Sverchok Documentation

Release 0.5

nortikin

Mar 27, 2017

Contents

Contents:

CHAPTER 1

Installation

Troubleshooting Installation Errors

NumPy

We now include NumPy code in Sverchok nodes, this means that you should have an up-to-date version of NumPy on your machine. Normally if you get your Blender precompiled NumPy will be included with Python, however this isn't always the case. The windows builds on blender buildbot may contain a cut down version of NumPy due to the licenses under which it can be spread in compiled form.

If you get an error when enabling Sverchok the last lines of the error are important, if it mentions:

- ImportError: No module named 'numpy'
- multiarray
- DLL failure
- Module use of python33.dll conflicts with this version of Python

then here are steps to fix that[1]:

- download and install Python 3.4.(1) for your os
- download and install NumPy 1.8 (for python 3.4) for your os.
- in the Blender directory rename the *python* folder to *_python* so Blender uses your local Python 3.4 install.

binaries:

- python: https://www.python.org/downloads/release/python-341/
- numpy: http://sourceforge.net/projects/numpy/files/NumPy/

To confirm that NumPy is installed properly on your system, for py3.4, launch your python34 interpretter/console and the following NumPy import should produce no error.:

[1] If you get an error, this means NumPy failed to install. We can't really troubleshoot that

```
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:38:22) <edited>
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>>
```

CHAPTER 2

Introduction to geometry

Basics

If you've ever created a mesh and geometry programatically then you can skip this section. If you are uncertain what any of the following terms mean then use it as a reference for further study:

```
List, Index, Vector, Vertex, Edge, Polygon, Normal, Transformation, and Matrix.
```

List

As a visual programming language Sverchok borrows many terms from text-based programming languages, specifically from `Python`. Sverchok uses *Lists* to store geometry, *Lists* offer fast and ordered storage. The items stored in Lists are called *Elements*. Each element of a List is assigned a unique sequential index.

Index

plural: Indices

Indices allow us to quickly reference a specific element of a List. The index of the first element is 0 and the index of the last element is equal to the number of total elements minus 1.

3D Geometry

Vector

The most fundamental element you need to know about is the *Vector*. Think of Vectors as things that have a multitude of properties (also called components). For example *House prices* are calculated depending on maybe 20 or more different properties: floor space, neighbourhood, age, any renovations, rooms, bathrooms, garage... The point is, a house can be seen as a Vector datapoint:

```
House_one = Vector((floor_space, neighbourhood, age, renovations, rooms, ...))

# or simply
House_one = (floor_space, neighbourhood, age, renovations, rooms, ...)
```

3D Geometry concentrates mostly on a small number of components. `X, Y, Z, and maybe W`. If you've ever scaled or moved a model in 3d space you have performed Vector Math on the locations of those 3d points. The concept of *3d location* or *point in 3d space* is so important that the Vector used to describe the idea has a special name; *Vertex*, and it is a special, limited, case of a *Vector*. More about this later.

Understanding Vectors and Vector math is an integral part of parametric modeling and generative design, and it's a lot easier than it might appear at first. You won't have to do the calculations yourself, but you will need to feed Sverchok meaningful input. The good news is that figuring out what Vector math operations produce which results can be learned through observation and understood by experimenting interactively.

The various ways in which Vectors can be manipulated will be covered in subsequent parts. If you want to do cool stuff with Sverchok spend time getting to understand Vector based math, it will be time well spent.

Vertex

plural: Vertices

A vertex is a point in 3d space described by 3 or 4 values which represent its X, Y and Z location. Optionally a 4th value can represent a property of the vertex, usually *influence* or *weight* and is denoted by **W**.

A quick Python example should clarify this. The following would make 3 vertices. In this case each vertex has 3 components.:

```
v0 = (1.0, 1.0, 0.0)
v1 = (0.5, 0.6, 1.0)
v2 = (0.0, 1.0, 0.0)
```

Mesh objects in Blender contain geometric data stored in *lists*. In Python and Sverchok an empty list looks like `[ ]` (open and closed square brackets). Vertices are stored in lists too, a list with 3 vertices might look like:

```
vertices = [
    (1.0, 1.0, 0.0),
    (0.5, 0.6, 1.0),
    (0.0, 1.0, 0.0)
]
```

Edges

Edges form a bond between 2 vertices. Edges are also stored in a list associated with the mesh object. For example the following sets up an empty list to hold the edges:

```
edges = []
```

If we want to declare edges, we reference the vertices by index. Below is an example of how 3 edges are created:

```
edges = [[0, 1], [1, 2], [2, 0]]
```

Here you see we are using lists inside lists to help separate the edges. This is called *Nesting*

Polygons

also called Faces or Polys

Polygons are built using the same convention as Edges. The main difference is that polygons include at least 3 unique vertex indices. For the purposes of this introduction we'll only cover polygons made from 3 or 4 vertices, these are called *Tris and Quads* respectively.

Now imagine we have a total of 6 vertices, the last vertex index is 5. If we want to create 2 polygons, each built from 3 vertices, we do:

```
polygons = [[0, 1, 2], [3, 4, 5]]
```

In Blender you might mix Tris and Quads in one object during the modelling process, but for Sverchok geometry you'll find it more convenient to create separate lists for each and combine them at the end.

An example that sets us up for the first Sverchok example is the following pyhon code:

```
# this code can be run from Blender Text Editor and it will generate a Cube.

import bpy

verts = [
    ( 1.0, 1.0,-1.0),
    ( 1.0,-1.0,-1.0),
    (-1.0,-1.0,-1.0),
    (-1.0, 1.0,-1.0),
    ( 1.0, 1.0, 1.0),
    ( 1.0,-1.0, 1.0),
    (-1.0,-1.0, 1.0),
    (-1.0, 1.0, 1.0)
]

edges = []  # empty list for now.

faces = [
    (0, 1, 2, 3),
    (4, 7, 6, 5),
    (0, 4, 5, 1),
    (1, 5, 6, 2),
    (2, 6, 7, 3),
    (4, 0, 3, 7)
]

mesh_data = bpy.data.meshes.new("cube_mesh_data")
mesh_data.from_pydata(verts, edges, faces)
mesh_data.update()

cube_object = bpy.data.objects.new("Cube_Object", mesh_data)

scene = bpy.context.scene
scene.objects.link(cube_object)
cube_object.select = True
```

If we extract from that the geometry only we are left with:

```
v0 = (1.0, 1.0, -1.0)
v1 = (1.0, -1.0, -1.0)
v2 = (-1.0, -1.0, -1.0)
```

```
v3 = (-1.0, 1.0, -1.0)
v4 = (1.0, 1.0, 1.0)
v5 = (1.0, -1.0, 1.0)
v6 = (-1.0, -1.0, 1.0)
v7 = (-1.0, 1.0, 1.0)

vertices = [v0, v1, v2, v3, v4, v5, v6, v7]

polygons = [
    (0, 1, 2, 3),
    (4, 7, 6, 5),
    (0, 4, 5, 1),
    (1, 5, 6, 2),
    (2, 6, 7, 3),
    (4, 0, 3, 7)
]
```

Side Effect of Defining Polygons

A chain of Vertex indices defines a polygon and each polygon has edges that make up its boundary. If a polygon has 4 vertices, then it also has 4 edges (or sides..if you prefer).

example 1

If we take the above polygons list as example and look at the first polygon (index=0), it reads `(0, 1, 2, 3)`. That polygon therefor defines the following edges `(0,1),(1,2),(2,3),(3,0)`. The last edge `(3,0)` is the edge that closes the polygon.

example 2

The polygon with index 3 reads `(1, 5, 6, 2)`, it implies the following edges `(1,5) (5,6) (6,2) (2,1).`

Ready?

I think this broadly covers the things you should be comfortable with before Sverchok will make sense.

Sverchok

This section will introduce you to a selection of nodes that can be combined to create renderable geometry. Starting with the simple Plane generator

CHAPTER 3

Introduction to Sverchok

> Dealga McArdle | December | 2014

You have installed the addon, if not then read the installation notes. If you've ever used Nodes for anything in Blender, Cycles / Compositor Nodes feel free to continue straight to Unit 01 if you see the RNA icon in the list of NodeView types.

Unit 00 - Introduction to NodeView and 3DView

The following Unit(s) will introduce you to the essential parts of the Blender interface that Sverchok uses: The 3DView and the Node Editor (NodeView)

Unit 00. Introduction to Blender, the NodeView and 3DView

Sverchok Installed, what now?

If you have a tickbox beside the Sverchok add-on in the Add-ons list in User Preferences, then it's safe to assume the add-on is enabled. To show the basics you need to have a NodeView open, and it's useful to have a 3DView open at the same time.

NodeView and 3DView

1. **Split a View**:

   To do this we can split the existing 3DView into two views, by leftclicking into the little triangle/diagonal lines in the bottom left of the 3dview, and dragging to the right.

2. **Switch a View**:

   Then you switch the resulting second 3DView to a NodeView (Node Editor)

3. **Sverchok Node Tree**:

   This icon shows that Sverchok Nodes can be loaded, you'll see it among the other Node Tree types.

4. **Make a new Tree**:

   When you start out you will have to press the New button to make a new node tree called (by default) NodeTree

   becomes

5. **Adding Nodes to the View**:

   This View is a NodeView, from here you can use the Add Menu.

Unit 01 - Introduction to modular components

The following Units will introduce no more than 10 node types per lesson. Take your time to get through the parts that are text heavy, some concepts take longer to explain not because they are difficult to understand, but because there is simply more to cover.

Introduction to modular components

prerequisites

You should have a general understanding of Vectors and Trigonometry, if not then soon enough parts of these lessons might be confusing. If you want to get the most of out Sverchok but don't have a strong Math background then work your way through related KhanAcademy content, it's a great resource and mind bogglingly comprehensive.

Lesson 01 - A Plane

Nodes covered in this lesson: `Math, Vector In, Float, Range Float, Viewer Draw, Stethoschope, Formula, Vector Math`.

Let's make a set of 4 vectors and combine them to represent a plane. I'll use the Trigonometric concept of the *unit-circle* to get coordinates which are *0.5 PI appart*.

We carefully pick points on the unit-circle so that when we connect them via edges it results in a square. To begin we want to create a series of numbers, to represent those points on the unit-circle. Essentially this sequence is `[0.25 pi, 0.75 pi, 1.25 pi, 1.75 pi]`. Because these aren't whole numbers, but so called `Floats`, we use a Node that generates a range of Floats: `Range Float`. (or 'Float series' as it's called when added to the node view).

Making a series of numbers

- `Add -> numbers -> Range Float`

This node has a set of defaults which output `[0.000, 1.000..... 9.000]`. We will tell the node to make `[0.25, 0.75, 1.25, 1.75]` and multiply them later with the constant PI.

Seeing the output of the Range Float node

- `Add -> Text -> Stethoscope`

Hook up the *Stethoscope* input into the *Float range* output, you'll see text printed onto the node view. You can change the color of the Stethoscope text using the color property if the background color is too similar to the text color.

Setting up the input values of Range Float to generate the right output

Set the Float Series mode to *Step* and make sure the *Start* value is 0.25 and *Step* value is 0.50. You should type these numbers in instead of adjusting the slider, it's fast and instantly accurate. Set the *Count* slider to 4, whichever way is fastest for you.

Multiplying the series by PI

- `Add -> numbers -> Math` (add two math nodes)

We know the output of the Float series now, what we will do is multiply the series by a constant PI. This is like doing `[0.25, 0.75, 1.25, 1.75] * pi`, which is what we wanted from the beginning, namely; `[0.25 * pi, 0.75 * pi, 1.25 * pi, 1.75 * pi]`.

1. Set one of the Math nodes to the constant `PI`
2. Switch the other Math node to a Multiplier node by selecting `Multiplication (*)` from its dropdowns.
3. Connect the output of PI to one of the input sockets of the Multiply Node
4. Connect the output of the Float Series into the other input of the Multiply Node.

The result should look something like this, hook up the Stethoscope to see the outputs.

Getting the Sine and Cosine of this series

- `Add -> numbers -> Math` (add two math nodes)

These new *Math* nodes will do the Trigonometry for us. Set one of them to a *Cosine* and the other to a *Sine*. These two nodes will now output the *cos* or *sin* of whatever is routed into them, in this case the series of Floats.

See the outputs of the Sine and Cosine node, each element represents a component of the set of Vectors we want to make. Sine will represent *Y* and Cosine will be *X*.

Making Vectors from a series of numbers

- `Add -> Vector -> Vector In`

The *Vector In* node takes as input 1 or more numbers per component. Sockets which are not explicitely connected to will be represented by a zero.

1. Connect the resulting *Cosine* series to the first component in of Vector in (x)
2. Connect the resulting *Sine* series to the second component in of Vector in (y)
3. Leaving Vector In's 3rd socket (z) empty puts a zero as the z component for all vectors generated by that node.

Display Geometry

- `Add -> Viz -> Viewer Draw`

Sverchok draws geometry using the Viewer Nodes, there are two types of viewer nodes but we'll focus on Viewer Draw for the moment. Stethoscope is useful for showing the values of any socket, but when we're dealing with final geometric constructs like Vectors often we want to see them in 3D to get a better understanding.

Connect the output of *Vectors In* into the *Vertices* on the Viewer Draw node. You should see 4 vertices appear on your 3d view (but don't worry if you don't immediately spot them):

Notice the 3 color fields on the Viewer Draw node, they represent the color that this node gives to its Vertices, Edges, and Faces. If (after connecting Vector In to ViewerDraw) you don't see the Vertices in 3dview, it is probably because your background 3dview color is similar to the Vertex color. Adjust the color field to make them visible.

Increasing the Size of the Vertex

Sometimes, especially while introducing Sverchok, it's preferred to display Vertices a little bigger than the default values of 3 pixels. If you had difficulty spotting the vertices initially you will understand why. The N-panel (*side panel*, or *properties panel*) for the Node View will have extra panels when viewing a [Sverchok Node Tree]. Some nodes have a dedicated properties area in this panel to hold features that might otherwise complicate the node's UI.

In the case of the *Viewer Draw*, there's quite a bit of extra functionality hidden away in the properties area. For now we are interested only in the Vertex Size property. In the image below it's marked with a (red) dot. This slider has a range between 0 and 10, set it to whatever is most comfortable to view. Here a close up:

I think you'll agree that the Vertices are much easier to see now:

Make some edges

We've created vertices, now we're going to generate edges. We have 4 vertices and thus 4 indices: `[0,1,2,3]`, the edges will be connected as `[[0,1],[1,2],[2,3],[3,0]]`.

Vertices Indexed:

- `Add -> Numbers -> Formula`

There are numerous ways to generate the index list for *edges*. For our basic example the simplest approach is to write them out manually. Eventually you will be making hundreds of Vertices and at that point it won't be viable to write them out manually. For this lesson we'll not touch that subject.

The formula node evaluates what you write into the *function* field, and then outputs the result to its out socket. Type into that field the following sequence `[[0,1],[1,2],[2,3],[3,0]]`. Now hook the output of Formula node into the `EdgPol` input of ViewerDraw. You should see the following:

Make a first Polygon

We will reuse the Vertices, you can disconnect the Formula node from Viewer Draw. Let's also reuse the Formula node by clearing the *function* field and replacing the content with the following sequence: `[[0,1,2,3]]`. Connect the output of this Formula node to the EdgPol input on Viewer Draw. You should now see the following:

Controlling the size of the Polygon

There are many ways to scale up a set of vectors, we'll use the Vector Math node.

- `Add -> Vector -> Vector Math`

Change the *Vector Math* node's *mode* to *Multiply Scalar*. This will let you feed a number to the Vectors to act as a multiplier. We'll add a `Float` node to generate the multiplier.

- `Add -> Numbers -> Float`

1. Hook up the *Float* node to the Scalar (green) input of the *Vector Math (Multiply Scalar)* node
2. Connect the output of the *Vector In* node into the top input of the Vector Math node.
3. Now connect the output of the *Vector Math* node into the Vertices socket of the Viewer Draw node.

You should have something like this.

Now if you change the slider on the *Float* node, you'll notice 2 things:

1. the header of the Float node gets the value of the slider, and more importantly,
2. the Polygon will start to increase and decrease in size because you are multiplying the *x, y, and z* components of the Vectors by that amount.

End of lesson 01

Save this .blend you've been working in now, somewhere where you will find it easily, as *Sverchok_Unit_01_Lesson_01*. We will use it as a starting point for the next lesson.

We'll stop here for lesson 01, if you've followed most of this you'll be making crazy shapes in a matter of hours. Please continue on to *Lesson 02 - A Circle*, but take a break first. Look outside, stare at a tree – do something else for 10 minutes.

Introduction to modular components

prerequisites

Same as lesson 01.

Lesson 02 - A Circle

This lesson will introduce the following nodes: `List Length, Int Range, List Shift, List Zip`

This will continue from the previous lesson where we made a plane from 4 vectors. We can reuse some of these nodes in order to make a Circle. If you saved it as suggested load it up, or download from **here**. You can also create it from scratch by cross referencing this image.

A Circle

Just like Blender has a Circle primitive, Sverchok also has a built in Circle primitive called the *Circle Generator*. We will avoid using the primtives until we've covered more of the fundamental nodes and how they interact.

Dynamic Polygons

In the collection of nodes we have in the Node View at the moment, the sequence used for linking up vertices to form a *polygon* is inputted manually. As mentioned earlier, as soon as you need to link many vertices instead of the current 4, you will want to make this *list creation* automatic. You will probably also want to make it dynamic to add new segments automatically if the vertex count is changeable.

Because this is a common task, there's a dedicated node for it called `UV Connect` (link) , but just like the *Circle* generator nodes we will avoid using that and for the same reason. Learning how to build these things yourself is the best way to learn Visual Programming with nodes.

Generating an index list for the polygon

In order to make the list automatically, we should know how many vertices there are at any given moment.

- `Add -> List Main -> List Length`

The *List Length* node lets you output the length of incoming data, it also lets you pick what level of the data you want to inspect. It's worth reading the **reference** of this node for a comprehensive tour of its capabilities.

1. hook the Vector In output into the *Data* input of *List Length*
2. hook a new Stethoscope up to the output of the the *List Length* node.
3. notice the *Level* slider is set to 1 by default, you should see Stethoscope shows output.

Notice that, besides all the square brackets, you see the length of the incoming data is *4*, as expected. We want to generate a sequence (list) that goes something like `[0,1,2,...n]` where n is the index of that last vertex. In Python you might write something like this this:

```
n = 4
range(start=0, end=n, step=1)
>> [0,1,2,...n]
```

To generate the index list for the polygon we need a node that outputs a sequential list of integers, Sverchok has exactly such a node and it accepts values for *start*, *step* and *count* as parameters. This is what the *Range Integer (count mode)* node does.

- `Add -> Numbers -> Range Int`

1. Set the mode of List Range int to *Count*.
2. Make sure *start* is 0 and *step* is 1

3. hook the output of *List Length* into the *count* socket of Range Int.
4. Disonnect the Formula node from the EdgPol socket
5. Connect the output of Range Int into EdgPol instead.
6. optionally you can connect a Stethoscope also to the output of Range Int in order to see the generated list for yourself.

Generating the set of circular verts

The 4 verts we've had from the very beginning are already points on a circular path, we can make a simple change to finally see this Circle immerge.

1. Set the *mode* of the Float series node to *Range*
2. Set the *stop* parameter to 2.0
3. Set the *step* to 0.2 for example.

`2.0 / 0.2 = 10`, this means the Float Series node will now output `[0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8]`. Notice that it does not output 2.0 at the end, because this mode excludes the terminating value. (called non inclusive)

You can see the beginnings of a circle.

Forcing an even spread of Vertices

Above we have the step set to 0.2, this manually sets the distance but calculation this step value soon gets cumbersome. We will add nodes to do the calculation for us. Think about how you might do that.

I would want to have something like `1 / number_vertices`, this calls for a Math node and an *Int* to represent the whole number of vertices.

- `Add -> Numbers -> Math`
- `Add -> Numbers -> Int`

1. set the Math node *mode* to `/ (division)` , and put 1.0 in the numerator
2. connect the Int node into the bottom socket of the division Math node.
3. adjust the integer value on the Int node to 18 for example.
4. in the image below I've connected a Stethoscope to the output of the Math Node to see the value of this computation.
5. Finally, hook up the output of the division Math node into the *step* socket of Float series.

You should see something like this, if not you can by now probably figure out what to do.

Notice this is starting to get crowded, let's minimize nodes

Before going any further I would like to draw attention to the fact that you can make nodes smaller. This minimizing feature is called *hide*, we can argue about how good or badly that option is named. With Any node selected press H, to 'minimize/hide'.

In Sverchok we added special functionality to certain nodes to draw information about themselves into their header area. This allows you to see what the node is supposed to be doing even when the UI is minimized. Currently the *Int, Float, Math, Vector Math* nodes have this behaviour because they are essential nodes and used very often.

In future lessons you will often see minimized/hidden nodes

Polygon is easy, what about Edges?

Remember, there are nodes that can take an incoming set of vertices and generate the required Edges index lists. But we're trying to explore the modular features of Sverchok – we'll build our own Edges generator this time.

The edge index list of the square looked like `[[0,1],[1,2],[2,3],[3,0]]`. For the Circle of a variable number of vertices that list will look like `[[0,1],[1,2],...,[n-1,n],[n,0]]`. Notice i'm just showing the start of the list and the end, to indicate that there might be a formula for it based purely on how many verts you want to link.

In python you might express this using a for loop or a list comprehension:

```
# for loop
n = 5
for i in range(n):
   print(i, (i+1) % n)

>> 0 1
>> 1 2
>> 2 3
>> 3 4
>> 4 0

# list comprehension
n = 5
edges = [[i, (i+1) % n] for i in range(n)]
print(edges)
>> [[0, 1], [1, 2], [2, 3], [3, 4], [4, 0]]
```

In Sverchok the end result will be the same, but we'll arrive at the result in a different way.

The second index of each edge is one higher than the first index, except for the last edge. The last edge closes the ring of edges and meets back up with the first vertex. In essenence this is a wrap-around. Or, you can think of it as two lists, one of which is shifted by one with respect the other list.

Sverchok has a node for this called *List Shift*. We'll zip the two lists together using *List Zip* node.

- `add -> List Struct -> List Shift`
- `add -> List Main -> List Zip`

1. Hook the output of *List Range Int* into the first Data socket of the *List Zip* node.
2. Hook the output of *List Range Int* also into the *List Shift* node.
3. To make the wrap-around, simply set the *Shift slider* to 1.
4. connect the output of *List Shift* to the second Data input of *List Zip*.
5. Make sure the level parameter on *List Zip* is set to 1.
6. Hook up a Stethoscope to the output of *List Zip* to verify

Notice in this image I have minimized/hidden (shortcut H) a few nodes to keep the node view from getting claustrophobic.

7. Or hook up the output of *List Zip* straight into the EdgPol socket of'Viewer Draw'.

End of lesson 02

// – todo

Addendum

`Viewer Draw` automatically generates Edges when you pass one or more Vertices and Polygons. This means in practice when you already have the Polygons for an object then you don't need to also pass in the Edges, they are inferred purely from the indices of the incoming Polygons.

Introduction to modular components

prerequisites

Same as lesson 01.

Status: WIP

Lesson 03 - A Grid

Grids are another common geometric primitive. A Grid can be thought of as a Plane subdivided over its *x* and *y* axes. Sverchok's *Plane* generator makes grids (including edges and polygons), but we will combine the elementary nodes to build one from scratch. Doing this will cover several important concepts of parametric design, and practical tips for the construction of dynamic topology.

What do we know about Grids?

For simplicity let's take a subdivided *Plane* as our template. We know it's flat and therefore the 3rd dimension (z) will be constant. We can either have a uniformly subdivided Plane or allow for x and y to be divided separately. A separate XY division is a little bit more interesting, let's go with that.

Where to start?

For non trivial objects we often use a notepad (yes, actual paper – or blender Greasepencil etc) to draw a simplified version of what we want to accomplish. On a drawing we can easily name and point to properties, see relationships, and even solve problems in advance.

I chose a Grid because it has only a few properties: *X Side Length, Y Side Length, X num subdivs, Y num subdivs*. These properies can be exposed in several ways. You could expose the number of divisions (edges) per side, or the amount of vertices per side. 1 geometric property, but two ways of looking at it.

Decide which variables you want to expose

The upside of building generators from scratch is that you can make decisions based on what is most convenient for you. Here I'll pick what I think is the most convenient, it can always be changed later.

- Division distance side X
- Division distance side Y
- Number Vertices on side X
- Number Vertices on side Y

Think in Whole numbers (ints) if you can

What I mean by this is, reduce the problem to something that is mathematically uncomplicated. Here's a grid drawn on an xy graph to illustrate the coordinates. The z-dimension could be ignored but it's included for completeness.

The reason I pick 4 verts for the X axis and 3 for Y, is because that's the smallest useful set of vertices we can use as a reference. The reason i'm not picking 3*3 or 4*4 is because using different vertex counts makes it clear what that *X* axis might have some relation to 4 and to *Y* to 3.

If you consider the sequence just by looking at the first index of each vertex, it goes `[0,1,2,3,0,1,2,3,0,1,2,3]`. We can generate sequences like that easily. When we look at the second index of these vertices that sequence is `[0,0,0,0,1,1,1,1,2,2,2,2]`, this also is easy to generate.

Using 'modulo' and 'integer division' to get grid coordinates

I hope you know Python, or at the very least what *% (modulo)* and *// (int div)* are. The sequences above can be generated using code this way – If this code doesn't make sense keep reading, it's explained further down:

```
# variables
x = 4
y = 3
j = x * y           # 12

# using for loop
final_list = []
for i in range(j):
   x = i % 4        # makes: 0 1 2 3 0 1 2 3 0 1 2 3
   y = i // 4       # makes: 0 0 0 0 1 1 1 1 2 2 2 2
   z = 0
   final_list.append((x, y, z))

print(final_list)
'''
>> [(0, 0, 0), (1, 0, 0), (2, 0, 0), (3, 0, 0),
>>  (0, 1, 0), (1, 1, 0), (2, 1, 0), (3, 1, 0),
>>  (0, 2, 0), (1, 2, 0), (2, 2, 0), (3, 2, 0)]
'''

# using list comprehension
final_list = [(i%4, i//4, 0) for i in range(j)]
```

With any luck you aren't lost by all this code, visual programming is very similar except with less typing. The plumbing of an algorithm is still the same whether you are clicking and dragging nodes to create a flow of information or writing code in a text editor.

Operands

We introduced the Math node in lesson 01 and 02, the Math node (from the Number menu) has many operations called operands. We'll focus on these to get the vertex components.

Operand	Symbol	Behaviour
Modulo (mod)	%	`i % 4` returns the division remainder of `i / 4`, rounded down to the nearest whole number
Integer Division	//	`i // 4` returns the result of `i / 4`, rounded down to the nearest whole number.

We can use:

- `i % 4` to turn `[0,1,2,3,4,5,6,7,8,9,10,11]` into `[0,1,2,3,0,1,2,3,0,1,2,3]`
- `i // 4` to turn `[0,1,2,3,4,5,6,7,8,9,10,11]` into `[0,0,0,0,1,1,1,1,2,2,2,2]`

Making vertices

A recipe which you should be able to hook up yourself by seeing the example image.

- `Add -> Vector -> Vector In`
- `Add -> Number -> Math` (3x) notice I minimized the Multiplication Node.
- `Add -> Number -> Integer` (2x)
- `Add -> Number -> Range Int`

We multiply `y=3` by `x=4` to get `12` this is the number of vertices. This parameter determines the length of the range `[0,1..11]` (12 vertices, remember we start counting indices at 0).

With all nodes hooked up correctly you can hook `Vector In`'s output to the *vertices* socket of a ViewerDraw node to display the vertices. To test if it works you can use the sliders on the two Integer nodes to see the grid of vertices respond to the two parameters. Remember to put these sliders back to 3 and 4 (as displayed in the image), to continue to the next step.

Making Polygons

This might be obvious to some, so this is directed at those who've never done this kind of thing before. This is where we use a notepad to write out the indexlist for the 6 polygons (two rows of 3 polygons, is the result of a x=4, y=3 grid). Viewing the vertices from above, go clockwise. The order in which you populate the the list of polygons is determined by what you find more convenient.

For my example, I think of the X axis as the Columns, and I go from left to right and upwards

Notice that between polygon index 2 and 3 there is a break in the pattern. The polygon with vertex indices `[3,7, 8,4]` doesn't exist (for a grid of x=4, y=3), if we did make that polygon it would connect one Row to the next like so:

We know how many polygons we need (let's call this number `j`), it is useful to think of an algorithm that produces these index sequences based on a range from `0 thru j-1` or `[0,1,2,3,4,5]`. We can first ignore the fact that we need to remove every n-th polygon, or avoid creating it in the first place. Whatever you decide will be a choice between convenience and efficiency - I will choose convenience here.

A polygon Algorithm

> Sverchok lets you create complex geometry without writing a single line of code, but you will not get the most out of the system by avidly avoiding code. Imagine living a lifetime without ever taking a left turn at a corner, you would miss out on faster more convenient ways to reach your destination.

It's easier for me to explain how an algorithm works, and give you something to test it with, by showing the algorithm as a program, a bit of Python. Programming languages allow you to see without ambiguity how something works by running the code.

WIP - NOT ELEGANT

this generates faces from a vertex count for x,y:

```python
ny = 3
nx = 4

faces = []
add_face = faces.append

total_range = ((ny-1) * (nx))
for i in range(total_range):
    if not ((i+1) % nx == 0):  # +1 is the shift
        add_face([i, i+nx, i+nx+1, i+1])  # clockwise

print(faces)
```

This is that same algorithm using the elementary nodes, can you see the similarity?

// – TODO

CHAPTER 4

Nodes

Generators

3 Point Arc

Functionality

Given a *start coordinate*, a *through coordinate*, and an *end coordinate* this Node will find the Arc that passes through those points.

Inputs

- arc_pts input is *[begin, mid, end, begin, mid, end, begin, mid, end.....]*
  - must be (len % 3 == 0)
- num verts is either
  - constant
  - unique
  - or repeats last value if the number of arcs exceeds the number of values in the *num_vert* list

Parameters

The UI is quite minimal.

- **num verts** can be changed via Slider input on the UI or as described above, it can be fed multiple values through the input sockets.

Output

- (verts, edges) : A set of each of these that correspond with a packet of commands like 'start, through, end, num_verts'
- verts needs to be connected to get output
- edges is optional

Examples

See the progress of how this node came to life here (gifs, screenshots)

2 Point Spline

Functionality

Single section Bezier Spline. Creates a *Spline Curve* from 2 sets of points. Analogue to the Blender native Curve object, but limited to 2 pairs of *knots* and *control points* per curve.

