Apus Game Engine Manual

 *This is a living document. Feel free to comment or request access to edit. You can find the older Engine3 manual here.*

 *This is an explanation document, not the reference. See in-code comments and examples for more info.*

About

The engine is published under the **BSD-3** license. So you are free to use it in any projects, modify the code, take parts/functions/snippets etc (TLDR reference).

Some of released games based on this engine:

- Spectromancer (Win / iOS) - Steam page, Game site, AppStoreAstral Masters - fantasy card game
- Astral Heroes (Win) - Steam page, Game site
- Astral Masters (Win / iOS) - Steam page, Game site
- Astral Towers (Win / Linux) - Steam page

History of the Engine (article in Russian, can be Google-translated) - Link

This codebase is not just a game framework: it contains many units (which I made as much independent as I can) for different purposes - logging, debugging, GUI, file formats, networking, resource management, etc. They can be used in different applications: web sites, GUI apps, command-line tools - wherever you want.

My Patrons

- **Eli M**
- **Dave W**
- **Ken Ramsey**
- David Millington
- Michalis Kamburelis (Castle Game Engine)
- Gary Chike
- Gerry Chike
- Tristan Marlow
- David York
- add your name - support me on Patreon!

Features

Platforms:

- **Windows** - used in many games, being actively developed

Compiler support:

- **Delphi** - primary compiler (10.4 Community Edition is OK)

- **Linux** - works via libSDL2
- **iOS** - was used for some games in past, currently not being developed
- **Android** - was used for demo projects, not being developed

Graphic APIs:

- **OpenGL / GLES** - primary API
- **Direct3D(8)** - deprecated

Graphic features:

- Auto generated or custom shaders
- Particles - regular and “linear”
- Shadowmaps
- 2D / 3D transformations

Subsystems:

- UI system with extendable styling
- Scripting
- In-game console and debug tools
- Logging

Supported file formats:

- Images - TGA, PNG, JPG, BMP
- 3D - OBJ, IQM

- **Lazarus** - was used for mobile platforms, trying to keep code compatibility (FPC version 3.2 is required)

Network:

- Reliable UDP-based messaging protocol
- HTTP-based messaging protocol

Sound:

- **Windows**: support for external libraries - iMixer Pro, SDL Mixer
- **iOS, Android**: native media players

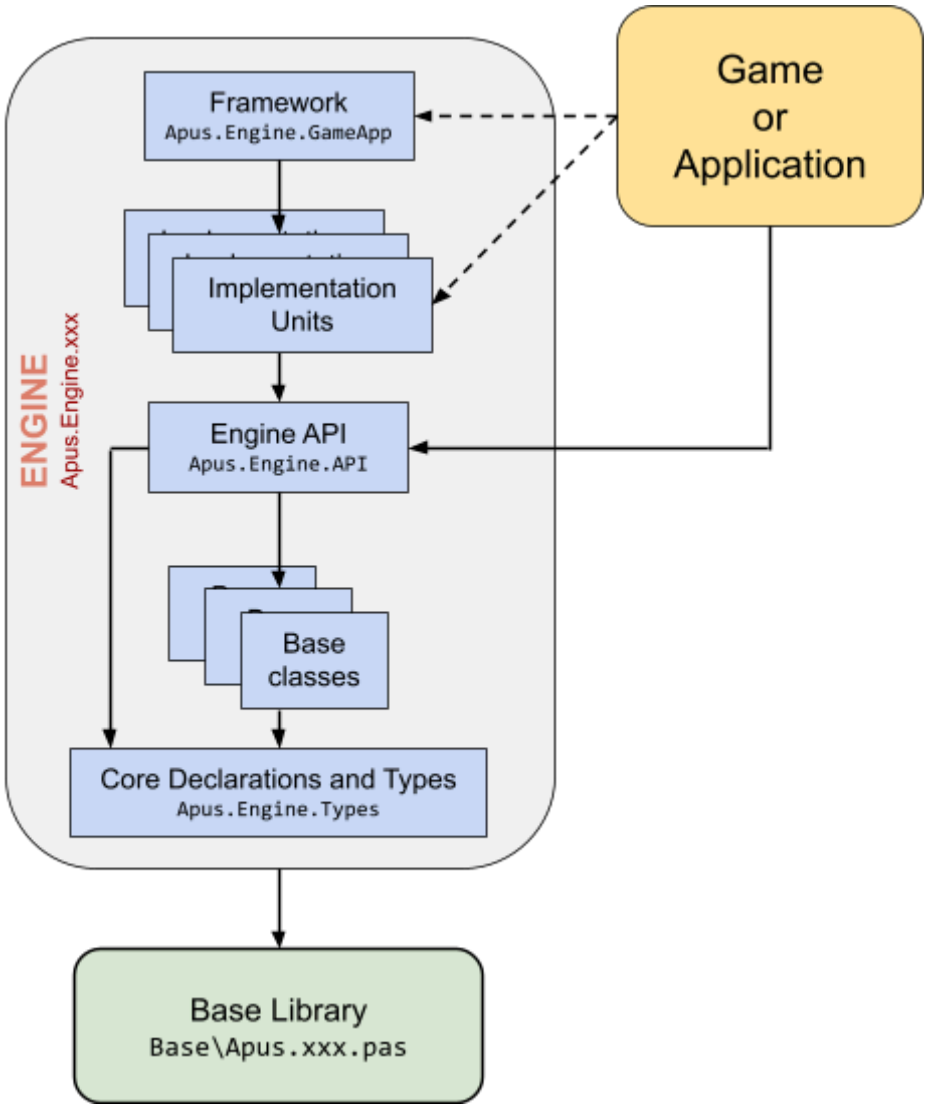
Other features:

- Vector fonts (using FreeTypeLib)
- Advanced text styling/formatting
- Transition effects
- Realtime screen blur effect
- Texture atlases

Links

- Main repository (with examples): https://github.com/Cooler2/ApusGameEngine
- Additional example projects: https://github.com/Cooler2/ApusEngineExamples
- Please support me on Patreon: https://www.patreon.com/ApusGameEngine
- Email: ivan@apus-software.com

Architecture



Step-by-Step Tutorial

Create a New Project

Although you can use the *SimpleDemo* project as a template, let's see how to make a new project from scratch.

