REALFLOW

C++ SDK USER'S MANUAL 1.0

NEXT LIMIT TECHNOLOGIES

1 INTRODUCTION

C++ SDK is the new programming interface in RealFlow. In version 4.0. our users could enjoy using the Python language to develop scripts to be run inside RealFlow. This new SDK brings all the functionality of the Python interface combined with the speed of the C++ language to deliver a powerful environment that allows the development of really fast plug-ins. Plug-ins that would run at almost the same speed as their counterparts in RealFlow.

The Python SDK has been improved to allow more access to the RealFlow Engine. And the C++ counterpart has the same functionality, though much faster.

2 RELATIONSHIP WITH THE PYTHON SDK

The Python API and the SDK API are basically the same, with some small differences due to the idiosyncrasy of the two languages. For instance,

- Python uses a List object. whereas the C++ SDK uses a std::vector Object.
- Some methods in Python return a variable (that could be of several different types) - the C++ SDK has to return a wrapper class that encapsulates the different types that can be returned.

3 LIMITATIONS

Although you can do a lot of different tasks with the SDK, you can't do everything and there are limits it is worth knowing about. There are places where you can't perform certain operations, or where it's unwise to do so. Although not exhaustive, the main SDK limits are listed below:

- Within an event, realwave, daemon or fluid plug-in, you cannot create or load a new scene, jump to other frames, or simulate individual or ranges of time substeps or frames. These can only be done within a batch plug-in script.The reasons for this should be obvious, since jumping to an entirely new scene in mid-simulation doesn't make a lot of sense and is likely to confuse both you and your workstation.

- There are a variety of parameters that you cannot set via plug-ins, such as preferences, fps and adaptive stepping settings. Most of these are not very important to getting results, and the ones that are related to simulations (e.g., adaptive steps) are not accessible on purpose, to avoid confusing the solver.

- For those plug-ins that allow multithreaded execution beware that only a fraction of the methods are thread-safe. These are marked in the documentation. The rest of the methods may cause instability or incorrect behavior in your program.

As the SDK opens up a lot of the internals of the engine, that allows the development of instability in the system by the added code, leading to the possibility of crashes related to bugs in the C++ code developed by the user.

The RealFlow team has tried to prevent as many crashes as possible. We have tried hard to catch errors produced in the plug-in code, and instead of crashing the whole application just inform the user that an error occurred. But this is not possible in all the cases. So we recommend that plug-in developers complete thorough testing of their plug-ins before releasing them to the general public.

4 INTRODUCTION TO THE DIFFERENT PLUG-INS

The way of enhancing RealFlow with the C++ SDK is by means of plug-ins. A plug-in can represent different objects in RealFlow. For instance you could develop a new Daemon plug-in to remove particles that match certain conditions, such as: velocity higher than a certain number, pressure in certain range and so on. This new Daemon will then appears in the menu of daemons. You just have to select it and add it to the scene. It will behave the same way as any other Daemon. You can add any number of these new daemons to the scene.

There are 5 types of plug-ins that you can add:

- Daemon plug-in
- Object plug-in
- ParticleSolver plug-in
- Wave plug-in
- Command plug-in / Event plug-in

Let´s have a quick look to them:

Daemon plug-in
When you program a new daemon plug-in you must specialize a base class where you will implement your desired behavior. With a daemon plug-in you can apply forces to the different dynamic entities in RealFlow; you can also remove particles in the case of particle-based fluids and even set a velocity field for a mist.

Wave plug-in
In a similar fashion to the Daemon plug-ins you can write the body of some methods in the Wave Plug-in class. In this plug-in there is only one method, which allows you to calculate the 3D displacement of all the points of the RealWave mesh respective of their initial positions.

Particle Solver plug-in
This plug-in allows you to define the behavior of the particles. When the plug-in is loaded you will be able to choose it from the drop-down menu of particle types.

Object plug-in
You can define your own object with your own properties. For version 5 of RealFlow the functionality of this plug-in is very limited; for instance you can't create geometry for your own object just set and get properties. In future versions of RealFlow the capabilities of this plug-in will be increased.

Command plug-in
As its name states this is used to execute a general set of instructions in the same way as as a Python script does. It has just one method (run) that will be called whenever the command is executed.

Event plug-in
This is similar in nature to the Command Plug-in. It is intended to be added as a set of codes to be executed at precise times in the simulation pipeline. You can add this type of plug-in to the new Simulation Events window. This window replaces the old Events Script window.

5 PLUG-IN TYPES

Let's have a brief description of the methods that the developer needs to implement when creating a new Plug-in.

5.01 All Plug-ins

```
// getNameId
virtual std::string getNameId() const = 0;
```

Name that identifies this plug-in. It is recommended to choose a name that will not likely clash with other plug-ins. For instance you could name your own Graviton daemon like "MyCompany_Graviton" where MyCompany stands for the name of your company.

```
// getClassId
virtual int getClassId() const = 0;
```

This is a unique worldwide ID for your plug-in. Using the tool provided with RealFlow you will generate a unique worldwide number for your plug-in. And this method should return that number.

```
// Get plug-in copyright policy
virtual std::string getCopyRight() const = 0;
```

This should return a line with information about the copyright holder of this plug-in. This information will be shown in the User Plug-ins manager window.

```
// Get plug-in long description
virtual std::string getLongDescription() const = 0;
```

This should return a line with a long description of what this plug-in does.

```
// Get plug-in short description
virtual std::string getShortDescription() const = 0;
```

This should return a line with a short description of what this plug-in does. This will appear in the User Plug-ins manager window.

```
// Is multithreading
virtual bool isMT( void ) const { return NL_true; };
```

This tells the RealFlow engine that this Plug-in is multithreading. That means that RealFlow will call the right method for those methods that have a dual implementation, like single-threading and multi-threading. If isMT returns false then the single-thread version will be called. By default all plug-ins are considered multi-threaded.

5.02 Daemon

```
/// applyForceToEmitter:
///
/// This function is called by the simulation engine when
/// external
/// forces should be applied to the
/// particles in the emitter.
///
/// @param [in/out] plgThis  / Sdk Daemon ( Owner of this
/// Daemon
///                                                  Plug-in )
/// @param [in/out] emitter / Sdk Emitter which particles
/// will be
///                                 affected by external ///
/// forces
/// @param [in] iter / Iterator over the Emitter´s Particles
//
virtual void applyForceToEmitter
   ( Daemon* thisPlg, PB_Emitter* emitter, PB_
Emitter::iterator iter )
   {
   applyForceToEmitter( plgThis, emitter, 0, iter );
   }
```

Single-Threaded version. It is called by the simulation engine when external forces should be applied to the particles in the particle-based emitter. For you convenience you get an iterator that helps you to iterate over all the particles of the emitter.

```
//-------------------------------------------------
//  Function: applyForceToEmitter
//  This function is called by the simulation engine
//  when external forces should be applied to the
//  particles in the emitter.
//  It is only used if the plug-in is declared
//  MultiThreading through
//  the isMT method.
//-------------------------------------------------
virtual void applyForceToEmitter
  ( Daemon* thisPlg, PB_Emitter* emitter, int nThread,
    PB_Emitter::iterator iter                          ) =
0;
```

This is the Multi-Threaded version of the above method. This method gets called by the simulation engine when is the time to compute the external forces that this Daemon may want to apply to the particles in the given Emitter. In a similar way to the Daemon plug-in the RealFlow engine creates by default a partition in n subsets of particles associated to the given Emitter. N also represents the number of threads that will be used to compute the forces. So this method will get called n times to produce a complete calculation. Each of these calls will get a number identifying which of the n threads is being used in the call of the method. This is given in the nThread parameter. The method will also get the default partition associated to that thread as an iterator over that partition.

```
//-------------------------------------------------
// Function: applyForceToBody
// This function is called by the simulation engine
// when external forces should be applied to the body.
//-------------------------------------------------
virtual void applyForceToBody( Daemon* thisPlg,  Object* obj
```

```
);
```

This is called by the simulation engine when external forces should be applied to a Body. It does nothing by default.

```
//-------------------------------------------------
// Function: applyForceToMultiBody
// This function is called by the simulation engine
// when external forces should be applied to the body.
//-------------------------------------------------
virtual void applyForceToMultiBody(
                            nl::rf_sdk::Daemon* plgThis,
                            nl::rf_sdk::MultiBody* obj
);
```

This is called by the simulation engine when external forces should be applied to a MultiBody. It does nothing by default.

```
//-------------------------------------------------
// Function: applyForceToMist
// This function is called by the simulation engine
// when external forces should be applied to the mist.
//-------------------------------------------------
virtual void applyForceToMist( nl::rf_sdk::Daemon* plgThis,
                               nl::rf_sdk::Mist* obj
);
```

This is called by the simulation engine when external forces should be applied to a Mist. It does nothing by default.

```
//-------------------------------------------------
// Function: applyForceToGridFluid
// This function is called by the simulation engine
// when external forces should be applied to the grid fluid.
//-------------------------------------------------
virtual void applyForceToGridFluid(
                      nl::rf_sdk::Daemon* plgThis,
                       nl::rf_sdk::GridDomain* obj );
```

This is called by the simulation engine when external forces should be applied to a grid-based Fluid. It does nothing by default.

```
//-------------------------------------------------
// Function: onSimulationBegin
// This function is called by the simulation engine
// when the simulation begins.
//-------------------------------------------------
virtual void onSimulationBegin( nl::rf_sdk::Daemon* plgThis
);
```

This is called by the simulation engine when the simulation begins. It does nothing by default.

```
//-------------------------------------------------
// Function: onSimulationResume
// This function is called by the simulation engine
// when the simulation resumes.
//-------------------------------------------------
virtual void onSimulationResume( nl::rf_sdk::Daemon* plgThis
);
```

This is called by the simulation engine when the simulation resumes. It does nothing by default.

```
//-------------------------------------------------
// Function: onSimulationStop
// This function is called by the simulation engine
// when the simulation stops.
//-------------------------------------------------
virtual void onSimulationStop( nl::rf_sdk::Daemon* plgThis
```

```
);
```

This is called by the simulation engine when the simulation stops. It does nothing by default.

```
//--------------------------------------------------
// Function: onSimulationFrame
This function is called by the simulation engine
// when the simulation starts the computation of
// the next frame.
//--------------------------------------------------
virtual void onSimulationFrame( nl::rf_sdk::Daemon* plgThis,
                                const unsigned int& frame );
```

This is called by the simulation engine when the simulation of a frame starts. It does nothing by default.

```
//--------------------------------------------------
// Function: removeParticles
// This function is called by the simulation engine
// when it is safe to remove particles.
//--------------------------------------------------
virtual void removeParticles(Daemon* plgThis,
              PB_Emitter* obj ) = 0;
```

It is called by the simulation engine when it is safe to remove particles for a particle-based fluid. The user may leave this method blank if he has no need to delete particles.

5.03 Object

This Plug-in class has no methods defined at the moment. In future version of RealFlow we will add more functionality to this plug-in type. For the moment it just can be used to get and set properties.

5.04 Particle Solver

```
// preComputeInternalForces
void preComputeInternalForces(
              rf_sdk::ParticleSolver* particleSolver,
              rf_sdk::Emitter* emitter )
```

This method gets called in a single-threaded way by the simulation engine before calling the multi-threaded method computeInternalForces. It simplifies the preparation calculations that may be needed to compute the internal forces - calculations that all threads need to have to compute the forces.

```
// computeInternalForces
virtual void computeInternalForces(
              rf_sdk::ParticleSolver* particleSolver,
              rf_sdk::Emitter* emitter,
              int nThread,
              RFEmitter_It iter )
```

This method gets called by the simulation engine when it is time to compute the ParticleSolver fluid internal forces. In a similar way to Daemon plug-in, the RealFlow engine creates by default a partition in n subsets of particles associated to the given Emitter. N also represents the number of threads that will be used to compute the internal forces. So this method will get called n times to produce a complete calculation over all the subsets of particles. Each of these calls will get the number of thread associated as the nThread parameter. The method will also get the default partition associated to that thread as an iterator over the partition.

```
// integrate
virtual void integrate(
        rf_sdk::ParticleSolver* particleSolver,
        rf_sdk::Emitter* emitter,
        int nThread,
        RFEmitter_It iter,
        float dt )
```

This method gets called by the simulation engine when the final position of the particles

must be calculated - the integration time dt that RealFlow is using is passed to the function. The same considerations about the way of calculation and use of partitions that have been explained in the above method can be directly applied to this calculation.

```
// getIntegrationTime
virtual float getIntegrationTime(
                rf_sdk::ParticleSolver* particleSolver)
```

The simulation engine will call this method to obtain the length of time of your integration step. It will use this to calculate the amount of calls it needs to do to your integration calculations during the calculation of a frame.

5.05 Waves

```
virtual void updateWave(
        rf_sdk::Wave* plgThis,
        std::vector<Vertex>& vertices,
        const std::vector<Vector>& initPosVrtxs  );
```

This method gets called by the simulation engine when is time to update this wave. The method gets the initial position of all the vertices of the wave mesh.
And it should return a 3D displacement of all these vertexes in the parameter `vertices`.

5.06 Commands

```
virtual void run ();
```

This method will get called every time the user launches the command from the GUI interface - either by choosing it with the mouse or with the shortcut key combination.

