Magus Manual

Version 0.4

Introduction

Magus is a (Windows) tool – developed with Qt – that generates (C++) code for GUI applications and offers a set of standard widgets to quickly create an application. Magus contains:

- **The Magus wizzard** – this application generates a C++ project, based on a template. The template can be modified in the application. Currently only Qt projects are supported.
- **An iconset** – The application comes with a set of icons (512x512) specific for 3D type of applications. It is possible to use your own icons.
- **The Magus library** – In addition, Magus comes with a set of extra widgets to support fast development. These widgets produce no additional c++ code. If you want to use them, you have to write your own code (which is relatively easy). Examples are included in the manual.

The figure below illustrates in a nutshell what Magus does:
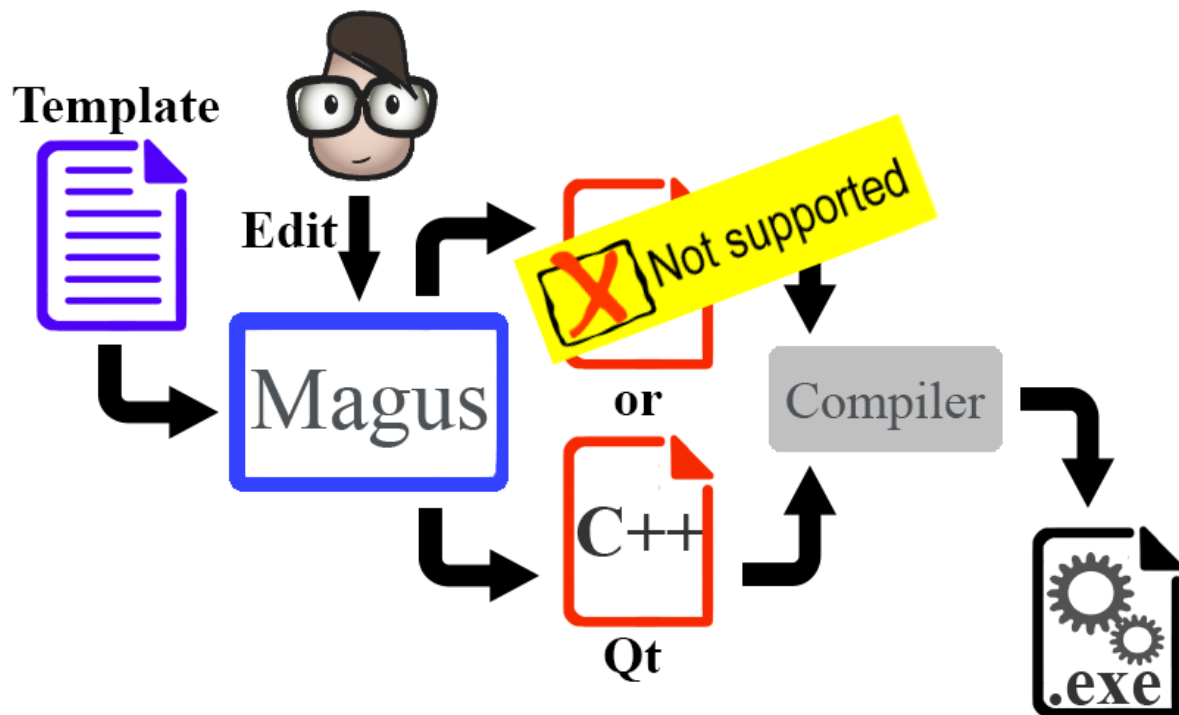
Magus – the application

Building Magus

Magus is an application build by means of Qt. Magus is hosted on Github (https://github.com/spookyboo/Magus). The structure of the project is:

bin		Contains all files needed to run the application (dll, exe, config (cfg) files, …etc.)
bin	platforms	This directory must be present to run the application stand-alone (outside Qt Creator). In case of Windows, it contains the file *qwindows.dll*.
common		Directory that contains generic items.
common	icons	Contains all the icons that are used in Magus, but also icons that can be used in the applications generated by Magus.
common	layout	Example images that represents a layout. The layout file (image) can be defined within a template file. The Magus application does not support defining a layout file for a particular project (yet). Defining the layout file in a template file must be done manually.
common	ogre	Includes generic assets used in an Ogre widget.
common	qt	Contains the Qt sources that are used in a generated Qt application (generated by Magus). The Magus application determines which Qt sources are relevant for a generated application. This is defined in the template file.
docs		Magus manual.
samples		The samples directory.
source		Contains the source (C++) files of the Magus application.
templates		Includes the template files. A template file contains the definition of a magus project.

Magus Gui wizzard

⏱ 37 commits	⑂ 1 branch	🏷 2 releases	👥 1 contributor

Branch: master ▾   **Magus** / +

Fixed generic_functions and magus_treewidget => added to generated pr... ···

spookyboo authored 25 days ago   latest commit 19bdba9030

📁 bin	added resourcetree widget, renamed texture widgets, updated docs	27 days ago
📁 common	Fixed generic_functions and magus_treewidget => added to generated pr...	25 days ago
📁 docs	Added Resource demo	25 days ago
📁 samples	Added Resource demo	25 days ago
📁 source	Fixed generic_functions and magus_treewidget => added to generated pr...	25 days ago
📁 templates	added resourcetree widget, renamed texture widgets, updated docs	27 days ago
📄 Magus.png	no message	a month ago
📄 README.md	no message	a month ago
📄 magus.pro	Added GLSphereWidget and FileReader to Tools (and updated Tool sample)	2 months ago

Open the file *magus.pro* in Qt Creator, compile it and 'Run' the application.

Manual

After starting the Magus wizzard application, an empty screen with a menu is displayed.

File menu

New project	Create a new project. This opens a dialog with a number of templates from were the user can make a choice. The template is a good start and already has some settings for docking windows, menu items and toolbars.
Open…	Open a project that was previously saved.
Save project	Save the project. By default, this is the name of the template, but another name can be given to the project.
Build	Generate all project files (C++ code, images, etc.)
Exit	Quit the application.

Config	Opens a configuration dialog. The configuration dialog updates the settings in the file *global.cfg.*
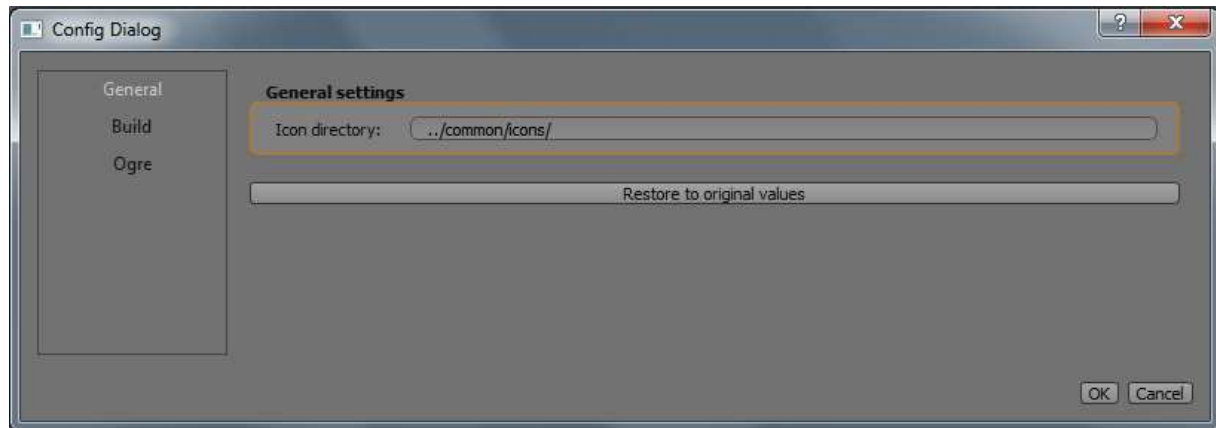
The config dialog has a few settings:

General > icon directory	The directory that refers to the icon images used by the Magus application.
Build > Output directory	The directory where Magus writes the 'builds'.
Ogre > environment variable	If selected, the environment variable is used to point to the Ogre root directory.
Ogre > root directory	If selected, the given directory is used as Ogre root directory.

