Algoryx®

AGX DYNAMICS

AGX Dynamics User Manual

Version 2.19.0.0

Algoryx Simulation AB

2017-06-28

Table of Contents

1 Legal notice

2 AGX Dynamics

AGX Dynamics is a modular physics simulation toolkit written in C++. It contains the necessary core functionality to perform high fidelity, stable and robust simulations of mechanical systems in real-time, and can also be configured to deliver faithful simulation for applications in science and engineering. AGX Dynamics contains a mix of solvers for the resulting linear and LCP problems, suitable for different purposes: iterative for fast approximate solutions, and direct solvers for machine precision accuracy.

AGX Dynamic's API also includes event handlers for injecting user code as listeners of contact events, step events etc.

3 Scope of document

The purpose of this document is to describe the current state of the AGX Dynamics API as well give an overview of how to use the API.

4 Overall structure

4.1 Platforms

AGX is built and tested under the following platforms and compilers:

- Windows Windows 7, Windows 8, Windows 10; both 32 and 64 bit. Compilers: Visual studio 2013, 2015
- Linux, Ubuntu, gcc >= 4.8.2
- Mac OS X clang >= 7.0.2

4.2 Libraries/namespaces

AGX consists of more than 500 classes which are separated into a number of namespaces and libraries listed in *Table 1*.

Each namespace is accessible through the include directive:

```
#include <namespace/headerfile.h>
```

NAMESPACE	*LIBRARY NAME*	*DESCRIPTION*
agxSabre	agxSabre.lib	Sparse matrix solver.
agx, agxData	agxCore.lib	Core dynamics library, including basic data types: Vec3, Quat, Matrices ref_ptr, etc.
agxCollide	agxPhysics.lib	Geometric intersection system for calculating intersection points, normals, penetration depth. Classes for various primitives such as Sphere, Cylinder, Box, …
agxIO	agxPhysics.lib	Contain classes for reading / (writing) various file formats (images, models).
agxSDK	agxPhysics.lib	Simulation framework. Higher level abstraction for modeling a system, stepping collision and dynamics forward in discrete time steps. Event handling, materials.
agxStream	agxPhysics.lib	Serialization classes.
agxUtil	agxPhysics.lib	Various utilities for reading CPU information etc.
agxNet	agxPhysics.lib	Classes for communication, Sockets, Compression.
agxOSG	agxOSG.lib	Various classes for integrating AGX to OpenSceneGraph. For testing/debug purposes.
agxModel	agxModel.lib	High level modeling primitives, Tree, Terrain etc.
agxWire	agxPhysics.lib	Implementation of lumped element wire.

Table 1: Available namespaces and libraries.

An application using physics simulation typically requires the following functionality:

Namespace	Functionality	Class name
agxCollide	Modeling API for geometry/shapes	Geometry, Shape, Sphere, Box, …
agxCollide	Intersection testing and extraction of contact data	Space, SweepAndPruneBroadPhase
agx	Modeling API for physical bodies	RigidBody, MassProperties

agx	**Modeling API for constraints**	**Hinge, BallJoint, LockJoint, …**
agx	**Solving non-linear systems for constraints/forces. Integration of velocities and positions.**	**ZorroSolver**
agxSDK	**Integration of user code for event handlers**	**StepEventLister, ContactEventListener, …**
agxIO	**Reading Wave front files, Collada, images…**	**MeshReaderOBJ, ImageReaderPNG, …**
agxSensor	**Reading gamepads etc.**	**Joystick, JoystickManager..**

Table 2: Typical use vs. namespace and class names

When building/running an application with dynamics, it is important that collision space (agxCollide::Space) and the dynamics are updated and stepped correctly. Therefore there is a class that integrates these two into a higher level functionality: **agxSDK::Simulation.** This class makes sure that both collision space and the dynamics system are updated when objects are added or removed. Even though it is possible to use *agxCollide::Space individually*, we recommend always using an agxSDK::Simulation instance when building simulations.

4.3 Debug vs. Release build mode

In the installer for Microsoft Windows, the debug libraries are available. Library/runtime files in debug build has the letter 'd' added to the library name:

Build mode	Sample names	Flags
Release	agxCore.lib, agxPhysics.dll, agxModel.lib	C++: -D_NDEBUG Linker: /MD
Debug	agxCored.lib, agxPhysicsd.dll,agxModel.lib	C++: -D_DEBUG Linker: /MDd

When building applications with Microsoft Visual Studio it is very important to never mix libraries built in debug and release mode. The reason for this is how memory allocation/STL containers work in Visual Studio. Mixing libraries breaks the ODR (One Definition Rule) (see more on the web).

4.4 Solver/Dynamics

The solver in AGX is a hybrid solver which is very flexible in terms of specifying solve type for various constraints. The solver contains both an *iterative* solver and a *direct* solver.

4.4.1 Iterative solver

An iterative solver gradually improves the solution with the number of iterations, and after a finite number of iterations it thus delivers a solution with finite precision. The most common type of iterative solver is the Gauss-Seidel relaxational method and an extension called Projected Gauss-Seidel (PGS) which is used to solve constrained linear systems of equations. It solves the equations one-by-one and updates all the dependent variables of the system before proceeding with the next equation. One iteration corresponds to a full pass through all the equations. Since the constraint equations are treated one-by-one to

solve for the constraint force and corresponding updated velocity, and the information about this local solution propagates to the coupled equations, it is often referred to as an *iterative impulse propagation method*. However, the term propagation is somewhat misleading since the ordering of the equations is not well defined in PGS, so information propagates randomly, unless specifically ordered. Relaxational solvers are moderately efficient and conceptually simple, at least when the local constraint violations are small, which is often true for systems with small mass ratios and week couplings. Therefore they can be fast and efficient for large scale stacking problems with homogeneous types of bodies, while they are less efficient for e.g. lines and wires with large mass and force ratios, where they result in rubber band behavior unless the number of iterations is set to an extremely high value. They are also not so good at treating loop closures in physical systems, unless these problems are given special treatment.

4.4.2 Direct solver

A direct solver finds the exact solution (down to machine precision) in a limited number of operations (provided there exist a well-defined solution). For sparsely connected systems, a sparse type of solver is much preferred over a general dense solver, and can give enormous performance benefits. The chances of finding a proper physical solution are improved through regularization, which introduces an extra inverse mass/inertia term in all equations.

Our direct solver computes solutions of quadratic programming (QP) problems using a block pivot method. This is similar to Newton-Raphson iterations without damping or line search and is generally very fast and almost always returns a good solution in just a few steps. The linear algebra operations rely on our own sparse symmetric indefinite factorization code which is tailored for multi-body dynamics, and can also handle ill-conditioned systems of equations arising from joint singularities, also known as gimbal lock. The overall algorithm is generally faster than publicly available QP solvers based on active set or interior point methods.

The direct solver is very robust and can handle large multi-body systems with very high mass ratios, and yet maintain constraints accurately, to second order with respect to the time step. This is slower than the iterative solver but the latter makes larger errors and cannot process systems that have large mass ratios, or many joints. The simulation of a heavy vehicle for instance requires the direct solver for stability.

4.4.3 Split solver

The default solver in AGX is a *hybrid* solver which solves all constraints except for frictional contacts with the direct solver. All constraints are first solved direct, and then in a last pass, the normal and frictional contacts are solved. This results in a very fast solution, even for moderately large contact scenes with friction. For even larger systems with stacking etc., it is more efficient to select a pure iterative solution (described in 13.15).

Selected constraints can also be solved in the iterative mode if told so (see chapter 15.14). The solver utilizes an in-house implementation of a sparse matrix solver: *SABRE*. SABRE is an optimized solver for systems which are results of rigid body simulations. It makes heavily use of BLAS-level3 which results in a fully utilized CPU pipeline.

The split solver is slower than the purely iterative one, but it produces much better results with residual errors that are at least two orders of magnitude smaller. This generally produces more stable stacking with much better dry friction in general. Because of a fundamental mathematical limitation however, the direct solver becomes slower and

slower for piling or volume filling configurations. The slowdown - the rate of change of the execution time divided by the number of objects depends on the topology of the problem. It is worst for volume packing, moderate for planar constructions, i.e. walls, and nil for linear ones, i.e., stacks.

The execution time for the purely iterative solver grows only linearly with the number of bodies if the number of iteration is kept constant.

4.5 Multi-core support

Modern processors usually have two or more execution units, cores, which can be used to parallelize systems.

AGX is built upon a task model, which allows for a whole tree of tasks, each depending on other tasks. This task-tree can be split and run in parallel based on each tasks predecessors. For example, update bounding volumes can operate in parallel on each geometry, which makes it parallelizable. On the other hand, the broad phase tasks, require the bounding volume to be updated, so it need to wait until update bounding volume is done. The directory Components in the binary directory of AGX consists of many task files (both binary and XML) which together build the complete structure of the AGX Dynamics engine. More on parallelization in section 22.

4.6 Time integration

The time stepper used in AGX is derived from a discrete variational principle applied to the discrete Lagrangian representing the constrained dynamic system. This results in a stepping equation that also is valid for non-smooth and dry frictional forces, which are represented through Rayleigh dissipation functions. In the presence of smooth forces only this stepping equation corresponds to the classical Leapfrog integration scheme. For constraint impacts, a second order term is used to correct the velocities and the corresponding impulses. The rotational degrees of freedom require special treatment since the integration of the angular momentum and the quaternions contain a non-linear coupling in the gyroscopic term. In many cases the gyroscopic term is neglected (always in game physics engines, and usually the approximation is not even known, and thought to be “exact”). Neglecting gyroscopic forces corresponds to conservation of angular velocity rather than angular momentum, which in turn can be said to correspond to an approximation where the inertia tensor is homogeneous, which is true for symmetric bodies (i.e. a sphere with homogeneous mass distribution). AGX also has advanced methods for dealing with integration of the gyroscopic forces that can be used if required.

The integration of rigid body motion in AGX is based on a Cartesian representation of body coordinates in world frame x and their velocities $v = \dot{x}$. Their orientations are represented with quaternions q which represent the transformation from the body frame to world coordinates, as well as the angular velocity ω in world frame. If we label f as the sum of external and constraint forces, and τ as the sum of external and constraint torques, the time integration is

$$\begin{aligned} v_{k+1} &= v_k + \frac{h}{m} f_k \\ \omega_{k+1} &= \omega_k + h\mathcal{J}_k^{-1}\tau_k \\ x_{k+1} &= x_k + hv_{k+1} \\ q_{k+1} &= q\, exp(\frac{h}{2}\omega_{k+1}) \end{aligned} \qquad (1)$$

where the quantities are defined according to
m the scalar mass
$\mathcal{J}$ the inertia tensor in world frame
h the timestep
k the discrete time
in addition, the exponential of the angular velocity is defined as the quaternion

$$exp(\frac{h}{2}\omega) = \begin{bmatrix} \cos(\frac{h}{2})\|\omega\| \\ \frac{1}{\|\omega\|}\sin\left(\frac{h}{2}\right)\omega \end{bmatrix} \quad (2)$$

where quaternions are defined so that the first component is the real part, and the other three are the imaginary part. For a quaternion q, we denote the real and imaginary parts as q_0 and q_v respectively. We also use the right handed quaternion product so that for two quaternions p, q we have

$$pq = \begin{bmatrix} p_0 \\ p_v \end{bmatrix}\begin{bmatrix} q_0 \\ q_v \end{bmatrix} = \begin{bmatrix} p_0 q_0 - p_v \cdot q_v \\ p_0 q_v + q_0 p_v + p_v \times q_v \end{bmatrix} \quad (3)$$

and $(\cdot)$ and $(\times)$ are the ordinary scalar and vector products, respectively. The latter is also right handed so that for two vectors $x, y \in \mathbb{R}^3$ then

$$x \times y = \begin{bmatrix} -x_3 y_2 + x_2 y_3 \\ x_3 y_1 - x_1 y_3 \\ -x_2 y_1 + x_1 y_2 \end{bmatrix} \quad (4)$$

The time integration described in Eqn. (1) corresponds to the velocity version of the Störmer-Verlet stepper [1]. The equations used to solve for the constraint forces are described elsewhere[see 2, Eqn.4.31 p. 100], and this is a symmetrized variant of the Shake algorithm [1] which includes robust constraint stabilization as well as constraint relaxation to prevent illconditioning in the linear algebra. Contact constraints are also described in [2], and the contact forces are computed by solving a linear complementarity problem.

4.7 Contact mechanics (Elastic Foundation Model)

Contact mechanics in AGX Dynamics is governed by a dynamic set of local contact constraints. These constraints exist where contacts are detected and are constantly updated as a simulation progresses. Mechanically they represent a linear-elastic material constitution which is specified individually for every ContactMaterial. Furthermore, the constraints dynamically assess and utilize the local contact surface, penetration and material volume in order to collectively achieve highly stable and powerful contact mechanics.

Note however that this contact mechanics model is not intended nor generally suitable for systems that are heavily influenced by detailed or global structural deformations. Such demanding contact interactions may require complementing high-resolution nonlinear FEA.

The approach used in AGX Dynamics will lead to slight interpenetration between the bodies, which models local linear deformation. The amount of interpenetration will depend on material parameters such as Young's Modulus, as well as external forces influencing the contact. For parameters influencing the contact mechanics, (see 13.14 Materials).

There are two different approaches for calculating the resulting stiffness between two interacting geometries/shapes: contact based and area based. Which version that is being used, can be controlled per ContactMaterial. See: 13.14.4 ContactMaterial.

4.7.1 Area based approach

Collision detection will provide points, which are a discretization of the contact patch. For each point, a separating normal, as well as a contact depth (in case of overlap) and a contact area (a part of the complete contact patch) will be computed. Each contact point can be considered as a spring. Assuming the ContactMaterial's Young's modulus E and a material rest length h, the spring's contact stiffess per unit area k can be defied as:

$$k = E/h \tag{1}$$

Note that the rest length h measures the total length of the bodies in contact direction, and it varies with the position of the contact point and the direction of the contact normal. The total length of the involved shapes belonging to each of the two bodies is measured along the point and normal, with an upper cut-off representing the maximum length of the locally elastic domain, as well as a lower cut-off for numerical reasons.

Given a depth d computed from collision detection, the local pressure P can then be computed as:

$$P = k \cdot d \tag{2}$$

Given the contact area A for the contact point, the force F for the contact point is computed as:

$$F = P \cdot A \tag{3}$$

This contact model is a simplified version of the Elastic Foundation Model, which is defined in the publications by Johnson [3] and Perez [4]. Note that the version presented here differs from the ones presented in the literature in the following ways:

- The effect of Poisson's ratio is ignored, thus corresponding to a Poisson's ratio of 0 (a Poisson's ratio of 0.3 would increase k with a factor of 1.35).
- The Young's Modulus is defined for each pair of materials, and the rest lengths of the two bodies are added, thus giving a common spring stiffness per unit area. This is a simplification of the original model, in which the stiffness per unit area is computed for each body given its Young's modulus, rest length (andPoisson's ratio), and a common spring stiffness per unit area is computed by $k = \frac{k_1 \cdot k_2}{k_1 + k_2}$.

4.7.2 Contact point based approach (default)

The default contact model is an approximation of the area-based model above, where h is set to 1m and the contact area is approximated.

This model has the advantage that it is computationally faster and gives good-enough stability and fidelity in scenes with objects of sizes 1dm and larger as well as heavy masses, but it is less exact, especially when dealing with smaller and lighter objects.

5 Documentation

In addition to this manual, there is also auto generated API documentation in html format available. It is installed at <agx-dir>/doc/html

In the Windows installer there are also links installed from the Start menu which give you a short-cut to all the documentation and all the tutorials and demonstration scripts.

Located in START\AgX-2.12.0.0\

- AGX Dynamics API Documentation – Link to auto generated API documentation
- AGX Dynamics Main Page – Link to a web page containing all the tutorials and demonstration scripts

6 Installation

This chapter describes the process of installing a binary version of AGX Dynamics.

The name of the installation executable informs you of the build environment used when building the libraries:

agx-setup-<*version*>-<*platform*>-<*compiler*>-<*precision*>.exe

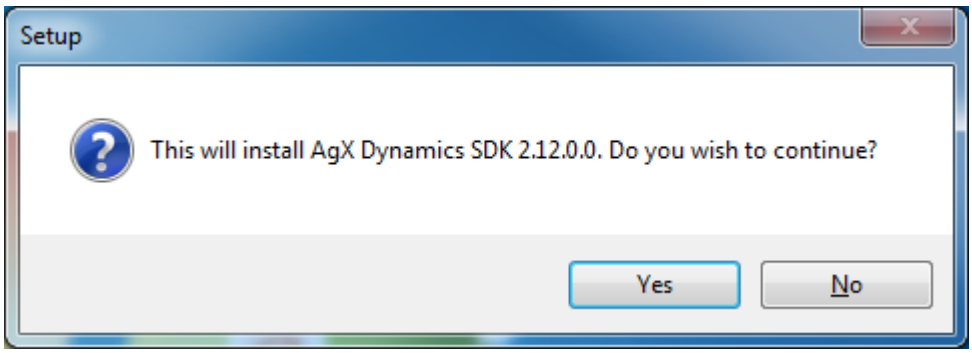
For example:

agx-setup-2.12.0.0-x64-VS2013-double.exe

AGX version 2.12.0.0
Platform: 64bit
Compiler: Visual Studio 2013
Precision: agx::Real is defined as double

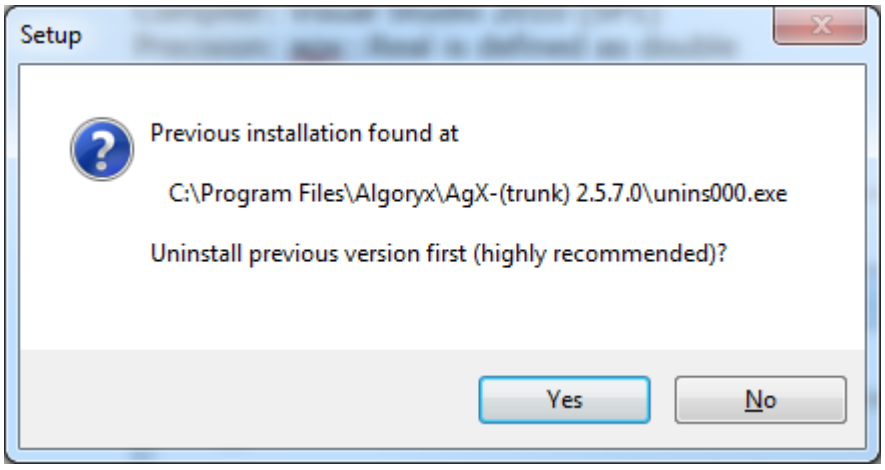
6.1 Install AGX in Windows

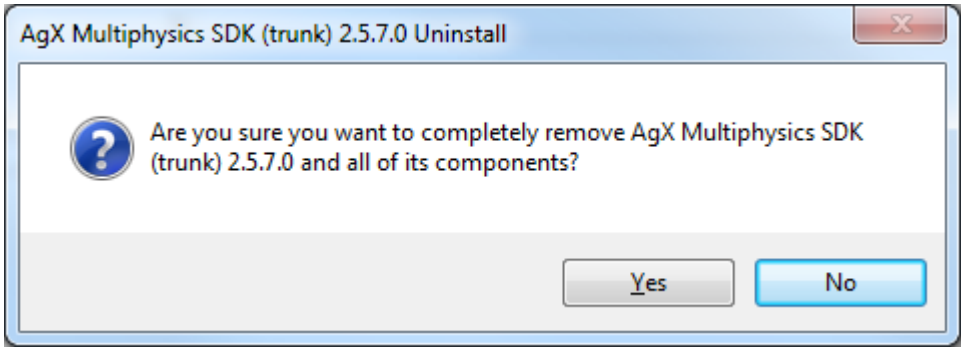
Start the installation executable.



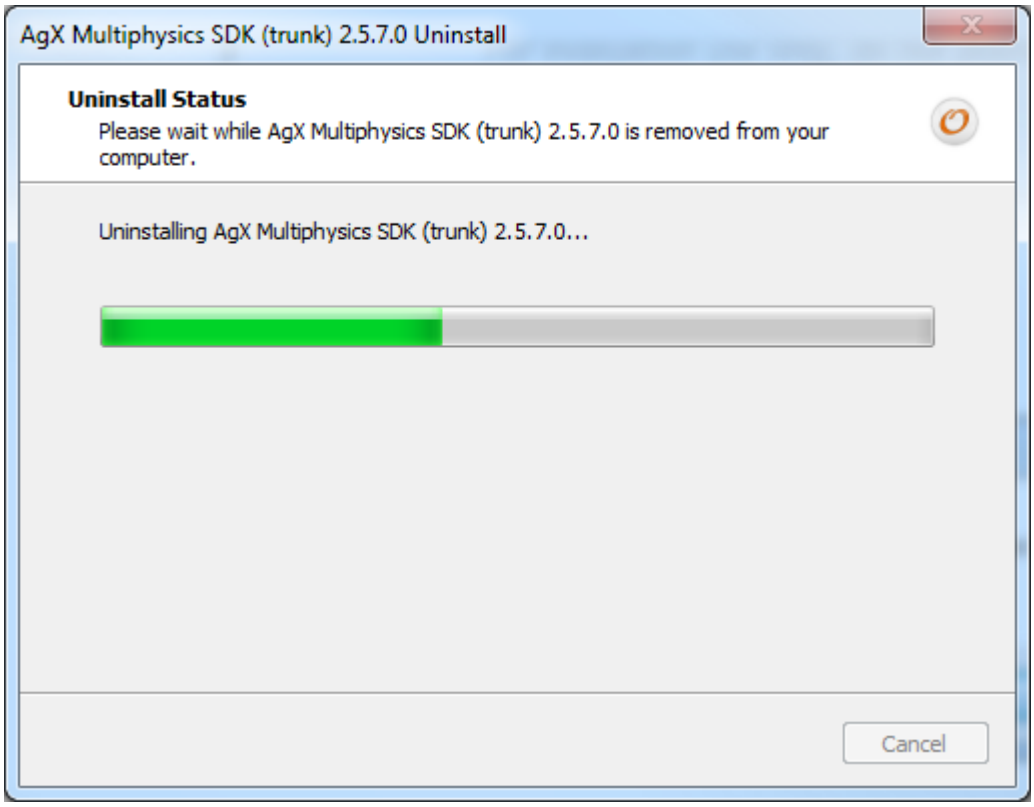
After the splash screen you will get a verification question (if you have UAC enabled) which will show you a verified installer from Algoryx Simulation.

If you have a prior installation of AGX you will get a query regarding uninstalling that version. It is recommended that you do so.

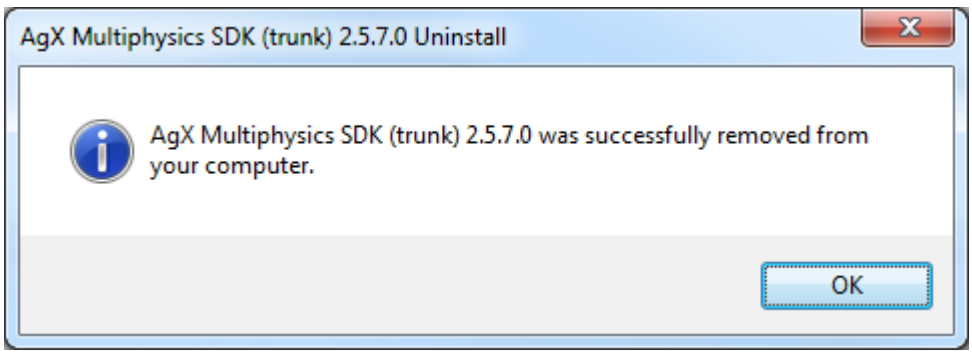




Next is the uninstall window, just press Yes to remove the previous installation. If you have created additional files, such as CMake, visual studio project files etc, they will *not* be removed. You will have to manually remove these files.

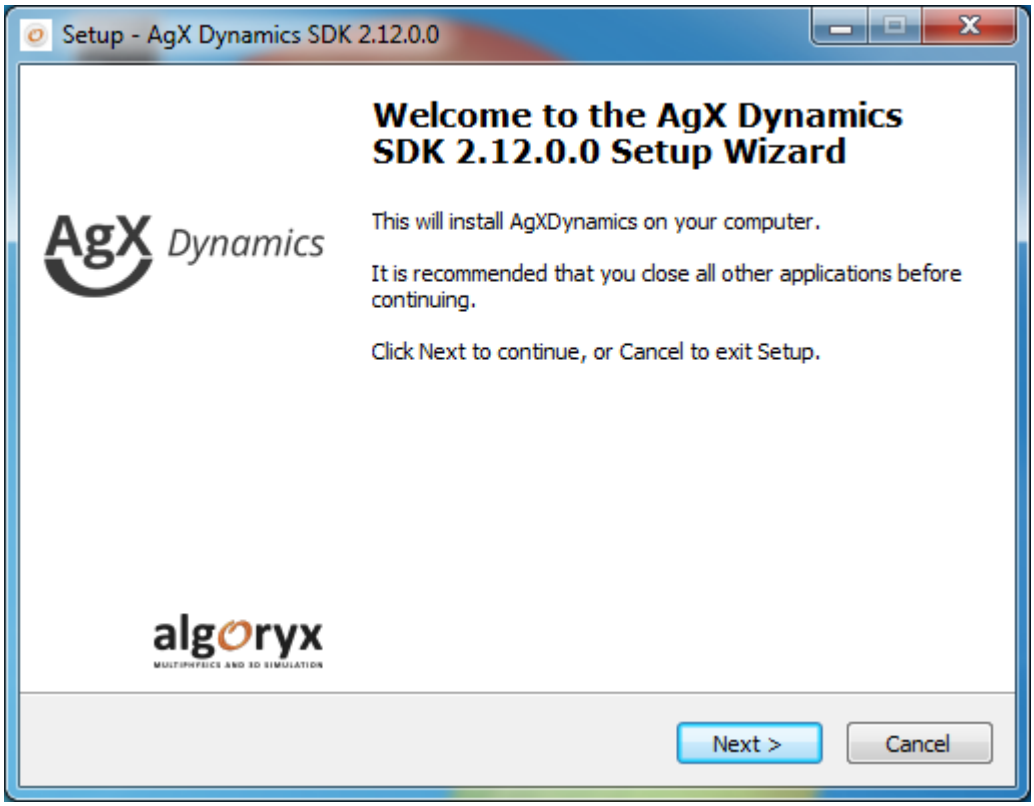


Uninstall window.

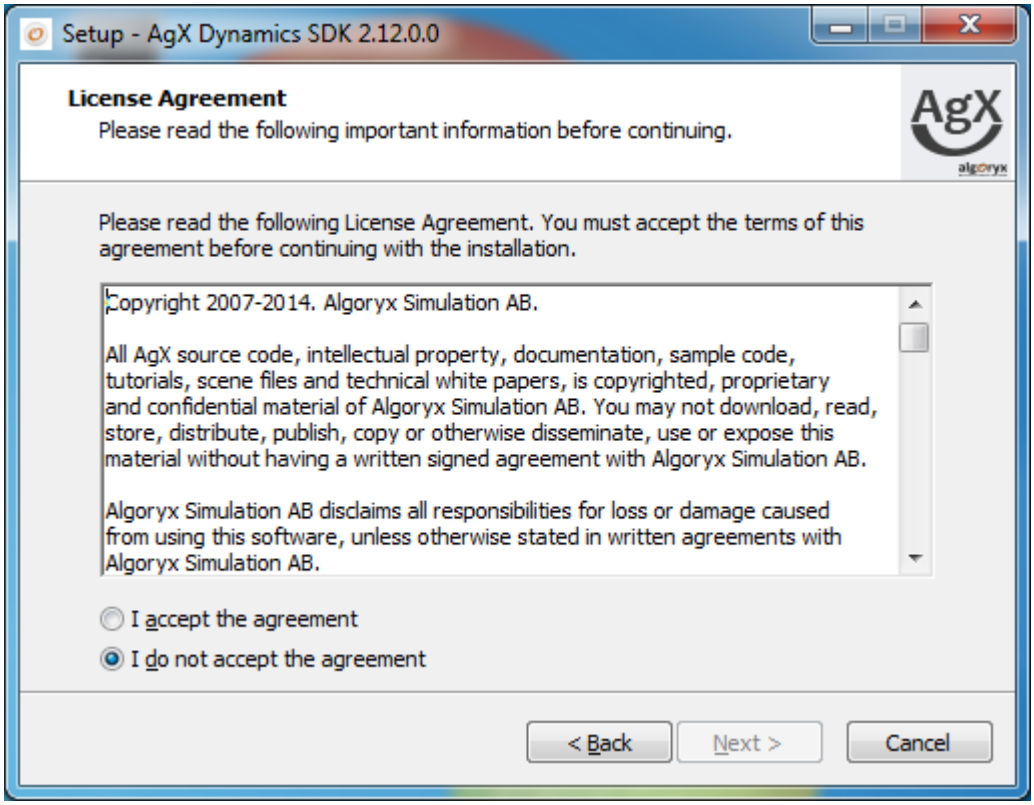


Message indicating uninstall is performed.

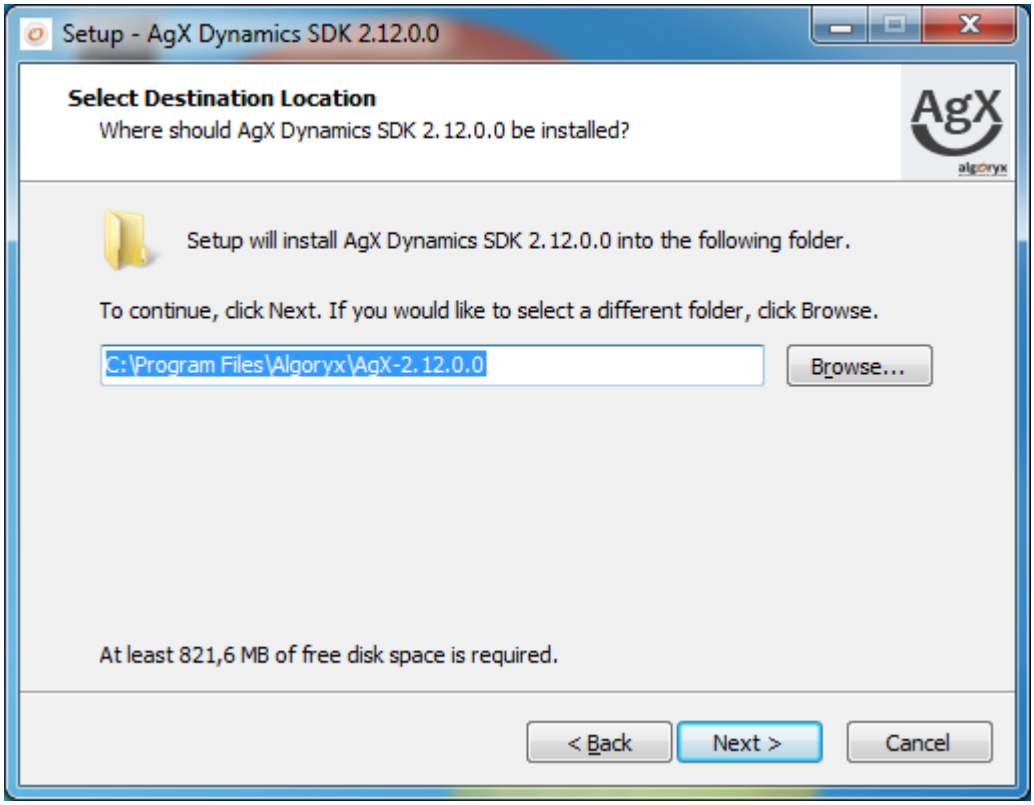
When the uninstall is finished. The installer will continue with the install of AGX.



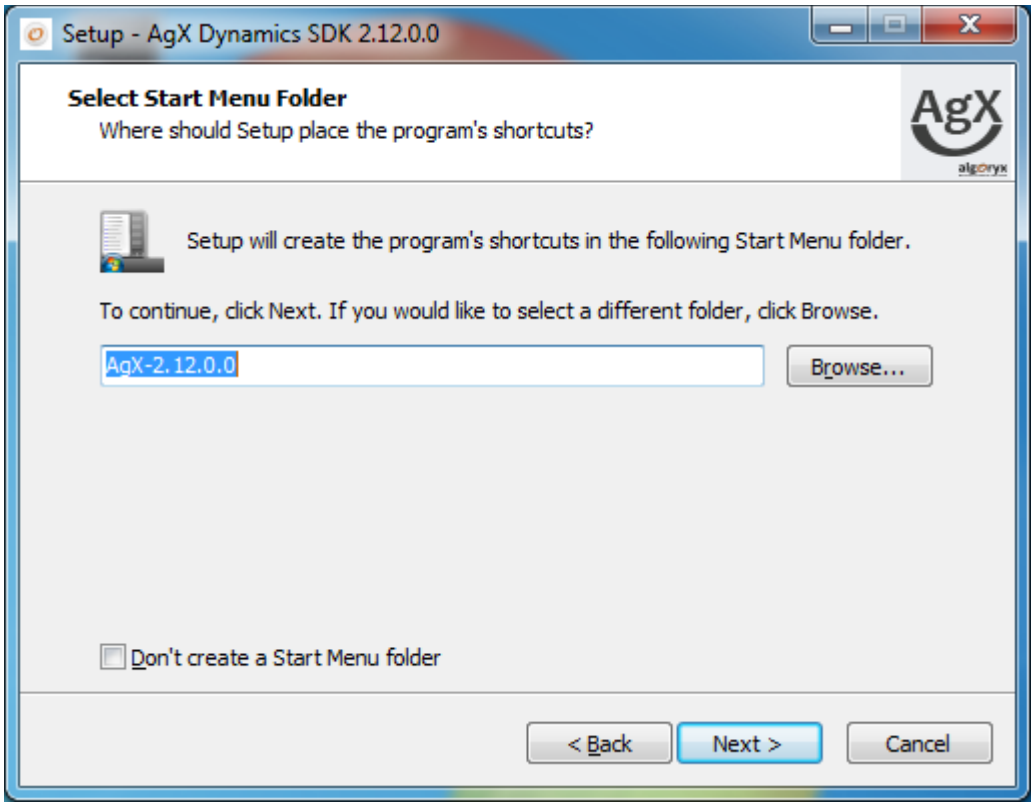
Initial screen



License agreement.

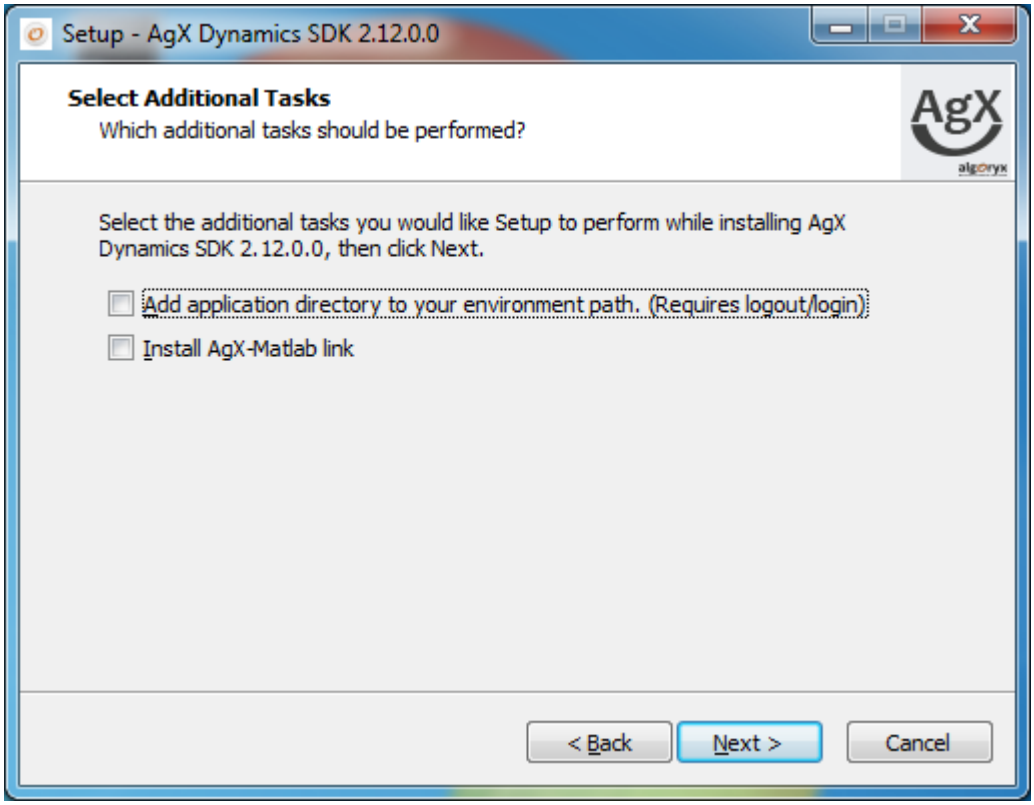


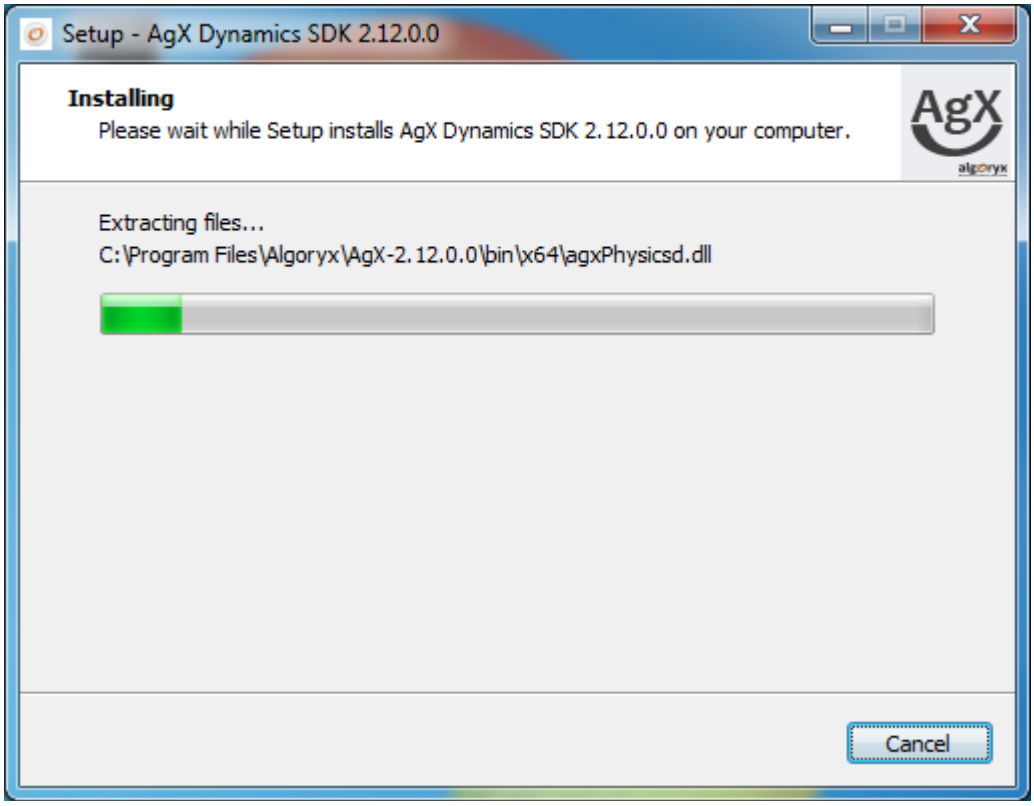
Install destination



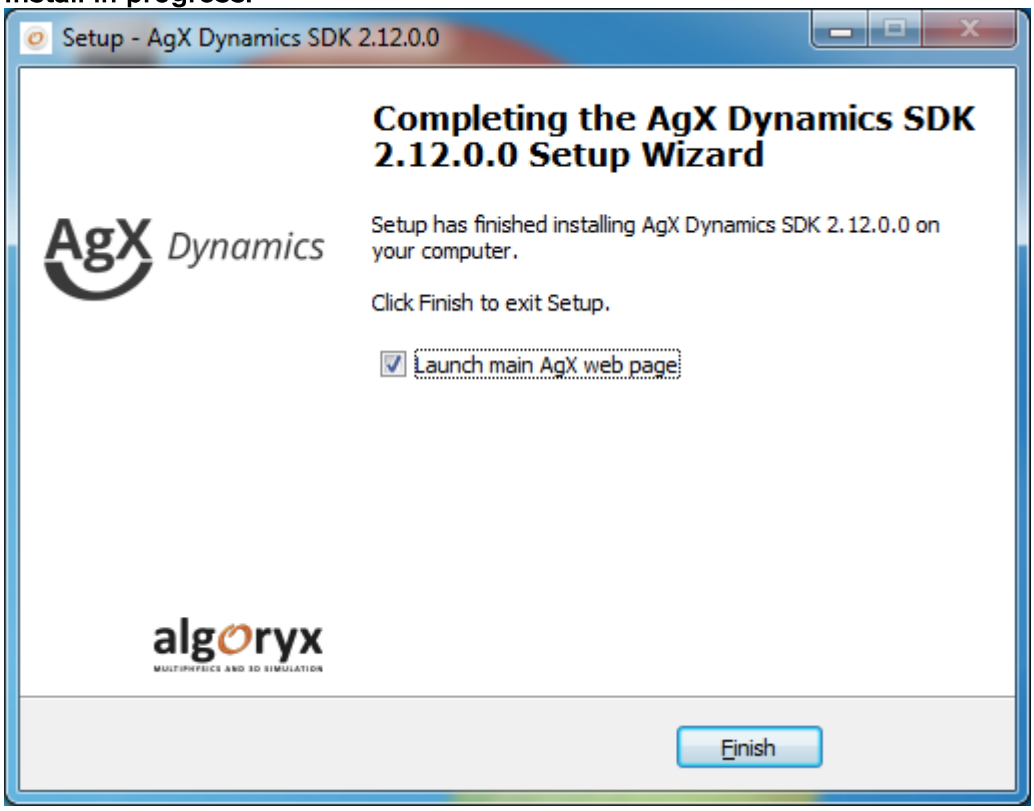
Name of entry in the Start menu.

It is optional to add AGX install directory to the path. This is only needed if you install the AGX-Matlab link.





Install in progress.



Installation done. If the checkbox is enabled, a webpage will be started with documentation, tutorials and examples:

6.2 Install on Ubuntu 16.04

Different Linux distributions and versions might have different installation procedures. The procedure described here matches the one used for installing a deb-file on Ubuntu 16.04.

By default, AGX Dynamics is installed into /opt/Algoryx/AgX-<version>

```
$ sudo dpkg -i agx-setup-2.16.0.3-x64-ubuntu_16.04-double.deb
(Reading database ... 220029 files and directories currently installed.)
Preparing to unpack agx-setup-2.16.0.3-x64-ubuntu_16.04-double.deb ...
Unpacking agx (2.16.0.3) over (2.16.0.3) ...
Setting up agx (2.16.0.3) ...
$ ls /opt/Algoryx/AgX-2.16.0.3/
AgX_build_settings.txt  bin             data     lib          setup_env.bash
agxOSG                  CMakeLists.txt  doc      LICENSE.TXT  tutorials
applications            CMakeModules    include  README.TXT
$
```

6.3 Install under MacOS

To install AGX on MacOS, double click the downloaded installer file and follow the instructions. The installed AGX environment will be available in `/opt/Algoryx/AgX-<version>`. You need to manually place your AGX license in the installation directory. Also, before running any AGX simulation the shell environment must be updated with some environment variables. Do this by starting a bash shell and execut the following command:

```
> source /opt/Algoryx/AgX-<version>/setup_env.bash
```

You can then start the tutorial applications or load Python/Lua scripts using agxViewer:

```
 > agxViewer beam.agxLua
```

6.4 Installing license

The license file agx.lic should be placed into the installation directory. Note, under windows that might require administrative rights if the installation directory is the default in c:\Program Files\

The license file can also be placed wherever you like, just as long as the file can be located with the agxIO::Environment utility. For more see information about how to specify paths to resource files in chapter 28.

7 Building your first application

AGX utilizes CMake for setting up a portable build environment. In the directory tutorials, there is a sample CMakeLists.txt which can be used as a template for building applications based on AGX. This chapter explains how to build the tutorials from a binary release of AGX on various platforms. It describes how to setup the include and library paths and which libraries are necessary to build an application based on AGX. First, make sure you have the correct libraries for your target platform (32/64 bit).

7.1 Windows/Visual Studio

AGX for the Windows platform is built using Visual Studio with runtime libraries from *MultiThreaded DLL* (*MultiThreaded Debug DLL* in the debug version). When you build your application, it is also important to link to the correct version of the libraries (debug/release), as it is not recommended to mix the two in the same application. AGX for windows is delivered with both debug and release libraries (see below).

When building the tutorials which come with AGX, the following steps have to be followed to build in-source (in the installation folder):

a. Install cmake (www.cmake.org) and add it to your system path
b. Open a command window (cmd). If you chose the default installation path, it will be in Program Files, where you need admin rights for writing – you should then start the cmd as admin. As an alternative, you can copy the whole installation folder into a separate folder where you have write access. This has the additional advanctage that you can modify the tutorial source files in this copied folder, and still have the originals left in the installation folder.
c. Create an empty directory somewhere on the disk
d. Execute setup_env.bat in the AGX Dynamics installation directory (or copied directory, if chosen above).
e. Change current directory to the empty directory
f. Execute cmake-gui (make sure you have CMake in your path) and supply the AGX installation directory (or copied directory, see above) as an argument.
g. Press configure and *select the correct generator* (VS2013, VS2015 *32/64* bit etc.) *that matches your installed AGX Dynamics binary distribution*. If not, you will get linking or runtime problems.
h. Change CMAKE_INSTALL_PREFIX to "." (just the dot).
i. Press Generate and close cmake-gui.
j. Now you can build the tutorials by double clicking the generated solution file and building.
k. Building the project INSTALL will copy the files to the folder earlier given by CMAKE_INSTALL_PREFIX.
l. However, unless you have AGX runtime libraries (and its dependencies) in the PATH, you will not be able to execute any of the programs.
m. To do this you can either (in a cmd window) execute setup_env.bat in the AGX installation folder or explicitly add the runtime library directory to the PATH environment variable.
n. Start Visual Studio with environment variables from setup_env: devenv /useenv agxBuild.sln
o. Build the solution
p. Build INSTALL.
q. Now you should be able to run and debug the tutorials. Select one of them (e.g. tutorial_cable) as a StartUp-project by right-clicking on it in the solution explorer, and choosing "set as StartUp-project".

r. Run externally with ctrl-F5, or debug it with F5.
s. Additional options can be passed via right-click in solution exporer – properties – debugging – Command arguments. One option would be --startPaused in order to start the simulation in paused state (press ‘E’ to toggle paused state while running).

Below is a session where the tutorials are prepared for building:

```
d:\temp>mkdir buildAgXTutorials
d:\temp>cd buildAgXTutorials
d:\temp\buildAgXTutorials>cmake-gui "c:\Program Files\Algoryx\AgX 2.12.0.0"
```

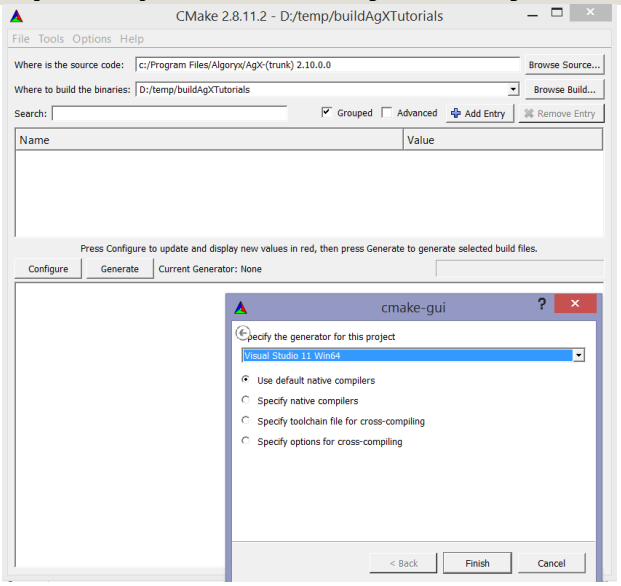


Figure 1: Press Configure and select generator

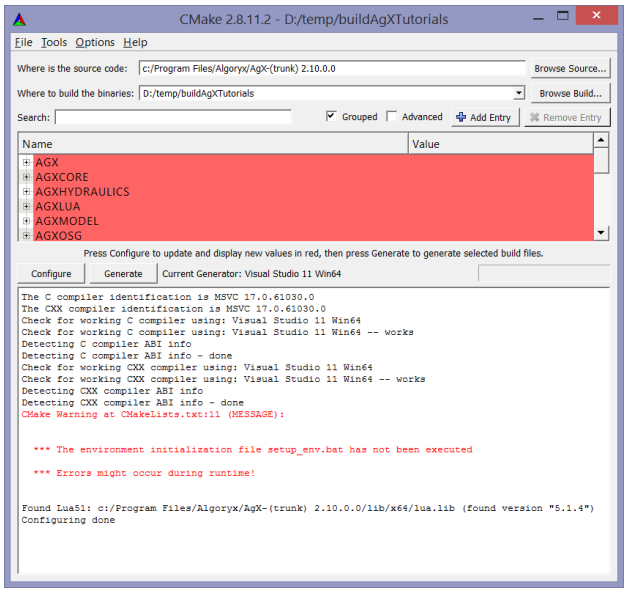


Figure 2: Warnings might occur if setup_env.bat is not executed.

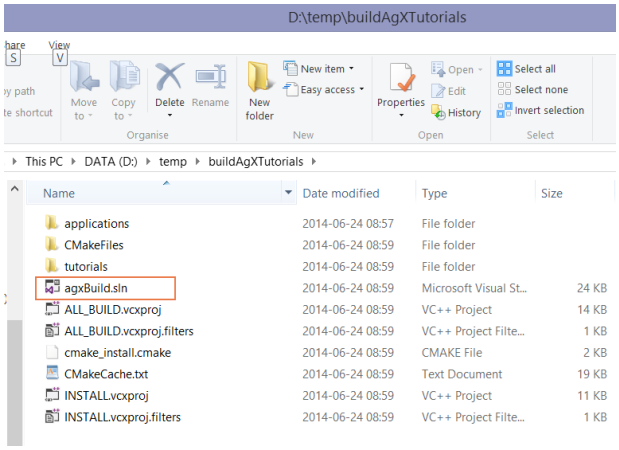


Figure 3: agxBuild.sln is the resulting solution file.

```
d:\temp\buildAgXTutorials>
d:\temp\buildAgXTutorials>dir Release\tutorial_bodies.exe
d:\temp\buildAgXTutorials>Release\tutorial_bodies.exe
d:\temp\buildAgXTutorials>set PATH=%PATH%;c:\Program Files\Algoryx\AgX 2.12.0.0\bin\x64
d:\temp\buildAgXTutorials>Release\tutorial_bodies.exe

d:\temp\buildAgXTutorials>Release\tutorial_basicSimulation.exe
        AGX Library 64Bit AgX 2.12.0.0 Algoryx(C)
        Basic simulation tutorial
        --------------------------------


Creating a simulation
Creating a body
Adding body to the simulation
Step system 3 seconds forward in time
Shutting down AGX

d:\temp\buildAgXTutorials>
```

If an AGX Dynamics application is started without the correct path you might get an error message regarding missing runtime libraries:

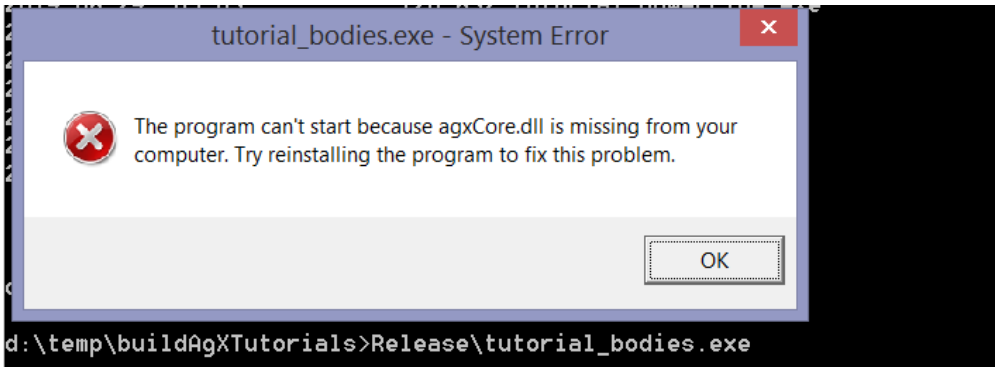


Figure 4: Starting an AGX application without correct PATH.

7.1.1 Building your own code

The CMakeLists.txt supplied with AGX Dynamics can be used for your own project. Just follow the instructions for building the tutorials, but first copy the <agx-install-directory>\CMakeLists.txt to your project and modify the line where the path to AGX is specified:

Change the line:

```
SET(AGX_INSTALL_DIRECTORY ${agxBuild_SOURCE_DIR})
```

to (obviously it should reflect the path to where AGX is installed):

```
SET(AGX_INSTALL_DIRECTORY "C:/Program Files/Algoryx/AgX 2.10.0.0"
```

Then modify the rest of the CMakeLists.txt to reflect your project. This should give you an overview of how to incorporate AGX into your own build system using CMake.

7.1.2 Compiling (without CMake)

If you want to build an application without using the CMake build tool you need to specify a few include paths:

- Path to AGX headers and dependencies (usually <agx-install-dir>/include)

The header file <agx/config.h> contains various settings that were used when AGX was built.

For Visual Studio you need to specify the include path to where the AGX headers are. Select your project and go to Project->Properties (Alt+F7). Specify the path to the include directory of AGX:

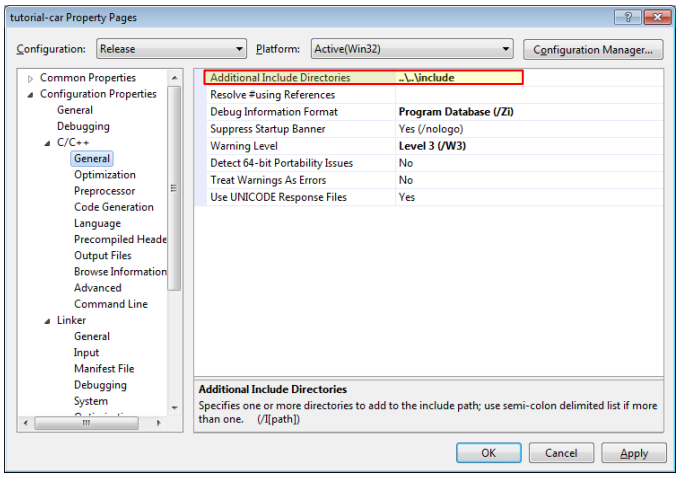


Figure 5: Specify include path for AGX.

7.1.3 Linking (without CMake)

First you need to specify the path to where the libraries can be found. Make sure you choose the correct platform (x86/x64):

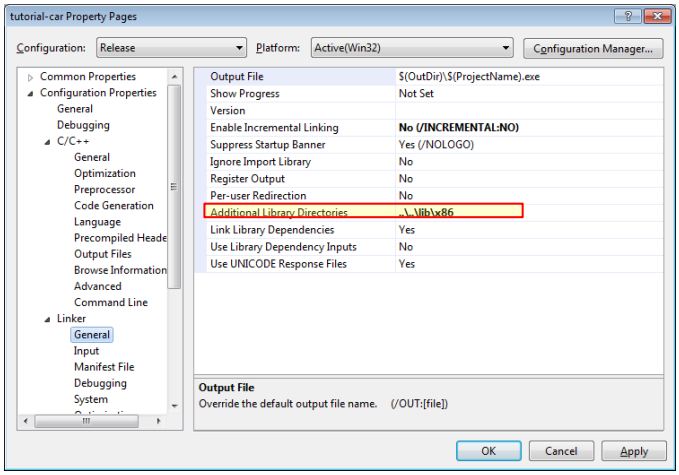


Figure 6: Specify library path.

To be able to link an application to AGX you will at least need the following libraries: **agxSabre.lib, agxCore.lib** and **agxPhysics.lib**. The tutorials also require **agxOSG.lib**, and osg.lib, osgViewer.lib, osgDB.lib (OpenSceneGraph for rendering). Debug version of the libraries has a suffix 'd', for example **agxSabred.lib**.

Additional functionality might require additional libraries. Such as Hydraulics, TireModel etc.

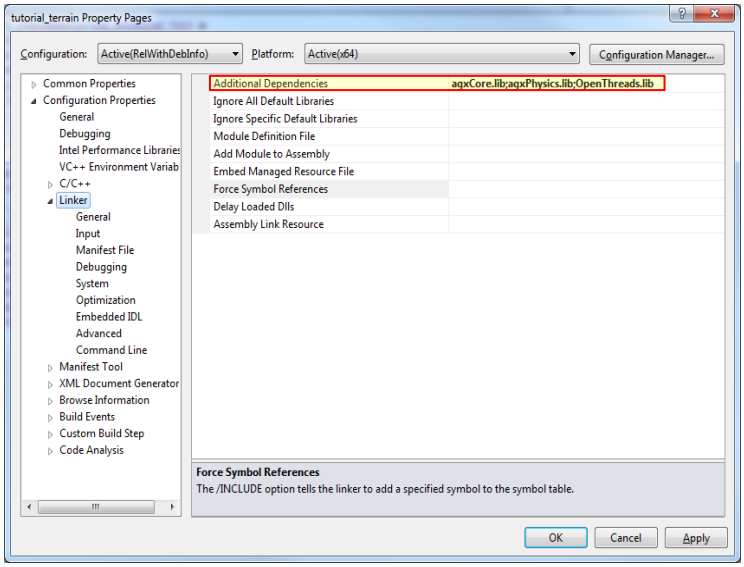


Figure 7: Specifying AGX libraries.

7.2 Linux/gcc

To build the tutorials from source code, you need CMake.

Also, you have to have the following tools and libraries installed which AGX Dynamics depends on:

7.2.1 Tools for building AGX Tutorials or own applications

```
cmake       // To create build files from CMakeLists.txt.
```

build-essential // GCC compiler.
cmake-curses-gui // Optional. For easier build configuration.
cmake-qt-gui // Optional. For easier build configuration.
ninja-build // Optional. Faster build system instead of good old make.
clang // Optional. Alternative compiler instead of gcc.

7.2.2 Libraries for building/running AGX binaries

zlib1g-dev // System library required by AGX.
libhdf5-serial-dev // System library required by AGX.
libpng12-dev // System library required by AGX.
libboost-random-dev // System library required by AGX.
libgl1-mesa-dev // System library required by AGX.
libglu1-mesa-dev // System library required by AGX.
libglew-dev // System library required by AGX.
libpcre++-dev // System library required by AGX.
libois-dev // Optional. For joystick/gamepad support.

ttf-mscorefonts-installer // Optional. Required in order for in-application text to render properly.

7.2.3 Building tutorials

```
# create an empty directory
%mkdir /tmp/agxBuild
%cd /tmp/agxBuild
# Configure cmake and create makefiles for the tutorials etc.
%cmake --DCMAKE_INSTALL_PREFIX /tmp/agxBuild /usr/home/agx-1.10.0
# Now run make using the created makefiles.
%make
```

7.3 Executing an AGX application

To be able to run an application build with AGX, you need to specify the path to the runtime libraries. See the documentation for your specific platform.

You also need to specify where the plug-ins and resource files, including the license file are located. This can either be done using environment variables (see chapter 41), or through the API (see chapter 0) for more details.

Part of AGX is built as modules, loaded during runtime. The path to these modules must be setup before a call to agx::init() (which loads the plug-ins).
Modules are also xml files, so both the RUNTIME_PATH as well as the RESOURCE_PATH have to be set to include the path to the module directory (usually placed in the bin directory of the AGX distribution).

#1 Specify the path to the plugin resource files (you are free to use '\\' or '/').

```
std::string pathToAgX = "c:\\Program Files\\Algoryx\\AgX 2.7.1.0\\";
agxIO::Environment::instance().getFilePath(
      agxIO::Environment::RESOURCE_PATH).push_back(pathToAgX+"/bin/x64/plugins");
```

#2 Specify the path for the runtime libraries/plugins:

```
agxIO::Environment::instance().getFilePath(
      agxIO::Environment::RUNTIME_PATH).push_back(pathToAgX+"/bin/x64/plugins");
```

#3 Specify the path to data files (.cfg, .schema, .agxLua, etc.)

```
agxIO::Environment::instance().getFilePath(
      agxIO::Environment::RESOURCE_PATH).push_back(pathToAGX+"/data");
```

You also need to make sure that the license file (agx.lic) exists in one of the directories specified for the RESOURCE_PATH.

7.3.1 Windows registry

For the Windows operating system, registry entries will be added by the AGX installer. These registry entries will be used whenever an AGX based application is executed. This *only* works if AGX is installed via the installer binaries, not when it is built from source.

7.3.2 Hello world

Below is a small example that demonstrates how to properly initialize AGX and create a simulation with one body.

```
#include <agxSDK/Simulation.h>
#include <agx/RigidBody.h>

int main()
{
  // Where AGX is placed/installed:
  std::string pathToAgX = "c:\\Program Files\\Algoryx\\AgX 2.13.0.0\\";

  // Probably where the license file is
  agxIO::Environment::instance()->getFilePath(
    agxIO::Environment::RESOURCE_PATH).pushbackPath(pathToAgX);

  // Text files for plugins
  agxIO::Environment::instance()->getFilePath(
    agxIO::Environment::RESOURCE_PATH).addFilePath(pathToAgX);

  // binary plugin files
  agxIO::Environment::instance()->getFilePath(
    agxIO::Environment::RUNTIME_PATH).addFilePath(pathToAgX + "/bin/x64/plugins");

  // resource files
  agxIO::Environment::instance()->getFilePath(
    agxIO::Environment::RESOURCE_PATH).addFilePath(pathToAgX + "/data");

  // AutoInit will call agx::init() which must be called before
  // using the AGX API creating resources such as bodies, geometries etc.
  agx::AutoInit init;
  {
    // Create a Simulation which holds the DynamicsSystem and Space.
    agxSDK::SimulationRef sim = new agxSDK::Simulation;

    // Create a rigid body (no geometry) with default mass etc.
    agx::RigidBodyRef body = new agx::RigidBody;

    // Add the body to the simulation.
    sim->add(body);

    // Simulate for 2 seconds
    // The Time step (dt) is determined by the so called TimeGovernour.
    // Changing the time step:
    sim->getDynamicsSystem()->getTimeGovernor()->setTimeStep(1.0 / 100.0);
    while (sim->getTimeStamp() < 2.0)
      sim->stepForward(); // Take one time step.
  }

  // The destructor for AutoInit will call agx::shutdown() automatically.
  // Unloads plugins, destroys threads etc.

  return 0;
}
```

7.4 Dependencies

AGX Dynamics has a number of dependencies built as runtime libraries (.dll in windows). Table 3 below list the required runtime libraries for executing an application linked against AGX Dynamics under the Windows platform.

Name	Description
AGXCORE.DLL	AGX Dynamics core library
AGXPHYSICS.DLL	AGX Dynamics core physics library
AGXSABRE.DLL	AGX Dynamics sparse blocked solver
COLAMD.DLL	Approximate minimum degree column ordering algorithm
GLEW.DLL	OpenGL Extension Wrangler Library
LIBPNG.DLL	PNG Image reader
MSVCP140.DLL*	Microsoft runtime libraries
MSVCR140.DLL*	Microsoft runtime libraries
OT20-OPENTHREADS.DLL**	OpenThreads runtime library
WEBSOCKETS.DLL	Mozilla web socket library
ZLIB.DLL	ZLib compression library

Table 3: Required runtime libraries.
*Depends on which version of Visual Studio that is being used.
**Depends on which version of OpenThreads that is being used

Table 4 list optional runtime libraries that might be used depending on the selected functionality in your application. If you use any classes from the agxModel name space, you will for example need to include the agxmodel.dll runtime library. The sample rendering framework utilizes OpenSceneGraph, which will require the files names osg*.dll.

<agx-dir>\bin\<platform>\plugins\ScriptPlugin contains plugins for the Lua scripting environment. If the AgXLua scripting is not part of your application, then this directory is not required.

AGXLUA.DLL	AGX Dynamics Lua scripting library
AGXMODEL.DLL	AGX Dynamics model library
AGXOSG.DLL	AGX Dynamics OpenSceneGraph wrapper
AGXLUA.DLL	AGX Dynamics Lua scripting library
LUA.DLL	Lua scripting language runtime library
OSG141-OSG.DLL	OpenSceneGraph runtime library
OSG141-OSGDB.DLL	OpenSceneGraph runtime library
OSG141-OSGGA.DLL	OpenSceneGraph runtime library
OSG141-OSGSHADOW.DLL	OpenSceneGraph runtime library
OSG141-OSGSIM.DLL	OpenSceneGraph runtime library
OSG141-OSGTEXT.DLL	OpenSceneGraph runtime library
OSG141-OSGUTIL.DLL	OpenSceneGraph runtime library
OSG141-OSGVIEWER.DLL	OpenSceneGraph runtime library

Table 4: Optional runtime libraries.

Additionally there is a list of data files *required* to run AGX Dynamics. These files contains *entity and kernel descriptions.* Entity is a core part of AGX and describes how data at a low level is stored and handled. Kernels describe execution units which operate on entities.

This data is located in the <agx-dir>/bin/<platform>/plugins/components.

