EECE 478: Assignment 1

Alex Kropivny

August 27, 2010

Contents

1 Introduction

Assignment 1 is mostly a straightforward refresher of general C/C++ programming techniques and basic OpenGL/GLUT API use.

To make things interesting, use unfamiliar technology, and provide well-documented code that's easy to understand and mark I have chosen to document my C code in a literate programming style, as described by Donald Knuth in 1984. If this goes well, the result should be a document describing the implementation of my C program with extreme clarity and detail.

1.1 Usage Instructions

The application controls are simple. With the mouse, use left click to rotate the camera and right click to move the model around. Arrow keys allow you to zoom and rotate the model, space bar switches between different models, and Q or ESC exit the program.

1.2 Literate Programming

This uncommon technology deserves a brief explanation. Literate programming combines natural language descriptions of a program (documentation) alongside the actual code of the program (implementation) in the same file. A preprocessor then takes this file, and from it produces two separate things; nicely formatted documentation, and the actual compileable code.

The natural language descriptions do what comments mostly fail to: provide a clear view of the programmer's train of thought. This is further assisted by a very simple macro system used by the LP preprocessor, which allows the programmer to structure code according to how a human thinks, rather than what the C compiler requires. (This is what makes it literate 'programming', rather than just fancy documentation.)

What this all boils down to are two programs that 'tangle' and 'weave' a .nw file (I use a LP preprocessor called noweb) into a bunch of .c and .h files and a .tex file, respectively. The .nw file is formatted into code and documentation chunks. Documentation chunks are unnamed, code chunks are named and can be inserted within other code chunks like include files in C.

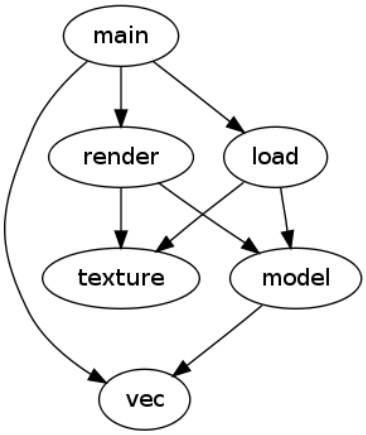
1.3 Modular Design

iterate programming helps write better code, but does not do so automatically. One problem it runs into is clear from the start, by the presence of the 'web' analogy. On a small scope it's normal for complex code to form a small web, but with larger scopes it is a nightmare scenario and should be avoided at all costs.

Thus, standard design techniques still apply. The first concern is factoring the project into modules and their interdependencies.

Since it's a fairly small project, Figure 1 says it all. Since we're dealing with C, the individual modules are .c files with interfaces defined in .h files.

Figure 1: Module dependencies



I choose 'noweb' as the literate preprocessor due to its' simplicity: the price of that simplicity is all the C code in the project will be stored in a single .nw file. This is atypical as far as programming goes, but I will give it a go. Despite everything being in one file, the code will still be split up according to the above design.

2 GLUT Application

We'll start with a top down look at the code.

The base of the program looks familiar.

3 ⟨*main.c* 3⟩≡

```
⟨Copyright 34b⟩
⟨Main includes 4a⟩
⟨Globals 5a⟩

⟨GLUT functions 6⟩

// registered with atexit
void cleanup()
{
    ⟨Global cleanup 13e⟩
}

int main(int argc, char** argv)
{
```

```
    ⟨Global initialization 4b⟩
    glutMainLoop();
    return 0;
}
```

Defines:
`cleanup`, used in chunk 4b.
`main`, never used.

2.1 Boilerplate

Initializing and configuring GLUT and OpenGL consists of mostly boilerplate code. The interesting details are mostly hidden in other modules.

4a ⟨*Main includes* 4a⟩≡ (3) 9b▷

```
#include <GL/glut.h>
#include <stdlib.h> // exit() and atexit()
#include <string.h> // strlen(), used once
```

4b ⟨*Global initialization* 4b⟩≡ (3) 5b▷

```
// window creation
glutInit(&argc, argv);
glutInitWindowSize(winHeight, winWidth);
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
glutCreateWindow("EECE478 Assignment 1");

// callback registration
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMotionFunc(motion); // in case I want to differentiate between buttons,
glutPassiveMotionFunc(motion); // I'll handle press detection myself
glutMouseFunc(click);
glutKeyboardFunc(keyboard);
glutSpecialFunc(special_key); // like glutKeyboardFunc, but handles non-ASCII keys
atexit(cleanup);
```

Uses `cleanup` 3, `click` 10, `display` 12, `keyboard` 6, `motion` 10, `reshape` 7, `special_key` 8, `winHeight` 5a, and `winWidth` 5a.

There's a bunch of mutable state that must be tracked, mostly for UI or option purposes.

5a ⟨*Globals* 5a⟩≡ (3) 13c▷

```
// dimensions
int winHeight = 480;
int winWidth = 640;
// mouse state
int lmbDown=0, rmbDown=0, mouseX, mouseY;
// trackball state
vec3 trackballPoint = {0,0,0};
// option states
int what_to_draw = 0;
#define DRAW_TEAPOT 0
#define DRAW_MODEL 1
#define DRAW_CUBE 2
#define DRAW_AXES 3
int do_lighting = 1;
int do_textures = 1;
```

Defines:
`do_lighting`, used in chunks 6 and 12.
`do_textures`, used in chunks 6 and 14.
`DRAW_AXES`, used in chunks 6 and 12.
`DRAW_CUBE`, used in chunk 12.
`DRAW_MODEL`, used in chunk 12.
`DRAW_TEAPOT`, used in chunk 12.
`lmbDown`, used in chunk 10.
`what_to_draw`, used in chunks 6 and 12.
`winHeight`, used in chunks 4b, 7, 10, and 12.
`winWidth`, used in chunks 4b, 7, 10, and 12.

Actual graphics options come next.