Inputs

Parameter	Type	Description
num verts	int	per curve this sets how many verts define the curve
knot 1	Vector	These place and adjust the shape of the curve. The knots are vectors on the curve, the controls are vectors to which the curve is mathematically attracted
control 1	Vector	
control 2	Vector	
knot 2	Vector	

The node accepts these The node will adjust to make sure the length of

Parameters

The Node is vectorized in the following way. If any of the *knots* or *control points* are given in a list that doesn't match the length of the other lists, then the last value of that shorter list is repeated to match the length of the longest.

This means; if *knot1, control1, knot2* are length 3, 4 and 8 and control2 is length 20 then *knot1, control1, knot2* will all get their last value repeated till the full list matches 20 values.

The same *filling* procedure is applied to the *Num Verts* parameter.

Outputs

- (verts, edges) : A set of each of these that correspond with a packet of commands like 'knot1, ctrl1, ctrl2, knot2'
- verts needs to be connected to get output
- edges is optional

optionals for visualizing the curve handles

- hnd. Verts
- hnd. Edges

Passing hnd.Verts and hnd.Edges to a ViewerDraw node helps visualize the Handles that operate on your Spline curve.

Examples

See the progress of how this node came to life here (gifs, screenshots)

Box

Functionality

Offers a Box primitive with variable X,Y and Z divisions, and overal Size.

Inputs

All inputs are expected to be scalar values. Divisions are given in *Integers* only, it will cast incoming *floats* to *int*.

- Size
- Div X
- Div Y
- Div Z

Parameters

None

Outputs

- Verts
- Edges
- Faces

Examples

None other than variables given by *Inputs*.

Notes

This is not a very fast implementation of Box, but it can act as an introduction to anyone interested in coding their own Sverchok Nodes. It is a very short node code-wise, it has a nice structure and shows how one migh construct a bmesh and use existing bmesh.ops to operate on it.

Bricks Grid

destination after Beta: Generators

Functionality

This node generates bricks-like grid, i.e. a grid each row of which is shifted with relation to another. It is also possible to specify toothing, so it will be like engaged bricks. All parameters of bricks can be randomized with separate control over randomization of each parameter.

Inputs & Parameters

All parameters except for `Faces mode` can be given by the node or an external input. All inputs are vectorized and they will accept single or multiple values.

Param	Type	Default	Description
Faces mode	Flat or Stitch or Center	Flat	What kind of polygons to generate: • Flat - generate one polygon (n-gon, in general) for each brick. • Stitch - split each brick into several triangles, with edges going across brick. • Center - split each brick into triangles by adding new vertex in the center of the brick.
Unit width	Float	2.0	Width of one unit (brick).
Unit height	Float	1.0	Height of one unit (brick).
Width	Float	10.0	Width of overall grid.
Height	Float	10.0	Height of overall grid.
Toothing	Float	0.0	Bricks toothing amount. Default value of zero means no toothing.
Toothing random	Float	0.0	Toothing randomization factor. Default value of zero means that all toothings will be equal. Maximal value of 1.0 means toothing will be random in range from zero to value of `Toothing` parameter.
Random U	Float	0.0	Randomization amplitude along bricks rows. Default value of zero means all bricks will be of same width.
Random V	Float	0.0	Randomization amplitude across bricks rows. Default value of zero means all grid rows will be of same height.
Shift	Float	0.5	Brick rows shift factor. Default value of 0.5 means each row of bricks will be shifted by half of brick width in relation to previous row. Minimum value of zero means no shift.
Seed	Int	0	Random seed.

Outputs

This node has the following outputs:

- **Vertices**
- **Edges**. Note that this output will contain only edges that are between bricks, not that splitting bricks into triangles.
- **Polygons**
- **Centers**. Centers of bricks.

Examples of usage

Default settings:

The same with Stitch faces mode:

The same with Centers faces mode:

Using `toothing` parameter together with randomization, it is possible to generate something like stone wall:

A honeycomb structure:

Wooden floor:

You can also find some more examples in the development thread.

Circle

Functionality

Circle generator creates circles based on the radius and the number of vertices. What does that mean? It means that if the number of vertices is too low, ir will stop being a circle and will be a regular polygon, in example:

```
- 3 vertices = triangle.
- 4 vertices = square
- ...
- 6 vertices =  hexagon
- ...
- Many vertices =  circle
```

This node will also create sector or semgent of circles using the **Degrees** option. See the examples below to see it working also with the **mode** option.

Inputs

All inputs are vectorized and they will accept single or multiple values. There is three inputs:

- **Radius**
- **N Vertices**
- **Degrees**

Same as other generators, all inputs will accept a single number, an array or even an array of arrays:

```
[2]
[2, 4, 6]
[[2], [4]]
```

Parameters

All parameters except **Mode** can be given by the node or an external input.

Param	Type	Default	Description
Radius	Float	1.00	radius of the circle
N Vertices	Int	24	number of vertices to generate the circle
Degrees	Float	360.0	angle for a sector/segment circle
Mode	Bollean	False	switch between two sector and segment circle

Outputs

Vertices, **Edges** and **Polygons**. All outputs will be generated. Depending on the type of the inputs, the node will generate only one or multiples independant circles. In example:

As you can see in the red rounded values, depending on how many inputs have the node, will be generated those same number of outputs.

If **Degrees** is minor than 0, depending of the **mode** state, will be generated a sector or a segment of a circle with that degrees angle.

Example of usage

In this first example we see that circle generator can be a circle but also any regular polygon that you want.

The second example shows the use of **mode** option and how it generates sector or segment of a circle based on the **degrees** value.

Cylinder

Functionality

Cylinder generator, as well as circle, is used to create a big variety of polyhedra based on the cyliner form: two polygons connected by a body. In the examples will see some possibilities.

Inputs

All inputs are vectorized and they will accept single or multiple values. There is three inputs:

- **Radius Top**
- **Radius Bottom**
- **Vertices**
- **Height**
- **Subdivisions**

Parameters

All parameters except **Separate** and **Caps** can be given by the node or an external input.

Param	Type	Default	Description
Radius Top	Float	1.00	radius of the top polygon
Radius Bottom	Float	1.00	radius of the bottom polygon
Vertices	Int	32	number of vertices to generate top and bottom poygons
Height	Float	2.00	height of the cylinder
Subdivisions	Int	0	number of the height subdivisions
Separate	Bollean	False	grouping vertices by V direction
Caps	Bollean	True	turn on and off top and bottom cap

Outputs

Vertices, **Edges** and **Polygons**. All outputs will be generated. Depending on the type of the inputs, the node will generate only one or multiples independant cylinders. If **Separate** is True, the only the top and the bottom polygons will be generated. With **Caps** with can enable or disable the top and bottom caps.

Example of usage

In this example with can see some examples of what can be done with this node.

Image

Line

Functionality

Line generator creates a series of connected segments based on the number of vertices and the length between them. Just a standard subdivided line along X axis

Inputs

N Verts and **Step** are vectorized. They will accept single or multiple values. Both inputs will accept a single number or an array of them. It also will work an array of arrays:

```
[2]
[2, 4, 6]
[[2], [4]]
```

Parameters

All parameters except **Center** can be given by the node or an external input.

Param	Type	Default	Description
N Verts	Int	2	number of vertices. The minimum is 2
Step	Float	1.00	length between vertices
Center	Boolean	False	center line around 0

Outputs

Vertices and **Edges**. Verts and Edges will be generated. Depending on the inputs, the node will generate only one or multiples independant lines. See examples below.

Example of usage

The first example shows just an standard line with 6 vertices and 1.20 ud between them

In this example the step is given by a series of numbers:

```
[0.5, 1.0 , 1.5, 2.0, 2.5]
```

NGon

destination after Beta: generators

Functionality

NGon generator creates regular (or not exactly, see below) polygons of given radius with given number of sides. As an example, it can create triangles, squares, hexagons and so on. In this sence, it is similar to Circle node.

Location of vertices can be randomized, with separate control of randomization along radius and randomization of angle. See the examples below.

Each vertex can be connected by edge to next vertex (and produce usual polygon), or some number of vertices can be skipped, to produce star-like polygons. In the last case, you most probably will want to pass output of this node to Intersect Edges node.

Inputs

All inputs are vectorized and they will accept single or multiple values. This node has the following inputs:

- **Radius**
- **N Sides**
- **RandomR**
- **RandomPhi**
- **Seed**
- **Shift**

Same as other generators, all inputs will accept a single number, an array or even an array of arrays:

```
[2]
[2, 4, 6]
[[2], [4]]
```

Parameters

All parameters can be given by the node or an external input.

Param	Type	Default	Description
Radius	Float	1.00	Radius of escribed circle. When `RandomR` is zero, then all vertices will be at this distance from origin.
N Sides	Int	5	Number of sides of polygon to generate. With higher values and `Shift` = 0, `RandomR` = 0, `RandomPhi` = 0, you will get the same output as from Circle node.
RandomR	Float	0.0	Amplitude of randomization of vertices along radius.
RandomPhi	Float	0.0	Amplitude of randomizaiton of angles. In radians.
Seed	Float	0.0	Random seed. Affects output only when `RandomR` != 0 or `RandomPhi` != 0.
Shift	Int	0	Also known as “star factor”. When this is zero, each vertex is connected by edge to next one, and you will get usual polygon. Otherwise, n’th vertex will be connected to (n+shift+1)’th. In this case, you will get sort of star.

Outputs

This node has the following outputs:

- **Vertices**
- **Edges**
- **Polygons**

If `Shift` input is not zero, then `Polygons` output will be empty - this node does not create degenerated polygons.

Depending on the type of the inputs, the node will generate only one or multiples independant circles.

Examples

Sides=5, Shift=0, RandomR=0, RandomPhi=0 (default values):

Sides=6, RandomPhi=0.3:

Sides=6, RandomR=0.3:

Sides=7, Shift=1, RandomR=0.24, RandomPhi=0.15:

Sides=29, Shift=9, RandomR=0, RandomPhi=0:

Plane

Functionality

Plane generator creates a grid in the plane XY, based on the number of vertices and the length between them in X and Y directions. It works in a similar way than Line, but creating a grid instead of a line.

Inputs

Just like in Line Node, all inputs are vectorized and they will accept single or multiple values. There is two basic inputs **N Vert** and **Step**, but referenced to both X and Y directions, so it results in 4 inputs:

- **N Vert X**
- **N Vert Y**
- **Step X**
- **Step Y**

Same as Line, all inputs will accept a single number or an array of them or even an array of arrays:

```
[2]
[2, 4, 6]
[[2], [4]]
```

Parameters

All parameters except **Separate** and **Center** can be given by the node or an external input.

Param	Type	Default	Description
N Vert X	Int	2	number of vertices in X. The minimum is 2.
N Vert Y	Int	2	number of vertices in X. The minimum is 2.
Step X	Float	1.00	length between vertices in X axis
Step Y	Float	1.00	length between vertices in Y axis
Separate	Boolean	False	grouping vertices by V direction
Center	Boolean	False	center plane around 0

Outputs

Vertices, **Edges** and **Polygons**. All outputs will be generated. Depending on the type of the inputs, the node will generate only one or multiples independant grids. If **Separate** is True, the output is totally different. The grid disappear (no more **polygons** are generated) and instead it generates a series of lines repeated along Y axis. See examples below to a better understanding.

Example of usage

The first example shows a grid with 6 vertices in X direction and 4 in Y. The distance between them is base on the next serie of floats:

```
[0.5, 1.0 , 1.5, 2.0, 2.5]
```

The second example is just like the first, but with **Separate** option activated, so the output is a series of lines unconnected instead of a complete grid.

Random Vector

Functionality

Produces a list of random unit vectors from a seed value.

Inputs & Parameters

Parameters	Description
Count	Number of random vectors numbers to spit out
Seed	Accepts float values, they are hashed into *Integers* internally.
Scale	Scales vertices on some value *Floats*.

Outputs

A list of random unit vectors, or nested lists.

Examples

Notes

Seed is applied per output, not for the whole operation (Should this be changed?) A unit vector has length of 1, a convex hull of random unit vectors will approximate a sphere with radius off 1.

Examples

SNLite Docs

There's no rst for this at present, please read the preliminary (but comprehensive) docs on :

https://github.com/nortikin/sverchok/issues/942

Sphere

Functionality

Sphere generator will create a sphere based on its Radius and de U and V subdivisions.

Inputs

All inputs are vectorized and they will accept single or multiple values. There is three inputs:

- **Radius**
- **U**
- **V**

Parameters

All parameters except **Separate** and **Caps** can be given by the node or an external input.

Param	Type	Default	Description
Radius	Float	1.00	radius of the sphere
U	Int	24	U subdivisions
V	Int	24	V subdivisions
Separate	Bolean	False	Grouping vertices by V direction

Outputs

Vertices, **Edges** and **Polygons**. All outputs will be generated. Depending on the type of the inputs, the node will generate only one or multiples independant spheres.

Example of usage

As you can see, lot of different outputs can be generated with this node.

Torus

Functionality

Torus generator will create a torus based on its radii sets, number of sections and section phases.

Inputs

All inputs are vectorized and they will accept single or multiple values.

- **Major Radius** [1]
- **Minor Radius** [1]
- **Exterior Radius** [2]
- **Interior Radius** [2]
- **Revolution Sections**
- **Spin Sections**
- **Revolution Phase**
- **Spin Phase**

Notes: [1] : Major/Minor radii are available when Major/Minor mode is elected. [2] : Exterior/Interior radii are available when Exterior/Interior mode is elected.

Parameters

The MODE parameter allows to switch between Major/Minor and Exterior/Interior radii values. The input socket values for the two radii are interpreted as such based on the current mode.

All parameters except **mode** and **Separate** can be given by the node or an external input.

Param	Type	Default	Description
Major Radius	Float	1.00	Major radius of the torus [1]
Minor Radius	Float	0.25	Minor radius of the torus [1]
Exterior Radius	Float	1.25	Exterior radius of the torus [2]
Interior Radius	Float	0.75	Interior radius of the torus [2]
Revolution Sections	Int	32	Number of sections around torus center
Spin Sections	Int	16	Number of sections around torus tube
Revolution Phase	Float	0.00	Phase revolution sections by a radian amount
Spin Phase	Float	0.00	Phase spin sections by a radian amount
Separate	Bolean	False	Grouping vertices by V direction

Notes: [1] : Major/Minor radii are available when Major/Minor mode is elected. [2] : Exterior/Interior radii are available when Exterior/Interior mode is elected.

Outputs

Vertices, **Edges**, **Polygons** and **Normals** All outputs will be generated when connected.

Example of usage

Torus Knot

Functionality

Torus Knot generator will create a torus knot based on its radii sets, curve resolution and phases.

Inputs

All inputs are vectorized and they will accept single or multiple values.

- **Major Radius** [1]
- **Minor Radius** [1]
- **Exterior Radius** [2]
- **Interior Radius** [2]
- **Curve Resolution**
- **Revolution Phase**
- **Spin Phase**

Notes: [1] : Major/Minor radii are available when Major/Minor mode is elected. [2] : Exterior/Interior radii are available when Exterior/Interior mode is elected.

Parameters

The MODE parameter allows to switch between Major/Minor and Exterior/Interior radii values. The input socket values for the two radii are interpreted as such based on the current mode.

All parameters except **mode** can be given by the node or an external input.

Param	Type	Default	Description
Major Radius	Float	1.00	Major radius of the torus [1]
Minor Radius	Float	0.25	Minor radius of the torus [1]
Exterior Radius	Float	1.25	Exterior radius of the torus [2]
Interior Radius	Float	0.75	Interior radius of the torus [2]
Curve Resolution	Int	100	Number of vertices in a curve (per link
Revolution Phase	Float	0.00	Phase revolution vertices by a radian amount
Spin Phase	Float	0.00	Phase spin vertices by a radian amount

Notes: [1] : Major/Minor radii are available when Major/Minor mode is elected. [2] : Exterior/Interior radii are available when Exterior/Interior mode is elected.

Extra Parameters

A set of extra parameters are available on the property panel. These parameters do not receive external input.

Extra Param	Type	Default	Description
Adaptive Resolution	Bool	False	Adjusts curve resolution dynamically
Multiple Links	Bool	True	Generate multiple links in a degenerate knot
Flip p	Bool	False	Flip REVOLUTION direction (P)
Flip q	Bool	False	Flip SPIN direction (Q)
P multiplier	Int	1	Multiplies the p count [1]
Q multiplier	Int	1	Multiplies the q count [1]
Height	Float	1.00	Scales the vertices along Z by this amount
Scale	Float	1.00	Scales both radii by this amount

Notes: [1] Used without adaptive resolution these allow to create aliased torus knots resulting in all sorts of interesting shaped knots.

Outputs

Vertices, **Edges** and **Normals** All outputs will be generated when connected.

Example of usage

Formula Shape

Usage

Generators Extended

Rounded box

Functionality

See the BlenderArtists thread by original author Phymec. This node merely encapsulates the code into a form that works for Sverchok. Internally the main driver is the amount of input vectors, each vector represents the x y z dimensions of a box. Each box can have unique settings. If fewer parameters are provided than sizes, then a default or the last parameter is repeated.

Inputs & Parameters

name	type	info
radius	single value or list	radius of corner fillets
arc div	single value or list	number of divisions in the fillet
lin div	single value or list	number of internal divisions on straight parts (`[0..1]` or `[1..20]`)
Vector Size	single vector or list	x y z dimensions for each box
div type	3way switch, integers	just corners, corners and edges, all
odd axis align	0..1 on or off	internal rejiggery, not sure.

Outputs

Depending on how many objects the input asks for, you get a Verts and Polygons list of rounded box representations.

Examples

Notes

see:

Round Cube, real Quadsphere, Capsule (snipped thread title):

original thread

Generative Art

destination after Beta: Generators

Functionality

This node can be used to produce recursive three dimensional structures following a design specified in a separate xml file. These structures are similar to fractals or lsystems.

The xml file defines a set of transformation rules and instructions to place objects. A simple set of rules can often generate surprising and complex structures.

Inputs & Parameters

This node has the following parameters:

- **xml file** - Required. This specifies the LSystem design and should be a linked text block in the .blend file.
- **r seed** - Integer to initialize python's random number generator. If the design includes a choice of multiple rules, changing this will change the appearance of the design
- **max mats** - To avoid long delays or lock ups the output of the node is limited to this number of matrices

This node has the following inputs:

- **Vertices** - Optional. A list of vertices to be joined in a ring and used as the basis for a tube structure. Typically the output of a Circle or NGon node.
- **data** - Optional. The xml file can have optional variables defined using {myvar} type format notation. Extra named data inputs are generated for each of these these variables. These variables can be used to control animations.

Outputs

- **Vertices, Edges and Faces** - If the Vertices input is connected, these outputs will define the mesh of a tube that skins the structure defined in the xml file.
- **Shapes Matrices** - For each *shape* atribute defined in the xml file a named output will be generated. This output is a list list of matrices that define the structure.

Examples of usage

A simplified description of the algorithm for the evaluation of a design.

The xml file (see below for examples and descriptions) consist of a set of rules, each rule has a list of instructions, each instruction defines a transform and either a call to a rule or an instruction to place an instance.

The system is implemented by a stack where each item in the stack consists of the next rule to call, the current depth of the system and the current state of the system. At each iteration of the processor the last item is removed from the stack and processed.

Each instruction in the rule removed from the stack is processed in turn. The current state of this system is set to that of the item removed from the stack. Any transform in the instruction is applied to the system state. If the instruction is a call to a rule, a new item is added to the stack with the new rule, the depth increased by one, and the new system state. If the instruction is to place an instance, the matrix representing the new system state is added to the output matrix list for that type of shape. The processor then proceeds to what is now the last item on the stack.

If the max_depth for the current rule is reached or the max_depth for overall design is reached then the processor goes back and processes what is now the last item on the stack without taking any other action. If the stack is empty or the maximum number of matrices has been reached the processor stops.

A simple example of an xml design file:

6 Spirals

```
<rules max_depth="150">
        <rule name="entry">
            <call count="3" transforms="rz 120" rule="R1"/>
            <call count="3" transforms="rz 120" rule="R2"/>
        </rule>
        <rule name="R1">
            <call transforms="tx 2.6 rx 3.14 rz 12 ry 6 sa 0.97" rule="R1"/>
            <instance  transforms="sa 2.6" shape="box"/>
        </rule>
        <rule name="R2">
            <call transforms="tx -2.6 rz 12 ry 6 sa 0.97" rule="R2"/>
            <instance transforms="sx 2.6" shape="box"/>
        </rule>
</rules>
```

This specifies the following design with 6 spirals.

The xml file consists of a list of rules. There must be at least one rule called entry. This is the starting point for the processor. Each rule consists of a list of instructions. These instructions can either be a call to another rule or an instruction to place an instance of an object.

Calls can be recursive. For the example above the first instruction in rule R1 also calls rule R1. This recursion stops when the max_depth value is reached or the max_mats value set in the node is reached. The max_depth can also be set separately for each rule and is added as an attribute eg `<rule name="R1" max_depth="10">`.

Each of these instructions can be modified with a set of transforms. If the transform is omitted it defaults to the identity transform.

A transform consist of translations, rotations and scaling operations. For example `tx 1.3` means translated 1.3 units in the `x` direction, `rz 6` means rotate 6 degrees about the `z` axis and `sa 0.99` means scale all axes by 0.99.

The full list of transforms that take one argument : `tx ty tz rx ry rz sx sy sz sa` In addition all three axes values for either a translation or scale can be applied at once with a triplet of values. For example: `t 1.1 2.2 3.3 s 0.9 0.9 0.7`

Instead of using a single *transform* attribute, each transform can be specified individually. For example `transforms="tx 1 rz 90 sa 0.75"` can be replaced with `tx="1" rz="90" sa="0.75"`.

The count attribute specifies how many times that instruction is repeated. if count is omitted it defaults to 1. For example the instruction `<call count="3" transforms="rz 120" rule="R1"/>` calls rule `R1` applying a 120 degree rotation about `z` in between each call.

An instance instruction tells the processor to add a matrix to the output list defining the state of the system at that point. The names used in the shape attribute are used as the names for the node's output sockets. If there is more than one type of shape each will have its own output socket.

Multiple Rule Definition Example

There can be multiple definitions of the same rule in an xml file.

For example

Tree

```
<rules max_depth="100">
    <rule name="entry">
        <call  rule="spiral"/>
    </rule>
    <rule name="spiral" weight="100">
        <call transforms="tz 0.1 rx 1 sa 0.995" rule="spiral"/>
        <instance transforms="s 0.1 0.1 0.15" shape="tubey"/>
    </rule>
    <rule name="spiral" weight="100">
        <call transforms="tz 0.1 rx 1 ry 4 sa 0.995" rule="spiral"/>
        <instance transforms="s 0.1 0.1 0.15" shape="tubey"/>
    </rule>
    <rule name="spiral" weight="100">
        <call transforms="tz 0.1 rx 1 rz -4 sa 0.995" rule="spiral"/>
        <instance transforms="s 0.1 0.1 0.15" shape="tubey"/>
    </rule>
    <rule name="spiral" weight="20">
        <call transforms="rx 15" rule="spiral"/>
        <call transforms="rz 180" rule="spiral"/>
    </rule>
</rules>
```

In the above xml file there are four definitions of the `spiral` rule. Each rule version has a weight attribute. The processor will call each version of the `spiral` rule in a random manner. The weight attribute will determine the probability a particular rule version is called.

The first three definitions of the `spiral` rule all place an object instance and then call the `spiral` rule with the same translation along the `z` axis and rotation about the `x` axis but different amounts of rotation about the `y` and `z` axis. The fourth definition calls the `spiral` rule twice without placing an instance. This causes the branches in the tree structure. Changing the value of the weight attribute for this rule version will change how often the tree branches. For a larger weight value, the rule gets called more often and there are more branches.

If the weight attribute is omitted each version will have equal weight. Changing the value of `r seed` in the node interface will change the generated structure for xml files with multiple rule definitions. This example had rseed = 1.

Successor Rule Example

Normally when the `max_depth` for a rule is reached that 'arm' of the structure is finished. If a rule defines a successor rule then this rule will be called when the `max_depth` is reached. In the following example when the `y180` rule gets called it will be called 90 times in succession and produce a 180 degree turn about the y axis. When it finishes the successor rule `r` will be called and either produce a 180 degree turn about the `y` axis or the `z` axis.

Nouveau variation

```
<rules max_depth="1000">
    <rule name="entry">
        <call count="2" transforms="rz 60" rule="r"/>
    </rule>
    <rule name="r"><call rule="y180"/></rule>
    <rule name="r"><call rule="z180"/></rule>
    <rule name="y180" max_depth="90" successor="r">
        <call rule="dbox"/>
        <call transforms="ry -2 tx 0.1 sa 0.996" rule="y180"/>
    </rule>
    <rule name="z180" max_depth="90" successor="r">
        <call rule="dbox"/>
        <call transforms="rz 2 tx 0.1 sa 0.996" rule="z180"/>
    </rule>
    <rule name="dbox">
        <instance transforms="s 0.55 2.0 1.25 ry 90 rz 45" shape="box"/>
    </rule>
</rules>
```

This example needs "max matrices" set to 5000 to get the above result.

Mesh Mode Example

Using the matrices output allows a separate object to be placed at each location. The vertices input and the mesh (vertices, edges, faces) output "skins" the mesh into a much smaller number of objects. The vertices input should be a list of vertices such as that generated by the "Circle" node or "NGon" node. It could also be a circle type object taking from the scene using the "Objects In" node. The list of vertices should be in order so they can be made into a ring with the last vertex joined to the first. That ring dosen't have to be planar.

The output will not always be one mesh. If the rule set ends one 'arm' and goes back to start another 'arm' these two sub-parts will be separate meshes. Sometimes the mesh does not turn out how you would like. This can often be fixed by changing the rule set.

Often a mesh tube will turn out flat rather than being tube like. This can usually be fixed by either rotating the vertex ring in the scene or by adding a rotation transform to the "instance" commands in the rule set.

For example change `<instance shape="s1"/>` to `<instance transforms="ry 90" shape="s1"/>`

In other cases the mesh can be connected in the wrong order.

For example the following two xml files will look the same when the matrix output is used to place objects, but have different output when they are used in mesh mode. Both sets of xml rules produce the same list of matrices just in a different order.

Fern 1

```
<rules max_depth="2000">
    <rule name="entry">
```

```
        <call  rule="curl" />
    </rule>

    <rule name="curl" max_depth="80">
        <call transforms="rx 12.5 tz 0.9 s 0.98 0.95 1.0" rule="curl"/>
        <instance shape="box"/>
        <call transforms="tx 0.1 ty -0.45 ry 40 sa 0.25" rule="curlsmall" />
    </rule>

    <rule name="curlsmall" max_depth="80">
        <call transforms="rx 25 tz 1.2 s 0.9 0.9 1.0" rule="curlsmall"/>
        <instance shape="box"/>
    </rule>

</rules>
```

Fern 2

```
<rules max_depth="2000">
    <rule name="entry">
        <call  rule="curl1" />
        <call  rule="curl2" />
    </rule>

    <rule name="curl1" max_depth="80">
        <call transforms="rx 12.5 tz 0.9 s 0.98 0.95 1.0" rule="curl1"/>
        <instance shape="box"/>
    </rule>

    <rule name="curl2" max_depth="80">
        <call transforms="rx 12.5 tz 0.9 s 0.95 0.95 1.0" rule="curl2"/>
        <call transforms="tx 0.1 ty -0.45 ry 40 sa 0.25" rule="curlsmall" />
    </rule>

    <rule name="curlsmall" max_depth="80">
        <call transforms="rx 25 tz 1.2 s 0.9 0.9 1.0" rule="curlsmall"/>
        <instance shape="box"/>
    </rule>
</rules>
```

Again these were both done with max mats set to 5000.

Constants and Variables Example

Constants and variables can be included in the xml file by replacing a numerical value with a pair of braces.

```
transforms = "tx 0.5 rx 20 sa 0.9"
```

becomes

```
transforms = "tx {x_const} rx 20 sa 0.9"
```

Constants are defined within the xml as follows:

```
<constants  x_const="0.5" />
```

Multiple constants can be defined within one element and several *constants* elements can be used as required in the xml file.

If a field name in between curly brackets is not given a value in a *constants* element then a named input socket will be added to the node. A *Float*, *Integer* or similar node input can be wired up to this input variable.

The example below uses a variable ({curl_angle}) to animate the amount of curl on the fern structure shown in the mesh mode example and two constants to fix the the value of the `tz` transform in the large curl and the scale ({sxy}) in all the curls.

Fern 3

```
<rules max_depth="2000">
    <constants zd="1.5" sxy="0.9" />
    <rule name="entry">
       <call  rule="curl1" />
       <call  rule="curl2" />
    </rule>

    <rule name="curl1" max_depth="60">
        <call transforms="rx {curl_angle} tz {zd} s {sxy} {sxy} 1.0" rule="curl1"/>
        <instance shape="box"/>
    </rule>

    <rule name="curl2" max_depth="40">
        <call transforms="rx {curl_angle} tz {zd} s {sxy} {sxy} 1.0" rule="curl2"/>
        <call transforms="tx 0.1 ty -0.45 ry 40 sa 0.25" rule="curlsmall" />
    </rule>

    <rule name="curlsmall" max_depth="40">
        <call transforms="rx 2*{curl_angle} tz 2.7 s {sxy} {sxy} 1.0" rule="curlsmall
↪"/>
        <instance shape="box"/>
    </rule>
</rules>
```

For this animation the index number of the current frame in the animation is translated from the range 1 to 250 to the range 16 to 6 via the “Map Range” node and wired into the `curl_angle` input of the “Generative Art” node. This cause the fern to unwind as the animation proceeds.

Simple maths can also be use in the transforms definition. This has been used above in the `curlsmall` rule. The `rx` rotation of the transform will always be twice that of the `rx` rotation in the `curl1` and `curl2` rules. There cannot be any spaces in any maths expressions for the rotation, translation or scale parameters when using a single transforms attribute string. To allow for more complicated expressions each transform can be separated out into its own attribute.

transforms as single attribute (no spaces allowed in maths expression)

```
<call transforms="tx 1 rz -1*{v1} ry {v2}" rule="R1"/>
```

each transform with its own attribute (can have spaces)

```
<call tx="1" rz="-1 * {v1}" ry="{v2}" rule="R1"/>
```

All this is implemented by first using python’s string `format` method to substitute in the variable value from the node data input. Then the resulting string is passed to python’s `eval()` function. The string must evaluate to a single number (float or integer). Using `eval()` is a potential security problem as in theory someone could put some malicious code inside an xml lsystem definition. As always don’t run code from a source you don’t trust.