7.5 Distributing an AGX Application

If you want to distribute a minimal runtime environment which contain AGX, you need to at least supply the following files:

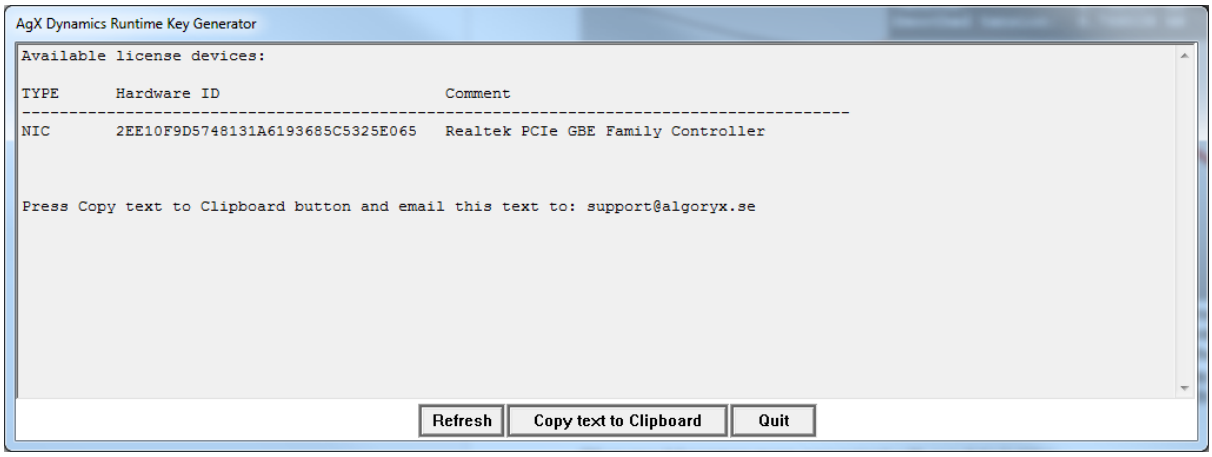
- All files from Table 3 (located in bin/x64/)
- Files in bin/x64/plugins/Components/
  - except for bin/x64/plugins/Components/Web
- data/cfg/settings*.*

8 License system

An application built with AGX requires a valid license to run. AGX can either be licensed with a *runtime*-license, or a *developer license.* A runtime-license usually does not have a time limit, but needs to be tied to a network adaptor in your computer.

License information for a node-locked license is generated through a three step process:

1. Execute the executable file RuntimeKeyGenerator which comes with the AGX distribution. This will generate a number of Hardware ID:s based on: network adaptors (NIC) and USB memory sticks (floating license).



2. Copy one of the Hardware ID:s and send the following information to support@algoryx.se:

   - User/company name
   - Contact/email
   - HardwareID (select one whole line from the output of RuntimeKeyGenerator)
   - Url (Personal or your company's)

3. You will receive a file in email which should be named agx.lic and placed in one of the directories specified by the environment variable AGX_FILE_PATH, or with the API agxIO::Environment::getFilePath(agx::Environment::RESOURCES) (*see chapter* 27).

8.1 Entering license information through API

It is possible to register the license information without having the license file. This can come in handy if you want to compile your application including the license file. Read the license file content into a const char * string and call:

```
bool valid = agx::Runtime::instance()->unlock( licenseFileContent );
if (valid)
  // We got a valid license file
```

9 Unicode

AGX uses utf-8 internally. One of the reasons is that it is backwards compatible with ascii. It will also save storage. std::string/agx::String works just as it is as a storage container for utf8 strings.

In <agx/Encoding.h> there are a number of utility methods for converting between Wide and utf8.

So for example, reading a string encoded in utf8 from the windows registry requires you to convert from utf8 to Wide:

```
const char *keyInUtf8; // Initialized with some utf8 string
std::wstring wKey = agx::utf8ToWide( key ); // Convert to Wide character string

agx::String& value; // Here is where we store the result

dwType=0;
DWORD dwLen=0;
status = RegQueryValueExW(hkey, wKey.c_str(), 0,&dwType,nullptr, &dwLen);

if (status== ERROR_SUCCESS &&  dwType == REG_SZ)
{
  agx::Vector<wchar_t> data;

  // Since we use wchar, this will overallocate, but that's ok.
  data.resize(dwLen+1);

  // Now get the value
  status = RegQueryValueExW(hkey, wKey.c_str(), nullptr,nullptr,(PBYTE)(data.ptr()), &dwLen);
  if (status== ERROR_SUCCESS)
  {
    // If we use RegQueryValueEx and UNICODE is not set, then the API call
    // will be RegQueryValueExA that will convert from wstring to ANSI if
    // the data in the registry key is written as unicode.
    //
    // We don't want that conversion since it might not be safe and having to mess
    // around with code pages is not fun.

    value = agx::wideToUtf8( std::wstring(data.ptr() ) );
    found = true;
  }
  RegCloseKey(hkey);
}
```

9.1 Limitations

Applications such as agxViewer, luaagx will not be able to read files in a path containing utf8 characters.

C++ streams do not handle unicode paths in windows, and there are no open methods that handle wstring. Hence basic_ios and classes such as ifstream must not be used for accessing files in paths with non-ascii characters.

10 Math classes and conventions in AGX

There are numerous classes related to math operations and conversions in AGX. This chapter tries to describe the conventions used for these classes. **agx::Real** is the data type used for all floating point operations in AGX. Common for all rotations is that *radians* are used for all methods. **agx:degreesToRadians()** and **agx::radiansToDegrees()** can be used to convert between degrees and radians. The header file <agx/Math.h> contains utility methods related to math operations such as:

absolute(a)	Return absolute value of argument.
equivalent(a, b)	Return true if the two arguments are equal.
equalsZero(a)	Return true if argument is equal to zero.
minimum(a, b), **maximum**(a, b)	Return minimum or maximum of the two arguments.
clamp(a, min, max)	Return clamped argument a below min and max.
square(a)	Return square of a.
signedSqare(a)	Return squared argument a.
clampAbove(a, minimum)	Return clamped argument above minimum.
clampBelow(a, maximum)	Return clamped argument below maximum.
sign(a)	Return sign of argument.

10.1 Small vectors

Vec2, Vec3 and Vec4 are vectors for storing points and vectors in 2, 3 and 4 dimensions. Each method has methods for dot product, cross products and overloaded operators for +,-, scalar operations such as *, / etc.

10.2 agx::AffineMatrix4x4

This is a 4x4 matrix that can store a concatenated rigid rotation and translation. It is stored in row order:

R_{x1}	R_{x0}	R_{x2}	0
R_{y1}	R_{y0}	R_{y2}	0
R_{z1}	R_{z0}	R_{z2}	0
T_y	T_x	T_z	1

A vector-matrix multiplication is performed as: v_M= M*v and its transpose is calculated as v_M'=v*M.

The AffineMatrix4x4 is always considered to be orthonormalized, which means that the call to AffineMatrix4x4::inverse() will perform a transpose, as the transpose of a orthonormalized matrix is the same as the full inverse of the same. An AffineMatrix4x4 should never contain scaling or shearing, only rigid transformations.

10.3 agx::OrthoMatrix3x3

This is a rotation matrix with the same conventions and methods (except everything related to translation) as the AffineMatrix4x4.

R_{x0}	R_{x1}	R_{x2}
R_{y0}	R_{y1}	R_{y2}
R_{z0}	R_{z1}	R_{z2}

10.4 agx::Quat

Quaternion is a compressed representation of a rotation based on a vector and a scalar: q = [<x, y, z>, w], where <x, y, z> is a three dimensional vector and w is a scalar. Quaternions can be interpolated using linear interpolation: agx::lerp(t, q1, q2) or spherical linear interpolation: Quat::slerp(t, q1, q2);

10.5 agx::EulerAngles

EulerAngles is a more intuitive way of specifying rotations, as rotations can be specified per axis. There are 24 different conventions for representing rotations in Euler. This class can convert back and forth between all representations. Most classes can convert between the different rotation specifications.

11 Reference pointers

AGX use a reference pointer system for all objects allocated on the heap. Reference pointers are a special kind of pointer that can only point to objects derived from **agx::Referenced**. A Referenced object keeps track of the number of reference pointers pointing to it. Whenever an instantiated class derived from Reference discovers that the number of referencing points is zero, it will automatically be de-allocated.

There are two types of reference pointers in AGX, ref_ptr and observer_ptr. ref_ptr is a *strong* pointer, which will during its lifetime keep the referenced object from being deleted. An observer_ptr on the other hand, will be notified when the referenced object is de-allocated. ref_ptr and observer_ptr both has a method isValid() that return false if it is pointing to a deallocated object. The type-cast operator is overloaded, so a ref_ptr<A> and observer_ptr<A> can be used anywhere in the code, as if it was of type A *. Example:

```
struct A : public agx::Referenced
{
};
agx::ref_ptr<A> a;
A* aPtr = a; // Ok, as the reference pointer will be cast to a A*
```

There are a few pitfalls when using reference pointers, an example:

```
// Problem #1: Local ref_ptr going out of scope
A* createA()
{
  agx::ref_ptr<A> aRef = new A;

  // When this function returns, aRef will go out if scope and the
  // memory be deallocated. The number of reference pointers pointing to
  // this object is decremented to zero, and it will be deallocated.
  // So we are returning an invalid object!
  return aRef;
}
```

In the above example one should either change the return type of the function to return an agx::ref_ptr<A>, or avoid using a ref_ptr inside the function. The latter solution will however be a potential memory leak, because memory will not be freed if an exception is thrown inside createA().

12 Building simulations: agxSDK

The namespace agxSDK contains classes that bridges agx (dynamics) and agxCollide (collision). All insertion/deletion of objects such as rigid bodies, geometries, and materials should go through the main class agxSDK::Simulation to make sure that that everything is registered/executed in the correct manner.

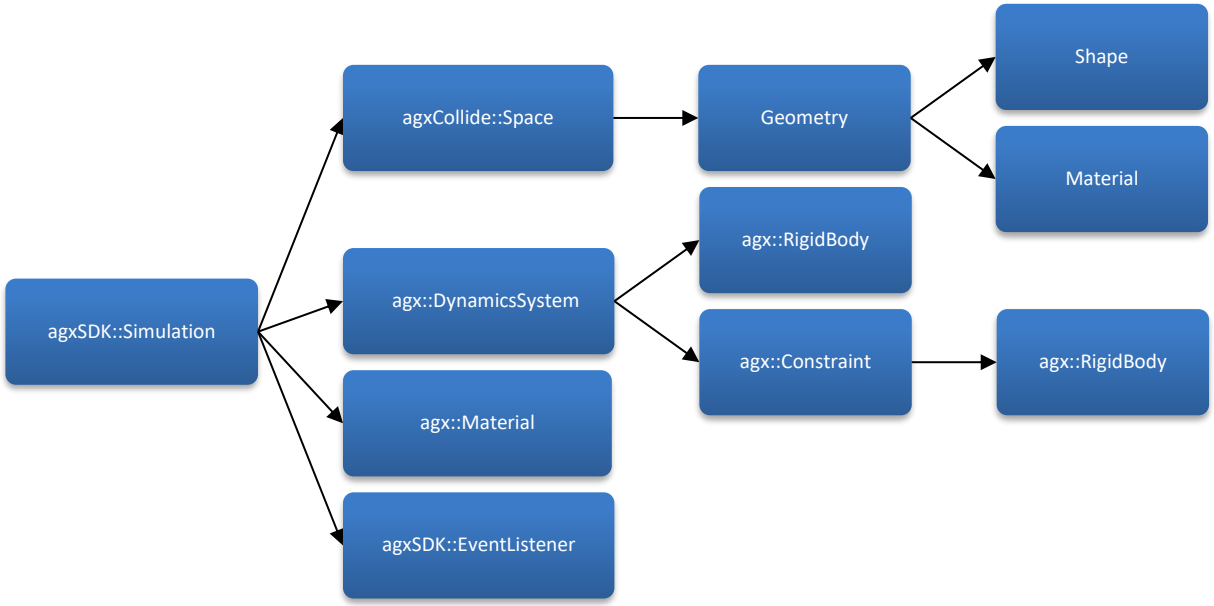


Figure 8: Structural description of a Simulation

Figure 8 show the structure of a simulation with the n-ary relationships between the various classes. An agxSDK::Simulation contain one agxCollide::Space and one agx::DynamicsSystem, it also contains any number of agx::Material and any number of agxSDK::EventListeners etc.

12.1 Simulation

The class **agxSDK::Simulation** is the class encapsulating most aspects regarding building and executing a simulation. This class holds a reference to an **agx::DynamicsSystem** that is responsible for integrating physical bodies and solving constraint equations, and an **agxCollide::Space** which generate contacts between geometries. All entities that should be part of a running simulation, such as Geometry, Constraint, and EventListener have to be added to the simulation. There can be several instances of the class Simulation, and they will each have their own Space and DynamicsSystem.

Always add/remove objects through the agxSDK::Simulation interface, not via DynamicsSystem or Space. However Space/DynamicsSystem can be used to access objects, such as getting a vector of all Geometries etc.

12.1.1 Timeline for Simulation::stepForward()

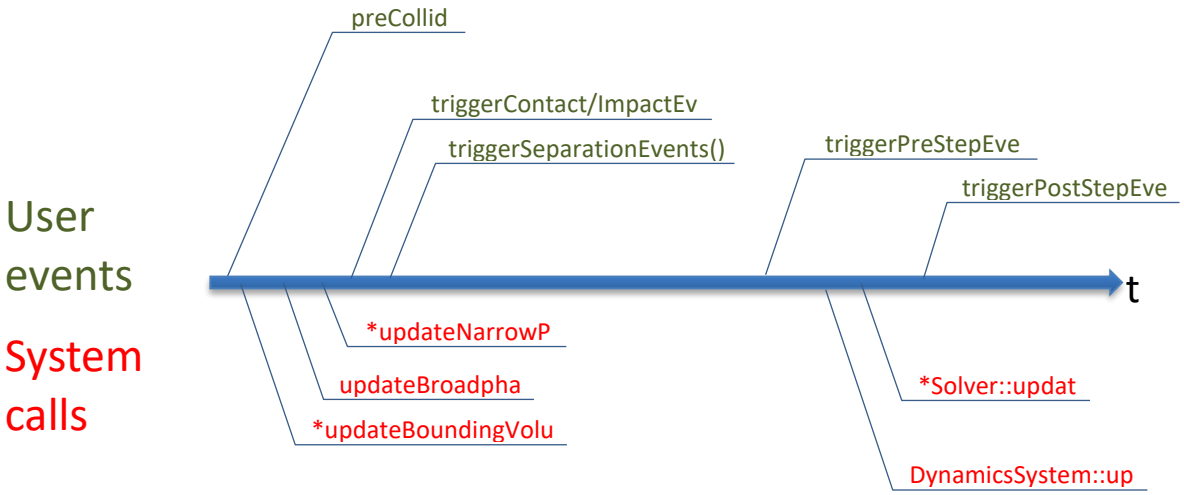


Figure 9: Timeline illustrated for a Simulation::stepForward() call. (* indicates parallelizable steps).

Figure 9 show a very general picture of flow of events during a call to Simulation::stepForward(). An asterix (*) indicate that the step is possible to parallelize. *User events* mean that it is possible to inject user code using EventListeners. In reality, there are many, many more tasks executed, many of them parallelizable.

12.1.2 Initialization and shutdown of AGX

To properly initialize AGX, an initialization method needs to be called before creating an AGX object (rigid bodies, geometries, constraints etc.):

```
agx::init();
```

Various resources in AGX are handled by Singletons, classes that ensure that only one instance exists. AGX depends on various plugins (components etc.) which are all initialized at the call to agx::init. To properly call the destructors and shut down any threads the last call to AGX should be a call to agx::shutdown():

```
agx::shutdown();
```

The shutdown method is used to release any resources currently allocated by AGX. This includes loaded plugins, running threads, search paths (Environment) etc.

New in 2.3.0.0: agx:init() can be called again after a call to agx::shutdown() to re-initialize AGX for use.

To properly shutdown any running threads, a call to **agx::shutdown()** should be done *before* the end of scope of main(). You should not register an atExit() callback and execute the call there, as this callback will be executed after the scope of main has ended. So always end your application with a call to the shutdown() method.

agx::AutoInit is a class that will upon construction call agx::init(), and upon its destruction call agx::shutdown(). This class can be used for handling init/shutdown within a scope. Hence, if an exception is thrown and not caught, the method agx::shutdown will still be called.

Just remember that any paths specified for agxIO::Environment will be lost after a call to the shutdown method, this goes for other resources too. You will need to re-initialize these to whatever value you want them to have, just as you did when you first started the application (initialized AGX).

12.1.3 Shutting down threads

Having running threads when leaving the scope of main can on the Windows platform cause hanging processes. Therefore, always make sure you either call agx::shutdown() or call the static method **agx::Thread::shutdown()** *before end of the scope of main*.

12.2 agx::DynamicsSystem

The DynamicsSystem is responsible for all rigid bodies and constraints in a simulation. It sets up a toolchain with solver, integration, etc. The DynamicsSystem stores an instance of an **agx::TimeGovernor** which is responsible for supplying a time step.

DynamicsSystem also has an instance **agx::MergeSplit** (section 30) which handles the merging/splitting of rigid bodies.

12.3 agxCollide::Space

Space handles geometric overlap tests such as broad phase tests for testing bounding volume overlaps, and near phase tests which result in detailed contact information: contact point, normal and penetration depth.

12.4 Events

Events are used for adding user code into the simulation. Types of Events implemented in AGX: *add/remove events*, *collision events*, *step events* and *gui* events. Event listeners are classes that can be implemented by the user through specialization of base classes supplied by the agxSDK namespace. All listeners are registered to the Simulation class and are triggered by the system whenever appropriate. All instances of the class EventListener have the methods:

- addNotification
- removeNotification.

These methods are called whenever an event listener is added/removed from a simulation and can be used for initialization etc. Those methods cannot be filtered away, they will always be executed.

The method **EventListener::getSimulation()** is used to get a reference to the Simulation that the listener is currently registered to.

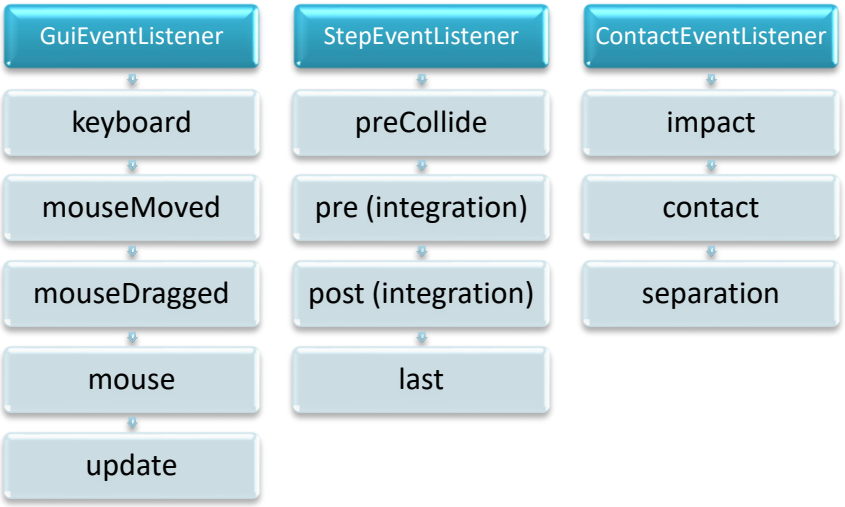


Figure 10: Available event listeners and methods.

12.4.1 Filter events

Event listeners can be masked/filtered, so that they only are invoked for selected events. Each sub-class of agxSDK::EventListener has a method setMask() and a number of enums specifying for which events the class will be activated. For example, specifying that a ContactEventListener should only react to IMPACT and SEPARATION events can be done as follows:

```
agx::ref_ptr<MyContactEventListener> l = new MyContactEventListener;
unsigned int mask = agxSDK::ContactEventListener::IMPACT|
                        agxSDK::ContactEventListener::SEPARATION;

// Only react to IMPACT and SEPARATION events.
l->setMask( mask );
```

Some event listeners have more detailed filters. For ContactEventListener, see the subchapter “Filters” below.

12.4.2 Order of execution

The order in which event listeners are executed can be controlled using a priority. This is available for ContactEventListeners and StepEventListener, but not currently for GuiEventListeners. For EventListeners with the same priority, the order should be considered undefined.

```
simulation->add( listenerA );
simulation->add ( listenerB ) ;
```

In the above example, listenerA *can* be executed before listenerB, but the opposite is equally true.

The order in which Contact/Step event listeners are triggered should be considered undefined when they all have the same priority. This can affect for example a ContactEventListener that removes a contact.
There is no guarantee that other ContactListeners will not see that contact.

```
simulation->addEventListener( listenerA, 10 ) ;
simulation->addEventListener( listenerB, 20 ) ;
```

In the example above, listener B will be executed before listener A. Priority is only valid within the range:

```
[EventManager::LOWEST_PRIORITY, EventManager::HIGHEST_PRIORITY]
```

12.4.3 Collision Events

During the discrete simulation of a dynamic system, geometric interference calculations are done. They are done in two different steps, one for simpler bounding volumes (currently Axis-Aligned-Bounding-Boxes, AABB) testing for overlap, a process called *broad phase* test.

For each pair of AABBs that overlap, a more detailed *narrow phase* test is performed, which can have two outcomes: intersect or disjoint. Disjoint means that even though the bounding volumes overlap, there is still no intersection between the underlying geometries. If the result is intersect, contact data such as contact points, normals and penetration depth are calculated in a class named **agxCollide::GeometryContact**. Each intersection is stored in a list for later use (events and simulation).

The events generated by the *narrow phase* test are: *IMPACT, CONTACT* and *SEPARATION.* IMPACT occurs when two geometries intersect for the first time. CONTACT occurs when two geometries that in the previous time step caused an IMPACT event to occur. SEPARATION occurs when two geometries that have intersected (and generated either an IMPACT or CONTACT event) are now disjoint.

Examples of event orders are:

IMPACT->SEPARATION->IMPACT->SEPARATION
IMPACT->CONTACT->CONTACT->SEPARATION

Collision events are caught by implementing a specialization of a class named **agxSDK::ContactEventListener** and overriding virtual methods with names corresponding to the events.

A ContactEventListener can also be told to listen only to a few of the above events by using the **setState()** method.

Event	Interface	Description
IMPACT	KeepContactPolicy **impact**(const agx::TimeStamp& agxCollide::GeometryContact *gc)	Called upon first impact between two geometries. No prior contact.
CONTACT	KeepContactPolicy **contact**(const agx::TimeStamp&, agxCollide::GeometryContact *)	Called when two geometries have narrow phase contact. Called after **impact**.
SEPARATION	void **separation**(const agx::TimeStamp& t, agxCollide::GeometryPair& cd);	Called when two previously overlapping geometries separates from narrow phase test.

Table 5: Available virtual methods for ContactEventListener

```
class MyContactListener : public agxSDK::ContactEventListener
{
public:
  KeepContactPolicy impact( const agx::TimeStamp&,
                            agxCollide::GeometryContact *gc);
};
```

EventListeners only operate together with an agxSDK::Simulation. Explicitly calling Space::update() will NOT generate any callbacks to event listeners and might result in missing events.

12.4.3.1 Filters

A *ContactEventListener* is activated during contact between two geometries. The question is how to specify for which two geometries the listener should be activated?

agxSDK::ExecuteFilter is a base class which is specialized by a number of classes:

- **agxSDK::GeometryFilter** - can be used to specify which Geometry or pair of Geometries should trigger a listener.
- **agxSDK::PropertyFilter** (String, Int, Float, …) - can be used to specify a property or pair of properties that should trigger a listener.
- **agxSDK::RigidBodyFilter** - can be used to specify which RigidBody or pair of RigidBodies should trigger a listener.
- **agxSDK::RigidBodyGeometryFilter** - can be used to specify which combination of RigidBody and Geometry should trigger a listener.
- **agxSDK::CollisionGroupFilter** - can be used to specify which combination of collision groups should trigger a listener.
- **Your own** – by inheriting from **agxSDK::ExecuteFilter** and overriding some of its functions, you can specify your own criteria.

Examples of *ContactEventListeners* can be found in the examples/agxSDK directory.
A filter is bound to the ContactEventListener:

```
agx::ref_ptr<MyContactListener> listener = new MyContactlistener;
listener->setFilter( new agxSDK::GeometryFilter(geom1) );

simulation->addEventListener( listener ):
```

12.4.4 Modifying contacts

The methods **impact** and **contact** have a return type of **KeepContactPolicy.** This is an enum with the following values:

KEEP_CONTACT	This contact will be kept and used in the contact solver. (if all other ContactEventListener also return KEEP_CONTACT for this contact)
REMOVE_CONTACT	This contact will be removed AFTER all other ContactEventListener have operated on it.
REMOVE_CONTACT_IMMEDIATELY	This contact will be removed directly after the return of this method, and no other ContactEventListener executed after this listener will see the contact.

Table 6: Values for the enum KeepContactPolity

So by using the values in Table 6 one can specify whether a contact should be kept or not after the callbacks has been executed.

Any data in an **agxCollide::GeometryContact** structure can be modified in a contact callback. If the methods *impact* and *contact* is to return REMOVE or REMOVE_IMMEDIATELY, the contact will be removed from the list of contacts.

12.4.5 Calculating relative velocity

Given that two geometries overlap in space, the result can be one or more intersection points.

The class **agxCollide::GeometryContact** contain all the information necessary for the solver to calculate the required impulses/forces to restore the overlap. This information can also be of use for the user of the toolkit. At a contact event, such as IMPACT, or CONTACT, the contact information is readily available as an argument to the event methods:

```
KeepContactPolicy impact( const agx::TimeStamp&,
                                agxCollide::GeometryContact* gc);

KeepContactPolicy contact( const agx::TimeStamp&,
                                 agxCollide::GeometryContact* gc);
```

The argument *gc* is a pointer to a buffer containing all overlaps in the system after the last update to the collision detection system.

The actual overlap data is available as:

```
for( size_t i =0; i < gc->points().size(); i++)
{
  ContactPointBuffer& p = gc->points()[i];

  p.normal(); // contact normal
  p.point();  // point of contact
  p.depth();  // penetration depth
  agx::Vec3 contactVel = gc->calculateRelativeVelocity( i );
}
```

The last call to the method **calculateRelativeVelocity(i)** calculates the relative velocity between two colliding rigid bodies. The argument *i* specifies the index in the points vector. This method will return a velocity in world coordinates that can be used for calculating sound etc.

After the solver is done, the ContactPointBuffer will also be updated with force information for each contact:

```
ContactPoint& p = gc->points()[i];
agx::Real normalForce = p->getNormalForceMagnitude();
agx::Real tangentialForce = p->getTangentialForceMagnitude();
```

However, this information is not available in a contact event listener, as this event occurs before the solver is done. So to access this information, you need to get a reference to the vector of contacts directly from agxCollide::Space, or by storing a pointer to the ContactPoint which you are interested in.

Remember that contact data is not valid between calls to agxSDK::Simulation::stepForward()

Below is an example of how to access the actual vector containing points from within an event listener Remember that all event listeners has a method for accessing the simulation in which they exist:

```
const agxCollide::GeometryContactPtrVector& contacts = getSimulation()->getSpace()-
>getGeometryContacts();
```

12.4.6 Step Events

A step event is generated before and after a discrete step is taken in the simulation. The two events generated are *PRE* and *POST*. PRE is generated after the collision detection phase but just before a step is taken in the dynamic simulation. POST is generated directly after a step is taken in the dynamic simulation.

agxSDK::StepEventListener is a base class which can be specialized by the user by inheritance.

Event	Interface	Description
PRE_COLLIDE	void **preCollide**(const agx::TimeStamp&)	Called before collision detection is performed. Last chance to add Geometries/Shapes to Space.
PRE_STEP	void **pre**(const agx::TimeStamp&)	Called before updating the dynamics information in the system (solver, integrating velocity/position).
POST_STEP	void **post**(const agx::TimeStamp&)	Called after the update of dynamics information. Bodies/Geometries have updated transformations. Contacts contain forces etc.
LAST_STEP	void **last**(const agx::TimeStamp&)	Called last in Simulation::stepForward(). Things done here will not be included into the timing for agxSDK::Simulation::stepForward() anymore.

Table 7: Available StepEventListener events.

12.4.7 GUI Events

GUI events are keyboard presses and mouse events. The Simulation class does not know of how to read of the mouse/keyboard for any specific system. Instead an abstract class agxSDK::GuiEventAdapter can be specialized and implemented for a specific system which will inject appropriate events into the Simulation object. An example of an implementation can be found in agxOSG::GuiEventAdapter which can be used together with OpenSceneGraph to generate GUI events.

To be able to receive GUI events a class named **agxSDK::GuiEventListener** is specialized and implemented. Examples of this can be found in `tutorials/tutorial_basic1.cpp`

12.5 agx::Frame

The class agx::Frame is used to handle multiple coordinate systems. This class contains a transformation (AffineMatrix4x4), velocities, parent/child relations and methods for converting points and vectors between local and world coordinate systems. Parent/child relationship can be constructed with the **Frame::setParent(Frame *)** method. This makes it possible to build up a scene of transformations.

Many classes in AGX such as RigidBody, Geometry, Assembly, etc. use an agx::Frame internally to store transformations and velocities.

An example of how to build a structure with parent/child relationship using Frames:

```
using namespace agx;
FrameRef parent = new Frame;
FrameRef child = new Frame;
```

```
child->setParent(parent);

parent->setTranslate( Vec3( 1, 0, 0) );

child->getTranslate(); // Now returns (1,0,0) due to its parent

parent->setRotate( EulerAngles(0, M_PI/2, 0) ); // Rotate 90 deg around Y.

Vec3 localVelocity( 1, 0, 0 ); // Local velocity, 1 in X.
Vec3 worldVelocity = child->transformVectorToWorld( localVelocity );
// World velocity is now (0,0,-1) as it is rotated around Y 90 deg.

child->getTranslate(); // Now returns (0,0,-1) due to its parents transformation.
```

12.6 Assembly

An agxSDK::Assembly is a class for logically (and coordinate wise) grouping objects together. It can store a set of:

- agx::RigidBody
- agxCollide::Geometry
- agx::Constraint
- agxSDK::StepEventListener
- agxSDK::Assembly
- agx::Material
- agx::ContactMaterial

This can be useful when building larger constructions where a logical grouping is needed. An Assembly also contains a Frame which holds a transformation.

Each RigidBody, Geometry and Constraint will derive the transformations specified in the Assembly's Frame. So if an Assembly is moved (**Assembly::setPosition()**) all contained objects will also be moved accordingly.

Schematic example of a function creating a car, returning a pointer to an assembly containing the whole car:

```
agxSDK::Assembly* buildCar()
{
  agxSDK::Assembly* parent = new agxSDK::Assembly();
  agx::RigidBodyRef chassis = new agx::RigidBody();
  // ...
  parent->add(chassis);
  parent->add(new agx::Material("rubber"));
  // ...
  return parent;
}
```

12.6.1 Adding/Removing an Assembly

An assembly can contain various objects such as RigidBody, Geometry, StepEventListener, GuiEventListener, ContactEventListener and Constraints.

When an Assembly is added to the simulation, all its contained items will also be added to the simulation. The same rule is applied when removing an Assembly from a simulation. This can however be controlled with the second argument to the

Simulation::remove method:

```
bool Simulation::remove(agxSDK::Assembly *assembly, bool removeAllEntries = true );
```

12.6.2 Derived transformation

When a RigidBody is added to an Assembly, the RigidBody derives the Assembly's transformation/velocity through a Frame/Parent interface. This means that a transformation made on the Assembly, will affect all rigid bodies contained in the Assembly. For example:

```
agx::RigidBodyRef body = new agx::RigidBody();
body->getFrame()->setLocalTranslate( agx::Vec3(1,0,0) ); // Set the transformation of the
rigidbody

agxSDK::AssemblyRef assembly = new agxSDK::Assembly();
assembly->add( body );

// Move the assembly up in z 1 unit.
assembly->setPosition( agx::Vec3(0,0,1) );

// The line below will be true as body will inherit the assembly's transformation
assert(body->getPosition() == agx::Vec3(1,0,1));
```

12.7 Collection

The class agxSDK::Collection is derived from Assembly. It has the same functionality except for the transformation. A Collection does not contain any transformation and it will not take ownership of added objects Frame. Changing the transformation of a Collection has no effect on its children.

The Collection class is useful for collecting objects when reading from files, building entities where it is important to keep previous frame/transformation hierarchies.

13 Creating objects

13.1 RigidBody

An **agx::RigidBody** is an entity in 3D space with physical properties such as mass and inertia tensor and dynamic properties such as position, orientation and velocity. A rigid body is created as follows:

```
agx::RigidBodyRef body = new agx::RigidBody();
simulation->add( body ); // Add body to the simulation
```

There are three motion control schemes for a RigidBody:

DYNAMICS	Affected by forces, position/velocity is integrated by the *DynamicsSystem*.
STATIC	Not affected by forces, considered to have infinite mass, will not be affected by the integrator.
KINEMATICS	Controlled by the user. By supplying a target transformation and a delta time, a linear and an angular velocity are calculated. This velocity will be used by DynamicsSystem to interpolate the position/orientation of the kinematic RigidBody. The angular velocity is considered to be constant during the interpolation.

Table 8: Motion control modes for a rigid body.

The type of motion control is set with the method:

```
RigidBody::setMotionControl(RigidBody::MotionControl);
```

13.1.1 Velocities

The angular and linear velocity of a rigid body is always understood as *the velocity of the center of mass*. Velocities of a rigid body are expressed in the world coordinate system.

```
agx::Vec3 worldVelocity(0,1,0);

// Specify the linear velocity in world coordinates for a rigidbody
body->setVelocity( worldVelocity );

// Specify angular velocity for a body in LOCAL coordinate system
agx::Vec3 localAngular(0,0,1);

// Convert from local to world coordinate system
agx::Vec3 worldAngular = body->getFrame()->transformVectorToWorld( localAngular);

// Set the angular velocity in World coordinates
body->setAngularVelocity( worldAngular );
```

13.2 Effective Mass

When for example simulating a ship in water, the effective mass can be substantially larger than the actual weight of the ship. This is due to the water that is added by hydrodynamic effects. The mass varies depending on depth of water and is also different

in different directions. The class **agx::MassProperties** has the capability of handling this added mass through the methods:

```
void setMassCoefficients( const Vec3& coefficients );
void setInertiaTensorCoefficients(const Vec3& coefficients);
```

This also means that AGX can handle objects that have different masses along each axis, so called *non-symmetric mass matrices.*

13.3 Modeling coordinates/Center of Mass coordinates

Center of mass is the point in the world coordinate system where the collected mass of a rigid body is located in space. This point can either automatically be calculated from the geometries/shapes that are associated to the rigid body, or it can also be explicitly specified to any point in space.

Since the center of mass moves when geometries are added *(if RigidBody::getMassProperties()::getAutoGenerate()==true)*, it is not a practical reference when building a simulation. For example, if a constraint is attached to a body relative to the body center of mass, adding another geometry and thereby moving the center of mass would change to attachment point for the constraint. This is usually not the expected/wanted behavior.

Therefore AGX provides two coordinate frames, the *model frame* and *the center of mass(CM) frame*.

Figure 11 shows one box with the model origin (axes) and the center of mass (yellow sphere) at the same positions. Figure 12 illustrates the situation after another box geometry is added to the body, and thus the center of mass (yellow sphere) is shifted along the y-axis. The model origin (axes) is still at the same position.

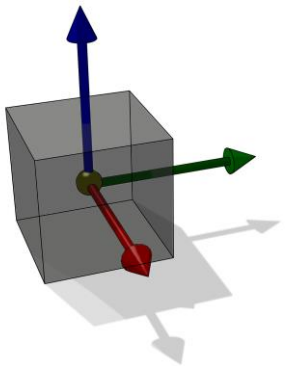


Figure 11: Model and Center of mass at the same position.

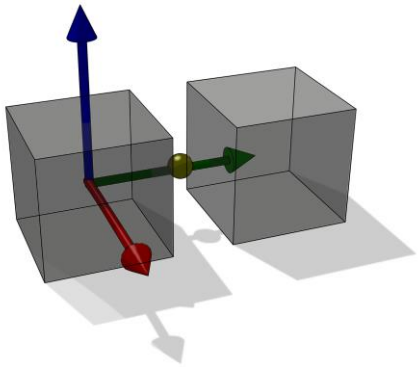


Figure 12: Center of mass translated due to added geometry.

Note that AGX only supports relative translation of the center-of-mass-frame with regard to the model frame, but not relative rotation.

The translation, rotation and velocity of the center of mass for a rigid body can be accessed via the methods:

```
agx::RigidBodyRef body = new agx::RigidBody;

// Get position of Center of Mass in world coordinate system
```

```
Vec3 pos = body->getCmPosition();

// Get orientation of Center of Mass in world coordinate system
Quat rot = body->getCmRotation();

// Get rotation and translation of Center of Mass in world
// coordinate system
AffineMatrix4x4 m = body->getCmTransform();

// Or the complete frame at one call
Frame *cmFrame = body->getCmFrame();

// Translation relative to model coordinate system
Vec3 pos = body->getCmFrame()->getLocalTranslate();

// In most situation, it is the model coordinate system that is interesting. This can be
accessed using the calls:
// Get position of Rigid Body in world coordinate system
Vec3 pos = body->getPosition();

// Get orientation of Rigid Body in world coordinate system
Quat rot = body->getRotation();

// Get rotation and translation of Rigid Body in world coordinate system
AffineMatrix4x4 m = body->getTransform();

// Or the complete frame at one call
Frame *cmFrame = body->getFrame();

// Get linear velocity of body
Vec3 v = body->getVelocity()

/**
\return the local offset of the center of mass position to the model origin (in model frame
coordinates).
*/
agx::Vec3 getCmLocalTranslate() const;

/**
Sets the local offset of the center of mass position to the model origin (in model frame
coordinates).
\param translate The new offset.
*/
void setCmLocalTranslate(const agx::Vec3& translate);
```

13.4 Kinematic rigid body

By specifying that a rigid body should be *kinematic*, we are telling the DynamicsSystem to stop simulating the dynamics of the body. This means that it will get an infinite mass, will not be affected by forces etc.

```
rigidbody->setMotionControl( agx::RigidBody::KINEMATICS );
```

It is also possible to interpolate the position/orientation based on a target transformation specified by the user. The body can then be specified to move from the current transform to a new during a specified time step with the call to:

```
agx::RigidBody::moveTo(AffineMatrix4x4& transform, agx::Real dt);
```

The above method will make a body move in a velocity calculated from its current transformation frame, to the specified during a time interval of *dt*.

When the interval *dt* is reached (current time + dt) the body will continue to move according to its velocity until next call to moveTo().

If you explicitly set a linear/angular velocity on a body, and specify it to be KINEMATICS, then that velocity will be used for integrating the transformation (position and orientation).

13.5 Geometry

A *Geometry* can be assigned to a rigid body to give it a geometrical representation. The geometry is also the entity that can intersect other geometries and create contacts.

A Geometry contains one or more shapes. A *Shape* is specialized by *Box*, *Sphere*, *Capsule* etc. see Table 9 for a complete list of available shapes.

13.5.1 Enable/disable

A Geometry can be enabled/disabled. A disabled geometry will not be able to collide with any other geometries. This is true for all the shapes in the geometry. *A disabled geometry will not contribute to the mass properties of a rigid body.*

```
void Geometry::setEnable( bool enable );
bool Geometry::getEnable() const;
bool Geometry::isEnabled() const;
```

Calling geometry->setEnable(false) will effectively remove the geometry from the mass calculation and collision detection.

13.5.2 Sensors

A geometry can be specified to be a *sensor*. This means that the collision system will generate contact information, such as contact points, normals and penetration, but no *physical* contact (contact constraint) will be created, hence the solver will not see this contact. A sensor can therefore penetrate any other geometry as a "ghost". This can be useful for realizing certain functionality such as proxy tests, vision etc. A geometry is specified to be a sensor through the call:

```
agxCollide::GeometryRef geometry = new agxCollide::Geometry;
geometry->setSensor( true ); // Tell the geometry to become a Sensor
```

A sensor geometry will not add any physical properties such as mass, center of mass, inertia tensor to the body it is part of. Calling **setSensor(true)** will effectively remove the geometry from the mass calculation.

13.6 Enabling/disabling Contacts

There are several different alternatives to control whether pairwise contacts should occur or not.

- Disable all collisions for a geometry: Geometry::setEnableCollisions(false)
- Enable/disable pair of geometries
- Geometry groups
- User supplied contact listeners

It should be noted that there is not direct priority between these different possibilities, but rather, if any of them disables the contact (even if others enable it), the contact generation will be treated as disabled.

13.6.1 Disable collisions for a geometry

To explicitly disable all collision generation for a geometry:

```
// Disable all contact generation for the geometry:
geometry1->setEnableCollisions( false );
```

13.6.2 Enable/disable pair

To disable/enable contacts for a specific geometry pair one can call the method Geometry::setEnableCollisions(Geometry*, bool)

```
// Disable contact generation between geometry1 and geometry2
geometry1->setEnableCollisions( geometry2, false );

// Enable contact generation between geometry1 and geometry2
geometry1->setEnableCollisions( geometry2, true );
```

No matter if any of the two geometries are removed and later added to simulations, the information about setEnableCollisions() will still be available.

13.6.3 GroupID

The class Geometry has a *groupID* attribute. This attribute can be set and read using the following methods:

```
      void agxCollide::Geometry::addGroup( unsigned );
      bool agxCollide::Geometry::hasGroup( unsigned ) const;
      bool agxCollide::Geometry::removeGroup( unsigned );
```

With the method addGroup() one can assign a geometry to a specified group of geometries.

Together with the following method in agxCollide::Space:

```
void agxCollide::Space::setEnablePair(unsigned id1,
                                      unsigned id2,
                                      bool enable)
```

This functionality can be used to disable collisions between groups of geometries. To disable collisions between geometries of the same group:

```
space->setEnablePair( id1, id1, false );
```

After the above call, geometries belonging to *id1* cannot collide with each other. It is worth to notice that the group ids are stored in a vector, so searching and comparing has a time complexity of O(n). So before adding a large number of group ids to a geometry, make sure there isn't another alternative. This is because the geometry groups will be compared to each other during the collision detection process.

13.6.4 Named collision groups

New in 2.3.0.0 is that there is also *named collision groups.* This has the same functionality as 13.6.3 GroupID, but it is defined using a string instead.

```
      void agxCollide::Geometry::addGroup( const agx::Name& );
      bool agxCollide::Geometry::hasGroup(const agx::Name& ) const;
      bool agxCollide::Geometry::removeGroup(const agx::Name& );
```

This functionality can be used to disable collisions between groups of geometries. To disable collisions between geometries of the same group:

```
space->setEnablePair( name1, name1, false );
```

After the above call, geometries belonging to *name1* cannot collide with each other.

13.6.5 User supplied contact listener

A *ContactEventListener* (see 12.4.3) can be registered to listen to specific collision events, between specified geometries, geometries with specific properties etc. By implementing a collision event listener that for a given pair of geometries return REMOVE_CONTACT or REMOVE_CONTACT immediately in the impact() method, one can achieve the same result as the previous two alternatives, namely, no contact is transferred to the contact solver. This is a very general way of specifying for which pairs contacts should be generated.

```
class MyContactListener : public agxSDK::ContactEventListener
{
public:
  MyContactListener()
  {
    // Only activate upon impact event
    setMask(agxSDK::ContactEventListener::IMPACT);
  }

  bool impact(const agx::TimeStamp& t, agxCollide::GeometryContact *cd) {

    RigidBody* rb1 = cd->geomA->getBody();
    RigidBody* rb2 = cd->geomB->getBody();

    // If both geometries have a body, and the mass of the first is over 10
    // then remove the contact
    if (rb1 && rb2 && rb1->getMassProperties()->getMass() > 10 )
      return REMOVE_CONTACT; // remove contact.

    return KEEP_CONTACT; // keep the contact
  }
};
```

Which geometries should trigger the ContactEventListener can be specified using an **ExecuteFilter**. This filter can also be user specified. By deriving from the class **agxCollide::ExecuteFilter** and implementing two methods, any user supplied functionality can be implemented for filtering out contacts:

```
class MyFilter : public agxCollide::ExecuteFilter
{
public:

  MyFilter() {}

  /**
  Called when narrow phase tests have determined overlap (when we have
  detailed intersection between the actual geometries).
  Return true if GeometryContact matches this filter:

  if both geometries have a body.
  */
  virtual  bool operator==(const agxCollide::GeometryContact& cd) const
  {
    bool f =
      (cd.geomA->getBody() && cd.geomB->getBody());
    return f;
  }

  /**
  Called when broad phase tests have determined overlap between the two
```

```
  bounding volumes.
  */
  virtual  bool operator==(const GeometryPair& geometryPair) const
  {
    bool f =
      (cd.geomA->getBody() && cd.geomB->getBody());
    return f;
  }
};
```

13.6.6 Surface velocity

A geometry can be given a surface velocity. It is specified in the local coordinate system of the geometry and can be used for simulating the surface of a conveyor belt. In its general form, it is independent of the contact point position relative to the geometry. If more control over that behavior is desired, it is possible to create a class inheriting from agxCollide::Geometry, and to override the method `agx::Vec3f Geometry::calculateSurfaceVelocity( const agxCollide::LocalContactPoint& point ) const.`
One such class is agx::SurfaceVelocityConveyorBelt, which is described in chapter 23.

13.7 Primitives for collision detection (Shapes)

Table 6 lists the currently available primitive tests in AGX. A cell marked with a × indicates that a collider plug-in is available that can calculate contact points, normals and penetration between the matching primitives. This matrix will continuously be updated as new tests are implemented.

13.7.1 Box

A box is defined by three half extents with its center of mass at origin.

```
Box::Box( const agx::Vec3& halfExtents )
```

13.7.2 Sphere

A sphere is defined by its radius and is created with its center of mass at origin.

```
Sphere::Sphere( const agx::Real& radius )
```

13.7.3 Capsule

A capsule is a cylinder, capped with two spheres at the end. It is defined by its height and its radius. The capsule will be created with its height along the y-axis and its center of mass at origin.

```
Capsule::Capsule( agx::Real radius, agx::Real height )
```

13.7.4 Cylinder

A cylinder is defined by its height and radius. The cylinder will be created with its height along the y-axis and its center of mass at origin.

```
Cylinder::Cylinder( const agx::Real& radius, const agx::Real& height )
```

13.7.5 Line

A line is defined by two points, begin and end points:

```
Line::Line( const agx::Vec3& start, const agx::Vec3& end )
```

13.7.6 Plane

A plane can be defined in a number of different ways: either by a point and a normal, or the four plane coefficients:

```
Plane::Plane( agx::Real const a, agx::Real const b,
              agx::Real const c, agx::Real d )
Plane::Plane( const agx::Vec3& normal , agx::Real d )
Plane::Plane( const agx::Vec3& normal, const agx::Vec3& point )
Plane::Plane( const agx::Vec4& plane)
Plane::Plane( const agx::Vec3& p1, const agx::Vec3& p2,
              const agx::Vec3& p3)
```

13.7.7 Triangle mesh

13.7.7.1 Triangle mesh construction (general case)

The triangle mesh shape in AGX supports general concave triangle meshes. The internal storage of triangle meshes in AGX is based on *vertices* (array of 3D points) *and indices* (indexes into the array of vertices), where three concurrent indices define a triangle each (a *triangle list*). Trimeshes share a common base class called **agxCollide::Mesh** with another mesh-based collision shape called **agxCollide::HeightField**.

An example of how a trimesh can be created from code:

```
agx::Vec3Vector vertices;
agx::UIntVector indices;

// Add vertices
vertices.push_back( agx::Vec3( -0.4, -0.4, 0.0 ) );
// ...

// Add indices
indices.push_back( 1 ); // starting first triangle
indices.push_back( 0 );
indices.push_back( 2 ); // finished first triangle
indices.push_back( 0 ); // starting second triangle, reusing one vertex
// ...

// Create the collision shape based on the vertices and indices
agxCollide::TrimeshRef triangleMesh = new agxCollide::Trimesh( &vertices,
&indices, "handmade pyramid" );
```

And from a file:

```

// Create the trimesh shape from a WaveFront OBJ file:
agxCollide::TrimeshRef trimeshShape =
      agxUtil::TrimeshReaderWriter::createTrimeshFromWavefrontOBJ(
                                                  "torusRoundForCollision.obj" );
```

The data will be copied by the trimesh constructor.

Furthermore, the triangles are expected to be given in counterclockwise order (*counterclockwise winding/orientation*). This is important since the order of the vertices defines which face of the triangle will point outwards and which inwards. This orientation is used within calculation of mass properties and collision detection. If the data is given in clockwise winding, this can be indicated by setting the according flag in **agxCollide::Trimesh::TrimeshOptionsFlags**. Mixed winding should be avoided since it will lead to unexpected behavior. The flag

agxCollide::Trimesh::RECALCULATE_NORMALS_GIVEN_FIRST_TRIANGLE can be used to adapt the winding to the first triangle, given that the mesh is watertight otherwise.

Internally, the triangle data will be stored in counterclockwise winding.

For good collision detection results, each trimesh should be a watertight (a geometrical manifold). Non-manifold trimeshes can be used, but the results may vary in quality. Non-manifold trimeshes may furthermore result in an invalid mass and inertia when used in dynamic rigid bodies for which mass properties are calculated automatically.

13.7.7.2 Triangle meshes as terrain

Trimeshes can also be used to model terrains. Here, they do not have to be closed on all sides, but may have an open direction. This direction is assumed to be upward in z direction in the trimesh's local coordinate system. The corresponding normal (0, 0, 1) is used as a tunneling safeguard, and the trimesh's bounding box is enlarged in opposite direction of this normal by a value called the terrain's "bottom margin".

It is recommended to avoid collisions at terrain borders unless they are met by a corresponding neighboring terrain, since this might give suboptimal results. The trimesh class will give out warnings during construction if a terrain does not have a rectangular shape (in the trimesh's local x and y coordinates), or contain holes. The usage as terrain should be indicated in the optionsMask.

One example for a terrain with a bottom margin of 5.0 below the mesh's deepest point in Z:

```
agx::Vec3Vector vertices;
agx::UIntVector indices;

// Add vertices
vertices.push_back( agx::Vec3( -0.4, -0.4, 0.0 ) );
…
// Add indices
indices.push_back( 1 ); // starting first triangle

uint32_t optionsMask = agxCollide::Trimesh::TERRAIN;
…
TrimeshRef terrain = new Trimesh( &vertices, &indices, "handmade terrain", optionsMask, 5.0 );
```

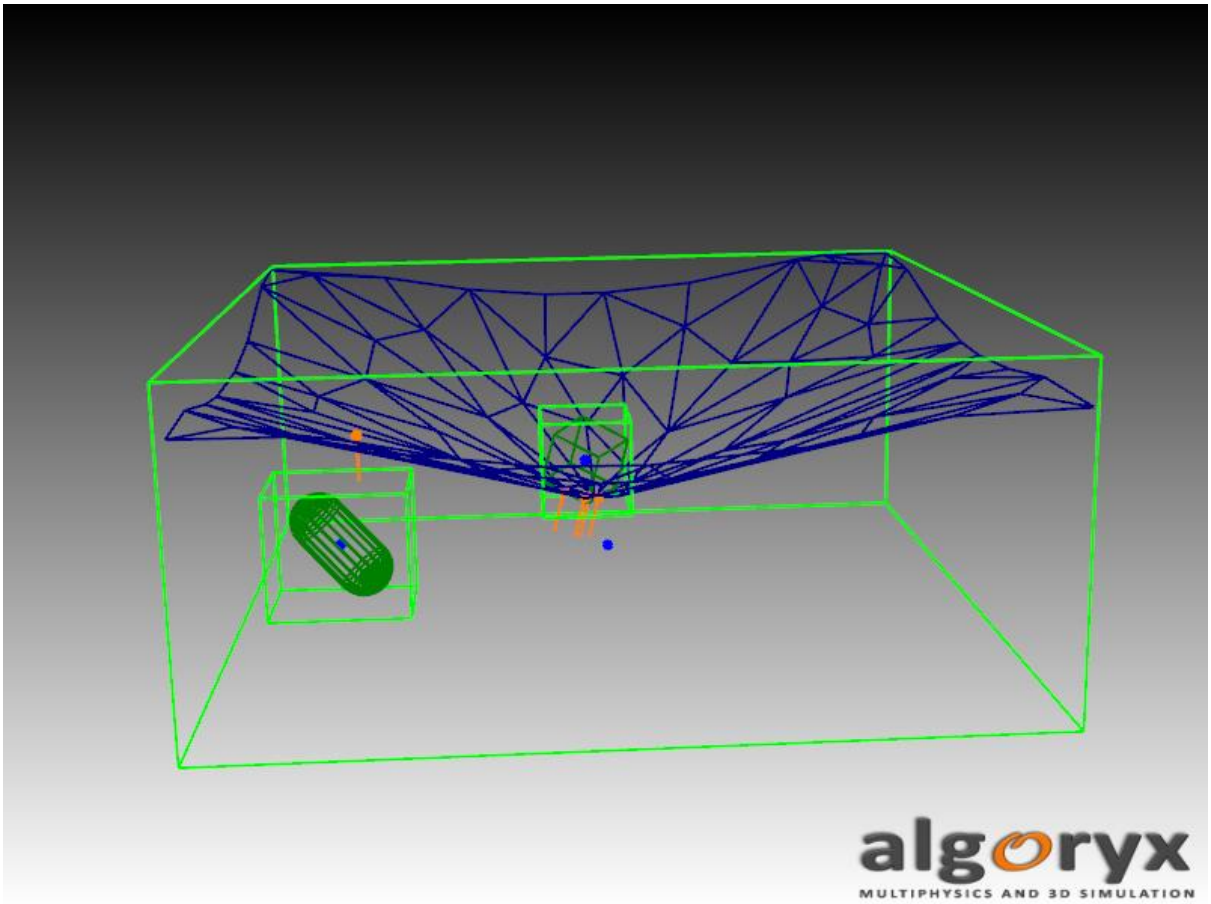


Figure 13: Terrain/trimesh intersecting a box, and a capsule which is prevented from tunneling. Note the enlarged bounding box in opposite direction of the terrain normal.

Using a height-field for collision detection as a terrain has the advantage of being able to modify the terrain during run-time. Trimeshes as terrains, on the other hand, can be modeled with fewer triangles by adapting the level of detail to the local change in terrain height, so that large plain parts can be covered by few large triangles.

13.7.7.3 Triangle meshes modeling

As mentioned above, special attention should be paid when using triangle meshes as collision primitives.

It is important that the models are adapted for the special needs arising from physics simulation, and thus are either manifolds or used as terrain.

Potential problems (which should be fixed externally before using the trimesh) are:

- Unmerged identical vertices
- Holes
- Wrong/inconsistent winding
- Triangles with zero area
- Unreferenced vertices
- Self-intersection.

Trimeshes used in collision detection create potentially very many contact points. The use of *contact reduction* is therefore recommended.

13.7.7.4 Triangle mesh traversal

If a trimesh is modeled in the above recommended way and thus has a half edge structure, it can be traversed via this neighborhood graph: each triangle consists of three half edges, where each half edge has a neighboring half edge in a bordering triangle, and where one half edge's starting vertex is the other one's ending vertex.

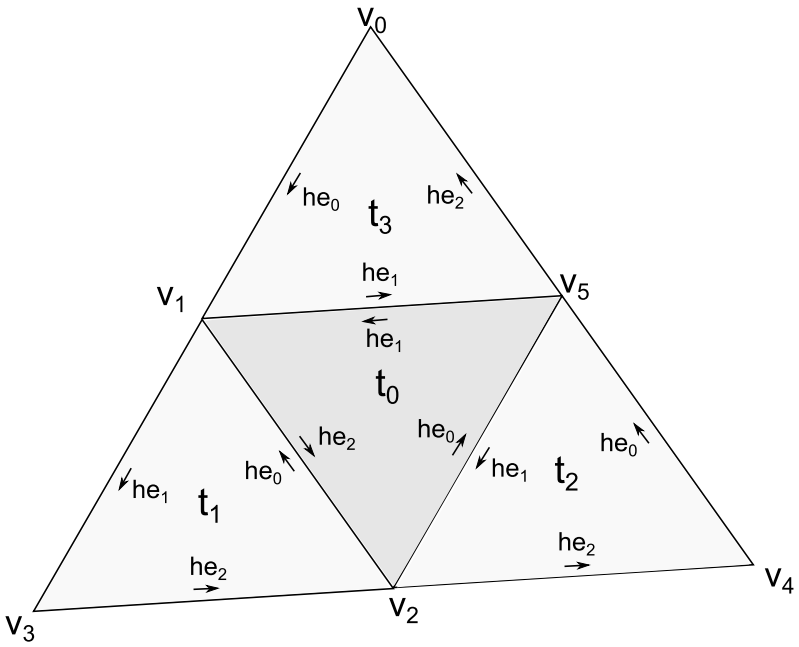


Figure 14: A trimesh consisting of 4 triangles and 6 vertices. The half edge structure is shown.

This data is stored as arrays of vertices, triangle half edges and half edge neighbors in the Trimesh class and can be used directly.

Some helper classes make traversal of triangles easier, though. The neighborhood structure can be traversed by using the helper class **agxCollide::Mesh::Triangle**.

```
// Create the trimesh shape from a WaveFront OBJ file:
agxCollide::TrimeshRef trimesh =
      agxUtil::TrimeshReaderWriter::createTrimeshFromWavefrontOBJ( "pin.obj" );
// Get first triangle in trimesh
agxCollide::Mesh::Triangle triangle = trimesh->getTriangle(0);
if (triangle.isValid()) // Important to check validity before use
      triangle = triangle.getHalfEdgePartner(0); // Get neighbor to first edge
```

Circling around a vertex can be done more comfortably using the helper class **agxCollide::Mesh::VertexEdgeCirculator**.

```
// Create the trimesh shape from a WaveFront OBJ file:
agxCollide::TrimeshRef trimesh =
      agxUtil::TrimeshReaderWriter::createTrimeshFromWavefrontOBJ( "pin.obj" );
// Get a circulator around the first vertex in the first triangle.
agxCollide::Mesh::VertexEdgeCirculator circulator =
      trimesh->createVertexEdgeCirculator( 0, 0 );
// The circulator might be invalid if created from invalid parameters.
if (circulator.isValid()) {
  // Has the circulator done a whole turn around the vertex?
  while (!circulator.atEnd()) {
    // Set it to next position. The VertexEdgeIterator will even manage
    //holes in the neighborhood structue by jumpoing over all holes.
    circulator++;
    if (circulator.hasJustJumpedOverHole())//Have holes been passed?
      std::cout << "Vertex circulator jumped over hole\n";
```

```
  }
}
```

More elaborate examples can be found in tutorial_trimesh.cpp.

13.7.8 Convex

Convex is a geometry that describes a convex shape built up from a point cloud. The properties of a convex is among the following: none of the surface normals of a convex will ever cross each other, any point on a 2D projection of a convex will only be covered twice, once from the surface facing away the projection plane and once from the surface facing the projection plane.

Convex shapes are created like from an array of vertices and indices, like Trimeshes.

```
agx::Vec3Vector vertices;
agx::UIntVector indices;
// ... add single vertices and indices to the containers...
ConvexRef convex = new Convex(&vertices, &indices);
```

13.7.8.1 Convex decomposition

The limitation of convex in that it can only describe a small subset of geometries (convex) can be overcome by taking a general (concave) triangle mesh and decompose it into convex meshes. The result will not be an exact representation of the original triangle mesh, but in many cases it will be good enough for use in simulations. AGX supports *convex decomposition* through the class **agxCollide::ConvexBuilder**.

```
#include <agxCollide/ConvexBuilder.h>

// Create a ConvexBuilder
agxCollide::ConvexBuilderRef builder = new agxCollide::ConvexBuilder;

// Createa mesh reader for Wavefront OBJ format.
agxIO::MeshReaderRef reader = new agxIO::MeshReaderOBJ;
if (!reader->readFile( "mesh.obj"))
  error();

// Try to parse the data read from the reader and create convex shapes
size_t numConvexShapes = builder->build( reader );
if (numConvexShapes)
  error();

// Get the vector of the created shapes
agxCollide::ConvexBuilder::ConvexShapeVector& shapes =
      builder->getConvexShapes();
// Now insert shapes into one or more geometries
```

13.7.9 WireShape

The wire shape is used internally for the collision handling of wire segments. It is based on the capsule, but might in the future have support for continuous collision detection. It is not advised to use this shape for other purposes, since implementation details might change.

13.7.10 HeightField

A height field is a geometric representation that can be used for terrain, sea beds etc. Height-fields are currently implemented as a scaled uniform grid centered at the local origin in x and y, and with z up. Furthermore, height-fields are treated like triangle meshes, where each rectangular grid cell is divided into a lower left and an upper right triangle. The corresponding class **agxCollide::HeightField** shares a common base class called **agxCollide::Mesh** with another mesh-based collision shape called **agxCollide::Trimesh**.

Compared to a trimesh, a height-field for collision detection as a terrain has the advantage of being able to modify the terrain under run-time. Trimeshes as terrains, on the other hand, can be modeled with fewer triangles by adapting the level of detail to the local change in terrain height, so that large plain parts can be covered by few large triangles.

Since trimeshes and height-fields share many routines for mesh traversal, please look into the trimesh chapter (13.7.7) for more information about that.

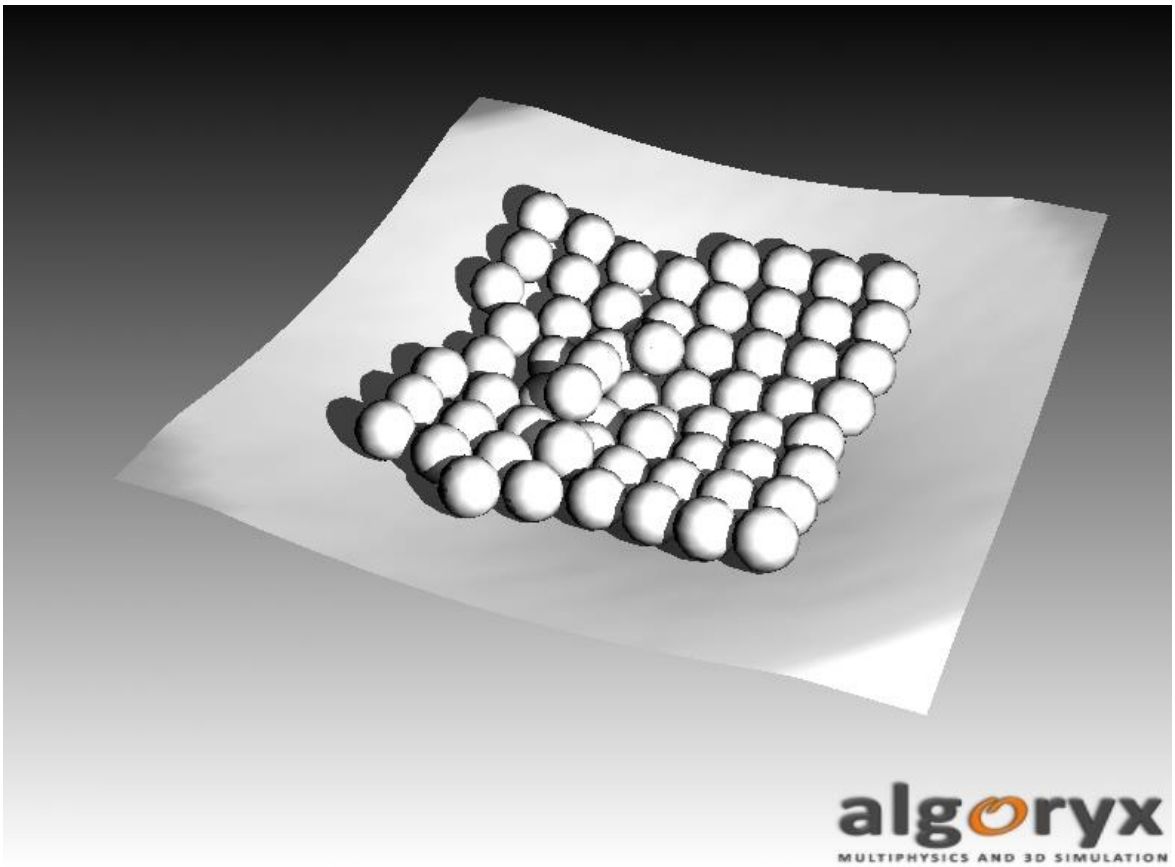


Figure 15: HeightField representing uneven surface.

Like trimeshes used as terrains, height-fields make use of a bottom margin in order to have a safeguard in case objects tunnel downwards in local z-direction through the triangle surface.

13.7.10.1 HeightField creation

The height values in a height field can either be specified through a call to **agxCollide::HeightField::setHeights()** which accepts a vector in row matrix order or via a **agxUtil::HeightFieldGenerator()**, which can read data from an image (.png) or a raw binary file (no endian handling supported).

```
HeightField(size_t resolutionX, size_t resolutionY, agx::Real sizeX, agx::Real sizeY,
agx::Real bottomMargin=agx::Real(1));
```

13.7.10.1 Modifying parts of a HeightField

The height values in a height field can also be changed in only a part of the whole shape. One possibility is by calling a version of **agxCollide::HeightField::setHeights()** which specifies which rectangular part of the grid to modify.

```
    bool setHeights(const agx::RealVector& heights,
      size_t minVertexX, size_t maxVertexX, size_t minVertexY, size_t maxVertexY);
```

Another possibility is to set heights at specific points in the grid using the function **agxCollide::HeightField::setHeight()**.

```
void setHeight(size_t xIndex, size_t yIndex, agx::Real height);
```

Note that all triangles around a changed height have to be updated. This update includes a recalculation of the triangle normal, as well as an update of the bounding volume hierarchy. Changing a single height will involve 6 updated triangles. Changing several heights within a rectangle can be done faster by using **agxCollide::HeightField::setHeights()** rather than **agxCollide::HeightField::setHeight()**, since this makes it unnecessary to update triangles several times.

13.7.10.2 HeightFields for dynamic simulation

There can be cases where it is of interest to combine the ease of modification of a height field with dynamical simulation, by making the height field part of the collision detection representation of an **agx::RigidBody** which is simulated dynamically. Since height-fields are generally assumed to be part of the static scenery, their mass properties are not calculated in the general case, since this comes with some computational cost. In order to get the mass-properties updated for dynamic simulation, call the method **agxCollide::HeightField::setDynamic()**.
In order to compute these, a sort of box-shape is assumed for the height-field, where the top has a relief defined by the heights, the sides go downwards at the borders of the grid, and the bottom is a rectangle parallel to the z plane at a position given by a specified minimum height. To specify this height, use **agxCollide::HeightField::setMinAllowedHeight()**.

Note that when changing parts of the height-field, the mass properties will not be updated. Instead, once all the updates are done and the new mass properties are needed for dynamic simulation, one should call **agxCollide::HeightField::calculateMassProperties()** which will recalculate the mass properties for the whole height-field and can potentially be costly. This might be optimized in a future release.

13.7.11 Composite Shapes

Composite geometries are geometries containing more than one Shape. This can be useful when building more complex geometries from primitives.

There are basically two ways of creating a more complex Geometry structure around a rigid body.

