_2REAL CODING 101

Version 0.1alpha

http://www.cadet.at

GENERAL

_2Real is an open-source C++ multi-threaded data flow framework which was conceived to allow for the development of general purpose visual programming languages and for reusable software modules in the form of dynamically loadable plugins.

Hence _2Real is an API for developing standardized reusable software blocks with C++ which can be dynamically loaded in other projects and furthermore it is a runtime engine which allows to connect those reusable blocks together in the fashion of dataflow programming. So the users/coders can build up functionality from simple base blocks and data flows to solve complex tasks and save them again as XML file for later reuse.

With the help of you reading this guide _2Real might grow in the future and hopefully will allow for the creation of real time interactive applications in the area of image and live video processing, sensor input, 2d and 3d camera tracking...

We wish you happy creative coding.

PREREQUISITES

Note: For now _2Real has been tested and deployed with Visual Studio Ultimate as 32bit Windows software. At a later stage of the project 64 bit and multiplatform compatibility should follow.

BUILDING _2REAL

1. _2Real uses http://pocoproject.org/ please download and install it. (right now it comes with the svn externalLibs folder)
2. Set the environment variable _2REAL_DEPENDENCIES_DIR to the start path of the external libs used (right now the externalLibs folder of the repository) in your system (refer to this guide on how to set an environment variable http://www.itechtalk.com/thread3595.html)
    a. Note: environment variables in Visual Studio are only recognized after a restart of the program!!
3. Open _2Real\kernel\build\vc10_2Real.vcxproj and batch build the two win32 libraries (one release, one debug), or if you don't have batch build feature (Visual Studio Express) choose the targets manually and build them.
4. After you were successful building the _2RealFramework you should be able to find the built libraries in the _2RealFramework/lib folder.

BUILDING BUNDLES AND UNIT TESTS

Bundles can be found in the directory _2RealFramework/bundles, and in there you find the directories core and experimental. Where bundles inside core reflect a stable tested version, and bundles in experimental haven't got accepted to the core yet due to stability or other reasons.

1. To build the bundles successfully you have to have built the _2Real Framework beforehand (see above).
2. Set the _2REAL_DIR environment variable to the main path of your _2Real installation path, and be sure you have set the _2REAL_DEPENDENCIES_DIR environment variable
3. Optional: If you want to build the unit test samples which are dependent on Qt you have to install Qt 4.8.2 and the Visual Studio Add In http://qt-project.org/
4. Go to the directory of the bundle you want to build, open the project file and build

_2REAL PROGRAMMING CONCEPTS

_2Real consists of 5 main conceptual elements: Bundles, Blocks, Inlets, Outlets and Links. Those elements can be found in most dataflow and visual programming languages and are sometimes named differently e.g. nodes, boxes, plugins, etc.

_2Real is a multithreaded environment so be very cautious with shared resources you use.

_2Real is written with modern C++ paradigms (templates, exceptions …). Please use try and catch whenever you use _2Real functionality to catch and handle exception generated by _2Real accordingly.

Note: If you program a bundle which should be considered as a core bundle please follow this naming convention:

BundleName_32.dll respectively BundleName_32d.dll (debug)

BUNDLES

A bundle can be thought of as a group or container of reusable functional blocks which most times belong to a common special field e.g. Image Processing. In _2Real a bundle is a dynamic shared library (.dll, .so, .dynlib), which is loaded and used at runtime by a client application, which then can use the contained function blocks (see below). If a bundle is dependent on 3rd party libraries, be aware that you can whether import them as static or dynamic libraries. To use dynamic 3rd party libraries is preferred.

Bundles define some metadata for the client application (mainly used for documentation and retrieval purposes) and more important they define the interfaces for an arbitrary number of blocks which are the functional units in _2Real.

Bundles have to implement a special function signature so _2Real is able to load and initialize them, be sure to write it like this:

```
void getBundleMetainfo( BundleMetainfo& info )

{

.. // in here you define metainfo about the bundle and the interface for the exported blocks

}
```

Sometimes blocks of a bundle want to share a limited resource, an api context or a device context, so all instances of the blocks are able to work with the same resource (compare Singleton). This can be done with a special ContextBlock in _2Real.

