A NEMETSCHEK COMPANY

3D FOR THE
REAL WORLD.

DevKitchen 2018
Adapting Plugins

05/11/2018

Adapting plugins

Plugins before R20

The **sdk.zip**

The **kernel_app** for R20 project

Building all the example projects

Setup & first compile

Getting through the building issues

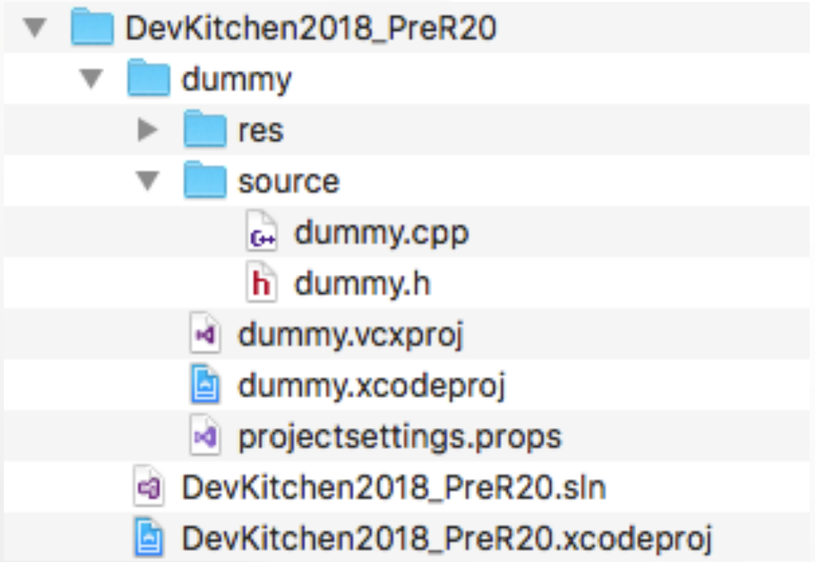
"Aptat et impera"

Q&A

Plugins before R20

Key-features

- Projects & solutions were cloned or manually created/edited.
- Plugins were located in the Cinema 4D "plugins" folder.
- Plugins were exclusively relying on *cinema.framework*.
- Plugins were binary-compatibility with upcoming Cinema 4D releases.
- SDK was natively shipped with the Cinema 4D application.



Notes on the folder structure:

- Project and solution files provided by default;
- Project files are in the project's root;
- Solution file is in the solution’s root.

The sdk.zip

Since R20, Cinema 4D SDK comes:

- as a separate ZIP archive (sdk.zip);
- with the option to be stored and used everywhere;
- with old and new examples sets;
- with *projectdefinition.txt* files for creating files for IDEs;
- with the *sourceprocessor* for enhanced API usability.

The kernel_app for R20 project

A brand-new tool responsible for:

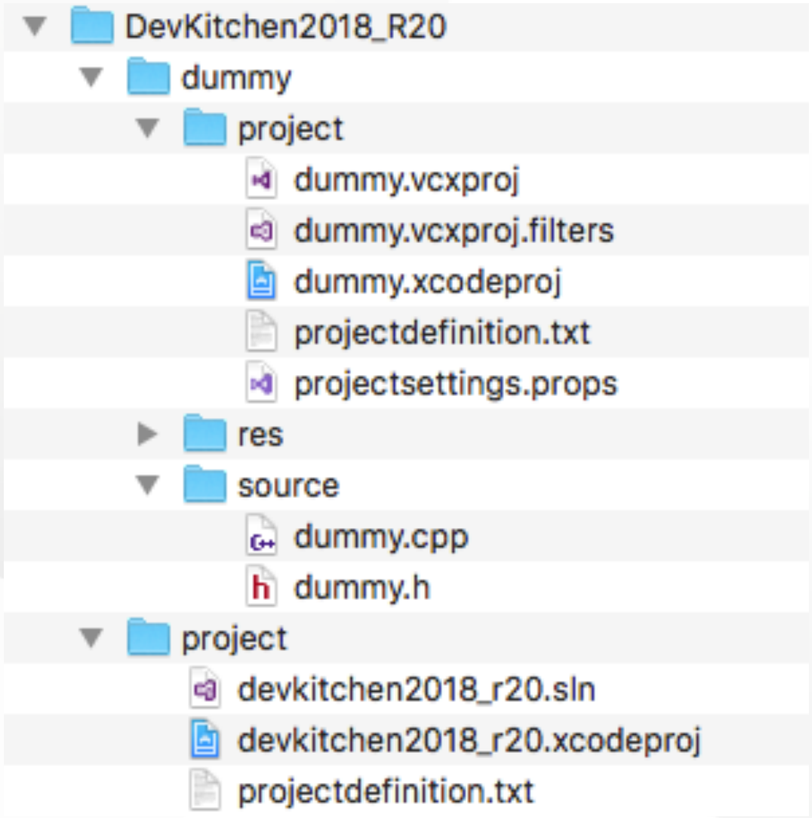
- creating “solution” and “project" files for new plugins;
- creating “project” files for new frameworks;
- generating files compatible with Visual Studio 2015 and Xcode 9.x*
- updating project files upon new files add/removal.

