Brown University ENGN2912B Fall 2017

Scientific Computing in C++

Lecture 14 | The IfsViewer Application

In this lecture we will start to build an interactive application of moderate complexity application for reading, visualizing, operating on, and saving polygon meshes and point clouds. This is a much more complex project than the ones we have worked on so far, which requires careful class design, partitioning the project files into libraries, and also linking your program using external libraries. These libraries are part of even more complex packages, which you will have to install in your machine, and configure for use in conjunction with Qt.

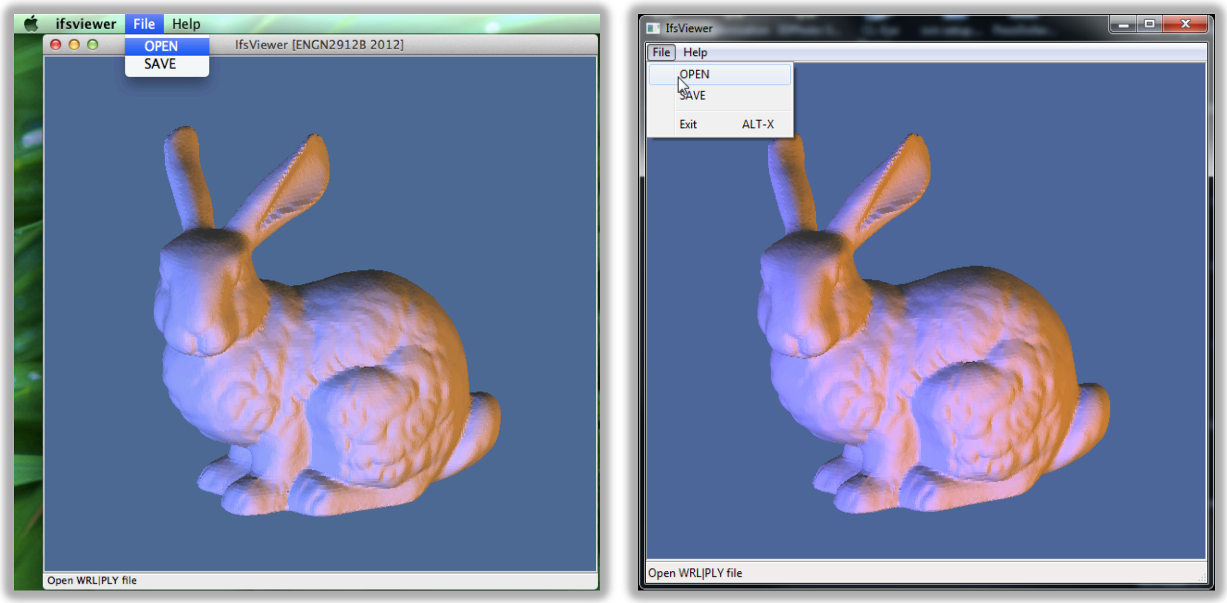

Figure 1 The IfsViewer application shown running in OSX and Windows.

This application, as shown above running in OSX and Windows, will be extended in subsequent lectures and homework assignments. What you will implement in this first assignment provides basic functionality to load, save, and operate on polygon meshes and point clouds. It will be able to load a polygon mesh or a point cloud from a file, visualize it, and save the polygon mesh or point cloud to a file. In this first homework assignment you will implement the main data structures for representing a polygon mesh or a point cloud in memory, a method to load a polygon mesh or point cloud from a file, and a method to save the polygon mesh or point cloud to a file. You will implement the loading and saving using the Factory framework, which we covered in lecture 13. In subsequent homework assignments you will implement additional loaders and savers to support other popular file formats used to store polygon meshes and point clouds. For this assignment, all the user interface and OpenGL graphics programming is provided. In subsequent assignments you will extend the application by writing, in addition to the loaders and savers mentioned above, a number of geometry processing operations. Since most of these algorithms do require additions to the user interface to set various parameters, we will design a unified framework to add these user interface components, which you will also implement in an incremental fashion.

Qt

The graphical user interface for IfsViewer is built using **Qt**. Your part of the code can be implemented in its vast majority using the standard C++ classes, but you can also use Qt classes, if you prefer to do so. We have discussed the installation process for Qt within the context of the QtCalc assignment. For this assignment you should download the archive file `Assignment9-IfsViewer.zip` from the course web site, unzip it, and load the project file `IfsViewer.pro` file located in the `src` directory.

