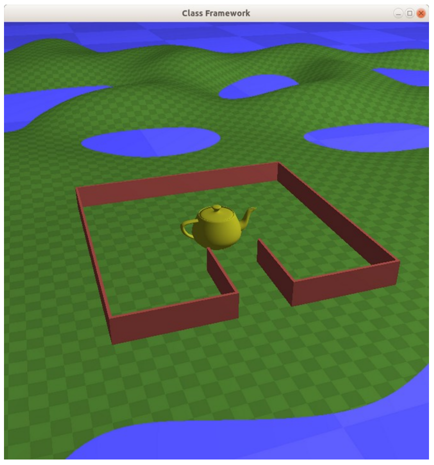
CS541 Project 1: Transformations and Navigation

Synopsis

Implement transformations for viewing and navigating a 3D scene.
Note: You will need to copy a file **libs/glbinding.dll** into your **Release** and **Debug** folders in order to run the executable.

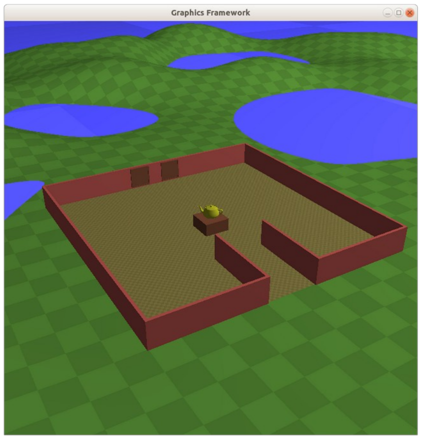


Instructions

Download and unzip the framework provided with this project.
This project is divided into thre tasks, all having to do with transformations.

1. Implement the basic 3D transformations,
2. Use the transformations for interactive viewing.
3. Use the transformations to implement game like navigation.

As distributed, the framework will produce image 1. Once the transformations are implemented (task 1), the framework will produce image 2.



Task 1: Implement 3D transformations.

The framework provides (in **transform.h** and **transform.cpp**) a class named **Transformation**, which provides all the operations necessary to implement transformations for viewing, navigation, and hierarchical modeling. The methods of class **Transformation** are mostly unimplemented, and for task 1, you are to provide those implementations.

The class **Transformation** provides the following data attributes and methods:

- **Rotate(axis, angle):** Returns a rotation matrix.
- **Scale(x,y,z):** Returns a scale matrix.
- **Translate(x,y,z):** Returns a translation matrix.
- **Perspective(rx, ry, front, back):** Returns a perspective projection matrix.

A note on indexing 2D arrays in C++ and OpenGL: Math notation for indexing elements in a 2D array is quite different from C++ and OpenGL indexing. Math notation is row-major and one's based while C++/OpenGL is column-major and 0-based. For instance, the three elements filled in for the translation matrix are:

- Math notation: m_{14}, m_{24}, m_{34} . In general m_{rc} .
- Programming: m[3][0], m[3][1], m[3][2]. In general m[c][r].
- 2D notation: $\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Task 2: Interactive scene viewing

In this task, you will implement the ability to manipulate the scene displayed on screen via mouse (and possibly) keyboard) interactions. You will be able to rotate the scene to any viewing angle as well as zoom in/out.
Mouse operations are referred to as **button-down**, **button-up** and **mouse-motion** and **scroll-wheel**.

Requirements:

- The sequence **left-button-down, mouse-motion, left-button-up** causes interactive rotations for as long as the button is down. Horizontal **mouse-motion** causes a turntable like rotation (see **spin**). Vertical **mouse-motion** causes the scene to tilt forward/backward to produce top/side/bottom views (see **tilt**).
- **Scroll-wheel** interactions zoom the scene in/out (see **zoom**).
- The sequence **right-button-down, mouse-motion, right-button-up** causes interactive screen oriented translations (see **tx**, **ty**).
- Any **button-down** should cause no change to the scene until a **mouse-motion** occurs.
- Any **button-up** should cause no change to the scene.
- **A note on mouse-motion directions:** For all three (left-button rotations, right-button translations and scroll-wheel actions), the direction of the motion on the screen should reflect the direction of the mouse-motion. (Beware: It is particularly easy to get this wrong for vertical mouse-motion, with an upward mouse-motion incorrectly causing a downward screen rotation.)
- **A note on mouse-motion scaling:** A comfortable amount of **mouse-motion** should cause a comfortable amount of screen motion, neither too much nor too little.

Tracking the interactions:
I use variables with the following names and meanings, all declared in scene.h and initialized in InitializeScene in scene.cpp.