5b ⟨*Global initialization* 4b⟩+≡ (3) ◁4b 13a▷

```
// graphics configuration
glEnable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);
glDepthMask(GL_TRUE);
glCullFace(GL_BACK);

glShadeModel(GL_SMOOTH);
```

6 ⟨*GLUT functions* 6⟩≡ (3) 7▷

```
void keyboard(unsigned char key, int x, int y)
{
    switch(key) {
    case 'Q':
    case 'q':
    case 0x1b: // exit with ESC|q|Q
        exit(0);
        break;
    case ' ': // toggle model to draw with SPACE
        what_to_draw = what_to_draw==DRAW_AXES ? 0 : what_to_draw+1;
        glutPostRedisplay();
        break;
    case 'l':
    case 'L': // doesn't work for more than 1 frame for some reason
        do_lighting = !do_lighting;
        if (do_lighting)
            glEnable(GL_LIGHTING);
        else
            glDisable(GL_LIGHTING);
        glutPostRedisplay();
        break;
    case 't':
    case 'T':
        do_textures = !do_textures;
        glutPostRedisplay();
        break;
    }
};
```

Defines:
`keyboard`, used in chunk 4b.
Uses `do_lighting` 5a, `do_textures` 5a, `DRAW_AXES` 5a, `model` 15, and `what_to_draw` 5a.

2.2 Camera Controls

Rather than track an actual camera position and orientation, then update the projection matrix to match them, here it's sufficient to use the projection matrix to keep track of all state. This greatly simplifies the code.

Rotations are always done about the origin, and aren't affected by camera translations. This interface is limited, but very easy to use.

The coordinate system treats positive y axis as 'up', and positive z axis (initially) pointed away from the user.

7 ⟨*GLUT functions* 6⟩+≡ (3) ◁6 8▷

```
void reshape(GLint width, GLint height)
{
    glViewport(0,0,width,height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective( 90.0 // field of view
                  , (float)width/height // aspect ratio
                  , 0.1  // near clipping plane
                  , 1000 // far clipping plane
                  );
    // set camera TODO
    gluLookAt(-10, 10, -10, // eye position
              0, 0, 0,  // focus position
              0, 1, 0);  // 'up' vector
    glMatrixMode(GL_MODELVIEW);
    winHeight = height;
    winWidth = width;
};
```

Defines:
`reshape`, used in chunk 4b.
Uses `winHeight` 5a and `winWidth` 5a.

With a confusing name, glutSpecialFunc() is equivalent to the glutKeyboardFunc() but covers non-ASCII keys. With it, and glutGetModifiers() all forms of keyboard input should be covered.

8 ⟨*GLUT functions* 6⟩+≡ (3) ◁7 10▷

⟨*Cam helper functions* 9a⟩

```
void special_key(int key, int x, int y)
{
    switch(key) {
    // rotate and zoom with the ARROW KEYS
    case GLUT_KEY_UP:
        zoomCam(1.1);
    break;
    case GLUT_KEY_DOWN:
        zoomCam(0.9);
    break;
    case GLUT_KEY_LEFT:
        rotCam(-5);
    break;
    case GLUT_KEY_RIGHT:
        rotCam(5);
    break;
    }
};
```

Defines:
special_key, used in chunk 4b.
Uses rotCam 9a and zoomCam 9a.

9a ⟨*Cam helper functions* 9a⟩≡ (8)

```
void zoomCam(float f)
{
    // Zooming the camera is just zooming about the origin, just need to
    // make sure the right matrix is set.
    glMatrixMode(GL_PROJECTION);
    glScalef(f,f,f);
    glMatrixMode(GL_MODELVIEW);
    glutPostRedisplay();
}

void rotCam(float f)
{
    // Same for rotations.
    glMatrixMode(GL_PROJECTION);
    glRotatef(f, 0, 1, 0); // y-axis is 'up'
    glMatrixMode(GL_MODELVIEW);
    glutPostRedisplay();
}
```

Defines:
`rotCam`, used in chunk 8.
`zoomCam`, used in chunk 8.
Uses `f` 32c.

The current 'trackball' mouse-rotation code is a bit simplistic and doesn't work as it ideally should. Rotations are always done about the same axes, even when previous rotations have made the apparent axes of rotation change.

As well, the code is too complex and independant to belong in main.c: it should be moved out into a separate camera module. Ideally, I'd like to figure out the whole quaternion business before deciding which parts to move out and which parts to leave, though.

9b ⟨*Main includes* 4a⟩+≡ (3) ◁4a 13b▷

```
#include <math.h>
#include <stdio.h>
#include "vec.h"
```

10 ⟨*GLUT functions 6*⟩+≡ (3) ◁8 12▷

```
// state tracking, no side effects
void click(int button, int state, int x, int y)
{
    switch(button) {
    case GLUT_LEFT_BUTTON:
        lmbDown = state==GLUT_DOWN;
    break;
    case GLUT_RIGHT_BUTTON:
        rmbDown = state==GLUT_DOWN;
    break;
    }
};

#define MIN(a, b) (a)<(b) ? (a) : (b)

#define PI 3.14159265
#define RAD_TO_DEG 180/PI

void motion(int x, int y)
{
    // Left mouse button rotates around the origin.
    if (lmbDown) {
        // find this point on a virtual sphere "trackball"
        //
        // first convert sx and sy from (0..width,0..height) to
        // (-1..1,-1..1) range
        //
        // to ignore aspect ratio, scale factor is identical for x and y
        float scale = (float)(MIN(winWidth,winHeight)/2);
        vec3 sph;
        sph.x = (float)x/scale-1.0;  sph.y = (float)y/scale-1.0;
        // z is calculated from x,y since they're on a unit sphere
        float tmp = (1-sph.x*sph.x-sph.y*sph.y);
        sph.z = tmp>0 ? sqrt(tmp) : 0; // guard against negative sqrt()
        // the result should be a unit vector if the point is ON the sphere
        // if you click outside it, though, it's not - so normalize!
        normalize(&sph);
        //scale = sqrt(sph.z*sph.z+sph.x*sph.x+sph.y*sph.y);
        //sph.x = sph.x/scale; sph.y=sph.y/scale; sph.z=sph.z/scale;
        //printf("Sphere: %f %f %f\n", sph.x, sph.y, sph.z);

        // compute axis and angle of rotation relative to previous state
        float theta = RAD_TO_DEG*acos(dot(trackballPoint,sph));
        vec3 axis   = cross(trackballPoint,sph);

        // implement rotation
        // TODO: quaternionize this
        glMatrixMode(GL_PROJECTION);
        glRotatef(theta, axis.x, axis.y, axis.z);
```

```
            //printf("\t\tRotate: %f $ %f %f %f\n", theta, axis.x, axis.y, axis.z);
            glMatrixMode(GL_MODELVIEW);
            glutPostRedisplay();

            // store state
            trackballPoint = sph;
        } else {
            // store null state if button not pressed, to avoid 'jerking' on
            // first press (cross product results in rotation on (0,0,0) axis)
            vec3 noRot = {0,0,0};
            trackballPoint = noRot;
        }
        // Right mouse button moves on the X and Z axes. (Non-vertical ones.)
        // The origin of the camera's rotations stays the same.
        if (rmbDown) {
            glMatrixMode(GL_PROJECTION);
            glTranslatef((float)(mouseX-x)/10,0,(float)(mouseY-y)/10);
            glMatrixMode(GL_MODELVIEW);
            glutPostRedisplay();
        }

        // keep track of last state for delta calculations
        mouseX = x; mouseY = y;
    }
```

