Table of Contents

Table of Contents

Table of Contents

Table of Contents

1 Argument

1.1 Definition

An argument is the value passed on when calling a function or process. The variable and its value inside the definition of a function or process is called a parameter.

1.2 Example

```
Function int my_proc( int parameter )
Begin
    //statements
    return 0;
End

Process Main()
Private
    int argument = 3;
Begin
    my_proc( argument );
    my_proc( 4);
End
```

Used in example: Function, Process

1.3 See also

- Parameter

2 Basic statements

Below are explained the basic statements of Program and Process and where to declare all the types of variables. For prototyping see Declare.

A list of basic statements:

- Program
- Process
- Function
- Declare
- Begin
- OnExit
- End
- Private
- Public
- Local
- Global
- Const

2.1 Basic statements

```
Program example;
Global // Start a global variables part of the program
End

Const // Start a constants part of the program
End

Local // Start a global variables part of the program
End

Private // Start a private variables part of the main process
End

Begin // Start the main code part of the main process
    proc1(); // create new instance of proc1
    Loop
        frame;
    End
OnExit // Start the exit code part of the main process
End

Global // Start a global variables part of the program
End

Const // Start a constants part of the program
End

Local // Start a global variables part of the program
End

Process proc1()

Public // Start the public variables part of the process
Private // Start the private variables part of the process

Begin // Start the main code part of the process
    Loop
        frame;
    End
OnExit // Start the exit code part of the process

End

Function int func1()
Public // Start the public variables part of the function
Private // Start the private variables part of the function
Begin // Start the main code part of the function
    return 0;
OnExit // Start the exit code part of the function
End
```

Used in example: **basic statements**, loop, return, frame

Global, constant, local and private parts of the program can be scattered through the code, between processes and functions. Sometimes the End can be left out, but it's good practice to keep it in. When a variable or constant is declared, it's only viewable or editable for statements *beneath* the declaration. For more info on that, see prototyping.

Note that when Declare is used, the Public variables have to be declared in the Declare block and not in the process block.

3 Bennu console

The Bennu console is a handy debugging tool. Many commands can be entered in it, like process manipulation or variable manipulation. The current state of Globals, Constants, Locals, Publics, Privates can also be monitored. One can add text to it in the programcode by using the function say().

To use the debugger, you need import the debug module (mod_debug) in your source code, run in debug mode by compiling your program in bgdc.exe with the argument "-g", and when you are running your application with the interpreter(bgdi.exe) you can activate it in-program with ALT+C.

3.1 Commands

3.1.1 Process Info

3.1.1.1 INSTANCES

List all running processes in a tree view. It shows what processes a processes called.

3.1.1.1.1 GLOBALS

List all global variables with their current values. Both predefined and user defined variables are listed.

3.1.1.1.2 LOCALS <processName|processID>

List all local variables of the specified process. The process can be specified by either its name or its processID. Both predefined and user defined variables are listed.

3.1.1.1.3 PRIVATES <processName|processID>

List all local variables of the specified process. The process can be specified by either its name or its processID.

3.1.1.1.4 PUBLICS <processName|processID>

List all local variables of the specified process. The process can be specified by either its name or its processID.

3.1.2 Debugging Commands

3.1.2.1 TRACE

Execute the next instruction.

3.1.2.1.1 BREAK

List all set breakpoints.

3.1.2.1.2 BREAK <processName|processID>

Add a breakpoint on the execution of the specified process. The process can be specified by either its name or its processID.

3.1.2.1.3 DELETE

Delete a breakpoint on the execution of the specified process. The process can be specified by either its name or its processID.

3.1.2.1.4 CONTINUE

Continue the execution (close the console).

3.1.2.1.5 DOFRAME

Execute the next frame.

3.1.3 Misc

3.1.3.1 SHOW <expression>

Evaluate and show the specified expression.

3.1.3.1.1 STRINGS

List all strings in memory and how many times they are used.

3.1.3.1.2 VARS

List all internal variables.

3.1.3.1.3 QUIT

Kill the program and exit.

3.1.4 Process Management

See signals for further explanation about the following signaling commands.

3.1.4.1 RUN <processName> [<argument>]

Start a process with optional arguments.

3.1.4.1.1 KILL <processName|processID>

Kill the specified process. The process can be specified by either its name or its processID.

3.1.4.1.2 WAKEUP <processName|processID>

Wake up the specified process. The process can be specified by either its name or its processID.

3.1.4.1.3 SLEEP <processName|processID>

Put the specified process to sleep. The process can be specified by either its name or its processID.

3.1.4.1.4 FREEZE <processName|processID>

Freeze the specified process. The process can be specified by either its name or its processID.

3.1.4.1.5 KILLALL <processName|processID>

If a processname is specified, kill all processes with that name; if a processID is specified, only kill that process.

3.1.4.1.6 WAKEUPALL <processName|processID>

If a processname is specified, wake up all processes with that name; if a processID is specified, only wake up that process.

3.1.4.1.7 SLEEPALL <processName|processID>

If a processname is specified, put all processes with that name to sleep; if a processID is specified, only put that process to sleep.

3.1.4.1.8 FREEZEALL <processName|processID>

If a processname is specified, freeze all processes with that name; if a processID is specified, only freeze that process.

3.1.5 Expressions

You can evaluate free expressions and see/alter the values of variables.

3.1.5.1 <variable> [= <value>]

Return or alter a global variable.

3.1.5.1.1 <processName|processID>.<variable> [= <value>]

If a processID is specified, the value of the local, private or public variable of that process is returned or altered. If a processName is specified, the process with that name and the lowest ID is assumed.

3.2 Example

```
Process Main()
Begin
    Repeat
        frame;
    Until(key(_ESC))
End
```

Template:Image

Template:Debugging

4 Bennu profiler

(We will keep this page as it is right now, as the profiler does not exist in Bennu yet.)

The Bennu profiler is a handy debugging tool. It shows how much power is needed for certain systems of Bennu, most particularly the drawing and the interpreting.

To use the profiler,import the debug module (mod_debug) in your sorce code and compile it in bgdc.exe with the argument "-g", and then show it in-program with ALT+P.

Shortcuts:

ALT-P	- Show/hide the Bennu profiler
ALT-R	- Resets the profile history of the Bennu Profiler.
ALT-S	- Activate/Deactivate the Bennu Profiler.

4.1 Example

```
Process Main()
Begin

    // To make sure the profiler is updated every frame
    restore_type = COMPLETE_RESTORE;

    A();

    Loop
        frame;
    End

End

Process A()
Begin
    Loop
        frame;
    End
End
```

Used in example: restore_type, process, frame

Template:Image

Template:Debugging

5 Constant

A constant is a container containing a value. This value cannot be changed apart from initialization, hence the name constant, as opposed to a variable of which the value can vary. A constant can be of any datatype and can contain a value according to its datatype. For example, an integer has whole numbers between -2^31 (-2147483648) and +2^31-1 (2147483647) (e.g. 23) , a float has a floating point number (e.g. 2.674) and a string has a series of characters (e.g. "Hello World!").

There is a number of predefined constants. These variables are predefined, which means they exist without the programmer making them. They can be useful in a some functions or when assigning them to some global variables.

5.1 See Also

- A list of predefined constants
- Variable

6 Control flow statements

Control flow statements are statements which influence the control flow (the order in which statements are executed) of the program.

- If, ElseIf, Else, End
- While, End
- Repeat, Until
- For, End
- From, To, Step, End
- Loop, End
- Switch, Case, Default, End

7 Datatypes

7.1 Description

Datatypes give meaning to data and dictate how a variable acts and reacts. Examples of datatypes are `int`s, `float`s and `string`s. Special cases are voids, arrays, varspaces and structs. User made types can also be defined, by use of the operator Type.

7.2 List

<DPL> category = datatypes mode = userformat columns = 1 listseparators = ,\n* %TITLE%,, redirects = include ordermethod = titlewithoutnamespace resultsfooter = \n%PAGES% datatypes </DPL>

7.3 Example

```
import "mod_draw"
import "mod_wm"
import "mod_key"
import "mod_map"

Type _point
    int x;
    int y;
End

Process Main()
Private
    _point A,B;
Begin

    // Init the points
    A.x = 100;
    A.y = 50;
    B.x = 250;
    B.y = 150;

    // Setup drawing
    drawing_map(0,0);
    drawing_color(rgb(0,255,255));

    // Draw a box
    drw_box(A,B);

    // Wait for key ESC or X button
    Repeat
        frame;
    Until(key(_ESC)||exit_status)

End

// Draw a box using two points
Function int drw_box(_point A, _point B)
Begin
    return draw_box(A.x,A.y,B.x,B.y);
End
```

Used in example: drawing_map(), drawing_color(), rgb(), key(), draw_box(), exit_status

8 Debugging

Debugging a Bennu program can be tricky. So here are a few tips to get you on your way.

8.1 Debug mode

To run a game in debug mode, you have to import the debug module("mod_debug") and to compile the .prg with the -g option (e.g. `bgdc.exe -g name.prg`). Then when you run the game, it is in debug mode. In this there's a number of shortcuts, with a variety in usefulness.

8.1.1 Shortcuts

ALT-C - Show/hide the Bennu console.
ALT-F - Go into fullscreen mode.
ALT-G - Make a screenshot and save it as `shotXXXX` where `XXXX` is the lowest possible number.
ALT-P - Show/hide the Bennu profiler
ALT-R - Resets the profile history of the Bennu Profiler.
ALT-S - Activate/Deactivate the Bennu Profiler.
ALT-W - Go into window mode.
ALT-X - Exit.
ALT-Z - Switch the MODE_16BITS flag of graph_mode on/off.

8.2 Bennu Console

The Bennu console is a pretty handy debugging tool. You can view all active processes, view values of any variable, perform one instruction, perform one frame, manage breakpoints on processes, manage processes, etc. A sure go for debugging.

8.3 Bennu Profiler

The Bennu profiler is somewhat handy, mostly for tracing where a program is the most power hungry.

Template:Debugging

9 Error

9.1 Definition

A Fenix error is an errorbox with some cryptic Spanish error message. When a function has a section *Errors*, it list what errors the function can cause. In later Fenix versions however (*from which version? Later than 0.84b probably*), the error isn't displayed in an error dialog, but is printed to stderr.txt and the Fenix program is terminated.

9.2 Example

Here's an example of an error, in Fenix 0.84a, when it is tried to blit a 16bit graph onto an 8bit one: Template:Image

10 Fenix console

The Fenix console is a handy debugging tool. Many commands can be entered in it, like process manipulation or variable manipulation. The current state of Globals, Constants, Locals, Publics, Privates can also be monitored. One can add text to it in the programcode by using the function say().

In the later Fenix versions (*from which version exactly?*) the Fenix console freezes the program when activated, until it is deactivated again.

To use the debugger, run in debug mode by compiling your program in FXC.exe with the argument "-g", and then activate it in-program with ALT+C.

10.1 Commands

10.1.1 Process Info

10.1.1.1 INSTANCES

List all running processes in a tree view. It shows what processes a processes called.

10.1.1.1.1 GLOBALS

List all global variables with their current values. Both predefined and user defined variables are listed.

10.1.1.1.2 LOCALS <processName|processID>

List all local variables of the specified process. The process can be specified by either its name or its processID. Both predefined and user defined variables are listed.

10.1.1.1.3 PRIVATES <processName|processID>

List all local variables of the specified process. The process can be specified by either its name or its processID.

10.1.1.1.4 PUBLICS <processName|processID>

List all local variables of the specified process. The process can be specified by either its name or its processID.

10.1.2 Debugging Commands

10.1.2.1 TRACE

Execute the next instruction.

10.1.2.1.1 BREAK

List all set breakpoints.

10.1.2.1.2 BREAK <processName|processID>

Add a breakpoint on the execution of the specified process. The process can be specified by either its name or its processID.

10.1.2.1.3 DELETE

Delete a breakpoint on the execution of the specified process. The process can be specified by either its name or its processID.

10.1.2.1.4 CONTINUE

Continue the execution (close the console).

10.1.2.1.5 DOFRAME

Execute the next frame.

10.1.3 Misc

10.1.3.1 SHOW <expression>

Evaluate and show the specified expression.

10.1.3.1.1 STRINGS

List all strings in memory and how many times they are used.

10.1.3.1.2 VARS

List all internal variables.

10.1.3.1.3 QUIT

Kill the program and exit.

10.1.4 Process Management

See signals for further explanation about the following signaling commands.

10.1.4.1 RUN <processName> [<argument>]

Start a process with optional arguments.

10.1.4.1.1 KILL <processName|processID>

Kill the specified process. The process can be specified by either its name or its processID.

10.1.4.1.2 WAKEUP <processName|processID>

Wake up the specified process. The process can be specified by either its name or its processID.

10.1.4.1.3 SLEEP <processName|processID>

Put the specified process to sleep. The process can be specified by either its name or its processID.

10.1.4.1.4 FREEZE <processName|processID>

Freeze the specified process. The process can be specified by either its name or its processID.

10.1.4.1.5 KILLALL <processName|processID>

If a processname is specified, kill all processes with that name; if a processID is specified, only kill that process.

10.1.4.1.6 WAKEUPALL <processName|processID>

If a processname is specified, wake up all processes with that name; if a processID is specified, only wake up that process.

10.1.4.1.7 SLEEPALL <processName|processID>

If a processname is specified, put all processes with that name to sleep; if a processID is specified, only put that process to sleep.

