

CSC630 – Computer Graphics

Term Project

“The Crazy Labyrinth”

Barbara Schneider and Sebastian Maier

12/11/2005

Table of contents

Table of contents	2
1. Introduction	3
2. Functionality	3
Player movement:	3
Testing keys :	3
3. The Implementation Process	3
4. The Strong Point – Collision Detection	5
4.1 A real a priori algorithm.	6
4.2 A real a posteriori algorithm.	7
4.3 A priori and a posteriori combination	7
5. The resulting source code	9
5.1 game.cpp	9
5.2 loadWorld.cpp	17
5.3 loadWorld.h	21
5.4 rendering.cpp	25
5.5 rendering.h	29
6. Appendix	33
a) game.cpp	33
b) loadWorld.cpp	42
c) loadWorld.h	45
d) rendering.cpp	46
e) rendering.h	47
f) Screenshots	49
1. First floor, the entrance of the labyrinth	49
2. First floor, the key for the elevator	50
3. First floor, the elevator	51
4. Second floor, after using the elevator	52
5. After exit has been found	53

1. Introduction

The task was to create a game with user interaction in 3D. The user should be able to take different view points. The project should have one strong point which could be almost anything we like.

For our term project, we decided to build a labyrinth game in OpenGL. The goal for the player is to enter the labyrinth through the entrance and leave it through the exit. On his way, he can collect gold coins and a key. The labyrinth consists of two floors, where the entrance is on the first floor and the exit on the second floor. The player can get from the first floor to the second floor by using an elevator which he can only use when he has already found the key and which beams him up to the second floor.

2. Functionality

<i>Player movement:</i>		<i>Testing keys :</i>	
Key	Function	Key	Function
w	<i>move forwards</i>	y	move up
s	<i>move backwards</i>	z	move down
a	<i>move left</i>	n	position player in front of the key
d	<i>move right</i>	m	position player in front of the elevator
q	<i>turn left</i>		
e	<i>turn right</i>		

3. The Implementation Process

First we decided to build the term project on top of assignment 3, as this already had a working OpenGL framework, the possibility to read in world data from a file and some functions for texture. We changed these functions during the development, but the basics are still there.

Our first two big goals were to create the labyrinth world and to implement the movement of the player. The first one could be achieved by simply changing the world file "world.txt", and although it was a lot of work it went pretty straight forward. Although it sounds easy, the second goal proved much more difficult. The first idea was to let the camera stay at the origin all the time with the word translating and

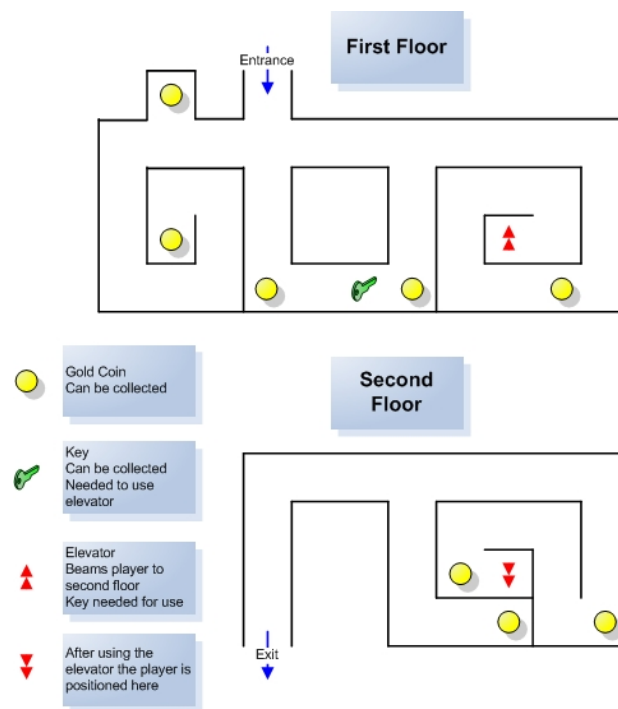
rotating around it. The implementation of forward, backward and sideward movement like this was easy, and also the implementation of a left and right turn was not difficult. But the tricky part was that as soon as we wanted to combine them, the player either still walked in the old directions when turned instead of in the direction he was facing, or he rotated around the origin instead of around himself which didn't correctly implement the turning. We then tried a lot of different possibilities: we tried to move the camera and rotate the world, rotate the camera and move the world, or implement the complete player's movement as camera movement with a fixed world. Neither of these changed anything to our problem. It was not until the very end of the project that we finally got it working. Now, like planned in the beginning, the camera always stays at the origin and the world turns and rotates around it. But the translation is now not only increasing/decreasing the x, y or z position, but the "movement vector" is rotated around the y-axis. It still took a lot of thoughts and debugging until it finally worked correctly.

The next thing was to find and load nice textures for the labyrinth. The code is done so far that all the walls have the same texture, and each floor / ceiling can have a different one. This separation can be changed easily in the file and by adding a few variables. We decided it looked nicer with the same texture for both floors, which is a darker version of the texture for the walls, and a sky with stars for the ceiling of the second floor. Like this, the labyrinth doesn't look to colorful, but you can well see the difference between the first and the second floor.

Now it was time to create items that the player can collect on his way through the labyrinth. We thought it would be nice to have a file from which all these items can be controlled, so they are read in from "events.txt". There are two different kinds of items: Gold coins (item type 1) and keys (item type 2). The position and the type of each item is read in from the file and rendered in a loop. To render the coins, we used the functions `gluDisk` and `gluCylinder`, where `gluCylinder` creates the cylinder which is closed on both sides by disks from `gluDisk`. The keys are a cylinder (created in the same way as the coins) and three cubes. We thought it would be more interesting if the items rotate all the time as you then can better see what they are and as like this they attract the attention of the player, so all items rotate all the time.

Having nice coins and keys, we also wanted a sign to indicate the elevator and decided to create a rotating arrow which points up. The arrow is hard-coded as it can not be considered as an item (it cannot be collected and it exists only once) and consists of three cubes.

The plan of the finished labyrinth now looks like this:



4. The Strong Point – Collision Detection

Like we already said in the previous direction, we used collision detection for different purposes. The first thing, of course, is that a player cannot walk through walls and therefore the collision of the player with a wall has to be detected and forbidden. The second thing is the detection of events: A message is put on the screen as soon as the player leaves the labyrinth through the entrance or the exit, and it must be detected if he finds a coin or a key or if he steps into the elevator. The cases are implemented as two different things, although the technique used is similar.

We will start with a survey of possibilities to implement collision detection. First you can divide collision detection algorithms in a priori and a posteriori algorithms. A priori algorithms check to see if a collision will happen before the actual collision, and a posteriori algorithms first change the position of all objects and then check if a

collision has happened. Both methods have advantages: a priori algorithms have to know what will happen in the future, whereas a posteriori algorithms just work on the current set of the objects. Still, it might be difficult to react on a collision after it has happened, as it may be necessary to undo the movement and this is not always simple.

We will begin with our first problem: the player cannot be allowed to walk through the walls. What we know about this problem is that:

- walls are represented as triangles, or more exactly, as the vertices of the triangles were two following triangles create one wall.
- they have a height and a width, but no depth.

The probably easiest way would be to use the vertices of the triangles to check collision with every triangle. But this means a waste of computational time: We can always combine two triangles to a rectangle, and each rectangle can be computed only by two opposite vertices. As each triangle contains two opposite vertices, we only need one triangle to know the position of one wall. This means that we can skip every second triangle.

From this, there are three obvious possibilities to handle the problem.

4.1 A real a priori algorithm.

Each time the player moves, we compute the distances to the next walls when moving forward, backward, left and right. We then only allow him to go this far. This computation could use the previous distances when moving, but on a turn we would have to recompute all values. We could use an algorithm to detect the intersection of a line and a plane and use it on all walls to find the nearest intersecting point. This would give us a running time of $O(w)$ each time the player moves or turns, as all the walls have to be checked 4 times and all the other computations are constant in their running time ($T(w) = 4 \cdot c_1 \cdot w/2 + c_2$ for some constants c_1 and c_2). We would achieve a physically correct answer, but the computations are pretty difficult to implement.

4.2 A real a posteriori algorithm.

We could also let the player move wherever he wants, and before rendering, we could check if he collides with a wall. This only takes a running time of $O(w)$ each time the player moves but not on turns: $T(w) = c*w/2$. But we have to note here that a collision takes two 3D-objects, as the probability for the player to exactly hit the 2D wall is close to 0. This means that we have to create virtual 3D walls, creating a space around them which also cannot be hit. This space has to be deep enough such that the player cannot simply step over it. For example, if he moves a distance of 4 units each step, but the wall only has a depth of 2, it can happen that the player first is on the one side and after his movement on the other side of the wall, which is not to be allowed. We also have to think about what happens after we have detected a collision: We would have to move him out of the wall in the direction where he came from. Desirable would be that he is moved to the same position he had before.

4.3 A priori and a posteriori combination

These conclusions lead to the third possibility and the one we decided to use. For our game, it is not important to have a physically exact solution, a wall with a certain depth works just fine. The walls have a depth of two times the step size of the player guaranteeing he will definitely be caught in a wall when trying to pass it. We solved the problem of undoing a step like this: Each time the player moves, his potential position is calculated. We then check this position on collisions with walls. Only if no collision is detected, the player is actually moved to his new position.

Now to the second problem: Detection of events. Events can again be splitted into the items the player can collect, further leaving the labyrinth through the entrance or through the exit and the item event. The detection works similar for all of them.

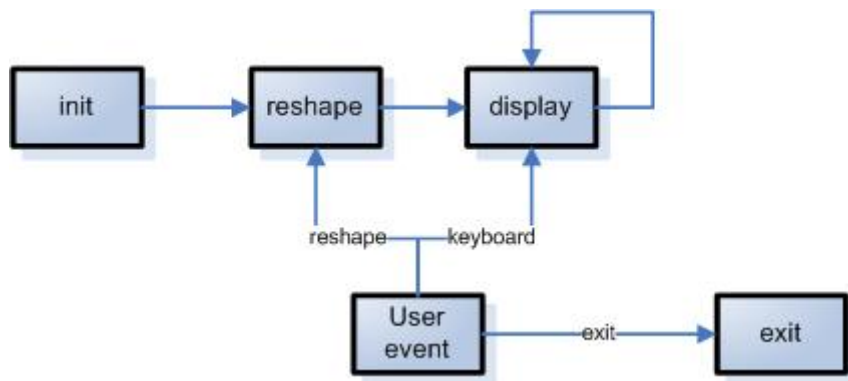
First we can notice that we do not have the problem here with the a posteriori approach that we had with the walls: We do not have to undo anything on a collision. This is why we used a real a posteriori algorithm here: The player is moved to his new position, and before rendering a function takes care of event detection. Again, we do not have to build in physical accuracy: a cube around the items work well, we do not have to implement the detection of the player actually hitting a part of a key,

for example, which would be much more expensive. Also, as the player is represented only as the position of his eyes, we have to keep in mind that his actual body is much bigger and therefore an exact collision detection would imply the modeling of his body. The check can be done similar to problem one: It is checked if the position of the player is inside one of the event cubes or not. If he is, the corresponding action is provoked. This gives us a running time of $O(e)$, where e is the number of events in the labyrinth. The difference between the items and the other events is that the cube around the items is calculated dynamically around the position of the item, whereas the position of the elevator, entrance and exit cannot be changed and is hardcoded.