1. Create an empty project (dpr file) with the following content:

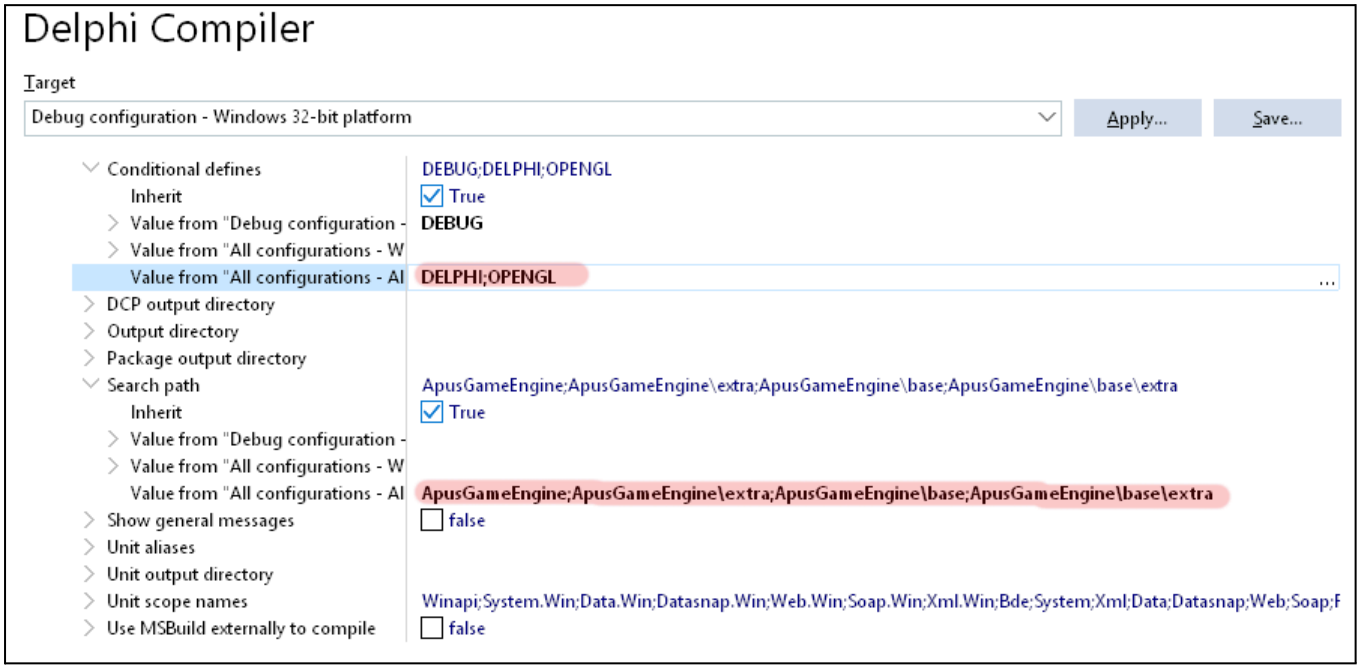
```
program Demo;
uses Apus.Engine.GameApp;
var
 app:TGameApplication;
begin
 app:=TGameApplication.Create;
 app.Prepare;
 app.Run;
```

```
  app.Free;
end.
```

2. Clone the repo into the project's folder:

```
git clone https://github.com/Cooler2/ApusGameEngine
```

3. Go to the project options and add these folders into the Search path. Add also these conditional defines:



4. Build and run. If everything is OK, you should see a black window with a running spinner.

What's Going On?

What you see is a window containing your Render Area of 1024x768 pixels. Running spinner is a predefined loader **Scene**. This scene is fullscreen i.e. it occupies the full render area so usually only one fullscreen scene can be visible. Press [Win] + [~] keys and you'll see a **Console Window** - this is a windowed scene.

Add a Scene

Basically, you need to use the **Apus.Engine.API** unit for most features.

```
// Create an object of TGameScene class or its descendants to add a scene
scene:=TGameScene.Create;
// Scenes are ordered by their zOrder
scene.zOrder:=2;
// Add scene to the game
game.AddScene(scene);
// Make it active (visible)
scene.SetStatus(ssActive);
```

Override scene's Render method to draw what you want. If you want just a green screen:

```
procedure TMyScene.Render;
Begin
 // Use gfx.target interface to access the render target
 gfx.target.Clear($FF008000); // clear screen with opaque green color
end;
```

See **Scenes** or **SimpleDemo** example projects for more details.

Drawing Primitives

Draw primitives with the **IDrawer** interface. Drawer object is declared as "draw" in the **Apus.Engine.API** unit and created by the Game object during start-up.

Simple primitives:

```
draw.FillRect(0, 0, 40, 20, $FF0000FF); // fill a rectangle with blue color
```

Color values are always in $AARRGGBB format.

Textured primitives:

```
draw.Image(0, 0, image); // draw an image with its upper-left corner at 0,0
```

With any images you can use tint color: default tint color is $FF808080 which means 1.0 scale factor for all the color components. You can use any scale factor in [0..1] range for the alpha component and any scale factor in [0..2] range for each of the color components. For example, the $4080FF00 value means: use 25% transparency, keep red values as-is, multiply green values by 2 and zero the blue channel.

Drawing Text

Use the **ITextDrawer** interface for text output. Text drawer object is declared as "txt" in the **Apus.Engine.API** unit and created by the Game object during start-up.