The python `math` and `random` modules exist in the namespace of the “Generative Art” node so for example a transform could be defined as:

```
tx="2**0.5"
```

or:

```
tx="math.sqrt(2)"
```

Only the transforms that take a single number that is `tx, ty, tz, rx, ry, rz, sx, sy, sz` and `sa` have been implemented using individual attributes. The ones that use triplets to specify all three translations or scales at once (`t` and `s`) can only be used in a transform string.

References

This node is closely based on Structure Synth but the xml design format and most of the code comes from Philip Rideout's lsystem repository on github.

Hilbert 2D

Functionality

Hilbert field generator. this is concept of dence flooding of space by continuous line, that achived with division and special knotting. Hilbert space can be only square, because of his nature.

Inputs

All inputs are not vectorized and they will accept single value. There is two inputs:

- **level**
- **size**

Parameters

All parameters can be given by the node or an external input.

Param	Type	Default	Description
level	Int	2	level of division of hilbert square
size	float	1.0	scale of hilbert mesh

Outputs

Vertices, **Edges**.

Example of usage

Smooth labirynth

Hilbert 3D

Functionality

Hilbert space generator. this is concept of dence flooding of space by continuous line, that achived with division and special knotting. Hilbert space can be only cube, because of his nature.

Inputs

All inputs are not vectorized and they will accept single value. There is two inputs:

- **level**
- **size**

Parameters

All parameters can be given by the node or an external input.

Param	Type	Default	Description
level	Int	2	level of division of hilbert square
size	float	1.0	scale of hilbert mesh

Outputs

Vertices, **Edges**.

Example of usage

Smooth labirynth

Hilbert 3D

Functionality

Hilbert image recreator. Based on hilbert space this node recreates image by interpolating it on pixels.

Inputs

- **level**
- **size**
- **sensitivity**

Parameters

All parameters can be given by the node or an external input.

Param	Type	Default	Description
RGB	float	0.3,0.59,0.11	RGB map of imported image, sensitivity to each color
image name	string	None	enumerate popup to choose image from stack
level	Int	2	level of division of hilbert square
size	float	1.0	scale of hilbert mesh
sensitivity	float	1.0	define scale of values to react and build image

Outputs

Vertices, **Edges**.

Example of usage

recreate image in hilbert

Hexa Grid

Functionality

Hexa Grid generator will create a set of staggered points suitable for creating a hexagonal grid (i.e. in conjuction with a hexagon mesh generated by the circle node).

The generated points are confined within one of the selected layouts: rectangle, triangle, diamond and hexagon.

Inputs

All inputs are vectorized and they will accept single or multiple values.

- **Level** [1]
- **NumX** [2]
- **NumY** [2]
- **Radius**

Notes: [1] : Level input is available for the triangle, diamond and hexa layout types [2] : NumX, NumY are available for the rectangular layout type

Parameters

The **Type** parameter allows to select one of the 4 layout types: RECTANGLE, TRIANGLE, DIAMOND and HEXAGON. The points will be geneated to fit within one of these layouts.

The **Center** parameters allows to center the grid around the origin.

All parameters except **Type** and **Center** can be given by the node or an external input.

All inputs are “sanitized” to restrict their values: - Level, NumX and NumY are integer with values >= 1 - Radius is float with value >= 0.0

Param	Type	Default	Description
Level	Int	3	Number of levels around the center point [1]
NumX	Int	7	Number of points along X [2]
NumY	Int	6	Number of points along Y [2]
Radius	Float	1.0	Radius of the grid tile

Notes: [1] : Level input is available for the TRIANGLE, DIAMOND AND HEXAGON layout type. [2] : NumX/NumY inputs are available for the RECTANGULAR layout type.

Outputs

Vertices Outputs will be generated when connected.

Example of usage

Image Decomposer

destination after Beta: analyzers

Functionality

To get output from this node you must connect something to the first 2 output sockets (xya and rgb), polygons is optional and only outputs faces when `Filter?` is off.

Takes the `rgba` components of an image currently loaded in Blender and decomposes them into `xya` and `rgb` Vertex-style sockets. `xy` are inferred by the number of pixels in the image and the image width. `z` doesn't make much sense in relation to a pixel and was replaced by the Alpha channel of the pixel (`a`).

If you don't have images loaded in the UV editor, they can be imported from N panel into Blender and loaded from there.

Inputs & Parameters

name	function
Skip n pixels	allows to sample a reduced grid of the image, every nth pixel in either direction.
xy_spread	the `xy` component of the `xya` socket can be multiplied to get a wider spread.
z_spread	this amplifies `rgb`, not `a` (which you can amplify yourself if that was needed.)
Filter?	uses a restricted eval to drop pixels using a simple typed command : example `r < 0.8 and g > 0.4` (more below)

Outputs

name	function
`xya`	the **x** and **y** of the pixel, combined with the Alpha channel. The value of **x** and **y** are multiplied by `xy_spread`.
`rgb`	each (unfiltered) pixel component is multiplied by `z_spread`
polygons	this output will generate sensible polygon index list for `xya` when pixels are unfiltered.

Examples

The development thread contains working examples of this Node used as preprocessor for game maps.

Notes

The loaded image gets a fake user automagically, tho perhaps this should be optional.

Profile Parametric Node

Profile Node implements a useful subset of the SVG path section commands. Currently the following segment types are available:

name	cmd	parameters
MoveTo	M, m	<2v coordinate>
LineTo	L, l	<2v coordinate 1> <2v coordinate 2> <2v coordinate n> [z]
CurveTo	C, c	<2v control1> <2v control2> <2v knot2> <int num_verts> <int even_spread> [z]
ArcTo	A, a	<2v rx,ry> <float rot> <int flag1> <int flag2> <2v x,y> <int num_verts> [z]
Close	X	
comment	#	must be first thing on the line, no trailing comment instructions.

```
<>   : mandatory field
[]   : optional field
2v   : two point vector `a,b`
         - no space between ,
         - no backticks
         - a and b can be
             - number literals
             - lowercase 1-character symbols for variables
int : means the value will be cast as an int even if you input float
      flags generally are 0 or 1.
z    : is optional for closing a line
X    : as a final command to close the edges (cyclic) [-1, 0]
      in addition, if the first and last vertex share coordinate space
      the last vertex is dropped and the cycle is made anyway.
#    : single line comment prefix
```

Mode 0: default behaviour, variables may be negated

```
M a,-a
L a,a -a,a -a,-a z
```

There are 2 slightly more elaborate evaluation modes:

Mode 1: Requires the use or parentheses to indicate where extra operations take place. Mode 1 is restrictive and only allows addition and subtraction

```
(a+b-c)
```

Mode 2: Also requires parentheses but allows a more liberal evaluation of operations. Allowed operations are:

```
(a*b(c/d))
```

To use Mode 2, you must enable the *extended parsing* switch in the N-panel for the Profile node.

Examples

If you have experience with SVG paths most of this will be familiar. The biggest difference is that only the LineTo command accepts many points, and we always start the profile with a M <pos>,<pos>.

```
M 0,0
L a,a b,0 c,0 d,d e,-e
```

CurveTo and ArcTo only take enough parameters to complete one Curve or Arc, unlike real SVG commands which take a whole sequence of chained CurveTo or ArcTo commands. The decision to keep it at one segment type per line is mainly to preserve readability.

The CurveTo and ArcTo segment types allow you to specify how many vertices are used to generate the segment. SVG doesn't let you specify such things, but it makes sense to allow it for the creation of geometry.

the fun bit about this is that all these variables / components can be dynamic

```
M 0,0
L 0,3 2,3 2,4
C 2,5 2,5 3,5 10 0
L 5,5
C 7,5 7,5 7,3 10 0
L 7,2 5,0
X
```

or

```
M a,a
L a,b c,b -c,d
C c,e c,e b,e g 0
L e,e
C f,e f,e f,-b g 0
L f,c e,a
X
```

More Info

The node started out as a thought experiment and turned into something quite useful, you can see how it evolved in the github thread

Example usage:

Gotchas

The update mechanism doesn't process inputs or anything until the following conditions are satisfied:

- Profile Node has at least one input socket connected
- The file field on the Node points to an existing Text File.

Keyboard Shortcut to refresh Profile Node

Updates made to the profile path text file are not propagated automatically to any nodes that might be reading that file. To refresh a Profile Node simply hit `Ctrl+Enter` In TextEditor while you are editing the file, or click one of the inputs or output sockets of Profile Node. There are other ways to refresh (change a value on one of the incoming nodes, or clicking the sockets of the incoming nodes)

Plane MK2

Functionality

Plane generator creates a grid in the plane XY/YZ or ZX, based on the number of vertices and the length between them in X and Y directions. It works in a similar way than Line, but creating a grid instead of a line.

Inputs

Just like in Line Node, all inputs are vectorized and they will accept single or multiple values. There is two basic inputs **N Verts** and **Step**, but referenced to both X and Y directions, so it results in 4 inputs:

- **N Verts X**
- **N Verts Y**
- **Step X**
- **Step Y**

Same as Line, all inputs will accept a single number or an array of them or even an array of arrays:

```
[2]
[2, 4, 6]
[[2], [4]]
```

Parameters

All parameters except **Separate**, **Direction**, **Center**, **Normalize**, **Size X** and **Size Y** can be given by the node or an external input.

Param	Type	Default	Description
N Verts X	Int	2	number of vertices in X. The minimum is 2.
N Verts Y	Int	2	number of vertices in X. The minimum is 2.
Step X	Float	1.00	length between vertices in X axis
Step Y	Float	1.00	length between vertices in Y axis
Separate	Boolean	False	grouping vertices by V direction
Direction	Enum	XY	generate grid in XY, YZ or ZX plane
Center	Boolean	False	center the plane around origin
Normalize	Boolean	False	normalize the plane sizes to specific values
Size X	Float	10.00	normalized plane size along X direction [1]
Size Y	Float	10.00	normalized plane size along Y direction [1]

Notes: [1] - the **Size X / Size Y** parameters are only available when the **Normalize** is on.

Outputs

Vertices, **Edges** and **Polygons**. All outputs will be generated. Depending on the type of the inputs, the node will generate only one or multiples independant grids.

If **Separate** is True, the output is totally different. The grid disappear (no more **polygons** are generated) and instead it generates a series of lines repeated along Y axis. See examples below to a better understanding.

Example of usage

Mesh Expression Node

Functionality

This node generates mesh from description in JSON format. Variables and mathematical expressions are allowed in definitions of vertex coordinates, so exact shape of mesh can be parametrized. All variables used in JSON definition become inputs of node. It is also possible to generate JSON description from existing mesh.

Usual workflow

1. Create some mesh object by using usual Blender's modelling techniques. Select that mesh.
2. Press "from selection" button in Mesh Expression node. New text buffer will appear in Blender.
3. Switch to Blender's text editor and select newly created buffer.
4. Edit defintion. You can replace any of vertex coordinates with expression enclosed in double-quotes, such as *"x+1"*. See also syntax description below.
5. Optionally, you can add "defaults" key to definition, with default values of variables.
6. In Mesh Expression node, all variables used in JSON definition will appear as inputs.

JSON syntax

For generic description of JSON, please refer to https://en.wikipedia.org/wiki/JSON.

Mesh Expression node uses JSON, which should be a dictionary with following keys:

- "vertices". This should be a list, containing 3-item lists, which are vertex coordinates. Each coordinate should be either integer or floating-point number, or a string with valid expression (see expression syntax below).
- "edges". This should be a list, containing 2-item lists of integer numbers, which are edges description in Sverchok's native format.
- "faces". This should be a list, containint lists of integer nubmers, which are mesh faces description in Sverchok's native format.
- "defaults". This should be a dictionary. Keys are variable names, and values are default variable values. Values can be only integer or floating-point numbers.

See also JSON examples below.

Expression syntax

Expressions used in place of vertex coordinates are usual Python's expressions.

For exact syntax definition, please refer to https://docs.python.org/3/reference/expressions.html.

In short, you can use usual mathematical operations (+, -, *, /, ** for power), numbers, variables, parenthesis, and function call, such as *sin(x)*.

One difference with Python's syntax is that you can call only restricted number of Python's functions. Allowed are:

- sin
- cos

- pi
- sqrt

This restriction is for security reasons. However, Python's ecosystem does not guarantee that noone can call some unsafe operations by using some sort of language-level hacks. So, please be warned that usage of this node with JSON definition obtained from unknown or untrusted source can potentially harm your system or data.

Examples of valid expressions are:

- "1.0"
- "x"
- "x+1"
- "0.75*X + 0.25*Y"
- "R * sin(phi)"

Inputs

Set of inputs for this node depends on used JSON definition. Each variable used in JSON becomes one input. If there are no variables used in JSON, then this node will have no inputs.

Parameters

This node has one parameter: **File name**. Its value should be the name of existing Blender's text buffer.

Operators

This node has one button: **from selection**. This button takes currently selected Blender's mesh object and puts it's JSON description into newly created text buffer. Name of created buffer is assigned to **File name** parameter.

Outputs

This node has the following outputs:

- **Vertices**
- **Edges**
- **Faces**

Examples of usage

Almost trivial, a plane with adjusable size:

```
{
  "faces": [
    [       0,       1,       3,       2    ]
  ],
  "edges": [
    [       0,       2    ],
    [       0,       1    ],
    [       1,       3    ],
    [       2,       3    ]
  ],
  "vertices": [
    [ "-Size",      "-Size",      0.0    ],
    [ "Size",      "-Size",      0.0    ],
    [ "-Size",      "Size",      0.0    ],
    [ "Size",      "Size",      0.0    ]
  ]
}
```

More complex example: Example JSON definition:

You can find more examples in the development thread.

Ring

Functionality

Ring generator will create a 2D ring based on its radii sets, number of sections and phase.

Inputs

All inputs are vectorized and they will accept single or multiple values.

- **Major Radius** [1]
- **Minor Radius** [1]
- **Exterior Radius** [2]
- **Interior Radius** [2]
- **Radial Sections**
- **Circular Sections**
- **Phase**

Notes: [1] : Major/Minor radii are available when Major/Minor mode is elected. [2] : Exterior/Interior radii are available when Exterior/Interior mode is elected.

Parameters

The MODE parameter allows to switch between Major/Minor and Exterior/Interior radii values. The input socket values for the two radii are interpreted as such based on the current mode.

All parameters except **Mode** and **Separate** can be given by the node or an external input.

Param	Type	Default	Description
Major Radius	Float	1.00	Major radius of the ring [1]
Minor Radius	Float	0.25	Minor radius of the ring [1]
Exterior Radius	Float	1.25	Exterior radius of the ring [2]
Interior Radius	Float	0.75	Interior radius of the ring [2]
Radial Sections	Int	32	Number of sections around the ring center
Circular Sections	Int	3	Number of sections accross the ring band
Phase	Float	0.00	Phase of the radial sections (in radians)
Separate	Bolean	False	Grouping vertices by V direction

Notes: [1] : Major/Minor radii are available when Major/Minor mode is elected. [2] : Exterior/Interior radii are available when Exterior/Interior mode is elected.

Outputs

Vertices, **Edges** and **Polygons**. All outputs will be generated when connected.

Example of usage

Scripted Node (Generator)

aka Script Node or SN. (iteration 1)

- Introduction
- Features
- Structure
- Templates
- Conveniences
- Examples
- Techniques to improve Python performance
- Limitations
- Future

Introduction

When you want to express an idea in written form and the concept is suitable for a one line Python expression then often you can use a Formula node. If you need access to imports, classes, temporary variables, and functions then you can write a script to load into ScriptNode.

ScriptNode (SN) allows you to write multi-line python programs that define the functionality of a Node, while avoiding some of the boilerplate associated with a regular Node. SN can be used as an environment for experimenting with algorithms. Scripts written for SN are easily converted to full PyNodes.

It's a prototype so bug reports are welcome.

Here's a short tutorial to SN1, see *An introduction and tutorial for the Scripted Nodes*

Features

allows:

- Loading/Reloading scripts currently in TextEditor
- imports and aliasing, ie anything you can import from console works in SN
- nested functions and lambdas
- named inputs and outputs
- named operators (buttons to action something upon button press)

Structure

At present all scripts for SN must (strict list - general):

- have 1 *sv_main* function as the main workhorse
- *sv_main* must take 1 or more arguments (even if you don't use it)
- all function arguments for `sv_main` must have defaults.
- each script shall define `in_sockets` and `out_sockets`
- *ui_operators* is an optional third output parameter

sv_main()

`sv_main()` can take int, float and list or nested list. Here are some legal examples:

```
def sv_main(vecs_in_multi=[[]], vecs_in_flat=[], some_var=1, some_ratio=1.2):
    pass

[[]]         # for nested input (lists of lists of any data type currently supported)
[]           # for flat (one list)
int, float   # for single value input
```

in_sockets

```
in_sockets = [
    [type, 'socket name on ui', input_variable],
    [type, 'socket name on ui 2', input_variable2],
    # ...
]
```

out_sockets

```
out_sockets = [
    [type, 'socket name on ui', output_variable],
    [type, 'socket name on ui 2', output_variable2],
    # ...
]
```

in_sockets and out_sockets

- Each *socket name on ui* string shall be unique.
- **type** are currently limited to

type id	type data
's'	floats, ints, edges, faces, strings
'v'	vertices, vectors, 3-tuples
'm'	matrices, 4 x 4 nested lists

ui_operators

```
ui_operators = [
    ['button_name', func1]
]
```

- Here *func1* is the function you want to call when pressing the button.
- Each *"button_name"* is the text you want to appear on the button. For simplicity it must be a unique and valid python variable name

  - with no special characters (`| () .\/...etc`)
  - doesn't start with a number
  - contains no spaces, use single underscores if you need word separation. The UI code replaces underscores with spaces.

return

Simple, only two flavours are allowed at the moment.

```
return in_sockets, out_sockets
return in_sockets, out_sockets, ui_operators
```

Templates

Sverchok includes a list of easily accessible examples and templates. They can be accessed from the SN node if nothing is loaded, or from the *Template Menu* in *TextEditor* as `sv NodeScripts`.

Conveniences

We vale our time, i'm sure you do too, so features have been added to help speed up the script creation process.

Text Editor

- has automatic `in_sockets` list creation when the key cursor is over `sv_main`. (please note: it doesn't attempt to detect if you want nested verts or edge/polygon so it assumes you want 'v')
  - kb shortcut: `Ctrl+I -> Generate in_sockets`
- can also convert a template description (like *kv lang* if you know Kivy) into valid ScriptNode ready python. Example available here
  - kb shortcut: `Ctrl+I -> Convert svlang`
- can refresh the Script Node which currently loads that script by hitting `Ctrl+Enter`

Examples

The best way to get familiarity with SN is to go through the templates folder. They are intended to be lightweight and educational, but some of them will show advanced use cases. The images and animations on this thread on github. may also provide some insight into what's possible.

A typical nodescript may look like this:

```
from math import sin, cos, radians, pi
from mathutils import Vector, Euler


def sv_main(n_petals=8, vp_petal=20, profile_radius=1.3, amp=1.0):

    in_sockets = [
        ['s', 'Num Petals',  n_petals],
        ['s', 'Verts per Petal',  vp_petal],
        ['s', 'Profile Radius', profile_radius],
        ['s', 'Amp',  amp],
    ]
    # variables
    z_float = 0.0
    n_verts = n_petals * vp_petal
    section_angle = 360.0 / n_verts
    position = (2 * (pi / (n_verts / n_petals)))

    # consumables
    Verts = []

    # makes vertex coordinates
    for i in range(n_verts):
        # difference is a function of the position on the circumference
        difference = amp * cos(i * position)
        arm = profile_radius + difference
        ampline = Vector((arm, 0.0, 0.0))

        rad_angle = radians(section_angle * i)
        myEuler = Euler((0.0, 0.0, rad_angle), 'XYZ')

        # changes the vector in place, successive calls are accumulative
        # we reset at the start of the loop.
        ampline.rotate(myEuler)
        x_float = ampline.x
        y_float = ampline.y
        Verts.append((x_float, y_float, z_float))

    # makes edge keys, ensure cyclic
    Edges = [[i, i + 1] for i in range(n_verts - 1)]
    Edges.append([i, 0])

    out_sockets = [
        ['v', 'Verts', [Verts]],
        ['s', 'Edges', [Edges]],
    ]

    return in_sockets, out_sockets
```

but we are not forced to have all code inside sv_main, we can also do:

```
def lorenz(N, verts):
    add_vert = verts.append
    h = 0.01
    a = 10.0
    b = 28.0
    c = 8.0 / 3.0

    x0 = 0.1
    y0 = 0
    z0 = 0
    for i in range(N):
        x1 = x0 + h * a * (y0 - x0)
        y1 = y0 + h * (x0 * (b - z0) - y0)
        z1 = z0 + h * (x0 * y0 - c * z0)
        x0, y0, z0 = x1, y1, z1

        add_vert((x1,y1,z1))

def sv_main(N=1000):
    verts = []
    in_sockets = [['s', 'N', N]]
    out_sockets = [['v','verts', [verts]]]

    lorenz(N, verts)
    return in_sockets, out_sockets
```

We can even define classes inside the .py file, or import from elsewhere.

Here's a *ui_operator* example, it acts like a throughput (because in and out are still needed by design). You'll notice that inside *func1* the node's input socket is accessed using *SvGetSockeyAnyType(...)*. It is probably more logical if we could access the input data directly from the variable *items_in*, currently this is not possible – therefor the solution is to use what sverchok nodes use in their internal code too. The upshot, is that this exposes you to how you might access the socket content of other nodes. Experiment :)

```
def sv_main(items_in=[[]]):

    in_sockets = [
        ['v', 'items_in', items_in]]

    def func1():
        # directly from incoming Object_in socket.
        sn = bpy.context.node

        # safe? or return early
        if not (sn.inputs and sn.inputs[0].links):
            return

        verts = SvGetSocketAnyType(sn, sn.inputs['items_in'])
        print(verts)

    out_sockets = [['v', 'Verts', items_in]]
    ui_operators = [['print_names', func1]]

    return in_sockets, out_sockets, ui_operators
```

Breakout Scripts

For lack of a better term, SN scripts written in this style let you pass variables to a script located in `/sverchok-master/..` or `/sverchok-master/your_module_name/some_library`. To keep your sverchok-master folder organized I recommend using a module folder. In the example below, I made a folder inside sverchok-master called `sv_modules` and inside that I have a file called *sv_curve_utils*, which contains a function *loft*. This way of coding requires a bit of setup work, but then you can focus purely on the algorithm inside *loft*.

```
from mathutils import Vector, Euler, Matrix
import sv_modules
from sv_modules.sv_curve_utils import loft

def sv_main(verts_p=[], edges_p=[], verts_t=[], edges_t=[]):

    in_sockets = [
        ['v', 'verts_p', verts_p],
        ['s', 'edges_p', edges_p],
        ['v', 'verts_t', verts_t],
        ['s', 'edges_t', edges_t]]
    verts_out = []

    def out_sockets():
        return [['v', 'verts_out', verts_out]]

    if not all([verts_p, edges_p, verts_t, edges_t]):
        return in_sockets, out_sockets()

    # while developing, it can be useful to uncomment this
    if 'loft' in globals():
        import imp
        imp.reload(sv_modules.sv_curve_utils)
        from sv_modules.sv_curve_utils import loft

    verts_out = loft(verts_p[0], verts_t[0])  #  this is your break-out code

    # here the call to out_sockets() will pick up verts_out
    return in_sockets, out_sockets()
```

Techniques to improve Python performance

There are many ways to speed up python code. Some slowness will be down to innefficient algorithm design, other slowness is caused purely by how much processing is minimally required to solve a problem. A decent read regarding general methods to improve python code performance can be found on python.org. If you don't know where the cycles are being consumed, then you don't know if your efforts to optimize will have any significant impact.

Read these 5 rules by Rob Pike before any optimization. http://users.ece.utexas.edu/~adnan/pike.html

Limitations

Most limitations are voided by increasing your Python and `bpy` skills.

Future

SN iteration 1 is itself a prototype and is a testing ground for iteration 2. The intention was always to provide multiple programming language interfaces, initially coffeescript because it's a lightweight language with crazy expressive capacity. iteration 2 might work a little different, perhaps working from within a class but trying to do extra introspection to reduce boilerplate.

The only reason in_sockets needs to be declared at the moment is if you want to have socket names that are different than the function arguments. It would be possible to allow *sv_main()* to take zero arguments too. So possible configurations should be:

```
sv_main()
sv_main() + in_sockets
sv_main() + out_sockets
sv_main(a=[],..)
sv_main(a=[],..) + in_sockets
sv_main(a=[],..) + out_sockets
sv_main(a=[],..) + in_socket + out_sockets
```

etc, with ui_operators optional to all combinations

That's it for now.

Scripted Node 2(Generator)

aka Script Node MK2

- Introduction
- Features
- Structure
- Templates
- Conveniences
- Examples
- Techniques to improve Python performance
- Limitations

Introduction

When you want to express an idea in written form and the concept is suitable for a one line Python expression then often you can use a Formula node. If you need access to imports, classes, temporary variables, and functions then you can write a script to load into Script Node 2.

Script Node MK2 differs from Script Node iteratrion 1 in that offers more control. It also has a prototype system where you could for example reuse the behavior of a generator and the template takes care of all the details leaving you to focus on the function. Scripts using the templates automatically becomes more powerful.

It's a prototype so bug reports, questions and feature request are very welcome.

Features

allows:

- Loading/Reloading scripts currently in TextEditor
- imports and aliasing, ie anything you can import from console works in SN2
- nested functions and lambdas
- named inputs and outputs
- named operators (buttons to action something upon button press)

Structure

At present all scripts for SN2 must:

- be subclasses SvScript
- include a function called process in the class
- have member attributes called `inputs` and `outputs`
- have one Script class per file, if more than one, last one found will be used

process(self)

`process(self)` is the main flow control function. It is called when all sockets without defaults are connected. Usually the template provides a `process` function for you.

inputs

Default can be a float or integer value, not other types are usable yet:

```
inputs = [
    [type, 'socket name on ui', default],
    [type, 'socket name on ui2', default],
    # ...
]
```

outputs

```
outputs = [
    [type, 'socket name on ui'],
    [type, 'socket name on ui 2'],
    # ...
]
```

inputs and outputs

- Each *socket name on ui* string shall be unique.
- **type** are currently limited to

type id	type data
's'	floats, ints, edges, faces, strings
'v'	vertices, vectors, 3-tuples
'm'	matrices, 4 x 4 nested lists

There are a series of names that have special meaning that scripts should avoid as class attributes or only used for the intended meaning. To be described:

`node` `draw_buttons` `update` `process` `enum_func` `inputs`

`outputs`

Templates

Sverchok includes a series of examples for the different templates.

Conveniences

We value our time, we are sure you do too, so features have been added to help speed up the script creation process.

Text Editor

- can refresh the Script Node which currently loads that script by hitting `Ctrl+Enter`

Main classes for your subclasses are:

- `SvScript`
- `SvScriptSimpleGenerator`
- `SvScriptSimpleFunction`

Limitations

Using `SvScriptSimpleGenerator` and `SvScriptSimpleFunction` you limit inputs to deal with one object. For plane, for example, you'll get next data:

> [(0.0, 0.0, 0.0), (1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (1.0, 1.0, 0.0)] [(0, 1, 3, 2)]

If you need Full support of Sverchok data - you'd better use `SvScript` class and `self.node.inputs[0].sv_get()` function.

Examples

The best way to get familiarity with Script Node 2 is to go through the templates folder. They are intended to be lightweight and educational, but some of them will show advanced use cases. The images and animations on this thread on github. may also provide some insight into what's possible.

A typical nodescript using the `SvScriptSimpleGenerator` may look like this, note that the third argument for outputs is specific to this template:

```
import numpy
import itertools

class GridGen(SvScriptSimpleGenerator):
    inputs = [("s", "Size", 10.0),
              ("s", "Subdivs", 10)]
    outputs = [("v", "verts", "make_verts"),
               ("s", "edges", "make_edges")]

    @staticmethod
    def make_verts(size, sub):
        side = numpy.linspace(-size / 2, size / 2, sub)
        return tuple((x, y, 0) for x, y in itertools.product(side, side))

    @staticmethod
    def make_edges(size, sub):
        edges = []
        for i in range(sub):
            for j in range(sub - 1):
                edges.append((sub * i + j, sub * i + j + 1))
                edges.append((sub * j + i, sub * j + i + sub))
        return edges
```

Note that here the name of the method that should be called for producing data for each socket in the final last arguments to `outputs` but we are not forced to have all code inside the class, we can also do

```
def lorenz(N, verts, h, a, b, c):
    add_vert = verts.append

    x0 = 0.1
    y0 = 0
    z0 = 0
    for i in range(N):
        x1 = x0 + h * a * (y0 - x0)
        y1 = y0 + h * (x0 * (b - z0) - y0)
        z1 = z0 + h * (x0 * y0 - c * z0)
        x0, y0, z0 = x1, y1, z1

        add_vert((x1,y1,z1))
```

```
class LorenzAttractor(SvScriptSimpleGenerator):

    inputs = [
        ['s', 'N', 1000],
        ['s', 'h', 0.01],
        ['s', 'a', 10.0],
        ['s', 'b', 28.0],
        ['s', 'c', 8.0/3.0]
    ]

    @staticmethod
    def make_verts(N, h, a, b, c):
        verts = []
        lorenz(N, verts, h, a, b, c)
        return verts

    @staticmethod
    def make_edges(N, h a, b, c:
        edges = [(i, i+1) for i in range(N-1)]
        return edges

    outputs = [
        ['v','verts', "make_verts"],
        ['s','edges', "make_edges"]
    ]
```

Here is a simple script for deleting loose vertices from mesh data, it also serves as an illustration for a type of script that uses the `` `SvScriptSimpleFunction` `` template that has one main function that decomposes into separate sockets. The methods don't have be static but in general it is good practice to keep them free from side effects.

```
from itertools import chain

class DeleteLooseVerts(SvScriptSimpleFunction):
    inputs = [
        ('v', 'verts'),
        ('s', 'pol')
        ]
    outputs = [
        ('v', 'verts'),
        ('s', 'pol')
        ]

    # delete loose verts
    @staticmethod
    def function(*args, **kwargs):
        ve, pe = args
        # find used indexes
        v_index = sorted(set(chain.from_iterable(pe)))
        # remap the vertices
        v_out = [ve[i] for i in v_index]
        # create a mapping from old to new vertices index
        mapping = dict(((j, i) for i, j in enumerate(v_index)))
        # apply mapping to input polygon index
        p_out = [tuple(map(mapping.get, p)) for p in pe]
        return v_out, p_out
```

Breakout Scripts

Scripts that needs to access the node can do so via the `` `self.node` `` variable that is automatically set.

```
class Breakout(SvScript):
    def process(self):
        pass

    def update(self):
        node = self.node
        node_group = self.node.id_data
        # here you can do anything to the node or node group
        # that real a real node could do including multisocket
        # adaptive sockets etc. templates and examples for this are
        # coming
```

Admit, you can call sockets data directly when using `` `SvScript` `` as `` `self.node.inputs[0].sv_get()` ``. And other `` `self.node.` `` operations possible from this class.

