A NEMETSCHEK COMPANY

3D FOR THE
REAL WORLD.

DevKitchen 2018
MAXON API - General Concepts

05/11/2018

General concepts

The **microsdk** and **maxonsdk**

Changes in toolchain and project setup

Frameworks

Interfaces

Published objects

Factories

Registries

Usage of all of the above

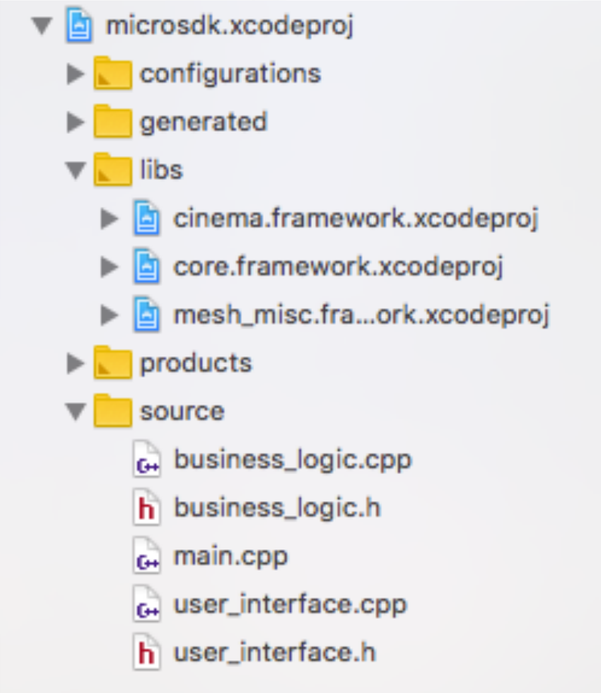
Q&A

The microsdk and maxonsdk

Together with the classic SDK example, R20 SDK comes with additional examples to show:

- MAXON API-only new functionalities — **maxonsdk;**
- Smallest and simplest possible plug-in — **microsdk.**

All new examples follows a proper code-design in order to fulfil with today's coding best-practice where business logic is kept separate from UI code and interface declaration or implementations from its use.



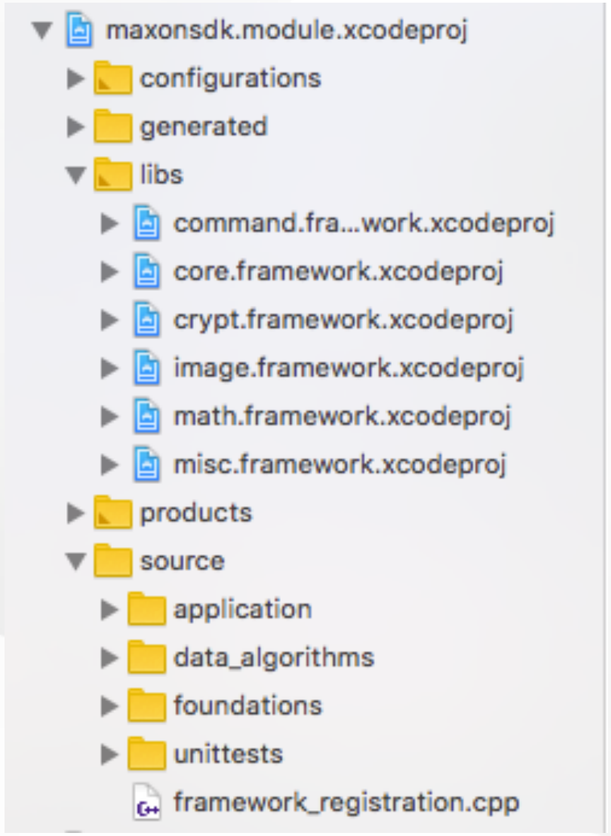
The microsdk

- Directly depends on core.framework and cinema.framework.
- Shows how classic API classes can live together with MAXON API where a very familiar action can be delivered (insert a cube) with almost no complexity.
- The action is delivered both upon user interaction or at Cinema start-up.
- It's the smallest example capable to execute code in Cinema 4D. It's the *hello world* Cinema 4D development.

```
static maxon::Result<void> MakeAndInsertCube()
{
  // "iferr_scope" needed for attributes like "iferr_return"
  iferr_scope;

  // check for main thread
  if (!maxon::ThreadRef::IsMainThread())
    return maxon::IllegalStateError(MAXON_SOURCE_LOCATION, "MakeAndInsertCube() must only be
     called from the main thread."_s);

  // get active document
  BaseDocument* const doc = GetActiveDocument();
  if (doc == nullptr)
    return maxon::UnexpectedError(MAXON_SOURCE_LOCATION, "Could not obtain active
     BaseDocument."_s);

  // create cube
  BaseObject* const cube = MakeCube() iferr_return;
  // insert cube into the given BaseDocument
  doc->InsertObject(cube, nullptr, nullptr);

  // update Cinema 4D
  EventAdd();

  return maxon::OK;
}
```