Example:

```
// Get handle for font named 'default' and size 10
font:=txt.GetFont('default',10);

// Draw 'Text' with black color using the font handle at position (10,50).
// This position is for BASELINE, not top or bottom
txt.Write(font, 10, 50, $FF000000, 'Text');
```

There is a built-in raster font named "Default" with sizes 6.0, 7.0 and 9.0.
Font name doesn't have to exactly match the name of a loaded font: you can load a font file with internal name "Times New" and access it with painter.GetFont('times') if there is no better match.
Write() assumes UTF8 encoding (String8 type). For 16-bit strings (WideString / UnicodeString / String16 types) use WriteW() with the same syntax.

Working With Images

Use **LoadImage**() or **LoadImageFromFile**() functions from the **Apus.Engine.API** unit to load image files into textures. LoadImageFromFile() is a lower-level function while LoadImage() is a wrapper for it.

The following file formats are supported:

- **TGA** - 24bit (x)RGB or 32-bit ARGB images
- **JPG** - 24bit (x)RGB images
- **JPG+RAW** - if there is a .RAW file along a .JPG file - it is loaded as an Alpha channel to make a 32bit ARGB image. RAW file is a 8-bit channel with RLE compression. Use the Tga2Jpg tool to convert a 32-bit TGA file to JPG+RAW pair or vice-versa.
- **DDS** - used for DXT-compressed textures (must be square PoT)
- **PVR** - used for PowerVR compressed textures (must be square PoT)
- **PNG** - always loaded as 32bit ARGB images, external LodePNG library is required. Add "LODEPNG" global symbol and proper DLL file to support PNG. (Free Pascal implementation can work without LodePNG - using fpImage library).

Example:

```
var texture:TTexture;
LoadImage(texture,'mysprite'); // load [defaultImagesDir]\mysprite.[tga/jpg/png/...]
```

You may specify file extension or omit it - in this case all supported extensions will be checked for existence.

Notes:

- If there is an image atlas loaded with a given image name - no file will be loaded, instead it returns a texture cloned from the atlas.
- It supports Preloader. Image files can be preloaded in separate threads for faster access. If the requested file was preloaded, the function returns a copy of the preloaded image.
- For Android: it first tries to load a resource from the application bundle, then - from a file.

Use IDrawer to draw texture images:

```
draw.Image(x,y, texture); // just draw image with top-left corner at x,y
draw.Image(x,y, texture, $FF808040); // draw tinted with color $FF808040
draw.RotScaled(x,y, scaleX,scaleY, angle, texture); // x,y - image center
```

Any draw operation uses tint color. Default tint color is $FF808080, which means that all R/G/B/A channels are kept as-is. The default texture blending mode is Modulate for the Alpha channel and Modulate2X for R/G/B channels.

Usually a texture consists of 2 images: one is the system RAM and one in the video RAM (VRAM). When loaded, a texture contains only the system RAM image (offline) and can't be used for GPU rendering. When you try to draw it, it is uploaded to the VRAM (made online). Also you don't have to worry about that, you can make it online (upload to the VRAM) manually - for example, if you're using custom shaders or custom rendering code instead of IDrawer:

```
resman.MakeOnline(texture); // just upload
```

or

```
shader.UseTexture(texture); // upload and make active
```

To access the texture content, use Lock() / Unlock() methods. When texture is locked, you can use the data / pitch fields to directly access its system RAM copy. When you unlock the texture, the modified area (or the whole texture) is marked as dirty so will be uploaded once the texture is made online.

Some textures may reside in the system RAM only: for example, if you use images for CPU processing or as a source for other textures.
Some textures may reside in the VRAM only - usually these are RenderTarget textures, which you use as canvas for GPU-based rendering. You can't access the content of such textures.

Input: Keyboard and Mouse

Coming soon...

Composing the Screen. Main Loop.

The whole image is composed from scenes. Consider your game as a desktop where scenes are windows on that desktop. Each scene can be *fullscreen* (assuming no scenes below can be visible) or *windowed* (partial). Each scene has its *status*:

- *ssActive* - scene is visible: **Render()** is called to draw the scene and **Process()** is called to update it
- *ssBackground* - scene is not visible, but it is alive - Process() method is called.
- *ssFrozen* - scene is not active.

Active scenes are sorted by their *zOrder*. So the screen is composed from the topmost fullscreen scene and all the windowed scenes above it.

Process() method is called with an average *frequency* which does not depend on FPS, so you can put your "ticks" handling there. Or you can add logic to the Render() method for things that have sense only when the scene is rendered.

Process() should return true if something is changed so the scene needs updating. If no scenes need updating, the game may decrease FPS.

Scene Effects

Although, you can change scene status to show/hide them, there is a better way: scene effects.
Each scene can have an effect attached to it. Scene effect is an object which overrides the standard scene rendering mechanism. Usually, it renders the scene onto an off-screen render target texture and then draws that texture with specific effect.

Built-in scene effects are defined in the **Apus.Engine.StdEffects** unit**.**
Scene effects may handle some logic: zOrder, UI layers, focused UI controls etc.
For transition from one fullscreen scene to another one use **TTransitionEffect** class.
To show/hide a windowed scene use **TShowWindowEffect** class.

Scene effects can also be used for other things than transitions. For example, the Blur effect is used to blur the scene image.

See the **Scenes** example for more info.

Adding UI Elements

Coming soon...

Scripts

There is a command processor in the **Apus.Engine.CmdProc** unit which executes string commands. You can use console window ([Win]+[~]) to play with it: each command you type there is passed to the command processor for execution. It can also run script files - by executing each line as a command.

Scripts may be useful for configuration, for example - UI layout and styling. So you can change UI or behaviour without altering the code. This is useful when the UI is designed by someone, who's not familiar with the code or can't build the executable. And this is especially useful for modding.

