CS 488/688: Introduction to Computer Graphics

Winter 2024 Assignment 1: OpenGL

Due: Thursday, January 25, 2024 at 9:00 AM

Summary

In this assignment you will learn the basics of programming real-time 3D graphics using OpenGL. You will create geometric primitives, draw them to the screen, and modify the contents of the 3D environment in response to user interaction. In particular, you will draw a maze from cubes. (There is no texturing, no shading, no redstone, no creepers, no minotaurs, etc.)

The Maze environment

The most obvious feature of the Maze is the grid, which lies on the ground (i.e., the plane $y = 0$) and defines the legal squares upon which blocks of the maze will sit. Each grid cell is a unit square, and the grid itself is an $N \times N$ arrangement of legal cells for some constant N. In the environment, we actually draw a grid of size $(N+2) \times (N+2)$, surrounding the main grid with an extra ring of cells.

You are provided with a routine to generate the maze. The maze consists of a grid of values; a grid cell has a value of 1 if there should be a block in the cell, and 0 of there should not be a block. The base of the block should fill the grid cell; i.e., if two adjacent grid cells both have blocks in them, then there should be no space between the blocks. The result is a 4 point connected arrangement of blocks that form a maze. The outer boundary of the maze has two openings (an entrance and an exit). You will be able to grow and shrink the height of the walls.

The blocks you use to draw the cells of the maze should have one colour, which you'll be able to change with the colour editor. You should also draw a polygon for the floor, which will lie on the grid; you will also be able to edit the colour of this floor. The floor colour should be different than the colour of the maze blocks. The floor should only cover on the NxN portion of the grid. The outer perimeter of the grid will always be visible.

In addition to drawing the maze, you will be able to move a spherical avatar through the maze. You can start by making the avatar a block that is the same size as the maze blocks, or a bit smaller. But eventually you will need to create a sphere shaped avatar for full marks. Regardless, the avatar should be a different colour than colours of the maze blocks (although the avatar may be of a single colour). You are free to model an avatar more complex than a sphere if you want. Your initial implementation could start by reusing the cube, but you need to algorithmically generate the triangles of a sphere for full credit.

For the most part, you will use the keyboard to control the modelling environment, as explained below. In addition, you can manipulate the view in a few simple ways via the mouse: you can rotate the grid around its centre to look at the design from all angles, and scale it up or down.

User interaction

The initial state should not display any maze, and should show the avatar at (0,0).

Your implementation of Maze should support (at least) the following commands and interactions:

- **Quit Application:** There should be a Quit Application button in the control panel that terminates. It should also be possible to perform this action using the **Q** key.
- **Reset:** There should be a Reset button that restores the grid to its initial, empty state, resets the view to the default, resets the colour to the initial colours, and moves the avatar back to the cell (0,0). (Hotkey: **R**)
- **Dig:** There should be a Dig button that will create the maze (or create a new maze if one already exists), and place the avatar at the start cell of the maze. (Hotkey: **D**)
- **Arrow keys:** The arrow keys move the avatar block around the maze. Use the **left**, **right**, **up/forward** and **down/back** arrows to move through the maze. The avatar should not move into a cell that has a wall of the maze in it.
- Holding down the **shift** key with the arrows should let the avatar remove the wall section in the direction of the arrow key (if one exits) and move into that section. The avatar should move in the direction of the arrow key regardless of whether or not a wall existed there before the movement, but regardless, after the movement the wall section should be gone.
- The avatar should not move off the ground grid at any time.
- **Colours:** There is a colour editor in the user interface. It should be extended to have three radio buttons, one each for the colour of the maze blocks, the colour of the floor, and the colour of the avatar. Clicking the radio button selects the object whose colour you are editing; it should also set the colour editor's colour to the current colour of the selected object. Editing the colour (e.g., changing the R, G and B sliders) should immediately affect *all* the appropriate cube faces. The radio buttons should be labelled to succinctly describe what colour is being manipulated.
- **Rotation:** Dragging the mouse to the left or the right should rotate the grid smoothly around its centre (as if it were sitting on a turntable). This feature lets the user examine the maze from all angles. Moving the mouse should produce a "reasonable" amount of rotation, proportional to the distance travelled by the mouse in the *x* direction.
- **Persistence:** You are also required to implement a feature sometimes known as "persistence" or "gravity". If, while rotating, the mouse is moving at the time that the button is released, the rotation should continue on its own. This decision should be made at the time of release; after that, it should persist independently of mouse movement, until the next button press. You may be able to use `A1::appLogic()` to handle this; alternatively, you could use a timer.
- **Scaling:** The mouse's scroll wheel (or the scroll gesture on a trackpad) should control the scaling of the grid. Scrolling down should make the grid smaller, scrolling up should make it bigger. The amount of scaling should be constrained by reasonably chosen maximum and minimum amounts.
- **Growing bars:** The **space** key should grow the walls by one unit. The **backspace** key should shrink the walls by one unit. It should not be possible to shrink a bar below the ground plane. The environment may impose an upper limit on bar height, but this limit is not required.

Skeleton code

If you haven't already, follow the instructions in Assignment 0 for downloading the framework and skeleton code. You'll find the skeleton code for this assignment in the `A1/` directory. It should compile and run out of the box using the same premake4/make commands used for Assignment 0. You'll see the the skeleton program sets up the control panel, including a Quit button and single example of a colour slider and radio button. It also draws a ground grid for you. Notice that the grid is of size 18×18. The live area of the grid is 16×16, as defined by the constant `DIM` in `A1.cpp`; the program automatically adds the extra ring of cells around the maze (i.e., the ground grid extends one beyond the walls of the maze). The program defines reasonable initial values for the projection, view and model matrices, and passes them into the vertex shader for you.
`maze.cpp` contains a class for generating a maze.