Techniques to improve Python performance

There are many ways to speed up python code. Some slowness will be down to innefficient algorithm design, other slowness is caused purely by how much processing is minimally required to solve a problem. A decent read regarding general methods to improve python code performance can be found on python.org. If you don't know where the cycles are being consumed, then you don't know if your efforts to optimize will have any significant impact.

Read these 5 rules by Rob Pike before any optimization. http://users.ece.utexas.edu/~adnan/pike.html

Limitations

Most limitations are voided by increasing your Python and `bpy` skills. But one should also realize what is approriate for a node script to do.

That's it for now.

An introduction and tutorial for the Scripted Nodes

> Dealga Mcardle | 2014 | October

In my opinion new users should avoid the Script Nodes until they understand a majority of the existing nodes and the Sverchok *Eco-system* as a concept. This suggestion applies to everyone, even competent coders.

Script Nodes are great when you want to encapsulate a behaviour which may not be easy to achieve with existing nodes alone. They are my prefered way to either 1) prototype code, or 2) write custom nodes that are too specific to be submitted as regular nodes.

At the moment Sverchok has 2 Scripted Node implementations: SN and SN2. Exactly how they differ from eachother will be shown later. Both offer *practical shorthand* ways to define what a node does, which sliders and sane defaults it might have, and what socket types can connect to it. These scripts have a minimal interface, are stored inside the *.blend* file as plain text python, and can be shared easily.

If you've ever written code for a Blender addon or script, you will be familiar with registration of classes. Nodes normally also need to be registered so Blender can find them, but Script Nodes don't because they are in essence a shell for your code – and the shell is already registered, all you have to do is write code to process input into output.

Scripted Node 1 – an informal introduction

Here is a classic 'Hello World' style example used to demonstrate graphics coding. It's called a Lorenz Attractor.

```
def lorenz(N, verts):
    add_vert = verts.append
    h = 0.01
    a = 10.0
    b = 28.0
    c = 8.0 / 3.0

    x0 = 0.1
    y0 = 0
    z0 = 0
    for i in range(N):
        x1 = x0 + h * a * (y0 - x0)
        y1 = y0 + h * (x0 * (b - z0) - y0)
        z1 = z0 + h * (x0 * y0 - c * z0)
        x0, y0, z0 = x1, y1, z1

        add_vert((x1,y1,z1))

def sv_main(N=1000):

    verts = []
    in_sockets = [['s', 'N', N]]
    out_sockets = [['v','verts', [verts]]]

    lorenz(N, verts)
    return in_sockets, out_sockets
```

Here's what this code produces.

Infact, here's the Node Interface that the script produces too

Compare the code with the image of the node and you might get a fair idea where the sockets are defined and where the default comes from. Look carefully at `in_sockets` and `out_sockets`, two of the elements are strings (socket type and socket name), and the third element is the Python variable that we automatically bind to those sockets.

Brief Guided Explanation

You've probably got a fair idea already from the example script. SN1 has a few conventions which let you quickly define sockets and defaults. What follows are short remarks about the elements that make up these scripts, aimed at someone who is about to write their first script for SN1.

Sockets

Sverchok at present has 3 main socket types: VerticesSocket, StringsSocket and MatrixSocket. Script Nodes refer to these socket types with only their first letter in lowercase. 's','v','m':

```
's' to hold: floats, ints, edges, faces, strings
'v' to hold: vertices, vectors, 3-tuples
'm' to hold: matrices, 4 x 4 nested lists
```

Socket Names

Each socket has a name. Take a minute to think about a good descriptive name for each. Socket names can always be changed later, but my advice is to use clear names from the very beginning.

Variable names

Variable names are used to expose the values of the associated socket to your script. If the socket is unconnected then the value of the variable will be taken from the specified default.

node function *(sv_main)*

The main function for SN1 is `sv_main`, in the body of this function is where we declare socket types, socket names, and variable names for input and output sockets. These are declared in two arrays `in_sockets` and `out_sockets`.

The argument list of `sv_main` is where you provide defaults values or the nestedness of an incoming datatype. (don't worry if this makes no sense, read it again later).

That's great, show me!

The easiest way to get started is to first load an existing script. Here are some steps:

- Go to *Generators / Scripted Node* and add it to the NodeView.
- Open a Blender TextEditor window so you can see the TextEditor and the NodeView at the same time.
- Paste the Lorenz Attractor script (from above) into the TextEditor and call it 'attractor.py'
- In NodeView look at the field on the second row of the Scripted Node. This is a file selector which shows all Text files in blender. When you click on it you will see "attractor.py"
- Select "attractor.py" press the button the right, the one that looks like a powersocket.
- This changes the way the Node appears. The node will now have 1 input socket and one output socket. It might even have changed to a light blue.

That's pretty much all there is to loading a script. All you do now is hook the output Verts to a Viewer Node and you'll see a classic Lorenz Attractor point set.

Study the sv_main

If you look carefully in `sv_main` there's not a lot to the whole process. `sv_main` has two **required** lists; `in_sockets` and `out_sockets`. sv_main also has a argument list which you must fill with defaults, here the only variable is N so the argument list was `sv_main(N=1000)`.

The lorenz function takes 2 arguments:

- **N**, to set the number of vertices.
- **verts**, a list-variable to store the vertices generated by the algorithm.

In this example the `verts` variable is also what will be sent to the output socket, because it says so in `out_sockets`. Notice that the lorenz function doesn't return the verts variable. All the lorenz function does is fill that list with values. Just to be clear about this example. At the time `sv_main` ends, the content of `verts` is full, but before `lorenz()` is called, `verts` is an empty list.

Here is the same lorenz attractor with more parameters exposed, see can you load it? https://github.com/nortikin/sverchok/blob/master/node_scripts/templates/zeffii/LorenzAttractor2.py

Lastly

If none of this makes sense, spend time learning about Python and dig through the `node_scripts/templates` directory.

Scripted Node 2(Generator)

aka Script Node MK2

- Introduction
- Features
- Structure
- Templates
- Conveniences
- Examples
- Techniques to improve Python performance
- Limitations

Introduction

When you want to express an idea in written form and the concept is suitable for a one line Python expression then often you can use a Formula node. If you need access to imports, classes, temporary variables, and functions then you can write a script to load into Script Node 2.

Script Node MK2 differs from Script Node iteratrion 1 in that offers more control. It also has a prototype system where you could for example reuse the behavior of a generator and the template takes care of all the details leaving you to focus on the function. Scripts using the templates automatically becomes more powerful.

It's a prototype so bug reports, questions and feature request are very welcome.

Features

allows:

- Loading/Reloading scripts currently in TextEditor
- imports and aliasing, ie anything you can import from console works in SN2
- nested functions and lambdas
- named inputs and outputs
- named operators (buttons to action something upon button press)

Structure

At present all scripts for SN2 must:

- be subclasses SvScript
- include a function called process in the class
- have member attributes called `inputs` and `outputs`

- have one Script class per file, if more than one, last one found will be used

process(self)

`process(self)` is the main flow control function. It is called when all sockets without defaults are connected. Usually the template provides a `process` function for you.

inputs

Default can be a float or integer value, not other types are usable yet:

```
inputs = [
    [type, 'socket name on ui', default],
    [type, 'socket name on ui2', default],
    # ...
]
```

outputs

```
outputs = [
    [type, 'socket name on ui'],
    [type, 'socket name on ui 2'],
    # ...
]
```

inputs and outputs

- Each *socket name on ui* string shall be unique.
- **type** are currently limited to

type id	type data
's'	floats, ints, edges, faces, strings
'v'	vertices, vectors, 3-tuples
'm'	matrices, 4 x 4 nested lists

There are a series of names that have special meaning that scripts should avoid as class attributes or only used for the intended meaning. To be described:

> `node` `draw_buttons` `update` `process` `enum_func` `inputs`

`outputs`

Templates

Sverchok includes a series of examples for the different templates.

Conveniences

We value our time, we are sure you do too, so features have been added to help speed up the script creation process.

Text Editor

- can refresh the Script Node which currently loads that script by hitting `Ctrl+Enter`

Main classes for your subclasses are:

- `SvScript`
- `SvScriptSimpleGenerator`
- `SvScriptSimpleFunction`

Limitations

Using `SvScriptSimpleGenerator` and `SvScriptSimpleFunction` you limit inputs to deal with one object. For plane, for example, you'll get next data:

> [(0.0, 0.0, 0.0), (1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (1.0, 1.0, 0.0)] [(0, 1, 3, 2)]

If you need Full support of Sverchok data - you'd better use `SvScript` class and `self.node.inputs[0].sv_get()` function.

Examples

The best way to get familiarity with Script Node 2 is to go through the templates folder. They are intended to be lightweight and educational, but some of them will show advanced use cases. The images and animations on this thread on github. may also provide some insight into what's possible.

A typical nodescript using the `SvScriptSimpleGenerator` may look like this, note that the third argument for outputs is specific to this template:

```python
import numpy
import itertools

class GridGen(SvScriptSimpleGenerator):
    inputs = [("s", "Size", 10.0),
              ("s", "Subdivs", 10)]
    outputs = [("v", "verts", "make_verts"),
               ("s", "edges", "make_edges")]

    @staticmethod
    def make_verts(size, sub):
        side = numpy.linspace(-size / 2, size / 2, sub)
        return tuple((x, y, 0) for x, y in itertools.product(side, side))

    @staticmethod
    def make_edges(size, sub):
        edges = []
        for i in range(sub):
            for j in range(sub - 1):
                edges.append((sub * i + j, sub * i + j + 1))
                edges.append((sub * j + i, sub * j + i + sub))
        return edges
```

Note that here the name of the method that should be called for producing data for each socket in the final last arguments to `outputs` but we are not forced to have all code inside the class, we can also do

```python
def lorenz(N, verts, h, a, b, c):
    add_vert = verts.append

    x0 = 0.1
    y0 = 0
    z0 = 0
    for i in range(N):
        x1 = x0 + h * a * (y0 - x0)
        y1 = y0 + h * (x0 * (b - z0) - y0)
        z1 = z0 + h * (x0 * y0 - c * z0)
        x0, y0, z0 = x1, y1, z1

        add_vert((x1,y1,z1))
```

```
class LorenzAttractor(SvScriptSimpleGenerator):

    inputs = [
        ['s', 'N', 1000],
        ['s', 'h', 0.01],
        ['s', 'a', 10.0],
        ['s', 'b', 28.0],
        ['s', 'c', 8.0/3.0]
    ]

    @staticmethod
    def make_verts(N, h, a, b, c):
        verts = []
        lorenz(N, verts, h, a, b, c)
        return verts

    @staticmethod
    def make_edges(N, h a, b, c:
        edges = [(i, i+1) for i in range(N-1)]
        return edges

    outputs = [
        ['v','verts', "make_verts"],
        ['s','edges', "make_edges"]
    ]
```

Here is a simple script for deleting loose vertices from mesh data, it also serves as an illustration for a type of script that uses the `` `SvScriptSimpleFunction` `` template that has one main function that decomposes into separate sockets. The methods don’t have be static but in general it is good practice to keep them free from side effects.

```
from itertools import chain

class DeleteLooseVerts(SvScriptSimpleFunction):
    inputs = [
        ('v', 'verts'),
        ('s', 'pol')
        ]
    outputs = [
        ('v', 'verts'),
        ('s', 'pol')
        ]

    # delete loose verts
    @staticmethod
    def function(*args, **kwargs):
        ve, pe = args
        # find used indexes
        v_index = sorted(set(chain.from_iterable(pe)))
        # remap the vertices
        v_out = [ve[i] for i in v_index]
        # create a mapping from old to new vertices index
        mapping = dict(((j, i) for i, j in enumerate(v_index)))
        # apply mapping to input polygon index
        p_out = [tuple(map(mapping.get, p)) for p in pe]
        return v_out, p_out
```

Breakout Scripts

Scripts that needs to access the node can do so via the `self.node` variable that is automatically set.

```
class Breakout(SvScript):
    def process(self):
        pass

    def update(self):
        node = self.node
        node_group = self.node.id_data
        # here you can do anything to the node or node group
        # that real a real node could do including multisocket
        # adaptive sockets etc. templates and examples for this are
        # coming
```

Admit, you can call sockets data directly when using `SvScript` as `self.node.inputs[0].sv_get()`. And other `self.node.` operations possible from this class.

Techniques to improve Python performance

There are many ways to speed up python code. Some slowness will be down to innefficient algorithm design, other slowness is caused purely by how much processing is minimally required to solve a problem. A decent read regarding general methods to improve python code performance can be found on python.org. If you don't know where the cycles are being consumed, then you don't know if your efforts to optimize will have any significant impact.

Read these 5 rules by Rob Pike before any optimization. http://users.ece.utexas.edu/~adnan/pike.html

Limitations

Most limitations are voided by increasing your Python and `bpy` skills. But one should also realize what is approriate for a node script to do.

That's it for now.

Transforms

Matrix Apply

Functionality

Applies a Transform Matrix to a list or nested lists of vectors (and therefore vertices)

Inputs

Inputs	Description
Vectors	Represents vertices or intermediate vectors used for further vector math
Matrices	One or more, never empty

Outputs

Nested list of vectors / vertices, matching the number nested incoming *matrices*.

Examples

Notes

The `update` function is outdated, functionally this is of no relevance to users but we should change it for future compatibility.

Mirror

Functionality

This node is used to make general mirroring over geometry. It works directly over vertices, not with matrixes. It offers 3 different types of mirror:

+=====================+===+ |Type of Mirror |Description | +=====================+=== |Vertex Mirror | Based on one single point | |Axis Mirror | Mirror around an axis defined by two points | |Plane Mirror | Mirror over a plane given by a matrix | +=====================+===+

Vertex Mirror

This mode let us define a center point and a make mirror of the incoming Vertices

Inputs

All inputs are vectorized and they will accept single or multiple values. There is two inputs:

- **Vertices**
- **Vert A**

Parameters

Defult value for **Vert A** is equal to `(0.0, 0.0, 0.0)`. Vertices need an input.

Param	Type	Default	Description
Vertices	Vertices	none	vertices to mirror
Vert A	Vertices	(0.0, 0.0, 0.0)	center of the mirroring

Outputs

Only **Vertices** will be generated. Depending on the type of the inputs, if they have more than one object, then more objects will be outputted.

Example of usage

In this example we use Vertex mirror to mirroring an arbitrary shape. As you can see every point goes through the center point.

Axis Mirror

This mode is used to make mirroring around an axis defined by two vertices.

Inputs

All inputs are vectorized and they will accept single or multiple values. There is three inputs:

- **Vertices**
- **Vert A**
- **Vert B**

Parameters

There is default values for **Vert A** and **vert B**. **Vertices** need an input.

Param	Type	Default	Description
Vertices	Vertices	none	vertices to mirror
Vert A	Vertices	(0.0, 0.0, 0.0)	first point to define the axis
Vert B	Vertices	(1.0, 0.0, 0.0)	second point to define the axis

Outputs

Only **Vertices** will be generated. Depending on the type of the inputs, if one or more inputs have multiple values, then more objects will be mirrored.

Example of usage

We define an axis with two vertices and use them to make a mirror. All the points reflect through the chosen axis.

Plane Mirror

This is the most common method of mirroring. We'll use a plane defined by a matrix.

Inputs

All inputs are vectorized and they will accept single or multiple values. There is two inputs:

- **Vertices**
- **Plane**

Parameters

Plane has a defult value, but Vertices need an input.

Param	Type	Default	Description
Vertices	Vertices	none	vertices to mirror
Plane	Matrix	Identity	matrix to define the mirror plane

Outputs

Only **Vertices** will be generated. Depending on the type of the inputs, if one or more planes are defined, then more objects will be mirrored.

Example of usage

In this last case we just mirror the shape over the selected plane, defined by a matrix.

Vector Move

Functionality

equivalent to a Translate Transform

Moves incoming sets of Vertex Lists by a *Vector*. The Vector is bound to a multiplier (Scalar) which amplifies all components of the Vector. The resulting Vector is added to the locations of the incoming Vertices.

You might use this to translate the center of an object away or towards from [0,0,0] in order to apply other transforms like Rotation and Scale.

Inputs & Parameters

	Description
Vertices	Vertex or Vertex Lists representing one or more objects
Vector	Vector to use for Translation, this is simple element wise addition to the Vector representations of the incoming vertices. If the input is Nested, it is possible to translate each sub-list by a different Vector.
Multiplier	Straightforward `Vector * Scalar`, amplifies each element in the Vector parameter

Outputs

A Vertex or nested Lists of Vertices

Examples

This works for one vertice or many vertices

translate back to origin

Move lists of matching nestedness. (whats that?! - elaborate)

Notes

Rotation

Functionality

This node is used to make general rotation over geometry. It works directly over vertices, not with matrixes. Just like Blender, it offers 3 different types of rotation:

Axis Rotation	Based on axis (X, Y, Z) and a rotation angle (W)

Type of Rotation	Description
Axis Rotation	Based on axis (X, Y, Z) and a rotation angle (W)
Euler Rotation	Using Euler Gimbal: 3 axis with a hierarchical relationship between them
Quaternion rotation	Based on four values (X, Y, Z, W). W value will avoid X, Y, Z rotation

If you want to learn deeply about all this types of rotation, visit this link: http://wiki.blender.org/index.php/User:Pepribal/Ref/Appendices/Rotation

Axis Rotation

This mode let us define an axis (X, y, Z), a center point and a rotation angle (W), in degrees, around the defined axis.

Inputs

All inputs are vectorized and they will accept single or multiple values. There is four inputs:

- **Vertices**
- **Center**
- **Axis**
- **Angle**

Parameters

All parameters except **Vertices** has a default value. **Angle** can be given by the node or an external input.

Param	Type	Default	Description
Vertices	Vertices	none	vertices to rotate
Center	Vertices	(0.0, 0.0, 0.0)	point to place the rotation axis
Axis	Vector	(0.0, 0.0, 1.0)	axis around which rotation will be done
Angle	Float	0.00	angle in degrees to make rotation

Outputs

Only **Vertices** will be generated. Depending on the type of the inputs, if more than one angle is set, then more objects will be outputted.

Example of usage

In this example we use axis rotation with multiple inputs in axis an angle to create a complex geometry from just one plane.

Euler Rotation

This mode is used to perform Euler rotations, refered to an Eular gimbal. A gimbal is a set of 3 axis that have a hierarchical relationship between them.

Inputs

All inputs are vectorized and they will accept single or multiple values. There is four inputs:

- **Vertices**
- **X**
- **Y**
- **Z**

Parameters

All parameters except **Vertices** has a default value. **X**, **Y** and **Z** can be given by the node or an external input.

Param	Type	Default	Description
Vertices	Vertices	none	vertices to rotate
X	Float	0.00	value to X axis rotation
Y	Float	0.00	value to Y axis rotation
Z	Float	0.00	value to Z axis rotation
Order	Enum	XYZ	order of the hierarchical relationship between axis

Outputs

Only **Vertices** will be generated. Depending on the type of the inputs, if one or more inputs have multiple values, then more objects will be outputted.

Example of usage

In the first example we use Euler rotation to perfomr a simple operation, we just rotate a plane around Z axis multiple times. The second is more complex, with multiple inputs in Y and Z to create a complex geometry from just one plane, simulating infinite loop.

Quaternion Rotation

In this mode rotation is defined by 4 velues (X, Y, Z, W), but it works in a different way than Axis Rotation. The important thing es the relation between all four values. For example, X value rotate the object around X axis up to 180 degrees. The effect of W is to avoid that rotation and leave the element with zero rotation. The final rotation is a combination of all four values.

Inputs

All inputs are vectorized and they will accept single or multiple values. There is five inputs:

- **Vertices**
- **X**
- **Y**
- **Z**
- **W**

Parameters

All parameters except **Vertices** has a default value. **X**, **Y**, **Z** and **W** can be given by the node or an external input.

Param	Type	Default	Description
Vertices	Vertices	none	vertices to rotate
X	Float	0.00	value to X axis rotation
Y	Float	0.00	value to Y axis rotation
Z	Float	0.00	value to Z axis rotation
W	Float	1.00	value to Z axis rotation

Outputs

Only **Vertices** will be generated. Depending on the type of the inputs, if one or more inputs have multiple values, then more objects will be outputted.

Example of usage

As we can see in this example, we try to rotate the plan 45 degrees and then set W with multiple values, each higher than before, but the plane is never get to rotate 180 degrees.

Scale

Functionality

This node will allow you to scale any king of geometry. It works directly with vertices, not with matrixes, so the output will be just scaled geometry from your original vertices.

Inputs

All inputs are vectorized and they will accept single or multiple values. There is three inputs:

- **Vertices**
- **Center**
- **Factor**

Parameters

All parameters except **Vertices** has a default value. **Factor** can be given by the node or an external input.

Param	Type	Default	Description
Vertices	Vertices	none	vertices to scale
Center	Vertices	(0.0, 0.0, 0.0)	point from which the scaling will be done
Factor	Float	1.0	factor of scaling

Outputs

Only **Vertices** will be generated. Depending on the type of the inputs, if more than one factor or centers are set, then more objects will be outputted. If you generate more outputs than inputs were given, then is probably that you need to use list Repeater with your edges or polygons.

Example of usage

In this example we use scale to convert a simple circle into a kind of parabola.

Analyzers

Area

Functionality

Area node is one of the analyzer type. It is used to get the area of any polygon, no matter the number of its vertices or its world position.

Inputs

Vertices and **Polygons** are needed. Both inputs need to be of the kind Vertices and Strings, respectively

Parameters

All parameters need to proceed from an external node.

Param	Type	Default	Description
Vertices	Vertices	None	vertices of the polygons
Polygons	Strings	None	polygons referenced to vertices
Count Faces	Boolean	True	output individual faces or the sum of all

Outputs

Area will be calculated only if both **Vertices** and **Polygons** inputs are linked.

Example of usage

In the example we have the inputs from a plane with 15 faces. We can use **Area** node to get the sum of all of them or the area of every face individually.

Bounding Box

Functionality

Generates a special ordered *bounding box* from incoming Vertices.

Inputs

Vertices, or a nested list of vertices that represent separate objects.

Outputs

Output	Type	Description
Vertices	Vectors	One or more sets of Bounding Box vertices.
Edges	Key Lists	One or more sets of Edges corresponding to the Vertices of the same index.
Mean	Vectors	Arithmetic averages of the incoming sets of vertices
Center	Matrix	Represents the *Center* of the bounding box; the average of its vertices

Examples

Mean: Average of incoming set of Vertices

Center: Average of the Bounding Box

Notes

GitHub issue tracker discussion about this node

Distance

Functionality

Finds distance from point to point, from matrix to matrix or between many points and one opposite.

Inputs

vertices1 and **matrix1** and **vertices2** and **matrix2**.

Parameters

Name	Type	Description
CrossOver	Boolean	for every point finds all opposite points, not one other, but many.

Outputs

distances in format [[12,13,14,15]]

Example of usage

Angles at the Edges

This node testing is in progress, so it can be found under Beta menu

Functionality

This node calculates angles at the edges of input mesh. Angles can be measured in radians or degrees.

Inputs

This node has the following inputs:

- **Vertices**.
- **Edges**. Note that this input should be connected in order for output angles to be in correct order.
- **Polygons**.

Parameters

This node has the following parameters:

Parameter	Type	Default	Description
Signed	Boolean	False	If checked, then the node will output negative values for concave edges. By default, it always outputs positive angles.
Complement	Boolean	False	Output complementary angle to one calculated by BMesh. BMesh assumes that angle between two complanar faces is zero. With this flag checked, the node will output PI (or 180) for angle between complanar faces.
Wire/Boundary value	Enum	Default	What to return as angle for wire or boundary edges. BMesh returns some angle by default for such edges, but in some cases these values do not make sense. This parameter is displayed only in N panel. **Default.** Use value returned by BMesh. **Zero.** Return zero. **Pi.** Return PI (or 180). **Pi/2.** Return PI/2 (or 90). **None.** Return None.
Angles mode	Enum	Radian	Whether to measure angles in radians or in degrees.

Outputs

This node has one output: **Angles**. The output contains calculated angles at the edges of input mesh. Angles are in the order of edges in the `Edges` input. If the `Edges` input is not connected or is empty, then angles will be in order returned by BMesh, which is, strictly speaking, random order.

Example of usage

Bevel only acute angles:

KDT Closest Verts

Functionality

For every vertex in Verts, it checks the list of Vertices in Check Verts. What it does exactly depends on the *Search Mode*. Search modes are mere examples of what is possible with Blender's KDTree module. The documentation for kdtree is found at the latest version of `mathutils.kdtree.html`.

Inputs

Mode	Input Name	type
All	Verts	vertices
	Check Verts	vertices
N	N	int, or list of ints
Radius	Radius	float, or list of floats

Verts and Check Verts do not need to be the same pool of verts, they don't even need to be the same length.

Parameters

Search Mode	Description
1	for each vertex in *Verts* return the vertex in *Check Verts* which is closest.
N	for each vertex in *Verts* return the list of N closest vertices found in Check Verts
Radius	for each vertex in *Verts* return the *vertices* of *Check Verts* that are found within radius-distance of that vertex.

Outputs

The meaning of each output differs between Modes, but essentially they are:

- Vertices coordinates
- Vertex Indices of related vertex in Check Verts
- Vertex Distance between Vertex in Verts and Check Verts

The output lists will be nested if the `Mode` allows mutiple outputs, as is the case in N and Radius Mode.

Examples

All kinds of crazy things are possible, see some examples here in the development thread

Notes

Design specs

```
'''
find(co)
    : internal function
    : < Find nearest point to co
    : > returns co, index, dist

    : inputs:
        1) Main Verts for kdtree to hold
        2) [cVert(s)] to check against
    : outputs:
        1) [Verts.co] from Main Verts that were closest
        2) [Verts.idx] from Main Verts that were closest


find_n(co, n)
```

```
    : internal function
    : > Find nearest n points to co
    : < returns iterable of (co, index, dist)

    : inputs:
        1) Main Verts for kdtree to hold
        2) [cVert(s)] to check against (size don't have to match)
        3) n, max n nearest
        optional?
        4) mask, [0, 0, 1, 0, 1]  (return 3rd and 5th closest)
        4) range clamp, [2:] (don't return first 2 closest)
    : outputs:
        for v in cVerts:
        1) ([Verts.co],..) from Main Verts closest to v.co
        2) ([Verts.idx],..) from Main Verts closest to v.co
        optional!
        3) could generate edges directly (Saves node noodle)


find_range(co, radius)
    : > Find all points within radius of co
    : < returns iterable of (co, index, dist)

    : inputs:
        1) Main Verts for kdtree to hold
        2) [cVert(s)] to check against (size don't have to match)
        3) [distance(s)] ,

    : outputs:
        options:
        1) grouped [.co for points in Main Verts in radius of v in cVert]
        2) grouped [.idx for points in Main Verts in radius of v in cVert]
        3) grouped [.dist for points in Main Verts in radius of v in cVert]

'''
```

If you need large kdtree searches and memoization or specific functionality you shall want to write your own Node to utilize the kdtree module. Part of the problem of making a *general use* node is that it becomes sub-optimal for certain tasks. On the up-side, having this node allows you to rip out the specifics and implement your own more specialized kdtree node. Recommend using a different Node name and sharing it with team Sverchok :)

KDT Closest Edges

Alias: KDTree Edges

Functionality

On each update it takes an incoming pool of Vertices and places them in a K-dimensional Tree. It will return the Edges it can make between those vertices pairs that satisfy the constraints imposed by the 4 parameters.

Inputs

- Verts, a pool of vertices to iterate through

Parameters

Parmameter	Type	Description
mindist	float	Minimum Distance to accept a pair
maxdist	float	Maximum Distance to accept a pair
maxNum	int	Max number of edges to associate with the incoming vertex
Skip	int	Skip first n found matches if possible

Outputs

- Edges, which can connect the pool of incoming Verts to eachother.

Examples

development thread has examples

Mesh Filter

destination after Beta: Analyzers

Functionality

This node sorts vertices, edges or faces of input mesh by several available criterias: boundary vs interior, convex vs concave and so on. For each criteria, it puts "good" and "bad" mesh elements to different outputs. Also mask output is available for each criteria.

Inputs

This node has the following inputs:

- **Vertices**
- **Edges**
- **Faces**

Parameters

This node has the following parameters:

- **Mode**. Which sort of mesh elements to operate on. There are three modes available: Vertices, Edges and Faces.
- **Filter**. Criteria to be used for filtering. List of criterias available depends on mode selected. See below.

Outputs

Set of outputs depends on selected mode. See description of modes below.

Modes

Vertices

The following filtering criteria are available for the `Vertices` mode:

Wire. Vertices that are not connected to any faces.

Boundary. Vertices that are connected to boundary edges.

Interior. Vertices that are not wire and are not boundary.

The following outputs are used in this mode:

- **YesVertices**. Vertices that comply to selected criteria.
- **NoVertices**. Vertices that do not comply to selected criteria.
- **VerticesMask**. Mask output for vertices. True for vertex that comly selected criteria.
- **YesEdges**. Edges that connect vertices complying to selected criteria.
- **NoEdges**. Edges that connect vertices not complying to selected criteria.
- **YesFaces**. Faces, all vertices of which comply to selected criteria.
- **NoFaces**. Faces, all vertices of which do not comply to selected criteria.

Note that since in this mode the node filters vertices, the indicies of vertices in input list are not valid for lists in `YesVertices` and `NoVertices` outputs. So in edges and faces outputs, this node takes this filtering into account. Indicies in `YesEdges` output are valid for list of vertices in `YesVertices` output, and so on.

Edges

The following filtering criteria are available for the `Edges` mode:

Wire. Edges that are not connected to any faces.

Boundary. Edges that are at the boundary of manifold part of mesh.

Interior. Edges that are manifold and are not boundary.

Convex. Edges that joins two convex faces. This criteria depends on valid face normals.

Concave. Edges that joins two concave faces. This criteria also depends on valid face normals.

Contiguous. Manifold edges between two faces with the same winding; in other words, the edges which connect faces with the same normals direction (inside or outside).

