where you write out code in plain english, known as pseudocode, or fake code (literally). That's exactly what PDL is.

When we talk about high-level languages, we talk about languages where the instructions you type represent English (as an example) more and more. Believe it or not, but QBasic is a higher-level language than C++! In case you don't understand this, I'll sidetrack just a bit to make sure everyone is clear. When we reference languages in terms of levels, the reference is the English language. The closer the code comes to looking like English, the "higher" up it is. If we were to compare ways to print a word on the screen in C++ and QBasic, you'll realize why QBasic is a higher language than C++:

C++ - cout << "hello world" << endl;

QBasic - PRINT "hello world"

You tell me which is the higher level language. Obviously PRINT is much, much closer to the English word (okay, so it's exact) than cout is. To be official, the ratio of high-level-language statements to equivilent assembly code in QBasic is 1:5, while in C++ it's only 1:2.5

Okay, enough delineating - back to the main topic. So as I was saying, PDL is the highest-level language you will ever see until they conceive computers that can interpret human speech as code. When you code in PDL, you are coding in plain English (or whatever language you use) and not one single bit of computer code. In fact, this ambiguity let's you code PDL and then use it to construct a routine in any computer language since you never used a peice of language specific code. Wonderful! But how the hell is all this supposed to help you? Nothing could ever run it! Be patient, I haven't instructed you on how to use it yet, have I?

Using PDL Correctly

Like anything else in this freakish world, PDL can be easily abused. How sad. But to avoid this tragedy I'll cue you in on what to do and what not to do. Let's kick off this section by dumping you with an example of using PDL, then I'll break it down for you to understand, pointing out do's and dont's along the way. Shall we?

Here's a little excercise - look at examples A and B and see if you can discern which PDL code is correct and which is not.

Example A

Lock the drawing surface to prevent access by other programs and check for

errors.
Get mem_pitch of the surface using (int)ddsd.lPitch.
*Get *video_buffer to draw to using ddsd.lpSurface.*

if pixel_mode == random then

Create three random values for x, y and color.
else if pixel_mode == linear then
Increment x value by 1.
Create a random value for color.

end if
*video_buffer[x+y*mempitch] = color*

Unlock the surface and check for errors.
return 1

Example B
Prevent the drawing surface from being accessed by other programs while we draw.
Retrieve the memory pitch of the surface memory so we do not draw out of bounds.
Retrieve the location of the surface so that we can draw to it successfully.
If the pixel mode is set to random
Create three random values that we can assign to the x, y and color attributes of the pixel.

Else if the pixel mode is set to linear
Update the horizontal position of the pixel.
Create a random value for the color of the pixel.

End If
Plot the pixel on the screen.

Release the surface for general use once again.
Return TRUE that the pixel was plotted.

Obviously the above two code examples document the plotting of a pixel on the screen. I chose this for an example because it is a basic task a lot of you should be familiar with. Now, were you able to figure out which was the bad PDL and which was the good? If not go back and read the end of the first section again, where I clue you in on a few of things that you should not find in PDL. Even still, I think I made the distinction rather obvious.

The answer is Example B. Surprised? I hope not. Example A could not possibly fit my current description of PDL in any way whatsoever. I specifically pointed out how PDL is considered a high-level language, and some of the lines in Example A are anything but. Here are some of the things that make Example A a PDL nightmare:

- Although you may not realize it, the term "Lock a surface" can easily be attributed to DirectX and its Lock() function. The same applies to the term Unlock. Since we want our PDL to be as ambiguous as possible, Example B cleans this up by using the terms prevent and release. Anyone who wants to argue that release is also DirectX specific will break their noses running into a brick wall - in this context, release does not describe the freeing of COM components.
- Direct reference to variable names in Example A is also a no-no. And following close behind them are direct code examples! Ack! Once again ambiguity must reign supreme, and there's no way you can use that code in Visual Basic or any other language.
- Also present is the reference to a language-specific feature - the C pointer. This immediatly kills any sense of ambuguity since all other language options for writing this routine fall out.
- Then we have another direct code example for plotting the pixel, again unnacceptable.
- Finally we have the ending "return 0". Another language-limiting statement, and one that explains absolutly nothing of its true purpose.