5.07 Events

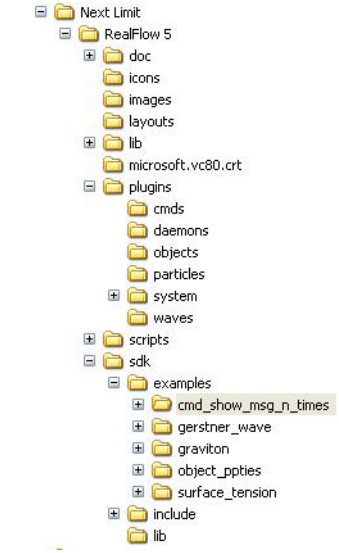
```
virtual void run ();
```

The plug-ins that are used to create a command in the GUI user interface are the same that can be used to create Event Commands. These Event Commands are added to the Pipeline of the engine through the Simulation Events window. Each Event Command will have its method run called whenever the event holder to which it belongs is fired. Have a look at the Simulation Pipeline in the Appendix B for clarification.

6 SETTING PROGRAMMING ENVIRONMENT

There are a few things you need to do before starting to code your shiny new plug-in. Depending on your environment you may have to do things in a slightly different way. But it will be mostly pretty similar.

System Variables (Windows)
The first step of course is to install the RealFlow package. Once you have done this you should have the following directory structure in your hard disk. The installation directory may vary.



F. 01 Directory structure in your hard disk

Once you have installed RealFlow the system variable RF_XXXX_PATH must have a value.
Something like *'H:|Program Files|Next Limit|RealFlow'*. This of course will be different in your system, depending on where RealFlow has been installed.
Make sure that this system environment exists and that it points to your RealFlow directory.

Setting up Development Environment
Please refer to the Appendix A to get the details about configuring a development environment from scratch. In the next sections we will use the examples provided with RealFlow, in which the development environment is already set up.

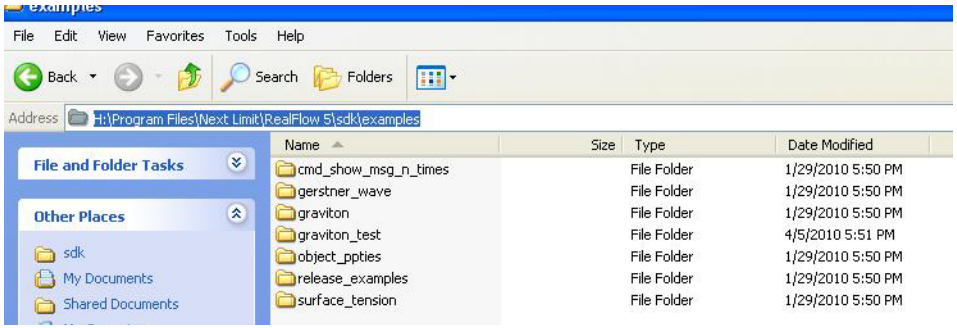
7 BUILDING A PLUG-IN

It could be a bit complex to have to set up everything that's needed in a development environment to build plug-ins. So to ease this task we provide examples for all possible types of plug-in. These examples are ready to be compiled, built and deployed easily in Windows through the Visual Studio C++ development environment. If you want to see the details of the settings used in the environment please go to the Appendix A, where details are given.
We will use one of the examples to show all the steps necessary to build and deploy a plug-in. In this case the graviton example will be used.

☑ *Imp: You need to have the Visual Studio for C++ installed in your computer.*

7.01 Copy example Graviton

We'll make a copy of the examples so we always have a pristine example template that we can use to create new plug-ins. Just go to the RealFlow installation directory and make a copy of the folder 'graviton' into a folder called graviton_test as shown below.



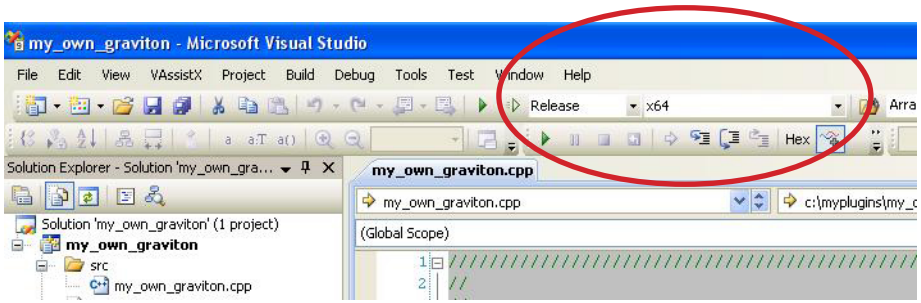
Go into the folder graviton_test and double-click the file 'graviton.vcproj'. The Visual Sudio Enviroment should open. You should see something like the window below.

7.02 Target Configuration

The plug-in can be build for 4 different configurations. These result from the combination of two possible platform targets, Win32 and X64, and two possible modes of compilation, Release and Debug.

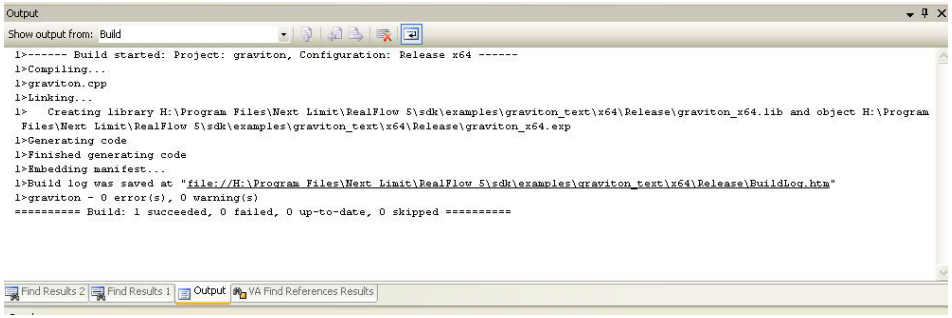
- **The platform target:** It must match the RealFlow one where you are going to deploy the plug-in. When RealFlow is installed you can choose either to install the Win32 or the x64 version.
- **Mode Type:** Generally speaking you will use the Debug mode while you are developing your plug-in and switch to Release once you have finished developing. Bear in mind that the Debug mode is to help the developer in finding errors but it is not intended to be used at deployment time. Debug mode could create huge issues with the performance of the plug-in - also it exposes information about your plug-in that you usually would prefer not to open to everybody using it.

You can choose these different target configurations in the menu bar, as is shown on the picture. If your RealFlow version is Win32 or x64 choose Win32 or X64 from the menu bar respectively. And choose Release mode.



7.03 Compiling & Linking

Go to the menu bar and choose 'Build Solution'. There should be no errors and the ouput tab should show 0 errors as shown below. The plug-in will be compiled and linked. It generates a 'graviton.dll' or a 'graviton_x64.dll' file depending on the platform you chose in the previous step.



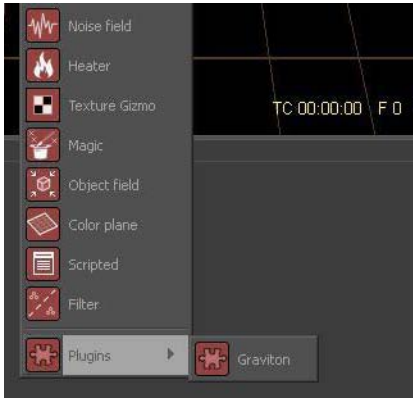
The generated Dll can be found in the directory *'$RF _ XXXX _PATH|sdk|examples|graviton_text|x64|Release'* or *'$RF _ XXXX _PATH|sdk|examples|graviton_text|Win32|Release'* if the platform is Win32.

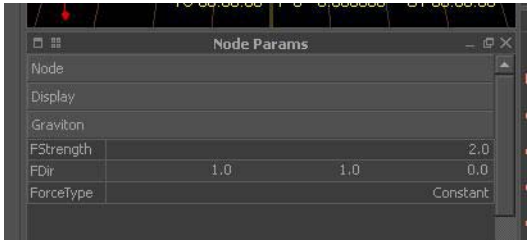
7.04 Deploying

RealFlow has a directory of plug-ins called *'$RF _ XXXX_PATH|plugins'* where there is a subdirectory for each type of plug-in. Each one of these subdirectories will be scanned recursively and all the plug-ins (.dll) found for the associated type will be loaded.
In this case we just need to copy the generated graviton_x64.dll or graviton.dll to the folder *' $RF_ XXXX _PATH|plugins|daemons'*.

7.05 Testing the Plug-in

This plug-in is similar to the standard Gravity daemon already part of RealFlow.
Launch RealFlow, add an object, for instance a cube, click on the Daemon menu, and when it pops up go to the plug-in menu. You should see the graviton daemon; select it to add it to the scene.





The picture above shows the properties of the Graviton plug-in. Play with them to affect the previously added cube. Remember that the cube needs to have the Dynamics properties set to "Rigid Body" in order for the daemon to have any effect on it.

Congratulations! You have added a new plug-in to RealFlow.

8 BUILDING YOUR OWN PLUGIN

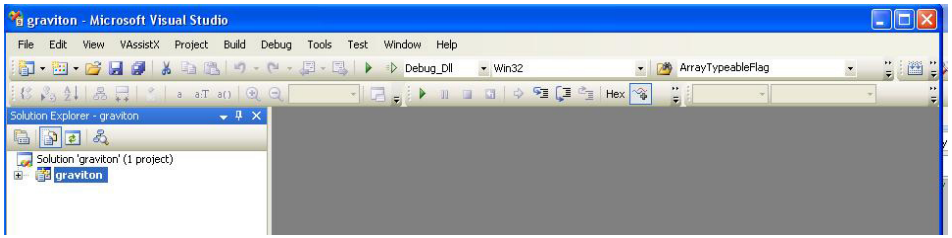
Once you know how to build one of the example plug-ins, then you are ready to build your own plug-in. The easiest way to do this is to use one of the examples as a template for the new plug-in. From there you can make the necessary changes to create your own plug-in.

The steps that are needed to accomplish this are described in detail below.
We will show how to build your own plug-in, in this case a new Daemon plug-in. You can apply these steps to all the other types of plug-ins in a similar fashion.

8.01 Copying From Example

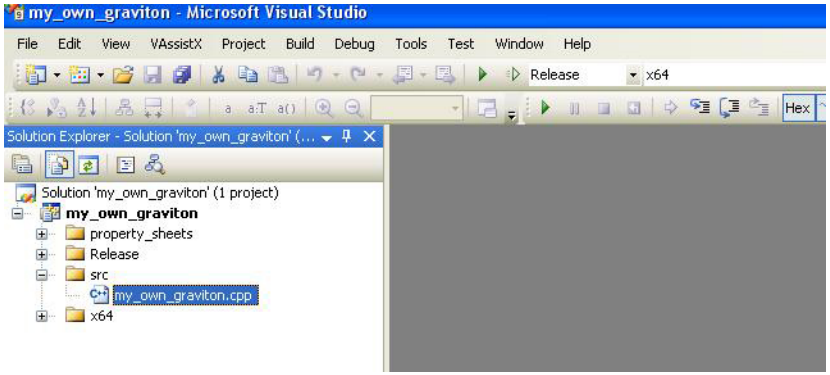
The best way to start developing a new daemon is from the Graviton example that can be found in the directory *"./examples/graviton"* of your RealFlow installation. Make a copy of the *"./examples/graviton"* folder so you always have a pristine example template that you can use to create new plugins. Name the new folder "my_own_graviton_own" as shown below:

Go into the folder my_own_graviton and double-click the file 'graviton.vcproj'. The Visual Studio Environment should open. You should see something like the window below.



8.02 Change Name project & source file

It is not strictly necessary to identify your plug-in, but it helps. Change the name of the solution from 'graviton' to 'my_own_graviton'. Change the name of the project 'graviton' to 'my_own_graviton'. And the same for the source file 'graviton.cpp' to 'my_own_graviton.cpp'. You should end up as shown below.



8.03 Creating your own plugin class

In this section we are going to modify the class graviton into a slightly different one, and in this way create a new plug-in.

Open the cpp file 'my_own_graviton.cpp'.

- Change the name of the class 'GravitonDaemonSDK ' to ' MyOwnGravitonDaemonSDK'.
- Change the name that will be shown inside RealFlow for this plug-in. Go to the method 'getNameId' and change the name being returned to 'MyOwnGraviton'.
- You will need a new classID for this plug-in. Every plug-in has a unique ID which is used to identify the plug-in. To obtain a new id you need to run RealFlow and go to the Tools menu and choose 'Generate Uuid'. This will give you back an unique ID. Now go back to the code and in the method 'getClassId' substitute the id being returned - that is the one assigned to the graviton plug-in - for the one that you just obtained.
- Change the name used in the macro RF_SDK_DECLARE_DAEMON_PLUGIN, from GravitonDaemonSDK to MyOwnGravitonDaemonSDK. This macro creates a few functions that RealFlow needs to recognize the new class as a plug-in. There is a macro for each type of plug-in. You can see them being used in the example plug-ins.

These are the minimum changes you will need to make to create your very own plug-in. Of course this plug-in will have exactly the same functionality as the graviton one, because we have not changed anything related to the behavior of this plug-in.