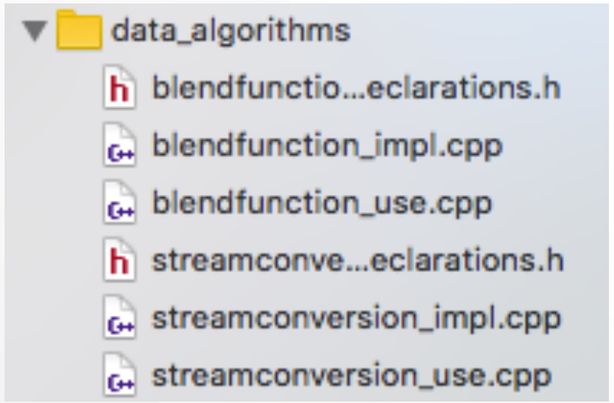
The microsdk

- Both approaches makes use of the same `MakeCube()` function.
- CommandData calls it in the `CommandData::Execute()` method.
- Executing on start-up calls it in the `MakeAndInsertCube()` when the g_executeMicroExample is valued “true" in the command line



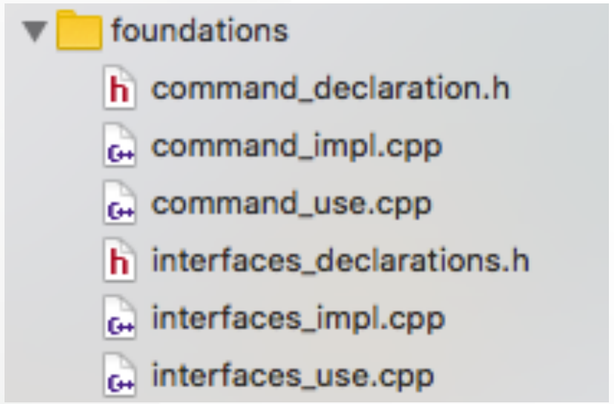
The maxonsdk

- It depends on an higher number of frameworks.
- It includes examples showing how to implement and use MAXON API-only functionalities.
- It will be constantly increased and aims to replace in a certain time the **cinema4dsdk** example set.
- Three use-cases (plus unit-tests) are delivered:
  - **Data algorithms**;
  - **Foundations**;
  - **Application**.



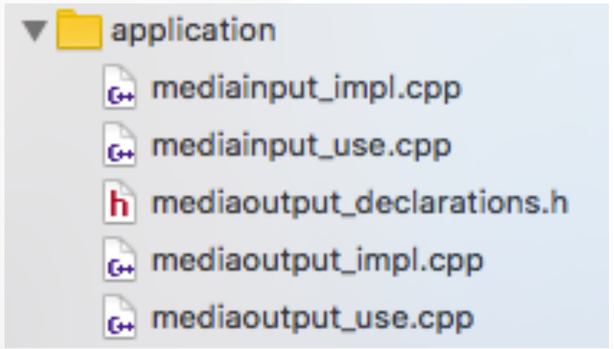
The maxonsdk - Data algorithms

- *Blendfunction_** files contain the implementation of the `BlendFunctionInterface*` to deliver a "step" blend-function and an example showing how to use it.
- *Streamconversion_** files contain the implementation of the `StreamConversionInterface*` to provide a Caesar-ciphering function and an example showing how to use it.
- Both examples are for beginners aiming to explain how to implement a native MAXON interface.



