TNM061 3D Computer Graphics
OpenGL hierarchical transformations
Lab instructions 2024

Martin Falk

January 12, 2024

Contents

1 Introduction

This lab exercise is concerned with hierarchical transformations which are used to animate a model of a solar system in C++. OpenGL (opengl.org) is used for rendering the small scene with multiple textured objects. Topics covered in this lab:

- Using CMake for setting up and compiling an OpenGL project
- Scene graphs and hierarchical transformations
- Transformations using a matrix stack
- Rendering with OpenGL in C++

2 Theory

1. Discuss briefly the differences between real-time graphics and offline rendering.
2. You want to model a small solar system with a central star and a planet traveling around the star. The star and the planet should rotate around their respective axes.
   (a) Draw the scene graph with the scene as the root node and the necessary transformations (translations and rotations).
   (b) Why does the order of the translation and rotation matter here?
   (c) Add a moon orbiting around the planet. The moon should rotate around its axis as well.
   (d) (Bonus) What changes are necessary so that the same side of the moon always faces the planet?
   Think of Earth's Moon, we always see the front side never the backside.

3 Assignments

You will be using OpenGL, a graphics programming API, in combination with C++ to solve these assignments. The general structure of the code is based on the labs from TNM046. The source code provides you with a working program using OpenGL. It provides functionality to manipulate the camera, handle triangle meshes, textures, and GLSL shaders. In addition, there is also a matrix stack which allows you to render a simple hierarchical animated scene.

You will need:

- CMake (cmake.org) for generating project files.
- An IDE for editing and compiling C++ code. We recommend using Visual Studio Community Edition, Visual Studio Code, or XCode. You may need to install a C++ toolchain for XCode separately.

In order to get started, unzip the lab files and start CMake. Set the *Source path* to the folder you just unzipped, then set the *Build path* to a different folder (preferably not inside `lab1-cpp`). Run *configure* and select the compiler of your choice matching the IDE you want to use. If successful, you can *generate* the project files for your IDE and then open them directly.

When you want to edit your code at a later point, you can load the project files directly in your IDE. They are located in the build folder. There is no need to start CMake and running configure/generate again.

3.1 Tasks

The main task is to create a scene similar to Figure 1: a solar system with the Sun, Earth, and the Moon. This task is divided into several steps outlined below. All objects should be textured and the matrix stack should keep track of all transformations. The vertex shader should only use the current matrix of the matrix stack and the projection matrix. Push and pop operations in the matrix stack should match each other. Otherwise, the stack will grow each time you render a frame and at some point run out of memory.

1. Familiarize yourself with the source code given in the lab material (see Appendix 4.1).
2. Identify the transformations in `glstack.cpp` and translate them back into a hierarchical transformation graph (hint: it should look similar to the one in the theory section).
3. Add a textured Moon to Earth by using the matrix stack (see Appendix 4.3). Try to minimize the use of `push();` and `pop();`.
4. Add two more textured planets orbiting the Sun.

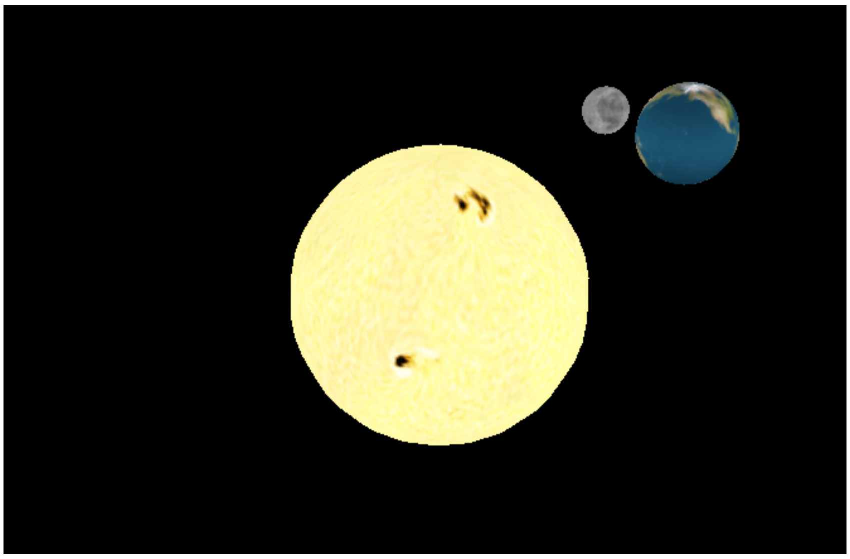
Figure 1: A planetary system with three celestial objects.