8.04 Change behavior

We will now change some of the properties and the methods 'applyForceToEmitter' and 'applyForceToBody' to generate a different behavior. This simply generates a force that changes over time in a way controlled by the old parameters and the new ones you have added. Just play with the parameters to see that it does in fact do something different to the 'Graviton' Daemon.
See below the code of the new plug-in you need to copy. Of course feel free to modify it as much as you want.

```
////////////////////////////////////////////////////////////
//////////////////////////////
//
// Copyright (C) 2008 Next Limit Technologies. All rights
// reserved.
//
// This file is just part of the C++ SDK examples provided
// with //RealFlow(c).
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND,
// INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A
// PARTICULAR //PURPOSE.
//
////////////////////////////////////////////////////////////
//////////////////////////////

#include <iostream>
#include <sstream>
#include <string>

#include <rf_sdk/sdk/appmanager.h>
#include <rf_sdk/sdk/scene.h>
#include <rf_sdk/sdk/vector.h>
#include <rf_sdk/sdk/daemon.h>
#include <rf_sdk/sdk/pb_particle.h>
#include <rf_sdk/sdk/pb_emitter.h>
#include <rf_sdk/sdk/ppty.h>
#include <rf_sdk/sdk/plgdescriptor.h>
#include <rf_sdk/daemons/daemonplgsdk.h>
#include <rf_sdk/sdk/rfsdklibdefs.h>
#include <rf_sdk/sdk/sdkversion.h>

////////////////////////////////////////////////////////////
///////

using namespace std;
using namespace nextlimit::rf_sdk;

////////////////////////////////////////////////////////////
///////


class MyOwnGravitonDaemonSDK: public DaemonPlgSdk
{

  public:


/// Constructor.
  MyOwnGravitonDaemonSDK()
  {
// Gives an unique local Id
    localID = MyOwnGravitonDaemonSDK::globalLocalID++;

    timesBeingCalled = 0;

    angleBody    = 0.0f;
    angleEmitter = 0.0f;
  }

/// Destructor.
  virtual ~MyOwnGravitonDaemonSDK() {};

/// Class id.
  virtual NL_INT32 getClassId() const
  {
    return ( id obtained in realflow );
  };

  /// Get plugin name.
  virtual std::string getNameId() const
  {
    return ( "MyOwnGraviton" );
  };
  // getCopyRight()
  virtual std::string getCopyRight() const
```

```
  {
    return std::string( "Copyright (C) 2008 Next Limit
Technologies. All rights reserved." );
  }

  // getCopyRight()
  virtual std::string getLongDescription() const
  {
    return std::string( "Adds a constant force in a certain
direction." );
  }

  // getCopyRight()
  virtual std::string getShortDescription() const
  {
    return std::string( "Adds Constant Force" );
  }


  /// Initialize plugin, add properties, etc.
  virtual void initialize( PlgDescriptor* plgDesc )
  {
    Ppty fStrength = Ppty::createPpty( "FStrength", 2.0f );
    plgDesc->addPpty( fStrength );

    Ppty fDir = Ppty::createPpty( "FDir", Vector( 1.0, 1.0,
0.0) );
    plgDesc->addPpty( fDir );

    Ppty fPerp = Ppty::createPpty( "FPerp", 1.0f );
    plgDesc->addPpty( fPerp );

    Ppty angleIncStep = Ppty::createPpty( "AngleIncStep",
1.0f );
    plgDesc->addPpty( angleIncStep );
  }
virtual void applyForceToEmitter(  Daemon* thisPlg, PB_
Emitter* emitter,
                                   PB_Emitter::iterator
iter)
  {
    applyForceToEmitter( thisPlg, emitter, 0, iter );
  }

  //------------------------------------------------
  //  Function: applyForceToEmitter
  //  This function is called by the simulation engine
  //  when external forces should be applied to the
  //  particles in the emitter.
  //------------------------------------------------
  virtual void applyForceToEmitter( Daemon* thisPlg, PB_
Emitter* emitter,
int nThread, PB_Emitter::iterator iter
  )
  {

    float  fstrength    = thisPlg->getParameter<float>  (
"FStrength"   );
    Vector fDir         = thisPlg->getParameter<Vector> (
"FDir"        );
    float angleIncStep = thisPlg->getParameter<float> (
"AngleIncStep");
    float fPerp = thisPlg->getParameter<float> ( "FPerp");

    fDir.normalize( );
    fDir.scale    ( fstrength );
    Vector fDirNorm( fDir );
    fDirNorm.normalize();

    Vector orthoGonal = Vector::cross( fDir, Vector( 0, 1, 0
) ) * fPerp;
    angleEmitter = angleEmitter + angleIncStep;
    orthoGonal.rotateAroundAxis( fDirNorm, angleEmitter );

    // Final force applied
    Vector finalForce = fDir + orthoGonal;
```

```
    // apply the daemon transformation to gravity direction
vector
    // unit direction vector
    Vector forceDirWrld = thisPlg->toWorld( finalForce );
    // assume all particles have the same mass
    Particle curpart = emitter->getFirstParticle();
    float massParticle = curpart.getMass();

    forceDirWrld.scale( massParticle );

    while( iter.hasNext() )
    {
      curpart = iter.next();

      curpart.setExternalForce( forceDirWrld );
    }
  }

  //-------------------------------------------------
  // Function: applyForceToBody
  // This function is called by the simulation engine
  // when external forces should be applied to the body.
  //-------------------------------------------------
  virtual void applyForceToBody(
        Daemon* thisPlg,  Object* obj )
  {

    float  fstrength    = thisPlg->getParameter<float>  (
"FStrength" );
    Vector fDir         = thisPlg->getParameter<Vector> (
"FDir" );
    float angleIncStep = thisPlg->getParameter<float> (
"AngleIncStep");
    float fPerp = thisPlg->getParameter<float> ( "FPerp");

    fDir.normalize( );
    fDir.scale    ( fstrength );
    Vector fDirNorm( fDir );
    fDirNorm.normalize();
    Vector orthoGonal = Vector::cross( fDir, Vector( 0, 1, 0
) ) * fPerp;

    angleBody = angleBody + angleIncStep;
    orthoGonal.rotateAroundAxis( fDirNorm , angleBody );

    // Final force applied
    Vector finalForce = fDir + orthoGonal;

    // apply the daemon transformation to gravity direction
vector
    // unit direction vector
    Vector forceDirWrld = thisPlg->toWorld( finalForce );

    // Force proportional to mass
    float massObj = obj->getParameter<float>( "@ mass" );
    forceDirWrld.scale( massObj );

    obj->setForce( forceDirWrld );

    // Example of the use of Local Vars
    timesBeingCalled++;

    Scene& scene =  AppManager::instance()-
>getCurrentScene();
    std::stringstream msg;
    msg << "ID = " << localID << " -> applyForceToBody -
calls: " <<
        timesBeingCalled;
    scene.message( msg.str() );
  }

  //-------------------------------------------------
  // Function: removeParticles
  // This function is called by the simulation engine
  // when it is safe to remove particles.
  //-------------------------------------------------
```

```
  virtual void removeParticles(
        Daemon* plgThis, PB_Emitter* obj )
  {
  }

  int timesBeingCalled;
  int localID;

  static int globalLocalID;

  // Degrees
  float angleBody;
  float angleEmitter;

};

int MyOwnGravitonDaemonSDK::globalLocalID = 0;

////////////////////////////////////////////////////////////
///////////////

RF_SDK_DECLARE_DAEMON_PLUGIN( MyOwnGravitonDaemonSDK );

////////////////////////////////////////////////////////////
///////////////
```

8.05 Target Configuration

Have a look at the previous section to choose the right configuration.

8.06 Compiling & Linking

Go to the menu bar and choose 'Build Solution'. There should be no errors and the ouput tab should show 0 errors as shown below. The plug-in will be compiled and linked. It generates a 'my_own_graviton.dll' or a 'my_own_graviton _x64.dll' file depending on the platform you chose in the previous step.

The generated Dll can be found in the directory '$RF_XXXX_PATH\sdk\examples\ my_own_graviton\x64\Release' or '$RF_XXXX_PATH\sdk\examples\ my_own_graviton\Win32\Release' if the platform is Win32.

8.07 Deploying

RealFlow has a directory of plug-ins called '$RF_XXXX_PATH\plugins' where there is a subdirectory for each type of plug-in. Each one of these subdirectories will be scanned recursively and all the plug-ins (.dll) found for the associated type will be loaded.
In our case we just need to copy the generated my_own_graviton_x64.dll or my_own_graviton.dll to the folder '$RF_XXXX_PATH\plugins\daemons'

That´s all! Now you are ready to use your new plug-in. Just launch RealFlow. You should now see under the Daemons plugi-ns menu your graviton plug-in with the name MyOwnGraviton.

Select it to experiment with it

8.08 Testing the Plug-in

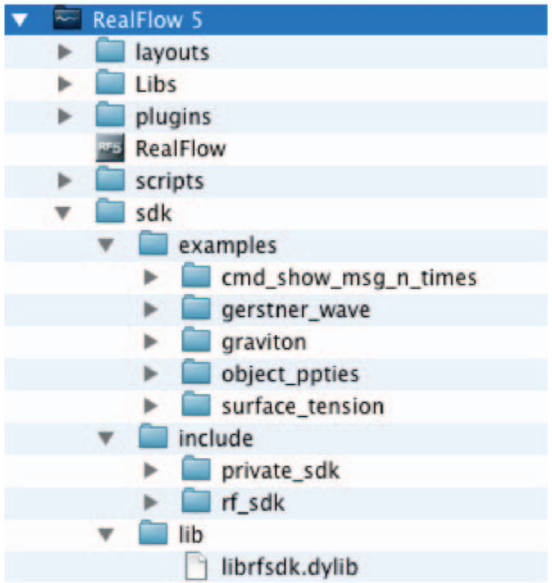
Go to the plug-ins submenu of the Daemons menu and choose the MyOwnGraviton Daemon.

9 CREATING A PLUG-IN FOR MAC OS X

So far you have learnt how to create your own plug-ins for Windows. RealFlow SDK is available for every platform it is distributed for: Windows, Mac and Linux. The steps to create a new plug-in on Mac OS X are very similar to those we have followed on Windows.

9.01 Locate your RealFlow installation

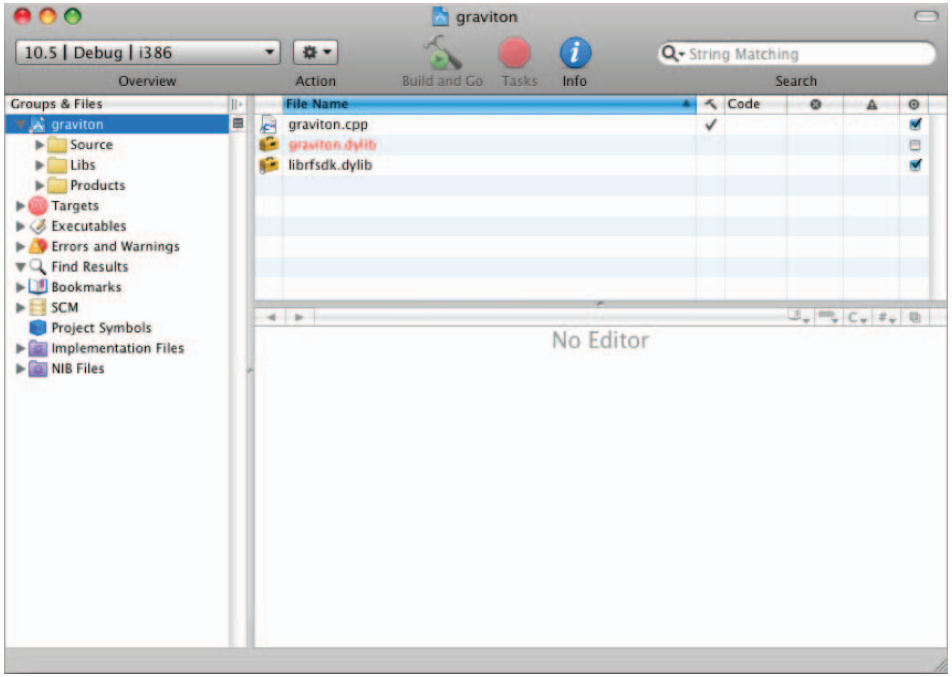
The most common place to install RealFlow is in the Applications folder. RealFlow for Mac features the following folder tree:



9.02 Duplicate the Graviton example

Just as we did on Windows, let's duplicate the 'graviton' folder and rename it something like 'graviton_text'.
If you navigate into this new folder you will find several files related to Visual Studio, and just one Xcode project: 'graviton.xcodeproj'

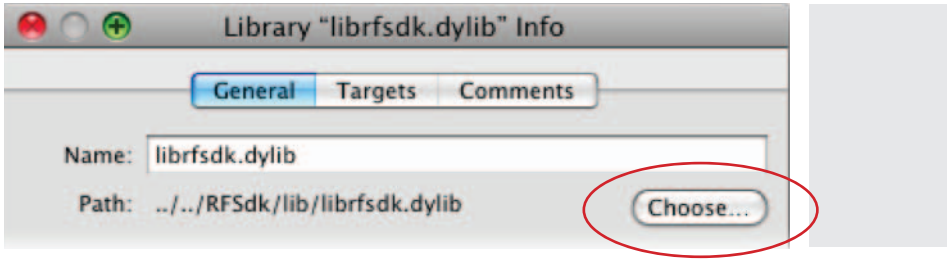
Opening that file will launch Xcode. The version we are using to illustrate the guide is 3.1.3 on Mac OS X 10.5.8. Newer versions like the one distributed with Snow Leopard may differ slightly. However, main views and configuration panels are almost identical.