Script example:

```
Create UIButton MsgBox\BtnYes
x=10
y=10
width=160
height=36
```

This is equivalent to the following code:

```
button:=TUIButton.Create(160,36, 'MsgBox\BtnYes', '', 0, parent);
button.SetPos(10, 10);
```

Let's look into details to see how this works. There are some predefined operators and it's possible to declare additional operators. Each command must contain one operator, which can be a prefix, a suffix or in the middle.

The 1-st line is: 'Create UIButton MsgBox\BtnYes'. The operator here is 'CREATE' which is implemented in the **Apus.Engine.UIScript** unit. The rest of the string is passed to the operator handler procedure. It creates a new UI item of type TUIButton named 'MsgBox\BtnYes' (if such an element already exists it's not re-created) and sets it as the current object context. Current object context is something you face when you use the "with" delphi keyword: so you don't have to write the object name to access it's properties.

Next line is 'x=10' - this is the assignment operator ('=') which is a mid-string operator. It uses the evaluation mechanism implemented in the **Apus.Publics** unit to evaluate the right part of the sentence (10) and assign it to the variable in the left part. Since the UI button is now the default object context, it assigns 10 to the X property of the button which maps to the TUIButton.position.x field.

Left part may contain arithmetical operations, mathematical functions - some of them are predefined, other you can add manually. String values are not quoted:

```
caption=Hello world!
```

You can declare constants and variables:

```
const centerX=renderWidth / 2 // constant works as string replacement
int middleX=renderWidth / 2 // new Integer variable initialized with evaluated value

x=centerX // centerX is replaced with renderWidth/2 and then evaluated
x=middleX // assignment only, value is already stored
```

You can use conditional execution:

```
IF expression // 0 or '' = false, everything else = true
 ...
ELSE
 ...
ENDIF
```

or conditional function:

```
x=IF(condition, value1, value2)
```

You can also run other script files:

```
run Scripts\MyScript.txt
```

Particles. Line Particles

Coming soon...

Using Tweaker

Coming soon...

Fonts. Advanced Text Features

Coming soon...

Game Objects

Coming soon...

Off-Screen Rendering

Coming soon...

Shaders

Coming soon...

3D

Coming soon...

Networking

Coming soon...

Base Library Explained

Apus.Common

This is a big collection of different functions and types (like SysUtils). I'll describe just some groups of them. Others are evident or described in the code.

String types

For strings I use the approach adopted from older Delphi and FPC without "DelphiUnicode" mode - there are 2 strings types used: 8-bit and 16-bit.

- String8 - such a string is an array of bytes, encoding is UTF8. It's efficient and compatible with C PChar (not DelphiUnicode PChar!), so can be used with external libraries. It's good for almost everything.
- String16 - this string is an array of words. Encoding can be UCS2 or UTF16. Such strings occupy more memory but can be indexed by characters instead of bytes (if you consider it UCS2, not UTF16).

Generally, you should not directly assign strings of different types to each other as implicit conversion may work or not work. Or work badly. Depending on the used compiler and platform.
Proper string conversion is:

```
str8:=EncodeUTF8(str16);
str16:=DecodeUTF8(str8);
```

Or use typecasting functions as they support other default string types:

```
st8:=Str8(any_string); // make UTF8 string from any string
st16:=Str16(any_string); // make 16-bit string from any string
```

Exceptions

EWarning, EError, EFatalError - these are general purpose exceptions that capture caller address (and call stack if possible). Use ExceptionMsg() to show this info:

```
try
 raise EError.Create('Test');
except
 on e:exception do writeln('Error: '+ExceptionMsg(e));
end;
```

Threads and Critical Sections

Each thread has its unique ThreadID, but it's more convenient to have a meaningful name. You can do this with **RegisterThread**('ThreadName'); Call **PingThread** periodically from each registered thread and you'll get reported about any stalled threads.

Use **TMyCriticalSection** for a named critical section. It contains additional info useful for deadlocks debugging. If a deadlock occur, you'll get a report like this:

```
Section A: owned by thread XXX (from [address1]), requested by thread YYY (from [address2])
Section B: owned by thread YYY (from [address3]), requested by thread XXX (from [address4])
```

This is comprehensive information to figure out the deadlock reason.

Even more, you can assign each critical section a **level** to ensure there are no circular dependencies, so deadlocks are impossible:

- Follow the rule: section A can be entered within section B only if A **has higher level** than B. This means your sections are organized as a tree, not graph. In this case deadlocks are impossible.
- Set **debugCriticalSections = true** to enforce runtime verification of that rule. You'll get an exception whenever the rule is broken.
- This is recommended only for debugging, because 1) it slows down the code and 2) you'll get an exception when a deadlock **is possible**, but not guaranteed. And it's still impossible to 100% guarantee that you've tested all the possible cases. Even if you tested all the code paths.

Apus.EventMan

This is the Event Bus system. It is used for callbacks, event handling, soft-linking different parts of the project and inter-thread communication. With EventMan you can emit signals in unit A and handle them in unit B while neither unit uses another one, but both use EventMan.

How to signal an event:

```
Signal('EventName', tag);
```

where EventName is a string in the form *'Part1\Part2\...\PartN'* and tag is an optional additional integer attribute.

To handle events set event handlers like this:

```
procedure EventHandler(event:EventStr; tag:TTag);
begin
 // handle event here
end;
...
SetEventHandler('EventName', EventHandler, mode);
```

When event 'A\B\C' is signalled, the system first tries to run handlers registered for 'A\B\C', then more generic 'A\B' handlers and finally the most generic 'A' handlers.

Event signalling is "blind" - you don't know if it will be handled or how many handlers will react to it. And there is no return value.