Defines:
`click`, used in chunk 4b.
`MIN`, never used.
`motion`, used in chunk 4b.
`PI`, never used.
`RAD_TO_DEG`, never used.
Uses `dot` 34a, `f` 32c, `lmbDown` 5a, `normalize` 34a, `printf` 19, `winHeight` 5a, and `winWidth` 5a.

2.3 Display

12 ⟨*GLUT functions* 6⟩+≡ (3) ◁10

⟨*Test render functions* 36⟩

```
void renderName()
{
    ⟨Name definition 35⟩
    glDisable(GL_LIGHTING);
    glColor3f(1,1,1);
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    glRasterPos2f((float)(winWidth-8*strlen(AUTHOR)*2) / winWidth,
                  (float)(26-winHeight) / winHeight);
    //glutBitmapString(GLUT_BITMAP_8_BY_13, AUTHOR);
    for (char* c=AUTHOR; *c!=0; c++) {
        glutBitmapCharacter(GLUT_BITMAP_8_BY_13, *c);
    }
    glPopMatrix();
    if (do_lighting) glEnable(GL_LIGHTING);
}

void display(void)
{
    // clear
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // draw
    if (acity) {
        ⟨Render city 32c⟩
    } else {
        switch(what_to_draw) {
        case DRAW_TEAPOT:
            glColor3f(0,0,1);
            glFrontFace(GL_CW); // teapot has 'wrong' normals
            glutSolidTeapot(2.5);
            glFrontFace(GL_CCW);
            break;
        case DRAW_MODEL:
            ⟨Draw a model 14⟩
            break;
        case DRAW_CUBE:
            Cube();
            break;
        case DRAW_AXES:
            Axes();
            break;
        }
    }
```

```
    renderName();
    // flip
    glutSwapBuffers();
}
```

Defines:
  display, used in chunk 4b.
  renderName, never used.
Uses AUTHOR 35, Axes 36, Cube 36, do_lighting 5a, DRAW_AXES 5a, DRAW_CUBE 5a, DRAW_MODEL 5a, DRAW_TEAPOT 5a, what_to_draw 5a, winHeight 5a, and winWidth 5a.

2.4 Lighting

13a ⟨*Global initialization* 4b⟩+≡ (3) ◁5b 13d▷

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

glLightfv(GL_LIGHT0, GL_POSITION, (GLfloat[4]){60,60,60,1.0});
glLightfv(GL_LIGHT0, GL_AMBIENT, (GLfloat[4]){0.5, 0.5, 1.0, 0.6});
glLightfv(GL_LIGHT0, GL_DIFFUSE, (GLfloat[4]){0.5, 0.5, 1.0, 1.0});
glLightfv(GL_LIGHT0, GL_SPECULAR, (GLfloat[4]){1.0, 1.0, 1.0, 1.0});

glLightModelfv(GL_LIGHT_MODEL_AMBIENT, (GLfloat[4]){0.1,0.1,0.1,1.0});
```

2.5 Model

All the work has been done in the load and render modules. Here, we merely use the interfaces those modules expose.

13b ⟨*Main includes* 4a⟩+≡ (3) ◁9b 31b▷

```
#include "load.h"
#include "render.h"
```

13c ⟨*Globals* 5a⟩+≡ (3) ◁5a 31c▷

```
model* abuilding=0;
```

Uses model 15.

13d ⟨*Global initialization* 4b⟩+≡ (3) ◁13a 31d▷

```
#define MODEL_FILE "./TechnologyEnterpriseFacility_Gregor/TechnologyEnterpriseFacility_Gregor.model"
abuilding = loadModel(MODEL_FILE);
if (!abuilding) exit(1);
```

Defines:
  MODEL_FILE, never used.
Uses model 15.

13e ⟨*Global cleanup* 13e⟩≡ (3) 32a▷

```
freeModel(abuilding);
```

Uses freeModel 17.

14 ⟨*Draw a model* 14⟩≡ (12)

```
if (do_textures) {
    glEnable(GL_TEXTURE_2D);
    renderModel(abuilding);
    glDisable(GL_TEXTURE_2D);
} else renderModel(abuilding);
```

Uses `do_textures` 5a and `renderModel` 24.

3 Model

There are two major tasks: loading the model from disk, and drawing it. Both use the same set of data structures, so there's three interdependant modules already.

The reasonable representation in C would be two separate .c files for loading and rendering (load.c, render.c, and their headers) and an additional header for the data structure declarations.

3.1 Data Representation

We'll begin with the data structure. In the file, the model structure is specified with a list of triangles that refers to a list of vertices using integer indexes. A very non-sequential access pattern, but compact and easy to implement. For small models, it'll work great without any modification.

The model is completely described by one big structure, to make loading and freeing them easy. The code is written for readability, not performance - preemptive optimization is a sin anyway.

Note that in C, structures need to have an associated typedef for this kind of code to compile. I think this makes it so you can avoid the "struct" keyword when defining variables with them later? You can avoid writing the name twice by only putting it at the end of the definition, but I think being able to see the name at the start makes it a lot more readable.[1]

15 ⟨*model.h* 15⟩≡

```
⟨Copyright 34b⟩
#ifndef _MODEL_H_
#define _MODEL_H_
#include "vec.h"
#include "texture.h"

typedef unsigned char list_index; // if we happen to need a bigger range later

typedef struct triangle {
    list_index a,b,c; // vertex indexes
    list_index texture; // hey, 4-byte alignment! :)
    vec2 tex_a,tex_b,tex_c; // texture x,y coordinates for each vertex
    vec3 normal; // is this used for facing? why not clockwise order?
                 // maybe pre-calculation for lighting?
} triangle;

typedef struct model {
    char* name;
    // list of vertexes
    vec3* verts; list_index num_verts;
```

[1] The names at the start and the end can actually be different - the first one is used in definitions that use the `struct` keyword, while the second gets used in the kind of definitions I use.