If you expand the three groups Source, Libs and Products will see the source code file (graviton.cpp), RealFlow C++ SDK dynamic library (librfsdk.dylib) and the dynamic library created for the plug-in (graviton.dylib).

9.03 Check library paths

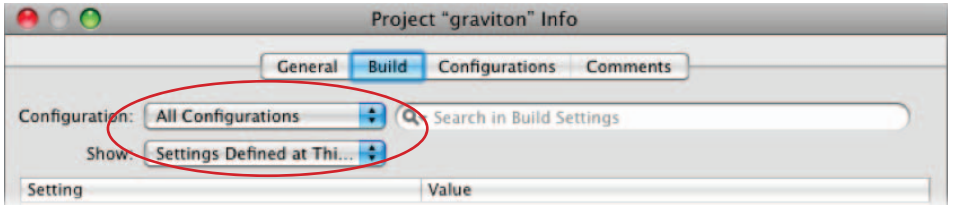
In this case, 'librfsdk.dylib' is successfully located by Xcode. Otherwise, its name would have appeared on red, just like 'graviton.dylib' which has not been compiled yet.
To check the path where 'librfsdk.dylib' is being searched, right-click on it and select 'Get Info'. You can select a different location by clicking on 'Choose...'.



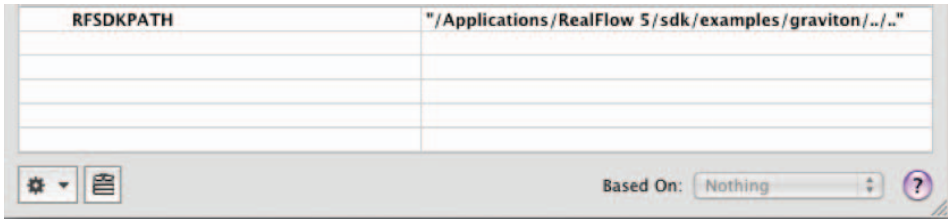
Now, let's check the header files are searched fo in the right place when compiling our plug-in.

To do so, select 'Edit Project Settings' on the Project menu. A new window will pop up showing the global preferences for this Xcode project.

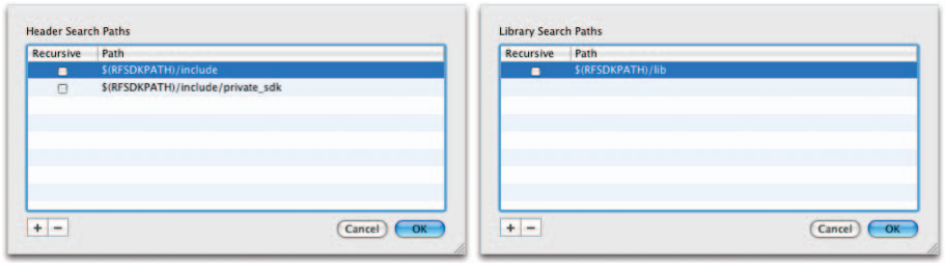
Click on the 'Build' tab and make sure 'All Configurations' is selected in 'Configuration:', Select 'Settings Defined at This Level' on 'Show:' to display only those properties that don't have a default value.



At the very bottom there is a section with all the user-defined variables. In this case you should find one called RFSDKPATH.



This variable must point to the RealFlow C++ SDK folder. In this case it is specified as a path relative to the project folder. Quotation is important to avoid problems with paths containing whitespaces.



'Header Search Paths' and 'Library Search Paths' must remain as:

- Header Search Paths
  - $(RFSDKPATH)/include
  - $(RFSDKPATH)/include/private_sdk
- Library Search Paths
  - $(RFSDKPATH)/lib

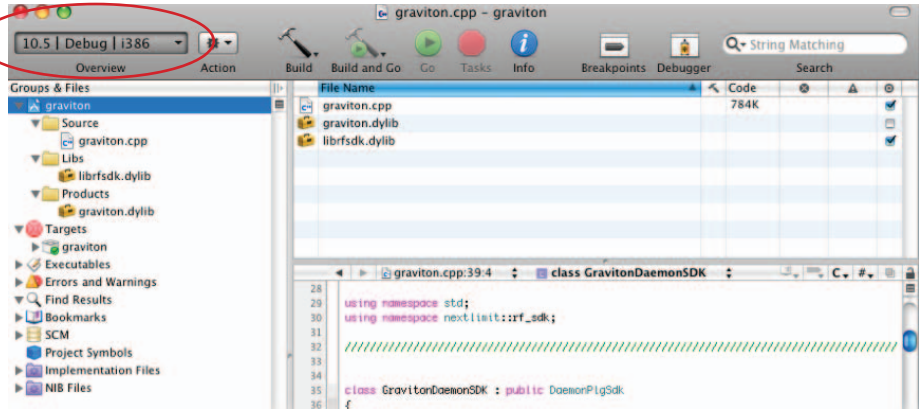
You might feel tempted to enable the "Recursive" checkbox and remove the second entry of the 'Header Search Paths'. However, leaving it this way will save some compiling time and some potential errors due to repeated path names.
This is all you need to check to make sure RealFlow plug-ins can be compiled on your Mac OS X machine.

9.04 Compiling & Linking

Now you can go back to the main window of the project and select a configuration that fits your current needs.

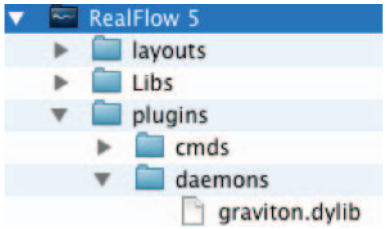
The Active Architecture value is pointless for the plug-in since RealFlow will load it on runtime and, therefore, use the same architecture it is being run on.

Regarding target architectures, RealFlow for Mac is distributed as Intel 32/64 bit and PowerPC 32 bit universal binaries. Your plug-in only needs to be compiled for the same architecture it is going to be used on. However, in case you plan to distribute it, you should probably compile it for as many architectures as possible.
Now just click on Build and wait until the plug-in is created. It should not take too long.



9.05 Deploying

The final step to make your plugin work is to copy it into the right RealFlow plug-in folder, just like for Windows. Right-click on 'graviton.dylib' and select 'Reveal in Finder'. This will open a Finder window with the plug-in file selected. Just drag it to the subfolder 'plugins/daemons' of your RealFlow installation and open RealFlow to check if it works in the same way as in the Windows instructions.



10 CREATING PLUG-INS FOR LINUX

The best option for creating your own plug-in in Linux is to use the makefiles provided with the examples. There is one example for every flavor of plug-in you can create, so you just need to find out what type of plug-in you want to create and then use the example provided.

10.01 Creating a Command Plug-in

There is one example in the folder:

$(RF_INSTALL_FOLDER)/sdk/examples/cmd_show_msg_n_times

The makefile in that folder looks as follows.

```
#============================================================================
# cmd_show_msg_n_times makefile
#
# (c) 2010 Next Limit Technologies
#
#============================================================================


CC = gcc

CFLAGS = -pipe -fPIC -03 -D_LINUX -w -c

INCLUDE = -I../../include \
        -I../../include/private_sdk

cmd_show_msg_n_times.so: cmd_show_msg_n_times.o
        $(CC) -fPIC -shared -o $@ $<

cmd_show_msg_n_times.o: ./src/cmd_msg_n_times.cpp
        $(CC) $(CFLAGS) $(INCLUDE) -c $< -o $@

install:
        cp -f cmd_show_msg_n_times.so ../../../plugins/cmds

clean:
        rm -f cmd_show_msg_n_times.o cmd_show_msg_n_times.so
~
```

To create your own command plug-in just replace the "cmd_show_msg_n_times" string with the name of your plug-in. If plug-in is made of several modules you might need to make several modifications to this makefile - please refer to the makefile documentation if you want to modify this one.

Once you have modified the makefile the procedure to compile and install is easy:

1. To make sure everything is clean just type:
   `[realflow@GRANJA3]$ make clean`

2. Now compile and link:
   `[realflow@GRANJA3]$ make`

3. Now just copy the plug-in to the appropriate folder:
   `[realflow@GRANJA3]$ make install`

Now you should be able to launch RealFlow and use the plug-in.

10.02 Creating a Daemon Plug-in

There is one example in the folder:

```
$(RF_INSTALL_FOLDER)/sdk/examples/graviton
```

The makefile in that folder looks as follows:

```
#============================================================================
# graviton makefile
#
# (c) 2010 Next Limit Technologies
#
#============================================================================


CC = gcc

CFLAGS = -pipe -fPIC -03 -D_LINUX -w -c

INCLUDE = -I../../include \
        -I../../include/private_sdk

graviton.so: graviton.o
        $(CC) -fPIC -shared -o $@ $<

graviton.o: ./src/graviton.cpp
        $(CC) $(CFLAGS) $(INCLUDE) -c $< -o $@

install:
        cp -f graviton.so ../../../plugins/daemons

clean:
        rm -f graviton.o graviton.so
```

To create your own command plug-in just replace the "graviton" string with the name of your plug-in. If your plug-in is made of several modules you might need to make several modifications to this makefile, please refer to the makefile documentation if you want to modify this one.

Once you have modified the makefile the procedure to compile and install is easy:

4. To make sure everything is clean just type:
   `[realflow@GRANJA3]$ make clean`
5. Now compile and link:
   `[realflow@GRANJA3]$ make`
6. Now just copy the plug-in to the appropriate folder:
   `[realflow@GRANJA3]$ make install`

Now you should be able to launch RealFlow and use the plug-in.

10.03 Creating a Object Plugin

There is one example in the folder:

`$(RF_INSTALL_FOLDER)/sdk/examples/object_ppties`

The makefile in that folder looks as follows.

```
#=============================================================================
# object_ppties makefile
#
# (c) 2010 Next Limit Technologies
#
#=============================================================================


CC = gcc

CFLAGS = -pipe -fPIC -03 -D_LINUX -w -c

INCLUDE = -I../../include \
        -I../../include/private_sdk

object_ppties.so: object_ppties.o
        $(CC) -fPIC -shared -o $@ $<

object_ppties.o: ./src/object_ppties.cpp
        $(CC) $(CFLAGS) $(INCLUDE) -c $< -o $@

install:
        cp -f object_ppties.so ../../../plugins/objects


clean:
        rm -f object_ppties.o object_ppties.so
```

To create your own command plug-in just replace the "object_ppties" string with the name for your plug-in. If your plug-in is made of several modules you might need to make several modifications to this makefile, please refer to the makefile documentation if you want to modify this one.

Once you have modified the makefile the procedure to compile and install is easy:

7. To make sure everything is clean just type:
   `[realflow@GRANJA3]$ make clean`
8. Now compile and link:
   `[realflow@GRANJA3]$ make`
9. Now just copy the plug-in to the appropriate folder:
   `[realflow@GRANJA3]$ make install`

Now you should be able to launch RealFlow and use the plug-in.

10.04 Creating a ParticleSolver Plug-in

There is one example in the folder:
$(RF_INSTALL_FOLDER)/sdk/examples/surface_tension
The makefile in that folder looks as follows:

```
#==============================================================================
# surface_tension makefile
#
# (c) 2010 Next Limit Technologies
#
#==============================================================================


CC = gcc

CFLAGS = -pipe -fPIC -O3 -D_LINUX  -w -c

INCLUDE = -I../../include \
        -I../../include/private_sdk

surface_tension.so: surface_tension.o
        $(CC) -fPIC -shared -o $@ $<

surface_tension.o: ./src/surface_tension.cpp
        $(CC) $(CFLAGS) $(INCLUDE) -c $< -o $@

install:
        cp -f surface_tension.so ../../../plugins/particles

clean:
        rm -f surface_tension.o surface_tension.so
```

To create your own command plug-in just replace the "surface_tension" string with the name for your plug-in. If your plug-in is made of several modules you might need to make several modifications to this makefile, please refer to the makefile documentation if you want to modify this one.
Once you have modified the makefile the procedure to compile and install is easy:

10. To make sure everything is clean just type:
    `[realflow@GRANJA3]$ make clean`
11. Now compile and link:
    `[realflow@GRANJA3]$ make`
12. Now just copy the plugin to the appropriate folder:
    `[realflow@GRANJA3]$ make install`

11 GENERALITIES

11.01 Scene Object

There is a single Object in the C++ SDK that represents the current scene in RealFlow . To get it you need to use the line:

```
Scene& scene =  AppManager::instance()->getCurrentScene();
```

Once you have a scene object, you can call the methods of the scene as defined in the API. For instance, to add a new cube object to the scene you would use:

```
Object obj = scene.addCube();
```

11.02 Properties

You can add properties of different types to all the plug-ins except the Event/Command ones.
You add these properties in the method initialize , that gets called when a plug-in is instantiated. All the instances of a Plug-in type share the same properties. Of course, each instance has its own values. And those values are saved with the scene.