The following outputs are used in this mode:

- **YesEdges**. Edges that comply to selected criteria.
- **NoEdges**. Edges that do not comply to selected criteria.
- **Mask**. Mask output.

Faces

For this mode, only one filtering criteria is available: interior faces vs boundary faces. Boundary face is a face, any edge of which is boundary. All other faces are considered interior.

The following outputs are used in this mode:

- **Interior**. Interior faces.
- **Boundary**. Boundary faces.
- **BoundaryMask**. Mask output. It contains True for faces which are boundary.

Examples of usage

Move only boundary vertices of plane grid:

Bevel only concave edges:

Extrude only boundary faces:

Select mesh elements by location

Functionality

This node allows to select mesh elements (vertices, edges and faces) by their geometrical location, by one of supported criteria.

You can combine different criteria by applying several instances of this node and combining masks with Logic node.

Inputs

This node has the following inputs:

- **Vertices**
- **Edges**
- **Faces**
- **Direction**. Direction vector. Used in modes: **By side**, **By normal**, **By plane**, **By cylinder**. Exact meaning depends on selected mode.
- **Center**. Center or base point. Used in modes: **By sphere**, **By plane**, **By cylinder**, **By bounding box**.
- **Percent**. How many vertices to select. Used in modes: **By side**, **By normal**.
- **Radius**. Allowed distance from center, or line, or plane, to selected vertices. Used in modes: **By sphere**, **By plane**, **By cylinder**, **By bounding box**.

Parameters

This node has the following parameters:

- **Mode**. Criteria type to apply. Supported criterias are:
  - **By side**. Selects vertices that are located at one side of mesh. The side is specified by **Direction** input. So you can select "rightmost" vertices by passing (0, 0, 1) as Direction. Number of vertices to select is controlled by **Percent** input: 1% means select only "most rightmost" vertices, 99% means select "all but most leftmost". More exactly, this mode selects vertex V if *(Direction, V) >= max - Percent * (max - min)*, where *max* and *min* are maximum and minimum values of that scalar product amongst all vertices.

  - **By normal direction**. Selects faces, that have normal vectors pointing in specified **Direction**. So you can select “faces looking to right”. Number of faces to select is controlled by **Percent** input, similar to **By side** mode. More exactly, this mode selects face F if *(Direction, Normal(F)) >= max - Percent * (max - min)*, where *max* and *min* are maximum and minimum values of that scalar product amongst all vertices.
  - **By center and radius**. Selects vertices, which are within **Radius** from specified **Center**; in other words, it selects vertices that are located inside given sphere. More exactly, this mode selects vertex V if *Distance(V, Center) <= Radius*.
  - **By plane**. Selects vertices, which are within **Radius** from specified plane. Plane is specified by providing normal vector (**Direction** input) and a point, belonging to that plane (**Center** input). For example, if you specify Direction = (0, 0, 1) and Center = (0, 0, 0), the plane will by OXY. More exactly, this mode selects vertex V if *Distance(V, Plane) <= Radius*.
  - **By cylinder**. Selects vertices, which are within **Radius** from specified straight line. Line is specified by providing directing vector (**Direction** input) and a point, belonging to that line (**Center** input). For example, if you specify Direction = (0, 0, 1) and Center = (0, 0, 0), the line will by Z axis. More exactly, this mode selects vertex V if *Distance(V, Line) <= Radius*.
  - **By edge direction**. Selects edges, which are nearly parallel to specified **Direction** vector. Note that this mode considers edges as non-directed; as a result, you can change sign of all coordinates of **Direction** and it will not affect output. More exactly, this mode selects edge E if *Abs(Cos(Angle(E, Direction))) >= max - Percent * (max - min)*, where max and min are maximum and minimum values of that cosine.
  - **Normal pointing outside**. Selects faces, that have normal vectors pointing outside from specified **Center**. So you can select “faces looking outside”. Number of faces to select is controlled by **Percent** input. More exactly, this mode selects face F if *Angle(Center(F) - Center, Normal(F)) >= max - Percent * (max - min)*, where max and min are maximum and minimum values of that angle.
  - **By bounding box**. Selects vertices, that are within bounding box defined by points passed into **Center** input. **Radius** is interpreted as tolerance limit. For examples:
    - If one point *(0, 0, 0)* is passed, and Radius = 1, then the node will select all vertices that have *-1 <= X <= 1*, *-1 <= Y <= 1*, *-1 <= Z <= 1*.
    - If points *(0, 0, 0)*, *(1, 2, 3)* are passed, and Radius = 0.5, then the node will select all vertices that have *-0.5 <= X <= 1.5*, *-0.5 <= Y <= 2.5*, *-0.5 <= Z <= 3.5*.
- **Include partial selection**. Not available in **By normal** mode. All other modes select vertices first. This parameter controls either we need to select edges and faces that have **any** of vertices selected (Include partial = True), or only edges and faces that have **all** vertices selected (Include partial = False).

Outputs

This node has the following outputs:

- **VerticesMask**. Mask for selected vertices.
- **EdgesMask**. Mask for selected edges. Please note that this mask relates to list of vertices provided at node input, not list of vertices selected by this node.
- **FacesMask**. Mask for selected faces. Please note that this mask relates to list of vertices provided at node input, not list of vertices selected by this node.

Examples of usage

Select rightmost vertices:

Select faces looking to right:

Select vertices within sphere:

Select vertices near OYZ plane:

Select vertices near vertical line:

Bevel only edges that are parallel to Z axis:

Select faces that are looking outside:

Select faces by bounding box:

Proportional Edit Falloff

Functionality

This node implements Blender's concept of "proportional edit mode" in Sverchok. It converts vertex selection mask into selection coefficients. Vertices selected by mask get the coefficient of 1.0. Vertices that are farther than specified radius from selection, get the coefficient of 0.0.

Supported falloff modes are basically the same as Blender's.

Inputs

This node has the following inputs:

- **Vertices**
- **Mask**
- **Radius**. Proportional edit radius.

Parameters

This node has the following parameters:

- **Falloff type**. Proportional edit falloff type. Supported values are:
  - Smooth
  - Sharp
  - Root
  - Linear
  - Sphere
  - Inverse Square
  - Constant

  The meaning of values is all the same as for standard Blender's proportional edit mode.
- **Radius**. Proportional edit radius. This parameter can be also provided by input.

Outputs

This node has one output: **Coeffs**. It contains one real value for each input vertex. All values are between 0.0 and 1.0.

Examples of usage

Drag a circle on one side of the box, with Smooth falloff:

All the same, but with Const falloff:

Example of usage with Extrude Separate node:

Vertex normal

Alias - vector normal

Functionality

Vertex normal node finds normals of vectors

Inputs

Vertices and **Polygons** are needed. Both inputs need to be of the kind Vertices and Strings, respectively

Parameters

All parameters need to proceed from an external node.

Param	Type	Default	Description
Vertices	Vertices	None	vertices of the polygons
Polygons	Strings	None	polygons referenced to vertices

Outputs

Vertices normals will be calculated only if both **Vertices** and **Polygons** inputs are linked.

Example of usage

Calculate Normals

Functionality

This node calculates normals for faces and edges of given mesh. Normals can be calculated even for meshes without faces, i.e. curves.

Inputs

This node has the following inputs:

- **Vertices**
- **Edges**
- **Polygons**

Outputs

This node has the following outputs:

- **FaceNormals**. Normals of faces. This output will be empty if **Polygons** input is empty.
- **VertexNormals**. Normals of vertices.

Examples of usage

Move each face of cube along its normal:

Visualization of vertex normals for bezier curve:

Normals can be also calculated for closed curves:

Centers Polygons

Functionality

Analizing geometry and finding centers of polygons, normals (from global zero), normals from local centers of polygons and matrices that find polygons rotation. Not works with edges.

Inputs

Vertices and **Polygons** from object that we analizing.

Outputs

Normals is normals from global zero coordinates, vector. **Norm_abs** is normals shifted to centers of polygons. **Origins** centers of polygons. **Centers** matrices that has rotation and location of polygons.

Example of usage

Problems

The code of matrix rotation based on Euler rotation, so when you rotate to plane X-oriented, it makes wrong. We need spherical coordinates and quaternion rotation here, needed help or something

Centers Polygons

Functionality

Analizing geometry and finding centers of polygons, normals (from global zero), normals from local centers of polygons and matrices that find polygons rotation. Not works with edges.

Inputs

Vertices and **Polygons** from object that we analizing.

Outputs

Normals is normals from global zero coordinates, vector. **Norm_abs** is normals shifted to centers of polygons. **Origins** centers of polygons. **Centers** matrices that has rotation and location of polygons.

Example of usage

Problems

The code of matrix rotation based on Euler rotation, so when you rotate to plane X-oriented, it makes wrong. We need spherical coordinates and quaternion rotation here, needed help or something

Raycast

Functionality

Functionality is almost completely analogous to the two built-in blender operators `bpy.context.scene.ray_cast` and `object.ray_cast`. Ray is casted from "start" vector to "end" vector and can hit polygons of mesh objects.

see docs: bpy.types.Object.ray_cast and bpy.types.Scene.ray_cast

Input sockets

Start - "start" vectors

End - "end" vectors

Parameters

parameter	description
object name	Name of object to analize. (For **object_space** mode only)
raycast modes	In **object_space** mode: node works like `bpy.types.Object.ray_cast` (origin of object- center of coordinate for Start & End). In **world_space** mode: node works like `bpy.types.Scene.ray_cast`.

Output sockets

socket name	description
Hitp	Hit location for every raycast
Hitnorm	Normal of hit polygon (in "object_space" mode-local coordinates, in "world_space"- global
Index/succes	For **object_space** mode: index of hit polygon. For **world_space** mode: `True` if ray hit mesh object, otherwise `False`.
data object	`bpy.data.objects[hit object]` or `None` type if ray doesn't hit a mesh object. (only in "world_space" mode)
hit object matrix	Matrix of hit/struck object. (only in "world_space" mode)

Usage

Volume

Alias: Volume

Functionality

Count Volume of every object. Output list of values

Inputs

- Vers vertices of object(s)
- Pols polygons of object(s)

Outputs

- Volume, corresponding to count of objects it outputs volumes in list.

Examples

Modifier Change

Bevel

destination after Beta: Modifier Change

Functionality

This node applies Bevel operator to the input mesh. You can specify edges to be beveled.

Inputs

This node has the following inputs:

- **Vertices**
- **Edges**
- **Polygons**
- **BevelEdges**. Edges to be beveled. If this input is not connected, then by default all edges will be beveled. This parameter is used only when `Vertex only` flag is not checked.
- **Amount**. Amount to offset beveled edge.
- **Segments**. Number of segments in bevel.
- **Profile**. Profile shape.

Parameters

This node has the following parameters:

Parameter	Type	Default	Description
Amount type	Offset or Width or Depth or Percent	Offset	• Offset - Amount is offset of new edges from original. • Width - Amount is width of new face. • Depth - Amount is perpendicular distance from original edge to bevel face. • Percent - Amount is percent of adjacent edge length.
Vertex only	Bool	False	Only bevel edges, not faces.
Amount	Float	0.0	Amount to offset beveled edge. Exact interpretation of this parameter depends on `Amount type` parameter. Default value of zero means do not bevel. This parameter can also be specified via corresponding input.
Segments	Int	1	Number of segments in bevel. This parameter can also be specified via corresponding input.
Profile	Float	0.5	Profile shape - a float nubmer from 0 to 1; default value of 0.5 means round shape. This parameter can also be specified via corresponding input.

Outputs

This node has the following outputs:

- **Vertices**
- **Edges**
- **Polygons**
- **NewPolys** - only bevel faces.

Examples of usage

Beveled cube:

Only two edges of cube beveled:

Another sort of cage:

You can work with multiple objects:

Delete Loose

Functionality

Delete Loose vertices, that not belong to edges or polygons that plugged in.

Inputs

- **Vertices**
- **PolyEdge**

Outputs

- **Vertices** - filtered
- **PolyEdge**

Examples of usage

Simple:

Intersect Edges

Functionality

The code is straight out of TinyCAD plugin's XALL operator, which is part of Blender Contrib distributions.

It operates on Edge based geometry only and will create new vertices on all intersections of the given geometry. This node goes through a recursive process (divide and conquer) and its speed is directly proportional to the number of intersecting edges passed into it. The algorithm is not optimized for large edges counts, but tends to work well in most cases.

implementation note

An Edge that touches the vertex of another edge is not considered an *Intersection* in the current implementation. *Touching* might be included as an intersection type in the future via an extra toggle in the Properties Panel.

Inputs

Verts and Edges only. Warning: Does not support faces, or Vectorized (nested lists) input

Parameters

Currently no parameters, but in the future could include a tolerance parameter and a setting to consider Touching Verts-to-Edge as an Intersection.

Outputs

Vertices and Edges, the mesh does not preserve any old vertex or edge index ordering due to the Hashing algorithm used for fast intersection lookups.

Examples

See the progress of how this node came to life here (gifs, screenshots)

Extrude edges

destination after Beta: Modifier Change

Functionality

You can extrude edges along matrices. Every matrix influence on separate vertex of initial mesh.

Inputs

This node has the following inputs:

- **Vertices**
- **Edgs/Pols**
- **Matrices**

Parameters

Nope

Outputs

This node has the following outputs:

- **Vertices**
- **Edges**
- **Polygons**
- **NewVertices** - only new vertices
- **NewEdges** - only new edges
- **NewPolys** - only new faces.

Examples of usage

Extruded circle in Z direction by sinus, drived by pi*N:

Extruded circle in XY directions by sinus and cosinus drived by pi*N:

Matrix input node can make skew in one or another direction:

Matrix input node can also scale extruded edges, so you will get bell:

Extrude Edges

This node testing is in progress, so it can be found under Beta menu

Functionality

This node applies Extrude operator to edges of input mesh. After that, matrix transformation can be applied to new vertices. It is possible to provide specific transformation matrix for each of extruded vertices.

Inputs

This node has the following inputs:

- **Vertices**
- **Edges**
- **Polygons**
- **ExtrudeEdges**. Edges of input mesh that are to be extruded. If this input is empty or not connected, then by default all edges will be processed.
- **Matrices**. Transformation matrices to be applied to extruded vertices. This input can contain separate matrix for each vertex. In simplest case, it can contain one matrix to be applied to all vertices.

Outputs

This node has the following outputs:

- **Vertices**
- **Edges**
- **Polygons**. All faces of resulting mesh.
- **NewVertices**. Newly created vertices only.
- **NewEdges**. Newly created edges only.
- **NewFaces**. Newly created faces only.

Examples of usage

Extrude only boundary edges of plane grid, along Z axis:

Extrude all edges of bitted circle, and scale new vertices:

Extrude Separate Faces

destination after Beta: Modifier Change

Functionality

This node applies Extrude operator to each of input faces separately. After that, resulting faces can be scaled up or down by specified factor. It is possible to provide specific extrude and scaling factors for each face.

Inputs

This node has the following inputs:

- **Vertices**
- **Edges**
- **Polygons**
- **Mask**. List of boolean or integer flags. Zero means do not process face with corresponding index. If this input is not connected, then by default all faces will be processed.
- **Height**. Extrude factor.
- **Scale**. Scaling factor.

Parameters

This node has the following parameters:

Parameter	Type	Default	Description
Height	Float	0.0	Extrude factor as a portion of face normal length. Default value of zero means do not extrude. Negative value means extrude to the opposite direction. This parameter can be also provided via corresponding input.
Scale	Float	1.0	Scale factor. Default value of 1 means do not scale.

Outputs

This node has the following outputs:

- **Vertices**
- **Edges**
- **Polygons**. All faces of resulting mesh.
- **ExtrudedPolys**. Only extruded faces of resulting mesh.
- **OtherPolys**. All other faces of resulting mesh.

Example of usage

Extruded faces of sphere, extruding factor depending on Z coordinate of face:

Sort of cage:

Extrude Region

Functionality

This node applies Extrude operator to the region of selected faces, as whole. After that, resulting faces can be either transformed by any matrix, or moved along normal and scaled. If transformation is specified by matrix, it is possible to provide specific matrix for each vertex.

Inputs

This node has the following inputs:

- **Vertices**
- **Edges**
- **Polygons**
- **Mask**. List of boolean or integer flags. Zero means do not process face with corresponding index. If this input is not connected, then by default all faces will be processed.
- **Height**. Extrude factor. Available only in **Along normal** mode.
- **Scale**. Scaling factor. Available only in **Along normal** mode.
- **Matrix**. Transformation matrices. Available only in **Matrix** mode.

Parameters

This node has the following parameters:

- **Transformation mode**. Controls how the transformation of extruded vertices is specified. There are two modes available:
  - **Matrix**. This is the default mode. Transformation is specified by matrix provided at **Matrix** input.
  - **Along normal**. Vertices are translated along normal and scaled. Please note, that by *normal* here we mean *average of normals of selected faces*. Scaling center is average center of selected faces.
- **Multiple extrude**. This parameter defines how to deal with multiple matrices passed into **Matrix** input or multiple values passed into **Height** and **Scale** inputs. This parameter is available only in **Matrix** mode; in **Along normal** mode, this parameter is always checked.
  - If not checked (and **Matrix** mode is used), then each matrix provided will be applied to corresponding extruded vertex. So number of matrices in input is expected to be from 1 to the number of vertices which are extruded.
  - If checked, or **Along normal** mode is used, then extrusion operation may be performed several times:
    - In **Along normal** mode, extrusion operation will be performed one time for each pair of **Height** and **Scale** input values.
    - In **Matrix** mode, extrusion operation will be performed one time for each matrix passed into **Matrix** input.
- **Keep original**. If checked, the original geometry will be passed to output as well as extruded geometry. This parameter is visible only in **Properties** (N) panel.
- **Height**. Available only in **Along normal** mode. Extrude factor as a portion of face normal length. Default value of zero means do not extrude. Negative value means extrude to the opposite direction. This parameter can be also provided via corresponding input.

- **Scale**. Available only in **Along normal** mode. Scale factor. Default value of 1 means do not scale. This parameter can be also provided via corresponding input.

Outputs

This node has the following outputs:

- **Vertices**. All vertices of resulting mesh.
- **Edges**. All edges of resulting mesh.
- **Polygons**. All faces of resulting mesh.
- **NewVerts**. Only newly created vertices.
- **NewEdges**. Only newly created edges.
- **NewFaces**. Only newly created faces.

Note 1: Indicies in **NewEdges**, **NewFaces** outputs relate to vertices in **Vertices** output, not to **NewVerts** ones.

Note 2: If multiple extrusion is used, then **NewVerts**, **NewEdges**, **NewFaces** outputs will contain only geometry created by *last* extrusion operation.

Examples of usage

Extrude along normal:

Extrude by scale matrix:

Multiple extrusion mode:

Fill Holes

Functionality

It fills closed countors from edges that own minimum vertices-sides with polygons.

Inputs

- **Vertices**
- **Edges**

Parameters

Param	Type	Default	Description
Sides	Float	4	Number of sides that will be collapsed to polygon.

Outputs

- **Vertices**
- **Edges**
- **Polygons**. All faces of resulting mesh.

Examples of usage

Fill holes of formula shape, edges of initial shape + voronoi grid + fill holes

Iterate

Functionality

This node iteratively applies affine transformation (specified by matrix) to input vertices, edges and polygons. So, given matrix `M` and vertex `V`, it will produce vertices `V`, `M*V`, `M*M*V`, `M*M*M*V` and so on.

If several matrices are presented on input, then on each iteration this node will apply *all* these matrices to input vertices. So, if 1 set of vertices and N matrices are passed, then on first iteration it will produce N sets of vertices, on second iteration - N*N more, and so on.

Note 1. Source set of vertices (edges, and faces) is always passed to output as-is. With minimal number of iterations, which is zero, this node will just copy input to output.

Note 2. Due to recursive nature of this node, with bigger iterations number and a several input matrices it can produce *a lot of* data. For example, if you pass 100 vertices, 10 matrices and specify number of iterations = 4, then it will produce 100 + 10*100 + 10*10*100 + 10*10*10*100 + 10*10*10*10*100 = 1111100 vertices.

Note 3. This node always produce one mesh. To split it to parts, use Separate Loose Parts node.

Inputs

This node has the following inputs:

- **Matrix**
- **Verices**
- **Edges**. Must be either empty (or not connected) or presenting number of edges sets, which is equal to number of vertices sets in `Vertices` input.
- **Polygons**. Must be either empty (or not connected) or presenting number of polygons sets, which is equal to number of vertices sets in `Vertices` input.
- **Iterations**. Can be used to pass value of Iterations parameter. If series of values is passed, then first value will be used for first set of vertices, second for second set of vertices, and so on.

Parameters

This node has one parameter: **Iterations**. This parameter can also be defined via corresponding input slot. This parameter defines a number of iterations to perform. Minimal value of zero means do not any iterations and just pass input to output as is.

Outputs

This node has the following outputs:

- **Vertices**
- **Edges**
- **Polygons**

- **Matrices**. Matrices that are applied to generated copies of source mesh.

If `Edges` or `Polygons` input is not connected, then corresponding output will be empty.

Examples

Circle as input, Iterations = 3; one matrix specifies scale by 0.65 (along all axis) and translation along X axis by 0.3:

One object as input, Iterations = 4; one matrix specifies scale by 0.6 along X and Y axis, and translation along Z by 1:

One Box as input, Iteration = 3, two matrices:

Iterate cubes along with pentagons:

Mesh Join

Functionality

Analogue to `Ctrl+J` in the 3dview of Blender. Separate nested lists of *vertices* and *polygons/edges* are merged. The keys in the Edge and Polygon lists are incremented to coincide with the newly created vertex list.

The inner workings go something like:

```
vertices_obj_1 = [
    (0.2, 1.5, 0.1), (1.2, 0.5, 0.1), (1.2, 1.5, 0.1),
    (0.2, 2.5, 5.1), (0.2, 0.5, 2.1), (0.2, 2.5, 0.1)]

vertices_obj_2 = [
    (0.2, 1.4, 0.1), (1.2, 0.2, 0.3), (1.2, 4.5, 4.1),
    (0.2, 1.5, 3.4), (5.2, 6.5, 2.1), (0.2, 5.5, 2.1)]

key_list_1 = [[0,1,2],[3,4,5]]
key_list_2 = [[0,1,2],[3,4,5]]

verts_nested = [vertices_obj_1, vertices_obj_2]
keys_nested = [key_list_1, key_list_2]

def mesh_join(verts_nested, keys_nested):

    mega_vertex_list = []
    mega_key_list = []

    def adjust_indices(klist, offset):
        return [[i+offset for i in keys] for keys in klist]
        # for every key in klist, add offset
        # return result

    for vert_list, key_list in zip(verts_nested, keys_nested):
        adjusted_key_list = adjust_indices(key_list, len(mega_vertex_list))
        mega_vertex_list.extend(vert_list)
        mega_key_list.extend(adjusted_key_list)

    return mega_vertex_list, mega_key_list

print(mesh_join(verts_nested, keys_nested))

# result
[(0.2, 1.5, 0.1), (1.2, 0.5, 0.1), (1.2, 1.5, 0.1),
(0.2, 2.5, 5.1), (0.2, 0.5, 2.1), (0.2, 2.5, 0.1),
(0.2, 1.4, 0.1), (1.2, 0.2, 0.3), (1.2, 4.5, 4.1),
(0.2, 1.5, 3.4), (5.2, 6.5, 2.1), (0.2, 5.5, 2.1)]

[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]
```

Inputs & Outputs

The inputs and outputs are *vertices* and *polygons / edges*.

Expects a nested collection of vertex lists. Each nested list represents an object which can itself have many vertices and key lists.

Examples

Notes

Separate Loose Parts

Functionality

Split a mesh into unconnected parts in a pure topological operation.

Input & Output

socket	name	Description
input	Vertices	Inputs vertices
input	Poly Edge	Polygon or Edge data
output	Vertices	Vertices for each mesh part
output	Poly Edge	Corresponding mesh data

Examples

Notes

Note that it doesn't take double vertices into account. There is no guarantee about the order of the outputs

Duplicate along Edge

destination after Beta: Modifier Change

Functionality

This node creates an array of copies of one (donor) mesh and aligns it along given recipient segment (edge). Count of objects in array can be specified by user or detected automatically, based on size of donor mesh and length of recipient edge. Donor mesh can be scaled automatically to fill all length of recipient edge.

Donor objects are rotated so that specified axis of object is aligned to recipient edge.

This node also can output transformation matrices, which should be applied to donor object to be aligned along recipient edge. By default, this node already applies that matrices to donor object; but you can turn this off, and apply matrices to donor object in another node, or apply them to different objects.

Inputs

This node has the following inputs:

- **Vertices**. Vertices of the donor mesh. The node will produce nothing if this input is not connected.
- **Edges**. Edges of the donor mesh.
- **Polygons**. Faces of the donor mesh.
- **Vertex1**. First vertex of recipient edge. This input is used only when “Fixed” input mode is used (see description of `Input mode` parameter below).
- **Vertex2**. Second vertex of recipient edge. This input is used only when “Fixed” input mode is used.
- **VerticesR**. Vertices of the recipient mesh. This input is used only when “Edges” input mode is used.
- **EdgesR**. Edges of the recipient mesh. These edges will be actually used as recipient edges. This input is used only when “Edges” input mode is used.
- **Count**. Number of objects in array. This input is used only in “Count” scaling mode (see description of `Scale mode` parameter below).
- **Padding**. Portion of the recipient edge length that should be left empty from both sides. Default value of zero means fill whole available length.

Parameters

This node has the following parameters:

Parameter	Type	Default	Description
Scaling mode	Count or Up or Down or Off	Count	• Count: specify number of objects in array. Objects scale will be calculated so that copies will fill length of recipient edge. • Up: count is determined automatically from length of recipient edge and size of donor mesh, and meshes are scaled only up (for example, if donor mesh is 1 unit long, and recipient edge is 3.6 units, then there will be 3 meshes scaled to be 1.2 units long each). • Down: the same as Up, but meshes are scaled only down. • Off: the same as Up, but meshes are not scaled, so there will be some empty space between copies.
Orientation	X or Y or Z	X	Which axis of donor object should be aligned to direction of the recipient edge.
Input mode	Edges or Fixed	Edges	• Edges: recipient edges will be determined as all edges from the `EdgesR` input between vertices from `VerticesR` input. • Fixed: recipient edge will be determied as an edge between the edge from `Vertex1` input and the vertex from `Vertex2` input.
Scale all axes	Bool	False	If False, then donor object will be scaled only along axis is aligned with recipient edge direction. If True, objects will be

Outputs

This node has the following outputs:

- **Vertices**
- **Edges**
- **Polygons**
- **Matrices**. Matrices that should be applied to created objects to align them along recipient edge. By default, this node already applies these matrices, so you do not need to do it second time.

This node will output something only when `Vertices` or `Matrices` output is connected.

Examples of usage

Cylinders duplicated along the segment between two specified points:

Spheres duplicated along the edges of Box:

You can also find more examples and some discussion in the development thread.

Offset

destination after Beta: Modifier Change

Functionality

Make offset for polygons with bevel in corners. Output inner and outer polygons separately.

Inputs

This node has the following inputs:

- **Vers** - vertices of objects
- **Pols** - polygons of objects
- **offset** - offset values. Vectorized for every polygon as [[f,f,f,f,f]]
- **nsides** - number of rounded sides
- **radius** - bevel radius. Vectorized for every polygon as [[f,f,f,f,f]]

Parameters

All parameters can be given by the node or an external input. `offset` and `radius` are vectorized and they will accept single or multiple values.

Param	Type	Default	Description
offset	Float	0.04	offset values.
nsides	Integer	1	number of rounded sides.
radius	Float	0.04	bevel radius.

Outputs

This node has the following outputs:

- **Vers**
- **Edgs**
- **OutPols** - get polygons that lay in outer polygon's line.
- **InPols** - get polygons that lay in inner polygon's line.

Examples of usage

Offset and radius are defined by distance between point and polygon's center, divided by some number:

Parameters' cases, that make different polygons (decomposer list node used to separate):

Upper image can be defined by one offset and list (range) of numbers, plugget to offset/radius, wich are vectorised:

Polygon Boom

Functionality

The vertices of each polygon will be placed into separate lists. If polygons share vertices then the coordinates are duplicates into new vertices. The end result will be a nested list of polygons with each their own unique vertices. This facilitates rotation of a polygon around an arbitrary points without affecting the vertices of other polygons in the list.

Inputs & Outputs

Lists of Vertices and Edge/Polygon lists. The type of data in the *edg_pol* output socket content depends on the what kind of input is passed to *edge_pol* input socket. If you input edges only, that's what the output will be.

Examples

The Box on default settings is a Cube with 6 polygons and each vertex is shared by three polygons. Polygon Boom separates the polygons into seperate coordinate lists (vertices).

Polygons to Edges

Functionality

Each polygon is defined by a closed chain of vertices which form the edges of the polygon. The edges of each polygon can be extracted. If a polygon is defined by a list of vertex indices (keys) as `[3,5,11,23]` then automatically the edge keys can be inferred as `[[3,5],[5,11],[11,23],[23,3]]`. Note here that the last key closes the edge loop and reconnects with the first key in the sequence.

Input & Output

socket	name	Description
input	pols	Polygons
output	edges	The edges from which the polygon is built

Examples

Notes

If you feed this node geometry and don't get the expected output, try a subset of the input geometry and hook the output up to a *debug node*. Seeing what the output really is helps get an understanding for how this Node has interpreted the data. Also view the incoming data to see if it's what you think it is, perhaps it has unexpected extra nesting or wrapping.

Doesn't currently work on Plane Generator, or any generator which expresses key lists using *tuples*.

Randomize

destination after Beta: Modifier Change

Functionality

This mode processes set of vertices by moving each of them by random distance along X, Y, and Z axis. You can specify maximum distance of moving for each axis.

Inputs

This node has the following inputs:

- **Vertices**
- **X amplitude**
- **Y amplitude**
- **Z amplitude**
- **Seed**

Parameters

All parameters can be given by the node or an external input. This node has the following parameters:

Parameter	Type	Default	Description
X amplitude	Float	0.0	Maximum distance to move vertices along X axis.
Y amplitude	Float	0.0	Maximum distance to move vertices along Y axis.
Z amplitude	Float	0.0	Maximum distance to move vertices along Z axis.
Seed	Int	0	Random seed.

Note. Each amplitude input specifies maximum distance to move vertices along corresponding axis. Vertices can be moved both in negative and positive directions. For example, for vertex X coordinate = 10.0, and `X amplitude` = 1.0, you can get output vertex coordinate from 9.0 to 11.0.

Outputs

This node has one output: **Vertices**.