4 Appendix

4.1 Source Files

The source code is distributed over a number of files. The main file being `glstack.cpp` while utility functions and classes are located in the `util/` folder. Fragment and vertex shader files are stored under `shaders/`. The intended use of all the classes is demonstrated by the example program `glstack.cpp`.

`glstack.cpp` The main file where you add your changes.

`Shader.cpp/.hpp` Class for managing GLSL shader programs, supports loading and compiling shaders from files.

`Texture.cpp/.hpp` Provides functionality for loading TGA files from disk and using them as OpenGL textures. Only uncompressed TGA files with 24 bit RGB or 32 bit RGBA are supported.

`TriangleSoup.cpp/.hpp` An abstraction layer for triangle meshes that manages OpenGL vertex buffers and vertex array objects (VAO).

`Rotator.cpp` Contains two classes: `MouseRotator` and `KeyRotator`. In the example code, a `MouseRotator` object is used to manipulate the view with the mouse.

`MatrixStack.cpp/.hpp` Implementation of a matrix stack that can be used in OpenGL. See next section for details.

`tnm061.cpp/.hpp` utility functions for printing error messages and displaying the frame rate in the window title

4.2 Hierarchical Transformations

Our focus here is going to be transformations, and specifically hierarchical transformations. Many scene descriptions are based on some kind of linking between objects, such that transformations for one "parent" object are inherited by a number of "child" objects, often in several levels. This can be described by a *scene graph*, a concept that we will explore further in the next lab session. In raw and unassisted OpenGL, there is no notion of a scene graph - it's all just polygons, matrices, shaders and textures. Before OpenGL 3 arrived, though, OpenGL had a built-in *matrix stack*. That functionality did not really belong in OpenGL, so it has been deprecated and is not recommended for use even on platforms where it's still available. However, a matrix stack is still a useful tool for managing hierarchical transformations, even if it's no longer a part of OpenGL. Therefore, we will introduce a matrix stack implemented in software, and ask you to use it to render a scene which has the fundamental properties of a scene graph.

4.3 Matrix Stack

A *stack* is a popular form of data structure that is used for many different purposes. It is the software equivalent of a physical stack of things, where you always place new items on top of the stack and always remove the topmost item first. The most popular analogy is a stack of plates in a cafeteria, placed in the kind of spring-loaded container that makes the top of the stack always be at a constant height. Putting one item on the stack is often referred to as "pushing" it onto the stack, and removing one item is referred to as "popping" it from the stack. Another less popular name for a stack is a "last-in-first-out" (LIFO) storage. See Figure 2.

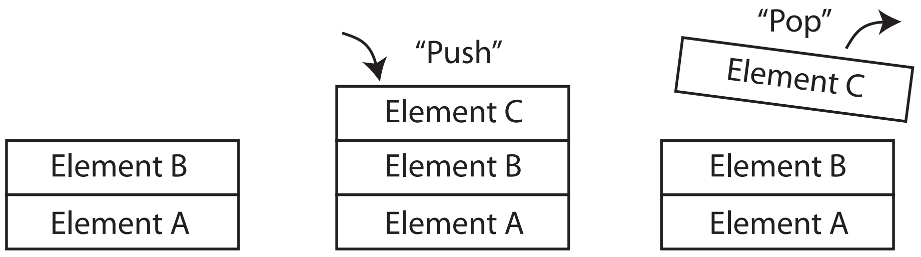

Figure 2: Pushing and popping items on a stack.

A stack is a useful tool when traversing a tree-like structure, like a scene graph. You can keep track of the current transformation for a node in the tree by simply remembering two things: when you go down a link in the tree, you push the corresponding matrix on the stack, and when you go back up a link, you pop one matrix from the stack. As a convenience for the particular use of a matrix stack, you can also multiply the matrix you add at the top with the previous matrix on the stack. This makes the topmost matrix on the stack describe the entire chain of composite transformations from the root of the tree all the way to the current node.