Started via cmd-line or terminal, the tool can be stored everywhere and should be downloaded from here.

Plugins from R20 onward

Key-features

- Projects & solutions are automatically created.
- Plugins can be arbitrarily* located on storage devices.
- Plugins are relying on a certain set of frameworks.
- Plugins code can use of advanced design approaches.
- C++11 is natively supported.
- Plugin recompiling is required for upcoming Cinema 4D releases.



Upon projecttool execution

Notes on the folder structure:

- Project and solution files aren't provided by default;
- Solution and projects folder have a **`project`** folder containing the **`projectdefinition.txt;`**
- Solution and project files are created by the Project Tool using the parameters found in the **`projectdefinition.txt`**.

Building all the example projects

Suggested steps:

1. unzip the provided sdk.zip in your favorite location;
2. download the Project Tool from developer.maxon.net;
3. unzip the tool and run it to create the project files;

```
./kernel_app [kernel_app_64bit.exe] g_updateproject=<full path to unzipped sdk>
```

4. open the generated solution file via the proper IDE and build it;
5. run Cinema 4D by specifying the g_additionalModulePath;

```
./“CINEMA 4D" [“CINEMA 4D.exe"] g_additionalModulePath=<full path to sdk/plugins>
```

Setup & first compile (objectdata_ruledmesh)

Suggested steps:

1. delete all the previous solution- and project-related files;
2. create a `project` folder both in the project and in the solution;
3. create a `projectdefinition.txt` in the folders created in step 2;
4. edit the files created in step 3 to properly define the solution and the project found in the solution;
5. load in your IDE and run a first compile.

Getting through the building issues

Suggested steps:

1. fix flaws from sourceprocessor check (`void`, indentation, spaces);
2. update the name of the string resource folder;
3. verify (fully-qualified) identifiers and global variables;
4. fix simple enumeration with enumerations classes;
5. consider to use new data types (e.g. vector, string, matrix);
6. replace `GePrint` with tools provided by *LoggerInterface*;
7. check and adapt code for error handling.

1. Fix flaws from source-processor check

The *source-processor*, which is part of the tool chain to build a plugin (see here), checks the source code and automatically creates additional code. This processor also checks the code style according to the level set in the project's projectdefinition.txt file (see here).

2. String resource folder rename

The naming scheme of string resource folders is changed.
The resource folder that was previously named "*strings_us*" is now named "*strings_en-US*" (see here).

3. Specify namespace

The complete MAXON API is defined in the *maxon* namespace. In contrast, the classic API is defined in the global namespace.
To avoid collisions and ambiguity it might be necessary to use fully-qualified names.

```
// This example uses explicit namespaces to use new and classic components.

  // creating a maxon::String
  maxon::String helloWorld { "Hello World" };

  // creating a classic String
  ::String fooBar { "FooBar" };
```

3. Replace `resource` with `g_resource`

The global resources instances is renamed as *g_resource*.

```
// This example loads the plugin's resources when
    // C4DPL_INIT_SYS is sent to PluginMessage().

    case C4DPL_INIT_SYS:
    {
      // don't start plugin without resource
      if (!g_resource.Init())
        return false;
      return true;
    }
```

4. Fix enumerations

Simple enumerations are reformatted as enumeration classes.
Since `"0"` is not a valid enumeration value, name it is replaced with "`NONE`".
The macro `ENUM_END_FLAGS()` has been removed and for custom enumeration it's requested use `MAXON_ENUM_LIST()` or `MAXON_ENUM_FLAGS()`.

5. Use new data types

The new maxon::String class is the base class of the classic String class.
To declare a string literal as a maxon::String one can use the "_s" qualifier.
New maxon::Vector and maxon::Matrix classes replace their classic counter parts. The Matrix class has a slightly different internal structure.
More explicit data types are available: maxon::ColorA is used instead of vectors for explicit color types.

6. Make use of a logging system

Cinema 4D R20 introduced a new, complex and customisable logger system where the standard console is just one of many available loggers where writing comes as easy as using `ApplicationOutput()`.

```
// define message
const maxon::String someMessage { "some message" };

// print the message to the console
DiagnosticOutput(someMessage);

// print the message to the application console window
ApplicationOutput(someMessage);
```

7. Avoiding ugly errors with nice errors

The MAXON API error system allows functions to return an explicit error state. Being several functions of the classic API already replaced with new functions, returning such an error state updating the code is mandatory.
Pay attention to:

- gracefully handle error in functions/methods returning an error state;
- where convenient return `maxon::Result<type>` instead of *`type`*;
- keep extending the error handling functionality where possible.

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