



Version 1.5.0

User Manual

Revision 3.0

Copyright © 2022

Frank-Vegar Mortensen

Contents

Introduction.....	3
Basic Concept	4
The SnaX Developer	6
Main Menu	7
Main Toolbar	10
Chips Panel	10
Template Panel.....	11
Search Panel	12
Instances Panel.....	13
Project Tree Panel	13
Message Log Panel	15
Chip Messages	17
Call Stack.....	18
Project View	19
Editor Panel	19
Editor View	21
Chips	21
Connections.....	27
Folders	28
Background items.....	29
Navigation and Common Operations.....	30
Class Diagram	31
Property Dialogs	31
Chip Property Dialogs.....	31
Class Instance Dialogs.....	32
Publish Wizard.....	33
Programming Guide	36

Introduction

SnaX is the new development tool for real-time 3D-graphics applications! It is a tool for creating games, simulator graphics or any other visualization solutions depending on high performance real-time 3D-graphics. The tool implements a powerful *real-time, visual* programming concept. *Real-time* because you can instantly view the effect of all updates you make to your application as you develop – There are no compile steps or any other tedious processing needed after every little program modification.

Visual because there is no traditional coding involved in the development process – Everything happens in a graphical environment, designed to make programming both an efficient and fun experience!

Even though SnaX offers a high-level development environment, the underlying programming model builds on concepts directly comparable to popular programming languages like C++ and Java. It is an object-oriented programming tool, highly flexible in how you want to build your application. This gives you great freedom to create exactly what you want, without placing too many restrictions to functionality and performance.

SnaX is built for Microsoft Windows 10 using Direct3D 12 for state-of-the-art graphics.

This user manual should be considered as work in progress and is not yet a complete documentation. Hopefully, it will still be enough to help you get started using SnaX. If you have any previous experience in (object-oriented) programming and system development, this will for sure be of help in your process of learning this new programming tool!

Basic Concept

SnaX is based around a very simple, yet extremely powerful concept. It will allow you to create applications with the raw power of a native programming language without having to do any traditional coding, compiling and other low level fiddling it takes years to master. You can focus your energy in developing the application you want, using a visual programming method that implements many of the concepts found in a modern programming language, to create efficient, well-structured and maintainable code.

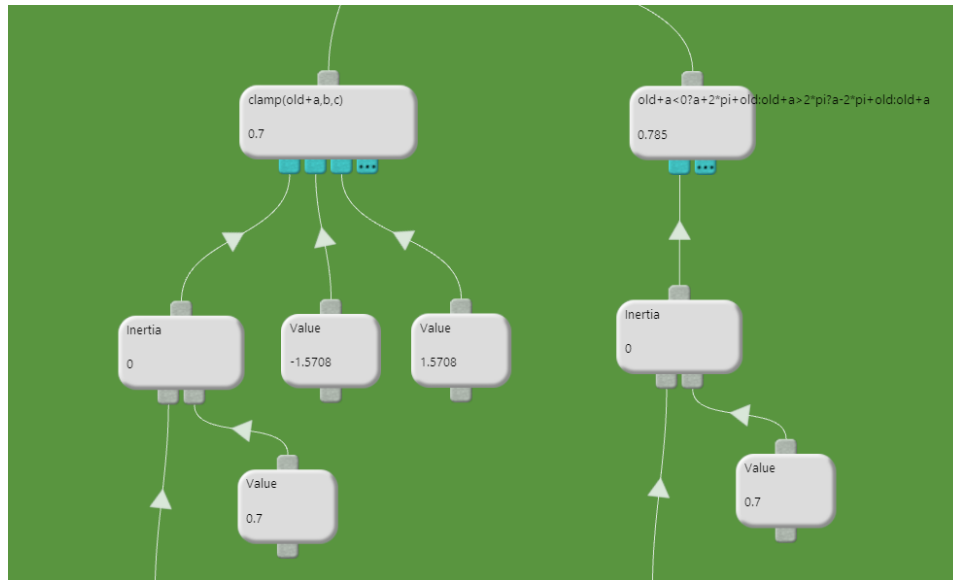


Figure 1 Chips are the building blocks of a SnaX application.

The secret behind this are small building blocks called *chips*. Just like the microchips found inside your computer, a chip is a black box with some limited, well defined functionality inside. You don't know how a chip is implemented, nor should you care. The important thing is for you to know what a certain type of chip is supposed to do, and how it can interact with other chips. The same way a microchip has a defined interface in form of a number of pins connecting it to a circuit board, connectors allow a chip to be linked to other chips, forming a piece of program logic as illustrated in Figure 1. Connecting chips with different functionality together allows you to construct increasingly complex functionality, until you finally have, for example, a fully functional computer game in front of you! This is what programming in SnaX is all about – Building virtual circuit boards, called *classes*, by adding, configuring and connecting different types of these virtual microchips. Programmers will recognize the term *class* as a concept of an object orientated language, and this is also the case for SnaX. It is a tool supporting many of the object-orientated concepts known from popular languages like C++ or Java.

The description of how the classes are made up by chips and connections, are saved as project files in a binary or an XML-based format, and can be considered the source code for a SnaX application. At the very basic, they only contain references for which chips to use, their configuration and how they should be connected. It is the chips that contain the executable, binary code that runs on your computer.

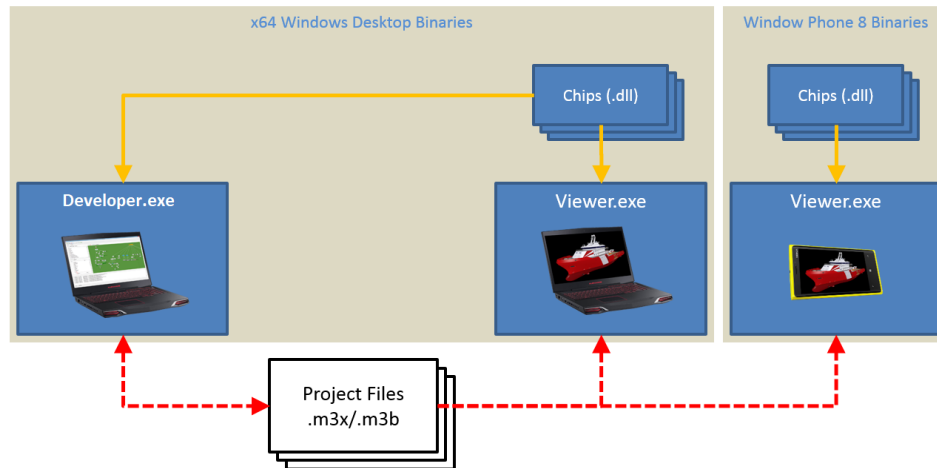


Figure 2 A schematic overview of the basic concept of SnaX.

This concept has one very interesting property. As your project files does not contain any executable code, they are mostly platform independent. Like most binaries, chips have to be compiled for the platform they are targeting. This means that the project you author with SnaX can run on any platform, as long as the chips they are referencing can be compiled for that specific platform. Currently, SnaX does only support Windows 10 desktop, but as illustrated in Figure 2, the concept has been tested on Windows Phone 8, which unfortunately was abandoned by Microsoft. The concept is still valid though, and plans are to support additional platforms in the future.

Another important fact about chips is that they are just plugins to a very generic framework. Still, the chips contain almost all of the functionality of SnaX. This is quite an interesting fact, because it means that SnaX is a very extendable tool. New functionality can easily be added by creating new types of chips, to for example add support for various third-party libraries (physics, sound, networking etc.), asset importers, hardware and so on. The game physics library by Nvidia, PhysX, is an example of this. The SnaX framework itself does not know anything about physics simulations, but support for PhysX was added by writing a packet containing chips interfacing this amazing library. The same is true for graphics. Graphics is provided through a set of chips building on the Direct3D 12 graphics library. It would be just as easy creating a set of chips interfacing other graphics libraries like for example OpenGL or Vulkan, which again would open the door for supporting other platforms like Linux or Android.

Chips are written in native C++ and are therefore extremely fast. Currently, there is about 100 different types of chips available, and the number is increasing. Plans are also to provide users with an SDK for them to be able to write their own chips in the near future.

The SnaX Developer

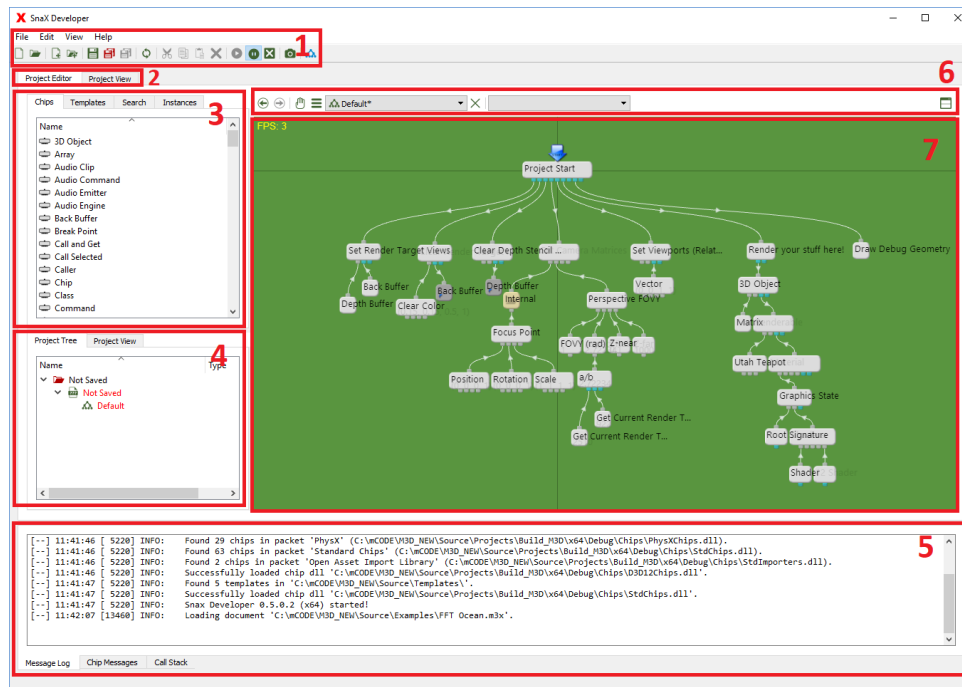


Figure 3 The SnaX Developer.