Note that every bundle only can have one single ContextBlock and because you are operating in a multithreaded environment, you as a programmer have to take care of synchronisation issues yourself in the use of your resource you share in the ContextBlock. So many blocks of the same time can be instantiated in a system and operate via a reference on the functions and member variables of a contextblock. The functions which are critical to race conditions must be locked by mutexing (this is an advanced topic, please refer to the sample implementation of the DeviceManager Tutorial).

BLOCKS

Blocks are the actual heart where the functionality has to be coded. They inherit from a base class and have to implement a minimum interface. Ideally you separated blocks in a header and a .cpp implementation file, and do so for every block.

```
class MyBlock : public Block
{
void setup( _2Real::bundle::BlockHandle &context );
```

This method is used to set default values for a valid start configuration of used variables and for a clean reset of the block.

More important this method hands you a reference to a context with which the programmer can interact with the inputs and outputs as defined in the bundle. You should define your Inlet and Outlet handles here as member variables.

```
void update();
```

This is the calculation update loop which will be called according to a set FPS or if certain policies regarding the Inlets are met.

```
void shutdown();
```

Here resource should be unloaded and memory should be cleaned up.

```
};
```

INLETS

Inlets are named input variables of various datatypes which are available in _2Real (image, sound, base numeric types … see Datatypes below). Their handles can be retrieved in the block setup function via the BlockHandle reference. From this we then can retrieve our InletHandles and save them to member variables for use in the update function.

```
InletHandle                                    m_DeviceIndexHandle;
```

```
class { …
	InletHandle	m_MyInletHandle;
};

void MyBlock::setup( BlockHandle &block )
{
	try
	{
		m_MyInletHandle	= block.getInletHandle("MyInlet");

	…
```

On the InletHandle we can then get the actual value and calculations can be produced in the update function.

Inlets are mainly set by other blocks outlets, so a data flow network can be established.

You can imagine a process chain in terms of filter operations. An easy example would be a CameraBlock retrieving a camera image and supplying it via an Outlet of type Image to another's block inlet of type image. This other block would take the image and might invert it in the update and output it via its own image outlet.

An application, the client can set Inlet values as well directly.

Another interesting consideration for advanced users is that the arriving dataflow to the inlets is buffered by default, but this behaviour can be switched off, respectively the buffer size can be set and matched to 0. Buffering in a multithreaded environment is useful for not loosing data due to different time lags and update rates of blocks. The buffer are implemented as ring buffers and are of course limited, so at some point you might want to consider optimizing the setup of your dataflow and set different update rates or update policies.

OUTLETS

Outlets are similar to inlets: they serve as named output variables of a _2Real data type. Outlets are set in the update function of the block via an outlet handle. Another block which is linked to an outlet is invoked with new data every time the update of the first block finishes. The programmer has the ability to discard an outlet means in an update loop nothing has changed and so no other blocks have to be invoked.

In a client application outlets can be read via registered callbacks. The programmer has the possibility of registering a general callback which gives a vector of all valid which haven't been discarded and updated by one block.

```
m_CameraBlockHandle.registerToNewData( *this, &BlockUnitTestWidget::receiveData );
```

Another possibility is to register a callback per outlet. So this callback is only invoked when this certain outlet has been updated.

Note: Updated doesn't refer to changed value, the system is unaware if a value changed, as this would be critical to check for large image datatypes so our design considerations were that the programmer of the block has to tell whether it should be considered as ne data or he/she should discard it explicitly to tell that this data is the same as in the last update.

LINKS

Links are used to connect outlets to inlets and therefore allow blocks to connect together to achieve higher functionality and generate a processing chain. _2Real allows outlets and inlets to be connected in various ways even cyclic connections of inlets and outlets are allowed.

Graphics to come….

DATATYPES IN _2REAL

SIMPLE TUTORIAL / WALK THROUGH

DEVICEMANAGER TUTORIAL