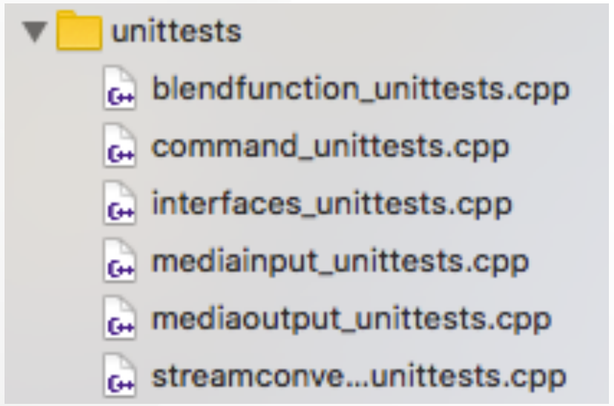
The maxonsdk - Foundations

- *command_** files contain the implementation of the `maxon::CommandClassInterface` and `maxon::CommandDataInterface` to provide a set of commands to compute average and median values out of an array of values, and an example showing their use.
- *interfaces_** files contain various **custom** interfaces declaration, implementation, re-implementation and use of *factories* plus an implementation of `maxon::DirectoryElementInterface`.
- These examples provide a basic understanding of interfaces definition and implementation.



The maxonsdk - Application

- *mediainput_impl.cpp* includes the implementation of `FileFormatInterface`, `MediaInputInterface` and `MediaSessionFileFormatHandlerInterface` to load an image based on the custom file generated by the saver example (same file format).
- *mediaoutput_impl.cpp* includes the implementation of `MediaOutputUrlInterface` to save an image based on the custom file compliant with the loader example.
- **_use.cpp* include implementations of the `ExecutionInterface` to execute the load and save tasks.



The maxonsdk - unittests

- A valuable collection of functions showing how to write and implement unit test based on the `maxon::UnitTestInterface`.
- All testing routines are executed inside the `UnitTestInterface::Run()`.

Changes in toolchain and project setup

The new API, and its interface-based paradigm, requires the toolchain to:

- include a new tool to create IDE-compatible project/solution files respecting frameworks dependencies;
- include a new source-processor to hide the complexity of the whole interface declaration/implementation/publishing mechanism;
- be flexible enough with regard to future API changes;
- be straightforward enough not to scare new developers;
- be customisable to accommodate needs of experienced developers.

The Project Tool (a.k.a kernel_app)

The tool, responsible to create project/solution files,:

- operates via command line and uses the `g_updateproject` variable;
- creates files for plugin projects, frameworks or solutions for the supported IDEs;
- uses data from `<folder>/project/projectdefinition.txt`;
- informs the source-processor on the strictness of style-checking;
- is updated, at least, on every new Cinema 4D release although intermediate releases can appear from time to time.

The Project Tool (a.k.a kernel_app)

The tool accommodate s for:

- general options;
- style-checking;
- includes;
- Visual Studio-related
- Xcode-related optio

Share your comments: we're here to **listen!**

Tailored for internal use, it's pretty new in the 3rd-party development toolset.

Frameworks

Cinema 4D SDK ships with various frameworks to include the public declaration of MAXON API interfaces, classes, structure and global functions.

These are:

- `core.framework`: for base classes, interfaces, data types, containers and data collections, threading, files, media & other miscellaneous classes;
- `cinema.framework`: for all the classic API components (to grant a smooth and constant transition from old to new paradigm);

Frameworks

- `command.framework`: for generic commands;
- `crypt.framework`: for data encryption on stream conversion;
- `geom.framework`: for various modelling classes;
- `image.framework`: for image- and media-handling classes;
- `math.framework`: for math-related classes;
- `mesh_misc.framework`: for mesh attributes;
- `misc.framework`: for various utility classes;
- `network.framework`: for handling network communication;

Frameworks

- `python.framework`: for handling the new Python-layer;
- `system.framework`: for internal system functions/classes;
- `volume.framework`: for handling volumes.

Interfaces

The new interface paradigm is a powerful toolset to define public API and hide implementation details.

Interfaces can be:

- declared;
- implemented (or re-implemented);
- inherited (from one or multiple interfaces);
- used in *factory* functions;
- used, together with their implementation, in *registries*.

Interface declaration

An interface declaration just provides a virtual class declaring the methods, the inheritance scheme and the interface's reference mode.

The interface is the base for its implementation and its reference class automatic-generation.

The interface naming scheme is **mandatory**:

- for interfaces it's: `ClassnameInterface`;
- for generated reference classes it's: `ClassnameRef`.

Cinema 4D should be notified of new interfaces by "publishing" them.

Interface declaration

During the interface declaration, it's likely to use *attributes* like:

- `MAXON_INTERFACE_BASES` to define the base interfaces inherited by an interface;
- `MAXON_INTERFACE` to mark a class declaration as a virtual interface declaration, defines the reference type and the interface ID;
- `MAXON_METHOD` to declare a member function as virtual leaving to a component its implementation;
- `MAXON_FUNCTION` to define a method that cannot be defined in a component and that must be defined in the interface declaration.