The SnaX Developer is the development tool you will use for creating your new 3D-graphics application. The Developer consists of the following main parts as indicated in Figure 3:

1. Main menu and main toolbar.
2. Selector for
 1. Main development environment.
 2. Large project view.
3. Tabbed panel for
 1. Chips.
 2. Templates.
 3. Search tool.
 4. Class instances.
4. Tabbed panel for
 1. Project tree.
 2. Small project view.
5. Tabbed panel for
 1. Message log.
 2. Chip messages.
 3. Call stack.
6. Tool bar for the editor panel with the editor view and embedded property dialog.
7. Editor view.

In the next sections of this chapter, we will go through the main parts of the Developer and describe their functionality.

Main Menu

The Main Menu consist of the following items:

- File
 - New Project

Closes the current project and creates a new one.
 - Open Project

Closes the current project and lets you open an existing project from disk.
 - Add New Class

Adds a new class to the current project. If a document is selected in the [project tree](#), the new class will be added to this document. If not, a new document will also be added to the project.
 - Import Class

Opens a file dialog and lets you select a file to import into the project. This could be an existing document containing classes, or a supported 3D-asset file format.
 - Save

Saves the *current document* to file. If the [project tree](#) has keyboard focus, the *current document* is the item selected. If the [editor view](#) has focus, the *current document* is the one containing the class currently open. The document will be saved to the file it was opened from. If it is a new document, a file dialog will be opened to let you select a file name and location.
 - Save As

This is equal to the 'Save' item, but a dialog box is always opened to allow you to change the file name and location of a document.
 - Save Dirty

Save all documents that have been modified since last save. Most changes you make to a class or chip will automatically mark the containing document as modified.
 - Save All

Save all documents currently open in the project. Be careful as this could make you save unintended changes! Consider using 'Save Modified' instead.
 - Load all project files

Recursively load all project files referenced in project files already loaded.
 - Publish

This will open the [publish wizard](#).
 - Recent Projects

This item will contain a list of the recently opened projects. Select an item in the list to reopen this project.
 - Exit

Closes SnaX Developer. You can not exit while in a break point, or while project files are currently being loaded.

- Edit
 - Cut

Copy and delete operation on the current widget. If the [editor view](#) has focus currently selected items are copied to the clipboard, and then deleted from the class.
 - Copy

Copy operation on the current widget. If the [editor view](#) has focus, currently selected items are copied to the clipboard.
 - Paste

Paste operation on the current widget. If the [editor view](#) has focus, previously copied items are pasted into the current class.
 - Delete

Delete operation on the current widget. If the [project tree](#) has the keyboard focus, the selected document or class will be deleted/removed from the project. You will be given the option if you want to just remove the document from the project only, or delete its file from disk. If the [editor view](#) has focus, selected items will be deleted.
 - Options

Opens a dialog box with various configuration settings. *Not yet implemented.*
- View
 - Class Diagram

Opens the [class diagram](#) in the [editor view](#).
 - Debug Geometry

The item's submenu lets you enable/disable various debugging geometries. The following options are available:

 - **World Grid:** Draws a simple grid in the XZ-plane around the origin of the world space.
 - **World Space AABB:** All 3D-objects will have their world space axis aligned bounding box (AABB) be drawn around them.
 - **Local Space AABB:** All 3D-objects will have their local AABB be drawn around them.
 - Profiling

This item's submenu lets you control SnaX's powerful [profiling functionality](#).

 - **Disabled:** If selected, all profiling functionality is disabled.
 - **Floating:** If selected, profiling is enabled and profiling data is reset at the beginning of every frame.
 - **Average:** If selected, profiling is enabled and profiling data accumulate over frames. This is the preferred option as it gives you the most accurate results over time.
 - **Reset:** Resets accumulated profiling data.
 - External Project View
 - **Enable External View:** Toggles the freely movable/sizeable [project view](#) window. The two internal project views are disabled while the external view is open.

- **Full Screen:** Switches the external project view between window mode and full screen mode.
- Continue

This button will continue program execution if it was paused because a [break point](#) triggered or an error was detected.
- Pause

Pause program execution even if a Project View is open
- Enable All Breakpoint

Globally enable/disable all active [break points](#). This setting will not affect individual break points that is already disabled.
- Break on Error

Enable/Disable if program execution should be paused when a [chip reports an error](#).
- Screenshot

Takes a screenshot of the [editor view](#) or the [project view](#), whoever currently has the keyboard focus. A dialog box will allow you to save the image to a file.
- Limit Frame Rate

Enable/Disable limitation of the maximum framerate to the monitor's refresh rate (VSync - Often 50-120 Hz). Enabling this (recommended) will reduce system load.
- Help
 - View Help

Opens the user manual.
 - About

About SnaX Developer.

Some of these items are also available as buttons on the main toolbar described in the next section.

Main Toolbar

All buttons on the toolbar as shown in Figure 4 are shortcuts to items in the [main menu](#).

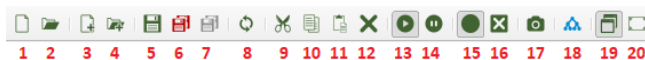


Figure 4 Main toolbar.

1. File -> New Project
2. File -> Open Project
3. File -> Add New Class
4. File -> Add Existing Class
5. File -> Save
6. File -> Save Modified
7. File -> Save All
8. File -> Load all project files
9. Edit -> Cut
10. Edit -> Copy
11. Edit -> Paste
12. Edit -> Delete
13. View -> Continue
14. View -> Pause
15. View -> Enable All Breakpoints
16. View -> Break on Error
17. View -> Screenshot
18. View -> Class Diagram
19. View -> Enable external project view
20. View -> Toggle external project view full screen

Chips Panel

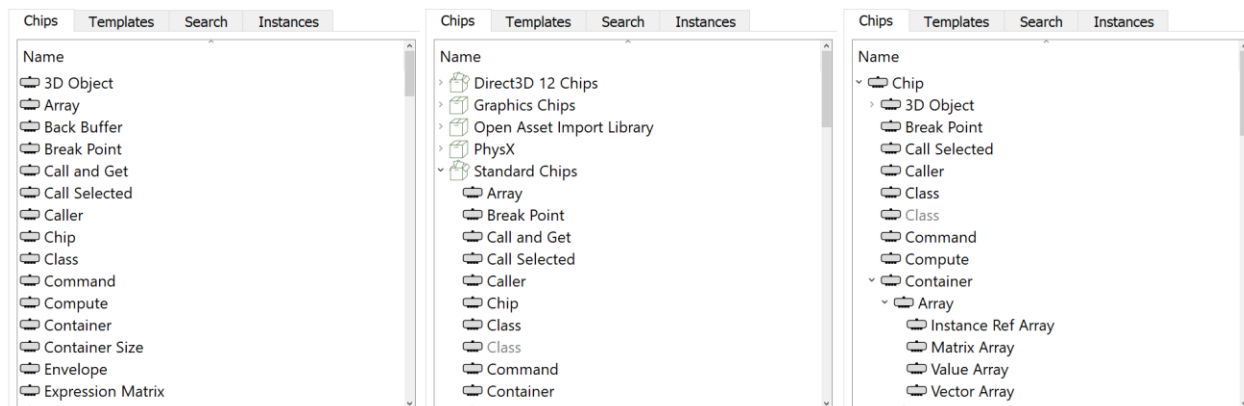


Figure 5 The three different view modes of the chips panel.

The chips panel contains a list of all chips available to the application. You can add a new chip to the class open in the [editor view](#) by pressing an item in the list using your left mouse button, drag it over to the editor view, and drop it at a suitable location where you can link it to the other chips.

The chips panel has three different view modes as shown in Figure 5. By right clicking the panel using your mouse, you can select which mode you want from a popup menu. You can choose between:

1. A plain list, ordered alphabetically, of all chips available for use in the application.
2. A packet list, where chips are grouped into their containing packets. Technically, a packet is a dynamic library (DLL-file in Windows) containing a set of chips. The chips contained in a single packet are often related to each other in some way. For example, are all chips that has to do with Direct3D graphics located in the same packet (Direct3D Chips). Packets are loaded on demand at the time a chip is requested by the application. When a packet is actually loaded into application memory, its icon in the list will change to an “open packet” icon.
3. A tree view showing the inheritance hierarchy between chips. The **Chip** is the root of the tree – All other chips inherit **Chip**, directly or through others. It defines a basic interface that allow the generic framework to interact with any other type of chip. All derived chips add new functionality to cover different purposes.

Note that grey items are unavailable for direct use to the application and cannot be dragged into the editor view.

Template Panel

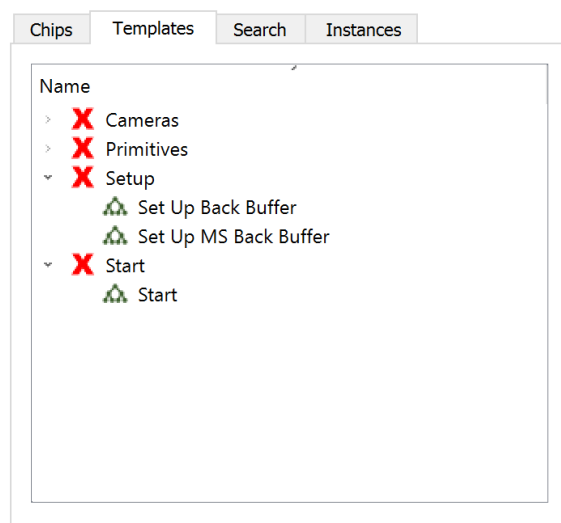


Figure 6 The templates panel.

The template panel, as shown in Figure 6, contains a list of all templates available to the system. A template is basically just the content of a class (chips, connections and background items) that you can drag and drop into the [editor view](#) using your mouse the same way you do from the [chips panel](#). It is a very powerful feature for inserting functionality that is used often, but tedious to recreate every time you have to use it. The list has two modes - One where the templates are groups by their containing documents, and one where documents are hidden. Right click the panel to switch between the two modes from the popup menu.

