Pocket Programming Language Newsletter
February 2007

Welcome to the monthly PPL newsletter. The PPL newsletter is open to contributions, if you have an article, tips, advertising or anything else related to PPL, we will be very happy to include your submissions.

Table of content

Editorial

Wow a new year. The last one has gone by so fast for us. Even though we are about one month late, Happy New Year to all of our readers. Like we were saying, last year went by real fast and this new year didn't start slow either. We are working hard on PPL on a daily basis, the forums are doing great, our users are the cream of all users really. We are very proud of what we have achieved so far and we will do everything so that it becomes even better.

What's new

For a couple weeks now, we have made available several versions of PPL as beta. That means that users can download a beta version without having to wait for an official release. New features have been added to the beta and many bugs have been fixed. The new features that were added lately are quite impressive.

As you read this letter, chances are that you are trying out these new features. We have redesigned the PIDE interface with version 1.12 to follow Windows GUI guidelines more closely. The Visual Form Builder in the PIDE has been redesigned almost from scratch. The controls are now sorted by categories and displayed in a list format. You can now drag controls to the form to where you want them to be placed exactly. The control display has also been re-coded to display the real control image. It will now be possible to create custom controls in PPL that will be displayed in the editor exactly as they would in the result form during execution. The properties editor has been re-coded to offer a more up to date look and comes with custom editors for every property in the list. Custom controls can easily be coded through a standard PPL library. The PIDE will allow to import custom controls that will be available via our web site. And finally the property editor will offer better data type editing.

Version 1.13 of PPL will serve as a bridge version between 1.12 and 1.20. 1.20 will include more features like a RAPI library, a new installer that will support Smartphones, a Pocket Outlook library and file based levels in the Game Level Editor and so much more (nothing new, you are getting used to get new features all the time :)).

Coding Tips section:

Regular Expressions (Regex):

From www.regular-expressions.info:

*“A regular expression (regex or regexp for short) is a special text string for describing a search pattern. You can think of regular expressions as wildcards on steroids. You are probably familiar with wildcard notations such as *.txt to find all text files in a file manager. The regex equivalent is .*\.txt .*

But you can do much more with regular expressions. In a text editor like EditPad Pro or a specialized text processing tool like PowerGREP, you could use the regular expression \b[A-Z0-9._%-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b to search for an email address. Any email address, to be exact. A very similar regular expression (replace the first \b with ^ and the last one with $) can be used by a programmer to check if the user entered a properly formatted email address. In just one line of code, whether that code is written in Perl, PHP, Java, a .NET language or a multitude of other languages.”

PPL supports a variety of expressions like:

\	Quote the next metacharacter
^	Match the beginning of the string
.	Match any character
$	Match the end of the string
|	Alternation
()	Grouping (creates a capture)
[]	Character class

==GREEDY CLOSURES==

*	Match 0 or more times
+	Match 1 or more times
?	Match 1 or 0 times
{n}	Match exactly n times
{n,}	Match at least n times
{n,m}	Match at least n but not more than m times

==ESCAPE CHARACTERS==

\t	tab	(HT, TAB)
\n	newline	(LF, NL)
\r	return	(CR)
\f	form feed	(FF)

==PREDEFINED CLASSES==

\l	lowercase next char
\u	uppercase next char
\a	letters
\A	non letters
\w	alphanimeric [0-9a-zA-Z]

\W	non alphanimeric
\s	space
\S	non space
\d	digits
\D	non nondigits
\x	exadecimal digits
\X	non exadecimal digits
\c	control charactrs
\C	non control charactrs
\p	punctation
\P	non punctation

To search a string using regular expression in PPL you will use the Search() function. You can also make sure that the string is an exact match of the regular expression you are providing with the Match() function.

Let's take the following example:

```
string$ = "Bill Clinton";
expr$ = "^(Bill|George|Renald) (Clinton|Bush|Reagan)$";

Search(expr$, string$, b$, e$);
ShowMessage(b$ + "," + e$);
```

The expr$ variable contains an expression that says, if the first word is either Bill, George or Renald and that the string ends with Clinton, Bush or Reagan, we have a match.