You can get/set values of these properties the same way that you do with all the other attributes of a RealFlow object. You can get a good feel for how you can create and use these attributes in any of the plug-in examples provided with the RealFlow package.

Adding Property to Plugin:

```
/// Initialize plugin, add properties, etc.
virtual void initialize( PlgDescriptor* plgDesc )
{
    Ppty fStrength = Ppty::createPpty( "FStrength", 2.0f );
    plgDesc->addPpty( fStrength );

    Ppty fDir = Ppty::createPpty( "FDir",
                                  Vector( 1.0, 1.0, 0.0) );
    plgDesc->addPpty( fDir );
}
```

As you can see basically you create a Property of the type you want by giving it a name and initial value. Then you can add it to the PlgDescriptor parameter. You can add as many properties as you like.

Getting & Setting

There is no difference to the standard attributes. In the piece of code below you can see some instructions that get/set attributes. Some of these come from the standard ones and some from the user-created ones. Their use is exactly the same as you can see in the piece of code below.

```
virtual void applyForceToEmitter(
        Daemon* thisPlg, Emitter* emitter,
        int nThread, Emitter::iterator iter )
  {

    float  fStrength  = thisPlg->getParameter<float>  (
 "FStrength" );
    Vector fDir       = thisPlg->getParameter<Vector> (
"FDir"      );
    bool   visible    = thisPlg->getParameter<bool>   (
"visible"   );

    thisPlg->setParameter( "FStrength", fStrength + 0.2 );
    thisPlg->setParameter( "visible", !visible );
```

11.03 Local Variables

For each RealFlow object (Daemon, Emitter, ParticleSolver-Fluid, Wave) that you add to the scene and that comes from a plug-in there is a unique instance of the plug-in associated to it. What this means is that you can define local variables in the plug-in class that will be visible only to your particular instance. That local variable will exist as long as the associated RealFlow object stays in the scene.
Bear in mind that these local variables will not be saved with the scene and that their values will be lost once that the objects are removed from the scene. They usually get initialized in the constructor of the Plug-in class. They can be modified in any of the methods of the plug-ins. See an example of its use in the Graviton example.

11.04 Global Variables

Have a look at the Python manual to see which uses you can give to global Variables. In essence these are variables that hold their values as long as you stay in the same scene. If you load/reload a scene or choose to create a new scene then all the global variables will get deleted.
When you load or when you create a scene or when you start RealFlow you have no global variables. You can create global variables from any of the methods of the plug-ins - even from the Initialize method of the plug-ins. All these variables will be accessible through the whole life of the session. - i.e until you change scene in RealFlow for any reason.
All the methods related to global variables are part of the Scene object. So the first thing that is needed is to get a reference to the scene object and then call the method you want.

This is shown in these basic examples for all the methods to manage global variables:

- Create Global Variable

```
Scene& scene =  AppManager::instance()->getCurrentScene();
Vector& vec = scene.createGlobalVariableValue<Vector>(
"VectorGlobal" );
```

- Get Global Variable

```
Scene& scene =  AppManager::instance()->getCurrentScene();
  vec = scene.getGlobalVariableValue<Vector>( "VectorGlobal"
);
```

- Set Global Variable

```
Scene& scene =  AppManager::instance()->getCurrentScene();
  scene.setGlobalVariableValue( "VectorGlobal", Vector( 1.0,
2.0, 3.0 ) );
```

- Remove all Global Variables

```
Scene& scene =  AppManager::instance()->getCurrentScene();
  scene.removeAllGlobalVariables();
```

- Check existence of a Global Variable

```
Scene& scene =  AppManager::instance()->getCurrentScene();
  if ( scene.existsGlobalVariable( "VectorGlobal " ) )
  {
 // DO SOMETHING!!
  }
```

Observe that some of these methods require that you provide the C++ type of the variable that you are accessing.

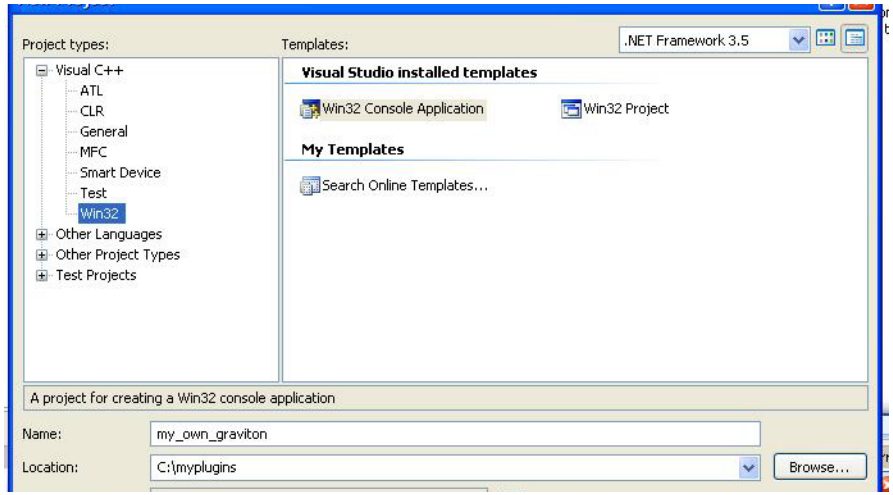
12 APPENDIXES

12.01 Setting Up Visual Studio from Scratch

Here we give detailed instructions for how to set up Visual Studio from scratch. This may be useful for advanced users that want to have more control over their development environment - and also to help understand all the settings that have been chosen to build a plug-in. This is actually very useful if an user wants to generate a plug-in for the different platforms in which RealFlow runs: Windows, Mac and Linux.
On this guide we describe how to configure Visual Studio to compile a plug-in. The steps necessary for other development environments will be quite similar and it should not be difficult to adapt them.

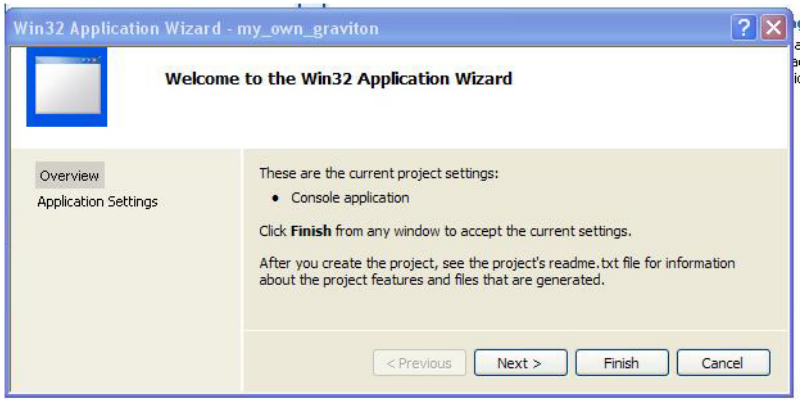
Setting Project inVisual Studio:

- Open Visual Studio, choose to create a new project. The following window will appear.



F. 01 RealFlow Installation directory • Choose 'Win32' project type and template 'Win32 Console Application' as shown above.

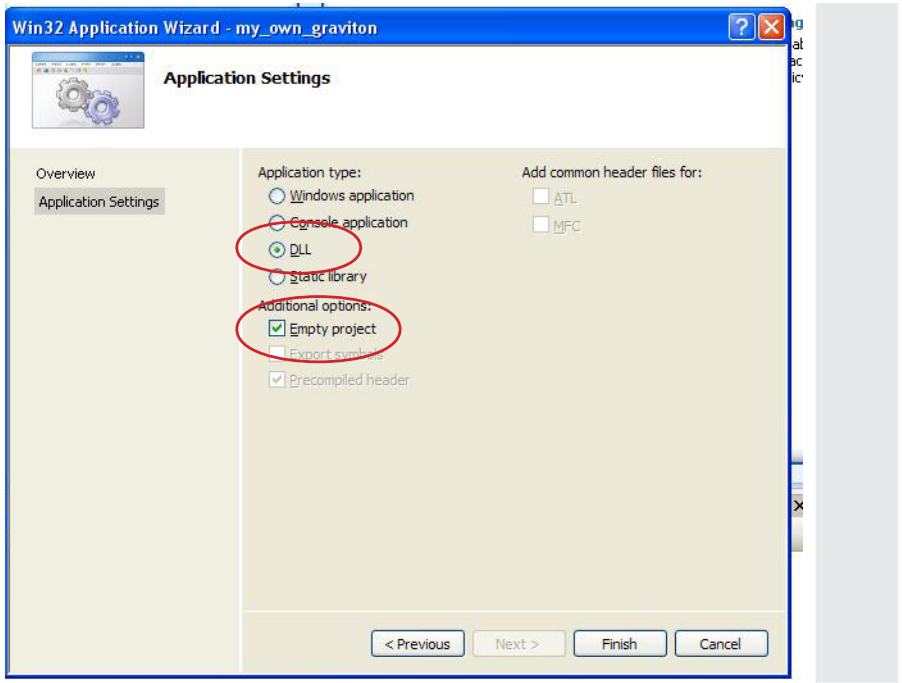
- Fill in the name field with 'my_own_graviton' or any other name.
- Choose a location for your project.
- Press ok.



RealFlow Installation directory

In **Welcome to the Win32 Application Wizard** window:

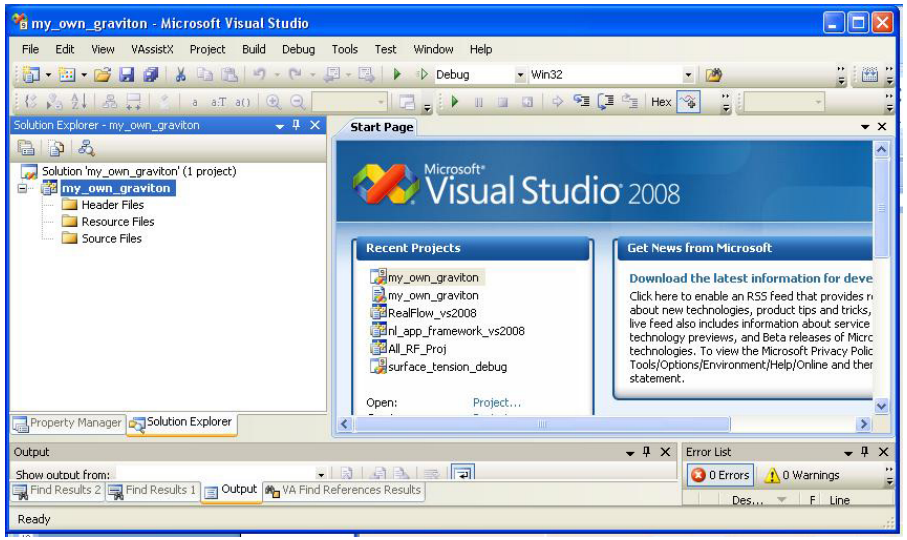
- Press Next



RealFlow Installation directory

In **Application Settings** window choose:

- Application type 'DLL'.
- Choose in Additional options 'Empty Project'.
- Press Finish

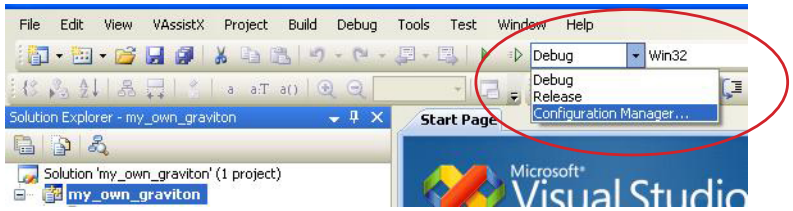


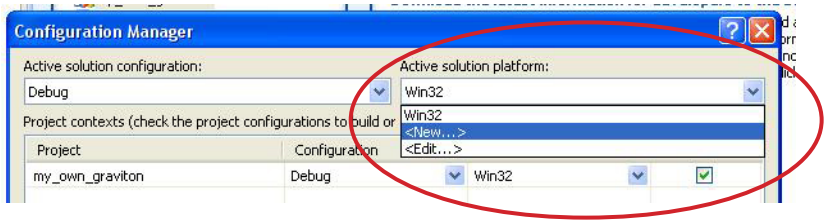
RealFlow Installation directory

Now you have an empty project, ready to be customized for developing plug-ins.
But before anything else you need to add support to compile in 64 bits. Right now it is ready only to produce code for win32.

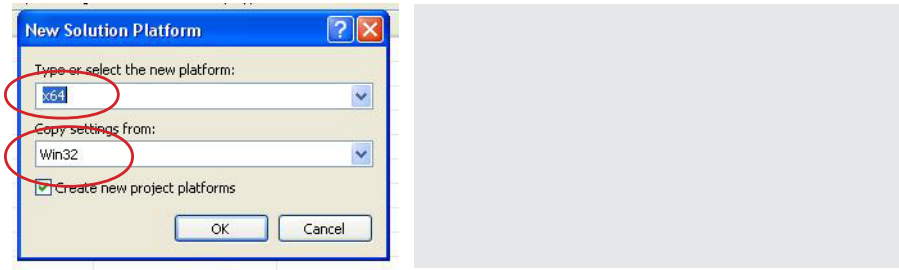
Adding x64 support

- Right-Click over your solution's name as shown below.