The templates are stored in the 'Templates'-folder in the Developer's install directory. A template can be created and edited the same way as any other class. You can even create your own templates by adding a document to this directory or to a subdirectory.

Search Panel

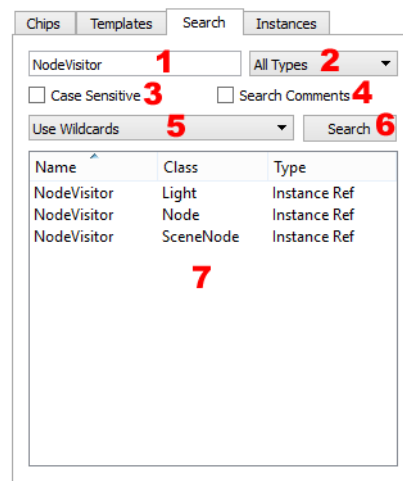


Figure 7 The search tool panel.

The search panel allow you to search the project for a given text element, and return the chips that contains this text in its name or comment field. The panel consist of a few elements as shown in Figure 7:

1. The input field where you enter the text you want to find.
2. You can limit the search to a single type of chip or you can search all types.
3. This checkbox allows you to do a case sensitive search or not.
4. This checkbox can be marked if you want to search the comment field in addition to the name.
5. You can choose between three matching algorithms:
 - **Match exactly:** The text must match the name/comment exactly.
 - **Use Wildcards:** You can use special characters like * and ? to widen the search. For example, 'Foo*' will match both 'Foo' and 'FooBar', while 'Fo?' will match 'Foo' and 'For', but not 'FooBar'.
 - **Use RegExp:** Use the power of regular expressions to do an advanced search. Look up regular expressions on the web for details!
6. Press the 'Search' button to do the search!
7. The search result is a list of chips in the project matching the given options. You can double click an item in the list to focus on the chip in the [editor view](#). In addition, if you right click an item, a popup menu will allow you to open the [chip's property dialog](#) instead. The list contains three columns:
 - **Name:** This is the name of the chip.
 - **Class:** This is the class where the chip is located.
 - **Type:** This is the type of the chip.

Note: Chips that are contained within other chips (for example inside a Class Instance) are also searched. When you double click the resulting item in the list, the (outer) containing chip will have focus in the editor view. In this case, opening the chip's property dialog could be a more useful option.

Instances Panel

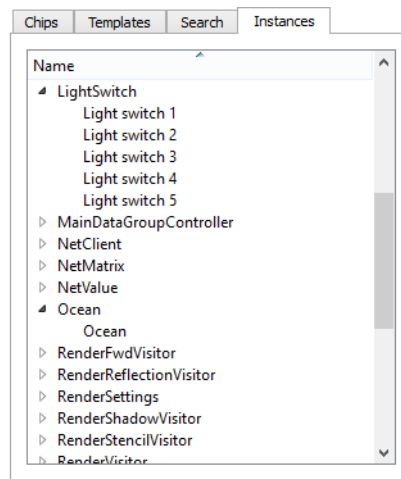


Figure 8 The instances panel.

The instances panel as shown in Figure 8, will list the class instances of your application. The instances are grouped by the class they are instance of. Double clicking a class item will open the class in the [editor view](#), while double clicking an instance item will open the [instance's property dialog](#). The items' label is the name of the instances. Unnamed instances will be labeled 'Unnamed (n)' where 'n' is the unique instance ID. The list has two modes you can toggle between from the context menu. The default mode (Enable Filtering) will only list instances that are a type of the class currently opened in the editor view. The second mode will list all instances living in your application. In a large project, this could be quite many!

Project Tree Panel

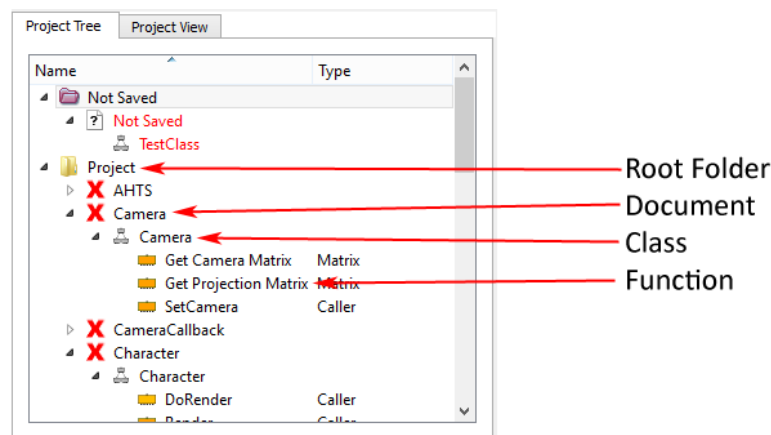


Figure 9 The project tree panel.

The project tree is a very important panel! All documents (files), classes and functions in your project are listed in this panel in a tree like structure as shown in Figure 9.

Folders

At the top level, we have the root folders of your project. The main one, the 'Project' folder, represent the file system directory containing the file (or document as we call it) that has the class where the [start](#)

[chip](#) of your project is located. All loaded documents located in this directory will be listed here, or in subfolders that correspond to the subdirectories of the file system. On the other hand, if you save or load a document in a directory outside the root directory, the document will be listed in the 'External' folder. It is important to remember that such files will not be bundled in the resulting packet when you publish your project! You should for the most part avoid this! A third root folder is the 'Not saved' folder. All documents you create in the Developer will first be located in this folder until you save the file to disk. You should preferably do this as soon as possible to make sure you not lose any work in case there is a power failure! The Developer has a recovery feature for saved documents that regularly takes a snapshot of the files you are currently working on. If the developer shuts down unexpectedly, you will be offered to load the newest snapshots next time you open your project.

Note that documents that are not (yet) loaded by the application are not in the list even if they are located within the application's root directory. Documents are normally only loaded on demand (when requested by your application), but you can use the [main menu](#)'s 'Load all Project Files' to make the system scan all open documents for references to other documents, and open these as well. This is useful if you want to work on parts of your project that is not loaded automatically.

Documents

The second level of items, located inside the folders, are the documents. These are the files containing your classes and are the source code for projects created using SnaX. Similar to text-based programming languages like C++ or java, the source code is what describes your project in a way the computer can understand and compile/execute. A document can contain any number of classes, and a class can be moved from one document to another using a standard drag and drop operation with your mouse. If you right click a document, you get a popup menu with a few options:

1. **New Class:** Adds a new class to this document.
2. **Save:** Similar to the [main menu](#)'s 'Save' item.
3. **Save As:** Similar to the [main menu](#)'s 'Save As' item.
4. **Delete:** Similar to the [main menu](#)'s 'Delete' item.
5. **Hide Documents:** You can choose to hide the document level using this option.

Another thing to note about documents are the file format:

1. **.m3x-files:** These are XML-based human readable files, possibly containing encoded binary data as well. They can be edited manually, but this should be done with great care and avoided in most cases! You should use this file format most of the time!
2. **.m3b-files:** These are binary files not human readable. They have the advantage over the XML-format of being faster to load and save, have a smaller file size, and they can be encrypted. Normally you don't need to save files to this format manually, but when [publishing](#) a project, the documents will by default be converted to an encrypted binary format that can't be opened in the Developer any more. This prevent others from gaining access to your source code by grabbing documents of your published project! A disadvantage of the binary format is that this does not play very well with version control systems. A text-based format is better for this, but care should be taken when trying to merge or resolve conflicts between different versions of a m3x-file. You can easily end up with perfectly valid XML that is still unreadable by SnaX.

One last thing to note about the document items are the text color. The standard color is black, but a red color indicates that the document has been modified in some way, and needs to be saved again to not lose any changes. You can press the “Save Modified” button on the [toolbar](#) to resave all modified documents.

Classes

Classes are grouped under the documents. Your project can contain a great number of classes, many of them small and closely related, and it can be a good idea to reduce the number of documents by collecting related classes into a single document. You can do a few operations on a class item:

1. Double clicking a class will open it in the [editor view](#).
2. A drag-and-drop operation on a class item to the editor view will add a new **Class** chip to the current workspace.
3. Similarly, dragging a class over to another document in the panel will move the class to this document. Remember to save both documents to avoid having a duplicate class next time you open your project!
4. As for the document level, right clicking a class item will open a popup menu. The options are the same as for the document level, but a new item called ‘Rename’ will allow you to rename the class.

Like documents, a class will have a red text color if it has been modified since last save!

Functions

Under the class level, we find the last level of items, that is, the functions that exist in a class. As we know from other programming languages, a class contains functions (methods) and data members. This is also the case for SnaX. A function is a procedure, or piece of code, that can be called from somewhere else in your program using a **Function Call**. The following operations are available for a function item:

1. Double clicking the item will open the containing class in the [editor view](#) and focus will be on the chip representing the function.
2. Dragging an item into the editor view will create a new **Function Call** targeting the function.
3. Holding your mouse over the item will open a tool tip displaying a C-like function signature.

For functions, the Type column shows the type of the chip representing the function. Also, note the item’s icon, which reflects the type of function, *static*, *non-virtual* or *virtual*. Read about [functions](#) in the section describing the [editor view](#) for more information!

Message Log Panel

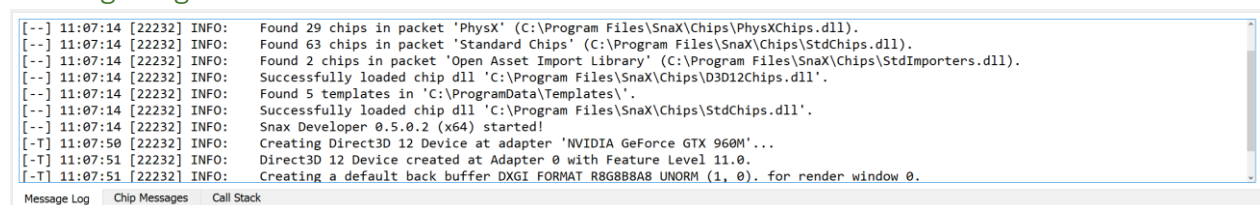


Figure 10 The message log panel.