Interface declaration

```
// ------------------------------------------------------------------
// Simple class that stores a maxon::Int number.
// ------------------------------------------------------------------
class SimpleClassInterface : MAXON_INTERFACE_BASES(maxon::Object)
{
  MAXON_INTERFACE(SimpleClassInterface, MAXON_REFERENCE_NORMAL,
"net.maxonexample.interfaces.simpleclass");

public:
  // ------------------------------------------------------------------
  // Sets the number to store.
  // ------------------------------------------------------------------
  MAXON_METHOD void SetNumber(maxon::Int number);

  // ------------------------------------------------------------------
  // Returns the stored number.
  // ------------------------------------------------------------------
  MAXON_METHOD maxon::Int GetNumber() const;
};
```

Interface declaration

After the interface declaration, it's likely to use *attributes* like:

- `MAXON_DECLARATION` to declare a public object to give access to a specific implementation of an interface.

Interface declaration

```
// The .hxx header files define the reference class "SimpleClassRef".

#include "simpleclass1.hxx"

// declare the published objects "SomeSimpleClass" and "OtherSimpleClass"
// that give access to implementations of SimpleClassInterface

// the declaration must be placed between the two hxx files since "SimpleClassRef" is defined
in the first hxx file

MAXON_DECLARATION(maxon::Class<SimpleClassRef>, SomeSimpleClass,
"net.maxonexample.somesimpleclass");
MAXON_DECLARATION(SimpleClassRef, OtherSimpleClass, "net.maxonexample.othersimpleclass");

#include "simpleclass2.hxx"
```

Interface implementation(s)

Implementation is the step where, given a certain interface, the functionality is delivered.

Multiple implementations are allowed.

When an implementation is provided a component is created.

As much as for the interfaces being published, new components should be “registered” to inform Cinema 4D about them.

Interface implementation

During an interface implementation, it's likely to use *attributes* like:

- `maxon::Component` to define the interface this component implements;
- `MAXON_COMPONENT()` to declare the class as a component and adds additional code if needed;
- `MAXON_METHOD` to identify the method the component implements previously declared in the interface.

Interface implementation(s)

```
// This class implements SimpleClassInterface.
class SimpleClassImplementation : public maxon::Component<SimpleClassImplementation,
SimpleClassInterface>
{
  MAXON_COMPONENT();

public:
  MAXON_METHOD void SetNumber(maxon::Int number)
  {
    _value = number;
  }
  MAXON_METHOD maxon::Int GetNumber() const
  {
    return _value;
  }

private:
  maxon::Int _value = 1;
};
```

Component registration

As long as a component is implemented, Cinema 4D should be notified about it.

- The registration uses an ID to return the component class.
- The ID “made of” a reverse domain notation and interface class name.
- Specific attributes are used to accomplish the registration process.

Component registration

The *attributes* available to register a component are:

- `MAXON_COMPONENT_CLASS_REGISTER` to register the component with the given `maxon::Id`;
- `MAXON_COMPONENT_ONLY_REGISTER` to register only a component but no class;
- `MAXON_COMPONENT_OBJECT_REGISTER` to register a component, to create an object class using it and finally to create an instance of the object class;
- `MAXON_STATIC_REGISTER()` to register a component where static methods are defined.

Component registration

```
// Register the component using the base identifier and the implementation class name.
MAXON_COMPONENT_CLASS_REGISTER(SimpleClassImplementation,
“net.maxonexample.class.somesimpleclass");
```

or

```
// Register the component and publishes it as the published object "SomeSimpleClass".
// The published object must be declared in a header file.
MAXON_COMPONENT_CLASS_REGISTER(OtherClassImplementation, SomeSimpleClass);
```

or

```
// Register the given implementation, creates an instance of the class
// and makes that object available with the given published object "OtherSimpleClass".
MAXON_COMPONENT_OBJECT_REGISTER(ThirdSimpleClassImplementation, OtherSimpleClass);
```

Using an interface

Upon an interface's declaration and its component implementation, its functionality can be delivered through an instance of its reference class.

Reference classes:

- are automatically created and stored in the .hxx headers;
- can give access to a component or to a `maxon::NullValue`;
- are usually *reference counted* (check the Reference Type).