In a matrix stack designed for traversing a scene graph, it is convenient to define three different operations:

- Manipulate the matrix on the top of the stack, the "current matrix", by multiplying it with other matrices. This is convenient for creating arbitrary transformations from simple fundamental operations like translation, rotation and scaling.
- Create a copy of the current matrix and place it on top of the stack. This means that a "Push" operation does not require you to specify what you push – you always push a copy of the topmost matrix.
- Remove a matrix from the stack to make the previous matrix the current matrix.

With the structure described above, the operations "push" and "pop" can also be thought of as "save" and "restore", a mechanism for "undoing" an arbitrary number of modifications to the current matrix that have been marked by a "save point". Several matrices may be stored on the stack by repeated use of "push". For each "pop", one matrix is removed from the stack and the current matrix is reset to what it was just before the most recent "push".

An illustration of this particular variant of a stack is in Figure 3, where you see a sequence of operations on the stack and the corresponding contents of the stack before and after each operation. Note that the topmost (current) matrix can be changed by several multiplications after each "push".

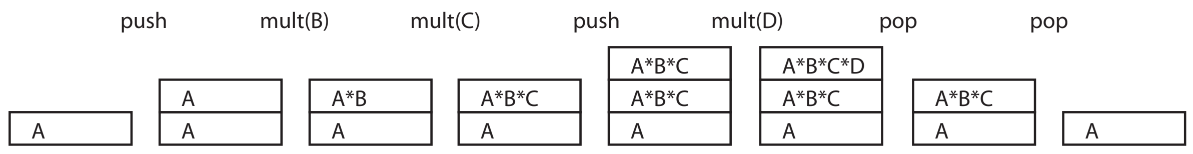

Figure 3: Using a matrix stack.

Read the source code in `MatrixStack.hpp` and `MatrixStack.cpp`, and make sure you understand what the code does. There may be some details that are not immediately obvious, but try to read past that and understand how the code does what it is supposed to do. A `MatrixStack` is basically just a dynamic linked list of nodes of type `Matrix`, with each node containing a 4x4 matrix of `float` values. The class `MatrixStack` and its methods are described below. The same information is the header file `MatrixStack.hpp`, albeit in a slightly more terse form.

`class MatrixStack`

`Matrix *currentMatrix;`
Pointer to the top element on the stack

`float* getCurrentMatrix();`
Get the pointer to the topmost (current) matrix array. This is used for sending the matrix to the shader. The matrix is not meant to be manipulated directly. Use the transformations below for that.

`void init();`
Clear the stack and set the topmost (current) matrix to the identity matrix. This is a potentially dangerous operation, because it completely destroys the hierarchy of the stack. This function should only be used right after the creation of a matrix stack, to initialize its contents. (The constructor calls this, which is really the only time it's needed.)

`void rotX(float angle);`
Multiply the topmost (current) matrix with a rotation around X. The angle is specified in radians.

`void rotY(float angle);`
Multiply the topmost (current) matrix with a rotation around Y. The angle is specified in radians.

`void rotZ(float angle);`
Multiply the topmost (current) matrix with a rotation around Z. The angle is specified in radians.

`void scale(float s);`
Multiply the topmost (current) matrix with a uniform scaling.

`void translate(float x, float y, float z);`
Multiply the topmost (current) matrix with a translation.

`void push();` Add a new level on the stack, making a copy of the current matrix and placing it on top.

`void pop();`
Remove one element from the stack, exposing the element below.

`void flush();`
Remove all elements except the last one from the stack. Under normal circumstances, this is an operation that *should not be used*. You should keep the stack clean by making sure to remove all elements you add to it before the rendering loop ends. *For every* ***`push()`*** *you do, there should be exactly one corresponding* ***`pop()`****.*

`int depth();`
Count the number of elements on the stack. Intended use are safety checks and debugging purposes. A valid `MatrixStack` object with `currentMatrix = nullptr` has depth 0. A newly created `MatrixStack` only contains an identity matrix, and thus has depth 1.

`void print();`
Print the entire contents of the stack to the console output, for debugging purposes.