Example of usage

Given simplest nodes setup:

you will have something like:

Recalculate Normals

This node testing is in progress, so it can be found under Beta menu

Functionality

This node recalculates normals of mesh faces, so that they all point either outside or inside. This is equivalent of Ctrl+N and Ctrl+Shift+N in the edit mode. This node is useful mainly when other nodes create degenerated geometry, or when geometry was generated by low-level operations with vertex indicies. It can be also used to just turn all normals inside out or vice versa.

Inputs

This node has the following inputs:

- **Vertices**
- **Edges**
- **Polygons**
- **Mask**. List of boolean or integer flags. Zero or False means do not process face with corresponding index. If this input is not connected, then all faces will be processed.

Parameters

This node has one parameter: **Inside** flag. The flag changes direction of normals to the opposite. By default, the flag is not set.

Outputs

This node has the following outputs:

- **Vertices**. This is just copy of input vertices for convinience.
- **Edges**.
- **Polygons**.

Example of usage

Visualisation of cube normals turned inside:

Making normals normal:

Remove Doubles

Functionality

This removes double vertices/edges/polygons, as do same-named command in blender

Inputs

- **Distance**
- **Vertices**
- **PolyEdge**

Parameters

Param	Type	Default	Description
Distance	Float	0.001	Maximum distance to weld vertices

Outputs

This node has the following outputs:

- **Vertices**
- **Edges**
- **Polygons**
- **Doubles** - Vertices, that was deleted.

Examples of usage

Triangulate Mesh

This node testing is in progress, so it can be found under Beta menu

Functionality

This node applies Triangulate operator (Ctrl+T in normal mode) to the mesh. It can triangulate all faces or only selected ones. This node is useful mainly when other node generates ngons, especially not-convex ones.

Inputs

This node has the following inputs:

- **Vertices**
- **Edges**
- **Polygons**
- **Mask**. List of boolean or integer flags. Zero or False means do not triangulate face with corresponding index. If this input is not connected, then all faces will be triangulated.

Parameters

This node has the following parameters:

- **Quads mode**. Method of quads processing. Available modes are:

  Beauty. Split the quads in nice triangles, slower method.

  Fixed Split the quads on the 1st and 3rd vertices.

  Fixed Alternate Split the quads on the 2nd and 4th vertices.

  Shortest Diagonal Split the quads based on the distance between the vertices.

- **Ngon mode**. Method of ngons processing. Available modes are:

  Beauty. Arrange the new triangles nicely, slower method.

  Scanfill. Split the ngons using a scanfill algorithm.

Outputs

This node has the following outputs:

- **Vertices**. This is just copy of input vertices for convinience.
- **Edges**.
- **Polygons**.
- **NewEdges**. This contains only new edges created by triangulation procedure.
- **NewPolys**. This contains only new faces created by triangulation procedure. If `Mask` input is not used, then this output will contain the same as `Polygons` output.

Examples of usage

Triangulated cube:

Triangulate only two faces of extruded polygon:

Mask Vertices

Functionality

Filter vertices with False/True bool values and automatically removes not connected edges and polygons.

Inputs

- **Mask**
- **Vertices**
- **Poly Edge**

Parameters

Param	Type	Default	Description
Mask	list of booleans	[1,0]	Mask can be defined with ListInput node or Formula node or other as list [n,n1,n2...ni] where n's can be 0 or 1 (False or True)

Outputs

- **Vertices**
- **Poly Edge**

Examples of usage

Modifier Make

Bisect

Functionality

This can give the cross section of an object shape from any angle. The implementation is from `bmesh.ops.bisect_plane`. It can also provide either side of the cut, separate or joined.

Inputs

Vertices, *PolyEdges* and *Matrix*

Parameters

Parameter	Type	Description
Clear Inner	bool	don't include the negative side of the Matrix cut
Clear Outer	bool	don't include the positive side of the Matrix cut
Fill cuts	bool	generates a polygon from the bisections

Outputs

Vertices, *Edges*, and *Polygons*.

Examples

Notes

Convex Hull

Functionality

Use this to skin a simple cloud of points. The algorithm is known as Convex Hull, and implemented in `bmesh.ops.convex_hull`.

Input

Vertices

Outputs

Vertices and *Polygons*. The number of vertices will be either equal or less than the original number. Any internal points to the system will be rejected and therefore not part of the output vertices.

Examples

Notes

Cross Section

Functionality

Sect object with blender operator to edges/polygons (F or Alt+F cases). In some cases work better than new bisect node.

Inputs

Vertices and **polygons** for object, that we cut, **matrix** for this object to deform, translate before cut. **Cut matrix** - it is plane, that defined by matrix (translation+rotation).

Parameters

table

Param	Type	Description
Fill section	Bool	Make polygons or edges
Alt+F/F	Bool	If polygons, than triangles or single polygon

Outputs

Vertices and **Edges/Polygons**.

Example of usage

Delaunay 2D

Adaptative Edges

Lathe

Functionality

Analogous to the *spin* operator and the Screw modifier. It takes a profile shape as input in the form of *vertices* and *edges* and produces *vertices* and *faces* based on a rotation axis, angle, center, delta and step count. Internally the node is powered by the bmesh.spin operator.

Inputs

It's vectorized, meaning it accepts nested and multiple inputs and produces multiple sets of output

Parameters

All Vector parameters (except axis) default to (0,0,0) if no input is given.

Param	Type	Description
cent	Vector	central coordinate around which to pivot
axis	Vector	axis around which to rotate around the pivot, default (0, 0, 1)
dvec	Vector	is used to push the center Vector by a Vector quantity per step
Degrees	Scalar, Float	angle of the total rotation. Default 360.0
Steps	Scalar, Int	numer of rotation steps. Default 20
Merge	Bool, toggle	removes double vertices if the geometry can be merged, usually used to prevent doubles of first profile and last profile copy. Default *off*.

Outputs

Vertices and **Poly**. Verts and Polys will be generated. The `bmesh.spin` operator doesn't consider the ordering of the Vertex and Face indices that it outputs. This might make additional processing complicated, use IndexViewer to better understand the generated geometry. Faces will however have consistent Normals.

Example of usage

See the progress of how this node came to life here (gifs, screenshots)

Glass, Vase.

Matrix Tube

destination after Beta: Modifier Make

Functionality

Makes a tube or pipe from a list of matrices. This node takes a list of matrices and a list of vertices as input. The vertices are joined together to form a ring. This ring is transformed by each matrix to form a new ring. Each ring is joined to the previous ring to form a tube.

Inputs

Matrices - List of transform matrices.

Vertices - Vertices of ring. Usually from a "Circle" or "NGon" node

Outputs

- **Vertices, Edges and Faces** - These outputs will define the mesh of the tube that skins the input matrices.

Example of usage

Pipe

Functionality

Making pipes from edges.

Inputs

Vers - Vertices of piped object.

Edgs - Edges of piped object.

diameter - Diameter of pipe.

nsides - Number of sides of pipe.

offset - Offset on length to awoid self intersection.

extrude - Scale the pipe on local X direction.

Properties

close - Close pipes between each other to make complete topology of united mesh.

Outputs

Vers - Vertices of output.

Pols - Polygons of output.

Examples

Adaptative Polygons

Functionality

Share one object's **verts+faces** to another object's **verts+faces**. Donor spreads itself onto recipient polygons, every polygon recieves a copy of the donor object and deforms according to the recipients face **normals**.

Limitations: This node was created primarily with Quads (quadrilateral polygons) in mind, and will output unusual meshes if you feed it Tris or Ngons in the recipient Mesh. Original code taken with permission from https://sketchesofcode.wordpress.com/2013/11/11/ by Alessandro Zomparelli (sketchesofcode).

Inputs

- **VersR** and **PolsR** is Recipient object's data.
- **VersD** and **PolsD** is donor's object data.
- **Z_Coef** is coefficient of height, can be vectorized.

Parameters

table

Param	Type	Description
Donor width	Float	Donor's spread width is part from recipient's polygons width

Outputs

Vertices and **Polygons** are data for created object.

Example of usage

Solidify

UV Connection

Functionality

Making edges/polygons between vertices objects it several ways.

Inputs

Vertices. Multysockets can eat many objects. every object to be connecting with other.

Parameters

table

Param	Type	Description
UVdir	Enum	Direction to connect edges and polygons
cicled	Bool, toggle	For edges and polygons close loop
polygons	Bool, toggle	Active - make polygon, else edge
slice	Bool, toggle	Polygons can be as slices or quads

Outputs

Vertices and **Edges/Polygons**. Verts and Polys will be generated. The Operator doesn't consider the ordering of the Vertex and Face indices that it outputs. This might make additional processing complicated, use IndexViewer to better understand the generated geometry. Faces will however have consistent Normals.

Example of usage

Voronoi 2D

Wafel

Functionality

This node make section possible to make as manufactured wafel structure. There is always pair of wafel nodes, two directions.

If you want start with this node - try open json first.

1. Import from import sverchok panel waffel_minimal.json to new layout.
2. Make some order in layout and developt to needed condition.

Inputs

vecLine - lines sected between sections of solid object. Form factor of object. each object has only two certices defining this section.

vecPlane - vectors of one side section.

edgPlane - closed edges (not planes) of one side section.

thickness - thickness of material to use in thickness of waffel slots.

Properties

threshold - threshold of line length from **vecLine** to limit cut material when producing.

Up/Down - direction of slots, there is only two cases, up or down. Not left and right and no XY directed vecLines never. Remember this.

Properties_extended

rounded - rounded edges.

Bind2 - circles to bind.

Contra - contrplane to define where to flip Up and Down direction. It is same as **vecPlane**.

Tube - case of section lines, making holes in body. It is same as **vecLine**.

Outputs

vert - vertices of output.

edge - edges of output.

centers - polygons centers.

Notes

Always make matrices rotations not orthogonal, it will not work 100%. Making something like (1.0,0.001,0) will work for matrix sections.

Always use Cross section nodes, not bisect, it will not work.

Examples

Wireframe

Number

Exponential Sequence

destination after Beta: Number

Functionality

This node produces specified number of items from exponential sequence, defined by formula x_n = x0*exp(alpha*n) or x_n = x0*base^n. Obviously, these formulas are equivalent when alpha = log(base).

Sequence can be re-scaled so that maximum of absolute values of produced items will be equal to specified value.

Note. Please do not forget well-known properties of exponential sequences:

- They grow very quickly when base is greater than 1.0 (or alpha greater than 0.0).
- They decrease very quickly when base is less than 1.0 (or alpha less than 0.0).

Inputs & Parameters

All parameters except for **Mode** can be given by the node or an external input. This node has the following parameters:

Parameter	Type	Default	Description
Mode	Enum: Log or Base	Log	If Log, then x_n = x0*exp(alpha*n). If Base, then x_n = x0*base^n.
X0	Float	1.0	Item of sequence for N=0.
Alpha	Float	0.1	Coefficient in formula exp(alpha*n). Used only in Log mode.
Base	Float	2.0	Exponential base in formula base^n. Used only in Base mode.
N from	Int	0	Minimal value of N.
N to	Int	10	Maximal value of N.
Max	Float	0.0	If non-zero, then all output sequence will be re-scaled so that maximum of absolute values will be equal to number specified.

Outputs

This node has one output: **Sequence**.

Inputs and outputs are vectorized, so if series of values is passed to one of inputs, then this node will produce several sequences.

Example of usage

Given simplest nodes setup:

you will have something like:

Fibonacci Sequence

destination after Beta: Number

Functionality

This node produces specified number of items from Fibonacci sequence:

```
1, 1, 2, 3, 5, 8, 13, 21 ...
```

Each next item is sum of two previous.

This node allows you to specify first two items for your sequence. Note that these numbers can be even negative.

Sequence can be re-scaled so that maximum of absolute values of produced items will be equal to specified value.

Inputs & Parameters

All parameters can be given by the node or an external input. This node has the following parameters:

Parameter	Type	Default	Description
X1	Float	1.0	First item of sequence.
X2	Float	1.0	Second item of sequence.
Count	Int	10	Number of items to produce. Minimal value is 3.
Max	Float	0.0	If non-zero, then all output sequence will be re-scaled so that maximum of absolute values will be equal to number specified.

Outputs

This node has one output: **Sequence**.

Inputs and outputs are vectorized, so if series of values is passed to one of inputs, then this node will produce several sequences.

Example of usage

Given simplest nodes setup:

you will have something like:

Float

Functionality

Float digit. Has maximum/minimum values and flexible labeling. Cached in Sverchok 3Dtoolbox.

Inputs & Parameters

float

Extended parameters

to-3d - boolean flag makes float catched in 3D toolbox

minimum - defines minimum value

maximum - defines maximum value

Outputs

float - only one digit.

Examples

Three cases of float. With output, as router and as input (not functional). Only first case will be catched in 'scan for propertyes' in 3Dtoolbox of sverchok.

This is 3D toolbox scanned float. Max and min values appears in 3d and node toolboxes (last in extended interface in propertyes panel). Label of node will apear in 3Dtoolbox and sorted depend on it. Flag 'to_3d' makes node catchable in 3D.

Notes

Float output only one digit, for ranges and lists reroute use route node.

Float to Integer

Functionality

Converts incoming *Float* values to the nearest whole number *(Integer)*. Accepts lists and preserves levels of nestedness.

Inputs

A *float*, alone or in a list

Outputs

An *int*, alone or in a list

Examples

```
1.0 becomes 1
-1.9 becomes -2
4.3 becomes 4
4.7 becomes 5

[1.0, 3.0, 2.4, 5.7] becomes [1, 3, 2, 6]
```

Formula

Functionality

Formula2 - support next operations:

- vector*vector, define hierarhy and calculate respectfully to it.
- Vector*scalar, the same. And output to vector.
- Moreover, you can define lists with formula, i.e. `` `0,1,2,3,4,5` `` for series or `` `(1,2,3),(1,2,3)` `` for vertices.
- **Supporting expressions beside * / - +:**
    - acos()
    - acosh()
    - asin()
    - asinh()
    - atan()
    - atan2()
    - atanh()

  - ceil()
  - copysign()
  - cos()
  - cosh()
  - degrees()
  - e
  - erf()
  - erfc()
  - exp()
  - expm1()
  - fabs()
  - factorial()
  - floor()
  - fmod()
  - frexp()
  - fsum()
  - gamma()
  - hypot()
  - isfinite()
  - isinf()
  - isnan()
  - ldexp()
  - lgamma()
  - log()
  - log10()
  - log1p()
  - log2()
  - modf()
  - pi
  - pow()
  - radians()
  - sin()
  - sinh()
  - sqrt()
  - str()
  - tan()

  - tanh()
  - trunc()
  - ==
  - !=
  - <, >
  - for, in, if, else
  - []

Inputs

X - main x that defines sequence. it can be range of vertices or range of floats/integers. If x == one number, than other veriables will be the same - one number, if two - two.

n[0,1,2,3,4] - multisocket for veriables.

Parameters

Formula - the string line, defining formula, i.e. `x>n[0]` or `x**n[0]+(n[1]/n[2])` are expressions. May have `x if x>n[0] else n[1]`

Outputs

Result - what we got as result.

Usage

Integer

Functionality

Integer digit. Has maximum/minimum values and flexible labeling. Cached in Sverchok 3Dtoolbox.

Inputs & Parameters

int

Extended parameters

to-3d - boolean flag makes integer catched in 3D toolbox

minimum - defines minimum value

maximum - defines maximum value

Outputs

int - only one digit.

Examples

Three cases of integer. With output, as router and as input (not functional). Only first case will be catched in 'scan for propertyes' in 3Dtoolbox of sverchok.

This is 3D toolbox scanned integer. Max and min values appears in 3d and node toolboxes (last in extended interface in propertyes panel). Label of node will apear in 3Dtoolbox and sorted depend on it. Flag 'to_3d' makes node catchable in 3D.

Notes

Integer output only one digit, for ranges and lists reroute use route node.

List Input

Functionality

Provides a way to creat a flat list of *Integers*, *Floats*, or *Vectors*. The length of the list is hardcoded to a maximum of **32** elements for integer or float and **10** vectors, we believe that if you need more then you should use a Text File and the Text In node.

Parameters

The value input fields change according to the Mode choice.

Output

A single *flat* `list`.

Examples

Useful when you have no immediate need to generate such lists programmatically.

Mix Numbers

Functionality

This node mixes two values using a given factor and a selected interpolation and easing functions.

For a factor of 0.0 it outputs the first value while the factor of 1.0 it outputs the last value. For every factor value between 0-1 it will output a value between the first and second input value. (*)

Note: (*) The Back and Elastic interpolations will generate outputs that are not strictly confined to the first-second value interval, but they will output values that start at first and end at second value.

Inputs & Parameters

All parameters except for **Type**, **Interpolation** and **Easing** can be given by the node or an external input.

This node has the following parameters:

Parameter	Type	Default	Description
Type	**Enum:** Int Float	Float	Type of inputs values to interpolate. When Float is selected the input value1 and value2 expect float values When Int is selected the input value1 and value2 expect int values.
Interpolation	**Enum:** Linear Sinusoidal Quadratic Cubic Quadric Quintic Exponential Circular Back Bounce Elastic	Linear	**Type of interpolation.** f(x) ~ x f(x) ~ sin(x) f(x) ~ x*2 f(x) ~ x^3 f(x) ~ x^4 f(x) ~ x^5 f(x) ~ e^x f(x) ~ sqrt(1-x*x) f(x) ~ x*x*x - x*sin(x) f(x) ~ series of geometric progression parabolas f(x) ~ sin(x) * e^x
Easing	**Enum** Ease In Ease Out Ease In-Out	Ease In-Out	**Type of easing.** Ease In = slowly departs the starting value Ease Out = slowly approaches the ending value Ease In-Out = slowly departs and approaches values
Value1	Int or Float	0 or 0.0	Starting value
Value2	Int or Float	1 or 1.0	Ending value
Factor	Float	0.5	Mixing factor (between 0.0 and 1.0)

Extra Parameters

For certain interpolation types the node provides extra parameters on the property panel.

- Exponential Extra parameters to adjust the base and the exponent of the exponential function. The Defaults are 2 and 10.0.
- Back Extra parameters to adjust the scale of the overshoot. The default is 1.0.
- Bounce Extra parameters to adjust the attenuation of the bounce and the number of bounces. The defaults are 0.5 and 4.
- Elastic Extra parameters to adjust the base and the exponent of the damping oscillation as well as the number of bounces (oscillations).

The defaults are 1.6, 6.0 and 6.

Outputs

This node has one output: **Value**.

Inputs and outputs are vectorized, so if series of values is passed to one of inputs, then this node will produce several sequences.

Example of usage

Given simplest nodes setup:

#

you will have something like:

#

Random

Functionality

Produces a list of random numbers from a seed value.

Inputs & Parameters

Parameters	Description
Count	Number of random numbers to spit out
Seed	Accepts float values, they are hashed into *Integers* internally.

What's a Seed? Read the Python docs here

Outputs

A list, or nested lists.

Examples

Notes

Providing a float values as a Seed parameter may be unconventional, if you are uncomfortable with it you could place a *FloatToInt* node before the Seed parameter. We may add more Random Nodes in future.

Range Float

Range Integer

Functionality

alias: List Range Int

Useful for generating sequences of Integer values. The code perhaps describes best what the two modes do:

```
def intRange(start=0, step=1, stop=1):
    '''
    "lazy range"
    - step is always |step| (absolute)
    - step is converted to negative if stop is less than start
    '''
    if start == stop:
        return []
    step = max(step, 1)
    if stop < start:
        step *= -1
    return list(range(start, stop, step))


def countRange(start=0, step=1, count=10):
    count = max(count, 0)
    if count == 0:
        return []
    stop = (count*step) + start
    return list(range(start, stop, step))
```

Inputs and Parameters

One UI parameter controls the behaviour of this Node; the `Range | Count` Mode switch. The last input changes accordingly.

Mode	Input	Description
Both	Start	value to start at
	Step	value of the skip distance to the next value. The Step value is considered the absolute difference between successive numbers.
Range	Stop	last value to generate, don't make values beyond this. If this value is lower than the start value then the sequence will be of descending values.
Count	Count	number of values to produce given Start and Step. **Never negative** - negative produces an empty list

A word on implementation:

This Node accepts only Integers and lists of Integers, so you must convert Floats to Int first. The reason is purely superficial - there is no reasonable argument not to automatically cast values.

Outputs

Integers only, in list form.

Examples

Non-vectorized

Int Range

```
intRange(0,1,10)
>>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
intRange(0,2,10)
>>> [0, 2, 4, 6, 8]

intRange(-4,1,6)
>>> [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5]

intRange(2,1,-4)
>>> [2, 1, 0, -1, -2, -3]
```

Count Range

```
countRange(0,1,5)
>>> [0, 1, 2, 3, 4]

countRange(0,2,5)
>>> [0, 2, 4, 6, 8]

countRange(-4,1,6)
>>> [-4, -3, -2, -1, 0, 1]

countRange(2,1,4)
>>> [2, 3, 4, 5]
```

Vectorized

Progress Thread in the issue tracker shows several examples.

Map Range

Math

Vector

Attraction Vectors

Functionality

This node calculates vectors directed from input vertices to specified attractor. Vector lengths are calculated by one of physics-like falloff laws (like 1/R^2), so it looks like attractor attracts vertices, similar to gravity force, for example. Output vectors can be used to move vertices along them, for example.

Inputs

This node has the following inputs:

- **Vertices**
- **Center**. Center of used attractor. Exact meaning depends on selected attractor type.
- **Direction**. Direction of used attractor. Exact meaning depends on selected attractor type. Not available if attractor type is **Point**.
- **Amplitude**. Coefficient of attractor power. Zero means that all output vectors will be zero. If many values are provided, each value will be matched to one vertex.

- **Coefficient**. Scale coefficient for falloff law. Exact meaning depends on selected falloff type. Available only for falloff types **Inverse exponent** and **Gauss**. If many values are provided, each value will be matched to one vertex.

Parameters

This node has the following parameters:

- **Attractor type**. Selects form of used attractor. Available values are: - **Point**. Default value. In simple case, attractor is just one point specified in **Center** input. Several points can be passed in that input; in this case, attracting force for each vertex will be calculated as average of attracting forces towards each attractor point. - **Line**. Attractor is a straight line, defined by a point belonging to this line (**Center** input) and directing vector (**Direction** input). - **Plane**. Attractor is a plane, defined by a point belonging to this line (**Center** input) and normal vector (**Direction** input).
- **Falloff type**. Used falloff law. Avalable values are: - **Inverse**. Falloff law is 1/R, where R is distance from vertex to attractor. - **Inverse square**. Falloff law is 1/R^2. This law is most common in physics (gravity and electromagnetizm), so this is the default value. - **Inverse cubic**. Falloff law is 1/R^2. - **Inverse exponent**. Falloff law is *exp(- C * R)*, where R is distance from vertex to attractor, and C is value from **Coefficient** input. - **Gauss**. Falloff law is *exp(- C * R^2 / 2)*, where R is distance from vertex to attractor, and C is value from **Coefficient** input.
- **Clamp**. Whether to restrict output vector length with distance from vertex to attractor. If not checked, then attraction vector length can be very big for vertices close to attractor, depending on selected falloff type. Default value is True.

Outputs

This node has the following outputs:

- **Vectors**. Calculated attraction force vectors.
- **Directions**. Unit vectors in the same directions as attracting force.
- **Coeffs**. Lengths of calculated attraction force vectors.

Examples of usage

Most obvious case, just a plane attracted by single point:

Plane attracted by single point, with Clamp unchecked:

Not so obvious, plane attracted by circle (red points):

Coefficients can be used without directions:

Torus attracted by a line along X axis:

Sphere attracted by a plane:

Vector X | Y | Z

Functionality

Sometimes a Vector is needed which only has a value in one of the 3 axes. For instance the rotation vector of the *Matrix In* node. Or the Axis parameter in the *Lathe Node*. Instead of using a *Vector* Node it can be useful to add this Node instead, which lets you easily toggle between:

```
X = 1, 0, 0
Y = 0, 1, 0
Z = 0, 0, 1
```

The added bonus is that the minimized state of the Node can show what type of Vector it represents.

Parameters

A toggle between `X | Y | Z`

Outputs

A single Vector output, only ever:

```
(1,0,0) or (0,1,0) or (0,0,1)
```

Examples

issue tracker thread

Vector Drop

Functionality

Reverse operation to Matrix apply. If matrix apply adding all transformations to vertices. Than vector drop substract matrix from vertices.

Inputs

Vertices - Vectors input to transform **Matrix** - Matrix to substract from vertices

Outputs

Vertices - vertices

Examples

Vector Interpolation Stripes

Functionality

Performs cubic spline STRIPES interpolation based on input points by creating a function `x,y,z = f(t)` with `tU=[0,1]`, `tU=[0,1]` and attractor vertex. The interpolation is based on the distance between the input points. Stripes outputs as two lines of points for each object, so UVconnect node can handle it and make polygons for stripes.

Input & Output

socket	name	Description
input	Vertices	Points to interpolate
input	tU	Values to interpolate in U direction
input	tV	Values to interpolate in V direction
input	Attractor	Vertex point as attractor of influence
output	vStripes	Interpolated points as grouped stripes [[a,b],[a,b],[a,b]], where a and b groups [v,v,v,v,v], where v - is vertex
output	vShape	Interpolated points simple interpolation
output	sCoefs	String of float coefficients for each point

Parameters

Factor - is multiplyer after produce function as sinus/cosinus/etc. **Scale** - is multiplyer before produce function as sinus/cosinus/etc. **Function** - popup function between Simple/Multiplyed/Sinus/Cosinus/Power/Square

Parameters extended

minimum - minimum value of stripe width (0.0 to 0.5) **maximum** - maximum value of stripe width (0.5 to 1.0)

Examples

Making surface with stripes separated in two groups of nodes for UVconnect node to process:

Vector Interpolation

Functionality

Performs linear or cubic spline interpolation based on input points by creating a function `x,y,z = f(t)` with `t=[0,1]`. The interpolation is based on the distance between the input points.

Input & Output

socket	name	Description
input	Vertices	Points to interpolate
input	t	Value to interpolate
output	Vertices	Interpolated points

Examples

Sine interpolated from 5 points. The input points are shown with numbers.

An interpolated surface between sine and cosine.

Notes

The node doesn't extrapolate. Values outside of `[0, 1]` are ignored. It doesn't support cyclic interpolation (TODO).

Vector Evaluate

Functionality

Vector Evaluate need two groups of vertices (or just 2) as inputs to evaluate al the global positions between them.

Inputs

- **Factor**
- **Vertice A**
- **Vertice B**

Only **Factor** can be set inside the node. There is no default values for Vertice A or B.

Parameters

All parameters need to proceed from an external node.

Param	Type	Default	Description
Vertice A	Vertices	None	first group of vertices
Vectice B	Vertices	None	second group of vertices
Factor	Float	0.50	distance percentage between vertices A and B

Outputs

EvPoint will need Vertices A and B to be generated. The output will be a new group of vertices between groups A and B, based on the factor setting. See example below.

Example of usage

In this example just two vertices are evaluated. The gif shows the output based on the factor setting.

Vector Math Node

This is a versatile node. You can perform 1 operation on 1000's of list-elements, or perform operations pairwise on two lists of 1000's of elements, even if they are nested. It is therefore what we call a *Vectorized* node, for an elaborate explanation of what this means see this [introduction]().

The node expects correct input for the chosen operation (called mode), but it will fail gracefully with a message in the console if the input is not right for the selected mode.

Input and Output

socket	description
inputs	Expect a Vector and Scalar (v,s), or two Vectors (u, v)
outputs	Will output a Scalar (s), or a Vector (w).

Depending on the mode you choose the sockets are automatically changed to accommodate the expected inputs and outputs types

Modes

Most operations are self explanatory, but in case they aren't then here is a quick overview:

Tables	inputs	outputs	description
Cross product	u, v	s	u cross v
Dot product	u, v	s	u dot v
Add	u, v	w	u + v
Sub	u, v	w	u - v
Length	u	s	distance(u, origin)
Distance	u, v	s	distance(u, v)
Normalize	u	w	scale vector to length 1
Negate	u	w	reverse sign of components
Noise Vector	u	w	[see mathutils]()
Noise Scalar	u	s	[see mathutils]()
Scalar Cell noise	u	s	[see mathutils]()
Vector Cell noise	u	w	[see mathutils]()
Project	u, v	w	u project v
Reflect	u, v	w	u reflect v
Multiply Scalar	u, s	w	multiply(vector, scalar)
Multiply 1/Scalar	u, s	w	multiply(vector, 1/scalar)
Angle Degrees	u, v	s	angle(u, origin, v)
Angle Radians	u, v	s	angle(u, origin, v)
Round s digits	u, s	v	reduce precision of components
Component-wise U*V	u, v	w	*w = (u.x*v.x, u.y*v.y, u.z*v.z)*

Vector Noise

This noise node takes a list of Vectors and outputs a list of equal length containing either Vectors or Floats in the range 0.0 to 1.0. The seed value permits you to apply a different noise calculation to identical inputs.

Inputs & Parameters

Parameters	Description
Noise Function	Pick between Scalar and Vector output
Noise Type	Pick between several noise types • Blender • Cell Noise • New Perlin • Standard Perlin • Voronoi Crackle • Voronoi F1 • Voronoi F2 • Voronoi F2F1 • Voronoi F3 • Voronoi F4 See mathutils.noise docs (Noise)
Seed	Accepts float values, they are hashed into *Integers* internally. Seed values of 0 will internally be replaced with a randomly picked constant to allow all seed input to generate repeatable output. (Seed=0 would otherwise generate random values based on system time)

Examples

Notes

This documentation doesn't do the full world of noise any justice, feel free to send us layouts that you've made which rely on this node.

Fractal

This fractal node takes a list of Vectors and outputs a list of equal length containing Floats in the range 0.0 to 1.0.

Inputs & Parameters

Parameters	Description
Noise Function	The node output only Scalar values
Noise Type	Pick between several noise types • Blender • Cell Noise • New Perlin • Standard Perlin • Voronoi Crackle • Voronoi F1 • Voronoi F2 • Voronoi F2F1 • Voronoi F3 • Voronoi F4 See mathutils.noise docs (Noise)
Fractal Type	Pick between several fractal types • Fractal • MultiFractal • Hetero terrain • Ridged multi fractal • Hybrid multi fractal
H_factor	Accepts float values, they are hashed into *Integers* internally.
Lacunarity	Accepts float values
Octaves	Accepts integers values
Offset	Accepts float values
Gain	Accepts float values

Examples

Basic example with a Vector rewire node.

json file: https://gist.github.com/kalwalt/5ef4f6b6018724874e3c51eaa255930c

Notes

This documentation doesn't do the full world of fractals any justice, feel free to send us layouts that you've made which rely on this node.