```
    // list of triangles built on the vertexes
    triangle* tris; list_index num_tris;
    // list of textures
    texture* tex; list_index num_tex;
    //  note: textures tend to get re-used, so this will probably just be a
    //  texture id 'lookup' table, that turns the id in the model file into
    //  an id used by the texture loading+storage system... Or something.
    // figure it out later
} model;

#endif
```

Defines:
_MODEL_H_, never used.
list_index, used in chunks 22, 24, 28a, and 30–32.
model, used in chunks 6, 13, 16, 17, 19, 23, 24, and 28–32.
triangle, used in chunks 19, 22, and 24.
Uses texture 25.

3.2 Loading

We've got a data type for the model! Now we need functions to operate on it. I could just use C++ and OOP, but it doesn't seem worth the effort, especially since the OpenGL interface is nowhere near OOP.

Loading a model from a file, and freeing memory used by a loaded model are the only operations we need.

16 ⟨*load.h* 16⟩≡

```
⟨Copyright 34b⟩
#ifndef _LOAD_H_
#define _LOAD_H_
#include "model.h"

// loads a model from the given filename
// on fail, prints error message to stderr and returns null
model* loadModel(char* filename);

// the model structure contains a bunch of dynamic size lists
void freeModel(model* m);

#endif
```

Defines:
_LOAD_H_, never used.
Uses freeModel 17 and model 15.

The implementation of freeModel() is straightforward. The complexity lies in loadModel(). Since the data structure uses simple lists of unknown length, we need to find how much memory the model will use before we can read it from disk. String manipulation in C can be tricky, so a few unexported helper functions are needed.

17 ⟨*load.c* 17⟩≡

```
⟨Copyright 34b⟩
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "model.h"

void freeModel(model* m)
{
    // ensure it's not a nullpointer
    if (!m) return;
    // clean up deep pointers before removing object
    free(m->name);
    free(m->verts);
    free(m->tris);
    free(m->tex);
    free(m);
}

// We'll be using this for string processing. Note that it sets an upper limit
// on line length in the model file - any longer, and bugs will occur!
#define BUFF_SIZE 512

⟨File IO utility functions 20⟩

model* loadModel(char *filename)
{
    // zero model memory on allocation
    model* m = (model*)calloc(1, sizeof(model));
    // general purpose string buffer for parsing
    char s[BUFF_SIZE]; s[0]=0;
    // figure out what folder the model+textures are in
    char filepath[BUFF_SIZE];
    strcpy(filepath,filename);
    for (int i=strlen(filepath); 1; i--) {
        if (filepath[i]=='/') {
            filepath[i+1]=0;
            break;
        }
        if (i==0) {
            strcpy(filepath,"./");
            break;
        }
    }
```

```
    printf("Loading [%s]...\n", filename);
    // try to open the file, handling errors
    FILE* f = fopen(filename, "rb");
    if (!f) {perror("loadModel() error:"); return NULL;}
    ⟨Count and allocate memory for vertices triangles and textures 19⟩
    ⟨Read vertices triangles and textures 22⟩
    fclose(f);
    printf("\tDone!\n");
    return m;
}
```

Defines:
BUFF_SIZE, used in chunks 19–22.
freeModel, used in chunks 13e, 16, and 31a.
Uses f 32c, model 15, and printf 19.

Our data is stored in individual sections. Sections begin with `<sectioname>` tags, and end with `</sectionname>` tags. The tags are on their own lines. Within a section, all lines that aren't empty or start with a `#` contain content.

Since we need to allocate memory for our data before we can read it, loading is a two-pass process. The first pass iterates over lines with fgets(), and counts all non-empty, non-comment lines between appropriate tags. To avoid copy and pasting similar code, a macro is used.[2]

Needless to say, string manipulation isn't the strong point of C. Excess whitespace should be dealt with by spaces around the `sscanf()` formats and `strstr()` rather than `strcmp()` tag searching, but comments not starting in the first column will break things, as will 'blank' lines with whitespace. So might many other things - the best I can promise is a semi-useful error message.

19 ⟨*Count and allocate memory for vertices triangles and textures* 19⟩≡ (17)

```
int ret; // general-purpose return code holder

// read name while we're passing through
if (!seekstr(f,s,"<name>")) return NULL;
fgets(s, BUFF_SIZE, f); // this keeps the \n at the end - should I fix it?
m->name = (char*)malloc(strlen(s)+1); // could just use strncpy and strlen-1...
strcpy(m->name, s);
if (!seekstr(f,s,"</name>")) return NULL;

// Helper macro, note the {...}! Not needed, but good practice.
// Since the scope, and usefulness of this macro is -extremely- local, the
//  benefit of brevity and readability outweighs the fact that macros suck.
#define COUNT_LINES(start,stop,count) {                         \
    if (!seekstr(f,s,start)) return NULL;                       \
    while ((ret=iter_line(f,s,stop))==ITER_NEXT) (count)++;     \
    if (ret==ITER_ERROR) return NULL;                           \
                                    }
COUNT_LINES("<textures>", "</textures>", m->num_tex);
COUNT_LINES("<vertices>", "</vertices>", m->num_verts);
COUNT_LINES("<triangles>", "</triangles>", m->num_tris);

// allocate space
m->tex = (texture*)calloc(m->num_tex, sizeof(texture));
m->verts = (vec3*)calloc(m->num_verts, sizeof(vec3));
m->tris = (triangle*)calloc(m->num_tris, sizeof(triangle));

printf("\t%d bytes of memory allocated (%d*%dtex %d*%dvrt %d*%dtri)\n",
        sizeof(texture)*m->num_tex+sizeof(vec3)*m->num_verts
            +sizeof(triangle)*m->num_tris+sizeof(model),
```

[2]This could be done with a function, but error checking would make the code longer and more repetitive. Repetition leads to copy and paste, copy and paste leads to bugs just as insidious and subtle as those caused by macros.

The macro basically simulates an exception: unconditional return from a specific function, triggered by one of its' children. This could be implemented via setjmp.h, but I assume goto is even worse than macros!

Macros do suck; however, copy and paste sucks more!

