STL FILE PARSER

This software aims to read, parse and count the number of facets and the total surface area of a model in a STL File (stereolithography file). It's implemented in Golang, a fast, ultra low latency, strong typed and compiled language that is growing in a stead pace.

1) Project directory structure

Despite the project simplicity, it is organized to separate the main structure parts in a easy and understandable way to anyone.

```
|-- stl-parser              //project root
    |-- bin
        |-- stlparser       //linux executable
        |-- stlparser-mac   //mac exectable
        |-- strparser.exe   //windows executable
    |-- model
        |-- ascii.go        //Concrete type for ascii STL files
        |-- binary.go       //Concrete type for binary STL files
        |-- facet.go        //Struct type for facet definition
        |-- model.go        //Interface type for model implementation
        |-- utils.go        //Shared utility functions
    |-- shape
        |-- ascii           //Ascii STL files samples
            |-- head.stl
            |-- humanoid.stl
            |-- ...
        |-- binary          //Binary STL files samples
            |-- calypso.stl
            |-- dragon.stl
            |-- ...
    |-- go.mod              //Manage module dependencies
    |-- main.go             //Entrypoint file
    |-- readme.md           //Readme file for the project
```

This project directory structure helps to keep source code well organized and ready to expand. For example, if a new format of STL file is needed to be parsed, it is a matter of creating a new Concrete Type for the file format, by extending the Model Interface Type, since every Concrete Type for parsing may need to implement the methods defined on the Interface.

2) Running the software

Start using `stlparser` is as easy as running the following command in a Linux terminal:

```
./stlparser ../shape/ascii/turbine.stl
```

You should see the following output:

```
Number of facets: 8404
Surface area: 395.2298239091326
```

Executable binaries are within `bin` directory. Under `bin` you will find executables for Linux, Windows and Mac. Assuming you are using a 64 bit OS, but if not, as a last resource you can run the `main.go` file located in the base project directory. Example:

```
go run . shape/ascii/moon.stl
```

You should see the same output. However, realize that running the source code is not the same as running the compiled file (binary). As a result, the performance will be degraded in relation to binary execution.

The executable binary expect a parameter, which is the path to the STL file. If no parameter is passed, the software will display a warning. You can drop the STL file under `shape` directory in the proper subdirectory, `ascii` or `binary`, yet you can pass the full path to the STL file. Pay attention to the file permissions, which must be readable by the user running the software. Example:

```
# Pass a relative path to the STL file within the project directory
./stlparser ../shape/ascii/turbine.stl
```

```
# Pass a absolute path to the STL file to any user readable directory
./stlparser /home/username/3DModels/shape/ascii/turbine.stl
```

3) Design decisions

A couple of design decisions were taken in order to make the source code extensible, easy to read, organized, with small binaries, error free, with usage of design patterns and software architecture and also with no external modules.

3.1) Extensible source code

The heart of the project lies inside `model` directory, which holds the `Model Module`, the unique Go module present in the project. By using the Go idiom for Interfaces and Concrete Types, the source code can be easily extendable by a matter of creating another Concrete Type definition and extend the Model Interface definition. If a new task is needed, it is a matter of creating a new method in the Model Interface definition. Even a new Interface can be created to accomodate new tasks not related to the existing Model Interface definiiton. If a new Struct Type is needed, it is a matter of defining it in a new file under `model` directory or creating a completly new module outside the `Model Module` . If a new shared utility method is needed, it is a matter of defining the new method within `utils.go` package.

By using intensively the Golang "Module Design Pattern", a well isolated software can be created. Keeping code well organized in modules and packages. By using the Golang idiom for public and private variables, methods, structs, interfaces, etc... a mode robust and memory error free environment can be achiaved.

3.2) Easy to read source code

Source code should be easly readable by humans. That's why clean and simple code was a target on this project. You can realize that by looking the `main.go` file. It defines only the minimal code to sanitize the input, call the `model` module and display the results. The `model` module is also simple and clean. It defines only the minimal code to parse the STL file, count the number of facets and calculate the total surface area. Golang idiom was used to keep the code clean and simple.

3.3) Organized source code

The project directory structure was designed to keep the source code organized. The `model` module is well isolated from the rest of the project. The `shape` directory holds the STL files samples. The `bin` directory holds the executable binaries. The `main.go` file is the entrypoint for the software. The `readme.md` file is the readme file for the project. The `go.mod` file is the module definition file for the project. All the effort was made to keep the source code organized, yet it can be improved by adding more directories and files, if needed.

3.4) Small binaries

The executable binaries are small in size mainly because the usage of external modules was avoided. External modules may use another external modules which may grow the size of source code and the resulting binaries. External modules may also overload the source code tree with useless code, introduce security issues, third party bugs, etc... By using only the standard Golang library, the source code and the resulting binaries are kept small in size. However, sometimes, external modules are needed to accomplish a task that is not ready available in the standard Golang library. In this case, the external module should be used, but with attention. For this project, the tasks were simple and the standard Golang library was enough to accomplish them.