1. Create a geometry *g1*, add shapes with relative transformations that builds up the final geometry. This also means that *g1* can have only one material. (Material is bound per Geometry).
   Then, create a body *b1* and add the created composite geometry *g1*.
2. Create a body *b1*, create the geometries needed *g1, g2, …* and for each geometry add shapes (Box, Sphere, …). Then, add each geometry to the body.
   In this case each geometry *g1, g2…* can each have different materials.

Note that there are two limitations when using composite shapes:

- If the single shapes forming the composite shape do not overlap, interacting objects might get in between them exactly where they touch. This might happen since the single shapes which the composite shape consists of do not know about each other. Therefore it is recommendable to let shapes overlap.
- However, when having overlapping shapes, the automatic mass property computation will count the overlapping volume twice. So here, manual mass property computation has to be used.

13.8 Shape colliders

A shape collider is a class that is responsible for calculating contact data between two shapes. The table below shows the available shapes and for which pair of shapes the AGX collision engine currently can calculate contact data.

13.8.1 Collider Matrix

In the following table, an overview over the functionality of colliders for different shape combinations is given. Reduced functionality means that basic functionality is given, but might have to be improved in the future in order to work perfectly in all purposes. Plane and HeightField make only sense as static geometries and can therefore not collide with themselves/each other.

	Box	Capsule	Convex	Cylinder	Line	Plane	Sphere	Trimesh	Composite	HeightField
Box	X	X	X	X	X	X	X	X	X	X
Capsule	X	X	X	X	X	X	X	X	X	X
Convex	X	X	X	X	X	X	X	X	X	X
Cylinder	X	X	X	X	X	X	X	X	X	X
Line	X	X	X	X	X	X	X	X	X	X
Plane	X	X	X	X	X	-	X	X	X	-
Sphere	X	X	X	X	X	X	X	X	X	X
Trimesh	X	X	X	X	X	X	X	X	X	X
Composite	X	X	X	X	X	X	X	X	X	X
HeightField	X	X	X	X	X	-	X	X	X	-

Table 9: Available primitive tests. ✕ Full functionality, (✕) Reduced functionality, — Not needed (since both are supposed to be static).

13.8.2 Point, normal and depth calculation

In case of an intersection between two shapes, a shape collider will report one or more contact points. Each contact point consists of

- A point *p*,
- a normal *n*,
- a depth *d*.

The normal *n* points in the direction which lets shape1 move out of shape2.

The depth *d* measures how much shape 1 would have to be moved out of shape2 along normal *n* in order to resolve the overlap and thus have the two shapes just touching.

The point *p* is placed halfway between the two points corresponding to the surfaces of the two shapes, so that $p_{\{shape_1\}} = p - n * d * 0.5$ and$p_{\{shape_2\}} = p + n * d * 0.5$.

13.8.3 Results for contacts with Line Shapes

The Line Shape is special in that its use is not in rigid body simulation, but in other fields as e.g. mouse picking or depth computation for unknown geometries. Therefore, the computation of contact point, normal and depth differs from that of the other Shapes.

The Line Shape represents mathematically a line segment, defined by two points P_0 and P_1, which define the non-normalized Line direction $d = P_1 - P_0$.

The Line can thus be presented as: $P_0 + t\,d, 0 \leq t \leq 1$.

If a Line intersects a Shape, the first point in contact P_c along the infinite version of the line is reported as contact point. The Shape surface normal is reported as contact normal. The parameter t_c is the value which realizes $P_c = P_0 + t_c d$.

The depth is $(1 - t_c) * |P_0 - P_1|$; this corresponds to how long the Line has to be pushed back in its direction in order to just touch the overlap point P_c.

Exceptions:

- For mesh-based shapes (Trimesh, HeightField and at the moment also Convex), lines which are completely inside the shape will not create any contact.
- The line-line collider will create one contact point with a normal which is orthogonal to both lines and a depth of 0.

13.9 Storing renderable data in shapes: agxCollide::RenderData

agxCollide::RenderData is a class which can be used for storing renderable data together with a shape.

```
// Create a RenderData object
agxCollide::RenderDataRef renderData = new agxCollide::RenderData;

// Specify the vertices
agx::Vec3Vector vertices;
renderData->setVertexArray( vertices );

// Specify the normals
agx::Vec3Vector normals;
renderData->setNormalArray( normals );

// Specify the indices, which constructs Triangles
agx::UInt32Vector indices;
renderData->setIndexArray( indices, agxCollide::RenderData::TRIANGLES);

// Store a diffuse color
renderData->setDiffuseColor( agx::Vec4(1,0,1,1) );
```

This data can then be associated with an agxCollide::Shape:

```
agxCollide::ShapeRef shape = new agxCollide::Sphere(1.0);
shape->setRenderData( renderData );
```

During serialization to disk, this data will also be stored together with a shape. Later it can be retrieved with the call:

```
agxCollide::RenderData *renderData = shape->getRenderData();
```

In the sample coupling to OpenSceneGraph, agxOSG namespace, the function agxOSG::createVisual(geometry, agxOSG::Group *) will parse this render data and create a visual representation based on the RenderData (if available), otherwise it will use the specified shape when the renderable data is created.

13.10 Mass Properties

agx::MassProperty is a class which stores and handles the mass properties of a rigid body namely the heavy and effective mass of a rigid body. Whenever Geometries are added/removed to/from a rigid body with the methods

```
RigidBody::add(Geometry*)
RigidBody::remove(Geometry*),
```

the inertia tensor is recalculated by default, unless a previous call to *RigidBody::getMassProperties()::setAutoGenerate(false)* has been done. In this case the inertia tensor and masses will NOT be updated when geometries are added and removed.

Currently, automatic mass property calculation (incremental or not) assumes all geometries and shapes to be disjoint (non-overlapping). If they overlap, the effect of this assumption is that the overlapping volume will be counted twice into mass and inertia computation. It is recommended to set the mass properties manually in this case.

An explicit inertia tensor can also be specified manually with the call:

```
body->getMassProperties()->setInertiaTensor(I);
```

where / can either be a SPDMatrix3x3 (semi positive definite) matrix or a Vec3 specifying the diagonal elements of the inertia tensor.

The method **MassProperties::setInertiaTensor** has a second parameter specifying whether the inertia tensor should be automatically recalculated, when set to false, the Inertia tensor for a rigid body will not be updated to reflect changes in associated Geometries.

During the calculation of mass properties, the density from the specified materials (described later) will be used to calculate the final mass. If no mass is specified explicitly for the body, the calculated mass will be used. If on the other hand the mass have been set by the user with the method **RigidBody::getMassProperties()::setMass()**, this mass will be used for the final result.

Unless the setAutoGenerate method has been called for a rigid body, the center of mass and inertia tensor will be updated for most operations such as adding/removing

geometries, removing shapes from geometries. However there are a number of operations that will NOT be detected, and hence the user has to tell the body to update its mass properties:

- Transforming geometries relative the body after they are added to the body.
- Changing material properties (density) for any geometry added to a body.
- Transforming shapes that is part of a geometry associated to a body.

In the above cases the mass properties can be updated with a call to:

```
agx::RigidBody::updateMassProperties(agx::MassProperties::AUTO_GENERATE_ALL);
```

If you only want to update the inertia tensor you can change the argument to updateMassProperties. For more information see the SDK documentation.

13.11 Adding user forces

The class *agx::RigidBody* has the following methods for adding forces and torques:

addForce(), addForceAtPosition(), addForceAtLocalPosition(), addTorque().

These added forces are cleared after a time step has been taken in the simulation. The force added to a rigid body during the last timestep (including the forces introduced by the user via addForce/addForceAtPosition/addTorque , or via force fields etc. – however, NOT the ones from contacts and constraints!) can be accessed with the methods: *RigidBody::getLastForce()/getLastTorque()*

13.12 ForceFields

The class **agx::ForceField** can be used to induce user forces on all or a selected part of the bodies in a DynamicsSystem. It is derived from agx::Interaction.

```
#include <agx/ForceField.h>
class MyForceField : public agx::ForceField
{
  public:
  MyForceField() {}
  ~MyForceField() {}

  // Implement this method to add forces/torques of all or selected bodies
  void updateForce( DynamicsSystem *system );
};

// Create the force field and add to the DynamicsSystem.
agx::InteractionRef forceField = new MyForceField;
simulation->getDynamicsSystem()->add( forceField );
```

For a more thorough example, see tutorial_bodies.cpp in <agx>/tutorials/agxOSG/tutorial_bodies.cpp

13.13 GravityField

Gravitational forces should be applied through an instance of an **agx::GravityField**. The reason for this is that certain mechanics inside for example the wire implementation (agxWire::Wire) needs to know the individual gravity for each body.

By default, an instance of a class **agx::UniformGravityField** is created in DynamicsSystem, with the default acceleration of G= 9.80665 m/s^2. Another gravitational model is **agx::PointGravityField**, which will add a force towards a specified origin point. This model can be used for simulating the gravity on the surface of the earth.

To register a gravity field with the simulation the following call is done:

```
agxSDK::SimulationRef sim = new agxSDK::Simulation;
sim->setGravityField( new agx::PointGravityField );
```

Simulation::setUniformGravity and Simulation::getUniformGravity are just wrapper methods to simplify the use of the standard uniform gravity. A call to Simulation::setUniformGravity() will return false if the current gravity field model is not a agx::UniformGravityField.

13.13.1 CustomGravityField

The standard gravity fields (UniformGravityField and PointGravityField) are implemented as computational kernels to achieve best possible performance. However, to implement a custom gravity field, it is possible to derive from the class **agx::CustomGravityField** and implement the virtual method **calculateGravity**. The overhead compared to the standard gravity fields is the call to virtual method which should be small.

It is however important to remember that the method calculateGravity() will be called simultaneously from several threads if AGX is using more than one thread. This means that the method must be made thread safe.

```
class MyGravityField : public agx::CustomGravityField
{
public:
  MyGravityField() {}
  /// overidden method to calculate the gravity for a point in 3D Space
  virtual agx::Vec3 calculateGravity(const agx::Vec3& position) const
  {
    return position*-1;
  }
protected:
  ~MyGravityField() {}

};

// Register the gravity field with the simulation
simulation->setGravityField( new MyGravityField() );
```

13.14 Materials

The material of a geometry will define how the geometry will get affected and affect the rest of the system. A material contains definitions of friction, restitution, density and other parameters that describe the physical surface, bulk and line attributes of a material. A material can be associated to a geometry with the method:

```
agxCollide::Geometry::setMaterial(agx::Material*)
```

An important thing to remember is that the name of a material has to be unique. Adding two materials with the same name will result in that only the first will be used in the system.

New materials must be registered with the *agxSDK::MaterialManager* before they can be used in the simulation (which is done when the material is added to the simulation).

When two geometries collide, the resulting contact parameters from the two involved materials are derived into an *agx::ContactMaterial*. The default contact material function can be overridden by adding explicit contact materials to the material manager.

Materials consist of three sub-classes which can be accessed through the calls:

```
material->getSurfaceMaterial();
material->getBulkMaterial();
material->geWireMaterial();
```

13.14.1 SurfaceMaterial

This class defines the behavior/surface attributes for contacts between geometries using two materials. The attributes currently available are:

- Roughness – Corresponds to friction coefficient, the higher value, the higher friction.
- Adhesion – determines a force used for keeping colliding objects together.
- Viscosity – Defines the compliance for friction constraints. It defines how “wet” a surface is.
- ContactReduction

13.14.1.1 ContactReduction

This parameter specifies whether contact reduction should be enabled for this material. If two geometries overlap, the resulting contact material will get contact reduction enabled only if both materials have contact reduction enabled (logical and).

13.14.1.2 Adhesion

Adhesion defines an attractive force that can simulate stickiness. It operates only on bodies that have colliding geometries, i.e. it is not a general force field.

Adhesion is set using two parameters:

```
SurfaceMaterial::setAdhesion( adhesionForce, adhesiveOverlap );
```

adhesionForce specifies a force >=0, that is an upper bound of the attractive force acting between two colliding rigid bodies in the normal direction.

adhesiveOverlap is an absolute distance that defines the contact overlap where the adhesive force is active. Because that the adhesive force only acts upon overlapping rigid bodies, it is important to specify an overlap > 0. Otherwise a separating force would immediately (within a timestep) cause two bodies to separate, hence no adhesive force would then act on the bodies. The adhesiveOverlap is a safe zone to ensure that there will be time for the adhesiveForce to act.

Because the adhesive force operates only in the normal direction of the contact, it is also important to increase the roughness/friction to get a sticky behavior also in the friction direction.

max_adhesive_force ~ adhesiveOverlap / contact_compliance

The adhesion distance will change the depth of the resting contact as seen in Figure 16.

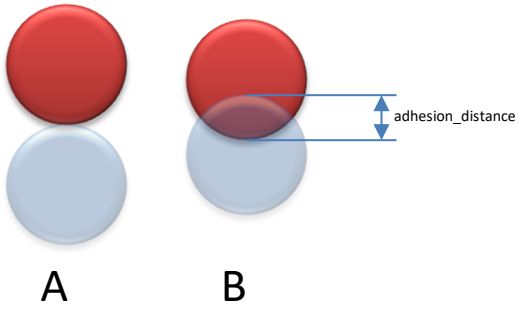


Figure 16: A: no adhesion. B: adhesion_distance=r/2

13.14.2 BulkMaterial

This class defines the density and other internal attributes of a simulated object. Lines use for example YoungsModulus and PoissonsRatio to calculate bending stiffness etc.

Attributes are:

- *Density*
- *YoungsModulus* – Stiffness of the material. Used when calculating contacts and the bending behavior of lines.
- *PoissonsRatio* – How much the material tends to expand/contract in two directions when it is stretched/compressed in the third direction.
- *Viscosity* – Used for calculating restitution.
- *Damping* – Used for controlling how much time it should take to restore the collision p*enetratio*n. A larger value will result in "softer" contacts.

13.14.3 WireMaterial

Defines the behavior when a geometry with this material collides with a line. Determines the stick and slide friction. When a line is colliding with a geometry (where the line is overlapping an edge of a box/trimesh, or onto a cylinder), a *ContactNode* is created. This node is allowed to slide along the edge and also in the direction along the line. To solve for the normal force, a 1D solver will analyze the system and solve how the contact node is allowed to move, depending on masses, gravity, physical properties in the line and the tension in the line. In some cases where this solver is not capable of finding a proper value for the friction, the *KinematicContactNodeVelocityScale* is used to control the velocity of the sliding node. So it will work as a fallback.

Attributes:

- *FrictionCoefficient* – The friction value between a line and other geometries.
- *KinematicContactNodeVelocityScale* – Fallback for the friction value, when the line contact solver is not able to resolve the system.

Damping and *YoungsModulus* can be specified for two attributes in the line material:

- *Bend* – Constraint which tries to straighten a wire.
- *Stretch* - Constraint which works against stretching the wire.

For more information, see the SDK documentation on Material.

13.14.4 ContactMaterial

The class agx::ContactMaterial is used by the system to derive the actual collision attributes when two materials collide. It can also be used to explicitly define a material used at contact calculations. All attributes found in the surface, bulk and line material is represented in the ContactMaterial, which is later used by the solver.

AGX use by default a contact point based method for calculating the corresponding response between two overlapping geometries.

There is also a different method available which is named *area contact approach.* This method will try to calculate the spanning area between the overlapping contact point. This can result in a better approximation of the actual overlapping volume and the stiffness in the response (contact constraint).

In general, this will lead to slightly less stiff, more realistic contacts and therefore the Young's modulus value usually has to be increased to get a similar simulation results as running the simulation without the contact area approach.

```
// Create a contact material based on two agx::Material's m1 and m2
agx::ContactMaterial *cm = simulation->getOrCreateContactMaterial(m1, m2);

// When two geometries collide, where m1 and m2 is involved, the contact area approach
// will be used.
cm->setUseContactAreaApproach( true );
```

```
agx::Real minElasticRestLength=0.0005, agx::Real maxElasticRestLength=0.05;
cm->setMinMaxElasticRestLength(minElasticRestLength, maxElasticRestLength);
```

The above code line determines the contact depth-span in which the material is elastic. AGX calculates this distance, but the material settings will put limits on this value. It can not be less than the minimum material rest length or greater than the maximum material rest length. For this reason, the minimum material rest length can not be greater than the maximum material rest length. For the theory background, see 4.7 Contact mechanics (Elastic Foundation Model).

13.15 Calculating contact friction

At present, there is no difference between static and kinetic friction coefficients. According to published data and common models, these two coefficients differ by 10-20%. This is relevant in simulation of gear dynamics but not in most other cases. Friction coefficients do depend on the two materials in contacts.

So given material1 with roughness 0.5 and material2 with roughness 0.2, what friction coefficient should be used when these two materials get in contact?

There is no clear picture of how to calculate this resulting friction coefficient in the general case.

Therefore, by default, the resulting friction when material1 and material2 collides will be the *geometric averaged* value:

```
sqrt(material1.roughness*material2.roughness) = 0.31623
```

This is of course a crude approximation which does not have any relation to the real world physics. To overcome this, AGX supports *explicit ContactMaterial, see 13.15.2.*

Contacts are handled in two separate stages, namely, *impact*, and *resting contact*. Impacts correspond to a rigid body which is found in geometric interference at the beginning of a step with sufficiently large incident velocity, i.e., when the relative velocity projected on the separating normal is sufficiently negative. During impact resolution, the impacting energy is *approximately* restituted to the level of the restitution coefficient. This is done by computing an impulsive stage which preserves all constraints but applies a Newton restitution model on impacting contacts, with post impact velocity a fraction of the pre-impact incident velocity. That fraction is the restitution coefficient and it is usually a number between 0 and 1. The restitution coefficient applies to all pairs of rigid bodies and is part of the pair-wise contact properties.

Because we do not locate the exact time of impact, there can be a *small* decrease in energy even if restitution is exactly 1.

After impact, constraint *regularization* and *stabilization* come into play. From the modeling perspective, this is no more and no less than a spring/damper system. Numerically, the spring and damper system is unconditionally for nonzero damping. The two parameters are accessible as *contact compliance* (1/YoungsModulus) and *damping coefficient* in the material properties.

Compliance is just the inverse of the desired spring constant, with the usual units. The AGX damping parameter (also called "Spook damping parameter") has units of time for technical reasons. Use the functions in `agxUtil::Convert` in order to convert from viscous damping coefficient to AGX' damping coefficient.

There are basically two ways of setting up the material table for a complete system: defining *implicit* or *explicit* material properties.

13.15.1 Defining implicit Material properties

By creating an agx::Material, setting the properties, assigning it to a geometry and adding it to the Simulation, we have told the system that this geometry is created from some kind of material.

When this geometry collides with another geometry, the pair of materials are used to calculate the resulting agx::ContactMaterial.

This is a fairly easy way of creating the materials in the system, as every pair of materials does not really have to be known in beforehand. The calculation of material properties for the implicit contact material *cm* between material *m1* and *m2* is done as follows:

- cm.surfaceFrictionCoefficient = sqrt(m1.roughness*m2.roughness)
- cm.restitution = sqrt((1-m1.viscosity) * (1-m2.viscosity))
- cm.adhesion = m1.adhesion+m2.adhesion
- cm.youngsModulus = m1.youngsModulus * m2.youngsModulus / (m1.youngsModulus + m2.youngsModulus)
- cm.damping = max(m1.damping, m2.damping)
- cm.surfaceViscosity = m1.surfaceViscosity+m2.surfaceViscosity

The frictional parameters for wire friction are using the same geometric medium as surface friction.

13.15.2 Defining explicit ContactMaterial properties

If the actual values used in the friction solver has to be known exactly, the above method is not appropriate. Instead an explicit *material table* should be created.

This can be done by associating two **agx::Material**, *M1* and *M2* to *one user created* **agx::ContactMaterial** *CM*. So whenever the two materials *M1* and *M2* are involved in a contact, the attributes of *CM* are used. However, the following attributes of M1 and M2

still have to be set, to correctly calculate the mass properties of a RigidBody and to have correct behavior for LineConstraints.

```
BulkMaterial::setDensity()
BulkMaterial::setYoungsModulus()
BulkMaterial::setPoissonsRatio()
```

At contact between two agx::Material's the following list of attributes are used and should therefore be set to the required value.

```
ContactMaterial::setRestitution()
ContactMaterial::setFrictionCoefficient()
ContactMaterial::setLineStickFriction()
ContactMaterial::setLineSlideFriction()
ContactMaterial::setDamping()
ContactMaterial::setEnableContactReduction()
```

Friction	Plastic	Metal	Wood
Plastic	??	0.7	0.3
Metal	-	??	0.2
Wood	-	-	??

Table showing friction coefficients between different materials, defined via explicit or implicit contact materials.

To build a table of explicit materials each ContactMaterial are associated to two agx::Material's. Code for setting up this table could look like:

```
agx::MaterialRef wood = new agx::Material("Wood");
agx::MaterialRef plastic = new agx::Material("Plastic");
agx::MaterialRef metal = new agx::Material("Metal");

// Define the material for plastic/metal
agx::ContactMaterialRef plastic_metal = new agx::ContactMaterial(plastic, metal);
plastic_metal->setRestitution( 0.6 );
plastic_metal->setFrictionCoefficient( 0.7 );
plastic_metal->setYoungsModulus( 0.1E9 ); // GPa
// Define the material for wood/metal
agx::ContactMaterialRef wood_metal = new agx::ContactMaterial(wood, metal);
wood_metal->setRestitution( 0.2 );
wood_metal->setFrictionCoefficient( 0.2 );
```

```
// Define the material for wood/plastic
agx::ContactMaterialRef wood_plastic = new agx::ContactMaterial(wood, plastic);
wood_plastic->setRestitution( 0.4 );
wood_plastic->setFrictionCoefficient( 0.3 );
wood_plastic->setYoungsModulus( 3.654E9 ); // GPa

// Add ALL materials to the simulation
simulation->add(wood)
simulation->add(metal)
simulation->add(plastic)
simulation->add(plastic_metal);
simulation->add(wood_metal);
simulation->add(wood_plastic);
```

The above code example still leaves to the system to automatically derive the ContactMaterial for wood/wood, plastic/plastic and metal/metal.

13.15.2.1 ContactReduction

This parameter controls whether contact reduction should operate on the contacts where this contact material is used. For details, see chapter “14 Contact reduction”.

13.15.2.2 Contact Area approach

If set to “true”, an approximation to the contact area will be geometrically computed for each contact involving this contact material. For each contact, its area will then be evenly distributed between its contact points. The contact compliance will be scaled with the inverse of the area for each contact point.

Note that this is an experimental feature, which has been tested on contacts involving meshes and boxes, but with a cruder approximation for contacts involving spheres or capsules.

Recommended for contact mechanics with a higher level of fidelity, such as grasping (involving meshes/boxes).

If set to “false”, unity area will be assumed (default).

```
// Define the material for wood/plastic
agx::ContactMaterialRef wood_plastic = new agx::ContactMaterial(wood, plastic);
wood_plastic->setUseContactAreaApproach(true);
```

13.15.3 Friction model

AGX has a hybrid solver which is very flexible in the way constraints can be configured. By default, contact constraints are solved in a *split* mode. This means that the normal force is solved in the direct solver and the friction is solved in the iterative solver. This behavior can be modified using ContactMaterials.

In the below examples, assume that the following code has been executed:

```
using namespace agx;
using namespace agxCollide;
// Create two materials
MaterialRef wood = new Material("wood");
MaterialRef steel = new Material("steel");

// Create two geometries with material steel and wood
GeometryRef geom1 = new Geometry;
geom1->setMaterial("wood");
GeometryRef geom2 = new Geometry;
geom2->setMaterial("steel");

// Create a contactmaterial that will be used when steel collides with wood
ContactMaterialRef steelWood = simulation->getMaterialManager()->getOrCreateContactMaterial(
wood, steel );
```

13.15.3.1 BoxFriction

A naive implementation of this friction model will have static bounds through the solve stage. This means that for a stack of boxes, the bottom box, will not feel the weight of the other boxes. The normal force will be constant, as specified by the user. In our implementation we are estimating the normal force as follows:

- If the contact is now (i.e., new to the solver), the normal force is estimated by the impact (relative) speed.
- If the contact is old (i.e. created in a previous time step), the last normal force is used to set the bounds.

```
BoxFrictionModelRef boxFriction = new BoxFrictionModel();

// Use box friction as the friction model for this contact material
steelWood->setFrictionModel (boxFriction);
```

13.15.3.2 ScaledBoxFriction

The ScaledBoxFriction model gets callbacks from the NLMCP (Non-Linear Mixed Complimentary) solver with the current normal force. I.e., the correct normal force is used when setting the bounds. Scale box friction is computationally more expensive than the box friction but with a more realistic dry friction.

```
ScaledBoxFrictionModelRef scaledBoxFriction = new ScaledBoxFrictionModel();

// Use scaled box friction as the friction model for this contact material
steelWood->setFrictionModel (scaledBoxFriction);
```

13.15.3.3 IterativeProjectedConeFriction

This friction model is the default, with splitting of the normal-tangential equations. I.e., the normal forces are first solved with a direct solve, and then both the normal and tangential equations are solved iteratively. *For complex systems this can lead to viscous friction*. This can be resolved by using DIRECT_AND_ITERATIVE solve model for constraints involved in the system. Iterative projected cone friction model is computationally cheap. Friction forces are projected onto the friction cone, i.e., you will always get friction_force = friction_coefficient * normal_force.

```
IterativeProjectedConeFrictionRef ipcFriction = new IterativeProjectedConeFriction();

// Use iterative projected cone friction as the friction model for this contact material
steelWood->setFrictionModel (ipcFriction);
```

13.15.4 Solve type

As with any other constraint, we can also define how the friction will be solved:

```
    FrictionModel::setSolveType (FrictionModel::SolveType solveType);
```

FrictionModel::SolveType	Description
DIRECT	Normal and friction equation are solved only in the direct solver.
ITERATIVE	Solved only in the iterative solver.
SPLIT	Normal force solved in direct solver, then normal and friction is solved in the iterative.
DIRECT_AND_ITERATIVE	Normal and friction are solved both in the direct and the iterative solver.

Table 10: Solve types for friction models.

13.16 Testing user geometry against Space

In some cases it is of interest to intersect the current configuration in the collision space with some geometries, without actually inserting a new geometry into the collision space.

As soon as a geometry is inserted into space, it is a part of the simulation. Also testing for contacts in between two simulation frames can sometime be used for example for path finding etc.

There are two methods for achieving this in agxCollide::Space:

```
bool Space::testBoundingVolumeOverlap(
      agxCollide::GeometryPtrVector& testGeometries,
      agxCollide::GeometryPairVector& result,
     bool skip = true );

bool Space::testGeometryOverlap(
     agxCollide::GeometryPtrVector& testGeometries,
     agxCollide::GeometryContactVector& result,
     bool skip = true );
```

testBoundingVolumeOverlap will take a vector of Geometries and return pairs of Geometries with overlapping bounding volumes.

testGeometryOverlap will take a vector of Geometries and return a vector of geometry contacts, including contact points, penetration, normals etc. ...

13.17 Reading forces from the simulation

Often when you build a simulation you also want to *read* forces/torques from the system. Assume you have built your scene with geometries, bodies, constraints and wires. Now how do you access the forces from your simulation?

Note: As in many parts of this documentation, the term “force” is here used interchangeably for a 3d force vector in contrast to a 3d torque vector and, where applicable, as a 6d vector consisting of both force and torque.

Forces can be divided into a few separate categories:

- External/ForceField/Gravity forces
- Constraint forces
- Contact forces

13.17.1 External/ForceField/Gravity forces

Forces and torques can be applied to a body by the methods:

`RigidBody::addForce, RigidBody::addTorque`, etc. see 13.11.

External forces also include forces added by ForceFields, GravityFields. These external forces can at any time be read from a given rigid body with the method:

```
const agx::Vec3& RigidBody::getLastForce() const;
const agx::Vec3& RigidBody::getLastTorque() const;
```

13.17.2 Constraint forces

It is possible to read constraint forces applied to rigid bodies during the last time step *for each constraint.* However, it is currently not possible to ask a rigid body for its constraint forces directly - you have to keep track of the constraints attached to a specific body yourself and ask those constraints for the forces.

13.17.2.1 Constraint::getLastForce per body

For computational reasons, the calculation of constraint forces is disabled by default. To enable the computation of constraint forces call the method:

```
void Constraint::setEnableComputeForces(bool);
```

once for each constraint that you want to read forces from in the future.

Then, *after the solver has been executed*, you can access (e.g. with a StepEventListener in POST) the forces/torques applied to the constrained bodies with a call to:

```
// If you know the index of your body in the Constraint (0 or 1)
bool Constraint::getLastForce( UInt bodyIndex, Vec3& retForce, Vec3& retTorque, bool
giveTorqueAtCm=false ) const;

// If you have pointer to the rigid body
bool Constraint::getLastForce( const RigidBody* rb, Vec3& retForce, Vec3& retTorque, bool
giveTorqueAtCm=false ) const;
```

Note that getLastForce is only valid when called for a rigid body with motion control DYNAMICS or KINEMATICS (not STATIC). Check the return value to detect invalid calls to getLastForce.

The returned force and torque by getLastForce are defined in the world coordinate system.

The returned torque is dependend on the value of the flag *giveTorqueAtCm*:

- A value of *true* gives the torque in the body's center of mass (corresponding to *RigidBody::addTorque(Vec3)*),
- whereas the default value of *false* gives the value in the constraint's attachment point *r* for the body, so that
  tau_attachment = tau_cm + r \cross F

13.17.2.2 Constraint::getCurrentForce per constraint-DOF

If you are interested in obtaining the force applied by the constraint at a certain degree of freedom (in the constraint's local coordinate system), you can use

```
// \return the current force magnitude (i.e., the force magnitude last time
step) in given DOF >= 0 && <= NUM_DOFS
bool Constraint::getCurrentForce( UInt degreeOfFreedom) const;
```

Note that the degrees of freedom have different meanings for each constraint type; the `enum DOF` which is defined per constraint should be used.

The method `getCurrentForce` exists also for secondary constraints (see 15.12.5).

The naming difference in `getCurrentForce` and `getLastForce` is misleading since both methods read the forces from the same source and time. These methods will likely be renamed in the future to reflect their behavior better.

13.17.3 Contact forces

Contact forces are in included in the contact data, together with contact points, normals depths, … . Contact forces are valid *after the solver has been executed*. You can access them (e.g. with a StepEventListener in POST) like in the following example.

Assume you want to collect the sum of the absolute values of all the contact forces (normal and friction) being applied on a specific rigid body:

```
#include <agxCollide/Contacts.h>
#include <agxCollide/Space.h>

// Return the sum of the magnitude for all contact forces being applied to a specific body.
agx::Real getContactForce(agx::RigidBody* body, agxCollide::Space* space)
{
  // Get the vector with all the contacts
  const agxCollide::GeometryContactPtrVector& contacts = space->getGeometryContacts();

  agx::Real contactForce = 0;
  // For all contacts in the system
  for (size_t i = 0; i < contacts.size(); ++i)
  {
    agxCollide::GeometryContact *gc = contacts[i];
    // Is the contact enabled?
    if (!gc->isEnabled())
      continue;
    // Is our body involved in this contact?
    if (gc->rigidBody(0) != body && gc->rigidBody(1) != body)
      continue;

    // Get all the points and sum the magnitude for the contact force (normal force)
    const agxCollide::ContactPointVector& points = gc->points();
    for (size_t n = 0; n < points.size(); ++n)
    {
      // Is the contact point enabled?
      if (!points[n].enabled())
        continue;
      contactForce += points[n].getNormalForceMagnitude();
    }
  }
  return contactForce;
}
```

14 Contact reduction

By using contact reduction, the number of contact points later submitted to the solver as contact constraint can be heavily reduced, hence improving performance.

14.1 Motivation

Assume a rigid body has one or several geometries associated to it. When these geometries collide with the geometries from another rigid body, contact points will be generated. Given a complicated geometry or shape, or many overlapping geometries, a lot of contact points will be created. Each contact point will later lead to a contact constraint which must be handled by the solver. Since frictional contacts are relatively expensive to solve, the fewer contact points/constraints, the, the better from a CPU usage point of view. Also, in some cases too many contacts can lead to an over-determined system and cause the simulation to deliver bad results.

For example: a box on a plane only needs 3-4 contacts to stand stable. Assume you have a height field as a plane, and a trimesh as a box, both being highly tessellated. This scenario could easily generate 100s of contacts, where only 3-4 are necessary.

14.2 Technical details

The reduction is done by the class agxCollide::ContactReducer, where contact points are reduced by a 6-dimensional binning algorithm in [nxp, n]-space. Here, n is the contact normal and p the contact point relative to a reference point (e.g. one body's center of mass, or the world origin).

In the overall physics pipeline, contacts can get reduced on two levels:

- Directly in the narrow phase stage, contact reduction is done on each geometry-geometry interaction (can be controlled, see below). Contact points that are reduced get removed immediately.
- *After* collision detection and *before* stepping the Dynamics system (PRE), contact reduction is done on some rigidbody-rigidbody interactions (can be controlled, see below).

The contact reduction within the physics pipeline can be controlled in three ways:

1. By setting the *reduction mode* (ContactReductionMode). Either no reduction, or per geometry overlap, or per geometry overlap AND per rigid body overlap.
2. By setting the bin resolution (number of bins per dimension). Given a bin resolution of n, this could lead to n^6 contacts being left after reduction in worst case. Average case is about n^2.
   The bin resolution can be set differently for geometry overlaps and rigid body overlaps.
3. By setting the threshold for doing contact reduction.
   In order not to waste time on doing contact reduction for contacts with small numbers of contact points, contact reduction is only done on contacts which exceed a certain number of contact points. This threshold can be set separately for geometry-geometry-overlaps and rigidbody-rididbody-overlaps.

It is possible to call *agxCollide::ContactReducer::reduce* manually.

One obvious use case is wanting to analyze all contact points before sending them to the solver. This can be done by

- setting the reduction mode to REDUCE_NONE for a contact material as in the example in the next chapter,
- and then in a ContactEventListener:
  - analyze the contacts as desired
  - do a manual call to contact reduction
- Note that this might interfere with other ContactEventListeners listening to the same contact!

Note that currently, the only directly supported way to do contact reduction and send the reduced contacts to the solver is in ContactEventListeners.

14.3 ReductionMode

AGX has two steps where contacts are reduced:

- For a pair of overlapping geometries (with potentially several shapes)
- For a pair of overlapping rigid bodies (with potentially several geometries).

By default, AGX does one contact reduction pass between each pair of overlapping geometries.

This behavior can be changed to one of the three modes:

```
      /// Specifies the mode for contact reduction
      enum ContactReductionMode {
        REDUCE_NONE,               /**< No contact reduction enabled */
        REDUCE_GEOMETRY,           /**< Default: Reduce contacts between geometries */
        REDUCE_ALL                 /**< Two step reduction: first between geometries, and then
between rigid bodies */
      };
```

The contact reduction mode can be set per contact material.
An example:

```
// We want to reduce contacts not only between each geometry, but between body and other body

// Assume we have a body and five geometries, each with e.g. one or more trimesh shapes.
rigidBody->add( geometry1 );
rigidBody->add( geometry2 );
rigidBody->add( geometry3 );
rigidBody->add( geometry4 );
rigidBody->add( geometry5 );

// Assign a material to the geometries
geometry1->setMaterial( material1 );
geometry2->setmaterial( material1 );

// Assume we also have another body which we also assign a material to
otherBody->add( otherGeometry );
otherGeometry->setMaterial( material2 );

// Create a new contact material
agx::ContactMaterialRef cm = simulation->getMaterialManager()-
>getOrCreateContactMaterial(material1, material2);

// Set contact reduction mode to REDUCE_ALL - now, contacts between rigidBody and otherBody
// get reduced together, not only each contact by itself.
cm->setContactReductionMode( agx::ContactMaterial::REDUCE_ALL );
```

Note that REDUCE_ALL is not the default for a reason:

Often, bodies are assigned different geometries because one would like to set different materials on these geometries, resulting in different physical behavior (e.g. vary friction, Young's modulus, …).
Contact reduction with REDUCE_ALL does not take these differences into account, but reduces over all contact points. However, each contact point will remain in the GeometryContact it belongs to, and only GeometryContacts with a contact material with REDUCE_ALL set will be reduced upon.

All geometries without a rigid body will be treated as belonging to the same one when doing contact reduction between rigid bodies. If you want to avoid geometry contacts from several geometries without rigid bodies to end up in the same reduction, move these geometries into separate (static) rigid bodies.

14.3.1 Bin resolution

The bin resolution gives the number of bins per dimension. Given a bin resolution of n, this could lead to n^6 contacts in worst case, where the average case is about c*n^2 (common contact region, like in convex overlaps).

The bin resolution can be set differently for geometry overlaps and rigid body overlaps, as well as for manual contact reduction.

In the general case, a bin resolution of between 1 and 10 is allowed, where typical values are 2 or 3.

Below is a more detailed description.

14.3.1.1 Geometry overlaps

The bin resolution for contact reduction for rigid body overlaps can only be given for the whole simulation by setting the value like in this example:

```
// Set contact reduction bin size to 3 for simulation (rigid body overlaps).
sim->setContactReductionBinResolution( 3 );
```

The bin resolution for contact reduction for geometry overlaps can be set per contact material:

```
// Create a new contact material
agx::ContactMaterialRef cm = simulation->getMaterialManager()-
>getOrCreateContactMaterial(material1, material2);

// Set contact reduction bin size to 3 for contact material (all geometry overlaps with this
contact material).
cm->setContactReductionBinResolution( 3 );
```

If the bin resolution is set to zero (not allowed in other contexts, see above) in the contact material or if no contact material is used, a fallback value from agxCollide::Space is used which can be specified like in this example:

```
// Set contact reduction bin size to 3 for space (fallback value for geometry overlaps).
agxCollide:;SpaceRef = sim->getSpace();
space->setContactReductionBinResolution( 3 );
```

14.3.1.2 Rigid body overlaps

The bin resolution for contact reduction for rigid body overlaps can only be given for the whole simulation by setting the value like in this example:

```
// Set contact reduction bin size to 3 for simulation (rigid body overlaps).
sim->setContactReductionBinResolution( 3 );
```

14.3.1.3 Manual contact reduction

The bin resolution for manual contact reduction can be given in the reduce-call to contact reducer (a static method):

```
// Set contact reduction bin size to 3 for simulation (rigid body overlaps).
size_t binResolution = 3;
agxCollide::ContactReducer::reduce(contactPoints, binResolution);
```

14.3.2 ContactReductionThreshold

The contact reduction between overlapping geometries is enabled by default (otherwise dependent on the material's reduction mode, see chapter above) and will be activated when the number of contact points between two geometries is larger than a specified threshold. This threshold is set globally in agxCollide::Space (since space is responsible for geometries):

```
agxCollide::SpaceRef space = new agxCollide::Space();
// More than 10 contact points will trigger contact reduction
space-> setContactReductionThreshold( 10 );
```

The additional contact reduction between overlapping rigid bodies is disabled by default (otherwise dependent on the material's reduction mode, see chapter above) and will be activated when the number of contact points between two rigid bodies is larger than a specified threshold. This threshold is set globally in agxSDK::Simulation:

```
agxSDK::SimulationRef sim = new agxSDK::Simulation();
// More than 10 contact points will trigger contact reduction
sim-> setContactReductionThreshold( 10 );
```

15 Constraints

A Constraint will define a geometrical relationship between one rigid body relative to the world, or two or more rigid bodies relative to each other, that should be met under certain conditions. Each constraint is built upon *elementary constraints,* which define the geometrical relationship. Acting upon these elementary constraints can be s*econdary order* constraints, motors, limits or locks.

Example of how to use the constraint API is showcased in the **tutorials/tutorial_tutorial5.cpp**

One important thing to keep in mind when constructing systems with constraints is that:

Always position/rotate the rigid bodies involved in a constrained system before attaching them to a constraint.

Available constraints in the current AGX version are:

Constraint name	Description
BallJoint	Ball and socket joint
DistanceJoint	Restore distance between two specified points on bodies.
Hinge	Allow rotation around one specified axis (revolve)
UniversalJoint	Transmission joint.
LockJoint	Removes all degrees of freedom between two bodies, keep the locked together.
Prismatic	Allows translation along one axis (slide)
PlaneJoint	Reduce translation movement to a plane
CylindricalJoint	Hinge and Prismatic combined
AngularLockJoint	Reduces all rotational DOF between two bodies.

Table 11: Available constraints.

15.1 Elementary constraints

Elementary constraints are constraints that ordinary, more complex constraints are based upon. These elementary constraints are:

Name	Description
Dot-1	Given two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, this constraint will retain $\mathbf{v}_1 \cdot \mathbf{v}_2 = 0$.
Dot-2	Given a separation vector $\mathbf{d}^{ij}$ between body *i* and *j* and and body fixed vector $\mathbf{a}^i$, this constraint will retain $\mathbf{d}^{ij} \cdot \mathbf{a}^i = 0$.
Spherical	This is a 3D constraint which computes $\mathbf{x}^i + \mathbf{p}^i - \mathbf{x}^j - \mathbf{p}^j = 0$, where **x** is a body fixed vector and **p** the position of that vector.
Distance	Given two positions $\mathbf{p}^i$ and $\mathbf{p}^j$, this constraint will retain $\lvert \mathbf{p}^i - \mathbf{p}^j \rvert = l_0$, where l_0 is some constant.
QuatLock	Locks all rotational degrees of freedom between two bodies.

Table 12: Elementary constraints.

For example, a hinge is a composite of two Dot-1 and one Spherical elementary constraint.

15.1.1 Converting spring constant and damping

In the agxUtil namespace there are functions for converting from the spring constant and damping to AGX regularization parameters:

Converting from a spring constant to compliance: compliance = 1/springConstant:

```
// Convert from a spring constant to compliance
agx::Real compliance = agxUtil::convert::convertSpringConstantToCompliance( springConstant );
// Use the compliance
constraint->setCompliance( compliance );
```

Convert (viscous) damping coefficient to spook damping, where:

dampingCoefficient – Damping. For linear dimensions, in force*time/distance; for rotational dimensions, in torque*time/radians.
springConstant – Spring constant. For linear dimensions, in force/distance; for rotational dimensions, in torque/radians.

```
agx::Real spookDamping = agxUtil::convert::convertDampingCoefficientToSpookDamping(
   dampingCoefficient, springConstant);

constraint->setDamping( spookDamping );
```

15.1.2 Regularization parameters

Ordinary constraints can be modeled using regularization parameters. These parameters specify the *compliance* i.e. stiffness of the constraint and *damping*, i.e. how fast the constraint should be restored to a non-violated state.

The regularization parameters are accessed with the call:

```
// Set compliance for all DOF:s
hinge->setCompliance( 1E-14 );
```

To access the regularization parameters for each degree of freedom (DOF):

```
// Set compliance for all DOF:s
for (size_t dof=0; dof < hinge->getNumDOF(); dof++)
  hinge->getRegularizationParameters( dof )->setCompliance( 1E-4 );
```

The ordering of DOF is different per constraint; see the `enum DOF` which is defined per constraint in order to set meaningful values per DOF.

15.1.3 Attachment frames

All constraints uses agx::Frame to define the geometrical relationship between the constrained body and the world, *or* between two constrained rigid bodies.

A Frame for a constraint defines a coordinate system attached to a rigid body. The constraint is satisfied/relaxed when these two coordinate systems/frames are coincident in the world. For example, assume a LockJoint is created between the two rigid bodies A and B below. When creating the constraint, two agx::Frame are used. The frames are translated relative to the origin of the bodies (which are assumed to be the green dot in the center).

```
agx::RigidBodyRef A, B; // Two bodies
// Create an attachment frame relative body A:s model origin
agx::FrameRef frameA = new agx::Frame();
frameA->setLocalTranslate(-0.5,0,0.5); // Translate the frame left and up relative to body A.
A->addAttachment(frameA, "frameA");

// Create an attachment frame relative body B:s model origin
agx::FrameRef frameB = new agx::Frame();
frameB->setLocalTranslate(0.5,0,-0.5); // Translate the frame left and up relative to body B.
B->addAttachment(frameB, "frameB");

// Create a LockJoint using the attachments
agx::LockJointRef lock = new agx::LockJoint(a, a->getAttachment("frameA"),
                                            b, b->getAttachment("frameB"));
```

Figure 17 below show the initial configuration (left) after executing the above code. To the right, the relaxed configuration is shown. *When the two attachment frames are coincident in space, the constraint is relaxed.*

Notice also that a frame can be associated to a rigid body using a name: **RigidBody::addAttachment(Frame *, const agx::String&)** This only means that the rigid body will keep a reference to the frame for later use. Think of attachments as “snap on” targets.

Figure 17: Left, initial configuration. Right, relaxed configuration.

15.2 Ordinary constraints

Ordinary constraints (derived from the class *agx::Constraint*) are composites of elementary and secondary constraints. They can be used in a simulation to build up machines, such as cars, elevators etc. Ordinary constraints also consist of *secondary constraints*, which can define motors and limits. A body without constraints, is free to *translate* (t_1, t_2, t_3) and *rotate* (r_1, r_2, r_3), a total of 6 degrees of freedom (DOF). A constraint will restrict the motion of an attached body to one or more DOF.

For example, a *Prismatic* will allow a body to translate along one axis only, the other five DOF are constrained, while a *LockJoint* will restrict the movement of a body to all six DOF. Table 13 below show the number of degrees of freedom removed for each constraint.

Constraint	#Degrees of freedoms removed	#Translational	#Rotational
Hinge	5	3	2
LockJoint	6	3	3
AngularJoint	3	0	3
Prismatic	5	2	3
CylindricalJoint	4	2	2
DistanceJoint	1	1	0
UniversalJoint	4	3	1
BallJoint	3	3	0

Table 13: Degrees of freedoms removed for constraints.

Table 14 lists each ordinary constraint with their respective elementary and secondary constraints.

Constraint	ElementaryConstraints	SecondaryConstraints
Hinge	1 x **Spherical**, 2 x **Dot1**	**Range1D**, **Motor1D**, **Lock1D**
LockJoint	1 x **Spherical**, 1 x **QuatLock**	None
AngularJoint	1 x **QuatLock**	None
Prismatic	3 x **Dot1**, 2 x **Dot2**	**Range1D**, **Motor1D**, **Lock1D**
CylindricalJoint	2 x **Dot1**, 2 x **Dot2**	2x**Range1D**, 2x**Motor1D**, 2x**Lock1D**
DistanceJoint	1 x **Distance**	None
UniversalJoint	1 x **Spherical**, 1 x **Dot1**	Motor
BallJoint	1 x **Spherical**	None

Table 14: Ordinary Constraints and their Elementary and Secondary constraints.

Each constraint declares an enum, specifying an index for each DOF that it removes:

```
enum DOF
{
  ALL_DOF=-1,        /**< Select all degrees of freedom */
  TRANSLATIONAL_1=0, /**< Select DOF for the first translational axis */
  TRANSLATIONAL_2=1, /**< Select DOF for the second translational axis */
  TRANSLATIONAL_3=2, /**< Select DOF for the third translational axis */
  ROTATIONAL_1=3,    /**< Select DOF corresponding to the first rotational axis */
  ROTATIONAL_2=4,    /**< Select DOF corresponding to the second rotational axis */
  ROTATIONAL_3=5,    /**< Select DOF corresponding to the third rotational axis */
  NUM_DOF=6          /**< Number of DOF available for this constraint */
};
```

Which can be used as follows:

```
// Set the compliance for translational movements for the hinge:
hinge->setCompliance( 1E-8, 1EHinge::TRANSLATIONAL_1 );
hinge->setCompliance( 1E-8, 1EHinge::TRANSLATIONAL_2 );
hinge->setCompliance( 1E-8, 1EHinge::TRANSLATIONAL_3 );
```

Most constraints will also return the number of DOF it applies onto a body with the method:

```
// Get the number of DOF for this constraint. -1 if numDof is not defined for the constraint.
int numDof = hinge->getNumDOF();
```

15.3 Hinge

A Hinge removes 5 degrees of freedom from the constrained body allowing for rotation around an axis. A hinge has an angular range, lock and motor.

The geometrical configuration of a hinge in AGX is defined by using Attachment frames (15.1.3). By default, a hinge revolves around the positive Z-axis.

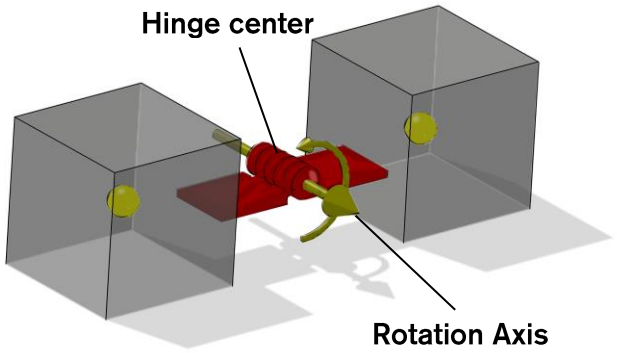


Figure 18: Illustration of a hinge where (1) is the hinge center and (2) the axis. The bodies are free to rotate around this axis.

Hinge example 1: Create a single body hinge (connected to world)

```
// body1 is a rigid body
agx::FrameRef frame = new agx::Frame();
frame->setLocalTranslate( -1.0, 0.0, 0.0 ); // Hinge axis translated -1 along X

// Align hinge axis along negative y-axis instead of Z.
frame->setLocalRotate( agx::Quat(agx::Vec3::Z_AXIS(), agx::Vec3::Y_AXIS()*(-1)) );
agx::ref_ptr< agx::Hinge > h = new agx::Hinge( frame, body1 );
```

Hinge example 2: Connect two bodies with a hinge which reproduces the example from Figure 18 assuming body1 is the one to the left:

```
// body1 and body2 are rigid bodies
agx::FrameRef frame1 = new agx::Frame;
agx::FrameRef frame2 = new agx::Frame;
frame1->setLocalTranslate( 1,0,0);
frame1->setLocalRotate( agx::EulerAngles(agx::degreesToRadians(90.0),0,0) );
frame2->setLocalTranslate( -1,0,0);
frame2->setLocalRotate( agx::EulerAngles(agx::degreesToRadians(-90.0),0,0) );

agx::HingeRef h = new agx::Hinge( body1, frame1, body2, frame2 );
```

Finally, to be realized, the constraint has to be added to the simulation.

```
  simulation->add( h.get() );
```

15.4 Ball Joint

A ball joint removes the three translational degrees of freedom for a constrained body.

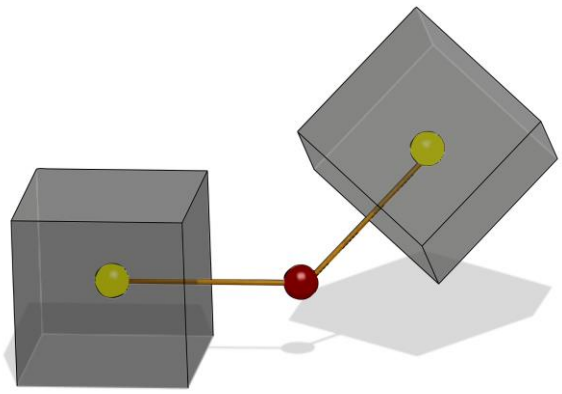


Figure 19: Illustration of a Ball joint where the red sphere is the ball joint center.

Ball joint example 1: Create a single body Ball joint (connected to world) at the origo of the rigid body:

```
// body1 is a rigid body
agx::FrameRef frame = new agx::Frame();
agx::BallJointRef b = new agx::BallJoint(body1  frame);
```

Ball joint example 2: Connect two bodies with a Ball joint

```
// body1 and body2 are rigid bodies
agx::FrameRef frame1 = new agx::Frame();
agx::FrameRef frame2 = new agx::Frame();

frame1->setLocalTranslate(-1,0,0);
frame2->setLocalTranslate(1,0,0);

// Create A BallJoint using the two frames
agx::BallJointRef b = new agx::BallJoint( body1, frame1, body2, frame2 );


simulation->add( b );
```

15.5 Universal joint

A universal joint transfer rotation via a central point through two rotation axes. The two rotation axes can be “bent” against each other. A universal joint is usually modelled with two hinges, however this requires an additional rigid body at the central point. The universal joint does not require this extra body. In addition the universal joint has motor, lock and range for the two remaining rotation dof’s.

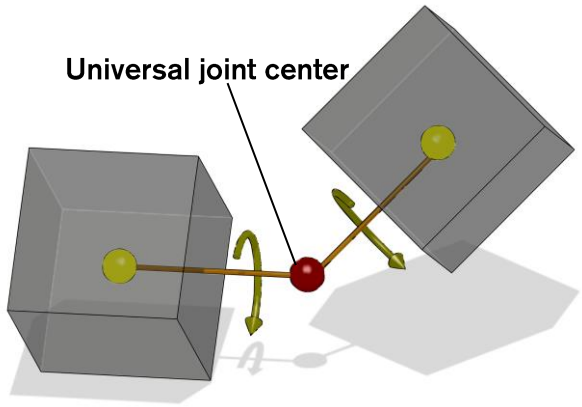


Figure 20: Illustration of a Universal joint where the red sphere is the universal joint center and the orange sticks are the rotational axes for each body, (rotations about this axis is equal for both bodies). The bodies are free to move around the universal center and rotation about the universal axis is constrained (equal rotation).

Universal joint example: Connect two bodies with a Universal joint

```
// Creating a universal joint, axis along world z.
agx::FrameRef rbFrame = new agx::Frame();
rbFrame->setTranslate(0,0,1.5);
agx::FrameRef rb2Frame = new agx::Frame();
rb2Frame->setTranslate(0,0,-1.5);
agx::UniversalJointRef universal = new agx::UniversalJoint( body1, rbFrame, body2, rb2Frame );
```

Then to realize the constraint it has to add the constraint into the simulation:

```
      simulation->add( universal );
```

15.5.1 Motor

A universal joint has two motors, one for each rotational axis which can be individually controlled:

```
// Enable the two motors
universal->getMotor1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_1 )->setEnable( true );
universal->getMotor1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_2 )->setEnable( true );

// Set a speed to the two motors
universal->getMotor1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_1 )->setSpeed( 0.5 );
universal->getMotor1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_2 )->setSpeed( 0.2 );
```

15.5.2 Range

A universal joint also has two ranges, one for each rotational axis:

```
// Enable the ranges
universal->getRange1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_1 )->setEnable( true );
universal->getRange1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_2 )->setEnable( true );
```

```
// Set the ranges
universal->getRange1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_1 )->setRange( -agx::PI*0.3,
agx::PI*0.3 );
universal->getRange1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_2 )->setRange( -agx::PI*0.3,
agx::PI*0.3 );
```

15.5.3 Lock

The universal joint also has a lock for each rotational axis:

```
// Enable the locks
universal->getLock1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_1 )->setEnable( true );
universal->getLock1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_2 )->setEnable( true );

// Set the lock positions
universal->getLock1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_1 )->setPosition(agx::PI);
universal->getLock1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_2 )->setPosition(agx::PI);
```

15.6 Prismatic universal joint

A prismatic universal joint is the same type of constraint as a universal joint but with an addition of a prismatic joint with its own motor, lock and range.

Constructing a prismatic universal joint is analogue to all other joints.

```
// Creating a universal prismatic joint, axis along world z.
agx::FrameRef rbFrame = new agx::Frame();
rbFrame->setTranslate(0,0,1.5);
agx::FrameRef rb2Frame = new agx::Frame();
rb2Frame->setTranslate(0,0,-1.5);
agx::PrismaticUniversalJointRef universal = new agx::PrismaticUniversalJoint(
                                                    body1, rbFrame, body2, rb2Frame );
```

Accessing the prismatic motor, range and lock is done through the same interface as for a universal joint:

```
// Enable the prismatic motor
universal->getMotor1D( agx::UniversalJoint::TRANSLATIONAL_CONTROLLER_1 )->setEnable( true );

// Set the force range and speed
universal->getMotor1D( agx::UniversalJoint::TRANSLATIONAL_CONTROLLER_1 )->setSpeed( 0.1 );
universal->getMotor1D( agx::UniversalJoint::TRANSLATIONAL_CONTROLLER_1 )->setForceRange(-
100,100);
```

15.7 Distance joint

Distance joints remove one translational degree of freedom for a constraint rigid body. A Distance joint can also be seen as a spring. It will try to restore the distance between two attached rigid bodies. (Or an attachment point in the world and one rigid body). A Distance joint also has a motor which can push constrained bodies away or pull them together. This can be used to simulate a hydraulic actuator…

Distance joint example1: Connect one body to world with a Distance joint

```
// body1 is a rigid body
// Given in body frame
// Position in the world where the attachment of the distance joint is
agx::Vec3 worldConnectPoint( 0.0, 0.0, 10.0 );
agx::FrameRef frame = new agx::Frame(); // Other end of joint is at origo of rigid body
agx::DistanceJointRef d = new agx::DistanceJoint(body1, frame ,worldConnectPoint );
```

Distance joint example 2: Connect two bodies to each other with a Distance joint and enable the motor:

```
// body1 and body2 are rigid bodies
agx::FrameRef frame1 = new agx::Frame()
frame1->setLocalTranslate(
agx::FrameRef frame2 = new agx::Frame()

agx::DistanceJointRef d = new agx::DistanceJoint(body1,frame1, body2, frame2);

// Disable the lock so that it will not interfere with the motor
d->getLock1D()->setEnable(false);

// Enable the motor and increase the distance between the two bodies with 0.1 m/s
d->getMotor1D()->setEnable(true);
d->getMotor1D()->setSpeed(0.1);

// set locked at zero speed, to avoid slipping when the distance is supposed to be constant
d->getMotor1D()->setLockedAtZeroSpeed(true);
```

15.8 LockJoint

A LockJoint will remove all six DOF from an attached body.

To lock a single body to the world or two bodies together it is possible to use the AGX Lock joint.

Lock joint example 1: Connect one body to the world with a Lock joint

```
// body1 is a rigid body
agx::LockJointRef l = new agx::LockJoint( body1 );

simulation->add( l.get() );
```

Lock joint example 2: Connect two bodies to each other with a Lock joint

```
// body1 and body2 are rigid bodies
body1->setPosition( agx::Vec3( 10.0, 0.0, 0.0 ) );
body2->setPosition( agx::Vec3( 5.0, 0.0, 0.0 ) );
agx::Frame frame1 = new agx::Frame;
frame1->setLocalTranslate(-2.5,0,0);
frame2->setLocalTranslate(2.5,0,0);

agx::LockJoint l = new agx::LockJoint(body1, frame1, body2, frame2);

simulation->add( l );
```

15.9 Prismatic

A prismatic constraint will remove 5 DOF from any attached rigid body. The body is free to translate along a specified axis. Compare to an elevator.

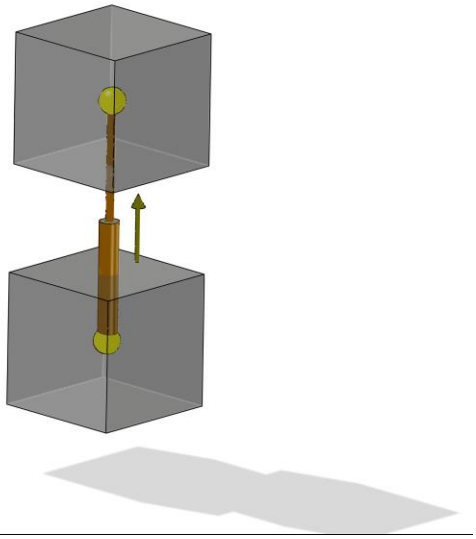


Figure 21: Illustration of a Prismatic joint. Bodies are free to translate along one axis. No rotation is allowed around this axis.

15.10 CylindricalJoint

A CylindricalJoint is a 2D constraint. It consists of a hinge *and* a prismatic. It means that two degrees of freedom are left for a constrained body, one rotational and one translational. Basically, the constrained body can translate along the specified axis as well as rotating around it. Both has a motor, range and lock.

Special for the cylindrical constraint is that it also has a Screw1D secondary constraint (more about secondary order constraints in 15.11).

Each of the motors, locks and ranges can be accessed through the call:

```
agx::Motor1D *angularMotor = cylindrical->getLock1D( agx::Constraint2DOF::FIRST );
agx::Motor1D *linearMotor = cylindrical->getLock1D( agx::Constraint2DOF::SECOND );
```

15.10.1 Screw1D

The Screw1D secondary order constraint connects the prismatic (linear) and the hinge (rotational) part of a CylindricalJoint.

If you try to translate the constrained body, it will also start to rotate. The ratio between the translational movement and the rotational is specified through the call to **setLead(agx::Real):**

```
// Create a Cylindrical joint (pseudo code)
agx::CylindricalJointRef cylindrical = new agx::CylindricalJoint(…);
// Enable the Screw1D
cylindrical->getScrew1D()->setEnable( true );

// For each revolution, the constrained body will translate 0.1 units.
cylindrical->setLead(0.1);
```

15.11 Constraint Forces

It is possible to read forces applied by a constraint. For more details, see 13.17.2.

15.12 Secondary order constraints

Secondary constraints are constraints on constraints such as limits and motors. The Secondary constraint will operate on the DOF which is not constrained. For example, a Hinge allows a body to rotate around one axis. The Secondary constraints of a Hinge will operate on that DOF.

Different constraints have different secondary order constraints. For example, a Hinge has *Lock1D*, *Motor1D* and *Range1D* secondary constraint while LockJoint does not have any secondary constraint at all. The secondary constraints can be requested through a call to the constraint API:

```
agx::Motor1D *motor = hinge->getMotor1D();
agx::Lock1D *lock = hinge->getLock1D();
agx::Range1D *range = hinge->getRange1D();
```

The CylindricalJoint (15.10) has one additional secondary constraint:

```
agx::Screw1D *screw1D = cylindrical->getScrew1D();
```

15.12.1 Motor1D

Motor1D is a controller which operates on the free degree of freedom for a constraint such as the rotational axis of a Hinge, the sliding axis of a Prismatic and or the rotational/sliding axis of a CylindricalJoint. A motor can be speed controlled either by specifying a speed or a force/torque, or both:

```
hinge->getMotor1D()->setSpeed( 1.2 ); // Set the desired speed in radians/sec
// Specify the available amount of torque in negative and positive direction
hinge->getMotor1D()->setForceRange( -1000, 1000 );  // Nm
```

A motor for a rotational constraint such as a hinge has the units of rotational speed and torque, whereas a linear constraint such as a prismatic uses linear distance and force.

A motor is a velocity constraint; it will apply a force/torque to reach the desired speed. This means that a Motor1D will not be able to keep a constrained system stationary over time. It will be subject to a drift in position. To keep a constraint at a specified position, the Lock1D secondary constraint should be used.

The Motor1D however has a utility method called **setLockedAtZeroSpeed()** which will automatically transform the Motor1D to a Lock1D when the desired speed is zero:

```
// Whenever the desired speed is set to 0, the motor1d will operate as a Lock1D
// The same force range will be used for locking the constrained bodies.
hinge->getMotor1D()-setLockedAtZeroSpeed( true );
```

15.12.2 Lock1D

Lock1D can be used to lock a constrained body to a specific position. A specified force/torque range can be given which will limit the max force/torque that the lock can apply to maintain the desired position.

A Lock1D can for example be used to specify a wounded spring:

```
// Specify a position of -100 radians, ie. a wounded up spring.
```

```
hinge->getLock1D()->setPosition( -100 );
hinge->getLock1D()->setCompliance( 1E-5 );
```

15.12.3 Range1D

Range1D defines a valid range for a constrained rigid body to move around/along the specified axis. At the limits of a range you effectively get a contact behavior. By default the limits are –inifinity, +infinity.

```
// Enable the range
hinge->getRange1D()->setEnable( true );
// Specify the range limits
hinge->getRange1D()->setRange( agx::RangeReal(-agx::PI, 10*agx::PI) );
```

The above code would limit the rotation of a body between PI and 10 PI radians.

15.12.4 Current Position

The secondary constraints has a current angle/position which can be queried with the call to getAngle(). For a hinge the call is:

```
// Create a hinge
agx::Constraint1DOFRef constraint = new agx::Hinge(… );

// Get the current angle in radians
agx::Real currentAngle = constraint->getAngle();
```

And for a prismatic constraint:

```
// Create a prismatic
agx::Constraint1DOFRef constraint = new agx::Prismatic(… );

// Same call but here we get the current position as it is a Prismatic constraint
agx::Real currentPosition = prismatic->getAngle();
```

Notice that for a hinge, the position is aware of winding. This means that the whole API for accessing/setting angles can accept values larger than 2*PI.

15.12.5 Current force

The current force applied by a secondary constraint can be accessed by a call to getCurrentForce, notice that the method is the same for both linear and angular constraints.

```
// Get the current torque applied by the hinge
agx::Real currentTorque = hinge->getMotor1D()->getCurrentForce();
```

For more details, see 13.17.2.

15.12.6 Regularization parameters

The regularization parameters can be used to model the effect of a Secondary constraint. For example, when a hinge constraint reaches a limit, what parameters are then used? Should it be a stiff limit? (Compliance reaches zero) or a softer limit (compliance> 0). Should the limit be fulfilled immediately, (damping=0), or will we allow violation for some time (damping > 0).

```
// Specify how stiff the limit should be:
hinge->getRange1D()->getRegularizationParameters()->setCompliance(1E-5);

// Specify the damping of the limit
hinge->getRange1D()->getRegularizationParameters()->setDamping(3.0/60);
```

15.13 Rebind

The geometry configuration of a constraint can be changed during runtime, the attachment frames can be modified, bodies can be moved relative to each other, however this requires a call to **Constraint::rebind()** which will use the new configuration as the initial configuration. The result is a relaxed system where the bodies are at their current/modified transformation.

Observe that explicitly transforming dynamic objects during a simulation can have unwanted results: instability, missing collision detection, large constraint violations etc.

15.14 Solve type

The solver allows a great flexibility when it comes to configuring constraints. A Constraint can be set to be solved in the iterative/direct/both solvers with the method:

```
Constraint::setSolveType( Constraint::SolveType solveType );
```

Solving a constraint in the iterative solver, means that it will have a lower maximum compliance, i.e. it will be softer/more springy. The direct solver is able to handle very stiff constraints. The benefit of solving constraints in the iterative solver, is commonly it is faster. So depending on your specific configuration of your system, some constraints can be iterative and some can be direct.

It is however important that constraints that *connects* two subsystem where one is solved with the iterative and the other with the direct solver is set to DIRECT_AND_ITERATIVE. This is necessary to transfer the correct forces between the two subsystems. If not, the system can become unstable.