In general, collision detection can be more complicated than in this application. Here, only the player is moving through the labyrinth, which means that we only have to detect collision of the player with objects of the game. If there are more moving objects (for example the balls in a billiard game), the collision of all these objects have to be checked. The more of these objects we have in the world, the more effort has to be put in the algorithms to gain an acceptable running time. One of the techniques which can be used here is temporal coherence, which takes into account that physical objects most of the time do not change their position by a large distance, to decrease the number of pairs which have to be checked for collision.

5. The resulting source code

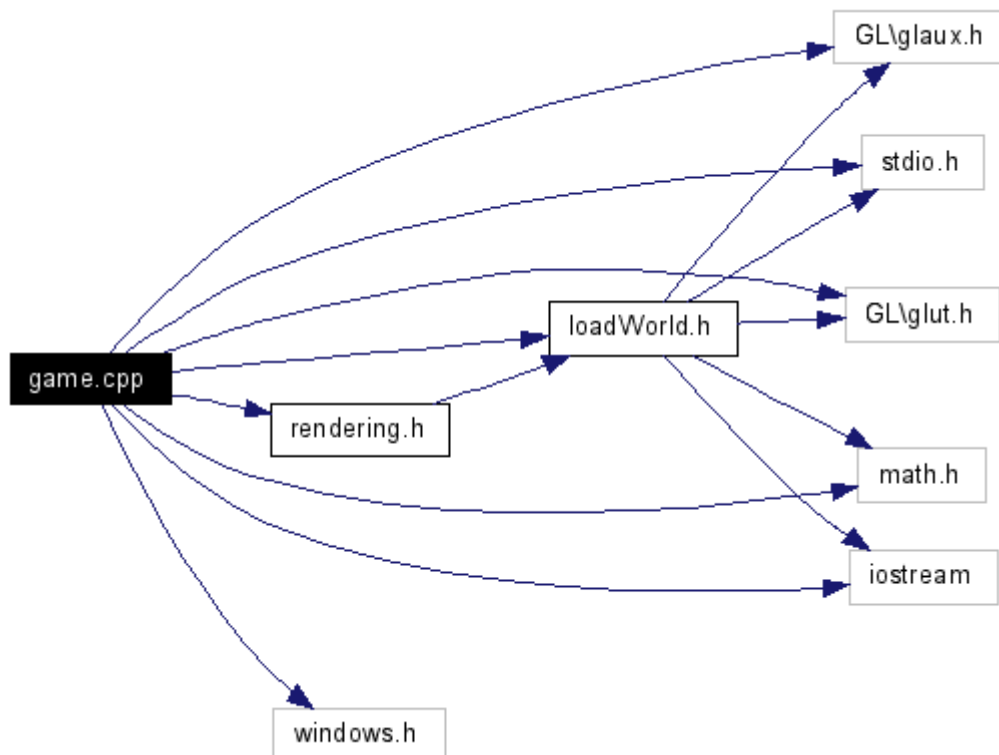
The flow of the program is still the same as it was for Assignment 3:



5.1 game.cpp

```
#include "rendering.h"  
#include "loadWorld.h"  
#include <GL\glaux.h>  
#include <stdio.h>  
#include <GL\glut.h>  
#include <math.h>  
#include <iostream>  
#include <windows.h>
```

Include dependency graph for game.cpp:



Namespaces

namespace	std
-----------	---------------------

Functions

void	display ()
void	init (void)
void	checkPosition ()
void	drawItems ()
void	reshape (int w, int h)
bool	checkCollision (double newX, double newY, double newZ)
void	keyboard (unsigned char key, int x, int y)
int	main (int argc, char **argv)

Variables

const double	movement = 0.2
int	noOfEvents
item *	events
bool	leftThroughEntrance = false
int	money = 0
int	xscale = 4
int	yscale = 6
int	zscale = 4
double	xpos = -0.5* xscale
double	ypos = -0.5* yscale
double	zpos = 0
double	yrot = 0
int	angle = 0
bool	fCheckPosition = true
const double	PI = 3.14159265

Function Documentation

```
bool checkCollision( double newX,  
                   double newY,  
                   double newZ  
                   )
```

The function `checkCollision(..)` is called each time the position of the player changes to check if the new position collides with a wall. It returns true if no collision is detected and false otherwise.

Definition at line [347](#) of file [game.cpp](#).

References [getSector1\(\)](#), [movement](#), [tagSECTOR::numtriangles](#), [tagSECTOR::numTrianglesFloorCeiling](#), [sector1](#), [tagSECTOR::triangle](#), [tagTRIANGLE::vertex](#), [tagVERTEX::x](#), [xscale](#), [tagVERTEX::y](#), [yscale](#), [tagVERTEX::z](#), and [zscale](#).

Referenced by [keyboard\(\)](#).

Here is the call graph for this function:



```
void checkPosition( )
```

Before a new position of the player is activated, the position is checked in [checkPosition\(\)](#). Here it is checked if the player left the labyrinth through the entrance or through the exit, if he collects an item or if he enters the elevator. The reaction is implemented accordingly.

Definition at line [164](#) of file [game.cpp](#).

References [money](#), [zpos](#), and [zscale](#).

Referenced by [display\(\)](#).

```
void display( void )
```

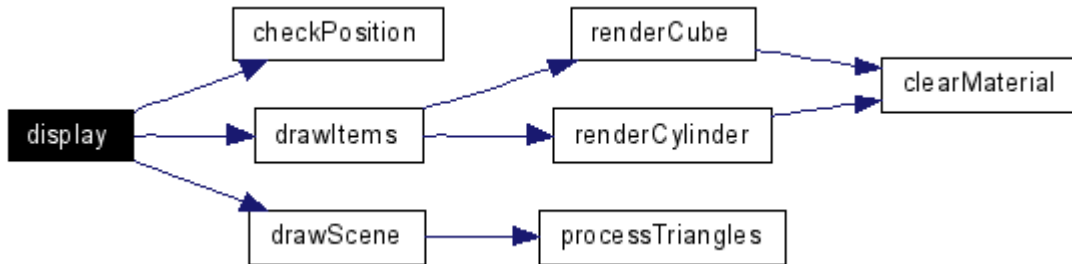
This function is called continuously during the runtime of the program. It calls all the functions necessary to draw the world and takes care of their position in the space: After checking the position with [checkPosition\(\)](#) [drawScene\(\)](#) and [drawItems\(\)](#) are called.

Definition at line [306](#) of file [game.cpp](#).

References [checkPosition\(\)](#), [drawItems\(\)](#), [drawScene\(\)](#), [fCheckPosition](#), [xpos](#), [xscale](#), [ypos](#), [yrot](#), [yscale](#), [zpos](#), and [zscale](#).

Referenced by [main\(\)](#).

Here is the call graph for this function:



void drawItems()

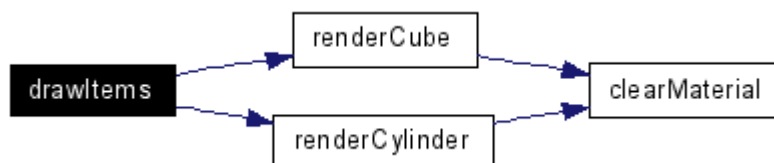
This function is used to draw the items in the labyrinth: The coins, the key and the up-arrow for the elevator.

Definition at line [236](#) of file [game.cpp](#).

References [angle](#), [item::p](#), [renderCube\(\)](#), [renderCylinder\(\)](#), [item::type](#), [xscale](#), [position::y](#), [yscale](#), [position::z](#), and [zscale](#).

Referenced by [display\(\)](#).

Here is the call graph for this function:



void init(void)

In the init-method, two light sources are set, the events are read from the file and saved in the array events, and the function [SetupWorld\(\)](#) is called to initialize the walls and floors of the labyrinth.

Definition at line [102](#) of file [game.cpp](#).

References [item::found](#), [noOfEvents](#), [item::p](#), [readstr\(\)](#), [item::type](#), [position::x](#), [position::y](#), and [position::z](#).

Referenced by [main\(\)](#).

Here is the call graph for this function:



```

void keyboard( unsigned char key,
               int          x,
               int          y
             )
  
```

This function implements the user interaction via keyboard. The keys 'a', 'w', 's' and 'd' represent the movement of the player on his x-z-plane and 'q' and 'e' turn the player left and right. Some letters are only implemented for testing reasons: the letters 'n' and 'm' move the player directly to the key and the elevator and 'x' and 'y' implement movement up and down. The function [checkCollision\(..\)](#) is

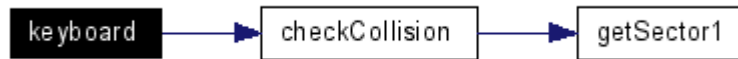
called to check if a movement lets the player hit a wall.

Definition at line [388](#) of file [game.cpp](#).

References [checkCollision\(\)](#), [movement](#), [PI](#), [xpos](#), [xscale](#), [ypos](#), [yrot](#), [zpos](#), and [zscale](#).

Referenced by [main\(\)](#).

Here is the call graph for this function:



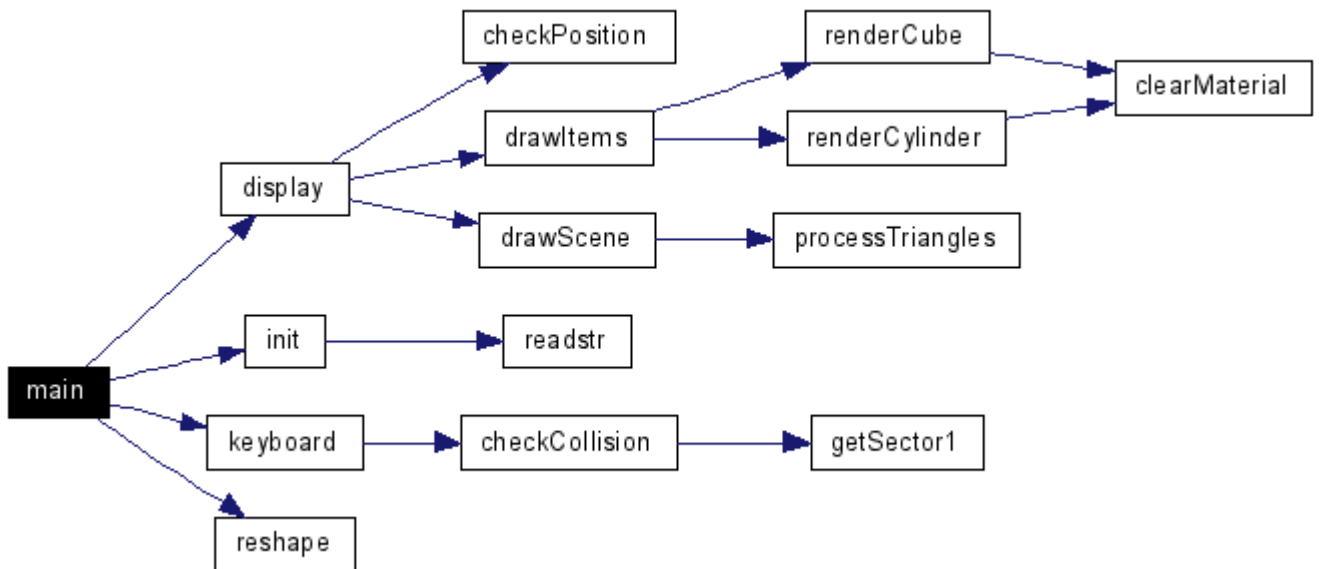
```
int main( int  argc,  
          char ** argv  
          )
```

The typical main function for an Open-GL application. The GLUT-library is initialized, the display mode is set to double buffered and rgb, the window size and position is set and the window created, the function [init\(\)](#) is called to initialize lighting and the labyrinth, the main loop function is set to [display\(\)](#) and the GLUT event processing loop is entered.

