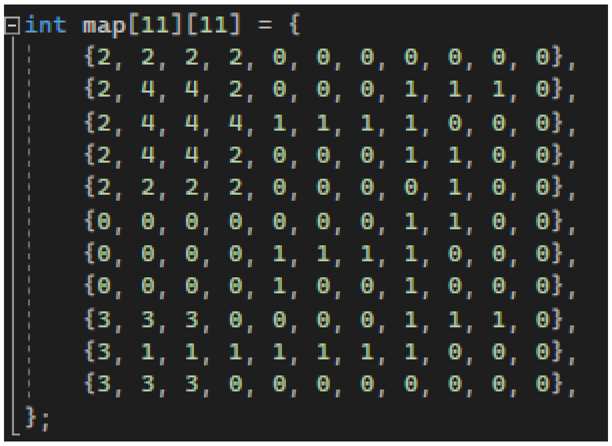
## Lösungsidee:

### Aufgabe1:

Den Raum und das Labyrinth (Also die Map an sich) Habe ich mithilfe eines 2Dimensionalen Arrays gerendert



```
int map[11][11] = {
    {2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0},
    {2, 4, 4, 2, 0, 0, 0, 1, 1, 1, 0},
    {2, 4, 4, 4, 1, 1, 1, 1, 0, 0, 0},
    {2, 4, 4, 2, 0, 0, 0, 1, 1, 0, 0},
    {2, 2, 2, 2, 0, 0, 0, 0, 1, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0},
    {0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0},
    {0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0},
    {3, 3, 3, 0, 0, 0, 0, 1, 1, 1, 0},
    {3, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0},
    {3, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0},
};
```

2 = Raum-Wände

4 = Boden und Decke

1 = Wege ( Also keine Wände)

0 = Labyrinth Wände

1.) Ähnlich wie Task 4 aus CGLab2 implementiert nur Anstatt eine Solid Sphere habe ich ein SolidTeapot und ein SolidIcosahedron ausgewhlt
2.) Nicht implementiert
3.) Schon in CGLab2 implementiert worden
4.) Habe ich mit Hilfe eines Bools (jumping) implementiert. Wenn man Leertaste drückt ändert sich der Bool auf True und die Y Position der camera bewegt sich sehr langsam nach oben bis die Höhe 4 erreicht wurde. Dann wird das Bool auf false gesetzt und die Y Position sehr langsam verringert bis auf sie 1 erreicht.

### Aufgabe2:

Das Labyrinth habe ich mittels eines 2Dim Arrays gerendert. Wichtig Hierbei ist der offset damit die Blöcke nicht ineinander gerendert werden.

1.) Es wird die position der Kamera auf der Map ausberechnet und geschaut ob es in mit einer Wand kollidiert.  Falls ja wird die Kamera auf Ursprüngliche Position gesetzt

### Aufgabe3:

Mit der moving_mouse Methode wird klargestellt dass sich der lookat Vektor der Kamera sich in richtung der Maus bewegt. Das habe ich mit Hilfe von glutWrapPointer implementiert das Meine Maus auf der Mitte des Displays locked und es so viel einfacher für mich macht das zu Implementieren. Ebenfalls wird mit Hilfe der lx und lz Variablen sichergestellt das z.B wenn man ‚w' drückt um sich nach vorne zu bewegen, dass sich dann die Kamera wirklich nach vorne (also in Richtung wo die Kamera gerade schaut) bewegt