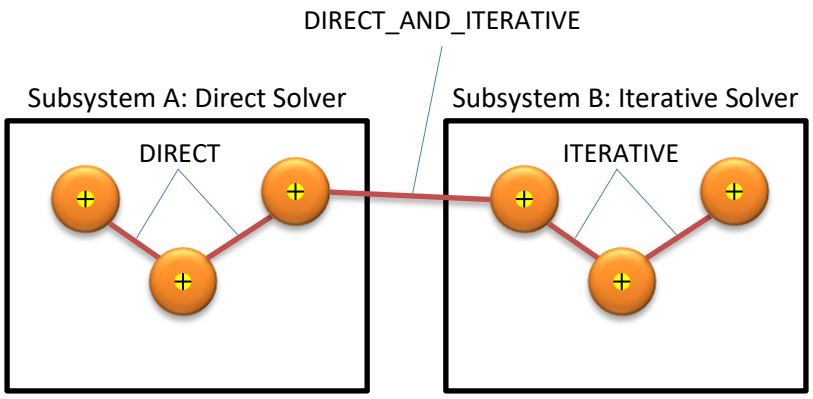


Figure 22: Coupling between subsystems. (Red lines indicate constraints between bodies)

15.15 Constraint utility methods

The class Constraint also has a number of utility methods for creating constraint configurations:

Given two rigid bodies, a point in the world and an axis, the method below will compute the two attachment frames to create a relaxed system:

```
agx::Vec3 worldPoint(0,0,0); // Anchor point for the constraint
agx::Vec3 worldAxis(agx::Vec3::Y_AXIS()); // Axis for the constraint

agx::Constraint::calculateFramesFromWorld(worldPoint, worldAxis, body1, frame1, body2, frame2
);

// Create a hinge with the rotation axis along Y
agx::HingeRef hinge = new agx::Hinge(body1, frame1, body2, frame2);
```

16 Particle systems

16.1 Basic usage

There are two types of particle systems available in AGX: RigidParticleSystem and ConstraintFluid. The RigidParticleSystem simulates a collection of hard spheres that are held apart by contact constraint solved by a Gauss-Seidel solver. The ConstraintFluid simulates a fluid where each particle samples a density field and the particles are held apart by density constraints solved by a conjugate-gradient solver. There are two-way interactions between the rigid particles and other parts of the simulation, but the constraint fluid currently only supports static geometries in the scene.

A particle system can be created either from C++ code or from a Lua scene script, but the setup procedure is very similar. Examples in code are given below for everything that is described in the following in both C++ and Lua code.

The first step is to create an instance of the particle system class in question, the RigidParticleSystem class for the hard sphere particles and ConstraintFluid for the density constrained fluid. The second step is to create a set of particles to simulate. They can either be allocated and initialized using the particle API, or created in a regular grid using the spawnParticlesInBound method available in the ParticleSystem base class.

Regardless of how the particles were created, individual particles can be accessed and manipulated using ParticlePtrs. A ParticlePtr is an accessor object that gives access to the individual attributes of a particle, stored as elements in the buffers that make up the EntityStorage that the particle in question is part of. The ParticlePtr exposes one accessor method for each attribute of the Particle entity.

- position
- velocity
- force
- life

16.1.1 Particle systems in C++

The following code snippet demonstrates how to create a simple rigid particle system in C++.

```
// Create and configure a rigid particle system.
ParticleSystemRef particleSystem = new RigidParticleSystem();
simulation->add( particleSystem );
particleSystem->setParticleMass( Real(10.0) );

// Create a single particle.
agx::Physics::ParticlePtr particle = particleSystem->createParticle();
particle.position() = Vec3(0, 0, 4.5);

// Create a bunch of particles in a bound.
agx::Bound3 spawnBound = agx::Bound3( Vec3(0.0, 0.0, 0), agx::Vec3(4.0, 1.0, 4.0) );
particleSystem->spawnParticlesInBound( spawnBound, agx::Vec3f(0.1) );
```

16.1.2 Particle systems in Lua

The following code snippet demonstrates how to create a simple rigid particle system in Lua.

```
-- Create and configure a particle system.
local particleSystem = agx.RigidParticleSystem()
sim:add( particleSystem )
particleSystem:setParticleMass( 10 )

-- Create a single particle.
local particle = particleSystem:createParticle()
particle:position():set( agx.Vec3(0, 0, 4.5) )

-- Create a bunch of particles in a bound.
local spawnBound = agx.Bound3( agx.Vec3(0, 0, 0), agx.Vec3(4,1,4) )
particleSystem:spawnParticlesInBound( spawnBound, agx.Vec3f(0.1) )
```

16.1.3 Particle emitter

The particle system comes with a collection of tasks designed to be part of a particle system update. One such task is the particle emitter, which creates new particles at a certain rate within a specified spawn volume. This task has an explicit Lua binding and can therefore be created directly from Lua code. The volume in which particles may be created is defined by a geometry and a method is used to set the volume.

```
local emitter = agx.ParticleEmitter( )
emitter:setVolume( agxCollide.Geometry( agxCollide.Cylinder(1.6, 10) ) )
simulation:add( emitter );
```

The emitter contains a parameter to control the rate of the particle creation, in particles per second. This parameter can be set from Lua using the following code snippet:

```
emitter:setRate( 15 )
```

To set the initial “life” of an emitted particle:

```
emitter:setLife( 10 );
```

17 agxWire: Simulating wires

The namespace agxWire contains numerous classes for simulating wires, ropes and chains. These classes can be used to simulate cranes, winches, pulleys etc. The wire is an implementation of a lumped element structure with dynamic resolution. The main difference between classical lumped elements, such as chains of bodies and ball socket joints (or hinges), is that the wire will adapt the resolution so that no unwanted vibrations will occur. Whenever a wire slides over a geometry (such as a box, cylinder or trimesh), it will create shape contact nodes, which will stabilize contacts even under very high tension. The Wire is stable even under very high mass ratios/tension. A weight of several thousand tons can be held with a wire where each mass element weighs 100kg. This is necessary to be able to model real life scenarios with wires, chains, ropes etc.

Wire-wire interaction can also be enabled (not enabled by default). The overlap test between wire-wire is performed with sweep tests (continuous collision detection) to reduce the risk for accidental missed overlaps.

17.1 Known limitations

The current limitations of the wire are:

1. A wire cannot handle torsion. However, rotation for Link's between Wire's can be controlled. See 18.6

17.2 agxWire::Wire

agxWire::Wire (wire) has one material for the whole wire. Operations on a wire are: cut, reverse, merge. It can be winched using the class agxWire::WireWinchController.

Below is a table of classes in the agxWire namespace.

17.3 Modelling primitives

Class	Description
agxWire::Wire	Class for simulating a wire of a single segment type.
agxWire::WireWinchController	A winch to be used for the agx::Wire class.
agxWire::WireSimpleDrumController	A drum implementation for use on the agx::Wire class.

Table 15: Classes related to wire simulation.

17.4 Wire nodes

Class	Description
agxWire::Node	Base class for a node used when routing an agxWire::Wire. A wire is forced to pass through this specified point.
agxWire::FreeNode	Nodes used for routing a wire at points in the world coordinate system.
agxWire::BodyFixedNode	Nodes used at end of wires to attach rigid bodies.
agxWire::EyeNode	A node which defines a "hole" through which the wire can slide. A Link can not slide through an EyeNode.

agxWire::ContactNode	OBSOLETE! A node which is either created by the user, or by the collision detection system to any edge where the line is overlapping geometries. Can only exist on geometries with single shapes. The shape can only be box, cylinder or some type of mesh.
agxWire::ShapeContactNode	A node which is either created by the user, or by the collision detection system to any edge where the line is overlapping geometries. No limitation on on the number of shapes for a geometry. Can exist on all shapes, expect lines, rays and WireShapes.
agxWire::ConnectingNode	A node commonly used to attach a rigid body between two agxWire::Wire. This node has additional stabilization constraints.

Table 16: Node classes used for routing wires.

17.5 Rendering/Storage classes

Class	Description
agxOSG::WireRenderer	Class for rendering an agxWire::Wire in OpenSceneGraph
agxWire::NodeConstIterator	Iterator for accessing nodes in a Wire.

Table 17: Classes used for rendering/Storage.

17.6 Overall description of a Wire

17.6.1 Dynamic Resolution

A wire's route is defined by its *nodes*. There are various nodes for different purposes. Nodes are explained later in the document.

A very important concept of a Wire is the ability to change resolution of nodes per length unit. A Wire with a certain material (radius/density) will have a certain mass. If this mass is distributed along the wire in many nodes (lumped elements), we will get small masses in each node. If a Wire with many small masses is affected by a large tension, we will get oscillations due to the large ration between tension and mass of each mass element. Therefore, the Wire will automatically remove nodes during high tension.

The maximum resolution is specified for a Wire is specified in its constructor

```
agx::Real resolution = 2; // Two lumped elements per meter is the max resolution.
agxWire::WireRef wire = new agxWire::Wire(
                  0.01,          // Radius
                  resolution ); // Number of elements per length unit
```

or through the call:

```
void agxWire::setResolutionPerUnitLength( agx::Real resolutionPerUnitLength );
```

17.6.2 Constraint model of a Wire

A wire consists of a set of bodies and constraints. From the attributes in the agxWire::Wire class, a structure is created which contain the physical model of a wire.

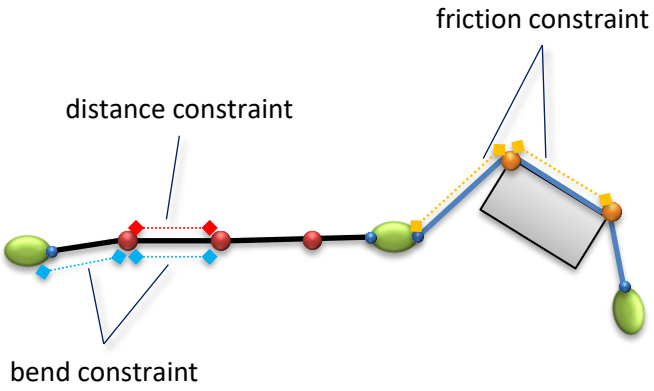


Figure 23: The various constraints involved in the modelling of a Wire.

Figure 23 illustrates the schematic structure of a simulated Wire. The constraints involved are:

Constraint model	Description
Distance constraint	A distance constraint will try to keep the two bodies locked at a specified distance. There are distance constraints between all nodes (bodies) in a wire.
Bend constraint	A bend constraint is a three body constraint that will try to restore the wire into a straight line. A Wire with a large radius is of course much more resistant against bending than a thin rope. A bend constraint also increases stability for a Wire, especially for Links.
Friction constraints	Whenever a Wire collides with the edge of another geometry, a shape contact node is created. Depending on the material attributes (friction coefficient) a friction constraint will try to work against moving the wire over the edge, both along the edge and orthogonal to it.

Table 18: Constraints involved in a wire simulation.

17.6.3 Geometry model of a Wire

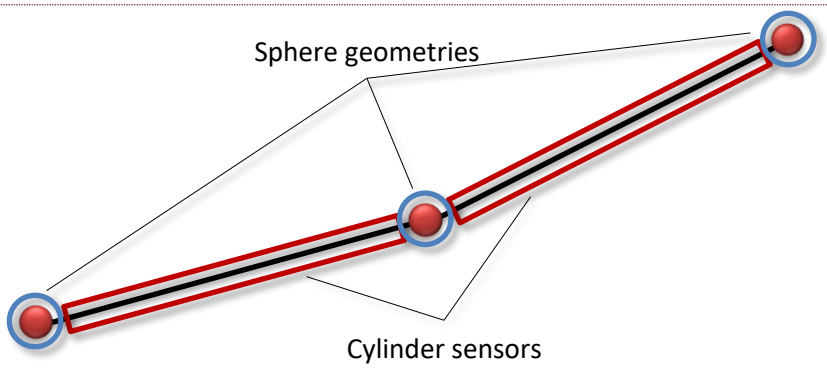


Figure 24: Geometries associated to a Wire.

To make the wire interact with rest of the simulation, it is also consisting of geometries. The geometries cause the wire to be able to collide with other geometries. A wire consists of two types of geometries:

1. **Sphere geometries**: Each lumped element (node with a rigid body) will be represented as a sphere (blue in above figure), with a radius derived from the radius of the wire/segment. This geometry collides with other geometry and creates contacts which the solver will see.
2. **Cylinder geometries**: Between each lumped element (node), a cylinder (red in the above figure) will be kinematically transformed between time steps. This cylinder is used for detecting contacts between the Wire and edges on other geometries. There are no contacts generated between this cylinder and other geometries that the solver will see.

Similar to an ordinary agxCollide::Geometry, the API for the wire has a number of methods to control which other geometries can collide with the geometries in the wire:

```
void setEnableCollisions(const agx::RigidBody* body, bool flag);
void setEnableCollisions(const agxCollide::Geometry* geometry, bool flag);
void addGroup(unsigned id);
```

17.6.4 Operations on a Wire

A wire is created in a *routing* process described below. When a wire is created it can be merged/cut/split/reversed for more information see chapter *17.11 Wire operations.*

17.6.5 Devices used with wires

A wire can be routed together with various devices, for example a *WireWinchController* and *WireSimpleDrumController.* A winch is a device which resembles a real life winch. It is capable of pulling in and feeding out wire just as a winch would. The mass of the wire that is pulled into a winch will not be simulated.

17.7 agxWire::Node

A Node is used for routing a wire and to attach a wire to a certain RigidBody. A Node defines a point through which a wire is forced to pass. This means that a Node attached to a RigidBody will give a 1-1 interaction between the RigidBody and the wire.

The nodes are stored in an ordered list. If a wire hasn't been cut or detached, the first and last nodes are the ones the user used when creating the wire.

BodyFixed nodes (or lumped element nodes) are nodes that will appear at a certain fix distance from each other or a multiple of that distance. Each body fixed node is placed in the center of mass of a rigid body, which carry some of the mass of the wire. The dynamic resolution makes body fixed nodes (and their bodies) disappear and appear as the tension increases and decreases to keep the simulation numerically stable. A distance constraint is only defined between two body fixed nodes.

Note: Body fixed nodes can also be used as fixed connection points on bodies. They can be located at arbitrary relative positions to that body.

Note: If collisions between the rigid body and the wire are enabled, it is important to place the BodyFixedNode so that the whole wire is initially non-overlapping. This includes especially one extra wire radius around the attachment point. So:

```
  agx::RigidBodyRef body = new agx::RigidBody(new agxCollide::Geometry(new
agxCollide::Box(agx::Vec3(1, 1, 1))));
  // ...
  agxWire::WireRef wire = new agxWire::Wire(0.1, 10);
  wire->add(new agxWire::BodyFixedNode(body, agx::Vec3(1 + 0.1, 0, 0))); // Note the extra 0.1
```

Eye nodes are nodes that can never start or end a wire. They consist of two points that both are fixed relative a rigid body which is NOT part of the wire. Their purpose is then to make the rigid body slide along the wire.

ShapeContact nodes work as eye nodes except for two major differences: they come and go as the wire collides and slips off geometries and they slide (with friction) along edges (see 17.8.1Wire Friction) of shapes.

Free nodes (could also be called *temporary lumped nodes*) are used for routing. They are kept as long as the wire is numerically stable. If the actual position of a free node along the wire will give us stability problems but still is considered necessary to keep to retain the orientation and most of the energy, the node may travel along the wire to a position where the tension could be handled by the solver. If the free node isn't necessary to keep it is removed forever. When a contact between the wire and geometry is supposed to be removed, the contact node is transformed into a FreeNode instead, to preserve the wire orientation.

17.7.1 agxWire::BodyFixedNode

BodyFixedNode is a node that attaches a wire to a point *relative* to a RigidBody.

```
// Arguments: the body and an attachment point relative the body
agx::BodyFixedNodeRef bf = new agxWire::BodyFixedNode( body, Vec3( 1,0,1 ));
```

17.7.2 agxWire::EyeNode

EyeNode is a node that can slide along a wire. It has two friction parameters, in two separate directions. *Positive* is specifying friction when a body is sliding along the wire towards the end of the wire, *Negative,* is when a body is sliding towards the beginning of a wire. The beginning of a wire is determined as the first point of routing.

An EyeNode can also specify a point and a normal, which makes it possible to attach a RigidBody and make it rotate as the wire moves and vice versa.

```
agxWire::EyeNodeRef eye = new agxWire::EyeNode( slidingObject.getRigidBody(),
                          Vec3( 1,0,1 ),    // First point relative to body.
                          Vec3( -2,0,0 ) ); // Vector pointing from first point
                                            // To the second point
```

The third argument is a vector pointing out the direction and distance to the second point of attachment between the wire and the rigid body.

An EyeNode can also have a radius which determines whether a Link with a specified radius can pass through or not.

To specify the friction of the wire sliding through the eye, use the wire material:

```
eyeNode->getMaterial()->setFrictionCoefficient( 0.5 );
```

It is also possible to specify different friction in different directions:

```
eyeNode->getMaterial()->setFrictionCoefficient( 0.5, agxWire::NodeMaterial::NEGATIVE );
eyeNode->getMaterial()->setFrictionCoefficient( 0.5, agxWire::NodeMaterial::POSITIVE );
```

17.7.3 agxWire::FreeNode

FreeNode are nodes that can be used during the routing procedure of a wire. They are nodes that will disappear when the tension increases in the wire.

```
wire->add( new agx::FreeNode( Vec3( 52,2,10 ) );  // Position in world coordinates
```

17.8 Wire Material

The class agxWire::Wire stores an instance of an agx::Material. This material is used for two things:

Bulk attributes: Density is used to calculate the internal material properties for a wire. Whenever this material is updated with new properties, the physical properties are recalculated.

Surface attributes: The SurfaceMaterial part is used during contact processing to calculate restitution and friction like any other geometry collision.

It is possible at any point in time to change the material of a segment:

```
const agx::Real resolution = 1.0;
const agx::Real radius = 0.01;
// Create a material
agx::MaterialRef wire_material = new agx::Material("wire_material");
simulation->add( wire_material );

agxWire::WireRef wire1 = new agxWire::Wire( radius, resolution )

// Associate the new material to this Segment
wire1->setMaterial( wire_material );

// Set properties
wire_material->getWireMaterial()->setYoungsModulusBend( 1E11 );
wire_material->getSurfaceMaterial()->setRoughness( 0.2 ); // Friction
```

It also possible to use the wire material when creating *explicit* ContactMaterials:

```
// Create an explicit contact material for this wire_material colliding
// with another material
agx::ContactMaterialRef wire_plastic_material = new agx::ContactMaterial( wire_material,
plastic_material );

// Set the attributes of the explicit material
wire_plastic_material->setFrictionCoefficient( 0.5 );

simulation->add( wire_plastic_material );
```

The material used for a Wire is internally modified (for point friction) which makes it unsuitable for use with other geometries in a simulation.

Avoid using the same material for an agxWire::Wire as other agxCollide::Geometries.

17.8.1 Wire Friction

A wire can interact with other geometries in two ways:

1. A wire segment intersects a geometry and a ShapeContactNode is inserted, with an edge to move along. This when using the kinematic wire contact algorithm, which solve the dynamics, including friction, of shape contact nodes in a co-simulation.
2. A wire collides on the surface of another geometry. Either a rigid wire segment when using the dynamic wire contact algorithm, described in section 17.16.2. or a BodyFixedNode.

Wire friction for BodyFixedNode's and for the DynamicWireContact is defined as friction for any two geometries colliding, see section 13.14 about materials.

Wire friction for a ShapeContactNode is defined both along the wire and along an edge defined orthogonal to the wire direction. See Figure 25 for an illustration. When the *wire* (iii) is in *contact* (v) with a *geometry* (iv) and sliding along its *edge* (vii) we find a *friction force* (ii) along the edge. If the *normal force* (i) is small enough we see sliding friction, otherwise the contact will not move. The algorithm for defining edges for a ShapeContactNode make use of the wire direction and the local curvature of the shape the wire is in contact with.

To distinguish friction along the wire and along the edge for a ShapeContactNode primary (along the wire) and secondary (along the edge) friction direction is specified to set them differently. By default the wire friction coefficients are the same as defined for the contact material for a wire-geometry contact. Creating an explicit contact material the default wire friction coefficients can be overridden by setting the wire friction coefficient.

```
// Create an explicit contact material for this wire_material colliding
// with another material
agx::ContactMaterialRef wire_plastic_material = new agx::ContactMaterial( wire_material,
plastic_material );

// Set the wire friction coefficient along the wire
wire_plastic_material->setWireFrictionCoefficient( 0.5, ContactMaterial.PRIMARY_DIRECTION );
// Set the wire friction coefficient along the shape contact edge
wire_plastic_material->setWireFrictionCoefficient( 0.1, ContactMaterial.SECONDARY_DIRECTION);

simulation->add( wire_plastic_material );
```

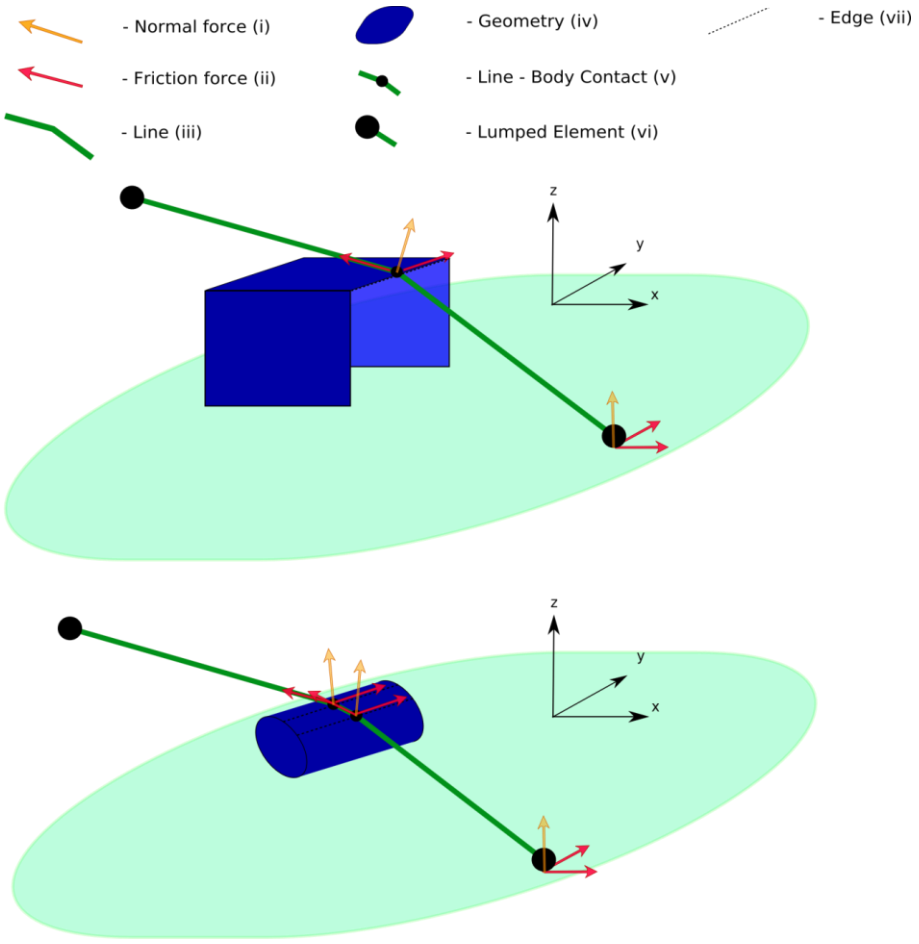


Figure 25: Illustration of wire friction.

17.9 Creating a agxWire::Wire

Below is a description of the process of creating a wire.

17.9.1 Routing a wire

A wire is placed into the world by a *routing process*. During the routing, winch controllers and nodes of various types are created and positioned attached to rigid bodies. The nodes are then added to the wire.

During wire routing, the route nodes added are stored locally in the wire. I.e. no communication occurs with the constraints. The wire has not been added to a simulation at this point. The routing ends when the wire is added to the simulation.

This means that adding a wire to the simulation should be done *after* all the routing is done.

When the first node is added, a check for a possible winch controller at the beginning of the wire is done. If one is present, and it for example does not have auto-feed enabled, we have to take care of the rolled in length and count that to the total routed length.

For all nodes added after the first one, we calculate the current route length. If the route length does not exceed the total length of the wire, we add the node to the local list of routed nodes.

17.9.1.1 Process for routing a wire

1. Initialization
    a. create all bodies (involved in routing)
    b. create WinchControllers at relative position to bodies (or in the world)
    c. create eyes, at positions relative to bodies
    d. Position bodies at their correct positions.
2. Routing:
    a. Route beginning of wire either through a winch, a BodyFixedNode, a ConnectingNode or a FreeNode.
    b. Route through WinchController (*wire->add()*)
    c. route through eyes (*wire->add()*)
    d. route through final WinchController() (*wire->add()*) *or*
    e. You always have to end the route with a WinchController, BodyFixedNode, ConnectingNode or a FreeNode.
3. Add wire to simulation
4. Simulate.

It is always important to remember to always position the rigid bodies/Nodes *before* the routing process starts. This is the same requirement as when configuring other constraints in AGX.

17.9.2 Add to simulation

When a wire is added to the simulation it will be assumed that the routing is finalized.

Starting at the end we check for winch controllers, if present, one of the following scenarios may occur:

1. **Winch at both ends, both with auto-feed enabled.**
   Pulled in length for the end winch is set to zero and all superfluous wire is moved to the first winch.

2. **Winch at both ends, first with auto-feed enabled and second with auto-feed disabled.**
   If this is the case, errors (warning messages, return false and ignoring the last winch during routing) occurs if the wire outside the winch is not long enough. Other than that this is a well-defined case.

3. **Winches at both ends, first with auto-feed disabled and second with auto-feed enabled.**
   This means that auto-feed flag is used wrong. The purpose of the flag is to pull in all wire inside the first winch and then route while the length inside the first winch is changed during routing. If this case appears, all superfluous wire is put in the first winch, the one that does not have auto-feed enabled.

17.10 Rendering a agxWire::Wire

Rendering a wire is done using *RenderIterators*. Basically, a RenderIterator contains enough information to get a world position and tension of a control point in the wire. The code for agxOSG::WireRenderer can be found in <agx-dir>/agxOSG/src/WireRenderer.cpp,.h

```
agx::WireRef wire = createAWire();
simulation->add(wire);
simulation->stepForward();      // Take a time step
agx::Real refMaxTension = 1E4; // Just to get some reference for setting color/weight of a
                               // controlpoint in the spline

// Create a graphical representation of a spline in your rendering engine
Rendering::Spline *spline = new Rendering::Spline();

agxWire::RenderIterator it        = wire->getRenderBeginIterator();
const agxWire::RenderIterator end = wire->getRenderEndIterator();

// Loop through all the nodes in the wire
while ( it != end ) {
  const agxWire::Node* node = *it;

  // We can also read tension at the node to scale, control the rendering
  spline->add( node->getWorldPosition(), Real( 0.9 ) );
  ++it;
}
```

17.11 Wire operations

17.11.1 Cut a wire

To cut a wire there are two methods available:

```
// Cut a wire at the specified wire length
agxWire::Wire* agxWire::Wire::cut( const agx::Real& length , size_t minimumResolution, bool
includeWinchPulledInLength = true );

// Cut the wire at the closest point one the wire
// from the specified position
agxWire::Wire* Wire::cut( const agx::Vec3& position, size_t minimumResolution );
```

Both methods return a pointer to the second part of the cut wire. The wire returned is already inserted into the simulation and is ready to use. The two wires does not share any information, so they can be removed independently from the simulation.

17.11.2 Merge two wires

Two wires can be merged into one using the *merge* method:

```
bool agxWire::Wire::merge( agxWire::Wire* wireToMergeWith );
```

An example:

```
agxWire::WireRef a = createAWire();
agxWire::WireRef b = createAWire();

a->merge( b );
```

In the above example b will be added to the end of a.
This method will only work if the last node in a and the first node in b are FreeNodes.

To be able to merge two wires wire *A* and wire *B*, the last node of *wire A* and the first node of *wire B* have to be FreeNodes.

If the two end points to be merged are not at the same position, additional *wire will be added to the last wire segment as a straight wire between the end of the first and beginning of the first*. This means that the resulting wire length after a merge is always >= length1+length2

The wire given as an argument to merge() will be deleted from the simulation. If no references are holding it, it will be deallocated.

17.12 agxWire::WinchController

agxWire::WireWinchController is an implementation of a winch. It can pull in and winch out wire. It is based on a constraint (Prismatic) which means that it applies forces on a wire just as any other constraint or RigidBody in the system. It has functionality to haul in Links (which it then automatically disables when it is winched in the winch). A Winch can only be routed to the beginning or the end of a wire.

A WireWinchController is always attached to a RigidBody:

```
agxWire::WireWinchControllerRef winch = new agxWire::WireWinchController(
                  rigidBody,              // RigidBody onto which sd is attached
                  agx::Vec3( 0, 0, -1 ),  // Position relative to rigidBody
                  agx::Vec3( 0, 0.0, -1); // Normal direction
```

A winch can be told to pull in any “loose” wire so that the wire is stretched after the routing:

```
winch->setAutoFeed( true );
```

Routing a wire to the winch is done by adding the winch to the wire:

```
// Add the winch to the wire, the second argument is the amount of wire pulled
// in to the winch at start.
wire->add( winch, wireLength );
```

The above call will result in that the winch will pull in all the resulting wire after it is routed.

A WireWinchController can be detached from a wire with a call to:

```
wire->detach(bool front);
```

where *front* has to be set to *true* if the WireWinchController is in the front end of the wire, *false* for the back.

The winching mechanism of a winch can be activated and controlled through the following API:

Set the maximum force for holding the wire (keeping velocity at 0):

```
void Winch::setBrakeForceRange( agx::Real brakeForce );
```

Set the desired winch in/out speed:

```
void Winch::setSpeed( agx::Real speed );
```

Set the maximum force by which the desired speed will be obtained:

```
void Winch::setForceRange( agx::Real winchForce );
```

17.13 Wire-wire interaction

Wires can also collide with other wires. This functionality has to be explicitly enabled between pairs of wires that are supposed to be able to interact. The class **agxWire::WireController** controls this behavior.

To enable wire-wire interaction between two instances of agx::Wire:

```
agxWire::WireRef wire1 = createWire(); // Create a wire
agxWire::WireRef wire2 = createWire(); // Create a wire

// Enable wire-wire collisions between the two wires
agxWire::WireController::instance()->setEnableCollisions(wire1, wire2, true);
```

The method `void setEnableAllWireWireCollisions( bool enable );` is for internal purposes only and should not be used.

For stability of wire-wire interaction, the coefficient of restitution for the contact material between the two wires should be set to 0.

```
auto* cm = simulation->getMaterialManager()->getOrCreateContactMaterial(wire1->getMaterial(),
wire2->getMaterial()); // Obtain the contact material

cm->setRestitution(0);
```

Also, the value of YoungsModulusBend for material of the respective wires should be set to realistic values.

```
wire->getMaterial()->getWireMaterial()->setYoungsModulusBend(1E9);
```

If a chain should be simulated instead of a relatively stiff wire, this can be achieved by turning off the bend resistance altogether, instead of setting it to zero:

```
wire->getMaterialController()->setIsBendResistant(false);
```

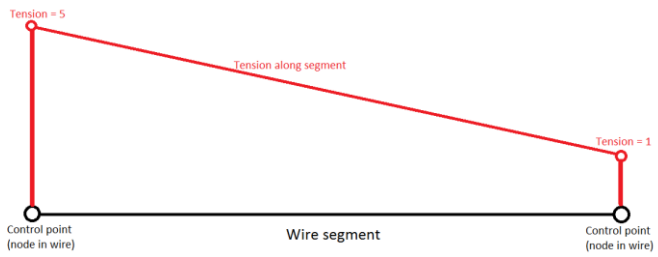
17.14 Reading tension/forces

After a time step (a call to simulation->stepForward()), the tension involving the wire is updated and available.

The tension at each node along the wire can be read through the specified node in the wire or through a point in space or a length along the wire.

17.14.1 Get tension given distance along wire or point in the world

The tension can be queried given a point in space, or a length from start along the wire. These two methods will do a linear search to find the correct point along the wire. The data will be returned into a struct: agxWire::WireSegmentTensionData containing various data. The data will be linearly interpolated between the nodes in the wire:

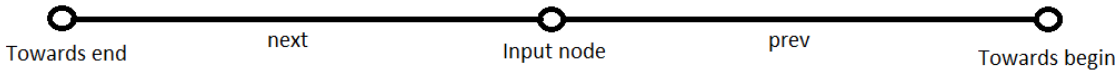



```
// Closest point to the wire will be used
double tension = wire->getTension( agx::Vec3(1,1,1) ).getAverageRaw();
```

```
// Get the tension in the wire a certain length along the wire
double tension = wire->getTension( 123 ).getAverageSmoothed(); // 123 m from start
```

17.14.1.1 Get tension given node

The node tension data, `agxWire::WireNodeTensionData`, contains data for two segments - the segment before and the segment after the node. I.e., the tension exactly before and exactly after the node.

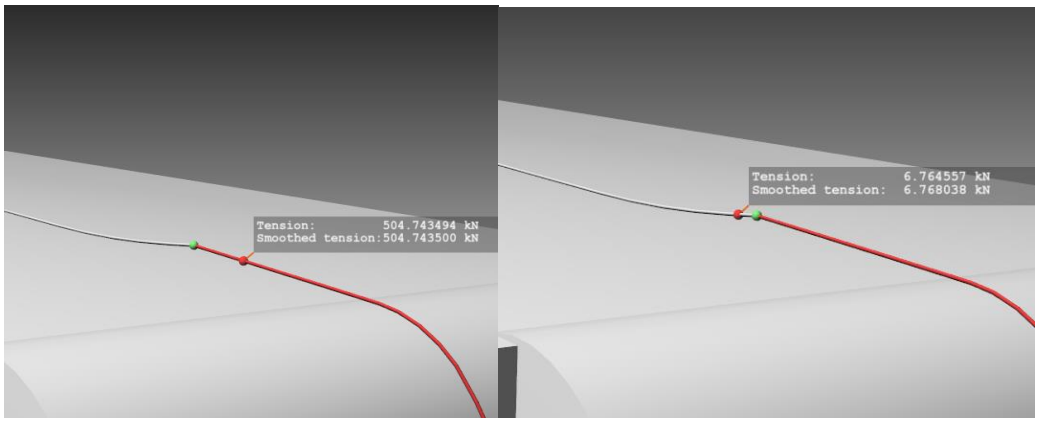



```
for ( agxWire::RenderIterator i = wire->getRenderBeginIterator(); i != wire-
>getRenderEndIterator(); ++i )
{
  double tension = 0;
  agxWire::WireNodeTensionData data = wire->getTension( *i ); // Get tension at the node
  tension = data.getAverageSmoothed();
}
```

In comparison to “get tension given distance along wire”, fetching data given node is computationally much cheaper.

17.14.1.2 Tension at nodes

The tension along a wire isn’t continuous - so beware of, and think of, what it means to read tension given nodes. Consider a closed Shark Jaw, as the two pictures below, modeled with an eye node (green sphere) that has infinite friction.



In this example, tension drops instantaneously from 504 kN to 7 kN over the eye node making it hard to interpret the exact tension value at the node. By definition (internally) the node at the end of a segment is carrying the tension data, i.e., if one ask the node what tension it feels it can either be 504 kN or 7 kN depending on in which way the wire is routed (or after reverse). It’s important to be consistent and think twice which value one expects in this special case.

Of course it becomes more convenient with our data holder, `agxWire::WireNodeTensionData`, which contains data for the segment before and after the node. But following the above

definition, `agxWire::WireNodeTensionData::getRaw()`, returns the raw tension value from the segment before the node.

17.14.2 Normal force between a geometry and the wire

To get the normal force between a specified geometry and a wire you first have to locate a shape contact node in the wire which is interacting with the specific geometry, then you can query the node's contact material for the normal force magnitude.

```
for ( agxWire::RenderIterator i = wire->getRenderBeginIterator(); i != wire-
>getRenderEndIterator(); ++i )
{
  double magnitude = 0;
  // Tension is only available at body fixed nodes
  agxWire::ShapeContactNode *cn = i.get()->getAsShapeContactContact();
  if (cn)
    magnitude = cn->getMaterial()->getNormalForceMagnitude();
}
```

17.15 Applying forces to a wire

If you want to simulate for example drag in water, wind, buoyancy or other external forces on a wire, you can do this with a *ForceField*. A ForceField will be updated inside the solve stage. This means that the callback in a user ForceField must never make any structural changes to the system. The only allowed operations is applying torque/forces onto rigid bodies, all other operations (except reading data) should be strictly avoided.

Below is a code snipped that creates a custom ForceField which applies forces to each rigid body of a wire.

```
// Derive from baseclass agx::ForceField
class MyForceField : public agx::ForceField
{
  public:
    MyForceField( agxWire::WireRef wire )
      : m_wire( wire ) {}

    // Virtual method called from within the solver
    virtual void updateForce( agx::DynamicsSystem* ) AGX_OVERRIDE
    {
      if ( m_wire == 0L )
        return;

      // Iterate over all nodes in the wire
      agxWire::RenderIterator it = m_wire->getRenderBeginIterator();
      while ( it != m_wire->getRenderEndIterator() ) {
        agxWire::Node* node = *it;
        // Check for internal, dynamic, mass nodes.
        if ( node->getType() == agxWire::Node::BODY_FIXED )
          node->getRigidBody()->addForce( calculateForce( node ) ); // Apply the force

        ++it;
      }
    }

  protected:

    // Some method for calculating a force
    // Could be calculating gravity, windforce, buyancy or something else
    agx::Vec3 calculateForce( const agxWire::Node* node ) const
    {
      return agx::Vec3( agx::PI );
```

```
    }

  protected:
    agx::observer_ptr< agxWire::Wire > m_wire;
};

...
// Add the custom force field to a simulation
simulation->add( new MyForceField( wire ) );
```

17.16 Wire collisions and contacts

Up until this point the described agxWire::Wire contact behavior assumes that the default wire contact model is used. This model is called *"default kinematic contact model"*. There are two other models to consider, resulting in improved interaction fidelity.

- ***Default kinematic contact model* – Default model with unconditionally stable behavior**
- ***Dynamics wire contact model* – Most dynamic/realistic behavior**

17.16.1 Default kinematic contact model

One of the features of a kinematic model is that it is unconditionally stable. Wire collisions result in kinematic nodes (agxWire::ShapeContactNode) on the collided geometry.

To achieve highest performance combined with stability under high tension the *default kinematic contact model* is recommended.

When modeling of complex scenes with complex geometry wire collision can be computationally expensive. Collision between wires and complex geometry is not recommended when high performance is prioritized. Instead specific wire collision geometries are recommended to be used. I.e. position geometries containing one box or one cylinder to represent the complex geometry for the wire collisions (low interaction fidelity).

17.16.1.1 Wire kinematic model for AGX versions previous 2.19

For AGX versions previous 2.19 the kinematic contact model generated agxWire::ContactNodes at collision. The ContactNode is now deprecated and replaced with the ShapeContactNode which handle all shapes and has no restrictions on the number of shapes per geometry. New from 2.19 is that the wire has a class called WireShapeContactController replacing a previous class called WireContactController.

Restoring a wire stored from a version of AGX lower than 2.19 could include ContactNodes. The restored wire by default use the old wire contact controller. All contact nodes can be converted to shape contact nodes for a specific wire. For the wire to generate shape contact nodes from future interactions, the contact controller type must be changed from *OLD_CONTACT_CONTROLLER* to *SHAPE_CONTACT_CONTROLLER*.

```
// Saves simulation including a wire
agxStream::InputArchive& storedScene;
// Wire object to restore
agxWire::Wire* wire;

// Choose to use the shape wire contact controller,
// for shape constact node generation at collision
wire->setActiveContactControllerType(agxWire::SHAPE_CONTACT_CONTROLLER);

//convert all contact nodes to shape contact nodes
wire-replaceContactNodesWithShapeContacts();
```

17.16.2 Dynamic wire contact model

For a more realistic/fully dynamic behavior of wire/geometry interaction is required the *Dynamic wire contact model* is recommended.

This model introduces new bodies, as part of the wire, wherever the wire has collided. It is therefore not expected to demonstrate the same stability under high tension as the kinematic model.

The dynamic wire contact model is enabled per geometry:

```
agxCollide::GeometryRef geometry = new agxCollide::Geometry();
agxWire::WireController::instance()->setEnableDynamicWireContacts(geometry, true);
```

This means that all wires will generate dynamic contacts when colliding with a geometry where the dynamic wire contacts are enabled. A limitation of the current implementation is that if different behavour is wanted from two different wires two different geometries must be used.

17.17 Automatic wire split for performance enhancement

agxWire::Wire support a notion of kinematic splitting during simulation. AGX Dynamics has an automatic partitioner which can analyze a system and split a simulation into subsystems (section 22.2.3). This allows for the task/threading system to solve different parts in different threads. Thus improving performance in the overall system.

```
// Enable kinematic splitting for a wire
wire->setEnableSplitting( true );
```

By enabling the kinematic splitting for a wire, we allow for the system to introduce *kinematic* bodies along the wire, but only intermittent during one time step. Which of the bodies in the wire that are made kinematic will be determined by the splitting algorithm. To avoid artefacts, the place for splitting will be moved between time steps.

For wires under high tension it might not be possible for the splitting algorithm to introduce a kinematic body. This would potentially create unstable/unrealistic simulations.

17.18 Hydro- and aerodynamics

A wire is affected by the fluid that surrounds it. A WindAndWaterController can be used to simulate the affects in water, air and other fluids, see chapter 27.

18 Connecting wires, agxWire::Link

In general, an agxWire::Wire has no knowledge of neighboring connected wires. So, for example if one would like to connect a wire to a chain using a swivel, or wires of different radii using a shackle, or three wires connected through a tow plate – certain functionalities of the wire would not work as expected since all features are handled locally in the agxWire::Wire object.

agxWire::Link enables such functionality to work over connected wire segments. A link contains a user defined rigid body (with arbitrary geometry), a number of connecting wires and algorithms to transfer functionality between the wires.

18.1 Features

An agxWire::Link can have an arbitrarily number of connections. Typically, one or two connections is recommended for all functionality to be enabled.

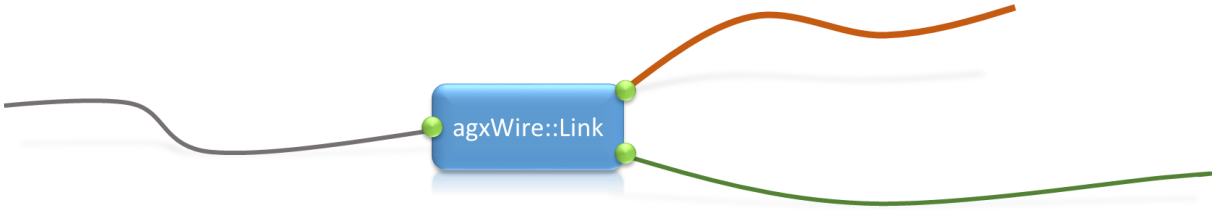


Figure 26: A link with a number of different wire segments connected to it.

The mass of a link is normally relatively small in relation to the tension the connected wires could have, so an agxWire::Link includes additional stabilization constraints when the link is exposed to large forces.

18.2 Configuration

For simplicity, assume this configuration:

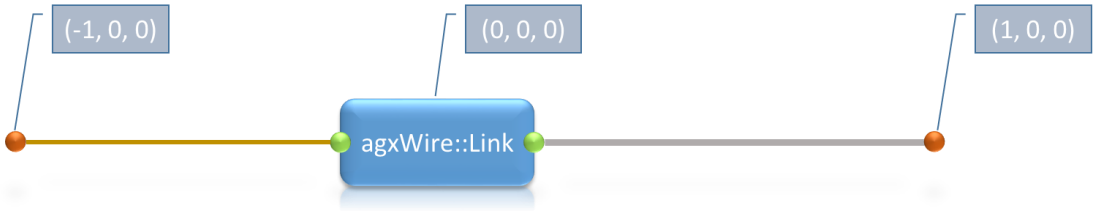


Figure 27: The most common configuration where two wire segments are connected with a link in between.

Two wires connected to a link. Wire 1 has a free begin at (-1, 0, 0), the link positioned in the origin and Wire 2 ends in a free end at (1, 0, 0).

Create the wires and the link:

```
agxWire::WireRef wire1 = new agxWire::Wire( wire1Radius, wire1Resolution );
agxWire::WireRef wire2 = new agxWire::Wire( wire2Radius, wire2Resolution );

// Link rigid body with box geometry.
agx::RigidBodyRef linkRb = new agx::RigidBody( new agxCollide::Geometry( new agxCollide::Box( linkHalfExtent ) ) );
linkRb->setPosition( 0, 0, 0 );
agxWire::LinkRef link = new agxWire::Link( linkRb );
```

The link *must* be positioned any time before the wires are initialized. When the wires are initialized the current position of the link defines the lengths of wires.

The link has to know where it's connected to the wires. A link is either connected to a wire's begin or end position of a wire and there are several ways of defining this.

Connection type from a links point of view:

```
/**
Connection type of the wires connected to this link, i.e., if this
link should be attached to begin or end of the wire.
*/
enum ConnectionType
{
  WIRE_BEGIN         = 0,    /**< Link attached at begin of wire. */
  WIRE_END           = 1,    /**< Link attached at end of wire. */
  INVALID_CONNECTION = 0xFF  /**< Invalid connection, e.g., if this link isn't connected to a give wire. */
};
```

Defining the connections to a link is only a mapping of how the wire composition relates. No nodes are added to the wires!

18.2.1 Routing with explicit connection type

Explicitly assigning the connection type:

```
// The link is connected to Wire 1 end. The connection position
// on the link is given in the link's coordinate system, and
// could in this case be: (-linkHalfExtent.x(), 0, 0)
link->connect( wire1, wire1ConnectionPosition, agxWire::Link::WIRE_END );

// The link is connected to Wire 2 begin. The connection position
// on the link is given in the link's coordinate system, and
// could in this case be: (linkHalfExtent.x(), 0, 0)
link->connect( wire2, wire2ConnectionPosition, agxWire::Link::WIRE_BEGIN );
```

The link now has two connections and now we can route the above system:

```
// Wire 1 first free node.
wire1->add( new agxWire::FreeNode( -1, 0, 0 ) );
// Add the link. The wire will find out how it relates to the
// link and add a node to the wire route. 'false' means that
// the wire shouldn't update the connection type and instead
// fail (return false) if the connection type is a mismatch.
wire1->add( link, false );

// Wire 2 starts at the link...
wire2->add( link, false );
// ...and ends at the free end (1, 0, 0).
wire2->add( new agxWire::FreeNode( 1, 0, 0 ) );
```

The above route is a valid route and the second argument to the **add** method also assures the connections are valid given the wanted route. If the connection type isn't that important, pass **true** as second argument (default is **true**) and the wire will match and update the connection type to the link. Example:

```
link->connect( wire1, wire1ConnectionPosition, agxWire::Link::WIRE_END );
print( link->getConnectionType( link->getConnectingNode( wire1 ) ) ); // "WIRE_END"
// Routing the other way around. Link is attached at wire begin.
wire1->add( link );
wire1->add( new agxWire::FreeNode( -1, 0, 0 ) );
print( link->getConnectionType( link->getConnectingNode( wire1 ) ) ); // "WIRE_BEGIN"
```

18.2.2 Routing with implicit connection pair

agxWire::Link has a static utility method to configure the most common configuration (Figure 27), so configuring such setup can be done with one call:

```
// Wire 1 will get WIRE_END connection type and Wire 2 WIRE_BEGIN.
agxWire::Link::connect( wire1, wire1ConnectionPosition, link, wire2, wire2ConnectionPosition );
```

And the actual routing:

```
wire1->add( new agxWire::FreeNode( -1, 0, 0 ) );
wire1->add( link );

wire2->add( link );
wire2->add( new agxWire::FreeNode( 1, 0, 0 ) );
```

18.2.3 Routing and connecting - all in one call

When routing, a wire has the possibility to know, given nodes already added or not, which connection type a wire should have relative the link. Routing and connecting the system in (Figure 27):

```
wire1->add( new agxWire::FreeNode( -1, 0, 0 ) );
// Will connect to the link and give Wire 1 connection type WIRE_END.
wire1->add( link, wire1ConnectionPosition );

// Will connect to the link and give Wire 2 connection type WIRE_BEGIN.
wire2->add( link, wire2ConnectionPosition );
wire2->add( new agxWire::FreeNode( 1, 0, 0 ) );
```

18.3 agxWire::Winch object

An **agxWire::Winch** is in many senses identical to an **agxWire::WireWinchController**, except a few additional features:

1. Can pull in links.
2. Can pull out links.
3. Can store completely inactive wires and their rest lengths.

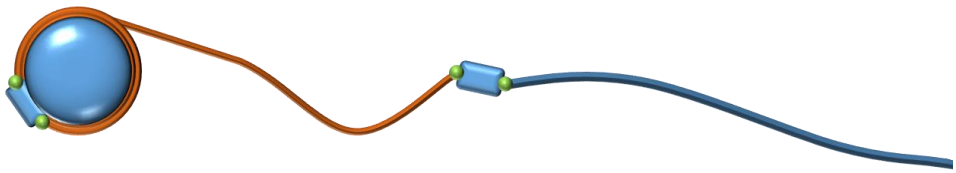


Figure 28: An agxWire::Winch is not a drum. Schematic figure of a wire composition with one completely pulled in segment connected to an (also inactive), pulled in link. The active wire is also connected to the inactive link and to another active link.

18.3.1 Length of a wire that hasn't been routed?

The **agxWire::Wire** object doesn't have a pre-defined length. The length of a wire is only defined by the pulled in length in a winches plus the active distance outside. So for a wire to have a length it must have at least two nodes.

A completely pulled in wire doesn't have active nodes. It only has a connection to a link – which also is inactive (i.e., no position). So for a winch to be able to properly configure a wire that never has been active, one has to 'report' the length of completely pulled in wire segments to the winch.

*How to create an **agxWire::Winch**, please have a look at the agxWire::WinchController section, the constructor and API are identical.*

```
// The completely pulled in wire segment.
agxWire::WireRef inactiveWire = new agxWire::Wire( inactiveWireRadius, inactiveWireResolution );
// The active wire segment that's connected to the winch.
agxWire::WireRef activeWire = new agxWire::Wire( activeWireRadius, activeWireResolution );

// Create the winch (API identical to agxWire::WireWinchController).
```

```
agxWire::WinchRef winch = new agxWire::Winch( winchRb, winchConnectionPoint, winchDirection );

// Report the inactive wire segment to the winch.
winch->add( inactiveWire, inactiveWireLength );
```

18.3.2 Routing with completely pulled in wire segments

The most important thing to remember when routing with inactive segments is not to forget to define the link connections. The following code snippet routes a system similar to Figure 28 but with two inactive, completely pulled in, segments:

```
// The completely pulled in wire segments.
agxWire::WireRef inactiveWire1 = new agxWire::Wire( inactiveWireRadius, inactiveWireResolution );
agxWire::WireRef inactiveWire2 = new agxWire::Wire( inactiveWireRadius, inactiveWireResolution );
// The active wire segment that's connected to the winch.
agxWire::WireRef activeWire1 = new agxWire::Wire( activeWireRadius, activeWireResolution );
// The other active wire segment.
agxWire::WireRef activeWire2 = new agxWire::Wire( activeWireRadius, activeWireResolution );

// DEFINE THE CONNECTIONS:
// Inactive wire 1 is connected to inactive wire 2.
agxWire::Link::connect( inactiveWire1, wire1ConnectionPosition, link1, inactiveWire2, wire2ConnectionPosition );
// Inactive wire 2 is connected to active wire 1.
agxWire::Link::connect( inactiveWire2, wire1ConnectionPosition, link2, activeWire1, wire2ConnectionPosition );
// Active wire 1 is connected to active wire 2.
agxWire::Link::connect( activeWire1, wire1ConnectionPosition, link3, activeWire2, wire2ConnectionPosition );

// Create the winch (API identical to agxWire::WireWinchController).
agxWire::WinchRef winch = new agxWire::Winch( winchRb, winchConnectionPoint, winchDirection );

// Report the inactive wire segments to the winch.
winch->add( inactiveWire1, inactiveWire1Length );
winch->add( inactiveWire2, inactiveWire2Length );

// Begin route from winch to link2. Note that the initial pulled in
// length is how much of activeWire1 that's pulled in.
activeWire1->add( winch, initialPulledInLength );
activeWire1->add( link3 );

activeWire2->add( link3 );
activeWire2->add( new agxWire::FreeNode( wire2EndPosition ) );

// Only the active wires have to be added to the simulation.
simulation->add( activeWire1 );
simulation->add( activeWire2 );
```

Only active wires have to be added to the simulation!

18.3.3 Pulled in length

The agxWire::Winch and agxWire::WireWinchController return the same value for pulled in length – i.e., the amount pulled in of the current segment.

An agxWire::Winch has a few additional methods that support the inactive, totally pulled in, wire segments. Example:

```
agx::Real inactiveWireLength = 10.0;
agx::Real activeWirePulledInLength = 5.0;

// Add inactive wire, 10 m long.
winch->add( inactiveWire, inactiveWireLength );
// Add the winch to the active wire with 5 m pulled in length initially.
activeWire->add( winch, activeWirePulledInLength );

print( winch->getPulledInLength() );      // "5.0"
print( winch->getTotalPulledInLength() ); // "15.0"
print( winch->getNumPulledInWires() );    // "1"

// Remove the inactive wire.
winch->remove( inactiveWire );

print( winch->getPulledInLength() );      // "5.0"
print( winch->getTotalPulledInLength() ); // "5.0"
print( winch->getNumPulledInWires() );    // "0"
```

18.4 agxWire::Link:Algorithm

The link functionality is coupled to separate implementations of agxWire::Link::Algorithm's. A link algorithm receives callbacks with the link at different stages of a time step and when the link changes state from enabled to disabled, and vice versa.

The (current) default algorithms:

1. Detect approaching (sliding) nodes and spawn new algorithms given node type. E.g., from the link's point of view, when a winch is approaching.
2. Not fully pulled in or out link algorithm. The state before a segment is completely pulled in, the link is on a cylindrical joint, and this algorithm handles when the state changes to fully pulled in or out wire segments.
3. The first inactive link in a winch receives callbacks to detect when it's time to go active.
4. Enables high resolution ranges to spread over wire segments.

It's possible to implement your own or add extra (i.e., not default) link algorithms. If you are implementing your own link algorithm it's important to not store/keep a configuration dependent state. Configuration dependent states could be a wire, a start and an end node (which gives a direction). The reason for this constraint is because the link will never know if a wire has been reversed or cut in two, when a node has been removed, been merged with another wire, removed from the simulation etc..

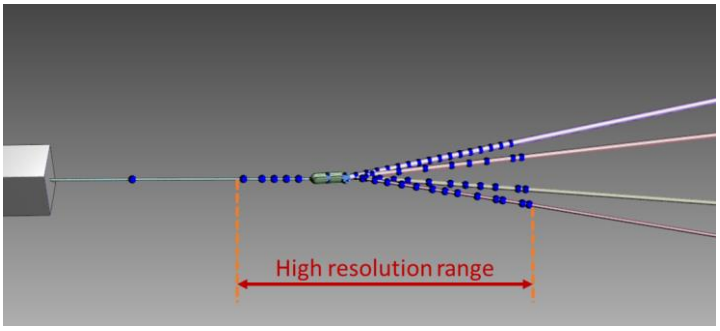


Figure 29: High resolution range over link with five connections.

18.4.1 Optional link algorithms

For usability or performance reasons, some link algorithms are optional.

18.4.1.1 Contact stabilizing algorithm

In some cases, where the tension is high in two or more connections, the link could become unstable when in contact with other objects. This optional link algorithm monitors the tension in the connections and creates additional, stabilizing constraints when the link is in contact with another object.

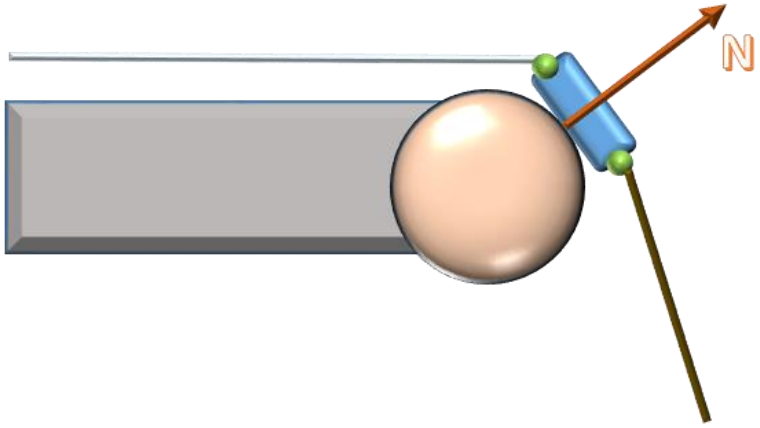


Figure 30: Case where it could be suitable with a contact stabilizing algorithm.

The contact stabilizing algorithm uses its own version of a contact execute filter. It's easy to implement a filter, e.g., this filter that matches a rigid body with name 'sternRoller':

```
#include <agxWire/LinkAlgorithms.h>

class SternRollerMatcher : public agxWire::LinkObjectStabilizingAlgorithm::LinkExecuteFilter
{
  public:
    SternRollerMatcher( const agxWire::Link* link )
      : agxWire::LinkObjectStabilizingAlgorithm::LinkExecuteFilter( link ) {}

    // otherGeometry is overlapping the link - check if match.
    virtual bool matchOther( const agxCollide::Geometry* otherGeometry ) const
    {
      return otherGeometry->getRigidBody() != 0L && otherGeometry->getRigidBody()->getName() == "sternRoller";
    }
};
```

So when **matchOther** returns true AND the tension is high enough, extra constraints will be added to stabilize the link.

There are some already implemented filters. For example filters that match a property (in agx::PropertyContainer) in either a geometry or a rigid body. Example with geometry:

```
// Filter that checks a geometry's property container.
agxWire::LinkObjectStabilizingAlgorithm::GeometryPropertyBoolFilterRef geometryPropertyFilter =
  new agxWire::LinkObjectStabilizingAlgorithm::GeometryPropertyBoolFilter( "stabilize", link );

link->add( new agxWire::LinkObjectStabilizingAlgorithm( geometryPropertyFilter ) );

// Enables algorithm when the link is in contact with this geometry.
someGeometry->getPropertyContainer()->addPropertyBool( "stabilize", true );
```

Example with rigid body:

```
// Filter that checks a rigid body's property container.
agxWire::LinkObjectStabilizingAlgorithm::GeometryPropertyBoolFilterRef rbPropertyFilter =
  new agxWire::LinkObjectStabilizingAlgorithm::GeometryPropertyBoolFilter( "stabilize", link );

link->add( new agxWire::LinkObjectStabilizingAlgorithm( geometryPropertyFilter ) );

// Enables algorithm when the link is in contact with this rigid body.
someRigidBody->getPropertyContainer()->addPropertyBool( "stabilize", true );
```

18.5 Known limitations

18.5.1 Links and EyeNodes

A Link can not slide through an EyeNode. This is a feature that might be added in a future release of AGX Dynamics.

18.5.2 Loops in the configuration

A valid link-wire composition cannot have loops. There are recursive sections in the code, assuming a tree structure of the connection assembly. A link can only detect a loop if the user tries to connect the same wire twice to a link, and other than that, no runtime checks nor parsing of the current configuration to detect loops are made.

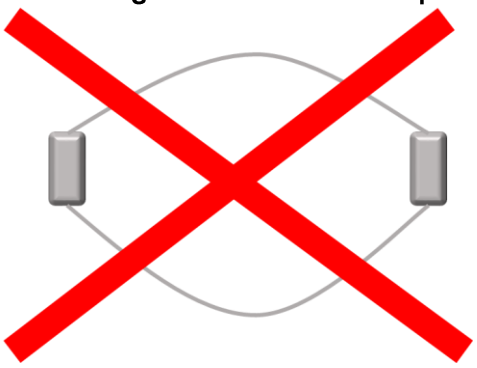


Figure 31: Loops in the link connections is undefined.

This loop limitation prevents the user from creating a lasso using links and the connect-interface. But the reason for links is functionality such as pulling them into/out from winches, and pulling a lasso into a winch doesn't often make that much sense.

The more reasonable scenario when pulling a lasso into a winch is that the winch stops, due to jamming (or something similar). To achieve such behavior:

1. Connect the main wire to the link (i.e., not the lasso wire). The link should only have one mapped connection, making the link a so called 'end link'. I.e., the link defines the end of a link-wire composition.
2. Create and route the lasso wire only using explicit connections. Explicit connections could be separate bodies locked to the link body or body fixed/connecting nodes with the link body as the rigid body.

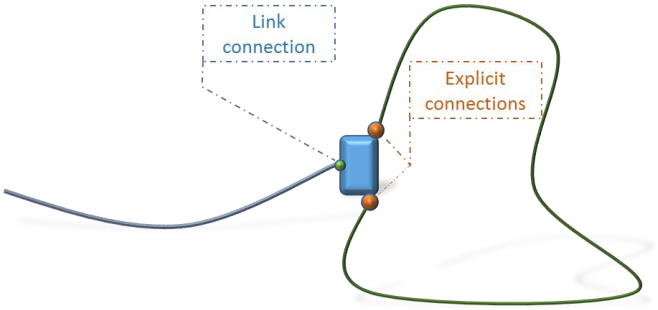


Figure 32: Lasso configuration so that a winch will automatically stop when the lasso is about to be pulled in.

18.5.3 Length of connections and winches

The maximum distance from a connection point, on the link surface, to the link center of mass, must be less than one meter for the winches to work with the link. In principle, this means that a somewhat symmetric link can't be longer than two meters AND be pulled into or out from a winch. If winches aren't involved, there's no limitation in the connection lengths.

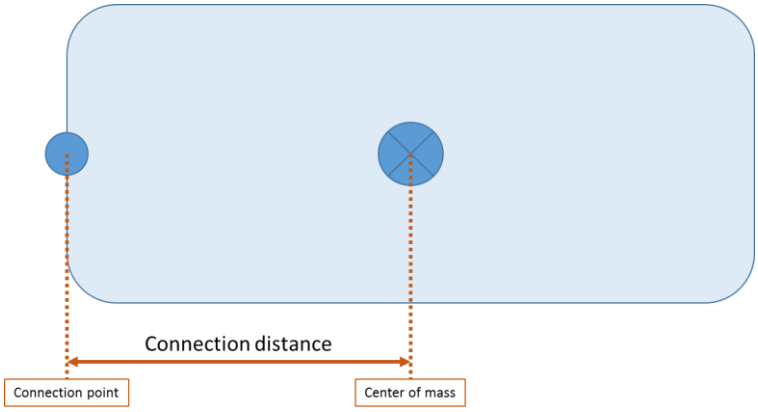


Figure 33: Definition of the connection distance going from the connection point to the center of mass.

18.6 Controlling Link behaviour

18.6.1 Link and bend

Connecting wires to small link objects are often problematic when simulating offshore operations due to the wire tension to link mass ratio difference. Early, AGX have had support for such connections, supporting stable simulation with wires and links, even when the system involves extreme potential energies. The solution to the initial problem has been to have a variable stiffness of the bend constraint over the link, i.e., controlling more degrees of freedoms to filter out high frequencies.

This control of the bend stiffness has been, and will always be, handled internally. When the tension is low the bend was completely ignored making the connection look like a ball joint type of connection. I.e., the link could rotate freely about the connecting point. This type of connection behavior isn't always desirable and the user should be able to control the bend stiffness when the link isn't under extreme stress.

18.6.2 Link and twist

If the wire going out from a link connection defines an axis, the twist is defined to be the rotation about that axis.

Until now, this degree of freedom has been completely ignored by the wire. Mainly because there're absolutely no information about it because the wire uses 3 DOF (particle) bodies internally. Bending is still defined when the geometry of this exist given three control points along a wire.

Since twist is a completely new concept for the links, consider it experimental, and more work has to be done to figure out the API to control all desirable "effects" of twisting a link about a wire.

18.7 Introducing interface to a Link Node and Connection Properties

18.7.1 agxWire::ILinkNode

This new node is only an interface to the implementation of the special node used when connecting an agxWire::Wire to an agxWire::Link. At this moment, the interface is minimalistic but can be extended easily for future features.

```
/**
Interface class for nodes connected to links.
*/
class AGXPHYSICS_EXPORT ILinkNode : public agxWire::ConnectingNode
{
```

```
  public:
    /**
    \return the connection properties for the connection between the wire and the link
    */
    virtual agxWire::ILinkNode::ConnectionProperties* getConnectionProperties() const = 0;
};
```

18.7.2 agxWire::ILinkNode::ConnectionProperties

New object holding parameters and properties of a connection between a wire and a link. Currently it's small and it'll be extended/changed when we fully understand these connections.

```
/**
Object holding parameters, such as bend and twist stiffness,
related to wire to link connections.
*/
class AGXPHYSICS_EXPORT ConnectionProperties
{
  public:
    /**
    Default constructor, with default bend and twist stiffness disabled.
    */
    ConnectionProperties();

    /**
    Construct given bend- and twist stiffness.
    */
    ConnectionProperties( agx::Real bendStiffness, agx::Real twistStiffness );

    /**
    Assign bend stiffness of the link to wire connection. The bend stiffness is
    interpreted as Young's modulus of the wire. I.e., the final stiffness
    is dependent on the wire radius and segment lengths. Default: 0.
    \param bendStiffless - new bend stiffness for this connection
    */
    void setBendStiffness( agx::Real bendStiffness );

    /**
    \return the currently used bend stiffness of this connection (default: 0)
    */
    agx::Real getBendStiffness() const;

    /**
    Assign twist stiffness of the link to wire connection. The twist stiffness is
    interpreted as Young's modulus of the wire. I.e., the final stiffness
    is dependent on the wire radius. Currently, the segment length is assumed
    to be 1 meter and default value of the stiffness is 0.
    \param twistStiffness - new twist stiffness for this connection
    */
    void setTwistStiffness( agx::Real twistStiffness );

    /**
    \return the currently used twist stiffness of this connection (default: 0)
    */
    agx::Real getTwistStiffness() const;
};
```

Bend- and twist stiffness may be any real value >= 0. By default the values are all 0 to maintain the old behavior.