The message log panel as shown in Figure 10 will display all kinds of status information about what is going on in the application. This is especially useful for debugging purposes! The information displayed here is also written to a log file (in the user's AppData folder) if this is enabled (default). This is useful in the unlikely event of the Developer terminating abnormally. The message log is a plain text view making it easy for you to copy the content for sharing with others. A message is made up of five columns:

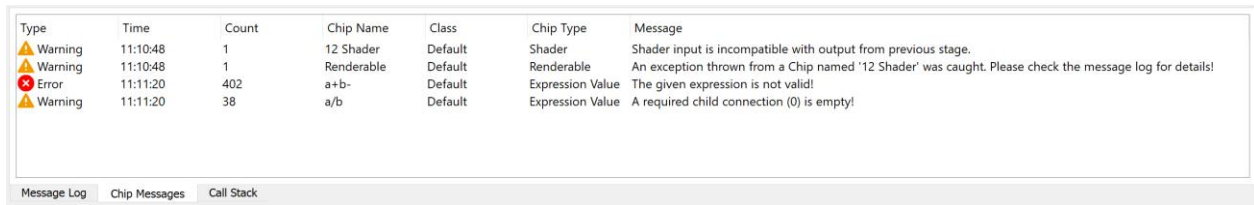
1. The [--] column can contain two letters. A [C-] tells you that a chip is linked to the message. Double clicking the message will focus on the chip in the [editor view](#). A [-T] tells you that a stack trace is available for this message. Right click to open the context menu where you find an item to open a stack trace window. [CT] tells you that both options are available.
2. The second columns show you the local clock time for when the message was generated.
3. The third column will tell you the operating system's internal ID for the thread that generated this message. SnaX uses multiple threads for many things, and the ID column will help you get a better understanding of which messages are related to each other. Often, a problem will generate more than one message, and the ID is very convenient in this case.
4. The forth column shows the message severity. All messages generated are assigned a level of severity to help you better sort out the most important ones. There are five severity levels:
 - a. **DEBUG**: This is just a plain debug message telling you some basic information that can be of value if something unusual happens. It is really only intended for low level debugging.
 - b. **INFO**: This is a message provided for your information. It is usually nothing to worry about, but can be worth looking at when debugging.
 - c. **NOTICE**: Such a message tells something you should take notice of. It is not an error message, but is often a bit more important than the previous level.
 - d. **WARNING**: A message of this level is always worth looking at. It will tell you that something is probably not as it should be, and you may have to take actions to correct it.
 - e. **FATAL**: This type of message tells you that something went wrong, and parts of your application is probably not working as it should. Actions must normally be taken to correct the problem. The real severity of the message can be anything from just a small problem to a fatal application crash, although the latter should be very rare!
5. The last column is the message itself. This is a custom text line.

Right clicking the panel will open a context menu with a few options:

1. **Copy**: Copies any selected text to the clipboard.
2. **Select All**: Selects all text for copying.
3. **Clear All**: Clears the log for all messages.
4. **Focus on Chip**: Focus on the chip in the [editor view](#). Only available for messages linked to a chip.
5. **Open Chip Dialog**: Opens the [chip's property dialog](#). Only available for messages linked to a chip.
6. **Show Call Stack**: Opens a window with the call stack present at the time the message was generated. Only available for messages with a call stack.
7. **Dump Call Stack**: Enable/Disable if the current call stack, if applicable, should be dumped and linked to new messages.
8. **Set Buffer Size**: Allows you to limit the maximum number of messages in the log. Setting a limit will discard old messages as new messages arrives. An extremely long message log may slow down your system.

9. **Filter:** Here you can choose to discard new messages of lower severities to get a smaller message log. It will not affect messages already present, nor messages written to the log file.

Chip Messages



Type	Time	Count	Chip Name	Class	Chip Type	Message
Warning	11:10:48	1	12 Shader	Default	Shader	Shader input is incompatible with output from previous stage.
Warning	11:10:48	1	Renderable	Default	Renderable	An exception thrown from a Chip named '12 Shader' was caught. Please check the message log for details!
Error	11:11:20	402	a+b-	Default	Expression Value	The given expression is not valid!
Warning	11:11:20	38	a/b	Default	Expression Value	A required child connection (0) is empty!

Message Log Chip Messages Call Stack

Figure 11 The chip messages panel.

This panel shown in Figure 11 is somewhat related the [message log](#), but there are some important differences! All chips are able to flag various issues using a standard reporting system. When an issue has been flagged, it will in most cases remain until you actively acknowledge (remove) it. The chip messages panel will list all currently active issues. The list has seven columns:

1. **Type:** This is the severity level of the issue. This correspond to the levels reported in the [message log](#) panel.
2. **Time:** This is the local clock time for when the issue was *last* reported.
3. **Count:** This is the number of times the issue has been reported. If you remove the issue, the counter will be cleared next time the message appears.
4. **Chip Name:** This is the name of the chip reporting the issue.
5. **Class:** This is the class containing the chip.
6. **Chip Type:** This is the type of the chip.
7. **Message:** This is a message describing the issue.

The big difference between this panel and the [message log](#) is that this panel is constantly updated. When an issue is reported for the first time, a message is added to the log, but no more information about the current status can be found. A repeating issue will not be added to the message log more than once to avoid flooding, but our 'Count' column will keep increasing. If you acknowledge an issue, it will be dumped to the message log again if it reappears.

Double clicking an issue will focus on the chip in the [editor view](#). A chip contained in another chip will focus on the (outermost) containing chip. Right clicking an item will open a context menu with the following options:

1. **Focus on Chip:** Same as double clicking.
2. **Open Chip Dialog:** Open the chip's [property dialog](#).
3. **Acknowledge Selected Messages:** Remove all *selected* messages from the list.
4. **Acknowledge All Messages:** Remove all messages from the list.

Call Stack

Frame	Depth	Instance	Class	Chip
20	9	175 - Ocean	Ocean	Break Point
19	9	175 - Ocean	Ocean	Render
18	8	72 - Unnamed	RenderVisitor	Node->Render
17	8	72 - Unnamed	RenderVisitor	ApplyNode
16	7	175 - Ocean	Node	NodeVisitor->ApplyNode
15	7	175 - Ocean	Node	Accept
14	6	108 - Root Node	SceneNode	Node->Accept
13	6	108 - Root Node	SceneNode	Traverse
12	5	72 - Unnamed	NodeVisitor	Node->Traverse
11	5	72 - Unnamed	NodeVisitor	Traverse
10	4	108 - Root Node	Node	NodeVisitor->Traverse

Figure 12 The call stack panel.

The call stack panel as shown in Figure 12 is an extremely useful panel when debugging your application. It is used together with [break points](#), or when the 'Break on Error' option is active. When program execution is paused, either when a break point triggered or when a chip reports an error, the current call stack is dumped to this panel. You will be able to see the program execution path, from the [start chip](#), through all function calls, until the interrupting chip. Note that only the execution path through functions and parameters are listed – not the path through individual chips! The call stack is very similar to what you will find in the debugger of other programming environments for languages like C++ or Java.

The panel has five columns:

1. **Frame**: This is the index of this entry in the stack. Starting at zero for the start chip.
2. **Depth**: When a function is called, the depth is *increased* by one. When a parameter is called, the depth is *decreased* by one.
3. **Instance**: This is the class instance currently active for this frame.
4. **Class**: This is the class where the targeted chip is located.
5. **Chip**: This is the chip that is the target for this frame. Its icon will tell you if this is a function call, a function, a parameter, a plain chip or a break point. It corresponds to colors found for these types of chips in the [editor view](#).

Double clicking a frame will focus on the targeted chip in the editor view. Right clicking will open a context menu with two options – One to open the [instance's property dialog](#), and one to open the [chip's property dialog](#).

Project View

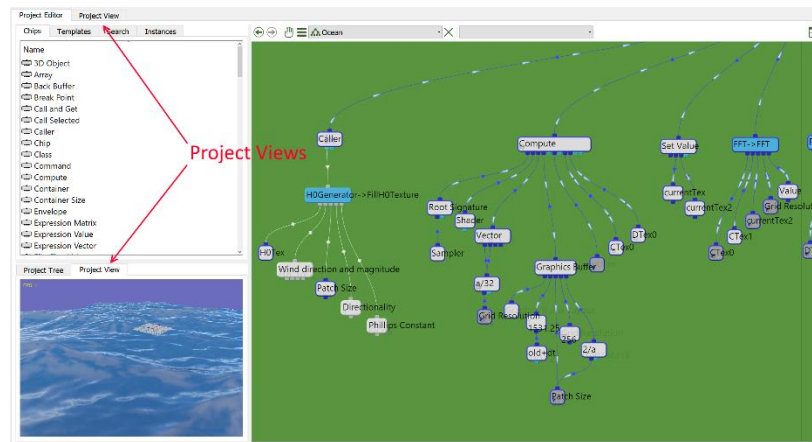


Figure 13 The two project view panels.

One of the great strengths of SnaX is the possibility to see your application running, as you develop it! You can see the effect of everything you do, when you do it! There are three windows available for this. A small one is located down left, sharing space with the [project tree](#) panel. Using this window allows you to see your project running while working in the [editor view](#). Note that the four main elements of the main workspace can be resized using your mouse to get a best composition of the available space. The second project view is sharing space with the main workspace of the Developer application. When you open this window by selecting the 'Project View' tab on top of the workspace, you will not be able to do most of the editing work as the editor view is hidden, but you can still use the various [property dialogs](#) for chips and instances. A third alternative is the external project view. This is a resizable window that can be positioned freely anywhere on your desktop. This is ideal for multi-monitor setups because it enables you to fully utilize the available screen space. The external view can even be toggled full screen, though only at the current desktop resolution. Like for the small project view, you can continue to work in the editor view while your project keeps running! You can pause program execution using the main menu/toolbar, or use the keyboard shortcuts Pause/F5 to switch between the running and paused states.

