

Introducing Virtools SDK

The rest of the SDK documentation assumes that the principles involved with Virtools usage (Behaviors, Parameters, Behavioral objects, scenes, scripts, etc..) and the basic architecture of the Virtools SDK are known and understood. If it's not your case, take the time to read this chapter...

Presentation

The Virtools SDK (Software Development Kit) is a set of development tools (libraries, DLLs, header files) that give programmers access to all the functionalities used in Virtools applications, enabling them to write software components that directly use these functionalities, like:

- custom applications using the Virtools engines as an underlying technology
- extensions to the Virtools engines, like :
 - Behavior Plugins,
 - Media Plugins,
 - Managers Plugins,
 - Extensions Plugins (specific Parameter Types),
 - Render Engines Plugins,
 - Rasterizers.

This current part of the documentation gives an overview of the internal principles and the inner workings of the Virtools engines.

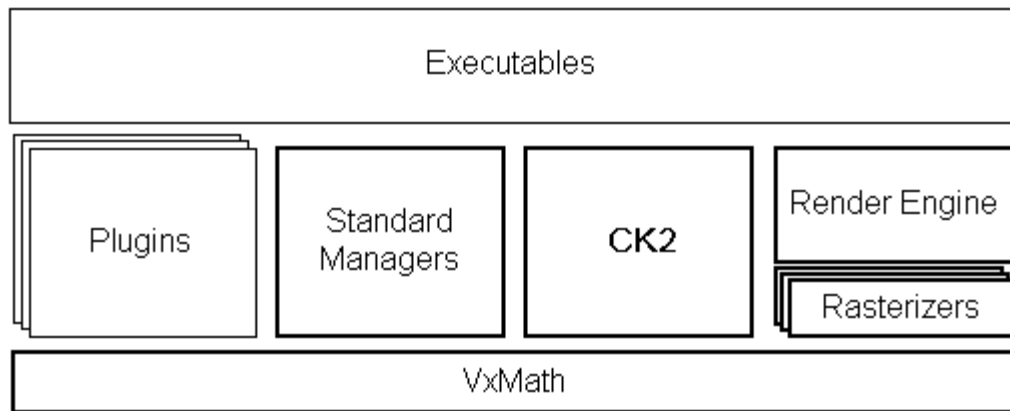
Plan of Part 1 - Virtools SDK Architecture and Principles

This overview contains the following information :

- 1. [Virtools API Architecture](#)
- 2. [Presentation of the Process Loop](#)
- 3. [Presentation of the Behavioral Engine](#) :
 - 3.1. [What are Behaviors ?](#)
 - 3.2. [What are Parameters ?](#)
 - 3.3. [What are Behavioral Objects ?](#)
 - 3.4. [Level Organization](#)
- 4. [Presentation of the Render Engine](#)
 - 4.1. [Changing the Render Engine](#)
 - 4.2. [About the Rendering Loop](#)
 - 4.3. [Presentation of the Rasterizers](#)
- 5. [Virtools Plugins Overview](#)
- 6. [Main Virtools Events](#)

Virtools API Architecture

The Virtools software components are organized in a very modular architecture. Here is the big scheme:



All the Virtools executables (Player, Creation, Dev, Web Plugin, etc..) are implemented on top of this common software architecture, made of dynamically linked libraries (DLLs), and which implement all the common functionalities of Virtools.

The central software component is made of two DLLs: CK2 and VxMath, with VxMath providing the low level functionalities, CK2 providing the core behavioral engine, and the glue for managing all the other software components.

All the other functionalities of the Virtools engine are implemented through Plugins. The Virtools software suite includes a number of standard plugins, and software developer can write their own Plugins using the SDK. See the [Virtools Plugins Overview](#) for more details about writing your own Plugins.

Also, using the SDK, developers can write their own custom executable using the Virtools core engine as a provider of 3D real-time behavioral capabilities. An example of a Standalone Player is provided in the SDK. The Virtools library still takes into account potential Plugins. See the [Creating a Standalone Application](#) chapter for more information.

Virtools APIs

Basically, the SDK consists in the API (header files .h) containing definitions of C++ functions, constants and classes, and stub libraries. Using the header files and linking against the libraries, software programmers can write software components in the form of DLLs or of executables, that can interact with the existing Virtools software components. See the [Using Virtools SDK](#) for details on how to use the APIs, and for details on development environments.

The APIs can be split into functional parts:

- VxMath APIs
- the CK2 APIs
- Standard External Managers APIs
- RenderEngine APIs
- Rasterizer APIs

VxMath API

VxMath is a DLL that deals with low level functions, types, and utilities, like vectors, matrices, mathematical functions, etc... Most of the Virtools software components use the objects of VxMath. See [The VxMath Library](#) chapter for more details.

CK2 API

CK2 is the main DLL of the Virtools engine and it corresponds to the core technology that Virtools brings to the world. The CK2 library is made of about 100 C++ classes, accessible through the CK2 API.

Basically, CK2 is centered around the behavioral engine, which deals with behaviors, behavioral objects, narrative elements like levels and scenes. All the elements that do not deal with spatial representation are implemented in this library (the elements dealing with geometry, space and visualization are implemented in the RenderEngine). For detailed information, see the [Presentation of the Behavioral Engine](#) chapter.

CK2 also implements internal system-wide functionalities in the form of internal Managers, like the TimeManager, the PluginManager, etc.. See the [Using the Existing Managers](#) chapter for more details.

Standard External Managers API

Managers implement system-wide functionalities. Some are internal and are considered to be part of CK2 (see above), and some are external to CK2, but are still accessible through an API. These Managers can be removed from the suite, or replaced by a custom implementation. Some external Managers are provided in source code in the SDK.

The standard external managers are the InputManager, the SoundManager, the CollisionManager, the GridManager and the FloorManager. See the [Using Managers](#) chapter for more information.

The RenderEngine API

The Render Engine contains all the 3D rendering functionalities, and provides an API to address rendering, like management of meshes, of the render context, of the textures, etc.. This API can be used in two ways. First, software component can address this API to control the rendering process. This is the case for behavioral objects for example. Also, the RenderEngine is itself a Plugin and can be replaced by a custom RenderEngine, in which case the API corresponds to the functionalities that the custom engine should implement.

See the [Presentation of the Render Engine](#) chapter for more information.

The Rasterizer API

The Rasterizer provides the functionalities corresponding to final drawing of the 3D scene : polygons, textures and rendering options. The Rasterizer API is mainly used by the RenderEngine. It is also the API that a custom Rasterizer must implement. See the [Presentation of the Rasterizers](#) chapter for details.

The standard Virtools rasterizers are provided in source code as part of the SDK. See the [Available Source Code](#) chapter for more information.

Presentation of the Process Loop

Real-time

Virtools is a real-time engine: it allows any behavior to react constantly and consistently to its environment, including the user/player. Each Virtools context (corresponding to one Virtools composition) runs in a single system process and thread, which means that it does its prioritizing and scheduling of tasks itself : it does not leave to the operating system the decision of what amount of CPU to assign to each algorithmic element. Real real-time operating systems do not exist and this way the Virtools engine precisely controls its own time.

When a Virtools composition (a context) is played, the Virtools core engine indefinitely loops in what is called the Process Loop. Each cycle of the loop is a succession of phases, which always happen in the same order. All that should happen happens in this Process Loop. The duration of one cycle is called one "**Frame**". The Process Loop loops until the content pauses or is reset.

This scheme, very classical in real time engines, means that all processing should be time-split. Indeed, the frame is the minimal delay between an action of the user and a reaction of the system, and the framerate is the measure of real-time systems. So each frame should last as little as possible : processes (like behaviors) that take a long time to accomplish should be designed to happen in an iterative manner, that is bit by bit, with every bit taking as little time to process as possible, so as not to slow down the other processes and not to lengthen the frame duration. Only one badly written behavior can slow down the whole system ! The actual time (in milliseconds) is accessible to behaviors through the TimeManager API, for those behaviors needing to index their actions on time rather than on frames.

Standard Process Loop

The Virtools core engine defines the process loop in a standard way. Note that if you write your own custom executable, you can change the process loop, and insert your own processing in the Process Loop.

Here are the phases of each cycle of the Standard Process Loop:



PreProcess and PostProcess

The PreProcess and PostProcess phases enable Managers to performs actions before or after the two main phases (Behavioral and Rendering). Note that if you write a Manager, you should remember the time-split constraints and write your pre- and post-processing algorithms in an iterative manner.

