miro - base code

by Wojciech Jarosz
Spring - 2006

This code uses flex and bison to generate a lexer and parser, respectively. Hopefully you've all taken compilers, and will sort of understand how to write production rules using a grammar. If not, just look at what's there already, you should be able to copy+paste to make new rules. flex and bison are included in most BSD and *nix systems, if not, they are available from GNU (for free). For Windows users, I'm including distributable executables that are used automatically in the workspace.

Windows specific information:
(if you don't use Windows you don't need the Win32/ subdirectory):
There is a `.dsw` workspace file included in the Win32/ subdirectory. It should work with MS Visual C++ 6, and you should be able to convert it to .NET if you use that.

The other files in the Win32 directory are as follows:

- bison.exe - program that generates a parser from your grammar
- bison.hairy - support file for bison
- bison.simple - same as above
- flex.exe - program that generates your lexer
- miro.dsp - project file for MSVC++6
- miro.dsw - workspace file for MSVC++6
- unistd.h - empty, but required for the files flex and bison generate

You probably don't need to modify these files.

The rest of this document applies to everyone.

Compiling:

Windows users can use the workspace provided, everyone else should be able to use the makefile. Compilation has been tested on Windows, Linux, and MacOSX, if you have any problems, let me know.

Quickstart:

Once miro is compiled, you can run it from the command line. The code currently accepts one command line argument: miro ⟨script file⟩ where you replace ⟨script file⟩ with your scene description file. An example, called `example.txt`, is provided. It sets the screen dimensions, background color, camera options, and loads a single mesh (bunny.obj).

If you run miro with miro example.txt you should see a white wireframe bunny on a black background (which is what example.txt specifies!). You can move around with

```
   W
A     S
   D
```

and Q and Z move the camera up and down, respectively. '+' and '-' speed up and slow down the camera movement, respectively. Holding the left mouse button while moving around with the mouse will change the view direction. Combining this with the WASDQZ keys you should be able to navigate pretty easily. Pressing 'r' will switch to raytracing, 'g' takes you back to OpenGL mode. Pressing ESC will quit the program. 'i' will save a screenshot to the current directory of whatever is currently visible.

How the code works:

main.cpp

The program starts executing here, in main(). The first thing it does is initialize the graphics subsystem (OpenGL) using glut. The code by default will give you a simple wireframe OpenGL rendering of any geometry you load into it, but in the future you will be using OpenGL to plot pixels.

The next thing that happens is main initializes the camera, scene, and image objects, and then opens the file specified by the first command line arguement. When the user uses the "⟨script file⟩" option, main calls ParseFile(). ParseFile() is the entry point of the program into the parser. If you look in parse.y you will see lots of production rules, and code that initializes everything based on the input scene file. After parsing, main gives the scene an opportunity to do any necessary precalculations before rendering. It then enters glutMainLoop(), which loops forever, calling back into the code in opengl.cpp when events occur.

Lexing with flex:
This is an explanation of the format provided to you, not a guide to using flex, if you want that, go to `http://www.monmouth.com/∼wstreett/lex-yacc/flex_1.html`
The file lexer.lex contains all of the lexing statements. If you open up lexer.lex, you'll see some cryptic commands, followed by some C code. You don't really want to start modifying anything until the `%x` statements.

The `%x` statements define state names. If you look at `example.txt`, you see that the first statement is `global`, and then several options (width, height) are specified. These names are used later to keep track of what options are available to the user at a given point in the scene file. At the moment, `global`, `camera`, `triangle`, and `mesh` are defined. The `%%` statement means there are no more state declarations.

The remainder of the file contains the actual lexographical rules. Lets start with an example:

```
<*>enable{WS} { return ENABLE; }
<*>disable{WS} { return DISABLE; }
<*>cos{WS} { return MATH_COS; }
<*>sin{WS} { return MATH_SIN; }
```

The * in brackets means that the following symbol (e.g. `enable`) can appear anywhere, regardless of the current state (I'll explain that in a bit). Next is the name of the symbol itself, `enable`, followed by `{WS}`, which is defined above as whitespace characters. Finally, a statement returning a token appears. The tokens (in all caps by convention), are defined in parse.y.

What's happening here is these statements are simply looking for strings, and when some text matches, flex returns a token to the parser. So if you have `cos` somewhere in your scene file, it will be detected, and the parser will receive a token called `MATH_COS`. We'll get to the parser in a minute.

