1 Welcome to The Golog high level API!

Welcome to the Golog High Level API! In this pdf, we will detail some high level functionality for controlling gologs and their displays. Hopefully, you have arrived at this PDF through an exploration of the documentation.golog. In which case, the graphical ontology this pdf resides in will give you an overall structure of the program.

By the end of this pdf, you should be able to do the following:

- Create and manipulate simplicial sets
- Create and manipulate gologs
- Create user interaction modes in mode_head
- Output graphics into a Window
- Save and Load gologs

2 Panda3D

Besides abstract simplicial sets, every part of the golog program relies upon the graphics engine panda3D, in particular, the actual rendering pipeline provided by a ShowBase object. Every bit of graphics is represented by a nodepath in a scene graph, even the cameras. It is highly suggested that the high level user understands at least the scene graph before creating scripts, although theoretically it is not necessary. If you wish to create more low-level scripts to control graphics, you will absolutely need to understand panda3D. To this end, it is suggested that one reads the panda3D manual, and attempts some tutorials:

Panda3D Manual
Scene Graph
Fireclaw Panda3D Tutorial Book

2.1 Running a Script with a ShowBase

If you just want to get into controlling golog through this API, then the only thing you NEED to have is a running ShowBase, within that showbase, you can run any script you'd like. A great example of initiating a ShowBase, is the run.py file in the golog program itself. The basic Components are below:

```
from direct.showbase.ShowBase import ShowBase

class runner(ShowBase):
    def __init__(self):
        ShowBase.__init__(self)
        self.script(*args,**kwargs)

    def script(self,*args,**kwargs):
        #put whatever script you may want here
        pass

r = runner()
r.run()
```

This code allows one to use panda3D's rendering pipeline given by

$$\text{base} == \text{self}$$

For example, in order for golog to load graphics from a file, it will call base.loader.loadModel(file_path). Any script that you run in the context of panda3D's graphics engine, must be run inside the 'runnner' class created above, and you will likely have to pass 'base'

3 sSet

The fundimental underlying mathematical structure in golog is that of a simplicial set. One does not need to understand the mathematics of a simplicial set in order to use golog, the only key ideas are:

- 0-simplecies
- 1-simplecies between two 0-simplecies
- Math_Data associated to simplecies

From this point of view, a simplicial set is just a graph database, however there are higher simplecies which the advanced user can utilize.

an abstract simplicial set does not require the usage of panda3Ds showbase.

3.1 Implementing / Controlling a simplicial set

to create a simplecial set, first import simpSet from the hcat.py module and instanciate the simpSet class

```
from hcat import simpSet

sSet = simpSet(label = 'hello ontology!')
```

to create a simplex in the simplicial set call the add function

```
simplex = sSet.add(ob, label = 'A simplex')
```

In this case, ob can be one of the following:

if ob is an integer n:
create a new simplex of level n and all of the subsimplecies that might support it

for example:
- sSet.add(0) creates a single 0-simplex
- sSet.add(1) creates a 1-simplex, and two 0-simplecies to act as it's faces

ob is a tuple of simplecies:
create a new simplex who's faces are the simplecies in the given tuple

for example:

```
simp0 = sSet.add(0, label = 'A')
simp1 = sSet.add(0, label = 'B')
f = sSet.add((face1,face0), label = 'f:A --> B')
```

if ob is an entire simplicial set:
copy entire simplicial set into this simplecial set.

To list all the simplecies in a simplicial set
call sSet.rawSimps, for example:

```
[simp.label for simp in sSet.rawSimps]
```

will return the labels of every simplex in the simplicial set.

3.2 Math_Data

To every simplex in a simplicial set, is some associate "Math_Data". There is nothing inherently mathematical about this data, the name is meant to differentiate from other "Data" in the golog program, such as "Graphics_Data" and "Export_Data". The Math_Data class is meant to be a meta wrapper for arbitrary data, with information on what type of data it is, and how to delete it. For the high level programmer, all that you need to know is how to instanciate a Math_Data. For this, we must import the Math_Data class from hcat.py

```
from hcat import Math_Data
```

Then we can instanciate a new math_data object:

```
math_data = Math_Data(type = string, math_data = actual_data)
```

to associate a math_data with a simplex, you can pass it as a kwarg upon creating a simplex, or add it directly:

```
simp = sSet.add(0, label = 'simplex with data', math_data = math_data)
simp.math_data = math_data
```

for example, one can create a math_data whose data is a simplicial set:

```
from hcat import simpSet, Math_Data

subsSet = simpSet(label = 'subontology')
sSet = simpSet(label = 'highest level ontology')

subsSet_math_data = Math_Data(type = 'simpSet', math_data = subsSet)
s1 = sSet.add(0, label = 'subontology simplex', math_data = subsSet_math_data)
s2 = sSet.add(0)
s2.math_data = subsSet_math_data
```

to get the sub simplicial set housed under the simplex we created above, we can just call the math_data attribute:

```
subsSet_reference = s1.math_data()
```