Definition at line [482](#) of file [game.cpp](#).

References [display\(\)](#), [init\(\)](#), [keyboard\(\)](#), and [reshape\(\)](#).

Here is the call graph for this function:



```
void reshape( int w,  
             int h  
             )
```

The traditional reshape function for OpenGL. Sets the Viewport and the perspective and initializes the modelview stack.

Definition at line [334](#) of file [game.cpp](#).

Referenced by [main\(\)](#).

Variable Documentation

int [angle](#) = 0

Implements the angle of the rotation of the coins, the key and the elevator arrow.

Definition at line [89](#) of file [game.cpp](#).

Referenced by [drawItems\(\)](#).

item* [events](#)

Array storing the events.

Definition at line [58](#) of file [game.cpp](#).

bool [fCheckPosition](#) = true

flag which is set to true if the function [checkPosition\(\)](#) is to be called in [display\(\)](#).

Definition at line [92](#) of file [game.cpp](#).

Referenced by [display\(\)](#).

bool [leftThroughEntrance](#) = false

Stores if the player has already left the labyrinth through the entrance. Like this, the message that he left through the entrance is only displayed once.

Definition at line [62](#) of file [game.cpp](#).

int [money](#) = 0

Stores the amount of gold coins found since the beginning of the game.

Definition at line [65](#) of file [game.cpp](#).

Referenced by [checkPosition\(\)](#).

const double [movement](#) = 0.2

Sets the movement of the camera.

Definition at line [51](#) of file [game.cpp](#).

Referenced by [checkCollision\(\)](#), and [keyboard\(\)](#).

int [noOfEvents](#)

Events are Coins and keys. They are read from file. The number of events is also read from the file and saved in noOfEvents.

Definition at line [55](#) of file [game.cpp](#).

Referenced by [init\(\)](#).

```
const double PI = 3.14159265
```

The circle constant PI.

Definition at line [95](#) of file [game.cpp](#).

Referenced by [keyboard\(\)](#).

```
double xpos = -0.5*xscale
```

x-position of the player.

Definition at line [78](#) of file [game.cpp](#).

Referenced by [display\(\)](#), and [keyboard\(\)](#).

```
int xscale = 4
```

Scaling of the world in x-direction. Also used to calculate the position of the player and to react to events based on his position.

Definition at line [69](#) of file [game.cpp](#).

Referenced by [checkCollision\(\)](#), [display\(\)](#), [drawItems\(\)](#), and [keyboard\(\)](#).

```
double ypos = -0.5*yscale
```

y-position of the player.

Definition at line [80](#) of file [game.cpp](#).

Referenced by [display\(\)](#), and [keyboard\(\)](#).

```
double yrot = 0
```

Rotation of the player around y-axis. It implements the turning of the player. The value is between -360 and 0.

Definition at line [86](#) of file [game.cpp](#).

Referenced by [display\(\)](#), and [keyboard\(\)](#).

```
int yscale = 6
```

Scaling of the world in y-direction. Also used to calculate the position of the player and to react to events based on his position.

Definition at line [72](#) of file [game.cpp](#).

Referenced by [checkCollision\(\)](#), [display\(\)](#), and [drawItems\(\)](#).

double [zpos](#) = 0

z-position of the player.

Definition at line [82](#) of file [game.cpp](#).

Referenced by [checkPosition\(\)](#), [display\(\)](#), and [keyboard\(\)](#).

int [zscale](#) = 4

Scaling of the world in z-direction. Also used to calculate the position of the player and to react to events based on his position.

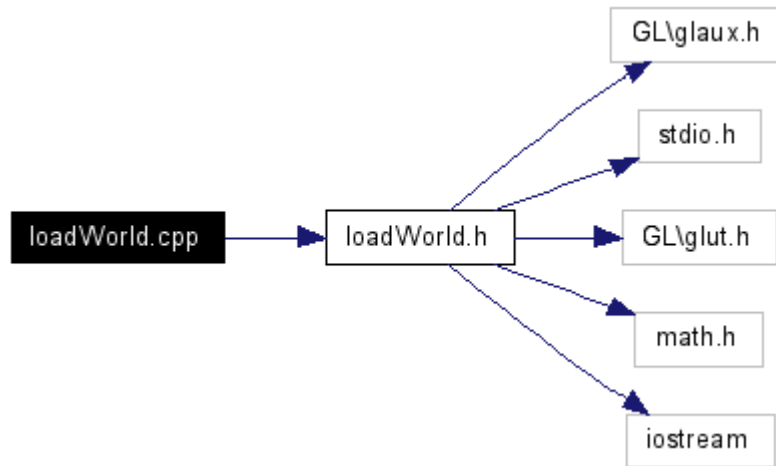
Definition at line [75](#) of file [game.cpp](#).

Referenced by [checkCollision\(\)](#), [checkPosition\(\)](#), [display\(\)](#), [drawItems\(\)](#), and [keyboard\(\)](#).

5.2 loadWorld.cpp

```
#include "loadWorld.h"
```

Include dependency graph for loadWorld.cpp:



Functions

SECTOR	getSector1 ()
void	readstr (FILE *f, char *string)
void	SetupWorld ()
AUX_RGBImageRec *	LoadBMP (char *Filename)
int	LoadGLTexture (char *textureName, GLuint *memory)
int	LoadGLTextures ()
void	processTriangles (int start, int end)
void	drawScene ()

Variables

GLuint	texture1 [3]
GLuint	texture2 [3]
GLuint	texture3 [3]
GLuint	texture4 [3]
SECTOR	sector1

Function Documentation

void drawScene()

This function coordinates the other drawing functions. It calls processTriangles(..) four times to draw the first floor, the second floor, the ceiling and the walls and binds the appropriate texture to them.

Definition at line [142](#) of file [loadWorld.cpp](#).

References [tagSECTOR::numtriangles](#), [tagSECTOR::numTrianglesFloorCeiling](#), [processTriangles\(\)](#), [texture1](#), [texture2](#), [texture3](#), and [texture4](#).

Referenced by [display\(\)](#).

Here is the call graph for this function:



SECTOR getSector1()

sector1 is the object of the struct [tagSECTOR](#) which actually stores the walls. It is defined in [loadWorld.cpp](#). The function [getSector1\(\)](#) makes it accessible from [game.cpp](#).

Definition at line [15](#) of file [loadWorld.cpp](#).

Referenced by [checkCollision\(\)](#).

AUX_RGBImageRec* LoadBMP(char * *Filename*)

Loads A Bitmap Image with the name Filename.

Definition at line [61](#) of file [loadWorld.cpp](#).

Referenced by [LoadGLTexture\(\)](#).

**int LoadGLTexture(char * *textureName*,
GLuint * *memory*
)**

Loads a Bitmap using LoadBMP with the name textureName and converts it to a texture. The texture then is stored in memory. Return value is the status.

Definition at line [82](#) of file [loadWorld.cpp](#).

References [LoadBMP\(\)](#).

Referenced by [LoadGLTextures\(\)](#).

Here is the call graph for this function:



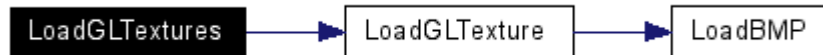
int LoadGLTextures()

This function calls LoadGLTexture(..) several times to load all the different textures into memory. Return value is the status.

Definition at line [116](#) of file [loadWorld.cpp](#).

References [LoadGLTexture\(\)](#), [texture1](#), [texture2](#), [texture3](#), and [texture4](#).

Here is the call graph for this function:



void processTriangles(int start, int end)

This function draws the triangles from sector1 from start to end and binds the vertices to their texture coordinates.

Definition at line [121](#) of file [loadWorld.cpp](#).

References [tagSECTOR::triangle](#), [tagVERTEX::u](#), [tagVERTEX::v](#), [tagTRIANGLE::vertex](#), [tagVERTEX::x](#), [tagVERTEX::y](#), and [tagVERTEX::z](#).

Referenced by [drawScene\(\)](#).

void readstr(FILE * f, char * string)

This function reads in a string from file f into string. It is used by [init\(\)](#) and [SetupWorld\(\)](#) to read in the data for the walls and the items.

Definition at line [20](#) of file [loadWorld.cpp](#).

Referenced by [init\(\)](#), and [SetupWorld\(\)](#).

void SetupWorld()

SetupWorld uses readstr(..) to reads in the wall data from world.txt into sector1.

Definition at line [29](#) of file [loadWorld.cpp](#).

References [tagSECTOR::numtriangles](#), [tagSECTOR::numTrianglesFloorCeiling](#), [readstr\(\)](#), [tagSECTOR::triangle](#), [tagVERTEX::u](#), [tagVERTEX::v](#), [tagTRIANGLE::vertex](#), [tagVERTEX::x](#), [tagVERTEX::y](#), and [tagVERTEX::z](#).

Here is the call graph for this function:



Variable Documentation

SECTOR [sector1](#)

sector1 stores the triangles of the walls, floors and ceiling as [tagSECTOR](#).

Definition at line [13](#) of file [loadWorld.cpp](#).

Referenced by [checkCollision\(\)](#).

GLuint [texture1\[3\]](#)

This array stores floor texture.

Definition at line [4](#) of file [loadWorld.cpp](#).

Referenced by [drawScene\(\)](#), and [LoadGLTextures\(\)](#).

GLuint [texture2\[3\]](#)

Stores ceiling texture.

Definition at line [6](#) of file [loadWorld.cpp](#).

Referenced by [drawScene\(\)](#), and [LoadGLTextures\(\)](#).

GLuint [texture3\[3\]](#)

Stores sky texture.

Definition at line [8](#) of file [loadWorld.cpp](#).

Referenced by [drawScene\(\)](#), and [LoadGLTextures\(\)](#).

GLuint [texture4\[3\]](#)

Stores wall texture.

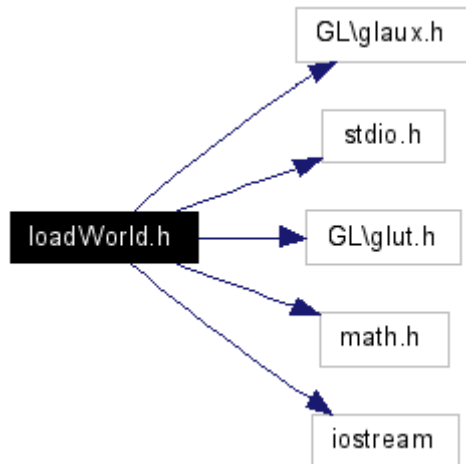
Definition at line [10](#) of file [loadWorld.cpp](#).

Referenced by [drawScene\(\)](#), and [LoadGLTextures\(\)](#).