10.1.4.1.8 FREEZEALL <processName|processID>

If a processname is specified, freeze all processes with that name; if a processID is specified, only freeze that process.

10.1.5 Expressions

You can evaluate free expressions and see/alter the values of variables.

10.1.5.1 <variable> [= <value>]

Return or alter a global variable.

10.1.5.1.1 <processName|processID>.<variable> [= <value>]

If a processID is specified, the value of the local, private or public variable of that process is returned or altered. If a processName is specified, the process with that name and the lowest ID is assumed.

10.2 Example

```
Process Main()
Begin
    Repeat
        frame;
    Until(key(_ESC))
End
```

Template:Image

Template:Debugging

11 Fenix profiler

(We will keep this page as it is right now, as the profiler does not exist in Bennu yet.)

The Fenix profiler is a handy debugging tool. It shows how much power is needed for certain systems of Fenix, most particularly the drawing and the interpreting.

To use the profiler, run in debug mode by compiling your program in FXC.exe with the argument "-g", and then show it in-program with ALT+P.

Shortcuts:

ALT-P	- Show/hide the Fenix profiler
ALT-R	- Resets the profile history of the Fenix Profiler.
ALT-S	- Activate/Deactivate the Fenix Profiler.

11.1 Example

```
Process Main()
Begin

    // To make sure the profiler is updated every frame
    restore_type = COMPLETE_RESTORE;

    A();

    Loop
        frame;
    End

End

Process A()
Begin
    Loop
        frame;
    End
End
```

Used in example: restore_type, process, frame

Template:Image

Template:Debugging

12 File

This is about the filetype. Did you mean the local variable file or the function file()?

Up to Filetypes

12.1 Description

A file is a container for graphics, identified by a non-negative integer (0 or higher). It holds all information about the contained graphics (pixels, width, height, depth, name, etc). Each of these graphics have a unique identifier inside the file (positive int).

A file can be created for example by loading an FPG (*Fichero Para Gráficos*, meaning "file for graphics") into it, by using fpg_load(), which creates a new file with the graphics from the loaded FPG and returns a unique identifier. Another option is to create a new, empty one by using fpg_new(). Don't let the name *fpg*_new() fool you: fpg_new() has nothing to do with the filetype FPG. This is because the FPG format is only for files and not for internal use. There are more ways to load graphics into a file.

A file can be used by using the local variable file or by using the identifier in the various functions with a file parameter.

Don't forget to unload it with fpg_unload() after use.

12.2 Example

```
import "mod_map"
import "mod_grproc"
import "mod_key"
import "mod_wm"

Global
    int file_id;
    int file_id2;
End

Process Main()
Begin

    // Load FPG
    file_id = load_fpg("example.fpg");
    file_id2 = load_fpg("example2.fpg");

    // Set locals for display of graph
    file = file_id;
    graph = 1;
    x = y = 50;

    // assign Ship to use example2.fpg
    Ship(300,100,5,file_id2,1); // undefined in this sample

    Repeat
        frame;
    Until(key(_ESC)||exit_status)

End
```

Used in example: load_fpg(), key(), x, y, **file**, graph. process

Media in example: example.fpg

Note: nothing will be seen unless you have an FPG "example.fpg" with a graphic with ID 1.

Template:Filetypes

13 FileHandle

13.1 Description

A **FileHandle** is an identifier for a file, for use in functions requiring a FileHandle, like fread(), fwrite() and fclose().

Consider the following code:

```
handle = fopen("file.txt",O_READ);
```

The fopen() function will try to open the "file.txt" for reading. If it fails to open the file for whatever reason (e.g. file not found or insufficient user rights) `handle` will have the value of 0. If opening the file for reading succeeded, the `handle` will have a value other than 0. What the value will be depends on the OS and runtime circumstances. It represents an OS internal pointer to the actual file.

The file will remain open for reading, until it is closed using fclose():

```
fclose(handle);
```

This is important, because else the operating system and other programs think you are still using it and this can result in them not being able to open the file.

13.2 Example

So always check if a file has been successfully opened and close the file when you are done with it.

```
import "mod_file"
import "mod_say"

Process Main()
Private
    int handle;   // handle for the loaded file
    string first_line; // here's where the first line of the file will go
Begin

    // Open the file "file.txt"
    handle = fopen( "file.txt",O_READ);

    if (handle == 0)
        say( "Could not open file" );
        return;
    end

    // Read the first line form the file
    first_line = fgets(handle);

    // Output the read line
    say( "Read from file:" );
    say('"' + first_line + '"');

    // Close the file (important!)
    fclose(handle);

End
```

Used in example: fopen(), say(), fgets(), fclose()

Template:Moduledocbox

14 FileID

14.1 Definition

FileID

A FileID is an identifier associated with a certain file (FPG). It is returned by load_fpg(), when a file is loaded to memory and can be used in many functions, wanting a FileID, for example map_put() or start_scroll(). A FileID can also be assigned to the local variable file of a process or function, which will make the process in question look in that file (FPG) for the graphic, associated with the GraphID specified by the process' graph local variable.

14.2 Notes

You can fill out 0 for the **FileID** to target the system file, for graphics with an ID of *1000* or higher, which are graphics created on the fly.

14.3 Example

```
Program files;
Global
    int file_id;
Begin

    // Load FPG
    file_id = load_fpg("example.fpg");

    // Set locals for display of graph
    file = file_id;
    graph = 1;
    x = y = 50;

    Repeat
        frame;
    Until(key(_frame))

End
```

Used in example: load_fpg(), key(), file, graph

15 Filetypes

15.1 Definition

Bennu knows multiple filetypes.

15.2 List

<DPL> category = filetypes mode = userformat columns = 1 listseparators = ,\n* %TITLE%,, redirects = include ordermethod = titlewithoutnamespace resultsfooter = \n%PAGES% filetypes </DPL>

16 Font

16.1 Definition

A font is a way of displaying text. A font is associated with a certain fontID.

17 FontID

17.1 Definition

FontID

A FontID is an identifier associated with a certain font. It is returned by functions that load fonts, for example load_fnt(). You can use the FontID in functions to specify the font used in the writing of texts like write(), write_int(), write_float(), write_string(), write_var() and write_in_map().

17.2 Notes

Font "0" is the built-in font, and can be used in functions as well.

17.3 Example

```
import "mod_text"
import "mod_key"

Global
    My_font;
End

Process Main()
Begin

    //Using the built-in font:
    My_font=0;
    Write(My_font,320/2,200/2,4,"Game programming is awesome!");

    Repeat
        frame;
    Until(key(_ESC))
End
```

Used in example: write(), key()

This will result in something like:
File:FontID.png

18 Fps

Up to Global Variables

18.1 Definition

INT fps

The global variable **fps** holds the current frames per second on which Bennu is running. This means how many times a frame is executed every second and how many times the screen is updated every second.

If a different FPS is needed, the target FPS can be altered by use of set_fps().

If a more accurate FPS is needed, use frame_time to calculate it.

Template:Globals

19 Function

Up to Basic Statements

List of Functions

19.1 Syntax

Template:Syntaxdocbox **Function** <returntype> <name> **(** [<parameters>] **)**
[**Public**

[<public variables>]

End]
[**Private**

[<private variables>]

End]
Begin

[<function code>]

[**OnExit**

[<exit code>]

]
End

19.2 Description

Function is a reserved word used to start the code of a function.

A function is a subroutine to which one or more of the following apply:

- it receives parameters
- it acts on the parameters
- it processes data located elsewhere
- it returns a value

The difference between a function and a process is that the calling process or function waits until the function is completed. When a process or function calls a process, it doesn't wait. This means that, even when the called function contains frame statements, the calling function or process still waits for the function to finish. This is shown in this tutorial.

For a list of functions, see this list of functions.

19.3 Example

```
Function int addInts( int a , int b )
Private // Declare private variables here
Begin // Start the main functioncode
    return a+b;
End // End the main functioncode
```

addInts(3,6); will return 9. One can see that the function does indeed:

- receive parameters.
- act on the parameters.
- return a value.

Template:Keywords

20 Function:File

Up to Files Functions

20.1 Syntax

STRING file (<**STRING** filename>)

20.2 Description

Returns the whole contents of a certain file.

20.3 Parameters

STRING filename - The filename of the file you wish to read from (including extension and possible path).

20.4 Returns

STRING: The contents of the file.

Template:Moduledocbox

21 Global variable

21.1 Definition

A global variable is a variable that is accessible from anywhere in the code; it is shared by the whole code.

There's also a number of predefined local variables.

To start the declaration of global variables, use Global.

22 GraphID

22.1 Definition

GraphID

A GraphID is an identifier associated with a certain graphic in the file (FPG) specified by a FileID.

GraphID's can be used to point to certain graphics in certain files (FPGs), for displaying them onscreen, or manipulating them in another way, like put() or map_put(). They can also be used to specify the graphic of a process or function, by assigning the GraphID to the local variable graph. This will make the process use the graphic in the file (FPG), specified by the local variable file, at the position specified by the local variable graph.

22.2 Example

```
Program files;
Global
    int file_id;
Begin

    // Load FPG
    file_id = load_fpg("example.fpg");

    // Set locals for display of graph
    file = file_id;
    graph = 1;
    x = y = 50;

    Repeat
        frame;
    Until(key(_frame))

End
```

Used in example: load_fpg(), key(), file, graph

23 Graphic

23.1 Definition

A graphic (graph or map) is a bitmap with certain properties, like width, height, depth and of course the pixel data. It is used to give the program a graphical look. One can load graphics from files with for example load_png() or use Image.DLL. They can also be obtained from within FPG's or FGC's files, which are graphic collection files. To create one on the fly, the function new_map() can be used. Here's a list of functions related to graphics and maps.

23.2 Displaying a Graphic

To display a graphic on the screen, there are two main ways of doing this:

- Using drawing operations listed here.
- Using processes and assigning graphics to their graph variable.

Combinations can also be used naturally.

23.2.1 Drawing Operations

Graphics can be drawn onto each other, by the use of various map functions. The functions use blitting as the method for drawing, which is a reasonably fast way, as Bennu makes use of MMX capabilities. A function like put() will draw a picture on the background, displaying it on the screen (if restore_type is not NO_RESTORE).

23.2.2 Process Graphs

Graphics can be assigned to processes, which also have coordinate and transformation data. This is done by the use of the local variables graph, which holds the GraphID of a graph. The graphic is displayed, according to a few local variables influencing the graphic, like x, y, z, angle, etc.

24 Hello World

The Bennu Hello World! Example.

```
import "mod_say" // import the module to output text to console, using say()

Process Main() // start the definition of the main process
Begin // start the code
    say("Hello World!");
End // end the definition of the main process
```

Used in example: import, Process, Begin, End, say()

This will import the module "*mod_say*" which contains the definition of the "*say()*" function. The function will echo the argument "Hello World" to the screen.

25 Import

25.1 Syntax

import " <filename> **"**

25.2 Description

Imports a Bennu DLL with name *filename* into the program, which allows the usage of additional functionality in a Bennu program. For more information, see the article on DLLs.

25.3 Example

```
import "mod_say"
import "my_dll";

Process Main()
Begin
End
```

Used in example: **import**

Template:Keywords

26 Include

26.1 Syntax

include " <filename> **"**

26.2 Description

When the compiler reaches an **Include** statement, it continues compilation at the included file (usually *.INC) and when it's done resumes compiling from the **Include** statement. In other words, these files contain code that gets inserted at the place of inclusion.

This is very handy for breaking up your code into pieces. The handling of video in one include file, audio in another, game logic in another, etc. This makes code more maintainable and understandable; moreover it makes code reusable. The video handling include file you made for one game can be used for another game (if it was coded in a generic fashion) without spitting through the whole sourcecode of the first game.

Also headers can be used to import DLLs and possibly give a little more functionality to that DLL. For example Network.DLL uses a .INC header file to assure the DLL is only imported once during compilation and provides a little more functionality.

26.3 Example

main.prg

```
// The code in "bar.inc" will be processed first:
include "bar.inc"

import "mod_say"

Process Main()
Private
    int barcode;
Begin
    barcode = bar();
    say(barcode);
End
```

bar.inc

```
import "mod_rand"

Function int bar()
Begin
    return rand(0,10);
End
```

Used in example: **include**, import, write_int(), key()

Template:Keywords

27 Joystick

27.1 Description

A joystick is a general term for not just actual joysticks. Joysticks can be, but are not limited to:

- (actual) joysticks
- steering wheels
- gamepads
- controllers on consols
- GP2X main controls

A joystick provides different controls for different purposes.

27.2 Input

27.2.1 Axis

An axis is an analogue input, used for input that should be precise or sensitive. Examples are the flight stick control of the joystick (an actual joystick), the steer of a steering wheel and the analogue control on a gamepad.

In Bennu, values of axes range between `-32768` and `32767`.

See joy_numaxes(), joy_getaxis()

27.2.2 Button

A button is a binary input, meaning it is either pressed or not.

In Bennu, pressed is `1` and not pressed is `0`.

See joy_numbuttons(), joy_getbutton()

27.2.3 Hat

A hat is an 8-way digital pad. They are also called POV-hats because they are mostly used to alter the Point Of View. Examples are POV-hats on joysticks and steering wheels, but also D-pads on a gamepad can be a hat.

In Bennu, hat constants are used to describe the position of a hat.

See joy_numhats(), joy_gethat()

27.2.4 Ball

A ball is like a mouse: only relative values are measured.

In Bennu, the relative values of balls range between `-32768` and `32767`.