18.7.3 Default connections

Links carries a default connection property. I.e., all new connections inherits the values of this default property.

```
/**
Object that defines the relation between different types of wires (agxWire::Wire). I.e.,
this link object enables functionality to (for example) move nodes from one wire to
another.
*/
class AGXPHYSICS_EXPORT Link : public agx::Referenced, public agxStream::Serializable
{
```

```
  public:
    ...

    /**
    Access default connection properties for each new wire that's connected to this link.
    \NOTE If any value is changed the current connections will not be updated with this value.
          Only new connection will receive this new default value.
    \return default connection properties
    */
    agxWire::ILinkNode::ConnectionProperties* getDefaultConnectionProperties();

    /**
    \return default connection properties
    */
    const agxWire::ILinkNode::ConnectionProperties* getDefaultConnectionProperties() const;

    ...
}
```

As stated in the API documentation, the changes are not propagated to all current connection. It'll only affect new connections, making it possible to have a default look, but custom if e.g., a chain, instead of a wire, is connected.

18.8 Twist again

The implementation of the link to wire twist constraint is basically a hinge axis. I.e., it's possible to control it within a range, motorize it or lock at certain angle. You'll have to test and see what the desirable behavior should be.

Currently there's no interface to change these behaviors. Default is a motor at zero speed, i.e., the twist stiffness controls the inertia to rotate the link about the twist axis.

18.9 Small Example

Example to change bend stiffness of individual nodes where the wires have identical radius.

```
void createScene( agxSDK::Simulation* simulation )
{
  const agx::Real radius = 0.015;

  agxWire::WireRef wire1 = new agxWire::Wire( radius, 2.0 );
  agxWire::LinkRef link1 = new agxWire::Link( createLink( simulation ) );

  agxWire::WireRef wire2 = new agxWire::Wire( radius, 2.0 );
  agxWire::LinkRef link2 = new agxWire::Link( createLink( simulation ) );

  agxWire::WireRef wire3 = new agxWire::Wire( radius, 2.0 );
  agxWire::LinkRef link3 = new agxWire::Link( createLink( simulation ) );

  link1->getRigidBody()->setPosition( 0, -1, 0 );
  link2->getRigidBody()->setPosition( 0,  0, 0 );
  link3->getRigidBody()->setPosition( 0,  1, 0 );

  simulation->add( new agx::LockJoint( link1->getRigidBody() ) );
  simulation->add( new agx::LockJoint( link2->getRigidBody() ) );
  simulation->add( new agx::LockJoint( link3->getRigidBody() ) );

  wire1->add( link1, agx::Vec3( 0.15, 0, 0 ) );
  wire1->add( new agxWire::FreeNode( 3, -1, 0 ) );

  wire2->add( link2, agx::Vec3( 0.15, 0, 0 ) );
  wire2->add( new agxWire::FreeNode( 3, 0, 0 ) );

  wire3->add( link3, agx::Vec3( 0.15, 0, 0 ) );
  wire3->add( new agxWire::FreeNode( 3, 1, 0 ) );

  simulation->add( wire1 );
  simulation->add( wire2 );
  simulation->add( wire3 );
```

```
  agxWire::ILinkNode* link1Node = link1->getConnectingNode( wire1 );
  agxWire::ILinkNode* link2Node = link2->getConnectingNode( wire2 );
  agxWire::ILinkNode* link3Node = link3->getConnectingNode( wire3 );

  // Closest wire in the picture.
  link1Node->getConnectionProperties()->setBendStiffness( 1.0E12 );
  // Middle wire.
  link2Node->getConnectionProperties()->setBendStiffness( 2.0E10 );
  // Farthermost wire.
  link3Node->getConnectionProperties()->setBendStiffness( 1.0E8 );
}
```

Result after some time. Wire 3 is swinging back and forth but the others are not moving.

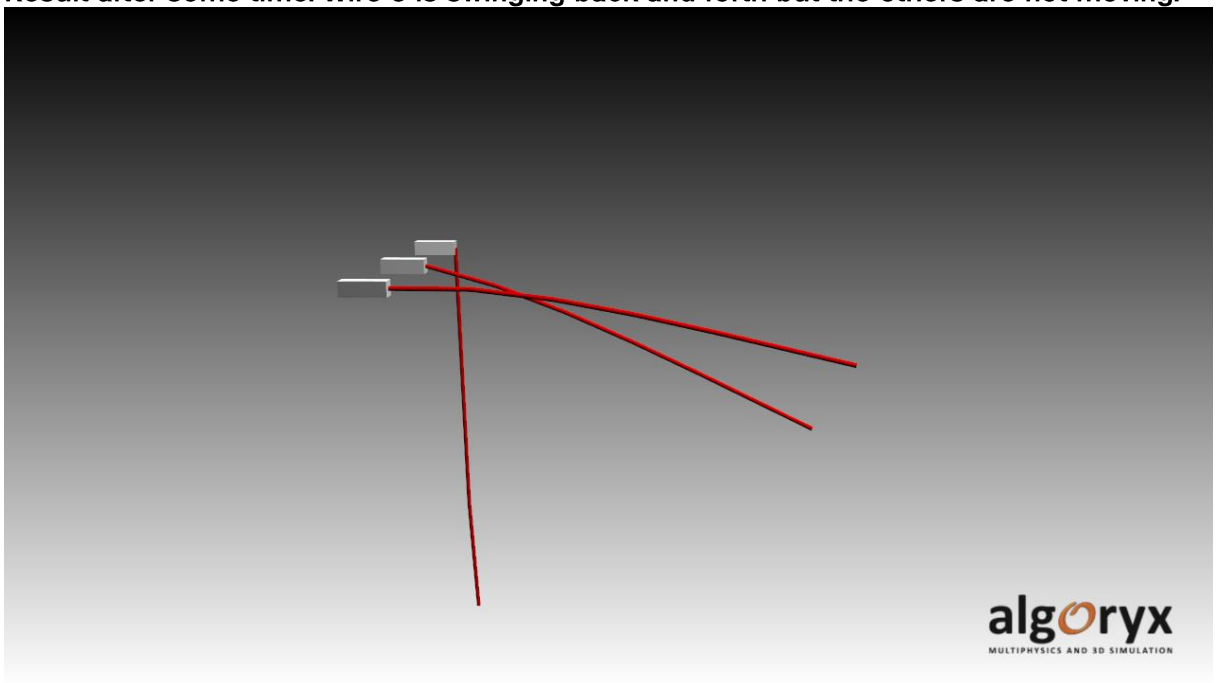


Figure 34 Three links and wires where the wires has identical radius but the connection bend stiffness differs.

Another example where all wires has different radius and the links has identical default connection properties:

```
void createScene( agxSDK::Simulation* simulation )
{
  const agx::Real radius1 = 0.050;
  const agx::Real radius2 = 0.025;
  const agx::Real radius3 = 0.010;

  agxWire::WireRef wire1 = new agxWire::Wire( radius1, 2.0 );
  agxWire::LinkRef link1 = new agxWire::Link( createLink( simulation ) );

  agxWire::WireRef wire2 = new agxWire::Wire( radius2, 2.0 );
  agxWire::LinkRef link2 = new agxWire::Link( createLink( simulation ) );

  agxWire::WireRef wire3 = new agxWire::Wire( radius3, 2.0 );
  agxWire::LinkRef link3 = new agxWire::Link( createLink( simulation ) );

  // Before making any connections, set default bend stiffness.
```

```
  // Closest wire in the picture.
  link1->getDefaultConnectionProperties()->setBendStiffness( 1.0E10 );
  // Middle wire.
  link2->getDefaultConnectionProperties()->setBendStiffness( 1.0E10 );
  // Farthest wire.
  link3->getDefaultConnectionProperties()->setBendStiffness( 1.0E10 );

  link1->getRigidBody()->setPosition( 0, -1, 0 );
  link2->getRigidBody()->setPosition( 0,  0, 0 );
```

```
  link3->getRigidBody()->setPosition( 0,  1, 0 );

  simulation->add( new agx::LockJoint( link1->getRigidBody() ) );
  simulation->add( new agx::LockJoint( link2->getRigidBody() ) );
  simulation->add( new agx::LockJoint( link3->getRigidBody() ) );

  wire1->add( link1, agx::Vec3( 0.15, 0, 0 ) );
  wire1->add( new agxWire::FreeNode( 3, -1, 0 ) );

  wire2->add( link2, agx::Vec3( 0.15, 0, 0 ) );
  wire2->add( new agxWire::FreeNode( 3, 0, 0 ) );

  wire3->add( link3, agx::Vec3( 0.15, 0, 0 ) );
  wire3->add( new agxWire::FreeNode( 3, 1, 0 ) );

  simulation->add( wire1 );
  simulation->add( wire2 );
  simulation->add( wire3 );
}
```

Resulting in this steady state:

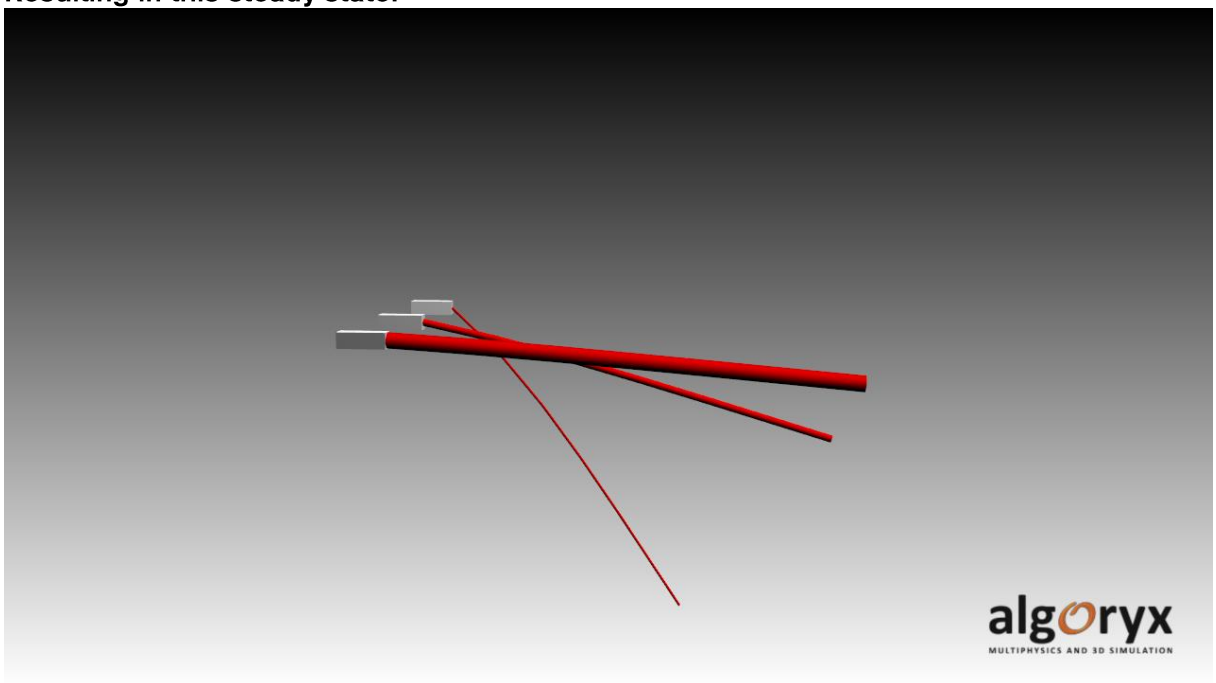


Figure 35 Three links and wires where the connection bend stiffness is identical but the wires has different radii.

A trained eye can see that the relation between the radius and stiffness is proportional to r^4.

19 agxCable – Simulating flexible structures

The agxCable module is used to simulate cables, hoses, ropes, short wires and dress packs. These are long structures with a circular cross section that can be bent, stretched and twisted. The agxCable namespace provides the Cable class which uses a lumped elements approach to model such structures. By using **agxCable::Cable** it is possible to create simulations with cable structures that behave realistically in contacts and interaction with other simulated objects. This can be used, for example, for training operations of assembling scenarios, measurement of load and wear on cables in a robot scenario or for cable motion analysis in articulated devices.

19.1 Model

The lumped elements approach means that the cable is represented by a sequence of rigid bodies linked together using constraints. Each rigid body constitutes what is called a segment. Together the segments define the shape, extent and location of the cable. In addition to a rigid body, each segment has a geometry that allows it to collide with other geometries in the simulation, including other segments of the same cable. Each geometry contains a capsule and consecutive segments are placed with a bit of overlap so that the flat sides of facing hemispheres coincide in a straight cable.

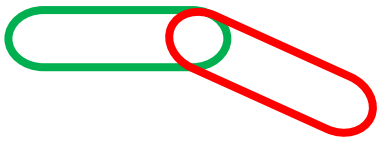


Figure 36: Two consecutive cable segments. Notice the overlap.

This makes the cable surface smoother at bends, compared to if cylinders had been used. The geometries for consecutive segments are exempted from contact generation.

The cable is a one-dimensional structure and thus each constraint connects exactly two segments.

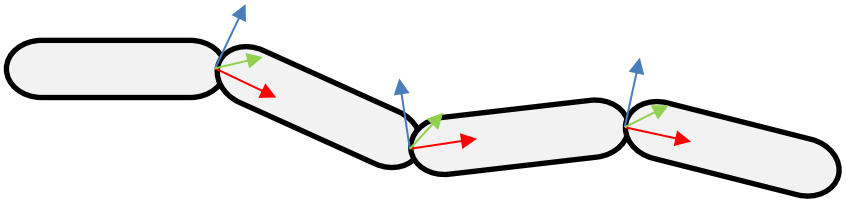


Figure 37: A Cable consisting of four segments and three LockJoints (constraints).

Parameters and properties set on the cable are translated into parameters on these bodies and constraints. This makes it possible to create cables that range from easily bendable to completely stiff.

Both the bodies and the constraints operate on all six degrees of freedom, which makes it possible to track bending, twisting and stretching of the cable independently.

19.2 Cable vs Wire

agxWire is another very efficient AGX component that is used to simulate long, bendable structures. Whereas the cable has a *fixed resolution*, meaning that the number

and size of the segments remain constant throughout the simulation, the wire has *dynamic resolution.* This is the main difference compared to agxWire, which can adjust the resolution locally in order to improve stability. This makes it possible to use agxWire in scenarios with extreme tension and in large scale scenarios with several kilometers of wire. On the other hand, the agxCable model supports modeling of plasticity and torsion, which are not available for agxWire. The wires, but not the cables, can be cut and merged, and also spooled in and out of winches.

Use Wire if you want to simulate:

- Very long wires/chains/ropes (50m or more) (Anchoring, hoisting, crane wires) with varying length.
- Wire winches (winching wire in and out)
- Be able to withstand high tension/large mass ratios
- Torsion is not relevant
- Typical scenarions: crane operations, lifting heavy containers, anchoring oil rigs, towing ships

Use Cable if you want to simulate:

- Hoses, cables, pipes of shorter distances with fixed length
- Limited tension
- High fidelity self interaction (collision)
- Requires torsion
- Requires plasticity
- Typical scenarios: cables on an industrial robot, hoses/cables in an assembly simulation of a car engine.

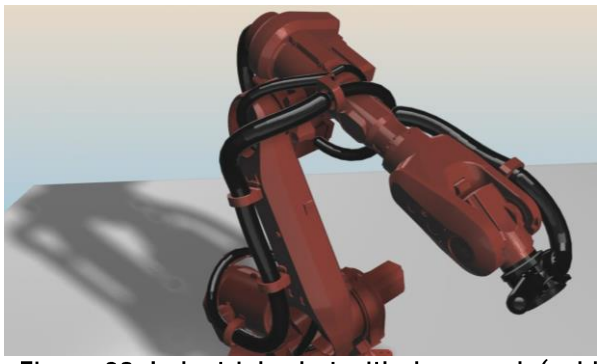


Figure 38: Industrial robot with dresspack (cables) attached.

19.3 Creating a Cable

There are two parts to creating a cable: routing and properties configuration.

19.3.1 Routing

Cable routing is the process that defines the initial path of the cable. The user specifies the path using routing nodes. A cable created from a collection of routing nodes will extend from node to node in the order that the nodes were added to the cable. Each such routing node pair is called a leg of the path. There are two types of routing nodes: FreeNode and BodyFixedNode. A FreeNode simply defines a location in space that the

cable should pass through. This type of node produces no special behavior once the simulation has started.

```
agx::Real radius(0.05);
agx::Real resolution(3.35);
agxCable::CableRef cable = new agxCable::Cable(radius, resolution);
// Add 3 points in world coordinate system
cable->add(new agxCable::FreeNode(0.0, 0.0, 0.0));
cable->add(new agxCable::FreeNode(1.0, 0.5, 0.0));
cable->add(new agxCable::FreeNode(2.0, 0.3, 0.0));
```

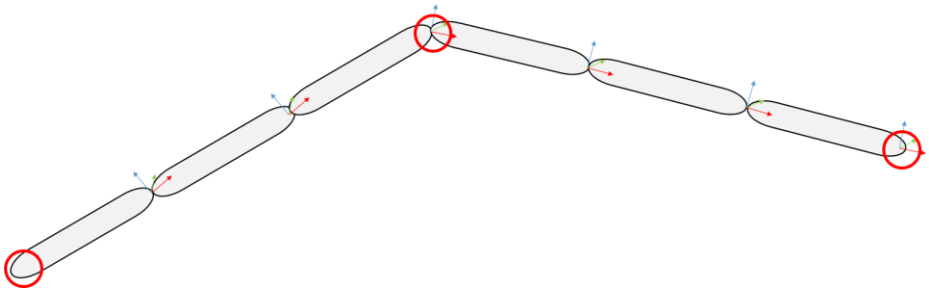


Figure 39: Cable created from three FreeNodes indicated by red circles.

The points supplied defines the range of the cable in space, but does not include the capsule hemispheres. The cable will therefore be one cable diameter longer than the total distance between the routing nodes.

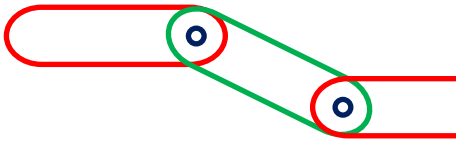


Figure 40: Three cable segments created from three routing nodes.

The extra extent of the cable becomes important when an endpoint of the cable should be on the surface of some object. As shown in Figure 41: Cable routed from the surface of a wall., placing the routing node on the surface will cause some initial overlap with the object. This will result in contacts and sudden cable movement when the simulation is started. Two possible solutions are to either place the routing node one cable radii away from the surface, or to disable collisions between the object and the first cable segment.

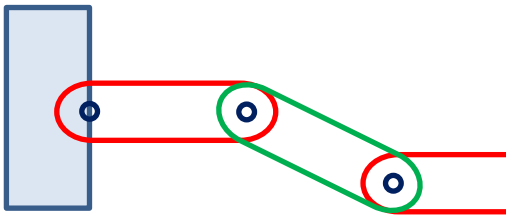


Figure 41: Cable routed from the surface of a wall.

BodyFixedNodes are used to attach the cable to some other rigid body so the cable and the body will follow each other as the simulation progresses. The BodyFixedNode can be configured to either allow or prevent the body from rotating relative to the cable by providing different arguments to the BodyFixedNode constructor. By providing only an offset, in the form of an agx::Vec3, we specify that only that point on the body, in the body's model coordinate frame, should be attached to the cable with a Ball Joint. By

providing a full transformation, in the form of an agx::AffineMatrix4x4, we specify how the whole body should be placed relative to the cable using a LockJoint.

```
agx::RigidBodyRef rotatingBody = new agx::RigidBody();
agx::RigidBodyRef fixedBody = new agx::RigidBody();

// Attach the rotatingBody with a point: results in a BallJoint attachment constraint
cable->add(new agxCable::BodyFixedNode(rotatingBody, agx::Vec3()));

// Attach the rotatingBody with an AffineMatrix4x4: results in a LockJoint attachment constraint
cable->add(new agxCable::BodyFixedNode(fixedBody, agx::AffineMatrix4x4()));
```

When using a transformation matrix, the transformation describes where the routing node should be placed and how the body should be oriented relative to the cable, where the Z axis is defined to be along the cable.

Class	Description
agxCable::FreeNode	Initial positioning of cable.
agxCable::BodyFixedNode	Attach cable to rigid body.

Table 19: Cable routing node types.

The routing phase ends when the cable is added to a simulation. At this point the routing is finalized and segmentation is performed. Segmentation is the process that creates and positions segments between the routing nodes. The routing nodes are strictly adhered to, meaning that, except for at the cable end points, the routing node defines the end of one segment and the start of another. It is the responsibility of the segmentation algorithm to find a segment length that as evenly as possible divides all the route node pairs. This makes the smallest inter-node distance a lower bound on the cable's resolution. The user supplied cable resolution is used as a start point for the search. It is not always possible to find a segment length that perfectly divides all path legs. In that case, some of the path legs will contain either small overlaps or small extra gaps. The intention is that these will be evened out during the first few time steps of the simulation. For this to work there must be some room for the cable to move. If two BodyFixedNodes are added next to each other then neither side can move and the cable may become unstable.

It is possible to dry-run the segmentation algorithm in order to gather information about the current routing without finalizing it. This is done by calling tryInitialize on the cable and returned is an initialization report that contains information such as the number of nodes created, the resolution of the cable, the largest segment error and the total length of the cable.

Additional routing nodes cannot be added to the cable after routing has been finalized. However, as an alternative to BodyFixedNodes the cable provides attachments that provide similar functionality. By attaching a rigid body to a cable segment the body will follow the cable as the simulation progresses. Attaching bodies this way follows the same translation/transformation rules as the BodyFixedNode. Note that adding attachments does not change the path of the cable, or the position of the attached object.

Attaching bodies requires a way to specify where along the cable the body should be attached. This is done using cable iterators, which are described in the section 19.6.

```
agx::RigidBodyRef attachedBody = ...;
agxCable::CableIterator segment = ...;
// Attach a body with a BallJoint to the segment.
cable->attach(segment, attachedBody, agx::Vec3());
```

19.4 Direct segment routing

Direct segment routing is an an alternative to the segmentation process described in Section 19.3.1. Using direct segment routing it is possible to precisely control both the position and orientation of each individual cable segment during routing. This is useful when the cable path and segmentation is defined by an external tool or is known though some other means.

A cable with direct routing support is created by passing an instance of `agxCable::IdentityRoute` to the cable constructor instead of the resolution parameter. The routing nodes added to such a cable describe not only the path of the cable, but also the position and orientation of each and every segment. No additional segments will be created, and every segment will correspond to a routing node.

The transformation of the segments are configured by setting the transformation of the RigidBody that the routing node contains.

The segmentation algorithm handles the last routing node separately from the rest of the nodes. There is no such special handling when using direct segment routing. Just like every other segment, the last segment is placed with its start at the location of the corresponding routing node. When using the segmentation algorithm the last segment is placed with its end at the last routing node.

```
   const agx::Real radius = agx::Real(0.04);
   const agx::Real resolution = agx::Real(3.0);
   agxCable::CableRef cable = new agxCable::Cable(radius, new
agxCable::IdentityRoute(resolution));

   agxCable::FreeNodeRef node1 = new agxCable::FreeNode(pos1);
   node1->getRigidBody()->setRotation(rot1);
   cable->add(node1);

   agxCable::FreeNodeRef node2 = new agxCable::FreeNode(pos1);
   node2->getRigidBody()->setRotation(rot2);
   cable->add(node2);

   agxCable::BodyFixedNodeRef node3 = new
agxCable::BodyFixedNode(attachedBody, attachmentPointInBodyFrame);
   node3->getRigidBody()->setPosition(pos3);
   node3->getRigidBody()->setRotation(rot3);
   cable->add(node3);
```

19.5 Properties

The cable provides several properties that control its behavior. These properties are accessed and modified via the `agxCable::CableProperties` class accessible using `Cable::getCableProperties`. Through the properties handle we can set things such as

Young's modulus and damping. Several cables can share the same properties handle, making it possible to easily configure and modify many cables at once.

Property name	Unit	Description
Young's modulus	Pressure	Controls the stiffness of the cable.
Poisson's ratio	unitless	Ratio of Stress over Strain. Will affect rotational stiffness when twisting.
Damping	Time	Controls how quickly potential energy in cable deformations dissipates.

The parameters can be set independently along the three dimensions of the cable: bend, twist and stretch. The following code snippet demonstrates how to set Young's modulus in bend, twist and stretch directions.

```
agxCable::CableProperties* properties = cable->getCableProperties();
properties->setYoungsModulus(1e7, agxCable::BEND);
properties->setYoungsModulus(1e7, agxCable::TWIST);
properties->setYoungsModulus(1e7, agxCable::STRETCH);
```

Care should be taken when simulating cables with very low Young's modulus in the stretch direction. The cable geometries are not resized when the cable is deformed leading to gaps in the cable during extreme elongation.

A different set of properties of the cable are those that all geometries have and that are represented by an instance of the agx::Material class. By assigning a material to a cable all geometries that make up that cable are assigned that same material. This, along with the contact materials created for the material assigned, control much of how the cable interacts with other objects in the scene. Some parameters share the same name as parameters found in the cable properties. Despite this, these are separate values with different purposes. For example, the Young's modulus in the the geometry material control stiffness in contacts while the cable property with the same name control stiffness within the cable itself.

The density set in the geometry material is used to compute the mass of each cable segment, but due to the overlap between consecutive segments the total mass of the cable will be higher than what a density times volume computation would give. A higher resolution cable will have more segments and thus more overlaps and a larger mass discrepancy.

In addition to the internal damping in the cable, it is possible to set both linear and angular velocity damping of the cable using setLinearVelocityDamping and setAngularVelocityDamping.

19.5.1 Plasticity

A cable can be given plastic deformation properties by adding a `CablePlasticity` component to it. The plasticity parameters are configured on the `CablePlasticity` object using the same direction parameters as when configuring other direction dependent cable parameters.

```
agxCable::CableRef cable = …;
agxCable::CablePlasticityRef plasticity = new CablePlasticity();
plasticity->setYieldPoint(1e9, agxCable::BEND);
cable->addComponent(plasticity);
```

The yield point specifies the torque required to permanently deform the cable, i.e., to cause a plastic deformation. Deformations in the stretch direction are always elastic.

19.6 Inspection using iterators

Some state of the cable, such as resolution and properties, apply to the whole cable while others, such as tension, vary along the cable. To inspect the latter the cable provides an iterator class that is used to iterate over the segments of the cable. Segmentation must have been performed on the cable before any iterators can be created. Using the iterator, we can find information such as the position of the segment, the rotational or translational tension and also get a hold of both the rigid body and the geometry that is used to represent the segment in the simulation.

```
for (agxCable::CableIterator segment = cable->begin();
!segment.isEnd();
++segment)
{
  // Use the accessor methods provided by CableIterator to inspect
  // dynamic cable state.
  agx::Real tension = segment->getStretchTension();
  agx::RigidBody* body = segment->getRigidBody();
  agxCollide::Geometry* geometry = segment->getGeometry();
}
```

The cable also supports range based for loops:

```
for (auto segment : *cable)
{
}
```

Once the cable has been initialized it is possible to create cable iterators directly from the routing nodes. This makes it possible to quickly get at the segment created near a particular node. This can be used, for example, to track the motion of some interesting point along the cable, or to find the geometry for a particular part of the cable in order to disable collisions between that geometry and geometries held by a body attached to the cable.

```
agxCable::CableRef cable = ...;
agx::RigidBodyRef body = ...;
agxCable::BodyFixedNodeRef node = new agxCable::BodyFixedNode(body, agx::Vec3());
cable->add(node);
simulation->add(cable);
agxCable::CableIterator segment(attachedNode);
agxUtil::setEnableCollisions(segment->getGeometry(), body, false);
```

A geometry, for example received as an argument to a contact event listener, can be passed to the static Cable::getCableForGeometry function and if that geometry is part of a cable then a pointer to that cable will be returned.

```
agxCollide::Geometry* geometry = ...;
// Return the Cable for which the geometry is part of. nullptr if the geometry is not
// part of any Cable.
agxCable::Cable* cable = agxCable::Cable::getCableForGeometry(geometry);
```

An initialized cable has two lengths: the rest length and the current length. The rest length is the sum och the lengths of the segments that make up the cable and the current length is the sum of the distances between the starting positions of consecutive segments, and the end position of the last segment. If there is no tension then the two lengths are the same, but if there is some load on the cable then the two lengths may differ. We say that the cable has become either stretched or compressed. The amount of stretch a given load produces is controlled by configuring Young's modulus on the cable.

19.7 Hydro- and aerodynamics

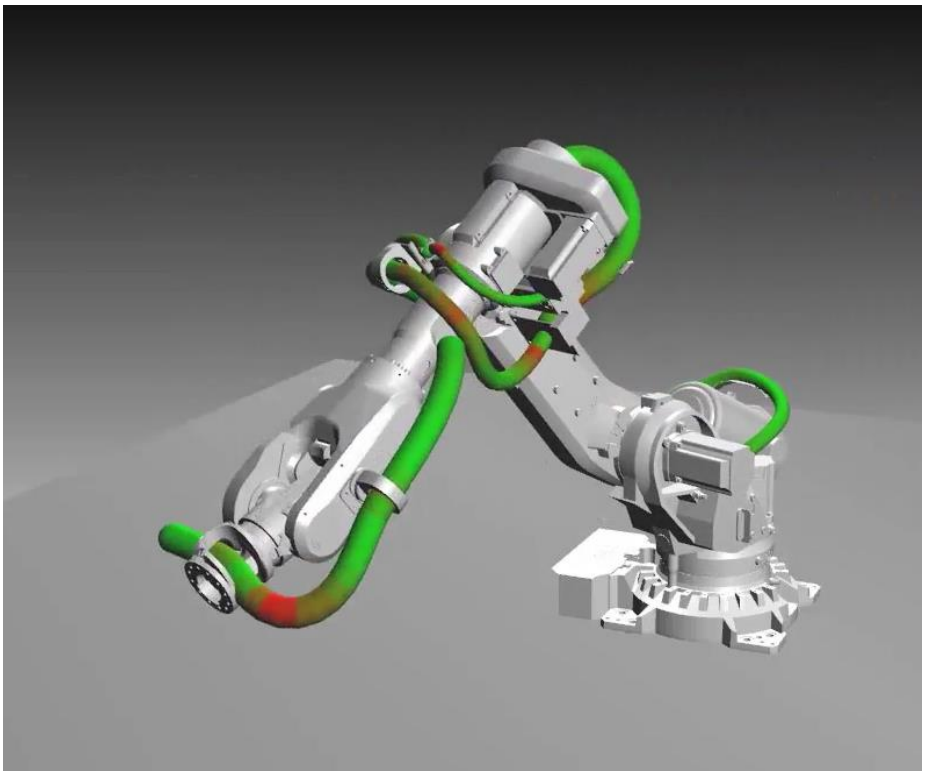
A cable is affected by the fluid that surrounds it. A WindAndWaterController can be used to simulate the affects in water, air and other fluids, see chapter 27.

19.8 Known limitations

- Mass not properly computed from density because of hemisphere overlaps.
- Geometries are not resized to match the current stretching of the cable.
- Cables must be circular.
- Cables must be homogeneous, both in size and properties.

20 Cable damage

An important factor in cable application design is cable wear. The cable damage component of the agxCable module provides a way to estimate damage caused to the cable during the simulation. Effects such as tension, deformation, and contacts are taken into account by the damage estimation model and applied to each cable segment individually.



Robot with several cables. Current cable damage visualized by color.

20.1 Basic usage

The damage estimation is performed by creating an instance of the `agxCable::CableDamage` class. It uses the simulation result of a cable to estimate damage and wear inflicted on the cable. The following code should be executed before the simulation starts to gather damage estimation for a cable named `cable`.

```
agxCable::CableDamageRef damage = new agxCable::CableDamage();
cable->addComponent(damage);
```

During or after the simulation the estimated damages can be accessed via instances of the `SegmentDamage` class, which the `CableDamage` instance provides. Damage is estimated per cable segment, and there is one `SegmentDamage` instance per segment of the simulated cable. We can read the damage accumulated over the simulation so far for a particular cable segment with the following code.

```
agx::Real wear = cableDamage->getAccumulatedDamageAt(segmentIndex).total();
```

The damage inflicted to a particular cable segment during the most recent simulation step is accessed using the following code.

```
agx::Real newWear = cableDamage->getCurrentDamageAt(segmentIndex).total();
```

In addition to the total damage, it is possible to read the contribution from the individual damage estimation sources for a particular segment. To better understand the available sources we must first understand the damage estimation model.

20.2 Damage estimation model

Cable damage is estimated each time step from a number of contribution sources. Each source is scaled according to a weight associated with the source and the weighted values are summed to form a total damage contribution for that time step.

The sources are categorized into four groups: deformation, deformation rate, tension, and contact force. The first three groups are further categorized based on the direction of deformation (or tension): bend, twist, and stretch. The fourth group, contact force, is split in normal force and friction force. This results in a total of eleven (3x3+1x2) damage estimation sources.

Deformation, deformation rate and tension have a threshold each, below which a deformation/tension gives no contribution to the damage estimation.

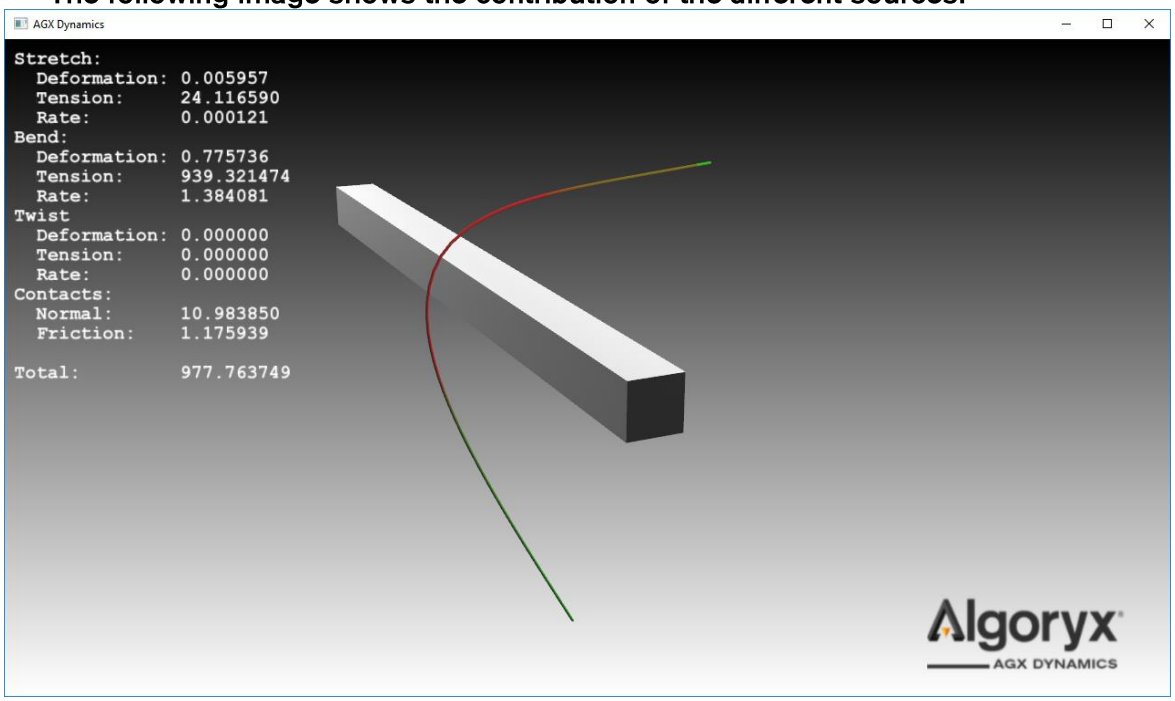
The complete damage estimation formula is as follows:

$$
\begin{aligned}
d = {} & W_{T_b} T_b^2 + \left(W_{R_b} R_b + W_{b_0}\right) \max(0, b - b_0)^2 \\
& + W_{T_t} T_t^2 + \left(W_{R_t} R_t + W_{t_0}\right) \max(0, t - t_0)^2 \\
& + W_{T_s} T_s^2 + \left(W_{R_s} R_s + W_{s_0}\right) \max(0, s - s_0)^2 \\
& + W_n F_n^2 + W_f F_f^2
\end{aligned}
$$

W	Weighting factor
T	Tension
R	Deformation rate
F	Contact force
b	Along bend direction
t	Along twist direction
s	Along stretch direction
n	Along contact normal
f	Along contact tangent

0	Deformation threshold

The following image shows the contribution of the different sources:



Damage estimation is tracked per cable segment. Dynamic cable properties that are defined as a relationship between two consecutive cable segments, such as tension and deformation, are distributed evenly over the two segments and thus the segment's damage contribution becomes the average of the damage contributions at the segment's two end points.

20.3 Inspecting sources

The `SegmentDamage` object returned by `CableDamage::getAccumulatedDamageAt` includes not only the total damage, but also a value for each of the damage estimation sources listed in the damage estimation model section above. They are accessed either by their name or by index.

```
const agxCable::SegmentDamage& segmentDamage =
cableDamage->getAccumulatedDamageAt(segmentIndex);
agx::Real bendTensionDamage = segmentDamage.bendTension();
agx::Real twistDeformationDamage = segmentDamage.twistDeformation();
agx::Real frictionDamage = segmentDamage.contactFrictionForce();

for (size_t i = 0; i < agxCable::DamageTypes:NUM_CABLE_DAMAGE_TYPES; i++) {
  std::cout << '[' << i << "]: " << segmentDamage[i] << '\n';
}
```

20.4 Parameters

The cable damage model contains two types of parameters: weights and thresholds. The weights control how much each individual damage estimation source should contribute to the final total. The thresholds control the smallest deformation required for the deformation related damage estimation sources to take effect. The following code

shows an example of how to set some of these parameters. The damage estimation model section details the available weights and thresholds.

```
damage->setStretchDeformationWeight(100.0);
damage->setTwistDeformationWeight(30.0));
damage->setBendRateWeight(10);
damage->setTwistTensionWeight(0.04);
damage->setFrictionForceWeight(2.0);
damage->setStretchThreshold(0.0);
damage->setBendThreshold(0.1);
```

The patterns are `set[Bend|Twist|Stretch][Deformation|Rate|Tension]Weight`, `set[Normal|Friction]ForceWeight`, and `set[Bend|Twist|Stretch]Threshold`.

Changing the parameters will affect simulation steps taken after the change, but not damage estimates for time steps already completed. This makes it possible to alter the cable damage parameters over time to account for changes to the cable or the environment during the simulation.

The cable damage calculations and its parameters do not interact in any way with the cable material parameters and accumulated cable damage does not influence the simulation behavior of the cable.

20.5 Tutorial

The C++ tutorial *tutorial_cable_damage.cpp* shows more details on how to set up a scene with cable damage.

21 Creating stable simulations

Creating stable simulations is an art. It requires the juggling of a large set of parameters and attributes. This chapter will try to summarize a few of the most important things to keep in mind.

21.1 Masses

Depending on the selected solver (iterative or direct), the mass ratio between interacting bodies can become a problem. By default a hybrid scheme where all constraints including ordinary constraints and contacts are handled with the direct solver except for the friction of contacts which are handled by the iterative solver. The hybrid solver is able to handle large mass ratios in stacking and constrained systems.

For the iterative solver (when specifying constraints to be solved iteratively) it is very important to keep the masses in the system in the same range of order of magnitude. For example a rigid body of mass 0.1 which interacts (collides to, are constrained to) with one that weigh 100 might cause instabilities due to large mass ratios. In general (but especially when using the iterative solver), try to keep mass ratios as low as possible, or at least for interacting objects. Having objects with mass 1gram in the same simulation as objects with 100Tons is ok, as long as they don't interact with each other.

21.2 Damping

Damping a simulation can sometimes be necessary. One can add damping using velocity *damping* on rigid bodies:

```
RigidBody::setVelocityDamping()
RigidBody::setLinearVelocityDamping()
RigidBody::setAngularVelocityDamping()
```

Valid values are between 0 and any real value, 0 is no damping (default). The velocity damping is a viscous damping.

Another damping parameter available is the damping of contacts:

```
Material::getBulkMaterial()::setDamping( )
```

Damping is also important when using constraints. Assume for example that you have created a cylinder which is attached to the world with a hinge. If the body is un-damped, and the hinge does not have a motor enabled (or a lock), nothing will stop the cylinder from rotating if it ever would start to rotate.

In real life, there is always friction in bearing, air-resistance etc. that will add "damping" to a system. This is something to keep in mind when you want to create stable simulations.

Use the functions in `agxUtil::convert` in order to convert from the viscous damping coefficient to AGX' damping coefficient. See 15.1.1 for more information.

21.3 Time step

The time step or *dt* used when stepping a simulation forward is very important detail for getting stable simulations. First of all, to get stable simulations one should always use consistent time steps, that is, always stepping the simulation forward with same *dt*.

Stepping a simulation with different *dt* might introduce energy and cause instabilities. Always strive to use the shortest possible *dt*, the shorter, the more stable simulations. Collision detection relies on that all collisions can be caught during one step. So if a geometry is moving fast relative to its size, or collides with a relative small geometry, penetration, or *tunneling* effects can occur. This will result in large penetrations or even missed collisions. Large penetrations will create large forces that will try to restore the penetrated geometry outwards.

21.4 Size of objects

If objects are small in comparison to their velocity (and the used time step), they can cause deep penetrations and missed collisions. Try to keep an eye on the size of geometries so they are not too small. It's all relative to the time step and the velocity, so there is no right or wrong here.

21.5 Material attributes

Try to reduce the number of materials so that you have control over which combinations that can occur.

For example, if you have 100 objects (geometries) in your simulation and each geometry have different materials. This will lead to that you have ~100^2 combinations of materials. This is because for each pair of colliding geometries, a *ContactMaterial* is created from the two geometries materials. If you don't have control over your Materials, then you can get undesired behavior.

21.6 Velocities

Try to keep the velocities of objects moderate. Too high velocity in relation to time step and geometry sizes can cause penetrations.

21.7 Valid initial states

When creating a simulation, it is important that the initial state, which is how geometries are transformed relative to each other, is valid. An example of an invalid state is when two geometries are positioned initially in contact. This can result in an "exploding" simulation, where the solver will try to push the two interacting geometries/bodies out of each other.

Another example of an initial state that can cause unstable simulations is when using constraints. Assume you have created a rigid body, this rigid body is also attached to a constraint. If this body is moved in such a way that it violates the constraint, *after* the constraint has been initialized, the solver will try to move the body so that the constraint stops being violated. For example, translating a body after it has been attached to a hinge and the world (only one rigid body used in the constraint.

22 Parallelization

In Figure 9 we could see that some of the items were marked with an asterix (*). This indicates that the subtask is a potential target for parallelization. In this section the various tasks will be dissected in more detail.

AGX is built upon the notion of *tasks*. A task can depend on other tasks and can also have sub-tasks. A Task which does not depend on other tasks can run in parallel with other tasks.

For example the class DynamicsSystem is built upon several tasks and subtasks. Such as “UpdateWorldMassAndInertia”, “IntegrateVelocity”, etc. Many of these tasks will split up the work and execute in parallel.

Another important feature of AGX is that all of the critical data for rigid bodies, shapes etc. are stored in *buffers*. These buffers are memory allocation blocks, aligned in memory appropriately for SSE optimization. Also, the memory can be made available for other implementation platforms such as OpenCL or CUDA. The executional part of a task is called a *kernel*. It can have several implementations, SSE, non-SSE, OpenCL, OpenGL etc. Which type of implementation that is needed, is selected when the task/kernel is initialized. A kernel is a small executional unit which operates on indata and supplies the result as out-data. It operates directly on the buffers for the fastest possible data access. This schema makes it possible to have some kernels executing on the graphics cards (for example OpenCL kernels), and some on the CPU.

22.1 Threads

In AGX there is a pool of threads used for all threaded jobs. The size of this pool is controlled with the call:

agx::setNumThreads(int n);

A negative value for *n* means that one thread is created per Core/CPU. The default value is 1. All parallelizable tasks are job-oriented. All jobs are put into a queue and scheduled for execution.

22.1.1 Executing AGX in threads

AGX creates and manages its own threads in a thread pool. However, to use the AGX API you need to be aware of a few things:

- Any thread that calls the AGX API need to be registered as an AGX thread. This is because various resources need to be available for each thread. To promote a thread to be an AGX Thread call:

  ```
  agx::Thread::registerAsAgxThread();
  ```

  To unregister a thread, call:

  ```
  agx::Thread::unregisterAsAgxThread();
  ```

- Callbacks from AGX such as contact events (12.4.3) or from event listeners (12.4.6) will always be done to the *main thread* for an agxSDK::Simulation. This is the thread that created the agxSDK::Simulation object. To make a thread the main thread for a simulation call (after a call to registerAsAgxThread()):

  ```
  simulation->setMainWorkThread( agx::Thread::getCurrentThread() );
  ```

For an example of this, see tutorial_threads.cpp

22.2 Parallel tasks

Table 20 below; show some of the tasks which are parallelizable. These tasks will however not run in parallel to each other, as they depend on data from the previous task. Compare to the time line for the call to Simulation::stepForward() (Figure 9).

Name	Description	Belongs to
NarrowPhase	Calculates contact data between two overlapping Geometries.	agxCollide::Space
Update bounding volumes	Update bounding volumes for geometries.	agxCollide::Space
ApplyGravity	Add gravitational force to bodies.	agx::DynamicsSystem
Solver	Solve the constraint system.	agx::DynamicsSystem
IntegratePositions	Integrate transformation and calculate acceleration.	agx::DynamicsSystem
IntegrateVelocity	Integrate Velocity	agx::DynamicsSystem
UpdateWorldMassAndInertia	Will calculate various items needed for the solver.	agx::DynamicsSystem

Table 20: Parallelizable tasks.

The Solver stage is only parallelizable if the partitioner can create disjoint groups of bodies which can be solved independently from each other.

22.2.1 NarrowPhase

This task will be given a list of overlapping bounding volumes (from the broad phase) calculate the exact contact data for two possibly overlapping geometries. This is a trivially parallelizable task, as there are no data dependency between the different overlapping geometries.

22.2.2 Update bounding volumes

This task will update the current bounding volume for a geometry (including its shape transformation and size).

Depending on the frame hierarchy, this task can occupy quite some time for a large number of geometries. There are no data dependencies between the geometries. It can therefore be parallelized.

22.2.3 Partitioner

Based upon the connectivity in the whole dynamics system, the partitioner can split the system into independent sub-systems.

Two bodies are connected if there is a constraint between them. Constraints including contacts can create large trees of interconnected bodies.

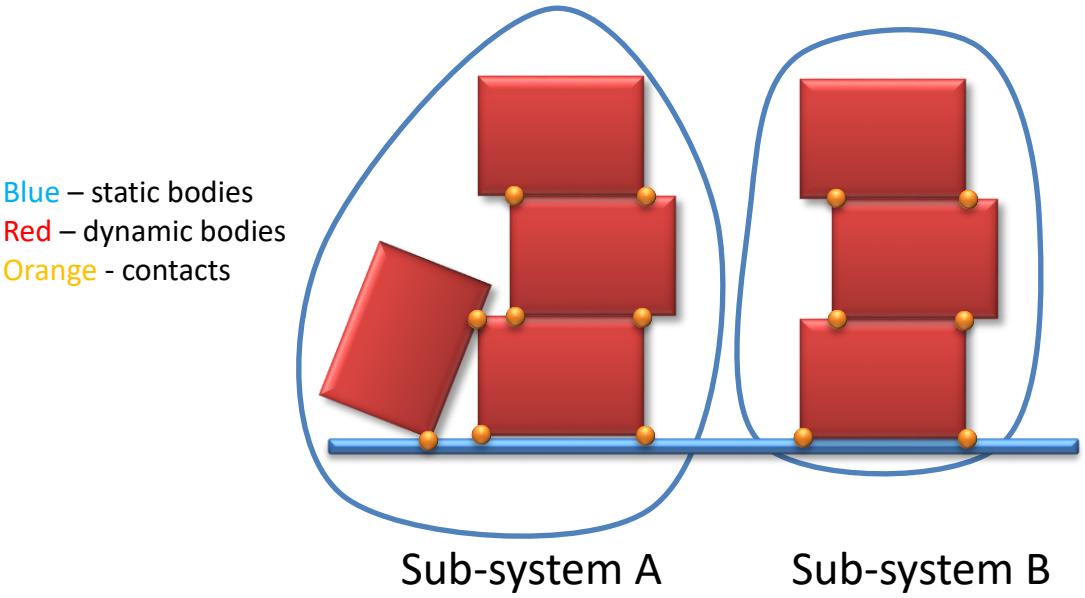


Figure 42: Two separate subsystems.

In the figure above, two separate systems can be identified which can be solved independently from each other, hence the above system would gain if the solver would run the two systems in one thread each. As soon as there is a connection (through a constraint/contact) between two systems, they are merged.

Kinematic bodies will analogous to static bodies split a system into two parts allowing for parallelization.

23 Surface Velocity Conveyor Belt

The class *agx::SurfaceVelocityConveyorBelt* can be used in order to simulate a basic conveyor belt. No internal mechanics in the conveyor belt are simulated, only velocity changes in contacts between conveyor belt and transported goods.

As the name indicates, the SurfaceVelocityConveyorBelt uses the surface velocity property inherent in contact points, which is described in chapter 13.6.6.

For simple cases, if might suffice to use an *agxCollide::Geometry*'s surface velocity. However, the latter is constant over the whole surface of the Geometry. For cases like in a conveyor belt, one wants the velocity to follow around the surface of the geometry which might be more complex (going up and down, …).

A SurfaceVelocityConveyorBelt has different surface velocity directions for different parts of the belt, but the length - or rather "speed" - remains the same. The different directions are computed from a list of points along the surface of the belt in direction of its movement - the single directions are computed from one point to the next.

For a given contact point, the closest line segment from one conveyor belt point to the next is chosen for the surface velocity. This might give wrong behavior in more advanced concave cases, which is a limitation of the model. For “typical” conveyor belt cases however, it should work fine.

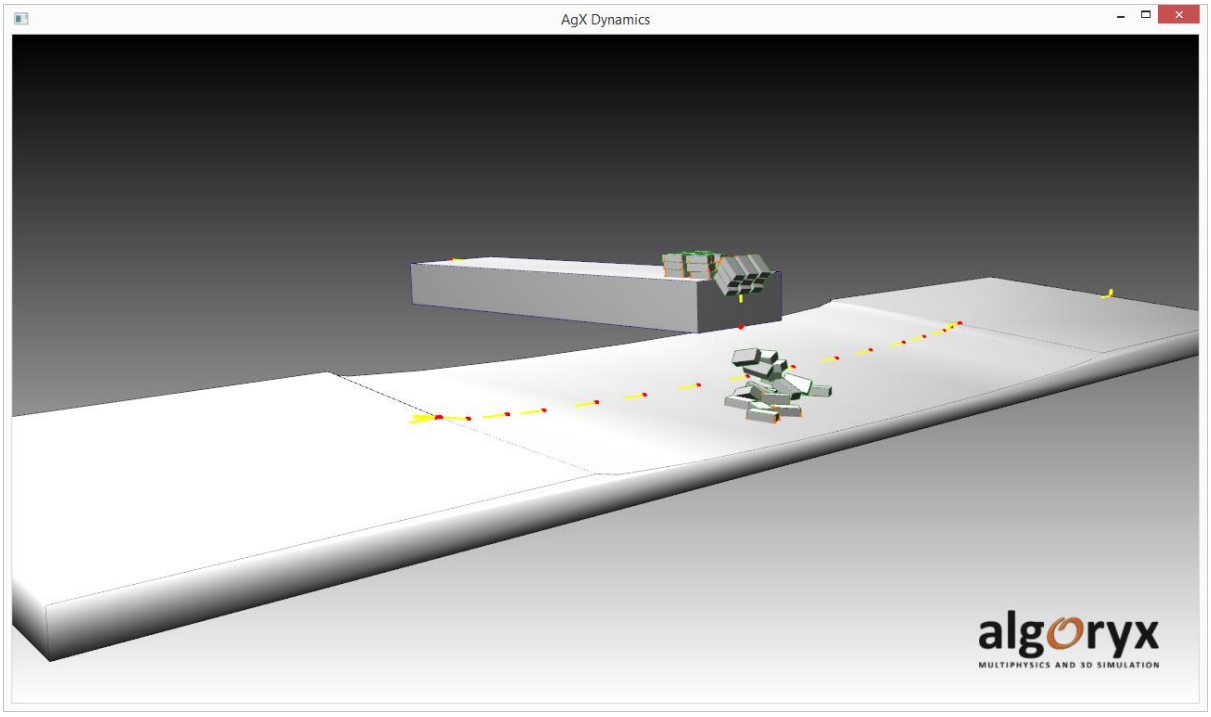


Figure 43 - Two conveyor belts. Ret dots are points, yellow lines indicate direction.

Implementationwise, the *agx::SurfaceVelocityConveyorBelt* inherits from *agxCollide::Geometry* and can be used as a geometry. It is constructed from a list of points, such as in the following example:

```
#include <agx/SurfaceVelocityConveyorBelt.h>

void createConveyorBelt(agxSDK::Simulation* sim)
{
  agx::Vec3Vector points;
  points.push_back(agx::Vec3(-1, 0, 0));
  points.push_back(agx::Vec3(0, 0, 1));
  points.push_back(agx::Vec3(1, 0, 0)); // Change in direction, from upwards to downwards.
```

```
  agx::SurfaceVelocityConveyorBeltRef belt = new agx::SurfaceVelocityConveyorBelt(points);
  sim->add(belt);
  // We should also add shapes to the conveyor belt, and they should map the points.
}
```

Given n points, the surface velocity conveyor belt will create n-1 directions. This allows for open connection. If a closed loop is desired, then the first point should be added another time as the last one, as in this example:

```
  points.push_back(a);
  points.push_back(b);
  points.push_back(c);
  points.push_back(d);
  points.push_back(a); // Close the rectangle loop.
```

The speed of a conveyor belt can be set by a call to `void setSpeed(agx::Real speed)`, where negative speeds will indicate a change in direction.

Debug rendering can be activated for a conveyor belt by a call to `bool enableDebugRendering(agxSDK::Simulation* sim)` (and disabled with a call to `bool disableDebugRendering()`). It will look like the red dots and yellow lines in Figure 43.

When creating an agx::SurfaceVelocityConveyorBelt for an agxCollide::Trimesh, one can cut the trimesh with a plane in order to obtain an ordered list of points. This list of points can be used as input to the constructor for the conveyor belt, instead of having to specify them all by hand. However, to close the loop, the points' first point should be added once more as its last, as in this example:

```
#include <agxCollide/Trimesh.h>
#include <agx/SurfaceVelocityConveyorBelt.h>

agx::SurfaceVelocityConveyorBelt* createConveyorBeltFromTrimesh(agxCollide::TrimeshRef
trimesh)
{
  auto plane = agx::Plane(agx::Vec3(0, 1, 0), 0);
  // This will increase and decrease the trimesh's ref count - make sure you have a ref
pointer to it.
  auto points = agxCollide::intersectTrimeshWithPlane(trimesh, plane,
agx::AffineMatrix4x4::translate(trimesh->getCenter()));
  if (points.size() == 0)
    return nullptr;
  points.push_back(points.front()); // Close the loop.
  auto belt = new agx::SurfaceVelocityConveyorBelt(points);
  belt->add(trimesh);
  return belt;
}
```

24 AgXModel

AgXModel is a library consisting of high level primitives which implement various simulation objects: trees, beams, terrain etc. They are built upon the basic construction elements found in the agXPhysics library: constraints, geometries, bodies and event listeners.

24.1 Beam

agxModel::Beam is a model used for simulating beam shaped objects with elastic and plastic properties. This could be used to simulate beams and poles that needs to behave realistically under pressure or impact.

The Beam model uses several rigid bodies that are constrained together with parameters calculated from the material. These bodies are also called “lumped elements”, since this approach of modelling is referred to as the “lumped element method”. The Beam class controls the constraints and sets their frames to make the beam deform. tutorial_beam.cpp gives a complete picture of the functionality found in the beam class.

Example of a simple plank shaped beam:

```
agx::Vec3 start(-2, 0, 1);
agx::Vec3 end(2, 0, 1);
agx::Real height = 0.4;
agx::Real width = 0.1;
agx::UInt32 numElements = 100;
// Create a new beam at a given start/end position and of the specified size.
agxModel::BeamRef boxBeam = new agxModel::Beam(start, end, height, width, numElements);

// Add it to the simulation
simulation->add( boxBeam );
```

The beam will however not possess any beam specific properties until the material is set with two specific properties (bulk Young’s modulus and Poisson’s ratio) as well as two yield parameters.

```
// Aluminum parameters
agx::Real youngsmodulus = 69e9;
agx::Real poissonsRatio = 0.35;
agx::Real yield = 5e7;
agx::MaterialRef beamMaterial = new agx::Material( "beamMat" );

beamMaterial->getBulkMaterial()->setYoungsModulus(youngsmodulus);
beamMaterial->getBulkMaterial()->setPoissonsRatio(poissonsRatio);

// use same yield for both regular rotation (first)  and torsion ( second )boxBeam-
>setMaterial(beamMaterial, yield, yield);
```

To get a better performing Beam we can turn off internal collisions.

```
agx::UInt32 beamID = simulation->getSpace()->getUniqueGroupID();
boxBeam->addGroupID(beamID);
simulation->getSpace()->setEnablePair( beamID, beamID, false );
```

24.1.1 Breakable

The Beam object can also be breakable. To make the beam breakable a threshold must be set and a BranchEventListener implementing the function onHighLoad must be

added. The function onHighLoad will be called giving the constraint with the most deformation once constraints have reached the threshold.

Below is an example of creating the BranchEventListener.

```
class MyBranchEventListener : public agxModel::Tree::BranchEventListener
private:
  agxSDK::Simulation *m_sim;
public:
  MyBranchEventListener(agxSDK::Simulation *simulation)
     : m_sim(simulation)
{}
  virtual void onCreate( agxModel::Tree::Branch* branch ){}
  virtual void onHighLoad(agxModel::Tree::Branch *branch)
  {
    agxModel::Beam *oldBeam = static_cast<agxModel::Beam
*>(agxModel::Tree::BranchEventListener::getTree());
    agxModel::Beam *newBeam = oldBeam->cut(segment);
    agx::ref_ptr<MyBranchEventListener> newListener = new MyBranchEventListener(m_sim);
    newBeam->setBranchEventListener(newListener);
  }
virtual MyBranchEventListener* clone() { return new MyBranchEventListener(m_sim); }
protected:
  virtual ~MyBranchEventListener() {}
};
```

This breaking will create an additional beam apart from the one previously created. The new beam needs to have a different BranchEventListener.

The BranchEventListener can then be added to the beam by setBranchEventListener:

```
boxBeam->setBranchEventListener(new MyBranchEventListener(simulation));
```

This beam will break each constraint that passes the deformation threshold. The new beam parts that are created when it breaks will not collide with each other. To get a more useful breakable property we reset the deformation on the not-yet-broken constraints and separate the collision groups for the old and new beam.

In onHighLoad, add:

```
// Sets the new beam so it can collide with the old
newBeam->removeIDs();

// Resets the deformation on all constraints to zero
agxModel::Beam::Segment *tempSegment =oldBeam->getRoot();
while(tempSegment) {
  tempSegment->setDeformation( tempSegment->getDeformation() * 0 );
  tempSegment = tempSegment->getChildSegment();
}
```

```
tempSegment = newBeam->getRoot();
while(tempSegment) {
  tempSegment->setDeformation( tempSegment->getDeformation() * 0 );
  tempSegment = tempSegment->getChildSegment();
}
```

24.1.2 Known limitations

High detailed beams can have several really thin objects constrained together, which when colliding with other objects or other beams can collide “in to” the other object. This can break the simulation by giving the beam to much force in a time step.

24.2 Terrain

The agxModel::Terrain implementation is a high level API that can simulate the behavior of deformable ground. The implementation covers four different types of interaction with the ground.

1. The default behavior when a rigid body collides with the agxModel::Terrain is compression of the ground.
2. A geometry could have a property telling the terrain that it is a "**deformer**", which will make the ground volume conserving, and push the ground sideways.
3. A geometry could also have the property of being a "**shovel**", which allows to convert volume from the terrain to particles.
4. A geometry could also be both a "deformer" and a "shovel" which improves the digging experience.

24.2.1 Physical model

The height field must have symmetric scale. I.e. the distance between vertices in x-direction must be the same as in y-direction. An agx::LOGGER_WARNING will tell you if you have a bad configured height field scale.

The implementation is done so that it is only valid to have the height field z axis in the negative gravitational direction.

The terrain can be created using one height field shape, or one geometry that has only one shape. A height field, with no transform relative to the geometry.

```
agxModel::Terrain( agxCollide::HeightField* heightField );
agxModel::Terrain( agxCollide::Geometry* terrainGeometry );
```

24.2.1.1 Colliding

An agxModel::Terrain contains one geometry with one shape, an agxCollide::HeightField. The user may set a material map to each vertex of the terrain using a PNG picture:

```
agxModel::Terrain->getDataInterface()->setMaterialMapping(const std::string& filePath)
```

The file will contain indices to materials that are supposed to be active at certain material indices. User must connect materials to material indices:

```
agxModel::Terrain->getDataInterface()->add(agx::Material* material, size_t materialIndex)
```

This call to add could also set three other Terrain specific material properties, particleAdhesion, deformability and angleOfRepose (these are zero by default and will be explained later).

```
agxModel::Terrain->getDataInterface()->add( const agx::Material* material, const size_t
materialIndex, const agx::Real particleAdhesion = agx::Real(0), const agx::Real deformability
= agx::Real(0), agx::Real angleOfRepose = agx::Real(0) );
```

24.2.1.2 Digging

To be able to dig into the terrain some geometry must act as a shovel (or bucket). When a geometry with the two properties "*ShovelUpProperty*" and "*ShovelForwardProperty*" collides with the terrain, it will be considered being a shovel. If there is any volume from

the terrain shape above (in height field coordinates) that volume will be removed from the height field and converted into particles.

The length of the shovel forward property will define the distance from the shovel geometry shape center to the tip of the shovel. The length of the shovel up property will define the thickness of the shovel when trying to cut through the ground. It should be orthogonal to the shovel forward direction, and point “up in the air” when the shovel is resting on the ground.

24.2.2 Configuration

There are several variables exposed for the configuration of the Terrain interaction. First, the general variables will be explained that will be used when any geometry is interacting with the terrain. Then the shovel specific variables are explained, and at last the variables that control the dynamic height field behavior.

24.2.2.1 Compression: Deformability

For a material which is added to a material index (in the material index map), the parameter *deformability* can be set. The deformability specifies how fast the ground gets harder when being compressed. For each delta length, dl, the ground is compressed, it will become exponentially harder (increased Young’s modulus) according to the formula:

youngsModulusMultiplier = (1/2)* 2 ^[(deformability + 1)*compression + 1]

The Young’s modulus for the local contact point will be multiplied with the youngsModulusMultiplier. The ground can’t become harder if the deformability is zero. And it will become harder exponentially faster with increased deformability.

24.2.2.2 Compression: Max compression multiplier

The ground will become harder and harder the more it gets compressed (unless the deformability is zero). The max compression multiplier specifies the upper limit for the compression multiplier. This scalar value should be a multiple of 2, since the log2 of the multiplier has to be calculated.

```
agxModel::Terrain->getDataInterface()->setMaxCompressionMultiplier(Real multiplier)
```

24.2.2.3 Compression: Depth for max compression

The deeper you dig, the more compressed the ground is, and the compression at a vertex will increase not only by compression, but also by removing volume (shovel or deformer can do that). So when digging into the ground, the ground will become harder and harder until you have reached the depth of max compression. At this depth and deeper the ground will be as hard as the max compression multiplier (and the young’s modulus of the contact) allows to.

```
agxModel::Terrain->getDataInterface()->setDepthForMaxCompression(Real depth)
```

24.2.2.4 Compression: Max Young’s change per time step

When objects are moving at a high velocity (in relation to the time step) the contact depths could be large, which could result in large deformations. The max young’s change per time step parameter could be set to control how compressed (and therefore how much harder) the ground can become during one time step.

```
agxModel::Terrain->getDataInterface()->setMaxYoungsChangePerTimestep( Real
maxMultiplierChange)
```

24.2.2.5 Compression: Lock boarders

The terrain can have fixed height at the boarders. It is useful if the terrain is for example a pile of dirt on the flatbed of a truck.

```
agxModel::Terrain->setEnableLockedBorders( bool enabled)
```

24.2.2.6 Digging: Overall

The terrain will create particles when digging. The materials for the particles are set up by the terrain. Also, the contact materials between particles and between particles and explicitly specified external materials.

A shovel does not have to be a “deformer”, but it is recommended.

24.2.2.7 Digging: Particle creation

The particles will be created in a FCC pattern. The particles will overlap with (some of) their neighbors with exactly:

adhesiveOverlap = (radius*sqrt(2)-radius).

The radius of the particles should be no more than the scale of the height field. Otherwise it is very hard to create them without overlapping the height field too much.

24.2.2.8 Digging: Particle radius

Set the particle radius to the particle attributes.

```
agxModel::Terrain.getParticleAttributes()->setParticleRadius( Real radius )
```

The maximum accepted particle radius can be found from:

```
agx::Real agxModel::Terrain->getMaximumParticleRadius();
```

24.2.2.9 Digging: Particle adhesion

The particles can act as a lump of dirt or as granular material. The particles should be constrained somehow to simulate a lump. This is done by setting an adhesive force (using the defined adhesive overlap, (see Digging: Particle creation) between particles. It is done in the call where you map a material to a material index. That material will then have the specified adhesive force between particles when the region of the height field with that specific material is converted to particles.

24.2.2.10 Digging: External material

The interaction between particles and other materials will become very soft (as soft as the ground material is). Between the shovel and the particles, it is important that the contacts are quite hard, otherwise they will easily fall through the shovel. Terrain controls the contact material between particles and materials added to the terrain as external materials. The Youngs modulus of these contacts will be the largest Youngs Modulus value of the external material and the material of the terrain where the particle is created.

```
agxModel::Terrain->getDataInterface()->addExternalMaterial( agx::Material* material );
```

24.2.2.11 Digging: Particle Young’s multiplier

The stiffness of the contacts between particles could be scaled with the particle Young’s multiplier. If this multiplier is 1, the particles will have the same Young’s modulus as the material of the terrain where the particle is created (which could be very soft).

```
agxModel::Terrain.getParticleAttributes()->setParticleYoungsModulusMultiplier(Real multiplier)
```

24.2.2.12 Digging: Shovel min angle to horizontal plane

When the shovel cuts through the ground, particles can be created. The terrain logic only allows particles to be created when the shovel normal vector has a component in the negative gravitational direction (the shovel is NOT upside down). When the shovel is not upside down but very close to point (with the shovel forward vector) straight down into the terrain, the volume where particles can be created is just a crack in the ground. It could be (depending on the radius of the particles) that no particle fit in that crack and therefore impossible create particles at valid positions. Buy setting the shovel minimum angle to the horizontal plane for particle creation, the particles are not created until the angle between the shovel and the gravitational direction is big enough.