Editor Panel

The main portion of the Developer's window area is the editor panel(s). An editor panel consist of a toolbar and a viewport which contains either an [editor view](#) with the view of a class, or an embedded chip [property dialog](#). Let's call those for pages. The panel can be split either horizontally or vertically. First, let's focus on the toolbar above the main viewport. This contains a few buttons and menus it is worth having a look at.

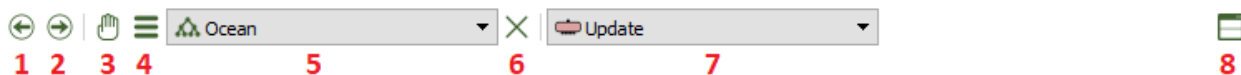


Figure 14 Toolbar for the editor view.

As shown in Figure 14 the toolbar contains these items:

1. Go back: Goes back to a previous view in the editor.
2. Go forward: Goes forward to a previous view in the editor.
3. Move editor: Press and drag using your mouse to move the current page to another panel.
4. Drop-down menu with the following elements:
 - a. Save: Saves the document containing the current class.
 - b. Close: Closes the current page.
 - c. Close All: Close all stacked (open) pages in this panel.
 - d. Close All But This: Close all pages in this page, except the on currently visible.
 - e. Open Containing Folder: Opens a file browser in the location of the current document.
 - f. Add Class: Adds a new class to the current project, and opens it in this panel.
 - g. Import Class: Imports an existing class to the project, and opens it in this panel.
5. List of all pages (classes or dialogs) that are open (stacked) in this panel.
6. Close: Close the current page.
7. The list of the current class' functions, or the tabs of a chip dialog. Select one of the items to move the view to this area.
8. Split panel: This allows you to split the editor panel either vertically or horizontally, creating two new panels in the same area, initially with the same pages. An example of this is seen in Figure 15 where the editor panel is split horizontally. The currently active panel is indicated with a yellow menu line. Any classes you open, or property dialogs you embed will be opened in the active panel. The panels can be resized by dragging the line separating the two views. When splitted, the menu contains another button to close the panel.

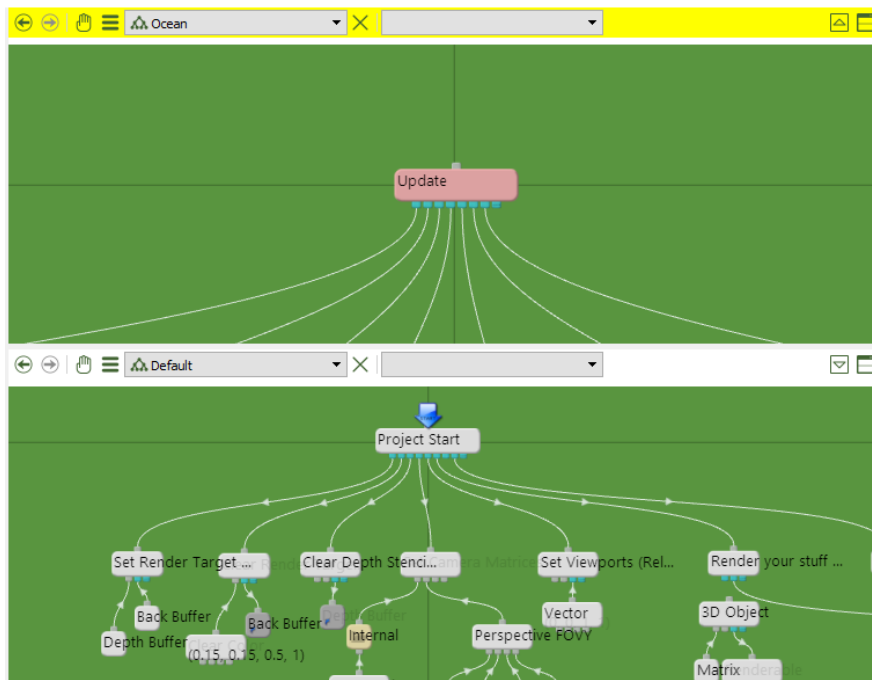


Figure 15 Splitted editor panels.

Editor View

The editor view is where you will spend most of your time creating an application using SnaX Developer! It can be compared to the text editor in development tools for other text-based programming languages. The big difference though, is that you will not be writing any code, but doing visual programming mainly using your mouse! The editor view is a very powerful development tool, and it is packed with features to help you develop your new 3D-application!

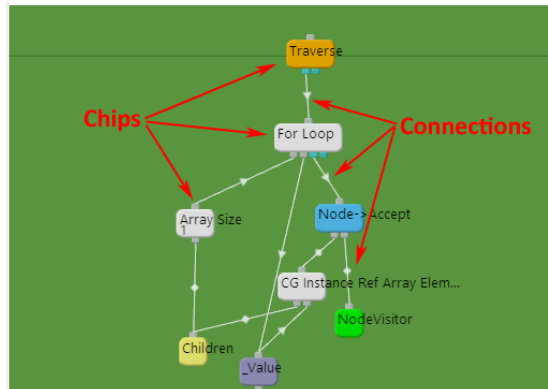


Figure 16 The editor view.

Let us have a look at the main elements of the editor view. The basics are quite simple actually! The editor view as shown in Figure 16 is a view of a class. This is comparable to a text editor being a view of a text document. A class primarily consists of two different elements: Chips and the connections between them. In addition, we can have a few support elements like folders, text elements and colored background rectangles. These will help you organize and document your code. Let's have a look at the different elements you will encounter in the editor view. Some details may be a bit difficult to grasp without understanding the big picture, and this section is not intended to be a complete description of the programming model, but reading the [Programming Guide](#) will hopefully help clear things up!

Chips

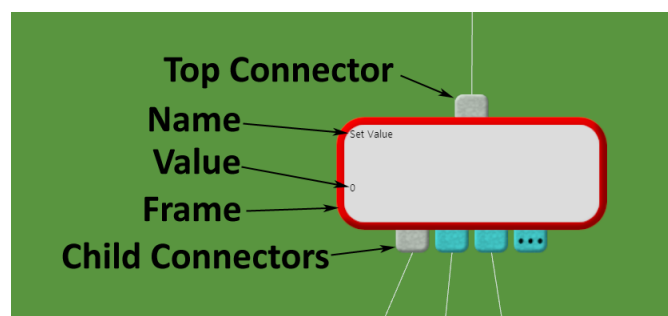


Figure 17 The different elements of a chip.

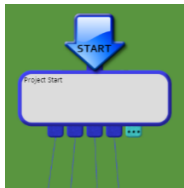
A chip consists of a few basic elements as shown in Figure 17:

- **Top Connector:** Every chip has a top connector that can be connected to a compatible child connector of another chip. The top connector can be connected to several child connectors at once. A chip does not know anything about the chips connected to its top connector.

- **Child Connectors:** A chip can have any number of child connectors ranging from zero to as many as it needs. The child connectors are the inputs for which the chip can perform operations, query data etc. The program flow goes from top to bottom in a tree of chips! There are two types of child connectors. The first type is the fixed one (gray), that can connect to exactly one compatible top connector of another chip. The second type is a growing connector (blue), which allows for an unlimited number of sub-connectors. When you link a chip to the last sub-connector (marked '...'), another sub-connector is added to the end. Each sub-connector can connect to exactly one compatible top connector of another chip. The chip determines the types of its own child connectors.
- **Body:** The body of a chip is used to visualize different states using colors and icons. Double clicking the chip will open its [property dialog](#).
- **Frame:** The frame is used to visualize a couple of states. A red border indicates that the chip is selected (like selecting text in a text editor). A blue border indicates that the chip is currently active within the execution of your application.
- **Name:** All chips can have a name, and this is written on the top left corner of the chip.
- **Value:** Some chips contains a state that it is natural to write below the name. The **Value** chip will for example write out the internal number it currently holds.

A chip can have a number of properties that will decide its body's color and icons. There are also a few type of chips of such importance they are given individual background colors to help identify them. The following sections will describe the meaning of the different icons and background colors a chip can have.

The start chip



The start chip is where program execution begins. This chip is *called* at the beginning of every frame of your application, allowing the connected program logic to run and eventually draw something cool to the screen. This typically happens 30-60 times a second, but it could be much more, or even less depending on your computer and the complexity of your program. You can select the start chip you want to use by right clicking a chip and select the 'Set as Start Chip' option from the context menu.

Refresh Mode

A chip's refresh mode decides how often a chip will recalculate or update itself when engaged. The exact meaning or behavior depends on the type of chip. An **Expression Value** will for example only calculate a new value if it passes the refresh check. If it does not pass, it will return the previously calculated value, saving valuable CPU time for the rest of your program. A **Caller** will only call its children if it passes the check.

You set the refresh mode for a chip in the **Chip**-page of its [property dialog](#). You can also select the chip in the editor and press Ctrl-R to cycle through the available refresh modes.

There are four refresh modes available:



Always

The chip will always recalculate. Use this setting inside a **For Loop** to make chips recalculate/update themselves for each iteration of the loop.

**Once per function**

This is the default setting. The chip will recalculate once for each function call.

Once per Frame

The chip will only recalculate once per application frame - That is, once every time the system has called the start chip.

Once

The chip will only recalculate the first time it is engaged. Use this setting for things that only needs to be done one time in the application, for example initialization code.

Functions

A *function*, or more correct a *function interface* or *declaration*, is a special property given to a chip making it remotely accessible by a **Function Call**. Normally, a chip is only accessible to other chips by linking them using the connectors, but a **Function Call** has the special property of being able to connect to a function remotely. Setting the function is done in the [chip's property dialog](#). There are three type of functions, given three different colors for the chip:

**Static Functions**

Static functions are called on the class itself. It does not require an instance.

Nonvirtual Functions

Nonvirtual functions are called on an instance that is a type of the class where the function is defined. They cannot be overridden in deriving classes.

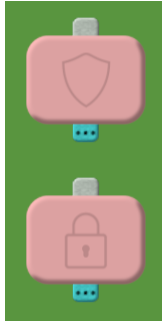
Virtual Functions

Virtual functions are like nonvirtual functions, except that they can be overridden by deriving classes.