The goal is to complete the IfsViewer application implementation

I have decided to partition the project into three subprojects. The files in the directory `IfsViewer/src/viewer` implement the user interface. In this assignment you don't need to make any changes to those files. The files in the directory `IfsViewer/src/util` implement some utility functions which in this assignment you don't have to change either. So far there is only one class in this directory. The `BBox` class provides functionality to represent a bounding box containing a set of points in d-dimensional space. In this application we are only using it for `d=3`. One of the constructors builds a bounding box from a set of points, stored as an instance of the `vector<float>` class which stores all the coordinates of the points as a linear array (x0,y0,z0),(x1,y1,z1),…,(xN,yN,ZN).

The VRML (Virtual Reality Modeling Language, pronounced vermal or by its initials) is an international standard file format for representing 3-dimensional (3D) interactive vector graphics, designed particularly with the World Wide Web in mind. It has been superseded by X3D, but still widely used.

https://en.wikipedia.org/wiki/VRML

As a reference, this is where you can read the VRML specification
http://www.web3d.org/documents/specifications/14772/V2.0/
You can download the **VRML 2.0 - Cheat Sheat**, by Jan Hardenbergh from the course web site. This is a summary of all the VRML nodes.
The "Annotated VRML 97 Reference Manual" by Rikk Carey and Gavin Bell, provides additional information about the VRML standard
http://accad.osu.edu/~pgerstma/class/vnv/resources/info/AnnotatedVrmlRef/about.htm

In Assignment 9 you will have to complete the implementation of several files in the directories `IfsViewer/src/ifs` and `IfsViewer/src/test`. First of all you will have to complete the implementation of the `Ifs` class. This class is intended to be a container for the information which can be represented in a VRML `IndexedFaceSet` node. This is how the `IndexedFaceSet` node is defined in the VRML standard

```
IndexedFaceSet {
  eventIn       MFInt32 set_colorIndex
  eventIn       MFInt32 set_coordIndex
  eventIn       MFInt32 set_normalIndex
  eventIn       MFInt32 set_texCoordIndex
  exposedField  SFNode  color             NULL
  exposedField  SFNode  coord             NULL
  exposedField  SFNode  normal            NULL
  exposedField  SFNode  texCoord          NULL
  field         SFBool  ccw               TRUE
  field         MFInt32 colorIndex        []        # [-1,)
  field         SFBool  colorPerVertex    TRUE
  field         SFBool  convex            TRUE
  field         MFInt32 coordIndex        []        # [-1,)
  field         SFFloat creaseAngle       0         # [ 0,)
  field         MFInt32 normalIndex       []        # [-1,)
  field         SFBool  normalPerVertex   TRUE
  field         SFBool  solid             TRUE
  field         MFInt32 texCoordIndex     []        # [-1,)
}
```

We are going to ignore the `eventIn` fields, as well as whether fields are "exposed" or not, and to keep things simpler we are not going to implement the `color`, `coord`, `normal`, and `texCoord` fields as separate classes. Instead, the fields will be represented as follows in the `Ifs` class. Various methods to access these private variables are defined in the `Ifs.hpp` header file. You have to implement them all. Pay particular attention to understanding the convention adopted in the standard to represent property bindings, and in particular you need to understand what is the meaning of properties bound PER_VERTEX, PER_FACE, PER_FACE_INDEXED, and PER_CORNER, as well as what is the role of the four index fields in all these cases.

```
class Ifs {
private:
  bool          _ccw;
  bool          _convex;
  float         _creaseAngle;
  bool          _solid;
  bool          _normalPerVertex;
  bool          _colorPerVertex;
  vector<float> _coord;
  vector<int>   _coordIndex;
  vector<float> _normal;
  vector<int>   _normalIndex;
  vector<float> _color;
  vector<int>   _colorIndex;
  vector<float> _texCoord;
  vector<int>   _texCoordIndex;

  // …

}
```

The remaining files in this directory implement Factory frameworks to load instances of the Ifs class from files, and to save instances of the Ifs class to files

```
IfsLoader.cpp
IfsLoader.h
IfsSaver.cpp
IfsSaver.h
IfsWrlLoader.cpp
IfsWrlLoader.h
IfsWrlSaver.cpp
IfsWrlSaver.h
Loader.h
Saver.h
```

You have to complete the implementation of these files as well. Most of the work to be done is concentrated in the classes `IfsWrlLoader` and `IfsWrlSaver` which are design to support the file extension `wrl`. VRML files may contain complex scene graphs with multiple `IndexedFaceSet` nodes, as well as many other different nodes. After opening the file for reading, the `IfsWrlLoader.load()` method will read the first line of the file, verify that the line just read contains a valid VRML header line, and then it will search for the first instance of an `IndexedFaceSet` node. If one is found, it will parse the whole node (all the fields) until the closing bracket, will skip the rest of the file, close the file and return the Boolean value `true` to report success. If no `IndexedFaceSet` node is found, it will empty the Ifs node and return `false`. You should pay particular attention to how you split the input stream into tokens. In particular, the ',' symbol should be treated as white space, and other delimiting symbols such as '{','[',']','{' should be regarded as separate tokens, whether they are surrounded by white space or not.