See joy_numballs(), joy_getball()

28 Local variable

28.1 Definition

A local variable is a variable that is specific to a process in the same way as a public variable: they are both accessible from other places in the code than the process/function itself. However, unlike a public variable, when a local variable is declared, *all* following processes will have that local.

There's also a number of predefined local variables.

To start the declaration of local variables, use Local.

28.2 Example

```
Local
    // insert local variables that you can use here
End

Process Main()
Begin
    // main code
End
```

Used in example: Process, Local, Begin, End

29 Local:File

Up to Local Variables

29.1 Description

INT file = 0

File contains the FileID of the file used to obtain the graphic indicated by the local variable GraphID.

29.2 Example

```
import "mod_map"
import "mod_grproc"
import "mod_key"
import "mod_wm"

Global
    int file_id;
End

Process Main()
Begin

    // Load FPG
    file_id = load_fpg("example.fpg");

    // Set locals for display of graph
    file = file_id;
    graph = 1;
    x = y = 50;

    Repeat
        frame;
    Until(key(_ESC)||exit_status)

End
```

Used in example: load_fpg(), key(), x, y, file, graph

Note: nothing will be seen unless you have an FPG "example.fpg" with a graphic with ID 1.

Template:Locals

30 Loops

30.1 Loops

LOOP-END, **WHILE-END**, **REPEAT-UNTIL**, **FOR-END**, **FROM-END**

Loops are used to create iterations in your code. The statements between these words will get repeated depending on a condition. There are several types of loops:

30.1.1 Loop ... End

```
Loop
    // Statements
End
```

- The statements will be repeated indefinitely.

30.1.2 While ... End

```
While(<condition>)
    // Statements
End
```

- The statements will be repeated while *condition* is fulfilled.

30.1.3 Repeat ... Until

```
Repeat
    // Statements
Until(<condition>)
```

- The statements will be repeated until *condition* is fulfilled.

30.1.4 For ... End

```
For( [ <initializer> ] ; [ <condition> ] ; [ <counting expression> ] )
    // Statements
End
```

- First, *initializer* will be executed. Then the statements will be repeated until *condition* is fulfilled. After each run of the statements, *counting expression* is executed.

30.1.5 From ... End

```
From <variable>=<startvalue> To <endvalue> [ Step <incrementvalue> ] ;
    // Statements
End
```

- First, *startvalue* will be assigned to the variable. Then the statements will be repeated and *incrementvalue* added to the variable. When the variable is greater than *endvalue*, the loop ends.
- Note that *incrementvalue* must be a constant. The `Step <incrementvalue>` part is optional.

30.2 Manipulating a loop

There are more ways to manipulate a loop, both internally and externally.

If the code inside a loop reaches a break; statement, the loop is immediately ended. This is particularly useful in the Loop-End loop, because that one has no other way to end the loop and continue code beneath it.

If the code inside a loop reaches a continue; statement, it executes the possible count- or stepping statement and then continues to the checking of the condition. So in a Repeat-Until loop, it would just skip to the Until() part.

If the loop is run in a specific process, one can influence the execution of the loop, by changing the state of that process, by use of signal() and signals.

30.3 Example

```
import "mod_say"

Const
    startvalue     = 1;
    endvalue       = 8;
    incrementvalue = 2;
End

Process Main()
Private
    int c;
Begin

    /* Loop */

    c = startvalue;
    Loop
        say("Loop: " + c);
        c+=incrementvalue;
        if(c>endvalue)
            break;
        end
    End
    say("End Loop: " + c);

    /* While */

    c = startvalue;
    While(c<=endvalue)
        say("While: " + c);
        c+=incrementvalue;
    End
    say("End While: " + c);

    /* Repeat */

    c = startvalue;
    Repeat
        say("Repeat: " + c);
        c+=incrementvalue;
    Until(c>endvalue)
    say("End Repeat: " + c);

    /* For */

    For(c=startvalue;c<=endvalue;c+=incrementvalue)
        say("For: " + c);
    End
    say("End For: " + c);

    /* From */

    From c=startvalue To endvalue Step incrementvalue;
        say("From: " + c);
    End
    say("End From: " + c);

End
```

Used in example: say(), constants, loop, if, break, while, repeat, for, from

In the example, all the loops do the same thing, illustrated by the use of the same constants. The output is:

```
Loop: 1
Loop: 3
Loop: 5
Loop: 7
End Loop: 9
While: 1
While: 3
While: 5
While: 7
End While: 9
Repeat: 1
Repeat: 3
Repeat: 5
Repeat: 7
End Repeat: 9
For: 1
For: 3
For: 5
For: 7
End For: 9
From: 1
From: 3
From: 5
```

```
From: 7
End From: 9
```

Template:Keywords

31 Operators

31.1 List of Operators

31.1.1 General

Operator	*- Description*
Type	- Get the ProcessTypeID of a ProcessType or define a new datatype. See Type.
. (period)	- Element access. `<struct>.<element>`. In case of a struct array, if no arrayelement is specified, it points to [0] (see example).

31.1.2 Maths

Operator	*- Description*
+	- Addition.
-	- Deduction
*	- Multiplication.
/	- Division.
%	- Modulus (remainder).

31.1.3 Logic

Operator	*- Description*
||	- OR. One or the other or both.
&&	- AND. Both.
^^	- XOR. One or the other, but not both.
!	- NOT.

31.1.4 Bitwise

(Logical operation per bit.)

Operator	*- Description*
|	- BOR. One or the other or both.
&	- BAND. Both.
^	- BXOR. One or the other, but not both.
~	- BNOT.
<<	- Bitshift left, causes bits to move left a certain number of positions.
>>	- Bitshift left, causes bits to move right a certain number of positions.

31.1.5 Memory

Operator	*- Description*
&	- OFFSET. Get the memory address of a variable. See pointer.
*	- POINTER. Get access to the variable a pointer is pointing to. See pointer.

31.2 Example

```
Type _point
    int x;
    int y;
End

Global
    int int_1 = 1;
    int int_3 = 3;
    int int_4 = 4;
    int someint = -5;
    String somestring = "AAP";
    String anotherstring = "BEER";
    byte somebyte = 6;
    signed byte sbyte = -2;
    byte b_5 = 5;
    byte b_12 = 12;
```

```
    Struct Person[9]
        string name;
        int age;
    End
    _point myPoint;
End

Process Main()
Begin

    say("---------- maths");

    say(int_3 + int_4);
    say(int_3 * int_4 + 1);

    say("---------- strings with numerical datatypes");

    say(somestring + anotherstring);
    say(somestring + ": " + int_3);
    say(anotherstring + ": " + int_3*sbyte);

    say("---------- mixed numberical types and typecasting");

    say(somebyte+someint);
    say((signed byte)someint);
    say((unsigned byte)someint);

    say("---------- logic");

    say(int_1&&int_4);
    say(int_4==int_3+int_1);
    say(!(somestring==anotherstring));

    say("---------- bitwise");

    say(b_5|b_12);  // 00000101
                    // 00001100
                    // -------- |
                    // 00001101 = 13

    say(b_5&b_12);  // 00000101
                    // 00001100
                    // -------- &
                    // 00000100 = 4

    say(b_5^b_12);  // 00000101
                    // 00001100
                    // -------- ^
                    // 00001001 = 9

    say(~b_12);     // 00001100
                    // -------- ~
                    // 11110011 = 243

    person.name    = "Mies"; // these are the same
    person[0].name = "Mies"; //

    person[1].name = "Aap";
    person[2].name = "Noot";
    // ...etc...
    person[9].name = "Last person"; // last array element

    setXY(&myPoint);

    Repeat
        frame;
    Until(key(_ESC))

End

Function int setXY(_point* p)
Begin
    p.x = 3; // this is actually (*p).x = 3, but . can be used like this
    p.y = 5; // this is actually (*p).y = 5, but . can be used like this
    return 0;
End
```

Used in example: say(), key(), Global, Type, Struct, Array, Pointer, period

This will result in something like:
Template:Image

32 Parameter

32.1 Description

A parameter is the variable and its value inside the definition of a function or process that is received from the calling environment. The value passed on when calling the function or process is called an argument.

If you use the same name as a local variable for a parameter, it will not make a new variable, but use the local variable instead. This means that passed arguments will modify the local variable. This is commonly used for x, y and graph.

32.2 Notes

There currently is a limit of 15 parameters that can be used per function or process.

32.3 Example

```
Process Main()
Private
    int argument = 3;
Begin
    my_proc( argument );
End

Process my_proc( int parameter )
Begin
    //statements
End
```

32.4 See also

- Argument

33 Precompiler

33.1 Definition

The precompiler is for executing commands before the actual compiling. This can be useful for many things:

- Defining words as other words or values
- Defining functions as a list of statements
- Protecting codefiles from being included multiple times
- Enable whole sections of code with one define

See the examples for more.

A list of precompiler statements:

- #define
- #ifdef
- #ifndef
- #endif
- #else
- #if

34 Private variable

34.1 Definition

A private variable is a variable that is specific to a process or function. Unlike a public variable, it can only be accessed from the process or function it was declared for.

To start the declaration of private variables, use Private.

34.2 Example

```
Function int SomeFunction()
Private
    int i,j; // declare private ints i and j
    String strtemp; // declare private string strtemp
Begin
    // ...
    return 0;
End
```

Used in example: Function, Private, Begin, End

The private variables i and j could be variables used for counting and the string probably would have some use as well.

35 Process

Up to Basic Statements

35.1 Syntax

Template:Syntaxdocbox **Process** <name> **(** [<parameters>] **)**
[**Public**

[<public variables>]

]
[**Private**

[<private variables>]

]
Begin

[<main code>]

[**OnExit**

[<OnExit code>]

]
End

35.2 Description

Process is a reserved word used to start the code of a process. If *name* is *Main*, that process will be started at the start of the program.

A process is a subroutine to which one or more of the following apply:

- it receives parameters
- it acts on the parameters
- it processes data located elsewhere

In addition to these possibilities, a process *always* has a frame; statement. The difference between a function and a process is a process is treated as a separate thread. This means one can't let a process return a value like a function, as the father process continues its code as well, as soon as the process hits a frame; statement or when the code is done. When that happens, the process 'returns' its ProcessID and continues the code (in the next frame).

When the frame; statement is reached in the code, a number of other local variables are defined or updated not only of the new process, but also of related processes. These are:

- The father variable of the new process.
- The son variable of the father process (updated).
- The bigbro variable of the new process.
- The smallbro variable of the processes called by the father immediately before the new process was called (updated).
- The son and smallbro variables are also defined of the new process, but do not yet carry values.

When there are no more processes alive, the program ends.

35.3 Local variables as parameters

When a process is declared with parameters that are actually local variables, arguments for these parameters will initialise those local variables. This may sound strange, but an example will clear things up.

For example, consider the local variables x, y, z, file and graph. To create a process to move a game sprite around, you can declare it as follows:

```
process Ship (x,y,z,file,graph)
begin

    // move left 1 pixel per frame
    repeat
        x -= 1; // move 1 pixel to the left
        frame; // this process is done for this frame, wait for the next
    until (x<0);

end
```

Calling the process with e.g. `Ship(300,100,5,0,1);` will have the Ship appear at the coordinates (300,100) on Z-Level 5 with the Sprite No.1 in the file number 0. The ship will move left until it leaves the screen. You can change movement by changing the x/y value of the process and animate the ship by changing the graph value.

35.4 Example

```
Process SpaceShip( int x, int y, int angle, int maxspeed, int maxturnspeed)
Public // Declare public variables here
Private // Declare private variables here
    int speed;
Begin // Start the main processcode
    graph = new_map(20,20,8);
    map_clear(0,graph,rgb(0,255,255));
    Loop
        speed+=key(_up)*(speed<maxspeed)-key(_down)*(speed>-maxspeed);
        angle+=(key(_left)-key(_right))*maxturnspeed;
        advance(speed);
        frame;
    End
OnExit // Start the exit code
    unload_map(0,graph);
End // End the main processcode
```

Now one can call this process for example by doing the following.

```
Process Main()
Begin
    SpaceShip(100,100,0,20,5000);
    Repeat
        frame;
    Until(key(_ESC))
    let_me_alone();
End
```

Used in example: new_map(), map_clear(), key(), advance(), unload_map(), let_me_alone(), **Process**, Begin, End, Loop, Repeat, graph, angle

And when the SpaceShip process ends - because the code of it reached the End or something sent an s_kill signal - the OnExit code starts. In this example it will unload the memory used for the created graphic. If there is no OnExit code, the process will just end.

This will make a SpaceShip with a cyan coloured block, able to move around the screen.

Template:Keywords

36 ProcessID

36.1 Definition

A ProcessID is a unique identification code, for one instance of a processType. A ProcessID is always odd and larger than 65536 (2^16).
This makes it possible to use for example the function collision() in an if() statement. This is because collision() returns the ProcessID the process calling collision() has collided with. Both processes and functions have processID's.

36.2 Usage

ProcessID is used when referring to specific process (an instance of a processType) and access/edit its local variables and/or public variables. The ProcessID of a process is found by one of the following methods:

- Returned when the frame; statement is found in the process.
- Returned by the get_id() function.
- Returned by the collision() function.
- Stored as the local variable id (holds the process' own ProcessID).
- Stored as the local variable father (holds the ProcessID of the process that [[call|called the current process).
- Stored as the local variable son (holds the ProcessID of the process last called from the current process).
- Stored]] as the local variable bigbro (holds the ProcessID of the process called by the father immediately before the current process).
- Stored as the local variable smallbro (holds the ProcessID of the process called by the father immediately after the current process).