One more example:

```
<INITIAL>global{WS} { yy_push_state(global); return GLOBAL; }
<global>width{WS}   { return WIDTH; }
<global>height{WS}  { return HEIGHT; }
<global>background{WS}color{WS} { return BGCOLOR; }
```

Here we're defining the global block seen in the example scene file. The keyword `INITIAL` means the string can only appear in the root state. The code for `global` calls the function `yy_push_state` with global as the argument before returning a token. Note that the remainder of the global statements all have `<global>` as their state, meaning they can *only* appear within the global state. The other state blocks are similar. Later on we see the statements

```
<*>"}" {
  yy_pop_state();
  return yytext[0];
}
```

which pop the state (the current state is stored on a stack), and then return the first character of the matched string (which is '}' in this case). The other rules are pretty self explanatory.

Parsing with bison
This is an explanation of the format provided to you, not a guide to using bison, if you want that, go to `http://www.monmouth.com/∼wstreett/lex-yacc/bison.html`
parse.y contains the production rules and token declarations for using bison.

The first thing you see is a bunch of C code, ending with some variable declarations. For now only a few things are there, used for variables in the script file, and for adding objects to the scene. You can add more later if you need to.

Next are the bison token declarations. Here you see the tokens we used in the .lex file before. To return a token from the lexer you *must* first define it here. The next set of code is setup for variable types and operator precedence, leave it the way it is unless you really feel like hacking. After the `%%` is the actual grammar.

The script file is a set of blocks following each other. If you've taken compilers this should look familiar, if not, here's a quick rundown. The parser will look at the script file, and when the lexer returns tokens, it matches text and tokens to the grammatical rules you write here. When something matches, it executes any code you include in { }. The arguments start with $, and begin at 1, not 0. Just look at the existing rules to get the feel for how to write expressions. By following a rule down to its lowest definitions, and looking at the example file, you should be able to get an idea of how things work.

The current grammar allows you to create variables in the script file, which can be any words starting with $ followed by a letter.

At the very end of the file is the ParseFile() function called by main().

OpenGL with glut
You should have seen glut and OpenGL in previous classes. We'll be using glut callbacks for the base code, simply because it provides an easy, clean, cross-platform interface to mouse, keyboard, and 3d windowing functions.

Just about all of the glut code is in opengl.cpp. Most of it should look very familiar to you if you've taken CSE167. You probably won't have to touch this file much, it does most everything for you, unless you add keyboard commands. I'll leave comprehension of this file to you, some of the things it does with the camera and vectors are part of the course material. The file opengl.h is provided which just includes glut.h from the proper location depending on the platform. If you are using MacOSX you may have to change this file to get the project to compile.

The most important thing to notice is that in Display(), we call the Camera's Click() function.

Camera
The provided camera class provides you with basic functionality; it lets you specify an eye position, view direction, and up vector, and calls the scene's rendering functions to actually draw pixels to the screen.

Scene
This class lets the parser add objects, and draws everything in OpenGL mode. It has empty functions for rasterizing and raytracing that you will fill in later.

Image
The Image class lets you write to a virtual frame buffer, and then draw pixels to the screen at once using OpenGL. You'll be using this class for project 1 to get pixels to the screen.

Objects
Object is a base class for all your object classes. You're given a sphere, triangle, triangle mesh, and instance objects. The biggest thing to note here is that TriangleMesh already gives you code for loading .obj files. You probably don't want to mess with that. It uses vertex buffers to draw things to the screen.

Utility code
There are also some files included for general use. console.cpp and console.h define general-purpose writing functions, feel free to use them anywhere you want. They even write colors to the screen! You get a Warning, Debug, Error, and Fatal writing routines. Take a look at the code to see the differences.

Vector.h defines several vector and matrix classes that you'll find pretty useful. Some of the operations are done using operator overloading:

- ˆ - Vector dot product. Note that ĥas a low operator precedence, so always surround an expression using it with ()'s.

- `*` - Multiplication or cross product, depending on the context. Note that if you do matrix*vector multiplication the matrix must be on the *left* side.

Other operations should be self explanatory (Normalize, Norm, etc.). You can always cast a `Vector3` to a `Vector4` and it will automatically get its `.w` component set to 1.