# Kangaroo 2.3.3  Release notes

11/07/2017


Kangaroo is a library and set of Grasshopper components for form-finding, physics simulation, and geometric constraint solving.

These notes are a brief overview and introduction to the features of Kangaroo. For more details, example files, or to ask any questions, please visit the Kangaroo section of the GH forum, and I will do my best to answer.

If you are using the Rhino WIP, Grasshopper and Kangaroo now come included, so you can skip the installation described below if you want. However, at the time of writing this Kangaroo release is more current than the one included in the current Rhino WIP, so you may choose to install it anyway. If you do, you will see some warnings when starting Grasshopper - you can simply click 'Replace All' to dismiss these and load this newer version in place of the default one.

You can install Kangaroo2 alongside the old Kangaroo (0.099), and earlier definitions will still work, but the actual ForceObjects/Goals are not cross-compatible, since Kangaroo 2 was a complete rewrite.
I am trying to phase out the old version - If there is some feature from it that you want and it is still missing in the current Kangaroo - let me know and I will try and add it. Also if you want to know how to achieve the same result as some example or file made with the old version - please let me know and I will try and show something similar in the current one.

*Please note that this is still a work-in-progress release, and all aspects are still subject to change.*

## Example files

Unlike previous releases where a collection of examples was included with the release download, I am now maintaining a collection of example Kangaroo definitions on GitHub here:
https://github.com/Dan-Piker/Kangaroo-examples
(To download the whole collection click 'Clone or Download' and 'Download ZIP')
This way I can more easily modify and add to the collection thout having to change this release on Food4Rhino, and you can know you are getting the most current versions of all the files.

## What's new in this version?

2.3.3 - Fixed bug where 6dof nodes were not updated in the 'zombie' solver component

2.3.2 - Includes a new 'RigidPointSet' goal

2.3.1 - Includes several new goals, including some to simplify common operations, such as preserving the edge lengths of a mesh, or applying gravity to its nodes. Also - the output of the Solver component will now reconstruct any data tree structure that was present in the input. This should simplify downstream operations where you need to separate the outputs.

This version also fixes some compatibility issues with the Rhino WIP.

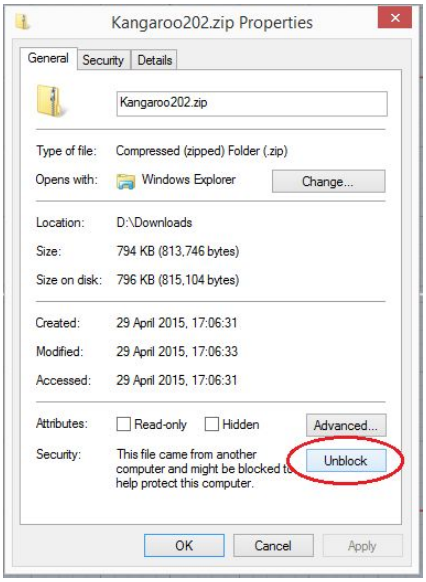
2.3.0 - includes a few more goals which use the 6dof nodes for mechanical mates.
Also new in this version are an EqualAngles goal, a Bomb goal, a Wind goal, several bug fixes, and a reorganization of the goals tab.
Also, the anchor goal now has an option to set the target location separately from the initial location, which can be useful when you want to move it during simulation.
There is also a new solver component to ease the production of animations.
Also, more utilities such as mesh processing tools are included with Kangaroo2, the intention being that new users should not need to install the old version, only those who have a particular need to open old definitions.

2.2.1 - introduced ***6 degree of freedom (6dof) nodes***. This allows for more robust rigid bodies and interactions between them, such as collisions.

Installation

You need a recent Rhino service release and the latest version of Grasshopper installed first (Rhino SR11, and GH 0.9.0076 at the time of writing)
So far this release is only tested on Rhino 5 64bit and might not work on 32bit.
You will also need a recent version of the .NET framework installed (http://www.microsoft.com/net)

First *unblock* the zip file
(by right clicking the file in Windows Explorer, and choosing properties. At the bottom of the General tab, a message about the file being blocked may appear. If so, click unblock, and apply),
*then* unzip it.




Put the 2 files (*Kangaroo.gha* and *KangarooSolver.dll*) in the Grasshopper components folder (replacing earlier versions of these files if they exist).
If you are having trouble loading the components, double check that both of these files are unblocked. (Unblocking the zip first *should* mean that the files after unzipping are all unblocked, but it seems this is not always guaranteed).
Note also that unblocking needs to be done to each file individually, and does not work on multiple selected items.