Here is a suggested sequence of steps to get you started with this assignment. (But feel free to work on the assignment in any order you want!)

1. First and foremost, get a cube on the screen. This is probably the hardest part of the assignment! Define the geometry of a unit cube, and get it into OpenGL by creating appropriate attribute arrays and vertex buffers. You *must* use modern OpenGL commands here and throughout the assignment. That is, eventually you'll draw cubes using a function like `glDrawArrays()` or `glDrawElements()`, and you should never call functions like `glVertex()`.
2. See if you can create walls that are more than one block high.
3. If you can do that, hook up the **space** and **backspace** keys to groaw and shrink the walls.
4. Now define and locate the avatar, and add in the functionality for moving the avatar with the arrow keys.
5. Add in the colour editor.
6. Add in mouse rotation. This is most easily accomplished by modifying the model-to-world transformation while holding the camera fixed. As is typical in interactive programs, it'll help to keep track of the *previous x* position of the mouse, so that you can rotate by an amount proportional to the *difference* in *x* every time you receive a mouse event. (Note that after rotating, the arrow keys might become quite unintuitive, since the grid is rotated relative to the directions of the keys. You don't have to worry about that for this assignment.)
7. Hook up a simple timer that calls down to the maze's render method and re-renders. You should now be able to implement persistence.
8. Add scaling by responding to scroll events. There are probably several ways to do this. One would be to scale the world itself by manipulating the model-to-world transformation. Another is to move the camera closer to or farther from the centre of the grid. Either technique is fine, as long as the scaling is bounded within reasonable limits.
9. Finally, add in the Reset button and its hotkey, setting all the values manipulated above back to the defaults.

Deliverables

The submission process for this assignment is basically the same as the one for Assignment 0. Prepare a ZIP file of the `A1/` directory, omitting unnecessary files (executables, build files, etc.). Upload the ZIP file to LEARN.

As with Assignment 0, you *must* submit two other files in your `A1/` directory: the `README` and a screenshot. If you omit either of the first two files, you will receive a deduction. See Assignment 0 for instructions about how to prepare these files.

Other thoughts

As OpenGL has evolved, the API has actually gotten a bit more streamlined. The part of OpenGL you talk to from inside a C++ application tends to focus on telling the GPU about the pieces of data that power your program, and maybe setting a couple of system-wide parameters; most of the actual "graphics" has been pushed into the shaders.

If it helps, here is the complete set of OpenGL API calls we used in the model solution. *This is for information only:* you should feel free to use a subset of these functions, and to include others not on the list. But you should *not* use deprecated OpenGL calls from the compatibility profile:
no `glBegin()`, `glVertex()`, or `glEnd()`!

- `glBindBuffer()`
- `glBindVertexArray()`
- `glBufferData()`
- `glClearColor()`
- `glDisable()`
- `glDrawArrays()`
- `glDrawElements()`
- `glEnable()`
- `glEnableVertexAttribArray()`
- `glGenBuffers()`
- `glGenVertexArrays()`
- `glUniform3f()`
- `glUniformMatrix4fv()`
- `glVertexAttribPointer()`

Ideas for extensions

The Maze environment is a great starting point for expansion in new directions. We'd be excited to see what additional features you can add to the base interface, if you've got the time. We *may* award a bonus point or two for especially ambitious extensions. Here are some random ideas for extensions:

- The look of Maze is pretty bare-bones. In fact, given that bars are flat, solid colours, the 3D geometry of the world can be pretty hard to understand. Try to add some kind of simple shading model to your cubes, possibly including lights. The simplest model is Lambertian shading based on an implied directional light source (though you'll discover that flat shaded meshes are surprisingly

annoying to set up in modern OpenGL). If you want to get really fancy, look around for Screen-Space Ambient Occlusion algorithms.
- Give a stone or brick look by applying textures to your cubes.
- Smoothly animate the motion of the avatar as it moves through the maze rather than jumping from cell to cell.
- Make the avatar more interesting than a single cube. If nothing else, making it smaller than the cubes that form the walls of the maze will give it more of an appearance of being in the maze.
- Load a 3D mesh model to use for the avatar. Animating its motion would be extra.
- Implement a way to load and save the contents of the grid, and maybe to load a grid from some kind of image.
- Add a WASD/mouselook interface so that you can walk (or fly) into the maze and look around as if it were a city grid. For extra fanciness, implement rudimentary collision detection so that you can't walk through cubes.

If you do extend the base interface, be sure to document your extensions in your `README` file. Keep in mind that you must still satisfy the core objectives listed here. If your changes are so radical that your modified program is incompatible with the original specification, you must include a "compatibility mode" that makes the interface behave like the requirements here. (Or consider creating an entirely separate executable.)

Step 5: Objective list

Every assignment includes a list of objectives. Your mark in the assignment will be based primarily on the achievement of these objectives, with possible deductions outside the list if necessary.

Assignment 1 objectives

- ☐ 1. The program is able to draw a 3D grid of cubes to represent the maze.

- ☐ 2. The **space** and **backspace** keys grow and shrink the height of the walls as described above.
- ☐ 3. There is a spherical avatar that can be moved using the arrow keys. The avatar's movement is blocked by walls.
- ☐ 4. The avatar can be moved using the shift arrow keys. If a wall exists in the cell into which the avatar is trying to move with the shifted arrow key, then that wall section is removed.
- ☐ 5. Three different colours are used to colour the blocks and the floor as described above.
- ☐ 6. The colour editor can be used to change the colour of the maze blocks, the maze floor, and the avatar.
- ☐ 7. Rotation works as described above.
- ☐ 8. Persistence works.
- ☐ 9. Scaling works as described above.

- ☐ 10. The reset button returns the user interface and grid stat to the starting state.