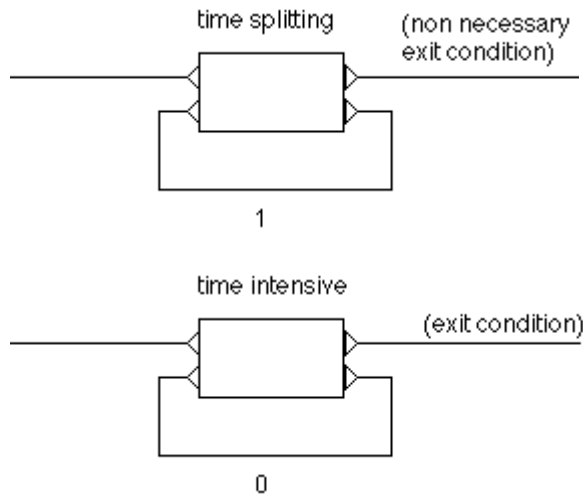
Behavioral Process

This is where behaviors are processed. In Virtools, almost every type of object can have behaviors attached to it.

During the behavioral process, all the active behaviors are executed, one after the other, using a complex prioritizing scheme : first the priority of the objects are considered, then the ones of the scripts attached to the objects, then the Building Blocks and Graphs inside the scripts. When two objects have the same priority, the order in which they are executed is unknown (it is the same if two scripts inside an object have the same priority, or two behaviors inside a script).

Each behavior, when executed, may activate other behaviors through behavior links. These links have a delay corresponding to the number of frames after which the newly activated behavior will really become active. This delay can be 0, in which case, the newly activated behaviors will be processed in the same Process Loop cycle (**frame**) as the one that activated it (at a moment corresponding to its priority).

Behavior links can point from a behavior to itself. In such a case, having a looped link delay of 1 is way of implementing the time-splitting that we talked about above, with each bit of processing taking place at each Process Loop cycle. Looped link delays of 0 are a way of implementing time-intensive algorithms. But beware of having an exit condition for these algorithms, so as not to introduce infinite loops that would halt the whole system.



See the [Presentation of the Behavioral Engine](#) for more details on behaviors and their execution.

Rendering

3d Objects (Characters, scene geometry) and 2d Objects (cursors, sprites, interface elements) are drawn during the rendering phase, by the current Render Context. The Render Context keeps a list of objects to be drawn. Objects can be added or removed from the Render Context during the Behavioral Process phase.

Note that the Rendering phase also allows for many pre- and post-processing, for highly customized rendering. See the [Presentation of the Render Engine](#) and [Custom Rendering](#) chapters for more details.

Presentation of the Behavioral Engine

General Presentation

The behavioral engine is the central core of the CK2 library, which means that it is the central part of Virtools. It manages and processes Behaviors, Behavioral Objects, Parameters, Attributes, Inputs, Outputs, etc.. The behavioral engine does not correspond to one class or function per se, but rather is a set of interrelated functions and classes that collaborate.

The management involves organizing the data corresponding to these object is memory, and providing read and write access to these objects, while keeping the overall consistency of the interrelationships between these objects.

The processing is started when a context is played, and stops when the context stops or pauses. It is called the Behavioral Process and is part of the main Process Loop (See the [Presentation of the Process Loop](#) chapter for more details). The Behavioral Process is the moment when all the behaviors are executed by the behavioral engine.

To be precise, here are the details of the Behavioral Process :

The first time a composition is played every behavior of every BeObject is reset and deactivated. BeObjects and their scripts that should be activated when a scene starts are activated and the input of these scripts is activated.

- For each active BeObject (sorted according to their priority)
 - For each of the active Scripts of this BeObject (also sorted according to their priority)
 - Execute the Script, which will in its turn execute its sub-behaviors (also sorted according to their priority).
 - Each behavior keeps up a list of active behaviors to execute this frame. The execution of a behavior can activate a new behavior to be executed in the same frame (Link delay of 0) which is then pushed on the stack of active behaviors. When the stack of behaviors to execute this frame is empty we go to the next active behavior.
 - If a Script does not have any active sub-behavior, the Script is deactivated.
 - If all the Scripts of the BeObject are deactivated, the BeObject is deactivated.

For more information check the [What are Behavioral Objects ?](#) and [What are Behaviors ?](#) chapter.

The behavioral engine and the SDK

The Behavioral Engine is accessed through the [CKContext](#)^S class. This class, among other things, gives access to Playing, Pausing, Resting, Processing of the engine. It also gives access to all the Objects (BeObjects, Parameters, etc..) that are in the engine.

You rarely have to access the Behavior Engine directly, except when you write your own executable, in which case you may want to create and Remove Objects, manage the Process Loop precisely, etc..

More Informations

For more information, you can check out the other parts of this chapter ::

- [What are Behaviors ?](#)
- [What are Parameters ?](#)
- [What are Behavioral Objects ?](#)
- [Level Organization](#)

What are Behaviors ?

General concept

Behaviors are at the heart of the Virtools engine. They are descriptions and implementations of the actions and reactions that an object can have relatively to time or to its environment. Every dynamic change on objects during the execution of a Virtools content is implemented with behaviors.

Behaviors represent the verbs in the vocabulary of algorithms in Virtools. Other elements of this vocabulary are the Parameters (the adjectives), Owners and Targets (the nouns) and the Behavior Links (the punctuation). Verbs describe actions. And as we are in a computer system, they not only describe the actions, but actually perform them, when asked to do so by the Behavior Engine, at runtime. This action description can be in the form of Graphs or of C++ language.

General Description

From the outside, all behaviors have the same structure and way of functioning.

Here is the visual representation of the "ParameterSelector" behavior, as displayed in the Schematic Editor of Virtools. Outlined in red are the inputs and outputs of this behavior.



A behavior is activated when one of its input is activated. According to which input is activated the behavior performs its action and input parameters can influence this action. When done a behavior activates one or more of its output to indicate a result or situation and propagate this way the activity flux. Behaviors can also store result information in output parameters. Finally, they decide whether or not they want to remain active. See the [Presentation of the Process Loop](#) chapter for more information on when and how behaviors are actually called.

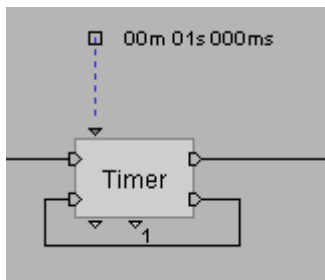
All behaviors have an owner, to which it is attached. The owner of a behavior never changes over time. The owner is a Behavioral Object. Some behaviors have a target, which is either its owner (default case), either a specific other object, which may change over time. The target of a behavior is also a Behavioral Object. See the [What are Behavioral Objects ?](#) chapter for more details.

Behaviors can use or generate data of about any type. Data passing between various elements of the Virtools engine, and especially between Behaviors, are called Parameters. See the [What are Parameters ?](#) chapter for more details.

Types of Behaviors

In Virtools there are three types of behaviors:

- The basic type of behavior is a **Building Block (BB)** that is implemented as a C++ function :

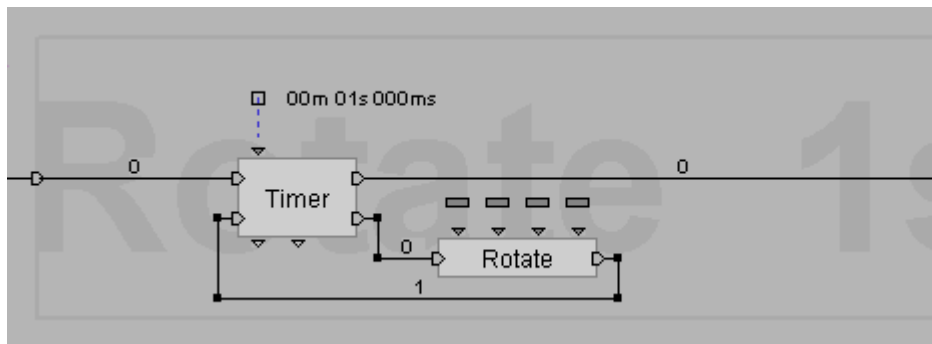


Since during one "frame" a lot of behaviors can be executed, each behavior is supposed to perform only a short operation. For example if we want a BB to be waiting 1s before triggering its output we must keep it active for as many "frames" as needed for the time to elapsed. That's why in this case the BB activates its second output (and this way reactivates itself for the next "frame") until the time specified in its input parameter is elapsed.

Building Blocks implementation reside in DLLs defining Behavior Plugins. All the behaviors provided with Virtools are Building Blocks. Using the SDK, software programmer can create their own Building Blocks, to add new functionalities to Virtools objects. Building Blocks are the way to go to add new behavioral capabilities to Virtools content. See the [Creating Building Blocks](#) chapter for more details.