After installing, close and reopen Rhino, and look for the Kangaroo2 tab in Grasshopper.
You may also find you need to turn COFF loading off (by typing GrasshopperDeveloperSettings from the Rhino command prompt and unchecking the box), then restarting Rhino.

If after following these steps the plugin still does not load, please let me know, with information about any error messages, and which versions of Windows, Rhino and Grasshopper you are running.



## Introducing *Goals*

While previous versions used *ForceObjects*, these have now been replaced with something called *goals* or *goal objects*.
These unify and encompass the ideas of forces, energies and constraints in a single system.

Essentially a *goal* can be any function which, given the current positions of some points, returns some target positions it 'wants' to move them to.
Kangaroo contains many different types of goals, which can be applied in any combination, and the solver will adjust the geometry in order to meet them all.
If the goals conflict with each other and no simultaneous solution to all of them is possible, the points will find a compromise between them. More specifically, it will minimize the sum of their weighted squared distances from their targets. Each goal can be assigned a scalar value for its weighting or relative importance.

Because of their fast convergence, these goals can be used for interactive constraint solving, including both under and over constrained problems. However, unlike constraints in the traditional sense, these *goals* can also include things like applied loads, which try and move a point in a given direction, but can never actually be met.
They can also include elastic material behaviour, with the goal geometry being the rest state.
Another way of seeing these goal objects is as quadratic energy functions to be minimized, with the target points as the zeros of the energy.

## Solver component

The solver component can be found in the main panel of the Kangaroo2 tab, and is the component which collects all the goals and solves the system.
The timer/remote control needed in the old Kangaroo is gone. The solver automatically runs until convergence and then stops. For simple setups this can be pretty much instant, so you will just see the equilibrium result. For larger or more complex sets of GoalObjects, it will display the current state every 15-30ms or so.

The *Tolerance* setting on the solver component determines the distance below which multiple points get combined into a single particle.

-GoalObjects can now be added and removed without resetting. Coincident points (within a tolerance) will be combined automatically.

-There is a special "Grab" GoalObject. When this is connected, it lets you move any of the simulation points in the Rhino viewport by holding down the *Alt* button on your keyboard and dragging with the Left mouse button.
(Making it so you have to hold the Alt key is a way of stopping this interfering with mouse clicks for normal Rhino operation)

Note- When using the Grab object, even in top view, points may get moved slightly out of plane, therefore if you want to keep some geometry on a 2d plane, you should include an OnPlane goal acting on all the points.

As of version 2.01 the Grab object also has a range setting, which controls how close you need to click to a point to select it.

-There is no longer any need to Flatten the GoalObjects input, this is done automatically.

-Individual GoalObjects have an option to output some data after the optimization. These come out of the O output of the Solver component. This can include geometry to display (currently the Length object outputs a line, and the Planarize component outputs a number for the twist amount of each face), but the idea is that this is a general holder for any information you might want to get out from a custom GoalObject, such as stresses etc. The outputs come out as a flat list in the order the GoalObjects are read in. In future the plan is to preserve the tree structure here for easier organization of results.

-The *Show* object (in the Main panel, with the lightbulb icon) is used to attach additional geometry to be displayed to the points involved in the simulation. It works with lines, polylines, arcs and meshes.

-Weighting/Stiffness/Strength values can be set arbitrarily high and the system should remain stable (this was a major limitation of earlier versions, particularly for simulation of real material stiffness). However, if you have extremely high ratios between the strongest and weakest constraints (like 1e12 and 1e-12), then movement towards the weaker goal will be slower, and it may take a long time to converge.
If you need one high priority goal much stronger than others (sometimes distinguished as hard vs soft constraints), but still need the lower priority goals to influence the form (such as when planarizing the quads of a mesh, but also wanting the overall form to be smooth), it is recommended to start with all goals at moderate ratios while creating the general form, then increase the weighting of the hard constraint until it is enforced as exactly as required.

## Goal catalogue

What follows is a brief description of each of the currently available goal objects in Kangaroo 2.

Some of these will be familiar from the old version, while many are completely new. Not all the force objects from the old version have a goal equivalent here yet, but the intent is to eventually add all of them and more. I am adding new goals all the time, and welcome suggestions for more, and you can also script your own (see the section at the end about scripting Kangaroo).