Another property related to functions are the access modifiers that decide from where a function is accessible by a **Function Call**. The access modifier is set together with the function in the chip's property dialog. The current access modifier is visualized using icons on the chip's body.

**Public**

Function Calls located in any class throughout the entire project can call the function. There is no icon for this type!

**Protected**

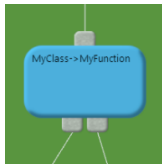
Only **Function Calls** located in the same class or in a derived class as the function may call it.

Private

Only **Function Calls** located in the very same class as the function may call it.

Chips Related to the Core Programming Model

As explained, your program is constructed on workspaces called classes. A class, as known from many other programming languages, consists of *data members* and *functions*, and the functions themselves are built up of program logic including *local data members* and *function calls*. We have already seen that a function's interface is implemented as a property given to any type (with a few exceptions) of chip. The rest of these elements are implemented as their own type of chip, and are so fundamental that they are given their own colors to help identify them. These are:

**Function Call**

Calling functions is the job for the **Function Call** chip. It will take the type of the connected function, and all parameters will be listed as child connections. The first child connection is reserved for the class instance we should call the function upon. It will be empty for a static function. **Functions Calls** are easiest created by dragging them in from the [project tree](#) panel!

**Parameter**

A function's parameters are implemented as the **Parameter** chip. This chip can take any type and is easiest created by first adding a chip of the type you want, then convert it (right click chip, then go to the 'Convert to' item in the context menu) to a **Parameter**. Selecting which parameters to use for a function is done in the function-chip's [property dialog page](#). When linked as a child somewhere within the program logic of a function, a **Parameter** will act as a physical substitute for the chip linked to the corresponding parameter-connector of the **Function Call**.

**Instance Data**

Nonstatic data members of a class are accessed through the **Instance Data** chip. All "physical" chips located in a class are in fact static data members. This also includes **Instance Data** chips, but they are in fact only a physical substitute for a chip located within a class instance. This internal chip is the actual instance data! You can easily create an **Instance Data** by first adding a chip of the type you want, and then convert it using the chip's context menu as explained for the **Parameter**.

**Function Data**

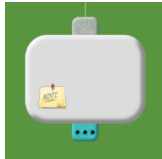
Local data within a function are implemented using the **Function Data** chip. This chip is quite equal to the **Instance Data**, but the chip it is a substitute for, are located on the function call stack. Compared to other programming languages, this should be quite familiar! Calling a function will instantiate the internal function data needed for the lifetime of the function call. For performance reasons, try not to go overboard with the number of these type of chips!



Instance THIS

The **Instance THIS** chip is a reference to the current class instance. When a nonstatic function is called, the class instance for which the function is called upon can be accessed through this chip. If calling a static function, this reference is just empty.

Comments



All chips have a comment field that is useful for documentation purposes. It can be accessed from the 'Comments' page in the [chip's property dialog](#). Chips that has a nonempty comment field will have a small note-icon displayed down left.

Issues



As described in the section about the [chip messages](#) panel, all chips are able to raise various issues using a standard reporting system. When a chip currently has an issue, a small icon is displayed down right. You can acknowledge (remove) the issue by pressing the 'A' key and click the chip using your left mouse button.

Shortcuts

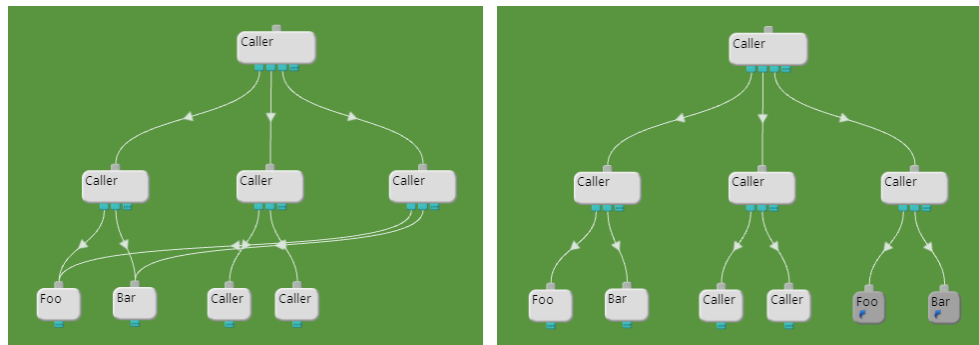


Figure 18 The benefits of using Shortcuts.

A **Shortcut** is a special type of chip that can act as a substitute for another chip when connecting it to a child connector. This is purely an aid in the development process to help you create cleaner and more readable code. Consider the examples in Figure 18. The two examples do exactly the same thing, but the rightmost example using shortcuts is much cleaner and easier to read. A **Shortcut** have a bit darker background color and a little arrow icon down left. You can create a **Shortcut** by selecting a chip, then press the 'S' key while positioning the cursor at the location where you want the shortcut to be created. You can iterate the shortcuts of a chip by selecting it, then press the 'N' or 'P' keys to go to the next or previous shortcut respectively. When you hold your mouse cursor over a **Shortcut**, an orange colored "virtual" connection is shown between the original chip's top connector and the child connectors the **Shortcut** is connected to. Likewise, if you hold the mouse cursor over a chip that has shortcuts, "virtual" connections are shown between the top connector and all child connectors the shortcuts are connected to. This helps you better understand where a chip is actually connected, shortcuts taken into account!

Break Points

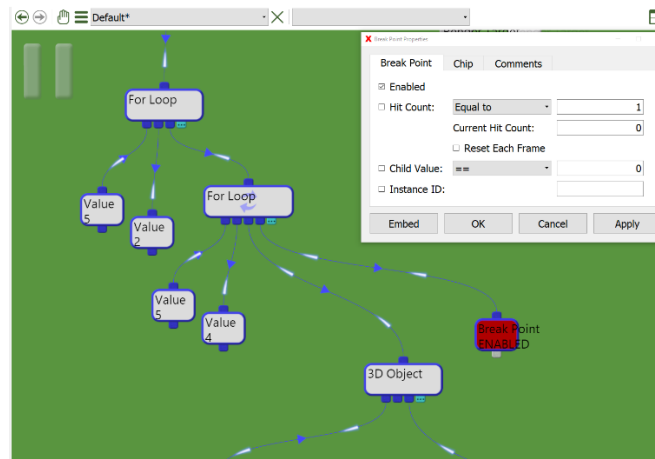


Figure 19 Example demonstrating the use of Break Points.

The last chip we are going to look at in this section is the **Break Point**. This highly useful chip implements a feature known from many other programming languages! It enables you to interrupt program execution at any given point in your code. This lets you examine the current state of your program, like chip values and the [call stack](#) right in the middle of a frame! You can even modify your program before continuing normal program execution, a feature known as *edit-and-continue*. Figure 19 illustrates how break points work. When the **Break Point** (dark red chip) is reached in the inner loop, program execution will be paused. You can configure the break point to only break when certain criterias are met, like when the break point has been called a certain number of times, for example. A “pause” icon is displayed in the upper left corner of the editor view while in the paused state. You can continue program execution by pressing the ‘Continue’ button on the [main toolbar](#). A **Break Point** is like any other chip found in the [list of chips](#)! Note that nothing of what you have rendered so far is actually drawn to the screen while in this state. This is because DirectX collect all render commands to an internal queue, and it does not flush this queue to the GPU for processing until the end of the frame. The result you see on the screen is actually delayed several frames from what your program is currently doing. This has to do with the asynchronous nature of the GPU.

Tooltips and Performance Monitoring

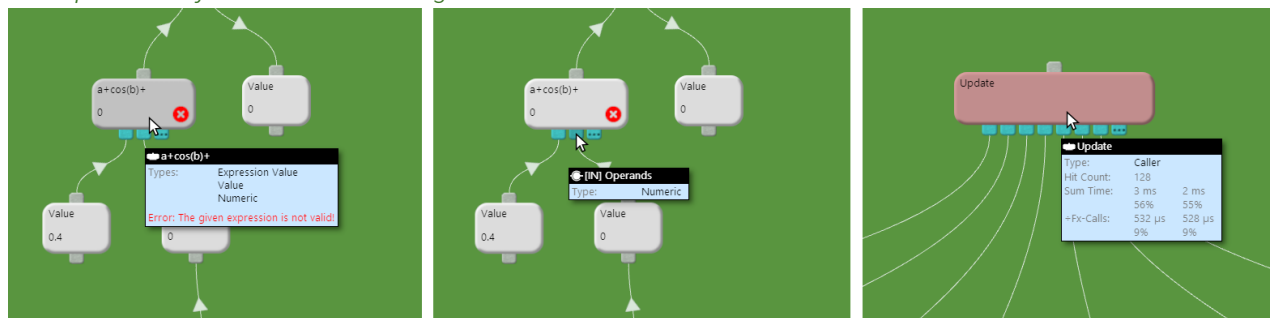


Figure 20 Tooltips for chips, child connectors and functions with profiling enabled.

Holding your cursor over a chip or connector will display a tooltip-style information box as shown in Figure 20. For a chip (or top connector) as shown to the left, this box contains:

- **Name:** This is the name given to the chip.
- **Types:** These are all the types the chip implement. The most specialized type is on top. The basic **Chip** type is omitted!
- **Error:** If the chip is reporting a problem, a short text will describe the problem(s) here.
- **Comment:** At the bottom, the chip's comment, if any, is written. It is truncated if too long!

As shown in the center of Figure 20, the box for a child connector contains:

- **Name:** A name that in most cases gives a hint about the purpose of the connector.
- **Type:** This is the type accepted by the connector. A chip must implement this type to be able to connect!

For functions, when profiling is enabled as shown to the right in the figure, information in addition to that for any chip, includes:

- **Hit Count:** This is the number of times the function was called.
- **Sum Time:** This is the total time (in time and percent) spent in the function, including time spent in sub-functions. There are two columns, where the leftmost is the average time spend in the function each time it was called. The right column is the average time spent in the function per frame.
- **-Fx-calls:** This is equal to the previous row, except that the time spent in sub-functions (functions called within this function) is excluded.