After going through above points, and comparing them to Example B, you'll fully understand what PDL should look like. If you really think about it, you could code the pixel-plotting routine in any language capable of handling the task. Wonderful! Notice how the routine is nothing but English - nor variable names, no comparison signs, no code whatsoever. Of course, this brings us to a final point about good PDL use.

It's very easy to go overboard when writing in PDL. You may get a bit too descriptive and realize that coding what you wrote may be a nightmare! Therefore writing PDL can be considered an iterative process. Write it once, then start to break it down as far as you can go without quite reaching code level and while maintaining the basic rules of PDL.

PDL Uses
Part 1: Routine Review

Now thay we know how to construct PDL routines correctly, we can learn how best to use them. One of their uses is routine review. Project managers and lead programmers don't like to waste time going over endless lines of code to learn whether or not the routine does what they want it to, or what it's spec'd out to do. Consider how much easier it would be

C++ Home Menu

- Algorithms & Data Structures
- C++ Contests
- C++ Forums
- C++ Homeworks
- C/C++ discussion
- Games and Graphics
- Maths & Physics

Topic : Using PDL for Code Design and Documentation
Author : Drew Sikora
Page : << Previous **2**
Go to page : 1 Go

for a lead to be able to read through PDL and decide whether the programmer is on the right track. Of course this won't catch programming flaws, but programming flaws can be avoiding by finding design flaws - so it's all good.

On top of that is the ease of which you can modify PDL - anybody can do it! People who don't know jack about coding but are seasoned designers can easily change the wordings of a routine to better match specification. This is the cheapest and most effective way to change a program architecture. Think of how hard it would be to have to rip out a few lines of code that could easily constitute only one line of PDL. Saving money is always a good thing, and PDL can help you do that.

Part 2: Code Documentation
Ahh, here we go - the reason I decided to write this whole article in the first place. I think another acceptable translation of PDL would be Program Documenting Language. Why? Easy - check out what I did with Example B.

```
// Prevent the drawing surface from being accessed by other programs while we draw.
lpdds7->Lock(NULL, &ddsd, DDLFLAGS, NULL);

// Retrieve the memory pitch of the surface memory so we do not draw out of bounds.
int mempitch = (int)ddsd.lPitch;

// Retrieve the location of the surface so that we can draw to it successfully.
UCHAR *video_buffer   = (UCHAR *)ddsd.lpSurface;

// If the pixel mode is set to random
```

```
if (pixel_mode == RANDOM) {
  // Create three random values that we can assign to the x, y and color
attributes of the pixel.
  UCHAR color   = rand()%256;
  int x      = rand()%800;
  int y      = rand()%600;
}
//Else if the pixel mode is set to linear
else if (pixel_mode == LINEAR) {
  // Update the horizontal position of the pixel.
  pixel_x++;

  // Create a random value for the color of the pixel.
  UCHAR color   = rand()%256;
} // End If

// Plot the pixel on the screen.
video_buffer[x+y*mempitch] = color;

// Release the surface for general use once again.
lpdds7->Unlock(NULL);

// Return TRUE that the pixel was plotted.
return 1;
```

Check that out. In all my past experiences I've always coded first, commented second. Who'd ever thought the other way around was better? Obviously somebody did. What an easy way to code and comment at the same time. For all you people against taking the time to comment you code (you'd be surprised how many people actually hate to comment, check out this thread for proof), this is the excuse that you need. You don't even have to think of it as commenting, just write out the routine in PDL to get the architecture and implementation straight, then just code it and don't delete the PDL, because it turns right into comments!

Conclusion

Well that's about it. PDL, as you can see, is a tool than belongs in any designer's and coder's toolbox. I'm sure glad I stumbled upon it, because it's streamlined my coding a lot. Writing out a whole routine in PDL and then just adding the code after perfecting it has saved me countless hours. I hope I did a good job explaining to you the correct way to use PDL to save you time and effort. Any questions and/or comments can be directed to gaiiden@hotmail.com - I'd love to hear from you. Until next time...

Page : << Previous **2**

C++ Home Menu

- Algorithms & Data Structures
- C++ Contests
- C++ Forums