```
        m->num_tex, sizeof(texture),
        m->num_verts, sizeof(vec3),
        m->num_tris, sizeof(triangle));
```

Defines:
COUNT_LINES, never used.
printf, used in chunks 10, 17, 26, and 30.
ret, used in chunk 30.
Uses BUFF_SIZE 17 26, f 32c, ITER_ERROR 21, iter_line 21, ITER_NEXT 21, model 15, seekstr 20, texture 25, and triangle 15.

To avoid (reduce) gratuitous copy and paste, some string IO utility functions have been implemented. The first simply seeks forward through a file until it passes a specific line. This is used to easily locate tags.

20 ⟨*File IO utility functions* 20⟩≡ (17) 21▷

```
// Seek a needle line in a haystack of lines.
// Returns 1 on success, 0 on EOF or error.
int seekstr(FILE* f, char* s, const char* needle)
{
    // Be wary that due to how fgets works, lines longer than BUFF_SIZE
    // will wreak havoc.
    // The needle can be part of the line, rather than whole. This
    // makes the code more robust with respect to excess whitespace.
    while (fgets(s, BUFF_SIZE, f)) {
        if (s[0]!='#' && strstr(s,needle)) return 1;
    }
    if (feof(f)) {
        fprintf(stderr,"seekstr() error: tag %s not found.\n", needle);
        return 0;
    }
    if (ferror(f)) {
        perror("Error while seeking string in file");
        return 0;
    }
    fprintf(stderr,"Unknown seekstr() error.\n");
    return 0;
}
```

Defines:
seekstr, used in chunks 19 and 22.
Uses BUFF_SIZE 17 26 and f 32c.

The second utility function allows us to easily iterate over all lines in a section.

21 ⟨*File IO utility functions* 20⟩+≡ (17) ◁20

```
// Reads the next non-comment, non-empty, non-stop line into s.
// Returns:
#define ITER_NEXT 0 // on success
#define ITER_STOP 1 // on encountering stop string
#define ITER_ERROR 2 // on error/eof, prints error to stderr
int iter_line(FILE* f, char* s, const char* stop)
{
    while (fgets(s, BUFF_SIZE, f)) {
        if ((s[0]=='#') | (s[0]=='\n')) continue; // skip comments/empty lines
        if (strstr(s,stop))
            return ITER_STOP;
        else
            return ITER_NEXT;
    }
    // fgets failed - why?
    if (feof(f)) {
        // if the file doesn't end with a \n, EOF is raised despite a valid
        // line being read - here's a handler for that edge case:
        //if (strstr(s,stop)) return ITER_STOP;
        // since we know the file can't have useful data on the very last line
        // only ITER_STOP must be handled
        fprintf(stderr,"iter_line() error: stop tag %s never found.\n", stop);
        return ITER_ERROR;
    }
    if (ferror(f)) {
        perror("Error while seeking string in file");
        return ITER_ERROR;
    }
    fprintf(stderr,"Unknown iter_line() error.\n");
    return ITER_ERROR;
}
```

Defines:
ITER_ERROR, used in chunk 19.
iter_line, used in chunks 19 and 22.
ITER_NEXT, used in chunks 19 and 22.
ITER_STOP, never used.
Uses BUFF_SIZE 17 26 and f 32c.

These utility functions let us neatly implement the second pass.

The second pass in model loading actually uses scanf while iterating over all lines to read the detailed data the model consists of.

(Note that returning NULL without freeing allocated memory results in a memory leak, though with this kind of failure it hardly matters.)

22 ⟨*Read vertices triangles and textures* 22⟩≡ (17)

```
rewind(f);
// not bothering to error check seekstr since the first pass succeeded:
// on the off chance that reads start failing, stderr will get spammed
// with a few error messages then a scanf() will end up failing
// (so, loadModel's invariants still hold since NULL gets returned)
seekstr(f,s,"<textures>");
for (list_index i=0; iter_line(f,s,"</textures>")==ITER_NEXT; i++) {
    // string manipulation in C is such a waste of time
    char path[BUFF_SIZE]; path[0]=0;
    strcat(path, filepath);
    strcat(path, s);
    path[strlen(path)-1]=0; // cut off \n
    m->tex[i] = loadTexture(path);
    if (!m->tex[i].data) {
        fprintf(stderr, "loadModel() failed: could not load texture.\n");
        return NULL;
    }
}

// Note spaces at beginning and end of sscanf format string.
// These should get rid of any leading/trailing whitespace that comes up.

seekstr(f,s,"<vertices>");
for (list_index i=0; iter_line(f,s,"</vertices>")==ITER_NEXT; i++) {
    vec3* v = &m->verts[i]; // & is lower precedence than -> and []
    if (3!=sscanf(s, " %f %f %f ", &v->x, &v->y, &v->z)) {
        fprintf(stderr,"loadModel(): error parsing vertex %d.\n", i);
        return NULL;
    }
}

seekstr(f,s,"<triangles>");
for (list_index i=0; iter_line(f,s,"</triangles>")==ITER_NEXT; i++) {
    triangle* t = &m->tris[i];
    if (13!=sscanf(s, " %d %d %d %f %f %f %d %f %f %f %f %f %f ",
                   &t->a, &t->b, &t->c,      // vertex indexes
                   &t->normal.x, &t->normal.y, &t->normal.z,
                   &t->texture,              // texture index
                   &t->tex_a.x, &t->tex_a.y, // tex coordinate tuples
                   &t->tex_b.x, &t->tex_b.y, //  for each vertex
                   &t->tex_c.x, &t->tex_c.y)) {
        fprintf(stderr,"loadModel(): error parsing triangle %d.\n", i);
        return NULL;
```

```
    }
}
```

Uses BUFF_SIZE 17 26, f 32c, iter_line 21, ITER_NEXT 21, list_index 15, seekstr 20, texture 25, and triangle 15.

3.3 Rendering

The model data structure is fairly intuitive to render.

23 ⟨*render.h* 23⟩≡

```
⟨Copyright 34b⟩
#ifndef _RENDER_H_
#define _RENDER_H_
#include "model.h"

void renderModel(model* m);

#endif
```

Defines:
_RENDER_H_, never used.
Uses model 15 and renderModel 24.

24 ⟨*render.c* 24⟩≡