Connections

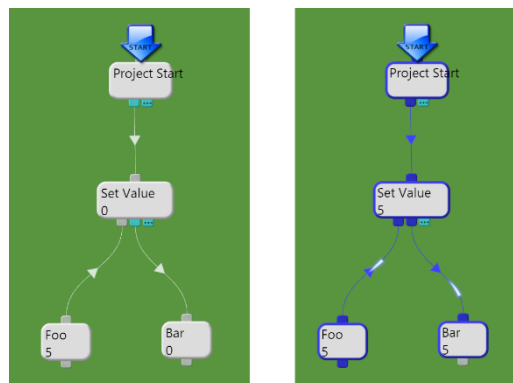


Figure 21 Chips and connections in a passive and active state.

In addition to chips, we have the connections between them! Those are what binds together individual islands of functionality to build a complex program structure. A connection always goes between the child connector of a chip to the top connector of another compatible chip as shown in Figure 21. In the middle of all connections are a small arrow indicating the logical direction of data or program flow. The figure gives an excellent example of this. The **Set Value** will ask the first **Value** (Foo) for its internal number and assign this to the second **Value** (Bar). The arrows indicate the data direction! There is also a third symbol with arrows in both directions indicating a bidirectional, unknown or undefined data/program flow. When running your application in the project view, *live* connections are given a blue color. A moving 'particle effect' will also help identify the live parts of your program. A connection is *live* when a chip queries a child connector for the connected chip. The connector itself will also turn blue,

even if there is no connection established. This can help you locate connectors you might have forgotten to connect, though this is most often reported by chips by a “Missing Child” message.

You can create a connection by pressing a connector using your left mouse button, then drag the new connection over to the connector you want to connect, and release the mouse button. After pressing a connector, compatible connectors turn green, while connectors you cannot connect to turns red. Dragging from a top connector to an occupied child connector will remove the old connection from it. If you release the mouse button on the chip itself, the connection will be set at the first free and compatible child connector on a left to right search. There are also some helpful operations that are good to know about:

- Holding down the ‘Shift’ key while placing a connection on a growing connector, will shift all subsequent connections to the right, placing the new connection where you wanted without disconnecting anything.
- Holding down the ‘Shift’ key while pressing your mouse button on an occupied child connector will allow you to move the connection to another connector. If the new connector is occupied, the connections will be swapped if possible, based on compatibility.
- Holding down the ‘Ctrl’ key while pressing your mouse button on an occupied child connector, is the same as dragging a new connection from the corresponding top connector.
- Dragging and releasing a connection from a child connector to an empty space in the class will open a popup menu listing all compatible chips for this connector. Selecting an item from the list will create a new chip of this type at the given location, and connect it to the child connector.
- If multiple chips are selected, dragging a connection from the top connector of a selected chip will try to create connections for all selected chips to the chip or child connector for which you release the mouse button upon. The connections will be set to the first free and compatible child connectors on a left to right search.
- Holding down the ‘L’ key while pressing your left mouse button on a chip, will try to create connections from the top connector of all selected chips. The connections will be set to the first free and compatible child connectors on a left to right search.
- Holding down the ‘K’ key while pressing your left mouse button on a chip will remove all empty sub-connectors.

Folders

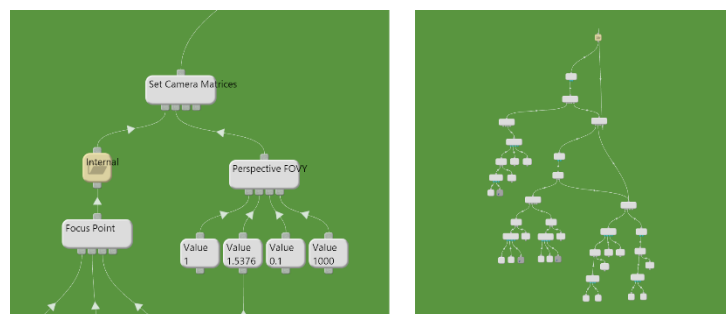


Figure 22 A folder and its internal content shown to the right.

A folder is an item used for grouping together pieces of logic that naturally belongs together, but often is too complex for you to want visible at all times. You can then hide them inside a folder as shown in Figure 22. Double clicking the folder will bring the view to inside the folder, as shown to the right. A

corresponding folder item lets you jump back out again. You can create a folder by selecting the items (chips, folders and background items) you want to move inside the new folder, and then press the 'Space' key. This will create the new folder on the location of the cursor. If you press the 'Space' key while only a single folder is selected, the content will be unpacked and the folder removed. You can also drag and drop items onto a folder to move them into there. A folder can be renamed by right clicking and selecting 'Rename' from the context menu!

Background items

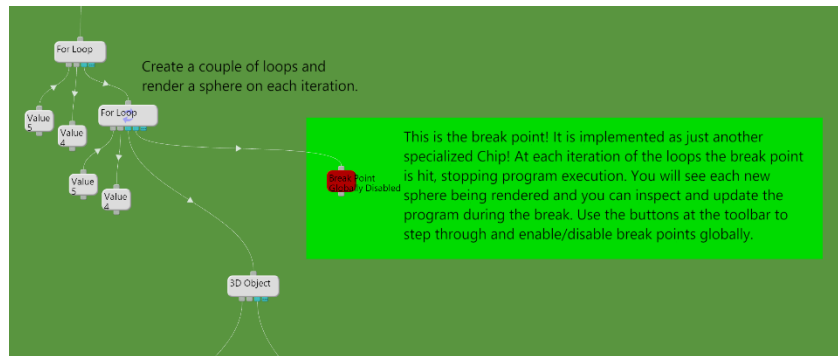


Figure 23 Background items like text and rectangles are useful for documentation purposes.

There are three types of background items that can be added to a class, namely text items, colored rectangles and images. These items are meant for documentation purposes and for making your code easier to read and understand. An example using background items are shown in Figure 23. Together with the comment field available for all chips, the background items are the tools for documenting your code. You can create a new background item by right clicking an empty space in your class, then select 'Add' from the context menu:

- **Rectangle:** Adding a rectangle will first let you select a color from a dialog box. You can move the rectangle around by placing your mouse cursor in the upper left corner of the rectangle, and then drag it using your left mouse button. The rectangle can be resized using a similar operation on the lower right corner. Holding down the 'Shift' key let you maintain the aspect ratio. Double clicking the upper left corner will open the color dialog again! The corner areas are also the selection areas for the items. A selected item will have a red border.
- **Image:** An image is very similar to a rectangle, but when adding an image, a file dialog will let you select an image file from disk instead of a color. It is moved, resized and changed the same way as a rectangle item!
- **Text:** Adding a text item is done the same way as for rectangles and images. A dialog box will let you enter your text, select color and text size. It is moved, resized and changed the same way as a rectangle and image item!

Navigation and Common Operations

Navigating around a class is done using your mouse:

- Hold down the middle or right button and move the mouse around to move (pan) the view around in the class.
- Scrolling your mouse wheel will zoom in and out of the view. The zoom operation is centered on the location of the cursor. Alternatively, press ALT + right mouse button and move the mouse to zoom.

A few operations are common to all items in a class:

- **Selecting items:** Any item can be selected by simply pressing your left mouse button on it. Any previously selected items will be unselected unless the given item is already selected, or if you hold down the 'Ctrl' key. The latter will unselect the given item, if it was already selected. Holding down the 'Shift' key while pressing a chip will also select all chips connected as children, grandchildren and so on.
- **Rubber band selection:** Press an empty space inside your class using your left mouse button, and drag the mouse around to select multiple items at once. Holding down the 'Ctrl' key while pressing the mouse button, will inverse the selection mode of items getting inside the rubber band.
- **Move items:** An item can be moved around by simply pressing your left mouse button on top of it, and move your mouse. Pressing an item already selected will also move all other selected items around.
- **Single deletion:** Hold down the 'D' key and press an item to remove it. No questions are asked, so be careful!

A few common editing operations known from any text editor etc. is also available. In addition to the standard keyboard shortcuts, they can be found in the [main toolbar](#) or in the context menu (right clicking) as well!

- **Cut:** Pressing 'Ctrl-X' will copy all selected items to the clipboard before the items are removed from the class.
- **Copy:** Pressing 'Ctrl-C' will copy all selected items to the clipboard.
- **Paste:** Pressing 'Ctrl-V' will paste all previously copied item from the clipboard to the current class. The new items are centered around the current cursor location.
- **Delete:** Pressing the 'Delete' key will delete all selected items from the class. A dialog box will confirm your operation!

Class Diagram

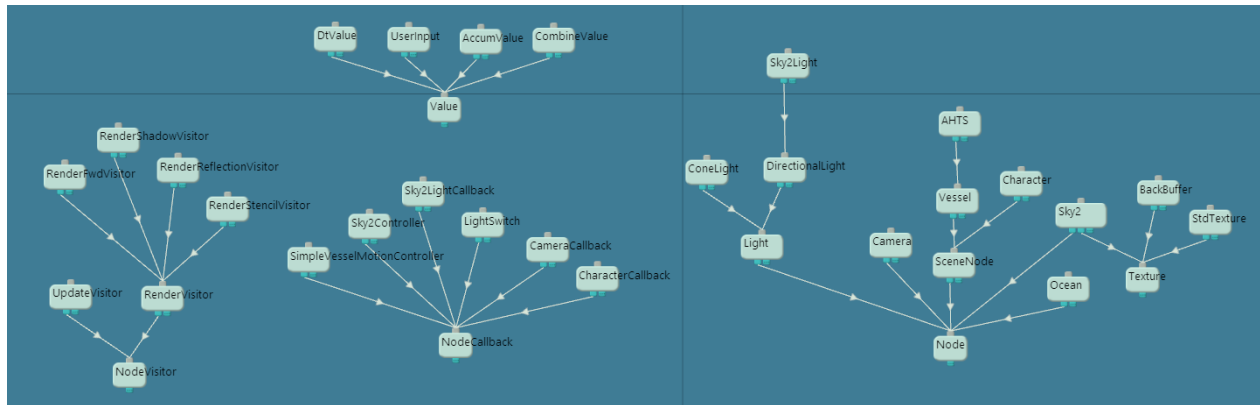


Figure 24 The class diagram.