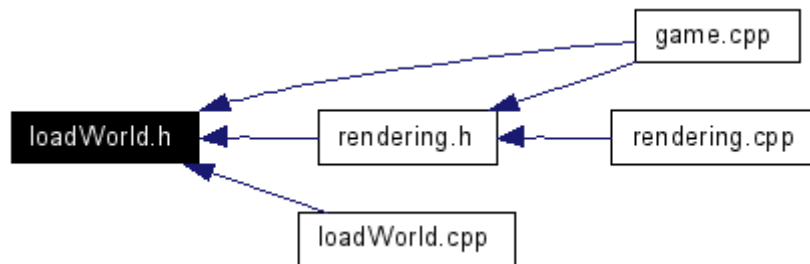
5.3 loadWorld.h

```
#include <GL\glaux.h>
#include <stdio.h>
#include <GL\glut.h>
#include <math.h>
#include <iostream>
```

Include dependency graph for loadWorld.h:



This graph shows which files directly or indirectly include this file:



Classes

struct	position
struct	item
struct	tagVERTEX
struct	tagTRIANGLE
struct	tagSECTOR

Typedefs

typedef	tagVERTEX	VERTEX
typedef	tagTRIANGLE	TRIANGLE
typedef	tagSECTOR	SECTOR

Functions

SECTOR	getSector1 ()
void	readstr (FILE *f, char *string)
void	SetupWorld ()
AUX_RGBImageRec *	LoadBMP (char *Filename)
int	LoadGLTexture (char *textureName, GLuint *memory)
int	LoadGLTextures ()
void	processTriangles (int start, int end)
void	drawScene ()

Typedef Documentation

typedef struct [tagSECTOR](#) SECTOR

This struct implements the data structure used to store the walls.

typedef struct [tagTRIANGLE](#) TRIANGLE

This is the data structure used to store a triangle.

typedef struct [tagVERTEX](#) VERTEX

[tagVERTEX](#) is the data structure used to store one vertex. It is used in [tagTRIANGLE](#) to store a triangle.

Function Documentation

void [drawScene](#)()

This function coordinates the other drawing functions. It calls [processTriangles\(..\)](#) four times to draw the first floor, the second floor, the ceiling and the walls and binds the appropriate texture to them.

Definition at line [142](#) of file [loadWorld.cpp](#).

References [tagSECTOR::numtriangles](#), [tagSECTOR::numTrianglesFloorCeiling](#), [processTriangles\(\)](#), [texture1](#), [texture2](#), [texture3](#), and [texture4](#).

Referenced by [display\(\)](#).

Here is the call graph for this function:



[SECTOR](#) [getSector1](#)()

`sector1` is the object of the struct [tagSECTOR](#) which actually stores the walls. It is defined in

[loadWorld.cpp](#). The function [getSector1\(\)](#) makes it accessible from [game.cpp](#).

Definition at line [15](#) of file [loadWorld.cpp](#).

Referenced by [checkCollision\(\)](#).

```
AUX_RGBImageRec* LoadBMP( char * Filename )
```

Loads A Bitmap Image with the name Filename.

Definition at line [61](#) of file [loadWorld.cpp](#).

Referenced by [LoadGLTexture\(\)](#).

```
int LoadGLTexture( char * textureName,  
                  GLuint * memory  
                  )
```

Loads a Bitmap using LoadBMP with the name textureName and converts it to a texture. The texture then is stored in memory. Return value is the status.

Definition at line [82](#) of file [loadWorld.cpp](#).

References [LoadBMP\(\)](#).

Referenced by [LoadGLTextures\(\)](#).

Here is the call graph for this function:



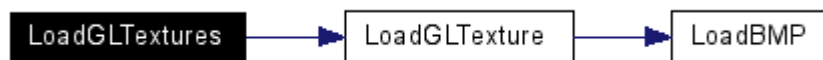
```
int LoadGLTextures( )
```

This function calls LoadGLTexture(..) several times to load all the different textures into memory. Return value is the status.

Definition at line [116](#) of file [loadWorld.cpp](#).

References [LoadGLTexture\(\)](#), [texture1](#), [texture2](#), [texture3](#), and [texture4](#).

Here is the call graph for this function:



```
void processTriangles( int start,  
                      int end  
                      )
```

This function draws the triangles from sector1 from start to end and binds the vertices to their texture coordinates.

Definition at line [121](#) of file [loadWorld.cpp](#).

References [tagSECTOR::triangle](#), [tagVERTEX::u](#), [tagVERTEX::v](#), [tagTRIANGLE::vertex](#), [tagVERTEX::x](#), [tagVERTEX::y](#), and [tagVERTEX::z](#).

Referenced by [drawScene\(\)](#).

```
void readstr( FILE * f,  
             char * string  
            )
```

This function reads in a string from file f into string. It is used by [init\(\)](#) and [SetupWorld\(\)](#) to read in the data for the walls and the items.

Definition at line [20](#) of file [loadWorld.cpp](#).

Referenced by [init\(\)](#), and [SetupWorld\(\)](#).

```
void SetupWorld( )
```

SetupWorld uses readstr(..) to reads in the wall data from world.txt into sector1.

Definition at line [29](#) of file [loadWorld.cpp](#).

References [tagSECTOR::numtriangles](#), [tagSECTOR::numTrianglesFloorCeiling](#), [readstr\(\)](#), [tagSECTOR::triangle](#), [tagVERTEX::u](#), [tagVERTEX::v](#), [tagTRIANGLE::vertex](#), [tagVERTEX::x](#), [tagVERTEX::y](#), and [tagVERTEX::z](#).

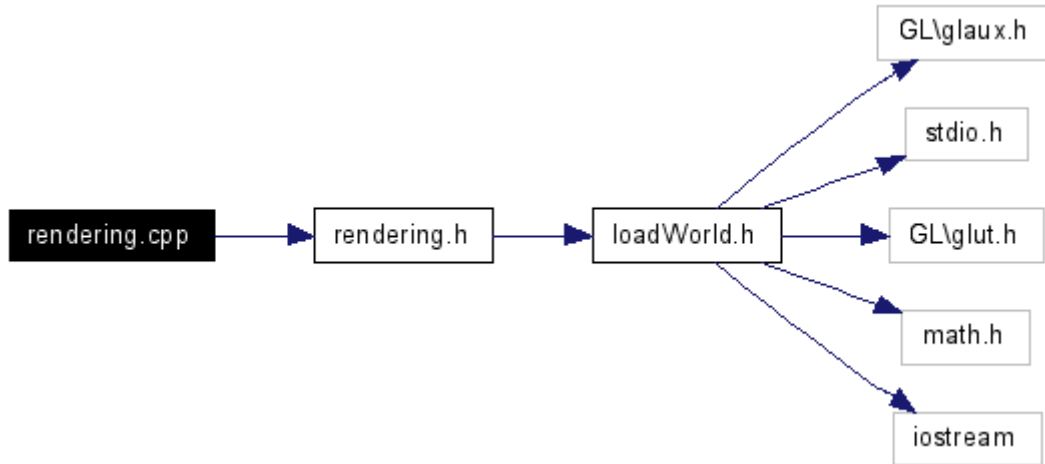
Here is the call graph for this function:



5.4 rendering.cpp

```
#include "rendering.h"
```

Include dependency graph for rendering.cpp:



Functions

void	clearMaterial ()
void	renderCube (GLdouble size, GLfloat ambr, GLfloat ambg, GLfloat ambb, GLfloat difr, GLfloat difg, GLfloat difb, GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
void	renderSphere (GLdouble radius, GLint slices, GLint stacks, GLfloat ambr, GLfloat ambg, GLfloat ambb, GLfloat difr, GLfloat difg, GLfloat difb, GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
void	renderCylinder (GLdouble base, GLdouble top, GLdouble height, GLint slices, GLint stacks, GLfloat ambr, GLfloat ambg, GLfloat ambb, GLfloat difr, GLfloat difg, GLfloat difb, GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)

Variables

GLfloat	ambient1 [] = {1.0, 1, 1.0, 1.0}
GLfloat	diffuse1 [] = {1, 1.0, 1.0, 1.0}
GLfloat	specular1 [] = {1.0, 1, 1.0, 1.0}
GLfloat	ambient2 [] = {1.0, 0.5, 1.0, 1.0}
GLfloat	diffuse2 [] = {1, 0.5, 1.0, 1.0}
GLfloat	specular2 [] = {1.0, 0.5, 1.0, 1.0}

Function Documentation

```
void clearMaterial( )
```

Sets the material to default values.

Definition at line [18](#) of file [rendering.cpp](#).

References [ambient1](#), [diffuse1](#), and [specular1](#).

Referenced by [renderCube\(\)](#), [renderCylinder\(\)](#), and [renderSphere\(\)](#).

```
void renderCube( GLdouble size,
                 GLfloat  ambr,
                 GLfloat  ambg,
                 GLfloat  ambb,
                 GLfloat  difr,
                 GLfloat  difg,
                 GLfloat  difb,
                 GLfloat  specr,
                 GLfloat  specg,
                 GLfloat  specb,
                 GLfloat  shine
               )
```

Sets the color values of the cube and calls `glutSolidCube(size)` to display it. The arguments are the length of the edges and the different color values in rgb mode. After displaying the cube, `clearMaterial` is called.

Definition at line [25](#) of file [rendering.cpp](#).

References [clearMaterial\(\)](#).

Referenced by [drawItems\(\)](#).

Here is the call graph for this function:



```
void renderCylinder( GLdouble base,
                    GLdouble top,
                    GLdouble height,
                    GLint    slices,
                    GLint    stacks,
                    GLfloat  ambr,
                    GLfloat  ambg,
                    GLfloat  ambb,
                    GLfloat  difr,
                    GLfloat  difg,
                    GLfloat  difb,
                    GLfloat  specr,
                    GLfloat  specg,
                    GLfloat  specb,
                    GLfloat  shine
                  )
```

Sets the color values of the cylinder and calls `gluCylinder(radius, base, top, height, slices, stacks)` as well as two times `gluDisk(radius, down(or up), 0, base, slices, stacks)` to display it. The arguments are the radius of the lower side and the upper side, the height, the number of subdivisions around the Z axis (slices), the number of subdivisions along the Z-axis (stacks) and the different color values in rgb mode. After displaying the cylinder, `clearMaterial` is called.

Definition at line [57](#) of file [rendering.cpp](#).

References [clearMaterial\(\)](#).

Referenced by [drawItems\(\)](#).

Here is the call graph for this function:



```
void renderSphere( GLdouble radius,
                  GLint    slices,
                  GLint    stacks,
                  GLfloat  ambr,
                  GLfloat  ambg,
                  GLfloat  ambb,
                  GLfloat  difr,
                  GLfloat  difg,
                  GLfloat  difb,
                  GLfloat  specr,
                  GLfloat  specg,
                  GLfloat  specb,
                  GLfloat  shine
                )
```

Sets the color values of the sphere and calls `glutSolidSphere(radius, slices, stacks)` to display it. The arguments are the radius, the number of subdivisions around the Z axis (slices), the number of subdivisions along the Z-axis (stacks) and the different color values in rgb mode. After displaying the sphere, `clearMaterial` is called.

Definition at line [41](#) of file [rendering.cpp](#).

References [clearMaterial\(\)](#).

Here is the call graph for this function:



Variable Documentation

GLfloat [ambient1](#)[] = {1.0, 1, 1.0, 1.0}

The array of GLfloats stores the ambient values of the first light source.

Definition at line [5](#) of file [rendering.cpp](#).

Referenced by [clearMaterial\(\)](#).

```
GLfloat ambient2[] = {1.0, 0.5, 1.0, 1.0}
```

The array of GLfloats stores the ambient values of the second light source.