```
⟨Copyright 34b⟩
#include "GL/gl.h"
#include "render.h"

void renderModel(model* m)
{
    for (list_index i=0; i<m->num_tris; i++) {
        triangle* tri = &m->tris[i];
        vec3 v;

        // quick hacky way to get some colour
        //glColor3f((float)(i%100)/100+0.1,
        //          (float)((i-100)%100)/100+0.1,
        //          (float)((i-200)%100)/100+0.1);

        if (m->num_tex) {
            glBindTexture(GL_TEXTURE_2D, m->tex[tri->texture].id);
            glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
        }

        glBegin(GL_TRIANGLES);

        v = m->verts[tri->a];
        glNormal3f(tri->normal.x,tri->normal.y,tri->normal.z);
        glTexCoord2f(tri->tex_a.x,tri->tex_a.y);
        glVertex3f(v.x,v.y,v.z);
        v = m->verts[tri->b];
        glNormal3f(tri->normal.x,tri->normal.y,tri->normal.z);
        glTexCoord2f(tri->tex_b.x,tri->tex_b.y);
        glVertex3f(v.x,v.y,v.z);
        v = m->verts[tri->c];
        glNormal3f(tri->normal.x,tri->normal.y,tri->normal.z);
        glTexCoord2f(tri->tex_c.x,tri->tex_c.y);
        glVertex3f(v.x,v.y,v.z);

        glEnd();
    }
}
```

Defines:
`renderModel`, used in chunks 14, 23, and 32c.
Uses `list_index` 15, `model` 15, `texture` 25, and `triangle` 15.

3.4 Textures

The code to load textures is a tad long, so it goes into separate files.

25 ⟨*texture.h* 25⟩≡

```
#ifndef _TEXTURE_H_
#define _TEXTURE_H_

typedef struct texture {
    unsigned int id;      // openGL texture ID
    char* data;           // pixel buffer
    int width,height,bpp; // bytes per pixel
} texture;

texture loadTexture(const char* filename);
void freeTexture(texture* tex);

#endif
```

Defines:
  `_TEXTURE_H_`, never used.
  `texture`, used in chunks 15, 19, 22, 24, and 26.
Uses `freeTexture` 26.

26 ⟨*texture.c* 26⟩≡

```
⟨Copyright 34b⟩
#include <stdio.h>
#include <stdlib.h>
#include <GL/gl.h>
#include "texture.h"

#define BUFF_SIZE 1024
#define ERR_STR "loadTexture() error:"
texture loadTexture(const char* filename) {
    printf("\tLoading [%s]...\n", filename);
    texture tex;
    tex.data = NULL;

    FILE* f = fopen(filename, "rb");
    if (!f) {perror(ERR_STR); goto loadTexError;}
    char s [BUFF_SIZE]; s[0]=0; // generic buffer
    int n; // generic counter
    // check magic word
    n = fread(s, 1, 2, f); // read 2 one-byte blocks
    if (n!=2) {
        if (ferror(f)) perror(ERR_STR);
        if (feof(f)) fprintf(stderr, "%s texture file %s too short!",
                                                    ERR_STR, filename);
        goto loadTexError;
    }
    if ((s[0]!='P') & (s[1]!='6')) {
        fprintf(stderr, "%s no magic byte found, non-PPM file!", ERR_STR);
        goto loadTexError;
    }
    // read width, height, maxval
    retry: // goto sucks, but people who don't follow format definitions suck more
    if (3!=fscanf(f, " %d %d %d ", &tex.width, &tex.height, &tex.bpp)) {
        fgets(s,BUFF_SIZE,f);
        if (s[0]=='#' || s[0]=='\n') goto retry;
        fprintf(stderr, "%s could not read header data.", ERR_STR);
        goto loadTexError;
    }
    // should be <, I think these ppm files are wrong
    tex.bpp = (tex.bpp<=256) ? 1 : 2;
    // read data
    int size = tex.width*tex.height*tex.bpp*3; // total size to read
    tex.data = (char*)malloc(size);
    if (size!=fread(tex.data, 1, size, f)) {
        if (feof(f)) fprintf(stderr, "%s file too short!", ERR_STR);
        if (ferror(f)) perror(ERR_STR);
    }
    printf(" (%d bytes)\n", size);
    fclose(f);
```

```
    // now register the texture with OpenGL
    glGenTextures(1, &tex.id);
    glBindTexture(GL_TEXTURE_2D, tex.id);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tex.width, tex.height, 0,
                          GL_RGB, GL_UNSIGNED_BYTE, tex.data);

    // success, clean up and return texture
    return tex;

    // poor man's exception handling (hey at least it's lightweight!)
    loadTexError:
    //free(tex);
    if (tex.data) free(tex.data);
    tex.data = NULL;
    //fclose(f);
    return tex;
}

void freeTexture(texture* tex)
{
    if (tex->data) free(tex->data);
    if (tex) free(tex);
}
```

Defines:
  BUFF_SIZE, used in chunks 19–22.
  ERR_STR, never used.
  freeTexture, used in chunk 25.
Uses f 32c, printf 19, and texture 25.

4 City

A city is a group of models, transformed in various ways. It's described by a .city file. Two files, city.c and city.h provide access to it.

4.1 Data Representation

We need a data structure equivalent to the information stored in a .city file. This is done by having a simple city structure, containing an array of buildings. The buildings wrap size rotation and translation data around models.

28a ⟨*city.h* 28a⟩≡ 28b▷

```
⟨Copyright 34b⟩
#ifndef _CITY_H_
#define _CITY_H_
#include "model.h"
#include "vec.h"

typedef struct building {
    model* model;
    char* path;    // where to load model from
    vec3 rt,tx,sc; // rotate translate scale
} building;

typedef struct city {
    char* name;
    list_index num_buildings;
    building* buildings;
} city;
```

Defines:
_CITY_H_, never used.
building, used in chunks 30 and 32c.
city, used in chunks 28–31.
Uses list_index 15 and model 15.

Like models, cities can be loaded and freed.

28b ⟨*city.h* 28a⟩+≡ ◁28a

```
city* loadCity(char* file);
void freeCity(city* c);

#endif
```

Uses city 28a and freeCity 31a.

4.2 Loading

29 ⟨*city.c* 29⟩≡

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "city.h"
#include "model.h"
⟨Loading a city 30⟩
⟨Freeing a city 31a⟩
```

Uses `city` 28a and `model` 15.