The ProcessID can either be used as an argument for a function such as get_angle() or get_dist() or as a way of using the local or public variable of another process. For example, to use the x and y local variables of a process.

36.3 Example

```
Program example;

Declare Process SpaceShip()
    Public
        int speed = 50;
        String name = "Galactica";
    End
End

Global
    SpaceShip ship;
Begin

    ship = SpaceShip(); // ship now holds the ProcessID of an instance of SpaceShip
                        // Code in SpaceShip will be executed until the first frame; is reached

    say(ship.name + ": " + ship.speed); // We address the public variables of ship

    signal(ship,S_KILL);

    Repeat
        frame;
    Until(key(_ESC))

End

Process SpaceShip()
Begin
    name = "Pegasus";
    speed = 60;
    Loop
        frame;
    End
End
```

Used in example: say(), signal(), key(), Program, Declare, Public, Global, Begin, Repeat, Loop, frame

37 ProcessType

A processtype is a definition of a process; not the actual instance, but the type.
There can be multiple instances of one processtype with differing ProcessID's, but there's always one processtype of the same type with one ProcessTypeID, which is as logical as it sounds. For example, one can have a SpaceShip processtype (see the example in process). In the example code, an instance of SpaceShip is created, by calling SpaceShip(). Consider one wants to check in the SpaceShip if it collides with another SpaceShip. This can be done using the following.

```
if(collision(type SpaceShip))
    // collision code
end
```

The type statement expects the name of a processtype after it and converts it to the ProcessTypeID associated with the given processtype.

38 ProcessTypeID

38.1 Definition

A ProcessTypeID is a unique identification code, for a processType. A ProcessTypeID is smaller than 65536 (2^16).

38.2 Example

Kill all SpaceShip()s (see process):

```
Begin
    signal(type SpaceShip,s_kill);
End
```

signal() signals all processes of processType SpaceShip the signal to die. This is done by using the constant s_kill as the signal.

39 Public variable

39.1 Definition

A public variable is a variable that is specific to a process or function in the same way as a private variable. Unlike a private variable, however, a public variable can be accessed from the rest of the program, by use of the ProcessID of that process. This ProcessID must be stored in a variable of type ProcessName. It is like a local variable, but just for one ProcessType instead of for all.

Because of the way the compiler works, public variables are only accessible for processes and functions below the declaration (which in fact is pretty normal). To assist in this matter, the statement Declare was created.

To start the declaration of public variables, use Public.

39.2 Example

```
import "mod_say"
import "mod_proc"

Declare Process SpaceShip()
    Public
        int speed = 50;
        String name = "Galactica";
    End
End


Global
    SpaceShip ship;
End

Process Main()
Begin

    ship = SpaceShip();

    say(ship.name + ": " + ship.speed);

    signal(ship,S_KILL);

End

Process SpaceShip()
Begin
    Loop
        frame;
    End
End
```

Used in example: say(), key(), signal(), Program, Declare, Public, Global, Begin, Repeat, Loop, frame

This is for example when a ship is needed, of which the speed and name want to be accessible from the rest of the program. This way, one can easily have multiple instances of the same ProcessType, but still have the appropriate variables easily accessible.

Notice the use of Declare. If we wouldn't use it, we could just move the process SpaceShip above the main process. This will have the same result.

40 Region

Up to Local Variables

40.1 Definition

40.1.1 Local variable

INT region = 0

Region is a predefined local variable. **Region** holds the RegionID of the **region** in which the process' graphic should only be displayed in. By default this is region *0*, the whole screen.

The graphic of the process is only displayed in its region, even if the x and y coordinates are outside of the region, the part inside the region will still be displayed.

40.1.2 Concept

A region is a rectangular field inside the screen. It can be defined with define_region() and can be useful for displaying graphics in only certain parts of the screen and for the function region_out(). There are 32 regions (*0..31*) and region *0* is the whole screen.

Template:Locals

41 RegionID

41.1 Definition

RegionID

A RegionID is an identifier associated with a certain region.

42 Resolution

Up to Local Variables

42.1 Definition

42.1.1 Local variable

INT resolution = 0

Resolution is used to alter the precision of the position of processes on screen; the level of precision is defined by the value of resolution.

This simulating of fractions in positions is useful for calculations or animations in need of a high precision in order to work properly. It causes the coordinates of all processes to be interpreted as being multiplied by the value of the local variable resolution, associated with that process. So when a process' graphic is displayed, it will appear as if the process' x and y values were divided by the value of resolution. A resolution of `0` is interpreted as if it were `1`.

The default value of **resolution** is 0, so set it to 1 if the correct value is needed.

42.1.2 Screen Resolution

The resolution of a screen is the dimensions of the screen in pixels. Bennu's default screen resolution is 320×200 pixels. This can be altered by use of set_mode().

42.2 Example

```
import "mod_grproc"
import "mod_time"
import "mod_key"
import "mod_video"
import "mod_map"
import "mod_draw"
import "mod_proc"
import "mod_wm"

Process Main()
Begin

    // Set screen resolution to 320x200 with a color depth of 8bit
    set_mode(320,200,8);

    // Set the FPS to 60
    set_fps(60,0);

    // Set resolution for this process (try changing it to see the effect)
    resolution = 100;

    // Create a 200x200 cyan circle and assign its graphID to the local variable graph
    graph = map_new(200,200,8);
    drawing_map(0,graph);
    drawing_color(rgb(0,255,255));
    draw_fcircle(100,100,99);

    // Set size
    size = 10;

    // Set the coordinates at screen position (160,180).
    x = 160 * resolution;
    y = 180 * resolution;

    // Move around in circles while leaving a trail behind
    Repeat
        trail(x,y,graph,(int)(0.2*size),get_timer()+1000); // create a mini reflection of this process,
                                                           // lasting one second
        advance(3*resolution); // advance (3 * resolution) units (= 3 pixels)
        angle+=2000; // turn 2 degrees left
        frame;
    Until(key(_ESC)||exit_status)

OnExit

    let_me_alone();
    map_unload(0,graph);

End
```

```
Process trail(x,y,graph,size,endtime)
Begin

    // Get the resolution of the process calling this one
    resolution = father.resolution;

    // Remain existent until the specified endtime was reached
    Repeat
        frame;
    Until(get_timer()>=endtime)

End
```

Used in example: set_mode(), set_fps(), map_new(), drawing_map(), drawing_color(), draw_fcircle(), get_timer(), key(), let_me_alone(), map_unload(), advance(), **resolution**, graph, size, x, y, angle, exit_status

Here are a few screenshots with different resolutions to display the effect it can have.

Template:Image

Template:Image

Template:Image

Template:Image

The effect is clearly visible, so when you are moving processes with graphics around the screen, you might want to consider using a resolution of at least 10 in those processes.

Template:Locals

43 Scancodes

43.1 Definition

Scancodes are used to identify keys. This is used in the function key() and the global variable scan_code. Note that the global variable ascii is very different from this.

43.2 List

Constant	- *Value*
_ESC	- 1
_1	- 2
_2	- 3
_3	- 4
_4	- 5
_5	- 6
_6	- 7
_7	- 8
_8	- 9
_9	- 10
_0	- 11
_MINUS	- 12
_PLUS	- 13
_BACKSPACE	- 14
_TAB	- 15
_Q	- 16
_W	- 17
_E	- 18
_R	- 19
_T	- 20
_Y	- 21
_U	- 22
_I	- 23
_O	- 24
_P	- 25
_L_BRACHET	- 26
_R_BRACHET	- 27
_ENTER	- 28
_C_ENTER	- 28
_CONTROL	- 29
_A	- 30
_S	- 31
_D	- 32
_F	- 33
_G	- 34
_H	- 35
_J	- 36
_K	- 37
_L	- 38
_SEMICOLON	- 39
_APOSTROPHE	- 40
_WAVE	- 41
_L_SHIFT	- 42
_BACKSLASH	- 43

_Z - 44
_X - 45
_C - 46
_V - 47
_B - 48
_N - 49
_M - 50
_COMMA - 51
_POINT - 52
_SLASH - 53
_C_BACKSLASH - 53
_R_SHIFT - 54
_C_ASTERISK - 55
_PRN_SCR - 55
_ALT - 56
_SPACE - 57
_CAPS_LOCK - 58
_F1 - 59
_F2 - 60
_F3 - 61
_F4 - 62
_F5 - 63
_F6 - 64
_F7 - 65
_F8 - 66
_F9 - 67
_F10 - 68
_NUM_LOCK - 69
_SCROLL_LOCK - 70
_HOME - 71
_C_HOME - 71
_UP - 72
_C_UP - 72
_PGUP - 73
_C_PGUP - 73
_C_MINUS - 74
_LEFT - 75
_C_LEFT - 75
_C_CENTER - 76
_RIGHT - 77
_C_RIGHT - 77
_C_PLUS - 78
_END - 79
_C_END - 79
_DOWN - 80
_C_DOWN - 80
_PGDN - 81
_C_PGDN - 81
_INS - 82
_C_INS - 82
_DEL - 83
_C_DEL - 83
_F11 - 87
_F12 - 88
_LESS - 89
_EQUALS - 90

_GREATER - 91
_ASTERISK - 92
_R_ALT - 93
_R_CONTROL - 94
_L_ALT - 95
_L_CONTROL - 96
_MENU - 97
_L_WINDOWS - 98
_R_WINDOWS - 99
Template:Moduledocbox

44 Scroll window

44.1 Definition

A **scroll window** is a region in which a scroll has been started by start_scroll() and setup with the global variable scroll. This scroll window provides a view window into the scroll that has been setup there, with certain background and foreground graphics. The view window can be 'scrolled' over the scroll (and the graphics).

The use of scrolls can be various, but mostly they are used for scrolling games, like sidescrollers.

Processes can also be added to a scroll (see the global variable ctype), in which case a lot of things related to that process change.

- The graphic of the process is not drawn to the screen, but to the scroll window, using the region of the scroll.
- The z of an added process only has a meaning to the scroll window and compared to other processes added to the same scroll, meaning from a user perspective, the process is drawn at the z value of the scroll. If two processes are added to the same scroll, the one with the lower z value will be drawn over the other one.
- The x- and y-coordinates are not relative to the upper left corner of the *screen*, but relative to the upper left corner of the *scroll*.

You can control scrolls in multiple ways.

- Change the scroll coordinates directly using `x0`, `y0`, `x1`, `y1` in the global variable scroll. Be sure to alter the correct scroll.
- Tell the scroll to 'follow' another scroll, by using `scroll.follow`.
- Use a 'camera' process, which the scroll try to follow, by using `scroll.camera`.

More about scrolls:

- A scroll with a lower z value will be drawn over a scroll with a lower z value.
- The foreground is the plane to be controlled and the background moves relative to the foreground.

44.2 Example

```
Const
    SCREEN_WIDTH = 320;
    SCREEN_HEIGHT = 200;
    SCREEN_DEPTH = 8;
    MAP_WIDTH  = 500;
    MAP_HEIGHT = 400;
    SHIP_WIDTH  = 30;
    SHIP_HEIGHT = 30;
    SHIP_SPEED = 6;
End

Declare Process playership(int x, int y)
End
Declare Function int makeMap(int width, int height, int depth)
End
Declare Function int makeShipGfx(int width, int height, int depth)
End

Process Main()
Private
    int map;
Begin
    // Setup the screen
    set_mode(SCREEN_WIDTH,SCREEN_HEIGHT,SCREEN_DEPTH);

    // Create the background graphic
    map = makeMap(MAP_WIDTH,MAP_HEIGHT,SCREEN_DEPTH);

    // Start a new scroll
    // scrollNumber = 0
    // fileID = 0
    // foregroundGraphID = map
    // backgroundGraphID = 0: no background
    // region = 0: use entire screen
    // lockindicator = 0: don't repeat graphics
    start_scroll(0,0,map,0,0,0);

    // Start a controllable object on the scroll
    playership(320,200);

    // Wait till the key ESC is pressed
    Repeat
        frame;
    Until(key(_esc))

OnExit
```

```
    // Kill all other processes
    let_me_alone();

    // Clean up memory used for map
    unload_map(0,map);
End


Process playership(int x, int y)
Private
    int sp = SHIP_SPEED;
    int halfWidth;
    int halfHeight;
Begin

    // This process should be displayed on a scroll only
    ctype = c_scroll;

    // This process should be tracked by the scroll
    Scroll.camera = id;

    // Create a graph for this process
    graph = makeShipGfx(SHIP_WIDTH,SHIP_HEIGHT,SCREEN_DEPTH);

    // Obtain the value of half the width of the graphic
    halfWidth  = graphic_info(0,graph,G_WIDTH )/2;
    halfHeight = graphic_info(0,graph,G_HEIGHT)/2;

    // React on keys LEFT and RIGHT
    Loop

        // Adjust x,y on input
        x+=(key(_right)-key(_left))*sp;
        y+=(key(_down)-key(_up))*sp;

        // Make sure the ship stays on the scroll
        if(x > MAP_WIDTH-halfWidth)
            x = MAP_WIDTH-halfWidth;
        elseif(x < halfWidth)
            x = halfWidth;
        end
        if(y > MAP_HEIGHT-halfHeight)
            y = MAP_HEIGHT-halfHeight;
        elseif(y < halfHeight)
            y = halfHeight;
        end

        frame;

    End

OnExit
    // Clean up used memory for graph
    unload_map(0,graph);
End

Function int makeMap(int width, int height, int depth)
Private
    int n;
    int x1,y1;
    int m;
Begin

    // New map
    m = new_map(width,height,depth);

    // Put on some pretty stars
    rand_seed(100);
    for(n=0;n<1000;n++)
        map_put_pixel(0,m,rand(0,width),rand(0,height),rgb(150,170,200));
    end

    // Return it
    return m;

End

Function int makeShipGfx(int width, int height, int depth)
Private
    int g;
Begin

    // Return a map of one color (RGB(150,200,170))
    g=new_map(width,height,depth);
    map_clear(0,g,rgb(200,100,170));
    return g;

End
```

Used in example: set_mode(), start_scroll(), key(), let_me_alone(), unload_map(), graphic_info(), new_map(), rand_seed(), rand(), rgb(), map_put_pixel(), map_clear(), ctype, scroll

45 SongID

45.1 Definition

SongID

A SongID is an identifier associated with a certain song loaded by load_song(). The identifier can be used in various other functions, like unload_song() to unload the song.