24.2.2.13 Digging: Shovel ground cutting force

If the shovel is both a deformer and a shovel and the interaction when the shovel is deforming and pushing soil is satisfactory, but it is too easy to dig, a constant friction force in the shovel forward direction can be added. This can also be changed in runtime, to simulate for example a full shovel/bucket that can't dig as easily as when it is not full.

```
agxModel::Terrain->getDataInterface()->setShovelGroundCuttingForce( Real force )
```

When the shovel is not a deformer, this value is used as a friction coefficient scale, not a constant force limit.

24.2.2.14 Digging: Particle friction solve type

The contacts between particles is default solved iteratively which could reveal a soft behavior, compared to solving the contacts with the direct solver. By setting the particle solve type to agx::FrictionModel::DIRECT the particles will behave stiffer when they are compressed. However, using a DIRECT solve type will have a negative impact on performance.

```
agxModel::Terrain->getDataInterface()->setParticleFrictionSolveType( agx.FrictionModel.DIRECT
);
```

To keep reasonable computational speed, we must make sure that the direct solver does not end up in any LCP iterations due to the particles. Therefore, the friction and the adhesion must be to infinite, also the adhesive overlap must be zero. See 13.14.4 ContactMaterial.

The surface material viscosity for the particles is controlled by the inverse friction multiplier.

```
agxModel::Terrain->getDataInterface()->setInverseFrictionMultiplier(Real inverseMutiplier)
```

The friction forces will become larger if the inverseMultiplier is small. It is an inverse multiplier since it should be considered as a scale of compliance for the friction constraint.

24.2.2.15 Dynamic Height field: Functionality

The dynamic height field functionality must be enabled for the terrain for it to treat contacts with geometries with the "DynamicHeightFieldDeformer" property different from others.

```
agxModel::Terrain->setEnableDynamicHeightField( bool enable )
```

An enabled dynamic height field will also trigger avalanches where the height field has larger angles than the angle of repose.

A geometry with the “DynamicHeightFieldDeformer” property will always have the property, weather the bool value is set true or false.

```
agxCollideRef geom = new agxCollide::Geometry();
bool useContactDepth = true;
geom->getPropertyContainer()->addPropertyBool("DynamicHeightFieldDeformer", useContactDepth)
```

The bool *useContactDepth* enables the possibility for the contacts with the terrain to push the object out from the terrain. If *useContactDepth* is set false the contacts will always have zero depth. For buckets, you usually like to set this Boolean to false. For objects lying on the ground, you like to have it set true.

The dynamic height field functionality will create two new geometry contacts between the terrain and the colliding geometry. One in the gravitational direction, and one in the horizontal plane. The contact points from the original geometry contacts will be used and projected to these directions. The Young’s modulus will be different between the two directions, and dependent on the contact surface area in each direction. The original contact will be very soft and handle the friction for the interaction.

24.2.2.16 Dynamic Height Field: Cut off velocity and angular velocity

Deformation is only active near geometries moving faster than the cut off velocity or rotating faster than the cut off angular velocity.

```
agxModel::Terrain->setDynamicHeightFieldCutOffVelocity(Real cutOffVelocity );
agxModel::Terrain->setDynamicHeightFieldCutOffAngularVelocity(Real cutOffAngularVelocity );
```

24.2.2.17 Dynamic Height Field: Vertical Young’s modulus area scale

```
agxModel::Terrain->setVerticalYoungsModulusAreaScaler(Real scale)
```

The stiffness (Young’s modulus) of the geometry contact acting in the vertical/gravitational direction is a multiple of the Young’s modulus for the defined contact material.

This value is defined for a percentage of a unit area. The minimum value will be the Young’s modulus for the contact material. The actual value for stiffness for a geometry contact will then be scaled with the area of the contact overlap. The larger area, the stiffer contact.

For example, if the scale is set to 0.01 and the contact area is 1 square meter, the contact will become 100*YoungsModulusForContactMaterial.

24.2.2.18 Dynamic Height Field: Horizontal Young’s modulus area scale

The horizontal scale works precisely as the vertical, but here we must consider a contact that represents soil that can be pushed sideways. This contact will also have zero contact depth, otherwise there could be a lot of states where contact point push in opposite directions that causes the solver to fail. This value must be found experimentally

for a specific scenario, but a rule of thumb is that it should be lower that the vertical scale, since the contact depth is zero.

```
agxModel::Terrain->setHorizontalYoungsModulusAreaScaler(Real scale)
```

24.2.2.19 Dynamic Height Field: Avalanche

The dynamic height field will trigger avalanches where geometries have been deforming the terrain and where particles have merged. There is an avalanche step life time that defines for how long (in time steps) a specific vertex will have avalanches triggered around it.

```
agxModel::Terrain->setAvalancheStepLifeTime(size_t numSteps)
```

There is no dynamics of these avalanches included in any solver, which leads to pure kinematic changing of the heights. A way to control how fast soil will fall is to set the avalanche decay percent. This is a value between 0-0.5 (0-50%). At 0 there will be no avalanches. At 0.5, half of the local height difference between two vertices will be transferred so that the resulting angle is the angle of repose (explained below).

```
agxModel::Terrain->setAvalancheDecayPercent( Real decayScale )
```

The avalanche can be set to be triggered in a local neighborhood of an interaction between a “deformer” and the terrain. The number of vertices away from the involved vertices from the interaction can be set.

```
agxModel::Terrain->setAvalanceVertexDistanceExtent( size_t numVertices );
```

24.2.2.20 Dynamic Height Field: Angle of repose

For a specific pile of granular matter, the angle of the pile will be material specific, i.e. the angle of repose. This is set in the same call where a material is mapped to a material index.

The dynamic height field logic will have the same angle of repose for any compression, when the material is dry. For a wet material, the angle of repose will increase linearly to compression (relatively to max compression). The wetness is possible to set between 0-1.

```
agxModel::Terrain->setWetnessScale( Real scale )
```

24.2.3 FAQ

24.2.3.1 Why can’t I dig?

- Make sure the only geometries on the shovel/bucket that can collide with the terrain has the “ShovelForwardProperty” and the “ShovelUpProperty”.
- Make sure that the contact material between the shovel geometry(ies) and the terrain material is set up correctly hand has a rather small Youngs modulus.
    - Also make sure that the terrain material is added to the material map at that specific area of the terrain.
- Why do the particles fall through the shovel/bucket?
- Make sure that the vertical (and horizontal) Young’s modulus scale is not too small.

24.2.3.2 Why do the particles fall through the shovel/bucket?

- Make sure you have added the material of the shovel geometry as an external material.
- Make sure that the material of the shovel geometry is correct. (Youngs modulus that corresponds to steel for example)

24.2.3.3 Why are there pointy peaks in the terrain after I have deformed it with my shovel with “deformer” property?

- Make sure that the angle of repose of the material is reasonable. If you have not defined it, it will become atan(friction_coefficient) for that material.
- If the material is wet, the angle of repose could be quite large when compressed. Consider changing the max compression multiplier, the wetness or the angle of repose.

24.2.3.4 Why are the particles created tunneling through the shovel?

- Make sure you have added the material of the shovel as an external material to the terrain.

24.2.3.5 Why is there no resistance when soil is pushed sideways using a geometry with “deformer” property?

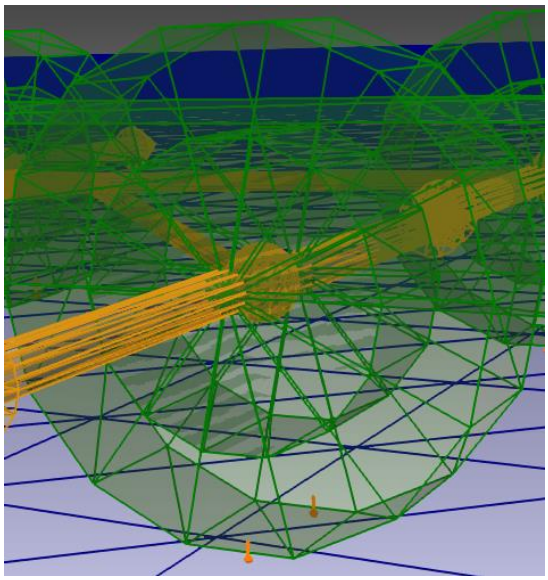
- The horizontal Young’s modulus area scale control the hardness of the contacts in the horizontal direction. A lower value will give stiffer contacts.

24.2.3.6 Why does the shovel sink through the ground?

- Make sure that the vertical Young’s area scale is not too small.

24.3 TireModel

agxModel::TwoBodyTireModel is an implementation of a tire model with two separate rigid bodies; the tire and the rim. The user has control over the deformation of the different degrees of freedom by setting stiffness and damping:



Creating a TwoBodyTireModel requires two bodies, one for the rim and one for the rigid body. It is assumed that the two bodies are positioned and added to the simulation before or after the creation of the TwoBodyTireModel

```
agx::Real tireRadius = 1;  // Not important, but can give information of
agx::Real rimRadius = 0.5; // amount of deformation of tire.
agx::Real stiffness = 2E6;

agxModel::TwoBodyTireModelRef tire =
      new agxModel::TwoBodyTireModel(tireBody, tireRadius, rimBody, rimRadius );
```

24.3.1 Stiffness

The stiffness of the tire will control the amount of deformation for the tire. There are four degrees of freedom for which the stiffness can be specified.

The unit for translational stiffness is **force/displacement** (if using SI: N/m). The unit for rotational stiffness is **torque/angular displacement** (if using SI: Nm/rad)

```
// Configure the deformation stiffness for all DOF of the tire
tire->setStiffness(stiffness*2, agxModel::TwoBodyTire::RADIAL)
tire->setStiffness(stiffness*10, agxModel::TwoBodyTire::LATERAL)
tire->setStiffness(stiffness*4, agxModel::TwoBodyTire::BENDING)
tire->setStiffness(stiffness*0.1, agxModel::TwoBodyTire::TORSIONAL)
```

24.3.2 Damping

The damping corresponds to the rate of which the deformation should be restored. It has the same degrees of freedom as the stiffness.

The unit for translational damping is **force * time/displacement** (if using SI: Ns/m). The unit for the rotational damping coefficient is **torque * time/angular displacement** (if using SI: Nms/rad)

```
local dampingCoefficient = 70000

tire->setDamping(dampingCoefficient*2, agxModel::TwoBodyTire::RADIAL)
tire->setDamping (stiffness*10, agxModel::TwoBodyTire::LATERAL)
tire->setDamping (stiffness*4, agxModel::TwoBodyTire::BENDING)
tire->setDamping (stiffness*0.1, agxModel::TwoBodyTire::TORSIONAL)
```

25 agxPowerLine

In AGX Dynamics, a power line represents a collection of components that transport power across a system, typically from some kind of source to a set of consumers. Examples of power lines are a drivetrain moving torque from an engine to the wheels of a car or a truck, and a hydraulic circuit converting pressure from a pump into linear motion of a hydraulic cylinder.

Despite the name the structure of a power line does not need to be a simple line, but arbitrary graphs are supported. Using converter components, it is also possible to mix hydraulic and drivetrain components in the same power line.

Using the AGX PowerLine library it is possible to share a power source among several consumers and thereby include the various system failures that can occur due to insufficient available power in the simulation.

25.1 Design philosophy

The most basic building blocks of the AGX power line is a one-dimensional body that represent the motion within the constituent components. For a drivetrain the motion is the rotation of the various shafts and gears, and for a hydraulic system it is the motion of fluid through pipes and valves. The toolkit can also represent linear motion. We call these the three dimension types. The one-dimensional bodies are connected together when the power line components are connected, which causes motion in one body to affect the neighboring bodies. The power line itself imposes no limitation in how components are connected, arbitrary graphs are allowed, but particular hydraulic or drivetrain components may limit how that particular component may be connected to other components.

Every component has an input and an output side. The sides may differ both in the number of connections that the side allows, and the type of dimension that it may connect to. The sides at which two components are connected define the direction of motion where output-to-input is the default and corresponds to positive-to-positive motion. Connecting output-to-output or input-to-input causes the two components to move in opposite local directions.

The power propagation can cross between the one-dimensional power line world and the six-dimensional mechanical world at specified transition points. A transition point is a pair consisting of an AGX constraint such as a hinge or a prismatic and a type of power line component called an Actuator.

The various components that make up a power line are owned by a particular PowerLine instance, which in turn is owned by the Simulation. To simulate a power line one first creates the PowerLine object and a collection of components. The components are added to the PowerLine and connected together, while the PowerLine is added to the Simulation.

25.2 Component types

The components that take part in a power line in AGX are of one of two types: Unit or Connector. Every class representing a drivetrain or hydraulic component inherit from one of these two classes, either directly or indirectly. The two types are fundamentally different internally, but are used in similar ways. The create-add-connect pattern applies to both, but see the section titled “Connecting components” for more details.

25.2.1 Unit

A Unit holds the one-dimensional bodies that represents motion within the power line. The bodies are actually owned by instances of subclasses of the PhysicalDimension class; either RotationalDimension, TranslationalDimension or FlowDimension depending on which dimension type the Unit contains. The PhysicalDimension exposes the rotational-, translational or flow velocities in the component in a uniform, one-dimensional way. This is called the value of the PhysicalDimension. The acceleration within the component is called the gradient of the PhysicalDimension. A PhysicalDimension also has a mass property that relates the load type of the dimension, e.g., force, torque or pressure, to acceleration within the Unit. The mass property is simply the mass or inertia in the case of translational and rotational dimensions, but more complicated for a hydraulic Unit.

The Unit defines connection points into the Unit. A connection point is a PhysicalDimension that other components may connect to. A Unit may have multiple connection points and may return different PhysicalDimension instances depending on whether the connection concerns the input or output side.

25.2.2 Connector

A Connector represents transfer of power between Units. Whenever two or more Units are connected a Connector is used to link them together. It does this by creating a constraint between the bodies held by the Units. The specifics of the constraint defines the operation of the Connector and the Connector may provide configuration options that are used to control the constraint parameters. The configuration options may have been given component-specific names in order to be more domain oriented.

Much like the Unit, the Connector defines connection points for its inputs and outputs. A connection between a Connector and a Unit can only be made if there are matching connection points for the sides being connected. More on making connections in the section titled “Connecting components”.

25.2.3 Composite components

Some components are more complicated than what a regular Unit or Connector can represent. In such cases the toolkit provides composite components, which are Units and Connectors that contains other components. All components have a primary type, so even if a composite component may contain both Units and Connectors, the component itself is always either a Unit or a Connector and can be used as-if it was a regular Unit or Connector.

A composite Unit may be empty in the sense that it doesn’t contain any PhysicalDimensions of its own. Instead it contains internal Units that provide them. When asked for its connection point the Unit will return one of the PhysicalDimensions of the internal Units. A similar pattern may be used for Connectors and constraints.

Composite components commonly provide accessor methods for the internal components. Care should be taken when manipulating such components so that any invariants maintained by the composite component isn’t broken. In general, connect calls involving internal components are advised against.

25.2.4 Actuator

An Actuator is a Unit that forms a bridge between the one-dimensional power line and the six-dimensional mechanical world within the simulation. Examples of actuators are wheel axles on a vehicle and hydraulic cylinders on an articulated arm. Whenever there is

motion within the power line there will be a corresponding motion in the mechanical system, and vice versa.

Actuators are always created from a regular AGX constraint and the effect of the power line onto the bodies is the same as-if the Motor1D of the constraint had been used. Creating an Actuator for a hinge will cause rotation about the hinge axis and an Actuator associated with a prismatic will cause translational motion along the prismatic's axis.

The coupling is two-way, so the power source in the power line will feel the load from the mechanical bodies, including forces from other constraints, contacts and external forces such as gravity.

25.2.5 Connecting components

Components are connected using the various connect methods. The most basic form:

```
component1->connect(component2);
```

connects the *output* side of component1 to the *input* side of component2. The general rule is that a connection must contain both a Unit and a Connector, but for the simplest cases the Connector can be created implicitly. The system may also chose to reuse a Connector that is already present at one of the connection points.

To create more complicated power lines it becomes necessary to specify on which side of a component that a connection should be made. This is done by passing side flags to the connect methods. The following example connects the input side of component1 to the input side of component2.

```
component1->connect(UNIT_INPUT, UNIT_INPUT, component2);
```

Units contain PhysicalDimensions of various types and the Connectors declare, for each side, which type or types of PhysicalDimensions the Connector can connect to. For a connection to be successful there must be at least one match between the dimension types that the Connector expects and the types that the Unit provides. Connectors can also have a maximum number of allowed connections for each side. Attempting to connect a Unit to a Connector that has reached the maximum number of connections at the given side will fail.

Units have a notion of forward, or positive, motion. It is defined as the direction moved when the value of the Unit's PhysicalDimension is positive. The normal case has two connection possibilities. Connecting two Units at different sides, i.e., output-to-input or input-to-output, causes the direction to be maintained, and connecting two Units at the same side, i.e., output-to-output or input-to-input, causes the motion to be reversed. Some Connectors, such as a gear box with a reverse gear in, can change this behavior.

26 agxHydraulics

agXHydraulics is a library for modelling a hydraulic system. For code examples, look into the C++ (tutorials/agxOSG) and Lua tutorials (data/luaDemos/tutorials).

Hydraulics is the technology that deals with the generation, control and transmission of power using pressurized liquids. The AGX Hydraulics library allows for fast, accurate and stable simulation of hydraulics components in real time. It provides hydraulic components such as pumps, pipes, flow control valves, restrictions and actuators.

The hydraulics library is built upon the AGX PowerLine framework and designed to integrate with both the drive train and the mechanical system. It provides new types of Units and Connectors that can be inserted into the same power line graph as the Units and Connectors already available.

Power can be generated using an electric motor or combustion engine and transmitted through a drive train to a hydraulic pump which converts torque in the drive train to pressure in the hydraulics circuit. The pressure can then be transmitted through the hydraulics to actuators that interact with the mechanical system, or to hydraulic motors in order to transition back to a drive train.

Using the combined functionality of the power line, hydraulics and mechanics of AGX it is possible to model and simulate complex systems such as an excavator, a crane, an offshore winch or similar.

A number of quantities can be read from the simulation, including pressure at junctions and couplings, flow rates through pipes and other components, fluid compression in pipes and junctions, and torques at pumps and motors.

26.1 Components

Name	Type	Description
Accumulator	FlowUnit	Spring loaded fluid container that only accepts connections on the input side. Is filled with fluid when the input pressure is higher than the internal pressure and emptied when the input pressure is lower. Often used in conjunction with a stop valve to control filling/emptying.
CheckValve	FlowUnit	Allows flow in one direction only. Flow in the other direction is blocked. Acts like a regular pipe when the flow is in the allowed direction. Can be used to prevent actuator collapse when the working pressure is not enough to lift or hold a load.
ConstantFlowValve	FlowUnit	Has two modes. Limits the flow rate, in absolute value, to be less than some given maximum, or enforces a given flow rate. Acts like a regular pipe when in the limiting mode and the flow rate is within

		bounds. Acts like a pressure source when in the pumping mode.
FlowConnector	Pressure Connector	The basic hydraulic junction. Connect any type of FlowUnit and it will distribute the incoming flow among all connected FlowUnits.
Motor	Rotational Flow Connector	Connector that converts hydraulic flow into shaft rotation. The FlowUnit connected to the input becomes the fluid chamber and the RotationalUnit connected to the output becomes the motor shaft.
NeedleValve	FlowUnit	A controllable orifice opening. Used much like a pipe, but can be used to dynamically and continuously open and close a restriction. The continuous version of the stop valve.
Pipe	FlowUnit	The most basic FlowUnit. Moves fluid between other components.
PistonActuator	Translational Actuator	Models a hydraulic piston. Forms the interface between the 1-DoF hydraulics and the 6-DoF mechanical world.
Pump	Connector	Connector that converts shaft rotation into hydraulic flow. The RotationalUnit connected to the input becomes the pump shaft and the FlowUnit connected to the output becomes the pump chamber.
ReliefValve	Unit	Multiple orificed component with one input, one output and a number of drains. The drains remain closed until the pressure in the relief valve exceeds the configured cracking pressure, at which point a number of drains are opened depending on the actual pressure in the relief valve. If the pressure is at or above the configured fully open pressure then all drains open.
SpoolValve	Connector	Discrete directional control valve. Accepts any number of connected FlowUnits and can create links between them in arbitrary ways. Used to direct flow from (multiple) sources to (multiple) users.
StopValve	FlowUnit	A valve that is either open or closed. The binary version of the needle valve.
Variable Displacement Pump	Pressure Connector	A pressure regulated pump. Has a default displacement that is in effect as long as the pressure at the pump is below the configured deadhead pressure. The displacement decreases as the pressure increases from the deadhead pressure to the cutoff pressure. At and above the deadhead pressure the displacement becomes zero and the pump passive.

Table 21: Hydraulic components

26.2 Class structure

The hydraulics library is built upon the power line framework and the abstractions that it provides. The hydraulics components are implemented as subclasses of agxPowerLine::Unit, agxPowerLine::Connector or agxPowerLine::Actuator. Many hydraulic

units inherit from agxPowerLine::Unit via the agxHydraulics::FlowUnit class. Similarly, many connectors in the AGX Hydraulics library inherit from agxPowerLine::Connector via agxHydraulics::PressureConnector.

26.2.1 FlowUnit

A power line is a graph of Units connected by Connectors. Units carry motion, which is managed by PhysicalDimensions. The motion that the hydraulic units carry is the motion of the fluid itself. We call the hydraulic units FlowUnits, and the physical dimension of flow is defined by the FlowDimension class. It represents the actual flow in and out of a FlowUnit.

As described in the DriveTrain section, all PhysicalDimensions define a value, a gradient and a mass property. The fluid flow rate, measured in units of volume per time, is the gradient of the flow dimension. The flow dimension's value is the total volume that has passed through the unit. This value may be inaccurate and should not be used. The mass property of a FlowDimension is not the mass of the fluid itself, but rather the resistance to change in flow rate. It increases with the length of the component and decreases with the area.

FlowUnits have a direction and an input and an output side. Forward, or positive, flow is defined as fluid entering at the input side and leaving at the output side. It is not required that FlowUnits are connected output-to-input. The effect of connecting input-to-input or output-to-output is that the two FlowUnits will have flow rates with opposite signs. Some hydraulic components behave differently depending on if it receives flow at the input or the output side, so it is important to keep track of the flow direction through each component.

The most basic FlowUnit is the Pipe, but several other hydraulic components are implemented as subclasses of FlowUnit as well. Hydraulic components that are not FlowUnits tend to inherit from PressureConnector instead.

26.2.2 PressureConnector

Connectors that propagate pressure through the FlowUnit graph are called pressure connectors. They all have a getPressure method, which returns the last pressure that the connector exerted on the connected FlowUnits.

An important subclass of PressureConnector is the FlowConnector. A FlowUnit can connect to many PressureConnectors, but no more than one FlowConnector per side. FlowConnectors are used to move fluid between FlowUnits. It ensures that all fluid that enters the FlowConnector from some connected FlowUnit is accounted for as flow through the other FlowUnits connected to the same FlowConnector.

26.2.3 Actuator

Actuators form the bridge between the one-dimensional world of the power line and the six-dimensional mechanical world of regular AGX simulations. An Actuator is always associated with a regular AGX constraint which defines the operation of the Actuator. An Actuator associated with a Hinge will cause rotation around the hinge axis and an Actuator associated with a Prismatic will cause translational motion along the prismatic axis.

26.3 Building circuits

In this section a simple example circuit is built and the required steps are demonstrated. The construction is split into three parts: flow control, translational actuation and pressure generation.

26.3.1 Foundation setup

The components of a hydraulic circuit must be part of a PowerLine object, which must be added to a simulation.

```
agxSDK::SimulationRef simulation = new agxSDK::Simulation();
agxPowerLine::PowerLineRef powerLine = new agxPowerLine::PowerLine();
simulation->add(powerLine);
```

26.3.2 Connecting FlowUnits

The agxHydraulics library provides a number of hydraulic components that can be connected together to create the hydraulic circuit. The simplest one is the pipe. The agxHydraulics::Pipe constructor takes the pipe's length and area as arguments, along with the fluid density.

This example uses the SI unit system, but that is not a general requirement. See separate section below.

```
agx::Real length(1);
agx::Real area(0.005);
agx::Real density(800);
agxHydraulics::PipeRef pipe = new agxHydraulics::Pipe(length, area, density);
```

Hydraulic components are joined together using FlowConnectors. A FlowConnector represents a junction where multiple pipes and other FlowUnits join. It cannot be connected to any other type of unit.

```
agxHydraulics::FlowConnectorRef junction = new agxHydraulics::FlowConnector();
```

We attach the pipe to the junction using the FlowConnector::connect method. The connector side argument is irrelevant for this particular type of connector, but the unit side argument is important. It specifies the side of the FlowUnit that is to be connected and thus also the direction of flow through the unit. In the following example we connect the output side of the pipe to the junction.

```
using namespace agxPowerLine::Side; // To reduce typing for connect calls.
junction->connect(pipe, CONNECTOR_INPUT, UNIT_OUTPUT);
```

Another type of FlowUnit is the stop valve. It is a manually operated valve that is either fully open or fully closed.

```
agxHydraulics::StopValveUnitRef stop =
new agxHydraulics::StopValveUnit(length, area, density);
```

By connecting the input side of the stop valve to the same junction that the output side of the pipe is connected to we have told the system that any flow that leaves the output side of the pipe should enter the input side of the stop valve.

```
junction->connect(stop, CONNECTOR_OUTPUT, UNIT_INPUT);
```

By creating additional units and connectors it is possible to create arbitrary graphs; including forks, loops, joins and connections to tank. A requirement is that a FlowUnit may only have at most a single FlowConnector at each side. Any FlowUnit side that is not connected to anything, such as the pipe input in the example, is implicitly connected to the tank and will be subjected to tank pressure. The tank pressure is set globally.

```
agxHydraulics::utils::setTankPressure(agx::Real(1e5));
```

The components must be added to the power line for the circuit to be included in the simulation. Adding one unit will bring with it all connected units into the PowerLine, and additional Units connected later will be added as well.

```
powerLine->add(pipe);
```

26.3.3 Coupling to mechanical constraints

The flow control part of the circuit is described above. The next part is actuation, which in the example is a hydraulic cylinder. Actuation, in this context, refers to the transition from the one dimensional hydraulics domain to the six dimensional mechanics domain. The interface for this transition is always a regular AGX constraint. The presence of pressure in the hydraulics at the actuator is translated into a force on the mechanical bodies according to the configuration of the constraint, and vice versa. The force acts in the non-constrained degree of freedom of the AGX constraint.

The example at hand contains a hydraulic cylinder. The class in the AGX Hydraulics library that provides that functionality is agxHydraulics::PistonActuator. It is a type of translational actuator that is created from a prismatic or a distance joint. Here it is assumed that a prismatic has been created previously.

```
agx::PrismaticRef prismatic = ...;
```

The details of the PistonActuator creation are not required for this discussion.

```
agxHydraulics::PistonActuatorRef cylinder =
                 new agxHydraulics::PistonActuator(prismatic, ...);
```

Actuators are connected to the hydraulics circuit in the same way as Units. Instead of explicitly creating and connecting a FlowConnector we make use of implicit connector creation. When two FlowUnits are connected in the following manner the system either creates a new FlowConnector or uses one that has already been created for the FlowUnit sides being connected. If both sides already have a FlowConnector then they are merged into one, moving all connections from one of them to the other.

```
stop->connect(UNIT_OUTPUT, UNIT_INPUT, cylinder);
```

The second argument controls which side of the actuator the unit on which the call is made should be connected to. For a PistonActuator this selects one of the two chambers. Flow into the input side extends the piston, while flow into the output side compresses it. The coupling is two-way, so any force acting along the cylinder direction on the two bodies that the constraint is attached to will cause a pressure increase in one of the cylinder chambers, and any relative movement of the two bodies in the same direction will cause a flow of fluid through the cylinder.

26.3.4 Coupling to the power line

The hydraulics library provides a number of Connectors that bridge between a drive train and the hydraulics. The Pump is one such component. It accepts a RotationalUnit at its input and a FlowUnit at its output and converts torques on the input into pressures at the output. The hydraulic motor performs the opposite operation. It accepts a single FlowUnit input and a single RotationalUnit output.

Both pumps and motors have a property that defines the conversion ratio when translating between angular velocity and flow rate, called the displacement. The displacement describe the volume of fluid moved per radian of rotation of the RotationalDimension. A pump with a displacement of one will produce one unit of pressure for every unit of torque supplied at the input, and it will move, on average, one unit of volume for every radian that the input unit is rotated. With a displacement $\propto$ the pump will produce $\propto$ units of flow for every radian of rotation at the input, and generate $\frac{1}{\propto}$ units of pressure for every unit of torque that is applied at the input. When using the SI unit system, it is common for pumps to have a displacement less than one.

The example circuit contains an engine and a pump, but no motor. Details for how to construct and configure an engine is provided in the section titled agxDriveTrain.

```
agxModel::HighLevelEngineRef engine = ...;
```

The pump constructor takes as its only argument the pump displacement.

```
agx::Real displacement(0.01);
agxHydraulics::PumpRef pump = new agxHydraulics::Pump(displacement);
```

The pump is connected to the engine and the pipe in the same way as the FlowConnector was connected between the pipe and the stop valve.

```
pump->connect(engine, CONNECTOR_INPUT, UNIT_OUTPUT);
pump->connect(pipe, CONNECTOR_OUTPUT, UNIT_INPUT);
```

The pipe has now, in essence, been turned into a pump. The pipe itself represents the fluid chamber inside the pump.

26.3.5 Run time system manipulation

Many of the hydraulic components in the library can be manipulated during runtime through their respective APIs. For some components this is an essential part of their operation. This is exemplified in the example by the stop valve whose main purpose is to open and close based on user input or some form of control logic. Examples of other properties that can be changed during runtime is pump and motor displacement, check valve orientation, relief valve pressure settings and constant flow valve target flow rate.

26.4 Data extraction

The state of each individual component can be inspected using each component's respective API. The example circuit constructed in the previous section is used below.

26.4.1 Pressure

Pressures are properties of the Connectors while flow rates are properties of the FlowUnits. The toolkit does not track pressures along a pipe, only at its end points. To read the pressure at a junction use the FlowConnector's getPressure method.

```
agx::Real sourcePressure = junction->getPressure();
```

The call above will return the pressure at the pump outlet, i.e., the pressure where the stop valve is connected to the pipe. The pressure at any FlowUnit side can be read by first fetching the FlowConnector that the side is connected to. The following statement is equivalent to the previous one.

```
agx::Real sourcePressure = stop->getInputFlowConnector()->getPressure();
```

This technique can be used to read the pressure at locations where the FlowConnector was implicitly created, such as between the stop valve and the hydraulic cylinder in the example.

```
agx::Real workingPressure = stop->getOutputFlowConnector()->getPressure();
```

The pump connector is another subclass of PressureConnector and can therefore also return a pressure. The pressure returned is the pressure that the pump contributes to the system and not the pressure within the FlowUnit that the pump is connected to.

26.4.2 FlowRate

Flow rates are properties of the FlowUnits and are accessed using the getFlowRate method.

```
agx::Real flowRate = pipe->getFlowRate();
```

The flow rate is measured in units of volume per unit time. The toolkit does not track the flow velocity of the fluid, but the average flow velocity can be calculated by dividing the flow rate with the area of the FlowUnit.

There is no flow rate associated with FlowConnectors and, unlike the corresponding situation with pressure, there is no obvious way to find a FlowUnit to read the value from since a FlowConnector may be attached to any number of FlowUnits and the FlowUnits may be attached in any direction.

26.5 Units of measurement

It is possible to use any set of units as long as the choice is consistent across entire simulation spanning drive train, hydraulics and mechanics. This means that if geometry dimensions and rigid body positions are given in meters then FlowUnit areas must be given in meters as well, and reported flow rates will be in cubic meters per second.

Some helper function (RPM conversions) assume that SI units are used.

26.6 Parameter configuration

- Density
  The density of the fluid, in units of mass per unit volume.
- Viscosity
  The kinematic viscosity of the fluid, in units of area per time. The viscosity has a strong influence on the pressure loss over pipes, with zero viscosity resulting in no pressure loss at steady state.
- FlowConnector compliance
  The compliance of the constraint held by the FlowConnector controls the compressibility of the system. The FlowConnectors will flex under pressure, which represents the flexibility of the junction itself, the expansion of the neighboring pipes

and the compressibility of the fluid in those pipes. It is measured in units of volume per unit pressure.
- FlowConnector damping
  Parameter that controls the rate at which pipe, junction and fluid deformations due to pressure variations are created and restored.
- Displacement
  Pumps and motors have a displacement parameter that defines the relationship between flow rate in the chamber and the rotation of the shaft.
- Flow coefficient
  The flow coefficient is currently only configurable for the needle valve. It is a measure of how efficiently fluid can pass through a component. The smaller this value is, the greater effect from the needle valve.
- Tank pressure
  The pressure that any unconnected, i.e., tank connected, FlowUnit side is subjected to.
- ...

26.7 Known limitations and assumptions

- Cavitation/vacuum inside pipes.
  The model assumes that all pipes and other components are always full with fluid and the constraints will ensure that this is the case. This may produce unrealistic pressures in cases where a real-world system would have produced cavitation and it shows up as negative pressures in the FlowConnectors. Common cases where this occurs is when the pump and load is working in the same direction and the pump is unable to keep up, e.g. a hydraulic cylinder is being extended faster than the pump can supply fluid and cavitation would be formed in the cylinder. Another scenario is a stop valve being closed while there is a large flow rate going through it.
- Gravity is ignored.
  Components don't have elevation and all pipes are laid out horizontally.
- Assumes circular pipe cross sections and uniform area over the entire pipe.
- Does not take geometry of junctions into account.
  Cannot model bends or multi-pipe junctions where the angle between the pipes is important.
- No heat simulation.

26.8 Troubleshooting

The following is a listing of common causes of malfunctioning hydraulic simulations.

- Missing PowerLine::add.
- Misconfigured input/output side of unit.
- Valve mistakenly closed.
- Illegal connection (pump or motor backwards, connecting to nullptr, ...).
- Zero displacement for pump/motor. Watch out for integer division.
- PistonActuator created from Prismatic without a properly configured range.
- Always check return values.

27 Hydro- and aerodynamics

When an object is moving through air, water or another fluid it is affected by hydrodynamic effects such as lift, drag and added mass. These effects can be simulated using a WindAndWaterController. Only one (1) controller is needed for each simulation.

```
// Create wind and water controller and add it to the simulation.
agxModel::WindAndWaterControllerRef controller = new agxModel::WindAndWaterController();
simulation->add( controller );
```

The WindAndWaterController performes hydro- and aerodynamic calculations on the following objects:

- Shapes
  - Based on a mesh
  - Boxes
  - Spheres
  - Capsules
  - Cylinders
- Wires
- Cables

For shapes, the effects hydro- and aerodynamic are calculated for each triangle in a mesh. If it is a primitive shape, e.g. a sphere, the calculations are made on a tessellation of that primitive shape.

Wires and cables are not tessellated but divided by the segments. For each segment, the calculations are performed assuming the segment consists of the curved surface of a cylinder.

Hydrodynamic effects will be calculated if the object is in contact with water, otherwise aerodynamic calculations will be performed. For a shape this means that each triangle will be tested. If the triangle is partially submerged the triangle will be clipped creating new triangles so that each triangle is considered to be either completely in air or completely in water. For a wire or cable each segment is tested and if it is partially submerged the segment is divided in two parts, one for hydrodynamics and one for aerodynamics.

The effects that can be calculated is:

- Buoyancy - a floating force in negative gravity direction.
- Pressure drag – a force depending on the normal relative velocity.
- Viscous drag – a force depending on the tangential relative velocity.
- Lift – a force in the normal direction depending on the tangential relative velocity. (Only for shapes, not for wires or cables)
- Added mass – from the surrounding fluid. (Only for shapes, not for wires or cables)

For hydrodynamic calculations, all these all effect will be calculated, whereas for for the aerodynamic calculations, buoyancy and added mass will be omitted due to the low density of air.

The default values for relevant constants are specified in Table 22.

Name	Value

Water density	1000
Air density	1.2
Pressure drag coefficient	0.6
Viscous drag coefficient	0.1
Lift coefficient	0.01

Table 22. Default values used in the simulation.

By changing the density of the fluid and the coefficients of the object, the WindAndWaterController can simulate many fluids other than water and air.

27.1 Parameters

For each shape, there are hydro- and aerodynamic parameters. These can be used to calibrate the behavior of objects in air or water. Note that the interaction between an object and a fluid is defined only by these parameters and not contact materials.

For a shape or wire, these can be reached in the following way:

```
// Hydrodynamic parameter for a shape
agxModel::WindAndWaterParametersRef hydrodynamicParameters = controller->
  getOrCreateHydrodynamicsParameters( shape );

// Aerodynamic parameters for a shape
agxModel::WindAndWaterParametersRef aerodynamicParameters = controller->
  getOrCreateAerodynamicsParameters( shape );

// Hydrodynamic parameter for a wire
agxModel::WindAndWaterParametersRef hydrodynamicParameters = controller->
  getOrCreateHydrodynamicsParameters( wire );

// Aerodynamic parameters for a wire
agxModel::WindAndWaterParametersRef aerodynamicParameters = controller->
  getOrCreateAerodynamicsParameters( wire );
```

For a cable, parameters can be stored both by cable and by segment of the cable.

```
// Hydrodynamic parameter for a cable
agxModel::WindAndWaterParametersRef hydrodynamicParameters = controller->
  getOrCreateHydrodynamicsParameters( cable );

// Aerodynamic parameters for a cable
agxModel::WindAndWaterParametersRef aerodynamicParameters = controller->
  getOrCreateAerodynamicsParameters( cable );

// Hydro- and aerodynamic parameters by segment of cable
agxCable::CableIterator iterator = cable->begin();
while ( !iterator.isEnd() )
{
  agxCollide::ShapeRef shape = iterator.getGeometry()->getShape();
  agxModel::WindAndWaterParametersRef aerodynamicParameters = controller->
    getOrCreateAerodynamicsParameters( shape );

  // Do something with the parameters

  iterator.advance();
}
```

The priority of parameters for a cable is as follows:

1. If parameters are set for a segment these parameters will be used.
2. If no parameters exists for the segment the parameters for the cable it belongs to will be used.
3. If no parameters exists for the cable default parameters will be used.

For a shape, the hydro- and aerodynamic effects will be calculated for each triangle in a mesh. If it is a primitive shape, e.g. a sphere, the tessellation can be chosen to be low, medium, high or ultra high.

```
// Tessellation level set to hydro- or aerodynamic parameters
parameters->setShapeTessellationLevel(
  agxModel::WindAndWaterParameters::ShapeTessellation::ULTRA_HIGH );
```

For a wire or a cable, the segments for calculation are defined by the resolution.

The hydro- and aerodynamic coefficients of pressure drag, viscous drag and lift can be set in the parameters. Note that lift will not be calculated for wires and cables due to symmetry.

```
// Change the pressure drag coefficient to 0.5
parameters->setCoefficient( agxModel::WindAndWaterParameters::PRESSURE_DRAG, 0.5 );

// Change the viscous drag coefficientto 0.5
parameters->setCoefficient( agxModel::WindAndWaterParameters::VISCOUS_DRAG, 0.5 );

// Change the lift coefficient to 0.5
parameters->setCoefficient( agxModel::WindAndWaterParameters::LIFT, 0.5 );
```

To change a coefficient for all shapes of a rigid body the following methods can be used:

```
//Setting the hydrodynamic lift coefficient to zero.
agxModel::WindAndWaterParameters::setHydrodynamicCoefficient( controller, rigidBody,
  agxModel::WindAndWaterParameters::LIFT, 0 );

//Setting the aerodynamic lift coefficient to zero.
agxModel::WindAndWaterParameters::setAerodynamicCoefficient( controller, rigidBody,
  agxModel::WindAndWaterParameters::LIFT, 0 );
```

The default values for the coefficients are specified in Table 22. Since the viscous drag becomes negligible compared to pressure drag with increasing velocity it might be a good idea to have the viscous drag coefficient decrease, and even become zero, at high velocities.

27.2 Hydrodynamics

To enable hydrodynamic calculations a water geometry has to be defined. The geometry has to contain exactly one shape that can be either a box, a plane, a heightfield or a sphere.

```
// Let the water geometry to be an box of size 60x60x30 meters.
agxCollide::ShapeRef waterShape = new agxCollide::Box( 30, 30, 15 );
agxCollide::GeometryRef waterGeometry = new agxCollide::Geometry( waterShape );

// Mark as water geometry. Contacts will be disabled for the water geometry, i.e.,
// no "normal" contacts will be created, only hydrodynamic forces.
agxModel::WindAndWaterControllerRef controller = new agxModel::WindAndWaterController();
controller->addWater( waterGeometry );
```

The default value for the water density is 1000 kg/m^3. By assigning a material to the geometry this value can be changed.

```
  // Create material to define the density of water.
  agx::MaterialRef waterMaterial = new agx::Material( "waterMaterial" );
  waterMaterial->getBulkMaterial()->setDensity( agx::Real( 900 ) );

  waterGeometry->setMaterial( waterMaterial );
```

When a water geometry is defined all geometries, wires and cables that overlap the water will be subject to hydrodynamic calculations which consist of buoyancy, pressure drag, viscous drag and for geometries also lift.

The wind- and watercontroller can contain several water geometries.

Examples of hydrodynamic simulations can be seen in **tutorial_hydrodynamics**.

27.2.1 Added mass

To enable added mass calculations for a shape a file containing static data of the shape has to be read:

```
  // Initiate added mass.
  hydrodynamicParameters->initializeAddedMassStorage( "fileName.dat" );
```

If there is no file that matches the specified file name an analysis of the shape will be perfomed creating a file with that name. Note that this analysis can be very time consuming, but will only be performed the first time.

For a wire or a cable there will be no added mass calculation.

27.2.2 Water flow

Water flow can be added by creating a WaterFlowgenerator. The interface can be found in agxModel::WindAndWaterFlowGenerator.

```
/**
Interface for generating water flow and wind.
*/
class AGXMODEL_EXPORT WindAndWaterFlowGenerator : public agx::Referenced,
                                                  public agxStream::Serializable
{
  public:
    /**
    Default constructor disables serialization. Any implementation
    has to enable this explicitly.
    */
    WindAndWaterFlowGenerator();


    /**
    Called before update of \p geometry.
    \param geometry - geometry to be updated
    */
    virtual void prepare( const agxCollide::Geometry* /*geometry*/ ) {}

    /**
    Calculate and return wind at a given position in world.
    \param worldPoint - point in world
    \return wind velocity in world coordinates
```

```
    */
    virtual agx::Vec3 getVelocity( const agx::Vec3& worldPoint ) const = 0;
};
```

```
/**
Interface for generating water flow.
*/
class AGXMODEL_EXPORT WaterFlowGenerator : public WindAndWaterFlowGenerator
{
  public:
  /**
  Default constructor disables serialization. Any implementation
  has to enable this explicitly.
  */
  WaterFlowGenerator();
};
```

An example of how to implement a WaterFlowGenerator can be seen in **tutorial_hydrodynamics.cpp**. Another example can be seen in **tutorial_wind.cpp** where a WindGenerator with shadowing is implemented. Note that the WindGenerator shares the same interface as the WaterFlowGenerator.

A water flow created with an implementation of this interface can then be added to the controller.

```
  // Add a water flow generator belonging to a given waterGeometry.
  controller->addWaterFlowGenerator( waterGeometry, waterFlowGenerator );
```

If there are several water geoemtries in the simulation, the water flow generator has to be added for each geometry, otherwise that water geometry will have no water flow. It is also possible to have different water flow generators for each water geometry.

Use ConstandWaterFlowGenerator to create constant, uniform water flow.

```
  // Create a constant water flow over a given water geometry
  const agx::Vec3 flowVelocity( 5, 0, 0 );
  agxModel::ConstantWaterFlowGenerator waterFlowGenerator =
    new agxModel::WaterFlowGenerator( flowVelocity );
  controller->addWaterFlowGenerator( waterGeometry, waterFlowGenerator );
```

27.3 Aerodynamics

Aerodynamics can be enabled to get the effect of air resistance or wind. By default, these calculations are disabled but they can be enabled for any geometry, rigid body, wire or cable.

```
  // Enable aerodynamics for a geometry.
  controller->setEnableAerodynamics( geometry, true );

  // Enable aerodynamics for a rigid body.
  controller->setEnableAerodynamics( rigidBody, true );

  // Enable aerodynamic calculations for a wire.
  controller->setEnableAerodynamics( wire, true );

  // For a cable aerodynamics can be enabled by cable or by segment
  // Enable aerodynamics for a cable
  controller->setEnableAerodynamics( cable, true );

  // Enable aerodynamics for segment of a cable.
  agxCable::CableIterator iterator = cable->begin();
  while ( !iterator.isEnd() )
  {
    agxCollide::GeometryRef geometry = iterator.getGeometry();
    controller->setEnableAerodynamics( geometry, true );
```

```
    iterator.advance();
  }
```

It is also possible to enable aerodynamics for all objects in the simulation.

```
  // Enable aerodynamics for all objects in the simulation
  controller->setEnableAerodynamics( true );
```

Note that this can be time consuming. For performance reasons it is best to limit the objects enabled for aerodynamics.

The aerodynamic calculations consist of pressure drag, viscous drag and for geometries also lift. Added mass and buoyancy are omitted due to the low density of air.

The default value for air density is 1.2 kg/m^3 and can be changed in the WindAndWaterController.

```
  // Set the air density to 1.4 instead of the default value of 1.2
  controller->setAirDensity( agx::Real( 1.4 ) );
```

27.3.1 Wind

Wind can be added to the simulation by creating a wind generator. The interface is found in agxModel::WindAndWaterFlowGenerator.

```
/**
Interface for generating water flow and wind.
*/
class AGXMODEL_EXPORT WindAndWaterFlowGenerator : public agx::Referenced,
                                                  public agxStream::Serializable
{
  public:
    /**
    Default constructor disables serialization. Any implementation
    has to enable this explicitly.
    */
    WindAndWaterFlowGenerator();

    /**
    Called before update of \p geometry.
    \param geometry - geometry to be updated
    */
    virtual void prepare( const agxCollide::Geometry* /*geometry*/ ) {}

    /**
    Calculate and return wind at a given position in world.
    \param worldPoint - point in world
    \return wind velocity in world coordinates
    */
    virtual agx::Vec3 getVelocity( const agx::Vec3& worldPoint ) const = 0;
};

/**
Interface for generating wind effects.
*/
class AGXMODEL_EXPORT WindGenerator : public WindAndWaterFlowGenerator
{
  public:
    /**
    Default constructor disables serialization. Any implementation
    has to enable this explicitly.
    */
    WindGenerator();
};
```

A wind created with an implementation of this interface can then be added to the controller.

```
//Adding a windGenerator to the WindAndWaterController
controller->setWindGenerator( windGenerator );
```

Use ContantWindGenerator to create a constant, uniform wind.

```
// Create a constant wind, with a given wind direction and magnitude.
const agx::Vec3 windVelocity = agx::Vec3( -15, 0, 0 );
agxModel::ConstantWindGeneratorRef windGenerator = new agxModel::ConstantWindGenerator(
  windVelocity );
controller->setWindGenerator( windGenerator );
```

An example of using a custom wind to create shadowing can be seen in tutorial_wind.cpp together with other examples. There is also an example of a windmill that is affected by aerodynamics in windmill.agxLua.

27.4 Wires and Cables

As explained above the hydro- and aerodynamic effects can be applied to wires with the effects of buoyancy, pressure drag and viscous drag. For a full example with wires and cables, see tutorial_wireWindAndWater.cpp and tutorial_cableWindAndWater.

27.5 Pressure field renderer

The pressure field renderer is used to visualize the pressure acting on a shape by the hydro- and aerodynamic forces.

```
// Create a pressure field renderer and give it to the controller
agx::Real scale = 1.01;
agxOSG::PressureFieldRendererRef pfr = new agxOSG::PressureFieldRenderer( root, scale );
controller->registerPressureFieldRenderer( pfr, shape );
```

The pressure field renderer is not available for wires and cables.

27.6 Known limitations

- **The object will not affect the water in any way. This means that:**
  - **An object will not be affected by another objects' movement nearby.**
  - **Object movement in water will not produce waves.**
- **All parts below the water surface will be subject to hydrodynamic calculations. This means that e.g. an empty bowl floating in water will experience a downward pressure from water if the inside of the bowl is beneath the surface level.**
- **Any geometry or wire segment can only be affected by one water geometry at any given time.**
- **Since these effects are calculated per shape, an object consisting of several shapes or geometries can lead to unwanted behavior.**
- **Wires and cables are considered as lines with a thickness. This means that a segment crossing the water surface will have correct behavior but a segment laying along the water surface will not.**

28 Locating files with agxIO::Environment

All resources (images, models, scripts) and runtime libraries (.dll/.so) are located using the singleton class *agxIO::Environment*. Currently there are two separate types of files:

- **Runtime files: .dll (WIN32), .so (Unix), .dylib (OSX)**
- **Resource files: (.png, .obj, .png, .cfg, .agxLua, .agxScene, .schema, .agxKernel)**

The path where these files can be found can be either controlled using environment variables as specified in chapter 41, or programmatically using the *agxIO::Environment class*. This should be done before creating any other AGX object where file reading will be involved.

```
// Get a file path container to where runtime files are searched
agx::FilePathContainer& runtime_path =
  agxIO::Environment::instance()->getFilePath(
      agxIO::Environment::RUNTIME_PATH );

runtime_path.clear(); // Remove all previous settings
runtime_path.pushbackPath( "MyPath/plugins" );

// Try to find a file in the specified path
std::string filename = runtime_path.find( "plugin.dll" );
```

Same procedure can be done for RESOURCE_PATH to specify in what directories images, models, scripts and other non-runtime resources should be searched for.

```
// Get a file path container to where resource files are searched
agx::FilePathContainer& resource_path =
  agxIO::Environment::instance()->getFilePath(
      agxIO::Environment::RESOURCE_PATH );

resource_path.clear(); // Remove all previous settings
resource_path.pushbackPath( "MyPath/data/images;MyPath/data/models;" );
resource_path.pushbackPath( "MyPath/data/textures" );

// Try to find a file in the specified path
std::string filename = resource_path.find( "heightfield.png" );
```

In the windows version ';' should be used to separate paths, and under UNIX the ':' letter is used. Paths can be absolute or relative.

29 AutoSleep

AutoSleep is a functionality for putting bodies which are not moving (below a certain threshold) to sleep. This means that the solver will no longer take these objects into account, hence reducing the CPU usage substantially.

A body can be put to sleep if its *absolute* velocity/acceleration goes below a certain threshold. AutoSleep is enabled for the system with a call to:

```
agxSDK::SimulationRef simulation = new agxSDK::Simulation;
agx::AutoSleep *autoSleep = simulation->getDynamicsSystem()->getAutoSleep();
// Enable autosleep functionality
autoSleep->setEnable(true);
```

AutoSleep also has to be enabled for each body:

```
agx::RigidBodyRef body = new agx::RigidBody
body->getAutoSleepProperties()->setEnable(true); // Enable AutoSleep for this body
```

A body will be awaken if

- Another non-sleeping body get into contact with its geometry
- Another body attached by a constraint is awaken
- Explicit awakened through an API call

A body can be explicitly awakened through a call to:

```
body->getAutoSleepProperties()->setSleeping( false );
```

29.1 Threshold

The acceleration, velocity and time threshold can be set globally for AutoSleep:

```
// When linear velocity goes below 0.3m/s it will be considered for sleeping
autoSleep->getThreshold()->setVelocity(0.3);

// When angular velocity goes below 0.1rad/s it will be considered for sleeping
autoSleep->getThreshold()->setAngularVelocity(0.1);

// When velocity AND acceleration has been below threshold for 0.7 seconds the body
// will fall to sleep
autoSleep->getThreshold()->setTime( 0.7 );
```

Or locally per body:

```
// Create an AutoSleepThreshold object and set the values
agx::AutoSleepThresholdRef t = new agx::AutoSleepThreshold;
t->setVelocity(0.1);
t->setAngularVelocity(0.1);
t->setTime(2.0);

// Attach it to a body
body->setAutoSleepThreshold( t );

// Several bodies can share the same AutoSleepThreshold
body2->setAutoSleepThreshold( t );
```

30 agx::MergedBody – many bodies simulated as one

A merged body is several agx::RigidBody objects merged together, and handled as one single agx::RigidBody in the solver. This functionality can be used to *dramatically reduce the system size*, as seen from the solver, but preserving the mass properties/inertia of a merged system. It can be seen as an extension of AutoSleep functionality, but with the huge benefit, that it support merging/sleeping on moving/dynamics bodies. Whereas AutoSleep has the big limitation that it only support sleeping on non moving bodies (zero velocity in world frame).

The functionality of agx::MergedBody can be used in two ways:

- Explicitly/manual: By explicitly creating EdgeInteractions (chapter 30.3) between bodies which are added to an agx::MergedBody the user has full control over how bodies are merged and split.
- Automatic: Chapter 31 describes functionality for automatic merge/split based on relative velocities, contacts and force measurements.

```
// Create a new merged body.
agx::MergedBodyRef mergedBody = new agx::MergedBody();
// Active the merged body by adding it to the simulation.
simulation->add( mergedBody );
// Deactivating the merged body by removing it from
// the simulation.
simulation->remove( mergedBody );
```

30.1 Applications

Let's have a look at same sample cases. A good start is to have a distinct parent body. This parent body can for example be a ship, the ground, a harbor, a flatbed, etc.

30.1.1 Hydraulic cranes on an offshore vessel

A set of hydraulic cranes on an offshore vessel. Each crane may consist of around 15 bodies and 15-20 constraints. The weight of the crane is significant, affecting the motion of the ship, so when the crane isn't in use, it cannot just be removed (e.g., only rendered instead).

15 bodies and 15-20 constraints are in general fast to solve but imagine three cranes per vessel and 10 vessels. It's now 450 extra rigid bodies and about 500 constraints that aren't explicitly in use.

Now, merging the bodies in the cranes to the ship, the constraints will become inactive (since no relative motion), and the solver will only have to solve for ONE rigid body instead of 45.

30.1.2 Logs or rocks on a flatbed

A few hundred logs or rocks on a moving flatbed can in general take a significant amount of time in the solver. For this to be a reasonable scenario, the vehicle pulling the flatbed has to feel the weight and inertia of the load, so the load cannot simply be removed to reduce the time in the solver.

If the stacked logs or rocks can be considered steady, i.e., not moving relative to each other, it's possible to merge the objects to the flatbed. Instead of a few hundred objects + thousands of contacts to solve, the solver only has one single body, attached to the vehicle to solve for.

30.2 The dynamics of merged bodies

The motion control of the final merged body is determined by the motion controls of the set of rigid bodies merged, with the following priority:

1. (if one is) KINEMATICS results in KINEMATICS
2. (if at least one is) STATIC and none KINEMATICS, the result is STATIC
3. (if all are) DYNAMICS and none STATIC or KINEMATICS, the result is DYNAMICS

Note: Only one rigid body with KINEMATICS motion control may occure in a merged body at the time! Having more than one is considered undefined since it's not possible to determine the final velocity of the bodies involved.

If the final motion control is DYNAMICS, the total mass becomes

$$m^{tot} = \sum_i m_i$$

the center of mass

$$p^{cm} = \frac{1}{m^{tot}} \sum_i m_i p_i^{cm}$$

and finally the inertia

$$I^{tot} = \sum_i I_i - m_i [p^{cm} - p_i^{cm}]^2$$

where

$$[r] = \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}.$$

If the motion control **is different from** DYNAMICS, the calculations of center of mass, mass and inertia are ignored.

In the case of KINEMATICS, the linear- and angular velocity will be the one of the kinematic body. In the DYNAMICS case, the final linear- and angular velocity becomes:

$$v^{tot} = \frac{1}{m^{tot}} \sum_i m_i v_i$$

$$\omega^{tot} = \frac{1}{m^{tot}} \sum_i m_i \omega_i$$

30.2.1 Advanced mass properties

agx::MergedBody supports rigid bodies with advanced mass properties features. Examples of advanced features are damping (44.2), added mass (44.1).

30.3 Edge interactions – merging objects together

The internal structure of an agx::MergedBody is a graph where the nodes in the graph is an agx::RigidBody and the edges, connecting the nodes, are so called edge interactions.agx

An edge interaction defines how the merged rigid bodies interacts and what happens when an edge is removed.

30.3.1 Empty edge interaction

An empty edge interaction is a pure logical connection between two rigid bodies.

```
// Merge rb1 <-> rb2 given an empty edge interaction.
mergedBody->add( new agx::MergedBody::EmptyEdgeInteraction( rb1, rb2 ) );
```

30.3.2 Contact generator edge interaction

A contact generator edge interaction is an edge that, depending on when the edge is removed, can perform collision detection between the geometries in the first and the second rigid body.

If the contact generator edge is removed before the collision detection is executed, the edge does nothing. I.e., it acts as an empty edge interaction, assuming the objects will participate in the collision detection given the usual update.

If the contact generator edge is removed *after* the collision detection pass has been run and *before* the solver, the edge will check for overlaps between the two bodies and add agxCollide::GeometryContact's to the system. This prevents the two objects to "fall into" each other when the solver solves the system.

```
// Merge rb1 <-> rb2 and generate geometry contacts
// between them if we split in a PRE_STEP callback.
mergedBody->add( new agx::MergedBody::ContactGeneratorEdgeInteraction( rb1, rb2 ) );
```

30.3.3 Geometry contact edge interaction

An edge interaction, constructed given an agxCollide::GeometryContact, taking advantage of the contact information, normal and friction forces.

To gain full advantage of this edge, the solver should have seen the geometry contact, i.e., normal and friction forces are available in the contact.

```
// Post-step: If the objects in the geometry contact
// are at rest, merge them.
for ( auto geometryContact : simulation->getSpace()->getGeometryContacts() ) {
  if ( isRestingContact( geometryContact ) )
    mergedBody->add( new agx::MergedBody::GeometryContactEdgeInteraction( *geometryContact ) );
}
```

Similar to agx::MergedBody::ContactGeneratorEdgeInteraction, the contact points and normal are added back, as a new agxCollide::GeometryContact, if the edge is removed after collision detection but before the solver executes.

30.3.4 Binary constraint edge interaction

A binary constraint edge interaction is an edge constructed given a one or two (hence binary) body constraint.

This edge does nothing when removed from an agx::MergedBody.

```
// If the hinge motor isn't enabled - merge.
if ( !hinge->getMotor1D()->getEnable() )
  mergedBody->add( new agx::MergedBody::BinaryConstraintEdgeInteraction( hinge ) );
```

30.4 Data and state of the merged rigid bodies

If an agx::RigidBody is merged, it's possible to access the agx::MergedBody it belongs to by doing:

```
// Check whether rb belongs to a merged body.
agx::MergedBody* mergedBody = agx::MergedBody::get( rb );
if ( mergedBody != nullptr )
  std::cout << rb->getName() << " is merged." << std::endl;
```

Having access to the agx::MergedBody object, it's possible to traverse/inspect/collect the nodes and edges:

```
// Collect edges connected to 'rb'.
agx::MergedBody::EdgeInteractionRefContainer rbEdges;
agx::MergedBody::EdgeInteractionVisitor visitor = [ &rbEdges ]( agx::MergedBody::EdgeInteraction* edge )
{
  rbEdges.push_back( edge );
};
// 'Visit all edges associated to rb'.
mergedBody->traverse( rb, visitor );

// Print the name of the bodies 'rb' is merged to.
for ( auto edge : rbEdges ) {
  const agx::RigidBody* otherRb = edge->getRigidBody1() == rb ? edge->getRigidBody2() :
                                                                edge->getRigidBody1();
  std::cout << "rb is merged to: " << otherRb->getName() << std::endl;
}
```

A similar example, visiting all neighboring nodes instead of collecting all edges (there may be an arbitrarily number of edges between two bodies):

```
agx::RigidBodyPtrVector otherBodies;
agx::MergedBody::RigidBodyVisitor visitor = [ &otherBodies ]( agx::RigidBody* otherRb )
{
  otherBodies.push_back( otherRb );
};
// 'Visit all neigboring nodes to rb'.
mergedBody->traverse( rb, visitor );

// Print the name of the bodies 'rb' is merged to.
for ( auto otherRb : otherBodies )
  std::cout << "rb is merged to: " << otherRb->getName() << std::endl;
```

Or one can simply do:

```
const auto* otherBodies = mergedBody->getNeighbors( rb );
if ( otherBodies == nullptr )
  std::cout << "rb is not merged in mergedBody" << std::endl;
else
  for ( auto otherRb : *otherBodies )
    std::cout << "rb is merged to: " << otherRb->getName() << std::endl;
```

30.4.1 Listeners

It's possible to add listeners to an agx::MergedBody. These listeners are passive, i.e., it's not valid to change the state and not possible to for example prevent a merge, from within a listener.

Callbacks:

- ***clone()***
  Create a clone of your listener. It's valid to return the same instance. The clone method is called when the merged body, the listener belongs to, is being split into several agx::MergedBody instances (see: agx::MergedBody::splitIslands).
- ***onAdd(agx::RigidBody* rb, const agx::MergedBody* mb)***
  Called when a rigid body is added to the merged body.

- ***onRemove(agx::RigidBody* rb, const agx::MergedBody* mb)***
  Called when a rigid body is being removed from the merged body.
- ***onAdded(EdgeInteraction* edge, const agx::MergedBody* mb)***
  Called when an edge has been added to the merged body.
- ***onRemoved(const EdgeInteractionRefContainer& edges, const agx::MergedBody* mb)***
  Called when a set of edges has been removed from the merged body.
- ***onMovedFromTo(const EdgeInteractionRefContainer& edges, const agx::MergedBody* fromMergedBody, const agx::MergedBody* toMergedBody)***
  Called when a set of edges has been moved from one merged body to another.

30.5 Splitting merged rigid bodies

If a rigid body is merged, there are several ways to split it. The most convenient way is to do:

```
// Split method finds the merged body associated to 'rb'
// and removes 'rb' from it.
const agx::Bool success = agx::MergedBody::split( rb );
if ( success )
  std::cout << rb->getName() << " successfully removed from its merged body." << std::endl;
else
  std::cout << rb->getName() << " failed to split. Was it merged in the first place?" << std::endl;
```

For more control, use the *remove* method:

```
// Find if 'rb' is merged.
agx::MergedBody* mergedBody = agx::MergedBody::get( rb );
// If merged, remove it (split).
if ( mergedBody != nullptr )
  mergedBody->remove( rb );
```

The *remove* method guarantees to remove any references to the agx::MergedBody as long as *rb* is present in *mergedBody*.

Note that when you're managing the agx::MergedBody objects, an instance may become empty when removing a rigid body. E.g., two bodies are merged, remove one of them will result in a single body without any edges being the only one left – so it will be removed as well. Leaving the agx::MergedBody object empty. An update to the above example:

```
// Find if 'rb' is merged.
agx::MergedBodyRef mergedBody = agx::MergedBody::get( rb );
// If merged, remove it (split).
if ( mergedBody != nullptr ) {
  mergedBody->remove( rb );
  // If the remove of 'rb' results in an empty 'mergedBody',
  // remove 'mergedBody' from the simulation.
  if ( mergedBody->isEmpty() )
    simulation->remove( mergedBody );
  mergedBody = nullptr;
}
```

It's also possible to split a rigid body by removing all the edges to its neighboring nodes/bodies:

```
// Collect edges connected to 'rb'.
agx::MergedBody::EdgeInteractionRefContainer rbEdges;
agx::MergedBody::EdgeInteractionVisitor visitor = [ &rbEdges ]( agx::MergedBody::EdgeInteraction* edge )
{
  rbEdges.push_back( edge );
};
// 'Visit all edges associated to rb'.
mergedBody->traverse( rb, visitor );
```

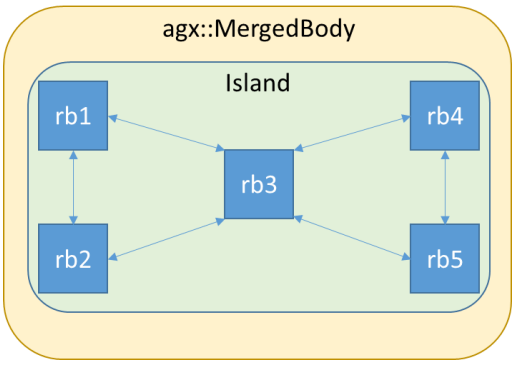
```
// Remove all the edges associated to our 'rb'.
for ( auto edge : rbEdges )
  mergedBody->remove( edge );

// When all edges has been removed, 'rb' hasn't got
// any connections left in 'mergedBody' so it will
// be removed.
agxAssert( agx::MergedBody::get( rb ) == nullptr );
```

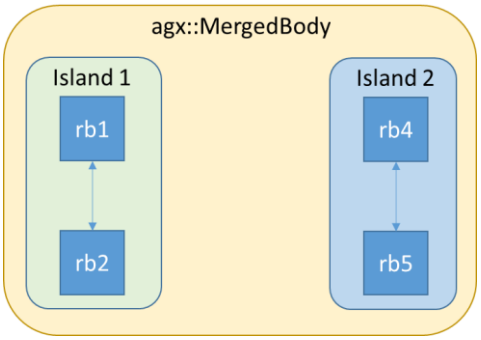
30.6 Separate islands within an agx::MergedBody

The agx::MergedBody doesn't assume that all rigid bodies/nodes are connected. E.g, if you have four bodies *rb1*, *rb2*, *rb3* and *rb4*, and add edge interactions *rb1* <-> *rb2* and *rb3* <-> *rb4* to the agx::MergedBody – it's a valid, but maybe not a desired state.