- **spin** and **tilt** contain the two rotation angles. (Initialize to 0 and 30 degrees respectively.)
- **tx**, **ty**, **zoom** contain the screen left/right/up/down and the zoom in/out translate values (Initialize to 0,0,25).
- **rx**,**ry** contain the perspective frustum width,height controls (Initialize ry=0.4 and always recalculate **rx=ry*WIDTH/HEIGHT** to maintain the same aspect ratio as your screen/window.)
- **front**, **back** contain distances to the front and back clipping planes. (Initialize to 0.5 and 5000.)

Building the transformations:
Replace the hard-coded matrix in scene.cpp (which was created only to give you an initial picture on the screen) with view and projection matrices created as follows:

WorldView = $T(\mathrm{tx}, \mathrm{ty}, -\mathrm{zoom}) * R_X(\mathrm{tilt}-90) * R_Z(\mathrm{spin})$

WorldProj = $P(\mathrm{rx}, \mathrm{ry}, \mathrm{front}, \mathrm{back})$

Task 3: Game-like navigation

Implement interactive navigation operations as found in many games, with the 'a', 'w', 's', and 'd' keys causing motion through the scene, and mouse-motion simulating turning your head left/right/up/down. The most straightforward implementation is:

- Variables
  - **eye** to contain the current position of the eye. (An initial value of (0, -20, 0) works well.
  - **speed** to the distance per second of the eye whenever one of the four keys is down. (An initial value of 10 units/second work well.)
  - Four Booleans **w_down, a_down, s_down, d_down** to record the current pressed/released state of each key.
  - A boolean **transformation-mode**, toggled by the TAB key, to record Task2-transformation mode or Task3 Game-navigation mode.
  - All the previous values **spin, tilt, ry**, **front**, and **back,** etc.
- Code in the interact.cpp callbacks to monitor the pressing and releasing of the four keys and set/reset the four Booleans accordingly, as well as a key to toggle **transformation-mode**.
- Code at the beginning of DrawScene that recalculates the position of the eye at every (1/30 of a second) refresh. Use **glfwGetTime()** to retrieve the time since the program started running, and take the difference between two such values to calculate the amount of time that has passed between successive refreshes.

```
time_since_last_refresh = …
step = speed*time_since_last_refresh;
w_key:      eye += step*vec3(sin(spin), cos(spin), 0.0);
s_key:      eye  -= step*vec3(sin(spin), cos(spin), 0.0);
d_key:      eye += step*vec3(cos(spin), -sin(spin), 0.0);
a_key:      eye  -= step*vec3(cos(spin), -sin(spin), 0.0);
```

- Code after the eye-position calculation to adjust the eye's Z-coordinate to maintain a constant height above the ground. (A height of about 2 meters is a good value here.) The ground level at the eye's (X,Y) can be found by calling **proceduralground->HeightAt(X,Y).**

- Code to calculate the transformations matrices. Depending on the state of **transformation-mode**, use the previous task's calculation, or (for this task)

  WorldView = $R_X(\text{tilt}-90) * R_Z(\text{spin}) * T(-eye)$

  WorldProj = $P(\text{rx}, \text{ry}, \text{front}, \text{back})$

The Framework

Look for comments containing **// @@** to indicate where your implementations should be placed. In particular:

- Implementation of transformations is in **transfom.cpp**
- Declaration of viewing parameters is in **scene.h**
- Initialization of viewing parameters is in **scene.cpp**
- Manipulation of viewing parameters in response to mouse actions in **interact.cpp**
- Building the **WorldView** and **WorldProj** matrices from the viewing parameters is in **scene.cpp**

What/when/how to submit

Use Moodle to submit a single zip file of your project.

- The zip file should contain only source code ***.cpp, *.h** and ***.vcxproj** files ***and nothing else.***
- **DO NOT** include the following in your submitted ZIP file: the **libs** folder, the .**vs**, **Debug**, and **Release** folders, various other created files like **framework.VC.db, framework.VC.opendb, framework.vcxproj.user.**

Grading Rubric	
Task 1: Transformation library is implemented (total 20%)	
All objects are correctly placed, scaled and oriented.	5%
Rotation animations are at a correct speed.	5%
Perspective is correct and responds to window size changes.	10%
Task 2: Interactive transformations: (total 30%)	
Rotations are around the scene center and in the correct direction.	10%
Scrolling in/out is implemented.	10%
Screen translates transform the scene in the correct direction.	10%
Task 3: Game-like navigation: (total 40%)	
Pressing a key (A, S, D, W) starts an appropriate motion.	10%
Releasing a key stops the motion.	5%
Movement is smooth and at a sensible and constant run/walk speed.	10%
Eye height is appropriate (about teapot height).	5%
Eye maintains constant height above the ground .	10%
Other: (5%)	
Keyboard TAB key alternates between tasks 2 and 3 viewing	5%
Zip file (total 5%)	
Zip file contains **source files** (*.cpp, *.h, *.vcxproj) and **nothing else**.	5%