Anchor

This keeps a particle in its original location.

AnchorXYZ

An Anchor with options for which of the world directions to restrain the point in. For example, to allow a point to move only in the world XY plane, you would set *X=false,Y=false,Z=true*.
To allow it to move only along the X-axis, you would set *X=false,Y=true,Z=true*.

Length

Tries to keep 2 points a given distance from each other. Note that if no input for the length is supplied to the component, it will take the starting length as the target length (instead of defaulting to 0 as in earlier versions).

The *length* goal object can also be seen as a linearly elastic 1D finite element, and its deformations under load can be assigned real quantitative meaning, by setting the strength to the spring constant in N/m, calculated as EA/L, where E is the Young's modulus of the material in Pa, A is the cross-sectional area in $m^2$, and L is the rest length in m.

ClampLength

Keeps the distance between 2 points between given bounds, but applies no force when the distance is within these limits. This is similar to the SpringCutoffs in previous versions of Kangaroo.

EqualLength

Tries to make a set of lines equal length. Applying one EqualLength goal for the 4 sides of a quad, and another for its 2 diagonals can be a way of making it more square, sometimes a desirable quality in meshes.

Load

A force vector. The length of the vector is the magnitude in Newtons of the applied load.

Coincident

Keep two points at the same location. This is equivalent to making a zero length goal between a pair of points.

Angle

Keeps 2 line segments at a given angle relative to each other. If no angle is supplied the starting angle is used. The line segments can be disconnected or they can share a common point, in which case this simulates the bending of an elastic rod of circular cross-section without torsion. The formulation of the bending energy used is from Adriaenssens and Barnes 2001. To assign values based on real material properties, use a Strength input of E*I, where E is the Young's modulus in Pa, and I is the second moment of area in m.

ClampAngle

Like Angle, except it allows you to set upper and lower bounds, between which it will not apply any force.

Direction

This rotates a line segment to align it with a given vector direction. If no direction is supplied, it will take the closest of the +/- world XYZ directions, and can be used to snap geometry to orthogonal.

Floor

Keeps particles from passing below Z=0

LengthSnap

Snaps the length of a line to the closest whole number multiple of a given value

AngleSnap

Snaps the angle between 2 lines to the closest whole number multiple of a given value

Planarize

Flattens each of the quads in a mesh.

CoPlanar

Pulls a set of points towards their best fit plane. This can act on any number of points, but if you have only sets of 4, use Planarize instead, as it will be faster.

CoSpherical

Pulls a set of points towards their best fit sphere

CoLinear

Pulls a set of points towards their best fit line

OnCurve

Pulls a point towards the closest point on a given curve

OnPlane

Pulls a point towards the closest point on a given plane

OnMesh

Pulls a point towards the closest point on a given mesh

Hinge

Bending resistance between a pair of adjacent triangles. Useful for simulation of plates and shells.
The formulation of this energy is based on Tachi 2013

PlasticHinge

Similar to the hinge described above, except an elastic/plastic threshold angle is given, and when bent more than this, the material deforms plastically.

CyclicQuad

Tries to make the 4 vertices of a quad lie on a common circle. Should be used in conjunction with planarize. Can be used for generating circular meshes, which have useful offset properties for beam structures.

TangentIncircles

Adjusts the vertices of a pair of adjacent triangles so that their incircles become tangent. This can be useful for generation of torsion free beam layouts and other structures.
Energy from Hoebinger 2009

PlasticAnchor

Like a regular anchor, this acts as an elastic force trying to keep a particle fixed at a particular target position. However, unlike the regular anchor, the target position of a PlasticAnchor will move if it is pulled hard enough. The parameter L sets the distance limit at which the anchor will start to shift.
It can be thought of as acting like the static friction that keeps objects sitting on a table from sliding around when pushed only lightly, except instead of a surface, the reference is absolute 3d space.

This can be useful for sculpting applications to stop particles from keeping drifting. For example, when you have a mesh with smoothing forces acting on it, if left alone, it will continually shrink and flatten itself out, even if the smoothing forces are small. Applying a PlasticAnchor to every vertex of the mesh will allow the smoothing to round out the shape, but once the smoothing force is less than the plastic limit of the anchor it will stop moving.

PlasticLength

This will behave elastically at first, like a regular Length goal, trying to keep a pair of particles a certain rest distance apart, but if stretched or compressed beyond the given limit, this rest distance will change, simulating plastic deformation.