For `IndexedFaceSet` nodes with texture coordinates, you have to parse both the texture coordinates and the texture coordinate indices (if present), although the IfsView application is not currently supporting textured models. But you don't have to identify the location of the texture image file described in the VRML file to texture such model, since it is not represented within the IndexedFaceSet node. We will look at this problem in a future assignment.

Perhaps you should implement the `IfsWrlSaver` class first, since the `IfsWrlSaver.save()` method only requires writing to a file, and you have already done some of this before. The output file must be a valid VRML file, comprising a single `Shape` node, with `IndexedFaceSet` node as its geometry field. You can also write a Material node as the appearance node of the `IndexedFaceSet` node to be able to specify material properties such as `diffuseColor`. If you do so it becomes a lot easier to edit the output file to change material properties after it is created by your application. The `IfsWrlSaver.save()` method has to be able to save all the fields of the `IndexedFaceSet` node, as represented in the Ifs class. You should implement and debug incrementally. For example, first make sure that the `coord` and `coordIndex` arrays are properly saved, ignoring the `color`, normal, and `texCoord` properties.

You can use MeshLab as a tool to debug you implementation. If you load a VRML file and then you save it under a different name, you can visualize both files and compare them. Of course they should be identical. It is usually difficult to debug interactive applications. In our case you will also implement a simple command line application to debug the Ifs, IfsLoader, and IfsSaver classes. The names of the input and output VRML files should be passed to the command-line application as command line parameters. Additional command line flags should be used for example to perform some simple operations on the Ifs class after it is loaded from the input file, but before it is saved to the output file. Simple operations can be: inverting the direction of the normal vector if normal vectors are present, and removing normal, colors, and/or texture coordinates. You have to make sure than when you apply these operations, the Ifs class remains valid in terms of the dimensions of the property arrays and variables, so that the output file represents valid VRML syntax. To do this you will have to modify the provided project files.

Parsing Command Line Parameters and ASCII Files

To complete the implementation of the IfsViewer application introduced in Lecture 13 you need to develop a simplified parser for VRML files, where you have to search for the first instance of an `IndexedFaceSet` node, parse it, and then quit. In this lecture we will discuss one well established technique to do so based on partitioning the input stream into “tokens”. In subsequent assignments we will extend the VRML parser, and will implement other parsers using the same methodology.

But we will first look at how to parse command line parameters, both for command line applications such as those we have been writing since the beginning of the course, as well as for wxWidgets applications, where we do not have access to the `main()` function. One important use for command line parameters is to be able to invoke your programs to run without user interaction. Command line parameters can be used to specify the names of input and output files, to specify values for Boolean variables, also called “switches”, and to specify values for numerical variables.

For example, to debug the code that you need to implement for the IfsViewer application, you will have to implement a command line application `ifstest` which will take as arguments the name of an input file, and the name of an output file. In addition, you will have to add a switch to turn on and off the printing of console messages used to monitor the progress of the program. The application will be run from a console by typing the following command

```
> ifstest –debug inputFile.wrl outputFile.wrl
```

In this case the command line parameters is an array of four C-style (`char*`) strings composed of: “`ifstest`”, “`-debug`”, “`inputFile.wrl`”, and “`outputFile.wrl`”. Note that the first parameter is the name of the application itself.

Parsing Command Line Parameters in Command Line Applications

A command line application executes the `main()` function. To get access to the command line parameters you must declare `main()` as follows

```
int main(int argc char* argv) {
  // process ...
}
```

The actual names of the arguments is not important, but this is the established convention: `argc` stands for **arg**ument **c**ount, and `argv` for **arg**ument **v**ector. Since `argv[0]` is always equal to the name of the application, it is guaranteed that `argc>=1.`

For the example given above, we could try to process the command line parameters as follows

```
#include<string>
int main(int argc, char** argv) {
  bool   debug   = false;
  string inFile  = “”;
  string outFile = “”;
  if(string(argv[1])==”-debug”)
    debug = true;
  inFile = string(argv[2]);
  outFile = string(argv[3]);
  // load input file
  // process ...
  // save output file
  return 0;
}
```

Note that since the command line parameters are passed as C-style strings, and we have decided to implement this program using C++ strings, we convert the parameters from `char*` to `string` using the `string class` constructor. We first initialize the variables debug, inFile, and outFile to default values, and then we set them to the values specified by the command line parameters. If `argv[1]` is equal to the string "`-debug`" then we set the `debug` variable to true. Then we set the string `inFile` to the value specified by the command line parameter `argv[2]`, and `outFile` to the value specified by the command line parameter `argv[3]`.