- Choose new



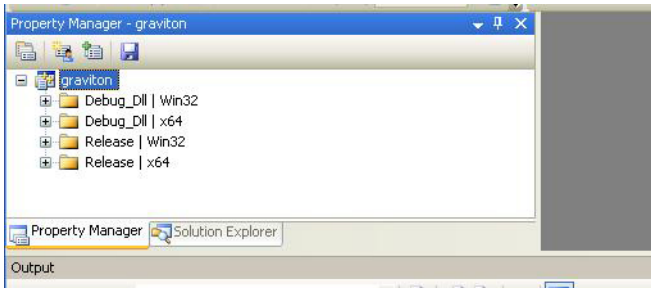
- Make sure you choose the values shown on the above figure
- Press Ok
- Press Close on the 'Configuration Manager' window

Done!. Now your project is ready to compile and build for 4 different configurations.

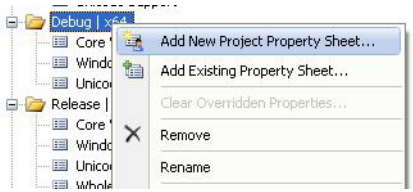
- Release for Win32
- Release for x64
- Debug for Win32
- Debug for x64

Do not worry too much about the debug/release options for the moment. Normally you just want to deploy in Release mode, but if something fails then you can use the Debug mode to trace your plug-in to find and fix bugs in your code.

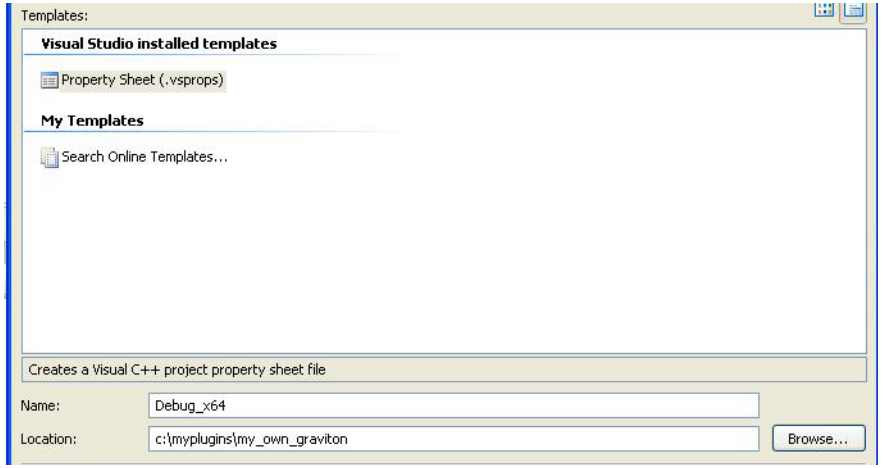
Setting Compilation and Linking Options



Choose the *'Property Manager'* tab. This should be just beside your *'Solution Explorer'* tab. If it is not visible go to the View menu and choose *'Property Manager'* submenu. Then it should appear. If it is floating you can embed it just beside the Solution Explorer tab for easy access.
Expand the 'my_own_graviton' folder to show the above picture. Each folder represents a possible configuration to build your plug-in. As we said before there are 4 possible ways to build a plug-in. The combinations of the target platform(Win32 or X64) plus the mode (Release or Debug).
Generally speaking you should use the Debug mode while you are developing your plug-in and switch to Release once you have finished developing. Bear in mind that the Debug mode is to help the developer in finding errors but it is not intended to be used at deployment time. Debug mode could create huge issues with the performance of the plug-in - also it exposes information about your plug-in that you would usually prefer not to open to everybody using it.
To be able to choose options for compilation you need to add what is called a property sheet to each configuration. This is just a convenient way to group options. To do so just right-click over each folder and choose "Add New Project Property Sheet' as shown below.



Just for convenience give each property sheet a similar name to the configuration that it belongs to. As shown below.



So you should end up with something like:

In the next section the different settings that are needed to build plugins will be shown for all the 4 possible configurations.

Compilation Settings (Release for x64)
To set settings right-click over one of the recently added property sheets and click Open. In the window that opens you can see several folders that configure many options related to compilation, linking, libraries, paths, and so on, which are needed to build an application. Don't worry about the quantity in most cases you will not need to change many of them. Most of them come with the right value already set.

Set the values as shown below:

- C++ Folder
  - General Tab
    - Additional Include Directories = "$RF_XXXX_PATH\sdk\include"
      Directory with the library headers.

    - Debug Information Format = Program Database (/Zi)
      Debug information type generated by the compiler

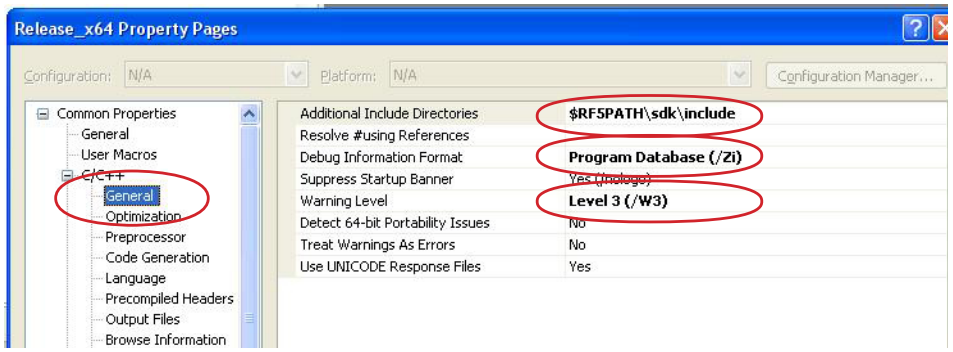
    - Warning Level = Level 3 (/W3) or Level 4 (/W4)
      Related to the detail of warnings you get while compiling. Warnings indicate that there could be some problems with the code. It will compile but it may be not correct.

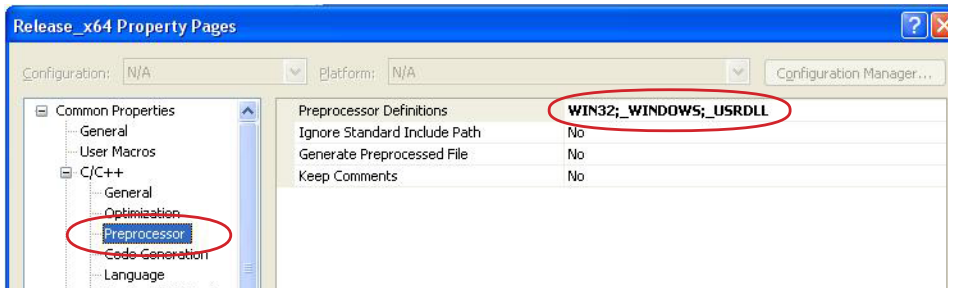
  - Optimization
    - Optimization = Maximize Speed (/O2)
    - Enable Intrinsic Functions = Yes (/Oi)

☑ *These options are related to the performance of the code generated. Please refer to the Visual Studio compiler for more information. In general you should not need to change them.*

  - Preproccesor
    - Preprocessor Definitions = NDEBUG;WIN32;_WINDOWS;_USRDLL
      Compilation variables. These need to be defined only for the Windows version. In a Linux or Mac environment they should be not defined.

  - Code Generation
    - Runtime Library = Multi-threaded DLL (/MD)





Linking Settings (Release for x64)

- Linker
  - General Tab
    - Output File = $(OutDir)\$(ProjectName)_x64.dll
      This chooses the name of the plugin dll. You can choose whatever name you prefer - it's good practice to identify the x64 version in a clear way. The same goes for the Debug mode. You will see later on that we recommend that you append a D to the dll name.

    - Enable Incremental Linking = No (/INCREMENTAL:NO)
      This just increases the speed of linking for projects with multiple files. In this case, because plugins tend to be small there is no need for this.

- Additional Library Directory = $RF_XXXX_PATH\sdk\lib
  Path to the sdk libraries needed to link a plugin. If the plugin needs more external libraries this is the place to add the paths to them.

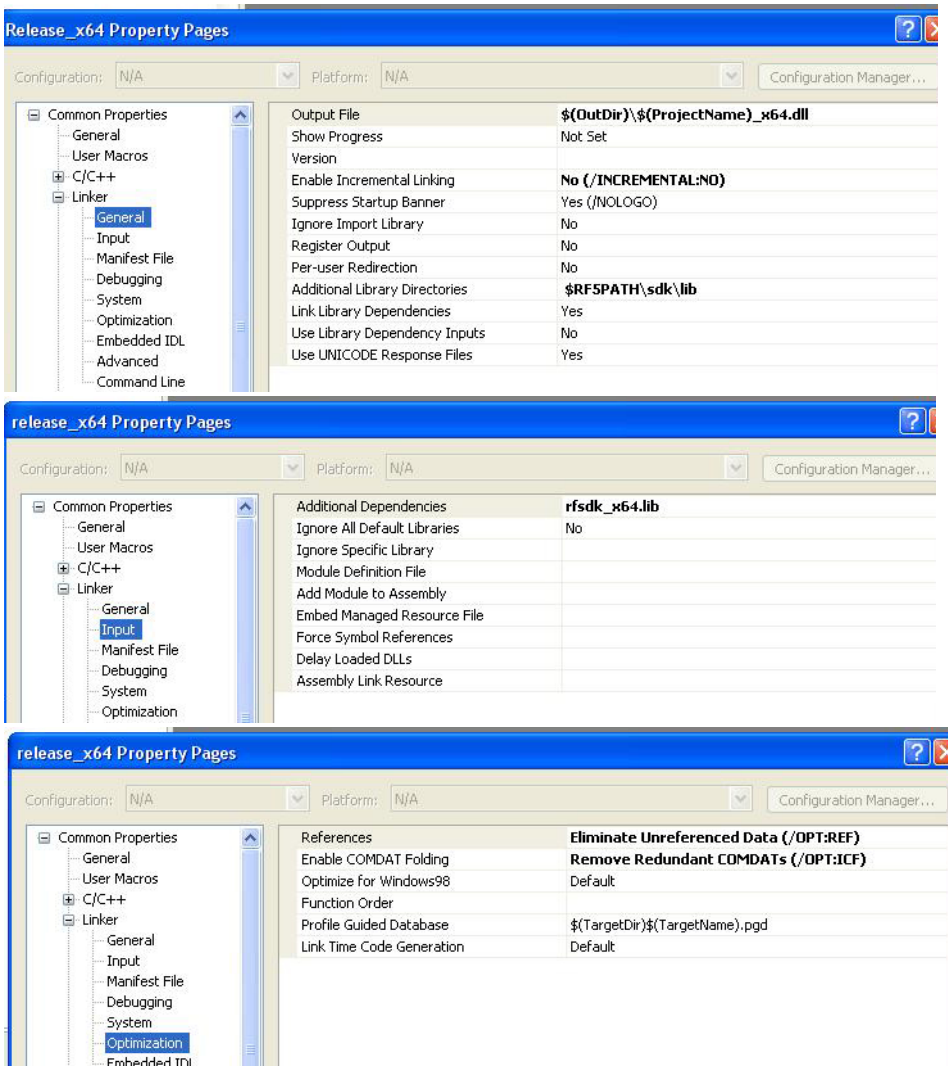
- Input
  - Additional Dependencies = rfsdk_x64.lib
    SDK Library used to link a plugin. If the plugin needs more external libraries this is the place to add the library files.

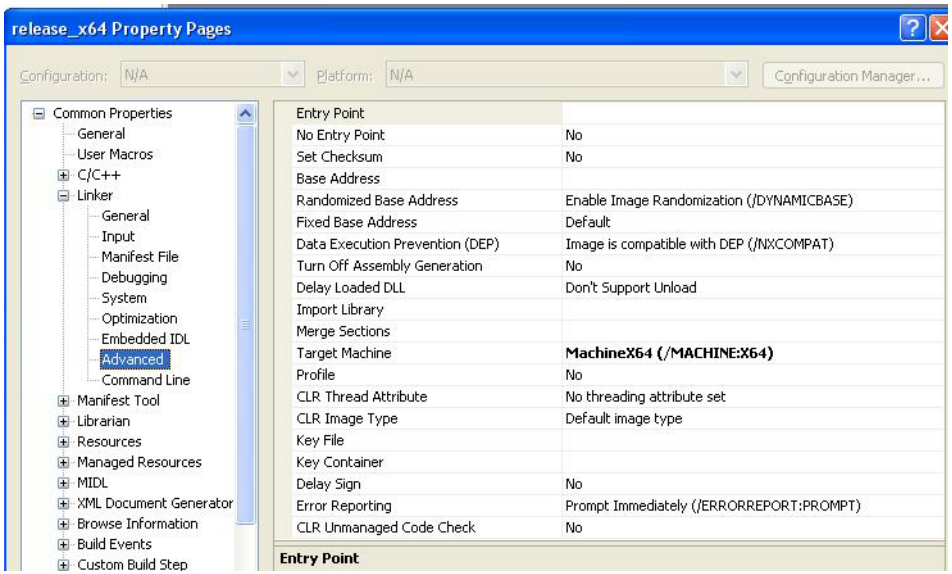
- System
  - System = Windows (/SUBSYSTEM:WINDOWS)
    Enviroment that the plugin is compiled for.

- Optimization
  - References = Eliminate Unreferenced Data (/OPT:REF)
  - Enable COMDAT Folding = Remove Redundant COMDATs (/OPT:ICF)
    These options are related to the performance of the code generated. Please refer to the Visual Studio compiler for more information. In general you should not need to change them.