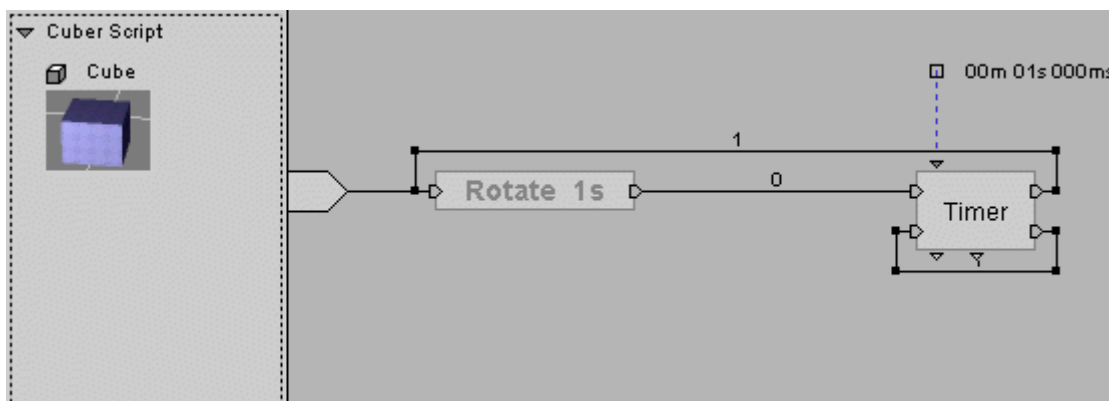
- Building Blocks or Behavior can be encapsulated in a larger behavior that we call **Behavior Graphs** :

When executed this behavior graph will execute the timer BB which will activate the rotate BB. Since the activation delay is 0 this BB will be also executed in the same "frame" and will in its turn activate the Timer BB to be executed next "frame".



Behavior Graphs can be recursively encapsulated inside other Behavior Graphs, and so on. Behavior Graphs are created using the Virtools application, and saved inside the Virtools content. You cannot create a Behavior Graph using the SDK, though you may access existing Behavior Graphs.

- The top level of a hierarchy of Behavior Graphs is called a **Script**. A Script is a Graph Behavior itself and has only one input :



This input is automatically triggered when a Virtools composition is started or when a script is reactivated at runtime.

Using the SDK, you can dynamically add or remove scripts from a Behavioral Object.

Behaviors and the SDK

The base class for all types of behaviors is [CKBehavior](#).

The SDK is primarily designed to define new behaviors in the form of Building Blocks encapsulated inside DLLs. You should never have to directly create an instance of [CKBehavior](#). The creation of the instance is done by Virtools, according to the description of the behavior that you provide in the form of a Behavior Prototype ([CKBehaviorPrototype](#)). This prototype also defines the C++ functions that will be called at various moments in the lifetime of the behavior, especially the execution function. These functions receive, as a C++ function parameter, the instance of the [CKBehavior](#) that it relates to, so you can get access to it. See the [Creating Building Blocks](#) chapter for more details.

You may also encounter behaviors while developing other kind of Virtools Plugins, like Managers.

In all cases, you can access the behavior's state and structure, and change them. You rarely directly "call" a behavior, rather, you change its settings to alter the moment or the way it is used by the Behavioral Engine.

What are Parameters ?

General concept

Parameters are used to transfer data between behaviors, and to add information to objects (through the use of attributes). A parameter is defined by the type of value it contains. Many value types have been already defined (float, string, integer, enum, object, etc.) but new types can be defined using the SDK (See the [Creating Extensions Plugins](#) chapter for more details).

Parameters and related objects

Kind of parameters

There are 3 kind of parameters : Input Parameters, Output Parameters, Local Parameters.

Input Parameter (pIn) :

Outlined in red are the two Input Parameters of the "Parameter Selector" Behavior, as represented in the Schematic Editor of the Virtools application.



Input parameters do not store any value, they are simply data inputs for behaviors and define the name and type of the data that is needed. input parameters are defined by the behavior. They get their value from a source which can be either another input parameter (from the enclosing Graph), an output parameter or a local parameter (i.e. the output of another behavior, the result of a parameter operation, a local parameter..). In the Virtools application, they appear on the top of the behaviors. In an analogy with C language an Parameter Input can be seen as a typed pointer: Either NULL or pointing to a local or output parameter.

When an input parameter value is asked, it is found by asking the value from its source. See below ([When are the values computed and propagated](#)) for more details about value propagation.

Output Parameters (pOut) :

Outlined in red is the only output parameter of the "Parameter Selector" behavior, as represented in the Schematic Editor of the Virtools application.



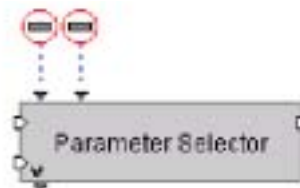
An pOut stores its own data in memory. As an pIn has a source, a pOut can have one or more destinations, which are pOuts or local parameters. In Virtools application, pOuts appear at the bottom of the behavior. They are also used when adding attributes to objects in Virtools.

When an pOut changes, its new value is automatically and instantly propagated to its destinations. See below ([When are the values computed and propagated](#)) for more details about value propagation.

Note that as destinations of pOuts are pOuts or local parameters, and not pIns, the destination of any pIn's source cannot be this pIn itself. And vice-versa : the destinations of a pOut do not have a source (as they are pOuts or local parameters).

Local Parameters :

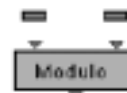
Outlined in red are two local parameters, which are the sources of two pIns of the "Parameter Selector" behavior, and belonging to the enclosing Graph, as represented in the Schematic Editor of the Virtools application. The "sources" links are represented by dashes.



Local parameters are like pOuts: they store their value with the exception that can not have any destinations. It can be either created in memory in a Building Block , in order to store local values, or directly in the Schematic Editor of the Virtools application, in which case it belongs to the enclosing Graph Behavior.

Parameter Operations

Here is a parameter operation, which two pIns have two local Parameters as sources, and of which Parameter Outputs does not have destinations, as represented in the Schematic Editor of the Virtools application.



Parameter operations define and implement simple operations between parameters such as addition, multiplication, etc... They are limited to two pIns, which may or may not have sources. They always have one pOut.

A Parameter operation is only evaluated when the value of its pOut is asked, and it is evaluated each time its pOut value is asked. See below ([When are the values computed and propagated](#)) for more details

about value propagation.

New Parameter Operations can be defined using the SDK. (See the [Creating Extensions Plugins](#) chapter for more details).

Parameter Links

Links between parameters do not have a class representation in CK2. They are just visual representations, in the Virtools application, of the source of an pIn, or of the destinations of an pOut. Note that these two types of Parameter Links have the same visual representation in Virtools, even though their meaning is different.

When are the values computed and propagated ?

Here are the rules:

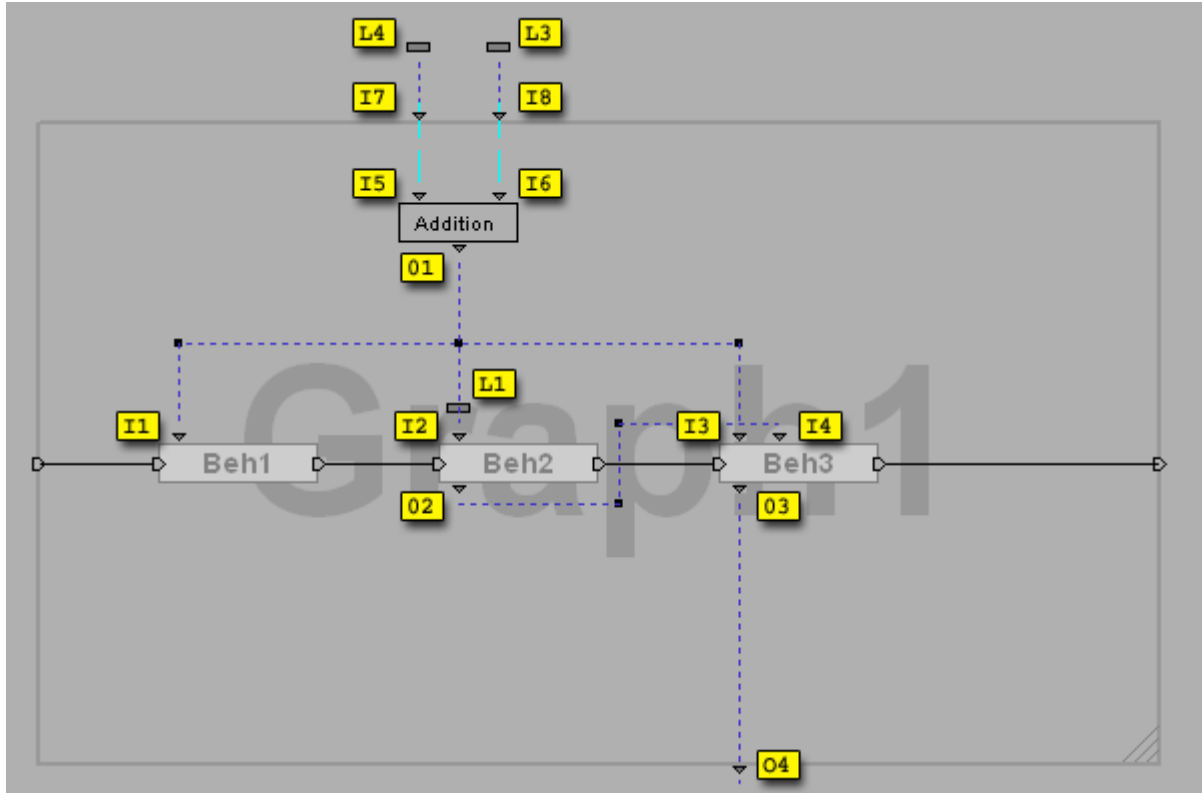
And pIn does not store its value, but pulls (asks) it from its (unique) source.