“Bill Clinton” will be the beginning of our result string, b$ and “” will be our ending string e$.

```
i$ = 0;
while(i$ <= subexpcount - 1)
  subexp(string$, i$, begin$, len$);
  ShowMessage("SubExp " + i$ + " = " + begin$ + "," + len$);
  i$++;
end;
```

In the previous example, we check each sub expression to see what matched in the string string$ and where it started and how many characters the sub expression took from string$.

Sub expression 0 will return “Bill Clinton” for a length of 12 because it do the whole expression.
Sub expression 1 will return “Bill Clinton” but for 4 characters only, the first sub expression “^(Bill|George|Renald)” is analyzed.
Sub expression 2 will return “Clinton” for 7 characters, the second sub expression “(Clinton|Bush|Reagan)$” is analyzed.

Compressing and Encrypting strings:

The Pro version of PPL comes fully loaded with different compression and decompression methods from two simple functions: Compress() and UnCompress(). Here are the different compression methods supported by PPL:

_RLE

RLE, or Run Length Encoding, is a very simple method for lossless compression. It simply replaces repeated bytes with a short description of which byte to repeat, and how many times to repeat it. Though simple and obviously very inefficient fore general purpose compression, it can be very useful at times (it is used in JPEG compression, for instance).

_HUFFMAN

Huffman encoding is one of the best methods for lossless compression. It replaces each symbol with an alternate binary representation, whose length is determined by the frequency of the particular symbol. Common symbols are represented by few bits, while uncommon symbols are represented by many bits. The Huffman algorithm is optimal in the sense that changing any of the binary codings of any of the symbols will result in a less compact representation. However, it does not deal with the ordering or repetition of symbols or sequences of symbols.

_LZ

There are many different variants of the Lempel-Ziv compression scheme. The Basic Compression Library has a fairly straight forward implementation of the LZ77 algorithm (Lempel-Ziv, 1977) that performs very well, while the source code should be quite easy to follow. The LZ coder can be used for general purpose compression, and performs exceptionally well for compressing text. It can also be used in combination with the provided RLE and Huffman coders (in the order: RLE, LZ, Huffman) to gain some extra compression in most situations.

Lets take the following code:

```
in$ = LoadStr(AppPath$+ “MyFile.txt”, insize$);
New(out$, insize$ * 2);
outsize$ = Compress(_RLE, in$, out$, insize$);
```

This code will load file MyFile.txt into variable in$, return the size in bytes in variable insize$. We then need to create an output buffer that is at least equal or preferably greater that the original input buffer. We then apply the RLE compression method to in$ and outputting the result in the out$ variable returning the new size of the out$ variable in outsize$.

You can then decompress the out$ buffer to a new newin$ buffer with the following:

```
New(newin$, outsize$ * 2);
newinsize$ = Compress(_RLE, out$, newinsize$, outsize$);
```

Encrypting strings in PPL:

What if you want to protect a file from sneaky eyes? The best solution is to encrypt the file using a very strong password that only you know about. PPL comes with a nice function called Encrypt() that can do this for you very easily.

```
s$ = "HELLO WORLD!";
Encrypt(s$, -1, "MYKEY", True);
ShowMessage(s$);
Encrypt(s$, -1, "MYKEY", False);
ShowMessage(s$);
```

In the following example we encrypt the string “HELLO WORLD!” using an encryption algorythm that uses the key “MYKEY” to encode the result string. The last parameter of the Encryt() function specify if we are encrypting or decrypting the string, true means encryt and false means decrypt. It is always a good idea idea never to leave a key as a regular string in your code even though the .ppc file that PPL generates is encrypted and compressed, it can be easier for a hacker to decode. Try to build your key string using code with mathematical code if possible.