This program will work as long as it run as described above, with the three arguments. But it will fail for example it is run without the "`-debug`" switch

```
> ifstest inputFile.wrl outputFile.wrl
```

In fact, this program most likely will crash while trying to set the value of the `outFile` variable, since the fourth argument `argv[3]` would not be defined. To prevent crashes we need to implement the parsing of command line parameters in a different way, where we analyze each command line parameter and set variables depending on its values. At the same time, we need to detect errors in the command line syntax, and exit the program reporting an error code in such cases. For example, consider the following code fragment

```
#include<string>
int main(int argc, char** argv) {
  bool   debug   = false;
  string inFile  = "";
  string outFile = "";
  for(int i=1;i<argc;i++) {
    if(string(argv[i])=="-debug")
      debug = true;
    else if(argv[i][0]=='-')
      return -1;
    else if(inFile=="")
      inFile = string(argv[i]);
    else if(outFile=="")
      outFile = string(argv[i]);
  }
  if(inFile=="") {
    if(debug) { cerr << "ERROR | no input file name" << endl; }
    return -2;
  }
  if(outFile=="") {
    if(debug) { cerr << "ERROR | no output file name" << endl; }
    return -3;
  }
  // load input file
  // process ...
  // save output file
  return 0;
}
```

In the loop `for(int i=1;i<argc;i++)` each command line parameters `argv[i]` is inspected and processed. In this implementation we are assuming that zero or more switches, an input file name, and

an output file name follow the application name in the command line. The application will neither crash if the “-debug” switch is not specified, nor if the input or output file names are missing. However, the application will subsequently exit with an error message if either one of the two file names are not specified. Also note that the statement `else if(argv[i][0]==’-‘) exit -1;` will make the application quit if any other switch is specified. By convention switches are specified with strings starting with a `‘-‘` character. Alternatively, we could replace this statement with `else if(argv[i][0]==’-‘) continue;` to skip unrecognized switches.

It is good practice to make the application print a message explaining what is the acceptable command line syntax. For example, in this code segment the application will print the usage massage and quit if it is run without parameters, if the “`-u`” or “`-usage`” flags are specified, and if any other unrecognized flag is specified. Square brackets “[]” describe optional parameters, and the symbol “|” specifies alternative syntax.

```
#include<string>
void usage() {
  cerr
  << “USAGE | ifstest [-d|-debug][-u|-usage] inputFile.wrl outputFile.wrl”
  << endl;
}
int main(int argc char* argv) {
  bool   debug   = false;
  string inFile  = “”;
  string outFile = “”;
  if(argc==1) {
    usage();
    return 0;
  }
  for(int i=1;i<argc;i++) {
    if(string(argv[i])==”-d” || string(argv[i])==”-debug”) {
      debug = true;
    } if(string(argv[i])==”-u” || string(argv[i])==”-usage”) {
      usage();
      return 0;
    } else if(argv[i][0]==’-‘) {
      usage();
      return -1;
    } else if(inFile==””) {
      inFile = string(argv[i]);
    } else if(outFile==””) {
      outFile = string(argv[i]);
    }
  }
  if(inFile==””) {
    if(debug) { cerr << “ERROR | no input file name” << endl; }
    return -2;
  }
  if(outFile==””) {
    if(debug) { cerr << “ERROR | no output file name” << endl; }
    return -3;
  }
  // load input file
  // process ...
  // save output file
  return 0;
}
```

As you can see in this example, adding more switches is straightforward. The convention in this implementation is that all the switches must precede the input and output file names in the command line. The switches can be specified in arbitrary order, but they have to be followed by the input and output files in that order. If we want to also be able to specify the input and/or output file names at arbitrary positions in the command line, we need to support the processing of command line parameters with additional values. For example, consider the following revised `usage()` function

```
void usage() {
  cerr << “USAGE | ifstest” <<endl;
  cerr << “         [-d|-debug        ]” << endl;
  cerr << “         [-u|-usage        ]” << endl;
  cerr << “         [-i inputFile.wrl ]” << endl;
  cerr << “         [-o outputFile.wrl]” << endl;
}
```

In this command line syntax the input file name is specified as the command line parameter following a command line parameter matching the string “`-i`”, and the output file name is specified as the command line parameter following a command line parameter matching the string “`-o`”. The command line processing loop can be modified as follows

```
  for(int i=1;i<argc;i++) {
    if(string(argv[i])==”-d” || string(argv[i])==”-debug”) {
      debug = true;
    } if(string(argv[i])==”-u” || string(argv[i])==”-usage”) {
      usage();
      return 0;
    } else if(string(argv[i])==”-i”) {
      inFile = string(argv[++i]);
    } else if(string(argv[i])==”-o”) {
      outFile = string(argv[++i]);
    } else if(argv[i][0]==’-‘) {
      usage();
      return -1;
    }
  }
```