46 Subroutine

46.1 Definition

A subroutine is a program segment that can be called and used by any other bit of the program. So basically, if there's something that you want to to quite a lot in your program, then you can write a subroutine for it and then just call this subroutine every time you want to to that. In Bennu this is achieved by the use of functions and processes

47 Text

47.1 Definition

A text is a text written on screen with texts functions, like write() or write_int(). A text is addressed using the associated TextID.

There are two kinds of texts:

- Static text is any text written with write(); it is static, because the content of the text cannot be changed after writing, but the text can be moved and deleted.
- Dynamic text is any text written with write_xxx() functions; it is dynamic because the content of the text always reflects the current value of the variable specified. Of course moving and deleting is also possible.

47.2 Writing texts

There are multiple ways to write texts: the write() function, write_xxx() functions and the write_in_map() function. With the first two, some the global variable text_z is important and with all three of them, the following are important:

- The functions set_text_color() and get_text_color()
- The font
- The alignment
- The global variable text_flags

Also the function move_text() can be handy, as it moves a previously written text to a different location.

47.3 Example

```
Program texts;
Const
    maxtexts = 10;
Private
    int textid[maxtexts-1];
    string str;
    float flt;
Begin

    // Set FPS
    set_fps(60,0);

    // Init text settings:
    text_z = 0;
    text_flags = 0;
    set_text_color(rgb(255,255,255));

    // Write some texts
    textid[0] = write(0,0,0,0,"FPS:");
    textid[1] = write_int(0,30,0,0,&fps);
    textid[2] = write_string(0,160,95,1,&str);
    textid[3] = write_float(0,160,105,0,&flt);

    // Show z workings
    set_text_color(rgb(50,150,150));
    textid[4] = write(0,20,20,0,"Underlying text");
    text_z = -1;
    set_text_color(rgb(255,255,255));
    textid[5] = write(0,22,22,0,"On top text");

    // Update the texts until ESC is pressed
    Repeat
        // Notice the texts get updated as the values of str and flt changes.
        // The same goes for the value of fps.
        str = "This program is running for " + timer/100 + " seconds.";
        flt = (float)timer/100;
        frame;
    Until(key(_esc));

    // Delete the texts (this section would be good for OnExit)
    for(x=0; x<maxtexts; x++)
        if(textid[x]!=0)
            delete_text(textid[x]);
        end
    end

End
```

Used in example: set_fps(), set_text_color(), write(), write_int(), write_string(), write_float(), key(), delete_text(), text_z, text_flags, fps

48 TextID

48.1 Definition

TextID

TextID is an identifier associated with a certain text. It is returned by various text functions, like write(), write_int(), write_string(), write_float() and move_text().

When a dynamic text is created, it has the color last set by set_text_color(). By default this is white (rgb(255,255,255)). Its Z value is equal to text_z at the moment of creation, which is -256 by default.

To move the dynamic text associated with a TextID, use move_text(). To delete the text, use delete_text(). There can be a total of 512 dynamic texts on screen simultaneously.

48.2 See also

- Text
- Text functions

49 Tutorial:Beginner's tutorial

This Beginner's tutorial aims to help beginning programmers to get to know Bennu. In its current state, the tutorial is reasonable advanced already, so if you don't understand something, please **leave a note**. Any comments or suggestions are also welcome.

50 The Beginning

Bennu, like every compiled language, has a compiler and an interpreter. The first one is used to convert the **source file**, the code that you write, to a file that the interpreter can understand. In this tutorial we are going to concentrate on the source file and move on from there.

50.1 Source File

A source file is simply a text file that respects certain syntax. This file can be created in any text editor, like the windows notepad. Bennu does not have an official IDE but there are patches and tutorials to make some free IDEs ready for Bennu (see Setting up Bennu).

For the sake of standardization there a few file extensions used primarily for Bennu sourcefiles:

- .prg: program code
- .inc: included code; code that is not meant to run stand-alone; header file (compare with C's .h).

Here's a basic source file:

```
import "mod_say" // import the module to output text to console, using say()

Process Main() // start the definition of the main process
Begin // start the code
    say("Hello World!");
End // end the definition of the main process
```

Used in example: mod_say, import, Process, Begin, End, say()

From Hello World

50.2 Introduction to processes

Like many other languages, Bennu has functions. Functions can be seen as parts of code you can run with only a single statement (the function call). The caller will wait for the function to finish. Let's clarify this with an example.

```
import "mod_say" // import the module to output text to console, using say()

Function int SayHelloWorld() // start the definition of a function called SayHelloWorld
Begin // start the code
    say("Hello World!");
End // end the definition of the function SayHelloWorld

Process Main() // start the definition of the main process
Begin // start the code
    SayHelloWorld();
    say("Hello again!"); // AFTER SayHelloWorld() is done
End // end the definition of the main process
```

Used in example: mod_say, import, Function, Process, Begin, End, say()

Here, we see the caller (`Main` and the called function `SayHelloWorld`).

Now you may be wondering why there are functions and processes. Processes are almost the same as functions, but with a slight difference we will discuss shortly. Try to run the same code, but making `SayHelloWorld` a process:

```
import "mod_say" // import the module to output text to console, using say()

Process int SayHelloWorld() // start the definition of a function called SayHelloWorld
Begin // start the code
    say("Hello World!");
End // end the definition of the function SayHelloWorld

Process Main() // start the definition of the main process
Begin // start the code
    SayHelloWorld();
    say("Hello again!"); // AFTER SayHelloWorld() is done
End // end the definition of the main process
```

Used in example: mod_say, import, Process, Begin, End, say()

You will notice it doesn't make a difference here.

50.3 More about processes

Bennu has a unique way of programming, because it makes use of processes and frames. Frames can be seen as a way of controlling the speed of your application. For example, you can tell Bennu to run at 60 frames per second, 100 frames per second or even 1000 frames per second. Of course, if

the machine can't handle it, 1000 FPS won't be reached. In each frame, every process performs its code and advances a frame.

To have processes run in parallel, just call it like you would a function. When a process is called, it causes the caller to wait for the process, also like a function. However, when the process reaches a **frame;** statement, it tells the caller it can continue and this is where the parallelism begins. It is important to note that it is **not** defined in what order the processes will be run after this. In most cases it is the order in which the processes were started, but that is not guarenteerd. Let's see this in action.

```
import "mod_say"

Process int SayHelloWorld()
Begin
    say("SayHelloWorld: 1"); // this gets run in the first frame
    frame; // wait for the second frame to start
    say("SayHelloWorld: 2"); // this one in the second frame
    frame; // wait for the third frame to start
    say("SayHelloWorld: 3"); // this one in the third frame
End

Process Main()
Begin
    SayHelloWorld();
    say("Main: 1"); // this gets run in the first frame and AFTER the first one SayHelloWorld
    frame; // wait for the second frame to start
    say("Main: 2"); // this one in the second frame, BEFORE the one in SayHelloWorld
    frame; // wait for the third frame to start
    say("Main: 3"); // this one in the third frame, BEFORE the one in SayHelloWorld
End
```

The output is:

```
SayHelloWorld: 1
Main: 1
Main: 2
SayHelloWorld: 2
Main: 3
SayHelloWorld: 3
```

In the first frame, `Main` waits for `SayHelloWorld`, so the `say`() in `SayHelloWorld` is executed before the one in `Main`. After that, the processes run in turns, in this case `Main`gets his turn before `SayHelloWorld`. However, it is possible to influence the order of execution, but more on that later.

50.4 Frames

Bennu's interpretation works with frames. You can limit the execution rate by limiting the allowed amount of frames per second. To tell Bennu a process is done for the current frame, the statement `frame;` can be used. Multiple processes will take turns in getting executed and will not be executed simultaneously. For the more advanced people reading this, this is indeed a lot like coroutines, where instead of *frame*, the keyword *yield* is used; In Bennu you can also use *yield* as you would use *frame*.

```
import "mod_say"
import "mod_time" // import a time module, for get_timer(), which returns the number
                  // of milliseconds after the program was started
Process Main()
Begin
    while(get_timer()<1000)
        say("Hello World! @" + get_timer());
        frame;
    end
End
```

Used in example: mod_say, mod_time, while, frame, say(), get_timer()

This example will output `Hello World! @...` as many times as possible in one second.

But suppose we were to limit the FPS to 25, then it would only output it 25 times, spread evenly over the second it ran. This would be pretty useful in cases. Currently, this is done by importing mod_video and using set_fps(). This means that each Bennu-frame the mod_video module 'tells' Bennu to wait until the next frame should start. *For example if the FPS is 25, the time between the start of frames is 40ms. If the execution of Bennu took 10ms, mod_video will tell Bennu to wait 30ms with execution, making sure the FPS is correct.*

50.5 Processes and functions

One of the key features of Bennu is the use of processes. The code in each process runs independently from each other, although never at the same time, but the execution rate is influenced by the frame statement. A frame-statement in a process says something like "this process is done for this frame" and the process will wait (at the frame-statement) for the next frame to begin and the process acquires running privileges again.

A function is almost the same, but not entirely independent. The process or function calling a function will wait for the function to return *even if that function contains a frame-statement.* A good example is the textinput tutorial.

Processes and functions are given a unique identification number when they are created. You can use this to manipulate its execution (using signal), access local variables and its public variables (using the `.` operator). We will discuss more on this later.

Now let's create another process:

```
import "mod_say"
import "mod_time"
import "mod_proc" // import a process manipulator module, for let_me_alone()

Process Another() // start the definition of another process
Begin // start the code
    Loop // endlessly say something
        say(get_timer() + " - Another");
        frame;
    End
End // end the definition of another process

Process Main()
Begin

    Another(); // start another process

    while(get_timer()<1000)
        say(get_timer() + " - Main");
        frame;
    end

    let_me_alone(); // kill ALL processes EXCEPT this one

End
```

Used in example: mod_say, mod_time, mod_proc, say(), get_timer(), let_me_alone()

Notice that Main and Another are executed one after the other. If we were to limit the FPS to 25, they both would run at 25FPS, one after the other.

50.6 Multiple source files

If you want to use multiple sources files - which you want for larger projects, you are able to accomplish this by using the include statement.

Consider the last example. We'll cut it up in a .prg and a .inc.

example.prg:

```
import "mod_say"
import "mod_time"
import "mod_proc"

include "example.inc" // include example.inc: it will be as if the contents of
                      // example.inc were located at this point in example.prg
Process Main()
Begin

    Another();

    while(get_timer()<1000)
        say(get_timer() + " - Main");
        frame;
    end

    let_me_alone();

End
```

example.inc:

```
import "mod_say"  // we use these modules in this file, so it is good practise
import "mod_time" // to (also) include them here

Process Another()
Begin
    Loop
        say(get_timer() + " - Another");
        frame;
    End
End
```

51 Variables and datatypes

51.1 Variables

51.2 Basics

There are multiple scopes of variables in Bennu:

- Global variables are variables available throughout the code (from the point of declaration)
- Constants are like global variables, but their value cannot be changed.
- Private variables are variables only available to the process or function in which they were declared.
- Public variables are like private variables, but they can also be accessed outside of the owning process.
- Local variables are like public variables, but they apply to all processes instead of a single one.