Linked-lists

Finally a chance to talk about the linked-lists. You have probably heard or learnt about them in school while studying C or maybe you've read about them on the internet or a magazine. Here is a great explanation from the great wikipedia.com:

“In computer science, a linked list is one of the fundamental data structures used in computer programming. It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes. A linked list is a self-referential datatype because it contains a pointer or link to another data of the same type. Linked lists permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access.”

In PPL linked-lists variables are extremely powerful and versatile. In each node you can have a different type of variable, structures or arrays. This opens up unlimited data storage in memory that is simply unmatched.

To create a list variable, there many ways, here is how you declare a list variable and how you add nodes to it:

```
List(l$);
Add(l$, 1, 2, 3, 4, 5);
Add(l$, “A”, “B”, “C”, “D”);
```

This new linked-list variable now contains 9 nodes with different values and types. Let's see how you can move through the nodes like other languages would allow you to:

```
First(l$);
while (1 == 1)
  ShowMessage(l$);
  if (Next(l$) == false)
    break;
  end;
end;
```

Here is how we iterate through the list in reverse order:

```
Last(l$);
while (1 == 1)
  ShowMessage(l$);
  if (Prev(l$) == false)
    break;
  end;
end;
```

The **First**() function moves the list internal pointer to the first node, **Next**() moves to the next node returning true if succeeded or false is past the end of the node list. The **Last**() function moves the internal list pointer to the last node in the list and the **Prev**() function moves to the previous node.

In PPL you can use the **ForEach**() statement to iterate through a list, an array, a structure or a matrix type variable:

```
ForEach(l$)
  ShowMessage(l$);
end;

ForEachRev(l$)
  ShowMessage(l$);
end;
```

If you need to store the list node value into another variable, you can place a second variable as a target in the **ForEach**() statement:

```
ForEach(l$, v$)
  ShowMessage(v$);
end;
```

Let's see how PPL can access nodes at random order, PPL can access list's nodes just like regular arrays:

```
ShowMessage(l$[0]);      // display 1
ShowMessage(l$[5]);      // display A
```

How do we go to a specific node position? Simple, by using the **Goto**() function. How do we know what node is the current one? Use the **Lpos**() function. How many nodes are in the list? Use the **Count**() function.

```
Goto(l$, 0);   // Like First()
ShowMessage(LPos(l$));

Goto(l$, Count(l$)-1);   // Like Last()
ShowMessage(LPos(l$));
```

We can also move nodes around using the Lmove() function:

```
Lmove(l$, 3, 1);    // This moves node 3 to node 1.
```

You can also insert nodes using the Ins() function:

```
Ins(l$, 0, 0);      // Insert value 0 at node 0.
```

To delete a node from the list, you can use the **Del**() function, if you want to empty the whole list, just use **Empty**().

```
First(l$);     // Move to first node.
Del(l$);  // Deletes first node.
Goto(l$, 5);   // Goto 5th node.
Del(l$);  // Deletes 5th node.

Empty(l$);     // Empty the whole list of all of its nodes.
```

In the next tutorial we will see how you can store different variable types like arrays and structures inside a list node.

Structures in Linked-lists

Linked-lists are very powerful and can allow for very complicated data storage. Now let's see how it possible to store a different structure type inside each list node.

First we need to create our list node:

```
List(l$);
Add(l$);
```

We now have one list node in l$ and our current internal pointer is placed on that first node. We can now define the variable type like we would with any other variable:

```
struct(l$, “a”, “b”);
l.a$ = 10;
l.b$ = 20;
```

We can now add a new node and store another structure into it:

```
Add(l$);
struct(l$, “c”, “d”);
l.c$ = 30;
l.d$ = 40;
```

Now let's iterate through the list and output the structure's element values:

```
ForEach(l$)
  if (Lpos(l$) == 0)
    ShowMessage(l.a$);
    ShowMessage(l.b$);
  else if (Lpos(l$) == 1)
    ShowMessage(l.c$);
    ShowMessage(l.d$);
  end;
end;
```

The **Lpos**() function returns the current pointer position of the list always starting with 0.