An pOut or a local parameter stores its value.

A pOut immediately pushes its (new) value towards its destination(s).

If a pOut is the output of a Parameter Operation's pOut, when asked its value, it triggers the execution of the operation (which updates its value) before returning it to the requesting pIn and pushing it towards its destinations.

Here is an example:



- When the Graph1 is executed, it executes the first behavior of its Graph, i.e. Beh1.
- Beh1 needs the value of its pIn I1. The source link of I1 is followed.

- The source of I1 is O1, the pOut of the Parameter Operation PO1 (an Addition). As it is the pOut of an Parameter Operation, the operation is executed. To execute PO1, the values of its plns, I5 and I6, are needed.
- The sources of I5 and I6 are also plns, but of the enclosing behavior Graph1. We follow their sources again.
- Their respective sources are L2 and L3, two Local Parameters of the Behavior enclosing Graph1. Their values are used as the inputs to PO1. The value of O1 is computed.
- O1 has one destination, L1, a Local Parameter of Graph1. The new value of O1 is propagated to L1, which stores it.
- The new value of O1 is also returned to I1, which requested it. So we now have I1's value: Beh1 is executed.
- Suppose that it activates its output. Beh2 should be executed.
- Beh2 needs the value of I2. The source of I2 is L1. L1 has its value stored and returns it to I2: Beh2 is executed.
- Beh2 has an pOut, O2, that it fills with a computed value. O2 stores its value.
- Suppose that Beh2 activates its output: Beh3 should be executed. It needs I3 and I4's values.
- For I3, the source is O1, the pOut of PO1. As PO1 is a Parameter Operation, it is again computed, using the same process as before (eventually getting the values from L2 and L3 and performing the operation, and O1 pushing its new value down to L1, and returning it to I3, which had requested it).
- For I4, the source is O2. O2 has a stored value and returns it to I4.
- We have the values for I3 and I4: Beh3 is executed.
- It fills up the value of its pOut, O3.
- O3 has a destination, O4, so it propagates its new value to O4, which stores it.
- Suppose that Beh3 activates its output: the output of Graph1 is also activated, and we are done executing Graph1.

Overall, during the execution of Graph1, the PO1 operation has been computed twice using the enclosing Local Parameters L2 and L3, and the pOut O4 has been set to a new value. We see that Local Parameters act as a kind of data buffer, that Parameter Operations are always computed, that plns pull their values from their source and that pOuts push their values down to their destinations.

In this example, we can also see that we could have optimized the process by having I3 get its value from L1, instead of O1, thus saving one execution of the Parameter Operation PO1.

Parameters and the SDK

First of all, with the SDK, you can create new Parameters Types. Usually, when you defines new types, you need to define new Parameter Operations that know how to act on these new types. By analogy, it is like creating a new class in C++, with its associated methods. See the [Creating Extensions Plugins](#) chapter for more details.

The second case where you encounter Parameters while using the SDK is when you write Building Blocks. In the Building Block's functions (execution function, callbacks), you usually need to access the Parameter Inputs and set the value of your Parameter Outputs. In the SDK, the [CKBehavior](#)^S class provides helper functions to iterate over, and access the plns and pOuts.

In a pln, you can get the source, which is a pOut or a Local Parameter, and get its value. There are helper functions that automatically walk the Parameter Links to get the value.

In a pOut, you can set the value, which is automatically propagated to its destinations.

When you write Building Blocks, you can also create a Local Parameter for the Behavior, which will not be seen from the outside, but can be used to store local data between frames. You can set and retrieve the value of your Local Parameter in your execution and callback functions.

What are Behavioral Objects ?

General Concept

Behavioral Object have Behaviors

While behaviors are the central concept inside Virtools, they would be of no use if they were not attached to objects. Inside Virtools, the Behavioral Objects (BeObjects) are the objects which can have behaviors attached to them. They do not necessarily have behaviors, but they can. The BeObjects are the nouns of the Virtools language. The behaviors attached to a BeObject may change this object or other objects, behavioral or not. All behaviors are attached to one and only one BeObject. BeObjects may have zero or many behaviors attached. When a BeObject has behaviors attached, it is said to be the **owner** of these behaviors.

Most of the objects found in the Virtools SDK are Behavioral Objects, but not all. For example, none of the objects of the VxMath library are BeObjects. All BeObjects are instances of classes derived from the important base class [CKBeObject](#)[§].

Behavioral Objects have Attributes

Behavioral objects can also have attributes, which is a way to attach almost any type of data to these objects. An attribute is defined by a type (or name) and the type of parameter value that is associated. Attributes data is stored inside Output Parameters so they can be used inside behaviors, through their Input Parameters.

Behavioral Objects and the Process Loop

Behavioral Objects are the entry point of the Behavioral Process of the Process Loop. All the processing that is done in this phase is done inside behaviors that are in a way or another attached to a BeObject.

Behavioral Objects have a priority that is used by the Behavioral Process to decide the order in which BeObjects are iterated.

See the [Presentation of the Behavioral Engine](#) chapter for more details.

Behavioral Objects and the SDK

In the SDK, you generally access BeObjects through a specialized class that derives from [CKBeObject](#)[§] (for example the [CKMesh](#)[§], or [CKTexture](#)[§]), which provides specialized functions.

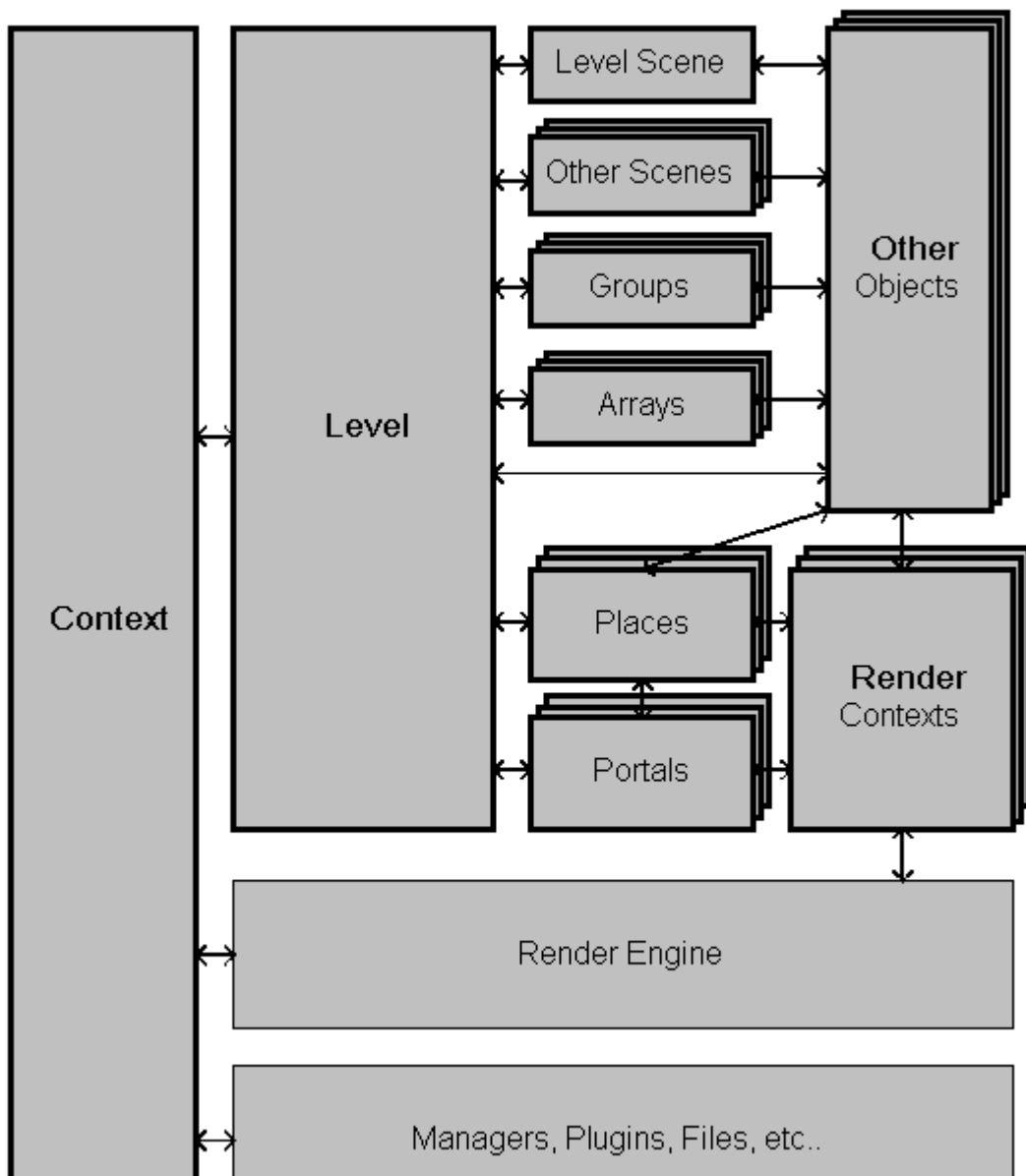
You may have to access the [CKBeObject](#)[§] class directly, for example to get or set an attribute's value, or the priority, or more rarely to attach or remove a script.