There are 3 different types of event handlers:

- **emInstant** - event handlers are called instantly inside Signal(), so processed in the same thread before returning from the Signal().
- **emQueued** - events are queued for the thread where the event handler is registered. The thread should periodically call HandleSignals() to handle queued events.
- **emMixed** - this works like Instant if an event is signalled from the same thread, otherwise it is processed like Queued.

Delayed events:

```
DelayedSignal('event', delay, tag); // event will be handled in delay ms (Queued mode only)
```

You can pass additional data with events:

- Directly in the Tag field.
- Using Tag as a pointer to external data (Tag size is guaranteed to be wide enough to contain a pointer value).
- By adding to the event name: 'XXX\YYY\additionalData'. But keep in mind that event names are quite limited in size.

You can link events, so if 'AAA' is linked to 'BBB' then Signal('AAA') will also Signal('BBB'):

```
Link('AAA','BBB');
```

Since there is no way to specify a new tag in the link, you can use "::tag_value" suffix:

```
Link('AAA','BBB::1234'); // equivalent to Signal('BBB',1234) when AAA happens
```

Tag suffix works with Signal() too:

```
Signal('MyEvent::$1234'); // equivalent to Signal('MyEvent',$1234)
```

Apus.Structs

This is a collection of some useful data structures.

THash

THash is a hashed list of rows:

```
Key1 -> value1.1, value1.2, ... value 1.n
Key2 -> value2.1
Key3 -> value3.1, value 3.2
```

Where keys are AnsiStrings and values are variants.
Use case:

```
var
 hash:THash;
...
 hash.Init(true); // true - allow multiple values for each row, otherwise only one value is allowed
 hash.Put('key', 123); // add value for the key
 hash.Put('key', 234); // add another value for the same key
 hash.Get('key'); // returns 123
 hash.GetAll('key"); // returns [123, 234]
```

THash is thread-safe except THash.Init, which is not.

TSimpleHash / TSimpleHashS / TSimpleHashAS

It's a simple hashed list where each key has only one associated value.
There are few hash versions:

- TSimpleHash: both keys and values are int64.
- TSimpleHashS: String -> int64
- TSimpleHashAS: String8 -> int64

TObjectHash

It's an open-address hash implementation which stores entries of TNamedObject type. The keys are object names and values are objects themself. This hash is case-insensitive (internally use FastHash which is also case-insensitive).

Apus.AnimatedValues

TAnimatedValue is a container for a floating point value that can smoothly change in time. It is widely used for transitions and storing values which are subjects for transitions. It is thread-safe.

Commander code calls **Animate**(), **AnimateIf**() to make it change.
Consuming code calls **Value**, **IntValue** to get it's current value.

Animations can overlap and there are the following rules:

- Final value will match the value of the last animation command, regardless of the duration and final time moments.
- Change is smooth regardless of overlapping.

Apus.Database

Currently supports MySQL database engine using libMySQL.dll
Usage example:

```
 // Configure
 DB_HOST := 'db_host';
 DB_DATABASE := 'my_base_name';
 DB_LOGIN := 'username';
 DB_PASSWORD := 'password';
 // Init connection
 db:=TMySqlDatabase.Create;
 db.Connect;
...
 // Done
 db.Disconnect;
 Free(db);
```

Run queries:

```
 db.Query('INSERT INTO xxx values(1,2,3)'); // simple query
 db.Query('SELECT name FROM users WHERE id=%d', [userID]); // Format() syntax
```

```
 db.Query('SELECT email FROM users WHERE name="%s"', [name]); // string values are sanitized automatically
```

Access the results:
First method - use return value. Query returns all the data as an array of strings:

```
list:=db.Query('SELECT id, name FROM users'); // list is array of strings with rowCount * colCount elements
for i:=0 to db.rowCount-1 do
 writeln(Format('id: %d, name: %s', [list[i*2], list[i*2+1]]));
```

Another method: use Next, NextInt, NextDate to fetch fields, NextRow to go to the next row.

```
db.Query('SELECT id, name FROM users');
If db.rowCount>0 then
 for i:=1 to db.rowCount do begin
  writeln(Format('id^ %d, name: %s', [db.nextInt, db.Next]); // fetch int and string
  db.NextRow; // go to the next row
 end;
```

Apus.Logging

There are 2 logging mechanisms: Basic logging (use Apus.MyServis) and Advanced logging (use Apus.Logging).

Basic logging

Basic logging is good enough for most client-side apps. It uses a single log file, no log rotation.
Initialization:

```
// Case 1: use 'game.log' file, flush it with every log message (can be VERY slow)
UseLogFile('game.log');

// Case 2: use 'game.log' file, keep it opened (faster, but you may not see the latest content)
UseLogFile('game.log', true);

// Case 3: enable log caching
UseLogFile('game.log');
LogCacheMode(true);

// Case 4: Launch a separate thread for log flushing
UseLogFile('game.log');
LogCacheMode(true, false, true);
```

Usage:

```
LogMessage('Regular message'); // regular message - can be cached
LogMessage('Time is %d',[GetTickCount]); // Format() syntax
ForceLogMessage('Alarm!'); // prioritized message - always flushed to file, never cached
```

Advanced logging

Advanced logging is designed for server-side applications.
It features:

- Daily logs rotation
- Different priority/importance level of log messages
- In-memory log storage for better performance: you can set it up to keep low-level messages only in RAM and not save on disk. So you can save time and disk space. However, you need some kind of management service to access these messages.

Usage:

```
InitLogging(100, 'Logs', logInfo); // use 100 MB for in-memory log, use folder 'Logs' for log files, store log messages with level logInfo and above
LogMsg('Test: x=%d, y=%d', [x,y], logInfo); // emit an info-level log message with formatting
```

This system doesn't spawn the log flushing thread, you should flush logs yourself by periodically calling **FlushLogs**.