Imagine the possibilities offered by such flexibility.

PASM from the Beginning (by Brad Manske)

"PASM can wait till after 1.0" was my reaction when I heard the plans to include a cross platform assembler with PPL. While it was not a lot of code, the complexity was way up there. So that it could run on multiple processors it had to be a virtual processor that was compiled to. There were 36 addressing modes for 22 assembly instructions that would potentially compile to a series of instructions for 2 different processors. The project required complex and intimate knowledge of the processors and the testing challenge was not going to be easy.

My earliest e-mail on this project (that I kept) is dated the 20th of March 2004. We had already worked together for almost 2 years when the topic came up. There is no backing down from a complex technical challenge, so even though the pressure to release 1.0 was high, PASM went forward. This article will introduce you to PASM and it just may be the raw speed boost your code needs.

I will start off with some explanation for the people who have not had much exposure to assembly language. Assembly is a text representation of the 1s and 0s that the computer actually executes. For example:

```
X$ = 10;
```

This would be an instruction to move the value 10 into the variable X$:

```
move x, 10
```

Sound simple? Well, yes, if the processor supports moving a value into memory without going through a register first. And if the value fits in a 32 bit register (Windows CE requires a 32 bit processor). And, etc...

This is the reason that PASM uses a virtual processor internally. We couldn't guarantee that these conditions would be met for each processor since all of the code written for PASM must run on all of our supported platforms.

The PASM virtual processor is made up of 4 general purpose registers named **R0**, **R1**, **R2** and **R3**. There are more specialized registers like the Stack Pointer (**SP**) and the Stack Frame (**SF**). The Arithmetic and Logic unit of our processor is a simplified RISC (Reduced Instruction Set Computing) like design. The instruction set consists of about 22 different assembly operations. This isn't much compared to the hundreds of instructions supported by some processors, but there are 36 addressing modes to offset the simplicity of the instruction set.

Here is a quick look at the MOV instruction and the addressing modes that it supports. For 32 bit values:

```
mov Register, Value                           move from Value to Register
mov Absolute, Value                           move from Value to Absolute address
mov [Register], Value                         move from Value to Indexed Register
mov [Absolute], Value                         move from Value to Indexed Absolute address
mov Register, Register                        move from Register to Register
mov Register, [Register]                      move from Indexed Register to Register
mov [Register], Register                      move from Register to Indexed Register
mov [Register], [Register]                    move from Indexed Register to Indexed Register
mov Absolute, Register                        move from Register to Absolute address
mov [Absolute], Register                      move from Register to Indexed Absolute address
mov Register, [Register+offset]               move from Index+Offset Register to Register
mov [Register+offset], Register               move from Register to Index+Offset Register
mov [Register+offset], [Register+offset]      move from Index+Offset Register to Index+Offset Register
mov Absolute+offset, Register                 move from Register to Absolute+Offset address
mov Absolute+offset, Value                    move from Value to Absolute+Offset address
mov [Absolute+offset], Register               move from Register to Index+Offset Absolute address
mov [Absolute+offset], Value                  move from Value to Index+Offset Absolute address
mov [Register+offset], Value                  move from Value to Index+Offset Register
```