A more natural example. Say we have five rigid bodies, merged like this:



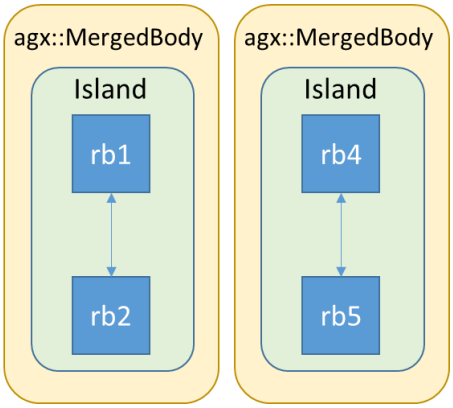
Removing *rb3* will result in the following merged state:



where there aren't any interactions connecting *island 1* to *island 2*. It's not possible to query the merged body how many islands it consists of since it needs a full graph traversal to determine this. Still, if the desired behavior is there shouldn't be separate islands in my merged bodies' it's possible to call agx::MergedBody::splitIslands. This method will determine separate islands, create new merged bodies and add them to the simulation.

```
// Checks for separate, non-interacting, islands in mergedBody.
// NOTE: mergedBody has to be in the simulation and all new
//       agx::MergedBody objects will be added to the same
//       simulation.
agx::MergedBodyRefVector newIslands = mergedBody->splitIslands();
std::cout << "Number of new islands: " << newIslands.size() << std::endl;
```

Resulting in:



30.7 Merging two already merged objects

For obvious reasons (it's rigid, it cannot move in different directions at the same time!), a rigid body may only belong to one agx::MergedBody. As a result of this constraint, an edge will be rejected by a merged body if one (or both) of the bodies included is merged to another merged body. This state has to be explicitly handled by the user.

Given two rigid bodies *rb1* and *rb2*, merged with other objects in *mergedBody1* and *mergedBody2*, it's possible to merge the two agx::MergedBody objects together given an edge between *rb1* and *rb2*:

```
agx::MergedBody* mergedBody1 = agx::MergedBody::get( rb1 );
agx::MergedBody* mergedBody2 = agx::MergedBody::get( rb2 );
// Copying all data from mergedBody2 into mergedBody1 and
// adding the empty edge interaction to mergedBody1.
agx::MergedBody::EmptyEdgeInteractionRef edge = new agx::MergedBody::EmptyEdgeInteraction( rb1, rb2 );
const agx::Bool success = mergedBody1->merge( mergedBody2, edge );
// If merge is successful, remove the empty mergedBody2
// from the simulation.
if ( success )
  simulation->remove( mergedBody2 );
```

30.8 Active and inactive

Consider the example scenario described in Hydraulic cranes on an offshore vessel. The crane is not in use so we have it merged. The state of the agx::MergedBody containing the objects, is *active*:

```
// Adding 'craneMergedBody' will active it and the containing
// rigid bodies will be merged.
simulation->add( craneMergedBody );
```

When the crane is ready for use, e.g., moving the boom, we want to inactivate the agx::MergedBody. This can of course be done in many different ways, but one convenient thing of the agx::MergedBody objects is that it will preserve its internal structure when removed from the simulation. The objects merged will simply not be merged anymore.

Let a prismatic be the constraint we control the boom hydraulic piston/cylinder with. When the prismatic motor is active we expect the objects to be able to move relative each other, and when the motor is inactive – we may merge the crane objects again. This can be achieved by adding and removing the *craneMergedBody*:

```
if ( prismatic->getMotor1D()->getSpeed() != 0.0 ) {
  // Motor is active, split all objects in 'craneMergedBody'
  // by removing it from the simulation.
  if ( craneMergedBody->isInSimulation() )
    simulation->remove( craneMergedBody );
```

```
  }
  else {
    // Motor isn't active, merge all objects in 'craneMergedBody'
    // by adding it to the simulation.
    if ( !craneMergedBody->isInSimulation() )
      simulation->add( craneMergedBody );
  }
```

This means that a rigid body can be associated to an agx::MergedBody but it's still not actively merged. It's possible to check whether an objects is actively merged (i.e., not seen by the solver) or passively merged (i.e., part of a merged body, but that merged body is not in a simulation):

```
  // agx::MergedBody::get returns the merged body 'craneBoom' is part of.
  const agx::Bool isPartOfAMergedBody = agx::MergedBody::get( craneBoom ) != nullptr;
  // agx::MergedBody::getActive returns the merged body 'craneBoom' is part
  // of IF that merged body is a part of the simulation.
  const agx::Bool isActivelyMerged = agx::MergedBody::getActive( craneBoom ) != nullptr;
```

During normal usage there's no need to use the *getActive* method.

31 agxSDK::MergeSplitHandler – AMOR

AGX Adaptive Model Order Reduction (AMOR) are algorithms where a simulated system adopts to a size where the dynamics is presumed to be preserved. It aims to reduce the number of degrees of freedoms given conditions of steady states. I.e., if the relative motion between two objects is zero, it's possible to consider these two objects as one, as long as the local system isn't affected by external interactions that may change their relative motion.

The purpose of AMOR is solely to reduce the workload of the dynamics solver and collision detection (broad phase excluded).

In the previous section we covered how to explicitly merge and split rigid bodies to/from each other, using the agx::MergedBody object. This section is about an *automatic version of merging and splitting rigid bodies*, where the adding, removing and managing of the agx::MergedBody objects is handled by an agxSDK::MergeSplitHandler object.

31.1 Enabling and disabling the agxSDK::MergeSplitHandler

Every instance of an agxSDK::Simulation has an instance of agxSDK::MergeSplitHandler. To enable or disable the merge split handler, simply:

```
simulation->getMergeSplitHandler()->setEnable( true );
simulation->getMergeSplitHandler()->setEnable( false );
```

Disabling the handler will split all objects merged by it.

31.2 agxSDK::MergeSplitProperties

The agxSDK::MergeSplitProperties defines if and how an object may merge and/or split to and from other objects. By default, the objects doesn't carry an instance of the agxSDK::MergeSplitProperties, and this is interpreted as 'merge and split of this object is disabled'.

To get an already created or to create a new instance of MergeSplitProperties for an object (RigidBody or Geometry):

```
agxSDK::MergeSplitProperties* properties = agxSDK::MergeSplitHandler::getOrCreateProperties( obj );
agxAssert( obj == nullptr || properties != nullptr );
```

Given *obj != nullptr* then *properties != nullptr*.

With the merge split properties it's possible to:

- ***Enable and disable merge (disabled by default).***
  When *merge* is enabled, it means that this object may merge to other objects.
- ***Enable and disable split (disabled by default).***
  When *split* is enabled, it means that this object (if merged), may split from the object it's merged to.
- ***Enable and disable merge and split (disabled by default).***
  Convenience method to enable/disable both merge and split.

Objects that can carry an agxSDK::MergeSplitProperties instance are; *agx::RigidBody*, *agxCollide::Geometry* and *agxWire::Wire*:

```
agxSDK::MergeSplitProperties* rbProperties       = agxSDK::MergeSplitHandler::getOrCreateProperties( rb );
agxSDK::MergeSplitProperties* geometryProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( geometry );
agxSDK::MergeSplitProperties* wireProperties     = agxSDK::MergeSplitHandler::getOrCreateProperties( wire );
```

To check if an objects has properties or to inspect the current state:

```
const agxSDK::MergeSplitProperties* rbProperties       = agxSDK::MergeSplitHandler::getProperties( rb );
const agxSDK::MergeSplitProperties* geometryProperties = agxSDK::MergeSplitHandler::getProperties( geometry );
const agxSDK::MergeSplitProperties* wireProperties     = agxSDK::MergeSplitHandler::getProperties( wire );
```

31.2.1 Properties carried by agxCollide::Geometry or its parent – agx::RigidBody

Since the context of AMOR is 'bodies', the properties agxCollide::Geometry behaves a bit different. It is, as shown above, possible to create and change the merge split properties of a geometry object making it possible to, for example, have merge enabled/disabled in different parts of a rigid body.

A geometry inherits the merge split properties of its parent rigid body. Assume we've created merge split properties for the rigid body and enabled merge:

```
agx::RigidBodyRef rb = new agx::RigidBody();
agxCollide::GeometryRef geometry = new agxCollide::Geometry( new agxCollide::Sphere( 0.5 ) );
rb->add( geometry );
//...
// Create merge split properties and enable merge for 'rb'.
agxSDK::MergeSplitProperties* rbProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( rb );
rbProperties->setEnableMerge( true );
```

Now, for geometries, the sematics of the *getProperties* method differs from the rigid body and wire versions in two ways:

1. There's a mutable version.
2. If the geometry hasn't got any merge split properties (i.e., *getOrCreateMergeSplitProperties* hasn't been called), the merge split properties of the parent rigid body will be returned (if created).

```
// 1.
agxSDK::MergeSplitProperties* geometryProperties = agxSDK::MergeSplitHandler::getProperties( geometry );
// 2.
agxAssert( geometryProperties == agxSDK::MergeSplitHandler::getProperties( rb ) );
```

Also, in the call to *getOrCreateProperties*, given a geometry, the merge split properties (if created) of the parent rigid body (if present) will be cloned. Continuing the example from above (remember *merge* is set to be enabled for *rb*):

```
agxSDK::MergeSplitProperties* newProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( geometry );
// We should get a new instance of the merge split properties.
agxAssert( newProperties != geometryProperties );
// The new properties should be a clone of 'rbProperties'.
agxAssert( newProperties->getEnableMerge() == rbProperties->getEnableMerge() );
```

31.3 Merge Split Thresholds

Similar to `agxSDK::MergeSplitProperties`, each object may carry a set of parameters/thresholds that controls the behavior when merging and splitting the object. The global (simulation specific) merge split thresholds are used by default.

31.3.1 Global (simulation specific) merge split thresholds

The `agxSDK::MergeSplitHandler` has the default thresholds used for objects without explicitly created thresholds:

```
auto globalContactThresholds = simulation->getMergeSplitHandler()->getGlobalContactThresholds();
// Set global contact thresholds for all objects without explicitly created thresholds.
configureGlobalContactThresholds( globalContactThresholds );
```

31.4 Object specific merge split thresholds

All objects that may have `agxSDK::MergeSplitProperties` may have a list of thresholds. E.g., `agx::RigidBody`, `agxCollide::Geometry`, `agxWire::Wire` and `agx::Constraint`. The thresholds may be accessed, created and removed via the merge split properties API.

There are two ways to assign object specific thresholds:

```
auto rbProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( rb );
// By default the thresholds should be null meaning global thresholds are used.
agxAssert( rbProperties->getContactThresholds() == nullptr );
auto explicitContactThresholds = new agxSDK::GeometryContactMergeSplitThresholds();
rbProperties->setContactThresholds( explicitContactThresholds );
```

```
auto rbProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( rb );
// By default the thresholds should be null meaning global thresholds are used.
agxAssert( rbProperties->getContactThresholds() == nullptr );
auto explicitContactThresholds = rbProperties->getOrCreateContactThresholds();
```

To remove the thresholds and go back to the global default, simply assign null:

```
rbProperties->setContactThresholds( nullptr );
```

The explicit thresholds instance may be shared arbitrarily:

```
auto explicitContactThresholds = new agxSDK::GeometryContactMergeSplitThresholds();
auto rbProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( rb );
auto wireProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( wire );
auto geometryProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( geometry );

rbProperties->setContactThresholds( explicitContactThresholds );
wireProperties->setContactThresholds( explicitContactThresholds );
geometryProperties->setContactThresholds( explicitContactThresholds );
```

31.5 Wire merge split thresholds

The wire merge split properties and thresholds are shared with all objects created/owned by the wire instance. There are currently two occasions with predefined behaviors:

1. **Cutting a wire**
   When cutting a wire the properties and thresholds for that wire are cloned, i.e., new instances of the properties and thresholds are created. The newly created wire will have identical values as before but when one, e.g., changes thresholds of the original wire it won't affect the part that has been cut from the original one.
2. **Merging wires**
   When merging two wires `wire1->merge( wire2 )` the properties and thresholds of `wire1` will be used for the whole wire.

31.6 Merge conditions

Two objects may merge when the objects involved has merge split properties and property *merge* enabled.

```
agxSDK::MergeSplitHandler::getOrCreateProperties( rb1 )->setEnableMerge( true );
agxSDK::MergeSplitHandler::getOrCreateProperties( rb2 )->setEnableMerge( true );
```

31.6.1 Resting contact

The contacts points carries three directions (an orthonormal basis), namely the normal N, the tangent U and the tangent V directions. U is the primary friction direction and V the secondary friction direction. Given two rigid bodies

$$rb_1 = \{p_1^{cm}, v_1, \omega_1\},$$
$$rb_2 = \{p_2^{cm}, v_2, \omega_2\},$$

where p_i^{cm} is center of mass position, v_i the linear velocity and ω_i the angular velocity. The speed s_d along a given direction d at point p_d is given by

$$s_d = d \cdot \left[v_1 + \omega_1 \times (p_d - p_1^{cm}) - [v_2 + \omega_2 \times (p_d - p_2^{cm})]\right].$$

Given the speeds s_N, s_U and s_V along each direction of the contact points p_i, the two objects may merge if:

$$\max_{p_i} |s_N| \leq \epsilon_N$$
$$\max_{p_i} |s_U| \leq \epsilon_{UV}$$
$$\max_{p_i} |s_V| \leq \epsilon_{UV}$$

Where ϵ_N is called MAX_RELATIVE_NORMAL_SPEED and ϵ_{UV} MAX_RELATIVE_TANGENT_SPEED. Since the conditions above recovers the rolling condition there's a third condition regarding rolling,

$$\|\omega_1 - \omega_2\| \leq \epsilon_\omega,$$

and ϵ_ω is named MAX_ROLLING_SPEED.

```
simulation->getMergeSplitHandler()->setThreshold( agxSDK::MergeSplitHandler::MAX_RELATIVE_NORMAL_SPEED, nS );
simulation->getMergeSplitHandler()->setThreshold( agxSDK::MergeSplitHandler::MAX_RELATIVE_TANGENT_SPEED, tS );
simulation->getMergeSplitHandler()->setThreshold( agxSDK::MergeSplitHandler::MAX_ROLLING_SPEED, rS );
```

When two objects merge due to a resting contact state, the current contact data (such as contact point, normal, normal- and friction forces) is stored and is used when the objects splits and to determine if they should split due to external interactions.

31.6.2 Constraint in a steady state

Similar to contact points, constraints keeps track of the speeds at the anchor point of the constraint. Since constraints (like Hinge, Prismatic, LockJoint etc.) are persistant on a different level compared to contacts, it's possible to include 'time' dependent data. It's not time in seconds, rather "this constraint has been under constant load for some time - merge!". The 'time' concept is introduced by using an Exponential Moving Average (EMA) operation on the relative speeds.

Given a statistic S and an i'th observation O_i the i'th EMA statistic is given by

$$S_i = \alpha O_i + (1 - \alpha) S_{i-1}$$

where $0 \leq \alpha \leq 1$ is a smoothing factor, controlling the sensitivity of the new observations. When α is small, the relative speeds has to be small for a (much) longer time than when α is close to one.

Note: α and other thresholds regarding merge and split of constraints are currently not exposed to the user. Work in progress.

31.7 Split conditions

An object may split, i.e., leave a merged state, if it has merge split properties and property *split* is enabled.

```
agxSDK::MergeSplitHandler::getOrCreateProperties( rb )->setEnableSplit( true );
```

31.7.1 Impacting contact

There're two different concepts to choose from:

- Physical impact
- Logical impact

The logical impact is determined by the state of the contact. I.e., if the state were "no contact" and the geometries are overlapping, the state becomes "impact". It's at this state the agxSDK::ContactEventListener::impact callbacks are executed.

The physical impact condition is tied to a speed threshold, preventing objects from splitting too often.

The logical impact concept is disabled by default. If enabled, the merged object will split when the state of the contact is "impact". If the state is different from "impact" the physical impact approach is testing the contact

The impact approach uses the relative speed s_N (from the Resting contact section), considering the sign of the speed as well:

$$\min_{p_i} s_N \leq -\epsilon_{impact}$$

Note that s_N is positive when the objects are separating and negative when they're approaching each other. The latter is the one we're catching and $\epsilon_{impact} \geq 0$ is a threshold called MAX_IMPACT_SPEED.

```
// Any value != 0.0 means enable, 0.0 is disable.
simulation->getMergeSplitHandler()->setThreshold( agxSDK::MergeSplitHandler::LOGICAL_IMPACT, 0.0 );
// Using pure physical impact splits at speeds > 0.01 m/s.
simulation->getMergeSplitHandler()->setThreshold( agxSDK::MergeSplitHandler::MAX_IMPACT_SPEED, 1.0E-2 );
```

```
// Using pure logical impacts...
simulation->getMergeSplitHandler()->setThreshold( agxSDK::MergeSplitHandler::LOGICAL_IMPACT, 1.0 );
// ...and disabling physical impacts.
simulation->getMergeSplitHandler()->setThreshold( agxSDK::MergeSplitHandler::MAX_IMPACT_SPEED, agx::Infinity );
```

31.7.2 Constraints

31.7.2.1 Changing state

Constraints can split bodies when the internal state of the constraint has been changed in a way that the dynamics may be affected. Currently, this functionality is focused to the secondary constraints/controllers and the constraint will split the bodies involved when:

- The motor is enabled with speed different from zero.
- The position of the lock has been changed (i.e., by calling constraint->getLock1D()->setPosition(newPosition)).
- The range of the range controller has been changed.
- When a controller state goes from inactive to active, e.g., when a range is hit or a lock is enabled.

It's, unfortunately, often not enough to just split the two bodies in the constraint. Consider an articulated, hydraulic crane where the whole structure is merged. To rise the boom the prismatic motor is activated, but nothing will move since it's only the hydraulic cylinder and piston that are free to move.

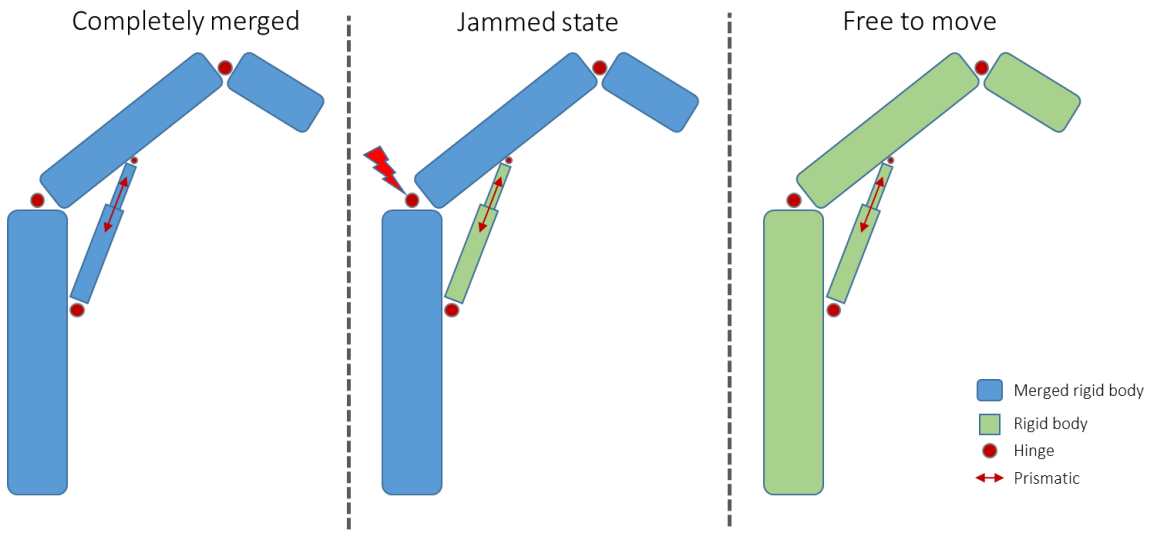


Figure 44: Articulated crane where a prismatic motor is used to rise the boom.

What happens is that the split algorithm first splits the two bodies directly involved in the constraint. The structure is now in a "jammed state" where it still cannot move. The split algorithm contines to traverse the merged structure, splitting all constrained bodies it finds. The split-traversal stops when it reaches a body that may not be split or when the edges connecting two bodies aren't related to constraints (i.e., different from agx::MergedBody::BinaryConstraintEdgeInteraction).

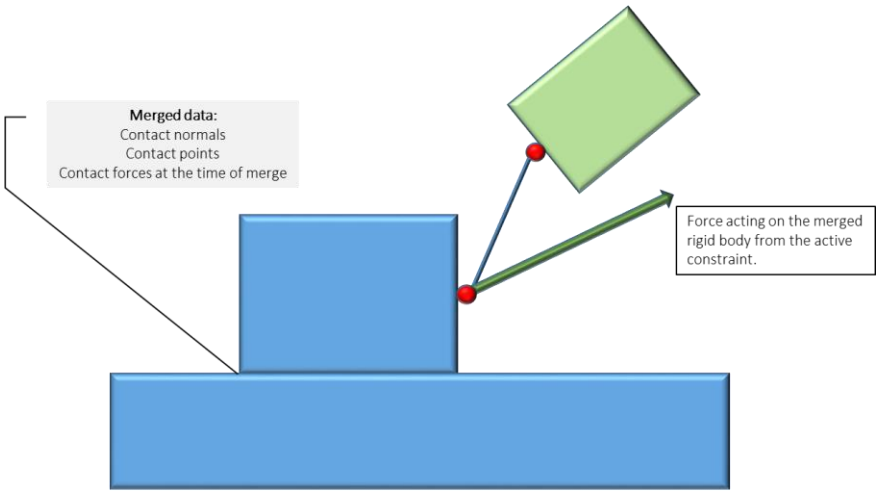
If, for example, hundreds of these cranes were to stand on a static ground or a dynamic ship, it's important that the ground or the ship doesn't have merge split property *split* enabled. In general there aren't any reasons to have that property enabled for "parent" objects.

31.7.2.2 Applying forces

When a constraint are applying forces on a merged object, it may split that object if that object has merge split property *split* enabled and is merged due to resting contacts.

Mentioned in the Resting contact section – contact and interaction data is saved when two objects merge. This data can be used to approximately determine "how much" external force is needed for the object to start to move, and we can call it *the strength of a contact edge*.

When the force exceeds the contact edge strength, the edge can be considered removed and if all edges are removed, the objects splits.



There's a utility function to check if a merged object should split given external forces:

```
// We've an external force acting on 'rb', which is merged.
// Check if this external force is large enough to split it.
auto splitInfo = agxSDK::MergeSplitUtils::checkSplitGivenExternalForce( rb,
                                                                        agx::MergedBody::get( rb ),
                                                                        externalForce,
                                                                        ignoredParameter );

if ( splitInfo.shouldSplit )
  agx::MergedBody::split( rb );
```

31.8 agx::MergedBody with agxSDK::MergeSplitHandler

As mentioned earlier, the agxSDK::MergeSplitHandler manages agx::MergedBody objects. If the agxSDK::MergeSplitHandler encounter a merged rigid body, but the rigid body wasn't merged by the handler – the rigid body is ignored.

This enables the possibility for the user to manage their own merged bodies. It can, for example, be used to initialize big scenes in a merged state.

For example – create a pile of boxes, enable merge and split and merge them in an agx::MergedBody:

```
// Create and add a merged body.
agx::MergedBodyRef mergedBody = new agx::MergedBody();
simulation->add( mergedBody );

agx::RigidBodyRef prevBox = nullptr;
for ( agx::UInt i = 0; i < numBoxes; ++i ) {
  // Create, position and add the box to the simulation.
  agx::RigidBodyRef box = createBoxInPile( i, simulation );
  // Enable merge+split for the box.
  agxSDK::MergeSplitHandler::getOrCreateProperties( box )->setEnableMergeSplit( true );
  // Merge with previous box.
  if ( prevBox != nullptr )
    mergedBody->add( new agx::MergedBody::ContactGeneratorEdgeInteraction( prevBox, box ) );
  prevBox = box;
}
```

Running this simulation, given a ground object with property *merge* enabled, the set of merged boxes will interact with the ground object, but not merge nor split.

For the agxSDK::MergeSplitHandler to be able to merge and/or split these explicitly merged objects, their agx::MergedBody has to be registered:

```
// Register 'mergedBody' to the handler. The objects continues to
// be merged after this call.
simulation->getMergeSplitHandler()->registerMergedBody( mergedBody );
```

Note: After an agx::MergedBody has been registered to the agxSDK::MergeSplitHandler, it's undefined behavior to add/remove edge interactions/bodies from the agx::MergedBody object.

32 MergeSplit

We recommend that you use the new Merge functionality described in chapter 31. It is more general and support added masses, constrained systems etc.

MergeSplit is a method that can be used to increase the performance in a scene with a medium number of simple/non-constrained bodies. A typical scenario is a tractor shuffling around rocks on a height field. Having all rocks/bodies dynamically simulated the whole time would generate a large number of contact constraints, which in turn would affect the performance negatively. MergeSplit will consider bodies which are in contact and analyze if bodies can be merged into one body containing multiple geometries, or have to be split into smaller parts again.

MergeSplit has the advantage over AutoSleep that it can be used for reducing complexity in scenes where bodies are in contact with a dynamic object.

Assume you are simulating a truck. A wheel loader is loading rocks on the bed of the truck, now even if the truck moves, bodies can still be merged with the bed of the truck. This means that instead of having hundreds or thousands of bodies simulated on the bed, there are only a few, as they are merged into one body. The mass, inertia and center of mass will still be accurate.

The criteria for a merge are:

- One or more of body *A*'s geometries are in contact with one or more of body *B*'s geometries.
- *AND* $|A.\text{velocity} - B.\text{velocity}|^2 <$ velocity threshold
- AND $|A.\text{acceleration} - B.\text{acceleration}|^2 <$ acceleration threshold
- One of the bodies is not attached to an enabled constraint (Hinge, LockJoint etc…)

In other words when the two bodies *A* and *B* can be considered moving as *one* rigid body, they are merged into one single rigid body. If Body1 is STATIC, it will become the "parent" of Body2. If Body1 is constrained (has a constraint associated to it) it will become the parent of Body2 and vice versa.

A constrained body can never become a "child" of another body.

This means that if a dynamic body falls onto a static ground, the dynamic body will be part of the static ground.

When a body *A* is merged into another body *B*, *A* will still get its transformation updated when *B* moves. Body *A* will however be disabled (not part of the dynamic simulation).

MergeSplit is enabled by a call to:

```
agxSDK::SimulationRef simulation = new agxSDK::Simulation;
agx::MergeSplit *mergeSplit = simulation->getDynamicsSystem()->getMergeSplit();
// Enable MergeSplit functionality
mergeSplit->setEnable(true);
```

MergeSplit also has to be enabled for each body:

```
agx::RigidBodyRef body = new agx::RigidBody
// Tell this body it is allowed to be merged with other bodies
body->getMergeSplitProperties()->setEnableMerge(true);
```

If the scenario is that bodies are falling onto a static ground, it is more efficient to only allow the bodies to be merged with the static ground, instead of with each other. So to build a large pile of bodies on a static ground, we can do the following call for each body:

```
// We can also tell a body ONLY to merge with STATIC bodies
body->getMergeSplitProperties()->setMergeOnlyWithStatic(true);
```

Bodies can also be explicitly merged/split:

```
// Merge body1 into body2
bool wasMerged = mergeSplit->merge(body1, body2);

// Split body1 from its parent
bool was Split = mergeSplit->split(body1);
```

It is important to remember that MergeSplit is modifying relations for geometries. Whenever a body is merged into a parent, geometries are transferred to the parent. Whenever a body is split, it will get its original geometries restored. Given a geometry, it is possible to ask which its original body was (before it was merged):

```
// If this geometry belongs to a merged body, the original body will be returned
// Otherwise NULL.
agx::RigidBodyRef originalBody = merged_geometry->getOriginalBody();
```

32.1 Merging

Merging two bodies from Body1 -> Body2 means the following:

Body *A* has the mass properties: [m_a, I_a(inertia tensor), and Cm_A (center of mass)] and the geometry G_a. Body *B* has respectively the mass properties: [m_B, I_B, and Cm_B] and geometry G_B.

After merging *A* into *B* we have: Body A with the mass properties: m_A+m_B, I_A+I_B and Cm_A+Cm_B. Body *B* will now be disabled which will save CPU. The geometry G_A will be transferred over to body *A* which will now in all senses act as the union between body *A* and body *B*.

32.2 Splitting

Splitting can occur when:

- a separation between geometries occur
- an impact between a the geometries of merged body and another geometry occur
- An explicit split using the MergeSplit API.

32.3 Known limitations

- Body attributes such as damping, Properties are not merged

33 Statistics

AGX has several methods for measuring time and obtain statistics about timings in different parts of the simulation.

33.1 agx::Timer

The agx::Timer can be used to measure the duration of the execution of compute-intensive code. The timer is based on hard-ware, using either the hardware's Time Stamp Counter or High Precision Event Timer (platform-dependent, please look at *include/agx/Timer.h* for details).

Timers can be started, stopped, (re-)started, or reset. They can be queried for their current time, which is given in milliseconds. The precision is between 100s of nanosecons and microseconds (depending on hardware and operating system).

An example of the usage:

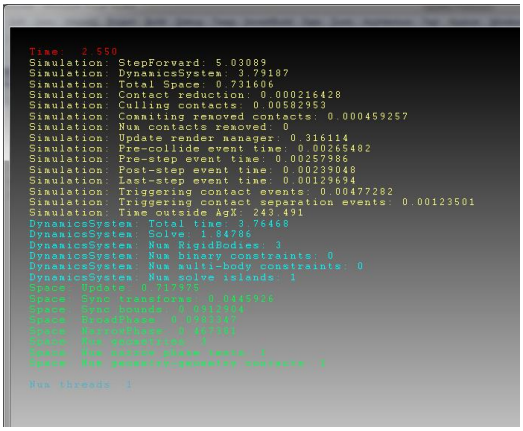
```
// Create a timer. It will started in the stopped (= "paused") state by default.
agx::Timer t;
//...do some thing else...
t.start();
// We want to measure this part of the code, call it section 1.
t.stop();
// We do NOT want to measure this part of the code, call it section 2.
t.start();
// Again, we want to measure this part of the code, call it section 3.
t.stop();
std::cout << "Section 1 and 3 together took " << t.getTime() << "ms.\n";
t.reset();
// Now, the timer is set to 0 again, and stopped. It can be used for other purposes now.
```

33.2 agx::Statistics

AGX has internal functionality for registering and reporting runtime statistics of a simulation. agx::Statistics is a singleton and can be accessed through a call to:

```
agx::Statistics::instance()
```

Various classes register into the statistics singleton and report both timing and other data related to statistics. The collection of data can be accessed in various ways. Through the debug rendering as overlay in the graphics of your application:



The statistics information presented in the graphics window is a subset from the complete set of data points stored. What is displayed in the graphics window can be changed in the file *settings.cfg* which should be located in <agx-dir>/data/cfg/settings.cfg.

To enable statistics rendering in the graphics window do the following calls:

```
agxSDK::SimulationRef sim = new agxSDK::Simulation;
agx::Statistics::instance()->setEnable( true );
sim->getRenderManager()->enableFlags(agxRender::RENDER_STATISTICS);
```

Or slightly more compressed:

```
sim->setEnableStatisticsRendering ( true );
```

If you want to get all the statistics information into a file, you can specify that through the simulation class:

```
agxSDK::SimulationRef sim = new agxSDK::Simulation;
sim->setStatisticsInterval( 1 ); // report every second

// Specifies a path to a file that will contain statistics info
sim->setStatisticsPath( "myStat.txt" );

sim->setEnableStatistics( true ); // Enable statistics reporting to file
```

This behavior can also be controller through environment variables read by the Simulation class when it is instanced. For more information see section 41 (Environment variables).

33.3 Statistics API

It is possible to readout the current statistics value after a call to Simulation::stepForward(). This is done through the singleton **agx::Statistics::instance()**

```
const char *module = "Simulation";
const char *data = "Step forward time";

agx::TimingInfo v = agx::Statistics::instance()->getTimingInfo(module, data);
double value = v.current; // The last reported value
double accumulated = v. accumulated; // Accumulated value since start of simulation
```

Statistics also collect other data, such as number of rigid bodies etc.:

```
const char *module = "Space";
const char *data = "Num narrow phase tests";

agx::UInt32 value=0;
// Access the abstract statistics data
agx::Statistics::AbstractData *ad = agx::Statistics::instance()->getData(module, data);
if (ad) { // Was the data available?
  agxData::Value *v = ad->toValue();
  value = v->get<agx::UInt32>(); // Get the actual value
}
```

The available data can be queried by enabling statistics, running an application with AGX Dynamics and look at the generated statistics file (agx_statistics.txt):

```
cmd> set AGX_STATISTICS_ENABLE=1
cmd> set AGX_STATISTICS_INTERVAL=1
cmd> tutorial_constraints
```

Below is a snippet from that file. Red is the name of the *module,* and bold black is the name of the *data*. These strings can be used in the code listed above to receive the statistics data from a running simulation.

```
*** Time step 3.05 ***
Root
- Children:
  Simulation
  - Timings:
    Inter-step time: 14.5213 (2608.82)
    Step forward time: 2.3789 (351.901)
    Dynamics-system time: 1.42312 (198.722)
    Collision-detection time: 0.787439 (116.886)
    Pre-collide event time: 0.0135979 (12.3506)
    Pre-step event time: 0.0120962 (1.77996)
    Triggering contact events: 0.0148825 (1.57511)
    Post-step event time: 0.0105585 (1.14688)
    Committing removed contacts: 0.00640043 (0.921192)
    Culling contacts: 0.00576196 (0.619406)
    Triggering contact separation events: 0.000594434 (0.115472)
    Last-step event time: 0 (0.0924153)
    Contact reduction: 0.000174818 (0.0256758)
  - Data:
    Num contacts removed: 0 (0)
```

34 Serialization

AGX Dynamics comes with a serialization system. This allows for a simulation to be stored on disk for later retrieval. Together with a callback system at restore, a stored simulation can be recreated together with custom references. This is very handy when debugging a system for example. If a problem occurs, a serialization of the simulation might be informative during debugging to locate any potential problems in the setup of the simulation. We recommend that a serialization is attached to reported tickets so that Algoryx can use that when analyzing the problem.

The serialization system will collect any references that a serialized object has. Meaning that serializing a rigid body will also store any geometries, shapes, materials that is associated to this rigid body. So it is not in general possible to do a partial store.

Serialization can be done to an ASCII (XML) file ".aagx", or a binary file ".agx". The ASCII format is not suitable for very large scenarios with huge height fields for example due to the size of the file. The binary file format is much faster and more compact and can in many case be written during runtime without a major effect on performance.

34.1 Universally unique identifier (UUID)

Each object derived from the base class **agxStream::Serializable** will inherit the property of a uuid. This is true for RigidBody, Geometry, Constraint, agxWire::Wire and many other classes.

When a simulation is stored in the serialization system, this uuid will be stored together will all the serialized objects. During restore, this uuid can be retrieved and used to identify objects. For an example, see 34.5.

34.2 Storing a simulation

To store an existing simulation there exists several options depending if you want to store it to memory or a file on disk. The simplest way is to use the writeFile function:

```
#include <agxIO/ReaderWriter.h>
// Create a simulation
agxSDK::SimulationRef simulation = new agxSDK::Simulation;
// Populate the simulation with bodies, geometries etc.
populateSimulation( simulation );
// Write to a file
if (!agxIO::writeFile("simulation.agx", simulation))
  error(…)
```

The file on disk will contain everything related to the simulation:

- Rigid bodies
- Geometries
- Shapes
- Constraints
- Wires
- GravityField
- Time step
- Current simulation time
- Collision groups

- ...

The serialization will contain the current state of the whole simulation including velocities, tension, constraint violations etc. This means that restoring the simulation from the file will lead to an identical state as when it was stored.

Notice that restoring multiple serializations to a simulation *might* give you an unpredicted result. For example if one simulation disables two collision groups and another enables them, it will be the last restored simulation that dictates the result. The same goes for time step, gravity fields etc...

It is possible to filter out some of the data when performing a restore with the agxSDK::Simulation::ReadSelectionMask enum:

```
enum ReadSelectionMask {
  READ_NONE = 0x0,      ///<! Select to read none of the items below
  READ_TIMESTEP = 0x1,  ///<! Select to read and restore the TimeStep of the Simulation.
  READ_TIMESTAMP = 0x2, ///<! Select to read the TimeStamp (current time of the simulation)
  READ_GRAVITY = 0x4,   ///<! Select to read and restore the Gravity model of the simulation.
  READ_ALL = READ_TIMESTEP + READ_TIMESTAMP + READ_GRAVITY,
  READ_DEFAULT = READ_TIMESTEP /* + READ_TIMESTAMP */ + READ_GRAVITY
};
```

```
// Do not restore timestep, timestamp nor gravity.
agxIO::readFile("mystoredsimulation.agx", simulation, 0L, agxSDK::Simulation::READ_NONE);
```

Notice that the TIMESTAMP (current time) is not restored by default. So if you are restoring to an existing simulation which has been simulated for a while, the current time will not be affected.

34.3 Serialization to/from memory

To serialize a simulation to memory, first create a stream and then use the **Simulation::write** method:

```
// Create a memory stream, make sure it is in binary format to avoid NL/CR problems
std::stringstream os(std::ios_base::out | std::ios_base::binary);

// Now write the simulation to the stream. The second boolean argument determines if the
// serialization format should be binary or ASCII. Use ASCII for faster
// process and smaller memory footprint.
bool binaryFormat = true;
simulation->write(os, binaryFormat); // Serialization data is stored into the stream
```

To restore from memory, use an input stream and the **Simulation::read** method. In this case we copy data from the output stream to a new input stream, but it could just as easily been read from disk or any other location:

```
// Restore the simulation from memory
os.seekg(0);

// We need to get the data into a istream to be able to read from it
std::stringstream is(std::ios_base::in | std::ios_base::binary);
is.str(os.str());

// Now read from the binary stream.
simulation->read(is, binaryFormat);
```

34.4 Continuous serialization to disk

It is also possible to continuously serialize a simulation to disk during runtime. This might be helpful when debugging a problematic simulation. This will however have a substantial effect on performance, depending on the size of the simulation. For example, a large height field contains a lot of data per frame, hence it will severely slow down your simulation.

To enable this frame by frame simulation, you can use the API:

```
// Enable the serializer
simulation->getSerializer()->setEnable(true);
// Specify an interval for writing a frame
simulation->getSerializer()->setInterval(1);
simulation->getSerializer()->setFilename("frames.agx"); // 00001_frames.agx, 00002_frames.agx
```

Or change the **Enabled** value in the configuration file <agx-dir>/data/cfg/settings.cfg:

```
SimulationSerializer
{
  Enabled 1 // If ==1 then for each frame a file will be dumped with complete simulation
content
  Mode 1 // When to write data: 0-PRE_COLLIDE, 1-PRE_STEP, 2-POST_STEP
  Interval 0.03333 // interval in seconds
  Filename "agx.agx" //.aagx for ascii, .agx for binary (faster/smaller)
}
```

With the settings above, a file named 00001_agx.agx, 00002_agx.agx will be written to disk with an interval of 33ms (30 Hz).

These files can then be packed and set to Algoryx for further analysis regarding any issue that might have surfaced during the simulation.

34.5 Listening to restore events

It is possible to listen to a restore process and get callbacks when an object is being restored. This allows for custom connection between rigid bodies, geometries, constraints wires etc. and your own system.

In **tutorial_io.cpp** there is an example of how to associate rendering information for each restored geometry. Notice also that each stored object will have a uuid which is also restored. This can be used to store a reference in form of a text string (uuid, usually in the form of `"8d276495-19b6-44d8-84ba-5c1830ee97a4"`) which later during the restore can be used to uniquely identify a restored object.

So if one has a GUI system with representations of a rigid body, the GUI system could store the uuid of the rigid body in its own file storage system. When an AGX simulation is serialized to disk, this uuid is also stored together with the rigid body.

Later at restore, the GUI representation of a rigid body can be created and the reference to the rigid body can be established based on this uuid.

35 Debug rendering

AGX has an internal scheme for rendering rigid bodies, geometries and constraints (including contacts). It is handled by a class **agxRender::RenderManager**. When debug-rendering is enabled, this class will dispatch calls for rendering and create **agxRender::RenderProxy's**. A render proxy is a placeholder for a class that can be rendered in the clients rendering system. By default if AGX is built with OpenSceneGraph, there is an implementation of render proxies for that scene graph.

Each primitive has its own render proxy which need to be implemented at the client side. Responsible for creating these proxies is the **agxRender::RenderProxyFactory**, a class which is derived to a specialization for each rendering system. The class **agxOSG::RenderProxyFactory** is one such specialization for OSG.

This rendering system should not be considered to be the rendering system for your simulations in simulators etc. It is merely a system for debugging your simulation, being able to see if bodies are enabled, static, kinematic, sleeping. To see where your constraints are attached to bodies etc. When you need a proper rendering engine, you should connect that directly through the geometries and rigid bodies.

Class	Description
agxRender::RenderManager	Manages the debug rendering system. Called from agxSDK::Simulation::stepForward()
***agxRender::RenderProxyFactory**	Abstract base class responsible for creating agxRender::RenderProxy for rendering in a specific rendering system.
agxRender::RenderProxy	Abstract base class for all renderable objects. Derived down to various primitives (sphere, cylinder, etc.)
***SphereProxy, *CylinderProxy, *LineProxy, *TextProxy, *PlaneProxy, *CapsuleProxy, *BoxProxy, *HeightfieldProxy, *TrimeshProxy**	Specialization of a RenderProxy for each supported shape.
agxRender::GraphRenderer	An abstract class for rendering graphs related to statistics information.

Table 23: Classes for rendering. (*) indicates classes that need to be implemented and adopted for your specific rendering engine.

Objects with different properties are rendered differently: geometries belonging to static bodies are rendered in blue, whereas geometries associated to dynamic bodies are rendered in green. Sensors are rendered with a lighter blue.

35.1 RenderManager

Each simulation has their own instance of the class agxRender::RenderManager. This class is responsible for updating all the RenderProxy instances that are created during debug rendering. Each render manager need a reference to a RenderProxyFactory, specific for a rendering engine. Several render managers can use the same RenderProxyFactory, so if you are using several agxSDK::Simulation, you can assign the same RenderProxyFactory to those simulations.

```
// Create a render proxy factory, specific for OpenSceneGraph
agxRender::RenderProxyFactoryRef factory = new agxOSG::RenderProxyFactory;

// Create a simulation
agxSDK::SimulationRef sim = new agxSDK::Simulation;

// Tell this simulation's render manager to use our factory
sim->getRendermanager()->setProxyFactory( factory );

// Create another simulation, use the same factory for the debug rendering
agxSDK::SimulationRef sim2 = new agxSDK::Simulation;
sim2->getRendermanager()->setProxyFactory( factory );
```

During a Simulation::stepForward(), a call to RenderManager::update() will be called which in turn will query the render proxy factory for spheres, lines, cylinders etc.

There is no cache functionality implemented in the RenderManager, that is, when a render proxy is required, a call to the render proxy factory will be executed. There will be no storage of unused render proxies in the render manager. If caching is required (to achieve better performance for scenes where the number of proxies varies a lot), it has to be done in the render proxy factory implementation

35.2 Render flags

There are three methods that control what should be rendered by the render manager:

```
void setFlags( unsigned int flags );
void enableFlags( unsigned int  flags );
void disableFlags( unsigned int  flags );
```

setFlags will override the current set of flags with the specified one. *enableFlags* will enable the specified flags and leave the rest untouched. *disableFlags* will disable the specific flags and leave the rest untouched.

As an example, to specify that the default set of flags should be used, plus the rendering of bounding volumes (which is not enabled by default) the following call can be done:

```
// Render default, plus the bounding volumes
simulation->getRenderManager()->setFlags(agxRender::RENDER_DEFAULT|
                                         agxRender::RENDER_BOUNDING_VOLUMES);
```

35.3 Renderable objects

There are various types of objects that will be rendered in the debug rendering:

35.3.1 Shapes

All shapes (agxCollide::Shape) for all enabled geometries will be rendered. Every time a new shape is added/removed from a geometry part of the simulation, it will be added/removed from the debug rendering system (if it is enabled). A proxy will be associated for each shape. When the shape is deleted/removed, the proxy will get a call to onChange(REMOVE) (see below). For each time step, the state (color, alpha) and shape (size/form) will be synchronized with the corresponding AGX shape. For most of the time, only transform changes will be updated for these shapes, which should be efficiently

handled by the rendering engine. The rendering of shapes will occur of the flag RENDER_GEOMETRIES is enabled in the render manager.

35.3.2 Constraints

The constraints in AGX has a virtual *render* method. This method will be called whenever the RENDER_CONSTRAINTS flag is enabled in the render manager. These render methods, will query the render manager for various shapes such as cylinder, spheres or lines. These queries will be dispatched to the specific RenderProxyFactory.

35.3.3 Renderables

There is a special class, agx::Renderable, which can be used for some specific rendering. It has a virtual method *render* which will be called from the RenderManager if the RENDER_RENDERABLES flag is enabled in the render manager.

35.3.4 Bodies

For each enabled RigidBody a sphere proxy will be rendered. These sphere proxies will be acquired via the render manager to the specific render proxy factory. This will occur if the RENDER_BODIES flag is enabled.

35.3.5 Contacts

Contacts will be rendered analogous to bodies. A sphere and a line will be acquired from the render manager. This occurs if the flag RENDER_CONTACTS is enabled.

35.3.6 Statistics

Statistics involve both the textual information (as collected by the agx::Statistics class) and graph drawing, showing some of the statistics as visual graphs on the screen. This occurs if the RENDER_STATISTICS is enabled.

35.4 Implementation of custom debug rendering

35.4.1 agxRender::RenderProxyFactory

This class need to be specialized for any specific rendering engine. In <agxOSG/RenderProxy.h> an implementation for OpenSceneGraph can be found. The virtual methods that need to be implemented are:

```
/// Interface for creating and returning a SphereProxy
virtual SphereProxy *createSphere( float radius ) = 0;

/// Interface for creating and returning a BoxProxy
virtual BoxProxy *createBox( const agx::Vec3& halfExtents ) = 0;

/// Interface for creating and returning a LineProxy
virtual LineProxy *createLine( const agx::Vec3& p1, const agx::Vec3& p2 ) = 0;

/// Interface for creating and returning a CylinderProxy
virtual CylinderProxy *createCylinder( float radius, float height ) = 0;

/// Interface for creating and returning a ConeProxy
virtual ConeProxy *createCone( float radius, float height ) = 0;

/// Interface for creating and returning a CapsuleProxy
```

```
virtual CapsuleProxy *createCapsule( float radius, float height ) = 0;

/// Interface for creating and returning TextProxy
virtual TextProxy *createText( const agx::String& text, const agx::Vec3& pos ) = 0;

/// Interface for creating and returning PlaneProxy
virtual PlaneProxy *createPlane( const agx::Vec3& normal, agx::Real distance ) = 0;

/// Interface for creating and returning HeightfieldProxy
virtual HeightFieldProxy *createHeightfield( const agxCollide::HeightField *hf ) = 0;

/// Interface for creating and returning TrimeshProxy
virtual TrimeshProxy *createTrimesh( const agxCollide::Trimesh *mesh ) = 0;
```

Each of these methods are responsible for returning a RenderProxy of a specific type based on the in-data. The RenderProxy is then responsible for holding this renderable reference into the render system in use. As an example, the code for creating a sphere OpenSceneGraph looks like this:

```
agxRender::SphereProxy* RenderProxyFactory::createSphere( float radius )
{
  ShapeData<osg::Sphere> sphereData;
  sphereData.shape = new osg::Sphere();
  sphereData.shape->setRadius( radius );
  sphereData.geode = createGeode( sphereData.shape, &sphereData.shapeDrawable );
  sphereData.geode->setName("SphereGeode");

  // Create a new Sphere proxy including the rendering representation in OSG
  agxOSG::SphereProxy *proxy = new agxOSG::SphereProxy( radius, sphereData, this );

  addChild( proxy->getNode() ); // Add the node to the scenegraph (hold by the factory)

  return proxy; // Return the proxy
}
```

The specific data required will differ between rendering engines.

35.5 agxRender::RenderProxy

Each subclass (primitive type) of RenderProxy need to be implemented for the specific render engine. The virtual methods that can/should be implemented are:

35.5.1 onChange

This method need to be implemented (pure virtual method) in your representation of a RenderProxy. It will receive calls from each of the set methods described below. Whenever a proxy is required to change color, shape, alpha or transform, this method will be called. Based on the event-type described in the enum agxRender::RenderProxy::EventType, you should update your rendering representation with the new current value. For the OpenSceneGraph implementation a code snipped looks like the following:

```
void onChange( RenderProxy::EventType type) {
  switch( type ){
      case (RenderProxy::ENABLE):
      setEnableOSG(getEnable());
      break;
      case (RenderProxy::ALPHA):
      setAlphaOSG(getAlpha());
      break;
      case (RenderProxy::TRANSFORM):
      setTransformOSG(getTransform());
      break;
...
```

So based on the event type different calls are done to the rendering API to reflect the changes into the rendering implementation.

35.5.2 updateShape

This method need to be implemented (pure virtual method) need to be implemented in your representation of a RenderProxy. It is specific for each shape. When this method is called (probably from your specific implementation of onChange(SHAPE)) the render proxy implementation need to update the rendering representation to reflect changes, such as changing radius for a sphere, change the triangle mesh structure or changing the size of a box.

A code snippet from the OpenSceneGraph representation illustrate how it can be done:

```
void SphereProxy::updateShape( )
{
  // Get the agx representation of the shape associated to this proxy
  const agxCollide::Sphere *sphere = getCastShape();
  if (sphere) // If there IS a shape associated, then read the radius from that
    m_radius = sphere->getRadius();

  // If radius is the same, then just dont do anything
  if (agx::_equivalent((float)m_radius, m_data.shape->getRadius()))
    return;

  // Change the radius of the osg-sphere
  m_data.shape->setRadius(m_radius);
  m_data.geode->getDrawable(0)->dirtyDisplayList();
}
```

35.5.3 setTransform

```
virtual void RenderProxy::setTransform( const agx::AffineMatrix4x4& transform );
```

This method will set the m_transform member of the RenderProxy, then it will call onChange(TRANSFORM) which indicates that the transform is changed. If m_transform == transform, no call to onChange will occur (to avoid unnecessary state changes. If you override this method, store transformation in m_transform and call onChange(TRANSFORM).

35.5.4 setColor

Store a color in m_color and call onChange(COLOR). If color == m_color, no call to onChange will occur to reduce state changes. If you override this method, update m_color and call onChange(COLOR).

35.5.5 getColor

This method return the color of the proxy, the value in m_color. You can override this method to return the color as stored somewhere else, just make sure you sync with the m_color member attribute.

35.5.6 setAlpha

Set the transparency value of the proxy. The implementation will store alpha in m_alpha and call onChange(ALPHA). If you override this implementation, you need to update m_alpha, and call onChange(ALPHA).

35.5.7 getAlpha

Return the value of the m_alpha member attribute. If you override setAlpha you might want to also override this method.

35.6 agxRender::GraphRenderer

This class is specific for each rendering engine. An implementation for OpenSceneGraph can be found in <agxOSG/Graphrenderer.h> and corresponding .cpp file.

For more information, see the doxygen generated documentation for the class.

36 File formats

This chapter explains the various file formats available for AGX Dynamics. All of the below listed file formats can be loaded with the application agxViewer.

File type	Description
.agx	Binary serialization of an AGX Dynamics simulation
.aagx	XML serialization of an AGX Dynamics simulation
.agxLua	Lua script using the AGX Dynamics Lua scripting API
.agxPy	Python script using the AGX Dynamics Python scripting API
.agxLuaz	Compressed archive including a .agxLua script file
.agxPyz	Compressed archive including a .agxPy script file

Table 24: File formats

36.1 .agx, .aagx

Chapter 34 Serialization describe the procedure for storing an existing AGX simulation to disk and how to later restore it. The file formats .agx and .aagx are binary and xml versions of these serializations.

36.2 .agxLua .agxPy

When AGX is installed in windows, the file formats .agxLua and .agxPy is associated to agxViewer.exe. These formats are Lua respectice Python scripts. In these file a function named buildScene() will be executed to build up the scene. For more information see respective chapter about Lua and Python scripting.

36.3 .agxLuaz .agxPyz

The file format .agxLuaz and .agxPyz are zipped archives containing a Lua or a Python script of the same name as the name of the archive. It can also contain textures/models and other data files required for the script to run. They can be loaded with the agxViewer application.

For example, the file ship.agxPyz *must* contain a script named: ship.agxPy in the root directory of the archive. This script will be executed by agxViewer when loaded:

```
> agxViewer ship.agxPy
```

Any data accessed by the script must be available in the archive (ship.agxPyz).

A standard zip command or the application agxArchive (see chapter 39.2agxArchive) can be used to create these archives.

37 AGX Python Scripting

Python (www.python.org) is a powerful and popular scripting language for scientific computing and engineering. The main reason behind this is the the vast number of libraries, toolkits and SDK's available.

The AGX Python library comes as SWIG-generated modules and extensions, mirroring the C++ native API as far as possible. Each C++ namespace that is part of the AGX C++ SDK and exposed to Python makes up a Python module.

AGX Python scripts can be run in two ways:

1. *In native mode* via the native, stand-alone Python interpreter
2. In *embedded mode* via an application based on agxOSG::ExampleApplication, such as agxViewer.

Depending on which of the two alternatives you choose, you need to provide the entry points for initialization and execution. This will be explained later.

37.1 Current limitations

AGX Python is only available for the 64-bit version of AGX Dynamics.

Currently version 3.5 or later of Python is required to use AGX Python under Windows. Under Linux version 3.0 or later is required.

37.2 Modules and Library structure

Each C++ namespace of AGX exposed to Python is exposed as its own Python module available for import. For example: agxSDK is encoded in _agxSDK.pyd. These modules are found together with their respective native extension shared library in:

- In Windows, beneath the directory of your AGX installation:
  <PathToAGXInstallation>\bin\x64\agxpy
- In Unix the default path should be something similar to:
  /usr/local/lib/python3.5/dist-packages/site-packages/
  This might be different on your specific Unix flavour.

For Python to be able to locate the AGX Python modules, you might need to set the PYTHONPATH environment variable. See the chapter Environment below.

You can verify the AGX Python installation by creating a Python script that contains the following:

```
import agx
agx.init()
```

If this works, then Python can locate the required libraries for AGX Python.

37.3 ScriptContext

The presence of a ScriptContext in the runtime of a script indicates an environment in which the script is run from some form of embedded mode hosted by a ScriptManager within AGX itself. This means the script is run from an instance of the class agxOSG::ExampleApplication. The ScriptContext is accessible through the agxPython::getContext() function. The following code demonstrates the use of the class:

```
import agx
import agxSDK
import agxPython

if agxPython.getContext() is None:
      # We come from the native, stand-alone Python interpreter
      #
      # This means that we have to create our own agxSDK.Simulation
      # instance in order to be able to accomplish anything, but not
      # before initializing agx using the AutoInit class
      init = agx.AutoInit()
      sim = agxSDK.Simulation()
else:
      # We come from the embedded Python interpreter of preinitialized AGX
      #
      # This means an agxSDK.Simulation instance is provided for us:
      sim = context.getSimulation()
```

37.4 AGX Python coding guide

Most primitive types, classes, functions and methods of the AGX C++ API exposed to Python have wrappers which work as one would expect in Python. For example, wherever a C++ NULL or nullptr would be used, you use None in Python. There are however some things to watch out for, especially when the philosophical and conventional similarities between C++ and Python end.

37.4.1 Proxy classes, their instances and reference counting

All C++ classes exposed to python will be accessible through a *proxy* class. Each namespace of AGX will belong to a separate Python module.

For example the class agx::RigidBody will be located in the module _agxPython which is imported to Python using the *import* keyword. Below is an example of how to import the agx namespace and instantiate a class in that namespace:

```
import agx
rb = agx.RigidBody()
```

C++ classes derived from agx::Referenced are automatically handled by Python, so that as long as there is a C++ agx::ref_ptr *and/or* a Python object referring to it, it will not be deallocated.

37.4.2 Attributes

Attributes added to proxy class instances in Python does not exist outside the Python object to which the attributes belong, and can't pass through the native C++ layer back to Python again. The following code demonstrates this:

```
rb_vector = agx.RigidBodyVector()
```

```
rb = agx.RigidBody()
rb.my_attr = 5.0 # An attribute added to the rb object
rb_vector.append(rb)

rb_retrieved = rb_vector[0]
print(rb.my_attr)            # Will print 5.0
print(rb_retrieved.my_attr)  # Will throw AttributeError exception! Because rb_retrieved is
                             # not the same Python object as rb
```

The reason for this is that: *Proxy class instances, or Python objects of proxy classes, are not exact representations of C++ objects and multiple Python proxy objects may reference the same C++ object.*

37.4.3 Function/method overload ambiguation

In Python it is forbidden to declare anything using the same identifiers or names in any scope where they already exist. In C++ however, it is commonplace to overload functions and methods by their signatures while sharing the same names. In AGX Python, proxy classes only have one representation for all possible versions of static functions or methods found in C++.

Ambiguation resolving is done within the Python extensions before a match is found based on which arguments the Python code passed to it during a call, where no match throws a NotImplementedError exception back to the caller.

37.4.4 Inherited classes/virtual methods

Some AGX classes allow for their virtual methods to be overridden in derived classes, where EventListener family of classes being the most relevant example of this. The table below show examples of classes which can be implemented in Python:

Python Class	Virtual Methods (in Python)

agxSDK.EventListener	`def addNotification(self) -> "void"`
	
	`def removeNotification(self) -> "void"`

agxSDK.StepEventListener	`def preCollide(`
	` self,`
	` t: "float"`
	`) -> "void"`
	
	`def pre(`
	` self,`
	` t: "float"`
	`) -> "void"`
	
	`def post(`
	` self,`
	` t: "float"`
	`) -> "void"`
	
	`def last(`
	` self,`
	` t: "float"`
	`) -> "void"`

agxSDK.ContactEventListener	`def impact(`
	` self,`
	` t: "float",`
	` gc: "agxCollide.GeometryContact"`
	`) -> "bool"`
	
	`def contact(`
	` self,`

	t: “double”, gc: “agxCollide.GeometryContact”) -> “bool” def separation(self, t: “double”, gp: “agxCollide.GeometryPair”) -> “void”
agxSDK.GuiEventListener	def mouseDragged(self, buttonMask: “MouseButtonMask”, x: “float”, y: “float”) -> "bool" def mouseMoved(self, x: “float”, y: “float”) -> "bool" def mouse(self, buttonMask: “MouseButtonMask”, state: “MouseState”, x: “float”, y: “float”) -> "bool" def keyboard(self, key: “int”, modKeyMask: “int”, x: “float”, y: “float”, keyDown: “bool”) -> "bool" def update(self, x: “float”, y: “float”) -> "void"
agxSDK::ContactEventListener	def impact(self, time: “double”, geometryContact: “agxCollide::GeometryContact”) -> “KeepContactPolicy”
	def contact(self, time: “double”, geometryContact: “agxCollide::GeometryContact”) -> “KeepContactPolicy”
	def separation(self, time: “double”, geometryPair: “agxCollide::GeometryPair”) -> “void”
agxSensor::JoystickListener	def axisUpdate(self, state: “agxSensor::JoystickState”, axis: “int”,) -> “bool”
	def buttonChanged(self, state: “agxSensor::JoystickState”, button: “int”, down: “bool”

```
) -> "void"
```

```
def povMoved(
 self,
 state: "agxSensor::JoystickState",
 pov: "int"
) -> "void"
```

```
def sliderMoved(
 self,
 state: "agxSensor::JoystickState",
 slider: "int"
) -> "void"
```

```
def addModification(
 self
) -> "void"
```

```
def removeModification(
 self
) -> "void"
```

It's important to not forget about the 'self' argument when you override any of these methods, and that the right type of any value they return is used. Violations to this are considered an error and will break your script.

Furthermore, it's important to follow Python rules as well, something known as "Zen of Python", and call the base class version of the method being overridden when you wish to return some default value. In Python, you do not wish to "override" base class functionality with polymorphic inheritance as you do in C++, instead you "enhance" with new functionality by adding instead of replacing.

37.4.5 Scope locality and garbage collection

In Python, all declarations made in some scope are also bound to that scope once evaluated. In other words, the default scope used in Python is "local" where for example Lua use global scope by default. Assigning or returning a value binds them to whatever object or scope that accepts the value, and the value will persist in memory for as long it is strongly referenced by another object or scope. If no such references exist for a value it gets garbage collected. Whether this happens right away or some time later can vary, but it is safe to assume it mustn't be touched by any code once it becomes garbage.

Reference-counted C++ objects complicate things sometimes if care is not given to avoid premature decrement reference counts of references not owned by the dereferencer. Always keep a reference around for AGX objects you create by keeping a Python object/proxy class instance alive for as long you want to be on the safe side.

37.4.6 Ref smart pointer objects

Sometimes it's not always a raw pointer object you deal with but their smart pointer counterparts. The smart pointer template class instances are suffixed with Ref, so for example smart pointer class for the Geometry class is called GeometryRef and for RigidBody, it's RigidBodyRef. Normally they expose the same methods as the original class with the addition of those declared by agx::ref_ptr<T> template class, such as get(), isValid and so on.

It is not, however, possible to pass them as arguments to methods expecting objects of template argument type; e.g. if a Geometry is expected, passing GeometryRef instead is not valid. Sometimes you deal with <Class>RefVectors, and any accessors will inevitably always return <Class>Ref objects - not <Class> objects.

To dereference these smart pointers, use the get() method on them, but check if they are valid before you do, or else you risk the execution of dereferencing null pointers within the C++ layer which will crash your application. Here's an example where we define a function which extracts all the geometries associated with a rigid body and returns a Python list with raw Geometry objects:

```
def rigidBodyGeometriesToList(rb):
 geometries = rb.getGeometries() # geometries is of type GeometryRefVector
 geometries_list = list()    # the list to return
 for geo_ref in geometries:   # geometries contain items of type GeometryRef
   geometries_list.append(geo_ref.get()) # geo_ref.get() gives us its Geometry
 return geometries_list    # geometries_list now contain Geometry items
```

37.5 C++ to Python guide

Below is a number of sample cases demonstrating the syntax difference between Python and C++:

37.5.1 Construction

C++:

```
agx::RigidBodyRef body = new agx::RigidBody();
```

Python:

```
body = agx.RigidBody()
```

37.5.2 Calling a normal method

C++:

```
body->getMassProperties()->setMass( 10 );
```

Python:

```
body.getMassProperties().setMass( 10 )
```

37.5.3 Calling a static method:

C++:

```
agx::Constraint::calculateFramesFromBody(agx::Vec3(1,2,3), agx::Vec3(1,0,0),
                                         body1, frame1,
                                         body2, frame2);
```

Python:

```
agx.Constraint.calculateFramesFromBody(agx.Vec3(1,2,3), agx.Vec3(1,0,0),
                                       body1, frame1,
                                       body2, frame2);
```

38 Lua scripting

Lua (www.lua.org) is a lightweight scripting language. The core Lua library is very small which makes it a good candidate for integration into other toolkits.

Most of the AGX programming API is exported into *luaplugins*, dynamically loadable modules which wrap the C++ classes into Lua.

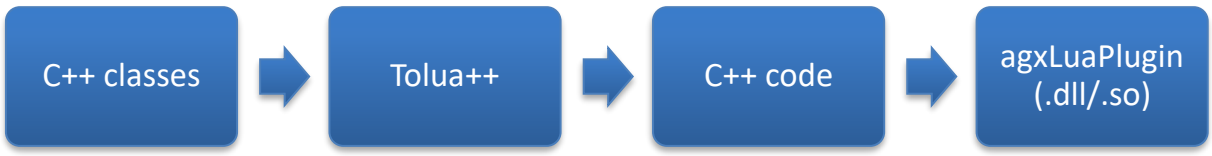


Figure 45: Plugin generation process

The process of creating plug-in is pictured in Figure 46. The header files containing the C++ classes are automatically parsed into Lua/C++ integration code.

A script plug-in is responsible for registering tables, classes, namespaces, enums and methods so that they become available from Lua.

Plugin name	Filename .dll/.so/.dylib	Namespace(s)	Description
agx	agx_luaplugin	agx, agxIO, agxCollide, agxCFG	Classes from the core AGX API including: agx::RigidBody, agxCollide::Geometry etc.
agxOSG	agxOSG_luaplugin	agxOSG	Classes from the agxOSG namespace/library for building sample applications with graphics.
agxModel	agxModel_luaplugin	agxModel	Classes from the agxModel namespace.

Table 25: Available plug-ins

38.1 ScriptManager

The singleton class **agxLua::ScriptManager** is responsible for the whole management of plug-ins and loading of scripts. There is only one global lua_State which is handled by the ScriptManager. The initialization and de-initialization of the ScriptManager is handled automatically as all other singletons in AGX. A new lua_State can be initialized and given to the script manager through the call:

```
lua_State *L = lua_open();
agxLua::ScriptManager::instance()->init( L );
```

As soon as a new lua_State is used, all information about loaded plug-ins, registered classes etc is lost.

38.2 Writing scripts

This section describes the details around writing Lua scripts for the plugin-system available in AGX. More detailed documentation of how to use the Lua scripting language can be found at (www.lua.org). A list of tutorials is described later (38.4 Tutorials). Scripts should be given the suffix .agxLua to be recognized by for example the *agxViewer* application.

38.2.1 Loading plugins

A plug-in can be requested from a Lua script using the function:

```
success = requestPlugin(<plug-in name>)
```

The plug-ins are located using the agxIO::Environment singleton, described in chapter 38. By default, the following paths are used:

For Win32 platform:

- Current directory (./ScriptPlugins)
- ./plugins/ScriptPlugins
- Any path specified in PATH environment variable (with or without /ScriptPlugins appended to the path)
- Any path specified in AGX_LUAPLUGIN_PATH environment variable

For Linux:

- Current directory (.ScriptPlugins)
- ./plugins/ScriptPlugins
- Any path specified in LIBRARY environment variable (with or without /ScriptPlugin appended to the path)
- Any path specified in AGX_LUAPLUGIN_PATH environment variable

Plug-ins can be dependent of each other, so when loading for example the agxOSG plug-in, the agx plug-in will automatically be loaded ().

Plugin name	Dependent on
agx	-
agxOSG	agx
agxModel	agx

Table 26: Plug-in dependencies.

When a plug-in is registered, the namespaces, classes, enums and methods exported by that plug-in are available for use in a Lua script.

38.2.2 Memory allocation/deallocation

Lua is an interpreting language with garbage collection. This means that memory allocated for tables and other variables which are under the responsibility of Lua will be de-allocated at any time after the variable has gone out of scope.