Magus project

Select 'New project' and select template *layout_2.ide*, for example:
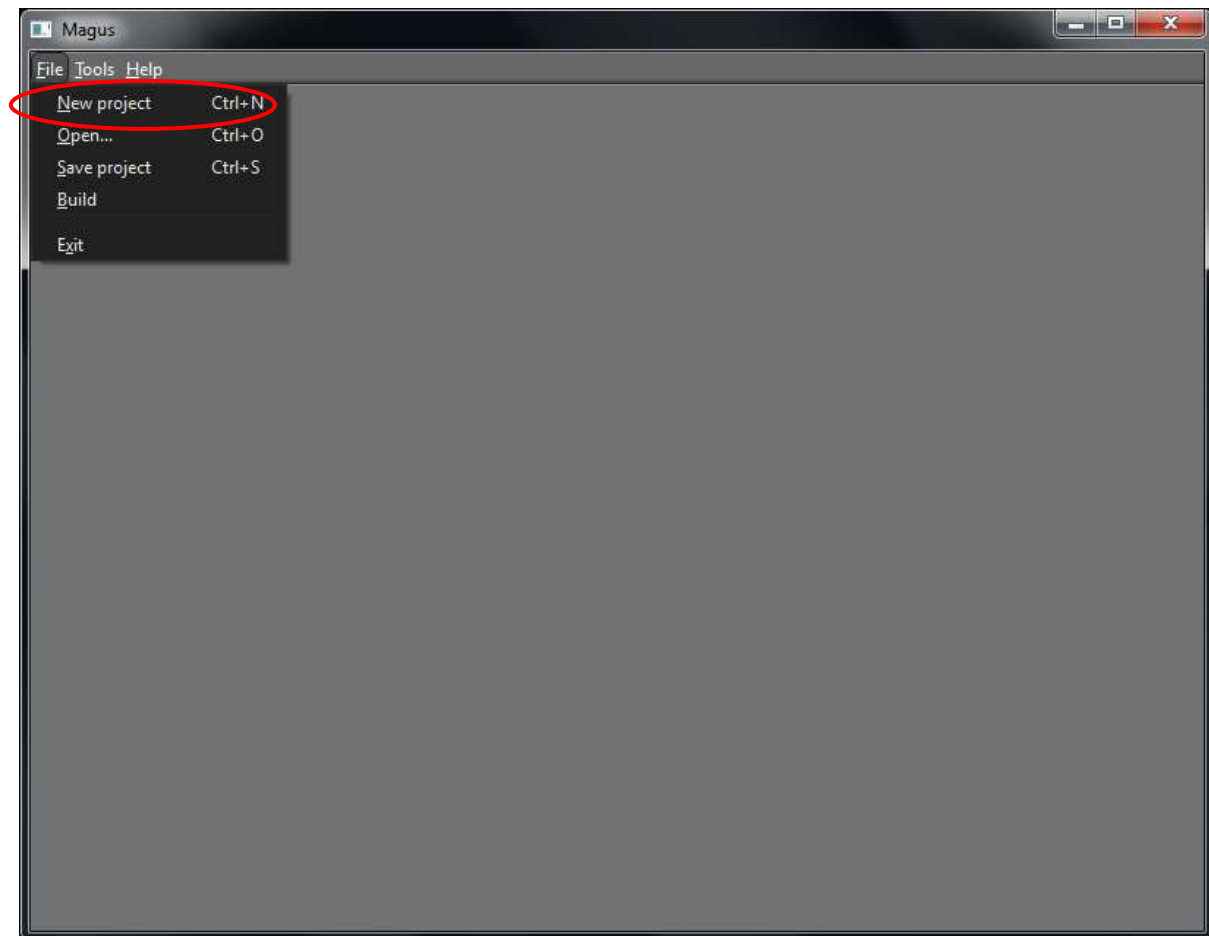
This results in 4 tabs, which represent generic settings, a main window (with a menu) and 2 dockable windows.

Select 'Build' from the menu. This generates all C++ code, including all addional files. In case of Qt, a *Layout2.po* file is generated, so the project can easily be used in combination with Qt Creator.
By default, Magus writes the project to *c:/temp/magus_out/Layout2*
For Qt, just open *Layout2.pro* in Qt Creator and run the application.

Note
The target (build) directory is defined in a configuration file. This file can be found in */magus/bin/* and is called *global.cfg*.
These settings can be changed by means of a configuration dialog (see next section).

Note, for Ogre3d usage:
If Ogre3d is used, Magus generates C++ code that refer to Ogre3d header files. It also generates a .po file with a reference to the Ogre3d source directory. This source directory is defined in the *global.cfg*. The tag ogre_root_env refers to an environment

variable. By default this environment variable is called OGRE_ROOT, but you also defined your own environment variable.
If no *ogre_root_env* tag is specified, it is also possible to specify the directory by means of *ogre_root*.

The Ogre settings can also be changed by means of the configuration dialog.

General project settings
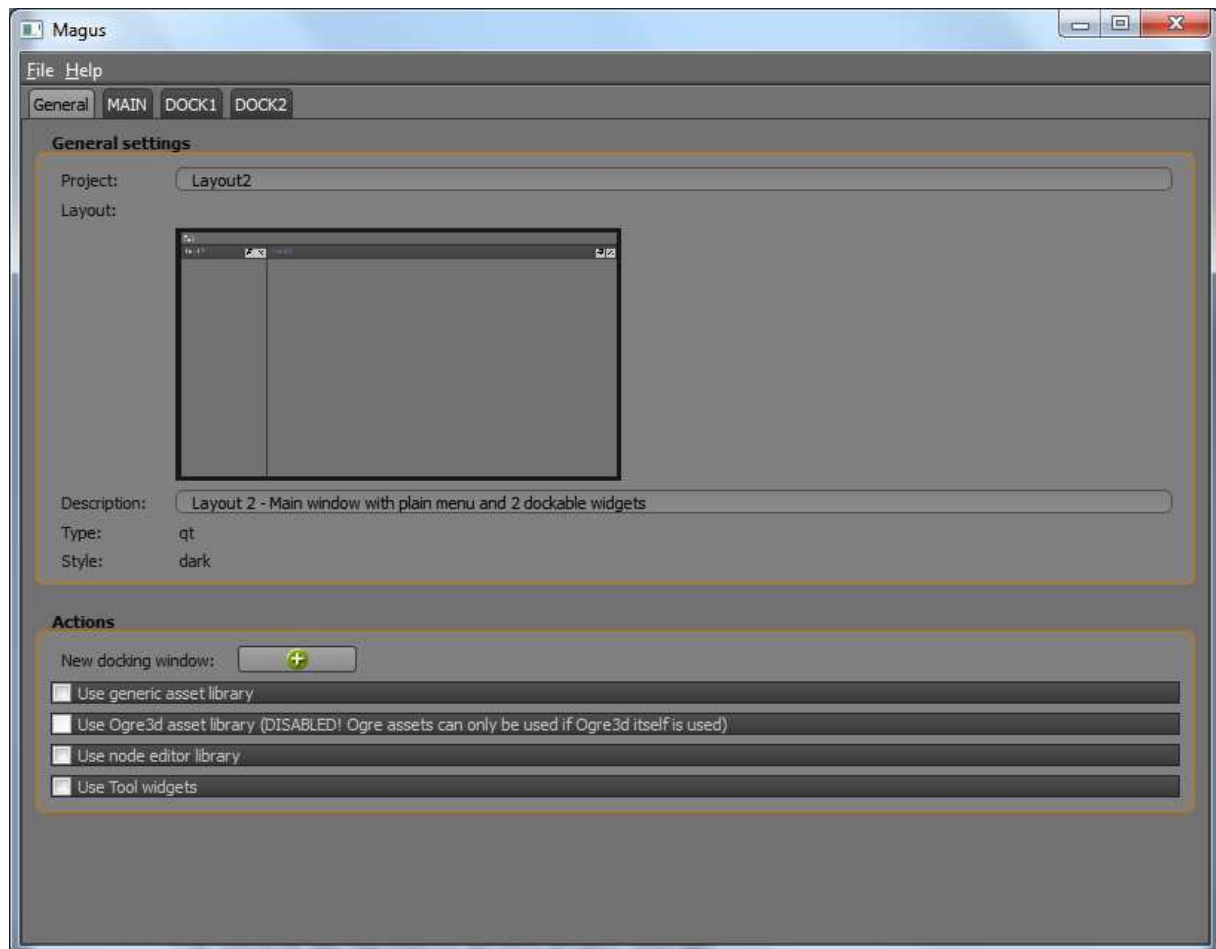
The *General* tab represents generic project settings:

General settings

Project	Name of the project. When the project is build, this becomes the name of the root directory.
Layout	Visual representation of the layout (this is a static image).
Description	Description of the project.
Type	Determines the type of project is generated. Default is a Qt project. Wxwidgets is not available yet.
Style	Style of the application. In case of Qt this is the 'dark' style by default. Other styles aren't available (yet).

Actions

New docking window	Adds another docking window to the project. This new window is represented by another tab in the Magus application.
Use generic asset library	Not everything can be build automatically. Magus provides additional asset widgets that can be used in the application. When this checkbox is checked, the generic asset files are added to the generated project. See next chapter for more details.
Use Ogre3d asset library	In addition, Magus also provides asset widgets specific for the Ogre rendering engine. The Ogre asset widgets depend on the generic asset widgets. When checked, all asset files are added to the generated project. See next chapter for more details.
Use node editor library	Include the c++ files of the node editor widget
Use Tools widgets	Additional widgets for image/icon selection, gradient creation, …etc.

Window settings

Each application, generate by Magus, has one main window and zero or more docking windows. The mainwindow differs on certain features; deleting the mainwindow and adding tabs is not possible.

Window - mainwindow

Title	Title of the window. This name is also used to generate C++ code (in the filename and the classname); any special character that is not allowed in a C++ classname is stripped from the title. If the checkbox 'Display the title of the window' is checked, the title is displayed in the titlebar.
Order	The order of a mainwindow is always 0. Order is used as a 'window index'.

Window - docking window

The window properties of a docking window has some additional options.

Title	See above
Order	See above; the order > 0
Dock area	Define the area in the mainwindow where this docking window initially is set.
Delete this window	The current tab is deleted, which implies that the docking window associated with this tab is not generated after a 'build'.

Menu

Each window (mainwindow and docking window) may have its own menu (horizontally positioned right under the titlebar.

Add menu item	Add a menu item to the menutree (i.e File, Edit, Window, Help). The menu items are distributed horizontally in the menu.
Add submenu item	Add a subitem to the menutree. A subitem is an item positioned under a menu item (vertically distributed).
Remove item/subitem	Remove the currently selected item or subitem.
Menutree	The menutree is a representation of the menu. Each menuitem and subitem can be edited by doubleclicking on it.

Default Actions

A default action is a preconfigured action, which can be associated with a submenu item. The association is based on the exact name of the subitem and the default action. E.g. if the application has an Exit-action, the subitem name must be named exactly 'Exit'.

Reset Window Layout	If this is checked and one of the subitems in the menu is named 'Reset Window Layout', all docking windows are positioned according their original position.
Exit	If checked and one of the subitems in the menu is named 'Exit', the application quits when this subitem is selected.
Quit	Same as 'Exit', but with the 'Quit' keyword.
About	If checked and one of the subitems in the menu is named 'About', an About messagebox is displayed when the subitem is selected. Note, that this is a basic About messagebox. You have to change it to your own needs.

Toolbar

Magus has a Toolbargroup with which it is possible to create a horizontal and /or vertical toolbar. This is done by dragging and dropping the icons from the iconoverview to the horizontal and/or vertical area. In the image below, the 'Primitives' icons are selected by means of the combobox. Two of these icons are dragged/droppen on the vertical toolbar. The horizontal toolbar contains 'File' icons.

| Combobox | Icons are grouped into categories. The categories are selected by means of this combobox. There is one special category named 'Separator'. An icon of this type is translated into a real separator in the generated toolbar.