MOV also supports a size modifier for 8 bits (byte) and 16 bit (word) values:

```
mov size Register, Register                   move size from Register to Register
mov size Register, [Register]                 move size from Indexed Register to Register
mov size [Register], Register                 move size from Register to Indexed Register
mov size [Register], [Register]               move size from Indexed Register to Indexed Register
mov size Absolute, Register                   move size from Register to Absolute address
mov size Absolute, Value                      move size from Value to Absolute address
mov size [Absolute], Register                 move size from Register to Indexed Absolute address
mov size [Absolute], Value                    move size from Value to Indexed Absolute address
mov size Register, Value                      move size from Value to Register
mov size [Register], Value                    move size from Value to Indexed Register
mov size Register, [Register+offset]          move size from Index+Offset Register to Register
mov size [Register+offset], Register          move size from Register to Index+Offset Register
mov size [Register+offset], [Register+offset] move size from Index+Offset Register to Index+Offset
Register
mov size Absolute+offset, Register            move size from Register to Absolute+Offset address
mov size Absolute+offset, Value               move size from Value to Absolute+Offset address
mov size [Absolute+offset], Register          move size from Register to Index+Offset Absolute address
mov size [Absolute+offset], Value             move size from Value to Index+Offset Absolute address
mov size [Register+offset], Value             move size from Value to Index+Offset Register
```

A few quick note about the notation above. The brackets [] above mean that the value of the expression inside the brackets is the memory location that will be operated on. Register is a register **R0** to **R3** or one of the special registers. Absolute, is a number representing a specific memory location. Offset an integer value that allows you to adjust the value of the memory address operated on without the need to modify the base.

Here is a very simple example:

```
#include "console.ppl"

func WinMain;
  InitConsole;
  ShowConsole;

  new(startVal$, tint);
  new(endVal$, tint);

  StartVal$ = 0;
  EndVal$ = 0;
  asmCall$ = asm(1024, {
#DEASM
  :main
    mov StartVal$, 1
    mov EndVal$, 2

    savesp
    pplpush [AARGSCOUNT$]
    ppl showmessage

    savesp
    pplpush [AARGS$]
    ppl showmessage
  });

  callasm(asmCall$, 20, 30);
  writeln("Test "+ startVal$ + ", "+ endVal$);

  freeasm(asmCall$);
  free(startVal$, endVal$);

  return(true);
end;
```

If you read my previous articles, you know that I'm a fan of using the console for my examples, so it should be no surprise that I first include the console. Next I declare some variables in the PPL memory space outside of PASM. Next is the assembly code followed by the **CallASM** instruction. Some values are written and the assembly code and variables are freed.

When compiled, the call to **ASM** takes 2 arguments the first being the size of the byte buffer that holds the assembled code and the second is the string of assembly instructions. The buffer is specified in bytes and a multiplier is used on the buffer size depending on what you are doing. For example, by running your code under debug, it is possible to step through and break on assembly instructions. In order to do this, extra machine code instructions are inserted to support doing this so the buffer must be expanded. It also means that your code will execute slower under debug than it will in run mode. The buffer is created at run time and the assembler runs against the 2nd argument which is the text with all of the assembly instructions. So keep in mind that if you make a change to the assembly code, any errors will not be found until run time. It also means if you plan on using the assembler you may want to place your **ASM** instructions at startup and keep them for the duration of the program so that the code is not reassembled during the execution of your program when you really need the speed.

The **CallASM** instruction invokes the code created in the buffer by the **ASM** command. **CallASM** can take additional arguments that will be passed into the assembly code as parameters. The parameters are placed into an AARGS$ array and the size of the array is placed into AARGSCOUNT$. Each of the parameters are treated as a 4 byte (32 bit) value. So the value of 20 is at [AARGS$] and the value of 30 is at [AARGS$+4].

The line ":main" above indicates the entry point to your assembly code. This is a label and is used as the target in jump instructions.

The line "#DEASM" above instructs the ASM instruction place the actual assembly instruction for your processor to be placed into the DebugLog file. This does take extra time, so it shouldn't be used in production programs. The #debugoff pragma can be used to disable this for the entire project.