The mix between using reference counted pointers for memory handled by AGX and the garbage collection scheme of Lua makes it very important to follow a strict schema

for how memory is allocated (variables are created) and de-allocated. Otherwise memory corruption or memory leaks can occur.

38.2.2.1 Explicit allocation schema

For any class with constructors exported to Lua, there are two methods: **:new()** and **:new_local()**. They operate as an explicit constructor, methods for creating a new instance of a class. The difference between these two is the following:

		Example

:new_local	Allocate memory on the heap. If it is a class derived from agx::Referenced, it will increment the reference count. When scope of variable goes out, it will decrement reference count.	`do`
		`  local v1 =  agx.Vec3:new_local()`
		`end`
		`-- Scope of v1 goes out, v1 is`
		`-- deallocated`
		
		`do`
		`  local b =  agx.RigidBody:new_local()`
		`end`
		`-- Scope of b goes out, v1 is`
		`-- deallocated`

:new	Allocate memory on the heap, responsibility of deallocating memory is left to C++.	`do`
		`  local b =  agx.RigidBody:new()`
		
		`  -- Simulation is now responsible`
		`  -- for deallocating`
		`  -- (when refcount goes to 0)`
		`  simulation:add( b )`
		`end`

default	Same as **new_local**	`do`
		`  -- Allocate a Vec3, deallocated`
		`  -- when function call is done`
		`  body:setVelocity(agx.Vec3())`
		`end`

Any class that is created on the *stack* in C++, such as: agx::Vec2, agx::Vec3, agx::Vec4, agx::Quat, agx::EulerAngles, agx::AffineMatrix4x4, agx::OrthoMatrix3x3 *should always* be created using the *new_local()* constructor or the default constructor (*implicit allocation schema*, explained later). This goes for any object that is handled as a reference or non-reference to a method in AGX.

> Stack allocated classes which is not derived from agx::Referenced, should be allocated with new_local(), or with the default constructor.

An example:

```
function foo()
  local velocity = agx.Vec3:new_local(1,2,3) -- Allocate a Vec3
end
```

When the scope of the function *foo*() ends, the variable *velocity* is scheduled for garbage collection in Lua, meaning that at any time, the destructor for an agx::Vec3 will be called.

There is one way of creating a memory leak in lua, which is by creating a class that does not inherit from agx::Referenced using **:new()**.

An example illustrates this:

```
function foo_leak1()
  local velocity = agx.Vec3:new(1,2,3) -- Allocate a Vec3 with :new()
end
-- Scope for velocity goes out, but Lua is NOT responsible for deallocating
-- the memory, and its not a reference object, so it will never be
-- deallocated.
```

38.2.2.2 Implicit allocation schema

A class can also be instantiated without a call to new/new_local. Lua is a flexible language. The code example below shows the alternatives for instancing a class. Omitting new/new_local is the same thing as calling :new_local().

```
body = agx.RigidBody("name") -- Equal to RigidBody:new_local("name")
body = agx.RigidBody'name'   -- Equal to RigidBody:new_local("name")

simulation:add( body ) -- Now body will be deallocated by AGX later on.
```

38.2.3 Lua and agx::Referenced derived classes

The classes derived from agx::Referenced is properly handled in Lua:

When <class>:new_local() is called, the refcount is incremented.
When the scope of a variable goes out, the refcount will be decremented.
This means that it is perfectly possible to do the following:

```
local sim = agxSDK.Simulation() -- == new_local()!
do
  local b = agx.RigidBody:new_local() -- refcount is now 1
  sim:add(b) -- refcount for b is now 2
end
-- scope for b goes out, refcount is decremented to 1
sim = nil -- refcount for sim goes to 0, which means that refcount for b
          -- also goes to 0, and it is deallocated.
```

38.2.4 C++ to Lua guide

Below is a number of sample cases demonstrating the syntax difference between Lua and C++:

38.2.4.1 Construction:

```
C++: agx::RigidBodyRef body = new agx::RigidBody();

Lua: body = agx.RigidBody()
Lua: body = agx.RigidBody:new()
```

38.2.4.2 Calling a normal method

```
C++: body->getMassProperties()->setMass( 10 );

Lua: body:getMassProperties():setMass( 10 )
```

38.2.4.3 Calling a static method:

```
C++: agxOSG::SimulationObject::createBox("box", agx::AffineMatrix4x4(),
                                                  agx::Vec3(1,1,1), simulation);

Lua: agxOSG.SimulationObject.createBox("box",
                                                 agx.AffineMatrix4x4(),
                                                 agx.Vec3(1,1,1),
                                                 simulation)
```

38.2.4.4 Using an enum:

```
C++: body->setMotionControl( agx::RigidBody::STATIC );

Lua: body:setMotionControl( agx. RigidBody.STATIC)
```

38.2.4.5 Object oriented access of a class pointer to it self

```
C++: this->setPosition( agx::Vec3() );

Lua: self:setPosition( agx.Vec3() )
```

From the above examples it is clear that namespaces are accessed using the '.' letter in Lua ('::' in C++). Methods (except constructors) are accessed using the ':' letter in Lua, no matter if it is a pointer or a reference which is used ('->' or '.' in C++).

The *this* pointer in C++ is called *self* in Lua.

38.2.5 Inherited classes/virtual methods

There are a number of classes in AGX (see *Table 1*) that can be derived and implemented by a user for various purposes; examples of these classes are: StepEventListener, GuiEventListener, ContactEventListener etc. These can also be implemented in Lua. This means that Lua code can be called from C++. A very important thing to notice is that a class that has virtual methods that will be overloaded, is renamed to: **Lua<className>**. This is because there is a derived class which has the virtual methods implemented in C++ and then exported to Lua.

C++ class name	Lua class name
agxSDK::StepEventListener	agxSDK.LuaStepEventListener
agxSDK::GuiEventListener	agxSDK.LuaGuiEventListener
agxSDK::ContactEventListener	agxSDK.LuaContactEventListener

Table 27: Classes with virtual overloaded methods in Lua.

So to get callbacks, you need to use the Lua<className> naming convention. See in the below example:

38.2.5.1 Example1: Creating a StepEventListener

```
function buildScene(simulation, application)
  local root = agxOSG.Group()

  -- Observe that we use the name LuaStepEventListener
  listener = agxSDK.LuaStepEventListener()
  listener:setName("MyListener")
```

```
  -- Which events are we listening to?
  local mask = agxSDK.StepEventListener.ActivationMask.PRE_STEP +
     agxSDK.StepEventListener.ActivationMask.POST_STEP

  listener:setMask(mask)
  listener.stopTime = 10

  function listener:pre( time )
    print("We are now at time "..time)
    if (time > self.stopTime) then
      self:getSimulation():remove( self )
    end
  end

  -- Register listener at simulation
  simulation:add ( listener )
  return root
end
```

The above script when executed, will print:

```
We are now at time 0.016
We are now at time 0.032
…
We are now at time 10.0
```

And then it will stop printing, because the step event listener has removed itself from the simulation, and hence stopped executing.

One very important thing to note here is that when a derived classes goes out of scope in lua, it will lose all its attributes defined in the lua file (after garbage collection), even if the C++ related memory will be kept by e.g. an agxSDK::Simulation which the object has been added to. So instead of

```
  local listener = agxSDK.LuaStepEventListener()
  sim:add(listener)
```

one should use

```
  listener = agxSDK.LuaStepEventListener()
  sim:add(listener)
```

and then be careful not to assign a new value to the same (global) variable later in the script.

38.3 Running scripts

38.3.1 agxViewer

agxViewer is an application based on the agxOSG::ExampleApplication framework which enables a debug and an “ordinary” 3D viewer of simulations (as described in chapter 35.5). The agxViewer loads one or more Lua scripts from the command-line. Each script will be bound to keys between 1…0 and F1…F11. So the first script will be executed using key ‘1’ etc.

The scripts loaded with the agxViewer must obey the following rule: They must have a function named **buildScene** as follows:

```
function buildScene(simulation, application)
  local root = agxOSG.Group:new()

  return root;
end
```

application, **and the** *returning of a agxOSG.Group* **is optional and can be left out if they are not needed.**

simulation **is a pointer to the current agxSDK::Simulation,** *application* **is a pointer to the current agxOSG::ExampleApplication.**

If buildScene() returns a value, it must be a valid reference to an object of type agxOSG.Group. After that it is all up to the function buildScene() to create whatever scene is needed. Whenever a new (or the same) scene is created again, we can be sure that all data from a previous scene is cleared.

A simple example of a scene can look like:

```
function buildScene(simulation, application)
  local root = agxOSG.Group:new()

  local radius = 1.0
  bool createGeometry = true
  bool createBody = true
  local m = agx.AffineMatrix4x4:new_local()
  -- Create a sphere at 3 meters above origin
  m:setTranslate( 0,0,3 )
  local obj = agxOSG.SimulationObject:createSphere("sphere",
      m, radius, root, simulation,
      agx.RigidBody.MotionControl.DYNAMICS, createGeometry, createBody )

  return root
end
```

Another alternative to assign graphics to physical objects is to use the function agxOSG.createVisual(). It accepts geometries and bodies and assemblies as an argument, creating visual representations for each geometry found:

```
function buildScene(simulation, application)
  local root = agxOSG.Group:new()

  local radius = 1.0
  local m = agx.AffineMatrix4x4()
  -- Create a sphere at 3 meters above origin
  m:setTranslate( 0,0,3 )
  local geom = agxCollide.Geometry(agxCollide.Sphere(radius))
  local body = agx.RigidBody()
  body:add(geom)
  agxOSG.createVisual( geom, root ) -- Create visual representation of the
                                    -- geometry/body

  return root
end
```

38.3.2 luaagx

luaagx is an adopted version of the ordinary lua interpretator (l*ua***).**

```
print("testing luaagx")
requestPlugin("agx") -- load the agx plugin

body = agx.RigidBody()
```

```
-- Aha, allocate a simulation with local
local sim = agxSDK.Simulation:new_local()

sim:add(body)

-- Open a datafile for writing
local out = assert(io.open("data.txt", "w"))

-- Loop 100 steps
for i=1,100 do
  sim:stepForward()
  local t = sim:getTimeStamp()
  local p = body:getPosition().z()
  out:write(string.format("%d %d\n",t,p))
end

out:close()
```

```
cmd> luaagx test.agxLua
```

It is possible to determine if a script is started via luaagx or agxViewer by the following code:

```
if (arg) then
  print("Started from luaagx"
else
  print("Started from something else than luaagx")
end
```

38.4 Tutorials

In the directory <agx-dir>/data/luaDemos/tutorials there is a list of tutorials explaining how to create lua script, build scenes and connect various mechanics to a simulation, such as EventListeners etc.

File	What this scene will teach you:
tutorial0.agxLua	• Demonstrates how to create a basic simulation with bodies, geometries and shapes. • Can be started with luaagx or agxViewer. However, no graphics is generated in this script, so when you start it with agxViewer, you need to press 'g' to see the geometry.
tutorial1.agxLua	• Create a simulation including graphics which is loadable with agxViewer • Load a graphical object from a file, and connect it to a geometry so that it moves with the geometry. • This file must be loaded with *agxViewer*, NOT luaagx

tutorial2.agxLua	• Create a lua script which can be loaded *both* with luaagx and agxViewer.
	• When using luaagx:
	  ○ How to use arguments
	  ○ How to connect several scenes to keys.

tutorial3.agxLua	• Create a LuaStepEventListener to listen for events in the simulation loop. • Access contact data/forces and write them to the screen. • Get all constraints in a system into a vector

	• Access constraint data and read forces. • Draw simple text onto the screen.
tutorial4.agxLua	• Spawn and remove objects during runtime. • Use AutoSleep • Create a LuaStepEventListener to listen for events in the simulation loop and perform tasks.
tutorial5.agxLua	• Read simulation data from .agxScene files • Group and move objects using agxSDK::Assembly. • Get objects from an Assembly. • Find and access objects from Simulation using names.
tutorial6.agxLua	• Use an agxSDK.LuaGuiEventListener for listening to keyboard and mouse events. • Change a shape in size (radius of a sphere), including the visual representation. • Use the script StatisticsWriter.agxLua to write statistic information to a file on disk.
tutorial7.agxLua	• Create a body, make it KINEMATIC and move it using a Spline in a StepEventListener • Make a Geometry a *sensor*. • Create a LuaContactEventListener which listens to contacts • Assign property values to objects. • Use ExecutionFilter for filtering out which contacts a ContactEventListener will see.
tutorial8.agxLua	• Create a hinged body onto bodies are dropped. • MergeSplit is used for merging the dropped bodies with the rotating box. • AutoSleep is enabled so that boxes that falls onto the static ground are disabled.
tutorial9.agxLua	• Demonstrates how to build a beam with “lumped” elements. Boxes are attached to each other with Lock constraints. • Demonstrates how geometries can be disabled against each other. • Demonstrates how compliance for constraints can be set.
tutorial_wire1_forces.agxLua	• Demonstrates how to route a wire with a winch and bodies. • How to read forces in contacts (frictional) and reading the forces from a Winch. • There are several weighs lying on an inclined plane. Each weight has a different friction. This means that the less friction, the more force in the winch (to hold up the weight), the total sum should stabilize to an equal value for all of the winches.
tutorial_wire2_cut_merge.agxLua	• Shows how to cut and merge a wire.
Tutorial_wire3_winch_pulley.agxLua	• Demonstrates how a simple pulley can be built using EyeNodes. • The force in the winch (red) should be ¼ of the force in the wire holding up the weight.
tutorial_multi_wire1_drum.agxLua	• Demonstrates how to setup and use the Drum class.

tutorial_wire4_render_iterators.agxLua	• Demonstrates how to use the render iterators for a wire to do custom rendering.

Table 28: Available tutorials for Lua.

39 ExampleApplication and command-line arguments

The tutorials and also the application **agxViewer** are all based on the class agxOSG::ExampleApplication. It is a class that is responsible for parsing command line arguments, reading files, creating a 3D window, stepping simulation and rendering the simulation. The 3D rendering is based on OpenSceneGraph.

39.1 agxViewer

agxViewer is an application for reading various files (.agx, .aagx, .agxScene, …). It is the default program for opening: .agx, .aagx, .agxLua, .agxPy, .agxLuaz and agxPyz … files.

If you want to use agxViewer (and any other example/tutorial) from the command line/terminal, make sure you execute the **setup_env.bat** file in the installed agx directory from the cmd/terminal-instance. Otherwise it will not be able to find all the required dll-files and configuration files.

Help can be obtained with the argument:

cmd> **agxViewer --help**

For a list of available arguments, see Table 30: Arguments to agxViewer.

Sample use of agxViewer:

```
agxViewer --sceneFile template_scene1.agxScene --startPaused --stopAfter 3.1 --capture 1 --
captureFPS 30 --saveScene store.agx --saveAfter 3
```

The above example will:

- Load the scene file template_scene1.agxScene
- Start paused (press 'e' to continue)
- Exit the application after 3.1 seconds of simulation time
- Enable the capture of each frame as a .bmp file (**agx__0001.bmp…**)
- Capture 30 images/second (in simulation time to achieve real-time)
- 3.0 seconds into the simulation, the scene will be stored to disk with the name **'store.agx'**

For a list of available key bindings of the agxViewer application, see Table 29: Keyboard bindings.

39.1.1 ImageCapture

In the example for agxViewer above there as example of logging images to file. The class used for this is agxOSG::ImageCapture.

You can obtain ExampleApplication's instance of ImageCapture
by executing (assuming you have an ExampleApplication called application):

```
agxOSG::ImageCapture* myCapture = application->getImageCapture();
```

ImageCapture has methods for

- changing the file format (`setPostfix`)
- changing the image logging frequency(`setFps`)
- turning logging on or off
- changing the image destination(`setDirectoryPath`)

The last one is important if the executable (agxViewer, your tutorial, …) is stored in a folder where the default user lacks writing access – in this case, logging the images would fail and a different location for the images should be given.

39.1.2 Key bindings

Key	Description
Numeric keys, function keys	Select scene
a	not used (reserved for use in specific scenes)
b	Toggle statistics reporting to screen (debug rendering)
c	contact reduction (trimesh) circles between binning3, off, binning2
d	contact reduction circles between off, binning2, binning3
e	Toggle simulation run/pause
f	Toggle full screen
g	Toggle debug rendering
h	move camera to the left
i + left mouse	information about selected body
j	move camera backwards
k	move camera to the right
l	move camera downwards
m + left mouse	camera follows selected body
m	reserved for memory debugging
n	Toggle statistics reporting to screen (debug rendering) and console
o	move camera upwards
p	Toggle merge/split functionality
q	Toggle real time synchronization for image capture.
r	In paused simulation, take a single time step
s	Toggle OSG statistics
t	Toggle synchronization of image capture with wall/simulation time
u	Move camera upwards
v	gravity points towards screen down
w	circle between osg shading modes (polygon, wire frame, points)
x + left mouse	remove selected body
left alt + x	move all non-static rigid bodies to the left
y	Toggle update debug render cache for simulation in pause mode
z	not used (reserved for use in specific scenes)
A	Toggle debug rendering of AABB trees
B	Toggle debug rendering of bounding boxes
C	Print camera data to console

D	Delete all objects from simulation.
E	Circle through solvers
F	Toggle text debug rendering.
G	Toggle OSG rendering
H	Print help screen to console
I	restore simulation from scene.agx
J	Update collision detection
K	Toggle debug coloring mode.
L	Toggle osg lightning
M	reserved for memory debugger
N	Reload scene
O	save simulation to scene.agx
P	run parallel performance test
Q	Toggle wait in calling update (real-time sync)
R	Re-read specified configuration file
S	Toggle osg on screen stats
T	Toggle enable scene decorator
U	dump screenshot for povray
V	gravity points in original direction
W	save simulation to saved_scene.scene
X	Toggle simulation dump
left alt + X	move all non-static rigid bodies to the right
Y	Toggle auto sleep of resting objects
Z	Dump scene to osg-file
+	Select next scene
-	Select previous scene
return	Shoot sphere in camera direction
drag left mouse	rotate camera
mouse wheel up/down	adjust camera movement speed
left ctrl + drag left mouse	add temporary ball joint to selected body
left ctrl + drag mid mouse	add temporary lock joint to selected body
left ctrl + drag right mouse	add temporary lock joint resetting rotation to selected body
left alt + drag left mouse	move selected body/geometry (no dynamics)
left alt + drag right mouse	rotate selected body/geometry (no dynamics)
space	Reset camera to view all objects
esc	Quit

Table 29: Keyboard bindings.

39.1.3 Arguments

Argument	Explanation
--useShadows	Enable shadowing
--shadowMethod <0,1,2>	Select shadow method 0- shadow textures(default), 1 – softShadows
--window width height	Specifies the size of the rendering window.
--clearColor r g b	Specifies the background color
--renderOsg <1/0>	Specify whether rendering using OSG primitives is enabled or not. (default 1)
--renderDebug <1/0>	Specify whether rendering using agx Debug renderer is enabled or not. (default 0)
--numPostImpactIterations <1..n>	- Number of iterations during impact stage 2 (can be less than for impact stage 1)
--numImpactIterations <1..n>	Number of iterations during impact stage 1
--numPostImpactIterations <1..n>	Number of iterations during impact stage 2 (can be less than for impact stage 1)
--numIterations <1..n>	Number of iterations during normal solve (i.e., no impacts)
--numIterationsForIterateLast <1..n>	Number of iterations performed on special constraints
--numLCPIterations <1..n>	Max number of iterations for LCP solver
--partitioner <1/0>	Enable/disable use of IslandPartitioner. (Default 1)
--capture <1/0>	Enable/disable screen capturing to disk (agx__0001.bmp)
--captureFPS <float>	Specify the desired capture rate for image capture
--numThreads <n>	Specify number of threads available for AGX
-- numThreadsSpace	Specify number of threads available for Space
--save <filename>	Create scene and write to specified filename
--startPaused	Start the simulation in paused mode.
--read <filename>	Start simulation with content in specified filename
--synchronize <1/0>	Synchronize image capture with Simulation time.
--wait <1/0>	Sync stepping simulation to clock time (1) or run as fast as possible (0) (Default 1).
--dumpEnable	Enable serialization of simulation content to disk.
--dumpInterval <X seconds>	Specifies the frequency of which the simulation is dumped to disk. (Default 0.0333).
--translate x y z	Translate the debug rendering of the scene
--gravity x y z	Specify the gravity vector, including magnitude
--saveScene <path>	When a store of the scene is triggered (either by 'O' key, or saveAfter) this is the filename that will used. The path will also be used when dumping files to disk (--dumpEnable).
--saveAfter <X seconds>	Step forward system until we read X seconds in simulation time then the content of the scene will be stored to disk.

--timeStep <real>	**Specify the timestep (dt) used for stepping the simulation.**
--groundPlane <heightOverZero>	Create a plane geometry with default material at specified height in Z.
--cfg <cfg-file>	Read in a configuration file
--stopAfter <X seconds>	Step forward system until we reach X seconds in simulation time, then exit.
--agxOnly	Only step AGX, no graphics. When combined with --stopAfter, will step through simulation as fast as possible and print out the complete wall time after finishing.
--attachScript <filename>	After a scene is created, execute buildFunction in the specified script. Useful for adding for example event listeners into existing scenes.
--scene <n>	Go to scene number n (in simulations with several scenes).
--help	Print a list of all command line arguments to cout (the console).
--version/-v	Print the current version of AGX.

Table 30: Arguments to agxViewer.

In order to get an overview over all command line arguments, use the --help switch:
agxViewer –help

39.2 agxArchive

agxArchive is an utility that can be used to convert between .agx and .aagx file, investigate a serialization file and creating a script archive.

39.2.1 Arguments

Argument	Explanation
--help/-h	Show help
--pack/p	Create a compressed script archive
--unpack/-u	Decompress a script archive to disk
--generate/-g	Generate a random UUID string

Table 31: Arguments to agxArchive.

39.3 Usage

39.3.1 Investigate a serialization

The general use of agxArchive is to investigate a serialized file:

```
> agxArchive submarine.agx

Archive information
-------------------
Header               Algoryx:AgX Library 64Bit
AGX version          : 2.12.0.0-23170
Serialization version: 37
BuildDate            : Mar 30 2015
```

```
BuildTime            : 17:54:35
File format          : binary
File size            : 216601 bytes
Serialization date   :
Build flags
     USE_PARTICLE_SYSTEM=1
     USE_OSG=1
     USE_OPENCL=0
     SABRE_USE_SSE3=1
     UNITTEST_ENABLED=0
     SABRE_USE_PADDING=1
     USE_SSE=1
     USE_64BIT_ARCHITECTURE=1
     USE_REAL_FLOAT=0
     USE_LUA=1
```

39.3.2 Convert files

By supplying two files (.agx/.aagx) one can convert between these two file formats (ascii/binary) for serialization.

```
>agxArchive submarine.agx output.aagx
Reading input file: submarine.agx
Writing output file: output.aagx
```

39.3.3 Pack a script archive

If you have a Python or Lua script file together with some data, you can use agxArchive to pack these files into a compressed archive. Directories will recursively be compressed into the archive.

Usage: agxArchive <archive-name> files and directories

```
> agxarchive --pack submarine.agxLuaz submarine.agxLua submarine.agx
submarine.agxLua
submarine.agx

Successfully archived 2 files to archive 'submarine.agxLuaz'
```

39.3.4 Unpack a script archive

Usage: agxArchive <archive-name> <target directory>

```
> agxArchive --unpack submarine.agxLuaz myScripts
Successfully decompressed archive 'submarine.agxLuaz'' into directory 'myScripts'
```

40 C++ Tutorials

All source code (C++) tutorials can be found in the *tutorial* subdirectory. The tutorials are usually associated to a namespace and are to be found in the various subdirectories agx, agxCollide, agxOSG, agxSDK,...

A new tutorial is added by creating a source code file named tutorial_X.cpp/(tutorial_X.h) which will automatically be picked up by the build system and added to the target build.

Common to all tutorials in the agxOSG directory is the **ExampleApplication** class which encapsulates functionality for capturing keyboard events, creating example scenes etc.

These tutorials are all built around different “scenes”. The current scene can be selected at run-time with keyboard presses (0...9, F1-F10) or via a command-line argument: --scene [1...]

41 Environment variables

Values within [] is the default value.

Environment variable	Description
AGX_LOG_OVERWRITE = [0]/1	Specify whether the log file will be overwritten, or if new files should be generated as AGX_LOG_FILE.0001 .0002, ...
AGX_LOG_ENABLE=1/[0]	Specify whether the messages will be logged to a file or not.
AGX_LOG_LEVEL =0/[1]/2/3	Specify the level for which messages will be logged to file.
AGX_LOG_FILE= agx.log	Specify the path to the log file that should be generated.
AGX_OUTPUT_LEVEL=0/[1]/2/3	Specify the level for which messages will be logged to the console.
AGX_THROW_ON_ERROR=[1]/0	Should an exception be thrown when an error message is written?
AGX_BREAK_ON_ERROR=[0]/1	Should a breakpoint be set when an error message is written? This only works in DEBUG build in VisualStudio.

Table 32: Environment variables controlling logging behavior.

Environment variable	Description
AGX_FILE_PATH	Specifying where resource files will be found (models, images, scripts…)
AGX_PLUGIN_PATH	Specifying where plugins can be found.
AGX_PROCESS_PRIORITY=[2]/0..4	0 is lowest, 4 is real time priority. 2 is default priority. Works in Win32 only
AGX_PROCESS_AFFINITY=[0]/1	Specify whether the main thread for agxSDK::Simulation will be restricted to run on the current executional unit, or if it is allowed to be scheduled by the OS. Works in Win32 only
AGX_STATISTICS_ENABLE=[0]/1	Enable writing statistics information to a file specified with AGX_STATISTICS_PATH
AGX_STATISTICS_PATH=<filepath>	Specifies the file to which the statistics information will be written if enabled.
AGX_STATISTICS_INTERVAL=<float>	Specifies how often statistics information will be written to file in seconds. 1==write once every second.

Table 33: Other environment variables.

AGX contains a mechanism for logging messages to file and/or console which is implemented through two classes, **agx::Logger** and **agx::Notify**. *Notify* is a class derived from std::ostringstream so it can be treated as any std::ostream using the overloaded << operator. The logging functionality is accessed through a number of macros defined in <agx/Logger.h>. There are four different levels of messages:

Value	Meaning	Description
0	Debug	Use for low level debug/test messages, seldom displayed
1	Info	Use for information of loading files, finding data, etc, shown by default
2	Warning	For warnings that occur.
3	Error	For fatal errors. Error can also throw std::runtime_exception() if it is enabled.

Table 34: Values for Notify level

Macro name	Description
LOGGER_DEBUG()	Start new debug message.
LOGGER_INFO()	Start a new information message.
LOGGER_WARNING()	Start a new warning message.
LOGGER_ERROR()	Start a new error message.
LOGGER_END()	Commit the started message (if the message is an error and an exception should be thrown, the actual error is thrown *after* this call).
LOGGER_ENDL()	Same as LOGGER_END() but add a std::endl to the stream.
LOGGER_STATE()	Change the output state for this message. Determines what is written as a message, line number, current function etc.

Table 35: Available macros for logging messages.

Examples of using the LOGGER macros:

```
LOGGER_WARNING() << "The file: " << filename << " cannot be read" <<
LOGGER_ENDL();

LOGGER_INFO() << LOGGER_STATE(agx::Notify::PRINT_NONE) <<
  "This is a plain message with no extra information " << LOGGER_ENDL();

LOGGER_ERROR() << "A serious problem has occurred" << LOGGER_END();
```

The last example with LOGGER_ERROR() will throw an exception if LOGGER().setExceptionOnError(true); is called prior to the call.

42 Matlab/Simulink plugin (optional)

AGX has a plugin to Matlab and Simulink, which can be used in order to run AGX as a co-simulation. This is done via a predefined AGX-block which appears in the Simulink library after installation of AGX, and which can easily be modified to create and load your own AGX simulations.

Furthermore, a lower level Matlab coupling is provided.

The Matlab/Simulink plugin is limited to Microsoft Windows operating systems. It has been tested on Windows 7 64bit and Matlab version R2015a.

If your AGX-license does not include a license for the Matlab/Simulink plugin, then please contact Algoryx Simulation for an evaluation license.

Technically, the Simulink coupling is done via an S-function, and the Matlab coupling via a mex-function. Both need to be explicitly compiled as shown below.

42.1 Installation

Both the Simulink S-function and the Matlab mex-function source code must be compiled prior to using the Agx functionality inside Simulink/Matlab.

All the source code can be found in the following directory:

```
<agx-dir>\data\Matlab\src
```

42.1.1 Manual Installation

During the installation process, the following file will be run to register the AGX path to Matlab's search path, and to compile the s-function and the mex file within Matlab:

```
<agx-dir>/data/Matlab/addAgXToMatlabAndSimulink.cmd
```

You can run this file manually (by double-clicking on it).

The install/uninstall scripts will ask for administrative rights to ensure that it can write in the AGX installation path.

In order to remove the AGX path from Matlab's search path and to remove the compiled s-function and mex file, the uninstaller will run

```
<agx-dir>/data/Matlab/removeAgXFromMatlabAndSimulink.cmd
```

Run this file manually (by double-clicking on it).

42.1.2 Compilation of s-function and mex file

When running the addAgXToMatlabAndSimulink.cmd script the Matlab *mex*-command will be called in order to compile the source files to the s-function and the mex-function.

These source files are:

Filename	Purpose
<agx-dir>/data/Matlab/src/agx_sfun.c	Code file for Simulink s-function.

<agx-dir>/data/Matlab/src/agx.c	Code file for Matlab mex-function.

Table 36: Source files for Matlab/Simulink plugin.

The Matlab mex-command is not a compiler in itself, but rather calls a compiler which has been assigned to it. The first time the command is called (which might be now when you install AGX), a compiler has to be chosen from the Matlab prompt by following the instructions there.

A compiler called *lcc* comes with most Matlab versions on windows, and the two files named above compile fine with it.

Another possible, free compiler under Windows is Microsoft Visual C++ 2015, or its free community version.
You might need to restart Matlab in order to find the newly installed compiler.

You can always change the default compiler for Matlab's *mex*-command by running **mex --setup** in the Matlab prompt and following the instructions there.

42.1.3 Manual compilation

The Matlab (agx.c) and Simulink (agx_sfun.c) code can manually be compiled from within Matlab with the following command:

```
➢ cd <AgX-dir>\data\Matlab\src
➢ mex agx.c
➢ mex agx_sfun.c
```

If you need to compile with debug information, use the **mex –g** command.

42.2 Limitation

Within one Matlab session, AGX can be used either only from Simulink using the s-function, or only from Matlab using the mex function. Using the two after each other or next to each other in the same Matlab session is not recommended.

42.3 Starting AGX from Simulink

After a successful installation, an AGX-library is going to appear in your Simulink library browser.

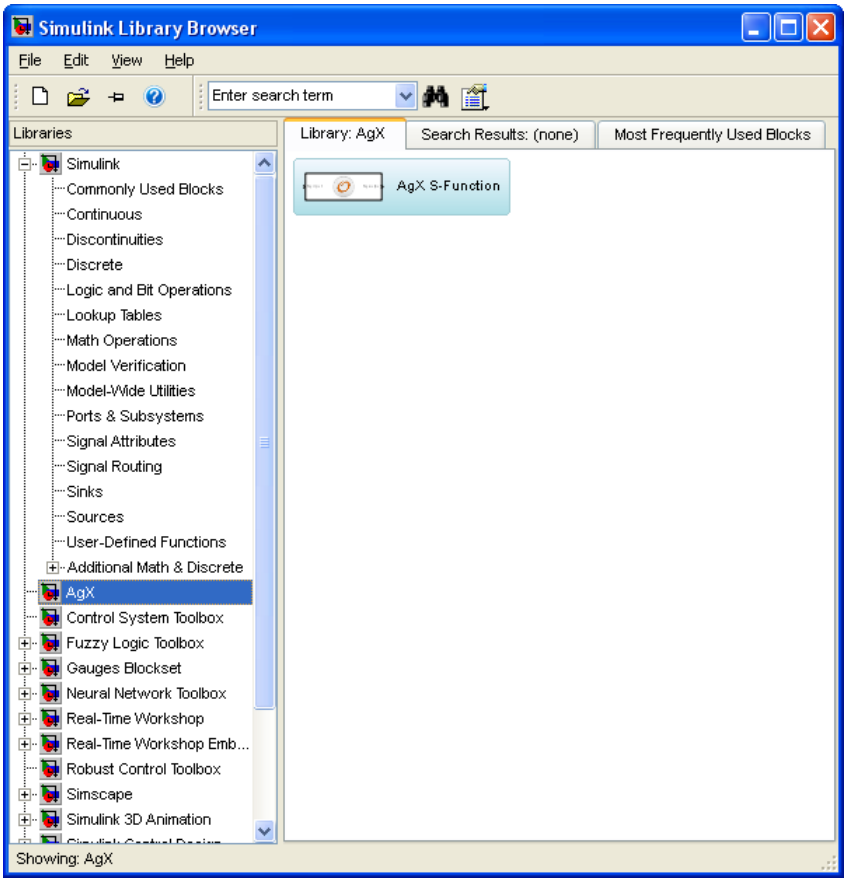


Figure 47: AGX-block in Simulink library view.

Simply drag a block from there into your model and connect its input and output ports. The simulation is ready to run.

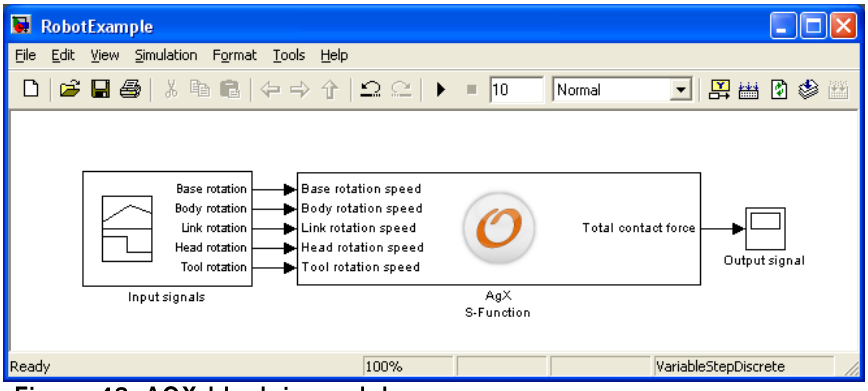


Figure 48: AGX-block in model

Note: There can only be one AGX-block per model. You will get a warning when pressing play in Simulink if you have more than one AGX-block per model.

You can modify the block by double-clicking on it.

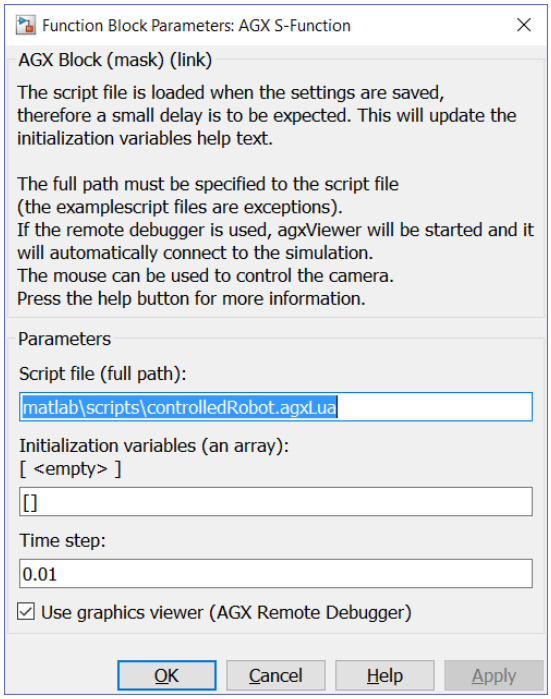


Figure 49: Modifying the AGX block.

The following changes can be made:

- *Script file (full path):*
  Contains the AGX script-file: It can either be an .py/.agxPy (Python) or an .agxLua (Lua) script (a script file with AGX commands) which setup the simulation *and* the coupling between AGX and Simulink. To learn more, look at the sample scripts in <agx-dir>\data\Matlab\scripts.
- *Initialization variables:*
  An array of values to initialize the scene. The length and meaning of the scalars depend on the scene. Descriptions of the parameters are shown in the above the field. If the wrong number of arguments is given, all are set to zero (and a message about that will be sent to the Matlab prompt). Otherwise, after pressing 'Ok', the initialization output will be written to the Matlab variable **agx_init_output**.
- *Time step:*
  The simulation time step. AGX uses an internal fixed-size stepper whose time step you can set here.
- *Use graphics viewer (AGX remote debugger):*
  Enables a 3D viewer which render the simulation scene using agxViewer.exe. Using graphics might slow down the simulation.

The Help-button gives not only useful help, but also links to a few premade examples which you might want to inspect!

After saving the settings in the block (via Ok/Apply), the AGX script-file will be reloaded. This is necessary in order to adapt the number of ports in the block to the ones in the new or modified script-file.

42.4 Starting AGX from Matlab

An AGX simulation can be controlled from Matlab using the

```
<result> = agx(<command>, arguments)
```

Available commands are:

Command	Description
load	Load and initialize a new simulation from a specified AGX script file.
step	Take one simulation step
visual	Enable/Disable the 3D rendering window

The syntax for loading an AGX-simulation in Matlab is:

```
>>  [sizeInitInput, initOutput, sizeStepInput, sizeStepOutput] = agx('load', <simulationFile>,
<initInput>, <timeStep>)
```

where the inputs consist of:

- **<simulationFile>** is the name of a AGX script file containing the simulation data (For example: *<AGX-dir>/data/matlab/scripts/rotatingBox.py*),
- **<initInput>** is a row vector of size sizeInitInput (one might have to call the function once in order to get to know the size from the error message in case it is not previously known) and
- **<timeStep>** is the simulation time step, e.g. 0.01 for 100Hz.

If **<initInput>** does not have the right format, all init input parameters will be set to 0.

The outputs are:

- **<sizeInitInput>** gives the size of the initial input data which is evaluated once at scene startup,
- **<initOutput>** holds the initial output data from scene startup,
- **<sizeStepInput>** gives the size of the input data to subsequent **agx('step' ...)** calls, and
- **<sizeStepOutput>** gives the size of the output data from subsequent **agx('step' ...)** calls.

Example: *data/luademos/matlabDemos/bouncingSphere.agxLua*

This example has a 3D vector as init input, the position of the sphere:

```
agxSDK.SimulationControl:instance():addInitInputArgument(agxMex.LuaInputArgument("SpherePos-
init", "spherePosInit", 3))
```

and a scalar for the restitution:

```
agxSDK.SimulationControl:instance():addInitInputArgument(agxMex.LuaInputArgument("Restitution",
"setRestitution", 1))
```

In total we have 4 scalar values: x,y,z and restitution:

```
>> cd data/luademos/matlabDemos
>> [sizeInitInput, initOutput, sizeStepInput, sizeStepOutput] = agx('load','
bouncingSphere.agxLua', [0,0,0, 0.5], 0.01)
```

The time step in the AGX simulation is set to 0.01 seconds (100Hz).

Running **agx('load' ...)** while there is a running agx simulation called from the same Matlab instance will end this simulation and load the new simulation file instead. In order to modify the simulation, the AGX script file that is loaded by **agx('load')** has to be modified.

For doing a simulation step, run from the Matlab prompt

```
stepOutput = agx('step', <stepInput>)
```

where **stepInput** is a row vector of size 1xsizeIn and represents the input parameters for the simulation step.

stepOutput is a row vector of size 1xsizeOut and represents the output parameters for the simulation step.

In order to enable visual output in a separate 3D window, run from the Matlab prompt

```
agx('visual', 1)
```

This will start the AGX Remote Debugger as a 3D graphics viewer. Using graphics might slow down the simulation.
After having enabled it, it can be disabled with `agx('visual', 0)`.

42.5 Graphics Viewer (AGX Remote Debugger)

The graphics viewer window opens if

- Simulink: the 'Use graphics viewer (AGX Remote Debugger)' checkbox has been activated in the AGX Simulink block.
- Matlab: The command `agx('visual', 1)` has been given in the prompt. Note that you have to reload the scene if you change the visual output setting from Matlab after the scene has been loaded.

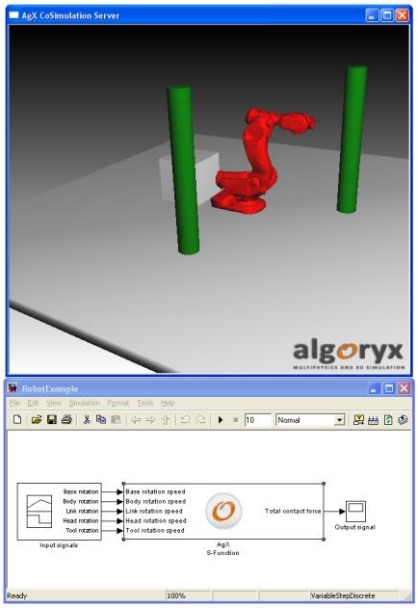


Figure 50: Model view and AGX Remote Debugger view of example scene.

The view can be changed (rotated, zoomed, panned) using the mouse.

For a list of available key bindings of the agxViewer application, see Table 29: Keyboard bindings.

On some systems, you might get warnings from your firewall when using the remote debugger for visual output, since data is sent via a socket on localhost from the plugin to the general AGX application. No internet connection is needed, so refusing the application connection to the internet should not impede the functioning of the program. If your firewall settings do not allow the opening of a socket on localhost and you do not want to/cannot change that, you might have to fall back to using the plugin without the visual output via remote debugger (and thus without graphics).

42.6 Defining input and output signals

The coupling between Matlab/Simulink and AGX Dynamics is done by setting up callback functions. These functions will be called at initialization or every time step for reading input and output data. Below are two examples, one written in Python and one in Lua.

We recommend that you use Python from AGX version 2.18.0.0 as Lua will gradually replaced by Python.

This example scenario consists only of one rigid body, a box, which is rotating along a hinge which is fixed in the world coordinate system. It receives one input per time step: The speed of the hinge motor. The simulation has one output per time step: The summed-up angle of rotations around the hinge axis (this angle can thus be $> 2\pi$ or $< -2\pi$ in case of several full rotations).

The following two subsections will explain how this input and output values are defined in the two different scripting languages.

42.6.1 Python

Here we will use *<AGX-dir>/data/matlab/scripts/rotatingBox.py* as an example.

42.6.1.1 Defining the parameters

When loading the Python script, AGX will first parse it and execute all commandos which are not inside in a function, that is all code in *global scope*.

Inputs and outputs (both for scene initialization and step-function) have to be defined outside of functions (global scope).

The scene itself is defined in the function **buildScene()** in the Python script.
This function will load the simulation scenario.

In the simple case of the example file, the whole scene is setup directly in this Python function. In more complex cases, you might want to call one or several separate Python files to set up the scene, or load in data from other files.

The input/output signals are implemented in classes derived from its respective classes: agxMex.PythonInputArgument for *input-signals* (data from matlab to AGX) and agxMex.PythonOutputArgument for *output-signals* (data from AGX to matlab):

```
class HingeSpeedInput(agxMex.PythonInputArgument):
    def __init__(self, name, numInput):
                super().__init__(name, numInput)

    # This method is called at the startup of the simulation
    # It will get the intialization data from Matlab/simulink
    def input(self, sim, data):
        assert(len(data) == 1)  # Make sure we have 1 element

        speed = input[1]
        constraint = sim.getHinge("Hinge")
        assert(constraint)
        motor = hinge.getMotor1D()
        motor.setSpeed(speed)
        return True
```

The init input control is then registered with the following two lines:

```
# Register the Input class to the controller
argA = HingeSpeedInput("Hinge", 1)
agxSDK.SimulationControl.instance().addInitInputArgument(argA)
```

Next we want to define the simulation output signals:

```
class HingeAngleOutput(agxMex.PythonOutputArgument):
    def __init__(self, name, numInput):
                super().__init__(name, numInput)

    # This method will be called every timestep
    # Reading data from AGX and feeding it back to Matlab/Simulink
    def output(self, sim, data):
        constraint = sim.getHinge("Hinge")
        assert(constraint)
        angle = hinge.getAngle()
        output.append(angle)
        return True
```

Which is then registered to the controller:

```
# Register the Output class to the controller
argB = HingeAngleOutput("HingeStep",1) # One output element
agxSDK.SimulationControl.instance().addOutputArgument(argB)
```

42.6.1.2 Evaluating inputs and outputs

Each time step, AGX does the following three steps:

1. **Call all input functions in the order of their declaration in the Python script, making use of the input from Simulink to control the AGX simulation.**
2. **Take one simulation step**
3. **Call all output functions in the order of their declaration in the Python script, writing results from the AGX Simulation back into the Simulink output data.**

42.6.1.3 Initialization input and output

If you want to make the scene setup depend on parameters, you can set these using the function:

```
agxSDK.SimulationControl.instance().addInitInputArgument(<object derived from
agxMex.PythonInputArgument>)
```

where <InputArgument> is the same kind of argument as for:

```
agxSDK.SimulationControl.instance().addInputArgument(<Object derived from
agxMex.PythonOutputArgument>)
```

If you want to get one-time output from the scene initialization, this can be specified using the function:

```
agxSDK.SimulationControl.instance().addInitOutputArgument(<object derived from
agxMex.PythonInputArgument>)
```

The order of execution at scene initialization is:

- Parse Python script, and register all arguments which must be added to **agxSDK.SimulationControl.**
- Execute all instances of agxMex.PythonInputArgument added with addInitInputArgument
- Execute the function **buildScene().**
- Execute all instances of agxMex.PythonOutputArgument added with addInitOutputArgument

42.6.2 Lua

Here we will use *<AGX-dir>/data/matlab/scripts/rotatingBox.agxLua* as an example.

42.6.2.1 Defining the parameters

When loading the Lua script, AGX will first parse it and execute all commandos which are not inside in a function, that is all code in *global scope*.

Inputs and outputs (both for scene initialization and step-function) must be defined outside of functions (global scope).

The scene itself is defined in the function **buildScene(sim)** in the Lua script. This function will load the simulation scenario.

In the simple case of the example file, the whole scene is setup directly in this Lua function. In more complex cases, you might want to call one or several separate Lua scripts to set up the scene, or load in data from other files.

In the example file, the code line 45:

```
agxSDK.SimulationControl:instance():addInputArgument(agxMex.LuaInputArgument("Hinge-speed",
"hingeInput", 1))
```

register one input argument with the name “Hinge-speed” and size 1. The function which will be executed for this argument at input (more about that later) is called “hingeInput”.

The code line 46:

```
agxSDK.SimulationControl:instance():addOutputArgument(agxMex.LuaOutputArgument("Hinge-angle",
"hingeOutput", 1))
```

register one output argument with the name “Hinge-angle” and size 1. The function which will be executed for this argument at input (more about that later) is called *hingeOutput*.

The constructor for agxMex.LuaInputArgument and agxMex.LuaOutputArgument takes the arguments *name*, *function*, *size*.

- *name* is a string that can be chosen freely, but is preferable short so that it can fully be displayed in the block in Simulink.
- *function* is a Lua function which should be executed in this file (see more in the next subsection).
- *size* is the number of floating point values to be read (or written) in the function.

42.6.2.2 Evaluating inputs and outputs

Each time step, AGX does the following three steps:

4. Call all input functions in the order of their declaration in the Lua script, making use of the input from Simulink to control the AGX simulation.
5. Take one simulation step
6. Call all output functions in the order of their declaration in the Lua script, writing results from the AGX Simulation back into the Simulink output data.

Consider again the example file. Here, the code line 45

```
agxSDK.SimulationControl:instance():addInputArgument(agxMex.LuaInputArgument("Hinge-speed",
"hingeInput", 1))
```

declare that we will have an input function with the name *hingeInput* and size 1. This function is defined like this:

```
function hingeInput(sim, input)
  assert(input:size() == 1)
  local speed = input:at(0)
  local constraint = sim:getConstraint("Hinge")
  assert(constraint)
  hinge = tolua.cast(constraint, "agx::Hinge")
  local motor = hinge:getMotor1D()
  motor:setSpeed(speed)
end
```

All input functions have to have the arguments **sim** and **input**, where **sim** is a reference to an **agxSDK::Simulation** and input is an **agx::RealVector** of the size declared in line 45.

In this example, we obtain the hinge from the AGX simulation and set its speed.

For more detailed information about using the AGX API via its Lua wrappers, see chapter 0.

After this, AGX will take its time step, since there are no more input functions.

The only output function in the file is:

```
function hingeOutput(sim, output)
```

```
  -- read output data from simulation
  local constraint = sim:getConstraint("Hinge")
  assert(constraint)
  local hinge = tolua.cast(constraint, "agx::Hinge")
  local angle = hinge:getAngle()
  output:push_back(angle)
end
```

Note that here the second argument changes from **input** to **output**, which is an initially empty **agx::RealVector**. You should make sure that the number of floating points numbers added in this function matches the size defined in the function declaration earlier in line 45.

Right now, the angle of the hinge around its axis is returned as the only output value.

42.6.2.3 Initialization input and output

If you want to make the scene setup depend on parameters, you can set these using the function:

```
agxSDK.SimulationControl:instance():addInitInputArgument(<InputArgument>)
```

where <InputArgument> is the same kind of argument as for:

```
agxSDK.SimulationControl:instance():addInputArgument(<InputArgument>)
```

If you want to get one-time output from the scene initialization, this can be specified using the function:

```
agxSDK.SimulationControl:instance():addInitOutputArgument(<OutputArgument>)
```

where <OutputArgument> is the same kind of argument as for:

```
agxSDK.SimulationControl:instance():addInputArgument(<OutputArgument>)
```

The order of execution at scene initialization is here:

- Parse lua file, and register all arguments which have to be added to **agxSDK.SimulationControl**.
- Execute all functions given as InitInputArguments, in the order in which they were added to **agxSDK.SimulationControl**.
- Execute the function **buildScene(sim, app)**.
- Execute all functions given as InitOutputArguments, in the order in which they were added to **agxSDK.SimulationControl**. The output is written to the Matlab variable **agx_init_output** in case you run AGX from Simulink, and returned as the output variable **initOutput** in case you run AGX from Matlab.

An example which makes use of scene initialization input and output is *data/matlab/scripts/bucketAndGravelPile.agxLua*.

42.7 Examples

42.7.1 Matlab

Examples of matlab scripts which how to use the AGX-Matlab connection is located in <AGX-dir>\data\Matlab\

All corresponding Lua and Python scripts are located in <AGX-dir>\data\Matlab\scripts

To run an example, start Matlab, change directory to <AGX-dir>\data\Matlab\ and enter the name of the matlab script.

42.7.2 Simulink

Examples of Simulink projects which demonstrates the AGX-Simulink connection is located in located in <AGX-dir>\data\Matlab\simulink\

42.8 Limitations

This version of the Matlab/Simulink integration has several limitations.

- Dynamically linked libraries like agxMex.dll, agxCore.dll, agxPhysics.dll and agxOSG.dll will not be unloaded once they have been loaded in Matlab, until Matlab is shut down.
- **You cannot use Matlab and Simulink in the same session. Please restart Matlab if you are working with the Matlab connection and need to use Simulink.**
- Only one AGX block allowed per model.
- After the Matlab command clear has been executed, AGX should not be used (this includes also calls to **clear mex** or **clear all**).

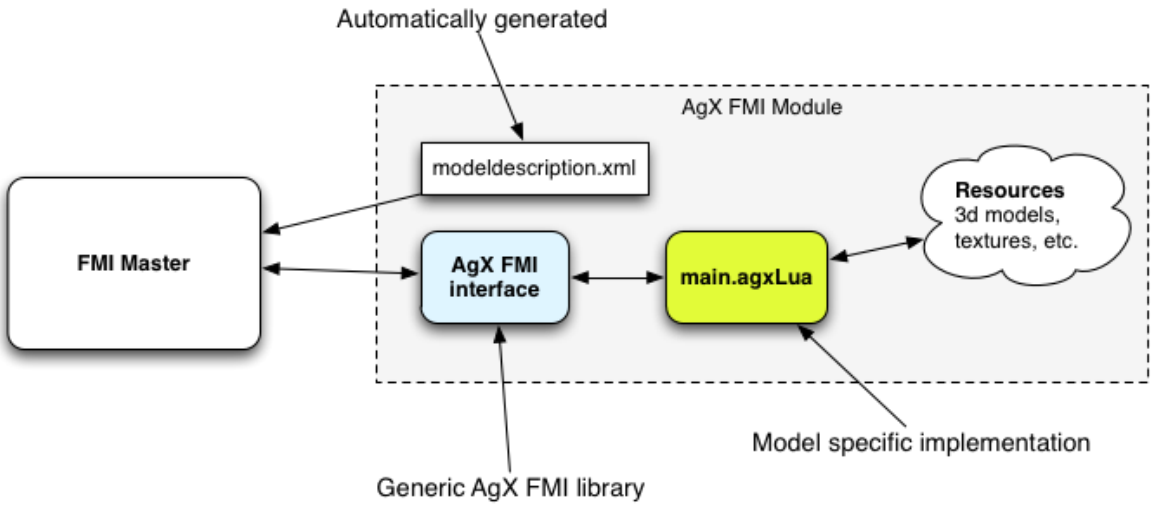
43 FMI Export (optional)

AGX Dynamics supports FMI export for co-simulation using the FMI-standard (see https://www.fmi-standard.org/). The standard (and its interface) is called Functional Mockup Interface (FMI), and one unit created or connected via this standard is called a Functional Mockup Unit (FMU).

The following tutorial demonstrates how to build an AGX Dynamics FMI module compatible with the FMI 2.0 standard. The resulting FMI module can, according to the FMI standard, also be referred to as an FMU (see above).

The FMI export interface is exposed in AgX using the `agxFMI1::Export` namespace. The interface bindings can be defined either through C++ or Lua, where the Lua interface is currently preferred and is used in this tutorial.

The following diagram displays the structure of an AgX FMI module:



Building an AgX FMU consists of

1. Setting up a directory structure matching the structure described in the FMI standard. The FMI module structure is described in section 4 (Model Distribution) in the FMI1 specification, and in section 2.3 (FMU Distribution) in the FMI2 specification.
2. Implementing the Lua script, which is responsible for
    1. Defining the AgX scene to be simulated
    2. Registering the FMI variables to be exported
3. Running the AgX FmiExporter tool which will
    1. Automatically generate the FMI XML interface definition
    2. Package the FMU as a zip file according to the standard

The FMU root directory, from here on referred to as `$FMU_ROOT`, must contain a Lua file named `main.agxLua`. This is the main entry point for building the scene and setting up the module parameters.

The `main.agxLua` file must contain a function called fmiInit, which takes a single parameter, the current AgX FMI module to initialize, an instance of agxFMI1::Export::Module.

This module contains a simulation instance, to which the simulated scene is added, an OSG render node for debug rendering, and methods for registering FMI parameters.

You can find an FMU example in the `data/FmiModules/Tutorial/SpinningBox` directory.

The AGX-FMUs are exported using the FmiExporter tool, which has the following syntax

```
FmiExporter --name <FMU module name> --output <FMU output path> --
fmiVersion <1 or 2> <FMU data root>
```

Name, output, and version have implicit values so you can simply run

```
FmiExporter myFmu
```

where myFmu is the directory containing the main.agxLua fille. In this case the module name will be myFmu, output path will be myFmu.fmu, and FMI version will be 2.

The FmiExporter will automatically generate a FMI modelDescription.xml and package the FMU zip file.

In order to run the FMU, AGX has to be installed on the computer with a valid license and valid environment variables.

44 Appendix 1

44.1 Added mass

When a body moves in a fluid (e.g. water), parts of the fluid has to move around the body. When the body accelerates, so must also the fluid. Therefore, the force required to give a body certain acceleration is larger than it would have to be in vacuum. AGX supports the notation of added mass. The added mass enters the equations of motion except for when calculating the gravitational force, i.e. the gravitational mass is unchanged. It is also used in the mass matrix when AGX solves for constraint forces. The added mass can be specified for each axis through the API calls:

```
rigidBody->getMassProperties()->setMassCoefficients( const Vec3& coefficients );
rigidBody->getMassProperties()->setInertiaTensorCoefficients( const Vec3& coefficients );
```

These coefficients will enter the velocity update equation in the C_A matrix.

Given the 6x6 mass matrix M_0 (linear mass and inertia) and the added mass coefficients C_A, the velocity from the previous time step V_k, the constraint Jacobian G and the Lagrange multipliers λ, time step h and external forces (including gravity) f_e we write the velocity update as:

$$V_{k+1} = V_k + [(1 + C_A)M_0]^{-1}[G^T \lambda + hf_e]$$

Gravity will be calculated with the original gravitational mass as: $F_g = mg$, where m is the original mass.

Note that C_A matrix with the added mass coefficient is a scale of the linear mass M_0.

44.2 Velocity damping

AGX has a damping term (angular and linear) one for each coordinate axis (in the body's local coordinate system). It can be specified through the API as:

```
rigidBody->setLinearVelocityDamping( const Vec3f& damping );
rigidBody->setAngularVelocityDamping( const Vec3f& damping );
```

With this damping term D and the gravitational mass m, we will get a new mass matrix M' such that:

$$M' = m \begin{bmatrix} (1 + D_x h) & & \\ & (1 + D_y h) & \\ & & (1 + D_z h) \end{bmatrix}$$

Analogous for the moment of inertia tensor.

Now given this effective mass matrix M' and the original mass matrix M we get the velocity integration equation:

$$V_{k+1} = M'^{-1} M V_k + M'^{-1} G^T \lambda + M'^{-1} h f_e$$

This means that if damping > 0, the term $M'M$ will be < 0 and the velocity will be damped with a viscous damping term. The effective mass (including damping) will enter the system of equation on the left side, illustrated in the simplified linearized stepping equation below:

$$\begin{bmatrix} \mathrm{M}' & -\mathrm{G}^{\mathrm{T}} \\ G & \varepsilon \end{bmatrix} \begin{bmatrix} \mathrm{V}_{\mathrm{k+1}} \\ \lambda \end{bmatrix} = \begin{bmatrix} -\mathrm{hf}_{\mathrm{e}} \\ \mathrm{g} + \mathrm{G}'\mathrm{v} \end{bmatrix}$$

For a more complete description see equation 4.31 in 2.

45 Appendix 2 Evaluation of AGX Dynamics

45.1 Running tutorials

The tutorials which are prebuilt, reside in the **bin** directory. They can be started from windows explorer just by clicking on them. Paths are setup so that the executables will look for data in the **data** directory.

Running applications linked to agx without having setup the runtime environment (PATH) will lead to an application crash.

If you suspect that one more dll files are missing, use the **depends** tool from Microsoft (freely available on their website):

```
depends tutorial_basicSimulation.exe
```

It will give you a list of all the required runtime files necessary for running the application. It can also be run on a dll file:

```
depends agxPhysics.dll
```

which lists all the runtime dependencies for AGX main library.

45.1.1 Testing performance

When testing performance of a physics engine, it is important to remember the following things:

- Do not run with graphics. The agxOSG library will not run AGX in a thread separate from the graphics. To achieve this use the –agxOnly argument with agxViewer. For example:

  ```
  > agxViewer beam.agxLua -agxOnly -stopAfter 10
  ```

  The above command would load and run beam.agxLua without graphics for 10 seconds and stop and report the time it took to run the simulation.
- Do not use more threads than you have actual execution units on your CPU (cores). On some systems *hyperthreading* is enabled. This means that you will see more cores than what the hardware actually have. Hyperthreading works well with “light threads”, which does not do intensive float/calculations. You will notice a very uneven performance in AGX if you enable too many threads. What will happen is that you will spend a lot of time scheduling threads in and out of the cores leaving less time for actual calculations. This means that calling agxViewer --numThreads 10 will certainly NOT run faster than --numThreads 2 on a two core machine.
- Make sure that the machine is idle before starting your testing.

45.2 agxOSG – Binding to a rendering engine

Chapter 39.1.2 documents keybindings for the applications built using the agxOSG::ExampleApplication framework. The ExampleApplication is a collection of utilities found in the **agxOSG** directory as source. This can work as an example of an integration of agx into a rendering engine (www.openscenegraph.org). agxOSG is mostly for internal use at Algoryx and is not very well documented.

Worth noticing is that rendering and simulation is executed from the same thread This means that if you want to measure the performance of AGX Dynamics only, you should

run the tutorial/application without interactive graphics. This can be done for example through the arguments through the agxOSG::ExampleApplication class which all of the tutorials and AGXViewer is based on:

```
cmd> tutorial_customConstraints --scene 1 --stopAfter 10 --agxOnly --timeStep 0.01 --
numThreads 2
```

The above command would do the following:

- start the application tutorial_customConstraints
- load scene 1 as defined in the application (--scene 1)
- run the simulation for 10 seconds (--stopAfter 10)
- Run as fast as possible with no graphics/window (--agxOnly)
- Use a time step of 100hz (--timeStep 0.01)
- Allocate and use two threads for AGX solver etc. (--numThreads 2)

46 Appendix 3: C#/.NET bindings of the AGX API

Some versions of AGX Dynamics will come with libraries which export the AGX Dynamics API to the .NET development environment (currently version 4.0). By utilizing swig (http://www.swig.org), a large portion (and growing for each version) of the AGX API is exposed to use in .NET. This means that you can utilize the development environment from Microsoft and build C#/VB-based applications with physics from AGX.

The .NET wrapper comes with two layers of run time libraries. First is the C++ wrapper:

Library name	Description
agxDotNetRuntime.dll	AGX C++ .NET binding

Then the top layer which implements the actual .NET wrapper in C#:

Library name	Description
agxDotNet.dll	Interface library for using AGX in .NET

agxDotNet.dll is the interface library to be referenced in your typical .NET application. The C++/.NET binding library agxDotNetRuntime.dll will be automatically located according to the rules by which Windows locates dynamic libraries.

46.1.1 Functions

Since there are no free functions in C# (functions which are not members of a class), only classes with namespaces, the free functions that are available in the C++ API will get a somewhat different look. They will be added into an extra namespace (class). For example, the function **agx::init()**: As this namespace (agx) resides in the agxDotNet plugin, it will in C# be accessible as:

```
agx.agxSWIG.init();
```

This convention goes for all (class-less) functions found in the AGX API.

46.1.2 Sample application using the C# binding

Below is a short code snippet, illustrating various API calls in the .NET version of the AGX programming API in C# code. For more examples, look into the directory: agx\swig\configuration\agxDotNet\testApplication

```
class Program
{
  static void Main(string[] args)
  {

    // Try to use the Environment class.
    agxIO.FilePathContainer fp = agxIO.Environment.instance().getFilePath();
    string script = fp.find("data/luaDemos/beam.agxLua");
    Console.WriteLine("Script {0}", script);

    // This must be the first call to AGX (except for setting up the FilePath).
    // This is because the call to agx.init() (or in .NET: agx.agxSWIG.init())
    // will try to load plugins and
    // various resources. You have to specify where AGX can find these plugins,
    // license file etc.
    // See the user manual for more (important) information about this.
    //
```

```
    agx.agxSWIG.init();
    agx.agxSWIG.setNumThreads(3);

    {
      agxSDK.Simulation sim = new agxSDK.Simulation();
      sim.setDynamicsSystem(new agx.DynamicsSystem());

      agx.RigidBody body = new agx.RigidBody();

      body.setName("this is a body");

      var frame = new agx.Frame();
      frame.setLocalTranslate(2, 0, 0);
      body.setPosition(4, 0, 0);
      var hinge = new agx.Hinge(body, frame);
      hinge.getMotor1D().setEnable(true);
      hinge.getMotor1D().setSpeed(0.1);
      sim.add(hinge);

      agx.RigidBodyRefVector bodies = sim.getDynamicsSystem().getRigidBodies();

      sim.stepForward();

      for (uint i = 0; i < bodies.size(); ++i)
      {
        var bodyName = bodies.at(i).getName();
        Console.WriteLine("A body: " + bodyName);//.c_str());
      }

      agx.TimeGovernor tg = sim.getDynamicsSystem().getTimeGovernor();

      sim.remove(body);

      sim = null;
      body = null;
      wire = null;
    }
    GC.Collect();
    GC.WaitForFullGCComplete();
    agx.agxSWIG.shutdown();
  }
}
```

46.1.3 Building the SWIG binding

The sub-directory <agx-install-dir>\SWIG contains everything needed to build the C#/.NET binding. For information of how to build the libraries, read the README.TXT in the SWIG directory.

46.1.4 Extending the C# interface

It is possible to extend the .NET interface by adding more classes/header files to the .i files found in the <agx>\swig\configuration sub directory. For more details about SWIG's interface files look at the SWIG documentation (www.swig.org).

Adding another header file to SWIG can be done with the following steps:

1. **Locate the .h file which contain the class you want to export to .NET**
2. **Identify which namespace the class/header file belongs to.**
   a. **For example agx::Constraint belongs to the *agx* namespace, then it should be included into *agx.i***
3. **Edit the interface file (for example agx.i) and add two include directives:**
   a. **Locate the use of the INCLUDE() macro. Add your header file: INCLUDE(namespace/headerfile.h) which will be added to the generated .cpp code.**

4. Run generate.bat /WITHDOTNET to allow SWIG to parse the .i files and generate .cpp and .cs files
5. Start visual studio with the build\agxDotNet\agxDotNet.sln: `devenv /useenv build\agxDotNet\agxDotNet.sln`. (The /useenv is important for locating the header files/library files).
6. To test that the new class exists, add code in program.cs that references the new class.
7. When it works, copy the dll files found in SWIG\build\agxDotNet\bin to the path where you keep the AGX runtime libraries.

47 References

[1] E. Hairer, C. Lubich, and G. Wanner. Geometric Numerical Integration, volume 31 of Springer Series in Computational Mathematics. Springer-Verlag, Berlin, 2001.

[2] Claude Lacoursiére. Ghosts and Machines: Regularized Variational Methods for Interactive Simulations of Multibodies with Dry Frictional Contacts. PhD thesis, Dept. of Computing Science, Ume˚a University, June 2007.

[3] K L Johnson. Contact Mechanics. 1985. doi: 10.1115/1.3261297. url: http://www.amazon.fr/Contact-Mechanics-K-L-Johnson/dp/ 0521347963.

[4] AntonioPérez-González et al. “Amodified elastic foundation contact model for application in 3D models of the prosthetic knee”. In: Medical Engineering & Physics 30.3 (2008), pp. 387–398. issn: 13504533. doi: 10.1016/j.medengphy.2007.04.001. url: http://linkinghub.elsevier.com/retrieve/pii/S1350453307000616.