Note: |
| --- | --- |

	The grouping in categories is defined in a file called *icons.cfg* (in the bin directory)
Icon area	The icons from where you can select are displayed in this area.
Horizontal toolbar area	Icons can be dragged from the Icon area to the Horizontal toolbar area
Vertical toolbar area	Icons can be dragged from the Icon area to the Vertical toolbar area
Bin	An icon in the Horizontal/Vertical toolbar area can be removed by dragging and dropping on the Bin.
Transformation	Additional (custom) widgets can be added to the toolbar. Currently, there is only one widget called 'Transformation'. This widget can be used to change Position, (Euler) Rotation and Scale of an object. This usually is a selected object in a renderwindow. *Note:* This widget on its own has no meaning. It must be used in combination with your own code.

Tab

Tabs can be added to each docking window.

Add tab	Create a new tab. The tab has a default name ('Doubleclick to edit') and a default icon (an 'i' icon). Both the tab text and the icon can be changed by doubleclicking on the tab. This displays a dialog in which both tabtitle and icon can be changed.
Remove selected tab	Remove the currently selected tab.

Use Ogre3d

Magus makes it possible to use Ogre3d in combination with the generated project. It adds an Ogre widget to a specific target (mainwindow, docking window or tab).
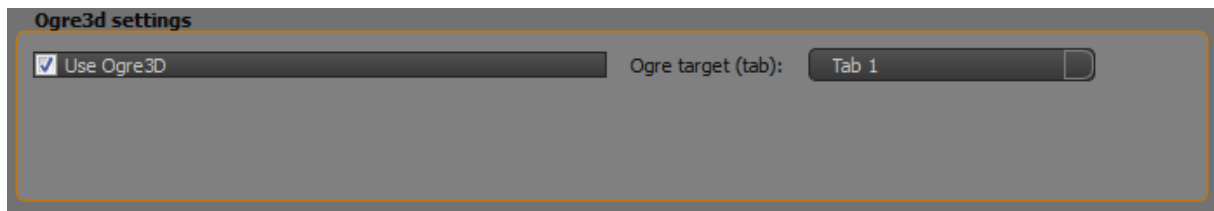
Use Ogre 3D	If checked, code is generated that adds an Ogre 3D renderwidget
Ogre target (tab)	By default the Ogre renderwidget is set in the window itself, but is tabs are defined, the renderwidget is set in one of the selected tab.

Magus – the library (Qt)

Magus only generates the code of a basic GUI framework; its purpose is not to create a fully working application. Creating a real application requires additional coding. Magus speeds up development with a set of additional widgets. These widgets are grouped into several libraries:

- Generic asset library – Widgets for assets and properties.
- Ogre3d asset library – Widgets for Ogre3d; these are to demonstrate how to use the generic asset library.
- Node editor – An easy to use node editor.
- Tool widgets – Additional widgets for various purposes.

The libraries can be added to a project by means of the Magus wizzard application. Create a new project and select one or more libraries in the *General* tab.
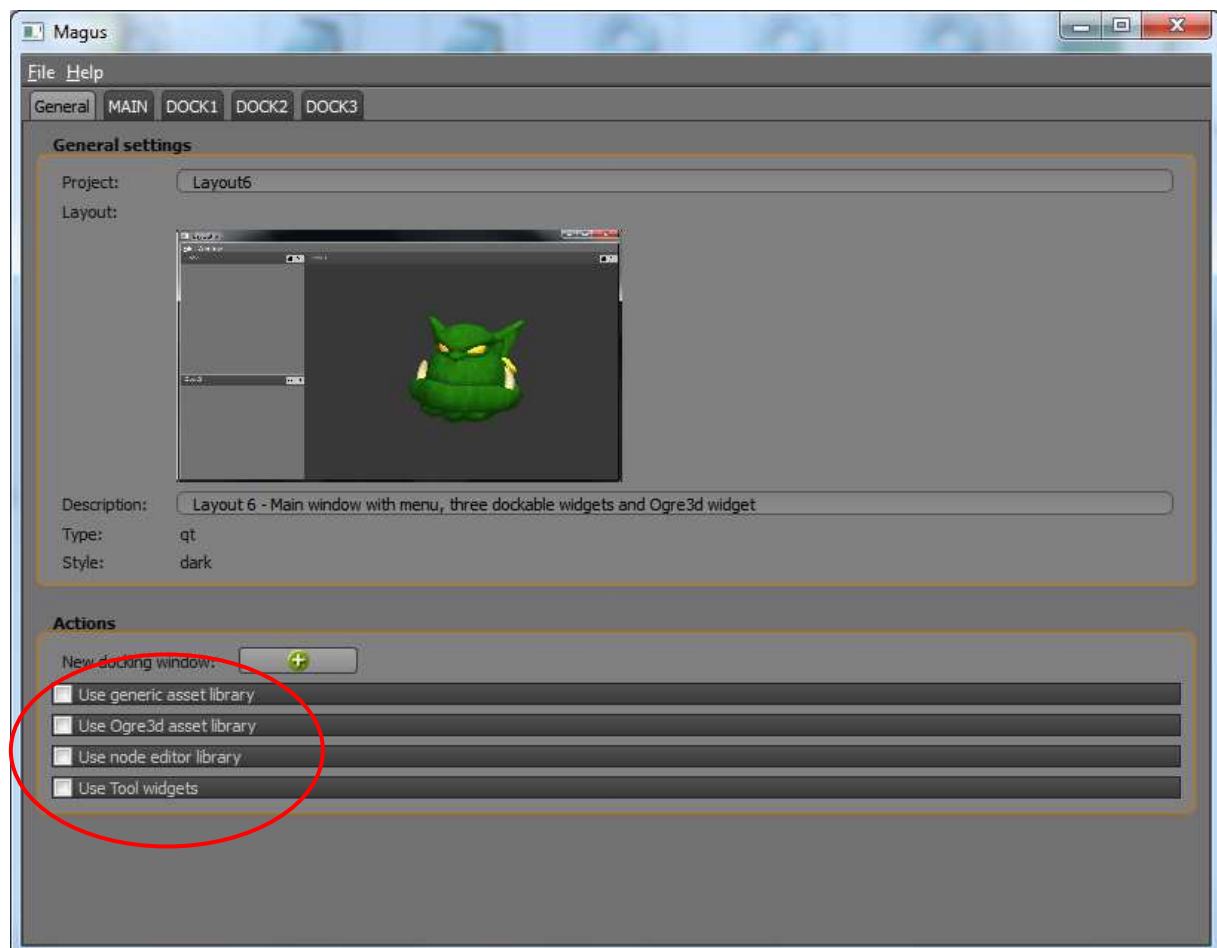
Currently only Qt libraries are present.

Generic asset library

The generic asset library is all about assets and properties. C++ files that are part of the Generic Asset library are:
asset*.h / asset*.cpp

- The main widget is `QtAssetWidget` (see *asset_assetwidget.h*). `QtAssetWidget` contains a header with title, an icon (optional) and 2 optional action icons. The action icons can be used to perform a certain action on the widget or the object that is associated with it.
- `QtAssetWidget` on its turn contains 0..n `QtContainerWidget` objects (see *asset_containerwidget.h*)
- Each `QtContainerWidget` contains either other `QtContainerWidget` objects or `QtProperty` objects. A `QtContainerWidget` can have 2 optional action icons assigned to it.
- `QtProperty` objects are the widgets that refer to properties. `QtProperty` is subclassed into different specialized `QtProperty` classes. Distinguished are:

QtCheckBoxProperty	Property for Boolean types.
QtColorProperty	Property with which a colorvalue can be defined. It contains numerical entry fields, but also a button that opens a colorpickerdialog.
QtCurveProperty	Property with which a curve can be created. A curve is defined by means of control points that are created with a curve editor (dialog).
QtDecimalProperty	Property for Numerical types.
QtQuaternionProperty	Property for Quaternions (w, x, y, z).
QtSelectProperty	Property that contains a combobox.
QtSliderProperty	Property with a slider and an edit field. Values can be entered in the edit field or changed by means of the slider.
QtStringProperty	Property for Strings.
QtTextureProperty	Property that provide access to a file dialog to select a texture.
QtXYProperty	Property for entering x and y values.
QtXYZProperty	Property for entering x, y and z values.

Signals on individual `QtContainerWidget` or `QtProperty` objects can be handled on the objects themselves, but it is more convenient to do this on a higher-level, via the `QtAssetWidget`.

Example: Create an Asset

```cpp
// Create an QtAssetWidget object
QVBoxLayout* mainLayout = new QVBoxLayout;
QtAssetWidget* assetWidget = new QtAssetWidget(QString("Test"), QString("test.png"), this);
assetWidget->setFileNameIconCollapsed(QString("collapse.png"));
assetWidget->setFileNameIconExpanded(QString("expand.png"));
assetWidget->setHeaderTitleBold();

// If the value of a property in the assetWidget changes, propertyValueChanged is called+
connect(assetWidget, SIGNAL(valueChanged(QtProperty*)), this,
                     SLOT(propertyValueChanged(QtProperty*)));
```

Example: Create a container and add properties to the container

```cpp
// Create an QtContainerWidget object
QtContainerWidget* container = assetWidget->createContainer(1, QString("Testcontainer"));
container->setTitleIcon(QString("cube_bold.png"));
container->setTitleBold();

// Add properties to the container (this can also be done by means of assetWidget)
container->createProperty(10, QString("String property"), QtProperty::STRING);
container->createProperty(11, QString("XYZ property"), QtProperty::XYZ);
```

Example: Value of a property changes

If the value of a property changes, a signal is emitted to both its parent container and to the assetWidget that includes the container. The easiest way to catch the signal is via the QtAssetWidget.

```cpp
// Function propertyValueChanged is a slot
void MyMain::propertyValueChanged(QtProperty* property)
{
    QtXYZProperty* xyzProperty = 0;
    QtStringProperty* stringProperty = 0;
    switch (property->mPropertyId)
    {
        case 11:
        {
            stringProperty = static_cast<QtStringProperty*>(property);
            QString str = stringProperty-> getString();

            // Do something
        }
        break;

        case 11:
        {
            xyzProperty = static_cast<QtXYZProperty*>(property);
            qreal x = xyzProperty->getX();
            qreal y = xyzProperty->getY();
            qreal z = xyzProperty->getZ();

            // Do something
        }
        break;
    }
}
```