Here is a simple example when using #DEASM. The 2 lines from above:

```
mov StartVal$, 1
mov EndVal$, 2
```

On Intel processors are translated into:

```
mov  edi, D45B28h(STARTVAL$)
mov  [edi], 01h
mov  edi, D45B98h(ENDVAL$)
mov  [edi], 02h
```

On Arm processors are translated into:

```
ldr  r9, 34C9B0h(STARTVAL$)
ldr  r10, #01h
str  r10, [r9]
str  r10, #01h
ldr  r9, 34CA40h(ENDVAL$)
ldr  r10, #02h
str  r10, [r9]
str  r10, #02h
```

The lines from the PASM example above:

```
    savesp
    pplpush [AARGSCOUNT$]
    ppl showmessage
```

Show an example of saving the position on the stack pushing some arguments onto the stack where ppl can get to it then calling a PPL function. Here is another example:

```
    savesp
    pplpushstr tstStr1$
    pplpushstr tstStr2$
    pplpushstr tstStr3$
    ppl  concat
    pplpull
```

In this example, the stack pointer is saved, all of the required arguments are placed onto the stack and the PPL concat function is called to concatenate the strings together. The stack is restored to its previous state after the PPL call, then the address of the new string is pulled from the stack. The new string is created in a new memory space that the garbage collector will automatically clean up from.

I'll leave you with one more example. This example demonstrates the usage of Jump instructions and the use of an assembly procedure. The entry point is at ":main". It tests the number of arguments passed into the assembly code to see if there is only one. In this case there is only one so the value of 20 is passed to the function "!asmCalc". As in high level code, the string in parenthesis becomes a variable for the function. The "Var FinSum" instructions declares a local var for use within the function. The function then calculates the Fibonacci series on the number passed to it adding all of the numbers from n + (n-1) + ... + 2 + 1.

```
#include "console.ppl"

func WinMain;
  InitConsole;
  ShowConsole;

  new(startVal$, tint);
  new(endVal$, tint);

  asmFib$ = asm(1024, {
#DEASM
  !asmCalc(Value)
      var FinSum
      mov startVal$, value
      mov R0, 0
    :FibLoop
      cmp value, 0
      jeq FibDone
      add R0, value
      sub value, 1
      jmp FibLoop
    :FibDone
      mov FinSum, R0
      mov endVal$, FinSum
      ret
    :main
      cmp [AARGSCOUNT$], 1
      jeq extArg
      jsr asmCalc(10)
      jmp asmDone
    :extArg
      jsr asmCalc([AARGS$])
    :asmDone
  });

  t$ = tick;
  callasm(asmFib$, 20);
  writeln("Fibbon("+ startVal$ + ")="+ endVal$ + "  time =" + (tick -
t$));

  freeasm(asmFib$);
  free(startVal$, endVal$);

  return(true);
end;
```

Play with PASM a while and let us know what you think in the Forums. In the next newsletter, I will address some more advanced examples.

Form Options Quick Reference (by Brad Manske)

Each of the options are listed below along with the effect that it has on the form.

Dialog Form
Default: Off

When selected, the form will be created using a call to NewDlg to create the window. If not selected, then NewForm will be used to create the window if it is full screen, DefaultForm is selected in styles and there are no extended styles. If these conditions are not met, then the NewFormEx function will be used to create the window.

Generate Library
Default: Off

When selected, the form will create PPL source code that can be included in with a larger project. If not selected, the form will have a WinMain function indicating the entry point for the project. It is possible to create a stand alone program in a single form file with this option deselected.

Simplified Event Handling
Default: On

When selected, PPL will route the windows messages to the correct handler functions. If not selected, then the more traditional case table type logic is needed to decode and process the messages sent to the handler functions.

Extended Event Code
Default: On

When selected, PPL will include code in the Simplified Event Handlers to do some of the more common decoding of the windows messages. If not selected, the code will be left off. To find the included code look in SWAPI.PPL for the #define for HandleEventParms.

Use Namespace
Default: On

When selected, the form will include the #NameSpace command in the generated PPL code. This will force global variables and controls into their own namespace to avoid items named the same on multiple forms. If not selected, then all global variables and controls will be placed into the global NameSpace.

PPL Game Programming - Part 3

The physic engine

Our world is made of physic, everything is physic! Gravity pushes every object to the ground, every step we take involves friction with the surface our shoes come in contact with, wind blows and pushes leaves around and balls bounces from the ground and walls with different elasticity. When applied to a game, physics can add more realism and can add possibilities you thought were too hard to code.