Links

Fractals description from wikipedia: https://en.wikipedia.org/wiki/Fractal

A very interesting resource is "the book of shaders", it's about shader programming but there is a very useful fractal paragraph:

http://thebookofshaders.com/13/ and on github repo: https://github.com/patriciogonzalezvivo/thebookofshaders/tree/master/13

Vector Lerp

Functionality

This node's primary function is to perform the linear interpolation between two Vectors, or streams of Vectors. If we have two Vectors A and B, and a factor 0.5, then the output of the node will be a Vector exactly half way on the imaginary finite-line between A and B. Values beyond 1.0 or lower than 0.0 will be extrapolated to beyond the line A-B.

Inputs

Vector Evaluate needs two Vertex stream inputs (each containing 1 or more vertices). If one list is shorter than the other then the shortest stream is extended to match the length of the longer stream by repeating the last valid vector found in the shorter stream.

Parameters

Param	Type	Default	Description
mode	Enum	Lerp	• **Lerp** will linear interpolate once between each corresponding Vector • **Evaluate** will repeatedly interpolate between each member of vectors A and B for all items in Factor input (see example)
Vertex A	Vector	None	first group of vertices (Stream)
Vertex B	Vector	None	second group of vertices (Stream)
Factor	Float	0.50	distance ratio between vertices A and B. values outside of the 0.0...1.0 range are extrapolated on the infinite line A, B

Outputs

The content of **EvPoint** depends on the current mode of the node, but it will always be a list (or multiple lists) of Vectors.

Example of usage

0.5

-0.5

1.5

range float ‘0.0 ... 1.0 n=10‘ Evaluate

range float ‘0.0 ... 1.0 n=10‘ Lerp

Lerp interpolation with noise

The development thread also has examples: https://github.com/nortikin/sverchok/issues/1098

Vector Rewire

Functionality

Use this node to swap Vector components, for instance pass X to Y (and Y to X). Or completely filter out a component by switching to the Scalar option. it will default to *0.0* when the Scalar socket is unconnected, when connected it will replace the component with the values from the socket. If the content of the Scalar input lists don’t match the length of the Vectors list, the node will repeat the last value in the list or sublist (expected Sverchok behaviour).

Inputs

Vectors - Any list of Vector/Vertex lists **Scalar** - value or series of values, will auto repeat last valid value to match Vector count.

Outputs

Vector - Vertex or series of vertices

Vector In

Functionality

Inputs vector from ranges or number values either integer of floats.

Inputs

x - velue or series of values **y** - velue or series of values **z** - velue or series of values

Outputs

Vector - Vertex or series of vertices

Examples

Vector Out

Functionality

Outputs values/numbers from vertices.

Inputs

Vector - Vertex or series of vertices

Outputs

x - velue or series of values **y** - velue or series of values **z** - velue or series of values

Examples

Vector Polar Input

destination after Beta: Vector

Functionality

This node generates a vector from it's cylindrical or spherical coordinates. Angles can be measured in radians or in degrees.

Inputs & Parameters

All parameters except for `Coordinates` and `Angles mode` can be specified using corresponding inputs.

Parameter	Type	Default	Description
Coordinates	Cylindrical or Spherical	Cylindrical	Which coordinates system to use.
Angles mode	Radians or Degrees	Radians	Interpret input angles as specified in radians or degrees.
rho	Float	0.0	Rho coordinate.
phi	Float	0.0	Phi coordinate.
z	Float	0.0	Z coordinate. This input is used only for cylindrical coordinates.
theta	Float	0.0	Theta coordinate. This input is used only for spherical coordinates.

Outputs

This node has one output: **Vectors**. Inputs and outputs are vectorized, so if you pass series of values to one of inputs, you will get series of vectors.

Examples of usage

An archimedian spiral:

Logariphmic spiral:

Helix:

With spherical coordinates, you can easily generate complex forms:

Vector Polar Output

destination after Beta: Vector

Functionality

This node decomposes a vector to it's cylindrical or spherical coordinates. Angles can be measured in radians or in degrees.

Parameters

This node has the following parameters:

- **Coordinates**. Cylindrical or Spherical. Default mode is Cylindrical.
- **Angles mode**. Should this node output angles measured in radians or in degrees. By default Radians.

Inputs

This node has one input: **Vectors** Inputs and outputs are vectorized, so if you pass series of vectors to input, you will get series of values on outputs.

Outputs

This node has the following outputs:

- **rho**. Rho coordinate.
- **phi**. Phi coordinate.
- **z**. Z coordinate. This output is used only for Cylindrical coordinates.
- **theta**. Theta coordinate. This output is used only for Spherical coordinates.

Examples of usage

Cube push-up:

Vector X Doubles

Vector Sort

Turbulence

This Turbulence node takes a list of Vectors and outputs a list of equal length containing Floats in the range 0.0 to 1.0. May output scalars or vectors. For some noise types, if your output goes to the texture viewer you need to remap them, otherwise your texture will be supersaturated or undersaturated. See below 'range table' for a detailed description.

Inputs & Parameters

Parameters	Description
Noise Function	Pick between Scalar and Vector output
Noise Type	Pick between several noise types • Blender • Cell Noise • New Perlin • Standard Perlin • Voronoi Crackle • Voronoi F1 • Voronoi F2 • Voronoi F2F1 • Voronoi F3 • Voronoi F4 See mathutils.noise docs (Noise)
Octaves	Accepts integers values The number of different noise frequencies used.
Hard	Accepts bool values: Hard(True) or Soft(False) Specifies whether returned turbulence is hard (sharp transitions) or soft (smooth transitions).
Amplitude	Accepts float values. The amplitude scaling factor.
Frequency	Accepts float values. The frequency scaling factor.

Range table

Scalar values from turbulence node with size(n.verts)=64x64, step=0.05, octaves=3, amplitude=0.5, frequency=2.0, random seed=0. Plug a map range node in the scalar output and map it to the desired range (min=0, max=1) as in the image below.

Noise Type	median	maximum	minimum
Blender	0.4574402868747711	1.2575798034667969	0.0
Stdperlin	0.37063807249069214	0.972740530967712	0.0
Newperlin	0.2982039898633957	0.7674642205238342	0.0
Voronoi_F1	0.5178706049919128	1.184566617012024	0.016996487975120544
Voronoi_F2	0.9441720247268677	1.696974754333496	0.07561451196670532
Voronoi_F3	1.3248268961906433	2.267115831375122	0.24465730786323547
Voronoi_F4	1.6119314432144165	2.4261345863342285	0.7868537306785583
Voronoi_F1F2	1.0320665836334229	1.7262239456176758	0.06919857859611511
Voronoi_Crackle	1.5918831825256348	1.75	0.12337762117385864
Cellnoise	0.9668738842010498	1.5000858306884766	0.1691771298646927

Examples

Basic example with a Scalar output and Vector rewire node.

Notes

This documentation doesn't do the full world of fractals any justice, feel free to send us layouts that you've made which rely on this node.

Links

Fractals description from wikipedia: https://en.wikipedia.org/wiki/Fractal

A Perlin Noise and Turbulence description by Prof. Paul Bourke: http://paulbourke.net/texture_colour/perlin/

An introduction on Noise and Turbulence by Dr. Matthew O. Ward: https://web.cs.wpi.edu/~matt/courses/cs563/talks/noise/noise.html

A very interesting resource is "the book of shaders", it's about shader programming but there is a very useful fractal paragraph:

http://thebookofshaders.com/13/ and on github repo: https://github.com/patriciogonzalezvivo/thebookofshaders/tree/master/13

Matrix

Apply Matrix to Mesh

Functionality

Applies a Transform Matrix to a list or nested lists of vertices, edges and faces. If several matrices are provided on the input, then this node will produce several meshes.

Note. Unless there is further processing going on which explicitly require the duplicated topology, then letting the `Viewer Draw` or `BMesh Viewer` nodes automatically repeat the index lists for the edges and faces is slightly more efficient than use of this node.

Inputs

This node has the following inputs:

- **Vertices**. Represents vertices or intermediate vectors used for further vector math.
- **Edges**
- **Faces**
- **Matrices**. One or more, never empty.

Parameters

This node has the following parameter:

Join. If set, then this node will join output meshes into one mesh, the same way as `Mesh Join` node does. Otherwise, if N matrices are provided at the input, this node will produce N lists of vertices, N lists of edges and N lists of faces.

Outputs

This node has the following outputs:

- **Vertices**. Nested list of vectors / vertices, matching the number nested incoming *matrices*.
- **Edges**. Input edges list, repeated the number of incoming matrices. Empty if corresponding input is empty.
- **Faces**. Input faces list, repeated the number of incoming matrices. Empty if corresponding input is empty.

Examples

Matrix Deform

Euler

Matrix Input

Matrix Interpolation

Matrix In & Out

Matrix In & Out

Matrix Shear

Functionality

Similar in behaviour to the `Transform -> Shear` tool in Blender (docs).

Matrix Shear generates a Transform Matrix which can be used to change the locations of vertices in two directions. The amount of transformation to introduce into the Matrix is given by two *Factor* values which operate on the corresponding axes of the selected *Plane*.

Inputs & Parameters

Parameters	Description
Plane	`options = (XY, XZ, YZ)`
Factor1 & Factor2	these are *Scalar float* values and indicate how much to affect the axes of the transform matrix

Outputs

A single `4*4` Transform Matrix

Examples

Usage: This is most commonly connected to Matrix Apply to produce the Shear effect.

Logic

Logic

Functionality

This node offers a variety of logic gates to evaluate any boolean inputs It also has different operations to evaluate a pair of numbers, like minor than or greater than.

Input and Output

Depending on the mode you choose the sockets are automatically changed to accommodate the expected inputs. Output is always going to be a boolean.

Parameters

Most operations are self explanatory, but in case they aren't then here is a quick overview:

Tables	inputs	type	description
And	x, y	integer	True if X and Y are True
Or	x, y	integer	True if X or Y are True
Nand	x, y	integer	True if X or Y are False
Nor	x, y	integer	True if X and Y are False
Xor	x, y	integer	True if X and Y are opposite
Xnor	x, y	integer	True if X and Y are equals
If	x	integer	True if X is True
Not	x	integer	True if X is False
<	x, y	float	True if X < Y
>	x, y	float	True if X > Y
==	x, y	float	True if X = Y
!=	x, y	float	True if X not = Y
<=	x, y	float	True if X <= Y
>=	x, y	float	True if X >= Y
True	none	none	Always True
False	none	none	Always False

Example of usage

In this example we use Logic with Switch Node to choose between two vectors depending on the logic output.

Elman neuro node layer 1

Neuro network node This node teachable. You may teach him rules, that he understand himself. Just put data and correct answer. When displace answer, he will find right answer himself. Input data. Inserting many objects - output many objects. Inserting one object with many parameters - output one object. Always insert constant numbers count of parameters, otherwise it will reset neuro data and start every time from beginning. Keep constant numbers count.

- coef_learning - learning speed coeffitient, accuracy influence (less - more accuracy);
- gisterezis - spread of input and etalon data;
- maximum - maximum number input (better define little overhang number);
- cycles - passes on one object;
- A layer - input layer cores (and it is number of objects);
- B layer - inner layer cores - more - smarter (overlearning is bad too);
- C layer - output layer cores - numbers quantity in output;
- epsilon - inner variable - argument offset in passes 'cycles' (not much influence totally);
- lambda - holding coefficient, to preserve data flooding;

- threshold - inner variable - defines reasonability limit in passes 'cycles' (not much influence totally).

Switch

Functionality

Switches between to sets of inputs.

Inputs

Input	Description
state	state that decides which set of sockets to use
T 0	If state is false this socket is used
F 0	If state is true this socket used

Parameters

Count

Number of sockets in each set.

state

If set to 1 T sockets are used, otherwise the F socket are used.

Outputs

Out 0 to Out N depending on count. Socket types are copied from first from the T set.

Examples

Switching between a sphere and cylinder for drawing using switch node.

List Main

List Decompose

Functionality

Inverse to list join node. Separate list at some level of data to several sockets. Sockets count the same as items count in exact level.

Inputs

- **data** - adaptable socket

Parameters

Parameter	Type	Default	Description
level	Int	1	Level of data to operate.
Count	Int	1	Output sockets' count. defined manually or with Auto set
Auto set	Button		Calculate output sockets' count based on data count on choosen level

Outputs

- **data** - multisocket

Example of usage

Decomposed simple list in 2 level:

List Math

Functionality

This nodes offers some operations to make over list, meaning a group of numbers.

Inputs

It will operate only with list of single numbers, not tuples or vectors.

Parameters

Level: Set the level at which to observe the List. **Function:** Select the type of operation.

Tables	description
Sum	sum of all the elements of the list
Average	average of element at selected level
Maximum	Maximum value of the list
Minimum	Minimum value of the list

Outputs

The output is always going to be a number, integer or float, depending on the input list.

- You can try to use this node with vectors, but it isn't going to work properly. For operations with vectors you should use **Vector Math** node.

Examples

In this example the node shows all the possible outputs.

List Join

Functionality

level 1: [[(1,2,3), (4,5,6)]] + [[(7,8,9), (10,11,12)]] = [[(1,2,3), (4,5,6), (7,8,9), (10,11,12)]]

level 2 mix: [[(1,2,3), (4,5,6)]] + [[(7,8,9), (10,11,12)]] = [[(1,2,3),(7,8,9),(4,5,6),(10,11,12)]]

level 2 wrap: [[(1,2,3), (4,5,6)]] + [[(7,8,9), (10,11,12)]] = [[[(1,2,3),(4,5,6)], [(7,8,9),(10,11,12)]]]

level 2 mix + wrap: [[(1,2,3), (4,5,6)]] + [[(7,8,9), (10,11,12)]] = [[[(1,2,3),(7,8,9)], [(4,5,6),(10,11,12)]]]

level 3: [[(1,2,3), (4,5,6)]] + [[(7,8,9), (10,11,12)]] = [[[1,2,3,4,5,6,7,8,9,10,11,12]]]

level 3 mix: [[(1,2,3), (4,5,6)]] + [[(7,8,9), (10,11,12)]] = [[[1,7,2,8,3,9,4,10,5,11,6,12]]]

level 3 wrap: [[(1,2,3), (4,5,6)]] + [[(7,8,9), (10,11,12)]] = [[[1,2,3,4,5,6],[7,8,9,10,11,12]]]

level 3 mix + wrap: [[(1,2,3), (4,5,6)]] + [[(7,8,9), (10,11,12)]] = [[[1,7],[2,8],[3,9],[4,10],[5,11],[6,12]]]

Inputs

data multisocket

Parameters

mix to mix (not zip) data inside **wrap** to wrap additional level **levels** level of joining

Outputs

data adaptable socket

List Length

Functionality

The Node equivalent of the Python `len()` function. The length is inspected at the Level needed.

Inputs

Takes any kind of data.

Parameters

Level: Set the level at which to observe the List.

Outputs

Depends on incoming data and can be nested. Level 0 is top level (totally zoomed out), higher levels get more granular (zooming in) until no higher level is found (atomic). The length of the most atomic level will be 1, for instance single ints or float or characters. The output will reflect the nestedness of the incoming data.

Examples

Often a few experiments with input hooked-up to a debug node will make the exact working of this Node instantly clearer than any explanation.

List Delete Levels

Functionality

This helps flatten lists, or make them less nested.

The analogy to keep in mind might be:

> knocking through walls in a house to join two spaces, or knock non load bearing walls between buildings to join them.

Incoming nested lists can be made less nested.

```
# del level 0, remove outer wrapping

[[0,1,2,3,4]]
>>> [0,1,2,3,4]

[[4, 5, 6], [4, 7, 10], [4, 9, 14]]
>>> [4, 5, 6, 4, 7, 10, 4, 9, 14]

[[5], [5], [5], [5], [5], [5]]
>>> [5, 5, 5, 5, 5, 5]
```

Usage

Type 1,2 or 2,3 or 1,3 or 1,2,3 or 3,4 etc to leave this levels and remove others.

Throughput

Socket	Description
Input	Any meaningful input, lists, nested lists
Output	Modified according to Levels parameter, or None

Parameters

Levels, this text field

Examples

Notes

List Match

List Summa

Functionality

This node operates the sum of all the values in the inputs, no matter the levels or sublist inside the input.

Inputs

Takes any kind of data: singles values, vectors or even matrixes.

Outputs

No matter the type of input, the output will be the sum of all the values in the input list. This node works in a different way of the "Sum" function in **List Math** node. See the example below to see the difference.

Examples

List Zip

Functionality

Making pares of data to mix togather as zip function.

x = [[[1,2],[3,4]]] y = [[[5,6],[7,8]]]

out level 1 = [[[[1, 2], [5, 6]], [[3, 4], [7, 8]]]] out level 1 unwrap = [[[1, 2], [5, 6]], [[3, 4], [7, 8]]] out level 2 = [[[[1, 3], [2, 4]], [[5, 7], [6, 8]]]] out level 2 unwrap = [[[1, 3], [2, 4]], [[5, 7], [6, 8]]] out level 3 = [[[[], []], [[], []]]]

Inputs

data multysocket

Properties

level integer to operate level of conjunction **unwrap** boolean to unwrap from additional level, added when zipped

Outputs

data adaptable socket

List Struct

List Flip

Functionality

Flips the data on selected level. [[[1,2,3],[4,5,6],[7,8,9]],[[3,3,3],[1,1,1],[8,8,8]]] (two objects, three vertices) with level 2 turns to: [[[1, 2, 3], [3, 3, 3]], [[4, 5, 6], [1, 1, 1]], [[7, 8, 9], [8, 8, 8]]] (three objects, two vertices) with level 3 turns to: [[1, 4, 7], [2, 5, 8], [3, 6, 9], [3, 1, 8], [3, 1, 8], [3, 1, 8]] (six objects with three digits) last example is not straight result, more as deviation. Ideally Flip has to work with preserving data levels and with respect to other levels structure. But for now working level is 2

Inputs

data - data to flip

Properties

level - level to deal with

Outputs

data - flipped data

Examples

List Item

Functionality

Select items from list based on index. The node is *data type agnostic*, meaning it makes no assumptions about the data you feed it. It shoudld accepts any type of data native to Sverchok..

Inputs

Input	Description
Data	The data - can be anything
item	Item(s) to select, allows negative index python index

Parameters

Level

It is essentially how many chained element look-ups you do on a list. If `SomeList` has a considerable *nestedness* then you might access the most atomic element of the list doing `SomeList[0][0][0][0]`. Levels in this case would be 4.

item

A list of items to select, allow negative index python indexing so that -1 the last element. The items doesn't have to be in order and a single item can be selected more than a single time.

Outputs

Item, the selected items on the specifed level. Other, the list with the selected items deleted.

Examples

Trying various inputs, adjusting the parameters, and piping the output to a *Debug Print* (or stethoscope) node will be the fastest way to acquaint yourself with the inner workings of the *List Item* Node.

List Repeater

Functionality

Allows explicit repeat of lists and elements. The node is *data type agnostic*, meaning it makes no assumptions about the data you feed it. It shoudld accepts any type of data native to Sverchok..

Inputs

Input	Description
Data	The data - can be anything
Number	The amount of times to repeat elements selected by the *Level* parameter

Parameters

Level and unwrap.

Level

It is essentially how many chained element look-ups you do on a list. If `SomeList` has a considerable *nestedness* then you might access the most atomic element of the list doing `SomeList[0][0][0][0]`. Levels in this case would be 4.

unwrap

Removes any extra layers of wrapping (brackets or parentheses) found at the current Level. If the element pointed at is `[[0,2,3,2]]` it will become `[0,2,3,2]`.

Outputs

Lists (nested). The type of *Data out* will be appropriate for the operations defined by the parameters of the Node.

Examples

Trying various inputs, adjusting the parameters, and piping the output to a *Debug Print* (or stethoscope) node will be the fastest way to acquaint yourself with the inner workings of the *List Repeater* Node.

A practical reason to use the node is when you need a series of copies of edge or polygon lists. Usually in conjunction with *Matrix Apply*, which outputs a series of *vertex lists* as a result of transform parameters.

List Reverse

Functionality

Reverse items from list based on index. It should accept any type of data from Sverchok: Vertices, Strings (Edges, Polygons) or Matrix.

Inputs

Takes any kind of data.

Parameters

Level: Set the level at which to observe the List.

Outputs

Depends on incoming data and can be nested. Level 0 is top level (totally zoomed out), higher levels get more granular (zooming in) until no higher level is found (atomic). The node will just reverse the data at the level selected.

Examples

In this example the node reverse a list a integers

List Shift

Functionality

Shifting data in selected level on selected integer value as:

> [0,1,2,3,4,5,6,7,8,9] with shift integer 4 will be [4,5,6,7,8,9] But with enclose flag: [4,5,6,7,8,9,0,1,2,3]

Inputs

data - list of data any type to shift **Shift** - value that defines shift

Properties

level - manipulation level, 0 - is objects shifting **enclose** - close data when shifting, that way ending cutted numbers turns to beginning

Outputs

data - shifter data, adaptive socket

Examples

List Shuffle

Functionality

Shuffle (randomize) the order of input lists.

Inputs

Input	Description
Data	The data - can be anything
Seed	Seed setting used by shuffle operation

Parameters

Level

It is essentially how many chained element look-ups you do on a list. If `SomeList` has a considerable *nestedness* then you might access the most atomic element of the list doing `SomeList[0][0][0][0]`. Levels in this case would be 4.

Seed

Affects the output order.

Outputs

Item, the selected items on the specified level. Other, the list with the selected items deleted.

Examples

The shuffle operation is based on the python random.shuffle. https://docs.python.org/3.4/library/random.html?highlight=shuffle#random.shuffle

Trying various inputs, adjusting the parameters, and piping the output to a *Debug Print* (or stethoscope) node will be the fastest way to acquaint yourself with the inner workings of the *List Shuffle* Node.

List Slice

Functionality

Select a slice from a list. The node is *data type agnostic*, meaning it makes no assumptions about the data you feed it. It shoudld accepts any type of data native to Sverchok.. Functionality is a subset of python list slicing, the stride parameter functionality isn't implemented.

Inputs

Input	Description
Data	The data - can be anything
Start	Slice start, allows negative python index
Stop	Slice stop, allows negative python index

Parameters

Level

It is essentially how many chained element look-ups you do on a list. If `SomeList` has a considerable *nestedness* then you might access the most atomic element of the list doing `SomeList[0][0][0][0]`. Levels in this case would be 4.

Start

Start point for the slice

Stop

Stop point for the slice.

Outputs

Slice, the selected slices. Other, the list with the slices removed.

Examples

Trying various inputs, adjusting the parameters, and piping the output to a *Debug Print* (or stethoscope) node will be the fastest way to acquaint yourself with the inner workings of the *List Item* Node.

Some slice examples. >>> l [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] >>> l[1:-1] [1, 2, 3, 4, 5, 6, 7, 8] >>> l[0:2] [0, 1] >>> l[-1:2] []

Notes

List Sort

Functionality

Sort items from list based on index. It should accept any type of data from Sverchok: Vertices, Strings (Edges, Polygons) or Matrix.

Inputs

Takes any kind of data.

Parameters

Level: Set the level at which to observe the List.

Outputs

Depends on incoming data and can be nested. Level 0 is top level (totally zoomed out), higher levels get more granular (zooming in) until no higher level is found (atomic). The node will just reverse the data at the level selected.

Examples

In this example the node sort a list a integers previously shuffled.

List Split

Functionality

Split list into chuncks. The node is *data type agnostic*, meaning it makes no assumptions about the data you feed it. It shoudld accepts any type of data native to Sverchok.

Inputs

Input	Description
Data	The data - can be anything
Split size	Size of individual chuncks

Parameters

Level

It is essentially how many chained element look-ups you do on a list. If `SomeList` has a considerable *nestedness* then you might access the most atomic element of the list doing `SomeList[0][0][0][0]`. Levels in this case would be 4.

unwrap

Unrwap the list if possible, this generally what you want. [[1, 2, 3, 4]] size 2. +———+——————–+ | Unwrap | Result | +========+===================+ | On | [[1, 2], [3, 4]] | +———+——————–+ | Off | [[[1, 2], [3, 4]]]| +———+——————–+

Split size

Size of output chuncks.

Outputs

Split

The list split on the selected level into chuncks, the last chunck will be what is left over.

Examples

Trying various inputs, adjusting the parameters, and piping the output to a *Debug Print* (or stethoscope) node will be the fastest way to acquaint yourself with the inner workings of the *List Item* Node.

List First & Last

Functionality

First and last items of data on some level

Inputs

Data - data to take items

Properties

level - leve to take first and last items

Outputs

Middl - all between first and last items **First** - First item **Last** - Last item

Examples

List Masks

List Mask Out

List Mask In

Functionality

This node use the mask list i.e. 1,0,0,0 as switch to mix two data list together.

0 means false, an item from the **Data False** will be appended to the output data;

1 will be considered as true (actually any value that evaluate as true in python), an item from the **Data True** will be appended to the output data. If the mask list is not long enough to cover all the inputs, it will be repeated as the mask for the rest of inputs.

Length of mask list affect output because every item (without Choice activated) corresponding to Inputs several times.

The main design reason behind this node is to be able to conditionally apply operations to one a masked list, for example select vertices based on location and move them or as shown below, select numbers and negate them.

Inputs

Mask: Input socket for mask list.

Data True: Input socket for True Data list.

Data False: Input socket for False Data list.

Parameters

Level: Set the level at which the items to be masked.

Choise: Make length of out list the same as length of input list

Outputs

Data: Mixed data of the incoming data, the length of Outputs depends on the **Data True** and **Data False** list length.

Example

List Mutators

List Modifier

This node offers an assortment of list modification functions. The node has both Unary and Binary modes.

- In Unary mode it will use the input of either sockets, it will use data1 first, then check data2
- If both are linked data1 is used.
- The node will draw the name of the current mode into the node header, useful for minimized nodes.

Behaviour

Modes	inputs	Behaviour Description

Set	unary	turns the valid input into a set
		`input = [0,0,0,1,1,1,3,3,`
		`↪3,5,5,5,6,7,8,4,4,4,6,6,`
		`↪6,7,7,7,8]`
		`output = [set(input)]`

Ordered Set by input	unary	only unique numbers but ordered by the original input sequence
		`input = [0,0,0,1,1,1,3,3,`
		`↪3,5,5,5,6,7,8,4,4,4,6,6,`
		`↪6,7,7,7,8]`
		`output = [0,1,3,5,6,7,8,4]`

Unique Consecutives	unary	no consecutive repeats
		`input = [0,0,0,1,1,1,3,3,`
		`↪3,5,5,5,6,7,8,4,4,4,6,6,`
		`↪6,7,7,7,8]`
		`output = [0,1,3,5,6,7,8,4,`
		`↪6,7,8]`

Sequential Set	unary	unique input values, ordered by their value
		`input = [0,0,0,1,1,1,3,3,`
		`↪3,5,5,5,6,7,8,4,4,4,6,6,`
		`↪6,7,7,7,8]`
		`output = [0,1,3,4,5,6,7,8]`

Sequential Set Rev	unary	unique input values, ordered by their value, reversed
		`input = [0,0,0,1,1,1,3,3,`
		`↪3,5,5,5,6,7,8,4,4,4,6,6,`
		`↪6,7,7,7,8]`
		`output = [8,7,6,5,4,3,1,0]`

Normalize	unary	scales down the values in the list to the range `-1.0 .. 1.0`

Accumulating Sum	unary	see `itertools.accumulate`
		`input =␣`
		`↪list(accumulate(range(10)))`
		`output = [0,1,3,6,10,15,`
		`↪21,28,36,45]`

Mask Subset	binary	generates a mask to indicate for each value in A whether it appears in B
		`A = [0,1,2,3,4,5,6,7]`
		`B = [2,3,4,5]`
		`output = [False, False,␣`
		`↪True, True, True, True,␣`
		`↪False, False]`

Intersection	binary	returns the set of items that appear in both A and B
Union	binary	returns the set of items A joined with B
Difference	binary	returns the set of items from A that don't appear in B

output as list

The boolean switch to *output as list* will be on by default, essentially it will wrap the output as a list because true sets don't have a defined order (which we do need most of the time).

Example

See the pullrequest for details : https://github.com/nortikin/sverchok/pull/884

also see the original thread : https://github.com/nortikin/sverchok/issues/865

Viz

3D View Properties

Features

Provides a convenient way to

- adjust the background colour and gradient of 3dview
- switch to show render-only geometry
- set world horizon colour
- set grid colour
- show parts of the grid
- set 3dview rotation types (Trackball | Turntable)
- has per settings per open 3dview, when new 3d views are added click on the node to trigger a redraw.

While working with Sverchok we often adjust various Blender settings, having this nodes allows you to configure 3dview without leaving NodeView.

Viewer BMesh

Functionality

aliases: BMeshViewer, BMView

Similar to ViewerDraw but instead of using OpenGL calls to display geometry this Node *writes* or *updates* Blender Meshes on every geometry update. The bonus is that this geometry is renderable without an extra bake step. We can use Blender's Modifier stack to affect the mesh. The only exception to the modifiers is the Skin Modifier but we aren't entirely sure why, maybe because BMview invalidates the BMesh between updates.

Inputs

- Verts
- Edges
- Faces
- Matrix

Parameters & Features

Features we rarely need or want to interact with are placed in the N-Panel / Properties Panel.

Location	Param	Description
Node UI	Update	Processing only happens if *update* is ticked
	Group	On by default, auto groups all meshes produced by incoming geometry
	Hide View	Hides current meshes from view
	Hide Select	Disables the ability to select these meshes
	Hide Render	Disables the renderability of these meshes
	Base Name	Base name for Objects and Meshes made by this node
	Select / Deselect	Select every object in 3dview that was created by this Node using Base Name
	Material Select	Assign materials to Objects made by this node
N Panel	Random Name	In the case of multiple BMview nodes, this button makes it easier to generate a new random name to prevent interference with existing Meshes. It will never produce and use the name of an existing Object, it will always append names with indices.
	Fixed Vert count	If you know the only change to the mesh is in vertex locations, then this toggle will use the foreach construct to overwrite the locations only, leaving existing edges and faces unchanged. Use it if you can.
	Smooth shade	Automatically sets *shade* type to smooth when ticked.

Outputs

Outputs directly to Blender `bpy.data.meshes` and `bpy.data.objects`

Examples

Viewer Index

Functionality

Displays indices of incoming geometry, much like what is possible in the debug mode of Blender. The individual indices of *Vertices, Edges and Faces* can be displayed with and without a small background polygon to help contrast the index numbers and the 3d view color.