Combining Basic and Advanced logging

Sometimes you write code that is used on both client and server sides. Which logging should you use? Actually it doesn't matter - you can set it up so Basic log messages will be forwarded to the Advanced log and Advanced log messages will be stored in the Basic log. Just decide what do you want.

Apus.Publics

This unit is used to:

- Publish typed variables, constants and functions so they can be found by name. So you can make your variables available from scripts.
- Evaluate expressions with published variables, constants and functions. This is used to execute scripts.
- Provide sets of global variables for run-time tweaking. This is used by the Tweaker scene to visually adjust any parameters.

Apus.SCGI

This is a SCGI server framework featuring templates translation engine. Its detailed description is quite large, so I'd suggest to read this article (it's in Russian, but Google translates it pretty good):

https://habr.com/ru/post/491272/

Engine Explained

Game Object. Main Loop.

Coming soon…

Screen Modes. Coordinate Systems.

Coming soon…

Images Preloader

We usually want to decrease game launch time. The best case is when the game can be used immediately without loading delay. One way to achieve this is Images Preloader: this system launches some threads at the very beginning and preloads key images during game initialization. Later the **LoadImageFromFile()** uses a preloaded image (if available) so it works much faster - it neither accesses the disk nor unpack the image.

Images Preloader is implemented in the **Apus.Engine.ImgLoadQueue** unit. You should call **StartLoadingThreads()** as early as you can in your game to launch the preloaders threads. Then call **QueueFileLoad()** for all the key images you want to be preloaded. That's all! :-)

You can inspect the log file to optimize the preloading and see if there are any bottlenecks. Keep in mind that the first launch (when game files are not cached) is very different from subsequent launches, so you may want to invalidate the filesystem cache before a test launch.

UI System

UI elements (controls, widgets) are objects of the TUIElement class and its descendants. These classes are declared in the **Apus.Engine.UIClasses** unit.

Let's start with differences between the Engine UI system and VCL (and similar UI libraries):

- Any element can be nested into any other element.
- All coordinates are floating point values.
- The pivot point (the point you use to specify elements position) doesn't have to be the top-left corner: you can use any point and use different points for different elements.
- Any element can be scaled
- UI rendering is separated from UI behavior and layout. You can use different rendering procedures (UI styles) for different elements.

The UI units are:

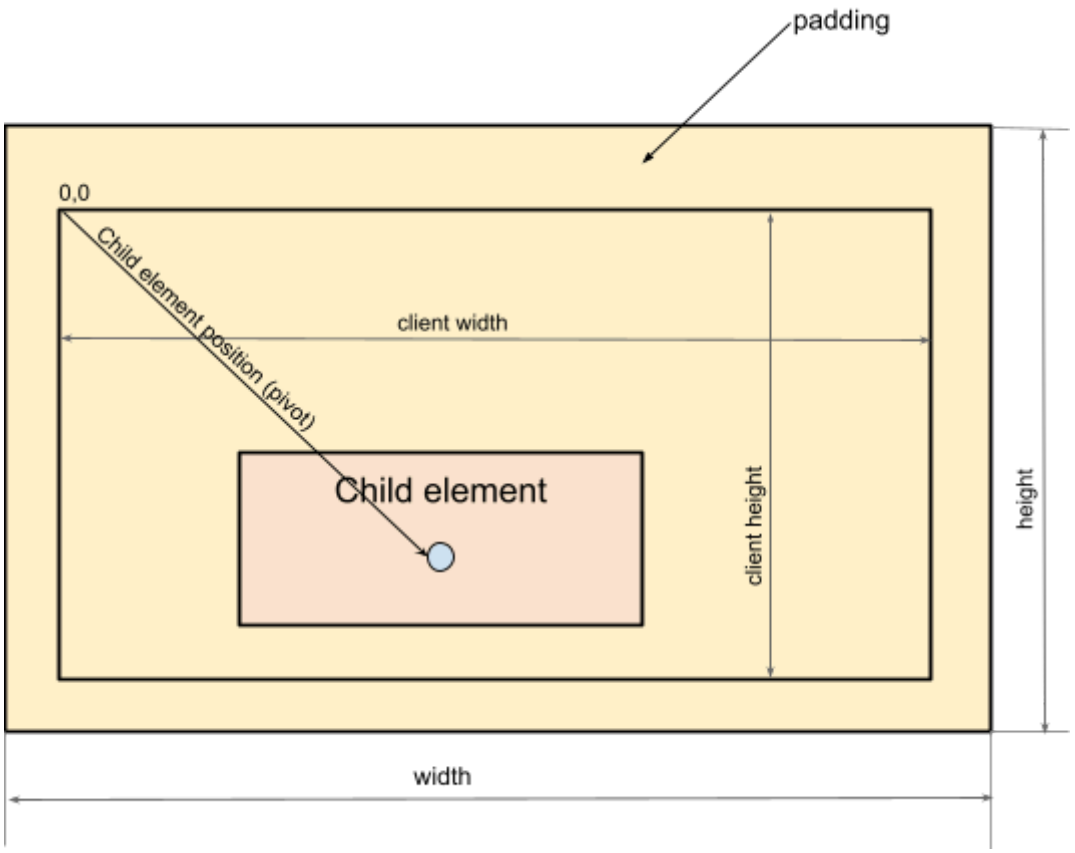
- **Apus.Engine.UIClasses** - declaration of the UI controls, their layout and behavior
- **Apus.Engine.UIRender** - the UI rendering system which implements the default UI style and supports additional (pluggable) UI styles

- **Apus.Engine.UIScene** - “glues” everything together: it implements a scene class with a fullscreen root UI element, transfers user input from the game object to the UI elements and displays its UI.
- **Apus.Engine.UIScript** - provides access to the UI elements and their properties from scripts (via the command processor). This allows you to create and/or configure the UI at runtime. Can also be useful for creating mods.