PPL comes with a 2D automated physic engine. When I say automated I mean it, you only need to set the mass of a sprite, set the global gravity and the physic will take care of pushing the object down. Objects can bounce back and forth by simply assigning an elasticity value. The physic is far from an advanced one that supports object deformation on collision and such but it can do a very good job simplifying your life while developing your game.

Lets first start by reviewing the principal functions used by the physic engine:

SetSpriteMass(Sprite$, Mass);

The mass is a percentage value that is compared to the other sprites with physic. A sprite with 0.5 mass will weight half the weight of a sprite with a mass of 1.0.

SetSpriteElasticity(Sprite$, Elasticity);

The elasticity value is a percentage compared to the other sprites. The bigger the value the more rebound will be applied when the sprite collides with another sprite.

SetSpriteFriction(Sprite$, Friction);

The friction is the amount of friction in percentage applied to reduce the movement speed of the sprite when it collides with another sprite.

SetGravity(Gravity);

This function will set the global gravity of the sprite engine, the value of the gravity is applied to the sprites movement speed each cycle. Default gravity is around 0.1.

SetFriction(Friction);

Set the global friction that is applied to sprite's movement speed each cycle. The default friction value is 0.00025.

There are options that need to be activated for the physic engine to consider moving the objects around. The first one is the **SO_KINETIC** to active the physic engine on the sprite itself. The second one is the **SO_BOUNCE**, it will make the sprite bounce around from other sprites. The bounce force is calculated based on the Elasticity of the sprite. Sprites must have the **SO_COLLIDECHECK** option set to them for bouncing to occur. You can have your sprites bounce from the screen edges by setting the **SO_BORDER** option. Most of the sprites you will want to bounce around will be of oval shape, you will want to set the **SO_OVAL** option for the physic engine to bounce the sprite like a real oval shaped object.

Lets review the bounce.ppl demo that comes with PPL. This demo involves 5 basketball balls bouncing around from the screen edges and from each other. In this example we create 5 sprites with the basketball ball image and the we set the options of each sprite to oval shape, collision checking, pixel checking and border collision check.

```
// Set global gravity.
SetGravity(0.1);
// Set global friction.
SetFriction(0.005);

i$ = 0;
while (i$ < 5)
  // Load ball sprite from disk.
  s$ = loadsprite(AppPath$ + "ball.bmp", G_RGB(255, 0, 255), 1, 0,
NULL);
  // Activate pixel perfect collision detection.
  AddSpriteOption(s$, SO_OVAL | SO_CHECKCOLLIDE | SO_PIXELCHECK |
SO_BORDER | SO_KINETIC);
  // Make the balls collide and never go over another.
  SetSpriteCollide(s$, "BALL");
  SetSpriteId(s$, "BALL");
  repeat
    MoveSprite(s$, random(g_width - 40), random(64));
    ProcessSprites(1, 0);
  until (Collide(s$, SpriteX(s$), SpriteY(s$), nx$, ny$) == NULL);
  // Set ball elasticity.
  SetSpriteElasticity(s$, 0.01);
  // Set some friction when the balls collide.
  SetSpriteFriction(s$, 0.01);
  // Set sprite's weight.
  SetSpriteMass(s$, 0.5);
  // Set the maximum velocity to 10 pixels.
  SetSpriteVelLimits(s$, 0, 10);
   i$++;
end;
```

Time to say goodbye

Thank you for your interest in PPL. We hope to see you in the forums, let's make that place a source of good information and the best possible library for PPL enthusiasts. If you any comments or questions, please visit our forums or contact us via support@arianesoft.ca

If you want to contribute to this newsletter with articles, tips, suggestions, please contact us, we will be happy to include them.

Regards,
Alain Deschenes
President
ArianeSoft Inc.
www.arianesoft.ca