- Advanced
  - Target Machine = MachineX64 (/MACHINE:X64)
    This is really important!. It tells the compiler to create the plugin for the x64 environment.

☑ *Both your RealFlow application and the plugins need to match the bits version. Either both are Win32 or both are X64. If not the plugins loading will fail.*





The rest of the configurations share the majority of the settings. So for them we will show only the differences in configuration to the one above.

Compiler &Linker Settings (Release for win32)

- C++ Folder
  - General Tab
    - Additional Include Directories = "$RF_XXXX_PATH\sdk\include"
    - Debug Information Format = Program Database (/Zi)
    - Warning Level = Level 3 (/W3) or Level 4 (/W4)
  - Optimization
    - Optimization = Maximize Speed (/O2)
    - Enable Intrinsic Functions = Yes (/Oi)
  - Preproccesor
    - Preprocessor Definitions = NDEBUG;WIN32;_WINDOWS;_USRDLL
  - Code Generation
    - Runtime Library = Multi-threaded DLL (/MD)
- Linker
  - General Tab
    - Output File = $(OutDir)\$(ProjectName).dll
    - Enable Incremental Linking = No (/INCREMENTAL:NO)
    - Additional Library Directory = $RF_XXXX_PATH\sdk\lib
  - Input
    - Additional Dependencies = rfsdk.lib
  - System
    - System = Windows (/SUBSYSTEM:WINDOWS)
  - Optimization
    - References = Eliminate Unreferenced Data (/OPT:REF)
    - Enable COMDAT Folding = Remove Redundant COMDATs (/OPT:ICF)
  - Advanced
    - Target Machine = MachineX86 (/MACHINE:X86)

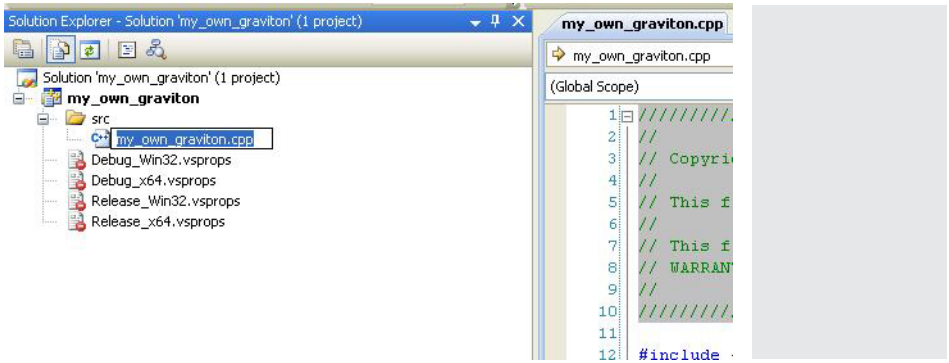
Compiler &Linker Settings (Debug for x64)

- C++ Folder
  - General Tab
    - Additional Include Directories = "$RF_XXXX_PATH\sdk\include"
    - Debug Information Format = Program Database (/Zi)
    - Warning Level = Level 3 (/W3) or Level 4 (/W4)
  - Optimization
    - Optimization = Disabled (/Od)
    - Enable Intrinsic Functions = No
  - Preproccesor
    - Preprocessor Definitions = NDEBUG;WIN32;_WINDOWS;_USRDLL
  - Code Generation
    - Runtime Library = Multi-threaded DLL (/MD)
    - Basic Runtime Checks = Both (/RTC1, equiv. to /RTCsu) Helps to catch some runtime errors
- Linker
  - General Tab
    - Output File = $(OutDir)\$(ProjectName)_DL.dll
    - Enable Incremental Linking = No (/INCREMENTAL:NO)
    - Additional Library Directory = $RF_XXXX_PATH\sdk\lib

- Input
  - Additional Dependencies = rfsdk.lib
- System
  - System = Windows (/SUBSYSTEM:WINDOWS)
- Optimization
  - References = Default
  - Enable COMDAT Folding = Default
    Because this is a Debug mode. The optimizations need to be off in order to be able to trace the whole code
- Advanced
  - Target Machine = MachineX86 (/MACHINE:X86))

Now you have a Visual Studio environment ready to start programming. For convenience we recommend that you create a folder for your source code. Lets create one and call it 'src'.

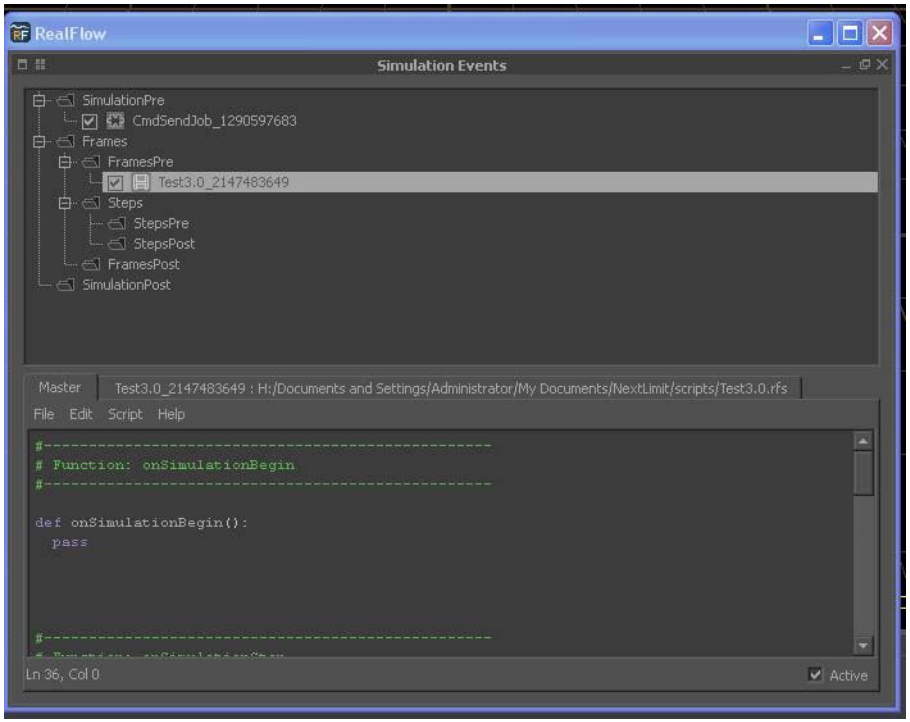
The easiest way to create a new plug-in is from a previously existing plug-in, using this previous one as a template for the new one. If for instance you are going to create a new Daemon plug-in you can use a starting point the file "graviton.cpp" from the graviton example in the examples folder ($RF_XXXX_PATH\sdk\examples\graviton). So you will have the next file structure in the my_own_graviton plug-in.



Now go inside the code and start changing the plug-in. See below the transformed code once the original code has been stripped away and only the skeleton of a Daemon plug-in has been left in place.

12.02 Simulation Events Pipeline

This will be a brief introduction to the use of the new Simulation Events Pipeline. Let´s first have a look at the new window 'Simulations Events'. Go to the menu 'Menu Bar -> Layout-> Simulation Events' or just press 'Ctrl + F2'. The following screen will appear:

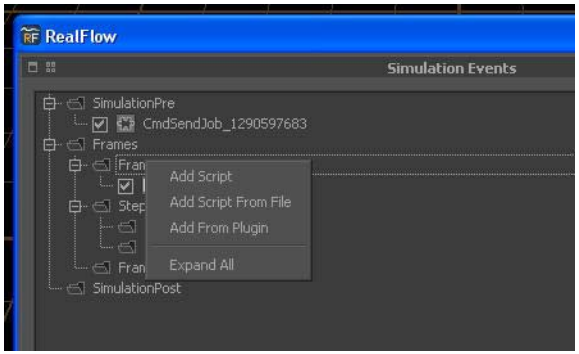


This is what we call a pipeline of events. It is divided in two windows.

In the bottom half you have what is basically a copy of the old "Events Script" window, now called "Master". It works in the same way as in RealFlow 4. You can switch it off through the active option in the bottom right corner of the window.

The top half of the window works in a similar fashion to the old "Events Script" window. There you had a script window where you could define some methods that were called at specific times in the simulation pipeline. So you had for instance a method called "onSimulationBegin" that was called at the beginning of the simulation. We have further developed further this concept into a more modular events manager. Now at the same specific times in the simulation pipeline you can choose to run any number of scripts or event plug-ins.

To add a script, select the point at which you want it to be called and pressing the right mouse button select the script type from the menu that pops up, as is shown in the picture.



In this screen you can see two scripts already added. Let´s play with the screen a bit. You can:

- Move the scripts to a new location
- Activate/deactivate them
- Remove them
- Rename them
- Edit them (Only the Python scripts)

There are three types of scripts you can add:

Embedded script (in the menu "Add Script"), this lets you write a Python script straight away.
Script from file ("Add Script From File"), this lets you choose a Python file as the script to be called.

And the last one will be the new type based on the plug-in system.

Script from plugin ("Add From Plugin") lets you choose a Event plug-in as the 'script' to be called.

It is that easy.

These two windows define events in an independent way. You can use whichever suits your needs, or you can use both at the same time. All the events coming from either window will be run at their corresponding step of the simulation.

When RealFlow starts it loads all the plug-ins - so later on you will be able to choose them from the Simulation Events screen.

Bear in mind that you can use the same Command/Event plugins loaded from the "....\plug-ins\cmds" to create either an Event Command or a Command to be run from the GUI interface.

12.03 Setting Up Xcode from Scratch

In this appendix we describe all the steps to convert an Xcode empty project into a RealFlow plug-in which is ready to compile and deploy.

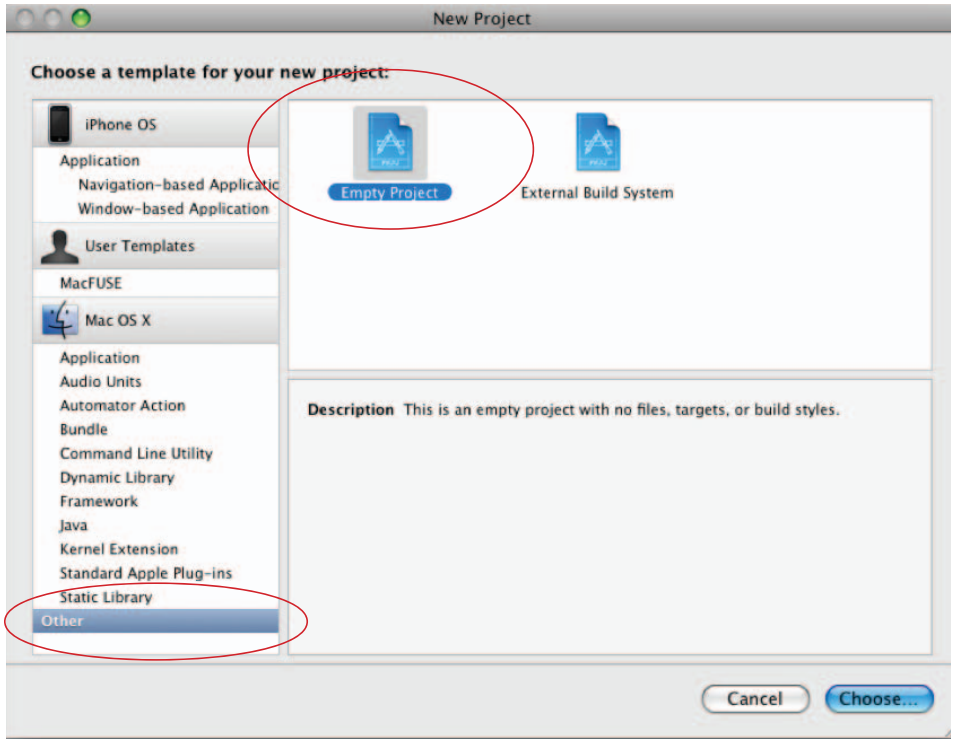
Xcode
Xcode is freely distributed as a bundle on every Mac OS X install disk.

Please refer to the Xcode Installation Guide at:

http://developer.apple.com/mac/library/DOCUMENTATION/Xcode/Conceptual/XcodeCoexistence/Contents/Resources/en.lproj/Basics/Basics.html

If you are registered as an Apple Developer you can get access to extra updated Xcode versions.

Creating a new project
The first thing you have to do is open Xcode and select 'New Project' in the File menu. Depending on your Xcode version the amount of project types may vary drastically. However, there is always the option of creating an Empty Project, from the 'Other' section. As project name, you can use the same as in the Windows section: 'my_own_graviton'. Choose any location for it.



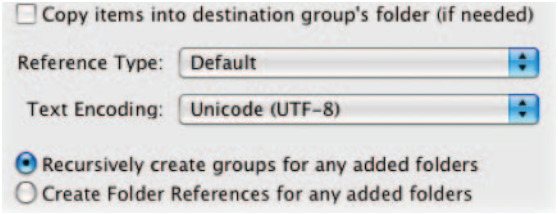
Adding RealFlow C++ SDK dynamic library
Once the project has been created you will be brought to the main project window.

Click on *'New Group'* from the Project menu, and set the group name to something like 'RFSDK Library'. Right-click on the group you just created and select 'Add > Existing Files...'.