3.5) Error free

Golang itself is a great language to keep running code errors under control. As a strong typed language with a compiler that checks for errors at development time, Golang binaries run with rock solid stability. However, where needed, check points for user input and system calls were added to keep the software running without errors. Message errors were also added to help the user to understand what is going on.

3.6) Design patterns and software architecture

Despite the project's small size, attention was targeted to keep the code made with design patterns and software architecture in mind. Golang itself uses under the hood a lot of desing patterns, like Modules, Packages, Interfaces, Concrete Types, SOLID and KISS concepts, etc. All the best practices were used to keep the code clean, simple, easy to read and organized following the Golang idiom.

3.7) No external modules

No need for External modules was verified for the project. All tasks could be accomplished with Golang Standard Librar. External modules may use another external modules which may grow the size of source code and the resulting binaries. External modules may also overload the source code tree with useless code, introduce security issues, third party bugs, etc... By using only the Golang Standard Library, the source code and the resulting binaries are kept small in size. However, sometimes, external modules are needed to accomplish a task that is not ready available in the Golang Standard Library. On this case, the external module should be used, but with attention. For this project, the tasks were simple and the Golang Standard Library was enough to accomplish them.

4) Implementation details

The software was designed to be fast and low latency. However, some key points about the software implementation are listed bellow:

4.1) Openning and reading the STL file

STL files can be huge in size. Some files can have Gigabytes of size. The software was designed with huge files in mind. So, proper standard library functions were used to read the STL file in chunks of 256 kilobytes. This way, the software can read the STL file in a steady pace, without overloading the memory. The software can read the STL file in a fast way, without overloading the CPU. The software can read the STL file in a low latency way, without overloading the network (for now the file is read from filesystem). For that task, `bufio` and `enconding\binary` packages were used. At first, `os` package was used just for openinig a valid file pointer to the file, yet it does not open the target file, `os.Open()` just returns a valid file pointer.

If all check points are passed, the file pointer will be passed to `bufio.NewScanner()` or `binary.Read()`, depending if STL file encoded in ASCII or BINARY. No matter which one, the file will be read in chunks of 256 kilobytes, and these chunks will be passed to the next step, which is the parsing of the STL file for counting the number of facets and calculating the total surface area.

4.2) Parsing the STL file

Parsing the STL file changes in implementation and resources used depending of the format of the STL file. There are two formats of STL files, ASCII and BINARY. The software was designed to parse both formats. Let's see how the software parses each format.

4.2.1) ASCII STL files

ASCII STL files has its contents human readable which facilitate the direct inspection and analisys of the model data. Usually, ASCII files are used for small models, since for the huge ones, usually BINARY format is prefered for performance reasons.

For counting the number fo facets in the model, it was a matter of counting how many times the sentence `facet normal` appears in the file. This was implemented in method `FacetCounter` of package `ascii.go`. Take this sample:

```
solid Moon
  facet normal -0.785875 0 -0.618385
   outer loop
    vertex 0.360463 0 2.525
    vertex 0 0 2.98309
    vertex 0.360463 0.2 2.525
   endloop
  endfacet
  facet normal -0.785875 0 -0.618385
   outer loop
    vertex 0 0 2.98309
    vertex 0 0.2 2.98309
    vertex 0.360463 0.2 2.525
   endloop
  endfacet
  facet normal -0.130526 0 0.991445
   outer loop
    vertex 0 0 2.98309
    vertex 0.128412 0 3
    vertex 0 0.2 2.98309
   endloop
  endfacet
endsolid Moon
```

By reading each line and checking for `facet normal` sentence, and summing up them, the final count of facets were achieved. This method is fast, low latency and memory efficient.

For calculating the total surface area of the model, first it was need to find, store and return the facets found in the model. For simplicity sake, this was implemented in `FacetCounter` too. The facets were stored in a slice of `Facet` struct, which holds the three vertices of the facet. The `Facet` struct is defined in `facet.go` file. The `FacetCounter` method returns the slice of `Facet` structs. This approuch is fast, low latency and memory efficient.

Model surface area is implemented in method `SurfaceArea` of package `ascii.go`. This method receives the slice of `Facet` structs and calculates the surface area of each facet, summing up them and returning the total surface area of the model. However, a tricky issue was forseen in this method. The surface area of each facet is calculated by using the `Cross Product` of two vectors. The `Cross Product` is a mathematical operation that returns a vector that is perpendicular to the two vectors being multiplied. These calculations could stuck and slow down the software execution. To avoid this, Golang `routines` was used. A `routine` is a lightweight thread managed by the Go runtime. The `SurfaceArea` method creates a `routine` for each chunk of facets, and each `routine` calculates the surface area of the chunk, and returns to the variable `areaCh` which is a `channel` to the `routine`. This approuch is fast, low latency and memory efficient.

To better improve the Golang `routines`, they are spread across the available CPU cores. This way, the software can take advantage of the CPU cores, which will improve the performance of the software. This approuch is fast, low latency and memory efficient.