The file format is significantly simpler than for models, which shortens the code significantly.

I should technically cleanup before returning, probably using an exception-like goto to do a freeCity() in the end, instead of just a return NULL. However, loadCity is currently a once-in-a-lifetime function so there's no point. Minor memory leaks are tolerable in this case.

30 ⟨*Loading a city* 30⟩≡ (29)

```
city* loadCity(char* file)
{
    printf("Loading city: %s\n", file);
    FILE* f = fopen(file, "rb");
    if (!f) {perror("loadCity() error:"); return NULL;}
    // memory allocation
    #define BUF_SIZE 4096
    city* c = (city*)calloc(1, sizeof(city));
    char* s = (char*)malloc(BUF_SIZE);
    // useful macro to get first noncomment/nonempty line
    // (scope local to this function)
    // inlines excessively, but eh, I'm in a hurry :)
    #define FETCH_LINE {do {                                                    \
        if (!fgets(s, BUF_SIZE, f)) {                                           \
            if (feof(f))                                                        \
                fprintf(stderr, "loadCity() error: file ended too soon!\n");    \
            if (ferror(f))                                                      \
                perror("loadCity() error reading file:");                       \
            return NULL;                                                        \
        }                                                                       \
    } while (s[0]=='\n' || s[0]=='#');} // find nonblank noncomment lines
    // get name
    FETCH_LINE;
    c->name = (char*)malloc(strlen(s)+1);
    strcpy(c->name, s);
    // get number of models in city
    FETCH_LINE;
    if (1!=sscanf(s, " %d ", &c->num_buildings)) {
        fprintf(stderr, "loadCity() error: could not read number of buildings.\n");
        return NULL;
    }
    // allocate memory for buildings
    c->buildings = (building*)calloc(c->num_buildings, sizeof(building));
    // load models!
    for (list_index i=0; i<c->num_buildings; i++) {
        FETCH_LINE;
        building* b = &c->buildings[i];
        b->path = (char*)malloc(512);
        int ret = sscanf(s, "%s %f %f %f %f %f %f %f %f %f %f ", b->path,
                                        &b->sc.x, &b->sc.y, &b->sc.z,
                                        &b->rt.x, &b->rt.y, &b->rt.z,
                                        &b->tx.x, &b->tx.y, &b->tx.z);
```

```
            if (ret!=10) {
                fprintf(stderr, "loadCity() error while parsing building #%d, "
                                "only %d parameters read! <%s>\n", i+1, ret, s);
                return NULL;
            }
            b->model = loadModel(c->buildings[i].path);
            if (!b->model) {
                fprintf(stderr, "loadCity() exiting due to model load failure.");
                return NULL;
            }
        }
        free(s);
        return c;
    }
```

Uses `building` 28a, `city` 28a, `f` 32c, `list_index` 15, `model` 15, `printf` 19, and `ret` 19.

Freeing the city is as simple as ensuring all allocated memory gets cleaned.

31a ⟨*Freeing a city* 31a⟩≡ (29)

```
    void freeCity(city* c)
    {
        if (c->name) free(c->name);
        for (list_index i; i<c->num_buildings; i++) {
            freeModel(c->buildings[i].model);
            free(c->buildings[i].path);
        }
        if (c->buildings) free(c->buildings);
    }
```

Defines:
`freeCity`, used in chunks 28b and 32a.
Uses `city` 28a, `freeModel` 17, `list_index` 15, and `model` 15.

4.3 Initializing

The city gets loaded if a parameter gets passed to the executable. A non-null "city" pointer means the renderer should probably display a movie in that city.

31b ⟨*Main includes* 4a⟩+≡ (3) ◁13b

```
    #include "city.h"
```

Uses `city` 28a.

31c ⟨*Globals* 5a⟩+≡ (3) ◁13c 32b▷

```
    city* acity = NULL;
```

Uses `city` 28a.

31d ⟨*Global initialization* 4b⟩+≡ (3) ◁13d

```
    if (argc==2) {
        acity = loadCity(argv[1]);
    }
```

32a ⟨*Global cleanup* 13e⟩+≡ (3) ◁13e

```
if (acity) freeCity(acity);
```

Uses `freeCity` 31a.

4.4 Rendering

32b ⟨*Globals* 5a⟩+≡ (3) ◁31c

```
int frameCounter=0;
```

Defines:
`frameCounter`, used in chunk 32c.

32c ⟨*Render city* 32c⟩≡ (12)

```
glEnable(GL_TEXTURE_2D);
// render them buildings
for (list_index i=0; i<acity->num_buildings; i++) {
    building* b = &acity->buildings[i];
    glPushMatrix();
        glTranslatef(b->tx.x, b->tx.y, b->tx.z);
        glScalef(b->sc.x, b->sc.y, b->sc.z);
        glRotatef(b->rt.z, 0, 0, 1);
        glRotatef(b->rt.y, 0, 1, 0);
        glRotatef(b->rt.x, 1, 0, 0);
        renderModel(b->model);
    glPopMatrix();

}
char* buff = (char*)malloc(640*480*3);
char fname[256]; fname[0]=0; sprintf(fname, "out/%d.ppm", frameCounter);
FILE* f = fopen(fname, "wb");
glReadPixels(0,0,640,480,GL_RGB,GL_UNSIGNED_BYTE,(void*)buff);
fprintf(f, "P6\n640\n480\n255\n");
fwrite(buff, 1, 640*480*3, f);
fclose(f);

frameCounter++;
```

Defines:
`buff`, never used.
`f`, used in chunks 9a, 10, 17, 19–22, 26, and 30.
`fname`, never used.
Uses `building` 28a, `frameCounter` 32b, `list_index` 15, `model` 15, and `renderModel` 24.

5 Utility

5.1 Vectors

2D and 3D vectors are useful things, but not in the C stdlib. I use them to neatly represent the model data structure, and in the camera controls, to name two places.

33 ⟨*vec.h* 33⟩≡

```
⟨Copyright 34b⟩
#ifndef _VEC_H_
#define _VEC_H_

typedef struct vec3 {float x,y,z;} vec3;
typedef struct vec2 {float x,y;} vec2;

float dot(vec3 a, vec3 b);
vec3 cross(vec3 a, vec3 b);
void normalize(vec3* v);

#endif
```

Defines:
_VEC_H_, never used.
Uses dot 34a and normalize 34a.