Note that the index is incremented before assigning the command line parameter of the variables `inFile` and `outFile`, and the application will most likely crash for example if the command line parameter “`-i`” is the last one in the command line and it is not followed by the input file name. One possibility is to add additional tests to catch this command line syntax error, such as

```
    } else if(string(argv[i])==”-i”) {
      if(++i >= argc) {
        if(debug) { cerr << “ERROR | no input file name” << endl; }
        return -4;
      }
      inFile = string(argv[i]);
    } ...
```

Another possibility is to skip the parameter, since the missing file name error will be cached later

```
    } else if(string(argv[i])==”-i”) {
      if(++i >= argc) continue;
      if(argv[i][0]==’-‘) {
        if(debug) { cerr << “ERROR | missing input file name after –i” << endl; }
        return -3;
      }
      inFile = string(argv[i]);
    } ...
```

Note that we added another test to catch a missing argument error occurring in the middle of the command line.

Parsing Numerical Parameters

In some cases it is necessary to specify numerical parameters in the command line. In those cases we need to convert the string representation of the numerical values to actual numerical values. As an example, suppose that we want to specify the center and radius of a sphere so that all the vertices of the input polygon mesh not contained within the sphere are deleted, as well as all the polygons containing those vertices. The radius will be specified as an integer, and the center as three floating point numbers. The following `usage()` function specifies the command line syntax

```
void usage() {
  cerr << “USAGE | ifstest” <<endl;
  cerr << “          [-d|-debug          ]” << endl;
  cerr << “          [-u|-usage          ]” << endl;
  cerr << “          [-i inputFile.wrl   ]” << endl;
  cerr << “          [-o outputFile.wrl  ]” << endl;
  cerr << “          [[-r|-radius] radius]” << endl;
  cerr << “          [[-c|-center] x y z ]” << endl;
}
```

Additional variables need to be added to the `main()` function

```
int main(int argc char* argv) {
  bool   debug   = false;
  string inFile  = “”;
  string outFile = “”;
  int    r       = 0;
  float  x       = 0.0f;
  float  y       = 0.0f;
  float  z       = 0.0f;
```

and statements need to be added to the command line processing loop

```
#include<cstdlib>
// ...

    } else if(string(argv[i])==”-r” || string(argv[i])==”-radius”) {
      radius = atoi(argv[++i]);
    } else if(string(argv[i])==”-c” || string(argv[i])==”-center”) {
      x = atof(argv[++i]);
      y = atof(argv[++i]);
      z = atof(argv[++i]);
    } ...
```

The functions `atoi()` and `atof()` are defined in the system header file `cstdlib`, which has to be included as well for the program to compile.

`int atoi (const char* str);`

Parses the C string str interpreting its content as an integral number, which is returned as an int value. The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many base-10 digits as possible, and interprets them as a numerical value. The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function. If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed and zero is returned.

`double atof (const char* str);`

Parses the C string str interpreting its content as a floating point number and returns its value as a double. The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes as many characters as possible that are valid following a syntax resembling that of floating point literals (see below), and interprets them as a numerical value. The rest of the string after the last valid character is ignored and has no effect on the behavior of this function. A valid floating point number for atof using the "C" locale is formed by an optional sign character (+ or -), followed by one of: A sequence of digits, optionally containing a decimal-point character (.), optionally followed by an exponent part (an e or E character followed by an optional sign and a sequence of digits). A 0x or 0X prefix, then a sequence of hexadecimal digits optionally containing a decimal-point character (.), optionally followed by an hexadecimal exponent part (a p or P character followed by an optional sign and a sequence of hexadecimal digits). INF or INFINITY (ignoring case). NAN or NANsequence (ignoring case), where sequence is a sequence of characters, where each character is either an alphanumeric character or the underscore character (_). If the first sequence of non-whitespace characters in str does not form a valid floating-point number as just defined, or if no such sequence exists because either str is empty or contains only whitespace characters, no conversion is performed and the function returns 0.0.