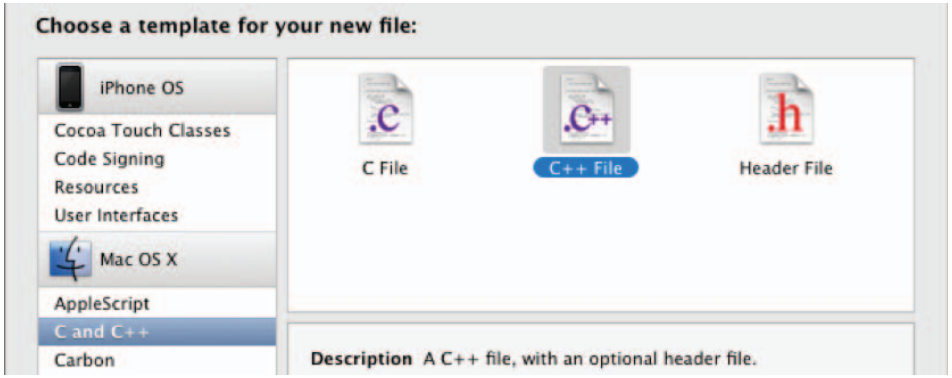
A dialog will pop up asking for the files to add. Navigate to your RealFlow installation (commonly '/Applications/RealFlow') and select the file named 'librfsdk.dylib' on 'sdk/lib' subfolder.

Right after your selection, Xcode will show another dialog asking the way the file must be imported.

Just make sure *'Copy items into destination group's folder'* is not checked and close the dialog by clicking on *'Add'*.



Adding source code
You can create a new group and call it something like 'Source'. In that group you can add existing source code files (usually .cpp and .h) or create new source code files.
If you decide to create new files, right-click on the group and select *'Add > New file...'*.
A dialog will show up asking you what type of file you want to create. Select 'C++ File' from the *'C and C++'* section. You will be prompted for the file name and location, and an optional header file.

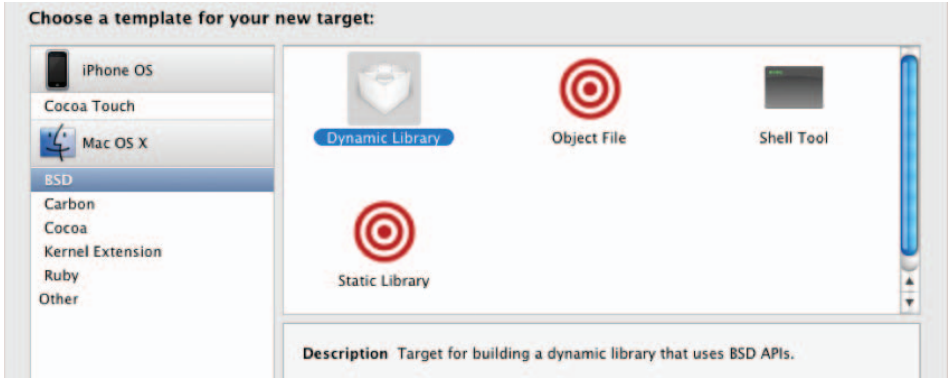


From now on, the description will continue as if you had added a 'my_own_graviton.cpp' file with the same content as in the Building your own Plug-in section.

Creating the plugin target

At the moment, all your project contains is a source code file and a reference to the RealFlow C++ SDK dynamic library.

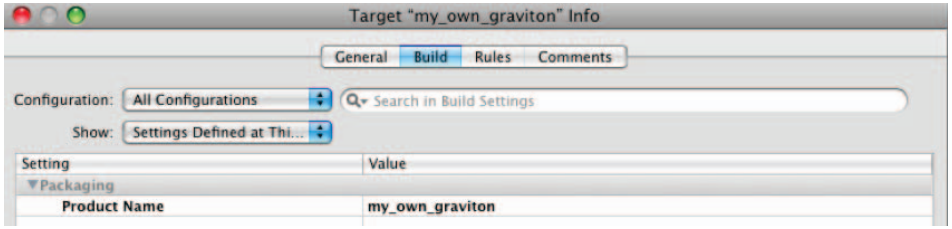
You need to create a dynamic library that will represent your plug-in so, click on 'New target…' from the Project menu.



From all the Dynamic Library choices all you need is the cleanest one, so you can set up your own flags and dependencies. Select the 'Dynamic Library' target from the 'BSD' section.
After typing a name for your target it will appear on the 'Groups & Files' section of the project window.

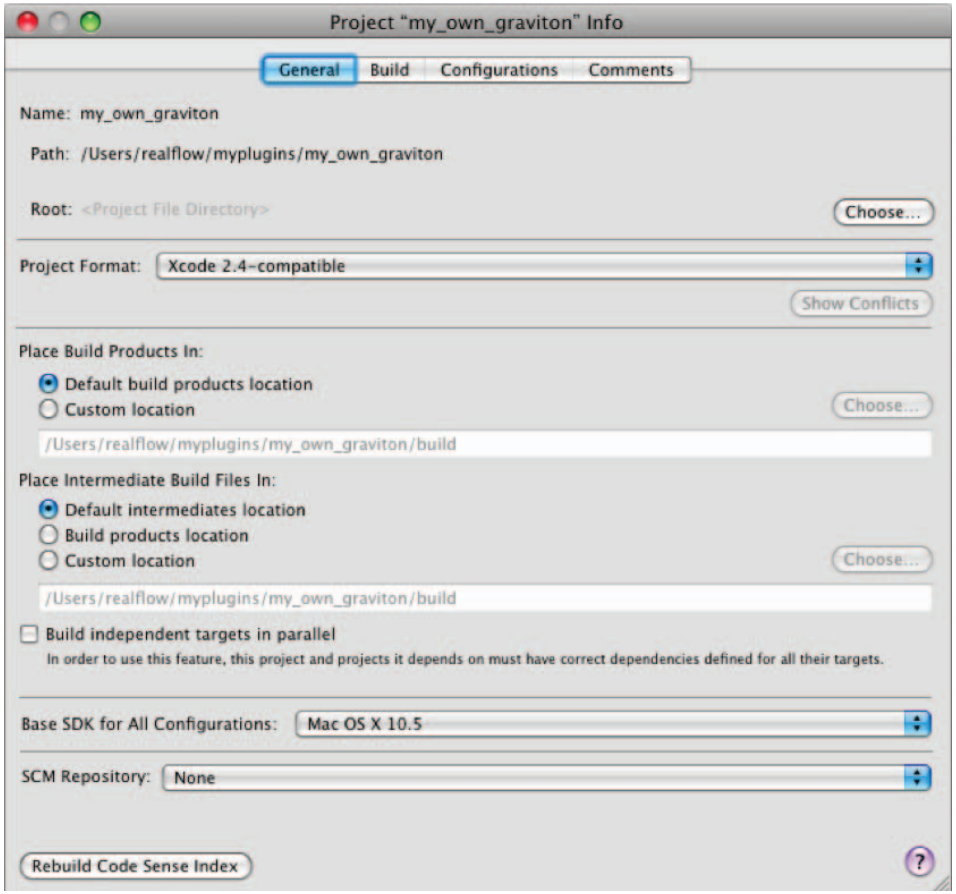
Right click on it and select *'Get Info'* to show a window with the target settings.

Activate the *'Build'* tab and set *'All Configurations'* as 'Configuration:' and *'Settings Defined at This Level'* on 'Show:'. You can remove everything except the *'Product Name'*. This way all the settings will have the default value inherited from the global settings of the project. We will check them in the next step.



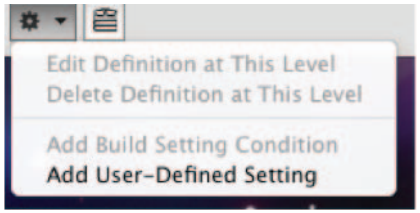
Tweaking project settings

This is the most important step. Click on 'Edit Project Settings' from the 'Project' menu.
On the 'General' tab, set 'Base SDK for All Configurations' to 'Mac OS X 10.5'. RealFlow is only compatible with Mac OS X version 10.5 or higher.
Here there is a screenshot to get a reference for all the settings from this tab:



Now switch to the 'Build' tab.
Make sure 'Configuration:' is set to *'All Configurations'*. You can leave 'Show:' with the *'All Settings'* value.

Click on the gear button located on the bottom left corner to Add a User-Defined Setting. Set its name to 'RFSDKPATH' and make it point to the 'sdk' folder of your RealFlow installation. It should be something like "/Applications/RealFlow/sdk". Double quotes must be present to avoid problems with paths containing whitespaces.

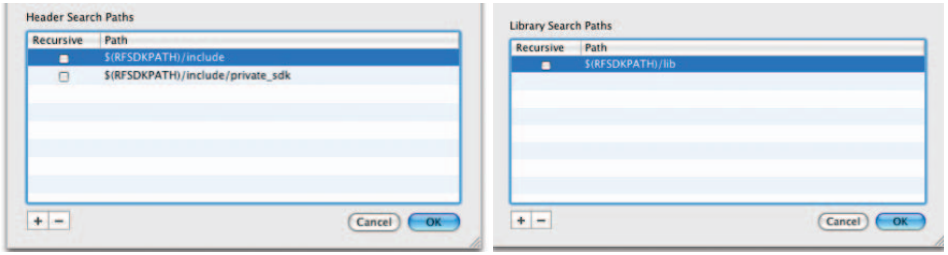


The next step is to set the header and library search paths to point to the RealFlow C++ SDK location.

Scroll through the settings to the 'Search Paths' section.

Set the following values:

- Header Search Paths
  - $(RFSDKPATH)/include
  - $(RFSDKPATH)/include/private_sdk
- Library Search Paths
  - $(RFSDKPATH)/lib



'Header Search Paths' represent the locations where header files (.h) included in your source code will be searched on at compiling time.
'Library Search Paths' represent the locations where third party libraries (.dylib or .a) will be searched on at linking time.

You are just a few more settings away from being able to create your plug-in.

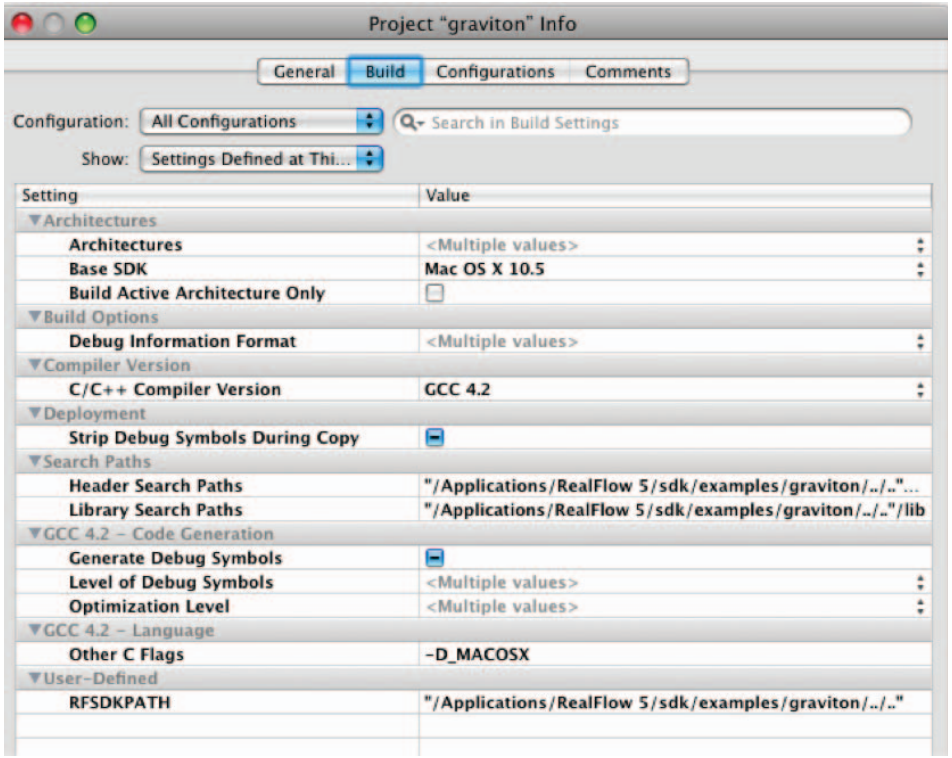
Locate the *'C/C++ Compiler Version'* under *'Compiler Version'* section. This setting must be set to 'GCC 4.2'.

At the very top, there is an 'Architectures' section. *'Architectures'* can be set at most to: i386 x86_64 ppc.

Any other architecture will fail to compile because RealFlow C++ SDK is only compiled for those.

'Base SDK' must be set to 'Mac OS X 10.5'.

Finally, under *'GCC 4.2 – Language'* section there is an *'Other C Flags'* setting. Set it to *'–D_MACOSX'*.



That's all. Setting 'Show:' to *'Settings Defined at This Level'* will reveal all the changes you have made plus some differences in optimization settings between Debug and Release configurations.
Close the dialog and go back to the main project window.

Building and deploying the plugin
The remaining steps are exactly the same as in *Creating a plugin for Mac OS X*. Just click on 'Build' and copy the resulting 'my_own_graviton.dylib' to the 'plugins/daemons' folder in your RealFlow installation folder.