The Framework

The "framework" of the `soy_export.py` script consists in part of a few parent classes and some conventions for some groups of classes. Namely the `WriteData` class, which is the parent class of all the `Write*` classes. And both the `Write*` classes and the `Grab*` classes have a number of conventions that should be upheld.

The `WriteData` class

The `WriteData` class contains two methods, `self.accumulate` and `self.write`.

The `self.accumulate` method assume the existence of the attributes `self.accumulated`, `self.objecttype`, and `self.Blocks`.

`self.accumulate` is responsible for accumulating the data of a single block into a string. Given a 2-tuple containing the name of the blocktype, and the data to be accumulated, it finds the correct method for packing the data in the `self.Blocks` dictionary, runs the data through this method, and adds it to the `self.accumulated` string with the appropriate block header, created from the `self.objecttype` attribute, the `blockid` value extracted from `self.Blocks` and the length of the data string, indicating the size of the block.

The `self.write` method assume the existence of the attributes `self.file`, `self.name`, `self.objecttype`, `self.version`, and `self.accumulated`.

`self.write` is responsible for writing the accumulated data to the file object contained in `self.file`. From the above attributes and the length of `self.accumulated`, it writes the appropriate object header and the string containing all the object data from `self.accumulated`.

`WriteData` also contains a number of methods used by `self.Blocks` for signifying how to write the different block types. Some of these methods are used by several `Write*` classes, and some are used by only one. There may be room for some improvement in this respect.

The `Write*` classes

All `Write*` classes inherit from `WriteData` and are responsible for initializing and containing all the data the `WriteData.accumulate` and `WriteData.write` methods need. Each `Write*` class contains the following:

- `self.Blocks`: Dictionary. Each key is the name of a block type. Each item is a 2-tuple containing the block ID number and the function used for accumulating/packing the data of that block.
- `self.name`: String. Initialized on instantiation. The name of the object itself.
- `self.objecttype`: Integer. The object type ID.
- `self.version`: 2-tuple of Integers. The major and minor version of this block.
- `self.file`: File object. Initialized on instantiation. The file to write to.
- `self.accumulated`: String. Data accumulated for this object.

Each `Write*` class is instantiated and used by one or more `Grab*` classes.

The `Grab*` classes

The `Grab*` classes unfortunately don't always follow strict guidelines on "how to do stuff". Each class may have some of its own oddities. But I will try to explain some of the requirements and main purposes of these classes here.

The main purpose of a `Grab*` class is to obtain, sometimes convert, and then store the information of a .soy object in a way that makes it easy to write by the help of a `Write*` class. In general, you could say that a `Grab*` class should contain a number of `self.Grab*` methods made to "Grab" the information for each block. Then the `self.__init__` and `self.__call__` method of the class will call these `Grab*` methods as appropriate.

The `self.__call__` method should be used when you are finished treating the data externally, for instance, and want to write the data to the file. Because in some cases, the data of one object is dependent on the data of other objects.

The `ConjureNode` class

Take `ConjureNode`, for instance. This is a bit of a special case class. Technically, it is in the group of `Grab*` classes, but `ConjureNode` never retrieves any data directly from the blender API. This is because first, the Blender API does not contain the notion of a "Node" in the traditional sense. Blender uses a "parent"/"child" structure which can be used, though. However, secondly, a blender "parent" object does not know any of its "children" objects. This makes it impossible to decide whether a blender Object is a "parent" until we have already iterated through all Objects once.

`ConjureNode` creates a `ReadNode` object from an `ReadEntity` which has been discovered to have children. It will then add the `ReadEntity` object to its list of childs and assign the `ReadEntity` a 4x4 unit matrix.

The Interface

The interface is handled by the `ExportDialog` class.

`ExportDialog` inherits from `DialogGrid`. `DialogGrid` is just meant to provide us with a few methods for easing the creation and modification of the interface. Namely, it is used for reducing the granularity of the coordinate system so we can work in columns and lines, and turning the y axis upside down, so (0, 0) is in the upper left corner.

`ExportDialog` contains three methods which are given as arguments to `Blender.Draw.Register`:

- `self.DrawDialog`: Called every time the interface is redrawn.
- `self.KeyEvent`: Called every time the users presses a key on the keyboard.
- `self.ButtonEvent`: Called every time the user uses something in the interface.

In addition, `ExportDialog` contains `self.DataGroup`, an inner class created to avoid duplication of a relatively large amount of code. This is used once for each .soy object type. It is used for drawing the boxes determining the level of filesplitting for that object type.

For actual export, the interface calls one of two methods, either `self.ModelExporter` or `self.LevelExporter`. `ModelExporter` Exports all groups as singular mesh objects, and uses `self.ExportGroup` to export each group. `self.LevelExporter` exports either one or all scenes in blender as .soy levels. That is, a top Node in its own file, describing the level, together with all the associated data, either in that same file, or organized in a directory structure. It uses `self.ExportScene` to export each scene.

`treatObject` and `bindNodes` are two special case functions. These are called from the `ExportDialog`'s `self.ExportScene` method. `treatObject` is called for every object in a scene. It's role is to find if an Object is merely an instantiated Group, and if not, to determine if a `MeshReader` was already created for that particular mesh. It also handles the first step of the parenting Node/child process by creating a dictionary of objects that are discovered as being parents.

The second and third steps of parenting is performed in `bindNodes`, which first links all the parents to their childs in the script's native data structure, and finally creates all the Nodes.