1.) Nicht Implementiert
2.) Nicht implementiert

## Quellcode:

```
#include "GL/freeglut.h"
#define _USE_MATH_DEFINES
#include <cmath>
#include <iostream>
#include "math.h"
#include <chrono>
#include <thread>

using std::cout;
using std::endl;
using namespace std::this_thread; // sleep_for, sleep_until
using namespace std::chrono; // nanoseconds, system_clock, seconds

int windowid; // the identifier of the GLUT window
bool jumping = false;

GLfloat matrot[][4] = {           // a rotation matrix
  { 0.707f, 0.707f, 0.0f, 0.0f}, // it performs a rotation around z
  {-0.707f, 0.707f, 0.0f, 0.0f}, // in 45 degrees
  { 0.0f,   0.0f,   1.0f, 0.0f},
  { 0.0f,   0.0f,   0.0f, 1.0f}
};


float hor_angle_glob = 0;
float ver_angle_glob = 0;
float wall_size = 7.0f;

GLfloat mattrans[][4] = {         // a translation matrix
  { 1.0f, 0.0f,  0.0f, 0.0f},     // it performs a translation along the
  { 0.0f, 1.0f,  0.0f, 0.0f},     // x-axis of 0.5 units and along
  { 0.0f, 0.0f,  1.0f, 0.0f},     // the z-axis of -1.5 units
  { 0.5f, 0.0f, -1.5f, 1.0f}
};
```

```
int map[11][11] = {
    {2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0},
    {2, 4, 4, 2, 0, 0, 0, 1, 1, 1, 0},
    {2, 4, 4, 4, 1, 1, 1, 1, 0, 0, 0},
    {2, 4, 4, 2, 0, 0, 0, 1, 1, 0, 0},
    {2, 2, 2, 2, 0, 0, 0, 0, 1, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0},
    {0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0},
    {0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0},
    {3, 3, 3, 0, 0, 0, 0, 1, 1, 1, 0},
    {3, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0},
    {3, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0},
};

// Navigation variables - required for TASK 5
GLfloat navX = 10.0f;
GLfloat navZ = 10.0f;
GLfloat navY = 1.0f;

// Angle for cube rotation - requi red for TASK 6
GLfloat angleCube = 1.0f;        //angle for cube1

// Camera motion variables - required for HOMEOWRK HELPER
GLdouble angle = 0.0f;          // angle of rotation for the camera direction
GLdouble lx = 0.0f, lz = -1.0f; // actual vector representing the camera's
                                // direction
GLdouble x = 0.0f, z = 5.0f;    // XZ position of the camera

/*
*/
//Taken from http://www.lighthouse3d.com/tutorials/glut-tutorial/keyboard-
example-moving-around-the-world/
void processSpecialKeys(int key, int xcoor, int ycoor) {
    float fraction = 0.1f;

    switch (key) {
    case GLUT_KEY_LEFT:
        angle -= 0.01f;
        lx = sin(angle);
        lz = -cos(angle);
        break;
    case GLUT_KEY_RIGHT:
        angle += 0.01f;
        lx = sin(angle);
        lz = -cos(angle);
        break;
    case GLUT_KEY_UP:
        x += lx * fraction;
        z += lz * fraction;
        break;
    case GLUT_KEY_DOWN:
        x -= lx * fraction;
        z -= lz * fraction;
        break;
    }
}

/* This is the keyboard function which is used to react on keyboard input.
   It has to be registered as a callback in glutKeyboardFunc. Once a key is
   pressed it will be invoked and the keycode as well as the current mouse
   coordinates relative to the window origin are passed.
   It acts on our FPS controls 'WASD' and the escape key. A simple output
   to the keypress is printed in the console in case of 'WASD'. In case of
   ESC the window is destroyed and the application is terminated. */
void keyboard(unsigned char key, int xcoor, int ycoor) {
    float oldX = navX;
    float oldZ = navZ;
    switch (key) {
    case 'a':
        navX += lz;
        navZ += -lx;
        break;
    case 'd':
        navX -= lz;
        navZ -= -lx;
        break;
    case 'w':
        navZ += lz;
        navX += lx;
        break;
    case 's':
        navZ -= lz;
        navX -= lx;
        break;
    case 32:
        jumping = true;
        break;
    case 27: // escape key
        glutDestroyWindow(windowid);
        exit(0);
        break;
    }
    int col_X = (int)(navX) / (int)(wall_size);
    int col_Z = (int)(navZ) / (int)(wall_size);
    if (map[col_Z][col_X] == 0 || map[col_Z][col_X] == 2
        || navX < 0
        || navZ < 0)
    {
        cout << "collision with: " << col_Z << "," << col_X << "=" <<
map[col_Z][col_X] << endl;
        navX = oldX;
        navZ = oldZ;
    }
    glutPostRedisplay();
}

/* This function should be called when the window is resized. It has to be
   registered as a callback in glutReshapeFunc. The function sets up the
   correct perspective projection. Don't worry about it we will not go into
   detail but we need it for correct perspective 3D rendering. */
void reshapeFunc(int xwidth, int yheight) {
    if (yheight == 0 || xwidth == 0) return;  // Nothing is visible, return

    glMatrixMode(GL_PROJECTION); // Set a new projection matrix
    glLoadIdentity();

    gluPerspective(40.0f, (GLdouble)xwidth / (GLdouble)yheight, 0.5f, 20.0f);
    glViewport(0, 0, xwidth, yheight);  // Use the whole window for rendering
}



/* This function will be used for composited objects and will be called from a
   display function. */
void drawObject(void) { // TASK 4:
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glPushMatrix(); //set where to start the current object transformations
    glColor3f(0.0f, 1.0f, 0.0f);    // change cube1 to green
    glutSolidCube(1.0f);            // cube
    glTranslatef(0.0f, 1.0f, 0.0f); // move cube1 to the top
    glutSolidSphere(0.5f, 20, 20);
    glPopMatrix();
}
```

```
/* This function will be used for composited objects and will be called from a
   display function. */
void drawTeaPot(void) { // TASK 4:
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glPushMatrix(); //set where to start the current object transformations
    glColor3f(0.0f, 1.0f, 0.0f);    // change cube1 to green
    glutSolidCube(1.0f);            // cube
    glTranslatef(0.0f, 1.0f, 0.0f); // move cube1 to the top
    glutSolidTeapot(0.5);
    glPopMatrix();
}

/* This function will be used for composited objects and will be called from a
   display function. */
void drawIcosahedron(void) { // TASK 4:
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glPushMatrix(); //set where to start the current object transformations
    glColor3f(0.0f, 1.0f, 0.0f);    // change cube1 to green
    glutSolidCube(1.0f);            // cube
    glTranslatef(0.0f, 1.5f, 0.0f); // move cube1 to the top
    glutSolidIcosahedron();
    glPopMatrix();
}

void render_objects() {
    glTranslatef(18.0f, -3.0f, 15.0f);
    glTranslatef(-1.0f, 0.0f, 0.0f);
    drawObject();
    glTranslatef(0.0f, 0.0f, 2.0f);
    drawTeaPot();
    glTranslatef(0.0f, 0.0f, 2.0f);
    drawIcosahedron();
}

void moving_mouse(int x, int y) {
    int vertical_center = glutGet(GLUT_WINDOW_HEIGHT) / 2;
    int horizontal_center = glutGet(GLUT_WINDOW_WIDTH) / 2;
    hor_angle_glob += (x - horizontal_center) * 0.001;
    float diff = -((float)y - (float)vertical_center) * 0.001;
    ver_angle_glob += diff;
    glutWarpPointer(800 / 2, 600 / 2);
    glutPostRedisplay();
}
```

```
void render_labyrinth() {
    int z = 0;
    for (int i = 0; i < 11; i++) {
        for (int j = 0; j < 11; j++) {
            if (map[i][j] == 0) {

                glPushMatrix();

                const float offset = wall_size / 2;
                glTranslatef((float)(j) * wall_size + offset, 0, (float)(i) *
wall_size + offset);

                glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
                glColor3d(0.3f, 0, 0);
                glutSolidCube(wall_size * 0.99);

                glPopMatrix();
            }
            if (map[i][j] == 2) {
                glPushMatrix();

                const float offset = wall_size / 2;
                glTranslatef(static_cast<float>(j) * wall_size + offset, 0,
static_cast<float>(i) * wall_size + offset);

                glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
                glColor3d(1.0f, 1.0f, 1.0f);
                glutSolidCube(wall_size * 0.99);

                glPopMatrix();
            }
            if (map[i][j] == 3) {
                glPushMatrix();

                const float offset = wall_size / 2;
                glTranslatef(static_cast<float>(j) * wall_size + offset, 0,
static_cast<float>(i) * wall_size + offset);

                glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
                glColor3d(1.0f, 1.0f, 0.0f);
                glutSolidCube(wall_size * 0.99);

                glPopMatrix();
            }
            if (map[i][j] == 4 || map[i][j] == 3)
            {
                glPushMatrix();

                const float offset = wall_size / 2;
                glTranslatef(static_cast<float>(j) * wall_size + offset, -9.0f,
static_cast<float>(i) * wall_size + offset);

                glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
                glColor3d(1.0f, 1.0f, 0.0f);
                glutSolidCube(10.0f);

                glTranslatef(0, 17.5f,0);
                glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
                glColor3d(1.0f, 1.0f, 0.0f);
                glutSolidCube(10.0f);

                glPopMatrix();
            }
        }
    }

}


void position_cam() {
    lx = sin(hor_angle_glob);
    lz = -cos(hor_angle_glob);
    gluLookAt(navX, navY, navZ,      // camera position
        navX + lx, navY + ver_angle_glob, navZ + lz,      // target position (at)
        0.0f, 1.0f, 0.0f);     // up vector
}

/* This function will replace the previous display function and will be used
   for rendering a cube and playing with transformations. */
void renderScene(void) {
    glMatrixMode(GL_MODELVIEW);
    // glClear(GL_DEPTH_BUFFER_BIT); // Helper to be used with drawObjectAlt
    glClear(GL_COLOR_BUFFER_BIT);
    glClear(GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glDepthMask(GL_TRUE);
    glDepthFunc(GL_LEQUAL);
    glDepthRange(0.0f, 1.0f);
    glClearDepth(1.0f);
    glLoadIdentity();
    position_cam();
    render_labyrinth();
    render_objects();
    glutSwapBuffers();
}


/* This function will registered as a callback with glutIdleFunc. Here it will
   be constantly called and perform updates to realise an animation. */
void idleFunc(void) {
    if (jumping)
    {
        if (navY > 4.0f) {
            jumping = false;
        }
        else
        {
            navY += 0.000005;
        }
    }
    else
    {
        if (navY > 1.0f)
        {
            navY -= 0.000005;
        }
    }
}

/* This is our main method which will perform the setup of our first OpenGL
   window. The command line parameters will be passed but not used in the
   context of our application. Callbacks have been registered already and
   are prepared for future use during the lab. */
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowPosition(500, 500); // initial position of the window
    glutInitWindowSize(800, 600);     // initial size of the window
    windowid = glutCreateWindow("Our Second OpenGL Window");
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

    glutReshapeFunc(reshapeFunc);
    glutPassiveMotionFunc(moving_mouse);
    glutKeyboardFunc(keyboard);
    glutDisplayFunc(renderScene);
    glutIdleFunc(idleFunc);

    glutMainLoop();
    return 0;
}
```