4.2.2) BINARY STL files

BINARY STL files has its contents encoded in binary which makes it hard to inspect and analize the model data. Usually, BINARY files are used for huge models, with gigabyes in size. Parsing BINARY files is a little bit more complex than ASCII files, since the data is encoded in binary. Yet, the same methods used for ASCII files were used for BINARY files, since both packages `ascii.go` and `binary.go` implements the same methods defined in `model.go` interface. However, the implementation of the methods are different, since the data is encoded in different formats. Take this example:

```
00000000  53 54 4c 42 20 41 54 46  20 31 32 2e 39 2e 30 2e  |STLB ATF 12.9.0.|
00000010  39 39 20 43 4f 4c 4f 52  3d 19 19 19 ff 20 20 20  |99 COLOR=....   |
00000020  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |                |
*
00000050  2c 00 05 00 08 be 7d 3f  ae 95 a2 3d 29 4d d9 3d  |,.....}?...=)M.=|
00000060  ec a8 d9 44 b7 52 ee 46  a3 f8 50 44 8f c8 d7 44  |...D.R.F..PD...D|
00000070  bf be ed 46 03 ef 80 44  d6 24 db 44 40 2a ed 46  |...F...D.$.D@*.F|
00000080  a3 f8 50 44 63 0c f5 ca  7d 3f 19 48 97 3d d3 a5  |..PDc...}?.H.=..|
00000090  dd 3d d6 24 db 44 40 2a  ed 46 a3 f8 50 44 8f c8  |.=.$.D@*.F..PD..|
000000a0  d7 44 bf be ed 46 03 ef  80 44 7a 6c d8 44 8a 42  |.D...F...Dzl.D.B|
000000b0  ed 46 e3 5d 80 44 63 0c  53 ef 7c 3f 3c c9 e6 3d  |.F.].Dc.S.|?<..=|
000000c0  06 d0 d7 3d d6 24 db 44  40 2a ed 46 a3 f8 50 44  |...=.$.D@*.F..PD|
000000d0  7a 6c d8 44 8a 42 ed 46  e3 5d 80 44 ad 30 db 44  |zl.D.B.F.].D.0.D|
000000e0  40 c6 ec 46 a3 78 5d 44  63 0c 66 64 7d 3f 5e 1f  |@..F.x]Dc.fd}?^.|
...
```

It is much more difficult for humans to read BINARY files directly, indeed. However, they are much more efficient in terms of size and performance. BINARY files are well structured, and data can be easely addressed. For example, the first 80 bytes of the file are the header, which is not used by the software. The next 4 bytes are the number of facets in the model, which is used by the software. The next 50 bytes are the first facet, which is used by the software. The next 50 bytes are the second facet, which is used by the software. And so on. The software was designed to read the file in chunks of 256 kilobytes, and each chunk is passed to `binary.Read()` function, which reads the chunk and returns the number of bytes read. This approuch is fast, low latency and memory efficient.

Supringly, surface area calculation for BINARY files has the same implementation of ASCII files. This happens because the `Facet` struct is the same for both ASCII and BINARY files. The `Facet` struct holds the three vertices of the facet. The `Facet` struct is defined in `facet.go` file. The `FacetCounter` method returns the slice of `Facet` structs. Only the `FacetCounter` method implementation is different for ASCII and BINARY files. `SufaceArea` method could be shared by both packages, yet for simplicity sake, it was implemented in both packages, since in the future the implementation may change. This approuch is fast, low latency and memory efficient.

5) QA - Quality Assurance improvements

Any software can be improved in terms of QA. For this project, some improvements can be made. Let's see some of them.

5.1) Unit tests

Unit tests are important to certify that the code performs as expected during runtime. Golang compiler do a great job at keeping the source code error free, yet this does not mean that the code will run as expected. For this project, unit tests were not implemented, yet they can be implemented in the future.

5.2) Logging, tracing and monitoring

For this project, logging, tracing and monitoring were not implemented, yet they can be implemented in the future. Logging is important to keep track of the software execution. Tracing is important to keep track of the software execution flow. Monitoring is important to keep track of the software execution performance. All these three tasks can be implemented in the future.

5.3) Error handling

For this project, a very basic Error Handling was implemented. The software checks for user input and system calls errors. However, a more robust Error Handling can be implemented in the future, for instance, by saving the errors in a remote database for further analisys. All these tasks can be implemented in the future.

5.4) Code refactoring and optimization

Software code can be refactored and optimized in the future. For instance, the `SurfaceArea` method can be refactored to use less memory and CPU. The `FacetCounter` method can be refactored to use less memory and CPU. The `main.go` file can be refactored to use less memory and CPU. All these tasks can be implemented in the future.

6) Conclusion

This project is a demonstration of how Golang can be used to read, parse and count the number of facets and the total surface area of a model in a STL File (stereolithography file). Golang is a fast, ultra low latency, strong typed and compiled language that is growing in a stead pace. The project directory structure helps to keep source code well organized and ready to expand. The software was designed to be fast and low latency. However, some key points about the software implementation were listed. Any software can be improved in terms of QA. This software can be used as a base for other projects.