Definition at line [12](#) of file [rendering.cpp](#).

```
GLfloat diffuse1[] = {1, 1.0, 1.0, 1.0}
```

The array of GLfloats stores the diffuse values of the first light source.

Definition at line [7](#) of file [rendering.cpp](#).

Referenced by [clearMaterial\(\)](#).

```
GLfloat diffuse2[] = {1, 0.5, 1.0, 1.0}
```

The array of GLfloats stores the diffuse values of the second light source.

Definition at line [14](#) of file [rendering.cpp](#).

```
GLfloat specular1[] = {1.0, 1, 1.0, 1.0}
```

The array of GLfloats stores the specular values of the first light source.

Definition at line [9](#) of file [rendering.cpp](#).

Referenced by [clearMaterial\(\)](#).

```
GLfloat specular2[] = {1.0, 0.5, 1.0, 1.0}
```

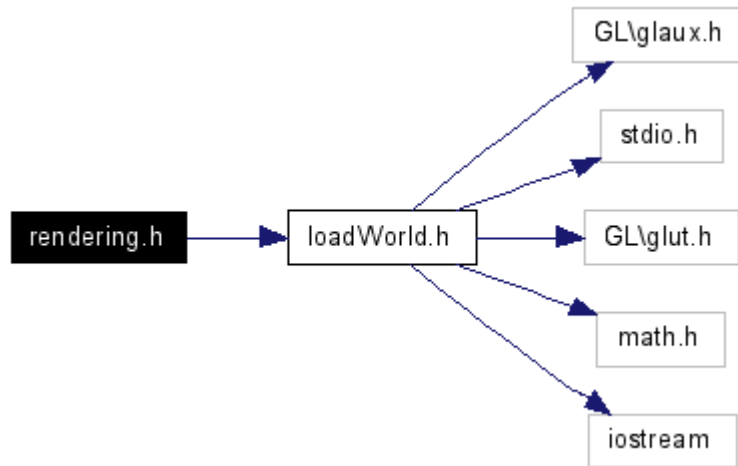
The array of GLfloats stores the specular values of the second light source.

Definition at line [16](#) of file [rendering.cpp](#).

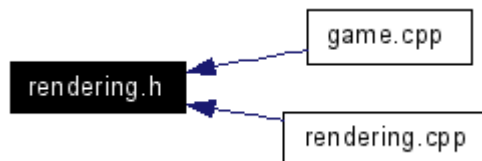
5.5 rendering.h

```
#include "loadWorld.h"
```

Include dependency graph for rendering.h:



This graph shows which files directly or indirectly include this file:



Functions

void	clearMaterial ()
void	renderCube (GLdouble size, GLfloat ambr, GLfloat ambg, GLfloat ambb, GLfloat difr, GLfloat difg, GLfloat difb, GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
void	renderSphere (GLdouble radius, GLint slices, GLint stacks, GLfloat ambr, GLfloat ambg, GLfloat ambb, GLfloat difr, GLfloat difg, GLfloat difb, GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
void	renderCylinder (GLdouble base, GLdouble top, GLdouble height, GLint slices, GLint stacks, GLfloat ambr, GLfloat ambg, GLfloat ambb, GLfloat difr, GLfloat difg, GLfloat difb, GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)

Variables

GLfloat	ambient1 []
GLfloat	diffuse1 []
GLfloat	specular1 []
GLfloat	ambient2 []
GLfloat	diffuse2 []

GLfloat [specular2](#) []

Function Documentation

void clearMaterial()

Sets the material to default values.

Definition at line [18](#) of file [rendering.cpp](#).

References [ambient1](#), [diffuse1](#), and [specular1](#).

Referenced by [renderCube\(\)](#), [renderCylinder\(\)](#), and [renderSphere\(\)](#).

```
void renderCube( GLdouble size,
                GLfloat ambr,
                GLfloat ambg,
                GLfloat ambb,
                GLfloat difr,
                GLfloat difg,
                GLfloat difb,
                GLfloat specr,
                GLfloat specg,
                GLfloat specb,
                GLfloat shine
                )
```

Sets the color values of the cube and calls glutSolidCube(size) to display it. The arguments are the length of the edges and the different color values in rgb mode. After displaying the cube, clearMaterial is called.

Definition at line [25](#) of file [rendering.cpp](#).

References [clearMaterial\(\)](#).

Referenced by [drawItems\(\)](#).

Here is the call graph for this function:



```
void renderCylinder( GLdouble base,
                   GLdouble top,
                   GLdouble height,
                   GLint slices,
                   GLint stacks,
                   GLfloat ambr,
                   GLfloat ambg,
                   GLfloat ambb,
                   GLfloat difr,
                   GLfloat difg,
                   GLfloat difb,
                   GLfloat specr,
                   GLfloat specg,
```

```

        GLfloat  specb,
        GLfloat  shine
    )

```

Sets the color values of the cylinder and calls `gluCylinder(radius, radius, height, slices, stacks)` as well as two times `gluDisk(radius, radius, height, slices, stacks)` to display it. The arguments are the radius of the lower side and the upper side, the height, the number of subdivisions around the Z axis (slices), the number of subdivisions along the Z-axis (stacks) and the different color values in rgb mode. After displaying the cylinder, `clearMaterial` is called.

Definition at line [57](#) of file [rendering.cpp](#).

References [clearMaterial\(\)](#).

Referenced by [drawItems\(\)](#).

Here is the call graph for this function:



```

void renderSphere( GLdouble radius,
                  GLint    slices,
                  GLint    stacks,
                  GLfloat  ambr,
                  GLfloat  ambg,
                  GLfloat  ambb,
                  GLfloat  difr,
                  GLfloat  difg,
                  GLfloat  difb,
                  GLfloat  specr,
                  GLfloat  specg,
                  GLfloat  specb,
                  GLfloat  shine
    )

```

Sets the color values of the sphere and calls `glutSolidSphere(radius, slices, stacks)` to display it. The arguments are the radius, the number of subdivisions around the Z axis (slices), the number of subdivisions along the Z-axis (stacks) and the different color values in rgb mode. After displaying the sphere, `clearMaterial` is called.

Definition at line [41](#) of file [rendering.cpp](#).

References [clearMaterial\(\)](#).

Here is the call graph for this function:



Variable Documentation

GLfloat [ambient1\[\]](#)

The array of GLfloats stores the ambient values of the first light source.

Definition at line [7](#) of file [rendering.h](#).

GLfloat [ambient2](#) []

The array of GLfloats stores the ambient values of the second light source.

Definition at line [14](#) of file [rendering.h](#).

GLfloat [diffuse1](#) []

The array of GLfloats stores the diffuse values of the first light source.

Definition at line [9](#) of file [rendering.h](#).

GLfloat [diffuse2](#) []

The array of GLfloats stores the diffuse values of the second light source.

Definition at line [16](#) of file [rendering.h](#).

GLfloat [specular1](#) []

The array of GLfloats stores the specular values of the first light source.

Definition at line [11](#) of file [rendering.h](#).

GLfloat [specular2](#) []

The array of GLfloats stores the specular values of the second light source.

Definition at line [18](#) of file [rendering.h](#).

The actual source code can be found in the appendix

game.cpp	Appendix a
loadWorld.cpp	Appendix b
loadWorld.h	Appendix c
rendering.cpp	Appendix d
rendering.h	Appendix e
Screenshots of the game	Appendix f

6. Appendix

a) game.cpp

```
00001 /*
00002  * Copyright (c) 1993-1997, Silicon Graphics, Inc.
00003  * ALL RIGHTS RESERVED
00004  * Permission to use, copy, modify, and distribute this software for
00005  * any purpose and without fee is hereby granted, provided that the above
00006  * copyright notice appear in all copies and that both the copyright notice
00007  * and this permission notice appear in supporting documentation, and that
00008  * the name of Silicon Graphics, Inc. not be used in advertising
00009  * or publicity pertaining to distribution of the software without specific,
00010  * written prior permission.
00011  *
00012  * THE MATERIAL EMBODIED ON THIS SOFTWARE IS PROVIDED TO YOU "AS-IS"
00013  * AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE,
00014  * INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR
00015  * FITNESS FOR A PARTICULAR PURPOSE.  IN NO EVENT SHALL SILICON
00016  * GRAPHICS, INC.  BE LIABLE TO YOU OR ANYONE ELSE FOR ANY DIRECT,
00017  * SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY
00018  * KIND, OR ANY DAMAGES WHATSOEVER, INCLUDING WITHOUT LIMITATION,
00019  * LOSS OF PROFIT, LOSS OF USE, SAVINGS OR REVENUE, OR THE CLAIMS OF
00020  * THIRD PARTIES, WHETHER OR NOT SILICON GRAPHICS, INC.  HAS BEEN
00021  * ADVISED OF THE POSSIBILITY OF SUCH LOSS, HOWEVER CAUSED AND ON
00022  * ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE
00023  * POSSESSION, USE OR PERFORMANCE OF THIS SOFTWARE.
00024  *
00025  * US Government Users Restricted Rights
00026  * Use, duplication, or disclosure by the Government is subject to
00027  * restrictions set forth in FAR 52.227.19(c)(2) or subparagraph
00028  * (c)(1)(ii) of the Rights in Technical Data and Computer Software
00029  * clause at DFARS 252.227-7013 and/or in similar or successor
00030  * clauses in the FAR or the DOD or NASA FAR Supplement.
00031  * Unpublished-- rights reserved under the copyright laws of the
00032  * United States.  Contractor/manufacturer is Silicon Graphics,
00033  * Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94039-7311.
00034  *
00035  * OpenGL(R) is a registered trademark of Silicon Graphics, Inc.
00036  */
00037
00038 #include "rendering.h"
00039 #include "loadWorld.h"
00040 #include <GL\glaux.h>
00041 #include <stdio.h>
00042 #include <GL\glut.h>
00043 #include <math.h>
00044 #include <iostream>
00045 #include <windows.h>
00046 using std::cout;
00047 using std::endl;
00048 using namespace std;
00049
00051 const double movement = 0.2;
00052
00055 int noOfEvents;
00056
```

```

00058 item *events;
00059
00062 bool leftThroughEntrance = false;
00063
00065 int money = 0;
00066
00069 int xscale = 4;
00072 int yscale = 6;
00075 int zscale = 4;
00076
00078 double xpos = -0.5*xscale;
00080 double ypos = -0.5*yscale;
00082 double zpos = 0;
00083
00086 double yrot = 0;
00087
00089 int angle = 0;
00090
00092 bool fCheckPosition = true;
00093
00095 const double PI = 3.14159265;
00096
00097 void display();
00098
00102 void init(void)
00103 {
00104     // Welcome message on the screen
00105     cout << "Welcome at the CRAZY LABYRINTH!\nGood luck for the game and...
don't be scared!\n" << endl;
00106
00107     // initialize events from file events.txt
00108     FILE* eventStream = fopen("events.txt", "rt");
00109     char oneline[255];
00110     readstr(eventStream,oneline);
00111     sscanf(oneline, " NUMITEMS %d \n", &noOfEvents);
00112     float x, y, z;
00113     int type;
00114     events = new item[noOfEvents];
00115     for(int i=0; i<noOfEvents; i++)
00116     {
00117         readstr(eventStream,oneline);
00118         sscanf(oneline, " %d %f %f %f", &type, &x, &y, &z);
00119         events[i].type = type;
00120         events[i].p.x = x;
00121         events[i].p.y = y;
00122         events[i].p.z = z;
00123         events[i].found = false;
00124     }
00125
00126
00127     if (!LoadGLTextures())
// Jump To Texture Loading Routine ( NEW )
00128     {
00129         return; // If
Texture Didn't Load Return FALSE ( NEW )
00130     }
00131
00132     GLfloat position1[] = {0.0, 3.0, 3.0, 0.0};
00133     GLfloat position2[] = {0.0, 3.0, -3.0, 0.0};