Using an interface (through an id)

```
// This example creates an instance of an interface
// and uses the created instance.

// define the ID of the component to use
const maxon::Id id { "net.maxonexample.class.somesimpleclass" };

// get component class of the given ID from the global maxon::Classes registry
const maxon::Class<SimpleClassRef>& componentClass = maxon::Classes::Get<SimpleClassRef>(id);

// create reference instance
const SimpleClassRef simpleClass = componentClass.Create() iferr_return;

// use reference
simpleClass.SetNumber(123);
const maxon::Int number = simpleClass.GetNumber();

DiagnosticOutput("Number: @", number);
```

Using an interface (through a published object)

```
// This example creates a reference class from a component
// registered at the given declaration.

// The "SomeSimpleClass" published object is declared in a header file.
// The obtained component is used to create a new instance using Create().
const SimpleClassRef simpleClass = SomeSimpleClass().Create() iferr_return;

// use reference
simpleClass.SetNumber(456);
const maxon::Int number = simpleClass.GetNumber();

DiagnosticOutput("Number: @", number);
```

Using an interface (through a registry)

```
// This example creates a reference object from the registry with the given ID.

// get the class with the given Id from the registry "ColorClasses"
const maxon::Id redID { "net.maxonexample.class.colors.red" };
const maxon::Class<ColorRef>* const componentClass = ColorClasses::Find(redID);

if (componentClass == nullptr)
  return maxon::UnexpectedError(MAXON_SOURCE_LOCATION, "Could not get color."_s);

// create reference object
const ColorRef color = componentClass->Create() iferr_return;

// use reference object
const maxon::String   colorName = color.GetName();

DiagnosticOutput("Color Name: @", colorName);
```

Published objects

A published object is an object being registered that is globally available.

Individual interface implementations or any other MAXON data type object can be published.

A published object:

- is declared using `MAXON_DECLARATION` attribute;
- its implementation is registered using `MAXON_DECLARATION_REGISTER`;
- is available where the header providing the declaration is included.

Published object (declaration)

```
// ----------------------------------------------------------------------
// The custom data type SimpleAtom represents a simple atom model.
// ----------------------------------------------------------------------
class SimpleAtom
{
public:
  SimpleAtom() { _protonCnt   = 1; _neutronCnt  = 0; _electronCnt = 1; };

  maxon::Int _protonCnt, _neutronCnt, _electronCnt;
};

// register as MAXON data type
MAXON_DATATYPE(SimpleAtom, "net.maxonexample.datatype.simpleatom");

#include "example_declarations1.hxx"
// register individual instances of the data type as published objects
MAXON_DECLARATION(SimpleAtom, Carbon, "net.maxonexample.atoms.carbon");
#include "example_declarations2.hxx"
```

Published object (implementation)

```
// Within the scope of MAXON_DECLARATION_REGISTER() the
// published object is created and handed over to the program.

MAXON_DECLARATION_REGISTER(Carbon)
{
  // create object
  SimpleAtom atom;

  // configure object
  atom._protonCnt   = 6;
  atom._neutronCnt  = 6;
  atom._electronCnt = 6;

  // return the object
  return atom;
}
```

Published object (usage)

```
// This example checks if the given published object is initialized.
// If so a reference to that published object is accessed.

// check if the published object "Carbon" is initialized
if (MAXON_UNLIKELY(Carbon.IsInitialized() == false))
  return maxon::UnexpectedError(MAXON_SOURCE_LOCATION, "\"Carbon\" not initialized"_s);

// get reference to published object "Carbon"
const SimpleAtom& carbon = Carbon();

DiagnosticOutput("Atom: @, @, @", carbon._protonCnt, carbon._neutronCnt, carbon._electronCnt);
```

Factories

Factories come with the `maxon::Factory` template and:

- are typically declared as published object with `MAXON_DECLARATION`;
- are defined in source code using `MAXON_DECLARATION_REGISTER`;
- their behaviour depends on a reference function invoked by the factory as:
  - `maxon::Factory::CreateFactory()` to reference a static function for creation of the desired object;
  - `maxon::Factory::CreateObjectFactory()` to reference a member function of a specific implementation.