QtCurveDialog

The `QtCurveProperty` makes use of a `QtCurveDialog` object. The `QtCurvedialog` object is part of the Generic Asset library, but can also be used as a stand-alone dialog for other purposes. Make sure to provide the icon directory when using the `QtCurveDialog`. The icon directory contains the path to the icons displayed in the toolbar of the `QtCurveDialog`.
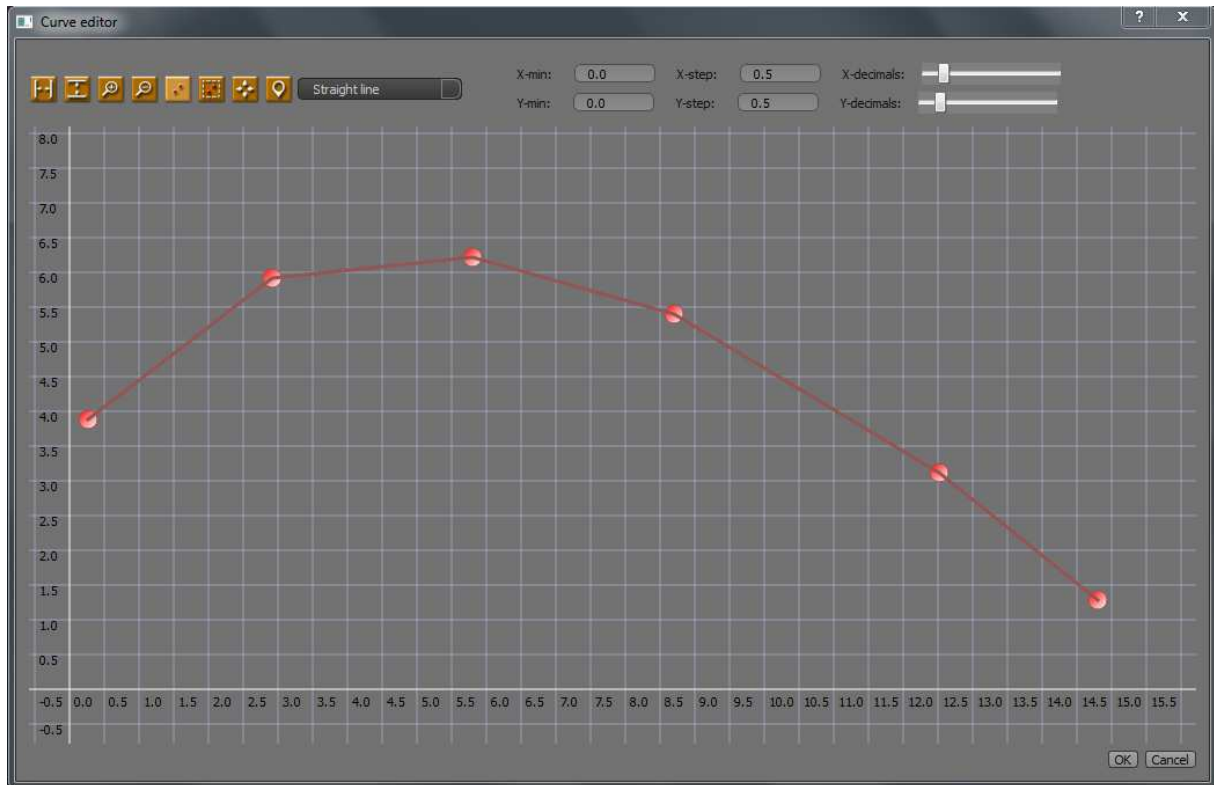
Files: asset_curve*.h / asset_curve*.cpp

Ogre asset library

Magus comes with a few Ogre asset widgets. These widgets are used for manipulating specific Ogre objects, such as cameras, entities, … By selecting the option 'Use Ogre3d asset library' in the Magus application, the associated files (including icons) are copied when a project is build. The project file is updated with the references.

If you want to use an Ogre asset widget, some additional code must be added.

An example:
Assume, the generated project contains a docking window, named 'Properties' with source *properties_dockwidget.cpp*. An Ogre camera asset widget can be added by means of the following piece of code:

```cpp
#include "ogre_asset_camera.h"

//***************************************************************************/
PropertiesDockWidget::PropertiesDockWidget(QString title, MainWindow* parent, Qt::WindowFlags flags) :
    QDockWidget (title, parent, flags),
    mParent(parent)
{
    mInnerMain = new QMainWindow();
    setWidget(mInnerMain);

    // Perform standard functions
    createActions();
    createMenus();
    createToolBars();

    // Ogre asset widget example
    Magus::QtOgreAssetCamera* assetCamera = new Magus::QtOgreAssetCamera(QString("../common/icons/"));
    Ogre::Camera* camera = parent->getOgreManager()->getOgreWidget(2)->mCamera; // Example
    assetCamera->bindObject(camera);
    mInnerMain->setCentralWidget(assetCamera);
}
```

An Ogre Camera (you decide which one, of course) is bound to the widget by means of:
```cpp
void QtOgreAssetCamera::bindObject (Ogre::Camera* camera)
```

This is uni-directional. If a change is made in the Ogre asset widget, the Ogre camera is automatically updated. If the Ogre camera is changed in some other way, the Ogre asset widget is not updated.

The Ogre asset library can be added to a project by means of the Magus wizzard application. Create a new project and select 'Use Ogre3d asset library' on the *General* tab.

Ogre asset widget overview

QtOgreAssetCamera
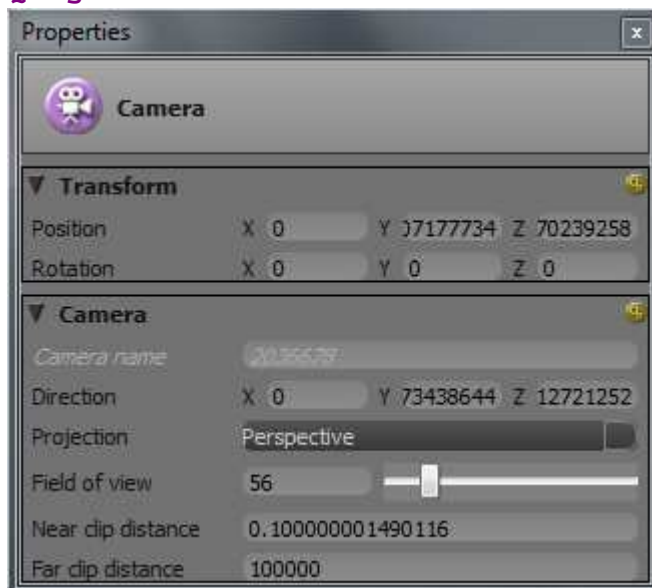
Files: ogre_asset_camera.h / ogre_asset_camera.cpp

QtOgreAssetEntity

Files: ogre_asset_entity.h / ogre_asset_entity.cpp

QtOgreAssetMaterial
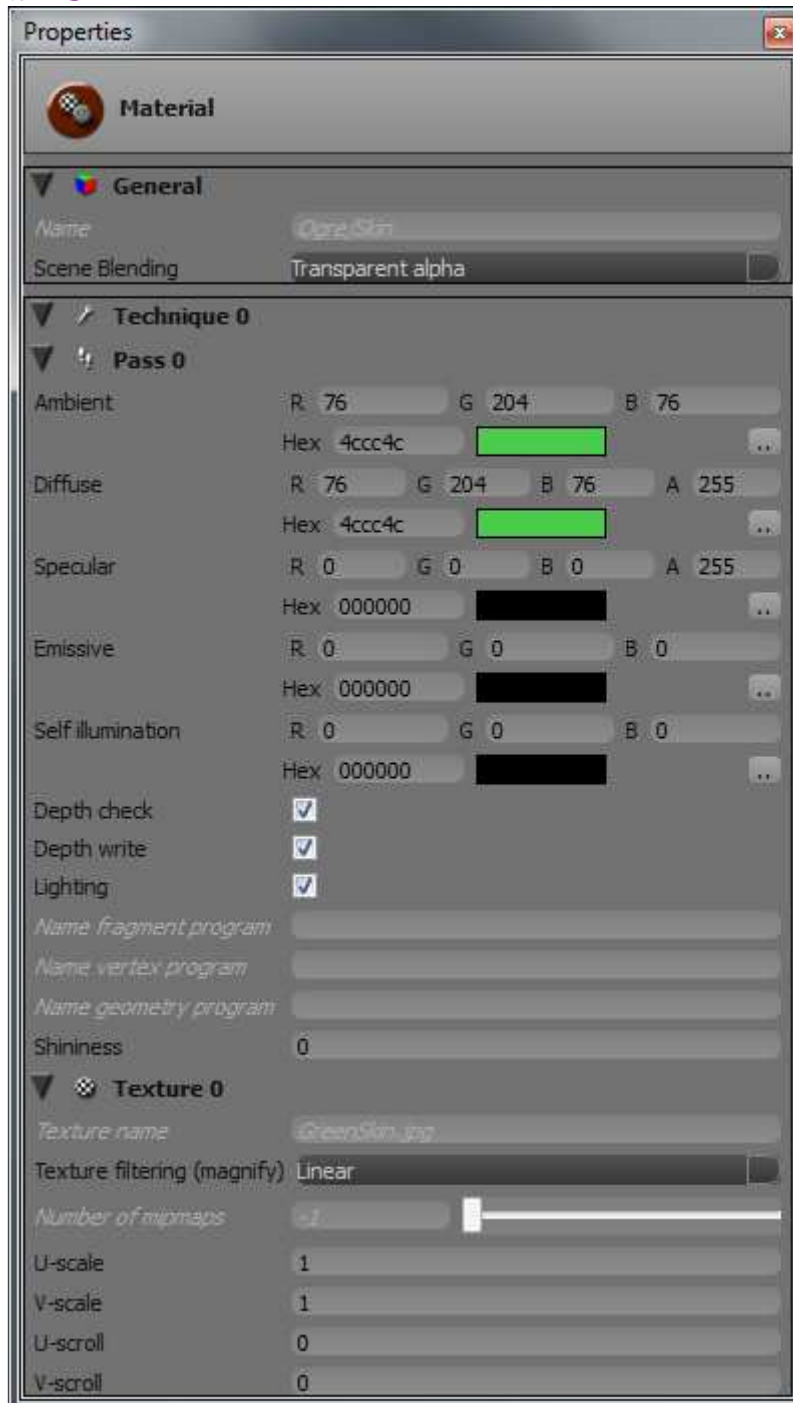
Files: ogre_asset_material.h / ogre_asset_material.cpp