```



```

00193         while(xpos > xposn)
00194         {
00195             xpos = xpos - 0.01;
00196             display();
00197         }
00198         fCheckPosition = true;
00199     }
00200
00201     // Elevator to move to lower level
00202     if(zpos>3.1*zscale && zpos<3.9*zscale && xpos<4.9*xscale &&
xpos>4.1*zscale && ypos<-1*yyscale)
00203     {
00204         fCheckPosition = false;
00205         double yposn = ypos+1*yyscale;
00206         while(ypos < yposn)
00207         {
00208             ypos = ypos+0.2;
00209             display();
00210         }
00211         fCheckPosition = true;
00212     }
00213
00214     // found item
00215     for(int i=0; i<noOfEvents; i++)
00216     {
00217         if( !events[i].found
00218             && -xpos<(events[i].p.x+0.5)*xscale && -
xpos>(events[i].p.x-0.5)*xscale
00219             && -ypos<(events[i].p.y+0.5)*yscale && -ypos>(events[i].p.y-
0.5)*yscale
00220             && -zpos<(events[i].p.z+0.5)*zscale && -zpos>(events[i].p.z-
0.5)*zscale )
00221         {
00222             if(events[i].type == 1) // gold coin
00223             {
00224                 money++;
00225                 cout << "You found a gold coin! Total amount of
gold coins found: " << money << endl;
00226             }
00227             if(events[i].type == 2)
00228                 cout << "You found a key." << endl;
00229             events[i].found = true;
00230         }
00231     }
00232 }
00233
00236 void drawItems()
00237 {
00238     for(int i=0; i<noOfEvents; i++) // render the events from events[]
00239     {
00240         if(!events[i].found)
00241         {
00242             glPushMatrix();
00243             glTranslatef(events[i].p.x*xscale, events[i].p.y*yyscale,
events[i].p.z*zscale);
00244             if(events[i].type == 1) // Coin
00245             {
00246                 glRotatef(angle, 0, 1, 0);
00247                 renderCylinder(0.3, 0.3, 0.1, 20, 20, 0.2, 0.2,
0, 1, 1, 0, 1, 1, 1, 1);

```

```

00248         }
00249         if(events[i].type == 2) // Keys
00250         {
00251             glRotatef(angle, 0, 0, 1);
00252             // Cylinder
00253             glPushMatrix();
00254             glTranslatef(-0.1, 0, 0.2*zscale);
00255             glRotatef(90, 0, 1, 0);
00256             renderCylinder(0.4, 0.4, 0.2, 20, 20, 0.0, 0.2,
0.0, 0.1, 1, 0.1, 0.4, 1, 0.4, 1);
00257             glPopMatrix();
00258             // "shaft"
00259             glPushMatrix();
00260             glScalef(0.19, 0.2, 0.45*zscale);
00261             renderCube(1, 0.0, 0.2, 0.0, 0.1, 1, 0.1, 0.7,
1, 0.7, 1);
00262             glPopMatrix();
00263             // first tooth
00264             glPushMatrix();
00265             glTranslatef(0, -0.3, -0.1*zscale);
00266             glScalef(0.19, 0.6, 0.2);
00267             renderCube(1, 0.0, 0.2, 0.0, 0.1, 1, 0.1, 0., 1,
0.7, 1);
00268             glPopMatrix();
00269             // second tooth
00270             glTranslatef(0, -0.3, -0.2*zscale);
00271             glScalef(0.19, 0.6, 0.2);
00272             renderCube(1, 0.0, 0.2, 0.0, 0.1, 1, 0.1, 0.7,
1, 0.7, 1);
00273         }
00274         glPopMatrix();
00275     }
00276 }
00277 int temp = angle+1;
00278 angle = temp%360;
00279
00280 // render up arrow for elevator
00281 glPushMatrix();
00282 glMatrixMode(GL_MODELVIEW);
00283 glTranslatef(-4.5*xscale, 0.3*yscale, -3.5*zscale);
00284 glRotatef(angle, 0, 1, 0);
00285 glPushMatrix();
00286 glScalef(0.4, 1, 0.4);
00287 renderCube(0.3*yscale, 0.2, 0, 0, 1, 0, 0, 1, 0, 0, 0.3);
00288 glPopMatrix();
00289 glPushMatrix();
00290 glTranslatef(0, 0.15*yscale, -0.065*zscale);
00291 glRotatef(45, 1, 0, 0);
00292 glScalef(1, 1, 0.2);
00293 renderCube(0.15*yscale, 0.2, 0, 0, 1, 0, 0, 1, 0, 0, 0.3);
00294 glPopMatrix();
00295 glTranslatef(0, 0.15*yscale, 0.065*zscale);
00296 glRotatef(-45, 1, 0, 0);
00297 glScalef(1, 1, 0.2);
00298 renderCube(0.15*yscale, 0.2, 0, 0, 1, 0, 0, 1, 0, 0, 0.3);
00299 glPopMatrix();
00300 }
00301
00306 void display(void)

```

```

00307 {
00308     if(fCheckPosition)
00309         checkPosition();
00310     GLfloat sceneroty = -yrot; // 360 Degree
Angle For Player Direction
00311
00312     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
00313     glLoadIdentity();
00314
00315     glPushMatrix(); // Player's View
00316
00317     glRotatef(sceneroty,0,1,0); // Rotate Depending On Direction Player
Is Facing
00318     glTranslatef(xpos, ypos, zpos);
00319
00320     glPushMatrix(); // draw World
00321     glPushMatrix();
00322     glScalef(xscale, yscale, zscale);
00323     drawScene();
00324     glPopMatrix();
00325     drawItems();
00326     glPopMatrix(); // draw World;
00327
00328     glPopMatrix(); // player rotation and translation
00329     glutSwapBuffers();
00330 }
00331
00334 void reshape (int w, int h)
00335 {
00336     glViewport (0, 0, (GLsizei) w, (GLsizei) h);
00337     glMatrixMode (GL_PROJECTION);
00338     glLoadIdentity ();
00339     gluPerspective(65.0, (GLfloat) w/(GLfloat) h, 0.1, 40.0);
00340     glMatrixMode(GL_MODELVIEW);
00341     glLoadIdentity();
00342 }
00343
00347 bool checkCollision(double newX, double newY, double newZ)
00348 {
00349     SECTOR sector1 = getSector1();
00350
00351     float ux, uy, uz, lx, ly, lz;
00352     for(int i = sector1.numTrianglesFloorCeiling; i < sector1.numtriangles;
i+=2)
00353     {
00354         ux = sector1.triangle[i].vertex[0].x;
00355         uy = sector1.triangle[i].vertex[0].y;
00356         uz = sector1.triangle[i].vertex[0].z;
00357         lx = sector1.triangle[i].vertex[2].x;
00358         ly = sector1.triangle[i].vertex[2].y;
00359         lz = sector1.triangle[i].vertex[2].z;
00360
00361         // check which vertex is left
00362         if (ux > lx || uz > lz) // exchange vertices such that
ux,uy,uz is upper left
00363         {
00364             int tmp = ux;
00365             ux = lx;
00366             lx = tmp;

```

```

00367             tmp = uz;
00368             uz = lz;
00369             lz = tmp;
00370         }
00371
00372         if (-newX <= (lx*xscale+movement) && -newX >= (ux*xscale-
movement)
00373             && -newZ <= (lz*zscale+movement) && -newZ >= (uz*zscale-
movement)
00374             && -newY <= (uy*yscale+movement) && -newY >= (ly*yscale-
movement) )
00375         {
00376             return false;
00377         }
00378     }
00379     return true;    // no wall hit
00380 }
00381
00382 void keyboard (unsigned char key, int x, int y)
00383 {
00384     double newXpos, newYpos, newZpos;
00385     double yrotn = 2*PI*yrot/360;
00386     switch (key)
00387     {
00388     case 'y': // moving up
00389         ypos += movement;
00390         glutPostRedisplay();
00391         break;
00392     case 'x': // moving down
00393         ypos -= movement;
00394         glutPostRedisplay();
00395         break;
00396     case 'a': // moving left
00397         newXpos = cos(yrotn) * movement + xpos;
00398         newZpos = -sin(yrotn) * movement + zpos;
00399         if(checkCollision(newXpos, ypos, newZpos))
00400         {
00401             zpos = newZpos;
00402             xpos = newXpos;
00403         }
00404         glutPostRedisplay();
00405         break;
00406     case 'd': // moving right
00407         newXpos = -cos(yrotn) * movement + xpos;
00408         newZpos = sin(yrotn) * movement + zpos;
00409         if(checkCollision(newXpos, ypos, newZpos))
00410         {
00411             zpos = newZpos;
00412             xpos = newXpos;
00413         }
00414         glutPostRedisplay();
00415         break;
00416     case 'w': // move forwards
00417         newXpos = sin(yrotn) * movement + xpos;
00418         newZpos = cos(yrotn) * movement + zpos;
00419         if(checkCollision(newXpos, ypos, newZpos))
00420         {
00421             zpos = newZpos;
00422             xpos = newXpos;

```

```

00429         }
00430         glutPostRedisplay();
00431         break;
00432     case 's': // move backwards
00433         newXpos = -sin(yrotn) * movement + xpos;
00434         newZpos = -cos(yrotn) * movement + zpos;
00435         if(checkCollision(xpos, ypos, newZpos))
00436         {
00437             xpos = newXpos;
00438             zpos = newZpos;
00439         }
00440         glutPostRedisplay();
00441         break;
00442     case 'q': // turn left
00443         yrot += 2;
00444         while(yrot > 0)
00445         {
00446             yrot = yrot-360;
00447         }
00448         glutPostRedisplay();
00449         break;
00450     case 'e': // turn right
00451         yrot -= 2;
00452         while(yrot <= 360)
00453         {
00454             yrot = yrot+360;
00455         }
00456         glutPostRedisplay();
00457         break;
00458     case 'n': // move to key
00459         xpos = 0.5*xscale;
00460         zpos = 4.5*zscale;
00461         yrot = 90;
00462         break;
00463     case 'm': // move to elevator
00464         xpos = 5.5*xscale;
00465         zpos = 3.5*zscale;
00466         yrot = -90;
00467         glutPostRedisplay();
00468         break;
00469     case 27:
00470         exit(0);
00471         break;
00472     default:
00473         break;
00474 }
00475 }
00476
00482 int main(int argc, char** argv)
00483 {
00484     glutInit(&argc, argv);
00485     glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
00486     glutInitWindowSize (500, 500);
00487     glutInitWindowPosition (100, 100);
00488     glutCreateWindow (argv[0]);
00489     init ();
00490     glutDisplayFunc(display);
00491     glutIdleFunc(display);