The class diagram is a very important part of SnaX Developer! It is accessed from the [main menu](#) or [toolbar](#), and is opened in the [editor view](#) as any other class. Similar to a class it contains chips and connections between them. The big difference here though, is that it is not possible to add chips as normally done for a class. The chips in the class diagram represents classes, and the connections between them defines inheritance relationships. All classes in your project are automatically added to the class diagram. All chips have one growing child connector where you can connect base classes. It not the intent to explain object-oriented programming here, but a class can extend or inherit other classes to build upon the existing functionality these classes have to offer. You can add multiple base classes to a single class, a feature known as *multiple inheritance*. You can also build deep class hierarchies of many levels. An example of a class diagram is shown in Figure 24. Compared to an UML-diagram, the class diagram of SnaX is turned upside down! Double clicking a class will open it in the editor view.

Property Dialogs

All chips and class instances have a property dialog you can access to tweak various setting.

Chip Property Dialogs

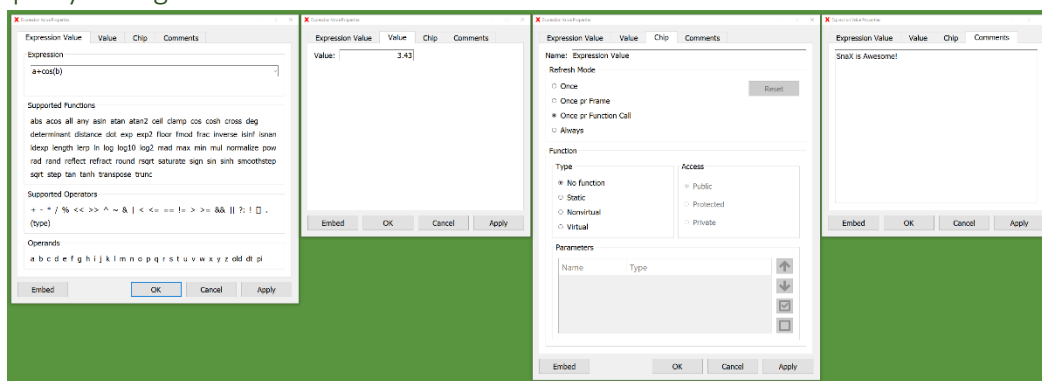


Figure 25 The four pages of the Expression Value's property dialog.

A chip's property dialog consists of a number of pages, one for each type the chip implement, plus a comment page. There are a few exceptions because some types do not implement a property page, while other types will remove any inherited pages. The most common situation is for all chips to have at least the [Chip's](#) property page. As seen in Figure 25 the [Expression Value](#) have its own page in

[illegible]

Another interesting feature of the chip property dialogs is the Embed/Release button. This will embed the dialog into the current [editor panel](#) like any other [class view](#). This is especially useful for advanced/large property pages like the [Shader's](#) page as shown in Figure 26. Embedding it into an editor panel, perhaps sharing space with an ordinary class view helps keep your workspace clean and easy to navigate.

MySceneNode

Type: SceneNode

Name: MySceneNode ID: 433

	Name	Type	Value
1	SceneNode::Children	Instance Ref ...	
2	Node::GlobalTransform	Matrix	
3	Node::ParentNode	Instance Ref	
4	Node::Transform	Instance Ref	
5	Node::UpdateCallback	Instance Ref	

☒ Hide _prefixed Data Members

OK

Each class instance has its own property dialog giving information about the instance as shown in Figure 27. The different elements are:

- 32

- **ID:** Each instance is given a unique ID for the lifetime of the application process. The ID can not be changed.
- **List of data members:** A three-column list shows all data members for this instance. Each data member is in fact a chip located within the instance, and is accessed by your application through the **Instance Data** chip. There is one data member for each **Instance Data**! The three columns are:
 - o **Name:** This is the name of the data member, also known as the name of the corresponding **Instance Data**.
 - o **Type:** This is the type of the chip this data member is of.
 - o **Value:** The current value for this member. For some types (**Value**, **Vector** and **Text**) this column shows the value directly as a text string. For other types, it's just an icon. Double clicking the element will bring up the member's **property dialog** where it can be configured.
- **Hide _prefixed Data Members:** Often there is a need for many “utility” instance data members in a class that often does not need to be edited directly. These can be given a name starting with ‘_’ as they will by default be hidden from the data member list. The checkbox can toggle this behavior.

There are also buttons to bring up the property dialog of the chip containing the instance, copy the instance reference and delete the instance. Note that the containing chip's property page also implement functionality common to that of the class instance' page. **Instance Ref** and **Instance Ref Array** are the most common such chips.

Publish Wizard

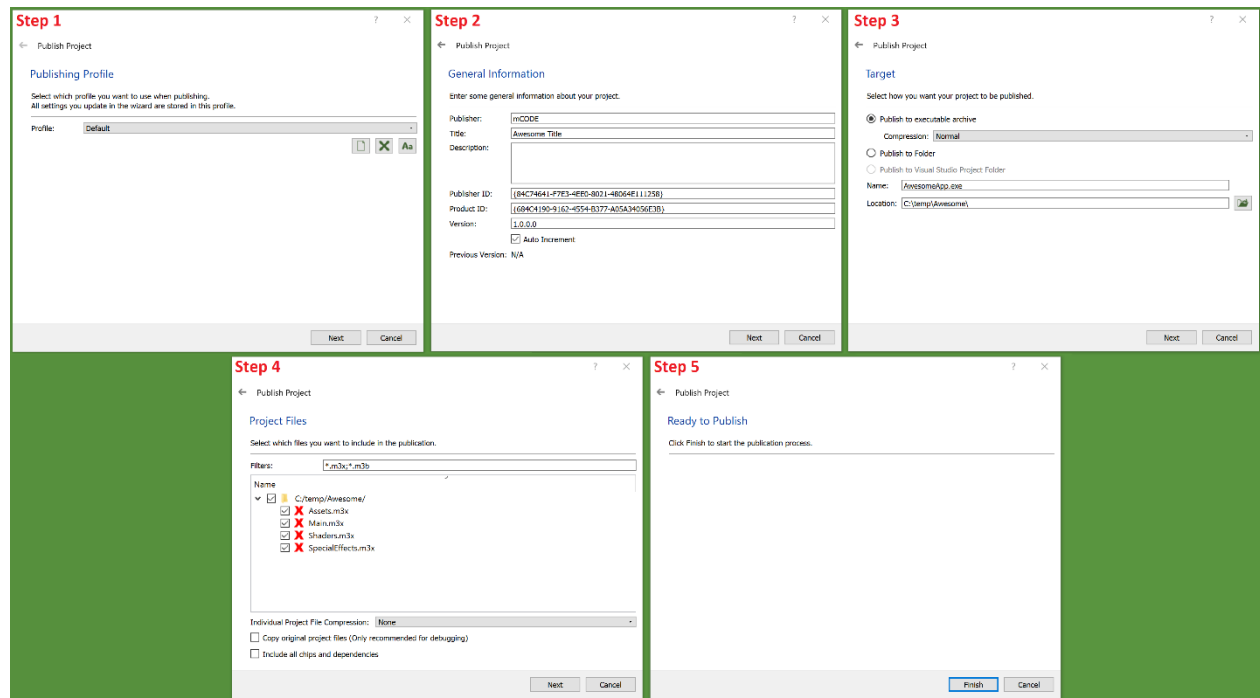


Figure 28 The five steps of the publish wizard.

The final steps of making your application ready for use is the publishing process. This is where all your project files are bundled together with the binaries in an appropriate format. This is a pretty simple step by step process made possible by the publish wizard.

Go to the [main menu](#) and select 'File -> Publish' to open the wizard. The process to publish your project are the following five steps, illustrated in Figure 28:

Step 1

The first step is to select a profile. A profile keeps track of all configurations you make in the following steps. Next time you open the wizard you can use the same profile to avoid having to redo all settings.

The profiles are stored in the project's main document where the [start chip](#) for your project is located. When you have published your project, you should save this document to preserve your settings for next time!

You can add, remove and rename profiles in the first step.

Step 2

The second step is to enter some general information about your project. This includes:

1. Name of the publisher.
2. Title of the application.
3. A short description of your application.
4. A unique publisher ID.
5. A unique product ID.
6. A version tag for your application.

It's up to the targeted platform how (if) the information is used (Some of these are reserved for future use when multiple platforms are supported).

Step 3

Step 3 is to select a file name, target location and format for your application. In general, the alternatives are:

1. **Publish to an (executable) archive:** This option will create a self-extracting executable archive. When starting your application, the archive will extract your files to a temporary directory and run the application for there. This is perfect for small applications as there will only be one file to distribute. You can also set the compression level for your archive. Usually, the 'Normal' setting is good enough.
2. **Publish to a folder:** This option will write all your files to a file system directory of your choice. This is useful if you for example want to create a custom installer for your application.

Step 4

The forth step is to select which project files you want to include with your application. There are a few options worth describing:

1. **Individual Project File Compression:** The default behavior is for the publisher to convert all your project files to a binary format. These files can be compressed individually. Whether or not to

compress the files is for you to decide. Compressing will give a smaller application size, but each file will take slightly longer to load.

2. **Copy original project files:** This option will not convert any of your project files to a binary format, but rather copy the original files directly. This is only recommended for debugging purposes, as your project files will not be encrypted. Anyone would be able to open your source code in the SnaX Developer, and you will probably not want that to be possible!
3. **Include all chips and dependencies:** This option will include all files (binaries) located in the 'Chips' and '3rd' folders. Normally this is not necessary (nor recommended), as the publisher will be able to decide which binaries and dependencies are required by your application!

Step 5

The last step is just to confirm you are done with all settings and ready to publish. Press 'Finish' to proceed! The publisher will then start collecting (and converting) your project files, figure out which binaries are needed, and build the resulting application as requested!

Programming Guide

To be written!

In the meantime, please have a look at the examples and tutorials to learn more about SnaX.