Typically, access to a BeObject happens inside the execution function of a Building Block, and on the owner or on the target of the BB. See the [What are Behaviors ?](#) chapter for more details.

Level Organization

A Virtools composition (a cmo file) contains one and only one Level. The Level is the global container for all objects of the composition. There are many other ways of organizing objects inside Virtools, which are all accessible both through the Virtools application and through the SDK. Here are these means, which all have a different meaning and usage :

- Context
- Level
- Scenes
- Places
- Portals
- Groups
- Arrays
- Render Context





Context

There is one context ([CKContext](#)^S class) per Virtools composition. The context contains and gives access to all the objects and algorithms corresponding to a Virtools composition, including file management, memory management, main Process Loop, Managers, Plugins, Render Engine, etc.. It is the main organizational entity of CK2.

There may be many contexts in memory at a given time, if there are many Virtools compositions. Each context correspond to one level. A context is not a Behavioral Object.

See the [CKContext](#)^S class chapter for more details.

Level

There is one and only one Level (class [CKLevel](#)^S) per Virtools composition. The Level gives access to all the composition objects : BeObjects, Behaviors, Places, Scenes, Parameters, etc.. It also manages one or many RenderContexts.

The Level is a Behavioral Object itself, so behaviors can be attached to it.

A Level has at least one Scene, which is created when the Level is created. This scene is called the Level Scene, and contains all the objects of the Level.

Scenes

There is one or more Scenes (class [CKScene](#)^S) in one Level. A default scene is created along with the Level, which is called the Level Scene.

There is always one and only one active Scene per Level at a given time, and only the objects of the currently active scene may be active. The objects that are not member of the active scene will not be active (they will not be visible and their scripts will not be executed). So scenes are the preferred way to segment the level into large scale narrative elements or large time-based periods. The Level Scene (containing all the objects of the level) is made the active scene of the Level at startup. When a new scene is made active (started), the previous active scene is automatically deactivated.

When a scene is made active, it puts all its renderable objects (class [CKRenderObject](#)^S) in the RenderContexts of the Level.

Each object that is referenced in a scene has settings about what occurs when the scene is started :

- Activity : To indicate if the object or script should be activated, deactivated or left in its current state when we start the scene.
- [Initial Conditions](#) : Objects can have [Initial Conditions](#) (IC) on them so they start in given state, for example in a given position and orientation for a character.

Scenes are Behavioral Objects so they can have behaviors attached to them.

See the [Scene Management](#) chapter for more details.

Places and Portals

Just like Scenes are a way to organize time segmentation inside a Level, Places (class [CKPlace](#)) are a way to organize space segmentation in a Level. There may be zero or many places in a Level. There is no specific link between Scenes and Places.

Places are linked to each other by Portals (class [CKPortal](#)). One portal links two places. A place may have many portals, even to the same other place.

Places may contain zero or many 3D entities ([CK3dEntity](#)), of which they are the "Parent", so places are part of the 3D entities hierarchy. To add an 3D entity to a place, one only needs to add its higher ancestor to the place.

Places and Portals are used to optimize the rendering phase. The decision of what to display is done through the active camera. When the Portal system is active (Through the "Portal Management" building block for example), the rule is the following :

Before each rendering phase, the list of places spatially containing the active camera of the level is computed, using the places bounding boxes. These places and all the places that have a portal with these places (and this recursively) are considered to be potentially visible, and processed through the rendering scheme. If an object is not member of one of these places, it will not be seen : it is immediately considered "out-of-view" (even though it may be active and its behaviors may be executed). If the Portal system is not active, there is no optimization of the rendering by places and portals (every 3D entity is considered potentially visible and only a hierarchical culling is done), but places may still be used to organize 3D entities hierarchically for other purposes.

Places are Behavioral Object and so can have behaviors attached to them.

See the [Using Portals](#) chapter for more details.

Groups

Groups are the simplest and most powerful organizational entity of Virtools. Groups are ordered lists of Behavioral Objects. They can correspond to any logical organization that you need. They may be created on the fly, reordered, iterated on, etc.. Many standard BuildingBlocks are provided for groups management.

Groups are themselves Behavioral Objects, so they can have behaviors attached to them, and be part of other groups.

See the [Using Groups](#) chapter for more details.

Arrays

Arrays are bidimensional matrices, which can contain simple data like floats, but also any kind of objects. Arrays are organized as typed columns (all the elements of a column must have the same type). They may be created on the fly, sorted, filtered, searched through, iterated on, etc.. Many standard Building Blocks are provided to arrays management.

Arrays are themselves Behavioral Objects, so they can have behaviors attached to them.

Set the [Using Arrays](#) chapter for more details.

Render Contexts

There are one or many Render Contexts (class [CKRenderContext](#)) per level. Render Contexts define which objects (class [CKRenderObject](#)) will be processed by the RenderEngine. Objects which are not in any Render Context will not be rendered at all. Objects which are in a RenderContext are eligible for

rendering (even though they may not be rendered, if they are out of view of the camera, if the Portal system disables them, if they are invisible, etc..)

The render contexts are automatically managed by the system: when a scene starts, all its RenderObjects are added to the RenderContexts of the Level.

RenderObjects can be added to a RenderContext, in which case they will be rendered, whether or not they belong to the active scene or even to the Level. This is a easy way to create temporary objects used for rendering.

See the [Using Render Contexts](#) for details.

Presentation of the Render Engine

The Virtools engine uses an in-house 3D Render Engine, very modular, and ported on three different 3D APIS (DirectX5, DirectX7, and OpenGL).

The Render Engine of Virtools is called CK2_3D.

Description

The Virtools Render Engine is a 3D engine abstraction, as its API is independent of specific 3D hardware and software. This abstraction isolates the rest of the libraries, and especially the Behavioral Engine, from the details of 3D implementation.

Here is the list of 3D features that it provides or implements:

- Hierarchical keyframe animation system with motion blending (mix of several animations) and warping (transitions)
- Level of Detail on character animations
- Support for Linear, Bezier and TCB animations
- Morph animations (with blending and warping too)
- Skin & Bones System
- Bicubic Patch Meshes (uniform tessellation)
- Progressive Meshes
- Spline Curves
- Per Vertex Color
- Procedural textures and movie textures

- Multi Texturing
- Rendering to a texture
- Billboarded or axis constrained 3D Sprites
- Sprites with hierarchical system
- Hierarchical culling
- DirectX5, DirectX7, OpenGL support
- Compressed textures (DX7 , OpenGL)
- Pentium III optimizations

And of course the standard set of default features (hardware dependant for some of them...):

- Dynamic RGB lights
- Vertex Fog
- Mipmapping
- Transparency
- Texture Filtering
- Perspective correction texture mapping
- etc...

A lot of behaviors implements additionnal visual effects and fonctionnalities such as :

- Particle system
- Portal culling
- Reflections
- Environment Mapping
- Collision Detections
- Simple Shadows, Shadow casting
- Motion blur
- Lens Flares
- Multi object morphing
- Level of Detail
- 360° environment map

Render Objects

The Render Engine actually implements all the Virtools objects having to do with geometry and imaging, like 2D entities, 3D entities, places, textures, lights, cameras, meshes, etc... Most of these object derive from the [CKRenderObject](#)^S class, but not all. All the instances of these objects live in the Render Engine, which is responsible for the memory management, the global consistency, and the interrelationships between these objects.

Also, the Render Engine manages the Render Contexts, that are used to keep a list of objects that should be rendered, and that are heavily used by the Behavioral Engine.

Rendering Loop

The Rendering Engine is responsible for the Rendering Process of the main Process Loop. This phase is called the Rendering Loop.

The Rendering Loop provides many callbacks that can be used by software elements inside Virtools in order to customize their rendering.

For more information, see the [About the Rendering Loop](#) chapter.

The main Process Loop is described in the [Presentation of the Process Loop](#) chapter.

Rasterizer

The CK2_3D Render Engine itself is built around a Rasterizer abstraction. A Rasterizer is an engine which, starting from a list of polygons, images, and rendering options, draws one frame. The rasterizer takes care of the relationships with specialized 3D hardware and software, thus isolating the Render Engine implementation from the very gory details.

The CK2_3D render engine provides three rasterizers: one for OpenGL, one for DirectX5, one for DirectX7.

For more information on Virtools rasterizers, see the [Presentation of the Rasterizers](#) chapter.