QtOgreAssetLight

Properties

Light

▼ General
Name	*MainLight*
Type	Point light
Attenuation constant	1
Attenuation linear	0
Attenuation quadratic	0
Attenuation range	100000
Power scale	1
Spotlight falloff	1
Spotlight nearclip distance	0
Spotlight inner angle	29.9999991876393
Spotlight outer angle	39.9999966401228

▼ Transform
Position	X	20	Y	80	Z	50
Direction	X	0	Y	0	Z	1

▼ Color
Diffuse	R	255	G	255	B	255
	Hex	ffffff			
Specular	R	0	G	0	B	0
	Hex	000000			

Files: ogre_asset_light.h / ogre_asset_ light.cpp

Node editor library

The node editor library comes with a node editor widget and some additional components. Selecting this option adds the relevant c++ files (and icons) to the project.
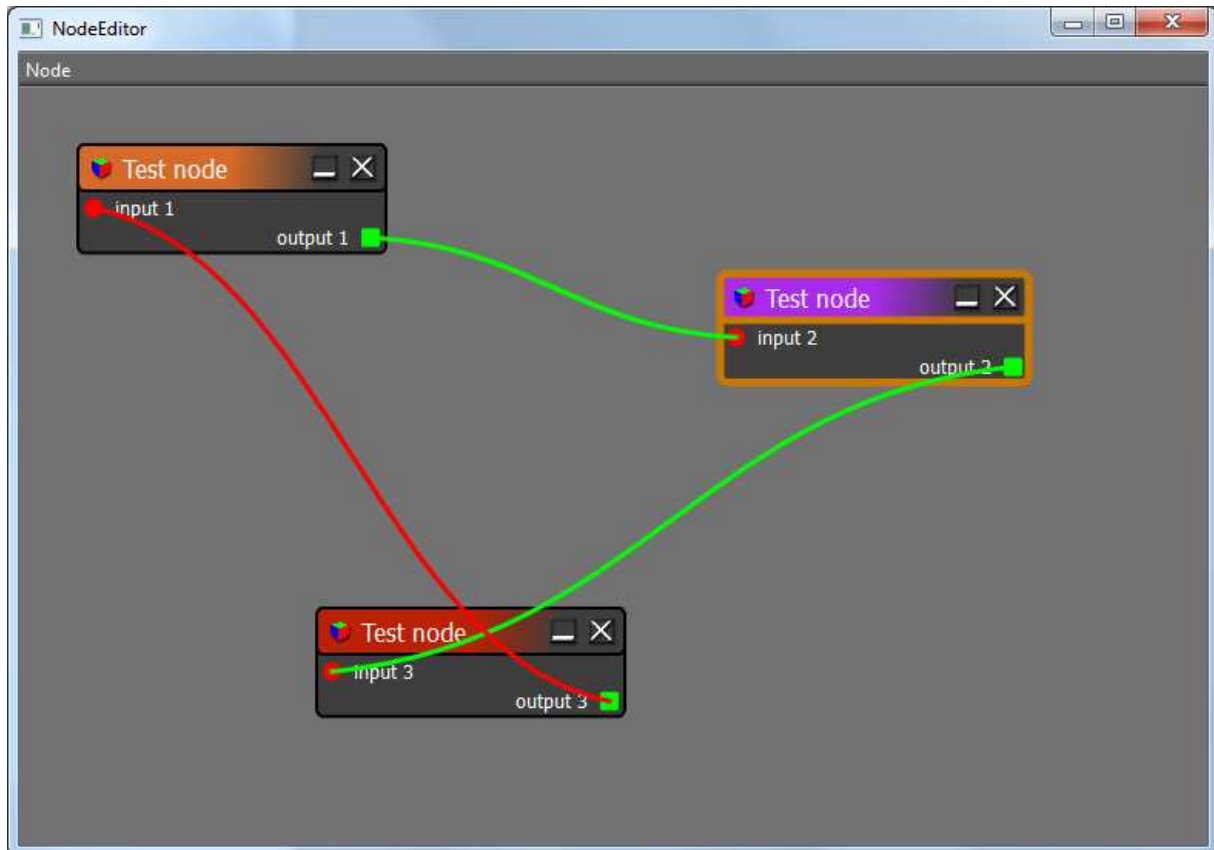
Files: node_*.h / node_*.cpp

The sample application 'Node Editor' shows how to use the node editor. Also the 'Simple Material Editor' application makes use of the Node editor library.

Tools library

The Tools library includes various widgets, such as a gradient editor and a texture selection widget.

Gradient widget

Create a gradient (QtGradientWidget), which can be exported to a vector of colors. See demo application '*Tools*' for an example.
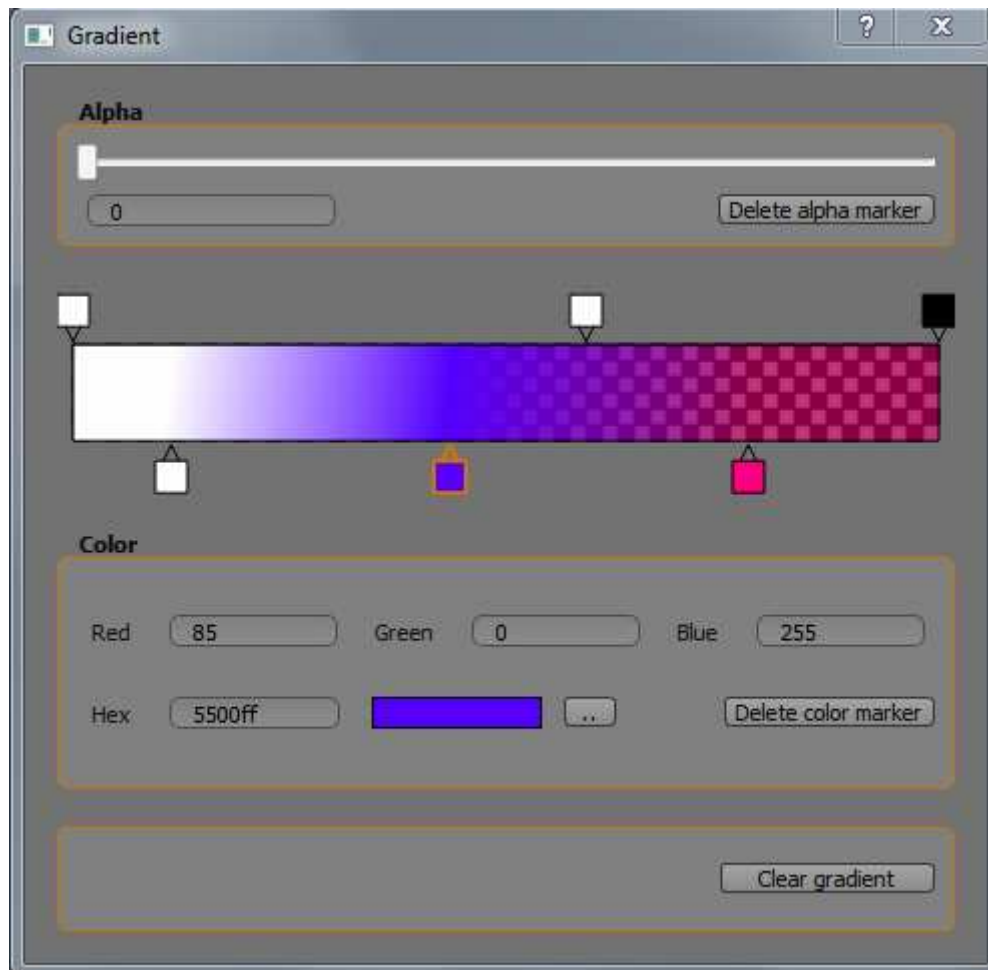
Files: tool_gradient*.h / tool_gradient*.cpp