```

```
00492     glutReshapeFunc(reshape);
00493     glutKeyboardFunc(keyboard);
00494     glutMainLoop();
00495     return 0;
00496 }
```

b) loadWorld.cpp

```
00001 #include "loadWorld.h"
00002
00004 GLuint texture1[3];
00006 GLuint texture2[3];
00008 GLuint texture3[3];
00010 GLuint texture4[3];
00011
00013 SECTOR sector1;
00014
00015 SECTOR getSector1()
00016 {
00017     return sector1;
00018 }
00019
00020 void readstr(FILE *f, char *string) // reads in a string from file f
00021 {
00022     do
00023     {
00024         fgets(string, 255, f);
00025     } while ((string[0] == '/') || (string[0] == '\n'));
00026     return;
00027 }
00028
00029 void SetupWorld() // reads in the wall data from world.txt into sector1
00030 {
00031     float x, y, z, u, v;
00032     int numtriangles;
00033     int numTrianglesFloorCeiling;
00034     FILE *filein;
00035     char oneline[255];
00036     filein = fopen("World.txt", "rt"); // File To Load World Data From
00037
00038     readstr(filein,oneline);
00039     sscanf(oneline, "NUMPOLLIES %d NUMFLOORCEILING %d\n", &numtriangles,
&numTrianglesFloorCeiling);
00040
00041     sector1.triangle = new TRIANGLE[numtriangles];
00042     sector1.numtriangles = numtriangles;
00043     sector1.numTrianglesFloorCeiling = numTrianglesFloorCeiling;
00044     for (int loop = 0; loop < numtriangles; loop++)
00045     {
00046         for (int vert = 0; vert < 3; vert++)
00047         {
00048             readstr(filein,oneline);
00049             sscanf(oneline, "%f %f %f %f %f", &x, &y, &z, &u, &v);
00050             sector1.triangle[loop].vertex[vert].x = x;
00051             sector1.triangle[loop].vertex[vert].y = y;
00052             sector1.triangle[loop].vertex[vert].z = z;
00053             sector1.triangle[loop].vertex[vert].u = u;
00054             sector1.triangle[loop].vertex[vert].v = v;
00055         }
00056     }
00057     fclose(filein);
00058     return;
00059 }
00060
```

```

00061 AUX_RGBImageRec *LoadBMP(char *Filename)           // Loads A Bitmap Image
00062 {
00063     FILE *File=NULL;                                  // File Handle
00064
00065     if (!Filename)                                   // Make Sure A Filename Was Given
00066     {
00067         return NULL;                                 // If Not Return NULL
00068     }
00069
00070     File=fopen(Filename,"r");                         // Check To See If The File Exists
00071
00072     if (File)                                        // Does The File Exist?
00073     {
00074         fclose(File);                                // Close The Handle
00075         return auxDIBImageLoad(Filename);
00076         // Load The Bitmap And Return A Pointer
00077     }
00078     return NULL;                                     // If Load Failed Return NULL
00079 }
00080
00081
00082 int LoadGLTexture(char* textureName, GLuint* memory)
00083 {
00084     int Status=false;                                // Status Indicator
00085
00086     AUX_RGBImageRec *TextureImage[2]; // Create Storage Space For The Texture
00087
00088     memset(TextureImage,0,sizeof(void *)*1);        // Set The Pointer To NULL
00089
00090     // Load The Bitmap, Check For Errors, If Bitmap's Not Found Quit
00091     if ((TextureImage[0]=LoadBMP(textureName)))
00092     {
00093         Status=true;                                 // Set The Status To TRUE
00094
00095         glGenTextures(3, &memory[0]);               // Create Three Textures
00096
00097         // Create MipMapped Texture
00098         glBindTexture(GL_TEXTURE_2D, memory[2]);
00099         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
00100         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
00101         gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[0]->sizeX,
TextureImage[0]->sizeY, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
00102     }
00103     if (TextureImage[0])                             // If Texture Exists
00104     {
00105         if (TextureImage[0]->data)                 // If Texture Image Exists
00106         {
00107             free(TextureImage[0]->data); // Free The Texture Image Memory
00108         }
00109
00110         free(TextureImage[0]);                       // Free The Image Structure
00111     }
00112
00113     return Status;                                   // Return The Status
00114 }
00115
00116 int LoadGLTextures()
00117 {

```

```

00118         return (LoadGLTexture("stone_floor3.bmp", texture1) &&
LoadGLTexture("stone_floor3.bmp", texture2) && LoadGLTexture("sternenhimmel.bmp",
texture3)&& LoadGLTexture("brick.bmp", texture4));
00119     }
00120
00121 void processTriangles(int start, int end)
00122 {
00123     GLfloat x_m, y_m, z_m, u_m, v_m;
00124
00125     for (int loop_m = start; loop_m < end; loop_m++)           // Loop
Through All The Triangles
00126     {
00127         glBegin(GL_TRIANGLES);
// Start Drawing Triangles
00128         glNormal3f( 0.0f, 0.0f, 1.0f);
// Normal Pointing Forward
00129         for(int vert = 0; vert <= 2; vert++)
00130         {
00131             x_m = sector1.triangle[loop_m].vertex[vert].x;
// X Vertex
00132             y_m = sector1.triangle[loop_m].vertex[vert].y;
// Y Vertex
00133             z_m = sector1.triangle[loop_m].vertex[vert].z;
// Z Vertex
00134             u_m = sector1.triangle[loop_m].vertex[vert].u;
// U Texture Coord
00135             v_m = sector1.triangle[loop_m].vertex[vert].v;
// V Texture Coord
00136             glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m); // Set
The TexCoord And Vertice
00137         }
00138         glEnd();           // Done
Drawing Triangles
00139     }
00140 }
00141
00142 void drawScene()
00143 {
00144     // walls
00145     int numtriangles = sector1.numtriangles;
00146     int numTrianglesFloorCeiling = sector1.numTrianglesFloorCeiling;
00147
00148     glBindTexture(GL_TEXTURE_2D, texture1[2]);
00149     processTriangles(0, 2); // Process Floor and Ceiling
00150
00151     glBindTexture(GL_TEXTURE_2D, texture2[2]);
00152     processTriangles(2, 10); // Process Floor and Ceiling
00153
00154     glBindTexture(GL_TEXTURE_2D, texture3[2]);
00155     processTriangles(10, numTrianglesFloorCeiling); // Process Floor and Ceiling
00156
00157     glBindTexture(GL_TEXTURE_2D, texture4[2]);
00158     processTriangles(numTrianglesFloorCeiling, numtriangles); // Process Walls
00159 }

```

c) loadWorld.h

```
00001 #include <GL\glaux.h>
00002 #include <stdio.h>
00003 #include <GL\glut.h>
00004 #include <math.h>
00005 #include <iostream>
00006 using std::cout;
00007
00008 #ifndef LOADWORLD
00009 #define LOADWORLD
00010
00012 struct position
00013 {
00015     float x;
00017     float y;
00019     float z;
00020 };
00021
00023 struct item
00024 {
00026     position p;
00028     int type;
00031     bool found;
00032 };
00033
00036 typedef struct tagVERTEX // used in tagTRIANGLE
00037 {
00039     float x;
00041     float y;
00043     float z;
00046     float u;
00049     float v;
00050 } VERTEX;
00051
00053 typedef struct tagTRIANGLE // used in tagSECTOR
00054 {
00056     VERTEX vertex[3];
00057 } TRIANGLE;
00058
00060 typedef struct tagSECTOR // data Structure to store the walls
00061 {
00063     int numtriangles;
00065     int numTrianglesFloorCeiling;
00067     TRIANGLE* triangle;
00068 } SECTOR;
00069
00073 SECTOR getSector1();
00074
00077 void readstr(FILE *f, char *string);
00078
00081 void SetupWorld();
00082
00084 AUX_RGBImageRec *LoadBMP(char *Filename);
00085
00088 int LoadGLTexture(char* textureName, GLuint* memory) ;
00089
00092 int LoadGLTextures();
```

```

00093
00096 void processTriangles(int start, int end);
00097
00101 void drawScene();
00102
00103 #endif

```

d) [rendering.cpp](#)

```

00001 #include "rendering.h"
00002
00003 // Material settings:
00005 GLfloat ambient1[] = {1.0, 1, 1.0, 1.0};
00007 GLfloat diffuse1[] = {1, 1.0, 1.0, 1.0};
00009 GLfloat specular1[] = {1.0, 1, 1.0, 1.0};
00010
00012 GLfloat ambient2[] = {1.0, 0.5, 1.0, 1.0};
00014 GLfloat diffuse2[] = {1, 0.5, 1.0, 1.0};
00016 GLfloat specular2[] = {1.0, 0.5, 1.0, 1.0};
00017
00018 void clearMaterial()
00019 {
00020     glMaterialfv(GL_FRONT, GL_AMBIENT, ambient1);
00021     glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse1);
00022     glMaterialfv(GL_FRONT, GL_SPECULAR, specular1);
00023 }
00024
00025 void renderCube(GLdouble size, GLfloat ambr, GLfloat ambg, GLfloat ambb, GLfloat difr,
GLfloat difg, GLfloat difb,
00026     GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
00027 {
00028     GLfloat mat[4];
00029
00030     mat[0] = ambr; mat[1] = ambg; mat[2] = ambb; mat[3] = 1.0;
00031     glMaterialfv(GL_FRONT, GL_AMBIENT, mat);
00032     mat[0] = difr; mat[1] = difg; mat[2] = difb;
00033     glMaterialfv(GL_FRONT, GL_DIFFUSE, mat);
00034     mat[0] = specr; mat[1] = specg; mat[2] = specb;
00035     glMaterialfv(GL_FRONT, GL_SPECULAR, mat);
00036     glMaterialf(GL_FRONT, GL_SHININESS, shine * 128.0);
00037     glutSolidCube(size);
00038     clearMaterial();
00039 }
00040
00041 void renderSphere(GLdouble radius, GLint slices, GLint stacks, GLfloat ambr, GLfloat
ambg, GLfloat ambb,
00042     GLfloat difr, GLfloat difg, GLfloat difb, GLfloat specr, GLfloat specg, GLfloat
specb, GLfloat shine)
00043 {
00044     GLfloat mat[4];
00045
00046     mat[0] = ambr; mat[1] = ambg; mat[2] = ambb; mat[3] = 1.0;
00047     glMaterialfv(GL_FRONT, GL_AMBIENT, mat);
00048     mat[0] = difr; mat[1] = difg; mat[2] = difb;
00049     glMaterialfv(GL_FRONT, GL_DIFFUSE, mat);
00050     mat[0] = specr; mat[1] = specg; mat[2] = specb;
00051     glMaterialfv(GL_FRONT, GL_SPECULAR, mat);
00052     glMaterialf(GL_FRONT, GL_SHININESS, shine * 128.0);