Render Engine and the SDK

Using the SDK, there are three ways software programmers can act on the rendering process.

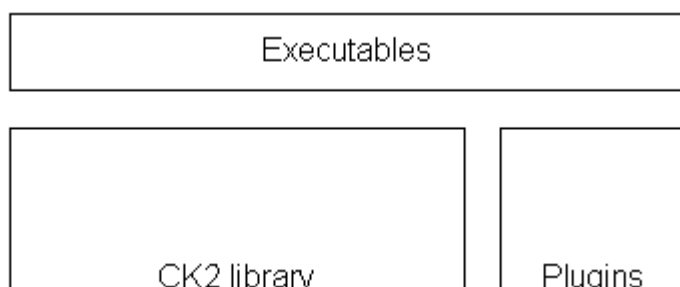
The simplest way, which should be preferred when possible, is to attach software elements to the Rendering Loop callbacks. This can be done at a global level, at the object level, at the behavior level. Usually, for each level, there is a pre-render callback and a post-render callback. See the [Custom Rendering](#) chapter for more details.

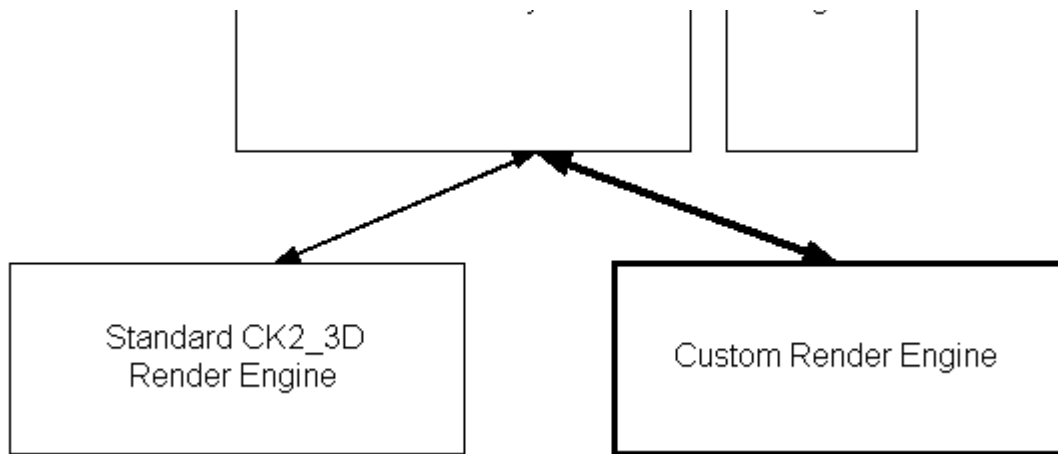
The second way is to replace the Rasterizer. The three Virtools rasterizers source code is provided with the SDK. See the [Presentation of the Rasterizers](#) chapter for more details.

The third way is to change or modify the RenderEngine. The CK2_3D source code is provided with the SDK. For details, see the [Changing the Render Engine](#) chapter.

Changing the Render Engine

The Virtools library is designed so that the standard Render Engine can easily be replaced. This can be useful if you have your custom 3D engine that you would like to use.





Remember that you can highly customize the rendering process of the Render Engine, using various settings and callbacks. You should prefer this solution, whenever possible. See the [Custom Rendering](#) chapter for more details.

Remember also that the Virtools Render Engine is built on a software abstraction called Rasterizer. It may be simpler to create a Rasterizer than to create a whole Rendering Engine altogether. See the [Presentation of the Rasterizers](#) chapter for more details.

Software architecture

The Virtools Render Engine API consists of pure virtual C++ classes. These classes are declared in the CK2 library, but implemented in the Render Engine Plugin.

These classes correspond to all the objects that have to do with rendering, visualization, geometry and space, like [CKMesh](#), [CKCamera](#), [CKLight](#), [CKTexture](#), etc.... The Render Contexts ([CKRenderContext](#)) are the main organizational entities of the Render Engine, as they keep track of which objects should be rendered. (not to be confused with CKContexts, which are related to the main execution context of a Virtools content).

Replacing the Virtools Render Engine Plugin consists in creating a DLL that provides the CK2 library with another implementation of these pure virtual classes. It is thus a matter of replacing the current Render Engine Plugin DLL (CK2_3D.dll) with a new one, implementing the same pure virtual C++ classes.

Using the provided Render Engine source code

The SDK provides the full source code of the Virtools render engine. The simplest and most foolproof way to create a new Render Engine Plugin is to start from the provided source code, to modify it in order to include your changes, optimizations or custom rendering engine. You may replace existing functions, modify them or add your own. Then, by recompiling the DLL you can have access to your custom rendering algorithms. See the [Available Source Code](#) for details.

Note that you should not use this source code outside of the Virtools SDK context, like in another software suite without permission from Virtools.

About the Rendering Loop

The Rendering Loop and the main Process Loop

The Rendering Loop is called by the main Process Loop to do the rendering. It corresponds to the Rendering Process, that happens in every main loop cycle, after the Behavioral Process. See the [Presentation of the Process Loop](#) chapter for an overview of the main Process Loop.

The rendering loop is managed by the Render Manager, which manages all the Render Contexts. The [CKRenderManager::Process](#)^S executes the Rendering Loop. Optionnally, your executable can keep trace of all its render contexts and use the [CKRenderContext::Render](#)^S function to do the rendering. This is the preferred way if you want to master all the rendering options, and this is the solutions used by all the Virtools executables.

The Render Manager and Render Context implementations are part of the Render Engine. See the [Presentation of the Render Engine](#) chapter for an overview of the Render Engine.

The Rendering loop is highly customizable, through options and through callbacks. It basically draws all the objects starting from the background 2D objects, continuing with the 3D objects, and finishing with the foreground 2D objects. For details, see the [Understanding the Render Loop](#) chapter.

Callbacks

The standard rendering loop should be suffisant for most of your needs. But you may want to insert some custom processing at some moments of the rendering loop, like for example to draw temporary objects, or to make some late computations that are only needed if the object is actually rendered (thus saving some processing time if the object is not rendered). Callbacks are possible at all phases of the Rendering Loop.

To get called back, you should first attach your functions at the right place (this can be at the object level, or at a global level). You specify the function that you want called back, and when you want it called back. Dont forget to remove your callback when you dont need it any more.

Your function is called back during the rendering process. It then has access to the current Render Context, which provides all the drawing primitives.

For details on render callbacks, see the [Custom Rendering](#) chapter.

Presentation of the Rasterizers

Presentation of the Rasterizer

The Rasterizer is a software abstraction introduced by Virtools, and on top of which the standard 3D engine is built. See the [Presentation of the Render Engine](#) for an overview of the Render Engine.

The Rasterizer is an API dealing with polygons, textures and rendering settings, and which basically, given these 3D elements, draws one 2D image intended to be shown on screen. It thus corresponds to the last phase of the rendering. It is a crucial phase, performance-wise, and it is often accelerated using specialized 3D hardware. As many such hardware, and hardware APIs exist, Virtools introduced this abstraction, in order to isolate the Render Engine itself from the underlying gory details.

Virtools comes with three rasterizers, for the most common 3D systems : DirectX5, DirectX7, and OpenGL.

Each rasterizer is implemented in a DLL. The Render Engine by default chooses the best rasterizer given

the environment, but it can be forced to choose any other Rasterizer.

Creating a new Rasterizer

The Render Engine is designed to accept more rasterizers than the standard ones. So you can write your own rasterizer and have the Render Engine use it.

Keep in mind that the rendering engine provides many ways to customize the rendering, so before starting to write your own rasterizer, which is not an easy task, you should check that the standard Render Engine options do not satisfy your needs. See the [Custom Rendering](#) chapter for more details.

The source code of the three standard rasterizer is provided with the SDK, so this is a good place to start if you want to write your own rasterizer. See the [Available Source Code](#) chapter for details.

The process of creating your own rasterizer is similar to (but different from) the other plugins:

You should write your custom Rasterizer class, deriving from the *CKRasterizer* class, and compile it into a DLL. This DLL should be placed along with other rasterizers (for example in "**RenderEngines**" directory in Virtools) so as to be taken into account by the Render Engine.

The exported *CKRasterizerGetInfo* function is called when the Render Engine starts up, and is supposed to declare the Rasterizer capabilities and entry points. This is the Registration phase.