Note that both `atoi()` nor `atof()` catch syntax errors in the input string, but rather than reporting the error, they return default values (0 and 0.0 respectively). Alternatively, you could implement your own numeric conversion functions based on the C-style `sscanf()`, which is defined in the header file `stdlib.h`

`int sscanf (const char* str, const char* format, ...);`
`where str` is the C string that the function is trying to parse; `format` is a C string that contains a format string that follows the same specifications as format in `scanf` (look for the `scanf` documentation on line for details), and ... are additional arguments. Depending on the format string, the function may expect a sequence of additional arguments, each containing a pointer to allocated storage where the interpretation of the extracted characters is stored with the appropriate type. There should be at least as many of these arguments as the number of values stored by the format specifiers. Additional arguments are ignored by the function. On success, the function returns the number of items in the argument list successfully filled. This count can match the expected number of items or be less (even

zero) in the case of a matching failure. In the case of an input failure before any data could be successfully interpreted, EOF is returned. For example, the following code fragments illustrate how to use sscanf() to parse an `int` and a `float`

```
const char* str = “-123”;
int radius = 0;
if(sscanf(str,”%d”,&radius)<1) {
  // ERROR
}

const char* str = “0.876”;
float x = 0.0f;
if(sscanf(str,”%f”,&x)<1) {
  // ERROR
}
```

Note however, that in certain cases which can be considered errors, `sscanf()` does not produce the expected result. For example, for `str=” -23sDS”,` the call `sscanf(str,”%d”,&radius)` will set the variable radius to the value `-23`, and will return the value `1.`

Exception Handling

As we add more command line switches and parameters the number of possible errors increase, and it becomes more and more difficult to handle them in an organized fashion. Exceptions provide a way to react to exceptional circumstances in our program, such as command line processing errors, by transferring control to special functions called *handlers*. To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a `try` block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored. An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword `catch`, which must be placed immediately after the try block. For example, we can modify our `ifstest` program as follows

```
#include<string>
void usage() {
  // ...
}
int main(int argc char* argv) {
  bool   debug   = false;
  string inFile  = “”;
  string outFile = “”;
  try {
    if(argc==1) trow 1;
    for(int i=1;i<argc;i++) {
      if(string(argv[i])==”-d” || string(argv[i])==”-debug”) {
        debug = true;
      } if(string(argv[i])==”-u” || string(argv[i])==”-usage”) {
        throw 2;
      } else if(argv[i][0]==’-‘) {
        trow 3;
      } else if(inFile==””) {
        inFile = string(argv[i]);
      } else if(outFile==””) {
```

```
      outFile = string(argv[i]);
      }
    }
    if(inFile==””) throw 4;
    if(outFile==””) throw 5;
  } catch(int e) {
    // print an appropriate error message and quit
    switch(e) {
      case 1: /* no command line parameters */ break;
      case 2: /* usage */ break;
      case 3: /* unrecognized switch */ break;
      case 4: /* no input file name */ break;
      case 5: /* no output file name */ break;
    }
    usage();
    return -1;
  }
  // load input file
  // process ...
  // save output file
  return 0;
}
```

A throw expression accepts one parameter (in this case an integer), which is passed as an argument to the exception handler. The exception handler is declared with the catch keyword. As you can see, it follows immediately the closing brace of the try block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught. We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

```
try {
  // ...
} catch(int eInt) {
  // ...
} catch(float eFloat) {
  // ...
} catch(...) {
  // ...
}
```

If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last. In this case the last handler would catch any exception thrown with any parameter that is neither an `int` nor a `float`. After an exception has been handled the program execution resumes after the `try-catch` block, not after the `throw` statement. It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

```
try {
  try {
    // ...
  } catch(int eInt) {
    throw; // throws exception to external try-catch block
  }
} catch(float eFloat) {
  // ...
} catch(...) {
  // ...
}
```

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a throw suffix to the function declaration:

```
float myfunction (char param) throw (int);
```

This declares a function called `myfunction` which takes one argument of type `char` and returns an element of type `float`. The only exception that this function might throw is an exception of type int. If it throws an exception with a different type, either directly or indirectly, it cannot be caught by a regular int-type handler. If this throw specifier is left empty with no type, this means the function is not allowed to throw exceptions. Functions with no throw specifier (regular functions) are allowed to throw exceptions with any type:

```
int myfunction (int param) throw(); // no exceptions allowed
int myfunction (int param);         // all exceptions allowed
```

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called exception and is defined in the `<exception>` header file under the namespace std. This class has the usual default and copy constructors, operators and destructors, plus an additional virtual member function called `what()` that returns a null-terminated character sequence (`char *`) and that can be overwritten in derived classes to contain some sort of description of the exception.

```
#include <iostream>
#include <exception>
using namespace std;
class myexception: public exception {
  virtual const char* what() const throw() {
    return "My exception happened";
  }
} myex;
int main () {
  try {
    throw myex;
  } catch (exception& e) {
    cout << e.what() << endl;
  }
  return 0;
}
```