Factories (declaration)

```
// ----------------------------------------------------------------------
// Class to store a string's hash value.
// ----------------------------------------------------------------------
class StringHashInterface : MAXON_INTERFACE_BASES(maxon::Object)
{
  MAXON_INTERFACE(StringHashInterface, MAXON_REFERENCE_NORMAL, "net.maxonexample.stringhash");

public:
  MAXON_METHOD maxon::Result<maxon::Bool> IsStringEqual(maxon::String str) const;
};

// This example shows the declaration of a factory creating a StringHashRef object from a given
maxon::String argument.
using StringHashFactoryType = maxon::Factory<StringHashRef(maxon::String)>;
MAXON_DECLARATION(StringHashFactoryType, StringHashFromStringFactory,
"net.maxonexample.factory.stringhashfromstring");
```

Factories (implementation)

```
// implementation of StringHashInterface
class SecretStringHashImplementation : public maxon::Component<SecretStringHashImplementation,
StringHashInterface>
{
  MAXON_COMPONENT();

public:
…
  // initialises the object with the given string
  maxon::Result<void> InitFromString(maxon::String& string){…}

  // Factory method to be used with Factory::CreateObjectFactory()
  maxon::Result<void> FactoryInit(maxon::FactoryInterface::ConstPtr, maxon::String string)
  {
    iferr_scope;
    InitFromString(string) iferr_return;
    return maxon::OK;
  }
};
```

Factories (registration)

```
// CreateObjectFactory() is calling a member function of the implementation class.
// It will create a new instance of this implementation and then call the given member function.

MAXON_DECLARATION_REGISTER(StringHashFromStringFactory)
{
  return StringHashFactoryType::CreateObjectFactory(&SecretStringHashImplementation::FactoryInit);
}
```

Registries

Registries are used to register specific implementations of a given interface. An implementation that is registered in such a registry can be anywhere accessed with its ID.

A registry can be:

- declared via the `MAXON_REGISTRY` attribute;
- registered via the `MAXON_COMPONENT_CLASS_REGISTER` attribute;
- used by accessing the component with its specific id from the registry;

Registries (declaration)

```
// Simple class to store a color.
class ColorInterface : MAXON_INTERFACE_BASES(maxon::Object)
{
  MAXON_INTERFACE(ColorInterface, MAXON_REFERENCE_NORMAL, "net.maxonexample.interface.color");

public:
 // Returns the name of the color.
  MAXON_METHOD maxon::String GetName() const;

 // Returns the RGB value of the color.
 MAXON_METHOD maxon::Color32 GetRGB() const;
};

#include "example_registry1.hxx"

// Define registry "ColorClasses" to give access to implementations of the ColorInterface interface
MAXON_REGISTRY(maxon::Class<ColorRef>, ColorClasses, "net.maxonexample.registry.colors");

#include "example_registry2.hxx"
```

Registries (implementation & registration)

```
class ColorRedImpl : public maxon::Component<ColorRedImpl, ColorInterface>
{
  MAXON_COMPONENT();

public:
  MAXON_METHOD maxon::String GetName() const { return "Red"_s; }

  MAXON_METHOD maxon::Color32 GetRGB() const
  {
    maxon::Color32 color;
    color.r = 1.0; color.g = 0.0; color.b = 0.0;
    return color;
  }
};
using namespace maxon;

// register component class and add it to registry "ColorClasses"
MAXON_COMPONENT_CLASS_REGISTER(ColorRedImpl, ColorClasses, "net.maxonexample.class.colors.red")
```

Registries (usage)

```
// get the class with the given Id from the registry "ColorClasses"
  const maxon::Id redID { "net.maxonexample.class.colors.red" };
  const maxon::Class<ColorRef>* const componentClass = ColorClasses::Find(redID);

  if (componentClass == nullptr)
    return maxon::UnexpectedError(MAXON_SOURCE_LOCATION, "Could not get color."_s);

  // create reference object
  const ColorRef color = componentClass->Create() iferr_return;

  // use reference object

  const maxon::String   colorName = color.GetName();
  const maxon::Color32 colorRGB = color.GetRGB();

  DiagnosticOutput("Color Name: @", colorName);
  DiagnosticOutput("Color RGB: @", colorRGB);
```

Usage of all of the above

The soon-to-be-published “Chart” example provides a nice mixture of all the previous topics in an easy to consume-and-digest recipe.

Based on the idea of a “chart creator”, at the moment, delivers:

- virtual interfaces;
- observables;
- MAXON data types;
- combined use of classic and MAXON API;
- factories;
- delegates.

Thanks for Your Interest and Patience!

Questions?

3D FOR THE REAL WORLD.

MAXON Computer GmbH
Max-Planck-Straße 20
D-61381 Friedrichsdorf
Germany

E info@maxon.de
T +49 (0) 6172 5906-0
F +49 (0) 6172 5906-30

maxon.net

05/11/2018