```

```

00053     glutSolidSphere(radius, slices, stacks);
00054     clearMaterial();
00055 }
00056
00057 void renderCylinder(GLdouble base, GLdouble top, GLdouble height, GLint slices, GLint
stacks,
00058     GLfloat ambr, GLfloat ambg, GLfloat ambb, GLfloat difr, GLfloat difg, GLfloat
difb,
00059     GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
00060 {
00061     GLUquadric* coin = gluNewQuadric();
00062     GLUquadric* up = gluNewQuadric();
00063     GLUquadric* down = gluNewQuadric();
00064
00065     GLfloat mat[4];
00066
00067     mat[0] = ambr; mat[1] = ambg; mat[2] = ambb; mat[3] = 1.0;
00068     glMaterialfv(GL_FRONT, GL_AMBIENT, mat);
00069     mat[0] = difr; mat[1] = difg; mat[2] = difb;
00070     glMaterialfv(GL_FRONT, GL_DIFFUSE, mat);
00071     mat[0] = specr; mat[1] = specg; mat[2] = specb;
00072     glMaterialfv(GL_FRONT, GL_SPECULAR, mat);
00073     glMaterialf(GL_FRONT, GL_SHININESS, shine * 128.0);
00074
00075     glPushMatrix();
00076     glTranslatef(0,0,height);
00077     gluDisk(down, 0, base, slices, stacks);
00078     glPopMatrix();
00079     gluCylinder(coin, base, top, height, slices, stacks);
00080     glRotatef(180, 1, 0, 0);
00081     gluDisk(up, 0, base, slices, stacks);
00082
00083     clearMaterial();
00084 }

```

e) rendering.h

```

00001 #include "loadWorld.h"
00002
00003 #ifndef RENDERING
00004 #define RENDERING
00005
00007 GLfloat ambient1[];
00009 GLfloat diffuse1[];
00011 GLfloat specular1[];
00012
00014 GLfloat ambient2[];
00016 GLfloat diffuse2[];
00018 GLfloat specular2[];
00019
00021 void clearMaterial();
00022
00026 void renderCube(GLdouble size, GLfloat ambr, GLfloat ambg, GLfloat ambb, GLfloat difr,
GLfloat difg, GLfloat difb,
00027     GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine);
00028
00034 void renderSphere(GLdouble radius, GLint slices, GLint stacks, GLfloat ambr, GLfloat
ambg, GLfloat ambb,

```

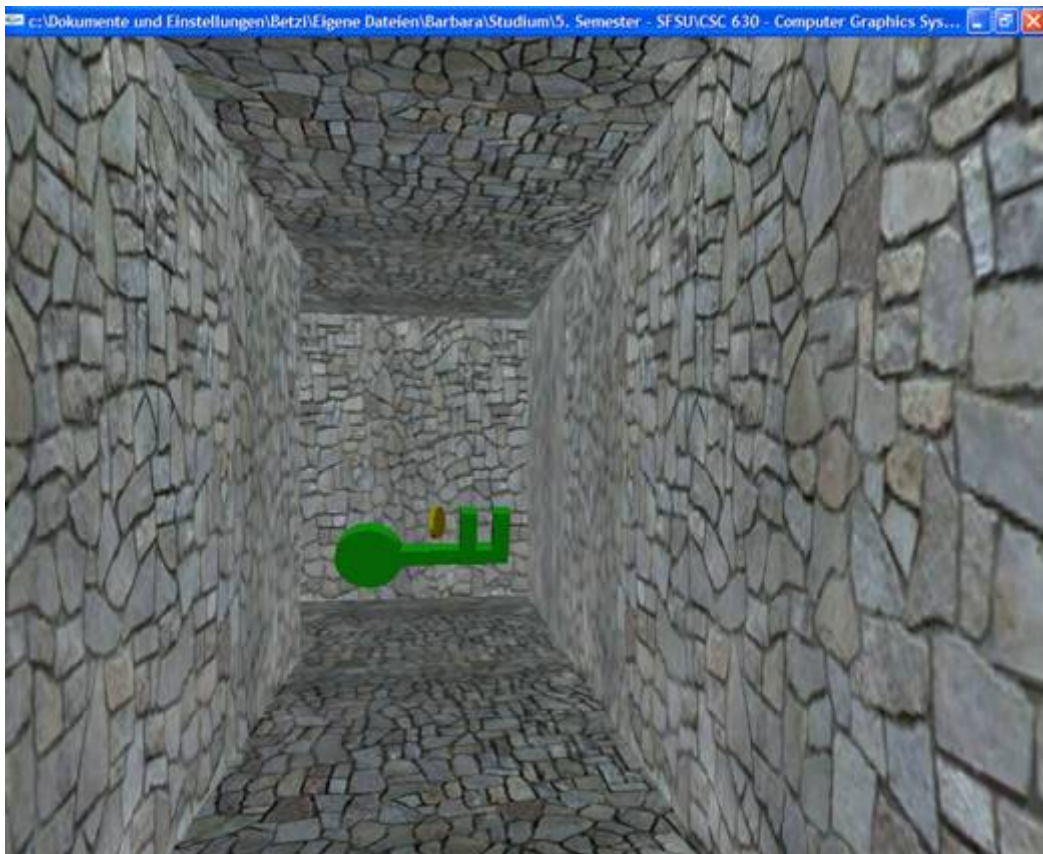
```
00035     GLfloat difr, GLfloat difg, GLfloat difb, GLfloat specr, GLfloat specg, GLfloat
specb, GLfloat shine);
00036
00044 void renderCylinder(GLdouble base, GLdouble top, GLdouble height, GLint slices, GLint
stacks,
00045         GLfloat ambr, GLfloat ambg, GLfloat ambb, GLfloat difr, GLfloat difg, GLfloat
difb,
00046         GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine);
00047
00048 #endif
```

f) Screenshots

1. First floor, the entrance of the labyrinth



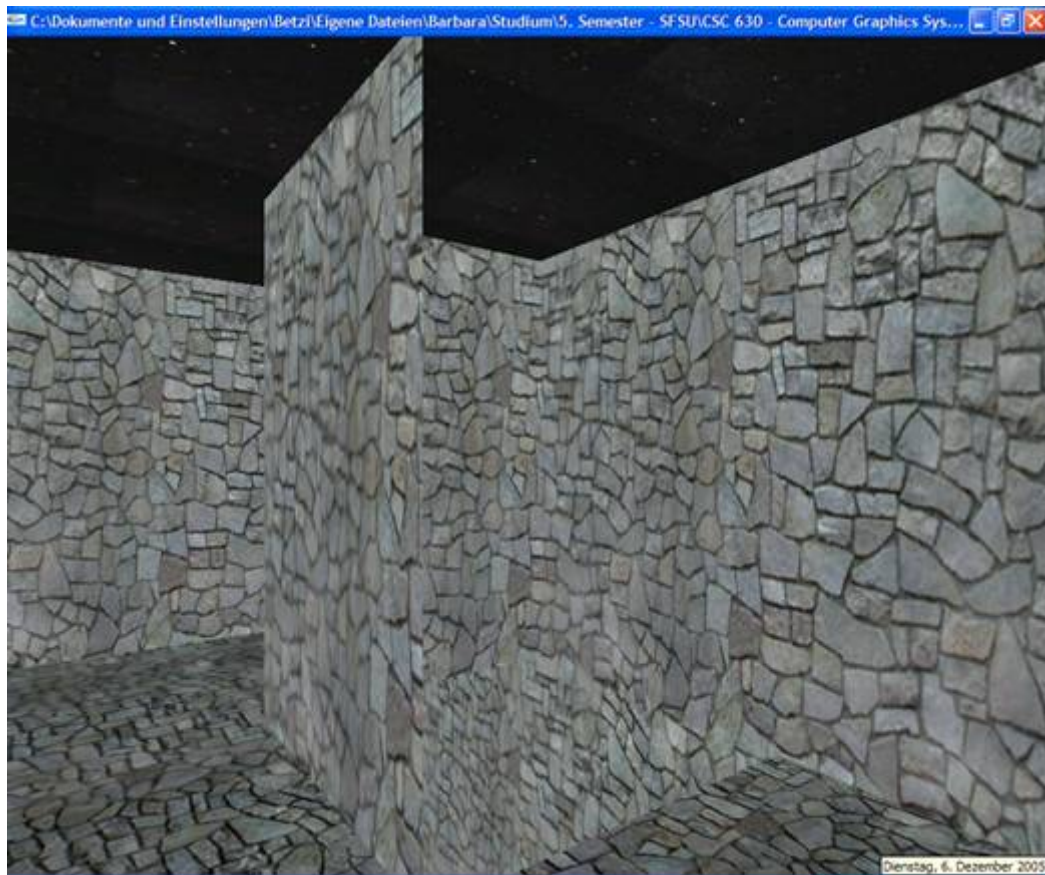
2. First floor, the key for the elevator



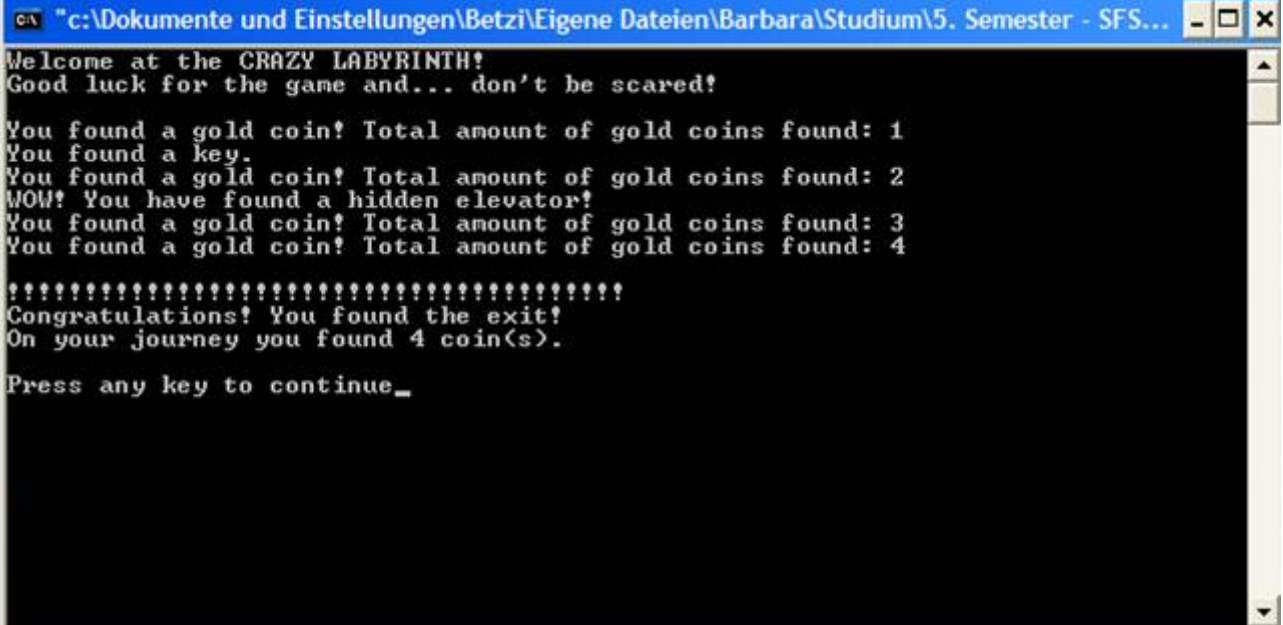
3. First floor, the elevator



4. Second floor, after using the elevator



5. After exit has been found



```

c:\Dokumente und Einstellungen\Betzi\Eigene Dateien\Barbara\Studium\5. Semester - SFS...
Welcome at the CRAZY LABYRINTH!
Good luck for the game and... don't be scared!

You found a gold coin! Total amount of gold coins found: 1
You found a key.
You found a gold coin! Total amount of gold coins found: 2
WOW! You have found a hidden elevator!
You found a gold coin! Total amount of gold coins found: 3
You found a gold coin! Total amount of gold coins found: 4

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Congratulations! You found the exit!
On your journey you found 4 coin(s).

Press any key to continue_

```