GL Sphere widget

Create a textured sphere in a GL widget. The GL Sphere widget (QtGLSphereWidget) is used in the extended texture widget, but is can also be used in a custom widget. The textured sphere has a basic vertex- and fragment shader, without any fancy lighting, but it can be easily extended with a more complex shader.
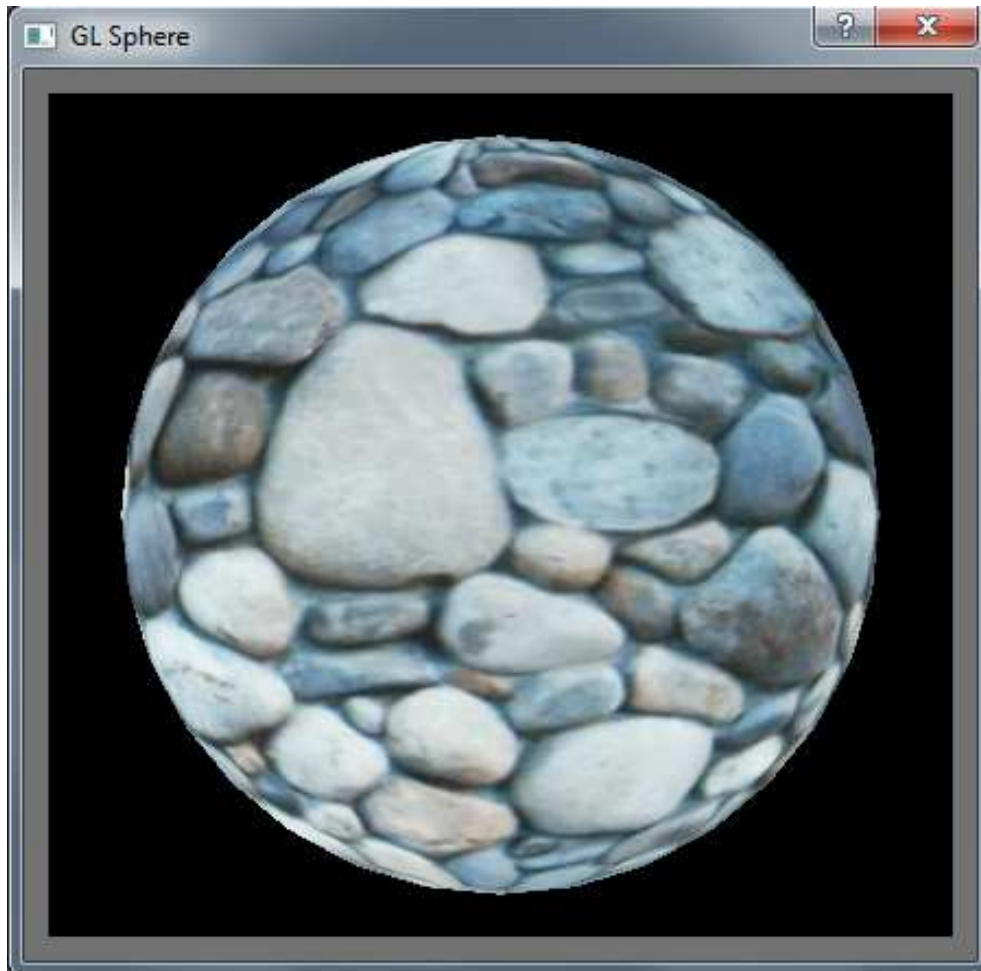
Files: tool_glspherewidget.h / tool_glspherewidget.cpp

Simple Texture widget

Texture widget (QtSimpleTextureWidget) without name. See demo application '*Tools*' for an example.
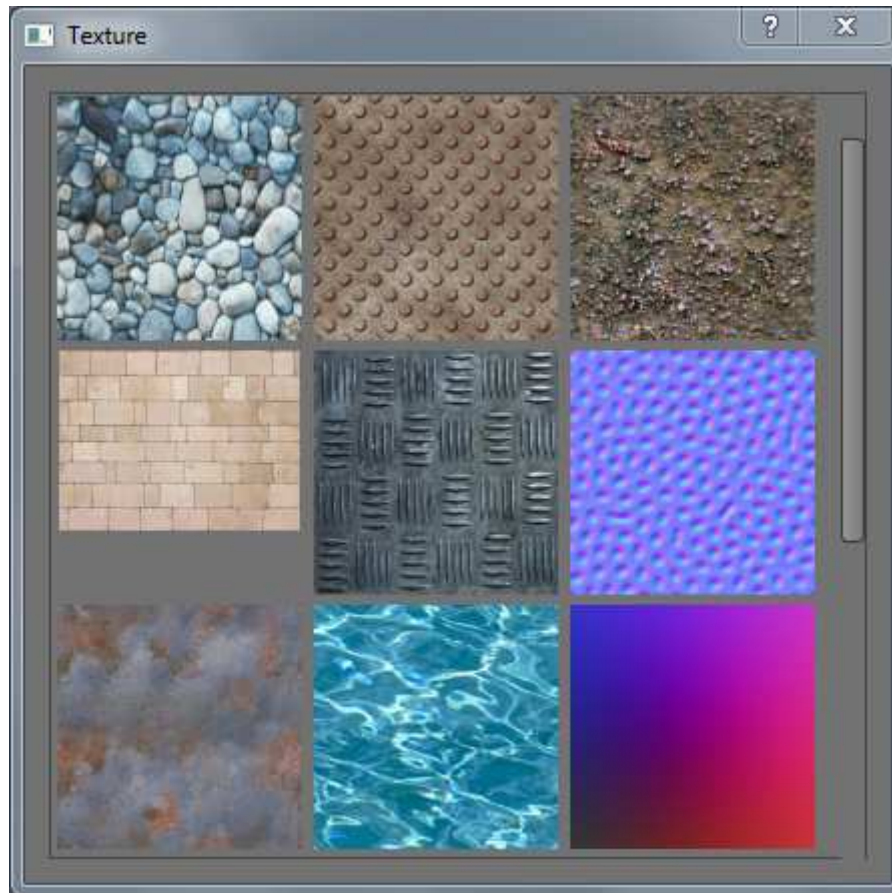

```
Files: tool_simple_texturewidget.h /
       tool_simple_texturewidget.cpp
       tool_simple_texturemodel.h /
       tool_simple_texturemodel.h
```

Default Texture widget

Texture widget with name (QtDefaultTextureWidget). See demo application '*Resources*' for an example.
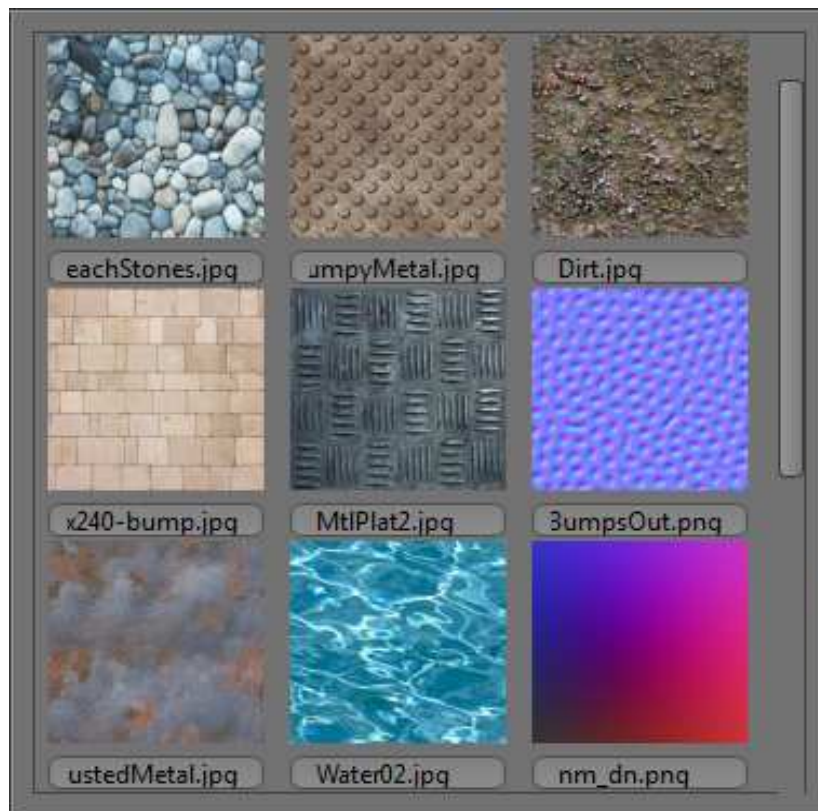
Files: tool_default_texturewidget.h /
*   tool_default_texturewidget.cpp*

Extended Texture widget

Texture widget with textures mapped on a sphere (QtExtendedTextureWidget). See demo application '*Tools*' for an example.
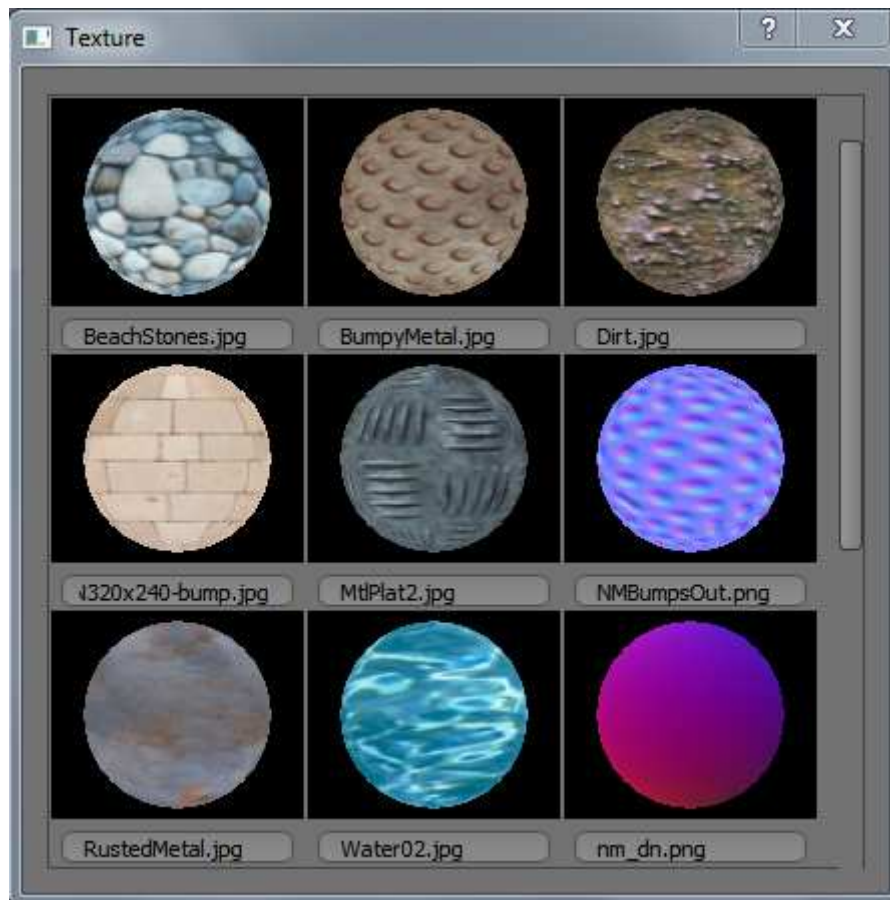

```
Files: tool_extended_texturewidget.h /
       tool_extended_texturewidget.cpp
       tool_glspherewidget.h /
       tool_glspherewidget.cpp
```

Audio widget

Widget (QtAudioWidget) that provides a list of audio assets (files). By means of a context menu (right mousebutton) or a double mouseclick the audio assets can be played.
See demo application '*Resources*' for an example.