UI elements are organized as a tree or forest: there are 1 or more root elements and children elements are located within parent elements.

Main properties:

- **position** - 2d vector specifying position of the elements pivot point relative to its parent elements client area
- **size** - 2d vector specifying elements dimension (width, height) in parents element coordinates
- **pivot** - 2d vector specifying relative position of the elements pivot point (hot point). (0,0) means top-left corner, (0.5, 0.5) - center.
- **scale** - 2d vector specifying child elements coordinate system transformation
- **paddingXXX** - width of the padding area (borders) which is deducted from (width, height) to make (clientWidth, clientHeight) area for children elements
- **scroll** - 2d vector which is subtracted from children position to make scrolling. This usually means that children elements should be clipped by the client area rect
- **anchorXXX** - controls how the element reacts on the parent element resize. Each value controls one side of the element. For example, anchorLeft impacts only on the left side of the element. 0.0 means no impact, 1.0 means 100% impact, i.e. if the parent element width increases by 10, then the left side of the element moves right by 10. Can be outside the 0.0-1.0 range to achieve fancy behavior.
- **order** - controls z-order of elements and their overlapping. Important for the root elements, usually the topmost root element receives user input.



- **enabled** - if the element is not enabled - it doesn't react on the user input and usually displayed as grayed out. Child elements inherit this (although their "enabled" state doesn't change).
- **visible** - if the element is not visible - it's not displayed at all. Its child elements aren't displayed too.
- **shape** - controls if the element can receive mouse/tap events or these events should be passed to the underlying elements. An element can be *empty* (no interaction), *opaque* or *shaped* - use a shape object to define which part should react on input (used for non-rectangular elements like shaped windows or buttons).
- **style** - numerical ID of an UI style that draws this element. 0 means the default style implemented in the **UIRender** unit.
- **styleInfo** - string property with additional info for the UI style (implementation dependent). For example, it can contain color values or texture image which should be used to draw the element
- **parentClip** - should element be clipped by its parent client rect area (default - yes). Disable this for child elements if you want to place them outside the client area (this also disables scrolling of this element!)
- **clipChildren** - should children elements be clipped by the elements client area rect (default - yes). You can disable this if you don't need scrolling for better performance.
- **layout** - here you can assign a specific Layouter object to control how child elements should be positioned, or how the element should behave.

Main UI Widgets:

These widgets are declared and implemented in the **Apus.Engine.UIClasses** unit:

- **TUIElement** - this is the base UI element. It is intended to make a UI structure i.e. to be a container or placeholder for other elements. However, it can also be used to display something or receive user input.
- **TUIImage** - this element is very similar to the base element, but it is intended to display images: it can be configured to display an image from a file, or use a custom draw procedure.
- **TUILabel** - displays a text label.
- **TUIButton** - a clickable button. There are 3 button modes: bsNormal - a regular push button, bsSwitch - toggle button (SpeedButton), bsCheckbox - a check box. Look at the NinePatch demo to see how to create a speed buttons group.
- **TUIEditBox** - a single line edit box. (example)
- **TUIScrollBar** - a scroll bar. Any UI element can link 2 scroll bars to scroll its content.
- **TUIListBox** - displays a list of strings. Each line can have its own hint.
- **TUIComboBox** - complex widget - a special kind of button which shows a pop-up listbox when clicked.

UI Layouters:

It is possible to attach a Layouter object to any UI element. Layouter (TLayouter class) is an object that layouts children elements, it is executed before the UI is rendered.

TRowLayout - is a layouter which aligns child elements in a row (or column) with equal space between elements. Have a look at the NinePatch demo to see how to use a Row Layouter to layout a group of buttons.

UI Styles

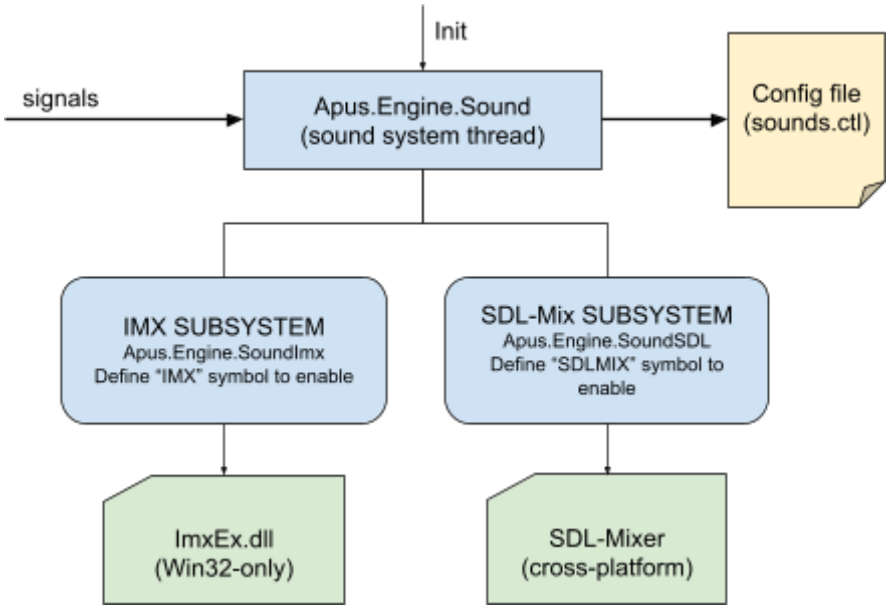
Different games may need very different approaches for the UI drawing, that's why there is no universal UI rendering. Instead, the Engine supports pluggable UI rendering styles. There is a default UI style (style=0) which is convenient for prototyping. Sometimes it can be also used for the production stage, but if you want a more sophisticated UI look and feel - you can use other built-in styles or create your own. Don't worry, it doesn't mean that you should write drawing routines for all the UI controls: you can arbitrarily mix UI styles and implement your own style just for the elements you need, for example - menu buttons.