As a debugging ai for task 2:

For task 2:

- Set your state variables to these values: spin=0, tilt=30, tx=ty=0, zoom=25, ry=0.4, front=0.5, and back=5000.
- Find and uncomment these two lines that print **WorldView** and **WorldProj**:

```
//std::cout << "WorldView: " << glm::to_string(WorldView) << std::endl;
//std::cout << "WorldProj: " << glm::to_string(WorldProj) << std::endl;
```

- Compare your results to this. Note that **glm::to_string** prints in in **column-major order**, producing a result transposed from the normal way of writing transformations.

```
WorldView: mat4x4(    (1.000000, 0.000000,  0.000000,   0.000000),
                      (0.000000, 0.500001, -0.866025,   0.000000),
                      (0.000000, 0.866025,  0.500001,   0.000000),
                      (0.000000, 0.000000, -25.000000, 1.000000))

WorldProj: mat4x4(    (2.500000, 0.000000,  0.000000,  0.000000),
                      (0.000000, 2.500000,  0.000000,  0.000000),
                      (0.000000, 0.000000, -1.000200, -1.000000),
                      (0.000000, 0.000000, -1.000100,  0.000000))
```

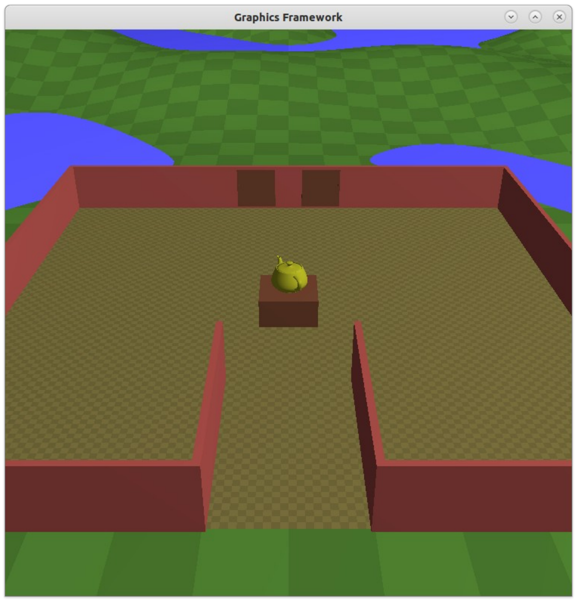
For task 3:

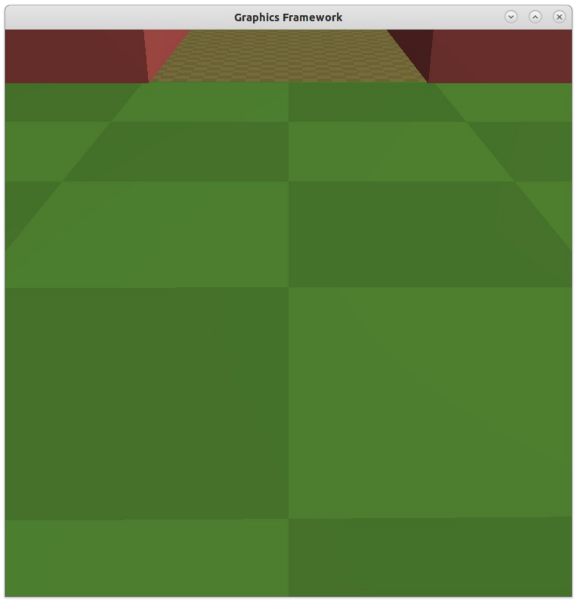
- With values spin=0, tilt=30, eye=(0,-20,0), ry=0.4, front=0.5, and back=5000.
- Expect these results:

```
WorldView: mat4x4(    (1.000000, 0.000000,    0.000000, 0.000000),
                      (0.000000, 0.500001,   -0.866025, 0.000000),
                      (0.000000, 0.866025,    0.500001, 0.000000),
                      (0.000000, 7.780648, -17.823555, 1.000000))

WorldProj: mat4x4(    (2.500000, 0.000000,  0.000000,  0.000000),
                      (0.000000, 2.500000,  0.000000,  0.000000),
                      (0.000000, 0.000000, -1.000200, -1.000000),
                      (0.000000, 0.000000, -1.000100,  0.000000)
```

Images from the above matrices (task2 and task3 respectively):


Graphics Framework


Graphics Framework