Files: tool_audiowidget.h / tool_audiowidget.cpp

Generic asset widget

Widget (QtGenericAssetWidget) that provides a list of undefined assets (files). It is similar to the default/extended Texture widgets and the Audio widget, but for other asset types, for example meshes, materials or scripts. Depending on its settings, double clicking opens a text viewer. This makes it very suitable for scripts.
The Generic asset widget can be customized by setting a suitable icon, associated to the type of asset.

Files: tool_generic_assetwidget.h /
*   tool_generic_assetwidget.cpp*

Layer widget

The layerwidget (QtLayerWidget) is used to create a list of layers to which assets can be assigned. By means of a right mouseclick a context menu is displayed. The contextmenu can be enabled and disabled. This also applies to individual menu-items.

The menu has a few items:
- Create empty layer
- Delete (selected) layer(s)
- Rename the selected layer
- Make all layers visible

See demo application '*Tools*' for an example.
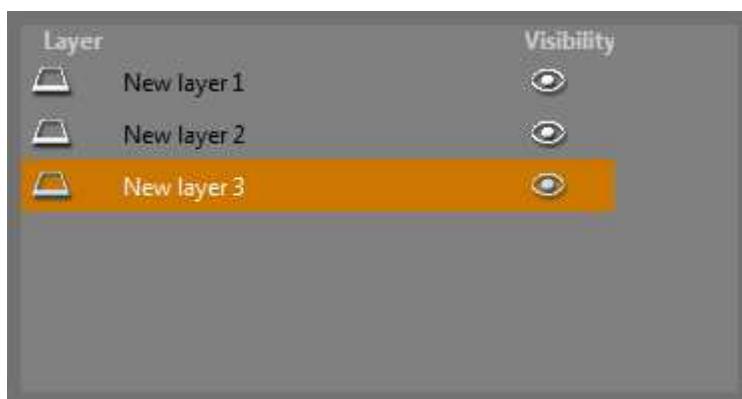
Files: tool_layerwidget.h / tool_layerwidget.cpp

Sceneview widget

The Sceneview widgets (QtSceneViewWidget) contains a list of assets, grouped by type. See demo application '*Tools*' for an example.
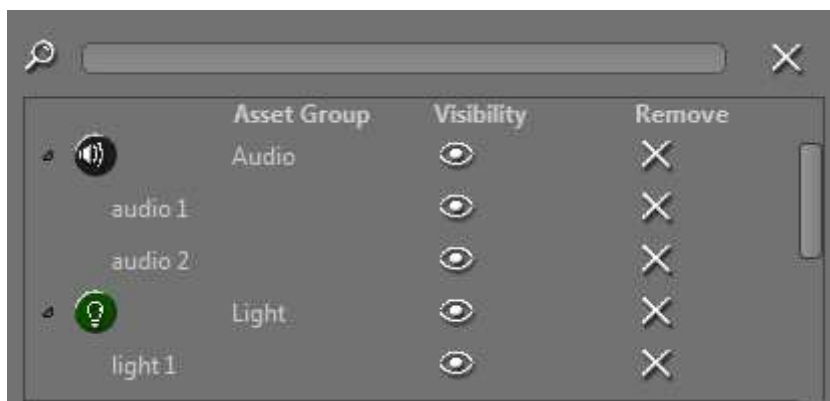
Files: tool_sceneviewwidget.h / tool_sceneviewwidget.cpp

Layered Sceneview widget

The Layered Sceneview widget (QtLayeredSceneViewWidget) is a helper widget, consisting of a Layer widget and a Sceneview widget. Both widgets interact with each other. Groups and assets can be dropped onto a layer (dragged by means of another Sceneview widget) and show up in the Sceneview widget. See demo application '*Tools*' for an example.
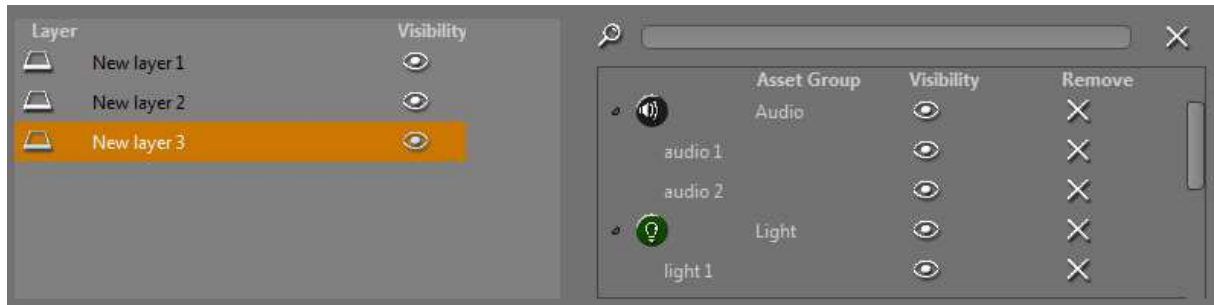

```
Files: tool_layered_sceneviewwidget.h /
       tool_layered_sceneviewwidget.cpp
       tool_sceneviewwidget.h / tool_sceneviewwidget.cpp
       tool_layerwidget.h / tool_layerwidget.cpp
```

Resourcetree widget

The Resourcetree (QtResourceTreeWidget) widgets contains a list of assets, grouped by type / subtype.
By means of a right mouseclick a context menu is displayed. The contextmenu can be enabled and disabled. This also applies to individual menu-items. See demo application '*Resources*' for an example.
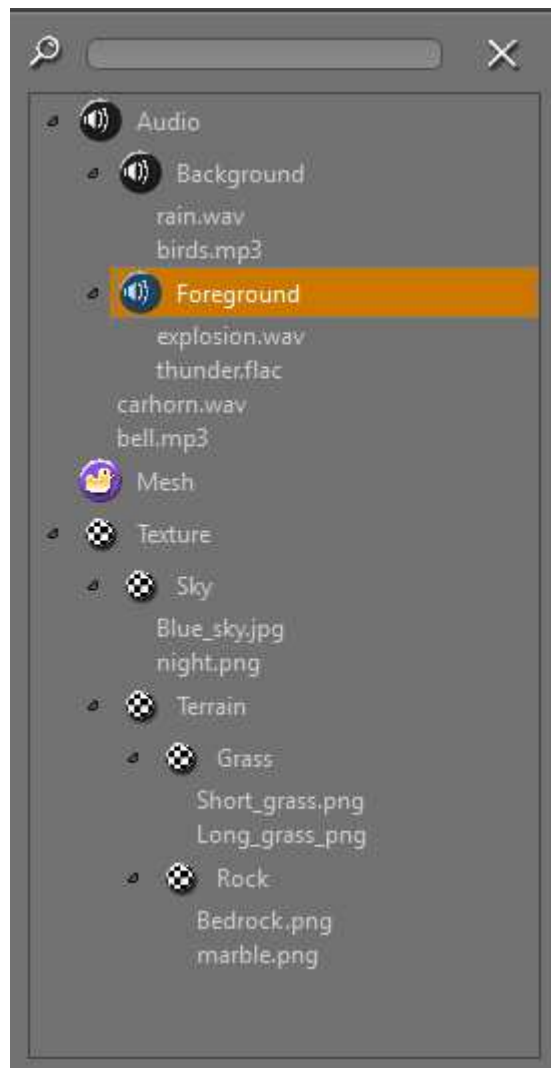

```
Files: tool_resourcetree_widget.h /
       tool_resourcetree_widget.cpp
```

Resource widget

The Resource widget (QtResourceWidget) contains a list of assets, grouped by type. On the right side, the assets are displayed as icons. See demo application '*Resource*' for an example.
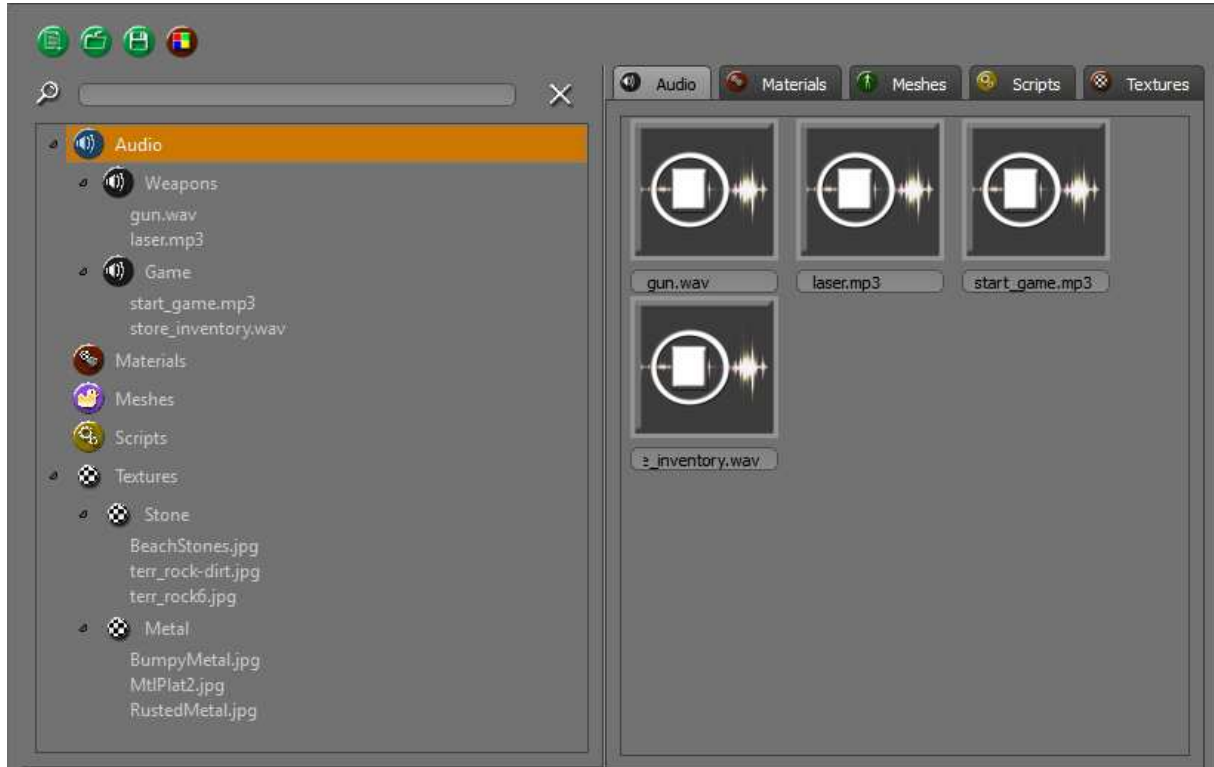
Files: tool_resourcewidget*.h / tool_resourcewidget*.cpp

QtResourceWidget is located in files *tool_resourcewidget.h* and *tool_resourcewidget.cpp*.