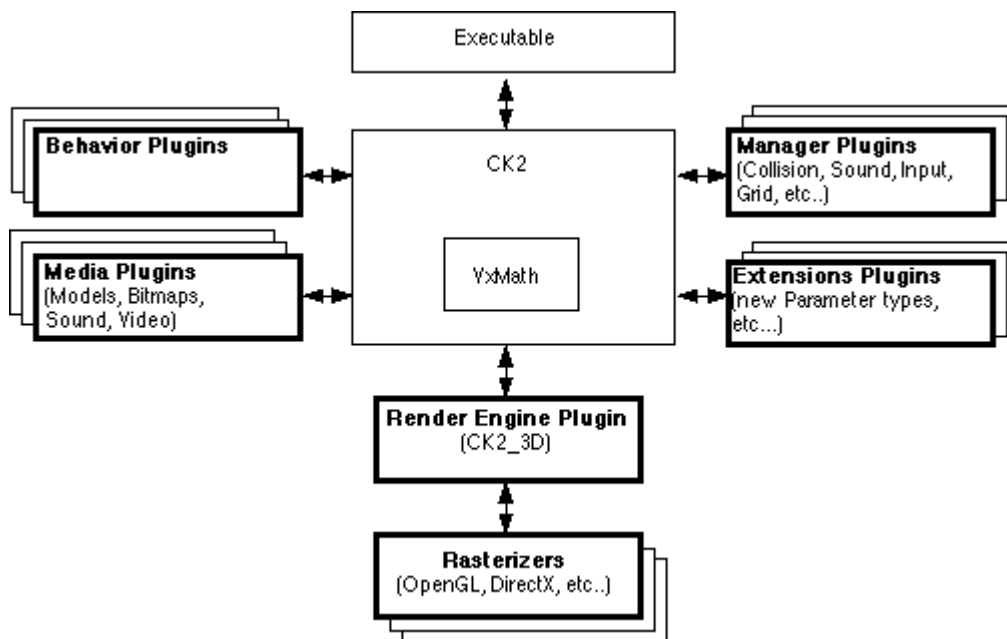
Later on, when needed, the Render Engine calls your Rasterizer for various operations, like uploading of textures, declarations of polygons, specification of setting, and of course rendering.

Virtools Plugins Overview

Around the CK2 library, which constitutes the core engine of Virtools, all the other functionalities of Virtools are implemented through plugins. A Plugin is defined as a DLL that comply to a certain set of rules and protocols, so that the core engine knows how to talk to it.

Virtools defines various kinds of Plugins :

- Behavior Plugins : implement behaviors that can be applied to objects,
- Media Plugins: import and export external media formats: bitmaps, video, sound or model
- Managers Plugins: perform system-wide operations such as collision management,
- Extensions Plugins : add new parameter types,
- Render Engines Plugins : implement 3D rendering
- Rasterizers : implement rasterizing

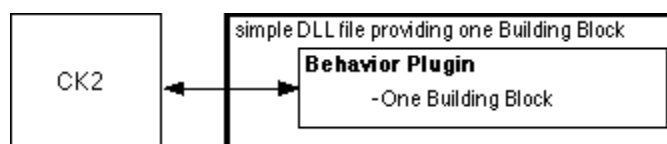


The SDK gives software programmers the ability to write any kind of Plugin, in order to extend Virtools functionalities.

Examples are provided in the SDK for all the Plugins types that can be written with the SDK. These examples are a good start for writing your own Plugins. See the [Available Source Code](#) chapter for more details.

General Plugin architecture information

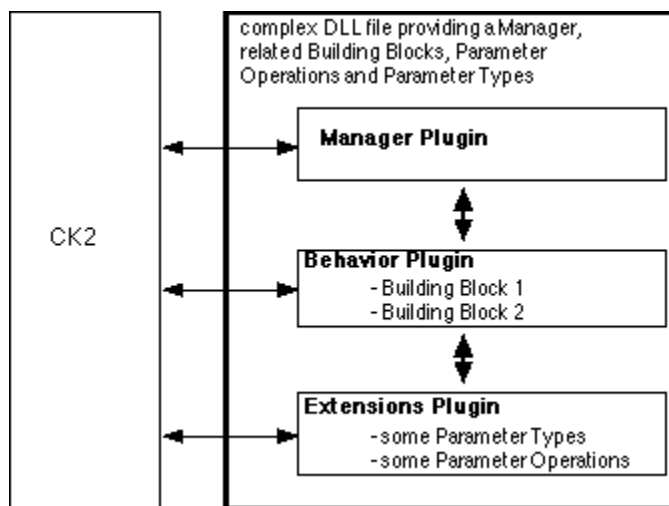
Plugins using the SDK should be written in C++, and be compiled into a DLL. The only supported development environment is VisualC++ version 6.0. See the [Compilation Settings](#) chapter for more details.



These DLLs should then be placed in specific places in the file system in order to be taken into account by the Virtools library. If you write your own custom executable on top of the CK2 library in addition to writing Plugins, you can change the list of places that are used to search for Plugins. See the [Using Plugin Manager](#) chapter for more details. In Virtools behaviors (and their related managers) plugins are placed in the "**VirtoolsBuildingBlock**" directory, standalone managers in the "**Managers**" directory, Render engines and rasterizers in the "**RenderEngines**" directory and medias plugins in the "**Plugins**" directory. These directory are only here to organize things but their name is not used to identify the type of plugin inside so all the plugins DLLs could be placed in any directory they would still be recognized.

The Virtools library parses these Plugin DLLs at startup (so you should restart the executable for new Plugins to be taken into account). The Plugins management is done by the PluginManager. For each Plugin, a special function of the DLL (*CKGetPluginInfo* and the associated *CKGetPluginInfoCount*) is called to know which type of Plugin it provides. This is called the Registration phase.

Note that one DLL can provide many Plugins of different types, like for example a Manager, some associated Parameters and Behaviors. During the registration phase, the DLL should register all the Plugins that it provides.

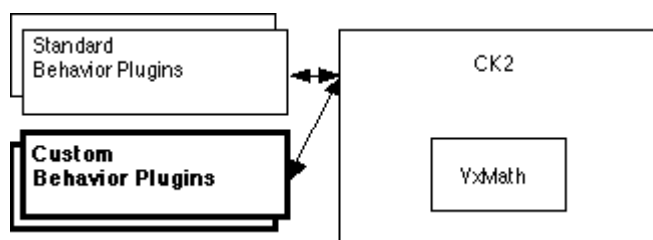


See the See the [Creating New Plugins](#) chapter for more details.

Later on in the life of the Plugin and of the CK2 library, other functions of the DLL are called back. Exactly which function is called back at which moment depends on the Plugin type. See later for details.

Here are the main types of plugins that the Virtools core engine expects and that can be written with the SDK:

Behavior Plugins



Behaviors are at the heart of Virtools. Basically a behavior is the description of how certain type of objects act and react in time and according to their environment. Behaviors can implement from very simple actions (an arithmetic operation) to very complex ones (AI simulation for characters).

The name *behavior* is quite generic, and there are actually three types of behaviors: Building Blocks, Behavior Graphs and Scripts. Behavior Graphs and Scripts are defined in a structural way inside Virtools content, and are created using the Schematic Editor of Virtools application. Building Blocks are implemented in the form of C++ code. This C++ code resides in compiled form inside Behavior Plugins contained in DLLs. See the [What are Behaviors ?](#) chapter for more details.

The behavioral engine of the CK2 library does not contain the code of behaviors, but only the knowledge on how to manage and process behaviors. All the standard Building Blocks provided with Virtools are implemented inside external Behavior Plugins. These Behavior Plugins are dynamically loaded inside the executable at runtime.

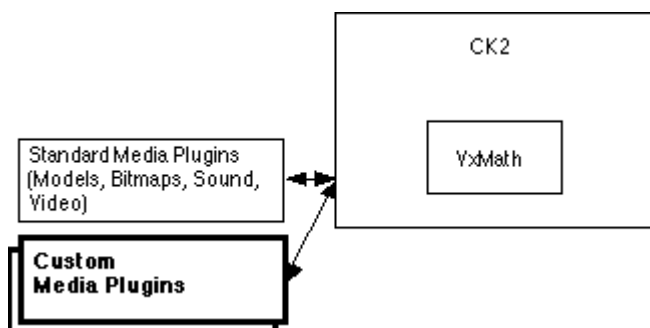
As we have seen the Behavior Plugins register their type and the Building Blocks that they provide. When a Building Block defined by the Behavior Plugin is needed (for example, when it is drag and dropped into the Schematic Editor for the first time), the DLL is dynamically loaded inside Virtools, is asked to create a Behavior Prototype corresponding to the needed Building Block. This prototype describes the entry points and the options of the Building Block, like for example to which type of object it applies, its execution function, whether or not it should be called back when specific events happen to the behavior (callback and render callback), etc... When the content is played and the behavior is activated, the execution function is called and should do its job. The callback and the render callback are optionally called for specific processing.

Note that the Behavior Plugin never creates an instance of the [CKBehavior](#) class : the instance is created and managed by the Behavioral Engine. The Behavior Plugin is a set of C functions, and not C++ classes. These functions act on the object instances provided by the Behavioral Engine.

Using the Virtools SDK, software programmers can write their own Behavior Plugins, using the provided examples and templates, and using the CK2 entry points like: [CKBehaviorPrototype](#), [CKBehavior](#), [CKParameterIn](#), [CKParameterOut](#), etc... The detailed description of how to write a Behavior Plugin and how to connect it to the main behavioral engine is detailed in the chapter [Creating Building Blocks](#).