Inputs

This node Accepts sets of *Verts, Edges, Faces, and Matrices*. In addition it accepts a Text input to display Strings at the locations passed in through the *Vertices* socket.

Parameters

default

parameters	type	description
show	bool	*activation* of the node
background	bool	display *background polygons* beneath each numeric (element of text)
verts, edges, faces	multi bool	set of toggles to choose which of the inputs are displayed.
Bake *	operator	bake text to blender geometry objects
Font Size *	float	size of baked text
Font *	string	font used to bake (import fonts to scene if you wish to use anything other than BFont)

** - only used for baking text meshes, not 3dview printing*

In the *Properties Panel* (N-Panel) of this active node, it is possible to specifiy the colors of text and background polygons.

extended

parameters	type	description
colors font	color	colors for vertices, edges, polygons
colors background	color	colors for vertices, edges, polygons background
show bake UI	bool	reveals extended bake UI features (Bake button, font properties)

We added a way to show extended features in the main Node UI.

font

With *show bake UI* toggled, the Node unhides a selection of UI elements considered useful for *Baking Text* in preparation for fabrication. If no font is selected the default BFont will be used. BFont won't be visible in this list until you have done at least one bake during the current Blender session.

Glyph to Geometry

Font glyph conversion is done by Blender. If it produces strange results then most likely the font's Glyph contains *invsibile mistakes*. Blender's font parser takes no extra precautions to catch inconsistant Glyph definitions.

Bake locations

Depending on the toggle set in `Verts | Edges | Faces`, the text can be shown and baked at various locations.

Mode	Location
Verts	directly on the vertex location (adjusted if Matrix is input too)
Edges	in-between the two vertices of the edge
Faces	the average location of all vertices associated with the polygon

Orientation of baked text

Currently only flat on the XY plane. `Z = 0`

Outputs

No socket output, but does output to 3d-view as either openGL drawing instructions or proper Meshes when Baking.

Examples

Viewer Draw MKII

destination after Beta: basic view

Functionality

Built on the core of the original ViewerDraw, this version allows all of the following features.

- Vertex Size (via N-Panel)
- Vertex Color
- Edge Width (via N-Panel)
- Edge Color
- Face shading: Flat vs Normal
- Vertex, Edges, Faces displays can be toggled.
- Defining Normal of Brigtness Source (via N-Panel)
- `Faux Transparancy` via dotted edges or checkered polygons.
- optional forced tesselation (via N-Panel)
- bake and bake all. (via N-Panel, show bake interface is not on by default)

draws using display lists

Uses OpenGL display list to cache the drawing function. This optimizes for rotating the viewport around static geometry. Changing the geometry clears the display cache, with big geometry inputs you may notice some lag on the intial draw + cache.

forced tesselation

Allows better display for concave polygons, by turning all faces with more than 4 verts into triangles. In this mode Edges will still draw the boundary of the original polygon and not the new tesselated polygon edges.

Inputs

verts + edg_pol + matrices

Parameters

Some info here.

Feature	info
verts	verts list or nested verts list.
edge_pol	edge lists or polygon lists, if the first member of any atomic list has two keys, the rest of the list is considered edges. If it finds 3 keys it assumes Faces. Some of the slowness in the algorithm is down to actively preventing invalid key access if you accidentally mix edges+faces input.
matrices	matrices can multiply the incoming vert+edg_pol geometry. 1 set of vert+edges can be turned into 20 'clones' by passing 20 matrices. See example

Outputs

Directly to 3d view. Baking produces proper meshes and objects.

Examples

development thread: has examples

Notes

Tips on usage

The viewer will stay responsive on larger geometry when you hide elements of the representation, especially while making updates to the geometry. If you don't need to see vertices, edges, or faces *hide them*. (how often do you need to see all topology when doing regular modeling?). If you see faces you can probably hide edges and verts.

System specs will play a big role in how well this scripted BGL drawing performs. Don't expect miracles, but if you are conscious about what you feed the Node it should perform quite well given the circumstances.

Texture Viewer

Functionality

This node allows viewing a list of scalar values and Vectors as a texture, very useful to display data from fractal, noise nodes and others, before outputting to a viewer_draw_mk2.

Uses OpenGl calls to display the data.

Inputs

Floats and Vectors input

Parameters

Feature	info
Float input	float and Vectors nested list
Show	may be *true* or *false*: display or not the texture
Pass	may be *true* or *false*: transfer data to the internal image viewer
Set texture display	choose the size of the texture to display: (64x64px,128x128px, 256x256px, 512x512px, 1024x1024px)
Set color mode	set the color mode: **BW** = grayscale image, **RGB** = image with red, green, blu channels **RGBA** = image with red, green, blu, alpha channels
Custom tex	may be *true* or *false*: enable custom size of texture
Width tex	must be *int*: set the width of the texture when Custom tex is enabled
Height tex	must be *int*: set the height of the texture when Custom tex is enabled

Outputs

Directly into node tree view in a blue bordered square or if you choose the `Pass` option the texture may be transfered to the internal image viewer/editor.

Properties panel

You can save the texture in the desired folder. You can choose the format:

##### jpeg, jp2, bmp, tiff, tga, tga(raw), exr, exr(multilayer), png

Save the texture clicking on the button `SAVE`. You can save also passing the image to the blender image editor with option `Pass`. This is much preferred because there are more saving options.

Examples

Basic usage:

Important notes

The `Texture viewer node` need adequate data size, this mean that number of input pixels should be equal to the output. If not you will receive an error. See the image below for an RGBA example:

Links

dev. thread: https://github.com/nortikin/sverchok/pull/1255 texture viewer proposal: https://github.com/nortikin/sverchok/issues/1248 Texture script by @ly29: https://github.com/Sverchok/Sverchok/issues/56

Text

Debug Print

Functionality

Prints raw socket data to console, useful for debug.

Inputs

Dynamic number of inputs named Data 0, Data 1, ... , Data N that will be printed in order if linked.

Parameters

Active or not, turn off printing of individual node. In the N-panel controll printing of inddividual sockets.

Outputs

None

Example of usage

Mostly useful for development. For usage see above.

Notes

Note that printing will be system console and not the blender console, for information about how to use the system console see blender documentation for your system.

GText

Functionality

Creates Text in NodeView using GreasePencil strokes. It has full basic English and Cyrillic Character map and several extended character types:

```
[ ] \ / ( ) ~ ! ? @ # $ % & ^ > < | 1234567890 - + * = _
```

Inputs

Gtext will display whatever text is currently in the system buffer / clipboard.

UI & Parameters

Node UI

Parameter	Function
Set	If clipboard has text, then Set will display that text beside the GText node.
Clear	This erases the GreasePencil strokes

GText Node will display the context of the clipboard after pressing the *Set* button. If no text is found in the clipboard it will write placeholder 'your text here'.

N-panel

Parameter	Function
Get from Clipboard	Gets and sets in one action, takes text from clipboard and writes to NodeView with GreasePencil
Thickness	sets pixel width of the GreasePencil strokes
Font Size	Scales up text and line-heights

Moving GText

To move GText strokes around in NodeView you must move the GText Node then press Set again. This may not be entirely intuitive but it hasn't bugged us enough to do anything about it.

Outputs

Outputs only to NodeView

Examples

Note

Functionality

Note node allow show custom text and convert custom text to format naming indeces of viewer_INDX. For last writing words separated with spaces will be i.e.: from **A B C Name index_** output will be **[['A'], ['B'], ['C'], ['Name'], ['index_']]**.

Inputs

Input	Description
Text_in	Input text to show or convert to mark indeces

Parameters extended

Param	Type	Description
text	String	Line to write custom text, will be shown in node
show_text	Bool, toggle	Show text line string field on node
Output socket	Bool, toggle	Use or not output socket
Input socket	Bool, toggle	Use or not input socket
From clipboard	Button	Use data, stored in clipboard to fill the node
To text editor	Button	Sent text to text editor

Outputs

Output	Description
Text_out	Output text formatted to INDX viewer

Examples

Using hidden power of note node - output socket to name indeces in INDX viewer for any part. Vertices called as strings, can be any text.

Stethoscope

destination after Beta: basic data

Functionality

The Node is designed to give a general sense of the connected data stream. After a short preprocessing step Stethoscope draws the data directly to the Node view.

The processing step

The first and last 20 sublists will be displayed. The data in between is dropped and represented by placeholder ellipses.
Float values are rounded if possible.

Inputs

Any known Sverchok data type.

Parameters

Currently a *visibility* toggle and *text drawing color* chooser.

Outputs

Direct output to Node View

Examples

Notes

Implementation is `POST_PIXEL` using `bgl` and `blf`

Text In

Functionality

Import data from text editor in formats csv, json or plain sverchok text.

Properties

Select - Select text from blender text editor **Select input format** - Property to choose between csv, plain sverchok and json data format

> **csv: Header fields** - to use headers from file **Dialect** - to choose dialect of imported table
>
> **Sverchok: Data type** - output data socket as selected type

Load - Load data from text in blend file

Outputs

vertices, **data**, **matrices** - if sverchok plain data selected

Col - if csv data selected

Random - if json data selected

Text Out

Functionality

Inserting and outputting data to text editor with preformatting in csv, json or plain sverchok text.

Inputs

Col - for csv type column definition multysocket **Data** - for sverchok single socket **Data** - for json multysocket

Properties

Select - Select text from blender text editor **Select output format** - Property to choose between csv, plain sverchok and json data format

> **csv: Dialect** - to choose dialect of table
>
> **Sverchok and json: Compact** and **Pretty** - overall view of data presented readable or not so much readable

Dump - sent your data to text editor to selected text **Append** - to add text not deleting old text

Viewer Text

Functionality

Looking for data and convert to readable format as:

node name: Viewer text

vertices: (1) object(s) =0= (8) [0.5, 0.5, -0.5] [0.5, -0.5, -0.5] ...

data: (1) object(s) =0= (12) [0, 4] [4, 5] ...

matrixes: (1) object(s) =0= (4) (1.0, 0.0, 0.0, 0.0) (0.0, 1.0, 0.0, 0.0) (0.0, 0.0, 1.0, 0.0) (0.0, 0.0, 0.0, 1.0)

Where **(1) object(s)** means that we have 1 object **=0= (8)** means first (zero is first) object consists of 8 lists if you add sublevels, there will be additional level like **=0= (1)** as vertices in separated sphere or plane generator gives. **[0.5, 0.5, -0.5]** means vector or other data

Inputs

Input	Description
Verts	Vertices
Edges/Polygons	Edges or Polygons data. Node understand what is linked
Matrix	Matrices data
Object	Object data

Parameters & Features

VIEW button will send formatted data to text editor, you have manually open text file called Sverchok_viewer, but after this it will be updated and in upper of text will be name of your node to identify it.

Examples of usage

Scene

Dupli Instances

Functionality

This node exposes the functionality of the Duplication types `VERTS` and `FACES` to the Sverchok node tree. The Node works in two ways. One mode accepts just Locations, the other mode accepts just Matrices.

Features	Description
Locations	the node generates a proper blender mesh internally, based on vertices. The duplication is set to VERTS.
Matrices	the node generates a vertex+face mesh using the transforms contained in individual matrices. First it makes a unit 1 triangle, then multiplies the vertex coordinates with a matrix. This is done for each of the passed matrices. Passing 4 matrices, makes 4 triangles : a total of 12 verts and 4 faces. The duplication is set to FACES.
Child Object	You pick the child Object from the UI.
Parent Object	(not exposed to the UI) , this is generated internally from the Locations or Matrices socket data

The name of the internal parent object in this example is 'booom' , but this can be changed and should probably be node specific.

Inputs

Input	Description
Locations	Vertices, coordinates, vectors, 3tuples, 3lists
Matrices	full on 4*4 transform matrices (but scale is converted to uniform)

Parameters

The only parameter is the Object selection, it needs to duplicate something

Limitations

It's worth mentioning that because the faces duplication relies on the area of the triangle to determin the scale, that the scale is a scalar, and therefor uniform (x,y,z are scaled equally).

Examples

More info: https://github.com/nortikin/sverchok/issues/740

Frame Info

Functionality

Give the node graph access to the frame information. Some transport controls have been added since these images were made.

Inputs

None

Parameters

None

Outputs

- Current Frame,
- Start Frame,
- End Frame
- Evaluate , this generates a value in the range (0.0 .... 1.0), and represents the position of the 'player head' in relation to the total amount of frames to play.

all wrapped to standard level.

Example of usage

Usage with Map Range node to give percentage of current frame range.

Get Property / Set Property

Functionality

These nodes can be used to control almost anything in Blender once you know the python path. For instance if you want to `Set` the location of a scene object called `Cube`, the python path is `` `bpy.data.objects['Cube'].location` ``.

By default these nodes don't expose their input / output sockets, but they will appear if the node was able to find a compatible property in the path you provided. The Socket types that the node generates depend on the kind of data path you gave it. It knows about Matrices and Vectors and Numbers etc..

There are also convenience aliases. Instead of writing `` `bpy.data.objects['Cube'].location` `` you can write `` `objs['Cube'].location` `` . The aliases are as follows:

```
aliases = {
    "c": "bpy.context",
    "C" : "bpy.context",
    "scene": "bpy.context.scene",
    "data": "bpy.data",
    "D": "bpy.data",
    "objs": "bpy.data.objects",
    "mats": "bpy.data.materials",
    "meshes": "bpy.data.meshes",
    "texts": "bpy.data.texts"
}
```

Input

In `Set` mode

Input	Description
Dynamic	Any of the Sverchok socket types that make sense

Output

In `Get` mode

Output	Description
Dynamic	Any of the Sverchok socket types that make sense

Parameters

The only parameter is the python path to the property you want to set or get. Usually we search this manually using Blender's Python console.

Limitations

(todo?)

Examples

Using aliases `objs` instead of `bpy.data.objects`

Object In

Functionality

Takes object from scene to sverchok. Support meshed, empties, curves, NURBS, but all converting to mesh. Empties has only matrix data. Than sorting by name. If you write group of objects to group field, all object in signed group will be imported. It understands also vertes groups, when activated, showing additional socket representing indexes, that you can use to sort or mask edges/polygons. or do any you want with vertex groups. All groups cached in one list, but without weight.

Inputs

None

Parameters

Param	Type	Description
G E T	Button	Button to get selected objects from scene.
group	String	Name of group to import every object from group to Sverchok
sorting	Bool, toggle	Sorting inserted objects by name
post	Bool, toggle	Postprocessing, if activated, modifiers applyed to mesh before importing
vert groups	Bool, toggle	Import all vertex groups that in object's data. just import indexes

Outputs

Output	Description
Vertices	Vertices of objects
Edges	Edges of objects
Polygons	Polyfons of objects
Matrixes	Matrices of objects
_Vers_grouped_	Vertex groups' indeces from all vertex groups

Examples

Importing cobe and look to indeces.

Object Remote

This node is a convenience node.

Its features are very limited:

- pick an object with the picker
- add vector sockets to control Location / Scale / Rotation (Euler in Rads)

That's it.

Implementation details

It's a nice short node, one thing it does to avoid setting your Object's scale to 0,0,0 is to detect if 0,0,0 is passed. If 0,0,0 is passed it turns it into 0.00001, 0.00001, 0.00001. The reason for this is that an Object's scale can not be set to 0,0,0 in Blender, it seems to wreck the internal transform matrix.

Objects

Weights

Functionality

automatically creates a group of vertices and allows you to assign each vertex weight in many different ways

Input sockets

vertIND - Connect here a list of needed vertex indexes, or node automatically creates a list of indexes of all vertices of the object

weights - vertex weights (floats less than 0.0 count as 0.0, bigger than 1.0 count as 1.0)

Parameters

clear unused - (on side panel) zero weights for all vertices which is not indexed in the list of indexes

object name - name of object to create vertex group for.

Usage

BPY data

Object ID Selector

Functionality

Has the ability to select items from **bpy.data.**. Some data types have more options than others. Currently the extended options are available for Images, Objects and Grease Pencil.

- Images : lets you pick a name, and get the flattened pixels (by ticking **pass pixels**)
- Objects : lets you pick by name, or leave the name blank and you'll get the list of objects for the selected type
- Grease Pencil : you can also pick a layer by name or leave blank, if you pick by name you'll get the option to pick **active_frame** for that layer or available frames. Ticking **pass_points** will pass

Example of usage

See the development thread: https://github.com/nortikin/sverchok/issues/1379#issuecomment-287331274

Layout

Socket Converter

Functionality

Converts sockets types if something go wrong or your node too alpha

Inputs

data

Outputs

vertices, **data**, **matrices**

Wifi In & Out

Functionality

Create a invisble noodle, useful for keeping layout clean for example constants that are resued in many place.

Concept

A named Wifi Input node can be listened to by any number of Wifi Output nodes. A Wifi Output node needs to be linked to a specific Wifi Input node useing the dropdown list.

Inputs

In the Wifi In node there are N inputs named after variable name.

Outputs

In a linked Wifi Out there are N-1 output of matching type.

Notes

Variable names for Wifi Input nodes need to be unique.

Sharing of data is at the moment only possible within one layout.

The virtual noodle has a small overhead that is small enough that it can ignored for most practical scenarios. In the future even this should disappear.

Wifi In & Out

Functionality

Create a invisble noodle, useful for keeping layout clean for example constants that are resued in many place.

Concept

A named Wifi Input node can be listened to by any number of Wifi Output nodes. A Wifi Output node needs to be linked to a specific Wifi Input node useing the dropdown list.

Inputs

In the Wifi In node there are N inputs named after variable name.

Outputs

In a linked Wifi Out there are N-1 output of matching type.

Notes

Variable names for Wifi Input nodes need to be unique.

Sharing of data is at the moment only possible within one layout.

The virtual noodle has a small overhead that is small enough that it can ignored for most practical scenarios. In the future even this should disappear.

Network

Beta Nodes

Set_dataobject

destination after Beta:

Functionality

It works with a list of objects and a list of Values

multiple lists can be combined into one with the help of ListJoin node

MK2 version can work with multiple nested lists

When there is only one socket Objects- performs **(Object.str)**

If the second socket is connected Values- performs **(Object.str=Value)**

If the second socket is connected OutValues- performs **(OutValue=Object.str)**

Do not connect both the Values sockets at the same time

First word in str must be property or metod of object

Use i. prefix in str to bring second property of same object

Inputs

This node has the following inputs:

- **Objects** - only one list of python objects, like **bpy.data.objects** or **mathutils.Vector**
- **values** - only one list of any type values like **tupple** or **float** or **bpy.data.objects**

Outputs

This node has the following outputs:

- **outvalues** - the list of values returned by the **str** expression for each object

Examples of usage

Alpha Nodes

CHAPTER 5

Contribute

Our workflow:

1. Freedom to code, only try to follow pep8, and avoid abuse.
2. Agile software development - look http://en.wikipedia.org/wiki/Agile_software_development here.
3. Ideas from other addons to implement to sverchok. It is big deal.
4. If you making something it is better to say in issues before.
5. Brainstorming and finding solutions. We are mostly new in python and programming, we are artists.

What not to do:

Doing these things will break old layouts and or create very ugly code in the node.

1. Change .bl_idname of a node
2. Remove or rename sockets
3. Adding new socket without updating upgrade.py
4. 'current_mode' names of properties are reserved for nodes, not use for anything else

Helpful hints

In blender console you can easily inspect the sverchok system by writing:

```
import sverchok
print(dir(sverchok.nodes))
```

To create a node:

1. Make a scripted node to test the idea.
2. Show your node to us in an issue or silently create branch or fork of master in github. If it is huge complex job we can make you collaborator.
3. Copy an existing node that is similar.
4. Change class name, class id, class description, class registration in your file
5. Add node id in menu.py in an approritate category.
6. Add file to category that you need
7. Add in `nodes/__init__.py` filename
8. Tell us to merge branches/forks

SOME RULES:

1. All classes that are subclasses of blender classes - have to have prefix Sv, ie SvColors.
2. node_tree.py contains base classes of node tree and, maybe you need to create collection property or new socket (for new type of socket tell us first), you make it here.
3. data_structure.py has magic of:
   (a) update definitions, that works pretty well with sockets' and nodes' updates
   (b) some bmesh functiones
   (c) cache – operates with handles – we use it to operate with nodes' data, it is dictionary, with node'name, node-tree'name and it's data
   (d) **list levels definitions - you must yse them:**
      i. data correct makes list from chaotic to 4th levels list, seems like [[[floats/integers]]] and not [[f/i]] or [[[[f/i]]]].

         usage: dataCorrect(data, nominal_dept=2), where data - is your list, nominal_depth - optioal, normal for your' case depth of list

      ii. data spoil - make list more nasty for value of depth

         usage: dataSpoil(data, dept), where data - your list, depth - level of nestynessing. Gives list from [] to [[]] etc

      iii. Levels of list - find level of nestiness.

         usage: levelsOflist(list) - returns level of list nestiness [[]] - is 2, [] - 1, [[[]]] - 3

   (e) matrix definitions - to make something with matrices/vertices
      i. Matrix_generate(prop) - make from simple list real mathutils.Matrix(()()()())
      ii. Matrix_listing(prop) - make from mathutils.Matrix() simple list like [[][][][]] with container is [[[][][][]]]
      iii. Matrix_location(prop, list=False) - return mathutils.Vector() of matrix' location
      iv. Matrix_scale(prop, list=False) - the same, but returns matrix' scale vectors

v. Matrix_rotation(prop, list=False) - return rotation axis and rotation angle in radians value as tuple (Vector((x,y,z)),angle)

vi. Vector_generate(prop) - makes from simple list real mathutils.Vector(), as Matrix generate def

vii. Vector_degenerate(prop) - as matrix listing, it degenerate Vectors to simple list

viii. Edg_pol_generate(prop) - define wether it is edges or polygons in list, and terurns tuple as (type,list)

ix. matrixdef(orig, loc, scale, rot, angle, vec_angle=[[]]) - defines matrix

(f) list definitions:

i. fullList(l, count) - makes list full till some count. last item multiplied to rest of needed length, ie [1,2,3,4] for count 6 will be [1,2,3,4,4,4]

ii. match_short(lsts) Shortest list decides output length [[1,2,3,4,5], [10,11]] -> [[1,2], [10, 11]]

iii. match_cross2(lsts) cross rference [[1,2], [5,6,7]] ->[[1, 2, 1, 2, 1, 2], [5, 5, 6, 6, 7, 7]]

iv. match_long_repeat(lsts) repeat last of shorter list [[1,2,3,4,5] ,[10,11]] -> [[1,2,3,4,5] ,[10,11,10,11,10]]

v. match_long_cycle(lsts) cycle shorts lists [[1,2,3,4,5] ,[10,11]] -> [[1,2,3,4,5] ,[10,11,10,11,10]]

vi. repeat_last(lst) creates an infinite iterator that repeats last item of list, for cycle see itertools.cycle

vii. some others to operate with exact nodes

(g) update sockets - definitions to operate with update

(h) changable type of socket - makes possible to use changable socket in your node - it calling

usage:

i. node has to have self veriables:

ii. **and in update:**

- inputsocketname = 'data' # 'data' - name of your input socket, that defines type
- outputsocketname = ['dataTrue','dataFalse'] # 'data...' - are names of your sockets to be changed
- changable_sockets(self, inputsocketname, outputsocketname)

(i) multi-socket multi_socket(node,min=1) - as used by ListJoin, List Zip, Connect UV

- multi_socket(node,min=1)
- base_name = 'data'
- multi_socket_type = 'StringsSocket'
- setup the fixed number of socket you need, the last of them is the first multi socket. minimum of one.
- then in update(self):
  - multi_socket(self, min=1, start=0, breck=False) - [where min - minimum count of input sockets;
  - start - starting of numeration, could be -1, -2 to start as in formula2 node; breck - to make breckets, as used in formula2 node]
- for more details see files mentioned above

4. **Utils** folder has:

(a) CADmodule - to provide lines intersection

(b) IndexViewerDraw - to provide OpenGL drawing of INDXview node in basics

   (c) sv_bmeshutils - self say name

   (d) sv_tools - it is toolbox in node area for update button, upgrade button and for layers visibility buttons, also update node and upgrade functional to automate this process.

   (e) text_editor_plugins - for sandbox node scripted node (SN) to implement Ctrl+I auto complete function

   (f) text_editor_submenu - templates of SN

   (g) upgrade - to avoid breaking old layouts. Defines new simplified interface override. if you change some property in def draw_buttons() than just bring new properties here to avoid break old layout

   (h) viewer_draw - for draw and handle OpenGL of Viewer Draw node (have to be remaked)

   (i) voronoi - for delaunai and voronoi functions of correspond nodes

5. **Node scripts** folder for every template for SN (see utils-e.)

6. **Nodes** folder for categorized nodes. not forget to write your nodes to init.py there

7. **To use enumerate property you have to assign index to items, never change the index of items added,** it will break if you add more functions.

8. Not make many nodes if you can do less multifunctional.

9. Use levels, findout how it works and use level IntProperty in draw to define what level is to operate. We operate with 1,2,3 - standart and additional 4... infinity. make sure, that your levels limited, dropped down by levelsOflist as maximum value

10. Keep order in node' update definition as if output: if input. To count input only if you have output socket assembled.

11. Look at todo list to know what is happening on and what you can do. use your nodes and test them.

12. There is no reason to auto wrap or make less levels of wrapping, than needed to proceed in other nodes. So, for now canonical will be [[0,1,2,3]] for simple data and [[[0,1,2,3]]] for real data as edge, vertex, matrix other cases may be more nasty, but not less nesty and wrapping need to be grounded on some reasons to be provided.

13. Do not use is_linked to test if socket is linked in `def update(self)`, check links. In `def process(self)` use `.is_linked`

14. to **CHANGE** some node, please, follow next:

   (a) Put old node file to ../old_nodes add the corresponding bl_idname in __init__.py in the table. (there is README file also);

   (b) Make new changed node as mk2(3,4...n) and place to where old node was placed with all changes as new node, change name and bl_idname (look 'To create a node:' in current instructions).

CHAPTER 6

Panels of Sverchok

Node Tree Panel

This panel allows to manage sverchok layouts as easy as you press buttons on an elevator.

Update

Update	description
all	Updates all trees of sverchok.
Update layout	Updates currently active layout

Layout manager

Box to quickly pick layout, switch between them with buttons instead of popup list. Also have settins:

Button	Function
B	bake this layout - will gather all layout's viewer draw and viewer text to bake them. Bake only if bakeable button is active on node, else ignore.
Show layout	Controlls all OpenGL viewer of this layout. Viewer, Stethoscop and Viewer Indices
Animate layout	to animate the layout (or not) - may preserve you time.
Process layout	Automaticlly evaluate layout while editing, disable for large or complex layouts
Fake User	Sets fake user so layout isn't deleted on save

Check for updates

Check for updates - finds if master branch on github has new version of sverchok. In future there will be releases, but now dangerouse update.

Upgrade Sverchok - upgrades Sverchok from github with new version - button appears only if 'check for updates' finds a new version.

3D Panel

With this panel your layout becomes addon itself. So, you making your life easy.

Scan for props

When layout is in, check for next nodes to embad them as properties:

- float node
- int node
- object in node

Sorting them by label, that user defined in node tree panel or if no label, the name of the node is used.

Update all

Forces update of all layouts.

Clean

Button to remove sverchok layouts, that has not users (0)

hard clean - boolean flag to remove layouts even if it has users (1,2...), but not fake user (F). Fake user layout will be left.

Clean layouts - remove layouts. Button active only if no node tree windiw around. Better to make active layout nothing or fake user layout to prevent blender crash. Easyest way - activate your Faked user layout, on 3D press **ctrl+UP** and press button. than again **ctrl+UP** to go back. No wastes left after sverchok in scene.

Use with care.

Properties

Layouts by box. Every layout has buttons:

Button	Function
B	bake this layout - will gather all layout's viewer draw and viewer text to bake them. Bake only if bakeable button is active on node, else ignore.
Show layout	show or hide all viewers - to draw or not to draw OpenGL in window, but bmesh viewer not handled for now.
Animate layout	to animate the layout (or not) - may preserve you time.
P	Process layout, allows safely manupilate monsterouse layouts.
F	Fake user of layout to preserve from removing with reloading file or with **clean layouts** button.

Properties has also gathered values:

floats and integers - digit itself, maximum and minimum vaues.

object in - button for object in node to collect selected objects.

Import Export Panel

location: N panel of any Sverchok Tree.

Import and export of the current state of a Sverchok Tree. This tool stores

- Node state: location, hidden, frame parent
- Node parameters: (internal state) like booleans, enum toggles and strings
- connections and connection order (order is important for dynamic-socket nodes)

Export

feature	description
Zip	When toggled to *on* this will perform an extra zip operation when you press Export. The zip can sometimes be a lot smaller that the json. These files can also be read by the import feature.
Export	Export to file, opens file browser in blender to let you type the name of the file, Sverchok will auto append the .json or .zip file extention - trust it.

Import

feature	description
Layout name	name of layout to use, has a default but you might want to force a name
Import	import to new layout with name (described above). Can import directly from zip file if there is only one .json in the zip. Warning to the descerned reader, only import from zip if the source is trusted. If you are not sure, resist the urge and take the time to learn a little bit about what you are doing.

Warnings

Consider this whole IO feature experimental for the time being. You use it at your own risk and don't be surprised if certain node trees won't export or import (See bug reporting below). The concept of importing and exporting a node tree is not complicated, but the practical implementation of a working IO which supports dynamic nodes requires a bit of extra work behind the scenes. Certain nodes will not work yet, including (but not limited to) :

Node	Issue
Object In	the json currently doesn't store geometry but an empty shell without object references instead
SN MK1	currently this auto imports by design, but perhaps some interruption of the import process will be implemented

Why make it if it's so limited?

Primarily this is for sharing quick setups, for showing people how to achieve a general result. The decision to not include geometry in the Object In references may change, until then consider it a challenge to avoid it. The way to exchange large complex setups will always be the `.blend`, this loads faster and stores anything your Tree may reference.

While importing I see lots of messages in the console!

Relax, most of these warnings can be ignored, unless the Tree fails to import, then the last couple of lines of the warning will explain the failure.

Bug Reporting

By all means if you like using this feature, file issues in this thread. The best way to solve issues is to share with us a screenshot of the last few lines of the error if we need more then we will ask for a copy of the *.blend*.

Groups Panel

Crete a node group (Monad) from selection. It can have vectorized inputs, adding or removing sockets. Sverchok groups is a beta feature, use a your own risk and please report bugs. Also while it is in beta old node groups may break. Bug reports.

Templates in menu panel of nodes area

You can use embedded templates in Sverchok. They are stored in json folder as jsons for import to Sverchok.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search