The struct initialization syntax used in cross() is worth noting. It has a major advantage over other methods, in that the names can be out of order. A disadvantage is that you can skip fields, and the compiler will silently set them to a default. (Probably zero.)

34a ⟨*vec.c* 34a⟩≡

```
⟨Copyright 34b⟩
#include "vec.h"
#include <math.h>

float dot(vec3 a, vec3 b) {
    return a.x*b.x + a.y*b.y + a.z*b.z;
};

vec3 cross(vec3 a, vec3 b) {
    // rarely used, but kinda cool syntax:
    return (vec3){. x = a.y*b.z - a.z*b.y
                  ,. y = a.z*b.x - a.x*b.z
                  ,. z = a.x*b.y - a.y*b.x
                  };
};

void normalize(vec3* v) {
    float scale = sqrt(v->z*v->z + v->x*v->x + v->y*v->y);
    v->x/=scale;
    v->y/=scale;
    v->z/=scale;
}
```

Defines:
`dot`, used in chunks 10 and 33.
`normalize`, used in chunks 10 and 33.

5.2 Copyright

Including a header with copyright information, credits, and a brief list of recent changes is good practice. In this case, I'm also required to do so by the assignment.

By using literate programming and putting this header in a separate chunk, much copy and pasting is eliminated!

34b ⟨*Copyright* 34b⟩≡ (3 15–17 23 24 26 28a 33 34a)

```
/**
 *  AUTHOR: Alex Kropivny
 *  Copyright information, TODO
 *
 *  Please see accompanying PDF/HTML file for full comments and documentation!
 */
```

Uses `AUTHOR` 35.

The only other place where my name appears is the definition in main, for the function that draw it on the screen.

35 ⟨*Name definition* 35⟩≡ (12)

```
#define AUTHOR "Alex Kropivny"
```

Defines:
AUTHOR, used in chunks 12 and 34b.

5.3 Test Functions

The following code was provided by the course instructor.

36 ⟨*Test render functions* 36⟩≡ (12)

```
// The following code (Cube() and Axes()) is not mine. It was provided by the
// the instructor for the course.
void Cube()
{
    glBegin(GL_QUADS);
        glColor3f(0.0f,1.0f,0.0f);          // Set The Color To Green
        glVertex3f( 1.0f, 1.0f,-1.0f);          // Top Right Of The Quad (Top)
        glVertex3f(-1.0f, 1.0f,-1.0f);          // Top Left Of The Quad (Top)
        glVertex3f(-1.0f, 1.0f, 1.0f);          // Bottom Left Of The Quad (Top)
        glVertex3f( 1.0f, 1.0f, 1.0f);          // Bottom Right Of The Quad (Top)
        glColor3f(1.0f,0.5f,0.0f);          // Set The Color To Orange
        glVertex3f( 1.0f,-1.0f, 1.0f);          // Top Right Of The Quad (Bottom)
        glVertex3f(-1.0f,-1.0f, 1.0f);          // Top Left Of The Quad (Bottom)
        glVertex3f(-1.0f,-1.0f,-1.0f);          // Bottom Left Of The Quad (Bottom)
        glVertex3f( 1.0f,-1.0f,-1.0f);          // Bottom Right Of The Quad (Bottom)
        glColor3f(1.0f,0.0f,0.0f);          // Set The Color To Red
        glVertex3f( 1.0f, 1.0f, 1.0f);          // Top Right Of The Quad (Front)
        glVertex3f(-1.0f, 1.0f, 1.0f);          // Top Left Of The Quad (Front)
        glVertex3f(-1.0f,-1.0f, 1.0f);          // Bottom Left Of The Quad (Front)
        glVertex3f( 1.0f,-1.0f, 1.0f);          // Bottom Right Of The Quad (Front)
        glColor3f(1.0f,1.0f,0.0f);          // Set The Color To Yellow
        glVertex3f( 1.0f,-1.0f,-1.0f);          // Bottom Left Of The Quad (Back)
        glVertex3f(-1.0f,-1.0f,-1.0f);          // Bottom Right Of The Quad (Back)
        glVertex3f(-1.0f, 1.0f,-1.0f);          // Top Right Of The Quad (Back)
        glVertex3f( 1.0f, 1.0f,-1.0f);          // Top Left Of The Quad (Back)
        glColor3f(0.0f,0.0f,1.0f);          // Set The Color To Blue
        glVertex3f(-1.0f, 1.0f, 1.0f);          // Top Right Of The Quad (Left)
        glVertex3f(-1.0f, 1.0f,-1.0f);          // Top Left Of The Quad (Left)
        glVertex3f(-1.0f,-1.0f,-1.0f);          // Bottom Left Of The Quad (Left)
        glVertex3f(-1.0f,-1.0f, 1.0f);          // Bottom Right Of The Quad (Left)
        glColor3f(1.0f,0.0f,1.0f);          // Set The Color To Violet
        glVertex3f( 1.0f, 1.0f,-1.0f);          // Top Right Of The Quad (Right)
        glVertex3f( 1.0f, 1.0f, 1.0f);          // Top Left Of The Quad (Right)
        glVertex3f( 1.0f,-1.0f, 1.0f);          // Bottom Left Of The Quad (Right)
        glVertex3f( 1.0f,-1.0f,-1.0f);          // Bottom Right Of The Quad (Right)
    glEnd();                        // Done Drawing The Quad
}

void Axes()
{
    glScalef(10,10,10);
    glBegin(GL_LINES); {
        // Axis
        glColor3f(0.0f, 1.0f, 1.0f);
        glVertex3f(0.0f, 0.0f, 0.0f);
        glVertex3f(1.0f, 0.0f, 0.0f);
```

```
            // Axis
            glColor3f(0.0f, 1.0f, 0.0f);
            glVertex3f(0.0f, 0.0f, 0.0f);
            glVertex3f(0.0f, 1.0f, 0.0f);

            // Axis
            glColor3f(0.0f, 0.0f, 1.0f);
            glVertex3f(0.0f, 0.0f, 0.0f);
            glVertex3f(0.0f, 0.0f, 1.0f);
        } glEnd();
        glScalef(0.1,0.1,0.1);
    }
```

Defines:
  `Axes`, used in chunk 12.
  `Cube`, used in chunk 12.

6 Index