Once written and compiled, Behavior Plugins can be distributed as closed software components that can be used inside Virtools, thus providing content developers with new functionalities that can be applied to objects. When a Behavior Plugin that is not part of the standard set of Behaviors is used in a content, the corresponding DLL must be distributed along with the content that uses it.

Media Plugins



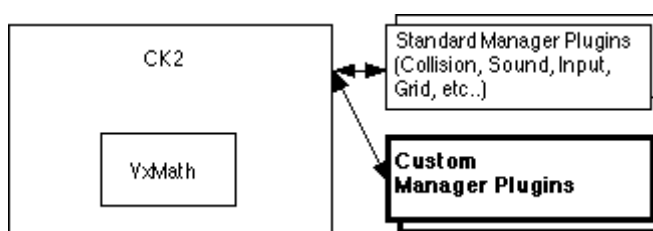
Virtools is an integration platform. Like behaviors, the media are external to Virtools, and Virtools has no knowledge of media formats. Media Plugins are software components used by CK2 to import outside media into a form usable in Virtools, thus transforming the outside data into representations that can be interpreted by CK2. Media Plugins can also optionally export media. Media Plugin import and export media from memory, files or URLs. The Virtools application suite contains a few Media Plugins corresponding to usual formats..

Using the SDK, programmers can write their own Media Plugins, to manage their specific media formats, like for example an in-house image format, or sound format. There are 4 subtypes of Media Plugins:

- Image Media Plugins
- Sound Media Plugins
- Video Media Plugins
- 3D Models Media Plugins

See the [Creation of Media Plugins](#) chapter for more details.

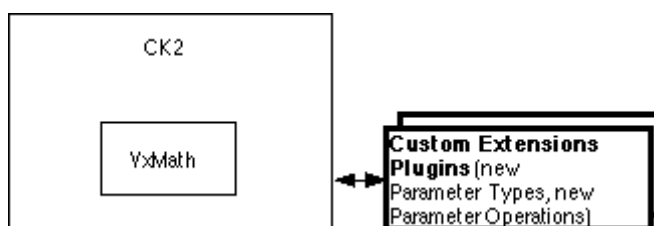
Managers Plugins



Managers Plugins are software components that manage system-wide algorithms. They typically plug into the main Virtools process loop, and have helper Behaviors related to them, that when attached to objects inside a Virtools composition, talk directly to the Manager for processing and data sharing. For example, the collisions, the grids, the attributes are all managed by Managers inside Virtools. Basically, Managers implement all the processing that is not part of the core behavioral engine, and that could be taken out of the system without removing anything else than the functionality that they implement. Also they implement all processing that needs shared processing or data between individual Behaviors.

Using, the SDK, software programmer will be able to write their own Managers, which all derive from the base class [CKBaseManager](#)^S. For more information see the [Creation of a New Manager](#) chapter.

Extensions Plugins



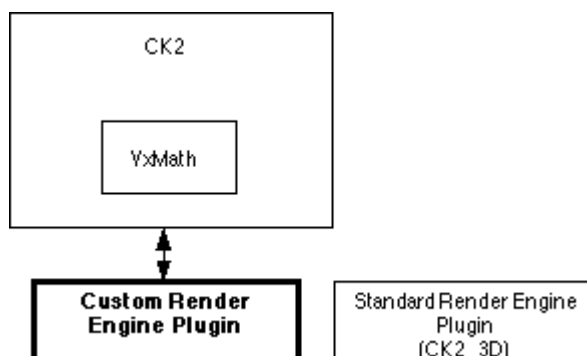
The Extensions Plugins is a generic name for all extensions to the CK2 library which are not Media Plugins, Behavior Plugins, Managers Plugins, RenderEngine Plugins or Rasterizers. Today, the only other type of extension are Parameter Types and Parameter Operations.

Parameters types declare and implement data types that can be processed by the Virtools behavioral engines. The Parameter Operations are small programs that apply simple operations on these parameters.

Using the SDK, software programmers can implement their own parameters types and parameter operations. Once these parameter types are defined, they can be used seamlessly inside Virtools to connect to Behaviors that accept these parameter types. Usually, new Parameter Types and Operations go along with new behaviors or managers. So one often finds one DLL containing a Behavior Plugin, a Manager Plugin and an Extensions Plugin.

For more information, see the [Creating Extension Plugins](#) chapter.

Render Engines



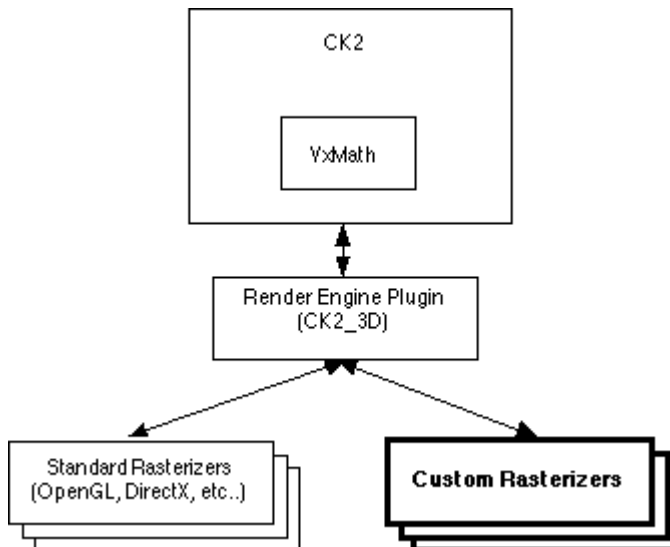
The Virtools core engine relies on a Render Engine for its 3D rendering. The Virtools software suite uses its own Render Engine, called CK2_3D. This engine manages and processes all the objects that need to be rendered (3D : meshes, animations, textures, 2D: sprites, other : camera, lights, etc..) It relies on a Rasterizer for the rasterizing phase, that is the transformation of polygons and textures into one 2D image.

Virtools is designed so that this standard 3D engine can be replaced by a custom engine. The SDK provides the source code of the CK2_3D render engine, which can freely be modified (as long as you use it

inside the Virtools core engine). You can either reuse the source code for your engine, or replace some functions implementation by hardwired functions that you provide.

For more information , see the [Presentation of the Render Engine](#) chapter.

Rasterizers



Rasterizers are software components that, given polygons, textures and rendering options, draw one 2D image for blitting on the screen. Rasterizers are an abstraction introduced by Virtools, and are used as an underlying API for the standard Virtools Render Engine. The Virtools software suite provides 3 rasterizers, for three different 3D hardware and system platforms: DirectX5, DirectX7, and OpenGL.

Using the SDK, you can write your own Rasterizer and have the Virtools core engine use it. The SDK contains the source code for all three standard rasterizers, which you can modify to write your own rasterizer (as long as you don't use this source code outside of the Virtools context). This can be useful to cope with specific 3D visualization hardware APIs, or to write your own faster than light software rasterizer for ultimate compatibility.

For more information , see the [Presentation of the Render Engine](#) chapter.

Main Virtools Events

Behaviors or managers developers should be aware of the main events that can happen in Virtools. Behaviors and managers can intercept these events to perform specific actions.

The classic example is for behaviors that add custom rendering effects on objects using callbacks, these effects should be available only when the composition is playing not when it is edited inside Virtools interface.

Reset :

The first thing that happens before a composition is played and also the way a composition is stopped. Behaviors and managers can be warned than a reset operation occurred. The behavior of our sample should remove its rendering callback when receiving this event.

Play (or Resume) : 

Managers and behaviors are warned through this event that we start/resume playing a composition. In this case the behavior of our sample would add its rendering callback if it is active.

Pause : 

Managers and behaviors are warned through this event that the execution of a composition is paused. The user has switch back to editing its composition inside Virtools for example.

Activation/deactivation of objects :

An object on which scripts where currently running can be (de)activated by another script. Behaviors that were on the objects won't be executed but can be warned of such events (in order to remove a rendering callback for example)

Scene Changes :

Since a composition can be decomposed into several scenes it may happen that, during the execution of a composition, a new scene is launched. Once again the behaviors and objects that are not referenced in the new scene won't be executed. So they may need to remove callbacks because they are not useful anymore.

Creation/deletion of objects :

At anytime an object can be created/loaded and added to the current scene. In the same way it can be deleted anytime, since developers may have added application data to such objects, these events should be handled if cleaning is required.

Rendering :

Since the behavioral process occurs before the rendering process, the execution of a behavior is not supposed to perform rendering but a behavior can add callbacks to modify/add rendering effects at almost any moment of the rendering loop.

Saving/loading :

Managers and behaviors are warned before and after a load/save operation occurs. The **load** event is often used by behavior to initialize their local data. In addition managers can add their own data to the file that is being saved.

Clear :

Managers and behaviors may have allocated memory during the edition/runtime of a composition. A "ClearAll" event is sent to managers when the user quits Virtools or starts editing a new composition.