PolygonArea

Pushes the segments of a closed polyline inwards or outwards so that they enclose a planar shape of the given area.
Note - this is a 2d goal, and works only in the XY plane.

Smooth

Uniform Laplacian smoothing, rounds any sharp features of a mesh.
Note that Smooth acts only on internal vertices, and does not affect the naked boundaries of a mesh, and these often need to also be fixed or smoothed in some way.

Transform

This is a goal with many potential uses, as it maintains a given transformation between a pair of points, so it can preserve various types of symmetry, periodic boundary conditions etc. Grasshopper transformation objects (such as reflection, rotation, translation, scaling...) have a *Transform(X)* output which can be used as the input here.

RigidBody

This treats a given solid mesh (you can also supply a Brep as input) as a rigid body - meaning it has 6 degrees of freedom (3 for translation and 3 for rotation), and this rigid body motion is treated separately from any deformation. You can attach points anywhere in space to the rigid body, and they will stay at this relative position as the body moves and rotates (note these attached points can be on or in the solid, but they do not need to be).

Mechanical assembly mating constraints, such as revolute hinges, pin-joints, sliders etc, can be applied by attaching the same points to 2 rigid bodies. For instance, if 2 bodies both have the 2 endpoints of a single line as attachment points, this line becomes a revolute hinge. If they share only one point, this will become a pin-joint, etc. These rigid bodies are a relatively recent addition, and still subject to further testing.

Collide2d

Collisions between closed polygons in a given plane.

Collider

Collisions between a mix of thickened line segments(capsules) and spheres of various radii (using a broad-phase collision to speed things up).

SphereCollide

Collisions between equal sized spheres, optimized for speed.

SolidPointCollide

Collisions between points and a closed solid. The solid can be supplied as either a mesh or a Brep. There is an option for whether you want to keep the points inside or outside the solid. From version 2.02, this component also includes an option for whether to use 1-way(Unidirectional) or 2-way interaction.
If UniDirectional=True, the solid is used just as an input, which you can change or move during simulation, but it is not itself affected by the physics (I think Maya calls these 'passive colliders', but I feel *UniDirectional* is more descriptive of how they actually work).
The default 2-way behaviour can be used for collisions where you want the mesh to also be moved and deformed by the collision forces.

Volume

Control the total volume of a mesh. This can be used for inflatables.

MagnetSnap

Applied to a collection of points, if any 2 points get closer together than the Range R, they will be 'snapped' together.

SoapFilm

An area minimizing element, which can be used for finding minimal surfaces. Since this is derived from continuum mechanics, the resulting shape is not dependent on the density of initial meshing used, as is the case if using zero-length springs for the edges. However, because these soap film elements produce forces only in the normal direction of the surface, they usually need some other goals to keep the points from drifting or the triangles becoming too distorted. In some cases this can be achieved by allowing freedom of the nodes in only one direction. Alternatively you can combine them with tangential smoothing (see below).

The SoapFilm goal object requires a list of 3 points as input. So to apply to a mesh, you must triangulate it, and input the 3 points of each face. (Note - take care if using the face polyline boundaries here, as exploding a triangle polyline will give 4 vertices, because it counts the first one twice. Instead, take the start points of the 3 segments of the exploded polyline)

Soap film elements are useful if you are after true zero-mean curvature minimal surfaces, or membranes where the stresses are the same in all directions. However, they are a bit more complex to use, and in many cases, simply taking the mesh edges as Length goals with a target length of zero is an easier method of tensile form-finding.

TangentialSmooth

Typically used in combination with SoapFilm elements. This smooths the vertex positions of a mesh, but only in the directions across the surface, with no effect in the surface normal direction. So it keeps points well distributed, but should not affect the overall shape.

Scripting Kangaroo

A big part of this rewrite has been making Kangaroo more modular, and allowing greater customization through scripting. Documentation for this library is slim at the moment, but is a work in progress, and I will also be happy to help on the forum if you have any questions in the meantime.

Defining a custom goal object is as simple as making a rule for a set of particles telling them where to move to (this is in contrast with other systems where you need to provide functions for first or second derivatives, or differential coordinates). The hope is that this simplicity will allow a wide variety of possible goals.

The examples folder includes some examples of creating custom goal objects and custom iterations using C# (though you can also use VB or Python).
The Kangaroo library is separate from the Grasshopper components, and does not depend on any GH functions, so you can also reference the library and call the functions from scripts running directly in Rhino.