Bennu has various default types of variables:

type	- *can contain*
int (32bits)	- `-2147483648..2147483647` (only whole numbers)
dword (unsigned int)	- `0..4294967295` (only whole numbers)
short (16bits)	- `-32768..32767` (only whole numbers)
word (unsigned short)	- `0..65535` (only whole numbers)
char (8bits)	- `-127..128` (only whole numbers)
byte (unsigned char)	- `0..255` (only whole numbers)
float (32bits float)	- `almost any real number`
string	- text

To declare global variables, use Global:

```
Global
    int i;
    float f;
End
```

To declare constants, use Const. Notice that there is not a datatype declared:

```
Const
    i = 0;
    f = 0.0;
End
```

To declare private variables, use Private inside a process or function definition:

```
Process myprocess()
Private
    int i;
    float f;
End
Begin
End
```

To declare public variables, use Public inside a process or function definition:

```
Process myprocess()
Public
    int i;
    float f;
End
Begin
End
```

To declare local variables, use Local:

```
Local
    int i;
    float f;
End
```

51.3 Arrays, Structs and pointers

We can also use the basic types to form arrays, structs and pointers. Arrays are formed using brackets:

```
Global // or somewhere else it is allowed to declare variables
```

```
    int my_array_of_ten_integers[9]; // this makes an array of ten integers 0..9
End
```

We can access these integers like this:

```
my_int = my_array_of_ten_integers[0];
my_index = 9;
my_int = my_array_of_ten_integers[my_index];
```

Structs are formed using the keyword struct:

```
Global // or somewhere else it is allowed to declare variables
    Struct MyStruct // makes a struct with two members
        int i;
        float f;
    End
End
```

We can access members by using the [[.]] operator:

```
my_int = MyStruct.i;
my_float = MyStruct.f;
```

We can also make an array of structs:

```
Global // or somewhere else it is allowed to declare variables
    Struct MyStruct[9] // makes ten structs
        int i;
        float f;
    End
End
```

Pointers are variables which can 'point' to variables. They actually hold the address where that variable is located. If for some reason you move the variable (maybe because you used realloc()), the pointer no longer points to the variable. This is important to understand: the pointer points to a memory address. The operators * and & are used when dealing with pointers:

```
import "mod_say"

Global
    int my_int;
    int* my_int_pointer;
End
Process Main()
Begin
    my_int = 4;
    my_int_pointer = &my_int; // the & operator returns the address of the variable
    say("my_int: " + my_int);
    say("my_int_pointer: " + my_int_pointer);
    say("*my_int_pointer: " + *my_int_pointer); // the * operator dereferences the variable and returns the contents
End
```

Output:

```
my_int: 4
my_int_pointer: 003FD134
*my_int_pointer: 4
```

So you see how you can use pointers. If you don't understand them yet or don't see their use, just forget about them for now, but remember they are there for when you need them.

51.4 Usermade datatypes

Making your own types can be useful at times. In Bennu, your own types are structs, meaning they can hold multiple variables. You can read more about it here. A quick example:

```
Type Point
    float x;
    float y;
End

Global
    Point A,B;
End

Process Main()
Begin
    A.x = 0;
    A.y = 1;
    B.x = A.x;
    B.y = 6;
End
```

51.5 ProcessID, Public and Local

A process or function returns its ID when it reaches a frame-statement. We can assign this to a variable and use it later on.

We could rewrite the earlier example to this:

```
import "mod_say"
import "mod_time"
import "mod_proc" // import a process manipulator module, for signal()

Process Another()
Begin
    Loop
        say(get_timer() + " - Another");
        frame;
    End
End

Process Main()
Private
    Another a;
    int b;
End
Begin

    a = Another(); // start another process
    b = Another(); // start another process

    while(get_timer()<1000)
        say(get_timer() + " - Main");
        frame;
    end

    signal(a,S_KILL); // kill a
    signal(b,S_KILL); // kill b

End
```

You might wonder what the difference is between `Another a` and `int b`. The difference is that when you make a variable with the datatype the name of a process, Bennu can use this information to address public variables. Bennu does not know the type of process b is holding, so it can't access its public variables, because it can't know it has any. Locals however apply to all processes, so they can be accessed with both *a* and *b*.

Consider:

```
import "mod_say"
import "mod_time"
import "mod_proc"

Local
    int loc = 1;
End

Process Another()
Public
    int pub = 2;
Begin
    Loop
        frame;
    End
End

Process Main()
Private
    Another a;
    int b;
Begin

    a = Another(); // start another process
    b = Another(); // start another process

    say("a.loc = " + a.loc);
    say("b.loc = " + b.loc);
    say("a.pub = " + a.pub);
    //say("b.pub = " + b.pub); // not possible

    signal(a,S_KILL); // kill a
    signal(b,S_KILL); // kill b

End
```

52 Moving Up

52.1 Arguments, Parameters and Return

You can pass processes and functions information using parameters. They can also return a value. When the process or function reaches a frame-statement, its processID is returned, which can be saved in a variable, or otherwise used. If it reaches a return statement first, then it will return the value supplied with the return statement. If `return;` is used, or the process or function reaches neither a return-statement or a frame-statement, the ID is also returned. This sums up to: *a process or function returns its ID, unless otherwise specified*.

For example:

```
import "mod_say"

Process Main()
Begin
    say("multiply(3,4) = " + multiply(3,4));
End
Function int multiply(int a, int b)
Begin
    return a*b;
End
```

And:

```
import "mod_say"
import "mod_proc"

Process Main()
Begin
    say("an object with ID: " + object());
    say("an object with ID: " + object());
    say("an object with ID: " + object());
    let_me_alone();
End
Process object()
Begin
    Loop
        frame;
    End
End
```

52.2 Declare

Sometimes, you want to know how a process is declared before it is defined. When you declare something, it is not yet defined, but if you define something, it is declared. It is good practice to *only use things that are declared when used*, otherwise you could get lost by tracing weird bugs you caused. But what if you want to call process A inside process B *and* the other way around? You would have to declare both processes beforehand, using Declare:

```
Declare Process A()
End
Declare Process B()
End

Process A()
Begin
    B();
End
Process B()
Begin
    A();
End
```

You may have already noticed that this code is pretty bad: A will call B, B will call A, A will call B, etc.

You can use Declare for private variables and public variables too:

```
Declare Process A()
    Public
        int i;
    End
End
Declare Process B()
    Public
        float f;
    End
End

Process A()
Private
    B b;
```

```
Begin
    b = B();
    b.f = 5;
End

Process B()
Begin
   f = 1;
   say("f: " + f);
   frame; // let A continue
   say("f: " + f);
End
```

52.3 Strings

Strings are a special kind of datatype. They could be described as a dynamic array of characters, meaning that the length of the characters can vary. Strings are mostly used for holding text.

```
Const
    MSG_ERROR = "Oops, error occured!";
    MSG_SUCCESS = "Success!";
End

Process Main()
Private
    int error; // doesn't really do anything
Begin
    if(error)
        say(MSG_ERROR);
    else
        say(MSG_SUCCESS);
    end
End
```

Used in example: string, say()

Of course, you can do stuff with strings too. Let *a* and *b* be strings: `a + b = the concatenation of a and b ("ab"+"cd"="abcd")`. More functionality for strings can be found in mod_string.

52.4 Goto Labels

52.5 Further reading

language:

- basic statements

modules:

- angle,x,y,z,graph,pi etc with processes (make this article!)
- fpg,fgc: graphics libraries (make this article!)
- blendops (this one is somewhere)
- graphics article (make this article!)
- text
- window manager

actually every module should have an article about it, telling what's it for

53 Tutorial:Setting up Bennu on 64 bit linux

At the time of the writing, Bennu doesn't yet offer a native 64 bit version for linux (or any other OS, for that matter). The 32 bit version does, however, work flawlessly provided you have the adequate dependencies installed.

This page includes installation instructions for the 64 bit versions of some popular linux distros. In essence, the process for all the distros is the same, it's only the package names and installation methods that change.

Please note that an active internet connection will be needed to be able to install Bennu.

53.1 Fedora Core 11 x86_64 (Leonidas)

Fedora Core 11 uses the YUM package manager, which is, in turn, based in the RPM package system. We'll be using a graphical interface to YUM called Packagekit.

53.1.1 Launching Packagekit

To launch the package manager, go to "System > Administration > Add/Remove Software" in the menu at the top of your screen (in the default GNOME configuration) as shown in the screenshot below.

You might be asked for the administrative password; enter it.

File:System-addpackages.jpg

53.1.2 Installing the required packages

In order to get Bennu to run you must install the following native packages and their corresponding 32 bit versions. To make things quicker, you can search for the package names in the search box, as shown in the screenshot below:

- SDL
- SDL_mixer
- libogg
- zlib
- libpng
- libX11
- libXrandr
- libXext
- libXrender
- libgcc

File:System-install-sdli586.png

Notice there are two SDL packages with the same name? The one ending in *(x86_64)* is the native 64 bit version of the package. The one ending in *(i586)* is the corresponding 32 bit version.

Please note that Packagekit won't let even show you the 32 bit version for a package whose 64 bit (*x86_64*) counterpart is not installed. It will be faster if you install all the 64 bit versions of the packages on the list above and then proceeding to install the 32 bit versions.

When you tell Packagekit to install the packages for you, it will check for mising dependencies and ask for your permission to install them. Depending on what packages you have already available for your system the number of missing dependencies may change.

File:System-additional-dependencies.png

53.1.3 Doing all of the above with a simple command line instruction

All of the above should be equivalent to doing -in a terminal where ou're logged in as root- the following:

```
yum install SDL SDL.i586 SDL_mixer SDL_mixer.i586 libogg libogg.i586 zlib zlib.i586 libpng libpng.i586 libX11 libX11.i586 libXrandr libXrandr
```

53.1.4 Actuallly installing Bennu

Now you can go to the Bennu download page and download the linux installation script to a location easy to find. Your home directory is a good choice, if you have no preference.

Now -supposing you saved the download script to your home directory- open a terminal and login as the root user and launch the installation script by doing:

```
su (this logs you as the root user, it will ask for root's password, enter it)
   chmod a+x bgd-1.0.0RC7\(r99\)-linux-installer.sh (replace with the version number you downloaded)
   ./bgd-1.0.0RC7\(r99\)-linux-installer.sh (Again, replace with the installer script filename you downloaded)
```

53.2 Ubuntu 9.04 x86_64 (Jaunty)

Ubuntu 9.04 uses the APT package manager, which is, in turn, based in the DEB package system. We'll be using the Synaptic graphical front-end to APT.

53.2.1 Running Synaptic

To run Synaptic, go to "System > Administration > Synaptic Package Manager" as shown below.

You might be asked for your password; enter it.

File:Synaptic-open.png

53.2.2 Installing required deps

For Bennu to run, you need to install the *ia32-libs* package and its dependencies. Search for it in Synaptic.

File:Synaptic-search-ia32.png

Depending in your system configuration, Synaptic might ask for permission to install additional packages:

File:Synaptic-install-dependencies-ia32.png

53.2.3 Doing all of the above with a single command

You can install ia32-libs and all of its dependencies by executing the following command in a terminal:

```
sudo apt-get install ia32-libs
```

You might be asked for your password; enter it. You might also be asked for permission to install additional packages; give the system permission.

53.2.4 Actually installing Bennu

Go to the Bennu download page and download the linux installation script to a location easy to find. Your desktop directory is a good choice, if you have no preference.

Now, open a terminal through "Applications->Accessories->Terminal" and run the following commands:

```
cd Desktop (Replace with the directory to where you downloaded the installation script).
   chmod a+x bgd-1.0.0RC7\(r99\)-linux-installer.sh (Replace with the file name you downloaded).
   sudo ./bgd-1.0.0RC7\(r99\)-linux-installer.sh (You might be asked for your password, enter it).
```

53.3 Run Bennu!

Now you should be ready to use Bennu! You can, for example, try the Beginner's tutorial, or your own sample code:

File:BennuGD-running-in-linux.png

54 Tutorial:Setting up Bennu on Linux

Bennu doesn't have an official release yet, but the release candidate versions are fully functional, and pretty stable. This page describes the process for installing Bennu through the recommended official installer or a unofficial online package repository.

54.1 Installing with the official script (for most Linux systems)

The package includes the three essential items: bgdc, bgdi and libbgdrtm.so. The first one (**B**ennu**GD C**ompiler) compiles your code to bytecode, which bgdi (**B**ennu**GD I**nterpreter) reads when you want to run your program. See their pages for more detailed info about them. In addition to these files, the package contains the official Bennu Modules.

To install Bennu on your Linux box, go to the Bennu download page and download the linux installation script to a location easy to find. Your home directory is a good choice, if you have no preference.

Now -supposing you saved the download script to your home directory- open a terminal and login as the root user and launch the installation script by doing:

```
su (this logs you as the root user, it will ask for root's password, enter it)
   chmod a+x bgd-1.0.0RC7\(r99\)-linux-installer.sh (replace with the version number you downloaded)
   ./bgd-1.0.0RC7\(r99\)-linux-installer.sh (Again, replace with the installer script filename you downloaded)
```

or, if that fails for you:

```
chmod a+x bgd-1.0.0RC7\(r99\)-linux-installer.sh (Replace with the file name you downloaded).
   sudo ./bgd-1.0.0RC7\(r99\)-linux-installer.sh (You might be asked for your password, enter it).
```

54.2 Installing through the Launchpad PPA (for Ubuntu)

If you are running Ubuntu or Debian in a 32 bits system, you can also install Bennu using an online repository (also known as PPA). This is very convenient, as you'll get updates automatically for Bennu, the official modules and other related software (i.e.: syntax highlighting for GTK+ based editors; like GEdit) through the system's standard tools **but** you'll be using a non-supported development version. This means that from time to time, and although it's unlikely, your Bennu installation might break.

Appart from that, the Launchpad PPA package is composed of the same parts as the supported one, plus many other packages not found elsewhere prepackaged for linux.

54.2.1 Adding the repository to your system

To install the packages from the Launchpad PPA, you have two option:

- In Ubuntu Karmic & over, just go to **System->Administration->Software Sources** then select **Third Party Software** tab and click **Add**. Then paste the following line in the text box:

```
ppa:josebagar/ppa
```

Now click on **Add source** and **Reload**.

- You can download this package and install it by double-clicking on the downloading file, as shown in the screenshot below. This will add the repo to the system and add the repository key to you installation so that you can be sure that the packages come from the repository and not from elsewhere.

File:Unofficial-bennugd-repo-installation-ubuntu.png

- In case you want to add the repository manually, you can also go to https://launchpad.net/~josebagar/+archive/ppa/ and follow the instructions there to add the repository and the signing key for the packages.