For example, if we use the operator new and the memory cannot be allocated, an exception of type bad_alloc is thrown:

```
try {
  int * myarray= new int[1000];
} catch (bad_alloc& bae) {
  cout << "Error allocating memory." << endl;
}
```

It is recommended to include all dynamic memory allocations within a try block that catches this type of exception to perform a clean action instead of an abnormal program termination, which is what happens when this type of exception is thrown and not caught. If you want to force a `bad_alloc` exception to see it in action, you can try to allocate a huge array. Because `bad_alloc` is derived from the standard base class `exception`, we can handle that same exception by catching references to the exception class:

```
#include <iostream>
#include <exception>
using namespace std;
int main () {
  int* myarray = (int*)0;
  try {
    myarray= new int[1000];
  } catch (exception& e) {
    cout << "Standard exception: " << e.what() << endl;
    return -1;
  }
  // myarray!=(int*)0 here
  return 0;
}
```

We have placed a handler that catches exception objects by reference (notice the ampersand & after the type), therefore this catches also classes derived from exception, like our myex object of class myexception. All exceptions thrown by components of the C++ Standard library throw exceptions derived from this `std::exception` class.

In the previous example the `what()` function of the `class  myexception` returns same string independently of what error you have detected. If you want to add an error message to the exception that you throw you can define your exception class as follows

```
class myexception: public exception {
  private:
    const string& _errorMsg;
  public:
  myexeption(const string& errorMsg):_errorMsg(errorMsg) {
  }
  virtual const char* what() const throw() {
    return _errorMsg.c_str();
  }
};
```

Now in your code should look like this

```
  try {
    // ...
    if(/* error 1 detected */) throw new myexception(“error 1 detected”);
    // ...
    if(/* error 2 detected */) throw new myexception(“error 2 detected”);
    // ...
  } catch (myexception* e) {
    cout << "mexception: " << e->what() << endl;
    delete e;
  } catch (...) {
    // handle other exceptions here
  }
```

Note that the throw statement throws a pointer to an instance of the `myexception class`, and since a new instance is created to be thrown, it has to be deleted after being catched. Also note that the argument to the first catch statement is a pointer to an instance of the `myexception class`, and as a result you get the error message through `e->what()`.

Parsing ASCII Files using a Tokenizer class

We will describe how to parse a VRML file as required to complete the first phase of the IfsViewer. The same techniques can be used to parse other ASCII files. Let us consider the following simple VRML file

```
#VRML V2.0 utf8
Shape {
 geometry IndexedFaceSet {
  coord Coordinate {
   point [
     0.0000    0.0000    1.0000
     0.7236    0.5257    0.4472
    -0.2764    0.8507    0.4472
    -0.8944    0.0000    0.4472
    -0.2764   -0.8507    0.4472
     0.7236   -0.5257    0.4472
     0.8944    0.0000   -0.4472
     0.2764    0.8507   -0.4472
    -0.7236    0.5257   -0.4472
    -0.7236   -0.5257   -0.4472
     0.2764   -0.8507   -0.4472
     0.0000    0.0000   -1.0000
   ]
  }
 coordIndex [
    0  1  5 -1  0  2  1 -1  0  3  2 -1  0  4  3 -1
    0  5  4 -1  1  6  5 -1  1  7  6 -1  1  2  7 -1
    2  8  7 -1  2  3  8 -1  3  9  8 -1  3  4  9 -1
    4 10  9 -1  4  5 10 -1  5  6 10 -1  6  7 11 -1
    6 11 10 -1  7  8 11 -1  8  9 11 -1  9 10 11 -1
  ]
 }
}
```

The first line of the file is required to start with the string “`#VRML V2.0 utf8`”. The remaining of the file can be described as a sequence of “tokens”, which in this case are “`Shape`”, “`{`“, “`geometry`”, “`IndexedFaceSet`”, “`{`“, ... (tokens describing the `IndexedFaceSet` node) ... ,”`}`”, and “`}`”. Your VRML parser should start by opening the file as an instance of the `ifstream` class, using the file name read from the command line or the file selection dialog to invocate the constructor. Then you should read the first line using for example the `ifstream::getline()` function, and verify that it matches the required VRML header line. Then you can use a `Tokenizer` class to split the remaining of the file into tokens. You can use the following header file to implement your `Tokenizer` class.

```
// Tokenizer.h
#ifndef _TOKENIZER_H_
#define _TOKENIZER_H_
#include <ifstream>
#include <string>
using namespace std;
class Tokenizer : public string {
  protected:
    ifstream& _ifstr;
  public:
    Tokenizer(ifstream& ifstr):_ifstr(ifstr) { }
    bool get();
    bool expecting(const string& s);
};
#endif /* _TOKENIZER_H_ */
```

The constructor takes a reference to the open `ifstream` class as a parameter. The function `get()` returns the value `true` when the Tokenizer is able to parse a new token, and `false` otherwise. Once the `get ()` function returns a `false` value the parsing should stop. The actual value of the parsed token is returned as the value of the Tokenizer itself, which is implemented as a subclass of the string class. The function

```
bool Tokenizer:expecting(const string& str) {
  return get() && (str == *this);
}
```

parses a token and compares it with an expected value. We will find this function useful in the implementation of our VRML parser.

