The Framework

The "framework" of the `soy_import.py` script consists of the `ObjectFinder` and the `ObjectReader` class, as well as the `ReadData` class, which all the `Read*` classes inherit from.

The `ObjectFinder` class

The point of the `ObjectFinder` class, is to be able to simply request a specific object, and then have the `ObjectFinder` take care of searching after, and finding it amongst all the .soy files. It simply works on the file level, instantiating `ObjectReader`s, and then letting them take care of finding the objects inside the files.

The `ObjectFinder` class contains two methods: `fetchReader` and `findObject`. When it is instantiated, the `__init__` method searches through, and indexes all the files that could be subject for search. This happens in three levels:

1. The original file, opened from the interface.
2. All the files of the directory of the original files, recursively.
3. All the files of the parent directory, recursively.

Afterwards, `findObject` will then be able to search these files with that priority. `findObject` uses `fetchReader` to make sure we do not instantiate an `ObjectReader` class more than once. When the files are searched, the instantiated `ObjectReader` classes are stored in the `self.readerdict` dictionary, and they can be reused if needed.

The `ObjectReader` class

The role of the `ObjectReader` is to read a file, index every .soy object in that file, and be able to read specific objects from it. It contains a list for each object type, containing all the instantiated `Read*` classes of that type. A dictionary `self.ObjectTypes` of the object types is also present. The `self.indexFile` method indexes and stores the instantiated `Read*` class of each object.

The `self.readNextHeader` method simply reads the next header in the file and returns it. And `getObject` finds the right `Read*` class instance in its lists and returns it. `getNextObject` is deprecated and unused. It should probably be removed. It stems from a time when we thought it might be good to read all the object sequentially. As it turns out, that was not a particularly good approach, since the dependencies between objects are anything but linear.

The `ReadData` class

The role of `ReadData` is to provide a few universal functions for all the other `Read*` classes to inherit. These are `self.__call__` and `self.readNextBlock`. `self.__call__` simply loops, calling `self.readNextBlock` until it has read all the way through the object.

`readNextBlock` reads the header of the next block and calls the `self.acceptblock` method. All `Read*` classes must contain this method. `self.acceptblock` receives a block ID and size as arguments, and then it is up to each `Read*` class to define how to read a block of that ID.

The `Read*` classes

All `Read*` classes have a number of attributes they should all have:

- `self.soyfile`: File object. The file this .soy object is contained in.
- `self.name`: String. The name of this object.
- `self.size`: Integer. Size of the data of this object.
- `self.datapos`: Integer. Position of first byte of the object data in this file.
- `self.amountread`: Integer. Amount of data that has been read from this object.
- `self.type`: Integer. Type ID of this object.
- `self.Blocks`: Dictionary. A 2-tuple containing the inner class for the block data, and the object containing either one or more of these classes.

The `self.Blocks` attribute adds the additional requirement that there should be an inner class for each block type. Currently, this is ordered so that even if you have a block with a `number_of` attribute, the block class defines only one instance of the information that can be in a block. So you have to determine in `acceptblock` if you are dealing with a `number_of` block or not. This may be inconvenient. And in fact, if this requirement was removed, it might even be possible to move the `acceptblock` method into `ReadData`.

The Interface

The interface is handled by the `ImportDialog` class, and the principles are much the same as `ExportDialog` in the `soy_export.py` script, with `DialogGrid` as the helper class for defining another coordinate system, and `self.DrawDialog`, `self.KeyEvent` and `self.ButtonEvent` as the three methods passed to `Blender.Draw.Register`.

When an import file has been set, you get a `Blender.Draw.Menu` with a choice of which type of data you want to import from it. When that has been chosen, the menu calls either `importMesh`, `importEntity` or `importNode`, depending on the type. These functions then calls a `Blender.Draw.PupMenu` which lists all the objects of that type from the file. Last in the chain of these type-specific methods are `fetchNMesh`, `fetchEntObject` and `fetchNodeObject`.

Aside from being called from the `import*` methods, these `fetch*` methods also calls each other in a sort of hierarchy. `fetchNodeObject` calls itself recursively and also calls `fetchEntObject`. And `fetchEntObject` calls `fetchNMesh`. Each `fetch*` methods always checks if a Blender Object of that name already exists, and if it does, it returns that Object instead of creating a new one. This is especially useful in `fetchNMesh`, though `fetchEntObject` and `fetchNodeObject` rarely needs it.