54.2.2 Actually installing Bennu

Once you've added the repository, open "System->Administration->Synaptic Package Manager".

File:Synaptic-open.png

In Synaptic, click on the "Refresh" button and then search for "bennugd", as shown below. File:Synaptic-search-bennugd.png

Once found, choose the packages from there that you want to install in your system and press "Apply".

Installing "bennugd-core" and "bennugd-modules" will give you the same setup as the official installation script, but you can also install unofficial additions to Bennu.

54.3 Testing your installation

To test the Bennu installation, you can use the Hello World example or your own code (the screenshot below shows some random code running in a Bennu installation). Save that code to a text file in a known location and navigate to that folder with a terminal.

Now, on the terminal, run:

```
bgdc helloworld.prg
```

This will generate a new file with the name "helloworld.dcb". Now you can run it:

```
bgdi helloworld
```

See what happens :)

54.4 IDE

To code comfortably, you want to use an IDE. This makes life much easier when it comes to coding, compiling and running your code, because it has syntax highlighting, compile/run hotkeys and more features.

For Bennu there is not yet an official IDE, so a universal one will have to be used. If you installed Bennu through the PPA, installing the "gtksourceview2.0-bennugd" package will add syntax highlighting to the standard GNOME editor, called GEdit.

54.5 Start coding

You can read on how to start coding in the Beginner's tutorial.

54.6 A note on 64 bit linux systems

Bennu doesn't yet offer a native 64 bit version. The 32 bits version (the one found in the download section of the website) should work just fine in 64 bit systems provided the required libraries are installed.

For detailed instructions on how to set up Bennu for your 64 bit system in the Setting up Bennu on 64 bit linux page.

55 Tutorial:Setting up Bennu on Windows

55.1 Download

Bennu doesn't have an official release yet, but the beta versions are fully functional. See the Latest_Bennu_version.

55.2 Package

The package includes the three essential items: bgdc.exe, bgdi.exe and libbgdrtm.dll. The first one (**B**ennu**GD C**ompiler) compiles your code to bytecode, which bgdi.exe (**B**ennu**GD I**nterpreter) reads when you want to run your program. See their pages for more detailed info about them, for now we will focus on basic operations.

In addition to these files, the package contains the Bennu Modules, adding functionality to Bennu. Some of these modules require additional libraries, like SDL. For your convenience, these libraries are bundled together in the package dlls-externals-mandatory-pack.rar.

55.3 Setting up

55.3.1 Basic

Bennu doesn't need installation, just extract the downloaded files to a folder:

1. The Bennu Core (bgdi, bgdc, bgdrtm)
2. The Bennu Modules
3. The external libraries (like SDL)

You can now use Bennu to compile and run programs, for example a Hello World example.

To compile, do:

```
bgdc helloworld.prg
```

This will generate a new file with the name "helloworld.dcb". Now you can run it:

```
bgdi helloworld
```

See Hello World for the code.

55.3.2 Path

If you are using commandline and wish to have easy access to executables, you can add the directory they're in to the global PATH. To do this, ***add*** the following to it: `;<path to exes>`, where *<path to exes>* is the path to the executables (bgdi.exe and bgdc.exe). Make sure the modules are in a known path too, otherwise they cannot be found)

For example, your PATH was `C:\Windows\System32\;C:\WINNT\` and you placed the executables in `D:\Bennu\bin\`, then the *new* PATH will be: `C:\Windows\System32\;C:\WINNT\;D:\Bennu\bin\`

55.3.2.1 Windows XP

- **Start -> Settings -> Control Panel -> System -> tab [Advanced] -> button [Environment Variables]**
- Under **System variables**, find the variable *PATH* (or *path* or *Path*).

55.4 IDE

To code comfortably, you want to use an IDEs. This makes life much easier when it comes to coding, compiling and running your code, because it has syntax highlighting, compile/run hotkeys and more features.

For Bennu there is not yet an official IDE, so a universal one will have to be used. For some there are patches and for some there is a small tutorial. The tutorial ones are more descriptive and flexible, while the patch ones are quicker (but dirtier).

55.4.1 IDE tutorials

These tutorials provide a way to setup the IDE of your choice step by step, without overwriting any of your previously made customizations. <DPL> titlematch=Setting up Bennu with % mode=userformat listseparators = ,\n* {{#sub:%TITLE%|{{#len:Setting up Bennu with }}|0}},, </DPL>

55.4.2 IDE patches

To see the original IDEs article click here

WARNING: these patches assume a **fresh** install, otherwise other customizations may be overwritten!

This is a list of IDEs and their respective patches. The installation is as follows:

1. Install the IDE.
2. Unzip the patch archive to the folder of the IDE. For some IDEs more actions are required, which are listed below.
3. Place the Bennu executables and their DLLs in the folder ?C:\Bennu\Bin\" (you can alter this after the install).

All patches provide syntax highlighting, compile and run hotkeys, output capturer. Some also contain basic templates of programs (in the style of "Hello World." and/or shortcuts to insert code templates.

IDEs:

55.4.2.1 Crimson Publishing

- Excellent editor, with a column edit mode like UltraEdit.
- Syntax highlighting
- Template "hello world"
- Ctrl+F1 = Compile
- Ctrl+F2 = Run
- Download the patch here

55.4.2.2 PSPad publishing

- Syntax highlighting
- Template "hello world"
- Shortcuts to insert code templates.
- Ctrl+F9 = Compile
- F9 = Run
- Execute the file ?PSPad-Bennu-Install.bat" to complete the installation.
- Download the patch here

NOTE: This IDE cannot capture console output.

55.4.2.3 Ultraedit

- Syntax highlighting
- Compile and run shortcuts
- List of functions and processes in the current PRG.
- Download the patch here

NOTE: Read the file "install.txt" contained in the package.

55.5 Start coding

You can read on how to start coding in the Beginner's tutorial.

56 Tutorial:Setting up Bennu with ConTEXT

Setting up Bennu with ConTEXT.

It is assumed Bennu Binaries are downloaded in a certain folder, thus in which bgdi.exe and bgdc.exe will be. Here and there this folder will be denoted as `XXXXXX`. You will need to replace `XXXXXX` with the path to the directory you extracted the binaries to.

For example, you extracted the binaries to `D:\Bennu\bin\`, then you will need to replace `XXXXXX` with `D:\Bennu\bin` (watch that last `\`, there only needs to be one).

56.1 Advantages

ConTEXT is a pretty handy text editor. It has quite a range of possibilities and the highlighter file is very up to date. When it is setup like described below, certain files are only needed in one directory: the bgdi/bgdc directory. These certain files contain: include files (*.inc), Fenix header files (*.fh), Bennu header files (*.bh) and DLL's (*.dll). Actually, any file can be included/imported if it's in the bgdi/bgdc directory. This brings advantages like saving disk space, updating a certain file is much easier and automatically affects all programs using it, not just one.

56.2 Syntax Highlighter

1. Place bennu.chl in the folder ConTEXT\Highlighters (Note: This is the fenix.chl and very similar. There is not a full Bennu ConTEXT Highlighter file as of yet.)
2. Restart ConTEXT
3. File -> New
4. Go to Tools -> Set Highlighter -> Customize Types..., select Foxpro, click on Edit and change "prg" to "prg2"; select Object Pascal, click on Edit and change "inc" to "inc2".

You can meddle with the colours all you like: Options -> Environment Options -> Colors. You can also edit the colours in bennu.chl, if you find this easier. This can be handy for changing all backgrounds in a quick way.

Should you add native Bennu keywords to the highlighter, which were forgotten, be so kind as to notify of the addition on the bennu.chl discussion page.

56.3 Hotkeys

The setting up here happens in Options -> Environment Options -> [Execute Keys]. To add hotkeys for Bennu, click on the button Add and enter `prg,inc,fh,bh` for extensions. Noted are generally used hotkeys between brackets, but this can be altered.

56.3.1 Compile with Debug (F9)

1. Setup Compile hotkey:
    1. Execute: `XXXXXX\bgdc.exe`
    2. Start in: `%p`
    3. Parameters: `-g -i "XXXXXX" %n`.
    4. Save: `All files before execution`.
    5. If you are having trouble with spaces in filenames, enable Use short DOS names (only enable if you have trouble!)
    6. Enable Capture console output.
    7. Compiler output parser rule: `%n:%l:*`.
    8. Enable Scroll console to the last line, because the first part of the output is less helpful.

- Replace `XXXXXX` by the path to the directory containing bgdc.exe.
- Note that you can include codefiles and libraries located in the XXXXXX directory automatically now (that's what the -i parameter does).

56.3.2 Compile (F10)

1. Setup Compile hotkey:
    1. Execute: `XXXXXX\bgdc.exe`
    2. Start in: `%p`
    3. Parameters: `-i "XXXXXX" %n`.
    4. Save: `All files before execution`.
    5. If you are having trouble with spaces in filenames, enable Use short DOS names (only enable if you have trouble!)
    6. Enable Capture console output.
    7. Compiler output parser rule: `%n:%l:*`.
    8. Enable Scroll console to the last line, because the first part of the output is less helpful.

- Replace `XXXXXX` by the path to the directory containing bgdc.exe.
- Note that you can include codefiles and libraries located in the XXXXXX directory automatically now (that's what the -i parameter does).

56.3.3 Run (F11)

1. Setup Run hotkey:
    1. Execute: `XXXXXX\bgdi.exe`.
    2. Parameters: `"%p%F"`.
    3. Save: `Nothing`.
    4. If you are having trouble with spaces in filenames, enable Use short DOS names (only enable if you have trouble!)
    5. Enable Pause after execution

- Replace `XXXXXX` by the path to the directory containing bgdi.exe.

56.3.4 Lookup on this wiki (F12)

1. Download lookup.zip: extract into bgdi/bgdc folder, if not done already.
2. Setup Lookup hotkey:
    1. Execute: `XXXXXX\lookup.bat`.
    2. Parameters: `%w`.
    3. Save: `Nothing`.

- Replace `XXXXXX` by the path to the directory you placed lookup.bat in.

This will allow you to hit a button (F9-F12) to lookup the word currently under your cursor in ConTEXT on BennuWiki. Alternatively you can use `%?` instead of `%w`, to enter the word you're looking for. A possible setup is `%w` on F9 and `%?` on F12.

56.4 File Associations

The setting up here happens in Options -> Environment Options -> [File Associations].

Repeat the following for the extensions wanted (prg, inc, fh, bh):

1. Click on the button Add and enter an extension.

57 Tutorial:Squarepush's side-scrolling Shoot'em Up Tutorial

Starting to code your first game is always a challenge but it is also the most enjoyable part. In this tutorial we will create a simple shoot'em up, to learn more advanced things in Bennu.

If you are not an experienced programmer it could be of help if you avoid this tutorial for now, especially if you are not familiar with programming. If so have a look at the Beginner's Tutorial. Otherwise, read on!

If you are an experienced programmer and have yet to set up Bennu, it is recommended to first check the tutorials on how to set up Bennu.

Even if you know other programming languages, you may think you don't understand much of Bennu now. However, the concepts will become clearer after this tutorial.

57.1 Outlining the game

We are going to write a very basic side scrolling shoot'em up in 2d.

The game resolution is to be 320x240 in 16-bit colour.

The game is set across 2 levels, scrolling right. The background will scroll horizontaly, to the left, to show your ship advancing through the level. In fact this game is going to be rather similar to R-Type the arcade game - just quite simplified.

You are effectively flying through a organic alien ship of some monstrous proportion, and the level is quite tunnel-like.

Basic Game rules:

There there are advancing waves of alien ships that fire bullets:

- If you are hit by a bullet you lose some ship armour (health).
- If you collide with a ship you lose all your health instantly. This means, you lose 1 ship and in losing a ship you are returned to the beginning of the level.

But at the end of the level, for excitement value and playability we have a large alien ship, that is the end of level guardian!

Summary of the Game:

- We are programming a very basic side scrolling shoot'em up in 2d
- The screen resolution is to be 320x240, with 16-bit colour depth.
- The object of the game is to shoot the advancing waves of ships until they explode.
- You must survive the onslaught of ships which fire bullets. Both ships and bullets can eventually destroy your ship.
- After so many waves of enemy ships, you reach a large guardian (or boss) alien ship.
- You must survive the weapons of the end-of-level guardian and successfully destroy it.
- In doing this you complete the level!

57.2 Getting started - Setting up the screen

First of all we need to set up the screen. Firstly, the game as stated before is to be 320x240 resolution. The frame rate is to be 60 frames per second.

So we start with this piece of code:

```
// Shoot'em up Tutorial Game
//
// A side scrolling shooter for Bennu Wiki - written by Squarepush

import "mod_video"

Process Main ( )

Begin

     set_mode ( 320 , 240 , 16 , MODE_WAITVSYNC );

     set_fps ( 60 , 1 );
```

Firstly we have to import mod_video to later give video instructions.

Then we begin the main process. Begin proceeds the start of the main process code.

In the code we have set_mode to 320 width by 240 length and at 16-bit colour. The setting MODE_WAITVSYNC is a selection from render_flags. This selects (in this case) what is basically a Vertical Sync on option.

Then we set_fps (frames per second) to 60 with frame skip of 1. Frame skipping is ideal if you think your game may slow down on lower spec computers, as it will decide to skip displaying a frame if needed.

57.3 Adding a background and a scroll

Now we have the screen resolution set-up, we want to add a scroll and a background. For we want the background to be independent of the game graphics.