Sample applications

Node Editor

The Node Editor is a sample application that demonstrates the Node Editor widget (`QtnodeEditor`). It is located in the directory 'samples/NodeEditor'.

Tools

This sample application shows the widgets of the Tools library in action. The Tools application is located in the directory 'samples/Tools'.

Resources

The 'Resources' sample application shows the widgets of the Tools library in action. The Resources application is located in the directory 'samples/Resources'. The application shows the use of the resource widget `QtResourceWidget` (which is located in files *tool_resourcewidget.h* and *tool_resourcewidget.cpp*). This widgets itself is a high-level widget, which also contains sub elements.


- `QtResourceWidget` is a widget that manages basic resources (assets).
- `QtResourceMain` is a `QMainWindow`, encapsulated in the `QtResourceWidget`.
- `QtSourcesDockWidget` contains the tree structure of the resources (groups and assets).

- `QtAssetsDockWidget` is a tab-based representation of the assets per category (audio, materials, etc.).
- The function of `QtCollectionsDockWidget` is to contain 'libraries' that can be added to the `QtSourcesDockWidget`. This class is not implemented (and therefore not shown), but it is added for future purpose.

Simple Material Editor

To see the node editor and the asset widgets in action, the Simple Material Editor is created, which works with version 1.9 of Ogre3d. The Simple Material Editor application is located in the directory 'samples/SimpleMaterialEditor'.
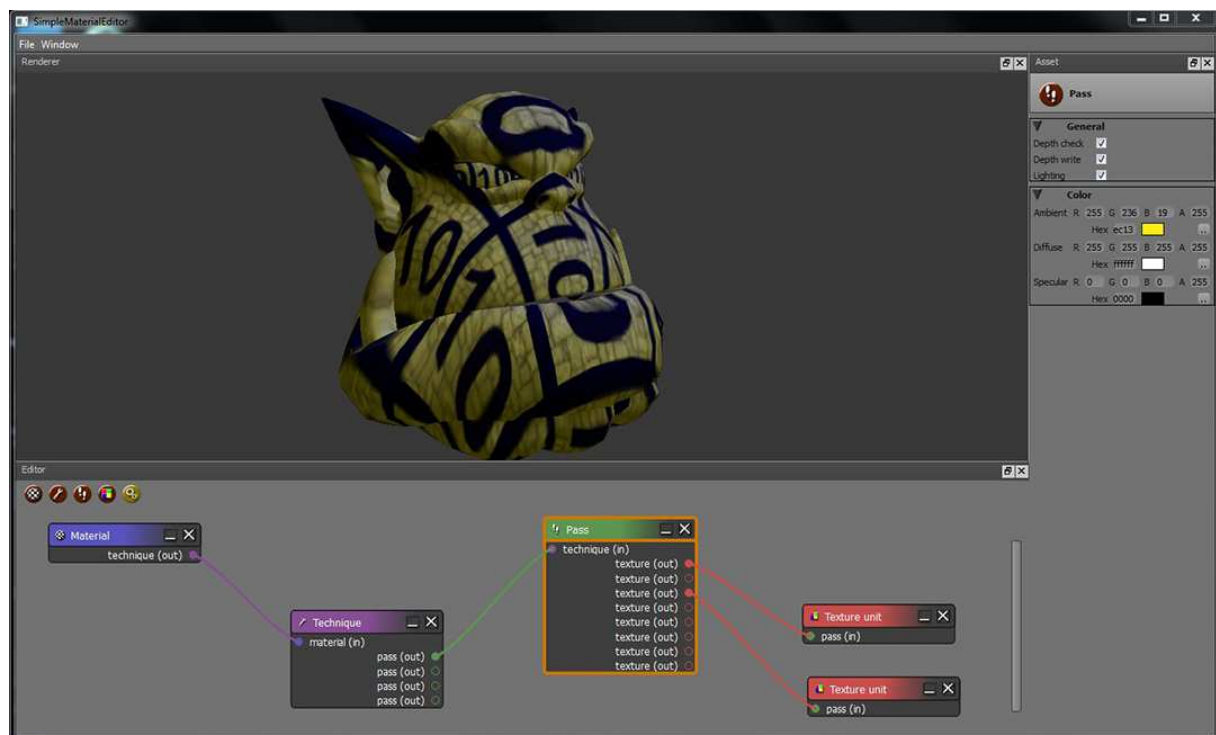
Let us analyse the application:

- The framework was generated by the Magus application. It contains the main.cpp and mainwindow.cpp / mainwindow.hpp and three docking windows:

asset_dockwidget.cpp/.h	Handles the asset widgets
editor_dockwidget.cpp/h	Contains the node editor code
renderer_dockwidget.cpp/.h	Contains the Ogre 3d rendering code

Some additional code was added, to get a working application.

- The asset library is included. These are all the files – except for asset_dockwidget.cpp/.h – with prefix 'asset_'. These files are generated and used as is.

- The node editor library. These are all the files with prefix 'node_'. These files are generated and used as is.

- The Ogre widget / rendermanager. These are all the files with prefix 'ogre_'. These files are generated and used as is.

- The remaining files are the ones, which were not generated by the Magus application and are added by hand. These files all have the prefix 'sme_'

EditorDockWidget – editor widget

In the constructor of the EditorDockWidget (in editor_dockwidget.cpp/h), the node editor widget (QtNodeEditor) is created:

```
// Create the node editor widget.
mNodeEditor = new Magus::QtNodeEditor(this);
connect(mNodeEditor, SIGNAL(nodeRemoved(QtNode*)), this, SLOT(nodeDeleted()));
connect(mNodeEditor, SIGNAL(nodeSelected(QtNode*)), this, SLOT(nodeSelected(QtNode*)));
mInnerMain->setCentralWidget(mNodeEditor);
```

The nodeSelected 'slot' of the EditorDockWidget is used to catch the nodeSelected 'signals' of the QtNodeEditor. Every time a node (QtNode) is selected, the corresponding asset is displayed in the AssetDockWidget (asset_dockwidget.cpp/h).

A small code excerpt from the nodeSelected 'slot' function:

```
void EditorDockWidget::nodeSelected(Magus::QtNode* node)
{
...
    if (node->getTitle() == NODE_TITLE_MATERIAL)
    {
        Magus::QtAssetMaterial* assetMaterial = mParent->mAssetDockWidget->mAssetMaterial;
        assetMaterial->setObject(static_cast<Magus::QtNodeMaterial*>(node));
        mParent->mAssetDockWidget->setAssetMaterialVisible(true);
    }
...
```

There are 4 derived QtNode classes for material, technique, pass and texture unit and are included in:
> sme_node_material.cpp/.h,
> sme_node_pass.cpp/.h,
> sme_node_technique.cpp/.h and
> sme_node_texture_unit.cpp).

These node classes are displayed on the QtNodeEditor 'scene' and contain their own attributes for material, technique, pass and texture unit. These attributes are used later to create the actual Ogre material (old materials; not the new 2.1 materials). The values of these attributes are manipulated by the asset widgets

In addition, the SME application also includes 4 asset classes that represent the material-, technique-, pass- and texture unit- properties (assets). These classes are included in files:
> sme_asset_material.cpp/.h,
> sme_asset_pass.cpp/.h,

sme_asset_technique.cpp/.h and
sme_asset_texture_unit.cpp/.h

The asset classes are displayed as widgets in the `AssetDockWidget` window.

EditorDockWidget – node creation

New nodes are placed on the editor widget by means of the toolbar buttons. The buttons activate the functions

EditorDockWidget::doMaterialHToolbarAction
EditorDockWidget::doTechniqueHToolbarAction
EditorDockWidget::doPassHToolbarAction
EditorDockWidget::doTextureHToolbarAction

EditorDockWidget – material generation

Function *EditorDockWidget:doCogHToolbarAction* creates the actual Ogre material. It uses the attributes of the nodes.

AssetDockWidget – asset creation

Asset widgets are created in the constructor of the `AssetDockWidget`.

Connection policies

If a port (`QtPort`) is created in a node (`QtNode`), there is an option to limit which other ports are allowed to connect. When a port is created, a port type (`QtPortType`) must be provided. This port type can be configured to define which connections are possible with other ports. An example:

```
// Define the connection policy
QtTechniqueOutPortType techniqueOutPortType;
QtMaterialInPortType materialInPortType;
techniqueOutPortType.addPortTypeToConnectionPolicy(materialInPortType);
```

In this example , *techniqueOutputPortType* only allows that connections with ports of type `QtMaterialInPortType` are allowed.

Magus – points of attention

- Sometimes, the build option from the Magus menu does not always create/copy all files. Try to build again if that happens. This is probably, because the file stream is not flushed properly. This is not a Magus bug.
- After a project is build again with Magus with some changed settings, compilation wit Qt Creator gives a link error. Delete the compilation dir (e.g. *build-Layout2test-Desktop_Qt_5_3_MSVC2010_OpenGL_32bit-Debug*) and compile again. This is not a Magus bug.
- Beware of namespace issues if you use signal/slots. If not properly used, the signals emitted from the Magus components are not received by the application; this is a known Qt 'issue'.