There are 2 additional styles in units:

- **CustomStyle** - this UI style is designed to decorate UI elements with images, it is highly oriented on customizing the UI via scripts, not the code.
- **BitmapStyle** - this UI style is based on the concept of building bitmap images "on fly" and using them for the UI decoration.

Sound System

Here is the sound system structure:



How it works: you call `InitSoundSystem()` to start the sound system, which works in a separate thread. If you're using the GameApp unit (framework) - it calls the initialization for you. Then it loads the config file (default: sounds.ctl) from the current folder. This file defines all sounds and music. If there is no config file - the sound system won't run and you don't need underlying libraries.

When the sound system is running, you use it by sending signals like this:

```
Signal('Sound\Play\Click'); // play sound event "Click" defined in config file
Signal('Sound\PlayMusic\track1'); // play music named "track1"
```

Keep in mind that you must define global symbols to enable and link specific sound subsystems in your project:

- "IMX" - for IMixerPro (ImxEx.dll)
- SDLMIX - for SDL-Mixer

Config file syntax and structure

```
$Section Settings
  PreloadSamples        "sample.ogg,sample2.ogg" ; preload these files
  PreloadSamples        ALL  ; preload all sample files referenced in any sound event
  PreloadSamples        None ; don't preload samples, each file will be loaded when first use
$EndOfSection

; Define music tracks. They're usually streamed (unpacked while playing)
$Section Music
  $Section Track1  ; Play this music with "Sound\PlayMusic\Track1"
    File     "music.ogg"
    volume   50
    loop     ON
  $EndOfSection
$EndOfSection
```

```
; Define sound events.
$Section SoundEvents
  ; Format: EventName  "filename,vol=xx,pan=yy"
  Sample    "sample.ogg"             ; Play this with "Sound\Play\Sample"
  sampleQuiet "sample.ogg,vol=30"    ; This event reference the same sound file, so it won't load twice
  sampleLeft  "sample.ogg,pan=-90"   ; You can also pass sound properties via signal's tag
$EndOfSection
```

Redistributable binaries

- IMixerPro: distribute `ImxEx.dll` file with your executable.
- SDL-Mixer: distribute these libraries:
    - SDL2-Mixer / libSDL2_mixer-2.0
    - sdl2 / libsdl2
    - libogg-0 - for OGG support, optional
    - libvorbis-0 - for OGG support, optional
    - libvorbisfile-0 - for OGG support, optional
    - libmpg123 - for MP3 support, optional
    - libmodplug-1 - for tracker music support, optional

Events reference

Please note that different subsystems support different sets of features, so some commands may work or not work depending on the used subsystem.

- **`Sound\Play\[name]`** - play sound named [name] with settings defined in the config file
  Additional options can be passed in TAG:
    - low byte = volume multiplier in % (0..255, 100 = 1.0)
    - 8..23 bits - alternate sample rate (if >250) or alternate playback speed (1..250%)
    - 24..31 bits - panning (-100..100)
- **`Sound\PlayMusic\[name]`** - play music. Previous music will stop (with optional [cross]fading)
- **`Sound\PlayMusic\None`** - stop current music.
- **`Sound\SetVolume\Sound`** - change master sounds volume (0..100)
- **`Sound\SetVolume\Music`** - change master music volume (0..100)
- **`Sound\Pause`** - pause music playback
- **`Sound\Resume`** - resume music playback
- **`Sound\ClearCache`** - discard all previously loaded sample files (useful when sound files are updated on disk)

Reference

Events

`ENGINE\*` - commands and interaction between engine parts (Game and Platform objects).
`UI\*` - events related to UI elements.
`MOUSE\*` - mouse and touchscreen input events.
`KBD\*` - keyboard input events.

JOYSTICK* - joystick/gamepad/wheel etc.

Engine events

UI events

Mouse events

MOUSE\GLOBALMOVE (LoWord = X, HiWord = Y) - mouse moved in global (screen) space. Used by Platform to notify the Game about mouse input.
MOUSE\MOVE (LoWord = X, HiWord= Y) - mouse moved in game space.
MOUSE\BtnDown ([1..n] button number) - mouse button pressed.
MOUSE\BtnUp ([1..n] button number) - mouse button released.
MOUSE\SCROLL (delta) - mouse wheel turned

Keyboard events

KBD\KeyDown (Virtual code [0..15], scancode [16..23]) - key pressed.
KBD\KeyUp (Virtual code [0..15], scancode [16..23]) - key released.
KBD\Char (ASCII code [0..7], unicode [8..23]) - char entered.

Joystick events

Updating & Migration

Main changes from Engine3 to Engine4

The last Engine3 revision is 40912385f6115c69401aef2efcf3008826bbf36f (master branch).

Units renamed

Now units use namespaces. Base units are Apus.XXX and engine's units are Apus.Engine.XXX
Many units were renamed:

Old unit name	New unit name
EngineAPI	Apus.Engine.API
StdEffects	Apus.Engine.SceneEffects
EngineTools	Apus.Engine.ImageTools, Apus.Engine.Tools
BasicGame	Apus.Engine.Game
BasicPainter	Apus.Engine.Painter2D

Now the most useful unit is Apus.Engine.API. It contains shortcuts to the most used functions and types, so you don't have to include some units. Such a structure is designed to require less dependencies.

Identifiers renamed

Old Identifier	New Identifier
TUIControl TUIControl.transpMode	TUIElement TUIElement.shape

Migration tool

There is a migration tool which performs renaming automatically. It's just a "bulk string replace" tool, so it's not 100% acurate. However, it does ~99% of the rough work.