So firstly, we need to load in the game graphics.

Here you can download the graphics file or .fpg, for use in this tutorial.

Make sure you place the FPG file ("shootem.fpg") in a suitable place for access by the program. And note the path to it, for example - "/ShootemData/shootem.fpg"

So, now check out this code:

```
// Shoot'em up Tutorial Game
//
// A side scrolling shooter for Bennu Wiki - written by Squarepush

import "mod_video"
import "mod_map"
import "mod_scroll"
import "mod_screen"
import "mod_key"
import "mod_proc"

Global

int shootemGfx;

End

Process Main ( )

Begin

Loop

     shootemGfx = load_fpg ( "shootemup.fpg" );

     scale_mode = scale_normal2x;
     set_mode ( 320 , 240 , 16 , mode_waitvsync );

     set_fps ( 60 , 1 );

     SetupScreen ( );

     Repeat

          frame;

          If ( key(_esc) )
               exit();
               End

          Until ( key(_space) )

     Repeat

          frame;

          Until ( key(_esc) )

End

End

Process SetupScreen ( )

Private

int f;

Begin

     define_region ( 1 , 0 , 0 , 320 , 212 );
     start_scroll ( 1 , shootemGfx , 50 , 0 , 1 , 1 );
```

```
    scroll[1].speed = 1;

    Loop
            For ( f = 1 ; f < 5 ; f = f + 1 )
        frame;
        End
            scroll[1].x0 = scroll[1].x0 + 1;
    End

End
```

We are importing more modules:

mod_map mod_scroll mod_screen mod_key mod_proc

You can if you wish, look these up to see what they enable Bennu to do. We are importing one at a time as used in the code. This is seen as good practice.

Now, in the first Repeat loop, we insert an exit command, so that the game exits from here if the escape key is pressed. It ceases to Repeat if the space bar is pressed, and the Game process is called.

We also prepare another loop for the game part, which we will fill in later on.

In process SetupScreen we make what is called a region, with region_define. In this case it is simply the game window. The first region in this tutorial is **nearly** the size of the screen - there is a space left at the bottom for a panel which will have a few things on it. But that is made later on in the tutorial.

A section about scrolls:

Next you can see we are starting a scroll by calling start_scroll. In short we use scroll [1] and choose the background graphic, which was loaded and assigned to "shootemGfx" earlier. The number of the graphic/map in the .fpg file is 100. We ask to use region 1 which we set up earlier too, with locking indicator 1, which is perfect for horizontal moving scrolls. It will also repeat-tile the background graphic for us, so it covers the whole screen's view and does not simply scroll off the screen!

Lastly we employ a scroll struct variable for x movement. This works out as scroll[1].x0. The scroll movement we want is slower than 1 pixel per frame, so we have to frame ahead 4 times for every pixel movement.

57.4 First time running the game (compilation)

Now, finally you can try and compile the program! You can download this first stage in a .prg here

There isn't much to see, but the main thing is we have our basis for the game, with a side-scrolling background!

If something went wrong in compilation, or you have a blank screen, try checking your code for errors. Also, check the file path to the .fpg file is correct.

57.5 Adding the ship

We are now going to add a new process for the ship. It will simply move around the screen, controlled by the arrow keys. Interested?

Also we are making a mock title screen - when the user loads the program he/she will arrive here.

Let's see the code with a ship process added:

```
// Shoot'em up Tutorial Game
//
// A side scrolling shooter for Bennu Wiki - written by Squarepush

import "mod_proc"
import "mod_grproc"
import "mod_screen"
import "mod_key"
import "mod_text"
import "mod_video"
import "mod_map"
import "mod_scroll"
import "mod_wm"
import "mod_rand"

Global

int shootemGfx;
int ship_id;

End

Process Main ( )

Begin
```

```
Loop

     shootemGfx = load_fpg ( "shootem.fpg" );

     set_icon ( shootemGfx , 1 );
     set_title ( "Shoot'em Up Tutorial Game" );

     scale_mode = scale_normal2x;
     set_mode ( 320 , 240 , 16 , mode_waitvsync );

     set_fps ( 60 , 1 );

     SetupScreen ( );

     TitleScreen ( );

     Repeat

          frame;

          If ( key(_esc) )
               exit();
               End

          Until ( key(_space) )

     Game ( );

     Repeat

          frame;

          Until ( key(_esc) )

End

End

Process SetupScreen ( )

Private

int f;

Begin

     define_region ( 1 , 0 , 0 , 320 , 212  );
     start_scroll ( 1 , shootemGfx , 50 , 0 , 1 , 1 );
     scroll[1].speed = 1;

     Loop
          For ( f = 1 ; f < 5 ; f = f + 1 )
               frame;
               End
          scroll[1].x0 = scroll[1].x0 + 1;
     End

End

Process TitleScreen ( )

Begin

     write ( 0 , 160 , 150 , 4 , "Squarepush's Shoot'em Up Tutorial Game" );
     write ( 0 , 160 , 170 , 4 , "PRESS SPACE TO PLAY" );

     REPEAT
     frame;
     UNTIL ( key(_space) )

     delete_text ( all_text );

End


Process Game ( )

Begin

     ship_id = Ship ( );

     End

Process Ship ( );

Private

     int xm;
     int ym;
```

```
Begin

     resolution = 10;
     file = shootemGfx;
     region = 1;
     graph = 1;

     x = 200;
     y = 1060;

     Loop

          If ( key(_right) || key(_left) )

          If ( key(_right) )
               If ( ! key(_left) )
                              xm = xm + 8;
                                  end
               end

          If ( key(_left) )
               If ( ! key(_right) )
                    xm = xm - 8;
                    end
               end

          If ( xm > 24 )
                    xm = 24;
                    end

          If ( xm < -24 )
                    xm = -24;
                    end

          else

                If ( xm > 0 )
                     xm = xm - 2;
                     end

                If ( xm < 0 )
                     xm = xm + 2;
                     end

              end

         If ( key(_up) || key(_down) )

          If ( key(_up) )
               If ( ! key(_down) )
                              ym = ym - 8;
                                  End
               End

          If ( key(_down) )
               If ( ! key(_up) )
                    ym = ym + 8;
                    End
               End

          If ( ym > 24 )
                    ym = 24;
                    End

          If ( ym < -24 )
                    ym = -24;
                    End

          else

                   If ( ym > 0 )
                        ym = ym - 2;
                            End

                   If ( ym < 0 )
                        ym = ym + 2;
                            End

              End

                   If ( y > 1980 )
                        y = 1980;
                    End

                   If ( y < 140 )
                        y = 140;
                        End

                   If ( x > 3000 )
```

```
                    x = 3000;
                End

               If ( x < 200 )
                    x = 200;
                    End

          x = x + xm;
          y = y + ym;

     frame;

     End

End
```

The ship process is called, and the id of the process is stored in ship_id. We will need this to access the ship process id values later.

As you may notice, the ship process has constants x and y co-ordinates. When you begin a new process these are usually at 0 to begin with. So we set an initial co-ordinate for them.

We put the ship in region 1, but we don't attach it to the scroll as the scroll will be independent to the front game graphics.

I've written some code to control the ship, which you can analyse - really most of this is to ensure the ship moves smoothly, while being at a decent rate to move around the screen. I defined xm and ym (movement factors) to act like additives to the x and y. And you can see i made a value for resolution. Which was 10. This is to ensure smooth movement again. Remember - when changing the resolution you must change all screen co-ords to match, in this case x10.

58 Variable

A variable is a container containing a value. This value can vary, hence the name variable, as opposed to a constant of which the value cannot be changed. A variable can be of any datatype and can contain a value according to its datatype. For example, an integer has whole numbers between -2^31 and +2^31-1 (e.g. 23) , a float has a floating point number (e.g. 2.674) and a string has a series of characters (e.g. "Hello World!").

There is a number of predefined variables, like local variables and global variables. These variables are predefined, which means they exist without the programmer making them. They can contain useful information or perform an action when changed.

A function or process can accept values when called. The value it's called with is called an argument and the variable or value inside the function or process is called a parameter.

58.1 See Also

- A list of predefined variables
- Constant

59 Windgate's tutorial

File:Trinit Logo.png

Trinit association logo
This tutorial, originally made by Windgate for Trinit, tries to serve as a great starting point for those new to Bennu.

It's still a work in progress, but this article will be updated when new lessons are translated.

59.1 Contents

60 Graphics I

When creating a video game, the first thing you'll need is graphics.

You could start with simple graphics made with Paint or a similar simple drawing program, but if you're going to take use several hours of work for your video-game, it's better to start with some nice graphics as can be your preferred game characters or the scenarios for that old game you spent hundreds of hours playing long ago.

First of all, create a folder where you'll store your work.

60.1 Sprites

A spriteset is a table of graphics with all the animations for a particular character.

File:Windgate tuto image 1-1.png

1-1 Spriteset Example
You'll have to choose several spritesets. You can find a great collection of spritesets on the Internet, be it in sites as http://spritedatabase.net or simply by searching in Google Images for ?sprite? and ?ripped?.

When choosing your spritesets please take into account:

Views	Top, side or isometric. It's usually easier to start with side-view spritesets.
Animations	Your character should NOT be able to perform more actions than those defined in its spriteset.
Resolutions	The size in pixels of each animation (Sprite) is a good indication of the quality of the animation.

Notice that Image I corresponds to an isometric-view spriteset, without any jump animation and with a medium resolution: not as high as that found in Street Fighter II nor as low as that found in the first Super Mario Bros.

60.2 Backgrounds

Background images are the images where your game's action will take place.

File:Windgate tuto image 1-2.png

1-2 Spriteset Example
You should choose several background images. Again, have a look at websites such as http://vgmaps.com, or look for them in Google Images using ?scroll?, ?background? and ?ripped? as keywords.

Just like sprites, background images can also use one of the views listed above so be sure to choose one that uses the same view as your sprites do. As stated above, it's better to start with a side-view game.

Regarding resolution, try to not use images with a very low resolution (below 320x240 pixels) as they might not fill the screen completely but also be sure not to use images that are too big (over 4096x4096 pixels) as they use too many resources and they could decrease your game's performance. We can always rescale the graphics, but that's something we'll try to avoid as of now.

Please note that background imagess and sprites are completely different things. In ripped background images you might sometimes find enemies or items kept from the original games, those are useless to us and are called ?dirty?, as you'll probably have to clean them before using them in your game.

Please note that in Image 2 can be a top-view background image, at a somewhat low resolution and that can be considered dirty if your idea is to make a game where the bushes can be cut down.

61 PUBLIC

We've already worked with most of Bennu's main utilities, but there's still a very important aspect of Bennu we have not dealt with that will certainly open you the doors to a very broad field of Bennu programming.

You can see that, by using PRIVATE variables, you cannot access data belonging to a process even if you know its process ID. That limitation makes it hard to guess things as simple as -for example- determining how much damage a certain shot makes in a certain enemy.

You could implement such a feature by using LOCAL variables, but this is discouraged as it modifies all the processes' varspace to hold that variable, no matter what their type is, thus increasing the amount of memory used.

To avoid that problem you can use variables with PUBLIC scope that only modify the varspace of those processes that have that variable declared. Its usage seems to be very simple and convenient but has a few rules that must be followed, as shown below.

61.1 DECLARE sentence

Although you can declare the PUBLIC variables in the same way you do with PRIVATE variables that won't generally allow us to access them, as Bennu will give that process type a pre-defined varspace and only contains some basic vars like graph, x, y, etc.

To overcome this limitation, Bennu implements the DECLARE sentence. You must use it for all the processes that declare PUBLIC variables and before the process declaration. In the example, we're going to declare some PUBLIC data for our main character and thus, before declaring its process we must add the following lines:

```
DECLARE PROCESS main_character ( <Parameters for this process> )
        PUBLIC
                int health;
                int strength;
                int agility;
        END
END
```

You can include that sentence right before declaring the main_character process. For that process, it'll be like if the variables had been declared in their PRIVATE scope but, as you'll now see, the rest of the game's processes can now access and modify them.

61.2 The data type associated to the process

The DECLARE sentence not only associates the new vars to the process but it also changes the process identifier so we can have access to them.

Up until now, the process identifier was stored in a variable of type integer (INT) and we could access its basic data (x, y, size, etc) as if it was of a TYPE variable.

This continues to be like that and you'll be able to use the identifier for the process like before but we'll now have another new data type that allows us to access all the PUBLIC variables for that process in the same way.

The new data type created by DECLARE is named after the process so in our case main_character will now also represent a valid data type. So now any process can hold a PRIVATE var of type ?main_character? and have access to its basic data and all its PUBLIC variables through it.

But beware: Variables of type ?main_character? and all the type vars associated to a process with PUBLIC scope vars in them cannot be modified before having a valid process identifier associated to them.

What does that mean?

That means that prior to PUBLIC variables usage, you must store a valid process identifier. For example, any enemy that wants to get the main_processes ?health? variable must first declare a PRIVATE variable of type ?main_character?, as shown below:

```
PRIVATE
        main_character objective;
```

And second, before accessing any of the fields in objective, it must refer to the actual main_character process, and check that the proess still exists. You could do:

```
        IF ( objective = collision ( TYPE main_character ) )
                objective.health = objective.health - 1;
        END
```

Once more, please note that before being able to use objective.health you must check there is a collision and therefore objective holds a valid process identifier.

And, of course, apart from collision(), you can also use get_id(), father and -in general- any other method that returns a process ID for main_character to work with the new data TYPE returned by DECLARE.