Within the `Tokenizer::get()` function, you can use the function

```
int ifstream::get();
```

to get one character at a time from the input stream. Other related functions of the `ifstream` class, which might prove useful are `peek()`, which reads and returns the next character without extracting it, i.e. leaving it as the next character to be extracted from the stream, and `putback(char c)`, which decrements the internal get pointer by one, and c becomes the character to be read at that position by the next input operation.

In the case of VRML files we can define as tokens the following strings: "{", "}", "[", "]", as well as any consecutive sequence of characters not including any one of these four characters or white space. White space includes the space character ' ', the horizontal tab character '\t', the new line character '\n', the vertical tab character '\v', the form feed character '\f', the carriage return character '\r', as well as the comma ',' character which sometimes appears in between numerical values in VRML files. To test for white space characters you can use the `ctype` function

```
int isspace(int c);
```

For a detailed chart on what the different ctype functions return for each character of the standard ANSII character set, see the reference for the <cctype> header.

Parsing Tokenized VRML Files

The following code segment describes a first attempt at implementing a simple VRML parser. An input stream is opened from the input file name. A line is read from the input stream and compared against the expected VRML header. Then a Tokenizer is constructed on top of the input stream, and tokens are read and skipped until an "IndexedFaceSet" token is found. The "IndexedFaceSet" should be followed by a "{" token, which eventually should match a "}" token, after which our parser should terminate. After the "{" token we could find a number of possible tokens, as described in the specification, but in a first pass you should only consider "coord" or "coordIndex". Only after the implementation of the parser supporting only these two fields is complete and debugged, you should attempt to add support for all the other IndexedFaceSet fields. Work incrementally by adding support for a few fields at a time, and only move on to add more nodes after debugging for the previous nodes is complete. You should create a few small test vrml files to help you debug your code. Of course, the file being parsed may contain a variety of syntax errors, which you need to detect and handle so that the parser does not crash. You should first implement the parser without paying attention to error handling, assuming that the input file has no syntax error, and you should test it with files which you had verified by visual inspection, or by loading the file into another VRML viewer such as MeshLab. Then you should enclose the whole parser in a try-catch block to handle errors in a consistent manner using exceptions. You may want to modify the Tokenizer get() and expecting() functions so that they throw exceptions. If you detect an error while parsing a VRML file, you should clear the Ifs class being constructed and return an error message or code. If you can identify the location in the input file where the error was detected, you should report it as part of the error message or code.

```
ifstream ifstr(inputFileName);
char cstr[512]
ifstr.getline(cstr,512);
string str = cstr;
if(str!=“#VRML V2.0 utf8”) { // ERROR }
Tokenizer tokenizer(ifstr);
// search for first occurrence of IndexedFaceSet
while(tokenizer.get() && tokenizer!=”IndexedFaceSet”);
// here we should have tokenizer==”IndexedFaceSet” or ... what else?
// what to do if no “IndexedFaceSet” token is found in the file ?
expecting(“{“);
while(tokenizer.get()) {
  if(tokenizer==”coord”) {
    expecting(“Coordinate“);
    expecting(“{“);
    expecting(“point“);
    expecting(“[“);
    while(tokenizer.get() && tokenizer!=”]”) {
      // ... parse float value from tokenizer and save value in Ifs._coord
      // ... throw exception if unable to
    }
    expecting(“}“);
  } else if(tokenizer==”coordIndex”) {
    expecting(“[“);
    while(tokenizer.get() && tokenizer!=”]”) {
      // ... parse int value from tokenizer and save value in Ifs._coordIndex
```

```
        // ... throw exception if unable to
      }

    // parse other IndexedFaceSet fields here, such as color, normal, ccw, etc
    // } else if(tokenizer==”ccw”) { ...

    } else if(tokenizer==”}”) {
      // matches “{“ found after “IndexedFaceSet”
      break;
    } else { // syntax ERROR
      throw string(“found unexpected token while parsing IndexedFaceSet”);
    }
  }
  // - you should throw an exception if any of the expecting() calls return false
  // - you should catch all the exceptions and exit gracefully in case of syntax errors
```